

CISP 430: DATA STRUCTURES



Suha Aljuboori
Folsom Lake College

Folsom Lake College
CISP 430: Data Structures

Suha Aljuboori

This text is disseminated via the Open Education Resource (OER) LibreTexts Project (<https://LibreTexts.org>) and like the thousands of other texts available within this powerful platform, it is freely available for reading, printing, and "consuming."

The LibreTexts mission is to bring together students, faculty, and scholars in a collaborative effort to provide an accessible, and comprehensive platform that empowers our community to develop, curate, adapt, and adopt openly licensed resources and technologies; through these efforts we can reduce the financial burden born from traditional educational resource costs, ensuring education is more accessible for students and communities worldwide.

Most, but not all, pages in the library have licenses that may allow individuals to make changes, save, and print this book. Carefully consult the applicable license(s) before pursuing such effects. Instructors can adopt existing LibreTexts texts or Remix them to quickly build course-specific resources to meet the needs of their students. Unlike traditional textbooks, LibreTexts' web based origins allow powerful integration of advanced features and new technologies to support learning.



LibreTexts is the adaptable, user-friendly non-profit open education resource platform that educators trust for creating, customizing, and sharing accessible, interactive textbooks, adaptive homework, and ancillary materials. We collaborate with individuals and organizations to champion open education initiatives, support institutional publishing programs, drive curriculum development projects, and more.

The LibreTexts libraries are Powered by [NICE CXone Expert](#) and was supported by the Department of Education Open Textbook Pilot Project, the California Education Learning Lab, the UC Davis Office of the Provost, the UC Davis Library, the California State University Affordable Learning Solutions Program, and Merlot. This material is based upon work supported by the National Science Foundation under Grant No. 1246120, 1525057, and 1413739.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation nor the US Department of Education.

Have questions or comments? For information about adoptions or adaptations contact info@LibreTexts.org or visit our main website at <https://LibreTexts.org>.

This text was compiled on 05/15/2026

TABLE OF CONTENTS

Licensing

1: Introduction

- 1.1: The Need for Efficiency
- 1.2: Stanley Milgram
- 1.3: Watts and Strogatz

2: Recursion

- 2.1: Activity 1 - Recursive Algorithm
- 2.2: Unit Summary

3: Stacks and Queues

- 3.1: Stacks
- 3.2: Applications of Stacks
- 3.3: Queues
- 3.4: Deques

4: Linked Lists

- 4.1: SLList - A Singly-Linked List
- 4.2: DLList - A Doubly-Linked List
- 4.3: SEList - A Space-Efficient Linked List
- 4.4: Discussion and Exercises

5: Trees

- 5.1: Traversal
- 5.2: Sample implementations for Tree Traversal
- 5.3: Examples of Tree Traversals
- 5.4: Balancing
- 5.5: Binary Search Trees
- 5.6: Example extract of Java code for binary tree delete operation
- 5.7: Red-Black trees
- 5.8: B Trees
- 5.9: Treaps
- 5.10: References

6: Graphs

- 6.1: Representing a Graph by a Matrix
- 6.2: A Graph as a Collection of Lists
- 6.3: Graph Traversal
- 6.4: Discussion and Exercises

7: Hash Tables

- 7.1: Time complexity and common uses of hash tables
- 7.2: Choosing a good hash function
- 7.3: Collision resolution
- 7.4: Table resizing
- 7.5: Ordered retrieval issue
- 7.6: Problems with hash tables
- 7.7: Other hash table algorithms
- 7.8: Implementations

8: Searching, Binary Search Trees and Heaps

- 8.1: Activity 1 - Searching Techniques
- 8.2: Activity 2 - Binary Search Tree
- 8.3: Activity 3 - Heaps
- 8.4: Unit Summary

9: Sorting Algorithms

- 9.1: Activity 1 - Sorting Algorithms by insertion method
- 9.2: Activity 2 - Sorting Algorithms by selection method
- 9.3: Activity 3 - Sorting Algorithms by Merge and Partition methods
- 9.4: Unit Summary

[Index](#)

[Glossary](#)

[Detailed Licensing](#)

Licensing

A detailed breakdown of this resource's licensing can be found in [Back Matter/Detailed Licensing](#).

CHAPTER OVERVIEW

1: Introduction

Every computer science curriculum in the world includes a course on data structures and algorithms. Data structures are *that* important; they improve our quality of life and even save lives on a regular basis. Many multi-million and several multi-billion dollar companies have been built around data structures.

How can this be? If we stop to think about it, we realize that we interact with data structures constantly.

- Open a file: File system data structures are used to locate the parts of that file on disk so they can be retrieved. This isn't easy; disks contain hundreds of millions of blocks. The contents of your file could be stored on any one of them.
- Look up a contact on your phone: A data structure is used to look up a phone number in your contact list based on partial information even before you finish dialing/typing. This isn't easy; your phone may contain information about a lot of people--everyone you have ever contacted via phone or email--and your phone doesn't have a very fast processor or a lot of memory.
- Log in to your favourite social network: The network servers use your login information to look up your account information. This isn't easy; the most popular social networks have hundreds of millions of active users.
- Do a web search: The search engine uses data structures to find the web pages containing your search terms. This isn't easy; there are over 8.5 billion web pages on the Internet and each page contains a lot of potential search terms.
- Phone emergency services (9-1-1): The emergency services network looks up your phone number in a data structure that maps phone numbers to addresses so that police cars, ambulances, or fire trucks can be sent there without delay. This is important; the person making the call may not be able to provide the exact address they are calling from and a delay can mean the difference between life or death.

[1.1: The Need for Efficiency](#)

[1.2: Stanley Milgram](#)

[1.3: Watts and Strogatz](#)

1: Introduction is shared under a [CC BY](#) license and was authored, remixed, and/or curated by LibreTexts.

1.1: The Need for Efficiency

In the next section, we look at the operations supported by the most commonly used data structures. Anyone with a bit of programming experience will see that these operations are not hard to implement correctly. We can store the data in an array or a linked list and each operation can be implemented by iterating over all the elements of the array or list and possibly adding or removing an element.

This kind of implementation is easy, but not very efficient. Does this really matter? Computers are becoming faster and faster. Maybe the obvious implementation is good enough. Let's do some rough calculations to find out.

Number of operations:

Imagine an application with a moderately-sized data set, say of one million (10^6), items. It is reasonable, in most applications, to assume that the application will want to look up each item at least once. This means we can expect to do at least one million (10^6) searches in this data. If each of these 10^6 searches inspects each of the 10^6 items, this gives a total of $10^6 \times 10^6 = 10^{12}$ (one thousand billion) inspections.

Processor speeds:

At the time of writing, even a very fast desktop computer can not do more than one billion (10^9) operations per second.¹ This means that this application will take at least $10^{12}/10^9 = 1000$ seconds, or roughly 16 minutes and 40 seconds. Sixteen minutes is an eon in computer time, but a person might be willing to put up with it (if he or she were headed out for a coffee break).

Bigger data sets:

Now consider a company like Google, that indexes over 8.5 billion web pages. By our calculations, doing any kind of query over this data would take at least 8.5 seconds. We already know that this isn't the case; web searches complete in much less than 8.5 seconds, and they do much more complicated queries than just asking if a particular page is in their list of indexed pages. At the time of writing, Google receives approximately 4,500 queries per second, meaning that they would require at least $4,500 \times 8.5 = 38,250$ very fast servers just to keep up.

The solution:

These examples tell us that the obvious implementations of data structures do not scale well when the number of items, n , in the data structure and the number of operations, m , performed on the data structure are both large. In these cases, the time (measured in, say, machine instructions) is roughly $n \times m$.

The solution, of course, is to carefully organize data within the data structure so that not every operation requires every data item to be inspected. Although it sounds impossible at first, we will see data structures where a search requires looking at only two items on average, independent of the number of items stored in the data structure. In our billion instruction per second computer it takes only 0.00000002 seconds to search in a data structure containing a billion items (or a trillion, or a quadrillion, or even a quintillion items).

We will also see implementations of data structures that keep the items in sorted order, where the number of items inspected during an operation grows very slowly as a function of the number of items in the data structure. For example, we can maintain a sorted set of one billion items while inspecting at most 60 items during any operation. In our billion instruction per second computer, these operations take 0.00000006 seconds each.

Footnotes

¹Computer speeds are at most a few gigahertz (billions of cycles per second), and each operation typically takes a few cycles.

1.2: Stanley Milgram

Stanley Milgram was an American social psychologist who conducted two of the most famous experiments in social science, the Milgram experiment, which studied people's obedience to authority (thinkcomplex.com/milgram) and the Small World Experiment, which studied the structure of social networks (thinkcomplex.com/small).

In the Small World Experiment, Milgram sent a package to several randomly-chosen people in Wichita, Kansas, with instructions asking them to forward an enclosed letter to a target person, identified by name and occupation, in Sharon, Massachusetts (which happens to be the town near Boston where I grew up). The subjects were told that they could mail the letter directly to the target person only if they knew him personally; otherwise they were instructed to send it, and the same instructions, to a relative or friend they thought would be more likely to know the target person.

Many of the letters were never delivered, but for the ones that were the average path length — the number of times the letters were forwarded — was about six. This result was taken to confirm previous observations (and speculations) that the typical distance between any two people in a social network is about “six degrees of separation”.

This conclusion is surprising because most people expect social networks to be localized — people tend to live near their friends — and in a graph with local connections, path lengths tend to increase in proportion to geographical distance. For example, most of my friends live nearby, so I would guess that the average distance between nodes in a social network is about 50 miles. Wichita is about 1600 miles from Boston, so if Milgram's letters traversed typical links in the social network, they should have taken 32 hops, not 6.

This page titled [1.2: Stanley Milgram](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)) .

- [3.1: Stanley Milgram](#) by [Allen B. Downey](#) is licensed [CC BY-NC-SA 3.0](#).

1.3: Watts and Strogatz

In 1998 Duncan Watts and Steven Strogatz published a paper in *Nature*, “Collective dynamics of ‘small-world’ networks”, that proposed an explanation for the small world phenomenon. You can download it from <http://thinkcomplex.com/watts>.

Watts and Strogatz start with two kinds of graph that were well understood: random graphs and regular graphs. In a random graph, nodes are connected at random. In a regular graph, every node has the same number of neighbors. They consider two properties of these graphs, clustering and path length:

- Clustering is a measure of the “cliquishness” of the graph. In a graph, a **clique** is a subset of nodes that are all connected to each other; in a social network, a clique is a set of people who are all friends with each other. Watts and Strogatz defined a clustering coefficient that quantifies the likelihood that two nodes that are connected to the same node are also connected to each other.
- Path length is a measure of the average distance between two nodes, which corresponds to the degrees of separation in a social network.

Watts and Strogatz show that regular graphs have high clustering and high path lengths, whereas random graphs with the same size usually have low clustering and low path lengths. So neither of these is a good model of social networks, which combine high clustering with short path lengths.

Their goal was to create a **generative model** of a social network. A generative model tries to explain a phenomenon by modeling the process that builds or leads to the phenomenon. Watts and Strogatz proposed this process for building small-world graphs:

1. Start with a regular graph with n nodes and each node connected to k neighbors.
2. Choose a subset of the edges and “rewire” them by replacing them with random edges.

The probability that an edge is rewired is a parameter, p , that controls how random the graph is. With $p=0$, the graph is regular; with $p=1$ it is completely random.

Watts and Strogatz found that small values of p yield graphs with high clustering, like a regular graph, and low path lengths, like a random graph.

This page titled [1.3: Watts and Strogatz](#) is shared under a [not declared](#) license and was authored, remixed, and/or curated by [Allen B. Downey](#) ([Green Tea Press](#)) .

CHAPTER OVERVIEW

2: Recursion

Unit Introduction

This unit introduces the learner to the processes that repeat themselves in a self-similar way. The processes can either call themselves directly or indirectly during their execution. The unit will enable the learner to understand the recursion process and solve problems that are recursive in nature.

Unit Objectives

Upon completion of this unit the learner should be able to:

- develop algorithms for recursive programs
- implement recursive formulation of a problem
- explain recursion as a form of iteration

Key Terms

- **Recursion:** This is a process whereby a method calls itself so as to achieve repetitious
- **Base case:** Is a boolean test at which the recursion ceases to “wind up” and starts to “unwind”.
- **Recursive step:** Is a method call which causes the recursion to “repeat” on each successive iteration (unless the base case is reached).

Learning Activities

[2.1: Activity 1 - Recursive Algorithm](#)

[2.2: Unit Summary](#)

This page titled [2: Recursion](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Harrison Njoroge](#) (African Virtual University) .

2.1: Activity 1 - Recursive Algorithm

Introduction

The learners are introduced to recursive processes and learn how a method calls itself in order to achieve repetitious behaviour. By using recursive algorithms, the learners will be in a position to solve problems of this nature.

Recursive Algorithm

Recursion is defined as a method of solving problems that involves breaking a problem down into smaller and smaller sub problems until you get to a small enough problem that it can be solved trivially. Simply this is done by the program calling itself to repeat the same steps.

Recursive problems are solved by developing a recursive algorithm. A recursive algorithm is defined as an algorithm which can call itself with smaller (or simpler) input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. Thus if a problem can be solved by utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then it means that one can use a recursive algorithm to solve that problem.

The recursive method comprises the following:

1. A base case where the recursion stops and begins to unwind, and
2. A recursive step where the method calls itself

A point worth noting is that a recursive algorithm is said to wind up until the base case is reached and then unwinds. During the unwinding process, further statement programs “pending” may be called.

Recursion is a mechanism whereby repetition can be achieved using a method that continues to call itself until some terminating value is reached, i.e. without using some specific repetition construct.

✓ Example 2.1.1

The factorial function “!”.

Its definition is:

- $0! = 1$
- for all $n > 0$, $n! = n \times (n - 1)!$

Thus, by repeatedly using the definition, we can work out that

$$\begin{aligned} 6! &= 6 \times 5! \\ &= 6 \times 5 \times 4! \\ &= 6 \times 5 \times 4 \times 3! \\ &= 6 \times 5 \times 4 \times 3 \times 2! \\ &= 6 \times 5 \times 4 \times 3 \times 2 \times 1! \\ &= 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 1 \\ &= 720 \end{aligned}$$

Again, notice that the definition of “!” includes both a base case (the definition of $0!$) and a recursive part.

Laws of Recursion

Recursive algorithms must obey the following laws:

1. A recursive algorithm must have a base case.
2. A recursive algorithm must change its state and move toward the base case.

3. A recursive algorithm must call itself, recursively.

An explanation of the laws is as follows:

First law: the base case allows the algorithm to stop recursing, i.e. a problem small enough to solve directly.

Second law: a change of state occurs moving the algorithm towards the base case. This occurs when a change of state takes place and which is best illustrated when a data modification happens.

Finally, third law: the algorithm must call itself.

✓ **Example 2.1.2**

Find the maximum value in a list of numbers.

The solution to the problem would look like the following:

```
Function find_max(list)
possible_max_1 = first value in list
possible_max_2 = find_max(rest of the list);
if(possible_max_1 > possible_max_2)
    answer is possible_max_1
else
    answer is possible_max_2
end
end
```

✓ **Example 2.1.3**

Adding three numbers.

The solution will be something like

```
function result = add_numbers(a, b, c)
if (nargin() == 2)
    result = a + b;
else if (nargin() == 3)
    result = add_numbers(a+b, c);
else
    error('oops');
end
end
```

Conclusion

This unit introduced the learner to the recursive algorithms. Examples were used to teach the learner how to write recursive algorithms that can solve self repeating functions. The examples used in the teaching of recursion are also not hinged on any computing language and can thus apply across all the languages. Finally the learners were introduced to the laws of recursion.

? **Assessment**

1. List the three key features of a recursive algorithm.
2. Write an algorithm that can multiply two numbers using recursion.

This page titled [2.1: Activity 1 - Recursive Algorithm](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Harrison Njoroge \(African Virtual University\)](#) .

- [2.1: Activity 1 - Recursive Algorithm](#) by Harrison Njoroge is licensed [CC BY-SA 4.0](#).

2.2: Unit Summary

In this unit, recursion was introduced. You should now be in a position to describe recursion and where it can be applied. The laws that a recursing algorithm must obey were also introduced as well as condition necessary for a process to be said to have recursed. Examples accompanied these explanations were provided where necessary.

? Unit Assessment

Check your understanding!

Class Exercise

Instructions

Answer the following questions on recursion.

1. Write an algorithm for finding the k-th even natural number
2. Explain how a recursive algorithm works

Feedback

1. The solution

Algorithm:

```
if k = 1, then return 0;  
else return Even(k - 1) + 2
```

2. The result of one recursion is the input for the next recursion. The repetition is in the self-similar fashion. The algorithm calls itself with smaller input values and obtains the results by simply performing the operations on these smaller values. Generation of factorial, Fibonacci number series are the examples of recursive algorithms

Grading Scheme

The marks will be awarded as shown below:

Question	Scores (marks)
1	2
2	Each line of explanation 2 marks; maximum 8
Total	10

Unit Readings and Other Resources

The readings in this unit are to be found at [course level readings and other resources](#).

This page titled [2.2: Unit Summary](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Harrison Njoroge](#) (African Virtual University).

- [2.2: Unit Summary](#) by Harrison Njoroge is licensed [CC BY-SA 4.0](#).

CHAPTER OVERVIEW

3: Stacks and Queues

[3.1: Stacks](#)

[3.2: Applications of Stacks](#)

[3.3: Queues](#)

[3.4: Deques](#)

3: Stacks and Queues is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

3.1: Stacks

A stack is a basic data structure that can be logically thought of as a linear structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack. The basic concept can be illustrated by thinking of your data set as a stack of plates or books where you can only take the top item of the stack in order to remove things from it. This structure is used all throughout programming.

The basic implementation of a stack is also called a LIFO (Last In First Out) to demonstrate the way it accesses data, since as we will see there are various variations of stack implementations.

There are basically three operations that can be performed on stacks. They are 1) inserting an item into a stack (push). 2) deleting an item from the stack (pop). 3) displaying the contents of the stack (peek or top).

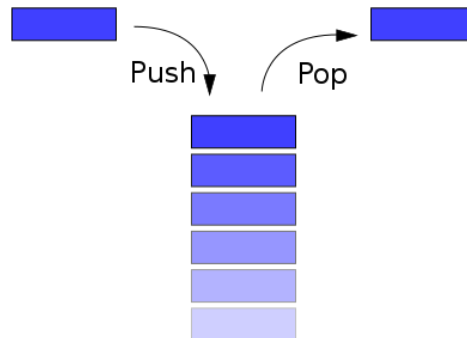


Figure 3.1.1 (Public Domain; by Boivie via Wikimedia Commons)

Note:

Depending on the language and implementation the data structure may share the name with an abstract data type that support all of the data structure characteristics.

Below are some of operations a **stack data type** normally supports:

Stack<item-type> Operations

push(new-item:item-type)

Adds an item onto the stack.

top():item-type

Returns the last item pushed onto the stack.

pop()

Removes the most-recently-pushed item from the stack.

is-empty():Boolean

True if no more items can be popped and there is no top item.

is-full():Boolean

True if no more items can be pushed.

get-size():Integer

Returns the number of elements on the stack.

All operations except **get-size()** can be performed in $O(1)$ time. **get-size()** runs in at worst $O(N)$.

Linked List Implementation

The basic linked list implementation is one of the easiest stack implementations you can do. Structurally it is a linked list.

```
type Stack<item_type>
  data list:Singly Linked List<item_type>
  "stack follows the LIFO (last in first out) operation"
  "queue follows the FIFO (first in first out) operation"
  constructor()
    list := new Singly-Linked-List()
  end constructor
```

Most operations are implemented by passing them through to the underlying linked list. When you want to **push** something onto the list, you simply add it to the front of the linked list. The previous top is then "next" from the item being added and the list's front pointer points to the new item.

```
method push(new_item:item_type)
  list.prepend(new_item)
end method
```

To look at the **top** item, you just examine the first item in the linked list.

```
method top():item_type
  return list.get-begin().get-value()
end method
```

When you want to **pop** something off the list, simply remove the first item from the linked list.

```
method pop()
  list.remove-first()
end method
```

A check for emptiness is easy. Just check if the list is empty.

```
method is-empty():Boolean
  return list.is-empty()
end method
```

A check for full is simple. Linked lists are considered to be limitless in size.

```
method is-full():Boolean
  return False
end method
```

A check for the size is again passed through to the list.

```
method get-size():Integer
  return list.get-size()
```

```
end method  
end type
```

A real Stack implementation in a published library would probably re-implement the linked list in order to squeeze the last bit of performance out of the implementation by leaving out unneeded functionality. The above implementation gives you the ideas involved, and any optimization you need can be accomplished by inlining the linked list code.

Performance Analysis

In a linked list, accessing the first element is an $O(1)$ operation. The list contains a pointer that checks for empty/fullness as done here are also $O(1)$ (depending on what time/space tradeoff is made). Most of the time, users of a Stack do not use the `getSize()` operation, and so a bit of space can be saved by not optimizing it.

Since all operations are at the top of the stack, the array implementation is now much, much better.

```
public class StackArray implements Stack  
{  
    protected int top;  
    protected Object[] data;  
    ...  
}
```

The array implementation keeps the bottom of the stack at the beginning of the array. It grows toward the end of the array. The only problem is if you attempt to push an element when the array is full. If so

```
Assert.pre(!isFull(), "Stack is not full.");
```

will fail, raising an exception. Thus it makes more sense to implement with Vector (see StackVector) to allow unbounded growth (at cost of occasional $O(n)$ delays).

Complexity:

All operations are $O(1)$ with exception of occasional push and clear, which should replace all entries by null in order to let them be garbage-collected. Array implementation does not replace null entries. The Vector implementation does...

3.1: Stacks is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- 5.1: Stacks by Wikibooks - Data Structures is licensed [CC BY-SA 4.0](#).

3.2: Applications of Stacks



Figure 3.2.1: Stack of books

Using stacks, we can solve many applications, some of which are listed below.

Converting a decimal number into a binary number

The logic for transforming a decimal number into a binary number is as follows:

```
* Read a number
* Iteration (while number is greater than zero)
    1. Find out the remainder after dividing the number by 2
    2. Print the remainder
    3. Divide the number by 2
* End the iteration
```

However, there is a problem with this logic. Suppose the number whose binary form we want to find is 23. Using this logic, we get the result as 11101, instead of getting 10111.

To solve this problem, we use a stack. We make use of the *LIFO* property of the stack. Initially we *push* the binary digit formed into the stack, instead of printing it directly. After the entire digit has been converted into the binary form, we *pop* one digit at a time from the stack and print it. Therefore we get the decimal number is converted into its proper binary form.

Algorithm:

```
1. Create a stack
2. Enter a decimal number which has to be converted into its equivalent binary form
3. iteration1 (while number > 0)
    3.1 digit = number % 2
    3.2 Push digit into the stack
    3.3 If the stack is full
        3.3.1 Print an error
        3.3.2 Stop the algorithm
    3.4 End the if condition
    3.5 Divide the number by 2
4. End iteration1

5. iteration2 (while stack is not empty)
    5.1 Pop digit from the stack
    5.2 Print the digit
6. End iteration2
7. STOP
```

Towers of Hanoi



Figure 3.2.2: Towers of Hanoi

One of the most interesting applications of stacks can be found in solving a puzzle called Tower of Hanoi. According to an old Brahmin story, the existence of the universe is calculated in terms of the time taken by a number of monks, who are working all the time, to move 64 disks from one pole to another. But there are some rules about how this should be done, which are:

1. You can move only one disk at a time.
2. For temporary storage, a third pole may be used.
3. You cannot place a disk of larger diameter on a disk of smaller diameter.^[1]

Here we assume that A is first tower, B is second tower & C is third tower.

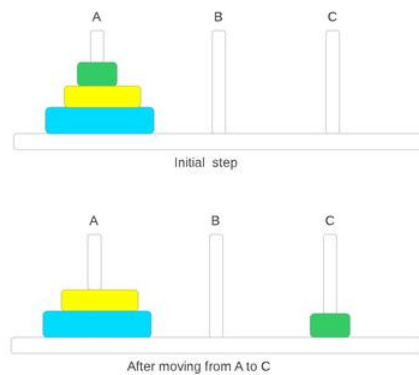


Figure 3.2.3: Towers of Hanoi step 1

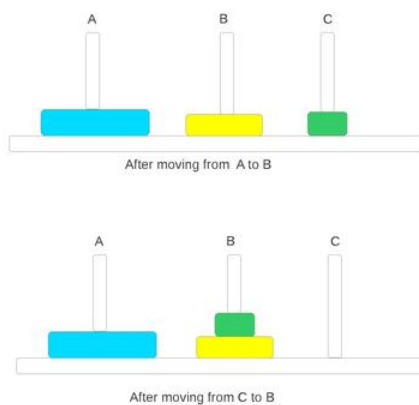


Figure 3.2.4: Towers of Hanoi step 2

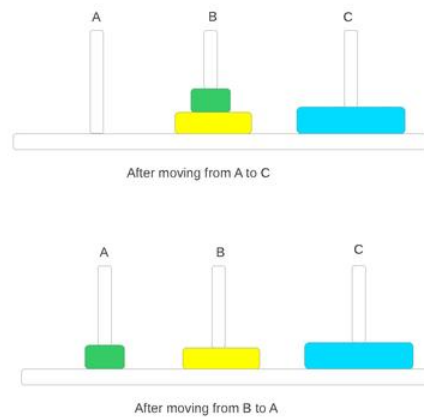


Figure 3.2.5: Towers of Hanoi step 3

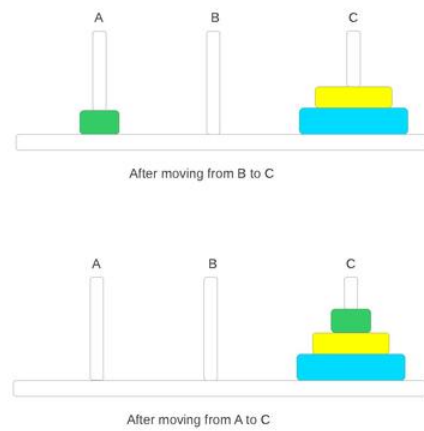


Figure 3.2.6: Towers of Hanoi step 4

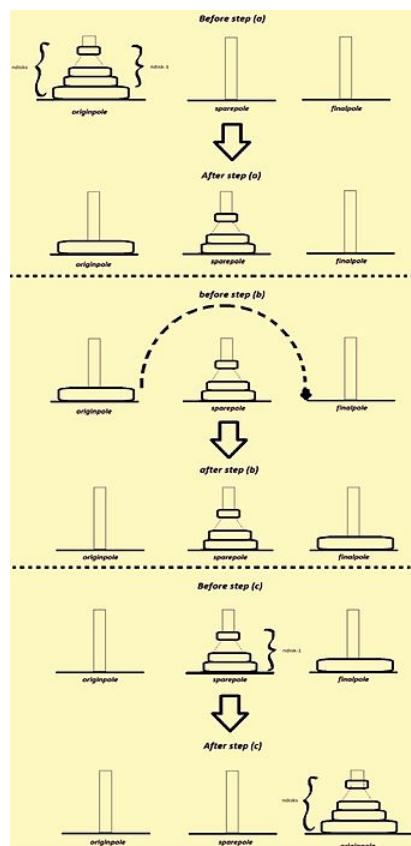


Figure 3.2.7: Tower of Hanoi

Output : (when there are 3 disks)

Let 1 be the smallest disk, 2 be the disk of medium size and 3 be the largest disk.

Move disk	From peg	To peg
1	A	C
2	A	B
1	C	B
3	A	C
1	B	A
2	B	C
1	A	C

Output : (when there are 4 disks)

Move disk	From peg	To peg
1	A	B
2	A	C
1	B	C
3	A	B
1	C	A
2	C	B
1	A	B
4	A	C
1	B	C
2	B	A
1	C	A
3	B	C
1	A	B
2	A	C
1	B	C

The C++ code for this solution can be implemented in two ways:

First Implementation (Without using Stacks)

Here we assume that A is first tower, B is second tower & C is third tower. (B is the intermediate)^[2]

```

01 void TowersofHanoi(int n, int a, int b, int c)
02 {
03     //Move top n disks from tower a to tower b, use tower c for intermediate
    storage.
04     if(n > 0)
05     {
06         TowersofHanoi(n-1, a, c, b); //recursion
07         cout << " Move top disk from tower " << a << " to tower " << b << endl
;
08         //Move n-1 disks from intermediate(b) to the source(a) back
09         TowersofHanoi(n-1, c, b, a); //recursion
10     }
11 }

```

Second Implementation (Using Stacks)

```
01 // Global variable, tower [1:3] are three towers
02 arrayStack<int> tower[4];
03 void TowerofHanoi(int n)
04 {
05     // Preprocessor for moveAndShow.
06     for (int d = n; d > 0; d--) //initialize
07         tower[1].push(d);      //add disk d to tower 1
08     moveAndShow(n, 1, 2, 3);    /*move n disks from tower 1 to tower 3
09                                tower 2 as intermediate tower*/
10 }
11
12 void moveAndShow(int n, int a, int b, int c)
13 {
14     // Move the top n disks from tower a to tower b showing states.
15     // Use tower c for intermediate storage.
16     if(n > 0)
17     {
18         moveAndShow(n-1, a, c, b); //recursion
19         int d = tower[a].top();    //move a disc from top of tower a to top
20         of                         //tower b
21         tower[a].pop();
22         tower[b].push(d);
23         showState();              //show state of 3 towers
24         moveAndShow(n-1, c, b, a); //recursion
25     }
```

However complexity for above written implementations is $O(2^n)$. So it's obvious that problem can only be solved for small values of n (generally $n \leq 30$). In case of the monks, the number of turns taken to transfer 64 disks, by following the above rules, will be 18,446,744,073,709,551,615; which will surely take a lot of time!^{[1] [2]}

Expression evaluation and syntax parsing

Calculators employing [reverse Polish notation](#) use a stack structure to hold values. Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form may be accomplished using a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most of the programming languages are [context-free languages](#) allowing them to be parsed with stack based machines.

Evaluation of an Infix Expression that is Fully Parenthesized

Input: (((2 * 5) - (1 * 2)) / (9 - 7))

Output: 4

Analysis: Five types of input characters

- * Opening bracket
- * Numbers
- * Operators
- * Closing bracket
- * New line character

Data structure requirement: A character stack

Algorithm

1. Read one input character
2. Actions at end of each input
 - Opening brackets (2.1) *Push* into stack and then Go to step (1)
 - Number (2.2) *Push* into stack and then Go to step (1)
 - Operator (2.3) *Push* into stack and then Go to step (1)
 - Closing brackets (2.4) *Pop* it from character stack
 - (2.4.1) if it is opening bracket, then discard it,
 - (2.4.2) *Pop* is used three times
 - The first popped element is assigned to op2
 - The second popped element is assigned to op1
 - The third popped element is assigned to op3
 - Evaluate op1 op op2
 - Convert the result into character and *push* into the stack
 - Go to step (2.4)
 - New line character (2.5) *Pop* from stack and print the answer
STOP

Result: The evaluation of the fully parenthesized infix expression is printed on the monitor as follows:

Input String: (((2 * 5) - (1 * 2)) / (9 - 7))

Input Symbol	Stack (from bottom to top)	Operation
(	(	
(	((	
(	(((	
2	(((2	
*	(((2 *	
5	(((2 * 5	
)	((10	$2 * 5 = 10$ & Push
-	((10 -	
(	((10 - (	
1	((10 - (1	
*	((10 - (1 *	
2	((10 - (1 * 2	
)	((10 - 2	$1 * 2 = 2$ & Push
)	(8	$10 - 2 = 8$ & Push
/	(8 /	
(	(8 / (	
9	(8 / (9	
-	(8 / (9 -	
9	(8 / (9 - 7	
)	(8 / 2	$9 - 7 = 2$ & Push
)	4	$8 / 2 = 4$ & Push
New line	Empty	Pop & Print

C Program

```
01 int main (int argc, char
02     struct ch *characterp;
03     struct integer *integertop;
04     char rd, op;
05     int i = 0, op1, op2;
06     characterp = cclearstack();
07     integertop = iclearstack();
08     while(1)
09     {
10         rd = argv[1][i++];
11         switch(rd)
12         {
13             case '+':
14             case '-':
15             case '/':
16             case '*':
17             case '(': characterp = cpush(characterp, rd);
18             break;
19             case ')': integertop = ipop (integertop, &op2);
20                     integertop = ipop (integertop, &op1);
21                     characterp = cpop (characterp, &op);
22                     while(op != '(')
23                     {
24                         integertop = ipush (integertop, eval(op, op1, op2));
25                         characterp = cpop (characterp, &op);
26                         if (op != '(')
27                         {
28                             integertop = ipop(integertop, &op2);
29                             integertop = ipop(integertop, &op1);
30                         }
31                     }
32                     break;
33             case '\0': while (! cemptystack(characterp))
34                     {
35                         characterp = cpop (characterp, &op);
36                         integertop = ipop(integertop, &op2);
37                         integertop = ipop(integertop, &op1);
38                         integertop = ipush(integertop, eval(op, op1, op2));
39                     }
40                     integertop = ipop(integertop, &op1);
41                     printf("\n The final solution is: %d\n", op1);
42                     return 0;
43             default: integertop = ipush(integertop, rd - '0');
44         }
45     }
46 }
47
48 int eval(char op, int op1, int op2)
49 {
50     switch (op)
51     {
52         case '+': return op1 + op2;
53         case '-': return op1 - op2;
54         case '/': return op1 / op2;
55         case '*': return op1 * op2;
56     }
57 }
```

Output of the program:

Input entered at the command line: $((2 * 5) - (1 * 2)) / (9 - 7)$ ^[3]

Evaluation of Infix Expression which is not fully parenthesized

Input: $(2 * 5 - 1 * 2) / (11 - 9)$

Output: 4

Analysis: There are five types of input characters which are:

- * Opening brackets
- * Numbers
- * Operators
- * Closing brackets
- * New line character ($\backslash n$)

We do not know what to do if an operator is read as an input character. By implementing the priority rule for operators, we have a solution to this problem.

The *Priority rule* we should perform comparative priority check if an operator is read, and then push it. If the stack *top* contains an operator of priority higher than or equal to the priority of the input operator, then we *pop* it and print it. We keep on performing the priority check until the *top* of stack either contains an operator of lower priority or if it does not contain an operator.

