Visualizing Data Structures

Copyright

Copyright © 2015 by Rhonda Hoenigman
All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the publisher except for the use of brief quotations in a book review or scholarly journal.
First Printing: 2015
ISBN: 978-1-329-13302-0
Rhonda Hoenigman
Boulder, CO 80303
Distributor:
Lulu Publishing
3101 Hillsborough St.
Raleigh, NC 27607
Cover photo by Justin Morse.

Preface

The first time I taught *CS2 - Data Structures and Algorithms* at the University of Colorado, the available course textbooks that I found were either advanced programming books that obscured the details on the data structures concepts or theory books that lacked sufficient details on implementations. Over the course of the semester, I wrote copious notes to fill the gaps in our selected course textbook and provided them to my students. By the end of the semester, I had a draft of a data structures book that was exactly the book for which I had been searching. I decided to publish the material as a self-published e-book so that it would be available as inexpensively as possible for anyone who was interested.

The intended audience for this book is second-semester computer science undergraduates. The focus is on fundamental concepts of data structures and algorithms and providing the necessary detail for students to implement the data structures presented. The content included herein is what I was able to cover in a one-semester course.

This book sets itself apart from other data structures books in the following ways:

• The data structures are presented in pictures. There are pictures of arrays, linked lists, graphs, trees, and hash tables that help students visualize the algorithms on these structures. General feedback from my students was that they really appreciated the pictures I drew in class, and I have included all of them here.

• There are step-by-step descriptions of how algorithms work. These descriptions illustrate the state of the data structure at key lines in an algorithm’s execution.

• The algorithms are presented in a language that I call “pseudocode with C++ tendencies.” In other words, there

is enough detail in the pseudocode for students to convert it to C++. In many cases, basic C++ is also provided.
I am utilizing this book in my current courses, and hope that you as a student or instructor will find this book useful. Good luck, and happy programming.

Rhonda Hoenigman, PhD
University of Colorado, Boulder
2015

About the author

Rhonda Hoenigman earned her Ph.D. in Computer Science in 2012 from the University of Colorado. She has been an Instructor in that Department since 2013. Dr. Hoenigman's research interests include predictive modeling and optimization of issues surrounding food waste and food justice systems. She has taught undergraduate courses in introductory computer programming, data structures and algorithms, discrete structures, and artificial intelligence.

Introduction

This book is intended for computer science students who understand the basics of programming and are ready to launch into a discovery of data structures, which are fundamental to an appreciation of the field of computer science. Typically, these are students with a semester of programming experience, and are in a second semester data structures course. To understand the material presented herein, the reader should have an understanding of C++ or another object-oriented language, such as Java.

The emphasis in this book is on presenting fundamental data structures and the algorithms used to access information stored in these structures. Many of the data structures are also presented in the context of an abstract data type (ADT), which is the implementation mechanism commonly used in an object-oriented language. Algorithms are presented as part of an ADT. Pseudo-code is used throughout the book, for both the algorithms as well as the ADT definitions.

What is a data structure?

A data structure is a specialized format for organizing related information. Depending on the type of information, one data structure could provide a better arrangement for storing the data than another data structure, where “better” refers to the ability to access and manipulate the data efficiently. Basic data structures are itemized below, with a brief description of each:

- **Arrays:** Fixed-length linear sequence of similar elements, where each individual element can be accessed by its index.
- **Lists:** Linear sequence of similar elements that can expand and contract as needed.
- **Trees:** Collection of elements with a hierarchical structure.
- **Maps:** Collection of elements that are accessed through one property of the element, known as a key.
- **Records:** Composite data type that is composed of other data elements, called fields or members.

This list of data structures is by no means exhaustive, nor is each data structure independent of the other structures on the list. For example, **maps** are generally composed of **records**, and an **array** is commonly used to implement a **map**. Modifying the behavior of the basic data structure can also create new data structures. For example, an **array** or **list** where elements can only be added and removed at the last position is called a **stack**.

What is an Abstract Data Type?

An Abstract Data Type (ADT) is a collection of data elements and the allowable operations on those elements. In an ADT, the operations are encapsulated; the user only has information about the inputs, outputs, and an explanation of the actions. The specific details of the operations are hidden. The data elements in an ADT are stored in a data structure. The algorithms to access and manipulate that data structure are implemented as methods in the ADT.

Algorithms and pseudocode

When presenting the details of how specific algorithms perform, a book must balance providing enough detail to convey the nuances of the algorithm with getting mired in the syntax of a particular programming language. For these reasons, algorithms are often presented in pseudocode in this and other books. Pseudocode is a simplified description of the algorithm generated from real code that is intended to be more readable than real computer code while still providing the detail necessary to understand the complexity of a specific algorithm. Just as with real code, it can take practice to read and understand pseudocode, and part of this understanding comes from an awareness of the pseudocode conventions used in a particular presentation. Pseudocode conventions used herein are described below.

Pseudocode conventions

The algorithms in this book are presented in a language that I will call “pseudocode with C++ tendencies”. The pseudocode presented herein may appear informal when compared to pseudocode in other data structures and algorithms books because it preserves more of the C++ language than typical pseudocode.

• Expressions are presented using = and == to represent assignment and equivalence, respectively, just as in real code.
• **For** loops include only an initial condition and an end state: **for x = 0 to A.end**, where *A.end* is the last index in the array.
• Indentation is used to specify which lines are included in an execution block.
• Data types are removed from variable definitions and return values.

• The words “and” and “or” replace && and || in conditionals.

The following snippets show the real code and the pseudocode for an algorithm that returns the index in an array for a specified search value.

Real code int findItem(int[ ] A, int v, int length) 1. int index = -1; 2. for(int x = 0; x < length; x++) {
3. if(A[x] == v) 4. return x; 5. }
6. return index;
Pseudocode findItem(A, v) 1. index = -1
2. for x=0 to A.end 3. if(A[x] == v) 4. return x 5. return index
In the function definition, the real code includes the return type of the function as well as the types of the function parameters. The real code also includes an additional parameter, *length*, which is the size of the array. In the pseudocode, *A.end* is used for the last index in the array to convey that the **for** loop will execute for each element in *A*.

Roadmap

The next chapter in this book presents an introduction to algorithms and why they are essential in any computer-science education. Computer memory is presented in Chapter 2 to provide the foundation for understanding how data structures are allocated and destroyed dynamically. Chapter 3 presents the concept of arrays and the algorithms necessary for array manipulation. Chapter 4 includes an introduction to sorting algorithms and the fundamentally different approaches to sorting, and how these approaches present tradeoffs in their implementation and behavior. The other chapters in the book cover the following data structures and ADTs: arrays, linked lists, stacks, queues, trees, and binary search trees, red-black trees, graphs, and hash tables. Linked lists are presented in Chapter 5. Stacks and queues are presented in Chapters 6 and 7, respectively. The discussion of trees, including binary search trees and red-black trees and recursive algorithms for traversing trees, is covered in Chapters 8 - 11. Finally, graphs are in Chapter 12 and hash tables are in Chapter 13.

1 Algorithms

In any computer program, there is a specific set of instructions that tells the computer what to do. This set of instructions is similar to a recipe in that there is an objective to accomplish (problem to solve), and a set of steps in a specified order to accomplish the objective. These instructions are also known as an algorithm: a defined set of steps that are followed to solve a problem. As an example, an algorithm that puts the following sequence of numbers in ascending order
<54, 34, 23, 45, 56, 90>
would produce an output of
<23, 34, 45, 54, 56, 90>.
Some algorithms are simple, such as an equation that adds two numbers. Other algorithms, such as a pattern-matching algorithm that compares two gene sequences, are very complex; it is these more complex algorithms that computer scientists generally care about. The primary concerns with algorithms are how are they specified and how they scale. On a small data set, an algorithm might work fine and produce the expected output in a reasonable amount of time. However, on a large data set, the algorithm might break down and take an intractable amount of time to produce a result. Provided that the algorithm produces the correct solution, understanding how an algorithm is going scale with the size of the input is the primary evaluation of whether an algorithm is “good.”

1.1 Specifying an algorithm

Algorithms generally have a set of inputs, and then transform these inputs in some way to produce an output. The specifications for an algorithm are documented by pre- and post-conditions, which inform anyone using the algorithm what to expect.

1.1.1 Pre-condition

The pre-conditions for an algorithm are the conditions that must be true prior to the algorithm's execution in order for it to work as defined. Pre-conditions can include the inputs to the algorithm and the restrictions on the types and range of values on those inputs. Pre-conditions can also include other dependencies, such as other algorithms that need to execute first.

1.1.2 Post-condition

The post-conditions for an algorithm are the expected changes, or the return value, after the algorithm executes. For example, a function to calculate the factorial of a particular number could look like:

```
factorial(n)
```

The pre-condition on *factorial(n)* is that *n* is an integer greater than 0. The function is not expected to work correctly for values of *n* that do not meet the conditions. The post-condition is the function returns the factorial of *n*.

1.2 Evaluating an algorithm

Given a problem to solve, such as sorting a sequence of numbers, it is important to evaluate whether one algorithm is better than another algorithm in terms of correctness, efficiency, and resource use.

1.2.1 Correctness

First and foremost, the algorithm needs to produce a correct solution; it does not matter how efficient the algorithm is, if it produces an incorrect answer. For example, if a sorting algorithm intended to sort integers from highest to lowest produces a solution such as
A = <90, 54, 23, 34, 45, 56>
then any other method of evaluation is irrelevant.

1.2.2 Cost

Given that an algorithm is correct, it can be evaluated by its cost, where cost is generally evaluated as the memory usage and runtime of the algorithm. It is difficult to evaluate an algorithm's runtime empirically as doing so would involve running the algorithm for a representative set of inputs and measuring the results, which could be affected by the hardware platform and the software implementation. Instead of an empirical evaluation, the cost is calculated theoretically by evaluating the number of lines of code that execute. In this approach, each line is assumed to have a cost of 1 to simplify the cost calculation. A count of the lines of code provides a high-level estimate of the algorithm's runtime that will be roughly proportional to the actual runtime.

An example of an algorithm specification that can be used to calculate the cost is shown in Algorithm 1.1. The specification shows the name and parameters of the

algorithm, the pre- and post-conditions, and the algorithm itself with line numbers.

Algorithm 1.1. findItem(A, v) Returns the index of the value *v* in the array *A*.

Pre-condition *A* is an array.
v is the same type as the elements in *A.*

Post-condition Returns the last index *x* where *A[x] = v.*

Algorithm findItem(A, v) 1. index = -1
2. for x=0 to A.end 3. if A[x] == v 4. index = x 5. return index

Example 1: Calculate the cost of findItem(A,v) for various inputs of A and v.

Example 1.1
A = < 45, 34, 32, 34 > findItem(A, 34)
Line number: Times executed Line 1: 1
Line 2: 5
Line 3: 4
Line 4: 2
Line 5: 1

In this call to *findItem(),* Line 1 executes one time. There are four elements in the array, and Line 2 executes once for each array element and once for the final evaluation of the **for** loop to check if *A.end* has been reached. Each line in the **for** loop can execute up to four times. Line 3 executes four times and line 4 executes twice, once for each time that 34 is found in the array. Line 5 executes one time. The total cost is 13.

Example 1.2

A = < 45, 34, 32, 34 > findItem(A, 25)
Line number: Times executed Line 1: 1
Line 2: 5
Line 3: 4
Line 4: 0
Line 5: 1

The cost difference between this call to *findItem()* and the previous example is in the number of times that the search value exists in the array. There are no instances of 25, so Line 4 never executes and the cost is reduced to 11.

Example 1.3
A = < 45, 34, 32, 34, 56, 23, 12 > findItem(A, 34)
Line number: Times executed Line 1: 1
Line 2: 8
Line 3: 7
Line 4: 2
Line 5: 1

The difference between this call to *findItem()* and the previous example that searched for the 34 is the size of the array *A*. This array has 7 elements instead of 4, which increases the cost. The **for** loop executes 7 times. There are two instances of 34 in the array, which results in Line 4 executing two times. Lines 1 and 5 still execute one time each. The total cost is 19.

Example 1.4
A = < 45, 45, 45, 45, 45, 45, 45 > findItem(A, 45)
Line number: Times executed Line 1: 1
Line 2: 8
Line 3: 7
Line 4: 7
Line 5: 1

In this example, every element in the array is the value being searched for. The result is that Line 4 will execute 7 times. All other costs are the same as those in the previous example with 7 elements. This scenario represents the worst-case cost for this algorithm; all lines in the algorithm execute the maximum number of times. The total cost is 23.

In all of these examples, the **for** loop is the biggest contributor to the cost of the algorithm, and the number of iterations of the **for** loop is determined by the size of the array *A*. Regardless of the contents of the array, the algorithm always loops through the entire array, and returns the last index of the search value.

In the next example, Algorithm 1.2 includes an exit when the search value is found. The **for** loop is still the biggest contributor to the algorithm cost, but the cost and the result of the algorithm both change for the specified input values.

Algorithm 1.2. findItemAndExit(A, v) Returns the first index of the value *v* in the array *A*.

Pre-condition *A* is an array.
v is the same type as the elements of A.

Post-condition Returns the first index *x* where *A[x] = v*.

Algorithm findItemAndExit(A, v) 1. index = -1
2. for i=0 to A.end 3. if A[i] == v 4. return i 5. return index

Example 2: Calculate the cost of findItemAndExit(A, v) for various inputs of A and v.

Example 2.1
A = < 45, 34, 32, 34 > findItemAndExit(A, 34)

Line number: Times executed Line 1: 1
Line 2: 2
Line 3: 2
Line 4: 1
Line 5: 0

In this example, the element is found in the array in the second position. Line 1 executes one time. Lines 2 and 3 both execute two times. On the second execution of Line 3, the conditional is true and Line 4 executes and the algorithm exits. Line 5 never executes. The total cost is 6.

Example 2.2
A = < 45, 34, 32, 34 > findItemAndExit(A, 25)
Line number: Times executed Line 1: 1
Line 2: 5
Line 3: 4
Line 4: 0
Line 5: 1

In this example, the search value is not found in the array. There are 4 iterations of the **for** loop, one for each element in the array, plus one additional evaluation of the **for** loop conditional. Line 3 executes 4 times and Line 4 executes zero times. Line 5 executes one time and the algorithm exits. The total cost is 11.

Example 2.3
A = < 45, 34, 32, 34, 56, 23, 12 > findItemAndExit(A, 34)
Line number: Times executed Line 1: 1
Line 2: 2
Line 3: 2
Line 4: 1
Line 5: 0

In this example, the search value is found in the array and the array size is larger than in the previous examples for this algorithm. The algorithm exits as soon as the value is found, so there are still only two iterations of the **for** loop and both Lines 2 and 3 execute two times. Line 4 executes one time and Line 5 never executes. The total cost is 6.

Example 2.4
A = < 45, 45, 45, 45, 45, 45, 45 > findItem(A, 45)
Line number: Times executed Line 1: 1
Line 2: 1
Line 3: 1
Line 4: 1
Line 5: 0

In this example, the value being searched for is the first element in the array. This scenario is the configuration with the minimum cost for this algorithm. There is only one iteration of the **for** loop. Line 4 executes one time and line 5 never executes. The total cost is 4.

The worst-case cost scenario is different for the *findItem()* (Algorithm 1.1) and *findItemAndExit()* (Algorithm 1.2) algorithms. For *findItem()*, the worst-case cost happens when every element in the array is the value being searched for. For *findItemAndExit()*, the worst-case cost happens when the value is not found in the array. However, common to both algorithms is that the **for** loop, which is set by the size of the input array, is the biggest contributor to the cost of the algorithm. As the size of the array grows, the **for** loop requires more iterations to traverse the entire array and the worst-case cost increases.

1.3 Algorithm Analysis

Evaluating how algorithms perform is called algorithm analysis. How the runtime of an algorithm scales with the size of the input can be described by a mathematical function. Presented here are common mathematical functions with behavior that is often observed in algorithm performance. In these examples, n is the input to the function and the value of n is the size of the data being evaluated in an algorithm.

1.3.1 The Constant Function

A constant function is defined as:
$f(n) = c$
where c is a fixed constant such as $c = 5$, $c = 1$, or $c = 5^{10}$. The variable n is the size of the data that needs to be evaluated. With a constant function, the output of the function is not dependent on the value or size of the input n, the output will always be the same regardless of n.

Constant function examples • Variable assignment, such as a = 5
• Inserting an element to the front of a linked list • Overwriting an element in an array • Accessing an element in a hash table

1.3.2 The Logarithmic Function

The logarithmic function is defined as:
$f(n) = \log_b n$
where b is the base of the logarithm. In computer science, $\log_2 n$ is so ubiquitous that the 2 is often left off and $\log_2 n$ is written as log n.

Logarithmic function examples • The minimum height of a binary search tree • Searching for an element in a binary search tree with a height of log n

1.3.3 The Linear Function

The linear function is defined as:
f(n) = n
The output of this linear function example is the value of *n* itself. A function that includes a constant, such as *f(n) = 2n* or *f(n) = n + 2*, also qualifies as a linear function.

Linear function examples • Traversing the elements in a linked list • Traversing the elements of a one-dimensional array • Shifting the elements in a one-dimensional array

1.3.4 The N-Log-N Function

The n-log-n function is defined as:
f(n) = nlog(n)
In an n-log-n function, the log(n) calculation is repeated *n* times.

N-Log-N function examples • *n* searches on a binary search tree with a height of log n • The merge sort algorithm

1.3.5 The Quadratic Function

The quadratic function is defined as:
$f(n) = n^2$

Quadratic function examples • Traversing a 2D matrix with *n* rows and *n* columns • Algorithms with two **for** loops, where one **for** loop nested in the other **for** loop • The bubble sort algorithm

1.3.6 The Cubic Function

The cubic function is defined as:
$f(n) = n^3$

Cubic function example • Algorithms with three nested **for** loops

1.3.7 The Exponential Function

The exponential function is defined as:

$f(n) = b^n$

where *b* is a positive constant called the base. In computer science, the most common base is 2, which means that an algorithm can be described on the order of

$f(n) = 2^n$

Exponential function example • Iterating through all possible combinations of binary data. For example, if given a binary string of 8 bits, there will be 2^8 combinations of those bits.

1.4 Growth-rate comparisons

The functions presented all have different asymptotic behavior, which is the behavior as the input n goes to infinity. In algorithm analysis, it is the asymptotic behavior that is of primary concern. Comparing the growth behavior of individual functions shows why one algorithm might be desirable over another algorithm with different asymptotic behavior. The functions described were presented in order of their growth rates slowest to fastest, with the constant function having a growth rate of zero and the exponential function growing the fastest.

In Figure 1, the growth rates for a constant, linear, and log(n) function are shown. On the x-axis, the parameter n is the size of the input data. For example, for an input size of 5 and 40, the linear function has an output of 5 and 40, respectively. The constant functions have the same behavior regardless of the values of n. The log(n) function has the slowest growth rate of the three, and the linear function has the fastest growth rate.

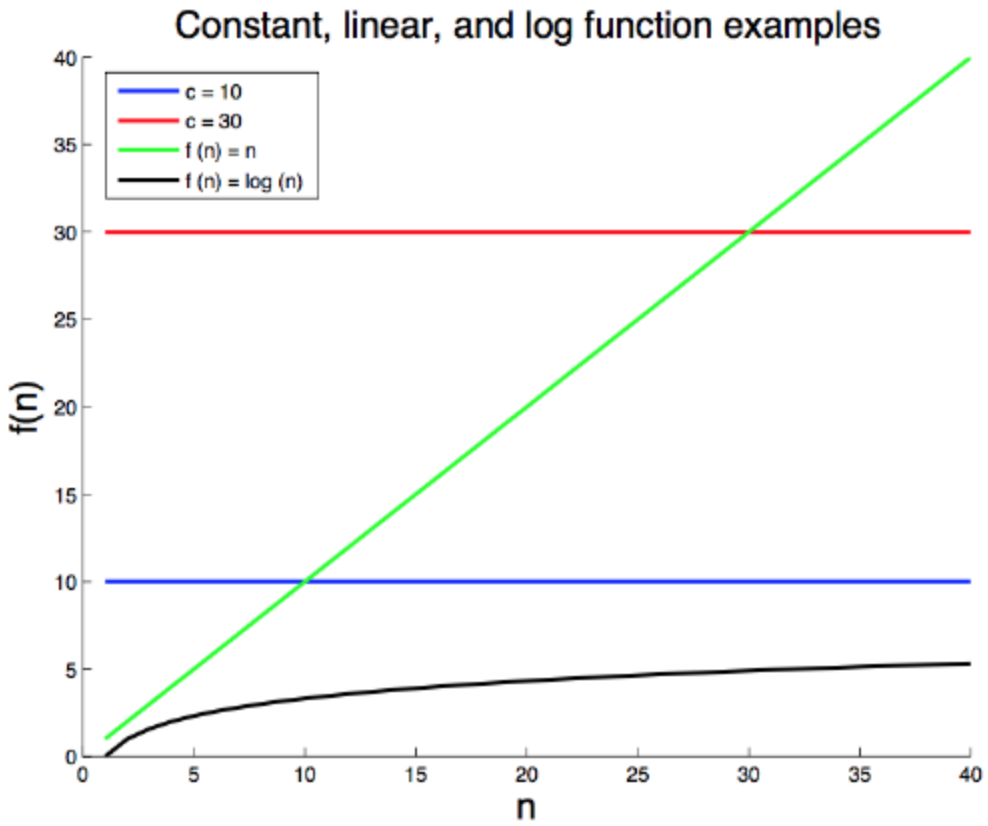


Figure 1. Comparison of growth rates for a constant, linear, and log(n) function for input size n. The linear function grows faster than the log function.

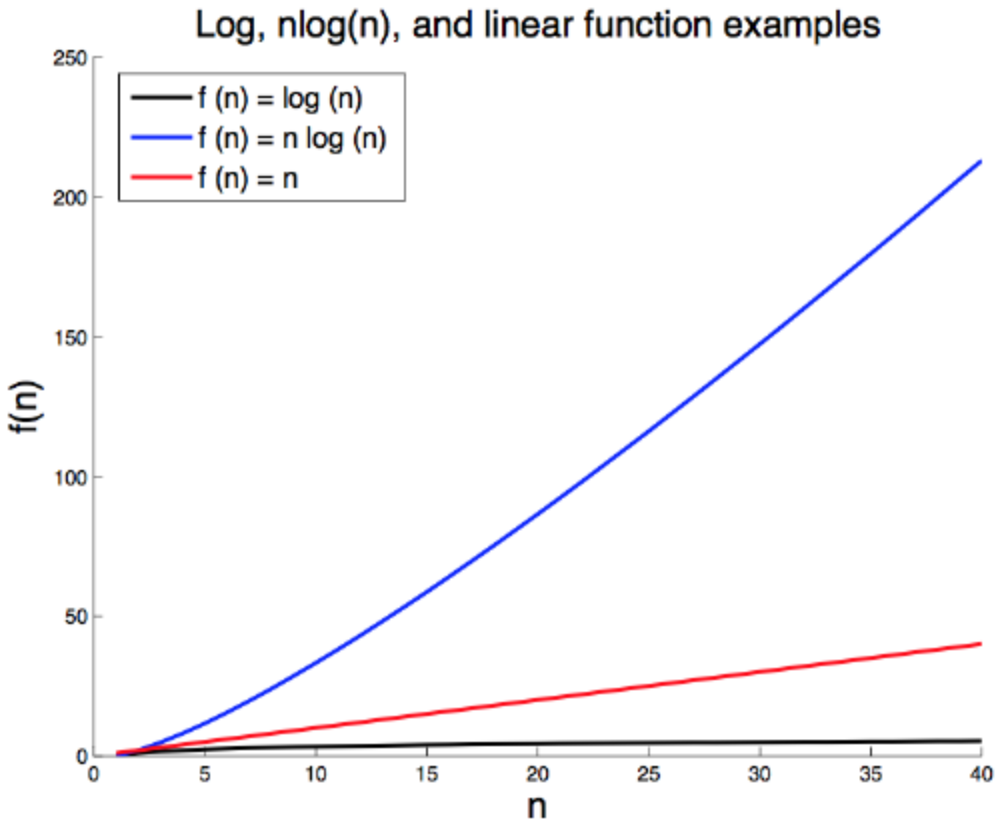


Figure 2. Comparison of growth rates for log(n), nlog(n), and linear functions for a given input size n. The nlog(n) function grows the fastest of the three functions.

The examples in Figure 2 show that, while the linear function grows more quickly than the log(n) function, an algorithm with nlog(n) behavior will have a longer runtime than either the linear or log(n) functions for large values of n. However, even the nlog(n) function doesn’t grow as fast as the quadratic function shown in Figure 3.

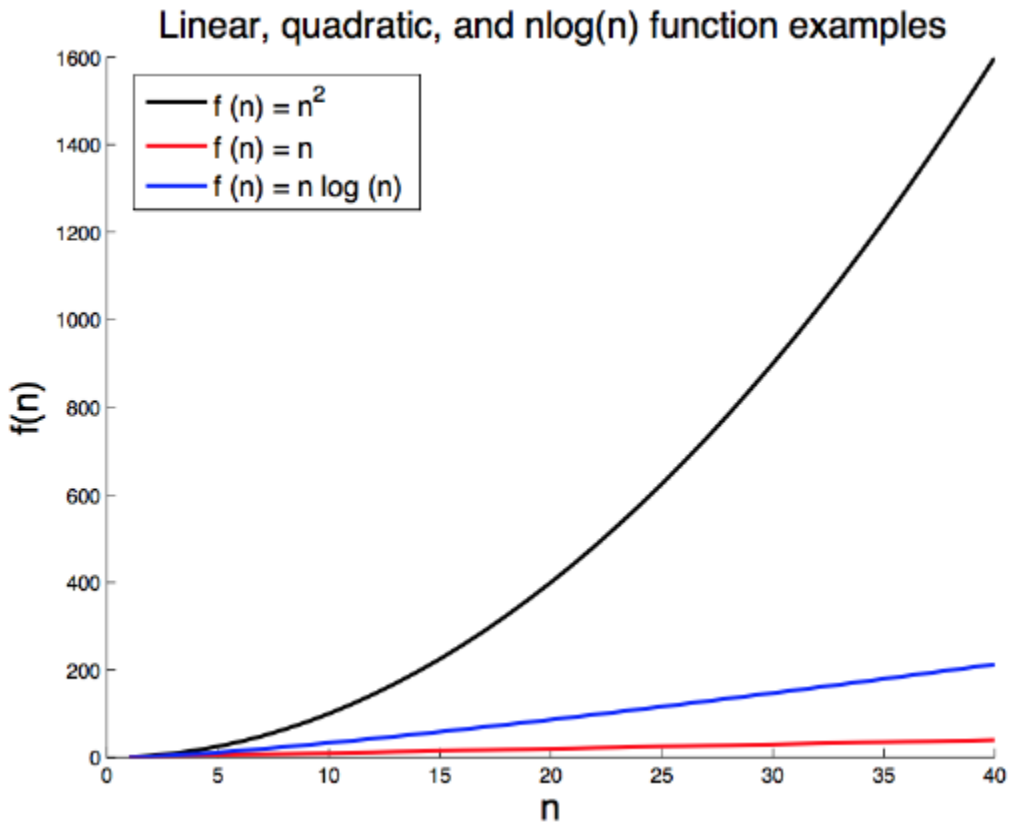


Figure 3. Comparison of growth rates for linear, quadratic, and nlog(n) functions. The x-axis is the input size of the data. The quadratic function is clearly the fastest growing of the three functions.

Even after only 40 data points, which is a small input, the quadratic function clearly is growing much more quickly than the linear or nlog(n) functions. However, none of the functions presented can compare to the growth of the exponential function, as shown in Figure 4. Any algorithm with exponential behavior will likely have very poor performance for large input sizes.

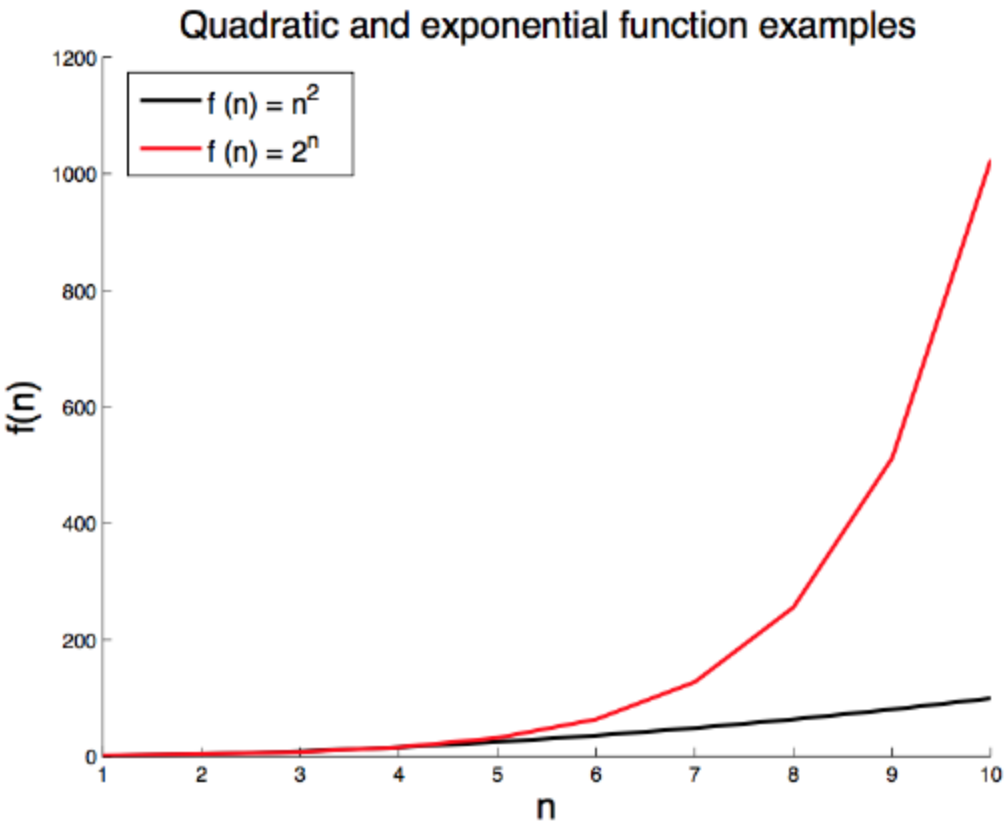


Figure 4. Comparison of growth rates for quadratic and exponential functions for a given input size n. Exponential growth rate typically equals very bad performance for large n.

1.5 Algorithmic Complexity

Counting how many times lines are executed in an algorithm provides an estimate for the runtime. The actual cost varies by computer and programming language. A faster computer will have a lower cost, and functionality can be 1 line of code in 1 language and 10 lines of code in another. Ultimately, the most important considerations for runtime are what contributes the most to its runtime and how the algorithm scales as the size of the input grows. For example, in the *findItem()* algorithm, the significant contributor to the runtime was the for loop. When the size of the array is small, e.g. 10, there are 10 iterations of the loop and three constant operations; the constants contribute significantly to the cost. However, as the array size grows, the constants become less significant. If the array size is 10,000, then 10,000 + 3 is not that different than 10,000.

1.5.1 Asymptotic Analysis

Suppose an algorithm for processing financial data takes 10,000 milliseconds to download the data from the Internet, and then 10 milliseconds to process each transaction (stocks bought and sold). Processing n transactions takes (10,000 + 10 n) milliseconds.

Even though 10,000 > 10, the "10 n" term will be more important if the number of transactions is very large. After 1000 transactions, the quantities will be equal. The values 10,000 and 10 are coefficients that will change with a faster computer or Internet connection, or use a different language or compiler. Analyzing the algorithm requires a means for expressing the theoretical speed of an algorithm that is independent of the environment in which the algorithm is implemented. This analysis is accomplished by ignoring the constant factors.

1.5.2 Big-Oh Notation

Big-Oh notation provides the upper bound on how quickly two functions grow as the input size *n -> infinity*.

Let *n* be the size of a program's (algorithm's) input. (The input is any data type: bits, numbers, words, or strings.).

Let *T(n)* be a function that represents the algorithm's precise running time in milliseconds, given an input of size *n*. This includes the specific instructions and the actual runtime of each instruction.

Let *f(n)* be a simple mathematical growth function, such as *f(n) = n,* a function that grows at a rate of *n*. The most common growth functions were given in the previous section.

The growth rate of *T(n)* can be expressed by relating it to another growth function. If *T(n)* grows no faster than *f(n)*, then:
T(n) is in O(f(n))
or

T(n) is in O(n)
if and only if
T(n) <= c f(n)
whenever *n* is big, for some large constant *c*.

The terms "big" and "large" are not very specific. The value of *n* needs to be big enough to make *T(n)* fit under *c f(n)* curve. The value of c needs to be large enough to make *T(n)* fit under the *c f(n)* curve.

Example 3: For the function *T(n) = 10,000 + 10n,* choose *c* to be large enough to make *T(n)* fit

underneath *cf(n)*.
Using *c = 20*, the graph in Figure 5 shows the asymptotic behavior of the algorithm. Above a certain value of *n*, the size of the data is a bigger contributor to runtime than the startup cost and *20*n* grows faster than *10,000*10n*.

When considering asymptotic behavior, multiplying by a positive constant does not change the result. A different constant can change where function values cross, i.e. where the cost of calculations dependent on input size outweighs any initialization costs. If *N* is the value where the function values cross, then in the financial data example with *c=20* and *N = 1000*. If there are only a few transactions, then the 10,000 millisecond startup might not be worth it and a less-efficient algorithm with lower startup costs could be a better choice. However, with many, many transactions, the startup cost becomes less of a contributor to the overall cost.

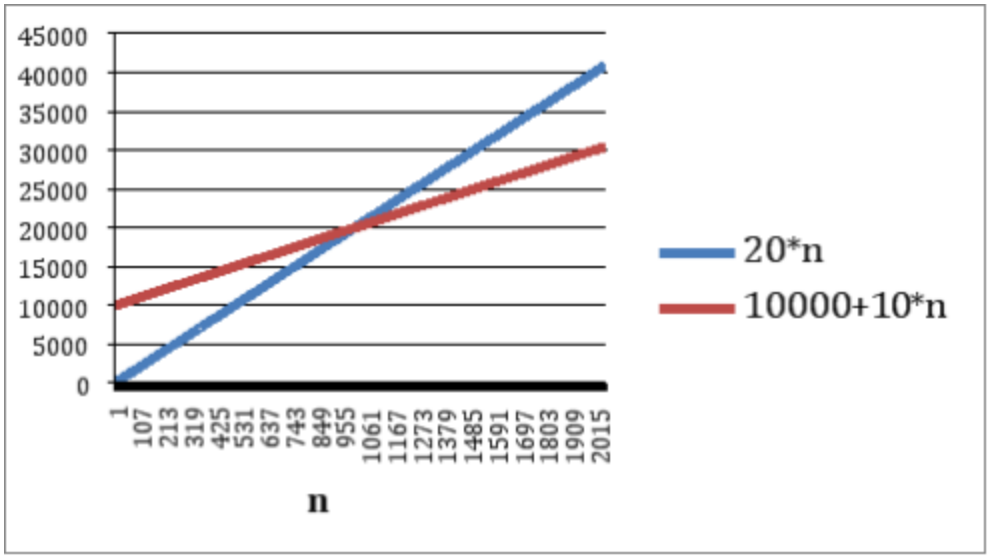


Figure 5. Growth of two functions for a given input n showing where the algorithm with faster growth rate becomes more costly than a function with slower growth but higher initial cost.

Big-oh notation provides a theoretical upper bound on an algorithm's growth rate. This theoretical upper bound is

also referred to as the complexity of the algorithm.

The complexity can be calculated from the cost by applying the following rules: • If *T(n)* includes multiple terms, keep the term with the largest growth rate, and discard the others.
• Any constants in *T(n)* can be omitted.

Example 4: Calculate the complexity of $T(n) = 5n^3 + 3n^2 + n + 5$.

• The term $5n^3$ has the largest growth rate and will dominate the other terms as n grows sufficiently large. The $3n^2$ and n terms can be discarded.

• $5n^3$ has a constant of 5 that can be omitted, leaving n^3.

• $T(n) = 5n^3 + 3n^2 + n + 5$ is in $O(n^3)$.

Example 5: Calculate the complexity of the findItem() algorithm (Algorithm 1.1) if its cost is T(n) = 3n + 3.

In the *findItem()* algorithm, n is the array size.
• The dominant term is $3n$; remove the constant 3.
• Drop the constant 3.
• $T(n)$ is in $O(n)$.

2 Computer memory

The bit is the smallest unit of information stored in a computer. Each bit can be in one of two possible states: 0 or 1, representing off or on or true or false. Individual bits are grouped together into groups of 8 to create bytes, and the byte is how numbers are actually stored. (Everything is stored as a number, even strings) The value of the byte is determined from the state of the bits.

Each position in the byte represents a value. Figure 1 shows an example of one byte where the left-most position has a decimal value of 128 and the right-most position has a decimal value of 1. The bottom row in the figure shows the power of 2 equivalent for the decimal value. If the bit at a position is set to 1, then that decimal value is included in calculating the byte value.

128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Figure 1. Example byte where the left-most bit position represents a value of 128 and the left-most bit position represents a value of 1.

An example of how to calculate the byte value from the bit pattern is shown in Figure 2.

128	64	32	16	8	4	2	1
0	0	0	1	1	0	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Figure 2. In this byte, the positions for 2, 8, and 16 are set, which makes the value of the byte 26.

The positions for the 2, 8, and 16 have a 1, which makes the value for this byte:

2 + 8 + 16 = 26.

The value can also be calculated from the powers of 2:

$2^1 + 2^3 + 2^4 = 26$

2.1 Binary and hexadecimal representation

The bit string of 0s and 1s, such as 00011010, is a binary digit, and the conversion to 26 is a binary to decimal conversion. But, addresses and values are generally reported as hexadecimal, which is base 16, instead of base 10. To convert from decimal to hexadecimal, the hexadecimal digits 0 – F represent the decimal values 0 – 15.

Digits 0 – 9 in decimal and hexadecimal are the same. However, digits 10 – 15 are A – F in hex.

Hex = Decimal

A = 10
B = 11
C = 12
D = 13
E = 14
F = 15

2.2 Maximum value of a byte

The maximum value of one byte is determined by setting all bit positions to 1, as shown in Figure 3. The sum of all bit positions in the byte is:

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255.

128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Figure 3. The maximum value for one byte is calculated by summing the values for all bit positions in the byte.

The same approach is used to calculate the maximum value of the first four bits in the byte, as shown in Figure 4. The sum of the first four bits is:

8 + 4 + 2 + 1 = 15.

128	64	32	16	8	4	2	1
0	0	0	0	1	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Figure 4. The maximum value of the first four bits in the byte is the sum of the first four bit positions.

2.3 Converting to hexadecimal

It’s not a coincidence that 15 is also the maximum hex value of F, since F = 15. To represent a byte’s value in hex, split the byte into two groups of four bits, called nibbles, and calculate the hex value for each nibble. Figure 5 shows the multiple representations of a byte. The first row in the image is the decimal value. The second row is the decimal value for a byte divided into two nibbles. The third row shows whether the bit for that position is set, and the fourth row is the power of two equivalent for the decimal value. The fifth row in the image is the hex value for the nibble.

128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
F				F			

Figure 5. The maximum byte value of 255 in decimal is the same as FF in hex.

2.4 Multiple bytes of information

More than one byte is needed to represent numbers larger than 255. For example, a number such as 1000 can be stored in two bytes, where there are 16 bits of information. Each bit in those two bytes represents a power of 2. In the second byte, the powers of 2 are between 2^8 and 2^{15}. An example of a two-byte value is shown in Figure 6.

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Figure 6. In two bytes of data, the maximum value for a bit position is 32,768. In this example, all bit positions are set to 1.

The same process for determining the value of one byte is used to determine the value of two bytes. Sum up the values of the bit positions set to 1. In Figure 7, there is an example of this calculation for two bytes. The value of those two bytes is calculated as:

32768 + 8192 + 2048 + 1024 + 128 + 32 + 8 + 2 = 44202

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
1	0	1	0	1	1	0	0	1	0	1	0	1	0	1	0
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Figure 7. An example of a two-byte data type. Sum up the values for all bit positions with a value of 1 to get the value stored in the two bytes.

2.4.1 Representing multiple bytes in hexadecimal

With multiple bytes of information, it is much easier to represent the value in hex than it is to calculate the decimal value. The computer only sees 0 and 1, so the representation here is just for the benefit of human readability. Divide the bytes into groups of four bits and calculate the hex value for each group of four bits.

Example 1: Calculate the decimal and hexadecimal values for the two-byte data type shown in Figure 8.

The decimal value is calculated by summing up the values for each of the bit positions set to 1, which is:

32768 + 8192 + 2048 + 1024 + 128 + 32 + 8 + 2 = 44202.

The hexadecimal value is calculated by dividing the two bytes into four groups of four bits, where each of the four bits has a value of 8, 4, 2, or 1. Sum up the total of each of the four bits and assign a value between 0-F to the group of bits. The hexadecimal value is ACAA:

8 + 2 = A
8 + 4 = C
8 + 2 = A
8 + 2 = A

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1
1	0	1	0	1	1	0	0	1	0	1	0	1	0	1	0
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}
A				C				A				A			

Figure 8. The hexadecimal value is calculated from the bit positions for two bytes of information.

2.5 Maximum value of n bytes

The maximum value of 2 bytes is the sum of all positions, 65535. This value can also be determined for n bytes of data by subtracting 1 from the next power of two. The maximum bit value is 2^{15} in two bytes. Therefore, the next bit value, if there were one, would be 2^{16}. The maximum value of two bytes is $2^{16} – 1$. In one byte of data, the maximum value is 255, which is $2^{8} – 1$. The leftmost position in one byte represents 2^{7}. The first position (right-most) in the second byte is 2^{8}. The general formula for n bytes is $2^{(n*8)} - 1$.

2.6 Variables and types

Variables are storage locations in memory. When we declare a variable, such as
int x;
a label called *x* is affiliated with a location in memory, which stores a value. The programmer who declares *x* doesn't need to know where in memory the data is stored; any reference to the data is handled by referencing *x*.

Each location in memory has an address where one byte of information can be stored. The amount of memory assigned to a variable depends on the type of that variable, which also determines the range of values that can be associated with the variable. Some common data types and their sizes are:
int: 4 bytes char: 1 byte float: 4 bytes long: 8 bytes double: 8 bytes
The bytes for a particular variable of a type listed above are stored in contiguous one-byte locations. For example, the value 0xACAA from Figure 8 would be stored in two contiguous bytes of memory. Assume the first byte is stored in a fictional memory location 0xFF01. Because the value is two bytes, it also occupies 0xFF02. Figure 9 shows a visual representation of memory addresses and how a value could be stored. The 0x in front of the address and value is to designate that it is a hexadecimal value.

Address	Value
0xFF06	
0xFF05	
0xFF04	
0xFF03	
0xFF02	AC
0xFF01	AA
0xFF00	

Figure 9. Examples of memory addresses are in the left column and the data stored at that address is in the right column. The data is stored in

two contiguous bytes of memory.

When a variable is associated with a location in memory, referencing that variable pulls all bytes associated with it. For example, consider the memory layout and variable assignments shown in Figure 10. The variable *X* is assigned to location 0xFF01 and uses four bytes of memory, and the variable *Y* is assigned to location 0xFFFA and also uses four bytes of memory. When either of these variables is referred to in code, all bytes associated with the variable are read and the value is reconstructed from those bytes. For the *X* variable, the value is obtained by converting the binary sequence 1 1 1 1 1 1 1 1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1. For the *Y* variable, the value is obtained by converting the binary sequence 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1.

Address	Value	Variable
0xFFFF		
0xFFFE		
0xFFFD	0x03	Y
0xFFFC	0x02	
0xFFFB	0xFF	
0xFFFA	0x01	
.		
.		
.		
0xFF04	0xFF	X
0xFF03	0x0A	
0xFF02	0x01	
0xFF01	0x05	
0xFF00		

Figure 10. Variables are associated with a memory address and include all bytes needed to store the correct value for that variable. The variable X includes memory addresses 0xFF01 through 0xFF04.

2.7 Pointers

In C++, there are variables that store the memory address of other variables. These variables are called pointers.

A regular integer variable called *X* is declared as:
int X;
A pointer variable called *ptrX* that points to the address of *X* is declared as:
int *ptrX = &X;
The value of the variable that *ptrX* points to is retrieved using **ptrX*:
cout<<*ptrX<<endl;
Important Notation: • In a variable declaration, the * signifies that the variable being declared is a pointer variable.
• The * is also a dereference operator to retrieve the value of the variable to which the pointer points.
• The & notation means “address of.”

The relationship between **ptrX* and *X* in memory is shown in Figure 11. The variable *X* is stored at 0xFF01 and uses four bytes. The variable *ptrX* is stored at 0xFFFA and holds the address of *X*.

Address	Value	Variable
0xFFFF		
0xFFFE		
0xFFFD		ptrX
0xFFFC		
0xFFFB	0xFF	
0xFFFA	0x01	
.		
.		
.		
0xFF04	0xFF	X
0xFF03	0x0A	
0xFF02	0x01	
0xFF01	0x05	
0xFF00		

Figure 11. The variable X is stored at 0xFF01 and uses four bytes. The variable ptrX is stored at 0xFFFA and holds the address of X.

When declaring a pointer variable, the variable must have a type because it specifies how much memory is associated with the address stored in the pointer.

In the example,
int x; int *ptrX = &x;
both variables are integers. Creating a pointer variable of one data type that points to a variable with a different data type would generate an error. For example,
int x; double *ptrX = &x;
would generate an error because different amount of memory are associated with each variable. The value of a variable is calculated from all of the bytes associated with that variable. If the pointer variable is reading four bytes, and the variable it points to is storing eight bytes, then there is an inconsistency in the amount of memory for each variable and potentially the value of the variable.

2.8 Declaring pointer variables

Example 1: Create a variable called intX with a value of 5. Print its value and its address.

int intX = 5; To retrieve the address, use the address of operator &.

cout<<"Value of intX: "<<intX<<endl; cout<<"Address of intX: "<<&intX<<endl; These **cout** statements will display: *Value of intX: 5*

Address of intX: 0x<address> where *<address>* is a large hexadecimal number that is the address of *intX*.

Example 2: Create a variable called intB that has a value of 50 and a pointer called ptrB that points to the address of intB.

int intB = 50; int *ptrB = &intB; **Example 2.1: Print the address of intB and the address that to which ptrB points.**

cout<<"Address of intB: "<<&intB<<endl; cout<<"Address pointed to by ptrB: "<<ptrB<<endl; These **cout** statements will both display the same hexadecimal address.

Example 2.2: Print the value of intB and the value of the variable to which ptrB points.

To retrieve the value of the variable that *ptrB* points to, include the dereference operator *, which can be read as "the value of the variable whose address is stored in" *ptrB*.

cout<<"Value of variable pointed to by ptrB: "<<*ptrB<<endl; cout<<"Value of intB: "<<intB<<endl; The output of these **cout** statements is: *Value of variable pointed to by ptrB: 50*

Value of intB: 50

Example 2.3: Add 10 to intB and print the value of intB and *ptrB.

intB = intB + 10; cout<<"Now the value of *ptrB is "<<*ptrB<<endl; cout<<"The value of intB is "<<intB<<endl; Changing the value of either *intB* or **ptrB* changes the value for both variables since they are both

accessing the same memory location. The output of these **cout** statements is: *Now the value of *ptrB is 60*
The value of intB is 60

3 Arrays

An array is a data structure used to store a collection of data, where each element in the collection is the same type and size. All array elements are stored in a contiguous block of computer memory. Arrays are often used to store data collected over time, and the index in the array establishes the order in the data.

An array data type is available in most programming languages and is also the underlying data structure for interfaces, such as Vectors, in higher-level languages. There are also common operations involving arrays in complicated algorithms for searching and sorting on large data sets. Having a solid grasp of arrays, both their strengths and limitations, will make it easier to understand these complicated algorithms.

Arrays are fast and simple to implement. Array elements are stored in contiguous memory, which makes them faster to access than if they were distributed in memory. Arrays are also easy to implement. Only one line of code is required to declare an array and individual elements can be accessed in one line of code using their index.

Arrays also have their limitations; primarily, they have a fixed size. At the time that an array is declared, a fixed amount of memory needs to be associated with the variable, and for applications where the array fills up or the size is not known at runtime, this limitation is often addressed with expensive array doubling algorithms.

3.1 Creating an array

Declaring an array is syntactically similar to declaring a scalar variable. The only difference is that the size of the array needs to be specified. For example, declare an array of 10 integers as follows:

```
int x[10];
```

The array variable *x* can be visualized as a sequence of 10 boxes, as shown in Figure 1, where each box can store an array element. Individual elements in the array are accessed using the index of the element. For example, *x[0]* accesses the first element in the array and *x[9]* accesses the last element in the array. The statement

```
int y = x[5];
```

creates a variable *y* and sets it's value to the value of *x[5]*. The statement

```
x[5] = y;
```

sets the value of *x[5]* to the value of variable *y*.

Figure 1. Visualization of an array of 10 integer elements. Each element can be accessed using the index of the element.

Arrays can be of any type, including the built-in types, such as **integers** and **doubles**, as well as user-created types using **classes** and **structs**. Arrays can also have one or more dimensions.

```
//an array of 10 doubles double x[10];
//an array of 5 integers int x[5];
//an array of 20 strings string x[20];
//a 2D array of integers that is 5 rows and 10 columns int x[5][10];
```

//an array of 10 WeatherData, where WeatherData defined by a struct struct WeatherData{
    double temperature double humidity double windVelocity }; WeatherData wd[10];

Example 1: Set the value of the elements in array x to the index of the element.

Updating *x* to store its index as its value is most-efficiently accomplished by looping through *x* and setting the loop iterator as the value for each array element.

for i = 0 to x.end x[i] = i
This algorithm writes the value of *i* to *x[i]* and generates the array shown in Figure 2.

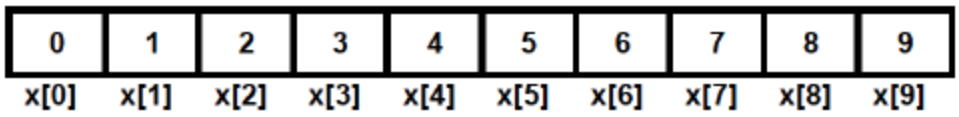


Figure 2. Array generated by algorithm in Example 1. The index i is written to x[i].

3.2 Array Operations

3.2.1 Searching an array

To search for a specified value in an array, iterate through the array using the array index until the value is found. The *searchArray()* algorithm in Algorithm 3.1 takes the array and the value to search for as arguments and returns the index in the array where the value is found.

Algorithm 3.1. searchArray(A, v) Returns the index of the value v in the array A.

Pre-condition *A* is an array.
v is a valid search value that is the same type as the elements in *A.*

Post-condition Returns the index *x* where *A[x] = v.*

Algorithm searchArray(A, v) 1. found = false 2. index = -1
3. x = 0
4. while(!found and x <= A.end) 5. if A[x] == v 6. found = true 7. index = x 8. else 9. x++
10. return index

Complexity of search operation 1D array: O(n) 2D array: $O(n^2)$
where *n* is the size of the array.

3.2.2 Adding an element to an array

To add an element to an array, the elements that are currently in the array need to be shifted out of the way to make room for the new element. The *insertArrayElement()* algorithm shown in Algorithm 3.2 takes the array, the value to insert and the position where it should be inserted, and the number of active elements in the array as arguments.

The algorithm updates the array to include the new element.

Algorithm 3.2. insertArrayElement(A, v, index, numElements) Adds the element *v* to the array *A* at the *index* position.

Pre-condition *A* is an array.
v is the same type as the elements in *A.*
index is a valid integer less than the size of *A*, *0 <= index <= A.end.*
numElements is the number of occupied indices in *A*.

Post-condition Array *A* is updated such that *A[index] = v.*
numElements is increased by 1.

Algorithm insertArrayElement(A, v, index, numElements) 1. for x = numElements-1 to index 2. A[x+1] = A[x]
3. A[index] = v 4. numElements = numElements + 1

Using the number of elements currently in the array instead of the size of the array reduces the number of computations when the array isn't full because unpopulated locations are ignored. The *insertArrayElement()* algorithm assumes there is space available in the array. If the array is full, the element in the last position in the array will be overwritten.

Example 2: Add a 5 to the array in Figure 3 at x[4].

For this example, the number of elements is 7 and the index is 4. The **for** loop on Line 1 of the *insertArrayElement()* algorithm will work backwards through the array, starting at the last populated element, *x[6],* and shift each element up to and including *x[4]*. The elements *x[6], x[5],* and *x[4]* will be shifted by 1 position to

the right to open a space at *x[4]*. The state of the array after the **for** loop is shown in Figure 4. Notice that the value of *x[4]* hasn't changed in the **for** loop. On Line 3 of the *insertArrayElement()* algorithm, *x[4]* will be overwritten with the new value to generate the final array, which is shown in Figure 5.

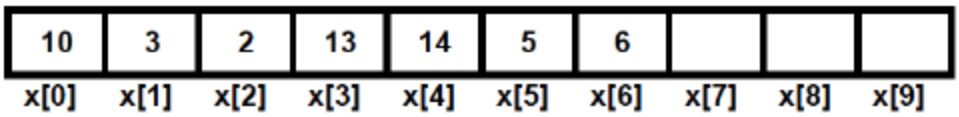


Figure 3. Initial state of array x for Example 2. A 5 needs to be added to the array at x[4] and all elements after x[4] need to be shifted out of the way to open a space.

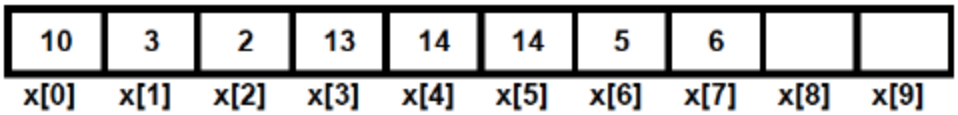


Figure 4. State of array x at the end of the for loop. The same value is stored at x[4] and x[5].

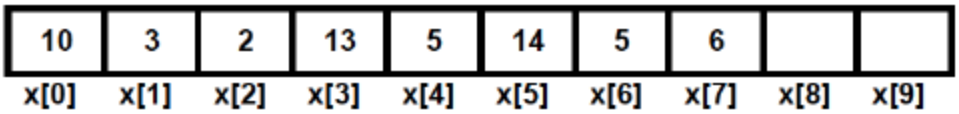


Figure 5. Final state of array x after 5 inserted at x[4]. The number of elements in the array has increased by 1.

Complexity of adding an element to an array O(n), where *n* is the size of the array.

3.2.3 Copying an array

The *copyArray()* algorithm shown here (Algorithm 3.3) writes the contents of one array into another array, and preserves the order of the data. The algorithm assumes that the array being copied to is at least as big as the array being copied from.

Algorithm 3.3. copyArray(A, B) Copies the elements of array *A* to the corresponding indices in array *B*.

Pre-conditions *A* and *B* are arrays of the same type. The size of *B* is at least as big as the size of *A.*

Post-conditions The elements of *B* in the positions *B[0 ... A.end]* are the same as the elements of *A* in the corresponding positions.

