C++ DATA STRUCTURES

Introduction 2

Chapter 00 : Built-in Data Types 10
Chapter 01 : Vector 24
Chapter 02 : Stack 45
Chapter 03 : Queue 75
Chapter 04 : Deque 97
Chapter 05 : Set 117
Chapter 06 : Linked List 144
Chapter 07 : List 169
Chapter 08 : Binary Tree 194
Chapter 09 : Binary Search Tree 220
Chapter 10 : Map 256
Chapter 11 : Sorting 277
Chapter 12 : Hash 315
Chapter 13 : Graph 337

A. ADT Properties 366
B. ADT Performance 367
C. Review: Pseudocode 369
D. Review: UML Class Diagram 371
E. Review: Operator Overloading 378
F. Review: Template 382
G. Review: Iterator 386
H. Review: Nested Class 389
I. Review: Namespace 393
J. Review: Separate Compilation 394
K. Review: Recursion 396
L. Debugging 401
M. Glossary 408
N. Index 413

What is a data-type and what is a data structure? In the simplest terms, a data-type is a way to represent data digitally. This enables us to store numbers, text, colors, locations, and just about anything else you can think of on a computer. Data structures are data-types except they represent collections of elements. Both data-types and data structures are tools to allow us to represent ideas or physical things in the world around us in a computer program.

The first generations of digital computers were not as data-centric as modern computers. In fact, they were little more than calculators. They were designed to accomplish only one specific task. These tasks were purely mathematical, such as predicting tides and computing artillery tables for the military. Claude Shannon changed all this in the 1930's when he explored methods for representing logical equations on the computer. This was followed in the 1950's with digital representations of letters and then text. As computers started filling more business and personal roles, the need for representing diverse elements digitally increased greatly. In 1986, less than 1% of the world's data was stored digitally. The 50% mark was crossed in 2002. Today, more than 99% of the world's data is stored digitally and the pace of digital content creation is increasing exponentially.

A key to this remarkable transformation to digital data is the extreme flexibility of computers. Not only can computers be programmed to accomplish a wide variety of tasks, but they can also be programmed to store a wide variety of data. Digital representation of data is a fundamental decision point of any programming problem and digital data is an important part of our everyday life. With a more complete understanding of data types and data structures, we can make more informed decisions and make our programs run more efficiently. This is the subject of this textbook.

Three Considerations

When working with a data-type or data structure, there are three important considerations: **correctness**, **size**, and **performance**.

Correctness

Correctness is a function of the data-type or data structure behaving the way the client expects. We call this the "contract." For example, the contract for a number data-type would consist of how it performs arithmetic operations. If the client cannot rely on 2+2=4, then what value is the number data-type?

All data-types and data-structures must have the correctness property both in the typical and common uses as well as in the atypical and marginal uses.

Size

Size is a function of the amount of space required to represent the data. For example, one might consider storing a pixel as three integers. Since an integer takes 64 bits or 8 bytes, that is a total 24 bytes for a single RGB (red, green, blue) pixel. By itself it does not seem like much, but when combined into a 20-megapixel image, it makes for a large file. If the same pixel was to be stored as three chars (8 bits apiece for 24 bits total), then the 20-megapixel image would take 1/8th the size!

Performance

The final consideration is performance. How long does it take to perform the operations associated with the data-type or data structure? For the most part, we care about how the performance scales as we use a large number of elements. Consider a list data-structure. If one were to need a loop to search through a list to find an element, then it would take on average 5.5 units of time (such as 5.5 milliseconds) to find the element in a list of 10 elements. If you double the size of the list, it will take twice as long (11 milliseconds). It will take on average half million units of time to find an element when there are a million elements. Since the cost is directly proportional to the size, we call this relationship linear. Thus, if the size of the list is n, then the cost of finding an element is O(n) which means roughly "n times some constant." The most common performance characteristics of a function are the following:

Notation	Name	Description
O(1)	Constant	Performance is unrelated to the size of the data structure
O(log n)	Logarithmic	Performance is a log of the size of the data structure
O(n)	Linear	Performance is directly related to size
O(n log n)	N-Log-N	Performance is linear times logarithmic
O(n^2)	N-Squared	Performance is the square of the size of the data structure
O(2^n)	2-to-the-N	Performance is really bad

To give you some idea of how these performance characteristics relate to each other, consider a data-set containing 1,000 elements. If it takes a blink of an eye to perform one operation (0.1 second if you are very quick), how long would a function take to complete?

Performance	Time	Performance Implication
O(1)	0.1 second	Faster than you will notice
O(log n)	1 second	A noticeable lag
O(n)	1½ minutes	Takes considerable patience
O(n log n)	14 minutes	The user will leave the computer to take a walk
O(n^2)	28 hours	The user lost all hope
O(2^n)	2 trillion times the age of the universe	

Consider what would happen if our one operation took us twice as long to perform (0.2 seconds as opposed to 0.1) over the same 1,000 element data-set. Executing this operation once would show a noticeable change; everything would take twice as long. However, if we used an algorithm that is O(n^2) as opposed to O(n log n), then the performance implications would be dramatic (56 hours for O(n^2) as opposed to 28 minutes for to O(n log n)). Thus, the exponent of the performance characteristic (meaning O(1) vs. O(n) vs. O(n^2)) is much more important than the duration of a single event (0.2 seconds per operation vs. 0.1 seconds).

This semester, we will consider algorithms that have all of the above performance characteristics except O(2^n). Now, the next obvious question is: how can we identify the performance characteristic of a given algorithm? This can be answered by looking at each of the various performance levels in turn.

O(1)

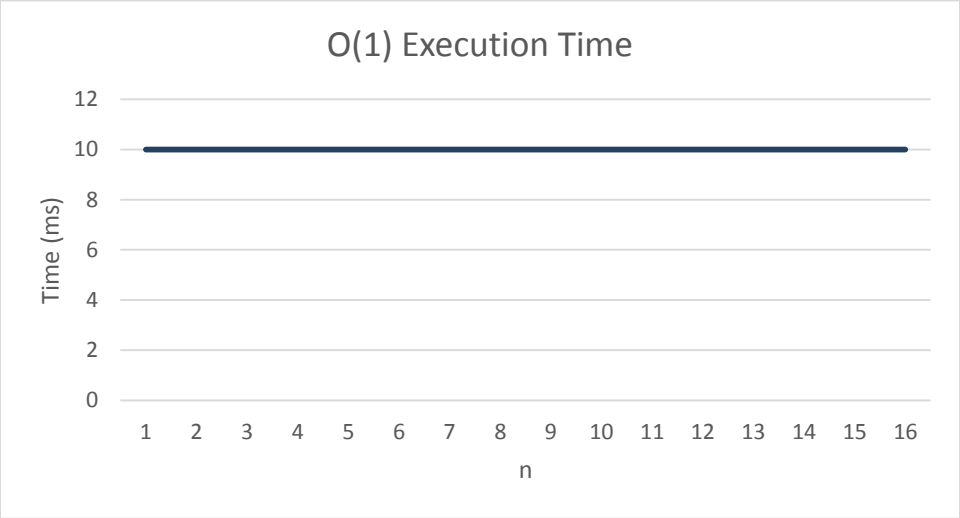
Constant performance yields comparable execution time regardless of the size of the data-set. This means the function cannot have a loop that is related to the size of the data-set. Consider the following function:

```
template <class T>
Stack <T> :: Stack(int size) throw (const char *)
{
   if (size > 0)                                          // line 1
      data = new T[size];                                 // line 2
   else                                                   // line 3
      throw "ERROR: Unable to allocate buffer";           // line 4
}
```

It takes the same amount of time to allocate a memory block of 4 bytes as it does 4 million bytes. Thus, regardless of the value of size, it takes the same amount of time to execute this function.

Line	Discussion	Cost
1	Only executed once, when the function is called	O(1)
2	Executed at most once	O(1)
3	Not technically a line of code. This is part of the IF statement	
4	Executed at most once, when the passed size is invalid	O(1)

If we were to graph this with the vertical axis being execution time and the horizontal axis being the size of the data-set, the graph would be flat:



This means that no matter the number of elements in the data-set (n), the execution time is always the same. Thus we see that constant time algorithms are very efficient. They are the gold standard that we strive to achieve in our designs but they are also not always possible.

O(log n)

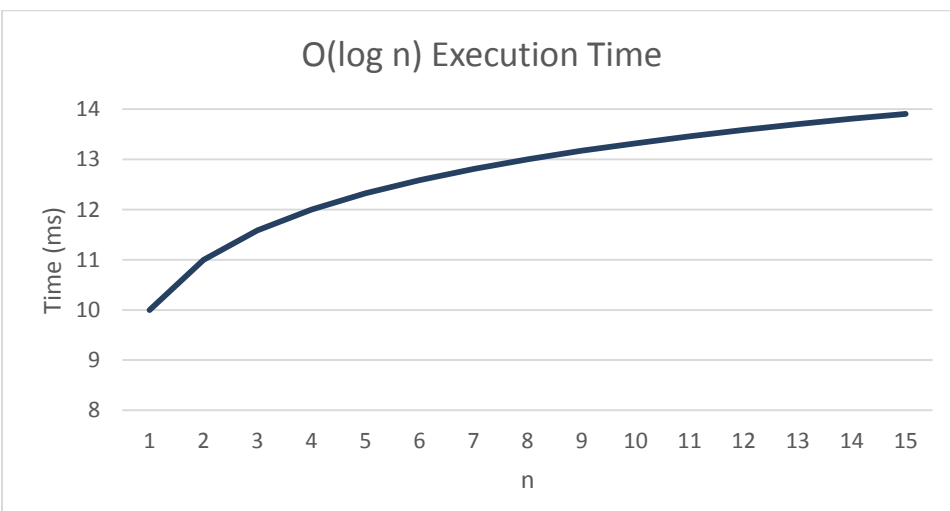
Logarithmic performance (and indeed everything but constant) does relate execution time to data-set size. However, the rate of increase slows down as the data set gets larger. An example is a binary search:

```
bool binarySearch(double data[], int n, double find)
{
   int iFirst = 0;
   int iLast = n - 1;                        // line 1

   while (iFirst <= iLast)                   // line 2
   {
      iMiddle = (iFirst + iLast) / 2;        // line 3
      if (data[iMiddle] == find)
         return true;                        // line 4
      if (data[iMiddle] > find)              // line 5
         iLast = iMiddle - 1;                // line 6
      else
         iFirst = iMiddle + 1;               // line 7
   }
   return false;                             // line 8
}
```

Notice in line 3 how the distance between `iFirst` and `iLast` gets cut in half every iteration. Thus it will take a single additional iteration of the main loop to search through a data-set that is twice the size. This is the distinguishing characteristic of logarithmic algorithms.

Line	Discussion	Cost
1	Only executed once, when the function is called	O(1)
2,3,5	Executed once for each loop. There are log n loops	O(log n)
4,8	Can only be executed once because we exit the function after it is called	O(1)
6,7	Executed about half the times the loop is run. This is still O(log n)	O(log n)

The graph of execution time to `n` increases with time but the slope decreases.



Many people find it difficult to internalize logarithmic growth. Consider this equation:

$$2^n = m \xrightarrow{yields} \log_2 m = n$$

Thus exponents and logarithms are inverses of each other. Since $2^{32} = 4\ billion$ then $\log_2 4\ billion\ = 32$. This means it will only take 32 iterations through our binary search to find an element in a list of 4 billion. Logarithmic algorithms are very efficient, even for large data-sets.

O(n)

In linear algorithms, twice as much data takes twice as long to execute. Consider the following algorithm to find the largest element in an un-sorted array. We will use a single loop to iterate through all members of the data-set:

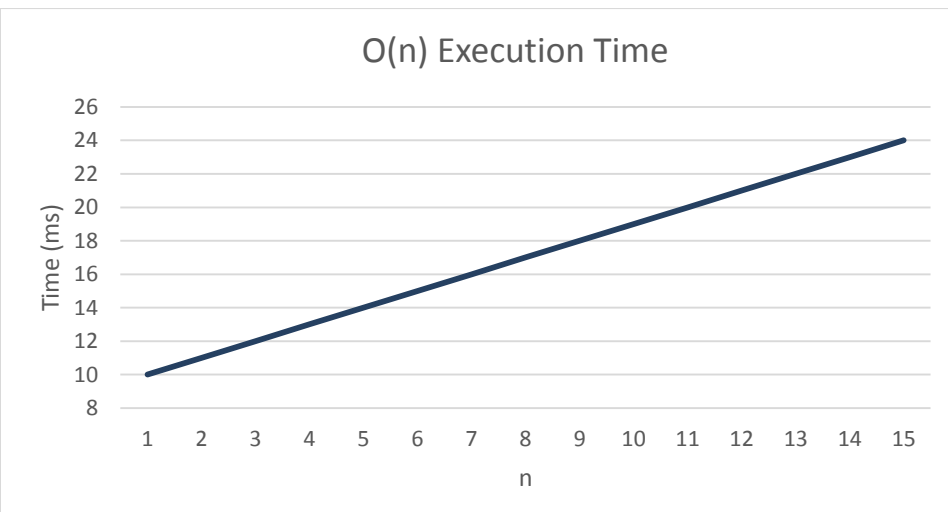
```
template <class T>
T & findLargest(T data[], int n)
{
   T * pT = data;                        // line 1
   for (int i = 1; i < n; i++)           // line 2
      if (*pT < data[i])                 // line 3
         pT = data + i;
   return *pT;                           // line 4
}
```

We can estimate the performance cost of this algorithm by counting how many times each line of code gets executed.

Line	Discussion	Cost
1	Only executed once, when the function is called	O(1)
2	Executed once per iteration of the loop	O(n)
3	Executed once per iteration of the loop	O(n)
4	Executed once at the end of the program	O(1)

Thus we can see that the overall cost of the function is O(n).

The graph of performance to data-set size is influenced by `n` and it increases steadily with time. Regardless of the value of `n`, the slope remains constant.



O(n log n)

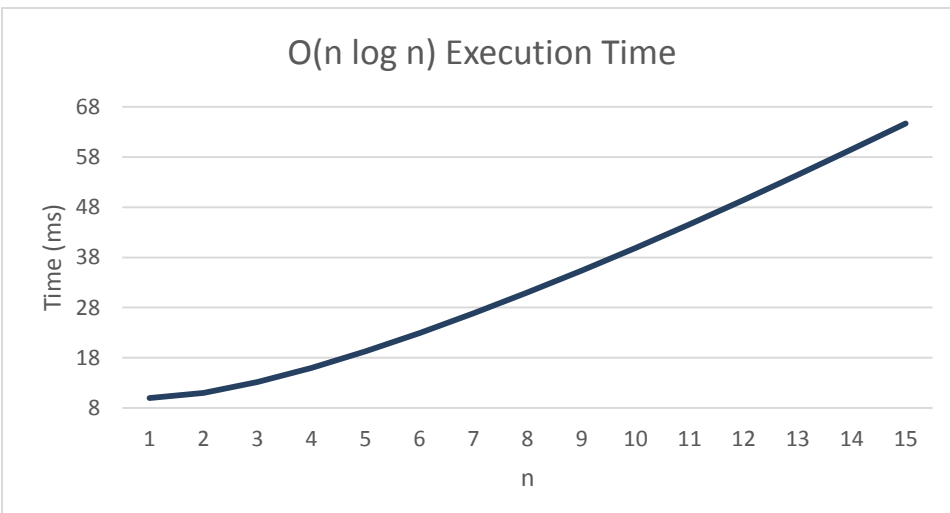
Some algorithms grow faster than linear but not much faster. For example, consider an algorithm that needs to perform an O(log n) search many times. If it needs to perform this search n times, then the algorithm is called O(n log n). Consider the following function:

```
bool sorted(double array[], int n)
{
   // look for each item in the list
   for (int i = 0; i < n; i++)                    // line 1
      if (!binarySearch(data, n, data[i]))        // line 2
         return false;                            // line 3
   return true;                                   // line 4
}
```

We can estimate the performance cost of this algorithm by counting how many times each line of code gets executed.

Line	Discussion	Cost
1	Executed once for every `n` in the array	O(n)
2	Since `binarySearch()` is O(log n) and it is executed n times, n × O(log n)	O(n log n)
3	At most this can be executed one time	O(1)
4	At most this can be executed one time	O(1)

Thus we can see that the overall cost of the function is O(n log n), which is the cost of the most expensive line of code. The graph of performance to data-set size shows that as `n` increases, the slope gradually increases. However, as `n` gets very large, the rate of increase becomes almost unnoticeable and appears linear.



$O(n^2)$

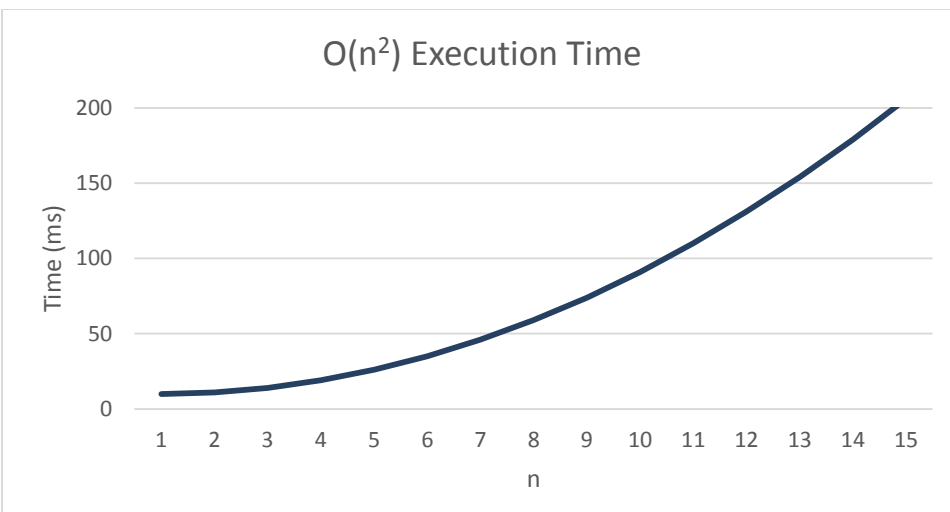
There are times when the cost of performing an action becomes much worse as `n` increases. There are examples in nature. For example the wind resistance on a car increases as a square of the speed. If you go twice as fast, it takes four times the amount of power to displace the air. As a general rule, programmers try to avoid algorithms that exhibit such a behavior. Consider the following code:

```
bool duplicatesExist(bool array[], int n)
{
   for (int i = 0; i < n; i++)                        // line 1
      for (int j = 0; j < n; j++)                     // line 2
         if (i != j && array[i] == array[j])          // line 3
            return true;                              // line 4
   return false;                                      // line 5
}
```

We can estimate the performance cost of this algorithm by counting how many times each line of code gets executed.

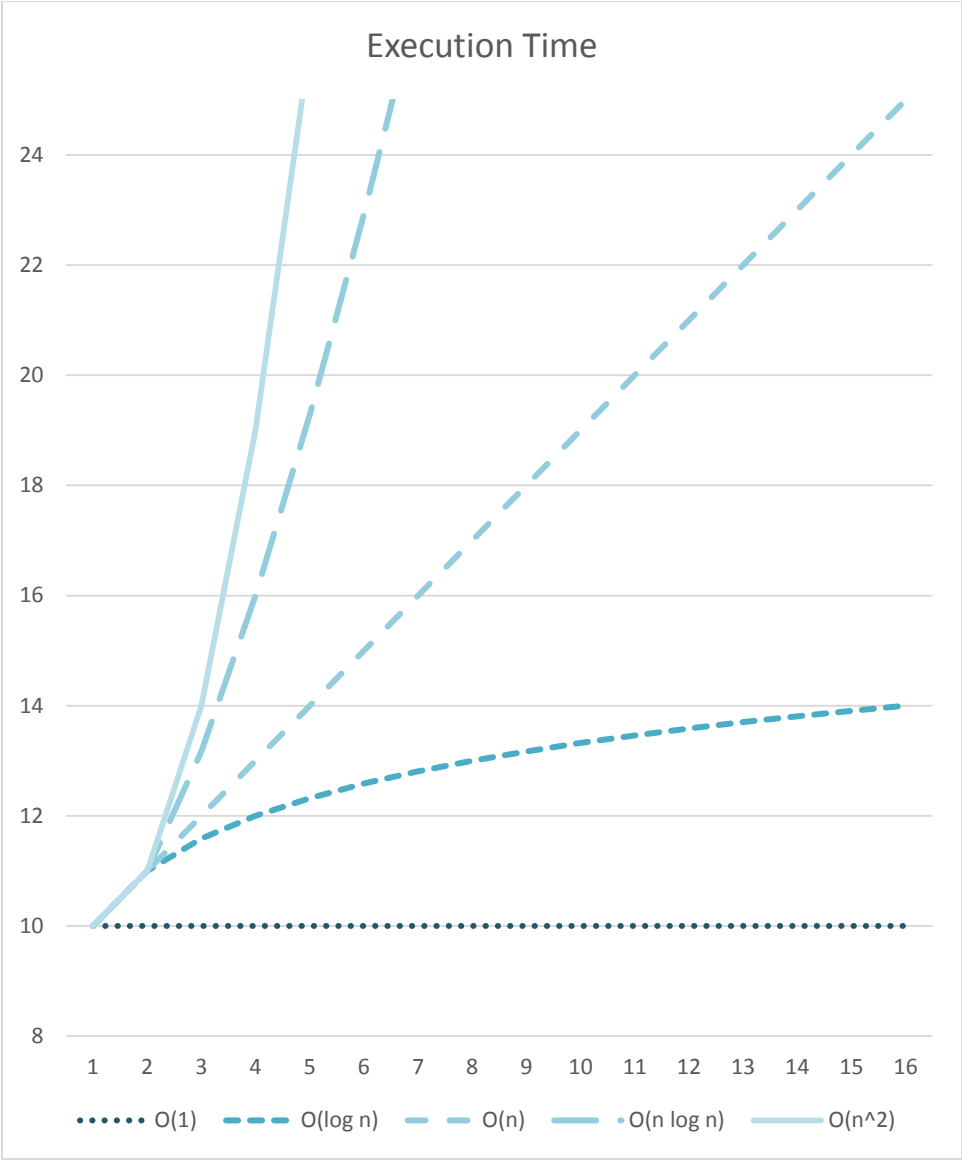
Line	Discussion	Cost
1	Executed once for every n in the array	$O(n)$
2	It will take $O(n)$ to complete this loop. However, it will be executed n times.	$O(n^2)$
3	It takes $O(1)$ to execute this like of code, but it will be executed n^2 times	$O(n^2)$
4	At most this can be executed one time	$O(1)$
5	At most this can be executed one time	$O(1)$

Thus we can see that the overall cost of the function is $O(n^2)$, which is the cost of the most expensive line of code. The graph of performance to data-set size shows that as `n` increases, the slope drastically increases.



Comparing algorithm complexity

This brief introduction to algorithmic complexity or "Big-O" does not cover all the complexities involved. The main point is to understand how to recognize O(1) vs. O(n) vs. O(n log n) etc. This is mostly accomplished by looking at the loops and seeing how n affects the number of times the body of the loops execute.



The second point is that O(1) and O(log n) are very similar. The same is true with O(n) and O(n log n). That being said, there is a huge difference between O(1) and O(n) and there is also a huge difference between O(n) and O(n^2). As programmers, we do all we can to avoid O(n^2) when an O(n) or O(n log n) algorithm is possible. Similarly we do all we can to avoid O(n) when an O(log n) or O(1) algorithm is possible.

00 : BUILT-IN DATA TYPES

A data structure is a format for organizing data. Generally, the term is associated with computers but it applies to other realms as well. For example, "11/12/1888" is a data structure associated with a date where the first field is the month (November), the second is the day of the month (the 12th), and the final is the year. Note that "12.11.1888" is the most common date data structure in Germany because they put the day of the month on the left. Data structures on computing systems are often far more complex than their physical world counterparts. They represent such things as songs, inventory parts, and bank transactions, as well as collections of songs, collections of parts, and collections of transactions. The data structures topic is important to computer science because a great deal of programming has to do with representing real-world constructs on a digital computer.

A data type is a specific format for representing data on a computer system. Since all data on a computer system is stored digitally as collections of 1's and 0's, there needs to be some way to translate this digital data into a format that is meaningful in a computing context. The data type specifies this translation. A data type specifies how to convert the concept of a number, a user's name, or a movie into 1's and 0's stored in memory or on a long-term storage device. It also specifies how these 1's and 0's can be translated back to the concept they are meant to represent. The concept of a data structure and a data type are closely connected. The data structure describes how data is organized and the data type describes the translation process.

There are three classifications of data types in a computing context:

- **Built-in data types**. These data types are so generic and are so universally useful that they are built into most computing systems and programming languages. Built-in data types are also called "primitive data types" or simply "primitives" because they are the building blocks of other data types. Examples of these include integers, characters, and floating-point numbers.
- **Custom data types**. These data types are designed for specific applications. If I were to build an application to play a card game, I might choose to create a data type to represent a single playing card. Custom data types are built in C++ using classes, structures, type definitions, and enumerations. They are a large focus of object oriented programming.
- **Abstract data types**. Often programmers build data types off of other data types. When these data types are built in such a way that they can work with just about any underlying data type, they are called Abstract Data Types (ADTs). Examples of these include arrays, linked lists, stacks, and vectors. These ADTs are the subject of most of this textbook.

We will begin our discussion of ADTs by first analyzing built-in data types. By learning how these data types are designed and used, we can better understand the design and use of more complex ADTs. Specifically, we will take a careful look at the three considerations: correctness, size, and performance. Recall that correctness is how well the data type honors the contract it is meant to fulfill. This has everything to do with the concept the data type is designed to represent as well as the expectations of the client. The size consideration is how much space it takes to represent the data type in memory. Built-in data types must be very size efficient so every bit is carefully used. The final consideration is performance. How much computational resources are required to perform the operations the client commonly requires?

Boolean Data

The simplest data-type to understand and to implement is the Boolean. This data-type was named after George Boole who explored the mathematical properties of logical data in his sentinel 1854 paper "An Investigation of the Laws of Thought on which are Founded the Mathematical Theories of Logic and Probabilities." This was then applied to the field of computer science by Claude Shannon in his 1938 paper "A Symbolic Analysis of Relay and Switching Circuits."

Boolean data is data that can have one of two values: true or false. This is used commonly in every-day life. Either I have completed class or I have not. Either I am male or I am not. Every time a decision needs to be made, some notion of Boolean data will need to be represented.

A Boolean data-type must have exactly two values, one corresponding to `true` and one for `false`. The operations supported by the Boolean data type are the following:

Name	Example	Description
Create	`bool b;`	Create a Boolean value.
Assignment	`b1 = b2;`	Make a copy of a bool.
Compare	`b1 == b2`	Returns `true` if the two Boolean values are the same.
Insertion	`cout << b;`	Display `1` if the value is `true`, `0` if the value is `false`.
Extraction	`cin >> b;`	Convert `0` to `false` and everything else to `true`.
Not	`!b`	Returns `true` when `false` and `false` when `true`.
And	`b1 && b2`	Returns `true` if both the left and right sides are `true`, `false` otherwise.
Or	`b1 || b2`	Returns `true` if either the left or right sides are `true`, `false` otherwise

Notice how the "standard" operations are above the thick black line. These correspond to the operations common to all built-in data types. The ones below the line are specific to Boolean data.

The method to encode the `true` value and the `false` value is arbitrary as long as it is unambiguous as to what corresponds to `true` and what corresponds to `false`. With digital systems, we use a `0` to represent `false` and `1` to represent `true` by convention. However, since most digital systems can access a unit of data no smaller than a byte (consisting of 8 bits), that leaves us with 256 possibilities instead of the required two (because $256 = 2^8$ where 8 corresponds to the number of bits in a byte and 2 corresponds to the use of digital data). The standard way to solve this problem is to correspond `00000000` to `false` and `00000001` to `true` (though all combinations of bits other than `00000000` are also treated as `true`).

Assignment

The assignment operator should work with the Boolean literal `false` on the right-hand-side (RHS) as well as it would with a Boolean variable.

	Code	Description	State
1	`bool b1 = false;`	Notice how the literal `false` corresponds to `00000000`.	`00000000`
2	`bool b2 = true;`	The literal `true` is `00000001`.	`00000001`
3	`bool b3 = b1;`	The assignment operator works with variables on the RHS.	`00000000`
4	`bool b4 = 47;`	Any integer but `0` becomes `true`. Note that `true` is always represented as `00000001`.	`00000001`
5	`bool b5 = (bool)'A';`	Again, any value but `0` becomes `true`.	`00000001`

The Boolean data type needs to be able to handle these conditions. Therefore, the pseudocode for the assignment operator would look something like this:

```
bool.assignment (lhs, rhs)
  IF rhs = 0 or rhs = false
    lhs ← false
  ELSE
    lhs ← true
```

Equivalent

On the surface, the equivalence operator should work by simply comparing the bits from the left-hand-side (LHS) and the RHS. However, this will not work correctly in all situations. Recall that the Boolean data-type can be no smaller than a byte and that all collections of bits having at least one `1` are equally considered true (only all `0`'s are considered `false`). This means that the following is true: `01000000 == 11110000` because `true == true`.

	Code	Description	State
1	`bool b1 = true;`	Simply assign `00000001` to `b1`.	`00000001`
2	`bool b2 = (b1 == b1);`	Because both the LHS and the RHS are the same, the result is `true`.	`00000001`
3	`bool b3 = (b1 == true);`	Works with literals as well as variables	`00000001`
4	`bool b4 = (b1 == (bool)9);`	The integer `9` will become `true` when cased to a `bool`.	`00000001`

The pseudocode for the equivalent operator for the Boolean data-type is:

```
Boolean.equivalent (lhs, rhs)
  IF lhs = 00000000 and rhs = 00000000
    RETURN true
  IF lhs ≠ 00000000 and rhs ≠ 00000000
    RETURN true
  ELSE
    RETURN false
```

Not

The users of the Boolean data-type will need to express the opposite of or negation of a Boolean value. Though the logical symbol for not is ¬, we use the exclamation point (`!`) in programming languages. For example, if a Boolean variable `isMale` is set to `true` for one individual, then the `isFemale` variable will have the opposite value.

	Code	Description	State
1	`bool isMale = !false;`	The literal false is `00000000`. `!false` is then `00000001`.	`00000001`
2	`bool isFemale = !isMale;`	The NOT operator works for variables as well as literals	`00000001`

The NOT operator is unary, taking a single operand (a.k.a. parameter). This operand is constant since it does not change. The pseudocode for the NOT operator for the Boolean data-type is:

```
bool.not (rhs)
  IF (rhs = 00000000)
    RETURN true
  ELSE
    RETURN false
```

And

The power of the Boolean data-type comes when combining variables to make Boolean expressions using Boolean algebra. One such operator is AND, which returns `true` if both parameters are `true`.

	Code	Description	State
1	`bool b1 = false && true;`	AND is picky. It will only evaluate to `true` if both sides are `true`.	`00000000`
2	`bool b2 = true && true;`	Since both sides are `true`, this evaluates to `true`.	`00000001`

The pseudocode for AND is:

```
bool.and (rhs, lhs)
  IF rhs ≠ false and lhs ≠ false
    RETURN true
  ELSE
    RETURN false
```

Or

The complement to AND is OR, which returns `true` if either of the operands are `true`.

	Code	Description	State
1	`bool b1 = false || true;`	OR is promiscuous, evaluating to `true` if either side is `true`.	`00000001`
2	`bool b2 = false || false;`	Since both sides are `false`, this evaluates to `false`.	`00000000`

The pseudocode for OR is:

```
Boolean.or (rhs, lhs)
  IF lhs = false AND rhs = false
    RETURN false
  ELSE
    RETURN true
```

Integers

Integers are positive and negative numbers lacking a decimal. They are formally defined as "all whole numbers and their opposites." Whole numbers are counting numbers "1, 2, 3, etc." with the addition of zero. The operations supported by the `int` data type are the following:

Name	Example	Description
Create Integer	`int i`	Create a signed 32 bit integer. There are other variations of this.
Create Short	`short s`	Create a 16 bit integer though can be larger. Can be declared as `short int`.
Create Long	`long l`	Create a large integer, at least 32 bits but always as big or bigger than `int`.
Create Long Long	`long long l`	Create an integer at least 64 bits in size. Can be declared as `long long int`.
Assignment	`i1 = i2`	Make a copy of an integer value.
Compare	`i1 == i2`	Return `true` if the two integer values are the same.
Insertion	`cout << i`	Convert the integer into a string of digits.
Extraction	`cin >> i`	Convert a string of digits into an integer.
Add	`i1 + i2`	Return the sum of two summands without changing either.
Subtraction	`i1 - i2`	Return the difference of a minuend and a subtrahend without changing either.
Multiplication	`i1 * i2`	Return the product of two multiplicands, leaving both unchanged.
Division	`i1 / i2`	Return the quotient of a dividend and a divisor, leaving both unchanged.
Modulus	`i1 % i2`	Return the remainder of a dividend and a divisor, leaving both unchanged.
Add Onto	`i1 += i2`	Sums two integers and puts the result in the LHS.
Subtract From	`i1 -= i2`	Subtract two integers and puts the result in the LHS.
Multiply Onto	`i1 *= i2`	Multiply two integers and puts the result in the LHS.
Divide By	`i1 /= i2`	Divide two integers and puts the result in the LHS.
Modulus From	`i1 %= i2`	Place the remainder of a dividend and a divisor into the dividend.
Negative	`-i`	Return the negative form of the RHS, leaving the RHS unchanged.
Increment Post	`i++`	Adds one to the integer but return the previous value.
Increment Pre	`++i`	Adds one to the integer and return the new value.
Decrement Post	`i--`	Subtract one to the integer but return the previous value.
Decrement Pre	`--i`	Subtract one to the integer and return the new value.
Greater Than	`i1 > i2`	Return `true` if `i1` is greater than `i2`, `false` otherwise.
Less Than	`i1 < i2`	Return `true` if `i1` is less than than `i2`, `false` otherwise.
Greater Or Equal	`i1 >= i2`	Return `true` if `i1` is greater than or equal to `i2`, `false` otherwise.
Less Or Equal	`i1 <= i2`	Return `true` if `i1` is less than or equal to `i2`, `false` otherwise.

Notice that there are a very large number of arithmetic operators. Some absent ones include absolute value, square root, exponents, trigonometric functions, and other mathematical operations.

All integer types can be declared with the `unsigned` modifier meaning the number is always positive.

In order to represent integers with a computer, we need the ability to translate the `1`'s and `0`'s of binary notation into numbers. To get an idea as to how we will do this, think about how we represent normal base-10 numbers. Consider the number 1,890. Here we recognize that the `0` is in the 1's place, `9` is in the 10's place, `8` is in the hundred's place, and `1` is in the thousand's place. In other words:

$$1890 = 1 \times 10^3 + 8 \times 10^2 + 9 \times 10^1$$

If we were to write this as an array of digits, it would be:

10^3	10^2	10^1	10^0
1	8	9	0

1000 + 800 + 90 + 0 = 1890

Thus we can store the number 1890 as:

```
int num[] = { 1, 8, 9, 0 };
```

Since computers are fundamentally binary, we don't use an array of digits but rather an array of bits. This means that each place in the array does not correspond to a power of 10 (for digits) but rather a power of 2 (for binary). The same number would be:

$$1890 = 2^{10} + 2^9 + 2^8 + 2^6 + 2^5 + 2^1$$

If we were to write this out as an array of bits, it would be:

2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	0	1	1	0	0	0	1	0

1024 + 512 + 256 + 0 + 64 + 32 + 0 + 0 + 0 + 2 + 0 = 1890

Thus we can store the number 1890 as:

```
bool num[] = { true, true, true, false, true, true, false, false, false, true, false };
```

The next problem we need to address is how to handle negative numbers. On the surface, this seems rather trivial. We will define a structure which consists of an array of Boolean values plus a Boolean sign variable.

```
struct Integer
{
   bool array[31];  // for the array of bits
   bool isNegative; // for negative numbers
};
```

There is a problem with this approach: there are two different representations for zero. The first would be an array of all zeros and `isNegative == false`. The second would be an array of all zeros with `isNegative == true`. Since `-0 == 0`, these two different integer representations would represent the same thing. In other words, I would have to do more than just compare the member variables to see if two `Integers` are the same. I would have to have a special IF statement to handle the zero case.

We can solve this problem with "**two's compliment**." With this scheme, we store an integer as an array of bits where the left-most bit corresponds to sign. If the bit is `0` then the number is positive and we can compute the value by summing the powers of 2 as we did in the above example. If the bit is 1, then things get more complicated. We first invert the bits (the complement part of "two's complement") and then add one.

For example, consider the 8-bit signed integer (called a `char` in C++) with the value -37. To find the binary representation, we will do the following:

Represent 37 as an 8-bit number:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	0	1	0	1

`0 + 0 + 32 + 0 + 0 + 4 + 0 + 1 = 37`

Find the compliment of all the bits:

1	1	0	1	1	0	1	0

Add 1.

1	1	0	1	1	0	1	1

Notice how the left-most bit is 1, indicating the number is negative.

This binary representation of signed (positive or negative) numbers has several desirable properties:

- First, there is only one `0` value. In other words, we don't run into the strange `-0 != 0` problem which was discussed previously.
- Second, it is easy to tell whether a number is negative: just check the left-most bit.
- Third, zero is represented as all `0`'s, which is the same test used for Boolean values.
- Finally, the increment and decrement operations work exactly the same for signed numbers as they do for unsigned numbers.

To illustrate, take a close look at the following table depicting the representation of 8-bit signed numbers between -4 and 4. We always add one to each binary number to get to the next number in the sequence. This is even true when going from -1 to 0 to 1.

Decimal	Binary							
-4	1	1	1	1	1	1	0	0
-3	1	1	1	1	1	1	0	1
-2	1	1	1	1	1	1	1	0
-1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0

Add

To understand how to add two `int`s, it is useful to reflect on how we add decimal integers. Recall the long addition algorithm you learned in elementary school. In this case, we will add 26 + 19:

```
   1
   2 6
 + 1 9
 -----
   4 5
```

Notice the small 1 in the top row. This is the carry value because 6 + 9 = 15. We will need to do this same carry operation when we add in binary.

			1			1		
	0	0	0	1	1	0	1	0
+	0	0	0	1	0	0	1	1
	0	0	1	0	1	1	0	1

This same process works with mixed positive and negative numbers. For example, we will add 26 + -19. We expect the value to be 7. Note how the left-most bit is dropped (called integer overflow). This is intentional and necessary for integer addition to work the way we expect.

	1	1	1	1				
	0	0	0	1	1	0	1	0
+	1	1	1	0	1	1	0	1
	0	0	0	0	0	1	1	1

Finally we can verify this works with two negative numbers. Here we will add -26 + -19. The value should be -45 of course.

	1	1		1	1			
	1	1	1	0	0	1	1	0
+	1	1	1	0	1	1	0	1
	1	1	0	1	0	0	1	1

We will verify this value by performing two's compliment back on the result. `11010011` becomes `00101100` when the complement is applied. From here, we will add one to `00101101`.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	1	0	1	1	0	1

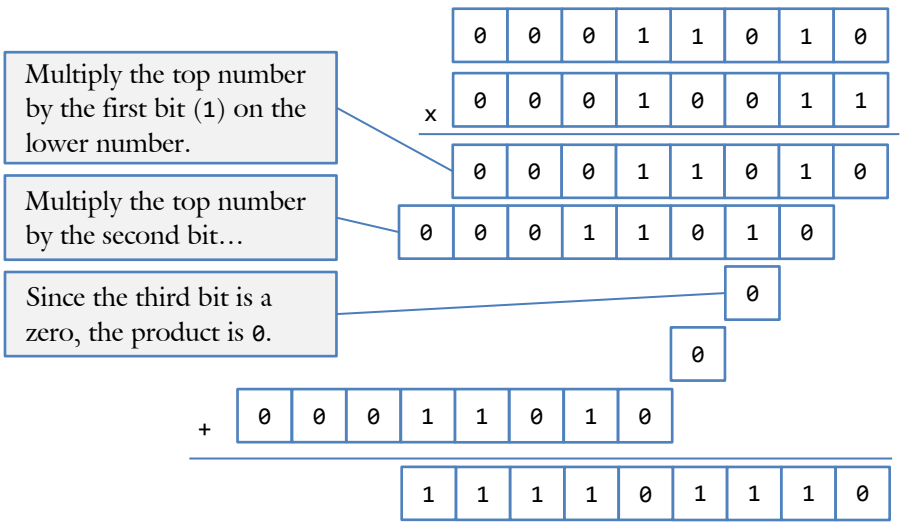
`0 + 0 + 32 + 0 + 8 + 4 + 0 + 1 = 45`

Multiplication

Binary multiplication works much like decimal multiplication. As with addition, we will first demonstrate integer multiplication then apply the same principle to binary numbers. We will start with: 26×19.

```
    5
    2 6
  x 1 9
  -----
  2 3 4
  2 6
  -----
  4 9 4
```

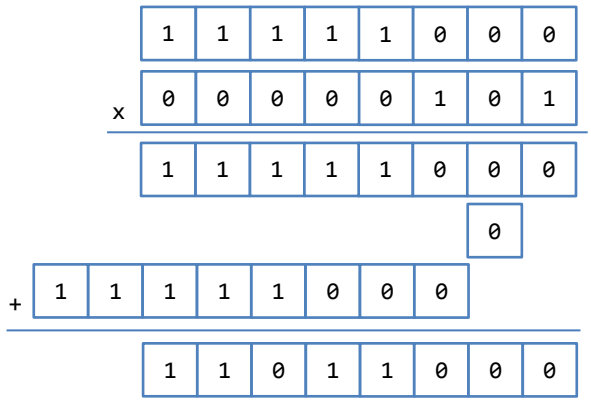
Notice how this is done in multiple steps. We compute $26 \times 9 = 234$ followed with $26 \times 1 = 26$. We will do the same thing in binary. Note that $0 \times 0 = 0$ and $0 \times 1 = 0$ and $1 \times 1 = 1$. In other words, one-bit multiplication works exactly like And.



There are three things to notice about this process. First, each step either makes a copy of the top number or the evaluation is zero. This means most of the work in the multiplication process is the addition at the end.

Second, the final number requires 9 bits to represent rather than the 8 bits we started with. This should not be surprising, 494 is larger than the maximum value for an 8-bit unsigned integer (256). The standard way to handle this is to truncate the resulting value. This is called overflow. In the case of our number, the result of the overflow would be an unsigned 238. If the number is signed, it would be -18.

The final issue is what to do about negative numbers. The answer is: nothing! Two's compliment "just works" with binary multiplication. To demonstrate this, consider: -8×5.



In the above example, `11011000` is -40 in two's compliment.

Characters

Originally computers were only used to perform mathematical operations. In fact, the name "computer" was first a job title for an individual who computed tide tables and artillery trajectories. When a machine was invented to do the same thing, the machine took on the name of the job of the person it replaced. It was not until the 1920's until people saw the need to represent textual data.

In order for a character data-type to fulfill the textual requirements of a programmer, the following operations must be supported:

Name	Example	Description
Create	`char c`	Create a character value.
Create Unicode	`char16_t c`	Create a Unicode character in the 16-bit format.
Create Unicode	`char32_t c`	Create a 32 bit Unicode character.
Assignment	`c1 = c2`	Make a copy of a character value.
Compare	`c1 == c2`	Return `true` if the two character values are the same.
Insertion	`cout << c`	Convert the ASCII value contained in `c` to the corresponding character.
Extraction	`cin >> c`	Convert the element in the input stream to the corresponding ASCII value.
Letter?	`isalpha(c)`	Return `true` if and only if the value is `A` through `Z` or `a` through `z`.
Digit?	`isdigit(c)`	Return `true` if and only if the value is 0 through 9.
Space?	`isspace(c)`	Return `true` if and only if the value is a space, tab, or newline.
Printable?	`isprint(c)`	Return `true` if the character is printable, `false` otherwise.
Uppercase	`toupper(c)`	Convert a lowercase letter into an uppercase letter, leaving the rest unchanged.
Lowercase	`tolower(c)`	Convert an uppercase letter into a lowercase letter, leaving the rest unchanged.
Increment	`++c`	Find the next character in alphabetical order.
Decrement	`--c`	Find the previous character in alphabetical order.
Greater Than	`c1 > c2`	Does the LHS come after the RHS in alphabetical order?
Less Than	`c1 < c2`	Does the LHS come before the RHS in alphabetical order?
Greater Or Equal	`c1 >= c2`	Is the LHS after or the same as the RHS in alphabetical order?
Less Or Equal	`c1 <= c2`	Is the LHS before or the same as the RHS in alphabetical order?

Note that it is possible to perform any arithmetic operation on a character that can be performed on an integer. However, the above list are those operations that make sense for a data type meant to represent a single character.

As with integers, the `unsigned` modifier can be added to `char`s. This has no effect on text representation; the only difference occurs when the character is used as an 8-bit integer.

At first every computer system had their own method for encoding letters. Some assigned the value 0 through 25 for 'A' through 'Z'. Others used 0 through 9 for '0' through '9' and then 10 through 35 for 'A' through 'Z'. This yielded over 60 distinct ways to encode these letters by 1963. Seeing this problem, Bob Bemer (an IBM engineer) proposed to the standards body ANSI a standard mapping from numbers to glyphs. This system was called ASCII for the American Standard Code for Information Interchange. President Lyndon B. Johnson made this a law for federal computers in 1968.

In order to create a mapping for numbers and integers a simple enumeration will suffice:

```
#define A 65
#define B 66
#define C 67
```

... code removed for brevity ...

```
#define Z 90
```

Notice that the number we assign to each letter is completely arbitrary, the only thing that is important is that everyone agrees to the same arbitrary mapping. In fact, if we are to honor the principle of encapsulation, this should be an implementation detail of which the clients need not to be aware.

In this international world, we have outgrown the 128-character space that ASCII provided. In 1987, a new standard called Unicode began development. This system uses a short (2 bytes or 16 bits) rather than a byte to encode characters. The result is 65,536 unique values, enough for hundreds of alphabets and languages. This standard has since been expanded to 4 bytes encoding 1,112,064 distinct characters.

Regardless of whether text is stored as one byte, two bytes, or four bytes, the algorithm for the common operations are the same. Note that the arithmetic operations on characters are built into the system and are no different than their integer equivalents. The functions that operate on them, on the other hand are not.

Is Alpha

The `isAlpha()` function returns true if the passed parameter is a letter, false otherwise. For ASCII values, this is quite simple:

```
isAlpha(character)
  RETURN a ≤ character ≤ z OR A ≤ character ≤ Z
```

Note that the Unicode version is quite more complex because there are so many scripts that have letters:

```
isAlpha(letter)
  RETURN  a ≤ character ≤ z
        OR A ≤ character ≤ Z
        OR À ≤ character ≤ ž
        OR α ≤ character ≤ ϡ
        OR ϧ ≤ character ≤ ϩ
        ...
```

This is problematic because there are literally thousands of letters and they do not necessarily exist in a predictable pattern in the Unicode table. Perhaps the best way to implement this function is to use a huge 64k array where a `1` bit corresponds to a letter and a `0` bit corresponds to everything else.

Real Numbers

Real numbers are distinct from integers in that decimals can be represented. Note that the mathematical representation of real numbers places no limits on the number of digits. There are three variations of real numbers in C++: `float`, `double`, and `long double`. The operations supported by these three data types are the same as those of integers:

Name	Example	Description
Create Float	`float f`	Create a "single precision" floating-point number, normally 4 bytes.
Create Double	`double d`	Create a "double precision" floating-point number, normally 8 bytes.
Create Long Double	`long double l`	Create an "extended precision" number of 10 or 16 bytes.
Assignment	`f1 = f2`	Make a copy of a real value.
Compare	`f1 == f2`	Return `true` if the two real values are the same.
Insertion	`cout << f`	Convert the real into a string of digits.
Extraction	`cin >> f`	Convert a string of digits into a real number.
Add	`f1 + f2`	Return the sum of two summands without changing either.
Subtraction	`f1 - f2`	Return the difference of two real numbers without changing either.
Multiplication	`f1 * f2`	Return the product of two multiplicands, leaving both unchanged.
Division	`f1 / f2`	Return the quotient of a dividend and a divisor, leaving both unchanged.
Add Onto	`f1 += f2`	Sums two reals and puts the result in the LHS.
Subtract From	`f1 -= f2`	Subtract two reals and puts the result in the LHS.
Multiply Onto	`f1 *= f2`	Multiply two reals and puts the result in the LHS.
Divide By	`f1 /= f2`	Divide two reals and puts the result in the LHS.
Negative	`-f`	Return the negative form of the RHS, leaving the RHS unchanged.
Increment Post	`f++`	Adds one to the reals but return the previous value.
Increment Pre	`++f`	Adds one to the real and return the new value.
Decrement Post	`f--`	Subtract one to the real but return the previous value.
Decrement Pre	`--f`	Subtract one to the real and return the new value.
Greater Than	`f1 > f2`	Return `true` if `f1` is greater than `f2`, `false` otherwise.
Less Than	`f1 < f2`	Return `true` if `f1` is less than than `f2`, `false` otherwise.
Greater Or Equal	`f1 >= f2`	Return `true` if `f1` is greater than or equal to `f2`, `false` otherwise.
Less Or Equal	`f1 <= f2`	Return `true` if `f1` is less than or equal to `f2`, `false` otherwise.

There are important differences between real numbers and floating-point numbers. The main difference is precision. For example 1/3 is 0.333 to an infinite number of 3's (represented as:$0.\overline{3}$). Similarly there are an infinite number of digits in π. Since it is difficult to represent infinity with a finite number of bits, the best we can do is approximate real numbers.

To understand how real numbers are represented on a computer, consider scientific notation. Here a number is broken into two parts: the decimal and the exponent.

$$14{,}973.42 = 1.497342 \times 10^5$$

The decimal component always has one digit to the left of the decimal point. When the decimal is in this format, it is called the "**canonical form**" because the number is in a standard (or canonical) format. The number of digits that extend to the right of the decimal point is the precision. As mentioned previously, we would need an infinite number of digits to accurately represent 1/3 which we obviously do not have. There is another way to represent scientific notation: the **E notation**:

```
1.497342e5
```

Notice the `e` before the last digit. This `e` serves two purposes. First, it announces that the number following is different than the number preceding. Namely, the number following is the exponent (the `e`). Second, the `e` separates the two components of the number: the decimal (`1.497342`) and the exponent (`5`).

Floating-point numbers work much the same way as scientific notation with one important exception: the numbers are represented in digital format (base-2) rather than in decimal format (base-10). The "decimal" component of scientific notation is called the "**mantissa**" or "**significand**" and the shorthand "`s`" refers to that field. For example `s=1.497342` is the significand of the number 14,973.42. The exponent of a floating-point number has the same name as that of scientific notation. The shorthand "`e`" refers to that field (`e=5`). In addition to this, there is an extra sign bit where `1` means negative and `0` means positive. Thus the three fields are:

+/-	Exponent	Significand

We can thus write the binary representation of a floating-point number as a field of three binary numbers:

```
0 10001100 11010011111010110101110
```

Exponent

The exponent is stored as a signed integer. However, rather than storing the signed value as two's compliment as we did for `int`s, it is stored as "**exponent bias**." This format was invented by IBM in 1954 to simplify comparing floating-point numbers.

Using the exponent bias format for an 8-bit number (as exponents are for a `float`), the integer 0 is `011111111`, 1 is `100000000` and 2 is `100000001` and so on. The advantage of this format is that it is possible to perform floating-point comparisons by comparing bits from the most significant (left most which is the sign bit) to the least significant (right-most) through the exponents and on through the significand.

Significand

Recall that scientific notation specifies that the decimal component will have one non-zero digit to the left of the decimal point (unless the number being represented is 0.0) to be in the canonical form. Floating-point numbers also have a canonical form. In this case, the left-most bit must be a 1. This is a waste of precious bits. We therefore *assume* there is a leading one and skip that field. Therefore we can represent a 24 bit number with 23 bits when we know the number is in canonical form. This is called the "**trailing significant field**."

Note that floating point format has a signing bit in the first slot. This means that the significand does not need to be signed. Thus it can be encoded in the standard unsigned integer format.

There are three types of floating-point numbers:

Format	Overall Size	Exponent Size	Significand Size
`float`	32	8	23
`double`	64	11	52
`long double`	128	15	112

Notice how the majority of the space allocated to a floating-point number is devoted to the significand. This is particularly evident in the larger sizes. Whereas a `float` devotes 25% of the bits to the exponent, the `long double` devotes only 11% of the bits.

Assignment

Converting real numbers into floating-point numbers is a two-step process. The number is first reduced to one large significand number, then the number is canonicalized. To demonstrate how this works, we will convert 14,973.42.

Since computers are fundamentally binary, we don't use an array of digits but rather an array of bits. This means that each place in the array does not correspond to a power of 10 (for digits) but rather a power of 2 (for binary). The same number would be:

$$1890.625 = 2^{10} + 2^{9} + 2^{8} + 2^{6} + 2^{5} + 2^{1} + 2^{-1} + 2^{-3}$$

If we were to write this out as an array of bits, it would be:

2^{10}	2^{9}	2^{8}	2^{7}	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	2^{-1}	2^{-2}	2^{-3}
1	1	1	0	1	1	0	0	0	1	0	1	0	1

`1024+ 512 + 256 + 0 + 64 + 32 + 0 + 0 + 0 + 2 + 0 + .5 + 0 +.125 =1890.625`

Notice how we represent the digits to the right of the decimal point with negative exponents. This would correspond to the floating-point number of `0 01111111 00000000011101100010.101`. For now, don't worry about the decimal in the binary representation. We are not quite done with the process. Let's check our work at this stage by plugging these numbers into our equation:

$$number = sign \times 2^{exponent} \times significand$$

This becomes (with the exponent encoded in the exponent bias notation) the following:

$$1890.625 = (+1) \times 2^{01111111} \times 00000000011101100010.101$$
$$= 1 \times 2^{0} \times 1890.625$$

The last step is to shift the decimal over until the significand is in the form of: `1.1101100010101`. This will get rid of the decimal point in our binary representation. With every shift of the decimal to the left, we will increase one number to the exponent.

a) `0 01111111 00000000011101100010.101`
b) `0 10000000 0000000001110110001.0101`

…

j) `0 10001001 0000000001.1101100010101`

Now that the decimal has migrated to the appropriate spot in the above step, the left-most `1` is removed and we get the final value: `0 10001001 11011000101010000000000`. We can verify that it is correct:

$$1890.628 = (+1) \times 2^{10001001} \times (1.)11011000101010000000000$$
$$= 1 \times 2^{10} \times 1.8463134765625$$

Consider Sue's dresser consisting of a collection of six drawers. When she puts her clean laundry away, she accesses the drawers by number (1st, 2nd, etc.) and in any order. This is also true when she picks out her clothes at the beginning of the day.

In many ways, this dresser behaves much like a vector. A vector is an ordered collection of elements which are accessed by index. The one major difference in this analogy is that a vector can grow to accommodate any number of elements where Sue's dresser can only ever have six drawers. The properties of a vector are:

- **Container**: The vector is designed to store a collection of data.
- **Random Access**: All elements are equally accessible by integer index.
- **Order Preserved**: The client specifies the order in which data is stored.
- **Expandable**: The capacity of the vector can be increased to meet the needs of the client.
- **Unrestricted Insertion**: It is possible to add an element to any location in the vector.

An array is a built-in data-type where elements reside in consecutive locations in memory. In most languages such as C++, the size of the array remains fixed. This is true whether the array is specified at compile-time or whether the array is dynamically allocated at run-time. If the user wishes to increase the size of an existing array, extra code must be written to do this. Vectors are abstract data-types designed to simplify this common task.

Definition

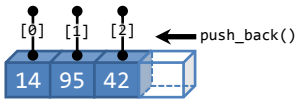
A vector is an array-like data-structure containing a collection of elements each referenced with an index. Vectors are different from arrays in that the capacity can increase at run-time. The common operations are:

Name	Example	Description
Create	`vector <int> v;`	Create a new vector. In this case, a vector of integers.
Assignment	`v1 = v2;`	Make a copy of a vector.
Empty	`f = v.empty();`	Returns `true` if the vector has no elements, `false` otherwise.
Size	`n = v.size();`	Returns the number of elements currently in the vector.
Capacity	`c = v.capacity();`	Returns the number of elements the vector can hold before reallocating.
Clear	`v.clear();`	Remove all elements from the vector.
Begin	`it = v.begin();`	Returns an iterator to the first element in the vector.
End	`it = v.end();`	Returns an iterator referring to the past-the-end element in the vector.
Push Back	`v.push_back(e);`	Adds an element to the end of the vector, thereby increasing size by one.
Access	`e = v[1];`	Access one element in the vector.

The operations above the thick line are common to almost all containers. Those below the line are specific to the vector. Note the shorthand in the above table. We will use this shorthand throughout the textbook:

Variable	Use
`v`	Short for `vector`. Also `s` is for `stack`, `q` for `queue`, and so on.
`f`	Flag which is another way of saying a Boolean value
`n`	Number of elements
`c`	Capacity
`it`	Iterator
`i`	Index into an array or counter in a loop
`e`	Element or one instance of a container

One way to visualize the vector is the following:



There are three things to notice about this figure. First, the blue cubes indicate elements that ara accessible by the client. In this case, all three elements are accessible through the square-bracket `[]` operator. With the vector, all elements can be accessed equally. If an element is ever not accessible, it will be indicated with a light blue cube. We represent access with vertical lines in the above figure.

Second, it is possible to add new elements to the vector through the `push_back()` method. Unlike the built-in array data-type which is fixed from the point of creation, the client can add elements to the vector at any time. We represent changes to the size of the vector with horizontal arrows. In this case, there is just one method to change the size: `push_back()`.

Third, this vector has three elements but has a capacity of four. The extra space is represented with a transparent cube. In most cases, the size and the capacity are the same. However, it does not have to be that way. In the Implementation section of this chapter we will learn of situations in which it is more efficient to make the capacity greater than the size.

Create

A vector is created using the default constructor (with the capacity set to zero), the non-default constructor (with the size specified), or the copy constructor.

```
vector <T> :: vector();
vector <T> :: vector(int size)                throw (const char *);
vector <T> :: vector(const vector <T> & rhs) throw (const char *);
```

The result of any of the above lines of code is an object of type vector. Recall that a class is the description of the data-type and an object is one instance of the data-type. We can also say that an object is a variable whose data-type is the class from which it was built.

In all cases, the data-type of the elements is specified. This is accomplished using the class template notation.

	Code	Description	State
1	`vector <char> v1;`	Default constructor. Notice how no data is allocated so there is no underlying buffer. Because this is a vector, the capacity may be increased later to accommodate the size needs of the client.	
2	`vector <int> v2(4);`	Non-default constructor. Here we are allocating a buffer size of 4 and filling it with the values of the default constructor of the base type. Since the base-type is an integer, the slots are filled with zeros.	0 0 0 0
3	`vector <int> v3(v2);`	Copy constructor. All the values of `v2` are copied into `v3`. Here `v3` has a buffer size big enough to handle all the elements of `v2`. In this case, there are four elements in `v2` so the size of `v3` is four. Note that if the capacity is greater than the size, then the size is copied but not the capacity.	0 0 0 0

It is also possible to initialize the values of the vector at declaration time.

```
vector <T> :: vector(int max, const T & initValue)      throw (const char *);
vector <T> :: vector(const T * pBegin, const T * pEnd)  throw (const char *);
```

	Code	Description	State
1	`vector <char> v1(4, 'a');`	Here we are creating a vector of four characters. We will initialize each slot with the value `'a'`.	a a a a
2	`char * text = "vector";` `vector <char> v2(text, text + 6);`	This constructor will compute the size of the capacity based on the spread of the `pBegin` and `pEnd` pointers. It will then copy the values from the source array into the new vector.	v e c t o r v e c t o r

Push Back

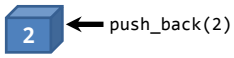
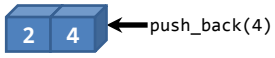
The capacity of a vector can be increased with the `push_back()` method.

```
void vector <T> :: push_back(const T & t) throw (const char *);
```

There are two main functions of `push_back()`:

a) The size of the vector will increase by one. This may or may not increase the capacity of the vector. If the capacity of the vector is larger than the size, then `push_back()` will not require the capacity to increase. On the other hand, if the capacity and the size are the same, then `push_back()` will require the vector to grow.
b) The last element in the vector will be given the value of the passed parameter.

The `push_back()` method is very important because it gives the client the easiest access to the reallocation feature that distinguishes vectors from arrays. In other words, the client can increase the size of the vector at any time by calling `push_back()`. Consider the following code:

	Code	Description	State
1	`vector <int> v;`	An empty vector is created with no buffer allocated.	
2	`v.push_back(2);`	By adding 2 to the vector, we will increase the size by one. This necessitates increasing the capacity by one.	
3	`v.push_back(4);`	By adding 4 to the vector, we will again increase the size. Since the capacity was 1, it will need to be increased again	

Observe in step 2 how the vector buffer is grown to accommodate the new value. The buffer is grown again in step 3 for value 4.

Note that there is another similar method called `insert()`. This will add an element into the middle of the vector and cause every element to the right to shift over by one slot. As you can imagine, this is expensive. In fact, it is always O(n) where n is the number of elements to the right of the newly inserted element. The syntax is:

```
iterator vector <T> :: insert(iterator it, const T & t) throw (const char *);
```

We specify the location with an iterator, and the return value is an iterator to the newly inserted element.

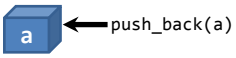
Standard Container Interfaces

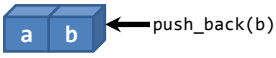
The vector maintains two different variables relating to the size of the underlying buffer: the **size** and the **capacity**. The size is the amount of space the user has specifically asked for and the capacity is the amount of space the container actually allocated.

```
int  vector <T> :: size() const;
int  vector <T> :: capacity() const;
bool vector <T> :: empty() const;
void vector <T> :: clear();
```

In most cases, the size and the capacity are the same for the vector. However, the vector may decide it is more efficient to allocate more space than is strictly required. Why would it do this? While this question will be addressed in more detail under the Implementation section, suffice it to say that there are valid performance reasons for increasing the capacity by more than one for a given allocation.

	Code	Description	Output	State
1	`vector <char> v;`	An empty vector is created		

2	`v.push_back('a');`
	`cout << v.size() << ' '`
	`  << v.capacity();`
		By adding an element, the size and capacity are increased to one	`1 1`	

3	`v.push_back('b');`
	`cout << v.size() << ' '`
	`  << v.capacity();`
		By adding another element, the size and capacity are increased again	`2 2`	

4	`v.push_back('c');`
	`cout << v.size() << ' '`
	`  << v.capacity();`
		Notice how we double the capacity with each reallocation	`3 4`	

5	`v.push_back('d');`
	`cout << v.size() << ' '`
	`  << v.capacity();`
		Again the size is increased by one. Now the size and capacity are the same	`4 4`	

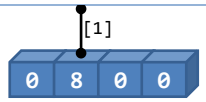
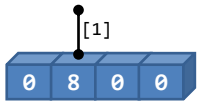
6	`cout << v.empty();`	Return true if the vector is empty, false otherwise.	`0`	
7	`v.clear();`	Remove all the elements from the vector. Notice how the size is zero but the capacity remains unchanged at 4.		
	`cout << v.empty();`	Now the vector is empty.	`1`	

Access

The main way to access elements in a vector is through the square-bracket operator. This allows the client to directly retrieve or set any element based on the provided index. In C++'s standard template library (STL), there are two variants. The first is for read-write access and the second is read-only.

```
      T & vector <T> :: operator [] (int index)       throw (const char *);
const T & vector <T> :: operator [] (int index) const throw (const char *);
```

The `[]` operator (pronounced "the square bracket operator") in the STL does not perform bounds checking. This means no error will be thrown if the client passes -1 as an index. There is a variant that does perform this check, called the `at()` method. Our implementation will break from the C++ standard and perform bounds checking with the square bracket operator.

	Code	Description	State
1	`vector <int> v(4);`	Allocate a vector with four elements. Note that they are all initialized to zero.	
2	`v[1] = 8;`	We are able to set the value of the 2nd element with the square bracket operator.	
3	`const vector <int> vConst(v);` `cout << vConst[1];`	The copy constructor will make a copy of `v`. This copy is `const` so it cannot be changed. Notice how the square bracket operator is able to retrieve a value from `vConst`.	

4	`v[-1] = 9;  // EXCEPTION!`
	`cout << v[4];  // EXCEPTION!`
		An exception is thrown when we attempt to access an element outside the valid range This is true for setters and getters.	

In step 1, we allocated four elements in the vector. Thus the valid indicies are 0-3. In step 2, we set one of the elements to the value 8. Notice how we do not need to fill element 0 first; we can access the elements in the vector in any order we choose.

In step 3, we are retrieving an element from the vector. This should work even if our vector is constant. In other words, there needs to be a version of the square bracket operator that allows access to the elements in the vector without making a change to the member variables.

Observe how step 4 performs bounds checking. This is different from how the standard template library implements the `std::vector` class. The standard template library uses the `at()` method to accomplish this.

Vector Iterator

The `Vector` class provides two main ways to access elements: the index method using the square bracket operator and iterators (See "Appendix G. Review: Iterator" for a refresher on iterators). Recall that iterators are classes built in C++ allowing us to use a single standard loop to iterate through all possible containers, regardless of their internal implementation. A `for` loop using the `vector` iterator would be the following:

```
for (vector <int> :: iterator it = v.begin(); it != v.end(); ++it)
   cout << *it << endl;
```

The two methods that need to be implemented on the vector side are `begin()` and `end()`:

```
vector <T> :: iterator vector <T> :: begin();
vector <T> :: iterator vector <T> :: end();
```

The return type is:
`vector <T> :: iterator`
Notice how the template variable is attached to `vector`

Both the `begin()` and `end()` methods are members of the vector class. Thus the full name is:
`vector <T> :: begin()`
`vector <T> :: end()`

In addition to these two methods, a `vector <T> :: iterator` class needs to be implemented. Note that iterator is a nested class defined within vector. Please see "Appendix H. Review: Nested Class" for details how to implement a nested class.

Declaration

The standard template library allows us to define a vector iterator as:

```
vector <int> :: iterator it;
```

Observe how the iterator is a template class taking the same data-type as the associated `vector` container. There are three other iterators you may encounter:

```
vector <int> :: const_iterator         it;
vector <int> :: reverse_iterator       it;
vector <int> :: const_reverse_iterator it;
```

Constant iterators are much like iterators except they do not provide the functionality of changing the data structure to which they refer. In other words, they are read-only. This is true both for `const_iterator` and `const_reverse_iterator`.

Reverse iterators are much like normal iterators except they go through the container backwards. In the case of the vector, they will iterate through the elements from the largest indice to the smallest. This is true both for the `reverse_iterator` which provides read-write access to the elements and `const_reverse_iterator` which is read-only.

Begin

The `begin()` method returns an iterator to the first element in the `vector`. Most uses of the `begin()` method set the return value to a variable which will be used in a FOR loop.

```
vector <T> :: iterator <int> it = v.begin();
```

There are several variations of the `begin()` method:

```
vector <T> :: iterator               <int> it = v.begin();
vector <T> :: const_iterator         <int> it = v.begin()   const;
vector <T> :: const_iterator         <int> it = v.cbegin()  const;
vector <T> :: reverse_iterator       <int> it = v.rbegin();
vector <T> :: const_reverse_iterator <int> it = v.rbegin()  const;
vector <T> :: const_reverse_iterator <int> it = v.crbegin() const;
```

If you would like to have read-only access to the vector, you need to use a constant iterator. This can be initialized through either the const version of `begin()` or through `cbegin()`.

You can get access to a reverse iterator through the use of the `rbegin()` method. This has both the read/write variant using the `reverse_iterator` class or the const variant using the `const_reverse_iterator` class.

Finally, you can get a constant version of the reverse iterator either through the `rbegin()` or the `crbegin()` method. Both require use of the `const_reverse_iterator` class.

End

The `end()` method indicates when a loop should be terminated. This is accomplished by returning an iterator to the first invalid element in the container. In other words, `end()` does not refer to the last element in the container but rather the first slot *after* the last element.

```
if (it == v.end())
   cout << "He's dead, Jim!";
if (it != v.end()
   cout << "Not dead yet!";
```

As with begin, there are several variations of the `end()` method:

```
vector <T> :: iterator               <int> it = v.end();
vector <T> :: const_iterator         <int> it = v.end()   const;
vector <T> :: const_iterator         <int> it = v.cend()  const;
vector <T> :: reverse_iterator       <int> it = v.rend();
vector <T> :: const_reverse_iterator <int> it = v.rend()  const;
vector <T> :: const_reverse_iterator <int> it = v.crend() const;
```

Increment

The iterator advances to the next element in the container with the `++` operator:

```
++it;     // both should work
it++;
```

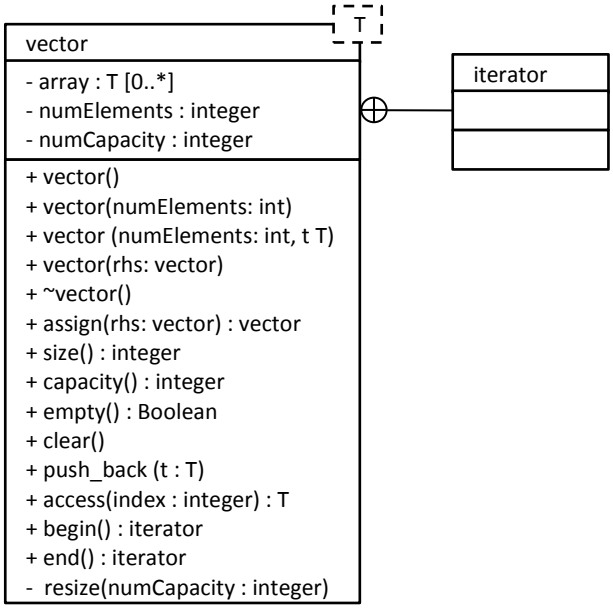
The vector iterator also allows the client to move backwards through the container:

```
--it;     // both should work
it--;
```

Note that the `++` operator for the `iterator` class works exactly the same as the `--` operator for the `reverse_iterator` class (and vice-versa). These methods are defined for the `iterator`, `const_iterator`, `reverse_iterator`, and the `const_reverse_iterator`.

Implementation

Because a vector is much like an array, it should be no surprise that the underlying data-structure is an array. The two other required member variables are size and capacity. The size is number of elements in the vector and is represented with the member-variable `numElements`. The capacity refers to the number of elements that can be held in the currently allocated storage and is represented with the `numCapacity` member-variable. The UML class diagram for our `Vector` class is:



Notice the `iterator` class to the right. This, coupled with the ⊕ symbol, indicates that `iterator` is a nested class. Please see Appendix H. Review: Nested Class for details as to how this work.

Standard Container Interfaces

The `size()` and `capacity()` methods will be simple getters returning the value of `numElements` and `numCapacity` respectively. The `empty()` method will simply return whether `numElements` is zero:

```
vector.empty()
  IF numElements = 0
    RETURN true
  ELSE
    RETURN false
```

The method `clear()` will remove all items from the `Vector` and set `numElements` to zero. This introduces the question: what will `clear()` do to the capacity? One option would be to set the capacity to zero and free the allocated buffer. However, any subsequent use of the vector will necessitate a new allocation. Since allocations are expensive from a performance standpoint, this is undesirable. A more efficient implementation is to simply leave the capacity unchanged.

```
vector.clear()
  numElements ← 0
```

Since elements are effectively abandoned and not destroyed, this may not be what the client intended. If the client wishes to wipe out that memory, he or she would need to assign values over each element to be eradicated before executing `clear()`. The standard-template-libarary version of `clear()` will call the destructor on those elements abandoned. This requires more sophisticated memory-management than can be presented here.

Assignment

The assignment operator makes a copy of one `vector` into another. Note that we need to overload this operator; otherwise C++ will simply copy the pointer and we wll have two `vector`s referring to the same buffer.

If the source and destination `vector` are the same size, this seems very straightforward: simply copy over the values from the source `vector` and place them in the corresponding location in the destination. However, what does one do when the sizes are not the same? Specifically, what does one do when the destination `Vector` has a smaller capacity than the source size? The answer is that we need to reallocate.

The process of reallocating involves creating a new buffer of suffient capacity to accomidate the desired size. When this new buffer is created, the values from the old buffer are copied over one at a time. Finally, the old buffer is deleted. The pseudocode is:

```
vector.resize(cNew)
  arrayNew ← new(cNew)
  FOR i ← 0 … numElements
    arrayNew[i] ← array[i]
  delete(array)
  array ← arrayNew
  numCapacity ← cNew
```

The next problem we need to solve is what to do when the source `vector` is smaller than the destination `vector`. On the surface it may seem that the best solution is to reallocate to the smaller size. However this results in a reallocation that is not really necessary. In other words, a `Vector` may have a larger buffer than necessary and still honor the `Vector` contract. Thus both the source and the destination `Vector`s must both be the same size and have the same values in the elements, but they do not need to be the same capacity. The pseudocode for the assignment operator for `Vector` is:

```
vector.assignment(rhs)
  numElements ← 0
  IF rhs.size() > capacity()
    resize(rhs.size())
  numElements ← rhs.size()
  FOR i ← 0 … numElements-1
    array[i] ← rhs.array[i]
```

Notice how we set `numElements` to zero as the first step. This will prevent the old data from being copied in the event that the buffer is reallocated.

Square Bracket

Perhaps the most useful method for the `vector` is the square bracket operator also known as the array index operator. Here, unlike the standard template library, we will do some bounds checking to ensure the client passes an index within the valid range.

```
vector.square-bracket (i)
  IF 0 ≤ i < numElements
    RETURN array[i]
 THROW error
```

There is one complication: there needs to be two implementations of this operator. One must be a constant method that returns a copy of the value in the element. This functions as a getter only. The second must return a reference to the value so the method can function as a setter. While the pseudocode for both are the same, in C++ they are implemented as two overloaded methods.

Push Back

The `push_back()` method is guaranteed to increase the size of the `vector`. Howevever it may not need to increase the capacity. In other words, if there is excess capacity, `push_back()` may only need to increment the value of `numElements`. What happens when the buffer must be reallocated? There are two common choices: increase the capacity by one or double the capacity.

Our first attempt will be to call a `resize()` function which will grow the stack by one element.

```
vector.push_back(e)
  IF size() = capacity()
    resize(capacity() + 1)
  array[numElements++] ← e
```

This appears efficient from a space point of view. We use the minimum amount of space and the cost of resizing is O(n) because the entire array needs to be copied to a newly allocated buffer. However, if one were to add `n` elements to the vector, then the performance is $O(n^2)$. This is because we have to perform `n` operations costing O(n). Thus, the total cost is n x O(n) = $O(n^2)$.

A better approach is to double the capacity of the `vector` with each reallocation:

```
vector.push_back(e)
  IF size() = capacity()
    resize(capacity() * 2)
  array[numElements++] ← e
```

Here we are wasting some space, but we have to allocate much less often. If one were to add 1,000 elements to a `vector`, it would be necessary to reallocate not 1,000 times but rather 10 times (at 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512). In other words, the reallocation would have to occur $\log_2 n$ times because $2^{10} = 1024$. This means the cost of adding 1,000 elements to the `Vector` is O(n log n) which is much better than $O(n^2)$. Consider the following code:

```
{
   vector <int> v;
   for (int i = 1; i < 10; i++)
   {
      v.push_back(1);
      cout << "size:" << v.size() << " capacity:" << v.capacity() << endl;
   }
}
```

The output would be the following:

```
size:1 capacity:1
size:2 capacity:2
size:3 capacity:4
size:4 capacity:4
size:5 capacity:8
size:6 capacity:8
size:7 capacity:8
size:8 capacity:8
size:9 capacity:16
```

Observe how the capacity doubles once we attempt to `push_back()` when the vector is full.