Data Structure Requirement for this problem: A character stack and an integer stack

Algorithm:

1. Read an input character
2. Actions that will be performed at the end of each input
 - Opening brackets (2.1) *Push* it into stack and then Go to step (1)
 - Digit (2.2) *Push* into stack, Go to step (1)
 - Operator (2.3) Do the comparative priority check
 - (2.3.1) if the character stack's *top* contains an operator of higher priority, then *pop* it into op
 - Pop* a number from integer stack into op2
 - Pop* another number from integer stack into op1
 - Calculate op1 op op2 and *push* the result into stack
 - Closing brackets (2.4) *Pop* from the character stack
 - (2.4.1) if it is an opening bracket, then discard it and Go to step (1)
 - (2.4.2) To op, assign the popped element
 - Pop* a number from integer stack and assign it to op2
 - Pop* another number from integer stack and assign it to op1
 - Calculate op1 op op2 and push the result into stack
 - Convert into character and *push* into stack
 - Go to the step (2.4)
 - New line character (2.5) Print the result after popping from the stack
STOP

Result: The evaluation of an infix expression that is not fully parenthesized is printed as follows:

Input String: $(2 * 5 - 1 * 2) / (11 - 9)$

Input Symbol	Character Stack (from bottom to top)	Integer Stack (from bottom to top)	Operation performed
(	(		
2	(	2	
*	(*		Push as * has higher priority
5	(*	2 5	
-	(*		Since '-' has less priority, we do $2 * 5 = 10$
	(-	10	We push 10 and then push '-'
1	(-	10 1	
*	(- *	10 1	Push * as it has higher priority
2	(- *	10 1 2	
)	(-	10 2	Perform $1 * 2 = 2$ and push it
	(	8	Pop - and $10 - 2 = 8$ and push, Pop (
/	/	8	
(	/ (	8	
11	/ (	8 11	
-	/ (-	8 11	
9	/ (-	8 11 9	
)	/	8 2	Perform $11 - 9 = 2$ and push it
New line		4	Perform $8 / 2 = 4$ and push it
		4	Print the output, which is 4

C Program

```
01 int main (int argc, char *argv[])
02 {
03     struct ch *characterop;
04     struct integer *integertop;
05     char rd, op;
06     int i = 0, op1, op2;
07     characterop = cclearstack();
08     integertop = iclearstack();
09     while(1)
10     {
11         rd = argv[1][i++];
12         switch(rd)
13         {
14             case '+':
15             case '-':
16             case '/':
17             case '*': while ((characterop->data != '(') && (!cemptystack(characterop->data)))
18                     {
19                         if(priority(rd) > (priority(characterop->data)))
20                             break;
21                         else
22                         {
23                             characterop = cpop(characterop, &op);
24                             integertop = ipop(integertop, &op2);
25                             integertop = ipop(integertop, &op1);
26                             integertop = ipush(integertop, eval(op, op1, op2));
27                         }
28                     }
29                     characterop = cpush(characterop, rd);
30                     break;
31             case '(': characterop = cpush(characterop, rd);
32             break;
33             case ')': integertop = ipop (integertop, &op2);
34                     integertop = ipop (integertop, &op1);
35                     characterop = cpop (characterop, &op);
36                     while(op != '(')
37                     {
38                         integertop = ipush (integertop, eval(op, op1, op2));
39                         characterop = cpop (characterop, &op);
40                         if (op != '(')
41                         {
42                             integertop = ipop(integertop, &op2);
43                             integertop = ipop(integertop, &op1);
44                         }
45                     }
46                     break;
47             case '\\0': while (!cemptystack(characterop))
48                     {
49                         characterop = cpop(characterop, &op);
50                         integertop = ipop(integertop, &op2);
51                         integertop = ipop(integertop, &op1);
52                         integertop = ipush(integertop, eval(op, op1, op2));
53                     }
54                     integertop = ipop(integertop, &op1);
55                     printf("\n The final solution is: %d", op1);
56                     return 0;
57             default: integertop = ipush(integertop, rd - '\\0');
58         }
59     }
60 }
61
```

```
62 int eval(char op, int op1, int op2)
63 {
64     switch (op)
65     {
66         case '+': return op1 + op2;
67         case '-': return op1 - op2;
68         case '/': return op1 / op2;
69         case '*': return op1 * op2;
70     }
71 }
72
73 int priority (char op)
74 {
75     switch(op)
76     {
77         case '^':
78         case '':
79
80
81
82         Content Reuse Link:
83         Under_Construction/Data_Structures/05%3A_Stacks_and_Queues/5.02%3A_Applications_
84
85         : return 3;
86         case '*':
87         case '/': return 2;
88         case '+':
89         case '-': return 1;
90     }
91 }
```

Output of the program:

Input entered at the command line: $(2 * 5 - 1 * 2) / (11 - 9)$

Output: 4 ^[3]

Evaluation of Prefix Expression

Input: $x + 6 * (y + z) ^ 3$

Output: ' 4

Analysis: There are three types of input characters

- * Numbers
- * Operators
- * New line character (\n)

Data structure requirement: A character stack and an integer stack

Algorithm:

1. Read one character input at a time and keep pushing it into the character stack line character is reached
2. Perform *pop* from the character stack. If the stack is empty, go to step (3)
 - Number (2.1) Push in to the integer stack and then go to
 - Operator (2.2) Assign the operator to op
 - Pop a number from integer stack and assign

Pop another number from integer stack and assign it to op2
Calculate op1 op op2 and push the output in stack. Go to step (2)

3. Pop the result from the integer stack and display the result

Result: The evaluation of prefix expression is printed as follows:

Input String: / - * 2 5 * 1 2 - 11 9

Input Symbol	Character Stack (from bottom to top)	Integer Stack (from bottom to top)	Operation performed
/	/		
-	/		
*	/ - *		
2	/ - * 2		
5	/ - * 2 5		
*	/ - * 2 5 *		
1	/ - * 2 5 * 1		
2	/ - * 2 5 * 1 2		
-	/ - * 2 5 * 1 2 -		
11	/ - * 2 5 * 1 2 - 11		
9	/ - * 2 5 * 1 2 - 11 9		
\n	/ - * 2 5 * 1 2 - 11	9	
	/ - * 2 5 * 1 2 -	9 11	
	/ - * 2 5 * 1 2	2	11 - 9 = 2
	/ - * 2 5 * 1	2 2	
	/ - * 2 5 *	2 2 1	
	/ - * 2 5	2 2	1 * 2 = 2
	/ - * 2	2 2 5	
	/ - *	2 2 5 2	
	/ -	2 2 10	5 * 2 = 10
	/	2 8	10 - 2 = 8
	Stack is empty	4	8 / 2 = 4
		Stack is empty	Print 4

C Program

```
01 int main (int argc, char *argv[])
02 {
03     struct ch *charactop = NULL;
04     struct integer *integertop = NULL;
05     char rd, op;
06     int i = 0, op1, op2;
07     charactop = cclearstack();
08     integertop = iclearstack();
09     rd = argv[1][i];
10     while(rd != '\0')
11     {
12         charactop = cpush(charactop, rd);
13         rd = argv[1][i++];
14     }
15     while(!emptystack(charactop))
16     {
17         charactop = cpop(charactop, rd);
18         switch(rd)
19         {
20             case '+':
21             case '-':
22             case '/':
23             case '*':
24                 op = rd;
25                 integertop = ipop(integertop, &op2);
26                 integertop = ipop(integertop, &op1);
27                 integertop = ipush(integertop, eval(op, op1, op2));
28                 break;
29             default:
30                 integertop = ipush(integertop, rd - '0');
31         }
32     }
33 }
34
35 int eval(char op, int op1, int op2)
36 {
37     switch (op)
38     {
39         case '+': return op1 + op2;
40         case '-': return op1 - op2;
41         case '/': return op1 / op2;
42         case '*': return op1 * op2;
43     }
44 }
45
46 int priority (char op)
47 {
48     switch(op)
49     {
50         case '^':
51         case '':
52
53
54
55         Content Reuse Link:
56         Under_Construction/Data_Structures/05%3A_Stacks_and_Queues/5.02%3A_Applications
57
58         : return 3;
59         case '*':
60         case '/': return 2;
```

```
61 |         case '+':
62 |         case '-': return 1;
63 |     }
64 | }
```

Output of the program:

Input entered at the command line: / - * 2 5 * 1 2 - 11 9

Output: 4 ^[3]

Conversion of an Infix expression that is fully parenthesized into a Postfix expression

Input: (((8 + 1) - (7 - 4)) / (11 - 9))

Output: 8 1 + 7 4 - - 11 9 - /

Analysis: There are five types of input characters which are:

- * Opening brackets
- * Numbers
- * Operators
- * Closing brackets
- * New line character (\n)

Requirement: A character stack

Algorithm:

1. Read an character input
2. Actions to be performed at end of each input
 - Opening brackets (2.1) *Push* into stack and then Go to step (1)
 - Number (2.2) Print and then Go to step (1)
 - Operator (2.3) *Push* into stack and then Go to step (1)
 - Closing brackets (2.4) *Pop* it from the stack
 - (2.4.1) If it is an operator, print it, Go to step
 - (2.4.2) If the popped element is an opening bracket discard it and go to step (1)
 - New line character (2.5) *STOP*

Therefore, the final output after conversion of an infix expression to a postfix expression is as follows:

Input	Operation	Stack (after op)	Output on monitor
(	(2.1) Push operand into stack	(	
(	(2.1) Push operand into stack	((	
(	(2.1) Push operand into stack	(((	
8	(2.2) Print it		8
+	(2.3) Push operator into stack	(((+	8
1	(2.2) Print it		8 1
)	(2.4) Pop from the stack: Since popped element is '+' print it	(((	8 1 +
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	((	8 1 +
-	(2.3) Push operator into stack	((-	
(	(2.1) Push operand into stack	((- (	
7	(2.2) Print it		8 1 + 7
-	(2.3) Push the operator in the stack	((- (-	
4	(2.2) Print it		8 1 + 7 4
)	(2.4) Pop from the stack: Since popped element is '-' print it	((- (	8 1 + 7 4 -
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	((-	
)	(2.4) Pop from the stack: Since popped element is '-' print it	((	8 1 + 7 4 - -
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	(	
/	(2.3) Push the operand into the stack	(/	
(	(2.1) Push into the stack	(/ (	
11	(2.2) Print it		8 1 + 7 4 - - 11
-	(2.3) Push the operand into the stack	(/ (-	
9	(2.2) Print it		8 1 + 7 4 - - 11 9
)	(2.4) Pop from the stack: Since popped element is '-' print it	(/ (	8 1 + 7 4 - - 11 9 -
	(2.4) Pop from the stack: Since popped element is '(' we ignore	(/	

Input	Operation	Stack (after op)	Output on monitor
	it and read next character		
)	(2.4) Pop from the stack: Since popped element is '/' print it	(	8 1 + 7 4 - - 11 9 - /
	(2.4) Pop from the stack: Since popped element is '(' we ignore it and read next character	Stack is empty	
New line character	(2.5) STOP		

Rearranging railroad cars

Problem Description

It's a very nice application of stacks. Consider that a freight train has n railroad cars. Each to be left at different station. They're numbered 1 through n & freight train visits these stations in order n through 1. Obviously, the railroad cars are labeled by their destination. To facilitate removal of the cars from the train, we must rearrange them in ascending order of their number (i.e. 1 through n). When cars are in this order, they can be detached at each station. We rearrange cars at a shunting yard that has **input track**, **output track** & k holding tracks between input & output tracks (i.e. **holding track**).

Solution Strategy

To rearrange cars, we examine the cars on the input from front to back. If the car being examined is next one in the output arrangement, we move it directly to **output track**. If not, we move it to the **holding track** & leave it there until it's time to place it to the **output track**. The holding tracks operate in a LIFO manner as the cars enter & leave these tracks from top. When rearranging cars only following moves are permitted:

- A car may be moved from front (i.e. right end) of the input track to the top of one of the **holding tracks** or to the left end of the output track.
- A car may be moved from the top of **holding track** to left end of the **output track**.

The figure shows a shunting yard with $k = 3$, holding tracks **H1**, **H2** & **H3**, also $n = 9$. The n cars of freight train begin in the input track & are to end up in the output track in order 1 through n from right to left. The cars initially are in the order 5,8,1,7,4,2,9,6,3 from back to front. Later cars are rearranged in desired order.

A Three Track Example

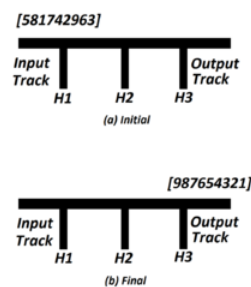


Figure 3.2.8: Railroad cars example

- Consider the input arrangement from figure, here we note that the car 3 is at the front, so it can't be output yet, as it to be preceded by cars 1 & 2. So car 3 is detached & moved to holding track **H1**.
- The next car 6 can't be output & it is moved to holding track **H2**. Because we have to output car 3 before car 6 & this will not possible if we move car 6 to holding track **H1**.
- Now it's obvious that we move car 9 to **H3**.

The requirement of rearrangement of cars on any holding track is that the cars should be preferred to arrange in ascending order from top to bottom.

- So car 2 is now moved to holding track H1 so that it satisfies the previous statement. If we move car 2 to H2 or H3, then we've no place to move cars 4,5,7,8. **The least restrictions on future car placement arise when the new car λ is moved to the holding track that has a car at its top with smallest label Ψ such that $\lambda < \Psi$. We may call it an assignment rule to decide whether a particular car belongs to a specific holding track.**
- When car 4 is considered, there are three places to move the car H1,H2,H3. The top of these tracks are 2,6,9. So using above mentioned Assignment rule, we move car 4 to H2.
- The car 7 is moved to H3.
- The next car 1 has the least label, so it's moved to output track.
- Now it's time for car 2 & 3 to output which are from H1 (in short all the cars from H1 are appended to car 1 on output track).

The car 4 is moved to output track. No other cars can be moved to output track at this time.

- The next car 8 is moved to holding track H1.
- Car 5 is output from input track. Car 6 is moved to output track from H2, so is the 7 from H3, 8 from H1 & 9 from H3.

Quicksort

Sorting means arranging a group of elements in a particular order. Be it ascending or descending, by cardinality or alphabetical order or variations thereof. The resulting ordering possibilities will only be limited by the type of the source elements.

Quicksort is an algorithm of the *divide and conquer* type. In this method, to sort a set of numbers, we reduce it to two smaller sets, and then sort these smaller sets.

This can be explained with the help of the following example:

Suppose **A** is a list of the following numbers:

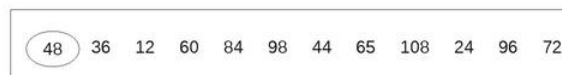


Figure 3.2.9

In the reduction step, we find the final position of one of the numbers. In this case, let us assume that we have to find the final position of 48, which is the first number in the list.

To accomplish this, we adopt the following method. Begin with the last number, and move from right to left. Compare each number with 48. If the number is smaller than 48, we stop at that number and swap it with 48.

In our case, the number is 24. Hence, we swap 24 and 48.

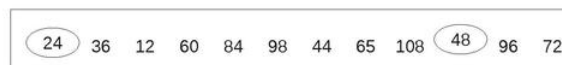


Figure 3.2.10

The numbers 96 and 72 to the right of 48, are greater than 48. Now beginning with 24, scan the numbers in the opposite direction, that is from left to right. Compare every number with 48 until you find a number that is greater than 48.

In this case, it is 60. Therefore we swap 48 and 60.



Figure 3.2.11

Note that the numbers 12, 24 and 36 to the left of 48 are all smaller than 48. Now, start scanning numbers from 60, in the right to left direction. As soon as you find lesser number, swap it with 48.

In this case, it is 44. Swap it with 48. The final result is:

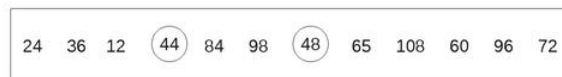


Figure 3.2.12

Now, beginning with 44, scan the list from left to right, until you find a number greater than 48.

Such a number is 84. Swap it with 48. The final result is:



Figure 3.2.13

Now, beginning with 84, traverse the list from right to left, until you reach a number lesser than 48. We do not find such a number before reaching 48. This means that all the numbers in the list have been scanned and compared with 48. Also, we notice that all numbers less than 48 are to the left of it, and all numbers greater than 48, are to its right.

The final partitions look as follows:

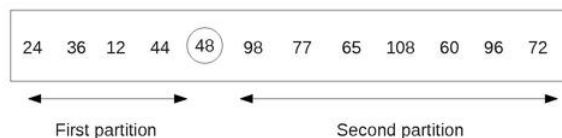


Figure 3.2.14

Therefore, 48 has been placed in its proper position and now our task is reduced to sorting the two partitions. This above step of creating partitions can be repeated with every partition containing 2 or more elements. As we can process only a single partition at a time, we should be able to keep track of the other partitions, for future processing.

This is done by using two **stacks** called LOWERBOUND and UPPERBOUND, to temporarily store these partitions. The addresses of the first and last elements of the partitions are pushed into the LOWERBOUND and UPPERBOUND stacks respectively. Now, the above reduction step is applied to the partitions only after its boundary values are *popped* from the stack.

We can understand this from the following example:

Take the above list A with 12 elements. The algorithm starts by pushing the boundary values of A, that is 1 and 12 into the LOWERBOUND and UPPERBOUND stacks respectively. Therefore the stacks look as follows:

LOWERBOUND: 1	UPPERBOUND: 12
---------------	----------------

To perform the reduction step, the values of the stack top are popped from the stack. Therefore, both the stacks become empty.

LOWERBOUND: {empty}	UPPERBOUND: {empty}
---------------------	---------------------

Now, the reduction step causes 48 to be fixed to the 5th position and creates two partitions, one from position 1 to 4 and the other from position 6 to 12. Hence, the values 1 and 6 are pushed into the LOWERBOUND stack and 4 and 12 are pushed into the UPPERBOUND stack.

LOWERBOUND: 1, 6	UPPERBOUND: 4, 12
------------------	-------------------

For applying the reduction step again, the values at the stack top are popped. Therefore, the values 6 and 12 are popped. Therefore the stacks look like:

LOWERBOUND: 1

UPPERBOUND: 4

The reduction step is now applied to the second partition, that is from the 6th to 12th element.

A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]
98	84	65	108	60	96	72
72	84	65	108	60	96	98
72	84	65	98	60	96	108
72	84	65	96	60	98	108

Figure 3.2.15

After the reduction step, 98 is fixed in the 11th position. So, the second partition has only one element. Therefore, we push the upper and lower boundary values of the first partition onto the stack. So, the stacks are as follows: LOWERBOUND: 1, 6 UPPERBOUND: 4, 10

The processing proceeds in the following way and ends when the stacks do not contain any upper and lower bounds of the partition to be processed, and the list gets sorted.

The Stock Span Problem

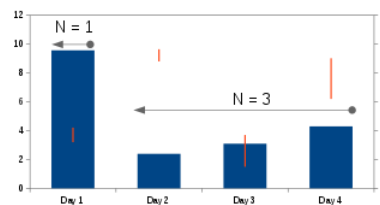


Figure 3.2.16: The Stockspan Problem

In the stock span problem, we will solve a financial problem with the help of stacks.

Suppose, for a stock, we have a series of n daily price quotes, the *span* of the stock's price on a particular day is defined as the maximum number of consecutive days for which the price of the stock on the current day is less than or equal to its price on that day.

An algorithm which has Quadratic Time Complexity

Input: An array P with n elements

Output: An array S of n elements such that $S[i]$ is the largest integer k such that $k \leq i + 1$ and $P[j] \leq P[i]$ for $j = i - k + 1, \dots, i$

Algorithm:

1. Initialize an array P which contains the daily prices of the stocks
2. Initialize an array S which will store the span of the stock
3. **for** $i = 0$ to $i = n - 1$
 - 3.1 Initialize k to zero
 - 3.2 Done with a *false* condition
 - 3.3 **repeat**
 - 3.3.1 **if** ($P[i - k] \leq P[i]$) **then**
Increment k by 1
 - 3.3.2 **else**

```

                                Done with true condition
3.4 Till ( $k > i$ ) or done with processing
    Assign value of  $k$  to  $S[i]$  to get the span of the stock
4. Return array  $S$ 
```

Now, analyzing this algorithm for running time, we observe:

- We have initialized the array S at the beginning and returned it at the end. This is a constant time operation, hence takes $O(n)$ time
- The *repeat* loop is nested within the *for* loop. The *for* loop, whose counter is i is executed n times. The statements which are not in the repeat loop, but in the *for* loop are executed n times. Therefore these statements and the incrementing and condition testing of i take $O(n)$ time.
- In repetition of i for the outer *for* loop, the body of the inner *repeat* loop is executed maximum $i + 1$ times. In the worst case, element $S[i]$ is greater than all the previous elements. So, testing for the *if* condition, the statement after that, as well as testing the *until* condition, will be performed $i + 1$ times during iteration i for the outer *for* loop. Hence, the total time taken by the inner loop is $O(n(n + 1)/2)$, which is $O(n^2)$

The running time of all these steps is calculated by adding the time taken by all these three steps. The first two terms are $O(n)$ while the last term is $O(n^2)$. Therefore the total running time of the algorithm is $O(n^2)$.

An algorithm that has Linear Time Complexity

In order to calculate the span more efficiently, we see that the span on a particular day can be easily calculated if we know the closest day before i , such that the price of the stocks on that day was higher than the price of the stocks on the present day. If there exists such a day, we can represent it by $h(i)$ and initialize $h(i)$ to be -1.

Therefore the span of a particular day is given by the formula, $s = i - h(i)$.

To implement this logic, we use a stack as an abstract data type to store the days i , $h(i)$, $h(h(i))$ and so on. When we go from day $i-1$ to i , we pop the days when the price of the stock was less than or equal to $p(i)$ and then push the value of day i back into the stack.

Here, we assume that the stack is implemented by operations that take $O(1)$ that is constant time. The algorithm is as follows:

Input: An array P with n elements and an empty stack N

Output: An array S of n elements such that $P[i]$ is the largest integer k such that $k \leq i + 1$ and $P[j] \leq P[i]$ for $j = i - k + 1, \dots, i$

Algorithm:

```

1. Initialize an array  $P$  which contains the daily prices of the stocks
2. Initialize an array  $S$  which will store the span of the stock
3. for  $i = 0$  to  $i = n - 1$ 
    3.1 Initialize  $k$  to zero
    3.2 Done with a false condition
    3.3 while not (Stack  $N$  is empty or done with processing)
        3.3.1 if (  $P[i] \geq P[N.top()]$  ) then
            Pop a value from stack  $N$ 
        3.3.2 else
            Done with true condition
    3.4 if Stack  $N$  is empty
        3.4.1 Initialize  $h$  to -1
    3.5 else
        3.5.1 Initialize stack top to  $h$ 
        3.5.2 Put the value of  $h - i$  in  $S[i]$ 
```

3.5.3 Push the value of i in N 4. Return array S

Now, analyzing this algorithm for running time, we observe:

- We have initialized the array S at the beginning and returned it at the end. This is a constant time operation, hence takes $O(n)$ time
- The *while* loop is nested within the *for* loop. The *for* loop, whose counter is i is executed n times. The statements which are not in the repeat loop, but in the *for* loop are executed n times. Therefore these statements and the incrementing and condition testing of i take $O(n)$ time.
- Now, observe the inner *while* loop during i repetitions of the *for* loop. The statement *done with a true condition* is done at most once, since it causes an exit from the loop. Let us say that $t(i)$ is the number of times statement *Pop a value from stack N* is executed. So it becomes clear that *while not (Stack N is empty or done with processing)* is tested maximum $t(i) + 1$ times.
- Adding the running time of all the operations in the *while* loop, we get:

$$\sum_{i=0}^{n-1} t(i) + 1$$

- An element once popped from the stack N is never pushed back into it. Therefore,

$$\sum_{i=1}^{n-1} t(i)$$

So, the running time of all the statements in the *while* loop is $O(n)$

The running time of all the steps in the algorithm is calculated by adding the time taken by all these steps. The run time of each step is $O(n)$. Hence the running time complexity of this algorithm is $O(n)$.

Related Links

- [Stack \(Wikipedia\)](#)

Footnotes

1. ↑ ^{a b} Dromey, R.G. *How to Solve it by Computer*. Prentice Hall of India.
2. ↑ ^{a b} Data structures, Algorithms and Applications in C++ by Sartaj Sahni
3. ↑ ^{a b c} Gopal, Arpita. *Magnifying Data Structures*. PHI.

3.2: Applications of Stacks is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

3.3: Queues

A queue is a basic data structure that is used throughout programming. You can think of it as a line in a grocery store. The first one in the line is the first one to be served. Just like a queue.

A queue is also called a FIFO (First In First Out) to demonstrate the way it accesses data.

Queue<item-type> Operations

enqueue(*new-item: item-type*)

Adds an item onto the end of the queue.

front(): *item-type*

Returns the item at the front of the queue.

dequeue()

Removes the item from the front of the queue.

is-empty(): *Boolean*

True if no more items can be dequeued and there is no front item.

is-full(): *Boolean*

True if no more items can be enqueued.

get-size(): *Integer*

Returns the number of elements in the queue.

All operations except **get-size**() can be performed in $O(1)$ time. **get-size**() runs in at worst $O(N)$.

Linked List Implementation

The basic linked list implementation uses a singly-linked list with a tail pointer to keep track of the back of the queue.

```
type Queue<item_type>
  data list: Singly Linked List<item_type>
  data tail: List Iterator<item_type>

  constructor()
    list := new Singly-Linked-List()
    tail := list.get-begin() # null
  end constructor
```

When you want to **enqueue** something, you simply add it to the back of the item pointed to by the tail pointer. So the previous tail is considered next compared to the item being added and the tail pointer points to the new item. If the list was empty, this doesn't work, since the tail iterator doesn't refer to anything

```
method enqueue(new_item: item_type)
  if is-empty()
    list.prepend(new_item)
    tail := list.get-begin()
  else
    list.insert_after(new_item, tail)
```

```
        tail.move-next()
    end if
end method
```

The **front** item on the queue is just the one referred to by the linked list's head pointer

```
method front():item_type
    return list.get-begin().get-value()
end method
```

When you want to **dequeue** something off the list, simply point the head pointer to the previous from head item. The old head item is the one you removed of the list. If the list is now empty, we have to fix the tail iterator.

```
method dequeue()
    list.remove-first()
    if is-empty()
        tail := list.get-begin()
    end if
end method
```

A check for emptiness is easy. Just check if the list is empty.

```
method is-empty():Boolean
    return list.is-empty()
end method
```

A check for full is simple. Linked lists are considered to be limitless in size.

```
method is-full():Boolean
    return False
end method
```

A check for the size is again passed through to the list.

```
method get-size():Integer
    return list.get-size()
end method
end type
```

Performance Analysis

In a linked list, accessing the first element is an $O(1)$ operation because the list contains a pointer directly to it. Therefore, enqueue, front, and dequeue are a quick $O(1)$ operations.

The checks for empty/fullness as done here are also $O(1)$.

The performance of `getSize()` depends on the performance of the corresponding operation in the linked list implementation. It could be either $O(n)$, or $O(1)$, depending on what time/space tradeoff is made. Most of the time, users of a Queue do not use the `getSize()` operation, and so a bit of space can be saved by not optimizing it.

Related Links

- [Queue \(Wikipedia\)](#)

3.3: [Queues](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- 5.3: [Queues](#) by Wikibooks - Data Structures is licensed [CC BY-SA 4.0](#).

3.4: Deques

A Deque is a homogeneous list of elements in which insertions and deletion operations are performed on both the ends. Because of this property it is known as double ended queue i.e. Deque

Deque has two types:

1. Input restricted queue: It allows insertion at only one end
2. Output restricted queue: It allows deletion at only one end

Related Links

- [Deque \(Double-Ended QUEUE\)](#)

3.4: Deques is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- [5.4: Deques](#) by [Wikibooks - Data Structures](#) is licensed [CC BY-SA 4.0](#).

CHAPTER OVERVIEW

4: Linked Lists

In this chapter, we continue to study implementations of the `List` interface, this time using pointer-based data structures rather than arrays. The structures in this chapter are made up of nodes that contain the list items. Using references (pointers), the nodes are linked together into a sequence. We first study singly-linked lists, which can implement `Stack` and (FIFO) `Queue` operations in constant time per operation and then move on to doubly-linked lists, which can implement `Deque` operations in constant time.

Linked lists have advantages and disadvantages when compared to array-based implementations of the `List` interface. The primary disadvantage is that we lose the ability to access any element using `get(i)` or `set(i, x)` in constant time. Instead, we have to walk through the list, one element at a time, until we reach the i th element. The primary advantage is that they are more dynamic: Given a reference to any list node u , we can delete u or insert a node adjacent to u in constant time. This is true no matter where u is in the list.

[4.1: SLList - A Singly-Linked List](#)

[4.2: DLList - A Doubly-Linked List](#)

[4.3: SEList - A Space-Efficient Linked List](#)

[4.4: Discussion and Exercises](#)

4: Linked Lists is shared under a [CC BY](#) license and was authored, remixed, and/or curated by LibreTexts.

4.1: SLList - A Singly-Linked List

An SLList (singly-linked list) is a sequence of Nodes. Each node u stores a data value $u.x$ and a reference $u.next$ to the next node in the sequence. For the last node w in the sequence, $w.next = null$

```
1 class Node {
2     T x;
3     Node next;
4 }
```

For efficiency, an SLList uses variables `head` and `tail` to keep track of the first and last node in the sequence, as well as an integer `n` to keep track of the length of the sequence:

```
1 Node head;
2 Node tail;
3 int n;
```

A sequence of Stack and Queue operations on an SLList is illustrated in Figure 4.1.1.

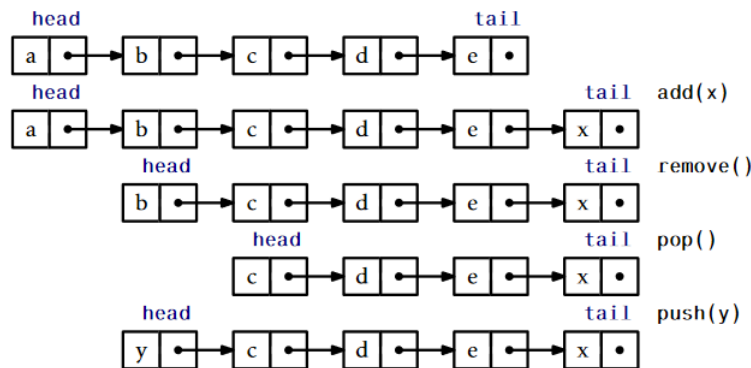


Figure 4.1.1: A sequence of Queue (`add(x)` and `remove()`) and Stack (`push(x)` and `pop()`) operations on an SLList.

An SLList can efficiently implement the Stack operations `push()` and `pop()` by adding and removing elements at the head of the sequence. The `push()` operation simply creates a new node u with data value x , sets $u.next$ to the old head of the list and makes u the new head of the list. Finally, it increments `n` since the size of the SLList has increased by one:

```
01 T push(T x) {
02     Node u = new Node();
03     u.x = x;
04     u.next = head;
05     head = u;
06     if (n == 0)
07         tail = u;
08     n++;
09     return x;
10 }
```

The `pop()` operation, after checking that the `SLList` is not empty, removes the head by setting `head = head.next` and decrementing `n`. A special case occurs when the last element is being removed, in which case `tail` is set to `null`:

```
1 T pop() {
2     if (n == 0) return null;
3     T x = head.x;
4     head = head.next;
5     if (--n == 0) tail = null;
6     return x;
7 }
```

Clearly, both the `push(x)` and `pop()` operations run in $O(1)$ time.

4.1.1 Queue Operations

An `SLList` can also implement the FIFO queue operations `add(x)` and `remove()` in constant time. Removals are done from the head of the list, and are identical to the `pop()` operation:

```
1 T remove() {
2     if (n == 0) return null;
3     T x = head.x;
4     head = head.next;
5     if (--n == 0) tail = null;
6     return x;
7 }
```

Additions, on the other hand, are done at the tail of the list. In most cases, this is done by setting `tail.next = u`, where `u` is the newly created node that contains `x`. However, a special case occurs when `n = 0`, in which case `tail = head = null`. In this case, both `tail` and `head` are set to `u`.

```
01 boolean add(T x) {
02     Node u = new Node();
03     u.x = x;
04     if (n == 0) {
05         head = u;
06     } else {
07         tail.next = u;
08     }
09     tail = u;
10     n++;
11     return true;
12 }
```

Clearly, both `add(x)` and `remove()` take constant time.

4.1.2 Summary

The following theorem summarizes the performance of an `SLList`:

Theorem 4.1.1.

An `SLList` implements the Stack and (FIFO) Queue interfaces. The `push(x)`, `pop()`, `add(x)` and `remove()` operations run in $O(1)$ time per operation.

An `SLList` nearly implements the full set of Deque operations. The only missing operation is removing from the tail of an `SLList`. Removing from the tail of an `SLList` is difficult because it requires updating the value of `tail` so that it points to the node `w` that

precedes `tail` in the `SLList`; this is the node `w` such that `w.next = tail`. Unfortunately, the only way to get to `w` is by traversing the `SLList` starting at `head` and taking $n - 2$ steps.

4.1: [SLList - A Singly-Linked List](#) is shared under a [CC BY](#) license and was authored, remixed, and/or curated by LibreTexts.

- 3.1: [SLList - A Singly-Linked List](#) by [Pat Morin](#) is licensed [CC BY 4.0](#).

4.2: DLList - A Doubly-Linked List

A DLList (doubly-linked list) is very similar to an SLList except that each node u in a DLList has references to both the node $u.next$ that follows it and the node $u.prev$ that precedes it.

```
1 class Node {
2     T x;
3     Node prev, next;
4 }
```

When implementing an SLList, we saw that there were always several special cases to worry about. For example, removing the last element from an SLList or adding an element to an empty SLList requires care to ensure that head and tail are correctly updated. In a DLList, the number of these special cases increases considerably. Perhaps the cleanest way to take care of all these special cases in a DLList is to introduce a dummy node. This is a node that does not contain any data, but acts as a placeholder so that there are no special nodes; every node has both a next and a prev, with dummy acting as the node that follows the last node in the list and that precedes the first node in the list. In this way, the nodes of the list are (doubly-)linked into a cycle, as illustrated in Figure 4.2.1.

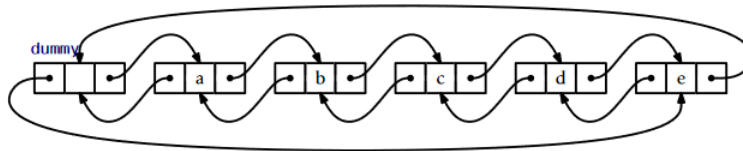


Figure 4.2.1: A DLList containing a,b,c,d,e.

```
1 int n;
2 Node dummy;
3 DLList() {
4     dummy = new Node();
5     dummy.next = dummy;
6     dummy.prev = dummy;
7     n = 0;
8 }
```

Finding the node with a particular index in a DLList is easy; we can either start at the head of the list ($dummy.next$) and work forward, or start at the tail of the list ($dummy.prev$) and work backward. This allows us to reach the i th node in $O(1 + \min\{i, n - i\})$ time:

```
01 Node getNode(int i) {
02     Node p = null;
03     if (i < n / 2) {
04         p = dummy.next;
05         for (int j = 0; j < i; j++)
06             p = p.next;
07     } else {
08         p = dummy;
09         for (int j = n; j > i; j--)
10             p = p.prev;
11     }
12     return p;
13 }
```

The `get(i)` and `set(i, x)` operations are now also easy. We first find the i th node and then get or set its x value:

```

01 T get(int i) {
02     if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
03     return getNode(i).x;
04 }
05 T set(int i, T x) {
06     if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
07     Node u = getNode(i);
08     T y = u.x;
09     u.x = x;
10     return y;
11 }

```

The running time of these operations is dominated by the time it takes to find the i th node, and is therefore $O(1 + \min\{i, n - i\})$.

4.2.1 Adding and Removing

If we have a reference to a node w in a `DLList` and we want to insert a node u before w , then this is just a matter of setting $u.next = w$, $u.prev = w.prev$, and then adjusting $u.prev.next$ and $u.next.prev$. (See Figure 4.2.2.) Thanks to the dummy node, there is no need to worry about $w.prev$ or $w.next$ not existing.

```

01 Node addBefore(Node w, T x) {
02     Node u = new Node();
03     u.x = x;
04     u.prev = w.prev;
05     u.next = w;
06     u.next.prev = u;
07     u.prev.next = u;
08     n++;
09     return u;
10 }

```

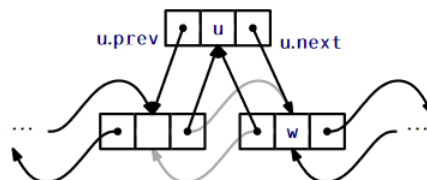


Figure 4.2.2: Adding the node u before the node w in a `DLList`.

Now, the list operation `add(i, x)` is trivial to implement. We find the i th node in the `DLList` and insert a new node u that contains x just before it.

```

1 void add(int i, T x) {
2     if (i < 0 || i > n) throw new IndexOutOfBoundsException();
3     addBefore(getNode(i), x);
4 }

```

The only non-constant part of the running time of `add(i, x)` is the time it takes to find the i th node (using `getNode(i)`). Thus, `add(i, x)` runs in $O(1 + \min\{i, n - i\})$ time.

Removing a node `w` from a `DLList` is easy. We only need to adjust pointers at `w.next` and `w.prev` so that they skip over `w`. Again, the use of the dummy node eliminates the need to consider any special cases:

```
1 void remove(Node w) {
2     w.prev.next = w.next;
3     w.next.prev = w.prev;
4     n--;
5 }
```

Now the `remove(i)` operation is trivial. We find the node with index `i` and remove it:

```
1 T remove(int i) {
2     if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
3     Node w = getNode(i);
4     remove(w);
5     return w.x;
6 }
```

Again, the only expensive part of this operation is finding the i th node using `getNode(i)`, so `remove(i)` runs in $O(1 + \min\{i, n - i\})$ time.

4.2.2 Summary

The following theorem summarizes the performance of a `DLList`:

Theorem 4.2.1.

A `DLList` implements the `List` interface. In this implementation, the `get(i)`, `set(i, x)`, `add(i, x)` and `remove(i)` operations run in $O(1 + \min\{i, n - i\})$ time per operation.

It is worth noting that, if we ignore the cost of the `getNode(i)` operation, then all operations on a `DLList` take constant time. Thus, the only expensive part of operations on a `DLList` is finding the relevant node. Once we have the relevant node, adding, removing, or accessing the data at that node takes only constant time.

This is in sharp contrast to array-based `List` implementations (see Open Data Structures, Chapter 2); in those implementations, the relevant array item can be found in constant time. However, addition or removal requires shifting elements in the array and, in general, takes non-constant time.

For this reason, linked list structures are well-suited to applications where references to list nodes can be obtained through external means. An example of this is the `LinkedHashSet` data structure found in the Java Collections Framework, in which a set of items is stored in a doubly-linked list and the nodes of the doubly-linked list are stored in a hash table (discussed in Chapter 7). When elements are removed from a `LinkedHashSet`, the hash table is used to find the relevant list node in constant time and then the list node is deleted (also in constant time).

4.2: `DLList` - A Doubly-Linked List is shared under a [CC BY](#) license and was authored, remixed, and/or curated by LibreTexts.

4.3: SEList - A Space-Efficient Linked List

One of the drawbacks of linked lists (besides the time it takes to access elements that are deep within the list) is their space usage. Each node in a `DLList` requires an additional two references to the next and previous nodes in the list. Two of the fields in a `Node` are dedicated to maintaining the list, and only one of the fields is for storing data!

An `SEList` (space-efficient list) reduces this wasted space using a simple idea: Rather than store individual elements in a `DLList`, we store a block (array) containing several items. More precisely, an `SEList` is parameterized by a block size b . Each individual node in an `SEList` stores a block that can hold up to $b + 1$ elements.

For reasons that will become clear later, it will be helpful if we can do `Deque` operations on each block. The data structure that we choose for this is a `BDeque` (bounded deque), derived from the `ArrayDeque` structure described in [Open Data Structures, Section 2.4](#). The `BDeque` differs from the `ArrayDeque` in one small way: When a new `BDeque` is created, the size of the backing array a is fixed at $b + 1$ and never grows or shrinks. The important property of a `BDeque` is that it allows for the addition or removal of elements at either the front or back in constant time. This will be useful as elements are shifted from one block to another.

```
1 class BDeque extends ArrayDeque<T> {
2     BDeque() {
3         super(SEList.this.type());
4         a = newArray(b+1);
5     }
6     void resize() { }
7 }
```

An `SEList` is then a doubly-linked list of blocks:

```
1 class Node {
2     BDeque d;
3     Node prev, next;
4 }
```

```
1 int n;
2 Node dummy;
```

4.3.1 Space Requirements

An `SEList` places very tight restrictions on the number of elements in a block: Unless a block is the last block, then that block contains at least $b - 1$ and at most $b + 1$ elements. This means that, if an `SEList` contains n elements, then it has at most

$$n/(b - 1) + 1 = O(n/b)$$

blocks. The `BDeque` for each block contains an array of length $b + 1$ but, for every block except the last, at most a constant amount of space is wasted in this array. The remaining memory used by a block is also constant. This means that the wasted space in an `SEList` is only $O(b + n/b)$. By choosing a value of b within a constant factor of $\sqrt{n}$, we can make the space-overhead of an `SEList` approach the $\sqrt{n}$ lower bound given in [Open Data Structures, Section 2.6.2](#).

4.3.2 Finding Elements

The first challenge we face with an `SEList` is finding the list item with a given index i . Note that the location of an element consists of two parts:

1. The node u that contains the block that contains the element with index i ; and

2. the index j of the element within its block.

```
1 class Location {
2     Node u;
3     int j;
4     Location(Node u, int j) {
5         this.u = u;
6         this.j = j;
7     }
8 }
```

To find the block that contains a particular element, we proceed the same way as we do in a `DLList`. We either start at the front of the list and traverse in the forward direction, or at the back of the list and traverse backwards until we reach the node we want. The only difference is that, each time we move from one node to the next, we skip over a whole block of elements.

```
01 Location getLocation(int i) {
02     if (i < n/2) {
03         Node u = dummy.next;
04         while (i >= u.d.size()) {
05             i -= u.d.size();
06             u = u.next;
07         }
08         return new Location(u, i);
09     } else {
10         Node u = dummy;
11         int idx = n;
12         while (i < idx) {
13             u = u.prev;
14             idx -= u.d.size();
15         }
16         return new Location(u, i-idx);
17     }
18 }
```

Remember that, with the exception of at most one block, each block contains at least $b - 1$ elements, so each step in our search gets us $b - 1$ elements closer to the element we are looking for. If we are searching forward, this means that we reach the node we want after $O(1 + i/b)$ steps. If we search backwards, then we reach the node we want after $O(1 + (n - i)/b)$ steps. The algorithm takes the smaller of these two quantities depending on the value of i , so the time to locate the item with index i is $O(1 + \min\{i, n - i\}/b)$.