Algorithm copyArray(A, B) 1. for x = 0 to A.end 2. B[x] = A[x]

Array copy complexity 1D array: O(n) 2D array: $O(n^2)$ where *n* is the size of the array.

3.2.4 Deleting an element from an array

An element can be removed from an array by shifting the elements in the array to overwrite the element to delete. The *deleteArrayElement()* algorithm in Algorithm 3.4 takes the array, index of the element to delete, and the number of elements in the array as arguments and shifts the elements to the left to overwrite the index position. The shift left is the opposite direction of the shift to insert an element into an array.

Algorithm 3.4. deleteArrayElement(A, index, numElements) Deletes the element in array *A* at the *index* position and decreases the number of elements in the array by 1.

Pre-conditions *A* is an array.
index is a valid integer, *0 <= index <= A.end.*
numElements is a valid integer*.*

Post-conditions The value at *A[index]* is overwritten. *numElements* is decreased by 1.

Algorithm deleteArrayElement(A, index, numElements) 1. for x = index to numElements - 2
2. A[x] = A[x+1]
3. numElements = numElements - 1

Example 3: Delete the 14 in the array in Figure 6 at x[4].

In this example, the number of elements is 7 and the index is 4. The **for** loop in the *deleteArrayElement()* algorithm will shift *x[5]* and *x[6]* to *x[4]* and *x[5]*, respectively. The state of the array at the end of the **for** loop is shown in Figure 7. On Line 3 of the algorithm, the number of elements in the array is decremented by 1 to reflect that an element was removed. The value of *x[6]* remains unchanged, but won't be accessed as long as *numElements* contains the correct number of "active" array elements.

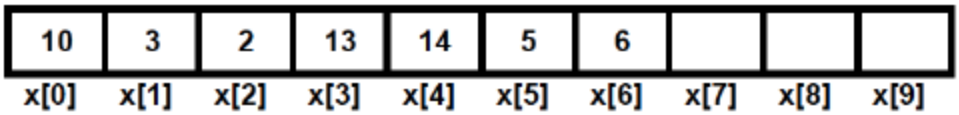


Figure 6. Array x for Example 3. The 14 at x[4] needs to be removed from the array using Algorithm 3.4.

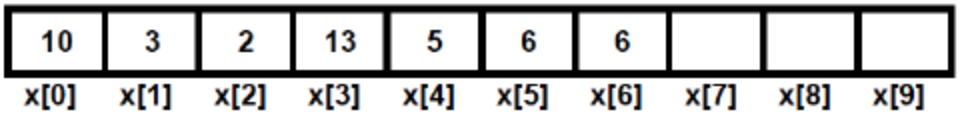


Figure 7. State of array x after x[4] is removed. There is still at 6 at x[6] but it won't be accessed because numElements will restrict the for loop to evaluate x[0 … 5] only.

Complexity of deleting an array element O(n)

3.3 Arrays in memory

When an array is declared, the name of the array is a pointer to the first element in the array in memory. For example, given an array

int x[10];

the command

cout<<x<<endl;

will display the address of x[0]. Referencing (x+1) returns the address of x[1], (x+2) returns the address of x[2], and so on. The data type of the array elements determines how much memory is associated with each element, and how much the address changes with each +1. In an array of **integers**, an array element occupies four bytes in memory. Therefore, adding +1 increases the address by four bytes. In an array of **doubles**, each element occupies 8 bytes, and incrementing by 1 increases the address by 8 bytes. An example for an array of 10 integers is shown in Figure 8. A theoretical address (generated from a real address by removing the first eight hex digits) is shown for each element.

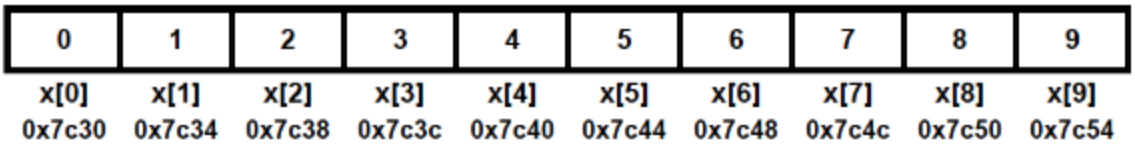


Figure 8. Example addresses for each element in an array of 10 integers. The address of each element increases by four over the address of the previous element in the array.

3.4 Dynamic memory allocation

When memory is statically allocated for a variable, that memory is reserved until the variable goes out of scope, e.g. the function where the variable was declared exits. In the case of arrays, if more memory is allocated than what is needed, i.e. the array is larger than the data it needs to store, then memory is wasted. If not enough memory is allocated, the array will fill up and not be able to store all of the data.

If more memory is needed during the runtime of the program than what is allocated when the program is compiled, that memory needs to be allocated dynamically. When memory is requested dynamically, an address is returned, and that the memory needs to be accessed with a pointer.

3.5 Stack and heap memory

There are two areas of memory where variables are stored, they are the call stack and the heap.

3.5.1 The Stack

Local variables are stored on the stack. For example, an array declared statically inside a function would be placed on the stack. The stack has limited space available, and is carefully managed by the processor to free memory when it is no longer needed.

3.5.2 The Heap

The heap is a pool of free memory that is much larger than the stack and used for storing variables that are created dynamically during runtime. Variables created using pointers are created on the heap.

Unlike the stack where memory is managed by the processor, the developer has to handle memory management of the heap. Variables allocated on the heap have to also be explicitly de-allocated.

Features of the heap • Allocated memory stays allocated until it is specifically de-allocated (Beware of memory leaks).
• Dynamically allocated memory must be accessed through a pointer.
• Allocate large arrays, structures, and objects on the heap.

Allocating variables dynamically To dynamically allocate a variable, use the **new** keyword in C++. Free the memory using the **delete** keyword.

//a dynamically allocated array of 10 doubles double *x = new double[10];

```
//a dynamically allocated array of 5 integers int *x = new int[5];
//a dynamically allocated array of 20 strings string *x = new string[20];
/*A 2D array of integers that is 5 rows and 10 columns can be created using an array of pointers, where each pointer points to the first element in an array.
*/
int rows = 5; int columns = 10; int **x = new int*[rows];
for(int y = 0; y < rows; y++) x[y] = new int[columns];
//a dynamically allocated array of 10 WeatherData
struct WeatherData{
    double temperature; double humidity; double windVelocity; }; WeatherData *wd = new WeatherData[10];
//set the values for the second element in the array
wd[1].temperature = 56; wd[1].humidity = 30;
wd[1].windVelocity = 10;
```

Variables created dynamically need to be deleted when they are no longer needed. The memory allocated to local variables created dynamically will not be freed when the variable goes out of scope. To free the memory use the **delete** keyword.

```
//Free a 1D array delete[ ] x; delete[ ] wd;
//Free a 2D array for(int y = 0; y < rows; y++) delete[ ] x[y]; delete[ ] x;
```

3.5.3 Array doubling

Array doubling is an algorithm used to increase the size of an array when an array is full, but additional space is needed to store more data. The algorithm, shown in Algorithm 3.5, generates a new array dynamically that is twice the size of the current array, then copies the values from the current array into the first half of the new array.

Algorithm 3.5. doubleArray(A) Creates a new array that is twice the length as the array *A*, copies the elements of *A* into the new array, and returns the new array.

Pre-conditions *A* is an array.

Post-conditions Returns an array that is the same type and twice the length as *A*.

Algorithm doubleArray(A) 1. B.length = A.length * 2
2. for x = 0 to A.end 3. B[x] = A[x]
4. return B

Note: By declaring a new array and returning that array from the algorithm, there is a memory leak in the code if the memory assigned to A is not freed after *doubleArray()* is called.

Array doubling complexity O(n)

3.6 Array Questions

1. Write a C++ function to double the size of an array an arbitrary number of times, and populate the second half of the array with values that are 2x the values in the first half of the array.
The function takes three arguments - the initial 1D array, its size, and the number of times to double it. The function should return the new array. The expected function prototype is: int *ArrayDynamicAllocation(int array[], int size, int number); For the following inputs: int arr[2] = {0, 1}; int arraySize = 2; int numberOfDoublings = 3; the expected output is an array that contains: < 0,1,0,2,0,2,0,4,0,2,0,4,0,4,0,8 >.
2. Write a C++ function that finds all instances of a specified value in an array and removes them. Each time an element is removed, the array should be shifted to fill the gap.
3. Write a C++ function to copy all elements of an array to a new array, except for a specified value. For example, copy all elements except the 5 from an array A to array B.
4. Write a C++ function to remove an element from an array and shift the array to fill the empty spot. The function takes three arguments - the input array, size of array, and the value to be removed from the array. The expected function prototype is: void arrayShift(int arr[],int length,int value); For the following inputs: int inputArray[5] = {10, 20, 30, 40, 50}; int length = 5; int value = 30; the expected result is an array that contains: < 10, 20, 40, 50 >.
5. Write a C++ function to find the second largest element in an array. The function takes two arguments - the input array and the size of the array, and returns the second largest element. The prototype for the expected function is: int secondLargest(int arr[],int size); For the following inputs: int inputArray[4] = {1, 2, 3, 4}; int length = 4; the expected return value from the function is 3.

6. Write a C++ function that finds the most common repeating element in an array. The function takes two arguments - the input array and the size of the array, and returns the most common element. The prototype for the expected function is: int commonRepeatingElement(int arr[],int length); For the following inputs: int inputArray[13] = {5, 5, 5, 3, 3, 1, 1, 3, 3, 3, 1, 3, 3}; int length = 13; the expected return value from the function is 3.

4 Sorting algorithms

A sorting algorithm is an algorithm that puts the elements in a collection in a specified order. These algorithms use different strategies, which affects the runtime and memory requirements of the algorithm.

4.1 Bubble sort

One of the simplest, but unfortunately slowest, sorting algorithms to implement is bubble sort, shown in Algorithm 4.1. In the bubble sort algorithm, individual elements "bubble" to their correct location through a series of individual swaps for elements out of order.

Algorithm 4.1. bubbleSort(A) The input array A is sorted in ascending order.

Pre-conditions A is an array.

Post-conditions A is in ascending order.

Algorithm bubbleSort(A){
1. for i = 0 to A.end - 1
2. for j = 0 to A.end - i - 1
3. if (A[j] > A[j+1]) 4. swap = A[j]
5. A[j] = A[j+1]
6. A[j+1] = swap

Starting from the beginning of the array, the first two elements are compared and swapped if they are out of order. Next, the second and third elements are compared and swapped if they are out of order. These comparisons and swaps go through an entire pass of the array, and then restart again from the first element in the array. Bubble sort is an in-place algorithm, which means that elements are moved around within the array without significant additional memory requirements.

Example 1: Given the array shown in Figure 1, show the state of the array after each iteration of a for loop in the bubbleSort() algorithm.

Figure 1. Initial state of array A for the bubble sort algorithm (Algorithm 4.1).

First iteration of outer for loop, *i = 0*
First iteration of the inner for loop, *j = 0*

• **Line 3**: compare the values of *A[0]* and *A[1]*, which are 10 and 40, respectively. The conditional is false, so Lines 4 - 6 are skipped and there are no changes to the array.

Second iteration of the inner for loop, *j = 1*

• **Line 3**: compare the values of *A[1]* and *A[2]*, which are 40 and 13. The conditional is true.
• **Lines 4 - 6**: the values of *A[1]* and *A[2]* are swapped, moving 13 into *A[1]* and 40 into *A[2]*. The new state of the array *A* is shown in Figure 2.

Figure 2. State of the array after A[1] and A[2] swapped in the first iteration of the outer for loop.

Third iteration of the inner for loop, *j = 2*

• **Line 3**: compare the values of *A[2]* and *A[3]*, which are 40 and 20. The conditional is true, so swap that values of *A[2]* and *A[3]*. This operation moves the 20 to *A[2]* and moves the 40 to *A[3]*. The new state of the array is shown in Figure 3.

Figure 3. State of the array after A[2] and A[3] are swapped in the first iteration of the outer for loop.

Fourth iteration of inner loop, *j = 3*

• **Line 3**: compare the values of *A[3]* and *A[4]*, which are 40 and 8. The conditional is true, so swap the values of *A[3]* and *A[4]*. The 40 moves to *A[4]* and the 8 moves to *A[3]*. This is the last iteration of the inner **for** loop, and the maximum value in the array should now be in the last array position. The new state of the array is shown in Figure 4.

Figure 4. State of the array after the last iteration of the inner for loop when i = 0. The maximum value in the array is now in the last array position.

Second iteration of the outer for loop, *i = 1*
First iteration of the inner for loop, *j = 0*

• **Line 3**: compare the values at *A[0]* and *A[1]*, which are 10 and 13. The conditional is false, so Lines 4 - 6 are skipped and there is no change to the array.

Second iteration of the inner for loop, *j = 1*

• **Line 3**: compare the values at *A[1]* and *A[2]*, which are 13 and 20. The conditional is false, so Lines 4 - 6 are skipped and there is no change to the array.

Third iteration of the inner for loop, *j = 2*

• **Line 3**: compare the values at *A[2]* and *A[3]*, which are 20 and 8. The conditional is true. Swap the values of *A[2]* and *A[3]*, which moves the 8 to *A[2]* and the 20 to *A[3]*. This is the last iteration of the inner for loop when *i = 1*. The new state of the array is shown in Figure 5.

Figure 5. State of the array after A[2] and A[3] are swapped. The second highest value in the array is now in the A[3] position.

Third iteration of the outer for loop, *i = 2*
First iteration of the inner for loop, *j = 0*
• **Line 3**: compare the values of *A[0]* and *A[1]*, which are 10 and 13. The condition is false, so Lines 4 - 6 are skipped and there is no change to the array.

Second iteration of the inner for loop, *j = 1*
• **Line 3**: compare the values of *A[1]* and *A[2]*, which are 13 and 8. The conditional is true. Swap the values of *A[1]* and *A[2]*, which moves 8 to *A[1]* and 13 to *A[2]*. The new state of the array is shown in Figure 6.

Figure 6. State of the array A after A[1] and A[2] are swapped. The elements A[2 … 4] are now in the correct positions.

Forth iteration of the outer for loop, *i = 3*
First iteration of the inner for loop, *j = 0*

• **Line 3**: compare the values of *A[0]* and *A[1]*, which are 10 and 8. The conditional is true. Swap the values of *A[0]* and *A[1]*, which moves 8 to *A[0]* and 10 to *A[1]*. The array is now in its final sorted state, shown in Figure 7.

Figure 7. Final state of the array after applying the bubble sort algorithm. All elements in the array are now sorted in ascending order.

Complexity of Bubble Sort: Big-Oh is $O(n^2)$
The initial array configuration that generates the greatest number of swaps in bubble sort is reverse sorted order. In this configuration, every element will be swapped every time it is evaluated.

4.2 Insertion Sort

In the insertion sort algorithm, shown in Algorithm 4.2, elements are moved to their correct location in the array one at a time, similar to how a person might sort a hand of cards. The element to be sorted is removed from the array, its correct location is identified and array elements are shifted out of the way to make room for the element, and the element is then added back to the array in the correct location.

Algorithm 4.2. insertionSort(A) The input array A is sorted in ascending order.

Pre-conditions A is an array.

Post-conditions The array A is in ascending order.

Algorithm insertionSort(A) 1. for i = 1 to A.end 2. index = A[i]
3. j = i 4. while(j > 0 and A[j - 1] > index) 5. A[j] = A[j - 1]
6. j = j - 1
7. A[j] = index

Example 2: For the array shown in Figure 8, show the state of the array after each loop iteration in the insertionSort() algorithm.

Figure 8. Initial state of the array A for Example 2 using the insertion sort algorithm.

First iteration of the for loop, *i = 1*

• **Line 2**: store *A[1]* in *index*, which sets *index = 4*.

• **Line 3**: set *j = i = 1*.
• **Line 4**: the **while** loop conditional checks the value of *j* and compares *A[0]* to *index*. Both conditions are true.
• **Line 5**: the value of *A[0]* is written to *A[1]*, which overwrites the 4 with the 5.
• **Line 6**: *j* is decremented and now *j = 0*.
• **Line 7**: *A[0]* is updated with the value of *index*, which is 4.

The new state of the array is shown in Figure 9.

Figure 9. The state of the array A after the first two elements, A[0] and A[1], are swapped.

Second iteration of the for loop, *i = 2*
• **Line 2**: store *A[2]* in index, which sets *index = 7*.
• **Line 3**: set *j = i = 2*.
• **Line 4**: the **while** loop conditional checks the value of *j* and compares *A[1]* to *index*. *Index* is greater than *A[1]*, which fails the second condition.
• Lines 5 and 6 are skipped.
• **Line 7**: *A[2]* is set to 7, which is its current value and there are no changes to the array.

Third iteration of the for loop, *i = 3*
• **Line 2**: store *A[3]* in *index*, which sets *index = 2*.
• **Line 3**: set *j = i = 3*.
• **Line 4**: the **while** loop conditional checks the value of *j* and compares *A[2]* to 2. Both conditions evaluate to true.
• **Line 5**: *A[3]* is overwritten with *A[2]*. The state of the array after Line 5 is shown in Figure 10.

Figure 10. State of array A after Line 5 executes and A[2] overwrites A[3]. The values of A[2] and A[3] are now the same.

• **Line 6**: *j* is decremented and now *j = 2*.
• The algorithm moves to Line 4 to evaluate the **while** loop conditions. *A[1]* is compared to *index*, which is still 2, and j > 0. Both conditions are true.
• **Line 5**: *A[2]* is overwritten with *A[1]*. The state of the array after Line 5 is shown in Figure 11.

Figure 11. State of array A after Line 5 executes and A[1] overwrites A[2]. The values of A[1] and A[2] are now the same.

• **Line 6**: *j* is decremented and now *j = 1*.
• The algorithm moves to Line 4 to evaluate the **while** loop conditions. *A[0]* is compared to *index*, which is still 2, and j > 0. Both conditions are true.
• **Line 5**: *A[1]* is overwritten with *A[0]*, which writes a 4 to *A[1]*. The new state of the array after Line 5 is shown in Figure 12.

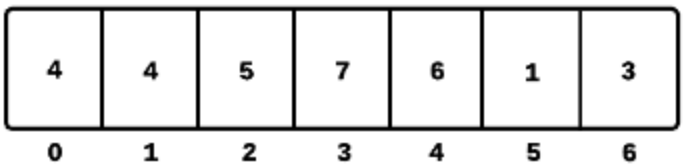


Figure 12. State of the array A after A[0] overwrites A[1]. The values of A[0] and A[1] are now the same.

• **Line 6**: *j* is decremented again and now *j = 0*.
• **Line 4**: the **while** loop conditions are evaluated. The first condition fails, since *j* is not greater than 0, and the **while** loop exits.
• **Line 7**: *index* is written to *A[0]*, which puts a 2 at that location. The new state of the array after Line 7 is shown in Figure 13.

Figure 13. State of the array A after the first four elements in the array are sorted.

Fourth iteration of the for loop, *i = 4*
• **Line 2**: store *A[4]* in *index*, which sets *index = 6*.
• **Line 4**: evaluate the **while** loop conditions. *A[3]* is compared to *index*, which is 6, and j > 0. Both conditions are true.
• **Line 5**: *A[3]* overwrites *A[4]*, which places the 7 at *A[4]*. The new state of the array after Line 5 is shown in Figure 14.

Figure 14. State of array A after A[3] overwrites A[4].

• The algorithm moves to Line 4 to evaluate the **while** loop conditions. The second condition fails because *A[2]* is less than *index*, which is 6. The **while** loop exits.
• **Line 7**: *index* overwrites *A[2]*, which puts a 6 at *A[2]*. The new state of the array after Line 7 is shown in Figure 15.

Figure 15. State of the array A after the first five elements are sorted using insertion sort.

Fifth iteration of the for loop, *i = 5*

• **Line 2**: store *A[5]* as *index*, which sets *index = 1*
• **Line 3**: set *j =i = 5*
• **Line 4**: compare *index* to *A[4]* in the **while** loop condition. Both conditions for the **while** loop are true.
• **Line 5**: *A[4]* overwrites *A[5]*, which puts a 7 at *A[5]* and overwrites the 1.
• **Line 6**: decrement the value of *j*, which sets *j = 4*.
• The algorithm returns to Line 4 and compares *A[3]* to *index* in the **while** loop conditions. Both conditions are true.
• **Line 5**: *A[3]* overwrites *A[4]*, which writes a 6 to *A[4]*.
• Repeat the process of decrementing *j* and comparing *index* to *A[j-1]*. Write *A[j-1]* to *A[j]* as long as *A[j-1]* is greater than *index*. In this example, the 5, 4, and 2 are all moved. After the **while** loop exits, the 1 is written to *A[0]*. The new state of the array is shown in Figure 16.

Figure 16. State of the array A after fifth iteration of for loop. The first six elements in the array are sorted. The only remaining element to sort is the 3 at A[6].

Sixth iteration of for loop, *i = 6*

The only element in the array left to sort is the 3 at *A[6]*.

• Store the 3 in *index*.
• Compare *index* to the values of *A[5] … A[0]* in the array.
• Shift array elements from *A[j-1]* to *A[j]* that are greater than *index*.
• Place *index* at *A[2]*, which writes a 3 to *A[2]*. The final state of the array, now completely sorted, is shown in Figure 17.

Figure 17. Final state of array A sorted using the insertion sort algorithm.

Complexity of Insertion Sort Big-Oh is $O(n^2)$

4.3 Quicksort

Quicksort uses an algorithmic approach called divide and conquer. The array to sort is divided into smaller and smaller sub-arrays that are sorted and then recombined. Dividing the array into smaller arrays reduces the number of iterations over the entire array that need to be performed, which speeds up the sorting process.

The sorting approach that quicksort uses is similar to the idea behind putting items into two piles, one pile with values greater than a specified value and one pile with values less than a specified value. Once those piles are created, each pile is then divided into two more piles using a middle value as the partitioning criteria. This process repeats with smaller and smaller piles until all items are in sorted order.

The difference the scenario just described and how quicksort works is that quicksort doesn't use any additional memory for creating separate piles. Array elements that are on the wrong side of a partitioning value are swapped with another out-of-place element.

There are three steps to the quicksort algorithm: **1. Divide:** Pick an element in the array, called the pivot, which will be used to partition the array. A value in the middle of the array is often selected as the pivot.
2. Partition: Divide the array into two sub-arrays: values less than or equal to the pivot are left of the pivot, and values greater than the pivot are right of the pivot. Divide the array by swapping values that are on the wrong side of the pivot.
3. Conquer: Recursively apply the Divide and Partition steps to sort the sub-arrays, and from these sorted sub-arrays, build a sorted array.

The quicksort algorithm is shown in Algorithm 4.3.

Algorithm 4.3. quickSort(A, left, right) Recursively sorts the array *A* in ascending order.

Pre-conditions *A* is an array.
left and *right* are valid integers that index array *A*.

Post-conditions The array A is sorted in ascending order
Algorithm quickSort(A, left, right) 1. i = left 2. j = right 3. pivot = A[(left + right) / 2]
4. while(i <= j) 5. while(A[i] < pivot) 6. i = i + 1
7. while(A[j] > pivot) 8. j = j - 1
9. if(i <= j) 10. tmp = A[i]
11. A[i] = A[j]
12. A[j] = tmp 13. i = i + 1
14. j = j - 1
15. if (left < j) 16. quickSort(A, left, j) 17. if (i < right) 18. quickSort(A, i, right)

Example 3: For the array shown in Figure 18, show the state of the array after each outer while loop iteration in the quickSort algorithm.

Figure 18. Initial state of array A for sorting using the quickSort algorithm.

• **Lines 1-3**: the values for *left*, *right*, and *pivot* are initialized. *Left* and *right* store the lowest and highest index in the array, which are 0 and 6 respectively. *Pivot* is the value at *A[3]*, which is 2.

First iteration of the outer while loop, *i = 0, j = 6*

• **Line 5**: the **while** loop condition that compares *A[0]* to *pivot* is false. There are no changes to *i*.

• **Line 7**: the **while** loop condition that compares *A[6]* to *pivot* is true. The value of *j* is decremented on Line 8, which sets *j = 5*. The **while** loop condition is evaluated again and is false. The values for *i*, *j*, and *pivot* on Line 9 of the algorithm are shown in Figure 19.

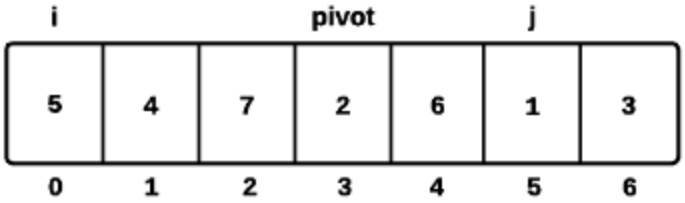


Figure 19. State of array A and the values for i, j, and pivot.

• **Lines 10-12**: the values at *A[0]* and *A[5]* are swapped, which puts 1 at *A[0]* and 5 at *A[5]*.

• **Lines 13-14**: the value of *i* is incremented and the value of *j* is decremented. The state of the array and the algorithm after Line 14 is shown in Figure 20.

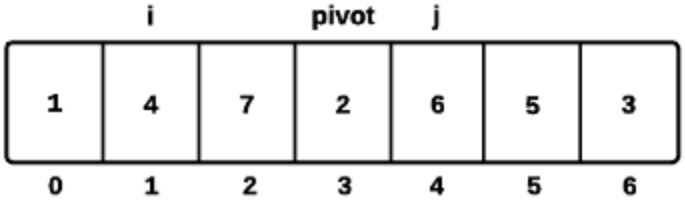


Figure 20. State of the array after A[0] and A[5] are swapped and i and j are updated.

Second iteration of outer while loop, *i = 1, j = 4*

• **Line 5**: the **while** loop condition that compares *A[1]* to *pivot* is false. There are no changes to *i*.

• **Line 7**: the **while** loop condition that compares *A[4]* to *pivot* is true. The value of *j* is decremented, and *j* and the

pivot index are now the same, as shown in Figure 21. The **while** loop condition is evaluated again and is false.

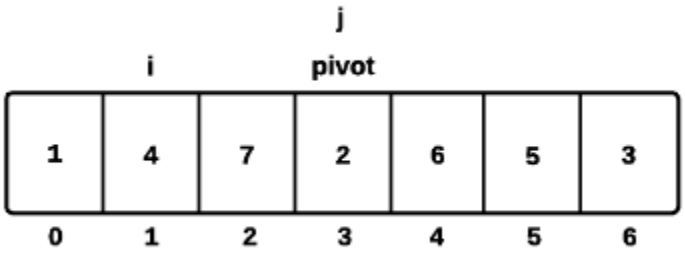


Figure 21. The value of j and the pivot are the same.

• **Lines 10-12**: the values of *A[1]* and *A[3]* are swapped, which puts 2 at *A[1]* and 4 at *A[3]*. The 2 is the pivot value, which means that this swap also moves the *pivot*.
• **Lines 13-14**: the values of *i* and *j* are updated. The new state of algorithm and the array is shown in Figure 22.

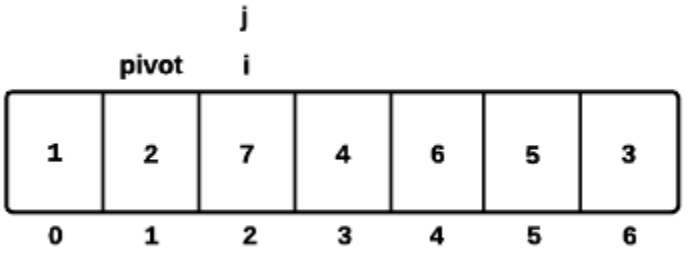


Figure 22. State of the array A after A[1] and A[3] swapped. This swap moves the pivot to A[1].

Third iteration of the outer while loop, *i = 2, j = 2*
• **Line 5**: the **while** loop condition that compares *A[2]* to *pivot* is false. The value of *i* is unchanged.
• **Line 7**: the **while** loop condition that compares *A[2]* to *pivot* is true. The value of *j* is decremented. On the next evaluation of the **while** loop condition, the evaluation is false. The current state of the algorithm on Line 9 is shown in Figure 23.

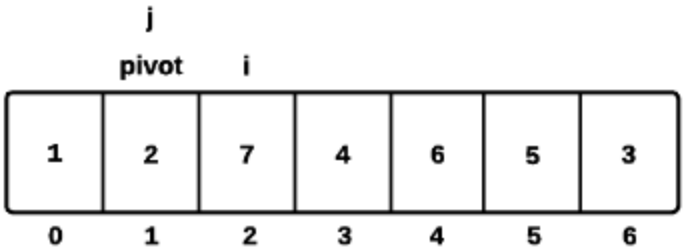


Figure 23. State of the array A after the variable j decremented past the variable i. This is the state of the array when it is passed in as a parameter to the recursive calls of quickSort().

• **Line 4**: the condition for entering the **while** loop again is false and the loop exits.

Outside the while loop • Line 15: if the condition is true it means that *j* has not been decremented all the way to the left edge of the array; there are array elements between the left-most element and *j*. The condition is true, since *left = 0* and *j = 1*.
• **Line 16**: call *quickSort( )* on array *A[0 … 1]*.
• **Line 18**: call *quickSort( )* on array *A[2 … 6]*.

Recursive call to quickSort( ) on A[0 … 1]
The two elements to sort on this call to *quickSort( ), A[0]* and *A[1]*, are already sorted. There won't be any changes to these array elements, but the steps that will execute on this call are as follows: • **Lines 1-3**: the algorithm is initialized, setting the values *i = 0, j = 1,* and *pivot = A[0]*, which is 1.
• **Line 5**: the **while** loop condition is false and *i* is unchanged.
• **Line 7**: the **while** loop condition is true, which decrements *j* to *j = 0*. The condition is evaluated a second time and is false.
• **Lines 10-12**: the values of *A[i]* and *A[j]* are swapped, which has no effect on the array since they are the same

value.
• **Line 13**: *i* is incremented, which sets *i = 1 = right*.
• **Line 14**: *j* is decremented, which sets *j = -1*.
• Both conditionals on Lines 15 and 17 will be false because *i = right* and *j < left*. There are no additional calls to *quickSort( )*.

Recursive call to quickSort() on A[2 ... 6]
• **Lines 1-3**: the algorithm is initialized, setting *i = 2, j = 6, pivot = A[4]*, which is 6. The initial state of the algorithm and the array *A* is shown in Figure 24.

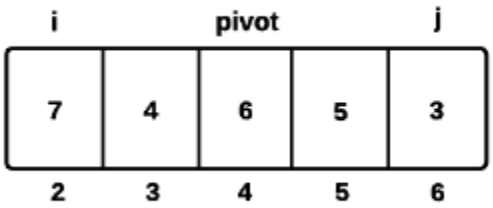


Figure 24. State of the array A and the algorithm parameters for the recursive call to quickSort() on the section of the array from the pivot value to the end of the array. The pivot is A[4], i is 2 and j is 6.

First iteration of outer while loop, *i = 2, j = 6*
• **Lines 10-12**: the values *A[2]* and *A[6]* are swapped, which puts 3 at *A[2]* and 7 at *A[6]*.
• **Lines 13-14**: *i* is incremented and *j* is decremented, which sets *i = 3* and *j = 5*. The new state of the array is shown in Figure 25.

Figure 25. State of the array after A[2] and A[6] swapped and i and j updated.

Second iteration of outer while loop, *i = 3, j = 5*

• **Line 5**: compare *A[3]* to the pivot in the **while** loop condition. The condition is true; increment *i* to 4. The **while** loop condition is false on the next evaluation, since *A[i]* is also the *pivot* value.

• **Line 7**: compare *A[5]* to the *pivot* in the **while** loop condition. The condition is false and *j* is unchanged.

• **Lines 10-12**: swap *A[4]* and *A[5]*, which puts 5 at *A[4]* and 6 at *A[5]*.

• **Lines 13-14**: increment *i* and decrement *j*. The state of the array at the end of this iteration of the **while** loop is shown in Figure 26.

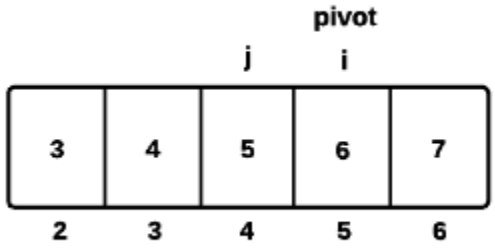


Figure 26. State of the array A at the end of the second iteration of the while loop, after A[4] and A[5] have been swapped.

Outside the while loop • Line 16: call *quickSort( )* on array *A[2 … 4]*.

• **Line 18**: call *quickSort( )* on array *A[5 … 6]*.

Recursive call to quickSort() on A[2 … 4]

The values of *A[2]*, *A[3]*, and *A[4]* are already sorted, as shown in Figure 27, and the array will not change on this call to the algorithm.

Figure 27. Array elements A[2 ... 4] to sort in a recursive call to the quick sort algorithm. These elements are already sorted, no additional calls to quickSort() are needed.

Recursive call to quickSort() on A[5 ... 6]

There is also a recursive call to *quickSort( )* on *A[5 ... 6],* which is also already sorted, as shown in Figure 28.

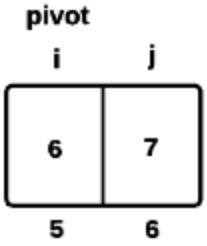


Figure 28. Array elements A[5 ... 6] to sort in a recursive call to quickSort(). These elements are already sorted, no additional calls to quickSort() are needed.

When the *quickSort( )* algorithm returns from these recursive calls, the array is sorted.

Complexity of Quicksort Big-Oh is $O(n^2)$

The choice of the pivot matters to the performance of this algorithm. The worst-case behavior is observed with a bad pivot selection. The average performance is O(n log n).

4.4 Heapsort

The heapsort algorithm uses a data structure, called a heap, to store data and then add it to an array in sorted order. Note: The heap data structure is not the same as the heap garbage collection in Java.

4.4.1 What is a heap?

The (binary) heap is a tree data structure, where the data is stored as a complete binary tree. Locations in the tree are filled at all levels, except the bottom level. There is also a defined relationship between the value of an individual node and the values of its parent and children. Figure 29 and Figure 30 show examples of heaps. Figure 29 shows an example of a min-heap, where the value of a node is less than the value of either of its children. The minimum value in the tree is the root of the tree. The other type of heap is a max-heap, shown in Figure 30. In a max-heap, the value of a node is greater than the value of either of its children.

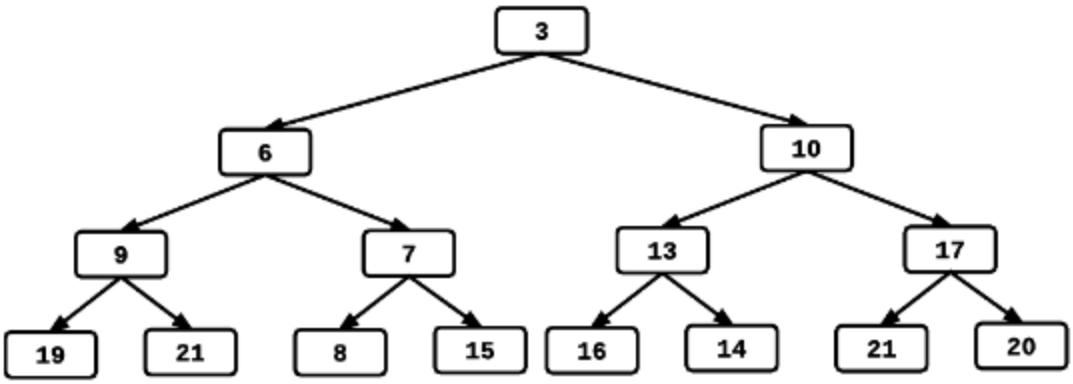


Figure 29. In a min-heap, the value of a node is less than the value of either of its children. The minimum value in the tree is the root node.

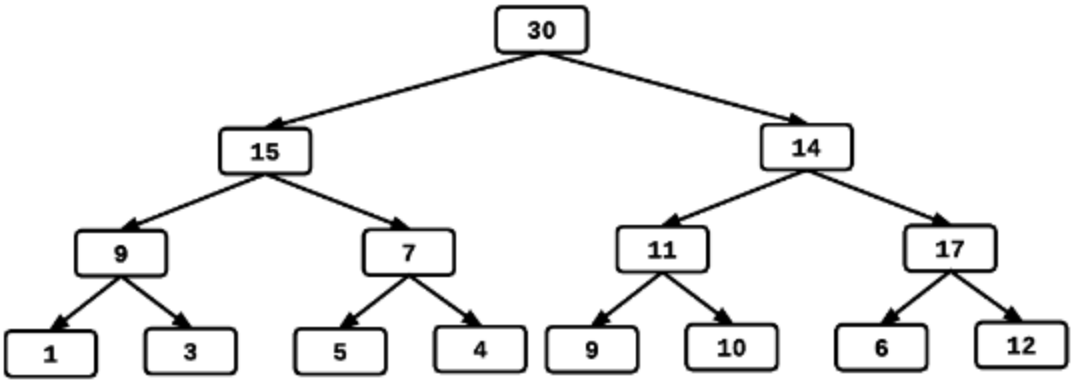


Figure 30. In a max-heap, the value of a node is greater than the value of either of its children. The root of the tree is the maximum value in the tree.

4.5 Heapsort

In the heapsort algorithm, the element at the root of the heap is removed and added to an array. When a min-heap is used, elements will be sorted in ascending order and with a max-heap, elements will be sorted in descending order. After the root is removed, the heap is restructured to move a new node into the root position. This new node will then be the next element removed from the heap and added to the sorted array. The process of selecting a new root, adding the root to the sorted array, and restructuring the heap continues until all nodes have been removed from the heap. The heapsort algorithm has a worst case performance that is O(n lg n).

There are two components to the heapsort algorithm:
Build the heap In the first part of heapsort, a heap is constructed from the data. The heap is stored as an array, where the indices in the array determine the position of the elements in the heap.

Remove elements from the heap and add to sorted array After the heap is constructed, the root of the heap is removed and added to the sorted array. The heap is then reorder to generate a new root. This process is repeated until the heap is empty.

4.5.1 Build the heap

The hierarchical structure of the heap can be captured in an array using an element's index in the array to assign it a position in a heap. For each element in the array, the element at index x has a left child in the heap stored at index $2x+1$ in the array and a right child in the heap stored at index $2x+2$ in the array. An example of an unsorted array and the corresponding binary tree is shown in Figure 31 and Figure 32.

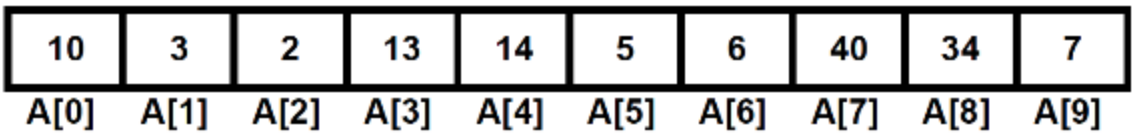


Figure 31. The position of the elements in this unsorted array determines the positions of the elements in the complete binary tree shown in Figure 32.

The element at the first position in the array, *A[0]*, is the root of the tree. The element at *A[1]* is the left child of the root, and the element at *A[2]* is the right child of the root. Continuing to the next level in the tree, the left and right children of *A[1]* in the array are *A[3]* and *A[4]*, respectively. The left and right children of *A[2]* in the array are *A[5]* and *A[6]*, respectively.

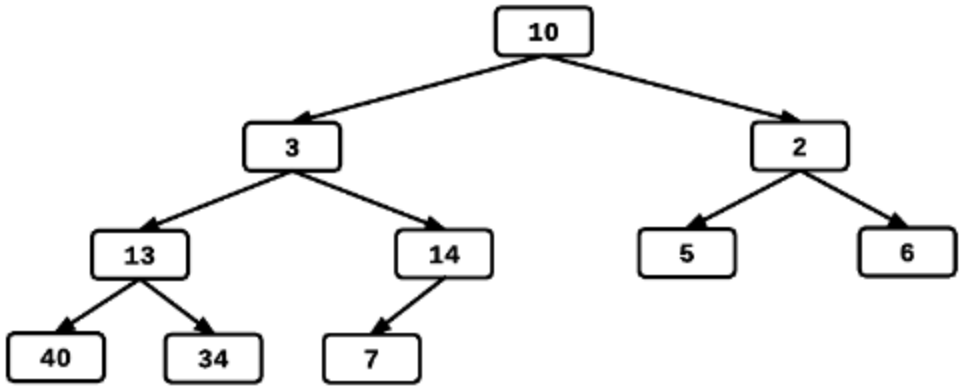


Figure 32. Complete binary tree constructed from the unsorted array in Figure 31. The buildHeap() algorithm will be used to move elements in the tree into their correct positions to create a max heap.

From the initial, unsorted condition, array elements are re-arranged to generate a heap. The *buildHeap()* algorithm, shown in Algorithm 4.4, calls *maxHeapify()* (Algorithm 4.5), to recursively move the largest element into position.

Algorithm 4.4. buildHeap(A) Reorder an unsorted array A into a max heap array.

Pre-condition Array A is a valid array. The *maxHeapify()* algorithm exists to move individual array elements to their correct position in the heap.

Post-condition A max heap array is constructed from the unsorted array.

Algorithm buildHeap(A) 1. for x = floor(A.length/2)-1 to 0
2. maxHeapify(A, x)

Algorithm 4.5. maxHeapify(A, x) Reorders individual elements in an array by moving larger elements to the top of the heap.

Pre-condition *x* is a valid index in the array and *A* is a valid array.

Post-condition An individual element in the x index of the heap array is moved into its correct position in the heap.

Algorithm maxHeapify(A, x) 1. left = 2 * x + 1
2. right = 2 * x + 2
3. if (left < A.length and A[left] > A[x]) 4. largest = left 5. else 6. largest = x 7. if right < A.length and A[right] > A[largest]) 8. largest = right 9. if largest != x 10. temp = A[x]
11 A[x] = A[largest]
12. A[largest] = temp 13. maxHeapify(A, largest)

Example 4: Build a max heap from the array in Figure 31.

First iteration of the for loop in *buildHeap()*, x = 4

• **Line 2**: Call *maxHeapify(A, 4)*.

maxHeapify(A,4) • **Line 1**: Set *left = 9*.

• **Line 2**: Set *right = 10*.

• **Line 3**: The first half of the conditional is true. The second half fails, since *A[left] = 34* and *A[x] = 14*.
• **Line 6**: Set *largest = x = 4*.
• **Line 7**: The first half of the conditional fails, since *right = A.length = 10*.
• **Line 9**: The conditional fails and *maxHeapify()* exits. The algorithm returns to *buildHeap()*.

Second iteration of the for loop in *buildHeap()*, x = 3
• **Line 2**: Call *maxHeapify(A, 3)*.

maxHeapify(A, 3) • **Line 1**: Set *left = 7*.
• **Line 2**: Set *right = 9*.
• **Line 3**: The conditional is true, since 7 < 10, *A[left] = 40*, and *A[x] = 13*.
• **Line 4**: Set *largest = 7*.
• **Line 7**: The second half of the conditional fails, since *A[right] = 7*, and *A[largest] = 40*.
• **Line 9**: The conditional is true. On lines 10-12, the values for *A[3]* and *A[7]* are swapped to produce the array shown in Figure 33, and the corresponding binary tree shown in Figure 34.
• **Line 13**: Call *maxHeapify(A, 7)*.

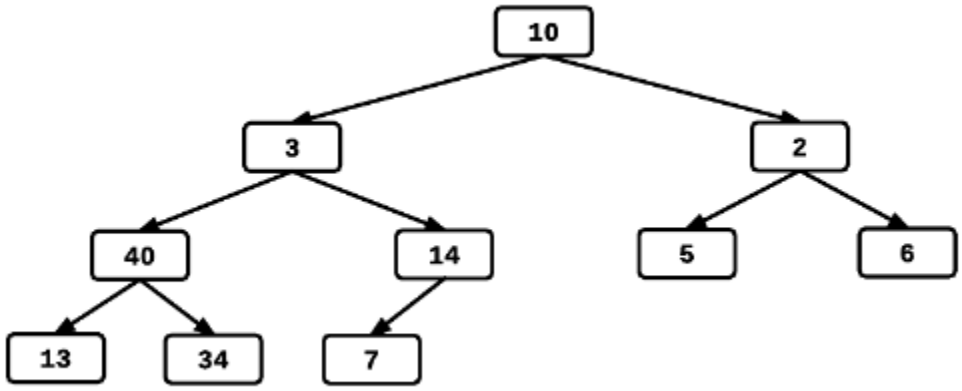


Figure 33. Binary tree after the 13 and 40 nodes are swapped in the buildHeap() algorithm. The corresponding array is shown in Figure 34.

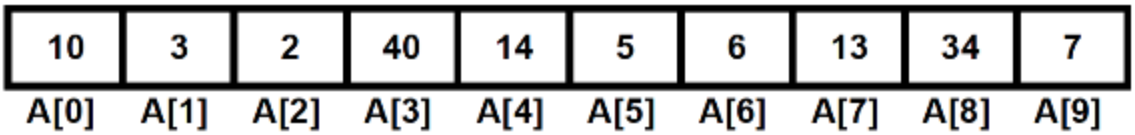


Figure 34. The array contents for the binary tree in Figure 33 after the 40 and 13 are swapped in the buildHeap() algorithm.

maxHeapify(A, 7) • **Line 1**: Set *left = 15*.
• **Line 2**: Set *right = 16*.
• **Line 6**: Set *largest = x = 7*.
• **Line 7**: The conditional is false because *right = 16*, and *A.length = 10*.
• **Line 9**: The conditional is false because *largest = x*, and this call to *maxHeap(A, 7)* exits. The program returns to the previous function call, *maxHeap(A, 3)*. The call to *maxHeap(A, 3)* exits, and the program returns to *buildHeap()*.

Third iteration of the for loop in *buildHeap()*, x = 2
• **Line 2**: Call *maxHeapify(A, 2)*.

maxHeapify(A, 2) • **Line 1**: Set *left = 5*.
• **Line 2**: Set *right = 6*.
• **Line 3**: The conditional is true because 5 < 10, and *A[5] > A[2]*.
• **Line 4**: Set *largest = left = 5*.
• **Line 7**: The conditional is true, since 6 < 10, and *A[6] > A[5]*.
• **Line 8**: Set *largest = right = 6*.
• **Line 9**: The conditional is true because *x = 2* and *largest = 6*.
• **Line 10-12**: Swap *A[2]* and *A[6]* . The new tree is shown in Figure 35, and the new array is shown in Figure 36.
• **Line 13**: Call *maxHeapify(A, 6)*.

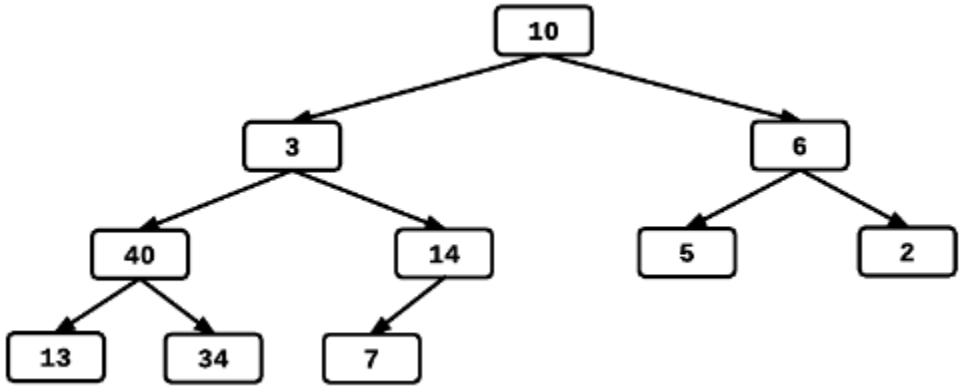


Figure 35. Binary tree after the 6 and the 2 are swapped in the buildHeap() algorithm. The corresponding array is shown in Figure 36.

10	3	6	40	14	5	2	13	34	7
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Figure 36. The array contents for the binary tree in Figure 35 after A[2] and A[6] are swapped in the buildHeap() algorithm.

maxHeapify(A, 6) • **Line 1**: Set *left = 13*.
• **Line 2**: Set *right = 14*.
• **Lines 3, 7, 9**: The conditionals are all false and the routine exits, and the program returns to *buildHeap()*.

Fourth iteration of the for loop in *buildHeap(), x = 1*
• **Line 2**: Call *maxHeapify(A, 1)*.

maxHeapify(A, 1) • **Line 1**: Set *left = 3*.
• **Line 2**: Set *right = 4*.
• **Line 3**: The conditional is true, since 3 < 10, and *A[3] > A[1]*.
• **Line 4**: Set *largest = left = 3*.
• **Line 7**: The conditional is false, since *A[4] < A[3]*.
• **Line 9**: The conditional is true, since *largest = 3* and *x = 1*.
• **Line 10-12**: Swap *A[1]* and *A[3]*. The new tree is shown in Figure 37 and the corresponding array is shown in

Figure 38.
• **Line 13**: Call *maxHeapify(A, 3)*.

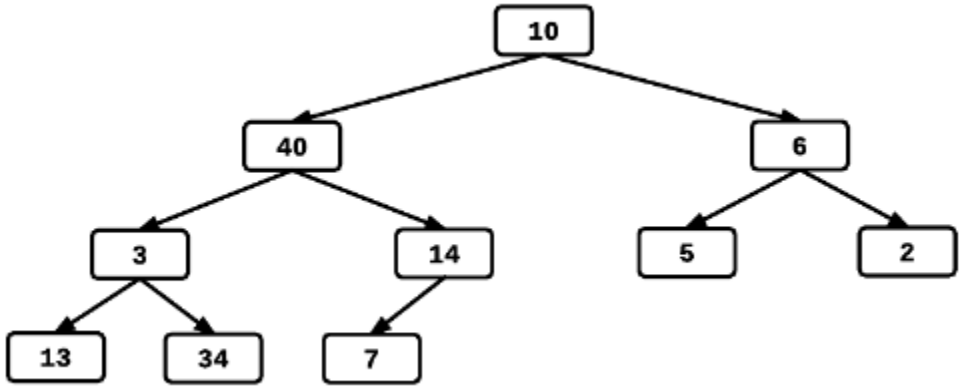


Figure 37. Binary tree after the 40 and the 3 are swapped in the buildHeap() algorithm. The corresponding array is shown in Figure 38.

10	40	6	3	14	5	2	13	34	7
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Figure 38. The array contents for the binary tree in Figure 37 after the 40 and the 3 are swapped in the buildHeap() algorithm.

maxHeapify(A, 3) • **Line 1**: Set *left = 7*.
• **Line 2**: Set *right = 8*.
• **Line 3**: The conditional is true, since 7 < 10 and *A[7] > A[3]*.
• **Line 4**: Set *largest = left = 7*.
• **Line 7**: The conditional is true, since 8 < 10 and *A[8] > A[7]*.
• **Line 8**: Set *largest = right = 8*.
• **Line 9**: The conditional is true, since *largest = 8* and *x = 3*.
• **Lines 10-12**: Swap *A[3]* and *A[8]*. The new tree is shown in Figure 39 and the corresponding array in Figure 40.
• **Line 13**: Call *maxHeapify(A, 8)*.

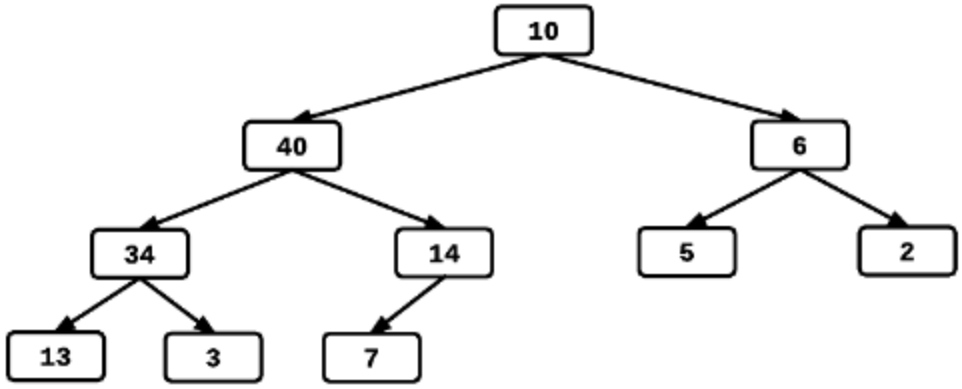


Figure 39. Binary tree after the 34 and the 3 are swapped in the buildHeap() algorithm. The corresponding array is shown in Figure 40.

10	40	6	34	14	5	2	13	3	7
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Figure 40. Array configuration for the binary tree in Figure 39 after the 34 and the 3 are swapped in the buildHeap() algorithm.

maxHeapify(A, 8) This call to *maxHeapify()* won't result in any changes to the array *A*. After the function exits, control returns to *buildHeap()*.

Fifth iteration of the for loop in *buildHeap(), x = 0*
• **Line 2**: Call *maxHeapify(A, 0)*.

maxHeapify(A, 0) • **Line 1**: Set *left = 1*.
• **Line 2**: Set *right = 2*.
• **Line 3**: The conditional is true, since 0 < 10, and *A[1] > A[0]*.
• **Line 4**: Set *largest = left = 1*.
• **Line 7**: The conditional fails, since *A[2] < A[1]*.
• **Line 9**: The conditional is true, since *largest = 1*, and *x = 0*.
• **Lines 10-12**: Swap *A[0]* and *A[1]*. The new tree is shown in Figure 41 and the corresponding array in Figure 42.
• **Line 13**: Call *maxHeapify(A, 1)*.

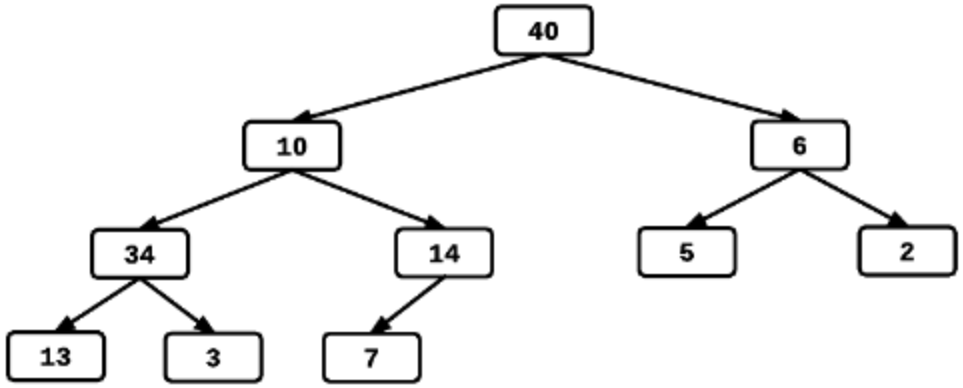


Figure 41. Binary tree after the 40 and the 10 are swapped in the buildHeap() algorithm.

40	10	6	34	14	5	2	13	3	7
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Figure 42. The array for the binary tree in Figure 41 after the 40 and the 10 are swapped in the buildHeap() algorithm.

maxHeapify(A, 1) • **Line 1**: Set *left = 3*.
• **Line 2**: Set *right = 4*.
• **Line 3**: The conditional is true, since 1 < 10 and *A[3] > A[1]*.
• **Line 4**: Set *largest = left = 3*.
• **Line 7**: The conditional is false, since *A[4] < A[3]*.
• **Line 9**: The conditional is true, since *largest = 3* and *x = 1*.
• **Lines 10-12**: Swap *A[3]* and *A[1]*. The new tree is shown in Figure 43, and the corresponding array is shown in Figure 44.
• **Line 13**: Call *maxHeapify(A, 3)*.