Vector Iterator

The iterator version of the standard FOR loop for an array needs to work much like the non-iterator version of the loop. To see how this works, consider how to iterate through an array using the pointer notation:

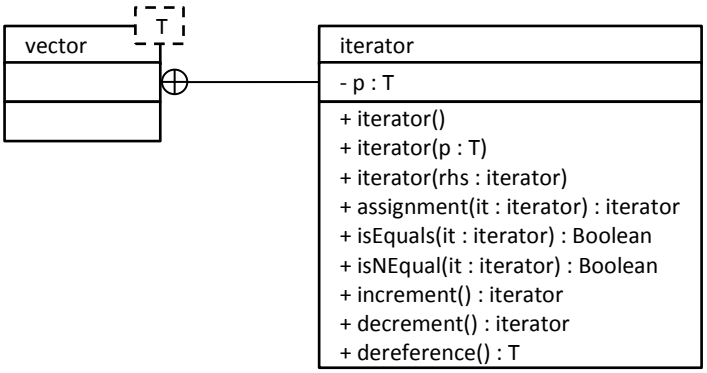
```
for (int * p = array; p != array + n; ++p)
   cout << *p;
```

Observe how `p` will point to the first element in the array initially (through `p = array`). It will then move through the array one element at a time (through `++p`) until the slot after the last element is reached (through `p != array + n`). Access to the value in each element is done through the dereference operator (through `*p`). This version of the loop will work the same as the iterator.

```
for (vector <T> :: iterator it = v.begin(); it != v.end(); ++it)
   cout << *it;
```

As with the pointer loop above:

- The `vector <T> :: iterator` will use a pointer to move the through the loop being the member variable within `iterator`.
- The loop will start by pointing to the first element in the vector where `it = v.begin()` works like `p = array`.
- `iterator` will move through the array one element at a time with `++it` working exactly like `++p`.
- The loop will continue as long as `it` does not point to the first element off the end of the array. The first element off the end is specified as `array + n` in the pointer loop and `v.end()` in the iterator loop.
- Data within the loop is reached with the dereference operator. It is dereferenced with `*p` in the pointer loop and calls the overloaded dereference operaetor (`*it`) which does the same thing in the iterator loop.



Here `vector <T> ::iterator` will store a pointer to a location in the array. Note that the `iterator` class is a nested class of `vector` (See Appendix G. Review: Iterator for a details). The `vector` class is a template class (we can tell because we instantiate an object with `vector <int> v;`) but the `iterator` class is not (we can tell because there is no `<int>` when we instantiate an `iterator` object with `vector <int> :: iterator it;`).

Create and Copy

The non-default constructor of the `vector <T> :: iterator` and the assignment operator both expect a pointer to a `Vector` element as a parameter. This will get assigned directly into the member variable `p`.

```
iterator.assignment(pLHS)
  p = pLHS;
```

Note that we expect `pLHS` to be a pointer to the first element in the `Vector`.

```
vector.begin()
  RETURN iterator(array)
```

On the part of vector `iterator`, we need to define several methods:

- The default constructor setting `p` to `NULL`.
- A non-default constructor taking a pointer to an element in the vector as a parameter.
- A copy constructor.
- An assignment operator taking a `vector <T> :: iterator` on the RHS.
- An assignment operator taking a pointer to a vector element on the RHS.

On the part of Vector, we need to define two methods:

- `begin()` which refers to the first element in the vector for forward traverses.
- `rbegin()` or "reverse begin" which refers to the last element in the vector for reverse traverses.

Compare

The comparison operator only accepts another vector `iterator` as the LHS parameter. It simply compares the values of the `p` member variable.

```
iterator.equivalent(lhs)
  RETURN p = lhs.p
```

For this to work in the iterator loop, the `lhs` variable must be a vector `iterator` pointing to the first element of the end of the vector. This means we need `vector::end()` to create a vector `iterator` pointing to that slot:

```
vector.end()
  RETURN iterator(array + n)
```

On the part of `vector <T> :: iterator`, we need two comparision operators to be defined: equals and not equals. On the part of vector, we also need two methods defined:

- `end()` which refers to the first element off the end of the vector for forward traversal.
- `rend()` or "reverse end" which refers to the first element off the beginning of the the vector for reverse traveral. This will be the `array - 1` location.

Increment

Vectors, just like arrays, guarantee that all the elements reside in a continuous block of memory. This means that if `p` points to the `i`th element in a vector, then `p + 1` points to element `i + 1`. Since the member variable of our vector `iterator` is a pointer, we simply need to increment the pointer in the increment operator.

```
iterator.increment()
  p++
```

Dereference

The final method to be defined by vector iterator allows access to the element to which the iterator is referring. Traditionally this is accomplished by overloading the dereference operator. Since the underlying member variable of the vector iterator is a pointer to a vector element, all that is needed is to dereference the pointer.

```
iterator.dereference()
  return *p
```

Application

There are as many uses for a vector as there are for an array. It would be impossible to enumerate all the uses! Instead an example will be shown that emphasizes the growing nature of vectors.

Fibanocci

Leonardo Fibanocci was a mathematician who lived in Pisa Italy in the 1200's. One of his many contributions to the field of mathematics was to describe a sequence of numbers where every value in the sequence is the sum of the previous two elements. This can formally be expressed as:

$$F(n) := \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

We will use a vector to solve this problem.

```
vector <int> getFibonacciSequence(int n)
{
   vector <int> sequence;

   // fill in the first two numbers
   if (n >= 1)
      sequence.push_back(0);
   if (n >= 2)
      sequence.push_back(1);

   // fill in the rest
   for (int i = 2; i <= n; i++)
      sequence.push_back(sequence[i - 1] + sequence[i - 2]);

   return sequence;
}
```

There are several things to notice about this solution. The first is that it is possible to return `Vector` by-value. This is possible because the copy constructor is defined for `vector`. In other words, we are not just returning a pointer to a freed array as we would if we used a simple array instead.

The second thing to notice is that we can access individual members of the sequence using the square bracket operator `[]`. This affords random access, a feature not present in many of the other data structures we will learn about in later chapers.

The final thing to notice is that the vector grows to accommodate whatever size we need. This is accomplished in the `push_back()` method. Of course this is very convenient; there is always enough space in the vector (or the vector will reallocate its buffer to accommodate a larger size). In this application, we do know the eventual size of the resulting array. The passed parameter `n` specifies this size. We can thus avoid the calls to `realloc()` that are inherit in the `push_back()` method with the following change to the way `sequence` is allocated:

```
   vector <int> sequence(n);
```

To illustrate how to generate a vector of Fibanocci numbers and to demonstrate how the buffer for a `vector` grows as elements are added, consider the following call to the Fibanocci function:

```
vector <int> result = getFibonacciSequence(8 /* n */);
```

How would `getFibonacciSequence()` compute the vector of Finonacci numbers? We will go through the algorithm one step at a time. For clarity's sake, the algorithm is repeated:

```
vector <int> getFibonacciSequence(int n)
{
   vector <int> sequence;

   // fill in the first two numbers
   if (n >= 1)
      sequence.push_back(0);
   if (n >= 2)
      sequence.push_back(1);

   // fill in the rest
   for (int i = 2; i <= n; i++)
      sequence.push_back(sequence[i - 1] + sequence[i - 2]);

   return sequence;
}
```

	Description	State
1	The first IF statement is executed. Since `n >= 1`, we add `0` to the vector. This will cause `realloc(1)` to be called to make `size == capacity == 1`.	0 0
2	The second IF statement is executed. Since `n >= 2`, we add `1` to the vector. This will cause another reallocation to be called making `size == capacity == 2`.	0 1 0 1
3	After the first iteration of the loop, we call `push_back(1)`. Since the vector was at full capacity, another reallocation is necessary. The buffer will be doubled this time so `capacity() == 4` and `size() == 3`. There is one empty slot at the end of the vector.	0 1 2 0 1 1
4	After the second iteration, we call `push_back(2)`. Once again the vector is full.	0 1 2 3 0 1 1 2
5	With the next call to `push_back()`, we find the need to reallocate. Since the old capacity was 4, we will call `reallocate(8)`. This will yield 3 empty slots at the end of the vector.	0 1 2 3 4 0 1 1 2 3
6	The previous two elements in the vector were `2` and `3`. Thus we call `push_back(5)`. Since there were 3 empty slots, we will not need to reallocate.	0 1 2 3 4 5 0 1 1 2 3 5
7	The previous two elements in the vector were `3` and `5`. Thus we call `push_back(8)`. Since there were 2 empty slots, we will not need to reallocate.	0 1 2 3 4 5 6 0 1 1 2 3 5 8
8	At the end of this iteration, our vector is once again full. This means `capacity() == 8` and `size() == 8`. Since `n == 8`, we are done with our function. If we were to perform one more loop and call `push_back(21)`, then the vector would call `reallocate(16)` and the buffer would double in size once again.	0 1 2 3 4 5 6 7 0 1 1 2 3 5 8 13

Examples

Example 01.1

What is the state of the vector at the end of execution?

```
{
   vector <char> v;          // #1
   s.push_back('A');         // #2
   s.push_back('B');         // #3
   s.push_back('C');         // #4
}
```

Answer:

#1		Initially empty
#2	0: A	Capacity goes to 1 and the first slot is filled
#3	0: A, 1: B	Capacity doubles and the second slot is filled
#4	0: A, 1: B, 2: C, 3:	Capacity doubles again to 4. The third slot is filled but the fourth remains uninitialized

Example 01.2

What is the state of the vector at the end of execution?

```
{
   vector <int> v(5);        // #1
   v[2] = 10;                // #2
   v[0] = 30;                // #3
   s.push_back(40);          // #4
}
```

Answer:

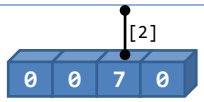
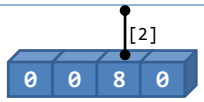
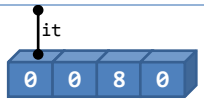
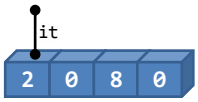
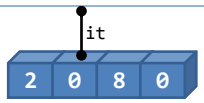
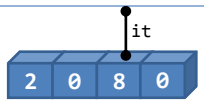
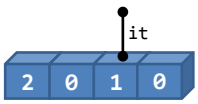
#1	0: 0, 1: 0, 2: 0, 3: 0, 4: 0	The size is set to 5 and all the elements are initialized to the default value for the data-type. In this case, the value is 0.
#2	0: 0, 1: 0, 2: 10, 3: 0, 4: 0	We can update the values in the vector in any order.
#3	0: 30, 1: 0, 2: 10, 3: 0, 4: 0	Again, we can update any element in the vector using the [] operator
#4	0: 30, 1: 0, 2: 10, 3: 0, 4: 0, 5: 40, 6: , 7: , 8: , 9:	Since the size equals the capacity, we need to grow the vector. This will double the capacity from 5 to 10. However, the size will only increase by one. The other empty slots will remain uninitialized.

Example 01.3

What is output of the following code?

```
{
   vector <int> v(4);                          // #1
   v[2] = 7;                                   // #2
   v[2] = 8;                                   // #3
   vector <int> :: iterator it = v.begin();    // #4
   *it = 2;                                    // #5
   it++;                                       // #6
   it++;                                       // #7
   *it = 1;                                    // #8

   cout << v[0] << ", "
        << v[1] << ", "
        << v[2] << ", "
        << v[3] << endl;
}
```

Answer:

#1		The size is set to 4 and all the elements are initialized to the default value for the data-type. In this case, the value is 0.
#2		The third element in the vector is set to the value 7.
#3		We update the third element again. This time we set the value to 8.
#4		We initialize an iterator and set it to the beginning of the vector.
#5		Through the iterator we can modify the vector. In this case, we are changing the first element. The same thing could be accomplished with the following code: `v[0] = 2;`
#6		The ++ operator advances the iterator to the right by one slot.
#7		Another instance of the ++ operator advances the iterator again. Now it is pointing to the third slot.
#8		We again modify the vector through the iterator. This has the same effect as the following code: `v[2] = 1;`

At the end of execution, the contents of the vector are displayed on the screen:

```
2, 0, 1, 0
```

Exercises

Exercise 01.1

Consider the following code:

```
{
   vector <int> v;

   v.push_back(15);
   v.push_back(3);
   v.push_back(15);

   cout << v.size() << endl;
}
```

What is the output of the program?

Answer:

_____

Please see page Push Back27 for a hint.

Exercise 01.2

Consider the following code:

```
{
   vector <int> v;

   v.push_back(10);
   v.push_back(20);
   v.push_back(30);

   cout << v[0] << ", " << v[1] << ", " << v[2] << endl;

   v[2] = 40;

   cout << v[0] << ", " << v[1] << ", " << v[2] << endl;
}
```

What is the output of the program?

Answer:

_____

_____

Please see page 29 for a hint.

Exercise 01.3

Consider the following code:

```
{
   vector <string> v;

   v.push_back(string("Vector"));
   v.push_back(string("Stack"));
   v.push_back(string("Queue"));
   v.push_back(string("Deque"));
   v.push_back(string("Set"));
   v.push_back(string("List"));
   v.push_back(string("Map"));

   for (int i = 0; i < v.size(); i++)
      cout << v[i] << ' ';
   cout << endl;
}
```

What is the output of the program?

Answer:

_____

Please see page 29 for a hint.

Exercise 01.4

Consider the following code:

```
{
   vector <string> v;

   v.push_back(string("Vector"));
   v.push_back(string("Stack"));
   v.push_back(string("Queue"));
   v.push_back(string("Deque"));
   v.push_back(string("Set"));
   v.push_back(string("List"));
   v.push_back(string("Map"));

   for (vector <string> :: iterator it = v.begin(); it != v.end(); ++it)
      cout << *it << ' ';
   cout << endl;
}
```

What is the output of the program?

Answer:

_____

Please see page 30 for a hint.

Exercise 01.5

Consider the following code:

```
{
   vector <char> v;

   for (int i = 1; i <= 10; i++)
   {
      v.push_back('z');
      cout << "  size:"     << v.size()
           << "  capacity:" << v.capacity() << endl;
   }
}
```

What is the output of the program?

Answer:

_____

_____

_____

_____

_____

_____

_____

_____

_____

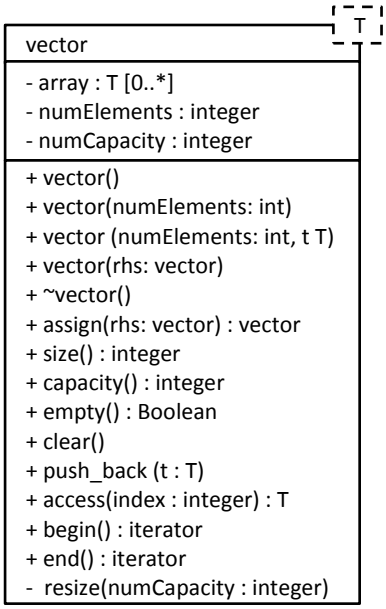
_____

Please see page 34 for a hint.

Problems

Problem 01.1

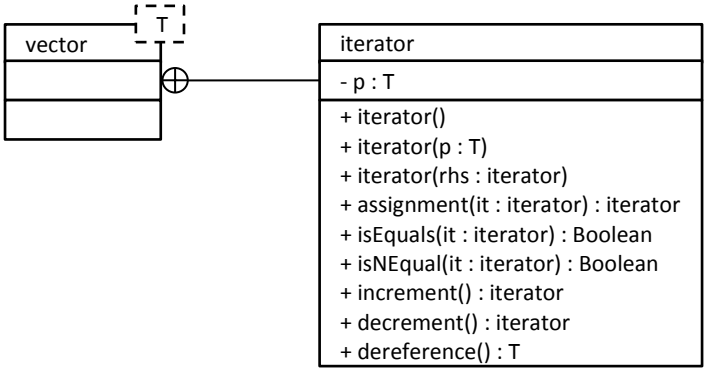
Implement your own version of the `std::vector` data structure. Call your implementation `custom::vector`. In other words, place your class in the custom namespace and implement all of the interfaces described in this chapter. The UML class diagram of the vector class is the following:



Please see page 32 for a hint.

Problem 01.2

Implement an iterator for the `vector` class created in Problem 01.1. This iterator should define the following methods:



Note that the iterator should be a nested class within the `vector` class.

As a challenge, implement a constant iterator associated with the `vector` class. This class will have the same interfaces as `vector <T> :: iterator` except the derefence operator will be a constant method. This can be implemented as return by-value or as a constant return by-reference.

Please see page 35 for a hint.

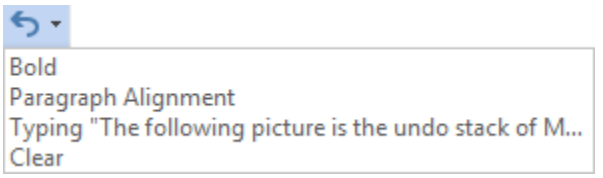
Like most people, Sam keeps his bowls and plates stacked in the cupboard of his kitchen. When he wants to eat his morning cereal, he takes a bowl off the top of the stack. When he is finished washing the bowl, he dries it and places it back on the top of the stack. In other words, the last bowl put on the stack is the first one taken off. The bottom-most bowl is the one that Sam has not used in the longest time. The top-most bowl is the one that has been most recently placed in the cupboard.

This organization and utilization of bowls is a stack, but it is incidental; Sam doesn't care which bowl he uses when he has his morning cereal. This analogy is also not quite perfect: there is nothing to stop him from grabbing four bowls at once. A true stack only allows access to the top-most element. To be a true stack, he would need to grab the four bowls one at a time.

Another example of the stack (and perhaps more familiar to us and relevant in our profession) is multi-level undo. Most programs enable the user to take back the last action. Since the late 1980's, programs would store all the actions performed in a given session in a stack so they can be undone one step at a time. The way undo records are stored in a computer is a stack. This illustrates the defining characteristics of a stack:

- **Container**: The stack is designed to store a collection of data.
- **Top Access**: Only the top element of the stack can be accessed.
- **Order Preserved**: The arrangement of elements is determined by insertion order.
- **Expandable**: The capacity of the stack can be increased to meet the needs of the client.
- **Restricted Insertion**: Adding elements to any location of the stack but the end is prohibited.

The following picture is the undo stack of Microsoft Word. Here the most recently added element (Bold) is presented at the top.



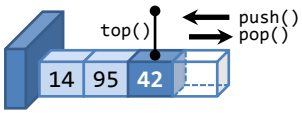
From this picture, we can see that the most recently performed action was "Bold", preceded by "Paragraph Alignment" and so on. If the user wished to undo "Typing" then first "Bold" and then "Paragraph Alignment" would need to be undone. Then and only could "Typing" be undone.

Definition

A stack is called a Last-In-First-Out (LIFO) data structure. It can equivalently be called First-In-Last-Out (FILO). A stack is a list of elements where the user has access only to one end called the top. The operations performed on a stack are the following:

Name	Example	Description
Create	`stack <int> s;`	Create a new stack.
Assignment	`s1 = s2;`	Make a copy of a stack.
Empty	`f = s.empty();`	Returns `true` if the stack has no elements, `false` otherwise.
Size	`n = s.size();`	Returns the number of elements currently in a stack.
Capacity	`c = s.capacity();`	Returns the number of elements the stack will hold before reallocating.
Clear	`s.clear();`	Remove all elements from the stack.
Push	`s.push(e);`	Adds an element to the top of the stack, thereby increasing size by one.
Pop	`s.pop();`	Remove the top-most element in the stack, thereby reducing the size by one.
Top	`e = s.top();`	Returns the top-most element in the stack.

Aside from the standard container interfaces, the stack has a `push()`, `pop()` and `top()` method. The stack can be represented with the following figure:



Notice how only one element of the stack can be accessed at a time: the top element. This is represented in the figure with only one vertical line reaching an element, and that element being depicted in dark blue. The other two elements (with the value `14` and `95`) are not accessible.

Elements can be added to the stack with the `push()` method, indicated with an arrow pointing into the stack. Elements can be removed from the stack with the `pop()` method, indicated with an arrow pointing away from the stack.

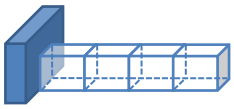
Finally, notice how an empy slot exists in the array (depicted with the transparent cube). As with the `vector`, the `stack` may have a capacity greater than the size.

Create

A stack is created using the default constructor (with the capacity set to zero), the non-default constructor (with the capacity specified), or the copy constructor.

```
stack <T> :: stack();
stack <T> :: stack(int max)               throw (const char *);
stack <T> :: stack(const stack <T> & rhs) throw (const char *);
```

In all cases, the data-type of the elements is specified.

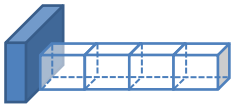
	Code	Description	State
1	`stack <char> s1;`	The default constructor does not allocate a buffer so the size and capacity are zero.	
2	`stack <int> s2(4);`	The non-default constructor allocates a buffer. However, the size remains zero because the extra slots in the array are not used.	
3	`stack <int> s3(s2);`	The copy constructor makes a copy of the stack. Notice how the buffer size does not need to be copied over.	

Capacity

The capacity is the maximum number of elements the stack can hold without reallocating, not the number of elements currently stored in the stack.

```
int stack <T> :: capacity() const;
```

It is up to the author of the stack class to manage the capacity to make best use of system resources. For this reason, the client should not depend on the value of this method. That being said, an integer passed as the non-default constructor can set the capacity to a given size.

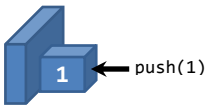
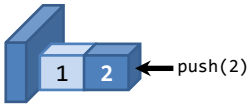
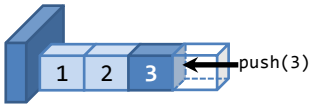
	Code	Description	State
1	`stack <int> s1;` `assert(s1.capacity() == 0);`	Capacity is initially zero	
2	`stack <int> s2(4);` `assert(s2.capacity() == 4);`	Capacity equal 4 due to the non-default constructor	

Push

Push involves adding an element to the top of the stack.

```
void stack <T> :: push(const T & t) throw (const char *);
```

In the case where the stack is full, a reallocation will be attempted. As with the vector, this involves allocating a new buffer and copying the current stack contents over. The situation may arise where no memory is available for a new buffer. In that case, an exception will be thrown.

	Code	Description	State
1	`stack <int> s;`	Initially the stack is empty.	
2	`s.push(1);`	One element is added to the top of the stack, thereby increasing the capacity and size by one.	
3	`s.push(2);`	With the second `push()`, the size is increased again. We can only access `2` because it is on top.	
4	`s.push(3);`	Another `push()` will cause the stack to increase size again.	

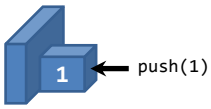
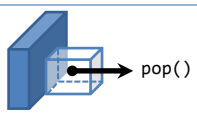
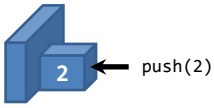
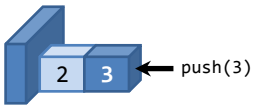
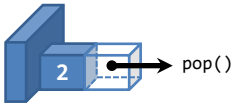
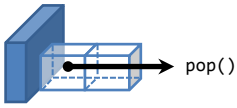
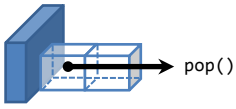
Note that the left side is the bottom of the stack (with the number `1`) and the right side is the top (with `3`) at the conclusion of step 4.

Pop

Pop involves removing the top-most element from the stack.

```
void stack <T> :: pop();
```

If the stack is already empty, then nothing happens.

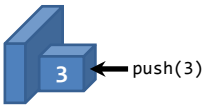
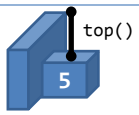
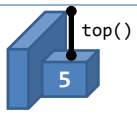
	Code	Description	State
1	`stack <int> s;`	Initially empty with the default constructor	
2	`s.push(1);`	One item added to the top.	
3	`s.pop();`	Remove 1 from the top.	
4	`s.push(2);`	Add 2 to the top.	
5	`s.push(3);`	Add 3 to the top of 2.	
6	`s.pop();`	Remove 3 from the top.	
7	`s.pop();`	Remove 2 from the top. Now the stack is empty.	
8	`s.pop();`	Attempting to pop from an empty stack does nothing. The stack does not change and an exception is not thrown.	

Top

Top accesses the last element added to the stack. There are two things to take into account. The first is that the stack might be empty. Attempts to access `stack::top()` on an empty stack will result in an exception being thrown. The second consideration is that the client needs to be able to modify the top element of the stack through the `stack::top()` statement. In other words, `stack::top()` needs to function both as a getter and as a setter.

```
      T & stack <T> :: top()       throw (const char *);
const T & stack <T> :: top() const throw (const char *);
```

Observe how there is both a read-write version of `top()`, as well as a constant read-only version.

	Code	Description	State
1	`stack <int> s;`	Initially empty.	
2	`s.push(3);`	Add one element to the top.	
3	`s.top() = 5;`	We can change the top element with the `top()` method. Thus `top()` functions as a setter.	
4	`cout << "Top: "` `  << s.top() << endl;`	Notice how `top()` is both a getter and a setter.	

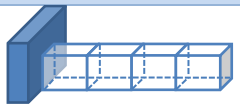
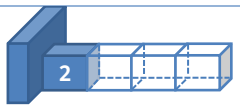
Note in step 3 how `top()` can be used to set the value in the stack. This means the output of step 4 will be `5`.

Assignment

The assignment operator makes a copy of the stack.

```
stack <T> & stack <T> :: operator = (const stack <T> & rhs) throw (const char *);
```

This does not mean that the copy needs to be identical to the original. Instead, it means that it has to behave the same from the client's perspective.

	Code	Description	State
1	`stack <int> s1(4);`	Here the stack has four empty slots.	
2	`s1.push(2);`	Add one element to the top of `s1`.	
3	`stack <int> s2;`	Create an empty stack.	
4	`s2 = s1;`	As we assign `s1` onto `s2`, the elements get copied but not the capacity.	

Notice how `a` is created in step 1 with the non-default constructor setting the capacity to 4. However, when it is copied to `b` in step 4, the capacity is not copied. Only the elements are. In other words, the assignment operator is obligated to copy the elements of the stack whereas it is up to the implementation to decide how to make best use of the capacity.

Standard Container Interfaces

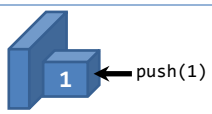
The size is simply the number of elements currently in the stack.

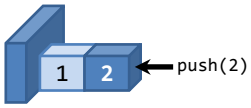
```
int  stack <T> :: size()  const;
bool stack <T> :: empty() const;
void stack <T> :: clear();
```

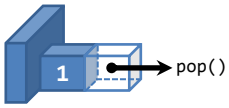
Size does not change the state of the stack so it can be a constant method.

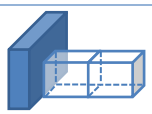
	Code	Description	State

1	`stack <int> s;`
	`assert(s.size() == 0);`
	`assert(s.empty());`
		The default constructor must yield the size to be zero.	

2	`s.push(1);`
	`assert(s.size() == 1);`
	`assert(!s.empty());`
		If we add one element, the size must be one.	

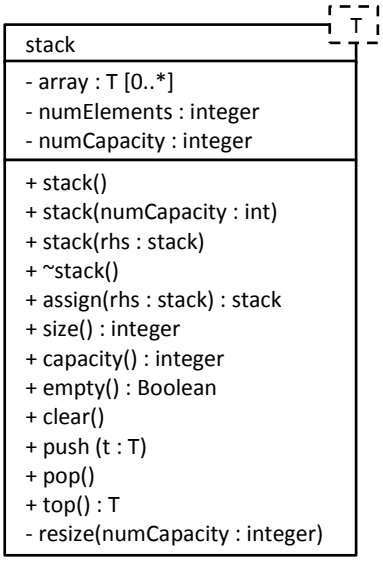
3	`s.push(2);`
	`assert(s.size() == 2);`
	`assert(!s.empty());`
		With the second `push()` the size must be 2.	

4	`s.pop();`
	`assert(s.size() == 1);`
	`assert(!s.empty());`
		By removing 2 from the stack, the size must be 1 again.	

5	`s.clear();`
	`assert(s.size() == 0);`
	`assert(s.empty());`
		By calling the `clear()` method, we remove any and all remaining elements from the stack..	

Implementation

The client (or user) of the stack class should not have to be aware of the implementation details. This means that we need to find an implementation that best meets the correctness (honors the stack contract), takes the smallest amount of memory (size), and uses the smallest amount of CPU resources (performance).



Before we begin, we have a decision to make: does the top-most element reside at the beginning of the array or the end? If we choose the beginning, then every `push()` will require shifting every element over by one slot. This will be O(n) in every single case. If we choose the end, then the rest of the array can remain unchanged when elements are added. Clearly the second choice is a better solution.

Standard Container Interfaces

This is a trivial function that returns the number of elements in the stack. Since we keep track of this value as a member variable, implementation is trivial.

```
stack.size()
  RETURN numElements
```

It is often desirable for the client to know if there are any elements in the stack. This can be done by checking the value of `size()`.

```
stack.empty()
  IF size() = 0
    RETURN true
  ELSE
    RETURN false
```

It is also very easy to define `clear()`:

```
stack.clear()
  numElements ← 0
```

Assignment

All data-structures need to provide a mechanism to make a copy of itself. The resulting copy needs to be identical as far as the client can tell. In other words, all the APIs defined above should behave exactly the same between the two copies. This does not mean that the two copies are identical! In other words, there can be more than one implementation of an API that behaves the same to the user.

To illustrate this point, our implementation of the stack will have three member variables: a dynamically allocated array of elements (`array`), the number of elements currently in the stack (`numElements`), and the number of elements that can be stored without allocating more data (`numCapacity`). Since the client has no guarentee how the capacity behaves, we are not obligated to preserve it across a copy. Thus, we will grow the stack only if it is not already big enough to contain the new elements, and that size will be big enough only to contain the new elements. We will never shrink the size of the stack.

```
stack.assignment(lhs, rhs)
  lhs.clear()
  IF lhs.capacity() < rhs.size()
    lhs.resize(rhs.size())
  FOR i ← 0 … rhs.size()-1
    lhs.array[i] ← rhs.array[i]
  lhs.numElements ← rhs.numElements
```

There are a few things to notice here. First, we only resize `array` if the current capacity is insufficient. We are always trying to minimize the amount of work that needs to be done to honor the stack contract.

Second, notice how the cost of calling the assignment operator is related to the number of elements in the RHS. Thus the cost is O(n) where `n == rhs.size()`.

Top

We can access the top element on the stack by indexing directly to it using the `numElements` variable.

```
stack.top()
  IF not empty()
    RETURN array[size() – 1]
  ELSE
    ERROR
```

Note that this method needs to function both as a getter and as a setter. The getter should be a `const` method because it is guaranteed to not change the member variables. However, the setter necessarily will change the member variables. This means that, in C++, we need to have two overloaded versions of the method. The first will be a `const` method returning a copy of the top element and the second a non-`const` method returning the value by-reference. See the Definition section of the chapter for an example of how this works.

The performance of the code in this function and `stack.empty()` are unrelated to how many elements in the stack. Thus the cost is O(1).

Push

Push involves adding an element to the top of the stack. Notice how it works exactly like `vector::push_back()`.

```
stack.push(e)
  array[numElements++] ← e
```

What happens when the stack is full? We could do nothing (violating the correctness requirement), we could throw an exception (probably not what the client expects), or we could grow the stack. The client expects the stack class to "just work" so we will need to grow. Our first attempt will be to call a `resize()` function which will grow the stack by one element.

```
stack.push(e)
  IF size() = capacity()
    resize(capacity() + 1)
  array[numElements++] ← e
```

This appears efficient from a space point of view. We use the minimum amount of space and the cost of resizing is O(n) because the entire array needs to be copied to a newly allocated buffer. However, if one were to add n elements to the stack, then the performance of that is $O(n^2)$. This is because we have to perform n operations costing O(n). Thus the total cost is n x O(n) == $O(n^2)$. A better approach is to double the size of the stack with each reallocation:

```
stack.push(e)
  IF size() = capacity()
    resize(capacity() * 2)
  array[numElements++] ← e
```

Here we are wasting some space, but we have to allocate much less often. If one were to add 1,000 elements to a stack, it would be necessary to reallocate not 1,000 times but rather 10 times (at 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512). In other words, the reallocation would have to occur $\log_2 n$ times because $2^{10} = 1024$. This means the cost of adding 1,000 elements to the stack is O(n log n) which is much better than $O(n^2)$.

Pop

The pop operation involves removing the topmost element from the stack. We could erase this element. In some high security situations, this may even be desirable. However, it does raise the question: replace the value with what? In the vast majority of situations, we just need to offer a guarantee that the removed element is no longer accessible.

The second decision is whether to shrink the array. This makes sense from the standpoint of minimizing the memory footprint. However, it is not optimal from a performance standpoint. Consider the following pseudocode:

```
stack.pop()
  resize(--numElements)
```

The cost of this algorithm is O(n) because n-1 elements will be copied from the old buffer and into the new. This means if one were to empty the stack through a series of pops, the collective cost will be $O(n^2)$ because we would be calling `stack.pop()` a total of n times.

A more efficient pop algorithm would simply abandon the last element on the list.

```
stack.pop()
  IF not empty()
    --numElements
```

This algorithm is O(1) and the total cost of emptying a stack through a series of pops becomes O(n). By not calling `resize()` we can realize a substantial performance increase.

One final consideration: what do we do if the client attempts to pop from an empty stack? In this case, we simply do nothing and return out of the function. Your code will need to handle this case.

Application

The stack is a surprisingly versatile and useful data structure. To present a taste of the types of problems that can be solved with a stack, four examples will be described: reversing a list, conversion of infix to postfix, and assembly language generation.

Reversing a List

Consider a list of elements in an array. For the sake of discussion, this could be a list of doubles but any type of elements would work here. How does one put the first element at the end and the last element at the beginning? Clearly there are many methods that would work to accomplish this, but we will use a stack to solve this problem.

We will start by iterating through the list and pushing the elements in a stack one at a time. This will mean that the top element in the stack will be the last element in the list. From here, one can put the elements back in the list through a series of pops. The C++ code for this algorithm is:

```
void reverse(double numbers[], int size)
{
   stack <double> s;

   // push loop
   for (int i = 0; i < size; i++)
      s.push(numbers[i]);

   // pop loop
   for (int i = 0; i < size; i++)
   {
      numbers[i] = s.top();
      s.pop();
   }
}
```

The first loop (labeled the "push loop" above) will make a copy of the elements in the numbers array into the stack. The first element will go on the top of the empty stack and the next will go on the top after that. This will continue until the last element in the array will be on the top. At the end of the first loop, all the elements in the array will be in the stack in their original order.

The second loop (labeled the "pop loop" above) will then remove elements from the stack and put them back in the array. However, since the index for the array is traversing the list forwards and the stack is yielding elements backwards, the end result is to reverse the elements of the array.

An example of reversing an array of numbers is the following. Consider an array of numbers with the elements `numbers == {14, 95, 42, 37}`.

	Array	Description	Stack
1	14 95 42 37	The stack is not allocated.	
2	14 95 42 37	Push one element in our stack. Note that the value 14 is copied into the stack array; it still remains in the numbers array.	14 push(14)
3	14 95 42 37	After the first push, the stack is full. Every allocation after the first will double the capacity of the stack, so the capacity is now 2.	14 95 push(95)
4	14 95 42 37	After the second allocation, the stack is once again full. When we double the capacity, it goes from 2 to 4 even though we only need three slots. This means the last slot is empty.	14 95 42 push(42)
5	14 95 42 37	This is our first push that does not require an allocation. However, after this push, the stack is once again full. Notice how the data in our stack is the same as our array of numbers.	14 95 42 37 push(37)
6	37 95 42 37	With the second loop, two things happen. First, the top of the stack is copied into the first element of the array. Second, we abandon the last element in the stack array through pop.	14 95 42 pop()
7	37 42 42 37	Since the last pop caused 42 to be at the top of our stack, we place it in the second slot of the numbers array. Also, we will pop the top element in the stack array.	14 95 pop()
8	37 42 95 37	With the end of the 3rd Pop, there is only one element left in our stack.	14 pop()
9	37 42 95 14	With the end of this loop, we are finished with our function. Notice how the stack is empty. Notice also that the numbers array has now been completely reversed.	pop()

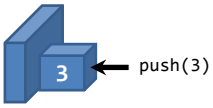
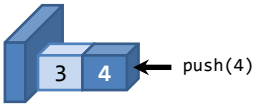
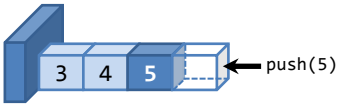
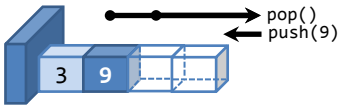
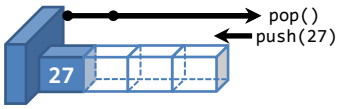
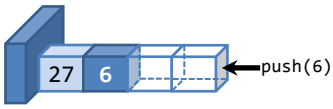
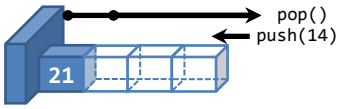
Interpreting Postfix Notation

Recall from CS 165 that there are three main ways to represent a function: infix, prefix, and postfix notation. Infix notation has the operator between the operands such as 4 + 5. Prefix notation has the operator before the operands such as `add(4, 5)` and `+ 4 5`. Postfix notation is less common and has the operator after the operands such as `4 5 +`. This second example will demonstrate how to evaluate postfix notation.

Postfix notation is also called "stack-based evaluation." The algorithm is as follows: numbers are pushed on the stack. When an operator is encountered, the top two numbers are popped off and the result of the operator is pushed back on. When the postfix array has been fully consumed, the remaining number is the answer:

```
evaluate (postfix, size)
  FOR i ← 0 .. size – 1
    IF postfix[i].isNumber
      stack.push(postfix[i])
    ELSE
      rhs ← stack.top()
      stack.pop()
      lhs ← stack.top()
      stack.pop()
      answer ← evaluateOperator (postfix[i], rhs, lhs)
      stack.push(answer)
  RETURN stack.top()
```

To see how this works, consider the following: `postfix == {3 4 5 + * 6 -}`.

	Input	Description	Evaluation	State
1		When the program begins, we have no elements in the stack.		
2	3	Since the first token (symbol or number to be evaluated) is a number, it is pushed directly onto the stack.		
3	4	The next element in our list of tokens is also a number so it gets added directly to the stack.		
4	5	Again, another number is added to the stack. Thus far, we have performed no evaluation		
5	+	With the plus operator, we pop the last two elements off the stack, perform the operation (addition), and push the result back on the stack	4 + 5	
6	*	Another operator, this one multiplication. Two elements are popped off the stack and the product is pushed back on	3 * 9	
7	6	Another number that is pushed directly on the stack		
8	-	The subtraction operator is encountered so we pop the two elements off the stack and put the difference back in. The result is the evaluated expression.	27 – 6	

Infix to Postfix

The next example will discuss how to convert an infix expression into a postfix expression. The infix expression `4 * 3 + 2` would become `4 3 * 2 +` where the operators are after the operands.

The first thing to note about this converson is that it is a nesting problem. Nesting problems naturally lend themselves to stacks because stacks facilitate working on the innermost problem while leaving the outermost parts of the problem stored away for later. If, for example, parentheses exist in the expression, clearly the content inside the expression will need to be evaluated first.

First attempt: All operators are the same

Let's go through a few simple examples to see how they will illuminate the problem. Consider the infix string "4 + 5" or `infix == {4 + 5}`. This clearly converts to `postfix == {4 5 +}`. Our first attempt for this algorithm is to go through the infix list one element at a time. As a number is encountered, it is put directly on the postfix list. The operator is saved for the end in the stack.

```
infixToPostfix(infix, size)
  FOR iInfix ← 0 .. size – 1
    IF infix[iInfix] is number
      postfix[iPostfix++] ← infix[iInfix]
    ELSE
      stack.push(infix[iInfix])
  WHILE not stack.empty()
    postfix[iPostfix++] ← stack.top()
    stack.pop()
```

A slightly more complex example occurs when there is more than one operator in the expression. Following the simple algorithm in the previous paragraph, `infix == {4 + 5 * 6}` becomes the `postfix == {4 5 6 * +}`. That is actually the same as `infix == { 4 + ( 5 * 6 ) }` because the "5 * 6" is evaluated first. Notice that we want to use a stack to hold the operators that we are saving because they are put in the postfix output array in the opposite order that they appear in the infix array.

This first algorithm does not take into account the order of operations. In other words, multiplication occurs before addition. If we follow the above algorithm, then the expression `infix == {4 * 5 + 6}` would be evaluated as `postfix == {4 5 6 + *}` which would be incorrect. It should be `postfix == {4 5 * 6 +}`.

	Input	Description	Postfix Array	Stack
1		When the program begins, we have no elements in the stack and the output array is empty		
2	4	The first element in the infix list is the number 4. Since it is not an operator, it gets put directly in the postfix array.	4	
3	+	The next element is the plus sign. Since it is an operator, we do not directly put it in the postfix array but rather in the stack.	4	+ push(+)
4	5	The next element is 5 which is a number. Thus it goes directly into the postfix array.	4 5	+
5	*	We have encountered another operator, this one the multiplication symbol (*). It goes directly on the stack.	4 5	+ * push(*)
6	6	The final number (6) is encountered which is put directly into the postfix array.	4 5 6	+ *
7		We are out of elements in the infix list. Now we need to pop off the operators from the stack and place them into the postfix array.	4 5 6 *	+ pop()
8		The final operator is popped off of the stack, thereby completing the problem.	4 5 6 * +	pop()

Second attempt: Order of operations

Infix expressions need to have an understanding of order of operations. In other words, if one operator appears before another in the order of operations, then it should be evaluated first. This brings us to our second iteration for an algorithm. Loop through all elements in the infix list. If a number is encountered, it is put directly on the postfix list. If an operator is encountered, it is saved for later only if it is before than the top most element on the stack of operators.

```
infixToPostfix(infix)
  FOR iInfix ← 0 .. infix.size() – 1
    IF infix[iInfix] is number
      postfix[iPostfix++] ← infix[iInfix]
    ELSE
      WHILE !stack.empty() &&
              infix[iInfix] ≤ stack.top()      ← here ≤ understands the order of operations
        postfix[iPostfix++] ← stack.top()
        stack.pop;
      stack.push(infix[iInfix])
  WHILE not stack.empty()
    postfix[iPostfix++] ← stack.top()
    stack.pop()
```

Let's try this algorithm with `infix == {4 * 5 + 6}`.

	Input	Description	Postfix Array	State
1		When the program begins, we have no elements in the stack and the output array is empty.		
2	4	The first element in the infix list is the number 4. Since it is not an operator, it gets put directly in the postfix array.	4	
3	*	The next element is the multiplication sign. Since it is an operator and there is no other operator on the stack, we can safely push it onto the stack.	4	* push(*)
4	5	The next element is 5 which is a number. Thus it goes directly into the postfix array.	4 5	*
5	+	Do we push the Plus operator on the stack or put it directly in the postfix array? Since Multiplication comes before Addition in the order of operations, we pop it off the stack before pushing the addition operator in its place.	4 5 *	+ pop() push(+)
6	6	The final number (6) is encountered which is put directly into the postfix array.	4 5 * 6	+
7		We are out of elements in the infix list. Now we need to pop off the remaining operators from the stack and put them in the postfix array	4 5 * 6 +	pop()

Final attempt: Parentheses

We are almost done. The last thing we need to do is to handle parentheses. These are special because they override all other operators and require two tokens (both the open '(' parenthesis and the close ')' as well need to be handled). When we encounter a close parenthesis, we need to pop all the operators off the stack up to and including the open parenthesis. In other words, the open means "stop here" and the close means "keep popping until you get to an open and them pop the open also." Our final algorithm is:

```
infixToPostfix(infix)
  FOR iInfix ← 0 .. infix.size() – 1
    SWITCH infix[iInfix]
      CASE number or variable
         postfix[iPostfix++] ← infix[iInfix]
      CASE (
         stack.push(infix[iInfix])
      CASE )
         WHILE stack.top() ≠ (
           postfix[iPostfix++] ← stack.top()
           stack.pop()
         stack.pop()
      DEFAULT
         WHILE !stack.empty() &&
                 infix[iInfix] ≤ stack.top()   ← here ≤ understands the order of operations
            postfix[iPostfix++] ← stack.top()
            stack.pop();
         stack.push(infix[iInfix])
  WHILE not stack.empty()
    postfix[iPostfix++] ← stack.top()
    stack.pop()
```

We will test this with `infix == {4 * (5 + 6) ^ 2}`.

Consider the following input: `infix == {4 * (5 + 6) ^ 2}`.

	Input	Description	Postfix Array	Stack
1		When the program begins, we have no elements in the stack and the output array is empty.		
2	4	The first element in the infix list is the number 4. Since it is not an operator, it gets put directly in the postfix array.	4	
3	*	The next element is the multiplication sign (*). Since it is an operator and there are no operators on the stack, we can safely push it onto the stack.	4	* push(*)
4	(	We always push the open parenthesis directly onto the stack, regardless of what else is there.	4	* (push(()
5	5	The number 5 goes directly in the postfix array.	4	* (
6	+	The plus operator is low on the order of operations but it does not appear before the open parenthesis (only the close parenthesis can do that).	4 5	* (+ push(+)
7	6	The number 6 is encountered which is put directly into the postfix array.	4 5 6	* (+
8	)	With the close parentheses, we pop off all the elements in the stack up to and including the close parentheses. Note that there are no parentheses in postfix notation.	4 5 6 +	* pop()
9	^	The exponent operation is extremely high in the order of operations and it is before the multiplication symbol. Thus ^ goes onto the stack.	4 5 6 +	* ^ push(^)
10	2	The final number (2) is encountered and it is put directly in the postfix array.	4 5 6 + 2	* ^
11		The final step is to empty the stack onto the output array. Now we are done.	4 5 6 + 2 ^ *	pop()

Postfix to Assembly

The final example is to demonstrate how to convert postfix notation to assembly language. As you may recall, assembly language is a step in the compilation process. It is a language very close to machine language, the language that CPUs consume on a computer. An example of assembly language is:

```
LOD 4
ADD 12
SAV A
```

There are a few things to know about assembly language:

- The CPU always has a value in a special memory location called the register. It can be thought of as the temporary storage location that is used by all computation.
- Usually addition works with three values: the left-hand-side, the right-hand-side, and the answer. With assembly, there are only two: the left-hand-side is the register, the right-hand-side is the value to the right of the operation, and the answer goes back into the register. Thus `ADD 12` is the same thing as `register = register + 12`.
- Load puts either a literal value (a number) or the value into the register.
- Save puts the value of the register into a variable.

Our assembly language will have the following commands:

Name	Description
LOD	Load a value from some memory location into the register
SET	Set the register to some literal value
SAV	Save the value in a register to some memory location
ADD	Add a value to the register
SUB	Subtract a value from what is currently in the register
MUL	Multiply the current value in the register by some value
DIV	Integer division for some value in the register
MOD	Find the remainder of the division of the number in the register with a given value
EXP	Modify the value in the register by raising it to a given power

To add the number `3` and `4` and put the answer in a variable called `A`, the following will be needed:

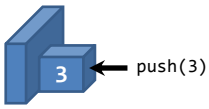
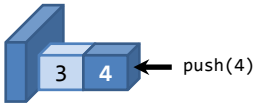
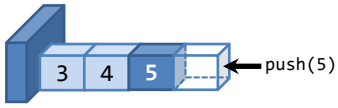
```
SET 3
ADD 4
SAV A
```

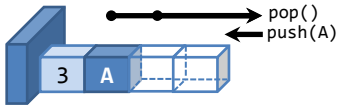
We will write a program to convert infix notation to a collection of assembly commands. The first thing to realize is that this is essentially the same problem as the postfix evaluation example we worked on previously. In other words, we will push the numbers and variables onto a stack and, with every operator, we will pop off the top two and evaluate the operator. The main difference is how we will evaluate the operator. Instead of just "doing the math" as we did previously, we will output three assembly statements exactly like the previous example:

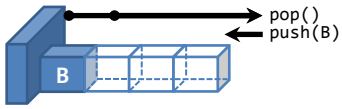
1. Load or set a value to the register
2. Perform the operation
3. Stores the result in a variable

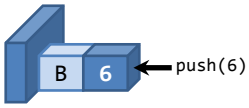
```
postfixToAssembly(postfix, size)
  FOR i = 0 .. size – 1
    IF postfix[i] is a number or variable
      stack.push(postfix[i])
    ELSE
      rhs ← stack.top()
      stack.pop()
      lhs ← stack.top()
      stack.pop()
      variable ← nextVariable++      ← Creates new variable names for later use
      IF lhs.isVariable
        assembly += "LOD " + lhs
      ELSE
        assembly += "SET " + lhs
      assembly += postfix[i] + rhs
      assembly += "SAV " + variable

      stack.push(variable)
  RETURN stack.top()
```

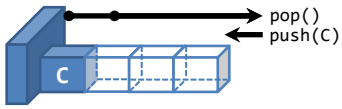
For example, consider `postfix == {3 4 5 + * 6 -}`. With three operators (`+` `*` and `-`), we expect to find three sets of assembly commands to be generated.

	Input	Next	Description	Stack	Output
1		A	When the program begins, we have no elements in the stack and the output is empty.		
2	3	A	The first element in the infix list is the number 3. Since it is not an operator, it gets put directly in the stack.		
3	4	A	Another number which goes directly on the stack.		
4	5	A	Yet another number, also going directly on the stack.		

5	+	B	With an operator, the last two elements are popped off the stack (`4` and `5`) and the result of the evaluation is put back on (`A`).		`SET 4`
					`ADD 5`
					`SAV A`

6	*	C	Another operator, causing another two pops (`3` and `A`) and another evaluation to be put on the stack (`B`)		`SET 3`
					`MUL A`
					`SAV B`

7	6	C	The number 6 is pushed to the end of the stack.		

8	-	D	Another operator, causing another two pops (`B` and `6`) and another evaluation to be put on the stack (`C`). Notice how `lhs` is a variable rather than a number so we need to use `LOD` rather than `SET`.		`LOD B`
					`SUB 6`
					`SAV C`

The end is reached when the postfix list is empty. The result of the expression is represented by the variable at the top of the stack (`C`).

Examples

Example 02.1

What is the state of the stack at the end of execution?

```
{
   stack <char> s;     // #1
   s.push('A');        // #2
   s.push('B');        // #3
   s.push('C');        // #4
}
```

Answer:

#1		Initially empty
#2	A	Capacity goes to 1 and the first slot is filled
#3	A B	Capacity doubles and the second slot is filled
#4	A B C	Capacity doubles again to 4. The third slot is filled but the fourth remains uninitialized

Example 02.2

What is the state of the stack at the end of execution?

```
{
   stack <int> s;      // #1
   s.push(100);        // #2
   s.push(200);        // #3
   s.pop();            // #4
   s.top() = 300;      // #5
}
```

Answer:

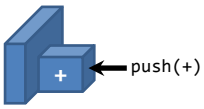
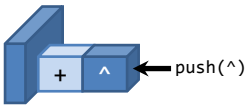
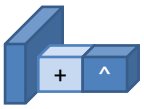
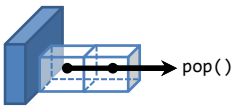
The values of `numElements` and `numCapacity` are omitted for the sake of brevity.

#1		Initially empty
#2	100	Capacity goes to 1 and the first slot is filled with 100
#3	100 200	Capacity doubles and the second slot is filled with 200
#4	100	The last element on the stack is abandoned
#5	300	The last element is accessed with `top()` and replaced with 300

Example 02.3

Convert the following infix notation into postfix: `44 + 2 ^ 3`

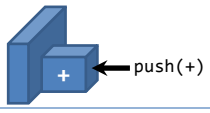
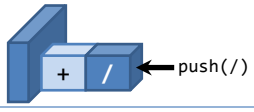
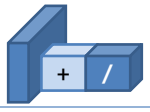
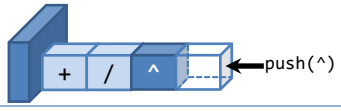
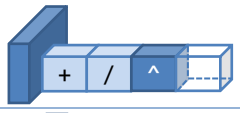
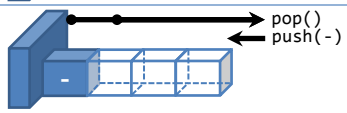
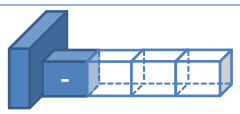
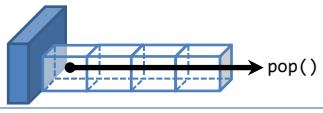
Answer:

Input	Stack	Output	Discussion
32		44	Numbers put directly in output
+		44	Operators are pushed on empty stacks
2		44 2	Numbers put directly in output
^		44 2	Exponent higher precedence than plus so pushed
3		44 2 3	Numbers put directly in output
		44 2 3 ^ +	At the end, pop off everything from the stack

Example 02.4

Convert the following infix notation into postfix: `1 + 2 / 3 ^ 4 - 5`

Answer:

Input	Stack	Output	Discussion
1		1	Numbers put directly in output
+		1	Operators are pushed on empty stacks
2		1 2	Numbers put directly in output
/		1 2	Divide higher precedence than plus so pushed
3		1 2 3	Numbers put directly in output
^		1 2 3	Exponent higher precedence than divide
4		1 2 3 4	Numbers are put in the output
-		1 2 3 4 ^ / +	First minus is compared against exponent and it is less. This means exponent is popped off the stack and put in the output. The same is done with divide. Finally we reach the plus. Since we can't have two operators on the stack with the same level of the order of operations, the addition is popped off before the subtraction is pushed on.
5		1 2 3 4 ^ / + 5	Another number directly in output stream
		1 2 3 4 ^ / + 5 -	At the end, pop off everything from the stack

Example 02.5

Convert the following infix notation into postfix: `4 / (3 ^ (1 + 2) * 5)`.

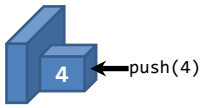
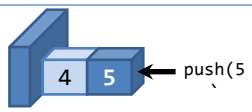
Answer:

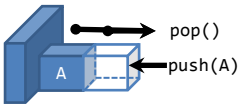
Input	Stack	Output	Discussion
4		4	Numbers are put directly in the output column
/	/ push(/)	4	Operators are pushed directly onto the empty stack
(	/ (push(()	4	Open parenthesis are pushed
3	/ (	4 3	Again, numbers put directly in the output column
^	/ (^ push(^)	4 3	Compare ^ with (so we put it on the stack
(	/ (^ (push(()	4 3	Another open parenthesis is pushed on the stack
1	/ (^ (	4 3 1	Numbers are put in the output
+	push(+) / (^ (+	4 3 1	Compare + with (so put it on the stack
2	/ (^ (+	4 3 1 2	Another number directly in output
)	pop() / (^	4 3 1 2 +	With close parenthesis, pop everything up to and including the (
*	pop() push(*) / (*	4 3 1 2 + ^	Because ^ is higher than * in the order of operations, we pop the ^ and push the *
5	/ (*	4 3 1 2 + ^ 5	Numbers go directly in the output stream
)	pop() /	4 3 1 2 + ^ 5 *	With), pop off all operators up to the (
	pop()	4 3 1 2 + ^ 5 * /	At the end, pop off everything from the stack

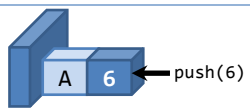
Example 02.6

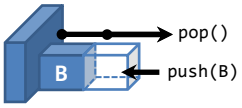
Convert the following postfix notation into assembly: `4 5 + 6 *`.

Answer:

Input	Stack	Output	Discussion
4			Numbers are put directly on the stack
5			The second number is put on the stack

+		`SET 4`
		`ADD 5`
		`SAV A`
			The top two values are popped off the stack, three assembly statements are generated, and the stored value is put back on the stack

6			Numbers are put directly on the stack

*		`LOD A`
		`MUL 6`
		`SAV B`
			Again, the top two values are popped off the stack and the corresponding assembly statements are generated. The resulting value is put back on the stack.

Exercises

Exercise 02.1

From memory, recite and define the operations associated with the stack ADT. Include both the read and the read/write versions of `top()`.

Answers:

_____

_____

_____

_____

Please see page 46 for a hint.

Exercise 02.2

Consider the expression "add one and two." Write this expression in Infix, Prefix, and Postfix

Answers:

Infix: _____

Prefix: _____

Postfix: _____

Please see page 58 for a hint.

Exercise 02.3

Write the following expressions in postfix:

Answers:

`1 + 2 * 3` _____

`1 / (2 - 3)` _____

`a * b + c - d` _____

`(a + b) / c + d` _____

`(a + b) / (c + d)` _____

`a + b / c + d` _____

`(a - b) * (c - (d + e))` _____

`(v * t) + a * t ^ 2 / 2` _____

Please see page 59 for a hint.

Exercise 02.4

Evaluate the following postfix expressions:

Answers:

```
4  5  *  44  2  /  +
```

_____

```
32 16 8 4 2 1 ^ * / + -
```

_____

Please see page 59 for a hint.

Exercise 02.5

Using the assembly described in the example, generate assembly for the following postfix expressions:

Answers:

`1 2 +`	`a b c * *`	`W X + Y Z * /`

Please see page 59 for a hint.

Exercise 02.5

Write out the state of stack's member variables after each line or group of lines are executed:

```
{
   stack <int> s;

   s.push(55);

   s.push(3);

   s.push(90);
}
```

numCapacity	numElements	array

Please see page 54 for a hint.

Exercise 02.6

Write out the state of stack's member variables after each line or group of lines are executed:

```
{
   stack <int> s;
   s.push(49);

   s.pop();
   s.push(85);

   s.pop();
   s.pop();
   s.push(26);

   s.pop();
}
```

numCapacity	numElements	array

Please see page 55 for a hint.

Exercise 02.7

Write out the state of stack's member variables after each line or group of lines are executed:

```
{
   stack <int> s;
   s.push(49);
   s.top() = 38;

   s.push(s.top());

   s.push(89);
   s.push(64);
   s.push(79);

   s.pop();
   s.pop();
   s.push(3);

   s.pop();
   s.pop();

   int x = s.top();
   s.pop();
   s.push(50);
```

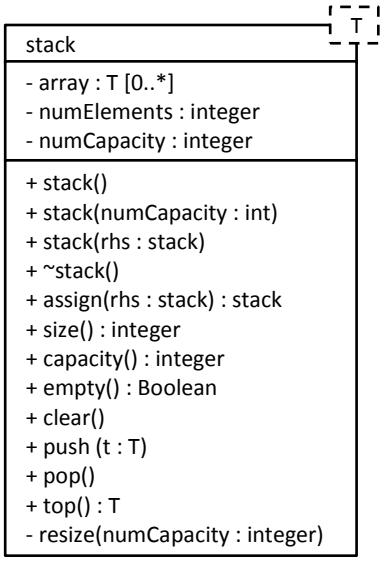
numCapacity	numElements	array

Please see page 55 for a hint.

Problems

Problem 02.1

Implement your own version of the `std::stack` data structure. Call your implementation `custom::stack`. In other words, place your class in the custom namespace and implement all of the interfaces described in this chapter. The UML class diagram of the stack class is the following:



Please see page 53 for a hint.

Problem 02.2

Using the STL stack or using your own implementation of `stack` from Problem 02.1, write a program to convert an infix expression into a postfix expression. Note that you will need to handle the following operators:

- () The parentheses are the highest level of the order of operations.
- ^ The exponent operator comes immediately after the parentheses. Note that the algorithm in the textbook cannot handle the exponent operator properly. You will need to handle this case.
- * / % Multiplication, division, and modulus are after the exponent operator. They are at the same level of the order of operations.
- + - Addition and subtraction are after multiplication, division, and modulus.

Please see page 62 for a hint.

Problem 02.3

Using the STL stack or your own implementation of `stack` from Problem 02.1, write a program to convert a postfix expression into assembly statements similar to the "Postfix to Assembly" application.

Please see page 64 for a hint.

Sue needs to go to the Department of Motor Vehicles (DMV) to register her car. When she gets there, about a hundred people are already waiting in line.

Sue would prefer to just go to the head of the line so she could complete her transaction and go home. However, there are certain rules; one can only enter the line at the back without incurring the wrath of those already there! She would also prefer if the employees at the DMV would call on her first rather than those at the head of the line. These employees have been instructed to do no such thing. Resigned to her fate, Sue settles into the back of the line and begins reading a book.

This example illustrates the defining characteristics of a queue (pronounced like the letter 'Q'):

- **Container**: The queue is designed to store a collection of data.
- **End Access**: Only elements on the front and the back of the queue are accessible.
- **Order Preserved**: The arrangement of elements is determined by insertion order.
- **Expandable**: The capacity of the queue can be increased to meet the needs of the client.
- **Restricted Insertion**: Adding elements to any location of the queue but the end is prohibited.

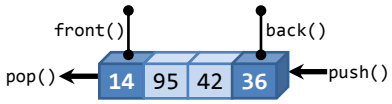
The queue is perhaps the most familiar data structure to the average person. It is commonly known as a line. When you are waiting in line to check out at the grocery store, you are waiting in a queue. When you wait your turn at a stop sign, you are in a queue. When you send a document to the printer, the document enters the printer's queue. Anytime someone or something is put in line, a queue is used.

Definition

A queue is called a First-In-First-Out (FIFO) data structure. It can equally be called Last-In-Last-Out (LILO). A queue is an ordered collection of elements where the client can remove elements from only one end (the front) and add access to the other (the back). The operations performed on a queue are the following:

Name	Example	Description
Create	`queue <int> q;`	Create a new queue.
Assignment	`q1 = q2;`	Assignment, make a copy of a queue.
Empty	`f = q.empty();`	Returns `true` if a queue has no elements, `false` otherwise.
Size	`n = q.size();`	Returns the number of elements currently in a queue.
Clear	`q.clear();`	Remove all elements from a queue.
Push	`q.push(e);`	Adds an element to the end of a queue. The size will increase by one.
Pop	`q.pop();`	Remove the front element from the queue, thereby reducing the size by one.
Front	`e = q.front();`	Returns the front element of the queue.
Back	`e = q.back();`	Returns the back element of the queue.

Aside from the standard container interfaces, the queue has a push(), pop(), front(), and back() method. The queue can be represented with the following figure:



Whereas elements were retrieved from the stack with the top() method, front() works the same way with the queue. Observe how the only elements rendered in dark blue are the ones under front() and back(), indicating that the one under front() is the next to be removed from the queue and the one under back() is the most recently added to the queue.

Elements are added to the queue on one end with push() but are removed from the other end with pop().

Push

Push involves adding an element to the end of the queue. This process is also called insert, enque, or add.

```
void queue <T> :: push(const T & t) throw (const char *);
```

As with the `stack`, the `queue` should grow to accommodate any number of elements; if the `queue` is full then it will reallocate. However, if no memory is available, an exception will be thrown.

	Code	Description	State
1	`queue <int> q;`	The default constructor does not allocate a buffer so the size and capacity are zero.	
2	`q.push(1);`	Add one to the back of the queue.	1 ← push(1)
3	`q.push(2);`	Add another element to the back of the queue. Notice how the front remains unchanged.	1 2 ← push(2)
4	`q.push(3);`	Again we add to the back and the front remains unchanaged.	1 2 3 ← push(3)

Note that the oldest element on the queue is on the left. This is the front of the queue (with the number `1`). The newest element is on the right, called the back or tail of the queue.

Pop

Pop involves removing the element currently at the head of the queue. It is also called remove, serve, or dequeue.

```
void queue <T> :: pop()
```

If the queue is already empty, then nothing will happen.

	Code	Description	State

1	`queue <int> q;`
	`q.push(1);`
	`q.push(2);`
	`q.push(3);`
		The default constructor does not allocate a buffer so the size and capacity are zero. Then we add the values `1`, `2`, `3` in order. When finished, `1` is at the front and `3` is at the back.	← push(1)
			← push(2)
			← push(3)
			1 2 3

2	`q.pop();`	Remove `1` from the front. This means that `2` is now at the front.	pop() ← 2 3
3	`q.pop();`	Remove `2` from the front. This means that `3` is now at the front.	pop() ← 3
4	`q.push(4);`	Now we will add `4` to the tail. This means that `3` remains at the front but `4` is the new back.	3 4 ← push(4)
5	`q.pop();`	When we remove `3` from the front, then `4` becomes both the front and the back of the queue.	pop() ← 4
6	`q.pop();`	By popping off the final element, we are now at an empty queue.	pop() ←
7	`q.pop();`	Attempting to pop from an empty queue does nothing. No exception is thrown and the queue does not change.	pop() ←

Front & Back

The `front()` method accesses the oldest element pushed onto the queue:

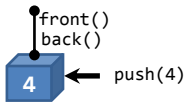
```
      T & queue <T> :: front()       throw (const char *);
const T & queue <T> :: front() const throw (const char *);
```

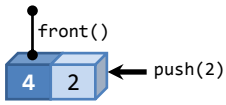
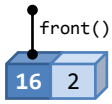
The `back()` method access the newest item pushed onto the queue:

```
      T & queue <T> :: back()        throw (const char *);
const T & queue <T> :: back()  const throw (const char *);
```

There are two things to take into account. The first is that the `queue` might be empty. Attempts to access `queue::front()` or `queue::back()` on an empty queue will result in an exception being thrown. The second consideration is that the client needs to be able to modify the top element of the queue through the `queue::front()` statement. In other words, `queue::front()` needs to work both as a getter and as a setter.

	Code	Description	State
1	`queue <int> q;`	From an empty queue, attempts to call `front()` will result in a throw exception.	

2	`q.push(4);`
	`assert(q.front() == 4);`
	`assert(q.back()  == 4);`
		When we add `4` to the queue, then we can equally access it with `front()` or `back()`. Here the front of the queue is the back of the queue: the value `4`.	

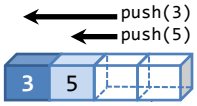
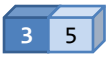
3	`q.push(2);` `assert(q.front() == 4);`	When we push `2` to the back, then `4` remains at the front.	
4	`q.front() = 16;` `assert(q.front() == 16);`	We can set the front to `16` from the old value of `4`. Here we are demonstrating `front()` as a setter and as a getter. Of course we could do the same thing with `back()`.	

Assignment

The assignment operator makes a copy of the queue.

```
queue <T> & queue <T> :: operator = (const queue <T> & rhs) throw (const char *);
```

This does not mean that the copy needs to be identical to the original. Instead, it means that it has to behave the same from the client's perspective.

	Code	Description	State
1	`queue <int> q1(4);`	Create a queue with four empty slots.	
2	`q1.push(3);` `q1.push(5);`	By adding two elements, two remain unfilled.	
3	`queue <int> q2;`	The default constructor leaves the queue empty.	
4	`q2 = q1;`	The assignment operator copies the values but not necessairely the capacity.	

This works exactly the same as the stack. The assignment operator is obligated to copy the contents of the queue, but not the capacity.

Standard Container Interfaces

The size is simply the number of elements currently in the `queue`, working the same as the `stack` and `vector`. Additionally we have a `clear()` and `empty()` method just like with the `stack` and `vector`.

```
int  queue <T> :: size()  const;
bool queue <T> :: empty() const;
void queue <T> :: clear();
```

Notice how `size()` and `empty()` are constant methods because they does not change `this`.

	Code	Description	State

1	`queue <int> q;`
	`assert(q.size() == 0);`
	`assert(q.empty());`
		Create an empty `queue`. The capacity is initially zero and, of course, it is empty.	

2	`q.push(10);`
	`assert(q.size() == 1);`
	`assert(q.empty() == false);`
		With one push, we should have size that is one. The capacity is also one and it is no longer empty.	10 push(10)

3	`q.push(20);`
	`assert(q.size() == 2);`
	`assert(q.empty() == false);`
		With the second push, the size is now two. The capacity has doubled.	10 20 push(20)

4	`q.pop();`
	`assert(q.size() == 1);`
	`assert(q.empty() == false);`
		With `10` popped off, we have only one item remaining.	pop() 20

5	`q.clear();`
	`assert(q.size() == 0);`
	`assert(q.empty() == true);`
		The `clear()` method removes all elements from the queue. Thus the size is zero and the `queue` is now empty.	pop()

Implementation

There are several implementation considerations for the `queue` data structure. First, and perhaps most obviously, is that the `queue` should grow to accommodate any amount of data. Thus a resize or grow function must be present.

The second is that the implementation should minimize the number of reallocations and make best use of the existing available memory. This consideration is particularly important for the `queue`. To illustrate this point, consider a `stack` consisting of two elements. If the client pushes two elements on the `stack`, then the `stack` is full. If the client then performs a succession of pop, push, then the `stack` still only requires two elements:

```
{
   stack <int> s;      // s == {}
   s.push(1);          // s == {1}
   s.push(2);          // s == {1, 2}
   s.pop();            // s == {1, ~~2~~}  <-- with the pop(), there is an empty slot
   s.push(3);          // s == {1, 3}  <-- the new element fills the vacated space of 2
   s.pop();            // s == {1, ~~2~~}
   s.push(4);          // s == {1, 4}  <-- again only two slots are needed here
}
```

If we continue this indefinitely, the `stack` will only ever require two slots in memory. This is not the case with the queue.

```
{
   queue <int> q;      // q == {}
   q.push(1);          // q == {1}
   q.push(2);          // q == {1, 2}
   q.pop();            // q == {~~1~~, 2}        <-- note how the 1 is released, not the 2
   q.push(3);          // q == {~~1~~, 2, 3}     <-- we needed to grow the queue
   q.pop();            // q == {~~1~~, ~~2~~, 3}
   q.push(4);          // q == {~~1~~, ~~2~~, 3, 4}  <-- note how we keep building to the right
}
```

If we continue this indefinitely, the `queue` will require more and more memory. We call this "queue creep." This is even the case when we only ever have two elements in the queue at any point in time. How can we avoid such a wasteful use of memory? We basically have three implementation choices for an array-based queue:

a) **Grow indefinitely**. This is the default option mentioned above. Though it is relatively efficient with regards to time, it is very inefficient with regards to space.
b) **Shift the queue**. Every time we `pop()` an element off the head of the `queue`, we shift all the elements in the `queue` over by one slot. This will clearly minimize the amount of memory used by the data structure, but it will be expensive. While `push()` will remain O(1) in the common case (and O(n) in the worst case when more memory is needed), `pop()` will become O(n) in all cases. There must be a better way.
c) **Wrap the queue**. This is a bit complex to explain. Instead of walking forward indefinitely, we should wrap back to the beginning of the array once the end is reached. If we could do this, we would `push()` in O(1) in the common case and always `pop()` in O(1).

Clearly the wrap option is preferable if we can do it. The question is: how? How can we know to restart at zero once we reach the end of the buffer? How can we know that the queue is full and not run into the other tail?

To visualize how this works, imagine an array of four elements. We will extend this array to the next four elements were index 0 is the same as index 4. If we continue this indefinitely, then the array looks like this:

3	0	1	2	3	0	1	2	3	0	1	2	3	0
					A	B	C	D					

If `index == 2`, then the result of incrementing our index by one yields `index == 3`. Since this is our greatest index, then incrementing by one will yield `index == 0`. In other words:

```
index = index % 4; // when the size of the array is 4
```

This allows our index variable to wrap. How do we wrap the entire `queue`? In order to do this, we need to keep track of both the index of the front (or head) of the queue (`iHead`) as well as the back (or tail) of the queue (`iTail`). As we add elements to the queue, the `iTail` index will increase by one. As we take elements off the front of the queue, the `iHead` index will increase by one. In both cases, we will wrap the indices with the modulus operator.

To illustrate how this works, consider a `queue` containing one element (the number `101`). From here we will add three numbers (`102`, `103`, and `104`) and then pop off two (`101` and `102`), and add a number (`105`):

	Code	Description	State
1	`queue q(4);` `q.push(101);`	The queue has four slots and one element is filled. `iHead == 0`, `iTail == 0`.	iHead, iTail: 101 | ? | ? | ?

2	`q.push(102);`
	`q.push(103);`
	`q.push(104);`
		All four slots in the queue are filled.
		`iHead == 0`, `iTail == 3`.
			iHead: 101 | 102 | 103 | 104 :iTail

3	`q.pop();` `q.pop();`	Now only the last two numbers remain. Slots 0 and 1 are now empty. `iHead == 2`, `iTail == 3`.	? | ? | iHead: 103 | iTail: 104
4	`q.push(105);`	Notice how the next value goes in slot 0. In other words, the queue now wraps around. `iHead == 2`, `iTail == 0`.	iTail: 105 | ? | iHead: 103 | 104

5	`q.pop();`
	`q.pop();`
	`q.pop();`
	`q.push(101);`
		The third pop yields an empty queue. From there, we can start again anywhere we want. In this case, we will start at slot zero.	iHead, iTail: 101 | ? | ? | ?

There is one more problem yet to be solved. How can one tell the difference between an empty queue and a full queue? Notice the case where there is one element in the queue array. In this case, `iHead` and `iTail` both refer to the same element in the array. Since we move `iTail` when we add elements to the `queue`, it follows that `iHead` should be at `0` when the `queue` is initialized and `iTail` should be at `-1` (or index 3 in this case). This is exactly the same condition when the queue array is full:

iHead			iTail
?	?	?	?

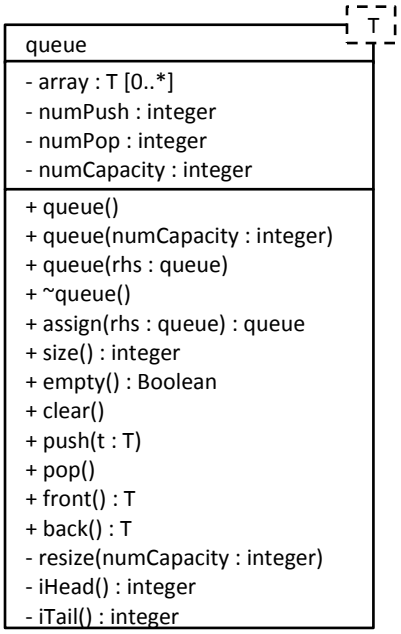
iHead			iTail
101	102	103	104

From here we have three options:

a) Keep an extra Boolean variable that indicates whether the queue is empty.
b) Keep one slot in the array always empty so we never get to the full state. When we attempt to put another element in the array and there is one slot left, we grow the array.
c) Keep a version of `iHead` and `iTail` that is not normalized (not between `0` and `numCapacity - 1`). We will call these `numPush` and `numPop`. This solution avoids the wasted space of the first two:

```
iTail ← (numPush – 1) % numCapacity;
iHead ← numPop % numCapacity
```

The third option is the best here because we avoid using an extra variable and we avoid wasting one slot in the array. Based on this, our UML class diagram of the queue is:



Size

Our `queue` implementation does not have a `numElements` member variable that our `stack` class did. This means we need to compute the number of elements based on the member variables that keep track of the head and tail of the array. Since there are many ways we can do this, our implementation of the size method will vary. Assuming that we use the `numPush` and `numPop` implementation, the pseudocode is:

```
queue.size()
  RETURN numPush - numPop
```

Assignment

The assignment operator is not quite as simple as blindly copying the elements from the right-hand-side into the left-hand-side. That would involve copying the unreferenced data and might require resizing the queue to a smaller size. Both of these are unnecessarily expensive.

A better solution is to walk through the right-hand-side array in order from `iHead` to `iTail` and push those elements into the left-hand-side.

```
queue.assignment (lhs, rhs)
  numPush ← numPop ← 0
  IF lhs.numCapacity < rhs.size()
    resize(rhs.size())
  FOR i ← rhs.numPop … rhs.numPush
    lhs.push(rhs.array[i % rhs.numCapacity])
```

Notice how we take elements from the head of the right-hand-side (starting at `rhs.numPop % rhs.numCapacity` and ending with `rhs.numPush % rhs.numCapacity`) and push them onto the left-hand-side. The new left-hand-side will honor the queue contract exactly the same as the right-hand-side queue. However, the `array`, the `numPush`, and `numPop` member variable will be different.

Push

Adding elements to the end of the queue is as simple as putting a new element in `iTail + 1`.

```
queue.push(e)
  array[++iTail] ← e
```

There are two problems with this simplistic design. First, the `queue` could already be full. If this is the case, then we need to resize the queue first. Second, we need to take wrapping into account.

We can handle the full queue case with a simple IF statement similar to our design with the `stack` and the `vector`. The wrapping case can be more complicated. We will avoid this complication by assuming we have a private method called `iTail()` which returns the index of the tail in the array after wrapping has been taken into account. Thus our new pseudocode for `push()` is the following:

```
queue.push(e)
  IF size() = capacity()
    resize(capacity() * 2)
  numPush++
  array[iTail()] ← e
```

All the operations in this method except `resize()` are O(1). This is because all of their executions are independent of the size of the array. Note that `resize()` is O(n) because the data from the old array must be copied into the new array one element at a time. One might think that makes this entire method O(n) but it is not! We only rarely need to resize the array. The number of times the array is resized is Log_2n. This means the method is O(1) in the common case, O(log n) in the average case, and O(n) in the worst case.

Pop

Removing elements off the head of the queue is as easy as advancing `numPop`.

```
queue.pop()
  IF not empty()
    numPop++
```

As you can imagine, there is a complication: What happens when the queue is already empty? In that case, we leave the queue unchanged and return out of the method.