Once we know how to locate the item with index i , the `get(i)` and `set(i, x)` operations translate into getting or setting a particular index in the correct block:

```
01 T get(int i) {
02     if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
03     Location l = getLocation(i);
04     return l.u.d.get(l.j);
05 }
06 T set(int i, T x) {
07     if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
08     Location l = getLocation(i);
09     T y = l.u.d.get(l.j);
10     l.u.d.set(l.j, x);
11     return y;
12 }
```

The running times of these operations are dominated by the time it takes to locate the item, so they also run in $O(1 + \min\{i, n - i\}/b)$ time.

4.3.3 Adding an Element

Adding elements to an SEList is a little more complicated. Before considering the general case, we consider the easier operation, `add(x)`, in which `x` is added to the end of the list. If the last block is full (or does not exist because there are no blocks yet), then we first allocate a new block and append it to the list of blocks. Now that we are sure that the last block exists and is not full, we append `x` to the last block.

```

1  boolean add(T x) {
2      Node last = dummy.prev;
3      if (last == dummy || last.d.size() == b+1) {
4          last = addBefore(dummy);
5      }
6      last.d.add(x);
7      n++;
8      return true;
9  }

```

Things get more complicated when we add to the interior of the list using `add(i, x)`. We first locate `i` to get the node `u` whose block contains the i th list item. The problem is that we want to insert `x` into `u`'s block, but we have to be prepared for the case where `u`'s block already contains $b + 1$ elements, so that it is full and there is no room for `x`.

Let $u_0, u_1, u_2, \dots$ denote `u`, `u.next`, `u.next.next`, and so on. We explore $u_0, u_1, u_2, \dots$ looking for a node that can provide space for `x`. Three cases can occur during our space exploration (see Figure 4.3.1):

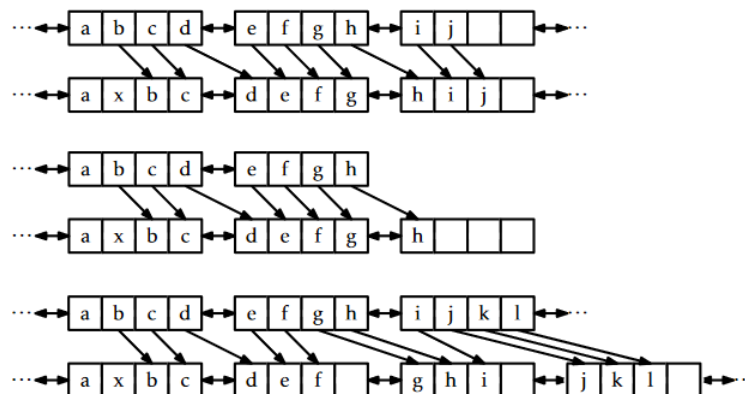


Figure 4.3.1: The three cases that occur during the addition of an item `x` in the interior of an SEList. (This SEList has block size b .)

1. We quickly (in $r + 1 \leq b$ steps) find a node u_r whose block is not full. In this case, we perform r shifts of an element from one block into the next, so that the free space in u_r becomes a free space in u_0 . We can then insert `x` into u_0 's block.
2. We quickly (in $r + 1 \leq b$ steps) run off the end of the list of blocks. In this case, we add a new empty block to the end of the list of blocks and proceed as in the first case.

3. After b steps we do not find any block that is not full. In this case, $u_0, \dots, u_{b-1}$ is a sequence of b blocks that each contain $b+1$ elements. We insert a new block u_b at the end of this sequence and spread the original $b(b+1)$ elements so that each block of $u_0, \dots, u_b$ contains exactly b elements. Now u_0 's block contains only b elements so it has room for us to insert x .

```

01 void add(int i, T x) {
02     if (i < 0 || i > n) throw new IndexOutOfBoundsException();
03     if (i == n) {
04         add(x);
05         return;
06     }
07     Location l = getLocation(i);
08     Node u = l.u;
09     int r = 0;
10     while (r < b && u != dummy && u.d.size() == b+1) {
11         u = u.next;
12         r++;
13     }
14     if (r == b) { // b blocks each with b+1 elements
15         spread(l.u);
16         u = l.u;
17     }
18     if (u == dummy) { // ran off the end - add new node
19         u = addBefore(u);
20     }
21     while (u != l.u) { // work backwards, shifting elements
22         u.d.add(0, u.prev.d.remove(u.prev.d.size()-1));
23         u = u.prev;
24     }
25     u.d.add(l.j, x);
26     n++;
27 }

```

The running time of the `add(i, x)` operation depends on which of the three cases above occurs. Cases 1 and 2 involve examining and shifting elements through at most b blocks and take $O(b)$ time. Case 3 involves calling the `spread(u)` method, which moves $b(b+1)$ elements and takes $O(b^2)$ time. If we ignore the cost of Case 3 (which we will account for later with amortization) this means that the total running time to locate i and perform the insertion of x is $O(b + \min\{i, n - i\}/b)$.

4.3.4 Removing an Element

Removing an element from an `SEList` is similar to adding an element. We first locate the node u that contains the element with index i . Now, we have to be prepared for the case where we cannot remove an element from u without causing u 's block to become smaller than $b - 1$.

Again, let $u_0, u_1, u_2, \dots$ denote $u, u.next, u.next.next$, and so on. We examine $u_0, u_1, u_2, \dots$ in order to look for a node from which we can borrow an element to make the size of u_0 's block at least $b - 1$. There are three cases to consider (see Figure 4.3.2):

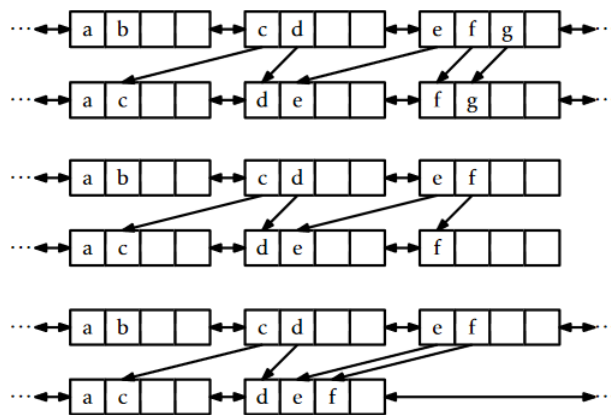


Figure 4.3.2: The three cases that occur during the removal of an item x in the interior of an SEList. (This SEList has block size b .)

1. We quickly (in $r + 1 \leq b$ steps) find a node whose block contains more than $b - 1$ elements. In this case, we perform r shifts of an element from one block into the previous one, so that the extra element in u_r becomes an extra element in u_0 . We can then remove the appropriate element from u_0 's block.
2. We quickly (in $r + 1 \leq b$ steps) run off the end of the list of blocks. In this case, u_r is the last block, and there is no need for u_r 's block to contain at least $b - 1$ elements. Therefore, we proceed as above, borrowing an element from u_r to make an extra element in u_0 . If this causes u_r 's block to become empty, then we remove it.
3. After b steps, we do not find any block containing more than $b - 1$ elements. In this case, $u_0, \dots, u_{b-1}$ is a sequence of b blocks that each contain $b - 1$ elements. We gather these $b(b - 1)$ elements into $u_0, \dots, u_{b-2}$ so that each of these $b - 1$ blocks contains exactly b elements and we remove u_{b-1} , which is now empty. Now u_0 's block contains b elements and we can then remove the appropriate element from it.

```

01 T remove(int i) {
02     if (i < 0 || i > n - 1) throw new IndexOutOfBoundsException();
03     Location l = getLocation(i);
04     T y = l.u.d.get(l.j);
05     Node u = l.u;
06     int r = 0;
07     while (r < b && u != dummy && u.d.size() == b-1) {
08         u = u.next;
09         r++;
10     }
11     if (r == b) { // b blocks each with b-1 elements
12         gather(l.u);
13     }
14     u = l.u;
15     u.d.remove(l.j);
16     while (u.d.size() < b-1 && u.next != dummy) {
17         u.d.add(u.next.d.remove(0));
18         u = u.next;
19     }
20     if (u.d.isEmpty()) remove(u);
21     n--;
22     return y;
23 }

```

Like the $\text{add}(i, x)$ operation, the running time of the $\text{remove}(i)$ operation is $O(b + \min\{i, n - i\}/b)$ if we ignore the cost of the $\text{gather}(u)$ method that occurs in Case 3.

4.3.5 Amortized Analysis of Spreading and Gathering

Next, we consider the cost of the `gather(u)` and `spread(u)` methods that may be executed by the `add(i, x)` and `remove(i)` methods. For the sake of completeness, here they are:

```

01 void spread(Node u) {
02     Node w = u;
03     for (int j = 0; j < b; j++) {
04         w = w.next;
05     }
06     w = addBefore(w);
07     while (w != u) {
08         while (w.d.size() < b)
09             w.d.add(0, w.prev.d.remove(w.prev.d.size() - 1));
10         w = w.prev;
11     }
12 }

```

```

1 void gather(Node u) {
2     Node w = u;
3     for (int j = 0; j < b - 1; j++) {
4         while (w.d.size() < b)
5             w.d.add(w.next.d.remove(0));
6         w = w.next;
7     }
8     remove(w);
9 }

```

The running time of each of these methods is dominated by the two nested loops. Both the inner and outer loops execute at most $b + 1$ times, so the total running time of each of these methods is $O((b + 1)^2) = O(b^2)$. However, the following lemma shows that these methods execute on at most one out of every b calls to `add(i, x)` or `remove(i)`.

Lemma 4.3.1.

If an empty SEList is created and any sequence of $m \geq 1$ calls to `add(i, x)` and `remove(i)` is performed, then the total time spent during all calls to `spread()` and `gather()` is $O(bm)$.

Proof. We will use the potential method of amortized analysis. We say that a node u is fragile if u 's block does not contain b elements (so that u is either the last node, or contains $b - 1$ or $b + 1$ elements). Any node whose block contains b elements is rugged. Define the potential of an SEList as the number of fragile nodes it contains. We will consider only the `add(i, x)` operation and its relation to the number of calls to `spread(u)`. The analysis of `remove(i)` and `gather(u)` is identical.

Notice that, if Case 1 occurs during the `add(i, x)` method, then only one node, u_r , has the size of its block changed. Therefore, at most one node, namely u_r , goes from being rugged to being fragile. If Case 2 occurs, then a new node is created, and this node is fragile, but no other node changes size, so the number of fragile nodes increases by one. Thus, in either Case 1 or Case 2 the potential of the SEList increases by at most one.

Finally, if Case 3 occurs, it is because $u_0, \dots, u_{b-1}$ are all fragile nodes. Then `spread( $u_0$ )` is called and these b fragile nodes are replaced with $b + 1$ rugged nodes. Finally, x is added to u_0 's block, making u_0 fragile. In total the potential decreases by $b - 1$.

In summary, the potential starts at 0 (there are no nodes in the list). Each time Case 1 or Case 2 occurs, the potential increases by at most 1. Each time Case 3 occurs, the potential decreases by $b - 1$. The potential (which counts the number of fragile nodes) is never less than 0. We conclude that, for every occurrence of Case 3, there are at least $b - 1$ occurrences of Case 1 or Case 2. Thus, for every call to `spread(u)` there are at least b calls to `add(i, x)`. This completes the proof. $\square$

4.3.6 Summary

The following theorem summarizes the performance of the `SEList` data structure:

Theorem 4.3.1.

An `SEList` implements the `List` interface. Ignoring the cost of calls to `spread(u)` and `gather(u)`, an `SEList` with block size b supports the operations

- `get(i)` and `set(i, x)` in $O(1 + \min\{i, n - i\}/b)$ time per operation; and
- `add(i, x)` and `remove(i)` in $O(b + \min\{i, n - i\}/b)$ time per operation.

Furthermore, beginning with an empty `SEList`, any sequence of m `add(i, x)` and `remove(i)` operations results in a total of $O(bm)$ time spent during all calls to `spread(u)` and `gather(u)`.

The space (measured in words)¹ used by an `SEList` that stores n elements is $n + O(b + n/b)$.

The `SEList` is a trade-off between an `ArrayList` and a `DLList` where the relative mix of these two structures depends on the block size b . At the extreme $b = 2$, each `SEList` node stores at most three values, which is not much different than a `DLList`. At the other extreme, $b > n$, all the elements are stored in a single array, just like in an `ArrayList`. In between these two extremes lies a trade-off between the time it takes to add or remove a list item and the time it takes to locate a particular list item.

Footnotes

¹See Open Data Structures, Section 1.4 for a discussion of how memory is measured.

4.3: `SEList` - A Space-Efficient Linked List is shared under a [CC BY](#) license and was authored, remixed, and/or curated by LibreTexts.

4.4: Discussion and Exercises

Both singly-linked and doubly-linked lists are established techniques, having been used in programs for over 40 years. They are discussed, for example, by Knuth [46, Sections 2.2.3-2.2.5]. Even the `SEList` data structure seems to be a well-known data structures exercise. The `SEList` is sometimes referred to as an unrolled linked list [69].

Another way to save space in a doubly-linked list is to use so-called XOR-lists. In an XOR-list, each node, `u`, contains only one pointer, called `u.nextprev`, that holds the bitwise exclusive-or of `u.prev` and `u.next`. The list itself needs to store two pointers, one to the dummy node and one to `dummy.next` (the first node, or dummy if the list is empty). This technique uses the fact that, if we have pointers to `u` and `u.prev`, then we can extract `u.next` using the formula

$$u.next = u.prev \oplus u.nextprev .$$

(Here $\oplus$ computes the bitwise exclusive-or of its two arguments.) This technique complicates the code a little and is not possible in some languages, like Java and Python, that have garbage collection but gives a doubly-linked list implementation that requires only one pointer per node. See Sinha's magazine article [70] for a detailed discussion of XOR-lists.

Exercise 4.4.1

Why is it not possible to use a dummy node in an `SList` to avoid all the special cases that occur in the operations `push(x)`, `pop()`, `add(x)`, and `remove()`?

Exercise 4.4.2

Design and implement an `SList` method, `secondLast()`, that returns the second-last element of an `SList`. Do this without using the member variable, `n`, that keeps track of the size of the list.

Exercise 4.4.3

Implement the `List` operations `get(i)`, `set(i, x)`, `add(i, x)` and `remove(i)` on an `SList`. Each of these operations should run in $O(1 + i)$ time.

Exercise 4.4.4

Design and implement an `SList` method, `reverse()` that reverses the order of elements in an `SList`. This method should run in $O(n)$ time, should not use recursion, should not use any secondary data structures, and should not create any new nodes.

Exercise 4.4.5

Design and implement `SList` and `DList` methods called `checkSize()`. These methods walk through the list and count the number of nodes to see if this matches the value, `n`, stored in the list. These methods return nothing, but throw an exception if the size they compute does not match the value of `n`.

Exercise 4.4.6

Try to recreate the code for the `addBefore(w)` operation that creates a node, `u`, and adds it in a `DList` just before the node `w`. Do not refer to this chapter. Even if your code does not exactly match the code given in this book it may still be correct. Test it and see if it works.

The next few exercises involve performing manipulations on `DLists`. You should complete them without allocating any new nodes or temporary arrays. They can all be done only by changing the `prev` and `next` values of existing nodes.

Exercise 4.4.7

Write a `DLList` method `isPalindrome()` that returns `true` if the list is a palindrome, i.e., the element at position i is equal to the element at position $n - i - 1$ for all $i \in \{0, \dots, n - 1\}$. Your code should run in $O(n)$ time.

Exercise 4.4.8

Implement a method `rotate(r)` that "rotates" a `DLList` so that list item i becomes list item $(i + r) \bmod n$. This method should run in $O(1 + \min\{r, n - r\})$ time and should not modify any nodes in the list.

Exercise 4.4.9

Write a method, `truncate(i)`, that truncates a `DLList` at position i . After executing this method, the size of the list will be i and it should contain only the elements at indices $0, \dots, i - 1$. The return value is another `DLList` that contains the elements at indices $i, \dots, n - 1$. This method should run in $O(\min\{i, n - i\})$ time.

Exercise 4.4.10

Write a `DLList` method, `absorb(12)`, that takes as an argument a `DLList`, `12`, empties it and appends its contents, in order, to the receiver. For example, if `11` contains a, b, c and `12` contains d, e, f , then after calling `11.absorb(12)`, `11` will contain a, b, c, d, e, f and `12` will be empty.

Exercise 4.4.11

Write a method `deal()` that removes all the elements with odd-numbered indices from a `DLList` and return a `DLList` containing these elements. For example, if `11`, contains the elements a, b, c, d, e, f , then after calling `11.deal()`, `11` should contain a, c, e and a list containing b, d, f should be returned.

Exercise 4.4.12

Write a method, `reverse()`, that reverses the order of elements in a `DLList`.

Exercise 4.4.13

This exercise walks you through an implementation of the merge-sort algorithm for sorting a `DLList`, as discussed in Section 11.1.1. In your implementation, perform comparisons between elements using the `compareTo(x)` method so that the resulting implementation can sort any `DLList` containing elements that implement the `Comparable` interface.

1. Write a `DLList` method called `takeFirst(12)`. This method takes the first node from `12` and appends it to the the receiving list. This is equivalent to `add(size(), 12.remove(0))`, except that it should not create a new node.
2. Write a `DLList` static method, `merge(11, 12)`, that takes two sorted lists `11` and `12`, merges them, and returns a new sorted list containing the result. This causes `11` and `12` to be emptied in the proces. For example, if `11` contains a, c, d and `12` contains b, e, f , then this method returns a new list containing a, b, c, d, e, f .
3. Write a `DLList` method `sort()` that sorts the elements contained in the list using the merge sort algorithm. This recursive algorithm works in the following way:
 - a. If the list contains 0 or 1 elements then there is nothing to do. Otherwise,
 - b. Using the `truncate(size()/2)` method, split the list into two lists of approximately equal length, `11` and `12`;
 - c. Recursively sort `11`;
 - d. Recursively sort `12`; and, finally,
 - e. Merge `11` and `12` into a single sorted list.

The next few exercises are more advanced and require a clear understanding of what happens to the minimum value stored in a Stack or Queue as items are added and removed.

Exercise 4.4.14

Design and implement a `MinStack` data structure that can store comparable elements and supports the stack operations `push(x)`, `pop()`, and `size()`, as well as the `min()` operation, which returns the minimum value currently stored in the data structure. All operations should run in constant time.

Exercise 4.4.15

Design and implement a `MinQueue` data structure that can store comparable elements and supports the queue operations `add(x)`, `remove()`, and `size()`, as well as the `min()` operation, which returns the minimum value currently stored in the data structure. All operations should run in constant amortized time.

Exercise 4.4.16

Design and implement a `MinDeque` data structure that can store comparable elements and supports all the deque operations `addFirst(x)`, `addLast(x)`, `removeFirst()`, `removeLast()` and `size()`, and the `min()` operation, which returns the minimum value currently stored in the data structure. All operations should run in constant amortized time.

The next exercises are designed to test the reader's understanding of the implementation and analysis of the space-efficient `SEList`:

Exercise 4.4.17

Prove that, if an `SEList` is used like a `Stack` (so that the only modifications to the `SEList` are done using `push(x)  $\equiv$  add(size(), x)` and `pop()  $\equiv$  remove(size() - 1)`), then these operations run in constant amortized time, independent of the value of `b`.

Exercise 4.4.18

Design and implement of a version of an `SEList` that supports all the `Deque` operations in constant amortized time per operation, independent of the value of `b`.

Exercise 4.4.19

Explain how to use the bitwise exclusive-or operator, $\wedge$, to swap the values of two `int` variables without using a third variable.

4.4: Discussion and Exercises is shared under a [CC BY](#) license and was authored, remixed, and/or curated by LibreTexts.

- 3.4: Discussion and Exercises by Pat Morin is licensed [CC BY 4.0](#).

CHAPTER OVERVIEW

5: Trees

A **tree** is a non-empty set with an element that is designated as the root of the tree while the remaining elements are partitioned into non-empty sets each of which is a subtree of the root.

Tree nodes have many useful properties. The **depth** of a node is the length of the path (or the number of edges) from the root to that node. The **height** of a node is the longest path from that node to its leaves. The height of a tree is the height of the root. A **leaf node** has no children—its only path is up to its parent.

See the [axiomatic development of trees](#) and its consequences for more information.

Types of trees:

Binary: Each node has zero, one, or two children. This assertion makes many tree operations simple and efficient.

Binary Search: A binary tree where any left child node has a value less than its parent node and any right child node has a value greater than or equal to that of its parent node.

[5.1: Traversal](#)

[5.2: Sample implementations for Tree Traversal](#)

[5.3: Examples of Tree Traversals](#)

[5.4: Balancing](#)

[5.5: Binary Search Trees](#)

[5.6: Example extract of Java code for binary tree delete operation](#)

[5.7: Red-Black trees](#)

[5.8: B Trees](#)

[5.9: Treaps](#)

[5.10: References](#)

5: Trees is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

5.1: Traversal

Many problems require we visit the nodes of a tree in a systematic way: tasks such as counting how many nodes exist or finding the maximum element. Three different methods are possible for binary trees: *preorder*, *postorder*, and *in-order*, which all do the same three things: recursively traverse both the left and right subtrees and visit the current node. The difference is when the algorithm visits the current node:

- **preorder**: Current node, left subtree, right subtree (DLR)
- **postorder**: Left subtree, right subtree, current node (LRD)
- **in-order**: Left subtree, current node, right subtree (LDR)
- **levelorder**: Level by level, from left to right, starting from the root node.

Note

Visit means performing some operation involving the current node of a tree, like incrementing a counter or checking if the value of the current node is greater than any other recorded.

5.1: Traversal is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- **6.1: Traversal** by Wikibooks - Data Structures is licensed [CC BY-SA 4.0](#).

5.2: Sample implementations for Tree Traversal

```
preorder(node)
  visit(node)
  if node.left ≠ null then preorder(node.left)
  if node.right ≠ null then preorder(node.right)
```

```
inorder(node)
  if node.left ≠ null then inorder(node.left)
  visit(node)
  if node.right ≠ null then inorder(node.right)
```

```
postorder(node)
  if node.left ≠ null then postorder(node.left)
  if node.right ≠ null then postorder(node.right)
  visit(node)
```

```
levelorder(root)
  queue<node> q
  q.push(root)
  while not q.empty do
    node = q.pop
    visit(node)
    if node.left ≠ null then q.push(node.left)
    if node.right ≠ null then q.push(node.right)
```

For an algorithm that is less taxing on the stack, see Threaded Trees.

5.2: Sample implementations for Tree Traversal is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- [6.2: Sample implementations for Tree Traversal](#) by Wikibooks - Data Structures is licensed [CC BY-SA 4.0](#).

5.3: Examples of Tree Traversals

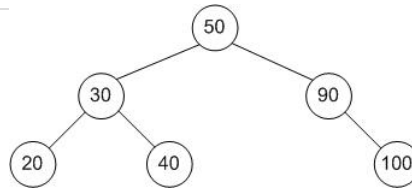


Figure 5.3.1 (Public Domain; via Wikimedia Commons)

preorder: 50, 30, 20, 40, 90, 100

inorder: 20, 30, 40, 50, 90, 100

postorder: 20, 40, 30, 100, 90, 50

5.3: Examples of Tree Traversals is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

5.4: Balancing

When entries that are already *sorted* are stored in a tree, all new records will go the same route, and the tree will look more like a list (such a tree is called a degenerate tree). Therefore the tree needs balancing routines, making sure that under all branches are an equal number of records. This will keep searching in the tree at optimal speed. Specifically, if a tree with n nodes is a degenerate tree, the longest path through the tree will be n nodes; if it is a balanced tree, the longest path will be $\log n$ nodes.

Algorithms/Left rotation: This shows how balancing is applied to establish a priority heap invariant in a Treap, a data structure which has the queueing performance of a heap, and the key lookup performance of a tree. A balancing operation can change the tree structure while maintaining another order, which is binary tree sort order. The binary tree order is left to right, with left nodes' keys less than right nodes' keys, whereas the priority order is up and down, with higher nodes' priorities greater than lower nodes' priorities. Alternatively, the priority can be viewed as another ordering key, except that finding a specific key is more involved.

The balancing operation can move nodes up and down a tree without affecting the left right ordering.

- **AVL:** A balanced binary search tree according to the following specification: the heights of the two child subtrees of any node differ by at most one.
- **Red-Black Tree:** A balanced binary search tree using a balancing algorithm based on colors assigned to a node, and the colors of nearby nodes.
- **AA Tree:** A balanced tree, in fact a more restrictive variation of a red-black tree.

5.4: Balancing is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- **6.4: Balancing** by [Wikibooks - Data Structures](#) is licensed [CC BY-SA 4.0](#).

5.5: Binary Search Trees

A typical binary search tree looks like this:

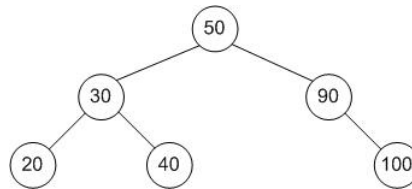


Figure 5.5.1 (Public Domain; via Wikimedia Commons)

Terms

Node: Any item that is stored in the tree.

Root: The top item in the tree. (50 in the tree above)

Child: Node(s) under the current node. (20 and 40 are children of 30 in the tree above)

Parent: The node directly above the current node. (90 is the parent of 100 in the tree above)

Leaf: A node which has no children. (20 is a leaf in the tree above)

Searching through a binary search tree

To search for an item in a binary tree:

1. Start at the root node
2. If the item that you are searching for is less than the root node, move to the left child of the root node, if the item that you are searching for is more than the root node, move to the right child of the root node and if it is equal to the root node, then you have found the item that you are looking for.
3. Now check to see if the item that you are searching for is equal to, less than or more than the new node that you are on. Again if the item that you are searching for is less than the current node, move to the left child, and if the item that you are searching for is greater than the current node, move to the right child.
4. Repeat this process until you find the item that you are looking for or until the node does not have a child on the correct branch, in which case the tree doesn't contain the item which you are looking for.

Example

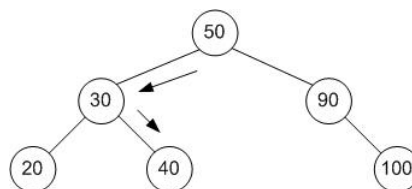


Figure 5.5.2 (Public Domain; via Wikimedia Commons)

For example, to find the node 40...

1. The root node is 50, which is greater than 40, so you go to 50's left child.
2. 50's left child is 30, which is less than 40, so you next go to 30's right child.
3. 30's right child is 40, so you have found the item that you are looking for :)

.....

Adding an item to a binary search tree

1. To add an item, you first must search through the tree to find the position that you should put it in. You do this following the steps above.
2. When you reach a node which doesn't contain a child on the correct branch, add the new node there.

For example, to add the node 25...

1. The root node is 50, which is greater than 25, so you go to 50's left child.
2. 50's left child is 30, which is greater than 25, so you go to 30's left child.
3. 30's left child is 20, which is less than 25, so you go to 20's right child.
4. 20's right child doesn't exist, so you add 25 there :)

Deleting an item from a binary search tree

It is assumed that you have already found the node that you want to delete, using the search technique described above.

Case 1: The node you want to delete is a leaf

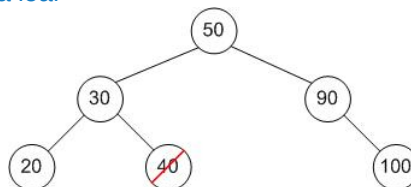


Figure 5.5.3 (Public Domain; via Wikimedia Commons)

For example, to delete 40...

- Simply delete the node!

Case 2: The node you want to delete has one child

1. Directly connect the child of the node that you want to delete, to the parent of the node that you want to delete.

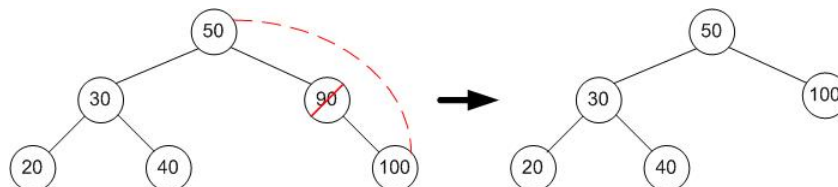


Figure 5.5.4 (Public Domain; via Wikimedia Commons)

For example, to delete 90...

- Delete 90, then make 100 the child node of 50.

Case 3: The node you want to delete has two children

One *non-standard* way, is to rotate the node into a chosen subtree, and attempt to delete the key again from that subtree, recursively, until Case 1 or Case 2 occurs. This could unbalance a tree, so randomly choosing whether to right or left rotate may help.

The *standard* way is to pick either the left or right child, say the right, then get the right's leftmost descendant by following left, starting from the right child, until the next left is null. Then remove this leftmost descendant of the right child, replacing it with its right sub-tree (it has a left child of null). Then use the contents of this former leftmost descendant of the right child, as replacement for the key and value of the node being deleted, so that its values now are in the deleted node, the parent of the right child. This still maintains the key ordering for all nodes. Example java code is below in the treap example code.

The following examples use the standard algorithm, that is, the successor is the left-most node in the right subtree of the node to be deleted.

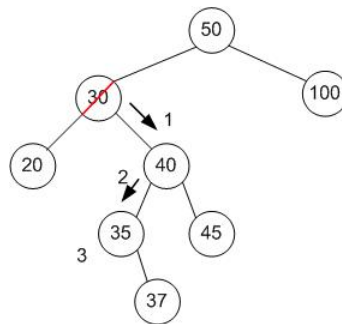


Figure 5.5.5 (Public Domain; by Linuxdude via Wikimedia Commons)

For example, to delete 30

1. The right node of the node which is being deleted is 40.
2. (From now on, we continually go to the left node until there isn't another one...) The first left node of 40, is 35.
3. 35 has no left node, therefore 35 is the successor!
4. 35 replaces 30, at the original right node, and the node with 35 is deleted, replacing it with the right sub-tree, which has the root node 37.

Case 1 of two-children case: The successor is the right child of the node being deleted

1. Directly move the child to the right of the node being deleted into the position of the node being deleted.
2. As the new node has no left children, you can connect the deleted node's left subtree's root as it's left child.

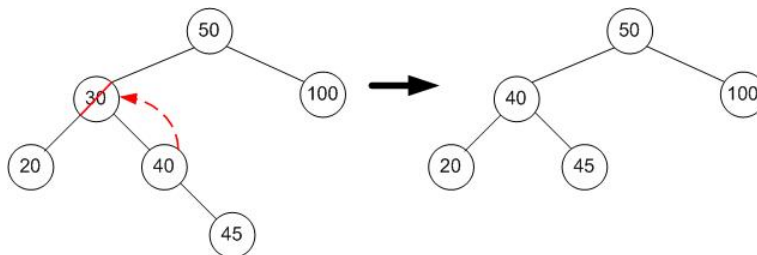


Figure 5.5.6 (Public Domain; by Linuxdude via Wikimedia Commons)

For example, to delete 30

1. replace the contents to be deleted (30), with the successor's contents (40).
2. delete the successor node (contents 40), replacing it with its right subtree (head contents 45).

Case 2 of two-children case: The successor isn't the right child of the node being deleted

This is best shown with an example

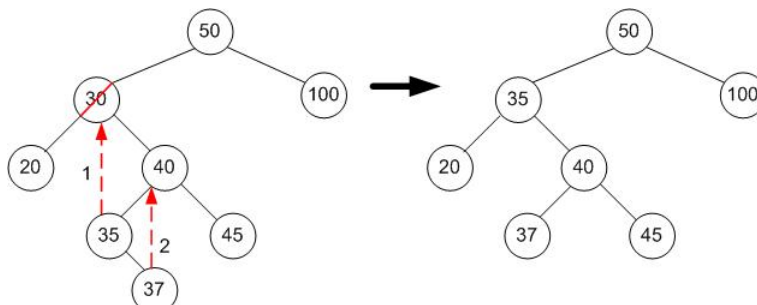


Figure 5.5.7 (Public Domain; via Wikimedia Commons)

To delete 30...

1. Replace the contents to be deleted (30) with the successor's contents (35).
2. replace the successor (35) with its right subtree (37). There is no left subtree because the successor is leftmost.

5.5: Binary Search Trees is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- 6.5: Binary Search Trees by Wikibooks - Data Structures is licensed [CC BY-SA 4.0](#).

5.6: Example extract of Java code for binary tree delete operation

```
01 private Treap1<K, V>.TreapNode deleteNode(K k, Treap1<K, V>.TreapNode node,  
Deleted del) {  
02  
03     if (node == null) {  
04         return null;  
05     } else if (k.compareTo(node.k) < 0) {  
06         node.left = (k, node.left, del) ;  
07     } else if (k.compareTo(node.k) > 0) {  
08         node.right = deleteNode(k, node.right, del);  
09  
10         // k.compareTo(node.k) == 0  
11     } else if ( node.left == null ) {  
12         del.success = true;  
13         return node.right;  
14     } else if ( node.right == null) {  
15         del.success = true;  
16         return node.left;  
17     } else if (node.left != null && node.right != null){  
18         /*  
19         // non-standard method,  
20         // left rotate and all delete on left subtree  
21  
22         TreapNode tmp = node.right;  
23         node.right = node.right.left;  
24         tmp.left = node;  
25         node = tmp;  
26         node.left = deleteNode(k , node.left, del);  
27  
28         */  
29  
30         // more standard method ? doesn't disturb tree structure as much  
31         // find leftmost descendant of the right child node and replace  
contents  
32  
33         TreapNode n2 = node.right;  
34         TreapNode previous2 = null;  
35         while (n2.left != null) {  
36             previous2 = n2;  
37             n2 = n2.left;  
38         }  
39  
40         if (previous2 != null) {  
41             previous2.left = n2.right;  
42             //n2 has no parent link, orphaned  
43         } else {  
44             node.right = n2.right;  
45             //n2 has no parent link, orphaned  
46         }  
47  
48         node.k = n2.k;  
49         node.val = n2.val;  
50         del.success = true;  
51         // once n2 out of scope, the orphaned node at n2 will be garbage  
collected,  
52     }  
53  
54     return node;  
55 }
```

5.6: Example extract of Java code for binary tree delete operation is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

5.7: Red-Black trees

A **red-black tree** is a self-balancing tree structure that applies a color to each of its nodes. The structure of a red-black tree must adhere to a set of rules which dictate how nodes of a certain color can be arranged. The application of these rules is performed when the tree is modified in some way, causing the rotation and recolouring of certain nodes when a new node is inserted or an old node is deleted. This keeps the red-black tree balanced, guaranteeing a search complexity of $O(\log n)$.

The rules that a red-black tree must adhere to are as follows:

1. Each node must be either red or black.
2. The root is always black.
3. All leaves within the tree are black (leaves do not contain data and can be modelled as *null* or *nil* references in most programming languages).
4. Every red node must have two black child nodes.
5. Every path from a given node to any of its descendant leaves must contain the same number of black nodes.

A red-black tree can be modelled as 2-3-4 tree, which is a sub-class of B tree (below). A black node with one red node can be seen as linked together as a 3-node, and a black node with 2 red child nodes can be seen as a 4-node.

4-nodes are **split**, producing a two node, and the middle node made red, which turns a parent of the middle node which has no red child from a 2-node to a 3-node, and turns a parent with one red child into a 4-node (but this doesn't occur with always left red nodes).

An in-line arrangement of two red nodes, is **rotated** into a parent with two red children, a 4-node, which is later **split**, as described before.



An optimization mentioned by Sedgwick is that all right inserted red nodes are left rotated to become left red nodes, so that only inline left red nodes ever have to be rotated right before splitting. AA-trees (above) by Arne Anderson, described in a paper in 1993, seem an earlier exposition of the simplification, however he suggested right-leaning 'red marking' instead of left leaning, as suggested by Sedgwick, but AA trees seem to have precedence over left leaning red black trees. It would be quite a shock if the Linux CFS scheduler was described in the future as 'AA based'.

In summary, red-black trees are a way of detecting two insertions into the same side, and levelling out the tree before things get worse. Two left sided insertions will be rotated, and the two right sided insertions, would look like two left sided insertions after left rotation to remove right leaning red nodes. Two balanced insertions for the same parent could result in a 4-node split without rotation, so the question arises as to whether a red black tree could be attacked with serial insertions of one sided triads of a $P < b$, and then the next triad's $P' < a$.

Python illustrative code follows

```
001 RED = 1
002 BLACK = 0
003 class Node:
004     def __init__(self, k, v):
005         # all newly inserted node's are RED
006         self.color = RED
007         self.k = k
008         self.v = v
009         self.left = None
010         self.right = None
011
012 class RBTree:
013     def __init__(self):
014         self.root = None
015
016     def insert(self, k, v) :
017         self.root = self._insert(self.root, k,v)
018
019     def _insert(self, n , k, v):
020         if n is None:
021             return Node(k,v)
022         if k < n.k :
023             n.left = self._insert(n.left, k , v)
024         elif k > n.k :
025             n.right = self._insert(n.right, k, v)
026         if n.right.color is RED:
027             #always on the left red's
028             #left rotate
029             tmp = n.right
030             n.right = tmp.left
031             tmp.left = n
032             n = tmp
033
034             #color rotation is actually a swap
035             tmpcolor = n.color
036             n.color = n.left.color
037             n.left.color = tmpcolor
038
039         if n.left <> None and n.left.left <> None and n.left.left.color == RED and
n.left.color == RED:
040             # right rotate in-line reds
041             print "right rotate"
042             tmp = n.left
043             n.left = tmp.right
044             tmp.right = n
045             n = tmp
046
047             #color rotation is actually a swap
048             tmpcolor = n.color
049             n.color = n.right.color
050             n.right.color = tmpcolor
051
052             if n.left <> None: print n.left.color, n.color, n.right.color
053
054             #no need to test, because after right rotation, will need to split 3-node ,
as right rotation has
055             #brought red left grandchild to become left red child, and left red child
is now red right child
056             #so there are two red children.
057
058             #if n.left <> None and n.right <> None and n.left.color == RED and
```

```
    n.right.color == RED:
059         print "split"
060         n.color = RED
061         n.left.color = BLACK
062         n.right.color = BLACK
063
064         return n
065
066     def find(self, k):
067         return self._find_rb(k, self.root)
068
069     def _find_rb(self, k, n):
070         if n is None:
071             return None
072         if k < n.k:
073             return self._find_rb(k, n.left)
074         if k > n.k:
075             return self._find_rb(k, n.right)
076         return n.v
077
078     def inorder(self):
079         self.inorder_visit(self.root, "0")
080
081     def inorder_visit(self, node, label=""):
082         if node is None: return
083         self.inorder_visit(node.left, label+"/L")
084         print label, "val=", node.v
085         self.inorder_visit(node.right, label+"/R")
086
087
088     def test1(N):
089         t = RBTree()
090         for i in xrange(0,N):
091             t.insert(i,i)
092
093         l = []
094         t.inorder()
095         for i in xrange(0,N):
096             x = t.find(i)
097             if x <> None:
098                 l.append((x, i) )
099         print "found", len(l)
100
101 if __name__ == "__main__":
102     import random
103     test1(100000)
104     test1(1000)
105     test1(100)
106     test1(10)
```

5.7: Red-Black trees is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- [6.7: Red-Black Trees](#) by Wikibooks - Data Structures is licensed [CC BY-SA 4.0](#).