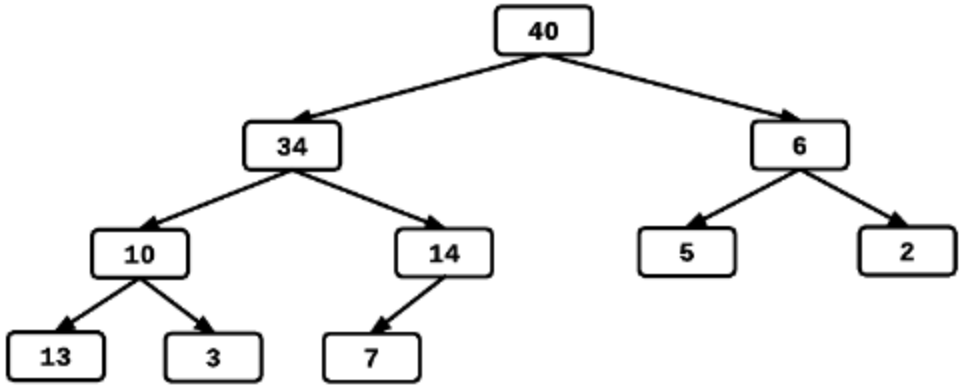


Figure 43. The binary tree after the 34 and the 10 are swapped in the buildHeap() algorithm.

40	34	6	10	14	5	2	13	3	7
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Figure 44. The array contents for the binary tree in Figure 43 after the 34 and the 10 are swapped in the buildHeap() algorithm.

maxHeapify(A, 3) • **Line 1**: Set *left = 7*.
• **Line 2**: Set *right = 8*.
• **Line 3**: The conditional is true, since 3 < 10 and *A[7] > A[3]*.
• **Line 4**: Set *largest = left = 7*.
• **Line 7**: The conditional is false, since *A[8] < A[7]*.
• **Line 9**: The conditional is true, since *largest = 7* and *x = 3*.
• **Lines 10-12**: Swap *A[3]* and *A[7]*. The new tree is shown in Figure 45, and the corresponding array in Figure 46.
• **Line 13**: Call *maxHeapify(A, 7)*.

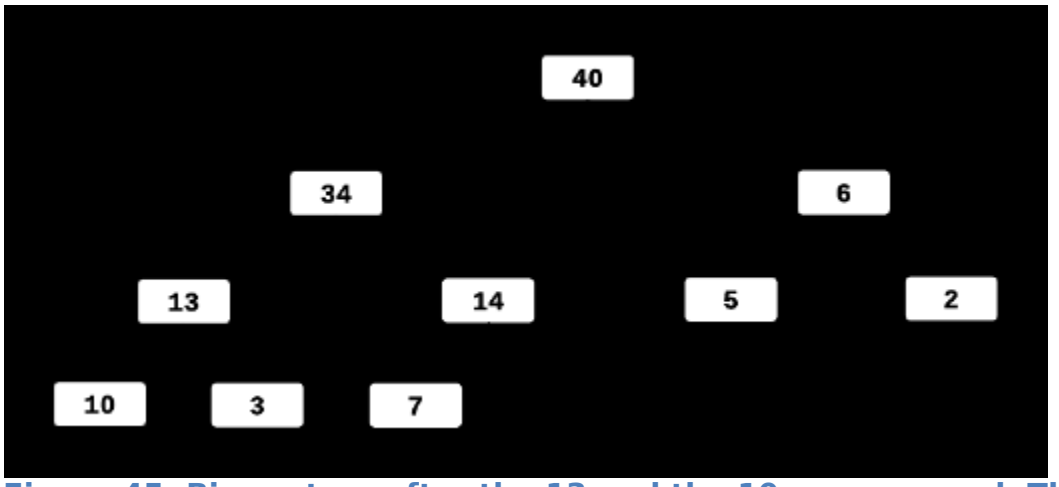


Figure 45. Binary tree after the 13 and the 10 are swapped. The binary tree is now a max heap, where the root is the maximum value in the tree and the value of every node is greater than the value of its children.

40	34	6	13	14	5	2	10	3	7
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Figure 46. The array contents for the max heap in Figure 45 after the 13 and the 10 are swapped in the buildHeap() algorithm.

maxHeapify(A, 7) This call to *maxHeapify()* exits without making any changes to A. Control returns to *buildHeap()*. The **for** loop has processed all elements in the array *A*. The heaping process is complete; the binary tree is now a max heap.

4.5.2 Sort the array

The heap sort algorithm works by removing the root of the heap and adding it to the array. The heap is then reordered to install a new root element selected from the heap. The heap sort algorithm is shown in Algorithm 4.6.

Algorithm 4.6. heapSort(A) Sort the array *A* by removing elements from the heap and adding them to the correct location in the sorted array.

Pre-condition Routings to build the heap and re-order the heap when the root is removed exist.

Post-condition Array *A* is sorted.

Algorithm heapSort(A) 1. buildHeap(A) 2. heapSize = A.length 3. for x = A.end to 1: 4. swap A[x] and A[0]
5. heapSize = heapSize - 1
6. maxHeapify (A, 0)

4.6 Merge Sort

Merge sort is another divide-and-conquer algorithm that has faster worst-case behavior than any of the sorting algorithms discussed so far. Merge sort operates by sorting small sub-arrays first, and then merges these small sub-arrays into larger and larger sub-arrays. The last merge builds the final, sorted array.

There are two steps to the merge sort algorithm
1. Divide: Divide the array of n items into n sub-arrays with 1 item each.
2. Conquer: Recursively recombine the sub-arrays into sorted sub-arrays. Combine the sorted sub-arrays to create the final sorted array.

4.6.1 Merging arrays

The merge sort algorithm uses a process of merging two arrays that are each individually sorted, but need to be combined in such a way that the new array is also sorted. For example, Figure 47 shows two sorted arrays, array A and array B, that need to be combined.

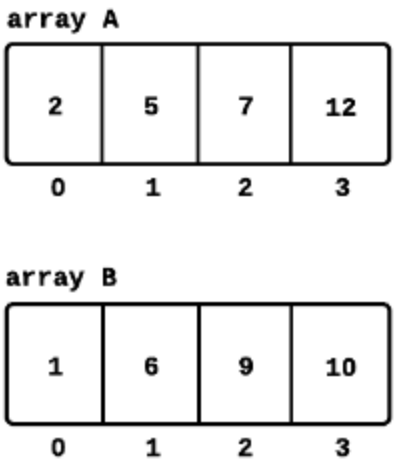


Figure 47. Example of two sorted arrays, array A and array B, which need to be merged together to created one sorted array.

Appending array *B* to the end of array *A* would generate the new array shown in Figure 48, which is clearly not sorted.

Figure 48. Array generated by appending array B to the end of array A. (Both arrays shown in Figure 47.) The array clearly is not sorted.

Another option for merging the arrays, which also won't work, is to weave the arrays together by selecting an element from one array, and then selecting an element from the other array, until all elements from both arrays have been added to the new array. For example, using the arrays *A* and *B*, select first from *A*, then *B*, then *A*, then *B*, and so on until all items have been merged together. Figure 49 shows the new arrays that would be generated by selecting *A* then *B* or by selecting *B* then *A*. Neither array is correctly sorted.

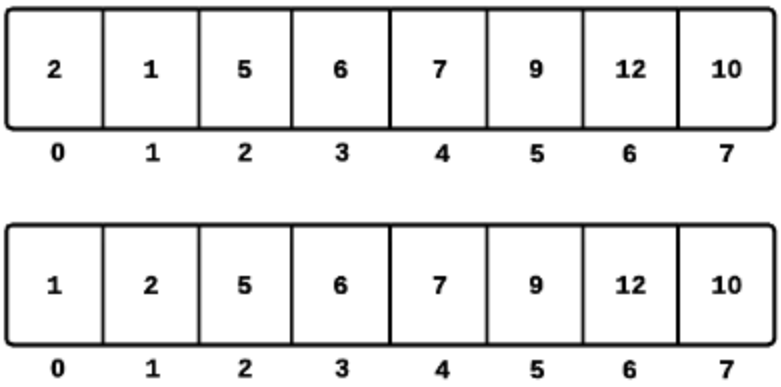


Figure 49. Resulting arrays of merging A and B by weaving together one element at a time from each array. The top array is the result of selecting an element from A and then an element from B. The bottom

array is the result of selecting an element from B and then an element from A. Neither array is correctly sorted.

The algorithm to correctly merge two arrays includes the following steps:

• Compare the first elements in each array and add the lowest value to a new array.

• Use an index on the original arrays to control which elements have been added to the new array. Increment the index when an element has been added from that array. For example, if *A[0]* is added to the array, then increment index to 1 so that *A[1]* will be the element evaluated next.

• Repeatedly compare the minimum values in each array that haven't been added to the new array and select the minimum of the two values. Increment the index each time a value is selected from an array.

Example 5: Merge the two arrays shown in Figure 32 into one sorted array.

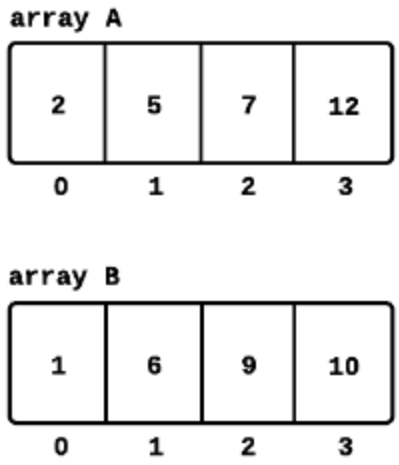


Figure 50. Arrays to merge in Example 5. Arrays A and B need to be combined into one sorted array.

Steps:

1. Compare A[0] and B[0], which are 2 and 1. The 1 is lowest, so add it to the new array *C* at *C[0]*. Increment the

B index to 1.
2. Compare *A[0]* and *B[1]*, which are 2 and 6. Add the 2 to the new array *C* at *C[1]*. Increment the *A* index to 1.
3. Compare *A[1]* and *B[1]*, which are 5 and 6. Add the 5 to *C* at *C[2]*. Increment the *A* index to 2.
4. Compare *A[2]* and *B[1]*, which are 7 and 6. Add the 6 to *C* at *C[3]*. Increment the *B* index to 2.
5. Compare *A[2]* and *B[2]*, which are 7 and 9. Add the 7 to *C* at *C[4]*. Increment the *A* index to 3.
6. Compare *A[3]* and *B[2]*, which are 12 and 9. Add the 9 to *C* at *C[5]*. Increment the *B* index to 3.
7. Compare *A[3]* and *B[3]*, which are 12 and 10. Add the 10 to *C* at *C[6]*. The *B* index does not need to be incremented because the end of the array has been reached.
8. Add *A[3]*, which is 12 to *C* at *C[7]*.

4.6.2 Merging arrays with merge sort

Merge sort works by repeatedly sorting small sections of the array and then merging them together into a larger sorted section of the array.

Example 6: Sort the array shown in Figure 51 using merge sort.

Figure 51. Example array that will be used to demonstrate the merge sort algorithm for Example 6.

Divide the array into seven sub-arrays each with one element, as shown in Figure 52. The one-element size is the smallest unit possible for an array, and guarantees that any two elements merged correctly will be a sorted two-element sub-array.

Figure 52. The first step in the merge sort algorithm is to divide the array into the smallest possible units, which is n sub-arrays each with one element.

Merge the adjacent one-element sub-arrays into sorted pairs using the merging process previously described. With an odd number of sub-arrays, there will be one sub-array after the merge that only has item. The result of the merging is shown in Figure 53.

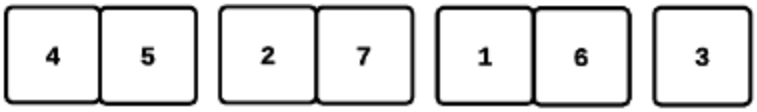


Figure 53. The result of merging the n, one-element sub-arrays by combining adjacent elements into sorted pairs.

Merge the adjacent two-element sub-arrays into sorted sub-arrays of three or four elements each. The result of this merge is shown in Figure 54.

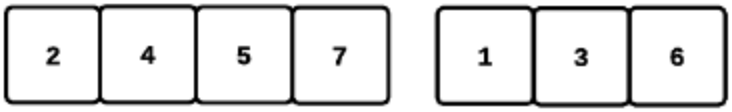


Figure 54. Combine adjacent, two-element sub-arrays into sorted, four-element sub-arrays. Each of the two sub-arrays is sorted with respect to the other elements in the subarray.

The next merge will produce the final sorted array. Merge the four-element sub-arrays to get one, seven element array that will be correctly sorted (Figure 55).

Figure 55. The final sorted array that is produced from merging the three- and four-element sub-arrays.

Complexity of merge sort

Big-Oh is O(n log n)

5 Linked lists

One of the limitations of arrays is that they have a fixed size. Allocating memory to store additional data once the array is full is generally handled with an array-doubling algorithm, which can be computationally expensive. Array doubling also allocates too much memory if only one or two additional elements need to be added.

A list is a data structure that allows for individual elements to be added and removed as needed. In a typical list implementation, called a linked list, memory is allocated for individual elements, and then pointers link those individual elements together.

5.1 Singly and doubly linked lists

There are two types of linked lists, singly linked and doubly linked lists. In a singly linked list, each element, which is also called a node, contains the data stored in the node and a pointer to the next node in the list (shown in Figure 1).

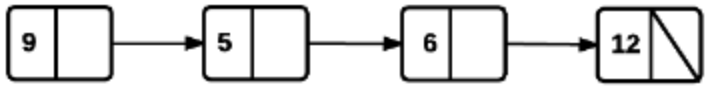


Figure 1. Singly linked list with four elements, called nodes. In this example, each node has an integer key value and a pointer to the next node in the list.

In the Figure 1 example, the node data is the integer key. The first node has a key value of 9, the second node has a key value of 5, the third node has a key value of 6, and the fourth node has a key value of 12. The next pointer for the final node in the list is set to NULL, which is shown by the slanted line.

In a doubly linked list, each node in the list contains the node data, a pointer to the next node in the list, and a pointer to the previous node in the list. A graphical example of a doubly linked list is shown in Figure 2. In this example, each node has three properties: an integer key, a pointer to the next node in the list, and a pointer to the previous node in the list. For the first node in the list, the previous pointer is set to NULL, and for the last node in the list, the next pointer is set to NULL. Nodes in a linked list can also be much more complex than these simple examples. Nodes could, for example, be built from a class that defines an object such as an Automobile or a Bicycle.

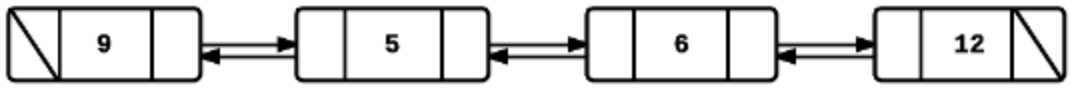


Figure 2. Doubly linked list with four nodes. Each node has an integer key value, a pointer to the previous node, and a pointer to the next node in the list.

One difference between storing data in a linked list or in an array is that the linked requires additional pointers to the neighboring nodes. Storing the same data from Figures 1 and 2 in an array would require a four-element integer array, such as the one shown in Figure 3.

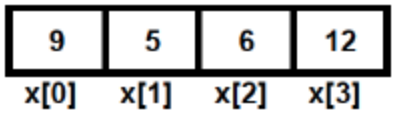


Figure 3. Array example showing how the data in the linked list in Figures 1 and 2 would be stored in an array.

5.1.1 Head and tail nodes

The node at the beginning of the list is called the head of the list. When implementing a linked list, a separate pointer should be stored to this node as it is the only entrance to the list. The last node in the list is called the tail of the list. In some implementations, a pointer to the tail of the list is also stored.

5.2 The linked-list ADT

In a linked-list ADT, shown in ADT 5.1, the data is stored in a linked list that is accessed through the head of the list. The *head* and *tail* of the list are stored as private variables, and there are public methods to initialize the list, add and delete nodes, traverse the list, and search the list.

ADT 5.1. Linked List LinkedList: 1. private: 2. head 3. tail 4. public: 5. Init() 6. insertNode(previousValue, value) 7. search(value) 8. traverse() 9. deleteNode(value) 10. deleteList()

5.2.1 C++ implementation of a node

All variables stored in memory have a memory location that can be accessed using a pointer variable. A linked list node can be implemented in C++ using a **class** or a **struct** and the *next* and *previous* pointers in the node reference another instance of the node.

The example code in Code 5.1 shows simple node definitions using a **struct**. The *singleNode* for a singly linked list includes an integer key and a *singleNode next* pointer. The *doubleNode* implementation for a doubly linked list includes a *next doubleNode* pointer and a *previous doubleNode* pointer.

Code 5.1. singleNode and doubleNode definitions //node implementation for singly linked list struct singleNode{

```
 int key; singleNode *next; }

//node implementation for doubly linked list struct
doubleNode{
int key; doubleNode *next; doubleNode *previous; }
```

5.3 Building a singly linked list in C++

The Linked List ADT defines an interface for the operations on a linked list. It can also be helpful to step through how a list is created independent of how the ADT is structured. The next example steps through creating a linked list with three nodes using the *singleNode* definition given in Code 5.1.

Example 1: Build a linked list with three nodes with key values of 5, 6, and 7.

Steps:
1. Using the node definition given above, create a new node dynamically:

```
singleNode *x = new singleNode;
```

2. Set the values for the *key* and *next* of *x*. The *key* value for the first node is 5. The *next* value is initialized to NULL because there are no other nodes in the list.

```
x->key = 5;
x->next = NULL;
```

To store this node as the head of the linked list, create an additional pointer *head* to point to *x*.

```
singleNode *head = x;
```

The linked list now has one node (Figure 4).

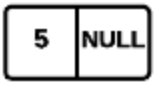


Figure 4. Singly linked list with one node that includes an integer key with a value of 5 and a pointer to the next node in the list. The next

pointer is set to NULL since there are no other nodes in the list to point to.

3. Create another node dynamically.

```
singleNode *n1 = new singleNode;
```

4. Set the values for the *key* and *next* of *n1*. The *next* pointer is initialized to NULL because this node is added to the end of the list.

```
n1->key = 6;
n1->next = NULL;
```

At this point, two nodes have been created, but there is no connection between them, illustrated in Figure 5. Both nodes have a *next* pointer that points to NULL.

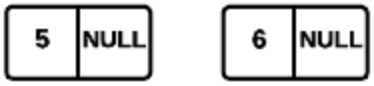


Figure 5. Two nodes have been created in memory for a singly linked list, but there isn't yet a link between them because the next pointer for both nodes is NULL.

5. Set the *x.next* pointer to connect the two nodes.

```
x->next = n1;
```

The address of *n1* is now stored in *x->next,* which establishes the link between the two nodes, as shown in Figure 6.

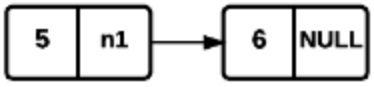


Figure 6. Singly linked list with two nodes. The next pointer of the first node is set to the address of the second node.

The arrow between the nodes in Figure 6 doesn't have any meaning in the code, but it does illustrate the possible movement direction. The node *x* contains a pointer to *n1*, which can be thought of as *x* knows about *n1*, and therefore, it's possible to traverse from *x* to *n1*. But, *n1* doesn't contain an arrow to *x* in the image, and in the code, *n1* doesn't contain a pointer to *x*. Since there isn't a pointer that connects *n1* to *x*, *n1* doesn't know about *x*, and it's not possible to go from *n1* to *x*.

6. Create another node dynamically and connect it to *n1*.

```
//create a new node
singleNode *n2 = new singleNode;
n2->key = 7;
n2->next = NULL;

//update the next pointer of n1 to point to the new node
n1->next = n2;
```

The final linked list with three nodes is shown in Figure 7.

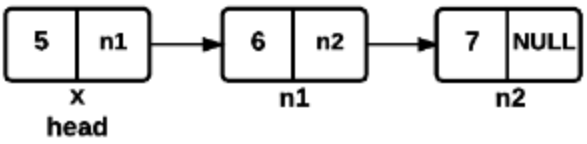


Figure 7. Singly linked list with three nodes. The key values for the three nodes are 5, 6, and 7. Each node also contains a pointer to the next node in the list, except the last node, which points to NULL. The first node is also stored as the head of the list.

5.4 Building a doubly linked list

The only difference between building a singly linked list and a doubly linked list is the additional *previous* pointer on each node in a doubly linked list.

Steps: 1. Create a new node dynamically:
doubleNode *n0 = new doubleNode;
2. Set the values for the *key, next,* and *previous* of *n0*. In this example, the *key = 0*, and both *next* and *previous* are initialized to NULL.

n0->key = 5; n0->next = NULL; n0->previous = NULL;
The linked list now has one node (Figure 8).

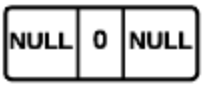


Figure 8. Doubly linked list with one node. The node has a next and a previous pointer to point to neighboring nodes in both directions in the list.

3. Create another node dynamically, and set the values for the *key, next,* and *previous* of *n1*.

doubleNode *n1 = new doubleNode; n1->key = 1; n1->next = NULL; n1->previous = NULL;
4. Connect the nodes by setting the *next* and *previous* pointers for both nodes (Figure 9).

n0->next = n1; n1->previous = n0;

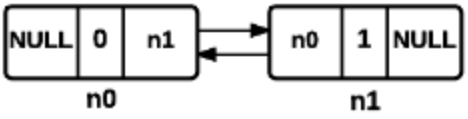


Figure 9. Doubly linked list with two nodes. Setting the next and previous pointers for both nodes establishes the connection between

them in both directions.

Just as with a singly linked list, the arrows between the nodes in this image don't have any meaning in the code, they just show the possible directions of movement between the nodes. Both nodes contain pointers to each other, indicating that *n1* is reachable from *n0* and vice versa. This is the difference between a singly and a doubly linked list, the list can be traversed both forwards and backwards.

5. Create another node dynamically and connect it to *n1*. The final linked list with three nodes is shown in Figure 10.

```
//create a new node doubleNode *n2 = new doubleNode;
n2->key = 2; n2->next = NULL; n2->previous = n1;
//update n1 to point to the new node n1->next = n2;
```

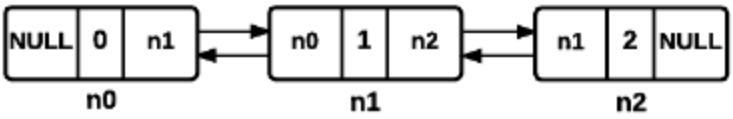


Figure 10. Doubly linked list with three nodes. The previous pointer for the first node in the list and the next pointer for the last node in the list are set to NULL, which signifies the beginning and the ending of the list.

5.5 Traversing a linked list

Unlike an array, where the individual elements are accessed through their index, linked list nodes are accessed through the pointers stored in the list. For example, using the linked list displayed in Figure 10, the command

singleNode *tmp = n0->next;

is equivalent to

singleNode *tmp = n1;

Neither of these commands allocates memory for a node, but rather, they create a pointer variable that points to an existing node in memory. To traverse a linked list, create a temporary variable that points to the head of the list, and then update the temporary variable to point to the next node in the list until the temporary variable points to NULL, which indicates that the end of the list has been reached. The algorithm to traverse a singly linked list is shown in Algorithm 5.1.

Algorithm 5.1. traverse() Traverse a linked list from the head node to the last node in the list.

Pre-conditions The *head* node is defined in the linked list ADT or included as an argument to the algorithm.

Post-conditions Values of the nodes in the list are displayed.

Algorithm traverse() 1. tmp = head 2. while(tmp != NULL) 3. print tmp.key 4. tmp = tmp.next

Example 2: Call traverse() on the linked list in Figure 11.

Steps in the *traverse()* algorithm:

Evaluate the head node • Line 1: the variable *tmp* points to the address of *head*, which is the first node in the list. This configuration is shown in Figure 11.

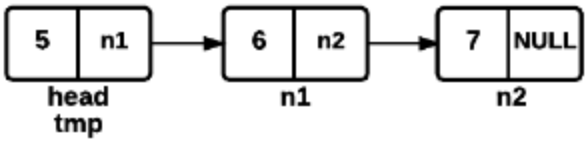


Figure 11. The variable tmp points to the first node in the linked list, which is also called the head of the list.

The properties of *head* can be accessed through *tmp*, which means that
tmp.next
will access the same data as
head.next.

• **Line 3**: the statement
print tmp.key
will display the key value for the first node in the list, which is 5.

• **Line 4**: *tmp* is updated to point to the next node in the list.

tmp = tmp.next
This step accesses the address of *n1* and sets *tmp* to point to that address. The *tmp* pointer now points to the second node in the list, as shown in Figure 12.

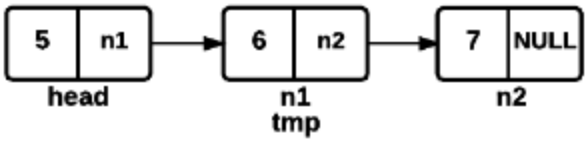


Figure 12. The tmp variable now points to the second node in the linked list.

Evaluate the second node • Line 2: check if *tmp* is NULL, and since it's pointing to *n1*, it is not NULL.
• **Line 3**: *tmp.key* displays the key value for *n1*, which is 6.
• **Line 4**: *tmp = tmp.next* changes *tmp* pointer to point to *n2*. The *tmp* pointer now points to the third node in the list, as shown in Figure 13.

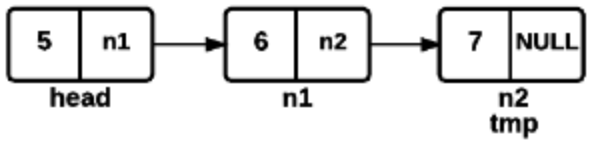


Figure 13. The tmp variable now points to the third node in the linked list, which is also the end of the list.

Evaluate the third node • Line 2: the check if *tmp* is NULL is still false, since *tmp* is pointing to *n2*.
• **Line 3**: the command *tmp.key* will display the *key* value for *n2*, which is 7.
• **Line 4**: *tmp = tmp.next* changes the *tmp* pointer to point to NULL. On the next evaluation of the conditional on Line 2, the conditional will be false and the loop will exit.

5.6 Searching a singly linked list

To search a linked list for a specified key, start at the head of the list and traverse the list until the key is found. The *search()* algorithm, shown in Algorithm 5.2, takes the search value as a parameter and returns a pointer to the node where the search value is found. This algorithm is similar to the *traverse()* algorithm (Algorithm 5.1) with additional steps to check if the value is found.

Algorithm 5.2. search(value) Traverse the linked list and return the node where the key matches the search value.

Pre-condition *value* is the same type as the key property.

Post-condition Returns the node that contains the value and NULL if the value does not exist in the list.

Algorithm search(value) 1. tmp = head 2. returnNode = NULL
3. found = false 4. while(!found and tmp != NULL) 5. if (tmp.key == value) 6. found = true 7. returnNode = tmp 8. else 9. tmp = tmp.next 10. return returnNode

5.7 Inserting a node into a singly linked list

When inserting a node to a linked list, there are three cases to consider: • Inserting a node at the head of the list • Inserting a node in the middle of the list • Inserting a node at the end of the list
In each case, the pointers in the nodes surrounding the new node need to be updated in a specified order for the operation to be successful.

Algorithm 5.3. insertNode(leftValue, value) Insert a new node into the linked list after the node with a key value of *leftValue*.

Pre-conditions *leftValue* is a valid key value for a node in the list, or NULL.
value is a valid key value
Post-conditions The new node has been added to the list after the *leftValue* node.

Note: This algorithm is for a singly linked list. To insert into a doubly linked list, additional steps are needed to handle the *previous* pointer for each node. When the new node is the head node, *node.previous = NULL*. Otherwise, *node.previous = left*.

Algorithm insertNode(leftValue, value) 1. left = search(leftValue) 2. node.key = value 3. if left == NULL //head node 4. node.next = head 5. head = node 6. else if left.next == NULL //tail node 7. left.next = node 8. tail = node 9. else //middle node 10. node.next = left.next 11. left.next = node

5.7.1 Inserting a new head node

Example 3: Insert a node at the head of the list in Figure 14 with a key value of 1.

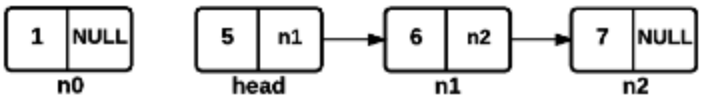
Figure 14. Linked list for Example 3, insert a new node at the head of the list.

In this example, the steps in the *insertNode()* algorithm are outlined, as well as the corresponding C++ commands to implement the algorithm.

Steps: • **Line 1:** Find the node in the list where *node.key = leftValue* using the *search()* algorithm. For a new head of the list, *leftValue* will be NULL and *left* will be NULL.

node *left = search(leftValue);
• **Line 2:** Allocate memory for the new node and set its *key* property (Figure 15).

singleNode *n0 = new node; n0->key = 1; n0->next =



NULL;

Figure 15. The new node has been created, called n0, but hasn't yet been linked to the list. The next pointer for the new node is still NULL.

• **Line 4:** Update the *next* property of the new node to point to the current *head* of the list (Figure 16).

n0->next = head;

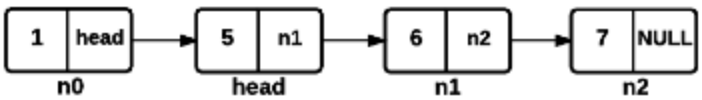


Figure 16. Connect the new node to the list by setting its next pointer to point to the head node.

• **Line 5:** Update the *head* pointer to point to the new node (Figure 17).

head = n0;

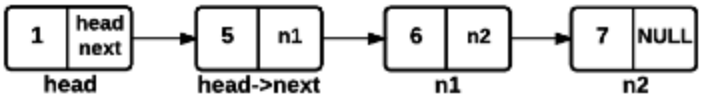


Figure 17. The new node becomes the new head node, and the previous head node is now head.next.

5.7.2 Inserting a new middle node

Example 4: Insert a node with a key value of 10 after the node with a key value of 2 in the linked list in Figure 18. The new linked list after the insert operation is shown in Figure 19.

Figure 18. Linked list for Example 4 before a new node is inserted to the list.

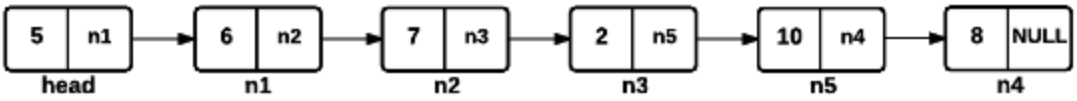


Figure 19. Linked list for Example 4 after the new node is inserted into the list.

Steps: • **Line 1:** Search for the node with *key = 2* using the *search()* algorithm. Figure 20 shows the results of the *search()* call.

singleNode *left = search(2);

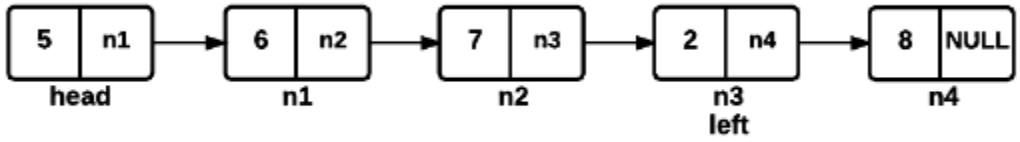


Figure 20. The node that will precede the new node is labeled as left. It was identified by calling the search() algorithm.

• **Line 2:** Allocate memory for the new node and set its *key* property.

singleNode *n5 = new node; n5->key = 10;

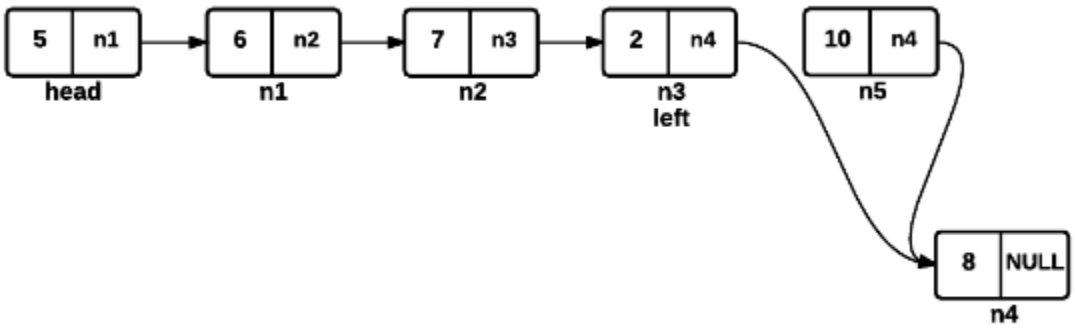


Figure 21. The new node has been created, called n5, and its next pointer points to the same node as left's next pointer.

• **Line 10:** Set the next pointer of the new node to point to *left.next*. There are now two nodes pointing to *n4* as their next node (Figure 21).

n5->next = left->next;
• **Line 11:** Update the *next* pointer for *left* to point to the new node (Figure 22).

left->next = n5;

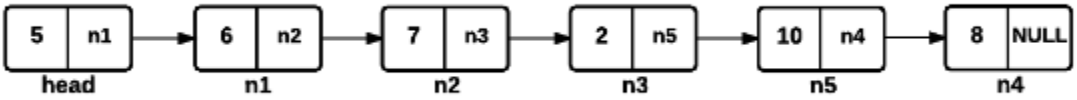


Figure 22. Final linked list after the new node is inserted and the pointers are updated.

5.7.3 Inserting a new tail node

Example 5: Insert a new node at the end of the linked list in Figure 23 with a key value of 10.

Figure 23. Linked list for Example 5, insert a new tail node at the end of the list with a key value of 10.

Steps: • **Line 1:** Search for node with *key = 8*.

singleNode *left = search(8);
(Note: without knowing the key value of the last node, the end of the list could also be accessed using the *tail* pointer.)
• **Line 2:** Allocate memory for the new node and assign it values of *key = 5* and *next = NULL*.

node *n5 = new node; n5->key = 10; n5->next = NULL;
• **Line 7**: Change the *next* pointer of the *left* node to point to the new node.

left->next = n5;
• **Line 8:** Update the new node to be the *tail*. The final linked list is shown in Figure 24.

tail = n5;

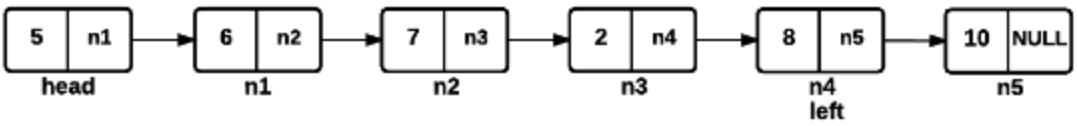


Figure 24. Linked list after new node inserted at the tail position.

5.8 Inserting a node into a doubly linked list

The only difference between inserting a node into a singly linked list and a doubly linked list is that the *previous* pointer for a node needs to be set on a doubly linked list. The algorithm for inserting a node into a doubly linked list is shown in Algorithm 5.4.

Algorithm 5.4. insertNodeDouble(leftValue, value) Insert a new node into the linked list after the node with a key value of *leftValue*.

Pre-conditions *leftValue* is a valid key value for a node in the list, or NULL for a new head node.
value is a valid key value
Post-conditions The new node has been added to the list after the *leftValue* node.

Algorithm insertNodeDouble(leftValue, value) 1. left = search(leftValue) 2. node.key = value 3. if left == NULL //head node 4. node.next = head 5. head.previous = node 6. head = node 7. else if left.next == NULL //tail node 8. left.next = node 9. node.previous = left 10. tail = node 11. else //middle node 12. left.next.previous = node 13. node.previous = left 14. node.next = left.next 15. left.next = node

Example 6: Insert a node with a key value of 3 into the doubly linked list in Figure 25 after the node with a key value of 1.

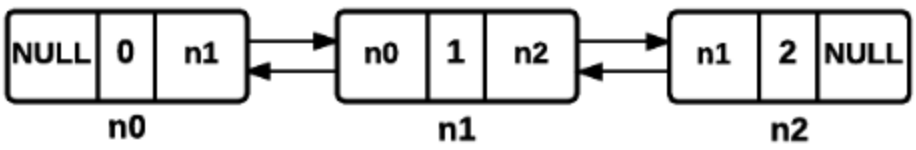


Figure 25. Doubly linked list for Example 6. Insert a new node with a key value of 3 after the node with the key value of 1.

Steps: • **Line 1:** Find the previous node in the list using the *search()* algorithm. The node is labeled *left* in Figure 26.

doubleNode *left = search(1);

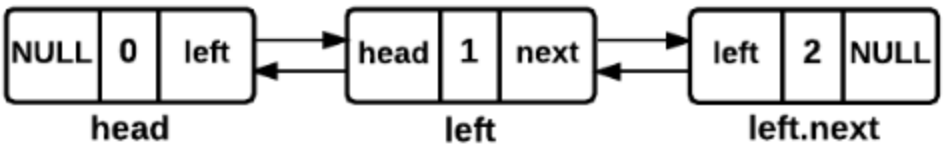


Figure 26. Doubly linked list with left node identified. The new node will be inserted after the left node.

• **Line 2:** Create the new node and assign it a *key,* and initialize its *next* and *previous* pointers. The state of the list after these steps is shown in Figure 27. The new node is labeled *n3*.

node *n3 = new node; n3->key = 3; n3->next = NULL; n3->previous = NULL;

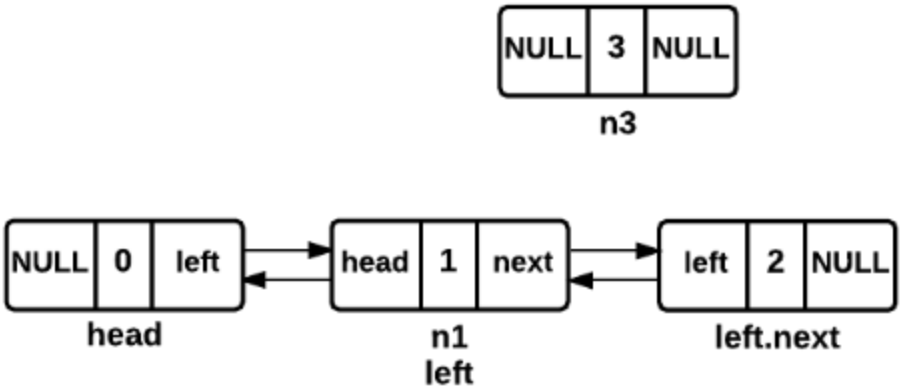


Figure 27. State of the doubly linked list with the new node inserted. The next and previous pointers are set to NULL, which means that the node is not yet connected to other nodes in the list.

• **Lines 12, 13:** Update the *previous* pointer for *left.next* to point to the new node as its previous node (Figure 28) and

set the *previous* pointer for the new node to point to *left*.

left->next->previous = n3; n3->previous = left;

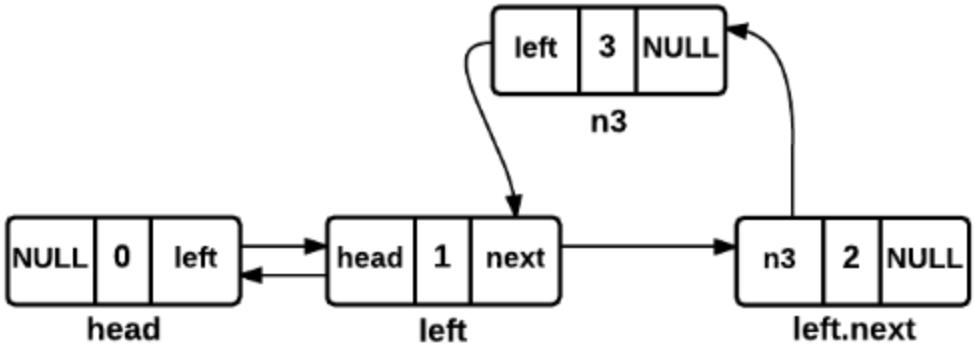


Figure 28. The previous pointer for n2 has been updated to point to the new node n3 instead of n1.

• **Line 14, 15:** Update the *next* pointer for the new node to point to *left.next*. Update the *next* pointer for *left* to point to the new node. (Figure 29).

n3->next = left->next; left->next = n3;

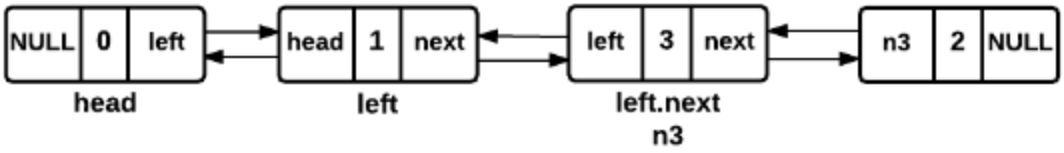


Figure 29. The next pointer for left has been updated to point to the new node n3. The linked list is now in its final state with the new node inserted and all pointers updated.

Example 7: Insert a node with a key value of 10 to the head of a doubly linked list.

Steps: • **Lines 2-3:** Create the new node and set its *key*, *next*, and *previous* properties.

node *n0 = new node; n0->key = 1; n0->next = head; n0->previous = NULL;

• **Line 5:** Update the previous pointer for the current *head* node to point to the new node.

```
head->previous = n0;
```

• **Line 6:** Update the *head* pointer to point to the new node.

```
head = n0;
```

5.9 Common pitfall when inserting a new node

When inserting a node into a linked list, it's important to update the next pointers in the correct order or a portion of the linked list can be lost. For example, consider this situation where memory has been allocated for the new node *n5*, but the *next* pointer to include it in the list hasn't yet been set (Figure 30).

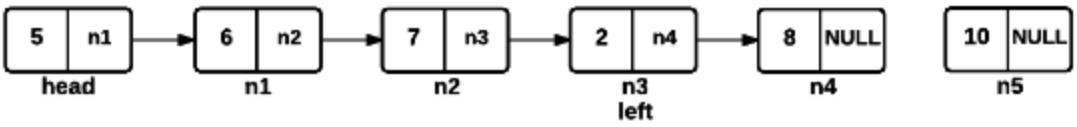


Figure 30. In this linked list, the memory for a new node has been allocated, but none of the pointers in the list have been updated to link in the new node.

If the *left.next* pointer is updated first to point to *n5* instead of *n4*, then *n4* will be disconnected from the list, as shown in Figure 31. The *next* pointer for *n4* is still NULL, and since it's a singly linked list, there's no connection back to *left* from *n4*.

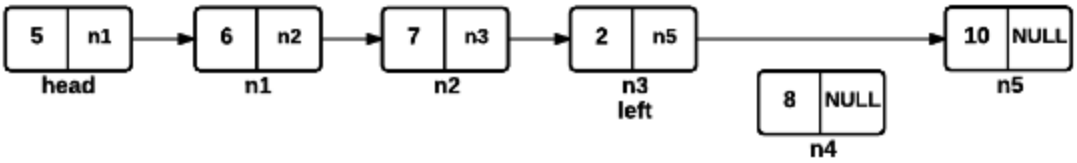


Figure 31. Example of how a section of the list can be lost if the pointers are not updated in the correct order. The node n4 is disconnected from the list if the left.next value is set before the n5.next value.

However, if the *n5.next* pointer is set first, then the *left.next* pointer can be updated to point to the new node and the list remains intact.

5.10 Deleting a node from a singly linked link

To delete a node from a linked list, update the pointers to bypass the node, and then free the memory associated with the node. Just as with the insert operations, the order of the steps is important to ensure that sections of the list are not lost and there are no memory leaks. There are three cases to consider when deleting a node from a linked list:

- Deleting the node at the head of the list.
- Deleting a node in the middle node of the list.
- Deleting the node at the tail of the list.

Algorithm 5.5 describes the delete operation. The algorithm takes the value of the node to delete and then searches for that value in the list. If the value is found, the node is deleted.

Algorithm 5.5. delete(value) Delete the node with the specified value.

Pre-conditions *head* and *tail* pointers are set in the linked list.
value is a valid search parameter.

Post-conditions Node where the key equals the value has been deleted from the list.

Algorithm delete(value) 1. if (head.key == value) 2. tmp = head 3. head = head.next 4. delete tmp 5. else //middle or tail 6. left = head 7. tmp = head.next 8. found = false 9. while tmp != NULL && !found 10. if tmp.key == value 11. left.next = tmp.next 12. if tmp == tail 13. tail = left 14. delete tmp 15. found = true 16. else 17. left = tmp 18. tmp = tmp.next

5.10.1 Delete the head node in a singly linked list

Example 8: Delete the node at the head of the list in Figure 32 using Algorithm 5.5.

This example shows the lines in the *delete()* algorithm that executes to delete the head of the list, as well as the corresponding C++ code to implement the algorithm.

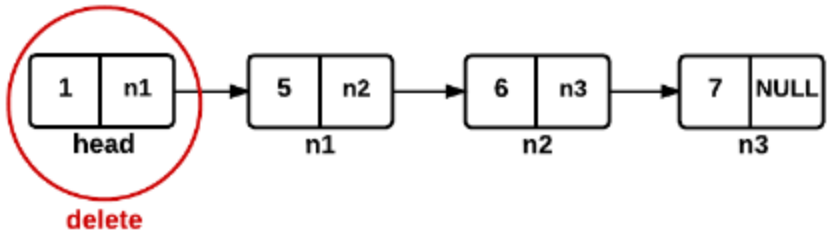


Figure 32. Linked list for Example 8, delete the first node in the list.

• **Line 2:** Create a pointer to point to the *head* of the list (Figure 33).

```
singleNode *tmp = head;
```

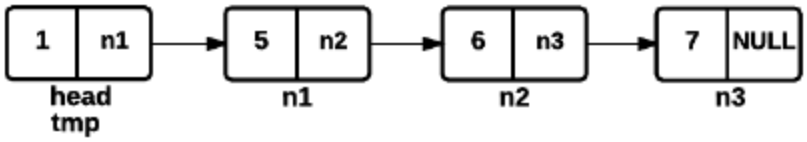


Figure 33. A variable tmp has been created and points to the first node in the linked list.

• **Line 3:** Set the *head* pointer to point to *head.next* (Figure 34).

```
head = head->next;
```

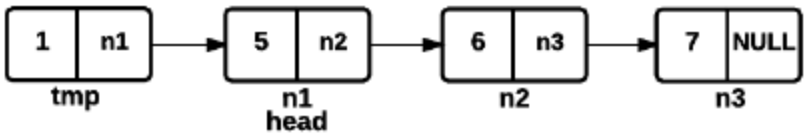


Figure 34. The head pointer is moved and now points to the second node in the linked list.

• **Line 4:** Delete *tmp* to free the memory allocated to the old *head* node (Figure 35).

delete tmp;

Figure 35. Free the memory associated with the tmp pointer, which removes the old head node from the list.

5.10.2 Delete a middle node in a singly linked list

Example 9: Delete the node with the key value of 5 from the linked list in Figure 36.

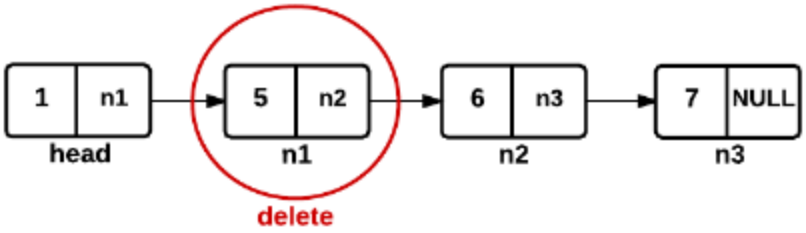


Figure 36. Linked list for Example 9, where the node with the key value of 5 needs to be deleted from the list.

Steps: • **Line 6**: Create a pointer to the *head* of the list.

singleNode *left = head;

• **Line 7**: Create a pointer to point to *head.next*.

singleNode *tmp = head->next;

• **Line 10:** Check if the value has been found in the list. At this point in the algorithm, *left* will be pointing to the preceding node and *tmp* will be the node to delete (Figure 37).

if(tmp->key == value)

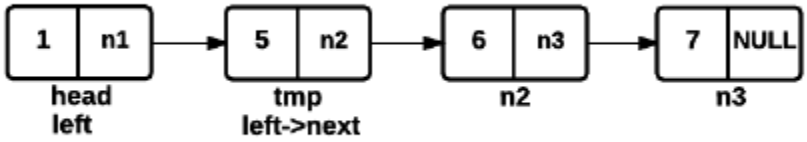


Figure 37. The variable tmp points to the node to delete.

• **Line 11**: Update the *left.next* pointer to bypass *tmp* and point to the node after *tmp,* which is *tmp.next.* (Figure 38).

left->next = tmp->next;

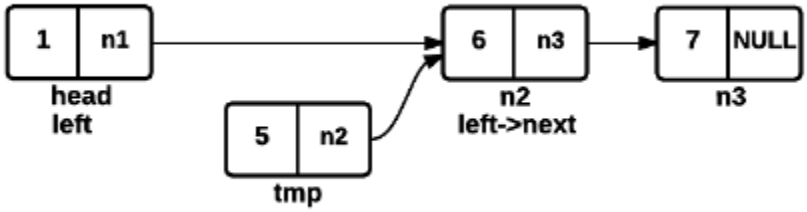


Figure 38. The next pointer for left has been updated to point to the node after tmp in the list. The tmp node can now be deleted without cutting the connection between left and the rest of the list.

• **Line 14**: Delete the *tmp* node (Figure 39).

delete tmp;

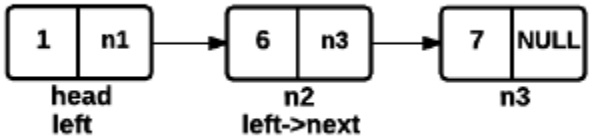


Figure 39. The memory associated with tmp has been freed and the node is no longer in the linked list.

5.10.3 Delete the tail node in a singly linked list

Example 10: Delete the tail node in the linked list in Figure 40.

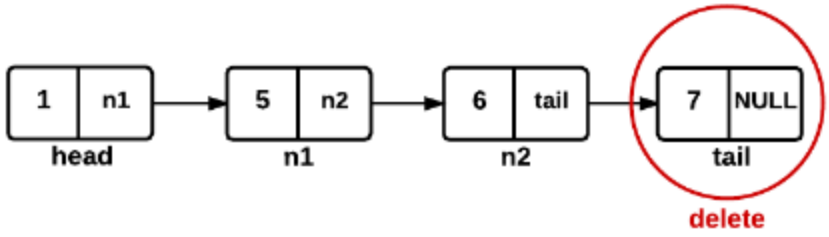


Figure 40. Linked list for Example 10, delete the tail node in the linked list.

• **Lines 12-13:** Handle the special case where the node to delete is the *tail* node. The node *left* is the node that precedes the *tail* node (Figure 41).

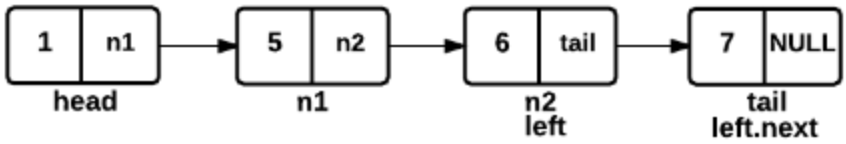
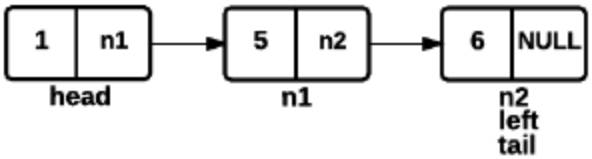


Figure 41. The left node is the node that precedes the tail node in the list.

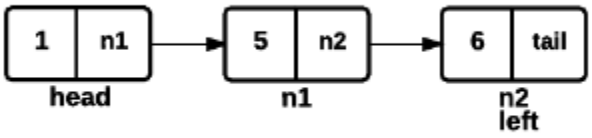
• **Line 13**: Update *left* to be the *tail* node and set its *next* pointer to NULL (Figure 42).



tail = left;

Figure 42. Final linked list after the tail node is removed and a new tail pointer is set. The variables left and tail now point to the same node.

• **Line 14**: Delete the *tmp* node, which is also *left.next* (Figure 43).



delete tmp;

Figure 43. Delete the tail first before setting the left pointer to be the new tail. This operation removes the tail node from the list.

5.11 Deleting a node from a doubly linked list

The operations to delete a node from a doubly linked list are similar to those for the deletion in a singly linked list with the addition of a previous pointer for each node. The three cases to consider when deleting a node from a doubly linked list are:

- Deleting the node at the head of the list.
- Deleting a node in the middle node of the list.
- Deleting the node at the tail of the list.

Algorithm 5.6 describes the delete operation. The algorithm uses a search algorithm to identify the node in the list to delete.

Algorithm 5.6. deleteDouble(value) Delete the node with the specified value.

Pre-conditions *head* and *tail* pointers are set in the linked list.
value is a valid search parameter.

Post-conditions Node where the key equals the value has been deleted from the list.

Algorithm deleteDouble(value) 1. node = search(value) 2. if (node == head) 3. head = head.next 4. head.previous = NULL
5. delete node 6. else //middle or tail 7. if (node == tail) 8. tail = tail.previous 9. tail.next = NULL
10. delete node 11. else 12. node.previous.next = node.next 13. node.next.previous = node.previous 14. delete node

Example 11: Delete the node with the key value of 5 from the linked list in Figure 44.

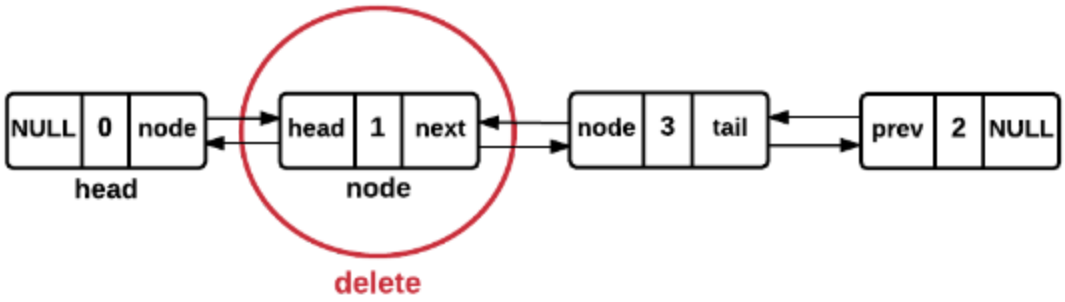


Figure 44. Delete the node with the key value of 1.

Line 1: Identify the node to delete using a search algorithm that returns a pointer to the node.

doubleNode *node = search(1);
Line 12: It's a middle node that will be deleted. Set the next pointer for the previous node to bypass the node to delete. In this example, the previous node is the head node. (Figure 45).

node->previous->next = node->next;

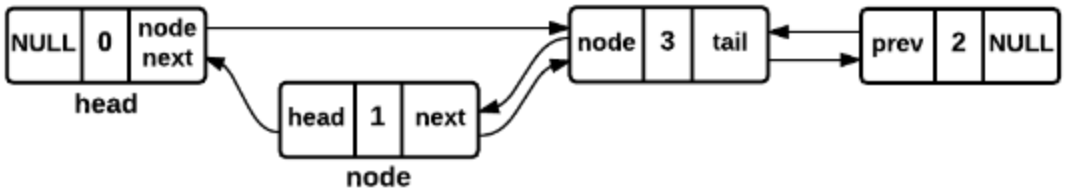


Figure 45. The next pointer for the head node has been updated to point to node.next.