Front & Back

Retrieving the element off the head of the queue is as easy as returning the value at `iHead`:

```
queue.front()
  IF empty()
    ERROR
  ELSE
    RETURN array[iHead()]
```

Of course, `back()` works the same way:

```
queue.back()
  IF empty()
    ERROR
  ELSE
    RETURN array[iTail()]
```

Application

As with the stack, the queue is a very commonly used data structure. Every time we want to preserve order in a collection of elements, a queue should be the first choice. To illustrate this, we work through four examples: a printer queue, a queue of students waiting to be helped by a lab assistant, and two CPU scheduling algorithms.

Printer queue

A printer queue is a set of documents set to the printer that will eventually be printed. By default, these are accomplished in a FIFO order. Each print job will have a name and number of pages. We will assume that our printer will be able to complete one page every 10 seconds. Also, every 10 seconds we will give the user the opportunity to add another element to the print queue. To do this, we will create a class representing a single printed page:

Page
- documentName - pageNumber
+ Page + display

In order to add documents to the queue, we will provide a function that takes the document name and number of pages as a parameter. Notice how a single document gets split into multiple pages, each page having a unique entry in the queue.

```
void addDocument(const string & documentName, int numPages,
                 queue <Page> & pages)
{
   for (int pageNumber = 1; pageNumber <= numPages; pageNumber++)
      pages.push(Page(documentName, pageNumber));
}
```

The final step is to create a loop that allows the user to add elements to the printer queue. Usually this loop is called the "spooler" or the "printer spooler queue." We will make this interactive. Each time unit (10 seconds) the printer will print another page while the user will have the opportunity to add a new document.

```
void spooler()
{
   queue <Page> pages;                 // initially the printer queue is empty

   for (;;)                            // the spooler runs as long as the OS is active
   {
      string input;
      cout << "> ";                    // with this prompt, the user either specifics
      cin  >> input;                   //    a document or enters "none"

      if (input != "none")             // add a new document to the queue
      {
         int numPages;
         cin >> numPages;
         addDocument(input, numPages, pages);
      }

      if (!pages.empty())              // if the queue is not empty, then print a page
      {
         pages.front().display();      // this will simulate a page being printed
         pages.pop();
      }
   }
}
```

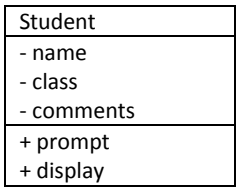
Now Serving

Imagine a university computer lab serving a large number of students working on assignments for several classes. This lab has three lab assistants, each wishing to serve students at a first-come-first-serve basis. A student indicates they need help by putting their name into the system. When a lab assistant finishes working with a student, then he or she takes the next student in line.

This clearly is an application for a queue (note that first-come-first-serve sounds a great deal like FIFO, the distinguishing characteristic of a queue). This application will have four parts: the data-structure containing the students, the function to add a student to the queue, the function displaying the name of the student currently being served, and the function removing a student off the queue.

Data Structure

To define the data structure containing the line of students, we will first need a class to contain the notion of one student. A UML class diagram describing this class might be the following:



From here we can define the entire data structure quite simply:

```
queue <Student> line;
```

Add

When a student needs help on their assignment, they need the abilty to add their name to the end of the line. This can be accomplished with the `push()` method.

```
void add(queue <Student> & line)
{
   Student s;
   s.prompt();

   line.push(s);
}
```

Display

At the front of the computer lab, a computer terminal displays the name of the student currently being served. This can be accomplished with the `front()` method.

```
void display(const queue <Student> & line)
{
   line.top().display();
}
```

Remove

The last operation that needs to be performed is to remove a student from the line once he or she has been helped with the problem. This can be accomplished with the `pop()` method.

```
void remove(queue <Student> & line)
{
   line.pop();
}
```

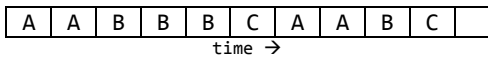
CPU Scheduling

One of the many tasks of an operating system (OS) (like Microsoft Windows, Apple OS-X, Linux, Google Android, and Apple iOS) is to make sure the CPU is shared by all the programs that want to use it. If, for example, you are running three apps on your phone (we will call them A, B, and C for convenience), then the OS will swap A, B, and C so that it appears to the user that your phone actually has three CPUs. This brings us to two questions:

a) How often does the OS shift control from one program to another?
b) How does the OS choose which program will take control next?

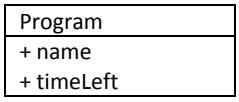
The answer to these questions is a topic called CPU Scheduling.

Consider the following figure. On the X-axis runs time from left to right. Every vertical line represents one unit of time (say 1 millisecond). The OS decides which program should be given access to the CPU and which should be dominant every unit of time. If a program is being executed for a certain unit of time, then the program name will be indicated in the space after the vertical line.



Observe how first program A controlled the CPU for 2 units of time, then program B for three units, then C for one unit, then back to A for two units and so on.

We will simulate CPU scheduling by creating a structure that contains the notion of a program. This structure will have two member variables: the name of the program and the remaining time until the program is complete.



There are many strategies employed by operating systems to schedule programs for the CPU. We will explore two: FCFS and Round Robin. As you can imagine, they both use the queue.

FCFS

First Come First Serve (FCFS) scheduling means the first program to be scheduled is given complete access to the CPU. Only when it is finished is control given to the second program and so on. You may have noticed that FCFS looks a great deal like FIFO. This is no coincidence; they are the same!

Our algorithm to simulate FCFS scheduling is the following:

```
void scheduleFCFS()
{
   queue <Program> schedule;
   schedule.push(Program("A", 3));              // 'A' will run for 3 units of time
   schedule.push(Program("B", 1));              // 'B' will run for 1 unit  of time
   schedule.push(Program("C", 2));              // 'C' will run for 2 units of time

   while (!schedule.empty())
   {                                            //   the program at the head of the
         cout << schedule.head().name << ' ';   //   queue has control. Output its
         schedule.head().timeLeft--;            //   name and decrement timeLeft

      if (schedule.head().timeLeft <= 0)        // if there is no time left, it is
         schedule.pop();                        //   finished so relinquish control
   }
}
```

To see how this works, let's trace the output as the program executes:

	Description	State	Output
1	The queue is initialized with three programs waiting for execution. 'A' is at the head of the queue, followed by 'B' and then 'C'.	A/3 B/1 C/2	
2	At the end of the first iteration, 'A's `timeLeft` is decremented by one.	front() A/2 B/1 C/2	A
3	'A' remains at the head of the queue and thus controls the CPU because there is still time left.	front() A/1 B/1 C/2	A A
4	This third time through the loop sees A's time left decrement to zero. It will be popped off next.	front() A/0 B/1 C/2	A A A
5	'A' was popped off the queue in the last round so now it has relinquished control. 'B' is currently running on the CPU now.	front() pop() B/0 C/2	A A A B
6	At the end of the fifth iteration, 'C' is currently running. It has only one cycle left.	front() pop() C/1	A A A B C
7	At the end of the fifth iteration, 'C' is popped off the queue and the simulation ends.	pop()	A A A B C C

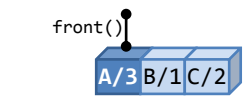
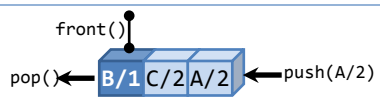
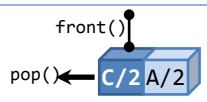
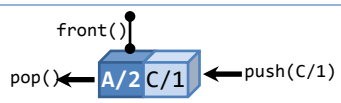
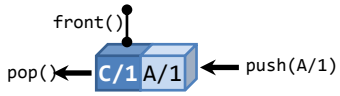
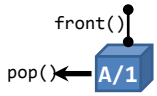
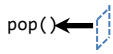
Round Robin

Another scheduling scheme is called Round Robin where only one unit of CPU time is given to each program before control is relinquished to the next program. Once a program has executed for one cycle, it is popped off the queue and put at the end.

Our algorithm to simulate Round Robin scheduling is the following:

```
void scheduleRoundRobin()
{
   queue <Program> schedule;
   schedule.push(Program("A", 3));            // 'A' will run for 3 units of time
   schedule.push(Program("B", 1));            // 'B' will run for 1 unit  of time
   schedule.push(Program("C", 2));            // 'C' will run for 2 units of time

   while (!schedule.empty())
   {
      Program p = schedule.head();            // the head element is the program being
      schedule.pop();                         //   executed. Remove it from the queue

      cout << p.name << ' ';                  // run the program for one unit and
      p.timeLeft--;                           //   decrement the timeLeft

      if (p.timeLeft > 0)                     // if there is any time left, add it
         schedule.push(p);                    //   to the back of the queue
   }
}
```

To see how this works, let's trace the output as the program executes:

	Description	State	Output
1	The queue is initialized with three programs waiting for execution. 'A' is at the head of the queue, followed by 'B' and 'C'. 'A' is then currently executing.		A
2	At the end of the first iteration, 'A' completed its execution for one time unit and got put at the back with the `timeLeft` is decremented by one. 'B' is now executing.		A B
3	Once 'B' completed its execution, it is finished so it did not get added to the back of the queue. 'C' is now executing.		A B C
4	Next 'C' completed its execution with one unit of time left. Now 'A' is executing.		A B C A
5	'C' is executing for the final time. When this unit is finished, it will be popped off the queue and not added back on.		A B C A C
6	With 'C' gone, only 'A' remains.		A B C A C A
7	At the end of the sixth iteration, 'A' has completed the last time unit and we are done.		A B C A C A

Stocks

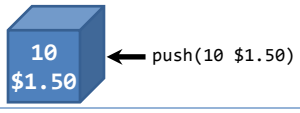
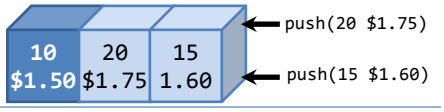
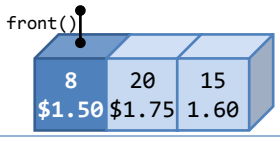
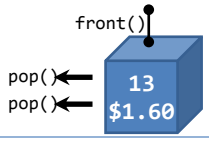
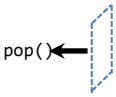
One way an individual could invest money is by purchasing stocks. A stock represents a partial ownership in a company. For example, a company may be worth $10,000,000. If there are 100,000 shares distributed by this company, than each share is worth $100. Now one day, this company has a great new product that everyone wants, the value of the company in the eyes of the shareholders may jump to $15,000,000. Now each stock is worth $150.

Write a program to represent a stock portfolio. An individual can buy stocks any number of stocks at the current market price. These stocks are then placed in the individual's portfolio. The person can also sell as many of the stocks in his portfolio as he likes at the current market price. When this is done, a record is made of which stocks were sold and for what price. To understand how this is done, a few concepts need to be explained:

- When a share of common stock of a company is sold, the profit (or loss) is the difference between the share's selling price and the share's purchase price. For example, if 10 shares are bought at $1.50 and then sold at $2.00, the profits are ($2.00 - $1.50) x 10 = $5.00.
- When we buy shares of stock over a long period of time and then decide to sell some (but not all) of them, we must identify the shares actually being sold. The accepted method for doing this is to sell the oldest ones first. For example, if 10 shares are bought at $1.50 and then 20 shares are bought for $1.75, then I will have 30 shares in my portfolio. If I then sell two shares from my portfolio for $2.50, then two shares are deduced from the oldest transaction: ($2.50 - $1.50) x 2 = $2.00. I will then have 8 shares remaining from the original batch and the full 20 from the second batch.

Note that one sell order can take more than one purchase batch to satisfy. Consider the case when I have made three purchases: 10 shares for $7.50, 12 shares for $7.80, and 4 shares for $9.00. All told, I have 26 shares. Now I wish to sell 25 shares for $10. My proceeds will be: ($10.00 - $7.50) x 10 + ($10.00 - $7.80) x 12 + ($10.00 - 9.00) x 3 = $54.40. I will also have one share left which was purchased for $9.00.

To see how this works, consider the following execution:

	Action	Description	State
1	`buy 10 $1.50`	The first step is to add 10 shares of $1.50 to the back of the queue. Now there is exactly one element.	
2	`buy 20 $1.75` `buy 15 $1.60`	Two more "buy"s make for a total of three elements in the queue. I can access the head of the queue and even the tail, but the middle transaction is unavailable.	
3	`sell 2 $2.00`	The first sell removes few stocks from my first batch but I still have three batches in the queue.	
4	`sell 30 $2.10`	This second sell requires stocks from all three batches. The first two batches are emptied and the third batch is partially reduced.	
5	`sell 13 $1.40`	My final act is to sell the remaining stocks. Note that I am selling them for less than the purchase price, resuling in a net loss.	

Examples

Example 03.1

What is the state of the queue after the following code is executed:

```
{
   queue <char> q;          // #1

   q.push('Q');             // #2
   q.push('W');             // #3
   q.push('E');             // #4
   q.push('R');             // #5
   q.push('T');             // #6

   q.pop();                 // #7
   q.pop();                 // #8
}
```

Answer:

#1		Initially empty.

#2	iHead, iTail
	Q
		Capacity goes to 1 and the first slot is filled. `numPush = 1` and `numPop = 0` so:
		`iTail = (numPush - 1) % numCapacity`
		`iHead = numPop % numCapacity`
		Thus `iTail == numPop == 0`.

#3	iTail, iHead Q W	Capacity doubles and the second slot is filled. Now `numPush = 2` so `iTail = 1`. The capacity doubled.
#4	iTail, iHead Q W E	Capacity doubles again to 4. The third slot is filled but the fourth remains uninitialized.
#5	iTail, iHead Q W E R	At this point, the size and the capacity of the queue are the same. The queue is filled.
#6	iTail, iHead Q W E R T	When we push an element onto a full queue, the capacity is doubled again.

#7	iHead, iTail
	W E R T
		With a `pop()`, the `iHead` index moves to the right by one slot because `numPop` now equals 1.
		`iTail = (5 - 1) % numCapacity = 5`
		`iHead = 1 % numCapacity = 1`

#8	iHead, iTail E R T	Another `pop()` and `iHead` moves to the right again and `numPop = 2`. The size is now 3. `iHead = 3 % numCapacity = 3`

Example 03.2

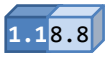
What is the state of the queue after the following code is executed:

```
{
   queue <float> q;          // #1

   q.push(2.2);              // #2
   q.push(8.8);              // #3
   q.pop();                  // #4
   q.push(1.1);              // #5
   q.push(0.0);              // #6
}
```

Answer:

#1		Initially empty
#2	iHead iTail `2.2`	Capacity goes to 1 and the first slot is filled. Both `iHead` and `iTail` are assigned to the same element.
#3	iHead iTail `2.2` `8.8`	Capacity doubles and the second slot is filled. Now `iTail` refers to a different element than `iHead`. The size is now 2.

#4	iHead
	iTail
	`8.8`
		With the pop, `iHead` and `iTail` both refer to the same element. Here `numPush = 2` and `numPop = 1`.
		`iTail = (numPush - 1) % numElements = (2 - 1) % 2 = 1`
		`iHead = (numPop) & numElements = 1 % 2 = 1`

#5	iTail iHead
	`1.1` `8.8`
		Now it gets confusing. The tail appears to be before the head. This corresponds to the following figure:
		 
		Note that `numPush = 3` and `numPop = 1`. Thus:
		`iTail = (numPush - 1) % numElements = (3 - 1) % 2 = 0`
		`iHead = (numPop) & numElements = 1 % 2 = 1`

#6	iHead iTail
	`8.8` `1.1` `0.0`
		Another push on the full queue causes the queue to reallocate. When we reallocate, we "unwrap" the queue and put the head at the start of the array. Here we set:
		`numPush = 3`
		`numPop = 0`

Exercises

Exercise 03.1

From memory, recite and define the operations associated with the queue ADT.

Answers:

_____

_____

_____

_____

_____

_____

Please see page 76 for a hint.

Exercise 03.2

Write out the state of queue's member variables after each line or group of lines are executed:

```
{
   queue <int> q;

   s.push(85);

   s.push(52);

   s.push(61);
}
```

numCapacity	numPush	numPop	array

Please see page 77 for a hint.

Exercise 03.3

Write out the state of queue's member variables after each line or group of lines are executed:

```
{
   queue <int> q;
   q.push(92);
   q.push(61);

   q.pop();

   q.push(49);

   q.pop();
}
```

numCapacity	numPush	numPop	array

Please see page 77 for a hint.

Exercise 03.4

Write out the state of queue's member variables after each line or group of lines are executed:

```
{
   queue <int> q;
   q.push(49);
   q.push(89);
   q.pop();
   q.push(64);

   q.push(79);

   q.pop();
   q.push(11);
   q.push(50);

   q.pop();
   q.pop();

   q.front() = 3;
   q.back()  = 99;

   q.push(49);
   q.push(73);
   q.push(64);
}}
```

numCapacity	numPush	numPop	array

Please see page 78 for a hint.

Exercise 03.5

Consider the following stock transactions:

```
> buy 20 $31.75
> buy 30 $35.61
> sell 10 $25.14
```

a. What is left in my stock portfolio?

b. What is in my transaction history?

c. How much money did I make or loose from these transactions?

Please see page 89 for a hint.

Exercise 03.6

Consider the following stock transactions:

```
> buy 20 $30.00
> buy 30 $35.00
> sell 20 $25.00
```

a. What is left in my stock portfolio?

b. What is in my transaction history?

c. How much money did I make or loose from these transactions?

Please see page 89 for a hint.

Exercise 03.7

Consider the following stock transactions:

```
> buy 125 $100.00
> buy 75 $110.00
> buy 150 $120.00
> sell 300 $125.00
```

a. What is left in my stock portfolio?

b. What is in my transaction history?

c. How much money did I make or loose from these transactions?

Please see page 89 for a hint.

Exercise 03.8

Consider the following stock transactions:

```
> buy 12 $105.00
> buy 15 $107.00
> buy 30 $110.00
> sell 30 $125.00
> buy 40 $106.00
> sell 50 $109.00
```

a. What is left in my stock portfolio?

b. What is in my transaction history?

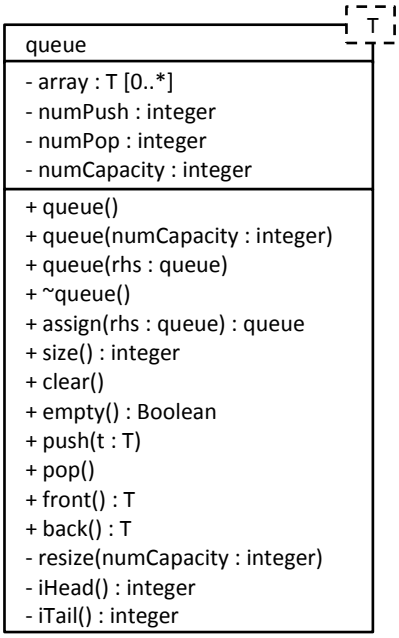
c. How much money did I make or loose from these transactions?

Please see page 89 for a hint.

Problems

Problem 03.1

Implement your own version of the `std::queue` data structure. Call your implementation `custom::queue`. In other words, place your class in the custom namespace and implement all of the interfaces described in this chapter. The UML class diagram of the queue class is the following:



Please see page 80 for a hint.

Problem 03.2

Using the STL `queue` or using your own implementation of `queue` from Problem 03.1, write a program to simulate the buying and selling of stock transactions. An individual can buy stocks any number of stocks at the current market price. These stocks are then placed in the individual's portfolio. The person can also sell as many of the stocks in his portfolio as he likes at the current market price. When this is done, a record is made of which stocks were sold and for what price. To understand how this is done, please see the Stocks application.

Please see page 89 for a hint.

Sam is playing wooden trains with his nephew Neil. Each car is connected to the other cars with a magnet at the front and at the back. Neil likes to periodically add and remove cars to the back and the front of the train as Sam pushes it around the track. It is simple to do this because cars easily snap onto the magnets. It is much more difficult to add or remove cars from the middle. Whenever Neil tries this, the cars come off the track and everything is messed up. For the reason, he just snaps cars onto the ends.

This toy train works much like a deque (pronounced "deck"): a data structure that gives the client full access to either end of the list but no access to the center. The name deque is an abbreviation for "double-ended queue." Whereas the client can only add elements to the tail of the queue and remove elements from the head, the client can add elements to either side of the deque and remove elements from either side as well.

The deque could just as easily be called a "double-ended queue." While the client can add and remove elements from *one* end of the stack, the client can add or remove elements from *either* side of the deque.

The essential characteristics of a deque are:

- **Container**: The deque is designed to store a collection of data.
- **End Access**: Only elements on the front and the back of the deque is accessible.
- **Order Preserved**: The arrangement of elements is determined by insertion order.
- **Expandable**: The capacity of the deque can be increased to meet the needs of the client.
- **Restricted Insertion**: Adding elements to any location of the deque but the ends is prohibited.

As you can imagine, it is trivial to implement a stack from a deque: simply ignore the add/remove operations from one side. It is also trivial to implement a queue from a deque: one only has to ignore the add operations from one side and ignore the remove operations from the other.

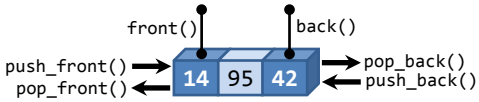
It is easy to see examples of the stack and queue in the world around us. The deque is not so commonly found. Perhaps the most accessible example is those wooden train models we played with in preschool like Sam's nephew Neil. Each train car is held to the next one by a magnet. The child can add or remove cars to either side of the train, but cars cannot be added or removed from the middle without lifting the train.

Definition

A deque is a sequence container that can be expanded or contracted on either end. The operations performed on a deque are the following:

Name	Example	Description
Create	`deque <int> d;`	Create a new deque.
Assignment	`d1 = d2;`	Make a copy of a deque.
Empty	`f = d.empty();`	Returns `true` if a deque has no elements, `false` otherwise.
Size	`n = d.size();`	Returns the number of elements currently in a deque.
Clear	`d.clear();`	Remove all elements from a deque.
Push Front	`d.push_front(e);`	Adds an element to the front of a deque, increasing the size by one.
Push Back	`d.push_back(e);`	Adds an element to the back of a deque.
Pop Front	`d.pop_front();`	Removes the front element from a deque, reducing the size by one.
Pop Back	`d.pop_back();`	Remove the back element from a deque.
Front	`e = d.front();`	Returns the front element of the deque.
Back	`e = d.back();`	Returns the back element of the deque.

The deque can be represented with the following figure:



Observe how the client has direct access to both the front and the back of the deque. This can be accomplished through the `front()` and `back()` methods. The middle elements (just `95` in this case) are not directly accessible.

It is possible to add elements from the front or back of the deque (with `push_front()` and `push_back()` respectively) as well as removing elements from the front or back of the deque (with `pop_front()` and `pop_back()` respectively).

Push Front & Back

Push front and push back involves adding an element to one end of the deque.

```
void deque <T> :: push_front(const T & t) throw (const char *);
void deque <T> :: push_back (const T & t) throw (const char *);
```

As with the queue and stack, the deque should grow to accommodate any number of elements; if the deque is full then it will reallocate. However, if no memory is available, an exception will be thrown.

	Code	Description	State
1	`deque <int> d;`	The default constructor does not allocate a buffer so the size and capacity are zero.	
2	`d.push_front(1);`	Pushing to the front and to the back of an empty deque have the same result.	push_front(1) → 1
3	`d.push_back(2);`	The back of the deque is the right side.	1 2 ← push_back(2)
4	`d.push_front(3);`	Add 3 to the front.	push_front(3) → 3 1 2
5	`d.push_back(4);`	Again back puts things on the right side.	3 1 2 4 ← push_back(4)

Pop Front & Back

Pop front removes an element from the front of the deque and pop back removes an element from the back as one would expect.

```
void deque <T> :: pop_front();
void deque <T> :: pop_back();
```

The standard template library does not specify what happens when the client attempts to pop from an empty deque. In our implementation, we will throw an exception and leave the deque unchanged.

	Code	Description	State

1	`deque <int> d;`
	`d.push_back(2);`
	`d.push_back(3);`
	`d.push_front(1);`
	`d.push_front(0);`
		Create an empty deque and add four elements. Notice that 2 and 3 are put on the back and 0, 1 are put on the front.	push_front(1) → push_front(0) → 0 1 2 3 ← push_back(2) ← push_back(3)

2	`d.pop_front();`	Remove the element off the front, the value 0 in this case	pop_front() ← 1 2 3
3	`d.pop_back();`	The back of the deque is the right side.	1 2 → pop_back()
4	`d.pop_back();`	Remove another from the back.	1 → pop_back()
5	`d.pop_front();`	`pop_front()` and `pop_back()` are the same when the size is one.	pop_front() ←
6	`d.pop_back();`	If you attempt to pop from an empty deque, then nothing happens: the deque remains unchanged.	→ pop_back()

Front & Back

Unlike the stack and queue, the deque gives the user full access to either side of the list. This means the user can both read from either end and can update the value on either end. Access to the front of the deque is achieved through the `front()` method which, like the stack and queue, has both a read-only and a read-write version:

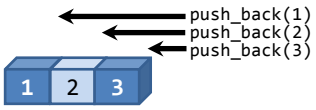
```
      T & deque <T> :: front()       throw (const char *);
const T & deque <T> :: front() const throw (const char *);
```

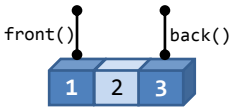
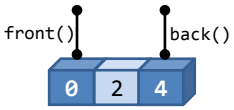
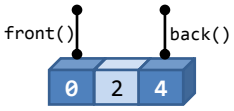
The back of the deque can be accessed with a pair of `back()` methods:

```
      T & deque <T> :: back()        throw (const char *);
const T & deque <T> :: back()  const throw (const char *);
```

As with pop, an exception will be thrown if the deque is empty when the user attempts to access `front()` or `back()`.

	Code	Description	State

1	`deque <int> d;`
	`d.push_back(1);`
	`d.push_back(2);`
	`d.push_back(3);`
		Create an empty deque and add three elements.	

2	`assert(d.front() == 1);` `assert(d.back() == 3);`	The front better have the value 1 and the back better have the value 3. Notice how `front()` and `back()` serve as getters in this instance.	
3	`d.front() = 0;` `d.back()  = 4;`	We can set the values at the front with `front()` and set the value at the back with `back()`. Notice how `front()` and `back()` serve as setters in this instance.	
4	`assert(d.front() == 0);` `assert(d.back() == 4);`	Verify that the front and back changed as we expect. Also we are using `front()` and `back()` as getters again.	

Standard Container Interfaces

The deque has the default constructor setting the capacity to zero and the buffer to NULL. There is also a non-default constructor for those times the client knows how many element are needed and would like to avoid unnecessary reallocations. Of course, the non-default constructor can throw because it might be that there is no memory available for allocation.

```
deque <T> :: deque();
deque <T> :: deque(int newCapacity) throw (const char *);
```

Whenever a class defines a dynamically-allocated buffer or works with pointers, then it is necessary to define the "big three." These include the copy constructor, the assignment operator, and the destructor. It is almost never the case that one would be needed without the other two.

Every time we have a copy constructor, we need also to have the assignment operator and destructor. These are known as the "big three":

```
deque <T> :: deque(const deque <T> & rhs) throw (const char *);
deque <T> & deque <T> :: operator = (const deque <T> & rhs) throw (const char *);
deque <T> :: ~deque();
```

Aside from these, the deque also returns the size, whether the container is empty, and provides a mechanism to clear the contents.

```
int  deque <T> :: size()  const;
bool deque <T> :: empty() const;
void deque <T> :: clear();
```

This is done exactly the same as with the vector, stack, and queue.

	Code	Description

1	`// set the capacity to 0`
	`deque <int> d1;`
	
	`// set capacity to 4 then add one element`
	`deque <int> d2(4);`
	`d2.push_back(8);`
		Create two deques. The first is empty and the second has a capacity of 4 and a size of 1.

2	`assert(d1.size() == 0);` `assert(d2.size() == 1);`	The size has nothing to do with how much space is allocated and has everything to do with how much space is actually used.
3	`assert(d1.empty() == true);` `assert(d2.empty() == false);`	The deque is empty only if the size is 0. This has nothing to do with the capacity.

4	`// remove all elements from d2`
	`d2.clear();`
	`assert(d2.size() == 0);`
	`assert(d2.empty());`
		We can always remove all the elements of a container with `clear()`. In most cases, this will not change the capacity. We will learn about an exception to this in Chapter 07.

5	`d1 = d2;`
	`assert(d1.size() == 0);`
	`assert(d1.empty());`
		The assignment operator only copies over the elements and thus changes the size. Since the size of `d2` was zero, it has the same effect as emptying `d1`.

Implementation

On the surface, it may seem that the implementation of the `deque` should be the same as the `queue`. After all, the `deque` is a "double-sided queue." Certainly we should follow a wrapping model similar to the circular queue we did earlier. As you may recall, we used two variables: `numPush` and `numPop`. While this is certainly a good place to start, it is not complete. After all, pushing to one side of the deque is quite different than pushing to the other side.

Consider the following code:

```
{
   deque <char> d(4);   // { ?,   ?,   ?,   ? }
   d.push_back('A');    // {'A',  ?,   ?,   ? }
   d.push_back('B');    // {'A', 'B',  ?,   ? }
   d.push_back('C');    // {'A', 'B', 'C',  ? }
   d.pop_front();       // { ?,  'B', 'C',  ? }
}
```

From this code, our deque is in the following state:

0	1	2	3
	B	C	

Notice how `push_back()` adds elements to the end (right) of the array and `pop_back()` removes elements from the right. Furthermore, `push_front()` adds elements to the front (left) of the array and `pop_front()` removes elements from front (left). To state this more formally:

- `push_back()` adds elements to the back (right), thereby moving the back to the *right* by one slot.
- `push_front()` adds elements to the front (left), thereby moving the front to the *left* by one slot.
- `pop_back()` removes an element from the back (right), thereby moving the back to the *left* by one.
- `pop_front()` removes an element from the front (left), thereby moving the front to the *right* by one.

We can represent this state (a `deque` with a capacity of four and a size of two) with two index variables: `iFront` and `iBack` (index of the front and index of the back). Thus `front()` would simply return `array[iFront]` and `back()` would return `array[iBack]`. This would make `pop_back()` little more than `iBack--` and `pop_front()` would be little more than `iFront++`. In the simple case this works, but it should be clear that we might run off the end of our array if this goes unchecked. We must therefore have a version of `iFront` and `iBack` that is normalized much like we did for the queue: `0 <= indexNormalized < d.capacity()`.

Our first approach should be to use the modulus operator as we did in the queue. Consider the case where the capacity is set at 4 and the size set to 1:

0	1	2	3
			X

In this case, `iBack` is actually 7 because `push_back()` was called 8 times (and presumable `pop_front()` was called 7 times). Thus we can find the absolute index of the back element with `iBack % 4`.

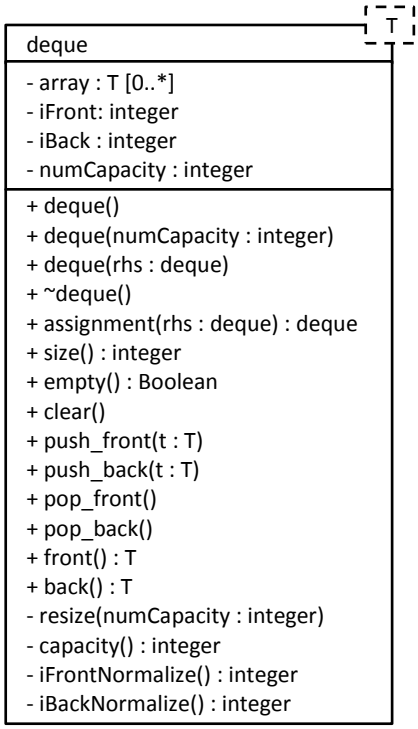
```
int iBackNormalized = iBack % capacity();  // WARNING: this is not a complete solution
```

This works as we expect because 7 (the value of `iBack`) modulus 4 (the capacity of the deque) equals 3.

What would happen if we did not call `push_back()` 7 times but rather called `push_front()` once? The deque would look the same but `iFront` would be -1. If we try to take the modulus of -1, we will get -1 when we should get 3. Clearly something will need to be done to handle this case. We will leave it as an exercise to figure out how to accomplish this. For now, we will assume that two private methods exist in the `deque` class:

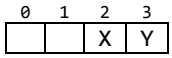
```
int deque <T> :: iFrontNormalized() const;
int deque <T> :: iBackNormalized()  const;
```

Based on these design decisions, we can describe the UML class diagram of our `deque` class.



Size

Size should return the number of elements in the `deque` regardless of how many times the indicies have wrapped to the right or left. Consider the `deque` with two elements in the following state:



Consider the following cases:

Case	Handling
`iFront == 2` `iBack  == 3`	This is the simplest case. We can compute the size with `RETURN iBack - iFront + 1`
`iFront == 6` `iBack  == 7`	Here we have completed one wrap to the right. The equation to compute the size would be the same: `RETURN iBack - iFront + 1` Note that it is impossible that `iFront == 2` beause there would be 6 elements in the deque.
`iFront == -2` `iBack  == -1`	Here we have completed one wrap to the left. Even in this case, the equation is the same. `RETURN iBack - iFront + 1` We need to keep the raw index (that can go negative or go greater than the capacity) instead of the absolute index (which we use to actually access elements in the array) so we can correctly compute the size.

With all the cases covered (no wrapping, wrapping to the right, and wrapping to the left), we can always compute the size by subtracinging `iBack` from `iFront` and adding one.

Notice that the cost of computing the size is always the same regardless of the values of the size of the deque. Thus the cost is O(1).

Push Front & Back

Pushing onto the back of a deque is exactly the same as that of a queue:

```
deque.push_back(e)
  IF size() = capacity()
    resize(capacity() * 2)
  iBack++
  array[iBackNormalize()] ← e
```

Pushing onto the front of the deque is also the same.

```
deque.push_front(e)
  IF size() = capacity()
    resize(capacity() * 2)
  iFront--
  array[iFrontNormalize()] ← e
```

This is also O(1) in the normal case when no allocation is required, O(n) in the worst case when we need to copy the entire array in `resize()`, and O(log n) in the average case because of how often a reallocation must occur.

Pop Front & Back

Popping from the front of a `deque` is also the same as the queue.

```
deque.pop_front()
  IF empty()
    ERROR
  iFront++
```

The back works exacty the same:

```
deque.pop_back()
  IF empty()
    ERROR
  iBack--
```

Since the values of `iFront` and `iBack` do not affect performance, the cost of both is O(1).

Front & Back

Retrieving data from the front of the back of the `deque` is trivial because most of the hard work is done in our `iFrontNormalize()` and `iBackNormalize()` utility functions.

```
deque.front()
  IF empty()
    ERROR
  ELSE
    RETURN array[iFrontNormalize())
```

The `back()` method is the same. These are both O(1) because the size does not affect the cost.

Clear

Consider a `deque` with one element in it:

0	1	2	3
X			

In this case, `iFront == iBack == 0`. If we want this to be result of a single `push_back('X')`, then what must the `deque` look like before the push happened and `size() == 0`? Since `push_back()` results in adding one to `iBack` and `iFront` must be unchanged, that means that the empty state is `iBack == -1`. Thus clearing the `deque` can be accomplished with:

```
deque.clear()
   iFront ← 0
   iBack ← -1
```

Assignment

As with the `queue`, copying a `deque` involves popping elements of the right-hand-side and pushing them into the left-hand-side. It is generally not a good idea to blindly copy the array over directly. This is especially true if the right-hand-side buffer is of a different size than the left-hand-side buffer. Instead, we will "rebuild" the left-hand-side with a succession of `push-back()`s.

```
deque.assignment (lhs, rhs)
  clear()
  IF lhs.capacity() < rhs.size()
    resize(rhs.size())
  FOR i ← rhs.iFront … rhs.iback
    lhs.push_back(rhs.array[iNormalize(i)])
```

There are a couple things to notice about this. First, is the function `iNormalize()`. This is essentually the same thing as `iFrontNormalize()` and `iBackNormalize()` except it takes an index parameter. Thus, we can write `iFrontNormalize()` as:

```
deque.iFrontNormalize()
  RETURN iNormalize(iFront)
```

The next thing to notice is how we take elements from the head of the right-hand-side (`rhs`). As with the `queue`, the new left-hand-side will honor the `deque` contract exactly the same as the right-hand-side `deque`. This does not mean that the two will have the same internal representations. For example consider a deque with two elements. The following are various valid implementations of this deque:

0	1
A	B

… which is the same as …

0	1	2	3
A	B		

… which is the same as …

0	1	2	3
	A	B	

… which is the same as …

0	1	2	3
B			A

To copy the elements from the right-hand-side, we need to "unroll" the elements and "push' them onto the left-hand-side. Thus the `array`, the `iFront`, and `iBack` member variable will be different. The loop dictates the cost to be O(n).

Application

There are many real-world applications of the stack and queue that we find during the course of a typical day. The same cannot be said about the deque. Most uses of the deque are actually stack or queue scenarios where special access is needed at the "wrong" side. A couple examples are a queue where occasionally elements need to be pushed onto the front, or a stack that occasionally runs backwards. One additional example will be provided that makes use of both sides of the deque equally: a palindrome checker.

DMV

In most big cities, the Department of Motor Vehicles (DMV) is the place where motorists go to renew their driver's license, register their cars, and pay tickets. Traditionally, this is also a place with very long lines. Sometimes a patron would wait a half hour in the line just to realize that an important piece of information is not available (such as the license plate number of your car). This requires the patron to retrieve the information before going back to the end of the half hour line. The manager of the DMV realized this was unfair so she instituted a new policy: you get front-of-the-line privileges if you need to step out of line during a transaction. This worked great for many years. One day the manager decided to computerize the entire operation. Realizing that the line management process is essentially a `queue`, she decided to implement her `Line` class with a `queue` as a member variable. Things seemed very straight-forward until she tried to implement her `Line::addExistingPatron()` method. The `queue` explicitly forbids adding elements to the front. She was about to give up when she learned about the `deque`.

```
class Line
{
private:
   deque <Patron> line;                  // mostly a queue but has a push_front()
public:
   Patron serveNext() throw (std::string)
   {
      if (empty())
         throw std::string("empty line");
      Patron next = line.front();        // retrieve the next
      line.pop_front();                  // the patron is no longer at the front
      return next;
   }

   void addNewPatron(const Patron & patron)
   {
      line.push_back(patron);            // new patrons go to the end of the line
   }

   void addExistingPatron(const Patron & patron)
   {
      line.push_front(patron);           // existing patrons to to the head of the line
   }

   bool empty() const
   {
      return line.empty();
   }
};
```

T-Shirt Drawer

Sam is not what one would call a snappy dresser (yes, I said "snappy dresser." Look it up! It used to be a common slang term for "well dressed."). In fact, he is what one would call a jeans-and-t-shirt guy. Every Saturday after he has done his laundry, he dumps all his t-shirts into the t-shirt drawer. Every morning thereafter, he simply takes the top shirt out of the drawer and puts it on. This is how it goes every week. He pretty much cycles through the same six t-shirts even though he has dozens in his over-stuffed t-shirt drawer.

There is only one break to this pattern. Every Friday night, Sam has a date with a girl Sue from his computer class. Wanting to impress her, he puts a fresh t-shirt on just before he heads out the door. However, he does not just put any old t-shirt on, he fishes out the bottom t-shirt from his drawer. This should guarantee that Sue has never seen the t-shirt before.

As Sam slides on a clean BYU-Idaho gym shirt, he begins to wonder. Will Sue only see fresh t-shirts or will he need a different algorithm. With a half hour until his date, he decides to write a simulator to see if a pattern emerges. Since his t-shirt drawer behaves like a `stack`, he starts with that data-structure. However, the `TShirtDrawer::getDateShirt()` method breaks all the `stack` rules! This forces him to use a `deque`.

```
class TShirtDrawer
{
private:
   deqeue <TShirt> drawer;             // mostly a stack except for Friday night

public:
   bool doLaundryRightNow() const
   {
      return drawer.empty();           // if there are no t-shirts in the drawer, panic
   }

   void addCleanShirt(const TShirt & shirt)
   {
      drawer.push_back(shirt);         // add t-shirts onto the top
   }

   TShirt getShirt() throw (std::string)
   {
      if (drawer.empty())
         throw std::string("better wear Sunday cloths today");
      TShirt shirt = drawer.back();
      drawer.pop_back();               // remove t-shirts from the top
      return shirt;
   }

   TShirt getDateShirt() throw (std::string)
   {
      if (drawer.empty())
         throw std::string("wear today's shirt and do laundry!");
      TShirt shirt = drawer.front();
      drawer.pop_front();              // remove the bottom t-shirt for a special night
      return shirt;
   }
};
```

Palindrome

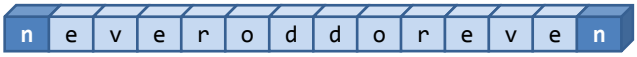
A palindrome is a string of text that reads the same forwards and backwards (not taking into account spaces, punctuation, or casing). Thus "Eve" is a palindrome but "ear" is not. The longest known palindrome in the English language is:

A man, a plan, a canal, Panama.

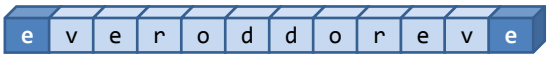
Our job is to write a function to verify if a given string is a palindrome. To do this, we will use a deque:

```
bool isPalindrome(const char * text)
{
   // first, put the text into a deque.  All in lowercase please!
   deque <char> letters;
   for (const char * p = text; *p; p++)
      if (isalpha(*p))
         letters.push_back(tolower(*p));

   // next, check the front letter against the back letter
   while (d.size() > 1)

      // if the front does not match the back, then we are not a palindrome
      if (d.front() != d.back())
         return false;

      // if they do match, then remove the front and back and continue
      else
      {
         d.pop_front();
         d.pop_back();
      }

   // if all of the fronts match the backs, we are a palindrome
   return true;
}
```

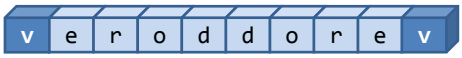
The first part of the program will put the lowercase version of all the letters into a deque. For example, consider "Never odd or even!" The following would be the state of the deque:



From here, we start the second loop. As long as the deque is larger than one element (all one letter strings are palindromes), we will compare the first and last letters. Here they are the same so they will be removed…



Next the first and last letters will be compared again. Since they do match, they will be removed…



This process continues until the deque is either empty, of size one, or a mismatch is found.

Now Serving

Consider a system to simulate students getting help in a computer lab. In order to make best use of the lab assistants' time and to keep the lab operating in an orderly manner, the Now Serving system was developed. This system works as follows: when a student needs help, the student enters a help request into the system. This help request consists of the class (CS235), the student name (Sam), and the number of minutes that will be required (5). This help request is put onto the back of the deque. If there is no student currently being helped by a lab assistant, then the assistant takes the next request off of the front of the deque and assists him or her for the desired number of minutes. Occasionally there is an emergency. A student can submit an emergency help request by preceding his request with two exclamation marks (!!). When this happens, the emergency help request is placed at the head of the deque.

Every minute, the system decrements the time-remaining value on the current help request. If the time-remaining value reaches zero then the next help request (if there is one) is taken from the deque. Also, every minute a new help request can be entered into the system. To see how this works, consider the following:

	Action	Description	Served	Deque
1	`<1> CS101 Sam 3`	Sam needs help in the lab. The first help request is for CS 101 of 3 minutes. His help request will not go in the deque because noone is currently being served.	3 Sam	
2	`<2> none`	No new help requests this minute. This means that Sam's time-left value will be decremented by one. When this reaches 0, then Sam will be removed from the Served spot.	2 Sam	
3	`<3> CS102 Sue 3`	Sue makes a help request for CS102 for 2 minutes. Note how she is now at the head of the deque. She does not displace Sam who is currently being served.	1 Sam	3 Sue ← push(Sue)
4	`<4> CS210 Sly 4`	Sam is finished so the head of the deque (Sue) is now being served. Meanwhile Sly is added to the back of the deque. There is still one person waiting to be helped.	3 Sue ← pop()	4 Sly ← push(Sly)
5	`<5> !! CS101 Abe 1`	Abe makes an emergency help request. This puts him ahead of Sly in the deque. This is where we need a deque: when students need to cut into the front of the line.	2 Sue push(Abe) →	1 Abe, 4 Sly
6	`<6> CS102 Steve 1`	Steve makes normal help request. His request is put at the end of the deque behind Sly. The deque now has three elements totaling six minutes	1 Sue	1 Abe, 4 Sly, 1 Steve ← push()
6	`<7> none`	No new help requests. Sue is finished so Abe is popped off of the deque. Now Sly is next up. If we continued the simulation, then Sly would be next to be helped.	1 Abe ← pop()	4 Sly, 1 Steve

Examples

Example 04.1

What is the state of the queue after the following code is executed:

```
{
   deque <int> d;        // #1

   q.push_back(7);       // #2
   q.push_back(8);       // #3
   q.push_front(6);      // #4
   q.push_front(5);      // #5
   q.push_back(9);       // #6

   q.pop_front();        // #7
   q.pop_back();         // #8
}
```

Answer:

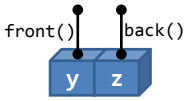
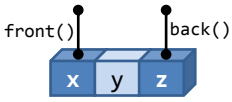
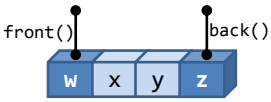
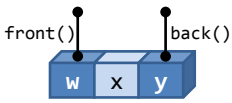
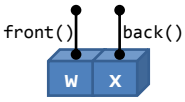
#1		Initially empty.
#2	iBack iFront 7	Capacity goes to 1 and the first slot is filled. Both `iFrontNormalized()` and `iBackNormalized()` are assigned to the same element.
#3	iFront iBack 7 8	Capacity doubles. This means that a new buffer is created and `7` is placed in the `0` slot. After that point, then the `push_back(8)` occurs.
#4	iBack iFront 7 8 6	Capacity doubles again to 4. First we bring over 7 and 8 into the 0 and 1 slot respectively. Then we add 6 onto the front. Note that the front is to the left of 7.
#5	iBack iFront 7 8 5 6	Now the deque is filled. Observe how the new element added to the front will be to the left of 6.
#6	iFront iBack 5 6 7 8 9	When we attempt to push onto a full deque, we must reallocate. This will "unwrap" the array and put the front in slot 0. After this is completed, then we will place 9 on the back of the deque.
#7	iFront iBack 6 7 8 9	With a `pop_front()`, the `iFront` index moves to the right by one slot.
#8	iFront iBack 6 7 8	With a `pop_back()`, the `iBack` moves to the left by one slot. The size is now 3. Notice how the deque looks very different here than it does in step #4 even though they both appear the same to the client.

Example 04.2

What is the state of the queue after the following code is executed:

```
{
   deque <char> d;        // #1

   q.push_front('z');     // #2
   q.push_front('y');     // #3
   q.push_front('x');     // #4
   q.push_front('w');     // #5

   q.pop_back();          // #6
   q.pop_back();          // #7
}
```

Answer:

#1		Initially empty.
#2	iBack iFront z	When the deque is empty, then `push_back()` and `push_front()` behave exactly the same.
#3	iBack iFront z y	Capacity doubles and z is copied to the new buffer. After that point, then y is added to the front. The front is the left of `z` but that wraps around to slot 1. The client view is: 
#4	iBack iFront y z x	As the capacity doubles again, the data in the previous buffer is "unwound." After that point, `x` is placed to the left of `y` which happens to be at the end of the deque's array. 
#5	iBack iFront y z w x	When we push w onto the front, it goes to the left o the previous front. The client view is: 
#6	iBack iFront y w x	When we pop off the back of the deque, the back goes to the left one slot. In this case, the back is now slot 0. 
#7	iFront iBack w x	Another pop off the back. Again, the back goes to the left one slot. Since it was at the beginning of the array in the previous step, it is now at the end of the array. 

Exercises

Exercise 04.1

From memory, recite and define the operations associated with the deque ADT. Only provide the read-write version of the front and back methods.

Answers:

_____

_____

_____

_____

_____

_____

Please see page 98 for a hint.

Exercise 04.2

Write a function to convert a relative index (that could be negative or much greater than `numCapacity`) into a normalized or absolute value (one guaranteed to honor `0` ≤ `iNormalize(i)` ≤ `numCapacity`).

```
template <class T>
int deque <T> :: iNormalize(int i) const;
```

The following table may help with this process:

i	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5
normalized	2	3	0	1	2	3	0	1	2	3	0	1

Answer:

Please see page 102 for a hint.

Exercise 04.3

Write out the state of deque's member variables after each line or group of lines are executed. This includes the four member variables (`num` for `numCapacity`, `iFront`, `iBack`, and `data`) as well as the state of the stack (drawn with boxes)

```
{
   deque <int> d;

   d.push_back(50);

   d.push_back(89);

   d.push_back(93);

   d.pop_back();

   d.pop_back();
}
```

num	iFront	iBack	data	state

Please see page 99 for a hint.

Exercise 04.4

Write out the state of deque's member variables after each line or group of lines are executed. This includes the four member variables (`num` for `numCapacity`, `iFront`, `iBack`, and `data`) as well as the state of the stack (drawn with boxes).

```
{
   deque <int> d;
   d.push_front(31);

   d.push_front(28);

   d.push_front(19);

   d.push_back(49);

   d.push_back(55);

   d.push_back(64);

   d.pop_front();

   d.pop_back();

   d.pop_front();

   d.pop_front();
   d.pop_front();
}
```

num	iFront	iBack	data	state

Please see page 110 for a hint.

Exercise 04.5

Write out the state of deque's member variables after each line or group of lines are executed. This includes the four member variables (`num` for `numCapacity`, `iFront`, `iBack`, and `data`) as well as the state of the stack (drawn with boxes).

```
{
   deque <int> d;
   d.push_back(38);
   d.push_back(49);
   d.push_back(55);

   d.push_front(26);

   d.push_front(11);

   d.push_front(3);
}
```

num	iFront	iBack	data	state

Please see page 110 for a hint.

Exercise 04.6

Write out the state of deque's member variables after each line or group of lines are executed. This includes the four member variables (`num` for `numCapacity`, `iFront`, `iBack`, and `data`) as well as the state of the stack (drawn with boxes).

```
{
   deque <int> d;
   d.push_back(99);
   d.push_front(67);

   d.push_front(26);

   d.pop_back();

   d.push_back(73);

   d.push_front(11);

   d.back() = 92;

   d.front() = 19;

   d.pop_back();

   d.push_front(12);

   d.push_back(99);

   d.push_front(3);
}
```

num	iFront	iBack	data	state

Please see page 110 for a hint.

Exercise 04.7

Write out the state of deque's member variables after each line or group of lines are executed while running the Now Serving program from the Application.

	num	iFront	iBack	served	data
`<0> CS124 Sam 4`					
`<1> CS165 Sue 2`					
`<2> none`					
`<3> none`					
`<4> none`					
`<5> none`					
`<6> none`					
`<7> finished`					

Please see page 109 for a hint.

Exercise 04.8

Write out the state of deque's member variables after each line or group of lines are executed while running the Now Serving program from the Application.

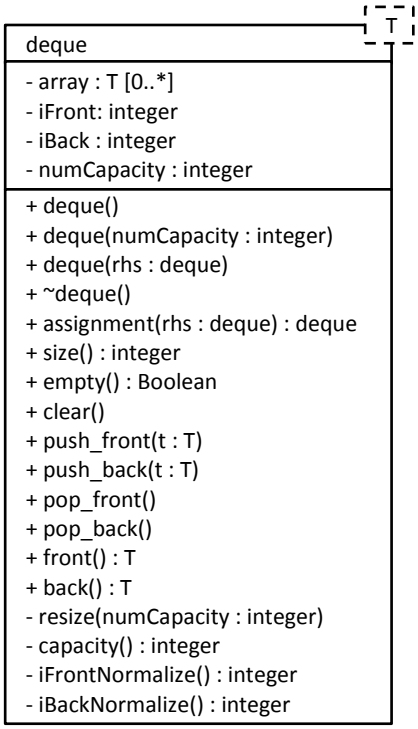
	num	iFront	iBack	served	data
`<0> CS101 A 4`					
`<1> !! CS102 B 1`					
`<2> !! CS103 C 1`					
`<3> CS104 D 1`					
`<4> none`					
`<5> none`					
`<6> none`					
`<7> none`					
`<8> finished`					

Please see page 109 for a hint.

Problems

Problem 04.1

Implement your own version of the `std::deque` data structure. Call your implementation `custom::deque`. In other words, place your class in the `custom` namespace and implement all of the interfaces described in this chapter. The UML class diagram of our `deque` class is the following:



Please see page 102 for a hint.

Problem 04.2

Using the STL `deque` or using your own implementation of `deque` from Problem 04.1, write a program to simulate a system to assist students getting help in a computer lab. This is explained in detail in the Appliation section of the chapter.

Please see page 109 for a hint.

Sue is playing a card game with her grandfather. His favorite game is Rummy where each player tries to get runs (4-5-6) or collections (King, King, King). With this game, the order of the cards in the hand does not matter; only the composition of the hand makes a difference. This illustrates the data structure called a set.

It is a common programming need to keep track of a collection of things. In this case, we do not need to keep track of the order of the collection, but rather what is contained in the collection. Consider, for example, a collector of baseball cards. The collector cares less about *the order* in which the cards were acquired, rather only *what* has been acquired. The defining characteristics of a set:

- **Container**: The set is designed to store a collection of data.
- **Sequential and Search Access**: Elements can be retrieved by searching or by iterating.
- **Order Arbitrary**: The order elements are stored depends on how the ADT is implemented.
- **Expandable**: The capacity of the set can be increased to meet the needs of the client.
- **Blind Insertion**: The client is not informed of where items are inserted into the set.

A set is an ADT representing a collection of unique elements. Mathematically, we define these collections with an uppercase letter such as S. A set consists of a collection of elements. To represent the elements in a set, use the curly braces. For example, the set of even natural numbers less than 10 are:

$$N = \{2, 4, 6, 8\}$$

Recall that for sets the order does not matter. Therefore the following sets are equivalent:

$$\{2, 4, 6, 8\} = \{4, 2, 8, 6\} = \{8, 6, 4, 2\}$$

Another distinguishing property of a set is that redundant or duplicate values are ignored. Therefore the following are equivalent:

$$\{2, 4, 6, 8\} = \{2, 2, 2, 2, 4, 6, 8\} = \{8, 6, 6, 6, 6, 4, 2\}$$

The cardinality of a set refers to the number of elements in the set. Therefore, the cardinality of the following sets are the same: 3

$$\{1, 2, 3\}$$
$$\{\mathrm{a}, \mathrm{a}, \mathrm{a}, \mathrm{a}, \mathrm{b}, \mathrm{c}\}$$
$$\{Sam, Sue, Stan\}$$

The empty set, of course, is the set with a cardinality of 0. This is represented by the empty-set symbol $\varnothing$ or by the empty curly braces $\{\}$.

A few sets that you have undoubtably encountered include:

Name	Description	Example
Boolean	The set of all bools, a finite set of cardinality 2	$B = \{ true, false \}$
Lowercase Letters	The set of all lowercase letters, a finite set of cardinality 26.	$L = \{ a, b, c, d, \ldots z \}$
Whole Numbers	The set of all whole numbers, an infinite set or a set with infinite cardinality.	$W = \{ 0, 1, 2, 3, 4, \ldots \}$
Playing Cards	The set of cards in a poker hand, a finite set of cardinality 5.	$H = \left\{ \begin{array}{l} AceSpaces, KingSpades, \\ QueenSpades, JackSpades, \\ 10Spades \end{array} \right\}$

Perhaps the final example is most illustrative of sets. A royal flush is a royal flush, regardless of the order in which the cards are arranged.

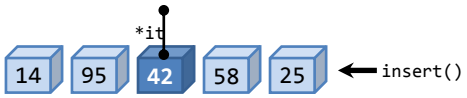
Definition

A set is an ADT that stores a collection of values with no repeats. The client of the set typically does not get to specify the order in which things are stored, but most often an order is preserved for efficiency reasons. A set is formally defined as:

Name	Example	Description
Create	`set <int> s;`	Create a new set.
Assignment	`s1 = s2;`	Make a copy of a set.
Empty	`f = s.empty();`	Returns `true` if a set has no elements, `false` otherwise.
Size	`n = s.size();`	Returns the number of elements currently in a set.
Capacity	`c = s.capacity();`	Returns the number of elements a set will hold before reallocating.
Clear	`s.clear();`	Remove all elements from a set.
Begin	`it = s.begin();`	Returns an iterator to the beginning of a set.
End	`it = s.end();`	Returns an iterator referring to the past-the-end element in a set.
Insert	`s.insert(e);`	Adds an element to a set. The size of the set will increase by one.
Find	`it = s.find(e);`	Returns an iterator pointing to the element if it exists in a set.
Erase	`s.erase(it);`	Removes an element from a set as referred to by the passed iterator.
Union	`s3 = s1 || s2;`	Returns a new set representing the elements in `s1` combined with those in `s2`.
Intersection	`s3 = s1 && s2;`	Returns a new set representing the elements in both `s1` and `s2`.
Difference	`s3 = s1 - s2;`	Returns a new set representing those in `s1` after `s2`'s elements are removed.

The most common operations one would want to perform on a set include finding, inserting, and deleting elements. Additional set operations include union, intersection, difference, and complement. These are not in the standard template library so they are listed below the second thick black line. Note that most other set libraries include union, intersection, and difference.

The set can be described by the following figure:



There are several things to observe about this figure. First is that the order of elements in the set are not preserved for the client. This is an implementation detail completely up to those who implement the `Set` class. Furthermore there is no guarantee that any two elements are adjacent to each other in memory.

Second, there is no way to access any element in the set except through an iterator. This iterator can be obtained through the `find()` method which will locate a specific element in the set. An iterator can also be obtained through the `begin()` and `end()` methods, allowing the client to reach all the elements in the set.

The final thing to notice is that the `insert()` method is how one adds an element to the set. Unlike `push()`, `push_back()` or `push_front()`, there is no guarantee where in the set the new element will reside. Again, this is an implementation detail that the client should not rely upon.

Insert

The first common operation to perform on a set is to add or insert an element. For now, we will use the plus symbol (+) to denote insert. Since sets do not contain duplicate elements, the first step in any insert operation is to check for the existence of the element in the set:

$$S.insert(x) = \begin{cases} S, & if\ x \in S \\ S + x, & if\ x \notin S \end{cases}$$

Consider the set of all whole numbers below 5 (W):

$$W = \{\, 0, 1, 2, 3, 4 \,\}$$

First we will attempt to add the number 3 to the set. Since the number 3 is already in the set, W remains unchanged:

$$W + 3 = \{\, 0, 1, 2, 3, 4 \,\}$$

Next, we will add the number 10 to the set. Since 10 is not currently in the set, the set will grow by one.

$$W + 10 = \{\, 0, 1, 2, 3, 4, 10 \,\}$$

The plus symbol is not actually (+) the standard symbol for inserting an element into a set; there is no symbol in fact. Most often, the Union operator (∪) is used to denote adding an element (discussed on page 124).

The prototype of the `insert()` method is:

```
void set <T> :: insert(const T & t) throw (const char *);
```

Notice how the method will throw an exception if memory is not available to accommodate the new element.

	Code	Description	State
1	`set <int> s;`	An empty set is created. There is nothing allocated and nothing used.	
2	`s.insert(2);`	We will add an element to the set.	2 ← insert(2)
3	`s.insert(4);`	A second element is added. We cannot depend on the order, but we can expect it to be somewhere.	2 4 ← insert(4)
4	`s.insert(6);`	Another element is added. Again, we cannot be sure where in the set the element is added.	2 4 6 ← insert(6)
5	`s.insert(4);`	Because `4` is already in `s`, the set remains unchanged.	2 4 6 ← insert(4)

Find

Another common operation performed on the set is to inquire whether a given element belongs to the set or if the element does not belong. The mathematical way to represent the "belongs to" notion is the "element of" symbol (∈). For example:

$$6 \in \{2, 4, 6\} \qquad 7 \notin \{2, 4, 6\}$$

We can thus mathematically define the find function for a given set S with:

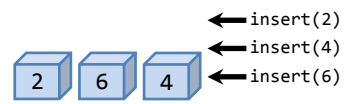
$$S.find(x) = \begin{cases} true, & if\ x \in S \\ false, & if\ x \notin S \end{cases}$$

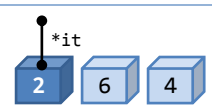
With C++, this will be done differently. We will return an iterator pointing to the element if the value is found in the set and we will return an iterator to `end()` if the value is not found.

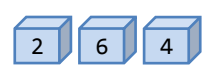
```
set <T> :: iterator set <T> :: find(const T & t);
```

Notice how the iterator must belong to the same template as the set itself.

	Code	Description	State

1	`set <int> s;`
	`s.insert(2);`
	`s.insert(4);`
	`s.insert(6);`
		Create an empty set and fill the values with 2, 4, 6. Note that the order the values are stored in the set depends on the implementation.	

2	`set <int> :: iterator it;`
	`it = s.find(2);`
	`assert(it != s.end());`
	`assert(*it == 2);`
		Now `it` points to the value 2.	

3	`it = s.find(1);` `assert(it == s.end());`	Since 1 is not in `s`, we expect `it` to be the same as `s.end()`. In other words, 1 was not found.	

Notice how the `find()` method is not const. This is because the resulting iterator is not const. In other words, we can change the state of the set through the returned iterator. For this reason, a second version of the `find()` method is often included:

```
set <T> :: const_iterator set <T> :: find(const T & t) const;
```

This version will be called when the underlying set is const. In other words, when working with a constant object, we only have access to methods that are const.

```
template <class T>
bool isValueInSet(const set <T> & s, const T & value)
{
   // we must use a const iterator because we cannot change 's'
   set <T> :: const_iterator it;

   // the const version of find() is used here.
   it = s.find(value);

   // check against the const version of end().
   return (it != s.cend());
}
```

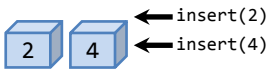
Erase

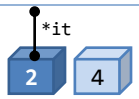
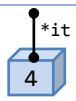
An element can be removed from the set with the `erase()` method.

```
set <t> :: iterator  set <T> :: erase(set <T> :: iterator & it);
```

Note that erasing an element that does not exist in the set has no effect.

	Code	Description	State

1	`set <int> s;`
	`s.insert(2);`
	`s.insert(4);`
		Create an empty set and fill the values with 2, 4	

2	`set <int> :: iterator it;` `it = s.find(2);`	Create an iterator and have it point to 2.	
3	`s.erase(it);`	Erase the element that `it` is pointing to. In this case, it is the element 2. The iterator will now point to the next element.	

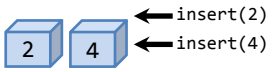
4	`it = s.find(5);`
	`assert(it == s.end());`
	`s.erase(it);`
		The value 5 is not found in the set so it will point to `end()`. If we attempt to erase from here, nothing will happen.	

There is a second version of erase: one that takes a value as a parameter rather than an iterator.

```
set <t> :: iterator  set <T> :: erase(const T & value);
```

As with the iterator version, this does nothing if the value is not found in the set.

	Code	Description	State

1	`set <int> s;`
	`s.insert(2);`
	`s.insert(4);`
		Create an empty set and fill the values with 2, 4	

2	`s.erase(3);`	Do nothing because the value '3' is not present in the set..	
3	`s.erase(2);`	The value 2 is found so we will remove the element from the set.	

Note that this second version is necessairely slower than the first version because it must first find the element before it can be erased.

Set Iterator

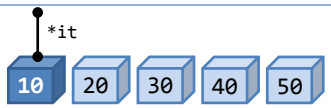
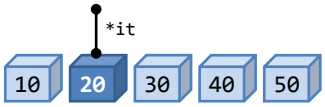
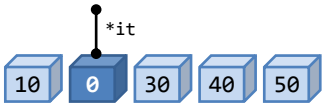
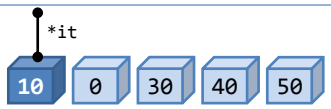
Recall that we had two ways to retrieve elements from the vector: using the array-index operator and using the iterator. Recall we used `stack :: top()` to retrieve elements from the end of a stack, `queue :: head()` to retrieve elements from the front of a queue, and `deque :: front()` & `deque :: back()` to retrieve elements from either end of a deque. Because the set is an unordered collection of elements, we cannot use `top()`, `head()`, `front()`, `back()` or the array-index operator to retrieve elements. Thus, aside from `find()`, the only way to retrieve elements from a set is through an iterator. This will work exactly the same as with the `vector` iterator:

```
for (set <int> :: iterator it = s.begin(); it != s.end(); ++it)
   cout << *it << endl;
```

An example of how this works with a set of integers is the following:

	Code	Description	State

1	`set <int> s;`
	`for (int i = 10; i <= 50; i += 10)`
	`  s.insert(i);`
		Create a set with five elements.	

2	`set <int> :: iterator it;`	Declare a set iterator.	
3	`it = s.begin();`	Point the iterator to the first element of the set.	
4	`++it;`	Advance the iterator to the next element in the set.	
5	`*it = 0;`	Modify the element with the dereference operator.	
6	`it--;`	The iterator should be bi-directional, working both forward and backwards.	

Of course the set class itself will need to support the `begin()` and `end()` method:

```
set <T> iterator set <T> :: begin();
set <T> iterator set <T> :: end();
```

As with vector, there are several variations:

```
set <T> :: iterator               <int> it = s.begin();
set <T> :: const_iterator         <int> it = s.begin()   const;
set <T> :: const_iterator         <int> it = s.cbegin()  const;
set <T> :: reverse_iterator       <int> it = s.rbegin();
set <T> :: const_reverse_iterator <int> it = s.rbegin()  const;
set <T> :: const_reverse_iterator <int> it = s.crbegin() const;
```

Union

One less-common operation associated with sets is union (∪). Union is a binary operation taking a set on the left-hand-side (`lhs`) and the right-hand-side (`rhs`). The result of this operation is a new set consisting of the elements that are either in `lhs` or `rhs`. Consider the two sets: S_1 and S_2:

$$S_1 = \{\, 0, 2, 4, 6, 8 \,\} \qquad S_2 = \{\, 4, 5, 6 \,\}$$
$$S_1 \;\cup\; S_2 = \{\, 0, 2, 4, 5, 6, 8 \,\}$$

Observe how the elements present in both S_1 and S_2 ($\{4,\ 6\}$) are not duplicated in $S_1 \cup S_2$ because sets do not have duplicate elements. Union is commonly used when two or more sets are to be combined.

In C++, we denote the union operation with the OR operator `||`.

```
set <T> set <T> :: operator || (const set <T> & rhs) const throw (const char *);
```

Observe how we return a new set by-value. Also, notice how `rhs` does not change nor does `this`.

	Code	Description	State

1	`set <int> s1;`
	`for (int i = 0; i <= 8; i += 2)`
	`  s1.insert(i);`
		Create a set with five elements.	0 2 4 6 8

2	`set <int> s2;`
	`for (int i = 4; i <= 6; i++)`
	`  s2.insert(i);`
		Create another set with three elements.	4 5 6

3	`set <int> s3 = s1 || s2;`	Our third set is the union of the first two.	0 2 4 5 6 8

Note that the union operator works the same as if a new set were created by inserting all the elements in the left-hand-side and then inserting all the elements in the right-hand-side.

Intersection

Set intersection (∩) is defined as the set of elements that two sets share in common. Consider the two sets S_1 and S_2:

$$S_1 = \{\, 0, 2, 4, 6, 8 \,\} \quad S_2 = \{\, 4, 5, 6 \,\}$$
$$S_1 \;\cap\; S_2 = \{\, 4, 6 \,\}$$

In C++, we denote the intersection operation with the AND operator `&&`.

```
set <T> set <T> :: operator && (const set <T> & rhs) const throw (const char *);
```

	Code	Description	State

1	`set <int> s1;`
	`for (int i = 0; i <= 8; i += 2)`
	`  s1.insert(i);`
	`set <int> s2;`
	`for (int i = 4; i <= 6; i++)`
	`  s2.insert(i);`
		Create a set with five elements.
		
		Create another set with three elements.
			0 2 4 6 8
			
			4 5 6

2	`set <int> s3 = s1 && s2;`	Our third set is the intersection of the first two.	4 6

Difference

The difference operator returns the members that are in one set but not in another. Consider set S_1 and S_2:

$$S_1 = \{0, 2, 4, 6, 8\} \quad S_2 = \{4, 5, 6\}$$

$$S_1 - S_2 = \{0, 2, 8\}$$

Unlike the union and intersection operators, the commutative property does not apply to the difference operator. In other words, $S_1 - S_2 \neq S_2 - S_1$. Thus:

$$S_2 - S_1 = \{5\}$$

In C++, we use the subtraction operator to perform set deletion.

```
set <T> set <T> :: operator - (const set <T> & rhs) const throw (const char *);
```

	Code	Description	State

1	`Set <int> s1;`
	`for (int i = 0; i <= 8; i += 2)`
	`  s1.insert(i);`
	`set <int> s2;`
	`for (int i = 4; i <= 6; i++)`
	`  s2.insert(i);`
		Create a set with five elements.
		
		Create another set with three elements.
			0 2 4 6 8
			
			4 5 6

2	`set <int> s3 = s1 - s2;`	Our third set is the difference of the first two.	0 2 8
3	`set <int> s4 = s2 - s1;`	Notice how `s2` and `s1` are reversed, yielding different sets.	5

First we will consider `{ 0, 2, 4, 6, 8 } - { 4, 5, 6 }`. The end result will be everything that is in `s1` but is not in `s2`. In other words, we will start with `{ 0, 2, 4, 6, 8 }` and remove `4`, then remove `5`, then remove `6`. Since `4` and `6` are in `s1`, those numbers will be erased. Since `5` is not, nothing will change. The end result is the set `s3` which contains `{ 0, 2, 8 }`.

We will follow exactly the same process with `{ 4, 5, 6 } - { 0, 2, 4, 6, 8 }`. The resulting set will be `s2` with `0`, `2`, `4`, `6`, and `8` erased in succession. Note that erasing `0`, `2` and `8` will have no effect because they are not present in `s2`. However, erasing `4` and `6` will yield `{ 5 }`.

Complement

The final set operator is complement operator. Perhaps this is best explained by example. Consider the set of all possible dice values (the cardinality is 6 of course) and a set containing the two highest values:

$$D_{all} = \{1, 2, 3, 4, 5, 6\} \qquad D_{5,6} = \{5, 6\}$$

The complement of $D_{5,6}$ is all the elements of D_{all} except those in $D_{5,6}$:

$$-D_{5,6} = \{1, 2, 3, 4\}$$

Note that the complement is similar to the difference operator with one exception: the complement operator is unary (taking a single operand) while the difference operator is binary. You can define the complement as the difference between the set of all elements and the operand:

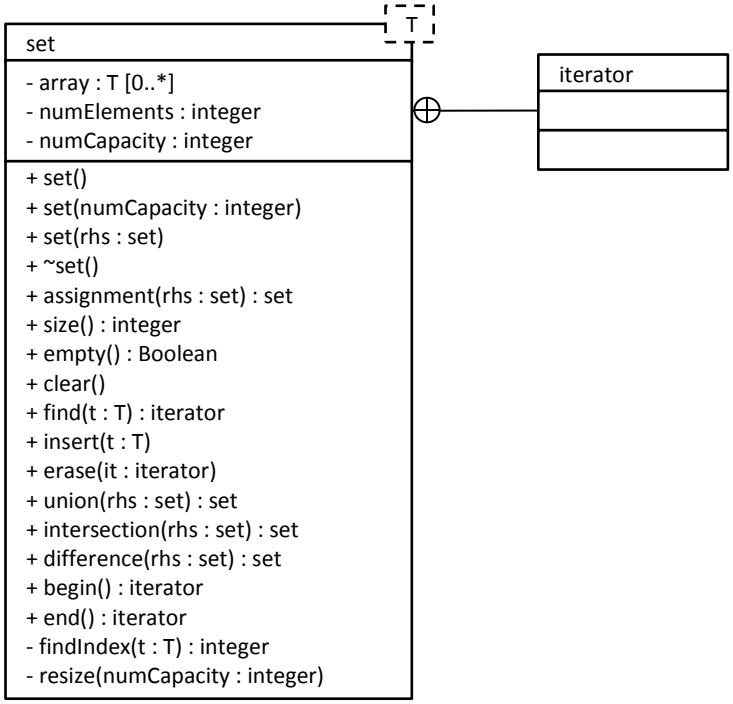
$$D_{all} - D_{5,6} = -D_{5,6}$$

In most situations, we think about the complement operator in the context of finite sets. For example, if you were to play Go Fish!, you must consider the set of all possible cards. The complement of the cards in your hand are the possible cards in your opponent's hand.

Implementation

Up to this point, we have made no references as to our implementation choices for a set. As long as the contract of the set is honored, the client (or user of the set) should not care about the implementation. This is a fundamental aspect of encapsulation of course!

Perhaps the simplest implementation of sets is to use arrays as the underlying data structure. Here, we have two choices: maintain a sorted array or leave the array unsorted. Since the definition of a set is "an unordered collection of unique objects," any order is equally valid. It thus behooves us to choose an order that is most convenient for implementation and most efficient. Right now, it is not readily apparent which implementation is best. We will therefore explore both an unsorted implementation and a sorted implementation. Note that there is a third option for implementing the `set`. This option is beyond the scope of this chapter because it involves a data structure we have not yet learned about: the Binary Search Tree (BST). We will explore a BST implementation of set in Chapter 09 : Binary Search Tree.



In the above UML class diagram, notice how the `iterator` is associated with the `set` class. The ⊕ symbol indicates that `iterator` is a nested class to `set`. This means that `iterator` is defined within `set`. Please see Appendix D. Review: UML Class Diagram, Appendix G. Review: Iterator, and Appendix H. Review: Nested Class for details as to how this works.

Notice how set is a template class. This means we need to specify a data-type when instantiating a `set`. Notice how `iterator` is not. We do not specify a data-type when declaring an `iterator`, but we do need to specify a data-type when declaring a `set`:

```
{
   // declare a set of integers
   // we specify the data-type in the <>s
   set <int> s;

   // declare an iterator for a set of integers
   // we specify the data-type for the set, but not for the iterator
   set <int> :: iterator it;
}
```

Find

The `find()` algorithm will be quite different between the sorted and unsorted implementation. With the unsorted array, the elements in the array could be in any order. This forces us to use a linear search. With the sorted implementation, we can take advantage of the improved speed of a binary search.

Unsorted: A linear search involves looping through every element in the array, comparing each with the searched element in turn. The pseudocode for this algorithm is:

```
set.find (element)
   FOR i ← 1 … numElements
      IF element = array[i]
         RETURN i
   RETURN numElements
```

Notice the FOR loop. In the best case, we will encounter a matching element at the beginning of the array. In the worst case, we will not find a matching element and have to loop through the entire array. In the common case, the larger the cardinality of the set, the more iterations through the loop. We call this linear because the cost of searching for an element in the set is directly related to the size of the array. If the array size doubles, then it will take twice as long on average to find a given element.

Sorted: A binary search involves keeping track of the beginning and the end of the range of indices in which the searched element may reside. With each iteration, the indices are moved closer to each other because half of the range is eliminated. The pseudocode for this algorithm is:

```
set.find (element)
   iBegin ← 0
   iEnd ← numElements – 1

   WHILE iBegin ≤ iEnd
      iMiddle ← (iBegin + iEnd) / 2
      IF element = array[iMiddle]
         RETURN iMiddle
      IF element < array[iMiddle]
         iEnd ← iMiddle – 1
      ELSE
         iBegin ← iMiddle + 1
   RETURN numElements
```

Though this algorithm is considerably more complex, it is much faster. The best case exists when the element is in the middle of the list. The worst case is when the element is not found. In those cases, we will need to search $\log_2 n$ elements. How much better is $\log_2 n$ than linear? If $n = 1{,}000$, then $\log_2 n = 10$. If $n = 1{,}000{,}000$, then $\log_2 n = 20$. Thus, $\log_2 n$ grows much slower than linear as n gets larger. Slower growth means faster execution time.

One final thought. Since both `find()` and `insert()` both require knowledge of where the element resides in the array, both need to use a similar binary search algorithm to look for the element. The difference is that the insert() method needs to know where the element will reside if it is not found. We will call this shared version `findIndex()`:

```
int set <T> :: findIndex(const T & element) const;
```

Insert

Recall that the `insert()` method adds an element to the set if the element is not currently found in the set. This means that two problems need to be solved: a means must be defined for determining if the element is in the set, and a means must be defined for inserting the element into the set.