5.8: B Trees

Summary

- Whereas binary trees have nodes that have two children, with the left child and all of its descendants less than the "value" of the node, and the right child and all of its children more than the "value" of the node, a B-tree is a generalization of this.
- The generalization is that instead of one value, the node has a list of values, and the list is of size n ($n > 2$). n is chosen to optimize storage, so that a node corresponds in size to a block for instance. This is in the days before ssd drives, but searching binary nodes stored on ssd ram would still be slower than searching ssd ram for a block of values, loading into normal ram and cpu cache, and searching the loaded list.
- At the start of the list, the left child of the first element of the list has a value less than the first element, and so do all its children. To the right of the first element, is a child which has values more than the first element's value, as do all of its children, but also less than the value of the second element. Induction can be used, and this holds so for the child between element 1 and 2, 2 and 3, ... so on until $n-1$ and n th node.
- To insert into a non-full B tree node, is to do a insertion into a sorted list.
- In a B+ tree, insertions can only be done in leaf nodes, and non-leaf nodes hold copies of a demarcating value between adjacent child nodes e.g. the left most value of an element's right child's list of nodes.
- Whenever a list becomes full e.g. there are n nodes, the node is "split", and this means making two new nodes, and passing the demarcating value upto the parent.

B Trees were described originally as generalizations of binary search trees, where a binary tree is a 2-node B-Tree, the 2 standing for two children, with $2-1 = 1$ key separating the 2 children. Hence a 3-node has 2 values separating 3 children, and a N node has N children separated by $N-1$ keys.

A classical B-Tree can have N -node internal nodes, and empty 2-nodes as leaf nodes, or more conveniently, the children can either be a value or a pointer to the next N -node, so it is a union.

The main idea with B-trees is that one starts with a root N -node, which is able to hold $N-1$ entries, but on the N th entry, the number of keys for the node is exhausted, and the node can be split into two half sized $N/2$ sized N nodes, separated by a single key K , which is equal to the right node's leftmost key, so any entry with key K_2 equal or greater than K goes in the right node, and anything less than K goes in the left. When the root node is split, a new root node is created with one key, and a left child and a right child. Since there are N children but only $N-1$ entries, the leftmost child is stored as a separate pointer. If the leftmost pointer splits, then the left half becomes the new leftmost pointer, and the right half and separating key is inserted into the front of the entries.

An alternative is the B+ tree which is the most commonly used in database systems, because only values are stored in leaf nodes, whereas internal nodes only store keys and pointers to other nodes, putting a limit on the size of the datum value as the size of a pointer. This often allows internal nodes with more entries able to fit a certain block size, e.g. 4K is a common physical disc block size. Hence, if a B+ tree internal node is aligned to a physical disc block, then the main rate limiting factor of reading a block of a large index from disc because it isn't cached in a memory list of blocks is reduced to one block read.

A B+ tree has bigger internal nodes, so is wider and shorter in theory than an equivalent B tree which must fit all nodes within a given physical block size, hence overall it is a faster index due to greater fan out and less height to reach keys on average.

Apparently, this fan out is so important, compression can also be applied to the blocks to increase the number of entries fitting within a given underlying layer's block size (the underlying layer is often a filesystem block).

Most database systems use the B+ tree algorithm, including postgresql, mysql, derbydb, firebird, many Xbase index types, etc.

Many filesystems also use a B+ tree to manage their block layout (e.g. xfs, NTFS, etc).

Transwiki has a java implementation of a B+ Tree which uses traditional arrays as key list and value list.

Below is an example of a B Tree with test driver, and a B+ tree with a test driver. The memory / disc management is not included, but a usable hacked example can be found at [Hashing Memory Checking Example](#).

This B+ tree implementation was written out of the B Tree, and the difference from the transwiki B+ tree is that it tries to use the semantics of SortedMap and SortedSet already present in the standard Java collections library.

Hence , the flat leaf block list of this B+ implementation can't contain blocks that don't contain any data, because the ordering depends on the first key of the entries, so a leaf block needs to be created with its first entry.

A B tree java example

```
001 package btreemap;
002
003 import java.util.ArrayList;
004 import java.util.Collection;
005 import java.util.Collections;
006 import java.util.Comparator;
007 import java.util.List;
008 import java.util.Map;
009 import java.util.Set;
010 import java.util.SortedMap;
011 import java.util.TreeMap;
012
013 /** can't work without setting a comparator */
014 public class BTreeMap<K, V> implements SortedMap<K, V> {
015
016     private static final int NODE_SIZE = 100;
017
018
019
020     @Override
021     public Comparator<? super K> comparator() {
022         // TODO Auto-generated method stub
023
024         return comparator;
025     }
026
027     Comparator< ? super K> defaultComparator = new
028     Comparator< K>() {
029
030         @Override
031         public int compare(K o1, K o2) {
032             // TODO Auto-generated method stub
033             Comparable c1 = (Comparable)o1;
034             Comparable c2 = (Comparable)o2;
035             return c1.compareTo(c2);
036         }
037     };
038     Comparator<? super K> comparator = defaultComparator;
039     BTBlock<K, V> root = new BTBlock<K, V>(NODE_SIZE, comparator);
040     /**
041     *
042     * @param comparator
043     * - this is mandatory for the tree to work
044     */
045     public void setComparator(Comparator<? super K> comparator) {
046         this.comparator = comparator;
047         root = new BTBlock<K, V>(NODE_SIZE, comparator);
048     }
049
050     /**
051     *
052     *
053     * @param <K>
054     * @param <V>
055     * the entry is associated with a right child block.
056     *
057     */
058
059     static class BlockEntry<K, V> {
060         K k;
061         V v;
```

```
062         BTBlock left;
063
064         BlockEntry() {
065
066         }
067
068         BlockEntry(K k, V v) {
069             left = null;
070             this.k = k;
071             this.v = v;
072         }
073     }
074
075     /**
076     *
077     * - this represents the result of splitting a full block into
078     *   a left block, and a right block, and a median key, the right
079     *   block and the median key held in a BlockEntry structure as above.
080     * @param <K>
081     * @param <V>
082     * @param <V>g
083     */
084     static class SplitRootEntry<K, V> {
085         BTBlock<K, V> right;
086         BlockEntry<K, V> entry;
087
088         SplitRootEntry(BlockEntry<K, V> entry, BTBlock<K, V> right) {
089             this.right = right;
090             this.entry = entry;
091         }
092
093         SplitRootEntry() {
094             super();
095         }
096     }
097
098     /**
099     * this is used to return a result of a possible split , during recursive
100     * calling.
101     *
102     *
103     *
104     * @param <K>
105     * @param <V>
106     */
107     static class resultAndSplit<K, V> {
108
109         /**
110         * null , if there is no split.
111         */
112         SplitRootEntry<K, V> splitNode;
113         V v;
114
115         resultAndSplit(V v) {
116             this.v = v;
117         }
118
119         resultAndSplit(SplitRootEntry<K, V> entry, V v) {
120             this.v = v;
121             this.splitNode = entry;
122         }
123     }
124
```

```
125  /**
126   * used to represent the insertion point after searching a block if compare
127   * is zero, then a match was found, and pos is the insertion entry if
128   * compare < 0 and pos == 0 , then the search ended up less than the
129   * leftmost entry else compare > 0 , and the search will be to the
    immediate
130   * right of pos.
131   *
132   *
133   *
134   */
135   static class PosAndCompare {
136       int pos = 0;
137       int compare = 0;
138   }
139
140   static class BTBlock<K, V> {
141       List<BlockEntry<K, V>> entries;
142       BTBlock<K, V> rightBlock = null;
143       private int maxSz = 0;
144
145       Comparator<? super K> comparator;
146
147       Comparator<? super K> comparator() {
148           return comparator;
149       }
150
151       public BTBlock(int size, Comparator<? super K> c) {
152           entries = new ArrayList<BlockEntry<K, V>>();
153           maxSz = size;
154           this.comparator = c;
155       }
156
157       /**
158        * PosAndCompare usage: if compare is zero, then a match was found, and
159        * pos is the insertion entry if compare < 0 and pos == 0 , then the
160        * search ended up less than the leftmost entry else compare > 0 , and
161        * the search will be to the immediate right of pos.
162        *
163        *
164        *
165        */
166       // private void blockSearch(K k, PosAndCompare pc) {
167       // for (int i = 0; i < entries.size(); ++i) {
168       // pc.compare = comparator.compare(k, entries.get(i).k);
169       // if (pc.compare == 0) {
170       // pc.pos = i;
171       // return;
172       // }
173       // if (pc.compare < 0 && i == 0) {
174       // pc.pos = 0;
175       // return;
176       // }
177       //
178       // if (pc.compare < 0) {
179       // pc.pos = i - 1;
180       // pc.compare = 1;
181       // return;
182       // }
183       //
184       // }
185       // pc.pos = entries.size() - 1;
186       // pc.compare = 1;
```

```
187 //
188 // // binary search, it's hard to get it right !
189 // // int left = 0;
190 // // int right = entries.size();
191 // //
192 // // while (left <= right && left < entries.size()) {
193 // // // pc.pos = (right - left) / 2 + left;
194 // // pc.pos = (left + right) / 2;
195 // // pc.compare = comparator().compare(k, entries.get(pc.pos).k);
196 // // if (pc.compare < 0) {
197 // // right = pc.pos - 1;
198 // // } else if (pc.compare > 0) {
199 // // left = pc.pos + 1;
200 // // } else {
201 // // return;
202 // // }
203 // // }
204 // //
205 // // BlockEntry<K, V> e = new BlockEntry<K, V>(k, null);
206 // // pc.pos = Collections.binarySearch(entries, e, cmp);
207 //
208 // }
209
210 Comparator<BlockEntry<K, V>> cmp = new Comparator<BlockEntry<K, V>>() {
211
212     @Override
213     public int compare(BlockEntry<K, V> o1, BlockEntry<K, V> o2) {
214         // TODO Auto-generated method stub
215         return comparator.compare(o1.k, o2.k);
216     }
217 };
218
219 resultAndSplit<K, V> put(K k, V v) {
220     V v2;
221     if (entries.size() == 0) {
222         entries.add(new BlockEntry<K, V>(k, v));
223         return new resultAndSplit<K, V>(v);
224     } else {
225
226         // PosAndCompare pc = new PosAndCompare();
227         BlockEntry<K, V> e = new BlockEntry<K, V>(k, v);
228         int res = Collections.binarySearch(entries, e, cmp);
229         int index = -res - 1;
230         // blockSearch(k, pc);
231
232         // a match
233         if (res >= 0) {
234             v2 = entries.get(res).v;
235             entries.get(res).v = v;
236             return new resultAndSplit<K, V>(v2);
237         }
238
239         // follow leftBlock if search is to left of first entry
240
241         if (index == entries.size() && rightBlock != null) {
242             resultAndSplit<K, V> result = rightBlock.put(k, v);
243             if (result.splitNode != null) {
244                 rightBlock = result.splitNode.right;
245                 entries.add(result.splitNode.entry);
246             }
247         } else if (index == entries.size() && rightBlock == null
```

```

250         && entries.size() == maxSz) {
251             rightBlock = new BTBlock<K, V>(this.maxSz, comparator);
252             resultAndSplit<K, V> result = rightBlock.put(k, v);
253
254         } else if (index < entries.size()
255                 && entries.get(index).left != null) {
256             // follow right block if it exists
257             resultAndSplit<K, V> result = entries.get(index).left.put(
258                 k, v);
259             if (result.splitNode != null) {
260                 entries.get(index).left = result.splitNode.right;
261                 // add to the left
262                 entries.add(index, result.splitNode.entry);
263
264             }
265
266         } else {
267             // add to the left
268             entries.add(index, e);
269
270         }
271
272         // check if overflowed block , split if it has.
273         if (entries.size() > maxSz) {
274             int mid = entries.size() / 2;
275
276             // copy right half to new entries list.
277             List<BlockEntry<K, V>> leftEntries = new
ArrayList<BlockEntry<K, V>>();
278             for (int i = 0; i < mid; ++i) {
279                 leftEntries.add(entries.get(i));
280             }
281
282             BlockEntry<K, V> centreEntry = entries.get(mid);
283
284             BTBlock<K, V> leftBlock = new BTBlock<K, V>(maxSz,
comparator);
285
286             leftBlock.entries = leftEntries;
287
288             // the median entry's left block is the new left block's
289             // leftmost block
290             leftBlock.rightBlock = centreEntry.left;
291             // the new right block becomes the right block
292             centreEntry.left = leftBlock;
293
294             // reduce the old block's entries into its left half of
295             // entries.
296             ArrayList<BlockEntry<K, V>> newEntries = new
ArrayList<BlockEntry<K, V>>();
297             for (int i = mid + 1; i < entries.size(); ++i)
298                 newEntries.add(entries.get(i));
299             this.entries = newEntries;
300             // create a return object, with the reduced old block as
301             the
302             // left block
303             // and the median entry with the new right block attached
304
305             SplitRootEntry<K, V> split = new SplitRootEntry<K, V>(
306                 centreEntry, this);
307
308             // the keyed value didn't exist before , so null
309             return new resultAndSplit<K, V>(split, null);

```

```
310
311     }
312     return new resultAndSplit<K, V>(v);
313 }
314
315 }
316
317 V get(K k) {
318
319     if (entries.size() == 0)
320         return null;
321
322     BlockEntry<K, V> e = new BlockEntry<K, V>(k, null);
323     int res = Collections.binarySearch(entries, e, cmp);
324     int index = -res - 1;
325
326     if (res >= 0) {
327         return entries.get(res).v;
328     }
329
330     if (index == entries.size() && rightBlock != null) {
331         return rightBlock.get(k);
332     } else if (index < entries.size()
333         && entries.get(index).left != null) {
334         return (V) entries.get(index).left.get(k);
335     } else
336         return null;
337 }
338
339 void getRange(SortedMap map, K k1, K k2) {
340     BlockEntry<K, V> e = new BlockEntry<K, V>(k1, null);
341     int res = Collections.binarySearch(entries, e, cmp);
342     int index = -res - 1;
343     BlockEntry<K, V> e2 = new BlockEntry<K, V>(k2, null);
344     int res2 = Collections.binarySearch(entries, e2, cmp);
345     int index2 = -res2 - 1;
346
347     int from = res >= 0 ? res : index;
348     int to = res2 >= 0 ? res2 : index2;
349
350     for (int i = from; i <= to; ++i) {
351
352         if (i < entries.size() && (i > from || res < 0)
353             && entries.get(i).left != null) {
354             entries.get(i).left.getRange(map, k1, k2);
355             // if (pc2.pos == pc.pos)
356             // break;
357         }
358         if (i < to || res2 >= 0)
359             map.put(entries.get(i).k, entries.get(i).v);
360
361         if (i == entries.size() && rightBlock != null) {
362             rightBlock.getRange(map, k1, k2);
363         }
364     }
365 }
366
367 }
368
369 K headKey() {
370     if (rightBlock != null) {
371         return rightBlock.headKey();
372     }
}
```

```
373         return entries.get(0).k;
374     }
375
376     K tailKey() {
377         int i = entries.size() - 1;
378         if (entries.get(i).left != null) {
379             return (K) entries.get(i).left.tailKey();
380         }
381         return entries.get(i).k;
382     }
383
384     void show(int n) {
385         showTabs(n);
386         for (int i = 0; i < entries.size(); ++i) {
387             BlockEntry<K, V> e = entries.get(i);
388             System.err.print("#" + i + ":( " + e.k + ":" + e.v + ") ");
389         }
390         System.err.println();
391         showTabs(n);
392
393         if (rightBlock != null) {
394             System.err.print("Left Block\n");
395             rightBlock.show(n + 1);
396         } else {
397             System.err.println("No Left Block");
398         }
399
400         for (int i = 0; i < entries.size(); ++i) {
401             BlockEntry<K, V> e = entries.get(i);
402             showTabs(n);
403             if (e.left != null) {
404
405                 System.err.println("block right of #" + i);
406                 e.left.show(n + 1);
407             } else {
408                 System.err.println("No block right of #" + i);
409             }
410         }
411         showTabs(n);
412         System.err.println("End of Block Info\n\n");
413     }
414
415     private void showTabs(int n) {
416         // TODO Auto-generated method stub
417         for (int i = 0; i < n; ++i) {
418             System.err.print(" ");
419         }
420     }
421
422 }
423
424 @Override
425 public SortedMap<K, V> subMap(K fromKey, K toKey) {
426     TreeMap<K, V> map = new TreeMap<K, V>();
427     root.getRange(map, fromKey, toKey);
428     return map;
429 }
430
431 @Override
432 public SortedMap<K, V> headMap(K toKey) {
433     // TODO Auto-generated method stub
434     return subMap(root.headKey(), toKey);
435 };
```

```
436
437 @Override
438 public SortedMap<K, V> tailMap(K fromKey) {
439     // TODO Auto-generated method stub
440     return subMap(fromKey, root.tailKey());
441 }
442
443 @Override
444 public K firstKey() {
445     // TODO Auto-generated method stub
446     return root.headKey();
447 }
448
449 @Override
450 public K lastKey() {
451     // TODO Auto-generated method stub
452     return root.tailKey();
453 }
454
455 @Override
456 public int size() {
457     // TODO Auto-generated method stub
458     return 0;
459 }
460
461 @Override
462 public boolean isEmpty() {
463     // TODO Auto-generated method stub
464     return false;
465 }
466
467 @Override
468 public boolean containsKey(Object key) {
469     // TODO Auto-generated method stub
470     return get(key) != null;
471 }
472
473 @Override
474 public boolean containsValue(Object value) {
475     // TODO Auto-generated method stub
476     return false;
477 }
478
479 @Override
480 public V get(Object key) {
481     // TODO Auto-generated method stub
482     return root.get((K) key);
483 }
484
485 @Override
486 public V put(K key, V value) {
487     resultAndSplit<K, V> b = root.put(key, value);
488     if (b.splitNode != null) {
489         root = new BTBlock<K, V>(root.maxSz, root.comparator);
490         root.rightBlock = b.splitNode.right;
491         root.entries.add(b.splitNode.entry);
492     }
493     return b.v;
494 }
495
496 @Override
497 public V remove(Object key) {
498     // TODO Auto-generated method stub
```

```
499         return null;
500     }
501
502     @Override
503     public void putAll(Map<? extends K, ? extends V> m) {
504         // TODO Auto-generated method stub
505
506     }
507
508     @Override
509     public void clear() {
510         // TODO Auto-generated method stub
511
512     }
513
514     @Override
515     public Set<K> keySet() {
516         // TODO Auto-generated method stub
517         return null;
518     }
519
520     @Override
521     public Collection<V> values() {
522         // TODO Auto-generated method stub
523         return null;
524     }
525
526     @Override
527     public Set<java.util.Map.Entry<K, V>> entrySet() {
528         // TODO Auto-generated method stub
529         return null;
530     }
531 }
532
533
534 package btreetmap;
535
536 import java.util.ArrayList;
537 import java.util.Comparator;
538 import java.util.List;
539 import java.util.Random;
540
541 public class TestBtree {
542
543     private static final int N = 50000;
544
545     public static void main(String[] args) {
546         BTreeMap<Integer, Integer> map = new BTreeMap<Integer, Integer>();
547         Random r = new Random();
548
549         ArrayList<Integer> t = new ArrayList<Integer>();
550
551         Comparator<Integer> comparator = new Comparator<Integer>() {
552
553             @Override
554             public int compare(Integer o1, Integer o2) {
555                 // TODO Auto-generated method stub
556                 return o1.intValue() - o2.intValue();
557             }
558
559         };
560         map.setComparator(comparator);
561         List<Integer> testVals = new ArrayList<Integer>();
```

```
562     for (int i = 0 ; i < N ; ++i) {
563         testVals.add(i);
564     }
565     for (int i = 0; i < N; ++i ) {
566         int x = r.nextInt(testVals.size());
567         x = testVals.remove(x);
568         //int x=i;
569         t.add(x);
570         map.put(x, x);
571     }
572
573     System.err.println("output " + N + " vals");
574
575     map.root.show(0);
576
577     for (int i = 0; i < N; ++i) {
578         int x = t.get(i);
579
580         if ( x != map.get(x))
581             System.err.println("Expecting " + x + " got " + map.get(x));
582
583     }
584     System.err.println("Checked " + N + " entries");
585 }
586 }
587 }
```

A B+ tree java example

Experiments include timing the run (e.g. `time java -cp . btreesmap.BPlusTreeTest1`), using an external blocksize + 1 sized leaf block size, so that this is basically the underlying entries TreeMap only, vs, say, 400 internal node size, and 200 external node size. Other

experiments include using a SkipListMap instead of a TreeMap.

```
001 package btreemap;
002
003 import java.util.Collection;
004 import java.util.Comparator;
005 import java.util.Map;
006 import java.util.Set;
007 import java.util.SortedMap;
008 import java.util.SortedSet;
009 import java.util.TreeMap;
010 import java.util.TreeSet;
011 /**
012  * a B+ tree, where leaf blocks contain key-value pairs and
013  * internal blocks have keyed entries pointing to other internal blocks
014  * or leaf blocks whose keys are greater than or equal to the associated key.
015  *
016  * @author syan
017  *
018  * @param <K> key type implements Comparable
019  * @param <V> value type
020  */
021 public class BPlusTreeMap<K, V> implements SortedMap<K, V> {
022
023     private int maxInternalBlockSize;
024     BPlusAnyBlock<K, V> root;
025     private int maxExternalSize;
026
027     BPlusTreeMap(int maxInternalSize, int maxExternalSize) {
028         this.maxInternalBlockSize = maxInternalSize;
029         this.maxExternalSize = maxExternalSize;
030
031     }
032
033     static class SplitOrValue<K, V> {
034         V v;
035         K k;
036         BPlusAnyBlock<K, V> left, right;
037         boolean split;
038
039         public boolean isSplit() {
040             return split;
041         }
042
043         public void setSplit(boolean split) {
044             this.split = split;
045         }
046
047         public SplitOrValue(V value) {
048             v = value;
049             setSplit(false);
050         }
051
052         public SplitOrValue(BPlusAnyBlock<K, V> left2,
053             BPlusAnyBlock<K, V> bPlusBlock) {
054             left = left2;
055             right = bPlusBlock;
056             k = right.firstKey();
057             setSplit(true);
058             // System.err.printf("\n\n** split occured %s**\n\n", bPlusBlock
059             // .getClass().getSimpleName());
060         }
061     }
```

```
062     }
063
064     static abstract class BPlusAnyBlock<K, V> {
065         public abstract SplitOrValue<K, V> put(K k, V v);
066
067         abstract SplitOrValue<K, V> splitBlock();
068
069         public abstract V get(K k);
070
071         public abstract boolean isEmpty();
072
073         public abstract K firstKey();
074
075     }
076
077     SortedSet<BPlusLeafBlock<K, V>> blockList = getLeafBlockSet();
078
079     SortedSet<BPlusLeafBlock<K, V>> getLeafBlockSet() {
080         // return new ConcurrentSkipListSet<BPlusLeafBlock<K, V>>();
081         return new TreeSet<BPlusLeafBlock<K, V>>();
082     }
083
084     static class BPlusLeafBlock<K, V> extends BPlusAnyBlock<K, V> implements
085         Comparable<BPlusLeafBlock<K, V>> {
086         SortedMap<K, V> entries = getEntryCollection();
087
088         static <K, V> SortedMap<K, V> getEntryCollection() {
089             return new TreeMap<K, V>();
090         }
091
092         int maxSize;
093         private BPlusTreeMap<K, V> owner;
094
095         public boolean isEmpty() {
096             return entries.isEmpty();
097         }
098
099         public BPlusLeafBlock(BPlusTreeMap<K, V> bPlusTreeMap,
100             SortedMap<K, V> rightEntries) {
101             this.owner = bPlusTreeMap;
102             maxSize = owner.maxExternalSize;
103             entries = rightEntries;
104         }
105
106         public SplitOrValue<K, V> put(K k, V v) {
107             V v2 = entries.put(k, v);
108
109             if (entries.size() >= maxSize)
110                 return splitBlock();
111             else
112                 return new SplitOrValue<K, V>(v2);
113         }
114
115         public SplitOrValue<K, V> splitBlock() {
116
117             SortedMap<K, V> leftEntries = getEntryCollection();
118             SortedMap<K, V> rightEntries = getEntryCollection();
119
120             int i = 0;
121
122             for (Entry<K, V> e : entries.entrySet()) {
123                 // System.out.println(this.getClass().getSimpleName() +
124
```

```
125         // " split entry " + e.getKey());
126         if (++i <= maxSize / 2)
127             leftEntries.put(e.getKey(), e.getValue());
128         else
129             rightEntries.put(e.getKey(), e.getValue());
130     }
131     BPlusLeafBlock<K, V> right = createBlock(rightEntries);
132     // System.out.println("finished block split");
133     // System.out.println("\nleft block");
134     // for (K ik : leftEntries.keySet()) {
135     //     System.out.print(ik + " ");
136     // }
137     // System.out.println("\nright block");
138     // for (K ik : right.entries.keySet()) {
139     //     System.out.print(ik + " ");
140     // }
141     // System.out.println("\n");
142     this.entries = leftEntries;
143
144     return new SplitOrValue<K, V>(this, right);
145 }
146
147 private BPlusLeafBlock<K, V> createBlock(SortedMap<K, V> rightEntries)
148 {
149     return owner.createLeafBlock(rightEntries);
150 }
151
152 @Override
153 public V get(K k) {
154     return entries.get(k);
155 }
156
157 @Override
158 public int compareTo(BPlusLeafBlock<K, V> o) {
159     return ((Comparable<K>) entries.firstKey()).compareTo(o.entries
160         .firstKey());
161 }
162
163 @Override
164 public K firstKey() {
165     return entries.firstKey();
166 }
167
168 }
169
170 static class BPlusBranchBlock<K, V> extends BPlusAnyBlock<K, V> {
171     SortedMap<K, BPlusAnyBlock<K, V>> entries =
172     createInternalBlockEntries();
173     int maxSize;
174
175     private BPlusAnyBlock<K, V> left;
176
177     public boolean isEmpty() {
178         return entries.isEmpty();
179     }
180
181     public BPlusBranchBlock(int maxSize2) {
182         this.maxSize = maxSize2;
183     }
184
185     public SplitOrValue<K, V> put(K k, V v) {
```

```
186         BPlusAnyBlock<K, V> b = getBlock(k);
187
188         SplitOrValue<K, V> sv = b.put(k, v);
189
190         if (sv.isSplit()) {
191
192             entries.put(sv.k, sv.right);
193
194             if (entries.size() >= maxSize)
195                 sv = splitBlock();
196             else
197                 sv = new SplitOrValue<K, V>(null);
198         }
199
200         return sv;
201     }
202
203     BPlusAnyBlock<K, V> getBlock(K k) {
204         assert (entries.size() > 0);
205         BPlusAnyBlock<K, V> b = entries.get(k);
206         if (b == null) {
207             // headMap returns less than k
208             SortedMap<K, BPlusAnyBlock<K, V>> head = entries.headMap(k);
209             if (head.isEmpty()) {
210                 b = left;
211
212                 // System.out.println("for key " + k
213                 // + " getting from leftmost block");
214                 // showEntries();
215
216             } else {
217                 b = entries.get(head.lastKey());
218
219                 // System.out.println("for key " + k
220                 // + " getting from block with key " + head.lastKey());
221                 // showEntries();
222             }
223         }
224         assert (b != null);
225         return b;
226     }
227
228     public void showEntries() {
229         System.out.print("entries = ");
230         for (K k : entries.keySet()) {
231             System.out.print(k + " ");
232         }
233         System.out.println();
234     }
235
236
237     public SplitOrValue<K, V> splitBlock() {
238
239         Set<Entry<K, BPlusAnyBlock<K, V>>> ee = entries.entrySet();
240         int i = 0;
241         BPlusBranchBlock<K, V> right = new BPlusBranchBlock<K, V>(maxSize);
242         SortedMap<K, BPlusAnyBlock<K, V>> leftEntries =
243         createInternalBlockEntries();
244
245         for (Entry<K, BPlusAnyBlock<K, V>> e : ee) {
246             // System.out.print("split check " + e.getKey() + ":"
247             // );
248             if (++i <= maxSize / 2)
```

```
248         leftEntries.put(e.getKey(), e.getValue());
249     else
250         right.entries.put(e.getKey(), e.getValue());
251
252     }
253     // System.out.println("\n");
254     this.entries = leftEntries;
255
256     return new SplitOrValue<K, V>(this, right);
257 }
258
259 private SortedMap<K, BPlusAnyBlock<K, V>> createInternalBlockEntries()
260 {
261     return new TreeMap<K, BPlusAnyBlock<K, V>>();
262 }
263
264 @Override
265 public V get(K k) {
266     BPlusAnyBlock<K, V> b = getBlock(k);
267
268     return b.get(k);
269 }
270
271 @Override
272 public K firstKey() {
273     return entries.firstKey();
274 }
275
276
277 @Override
278 public SortedMap<K, V> subMap(K fromKey, K toKey) {
279     TreeMap<K, V> map = new TreeMap<K, V>();
280     BPlusLeafBlock<K, V> b1 = getLeafBlock(fromKey);
281
282     BPlusLeafBlock<K, V> b2 = getLeafBlock(toKey);
283
284     SortedSet<BPlusLeafBlock<K, V>> range = blockList.subSet(b1, b2);
285
286     for (BPlusLeafBlock<K, V> b : range) {
287         SortedMap<K, V> m = b.entries.subMap(fromKey, toKey);
288         map.putAll(m);
289     }
290
291     return map;
292 }
293
294 private BPlusLeafBlock<K, V> getLeafBlock(K key) {
295     BPlusAnyBlock<K, V> b1;
296     b1 = root;
297     while (b1 instanceof BPlusBranchBlock<?, ?>) {
298         b1 = ((BPlusBranchBlock<K, V>) b1).getBlock(key);
299     }
300
301     return (BPlusLeafBlock<K, V>) b1;
302 }
303
304
305 public BPlusLeafBlock<K, V> createLeafBlock(SortedMap<K, V> rightEntries) {
306     BPlusLeafBlock<K, V> b = new BPlusLeafBlock<K, V>(this, rightEntries);
307     blockList.add(b);
308     return b;
309 }
```

```
310
311 @Override
312 public SortedMap<K, V> headMap(K toKey) {
313     return subMap(firstKey(), toKey);
314 };
315
316 @Override
317 public SortedMap<K, V> tailMap(K fromKey) {
318     return subMap(fromKey, lastKey());
319 }
320
321 @Override
322 public K firstKey() {
323     return blockList.first().entries.firstKey();
324 }
325
326 @Override
327 public K lastKey() {
328     return blockList.last().entries.lastKey();
329 }
330
331 @Override
332 public int size() {
333     return (int) getLongSize();
334 }
335
336 private long getLongSize() {
337     long i = 0;
338     for (BPlusLeafBlock<K, V> b : blockList) {
339         i += b.entries.size();
340     }
341     return i;
342 }
343
344 @Override
345 public boolean isEmpty() {
346     return root.isEmpty();
347 }
348
349 @Override
350 public boolean containsKey(Object key) {
351     return get(key) != null;
352 }
353
354 @Override
355 public boolean containsValue(Object value) {
356     return false;
357 }
358
359 @Override
360 public V get(Object key) {
361     return (V) root.get((K) key);
362 }
363
364 @Override
365 public V put(K key, V value) {
366     if (root == null) {
367         SortedMap<K, V> entries = BPlusLeafBlock.getEntryCollection();
368         entries.put(key, value);
369         root = createLeafBlock(entries);
370         return null;
371     }
372     SplitOrValue<K, V> result = root.put(key, value);
```

```
373         if (result.isSplit()) {
374             BPlusBranchBlock<K, V> root = new BPlusBranchBlock<K, V>(
375                 maxInternalBlockSize);
376             root.left = result.left;
377             root.entries.put(result.k, result.right);
378             this.root = root;
379         }
380         return result.v;
381     }
382
383     @Override
384     public V remove(Object key) {
385         return null;
386     }
387
388     @Override
389     public void putAll(Map<? extends K, ? extends V> m) {
390         for (K k : m.keySet()) {
391             put(k, m.get(k));
392         }
393     }
394
395     @Override
396     public void clear() {
397     }
398
399
400     @Override
401     public Set<K> keySet() {
402         TreeSet<K> kk = new TreeSet<K>();
403         for (BPlusLeafBlock<K, V> b : blockList) {
404             kk.addAll(b.entries.keySet());
405         }
406         return kk;
407     }
408
409     @Override
410     public Collection<V> values() {
411         // TODO Auto-generated method stub
412         return null;
413     }
414
415     @Override
416     public Set<java.util.Map.Entry<K, V>> entrySet() {
417         // TODO Auto-generated method stub
418         return null;
419     }
420
421     @Override
422     public Comparator<? super K> comparator() {
423         // TODO Auto-generated method stub
424         return null;
425     }
426
427     public void showLeaves() {
428
429         for (BPlusLeafBlock<K, V> b : blockList) {
430             System.out.println("Block");
431             for (Entry<K, V> e : b.entries.entrySet()) {
432                 System.out.print(e.getKey() + ":" + e.getValue() + " ");
433             }
434             System.out.println();
435         }
436     }
```

```
436     }
437 }
438
439 package btreemap;
440
441 import java.util.ArrayList;
442 import java.util.Comparator;
443 import java.util.List;
444 import java.util.Random;
445
446 /** driver program to test B+ tree */
447
448 public class BPlusTreeTest1 {
449
450     private static final int N = 1200000;
451
452     public static void main(String[] args) {
453         BPlusTreeMap<Integer, Integer> map = new BPlusTreeMap<Integer, Integer>
454         (
455             400, 200 );// 5000002);
456         Random r = new Random();
457         ArrayList<Integer> t = new ArrayList<Integer>();
458         Comparator<Integer> comparator = new Comparator<Integer>() {
459
460             @Override
461             public int compare(Integer o1, Integer o2) {
462                 // TODO Auto-generated method stub
463                 return o1.intValue() - o2.intValue();
464             }
465         };
466
467         List<Integer> testVals = new ArrayList<Integer>();
468         for (int i = 0; i < N; ++i) {
469             testVals.add(i);
470         }
471
472         for (int i = 0; i < N; ++i) {
473             int x = r.nextInt(N);
474             int z = testVals.set(x, testVals.get(0));
475             testVals.set(0, z);
476         }
477
478         for (int i = 0; i < N; ++i) {
479             map.put(testVals.get(i), testVals.get(i));
480             showProgress("put", testVals, i);
481         }
482         System.err.println("output " + N + " vals");
483
484         try {
485             for (int i = 0; i < N; ++i) {
486                 showProgress("get", testVals, i);
487                 int x = testVals.get(i);
488                 if (x != map.get(x))
489                     System.err.println("Expecting " + x + " got " +
490 map.get(x));
491             }
492             System.err.println("\nChecked " + N + " entries");
493         }
```

```
497     } catch (Exception e) {
498         // TODO: handle exception
499         e.printStackTrace();
500         map.showLeaves();
501     }
502 }
503
504 private static void showProgress(String label, List<Integer> testVals, int
i) {
505     if (i % (N / 1000) == 0) {
506         System.out.printf("%s %d:%d; ", label, i, testVals.get(i));
507         if (i % (N / 10100) == 0)
508             System.out.println();
509     }
510 }
511 }
```

5.8: B Trees is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- **6.8: B Trees** by Wikibooks - Data Structures is licensed [CC BY-SA 4.0](#).

5.9: Treaps

The invariant in a binary tree is that left is less than right with respect to insertion keys. e.g. for a key with order, $\text{ord}(L) < \text{ord}(R)$. This doesn't dictate the relationship of nodes however, and left and right rotation does not affect the above. Therefore another order can be imposed. If the order is randomised, it is likely to counteract any skewness of a plain binary tree e.g. when inserting an already sorted input in order.

Below is a java example implementation, including a plain binary tree delete code example.

```
001 import java.util.Iterator;
002 import java.util.LinkedList;
003 import java.util.Random;
004
005 public class Treap1<K extends Comparable<K>, V> {
006     public Treap1(boolean test) {
007         this.test = test;
008     }
009
010     public Treap1() {}
011     boolean test = false;
012     static Random random = new Random(System.currentTimeMillis());
013
014     class TreapNode {
015         int priority = 0;
016         K k;
017         V val;
018         TreapNode left, right;
019
020         public TreapNode() {
021             if (!test) {
022                 priority = random.nextInt();
023             }
024         }
025     }
026
027     TreapNode root = null;
028
029     void insert(K k, V val) {
030         root = insert(k, val, root);
031     }
032
033     TreapNode insert(K k, V val, TreapNode node) {
034         TreapNode node2 = new TreapNode();
035         node2.k = k;
036         node2.val = val;
037         if (node == null) {
038             node = node2;
039         } else if (k.compareTo(node.k) < 0) {
040             node.left = insert(k, val, node.left);
041         } else {
042             node.right = insert(k, val, node.right);
043         }
044
045         if (node.left != null && node.left.priority > node.priority) {
046             // left rotate
047             TreapNode tmp = node.left;
048             node.left = node.left.right;
049             tmp.right = node;
050             node = tmp;
051         } else if (node.right != null && node.right.priority > node.priority) {
052             // right rotate
053             TreapNode tmp = node.right;
054             node.right = node.right.left;
055             tmp.left = node;
056             node = tmp;
057         }
058     }
059 }
```

```
062         return node;
063     }
064
065     V find(K k) {
066         return findNode(k, root);
067     }
068
069     private V findNode(K k, Treap1<K, V>.TreapNode node) {
070
071         if (node == null)
072             return null;
073         if (k.compareTo(node.k) < 0) {
074             return findNode(k, node.left);
075         } else if (k.compareTo(node.k) > 0) {
076             return findNode(k, node.right);
077         } else {
078             return node.val;
079         }
080     }
081
082     static class Deleted {
083         boolean success = false;
084     }
085
086     boolean delete(K k) {
087         Deleted del = new Deleted();
088         root = deleteNode(k, root, del);
089         return del.success;
090     }
091
092     private Treap1<K, V>.TreapNode deleteNode(K k, Treap1<K, V>.TreapNode node,
Deleted del) {
093
094         if (node == null) {
095             return null;
096         } else if (k.compareTo(node.k) < 0) {
097             node.left = deleteNode(k, node.left, del);
098         } else if (k.compareTo(node.k) > 0) {
099             node.right = deleteNode(k, node.right, del);
100
101             // k.compareTo(node.k) == 0
102         } else if ( node.left == null ) {
103             del.success = true;
104             return node.right;
105         } else if ( node.right == null ) {
106             del.success = true;
107             return node.left;
108         } else if (node.left !=null && node.right != null){
109             /*
110             // left rotate and all delete on left subtree
111             TreapNode tmp = node.right;
112             node.right = node.right.left;
113             tmp.left = node;
114             node = tmp;
115             node.left = deleteNode(k , node.left, del);
116             */
117             // more standard method ? doesn't disturb tree structure as much
118             // find leftmost descendant of the right child node and replace
contents
119             TreapNode n2 = node.right;
120             TreapNode previous2 = null;
121             while (n2.left != null) {
122                 previous2 = n2;
```

```
123         n2 = n2.left;
124     }
125
126     if (previous2 != null) {
127         previous2.left = n2.right;
128         //n2 has no parent link, orphaned
129     } else {
130         node.right = n2.right;
131         //n2 has no parent link, orphaned
132     }
133
134     node.k = n2.k;
135     node.val = n2.val;
136     del.success = true;
137     // once n2 out of scope, the orphaned node at n2 will be garbage
    collected,
138 }
139
140     return node;
141 }
142
143 public static void main(String[] args) {
144     LinkedList<Integer> dat = new LinkedList<Integer>();
145
146     for (int i = 0; i < 15000; ++i) {
147         dat.add(i);
148     }
149
150     testNumbers(dat, true); // no random priority balancing
151     testNumbers(dat, false);
152 }
153
154 private static void testNumbers(LinkedList<Integer> dat,
155     boolean test) {
156     Treap1<Integer, Integer> treap = new Treap1<>(test);
157
158     for (Integer integer : dat) {
159         treap.insert(integer, integer);
160     }
161
162     long t1 = System.currentTimeMillis();
163     Iterator<Integer> desc = dat.iterator();
164     int found = 0;
165     while (desc.hasNext()) {
166         Integer j = desc.next();
167         Integer i = treap.find(j);
168         if (j.equals(i)) {
169             ++found;
170         }
171     }
172     long t2 = System.currentTimeMillis();
173     System.out.println("found = " + found + " in " + (t2 - t1));
174
175     System.out.println("test delete");
176     int deleted = 0;
177
178     for (Integer integer : dat) {
179         if (treap.delete(integer))
180             ++deleted;
181     }
182     System.out.println("Deleted = " + deleted);
183 }
184 }
```

5.9: Treaps is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- 6.9: Treaps by [Wikibooks - Data Structures](#) is licensed [CC BY-SA 4.0](#).