Line 13: Set the previous pointer for *node.next* to bypass node and point to head. (Figure 46)
node->next->previous = node->previous;

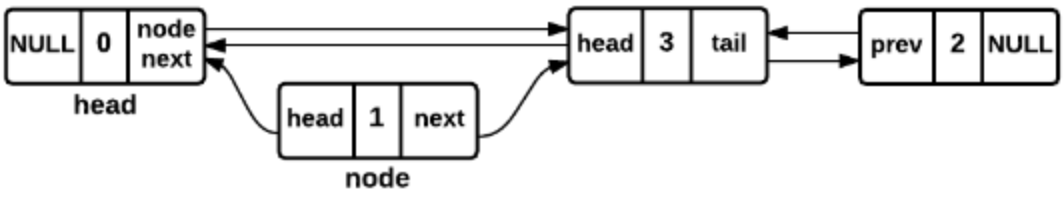


Figure 46. The node.next node now points to head as the previous node in the list.

Line 14: Delete the node. (Figure 47)

```
delete node;
```

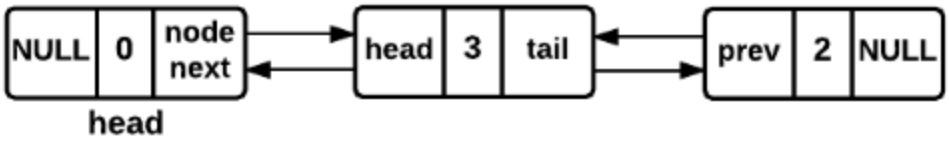


Figure 47. The node is removed from the list.

5.12 Complexity of linked list operations

Insert

• Insert a node at the head of the list: O(1).
• Insert a node at the end of the list: O(n), if there isn’t a tail pointer for the list, and O(1) if there is a tail pointer.
• Insert a node in the middle of the list: O(1) if the search complexity is calculated separately, and O(n) otherwise.

Search

• Search for a node within a specified key value: O(n).

Delete

• Delete the head of the list: O(1).
• Delete the tail of the list: O(n) for singly linked list and O(1) for doubly linked list.

5.13 Linked List Exercises

1. Write a C++ function to find the maximum value in a singly linked list. The function takes one argument - the head of the linked list and returns the maximum integer value in the list. The node and function prototype are:

   struct node {
   int value; node *next; };

   int LinkedListFindMax(node *head);
   Test case:
   For the following linked list

   9->3->100->1000->-3->9876

   the function should return a value of 9876.

2. Write a C++ function to reverse a linked list. The function takes one argument - the head of the linked list and returns the head of the new list. Your function should work for an arbitrary number of nodes. The node and function prototype are: struct node{
   int value; node *next; }

   node *ReverseLinkedList(node *head);
   Test case:
   For the following linked list

   0 -> 0 -> 50 -> 100

   the function should return

   100 -> 50 -> 0 -> 0

6 Stacks

A stack is a data structure that stores a collection of elements and restricts which element can be accessed at any time. Stacks work on a last in, first out principle (LIFO): the last element added to the stack is the first item removed from the stack, much like a stack of cafeteria plates. Elements are added to the top of the stack, and the element on the top is the only element that can be removed.

Example 1: Add the words of this classic *Napolean Dynamite* quote,

A liger it's pretty much my favorite animal

to a stack.

Each word in the sentence occupies one position on the stack. Words are added at the top position, and each time a word is added, the top position moves up by one. To begin, the top of the stack is at position 0 and the word “A” is added at that position. The top then moves to position 1 and “liger” is added at that position. After all words are added, the top is at position 8 (Figure 1).

Figure 1. Contents of the stack and the position of the top of the stack after all words have been added. Words are added at the top of the stack only, and then also removed from the top of the stack only.

Words are also removed from the top of the stack, which moves the top position. Removing the words from the stack in Figure 1 generates the following sequence of words:

animal. favorite my much pretty it's liger A

After all words are removed, the position of the top is 0.

Definitions:
When an element is added to a stack, it is “pushed” onto the stack.

When an element is removed from a stack, it is "popped" off the stack.

6.1 The stack ADT

The stack ADT includes a variable that tracks the top of the stack, the stack data, and methods to manipulate the stack by adding and removing elements. Stack data is typically stored in a data structure such as an array or a linked list. The terminology for interacting with the stack is the same regardless of the data structure used, but the implementation details vary. The stack ADT shown in ADT 6.1 is intentionally generic due to the differences in an array or linked list implementation.

ADT 6.1. Stack Stack: 1. private: 2. top 3. data 4. maxSize 5. public: 6. Init() 7. isFull() 8. isEmpty() 9. push(value) 10. pop()

6.2 Pushing and popping stack elements

6.2.1 Array implementation of a stack

In an array implementation of a stack, data elements are stored in an array and the top of the stack refers to the index where the next element will be added. The elements, *data[0... top-1]* are the contents of the stack.

• When *top = 0*, the stack is empty.
• When *top = maxSize*, the stack is full. (*maxSize* is the size of the array) • When *top > maxSize*, the condition is called stack overflow. Yes, it's called stack overflow.

6.2.2 Linked list implementation of a stack

In the linked list implementation of a stack, the data elements in the stack are nodes in a singly linked list, where each node has a pointer to the previous node in the list. The node at the bottom of the list has a previous pointer to NULL (Figure 2). The *top* of the stack is a pointer to a node instead of an index in an array.

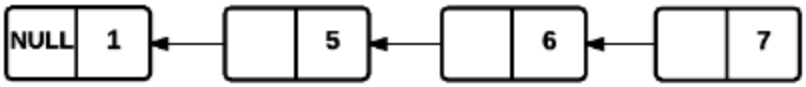


Figure 2. Example of a stack implemented with a linked list. Each node in the list has a pointer to the previous node in the list. This stack has four elements. The top of the stack is the node with the key value of 7.

• When *top = NULL*, the stack is empty.
• When *count = maxSize*, the stack is full. (*maxSize* is the maximum size of the stack. This example assumes that an additional *count* variable is used to track the number of nodes in the stack.)

6.2.3 Push an element onto an array stack

The algorithm to push an element onto a stack implemented with an array is shown in Algorithm 6.1.

Algorithm 6.1. push(value) Add an element with the specified value to an array stack.

Pre-conditions *value* is a valid input value.
A method, *isFull()* exists to check if the stack is full.

Post-conditions The *value* is added to the stack and the *top* index is incremented by 1.

Algorithm push(value) 1. if(!isFull()) 2. data[top] = value 3. top = top + 1

In this algorithm, *data* is the stack data structure and *value* is the element to add to the stack. The parameter *top* is initialized to 0 when the stack is initially created. On Line 1, the conditional to check for a full stack calls the *isFull()* method in the ADT, which checks if *top = maxSize*. When *top = maxSize*, *isFull()* returns true. Otherwise, *isFull()* returns false.

Example 2: Push the values 12 and 9 onto the empty stack S shown in Figure 3.

• Start with an empty stack, *S*

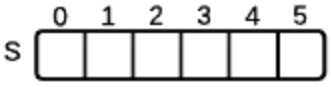


Figure 3. Empty array stack. The top of the stack is S[0].

• Pushing two elements onto the stack is handled in two separate calls to *push()*:
push(12) push(9)

Those two calls generate the stack configuration shown in Figure 4.

Figure 4. Stack contents and index of the top variable after two elements pushed onto the stack.

6.2.4 Push an element onto a linked list stack

In the *push()* operation on a linked list (shown in Algorithm 6.2), the input to the algorithm is the value for the new node to push onto the stack. The variable *top* is a pointer to the node at the top of the list. When the stack is empty, *top = NULL*.

Algorithm 6.2. push(value) Push a node with the specified value onto a stack implemented with a linked list.

Pre-conditions *value* is a valid linked list node value.

Post-conditions The new node is the new top of the stack.

Algorithm push(value) 1. node.key = value 2. if (!isEmpty()) 3. node.previous = top 4. else 5. node.previous = NULL
6. top = node

Example 3: Push a value of 10 onto the linked list stack shown in Figure 5.

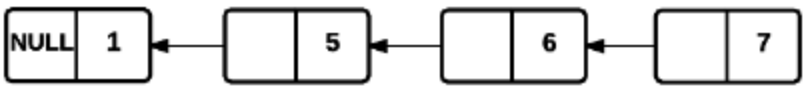


Figure 5. Push a new node onto this stack with a key value of 10.

push(10)

The sequence of operations involved in pushing the element onto the stack is shown color-coded in Figure 6.
• The initial state of the stack variables is shown in red. The *top* is the node with a key value of 7. The *x* node is the new node, which will become the new *top*.
• **Line 1:** Create a new node with the specified key value.
• **Line 3**: Set the *previous* pointer for *x* to connect the node to the stack, shown with the blue arrow.
• **Line 6**: Move *top* to point to *x,* shown in green.

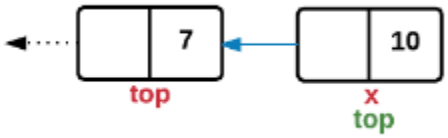


Figure 6. Steps for pushing a node onto a linked list stack. The initial state of the linked list is shown in red. The blue arrow connects the new 10 node to the existing list. The top is then reset to point to the new node.

6.2.5 Pop an element from an array stack

The algorithm to remove the *top* element from an array stack is shown in Algorithm 6.3.

Algorithm 6.3. pop() Pop an element from an array stack.

Pre-conditions None
Post-conditions Element at the top of the stack is returned and *top* decremented by 1.

Algorithm pop() 1. if(!isEmpty()) 2. top = top - 1
3. else 4. print("underflow error") 5. return data[top]

Example 4: Pop an element from the array stack shown in Figure 7.

Figure 7. A pop() operation on this array stack will return a 9 and decrement the top to S[1].

pop()
This call to *pop()* will return a 9.
• **Line 1**: the value of *top* is checked, and the conditional is false since *top = 2*.
• **Line 2**: *top* is decremented and is now 1.
• **Line 5**: the value of *S[1]* is returned, which is 9.

6.2.6 Pop an element off a linked list stack

The algorithm to pop an element off a linked list stack is shown in Algorithm 6.4. In a linked list stack implementation, the *pop()* operation returns a node in the list. In this algorithm, the variable *x* is a pointer to the node at the top of the stack. Deleting the node is handled outside the algorithm.

Algorithm 6.4. pop() Pop an element from a linked list stack.

Pre-conditions None
Post-conditions Node at the top of the stack is returned and the top position moves to the previous node in the stack.

Algorithm pop() 1. if (!isEmpty()) 2. x = top 3. else 4. print("underflow") 5. top = top.previous 6. return x

Example 5: Pop an element from the linked list stack shown in Figure 8.

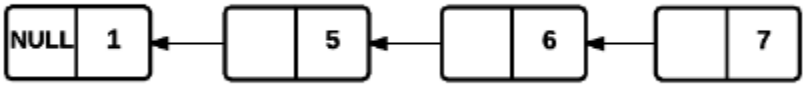


Figure 8. A pop() operation on this linked list will return the node with a key value of 7 and move the top to point to the node with a key value of 6.

pop()

This call to *pop()* returns the node with a key value of 7. When *pop()* is called, the *top* pointer is pointing to the node with a key value of 7 (Figure 9).

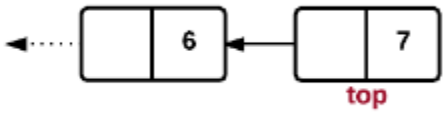


Figure 9. The top of this linked list stack is the node with a key value of 7. The top node will be removed with a pop operation.

• **Line 1**: The conditional fails, since *top* is pointing to a valid node.

• **Line 2**: The variable *x* points to the *top* node (Figure 10).

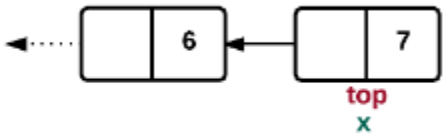


Figure 10. Both x and top point to the same node at the top of the stack.

• **Line 5**: *top* is moved to point to the previous node in the stack (Figure 11).

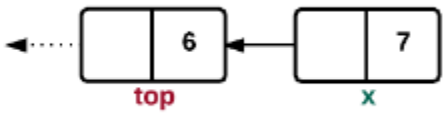


Figure 11. The top pointer is moved to point to the previous node in the list. The x pointer hasn't changed and still points to the top of the stack.

• **Line 6**: The node that *x* points to is returned. Once the node is processed, the memory associated with the node should be freed so that the linked list will have one fewer nodes and there won't be any memory leaks (Figure 12).

Figure 12. The top of the linked list is the 6 node after the node that x points to is popped from the stack.

6.3 Where stacks are used - Computer program execution

During the execution of a computer program, information about currently active subroutines is stored on a call stack. As new subroutines become active, they are added to the stack, and as subroutines complete, they are popped off the stack. For example, consider the simple program in Code 6.1.

Code 6.1. Call stack example.

```
int addNums(int a, int b){
return a + b; }

void callAddNums(){
int c = addNums(5, 6); }

int main(){
callAddNums(); }
```

The program starts at the function *main()*, which is pushed onto the call stack (Figure 13).

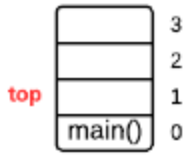


Figure 13. State of the call stack after the main() function is placed on the stack.

Next, *main()* calls *callAddNums()*, which makes that subroutine active and it is pushed onto the call stack above *main()* (Figure 14). The most-recently called routine is at the top of the stack.

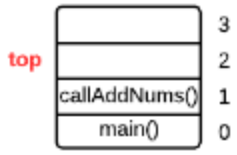


Figure 14. State of the call stack after main() calls callAddNums() and it is pushed onto the call stack. The most-recently called routine is at the top of the stack.

Next, *callAddNums()* calls *addNums()*, which makes that subroutine active and it is pushed onto the stack. At this point, all three subroutines are active because none of them has completed execution (Figure 15).

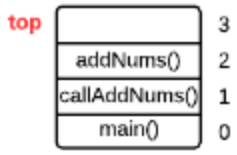


Figure 15. State of the call stack after addNums() routine is pushed onto the stack. There are three active subroutines on the stack, with the most recent one at the top of the stack.

As each of the functions completes, it is popped off the stack. First, *addNums()* completes and is popped off the stack, which returns the stack to the configuration shown in Figure 14. Next, *callAddNums()* completes and it is popped off the stack, leaving only *main()* on the stack (Figure 13). Finally, *main()* completes and is popped off the stack, which completes program execution.

6.4 Complexity of stack operations

All stack operations occur at the top of the stack, which makes their complexity independent of the size of the stack.

Push: O(1)
Pop: O(1)

6.5 Stack exercises

Given an initially empty stack of size 6 and an array A:
A = <10, -1, 3, 2, 3, -1, 4, 5, 6, -1, 2, 3, -1>
What are the contents of the stack after running the following algorithm?

1. Push the values of the array A onto the stack until a -1 value is found. Do not push the -1.
2. Pop the values in the stack until an even value is found or the stack is empty. The even value should be popped.
3. Repeat steps 1 and 2 until the stack is full or the array is read completely.

7 Queues

A queue is a data structure similar to a stack in that it stores a collection of elements and restricts which element can be accessed at any time. However, unlike a stack which works on a LIFO principle, elements in a queue are accessed first-in-first-out (FIFO): the first element added to the queue is the first element removed from the queue, much like the line at the grocery store.

Example 1: Add the words of this classic *Napolean Dynamite* quote,
A liger it's pretty much my favorite animal
to a queue.

Each word in the sentence occupies one position in the queue. Words are added to the queue at the tail position, which moves each time a word is added to the queue (Figure 1).

A liger it's pretty much my favorite animal.

Figure 1. Words stored in a queue. Each word is added to the queue at the tail position and will be removed from the queue at the head position.

Each word in the sentence occupies one position in the queue. Words are added at the *tail* position and removed from the *head* position (Figure 1). The positions of the *tail* and *head* move as elements are added to and removed from the queue.

Removing the words from the queue in Figure 1 generates the following sequence of words:
A liger it's pretty much my favorite animal.

Definitions: When an element is added to a queue, it is "enqueued". Elements are enqueued at the "tail" of the queue.

When an element is removed from a queue, it is "dequeued". Elements are dequeued from the "head" of the queue.

7.1 The queue ADT

In the queue ADT, there are parameters for the head and tail of the queue and the queue size, the data structure to store the queue data, and methods to enqueue and dequeue data. Queue data is typically stored in a data structure such as an array or a linked list. The terminology for interacting with the queue is the same regardless of the data structure used, but the implementation details vary. A queue ADT is shown in ADT 7.1.

ADT 7.1. Queue Queue: 1. private: 2. head 3. tail 4. data 5. queueSize 6. maxQueue 7. isEmpty() 8. isFull() 9. public: 10. Init() 11. enqueue(value) 12. dequeue()

7.2 Enqueue and dequeue queue elements

7.2.1 Array implementation of a queue

In an array implementation of a queue, data elements are stored in an array and the *head* of the queue is the index where the next element will be removed and the *tail* of the queue is the index where the next element will be added. The elements in the array are the contents of the queue.

The simplest, but least efficient, array implementation of a queue involves shifting the elements when the *head* element is dequeued.

Example 2: Dequeue an element from the queue in Figure 2 and shift the remaining elements to fill the space.

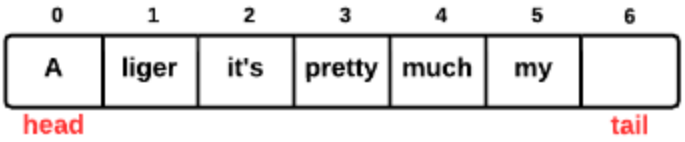


Figure 2. A dequeue operation on this queue removes the element at the head position. Shifting all elements over to fill the space is a costly array shifting algorithm.

A *dequeue()* operation removes the element at the head position, which is an "A". The remaining elements are shifted to fill the space in the array (Figure 3). The position of the *head* doesn't change, but the *tail* shifts by one.

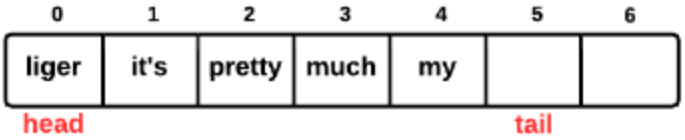


Figure 3. After the head is dequeued, the other elements in the array are shifted by one. The position of the head doesn't change, but the tail position shifts to the left.

Circular array queue Shifting the elements in an array is costly - O(n) in the worst case when the array is full. A much more efficient way to build a queue is to let the *head* and *tail* positions wrap around back to the beginning of the array as elements are enqueued and dequeued.

7.2.2 Enqueue to an array queue

With an array queue, the *enqueue()* operation needs to include a check for if the queue is full. There are multiple ways to check this, and the simplest approach is to keep a count of the number of elements in the queue and the queue size, and only add elements when there's room. The Queue ADT includes variables for *queueSize* and *maxQueue*. Then *queueSize = maxQueue*, the queue is full. The *enqueue()* algorithm is shown in Algorithm 7.1.

Algorithm 7.1. enqueue(value) Add the specified value to the queue at the *tail* position.

Pre-conditions *value* is a valid queue value.

Post-conditions *value* has been added to the queue at the *tail* position, *queue[tail] = value*.
The *tail* position increases by 1.

Algorithm enqueue(value) 1. if (!isFull()) 2. data[tail] = value 3. queueSize++
4. if (tail == maxQueue - 1) 5. tail = 0
6. else 7. tail++
8. else 9. print("queue full")

7.2.3 Dequeue from an array queue

In the *dequeue()* operation, there is a check for if the queue is empty. If not, the element at the *head* position is returned. The tail position is unchanged in the *dequeue()* operation. The *dequeue()* algorithm is shown in Algorithm 7.2.

Algorithm 7.2. dequeue() Remove the queue element at the *head* position.

Pre-conditions None
Post-conditions Value at *data[head]* returned.
head moves by one position in the array.

Algorithm dequeue() 1. if (!isEmpty()) 2. value = data[head]
3. queueSize--
4. if (head == maxQueue - 1) 5. head = 0
6. else 7. head++
8. else 9. print("queue empty") 10. return value

Example 3: Show the state of the queue, Q, for the following set of dequeue() and enqueue() operations using the queue in Figure 4.

• dequeue() • dequeue() • dequeue() • dequeue() • enqueue(6) • enqueue(10) • enqueue(12) • enqueue(2) • enqueue(5) • enqueue(13) • dequeue() • dequeue()

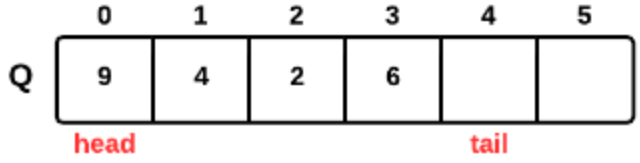


Figure 4. Initial state of the queue for Example 3. The head is Q[0] and the tail is Q[4].

Steps: • The queue has four elements with the *head* at *Q[0]* and the *tail* at *Q[4]*.

• *dequeue()* returns the value at *Q[0]*, which is 9, and moves the *head* to *Q[1]* (Figure 5).

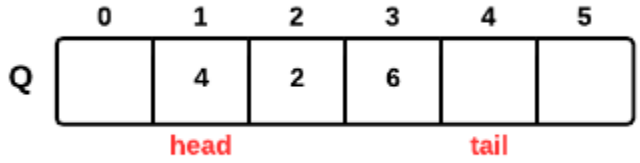


Figure 5. The dequeue() operation returns 9 and moves the head to Q[1]. The tail position is unchanged.

• *dequeue()* returns the value at *Q[1]*, which is 4, and moves the *head* to *Q[2]* (Figure 6).

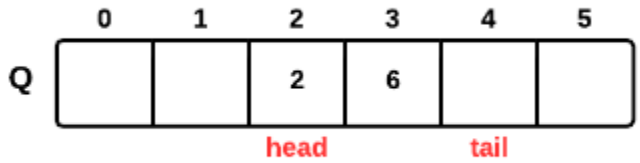


Figure 6. The dequeue() operation returns 4 and moves the head to Q[2].

• *dequeue()* returns the value at *Q[2]*, which is 2, and moves the *head* to *Q[3]* (Figure 7).

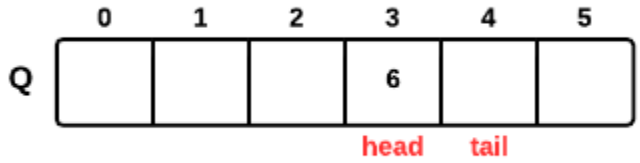


Figure 7. The dequeue() operation returns a 2 and moves the head to Q[3].

• *dequeue()* returns the value at *Q[3]*, which is 6, and moves the *head* to *Q[4]*. The *head* and *tail* are now at the same position and the queue is empty (Figure 8).

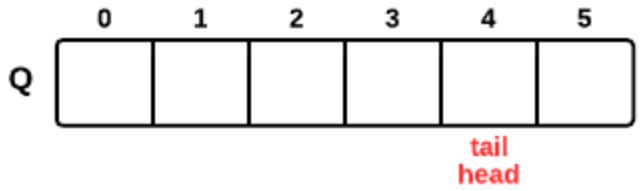


Figure 8. The dequeue() operation returns the 6. The head and tail are now both at Q[4] and the queue is empty.

• *enqueue(6)* adds a 6 to the queue at the *tail* position *Q[4]* and increments the *tail* to *Q[5]*. The *head* is still at *Q[4]* (Figure 9).

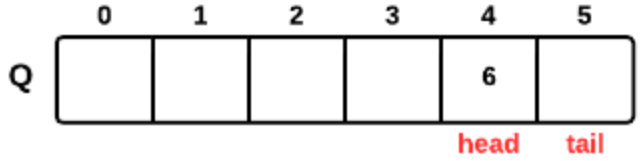


Figure 9. The enqueue(6) operation writes a 4 to the tail position Q[4] and increments the tail to Q[5]. The head is still at Q[4].

• *enqueue(10)* adds a 10 to the queue at the *tail* position *Q[5]* and increments the *tail*. Since the *tail* is pointing to the last position in the queue, it wraps around back to the beginning. The *tail* is now *Q[0]* (Figure 10).

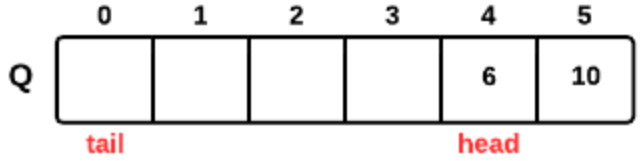


Figure 10. The enqueue(10) operation writes a 10 to the tail position at Q[5] and increments the tail back to the beginning of the queue at Q[0].

• *enqueue(12)* adds a 12 to the queue at the *tail* position *Q[0]* and moves the *tail* to *Q[1]* (Figure 11).

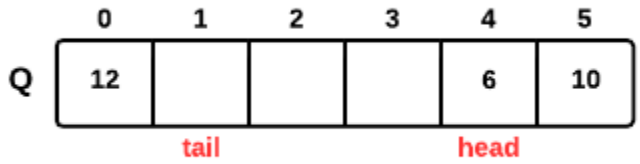


Figure 11. The enqueue(12) operation adds a 12 to the queue at Q[0] and increments the tail to Q[1].

• *enqueue(2)* adds a 2 to the queue at the *tail* position *Q[1]* and moves the *tail* to *Q[2]* (Figure 12).

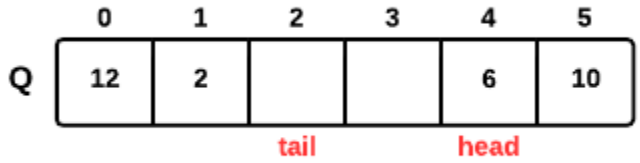


Figure 12. The enqueue(2) operation writes a 2 to the queue at Q[1] and increments the tail to Q[2].

• *enqueue(5)* adds a 5 to the queue at the *tail* position *Q[2]* and moves the *tail* to *Q[3]* (Figure 13).

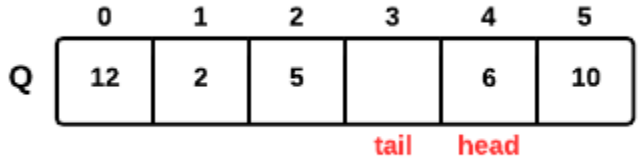


Figure 13. The enqueue(5) operation writes a 5 to the queue at Q[2] and increments the tail to Q[3].

• *enqueue(13)* adds a 13 to the queue at the *tail* position *Q[3]* and moves the *tail* to *Q[4]*. The *head* and *tail* are now both at the same position and the queue is full (Figure 14).

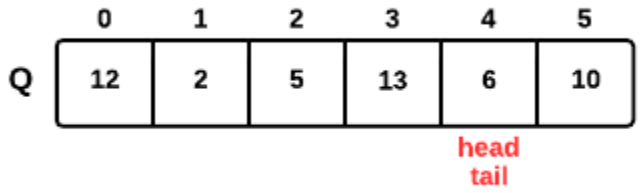


Figure 14. The enqueue(13) operation writes a 13 to the queue and increments the tail to Q[4]. The head and tail are at the same position and the queue is full.

If another element were to be enqueued now, the value currently at *Q[4]* would be overwritten. For example, calling *enqueue(1)* would write a 1 to *Q[4]* and overwrite the 6 that is currently stored there. There needs to be a *dequeue()* operation first before any more data can be added to the queue.

• *dequeue()* returns the value at *Q[4],* which is 6, and moves the *head* to *Q[5]* (Figure 15).

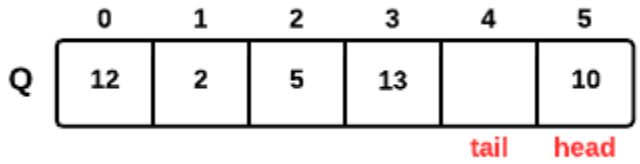


Figure 15. The dequeue operation returns a 6 and increments the head to Q[5].

• *dequeue()* returns the value at *Q[5],* which is 10, and increments the *head*. Since the *head* is currently at the last position in the queue, it wraps around back to the beginning of the queue *Q[0]* (Figure 16).

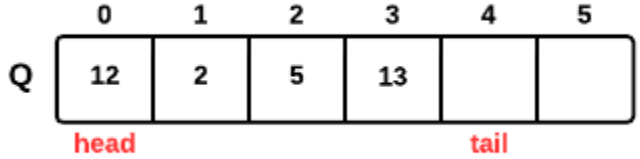


Figure 16. The dequeue() operation returns a 10 and moves the head to Q[0].

Features of a circular queue: • Both the *tail* and the *head* can wrap around from the last position in the array back to the beginning. In the previous examples, when the *head* or *tail* reached *Q[5]*, they were reset to *Q[0]*.

• The condition where *head = tail* can mean that the queue is empty or full, which needs to be resolved in the *enqueue()* and *dequeue()* algorithms. Calling *enqueue()* when the queue is full can result in overwriting data if the algorithm doesn't check for a full queue. Calling *dequeue()* when the queue is empty can result in an unexpected return value if the algorithm doesn't check for an empty queue.

• The circular queue is more computationally efficient than a queue that uses array shifting, but it is more complicated to implement.

7.2.4 Linked list implementation of a queue

In a linked list implementation of a queue, the *head* and *tail* of the queue are nodes in the list. The size of the queue can change dynamically as elements are added and removed. To implement a linked list queue with a fixed size, variables for the current queue size and the maximum queue size could be included in the ADT. In the linked list queue shown in Figure 17, each node has a pointer to the next node in the list. The *tail* node is the last node and it has a *next* pointer of NULL.

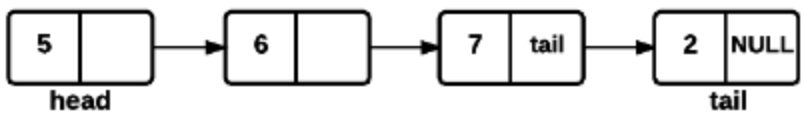


Figure 17. Queue implemented with a linked list. The head is the first node in the list. Each node has a pointer to the next node in the list. The tail node, as the last node in the list, has a next pointer that points to NULL.

In another implementation of a linked list queue, the queue is a circular buffer where the *tail* node points to the *head* node as the next node in the list. Figure 18 shows an example of a circular, linked list queue.

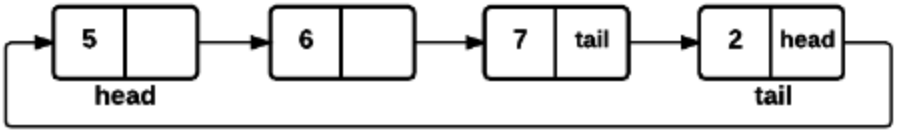


Figure 18. Example of a circular queue implemented with a linked list. The next pointer for the tail node points to the head node in the list.

7.2.5 Enqueue to a linked list queue

The algorithm to enqueue a node to a linked list queue, shown in Algorithm 7.3, takes the value to enqueue and creates a node with that queue value. The new node is added at the *tail* position.

Algorithm 7.3. enqueue(value) Add a node to a queue implemented with a linked list.

Pre-conditions *value* is a valid linked list node value.

Post-conditions A new node with the specified key value is added to the queue.
tail points to the newly added node.

Algorithm enqueue(value) 1. node.key = value 2. node.next = NULL
3. if tail != NULL
4. tail.next = node 5. tail = node 6. else 7. tail = node 8. head = tail

7.2.6 Dequeue from a linked list queue

The dequeue algorithm, shown in Algorithm 7.4. dequeue(), returns the head of a linked list queue, and moves the head of the queue to the next position in the list.

Algorithm 7.4. dequeue() Return the head of the linked list queue.

Pre-conditions None
Post-conditions Head of the queue returned.
head position moved to the next node in the list.

Algorithm dequeue() 1. if head != NULL
2. node = head 3. head = head.next 4. else 5. print("queue empty") 6. tail = head 7. return node

Example 4: Show the state of the queue for the following set of dequeue and enqueue operations using the linked list queue in Figure 19 using the enqueue() and dequeue() algorithms in Algorithm 7.3 and Algorithm 7.4, respectively.

• dequeue() • dequeue() • dequeue() • dequeue() • enqueue(7) • enqueue(10) • enqueue(12)

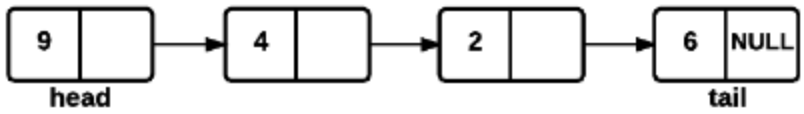


Figure 19. Initial state of linked list queue for Example 4.

Steps: • *dequeue()* returns the *head* node, which contains a value of 9. The *head* is moved to *head.next*, which is the node that contains a value of 4 (Figure 20). The *tail* is unchanged.

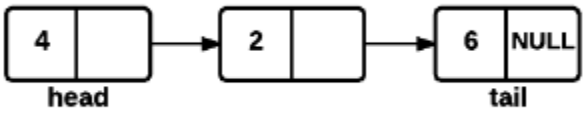


Figure 20. Contents of the queue after the head node dequeued and the head moves to the node that contains a value of 4.

• *dequeue()* returns the *head* node, which contains a value of 4. The *head* moves to the next node, which contains a value of 2, and the *tail* is unchanged. The state of the queue after the *dequeue()* operation is shown in Figure 21.

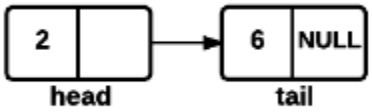


Figure 21. Contents of the queue after the head node is dequeued and the head position moves to the node with a value of 2.

• *dequeue()* returns the *head* node, and the *head* position moves to the next node, which is the same as the *tail* node. The state of the queue after the dequeue() operation is shown in Figure 22.

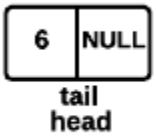


Figure 22. Contents of the linked list after the dequeue operation. The head and tail now point to the same node in the list.

• *dequeue()* returns the only node in the queue and sets the *head* to NULL, since *head.next = NULL*. Another *dequeue()* here would result in the message *"queue empty"* being displayed, and the *tail* being set to NULL too.

• *enqueue(7)* adds a node with a value of 7 to the queue at the *tail* position. The *tail* is NULL, which causes Lines 7-8

in the *enqueue()* algorithm to execute. The *head* and *tail* are both set to the new node, as shown in Figure 23.

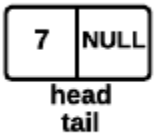


Figure 23. The enqueue() operation adds a new node to the empty list. The head and tail point to the same node in the list.

• *enqueue(10)* adds a node with a value of 10 to the queue at the *tail* position. The *head* and *tail* now point to different nodes, as shown in Figure 24.

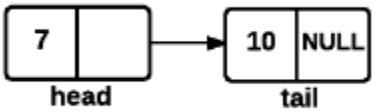


Figure 24. There are now two nodes in the queue after the enqueue() operation. The head and tail point to different nodes.

• *enqueue(12)* adds a node with a value of 12 to the queue at the *tail* position. There are now three nodes in the queue, and the *tail* is set to the new node (Figure 25).

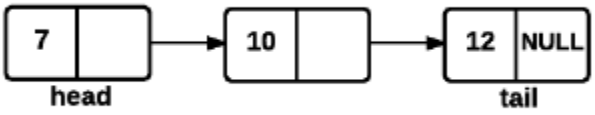


Figure 25. After another enqueue() operation, there are three nodes in the queue. The tail position is set to the new node.

7.3 Queue Exercises

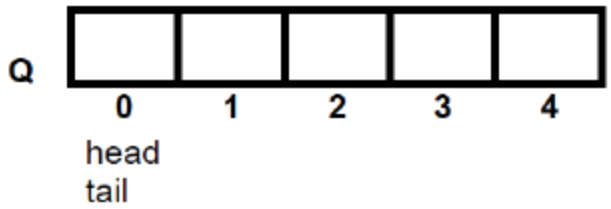


Figure 26. Queue of size 5. The head and tail are both pointing to Q[0].

1. Using the queue shown in Figure 26, show the output for each dequeue operation in the sequence:

   Enqueue(13) Dequeue( ) Enqueue(27) Enqueue(8) Enqueue(11) Dequeue( ) Dequeue( ) Enqueue(27) Dequeue( ) Enqueue(16)
2. Draw the state of the queue, including the position of the head and tail, after each operation in the sequence in Question 1.

8 Binary Trees

Imagine a simplistic transportation network, such as the one shown in Figure 1, that starts in Denver, and from Denver, there is a road that goes east to Chicago and a road that goes west to Las Vegas. From Chicago, there is a road east to Boston and a road west to St. Paul, and from Las Vegas, there is travel east to Phoenix and west to San Diego. In the scenario just described, from each city, there are two choices for which city to visit next.

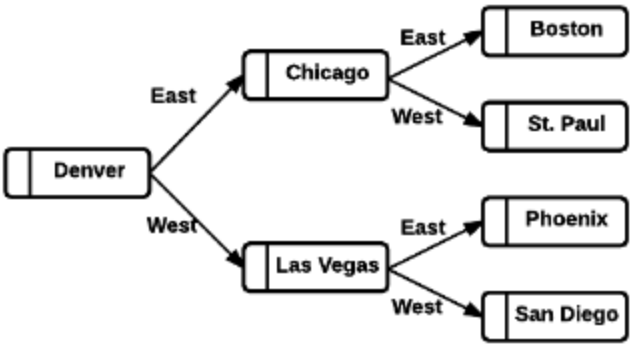


Figure 1. Example of a binary tree that represents travel between cities. Starting from the root city, Denver, there are two choices for which city to visit next, Chicago and Las Vegas.

Having two options (at most) for going to a next node from the current node is a feature of a structure called a binary tree. The city network example in Figure 1 can be viewed as a tree by putting the starting city, also known as the root, at the top of the tree, and putting the two possible destination cities as the children. The top three nodes in the tree would feature Denver as the root, Las Vegas as the left child, and Chicago as the right child (Figure 2).

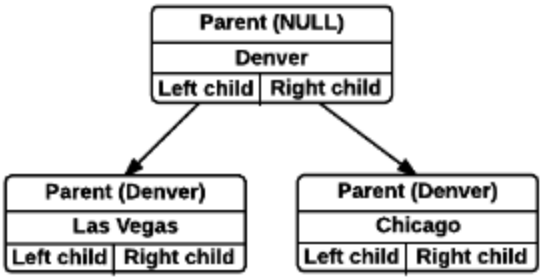


Figure 2. Example nodes in a binary tree with properties for the parent, key, left child, and right child. The root of the tree is Denver, and Denver has a left child, Las Vegas, and a right child, Chicago. The parent of Las Vegas and Chicago is Denver. The key for each node is the name of the city.

8.1 Properties of binary trees

Binary trees are similar to linked lists in that the nodes in the tree can be created dynamically and then linked together to create a structure that can be easily modified to support dynamic data. The *next* and *previous* pointers of a doubly linked list are replaced with *parent* and *left* and *right child* pointers in binary trees to make it possible to represent a hierarchical structure in a data set, which is one advantage that trees have over linked lists. Trees can also be searched and modified with minimal computational effort. These advantages mean that binary trees are extremely useful and frequently used over other data structures.

Pointers in a doubly linked list:
- next
- previous

Pointers in a binary tree:
- parent
- left child
- right child

All nodes in the tree can be a parent or a child to other nodes (except for the *root*). There are general properties that all nodes exhibit, as well as properties that nodes exhibit as parent nodes and as the *root* node of the tree.

8.1.1 Parent node properties

- Each node in the tree has a *parent*. In Figure 2, the *parent* of both Las Vegas and Chicago is Denver.
- Each node in the tree is a *parent* node for at most two children, a left and a right child.

8.1.2 Root node properties

• The topmost node in the tree is called the *root*.
• The *parent* of the *root* is NULL.

8.1.3 Node properties

• Each node in the tree has a *key* that identifies it. In the city example in Figure 2, the *key* is the city name.
• If a node doesn’t have a left child, then its *left child* property is NULL.
• If a node doesn’t have a right child, then its *right child* property is NULL.
• If a node doesn’t have a left or a right child, then it is a *leaf* node.

8.2 Trees and Sub-trees

An interesting feature of any binary tree is that it is defined in terms of the smaller sub-trees within it. This is called self-similarity and it's computationally significant because it means there are elegant ways to search the tree by examining smaller and smaller sub-trees. For example, consider the binary tree *T* in Figure 3. The tree has a *root* node, which has a left and a right child.

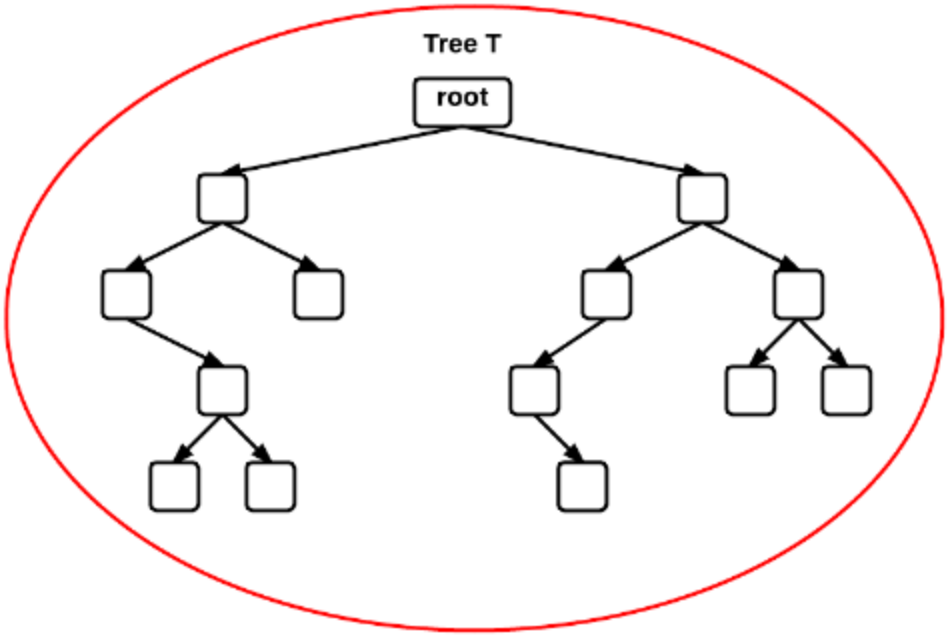


Figure 3. Example of a binary tree. The tree has a root, and from the root, there is a left and right sub-tree.

The tree can also be considered by identifying the left or right child of the *root* as the new *root* of a sub-tree *T'*. For that new sub-tree *T'*, that *root* also has a left and a right child. This relationship is shown in Figure 4.

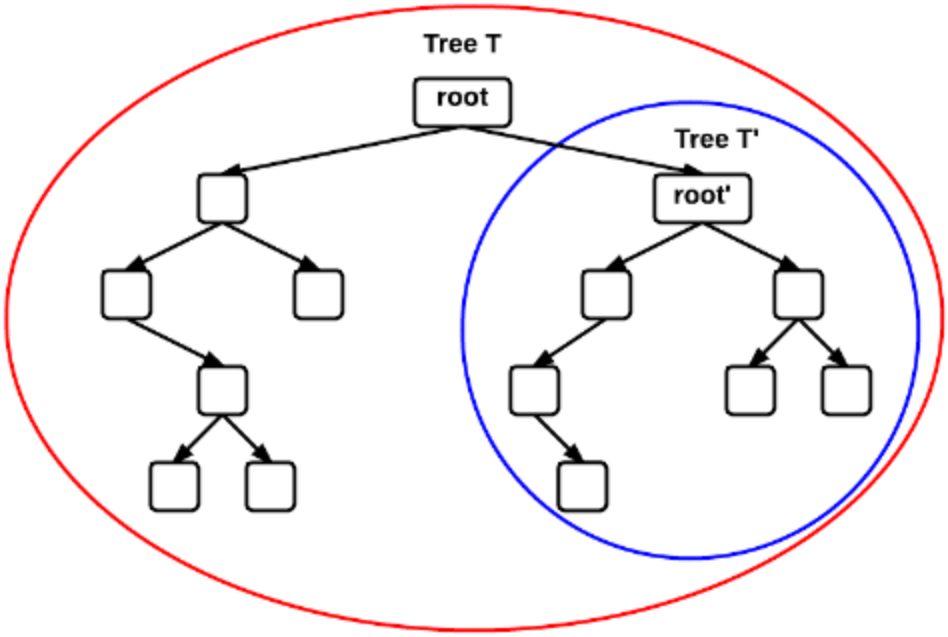


Figure 4. Binary tree T and a sub-tree T'. The root of T' is the right child of the root of T.

The pattern continues down to the smallest sub-tree in *T*, which contains a *root* and two children, but no additional nodes from the children (Figure 5).

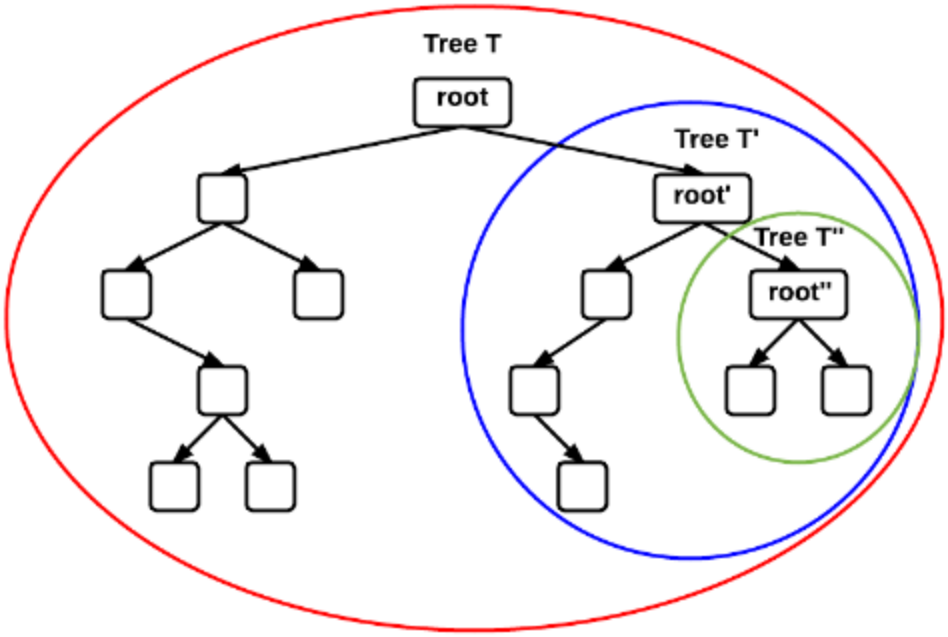


Figure 5. The tree T can contain multiple nested sub-trees, each with the same basic structure as T. Each of the sub-trees has a root with 0, 1, or 2 child nodes.

9 Binary Search Trees

A binary search tree (BST) is a special case of a binary tree where the data in the tree is ordered. For any node in the tree, the nodes in the left sub-tree of that node all have a value less than the node value, and the nodes in the right sub-tree of that node all have a value greater than or equal to the node value. For example, consider the BST in Figure 1, where the *key* value of each node is an integer.

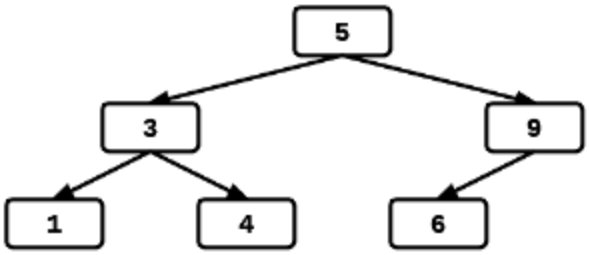


Figure 1. Example of a binary search tree (BST). All nodes left of any node have a value less than that node and all nodes right of any node have a value greater than that node.

The root of the tree has a *key* value of 5. All nodes left of the root have a *key* value less than 5 and all values to the right of the root have a *key* value greater than or equal to 5. These properties hold for all other nodes in the tree. For example, all nodes to the left of the 3 have a *key* value less than 3 and all nodes to the right of the 3 have a value greater than or equal to 3, but less than 5.

A **binary search tree** is defined as follows:
Let *x* and *y* be nodes in a binary search tree. If *y* is in the left sub-tree of *x*, then *y.key* < *x.key*. If *y* is in the right sub-tree of *x*, then *y.key* >= *x.key*.

Example 1: Build a binary search tree from the following integer keys: < 4, 2, 6, 9, 1, 3 >.

The integers are added to the tree in the order they are observed, i.e. 4 is added first and 3 is added last.

Steps: 1. Add 4 as the root.

2. Evaluate the next value in the list, 2. Since 2 < 4, go left of the root and add the 2 to the tree as the left child of the 4.

3. Evaluate the next value in the list, 6. Since 6 > 4, go right of the root and add the 6 to the tree as the right child of the 4. The partial tree containing the first three nodes added is shown in Figure 2.

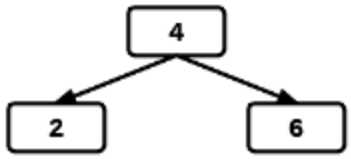


Figure 2. BST after the first three nodes are added. The root is 4, the left child of the root is 2 and the right child of the root is 6.

4. Evaluate the next value in the list, 9. Since 9 > 4, go right of the root and compare the 9 to the 6. Since 9 > 6, go right of the 6 and add the 9 as right child of the 6.

5. Evaluate the next value in the list, 1. Since 1 < 4, go left of the root and compare the 1 to the 2. Since 1 < 2, go left of the 2 and add the 1 as the left child of the 2. The partial tree containing the first five nodes is shown in Figure 3.

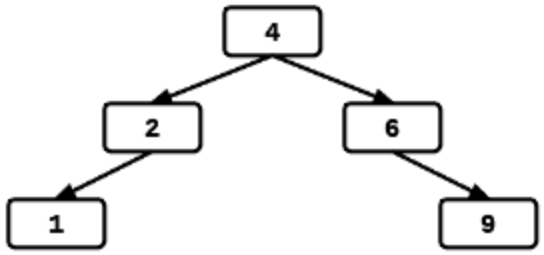


Figure 3. BST after the first five nodes are added to the tree.

6. Evaluate the next value in the list, 3. Since 3 < 4, go left of the root and compare the 3 to the 2. Since 3 > 2, go right of the 2 and add the 3 as the right child of the 2. The final tree is shown in Figure 4.

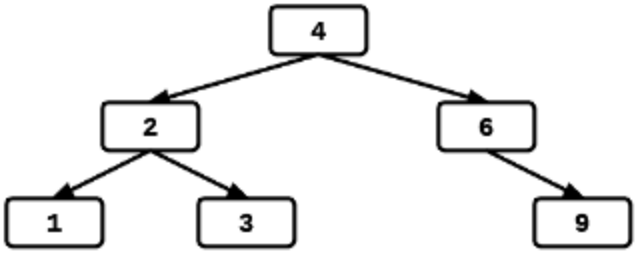


Figure 4. Binary search tree built from the input sequence: < 4, 2, 6, 9, 1, 3 >. Each integer is added to the tree in order, i.e. the 4 is added first and the 3 is added last.

Example 2: Build a binary search tree from the following string keys: < DEN, LA, CHI, VEGAS, SD, DET, NY >.

A string is compared to another string through the ASCII values of the individual characters in the string. A string is less than another string if it appears first alphabetically.

Steps: 1. Add DEN as the root.

2. Evaluate LA by comparing the ASCII value of *D* to the ASCII value of *L*, the first letters in DEN and LA. *L* has an ASCII value of 76 and *D* has an ASCII value of 68, which makes DEN < LA. Add LA as the right child of DEN.

3. Evaluate CHI by comparing the ASCII value of *D* to the ASCII value of *C*. *C* is less than *D*. Add CHI as the left child of DEN. The BST with the first three nodes is shown in Figure 5.

Figure 5. BST with the first three nodes added to the tree. DEN is the root of the tree. The left child of the root is CHI and the right child of the root is LA.

4. Evaluate VEGAS by comparing the ASCII value of *D* to the ASCII value of *V*. *V* is greater than *D*, so go to the right child of DEN and compare VEGAS to LA. Compare the ASCII value of *V* to the ASCII value of *L* in LA. *V* is greater than *L*. Add VEGAS as the right child of LA.

5. Evaluate SD by comparing the ASCII value of *D* in DEN to the ASCII value of *S* in SD. *S* is greater than *D*, so go to the right child of DEN and compare SD to VEGAS. *S* is less than *V*. Add SD as the left child of VEGAS. The BST with the first five nodes added is shown in Figure 6.

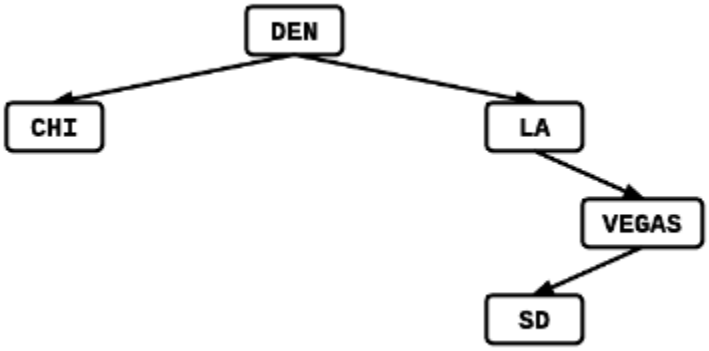


Figure 6. BST with the first five nodes added to the tree.

6. Evaluate DET by comparing the *D* in DEN to the *D* in DET. Since they are equal, evaluate the second letter in DEN and DET. They are also equal, so move to the third letter and compare *T* and *N*. *T* is greater than *N*, so go to the right child of DEN and compare DET to LA. *D* is less than *L*. Add DET as the left child of LA (Figure 7).

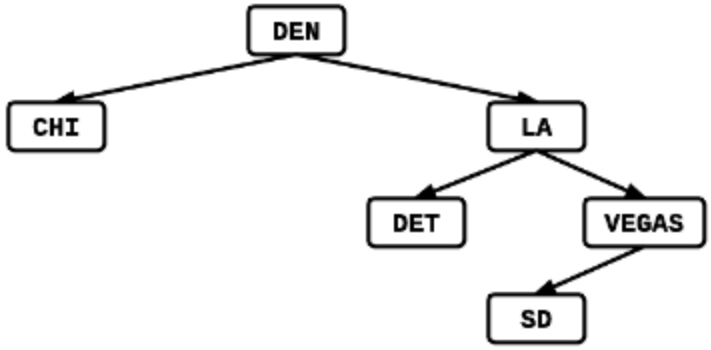


Figure 7. BST after DET added as the left child of LA.

7. Evaluate NY, the final key in the sequence, by comparing the ASCII value of *D* in DEN to the ASCII value of *N* in NY. *N* is greater than *D*. Go right. Compare NY to LA by comparing the ASCII value of *N* to the ASCII value of *L*. *N* is greater than *L*. Go to the right child of LA and compare NY to VEGAS. The ASCII value of *N* is less than the ASCII value of *V*; go to the left child of VEGAS and compare NY to SD. *N* is less than *S*. Add NY as left child of SD. The final tree is shown in Figure 8.