Unsorted: We will start with a set implementation done with an unsorted array. Since the only way to find an element in an unsorted array is to use a linear search, we must write a loop to pass through every element in the array. If the element is found, nothing is done. Otherwise the new element is inserted at the end of the array. The pseudocode for this algorithm is:

```
set.insert(element)
   FOR i ← 1 … numElements
      IF element = array[i]
         RETURN
   array[numElements++] ← element
```

The cost of inserting an element into an unsorted set is the same as finding an element in an unsorted set: linear. Might this be faster if we use a sorted array?

Sorted: When inserting an element in a sorted array set, there are two steps: first find the location where the element should go, then shift the remaining elements in the array over by one. To do this, assume that our `find()` function above returned an index to the spot in the array where the element would go if found. In this case, the insert pseudocode would be:

```
set.insert(element)
    iInsert ← findIndex(element)
    if array[iInsert] ≠ element
       FOR i ← numElements … iInsert by -1's
          array[i + 1] ← array[i]
       array[iInsert] ← element
       numElements++
```

The best-case is $\log_2 n$ which occurs when the element already exists in the list. The cost for this comes from the binary-search in the find() function. This is considerably faster than the linear case in the un-sorted implementation. The worst-cast occurs when the element is not found and we need to shift the rest of the array. In the case that the spot where element is inserted is at the beginning of the list (iInsert = 0), then the entire array must be shifted. Thus the cost is $\log_2 n + n$. Because n is so much larger than $\log_2 n$, this is still considered a linear algorithm.

Erase

To erase an element from a set, two problems must be solved: the location must be found of the element to be removed, and filling in the spot with another element in the set.

Unsorted: To erase an element from a set implemented with an un-sorted array, it is necessary to first find the element to be erased and then to shift all the elements over by one slot. This requires two linear algorithms: one to find the element and the second to shift the rest of the array. Note that we can do an interesting optimization. Instead of shifting the array, we can simply move the last element in the set to the slot formerly occupied by the deleted element.

```
set.erase(element)
   FOR i ← 1 … numElements
      IF element = array[i]
         array[i] ← array[--numElements]
         RETURN
```

On average, we will have to loop through half the elements in the array before the desired element is found. The best case is when the first element in the set is to be erased, the worst is when the element is not found. Since the cost increases linearly with the number of elements in the set, the cost is linear.

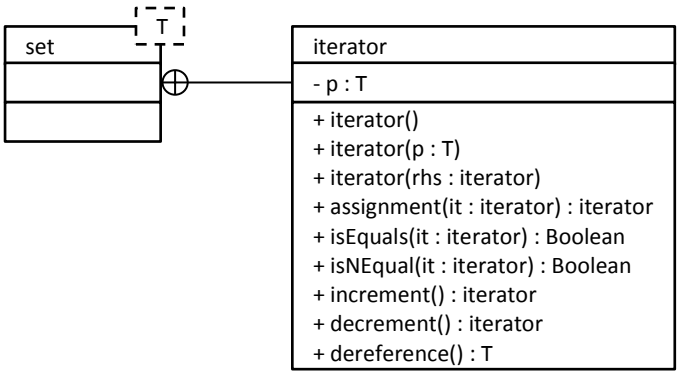
Sorted: These same basic operations must be performed when we have a sorted array implementation. First we locate the element to be erased using our $\log_2 n$ `find()` function, then shift the rest of the list over by one spot. In many ways, the algorithm is similar to that of our insert function:

```
set.erase(element)
    iErase ← find(element)
    IF array[iErase] = element
       FOR i ← iErase … numElements
          array[i] ← array[i + 1]
       numElements--
```

This algorithm is also linear. In the best case, we fail to find the element in $\log_2 n$ time and do not need to modify the list. In the worst case, we find the element in $\log_2 n$ time and then need to shift the entire list (linear).

Set Iterator

The set iterator is built almost exactly the same way as the vector iterator. This should not be surprising: the both the vector and the set iterator use an array as the underlying data structure and both honor the same iterator interface. The UML class diagram of the set iterator is the following:



As with the `vector` class, the `set` class is a template but the iterator is not. The `iterator` is nested in `set`.

Union

The union operator involves creating a new set that consists of all the elements in either set1 or set2.

Unsorted: There are several ways to implement the union operator. The first is to copy the first set then insert the elements in the second set one-by-one. The algorithm is:

```
set.union(setLhs, setRhs)
    setReturn ← setLhs
    FOR iRhs ← 0 … setRhs.numElements
       setReturn.insert(setRhs.array[iRhs])
    RETURN setReturn
```

This algorithm looks very simple, but is actually quite expensive. The first operation (setReturn = setLhs) will require a loop to pass through all the elements in setLhs and copy them to setReturn. We did not discuss this algorithm, but at best it is done in linear time with respect to setLhs.num_elements. After this, we need to pass through each element in setRhs and, for each element, insert it into the setReturn. Recall that each call to insert is linear. Since this needs to be done n times, the cost is n^2. How expensive is n^2? If we have 100 elements in the set, it will take on the order of 10,000 iterations. If we have 1,000 elements, then a million iterations will be performed. In other words, this is quite expensive!

Sorted: A more efficient algorithm can be followed if we use the sorted array implementation. We will start with an index referring to the head of each array (setLhs.array and setRhs.array). Since both arrays are sorted, we know the lower of the first element from setReturn must be in setLhs.array[0] or setRhs.array[0]. This gives us three cases:

- setLhs.array[iLhs] == setRhs.array[iRhs]: insert one of them to the beginning of setReturn. Since they are the same, it does not matter which you choose. After it is inserted, then advance the index in both setLhs and setRhs with (iLhs++ and iRhs++).
- setLhs.array[iLhs] < setRhs.array[iRhs]: insert setLhs.array[iLhs] into setReturn and advance the index for setLhs with (iLhs++).
- setLhs.array[iLhs] > setRhs.array[iRhs]: insert setRhs.array[iRhs] into setReturn and advance the index for setRhs with (iRhs++).

This process is repeated until all the elements from setLhs and setRhs have been inserted into setReturn. The pseudocode is:

```
set.union(setLhs, setRhs)
   iLhs ← 0
   iRhs ← 0
   WHILE  iLhs < setLhs.numElements OR iRhs < setRhs.numElements
      IF iLhs = setLhs.numElements
         setReturn.addToEnd(setRhs.array[iRhs++])
      ELSE IF iRhs = setRhs.numElements
         setReturn.addToEnd(setLhs.array[iLhs++])
      ELSE IF setLhs.array[iLhs] = setRhs.array[iRhs]
         setReturn.addToEnd(setLhs.array[iLhs])
         iLhs++
         iRhs++
      ELSE IF setLhs.array[iLhs] < setRhs.array[iRhs]
         setReturn.addToEnd(setLhs.array[iLhs++])
      ELSE
         setReturn.addToEnd(setRhs.array[iRhs++])
   RETURN setReturn
```

While this is quite a bit more complex than the earlier implementation, it is also quite a bit more efficient. Since every element in setLhs and setRhs is viewed exactly once, the total cost of the function is $(n + m)$ where $n =$ setLhs.numElements and $m =$ setRhs.numElements. This is also linear.

Intersection

Implementation of the intersection operator is much like that of the union. In both cases, each element of the first set must be compared against each element of the second.

Unsorted: As with the union operator, there are several ways to implement the intersection operator. Perhaps the simplest involves searching for each member of setLhs in setRhs. If it exists, then insert it into setReturn:

```
set.intersection(setLhs, setRhs)
   FOR iLhs ← 0 … setLhs.numElements
      IF setRhs.find(setLhs.array[iLhs])
         setReturn.insert(setLhs.array[iLhs])
   RETURN setReturn
```

The cost of this algorithm is quite high. Since we need to go through each element in setLhs, then the cost is at least linear with respect to setLhs.num_elements. However, for each element in setLhs, we need to search through setRhs. This would also be linear. If setLhs and setRhs are comparable in size, then we have n^2 which is quite expensive.

Sorted: As you can imagine, with a sorted implementation of setLhs and setRhs, a more efficient algorithm is possible. There are essentially three cases:

- setLhs.array[iLhs] == setRhs.array[iRhs]: insert one of them to the beginning of setReturn. Since they are the same, it does not matter which you choose. After it is inserted, then advance the index iLhs++ and iRhs++.
- setLhs.array[iLhs] < setRhs.array[iRhs]: advance the index iLhs++.
- setLhs.array[iLhs] > setRhs.array[iRhs]: advance the index iRhs++.

The pseudocode is:

```
set.intersection(setLhs, setRhs)
   iLhs ← 0
   iRhs ← 0
   WHILE  iLhs < setLhs.numElements OR iRhs < setRhs.numElements
      IF iLhs = setLhs.numElements
         RETURN setReturn
      ELSE IF iRhs = setRhs.numElements
         RETURN setReturn
      ELSE IF setLhs.array[iLhs] = setRhs.array[iRhs]
         setReturn.addToEnd(setLhs.array[iLhs])
         iLhs++
         iRhs++
      ELSE IF setLhs.array[iLhs] < setRhs.array[iRhs]
         iLhs++
      ELSE
         iRhs++
   RETURN setReturn
```

As with Union, the cost of this algorithm is $(n + m)$.

Application

We can use a set to solve a wide variety of problems. Three examples will be shown here: to hold a hand of playing cards, to keep track of a collection of students in a class, and to figure out which classes are required for graduation.

Hand of Playing Cards

The game of "21" is played as followed: two cards are dealt to a player. Every card is assigned a point value according to its rank. Number cards are given a point value equal to their face value. Face cards are given 10 points. The player gets to choose whether the Ace is worth 1 point or 11. If the player gets 21 points on the deal, an event called "black jack," then they automatically win the hand. Our goal is to write a function to verify if a given hand is a black jack.

Perhaps the easiest way to do this is to see if one card is an Ace and the other card is either a face card or a ten. We can verify if exactly one Ace is in the hand by finding the intersection between the set of all Aces and the user's hand:

```
bool oneAce(const set <Card> & sHand)
{
   // create a set of all the Aces
   char * aces[4] = { "AC", "AS", "AD", "AH" }; // Ace of clubs spades diamonds hearts
   set <Card> sAces;
   for (int i = 0; i < 4; i++)
      sAces.insert(Card(aces[i]));              // fill the sAces set

   // the intersection of Aces and cards in the hand will yield aces in the hand
   set <Card> sIntersection = sHand && sAces;

   // if there is one Ace in the hand, then oneAce is true. Otherwise, not!
   return sIntersection.size() == 1;
}
```

Will need to do the same thing for a `oneTen()` function. However, the tens stack will consist of 16 cards (10, Jack, Queen, and King for all four suits). However, the `sIntersection` set still must have a cardinality of exactly one. This makes our `isBlackJack()` function almost trivial:

```
bool isBlackJack(const set <Card> & sHand)
{
   return oneAce(sHand) && oneTen(sHand);
}
```

Note that the AND operator `&&` refers to the logical "AND" rather than set union. We can tell because both `oneAce()` and `oneTen()` return a Boolean. When the left-hand-side and the right-hand-side are Booleans, then we are using logical "AND." When the left-hand-side and the right-hand-side are sets, we are using union.

Students in a Class

Every class consists of a collection of students. The question is: what kind of collection? They are not a stack or a queue because any student can be accessed, not those on the end. They are not in a vector because the order does not matter. This means that a set is the obvious choice.

What kinds of things will a teacher want to do with the set of students? One thing is to find a given student to return an assignment.

```
bool returnAssignment(set <Student> & sClass,   // the students in the class
                      const Student & student,  // return the assign to this student
                      Assignment & assignment)  // the assignment to return
{
   set <Student> :: iterator it = sClass.find(student);
   if (sClass.end() != it)
   {
      it->returnAssignment(assignment);
      return true;
   }
   else
      return false;
}
```

Here first we find the student in the class list. If the student is found, then we return the assignment to the student. Otherwise, we report failure.

Another thing a teacher may want to do with the set of students is to discover who is missing on a given day. Here we need to start with the class roll. This is the set of students enrolled in the class. Next we will take a roll for the day. By comparing these two rolls, we can determine who is absent.

```
set <Student> getMissingStudents(const set <Student> & sClass, // students in the class
                                 const set <Student> & sRoll)  // who is here today
{
   return sClass - sRoll;
}
```

The difference operator will create a new set from the two given ones. It will start with `sClass` representing all students who should be in class. Next, it will remove from this set all the students who actually showed up today (`sRoll`). The remainder is those who should be present but are not. Notice that if everyone showed up, then `sRoll == sClass` so `sRoll - sClass == ∅`. Notice also that if the roll is empty (say the teacher passed out the roll on Saturday), then `sRoll == ∅` and `sClass - sRoll == sClass`.

The final thing the teacher might want to do is to create a roll at the beginning of class. Here the teacher will iterate through the class list and call out the students' name. If they respond, then they are part of the roll.

```
set <Student> getRoll(const set <Student> & sClass)
{
   set <Student> sRoll;
   set <Student> :: const_iterator it;
   for (it = sClass.begin(); s != sClass.end(); ++it)
   {
      char response;
      cout << "Is " << it->firstName() << " here? ";
      cin >> response;
      if (response == 'y' || response == 'Y')
         sRoll.insert(*it);
   }
   return sRoll;
}
```

Graduation Requirements

The first example is to figure out which classes are required for graduation. Consider the scenario when a student wants to major in Software Engineering and have a minor in Electrical Engineering. To figure out which classes are needed for graduation, it is necessary to consider the set of foundations classes required for any major, the set of classes specific to the Software Engineering, and the set of classes required for the Electrical Engineering minor.

There are a few things to note about this problem. First, there may be classes require for both Software Engineering and Electrical Engineering. This does not mean the student needs to take those classes twice! Second, the resulting set of classes will be the combination of the three requirements. This means we need to use union, not intersection.

The resulting function would look something like this:

```
set <Course> requiredCourses(const set <Course> & sFoundations,
                             const set <Course> & sMajorRequirements,
                             const set <Course> & sMinorRequirements)
{
   return sFoundations || sMajorRequirements || sMinorRequirements;
}
```

It is important to remember that the || operator means set union, not logical OR.

Examples

Example 05.1

Consider the following sets:

$$S_1 = \{ a, b, c, d, e, f, g, h, i, j, k, l, m, n \}$$
$$S_2 = \{ a, e, i, o, u \}$$
$$S_3 = \{ s, e, t \}$$

What is the result of the following:

a. $S_1.find(a)$	true or iterator pointing to 'a'
b. $S_1.find(t)$	false or the end() iterator
c. $S_2.find(a)$	true
d. $S_2.find(t)$	false
e. $S_3.find(a)$	false
f. $S_3.find(t)$	true

Example 05.2

Consider the following sets:

$$S_1 = \{ a, b, c, d, e, f, g, h, i, j, k, l, m, n \}$$
$$S_2 = \{ a, e, i, o, u \}$$
$$S_3 = \{ s, e, t \}$$

What is the result of the following:

a. $S_1.insert(a)$	$S_1 = \{ a, b, c, d, e, f, g, h, i, j, k, l, m, n \}$
b. $S_1.insert(t)$	$S_1 = \{ a, b, c, d, e, f, g, h, i, j, k, l, m, n, t \}$
c. $S_2.insert(a)$	$S_2 = \{ a, e, i, o, u \}$
d. $S_2.insert(t)$	$S_2 = \{ a, e, i, o, u, t \}$
e. $S_3.insert(a)$	$S_3 = \{ s, e, t, a \}$
f. $S_3.insert(t)$	$S_3 = \{ s, e, t \}$

Example 05.3

Consider the following sets:

$$S_1 = \{ a, b, c, d, e, f, g, h, i, j, k, l, m, n \}$$
$$S_2 = \{ a, e, i, o, u \}$$
$$S_3 = \{ s, e, t \}$$

What is the result of the following:

a. $S_1.erase(a)$	$S_1 = \{ b, c, d, e, f, g, h, i, j, k, l, m, n \}$
b. $S_1.erase(t)$	$S_1 = \{ a, b, c, d, e, f, g, h, i, j, k, l, m, n \}$
c. $S_2.erase(a)$	$S_2 = \{ e, i, o, u \}$
d. $S_2.erase(t)$	$S_2 = \{ a, e, i, o, u \}$
e. $S_3.erase(a)$	$S_3 = \{ s, e, t \}$
f. $S_3.erase(t)$	$S_3 = \{ s, e \}$

Example 05.4

Consider the following sets:

$$S_1 = \{ a, b, c, d, e, f, g, h, i, j, k, l, m, n \}$$
$$S_2 = \{ a, e, i, o, u \}$$
$$S_3 = \{ s, e, t \}$$

What is the result of the following:

a. $S_1 \cup S_2$	$\{ a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, u \}$
b. $S_1 \cup S_3$	$\{ a, b, c, d, e, f, g, h, i, j, k, l, m, n, s, t \}$
c. $S_2 \cup S_3$	$\{ a, e, i, o, u, s, t \}$
d. $S_1 \cup S_1$	$\{ a, b, c, d, e, f, g, h, i, j, k, l, m, n \}$
e. $S_1 \cup \emptyset$	$\{ a, b, c, d, e, f, g, h, i, j, k, l, m, n \}$
f. $S_1 \cup S_2 \cup S_3$	$\{ a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, s, t, u \}$

Example 05.5

Consider the following sets:

$$S_1 = \{ a, b, c, d, e, f, g, h, i, j, k, l, m, n \}$$
$$S_2 = \{ a, e, i, o, u \}$$
$$S_3 = \{ s, e, t \}$$

What is the result of the following:

a. $S_1 \cap S_2$	$\{ a, e, i \}$
b. $S_1 \cap S_3$	$\{ e \}$
c. $S_2 \cap S_3$	$\{ e \}$
d. $S_1 \cap S_1$	$\{ a, b, c, d, e, f, g, h, i, j, k, l, m, n \}$
e. $S_1 \cap \emptyset$	$\emptyset$
f. $S_1 \cap S_2 \cap S_3$	$\{ e \}$

Exercise 05.6

Consider the following sets:

$$S_1 = \{ a, b, c, d, e, f, g, h, i, j, k, l, m, n \}$$
$$S_2 = \{ a, e, i, o, u \}$$
$$S_3 = \{ s, e, t \}$$

What is the result of the following:

a. $S_1 - S_2$	$\{ b, c, d, f, g, h, j, k, l, m, n \}$
b. $S_2 - S_1$	$S_2 = \{ o, u \}$
c. $S_1 - S_3$	$\{ a, b, c, d, f, g, h, i, j, k, l, m, n \}$
d. $S_2 - S_3$	$S_2 = \{ a, i, o, u \}$
e. $S_1 - S_1$	$\emptyset$
f. $S_1 - \emptyset$	$\{ a, b, c, d, e, f, g, h, i, j, k, l, m, n \}$

Exercises

Exercise 05.1

Consider the following sets:

$$S_1 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \}$$

$$S_2 = \{ 2, 3, 5, 7, 11, 13, 17, 19 \}$$

$$S_3 = \{ 1, 2, 4, 8, 16 \}$$

What is the result of the following:

a. $S_1.find(7)$	
b. $S_1.find(8)$	
c. $S_2.find(7)$	
d. $S_2.find(8)$	
e. $S_3.find(7)$	
f. $S_3.find(8)$	

Please see page 121 for questions

Exercise 05.2

Consider the following sets:

$$S_1 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \}$$

$$S_2 = \{ 2, 3, 5, 7, 11, 13, 17, 19 \}$$

$$S_3 = \{ 1, 2, 4, 8, 16 \}$$

What is the result of the following:

a. $S_1.insert(7)$	
b. $S_1.insert(8)$	
c. $S_2.insert(7)$	
d. $S_2.insert(8)$	
e. $S_3.insert(7)$	
f. $S_3.insert(8)$	

Please see page 120 for questions

Exercise 05.3

Consider the following sets:

$$S_1 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \}$$

$$S_2 = \{ 2, 3, 5, 7, 11, 13, 17, 19 \}$$

$$S_3 = \{ 1, 2, 4, 8, 16 \}$$

What is the result of the following:

a. $S_1.delete(7)$	
b. $S_1.delete(8)$	
c. $S_2.delete(7)$	
d. $S_2.delete(8)$	
e. $S_3.delete(7)$	
f. $S_3.delete(8)$	

Please see page 121 for a hint.

Exercise 05.4

Consider the following sets:

$$S_1 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \}$$

$$S_2 = \{ 2, 3, 5, 7, 11, 13, 17, 19 \}$$

$$S_3 = \{ 1, 2, 4, 8, 16 \}$$

What is the result of the following:

a. $S_1 \cup S_2$	
b. $S_1 \cup S_3$	
c. $S_2 \cup S_3$	
d. $S_1 \cup S_1$	
e. $S_1 \cup \emptyset$	
f. $S_1 \cup S_2 \cup S_3$	

Please see page 122 for a hint.

Exercise 05.5

Consider the following sets:

$$S_1 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \}$$

$$S_2 = \{ 2, 3, 5, 7, 11, 13, 17, 19 \}$$

$$S_3 = \{ 1, 2, 4, 8, 16 \}$$

What is the result of the following:

a. $S_1 \cap S_2$	
b. $S_1 \cap S_3$	
c. $S_2 \cap S_3$	
d. $S_1 \cap S_1$	
e. $S_1 \cap \emptyset$	
f. $S_1 \cap S_2 \cap S_3$	

Please see page 124 for a hint.

Exercise 05.6

Consider the following sets:

$$S_1 = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 \}$$

$$S_2 = \{ 2, 3, 5, 7, 11, 13, 17, 19 \}$$

$$S_3 = \{ 1, 2, 4, 8, 16 \}$$

What is the result of the following:

a. $S_1 - S_2$	
b. $S_2 - S_1$	
c. $S_1 - S_3$	
d. $S_2 - S_3$	
e. $S_1 - S_1$	
f. $S_1 - \emptyset$	

Please see page 125 for a hint.

Exercise 05.7

From memory, recite and define the operations associated with the set ADT. Only provide the read-write version of the front and back methods.

Answers:

_____

_____

_____

_____

_____

_____

Please see page 119 for questions

Exercise 05.8

Write out the state of the set's member variables after each line or groups of lines are executed. This includes the four member variables (`numCapacity`, `numElement`, and `array`).

```
{
   set <int> s;

   s.insert(9);

   s.insert(31);
   s.insert(85);

   s.insert(90);
   s.insert(99);


   s.insert(50);

   set <int> :: iterator it;
   it = s.find(31);

   s.erase(it);

   set <int> s2;
   s2.insert(12);
   s2.insert(31);
   s2.insert(19);
   s2.insert(85);

   set <int> s3;
   s3 = s1 || s2;

   set <int> s4;
   s4 = s2 - s1;
}
```

numCapacity	numElements	array

Please see page 128 for a hint.

Exercise 05.9

Consider the following sets represented as sorted arrays:

$S_{\text{lhs}} =$

12	31	38	49	64	73
0	1	2	3	4	5

$S_{\text{rhs}} =$

3	9	31	49	79	99
0	1	2	3	4	5

Following the `union()` algorithm for sorted arrays presented on page 130, consider the case where the algorithm is in the middle of running. In other words, fill in the next element in a desk check. The values of the current state of the algorithm are the first three columns. What is the next step in the execution? Fill in the new values for `iLhs`, `iRhs`, and `setReturn`. The first two rows are filled out for you.

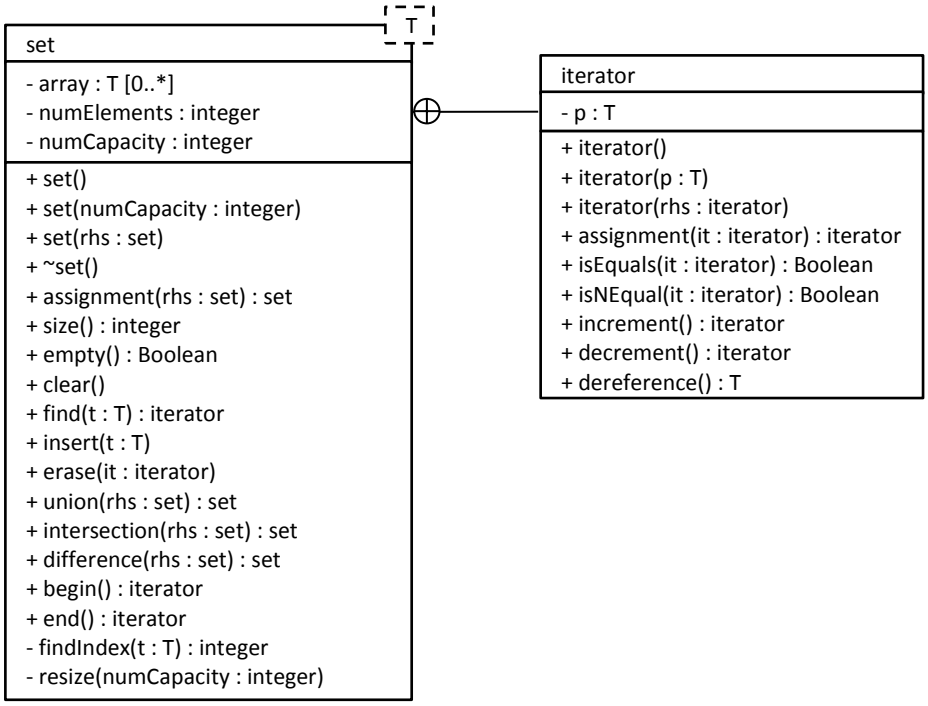
`setReturn`	`iLhs`	`iRhs`
{ }	0	0
{ 3 }	0	1

Please see page 130 for a hint.

Problems

Problem 05.1

Implement your own version of the `std::set` data structure. Call your implementation `custom::set`. In other words, place your class in the custom namespace and implement all of the interfaces described in this chapter. The UML class diagrams of our set and set iterator classes are the following:



Please see page 126 for a hint.

Problem 05.2

Using the STL `set` or using your own implementation of `set` from Problem 05.1, write a program to play the game Go Fish!

Read a hand (a `set` object) from a file. This will be the hand we will play the game with. The game will consist of five turns. With each turn:

- Prompt the user for a card (using the Card class' extraction operator)
- Test to see if the user's card it is the hand.
- If it is in the hand, remove it and display the message "`Found`." If it is not found, display the message "`Go Fish!`"

After the five turns are finished, display the number of matches and the cards remaining in the hand.

Please see page 132 for a hint.

Sam is sitting on a beach in a remote Caribbean island when he uncovers a bottle with a note rolled up inside. Intrigued, he opens the bottle and extracts the note. The note is a treasure map to a pirate's long-lost cache. Always ready for adventure, Sam decides to go on a little treasure hunt. He starts by going to the first point on the map. There he uncovers another bottle with the next leg of the trip. Going there, he finds another bottle with another map. This continues for a dozen legs until he finally uncovers the treasure: some rusted swords and a few copper coins.

This treasure hunt journey is much like a linked list. From each node, Sam has directions (a pointer) to the next node in the sequence. There is no guarantee that nodes are adjacent to each other in memory. In fact, they seldom are.

This chapter represents a rather drastic departure from the previous ones. Up to this point, we have been working with abstract data types based on arrays. This chapter is not about an abstract data type, nor does it does work with arrays. The linked list is a data structure with which we will build a full abstract data type in the next chapter. Linked lists also do not work with arrays. Instead the fundamental building block is called a node.

A node is a structure or a class consisting of a single data element and a pointer to the next node in the collection. The advantage to this configuration is that every element in the collection is independent of the rest, making it extremely easy to reorder or modify the collection. For example, recall how to push an element onto the end of a full vector or stack. To accomplish this, it is necessary to allocate a new buffer and copy all the elements over one by one. When working with nodes, we can do this by simply allocating a new node and connecting it to the rest. No copying or moving is required.

The essential characteristics of a linked list are:

- **Container**: The linked list is designed to store a collection of data.
- **Sequential Access**: Elements can be retrieved only by iterating through the linked list.
- **Order Preserved**: The client specifies the order in which data is stored.
- **Expandable**: The capacity of the linked list can be increased to meet the needs of the client.
- **Unrestricted Insertion**: It is possible to add an element to any location in the linked list.

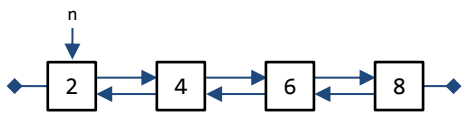
Definition

A linked list is a data structure where elements can only be accessed in sequence, not through random access. In other words, it is possible to find the ith element in the collection only after having traversed the preceding `i` elements.

The linked list is not an abstract data type, but rather it is a building-block from which abstract data types are built. Thus, the client of the linked list must be aware of the implementation details and perform manipulations manually. The most common operations to perform on a linked list are the following:

Name	Example	Description
Create	`node <int> * n = new Node(e);`	Create a new node
Insert	`insert(n, e);`	Insert a new node containing `e` immediately before node `n`
Copy	`n2 = copy(n1)`	Copy the contents of one linked list (`n1`) into another (`n2`)
Clear	`clear(n);`	Remove all elements from the linked list
Find	`n = find(n1, e);`	Returns a pointer to the first node in `n1` that contains the element `e`
Remove	`remove(n);`	Remove one element from the linked list

We will represent a linked list with the following figure:



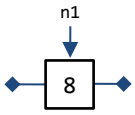
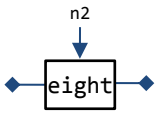
Here we have a linked list of 4 nodes with a pointer to the first. Note that the diamond at each end of the linked list (⟶◆) indicates the list does not extend beyond that point. The diamond indicates a `NULL` pointer.

Create

Every node in a linked list must be allocated. Since the new node might have a value, we create a node with a non-default constructor. If the no value is specified, then the default constructor is used for that datatype `T`.

```
Node <T> :: Node();
Node <T> :: Node(const T & t);
```

The client can assume that the resulting node has a `NULL` pointer.

	Code	Description	State
1	`Node <int> * n1 = new Node(8);`	Create a new node that is initialized to the value of 8. The pointers are set to `NULL`.	
2	`Node <string> * n2;` `n2 = new Node("eight");`	Create a new node that is initialized to the value.	

Insert

The `insert()` function is how elements are added to a linked list. This is true whether adding elements to the front, somewhere in the middle, or at the end of the list. The `insert()` function takes a `Node` as a parameter, indicating the node immediately after the new node to be added. We will call this node `pNode`. This node points to two other nodes, the next and previous nodes in the sequence. The `insert()` function needs to update these points so the linked list remains valid. The second parameter is the value to be inserted into the node. This second parameter is a template variable.

```
Node <T> * insert(Node <T> * pNode, const T & t) throw (const char *);
```

Unfortunately, this interface will not be sufficient for our needs. We need to handle two special cases:

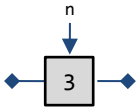
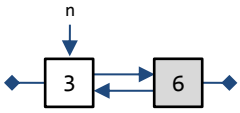
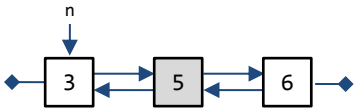
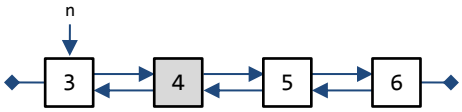
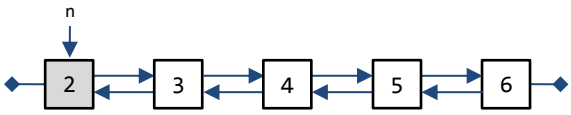
a) How do we insert onto the end of an existing list?
b) How do we insert onto the beginning of an existing list?

The first special case represents a challenge because we designed our `insert()` method to put elements *before* the passed node. Since there is no node that we can pass that will serve to put a new node at the end of the list, we need to change the interface. We will add a third parameter to specify that the new node is to go after the passed node. This will be done with a default parameter where inserting before is the default case.

The second special case presents a unique challenge in that we need to somehow indicate that the newly allocated node will be the new head of the list. Unfortunately there is currently no return value to send a pointer to the new head back to the caller. Fixing this requires changing our prototype again:

```
Node <T> * insert(Node <T> * pNode,      // our new node will be before this node
                  const T & t,           // the value that is to reside in the new node
                  bool after = false)    // by default, we insert before
                   throw (const char *); // anything that allocates could throw!
```

To see how these two cases work, consider the following:

	Code	Description	State
1	`Node <int> * n = NULL;`	Create an empty linked list.	
2	`n = insert(NULL, 3);`	Create a new node with the value 3. We typically create nodes with `insert()`.	
3	`insert(n, 6, true /*after*/);`	Create a new node with the value 6 and place it after the 3 node.	
4	`insert(n, 5, true /*after*/);`	Place the node 5 after the node 3 and before the node 6.	
5	`insert(n->pNext, 4);`	Place the node 4 before the node 5. Note that `n->pNext` refers to 5 before insertion time.	
6	`n = insert(n, 2);`	Place the node 2 before the node 3. Note that this requires the head to be updated.	

Find

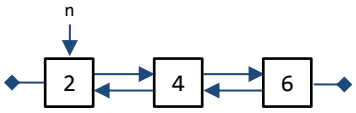
Recall that there are two ways to find elements in a sorted list: a linear search and a binary search. The linear involves walking through the array one slot at a time until the given element is found. The other way is to perform a binary search, jumping to the middle of the list and searching from there. With linked lists, we are forced to perform linear searches because this is the only way to traverse a linked list. In other words, it is only possible to perform a binary search when you can jump to (randomly access) a given index. This jumping or random access is impossible with linked lists.

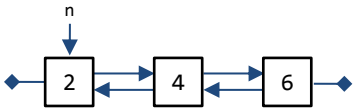
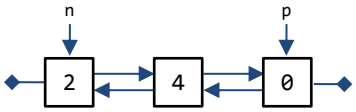
The `find()` function starts at the passed node `pHead` and traverses the list until an element is found with a value matching the passed parameter `t`. It then returns a pointer to that node.

```
template <class T>
Node <T> * find(Node <T> * pHead, const T & t);
```

From here it is up to the client to determine what is to be done with the resulting pointer.

	Code	Description	State

1	`Node <int> * n = insert(NULL, 2);`
	`insert(n, 4, true);`
	`insert(n->pNext, 6, true);`
		Create a linked list with three elements	

2	`Node <int> * p;` `p = find(n, 5);`	Since 5 is not in the list, then `p` will receive the `NULL` address.	
3	`p = find(n, 6);` `p->data = 0;`	Since 6 is in the list, then `p` will point to the node containing the value 6. With the pointer, it is possible to directly change the value in the node.	

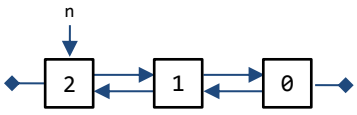
Clear

It is common to need to delete an entire list.

```
template <class T>
void clear(Node <T> * & pHead);
```

This will require every node in succession to be freed.

	Code	Description	State

1	`Node <int> * n = NULL;`
	`for (int i = 0; i < 3 i++)`
	`  n = insert(n, i);`
		Create a linked list with three elements. Since we are inserting onto the head of the list, the numbers are in reverse order.	

2	`clear(n);` `assert(NULL == n);`	Now `n` is empty. Note how `n` is set to `NULL`.	

Note that since linked lists are dynamically allocated, we must delete our list manually when the data-structure falls out of scope. If the client forgets to do this, then there will be a memory leak.

We will find out how to avoid this in the next chapter when we wrap the linked list in another class called `List`.

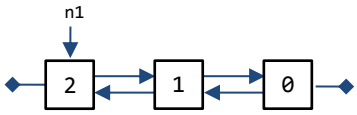
Copy

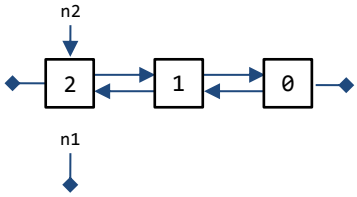
The `copy()` function makes a copy of each and every node in the source linked list into a new linked list. In other words, it is not enough to simply copy the pointer to the first element. Instead a new node must be allocated for each element in the source list.

```
template <class T>
Node <T> * copy(const Node <T> * pSource) throw (const char *);
```

After the copy function executes, a duplicate set of nodes is allocated.

	Code	Description	State

1	`Node <int> * n1 = NULL;`
	`for (int i = 0; i < 3 i++)`
	`  n1 = insert(n1, i);`
		Create a linked list with three elements. Since we are inserting onto the head of the list, the numbers are in reverse order.	

2	`Node <int> * n2 = NULL;`	Initially `n2` is empty.	
3	`n2 = copy(n1);` `clear(n1);`	Three nodes are allocated for `n2` and a copy of all the values from `n1` are brought over. Now we will remove all the nodes from `n1`. Notice how `n2` keeps its nodes.	

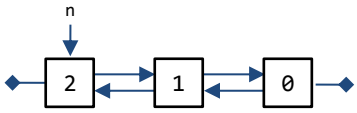
Remove

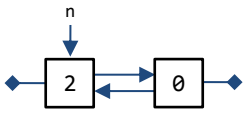
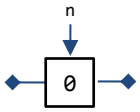
The purpose of the `remove()` function is to delete a single node from a linked list. The resulting list must be well-formed. Think of a linked list as a chain where each node serves as a link. If one node is removed from the node without updating the pointer of the preceding node, then the chain will be broken and the second part of the list will never be found. It is thus necessary to update the pointers.

```
template <class T>
Node <T> * remove (const Node <T> * pRemove);
```

There is one special case: what happens when you attempt to remove a node from the head of the list? Obviously, the head will move to the next node. This needs to be handled.

	Code	Description	State

1	`Node <int> * n = NULL;`
	`for (int i = 0; i < 3 i++)`
	`  n = insert(n, i);`
		Create a linked list with three elements. Since we are inserting onto the head of the list, the numbers are in reverse order.	

2	`remove(n->pNext);`	Remove the middle node, the node after `n` as indicated by `n->pNext`.	
3	`n = remove(n);`	Removing the head necessitates updating the head pointer. This is where the return value of `remove()` is important.	
4	`n = remove(n);` `assert(n == NULL);`	The final removal will set the pointer to `NULL`. We need to indicate the linked list is empty.	

Implementation

Manipulation of linked lists requires familiarity with pointers. If you are not comfortable with pointers by this time in your education, you will be after working with nodes for a few weeks!

When first learning about linked list manipulation, it may seem daunting. It is so easy to assign the wrong pointer or to do things in the wrong order. The easiest way to avoid these problems is to physically draw the nodes on a sheet of paper as you add lines of code.

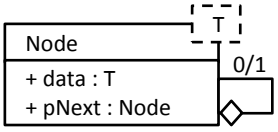
So how do we define a linked list? The answer is that we don't! Instead we define a node. A linked list is simply a collection of nodes. So how do we define a node? There are two big decisions to be made: 1) should we use a structure or a class, and 2) should we use a singly linked list or a doubly linked list?

Structure vs. Class

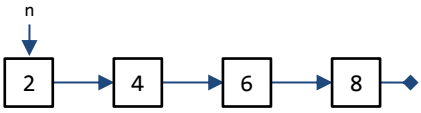
The first decision is whether to use a structure or a class. The object-oriented purest would point out that one should always use classes to hide implementation details from the client. However it is impossible to do this with a linked list; the client has to be intimately aware of the structure of the linked list or the list could be broken. For this reason, the benefits of the class implementation are lost so the structure implementation wins out. We will encapsulate the list into an abstract data type next chapter. This `list` ADT will be a class.

Singly Linked List

A singly linked list consists of a node with two elements: the data pertaining to the node and a pointer to the next node in the sequence. The UML class diagram of a singly linked list is:



If there are four elements in our linked list, it will appear as:



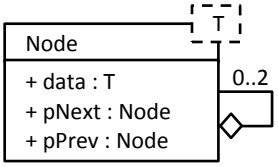
Notice how each arrow coming out of the right of a box is `pNext` and the value in the box is `data`. Singly linked lists have a couple interesting properties:

- It is impossible to skip elements in the sequence because each node only knows about the next node in the chain. In the above illustration, you cannot get from the node that has the value '2' in it to the node with '6' without passing through '4' first. This is because only '4' knows where '6' is located.
- It is impossible to go backwards. It might be that node '2' is the first element in the list, or it might be that there is another node that points to it. We simply can't tell.
- It is impossible to tell if we are at the front. In fact, we need to treat each node as the front because we have no way to tell if another node is before us.
- It is difficult to remove a node from a singly linked list. If, for example, you were to delete the node with the value 4, how would you fix the pointer in node 2 to now point to node 6? While this can be fixed, it is not an elegant solution.

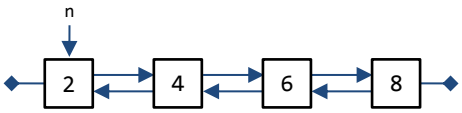
While it is not uncommon to define a singly linked list, these limitations present significant shortcomings. For this reason, most implementations use a doubly linked list.

Doubly Linked List

There is another variant called a doubly linked list. In this case, each node points to the next node in the sequence and the previous node. The UML class diagram of a doubly linked list is:



If there are four elements in our doubly linked list, it would appear as:



Notice how the arrows coming out of the right of each box represent the `pNext` pointer and the arrows coming out of the left represent `pPrev`. Other than that, the two implementations (singly and doubly linked lists) are the same. This extra pointer gives us some flexibility:

- As with singly linked lists, it is impossible to skip elements.
- It is possible to go forward or backwards in the linked list from a given node.
- It is possible to determine if a node precedes a given node in the linked list.

Note that a doubly linked list node is bigger than a singly linked list node. If the size of `data` is 4 bytes on a 64-bit computer, adding the extra pointer will increase the overall size of a node from 12 bytes to 20. Linked list operations are also slightly slower because more pointers need to be manipulated.

Because of the advantages of doubly linked implementations, we will follow a doubly linked design and leave the singly linked variant as an exercise.

Create

There are two steps to creating a node for a linked list. The first is to actually declare the variable. Since a node is a template variable, it is necessary to specify the template at declaration. Also, we use `Node` pointers to access nodes. Therefore, the variable will be a pointer. The following is the declaration of a pointer to a character node:

```
Node <char> * p;
```

Next, after the variable is declared, we need to allocate it. This requires calling the appropriate constructor. In the case of `Nodes`, we almost always use the non-default constructor so the data member variable can be initialized:

```
p = new Node('Q');
```

Creating of a node simply entails setting the three member variables.

```
Node(e)
   data ← e
   pNext ← NULL
   pPrev ← NULL
```

It is possible to have a default constructor for the node. In that case, initialization of the data member variable is performed by calling the element's default constructor.

The performance characteristic of the constructor is O(1) because initializing the member variables is completely independent of the size of the linked list the node is to be added to.

Clear

The `clear()` function deletes all the nodes in a linked list.

```
clear(p);
```

This involves iterating through all the nodes in the linked list and freeing all the associated memory. The assumption is that the nodes are allocated with `new`, which is commonly the case.

```
clear(pHead)
  WHILE pHead ≠ NULL
    pDelete ← pHead
    pHead ← pHead.pNext
    DELETE pDelete
```

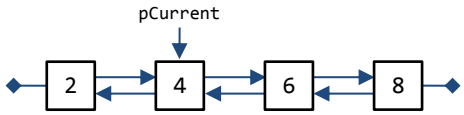
Once a node is deleted, the program should not reference the memory again. This means the iteration of the `pHead` pointer (with `pHead = pHead->pNext`, note how the pseudocode uses a dot and the C++ uses an arrow) should happen before the node is deleted (through `delete pHead`).

Notice how the WHILE loop iterates through every node in the linked list. Since every node is visited exactly once, the performance of the `clear()` function is O(n).

Insert

Inserting a node consists of little more than allocating the node, initializing the data member variable, and fixing up the pointers. The difficult part is to fix up the pointers.

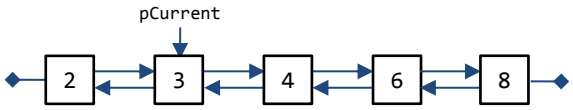
Consider the following linked list with the pointer `pCurrent` referring to some location in the middle:



We will call the `insert()` function with the following parameters:

```
insert(3,        /* the element value */
       pCurrent, /* a pointer to the '4' node */
       false);   /* the new node will not be after the '4' node, it will be before */
```

When finished, the new state of the linked list will be the following:

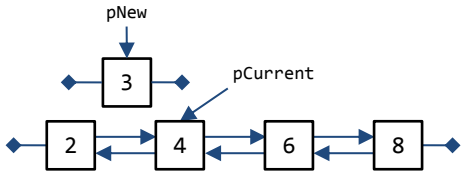


Notice that the node with the '2' did not move, nor did the node with the '4'. In fact, the entire linked list is unaffected by the insertion. Only the pointers in nodes '2' and '4' are changed.

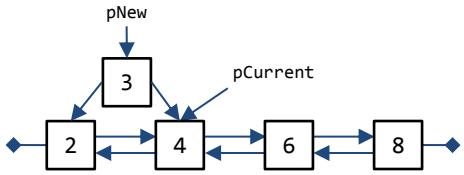
There are two potential pitfalls when performing an insertion. The first pitfall is changing the pointers in the wrong order, resuling in a broken chain and lost nodes. For example, what happens if you start by modify `pCurrent->pPrev`? The answer is that we lose track of the entire first half of the linked list! This is because the *only* connection we have to the beginning of the list from `pCurrent` is `pCurrent->pPrev`. Therefore, this critical link must be done last.

The second pitfall is that there may not be a node at `pCurrent->pPrev`. In the above example, this means that node '2' may not exist. We need to detect this case and handle it appropriately. Similarly, there may not even be a node at `pCurrent`. Here again, the `insert()` function should handle this case.

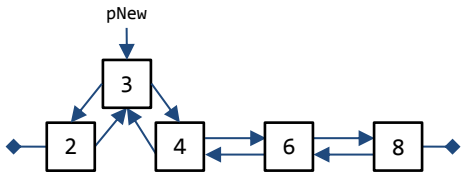
The way to accomplish an insertion is to first create a new node, then hook up the pointers of the new node to the existing nodes in the linked list, and finally fixing the previous and next node's pointers.



With the newly allocated node `pNew`, we next attach it to the correct location in the linked list.



At this point, the nodes in the linked list have not been modified and thus know nothing of the newly added node. The next step is to modify their pointers so the new node is in the proper sequence.



The pseudocode for inserting into a doubly-linked list is the following:

```
insert(e, pCurrent, after)
   pNew ← NEW Node(e)

   IF pCurrent ≠ NULL and after = false
      pNew.pNext ← pCurrent
      pNew.pPrev ← pCurrent.pPrev
      pCurrent.pPrev ← pNew
      IF pNew.pPrev
         pNew.pPrev.pNext ← pNew
   IF pCurrent ≠ NULL and after = true
      … something similar…
   RETURN pNew
```

This is performed in constant time, the performance largely depended on the performance of the assignment operator of the template type. In other words, adding items to the middle of a large link list takes no longer than adding an item to a list with only one element. Thus `insert()` is O(1).

Copy

The `copy()` function will copy all the nodes from one linked list into a new linked list. Consider, for example, a linked list with `pSource` as the head. If we wish to copy this into `pDestination`, the following code would be used:

```
Node <int> * pDestination = copy(pSource);
```

The process of copying a linked list is to iterate through the first linked list. For each element in the first list, we insert a new element onto the end of the second linked list. We must retain a pointer (`pDestination`) to the head of the second list so it can be the return value of the `insert()` function.

```
copy(pSource)
  pDestination ← new Node(pSource->data)
  pSrc ← pSource
  pDes ← pDestination

  FOR pSrc ← pSrc.pNext … end of the list
    pDes ← insert(pSrc.data, pDes, true)
  RETURN pDestination
```

This requires the copy function to have the ability to traverse the list. To do this we will use the standard FOR loop for iterating through a linked list:

```
for (Node * p = pHead; p; p = p->pNext)
   cout << p->data;
```

The first thing to notice about this loop is that the end condition is when the current-node pointer (`p`) is `NULL` (the only pointer value that is `false` because `NULL == 0x00000000`). The second thing to notice is that the increment operation involves advancing the pointer to the value of `pNext`.

The `copy()` process takes O(n) time because each node in the linked list needs to be visited exactly once.

Find

The `find()` function will return a pointer to the node containing a given element. This is accomplished by traversing the linked list until a node can be found whose element matches the passed parameter. Consider, for example, a linked list of integers. We will look for the value 42 in the linked list.

```
Node <int> * p = find(pLinkedList, 42);
```

To accomplish this, the `find()` function needs to traverse the linked list in much the same way the `copy()` function traverses the linked list.

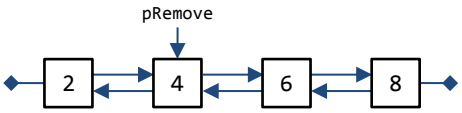
```
find(pHead, e)
 FOR p ←pHead … end of the list
  IF p.data = e
   RETURN p
 RETURN NULL
```

Notice how the loop stops as soon as the element is found. In other words, the `find()` function returns the first instance of the element in the linked list. Also, notice how the `find()` function returns the `NULL` address in the case of the element not being present in the list.

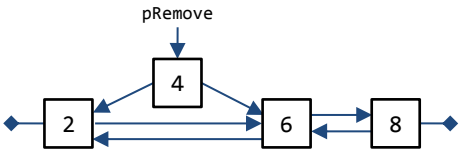
The worst case for the `find()` function is when the element is not found in the linked list. In this case, the performance is O(n). The best case is O() if it happens to be the first element in the linked list. The average case is n/2 so the overall performance is O(n).

Remove

The `remove()` method deletes the node `pRemove` in the linked list.



The first step is to modify the previous node and the next node so they no longer refer to the node that is to be removed.



Now that the underlying linked list is consistent without the node of `pRemove`, it is safe to delete it.

The pseudocode for removing a node is:

```
remove (pRemove)
 IF NULL = pRemove
  RETURN

 IF pRemove.pPrev
  pRemove.pPrev.pNext ← pRemove.pNext
 IF pRemove.pNext
  pRemove.pNext.pPrev ← pRemove.pPrev

 IF pRemove.pPrev
  pReturn ← pRemove.pPrev;
 ELSE
  pReturn ← pRemove.pNext

 DELETE pRemove
 RETURN pReturn
```

Observe how the list is completely unaffected aside from the pointers in the previous and the next node. Since all the operations can occur in constant time, the cost of this algorithm is O(1).

Application

We can use a linked list to solve a wide variety of problems. One examples will be shown here: to store the directions to a treasure in a treasure hunt.

Treasure Hunt

After having returned from a successful voyage of plunder and mayham, I would like to bury my treasure in a secret location on an island. In order to get to this location, the treasure seeker will need to visit a sequence of locations:

a) Visit the top of the tallest mountain on the island.
b) Go to the northern most point on the island.
c) In the center of Pirate's cove, there is a giant rock. Just north of the rock is a bottle underwater.
d) Travel to mainstreet pier and look under second to last plank.

The first task is to implement these instructions in a linked list. We will accomplish this by noting that the instructions are simple strings.

```
Node <string> * createTreasureMap()
{
   // create the start of the list
   Node <string> * pHead = Node("Visit the top of the tallest mountain on the island");

   // add the next location
   insert(pHead,
          "Go to the northern most point on the island",
          true /* insert after */);

   // and next
   insert(pHead->pNext,
          "In the center of Pirate's cove, there is a giant rock.  " /
          "Just north of the rock is a bottle underwater",
          true /* insert after */);

   // add final location
   insert(pHead->pNext->pNext,
          "Travel to mainstreet pier and look under second to last plank",
          true /* insert after */);

   return pHead;
}
```

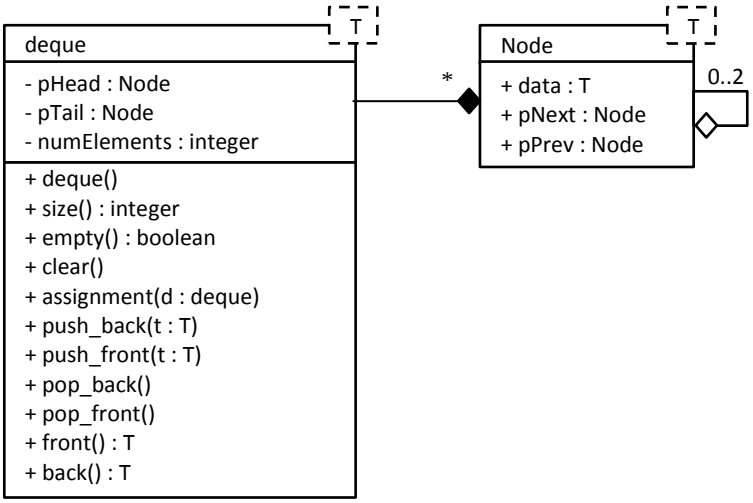
At this point, we realize that it is entirely too easy to find the treasure. I will add a function to make it much harder…

```
void makeItHarder(Node <string> * pHead)
{
   // make it way, way harder!
   insert(pHead->pNext,
          "Wrestle the most fierce bear and read what is written on his back tooth",
          true /* insert after */);
}
```

Notice how we can add an element in the list without modifying any of the other nodes.

Deque

It turns out that we can implement the `deque` data ADT with an array (as we did in Chapter 04) or with a linked list. With a pointer to both the head and the tail of the linked list, `push_back()` and `pop_front()` should be quite easy to implement. The UML class diagram of our linked list implementation is the following:



We will start with the class definition:

```
namespace custom
{
   template <class T>
   class deque
   {
   public:
      // constructors, destructors, and assignment operator
      deque()                      : pHead(NULL), pTail(NULL), numElements(0) {}
      deque(const deque <T> & rhs) : pHead(NULL), pTail(NULL), numElements(0)
      {
         *this = rhs;
      }
      deque <T> & operator = (const deque <T> & rhs) throw (const char *);
     ~deque()
      {
         clear(pHead);
      }

      // standard container interfaces
      int  size()  const { return numElements; }
      bool empty() const { return size() == 0; }
      void clear();

      // deque-specific interfaces
      void push_back( const T & t) throw (const char *);
      void push_front(const T & t) throw (const char *);
      void pop_back()              throw (const char *);
      void pop_front()             throw (const char *);
      T &  front()                 throw (const char *);
      T &  back()                  throw (const char *);
      T front() const              throw (const char *);
      T back()  const              throw (const char *);

   private:
      Node <T> * pHead;      // pointer to the first or head node if it exists
      Node <T> * pTail;      // pointer to the last  or tail node if it exists
      int numElements;       // count of the number of elements in the deque
   }; // class deque
}; // namespace custom
```

From here, several methods need to be defined. We will show `front()`, `push_front()`, `pop_front()`. Note that `back()`, `push_back()`, and `pop_back()` should work very much the same as their front counterparts.

Front

The `front()` method returns the element at the head of the `deque`. Since our `deque` is a linked list, it will return the data from the `pHead` node.

```
template <class T>
T & custom :: deque <T> :: front() throw (const char *)
{
   if (!empty())
      return pHead->data;
   else
      throw "ERROR: unable to access data from an empty deque"
}
```

Push Front

To add an element to the front of a `deque`, we will need to use our link list `insert()` function. For this example, we will simply call the `insert()` function as defined in this chapter. However, a more elegant implementation would be to put that code directly into the `push_front()` method. In other words, we would remove all of the link list code and place all the functionality directly into the `deque` class. We will do this next chapter with the `list` class.

```
template <class T>
void custom :: deque <T> :: push_front(const T & t) throw (const char *)
{
   try
   {
      pHead = insert(pHead, t);
      numElements++;
   }
   catch (...)
   {
      throw "ERROR: Unable to allocate a new node for deque"
   }
}
```

Pop Front

To remove the first element from the `deque`, we will use our link list `remove()` function. As with `insert()`, it would be more elegant to place this functionality directly in the `pop_front()` method rather than calling an external function. We will leave this modification as an exercise.

```
template <class T>
void custom :: deque <T> :: pop_front() throw (const char *)
{
   if (!empty())
   {
      pHead = remove(pHead);
      numElements--;
   }
   else
      throw "ERROR: unable to pop from the front of empty deque";
}
```

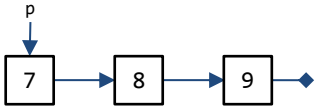
Examples

Example 06.1

Draw the linked list as it is at the end of execution of the following code:

```
{
   struct Node
   {
      int data;
      Node * pNext;
   };

   Node * p = new Node;
   p->data = 7
   p->pNext = new Node;

   p->pNext->data = 8;
   p->pNext->pNext = new Node;

   p->pNext->pNext->data = 9;
   p->pNext->pNext->pNext = NULL;
}
```

Answer:

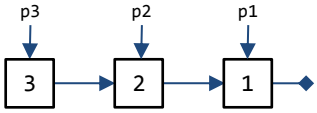


Example 06.2

Draw the linked list as it is at the end of execution of the following code:

```
{
   struct Node
   {
      int data;
      Node * pNext;
   };

   Node * p1 = new Node;
   Node * p2 = new Node;
   Node * p3 = new Node;

   p1->data = 1;
   p2->data = 2;
   p3->data = 3;

   p1->pNext = NULL;
   p2->pNext = p1;
   p3->pNext = p2;
}
```

Answer:

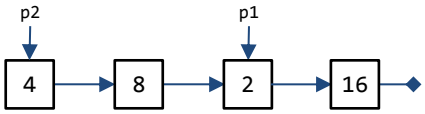


Example 06.3

Draw the linked list as it is at the end of execution of the following code:

```
{
   struct Node
   {
      int data;
      Node * pNext;
   };

   Node * p1 = new Node;
   p1->pNext = new Node;
   Node * p2 = new Node;
   p2->pNext = new Node;

   p1->data = 2;
   p2->data = p1->data * 2;
   p2->pNext->data = p2->data * 2;
   p1->pNext->data = p2->pNext->data * 2;

   p2->pNext->pNext = p1;
}
```

Answer:

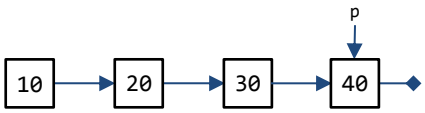


Example 06.4

Draw the linked list as it is at the end of execution of the following code:

```
{
   struct Node
   {
      int data;
      Node * pNext;
   };

   Node * p = new Node;
   p->data = 10;

   for (int i = 20; i < 40; i++)
   {
      p->pNext = new Node;
      p = p->pNext;
      p->data = i;
      p->pNext = NULL;
  }
}
```

Answer:



Exercises

Exercise 06.1

From memory, recite the functions associated with manipulating a linked list. Write the entire function definition in C++.

Answers:

_____

_____

_____

_____

_____

Please see page 145119 for questions

Exercise 06.2

Consider the following code:

```
{
   class Node
   {
   public:
      Node(int data) : data(data), pNext(NULL) {}
      int data;
      Node * pNext;
   };

   Node * p1 = new Node(11);
   Node * p2 = new Node(31);
   Node * p3 = new Node(61);

   p1->pNext = p2;
   p3->pNext = p2;
}
```

Draw the state of the linked list after the code is executed.

Please see page 149 for a hint.

Exercise 06.3

Consider the following code:

```
{
   class Node
   {
   public:
      Node(int data) : data(data), pNext(NULL) {}
      int data;
      Node * pNext;
   };

   Node * p1 = new Node(11);
   Node * p2 = new Node(31);
   Node * p3 = new Node(61);

   p1->pNext = p2;
   p2->pNext = p3;

   p1 = p2;
}
```

Draw the state of the linked list after the code is executed.

Please see page 149 for a hint.

Exercise 06.4

Consider the following code:

```
{
   class Node
   {
   public:
      Node(int data) : data(data), pNext(NULL) {}
      int data;
      Node * pNext;
   };

   Node * p1 = new Node(11);
   Node * p2 = new Node(31);
   Node * p3 = new Node(61);

   p1->pNext = p2;
   p2->pNext = p3;

   *p1 = *p2;
}
```

Draw the state of the linked list after the code is executed.

Please see page 149 for a hint.

Exercise 06.5

Consider the following code:

```
{
   class Node
   {
   public:
      Node(int data) : data(data), pNext(NULL) {}
      int data;
      Node * pNext;
   };

   Node * p1 = new Node(11);
   Node * p2 = new Node(31);
   Node * p3 = new Node(61);

   p1->pNext = p2;
   p2->pNext = p1;
   p3->pNext = p1;

   p1->pNext->pNext->data = 79;
   p2->pNext->pNext->data = 89;
   p3->pNext->pNext->pNext->data = 99;
}
```

Draw the state of the linked list after the code is executed.

Please see page 149 for a hint.

Exercise 06.6

Consider the following code:

```
{
   class Node
   {
   public:
      Node(int data) : data(data), pNext(NULL) {}
      int data;
      Node * pNext;
   };

   Node * p1 = new Node(11);
   Node * p2 = new Node(31);
   Node * p3 = new Node(61);

   p1->pNext = p2;
   p2->pNext = p3;

   Node * p4 = new Node(99);
   p4->pNext = p3;
   p2->pNext = p4;
}
```

Draw the state of the linked list after the code is executed.

Please see page 149 for a hint.

Exercise 06.7

Consider the following code:

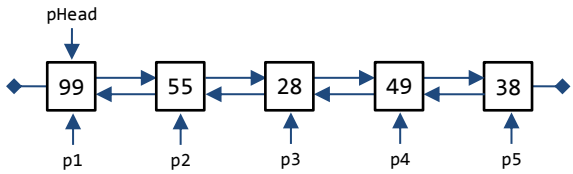
```
{
   class Node
   {
   public:
      Node(int data) : data(data), pNext(NULL) {}
      int data;
      Node * pNext;
   };

   Node * p1 = new Node(11);
   Node * p2 = new Node(31);
   Node * p3 = new Node(61);

   p1->pNext = p2;
   p2->pNext = p3;

   p1->pNext = p3;
   delete p2;
   p2 = NULL;
}
```

Draw the state of the linked list after the code is executed.

Please see page 149 for a hint.

Exercise 06.8

Consider the following list:



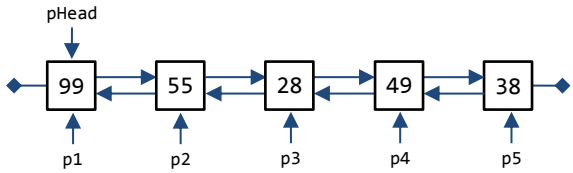
Draw the state of the linked list after the following code is executed:

```
{
   pHead = pHead->pNext;
   pHead->data = 3;
   pHead->pNext->data = 11;
}
```

Please see page 149 for a hint.

Exercise 06.9

Consider the following list:



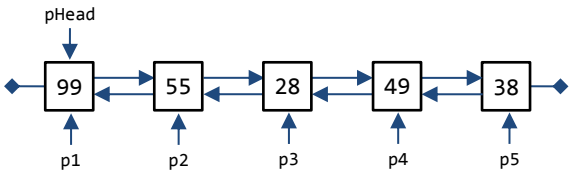
Draw the state of the linked list after the following code is executed:

```
{
   while (pHead->data > 30)
      pHead = pHead->pNext;
}
```

Please see page 149 for a hint.

Exercise 06.10

Consider the following list:



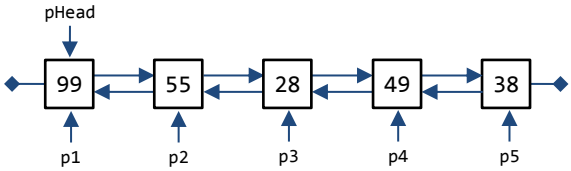
Draw the state of the linked list after the following code is executed:

```
{
   for (Node * p = pHead; p; p = p->pNext)
     if (p->data % 7 == 0)
        p->data = 31;
}
```

Please see page 149 for a hint.

Exercise 06.11

Consider the following list:



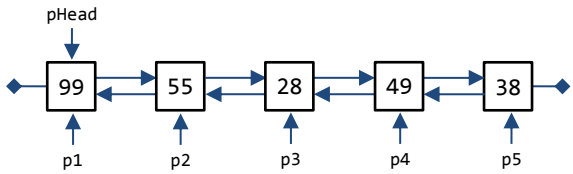
Draw the state of the linked list after the following code is executed:

```
{
   Node * p = new Node(0);

   p->pNext = p4;
   p->pPrev = p4->pPrev;
   p4->pPrev->pNext = p;
   p4->pPrev = p;
}
```

Please see page 149 for a hint.

Exercise 06.12

Consider the following list:



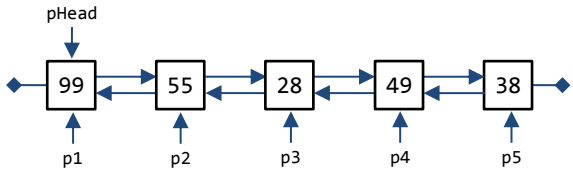
Draw the state of the linked list after the following code is executed:

```
{
   p3->pNext->pPrev = p3->pPrev;
   p3->pPrev->pNext = p3->pNext;
   delete p3;
   p3 = NULL;
}
```

Please see page 149 for a hint.

Exercise 06.13

Consider the following list:



Draw the state of the linked list after the following code is executed:

```
{
   Node * pPrevious;
   pPrevious = new Node(pHead->data);
   for (Node * p = pHead->pNext;
        p;
        p = p->pNext)
   {
      Node * pNew = new Node(p->data);
      pNew->pPrev = pPrevious;
      pPrevious->pNext = pNew;
      pPrevious = pNew;
   }
}
```

Please see page 149 or a hint.

Exercise 06.14

write out the state of the link list's member variables after each line or groups of lines are executed.

	State
`{`	
`Node <int> * p = new Node(50);`	

`insert(p,`
`  99,`
`  true /*after*/);`	

`insert(p, 38);`	
`p = insert(p, 79, true);`	
`p = insert(p->pPrev->pPrev,` `  12);`	
`Node <int> * p2;` `p2 = find(p, 50);`	
`remove(p2);`	
`insert(p->pNext->pNext->pPrev,` `  49, true);`	
`p2 = find(p, 49);`	
`Node <int> * p3 = NULL;`	
`p3 = copy(p2);`	
`clear(p3);`	
`p3 = copy(p);`	
`clear(p);`	

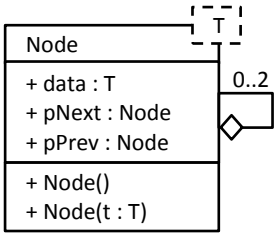
`p  = p3->pNext->pNext->pNext;`
`p  = p->pNext->pPrev->pPrev;`
`p2 = p->pNext->pNext;`
`}`	

Please see page 145 for a hint.

Problems

Problem 06.1

Implement your own linked list structure and the associated functions as described in this chapter. Call your implementation `Node`. The UML class diagram of the `Node` class is the following:



In addition, implement six functions as described in the chapter:

- Create
- Clear
- Insert
- Copy
- Find
- Remove

Please see page 149 for a hint.

Problem 06.2

Implement an insertion sort. As you may recall, a sort is an algorithm that converts an un-ordered collection of values into an ordered collection of values. By convention, sorts order the collection from least to greatest. In order for this to happen, sort algorithms typically need the collection data-type to support the less-than operator and the assignment operator.

The insertion sort is a sort similar to the method most humans use when ordering a deck of cards or a collection of papers. There are two parts to an insertion sort: the unsorted list and the sorted list. Initially the unsorted list consists of the entire collection and the sorted list is empty. With each iteration, items are removed from the unsorted list and inserted into the sorted list. This operation is continued until the unsorted list is empty and the sorted list consists of all the items in the collection.

You are to implement the insertion sort using your linked-list. This will be done because inserting items into the middle of a linked-list can be done in constant time, unlike array implementations where shifting of the array is required. The input will be an unsorted array of values. Build a sorted linked-list from this array and, when finished, copy the elements from the linked-list back into the input array.

Sue's grandfather managed a railroad yard a half century ago where trains came to unload cars, added new cars, or simply waited until the cars were needed again. In most cases, cars heading to a given location were located next to each other on the train. It was typical for a train to decouple the last 10 cars and add 12 new ones onto the back.

At first, this looks like an array application where each car is indexed by its order from the front of the train. However, there are some important differences. First, in order to remove one car from the train, it is necessary to traverse the train until the desired car is reached. Arrays do not need to traverse to get a given item; it is possible to simply index to the needed elment. Second, the way cars are removed from the train is different than how elements are removed from an array. When a car is removed from a train, the train is split into two halves. It is therefore necessary to reconnect the two halves to make the original train whole again. Finally, connecting two parts of a train into one does not involve moving any of the cars in the middle. Arrays require at least one of the halfs to be shifted into the new spot. With trains, no shifting is required. Only the ends that are connected need to be touched.

This example illustrates the fundamental properties of a list:

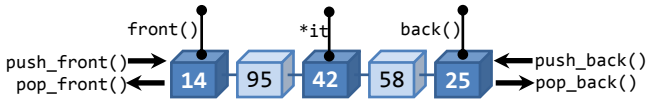
- **Container**: The list is designed to store a collection of data.
- **Sequential Access**: Elements can be retrieved only by iterating through the list.
- **Order Preserved**: The client specifies the order in which data is stored.
- **Expandable**: The capacity of the list can be increased to meet the needs of the client.
- **Unrestricted Insertion**: It is possible to add an element to any location in the list.

Definition

A list is a sequence container that can be expanded or contracted on either end. The operations performed on a list are the following:

Name	Example	Description
Create	`list <int> l;`	Create a new integer list.
Assignment	`l1 = l2;`	Make a copy of a list.
Empty	`f = l.empty();`	Returns `true` if a list has no elements, `false` otherwise.
Size	`n = l.size();`	Returns the number of elements currently in a list.
Clear	`l.clear();`	Remove all elements from a list.
Begin	`it = l.begin();`	Returns an iterator to the beginning of a list.
End	`it = l.end();`	Returns an iterator referring to the past-the-end element in a list.
Find	`it = l.find(e);`	Find a given element in a list. Return an iterator to the found element.
Insert	`l.insert(it, e);`	Insert an element into a list before the location specified by the iterator `it`.
Push Back	`l.push_back(e);`	Add an element to the back of a list.
Push Front	`l.push_front(e);`	Add an element to the front of a list.
Pop Back	`l.pop_back();`	Remove an element from the back of a list.
Pop Front	`l.pop_front();`	Remove an element from the front of a list.
Erase	`l.erase(it);`	Erase one element from a list as specified by the iterator `it`.
Front	`e = l.front();`	Returns the front element of a list.
Back	`e = l.back();`	Returns the back element from a list.

A list can be represented with the following figure:



There are several things to notice about this figure. First, the elements are not guaranteed to be next to each other in memory. There is a link between them. The exact nature of the link is not something the client needs to be concerned with.

Second, there is no `capacity()` method. This is because the underlying data-structure is a linked list and not an array. Thus the capacity is always the size.

Third, notice that the front and back of the list are directly accessible through the `front()` and `back()` method. It is also possible to reach elements through an iterator which can refer to any element in the list.

Finally, we can add elements to the list with the `push_back()` method. We can also add elements with the `insert()` method and remove elements with the `erase()` method.

Push & Pop Back & Front

The `list` class has four methods similar to the deque: `push_back()`, `push_front()`, `pop_back()`, and `pop_front()`.

```
void list <T> :: push_back (const T & t) throw (const char *);
void list <T> :: push_front(const T & t) throw (const char *);
void list <T> :: pop_back ();
void list <T> :: pop_front();
```

Observe how `push_back()` and `push_front()` could throw. This is because a node is always allocated when an element is added to either end of the list.

	Code	Description	State
1	`list <int> l;`	Create an empty list with no allocated nodes.	
2	`l.push_back(4);`	Add one element, with integer value 4 as the value in this case.	4 ← push_back(4)
3	`l.push_back(6);`	Here another new node is created and placed on the back of the list.	4 6 ← push_back(6)
4	`l.push_front(2);`	Another node is created, this one added to the front of the list.	push_front(2) → 2 4 6
5	`l.push_front(0);`	A fourth node is added, again to the front of the list.	push_front(0) → 0 2 4 6
6	`l.pop_front();`	The element 0 is removed from the front of the list.	pop_front() ← 2 4 6
7	`l.pop_back();`	The element 6 is removed from the back of the list.	2 4 → pop_back()

From the client's perspective, the list behaves exactly the same as a `deque` here. In fact, the `deque` can be implemented with a `list` instead of an array as the underlying data structure.

Front & Back

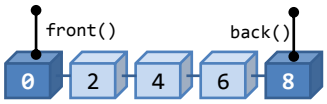
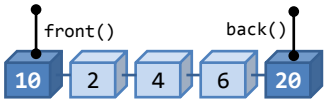
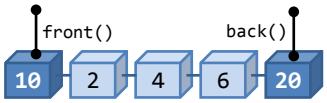
Also exactly the same as the `deque`, the `list` class has a `front()` and `back()` method which returns the element at the head or at the tail of the linked list.

```
T & list <T> :: back()  throw       (const char *);
T & list <T> :: front() throw       (const char *);
T   list <T> :: back()  const throw (const char *);
T   list <T> :: front() const throw (const char *);
```

Notice how there are two versions of `back()` and `front()`: one that serves both as a getters and setters, and another that serves only as a getter. The getter version returns a copy of `T` and is a constant method. The setter version returns the value by-reference.

	Code	Description	State

1	`list <int> l;`
	`for (int i = 0; i < 10; i += 2)`
	`  l.push_back(i);`
		Create a list and insert five elements in order.	

2	`assert(l.front() == 0);` `assert(l.back()  == 8);`	As with the deque, `front()` and `back()` operate as getters.	
3	`l.front() = 10;` `l.back()  = 20;`	As with the deque, `front()` and `back()` can operate as setters also.	
4	`assert(l.front() == 10);` `assert(l.back()  == 20);`	Again `front()` and `back()` operate as getters.	

List Iterator

As with `set`, iterators are the mechanism used to access elements in a `list`. The declaration for a standard forward-iterator is:

```
list <T> :: iterator it;
```

There are a total of four variations of the list iterator:

```
list <T> :: iterator               itForward;
list <T> :: const_iterator         itConstantForward;
list <T> :: reverse_iterator       itReverse;
list <T> :: const_reverse_iterator itConstantReverse;
```

Of course, the reverse iterator and the constant reverse iterator can only work if the underlying data-structure is a doubly-linked list. This is because we need a `pPrev` pointer in the `Node` class to move backwards.

The `list` class implements two main methods for creating iterators:

```
list <T> :: iterator list <T> :: begin();
list <T> :: iterator list <T> :: end();
```

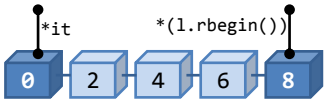
Recall that `begin()` returns the first element in the list and `end()` returns the first slot off the end of the list. This also means that the actual element can be retrieved from the iterator using the dereference operator. Recall that the "r" version of `begin()` And `end()` are for reverse iterators. Thus `rbegin()` points to the last element in the list. As with the `vector` and `set`, there are many variations of `begin()` and `end()`. The following are just the variations of `begin()`:

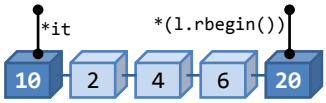
```
list <T> :: iterator list <T>               ::   begin();
list <T> :: const_iterator list <T>         ::   begin() const;
list <T> :: const_iterator list <T>         ::  cbegin() const;
list <T> :: reverse_iterator list <T>       ::  rbegin();
list <T> :: const_reverse_iterator list <T> ::  rbegin() const;
list <T> :: const_reverse_iterator list <T> :: crbegin() const;
```

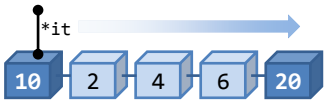
An example of how these iterators work is the following:

	Code	Description	State

1	`list <int> l;`
	`for (int i = 0; i < 10; i += 2)`
	`  l.push_back(i);`
		Create a list and insert five elements in order.	

2	`list <int> :: iterator it;`
	`it = l.begin();`
	`assert(*it == 0);`
	`assert(*(l.rbegin()) == 8);`
		The iterators can function as getters, allowing the client to directly access either end of the list.	

3	`*it = 10;` `*(l.rbegin()) = 20;`	Of course the iterators can function as setters as well as getters.	

4	`for (it = l.begin();`
	`  it != l.end();`
	`  ++it)`
	`  cout << *it << endl;`
		The most common thing to do with an iterator is to traverse the container. Here we will use the standard iterator FOR loop.	

Find

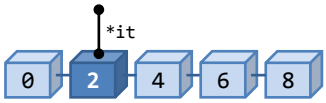
Find will return an iterator to the first instance of the list matching the passed value.