5.10: References

- William Ford and William Tapp. *Data Structures with C++ using STL*. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2002.

External Links

- [Presentation of Binary Search Trees](#)
 - [Data Structure Trees](#)
-

5.10: References is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- **6.10: References** by [Wikibooks - Data Structures](#) is licensed [CC BY-SA 4.0](#).

CHAPTER OVERVIEW

6: Graphs

In this chapter, we study two representations of graphs and basic algorithms that use these representations.

Mathematically, a (directed) graph is a pair $G = (V, E)$ where V is a set of vertices and E is a set of ordered pairs of vertices called edges. An edge (i, j) is directed from i to j ; i is called the source of the edge and j is called the target. A path in G is a sequence of vertices $v_0, \dots, v_k$ such that, for every $i \in \{1, \dots, k\}$, the edge (v_{i-1}, v_i) is in E . A path $v_0, \dots, v_k$ is a cycle if, additionally, the edge (v_k, v_0) is in E . A path (or cycle) is simple if all of its vertices are unique. If there is a path from some vertex v_i to some vertex v_j then we say that v_j is reachable from v_i . An example of a graph is shown in Figure 6.1.

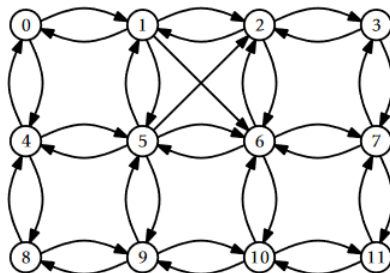


Figure 6.1: A graph with twelve vertices. Vertices are drawn as numbered circles and edges are drawn as pointed curves pointing from source to target.

Due to their ability to model so many phenomena, graphs have an enormous number of applications. There are many obvious examples. Computer networks can be modelled as graphs, with vertices corresponding to computers and edges corresponding to (directed) communication links between those computers. City streets can be modelled as graphs, with vertices representing intersections and edges representing streets joining consecutive intersections.

Less obvious examples occur as soon as we realize that graphs can model any pairwise relationships within a set. For example, in a university setting we might have a timetable conflict graph whose vertices represent courses offered in the university and in which the edge (i, j) is present if and only if there is at least one student that is taking both class i and class j . Thus, an edge indicates that the exam for class i should not be scheduled at the same time as the exam for class j .

Throughout this section, we will use n to denote the number of vertices of G and m to denote the number of edges of G . That is, $n = |V|$ and $m = |E|$. Furthermore, we will assume that $V = \{0, \dots, n-1\}$. Any other data that we would like to associate with the elements of V can be stored in an array of length n .

Some typical operations performed on graphs are:

- `addEdge(i, j)`: Add the edge (i, j) to E .
- `removeEdge(i, j)`: Remove the edge (i, j) from E .
- `hasEdge(i, j)`: Check if the edge $(i, j) \in E$
- `outEdges(i)`: Return a List of all integers j such that $(i, j) \in E$
- `inEdges(i)`: Return a List of all integers j such that $(j, i) \in E$

Note that these operations are not terribly difficult to implement efficiently. For example, the first three operations can be implemented directly by using a `HashSet`, so they can be implemented in constant expected time using the hash tables discussed in Chapter 7. The last two operations can be implemented in constant time by storing, for each vertex, a list of its adjacent vertices.

However, different applications of graphs have different performance requirements for these operations and, ideally, we can use the simplest implementation that satisfies all the application's requirements. For this reason, we discuss two broad categories of graph representations.

[6.1: Representing a Graph by a Matrix](#)

[6.2: A Graph as a Collection of Lists](#)

[6.3: Graph Traversal](#)

[6.4: Discussion and Exercises](#)

6: [Graphs](#) is shared under a [CC BY](#) license and was authored, remixed, and/or curated by LibreTexts.

6.1: Representing a Graph by a Matrix

An adjacency matrix is a way of representing an n vertex graph $G = (V, E)$ by an $n \times n$ matrix, a , whose entries are boolean values.

```
1 int n;  
2 boolean[][] a;  
3 AdjacencyMatrix(int n0) {  
4     n = n0;  
5     a = new boolean[n][n];  
6 }
```

The matrix entry $a[i][j]$ is defined as

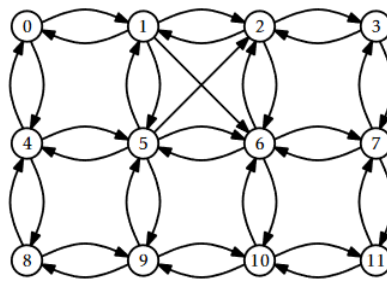
$$a[i][j] = \begin{cases} \text{true} & \text{if } (i, j) \in E \\ \text{false} & \text{otherwise} \end{cases}$$

The adjacency matrix for the graph in Figure 6.1 is shown in Figure 6.1.1.

In this representation, the operations `addEdge(i, j)`, `removeEdge(i, j)`, and `hasEdge(i, j)` just involve setting or reading the matrix entry $a[i][j]$:

```
1 void addEdge(int i, int j) {  
2     a[i][j] = true;  
3 }  
4 void removeEdge(int i, int j) {  
5     a[i][j] = false;  
6 }  
7 boolean hasEdge(int i, int j) {  
8     return a[i][j];  
9 }
```

These operations clearly take constant time per operation.



	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1	0	0	1	0	0	0	0	0	0	0
1	1	0	1	0	0	1	1	0	0	0	0	0
2	1	0	0	1	0	0	1	0	0	0	0	0
3	0	0	1	0	0	0	0	1	0	0	0	0
4	1	0	0	0	0	1	0	0	1	0	0	0
5	0	1	1	0	1	0	1	0	0	1	0	0
6	0	0	1	0	0	1	0	1	0	0	1	0
7	0	0	0	1	0	0	1	0	0	0	0	1
8	0	0	0	0	1	0	0	0	0	1	0	0
9	0	0	0	0	0	1	0	0	1	0	1	0
10	0	0	0	0	0	0	1	0	0	1	0	1
11	0	0	0	0	0	0	0	1	0	0	1	0

Figure 6.1.1: A graph and its adjacency matrix.

Where the adjacency matrix performs poorly is with the `outEdges(i)` and `inEdges(i)` operations. To implement these, we must scan all n entries in the corresponding row or column of `a` and gather up all the indices, j , where `a[i][j]`, respectively `a[j][i]`, is true.

```

01 List<Integer> outEdges(int i) {
02     List<Integer> edges = new ArrayList<Integer>();
03     for (int j = 0; j < n; j++)
04         if (a[i][j]) edges.add(j);
05     return edges;
06 }
07 List<Integer> inEdges(int i) {
08     List<Integer> edges = new ArrayList<Integer>();
09     for (int j = 0; j < n; j++)
10         if (a[j][i]) edges.add(j);
11     return edges;
12 }

```

These operations clearly take $O(n)$ time per operation.

Another drawback of the adjacency matrix representation is that it is large. It stores an $n \times n$ boolean matrix, so it requires at least n^2 bits of memory. The implementation here uses a matrix of `boolean` values so it actually uses on the order of n^2 bytes of memory. A more careful implementation, which packs w boolean values into each word of memory, could reduce this space usage to $O(n^2/w)$ words of memory.

Theorem 6.1.1

The *AdjacencyMatrix* data structure implements the *Graph* interface. An *AdjacencyMatrix* supports the operations

- `addEdge(i, j)`, `removeEdge(i, j)`, and `hasEdge(i, j)` in constant time per operation; and
- `inEdges(i)`, and `outEdges(i)` in $O(n)$ time per operation.

The space used by an *AdjacencyMatrix* is $O(n^2)$.

Despite its high memory requirements and poor performance of the `inEdges(i)` and `outEdges(i)` operations, an *AdjacencyMatrix* can still be useful for some applications. In particular, when the graph G is dense, i.e., it has close to n^2 edges, then a memory usage of n^2 may be acceptable.

The *AdjacencyMatrix* data structure is also commonly used because algebraic operations on the matrix a can be used to efficiently compute properties of the graph G . This is a topic for a course on algorithms, but we point out one such property here: If we treat the entries of a as integers (1 for `true` and 0 for `false`) and multiply a by itself using matrix multiplication then we get the matrix a^2 . Recall, from the definition of matrix multiplication, that

$$a^2[i][j] = \sum_{k=0}^{n-1} a[i][k] \cdot a[k][j] .$$

Interpreting this sum in terms of the graph G , this formula counts the number of vertices, k , such that G contains both edges (i, k) and (k, j) . That is, it counts the number of paths from i to j (through intermediate vertices, k) whose length is exactly two. This observation is the foundation of an algorithm that computes the shortest paths between all pairs of vertices in G using only $O(\log n)$ matrix multiplications.

6.1: Representing a Graph by a Matrix is shared under a [CC BY](#) license and was authored, remixed, and/or curated by LibreTexts.

The `removeEdge(i, j)` operation searches through the list `adj[i]` until it finds `j` and then removes it:

```
1 void removeEdge(int i, int j) {
2     Iterator<Integer> it = adj[i].iterator();
3     while (it.hasNext()) {
4         if (it.next() == j) {
5             it.remove();
6             return;
7         }
8     }
9 }
```

This takes $O(\deg(i))$ time, where $\deg(i)$ (the degree of i) counts the number of edges in E that have i as their source.

The `hasEdge(i, j)` operation is similar; it searches through the list `adj[i]` until it finds `j` (and returns true), or reaches the end of the list (and returns false):

```
1 boolean hasEdge(int i, int j) {
2     return adj[i].contains(j);
3 }
```

This also takes $O(\deg(i))$ time.

The `outEdges(i)` operation is very simple; it returns the list `adj[i]` :

```
1 List<Integer> outEdges(int i) {
2     return adj[i];
3 }
```

This clearly takes constant time.

The `inEdges(i)` operation is much more work. It scans over every vertex j checking if the edge (i, j) exists and, if so, adding j to the output list:

```
1 List<Integer> inEdges(int i) {
2     List<Integer> edges = new ArrayList<Integer>(Integer.class);
3     for (int j = 0; j < n; j++)
4         if (adj[j].contains(i)) edges.add(j);
5     return edges;
6 }
```

This operation is very slow. It scans the adjacency list of every vertex, so it takes $O(n + m)$ time.

The following theorem summarizes the performance of the above data structure:

Theorem 6.2.1

The AdjacencyLists data structure implements the Graph interface. An AdjacencyLists supports the operations

- `addEdge(i, j)` in constant time per operation;
- `removeEdge(i, j)` and `hasEdge(i, j)` in $O(\deg(i))$ time per operation;
- `outEdges(i)` in constant time per operation; and
- `inEdges(i)` in $O(n + m)$ time per operation.

The space used by a AdjacencyLists is $O(n + m)$.

As alluded to earlier, there are many different choices to be made when implementing a graph as an adjacency list. Some questions that come up include:

- What type of collection should be used to store each element of `adj`? One could use an array-based list, a linked-list, or even a hashtable.
- Should there be a second adjacency list, `inadj`, that stores, for each `i`, the list of vertices, `j`, such that $(j, i) \in E$? This can greatly reduce the running-time of the `inEdges(i)` operation, but requires slightly more work when adding or removing edges.
- Should the entry for the edge (i, j) in `adj[i]` be linked by a reference to the corresponding entry in `inadj[j]`?
- Should edges be first-class objects with their own associated data? In this way, `adj` would contain lists of edges rather than lists of vertices (integers).

Most of these questions come down to a tradeoff between complexity (and space) of implementation and performance features of the implementation.

6.2: A Graph as a Collection of Lists is shared under a [CC BY](#) license and was authored, remixed, and/or curated by LibreTexts.

6.3: Graph Traversal

In this section we present two algorithms for exploring a graph, starting at one of its vertices, i , and finding all vertices that are reachable from i . Both of these algorithms are best suited to graphs represented using an adjacency list representation. Therefore, when analyzing these algorithms we will assume that the underlying representation is an `AdjacencyLists`.

6.3.1 Breadth-First Search

The breadth-first-search algorithm starts at a vertex i and visits, first the neighbours of i , then the neighbours of the neighbours of i , then the neighbours of the neighbours of the neighbours of i , and so on.

This algorithm is a generalization of the breadth-first traversal algorithm for binary trees (Open Data Structures, Section 6.1.2), and is very similar; it uses a queue, q , that initially contains only i . It then repeatedly extracts an element from q and adds its neighbours to q , provided that these neighbours have never been in q before. The only major difference between the breadth-first-search algorithm for graphs and the one for trees is that the algorithm for graphs has to ensure that it does not add the same vertex to q more than once. It does this by using an auxiliary boolean array, `seen`, that tracks which vertices have already been discovered.

```

01 void bfs(Graph g, int r) {
02     boolean[] seen = new boolean[g.nVertices()];
03     Queue<Integer> q = new SLList<Integer>();
04     q.add(r);
05     seen[r] = true;
06     while (!q.isEmpty()) {
07         int i = q.remove();
08         for (Integer j : g.outEdges(i)) {
09             if (!seen[j]) {
10                 q.add(j);
11                 seen[j] = true;
12             }
13         }
14     }
15 }

```

An example of running `bfs(g, 0)` on the graph from Figure 6.1 is shown in Figure 6.3.1. Different executions are possible, depending on the ordering of the adjacency lists; Figure 6.3.1 uses the adjacency lists in Figure 6.2.1.

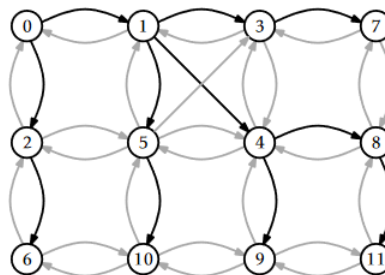


Figure 6.3.1: An example of breadth-first-search starting at node 0. Nodes are labelled with the order in which they are added to q . Edges that result in nodes being added to q are drawn in black, other edges are drawn in grey.

Analyzing the running-time of the `bfs(g, i)` routine is fairly straightforward. The use of the `seen` array ensures that no vertex is added to q more than once. Adding (and later removing) each vertex from q takes constant time per vertex for a total of $O(n)$ time. Since each vertex is processed by the inner loop at most once, each adjacency list is processed at most once, so each edge of G is processed at most once. This processing, which is done in the inner loop takes constant time per iteration, for a total of $O(m)$ time. Therefore, the entire algorithm runs in $O(n + m)$ time.

The following theorem summarizes the performance of the `bfs(g, r)` algorithm.

Theorem 6.3.1

When given as input a Graph, g , that is implemented using the *AdjacencyLists* data structure, the $\text{bfs}(g, r)$ algorithm runs in $O(n + m)$ time.

A breadth-first traversal has some very special properties. Calling $\text{bfs}(g, r)$ will eventually enqueue (and eventually dequeue) every vertex j such that there is a directed path from r to j . Moreover, the vertices at distance 0 from r (r itself) will enter q before the vertices at distance 1, which will enter q before the vertices at distance 2, and so on. Thus, the $\text{bfs}(g, r)$ method visits vertices in increasing order of distance from r and vertices that cannot be reached from r are never visited at all.

A particularly useful application of the breadth-first-search algorithm is, therefore, in computing shortest paths. To compute the shortest path from r to every other vertex, we use a variant of $\text{bfs}(g, r)$ that uses an auxiliary array, p , of length n . When a new vertex j is added to q , we set $p[j] = i$. In this way, $p[j]$ becomes the second last node on a shortest path from r to j . Repeating this, by taking $p[p[j]]$, $p[p[p[j]]]$, and so on we can reconstruct the (reversal of) a shortest path from r to j .

6.3.2 Depth-First Search

The depth-first-search algorithm is similar to the standard algorithm for traversing binary trees; it first fully explores one subtree before returning to the current node and then exploring the other subtree. Another way to think of depth-first-search is by saying that it is similar to breadth-first search except that it uses a stack instead of a queue.

During the execution of the depth-first-search algorithm, each vertex, i , is assigned a colour, $c[i]$: *white* if we have never seen the vertex before, *grey* if we are currently visiting that vertex, and *black* if we are done visiting that vertex. The easiest way to think of depth-first-search is as a recursive algorithm. It starts by visiting r . When visiting a vertex i , we first mark i as *grey*. Next, we scan i 's adjacency list and recursively visit any white vertex we find in this list. Finally, we are done processing i , so we colour i *black* and return.

```

01 void dfs(Graph g, int r) {
02     byte[] c = new byte[g.nVertices()];
03     dfs(g, r, c);
04 }
05 void dfs(Graph g, int i, byte[] c) {
06     c[i] = grey; // currently visiting i
07     for (Integer j : g.outEdges(i)) {
08         if (c[j] == white) {
09             c[j] = grey;
10             dfs(g, j, c);
11         }
12     }
13     c[i] = black; // done visiting i
14 }

```

An example of the execution of this algorithm is shown in Figure 6.3.2.

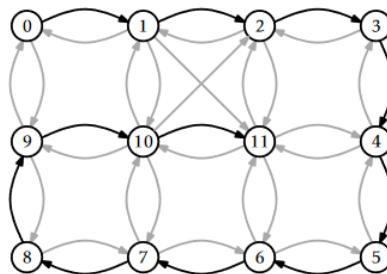


Figure 6.3.2: An example of depth-first-search starting at node 0. Nodes are labelled with the order in which they are processed. Edges that result in a recursive call are drawn in black, other edges are drawn in grey.

Although depth-first-search may best be thought of as a recursive algorithm, recursion is not the best way to implement it. Indeed, the code given above will fail for many large graphs by causing a stack overflow. An alternative implementation is to replace the

recursion stack with an explicit stack, *s*. The following implementation does just that:

```

01 void dfs2(Graph g, int r) {
02     byte[] c = new byte[g.nVertices()];
03     Stack<Integer> s = new Stack<Integer>();
04     s.push(r);
05     while (!s.isEmpty()) {
06         int i = s.pop();
07         if (c[i] == white) {
08             c[i] = grey;
09             for (int j : g.outEdges(i))
10                 s.push(j);
11         }
12     }
13 }

```

In the preceding code, when the next vertex, *i*, is processed, *i* is coloured *grey* and then replaced, on the stack, with its adjacent vertices. During the next iteration, one of these vertices will be visited.

Not surprisingly, the running times of *dfs*(*g*, *r*) and *dfs2*(*g*, *r*) are the same as that of *bfs*(*g*, *r*):

Theorem 6.3.2

*When given as input a Graph, *g*, that is implemented using the AdjacencyLists data structure, the *dfs*(*g*, *r*) and *dfs2*(*g*, *r*) algorithms each run in $O(n + m)$ time.*

As with the breadth-first-search algorithm, there is an underlying tree associated with each execution of depth-first-search. When a node $i \neq r$ goes from *white* to *grey*, this is because *dfs*(*g*, *i*, *c*) was called recursively while processing some node *i'*. (In the case of *dfs2*(*g*, *r*) algorithm, *i* is one of the nodes that replaced *i'* on the stack.) If we think of *i'* as the parent of *i*, then we obtain a tree rooted at *r*. In Figure 6.3.2, this tree is a path from vertex 0 to vertex 11.

An important property of the depth-first-search algorithm is the following: Suppose that when node *i* is coloured *grey*, there exists a path from *i* to some other node *j* that uses only *white* vertices. Then *j* will be coloured first *grey* then *black* before *i* is coloured *black*. (This can be proven by contradiction, by considering any path *P* from *i* to *j*.)

One application of this property is the detection of cycles. Refer to Figure 6.3.3. Consider some cycle, *C*, that can be reached from *r*. Let *i* be the first node of *C* that is coloured *grey*, and let *j* be the node that precedes *i* on the cycle *C*. Then, by the above property, *j* will be coloured *grey* and the edge (*j*, *i*) will be considered by the algorithm while *i* is still *grey*. Thus, the algorithm can conclude that there is a path, *P*, from *i* to *j* in the depth-first-search tree and the edge (*j*, *i*) exists. Therefore, *P* is also a cycle.

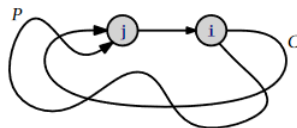


Figure 6.3.3: The depth-first-search algorithm can be used to detect cycles in *G*. The node *j* is coloured *grey* while *i* is still *grey*. This implies that there is a path, *P*, from *i* to *j* in the depth-first-search tree, and the edge (*j*, *i*) implies that *P* is also a cycle.

6.3: Graph Traversal is shared under a CC BY license and was authored, remixed, and/or curated by LibreTexts.

6.4: Discussion and Exercises

The running times of the depth-first-search and breadth-first-search algorithms are somewhat overstated by the Theorems 6.3.1 and 6.3.2. Define n_r as the number of vertices, i , of G , for which there exists a path from r to i . Define m_r as the number of edges that have these vertices as their sources. Then the following theorem is a more precise statement of the running times of the breadth-first-search and depth-first-search algorithms. (This more refined statement of the running time is useful in some of the applications of these algorithms outlined in the exercises.)

Theorem 6.4.1

When given as input a Graph, g , that is implemented using the `AdjacencyLists` data structure, the $\text{bfs}(g, r)$, $\text{dfs}(g, r)$ and $\text{dfs2}(g, r)$ algorithms each run in $O(n_r + m_r)$ time.

Breadth-first search seems to have been discovered independently by Moore [52] and Lee [49] in the contexts of maze exploration and circuit routing, respectively.

Adjacency-list representations of graphs were presented by Hopcroft and Tarjan [40] as an alternative to the (then more common) adjacency-matrix representation. This representation, as well as depth-first-search, played a major part in the celebrated Hopcroft-Tarjan planarity testing algorithm that can determine, in $O(n)$ time, if a graph can be drawn, in the plane, and in such a way that no pair of edges cross each other [41].

In the following exercises, an undirected graph is one in which, for every i and j , the edge (i, j) is present if and only if the edge (j, i) is present.

Exercise 6.4.1

Draw an adjacency list representation and an adjacency matrix representation of the graph in Figure 6.4.1.

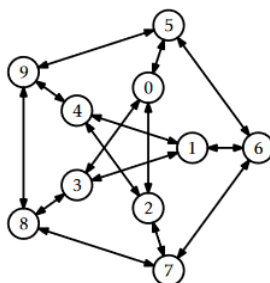


Figure 6.4.1: An example graph.

Exercise 6.4.2

The incidence matrix representation of a graph, G , is an $n \times m$ matrix, A , where

$$A_{i,j} = \begin{cases} -1 & \text{if vertex } i \text{ is the source of edge } j \\ +1 & \text{if vertex } i \text{ is the target of edge } j \\ 0 & \text{otherwise.} \end{cases}$$

1. Draw the incident matrix representation of the graph in Figure 6.4.1.
2. Design, analyze and implement an incidence matrix representation of a graph. Be sure to analyze the space, the cost of `addEdge(i, j)`, `removeEdge(i, j)`, `hasEdge(i, j)`, `inEdges(i)`, and `outEdges(i)`.

Exercise 6.4.3

Illustrate an execution of the $\text{bfs}(G, 0)$ and $\text{dfs}(G, 0)$ on the graph, G , in Figure 6.4.1.

Exercise 6.4.4

Let G be an undirected graph. We say G is connected if, for every pair of vertices i and j in G , there is a path from i to j (since G is undirected, there is also a path from j to i). Show how to test if G is connected in $O(n + m)$ time.

Exercise 6.4.5

Let G be an undirected graph. A connected-component labelling of G partitions the vertices of G into maximal sets, each of which forms a connected subgraph. Show how to compute a connected component labelling of G in $O(n + m)$ time.

Exercise 6.4.6

Let G be an undirected graph. A spanning forest of G is a collection of trees, one per component, whose edges are edges of G and whose vertices contain all vertices of G . Show how to compute a spanning forest of G in $O(n + m)$ time.

Exercise 6.4.7

We say that a graph G is strongly-connected if, for every pair of vertices i and j in G , there is a path from i to j . Show how to test if G is strongly-connected in $O(n + m)$ time.

Exercise 6.4.8

Given a graph $G = (V, E)$ and some special vertex $r \in V$, show how to compute the length of the shortest path from r to i for every vertex $i \in V$.

Exercise 6.4.9

Give a (simple) example where the `dfs(g, r)` code visits the nodes of a graph in an order that is different from that of the `dfs2(g, r)` code. Write a version of `dfs2(g, r)` that always visits nodes in exactly the same order as `dfs(g, r)`. (Hint: Just start tracing the execution of each algorithm on some graph where r is the source of more than 1 edge.)

Exercise 6.4.10

A universal sink in a graph G is a vertex that is the target of $n - 1$ edges and the source of no edges.¹ Design and implement an algorithm that tests if a graph G , represented as an `AdjacencyMatrix`, has a universal sink. Your algorithm should run in $O(n)$ time.

Footnotes

¹A universal sink, v , is also sometimes called a celebrity: Everyone in the room recognizes v , but v doesn't recognize anyone else in the room.

CHAPTER OVERVIEW

7: Hash Tables

A **hash table**, or a **hash map**, is a data structure that associates *keys* with *values*. The primary operation it supports efficiently is a *lookup*: given a key (e.g. a person's name), find the corresponding value (e.g. that person's telephone number). It works by transforming the key using a [hash function](#) into a *hash*, a number that the hash table uses to locate the desired value. This hash maps directly to a bucket in the array of key/value pairs, hence the name hash map. The mapping method lets us directly access the storage location for any key/value pair.

Hash table<Element> Operations

`make-hash-table(integer n): HashTable`

Create a hash table with *n* buckets.

`get-value(HashTable h, Comparable key): Element`

Returns the value of the element for the given *key*. The *key* must be some comparable type.

`set-value(HashTable h, Comparable key, Element new-value)`

Sets the element of the array for the given *key* to be equal to *new-value*. The *key* must be some comparable type.

`remove(HashTable h, Comparable key)`

Remove the element for the given *key* from the hash table. The *key* must be some comparable type.

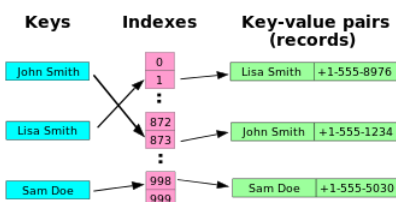


Figure 7.1: A small phone book as a hash table.

[7.1: Time complexity and common uses of hash tables](#)

[7.2: Choosing a good hash function](#)

[7.3: Collision resolution](#)

[7.4: Table resizing](#)

[7.5: Ordered retrieval issue](#)

[7.6: Problems with hash tables](#)

[7.7: Other hash table algorithms](#)

[7.8: Implementations](#)

7: Hash Tables is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

7.1: Time complexity and common uses of hash tables

Hash tables are often used to implement [associative arrays](#), [sets](#) and [caches](#). Like [arrays](#), hash tables provide constant-time $O(1)$ lookup on average, regardless of the number of items in the table. The (hopefully rare) worst-case lookup time in most hash table schemes is $O(n)$.^[1] Compared to other associative array data structures, hash tables are most useful when we need to store a large numbers of data records.

Hash tables may be used as in-memory data structures. Hash tables may also be adopted for use with [persistent data structures](#); database indexes commonly use disk-based data structures based on hash tables.

Hash tables are also used to speed-up string searching in many implementations of [data compression](#).

In [computer chess](#), a hash table can be used to implement the [transposition table](#).

Footnotes

1. ↑ The simplest hash table schemes -- "open addressing with linear probing", "separate chaining with linked lists", etc. -- have $O(n)$ lookup time in the worst case where (accidentally or maliciously) most keys "collide" -- most keys are hashed to one or a few buckets. Other hash table schemes -- "cuckoo hashing", "dynamic perfect hashing", etc. -- guarantee $O(1)$ lookup time even in the worst case. When a new key is inserted, such schemes change their hash function whenever necessary to avoid collisions.

7.1: Time complexity and common uses of hash tables is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

7.2: Choosing a good hash function

A good hash function is essential for good hash table performance. A poor choice of hash function is likely to lead to *clustering* behavior, in which the probability of keys mapping to the same hash bucket (i.e. a *collision*) is significantly greater than would be expected from a random function. A nonzero probability of collisions is inevitable in any hash implementation, but the number of operations to resolve collisions usually scales linearly with the number of keys mapping to the same bucket, so excess collisions will degrade performance significantly. In addition, some hash functions are computationally expensive, so the amount of time (and, in some cases, memory) taken to compute the hash may be burdensome.

Choosing a good hash function is tricky. The literature is replete with poor choices, at least when measured by modern standards. For example, the very popular multiplicative hash advocated by Knuth in [The Art of Computer Programming](#) has particularly poor clustering behavior. However, since poor hashing merely degrades hash table performance for particular input key distributions, such problems go undetected far too often.

The literature is also sparse on the criteria for choosing a hash function. Unlike most other fundamental algorithms and data structures, there is no universal consensus on what makes a "good" hash function. The remainder of this section is organized by three criteria: simplicity, speed, and strength, and will survey algorithms known to perform well by these criteria.

Simplicity and speed are readily measured objectively (by number of lines of code and CPU benchmarks, for example), but strength is a more slippery concept. Obviously, a [cryptographic hash function](#) such as SHA-1 would satisfy the relatively lax strength requirements needed for hash tables, but their slowness and complexity makes them unappealing. In fact, even a cryptographic hash does not provide protection against an adversary who wishes to degrade hash table performance by choosing keys all hashing to the same bucket. For these specialized cases, a universal hash function should be used instead of any one static hash, no matter how sophisticated.

In the absence of a standard measure for hash function strength, the current state of the art is to employ a battery of [statistical](#) tests to measure whether the hash function can be readily distinguished from a random function. Arguably the most important such test is to determine whether the hash function displays the avalanche effect, which essentially states that any single-bit change in the input key should affect on average half the bits in the output. Bret Mulvey advocates testing the *strict avalanche condition* in particular, which states that, for any single-bit change, each of the output bits should change with probability one-half, independent of the other bits in the key. Purely additive hash functions such as CRC fail this stronger condition miserably.

Clearly, a strong hash function should have a uniform distribution of hash values. Bret Mulvey proposes the use of a chi-squared test for uniformity, based on power of two hash table sizes ranging from 2^1 to 2^{16} . This test is considerably more sensitive than many others proposed for measuring hash functions, and finds problems in many popular hash functions.

Fortunately, there are good hash functions that satisfy all these criteria. The simplest class all consume one byte of the input key per iteration of the inner loop. Within this class, simplicity and speed are closely related, as fast algorithms simply don't have time to

perform complex calculations. Of these, one that performs particularly well is the Jenkins One-at-a-time hash, adapted here from [an article by Bob Jenkins](#), its creator.

```

01  uint32 joaat_hash(uchar *key, size_t len)
02  {
03      uint32 hash = 0;
04      size_t i;
05
06      for (i = 0; i < len; i++)
07      {
08          hash += key[i];
09          hash += (hash << 10);
10          hash ^= (hash >> 6);
11      }
12      hash += (hash << 3);
13      hash ^= (hash >> 11);
14      hash += (hash << 15);
15      return hash;
16  }

```

The avalanche behavior of this hash is shown below. The image was made using Bret Mulvey's AvalancheTest in his [Hash.cs toolset](#). Each row corresponds to a single bit in the input, and each column to a bit in the output. A green square indicates good mixing behavior, a yellow square weak mixing behavior, and red would indicate no mixing. Only a few bits in the last byte are weakly mixed, a performance vastly better than a number of widely used hash functions.

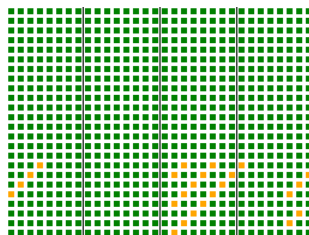


Figure 7.2.1: Avalanche behavior of Jenkins One-at-a-time hash over 3-byte keys

Many commonly used hash functions perform poorly when subjected to such rigorous avalanche testing. The widely favored FNV hash, for example, shows many bits with no mixing at all, especially for short keys. See the [evaluation of FNV](#) by Bret Mulvey for a more thorough analysis.

If speed is more important than simplicity, then the class of hash functions which consume multibyte chunks per iteration may be of interest. One of the most sophisticated is "lookup3" by Bob Jenkins, which consumes input in 12 byte (96 bit) chunks. Note, though, that any speed improvement from the use of this hash is only likely to be useful for large keys, and that the increased complexity may also have speed consequences such as preventing an optimizing compiler from inlining the hash function. Bret Mulvey analyzed an [earlier version, lookup2](#), and found it to have excellent avalanche behavior.

One desirable property of a hash function is that conversion from the hash value (typically 32 bits) to an bucket index for a particular-size hash table can be done simply by masking, preserving only the lower k bits for a table of size 2^k (an operation equivalent to computing the hash value modulo the table size). This property enables the technique of incremental doubling of the size of the hash table - each bucket in the old table maps to only two in the new table. Because of its use of XOR-folding, the FNV hash does not have this property. Some older hashes are even worse, requiring table sizes to be a prime number rather than a power of two, again computing the bucket index as the hash value modulo the table size. In general, such a requirement is a sign of a fundamentally weak function; using a prime table size is a poor substitute for using a stronger function.

7.2: Choosing a good hash function is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

7.3: Collision resolution

If two keys hash to the same index, the corresponding records cannot be stored in the same location. So, if it's already occupied, we must find another location to store the new record, and do it so that we can find it when we look it up later on.

To give an idea of the importance of a good collision resolution strategy, consider the following result, derived using the birthday paradox. Even if we assume that our hash function outputs random indices uniformly distributed over the array, and even for an array with 1 million entries, there is a 95% chance of at least one collision occurring before it contains 2500 records.

There are a number of collision resolution techniques, but the most popular are *chaining* and *open addressing*.

Chaining

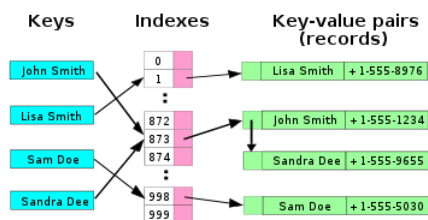


Figure 7.3.1: Hash collision resolved by chaining. (Public Domain; via Wikimedia Commons)

In the simplest chained hash table technique, each slot in the array references a linked list of inserted records that collide to the same slot. Insertion requires finding the correct slot, and appending to either end of the list in that slot; deletion requires searching the list and removal.

Chained hash tables have advantages over open addressed hash tables in that the removal operation is simple and resizing the table can be postponed for a much longer time because performance degrades more gracefully even when every slot is used. Indeed, many chaining hash tables may not require resizing at all since performance degradation is linear as the table fills. For example, a chaining hash table containing twice its recommended capacity of data would only be about twice as slow on average as the same table at its recommended capacity.

Chained hash tables inherit the disadvantages of linked lists. When storing small records, the overhead of the linked list can be significant. An additional disadvantage is that traversing a linked list has poor cache performance.

Alternative data structures can be used for chains instead of linked lists. By using a self-balancing tree, for example, the theoretical worst-case time of a hash table can be brought down to $O(\log n)$ rather than $O(n)$. However, since each list is intended to be short, this approach is usually inefficient unless the hash table is designed to run at full capacity or there are unusually high collision rates, as might occur in input designed to cause collisions. Dynamic arrays can also be used to decrease space overhead and improve cache performance when records are small.

Some chaining implementations use an optimization where the first record of each chain is stored in the table. Although this can increase performance, it is generally not recommended: chaining tables with reasonable load factors contain a large proportion of empty slots, and the larger slot size causes them to waste large amounts of space.

Open addressing

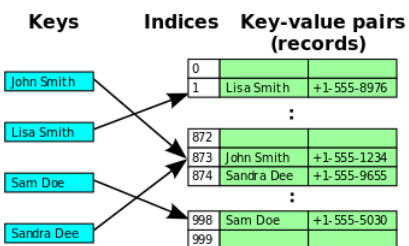


Figure 7.3.2: Hash collision resolved by linear probing (interval=1). (Public Domain; via Wikimedia Commons)

Open addressing hash tables can store the records directly within the array. A hash collision is resolved by *probing*, or searching through alternate locations in the array (the *probe sequence*) until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. Well known probe sequences include:

linear probing

in which the interval between probes is fixed—often at 1,

quadratic probing

in which the interval between probes increases linearly (hence, the indices are described by a quadratic function), and

double hashing

in which the interval between probes is fixed for each record but is computed by another hash function.

The main tradeoffs between these methods are that linear probing has the best cache performance but is most sensitive to clustering, while double hashing has poor cache performance but exhibits virtually no clustering; quadratic hashing falls in-between in both areas. Double hashing can also require more computation than other forms of probing. Some open addressing methods, such as last-come-first-served hashing and cuckoo hashing move existing keys around in the array to make room for the new key. This gives better maximum search times than the methods based on probing.