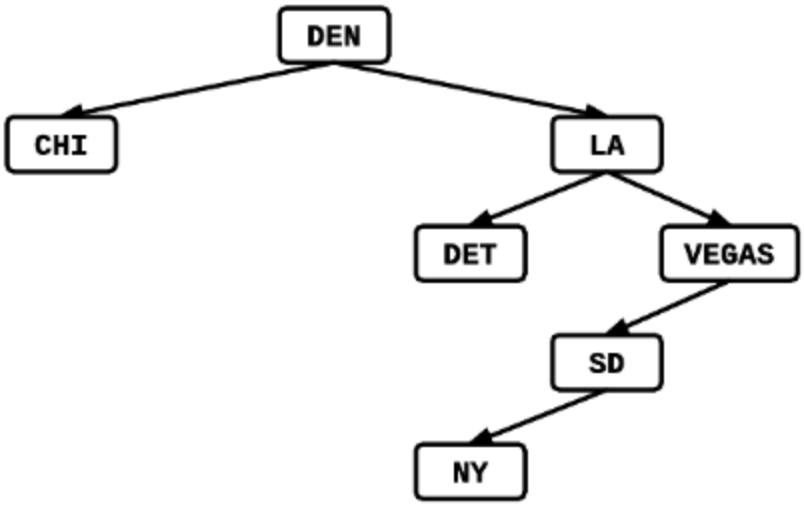


Figure 8. Binary search tree built from the input sequence of cities: <DEN, LA, CHI, VEGAS, SD, DET, NY>.

9.1 Binary search tree ADT

In a binary search tree ADT, shown in ADT 9.1, the data is stored in a tree that is accessed through the root of the tree. The *root* is stored as a private variable, and there are public methods to initialize the tree, insert and delete nodes, traverse the tree, and search the tree. A private search method is also included to support the recursive search functionality in the tree.

ADT 9.1. Binary Search Tree BinarySearchTree: 1. private: 2. root 3. searchRecursive(node, value) 4. public: 5. Init() 6. insert (value) 7. search(value) 8. traverseAndPrint() 9. delete(value) 10. deleteTree()

9.1.1 C++ implementation of a binary tree node

A binary tree node in C/C++ can be built with a **struct**, just like a node in a linked list, where the members of the **struct** include the *key*, a pointer to the *parent* node, pointers to the *leftChild* and *rightChild* nodes, and any additional data that the program needs to store to operate successfully.

```
struct node{
  int key node *parent node *leftChild node *rightChild }
```

9.2 Searching a BST

The BST ordering generates a structure, whereby, from any given node, a search operation can identify a section of the tree that might contain the search value and eliminate the rest of the tree from consideration. For example, if the search value is less than the value of a given node, then all nodes to the right of that node don’t need to be evaluated. This ordering prunes the search space by removing branches that won't contain the search value.

The BST search can be performed recursively or iteratively. In the recursive version, the *search()* algorithm, shown in Algorithm 9.1, takes the value to search for as an argument and calls another algorithm *searchRecursive()* (shown in Algorithm 9.2), to recursively search the tree starting at the root. The recursive algorithm takes the search value and the node to evaluate as arguments. The search returns a pointer to the node when the value is found, or returns NULL if the value doesn’t exist in the tree.

Algorithm 9.1. search(searchKey) Returns a pointer to the node with a *key* that matches *searchKey*. The search calls *searchRecursive()* to evaluate individual nodes.

Pre-conditions *searchKey* is a valid search parameter that is the same type as the node *key*.

Post-conditions Returns a pointer to the node with a *key* that matches the *searchKey* or NULL if the *key* is not found.

Algorithm search(searchKey) 1. return searchRecursive(root, searchKey)

Algorithm 9.2. searchRecursive(node, searchKey) Returns a pointer to the node with a *key* that matches *searchKey*.

Pre-conditions *searchKey* is a valid search parameter that is the same type as the node *key*.
node is a valid node being evaluated in a BST or NULL.

Post-conditions Returns pointer to the node with a *key* that matches the search value or NULL if the *key* does not exist in the tree.

Algorithm searchRecursive(node, searchKey) 1. if (node!=NULL) 2. if (node.key == searchKey) 3. return node 4. else if (node.key > searchKey) 5. return searchRecursive(node.left, searchKey) 6. else 7. return searchRecursive(node.right, searchKey) 8. else 9. return NULL

In the recursive *searchRecursive()* algorithm, *node* is the node in the BST to evaluate and *value* is the value to search for. On Lines 5 and 7 of the algorithm, *searchRecursive()* is called recursively on the left or right child of a node to evaluate either the left or right sub-tree of the node. If *searchKey* is less than *node.key*, the left branch is searched and if *searchKey* is greater than or equal to the *node.key*, then the right branch is explored. If the algorithm doesn't find the specified value in the tree, the algorithm returns NULL.

Searching a BST does not need to be done recursively. In the non-recursive algorithm, a **while** loop is used to check for when the bottom of the tree has been reached or the value has been found. An iterative search algorithm is shown in Algorithm 9.3.

Algorithm 9.3. searchIterative(searchKey) Returns a pointer to the node where the *key* matches the search *searchKey*.

Pre-conditions *searchKey* is a valid search value that is the same type as the node *key*.

Post-conditions Returns a pointer to the node where the search value matches the *key* or NULL if the *key* is not found in the tree.

Algorithm searchIterative(searchKey) 1. node = root 2. while(node != NULL) 3. if (node.key > searchKey) 4. node = node.left 5. else if(node.key < searchKey) 6. node = node.right 7. else 8. return node 9. return NULL

In the *searchIterative()* algorithm, the **while** loop on Line 1 checks that node is not NULL, which will be true when the tree is not empty and the bottom of the tree has not been reached. This condition will fail when the search has reached a left or right child that is set to NULL at the bottom of the tree. If this happens, then *searchKey* doesn't exist in the tree and the algorithm returns NULL. Otherwise, *node* is updated to move through the tree by pointing to its left or right child depending on *node.key* and *searchKey*.

9.2.1 Inserting a node into a BST

Inserting a node into a BST involves first, searching for the correct placement of the node, and then, modifying the tree to add the node. The *insert()* algorithm is shown in Algorithm 9.4.

Algorithm 9.4. insert(value) Inserts a node with the specified *value* into a BST at the appropriate position.

Pre-conditions *value* is a valid node value.

Post-conditions Memory for the node is allocated and the BST has been modified correctly to include the new node.

Algorithm insert(value) 1. tmp = root 2. node.key = value 3. node.parent = NULL
4. node.leftChild = NULL
5. node.rightChild = NULL
6. while(tmp != NULL) 7. parent = tmp 8. if(node.key < tmp.key) 9. tmp = tmp.leftChild 10. else 11. tmp = tmp.rightChild 12. if (parent == NULL) 13. root = node 14. else if(node.key < parent.key) 15. parent.leftChild = node 16. node.parent = parent 17. else 18. parent.rightChild = node 19. node.parent = parent

In the *insert()* algorithm, the node is created with *value* as the key value and the *parent*, *leftChild*, and *rightChild* pointers initialized to NULL. The **while** loop on Lines 6-11 identifies the correct placement for the new node by searching for a node with a NULL pointer its left or right child. At the end of the **while** loop, the value of *tmp* will be NULL because it will be pointing to the child. Lines 12-19 of the algorithm add the node to the tree as either the *root* if the tree is empty, or the left or right child. The *parent* value of the new node is also set.

Example 3: Insert a 3 into the BST shown in Figure 9.

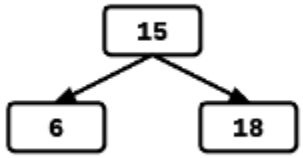


Figure 9. Add a node with a key value of 3 to this BST for Example 3.

Steps:

1. On Line 1, set *tmp* to point to the *root* node, which points *tmp* to the node with the *key* value of 15.

2. On Line 8, compare the 3 to the 15, and since 3 < 15, go left to evaluate the nodes in the left sub-tree of the 15. Set

tmp to the left child of the *root* on Line 9, which points *tmp* to the 6.

3. Evaluate the **while** loop condition again on Line 6; *tmp* is not NULL, since it's pointing to the 6.

4. On Line 8, compare the 3 to the 6, and since 3 < 6 and go left again. The 6 doesn't have a left child, therefore, on Line 9, *tmp* is set to NULL.

5. Evaluate the **while** loop condition again, which fails because the value of *tmp* is NULL. At this point, *parent* is the 6. The new node will be added as either a left or right child of the 6.

6. On Line 12, check if the tree is empty, which is true if *parent* is NULL.

7. On Line 14, the conditional is true, since 3 < 6. This means that the 3 should be added as the left child of the 6, which is accomplished by setting the 6's *leftChild* property to point to the 3 on Line 15. The *parent* property of the new node is updated on Line 16 to point to the 6. The new tree is shown in Figure 10.

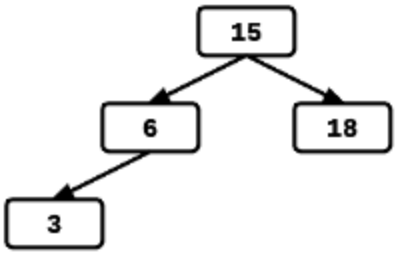


Figure 10. BST after the 3 is added as the left child of the 6.

9.2.2 Deleting a node from a BST

When a node is deleted from the tree, the node may need to be replaced with another node in the tree. The replacement

node needs to be selected such that the BST properties are preserved.

There are three cases to consider when deleting a node. Exactly one of the following conditions is true about the deleted node: 1. The node has no children.
2. The node has one child.
3. The node has two children.

Figure 11 shows a BST with examples of nodes with 0, 1, or 2 children. The nodes with values of 3, 9, 17, and 20 have no children. The node with a value of 7 has one child, and the nodes with values of 6, 15, and 18 have two children. The *delete()* algorithm that handles all three cases is shown in Algorithm 9.5. For brevity, only the case where the deleted node is its parent's left child is shown.

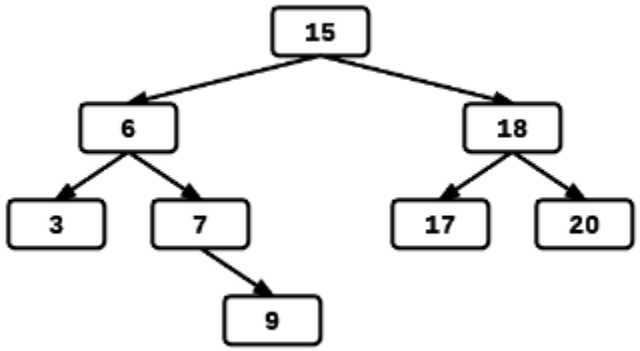


Figure 11. BST with nodes that have no children (3, 9, 17, 20), one child (7), and two children (6, 15, 18).

Algorithm 9.5. delete(value) Deletes the node where the value matches the node key value.

Pre-conditions *value* is a valid search value whose type matches the node key type.
search() algorithm exists to identify the node to delete.
treeMinimum() algorithm exists to identify the minimum value in a sub-tree, which will be the replacement node for a

deleted node with two children. (Algorithm 9.5)
Post-conditions Node with specified *key* value is deleted from the tree.
parent, *left child*, and *right child* pointers for the deleted node and neighboring nodes are reset accordingly.

Algorithm *(Note: this is not the complete delete() algorithm. For the one- and two-children cases, only the case where the deleted node is the left child of its parent is shown. Additional cases are needed to handle when the deleted node is the right child.)*
delete(value) 1. node = search(value) 2. if(node != root) 3. if(node.leftChild == NULL and node.rightChild == NULL) //no children 4. node.parent.leftChild = NULL
5. else if(node.leftChild != NULL and node.rightChild != NULL) //two children 6. min = treeMinimum(node.rightChild) 7. if (min == node.rightChild) 8. node.parent.leftChild = min 9. min.parent = node.parent 10. min.leftChild = node.leftChild 11. min.leftChild.parent = min 10. else 11. min.parent.leftChild = min.rightChild 12. min.rightChild.parent = min.parent 13. min.parent = node.parent 14. node.parent.leftChild = min 15. min.leftChild = node.leftChild 16. min.rightChild = node.rightChild 17. node.rightChild.parent = min 18. node.leftChild.parent = min 19. else //one child 20. x = node.leftChild 21. node.parent.leftChild = x 22. x.parent = node.parent 23. else 24. //repeat cases of 0, 1, or 2 children 25. //replacement node is the new root 26. //parent of replacement is NULL
27. delete node
Node has no children In the tree shown in Figure 13, nodes 3, 9, 17, 20 don’t have any children. To delete a node with no children: • Reset the appropriate *leftChild* or *rightChild* pointer for the *parent* of the deleted node to NULL.
• Free the memory to delete the node.

Example 4: Delete the 3 from the BST in Figure 11.

The 3 is the *left child* of the 6.

Steps:
1. Set the *left child* pointer for the 6 to NULL.

2. Delete the 3 node.

Node has one child In the BST shown in Figure 13, the 7 has only one child. To delete a node with one child: • Update the node's parent to point to the node's child.
• Delete the node.

Example 5: Delete the 7 from the BST in Figure 11.

The 7 is the *right child* of the 6.

Steps:
1. Set the *right child* pointer of the 6 to point to the 9.
2. Delete the 7 node.

The node has two children In the tree shown in Figure 13, the 6, 15, and 18 have two children. To delete a node with two children, a node in its right sub-tree that doesn't have a left child should replace the deleted node, i.e. the minimum value in the right sub-tree. Figure 14 shows the right sub-tree of the 6. The replacement for the 6 in a delete operation is the 7, since it is the first node found that doesn't have a left child.

Example 6: Delete the 6 from the BST in Figure 12.

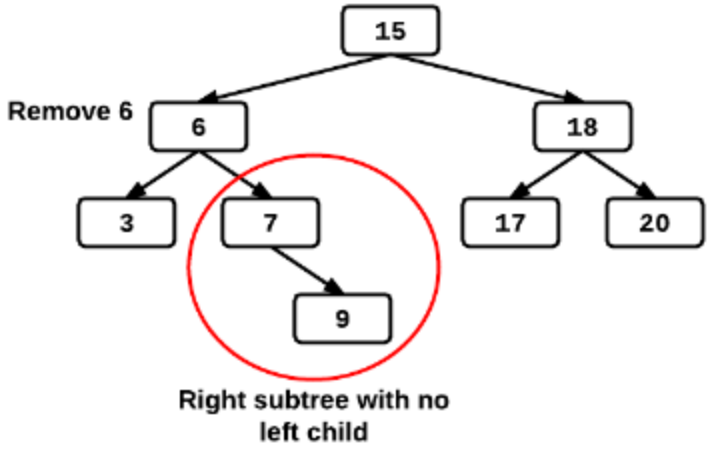


Figure 12. The right sub-tree of the 6 includes the 7 and the 9. The first node encountered that doesn't have a left child is the 7, which makes it the minimum value in the sub-tree and the replacement for the 6.

In this example, the replacement for the 6 is its *rightChild*, the 7. Reset the *parent* and *leftChild* properties for the nodes surrounding the 6, including the 3, 7 and 15. The rest of the right sub-tree is unmodified.

Steps:

1. The *parent* property of the 3 is updated to point to the 7.

2. The *leftChild* property of the 15 is updated to point to the 7.

3. The *parent* property of the 7 is updated to point to the 15.

4. Delete the 6 node. The new BST is shown in Figure 13.

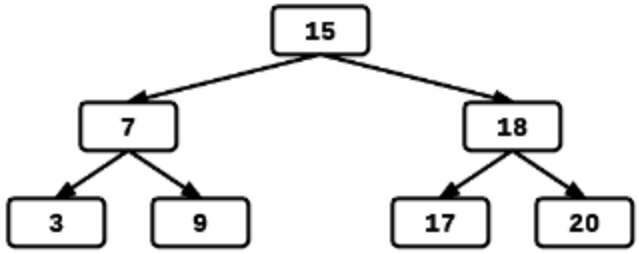


Figure 13. BST after the 6 is deleted and replaced by the 7. Properties of the 15, 3, and 7 were updated to restructure the tree and delete the 6.

In Example 6, the replacement node was the right child of the node to delete. There is another case where the replacement node is the minimum value in the right sub-tree, but it is not the right child of the deleted node.

Example 7: Remove the 6 from the BST in Figure 14.

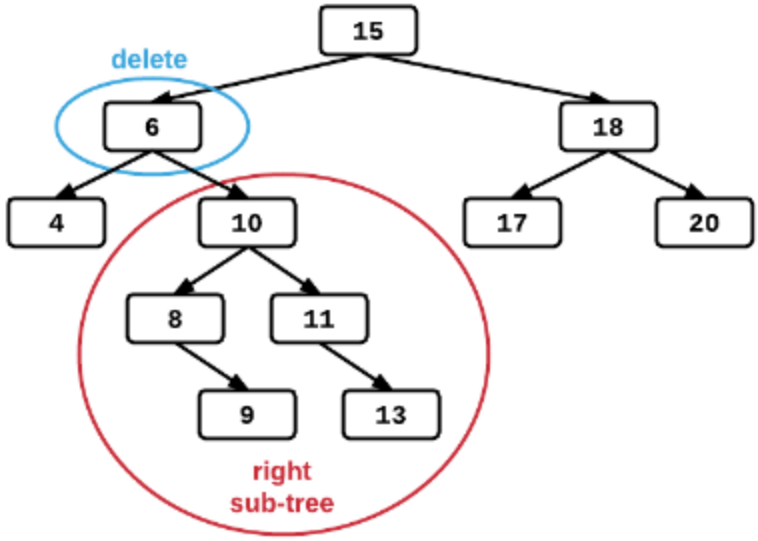


Figure 14. Delete the 6 from this tree and replace it with the minimum value in its right sub-tree. The minimum value is found by traversing left down the right sub-tree until a node without a left child is found.

Identify the replacement for the 6. The replacement is in its right sub-tree, but it not its right child. The replacement node has to be the *minimum* value in the right sub-tree to preserve the BST properties. To find the minimum, start at

the right child and traverse left until the node doesn’t have a left child. In this example, the right child is the 10, which has a left child. The 10 can’t be the minimum in the sub-tree and therefore, is not the replacement node. Next, evaluate the left child of the 10, which is the 8. The 8 doesn't have a left child, which makes it the minimum value in the sub-tree. Replace the 6 with the 8 by updating the *parent*, *leftChild*, and *rightChild* properties for the surrounding nodes of the 6 and the 8.

Steps:
1. Update the *leftChild* property of the 10 to point to the 9.

Line 11: *min.parent.leftChild = min.rightChild*
2. Update the *parent* property of the 9 to point to the 10.

Line 12: *min.rightChild.parent = min.parent*
3. Update the *parent* property of *min* to point to the 15.

Line 13: *min.parent = node.parent*
4. Update the *leftChild* property of the 15 to point to the 8.

Line 14: *node.parent.leftChild = min*
5. Update the *leftChild* property of the 8 to point to the 4.

Line 15: *min.leftChild = node.leftChild*
6. Update the *rightChild* property of the 8 to point to the 10.

Line 16: *min.rightChild = node.rightChild*
7. Update the *parent* property of the 10 to point to the 8.

Line 17: *node.rightChild.parent = min*
8. Update the *parent* property of the 4 to point to the 8.

Line 18: *node.leftChild.parent = min*

The final tree with the 6 removed is shown in Figure 15.

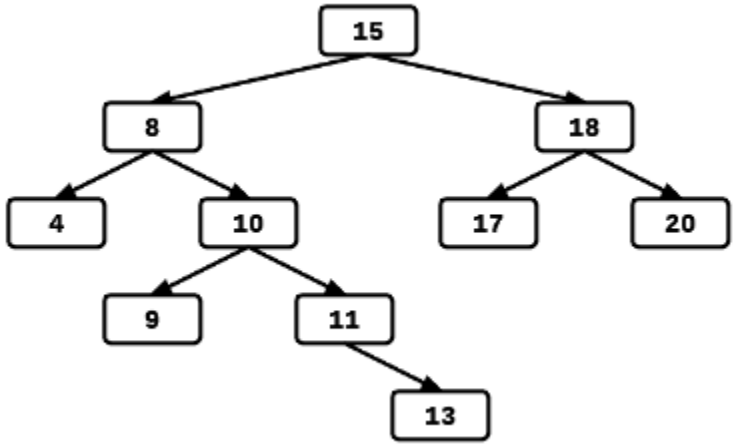


Figure 15. BST after the 6 is deleted and the tree is reordered to maintain the BST properties.

9.2.3 Find minimum or maximum value in a BST

The minimum value of a tree or sub-tree can be found by traversing left until reaching a node whose left child is NULL. That node is the minimum value. For example, in the tree shown in Figure 17, the minimum value in the tree is found by starting at the 15, going left to the 8, and then left to the 4. The 4 doesn't have a left child, which makes it the minimum value in the tree. The algorithm to find the minimum value in a BST is shown in Algorithm 9.6.

Algorithm 9.6. treeMinimum(node) Returns a pointer to the node with the minimum *key* value in a sub-tree, where the *root* of the sub-tree is *node*.

Pre-conditions *node* is a valid node in a BST and the starting node in the search.

Post-conditions Returns a pointer to the node with the minimum *key* value in a sub-tree by finding the node whose *leftChild* property is NULL.

Algorithm treeMinimum(node) 1. while (node.leftChild != NULL) 2. node = node.leftChild 3. return node

A similar approach is used to find the maximum value in a tree or sub-tree. Traverse right in the tree until reaching a node whose *rightChild* property is NULL. This node will be the maximum value in the tree. The algorithm to find the maximum value in a BST is shown in Algorithm 9.7.

Algorithm 9.7. treeMaximum(node) Return a pointer to the node with the maximum *key* value in a sub-tree, where the *root* of the sub-tree is *node*.

Pre-conditions *node* is a valid node in a BST and the starting node in the search
Post-conditions Returns a pointer to the node with the maximum *key* value in a sub-tree by finding the node whose *rightChild* property is NULL.

Algorithm treeMaximum(node) 1. while (node.rightChild != NULL) 2. node = node.rightChild 3. return node

The *treeMinimum()* and *treeMaximum()* algorithms can take any node in the tree as the starting node. For example, consider the tree T and sub-tree T' in Figure 16.

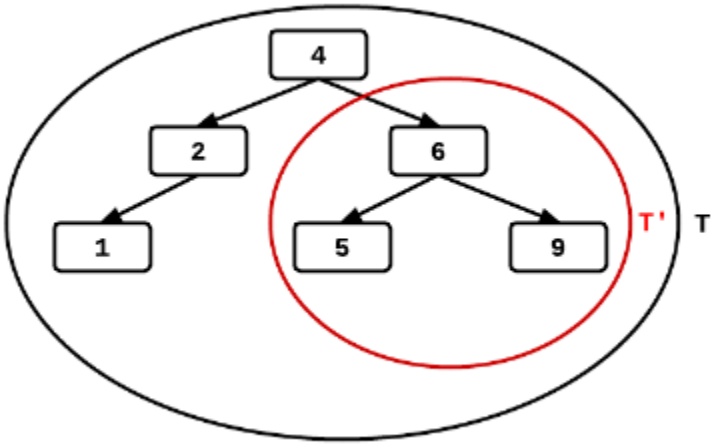


Figure 16. BST T and T'. The BST T' is a sub-tree within the BST T.

Finding the minimum or maximum of *T* and *T'* can produce a different result.

Example 8: Find the minimum value in T.

treeMinimum(T.root) = 1

In this call to *treeMinimum()*, the algorithm searches from the *root* of *T*, and finds that the minimum value in the BST is 1. This is the minimum value for the entire tree.

Example 9: Find the minimum value in T'.

treeMinimum(T'.root) = 5

In this call to *treeMinimum()*, the algorithm searches from the *root* of *T'*, which is the 6. The minimum value in that sub-tree is not the minimum for the entire tree, and the algorithm returns a 5.

Example 10: Find the maximum value for T and T'.

treeMaximum(T.root) = 9

treeMaximum(T'.root) = 9

When *treeMaximum()* is called on either *T* or *T'*, the same value is returned. *T'* contains the right-most node in the tree and therefore will contain the maximum value in the entire tree. The same would be true for the minimum value and a sub-tree that contains the left-most node in the entire tree.

9.2.4 Find nodes within a range of values

The properties of BSTs make it possible to efficiently identify all nodes in the tree with key values in a specified

range, such as nodes with values greater than 5 and less than 12.

Example 11: Find all nodes in T in Figure 16 with a value less than 3.

Steps:
1. Starting at the root, compare 3 to the *root.key* value. Since 3 < 4, go left and evaluate the nodes in the left sub-tree of the root. All nodes in the right sub-tree of the root can be excluded from evaluation; they will have values greater than 3.

2. Evaluate the left child of the root, which is the 2. Since 2 < 3, the entire left sub-tree of the 2 will also be less than 3. The only value in the left sub-tree is the 1.

3. The 2 doesn't have a right sub-tree to evaluate.

The values in the tree less than 3 are < 1, 2 >.

Example 12: Find all nodes with key values less than 10 in the BST in Figure 17.

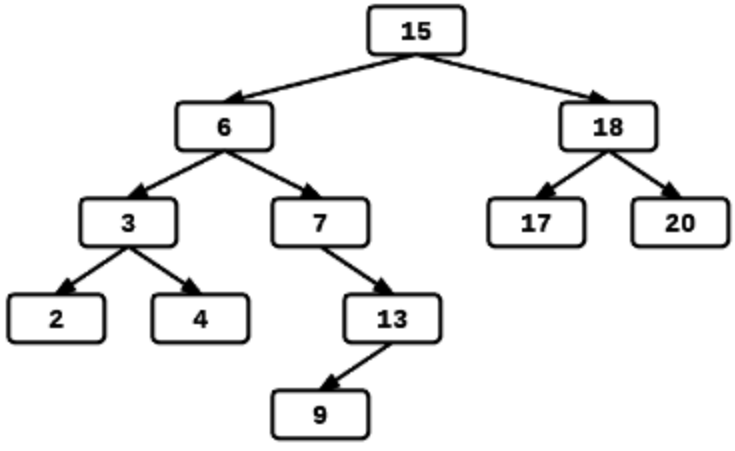


Figure 17. Find all nodes with key values less than 10 in this BST.

Steps:
1. Starting at the root, compare 10 to the root value. Since 10 < 15, go left and evaluate the nodes in the left sub-tree of the root. The right sub-tree of the root can be excluded from evaluation.

2. Evaluate the left child of the root, which has a value of 6, and since 6 < 10, that node and all nodes left of it are less than 10. The list of values less than 10 currently includes < 2, 3, 4, 6>.

3. Evaluate the right sub-tree of the 6, which will contain values between 6 and 14 by the BST properties. The values will be greater than or equal to 6 because it's the right sub-tree of the 6 and and less than 15 because it's the left sub-tree of the 15.

4. Evaluate right child of the 6, which is 7. Add 7 to the list of values less than 10, which now includes < 2, 3, 4, 6, 7>.

5. The 7 does not have a left child to evaluate.

6. Evaluate the right child of the 7, which has a value of 13. The left sub-tree of the 13 can contain values between 7 and 12. The right sub-tree of 13 will contain values greater than or equal to 13, which are all greater than 10.

7. Evaluate the left child of the 13, which has a value of 9. The 9 doesn't have any children to evaluate, which means that there are no additional node to check.

The final list of nodes in the BST with values less than 10 is < 2, 3, 4, 6, 7, 9 >.

9.3 BST Complexity

The runtime for the *search()*, *insert()*, and *delete()* algorithms on a BST depends on how the tree is built. For example, the following sequence of integers:
< 15, 8, 4, 18, 10, 17, 20, 9, 11 >
generates the BST shown in Figure 18. The BST is balanced; the left and right child positions for all nodes in the tree are occupied at all levels except the last level, and there is only a one-level difference in the left and right sub-trees of the root.

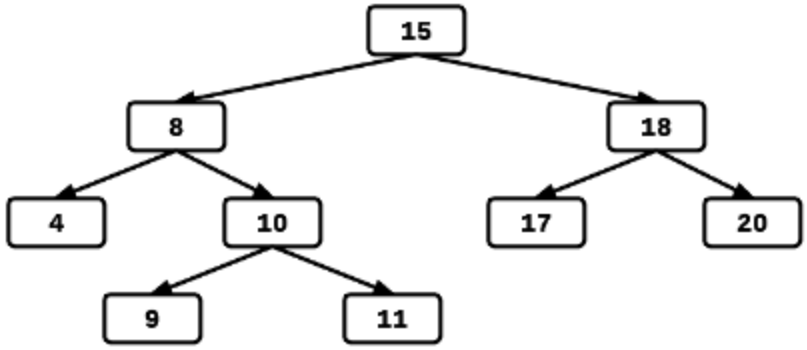


Figure 18. Example of a balanced BST. The left and right child positions are occupied at all levels except the last and there is only a one-level difference in the left and right sub-trees of the root.

However, a BST built from a sorted sequence of data:
<4, 8, 9, 10, 11, 15, 17, 18, 20>
generates the BST in Figure 19.

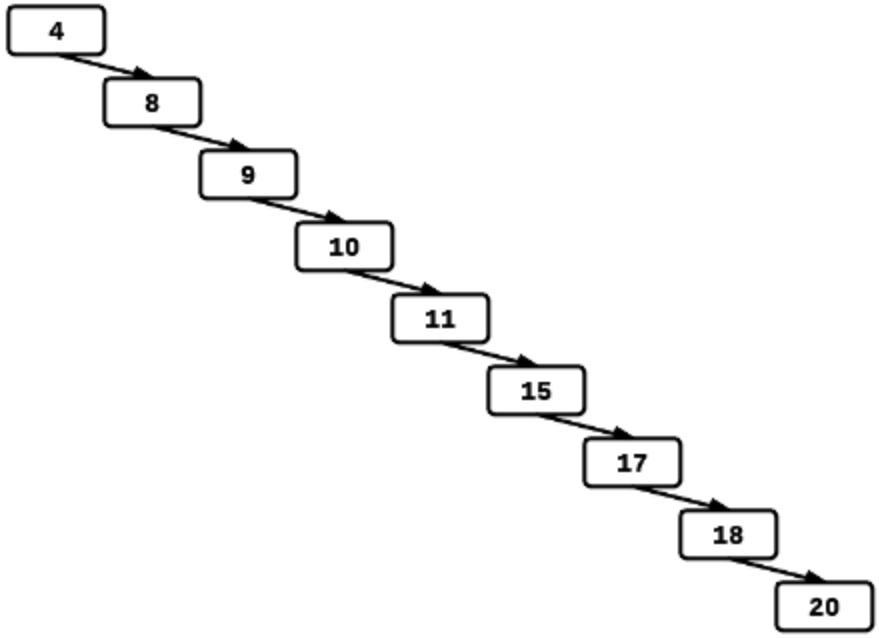


Figure 19. Example of a BST built from sorted data. The BST is effectively a linked list and operations on this BST will have O(n) behavior.

Definitions Leaf node *A leaf node is a node in a tree that does not have any children.*

Height of a tree *The height of a tree is the number of edges between the root to the deepest leaf node.*

Definition of balanced tree *A balanced binary search tree has the minimum possible maximum height. For each node x, the heights of the left and right sub-trees of x differ by at most 1.*

In a BST, basic operations to search, insert, and delete run in O(h) time, where h is the height of the tree. When $n = h$, where n is the number of nodes, these operations are O(n) and the BST has the same runtime properties as a linked list. When the tree is balanced, as in Figure 18, the distance from the root to any leaf node at the bottom of the tree, is $\log_2(n)$, where n is the number of nodes in the tree.

In a BST with 9 nodes, there are 3-4 levels, and in the worst case, there would be 4 comparisons to find a node in the tree. Calculating $\log_2(9) \approx 3.16$ shows that log(n) is a good approximation.

In contrast to the height-balanced tree in Figure 18, the unbalanced tree in Figure 19 would require *n* comparisons to search for a node at the bottom of the tree. Searching for the 20, for example, would require evaluating all 9 nodes in the tree. On smaller trees, it may not matter that much if the tree is structured like an n-node linked list, but with thousands or millions of nodes, having a balanced tree can significantly improve the runtime of operations on the BST. For example, consider a tree with a million nodes. If the tree is balanced, then the height of the tree is $\log_2(1000000) \approx 19$, anything can be found in the tree in approximately 19 comparisons. However, if the tree is unbalanced, up to 1,000,000 comparisons could be required.

10 Recursion

Recursion is the process of a function calling itself, and it is frequently used to evaluate structures that can be defined by self-similarity, such as trees. A recursive call to a function evaluates a smaller and smaller instance of the structure until the smallest case is reached.

Recursion is typically used on problems where the structure of the data is also recursive, such as a file system on a computer. The directory structure is defined recursively in terms of smaller and smaller directory structures. At the top level, there are directories and files. Within the directories, there are other directories and files, and within those directories there can be other directories, and so on. Searching through the file system reveals a repeating pattern down to a level where there are only files. In a recursive search algorithm, if the search gets to this level and doesn’t find the specified file, then the algorithm returns that the file is not found in the file system.

10.1 Rules for recursive algorithms

There are two rules that define the structure of any recursive algorithm. The algorithm needs to include:

• **A base case.** This is the smallest unit of the problem that can be defined. Once the base case is reached, the algorithm should return without additional recursive calls.

• **A set of rules that can reduce all cases down to the base case.** The base case is the exit strategy for a recursive algorithm. If the algorithm never reaches the base case, then it will never exit.

The base case is defined by the structure of the data. In a file system, the base case is an individual file. Traversal of a directory structure down to the base case means going down to a level where there are only files and no additional directories. In a BST, the base case is an individual node with no children. Smaller and smaller sub-trees can be evaluated until a sub-tree is reached that is a single node.

Consider the following function called *printNode()* in Code 10.1 to traverse a binary tree and print the key values of all nodes in the tree. The function takes a node in the tree as an argument and then recursively visits the left and then the right children of all nodes in the tree.

Code 10.1. printNode(node *n) Traverse a binary tree by recursively evaluating the left and then the right children of a node.

```
void printNode(node *n) 1. cout<<"key: "<<n->key>>endl; 2. if(n->leftChild!=NULL) 3. printNode(n->leftChild); 4. if(n->rightChild!=NULL) 5. printNode(n->rightChild);
```

The first time *printNode()* is called, it is passed the *root* of the tree as an argument. If the node has a left child, then *printNode()* is called again on Line 3 with the *node.leftChild* is the argument. If the node doesn't have a

left child, the algorithm checks if the node has a right child on Line 4, and if so, *printNode()* is called on *node.rightChild*. The *printNode()* function will be called on every node in the tree until there are no more left or right children to evaluate.

In the *printNode()* function, the base case occurs when the left and right children of a node are NULL. When this condition is reached, the current instance of the function exits. The rule that reduces a case to the base case is calling *printNode()* on a child node. Eventually, the bottom of the tree will be reached.

10.2 Tree traversal algorithms

Recursion is often used in tree traversal algorithms. Just as the contents of an array can be traversed, so can the nodes in a tree to determine the values in the tree. With any tree traversal algorithm, the objective is to evaluate every node in the tree exactly once, and the algorithm used determines the order in which the nodes are visited.

There are three orderings to consider for tree traversals: • **In-order** - Nodes are visited in the order *left child - parent - right child*, which can generate a sorted output in a binary search tree (BST).
• **Pre-order** - Nodes are visited in the order *parent - left child - right child*.
• **Post-order** - Nodes are visited *left child - right child - parent*.

10.2.1 In-order tree traversal

Consider the three-node tree shown in Figure 1.

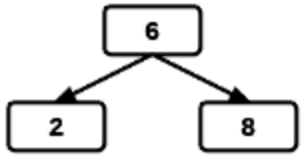


Figure 1. BST with three nodes.

If the nodes were listed in sorted order, the output would be 2, 6, and 8. From the root of this three-node tree, that ordering can also be expressed in terms of the tree structure as *left child*, *parent*, *right child*. In a larger tree, that same *left child*, *parent*, *right child* pattern would be applied to generate an ordered output for the entire tree. An in-order traversal algorithm that prints the keys in a binary tree is shown in Algorithm 10.1.

Algorithm 10.1. printNode(node) Traverse a binary tree by visiting nodes in the order *left child - parent - right child*.

Pre-conditions *node* is a valid node in the tree.

Post-conditions All *key* values in the tree are displayed.

Algorithm printNode(node) 1. if (node.leftChild != NULL) 2. printNode(node.leftChild) 3. print(node.key) 4. if (node.rightChild != NULL) 5. printNode(node.rightChild)

Example 1: Traverse the BST in Figure 2 using an in-order tree traversal algorithm.

An in-order traversal needs to process the minimum value in the tree first, which is found by traversing left until a node is found that has no left child. In this example, the minimum value in the tree is the 3.

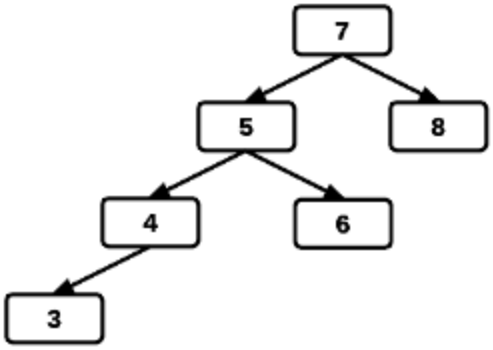


Figure 2. BST to evaluate using an in-order traversal. The first node that will be processed is the node with the value 3, and the last node that will be processed is the node with the value 8.

Examining how calls to the *printNode()* algorithm are pushed and popped from the call stack can make it easier to understand why the algorithm produces an in-order output. Calls to *printNode()* are pushed onto the stack, and then popped off the stack when they complete.

Steps: 1. Call *printNode()* from the *root*, which pushes it onto the call stack as *printNode(7)*. A visual representation of the call stack is shown in Figure 3.

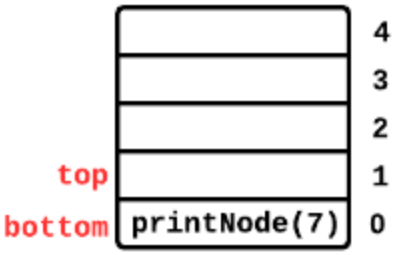


Figure 3. Call stack after printNode(7) called on the root of the BST in Figure 2.

2. On Line 1, the conditional *node.leftChild != NULL* is true, and *printNode()* is called again as *printNode(5)* on the *left child* of the *root*. The call stack is shown in Figure 4.

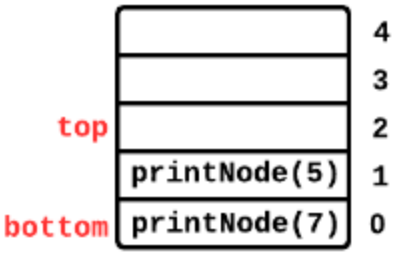


Figure 4. Call stack after printNode(7) and printNode(5) called.

3. The call to *printNode(5)* is now the currently running version, and on Line 1, *node.leftChild != NULL* is evaluated again for the 5. The 5 does have a *left child*, which is the 4, and *printNode()* is called again as *printNode(4)* and pushed onto the call stack (Figure 5).

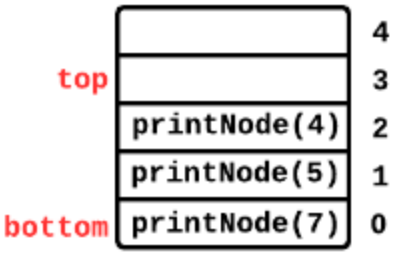


Figure 5. Call stack after printNode(7), printNode(5), and printNode(4) are called.

4. The call to *printNode(4)* is now the currently running version, and on Line 1, *node.leftChild != NULL* is evaluated again for the 4. The 4 does have a *left child*, which is the 3, and *printNode()* is called again as *printNode(3)* and pushed onto the call stack (Figure 6).

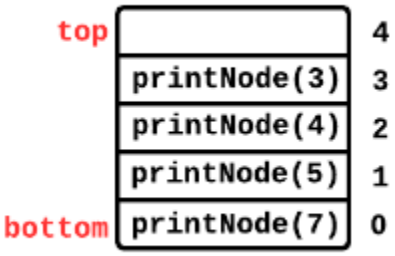


Figure 6. Call stack after printNode(7), printNode(5), printNode(4), and printNode(3) called.

5. The call to *printNode(3)* is now the currently running version. The conditional on Line 1, *node.leftChild != NULL* is false because the 3 doesn't have a *left child*. The call to *printNode()* on Line 2 is skipped.

6. On Line 3, the *key* value of the node is printed, which outputs a 3. This is the minimum value in the tree and it is the first value to be printed.

7. On Line 4, the conditional *node.right != NULL* is false because the 3 doesn't have a *right child*. The call to *printNode(3)* completes and is popped off the stack (Figure 7).

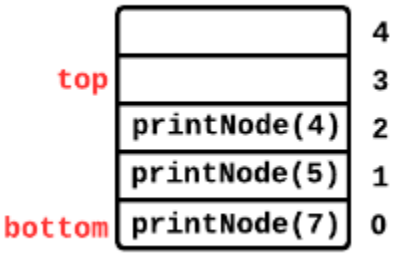


Figure 7. Call stack after printNode(3) completes and is popped off the stack.

8. Program execution returns to *printNode(4)* at the spot where *printNode(3)* was called, which is Line 2. The next line to execute is Line 3, which prints a 4. At this point, the program has output 3, 4.

9. On Line 4, the conditional *node.rightChild != NULL*, is false because the 4 doesn't have a *right child*. The call to *printNode()* on Line 5 is skipped and *printNode(4)* exits and is popped off the stack. The state of the stack is shown in Figure 8.

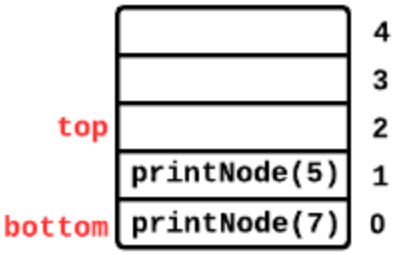


Figure 8. Call stack after printNode(4) completes and is popped off the stack.

10. Program execution returns to the point in *printNode(5)* where *printNode(4)* was called, which is on Line 2. The next line to execute is Line 3, which prints a 5. At this point in the program, the output is 3, 4, 5.

11. On Line 4, the conditional checks if the 5 has a *right child*, which it does. The algorithm is called again as *printNode(6)* and pushed onto the call stack (Figure 9).

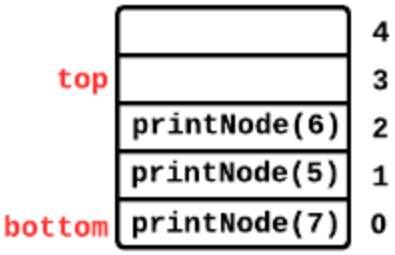


Figure 9. Call stack after printNode(6) is called and pushed onto the stack.

12. The currently running version is now *printNode(6)*. The 6 does not have a *left child*, so the conditional on Line 1 is false. The algorithm advances to Line 3 and prints a 6. At this point, the program output is 3, 4, 5, 6.

13. On Line 4, the conditional checks if the 6 has a *right child*, which it doesn't. The call to *printNode(6)* completes and is popped off the stack (Figure 10).

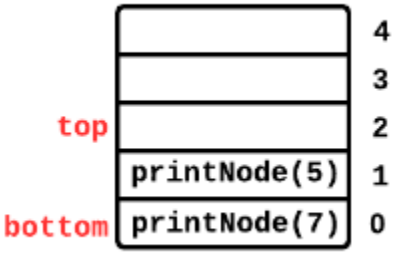


Figure 10. Call stack after printNode(6) completes and is popped off the stack.

14. Program execution returns to *printNode(5)* at the spot where *printNode(6)* was called, which is Line 5. The *printNode(5)* call completes and is popped off the stack (Figure 11).

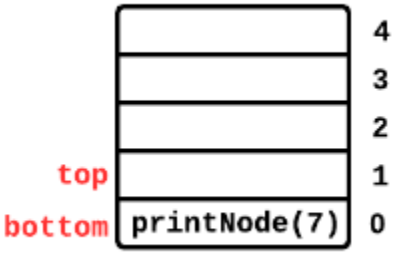


Figure 11. Call stack after printNode(5) completes and is popped off the stack.

15. Program execution returns to the spot in *printNode(7)* where *printNode(5)* was called, which is Line 2. The next line to execute is Line 3, which prints a 7. At this point in the program, the output is 3, 4, 5, 6, 7.

16. On Line 4, the conditional checks if the 7 has a *right child*, which it does. On Line 5, *printNode()* is called again as *printNode(8)* and pushed onto the call stack (Figure 12).

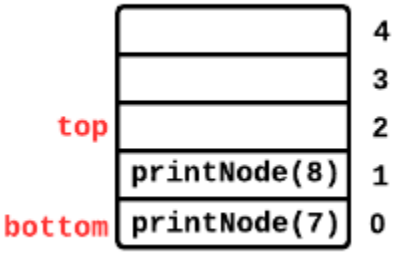


Figure 12. Call stack after printNode(8) is called and pushed onto the stack.

17. The call to *printNode(8)* is now the currently running version. The 8 doesn't have a *left* or a *right child*, so neither of the conditionals will be true and no additional calls to *printNode()* will be made. On Line 3, *printNode(8)* displays the value of the node. The output of the program after *printNode(8)* executes is 3, 4, 5, 6, 7, 8.

18. The call to *printNode(8)* completes and is popped off the stack. The program returns to *printNode(7)* on Line 5 where *printNode(8)* was called.

19. The *printNode(7)* function completes and is popped off the stack.

20. The stack is empty, and the tree traversal is complete. The output of the program was 3, 4, 5, 6, 7, 8.

10.2.2 Pre-order tree traversal

In a pre-order tree traversal, the value of the root node is printed before the values of the child nodes. In the recursive pre-order traversal algorithm in Algorithm 10.2, the print statement to output the value of the node appears before additional calls to the algorithm for each of the node's children. Stepping through the algorithm and

drawing the output and the call stack can demonstrate the expected output for a pre-order traversal. Using the same tree as in the in-order traversal example, the output for the pre-order traversal is 7, 5, 4, 3, 6, 8.

Algorithm 10.2. printNodePreorder(node) Display node values in a tree using a pre-order traversal that evaluates *parent - left child - right child*.

Pre-conditions *node* is a valid node in the tree
Post-conditions All *key* values in the tree are displayed.

Algorithm printNodePreorder(node) 1. print(node.key) 2. if (node.leftChild != NULL) 3. printNodePreorder(node.leftChild) 4. if (node.right != NULL) 5. printNodePreorder(node.rightChild)

10.2.3 Post-order tree traversal

In a post-order tree traversal, the children are printed before the root value.
Using the post-order algorithm shown in Algorithm 10.3 and the tree from the previous examples, the output would be 3, 4, 6, 5, 8, 7.

Algorithm 10.3. printNodePostorder(node) Display node values in a tree using a post-order traversal that evaluates *left child - right child - parent*.

Pre-conditions *node* is a valid node in the tree
Post-conditions All *key* values in the tree are displayed.

Algorithm printNodePostorder(node) 1. if (node.leftChild != NULL) 2. printNodePostorder(node.leftChild) 3. if (node.rightChild != NULL) 4. printNodePostorder(node.rightChild) 5. print(node.key)

11 Tree Balancing

Binary search trees (BST) provide an efficient structure for storing and retrieving data. When the BST is balanced with a height of O(log n), the complexity of insert, search, and delete operations is also O(log n), where *n* is the number of nodes in the tree. However, in an unbalanced tree, the complexity of these operations can be O(n) in the worst case.

Tree-balancing algorithms are applied to BSTs to ensure that, as nodes are added to the tree, the tree remains balanced with an $O(\log_2 n)$ height and the complexity of operations on the tree is also $O(\log_2 n)$.

11.1 Red-black trees

One common approach to tree balancing is to build the BST as a red-black tree. In the red-black tree algorithm, each node in the BST is assigned a color, either red or black, and the nodes in the tree are ordered such that no path from the root to a leaf can be more than twice as long as any other path. This coloring results in red-black trees having a height of O(log n), which guarantees a worst-case runtime of O(log n) on search, insert, and delete operations.

11.1.1 Red-black node properties

Each node in a red-black tree has at least the following properties: • color
• key
• left child • right child • parent

The only red-black property not found in a regular BST is the color, which is added to the nodes to create the structure in the tree. The properties that the tree must exhibit in order to be a valid red-black tree are:
Property 1: A node is either red or black.
Property 2: The root node is black.
Property 3: Every leaf (NULL) node is black.
Property 4: If a node is red, then both of its children must be black.
Property 5: For each node in the tree, all paths from that node to the leaf nodes contain the same number of black nodes.

Another difference between a regular BST and a red-black tree is how the leaf nodes are represented. In a red-black tree, the leaf nodes are external sentinel nodes with all of the same properties as a regular node, but they are

effectively empty nodes. The leaf nodes are black to satisfy Property 3 of a red-black tree.

Figure 1 shows the features of a red-black tree node. The root node is black to satisfy Property 2. The nodes are either red or black to satisfy Property 1. The sentinel nodes, shown as the smaller boxes, are also black as required by Property 3, and Property 4 requiring children of red nodes to be black.

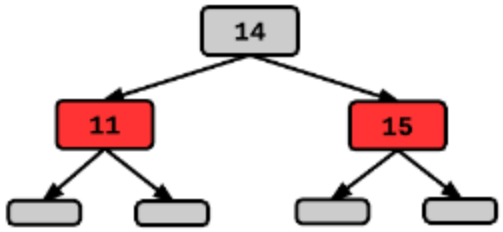


Figure 1. An example of a three-node red-black tree. The tree has red and black nodes, and the root of the tree is black. The smaller boxes are the sentinel nodes, which are black.

11.1.2 Red-black tree ADT

The red-black tree ADT contains similar functionality to the BST ADT to insert and delete nodes in the tree. Searching a red-black tree uses the same algorithm as searching a regular BST, shown in Algorithm 9.1. The red-black tree ADT, shown in ADT 11.1, contains public methods for the insert, delete, and search operations, as well as private methods to support these operations.

ADT 11.1. Red-black Tree RedBlackTree: 1. private: 2. root 3. leftRotate(node) 4. rightRotate(node) 5. insertRB(value) 6. rbBalance(node) 7. public: 8. Init() 9. redBlackInsert(value) 10. redBlackDelete(value) 11. search(value) 12. deleteTree()

11.1.3 Red-black tree balancing

When nodes are added or deleted from a red-black tree, the operation can destroy the red-black properties of the tree. For example, deleting the 4 from the red-black tree in

Figure 2 potentially creates a configuration where the 2, which is red, has a red child. This configuration violates Property 4, which states that children of a red node must be black. To resolve this violation, the tree can be balanced using rotation and recoloring to restore the red-black properties. The particular balancing algorithm needed depends on the operation and the violation.

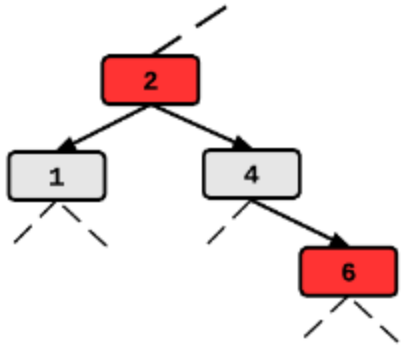


Figure 2. In this red-black tree, a delete operation on the 4 potentially creates a configuration where the red 2 node has a red child, which violates Property 4.

11.1.4 Left and right rotations

Rotations are local operations on nodes that reorder the nodes in the tree in a way that preserves the BST properties and set the tree up for recoloring to restore the red-black properties. Rotations are used in red-black trees as well as in most other tree-balancing algorithms.

There are two types of rotations: a left rotation and a right rotation. Figure 3 shows the tree that results from both a left and a right rotation. These rotations are inverses of each other: a tree rooted at *x*, shown in the right-side image in Figure 3, that undergoes a left rotation produces the tree shown in the left-side image in Figure 3. Performing a right rotation on that same tree, rooted at *y*, returns the tree to its original state.

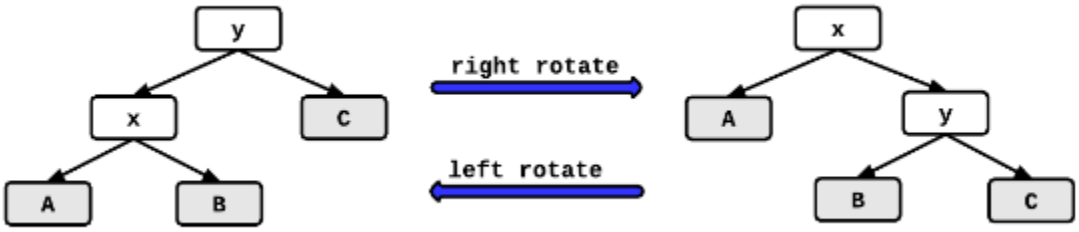


Figure 3. Left and right rotations change the structure of the tree while still maintaining the BST properties. The rotations are inverses of each other: a sub-tree rotated in one direction will return to its original state if it is rotated in the other direction.

The *leftRotate()* algorithm (Algorithm 11.1) takes the node to rotate about as the argument and produces the rotated tree. The algorithm uses the *nullNode* variable as the sentinel node for the tree.

Algorithm 11.1. leftRotate(x) Rotate the sub-tree about the node *x* in a red-black tree.

Pre-conditions *x* is a valid node in a red-black tree.
nullNode defined as the empty sentinel node of the red-black tree.
root defined as the root of the red-black tree.

Post-conditions Tree rotated about *x*.

Algorithm leftRotate(x) 1. y = x.rightChild 2. x.rightChild = y.leftChild 3. if(y.leftChild != nullNode) 4. y.leftChild.parent = x 5. y.parent = x.parent 6. if (x.parent == nullNode) 7. root = y 8. else 9. if(x == x.parent.leftChild) 10. x.parent.leftChild = y 11. else 12. x.parent.rightChild = y 13. y.leftChild = x 14. x.parent = y

The *rightRotate()* algorithm, shown in Algorithm 11.2, is reversed from the left rotation by swapping the references to the left and right children for a node and the references to the *x* and *y* nodes.

Algorithm 11.2. rightRotate(y) Rotate the sub-tree about the node *y* in a red-black tree.

Pre-conditions *y* is a valid node in a red-black tree.
nullNode defined as the empty sentinel node of the red-black tree.
root defined as the root of the red-black tree.

Post-conditions Tree rotated about *y*.

Algorithm rightRotate(y) 1. x = y.leftChild 2. y.leftChild = x.rightChild 3. if(x.rightChild != nullNode) 4. x.rightChild.parent = y 5. x.parent = y.parent 6. if(y.parent == nullNode) 7. root = x 8. else if (y == (y.parent.leftChild)) 9. y.parent.leftChild = x 10. else 11. y.parent.rightChild = x 12. x.rightChild = y 13. y.parent = x

Example 1: Demonstrate the left rotation on the red-black tree shown in Figure 4 using the leftRotate() algorithm in Algorithm 11.1.