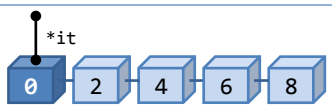
```
list <T> :: iterator list <T> :: find(const T & t);
```

It will call the equivalence operator on the passed parameter and, if the equivalence operator returns `true`, will return an iterator referring to that element. Because the `list` cannot be assumed to be sorted, the `find()` method will perform a linear search.

	Code	Description	State

1	`list <int> l;`
	`for (int i = 0; i < 10; i += 2)`
	`  l.push_back(i);`
		Create a list and insert five elements in order.	

2	`List <int> :: iterator it;`
	`it = l.find(2);`
	`assert(*it == 2);`
		Since `2` is the second element in the list, `it` will point to the second element.	

3	`it = l.find(0);` `assert(it == l.begin());`	Since `0` is at the head of the list, `it` will point to the same location as `l.begin()`.	
4	`it = l.find(5);` `assert(it == l.end());`	Since `5` is not in the list, `it` will point to `l.end()`. This is the "not found" location.	

Erase

It is possible to remove any element from a list efficiently. However, since we cannot refer to an element in the list with an index, we need an iterator to tell `erase()` which element to remove.

```
list <T> :: iterator list <T> :: erase(list <T> :: iterator it);
```

This prototype is quite complex. Notice that the return value is a `list <T> :: iterator`. This is an iterator to the element after the one which is erased. Why is this done? Imagine that the client was iterating through a list of elements for the purpose of removing a few that meet a certain criteria. In the following example, that criteria is when the `invalid()` method of the template parameter returns true:

```
void removeInvalid(list <Stuff> & listOStuff)
{
   for (list <Stuff> :: iterator it = listOStuff.begin(); it != listOStuff.end(); ++it)
      if (it->invalid())
         listOStuff.erase(it);   // ERROR
}
```

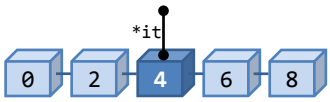
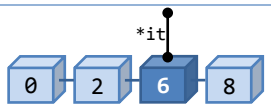
When the criteria is met, the `erase()` method will be called. Unfortunately, after the erase method is called, the iterator is no longer valid because it does not refer to a node in the linked list. This would mean that the loop would not continue. Instead, we need to update the value of the iterator from the return value of the `erase()` method.

```
void removeInvalid(list <Stuff> & listOStuff)
{
   for (list <Stuff> :: iterator it = listOStuff.begin(); it != listOStuff.end();   )
      if (it->invalid())
         it = listOStuff.erase(it);
      else
         ++it;
}
```

Rather than incrementing our iterator in the third field of the FOR loop, we increment it in the IF statement. When an `invalid()` element is found, we update the iterator based on the return type of `erase()`. Otherwise, we increment the iterator with the `++` operator.

	Code	Description	State

1	`list <int> l;`
	`for (int i = 0; i < 10; i += 2)`
	`  l.push_back(i);`
		Create a list and insert five elements in order.	

2	`list <int> :: iterator it;` `it = l.find(4);`	Point to the 3rd element in the list.	
3	`it = l.erase(it);`	The element with the value `4` was removed from the list. Observe how it now points to `6`, the next element.	
4	`l.erase(l.begin());`	We can erase an element using the iterator generated from `l.begin()`. This is the same thing as calling `l.pop_back()`.	

Insert

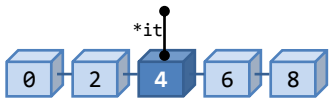
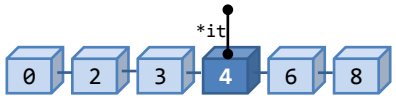
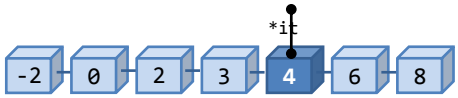
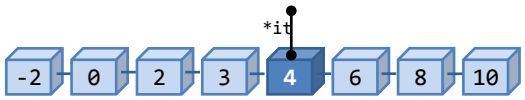
It is also possible to insert an element into the middle of a list efficiently. This is done without shifting elements as we had to do with `vector`.

```
list <T> :: iterator list <T> :: insert(list <T> :: iterator it,
                                        const T & t) throw (const char *)
```

As with `erase()`, it is necessary to specify an iterator indicating the spot before which the new element goes.

	Code	Description	State

1	`list <int> l;`
	`for (int i = 0;`
	`  i < 10;`
	`  i += 2)`
	`  l.push_back(i);`
		Create a list and insert five elements in order.	

2	`list <int> :: iterator it;` `it = l.find(4);`	Point to the 3rd element in the list.	
3	`l.insert(it, 3);`	Notice how we insert an element with the value `3` before the slot of the iterator.	
4	`l.insert(l.begin(), -2);`	Since `l.begin()` refers to the first element, we will place a new element with the value `-2` at the beginning of the list. This has the same effect as if we called: `l.push_front(-2)`.	
5	`l.insert(l.end(), 10);`	Since `l.end()` refers to the slot off the end of the list, we will place a new element with the value 10 at the end of the list. This is the same thing as calling `l.push_back(10)`.	

Standard Container Interfaces

As one would expect, the list implements all of the standard container interfaces. This is done exactly the same as with the `vector`, `stack`, `queue`, and `deque`.

```
int  list <T> :: size()  const;
bool list <T> :: empty() const;
void list <T> :: clear();
```

There is no `capacity()` method with the `list`. This is because the size is always the capacity; there is never an allocated but unused node.

	Code	Description

1	`list <int> l1;`
	`for (int i = 0; i < 10; i += 2)`
	`  l1.push_back(i);`
		Create a list and insert five elements in order.

2	`assert(l1.size() == 5);`	There should be 5 elements in the list.
3	`assert(l1.empty() == false);`	Because there are 5 elements, the list is not empty.

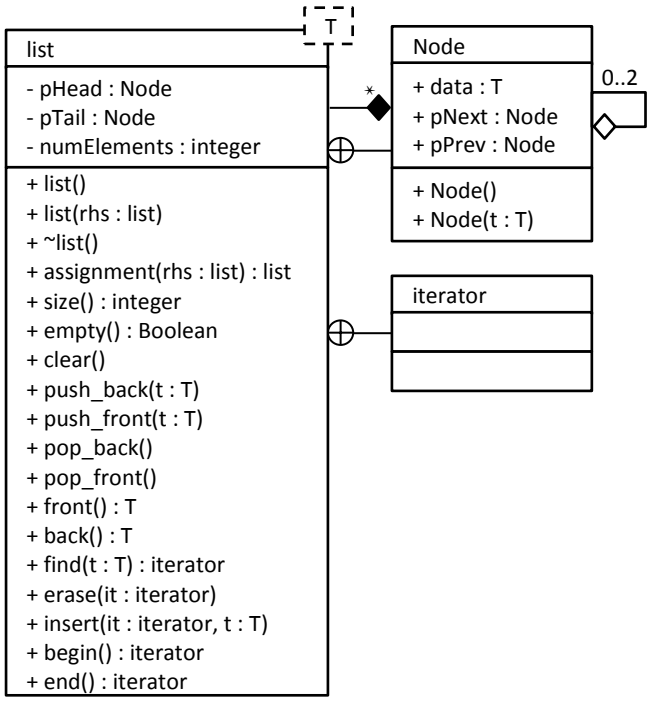
4	`list <int> l2;`
	`l2 = l1;`
	`assert(l2.size() == 5);`
		The assignment operator will create a copy of the list.

5	`l1.clear();`
	`assert(l1.size() == 0);`
	`assert(l2.size() == 5);`
		When we clear `l1`, we expect the size to be set to zero. However, the size of `l2` must remain unchanged.

Implementation

Most of the public interfaces of `list` are simply the linked list functions we discussed last chapter. In fact, the vast majority of the work needed to implement the `list` class is to turn the linked list functions into `list` methods. Recall that the linked list data structure required the client to know certain implementation details. The whole purpose of encapsulating the linked list into the `list` container is to relieve the client from the burden of knowing these details.

The first thing we need to understand is that the `list` class is able to work from both sides of the linked list equally. This includes the necessity to traverse the linked list backwards. The main fallout of this constraint is that we need to use the doubly linked version of the linked list. With this decision, we can define our UML class diagram:



There are several things to notice about this UML class diagram:

- Notice how we have two `Nodes`: `pHead` and `pTail`. Both are doubly-linked nodes, one pointing to the head of the `list` and one pointing to the tail.
- Notice how `iterator` is a nested class from `list` (we can tell from the ⊕ symbol). This works the same as `iterators` for `set` and `vector`.
- Notice how `Node` is also a nested class (using the ⊕ symbol). This means that `Node` is defined within `list`. Since the client should not need to be aware of the underlying `Node` class, it should be `private`. Also notice that `list` contains an unspecified number of `Nodes`. We can see this from the ♦ symbol). Thus `Node` is a nested class within `list` and `list` contains several `Nodes`.

For a review of the UML class diagram notation used above, please see Appendix D. Review: UML Class Diagram.

Constructor

The constructor needs to create an empty list of size zero. This is done by setting both the head and tail pointer to `NULL` and setting `numElements` to 0.

```
list.list
  pHead ← pTail ← NULL
  numElements ← 0
```

Empty

Another trivial method is to determine if the list is empty. This could be done by checking the head pointer for `NULL`, checking the tail pointer for `NULL`, or by checking `numElements`. While any of these are acceptable, a more robust design would do some consistency checking to make sure they agree.

```
list.empty
  RETURN pHead = NULL
```

Front & Back

Front and Back both return the data of nodes, one of the first node of the list and the second of the last. Aside from some error-checking, the algorithm is very straight-forward.

```
list.front
  IF NOT empty()
    RETURN pHead.data
  THROW empty list exception
```

Push Back & Front and Insert

Adding elements to the end of the list involves allocating new nodes and doing all the pointer matchup. While this may seem very complex, we will just rely on the work we did in linked lists and re-purpose the `insert()` function. In other words, the `insert()` function becomes an `list::insert()` method with a small modification.

```
list.insert(e, it)
   pNew = NEW Node(e)

   IF it ≠ NULL
      pNew.pNext ← it
      pNew.pPrev ← it.pPrev
      it.pPrev ← pNew
      IF pNew.pPrev
         pNew.pPrev.pNext ←pNew
   RETURN pNew
```

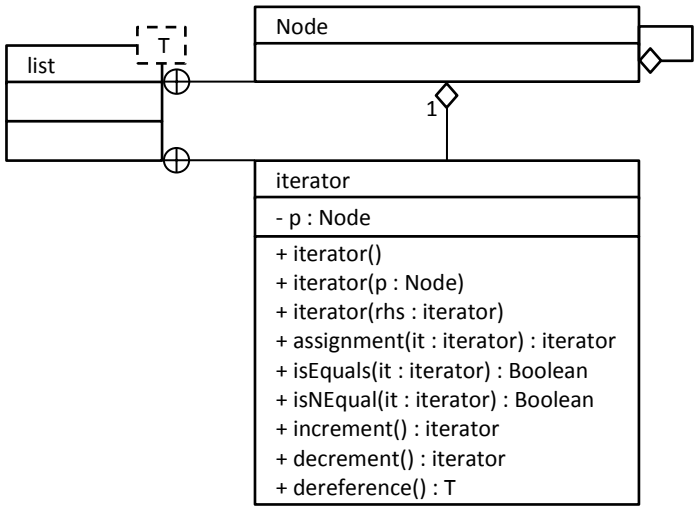
Notice how `insert()` needs to have access to the `p` member variable in the iterator. This is difficult because, as we see below, the `p` member variable is a private. How can we grant this access? The answer is to make the `list::insert()` method a friend of the `list::iterator` nested class.

```
friend iterator list <T> :: insert(iterator & it, const T & data) throw (const char *);
```

List Iterator

While defining an iterator is a convenience for working with vectors (the client can always iterate through the contents with an index and the `[]` operator), it is essential for working with a list. This is because there is no random-access and no `[]` operator.

The `begin()` method returns an iterator to the first element in the list. This requires us to define the list iterator.



This UML class diagram is quite complex. Let's look at it one step at a time.

- First, `list` is a template class. In order to instantiate a `list`, one needs to specify a data-type.
- Next, `iterator` is a nested class within `list`. We can tell this from the ⊕ symbol. Also, like set and `vector` before it, `iterator` is not a template class.
- `Node` is also a nested class (we can tell from the ⊕ symbol also).
- Notice the open diamond arrow ◇ between `iterator` and `Node`. This means that `iterator` contains a reference to `Node`. This reference is in the member-variable `p`. We use an open diamond because we contain a reference to a `Node`, but we do not own the `Node`. In other words, when the `iterator` is destroyed, the `Node` is not destroyed with it. If the `Node` was destroyed, then we would use a closed diamond ♦.

UML class diagrams can be quite complex and difficult to understand. They key is to focus on one relationship at a time rather than trying to understand the entire diagram in one take.

Begin

The `begin()` method must return an iterator whose member variable points to the the first node in the linked list. We must therefore define both the `list::begin()` method and a non-default constructor for `list <T> :: iterator` that takes a pointer to a `Node` as a parameter. The non-default constructor of the iterator is the following:

```
iterator.iterator(pNode)
  p ← pNode
```

Now we can define the `begin()` method for the `list` class.

```
list.begin()
  RETURN iterator(pBegin)
```

The cost of both of these operations is independent of the number of elements in the list. Therefore their algorithmic complexity is O(1).

End

The purpose of the `end()` method for iterators is to indicate when to stop the iterator once the end of the list is reached. As you may recall, `vector::end()` points to the first element off the end.

```
vector <int> iterator it;
for (it = v.begin(); it != v.end(); ++it)     // using an iterator to traverse a vector
   …
```

Thus when `it == v.end()`, we know we should stop the iterator. While it is possible to point to the first element off the end of the range with a vector (this being the slot adjacent to the last element), we cannot do this because we are working with nodes and not arrays. Therefore we simply define the `end()` method as having a `NULL` node:

```
list.end()
  RETURN iterator(NULL)
```

Increment

The final method that needs to be implemented is the increment operator in the iterator class. This is accomplished by advancing the pointer to the next node in the linked list.

```
iterator.advance()
  IF p.pNext
    p ← p.pNext
```

Going backwards works much the same way. This can either be accomplished with the forward iterator …

```
iterator.retreat()
  IF p.pPrev
    p ← p.pPrev
```

… or with the reverse iterator…

```
reverse_iterator.advance()
  IF p.pPrev
    p ← p.Prev
```

In all cases, a special check needs to be performed to make sure we are not already at the end of the list.

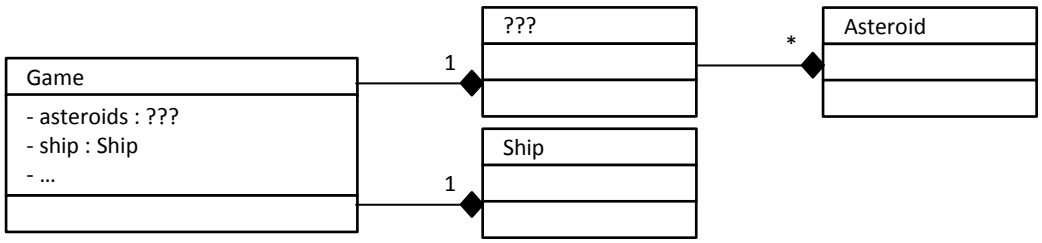
Application

Two examples will be presented to demonstrate how to utilize the list ADT: Asteroids and large numbers.

Asteroids

Atari introduced Asteroids in 1979 to the delight of video game enthusiasts everywhere. The game features a single ship controlled by the user and a collection of asteroids which destroy the ship if they come in contact with it. The object of the game is to shoot the asteroids. Each successive hit causes an asteroid to split and become successively smaller until it disappears. The game continues until there are no asteroids left.

To implement this game, it is necessary to have a member variable representing the ship and a collection of member variables representing the asteroids.



The big question is: how does one represent the collection of asteroids? There are a couple considerations:

a) When a new asteroid is created (because a larger one was hit by a bullet causing it to split), it can be placed anywhere in the collection. The location simply does not matter.
b) A bullet can hit and destroy any asteroid in the collection. In other words, removing an element from the middle of the collection is a common occurance.
c) When drawing the asteroids on the screen, they can be visited in any order.
d) When performing hit detection, the asteroids in the collection can be checked in any order.

Based on these considerations, we can evaluate the effectiveness of various data types:

- **Vector**: The first, third, and fourth consideration can be handled in O(1). However, the second must be done in O(n) if an element from the middle is to be removed. This is very expensive.
- **Stack**: The second, third, and fourth consideration invalidate the stack because every element in the collection must be visited.
- **Queue**: As with the stack, the second, third, and fourth consideration invalidate the queue because every element in the collection must be visited.
- **Deque**: Same problem as the stack and queue.
- **Set**: The set precludes having more than one element that is the same. Since the `==` and the `!=` operator are not defined for the `Asteroid` class, we cannot use the set.
- **List**: We can add new asteroids to the end of the list in O(1) with `push_back()`. We can remove an element in O(1) with `erase()` provided we have an iterator. We can iterate through the list as long as we do so in sequence, allowing us to draw the asteroids on the screen and perform hit detection.

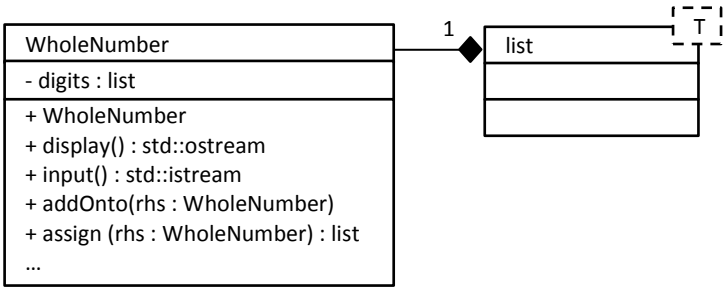
The best tool for the job appears to be the list because all the necessary operations are supported and can be done in O(1) time.

```
class Game
{
... code removed for brevity
private:
   list <Asteroid *> asteroids;
}
```

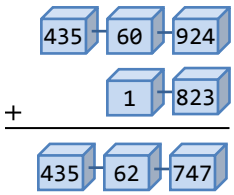
Large Numbers

The largest number that can be stored in an unsigned integer is about 4 billion. The reason for this limitation is that integers are stored as an array of 32 bits. Since the array is of fixed size, the maximum value is also fixed. So how can we store an integer of an arbitrary size? The answer is to use a data structure that can grow to accommodate large amounts of data.

One potential solution is to use a `vector` or `list` of bits. This would allow us to use the exact same algorithm that was used for the built-in `int` data-type. However, we can do better. Consider a `vector` or a `list` where every element contained a 3-digit `int`. Here a much smaller number of elements would be required to store a given number. A single 3-digit element can store the number `999`, requiring ten Boolean elements (because $2^{10} = 1024$) to do the same. This brings us to our `WholeNumber` class:



The difficult part of our `WholeNumber` class is that we have to perform arithmetic manually. For example, consider adding the following two numbers: `435,060,924 + 1,823`.



Just as we learned how to perform long addition in elementary school, we need to do the same algorithm here. When we add the end-elements, we arrive at `1747`. Since this is greater than `999`, we have to carry the 1 to the next group of digits and set `747` to the node in the right-most slot. This process continues until we have passed through the entire number.

When displaying a `WholeNumber`, we would like to insert commas as thousands separators. Thus, the number `435062747` is rendered as "`435,062,747`". We can accomplish this because the elements in our list are already grouped by thousands. The pseucode for this is the following:

```
display(number)
  it ← number.begin()
  DISPLAY *it
  WHILE it ≠ number.end()
    DISPLAY , *it
    ++it
```

There is one nagging bug with this algorithm, however. We want to display "`435,062,747`" instead of "`435,62,747`". In other words, we need to put a leading zero for 3-digit numbers that are between `10` and `99`, and we want to put two leading zeros for 3-digit numbers that are between `0` and `9`.

One final note about this `WholeNumber` class. It is not necessary to implement the assignment operator in this case. If we ever fail to provide the assignment operator for a class, then C++ generates a default one for us. This is accomplished by calling the assignment operator on each of the member variables. Since `list` has an assignment operator, it will work as we expect. This would not be the case if the member variable was a pointer of course.

Examples

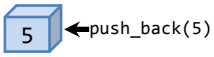
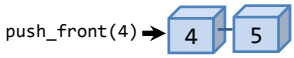
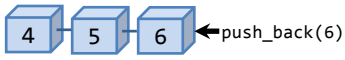
Example 07.1

Consider the following code:

```
{
   list <int> l;            // #1

   l.push_back(5);          // #2
   l.push_front(4);         // #3
   l.push_back(6);          // #4
   l.push_front(3);         // #5

   for (list <int> :: iterator it = l.begin(); it != l.end(); ++it)
      cout << *it << ' ';
}
```

What is the output?

Answer:

#1		Initially empty
#2		As with the `deque`, `push_back()` and `push_front()` work the same on an empty `list`.
#3		Add an element to the front of the `list`
#4		When the third element is added to the `list`, we only have direct access to the first and the third.
#5		Now we can only directly access element 3 and element 6. The only way to reach the middle elements is through an iterator.

The final output is:

```
3 4 5 6
```

Example 07.2

What is the output from the following code:

```
{
   list <int> l;

   for (int i = 0; i < 10; i++)
      l.push_back(i);
   list <int> :: iterator it = l.find(4);

   cout << (it != l.end() ? "Found!" : "Not present") << endl;
}
```

Answer:

```
Found!
```

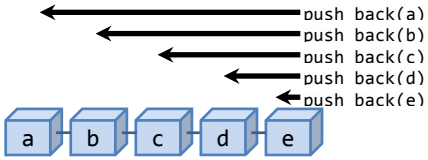
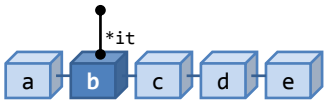
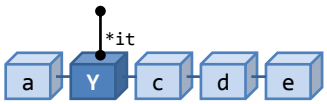
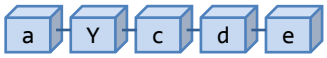
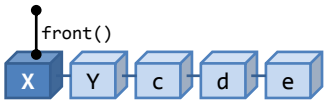
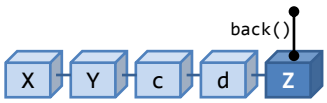
Example 07.3

Consider the following code:

```
{
   list <char> l;                    // #1

   for (char letter = 'a';
        letter <= 'e';
        letter++)
      l.push_back(letter);           // #2

   list <char> :: iterator it;
   it = l.find('b');                 // #3

   *it = 'W';                        // #4

   it = l.find('z');                 // #5

   l.front() = 'X';                  // #6

   l.back()  = 'E';                  // #7
}
```

What is state of the list at the end of execution?

Answer:

#1		Initially empty.
#2		This loop counts from `'a'` to `'e'` pushing them back one at a time. The result is a list of five elements.
#3		Search for the value `'b'` which is found in the 2nd node. Because we have an iterator pointing to this node, it is now accessible.
#4		Change the value `'b'` to `'Y'` in the second element.
#5		Since the value `'z'` is not found in the list, the iterator `it` no longer points to any elements.
#6		We can directly access the first element in the list with the `front()` method. We can both read and write with this method.
#7		We can also directly access the last element with the `back()` method.

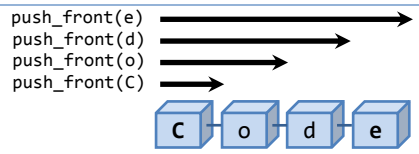
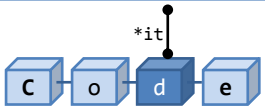
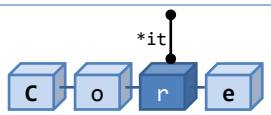
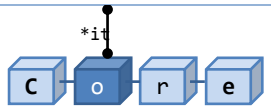
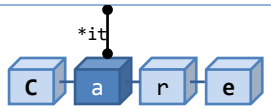
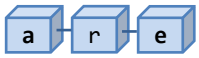
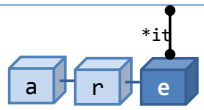
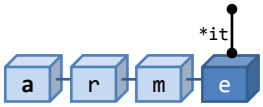
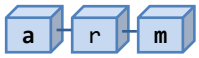
Example 07.4

Consider the following code:

```
{
   const char * text = "edoC";
   list <char> l;
   for (const char * p = text; *p; p++)
      l.push_front(*p);
                                        // #1
   list <char> :: iterator it;
   it = l.find('d');                    // #2
   *it = 'r';                           // #3
   --it;                                // #4
   *it = 'a';                           // #5

   l.erase(l.begin());                  // #6
   it = l.find('e');                    // #7
   l.insert(it, 'm');                   // #8
   l.erase(it);                         // #9
}
```

What is state of the list at the end of execution?

Answer:

#1		Because we are using `push_front()` rather than `push_back()`, elements get pushed onto the list in reverse order. As a result, the string `"edoC"` is reversed to `"Code"`.
#2		The element containing the value `'d'` is found, being the 3rd element in the list.
#3		Replace the value in the 3rd element with `'r'`.
#4		Move the iterator to the left by one element.
#5		Replace the value in the 2nd element with the value `'a'`
#6		Point to the first element in the list with `l.begin()`. This will be removed from the list with `l.erase()`.
#7		Direct the iterator to the last element in the list.
#8		Insert an element with the value 'm' immediately before the last item.
#9		Delete the last element in the list.

Exercises

Exercise 07.1

From memory, recite the methods associated with `list`. Write the entire function definition in C++.

Answers:

_____

_____

_____

_____

_____

_____

_____

_____

_____

Please see page 170 for questions

Exercise 07.2

For each of the following containers, determine whether the container has a given degree of functionality. If it does, indicate the name of the method. Indicate the algorithmic efficiency if it exists. The functionality is: **adding an element to the front of the collection**.

	name	best	typical	worst
`std::array`				
`std::vector`				
`std::stack`				
`std::queue`				
`std::deque`				
`std::set`				
`std::list`				

Exercise 07.3

For each of the following containers, determine whether the container has a given degree of functionality. If it does, indicate the name of the method. Indicate the algorithmic efficiency if it exists. The functionality is: **inserting an element somewhere into the middle fo the collection**.

	name	best	typical	worst
`std::array`				
`std::vector`				
`std::stack`				
`std::queue`				
`std::deque`				
`std::set`				
`std::list`				

.

Exercise 07.4

For each of the following containers, determine whether the container has a given degree of functionality. If it does, indicate the name of the method. Indicate the algorithmic efficiency if it exists. The functionality is: **remove an element from somewhere in the middle of the collection**.

	best	typical	worst
`std::array`			
`std::vector`			
`std::stack`			
`std::queue`			
`std::deque`			
`std::set`			
`std::list`			

Exercise 07.5

For each of the following containers, determine whether the container has a given degree of functionality. If it does, indicate the name of the method. Indicate the algorithmic efficiency if it exists. The functionality is: **remove an element from the back or tail of the collection**.

	name	best	typical	worst
`std::array`				
`std::vector`				
`std::stack`				
`std::queue`				
`std::deque`				
`std::set`				
`std::list`				

Exercise 07.6

For each of the following containers, determine whether the container has a given degree of functionality. If it does, indicate the name of the method. Indicate the algorithmic efficiency if it exists. The functionality is: **remove an element from the front or head of the collection**.

	best	typical	worst
`std::array`			
`std::vector`			
`std::stack`			
`std::queue`			
`std::deque`			
`std::set`			
`std::list`			

Exercise 07.7

Write out the state of the link list's member variables after each line or groups of lines are executed.

```
{
   list <int> l;

   l.push_back(79);
   l.push_front(38);
   l.push_back(99);

   list <int> :: iterator it;
   it = find(79);


   l.inset(it, 50);


   l.insert(l.begin(), 12);


   it = find(50);


   it = l.erase(it);


   it = l.begin();
   it++;
   it++;


   l.insert(it, 49);



   list <int> l2;


   l2 = l;


   l.clear();

   for (it = l2.begin();
        it != l2.end();
        ++it)
     l.push_front(*it);

   l2.clear();
}
```

State

Please see page 170119 for a hint.

Exercise 07.8

write out the state of the link list's member variables after each line or groups of lines are executed.

```
{
   WholeNumber n1;


   n1 = 3;


   n1 = 99;


   n1 = 397001496;


   n1 += 99000;


   n1 -= 397;


   WholeNumber n2(1396487001);


   n1 = n1 + n2;

   n1 = 999;
   n2 = 1;


   WholeNumber n3 = n1 + n2;

   n1 = 999999999;
   n2 = 1;
   n3 = n1 + n2;
}
```

State

Please see page 183119 for a hint.

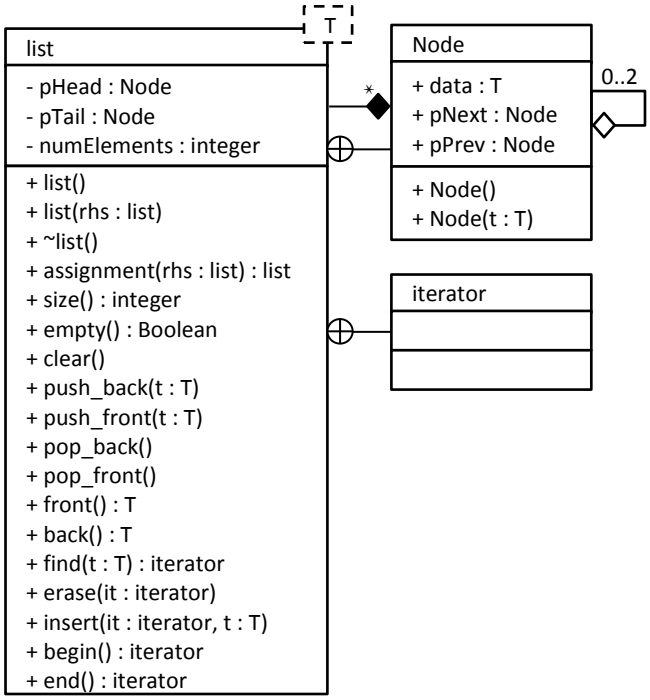
Exercise 07.9

Write the first 20 Fibonacci numbers, computed by hand.

Problems

Problem 07.1

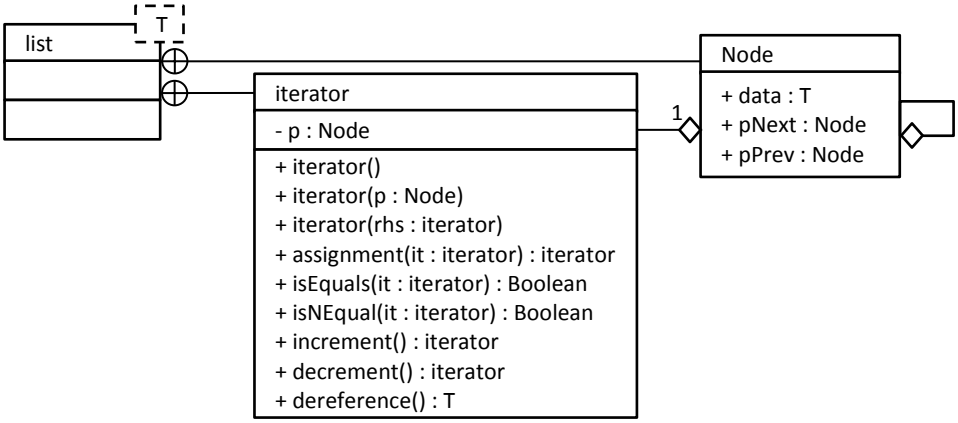
Implement your own version of the `std::list` data structure. Call your implementation `custom::list`. In other words, place your class in the custom namespace and implement all of the interfaces described in this chapter. The UML class diagram of our list class is the following:



Please see page 178 for a hint.

Problem 07.2

Implement an iterator for our `list` class:



Please see page 178 for a hint.

Problem 07.3

The Fibonacci sequence is a sequence of numbers defined in such a way that the first element is 1, the second element is 1, and the third and subsequent elements are the sum of the previous two elements. Thus the first ten numbers in the Fibonacci sequence are:

$$\{1, 2, 3, 5, 8, 13, 21, 34, 55\}$$

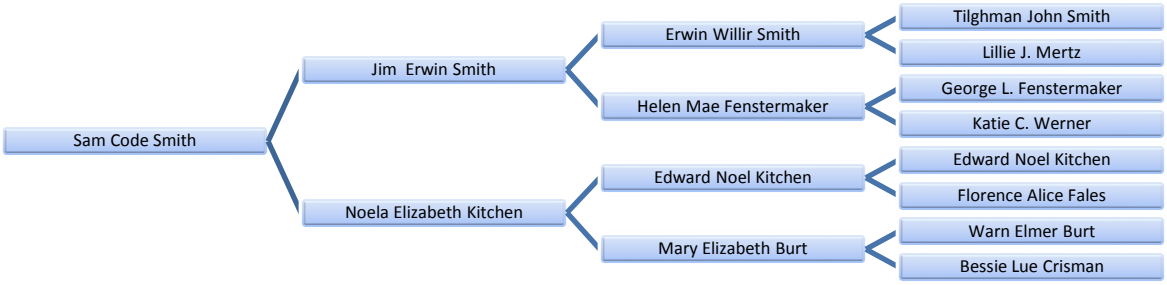
Write a program to display Fibonacci numbers. There is one caviot to this problem: Your program must be able to output very large Fibonacci numbers:

```
Which Fibonacci number would you like to display? 100
        354,224,848,179,261,915,075
```

To accomplish this, you will need to create your own data-type that is able to handle numbers of arbitrary length. This data-type will need to use your `custom::list` class where each node is a three-digit grouping of numbers. The above value will be seven nodes. In other words, integers can only represent numbers up to 4 billion (11 digits) whereas the above number has 21 digits. As you can imagine, your new data-type will need to be able to perform addition, copy/assignment, and display.

Please see page 183 for a hint.

Shortly after getting engaged to his high school sweatheart, Sam invites Sue to a large family reunion. There are cousins, nieces and nephews, aunts and uncles, and more distant relations. Almost a hundred people are in attendance, all related to him in some capacity. To help her make sense of it all, Sam draws out a simple family tree with his direct descendents.



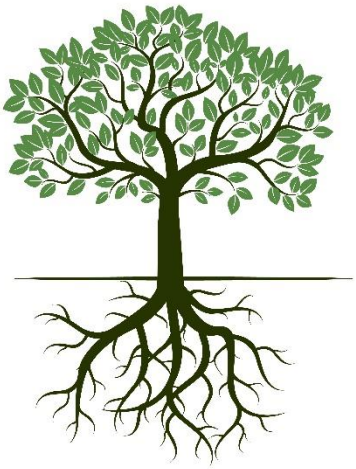
This family tree diagram is a binary tree. A binary tree is a collection of nodes connected together somewhat like those of a linked list. The big difference is that whereas every linked list node has two pointers (previous and next), binary tree nodes have three (parent, left child, and right child).

The essential properties of a binary tree are:

- **Container**: The binary tree is designed to store a collection of data.
- **Search Access**: Elements can be retrieved by searching from the root.
- **Order Preserved**: The client specifies the order in which data is stored.
- **Expandable**: The capacity of the binary tree can be increased to meet the needs of the client.
- **Unrestricted Insertion**: It is possible to add an element to any location in the binary tree.

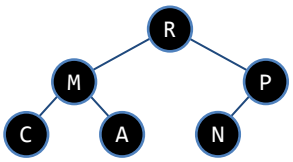
Recall from earlier chapters how the list abstract data type was built from the linked list data structure. In other words, the linked list provided the structure on which we were able to build a truly generic and useful abstract data type. We will do the same thing in this and several future chapters with the binary node and the binary tree.

One final thought… Think of every node in a binary tree as a tree where the branches diverge. Thus the trunk becomes two branches which then become four and so on. Sam's family tree diagram above would have to be rotated 90˚ counter-clockwise to make it look like this. The problem with this tree analogy is that, by convention, the root is on top and we branch down. This means that it is more similar to the root system of a tree rather than the branch system of a tree.



Definition

A binary tree is a collection of connected binary nodes:



Before we define the binary tree, a few node definitions are necessary to understand:

- **Node**: A data structure containing a single value.
- **Binary Node**: A data structure consisting of a value and three pointers: left, right, and parent.
- **Parent Node**: The binary node connected to at least one other node with a left or right pointer.
- **Child Node**: A binary node connected to a parent node through a parent pointer.
- **Tree**: A collection of connected binary nodes.
- **Arc**: The connection between nodes in a tree.
- **Directed Arc**: A one-way connection between nodes in a tree.
- **Root**: The binary node having a `NULL` parent pointer from which all the other nodes can be reached.
- **Leaf**: A binary node with no child nodes.
- **Siblings**: Two binary nodes sharing the same parent node.
- **Ancestor Node**: A node that can be reached by following a succession of parent pointers.
- **Descendant Node**: A node that can be reached by following a succession of right or left pointers.

The root node is R because R has no parents. M is the left child of R and P is the right child. Thus, both M and P are siblings and child nodes of R. Note that C and A are also siblings. The set of leaf nodes for our binary tree are {C, A, N} because they have no children. From node C, we can see that the set of ancestor nodes are {M, R}. The same is true with C's sibling A. The ancestor nodes of N are {P, R}. The set of descendant nodes of M are {C, A} whereas the set of descendant nodes of R are {C, M, A, N, P}.

We can define binary trees with these definitions of terms related to binary nodes:

- **Binary Tree**: A collection of binary nodes connected by pointers.
- **Subtree**: The collection of descendant nodes relative to a given reference node.
- **Path**: The sequence of nodes one would visit to reach one node from another by following pointers.
- **Height**: The length of the path from a given node to the root node plus one. The root has a height of 1, the next level has a height of 2, and so on.
- **Level**: The set of binary nodes in a given binary tree that have the same height.
- **Perfect**: A binary tree where all the levels are filled. For example, a perfect binary tree with 3 nodes will have a height of 2 and a perfect binary tree of height 3 will have 7 nodes.
- **Complete**: A binary tree where all the levels are filled except possibly for the final level. All the nodes on the final level are as far left as possible.
- **Balanced**: A binary tree where the height of the left sub-tree differs from that of the right sub-tree by at most one for all nodes in the tree having less than 2 children.

The set of nodes {C, M, A, R, N, P} form a binary tree. The sub-tree defined by M are the nodes {C, M, A} whereas the sub-tree defined by P is {P, N}. The path from C to R is the sequence [C, M, R]. Note that a path is not a set because the order is relevant. The path from A to N is [A, M, R, P, N]. Observe how the path can lead up or down to get to a given node. The height of R is 0, the height of M and

Ⓟ is 1, and the height of Ⓒ, Ⓐ, and Ⓝ is 2. This is not a perfect tree because Ⓟ has no right sibling. However, it is a complete tree because all the levels are filled except level 2.

The operations commonly performed on a binary tree are the following:

Name	Example	Description
Create	`BNode <int> * b = new BNode(e);`	Create a new binary node.
Size	`n = sizeBTree(b);`	Returns the number of nodes in `b`'s sub-tree.
Copy	`b2 = copyBTree(b1);`	Copy one binary tree (`b1`) into another (`b2`).
Clear	`deleteBTree(b);`	Remove all elements from the binary tree.
Add Left	`addLeft(b, e);`	Add an element to the left branch of a node.
Add Right	`addRight(b, e);`	Add an element to the right branch of a node.
Display	`cout << b;`	Display all the elements in a tree.

Observe how these operations are supported with stand-alone functions rather than member functions.

Create

A binary node is created using the default constructor or the non-default constructor.

```
BNode <T> :: BNode();
BNode <T> :: BNode(const T & t);
```

In all cases, the data-type of the elements is specified.

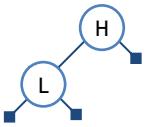
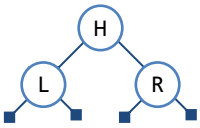
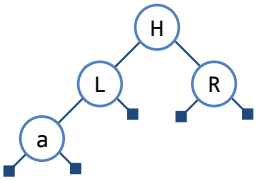
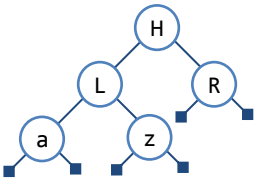
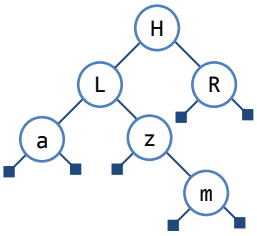
	Code	Description	State
1	`BNode <int> b1;`	The default constructor will set both `pLeft` and `pRight` to `NULL`. It will also call the default constructor on the template variable. In this case, setting the value to `0`.	
2	`BNode <int> b2(4);`	The non-default constructor sets the template variable to 4.	
3	`BNode <char> b3('z');`	The non-default constructor will initilize this `char-BNode` to the letter 'z'.	

Add Left & Right

A binary node is created using the default constructor or the non-default constructor.

```
void addLeft (BNode <T> * pNode, BNode <T> * pAdd);
void addRight(BNode <T> * pNode, BNode <T> * pAdd);
void addLeft (BNode <T> * pNode, const T & t) throw (const char *);
void addRight(BNode <T> * pNode, const T & t) throw (const char *);
```

There are two variants of `addLeft()` and `addRight()`. One will simply connect two existing nodes, the second will allocate a node for a given template value. In the case of the second, the function can throw if there is a problem allocating the node.

	Code	Description	State
1	`BNode <char> * b = BNode('H');`	Allocate a node with the value 'H'.	
2	`addLeft(b, 'L');`	Create a new node with the value 'L' to the left of the root node. Note that a new node will be allocated for this value.	
3	`addRight(b, 'R');`	Create a new node with the value 'R' on the right.	
4	`BNode <char> b2('a');` `addLeft(b->pLeft, &b2);`	Allocate a new node with the value 'a'. This node will be placed to the left of the 'L' node. Notice how `b2` is a `BNode`, not a pointer to a `BNode`. This means the address-of operator (`&`) must be used.	
5	`BNode <char> b3('z');` `addRight(b->pLeft, &b3);`	Another new child will be added to 'L'. In this case, it will be off the right side.	
6	`addRight(b->pLeft->pRight, 'm');`	Things are getting very complicated now. We are adding a new node with the value 'm'. We start at the head ('H'), then go left ('L'), then go right ('z'), and the new node will be to the right of that.	

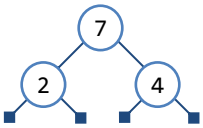
Size

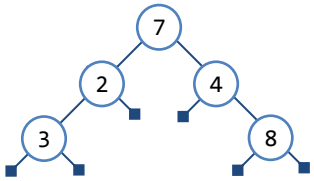
The size of a binary tree is the number of nodes reachable from the root.

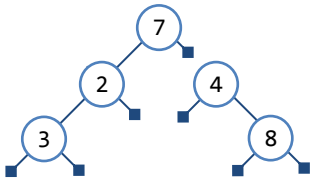
```
int sizeBTree(const BNode <T> * pRoot);
```

Note how the input parameter is a constant. This will guarantee that `sizeBTree()` will not affect any of the nodes.

	Code	Description	State
1	`BNode <int> * b = NULL;` `cout << sizeBTree(b);`	A pointer to `NULL` should return `0`, indicating there are no nodes in the tree.	
2	`b = new BNode(7);` `cout << sizeBTree(b);`	With one node, the size should be 1.	

3	`addRight(b, 4);`
	`addLeft (b, 2);`
	`cout << sizeBTree(b);`
		With two additional nodes, the size should be 3.	

4	`addRight(b->pRight, 8);`
	`addLeft (b->pLeft,  3);`
	`cout << sizeBTree(b);`
		With two more, the size should be 5.	

5	`b->pRight = NULL;` `cout << sizeBTree(b);`	By setting `pRight` to `NULL`, we are cutting a link in the binary tree. This is very bad; we have no reference to node 4 and 8 so they can never be deleted. However, the output will be 3.	

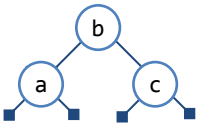
Delete

A binary tree must be explicitly deleted just like a linked list must be explicitly deleted.

```
void deleteBTree(BNode <T> * & pNode);
```

Observe how the pointer to the node is passed by-reference. This is because the pointer itself needs to be set to `NULL` indicating that all the nodes in the binary tree, including the root, are deleted.

	Code	Description	State
1	`BNode <char> * b = NULL;`	Create a `NULL` pointer with no associated node.	
	`deleteBTree(b);` `assert(b == NULL);`	This will do nothing because there are no nodes to delete.	
2	`b = new BNode <char> ('A');` `assert(b != NULL);`	Create a single node binary tree. Note that the pointer `b` cannot equal `NULL`.	
3	`deleteBTree(b);` `assert(b == NULL);`	Delete will remove the single node. At the end of this, the pointer must contain the `NULL` address.	

4	`b = new BNode <char> ('b');`
	`addRight(b, 'a');`
	`addLeft (b, 'c');`
	`assert(sizeBTree(b) == 3);`
		There should be three nodes now.	

5	`deleteBTree(b);` `assert(b == NULL);`	All three nodes are deleted.	

Copy

The process of copying a binary tree is the process of making a copy of every node in the binary tree. This requires a method to iterate through the source tree as well as a method to create the nodes. The prototype of the function copying a binary tree is the following:

```
BNode <T> * copyBTree(const BNode <T> * pSrc) throw (const char *);
```

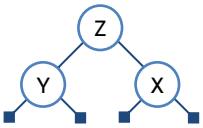
Note how the input parameter is a constant. This will guarantee that `copyBTree()` will not affect any of the nodes in `pSrc`.

	Code	Description	State of b2

1	`BNode <char> * b1 = new BNode('A');`
	`BNode <char> * b2;`
	`b2 = copyBTree(b1);`
		The single node tree will be copied into `b2`.	

2	`deleteBTree(b1);`	Even when `b1` was deleted, `b2` remains unchanged.	

3	`b1 = new Node('Z');`
	`addRight(b1, 'X');`
	`addLeft (b1, 'Y');`
		These changes to `b1` should not affect `b2`.	

4	`deleteBTree(b2);`	Deleting `b2` will not affect `b1`.	
5	`b2 = copyBTree(b1);`	Now `b2` will look like `b1`.	

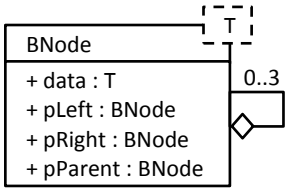
Implementation

There are two big decisions to be made when creating a binary node: whether to include a pointer to the parent node, and structure vs. class.

With a linked list, we can either have a singly-linked list (each node only points to the next node in the sequence) or a doubly-linked list (each node points to both the next and previous node). While there are many pros and cons to this decision, most implementations prefer the doubly-linked version due to the flexibity it offers. The same can be said with binary nodes. Without a pointer to the parent node, it would be impossible to tell whether a given binary node is the root of the tree or somewhere in the middle. Similarly, the pointer to the parent node facilitates many binary tree operations we will need to perform later.

The second decision is whether to use a structure or a class. As with the linked list, it is impossible to manipulate a binary tree without intimate knowledge of how the tree is built. Without such knowledge, our encapsulation efforts would be frustrated. Nevertheless, some operations are slightly more convenient to implement as methods than as functions. Ultimately the choice is yours. Here we will use a structure implementation for the `BNode` and later encapsulate the binary tree into another class when we build an abstract data type.

Based on this reasoning, the UML class diagram of our binary node is:



Notice how the functions operating on the `BNode` are not present in the UML class diagram. This is because the `BNode` is a structure and the functions are stand-alone. Next chapter we will encapsulate the `BNode`s as well as all the functions operating on them into a single class called `BST`.

Create

Creating a binary node is simply the process of setting all the member variables to `NULL`. It is also necessary to set the data value to the default value for `T`.

```
BNode.BNode()
  pParent ← pLeft ← pRight ← NULL
  data ← T()
```

There is also a non-default constructor which will initialize the pointers as well as the data value.

```
BNode.BNode(t)
  pParent ← pLeft ← pRight ← NULL
  data ← t
```

The constructors are the only methods in our class. Note that we could implement this just as easily as a structure because structures can also have methods.

Add Left and Right

When adding a node to the left of a given node, it is necessary to set up the various pointers. This means `pLeft` needs to be set as well as `pParent` of the resulting node.

```
addLeft(pNode, t)
  pAdd ← NEW BNode(t)
  pAdd.pParent ← pNode
  pNode.pLeft ← pAdd
```

Of course this function should gracefully handle memory allocation failures. There is also a version of this function which will take a node as a parameter.

```
addLeft(pNode, pAdd)
  IF pAdd ≠ NULL
    pAdd.pParent ← pNode
  pNode.pLeft ← pAdd
```

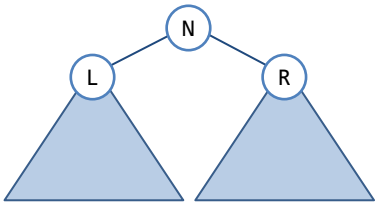
Notice the `NULL` check on `pAdd`. It should be possible to set the `NULL` address to the pointer to the left of a given node.

There is also an `addRight()` version of both of these functions which work much the same way. Because the differences are trivial, the pseudocode will be left as an exercise.

In all cases, the performance of the `addLeft()` and `addRight()` are the same: O(1). This is because the time it takes to allocate a node and assign the pointers is independent of the size of the tree to which the new node is attached.

Size

To determine the size of a binary tree, it is necessary to visit and count every node from a given root node. This requires traversing the binary tree. Binary tree traversal is quite a bit more complex than traversing a linked list because, with every node, both the left and right child need to be visited. Consider the following binary tree:



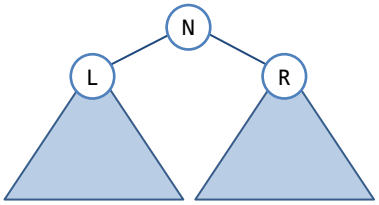
We are trying to compute the size under node `N`. This node serves as the root node under this sub-tree. It may be the root of the entire tree or not; we don't care at this point. The number of nodes under `L` and `R` may be very large or may be zero. We also don't know or care at this point. There is one thing we do know, however. The size of the sub-tree under `N` is the same as the size of the sub-tree under `L` plus the size of the sub-tree under `R` plus one. In other words: `sizeBTree(n) = sizeBTree(l) + sizeBTree(r) + 1`. This can be expressed in pseudocode recursively as:

```
sizeBTree(node)
  IF node = NULL
    RETURN 0
  ELSE
    RETURN sizeBTree(node.pLeft) + 1 + sizeBTree(node.pRight)
```

Observe how there is a well-defined stop condition (when the pointer we are currently visiting does not exist), and we make progress towards the solution (a leaf node) with every interation. Since we visit every node once, the overall performance of this function is O(n).

Delete

To delete a binary tree, it is necessary to visit every node in the tree and delete it individually. In many ways, this seems to be the same problem as the size function discussed earlier. However, there is an important detail. After a node is deleted, it is no longer possible to look at the member variables associated with the node. Thus, we cannot delete a node until all the children nodes have been deleted. To illustrate this point, consider again our figure depicting a binary tree:



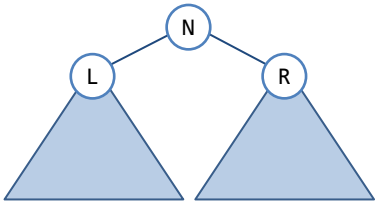
If we attempt to delete the left sub-tree, then the node `N`, then the right sub-tree, we will lose access to `pRight` and thus be unable to delete the right sub-tree. This means that deleting the node will be the last thing we will do:

```
deleteBTree(node)
  IF node = NULL
    RETURN
  deleteBTree(node.pLeft)
  deleteBTree(node.pRight)
  DELETE node
```

This is called post-order traversal, appropriate when the underlying node must be visited last.

Copy

To copy a binary tree, it is necessary to traverse the entire tree just like with `sizeBTree()` and `deleteBTree()`. The big difference here is that we must create a new node before the left and right sub-trees are created. In other words, this is the exact opposite of the `deleteBTree()` constraint. To illustrate this point, consider again our figure depicting a binary tree:



We need to first copy the node over before we copy the left and right sub-trees:

```
copyBTree(source)
  IF source = NULL
    RETURN
  destination←NEW BNode(source.data)

  destination.pLeft ← copyBTree(source.pLeft)
  IF destination.pLeft ≠ NULL
    destination.pLeft.pParent ← destination

  destination.pRight ← copyBTree(source.pRight)
  IF destination.pRight ≠ NULL
    destination.pRight.pParent ← destination

  RETURN destination
```

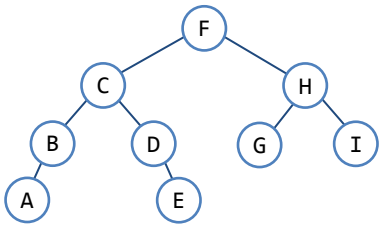
This is called pre-order traversal, appropriate when the underlying node must be visited first.

Application

There are several uses for binary trees. The following section will illustrate how to perform several traversals of a binary tree (LVR, RVL, VLR, LRV, and level-order) as well as how to use binary trees to generate Huffman Codes.

Display tree

There are several ways to traverse a binary tree: LVR, RVL, VLR, LRV, and level-order traversal. The strange three-letter acronyms such as LVR refer to the order in which nodes are visited. L refers to the left sub-tree, V refers to visiting the current node, and R refers to the right sub-tree. Each will be demonstrated with the following tree:

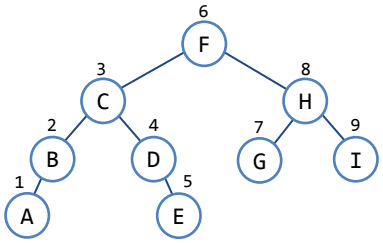


LVR: In-order Traversal

In-order traversal involves the following recursive steps:

a) Check if the underlying node is `NULL`. If it is `NULL`, return.
b) Go to the Left sub-tree.
c) Visit the underlying node.
d) Go to the Right sub-tree.

In the above tree, nodes will be visited in the following order:



The code to display these nodes is thus:

```
template <class T>
void displayLVR(const Node <T> * pHead)
{
   if (pHead == NULL)
      return;

   displayLVR(pHead->pLeft);      // L
   cout << pHead->data;           // V
   displayLVR(pHead->pRight);     // R
}
```

The output of executing this code in the above binary tree would be:

```
ABCDEFGHI
```

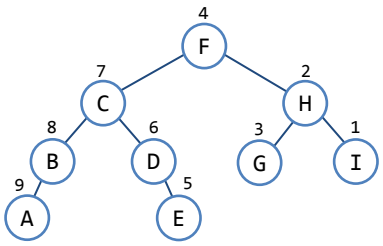
LVR is called in-order traversal because elements in the tree are visited from left-to-right. Since the elements in our tree (not all binary trees, of course) are sorted from left-to-right, the resulting output is in sorted order.

RVL: Backwards Traversal

Backwards traversal involves the following recursive steps:

a) Check if the underlying node is `NULL`. If it is `NULL`, return.
b) Go to the Right sub-tree.
c) Visit the underlying node.
d) Go to the Left sub-tree.

In the above tree, nodes will be visited in the following order:



The code to display these nodes are thus:

```
template <class T>
void displayRVL(const Node <T> * pHead)
{
   if (pHead == NULL)
      return;

   displayRVL(pHead->pRight);    // R
   cout << pHead->data;          // V
   displayRVL(pHead->pLeft);     // L
}
```

The output of executing this code in the above binary tree would be:

```
IHGFEDCBA
```

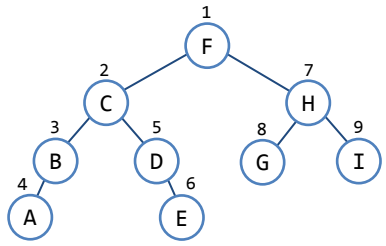
RVL is called backwards traversal because elements in the tree are visited from right-to-left. Since the elements in our tree are sorted from left-to-right, the resulting output is in reverse sorted order.

VLR: Pre-order Traversal

Pre-order traversal is useful when the underlying node must be visited first. An example would be our `copyBTree()` function where it was necessary to copy a given node before filling in the sub-trees below it. The recursive steps for pre-order traversal are the following:

a) Check if the underlying node is `NULL`. If it is `NULL`, return.
b) Visit the underlying node.
c) Go to the Left sub-tree.
d) Go to the Right sub-tree.

In the above tree, nodes will be visited in the following order:



The code to display these nodes are thus:

```
template <class T>
void displayVLR(const Node <T> * pHead)
{
   if (pHead == NULL)
      return;

   cout << pHead->data;          // V
   displayVLR(pHead->pLeft);     // L
   displayVLR(pHead->pRight);    // R
}
```

The output of executing this code in the above binary tree would be:

```
FCBADEHGI
```

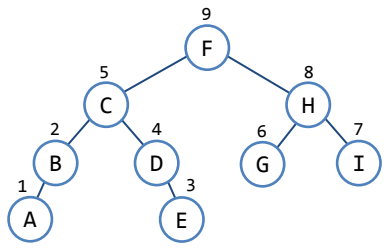
With VLR traversal, the root node will always be the first visited and a leaf node will always be the last.

LRV: Post-order Traversal

Post-order traversal is useful when the underlying node must be visited last. An example would be our `deleteBTree()` function where it was necessary to delete all the underlying sub-trees deleting the underlying node. The recursive steps for post-order traversal are the following:

a) Check if the underlying node is `NULL`. If it is `NULL`, return.
b) Go to the Left sub-tree.
c) Go to the Right sub-tree.
d) Visit the underlying node.

In the above tree, nodes will be visited in the following order:



The code to display these nodes are thus:

```
template <class T>
void displayLRV(const Node <T> * pHead)
{
   if (pHead == NULL)
      return;

   displayLRV(pHead->pLeft);    // L
   displayLRV(pHead->pRight);   // R
   cout << pHead->data;         // V
}
```

The output of executing this code in the above binary tree would be:

```
ABEDCGIHF
```

With LRV traversal, the root node will always be the last visited and a leaf node will always be the first.

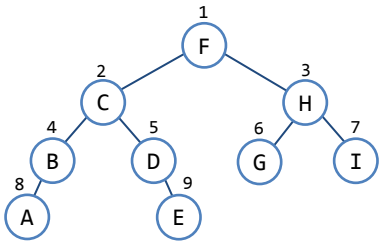
Level Order Traversal

Level order traversal involves first visiting the node at the top level (the root), then visiting the nodes at the next level (the children of the root), then visiting the nodes at the third level (the grand-children of the root), and so on. This way all the siblings (and first cousins and second cousins) are visited at about the same time.

The steps for level-order traversal are the following:

a) Check if the underlying node is `NULL`. If it is `NULL`, return.
b) Add the head of the tree to a queue.
c) While the queue is not empty…
    a. Add the left sub-tree to the queue if one exists.
    b. Add the right sub-tree to the queue if one exists.
    c. Visit the top node in the queue and pop it off.

In the above tree, nodes will be visited in the following order:



The code to display these nodes are much more complex because we need to remember which sub-trees have not yet been explored. We will do this with a queue to preserve the order of the sub-trees we will need to visit next. This algorithm is not recursive.

```
template <class T>
void displayLevel(const Node <T> * pHead)
{
   // do nothing if there is nothing to do
   if (NULL == pHead)
      return;

   // push the head of the tree onto the queue
   queue <Node <T> *> q;
   q.push(pHead);

   // while there are still sub-trees to visit…
   while (!q.empty())
   {
      // add the left and right sub-tree to the queue
      if (q.top()->pLeft  != NULL)
         q.push(q.top()->pLeft);
      if (q.top()->pRight != NULL)
         q.push(q.top()->pRight);

      // visit the current node
      cout << q.top()->data;
      q.pop();
   }
}
```

The output of executing this code in the above binary tree would be:

```
FCHBDGIAE
```

Huffman Codes

Huffman codes are a form of data compression where tokens are assigned binary names guaranteed to consist of the smallest number of bits.

The first step in genenerating Huffman Codes for a given token set is a frequency map. For example, consider the following five tokens:

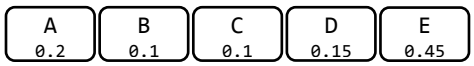
Token	Frequency
A	20%
B	10%
C	10%
D	15%
E	45%

In this example, A is twice as common as B and E is three times as common as D. The output of a Huffman Code is to assign binary values to each of the tokens with the goal of consuming the smallest amount of storage space. Thus Huffman Codes is a data compression algorithm. The resulting code for the above frequency table is:

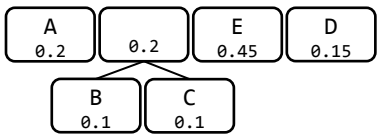
Token	Huffman Code
A	`111`
B	`100`
C	`101`
D	`110`
E	`0`

Notice how E only takes one bit while the rest take three.

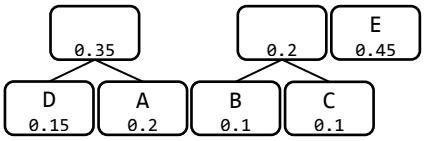
Huffman codes are generated using binary trees. First, each token is assigned to its own binary tree. Thus, in the above example, the list of trees is the following:



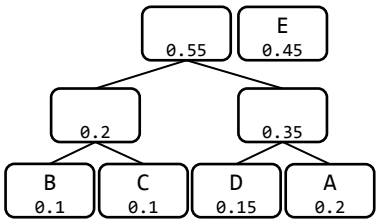
Next the trees with the smallest frequency are combined into a single tree. This will reduce the number of trees by one. The new tree will have the combined frequencies of the children: `0.2`.



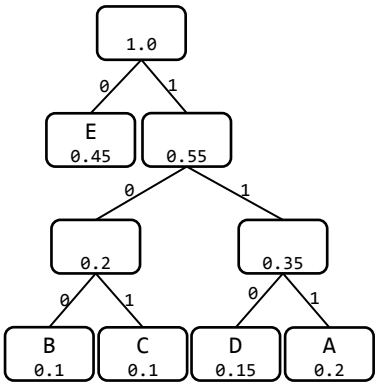
Notice how the left branch containing the Ⓑ node is assigned the value `0` and the right branch containing the Ⓒ node is assigned the value `1`. This process is continued. In the next round, the smallest values are Ⓐ and Ⓓ. They are combined into a tree with the resulting frequency of `0.35`.



Now we are down to three binary trees. Clearly the first and second trees in the list have the smallest frequency so we combine them in the next step.



With two binary trees in our list, only a single iteration is required to complete our binary tree.



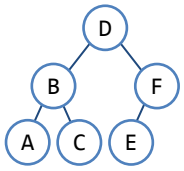
From here, we can derive our Huffman Codes. Notice how the numbers `0` and `1` are placed on each edge. By convension, we always put `0` on the path to the left child and `1` on the path to the right. The path from the root to E follows the `0` edge so the resulting Huffman Code for E is `0`. The path from the root to B is {`1, 0, 0`} so the resulting Huffman Code for B is `100`. This continues for all the leaves in the binary tree:

Token	Huffman Code
A	`111`
B	`100`
C	`101`
D	`110`
E	`0`

Examples

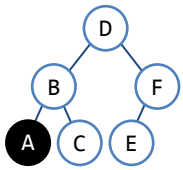
Example 08.1

In the following binary tree, perform an in-order traverse:



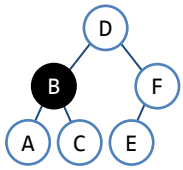
Answer:

```
void displayLVR(const Node <T> * pHead)
{
   if (pHead == NULL)
      return;

   displayLVR(pHead->pLeft);
   cout << pHead->data;
   displayLVR(pHead->pRight);
}
```



With the initial call of `displayLVR(D)`, we immediately call `displayLVR(B)` which calls `displayLVR(A)`. Since there is no left child. We display the contends of node A. The callstack is:

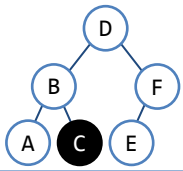
`displayLVR(D)` → `displayLVR(B)` → `displayLVR(A)`



After having completed `displayLVR(A)` so we return out. The callstack is now:

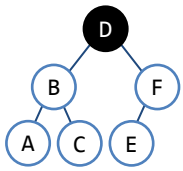
`displayLVR(D)` → `displayLVR(B)`

Since we have already outputted the left child, we can display the contents of B.



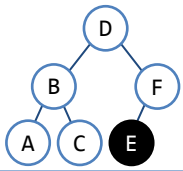
After displaying `B`, we call `displayLVR(C)`. The callstack is:

`displayLVR(D)` → `displayLVR(B)` → `displayLVR(C)`



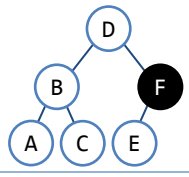
After having completed `displayLVR(C)` so we return out. Now we have also completed `displayLVR(B)` so we return out. This brings us to the root. The callstack is:

`displayLVR(D)`



Having displayed the contents of `D`, we not explore the right path by calling `displayLVR(F)`. This then causes us to explore the left child. The callstack is:

`displayLVR(D)` → `displayLVR(F)` → `displayLVR(E)`

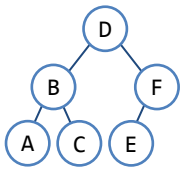


After having completed `displayLVR(E)` and since E has no children, we return back to the parent. The callstack is:

`displayLVR(D)` → `displayLVR(F)`

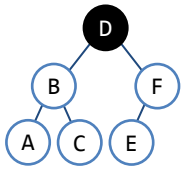
Example 08.2

In the following binary tree, perform an pre-order traverse:



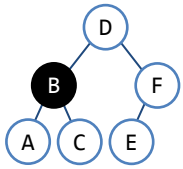
Answer:

```
void displayVLR(const Node <T> * pHead)
{
   if (pHead == NULL)
      return;

   cout << pHead->data;
   displayVLR(pHead->pLeft);
   displayVLR(pHead->pRight);
}
```



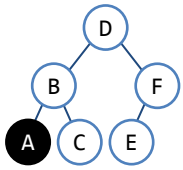
With the initial call of `displayLVR(D)`, we display the contents of the node before we call the left or right branch. The callstack is:

`displayLVR(D)`



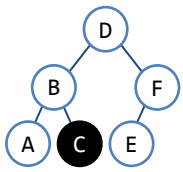
After having displayed the root node, we immediately go to the left. This will first display the value `B` before calling the left or right branch. The callstack is now:

`displayLVR(D)` → `displayLVR(B)`



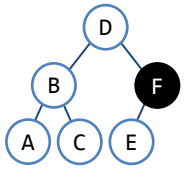
After having displayed the `B` node, we immediately go to the left. This will first display the value `A` before looking to the left or right. The callstack is now:

`displayLVR(D)` → `displayLVR(B)` → `displayLVR(A)`



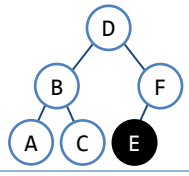
After `A` has been displayed and since there are no children under A, we return back to B. This sends us down the right side to C. Once there, we immediately display the contents of C. The callstack is now:

`displayLVR(D)` → `displayLVR(B)` → `displayLVR(C)`



Since C has no children, we are finished with C. Since C was the right child of B, we are finished with B and return to D. Now we explore the right side of D. The callstack is now:

`displayLVR(D)` → `displayLVR(F)`



After having completed `displayLVR(F)` we explore the left side of F. This brings us to E. The callstack is:

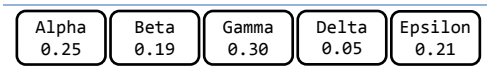
`displayLVR(D)` → `displayLVR(F)` → `displayLVR(E)`

Example 08.3

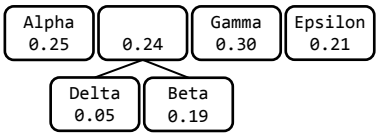
Find the Huffman codes for the following frequency table:

Token	Frequency
Alpha	25%
Beta	19%
Gamma	30%
Delta	5%
Epsilon	21%

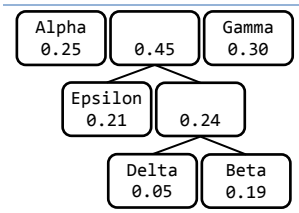
Answer:



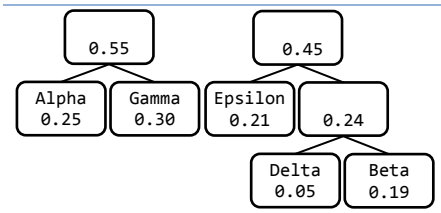
The initial state is to create five binary trees, each with a single node.



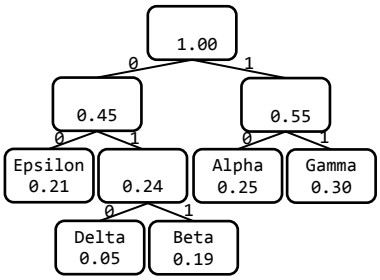
Next we combine the two trees with the smallest frequency value. In this case it is Beta and Delta. The smaller goes to the right. The new node in Beta's place has no name but the frequency is the sum of the children.



Now the smallest trees are `0.21` and `0.24`. These go to the right-most place (`0.24`'s slot). Notice how the entire `0.24` tree became a child of the newly created `0.45` node.



The next two smallest trees are Alpha and Gamma. They are combined to form one tree. Again, we are sure to put Alpha to the right of Gamma because it has a smaller value (`0.25` as opposed to `0.30`).



The final step is to combine the two binary trees into one. Again special care needs to be taken to ensure that the smaller of the two values is on the left.

Notice how there are 0's and 1's placed beside each edge on this graph. These are for illustration purposes only to make it easier to visualize how to derive the Huffman codes from the binary tree.

The path from the root to a given leaf node describes the Huffman code.

Token	Frequency
Alpha	`10`
Beta	`011`
Gamma	`11`
Delta	`010`
Epsilon	`00`

Exercises

Exercise 08.1

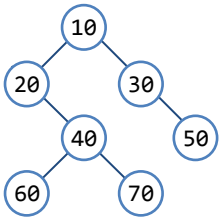
Identify the name for each of the following binary tree definitions:

- An element representing a single value_____
- A finite collection of connected elements_____
- A connection between elements in a tree _____
- A node having no incoming arcs but from which every other node can be reached _____
- The collection of arcs leading from one element to another_____
- When one element has an arc pointing to a second, this is the name of the first element_____
- When one element has an arc pointing to a second, this is the name of the second _____
- Two children sharing the same parent _____
- A special type of tree where each element has zero, one, or two children_____
- A collection consisting of a node and all the descendants of the node _____
- A node that can be reached by following a path from another node_____
- A node that is a parent, a parent's parent, and so on _____
- Nodes with no outgoing arcs _____
- A tree where all the levels are filled in_____
- A tree where the number of anscestors for each leaf are different than no more than one _____

Please see page 195 for a hint.

Exercise 08.2

Identify the nodes from the following tree that meet the specified criteria:

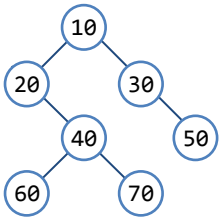


a. Root(s)	
b. Leaf(s)	
c. Ancestor(s) of node 20	
d. Ancestor(s) of node 50	
e. Ancestor(s) of node 10	
f. Descendant(s) of node 30	
g. Descendant(s) of node 40	
h. Descendant(s) of node 10	
i. Sibling of node 70	
j. Sibling of node 40	

Please see page 195 for a hint.

Exercise 08.3

Answer the following questions about this graph:

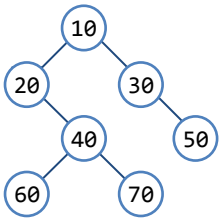


a. Is this graph a tree?	
b. What is the height of node 10	
c. What is the height of node 50	
d. What is the height of node 70	
e. Is this balanced?	
f. Is this complete?	

Please see page 195 for a hint.

Exercise 08.4

List the nodes in order corresponding to the following traversals:



a. In-order	
b. Pre-order	
c. Post-order	
d. Backwards	
e. Level-order	

Please see page 195 for a hint.

Exercise 08.5

Write out the state of the the binary tree after each line is executed

	state

`{`
`  BNode <int> * b;`
`  b = new BNode <int>(50);`	

`  addLeft(b, 30);`	
`  addRight(b, 70);`	
`  addLeft(b->pLeft, 20);`	
`  addLeft(b->pRight, 60);`	

`  BNode <int> * b2;`
`  b2 = new BNode <int> (90);`
`  addRight(b2, 95);`
`  addLeft(b2,85);`	

`  addRight(b->pRight, b2);`	
`  addRight(` `  b->pRight->pHead->pLeft, 40)`	

`  cout << "b: "`
`  << sizeTree(b);`
`  << \nb->pLeft: "`
`  << sizeTree(b->pLeft)`
`  << endl;`
`}`	

Please see page 195 for a hint.

Exercise 08.6

Consider the following letters and their associated frequencies:

Letter	Frequency	Code
α	48	
β	7	
γ	9	
δ	26	
ε	4	
θ	6	

Compute the Huffman code

Answer:

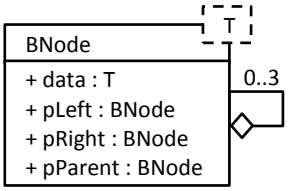
0.	
1.	
2.	
3.	
4.	
5.	

Please see page 209 for a hint.

Problems

Problem 08.1

Create a class to capture the notion of a single node in a binary tree. There will be four member variables: a pointer to the node to the right child (`pRight`), a pointer to the node to the left child (`pLeft`), a pointer to the parent (`pParent`), and the data associated with the node. Of course, any data-type will need to be supported, so your class will be a template class. It will need to be defined in its own header file (`bnode.h`). The UML class diagram of our binary node class/structure is the following:



Please also create the following functions: add a left node, add a right node, delete the binary tree, copy a binary tree, and display the tree using an in-order traversal:

```
void addLeft( BNode <T> * pNode, BNode <T> * pAdd);
void addRight(BNode <T> * pNode, BNode <T> * pAdd);
void addLeft( BNode <T> * pNode, const T & tAdd) throw (const char *);
void addRight(BNode <T> * pNode, const T & tAdd) throw (const char *);

void deleteBTree(BNode <T> * pNode);
BNode <T> * copyBTree(const BNode <T> * pSrc) throw (const char *);
std::ostream & operator << (std::ostream & out, BNode <T> * rhs);
```

Please see page 201 for a hint.

Problem 08.2

Write a program to generate Huffman Codes. Start by reading the tokens and their associated frequencies from a file. From this, display the corresponding Huffman Codes:

```
Enter the filename containing the value frequencies: huffman.txt
A = 111
B = 100
C = 101
D = 110
E = 0
```

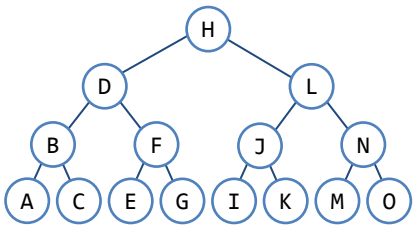
Please see page 209 for a hint.

The Binary Search Tree (BST) is a special type of tree where each value in each node is ordered from left to right. In other words, every left node is less-than or equal to the parent node, and each parent node is less-than or equal to the right node. There really are no real-world analogies for the BST. In fact, it is not even an abstract data type. Instead, it is a data-structure we use to build several ADTs.

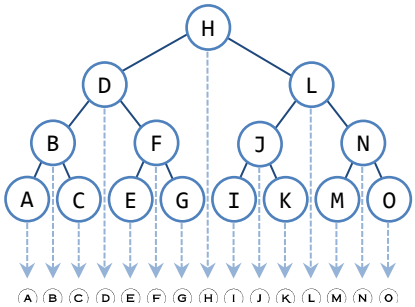
The essential properties of BSTs are the following:

- **Container**: The BST is designed to store a collection of data.
- **Search Access**: Elements can only be retrieved by searching from the root.
- **Order Sorted**: Elements in a BST are stored in sorted order.
- **Expandable**: The capacity of the BST can be increased to meet the needs of the client.
- **Restricted Insertion**: Elements added to the BST must be placed in sorted order.

The following is an example of a BST:



In the above figure, if one were to flatten the tree so all the nodes were in one horizontal line, the nodes would be in sorted order. Perhaps the following process best explains the ordering of a BST that makes it distinct from a standard binary tree.



If flattening a binary tree (otherwise known as in-order traversal or LVR traversal) results in the elements being in sorted order, then the tree is a BST. In other words, for every node in the tree, the value of the left child (`n.left.data`) is less than or equal to the value of the node (`n.data`) which in turn is less than or equal to the value of the right child (`n.right.data`). We will call this the BST constraint.