A critical influence on performance of an open addressing hash table is the *load factor*; that is, the proportion of the slots in the array that are used. As the load factor increases towards 100%, the number of probes that may be required to find or insert a given key rises dramatically. Once the table becomes full, probing algorithms may even fail to terminate. Even with good hash functions, load factors are normally limited to 80%. A poor hash function can exhibit poor performance even at very low load factors by generating significant clustering. What causes hash functions to cluster is not well understood, and it is easy to unintentionally write a hash function which causes severe clustering.

Example pseudocode

The following pseudocode is an implementation of an open addressing hash table with linear probing and single-slot stepping, a common approach that is effective if the hash function is good. Each of the **lookup**, **set** and **remove** functions use a common internal function **findSlot** to locate the array slot that either does or should contain a given key.

```
record pair { key, value }
var pair array slot[0..numSlots-1]

function findSlot(key)
    i := hash(key) modulus numSlots
    loop
        if slot[i] is not occupied or slot[i].key = key
            return i
        i := (i + 1) modulus numSlots

function lookup(key)
    i := findSlot(key)
    if slot[i] is occupied // key is in table
        return slot[i].value
    else // key is not in table
        return not found

function set(key, value)
    i := findSlot(key)
    if slot[i].key = key
```

```
// (Key already in table. Update value.)
slot[i].value := value
else
  // (Insert key and value in un-occupied slot.)
  // (But first, make sure insert won't overload the table)
  if the table is almost full
    rebuild the table larger (note 1)
    i := findSlot(key)
  slot[i].key := key
  slot[i].value := value
```

Another example showing open addressing technique. Presented function is converting each part(4) of an Internet protocol address, where NOT is bitwise NOT, XOR is bitwise XOR, OR is bitwise OR, AND is bitwise AND and << and >> are shift-left and shift-right:

```
// key_1, key_2, key_3, key_4 are following 3-digit numbers - parts of ip address xxx.x
function ip(key parts)
  j := 1
  do
    key := (key_2 << 2)
    key := (key + (key_3 << 7))
    key := key + (j OR key_4 >> 2) * (key_4) * (j + key_1) XOR j
    key := key AND _prime_ // _prime_ is a prime number
    j := (j+1)
  while collision
  return key
```

Note 1

Rebuilding the table requires allocating a larger array and recursively using the **set** operation to insert all the elements of the old array into the new larger array. It is common to increase the array size exponentially, for example by doubling the old array size.

```
function remove(key)
  i := findSlot(key)
  if slot[i] is unoccupied
    return // key is not in the table
  j := i
  loop
    j := (j+1) modulus numSlots
    if slot[j] is unoccupied
      exit loop
  k := hash(slot[j].key) modulus numSlots
  if (j > i and (k <= i or k > j)) or
    (j < i and (k <= i and k > j)) (note 2)
    slot[i] := slot[j]
```

```
i := j
mark slot[i] as unoccupied
```

Note 2

For all records in a cluster, there must be no vacant slots between their natural hash position and their current position (else lookups will terminate before finding the record). At this point in the pseudocode, i is a vacant slot that might be invalidating this property for subsequent records in the cluster. j is such as subsequent record. k is the raw hash where the record at j would naturally land in the hash table if there were no collisions. This test is asking if the record at j is invalidly positioned with respect to the required properties of a cluster now that i is vacant.

Another technique for removal is simply to mark the slot as deleted. However this eventually requires rebuilding the table simply to remove deleted records. The methods above provide $O(1)$ updating and removal of existing records, with occasional rebuilding if the high water mark of the table size grows.

The $O(1)$ remove method above is only possible in linearly probed hash tables with single-slot stepping. In the case where many records are to be deleted in one operation, marking the slots for deletion and later rebuilding may be more efficient.

Open addressing versus chaining

Chained hash tables have the following benefits over open addressing:

- They are simple to implement effectively and only require basic data structures.
- From the point of view of writing suitable hash functions, chained hash tables are insensitive to clustering, only requiring minimization of collisions. Open addressing depends upon better hash functions to avoid clustering. This is particularly important if novice programmers can add their own hash functions, but even experienced programmers can be caught out by unexpected clustering effects.
- They degrade in performance more gracefully. Although chains grow longer as the table fills, a chained hash table cannot "fill up" and does not exhibit the sudden increases in lookup times that occur in a near-full table with open addressing. (see right)
- If the hash table stores large records, about 5 or more words per record, chaining uses less memory than open addressing.
- If the hash table is sparse (that is, it has a big array with many free array slots), chaining uses less memory than open addressing even for small records of 2 to 4 words per record due to its external storage.

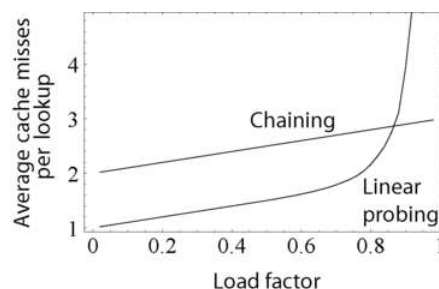


Figure 7.3.3: This graph compares the average number of cache misses required to lookup elements in tables with chaining and linear probing. As the table passes the 80%-full mark, linear probing's performance drastically degrades. (Public Domain; by Derrick Coetzee via Wikimedia Commons)

For small record sizes (a few words or less) the benefits of in-place open addressing compared to chaining are:

- They can be more space-efficient than chaining since they don't need to store any pointers or allocate any additional space outside the hash table. Simple linked lists require a word of overhead per element.
- Insertions avoid the time overhead of memory allocation, and can even be implemented in the absence of a memory allocator.
- Because it uses internal storage, open addressing avoids the extra indirection required for chaining's external storage. It also has better locality of reference, particularly with linear probing. With small record sizes, these factors can yield better performance than chaining, particularly for lookups.
- They can be easier to serialize, because they don't use pointers.

On the other hand, normal open addressing is a poor choice for large elements, since these elements fill entire cache lines (negating the cache advantage), and a large amount of space is wasted on large empty table slots. If the open addressing table only stores references to elements (external storage), it uses space comparable to chaining even for large records but loses its speed advantage.

Generally speaking, open addressing is better used for hash tables with small records that can be stored within the table (internal storage) and fit in a cache line. They are particularly suitable for elements of one word or less. In cases where the tables are expected to have high load factors, the records are large, or the data is variable-sized, chained hash tables often perform as well or better.

Ultimately, used sensibly any kind of hash table algorithm is usually fast *enough*; and the percentage of a calculation spent in hash table code is low. Memory usage is rarely considered excessive. Therefore, in most cases the differences between these algorithms is marginal, and other considerations typically come into play.

Coalesced hashing

A hybrid of chaining and open addressing, coalesced hashing links together chains of nodes within the table itself. Like open addressing, it achieves space usage and (somewhat diminished) cache advantages over chaining. Like chaining, it does not exhibit clustering effects; in fact, the table can be efficiently filled to a high density. Unlike chaining, it cannot have more elements than table slots.

Perfect hashing

If all of the keys that will be used are known ahead of time, and there are no more keys that can fit the hash table, perfect hashing can be used to create a perfect hash table, in which there will be no collisions. If minimal perfect hashing is used, every location in the hash table can be used as well.

Perfect hashing gives a hash table where the time to make a lookup is constant in the worst case. This is in contrast to chaining and open addressing methods, where the time for lookup is low on average, but may be arbitrarily large. There exist methods for maintaining a perfect hash function under insertions of keys, known as dynamic perfect hashing. A simpler alternative, that also gives worst case constant lookup time, is cuckoo hashing.

Probabilistic hashing

Perhaps the simplest solution to a collision is to replace the value that is already in the slot with the new value, or slightly less commonly, drop the record that is to be inserted. In later searches, this may result in a search not finding a record which has been inserted. This technique is particularly useful for implementing caching.

An even more space-efficient solution which is similar to this is use a bit array (an array of one-bit fields) for our table. Initially all bits are set to zero, and when we insert a key, we set the corresponding bit to one. False negatives cannot occur, but false positives can, since if the search finds a 1 bit, it will claim that the value was found, even if it was just another value that hashed into the same array slot by coincidence. In reality, such a hash table is merely a specific type of Bloom filter.

7.3: Collision resolution is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- **9.3: Collision resolution** by Wikibooks - Data Structures is licensed [CC BY-SA 4.0](#).

7.4: Table resizing

With a good hash function, a hash table can typically contain about 70%–80% as many elements as it does table slots and still perform well. Depending on the collision resolution mechanism, performance can begin to suffer either gradually or dramatically as more elements are added. To deal with this, when the load factor exceeds some threshold, we allocate a new, larger table, and add all the contents of the original table to this new table. In Java's `HashMap` class, for example, the default load factor threshold is 0.75.

This can be a very expensive operation, and the necessity for it is one of the hash table's disadvantages. In fact, some naive methods for doing this, such as enlarging the table by one each time you add a new element, reduce performance so drastically as to make the hash table useless. However, if we enlarge the table by some fixed percent, such as 10% or 100%, it can be shown using amortized analysis that these resizings are so infrequent that the average time per lookup remains constant-time. To see why this is true, suppose a hash table using chaining begins at the minimum size of 1 and is doubled each time it fills above 100%. If in the end it contains n elements, then the total add operations performed for all the resizings is:

$$1 + 2 + 4 + \dots + n = 2n - 1.$$

Because the costs of the resizings form a geometric series, the total cost is $O(n)$. But we also perform n operations to add the n elements in the first place, so the total time to add n elements with resizing is $O(n)$, an amortized time of $O(1)$ per element.

On the other hand, some hash table implementations, notably in real-time systems, cannot pay the price of enlarging the hash table all at once, because it may interrupt time-critical operations. One simple approach is to initially allocate the table with enough space for the expected number of elements and forbid the addition of too many elements. Another useful but more memory-intensive technique is to perform the resizing gradually:

- Allocate the new hash table, but leave the old hash table and check both tables during lookups.
- Each time an insertion is performed, add that element to the new table and also move k elements from the old table to the new table.
- When all elements are removed from the old table, deallocate it.

To ensure that the old table will be completely copied over before the new table itself needs to be enlarged, it's necessary to increase the size of the table by a factor of at least $(k + 1)/k$ during the resizing.

Linear hashing is a hash table algorithm that permits incremental hash table expansion. It is implemented using a single hash table, but with two possible look-up functions.

Another way to decrease the cost of table resizing is to choose a hash function in such a way that the hashes of most values do not change when the table is resized. This approach, called consistent hashing, is prevalent in disk-based and distributed hashes, where resizing is prohibitively costly.

7.4: Table resizing is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- [9.4: Table resizing](#) by Wikibooks - Data Structures is licensed [CC BY-SA 4.0](#).

7.5: Ordered retrieval issue

Hash tables store data in pseudo-random locations, so accessing the data in a sorted manner is a very time consuming operation. Other data structures such as self-balancing binary search trees generally operate more slowly (since their lookup time is $O(\log n)$) and are rather more complex to implement than hash tables but maintain a sorted data structure at all times. See a comparison of hash tables and self-balancing binary search trees.

7.5: Ordered retrieval issue is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- **9.5: Ordered retrieval issue** by [Wikibooks - Data Structures](#) is licensed [CC BY-SA 4.0](#).

7.6: Problems with hash tables

Although hash table lookups use constant time on average, the time spent can be significant. Evaluating a good hash function can be a slow operation. In particular, if simple array indexing can be used instead, this is usually faster.

Hash tables in general exhibit poor locality of reference—that is, the data to be accessed is distributed seemingly at random in memory. Because hash tables cause access patterns that jump around, this can trigger microprocessor cache misses that cause long delays. Compact data structures such as arrays, searched with linear search, may be faster if the table is relatively small and keys are cheap to compare, such as with simple integer keys. According to Moore's Law, cache sizes are growing exponentially and so what is considered "small" may be increasing. The optimal performance point varies from system to system; for example, a trial on Parrot shows that its hash tables outperform linear search in all but the most trivial cases (one to three entries).

More significantly, hash tables are more difficult and error-prone to write and use. Hash tables require the design of an effective hash function for each key type, which in many situations is more difficult and time-consuming to design and debug than the mere comparison function required for a self-balancing binary search tree. In open-addressed hash tables it's even easier to create a poor hash function.

Additionally, in some applications, a black hat with knowledge of the hash function may be able to supply information to a hash which creates worst-case behavior by causing excessive collisions, resulting in very poor performance (i.e., a denial of service attack). In critical applications, either universal hashing can be used or a data structure with better worst-case guarantees may be preferable. For details, see Crosby and Wallach's [Denial of Service via Algorithmic Complexity Attacks](#).

7.6: Problems with hash tables is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- **9.6: Problems with hash tables** by Wikibooks - Data Structures is licensed [CC BY-SA 4.0](#).

7.7: Other hash table algorithms

Extendible hashing and **linear hashing** are hash algorithms that are used in the context of database algorithms used for instance in index file structures, and even primary file organization for a database. Generally, in order to make search scalable for large databases, the search time should be proportional $\log N$ or near constant, where N is the number of records to search. $\log N$ searches can be implemented with tree structures, because the degree of fan out and the shortness of the tree relates to the number of steps needed to find a record, so the height of the tree is the maximum number of disc accesses it takes to find where a record is. However, hash tables are also used, because the cost of a disk access can be counted in units of disc accesses, and often that unit is a block of data. Since a hash table can, in the best case, find a key with one or two accesses, a hash table index is regarded as generally faster when retrieving a collection of records during a join operation e.g.

```
SELECT * from customer, orders where customer.cust_id = orders.cust_id and cust_id = X
```

i.e. If orders has a hash index on `cust_id`, then it takes constant time to locate the block that contains record locations for orders matching `cust_id = X`. (although, it would be better if the value type of orders was a list of order ids, so that hash keys are just one unique `cust_id` for each batch of orders, to avoid unnecessary collisions).

Extendible hashing and linear hashing have certain similarities: collisions are accepted as inevitable and are part of the algorithm where blocks or buckets of collision space is added; traditional good hash function ranges are required, but the hash value is transformed by a dynamic address function: in **extendible hashing**, a bit mask is used to mask out unwanted bits, but this mask length increases by one periodically, doubling the available addressing space; also in extendible hashing, there is an indirection with a directory address space, the directory entries being paired with another address (a pointer) to the actual block containing the key-value pairs; the entries in the directory correspond to the bit masked hash value (so that the number of entries is equal to maximum bit mask value + 1 e.g. a bit mask of 2 bits, can address a directory of 00 01 10 11, or $3 + 1 = 4$).

In **linear hashing**, the traditional hash value is also masked with a bit mask, but if the resultant smaller hash value falls below a 'split' variable, the original hash value is masked with a bit mask of one bit greater length, making the resultant hash value address recently added blocks. The split variable ranges incrementally between 0 and the maximum current bit mask value e.g. a bit mask of 2, or in the terminology of linear hashing, a "level" of 2, the split variable will range between 0 and 3. When the split variable reaches 4, the level increases by 1, so in the next round of the split variable, it will range between 0 and 7, and reset again when it reaches 8.

The split variable incrementally allows increased addressing space, as new blocks are added; the decision to add a new block occurs whenever a key-and-value is being inserted, and overflows the particular block the key-and-value's key hashes into. This overflow location may be completely unrelated to the block going to be split pointed to by the split variable. However, over time, it is expected that given a good random hash function that distributes entries fairly evenly amongst all addressable blocks, the blocks that actually require splitting because they have overflowed get their turn in round-robin fashion as the split value ranges between 0 - N where N has a factor of 2 to the power of Level, level being the variable incremented whenever the split variable hits N .

New blocks are added one at a time with both extendible hashing, and with linear hashing.

In **extendible hashing**, a block overflow (a new key-value colliding with B other key-values, where B is the size of a block) is handled by checking the size of the bit mask "locally", called the "local depth", an attribute which must be stored with the block. The directory structure, also has a depth, the "global depth". If the local depth is less than the global depth, then the local depth is incremented, and all the key values are rehashed and passed through a bit mask which is one bit longer now, placing them either in the current block, or in another block. If the other block happens to be the same block when looked up in the directory, a new block is added, and the directory entry for the other block is made to point to the new block. Why does the directory have entries where two entries point to the same block ? This is because if the local depth is **equal to** the global depth of the directory, this means the bit mask of the directory does not have enough bits to deal with an increment in the bit mask length of the block, and so the directory must have its bit mask length incremented, but this means the directory now doubles the number of addressable entries. Since half the entries addressable don't exist, the directory simply copies the pointers over to the new entries e.g. if the directory had entries for 00, 01, 10, 11, or a 2 bit mask, and it becomes a 3 bit mask, then 000 001 010 011 100 101 110 111 become the new entries, and 00's block address go to 000 and 001 ; 01's pointer goes to 010 and 011, 10 goes to 100 and 101 and so on. And so this

creates the situation where two directory entries point to the same block. Although the block that was going to overflow, now can add a new block by redirecting the second pointer to a newly appended block, the other original blocks will have two pointers to them. When it is their turn to split, the algorithm will check local vs global depth and this time find that the local depth is less, and hence no directory splitting is required, only a new block be appended, and the second directory pointer moved from addressing the previous block to addressing the new block.

In **linear hashing**, adding a similarly hashed block does not occur immediately when a block overflows, and therefore an overflow block is created to be attached to the overflowing block. However, a block overflow is a signal that more space will be required, and this happens by splitting the block pointed to by the "split" variable, which is initially zero, and hence initially points to block zero. The splitting is done by taking all the key-value pairs in the splitting block, and its overflow block(s), hashing the keys again, but with a bit mask of length current level + 1. This will result in two block addresses, some will be the old block number, and others will be

$$a2 = \text{old block number} + (N \times 2^{\text{level}})$$

Rationale

Let $m = N \times 2^{\text{level}}$; if h is the original hash value, and old block number $= h \bmod m$, and now the new block number is $h \bmod (m \times 2)$, because $m \times 2 = N \times 2^{(\text{level}+1)}$, then the new block number is either $h \bmod m$ if (h / m) is even so dividing h/m by 2 leaves a zero remainder and therefore doesn't change the remainder, or the new block number is $(h \bmod m) + m$ because h / m is an odd number, and dividing h / m by 2 will leave an excess remainder of m , + the original remainder. (The same rationale applies to extendible hashing depth incrementing).

As above, a new block is created with a number $a2$, which will usually occur at +1 the previous $a2$ value. Once this is done, the split variable is incremented, so that the next $a2$ value will be again old $a2 + 1$. In this way, each block is covered by the split variable eventually, so each block is preemptively rehashed into extra space, and new blocks are added incrementally. Overflow blocks that are no longer needed are discarded, for later garbage collection if needed, or put on an available free block list by chaining.

When the split variable reaches $(N \times 2^{\text{level}})$, level is incremented and split variable is reset to zero. In this next round, the split variable will now traverse from zero to $(N \times 2^{(\text{old\_level} + 1)})$, which is exactly the number of blocks at the start of the previous round, but including all the blocks created by the previous round.

A simple inference on file storage mapping of linear hashing and extendible hashing

As can be seen, extendible hashing requires space to store a directory which can double in size.

Since the space of both algorithms increase by one block at a time, if blocks have a known maximum size or fixed size, then it is straight forward to map the blocks as blocks sequentially appended to a file.

In extendible hashing, it would be logical to store the directory as a separate file, as doubling can be accommodated by adding to the end of the directory file. The separate block file would not have to change, other than have blocks appended to its end.

Header information for linear hashing doesn't increase in size : basically just the values for N , level, and split need to be recorded, so these can be incorporated as a header into a fixed block size linear hash storage file.

However, linear hashing requires space for overflow blocks, and this might best be stored in another file, otherwise addressing blocks in the linear hash file is not as straight forward as multiplying the block number by the block size and adding the space for N , level, and split.

In the next section, a complete example of linear hashing in Java is given, using a in memory implementation of linear hashing, and code to manage blocks as files in a file directory, the whole contents of the file directory representing the persistent linear hashing structure.

7.7: Other hash table algorithms is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

- 9.7: Other hash table algorithms by Wikibooks - Data Structures is licensed [CC BY-SA 4.0](#).

7.8: Implementations

While many programming languages already provide hash table functionality,^[1] there are several independent implementations worth mentioning.

- [Google Sparse Hash](#) The Google SparseHash project contains several hash-map implementations in use at Google, with different performance characteristics, including an implementation that optimizes for space and one that optimizes for speed. The memory-optimized one is extremely memory-efficient with only 2 bits/entry of overhead.
- [MCT](#) provides hashtables similar to Google's `dense_hash_map` , but without restriction on contained values; it also comes with exception safety and support for C++0x features.
- A number of runtime languages and/or standard libraries use hash tables to implement their support for associative arrays because of their efficiency.

A python implementation of extendible hashing

The file - block management routines aren't there, so this could be added in to make this a real implementation of a database hash index.

A full page is split according to the (local depth)th bit, first by collecting all the directory indices pointing to the full page, updating the pointers according to the d bit being 0 or 1 corresponding to first and second new pages, then reloading all the key-values after

hashing each key and using the d bit of each hash to see which page to allocate to. The local depth of each new page is one greater than the old page's local depth, so that the next d bit can be used next time when splitting.

```
01 PAGE_SZ = 20
02
03 class Page:
04
05     def __init__(self):
06         self.m = {}
07         self.d = 0
08
09     def full(self):
10         return len(self.m) > PAGE_SZ
11
12     def put(self, k, v):
13         self.m[k] = v
14
15     def get(self, k):
16         return self.m.get(k)
17
18 class EH:
19
20     def __init__(self):
21         self.gd = 0
22         p = Page()
23         self.pp = [p]
24
25     def get_page(self, k):
26         h = hash(k)
27         p = self.pp[ h & (( 1 << self.gd) - 1)]
28         return p
29
30     def put(self, k, v):
31         p = self.get_page(k)
32         if p.full() and p.d == self.gd:
33             self.pp = self.pp + self.pp
34             self.gd += 1
35
36         if p.full() and p.d < self.gd:
37             p.put(k, v);
38             p1 = Page()
39             p2 = Page()
40             for k2 in p.m.keys():
41                 v2 = p.m[k2]
42                 h = k2.__hash__()
43                 h = h & ((1 << self.gd) - 1)
44                 if (h | (1 << p.d) == h):
45                     p2.put(k2, v2)
46                 else:
47                     p1.put(k2, v2)
48             l = []
49             for i in xrange(0, len(self.pp)):
50                 if self.pp[i] == p:
51                     l.append(i)
52             for i in l:
53                 if (i | ( 1 << p.d) == i):
54                     self.pp[i] = p2
55             else:
56                 self.pp[i] = p1
57
58     else:
59         self.pp[i] = p1
```

```
60         p1.d = p.d + 1
61         p2.d = p1.d
62     else:
63         p.put(k, v)
64
65     def get(self, k):
66         p = self.get_page(k)
67         return p.get(k)
68
69
70
71 if __name__ == "__main__":
72     eh = EH()
73     N = 10000
74     l = []
75     for i in range(0, N):
76         l.append(i)
77
78     import random
79     random.shuffle(l)
80     for i in l:
81         eh.put(i, i)
82     print l
83
84     for i in range(0, N):
85         print eh.get(i)
```

A Java implementation of extendible hashing

A direct Java translation of the above python code, tested to work.

```
001 package ext_hashing;
002
003 import java.util.ArrayList;
004 import java.util.HashMap;
005 import java.util.List;
006 import java.util.Map;
007 import java.util.Random;
008
009 public class EH2<K, V> {
010     static class Page<K, V> {
011         static int PAGE_SZ = 20;
012         private Map<K, V> m = new HashMap<K, V>();
013         int d = 0;
014
015         boolean full() {
016             return m.size() > PAGE_SZ;
017         }
018
019         void put(K k, V v) {
020             m.put(k, v);
021         }
022     }
023
024     V get(K k) {
025         return m.get(k);
026     }
027 }
028
029 int gd = 0;
030
031 List<Page<K, V>> pp = new ArrayList<Page<K, V>>();
032
033 public EH2() {
034     pp.add(new Page<K, V>());
035 }
036
037 Page<K, V> getPage(K k) {
038     int h = k.hashCode();
039     Page<K, V> p = pp.get(h & ((1 << gd) - 1));
040     return p;
041 }
042
043 void put(K k, V v) {
044     Page<K, V> p = getPage(k);
045     if (p.full() && p.d == gd) {
046         List<Page<K, V>> pp2 = new ArrayList<EH2.Page<K, V>>(pp);
047         pp.addAll(pp2);
048         ++gd;
049     }
050
051     if (p.full() && p.d < gd) {
052         p.put(k, v);
053         Page<K, V> p1, p2;
054         p1 = new Page<K, V>();
055         p2 = new Page<K, V>();
056         for (K k2 : p.m.keySet()) {
057             V v2 = p.m.get(k2);
058
059             int h = k2.hashCode() & ((1 << gd) - 1);
```

```
060
061         if ((h | (1 << p.d)) == h)
062             p2.put(k2, v2);
063         else
064             p1.put(k2, v2);
065     }
066
067     List<Integer> l = new ArrayList<Integer>();
068
069     for (int i = 0; i < pp.size(); ++i)
070         if (pp.get(i) == p)
071             l.add(i);
072
073     for (int i : l)
074         if ((i | (1 << p.d)) == i)
075             pp.set(i, p2);
076         else
077             pp.set(i, p1);
078
079     p1.d = p.d + 1;
080     p2.d = p1.d;
081
082     } else
083         p.put(k, v);
084 }
085
086 public V get(K k) {
087     return getPage(k).get(k);
088 }
089
090 public static void main(String[] args) {
091
092     int N = 500000;
093
094     Random r = new Random();
095     List<Integer> l = new ArrayList<Integer>();
096     for (int i = 0; i < N; ++i) {
097         l.add(i);
098     }
099
100     for (int i = 0; i < N; ++i) {
101         int j = r.nextInt(N);
102         int t = l.get(j);
103         l.set(j, l.get(i));
104         l.set(i, t);
105     }
106
107     EH2<Integer, Integer> eh = new EH2<Integer, Integer>();
108     for (int i = 0; i < N; ++i) {
109         eh.put(l.get(i), l.get(i));
110     }
111
112     for (int i = 0; i < N; ++i) {
113         System.out.printf("%d:%d ", i, eh.get(i));
114         if (i % 10 == 0)
115             System.out.println();
116     }
117 }
118 }
119 }
```

A Java implementation of linear hashing

(usable for simple database indexing of arbitrary size, (almost ?))

This code evolved out of a need for a bigger Java HashMap. Originally, a Java HashMap standard object was used as a Map to index a heap like database file (DBF format). But then, a file was encountered with so many records that the OutOfMemoryError was being thrown, so linear hashing seemed like a reasonably simple algorithm to use as a disk based index scheme.

Initially, linear hashing was implemented behind the Java Map interface, mainly the put(k,v) and get(k,v) methods. Generics was used, in order not to get too involved in the key and value details.

debugging to achieve functionality

Custom dumps to System.err was used to verify that blocks were being created, filled, and overflow blocks were of expected number if they existed (no. = 1). This was all done in a purely in-memory implementation.

Later, the standard Java decoupling was introduced, where the original LHMap2 class accepted listeners to events, such as when a block was needed. The listener was then implemented as a block file manager, loading blocks into the block list of the memory LHMap2 object whenever a null block was encountered on the block list, and checking whether virtual machine runtime free memory was low, and retiring blocks using just a basic first-to-last-listed cache removal algorithm (not least recently used (LRU), not least often used) by saving them to disk file, and then actively invoking the system garbage collector.

Because an application use case existed, which was to externally index a large DBF table, this use case was used as the main test

harness for the algorithm implementation.

```
001 package linearhashmap;
002
003 import java.io.Externalizable;
004 import java.io.IOException;
005 import java.io.ObjectInput;
006 import java.io.ObjectOutput;
007 import java.io.Serializable;
008 import java.util.ArrayList;
009 import java.util.Collection;
010 import java.util.Iterator;
011 import java.util.LinkedList;
012 import java.util.List;
013 import java.util.Map;
014 import java.util.Set;
015 import java.util.TreeMap;
016 import java.util.TreeSet;
017
018 /**
019  *
020  * @param <K>
021  *      key type , must implement equals() and hashCode()
022  * @param <V>
023  *      value type
024  *
025  */
026 */
027 public class LHMap2<K, V> implements Map<K, V>, Serializable {
028
029     /**
030      *
031      */
032     private static final long serialVersionUID = 3095071852466632996L;
033
034     /**
035      *
036      */
037
038     public static interface BlockListener<K, V> {
039         public void blockRequested(int block, LHMap2<K, V> map);
040     }
041
042     List<BlockListener<K, V>> listeners = new ArrayList<BlockListener<K, V>>();
043
044     // int savedBlocks;
045     int N;
046     int level = 0;
047     int split = 0;
048     int blockSize;
049     long totalWrites = 0;
050
051     List<Block<K, V>> blockList = new ArrayList<Block<K, V>>();
052
053     public void addBlockListener(BlockListener<K, V> listener) {
054         listeners.add(listener);
055     }
056
057     void notifyBlockRequested(int block) {
058         for (BlockListener<K, V> l : listeners) {
059             l.blockRequested(block, this);
060         }
061     }
```

```
062
063     public LHMap2(int blockSize, int nStartingBlocks) {
064         this.blockSize = blockSize;
065         this.N = nStartingBlocks;
066         for (int i = 0; i < nStartingBlocks; ++i) {
067             addBlock();
068         }
069
070         Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
071
072             @Override
073             public void run() {
074                 showStructure();
075             }
076         }));
077     }
078
079     public static class Block<K, V> implements Externalizable {
080         /**
081          *
082          */
083
084         int j = 0;
085
086         Block<K, V> overflow = null;
087         LinkedList<K> keyList = new LinkedList<K>();
088         LinkedList<V> valueList = new LinkedList<V>();
089         transient private LHMap2<K, V> owner;
090         transient private Map<K, V> shadow = new TreeMap<K, V>();
091
092         private boolean changed = false;
093
094         private int size = 0;
095
096         public LHMap2<K, V> getOwner() {
097             return owner;
098         }
099
100         public void setOwner(LHMap2<K, V> owner) {
101             this.owner = owner;
102             Block<K, V> ov = overflow;
103             while (ov != null) {
104                 overflow.setOwner(owner);
105                 ov = ov.overflow;
106             }
107         }
108
109         public Block() {
110             super();
111         }
112
113         public Block(LHMap2<K, V> map) {
114             setOwner(map);
115         }
116
117         public V put(K k, V v) {
118             setChanged(true);
119
120             V v2 = replace(k, v);
121             if (v2 == null) {
122                 ++size;
123                 if (keyList.size() == getOwner().blockSize) {
124
```

```
125
126         if (overflow == null) {
127             getOwner().blockOverflowed(this, k, v);
128         } else {
129             overflow.put(k, v);
130         }
131
132     } else {
133         keyList.addFirst(k);
134         valueList.addFirst(v);
135     }
136
137     }
138
139     return v2;
140 }
141
142 void setChanged(boolean b) {
143     changed = b;
144 }
145
146 public Map<K, V> drainToMap(Map<K, V> map) {
147
148     while (!keyList.isEmpty()) {
149         K k = keyList.removeLast();
150         V v = valueList.removeLast();
151         map.put(k, v);
152     }
153
154     if (overflow != null)
155
156         map = overflow.drainToMap(map);
157
158     garbageCollectionOverflow();
159
160     return map;
161 }
162
163
164 public void updateMap(Map<K, V> map) {
165     Iterator<K> ik = keyList.descendingIterator();
166     Iterator<V> iv = valueList.descendingIterator();
167     while (ik.hasNext() && iv.hasNext()) {
168         map.put(ik.next(), iv.next());
169     }
170
171     if (overflow != null)
172         overflow.updateMap(map);
173
174 }
175
176 private void garbageCollectionOverflow() {
177     if (overflow != null) {
178         overflow.garbageCollectionOverflow();
179         overflow = null;
180     }
181 }
182
183
184 public void addOverflowBucket() {
185
186     // assert overflow is needed
187     if (keyList.size() < getOwner().blockSize)
```

```
188         return;
189
190         if (overflow == null) {
191             overflow = new Block<K, V>(getOwner());
192         } else {
193             overflow.addOverflowBucket();
194         }
195     }
196
197     public V replace(K key, V v2) {
198
199         if (overflow != null) {
200             V v = overflow.replace(key, v2);
201             if (v != null)
202                 return v;
203         }
204
205         Iterator<K> i = keyList.listIterator();
206
207         int j = 0;
208
209         while (i.hasNext()) {
210
211             if (key.equals(i.next())) {
212
213                 V v = valueList.get(j);
214
215                 if (v2 != null) {
216                     valueList.set(j, v2);
217
218                 }
219
220                 return v;
221             }
222             ++j;
223         }
224
225         return null;
226     }
227
228     public boolean isChanged() {
229         return changed;
230     }
231
232     @Override
233     public void readExternal(ObjectInput arg0) throws IOException,
234         ClassNotFoundException {
235         int sz = arg0.readInt();
236         for (int i = 0; i < sz; ++i) {
237             K k = (K) arg0.readObject();
238             V v = (V) arg0.readObject();
239             shadow.put(k, v);
240         }
241     }
242
243     public void loadFromShadow(LHMap2<K, V> owner) {
244         setOwner(owner);
245         Block<K, V> b = this;
246         for (K k : shadow.keySet()) {
247             if (b.keyList.size() == owner.blockSize) {
248                 Block<K, V> overflow = new Block<K, V>(owner);
249                 b.overflow = overflow;
250             }
```

```
251         b = overflow;
252     }
253     b.keyList.add(k);
254     b.valueList.add(shadow.get(k));
255
256     }
257     shadow.clear();
258 }
259
260 @Override
261 public void writeExternal(ObjectOutput arg0) throws IOException {
262     if (!changed)
263         return;
264     Map<K, V> map = new TreeMap<K, V>();
265
266     updateMap(map);
267     int sz = map.size();
268     arg0.writeInt(sz);
269     for (K k : map.keySet()) {
270         arg0.writeObject(k);
271         arg0.writeObject(map.get(k));
272     }
273     setChanged(false);
274
275 }
276
277 }
278
279 void init() {
280
281     for (int i = 0; i < N; ++i) {
282         addBlock();
283     }
284 }
285
286 /**
287  * @param hashValue
288  * @return a bucket number.
289  */
290 int getDynamicHash(int hashValue) {
291
292     long unsignedHash = ((long) hashValue << 32) >>> 32;
293     // ^^ this long cast needed
294     int h = (int) (unsignedHash % (N << level));
295     // System.err.println("h = " + h);
296     if (h < split) {
297
298         h = (int) (unsignedHash % (N << (level + 1)));
299         // System.err.println("h < split, new h = " + h);
300     }
301     return h;
302
303 }
304
305 @Override
306 public V put(K k, V v) {
307     ++totalWrites;
308     int h = getDynamicHash(k.hashCode());
309     Block<K, V> b = getBlock(h);
310
311     b.put(k, v);
312
313     return v;
```

```
314
315 }
316
317 public long getTotalWrites() {
318     return totalWrites;
319 }
320
321 private Block<K, V> getBlock(int h) {
322     notifyBlockRequested(h);
323     return blockList.get(h);
324 }
325
326 void blockOverflowed(Block<K, V> b, K k, V v) {
327
328     splitNextBucket();
329
330     b.addOverflowBucket();
331     b.put(k, v);
332 }
333
334 private void splitNextBucket() {
335     Block<K, V> b = getBlock(split);
336     TreeMap<K, V> map = new TreeMap<K, V>();
337     b.drainToMap(map);
338     addBlock();
339     System.err.printf("split N LEVEL %d %d %d \n", split, N, level);
340     if (++split >= (N << level)) {
341         ++level;
342
343         split = 0;
344     }
345
346     for (K k : map.keySet()) {
347         V v = map.get(k);
348         int h = getDynamicHash(k.hashCode());
349         System.err.println(h + " ");
350         Block<K, V> b2 = getBlock(h);
351         b2.put(k, v);
352     }
353 }
354
355 private Block<K, V> addBlock() {
356     Block<K, V> b = new Block<K, V>(this);
357     blockList.add(b);
358
359     return b;
360 }
361
362 @Override
363 public void clear() {
364     blockList = new ArrayList<Block<K, V>>();
365     split = 0;
366     level = 0;
367     totalWrites = 0; // savedBlocks = 0;
368
369 }
370
371 @Override
372 public boolean containsKey(Object key) {
373     return get(key) != null;
374 }
375
376 @Override
```

```
377     public boolean containsValue(Object value) {
378         return values().contains(value);
379     }
380
381     @Override
382     public Set<java.util.Map.Entry<K, V>> entrySet() {
383         TreeSet<Map.Entry<K, V>> set = new TreeSet<Map.Entry<K, V>>();
384         Set<K> kk = keySet();
385         for (K k : kk) {
386             final K k2 = k;
387             set.add(new Entry<K, V>() {
388
389                 @Override
390                 public K getKey() {
391                     return k2;
392                 }
393
394                 @Override
395                 public V getValue() {
396                     return get(k2);
397                 }
398
399                 @Override
400                 public V setValue(V value) {
401                     return put(k2, value);
402                 }
403             });
404         }
405         return set;
406     }
407
408     @Override
409     public V get(Object key) {
410         int h = getDynamicHash(key.hashCode());
411         Block<K, V> b = getBlock(h);
412         return b.replace((K) key, null);
413     }
414
415     @Override
416     public boolean isEmpty() {
417         return size() == 0;
418     }
419
420     @Override
421     public Set<K> keySet() {
422         TreeSet<K> kk = new TreeSet<K>();
423         for (int i = 0; i < blockList.size(); ++i) {
424             Block<K, V> b = getBlock(i);
425             kk.addAll(b.keyList);
426             Block<K, V> ov = b.overflow;
427             while (ov != null) {
428                 kk.addAll(ov.keyList);
429                 ov = ov.overflow;
430             }
431         }
432         return kk;
433     }
434
435     @Override
436     public void putAll(Map<? extends K, ? extends V> m) {
437         for (K k : m.keySet()) {
438             put(k, m.get(k));
439         }
```

```
440     }
441
442     @Override
443     public V remove(Object key) {
444         return null;
445     }
446
447     @Override
448     public int size() {
449         long sz = longSize();
450         return (int) (sz > Integer.MAX_VALUE ? Integer.MAX_VALUE
451             : sz);
452     }
453
454     public long longSize() {
455         long sz = 0;
456         for (Block<K, V> b : blockList) {
457             Block<K, V> b1 = b;
458             while (b1 != null) {
459                 sz += b1.size;
460                 b1 = b1.overflow;
461             }
462         }
463         return sz;
464     }
465
466     @Override
467     public Collection<V> values() {
468         ArrayList<V> list = new ArrayList<V>();
469         Set<K> kk = keySet();
470         for (K k : kk) {
471             list.add(get(k));
472         }
473         return list;
474     }
475
476     private void showStructure() {
477         for (int i = 0; i < blockList.size(); ++i) {
478
479             Block<K, V> b = getBlock(i);
480             Block<K, V> ov = b.overflow;
481             int k = 0;
482             while (ov != null) {
483                 ov = ov.overflow;
484                 ++k;
485             }
486
487             System.out.println("Block " + i + " size " + b.keyList.size()
488                 + " overflow blocks = " + k);
489         }
490     }
491 }
492
493 }
```