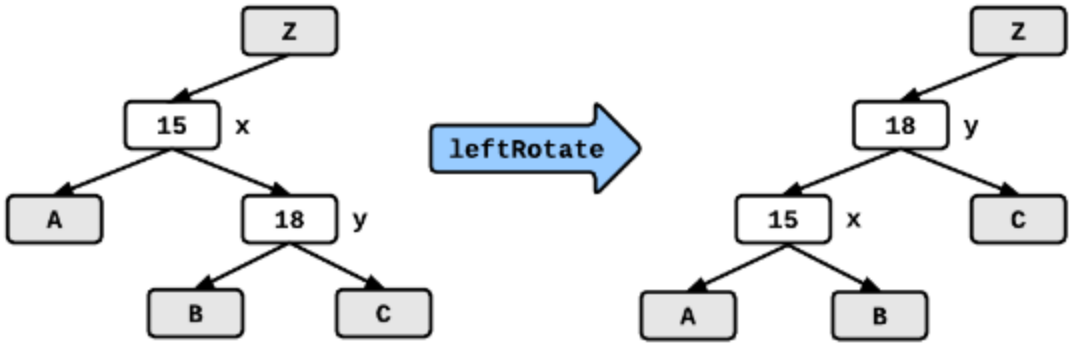


Figure 4. A left rotation on the node x in the red-black tree on the left produces the tree shown on the right.

In this example, the 15 is the *x* and the 18 is the *y* in the *leftRotate()* algorithm. The nodes that change in the rotation are:
• x.parent • y.parent • x.rightChild • y.leftChild
The nodes that don't change in the rotation are: • y.rightChild • x.leftChild

Steps:

• **Line 2:** changes the right child of *x* to point to the left child of *y.* Figure 5 shows that the right child of *x* is now the *B* node.

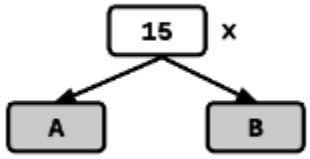


Figure 5. The right child of the node x is the B node, which was the left child of the node y. The left child of x hasn't changed.

• **Lines 3 and 4:** *x's* new right child is updated to set *x* as its parent (Figure 6).

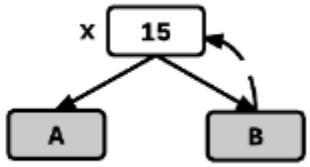


Figure 6. The parent of B is set to be the node x.

• **Line 5:** *y's* parent is set to *x's* parent. This operation effectively moves *y* into *x's* position in the tree. The previous parent for *y* was *x*. After this update, *y's* parent is *z* (Figure 7).

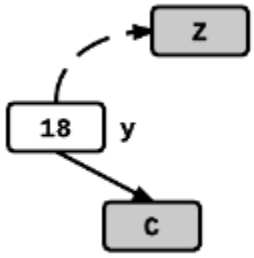


Figure 7. Update y's parent to move y into x's position in the tree. The previous parent for y was x, and now y's parent is z. The right child of y hasn't changed.

• **Lines 6-12:** update *x's* parent to point to *y* as a child instead of *x*. There are separate cases for whether *x* was its

parent’s left or right child or the root of the tree. The left child case is handled on Line 10, and the right child case is handled on Line 12. In this example, *x* is its parent's left child (Figure 8). After the update, *y* is *z’s* left child.

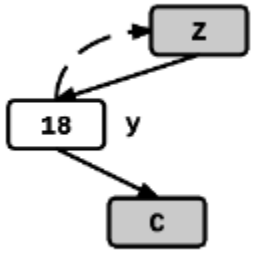


Figure 8. The node z is updated to point to y as its left child instead of x.

• **Lines 13-14:** the node *x* replaces *B* as *y's* left child by setting *y’s* left child pointer to *x* and *x’s* parent pointer to *y* (Figure 10). This final step completes the rotation.

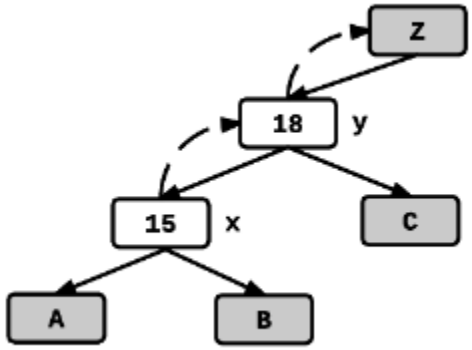


Figure 9. Move x to replace B as the left child of y by setting y's left child pointer to x and x's parent pointer to y. The left rotation is now complete.

11.1.5 Inserting a node into a red-black tree

Nodes are added to red-black trees in the same way they are added to a regular BST. However, when a node is added, the operation can destroy the red-black tree properties, which requires that there are additional steps in the algorithm to restore these properties.

There are three changes to the BST insert operation needed to support a red-black tree.

In a red-black tree:
1. Replace all instances of NULL in the BST *insert()* algorithm (Algorithm 9.3) with the sentinel node *nullNode*. This change sets the parent of the root to *nullNode* and the left and right children of a new node to *nullNode*.
2. Set the color of the new node to red.
3. Resolve any violation of the red-black properties using tree balancing.

If *x* is a node added to a red-black tree, then the initial conditions on *x* are: • x.color = red • x.leftChild =nullNode • x.rightChild = nullNode
When a node is added to the tree, the two properties that can be violated are.
1. The root must be black.
2. The children of a red node must be black.

Both violations are possible because a new node is initially colored red. There are six possible configurations that a red-black tree can take on when a new node is inserted into the tree. Three configurations are symmetric to the other three depending on whether the parent of the new node is the left or right child of its parent. Figure 10 shows an example where the parent is the left child. In this figure, the new node is labeled *x*, and its parent is the 15. The parent of the 15 is the 18.

The steps needed to rebalance the tree depend on the color of the new node's "uncle" node. Figure 10 also shows an example of an “uncle” node. The new node x has an “uncle” that is *x.parent.parent.rightChild.* If *x’s* parent were a right child, then *x’s* “uncle” would be *x.parent.parent.leftChild.*

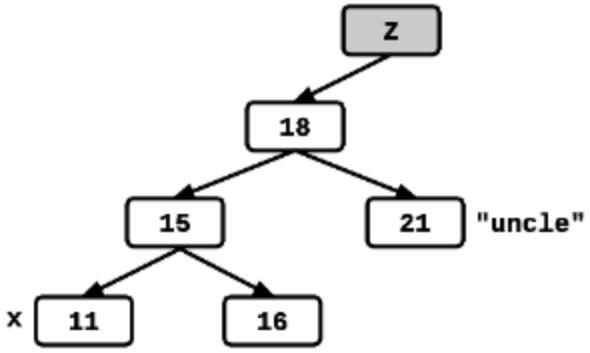


Figure 10. The “uncle” of x is x.parent.parent.right. The color of x's "uncle" node determines the steps needed to rebalance the tree after inserting a node.

Case 1: The “uncle” node is red.

If the *parent.parent.rightChild* of the new node is red (shown as “uncle” in Figure 11), then *parent* of the new node is also red, and the *parent.parent* of the new node is black. In Figure 11, the new node is labeled *x*. It is initially colored red, and its *parent* and “uncle” are also red.

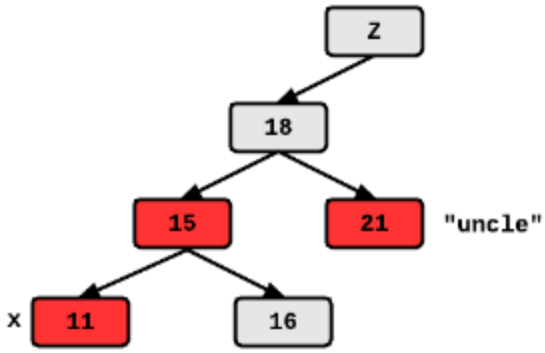


Figure 11. Case 1 example, where the "uncle" node is red. The x points to the new node. The parent of x is also red, which violates the constraint that a red node can’t have a red child.

Steps to resolve a Case 1 violation in the tree: 1. Recolor both the *parent* and the “uncle” of the new node to be black, and recolor the *parent.parent* of the new node to be red. This recoloring resolves the violation up to the *parent.parent* level in the tree.

2. Move up two levels in the tree by setting *x* = *x.parent.parent.* Figure 12 shows the red-black tree after the violations have been resolved. The *x* in Figure 12 points to the node that would be recolored next, if additional iterations of recoloring were necessary.
3. Repeat Steps 1 and 2 until *x* is the root of the tree or *x's* parent is black.

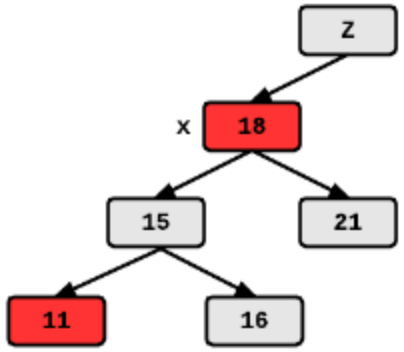


Figure 12. The red-black tree after nodes have been recolored to fix the violation of a red node having a red child.

Case 2: The new node is a right child and its uncle is black.

Case 3: The new node is a left child and its uncle is black.

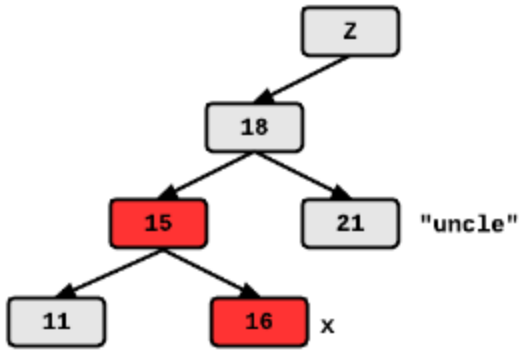


Figure 13. Case 2 where the "uncle" node is black and the new node x is a right child. The violation is that a red node has a red child.

In both Case 2 and Case 3, the “uncle” node is black. The difference in the cases is whether the new node is a left or right child of its parent. Figure 13 shows an example of Case 2, where *x* is the new node and it’s a right child of the 15.

Steps to resolve a Case 2 violation in the tree: 1. Set *x = x.parent.*
2. Apply the *leftRotate()* algorithm to *x* to convert a Case 2 configuration to a Case 3 configuration. Additional rebalancing can then be applied to resolve the Case 3 violation. The result of the left rotation on the tree in Figure 13 is shown in Figure 14. The new node is now a left child of its parent.

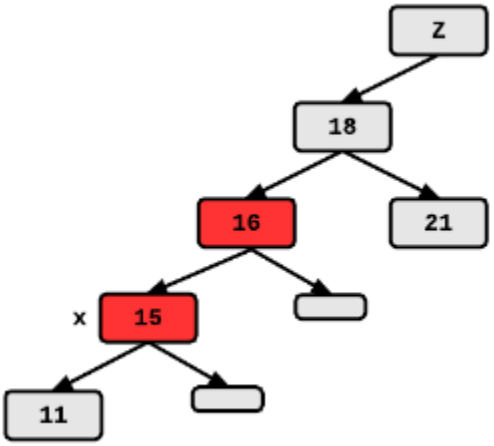


Figure 14. A Case 3 configuration is generated from a left rotation on x's parent on the tree in Figure 13.

Steps to resolve a Case 3 violation in the tree: 1. Recolor *x.parent* and *x.parent.parent*.
2. Apply a right rotation about *x.parent.parent* on the tree in Figure 14 to get the tree in Figure 15.

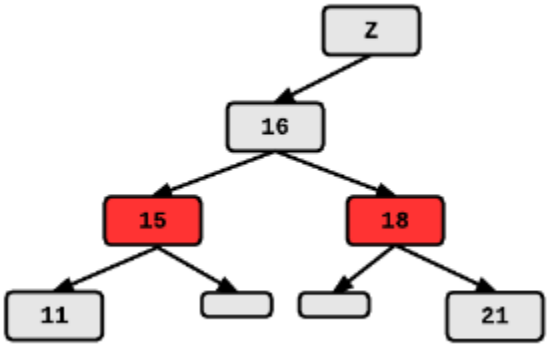


Figure 15. Red-black sub-tree after a right rotation on the tree shown in Figure 14. The tree is now balanced; all red-black violations have been resolved.

The algorithm to insert a node into a red-black tree is shown in Algorithm 11.3. The *redBlackInsert()* algorithm takes the value of the node to insert as an argument, and calls *insertRB(),* shown in Algorithm 11.6 to add the node to the tree.

Algorithm 11.3. redBlackInsert(value) Insert a node into a red-black tree and apply the appropriate tree-balancing algorithm to restore the red-black properties.

Pre-conditions *value* is a valid node key value. *insertRB()* exists to create and return a pointer to the new node.

Post-conditions New node inserted into a red-black tree with no violations of the red-black properties.

Algorithm redBlackInsert(value) 1. x = insertRB(value) 2. while ((x != root) and (x.parent.color == red)) 3. if (x.parent == x.parent.parent.left) 4. uncle = x.parent.parent.right 5. if (uncle.color == red) 6. RBCase1Left(x, uncle) 7. x = x.parent.parent 8. else 9. if (x == x.parent.right) 10. x = x.parent 11. leftRotate(x) 12. //Case 3 - x is now left child 13. RBCase3Left(x) 14. else 15. //x’s parent is a right child. 16. //Exchange right and left 17. root.color = black

Algorithm 11.4. RBCase1Left(x, uncle) Recolors the red-black tree for the case where *uncle* is red and the parent of the new node *x* is a left child.

Pre-conditions *x* and *uncle* are valid nodes in a red-black tree.

Post-conditions Color of the parent, grandparent, and uncle of the new node are changed.

Algorithm RBCase1Left(x, uncle) 1. x.parent.color = black 2. uncle.color = black 3. x.parent.parent.color = red

Algorithm 11.5. RBCase3Left(x) Recolors and rotates a red-black tree for the case where the new node *x* is a left child and the "uncle" node is black.

Pre-conditions *x* is a valid node in a red-black tree.

Post-conditions Color of the parent and grandparent of the new node are changed and the tree is right rotated about the grandparent of the new node.

Algorithm RBCase3Left(x) 1. x.parent.color = black 2. x.parent.parent.color = red 3. rightRotate(x.parent.parent)

Algorithm 11.6. insertRB(value) Private method called from *redBlackInsert()* to insert a node into a red-black tree and return a pointer to the node.

Pre-conditions *value* is a valid node key value.

Post-conditions New node inserted into the tree. Returns a pointer to the new node.

Algorithm insertRB(value) 1. node.left = nullNode 2. node.right = nullNode 3. node.color = red 4. node.key = value 5. tmp = root 6. while(tmp != nullNode) 7. parent =

tmp 8. if(node.key < tmp.key) 9. tmp = tmp.leftChild 10. else 11 tmp = tmp.rightChild 12. if (parent == nullNode) 13. root = node 14. else if(node.key < parent.key) 15. parent.leftChild = node 16. else 17. parent.rightChild = node 18. return node

Example 2: Add a 4 to the red-black tree shown in Figure 16 using the redBlackInsert() algorithm in Algorithm 11.3.

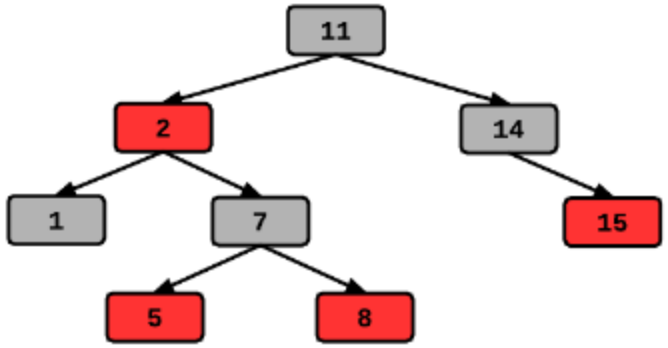


Figure 16. When a 4 is added to this red-black tree, it is added as the left child of the 5. The operation will create a violation because the red 5 will have a red child. The Case 1 rebalancing algorithm needs to be applied; the "uncle" is the red 8 node.

The 4 is added as the left child of the 5, which creates a situation where the red 5 has a red child and violates Property 4 that a red node can't have a red child. The "uncle" of the new node is the red 8, which results in a Case 1 violation: the new node is a left child with a red "uncle".

Steps:

1. **Line 6 *redBlackInsert():*** call *RBCase1Left()* with the new node and "uncle" node as arguments.
2. **Line 1 *RBCase1Left()*:** Recolor the 5 to be black.
3. **Line 2 *RBCase1Left()*:** Recolor the 8 to be black.
4. **Line 3 *RBCase1Left():*** Recolor the 7 to be red. The state of the tree after recoloring is shown in Figure 17. All nodes below the 7 in the tree should now have the correct

coloring. However, the recoloring created a violation between the 2 and 7 nodes, where the 2 has a red child.

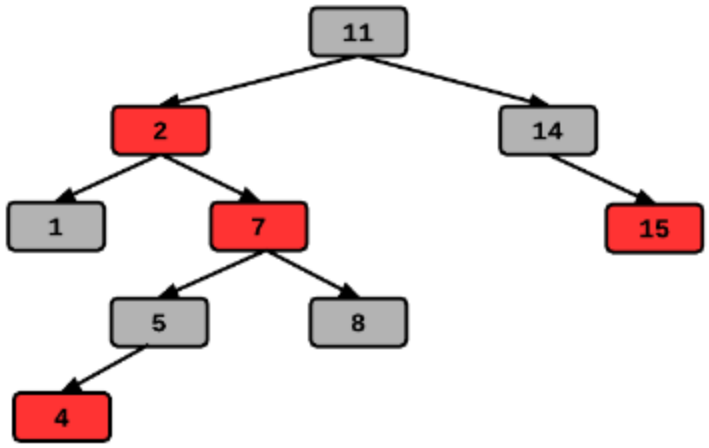


Figure 17. Red-black tree after applying the Case 1 rules to recolor the 5, 7, and 8 nodes. Now there is a violation between the 2 and 7, where the 2 has a red child.

5. **Line 7** ***redBlackInsert():*** After the *RBCase1Left()* algorithm exits, control returns to *redBlackInsert()*. Move up two levels in the tree to examine the properties of the tree starting at the 7 and working toward the root of the tree. The *x* now points to the 7.
6. **Line 4** ***redBlackInsert():*** Set the "uncle" to be the 14, shown as *y* in Figure 18. The tree is now in a Case 2 configuration - *x* is a right child and the "uncle" is black.

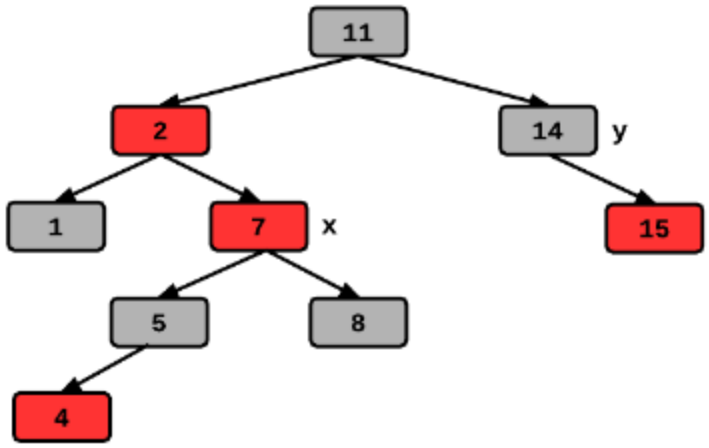


Figure 18. Positions of x and "uncle" in the red-black tree-balancing algorithm. The x node is the node currently being evaluated, and the "uncle" of x is the y node.

7. **Lines 10-11** ***redBlackInsert():*** Set *x* to point to *x.parent,* which is the 2 node. Apply the *leftRotate()* algorithm to *x*. The resulting tree is shown in Figure 19. The 7 has moved up one level in the tree and x is now its left child. The tree is now an example of a Case 3 configuration: *x* is a left child with a black "uncle".

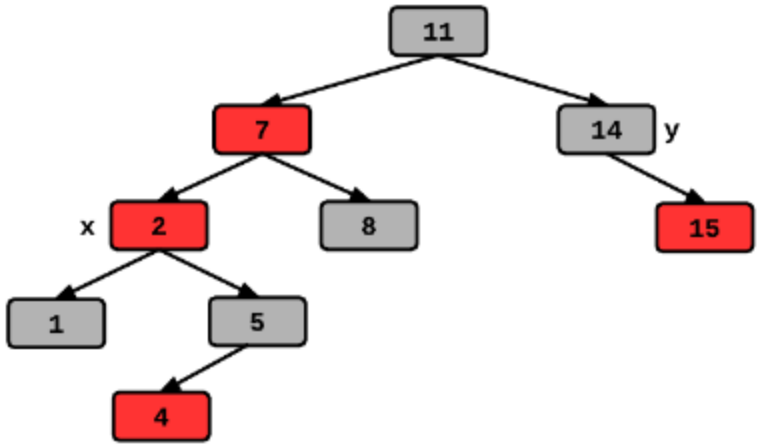


Figure 19. Red-black tree configuration after a left rotation about the 2. The 7 has moved up one level in the tree, and the 2 is now its left child. The tree is an example of a Case 3 configuration: x is a left child with a black "uncle".

8. **Line 13** ***redBlackInsert():*** Call *RBCase3Left()* on the 2, which recolors the 7 to be black and the root to be red, and generates the configuration shown in Figure 20. The red root violates Property 2.

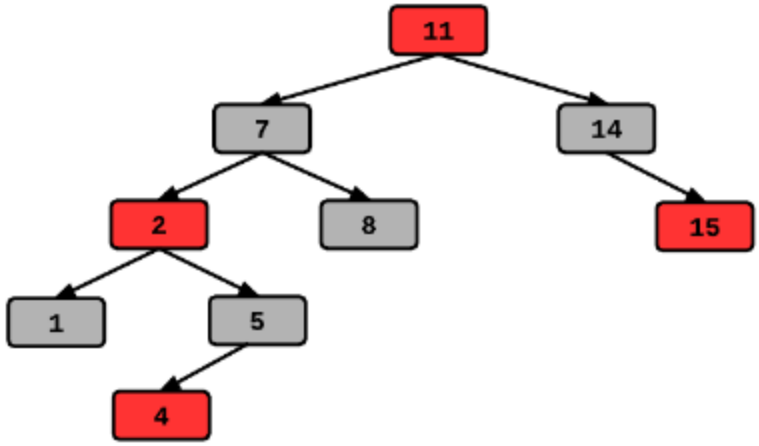


Figure 20. The color violations have been resolved in this tree, except for the violation that the root cannot be red. A rightRotate() about the root can resolve this issue.

9. **Line 3 *RBCase3Left()*:** Apply the *rightRotate()* algorithm to *x.parent.parent,* which is the 11 to generate the tree shown in Figure 21. All conditions on the red-black tree are now satisfied.

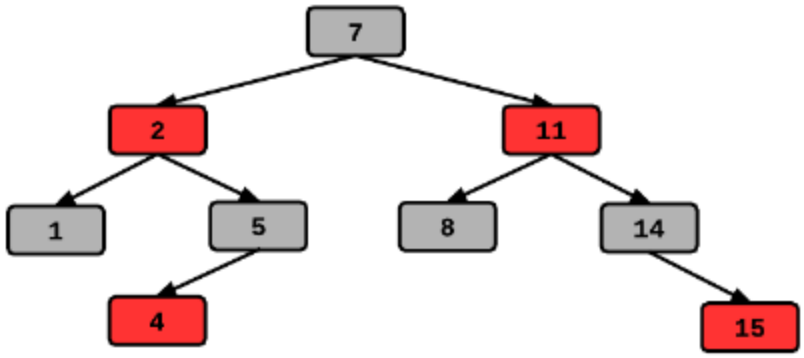


Figure 21. The final red-black tree after the right rotation about the root. The tree now satisfies all red-black tree constraints.

Example 3: Build a red-black tree from the following sequence
<15, 11, 14, 2, 1 >
of integers using the redBlackInsert() algorithm in Algorithm 11.3.

Call *redBlackInsert()* for each value in the sequence to add it to the tree. The *redBlackInsert()* algorithm calls *insertRB()* (Algorithm 11.6) to add the integer as a node to the tree, and then resolves red-black violations.

Steps:
1. **Line 1:** Call *insertRB(15)* to add the 15 as the root node and color it red. On **Line 17**, recolor the node to be black.
2. **Line 1:** Call *insertRB(11)* to add the 11 as the left child of 15 and color it red.
3. **Line 1:** Call insertRB(14) to add the 14 as the right child of the 11 and color it red. The 11 is also red, which violates the property that a red node can't have a red child. The current configuration of the tree is shown in Figure 22. The new node is labeled *x* and the *nullNode* sentinel nodes are shown as the smaller black nodes. This configuration is an example of Case 2: *x* is a right child and the "uncle" node, which is the right-child *nullNode* of the 15, is black.

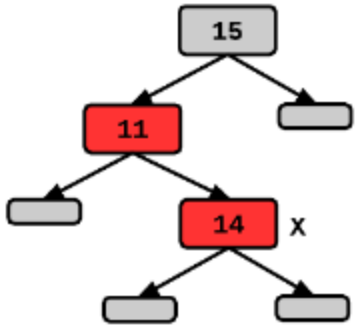


Figure 22. Red-black tree after three nodes, with values 15, 11, and 14, have been added. There is a violation of the red-black property that a red node can't have a red child between the 11 and the 14.

4. **Line 10-11:** Apply *leftRotate()* to *x.parent* to generate the tree shown in Figure 23. The tree is now an example of Case 3: *x* is a left child and the "uncle" node is black.

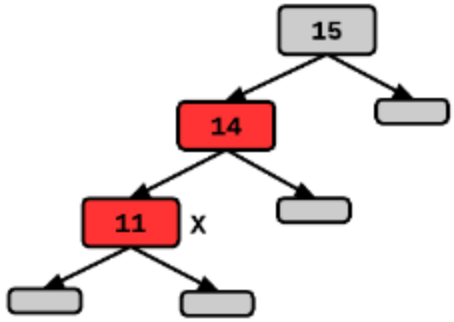


Figure 23. Result of a left rotation on the red-black tree in Figure 23. The node x is now a left child and the Case 3 rules can be applied.

6. **Line 13:** Call *RBCase3Left()* to recolor the *x.parent* and *x.parent.parent* nodes and apply the *rightRotate()* algorithm on *x.parent.parent*. The state of the tree after the recoloring is shown in Figure 24, and the result of the right rotation is shown in Figure 25.

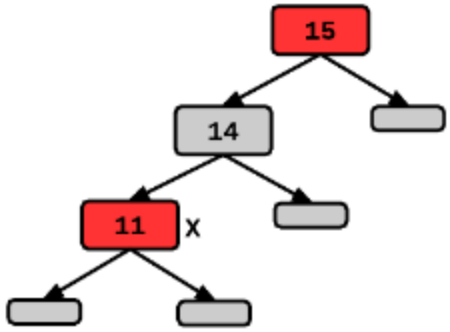


Figure 24. Result of applying the Case 3 recoloring to x.parent and x.parent.parent in the tree in Figure 23.

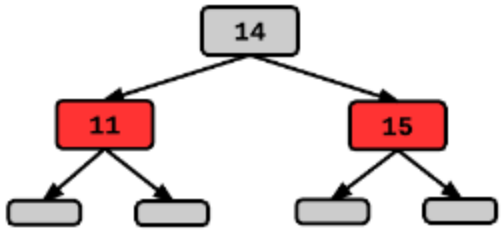


Figure 25. Result of applying the Case 3 right rotation on the 15, which was x.parent.parent in Figure 24.

7. **Line 1:** Call *insertRB(2)* to add the 2 as the left child of the 11. This operation produces the Case 1 configuration shown in Figure 26, where the “uncle” node, which is the 15, is red.

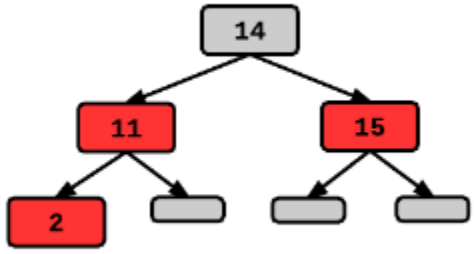


Figure 26. Adding the 2 to the red-black tree in Figure 25 generates a Case 1 configuration where the new node has a red "uncle".

8. **Line 6:** Call *RBCase1Left()* to recolor the *x.parent* and “uncle” nodes to be black, and the root node to be red. On **Line 17**, the root node is recolored to be black. The new tree is shown in Figure 27.

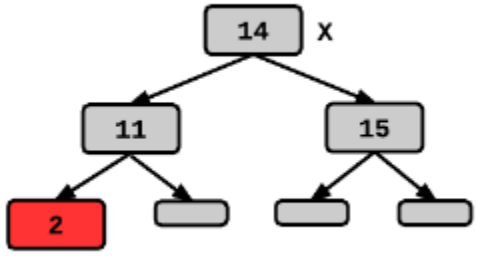


Figure 27. State of the red-black tree after Case 1 recoloring is applied and the red-black properties are restored.

9. **Line 1:** Call *insertRB(1)* to add the 1 as the left child of the 2, which creates a Case 3 violation where the 1 is a left child and the “uncle” is the *nullNode* right child of the 11, which is black.
10. **Line 13:** Call *RBCase3Left()* to recolor and apply a right rotation. The colors of *x.parent* and *x.parent.parent* are changed, which makes the 2 black and the 11 red. The right rotation on *x.parent.parent*, which is the 11, produces

the red-black tree shown in Figure 28. All values have been added to the tree and all red-black violations have been resolved.

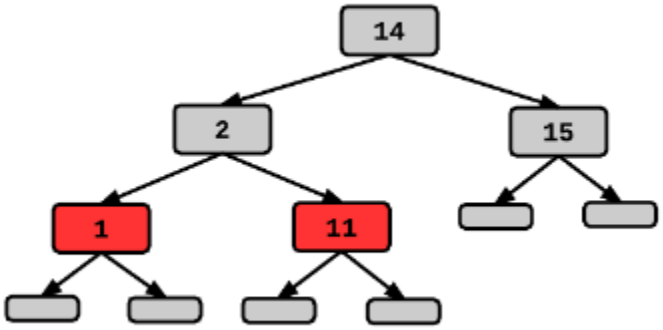


Figure 28. The Case 3 rules were applied to the tree in Figure 27 after the 1 was added as the left child of the 2. The 2 and the 11 were recolored, and a right rotation was applied to the 11 to produce this tree, which satisfies the red-black properties.

11.1.6 Deleting a node in a red-black tree

The algorithm for deleting a node from a red-black tree is the same as the algorithm for deleting a node from a regular BST, with the addition of steps to address violations introduced to the red-black properties.

The rules for deleting a node from a red-black tree are the same as the rules for deleting a node from a regular BST: • If the node has no children - delete the node.
• If the node has one child - replace the node with its remaining child.
• If the node has two children - replace the node with the minimum node in its right branch.

A violation can occur if the node's replacement is black, which can change the number of black nodes along a path from the root to a leaf node in the tree. In this situation, the violations need to be resolved by rebalancing the tree. For example, consider what will happen when the 10 is deleted from the red-black tree shown in Figure 29. The 12, labeled

as *y* in Figure 29, will take the 10's position in the tree, and the *x* will move into the position held by the *y*. This deletion and replacement will change the number of black nodes on the path from the 15 through the left sub-tree and require that the tree be rebalanced to address the violation.

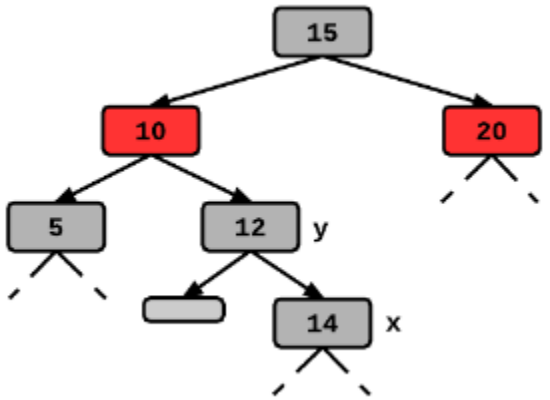


Figure 29. If the 10 is deleted from this red-black tree, it will be replaced with the 12. The number of black nodes on the path from the 15 to the leaf nodes in its left branch will change and introduce a violation of the red-black properties.

The algorithm to delete a node from a red-black tree is shown in Algorithm 11.7. The algorithm is broken down into cases for 0, 1, or 2 children and whether the node to delete is the root of the tree. An additional algorithm to rebalance the tree after the deletion is shown in Algorithm 11.8.

Algorithm 11.7. redBlackDelete(value) Delete a node from a red-black tree. Calls *rbBalance()* to resolve red-black violations caused by the delete operation.

Pre-conditions *value* is a valid node search value.
search() exists the returns a pointer to the node to delete.
treeMinimum() exists to find the minimum-valued node in a branch in the tree.
rbBalance() exists to resolve red-black violations.

Post-conditions The node with the specified value is deleted from the red-black tree and all red-black violations are resolved.

Note: This algorithm only handles the case where the node to delete is the left child of its parent. Additional steps are needed to handle the right-child case.

Algorithm redBlackDelete(value) 1. node = search(value) 2. nodeColor = node.color 3. if(node != root) 4. if(node.leftChild == nullNode and node.rightChild == nullNode) //no children 5. node.parent.leftChild = nullNode 6. x = node.parent.leftChild 7. else if(node.leftChild != nullNode and node.rightChild != nullNode) //two children 8. min = treeMinimum(node.rightChild) 9. nodeColor = min.color //color of replacement 10. x = min.rightChild 11. if (min == node.rightChild) 12. node.parent.leftChild = min 13. min.parent = node.parent 14. min.leftChild = node.leftChild 15. min.leftChild.parent = min 16. else 17. min.parent.leftChild = min.rightChild 18. min.rightChild.parent = min.parent 19. min.parent = node.parent 20. node.parent.leftChild = min 21. min.leftChild = node.leftChild 22. min.rightChild = node.rightChild 23. node.rightChild.parent = min 24. node.leftChild.parent = min 25. min.color = node.color //replacement gets nodes color 26. else //one child 27. x = node.leftChild 28. node.parent.leftChild = x 29. x.parent = node.parent 30. else 31. //repeat cases of 0, 1, or 2 children 32. //replacement node is the new root 33. //parent of replacement is nullNode 34. if nodeColor == BLACK 35. rbBalance(x) 36. delete node

The *rbBalance()* algorithm (shown in Algorithm 11.8) to restore the red-black properties is called on the *x* node, which is the replacement for the *min* node in the

redBlackDelete() algorithm. *rbBalance()* is called when the color of *min,* which is the minimum value in *node's* right branch, is black.

Algorithm 11.8. rbBalance(x) Restores the red-black properties to a tree following a node deletion.

Pre-conditions *x* points to the node that replaced the return value of *treeMinimum()* in the deletion.

Post-conditions Red-black properties restored to the tree.

Algorithm RBBalance(x) 1. while (x != root and x.color == BLACK) 2. if (x == x.parent.leftChild) 3. s = x.parent.rightChild 4. if (s.color == RED) //Case 1
5. s.color = BLACK
6. x.parent.color = RED
7. leftRotate(x.parent) 8. s = x.parent.rightChild 9. if (s.leftChild.color == BLACK and s.rightChild.color == BLACK) //Case 2
10. s.color = RED
11. x = x.parent 12. else if(s.leftChild.color == RED and s.rightChild.color == BLACK) //Case 3
13. s.leftChild.color = BLACK
14. s.color = RED
15. rightRotate(s) 16. s = x.parent.rightChild 17. else 18. s.color = x.parent.color //Case 4
19. x.parent.color = BLACK
20. s.rightChild.color = BLACK
21. leftRotate(x.parent) 22. x = root 23. else 24. //x is a right child 25. //exchange left and right 26. x.color = BLACK

In the *rbBalance()* algorithm there are four cases that can be observed regarding the color of the replacement node's

“sibling” node. The “sibling” node is identified on Line 3 of *rbBalance()*.

Case 1: Min’s replacement has a red “sibling”. Figure 30 shows an example of a Case 1 configuration. The replacement node is labeled *x* and the “sibling” node is labeled *s*.

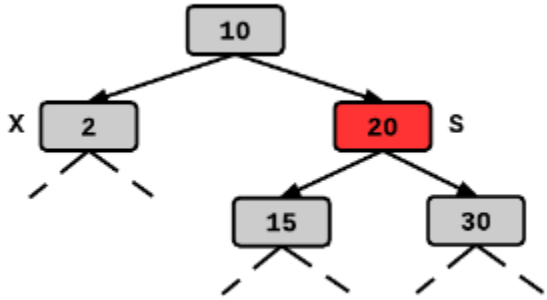


Figure 30. This red-black tree is an example of a Case 1 configuration in the rbBalance() algorithm. The replacement node, labeled X, has a red "sibling", labeled S.

Steps for resolving a Case 1 violation: 1. **Line 5-6:** The color of *S* and its parent are switched to produce the configuration shown in Figure 31.

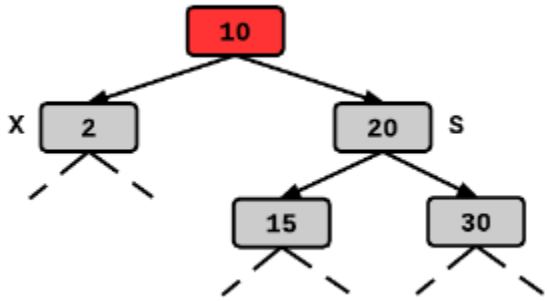


Figure 31. After Lines 5-6 in rbBalance(), the color of the S node and it's parent are swapped.

2. **Line 7-8:** Apply a left rotation to *x.parent* and re-assign *S* to point to the right child of *x’s* parent to produce the configuration shown in Figure 32.

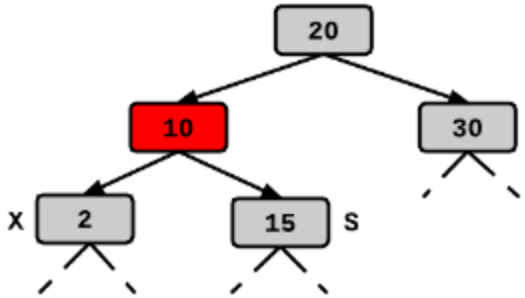


Figure 32. After Lines 7-8 in rbBalance(), the tree has been left rotated about x's parent and S points to the new right child of x's parent.

Case 2: Both of "sibling" *S's* children are black, and S is also black.

Case 2 evaluates the color of the left and right children of the "sibling" *S*. The Case 2 configuration can be evaluated after the Case 1 rebalancing or independent of Case 1. Figure 33 shows an example of a Case 1 configuration.

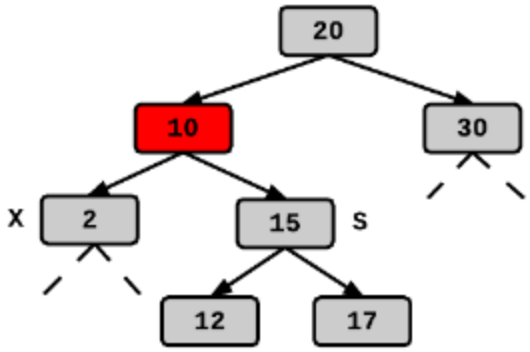


Figure 33. An example of a Case 2 configuration where the sibling S is black and has black children.

Steps to resolve a Case 2 violation: 1. **Line 10:** Recolor *S* to be red.
2. **Line 11:** Set *x* to point to its parent. The new tree configuration is shown in Figure 34.

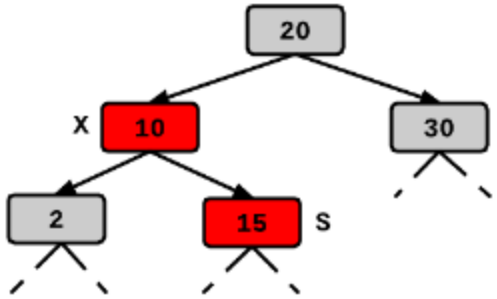


Figure 34. The Case 2 steps in rbBalance() recolor S and reset x to point to x.parent.

• **Line 24:** Recolor *x* to black, which produces the configuration shown in Figure 35.

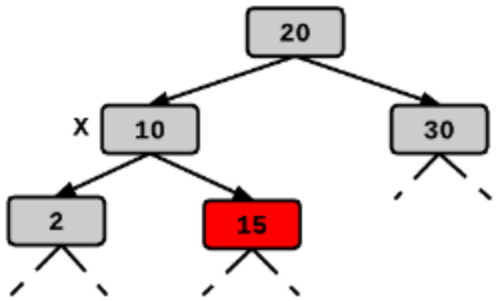


Figure 35. After the Case 2 algorithm is executed, S (which is the 15) is red.

Case 3: The "sibling" *S* is black and has a red left child and a black right child.

An example of a Case 3 configuration is shown in Figure 36. The replacement for the *min* node, labeled *x*, has a black sibling *S* with a red left child and a black right child.

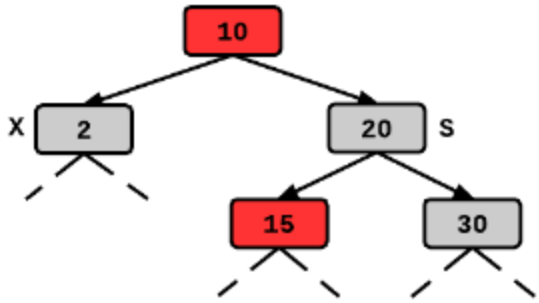


Figure 36. Example of a Case 3 configuration. The replacement node, labeled x, has a black "sibling", labeled S, with a red left child and a black right child.

Steps to resolve a Case 3 violation: 1. **Lines 13-14:** Recolor *S* and its left child to generate the configuration in Figure 37.

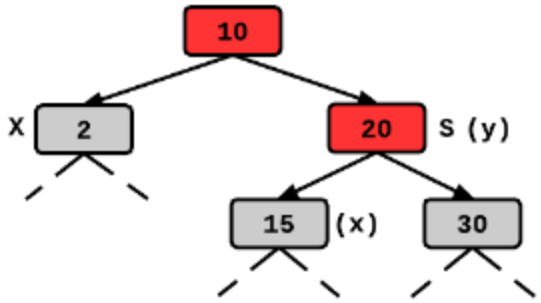


Figure 37. Configuration of the red-black tree after S and its left child are recolored. The (x) and (y) labels show the nodes are x and y in the right rotation, which is the next step in the algorithm.

2. **Line 15:** Apply a right rotation to *S*. The result is shown in Figure 38.

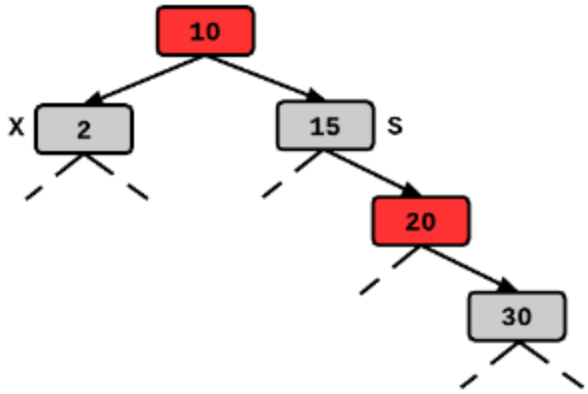


Figure 38. Red-black tree after the right rotation on S. The tree is now an example of Case 4: a black “sibling” S with a black left child and a red right child.

The Case 3 algorithm transforms the tree into a Case 4, where the “sibling” has a black left child and a red right child.

Case 4: Min's replacement's "sibling" is black and has a black left child and a red right child.
Steps to resolve a Case 4 violation: 1. **Lines 18-20:** Recolor *S*, its parent, and its right child produce the configuration in Figure 39.

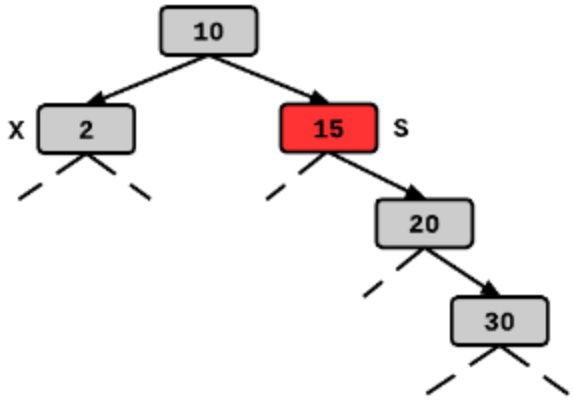


Figure 39. Configuration of the red-black tree after recoloring S, its parent, and its right child as part of resolving a Case 4 violation.

2. **Line 21:** Apply the *leftRotate()* algorithm to *x.parent* to produce the tree shown in Figure 40.

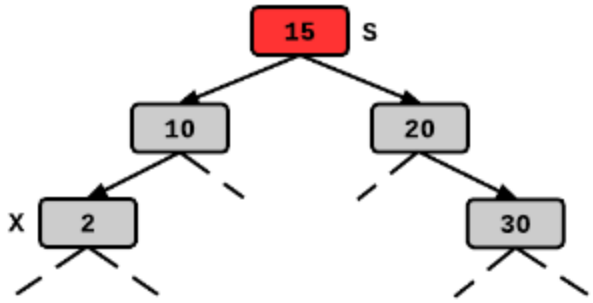


Figure 40. Red-black tree after the left rotation at the end of the Case 4 algorithm. If the S is the root of the tree, it will be recolored on the last line of the rbBalance() algorithm.

3. **Line 22-23:** Set *x* to the root of the tree to exit the while loop and ensure that the root of the tree is colored black on the last line of the *rbBalance()* algorithm.

Example 4: Delete the 8 from the red-black tree in Figure 41.

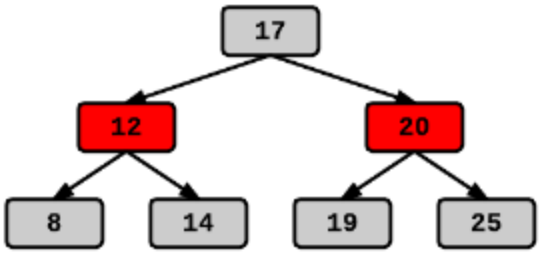


Figure 41. Deleting the 8 node from this red-black tree executes the portion of the delete algorithm for a node with zero children.

Steps for deleting a node with zero children 1. **Line 2:** The variable *nodeColor* is set to black.
2. **Line 5:** The left child of the 12 node is set to the null node (Figure 42).
3. **Line 6:** A variable *x* is created that is a null node. This *x* variable will be the input to the *rbBalance()* algorithm on **Line 35**.

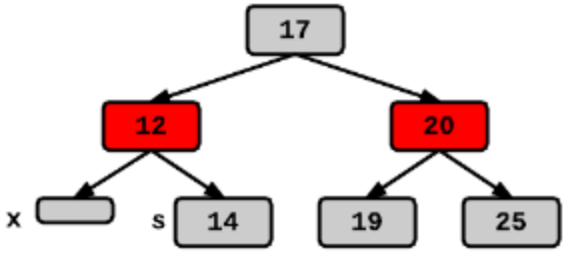


Figure 42. The 8 is replaced by a null node, which is then labeled x in the delete algorithm. The sibling of x is the 14.

4. **Line 3 rbBalance():** Identify the sibling of *x*, which is the 14 and labeled *s* in Figure 42.
5. **Line 10 rbBalance():** Recolor the sibling *s* red.
6. **Line 11 rbBalance():** Change *x* to point to *x.parent*, which is the 12 (Figure 43).

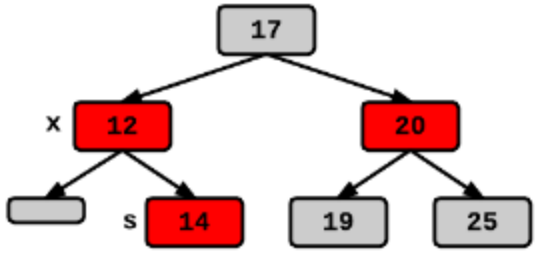


Figure 43. Recolor the sibling s to be red and change x to point to x.parent.

7. **Line 26 rbBalance():** Recolor *x* to be a black node (Figure 44).

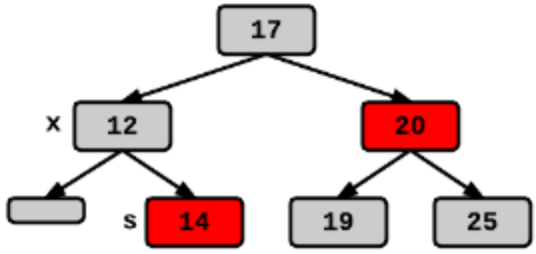


Figure 44. Recolor x to be a black node.

12 Graphs

The map of the Midwestern United States in Figure 1 shows the major cities between Denver and Kansas City and the roads that connect them. Anyone interested in travelling between any two cities on the map, would first identify the two cities and then identify the roads between them.

Figure 1. Map of the major roads between Denver and Kansas City.

Graphs provide a structure for representing connections between people, places, or things that captures the essence of the connections. A person wanting to between Denver and Kansas City would likely due so using major highways. They would want to identify the path between the two cities, including the distance and if the path goes through other locations along the way. They would be less interested in the smaller roads that connect the two places.

12.1 Adjacency Matrix

An adjacency matrix is a structure for representing direct connections between entities in a graph, such as locations. In a 2D matrix, these entities are listed on both the horizontal and vertical axis. If there is a direct connection between two entities, it means they are adjacent and there is a 1 at that location in the matrix. If there isn't a direct connection, they are not adjacent and there is a 0 at that location in the matrix.

Example 1: Generate an adjacency matrix for the cities shown in Figure 1 - Denver, Colorado Springs, Pueblo, Fort Collins, Lincoln, Omaha, Kansas City, Lawrence, and Wichita - showing which locations are directly connected by a major highway.

Steps:

1. Generate a blank 2D matrix that lists the locations on both axes. The vertical axis is the starting location and the horizontal axis is the destination (Figure 2).

	Denver	Colo Springs	Pueblo	Ft. Collins	Lincoln	Omaha	KC	Lawrence	Wichita
Denver									
Colo Springs									
Pueblo									
Ft. Collins									
Lincoln									
Omaha									
Kansas City									
Lawrence									
Wichita									

Figure 2. Empty adjacency matrix with the starting location on the vertical axis and the destination location on the horizontal axis.

2. Add a 1 to the matrix if two locations have a road between them and add a 0 to the matrix if they don't. The roads generate the adjacency matrix shown in Figure 3.

	Denver	Colo Springs	Pueblo	Ft. Collins	Lincoln	Omaha	KC	Lawrence	Wichita
Denver	0	1	0	1	1	0	0	1	1
Colo Springs	1	0	1	0	0	0	0	0	0
Pueblo	0	1	0	0	0	0	0	0	0
Ft. Collins	1	0	0	0	1	0	0	0	0
Lincoln	1	0	0	1	0	1	0	0	0
Omaha	0	0	0	0	1	0	1	0	0
Kansas City	0	0	0	0	0	1	0	1	1
Lawrence	1	0	0	0	0	0	1	0	1
Wichita	1	0	0	0	0	0	1	1	0

Figure 3. Adjacency matrix showing the locations connected by a major highway in the map shown in Figure 1. The 1 means there is a path directly between the locations and the 0 means there isn't a path between the locations.

• The first row in the matrix represents the scenario of starting in Denver and going directly to another location, without going through any other locations. There is a road from Denver to Colorado Springs, Ft. Collins, Lincoln, Lawrence, and Wichita. There is not a road from Denver to Pueblo, Omaha, and Kansas City, which doesn't mean that there's no way to go between Denver and these places, it only means that any path between these locations has to go through at least one other location.

• The second row in the matrix shows a starting location of Colorado Springs. There are roads from this location to Denver and Pueblo only.

• The third row in the matrix shows a starting location of Pueblo and the only direct connection is to Colorado Springs.

• The fourth row in the matrix shows a starting location of Ft. Collins, which connects to Denver and Lincoln only.

• The values on the matrix diagonal represent the condition of staying in the current location, e.g. going from Denver to Denver. In this example, it is assumed that this is not possible, and therefore, these values are all 0.

This adjacency matrix is symmetrical, which means that the path going from Denver to Lincoln is the same as the path going from Lincoln to Denver.

12.2 Graph Representation

The information in an adjacency matrix can be represented as a graph, where a graph structure is defined as $G = (V, E)$; graph G has a set of V vertices connected by a set of E edges. If the adjacency matrix has a 1 in a cell, then the graph has an edge between those two vertices.

Consider the graph G in Figure 4.

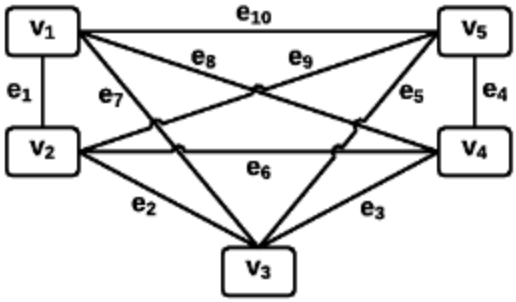


Figure 4. Graph G, with edges <e_1, ... e_n > and vertices < v_1, ... v_k >.

The edges E in the graph are labeled
< e_1, e_2, ... e_n >,
and the vertices V in the graph are labeled
< v_1, v_2, ... v_k >,
where n is the number of edges and k is the number of vertices. All of the vertices and edges make up the graph G:
G = { < v_1, v_2, ... v_k >, < e_1, e_2, ... e_n > }

The graph for the adjacency matrix in Figure 3 is shown in Figure 5.

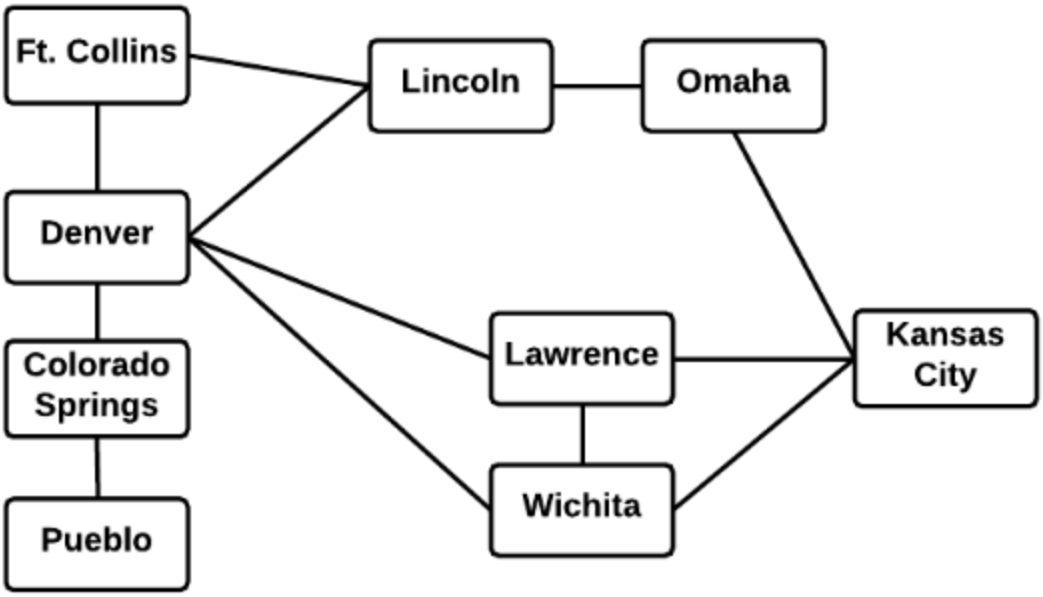


Figure 5. Graph constructed from the adjacency matrix shown in Figure 3. An edge between two vertices means there is a connection between them, e.g. there is an edge between Denver and Lincoln, but not between Lincoln and Lawrence.