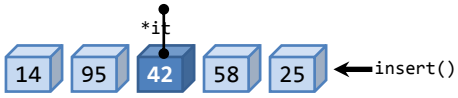
We should be able to create a BST out of any data-type that understands the < operator (so we can maintain the order of the tree) and the = operator (so we can copy a value into the tree).

Definition

A BST is a class that encapsulates a binary tree that maintains the BST constraint. The operations performed on a BST are the following:

Name	Example	Description
Create	`BST <int> b;`	Create an integer BST.
Assignment	`b1 = b2;`	Make a copy of the BST.
Empty	`f = b.empty();`	Returns `true` if the BST has no elements, `false` otherwise.
Size	`n = b.size();`	Returns the number of elements currently in the BST.
Clear	`b.clear();`	Remove all elements from the BST.
Begin	`it = b.begin();`	Returns an iterator to the beginning of the BST.
End	`it = b.end();`	Returns the NULL iterator.
Insert	`b.insert(e);`	Insert an element into the BST.
Find	`it = b.find(e);`	Find a given element in the BST. Return an iterator to the found element.
Erase	`b.erase(it);`	Erase one element from the BST as specified by the iterator `it`.

You may notice that these operations are much like the set. This is true. In fact, the set container in the STL is implemented as a BST.



Create

There are two constructors in the BST class: a default constructor and the copy constructor. In the first case, no nodes are added to the tree. In the second, a copy of every node in the RHS is created for the new tree.

```
BST <T> :: BST();
BST <T> :: BST(const BST <T> & rhs) throw (const char *);
```

Observe how the default constructor does not throw. This should not be surprising because nothing is allocated. The copy constructor, on the other hand, can throw because nodes are created.

	Code	Description
1	`BST <int> b1;`	Create an empty BST of integers.
2	`BST <string> b2;`	Create an empty BST of strings. Again, no nodes are created.
3	`BST <int> b3(b2); // ERROR!`	Compile error. Note that `b1` is a BST of integers and `b2` is a BST of strings.
4	`BST <int> b4(b1);`	Copy the nodes and structure from `b1` into `b4`. Note that `b1` is empty so we still have an empty tree.

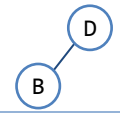
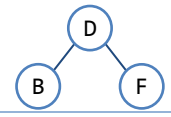
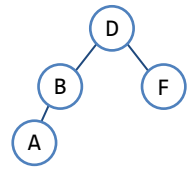
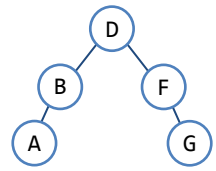
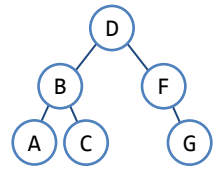
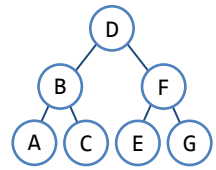
The default constructor takes constant time O(1) while the copy constructor takes linear time O(n) with respect to the size of the RHS.

Insert

The `insert()` method will find the appropriate location in the BST for the new element. In other words, it will search for the correct location in the BST to place the new element.

```
void BST <T> :: insert(const T & t) throw (const char *);
```

Notice how `insert()` can throw if there is not sufficient resources to allocate a new node.

	Code	Description	State
1	`BST <char> b;`	Create an empty BST of characters.	
2	`b.insert('D');`	Place the letter `D` at the root of the tree.	
3	`b.insert('B');`	Place the letter `B` in the tree. Since `B < D`, this new node will be to the left of `D`.	
4	`b.insert('F');`	Place the letter `F` in the tree. Since `F > D`, this new node will be to the right of `D`.	
5	`b.insert('A');`	From the root, we notice that `A < D` so we move down the left side to `B`. Since `A < B`, we add a new node to the left of `B`.	
6	`b.insert('G');`	From the root, we compare `G > D` so we move down the right side to `F`. Since `G > F`, we add a new node to the right of `F`.	
7	`b.insert('C');`	Since `C < D`, we move to `B`. Since `C > B`, we add a node to the right of `B`.	
8	`b.insert('E');`	Since `E > D`, we move to `F`. Since `E < F`, we add a node to the left of `F`.	

If the tree is balanced, then the maximum height of the tree (distance from the root to the leaf) is log n. To insert a node, we need to navigate to an empty spot in the tree where the new value would go. In the worst case, this will require navigating all the way to a leaf. This is O(log n). Since the cost of allocating a new node is O(1), the total cost is O(log n).

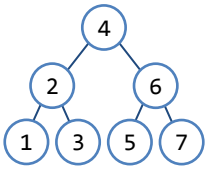
Find

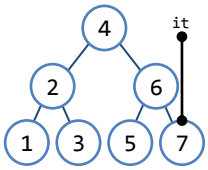
The `find()` method will return an iterator pointing to the element to be found.

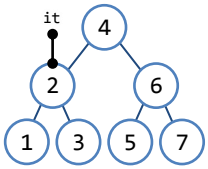
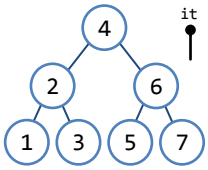
```
BST <T> :: iterator BST <T> :: find(const T & t);
```

Note that if the node is not found, then the NULL iterator is returned.

	Code	Description	State

1	`BST <int> b;`
	`b.insert(4);`
	`b.insert(2);`
	`b.insert(6);`
	`b.insert(1);`
	`b.insert(3);`
	`b.insert(5);`
	`b.insert(7);`
		A BST will be created with the values 1-7. Note how the order is preserved from left with 1 to right with 7.	

2	`BST <int> :: iterator it;`
	`it = b.find(7);`
	`assert(*it == 7);`
		The iterator will point to the node containing the value 7 in the BST. It will then be possible to access the value in the node with the dereference operator.	

3	`it = b.find(2);` `assert(*it == 2);`	Regardless of where the value may be located in the tree, the `find()` method will return an iterator to it.	
4	`it = b.find(10);` `assert(it == b.end());`	If the value is not to be found in the tree, then the `end()` iterator is returned.	

Note that changing the value in a BST node does not guarantee that the resulting tree conforms to the BST constraint. In other words, if we change the value in step 2 to `*it = 0`, then the binary tree would not be a BST anymore.

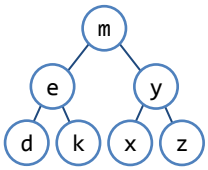
Erase

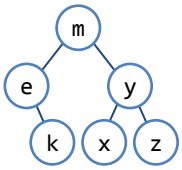
The `erase()` method will remove the `BNode` in the BST referred to by the passed iterator.

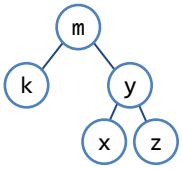
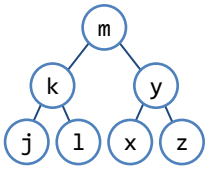
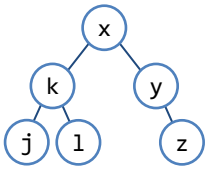
```
void BST <T> :: erase(BST <T> :: iterator & it);
```

Note that the resulting tree must maintain all the BST properties: it must be a valid binary tree and adhere to the BST constraint. This means the BST may need to be re-organized when nodes are erased.

	Code	Description	State

1	`BST <char> b;`
	`b.insert('m');`
	`b.insert('e');`
	`b.insert('b');`
	`b.insert('k');`
	`b.insert('y');`
	`b.insert('x');`
	`b.insert('z');`
		A BST will be created with letters. Notice how the sorted order is enforced even when there are gaps in the sequence.	

2	`BST <char> :: iterator it;`
	`it = b.find('d');`
	`b.erase(it);`
		The leaf node containing `d` is removed from the BST. Notice how the tree without the `d` node remains a valid BST.	

3	`b.erase(b.find('e'));`	This case is more complicated. The `e` node has one child: the `k` node. In this case, the `k` node will take `e`'s place in the BST.	
4	`b.insert('j');` `b.insert('l');`	Add a few nodes to set us up for the next case.	
5	`b.erase(b.find('m'));`	This next case is more complex than the previous: Since our children each have two children, we cannot simply put a child in our place. In this case, the in-order successor will be found (`x`). We will learn how this works in the Implementation part of the reading.	

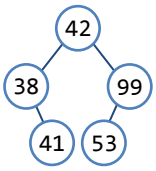
BST Iterator

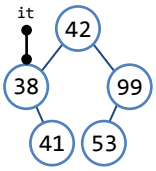
The BST must provide iterators to traverse the binary tree or there would be no way to retrieve all the contained elements. This requires the BST to support several interfaces:

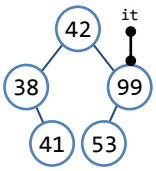
```
BST <T> :: iterator BST <T> ::  begin();
BST <T> :: iterator BST <T> :: rbegin();

BST <T> :: iterator BST <T> ::  end();
BST <T> :: iterator BST <T> :: rend();
```

Recall that `begin()` returns the first element in the BST and `end()` returns the first slot off the end of the BST. This also means that the actual element can be retrieved from the iterator using the dereference operator. Recall that the "`r`" version looks at the BST backwards. Thus, `rbegin()` points to the last element in the BST.

	Code	Description	State

1	`BST <int> b;`
	`b.insert(42);`
	`b.insert(99);`
	`b.insert(53);`
	`b.insert(38);`
	`b.insert(41);`
		Create a BST with five nodes.	

2	`BST <int> :: iterator it;`
	`it = b.begin();`
	`assert(*it == 38);`
		The `begin()` method points to the node with the lowest value. In this case, the lowest value is 38.	

3	`it = b.rbegin();` `assert(*it == 99);`	The reverse iterator points to the last element in the sequence. Another way of saying this is the right-most node.	

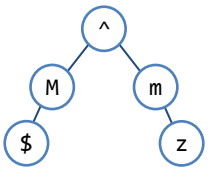
Standard Container Interfaces

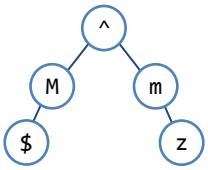
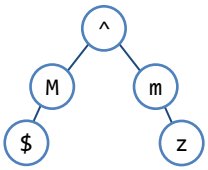
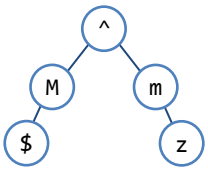
The BST container supports the usual interfaces:

```
BST <T> & BST <T> :: operator = (const BST <T> & rhs) throw (const char *);
bool      BST <T> :: empty() const;
int       BST <T> :: size()  const;
void      BST <T> :: clear();
```

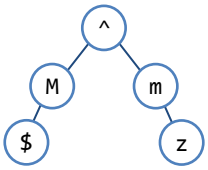
The standard container interfaces work as one would expect.

	Code	Description	State

1	`BST <char> b1;`
	`b1.insert('^');`
	`b1.insert('M');`
	`b1.insert('m');`
	`b1.insert('$');`
	`b1.insert('z');`
		Since BSTs are sorted alphabetically, the ASCII value for characters determines the order of inserted nodes.	

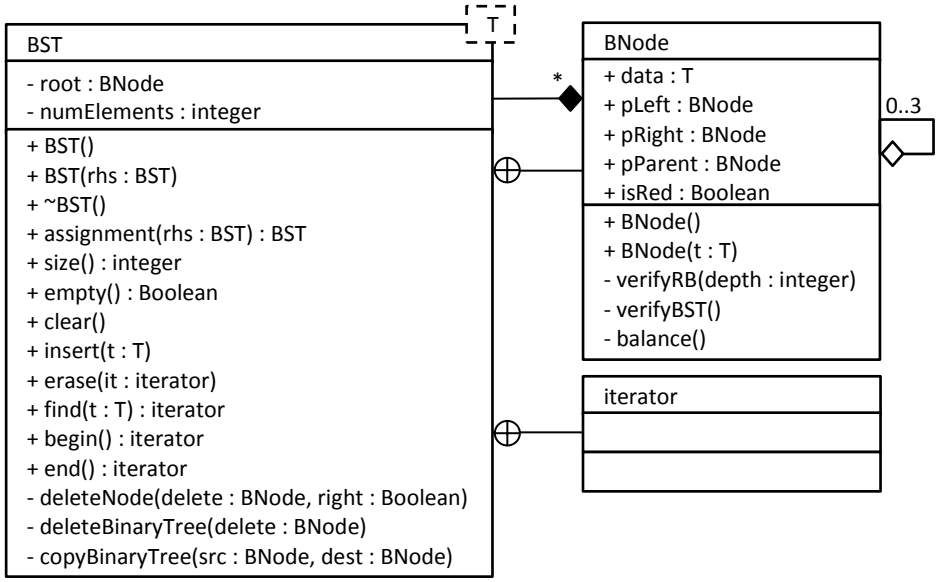
2	`assert(b1.size() == 5);`	There are five nodes in the BST.	
3	`assert(b1.empty() == false);`	This BST is not empty.	
4	`BST <char> b2;` `b2 = b1;`	Make a copy of `b1` into `b2`. This could just as easily have been done with the copy constructor of course.	
5	`b1.clear();`	The tree `b1` is removed of all nodes.	

6	`assert(b1.size() == 0);`
	`assert(b1.empty() == true);`
	`assert(b2.size() == 5);`
	`assert(b2.empty() == false);`
		The `b1` tree should have no nodes and be empty. However, the `b2` tree should still contain five nodes and not be empty.	

7	`b1 = b2;`	It should be possible to copy nodes back onto a BST at any time using the assignment operator.	

Implementation

As the list class encapsulated the linked list with a standard container interface, the BST class encapsulates a binary tree with a standard interface.



The UML class diagram of the `BST` should look somewhat familiar: it is basically the same as that of the `list` class from Chapter 7. In other words:

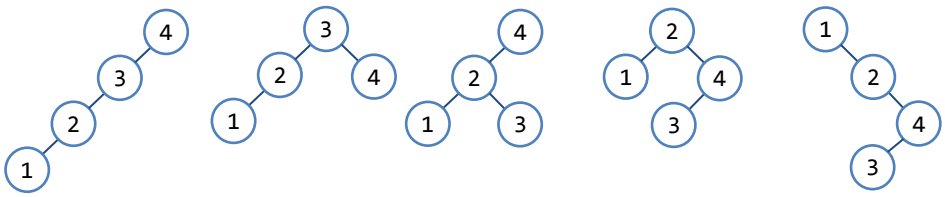
- `iterator` is a nested class within `BST` but `BST` does not contain a reference to iterator.
- `BNode` is a nested class within `BST` and `BST` contains an unspecified number of references to `BNode`. When the `BST` is destroyed, then all references to `BNode` are destroyed (as indicated by the closed diamond ♦ symbol).
- `BNode` refers to up to three other `BNodes`, but destroying one `BNode` does not necessitate destroying all the rest of the `BNodes` (as indicated by the open diamond ◇ symbol),

For a review of the UML class diagram notation used above, please see Appendix D. Review: UML Class Diagram. Note that there are a few private methods:

- `BST::deleteNode()` : Useful for deleting a specific `BNode` from the tree. Called from `erase()`.
- `BST::deleteBinaryTree()` : Recursive, called directly `clear()`.
- `BST::copyBinaryTree()` : Recursive, called from the assignment operator.
- `BNode::verifyRB()` : Recursive, useful for verifying property d) of a red-black tree.
- `BNode::verifyBST()` : Ensures that the subtree under a given node is a valid `BST`.
- `BNode::balance()` : Contains all the red-black balancing logic.

Unbalanced Trees

Unfortunately, the problem is a bit more complex than this. Consider four numbers {1, 2, 3, 4} to be inserted into a BST. From these numbers, several trees are possible which all honor the BST constraint:



Note that while all of these are valid BSTs, they do not have the same performance characteristics. The tree on the left, for example, never makes use of the `pRight` pointer. In fact, it looks more like a linked list than a BST. The cost of finding a given element is O(n) rather than the O(log n) we desire.

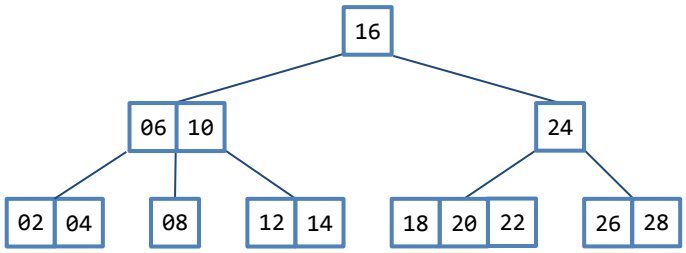
In order for a BST to perform to its potential, it must be mostly balanced. A balanced BST is one where the length of the path from a leaf node to the root node is roughly the same no matter which leaf is chosen.

2-3-4 Tree

One method to keep a binary tree balanced is a 2-3-4 tree. This may seem paradoxical because the 2-3-4 tree is not even a binary tree! However, after we see how 2-3-4 trees maintain a balanced state, we will learn how to adapt the same technique to binary trees.

A 2-3-4 tree is an *m*-ary tree, a tree where each node can have up to *m* children. Note that a binary tree is also an *m*-ary tree where *m*=2. In the case of a 2-3-4 tree, each node can have up to 4 children (*m*=4). Thus there can be four pointers below each node.

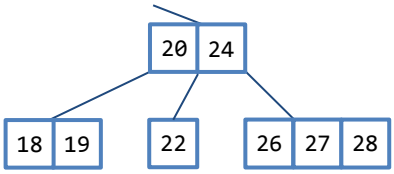
Each node in a 2-3-4 tree can have up to three values, each arranged from smallest to largest. The sub-tree under the left-most pointer of a given node contains only values less than or equal to the first value in the node. The sub-tree under the next pointer is greater than or equal to the first value of the node but less than or equal to the second value of the node. This continues for the other values in the node. In other words, the 2-3-4 has the same structure as a BST, it is just not a binary tree.



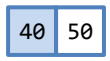
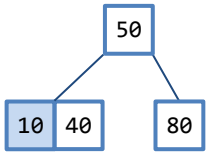
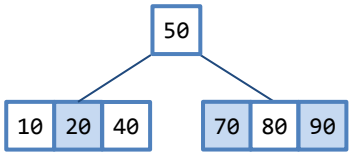
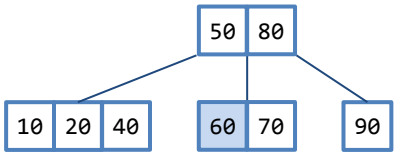
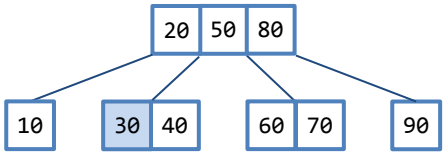
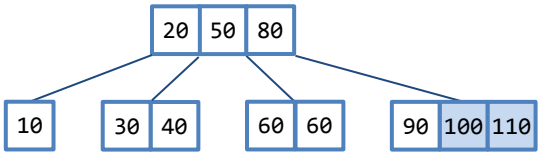
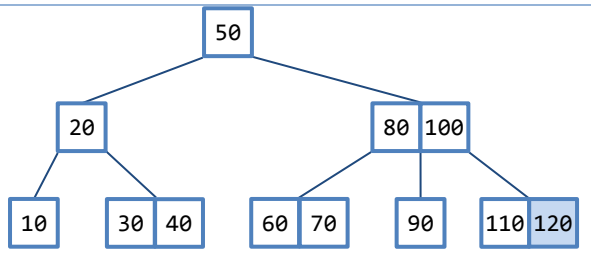
The 2-3-4 tree has the additional property of being a complete tree. This means the path from the leaf to the root is always the same length.

What would happen if the value `27` were added to this tree? Since `27` is greater than `16` it would go on the right side of the root. Since it is greater than `24` it would go to the right side of that node. Finally, it would be added to the `26 28` node to become `26 27 28`.

What would happen when `19` is added to this tree? Since `18 20 22` is already full, it would seem that one would have to add a leaf. However, you cannot add a new node to one leaf without adding one to all leafs. This would require moving a value from `18 20 22` into `24` and creating a new leaf out of that.



Notice how a 2-3-4 tree is roughly balanced. Since the tree must be complete, the most lopsided the tree would be when all the nodes on one side have three values while the nodes on the other side have only one. Perhaps it is best to explain a 2-3-4 tree by building one from scratch.

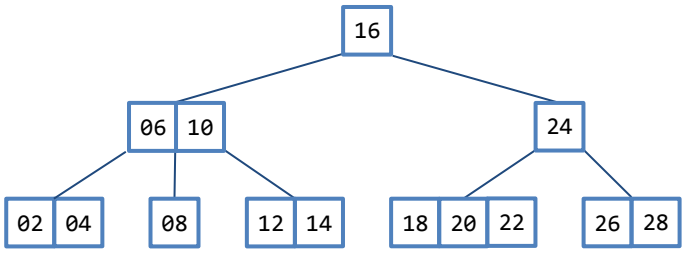
	Insert	Description	State
1	50	One node is created with one value.	
2	40	Another value is added to the node.	
3	80	There is still room for another value.	
4	10	There is no room for another value so we need to split our node. First the 40 50 80 node becomes three with 50 at the root, then 10 is added to the left.	
5	20 70 90	Adding these three values fills the two child nodes but does not impact the overall structure of the 2-3-4 tree.	
6	60	Adding 60 to the full 70 80 90 node causes the node to split. Thus 80 gets raised to the root leaving 70 and 90 as its two children. After this, 60 is added.	
7	30	Adding 30 to the full node 10 20 40 causes it to split, sending 20 to the parent. Then 30 is added to 40. Note how the root is now full.	
8	100 110	We will now fill up the right node of 90 to become 90 100 110. Notice how we always add values to the bottom of the tree.	
9	120	Values are added to the leaves of 2-3-4 trees. As we try to add 120 to 90 100 110, we discover that node is full. We then send 100 to the parent. The parent is full also. This requires us to split the parent, which sends 50 to be our new root.	

From this example, we can see a few things:

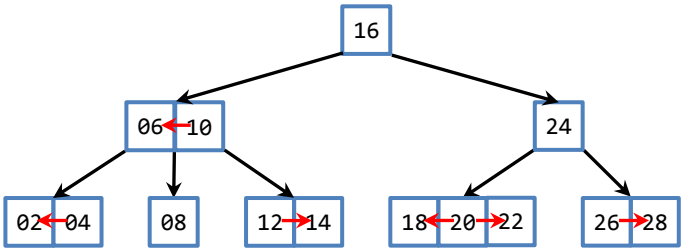
- Elements are always added to the bottom or leaf nodes of the tree. We never add an element to the root unless the root also happens to be the leaf.
- When a node is full, two things happen. First, the middle value is sent up to the next level. This in turn may require that node (the parent of the previously full node) to the split and so on. After all this has happened, then the value is added to the newly split leaf node. Again, values are always added to the bottom.
- A node can have 1, 2, or 3 values.
- A node can have 0, 2, 3, or 4 children. This is where the 2-3-4 name comes from.
- If a node is a leaf, all its siblings and cousins are also leafs. In other words, each level is always completely full.

Red-Black Tree

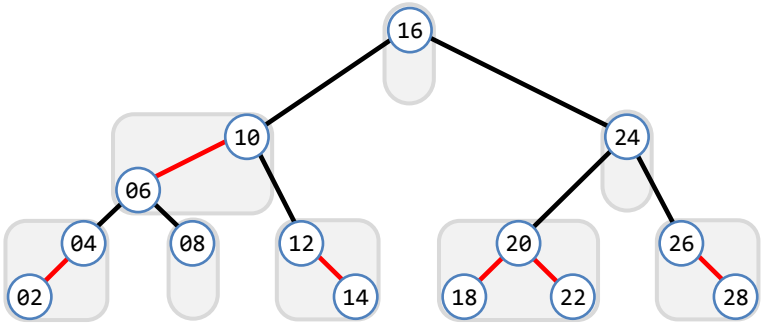
A Red-Black tree is a binary representation of a 2-3-4 tree. Recall that a 2-3-4 tree is a tree with between 1 and 3 values in each node, each node having either 0, 2, 3, or 4 children.



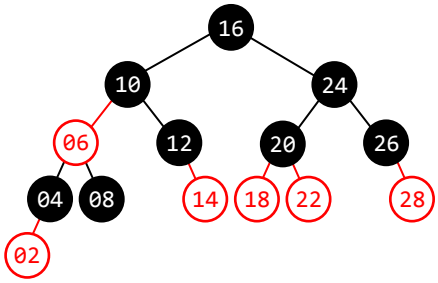
There are two types of links between each of the adjacent numbers: a line from the parent node to the child node (black) and the link between numbers within a node (red). If we were to add the red links to this graph, we would get:



From here, we can see that there are no nodes having more than two children. Thus we can create a binary search tree representation of the above 2-3-4 tree:



Any 2-3-4 tree can be represented as a binary search tree if we are able to distinguish black links (*between* 2-3-4 nodes) from red links (between elements *within* a given 2-3-4 node). Now if we color the node itself with the color of the parent link, we get the Red-Black tree:



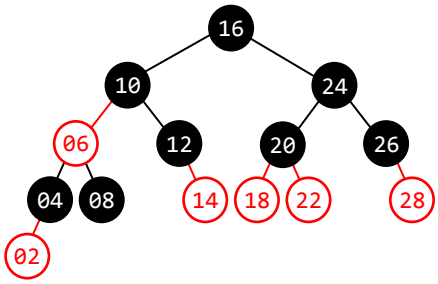
Normally we just color the nodes of a Red-Black tree red, not the links as well. The above illustration shows the links colored red so the relationship between the Red-Black tree and the 2-3-4 tree can be more readily seen. In other words, the color of a node is the color of the link leading into the node.

Four properties

Red-Black trees have four properties:

a) **Red or Black**: Every node is either red or black.
b) **Black Root**: The root of every Red-Black tree is black. This corresponds to the center element of a 2-3-4 tree.
c) **Red Nodes have Black Children**: If a node is red, then both the children are black. In a 2-3-4 tree, this means that every node points to the center of the node below. You cannot have a red node point to another red node. If this were possible, then there would be more than 3 elements in a single 2-3-4 node.
d) **Same Number of Black Nodes**: The path from every leaf to the root contains the same number of black nodes. Since there is exactly one black node in every corresponding 2-3-4 node, this means that every 2-3-4 tree has the same number of nodes in every path from a leaf to the root. This is the rule that ensures we have a balanced tree.

The trick to maintaining a balanced Red-Black tree is to maintain the four properties when nodes are inserted and deleted. Does the following tree follow all four Red-Black rules?



Rule a) is satisfied because all the nodes are colored with red or black.

Rule b) is satisfied because the root (`16`) is black.

Rule c) is satisfied because all the red nodes {`02`, `06`, `14`, `18`, `22`, `28`} have either no children, one black child, or two black children. Thus, no red node has a red child.

Rule d) is satisfied because there are exactly 3 black nodes connecting every leaf to the root (`16`). These leaves are {`02`, `08`, `14`, `18`, `22`, `28`}.

Insert

How does a Red-Black tree maintain balance when nodes are inserted? New nodes are colored red when first inserted into the tree. If one of the four rules is violated, then the tree needs to be fixed. To simplify discussion, we will use the following shorthand:

Symbol	Meaning
N N	The new `BNode`
P P	The Parent of the new `BNode` N
S S	The Sibling of the new `BNode`. This is the parent's other child.
G G	The Grandparent of N or the parent of P
A A	The Aunt of the new `BNode`. This is the parent's sibling.

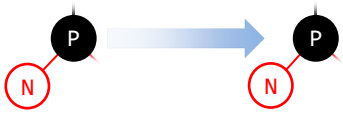
There are four cases when inserting into a Red-Black tree.

Case 1: No parent

When the new node is the root, the color is black. This corresponds directly to Property b) of the Red-Black tree.

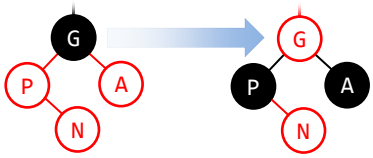
Case 2: Parent is black

If the new node's parent is black P, then there is nothing left to do; all the properties of the Red-Black tree are maintained when adding a red node to a black parent.



Case 3: Parent and aunt are red

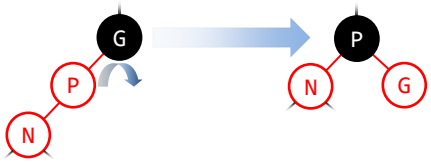
If the new node's parent is red P, grandparent is black G, and aunt is red A and exists, then we re-color the three. The grandparent becomes red, the parent becomes black, and the aunt becomes black. Note that we may need to do more work if the great-grandparent is red.



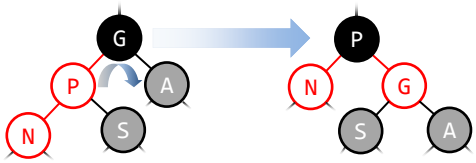
Case 4: Parent is red, aunt is black or non existant

If the new node's parent is red (P), grandparent is black (G), and aunt is black (A), we need to rotate. The parent becomes black and takes the grandparent's place with two children: the new node and the grandparent. The grandparent becomes red and has one of the parent's children as a child and the aunt as the other child. There are actually four sub-cases here:

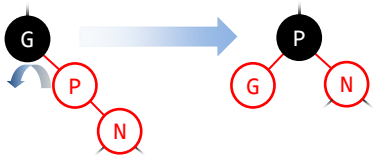
a) (N) is the left child of (P) and (P) is the left child of (G): In this case, we want to right-rotate. This will put the parent in grandmother's place.



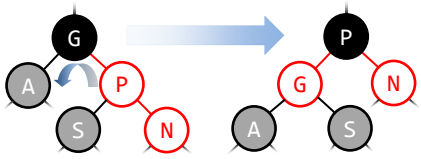
Note that we may have an aunt and/or a sibling. They must be redistributed.



b) (N) is the right child of (P) and (P) is the right child of (G): In this case, we want to left-rotate. This is exactly the opposite of a). where we right-rotated:



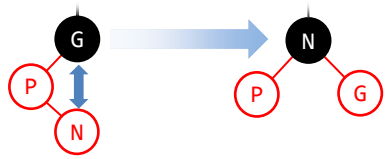
As with case a), we may have an aunt and/or a sibling.



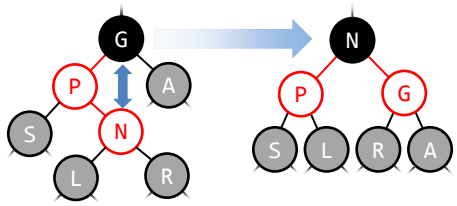
The pseudocode for case b) is:

```
IF pParent.isRed() AND pGranny.isBlack() AND
   pSibling.isBlack() AND pAunt.isBlack() AND
   pParent.pRight = pNew AND pGranny.pRight = pParent

      pParent.addLeft(pGranny)
      pGranny.addRight(pSibling)
      pHead ← pParent

      pGranny.setRed()
      pParent.setBlack()
```

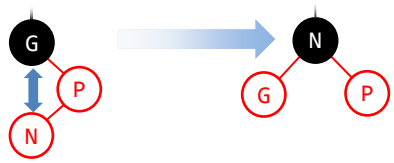
c) N is the right child of P and P is the left child of G: First, we need to put N, and P and G in order. This will mean swapping N and G.



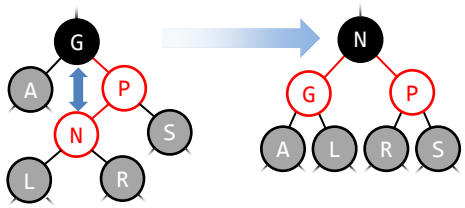
It is a bit more complex than it may seem because we must distribute N's children if there are any (the left child or L and the right child or R) to P and G.



d) N is the left child of P and P is the right child of G: This is exactly the opposite of case c).



Of course, there may be children to re-distribute as well:



The pseudocode for case d) is:

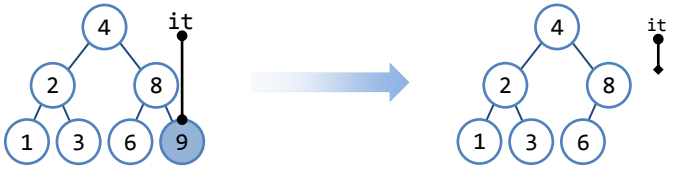
```
IF pParent.isRed() AND pGranny.isBlack() AND
   pSibling.isBlack() AND pAunt.isBlack() AND
   pParent.pLeft = pNew AND pGranny.pRight = pParent

      pGranny.setRight(pNew.pLeft)
      pParent.setLeft(pNew.pRight)

      IF pGranny.pParent = NULL
         pNew.pParent ← NULL
      ELSE IF pGranny.pParent.pRight = pGranny
         pGranny.pParent.pRight ← pNew
      ELSE
         pGranny.pParent.pLeft ← pNew

      pNew.setLeft(pGranny)
      pNew.setRight(pParent)

      pGranny.setRed()
      pNew.setBlack()
```

Erase

Erasing a node from a BST is more difficult than inserting a node. There are three cases:

Case 1: No Children

The first case involves the erased node having no children. In this case, we can safely remove the node without making further alterations to the BST. Consider the following tree:

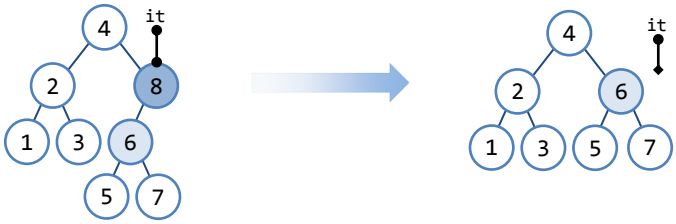


If we are to erase node 9, then we can simply delete the node. This involves causing the parent node to forget about the erased child and then freeing the affected node. The pseudocode for case 1 is the following:

```
IF it.pRight = NULL and it.pLeft = NULL
  IF it.pParent ≠ NULL AND it.pParent.pRight = it
    it.pParent.pRight ← NULL
  IF it.pParent ≠ NULL AND it.pParent.pLeft = it
    it.pParent.pLeft ← NULL
  DELETE it
```

Case 2: One Child

The second case involves the erased node having a single child. Note that there may be any number of descendants to this node. In other words, there may or may not be grandchildren to the erased node. In this case, the child node simply takes the place of the erased node. Consider the following tree.



In the above case, the erased node has no right children so it is guaranteed to be greater than the left node or any of the left node's children. This makes it safe to move the left child into the erased node's place. The same could be said if there were no left children and the right node was to be moved into the erased node's place. The pseudocode for case 2 is the following:

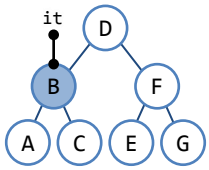
```
IF it.pRight = NULL and it.pLeft ≠ NULL
  it.pLeft.pParent ← it.pParent
  IF it.pParent ≠ NULL AND it.pParent.pRight = it
    it.pParent.pRight ← it.pLeft
  IF it.pParent ≠ NULL AND it.pParent.pLeft = it
    it.pParent.pLeft ← it.pLeft
  DELETE it

IF it.pLeft = NULL and it.pRight ≠ NULL
  it.pRight.pParent ← it.pParent
  IF it.pParent ≠ NULL AND it.pParent.pRight = it
    it.pParent.pRight ← it.pRight
  IF it.pParent ≠ NULL AND it.pParent.pLeft = it
    it.pParent.pLeft ← it.pRight
  DELETE it
```

The first IF statement deals with the case where there is no left child. The second IF statement deals with the case where there is no right child.

Case 3: Two Children

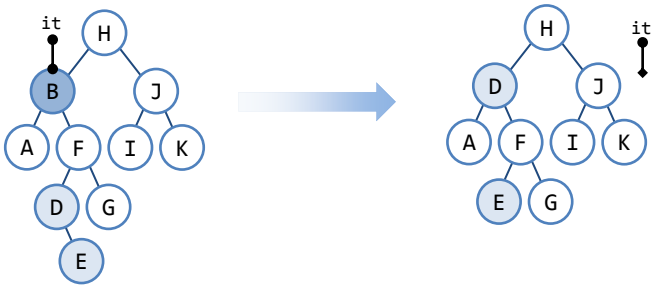
By far the most complex case is when the erased node has two children. In this case, we have two options: either replace the erased node with the in-order successor or to replace the node with the in-order predecessor.

The **in-order successor** is the node that follows the current node when performing an in-order traverse of the BST. Another way of saying this is that it is the next node in sorted order. Before we begin a discussion on how to find the in-order successor, consider the following: because the node has two children (otherwise we would be in Case 1 or Case 2), there must be at least one node that is greater than the erased node and there must be at least one node that is less than the erased node. Thus, there is guaranteed to be a successor and predecessor when we are in case 3. In the following BST, `A` is a predecessor and `C` is a successor.



The second thing to notice is that we are taking the very next node in succession. This means that there cannot be a node between the erased node and the successor. Thus if we replace the erased node with the in-order successor (or predecessor for that matter), the BST is guaranteed to remain valid.

The final thing to notice is that the in-order successor can have at most one child. In fact, if it has any children, it must only have a right-child. This fact is a little less intuitive than the other two. Notice that the in-order successor must be on a right-child descendent of the erased node. If it was a left-child, then it would be before the erased node in sorted order. Now if the in-order successor had a left child, then that child would have a smaller value than the in-order successor (because left-children are smaller in a BST) and it would have a larger value than than the erased node (because the in-order successor must be a right-descendent of the erased node). Consider the following BST:



In this case, the in-order successor is node `D`. If `D` had a left child (the value `C`), then that child would be the in-order successor. The existence of a right child to `D` does not factor into `D` being the in-order successor.

Once the in-order successor is found, there are two parts to the algorithm:

a) The in-order successor (`D` in the above example) is stripped of its right child (`E`) and placed in the tree where the erased node (`B`) used to preside. This means that the in-order successor (`D`) will now inherit both a right (`F`) and a left (`A`) child.
b) If the in-order successor (`D`) had a right child (`E`), it would take its parent (`D`) place in the BST. Any children of the right child would remain unaffected, keeping the children of the right child in their places.

The pseudocode for this case will be left as an exercise.

BST Iterator

Perhaps the most difficult iterator to implement of all the abstract data-types is the BST iterator. This is because one must traverse the binary tree to find the next or previous node in the iteration. A complicating aspect of this challenge is that we cannot use recursion. As you may recall from the previous chapter, one can perform an in-order (LVR) or backwards (RVL) traversal of a binary tree using recursion. In fact, it is almost trivial to do so if one fully internalizes recursion. However, recursion only works in certain circumstances.

Recursion works by storing the state of the iteration in the **call stack** (the collection of functions that were called before the current function was executed where the first function in the stack is `main()`). To illustrate this point, consider the following code:

```
void displayString(const char * text)
{
   // end condition
   if (*text == '\0')
      return;

   // display the string
   cout << *text;
   displayString(text + 1);
}
```

This function will display the contents of a c-string on the screen. The first time `displayString()` is called, the first letter in the c-string will be displayed. The second time, the second letter. This will continue until the null-character is encountered. Now the client makes the following function call:

```
displayString("Iterator");
```

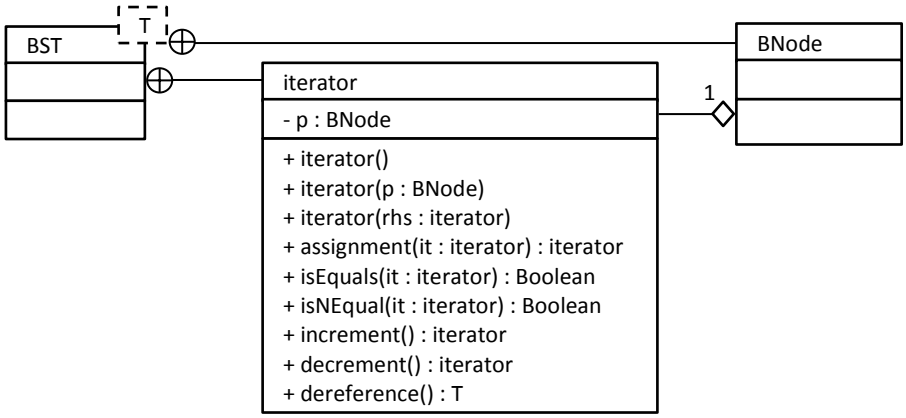
Notice how the letter `'a'` is the 5th element in the c-string. At the point in time when the letter `'a'` is called, the function `displayString()` will be called 5 times. Thus, there will be 5 instances of `displayString()` in the call stack.

At this point, you are probably wondering what this has to do with the BST iterator. The point is this: the state of the recursion is stored in the call stack. It is how `displayString()` *knows* how far it has come and what state it is in for the iteration. Iterators cannot store state in the call stack; it must store the state in a variable. In other words, all the information needed to store where in the iteration we are and all the information necessary to find the next element in the iteration must be stored in the member variables of the iterator class. Using recursion as in the `displayString()` example above, there is no one variable that stores the state. Instead, there are many variables: one instance of `text` for each instance of `displayString()` on the stack.

So how do we represent the state of our iteration in a BST iterator? We have two choices:

a) Keep a stack of nodes representing the path back to the root. This is essentially what the LVR traversal did in the recursive solution from Chapter 08, except it relied on the call-stack to maintain this state. For the iterator solution, we will need to maintain this stack ourselves.
b) Point to the current `BNode` in the iteration. From here, we can use the in-order successor algorithm to implement the `++` operator and the in-order predecessor algorithm to implement the `--` operator.

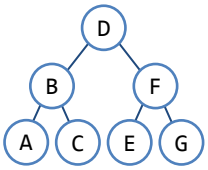
Because the second choice involves less size (a pointer is smaller than a stack of pointers), we will present that solution here. The UML class diagram of the BST iterator is the following:



Notice how the iterator maintains a pointer to a `BNode`, but it does not "own" the `BNode`. If the iterator was destroyed, the `BNode` would not be destroyed as a result. This means the relationship is **aggregation**.

Increment

The most complex method of the BST iterator class is the increment operator. Here, it is necessary to find the next node in the tree using the in-order successor algorithm. Consider the following BST:



There are several cases:

1. **Have a right child** (nodes `B`, `D`, and `F`): If the current node has a right child, then the in-order successor must be down that path. Notice that the left child must be smaller than the current node due to the fundamental property of the BST (left child ≤ node ≤ right child) and that the parent node must be less than the current node (if the current node is the right child of the parent as is the case with `F`) or greater than all the right children (if the current node is the left child of the parent as is the case with `B`). Therefore, the in-order successor must be down the right path. We can find the in-order successor by taking the first right and then following the path to the left as far as it will go. This is called "jig down/right":

```
IF pCurrent.pRight
  pCurrent ← pCurrent.pRight
  WHILE pCurrent.pLeft
    pCurrent ← pCurrent.pLeft
  RETURN pCurrent
```

2. **No right child and we are our parent's left child** (nodes `A` and `E`). In the case that we have no right child, then the next node in succession must be our parent if our parent is bigger than us.

```
IF pCurrent.pRight = NULL and pCurrent.pParent.pLeft = pCurrent
  RETURN pCurrent ← pCurrent.pParent
```

3. **No right child and we are our parent's right child** (nodes `C` and `G`). In the case that we have no right child and our parent is smaller than us, we need to find our parent's successor. In the case of `C`, we loop just once to get the `B`, then stop at `B`'s parent (`D`). In the case of `G`, we loop until the top of the tree then stop because we are done. This is indicated by the `NULL` pointer. Here, we "jig up/left:"

```
IF pCurrent.pRight = NULL and pCurrent.pParent.pRight = pCurrent
  WHILE pParent and pCurrent.pParent.pRight = pCurrent
    pCurrent ← pCurrent.pParent
  RETURN pCurrent ← pCurrent.pParent
```

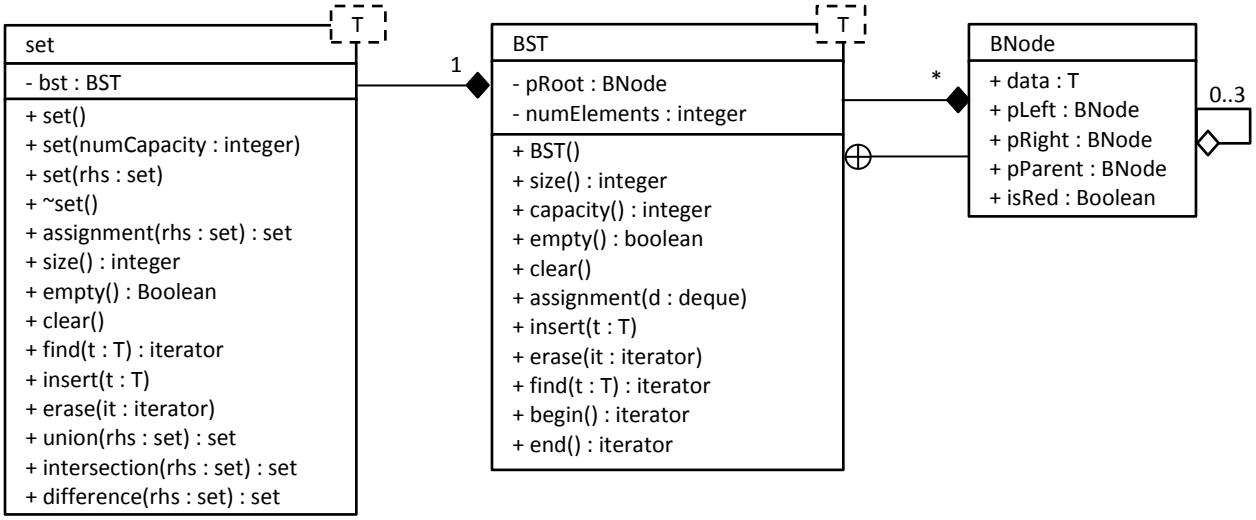
Application

There are two examples to be presented here: a BST implementation of the set and a sorting algorithm called the Binary Sort.

Set

In Chapter 05 : Set, we implemented the set ADT using an array. We discovered that we can realize a significant performance advantage in most scenarios when we keep the data sorted. However, it still took O(n) to insert a new element in the set. Using a BST as the underlying data-structure, we can do much better. We can have O(log n) insertion as well as O(log n) find performance.

By using a BST as the underlying data-structure, the UML class diagram of our improved set class is the following:



The first thing to notice is that the outwardly-facing interfaces (namely the public methods) are exactly the same regardless of the underlying data-structure. This is by design. One of the goals of encapsulation is to shield the user of a class from having to know about how the class is implemented. To demonstrate how this will work, a few of the critical methods of the set class will be implemented.

Insert

Recall that the interface for the the `insert()` method in the `set` class is the following:

```
void set <T> :: insert(const T & t) throw (const char *);
```

Fortunately, there is a corresponding method in the BST class which does exactly the same thing.

```
template <class T>
void set <T> :: insert(const T & t) throw (const char *)
{
   try
   {
      bst.insert(t);
   }
   catch (...)
   {
      throw "ERROR: Unable to allocate buffer";
   }
}
```

Observe how we catch the BST exception and throw a new one so the client cannot tell how the underlying data-structure is utilized.

Find

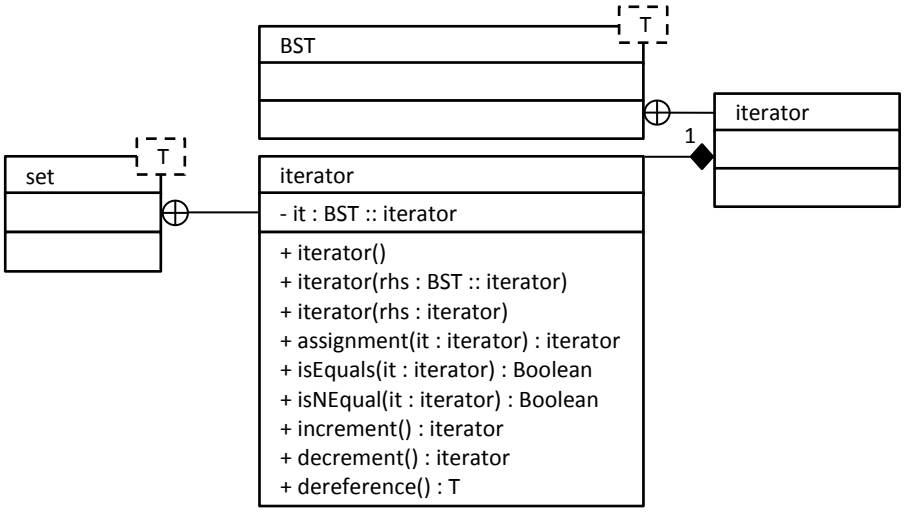
The find method in the set class will return a set iterator referring to the element in the set the client is searching for. The interface is defined as:

```
set <T> :: iterator <T> set <T> :: find(const T & t);
```

Again, we can leverage the BST's `find()` method to solve this problem. The only hurdle is to convert the BST's iterator into the set iterator.

```
template <class T>
set <T> :: iterator <T> set <T> :: find(const T & t)
{
   BST <T> :: iterator it = bst.find(t);
   return set <T> :: iterator(it);
}
```

Notice how we initialize the set iterator with the BST iterator. This is trivial because our set iterator has a BST iterator as a member variable.



This will work much like the `map` iterator next chapter. Please see Chapter 10 for pseudocode.

Binary Sort

Sorting algorithms are functions that order data according to a pre-defined criterion. Usually, the criterion is the less-than relationship. After the data has been sorted into $a_1, a_2, \ldots a_n$, then $a_1 \leq a_2 \leq \ldots \leq a_n$. One sorting algorithm is called the binary sort. Using this algorithm, the data to be sorted is placed in a BST and then retrieved from the BST using in-order traversal.

```
template <class T>
void binarySort(T data[], int num)
{
   BST <T> bst;
   int i;
   for (i = 0; i < num; i++)
      bst.insert(data[i]);

   i = 0;
   for (BST <T> :: iterator it = bst.begin(); it != bst.end(); ++it)
      data[i++] = *it;
}
```

How does this sort the numbers? Well, most of the work is done in the `BST::insert()` method. As elements are put into the BST, the `insert()` method finds the spot in the tree that will accommodate an element in such a way as to honor the ≤ constraint. In other words, the BST methods maintain sorted order.

Unfortunately, once data is stored in the BST in sorted order, retrieving the data and placing it back into the data array is non-trivial. When the client called the `binarySort()` method, he or she expected the resulting elements to be placed back in the `data` array rather than in a binary tree. So how can the elements be placed back into the `data` array? The BST iterator is designed specifically for this purpose.

The BST iterator performs an in-order traversal of the binary tree. Since the BST is in sorted order, then the in-order traversal results in visiting the elements in sorted order. Thus, the second part of the binary sort is simply iterating through the BST and placing each element in turn back into the data array.

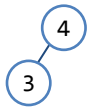
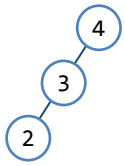
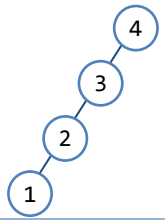
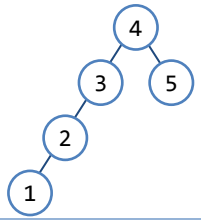
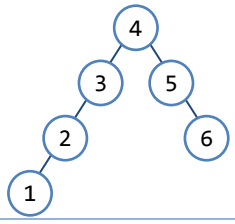
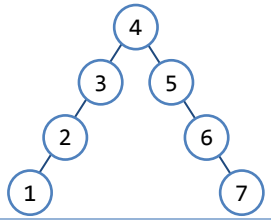
Examples

Example 09.1

Please build a BST from the following sequence of values assuming there is no Red-Black balancing:

4, 3, 2, 1, 5, 6, 7

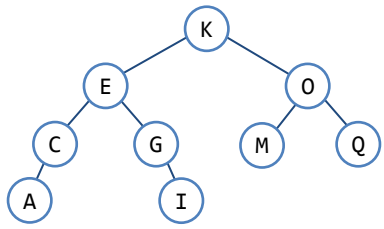
Answer:

4		When adding an element to an empty tree regardless of its value, it is always placed at the root.
3		Since 3 ≤ 4, the new node goes to the left of 4. Since 4 has no children, it remains there.
2		Since 2 ≤ 4, the new node goes to the left of 4. Here, we encounter node 3. Since 2 ≤ 3, we insert to the left of that node. Since 3 has no children, it remains there.
1		We go left of 4 because 1 ≤ 4. We go left of 3 because 1 ≤ 3. We add a new node to the left of 2 because 1 ≤ 2.
5		Since it is *not* true that 4 ≤ 5, the new node goes to the right of 4.
6		Since 6 > 4, we go to the right of 4. Since 6 > 5, we go to the right of 5. Since 5 has no children, the new node gets created there.
7		Since 7 > 4, we go to the right of 4. Since 7 > 5, we go to the right of 5. Since 7 > 6, we go to the right of 6. Since 6 has no children, the new node goes there.

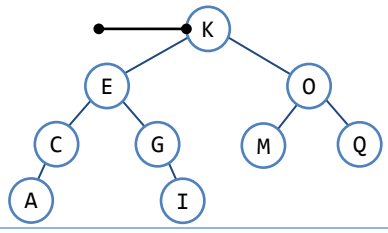
Observe how the resulting tree is not balanced. Red-black balancing would result in a significantly different tree.

Example 09.2

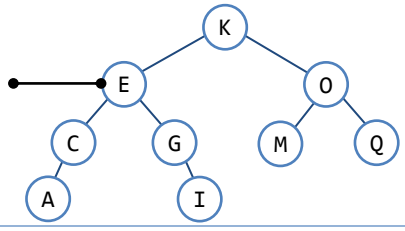
Given the following BST, add the node `B` assuming there is no Red-Black balancing:



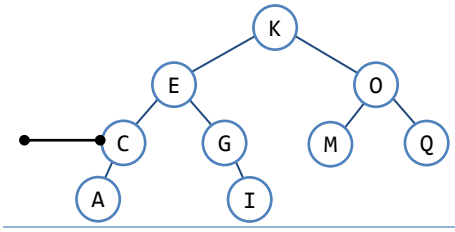
Answer:



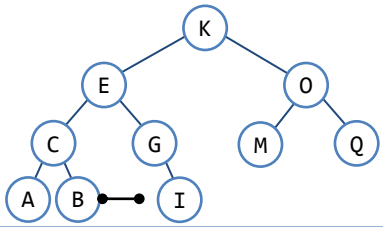
We start at the root. Since `B ≤ K`, we go to the left.



From here, we compare `B` with `E`. Since `B ≤ E`, we go to the left again.



At node `C`, we notice `B > C`. We would go to the right but notice that thre is no node to the right. This will require us to place a new node there.



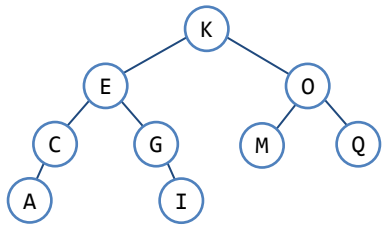
The new node is a right child of `C`.

From here, we should verify that the BST is still well-formed.

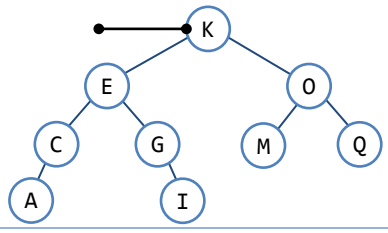
- `B > C`. We must be larger than our in-order predecessor. This is true.
- `B ≤ E`. We must be smaller than our in-order successor. This is also true.

Example 09.3

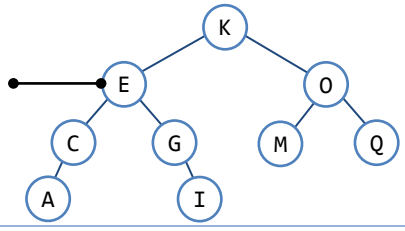
Given the following BST, erase node `G`:



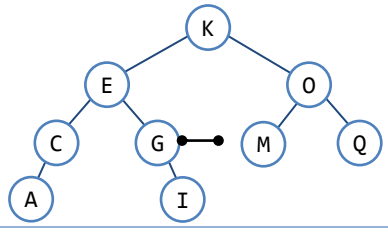
Answer:



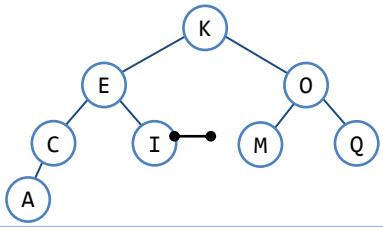
We start at the root. Since `G ≤ K`, we go to the left.



From here, we compare `G` with `E`. Since `G > E`, we go to the right.



Now we have found our node. Since it has only one child, we can put the child in our place.



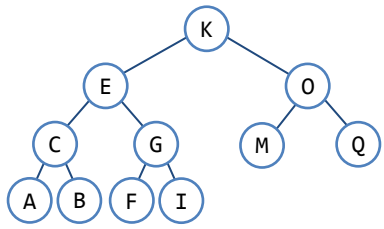
Now that `I` has taken the place of `G`, we are finished.

From here, we should verify that the BST is still well-formed.

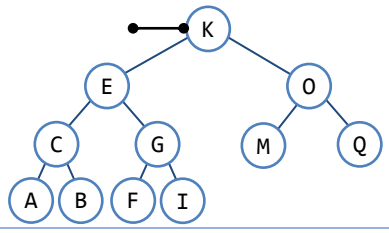
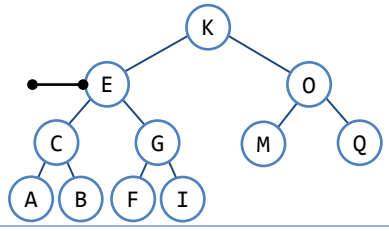
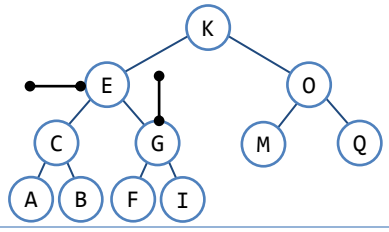
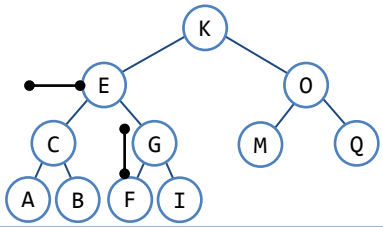
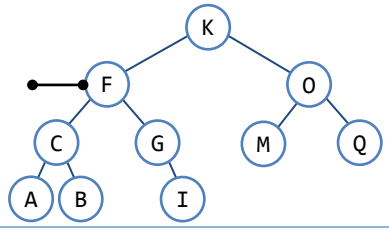
- `I > E`. We must be larger than our in-order predecessor. This is true.
- `I ≤ K`. We must be smaller than our in-order successor. This is also true.

Example 09.4

Given the following BST, erase node `E`:



Answer:

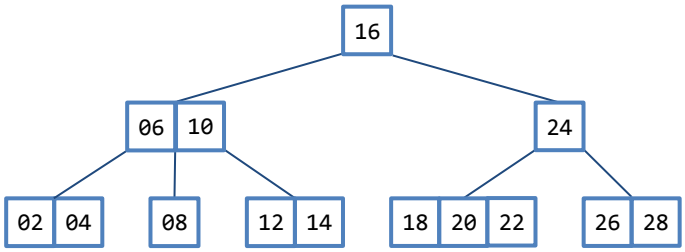
	We start at the root. Since `E` ≤ `K`, we go to the left.
	From here, we find our node. We will erase node `E`. Notice that `E` has two children, complicating the process. We will then go to Case 3 and find the in-order successor.
	First, we go to the right child (the successor must be down this path if there is a right path). This will point to node G.
	After the initial right fork, we continue down the left forks until we reach the end (a node with no left links). In this case, we go just one level down in the tree to node `F`. This node will take the place of the deleted node.
	Now place the value of node `F` is placed in the node that used to contain the value `E`. This erases the value `E`. Finally, the node `F` is deleted from the tree.

From here, we should verify that the BST is still well-formed.

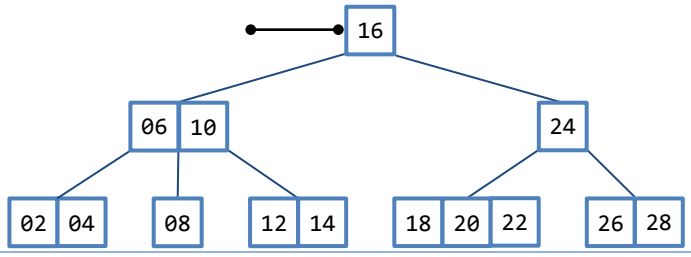
- `F` > `B`. We must be larger than our in-order predecessor. This is true.
- `F` ≤ `G`. We must be smaller than our in-order successor. This is also true.

Example 09.5

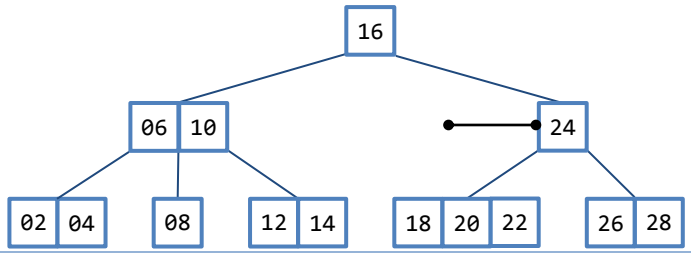
Consider the following 2-3-4 tree. What happens when 21 is inserted into the tree?



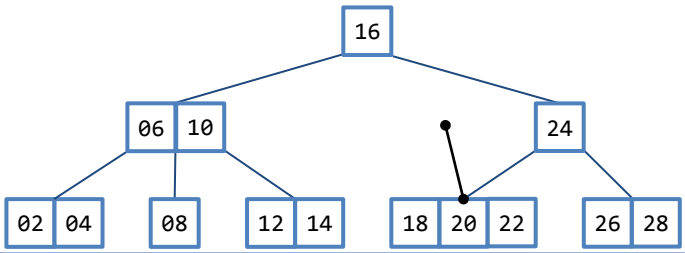
Answer:



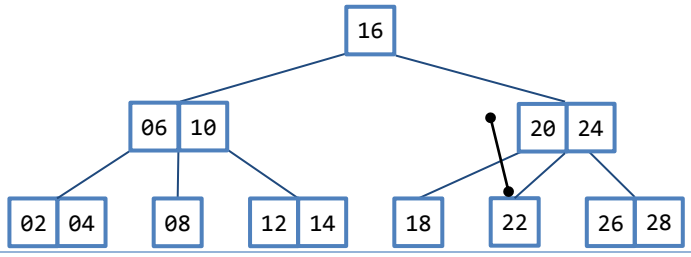
We start at the root. Since 21 > 16, we go to the right.



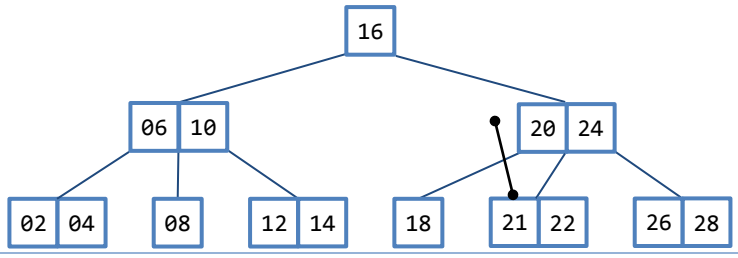
From here, we compare 21 with 24. Since 21 ≤ 24, we go to the left.



Now we found our 3-value node that will accept the value 21. However, this node is full. We thus need to split this node. This will entail moving 21 up to the parent.



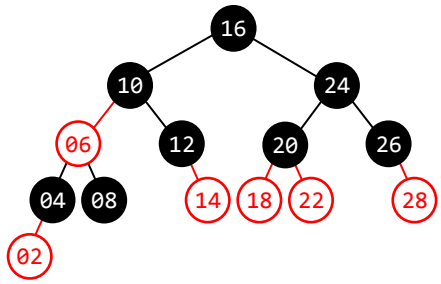
With 20 moved to join 24, we can now insert 21 into the tree. Since 20 ≤ 21 ≤ 24, we will place it beside the 22 node.



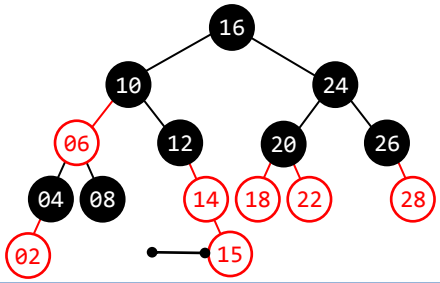
Now we insert node 21 and the tree remains balanced.

Example 09.06

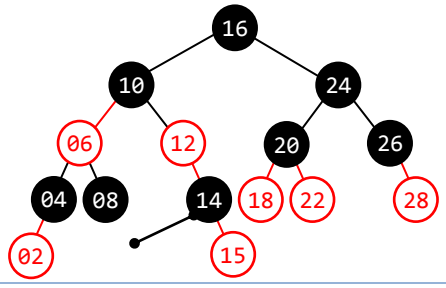
What happens when 15 is inserted to the right of 14 in the following Red-Black tree?



Answer:



We start by inserting 15 to the right of 14 as a red node. All new nodes are inserted as red.



With two red nodes in a row, we violate the Red-Black property. Thus, something needs to be done. Here, case 3 is activated: The parent is red and the aunt is red or non-existant. This causes us to recolor our grandparent as red and make our parent black.

Now we need to verify if the above tree is in fact a Red-Black tree.

a) **Red or Black**: Yes, every node is in fact either red or black.
b) **Black Root**: The root node is 16 and is black.
c) **Red Nodes have Black Children**: All the red nodes (02, 06, 12, 15, 18, 22, and 28) have either no children or black children.
d) **Same Number of Black Nodes**: Every path from the root down to the leaf has exactly three black nodes.

Exercises

Exercise 09.1

Please build a BST from the following sequence of values assuming there is no Red-Black balancing:

55, 28, 61, 12, 31, 79, 49

Answer:

Please see page 222 for a hint.

Exercise 09.2

Please build a BST from the following sequence of values assuming there is no Red-Black balancing:

49, 31, 55, 28, 61, 12, 79

Answer:

Please see page 222 for a hint.

Exercise 09.3

Please build a BST from the following sequence of values assuming there is no Red-Black balancing:

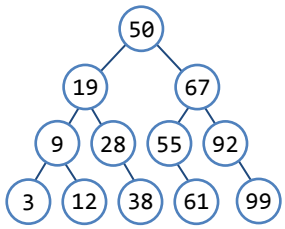
79, 49, 28, 61, 12, 31, 55

Answer:

Please see page 222 for a hint.

Exercise 09.4

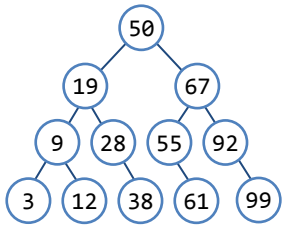
Given the following BST, add the node 11 assuming there is no Red-Black balancing.



Please see page 222 for a hint.

Exercise 09.5

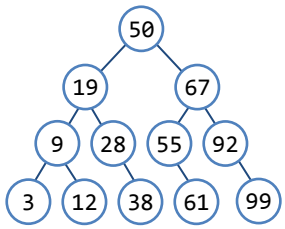
Given the following BST, add the node 52 assuming there is no Red-Black balancing. Hint: You may need to re-draw the BST.



Please see page 222 for a hint.

Exercise 09.6

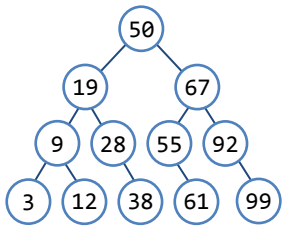
Given the following BST, add the node 26 assuming there is no Red-Black balancing. Hint: You may need to re-draw the BST.



Please see page 222 for a hint.

Exercise 09.7

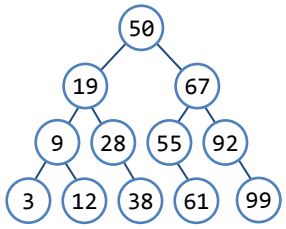
Given the following BST, erase node `9`. Hint: You may need to re-draw the BST.



Please see page 224 for a hint.

Exercise 09.8

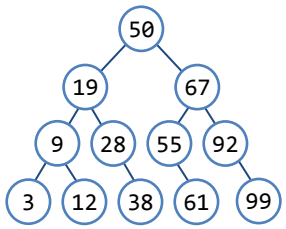
Given the following BST, erase node `67` assuming there is no Red-Black balancing. Hint: You may need to re-draw the BST.



Please see page 224 for a hint.

Exercise 09.9

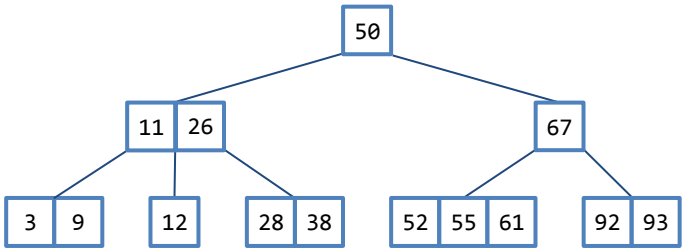
Given the following BST, erase node `50` assuming there is no Red-Black balancing. Hint: You may need to re-draw the BST.



Please see page 224 for a hint.

Exercise 09.10

Consider the following 2-3-4 tree. Which of the following nodes can accept a new value without splitting?



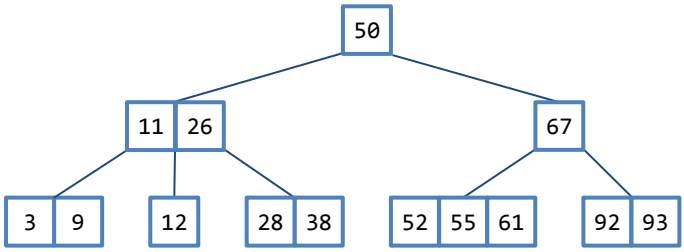
Answer:

_____

Please see page 228 for a hint.

Exercise 09.11

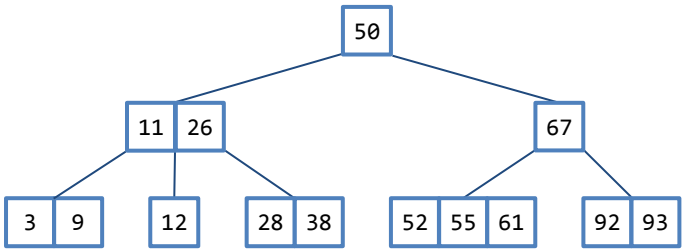
Consider the following 2-3-4 tree. What happens when `19` is inserted into the tree?



Please see page 228 for a hint.

Exercise 09.12

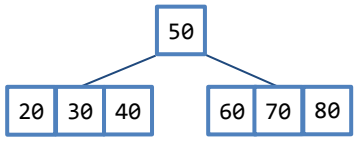
Consider the following 2-3-4 tree. What happens when `64` is inserted into the tree? Hint: You may need to re-draw the 2-3-4 tree.



Please see page 228 for a hint.

Exercise 09.13

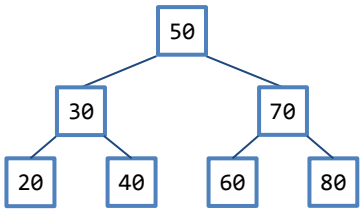
Draw the following 2-3-4 tree as a red-black tree.



Please see page 228 for a hint.

Exercise 09.14

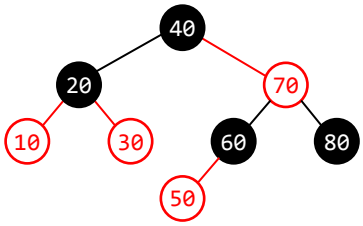
Draw the following 2-3-4 tree as a red-black tree.



Please see page 228 for a hint.

Exercise 09.15

Make a 2-3-4 tree corresponding to the following red-black tree.



Please see page 228 for a hint.

Exercise 09.16

Insert the following nodes to create a 2-3-4 tree: 80, 70, 60, 50, 40, 30, 20, 10.

Please see page 228 for a hint.

Exercise 09.17

Write out the state of the red-black BST after each line or groups of lines are executed.

	State

`{`
`  BST <int> tree;`
`  tree.insert(60);`	

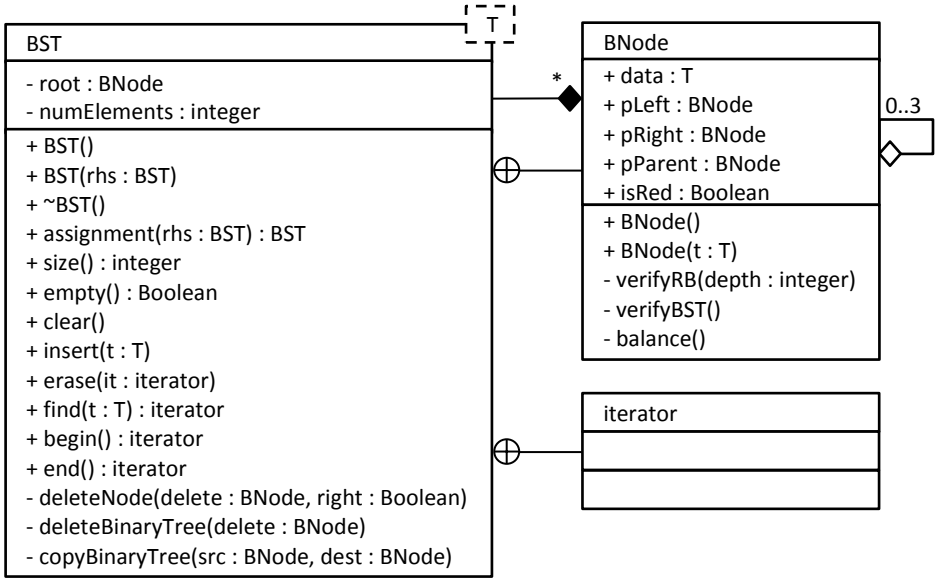
`  tree.insert(50);` `  tree.insert(70);`	
`  tree.insert(20);`	
`  tree.insert(10);`	
`  tree.insert(30);`	
`  tree.insert(40);`	
`  tree.insert(90);`	
`  tree.insert(95);`	
`  tree.insert(85);`	
`  tree.insert(80);`	
`  tree.insert(35);` `}`	

Please see page 230 for a hint.

Problems

Problem 09.1

Create a class to capture the notion of a binary search tree. This will encapsulate the Binary Node, so the clients of the class will have no knowledge or access to any of the implementation details. Of course, any data type will need to be supported, so your class will be a template class. It will need to be defined in its own header file (`bst.h`). The UML class diagram of our BST class is the following:



Please see page 227 for a hint.

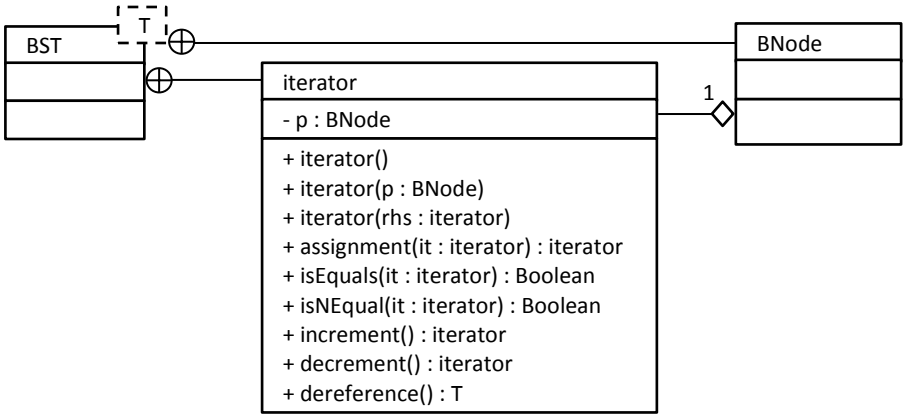
Problem 09.2

Augment your `BST` class to handle Red-Black balancing. This will require a Boolean value in the `BNode` class. Note that you may wish to add some private debug methods to `BST` to verify that the tree is properly balanced.

Please see page 232 for a hint.

Problem 09.3

Augment your BST class to include an iterator. You will need to add the `begin()` and `end()` method to the `BST` class as well as fully implement `BST::iterator`. This should be a nested-class to the BST.



Please see page 237 for a hint.