Each block is a file in this implementation, because blocks are variably sized here to account for generics and variable sized key and value pairs, and overflow blocks are conceptual, not actual as on disc storage, because the block's contents and that of its overflow bucket(s), are saved as a list of alternating keys and values in this implementation. Methods for saving and loading to

Object streams was used in a standard java customized object persistence way by implementing the Externalizable interface in the Block data class.

```
001 package linearhashmap;
002
003 import java.io.ByteArrayInputStream;
004 import java.io.ByteArrayOutputStream;
005 import java.io.File;
006 import java.io.FileInputStream;
007 import java.io.FileNotFoundException;
008 import java.io.FileOutputStream;
009 import java.io.IOException;
010 import java.io.ObjectInputStream;
011 import java.io.ObjectOutputStream;
012 import java.io.Serializable;
013 import java.util.ArrayList;
014 import java.util.Random;
015 /**
016  * This class manages the LHMap2 class block disk swapping, and saves and load
017  * an instance of the LHMap2 class.
018  * It has been used to externally index a legacy file based database of
019  * 100,000 record master table, and 1,000,000 record
020  * sized child tables, and accounts for heap space available in the java
021  * virtual machine, so that OutOfMemory errors
022  * are avoided when the heap space is low by putting blocks back on files, and
023  * garbage collecting them.
024  * The main performance bottleneck appeared when loading a million record table
025  * for an index , on initial creation
026  * of the index.
027  * @author doctor
028  *
029  * @param <K>
030  * @param <V>
031  */
032
033 public class LHMap2BlockFileManager<K, V> implements
034     LHMap2.BlockListener<K, V>, Serializable {
035
036     /**
037      *
038      */
039
040     private static final long serialVersionUID = 2615265603397988894L;
041     LHMap2BlockFileManagerData data = new LHMap2BlockFileManagerData(
042         new byte[10000], new Random(), 0, new ArrayList<Integer>(), 0);
043
044     public LHMap2BlockFileManager(File baseDir, String name, int maxBlocks,
045         double unloadRatio) {
046         data.home = new File(baseDir, name);
047         if (!data.home.exists())
048             data.home.mkdir();
049         this.data.maxBlocks = maxBlocks;
050         this.data.unloadRatio = unloadRatio;
051     }
052
053     @Override
054     public void blockRequested(int block, LHMap2<K, V> map) {
055         LHMap2.Block<K, V> b = map.blockList.get(block);
056
057         if (b == null) {
058             int tries = 3;
059             File f = new File(data.home, Integer.toString(block));
060             do {
```

```
055         if (f.exists())
056             break;
057
058         if (!f.exists()) {
059             if (--tries >= 0)
060                 fatal(block);
061             try {
062
063                 Thread.sleep(100);
064             } catch (InterruptedException e) {
065                 e.printStackTrace();
066             }
067
068         }
069
070     } while (true);
071     try {
072         ByteArrayInputStream bis = new ByteArrayInputStream(data.buf);
073         FileInputStream fis = new FileInputStream(f);
074         int sz = fis.read(data.buf);
075         fis.close();
076         addByteStats(sz);
077         ObjectInputStream ois = new ObjectInputStream(bis);
078         b = new LHMap2.Block<K, V>();
079
080         b.readExternal(ois);
081         ois.close();
082         b.loadFromShadow(map);
083
084         map.blockList.set(block, b);
085         --data.retired;
086
087     } catch (FileNotFoundException e) {
088         e.printStackTrace();
089         fatal(block);
090     } catch (IOException e) {
091         e.printStackTrace();
092         fatal(block);
093     } catch (ClassNotFoundException e) {
094         e.printStackTrace();
095         fatal(block);
096     }
097
098 }
099 int size = map.blockList.size();
100
101 try {
102     long freeMemory = Runtime.getRuntime().freeMemory();
103
104     long limitMemory = (long) (data.avgBlockSize * data.unloadRatio *
size);
105
106     if (block % 30 == 0)
107         System.err.println("free memory =" + freeMemory + " limit "
+ limitMemory);
108
109
110
111     if (map.split % 20 == 19) {
112         // this is just to add statistics before really needing to
retire
113         retireRandomBlock(map, block);
114         ++data.retired;
115     }
```

```
116
117     } else if (freeMemory < limitMemory) {
118         for (int i = 0; i < size / 4; ++i) {
119             retireRandomBlock(map, block);
120             ++data.retired;
121         }
122         System.runFinalization();
123         System.gc();
124     }
125
126     } catch (FileNotFoundException e) {
127         e.printStackTrace();
128     } catch (IOException e) {
129         e.printStackTrace();
130     }
131
132 }
133
134 private void addByteStats(int sz) {
135     ++data.avgCount;
136     data.avgBlockSize = (int) ((data.avgBlockSize
137         * (data.avgCount - 1) + sz) / data.avgCount);
138 }
139
140 public void retireRandomBlock(LHMap2<K, V> map, int notThisOne)
141     throws FileNotFoundException, IOException {
142     int pick = 0;
143     int size = map.blockList.size();
144     LHMap2.Block<K, V> b = null;
145
146     for (pick = 0; pick < size
147         && (pick == notThisOne || (b = map.blockList.get(pick)) ==
148         null); ++pick)
149     ;
150     if (pick < size)
151         retireOneBlock(map, pick, b);
152 }
153
154 private void retireOneBlock(LHMap2<K, V> map, int pick, LHMap2.Block<K, V>
155 b)
156     throws IOException, FileNotFoundException {
157     if (b == null)
158         return;
159
160     if (b.isChanged()) {
161         // System.err.println("Retiring " + pick);
162         File f = new File(data.home, Integer.toString(pick));
163         ByteArrayOutputStream bos = new ByteArrayOutputStream();
164
165         ObjectOutputStream oos = new ObjectOutputStream(bos);
166         b.writeExternal(oos);
167         oos.flush();
168         oos.close();
169         FileOutputStream fos = new FileOutputStream(f);
170         byte[] bb = bos.toByteArray();
171
172         fos.write(bb);
173         fos.flush();
174         fos.close();
175         if (bb.length > data.buf.length) {
176             data.buf = bb;
```

```
177     }
178 }
179 map.blockList.set(pick, null);
180 b = null;
181 }
182
183 private void fatal(int block) {
184     Exception e = new Exception();
185     try {
186         throw e;
187     } catch (Exception e2) {
188         e2.printStackTrace();
189     }
190     System.err.println("block " + block
191         + " requested and it is not in blocklist and not a file");
192     for (int i : data.retirees) {
193         System.err.print(i + " ");
194     }
195     System.err.println(" were retired");
196     System.exit(-2);
197 }
198
199 public static boolean metaExists(File indexDir, String name) {
200     File home = new File(indexDir, name);
201     return new File(home, "LinearMap2").exists();
202 }
203
204 public static <K, V> LHMap2<K, V> load(File baseDir, String name)
205     throws FileNotFoundException, IOException, ClassNotFoundException {
206     File home = new File(baseDir, name);
207
208     File f2 = new File(home, "LinearMap2");
209     ObjectInputStream ois = new ObjectInputStream(new FileInputStream(f2));
210     LHMap2<K, V> map = (LHMap2<K, V>) ois.readObject();
211     ois.close();
212     loadBlocks(map);
213
214     return map;
215 }
216
217 private static <K, V> void loadBlocks(LHMap2<K, V> map) {
218     LHMap2BlockFileManager<K, V> mgr = getBlockManagerListener(map);
219     int size = map.blockList.size();
220     for (int i = 0; i < size; ++i) {
221         mgr.blockRequested(i, map);
222     }
223 }
224
225 public static <K, V> LHMap2BlockFileManager<K, V> getBlockManagerListener(
226     LHMap2<K, V> map) {
227     LHMap2BlockFileManager<K, V> mgr = (LHMap2BlockFileManager<K, V>)
228     map.listeners
229         .get(0);
230     return mgr;
231 }
232
233 public static void save(File indexDir, String name,
234     LHMap2<?, ?> offsetMap) throws FileNotFoundException, IOException {
235     retireAllBlocks(offsetMap);
236
237     File home = new File(indexDir, name);
238     File f2 = new File(home, "LinearMap2");
239     ObjectOutputStream oos = new ObjectOutputStream(
```

```
239         new FileOutputStream(f2));
240         oos.writeObject(offsetMap);
241         oos.close();
242     }
243
244     private static <K, V> void retireAllBlocks(LHMap2<K, V> offsetMap)
245         throws FileNotFoundException, IOException {
246         LHMap2BlockFileManager<K, V> mgr = getBlockManagerListener(offsetMap);
247         int sz = offsetMap.blockList.size();
248         for (int i = 0; i < sz; ++i) {
249             LHMap2.Block<K, V> b = offsetMap.blockList.get(i);
250             // offsetMap.blockList.set(i, null); // mark for reloading as block
251             // destroyed after writing
252             if (b != null) {
253                 mgr.retireOneBlock(offsetMap, i, b);
254             }
255         }
256     }
257 }
258 }
```

```
01 package linearhashmap;
02
03 import java.io.File;
04 import java.io.Serializable;
05 import java.util.List;
06 import java.util.Random;
07
08 public class LHMap2BlockFileManagerData implements Serializable{
09     /**
10      *
11      */
12     private static final long serialVersionUID = 1L;
13     public byte[] buf;
14     public Random r;
15     public File baseDir;
16     public File home;
17     public int maxBlocks;
18     public int retired;
19     public double unloadRatio;
20     public List<Integer> retirees;
21     public int avgBlockSize;
22     public long avgCount;
23
24     public LHMap2BlockFileManagerData(byte[] buf, Random r, int retired,
25         List<Integer> retirees, long avgCount) {
26         this.buf = buf;
27         this.r = r;
28         this.retired = retired;
29         this.retirees = retirees;
30         this.avgCount = avgCount;
31     }
32
33
34 }
```

Footnotes

1. ↑ [Wikipedia: Comparison of programming languages \(mapping\)](#) shows how many programming languages provide hash table functionality.

7.8: Implementations is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by LibreTexts.

CHAPTER OVERVIEW

8: Searching, Binary Search Trees and Heaps

Unit Introduction

Searching is one of the most fundamental problems in Computer Science. It is the process of finding a particular item in a collection of items. Typically, a search answers whether the item is present in the collection or not. This unit discusses two important search methods: sequential search and binary search. A Sequential (linear) search is the simplest method to solve the searching problem. It finds an item in a collection by looking for it in a collection one at a time from the beginning till the end. Binary Search on the other hand is use a divide and conquer strategy. In divide and conquer, a larger problem is reduced into smaller sub-problems recursively until the problem is small enough that the solution is trivial. A condition for binary search is that the array should be a sorted array.

The unit also discusses two important data structures; a binary search tree (BST) and a heap. A BST is a binary tree based data structure that is viewed to support efficiently the dynamic set operations, including search and insert operations amongst others. A heap on the other hand is a specific tree based data structure in which all the nodes of tree are in a specific order. The maximum number of children of a node in the heap depends on the type of heap. However, a binary heap is the most commonly used heap type, where there are at most 2 children of a node.

Unit Objectives

Upon completion of this unit you should be able to:

- Demonstrate knowledge of searching algorithms
- Implement a searching algorithm based on efficiency requirements
- Demonstrate knowledge of advanced data structures including binary search trees and binary heap
- Implement a data structure based on efficiency requirements

Key Terms

Searching: A process of finding a particular item in a collection of items

Linear search: A search that sequentially looks for an item in a collection from beginning to the end

Binary Search: A search based on divide and conquer strategy

Binary tree: A tree made of nodes, where each node contains a “left” pointer, a “right” pointer, and a data element.

Binary search tree (BST): A type of binary tree where the nodes are arranged in order, that is, for each node all keys in its left sub-tree are less-or-equal to its key and all the keys in its right sub-tree are greater than its key.

In-order Traversal (Walk): A BST walk that recursively prints the BST keys in monotonically increasing order

Red-black trees: A variant of BST that ensure that the tree is balanced, that is, its height is $O(\lg n)$, where n is the number of nodes.

Heap: A specific tree based data structure in which all the nodes of tree are in a specific order.

Learning Activities

[8.1: Activity 1 - Searching Techniques](#)

[8.2: Activity 2 - Binary Search Tree](#)

[8.3: Activity 3 - Heaps](#)

[8.4: Unit Summary](#)

This page titled [8: Searching, Binary Search Trees and Heaps](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Godfry Justo](#) (African Virtual University) .

8.1: Activity 1 - Searching Techniques

Introduction

Searching is one of the most fundamental problems in Computer Science. It is the process of finding a particular item in a collection of items. Typically, a search answers whether the item is present in the collection or not. For simplification for all our examples, we will be taking an array as the collection. This activity will present two searching methods: the sequential search and the binary search methods.

Activity Details

Sequential Search

Sequential (linear) search is the simplest method to solve the searching problem. It finds an item in a collection by looking one item at a time from the beginning to end of collection if it does not find it. Basically, it checks each item in the sequence until the desired element is found before all the elements of the collection are exhausted.

The code snapshot below presents a linear search method.

```
bool linearSearch (int A[], int length, int item) {
    for (int i = 0 ; i < length ; ++i)
        if (item == A[i])
            return true;          // Item is found in array A
    return false;                // Item is not found in array A }
```

Consider an array $A = \{1, 9, 2, 4, 6, 3, 7, 5, 8, 0\}$ and suppose that we wish to find 3 from the given array. The linear search starts from the beginning and checks if current item matches with the item we are searching. In the provided example the item = 3 and the array length = 10.

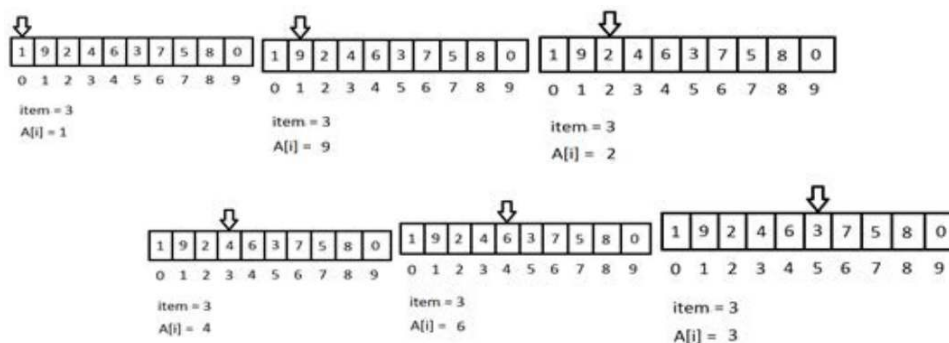


Figure 2.1.1 : Trace of Linear Search for item = 3

Figure 2.1.1 shows the trace of the sequential search algorithm to find item 3. Note that for $i = 0$, $A[i] = 1$ and the item being sought is not equal to $A[i]$. Similarly, for $i = 1$, $A[i] = 9$ and item is not equal to $A[i]$, for $i = 2$, $A[i] = 2$ and item is not equal to $A[i]$, for $i = 3$, $A[i] = 4$ and item is not equal to $A[i]$ and for $i = 4$, $A[i] = 6$ and item is not equal to $A[i]$. But for $i = 5$, $A[i] = 3$ and item is equal to $A[i]$. Therefore, the `linearSearch()` function will return true showing that the item is found. Note that, if the item = 10, `linearSearch()` function will iterate over all the elements in the array A , and return false (showing that the item is not found), because the 'if' condition will never be true.

Now, let us consider the modified search where we need to return the position, where the item is found. To solve this modified search problem, we just need to change two lines in the `linearSearch()` function as shown below:

```
int linearSearch ( int A[ ], int length, int item) {
    for (int i = 0; i < length ; ++i)
        if (item == A[i])
```

```
        return i;        //Returning the index of the element found.
    return -1;           // Item is not found in array A
}
```

In the above codes -1 denotes that the item is not found. Note that if there are duplicate elements, the above code will return the first occurrence of the item. The time complexity of linear search is **O(size of the array)**.

Binary Search

Binary Search is a divide and conquer algorithm. In divide and conquer, a larger problem is reduced into smaller sub-problems recursively until the problem is small enough that the solution is trivial. Unlike sequential search, binary search requires that the input array be a sorted array. If the array is sorted in ascending order, binary search compares the item with the middle element of the array. If the item matches, then it returns true. Otherwise, if the item is less than the middle element then it will recursively search on the sub-array to the left of the middle element or if the item is greater it will recursively search on the sub-array to the right of the middle element. Therefore, at each step the size of array will be reduced to half. After repeating this recursion, it will eventually be left with a single element.

The pseudocode for the binary search algorithm is presented below:

```
Algorithm binarySearch(A, left, right, item) {
    if left is less than or equal to right then :
        mid = (left + right) / 2
        if A[mid] is equal to item then return true
        else if item is less than A[mid] then recursively search on the left sub-array
        else if item is greater than A[mid] recursively on the right sub-array
    else
        return false }
```

Remember that an array should be sorted for the binary search to work!! A recursive implementation of binary search is presented below:

```
//Recursive Binary Search
bool binarySearchRecur(int A[], int left, int right, int item)
{
    if (left <= right)
    {
        int mid = left + (right - left) / 2; // finding middle index
        if (A[mid] == item)
            return true;                    // item found
        else if (item < A[mid])
        {
            // recursively search on the left sub-array
            return binarySearchRecur(A, left, mid-1, item);
        }
        else
        {
            // recursively search on the right sub-array
            return binarySearchRecur(A, mid+1, right, item);
        }
    }
}
```

```
else
    return false; // item not found )
```

Similarly, an iterative implementation of binary search is given below:

```
//Iterative Binary Search
bool binarySearchIter(int A[], int length, int item)
{ int left = 0, right = length - 1, mid;
  while (left <= right)
  { mid = left + (right - left) / 2; // finding middle index
    if (A[mid] == item)
        return true; // item found
    else if (item < A[mid])
        right = mid - 1; // search on left sub-array
    else
        left = mid + 1; // search on right sub-array
  }
  return false; // item not found }
```

Suppose now we are given array $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and we need to find 3 in the array A. Therefore, we have that search item = 3 and array length = 10. Table 2.1.1 traces the execution of the iterative binary search for item =3 on array A. Note that initially left = 0 and right = length – 1 = 10 – 1 = 9. Figure 2.1.2 depicts that the array diminishes during iterations.

Table 2.1.1 : A trace for Iterative Binary Search on array A

Iteration	left	Right	mid	Remarks
1	0	9	$\text{mid} = \text{left} + (\text{right} - \text{left}) / 2 = 0 + (9 - 0) / 2 = 4$	$A[\text{mid}] = 4$ is greater than item. Since the array A is sorted we conclude that the item being found is in the left sub-array
2	0	$\text{mid} - 1 = 4 - 1 = 3$	$\text{left} + (\text{right} - \text{left}) / 2 = 0 + (3 - 0) / 2 = 1$	$\text{mid} = A[\text{mid}] = 1$ is smaller than item. Since the array A is sorted the item must be in the right sub-array.
3	$\text{mid} + 1 = 1 + 1 = 2$	right = 3	$\text{left} + (\text{right} - \text{left}) / 2 = 2 + (3 - 2) / 2 = 2$	$\text{mid} = A[\text{mid}] = 2$ is smaller than item. Since the array A is sorted we can say that the item must be in the right sub-array
4	$\text{mid} + 1 = 2 + 1 = 3$	right = 3	$\text{mid} = \text{left} + (\text{right} - \text{left}) / 2 = 3 + (3 - 3) / 2 = 3$	$A[\text{mid}] = 3$ is equal to the item, so the binarySearchIter() function will return true and the item is found

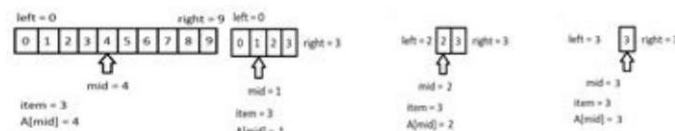


Figure 2.1.2 : Array diminishes during iterations

Time complexity of both binarySearchRecur() and binarySearchIter() is $O(\log N)$ where N is the size of the array. This is because after each recursion or iteration the size of problem is reduced into half.

Conclusion

In this activity we presented two search algorithms, sequential search and binary search. The sequential search method is based on exhaustive search strategy while the binary search method is based on divide and conquers search strategy. Binary search also requires that the input array to be sorted.

? Assessment

1. Modify the `linearSearch()` function discussed in this activity such that it will return the last occurrence of the item, if the item is in the array, otherwise, return -1. (Hint: Store the position of the last found occurrence of the item and update it as soon as the item occurs again).
2. Suppose you are given a sorted array with possible duplicate elements within it. Write the code segment that finds the first occurrence of a given input item (assuming the given item exists in the given array).

This page titled [8.1: Activity 1 - Searching Techniques](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Godfry Justo \(African Virtual University\)](#).

- [2.1: Activity 1 - Searching Techniques](#) by [Godfry Justo](#) is licensed [CC BY-SA 3.0](#).

8.2: Activity 2 - Binary Search Tree

Introduction

A binary tree is made of nodes, where each node contains a left pointer, a right pointer, and a data element also called a key. The root pointer points to the topmost node in the tree. The left and right pointers recursively points to left and right sub-tree respectively. A null pointer represents a binary tree with no elements, that is, an empty tree. Figure 2.2.1 depicts a binary tree.

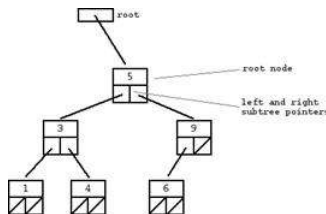


Figure 2.2.1 : A binary tree

A **binary search tree** (BST) also called an ordered binary tree is a type of binary tree where the nodes are arranged in order. That is, for each node, all elements in its left sub-tree are less-or-equal to its element, and all the elements in its right sub-tree are greater than its element. The tree shown in Figure 2.2.1 is a binary search tree since the root node element is a 5, and its left sub-tree elements (i.e., 1, 3, 4) are less than 5, and its right sub-tree elements (i.e., 6, 9) are greater than 5. Recursively, each of the sub-trees must also obey the binary search tree constraint, that is, in the (1, 3, 4) sub-tree, the node with element 3 is the root, and that $1 \leq 3$ and $4 > 3$. In this regard, note that a binary search tree is different from a binary tree. The nodes at the bottom edge of the tree which have empty sub-trees and are called leaf nodes (i.e., 1, 4, 6), while the others are called internal nodes (i.e., 3, 5, 9).

Activity Details

A binary tree is recursively defined as follows:

- i. An empty tree is a binary tree
- ii. A node with two child sub-trees is a binary tree
- iii. Only what you get from (i) by a finite number of applications of (ii) is a binary tree.

Now check whether Figure 2.2.2 is a binary tree.

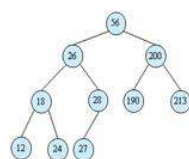


Figure 2.2.2 : A tree

A binary search tree (BST) is a fundamental data structure that can support dynamic set operations including: Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete. The efficiency of its operations (especially, the search and insert operations) make it be suited to use of building dictionaries and priority queues. Its basic operations take time proportional to the height, h , of the tree: i.e., $O(h)$.

Like a binary tree, binary search tree is represented by a linked data structure of nodes. The root (T) points to the root of tree T. Each node contains three fields: key, left (pointer to left child/root of left sub-tree) and right (pointer to right child/root of right sub-tree).

A binary tree is a binary search tree if the stored keys satisfy the following two binary search tree properties:

- $\forall y$ in left sub-tree of x , then $\text{key}[y] \leq \text{key}[x]$.
- $\forall y$ in right sub-tree of x , then $\text{key}[y] \geq \text{key}[x]$.

Do the keys of the binary tree in Figure 2.2.3 satisfy binary search tree properties?

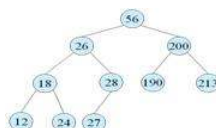


Figure 2.2.3 : A binary tree

A binary search tree property allows its keys to be recursively visited/printed in monotonically increasing order, called an in-order traversal (walk). Figure 2.2.4 presents a pseudo-code for the in-order traversal. The correctness of the in-order walk can be elaborated through induction approach on the size of tree as follows: For an empty tree (i.e., size=0), this is easy. Suppose size > 1, then

- Prints left sub-tree in order by induction
- Prints root, which comes after all elements in left sub-tree (still in order)
- Prints right sub-tree in order (all elements come after root, so still in order)

```

Inorder-Tree-Walk(x)
1. if x ≠ NIL
2.   then Inorder-Tree-Walk(left[p])
3.   print key[x]
4.   Inorder-Tree-Walk(right[p])
  
```

Figure 2.2.4 : In-order traversal

Recall that all dynamic set search operations on a binary search tree can be supported in $O(h)$ time, where h is the tree height. Note that $h = \Theta(\lg n)$ for a balanced binary tree (and for an average tree built by adding nodes in random order) and that $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of n nodes in the worst case.

In the following we present the algorithms for some of the dynamic set operations. In Figure 2.2.5 and 2.2.6 presents the algorithms for the tree search operation. Figure 2.2.5 presents the algorithm for the search operation using both the recursive and iterative method respectively. The running time for both algorithms is $O(h)$. However, the iterative tree search method is more efficient on most computers, while the recursive tree search is elegantly straightforward.

Tree-Search(x, k)	Iterative-Tree-Search(x, k)
1. if $x = \text{NIL}$ or $k = \text{key}[x]$	1. while $x \neq \text{NIL}$ and $k \neq \text{key}[x]$
2. then return x	2. do if $k < \text{key}[x]$
3. if $k < \text{key}[x]$	3. then $x \leftarrow \text{left}[x]$
4. then return Tree-Search(left[x], k)	4. else $x \leftarrow \text{right}[x]$
5. else return Tree-Search(right[x], k)	5. return x

Figure 2.2.5 : A recursive and iterative search algorithm

The other important dynamic set operation is finding Minimum and Maximum key respectively. The binary search tree properties guarantees that the minimum key is located at the left-most node and the maximum key is located at the right-most node. Figure 2.2.6 presents the algorithm for finding minimum and maximum keys.

Tree-Minimum(x)	Tree-Maximum(x)
1. while $\text{left}[x] \neq \text{NIL}$	1. while $\text{right}[x] \neq \text{NIL}$
2. do $x \leftarrow \text{left}[x]$	2. do $x \leftarrow \text{right}[x]$
3. return x	3. return x

Figure 2.2.6 : Finding Min and Max

The other dynamic set operations on a binary search tree are finding the Predecessor and Successor keys. The successor of node x is the node y such that $\text{key}[y]$ is the smallest key greater than $\text{key}[x]$. The successor of the largest key is NIL. The search for successor key consists of two cases:

- If node x has a non-empty right sub-tree, then x 's successor is the minimum in the right sub-tree of x .
- If node x has an empty right sub-tree, then
 - As long as we move to the left up the tree (move up through right children), we are visiting smaller keys.
 - x 's successor y is the node that x is the predecessor of (x is the maximum in y 's left sub-tree).

That is, x 's successor y , is the lowest ancestor of x whose left child is also an ancestor of x .

Figure 2.2.7 presents the pseudo-code for the successor operation. The code for predecessor operation is symmetric. The running time in both cases is $O(h)$.

```

Tree-Successor(x)
1. if right[x]  $\neq$  NIL
2.   then return Tree-Minimum(right[x])
3.  $y \leftarrow p[x]$ 
4. while  $y \neq$  NIL and  $x =$  right[y]
5.   do  $x \leftarrow y$ 
6.    $y \leftarrow p[y]$ 
7. return y
  
```

Figure 2.2.7 : Finding successor

Insertion is also an important dynamic set operation on binary search tree. The insertion operation changes the dynamic set represented by a BST. The operation should ensure the binary search tree properties holds after a change. Generally, insertion is easier than deletion. Figure 2.2.8 presents the pseudo-code for insertion operation.

```

Tree-Insert(T, z)
1.  $y \leftarrow$  NIL
2.  $x \leftarrow$  root[T]
3. while  $x \neq$  NIL
4.   do  $y \leftarrow x$ 
5.     if key[z] < key[x]
6.       then  $x \leftarrow$  left[x]
7.       else  $x \leftarrow$  right[x]
8.  $p[z] \leftarrow y$ 
9. if  $y =$  NIL
10.  then root[t]  $\leftarrow$  z
11.  else if key[z] < key[y]
12.    then left[y]  $\leftarrow$  z
13.    else right[y]  $\leftarrow$  z
  
```

Figure 2.2.8 : BST Insertion operation

The analysis of insertion operation is as follows:

- Initialization takes $O(1)$
- The while loop in lines 3-7 searches for place to insert z , maintaining parent y . This takes $O(h)$ time.
- Lines 8-13 insert the value in $O(1)$.

Thus, overall it takes $O(h)$ time to insert a node.

The delete operation in a BST is somehow involving. The BST deletion operation Tree-Delete (T, x) considers three cases:

- case 0 : if x has no children then remove x
- case 1 : if x has one child then make $p[x]$ point to child
- case 2 : if x has two children (sub-trees), then swap x with its successor and perform case 0 or case 1 to delete it.

Overall delete operation takes $O(h)$ time to delete a node. The pseudo-code for deleting a key in a BST is presented below:

```

Tree-Delete(T, z)
/* Determine which node to splice out: either z or z's successor. */
if left[z] = NIL or right[z] = NIL
then  $y \leftarrow z$ 
else  $y \leftarrow$  Tree-Successor[z]
/* Set x to a non-NIL child of x, or to NIL if y has no children. */
if left[y]  $\neq$  NIL
then  $x \leftarrow$  left[y]
else  $x \leftarrow$  right[y]
/* y is removed from the tree by manipulating pointers of p[y] and x */
if  $x \neq$  NIL
then  $p[x] \leftarrow p[y]$ 
if  $p[y] =$  NIL
then root[T]  $\leftarrow$  x
  
```

```
else if y ← left[p[i]]
then left[p[y]] ← x
else right[p[y]] ← x
/* If z's successor was spliced out, copy its data into z */
if y ≠ z
then key[z] ← key[y]
copy y's satellite data into z.
return y
```

How do we know that case 2 should go to case 0 or case 1 instead of back to case 2? Due to the fact that when x has 2 children, its successor is the minimum in its right sub-tree, and that successor has no left child (hence 0 or 1 child). Equivalently, we could swap with predecessor instead of successor. It might be good to alternate to avoid creating unbalanced tree.

Conclusion

In this activity binary search trees were presented. Binary search trees is among advanced data structures of practical interest due to the efficiency of its basic operations including search and insert that make it attractive for implementing dictionaries and priority queues.

? Assessment

1. Explain each of the following terms:
 - a. Binary tree
 - b. Binary search tree
 - c. In-order tree traversal
2. Draw the BST that results when you insert items with keys
AFRICAN VIRTUAL UNIVERSITY
in that order into an initially empty tree.
3. Suppose we have integer values between 1 and 1000 in a BST and search for 363. Which of the following cannot be the sequence of keys examined (Hint: Draw the BST as if you were tracing the keys in the order given).
 - a. 2 252 401 398 330 363
 - b. 399 387 219 266 382 381 278 363
 - c. 3 923 220 911 244 898 258 362 363
 - d. 4 924 278 347 621 299 392 358 363
 - e. 5 925 202 910 245 363

This page titled [8.2: Activity 2 - Binary Search Tree](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Godfry Justo](#) (African Virtual University).

- [2.2: Activity 2 - Binary Search Tree](#) by [Godfry Justo](#) is licensed [CC BY-SA 3.0](#).

8.3: Activity 3 - Heaps

Introduction

A heap is a specific tree based data structure in which all the nodes of tree are in a specific order. Suppose that X is a parent node of Y, then the value of X follows some specific order with respect to value of Y and the same order will be followed across the tree. The maximum number of children of a node in the heap depends on the type of heap. However, a binary heap is the more commonly used heap type, in which there are at most 2 children of a node.

Activity Details

In binary heap, if the heap is a complete binary tree with N nodes, then it has smallest possible height which is $\log_2 N$. For example, from Figure 2.3.1, observe a particular sequence that each node has greater value than any of its children. Suppose there are N Jobs in a queue to be done, and each job has its own priority. The job with maximum priority will get completed first than others. At each instant we are completing a job with maximum priority and at the same time we are also interested in inserting a new job in the queue with its own priority.



Figure 2.3.1 : A binary heap

So at each instant we have to check for the job with maximum priority to complete it and also insert if there is a new job. This task can be very easily executed using a heap by considering N jobs as N nodes of the tree.

We can use an array to store the nodes of the heap tree. Consider a heap depicted in Figure 2.3.2 and suppose that there are 7 elements with values {6, 4, 5, 3, 2, 0, 1}. We can use an array to simulate a tree in the following way: If we are storing one element at index, i, in array Arr, then its parent will be stored at index $i/2$ (unless it's a root, since a root has no parent) and can be accessed by $Arr[i/2]$, and its left child can be accessed by $Arr[2 * i]$ and its right child can be accessed by $Arr[2 * i + 1]$. The index of root in an array is 1.

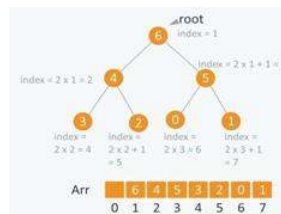


Figure 2.3.2 : Heap and Corresponding Array representation

There are two basic types of heap, max heap and min heap. In a max heap the value of parent node is always greater than or equal to the value of child node across the tree and the node with highest value is the root node of the tree. In a min heap the value of parent node is always less than or equal to the value of child node across the tree. Therefore the node with lowest value is the root node of tree.

Max heap implementation

Suppose that you are given heap elements which are stored in array Arr. We can convert the array Arr into a heap structure as follows: Pick a node in the array, check if the left sub-tree and the right sub-tree are max heaps, among themselves and the node itself is a max heap (i.e., its value should be greater than all the child nodes). To perform this operation we define a function `max_heapify` that maintain the property of max heap (i.e. each element value should be greater than or equal to any of its child and smaller than or equal to its parent) as presented below:

```

void max_heapify (int Arr[ ], int i, int N)
{
    int left  = 2*i           //left child
    int right = 2*i + 1       //right child
  
```

```

if(left <= N and Arr[left] > Arr[i] )
    largest = left;
else
    largest = i;
if(right <= N and Arr[right] > Arr[largest] )
    largest = right;
if(largest != i )
{ swap (Arr[i] , Arr[largest]);
  max_heapify (Arr, largest,N); }

```

The function max_heapify has time Complexity of $O(\log N)$.

Now consider the example given in Figure 2.3.3 depicts the max_heapify tree transformation steps. Initially 1st node (root node) is violating property of max-heap as it has smaller value than its children, so we are performing max_heapify function on this node with value 4. Since 8 is greater than 4, then 8 is swapped with 4 and max_heapify is performed again on 4, but on different position. Now in step 2, 6 is greater than 4, so 4 is swapped with 6 and we obtain a max heap, as now 4 is a leaf node, so further call to max_heapify will not create any effect on the heap. Therefore we can correct a violated max heap property by using max_heapify function.



Figure 2.3.3 : Transformation steps to max heap

Now consider the heap property which states:

An N element heap stored in an array has leaves indexed by $N/2+1$, $N/2+2$, $N/2+3$ up to N.

Figure 2.3.4 demonstrates this heap property which is the heap derived from Figure 2.3.3 whose elements have values {8, 7, 6, 3, 2, 4, 5}. Observe that the elements 3, 2, 4, 5 are indexed by $N/2 + 1$ (i.e 4), $N/2+2$ (i.e 5) and $N/2+3$ (i.e 6) and $N/2+4$ (i.e 7) respectively.

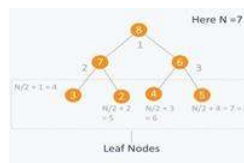


Figure 2.3.4 : Array indices for heap leaf elements

We can build a max heap out of the array elements. That is, suppose that we have N elements stored in the array Arr indexed from 1 to N, and possibly not following the property of a max heap. We use the max-heapify function to make a max heap out of the array. Based on the property stated above, we have that the elements from Arr[$N/2+1$] to Arr[N] as leaf nodes, and each node is a 1 element heap. The max_heapify function can be used in a bottom up manner on remaining nodes, to cover each node of the heap tree. The code presented below captures these concepts to build up a max heap from a given array:

```

void build_maxheap (int Arr[ ])
{ for(int i = N/2 ; i >= 1 ; i-- )
    { max_heapify (Arr, i) ; }
}

```

Note that the time complexity for the build_maxheap function is $O(N)$, since max_heapify function has complexity $\log N$ and the build_maxheap functions runs for only $N/2$ times.



Figure 2.3.5 : An array Arr

Now we demonstrate the max_heap function using a 7 elements array Arr, shown in Figure 2.3.5 . Here $N = 7$, so starting from node having index $N/2 = 3$, (i.e., with value 3 in Figure 2.3.5), call max_heapify from index $N/2$ to 1. Figure 2.3.6 depicts the build steps in which:

During step 1, in max_heapify(Arr, 3), since 10 is greater than 3, 3 and 10 are swapped and further call to max_heap(Arr, 7) will have no effect as 3 is a leaf node.

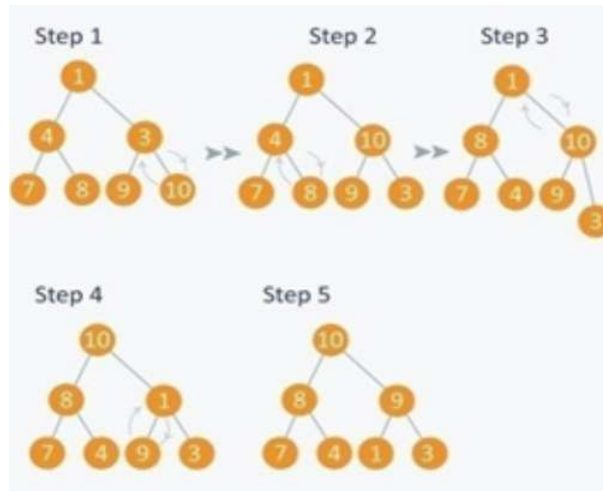


Figure 2.3.6 : Tracing Max heap creation

During step 2, calling max_heapify(Arr, 2), sees node indexed with 2 has value 4, thus 4 is swapped with 8 and further call to max_heap(Arr, 5) will have no effect, since 4 is a leaf node.

During step 3, calling max_heapify(Arr, 1), sees node indexed with 1 has value 1, hence 1 is swapped with 10.

Step 4 is a subpart of step 3, as after swapping 1 with 10, again a recursive call of max_heapify(Arr, 3) will be performed, and 1 will be swapped with 9. Now further call to max_heapify(Arr, 7) will have no effect, as 1 is a leaf node.

In step 5, we finally get a max-heap and the elements in the array Arr will be as shown in Figure 2.3.7 .