For locations that don’t have an edge between them, there is a 0 in the adjacency matrix. Neither the placement of the vertices in the graph, nor the length of the edges has any meaning relative to the original map. This graph only captures the connections between vertices.

12.3 Adjacency-list Representation

Another method for storing graph data is to use an adjacency list instead of an adjacency matrix. In this approach, the vertices in the graph are stored in an array, and each vertex in the array contains a pointer to a list of its adjacent vertices. For example, an array of vertices for the road map would each contain a list of the vertices that each vertex connects to, as shown in the adjacency list in Figure 6. This list contains the same data shown in the adjacency matrix in Figure 3 and the graph in Figure 5. For example, the Denver vertex has five adjacent vertices - Colorado Springs, Ft. Collins, Lawrence, Lincoln, and Wichita stored in a list. The Colorado Springs vertex only has one adjacent vertex - Denver.

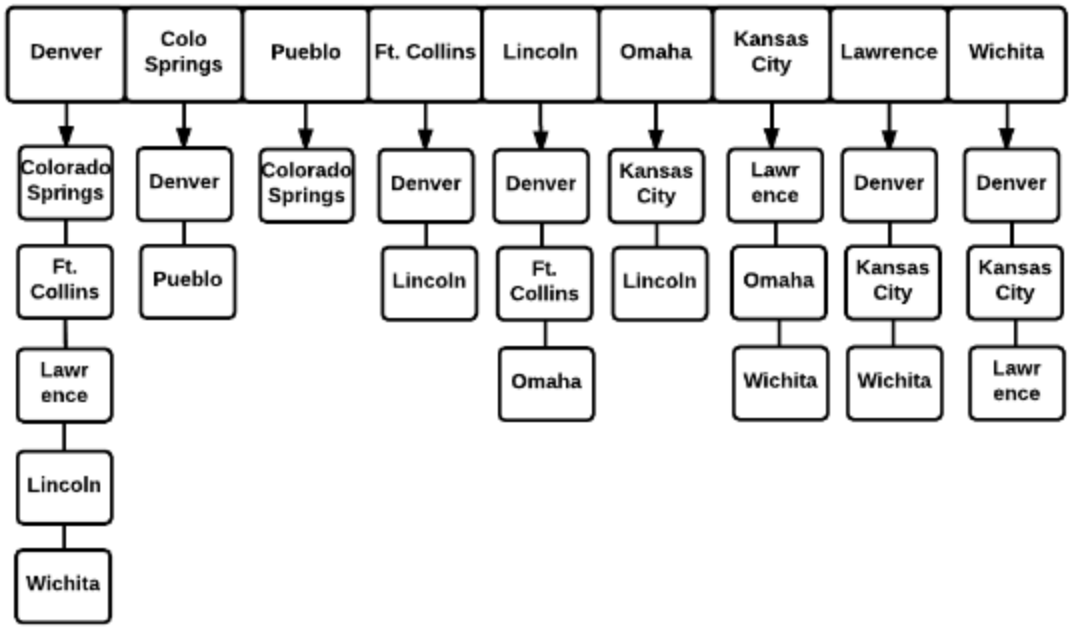


Figure 6. Adjacency list for the adjacency matrix in Figure 3. The vertices in the graph are stored in an array. Each vertex in the array stores a pointer to a list of vertices that it is adjacent to in the graph.

12.4 Directed and Undirected Graphs

12.4.1 Undirected Graph

The graph shown in Figure 5 is an example of an undirected graph. The edge between two vertices exists in both directions. For example, an edge between Denver and Lincoln also means there is an edge between Lincoln and Denver.

12.4.2 Directed Graph

In a directed graph, the edges between two vertices have a direction associated with them, and the edge may be different, or not exist, in one direction. To illustrate, assume a few of the roads in the Figure 1 map are converted to one-way roads, e.g there is a path from Denver to Lincoln, but not the other direction. An adjacency matrix or an adjacency list can capture the information in a directed graph. Just as in an undirected graph, a 1 in the adjacency matrix represents an edge in a direction, and a 0 represents no edge in a direction. The adjacency matrix won't be symmetrical in a directed graph if there are edges that only go in one direction.

Example 2: Generate the adjacency matrix and graph if there are three, one-way roads on the road map used in Example 1.

The three one-way roads are:
Lincoln to Denver
Omaha to Lincoln
Kansas City to Wichita

The adjacency matrix needs to reflect that there are no longer edges for:
Denver to Lincoln

Lincoln to Omaha
Wichita to Kansas City

A 0 is added to the matrix for those edges. The new adjacency matrix is shown in Figure 7. For each of the edges that no longer exist in the graph, there is a 0 in the matrix.

	Denver	Colo Springs	Pueblo	Ft. Collins	Lincoln	Omaha	KC	Lawrence	Wichita
Denver	0	1	0	1	0	0	0	1	1
Colo Springs	1	0	1	0	0	0	0	0	0
Pueblo	0	1	0	0	0	0	0	0	0
Ft. Collins	1	0	0	0	1	0	0	0	0
Lincoln	1	0	0	1	0	0	0	0	0
Omaha	0	0	0	0	1	0	1	0	0
Kansas City	0	0	0	0	0	1	0	1	1
Lawrence	1	0	0	0	0	0	1	0	1
Wichita	1	0	0	0	0	0	0	1	0

Figure 7. Adjacency matrix for Example 2 with three, one-way directed edges. There is an edge from Lincoln to Denver, but not Denver to Lincoln. There is also not an edge from Lincoln to Omaha and Wichita to Kansas City.

To draw a directed graph, add arrows to the edges between the vertices to indicate the direction of the edge. The directed graph for the adjacency matrix in Figure 6 is shown in Figure 8. For edges that go in both directions, there is an arrow at both ends of the edge. For example, there is an edge in both directions between Omaha and Kansas City, but the edge between Kansas City and Wichita only goes in one direction. Directed graphs can also be drawn with separate edges for each direction.

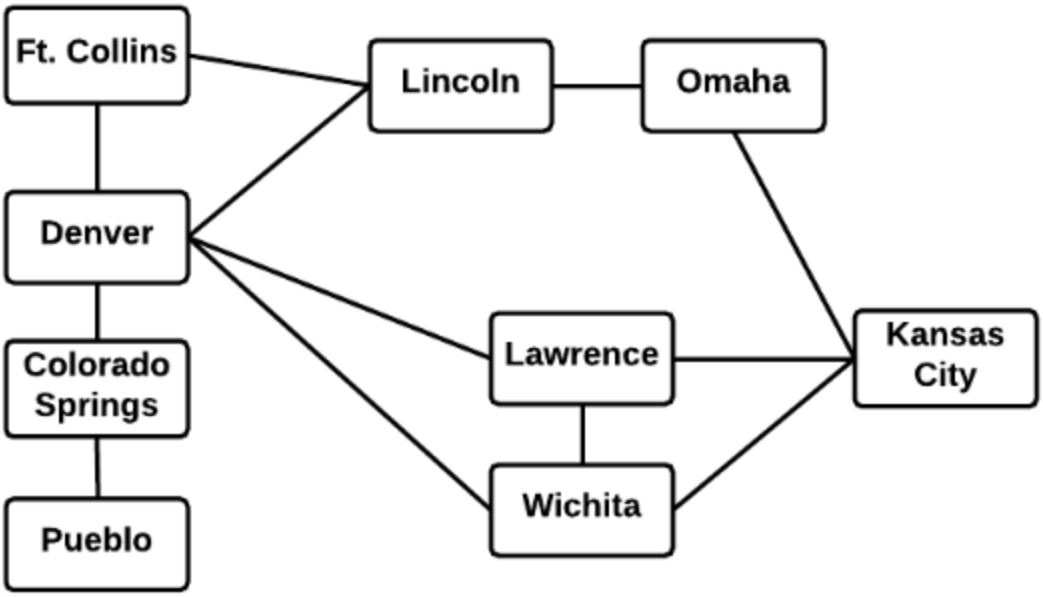


Figure 8. Example of a directed graph where an arrow shows the direction of the edge. For edges that go in both directions between two vertices, there is an arrow on both ends of the edge.

12.5 Weighted Graphs

In the examples so far, the edges represent a connection between two places, but do not contain any other information, such as the distance between the places. In a weighted graph, the edge has a weight that provides information about the connection, such as the distance, the cost of travel between vertices, or the flow of goods between two vertices. Using a weighted graph, questions such as, “What is the shortest distance between all vertices?” or “What is the cheapest path between two or more cities?” can be answered.

The approximate distances (according to Google Maps) between the road map locations in Figure 1 are shown in the adjacency matrix in Figure 9.

	Denver	Colo Springs	Pueblo	Ft. Collins	Lincoln	Omaha	KC	Lawrence	Wichita
Denver	0	70	-1	64	490	-1	-1	566	518
Colo Springs	70	0	44	-1	-1	-1	-1	-1	-1
Pueblo	-1	44	0	-1	-1	-1	-1	-1	-1
Ft. Collins	64	-1	-1	0	467	-1	-1	-1	-1
Lincoln	490	-1	-1	467	0	54	-1	-1	-1
Omaha	-1	-1	-1	-1	54	0	188	-1	-1
Kansas City	-1	-1	-1	-1	-1	188	0	41	200
Lawrence	566	-1	-1	-1	-1	-1	41	0	162
Wichita	518	-1	-1	-1	-1	-1	200	162	0

Figure 9. Distances between adjacent locations, according to Google Maps. These numbers are the edge weights in the graph. A weight of -1 means there is not an edge between those vertices.

There are a few changes to the adjacency matrix to represent the information in a weighted graph. The matrix now includes a 0 for the distance between the place and itself is 0. For vertices that don’t have a connecting edge,

such as downtown to the dorms, there is a -1 in the matrix. For all other entries in the matrix, the weight of the edge connecting those vertices is shown. As an example, the weight on the edge connecting Denver and Colorado Springs is 70. Figure 10 shows the graph for the adjacency matrix in Figure 9.

Using the edge weights, the final distance between any two places can be calculated. For example, to go from Denver to Kansas City by way of Lawrence, the distance is 566 + 41 = 607 miles. It's 566 miles from Denver to Lawrence and then 41 miles from Lawrence to Kansas City.

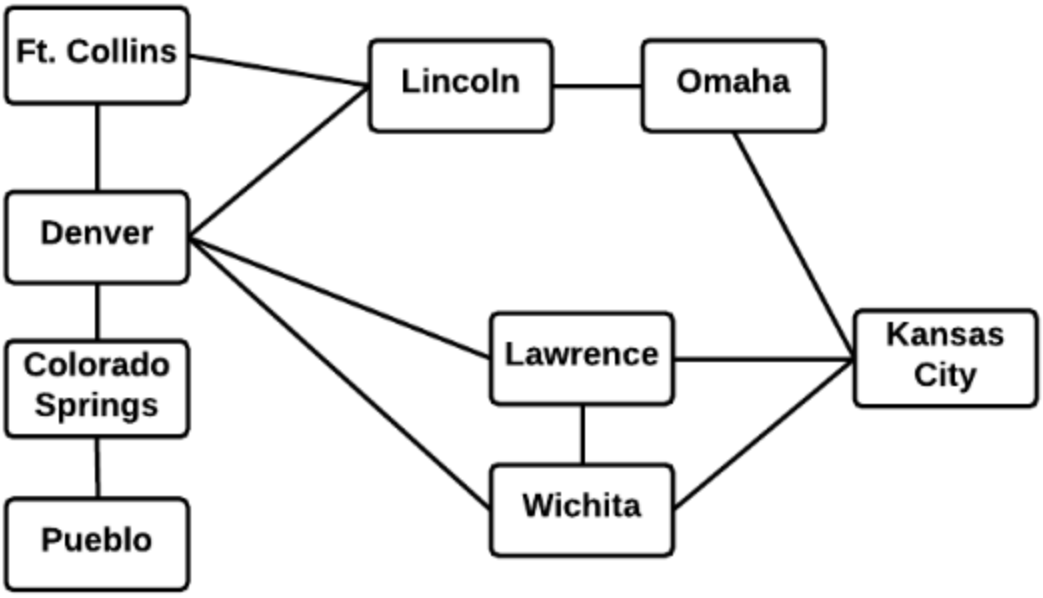


Figure 10. Weighted graph generated from the adjacency matrix in Figure 9. The edges with a weight of 0 are removed.

12.6 Graph ADT

In the graph ADT, the vertices in the graph are stored as a private variable. There are public methods to initialize the graph, insert and delete edges and vertices, print the graph, and search the graph. The edges are stored in an adjacency list for each vertex in the vertices variable, and therefore, don’t need to be represented separately as private variables in the graph ADT. A suggested set of minimum functionality is shown in ADT 12.1.

ADT 12.1. Graph Graph: 1. private: 2. vertices 3. public: 4. Init() 5. insertVertex(value) 6. insertEdge(startValue, endValue, weight) 7. deleteVertex(value) 8. deleteEdge(startValue, endValue) 9. printGraph() 10. search(value)

12.7 Implementing a Graph class in C++

12.7.1 Vectors

C++ has a container data type called a vector in the Standard Template Library that behaves like a linked list and an array. Elements in a vector can be indexed like elements in an array, and they can be added and removed one at a time like elements in a linked list without the developer having to explicitly handle expensive operations such as array shifting and doubling.

This graph implementation uses vectors instead of an array or a linked list to simplify the memory management of the graph.

To declare a vector variable:
vector<type> variable;
where *<type>* is the data type, such as **int**, **double**, or a user-defined type, and *variable* is the name of the vector variable.

12.7.2 Creating graph vertices and edges

In code, the graph can be represented in a Graph class:
class Graph{
private: //vertices and edges definition goes here public:
//methods for accessing the graph go here }

Each vertex in the graph is defined by a **struct** with two members: a *key* that serves as the key value for the vertex, and a vector *adjacent* to store the adjacency list for the vertex.

A vertex is defined as:

```
struct vertex{
std::string key; std::vector<adjVertex> adjacent; }
```

The *adjVertex* data type is also defined by a **struct** with two members: contains a pointer to the adjacent vertex *v,* and an integer *weight* that stores the edge weight between the two vertices.

```
struct adjVertex{
vertex *v; int weight; };
```

An empty vector of *vertex* can be created using the statement:

```
std::vector <vertex> vertices;
```

The *adjVertex* **struct** only stores the destination vertex in *v* because the origin vertex is stored in the *vertices* vector. In this design, each vertex in the graph has a vector of adjacent vertices that contains the vertex at the other end of the edge. The number of adjacent vertices can vary for each vertex in *vertices*. The size of the *adjacent* vector is also dynamic for each vertex.

A visual representation of the setup is shown in Figure 11.

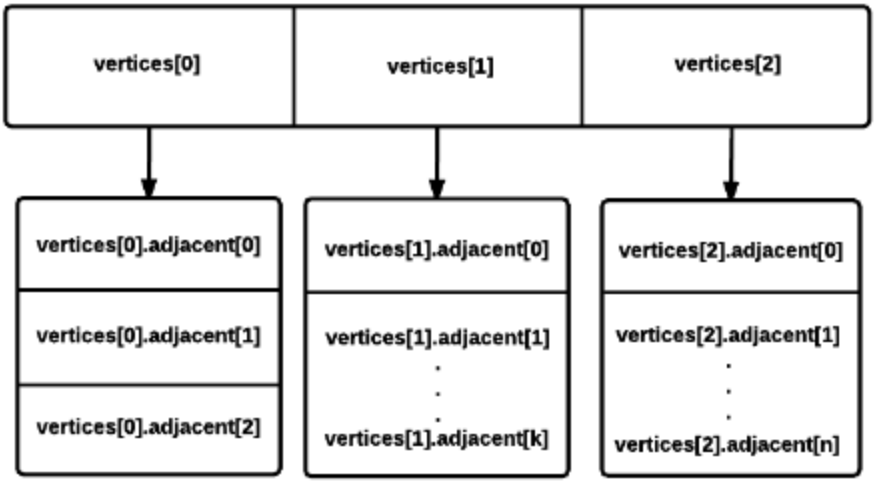


Figure 11. A visual representation of the vertices vector and the adjacent vector for each vertex. The vertices vector contains all of the

vertices in the graph. The adjacent vector is the adjacency list for each vertex.

In Figure 11, *vertices[0]* has only 3 adjacent vertices. For *vertices[1]* and *vertices[2]*, there are *k* and *n* adjacent vertices, respectively. This diagram illustrates that each *vertices[i]* can have a different number of adjacent vertices stored in its *adjacent* vector.

12.7.3 Insert vertex

Adding vertices to the graph is handled through a public method that takes the vertex *key* value as an argument. The *insertVertex()* algorithm is shown in Algorithm 12.1. A vertex with the specified key value is added to the vertices variable. Memory for the vertex is allocated when it is added to *vertices*.

Algorithm 12.1. insertVertex(value) Add a vertex with the specified value to a graph.

Pre-conditions *value* is a valid key value of the same type as the *key* parameter in *vertex*.
vertices is an array of graph vertices.

Post-conditions Vertex added to *vertices* if it doesn't already exist.

Algorithm insertVertex(value) 1. found = false 2. for x = 0 to vertices.end 3. if(vertices[x].key == value) 4. found = true 5. break 6. if(found == false) 7. vertex.key = value 8. vertices.add(vertex)

The *insertVertex()* algorithm is shown in C++ in Code 12.1, which uses a vector to store the vertices. In the *insertVertex()* method, a **bool** called *found* controls the search of the *vertices* vector. If the *key* already exists in the vector, set *found* to true and break out of the loop. If the

key isn't found in *vertices*, it is added to *vertices* using the vector *push_back()* method.

Code 12.1. insertVertex(string value) void Graph::insertVertex(string value){

```
1. bool found = false; 2. for(int i = 0; i < vertices.size();
i++){
3. if(vertices[i].key == value){
4. found = true; 5. cout<<vertices[i].key<<" found."
<<endl; 6. break; 7. }
8. }
9. if(found == false){
10. vertex v; 11. v.key = value; 12. vertices.push_back(v);
13. }
14. }
```

The *insertVertex()* method can be called as follows:
Graph g; g.insertVertex("Boulder");
to create an instance of Graph, called *g*, and add a vertex with the *key* "Boulder".

12.7.4 Insert edge

After vertices have been added to the graph, edges can be added to connect them. The *insertEdge()* algorithm, shown in Algorithm 12.2, takes the two *key* values of the vertices to connect and the weight of the edge between them, and adds an element to the *adjacent* list for the source vertex. The *insertEdge()* algorithm first checks that the two *key* parameters, *v1* and *v2*, exist in *vertices*. If they are in *vertices*, then the *adjacent* vector for the source vertex *v1* is updated to include the new edge with a *weight* and a pointer to the destination vertex *v2*.

Algorithm 12.2. insertEdge(v1, v2, weight) Add an edge between vertices *v1* and *v2* with the specified weight.

Pre-conditions *v1* and *v2* exist in the graph and there isn't an existing edge from *v1* to *v2*.

Post-conditions Entry added to the adjacency list for *v1* connecting it to *v2* with the specified weight.

Algorithm insertEdge(v1, v2, weight) 1. for x = 0 to vertices.end 2. if(vertices[x].key == v1) 3. for y = 0 to vertices.end 4. if(vertices[y].key == v2 and x != y) 6. adjacent.vertex = vertices[y]
7. adjacent.weight = weight 8. vertices[x].adjacent.add(adjacent)

The *insertEdge()* algorithm is shown in C++ in Code 12.2, which uses a vector to store the adjacency list of vertices. If both vertices are found in vertices, then the *adjacent* vector is updated to add an edge using the vector *push_back()* method.

Code 12.2. insertEdge(string v1, string v2, int weight) void Graph::insertEdge(string v1, string v2, int weight){

```
1. for(int x = 0; x < vertices.size(); x++){
2. if(vertices[x].key == v1){
3. for(int y = 0; y < vertices.size(); y++){
4. if(vertices[y].key == v2 && x != y){
5. adjVertex av; 6. av.v = &vertices[y]; 7. av.weight =
weight; 8. vertices[x].adjacent.push_back(av); 9. }
10. }
11. }
12. }
13. }
```

• On Lines 1-2 of Code 12.2, the *vertices* vector is checked for a *key* value that matches *v1*, and if it is found, then on Lines 3-4, the *vertices* vector is checked again for *v2*.

• On Lines 5-8, the destination vertex *vertices[y]* is added to the adjacency list for the source vertex *v1*. The *adjVertex* **struct** stores a pointer to the vertex, which is why the address of *vertices[y]* is used on Line 6.

The *insertEdge()* method adds an edge in one direction only. In an undirected graph, the method would need to be called twice with the source and destination vertices swapped to add the edge between two vertices in both directions. For example, to add an undirected edge between *Boulder* and *Denver*,
g.insertEdge("Boulder", "Denver", 30);
g.insertEdge("Denver", "Boulder", 30);
calls *insertEdge()* the first time with *Boulder* as the source and *Denver* as the destination and calls *insertEdge()* the second time with *Denver* as the source and *Boulder* as the destination.

12.7.5 Printing the graph

Printing the graph vertices and adjacent vertices is a simple way to verify that the graph is set up as expected. To print the graph, traverse the *vertices* variable, and print all elements of the *adjacent* variable for each vertex. Two loops are needed, one for the elements in *vertices*, and one for the elements of adjacent for each vertex in vertices. An algorithm to print the graph is shown in Algorithm 12.3 and the C++ code for the algorithm is shown in Code 12.3.

Algorithm 12.3. printGraph() Display the vertices and the adjacent vertices for each vertex in the graph.

Pre-conditions None
Post-conditions *vertices* and adjacent vertices displayed.

Algorithm printGraph() 1. for x = 0 to vertices.end 2. print(vertices[x].key) 3. for y = 0 to vertices[x].adjacent.end

```
4. print(vertices[x].adjacent[y].vertex.key)
```

Code 12.3. printGraph() void Graph::printGraph(){

```
1. for(int x = 0; x < vertices.size(); x++){
2. cout<<vertices[x].key<<"-->"; 3. for(int y = 0; y <
vertices[x].adjacent.size(); y++){
4. cout<<vertices[x].adjacent[y].v->key<<" "; 5. }
6. cout<<endl; 7. }
8. }
```

• On Line 3 of Code 12.3, *vertices[x].adjacent.size()* gets the size of the adjacency list, stored as the vector *adjacent*, for *vertices[x]*.

• On Line 4, the *key* of the vertex stored in the adjacency list is printed. The *v->key* notation is needed because *v* is a pointer to an existing element in *vertices*, and *v->key* dereferences the pointer and gets the *key* value of that *vertices* element.

12.7.6 Searching a graph

The *search()* algorithm in Algorithm 12.4 takes the *key* value to search for as a parameter and traverses *vertices* for a vertex that contains that value as its key. The algorithm returns the vertex where the value is found.

Algorithm 12.4. search(value) Returns the vertex with a *key* value that matches the search value.

Pre-conditions *value* is a valid search parameter with the same type as the *key* in *vertices*
Post-conditions Returns the vertex in *vertices* where *vertex.key = value.* Returns NULL if the value isn't found.

Algorithm search(value) 1. for x = 0 to vertices.end 2. if vertices[x].key == value 3. return vertices[x]

4. return NULL

12.8 Graph traversal algorithms

Graph traversal algorithms reveal features of the information stored in the graph by visiting the vertices in a specified order. The appropriate traversal algorithm to use depends on the question that needs to be answered about the graph.

Some questions to answer about graphs include the following:

- Does a vertex with a specified value exist in a graph?
- How many adjacent vertices does a particular vertex have?
- Is there a path between two vertices?
- What is the shortest path between two vertices?

12.8.1 Traversing graph edges

Each edge in a graph can be thought of as a step between two vertices in the graph. Counting the number of steps between two vertices is the same as counting the number of edges between the vertices. Vertices that are one step from another vertex are the adjacent vertices; they are connected directly by an edge. The vertices that are two steps away are the vertices that can be reached by traversing two edges, which requires going through an intermediate vertex. Vertices that are three steps apart are separated by two intermediate vertices, and so on.

Example 5: Using the graph in Figure 12, determine which vertices are one, two, and three steps from vertex A?

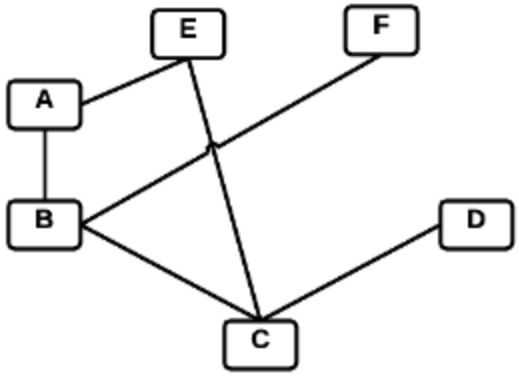


Figure 12. Graph for Example 5. Determine which vertices are 1, 2, and 3 steps from vertex A.

In this example, the adjacent vertices to *A* are *B* and *E*. The vertices that are two steps from *A* are *C* and *F*, which can be reached by going *A*-*B*-*C*, *A*-*E*-*C*, or *A*-*B*-*F*. The vertex that is three steps from *A* is *D,* which can be reached by going *A*-*B*-*C*-*D* or *A*-*E*-*C*-*D*.

Another approach to help visualize graph traversals is to view the graph is as a tree rooted at the starting vertex. In this approach, *A* is the root of the tree, and the vertices adjacent to *A* are its children. The vertices two steps from *A* are its children's children, and so on. Figure 13 shows the graph in Figure 12 represented as a tree. The vertices *B* and *E* are *A's* children. From vertex *B*, there is an edge back to vertex *A* and an edge to vertices *C* and *F*. Edges are only shown once in this tree, which is why the edges back to a previous vertex are not represented. From the vertex *E*, there is only one option, vertex *C*, and from there, there is also just one connecting vertex, which is vertex *D*.

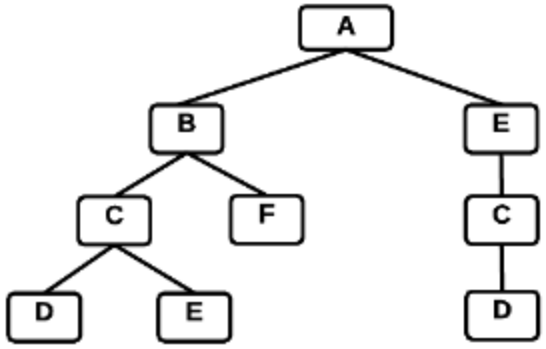


Figure 13. Shown here is the graph in Figure 12 represented as a tree rooted at vertex A. The children of each vertex in the tree are the adjacent vertices to that vertex in the graph. For example, B and E are adjacent to A.

12.8.2 Breadth-first search

Redrawing the graph as a tree can make it easier to examine the order that vertices are traversed with a particular search algorithm. For example, in the tree in Figure 13, vertices could be examined going across one level in the tree before the vertices at deeper levels in the tree are evaluated. In this case, the children of the root, which are vertices *B* and *E*, would be evaluated before either of *B* or *E's* children, vertices *C* and *F,* are evaluated.

Evaluating the vertices at the same level of a graph before evaluating nodes at deeper levels is a *breadth-first* evaluation. A search algorithm that evaluates nodes in a breadth-first ordering is called a breadth-first search. Figure 14 shows the order that nodes are evaluated in a breadth-first evaluation. The number assigned to the node is the evaluation order, e.g. the root is evaluated first, followed by the root's children going from left to right or right to left.

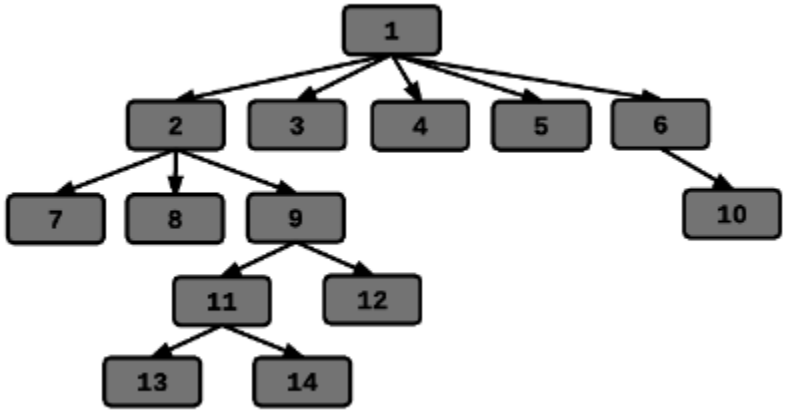


Figure 14. Search order for the breadth-first search algorithm. Nodes are evaluated across one level in the tree before moving to deeper levels in the tree.

The root in this tree is the starting vertex in a graph. The children of the root, which are all vertices adjacent to the root vertex in a graph, are evaluated next. Then, the children's children, which are all vertices two steps from the root in a graph, are evaluated. Each level in the tree is the same as one edge in a graph. For example, the node assigned the number 14 in the tree would be four edges away from the node assigned the number 1 in the tree if the two nodes were in a graph.

In a breadth-first search of the graph, the objective is to find a value in the graph using the breadth-first evaluation order just described. Each vertex is visited exactly once. The search ordering is often controlled using a queue to store the vertices. When a vertex is visited, its children are added to the queue. When a vertex is dequeued and evaluated, its children are enqueued. This process continues until there are no vertices left to evaluate and the queue is empty.

The *breadthFirstTraversal()* algorithm in Algorithm 12.5 evaluates an entire graph using a breadth-first ordering.

The *vertex* **struct** includes an additional member, called *visited,* that tracks if the vertex has been evaluated. Only vertices where *visited = false* are added to the queue. The *adjVertex* **struct** previously introduced is unchanged.

```
struct vertex{
std::string key; std::vector<adjVertex> adjacent; bool
visited; }
```

Algorithm 12.5. breadthFirstTraversal(value) Print the key values of the vertices in the graph in a breadth-first order, starting at the vertex where *vertex.key = value*.

Pre-conditions *value* is a valid key value for a vertex in the graph.
search() algorithm exists to find the vertex where *vertex.key = value.*
visited property initialized to false for all vertices
Post-conditions The vertices in the graph are displayed in breadth-first order from the starting vertex.

Algorithm breadthFirstTraversal(value) 1. vertex = search(value) 2. vertex.visited = true 3. queue.enqueue(vertex) 4. while(!queue.isEmpty()) 5. n = queue.dequeue() 6. for x = 0 to n.adjacent.end 7. if(!n.adjacent[x].visited) 8. n.adjacent[x].visited = true 9. print(n.adjacent[x].v.key) 10. queue.enqueue(n.adjacent[x].v)

Example 6: Using the graph in Figure 15, show the order that vertices are enqueued and dequeued in the breadthFirstTraversal() algorithm, starting from vertex A.

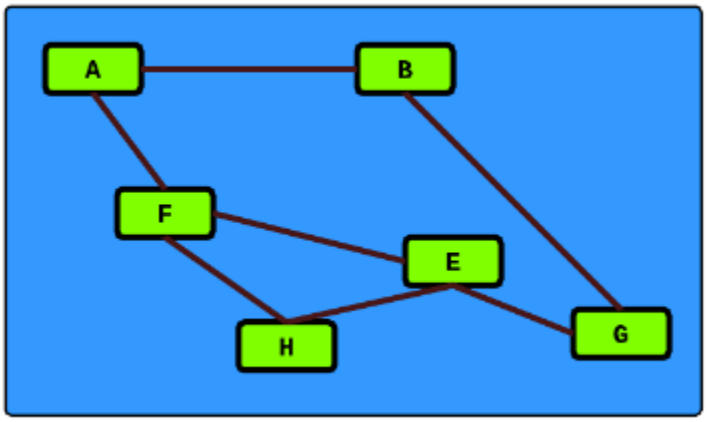


Figure 15. Show the order that the vertices in this graph are enqueued and dequeued in the breadthFirstTraversal() algorithm, starting from vertex A.

Steps:

1. **Line 1:** the *visited* property for the starting vertex *A* is set to true.
2. **Line 2:** the starting vertex *A* is added to the queue, and then dequeued on Line 4 for evaluation.
3. **Lines 5-6:** the loop checks the vertices adjacent to *A*, which are *B* and *F*, for whether they have already been visited. The *visited* property for both vertices is false.
4. **Line 7:** the adjacent vertices *B* and *F* are marked as *visited*.
5. **Line 9:** vertex *B* is added to the queue, and then vertex *F* is added to the queue the next time through the loop.
6. Back to **Line 3**, the **while** conditional checks if the queue is empty. It isn't, since *B* and *F* were just added.
7. **Line 4:** vertex *B* is dequeued. The vertices adjacent to *B*, which are *A* and *G*, are checked for whether they have been visited. Only the *G* has not been visited.
8. **Line 9:** the vertex *G* is added to the queue.
9. Back to **Line 3**, the **while** conditional checks if the queue is empty. It isn't, since it contains the *F* and *G*.
10. **Line 4:** vertex *F* is dequeued. The vertices adjacent to *F* are *A*, *E*, and *H*. The *E* and *H* have not been visited.

11. **Line 9:** the vertices *E* and *H* are added to the queue; one vertex is added each time through the loop.
12. Back to **Line 3**, the queue still isn't empty, since it contains the *G*, *E*, and *H*.
13. **Line 4:** the vertex *G* is dequeued, All of *G's* adjacent vertices have been visited, so there is nothing to enqueue.
14. Back to **Line 3**, the queue still isn't empty, since it contains the *E* and *H*.
15. **Line 4:** the vertex *E* is dequeued. All of *E's* adjacent vertices have been visited, so there is nothing to enqueue.
16. Back to **Line 3**, the queue still isn't empty, since it contains the *H*.
17. **Line 4:** the vertex *H* is dequeued. All of *H's* adjacent vertices have been visited, so there is nothing left to enqueue.
18. The vertex *H* was the last vertex in the queue. On **Line 3**, the conditional is false and the *breadthFirstTraversal()* routine exits.

The evaluation order of the graph vertices in Figure 15., viewed as a tree, is shown in Figure 16. The vertex *A* is evaluated first, followed by *B, F, G, E,* and *H* in that order.

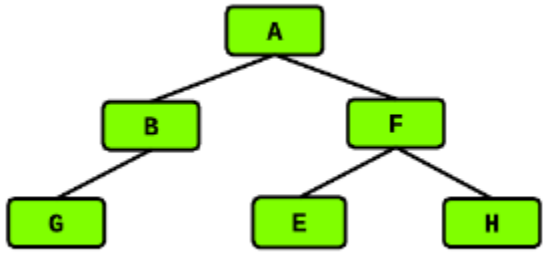


Figure 16. The evaluation order of the graph vertices in Figure 15. using the breadthFirstTraversal() algorithm. The vertex A is evaluated first, followed by B, F, G, E, and H.

12.8.3 Shortest distance in an unweighted graph

The *breadthFirstTraversal()* algorithm traverses the graph, but it doesn't provide information about the graph, such as the distance between any two vertices or whether a value exists in the graph. The algorithm can be modified to search for a specified vertex, and calculate the number of edges traversed to that vertex from the starting vertex. A *distance* parameter is added to the *vertex* **struct** to store the number of edges. Each time a vertex is enqueued, meaning that an edge was traversed to reach the vertex, its distance is incremented by 1.

```
struct vertex{
std::string key; std::vector<adjVertex> adjacent; bool
visited; int distance; }
```

In the *breadthFirstSearch()* algorithm in Algorithm 12.6, the arguments to the routine are the value of the starting vertex and the value to search for are arguments to the algorithm. A *search()* algorithm (Algorithm 12.4) is needed to support *breadthFirstSearch()*.

Algorithm 12.6. breadthFirstSearch(startValue, searchValue) Calculates the distance between the *startValue* and *searchValue* vertices, and returns the *searchValue* vertex.

Pre-conditions *startValue* and *searchValue* are valid search parameters with a type that matches the *key* value of the vertices in the graph.
search() takes *startValue* as a parameter and returns the vertex in the graph with that *key* value.
visited initialized to false for all vertices in the graph.
Post-conditions Returns the vertex with a *key* that matches the *searchValue*. Included in the vertex is the shortest distance back to the *startValue* vertex.
Returns NULL if the value isn't found.

Algorithm breadthFirstSearch(startValue, searchValue) 1. vertex = search(startValue) 2. vertex.visited = true 3. vertex.distance = 0
4. queue.enqueue(vertex) 5. while(!queue.isEmpty()) 6. n = queue.dequeue() 7. for x = 0 to n.adjacent.end 8. if(!n.adjacent[x].v.visited) 9. n.adjacent[x].v.distance = n.distance + 1
10. if(n.adjacent[x].v.key == searchValue) 11. return n.adjacent[x].v 12. else 13. n.adjacent[x].v.visited = true 14. queue.enqueue(n.adjacent[x].v) 15. return NULL

The *breadthFirstSearch()* algorithm will return the vertex if it's found and NULL if it's not. Stored in the vertex is the distance back to the starting vertex.

Example 7: What is the shortest distance between vertices A and G in the graph in Figure 17 using Algorithm 12.6?

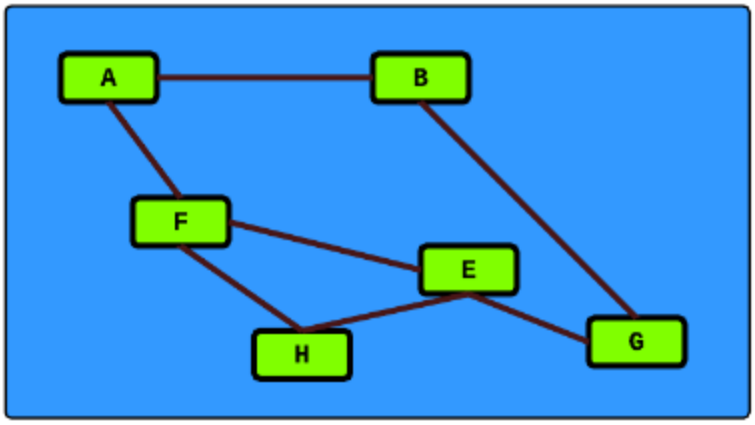


Figure 17. Find the shortest path in this graph between vertex A and the other vertices in the graph, where the shortest path is the path that covers the fewest edges.

The primary difference in the *breadthFirstTraversal()* and *breadthFirstSearch()* algorithms is the distance calculation.

Steps:
1. **Line 1:** Identify the starting vertex using *search()*.
2. **Lines 2-3:** Initialize the distance to the starting vertex to 0, and mark the vertex *visited*.
3. **Line 4:** Enqueue the starting vertex *A*.
4. **Line 6:** Dequeue the starting vertex *A*.
5. **Lines 7-14:** Examine the vertices adjacent to *A*, which are *B* and *F*. If *visited* is false for any vertex, mark it *visited* and add it to the queue.
6. **Line 8:** Check if vertex *B* has been visited.
7. **Line 9:** Store the distance to vertex *B* as the (distance to *A*) + 1. The distance to *A* is 0, so the distance to *B* is 1.
8. **Line 10:** Check if vertex *B* is the search value, which it is not.
9. **Lines 13-14:** Mark vertex *B* as visited and add it to the queue.
10. **Repeat lines 8-14 for vertex *F*.**
11. **Line 6:** Dequeue vertex *B*. The only vertex adjacent to *B* that hasn't been visited is *G*, which is also the value being searched for.
12. **Line 9:** Calculate the distance to *G,* which is (distance to *B*) + 1, which makes the distance 2.
13. **Line 10:** Check if *G* is the value being searched for, and since it is, the vertex is returned on Line 11.

12.8.4 Breadth-first shortest path

The distance between two vertices in an unweighted graph is the number of edges traversed to go from one vertex to the other vertex. The path between two vertices is the list of the vertices visited on the path between two vertices. In Example 7, the shortest distance between *A* and *G* is 2, and the shortest path is *A*-*B*-*G*. To find the path, information about the vertices visited needs to be stored along with the distance.

There are two options commonly used for storing path information.
1. Create an array to store the index of the parent vertex, and then trace back through the array to re-create the path.
2. Store a pointer to the parent vertex in each vertex, and follow the pointers back to the root vertex to re-create the path.

Storing a parent pointer The *breadthFirstSearch()* algorithm can be easily modified to store path information using a pointer to the parent vertex. The *vertex* **struct** is modified to include a *vertex *parent* that is initially NULL and then assigned during the search.

struct vertex{
1. std::string key; 2. std::vector<adjVertex> adjacent; 3. bool visited; 4. int distance; 5. vertex *parent; };
The *breadthFirstSearch()* algorithm in Algorithm 12.7 shows how this approach is used to store path information between two vertices in a graph. The algorithm takes the value of the starting vertex and the destination vertex as arguments. On **Line 11** of the algorithm, the parent of the vertex is assigned.

Algorithm 12.7. breadthFirstSearch(startValue, endValue) Find the path and the distance from the starting vertex to the destination vertex.

Pre-conditions *startValue* is valid vertex *key* value.
endValue is a valid vertex key value.
search() algorithm exists to identify starting vertex in the graph.

Post-conditions Returns the vertex with a *key* that matches the *endValue*. Included in the vertex is the shortest distance back to the *startValue* vertex. All vertices on the

shortest path have a *parent* property, which points to the previous vertex in the path.

Algorithm breadthFirstSearch(startValue) 1. vertex = search(startValue) 2. vertex.visited = true 3. vertex.distance = 0
4. queue = new queue() 5. queue.enqueue(vertex) 6. while(!queue.isEmpty()) 7. n = queue.dequeue() 8. for x = 0 to v.adjacent.end 9. if(!n.adjacent[x].v.visited) 10. n.adjacent[x].v.distance = n.distance + 1
11. n.adjacent[x].v.parent = n 12. if(n.adjacent[x].v.key == endValue) 13. return n.adjacent[x].v 14. else 15. n.adjacent[x].v.visited = true 16. queue.enqueue(n.adjacent[x].v) 17. return NULL

Storing a parent index Another approach for storing path information is to assign each vertex in the graph an integer index and store the parent index of each vertex in an array. The *vertex* **struct** is modified to assign each vertex an integer *ID* that will serve as its array index.

struct vertex{
1. std::string key; 2. std::vector<adjVertex> adjacent; 3. bool visited; 4. int distance; 5. int ID; };
A *breadthFirstSearch()* algorithm that finds the shortest path from a starting vertex to all other vertices in the graph is shown in Algorithm 12.8. This approach is slightly different than the approach in Algorithm 12.7 in that it stores the shortest path between the starting vertex and all other vertices in the graph.

Algorithm 12.8. breadthFirstSearch(startValue) Find the path and the distance from the starting vertex to all other vertices in the graph.

Pre-conditions *previous* array exists that is same size as the number of vertices in the graph. All previous values

initialized to -1.
startValue is valid vertex *key* value.
search() algorithm exists to identify starting vertex in the graph.

Post-conditions *previous* populated with parent ID for all vertices on the path from the starting vertex.

Algorithm breadthFirstSearch(startValue) 1. vertex = search(startValue) 2. vertex.visited = true 3. vertex.distance = 0
4. queue = new queue() 5. queue.enqueue(vertex) 6. while(!queue.isEmpty()) 7. n = queue.dequeue() 8. for x = 0 to v.adjacent.end 9. if(!n.adjacent[x].v.visited) 10. n.adjacent[x].v.distance = n.distance + 1
11. previous[n.adjacent[x].v.ID] = n.ID
12. n.adjacent[x].v.visited = true 13. queue.enqueue(n.adjacent[x].v)

Example 8: Find the shortest path between vertex A and the other vertices in the graph in Figure 17 using Algorithm 12.8.

Each of the vertices in Figure 17 has been assigned an integer index, which is stored in the *ID* property of the vertex. For example, vertex *A* has an *ID* of 0; and vertex B has an ID of 1.

Steps:
1. Create an array *previous* of length *n*, where *n* is the number of vertices in the graph.

int previous[n];
2. Initialize all values in the *previous* array to -1 to produce the previous array shown in Figure 18.

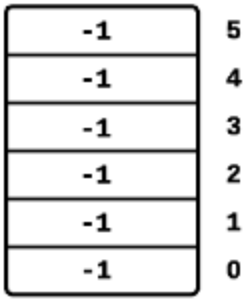


Figure 18. The shortest path algorithm uses an array called previous. Initially, all values in previous are -1. The size of previous is the same size as the number of vertices in the graph.

3. **Line 7:** vertex A is dequeued (enqueued on Line 5) and its adjacent vertices are evaluated and enqueued on Lines 8-13.

4. **Line 11:** the *previous[ID]* value for each of vertex *A's* adjacent vertices is set to the ID of *A*, which is 0. The *ID* for vertex *B* is 1 and the *ID* for vertex *F* is 4, which sets the values for *previous[1]* and *previous[4]* to 0. The state of the previous array is shown in Figure 19.

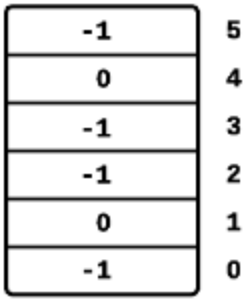


Figure 19. The previous values for vertex A's adjacent vertices are updated to include the index of A. The vertex A has an ID of 0, therefore, a 0 is written to positions 1 and 4, the IDs of A's adjacent vertices B and F.

5. **Line 7:** dequeue the vertex *B* and evaluate and enqueue it's unvisited adjacent vertices on Lines 8-13. The only unvisited vertex is *G*, which has an index of 2. Set *previous[2] = 1* to show that *B* is *G's* parent. The *ID* of

vertex *B* is 1. The updated *previous* array is shown in Figure 20.

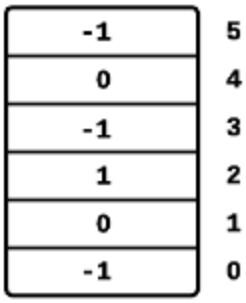


Figure 20. State of the previous array after vertex G's parent is added to the array. G has an index of 2 and G's parent has an index of 1, therefore, a 1 is stored in previous[2].

6. **Line 7:** dequeue the vertex *F,* which has an *ID* of 4. The unvisited adjacent vertices to *F* are the *E* and *H*.
7. **Line 11:** set the *previous* array for the *E* and *H* vertices. The *ID* of *E* is 3 and the *ID* of *H* is 5. The values for *previous[3]* and *previous[5]* are set to 4. The final *previous* array is shown in Figure 21.

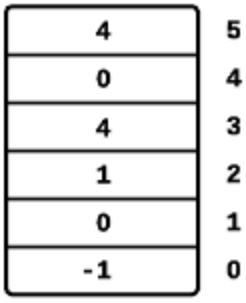


Figure 21. The final previous array after all vertices evaluated. The array can be used to re-create the path from any vertex back to vertex A.

Once the *previous* array has been generated, it can be used to re-create the path back to the starting vertex from any vertex in the graph. To re-create the path from the vertex *G* back to the vertex *A*:

Steps:

1. Start at the *ID* of *G* in the *previous* array, which is 2, and examine *previous[2]* to get the parent *ID* of *G*, which is 1. The vertex with an *ID* of 1 is *B,* which means that the path to *G* goes through *B*.
2. Go to *previous[1]* to get the parent *ID* of *B*, which is 0. The vertex with an *ID* of 0 is *A*.
3. Go to *previous[0]* to get the parent *ID* of *A*, which is -1. The -1 signifies that *A* is the starting vertex and doesn't have a parent.

The shortest path from *A* to *G* goes through the vertices *A*-*B*-*G*.

12.8.5 Depth-first search

Another ordering for searching the vertices in a graph, called depth-first search (DFS), evaluates the vertices along one path before evaluating other paths. DFS is used in the tree-traversal algorithms for binary trees (shown in Algorithm 10.1, 10.2, and 10.3) that print the nodes in the tree. Those algorithms recursively traversed all the way to the leaf nodes in the tree, following the left or right branch, before evaluating any of the other branches in the tree. For the graph in Figure 22, a DFS that starts at vertex A and selects the next adjacent vertex alphabetically, would evaluate vertices in the order *A*-*B*-*C*-*D* backing up and selecting a different path to evaluate vertices *E* and *F*. This ordering differs from a breadth-first search, which would evaluate vertices *B* and *E* before evaluating vertices *C* and *D*. Also, unlike breadth-first search, which would find the shortest path between two vertices in an unweighted graph if a path exists, depth-first search would find a path, but not necessarily the shortest path.

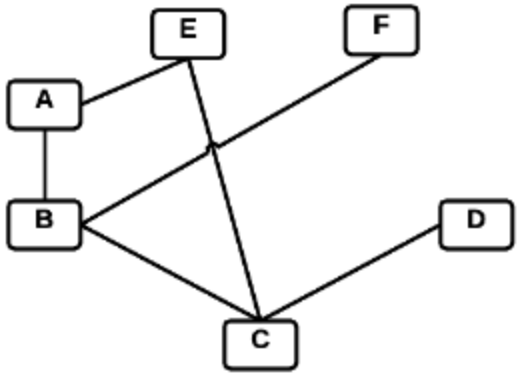


Figure 22. To find a path from A to D in this unweighted graph using depth-first search, B, C, and D would be evaluated before E.

The evaluation order of vertices in a graph is shown in the tree in Figure 23. The child nodes in the tree are the adjacent vertices in a graph. For example, a search that starts at vertex *A* in Figure 22 would place *A* as the root of a tree. Vertex *B* is the left-child of *A* in a tree and vertex *C* is the left-child of *B* in the tree. These are the first three nodes evaluated in a DFS.

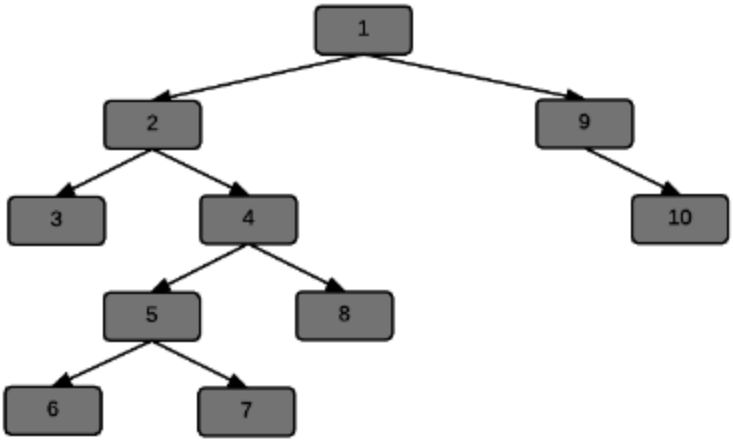


Figure 23. Evaluation order in a depth-first search. Nodes at deeper levels in the tree are evaluated before nodes at the same level in the tree.

Once the bottom of the left branch in a tree is reached, which is equivalent to following a path in a graph until

there are no unvisited, adjacent vertices on that path to evaluate, DFS will evaluate all nodes in the right branch. In a graph, following a different branch means selecting a different vertex at the last decision point. In the graph in Figure 22, after the path *A-B-C-D* is evaluated, the algorithm would return to vertex *C* and select vertex *E* to evaluate the path *A-B-C-E*.

DFS can be implemented using a recursive or a non-recursive algorithm. Examples of both algorithms are shown in Algorithm 12.9 and Algorithm 12.10, respectively. Both algorithms take the key value of the starting vertex as an argument and employee a *search()* algorithm (Algorithm 12.4) to find the starting vertex in the graph and traverse the graph in a depth-first ordering from that vertex. Non-recursive implementations of DFS typically use a stack data structure to store the vertices as they are visited. Using an external stack generates similar behavior to the recursive algorithm, which uses the program stack. The stack generates an ordering where the most-recently visited vertices are popped off the stack and processed before vertices that were encountered at higher levels in the tree are processed.

Algorithm 12.9. depthFirstSearch(value) Print the *key* values in the graph using a depth-first traversal of the vertices.

Pre-conditions *value* is a valid vertex *key* value for the root of the search tree.
search() algorithm exists to identify the vertex in the graph where *vertex.key = value*
Post-conditions *key* values displayed in a depth-first order from the starting vertex.

Algorithm DFS(vertex) 1. vertex.visited = true 2. for x = 0 to vertex.adjacent.end 3. if(!vertex.adjacent[x].v.visited) 4. print(vertex.adjacent[x].v.key) 5. DFS(vertex.adjacent[x].v) depthFirstSearch(value) 1. vertex = search(value) 2. print(vertex.key) 3. DFS(vertex)

Algorithm 12.10. depthFirstSearchNonRecursive(value) Print the key values in a graph using a non-recursive depth-first search.

Pre-conditions *value* is a valid vertex *key* value for the root of the search tree.
search() algorithm exists to identify the vertex in the graph where *vertex.key = value.*

Post-conditions *key* values in the graph displayed in a depth-first order from the starting vertex.

Algorithm depthFirstSearchNonRecursive(value) 1. vertex = search(value) 2. vertex.visited = true 3. vertex.distance = 0
4. stack.push(vertex) 5. while(!stack.isEmpty()) 6. ve = stack.pop() 7. print(ve.key) 8. for x = 0 to ve.adjacent.end 9. if(!ve.adjacent[x].v.visited) 10. ve.adjacent[x].v.visited = true 11. stack.push(ve.adjacent[x].v)

12.9 Dijkstra’s algorithm

The breadth-first search algorithms found the shortest distance in an unweighted graph, where the shortest distance is the path that traverses the fewest number of edges.

In a weighted graph, the path with the shortest distance between two vertices is the path with the lowest cumulative edge weight, which makes the calculation a bit more complicated than in an unweighted graph. The path with the lowest weight won’t necessarily be the path that traverses the fewest number of edges.

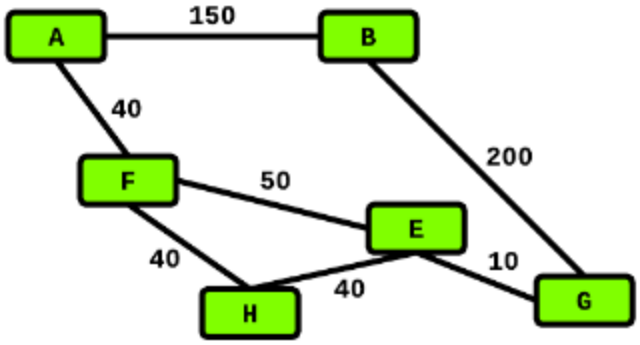


Figure 24. In this weighted graph, there are multiple paths between vertices A and G, each with a different cost. The path with the lowest cost is the one with the lowest total weight for the edges traversed.

In the weighted graph in Figure 24, there are multiple paths between vertices *A* and *G*. One path traverses *A*-*B*-*G* with a weight of 350. Another path traverses *A*-*F*-*E*-*G* with a weight of 100. There is also a path that traverses *A*-*F*-*H*-*E*-*G* with a weight of 130. The shortest path in this weighted graph is the one with a weight of 100: *A*-*F*-*E*-*G*, even though it traverses more edges than the *A*-*B*-*G* path.

The shortest distance in weighted graph, where the edge weights are strictly positive, can be found using Dijkstra's algorithm, which is a greedy algorithm that chooses the

lowest cumulative weight to any adjacent vertex that hasn't yet been solved. The shortest path overall is found from these individual lowest weight decisions.

Example 9: Find the shortest path in the graph in Figure 25 between the Start and End vertices.