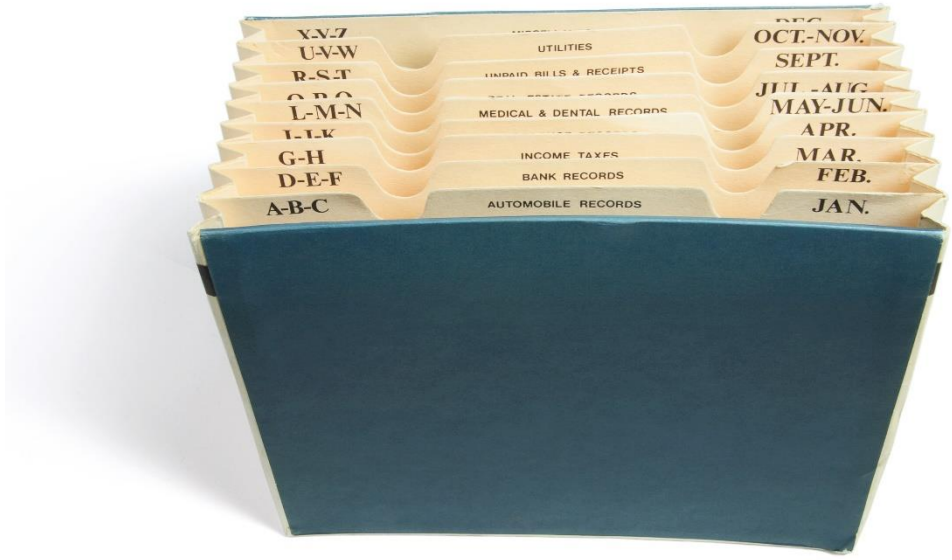
Problem 09.4

Using your BST class, implement the binary sort. A stub of the sort function is the following:

```
template <class T>
void binarySort(T array[], int num)
{
   // your code here
}
```

Please see page 241 for a hint.

Sue manages the finances in her family and likes to keep her records very organized. To help with this task, she purchased an accordion file. In the file, Sue has a slot for automobile records, bank records, income taxes, and a host of other things. She could assign a number to each slot, but "Slot 2" does not feel like "Income Taxes" to her and she is afraid she might misfile something. In other words, she needs a way to name the slots rather than remembering their order in a sequence.



Notice how there are two components: the name of the element to be stored and retrieved (called the key) and the data itself (called the value). In the above example, the key might be "Utilities" and the value might be all the data associated with utilities.

On the surface, the set ADT may appear to be the tool for the job. This is not the case because the set makes no distinction between the key and the value. When searching for an element in the set, one passes all the stored data to the `set::find()` method. In this scenario, we pass only the key leaving the `find()` method to return the value. The map provides this functionality for us.

The defining characteristics of a map are:

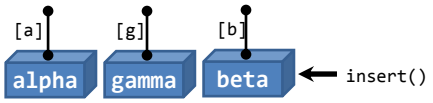
- **Container**: The map is designed to store a collection of data.
- **Random Access**: All elements are equally accessible by key index.
- **Order Arbitrary**: The order elements are stored depends on how the map is implemented.
- **Expandable**: The capacity of the map can be increased to meet the needs of the client.
- **Blind Insertion**: The client is not informed of where items are inserted into the map.

Definition

A map, otherwise known as an associative array, a dictionary, or a symbol table, is an abstract data type storing a collection of values that are indexed by a key. Each entry in the map consists of a key-value pair where the key is used to organize the data. The map ensures that there are no duplicate keys in the collection. The operations performed on a map are the following:

Name	Example	Description
Create	`map <int, string> m;`	Create a new integer-string map.
Assignment	`m1 = m2;`	Make a copy of a map.
Empty	`f = m.empty();`	Returns `true` if a map has no elements, `false` otherwise.
Size	`n = m.size();`	Returns the number of elements currently in a map.
Clear	`m.clear();`	Remove all elements from a map.
Begin	`it = m.begin()`	Returns an iterator pointing to the first element in a map.
End	`it = m.end();`	Returns an iterator indicating the end of a map had been reached.
Insert	`m.insert(k,v);`	Adds a value to a map based on the passed key and value.
Access	`v = m[k];`	Retrieves a value from a map based on the passed key.
Find	`it = m.find(value);`	Returns an iterator of the element corresponding to the passed key.

The map can be represented with the following figure:



In many ways, the map works like an array. The big difference is that we access elements with key (a character in this case) rather than with an integer. This different access method is indicated with the key in the square brackets being a letter rather than a number.

Another thing to notice is that the cubes are not connected. We are given no guarantee that they reside in a continuous block of memory as is the case with vector and other array-based data-types.

Finally, notice how we can add new elements to the map using the square-bracket operator `[]`.

Create

Unlike the ADTs we have defined previously, the map is a template class with two data-types:

```
map <K, V> :: map();
map <K, V> :: map(const map <K, V> & rhs) throw (const char *);
```

The first template is commonly called `K` for "key." This how we will retrieve an element from the map. The second template is called `V` for "value." This is the value to be stored in the map. Both of these needs to be specified when creating a map. The key and the value can be any data-type as long as the following conditions are met:

- The key must support the greater-than operator and the assignment operator
- The value must support the assignment operator

	Code	Description
1	`map <int, string> m1;`	The key is an `int`, the value is a `string`. This is valid because integers support the greater-than operator and the `string` supports the assignment operator.
2	`map <string, float> m2;`	The key is a `string`, the value is a `float`. This is valid because the `string` supports the greater-than operator (returning true if the right-hand-side sorts before the left-hand-side) and the `float` supports the assignment operator.
3	`map <Person, string> m3;`	The key is a `Person` and the value is a `string`. In order for this to work, the `Person` class must define the greater-than operator. Otherwise, this would be invalid.
4	`map <int, char *> m4;`	The key is an integer and the value is a pointer to a character. There is no problem with the integer because the greater-than operator is supported. However, there is a problem with the pointer. By default, the assignment operator for pointers only copies the address, not the data the pointer refers to. This means that the resulting map will probably not work as the client expects.

Insert

It is possible to add an element to the map with the `insert()` method. Note that both the key and the value need to be provided.

```
void map <K, V> :: insert(const pair <K, V> & input) throw (const char *);
void map <K, V> :: insert(const K & k, const V & v)  throw (const char *);
```

If the key does not already exist in the map, then a new element is created and the size will increase by one. If the key does exist, then the corresponding element is updated and the size will not increase by one.

	Code	Description	State
1	`map <int, string> m;`	Create an empty map where the key is an integer and the value is a string.	
2	`m.insert(3, string("three"));`	Insert the value "`three`" under the key of `3`.	[3] ← insert(3, "three") three
3	`m.insert(9, string("nine"));`	Insert the value "`nine`" under the key of `9`.	[9] ← insert(9, "nine") nine three
4	`m.insert(6, string("seis"));`	Add another element to the map. We do not get to specify the order.	[6] ← insert(6, "seis") nine seis three
5	`m.insert(6, string("six"));`	Update the value in the `6` slot with "`six`".	[6] ← insert(6, "six") nine six three

Standard Container Interfaces

All the standard container interfaces are supported by the map.

```
int  map <K, V> :: size()  const;
bool map <K, V> :: empty() const;
void map <K, V> :: clear();
map <K, V> & map <K, V> :: operator = (const map <K, V> & rhs) throw (const char *);
```

These methods work much like the other containers we have previously learned about.

```
{
   map <int, string> m1;
   m1.insert(0, string("zero"));
   m1.insert(1, string("one"));
   m1.insert(2, string("two"));

   cout << m1.size() << endl;       // there are three elements in m1

   map <int, string> m2;
   m2 = m1;                         // copy the contents of m1 into m2

   cout << m1.empty() << endl;      // m1 is not empty so this is false

   m1.clear();                      // remove all elements out of m1
   cout << m1.empty() << endl;      // now m1 is empty
}
```

Access

One of the most powerful and useful methods of the map class is the square bracket operator. This operator is designed to make the map work like an array. It is possible to retrieve values out of the map, update elements in the map, and add elements to the map.

```
V & map <K, V> :: operator [] (const K & k)       throw (const char *);
V   map <K, V> :: operator [] (const K & k) const throw (const char *);
```

The parameter inside the brackets is the key and the value is returned. Thus the return type is the data-type of the value and the parameter data-type is that of the key.

	Code	Description	State
1	`map <char, int> m;`	Create an empty map where the key is a character and the value is an integer.	
2	`m['Z'] = 329;`	Insert the value `329` under the key of `Z`.	[Z] 329
3	`m['%'] = 21;` `m['9'] = 97;`	Add two other elements	[%] 21, 329, [9] 97
4	`cout << m['Z'];`	Displays the value 329.	21, [z] 329, 97
5	`m['Z'] = 0;`	Update the value in the `Z` slot with `0`.	21, [z] 0, 97

Map Iterator

As with set, iterators are the most important mechanism used to access elements in a map. The declaration for a standard forward-iterator is:

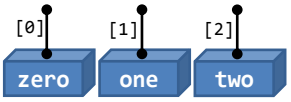
```
map <K, V> :: iterator it;
```

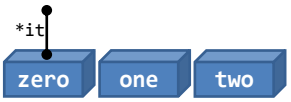
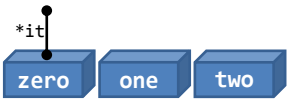
The map class implements two main methods for creating iterators:

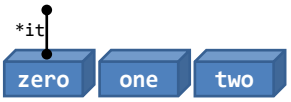
```
map <K, V> :: iterator map <K, V> :: begin();
map <K, V> :: iterator map <K, V> :: end();
```

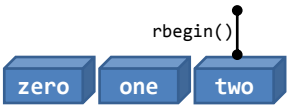
An example of how these iterators work is the following:

	Code	Description	State

1	`map <int, string> m;`
	`m.insert(0, string("zero"));`
	`m.insert(1, string("one"));`
	`m.insert(2, string("two"));`
		Create a map with three elements.	

2	`map <int, string> :: iterator it;` `it = m.begin();`	Create an iterator and point it to the beginning of the map. Note that it is sorted by the key.	
3	`pair <int, string> p;` `p = *it;`	The dereference operator retrieves a pair. This has two member variables: `first` and `second`.	

4	`cout << "key:   " << p.first`
	`  << endl`
	`  << "value: " << p.second`
	`  << endl;`
		The first field of the pair will return the key, the second the value. The output will be:
		`key:   0`
		`value: zero`
			

5	`cout << "key:   "`
	`  << m.rbegin()->first`
	`  << endl;`
	`  << "value: "`
	`  << m.rbegin()->second`
	`  << endl;`
		Recall that `rbegin()` refers to the end of the map. Also notice how we use the `->` operator to access the member variables of pair. The output will be:
		`key:   2`
		`value: two`
			

As with `vector`, `set`, and `list`, there are a total of four variations of the map iterator:

```
map <K, V> :: iterator               itForward;
map <K, V> :: const_iterator         itConstantForward;
map <K, V> :: reverse_iterator       itReverse;
map <K, V> :: const_reverse_iterator itConstantReverse;
```

There are also many variations of the `begin()` and `end()`. The following are just the variations of `begin()`.

```
map <K, V> :: iterator               map <K, V> ::   begin();
map <K, V> :: const_iterator         map <K, V> ::   begin() const;
map <K, V> :: const_iterator         map <K, V> ::  cbegin() const;
map <K, V> :: reverse_iterator       map <K, V> ::  rbegin();
map <K, V> :: const_reverse_iterator map <K, V> ::  rbegin() const;
map <K, V> :: const_reverse_iterator map <K, V> :: crbegin() const;
```

Implementation

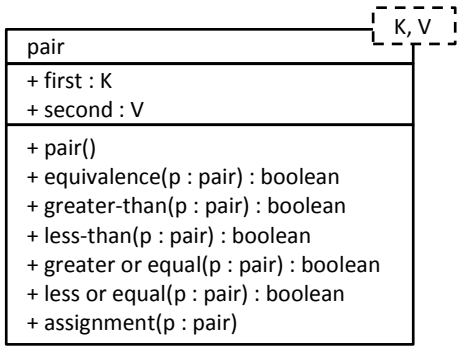
On the surface, the map data structure appears to be very difficult and complex to implement. After all, we have not encountered anything quite like it up to this point. That being said, consider the BST as a candidate for the underlying data structure.

If we create a map class with a BST as a member variable, then we can implement the [] operator as a variant of the `BST::find()` method. The only thing we would have to do is to find the element in the BST corresponding to the key and then return the value.

This works surprisingly well because insertion and retrieval from a BST occurs in O(log n). There is one big problem with this approach: there is no distinction between the key and the value.

The difference between the map ADT and the others discussed earlier is that elements are organized according to the key that is independent of the underlying value. Fortunately there is a simple solution to this.

Consider the `pair` class. It is a non-expandable ADT that stores two independent pieces of data.



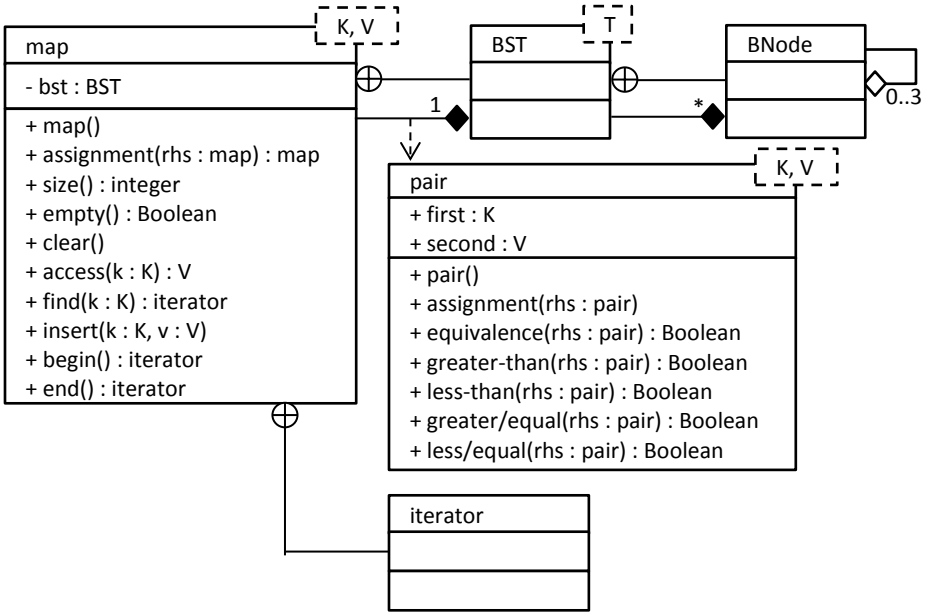
Here we can create a `pair` where the first element is a string and the second is a number:

```
{
   pair <string, int> phoneNumber;
   p.first = string("Dr. Drake Ramoray");
   p.second = 5551263;
}
```

There is one other relevant detail for the `pair` class: the comparison operators work only on the *first* element whereas the assignment operator works on *both* the first and second.

```
template <class K, class V>
bool operator < (const pair <K, V> & lhs, const pair <K, V> & rhs)
{
   return lhs.first < rhs.first;
}
```

Why is this detail important? Consider a BST of `pair`s. The first element of the `pair` is called the key and the BST uses that element to organize nodes in the tree. The second element of the `pair` is called the value. Using this setup, it should be possible to easily define all the methods of the `map`.



The above UML class diagram is our most complicated one of the entire textbook. A few things to note:

- As with other iterators, `iterator` is a nested class of `map`.
- `BST` is also a nested class of `map`, in this case a private one. In turn, `map` contains one instance of `BST`. If the map is destroyed, then the BST is destroyed, hence the closed diamond arrow ♦.
- Of course, `BST` has many `BNodes`, where `BNode` is a nested class of `BST`. This is unchanged from the previous chapter.
- The arrow extending down from the connection between the map and the `BST` indicates that we are using a `pair` to turn the two templates of map (`K` and `V`) into a single template of `BST` (`T`). In other words, `BST`'s `T` will be a `pair` of `K`,`V`.

Create and Assign

Creating a map consists of simply creating an empty BST. In other words, the default constructor is trivial. However, the copy constructor is not. Every node in the right-hand-side must be copied to the left-hand-side. One way to accomplish this is to simply iterate through the right-hand-side and then insert these nodes into the left-hand-side. While this certainly works, it results in a highly un-balanced tree. Notice that iterating through a BST results in the nodes being visited in sorted order and, without Red-Black balancing, the nodes will be added solely to the left side of the tree. Even with balancing, the tree will need to be rotated too many times. A better solution is to use the copy function associated with the `BST`. This copy function is the assignment operator. Note that C++ will call the BST assignment operator if we do not explicitly call it ourselves. However, it is good programming practice to be unambiguous in situations such as these.

```
map.map(rhs)
  bst ← rhs.bst
```

The performance of this is O(n) where n is the number of elements in the map.

Of course, the assignment operator works much the same way.

```
map.assignment(rhs)
  bst ← rhs.bst
  RETURN this
```

While the copy constructor and the assignment operator may look trivial, we must remind ourselves that the underlying BST data-structure is doing all the heavy lifting.

Insert

Inserting elements into the map involves searching for the existence of a node with that key and, if it does not exist, then inserting a new node. If a node does exist, then the value in that node will be updated.

```
map.insert(pair)
  it ←bst.find(pair)
  IF it ≠ NULL
    *it ← pair
  ELSE
    bst.insert(pair)
```

Since the BST is balanced (due to the Red-Black implementation), the cost of finding an element in the tree is O(log n). The cost of adding a new element to the tree is also O(log n). Thus the total cost of `insert()` is O(log n).

Empty, Size, and Clear

The assignment, empty, size, and clear methods are all directly called from the BST underlying data-structure.

```
map.size()
  RETURN bst.size()
```

```
map.empty()
  RETURN bst.empty()
```

```
map.clear()
  bst.clear()
```

Since our BST class supported all of these common container interfaces, we can simply pass on the requests to our underlying data structure.

Access

In many ways, the defining method of the map is the square bracket operator. Through this method the client is given access to values within the map as well as allowed to add elements to the map.

```
map.operator[key]
  pair ← key, Value()
  it ← bst.find(pair)
  IF it ≠ NULL
    RETURN it.second
  ELSE
    bst.insert(pair)
    RETURN bst.find(pair).second
```

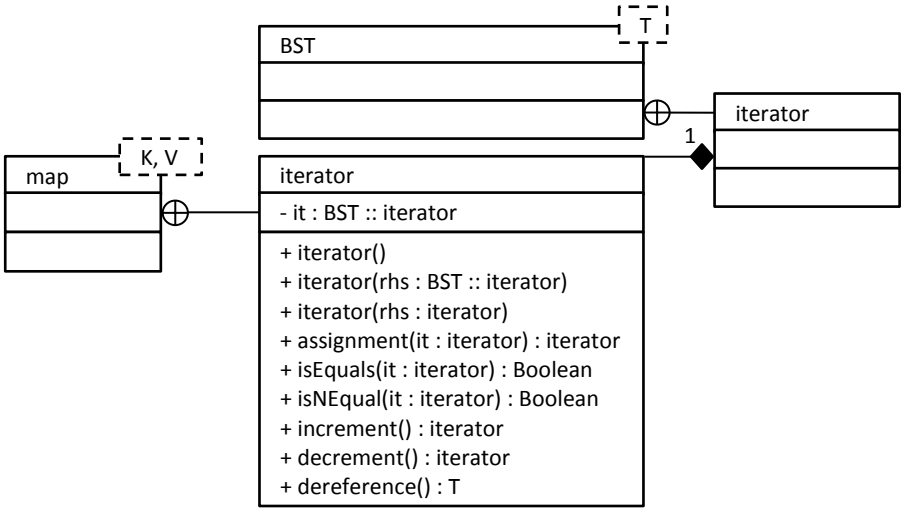
This requires a bit of an explaination. The first line of code creates a `pair` out of the key and a default value (calling the default constructor). When we call `BST::find()`, we get an iterator referring to the node with the corresponding key. Recall that the `pair` only compares the `first` element which is our `key`.

If the iterator refers to an actual element in the BST, then we return the second element of the `pair` (which is our `value`). This serves to retrieve the data from the associative array that the client requested.

If the iterator does not point to an element in the BST, then a new element is created. This new element has the passed key and the default value. We will then return this default value. Since the return type is by-reference (which we cannot show in pseudocode), the client can then update the value with the assignment operator.

Map Iterator

Because the underlying data-structure of the map is a BST and because the BST has an iterator, the map iterator is actually trivial.



A few things to notice about this UML class diagram:

- First, the `iterator` is a nested class within `map`. This works exactly the same as the `vector::iterator`, `set::iterator`, and `list::iterator`. We can tell that `iterator` is a nested class within `map` because of the ⊕ symbol.
- Next, notice the `iterator` on the far right. It is a `BST` iterator (`BST::iterator`). We know this because it is nested within the `BST` (again, with the ⊕ symbol on the line connecting these two classes).
- Also, notice that the `map::iterator` contains one `BST::iterator`. We have two indications of this. First, the iterator's member variable is a `BST::iterator`. Second, there is a closed diamond ♦ connecting the two classes. This means that a `BST::iterator` will be created when a `map::iterator` is created and the `BST::iterator` will be destroyed when the `map::iterator` is destroyed.
- The final thing to observe about this UML class diagram is that lots of details are removed for brevity's sake. The specifics of the `map`, `BST`, and `BST::iterator` class are defined elsewhere. Also, the relationship between `map` and `BST` are not presented here.

All the `map::iterator` methods simply call the comparable `BST::iterator` methods. For example:

```
map::iterator.isEquals(rhs)
  RETURN it = rhs.it
```

To advance the iterator, simply call the BST advance method:

```
map::iterator.advance()
  ++it
```

We can also create the `map <T> :: begin()` and `map <T> :: end()` method by calling upon the corresponding BST interfaces:

```
map.begin()
  RETURN bst.begin()
```

```
map.end()
  RETURN bst.end()
```

In other words, it may look like the map iterator is trivial. This is the furthest thing from the truth! The map iterator *looks* trivial because the BST iterator (which is anything but trivial) is doing *all* the work.

Application

Two examples will be shown that utilize the map.

Phone book

Write a program to associate a phone number with an individual's name. Note that the name can be any text field. For this, we will use the standard `string` class. The phone number is a challenge. It could be an integer, but most people expect the number to be represented in a human-readable format. For example, most would prefer "(208) 496-7608" rather than 2084967608. Thus we need to implement a class to store the phone number:

```
class PhoneNumber
{
public:
   PhoneNumber(long number) : number(number)                 {}
   PhoneNumber(const PhoneNumber & rhs) : number(rhs.number) {}
   PhoneNumber & operator = (const PhoneNumber & rhs)
   {
      number = rhs.number;
      return *this;
   }

   void display() const
   {
      if (number <= 9999)
         cout << 'x' << number;
      else if (number <= 9999999)
         cout << number / 10000 << '-' << number % 10000;
      else
         cout << '(' << number / 10000000 << ") "
              << (number % 10000000) / 10000 << '-'
              << number % 10000;
   }
private:
   long number;
};
```

With the `PhoneNumber` class defined, we can now define the `map` which will store the data. Since we are going to be looking up numbers based on the user name, the `Key` part of the `map` will be a `std::string`. Since the data we will be retrieving will be a `PhoneNumber`, the `Value` part of the `map` will be a `PhoneNumber`.

```
map <std::string, PhoneNumber> phoneBook;
```

Recall that the key must support the less-than operator for it to function within a map. In this case, the `std::string` does define this method, returning true of the left-hand-side sorts before the right-hand-side in alphabetical order. A second constraint is that the value must support the assignment operator. We can verify that this is the case because we have a definition of `PhoneNumber::operator=()`.

We can now write a function that will retrieve an individual's phone number based on the name:

```
void displayNumber(const map <string, PhoneNumber> & phoneBook, const string & name)
{
   cout << "The phone number of " << name << " is ";
   phoneBook[name].display();
}
```

Grade book

In this second application, I wish to store the grades for a given class. This class will have a number of students and a number of assignments. Each assignment-student `pair` will have a single numeric grade.

On first glance, this does not appear to be a job for the map class. The reason for this is that we have two keys: both the assignment name and the student name. However, we can overcome this by writing a class:

```
class GradeItem
{
public:
   GradeItem(const string & assignment, const string & name)
   {
      data = assignment + name; // combine the assignment and the student name
   }
   GradeItem(const GradeItem & rhs) : data(rhs.data) {}
   GradeItem & operator = (const GradeItem & rhs)
   {
      data = rhs.data;
      return *this;
   }
   bool operator < (const GradeItem & rhs) const
   {
      return data < rhs.data;
   }
private:
   string data;
};
```

Notice how we simply append the assignment name with the student name into one string and compute less-than based on it. Since `GradeItem` is to be our key, it must support the less-than operator.

With the `GradeItem` defined, we can now define the `gradeBook` object:

```
map <GradeItem, int> gradeBook;
```

From here, we will fill the `gradeBook` with 10 grades.

```
void enterGrades(map <GradeItem, int> & gradeBook)
{
   // retrieve the assignment name;
   string assignment;
   cout << "What is the assignment name? ";
   cin >> assignment;

   // retrieve the grades for the 10 students
   cout << "Please enter the name followed by the grade for the 10 students\n";
   for (int i = 0; i < 10; i++)
   {
      // prompt for the student name and the grade
      string name;
      int    grade;
      cout << "> ";
      cin  >> name >> grade;

      // enter the grade item into the gradebook
      GradeItem item(assignment, name);
      gradeBook[item] = grade;
   }
}
```

Notice how we use the array-index operator to insert items into the gradebook. We also could have used the `insert()` method with `gradeBook.insert(item, grade);`.

Examples

Example 10.1

Consider the following code:

```
{
   map <float, string> m;

   // we can add elements using the insert() method...
   m.insert(4.00, string("A"));
   m.insert(3.67, string("A-"));
   m.insert(3.33, string("B+"));
   m.insert(3.00, string("B"));

   // ... or we can add elements using the [] operator
   grades[2.67] = string("B-");
   grades[2.33] = string("C+");
   grades[2.00] = string("C");
}
```

How can I display a table of the letter grades?

```
4.00 : A
3.67 : A-
3.33 : B+
3.00 : B
…
```

Answer:

The tool for the job is to use an iterator. Note that when we dereference the iterator, we get a `pair`. The `first` element is the key, the `second` is the value.

```
{
   // configure cout to display the GPA to 2 decimal places
   cout.setf(ios::fixed | ios::showpoint);
   cout.precision(2);

   // display the contents of the map from lowest to greatest
   map <float, string> :: iterator it;
   for (it = m.begin(); it != m.end(); ++it)
      cout << it->first
           << " : "
           << it->second
           << endl;
}
```

Recall that the map stores the grades in sorted order because the underlying data-structure is a BST. Thus, if we want to display the 4.00 grade on top, we need to use a reverse iterator.

```
{
   // display the contents of the map from greatest to smallest
   map <float, string> :: reverse_iterator it;
   for (it = m.rbegin(); it != m.rend(); ++it)
      cout << it->first
           << " : "
           << it->second
           << endl;
}
```

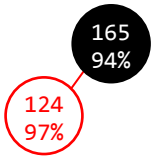
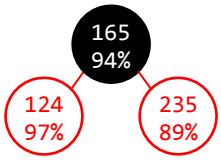
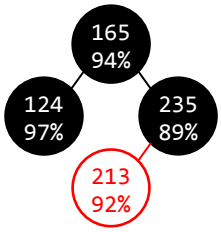
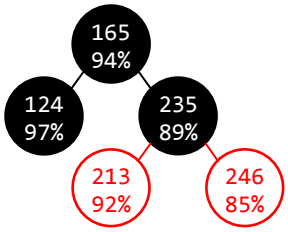
Example 10.2

Consider the following code:

```
{
   map <int, float> grades;
   grades[165] = 0.94;
   grades[124] = 0.97;
   grades[235] = 0.89;
   grades[213] = 0.92;
   grades[246] = 0.85;
}
```

Draw the underlying data structure to represent the map `grades`.

Answer:

`grades[165] = 0.94;`	With the first insertion into the BST, just a single node is created. It contains the class name as the key (165 for "CS 165") and the grade of 95% as the value. This is a black node because it is the root (from Red-Black Case 1).	
`grades[124] = 0.97;`	A new element is added to the map. Since the key 124 is less than the root 165, we add a new node to the left of the root. Notice that this node is red because it is not the root and remains red because the parent is black. This is Red-Black Case 2.	
`grades[235] = 0.89;`	Another element is added. This time the key 235 is greater than the root so it gets added to the right of the root. As with the previous step, this new node is red (also Red-Black Case 2).	
`grades[213] = 0.92;`	With this next addition, we observe that 213 > 165 so it gets added on the right side. Furthermore 213 ≤ 235 so it gets added to the left of 235. This new node is initially red but, since the parent is red, some adjustment needs to be made. This activates Case 3 of Red-Black balancing because the aunt (node 124) is also red. Thus the parent and aunt become black.	
`grades[246] = 0.85;`	Since 246 > 165, the new node goes to the right of the root. Since 246 > 235, the new node goes to the right as well. New nodes are red by default ans, since the parent is black (Case 2), not node recoloring or balancing is required.	

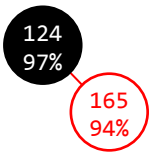
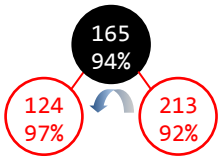
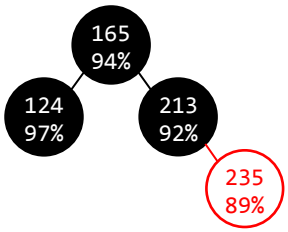
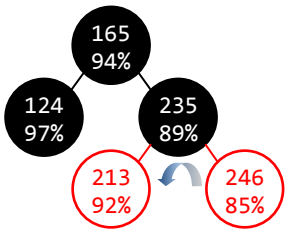
Example 10.3

Consider the following code:

```
{
   map <int, float> grades;
   grades[124] = 0.97;
   grades[165] = 0.94;
   grades[213] = 0.92;
   grades[235] = 0.89;
   grades[246] = 0.85;
}
```

Draw the underlying data structure to represent the map `grades`.

Answer:

`grades[124] = 0.97;`	As with Example 10.1, we initially have a single black node (Red-Black Case 1).	
`grades[165] = 0.94;`	Since 165 > 124, we create a new red node to the right (Red-Black Case 2).	
`grades[213] = 0.92;`	Another element is added. Since it is the largest key thus far, it gets added to the far right of the tree. Since we have two red nodes in a row (the newly added 213 and 165), we need to balance. This will activate Case 4 variant b.	
`grades[235] = 0.89;`	Again our key is greater than any encountered before so a new red node gets added to the bottom right corner of the tree. Again two red nodes are in a row (213 and 235). Since the parent and aunt are both red, this activates Case 3.	
`grades[246] = 0.85;`	Our final insertion puts yet another node in the bottom right corner of the tree with another two red nodes in a row (235 and 246). This will require another rotation, Case 4 variant b.	

Notice how two separate rotations were required when inserting nodes in this sequence, whereas no rotations were required in Example 10.3. The order in which nodes are inserted in a balanced BST greatly affects the efficiency of the insert operations.

One final note. Without Red-Black tree balancing, 124 would remain the root and all the nodes would be in a single line off to the right. This would make access time O(n) whereas a balanced tree gives us O(log n). The extra work required to keep the BST balanced yields large performance gains.

Exercises

Exercise 10.1

From memory, recite the methods associated with the `map`. Write the entire function definition in C++.

Answers:

_____

_____

_____

Please see page 257 for questions

Exercise 10.2

From memory, draw the complete UML class diagram of the map excluding the member variables and member functions.

Answers:

Please see page 263 for questions

Exercise 10.3

Draw the underlying data structure (red-black tree) corresponding to each successive line of code.

```
{
   map <int, string> m;
   m[60] = string("G");

   string s = "H";
   pair <int, string> p(50, "F");
   m.insert(p);
   m[70] = s;


   m[10] = string("!");



   m[30] = string("C");
   map <int, string> :: iterator it;
   it = m.find(10);
   it->second = "A";


   p.first = 40;
   p.second = "?";
   m.insert(p);



   m[90] = string("K");
   m[40] = string("E");



   m[95] = string("L");



   m[85] = string("J");



   m[80] = string("I");



   m[35] = string("D");

}
```

State

Please see page 269 for a hint.

Exercise 10.4

Consider the following code:

```
{
   map <char, string> m;
   m['b'] = string("Bravo");
   m['a'] = string("Alpha");
   m['c'] = string("Charlie");
   m['d'] = string("Delta");

   cout << m.size() << endl;
}
```

What is the output?

_____

Please see page 260 for a hint.

Exercise 10.5

Consider the following code:

```
{
   map <char, string> m;
   m['b'] = string("Bravo");
   m['a'] = string("Alpha");
   m['c'] = string("Charlie");
   m['d'] = string("Delta");

   cout << "a=" << m['a'] << endl;
}
```

What is the output?

_____

Please see page 261 for a hint.

Exercise 10.6

Consider the following code:

```
{
   map <char, string> m;
   m['b'] = string("Bravo");
   m['a'] = string("Alpha");
   m['c'] = string("Charlie");
   m['d'] = string("Delta");

   if (m['x'] == m['y'])
      cout << m.size() << endl;
   else
      cout << m.size() << endl;
}
```

What is the output?

_____

Please see page 261 for a hint.

Exercise 10.7

Consider the following class:

```
class Weird
{
public:
   Weird(int number)      { data = new int(number); }
  ~Weird()                { delete data;            }
private:
   int * data;
};
```

If I were to create a map using this new `Weird` class, what would I need to add to `Weird` so the code would work? Please provide an implementation for the necessary function.

```
{
   map <Weird, int> m;
}
```

Answer:

Please see page 257 for a hint.

Exercise 10.8

Consider the following class:

```
class Weird
{
public:
   Weird(int number)      { data = new int(number); }
  ~Weird()                { delete data;            }
private:
   int * data;
};
```

If I were to create a map using this new `Weird` class, what would I need to add to `Weird` so the code would work? Please provide an implementation for the necessary function.

```
{
   map <int, Weird> m;
}
```

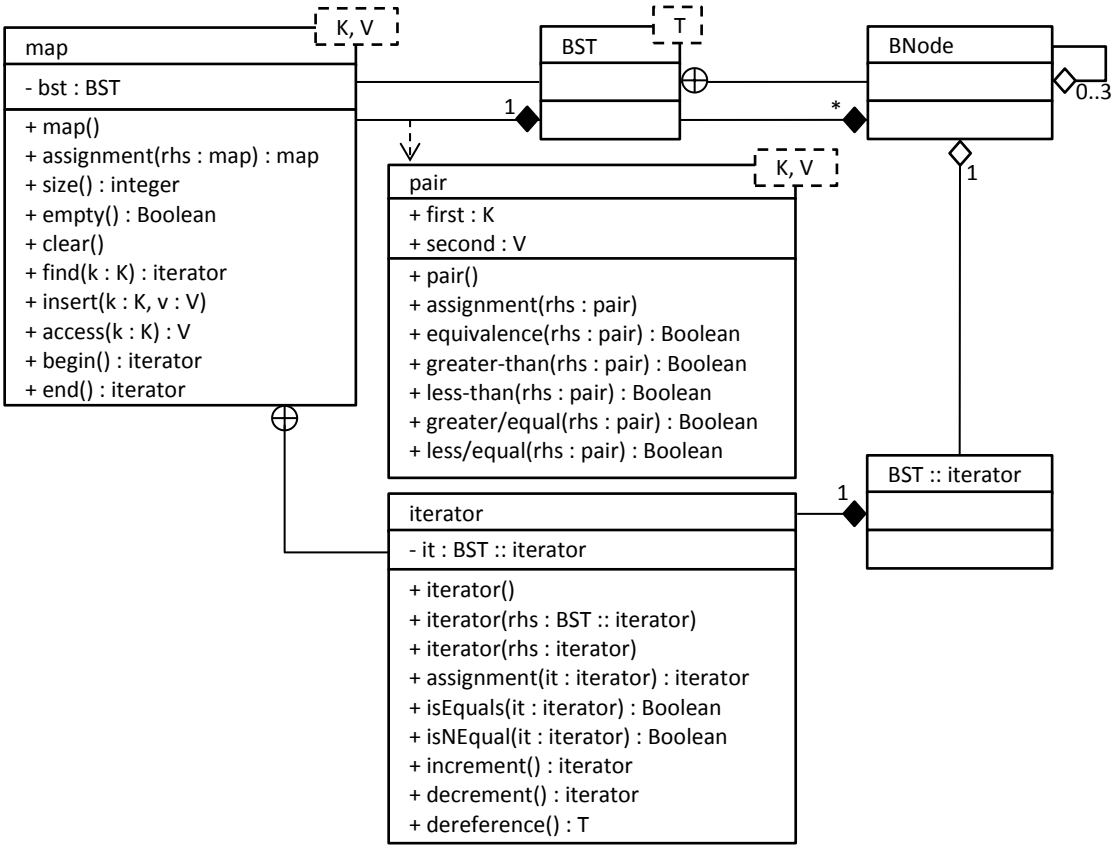
Answer:

Please see page 257 for a hint.

Problems

Problem 10.1

Implement your own version of the `std::map` data structure. Call your implementation `custom::map`. In other words, place your class in the custom namespace and implement all of the interfaces described in this chapter. The UML class diagram of our `map` class is the following:



Additionally, create an iterator for the map class.

Please see page 263 for a hint.

Problem 10.2

Using the STL `map` or using your own implementation of `map` from Problem 10.1, write a program to count the number of occurrences of words from a data file. Consider the following file:

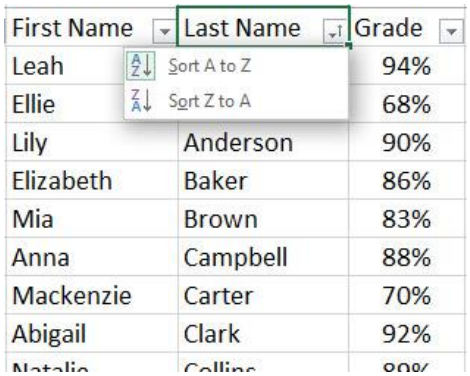
```
apple apple orange
```

When the user runs the program, the following output results:

```
What is the filename to be counted? fruit.txt
What word whose frequency is to be found. Type ! when done
> apple
        apple : 2
> orange
        orange : 1
> grapes
        grapes : 0
```

11 : Sorting

Sorting is the process of ordering a collection of elements by some criteria. Consider, for example, a pile of graded assignments. The teacher may wish to alphabetize these assignments for the purpose of entering the grades into a gradebook. In this case, the collection of elements is the graded assignments. The criteria is to alphabetize the assignments by the students' last name. After accomplishing this, the teacher may wish to sort the same assignments by grade so as to see which students did the best on the assignment and which need help. In this second example, the collection of elements remains the same as before. However, the criteria by which the collection is sorted has changed. In both of these examples, the teacher sorted the collection of assignments.



The process of sorting elements in a collection is a widely used operation in many programming contexts. Lists of files are sorted before they are presented to the user in a File Open dialog box. Elements in a database are often sorted in order to facilitate more efficient retrieval. Anytime elements need to be retrieved from a large collection where performance matters, sorting is part of the decision process.

On the surface, sorting may appear to be a better topic for an algorithms class rather than a class on data structures. However, many of the most common sorting algorithms make use of data structures we have already studied. In addition to that, one sort introduces a data structure that is new to us.

Sorting

A sort is a function that takes two parameters and has no return type. The first parameter is the array to be sorted. This parameter serves both as input to the sort function (the array of unsorted elements) and the output from the sort function (the newly sorted array). The elements are to be sorted from smallest to largest. The second parameter is the number of elements in the array. Thus the C++ prototype of a sort function is:

```
template <class T>
void sort(T array[],    // the input and output parameter
          int n);       // input only, the number of elements in the array
```

This sort function should work for any data-type `T` that provides the following operations:

- **Assignment**: `T` must have the ability to copy itself through the assignment operator.
- **Less-than**: `T` must provide a means to determine the relative order of any two values. It is by this criteria that the array is sorted.

The output of the sort algorithm is the array such that `array[0]` ≤ `array[1]` ≤ … ≤ `array[n - 1]`. This may seem contradictory at first; how can the array be ordered with by the ≤ criteria when only the less-than operator is guaranteed to exist? Fortunately the following equivalence relations exist: `t1` < `t2` ↔ `t2` > `t1` and `t1` ≤ `t2` ↔ !(`t1` > `t2`).

There are a very large number of sorts that have been described through the years. They all honor the same contract (that `array[0]` ≤ `array[1]` ≤ … ≤ `array[n - 1]`) but the way this contract is honored varies greatly. Some perform better on partially sorted arrays. Some perform well only on small data-sets. Some are reliably fast while others are fast in certain conditions. We will explore seven sorts that use different methods for ordering the elements in an array and have different performance characteristics: bubble sort, selection sort, insertion sort, binary sort, merge sort, quick sort, and heap sort.

Each sorting algorithm will be described four ways:

a) **High level plain-English**. Each algorithm will be described in such a way that a non-technical person could understand it. Though important details will be left out, it should be sufficient to get a general idea of how the sort functions.
b) **Detailed description**. A detailed description of the various parts of the algorithm. The goal here is to describe the sort in such detail that an individual could correctly conduct the sort by hand using sheets of paper.
c) **Pseudocode**. The algorithm will be presented in a format that should allow a straight-forward translation into a high-level programming language of your choice.
d) **Example**. Finally, an example will be presented and the performance characteristics of the algorithm will be discussed.

As you read this chapter, please strive to understand all four explanations of the algorithm. This way you will have something to fall back on if you don't understand one of the explanations.

Bubble Sort

The bubble sort is perhaps the easiest sort to implement and one of the easiest sorts to understand. The first mention of the bubble sort can be traced to a 1956 article in the *Journal of the ACM* titled "Sorting on Electronic Computer Systems" that referred to the algorithm as "sorting by exchange." Because of its poor performance characteristics, the bubble sort should not be used in practice in all but a very small number of applications. In short, the bubble sort is often optimal when the data-set contains five elements or less. The bubble sort can be characterized by:

Bubble Sort	Generally the slowest of the common sorting algorithms	
Average case	$O(n^2)$	Almost all cases are $O(n^2)$
Best case	$O(n)$	When the array is already sorted, only a single pass is required
Worst case	$O(n^2)$	When the array is sorted in the wrong direction
Space	$O(1)$	No extra allocations are required

Algorithm in Plain English

In the simplest terms, the bubble sort first looks for the largest element in the array. This element is put in the right-most slot (the highest index). Next, the second largest element is found and put in the slot second from the right. Next the 3rd largest elements is found, then the 4th, and so on. This continues until each element is in the proper slot.

Detailed Description of the Algorithm

There are two variables necessary to implement the bubble sort:

- `iPivot`: an index into the array separating the array into three parts: 1) the part of the array that remains to be sorted (everything left of `iPivot`), 2) from the slot in the array we are currently trying to fill next (the value under `iPivot`), and 3) the part of the array that has already been sorted (everything to the right of `iPivot`).
- `iCheck`: an index iterating through the unsorted portion of the array used to find the element belonging under `iPivot`.

The central idea of the bubble sort is the pivot represented with the `iPivot` variable. With the first iteration of the algorithm, `iPivot` is assigned to the right-most slot (`iPivot = num - 1`) and is filled with the largest element in the array. As the pivot moves left (`--iPivot`), it is filled with progressively smaller elements. When the pivot reaches the left side of the array (`iPivot == 1`), then the number of elements in the unsorted side is one (the smallest item in the array) and the rest of the array is sorted. At this point, the algorithm is complete.

The second part of the bubble sort is the method by which the algorithm finds the largest remaining element in the unsorted side of the array. It is this part of the algorithm where the bubble sort derived its name. The counter `iCheck` moves from the zero index to the pivot for the purpose of bringing the largest element to that slot in the array. This counter compares every element with the element to the right. If they are not in order, the pair of elements are swapped. After the conditional swap has been handled, the counter simple progresses to the right (`++iCheck`). Thus the largest remaining element in the unsorted side of the array is carried to the right to `iPivot`. If you watch `iCheck` move through the list, it appears to bubble the largest element to the right.

There is one final optimization to the bubble sort. If `iCheck` makes it all the way across the unsorted part of the array without swapping any elements, then that half of the array is actually sorted. At that point, it is safe to return out of the function because the entire array is in sorted order.

Algorithm in Pseudocode

The bubble sort has two loops: the outer loop representing `iPivot`'s progress from right to left, and the inner loop representing `iCheck`'s effort to bring the highest remaining element from left to right into `array[iPivot]`. It is important that these two loops move in opposite directions! An optional part of the algorithm is the `swapped` Boolean variable recording if any pair of elements in the `iCheck` loop have been discovered to be out of order. If `swapped` remains `false` at the end of the inner loop, then the function is complete.

```
sortBubble(array, n)
   FOR iPivot ← n-1 … 1
      swapped ← false
      FOR iCheck ← 0 … iPivot-1
         IF array[iCheck] > array[iCheck+1]
            swap array[iCheck] and array[iCheck+1]
            swapped ← true
      IF !swapped
         RETURN
```

Example

To illustrate the bubble sort, we will first consider the inner loop where `iCheck` moves from left (low index) to right (to the pivot). Initially `array == {31, 72, 10, 32, 18, 95, 25, 50}` and `iPivot == 7`.

iCheck	Description	State
0	Start by comparing 31 with 72. Since `array[iCheck] < array[iCheck + 1]`, we do not swap these two elements.	**31** 72 10 32 18 95 25 50
1	Compare 72 with 10. Since the element on the right is smaller, the two elements need to be swapped.	31 **72** 10 32 18 95 25 50 swap!
2	Since 32 < 72, we swap these two elements. Notice how 72 is getting carried (or "bubbled") to the right.	31 10 **72** 32 18 95 25 50 swap!
3	As with the previous step, 18 < 72 so we swap again.	31 10 32 **72** 18 95 25 50 swap!
4	For the first time in a while, the value under `iCheck` is smaller. This means we do not swap 72 with 95.	31 10 32 18 **72** 95 25 50
5	Compare 95 with 25 and, because 95 is larger, another swap is required.	31 10 32 18 72 **95** 25 50 swap!
6	The final comparisons. Since there were eight elements, there must be seven comparisons. This concludes the inner loop.	31 10 32 18 72 25 **95** 50 swap!

The above example illustrates how the inner loop works. Next we need to see how the outer loop works.

The outer loop of the bubble sort moves `iPivot` from the right (highest index) to the left (zero index). As with before, we will start with `array == {31, 72, 10, 32, 18, 95, 25, 50}` and `iPivot == 7`.

iPivot	Description	Array
	This is the initial unsorted list	31 | 72 | 10 | 32 | 18 | 95 | 25 | 50
7	First we will solve for the 7 slot. Note how 10 moved to the left and 72 moved to the right as 95 finds is place under `iPivot`.	31 | 10 | 32 | 18 | 72 | 25 | 50 | **95**
6	After 72 is brought to its slot, 72 and 95 are in sorted order. Also note how 10 slid left through the bubbling process.	10 | 31 | 18 | 32 | 25 | 50 | **72** | 95
5	With this iteration we brought 32 over but 50 was already in the correct slot.	10 | 18 | 31 | 25 | 32 | **50** | 72 | 95
4	Now 31 got moved into the correct slot while 32 was already there.	10 | 18 | 25 | 31 | **32** | 50 | 72 | 95
3	As we solve for `iPivot = 3`, we make no swaps during our `iCheck` loop. This means we are done!	10 | 18 | 25 | **31** | 32 | 50 | 72 | 95

There are two things to observe from the above illustrations. First, the pivot separates the sorted side (on the right) from the un-sorted side (on the left). The selection and insertion sort will behave the same.

The second thing to observe is that the bubbling process also tends to order the elements to the left of the pivot. Notice how the value 10 moved from index 2 to index 0 after a few iterations. Perhaps more dramatically, the value 18 moved from index 4 to index 1 during this time.

Performance

In order to characterize the performance of the bubble sort, we will look at five specific points in the algorithm.

```
sortBubble(array, n)
   FOR iPivot ← n-1 … 1                                  Line 1
      swapped ← false
      FOR iCheck ← 0 … iPivot-1                          Line 2
         IF array[iCheck] > array[iCheck+1]              Line 3
            swap array[iCheck] and array[iCheck+1]       Line 4
            swapped ← true
      IF !swapped
         RETURN                                          Line 5
```

The overall performance characteristic is described below:

Line	Discussion	Cost
1	The outer loop will execute `n - 1` times unless it is cut short by the swapped return function. Thus in the common case, it will be O(n) but in the best case (when the list is sorted), it will execute O(1) times	O(n)
2	The inner loop will execute `n - 1` times with the first iteration of `iPivot`. It will then execute `n - 2` times, then `n - 3` until the end. Thus it is directly influenced by `n` so the performance is O(n). However, the inner loop itself is executed `n` times. That brings the total cost to $O(n^2)$	$O(n^2)$
3	The body of the inner loop has an IF statement which is not dependent on the size of the array (`n`). Thus the performance is O(1). However it will be executed n^2 times so the total cost is $O(n^2)$	$O(n^2)$
4	Executed once per swap which can happen no more than n^2 times (in the case when the array is sorted backwards.) The typical case is still about half the time. Thus the overall cost is $O(n^2)$	$O(n^2)$
5	The RETURN statement can be executed at most once, regardless of the size of the array.	O(1)

The overall cost of the bubble sort is the highest cost of any single line. In this case, line 2, 3, and 4 are $O(n^2)$.

Selection Sort

The author of the selection sort is difficult to determine because many consider it the natural way that people sort things. It was first mentioned in a publication in 1956 as an enhancement to a widely known algorithm. The selection sort takes a very large number of comparisons to order the values, but takes a minimum number of assignments. In fact, it takes no more than three assignments per element to be sorted. The selection sort can be characterized by:

Selection Sort	A very large number of comparisions are required	
Average case	$O(n^2)$	All cases make $O(n^2)$ comparisons
Best case	$O(n^2)$	When the array is sorted, no assignments are made
Worst case	$O(n^2)$	Most cases are the worst case
Space	$O(1)$	No extra allocations are required

Algorithm in Plain English

Imagine a teacher sorting a collection of assignments by the student's last name. She picks up the stack of papers (representing the unsorted array) and looks for the last name that sorts last (the highest value). Because the stack is unsorted, she must perform a linear search. When the highest value is found, she places it on her desk (representing the sorted portion of the array) and continues by looking for the next highest value. As she continues, the stack of papers in her hand (the unsorted array) gets smaller and the stack of papers on the desk (the sorted array) gets larger. When there are no papers remaining in her hand, the stack on the desk is the resulting sorted list.

Detailed Description of Algorithm

As with the bubble sort, a central idea of the selection sort is the pivot, which separates the side of the array that is already sorted from the side that remains to be sorted. In the first iteration, `iPivot` refers to the right-most slot (`iPivot = num - 1`). This represents the slot where the largest element in the unsorted side of the array must go.

Now it is necessary to find the index of the largest element to the left of the pivot. Two variables are required: `iCheck` representing the index in the unsorted side of the array that is currently being searched, and `iLargest` representing the largest element found thus far. If the element under `iCheck` is greater than `iLargest`, then `iLargest` receives the value of `iCheck`. This continues as `iCheck` goes from the slot one to the left of `iPivot` all the way to slot `0`. Now all that remains is to swap the element under `iPivot` with the element under `iLargest` if the element under `iLargest` happens to be bigger than the one under `iPivot`.

Algorithm in Pseudocode

There are two loops in the selection sort: the outer loop moving `iPivot` from right to left, and the inner loop where `iCheck` looks for the largest element (represented with `iLargest`).

```
sortSelection(array, n)
   FOR iPivot ← n-1 … 1
      iLargest ← 0
      FOR iCheck ← 1 … iPivot-1
         IF array[iLargest] ≤ array[iCheck]
            iLargest ← iCheck
       IF array[iLargest] > array[iPivot]
          swap array[iLargest], array[iPivot]
```

Example

To illustrate the selection sort, we will first consider the outer loop where `iCheck` moves from left (low index) to right (to the pivot). Initially `array == {31, 72, 10, 32, 18, 95, 25, 50}` and `iPivot == 7`. At the beginning of the loop, everything to the right of `iPivot` is sorted. At the end, the value under `iPivot` is smaller than everything to the right of that element and larger than anything to the left. The element to the right of the bar is the element under `iPivot`. After each iteration, the element that was in that location gets moved to a new location represented by the shaded square.

iPivot	State	Description
	31 | 72 | 10 | 32 | 18 | 95 | 25 | 50	Initially the list is unsorted and no items are in their proper location
7	31 | 72 | 10 | 32 | 18 | 50 | 25 ▌ 95	Looking at the array to the right of `iPivot`, we discover that `95` is the largest value. Thus `95` and `50` are swapped.
6	31 | 25 | 10 | 32 | 18 | 50 ▌ 72 | 95	The largest value is now `72` in slot 1. It is swapped with `25` in slot `6`.
5	31 | 25 | 10 | 32 | 18 ▌ 50 | 72 | 95	Since the search space includes the the element under `iPivot`, we discover that `50` is the highest. Nothing changes
3	31 | 25 | 10 | 18 ▌ 32 | 50 | 72 | 95	The next highest value is `32` so slots `3` and `4` are swapped.
4	18 | 25 | 10 ▌ 31 | 32 | 50 | 72 | 95	A search through `{31, 25, 10, 18}` yields slot `0` with the greatest value. Thus we swap `31` and `18`
5	18 | 10 ▌ 25 | 31 | 32 | 50 | 72 | 95	The next search through `{18, 25, 10}` yields slot `1` with the highest value. Thus we swap `25` with `10`
6	10 ▌ 18 | 25 | 31 | 32 | 50 | 72 | 95	The last step is to search through `{18, 10}`. Since the value under `0` is larger, we perform the last swap of the sort.
7	▌ 10 | 18 | 25 | 31 | 32 | 50 | 72 | 95	Now we are sorted.

Performance

In order to characterize the performance of the selection sort, we will look at five parts of the algorithm:

```
sortSelection(array, n)
   FOR iPivot ← n-1 … 1                          Line 1
      iLargest ← 0
      FOR iCheck ← 1 … iPivot-1                  Line 2
         IF array[iLargest] ≤ array[iCheck]
            iLargest ← iCheck                    Line 3
       IF array[iLargest] > array[iPivot]        Line 4
          swap array[iLargest], array[iPivot]    Line 5
```

The overall performace characteristic is described below:

Line	Discussion	Cost
1	The outer loop will go through `n` iterations. This will be O(n) regardless of the data.	O(n)
2	The inner loop will execute `n` times from the outer loop and will increment `iPivot` times. Since `iPivot` is the variable of the outer loop, this line it will execute $O(n^2)$ times.	$O(n^2)$
3	Every time a new largest item is found, line 3 will be executed. This will happen not quite as frequently as the inner loop. In fact, it will occur log n time because the deeper we are into the `iCheck` loop, the less frequently a new largest item will be found.	O(n log n)
4	This is executed once for every execution of the outer loop.	O(n)
5	The IF statement will be true almost every time.	O(n)

The overall cost of the selection sort is $O(n^2)$ because line 2 will be executed n^2 times. In other words, finding the largest item is the most expensive part of the sort.

Insertion Sort

As with the selection sort, it is difficult to determine exactly who invented the insertion sort because it is one of the sorts people tend to naturally do. It was first mentioned in 1946 in a publication in connection to computer sorting by John Mauchly though he implied the sort was well know at that time. The insertion sort is important for two reasons: first it makes the smallest number of comparisons of any of the common sorts, and second it was the inspiration for many of the higher performance sorts that followed.

The insertion sort can be characterized by:

Insertion Sort	A very large number of assignments are required	
Average case	$O(n^2)$	Almost all cases are $O(n^2)$
Best case	$O(n)$	When the array is sorted
Worst case	$O(n^2)$	Most cases are the worst case
Space	$O(1)$	No extra allocations are required

Algorithm in Plain English

Imagine a teacher sorting a collection of assignments by the student's last name. She leaves the unsorted pile on her desk and picks up the first paper from the stack. Next she picks up the next paper from her desk and places it before or after the paper in her hand depending on the value. Now her hand represents the sorted pile while the desk is the unsorted pile (the opposite of the selection sort). She picks up the next paper from the desk and finds where it goes. Notice how she can use a binary search in the sorted pile to find where the new paper goes because she is inserting the paper into a sorted list. When there are no papers left on the desk and all the papers are in her hand, the collection is sorted.

Detailed Description of Algorithm

As with the bubble sort and the selection sort, `iPivot` will divide the sorted side from the unsorted side. It will start at right-most slot (`iPivot = num - 1`) and moves left. With every iteration (as `iPivot` decrements by one), the element under `iPivot` is put to the appropriate location in the sorted side of the array. This will require shifting every element to the left by one slot.

Before the new element is inserted into the sorted side, every element is less than or equal to the element to the left (the definition of a sorted sub-list). This is true with all indices from `iPivot + 1` to `num - 1`. This means that `array[iPivot + 1]` ≤ `array[iPivot + 2]` ≤ … ≤ `array[num - 1]`. After insertion, the new element needs to honor this relation. This means three things need to happen:

a) The location of the new slot for `array[iPivot]` needs to be found. We will call this `iInsert`.
b) Everything that is currently between `iPivot + 1` and `iInsert` needs to be scooted over to the left by one slot.
c) The value that was under `iPivot` needs to be inserted into the new location `iInsert`.

We have two options for finding `iInsert`. The first is to conduct a binary search through the sorted side and then shift from there. The second is to shift from `iPivot` up to the place where `iInsert` is located. Which is better?

If we conduct a binary search, then it will take $O(\log n)$ compares to find the location of `iInsert`. From there, it will take $O(n)$ assignments to shift everything over. This will be two separate loops taking $O(\log n + n)$ time which can be simplified to $O(n)$.

The second option is to shift as we go. We can hold the value of `iPivot` in a temporary location (`pivotValue = array[iPivot]`) and then begin the shifting (`array[iPivot] = array[iPivot + 1]`). Because we will want to remember the location of `iPivot`, we will do this with a separate variable: `iShift`. We will loop `iShift` from `iPivot` to the place where `array[iShift] > pivotValue`. When this happens, then we will simply plop `pivotValue` into `array[iShift]` (`array[iShift] = pivotValue`). How will this compare with the other method? This certainly *seems* faster because there is one loop rather than two. We will conduct O(n) assignments as before. The disadvantage is that we will also conduct O(n) compares because we are looking for `iInsert` as we shift.

Both algorithms take O(n) to insert a single element into the sorted side of the array. The "binary search" option takes O(log n) compares while the "shift as we go" option takes O(n) compares. If a compare is trivial (such as comparing two integers) then the "shift as we go" option seems better. However in the case were the comparison operation is complex or time consuming (such as comparing large strings), then we can easily lose any performance advantage. For this reason, the "binary search" implementation is the preferred algorithm.

Algorithm in Pseudocode

There are two loops in the insertion sort: the outer loop moving `iPivot` from right to left, and the inner loop left-shifting those elements to the right of `iPivot` until `iInsert` is found.

```
sortInsertion(array, n)
   FOR iPivot ← n-2 … 0
      valuePivot ← array[iPivot];
      iInsert ← binarySearch(array, valuePivot, iPivot + 1, n – 1)
      iInsert--
      for iShift ← iPivot  … iInsert
         array[iShift] ← array[iShift + 1]
      array[iShift] ← valuePivot
```

In order for this to work, we need a function `binarySearch()` which will return the location of `iInsert`. Specifically, it needs to return the index of the location of where `valuePivot` would go assuming everything to the right is slid. Note that we are not sliding to the right but rather to the left (because we are removing the value under `iPivot`). Hence we need to reduce one from `iInsert`.

The next question is: how will we implement `binarySearch()` so it returns `iInsert`? The answer is to use a standard recusive binary search algorithm:

```
binarySearch(array, search, iBegin, iEnd)
  iMiddle ← (iBegin + iEnd)/2
  IF iBegin > iEnd
    RETURN iBegin
  IF array[iMiddle] = search
    RETURN iMiddle
  IF search > array[iMiddle]
    RETURN binarySearch(array, search, iMiddle + 1, iEnd)
  ELSE
    RETURN binarySearch(array, search, iBegin, iMiddle – 1)
```

Initially `iBegin` and `iEnd` represent the entirety of the array (`iBegin = 0` and `iEnd = num - 1`). With each successive call to `binarySearch()`, the range of values between `iBegin` and `iEnd` is cut in half in much the same way one finds the hymn in a hymnal by successively moving the right and left hand closer to the desired page.

Example

To illustrate the insertion sort, we will first consider the outer loop where `iPivot` moves from right (high index) to left (to the zero index). Initially `array == {31, 72, 10, 32, 18, 95, 25, 50}` and `iPivot == 6`. At the beginning of the loop, everything to the right of `iPivot` is sorted. In the below illustrations, we will show `iPivot` as a strong vertical bar. The element to the left of the bar is the element under `iPivot`. After each iteration, the element that was in that location gets moved to a new location represented by the shaded square.

iPivot	Description	State
7	Start with a single item to the right of `iPivot`. This will be the sorted side. This means `iPivot = 7` and `array[iPivot] = 50`.	31 | 72 | 10 | 32 | 18 | 95 | 25 ▌ 50
6	When we binary search through `{50}`, it returns the index where the element would go. This is slot `6` so nothing is shifted	31 | 72 | 10 | 32 | 18 | 95 ▌ 25 | 50
5	Next we search through `{25, 50}` and discover that `95` goes in slot `7`. We shift over the `25` and the `50`.	31 | 72 | 10 | 32 | 18 ▌ 25 | 50 | 95
3	A search through `{25, 50, 95}` yields slot `4` so nothing gets shifted since because `18` goes in the slot of `iPivot`.	31 | 72 | 10 | 32 ▌ 18 | 25 | 50 | 95
4	A search through `{18, 25, 50, 95}` yields slot `5` so we shift over `18` and `25`.	31 | 72 | 10 ▌ 18 | 25 | 32 | 50 | 95
5	The next search through `{18, 25, 32, 50, 95}` yields slot `2` so nothing gets shifted over since `10` is in the correct location.	31 | 72 ▌ 10 | 18 | 25 | 32 | 50 | 95
6	A search for `72` in `{10, 18, 25, 32, 50, 95}` yields slot `6` so most of the array gets shifted over.	31 ▌ 10 | 18 | 25 | 32 | 50 | 72 | 95
7	The last step searches for `31` in `{10, 18, 25, 32, 50, 72, 95}` yields slot `3` so `{10, 18, 25}` get shifted over.	▌ 10 | 18 | 25 | 31 | 32 | 50 | 72 | 95

Performance

In order to characterize the performance of the selection sort, we will look at five parts of the algorithm:

```
sortInsertion(array, n)
   FOR iPivot ← n-2 … 0                                          Line 1
      valuePivot ← array[iPivot];                                Line 2
      iInsert ← binarySearch(array, valuePivot, iPivot + 1, n – 1)  Line 3
      iInsert--
      for iShift ← iPivot  … iInsert                             Line 4
         array[iShift] ← array[iShift + 1]                       Line 5
      array[iShift] ← valuePivotpivot;
```

The overall performace characteristic is described below:

Line	Discussion	Cost
1	The outer loop will go through n iterations. This will be O(n) regardless of the data.	O(n)
2	For each iteration of the loop, this statement will be executed once. Since the loop will be executed n times, this is O(n).	O(n)
3	The binary search takes O(log n). Since it will be executed n times, the total cost of this line is O(n) x O(log n) = O(n log n)	O(n log n)
4	On average, each shift will move half of the sorted side of the pivot. Since the size of the sorted size is on average half of the size of the array, this means shifting will move about n/4 of the elements. This is O(n). Since the shifting happens n times the overall cost is $O(n^2)$	$O(n^2)$
5	The body of the loop will be executed once for every iteration of the loop. This will occur O(n) times.	O(n)

The overall cost of the insertion sort is $O(n^2)$ because line 4 will be executed n^2 times. In other words, the shifting is the most expensive part of the sort.

Binary Sort

The binary sort was first described by Windley, Booth, Colin, and Hibbard in 1960. The binary sort can be characterized by:

Binary Sort	All the allocated nodes results in a large amount of overhead	
Average case	O(n log n)	Most cases are O(n log n)
Best case	O(n log n)	This is the common case
Worst case	$O(n^2)$	The worst case is when the array is sorted or reverse sorted unless red-black balancing is performed. Then O(n log n)
Space	O(n)	A binary node must be allocated for every element in the array

While on paper it may seem to be the fastest sort, the necessity of allocating O(n) binary nodes makes it slightly slower than other "fast" sorts.

Algorithm in Plain English

Our teacher is again sorting a pile of papers. Frustrated that it has taken so long to sort the papers when there are more than 50 students, she decides to try something radically different.

The teacher starts with a very large table with a binary tree drawn on it with white tape (contrived example, I know…). On this table, she takes the first paper from the unsorted pile and places it at the top of the tree. She takes the next paper and, if it is *after* the first in alphabetical order, she places it to the *right* and down from the first. If it is *before* in alphabetical order, she places it to the *left* and down from the first. She then takes the next paper and finds its location in the binary tree by moving down-and-right or down-and-left until an empty node is found. This continues as she removes papers from the pile and place them in this large binary tree drawn on the table. If the table is large enough and if she keeps the papers tidy, she will have a fully formed binary tree in a few minutes. From here, she carefuly collects the papers from left to right and the resulting pile of papers is sorted.

Detailed Description of Algorithm

There are two steps in the binary sort: filling a binary search tree (BST from Chapter 09) with data from the unsorted array, and then filling the array with data from the BST.

To fill the BST with array data, first an empty BST is created. Next a loop traverses the unsorted array. With each iteration of this loop, the corresponding array element is inserted into the BST. This will require a binary node to be created for every element in the array and for the correct spot in the BST to be found. Once the array loop has finished, the resulting BST will contain a copy of all the array data.

The second step will be to remove data from the BST and place it back in the array. This will result in a sorted array because BSTs organize data in sorted order. An iterator will traverse the BST and, with each iteration of the loop, the corresponding tree element is placed back in the array in successive slots. Thus when the BST iterator is complete, all the data will be copied from the BST back into the array.

Algorithm in Pseudocode

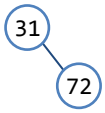
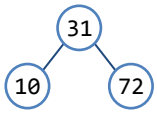
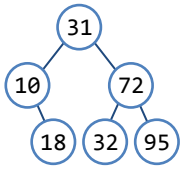
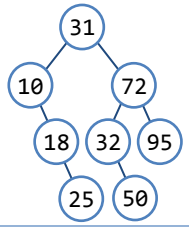
There are two parts to the pseudocode: building the BST and then refilling the array.

```
sortBinary(array, n)
   FOR i ← 0 … n – 1
      bst.insert(array[i])

   i ← 0
   FOR it ← bst.begin() … bst.end()
      array[i++] ← *it
```

From this pseudocode, it appears that the binary sort is the simplest sort thus far. This perception is misleading. The `BST::insert()` method is quite complex because it must find the location in the BST where the new element is to reside. Similarly the `++it` method is quite complex because the successor in the BST must be found. In other words, all the hard work is done in the `BST` class and the `BST::iterator` class.

Example

To illustrate the bubble sort, we will first build the BST (unbalanced). Initially `array == {31, 72, 10, 32, 18, 95, 25, 50}`.

array[i]	Description	State
`31`	Initially the node 31 is placed as the root to a new BST.	
`72`	Because 72 > 31, the new node gets added to the right of the root.	
`10`	Because 10 ≤ 31, the new node gets added to the left of the root.	
`32, 18, 95`	We will have to perform two "hops" to get to the correct slot. First we go to the right of 31 because 32 > 31. Next we go to the left of 72 because 32 ≤ 72. The same thing is done for 18. We go left (18 ≤ 31) then right (18 > 10). Finally, we add 95. Since it is the largest, it goes to the bottom-right corner of the tree.	
`25, 50`	The last two elements are added to the BST. At this point, we can see that the left branch is not balanced but the right branch is. The binary sort can be implemented as a balanced Red-Black BST tree or as an unbalanced BST.	

The final step is to iterate through the BST and place the results back into the array. To accomplish this, the default in-order traversal is performed.

Performance

In order to characterize the performance of the binary sort, we will look at five parts of the algorithm:

```
sortBinary(array, n)
   FOR i ← 0 … n – 1                          Line 1
      bst.insert(array[i])                    Line 2

   i ← 0                                      Line 3
   FOR it ← bst.begin() … bst.end()           Line 4
      array[i++] ← *it                        Line 5
```

The overall performace characteristic is described below:

Line	Discussion	Cost
1	The outer loop will go through n iterations. This will be O(n) regardless of the data.	O(n)
2	It takes O(log n) to insert an element into a BST. Since this occurs n times, the cost is O(n) x O(log n) = O(n log n)	O(n log n)
3	This line gets executed exactly once per sort.	O(1)
4	On the surface this looks like O(n) because we advance the iterator n times. However the increment operator of the iterator is not O(1). It takes O(log n) to find the next node in the tree. This means the cost of this line is O(n log n)	O(n log n)
5	Dereferencing the iterator takes O(1), as does assigning a value to an array. Since this is performed O(n) times, the cost here is O(n).	O(n)

The overall cost of the insertion sort is O(n log n) because of line 2 and 4.

Merge Sort

The merge sort is one of the first sorts to be formally described, invented by John von Neumann in 1945. It was the first "divide and conquer" algorithm, leveraging the fact that it is much easier to sort smaller lists over larger ones. The merge sort can be characterized by:

Merge Sort	Always fast and does not require the entire array to be in memory	
Average case	O(n log n)	Almost all cases are O(n log n)
Best case	O(n)	When the array is already sorted, only a single pass is required
Worst case	O(n log n)	When the array is sorted in the wrong direction
Space	O(n)	One extra copy of the array must be created

Another advantage of the merge sort is that there are variations that do not require the entire array to be loaded in memory at one time. This allows for "disk-sorting," where only small parts of the data-set are loaded from external storage at a time.

Algorithm in Plain English

Our teacher is frustrated with how long it takes to sort the pile of papers; the bigger the pile the longer it takes! She decides two smaller piles will be easier than one. These are also too big so they are further sub-divided. Soon she has a large number of single- or 2-paper piles. In a few seconds she sorts these small piles by either swapping the papers or leaving piles alone. Next she takes the two smallest piles and merges them into one. This is very easy because they individually are sorted. She next merges the next two small piles. Soon all the small piles are medium sized piles. She then merges them in exactly the same way the smaller piles were merged. After the medium sized piles are all sorted and have become large piles, she merges them. This continues until only one pile remains.

Detailed Description of Algorithm

The first part of the merge sort algorithm is a function that merges two sorted sub-arrays into one. To do this efficiently, the destination array (`destination`) needs to be in a different location than the source arrays (`source1` and `source2`). In-place merging is much more complex and not as efficient. Now since the two source arrays are sorted, we can assume that the smallest element in the two arrays is at the beginning of one of the two arrays. Therefore `destination[0]` must be filled with either `source1[0]` or `source2[0]`. If the smallest was in `source1`, then we move the head of `source1` by one slot. In other words:

```
destination[iDestination++] = source1[iSource1++];
```

Clearly if the smallest was in source2, then we would execute the following code:

```
destination[iDestination++] = source2[iSource2++];
```

This process continues until `iSource1` reaches the end of `source1` or `iSource2` reaches the end of `iSource2`. When that happens, all the elements in the other array are copied directly into the end of `destination`.

The second part of the merge sort is to identify the sub-lists to be merged. Rather than successively sub-dividing the piles as we did in our teacher example which resulted in a plethora of arrays, we will have only two arrays: a source array (`source`) and a destination array (`destination`). From our source array, we will identify two sub-arrays to be merged. Recall that our merge algorithm mandated that the sub-arrays need to be sorted. It did not mandate that the sub-arrays needed to be the same size! Thus we can find the first run of sorted values (where `source[iBegin1] ≤ source[iBegin1 + 1] ≤ source[iBegin1 + 2] ≤ … ≤ source[iEnd1]`). This run may consist of a single value or thousands. Starting with the next index after `i1End`, we will look for

another run of sorted values. This will be called sub-array 2. When finished, we found an `iBegin1`, `iEnd1`, `iBegin2`, and `iEnd2` such that:

- Sub-array 1 is sorted: `source[iBegin1] ≤ source[iBegin1 + 1] ≤ … ≤ source[iEnd1]`
- Sub-array 2 is sorted: `source[iBegin2] ≤ source[iBegin2 + 1] ≤ … ≤ source[iEnd2]`
- The two sub-arrays are next to each other in memory: `iEnd1 + 1 = iBegin2`

Note that we do not need four variables to identify these two sub-arrays. We can omit `iEnd1` because it is simply `iBegin2 – 1`.

Next we need a merge function to merge these two sub-arrays into the destination:

```
void merge(T destination[], T source1[], int size1, T source2[], int size2);
```

This function requires several parameters:

- `destination`: The array which will receive the result of the merge.
- `source1`: A pointer ot the beginning of the first sub-array.
- `size1`: The size of the first sub-array.
- `source2`: A pointer ot the beginning of the second sub-array.
- `size2`: The size of the second sub-array.

The merge algorithm proceeds as follows. A counter proceeds through the destination array (`iDestination`) from the beginning of the range (`0`) until the end (`size1`). From here, the smallest of `source1[i1]` and `source2[i2]` is placed in `destination[iDest]`. If the `i1` value is used, it is incremented. Otherwise, `i2` is incremented. Note that `i1` starts at `0` and we end the loop if `i1 > size1`.

Now that we have identified our sub-array selection strategy and the merge strategy, the merge sort is very easy. Simply look for two sub-arrays and merge them into the destination. Continue doing this until all of the source values are merged into the destination. Next swap the source and destination and begin anew. With every iteration, the sub-array size will at least double. Soon a sub-array consists of the entire destination. At this point, the merge sort is finished.

Algorithm in Pseudocode

The key to the merge sort is an efficient merge algorithm.

```
merge(destination, source1, size1, source2, size2)
  i1 ← 0
  i2 ← 0

  FOR iDestination ← 0 … size1 + size2
    IF i1 ≤ size1 and (i2 = size2 or source1[i1] < source2[i2])
      destination[iDestination] ← source1[i1++]
    ELSE
      destination[iDestination] ← source2[i2++]
```

Notice how we take from the source pile at the first sub-array (`source1[i1++]`) or from the second (`source2[i2++]`). Notice also that we have to write special code from when one or the other sub-array is "finished." This code is in the IF statement.

The next part of the merge sort is the sort itself. This is quite complex because it needs to find the extent of the two sub-arrays and deal with the case of there being just one sub-array remaining.

```
sortMerge(array, n)
  source ← array
  DO
    numIterations ← 0

    WHILE iBegin1 < n
        numIterations++
        FOR iEnd1 ← iBegin1 + 1 … n and !(source[iEnd1 – 1] > source[iEnd1])
        iBegin2 ← iEnd1 + 1
        FOR iEnd2 ← iBegin2 + 1 … n and !(source[iEnd2 – 1] > source[iEnd2])

        IF iBegin2 < n
          merge(destination,
                  source + iBegin1, iEnd1 – iBegin1 + 1,
                  source + iBegin2, iEnd2 – iBegin2 + 1)
        iBegin1 ← iEnd2+ 1

    SWAP source and destination pointers

  WHILE numMerges > 1

  IF array ≠ source
    FOR i ← 0 … n - 1
      array[i] ← source[i]
```

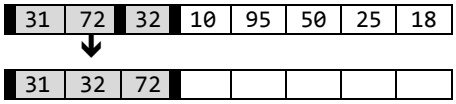
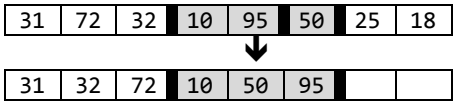
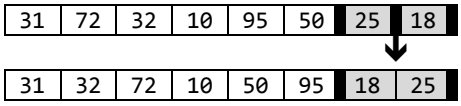
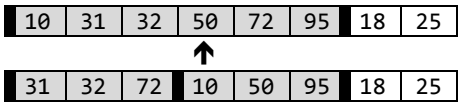
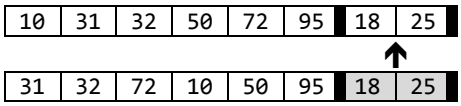
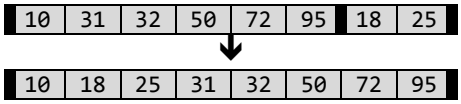
The outer DO-WHILE loop continues until only one merge was done in the inner loop. Since each merge reduces the number of sub-arrays by half, then if there was only one merge during the previous iteration, we are done!

The inner WHILE loop looks for sub-arrays to merge. It does this by looking for the first sorted sub-array (the first FOR loop), then looking for the second sorted sub array immediately after the first (the second FOR loop). If the second sub-array is not empty, then perform the merge.