Figure 2.3.7 : Resulting Max-heap array representation

The second heap type, min heap, has the value of parent node always less than or equal to the value of child node across the tree. Therefore the node with lowest value is the root node as depicted in Figure 2.3.8 . This figure also captures the fact that a node has a smaller value than the values of its children.



Figure 2.3.8 : A min-heap

In a mini heap we can perform similar operations as in the max heap. Below we present a function min_heapify which maintains the min heap property:

```
void min_heapify (int Arr[ ], int i, int N)
{ int left = 2*i;
  int right = 2*i+1;
  int smallest;
  if(left <= N and Arr[left] < Arr[ i ] )
    smallest = left;
  else
    smallest = i;
```

```

if(right <= N and Arr[right] < Arr[smallest] )
    smallest = right;
if(smallest != i)
{ swap (Arr[ i ], Arr[ smallest ]);
  min_heapify (Arr, smallest,N); }

```

Note that as in the max_heapify function, the time complexity of the min_heapify function is $O(\log N)$.

We demonstrate the above function by transforming the array $\text{Arr} = \{4, 5, 1, 6, 7, 3, 2\}$. Note from Figure 2.3.9, that the element at index 1 is violating the property of min-heap, so performing $\text{min\_heapify}(\text{Arr}, 1)$ preserves the min-heap property.



Figure 2.3.9 : Transforming to a min-heap

Apply the min_heapify function on remaining nodes other than leaves (since a leaf node is a 1 element heap), as defined by function build_minheap below to obtain to a min heap.

```

void build_minheap (int Arr[ ])
{   for( int i = N/2 ; i >= 1 ; i--)
    min_heapify (Arr, i); }

```

Note that as for the build_maxheap function the time complexity for the build_minheap function is $O(N)$.

We demonstrate the build_minheap function by transforming the elements in array $\{10, 8, 9, 7, 6, 5, 4\}$. To do so, need to run min_heapify on nodes indexed from $N/2$ to 1, where the node indexed at $N/2$ has value 9. The build_minheap repeats this through a number of steps to obtain a min_heap as depicted in Figure 2.3.10.

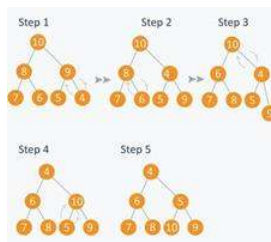


Figure 2.3.10 : Building min-heap from array

Conclusion

In this activity we presented heap data structure. We discussed two types of heap, max heap and min heap, along with the key operation of building a specified heap from an array. Heaps are considered as partially ordered tree, since the nodes of tree do not follow any order with their siblings (nodes on the same level). They are mainly used when setting priority to smallest or the largest node in the tree as we can extract these node very efficiently using heaps.

? Assessment

1. A learner is required to undertake background reading, which is supported by lectures to explain various notions and to show the application of various techniques using examples. The coursework requires the students to solve exercises each week. Feedback for and help with this work is provided in the examples classes.

This page titled [8.3: Activity 3 - Heaps](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Godfry Justo](#) (African Virtual University).

- [2.3: Activity 3 - Heaps](#) by [Godfry Justo](#) is licensed [CC BY-SA 3.0](#).

8.4: Unit Summary

In this unit we presented the searching algorithm and two advanced data structures, the binary search tree and the heap. Two searching algorithms were presented, the sequential search and binary search. The binary search algorithm is generally more efficient than sequential search, although it requires that the input collection be sorted.

The binary search tree operations were presented. The binary search tree data structure is of practical interest due to the efficiency of its basic operations including search and insert that make it attractive for implementing dictionaries and priority queues.

The heap data structure was categorised into two main types, max heap and min heap. For each heap type the operations for building a heap from an array were presented. Heaps are generally useful in priority setting applications through exploiting smallest or largest node in the tree as we can extract these node very efficiently using heaps.

? Unit Assessment

Check your understanding!

General Programming Exercises

1. Suppose you are given a sorted array with possible duplicate elements within it. Write a program that finds the last occurrence of a given input item (assuming the given item exists in the given array).
2. Write a program that reads strings (words) as command line arguments and inserts them into a binary search tree and outputs them in in-order fashion. (You should be able to use your program with files by running your program with the standard input redirected from a file)
3. Create a binary heap with a limited heap size. That is, the heap should keep track of a fixed number of most important items (say n). If the heap grows in size to more than n items the least important item should be dropped.

Answers

<mailto:njulumi@gmail.com>

Grading Scheme

As per the offering Institution grading policy.

Unit Readings and Other Resources

The readings in this unit are to be found at course level readings and other resources.

This page titled [8.4: Unit Summary](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Godfry Justo](#) (African Virtual University) .

- [2.4: Unit Summary](#) by [Godfry Justo](#) is licensed [CC BY-SA 3.0](#).

CHAPTER OVERVIEW

9: Sorting Algorithms

Unit Introduction

A sorting algorithm is an algorithm that puts elements of a collection such as array or list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms, such as in search and merge algorithms that require sorted collections to work correctly. It is also often useful for normalization of data and for producing human-readable output. More precisely, the output must satisfy two conditions (i) the output is in non-decreasing order (each element is no smaller than the previous element according to the desired total order) and (ii) the output is a permutation, or reordering, of the input.

The Sorting algorithms used in computer science are often classified by:

1. Computational complexity (worst, average and best case behavior) in terms of the size of the list (n) - For typical sorting algorithms good behavior is $O(n \log n)$ and bad behavior is $O(n^2)$. Ideal behavior for a sort is $O(n)$. Sort algorithms which only use an abstract key comparison operation often require at least $O(n \log n)$ comparisons on average.
2. Memory usage (and use of other computer resources) - In particular, some sorting algorithms are “in place”, such that little memory is needed beyond the items being sorted, while others need to create auxiliary locations for data to be temporarily stored.
3. Stability - Stable sorting algorithms maintain the relative order of records with equal keys (i.e. values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.
4. Whether or not is a comparison sort - A comparison sort examines the data only by comparing two elements with a comparison operator.
5. General sort method – whether uses insertion, exchange, selection, merging, etc.

Table 9.1 provides a summary of some sorting algorithms and their classification. When equal elements are indistinguishable, such as with integers, stability is not an issue. However, assume that the following pairs of numbers are to be sorted by their first coordinate: (4, 1) (3, 1) (3, 7) (5, 6). In this case, two different results are possible, that is, one which maintains the relative order of records with equal keys, and one which does not as shown below:

(3, 1) (3, 7) (4, 1) (5, 6) (order maintained)

(3, 7) (3, 1) (4, 1) (5, 6) (order changed)

Unstable sorting algorithms may change the relative order of records with equal keys, but stable sorting algorithms never do so. Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original data order as a tie-breaker. Remembering this order, however, often involves an additional space cost.

Table 9.1, presents a classification summary of the sorting algorithms considered in this unit. In the table, n is the number of records to be sorted. The columns “Best”, “Average”, and “Worst” give the time complexity in each case. “Memory” denotes the amount of auxiliary storage needed beyond that used by the collection list itself. “Cmp” indicates whether the sort is a comparison sort.

Table 9.1: A Sorting Algorithms Classification

Name	Best	Average	Worst	Memory	Stable	Cmp	Method
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Yes	Selection
Insertion sort	$O(n)$	—	$O(n^2)$	$O(1)$	Yes	Yes	Insertion
Binary tree sort	$O(n \log(n))$	—	$O(n \log(n))$	$O(1)$	Yes	Yes	Insertion
Merge sort	$O(n \log(n))$	—	$O(n \log(n))$	$O(n)$	Yes	Yes	Merging
Heap sort	$O(n \log(n))$	—	$O(n \log(n))$	$O(1)$	No	Yes	Selection
Quick sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(\log n)$	No	Yes	Partitioning

Unit Objectives

Upon completion of this unit you should be able to:

- Demonstrate understanding of various sorting algorithm
- Select best sorting method based on performance requirement
- Apply sorting algorithms in problem solving

Key Terms

Sorting: A process that puts elements of a collection such as array or list in a certain order

Learning Activities

[9.1: Activity 1 - Sorting Algorithms by insertion method](#)

[9.2: Activity 2 - Sorting Algorithms by selection method](#)

[9.3: Activity 3 - Sorting Algorithms by Merge and Partition methods](#)

[9.4: Unit Summary](#)

This page titled [9: Sorting Algorithms](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Godfry Justo](#) (African Virtual University).

9.1: Activity 1 - Sorting Algorithms by insertion method

This activity will present two insertion based sorting algorithms, the insertion sort and the binary tree sort algorithms.

Activity Details

Insertion sort Algorithm

The insertion sort is good for sorting moderately sized arrays (up to some tens of elements). For larger arrays, we will see later a much faster algorithm. The basic idea of insertion sort is as follows: Assume that at the start of iteration i of the algorithm, the first i elements of the array are already sorted. The array therefore consists of a left part which is sorted and a right part which is unsorted, as shown in Figure 3.1.1 .

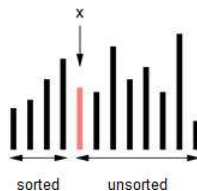


Figure 3.1.1 : Insertion sort progress

Next the algorithm takes the $(i+1)$ -th element x and inserts it into the appropriate position of the sorted part to the left of x by shifting the elements larger than x one position to the right. Thus, we have that the first $i+1$ element sorted. After n iterations (where n is the length of the array), the whole array is sorted. The insertion sort algorithm is presented below:

```
insertionSort(int[] a)
{
    int n = a.length;
    for (int i = 1; i < n; i++)
    {
        // a is sorted from 0 to i-1
        int x = a[i];
        int j = i-1;
        while (j >= 0 && a[j] > x)
        {
            a[j+1] = a[j];
            j = j-1; }
        a[j+1] = x; }
}
```

The algorithm uses the variable x as a temporary buffer to hold the value of $a[i]$ while the larger elements in the sorted part of the array are shifted one position to the right. When all elements have been shifted, this value is written back into the appropriate position of the array.

In the first iteration of the outer loop, the algorithm performs at most one comparison operation, in the second iteration at most two, and so on. The worst case occurs when the input array is sorted in descending order, in which we have that

$$1+2+\dots+(n-1) = n*(n-1)/2 = n^2/2$$

comparisons (the number of element swap operations is approximately the same). Therefore, the time complexity of the algorithm is quadratic in the length n of the array. In average, however, in each iteration of the outer loop only half of the maximum number of comparisons has to be performed until the appropriate insertion point is found. Why? In this case the algorithm performs an average of approximately $n^2/4$ steps.

Sorting an array by this algorithm is therefore much slower than searching for an element in an array (which takes at most n steps, if the array is unsorted, and at most $\log_2 n$ steps, if the array is sorted). For instance, if the size of the array is 1024, approximately 262000 comparison operations have to be performed to sort the array. If the array has 8192 elements, approximately 16 million comparisons have to be performed.

However, for data that are already sorted or almost sorted, the insertion sort does much better. When data is in order, the inner loop is never executed. Why? In this case, the algorithm needs only n steps. If the data is almost sorted, the inner loop is only very infrequently executed and the algorithm runs in “almost” n steps. It is therefore a simple and efficient way to order a collection that is only slightly out of order.

Binary Tree Sort Algorithm

Given an array of elements to sort, the tree sort algorithms proceeds as follows:

- i. Build a binary search tree out of the elements
- ii. Traverse the tree in order and copy the elements back into the array.

This algorithm time complexity is as follows:

Worst case - Occurs when the binary search tree is unbalanced, for example when the data is already sorted. Building the binary search tree takes $O(n^2)$, and to traverse the binary tree takes $O(n)$ time. The total is $O(n^2) + O(n) = O(n^2)$.

Best case - Building the binary search tree takes $O(n \log n)$ and to traverse the binary tree takes $O(n)$. The total is $O(n \log n) + O(n) = O(n \log n)$.

Conclusion

This activity presented two sorting algorithms based on the insertion method. Both algorithms' performance were quadratic in the worst case. Such algorithms may be suited to sorting small data collections.

? Assessment

1. Write one to three sentences to answer each of the following questions:
 - a. Insertion sort is better for nearly-sorted lists than reverse-ordered lists. Why?
 - b. How much extra memory is required for each sort?
 - c. If the list is composed of four identical elements, how many elements change position when insertion sort is used? Why?
2. For each of the following lists, draw the result of each insertion of the insertion sorting routine. You do not need to show the result of each comparison, just the final insertion of the element.
 - a. [5 | 4 | 1]
 - b. [3 | 1 | 3]
 - c. [2 | 5 | 4 | 0]
3. Consider the following unsorted list: [6 | 8 | 3 | 5 | 1 | 9 | 2 | 2]. Show the steps used by the tree sort algorithm to obtain a sorted list.

This page titled [9.1: Activity 1 - Sorting Algorithms by insertion method](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Godfry Justo \(African Virtual University\)](#).

- [3.1: Activity 1 - Sorting Algorithms by insertion method](#) by Godfry Justo is licensed [CC BY-SA 3.0](#).

9.2: Activity 2 - Sorting Algorithms by selection method

Introduction

This activity will present two selection based sorting algorithms, the selection sort and the heap sort algorithms.

Activity Details

Selection sort

The selection sort algorithm works by repeatedly exchanging elements: first finds the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted. Below we present the selection sort algorithm:

SELECTION_SORT (A)

```
for i ← 1 to n-1 do
  min j ← i;
  min x ← A[i]
  for j ← i + 1 to n do
    If A[j] < min x then
      min j ← j
      min x ← A[j]
  A[min j] ← A [i]
  A[i] ← min x
```

Selection sort is among the simplest of sorting techniques and it work very well for small files. Furthermore, despite its evident “simplistic approach”, selection sort has a quite important application because each item is actually moved at most once, thus selection sort is a method of choice for sorting files with very large objects (records) and small keys. Below we present the selection sort implementation:

```
void selectionSort(int numbers[], int array_size)
{
  int i, j;
  int min, temp;
  for (i = 0; i < array_size-1; i++)
  {
    min = i;
    for (j = i+1; j < array_size; j++)
    { if (numbers[j] < numbers[min])
      min = j; }
    temp = numbers[i];
    numbers[i] = numbers[min];
    numbers[min] = temp; }
}
```

The complexity of the selection sort algorithm is as follows. The worst case occurs if the array is already sorted in descending order. Nonetheless, the time required by selection sort algorithm is not very sensitive to the original order of the array to be sorted: the test “if $A[j] < \text{min } x$ ” is executed exactly the same number of times in every case. The variation in time is only due to the number of times the “then” part (i.e., $\text{min } j \leftarrow j$; $\text{min } x \leftarrow A[j]$) of this test are executed. Selection sort is quadratic in both the worst and the average case, and requires no extra memory.

Note that for each i from 1 to $n - 1$, there is one exchange and $n - i$ comparisons, so there is a total of $n - 1$ exchanges and $(n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1)/2$ comparisons. These observations hold no matter what the input data is. In the worst case, this could be

quadratic, but in the average case, this quantity is $O(n \log n)$. It implies that the running time of selection sort does not depend on how the input may be organized.

Heaps Sort

We can use heap for sorting a collection in a specific order efficiently. Suppose we want to sort elements of array `Arr` in ascending order. We can use max heap to perform this operation through the following steps:

1. Build a max heap of elements in `Arr`.
2. Now the root element, `Arr[1]`, contains maximum element of `Arr`. Exchange this element with the last element of `Arr` and again build a max heap excluding the last element which is already in its correct position. Thus decreases the length of heap by one.
3. Repeat step (ii) until all elements are in their correct position.
4. Get a sorted array.

In the following the implementation of heap sort algorithm is provided (Assuming there are N elements stored in array `Arr`):

```
void heap_sort(int Arr[ ])
{
    int heap_size = N;
    build_maxheap(Arr);
    for(int i = N; i>=2 ; i-- )
    {
        swap(Arr[ 1 ], Arr[ i ]);
        heap_size = heap_size-1;
        max_heapify(Arr, 1, heap_size); }
}
```

The time complexity of the heap-sort algorithm is as follows: Recall from [Chapter 8](#) that `max_heapify` has complexity $O(\log N)$, `build_maxheap` has complexity $O(N)$ and the fact that the algorithm calls `max_heapify` $N-1$ times, therefore the complexity of `heap_sort` function is $O(N \log N)$.

We now demonstrate the steps of the heap sort algorithm using an example of an unsorted array `Arr` having 6 elements and corresponding max-heap as in Figure 9.2.1.

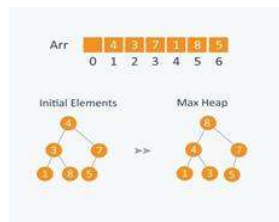


Figure 9.2.1: An initial array `Arr` with corresponding max-heap

After building the corresponding max-heap, the elements in the array `Arr` are now as in Figure 9.2.2.



Figure 9.2.2: An array `Arr` obtained from max-heap

The subsequent algorithm processing steps are outlined below and further depicted in Figure 9.2.3.

1. 8 is swapped with 5.
2. 8 is disconnected from heap as 8 is in correct position now and.
3. Max-heap is created and 7 is swapped with 3.
4. 7 is disconnected from heap.
5. Max heap is created and 5 is swapped with 1.
6. 5 is disconnected from heap.

7. Max heap is created and 4 is swapped with 3.
8. 4 is disconnected from heap.
9. Max heap is created and 3 is swapped with 1.
10. 3 is disconnected.

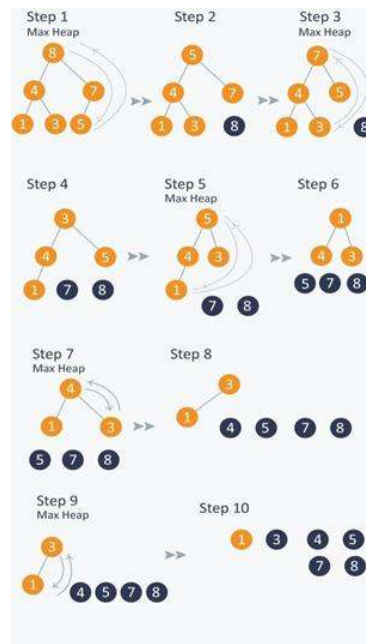


Figure 9.2.3: Heap-sort algorithm steps

Upon completion of all the steps, we obtain a sorted array shown in Figure 9.2.4.



Figure 9.2.4: A final sorted array

Conclusion

In this activity the selection based sorting algorithms were presented, the selection sort and the heap sort. The complexity analysis for the two algorithms shows that the heap sort is more efficient, since the selection sort has a quadratic worst case performance. The selection sort is suited to sorting small collections, and the order of input does not influence its performance.

Assessment

1. Show the steps taken when sorting the following list of integers [80, 30, 50, 70, 60, 90, 20, 30, 40] using
 - a. Selection sort

This page titled [9.2: Activity 2 - Sorting Algorithms by selection method](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Godfry Justo \(African Virtual University\)](#).

- [3.2: Activity 2 - Sorting Algorithms by selection method](#) by [Godfry Justo](#) is licensed [CC BY-SA 3.0](#).

9.3: Activity 3 - Sorting Algorithms by Merge and Partition methods

Introduction

This activity will present two sorting algorithms. The first algorithm called merge sort is based on merge method, and the second algorithm called quick sort is based on partition method.

Activity Details

Merge Sort Algorithm

The Merge sort algorithm is based on the divide-and-conquer strategy. Its worst-case running time has a lower order of growth than insertion sort. Since the algorithm deals with sub-problems, we state each sub-problem as sorting a sub-array $A[p \dots r]$. Initially, $p = 1$ and $r = n$, but these values change as recursion proceeds through sub-problems.

To sort $A[p \dots r]$, the merge sort algorithm proceeds as follows:

1. Divide step - If a given array A has zero or one element, simply return as it is already sorted. Otherwise, split $A[p \dots r]$ into two sub-arrays $A[p \dots q]$ and $A[q + 1 \dots r]$, each containing about half of the elements of $A[p \dots r]$.

That is, q is the halfway point of $A[p \dots r]$.

2. Conquer step - Conquer by recursively sorting the two sub-arrays $A[p \dots q]$ and $A[q + 1 \dots r]$.
3. Combine step - Combine the elements back in $A[p \dots r]$ by merging the two sorted sub-arrays $A[p \dots q]$ and $A[q + 1 \dots r]$ into a sorted sequence. This step will be accomplished by procedure $MERGE(A, p, q, r)$.

Note that the recursion ends when the sub-array has just one element, so that it is trivially sorted (base case). The following are the description of the merge sort algorithm: To sort the entire sequence $A[1 \dots n]$, make the initial call to the procedure $MERGE-SORT(A, 1, n)$.

$MERGE-SORT(A, p, r)$

1. IF $p < r$ // Check for base case
2. THEN $q = \text{FLOOR}[(p + r)/2]$ // Divide step
3. $MERGE(A, p, q)$ // Conquer step
4. $MERGE(A, q + 1, r)$ // Conquer step
5. $MERGE(A, p, q, r)$ // Conquer step

Figure 3.3.1 depicts a bottom-up view of the merge sort procedure for $n = 8$.

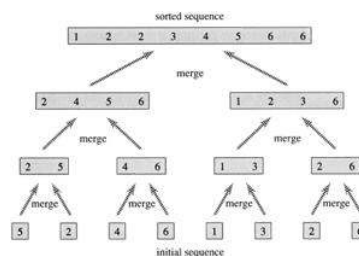


Figure 3.3.1 : merge sort for $n=8$

What remains is the $MERGE$ procedure. The following is the input and output of the $MERGE$ procedure.

INPUT - Array A and indices p, q, r such that $p \leq q \leq r$ and sub-array $A[p \dots q]$ is sorted and sub-array $A[q + 1 \dots r]$ is sorted. By restrictions on p, q, r , neither sub-array is empty.

OUTPUT- The two sub-arrays are merged into a single sorted sub-array in $A[p \dots r]$.

The merge procedure takes $\Theta(n)$ time, where $n = r - p + 1$, which is the number of elements being merged. The idea behind linear time merging can be easily understood by thinking of two piles of cards; each pile is sorted and placed face-up on a table with the smallest cards on top. Merge these into a single sorted pile, face-down on the table, by completing the following basic steps:

1. Choose the smaller of the two top cards.
2. Remove it from its pile, thereby exposing a new top card.
3. Place the chosen card face-down onto the output pile.
4. Repeat steps 1 - 3 until one input pile is empty.
5. Eventually, take the remaining input pile and place it face-down onto the output pile.

Each basic step takes constant time, i.e., $O(1)$, since it involves checking the two top cards. There are at most n basic steps, since each basic step removes one card from the input piles, and we started with n cards in the input piles. Therefore, this procedure should take $\Theta(n)$ time.

Now the question is do we actually need to check whether a pile is empty before each basic step?

The answer is no, we do not. Put on the bottom of each input pile a special guard card. It contains a special value that we use to simplify the code. We use ∞ , since that's guaranteed to drop to any other value. The only way that ∞ cannot drop is when both piles have ∞ exposed as their top cards. But when that happens, all the non-guard cards have already been placed into the output pile. We know in advance that there are exactly $r - p + 1$ non-guard cards so stop once we have performed $r - p + 1$ basic steps. There is no need to check for guards, since they will always drop. Rather than even counting basic steps, just fill up the output array from index p up through and including index r .

The pseudocode of the MERGE procedure is as follows:

MERGE (A, p, q, r)

```
1.  n1 ← q - p + 1
2.  n2 ← r - q
3.  Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
4.  FOR i ← 1 TO n1
5.    DO L[i] ← A[p + i - 1]
6.  FOR j ← 1 TO n2
7.    DO R[j] ← A[q + j ]
8.  L[n1 + 1] ← ∞
9.  R[n2 + 1] ← ∞
10. i ← 1
11. j ← 1
12. FOR k ← p TO r
13.  DO IF L[i] ≤ R[j]
14.    THEN A[k] ← L[i]
15.    i ← i + 1
16.  ELSE A[k] ← R[j]
17.    j ← j + 1
```

The time complexity of the merge procedure is as follows: The first two for loops (that is, the loop in line 4 and the loop in line 6) take $\Theta(n1 + n2) = \Theta(n)$ time. The last for loop (that is, the loop in line 12) makes n iterations, each taking constant time, for $\Theta(n)$ times. Therefore, the total running time is $\Theta(n)$.

The analysis of the merge sort algorithm is as follows: For convenience assume that n is a power of 2 so that each divide step yields two sub-problems, both of size exactly $n/2$. The base case occurs when $n = 1$.

When $n \geq 2$, time for merge sort steps are:

- Divide - Just compute q as the average of p and r , which takes constant time, i.e., $\Theta(1)$.
- Conquer- Recursively solve 2 sub-problems, each of size $n/2$, which is $2T(n/2)$.

- Combine- MERGE on an n-element sub-array takes $\Theta(n)$ time.

Summed together they give a function that is linear in n, which is $\Theta(n)$. Therefore, the recurrence for merge sort running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Solving the merge sort recurrence we can show that $T(n) = \Theta(n \lg n)$.

The implementation of the merge sort algorithm proceeds as follows:

```
void mergeSort(int numbers[], int temp[], int array_size)
{   m_sort(numbers, temp, 0, array_size - 1);   }

void m_sort(int numbers[], int temp[], int left, int right)
{   int mid;
    if (right > left)
    {   mid = (right + left) / 2;
        m_sort(numbers, temp, left, mid);
        m_sort(numbers, temp, mid+1, right);
        merge(numbers, temp, left, mid+1, right);   }

void merge(int numbers[], int temp[], int left, int mid, int right)
{   int i, left_end, num_elements, tmp_pos;
    left_end = mid - 1;
    tmp_pos = left;
    num_elements = right - left + 1;
    while ((left <= left_end) && (mid <= right))
    {   if (numbers[left] <= numbers[mid])
        {   temp[tmp_pos] = numbers[left];
            tmp_pos = tmp_pos + 1;
            left = left + 1;   }
        else
        {   temp[tmp_pos] = numbers[mid];
            tmp_pos = tmp_pos + 1;
            mid = mid + 1;   }
    while (left <= left_end)
    {   temp[tmp_pos] = numbers[left];
        left = left + 1;
        tmp_pos = tmp_pos + 1;   }
    while (mid <= right)
    {   temp[tmp_pos] = numbers[mid];
        mid = mid + 1;
        tmp_pos = tmp_pos + 1;   }
    for (i = 0; i <= num_elements; i++)
    {   numbers[right] = temp[right];
        right = right - 1;   }
```

Quick Sort Algorithm

The quick sort algorithm operates on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations because it is not difficult to implement, it is a good “general purpose” sort and it consumes relatively fewer resources during execution. The following are some pros and cons of the quick sort algorithm:

Pros

- well established and uses only a small auxiliary stack
- requires only $n \log(n)$ time to sort n items.
- has an extremely short inner loop

Cons

- is recursive. Especially if recursion is not available, the implementation is extremely complicated.
- is fragile i.e., a simple mistake in the implementation can go unnoticed and cause it to perform badly.

The quick sort algorithm works by partitioning a given array $A[p \dots r]$ into two non-empty sub array $A[p \dots q]$ and $A[q+1 \dots r]$ such that every key in $A[p \dots q]$ is less than or equal to every key in $A[q+1 \dots r]$. Then the two sub-arrays are sorted by recursive calls to quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure. In general, the quick sort algorithm performs the following steps:

Quick_Sort

1. If $p < r$ then
2. $q \leftarrow \text{Partition}(A, p, r)$
3. Recursive call to Quick Sort (A, p, q)
4. Recursive call to Quick Sort ($A, q + 1, r$)

Note that to sort the entire array, initially call $\text{Quick\_Sort}(A, 1, \text{length}[A])$.

As a first step, the quick sort algorithm would choose as a pivot an element from one of the items in the array to be sorted. Subsequently, the array is partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

The partition procedure is defined below:

PARTITION (A, p, r)

```
1.  $x \leftarrow A[p]$ 
2.  $i \leftarrow p-1$ 
3.  $j \leftarrow r+1$ 
4. while TRUE do
5. Repeat  $j \leftarrow j-1$ 
6. until  $A[j] \leq x$ 
7. Repeat  $i \leftarrow i+1$ 
8. until  $A[i] \geq x$ 
9. if  $i < j$ 
10. then exchange  $A[i] \leftrightarrow A[j]$ 
11. else return  $j$ 
```

The partition procedure selects the first key, $A[p]$ as a pivot key about which the array will be partitioned:

- Keys $\leq A[p]$ will be moved towards the left
- Keys $\geq A[p]$ will be moved towards the right

The time complexity of the partition procedure is $\Theta(n)$ where $n = r - p + 1$ which is the number of keys in the array.

The time complexity of quick sort algorithm depends on whether partition is balanced or unbalanced, which in turn depends on which elements of an array to be sorted are used for partitioning. A very good partition splits an array up into two equal sized arrays. A bad partition, on other hand, splits an array up into two arrays of very different sizes. The worst partition puts only one element in one array and all other elements in the other array. If the partitioning is balanced, the quick sort runs asymptotically as fast as merge sort. On the other hand, if partitioning is unbalanced, the quick sort runs asymptotically as slow as insertion sort.

In the best case each partitioning stage divides the array exactly in half. In other words, the best to be a median of the keys in $A[p \dots r]$ every time procedure PARTITION is called. The procedure PARTITION always split the array to be sorted into two equal sized arrays. If the procedure PARTITION produces two regions of size $n/2$, the recurrence relation is then:

$$\begin{aligned} T(n) &= T(n/2) + T(n/2) + \Theta(n) \\ &= 2T(n/2) + \Theta(n) \end{aligned}$$

And

$$T(n) = \Theta(n \lg n).$$

The worst-case occurs if the input array $A[1 \dots n]$ is already sorted. The call to PARTITION (A, p, r) always returns p , so successive calls to partition will split arrays of length $n, n-1, n-2, \dots, 2$. Hence, the running time is proportional to $n + (n-1) + (n-2) + \dots + 2 = [(n+2)(n-1)]/2 = \Theta(n^2)$. The worst-case also occurs if $A[1 \dots n]$ starts out in reverse order.

The implementation of the quick sort algorithm is presented below:

```
void quickSort(int numbers[], int array_size)
{q_sort(numbers, 0, array_size - 1);
}

void q_sort(int numbers[], int left, int right)
{
    int pivot, l_hold, r_hold;
    l_hold = left;
    r_hold = right;
    pivot = numbers[left];
    while (left < right)
    {
        while ((numbers[right] >= pivot) && (left < right))
            right--;
        if (left != right)
        {
            numbers[left] = numbers[right];
            left++;
        }
        while ((numbers[left] <= pivot) && (left < right))
            left++;
        if (left != right)
        {
            numbers[right] = numbers[left];
            right--;
        }
    }
}
```

```
numbers[left] = pivot;
pivot = left;
left = l_hold;
right = r_hold;
if (left < pivot)
    q_sort(numbers, left, pivot-1);
if (right > pivot)
    q_sort(numbers, pivot+1, right);
```

Conclusion

In this activity we presented two sorting algorithms, the merge sort and quick sort algorithm based on merge and partition approach respectively. Both algorithms use the divide-and-conquer strategy and have comparable run time complexity on average. Quick sort is a well established sorting algorithm whose worst-case running time is $\Theta(n^2)$ and expected running time is $\Theta(n \lg n)$ where the constants hidden in $\Theta(n \lg n)$ are small.

? Assessment

1. Why would in practice one use quick sort algorithm than merge sort algorithm even though, merge sort promises a run time complexity of $(n \lg n)$ for both worst and average cases?
2. Show the steps taken when sorting the following list of integers [80, 30, 50, 70, 60, 90, 20, 30, 40] using
 - a. merge sort
 - b. quick sort

This page titled [9.3: Activity 3 - Sorting Algorithms by Merge and Partition methods](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Godfry Justo](#) ([African Virtual University](#)) .

- [3.3: Activity 3 - Sorting Algorithms by Merge and Partition methods](#) by [Godfry Justo](#) is licensed [CC BY-SA 3.0](#).

9.4: Unit Summary

This unit presented six sorting algorithms based on insertion, selection, merge and partition methods. The insertion sort based algorithms, the insertion sort and binary tree sort have quadratic performance in the worst case, thus may be suited to sorting small data collections. The selection based sorting algorithms, the selection sort and the heap sort, give slight improvement in run time efficiency, though, generally also suffer from the quadratic worst case performance.

The merge sort and quick sort algorithm are based on merge and partition approach respectively, and both algorithms use the divide-and-conquer strategy. They have comparable run time complexity on average of $\Theta(n \lg n)$. Quick sort is a well established sorting algorithm whose worst-case running time is $\Theta(n^2)$ and expected running time is $\Theta(n \lg n)$ where the constants hidden in $\Theta(n \lg n)$ are small.

? Unit Assessment

Check your understanding!

Programming Exercises

Instructions

1. A merge of two sorted lists, e.g. `merge( [1;4;9;12], [2;3;4;5;10;13])` leads to a new sorted list `[1;2;3;4;4;5;9;10;12;13]`, made up from the elements of the arguments in the merge function. This operation can be declared so that it has a worst-case running time proportional to the sum of the length of the argument lists. Declare such a function. Test this function so that you are sure that all branches of the declaration work correctly.

Answers

<mailto:njulumi@gmail.com>

Grading Scheme

As per the offering Institution grading policy.

Unit Readings and Other Resources

The readings in this unit are to be found at course level readings and other resources.

This page titled [9.4: Unit Summary](#) is shared under a [CC BY-SA](#) license and was authored, remixed, and/or curated by [Godfry Justo](#) ([African Virtual University](#)) .

- [3.4: Unit Summary](#) by [Godfry Justo](#) is licensed [CC BY-SA 3.0](#).

Index

L

laws of recursion

[2.1: Activity 1 - Recursive Algorithm](#)

R

recursive algorithm

[2.1: Activity 1 - Recursive Algorithm](#)

S

sorting

[9.2: Activity 2 - Sorting Algorithms by selection method](#)

Glossary

Sample Word 1 | Sample Definition 1

Detailed Licensing

Overview

Title: CISP 430: Data Structures (Aljuboori)

Webpages: 62

All licenses found:

- [CC BY-SA 4.0](#): 61.3% (38 pages)
- [Undeclared](#): 19.4% (12 pages)
- [CC BY 4.0](#): 19.4% (12 pages)

By Page

- [CISP 430: Data Structures \(Aljuboori\)](#) - *Undeclared*
 - [Front Matter](#) - *Undeclared*
 - [TitlePage](#) - *Undeclared*
 - [InfoPage](#) - *Undeclared*
 - [Table of Contents](#) - *Undeclared*
 - [Licensing](#) - *Undeclared*
 - [1: Introduction](#) - *CC BY 4.0*
 - [1.1: The Need for Efficiency](#) - *CC BY 4.0*
 - [1.2: Stanley Milgram](#) - *Undeclared*
 - [1.3: Watts and Strogatz](#) - *Undeclared*
 - [2: Recursion](#) - *CC BY-SA 4.0*
 - [2.1: Activity 1 - Recursive Algorithm](#) - *CC BY-SA 4.0*
 - [2.2: Unit Summary](#) - *CC BY-SA 4.0*
 - [3: Stacks and Queues](#) - *CC BY-SA 4.0*
 - [3.1: Stacks](#) - *CC BY-SA 4.0*
 - [3.2: Applications of Stacks](#) - *CC BY-SA 4.0*
 - [3.3: Queues](#) - *CC BY-SA 4.0*
 - [3.4: Deques](#) - *CC BY-SA 4.0*
 - [4: Linked Lists](#) - *CC BY 4.0*
 - [4.1: SLList - A Singly-Linked List](#) - *CC BY 4.0*
 - [4.2: DLList - A Doubly-Linked List](#) - *CC BY 4.0*
 - [4.3: SEList - A Space-Efficient Linked List](#) - *CC BY 4.0*
 - [4.4: Discussion and Exercises](#) - *CC BY 4.0*
 - [5: Trees](#) - *CC BY-SA 4.0*
 - [5.1: Traversal](#) - *CC BY-SA 4.0*
 - [5.2: Sample implementations for Tree Traversal](#) - *CC BY-SA 4.0*
 - [5.3: Examples of Tree Traversals](#) - *CC BY-SA 4.0*
 - [5.4: Balancing](#) - *CC BY-SA 4.0*
 - [5.5: Binary Search Trees](#) - *CC BY-SA 4.0*
 - [5.6: Example extract of Java code for binary tree delete operation](#) - *CC BY-SA 4.0*
 - [5.7: Red-Black trees](#) - *CC BY-SA 4.0*
 - [5.8: B Trees](#) - *CC BY-SA 4.0*
 - [5.9: Treaps](#) - *CC BY-SA 4.0*
 - [5.10: References](#) - *CC BY-SA 4.0*
 - [6: Graphs](#) - *CC BY 4.0*
 - [6.1: Representing a Graph by a Matrix](#) - *CC BY 4.0*
 - [6.2: A Graph as a Collection of Lists](#) - *CC BY 4.0*
 - [6.3: Graph Traversal](#) - *CC BY 4.0*
 - [6.4: Discussion and Exercises](#) - *CC BY 4.0*
 - [7: Hash Tables](#) - *CC BY-SA 4.0*
 - [7.1: Time complexity and common uses of hash tables](#) - *CC BY-SA 4.0*
 - [7.2: Choosing a good hash function](#) - *CC BY-SA 4.0*
 - [7.3: Collision resolution](#) - *CC BY-SA 4.0*
 - [7.4: Table resizing](#) - *CC BY-SA 4.0*
 - [7.5: Ordered retrieval issue](#) - *CC BY-SA 4.0*
 - [7.6: Problems with hash tables](#) - *CC BY-SA 4.0*
 - [7.7: Other hash table algorithms](#) - *CC BY-SA 4.0*
 - [7.8: Implementations](#) - *CC BY-SA 4.0*
 - [8: Searching, Binary Search Trees and Heaps](#) - *CC BY-SA 4.0*
 - [8.1: Activity 1 - Searching Techniques](#) - *CC BY-SA 4.0*
 - [8.2: Activity 2 - Binary Search Tree](#) - *CC BY-SA 4.0*
 - [8.3: Activity 3 - Heaps](#) - *CC BY-SA 4.0*
 - [8.4: Unit Summary](#) - *CC BY-SA 4.0*
 - [9: Sorting Algorithms](#) - *CC BY-SA 4.0*
 - [9.1: Activity 1 - Sorting Algorithms by insertion method](#) - *CC BY-SA 4.0*
 - [9.2: Activity 2 - Sorting Algorithms by selection method](#) - *CC BY-SA 4.0*
 - [9.3: Activity 3 - Sorting Algorithms by Merge and Partition methods](#) - *CC BY-SA 4.0*
 - [9.4: Unit Summary](#) - *CC BY-SA 4.0*
 - [Back Matter](#) - *Undeclared*
 - [Index](#) - *Undeclared*

- [Glossary](#) - *Undeclared*

- [Detailed Licensing](#) - *Undeclared*