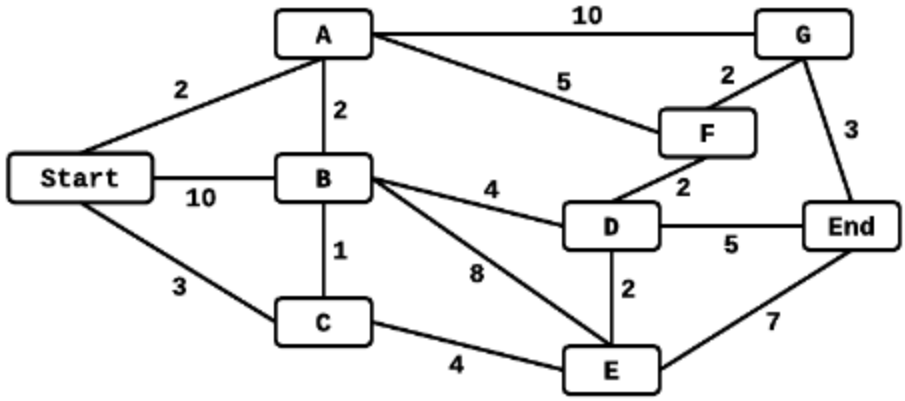


Figure 25. Find the shortest path between the Start and End vertices in this weighted graph.

The *vertex* **struct** is modified to included a *solved* parameter that serves a similar purpose to the visited property in the *breadthFirstSearch()* and *depthFirstSearch()* algorithms. The **struct** is also modified to store a pointer to the parent vertex.

struct vertex{
1. std::string key; 2. std::vector<adjVertex> adjacent; 3. bool solved; 4. int distance; 5. vertex *parent; };
Steps: 1. Mark the *Start* vertex as solved, and set the distance to *Start* as 0.
2. Find the unsolved (unvisited) vertices adjacent to *Start* and calculate the distance to those vertices as the distance to *Start* + the edge weight connecting the vertex to *Start*. The unsolved vertices adjacent to *Start* are *A*, *B*, and *C* (shown in blue in Figure 26). The distance to *A* is 2, to *B* is 10, and to *C* is 3.

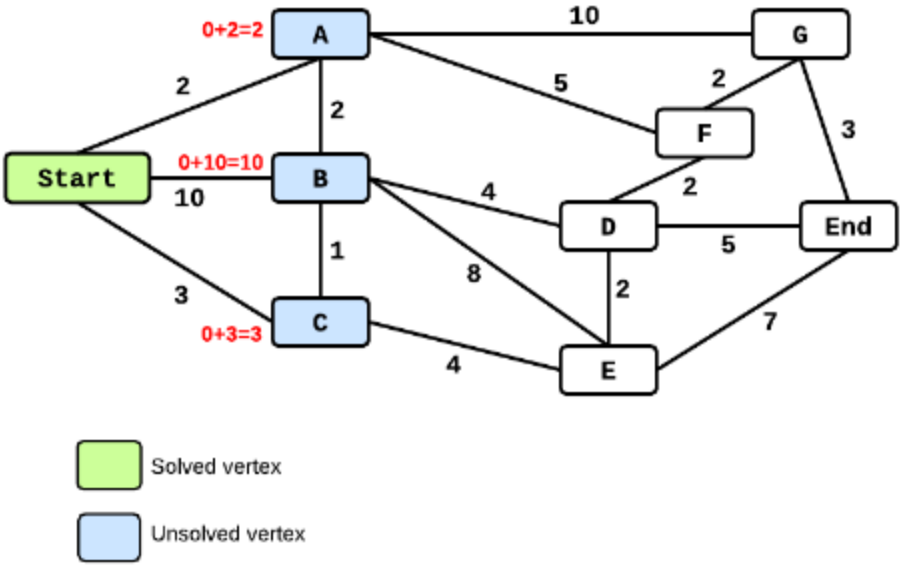


Figure 26. From the Start vertex, identify the unsolved vertices adjacent to Start and calculate the distance to each vertex. Those vertices are A, B, and C.

3. Select the vertex with the shortest distance and mark it as solved. Update the vertex to show its distance to *Start*. In this example, the shortest distance is to vertex *A*. The *A* is now solved, with a distance of 2 (shown in green in Figure 27), and a parent of *Start*. The vertex won't be solved again through another path.

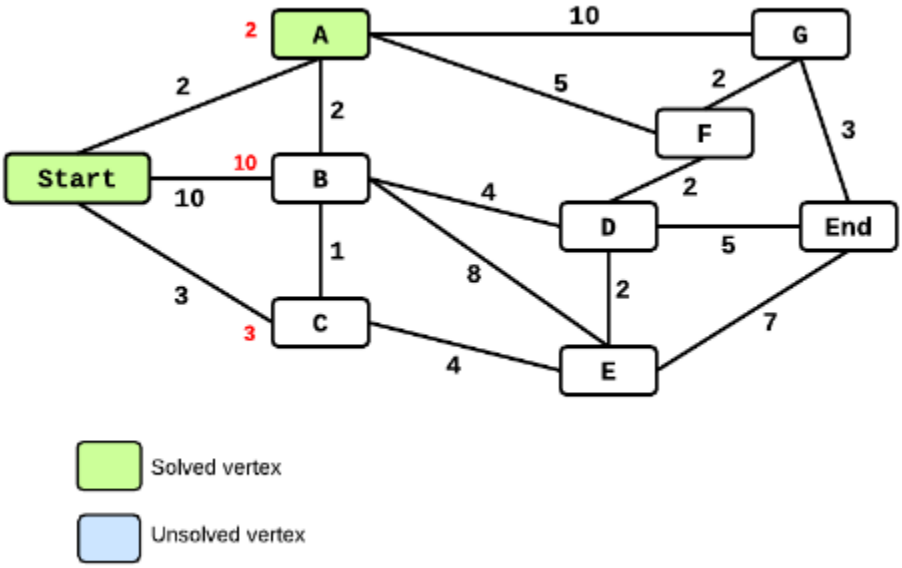


Figure 27. The vertex closest to Start is A with a distance of 2. The vertex A is now marked solved with its parent as Start and won't be solved again through another path.

4. Repeat the process of selecting all unsolved vertices adjacent to all solved vertices, and calculating the distance to those unsolved vertices. The solved vertices are *Start* and *A*, and the unsolved vertices adjacent to *Start* and *A* are *B* with a shortest distance of 4 going through *A*, *C* with a distance of 3, *F* with a distance of 7, and *G* with a distance of 12 (Figure 28).

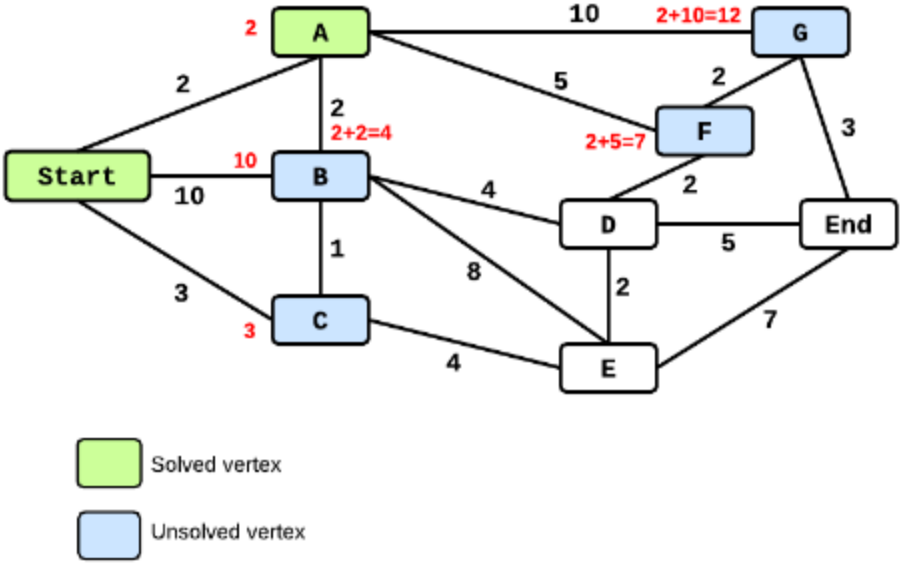


Figure 28. The vertices Start and A are both solved. The unsolved adjacent vertices to Start and A are B, C, F, and G.

5. The distance of 3 to vertex *C* is the shortest. Mark *C* as *solved* and its parent is *Start*. The solved vertices are shown in Figure 29.

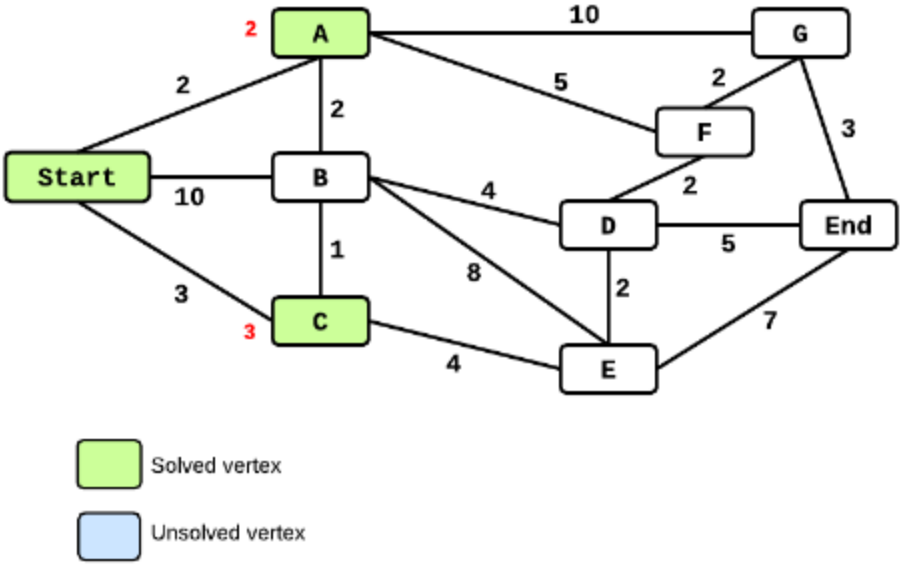


Figure 29. The vertices Start, A, and C are now solved.

6. Select all unsolved vertices adjacent to all solved vertices and calculate their distances. The unsolved vertices adjacent to *Start*, *A*, and *C* are *B* with a shortest distance of 4, *F* with a distance of 7, *E* with a distance of 7, and *G* with a distance of 12 (Figure 30).

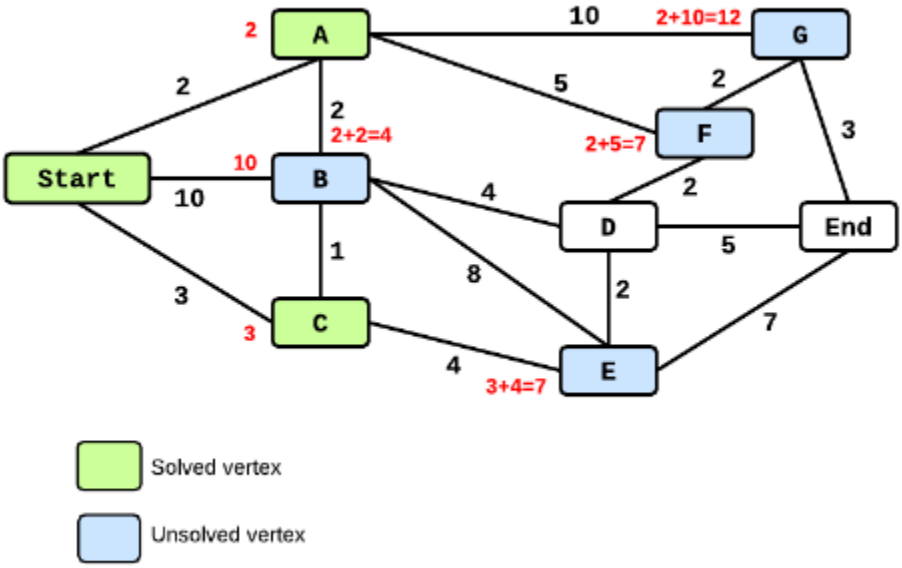


Figure 30. The solved vertices are Start, A, and C. The vertices adjacent to the solved vertices are B, E, F, and G.

7. The shortest path is to *B* through *A* with a distance of 4. Mark vertex *B* as solved with a distance of 4 (Figure 31). The parent of *B* is *A*.

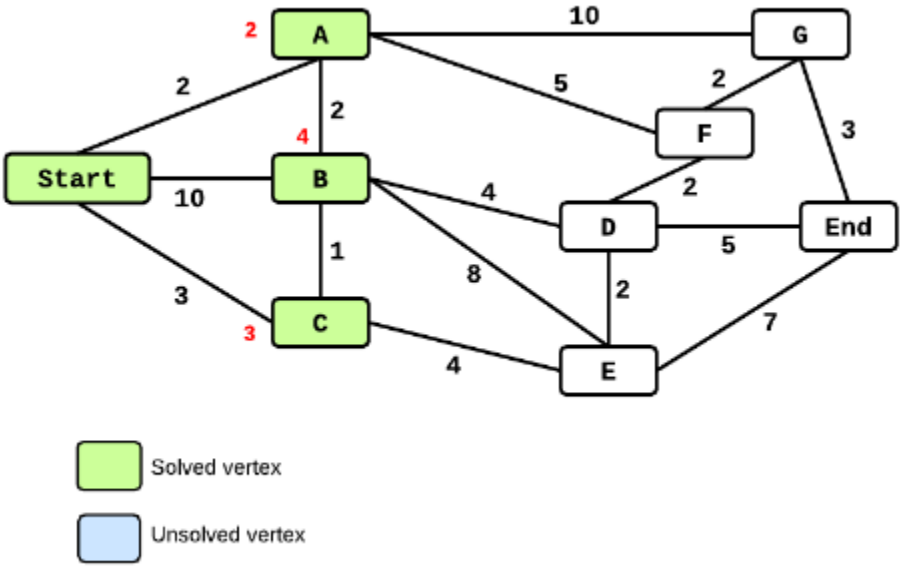


Figure 31. The vertices Start, A, B, and C are now solved. There are no unsolved vertices adjacent to the Start vertex.

8. Select all unsolved vertices adjacent to all solved vertices and calculate their distances to the *Start* vertex. All vertices adjacent to *Start* have been marked as solved. The unsolved adjacent vertices to *A*, *B*, and *C* are *D* with a distance of 8, *E* with a shortest distance of 7, *F* with a distance of 7, and *G* with a distance of 12 (Figure 32).

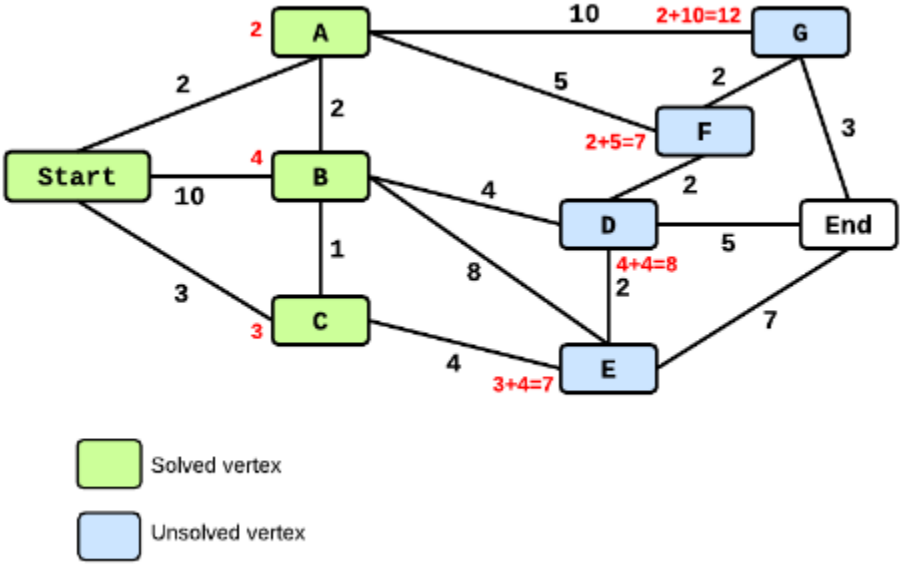


Figure 32. The solved vertices are Start, A, B, and C. The unsolved vertices adjacent to the solved vertices are D, E, F, and G.

9. There is a tie between *E* and *F*. Choose *E* using an alphabetical tie-breaker, and mark *E* as solved with a distance of 7 to the *Start* vertex (Figure 33). The parent of *E* is *C*.

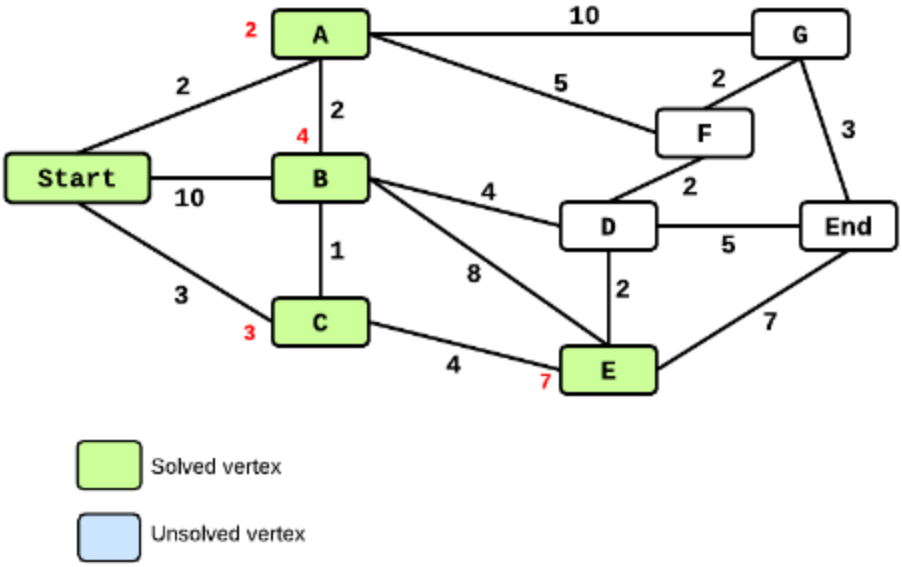


Figure 33. The vertex E is marked solved with a distance of 7. The solved vertices are Start, A, B, C, and E.

10. Calculate the distance to all unsolved vertices adjacent to solved vertices. The shortest distance is to *F* with a distance of 7 (Figure 34).

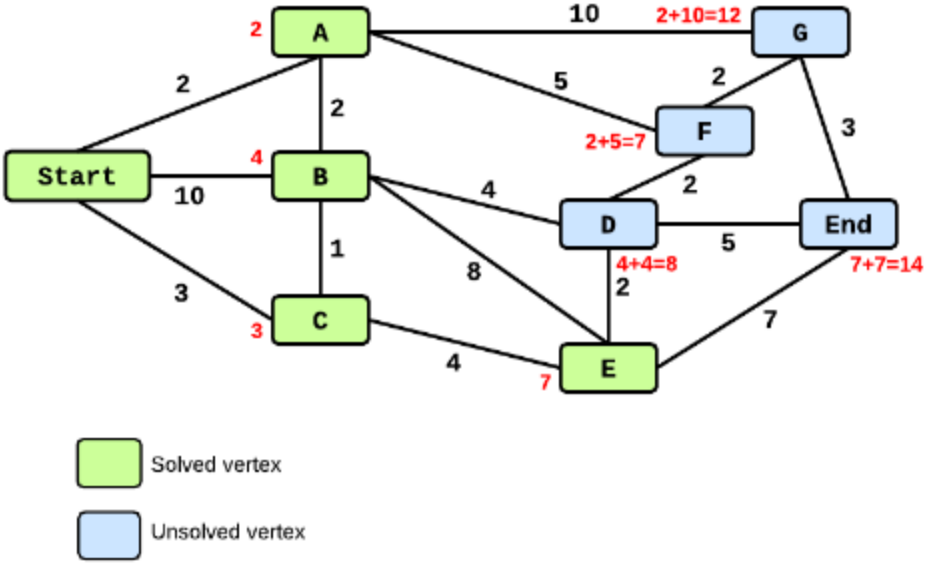


Figure 34. The solved vertices are Start, A, B, C, and E. The unsolved vertices are D, F, G, and End. The shortest distance is to F with a cost of 7.

11. Mark *F* as solved with a distance of 7 back to *Start*. The parent of *F* is vertex *A*.

12. Calculate the distance to unsolved vertices from all solved vertices. The solved vertices are *Start, A, B, C, E,* and *F*. The unsolved vertices are *D* with a shortest distance of 8, *G* with a shortest distance of 9, and *End* with a distance of 14 (Figure 35).

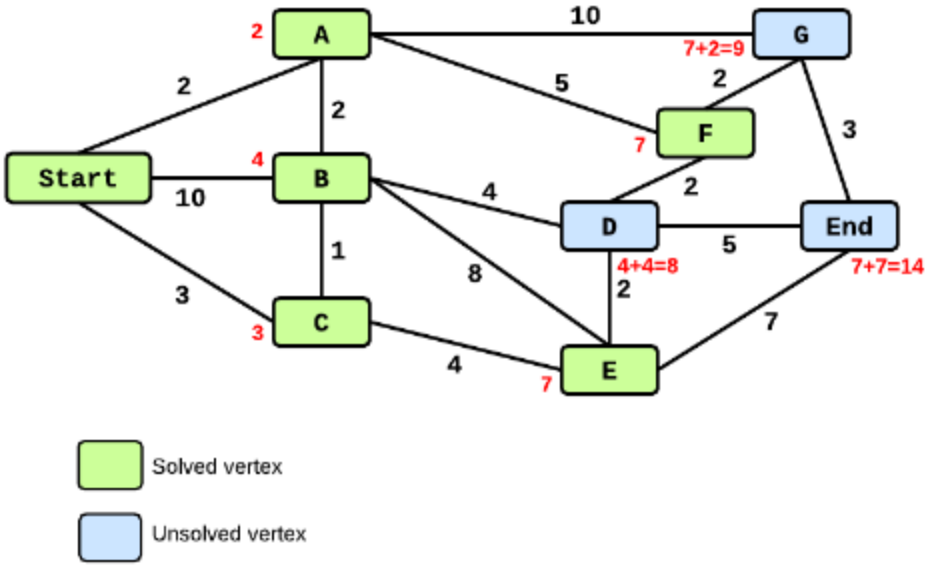


Figure 35. The vertex F is now solved. The unsolved vertices are D, G, and End with costs of 8, 9, and 14 respectively.

13. The shortest distance is to *D* with a distance of 8. Mark vertex *D* as solved. The parent of *D* is vertex *B*. The remaining unsolved vertices are *G* with a shortest distance of 9 and *End* with a shortest distance of 13 (Figure 36).

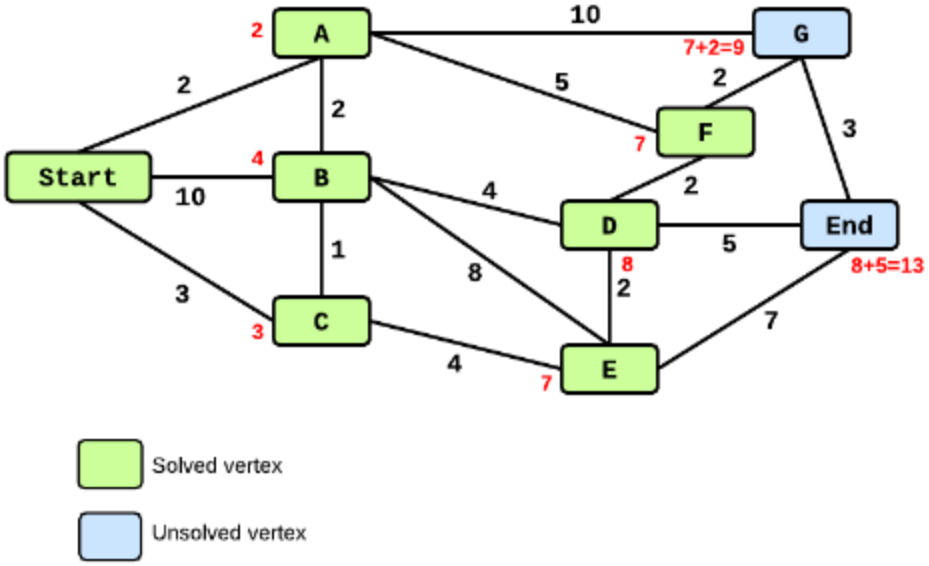


Figure 36. The only unsolved vertices are G with a shortest distance of 9 and End with a shortest distance of 13.

14. Mark *G* as solved with a distance of 9. The parent of *G* is vertex *F*. The only remaining unsolved vertex is *End,* which now has a shortest distance of 12 following a path that goes through *G* (Figure 37).

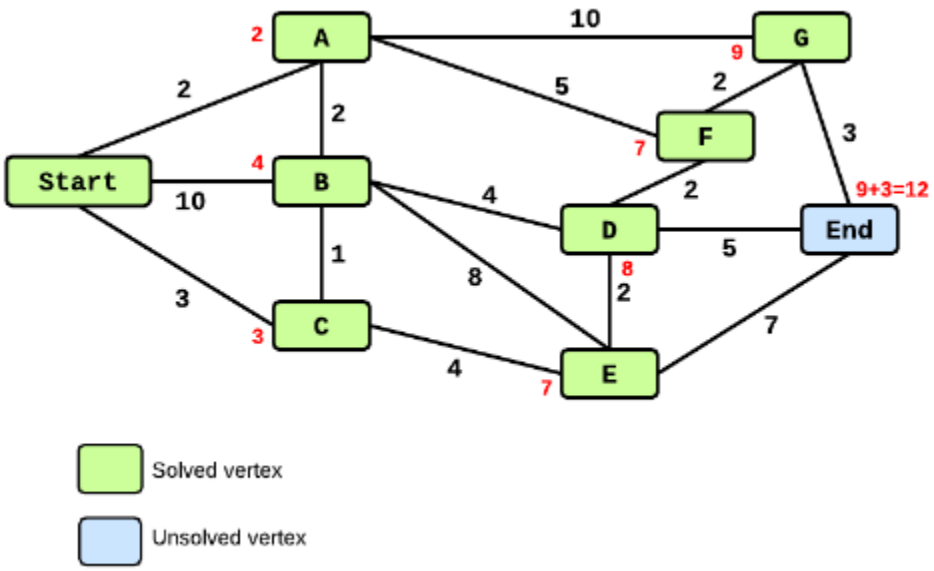


Figure 37. The vertex G is marked as solved with a distance of 9. The only unsolved vertex is End with a distance of 12.

15. Mark *End* as solved (Figure 38). The parent of *End* is the vertex *G*. The shortest distance from *Start* to *End* is 12 following the path: *Start-A-F-G-End*.

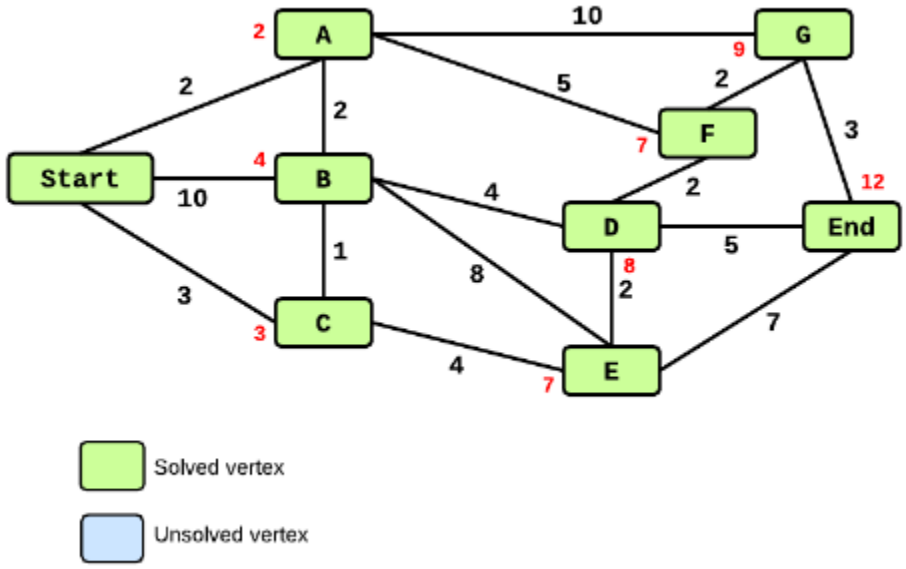


Figure 38. The shortest distance from Start to End is 12 following the path Start-A-F-G-End.

12.9.1 Implementing Dijkstra's algorithm

Dijkstra's algorithm uses a breadth-first search to identify the unsolved vertices at each step. The algorithm builds a list of solved vertices, where each vertex in the list includes the distance back to the root vertex and a pointer to its parent vertex. The algorithm is shown in Algorithm 12.11.

Algorithm 12.11. Dijkstra(startValue, endValue) Find the shortest path between the *startValue* and *endValue* vertices in a graph.

Pre-conditions *startValue* and *endValue* are valid key values *search()* algorithm finds the *startValue* and *endValue* vertex in the graph INT_MAX contains maximum system integer value
Post-conditions Returns the vertex with a *key* that matches the *endValue*. Stored in the vertex distance property is the shortest distance back to the *startValue* vertex. All vertices on the shortest path have a *parent* property, which points to the previous vertex in the path.

Algorithm Dijkstra(start, end) 1. startV = search(start) 2. endV = search(end) 3. startV.solved = true 4. startV.distance = 0
5. solved = {startV} //list of solved vertices 6. while (!endV.solved) 7. minDistance = INT_MAX
8. solvedV = NULL
9. for x = 0 to solved.end 10. s = solved[x]
11. for y = 0 to s.adjacent.end 12. if(!s.adjacent[y].v.solved) 13. dist = s.distance + s.adjacent[y].weight 14. if(dist < minDistance) 15. solvedV = s.adjacent[y].v 16. minDistance = dist 17. parent = s 18. solvedV.distance = minDistance 19. solvedV.parent = parent 20. solvedV.solved = true 21. solved.add(solvedV) 22. return endV

13 Hash tables

A hash table, also known as a hash map, is a data structure that stores data using a parameter in the data, called a *key*, to map the data to an index in an array. The data is also called a record, and the array where records are stored is called a hash table.

There are two necessary components to a hash table: the array where the records are stored and a hash function that generates the mapping to an array index.

For example, imagine the hash table is used to store records of movies. Each movie record contains the *Title, Ranking*, and *Year* of the movie. The movie could be defined with a **struct** as follows:

```
struct movie{
   string Title;
   int Ranking;
   int Year;
}
```

Figure 1 shows the process of how individual movie records are stored in a hash table. Each movie is a record that contains the properties of the movie. The key for the record, which in this case is the title, is input to a hash function, shown in Figure 1 as *h(Title)*. The hash function uses the ASCII characters in the title, generates a unique integer value for that title. That integer value is the index in the hash table array where the movie record is then stored. For example, if the hash function returned a value of 2, then the movie would be stored at index = 2 in the hash table array.

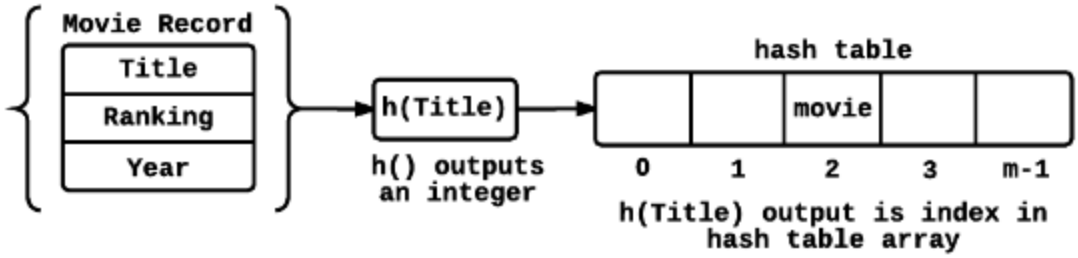


Figure 1. Process showing how individual movie records are stored in a hash table. The movie Title is the input to a hash function, which outputs an integer that serves as the index for the movie in an array.

13.1 Hash functions

Hash functions convert the key into an integer index to store the record in the hash table. One of the simplest hash functions converts a string to an integer by summing the ASCII values of all letters in the string and then modding the sum by the array size. The mod operation ensures that the integer is within the bounds of the array. The algorithm for this hash function is shown in Algorithm 13.1. The algorithm takes the key and the size of the hash table as arguments and returns the index where the record is to be stored in the hash table.

Algorithm 13.1. hashSum(key, tableSize) Calculates the index in a hash table for a record with a specified *key* value.

Pre-conditions *key* is a string or character array. *tableSize* is the size of the array.

Post-conditions Returns integer index, where *0<= index < tableSize*.

Algorithm hashSum(key, tableSize) 1. sum = 0
2. for x = 0 to key.end 3. sum =sum + key[x]
4. sum = sum % tableSize 5. return sum

Example 1: Calculate the hash value of *Shawshank Redemption* if the size of the hash table is 50 using Algorithm 13.1.

• **Lines 2-3:** The ASCII values for the letters in *Shawshank Redemption* are summed and stored in the variable *sum*. The ASCII values for each letter are shown in Figure 2. The sum of the ASCII values is 2015.

Character	ASCII value	Character	ASCII value	Character	ASCII value
S	83	n	110	m	109
h	104	k	107	p	112
a	97	< space >	32	t	116
w	119	R	82	i	105
s	115	e	101	o	111
h	104	d	100	n	110
a	97	e	101		

Figure 2. ASCII values for the characters in “Shawshank Redemption”.

• **Line 4:** The sum is modded by the *tableSize*, which is 50, to scale the sum to a value between 0 and *tableSize*. The result: 2015 % 50 = 15. *Shawshank Redemption* is stored at index 15 in the hash table.

13.2 Using a hash function in C++

An example of how the *hashSum()* algorithm is used in a hash table is shown below in C++. An array of a specified size is created to store the hash table elements. In this example, they are movies that are defined by the *movie* **struct**.

• Create an array to store 50 movie objects: movie hashTable[50]; • Create an instance of movie: movie m; • Set the *Title* property of the movie to “Shawshank Redemption”: m.Title = "Shawshank Redemption"; • Use the movie title as the argument to *hashSum(),* which returns the index in the hash table where the title will be stored: int index = hashSum(m.Title, 50); • Use the index to store the movie: hashTable[index] = m; • To retrieve a record from a hash table, perform the steps in reverse. Calculate the hash value for the key, and then retrieve the information at that index in the hash table.

int index = hashSum("Shawshank Redemption", 50); movie m = hashTable[index];

13.3 Collisions

If every key always mapped to a unique index in a hash table, then hash tables would be used for everything. There would be no reason to store data in any other data structure because hash tables would be the faster than any other data structure available. However, in any large data set, the reality is that multiple records often have the same hash value, and therefore, need to be stored in the same index in a hash table. When this happens, it's called a collision, which is when two or more keys hash to the same index.

Formally, given a hash function *h*, a collision occurs when: $h(k_1) = h(k_2)$, $k_1 != k_2$, where k_1 and k_2 are keys.

Consider the strings: Go Cat Go.

and

Go Dog, Go The sum of the ASCII characters in *Go Cat Go.* is 71 + 111 + 32 + 67 + 97 + 116 + 32 + 71 + 111 + 46 = 754.

The sum of the ASCII characters in *Go Dog, Go* is 71 + 111 + 32 + 68 + 111 + 103 + 44 + 32 + 71 + 111 = 754.

Using the *hashSum()* algorithm as the hash function, these two strings will hash to the same index in the hash table and produce a collision. To store multiple elements at the same location in a hash table, an additional data structure, such as a linked list, needs to be added to the hash table.

13.4 Hash functions

In the real world, collisions happen, a lot. The challenge of designing a good hash table is in designing a hash function that limits the frequency of collision. The hash-function design needs to consider both the size of the hash table as well as the hash function that maps keys to indices. A table size that is too small for the number of records that need to be stored will result in collisions and inefficiency. A hash table that is too big will result in wasted space in memory.

13.4.1 Perfect hash functions

A perfect hash function assigns all records to a location in the hash table without collisions or wasted space.

Example 2: Store 100 phone numbers, with unique values between 3034841000 and 3034841099 in a hash table of size 100.

The values for the phone numbers are:

3034841000
3034841001
3034841002
.
.
.
3034841099

A hash function that just mods the phone number by the hash table size of 100:

phoneNumber % 100

will return the last two digits in the number:

3034841**00**
.
.
.
3034841**50**
.
.
.
3034841**99**

Those last two digits can be used as the hash table index for each phone number and the numbers will all be stored in a unique location with no collisions.

13.4.2 Imperfect hash functions

With an imperfect hash function, multiple keys can be assigned to the same index and result in collisions.

Example 3: Store 100 phone numbers, with unique values between 3034841000 and 3034841099 in a hash table of size 10.

The values for the phone numbers are the same as in the previous example:
3034841000
3034841001
.
.
.
3034841099

A hash function that mods the phone number by 10:

phoneNumber % 10

will return the last digit in the phone number:

3034841100**0**
3034841100**1**
.
.
.
3034841105**0**
.
.
.
3034841109**9**

If the last digit is used as the index in the hash table, then only indices 0 - 9 in the hash table will be used, and the indices 10 - 49 will be empty. The 10 used indices are only 10% of the table; the other 90% will be wasted. To store all 100 values in 10% of the table requires that 10 phone numbers be assigned to each used index.

Considerable effort goes into improving the performance of hash tables by designing new hash functions or finding the appropriate hash function for the type of data that needs to be stored. Many of these approaches involve complicated bit shifting and masking operations that are beyond the scope of this book. The hash functions shown here are meant as an introduction to how a few simple hash functions perform, the challenges of designing a hash function, and how heuristics are used for choosing a good hash table size.

In designing a hash function, it’s important to recognize how the function will operate on a particular set of data. Consider the following cases that use the *hashSum()* algorithm previously presented.

Example 4: Calculate the range of hash values for strings of 10 uppercase letters for various table sizes.

The ASCII values in a string of 10 *A*'s will sum to 650 (the ASCII value of *A* is 65). The ASCII values in a string of 10 *Z*'s will sum to 900. All other strings of 10 capital letters will have a sum between 650 and 900, which means that there is a maximum range of 250. For a table size of 1000, only 25% of the table will ever be used. Increasing the table size to accommodate additional data won't reduce collisions because the hash function doesn't distribute records evenly throughout the table.

Example 5: Calculate the range of hash values for strings of 10 uppercase and lowercase letters for various table sizes.

The ASCII values in a string of 10 *A*'s will sum to 650 and a string of 10 lowercase *z*'s will sum to 1220. All other strings of 10 capital or lowercase letters will sum to between 650 and 1220, which means there is a maximum range of 570. If the table size is 1000, the *hashSum()* algorithm will return values between 0 and 570. Only 57% of the table will ever be used. Just as with Example 4 using 10 capital letters, increasing the table size won't reduce collisions because the hash function is not distributing records evenly throughout the table.

13.4.3 Multiplication method

Another category of hash functions uses multiplication as part of the function. A simple multiplication method includes the following steps:

1. Given a key *k* (*k* is a string), generate the sum of the ASCII values for the characters in *k*.
2. Multiply *k* by a constant *A*, where $0 < A < 1$.

2. Store the fractional part of *kA*.
3. Multiply fractional part of *kA* by a constant, m, and take the floor of the result.

The value of *m* is generally selected to be a power of 2, and *A = 13/32*. Empirical studies have shown these values for *m* and *A* to work reasonably well.

Example 6: Use the multiplication method to calculate the hash value of a string of 10 uppercase A's, let m = 1024 and A = 13/32.

The sum of the ASCII values of 10 *A*'s is 650.

kA = (650 * (13/32)) = 264.0625

The fractional part is .0625.

0.0625m = 0.0625 * 1024 = 64

Since there is no decimal component of 64, that is the hash value.

Example 7: Use the multiplication method to calculate the hash value of a string of 10 lowercase z's, let m = 1024 and A = 13/32.

The sum of the ASCII values of 10 *z*'s is 1220.

kA = (1220 * (13/32) = 495.625

The fractional part is .625.

0.625m = 0.625 * 1024 = 640

Since there is no decimal component of 640, that is the hash value.

13.5 Collision resolution

There's no such thing as a perfect hash function for real data, which means that collisions happen. There are two common methods for handling collisions - open addressing and chaining. In open addressing algorithms, once a collision has been identified, a separate function is used to traverse the hash table array and place the record in the first available location. In chaining algorithms, the hash table is set up as an array of pointers that serve as the head of a linked list for a given hash value. Records are added to the linked list for the calculated hash value.

13.5.1 Collision resolution by open addressing

In algorithms that resolve collisions by open addressing, records are added to the hash table at the first identified open location. Figure 3 shows an example of how records would be stored with open addressing using an algorithm called linear probing. The hash table *T* has a size of 7. The entries *T[0 ... 3]* are unused. The entries *T[4 ... 6]* are used. The keys *Casablanca* and *The Godfather* both generate a hash value of 5 using the *hashSum()* function in Algorithm 13.1 with a table size of 7. In this example, Casablanca is added first and when *The Godfather* is added, the location is already occupied. The algorithm traverses the table linearly to find the next open location in the hash table, which is at an index value of 6. *The Godfather* record is added to the hash table at that location.

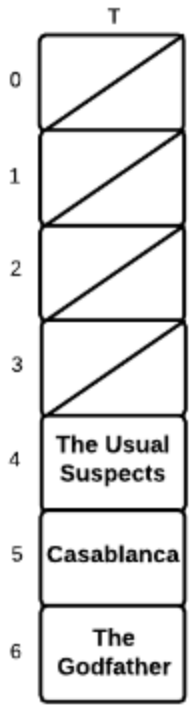


Figure 3. Hash table example that uses collision resolution by open addressing. The record "Casablanca" generates a hash value of 5 using the hashSum() hash function in Algorithm 13.1 for a table size of 7. The record is added to the next available index in the hash table if linear probing is used.

13.5.2 Collision resolution by chaining

An example of a hash table that stores movie records using chaining is shown in Figure 4. The hash table *T* has a size of 7. The entries *T[0 … 3]* are unused. The entries *T[4 … 6]* contain pointers to the head of a linked list, where each node in the list is a movie record. For example, both *Casablanca* and *The Godfather* have a hash value of 5 using the *hashSum()* algorithm in Algorithm 13.1 with a table size of 7. They are both included as nodes in a linked list at *T[5]* in alphabetical order.

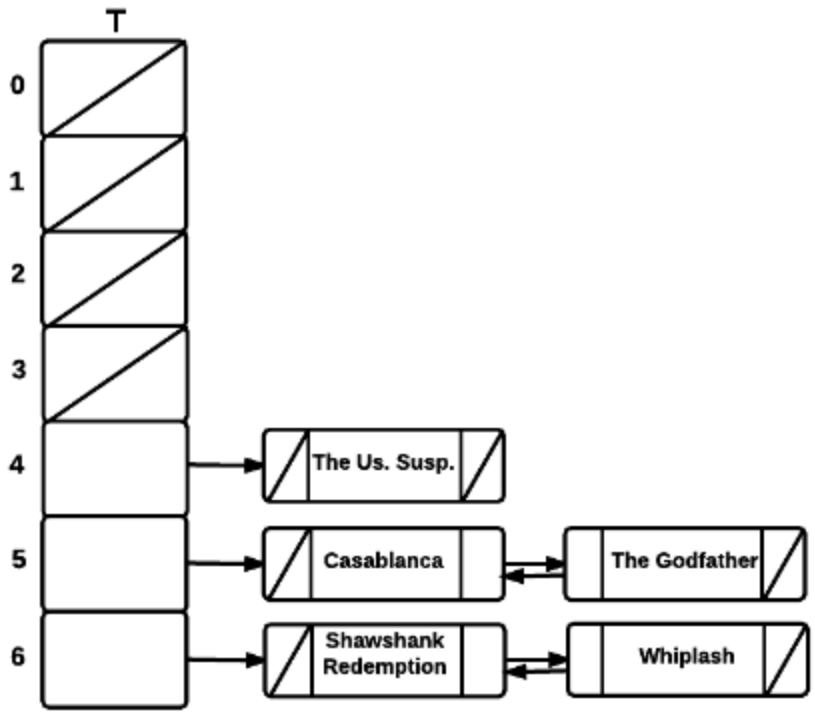


Figure 4. Example of a hash table that uses collision resolution by chaining. Each element in the table is a pointer to a linked list that stores the records that have the same hash value.

13.5.3 Creating a hash table with chaining

When the hash table is created, all indices in the table are initialized to NULL, indicating that there are no elements for that index. When an element is added to the hash table, the NULL entry is updated to point to the head of a linked list. In C++, vectors can also be used in place of a linked list.

Example 8: Add Shawshank Redemption to an empty hash table that uses chaining. The movie title is the key, the hash table size is 7, and the hash function is the *hashSum()* algorithm.

The *hashSum()* algorithm returns a value of 6. Since the table is empty, the movie *Shawshank Redemption* is the first entry in a doubly linked list at index 6 (Figure 4).

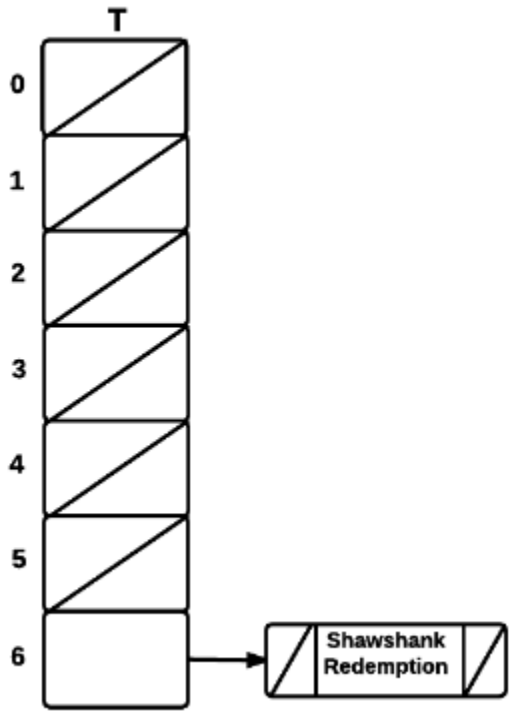


Figure 5. Hash table example after the movie Shawshank Redemption is added at index 6.

The *next* and *previous* pointers of the *Shawshank Redemption* movie node are set to NULL.

Example 9: Add the following movies to the hash table in Figure 5 using the *hashSum*() algorithm.

The Godfather The Usual Suspects Casablanca Whiplash
The return value of *hashSum("The Godfather")* is 1237 % 7 = 5.
The return value of *hashSum("The Usual Suspects")* is 1733 % 7 = 4.
The return value of *hashSum("Casablanca")* is 985 % 7 = 5.
The return value of *hashSum("Whiplash")* is 832 % 7 = 6.

When these four movies are added to the hash table, there is a collision at index 5 with *The Godfather* and *Casablanca,*

and a collision at index 6 with *Shawshank Redemption* and *Whiplash*.

The order that the movies are stored in the hash table is shown in Figure 6. The *next* and *previous* pointers in the doubly linked list connect the movie nodes and make it easier to add new movie nodes sorted alphabetically. As an example, the movie *Casablanca* at *T[5]* has a previous pointer of NULL, and a next pointer to the movie node *The Godfather*. *The Godfather's* previous pointer points to the *Casablanca* node, and its next pointer points to NULL.

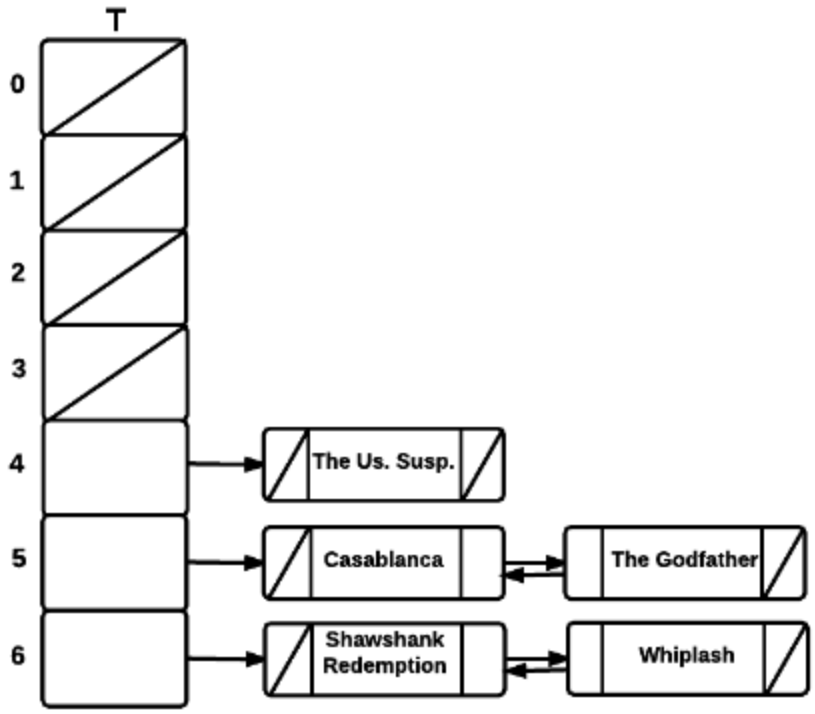


Figure 6. Hash table with chaining after Shawshank Redemption, The Usual Suspects, Casablanca, The Godfather, and Whiplash are added to the table.

13.6 The hash table ADT

The functionality for a hash table ADT that uses a linked list to implement collision resolution with chaining is shown in ADT 13.1. The size of the hash table and the hash table array are private variables in the ADT. The public methods to insert, search, and delete take the key value as an argument. The *insert()* method adds the record to the hash table chain in alphabetical order. The *search()* method returns a pointer to the record if it is found and NULL if it is not found. The *delete()* method searches the hash table for the value, resets the pointers in the chain to bypass the record, and then frees the memory for that node.

ADT 13.1. Hash Table HashTable: 1. private: 2. tableSize 3. hashTable 4. public: 5. Init() 6. insert(value) 7. search(value) 8. delete(value) 9. deleteTable()

13.6.1 Searching for a record

The *search()* algorithm, shown in Algorithm 13.2. search(value), first calculates the hash value for the search key to determine the index in the *hashTable* array where the record should be stored. On Line 2, the algorithm checks if the *hashTable* array is NULL at that index. If it is NULL, then the key value is not in the hash table and the algorithm returns NULL on Line 9. If *hashTable[index]* is not NULL, then the key could exist in the linked list chain for that index. Lines 4 - 8 traverse the chain checking for records where the key matches the search key.

Algorithm 13.2. search(value) Search for a node in the hash table with the specified key value and return a pointer to the node.

Pre-conditions Unused indices in the hash table are set to NULL.
value is a valid key search value for the hash table.

Post-conditions Returns a pointer to the node in the hash table chain where *node.key = value*.

Algorithm search(value) 1. index = hashSum(value, tableSize) 2. if (hashTable[index] != NULL) 3. tmp = hashTable[index]
4. while(tmp != NULL) 5. if (tmp->key == value) 6. return tmp 7. else 8. tmp = tmp.next 9. return NULL

13.6.2 Inserting a record

The algorithm to insert a record into a hash table, shown in Algorithm 13.3, calculates the hash value of the new record from its key, which is an argument to the algorithm. On Lines 2 -3, the *key* and *next* properties of the new record are set. Line 4 checks if there are already entries in the hash table for that hash value. If there are no entries, the new record is added as the first element at that index on Lines 5 - 6. If there are entries, Lines 9 - 13 check if the record is already in the hash table. Lines 15 - 16 traverse the chain for the correct position for the new record. On Lines 17 - 20, the pointers for the existing records are updated to include the new record.

Algorithm 13.3. insert(value) Insert a record into a hash table.

Pre-conditions Unused indices in the hash table are set to NULL.
value is a valid hash table key value.

Post-conditions Record inserted into the hash table at the correct location, as specified by the hash function.

Algorithm insert(value) 1. index = hashSum(value, tableSize) 2. hashElement.key = value 3. hashElement.next = NULL
4. if (hashTable[index] == NULL) 5. hashElement.previous = NULL
6. hashTable[index] = hashElement 7. else 8. tmp = hashTable[index]

9. while(tmp != NULL) 10. if(tmp.key == value) 11. print("duplicate") 12. return 13. tmp = tmp.next 14. tmp = hashTable[index]
15. while(tmp != NULL && hashElement.title > tmp.title) 16. tmp = tmp.next 17. hashElement.next = tmp 18. hashElement.previous = tmp.previous 19. tmp.previous.next = hashElement 20. tmp.previous = hashElement

13.6.3 Deleting a record

The steps to delete a record from a hash table, shown in Algorithm 13.4, follow the same pattern initially as the steps to insert and search for a record. The index for the element to delete is identified on Line 1 with a call to the hash function. Once the key value is found in the hash table chain, on Line 5, the *next* and *previous* pointers for the surrounding nodes in the chain are updated on Lines 6 - 9. One Line 10, the node is deleted, which frees the memory.

Algorithm 13.4. delete(value) Delete a record from a hash table.

Pre-conditions Unused indices in the hash table are NULL. *value* is a valid search key for a record in the hash table.

Post-conditions Node with the specified key value is deleted.
Pointers are updated to bypass the deleted node.

Algorithm delete (value) 1. index = hashSum(name,tableSize) 2. if (hashTable[index] != NULL) 3. tmp = hashTable[index]
4. while(tmp != NULL) 5. if(tmp.key == value) 6. if(tmp.previous != NULL) 7. tmp.previous.next = tmp.next 8. if(tmp.next != NULL) 9. tmp.next.previous = tmp.previous 10. delete tmp 11. break

13.7 Complexity of hash tables

The average performance of searching and inserting records in a hash table is O(1); it's constant and doesn't depend on the size of the hash table or the number of records to store.

The worst-case performance of hash-table operations occurs when all keys hash to the same location. The n records in the hash table would generate a linked-list of n elements, and the worst-case behavior would be the same as that of a linked list: O(n).

13.7.1 Selecting hash table size

There are a few things to consider when selecting the size of the hash table.
1. How many records need to be stored?
2. What is an acceptable number of evaluations in an unsuccessful search?

There is a heuristic (educated guess) for selecting the size of the hash table. Choose a prime number closest to number of records / acceptable evaluations. For example, if there are 1000 records, and 3 evaluations of a linked list is acceptable, then 1000/3 ≈ 333. The closest prime number to 333 is 331, which would be the table size.

13.7.2 Hash table load factor

The size of the hash table influences the load factor of the table, and in turn, the performance of the hash table. The load factor is calculated as n/N, where n is the number of keys stored and N is the size of the hash table. If the load factor stays below 1, and the hash function produces a minimal number of collisions, then performance of the hash table will be good. Searching and inserting will be O(1) operations. If, however, $n > N$ and the load factor is too

large, then the number of collisions will increase and more operations will be needed to find records stored in the table. Performance will no longer be constant, but rather, closer to O(n).

13.8 Hash Table Exercises

1. Using the *HashTable* definition in Code 13.1, where the elements in the hash table are movie records that contain the movie title and year it was released, implement the *insert()* and *hashSum()* methods to add records to the hash table.

Code 13.1 HashTable struct Movie{

```
std::string title; Movie *next; Movie(){}; Movie(std::string
in_title) {
title = in_title; next = NULL; previous = NULL; }
};
class HashTable{
public: HashTable(); ~HashTable(); void insert(std::string
title); private: int hashSum(std::string x, int tablesize); int
tableSize; Movie * hashTable[10]; };
```