Notice how we swap the source and destination with each iteration of the outer loop. We start by putting the `array` into `source` and merging into `destination`. Then we swap `source` and `destination` so the old `source` becomes the new `destination`. This continues an unknown number of times. Eventually the merge sort completes. Either we got lucky and the final merge was into `array` or we were unlucky and the final merge was into a temporary array. In the later case, we need to loop through the `source` and copy the final sorted array into the passed parameter. This is the final FOR loop at the end of the pseudocode.

Example

To illustrate the merge sort, start with `array == {31, 72, 32, 10, 95, 50, 25, 18}`. Initially we will sort from the top array into the bottom array (notice the arrow ↓). After the first pass, we will next merge from the bottom array into the top one (indicated with the ↑). With each pass through the array, the sub-lists or runs will be larger than the previous iteration. This will continue until we merge the final two runs.

Description	State
The first run of sorted numbers is `{31, 72}`. The second run is just `{32}` because `10` (the number after) is smaller. These are the two sub-lists to be merged into `{31, 32, 72}`.	
The next run of sorted numbers is just the numbers `{10, 95}`. The following run has the single number {50}. When they are merged, the resulting list is `{10, 50, 95}`.	
The final runs of this iteration will consist of the single item `{25}` and the single item `{18}`. When they are merged, their slots will be reversed.	
Now that we have made one pass through the array, we now make another pass. This time we swap the source and target arrays. In other words, instead of merging down, we will merge up. We will expect the runs to be bigger this time around. The first run is `{31, 32, 72}` and the second run is `{10, 50, 95}`. They merge to `{10, 31, 32, 50, 72, 95}`	
Our next run is `{18, 25}`. However, since there is not another run to merge with, we need to stop here. This means we have completed another iteration.	
For this final iteration, we are merging down again. The first run consists of `{10, 31, 32, 50, 72, 95}` and the second consists of `{18, 25}`. When these two runs are merged, we get the sorted list.	

Performance

In order to characterize the performance of the merge sort, we will look at seven parts of the algorithm:

```
sortMerge(array, n)
  source ← array
  DO
    numIterations ← 0                                          Line 1

    WHILE iBegin1 < n                                          Line 2
      numIterations++
      FOR iEnd1 ← iBegin1 + 1 … n or                           Line 3
          !(source[iEnd1 – 1] > source[iEnd1])
      iBegin2 ← iEnd1 + 1
      FOR iEnd2 ← iBegin2 + 1 … n or                           Line 4
          !(source[iEnd2 – 1] > source[iEnd2])

      IF iBegin2 < n
          merge(destination,
                source + iBegin1, iEnd1 – iBegin1 + 1,
                source + iBegin2, iEnd2 – iBegin2 + 1)         Line 5
      iBegin1 ← iEnd2+ 1

   SWAP source and destination pointers                        Line 6

  WHILE numMerges > 1

  IF array ≠ source
    FOR i ← 0 … n-1                                            Line 7
      array[i] ← source[i]
```

The overall performace characteristic is described below:

Line	Discussion	Cost
1	The outer loop will go through log n iterations. This is because with each iteration the size of the sub-arrays double.	O(log n)
2	When there are a large number of sub-lists, this executes O(n) times. As the number of sub-lists increase, this decreases towards O(1). Due to the outer loop, the cost is thus O(n log n)	O(n log n)
3,4	This gets a bit confusing. The combination of the above loop at line 2 and these inner loops at line 3 and 4 make a single complete pass through the list. The cost is thus O(n). Since the outer loop gets executed O(log n), this is O(n log n)	O(n log n)
5	This is also a bit complicated. The `merge()` function makes a complete pass through the two sub-arrays. It, combined with the WHILE loop at line 2, makes a complete pass through the array. It is therefore O(n). Since the outer loop gets executed O(log n), this is O(n log n)	O(n log n)
6	Swaping pointers to the source and destination arrays takes constant time. However it occurs O(log n) times.	O(log n)
7	Copying the array takes O(n) and occurs half the time. The cost is thus O(n)	O(n)

The overall cost of the insertion sort is O(n log n) because most of the operations within the outer loop take O(n) and the outer loop itself can take no longer than O(log n).

Quick Sort

Quicksort was invented in 1960 by Tony Hoare. It is another "divide and conquer" algorithm, leveraging the fact that it is much easier to sort smaller lists over larger ones. The quick sort can be characterized by:

Quick Sort	Generally the fastest of the common sorting algorithms	
Average case	O(n log n)	Almost all cases are O(n log n)
Best case	O(n)	When the array is already sorted, only a single pass is required
Worst case	$O(n^2)$	When the array is sorted in the wrong direction
Space	O(n)	One extra copy of the array must be created

In most scenarios, the quick sort is the fastest sorting algorithm. Only in special cases (such as the size of the list is under 5 or the cost of comparisons is very high) do other sorting algorithms outperform the quicksort.

Algorithm in Plain English

Our teacher tries another divide-and-conquer technique. She picks a random paper out of the pile and calls this the pivot. She then divides the pile into two: on the left go papers that are smaller than the pivot, on the right are papers larger than the pivot. Keeping these two piles separate, she then does the same to the smaller piles by picking new pivots from within. As she continues, the number of piles increases and their size decreases. When a pile reaches size 2, it is either declared sorted or its papers are swapped and it becomes sorted. Now she simply collects the piles from left to right and she has a single sorted pile of papers.

Detailed Description of Algorithm

The quick sort algorithm is recursive in nature. It consists of a function that takes an array, and a number of elements as a parameter. If the array consists of one element or zero, then it returns because the array is sorted. This is the recursion end-condition.

If there is more than one element in the array, then a pivot point is picked. There are several strategies that can be chosen for this pivot, but we will use the middle element in this case. Next we will have two counters, one that starts at the beginning of the array and moves up (`iUp`), and the second that starts at the end of the array and moves down (`iDown`). When `iUp` (the upward bound counter) encounters an element that is greater than the pivot, it will stop. When `iDown` encounters an element that is less than the pivot, it will stop. Now both counters refer to elements that need to be swapped. A swap is then conducted (between `array[iUp]` and `array[iDown]`) and the counters continue on their course. Eventually the counters reach each other at the pivot. When this happens, everything below `iUp` is less than the pivot and everything above `iDown` is greater than the pivot. From this point, the quick sort function is called again on the lower half of the array (from `0` through `iUp`) and on the upper half of the array (from `iDown` to `num - 1`).

Algorithm in Pseudocode

The algorithm for the quick sort is recursive in nature. This means `quickSort()` will call itself on successively small ranges of numbers until `n` is less than or equal to 1.

```
sortQuick(array, n)
  IF n ≤ 1
    RETURN
  iDown ← iEnd ← n – 1
  iUp ← 0
  pivotValue ← array[iEnd/2]


  WHILE iUp ≤ iDown
    WHILE iUp ≤ iEnd and pivotValue > array[iUp]
      iUp++
    WHILE iDown ≥ 0 and array[iDown] > pivotValue
      iDown--
    IF iUP ≤ iDown
      SWAP array[iUp++], array[iDown--]

  sortQuick(array, iUp)
  sortQuick(array + iUp, iEnd – iUp + 1)
```

Recall that recursive functions need to have two parts: they need to have an end condition and they need to make progress to this end condition with every iteration. The two calls to `sortQuick()` are guaranteed to involve smaller chunks of the original array than the initial call. This satisfies the first condition. The IF statement at the beginning of the function satisfies the second condition.

The rest of the quick sort algorithm consists of the outer WHILE loop which serves to move values to the correct side of the pivot. Note that each call of `sortQuick()` (of which there are typically log n) appears to take $O(n^2)$. This is because there are nested WHILE loops. Here appearances are deceiving. Both the outer WHILE loop and the inner loops work on the same pair of variables. In the end, no line of coder is executed more than n time. This makes the cost of an individual execution of `sortQuick()` – aside from the recursive component of the function – O(n).

Example

To illustrate the quicksort, we will start with `array == {31, 72, 10, 32, 18, 95, 25, 50}` and `num == 8`.

pivotValue	State	Description
32	31 72 10 **32** 18 95 25 50	With the `pivotValue` = 32, we will start with moving `iUp` from slot 0 to the right. The first value it finds that is greater than 32 is 72. It will stop there. Next, `iDown` will start from slot 7 and move to the left until a value is found that is less than 32. It will stop at 25. The next step is to swap these two values.
32	31 25 10 **32** 18 95 72 50	After 72 and 25 are swapped, `iUp` continues to the right until it reaches 32 without finding anything larger. It stops there. Next `iDown` continues the left until 18 is found. It needs to be swapped.
32	31 25 10 18 32 95 72 50	Once 18 and 32 are swapped, we have completed this round of the quicksort. Next we will sub-divide the array into two sub-lists at the pivot and continue again.
25	31 **25** 10 18 32 95 72 50	With this next call to `quickSort()`, we will only consider the left side of the array. For now, we will ignore the right. The new pivot is 25. We reset `iUp = 0` and find that the first element is greater than the pivot. We set `iDown = 3` and find that element is less than the pivot. Thus they need to be swapped.
25	18 **25** 10 31 32 95 72 50	After swapping 18 and 31, we move `iUp` to the pivot. We also move `iDown` to 10 because it is out of place. Another swap is necessary.
25	18 10 25 31 32 95 72 50	Once we swap 25 and 10, we have finished this round of the quicksort. Next we will sub-divide the array further and continue again.
18	**18** 10 25 31 32 95 72 50	Here our pivot is 18 so `iUp` stops immediately. `iDown` finds 10 out of place so another swap is necessary.
18	10 18 25 31 32 95 72 50	After swapping 10 and 18, we are finished with this round. Note that these are the two smallest elements in the array. The next step is to complete the other half of the recursion.
25	10 18 25 31 32 95 72 50	We will next call `quickSort()` on indices 2 and 3. They will be sorted (notice how we skip a few steps here).
95	10 18 25 31 32 95 72 50	Next we will handle the right side of the array. Again, we will skip some of the details here.
32	10 18 25 31 32 50 72 95	In order to complete the sort of the right side, we need to call `quickSort()` on index 4 and 5.
72	10 18 25 31 32 50 72 95	The final call to `quickSort()` handles index 6 and 7. With this, we are finished.

Performance

In order to characterize the performance of the quicksort, we will look at eight parts of the algorithm:

```
sortQuick(array, n)
  IF n ≤ 1                                          Line 1
    RETURN                                          Line 2
  iDown ← iEnd ← n – 1
  iUp ← 0
  pivotValue ← array[iEnd/2]


  WHILE iUp ≤ iDown                                 Line 3
    WHILE iUp ≤ iEnd and pivotValue > array[iUp]    Line 4
      iUp++
    WHILE iDown ≥ 0 and array[iDown] > pivotValue   Line 5
      iDown--
    IF iUP ≤ iDown                                  Line 6
      SWAP array[iUp++] array[iDown--]

  sortQuick(array, iUp)                             Line 7
  sortQuick(array + iUp, iEnd – iUp + 1             Line 8
```

The overall performace characteristic is described below:

Line	Discussion	Cost
1	The IF statement will be executed exactly once for every iteration of the quick sort. The function itself will get called O(log n) times because it is recursive.	O(log n)
2	We return out of the quick sort algorithm once for every time the list size is 1. This will occur exactly O(n) times.	O(n)
3,4,5	As with the merge sort, these three WHILE loops work together to make one complete pass through the array. Thus the combined cost is O(n). Since the function itself will be called O(log n) times, this is O(n log n)	O(n log n)
6	We swap every time a pair of numbers find themselves on the wrong side of the `pivotValue`. Since a number has a 50% chance of being on the wrong side of the pivot and since a swap moves two misplaced values, it will occur n/4 times. Thus the cost of this line by itself is O(n). Since the function itself will be called O(log n) times, this is O(n log n)	O(n log n)
6,7	Each of these functions will get called O(log n) times. When they do, they will cost O(n log n) per call for increasingly small portions of the list. Because the portions are also decreasing in a logarithmic fashion, the cost is O(n log n).	O(n log n)

The overall cost of the insertion sort is O(n log n) because most of the operations within the outer loop take O(n).

Heap Sort

Heapsort was invented by J. W. J. Williams in 1964. It leverages a data-structure called a heap which provides a convenient way to move the list into a partially sorted state. The heap sort can be characterized by:

Heap Sort	Guaranteed to always be fast	
Average case	O(n log n)	All cases are O(n log n)
Best case	O(n log n)	Again, all cases are the same
Worst case	O(n log n)	Am I being repetitive here?
Space	O(1)	Only a single temporary value is required

The heap sort is unique in that it is always O(n log n). It takes O(n log n) to turn the list into a heap and takes another O(n log n) to turn the heap into a sorted list.

Algorithm in Plain English

As with the bubble sort, the heap sort is not a sort that can realistically be done by a human. The reason is that a heap, while a useful data structure, is not a data structure conveniently done in the physical world and somewhat difficult to explain.

Imagine a binary tree of values that are not in sorted order (and thus not a BST). The constraint instead is that every node has a value at least as large as that of the two child nodes. If all the nodes in the tree are organized with this constraint (every node is greater or equal to its two child nodes), then the binary tree is said to be a heap.

One interesting property of a heap is that the largest value in the tree must be the root element. You can convince yourself of this by imagining another node in the tree that has a higher value. If the parent is smaller, then the node and the parent would have to be swapped. We would then have to check the parent's parent to see if it is bigger. This continues until the largest node in the tree works its way to the top. We call this process "heapify."

The heap sort is simply the process of creating a heap out of an array. With the heap, take the top element out (replacing it from a leaf node from the bottom) and put that top element in the highest slot of the destination array. Now heapify the heap again. Observe how the second largest item is now at the head of the heap (the largest was removed and put in the destination array). Again remove that top element and place it in the second slot of the destination array. This process continues until the heap is empty and the resulting array is sorted.

Detailed Description of Algorithm

A key component of the heap sort is the `heapify()` function. This is the function that turns a given tree into a heap. This `heapify()` function works by calling `percolateDown()` multiple times. Since this is a key aspect of understanding the heap, we will start there.

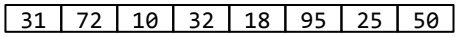
The `percolateDown()` function will assume that all the nodes below the current node are in proper heap order though the status of the current node is in question. Thus if the current node is determined to be larger than either of the children, then nothing remains to be done. However, if one of the children is found to be larger than the current node, the positions of the two are swapped. This means the `percolateDown()` function needs to be run on the child receiving the old current node that was recently swapped. Consider the case where a perfect heap exists except for the head node of the tree. This node happens to be the smallest node in the entire tree! As we run `percolateDown()` on that node, we swap it with the larger of the two children. Now we run `percolateDown()` on that child and, because it is the smallest in the whole tree, we need to swap again.

This continues as the small node percolates down to a root position on the tree. The name "percolate down" refers to this process as smaller nodes are moved or percolated to the bottom of the tree.

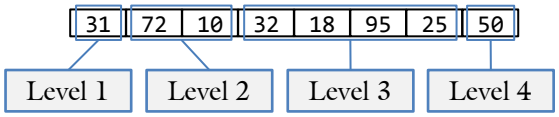
The function `heapify()` makes heavy use of `percolateDown()`. It starts by calling `percolateDown()` on the "last" node in the heap (as defined by level-order traversal) that has a child. After `percolateDown()` is called on that node, the next node is percolated. This continues on up the tree until finally the root node is percolated. When this is finished, the tree is a heap.

With an understanding of `heapify()` and `percolateDown()`, we can describe the `sortHeap()` function. The first step is to turn the array into a heap with `heapify()`. From here we will remove the top element from the heap and replace it with the last node (again, "last" as defined by level-order traversal). This top element will go at the end of the target array. We will call `percolateDown()` on this head node to make the tree a heap again. The process will be continued, each iteration shrinking the size of the heap by one node. When the heap is only a single node, the resulting array is sorted.

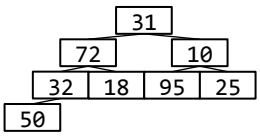
This leaves us with an important implementation detail: how to represent the heap? One option is to use `BNodes`. This works but there might be a better way. Since heaps are always perfectly balanced binary trees, we can represent the heap as an array. Imagine the root at level 0 placed in slot 1 of an array. The root's two child nodes in level 1 are in slot 2 and 3 of an array. Level 3 will have 4 nodes and they go into slot 4, 5, 6. and 7 in the array. If we follow this strategy, we can always find the parent of a given node by dividing the index by 2. Check this by noticing that slots 4 and 5 have a parent that is in slot 2 (4/2 == 2 and 5/2 == 2). This array representation of a binary tree only works when we know that the tree is balanced and complete. Fortunately, it is easy to guarantee this with a heap. Consider the following array: `array == {31, 72, 10, 32, 18, 95, 25, 50}`. This can be represented graphically as follows:



Now we will call the first node the root or the only node at level 1. The next two nodes are at level 2, the next nodes are at level 5 and so on.



Tis array can then be considered as a binary tree:



Algorithm in Pseudocode

It is most convenient to implement a heap in a class. This is because the heap is a thing (a noun) that has supporting operations (helper verbs). The UML class diagram of our `Heap` class is:

Heap
- array : T - num : integer
+ Heap(array : T, num : integer) + getMax() : T + deleteMax() + sort() - percolateDown(index : integer) - swap(i1 : integer, i2 : integer)

We will start with `Heap::percolateDown(int index)`:

```
Heap.percolateDown(index)
  indexLeft ← index x 2
  indexRight ← indexLeft + 1

  IF indexRight ≤ num and
        array[indexRight] > array[indexLeft] and
        array[indexRight] > array[index]
    swap(index, indexRight)
    percolateDown(indexRight)
 ELSE IF indexLeft ≤ num and array[indexLeft] > array[index]
    swap(index, indexLeft)
    percolateDown(indexLeft)
```

Notice how we swap and continue the percolation process only if a child is greater than the siblings and greater than the head. If the head is the largest, then we stop. With `percolateDown()` written, then we can write the constructor which converts the passed array into a heap. Oftentimes, this process is called "heapify."

```
Heap.Heap(arrayInput, numInput)
   array ← arrayInput
   num ← numInput
   FOR index ← num/2 … 1
      percolateDown(index)
```

The final non-trivial method in the `Heap` class is the `sort()` method.

```
Heap.sort()
   WHILE num > 1
      swap(1, num)
      num--
      percolateDown(1)
```

Notice how this serves to swap the head element (largest) with the current end of the array. Once this is done, we can forget about that element (`num--`) and percolate the head.

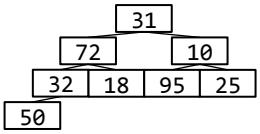
This `Heap` class helps to sort the array but does not adhere to the syntax of a sort algorithm that the client expects. We therefore need one final function:

```
sortHeap(array, num)
    Heap h(array, num)
    h.sort()
```

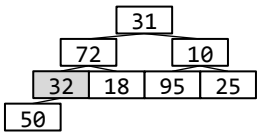
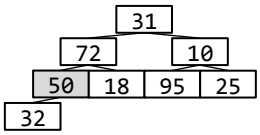
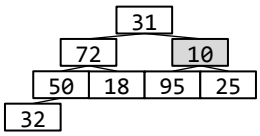
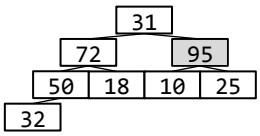
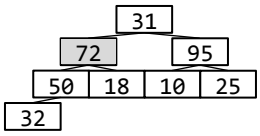
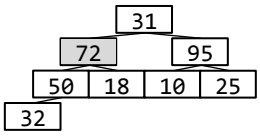
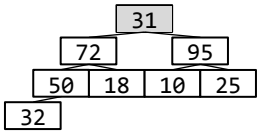
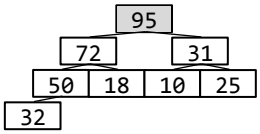
This function simply creates a heap out of the array then performs the sort.

Example

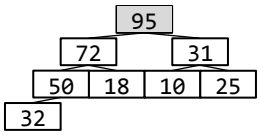
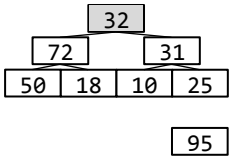
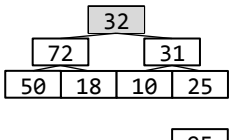
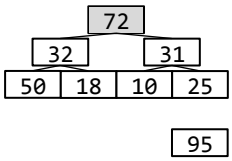
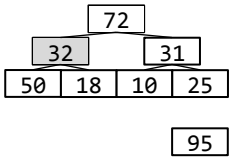
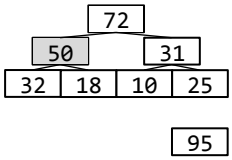
We will illustrate the heap sort with `array == {31, 72, 10, 32, 18, 95, 25, 50}`. Initially we will create a heap by calling `percolateDown(index)` a number of times. Though the heap is stored in memory as a simple array, conceptually it is a tree. Thus we will represent the above array as:



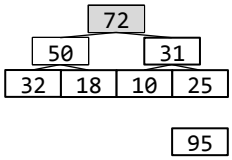
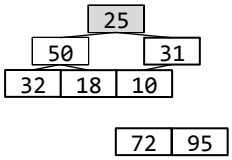
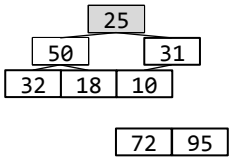
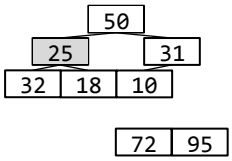
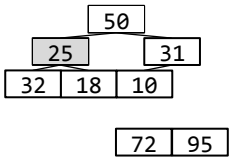
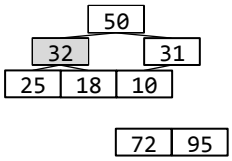
We the first element (slot 0) is at the root of the tree at Level 1. Level 2 are the next two elements and so on.

Before	Description	After
	When we call `percolateDown(3)`, we will ensure that the value in that node (the value being 32) is greater than either of the children (only 50 is a child). This will require us to swap the values.	
	Calling `percolateDown(2)` will place the larger of 10's two children in its spot. In this case, 95 is the larger of the two children.	
	Next `percolateDown(1)` will ensure that 72 is the largest of the two children. Since it is bigger than both 50 and 18, no swapping happens.	
	The final step is to call `percolateDown(0)` on the root. Here 95 is the largest of the siblings so it gets swapped with 31. At this point, we need to check that 31 is in the right spot. Since it is larger than 10 and 25, no additional swapping happens.	

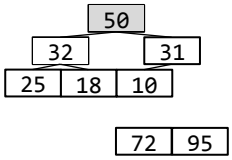
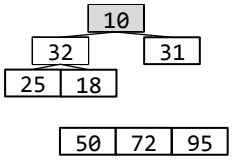
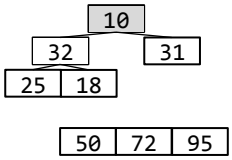
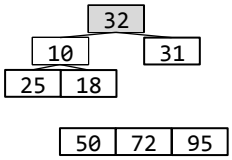
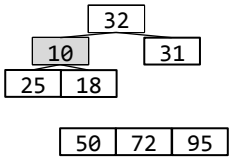
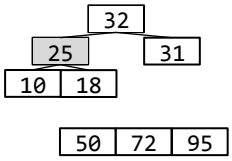
Now that we have a heap (verify that each node is larger than either of the children), we can remove elements off the heap. The root is removed first, placed in the sorted array.

Before	Description	After
	First we swap the largest element in the heap (the root) with the last element in the array. This element is now ignored, no longer part of the tree. Notice how the tree is no longer a heap.	
	We need to return this tree back to the heap state. This requires us to call `percolateDown(0)` on the root. Since 72 > 32, we need to swap these. Now it appears that 32 is not correctly heapified. We need to go fix that slot by calling `percolateDown(1)`.	
	Since 50 > 32, we need to swap these. Fortunately this brings us to the bottom of the tree so we do not have to call `percolateDown()` again.	

Another iteration, this time removing 72 from the heap.

Before	Description	After
	First we swap the largest element in the heap (the root) with the last element in the array. This element is now ignored, no longer part of the tree. Notice how the tree is no longer a heap.	
	We need to return this tree back to the heap state. This requires us to call `percolateDown(0)` on the root. Since 50 > 25, we need to swap these. Now it appears that 25 is not correctly heapified. We need to go fix that slot by calling `percolateDown(1)`.	
	Since 32 > 25, we need to swap these. Fortunately this brings us to the bottom of the tree so we do not have to call `percolateDown()` again.	

One more iteration:

Before	Description	After
	First we swap the largest element in the heap (the root) with the last element in the array. This element is now ignored, no longer part of the tree. Notice how the tree is no longer a heap.	
	We need to return this tree back to the heap state. This requires us to call `percolateDown(0)` on the root. Since 32 > 10, we need to swap these. Now it appears that 10 is not correctly heapified. We need to go fix that slot by calling `percolateDown(1)`.	
	Since 25 > 10, we need to swap these. Fortunately this brings us to the bottom of the tree so we do not have to call `percolateDown()` again.	

This continues, removing the root from the heap and re-heapifying again and again until the heap is empty and the sorted array is full.

Performance

There are three parts to the heap sort: percolate-down, heapify, and finally the heap sort:

Percolate-Down

In order to characterize the performance of the heap sort, first look at the `percolateDown()` algorithm:

```
Heap.percolateDown(index)
  indexLeft ← index  x  2
  indexRight ← indexLeft + 1                        Line 1

  IF (indexRight ≤ num and
      array[indexRight] > array[indexLeft] and
      array[indexRight] > array[index]
    swap(index, indexRight)                         Line 2
    percolateDown(indexRight)                       Line 3
 ELSE IF array[indexLeft] > array[index]
    swap(index, indexLeft)                          Line 4
    percolateDown(indexLeft)                        Line 5
```

The performace characteristic of `percolateDown()` is described below:

Line	Discussion	Cost
1	Only takes constant time to perform this math and make the assignments.	O(1)
2, 4	Swapping two elements only takes constant time.	O(1)
3, 5	With every call to `percolateDown()`, we could possibly need to call `percolateDown()` on a child. Since we are moving down a balanced binary tree, this can happen O(log n) times.	O(log n)

Heapify

Next we need to analyze the performance of the `heapify()` method:

```
Heap.heapify()
   FOR i ← num2/ … 1                                Line 1
      percolateDown(i)                              Line 2
```

The performace characteristic of `percolateDown()` is described below:

Line	Discussion	Cost
1	This will be executed half as many times as there are elements in the array.	O(n)
2	Since each call to `percolateDown()` takes O(log n), the cost of this line is n × O(log n)	O(n log n)

Heap Sort

The final part is the heap sort itself:

```
Heap.sort()
   WHILE n > 1
      swap(1, n)                                    Line 1
      n--
      percolateDown(1)                              Line 2
```

The performace characteristic of `percolateDown()` is described below:

Line	Discussion	Cost
1	This will be executed once for each element in the array.	O(n)
2	Since each call to `percolateDown()` takes O(log n), the overall cost of this line is n O(log n)	O(n log n)

Thus the overall cost of the heap sort is O(n log n).

Exercises

Exercise 11.1

For each of the following descriptions, identify the sort:

a) A pivot is identified. All items larger than the pivot are put on one side, all smaller items are put on the other. Then the two sub-lists delimited by the pivot are recursively sorted.
b) All the items in a collection are inserted into a BST. The tree is then traversed using infix traversal.
c) Systematically interchange pairs of elements until all possible pairs are in the correct order.
d) All the elements in the collection are put in a binary tree. Each parent gets the largest value of the two children. The process is continued until the entire tree is sorted using a level-order traverse.
e) Find the smallest item in a collection and put it in the head of the sorted list, then find the second smallest, and so on.
f) A large list is sub-divided into two small lists. They are recursively sorted then the two sub-lists are combined into a single sorted list.
g) Take the first item from a collection and put it in the sorted list, then take the second item and find where it goes in the sorted list, and so on.

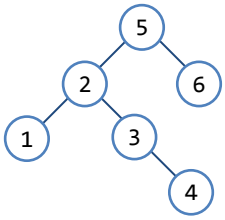
Exercise 11.2

For each of the following sorts, describe the algorithmic efficiency (such as O(n log n)).

- Bubble Sort _____
- Selection Sort _____
- Insertion Sort _____
- Binary Sort _____
- Merge Sort _____
- Quick Sort _____
- Heap Sort _____

Exercise 11.3

Is the following binary tree a heap? Why or why not?



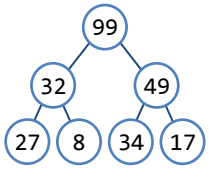
Answer:

_____

Please see page 302 for a hint.

Exercise 11.4

Is the following binary tree a heap? Why or why not?



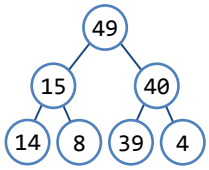
Answer:

_____

Please see page 302 for a hint.

Exercise 11.5

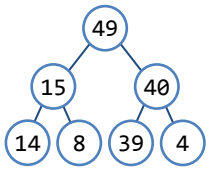
Consdier the following heap. What would happen if the number 16 were added to this heap? Hint: you may have to re-draw the tree to capture the new state.



Please see page 302 for a hint.

Exercise 11.6

Consider the following heap. What would the array representation of this heap look like?



Please see page 308 for a hint.

Exercise 11.7

Consider the following data:

28	66	54	69	19	16	90	62

Sort these numbers using the **Bubble Sort**. To accomplish this, you may wish to cut out 8 cards and write the above numbers on those cards.

Please see page 280 for a hint.

Exercise 11.8

Sort the following data using the **Selection Sort**.

28	66	54	69	19	16	90	62

Please see page 284 for a hint.

Exercise 11.9

Sort the following data using the **Insertion Sort**.

28	66	54	69	19	16	90	62

Please see page 288 for a hint.

Exercise 11.10

Sort the following data using the **Binary Sort** without using Red-Black balancing.

28	66	54	69	19	16	90	62

Please see page 291 for a hint.

Exercise 11.11

Sort the following data using the **Merge Sort**.

28	66	54	69	19	16	90	62

Please see page 296 for a hint.

Exercise 11.12

Sort the following data using the **Quick Sort**.

28	66	54	69	19	16	90	62

Please see page 300 for a hint.

Exercise 11.13

Sort the following data using the **Heap Sort**.

28	66	54	69	19	16	90	62

Please see page 305 for a hint.

Exercise 11.14

Consider the following data:

1	2	3	4	5	6	7	8

Sort these numbers using the **Bubble Sort**. To accomplish this, you may wish to cut out 8 cards and write the above numbers on those cards.

Please see page 280 for a hint.

Exercise 11.15

Sort the following data using the **Selection Sort**.

1	2	3	4	5	6	7	8

Please see page 284 for a hint.

Exercise 11.16

Sort the following data using the **Insertion Sort**.

1	2	3	4	5	6	7	8

Please see page 288 for a hint.

Exercise 11.17

Sort the following data using the **Binary Sort** without using Red-Black balancing.

1	2	3	4	5	6	7	8

Please see page 291 for a hint.

Exercise 11.18

Sort the following data using the **Merge Sort**.

1	2	3	4	5	6	7	8

Please see page 296 for a hint.

Exercise 11.19

Sort the following data using the **Quick Sort**.

1	2	3	4	5	6	7	8

Please see page 46 for a hint.

Exercise 11.20

Sort the following data using the **Heap Sort**.

1	2	3	4	5	6	7	8

Please see page 305 for a hint.

Exercise 11.21

Consider the following data:

90	80	70	60	50	40	30	20

Sort these numbers using the **Bubble Sort**. To accomplish this, you may wish to cut out 8 cards and write the above numbers on those cards.

Please see page 280 for a hint.

Exercise 11.22

Sort the following data using the **Selection Sort**.

90	80	70	60	50	40	30	20

Please see page 284 for a hint.

Exercise 11.23

Sort the following data using the **Insertion Sort**.

90	80	70	60	50	40	30	20

Please see page 288 for a hint.

Exercise 11.24

Sort the following data using the **Binary Sort** without using Red-Black balancing.

90	80	70	60	50	40	30	20

Please see page 291 for a hint.

Exercise 11.25

Sort the following data using the **Merge Sort**.

90	80	70	60	50	40	30	20

Please see page 296 for a hint.

Exercise 11.26

Sort the following data using the **Quick Sort**.

90	80	70	60	50	40	30	20

Please see page 300 for a hint.

Exercise 11.27

Sort the following data using the **Heap Sort**.

90	80	70	60	50	40	30	20

Please see page 305 for a hint.

Problems

Problem 11.1

Implement the **Bubble Sort**. A stub function for the sort is:

```
template <class T>
void sortBubble(T array[], int num)
{
}
```

Please see page 280 for a hint.

Problem 11.2

Implement the **Selection Sort**. A stub function for the sort is:

```
template <class T>
void sortSelection(T array[], int num)
{
}
```

Please see page 284 for a hint.

Problem 11.3

Implement the **Insertion Sort**. A stub function for the sort is:

```
template <class T>
void sortInsertion(T array[], int num)
{
}
```

Please see page 287 for a hint.

Problem 11.4

Implement the **Binary Sort**. A stub function for the sort is:

```
template <class T>
void sortBinary(T array[], int num)
{
}
```

Please see page 291 for a hint.

Problem 11.5

Implement the **Merge Sort**. A stub function for the sort is:

```
template <class T>
void sortMerge(T array[], int num)
{
}
```

Please see page 294 for a hint.

Problem 11.6

Implement the **Quick Sort**. A stub function for the sort is:

```
template <class T>
void sortQuick(T array[], int num)
{
}
```

Please see page 299 for a hint.

Problem 11.7

Implement the **Heap Sort**. A stub function for the sort is:

```
template <class T>
void sortHeap(T array[], int num)
{
}
```

Please see page 304 for a hint.

Sam grew up in an old apartment building in Philadelphia. Every day, the postal worker would drop off the mail to the building manager and she would sort all the letters into their corresponding slots. Some slots (or "bucket") would be empty, some would have just one letter, and others would have a collection of letters. This works much like a hash table.

There are a few things to notice about this scenario. First, there must be a way to map each letter (through the address) into the appropriate slot or bucket. This is called a hash function. Second, letters inside each bucket are also not sorted or organized in any way. If more than one person resides in a given apartment, the letters to all the occupants are still placed in the same slot or bucket. Finally, performance of adding items to the hash and removing items from the hash must be very fast. All these things describe the essential aspects of a hash:

- **Container**: The hash is designed to store a collection of data.
- **Random Access**: All elements are equally accessible by the key index.
- **Order Arbitrary**: The order elements are stored depends on how the hash is implemented.
- **Expandable**: The capacity of the map can be increased to meet the needs of the client.
- **Blind Insertion**: The client is not informed of where items are inserted into the hash.

A hash (also known as a hash map or hash table) is a data structure designed to enable adding and retrieving elements in nearly constant time (otherwise known as O(1)). As a point of comparison, an unsorted array can add elements in O(1) but retrieve elements in O(n). A sorted array can add elements in O(n) but retrieve them in O(log n). A binary search tree can add elements in O(log n) and retrieve them in O(log n). Clearly a hash table is much more efficient.

Definition

A hash table is an abstract data type where elements are organized into a collection of buckets according to a hash function. Notice that there are two parts to this definition: the collection of buckets and the hash function.

The collection of buckets is the part of the hash table that actually stores the data. Since the hash function must have random access to the buckets in the table, the collection of buckets is most often implemented as an array. Each bucket may be empty, contain a single element, or contain many elements. In the latter case, a secondary data structure like a list may be needed.

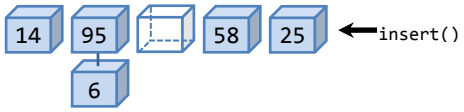
The second part is the hash function. This function maps a given element to a bucket. In other words, it accepts as input an element from the collection and returns the index of the associated bucket in which the element is to reside. Because the behavior of this function will depend a great deal on the type of elements in the collection, the distribution of elements in the collection, and the size of the hash table, the hash function needs to be custom written for each hash application.

The following actions can be performed on a hash:

Name	Example	Description
Create	`HashText h(10);`	Create a text hash with the capacity of 10 buckets.
Assignment	`h1 = h2;`	Assignment, make a copy of a hash.
Empty	`f = h.empty();`	Returns `true` if a hash has no elements, `false` otherwise.
Size	`n = h.size();`	Returns the number of elements currently in a hash.
Capacity	`c = h.capacity();`	Returns the number of buckets in a hash.
Clear	`h.clear();`	Remove all elements from a hash.
Insert	`h.insert(e);`	Add one element to the hash.
Find	`f = h.find(t);`	Returns `true` if `e` in the hash.

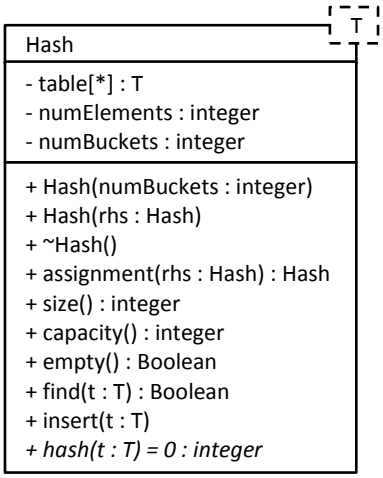
You will notice that the syntax for creating a hash is different than that of creating the other abstract data types: there is no template. Instead a derived class is specified, one that includes the data type of the elements being stored as well as a hash function.

The following figure represents a hash. Note that it appears to the client in much the same way as a set. The only important difference is that the performance characteristic of the hash is much more favorable than that of the set.



Base Class

To demonstrate how a hash function works, it is necessary to first describe a base hash class. The UML class diagram of the base class is the following:



At first glance, this appears to be a normal container ADT. The `table` member variable contains the hash table and `numElements` records how many elements are currently in the hash. The `numBuckets` variable is unique. It refers to the number of buckets in the hash table. Back to our mail slot analogy earlier, `numBuckets` refers to the number of slots where `numElements` refers to the number of individual letters.

The `find()` and `insert()` method work as one would expect in any other container class. However, the `hash()` function is unique. This is the pure virtual function that must be implemented in the derived class specifying how to assign elements to buckets.

Integer Derived Class

To demonstrate how the hash class works, we will present a couple derived hash classes. The first derived class will be an integer hash class, a hash storing arbitrary integers.

```
/****************************************
 * I HASH
 * A simple hash of integers
 ****************************************/
class IHash : public Hash <int>
{
public:
   IHash(int numBuckets)    throw (const char *) : Hash <int> (numBuckets) {}
   IHash(const IHash & rhs) throw (const char *) : Hash <int> (rhs)        {}

   // hash function for integers is simply to take the modulous
   int hash(const int & value) const
   {
      int index = abs(value) % capacity();
      assert(index >= 0 && index < capacity());
      return index;
   }
};
```

Observe how the base class (`Hash <int>`) is a template class. In this case, the template is specified to be an integer. The derived class is not a template class.

There is no non-default constructor. It is necessary for the client to specify the number of buckets so the class can allocate the hash table. In the case of the default and the copy constructor, most of the work is done by the base class.

The non-trivial function is the hash function. The purpose of the hash function is to assign the input `value` into a specific bucket in the hash table. If there are ten buckets (`numBuckets == 10`), then the return value must be between 0 and 9 inclusive.

Float Derived Class

A second example will be more complex. Here we wish to create a hash function for a floating-point number. For a hash to be effective, the hash function needs to evenly distribute values into the various hash buckets. If the hash function returns the same bucket for all values, then the benefits of the hash are eliminated.

The hash function for integers was relatively easy to write. The modulus operator does a good job of distributing values into buckets whether the values are very large (e.g. in the millions) or very constrained (e.g. between 0 and 100). We cannot do quite the same thing with floating-point numbers for several reasons, the first of which is that there is no modulus operator for floats! If we simply try to cast the `float` to an `int`, then we could find ourselves in an overflow condition or, in the case when the numbers are very small, might find that everything gets assigned to the `0` bucket.

There are many ways around this problem. Defining hash functions has a certain art to it as we strive to make the distribution of values in the buckets as even as possible. In our case, we will require the client to specify the minimum and maximum values of the floats.

```
/****************************************
 * F HASH
 * A hash of Floating-point numbers
 ****************************************/
class FHash : public Hash <float>
{
public:
   FHash(int numBuckets, float min, float max) throw (const char *) :
         Hash <float> (numBuckets), min(min), max(max)  {}
   FHash(const FHash & rhs) throw (const char *) :
         Hash <float> (rhs), min(rhs.min), max(rhs.max) {}

   // hash function for strings will add up all the ASCII values
   int hash(const float & value) const
   {
      int index = (int)((value - min) / (max - min) * capacity()) % capacity();
      assert(index <= index && index < capacity());
      return index;
   }
private:
   float min;
   float max;
};
```

The first thing to notice is that our non-default constrictor now requires three parameters. Two (`min` and `max`) are assigned directly to member variables specific to the derived class and one (`numBuckets`) gets sent to the base class constructor. There is no problem here; no rule in C++ mandates that a derived class has to have the same function signature for the constructors as the base class.

The second thing to notice is how much more complex the hash function needs to be. Here the value form the beginning of the range (`value - min`) is divided by the size of the range (`max - min`) to produce a number between `0` and `1`. This is then multiplied by the capacity to give a value between `0` and `numList`. From this point, it should be sufficient to return the result. However, if the client passes a value outside that range, the extra modulus provides protection.

Create

The constructor for the hash function is provided by the derived class and is thus written by the client. The first example is the integer hash:

```
IHash :: IHash(int numBuckets)    throw (const char *);
IHash :: IHash(const IHash & rhs) throw (const char *);
```

Notice how this is not a template class so the template type is not specified.

	Code	Description	State
1	`IHash h1(4);`	Create an integer hash with four buckets.	
2	`IHash h2(2);`	Create an integer hash with 2 buckets.	
3	`IHash h3(h1);`	Copy `h1` including the number of buckets.	

The second example is the floating-point hash:

```
FHash :: FHash(int numBuckets, float min, float max) throw (const char *);
FHash :: FHash(const FHash & rhs)                    throw (const char *);
```

As with `IHash`, no template is specified.

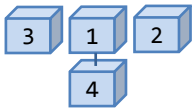
	Code	Description	State
1	`FHash h1(5, 0.00, 0.01);`	Create an floating-point hash with five buckets. Note that the range of values is: 0.00 ≤ value ≤ 0.01	
2	`FHash h2(3, -421.0, 91213.0);`	Create a `float` hash with 3 buckets. Here the range is quite different: -421.0 ≤ value ≤ 91,213.0	
3	`FHash h3(h1);`	Copy `h1` including the number of buckets.	

Clear

Remove all the elements in the hash that have been added with `insert()`. The end result is an empty table.

```
void IHash :: clear();
```

	Code	Description	State
1	`IHash h(3);`	Create a hash of three buckets.	

2	`h.insert(1);`
	`h.insert(4);`
	`h.insert(2);`
	`h.insert(3);`
		Add 4 elements. Details of how `insert()` works are on the following page.	

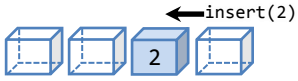
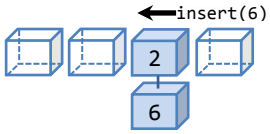
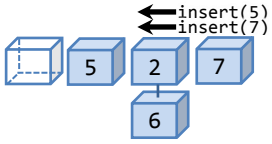
3	`h.clear();`	Remove all the elements so the hash is empty.	

Insert

Items are inserted into a hash in much the same way they are inserted into a set: the client is given a guarantee that the items are stored, but not informed of the details.

```
void Hash <T> :: insert(const T & t) throw (const char *);
```

Note that the `insert()` method belongs in the base class. Its behavior depends a great deal on how the `hash()` function from the derived class behaves. In the following example, we will be using the `IHash` implementation which computes the hash value by applying the modulus operator on the inserted value.

	Code	Description	State
1	`IHash h(4);` `h.insert(2);`	Since `IHash::hash(int value)` takes the modulus of the input parameter, the number `2` will be placed in the `2` bucket.	
2	`h.insert(6);`	Notice how `6 % 4 == 2` so the value `6` will go in the same bucket as `2`. Two elements in the same bucket is called a "collision."	
3	`h.insert(5);` `h.insert(7);`	There is no collision here because `5` goes into bucket `1` (because `5 % 4 == 1`) and `7` goes into bucket `3` (because `7 % 4 == 3`).	

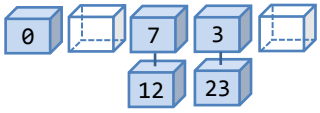
Find

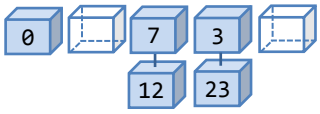
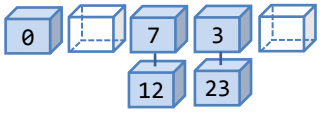
The final operation performed on a hash is `find()`. Here we return the existence of an item in the hash.

```
bool Hash <T> :: find(const T & t) const;
```

As with the `insert()` method, `find()` is defined in the *base* class. Also, as with the `insert()` method, `find()` uses the `hash()` function defined by the *derived* class.

	Code	Description	State

1	`IHash h(5);`
	`h.insert(7);`
	`h.insert(12);`
	`h.insert(3);`
	`h.insert(23);`
	`h.insert(0);`
		Insert several integers into the hash. Two buckets have collisions, one does not.	

2	`assert(false == h.find(5));`	Since `5 % 5 == 0` we look in the 0 bucket for the value. This bucket has the number 0 but not the number 5. Therefore, `find()` will return `false`.	
3	`assert(true == h.find(12));`	Since `12 % 5 == 2` we look in the `2` bucket for the value. This bucket has the number `7` and `12`. Therefore, `find()` will return `true`.	

Implementation

Hashes are implemented as abstract template classes. They are abstract because there exists one pure virtual function: the hash function. A derived class must define a hash function for the hash to work. The hash classes are template classes because the base class needs to support any data type the derived class may specify. However, the derived class is not and cannot be a template. This is because the derived class must know a great deal about the contained data type in order to implement a hash function.

On the surface, implementing a hash function should be very easy: simply index into the hash table with hash function:

```
Hash.insert(value)
  table[hash(value)] ← value
```

Unfortunately, it is not quite this easy. What do you do when there already exists an element in that table? This scenario (which we call "**collision**") governs most of the implementation decisions of a hash function.

There are several standard ways to address insertion collisions: open addressing, reallocation, and chaining.

Open Addressing

Open addressing is the process of looking for other locations in the hash table for an empty slot. For example, if there were 10 slots in the hash table and the hash function returned index 8, then slot 9 might be tried if slot 8 was full. This process would continue until an empty slot is found.

There are several obvious problems with this strategy. First, it would not be possible to store more elements than there are buckets in the hash table. This would violate the desirable "Expandable" properties that most ADTs should exhibit.

The second problem is that it could take linear time O(n) to find an empty slot upon insertion as the hash table nears capacity. In other words, the hash table must not only be non-full, but it must be nearly empty.

The third problem is that it could take linear time O(n) to determine if an element is not present in the hash table. If the element under the hash index is not the requested item, then all the adjacent slots must be checked until a blank slot is found. Then and only then can it be determined that the element is not present in the hash.

The final problem is that deletion is very difficult. First the desired element must be found. Once this occurs and the item is deleted, then a blank spot will exist in the hash table. Normally this would not be a problem. However, if there are 5 collisions for a given slot in the hash table and the second is deleted, it would be very hard to find if the other 3 are in the table. In other words, when searching for the element, it would be a mistake to stop when the first empty slot is found because that empty slot might be recently vacated.

There are three well-known open addressing strategies: linear probing, quadratic probing, and double hashing. **Linear probing** is the simplest probing strategy where the next slot in the hash table is tried if a given location is filled. Linear probing often results in entries being "bunched up" around certain indices. Quadratic probing is similar to linear probing except a more complex equation is used to find the next slot to probe. This minimized the "bunching" problem of linear probing but the other problems with open addressing still exist. Double hashing involves writing a second hash function to find the next slot to probe. All these probing strategies run into the same four problems common to all open addressing strategies and are therefore not commonly used.

Reallocation

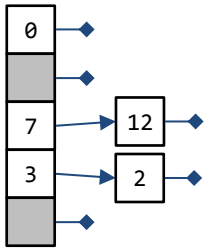
Reallocation is the process of reallocating the hash table once it gets full or nearly full. Since the open addressing problems only manifest themselves in certain circumstances, simply increasing the size of the hash table when these circumstances arise can minimize them.

Reallocation takes O(n) because every item in the table needs to be re-inserted. Clearly this can only be done a small number of times before the expense outweighs any benefit.

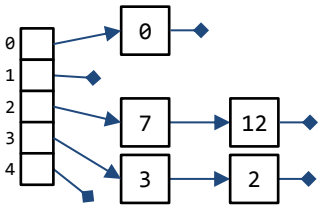
Chaining

Chaining is the process of allowing more than one element to exist in a given slot in the hash table. Typically this is implemented as a linked list but other data-structures could be used. For example, if the number of collisions becomes very large, a set (with the underlying data structure being a Red-Black BST) could be the best implementation. However since a properly sized hash table (where the table is larger than the number of elements) and a good hash function (with elements evenly distributed in the table), the number of elements in a chain is usually not large enough to justify the overhead of anything more complex than a list.

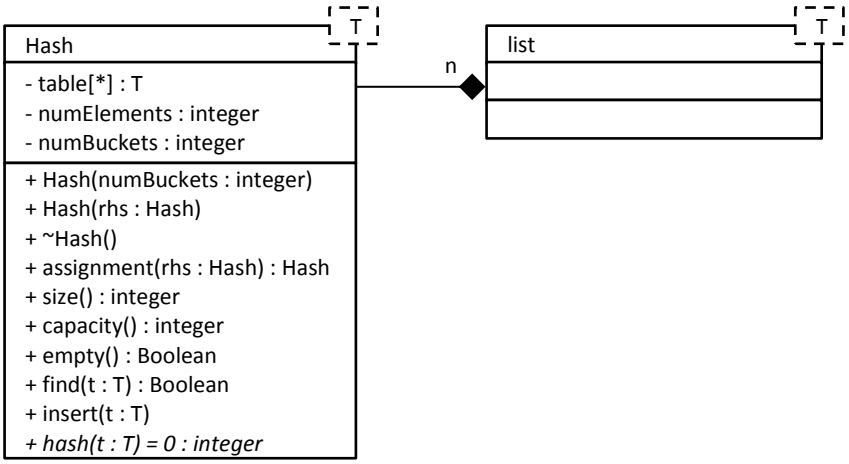
There are three main approaches to chaining. The first is to use a hash table of nodes. Thus every element in the hash table is a data item paired with a pointer to the next element in the linked list. Problems immediately surface with this solution. How can you tell if a slot is vacant? You need an extra Boolean value to indicate whether the slot is full. Also the size of an empty hash table may be excessive if the base data-type is large. However, there is an advantage to this solution. If there are no collisions in the table, then elements can be retrieved from the table without an additional pointer redirection.



The second approach is to use a hash table of pointers. Each pointer may contain the `NULL` address indicating that the slot is not filled, or it may point to a node. The advantage here is that the hash table is very small, consisting only of pointers. When the table is not very dense (i.e. there are many empty slots), this may prove optimal. The disadvantage is that a pointer redirection will always be required when retrieving an element from the hash table.



The final approach is to use a hash table of lists. This works exactly the same as the hash table of pointers except we are letting the list ADT handle all the details of managing the linked list. There are some inefficiencies with this approach because the list ADT has functionality that we do not need in this application (such as being doubly-linked and keeping a member variable containing the number of elements). However the overhead is small and this solution is the most common for chaining:



Constructor

There are two constructors for hash tables: the base class constructor, which *must* be a template, and the derived class which *must not* be a template. The base class constructor allocates the hash table and initializes the two other member variables.

```
Hash.Hash (num)
  numBuckets ← num
  numElements ← 0
  table ← NEW list[num]
```

The derived constructor usually relies on the base constructor to do all the work. An exception would be when the hash function needs additional information, as was the case with our floating-point hash.

```
IHash.IHash(num)
  Hash(num)
```

Insert

The `insert()` method must first identify the correct bucket for the new element to reside in the hash table. This is accomplished with the `hash()` function that returns an index to a bucket. With this accomplished, the next step is to add the new element to the chaining list. If the list is empty, then a new node is added. If the list is not empty, then a new node is added to the list. In both cases, the same `list::insert()` method is used.

```
Hash.insert(e)
  i ← hash(e)
  table[i].insert(e)
```

The cost of the hash function is O(1) and the cost of the insert method is also O(1). Thus insertion is O(1).

Find

The `find()` method returns `true` of the element exists in the hash and `false` otherwise. As with the `insert()` method, it is first necessary to find the appropriate bucket where the element must reside. Again the `hash()` function fulfills this purpose. From here, it is necessary to search the list to find if the element exists. Fortunately the `list::find()` method accomplishes this.

```
Hash.find(e)
  i ← hash(e)
  RETURN table[i].find(e)
```

The cost of the hash function is O(1) but the cost of the `list::find()` method is linear with respect to the size of the list (O(m)). Thus performance of `find()` is O(1) when the hash function does a good job of

distributing elements into buckets and degrades to O(n) as more and more elements are assigned to the same bucket.

Erase

The `erase()` method works much like the `find()` method in that the target item must first be located in the hash table.

```
Hash :: erase(e)
  i ← hash(e)
  it ← table[i].find(e)
  IF it ≠ NULL
    table[i].erase(it)
```

To compute the cost of this algorithm, three things must be determined. First, the cost of the hash function is O(1). Second, the cost of finding an element in the list should be O(1) but might degrade to O(n) in the worst-case scenario. Finally, the cost of removing an element from a list is O(1). Thus, like `find()`, the cost of `erase()` is O(1) in the common case and O(n) in the worst case.

Hash

The final component of implementing the hash is the `hash()` function itself. This function should have three properties: fast, systematic, and uniform.

- **Fast**: It should compute the index for a given element in O(1). If the hash function is slow, then all the performance gains of the hash may be lost.
- **Systematic**: It should systematically assign buckets to elements. This means that no matter how many times the function is called, it will always return the same index for a given element. This is completely determined by the input value and not by anything else.
- **Uniform**. It should evenly distribute elements into buckets. As we learned from `find()` and `erase()`, if too many elements are assigned to the same bucket, then the performance benefits of the hash are diminished. This often means producing very different index values from very similar input data.

The way to accomplish these three imperatives is to leverage what is known about the input. For example, consider a hash function designed to work with individual's first names. One can imagine several options:

a) Use the length of the name to determine the index. This would satisfy the fast imperative (it can be computed in constant time if the string class is used), the systematic imperative ("Bob" would always map to slot 3), but not the uniform imperative. Because there are many names with 5 letters, all of which mapping to the same slot in the hash table, the number of collisions would be excessive.
b) Use a random number generator. This would satisfy the fast imperative (it takes O(1) to call `rand()`), and the uniform imperative (random numbers are evenly distributed. This is what makes them random). However, it would not satisfy the systematic imperative. Calling `hash("Bob")` three different times would result in three different indices.
c) Create a list of all possible names and manually assign each one to an index. This would satisfy the systematic imperative ("Bob" would always be associated with the same index) and the uniform imperative (assuming the mapping of names to indices was done correctly) but would not satisfy the fast imperative. Storing and maintaining such a large table would be excessive and costly.
d) Add up the ASCII values of all the letters in the name and then modulate it by the number of buckets. This would satisfy the fast constraint (addition does not take very long), systematic (the sum is always the same for a given string) and uniform. In other words, it is an excellent candidate for a hash function.

Since option d) satisfies the three desired `hash()` properties, it would be the best choice.

Application

The collection of problems that can be solved with a hash are the same that can be solved with a set. The difference is that the performance characteristics of the hash are much better. So why would one choose the set over the hash? The answer is this: in order to use the set, it is only necessary that the base-type supports the < operator. To use the hash, one must define a hash function. In other words, it is necessary to know a great deal more about the base-type to use a hash. To illustrate this point, the same two examples will be presented for the hash as we used for the set: a hand of playing cards and students in a class.

Hand of Playing Cards

Recall from Chapter 05 that the game of "21" is played as two cards are dealt to each player. Every card is assigned a point value according to its rank. Number cards are given a point value equal to their face value. Face cards are given 10 points. The player gets to choose whether the Ace is worth 1 point or 11. If the player gets 21 points on the deal, an event called "black jack," then they automatically win the hand. Our goal is to write a function to verify if a given hand is a black jack.

The first thing we need to do is to define a hash function for our card class. This function will return an integer for each card. To see how this will work, we must first understand how cards are represented with our program:

```
class Card
{
public:
```

... code removed for brevity ...

```
private:
   // holds the value.  Though there are 256 possible, only 52 are used
   unsigned char value;                // internal representation
```

... code removed for brevity ...

```
};
```

This is rather convenient. The internal representation of a playing card is a value from 0 … 51. This should map nicely to the results of our hash function. In fact, the only thing we need to do is first expose this value to the `CardHash` class and then write our hash function. First, the modification to the `Card` class:

```
// return the internal representation of the Card so we can write a hash function
int Card :: getHashValue() const
{
   return (int)value;
}
```

From here, we can easily write our `CardHash` class:

```
class CardHash : public Hash <Card>
{
public:
   // the usual constructors
   CardHash()                      throw (const char *) : Hash <Card> (52)  {}
   CardHash(const CardHash & rhs) throw (const char *) : Hash <Card> (rhs) {}

   // hash function
   int hash(const Card & card) const
   {
      assert(card.getHashValue() >= 0 && card.getHashValue() < 52);
      return card.getHashValue();
   }
};
```

There are a few things to notice about the changes we made to the `Card` class and our new `HashClass`. The first is that we apparently violated the principle of encapsulation in the `Card` class by revealing an important implementation detail: that the `Card` class stores its value in a number that is between 0 and 52. Unfortunately, this is a necessity when working with a hash. Whereas a set needs to know nothing about the underlying data-type except how to perform the < operator, the hash needs to be aware of the implementation details. As a general rule, the more one knows about the underlying data-type, the better the hash function you can write. Therefore `CardHash` must have some access to `Card`'s underlying data-type. Of course it would be better if *only* `CardHash` had this access, but there is no way in C++ do specify this degree of access.

The second thing to realize is that the default constructor of the `CardHash` class creates a hash that is always 52 buckets in size. Why 52? The full performance potential of the hash cannot be realized if there are many collisions in the hash table. Ideally we would avoid all collisions. By making one bucket for each card, we are removing the possibility of a collision unless more than one deck of cards is used in our game. This is called a "**perfect hash**." As a general rule, we are willing to sacrifice space efficiency for algorithmic efficiency when using a hash.

The final thing to notice about the `CardHash` class is the assert in the `CardHash::hash()` function itself. Because we have a dependency on another class, we need to write an assert to validate an assumption we made as to how `Card::getHashValue()` works. If, for example, the implementers of the `Card` class decided to add jokers to the deck, then the return value of `getHashValue()` might range from $0 \leq$ `getHashValue()` < 54. This would constitute a buffer overrun in our hash array and would result in a bug (or worse!).

With this finished, we can write our function to see if there is one ace in our hand:

```
bool oneAce(const CardHash & hand)
{
   // all possible types of aces
   Card aces[4] =
   {
      Card("AC"),   // ace of clubs
      Card("AS"),   // ace of spades
      Card("AD"),   // ace of diamonds
      Card("AH")    // ace of hearts
   };

   // determine how many aces are in the hand
   int countAces = 0;
   for (int i = 0; i < (sizeof(aces)/sizeof(aces[0])); i++)
      countAces += (hand.find(aces[i]) ? 1 : 0);

   return countAces == 1;
}
```

If we write a similar function to determine if there is one 10-point card in the hand, then we can write our `isBlackJack()` function:

```
bool isBlackJack(const set <Card> & sHand)
{
   return oneAce(sHand) && oneTen(sHand);
}
```

Students in a Class

Our second example will store the students in a section or class. Again, as with the playing cards example, we will need to start by defining a hash function. At this point, our student has only a name. This means our hash function must return an index from text. Recall that our hash function must have three properties:

- **Fast**: It should compute the index for a given element in O(1).
- **Systematic**: It should systematically assign buckets to elements.
- **Uniform**. It should evenly distribute elements into buckets.

One possibility is to compute the hash value based on the first letter of the student's name. This will certainly be fast and systematic. However, it will not be uniform. There are many more students whose name begins with T than begin with Z. Furthermore, there will be only 26 buckets possible. While this might be appropriate for a group of 10 students, it would not for a graduating class of 3,200.

Another possibility is to add up the ASCII values of all the letters in the student's name and divide it by the number of buckets. The remainder of this division would be the hash value. This solution would be relatively fast (O(m) where m is the number of characters in the student's name), systematic (we always return the same has value for a given name), and uniform (the distribution of hash values would be somewhat random). This hash class would appear as the following:

```
class StudentHash : public Hash <Student>
{
public:
   // the usual constructors
   StudentHash(int numBuckets)          throw (const char *) :
                                                     Hash <Student> (numBuckets) {}
   StudentHash(const StudentHash & rhs) throw (const char *) :
                                                     Hash <Student> (rhs)        {}

   // hash function for students
   int hash(const Student & student) const
   {
      int sum = 0;
      for (const char * p = student.getName(); *p; p++)
         sum += *p;
      return *p % numBuckets;
   }
};
```

Unlike the playing card example in the previous pages, we allow the client to specify the number of buckets for the `StudentHash` class. This is done because one would want to have a very different hash size for a group of 10 students than we would for a group of 3,200. As a general rule, the number of buckets should be 1.5 times larger than the number of elements to be stored in the hash. This is done to achieve the minimal number of collisions and while wasting the minimal amount of space.

Just as we completed this `StudentHash` class, it was brought to our attention that every student was assigned a unique student ID. This is a number given to each student they day he or she registers at the local school. The number of a random 8-digit number that, when generated by the school registration system, is verified to be unique. We can retrieve this ID with the following method:

```
unsigned int Student :: getID() const;
```

This is quite a discovery! Before we had to settle for O(m) in our hash function where m is the number of letters in the student's name. Furthermore, we were somewhat uneasy that the sum of the ASCII letters would give us uniform distribution of hash values throughout a variety of hash table sizes. Both of these issues can be addressed with our new hash function:

```
class StudentHash : public Hash <Student>
{
public:
   // the usual constructors
   StudentHash(int numBuckets)          throw (const char *) :
                                                    Hash <Student> (numBuckets) {}
   StudentHash(const StudentHash & rhs) throw (const char *) :
                                                    Hash <Student> (rhs)        {}

   // hash function for students
   int hash(const Student & student) const
   {
      return student.getID() % numBuckets;
   }
};
```

Let's see how this hash function does according to our three metrics:

- **Fast**: The `Student::getID()` method simply returns the value of the member variable which means it is accomplished in just one or two assembly instructions. This, combined with the modulus operator, is not only O(1), but probably less than three assembly instructions.
- **Systematic**: Unless the student's ID changes with time (which it shouldn't but we cannot verify), this hash function should always return the same value for a given student.
- **Uniform**. We were told that the student's ID was randomly generated at registration time. If this is true, then the collection of all student IDs should be randomly distributed over the set of 8-digit numbers. Taking the modulus of this value should thus yield a random distribution over all values of `numBuckets`.

As you can see, this hash function performs much better than the previous one operating off of the students' names. In other words, either implementation of `StudentHash` should behave the same. However, the implementation using the `getID()` method as opposed to the `getName()` method will have better performance characteristics.

Examples

Example 12.1

Consider the following code:

```
#define SIZE 5

int main()
{
   string table[SIZE];

   table[hash(string("int" ))] = string("int" );
   table[hash(string("bool"))] = string("bool");
   table[hash(string("char"))] = string("char");

   cout << '(' << table[0];
   for (int i = 1; i < SIZE; i++)
      cout << ", " << table[i];
   cout << ")\n";

   return 0;
}
```

This code has the following hash function:

```
int hash(const string & s)
{
   int value = 0;
   for (const char * p = s.c_str(); *p; p++)
      value += *p;
   return value % SIZE;
}
```

What is the output of the program?

Answer:

This hash function sums up the ASCII values of characters similar to the Students example on page 325. The values are:

Input	Computation	Hash Value
`int`	(105 + 110 + 116) % 5 = 331 % 5 = 1	`1`
`bool`	(100 + 111 + 111 + 108) % 5 = 430 % 5 = 0	`0`
`char`	(99 + 104 + 97 + 114) % 5 = 414 % 5 = 4	`4`

Fortunately there are no collisions in this table. The output is:

```
(bool, int, , , char)
```

Example 12.2

Consider the following code:

```
#define SIZE 5

int main()
{
   int input[5] = {26, 42, 15, 44, 92};
   int table[5] = {0,  0,   0,  0,  0};
   for (int i = 0; i < SIZE; i++)
      table[hash(input[i])] = input[i];

   cout << '(' << table[0];
   for (int i = 1; i < SIZE; i++)
      cout << ", " << table[i];
   cout << ")\n";

   return 0;
}
```

This code has the following hash function:

```
int hash(int value)
{
   return value % SIZE;
}
```

What is the output of the program?

Answer:

This hash function return values are:

Input	Computation	Hash Value
26	26 % 5 = 1	1
42	42 % 5 = 2	2
15	15 % 5 = 0	0
44	44 % 5 = 4	4
92	92 % 5 = 2	2

Notice how there is a collision in this hash table. Since the implementation does not handle collisions with open addressing, reallocation, or chaining, we will loose data. In this case, the value 42 is overwritten with the value 92 in slot 2. The output is:

```
(15, 26, 92, 0, 44)
```

Example 12.3

Using the floating point hash defined on page 318:

```
class FHash : public Hash <float>
{
public:
   FHash(int numBuckets, float min, float max) throw (const char *) :
         Hash <float> (numBuckets), min(min), max(max)  {}
   FHash(const FHash & rhs) throw (const char *) :
         Hash <float> (rhs), min(rhs.min), max(rhs.max) {}

   // hash function for strings will add up all the ASCII values
   int hash(const float & value) const
   {
      int index = (int)((value - min) / (max - min) * capacity()) % capacity();
      assert(index <= index && index < capacity());
      return index;
   }
private:
   float min;
   float max;
};
```

Consider following code:

```
{
   FHash temperature(8, -40.0, 110.0);

   temperature.insert(16.0);
   temperature.insert(72.4);
   temperature.insert(-21.5);
   temperature.insert(101.1);
   temperature.insert(98.6);
}
```

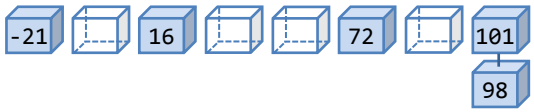
What is the state of the hash at the end of execution?

Answer:

This hash function sums up the ASCII values of characters similar to the Students example on page 325. The values are:

Input	Computation	Hash Value
16.0	(16.0 - -40.0) / (110.0 - -40.0) × 8 = 56.0 / 150.0 × 8 = 2.986	2
72.4	(72.4 - -40.0) / (110.0 - -40.0) × 8 = 112.4 / 150.0 × 8 = 5.995	5
-21.5	(-21.5 - -40.0) / (110.0 - -40.0) × 8 = 18.5 / 150.0 × 8 = 0.986	0
101.1	(101.1 - -40.0) / (110.0 - -40.0) × 8 = 141.1 / 150.0 × 8 = 7.525	7
98.6	(98.6 - -40.0) / (110.0 - -40.0) × 8 = 138.6 / 150.0 × 8 = 7.392	7

There is one collision in this table and it is handled by chaining. The state of the temperature hash at the end of execution is:



Notice how half of the slots in the hash table are empty and there is one collision.

Exercises

Exercise 12.1

Consider the following code:

```
#define SIZE 5

int main()
{
   string table[SIZE];

   table[hash(string("dog"))] = string("dog");
   table[hash(string("cat"))] = string("cat");

   cout << '(' << table[0];
   for (int i = 1; i < SIZE; i++)
      cout << ", " << table[i];
   cout << ")\n";

   return 0;
}
```

This code has the following hash function:

```
int hash(const string & s)
{
   int value = 0;
   for (const char * p = s.c_str(); *p; p++)
      value += *p;
   return value % SIZE;
}
```

What is the output of the program?

Answer:

_____

Please see page 327 for a hint.

Exercise 12.2

Consider the same program as Exercise 12.1 except the following hash function is used:

```
int hash(const string & s)
{
   return s.size() ?
      abs(s[0] - s[1]) % SIZE : 0;
}
```

What is the output of the program?

Answer:

_____

Please see page 327 for a hint.

Exercise 12.3

Consider the following code:

```
#define SIZE 11

int main()
{
   int input[11] = {26, 42, 15, 44, 92, 59, 40, 36, 12, 60, 80};
   int table[11] = {0,  0,   0,  0,  0,  0,  0,  0,  0,  0,  0};
   for (int i = 0; i < SIZE; i++)
      table[hash(input[i])] = input[i];

   cout << '(' << table[0];
   for (int i = 1; i < SIZE; i++)
      cout << ", " << table[i];
   cout << ")\n";

   return 0;
}
```

This code has the following hash function:

```
int hash(int value)
{
   return value % SIZE;
}
```

What is the output of the program?

Answer:

_____

Please see page 317 for a hint.

Exercise 12.4

Consider the same program as Exercise 12.3 except the following hash function is used:

```
int hash(int value)
{
   return value / 10;
}
```

What is the output of the program?

Answer:

_____

Please see page 31746 for a hint.

Exercise 12.5

Consider the following hash function:

```
class IHash : public Hash <int>
{
public:
   IHash(int numBuckets   ) : Hash <int> (numBuckets) {}
   IHash(const IHash & rhs) : Hash <int> (rhs)        {}

   int hash(const int & value) const
   {
      int index = (value / 10) % capacity();
      assert(0 <= index && index < capacity());
      return index;
   }
};
```

The following numbers will be inserted into the hash in order.

```
26 28 93 3 85 73 31 55 92 98 31 11 52 89 19
```

What is the state of the hash when there are five buckets?

```
IHash data(5);
```

Please see page 327 for a hint.

Exercise 12.6

The same problem as 12.5 except there are ten buckets:

```
IHash data(10);
```

Please see page 327 for a hint.

Exercise 12.7

Consider the following hash function:

```
class IHash : public Hash <int>
{
public:
   IHash(int numBuckets   ) : Hash <int> (numBuckets) {}
   IHash(const IHash & rhs) : Hash <int> (rhs)        {}

   int hash(const int & value) const
   {
      int index = (value % 10 + value / 10) % capacity();
      assert(0 <= index && index < capacity());
      return index;
   }
};
```

The following numbers will be inserted into the hash in order.

```
26 28 93 3 85 73 31 55 92 98 31 11 52 89 19
```

What is the state of the hash when there are four buckets?

```
IHash data(4);
```

Please see page 327 for a hint.

Exercise 12.8

The same problem as 12.7 except there are seven buckets:

```
IHash data(7);
```

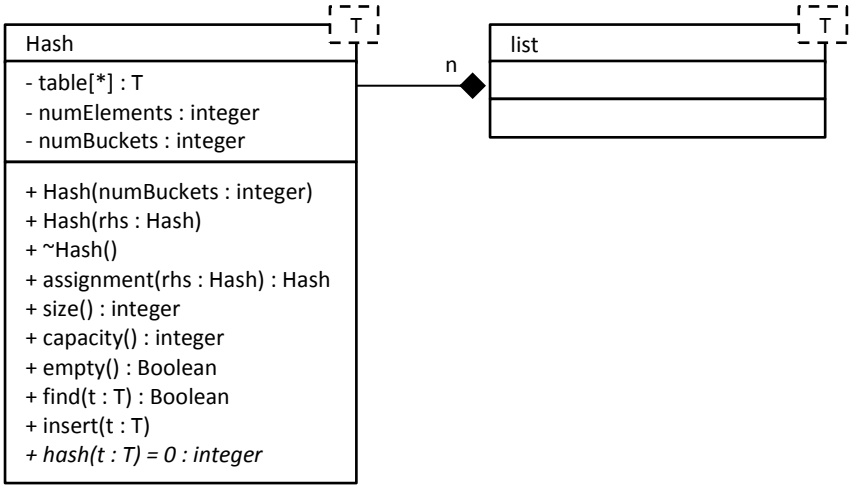
Please see page 327 for a hint.

Problems

Problem 12.1

Create a class to capture the notion of a hash. This class will store a collection of elements and be able to perform constant-time insertion and lookup. Additionally, this class will need to support any data type. Since a hash function needs to be provided and since hash functions are very much data type dependent, we will have to do this in two steps. First, create an abstract class representing a hash where the `hash()` method is a pure virtual function. Then create a collection of derived classes, each specific to a given data type.

This implementation is to use list-based chaining.



Please see page 322 for a hint.

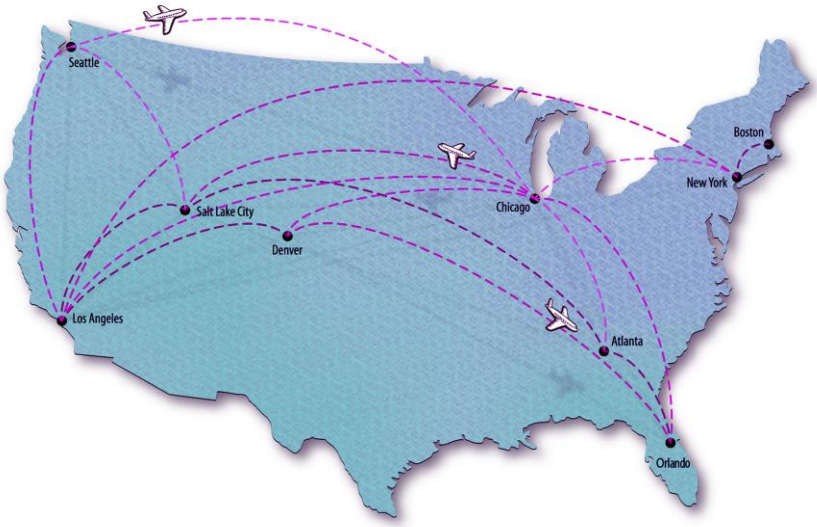
Problem 12.2

Using the Hash class from Problem 12.1, implement a spell checker. Start by reading a dictionary from a file into a Hash object, then walk through a file looking for spelling errors. If a word in the file is not in the dictionary, it is determined to be spelled incorrectly. Otherwise, it is assumed to be correct.

It is critically important that a spell checker be fast. Ideally, we should be able to confirm the spelling of a word in constant O(1) time. Clearly, a hash table is a great tool for the job.

Please create a derived class from `Hash` designed specifically for this spell-checker job. It needs to read a dictionary from a file, which will be inserted into the hash table. You can make the capacity of the hash anything you like. Here you need to balance the amount of wasted space with the number of collisions..

Sue is planning to fly from Salt Lake City to Boston. As she reviews the various flight options, Sue notices that there are no direct flights. From Salt Lake, she can get a flight to Orlando, Denver, and Las Vegas. From there, the options compound. All in all, there are dozens of flights but no clear path from Salt Lake City to Boston. In fact, it is not even clear that it is possible to make this flight.



This problem can be represented as a graph: the collection of airports to which the flights travel combined with the routes that connect airports. From this graph, Sue realizes that the only way to Boston is through New York. There are only two ways to New York: through Los Angeles or Chicago. Since there is a direct route directly from Salt Lake City to both Los Angeles and Chicago, she knows it is possible to make the flight.

This pathing-finding problem manifests itself in many contexts. You use it when trying to decide how to drive or walk to the grocery store. Packets of data use it when trying to find their way from your web site to your browser. In all these cases, there are vertices (points on the graph), and edges (connections between the edges). The combination of the two are a graph.

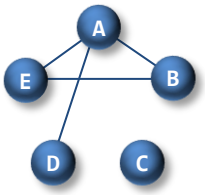
The essential characteristics of a graph are:

- **Relations**: The graph stores the existence of edges between values, but no user data.
- **Random Access**: Any edge can be queried at any time.
- **Edge Insertion Efficient**: It is easy to add an edge to a graph.
- **No Vertex Insertion**: Many implementations of graphs do not allow for adding vertices.

Definition

A vertex is a single named point in a graph. Other names for verticies include node, point, junction, or endpoint. In the case of our flight plan example, a vertex is an airport. Vertices are commonly represented in a graph as a dot or as a circle with a label such as `A`.

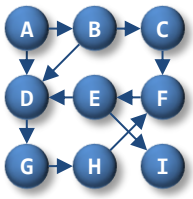
Two vertices on a graph can be connected with an edge. Other names for an edge include: arc, line, branch, and link. An edge is represented as an ordered pair of verticies. For example, an edge between `A` and `B` is represented as (`A`, `B`). An undirected edge is an edge were (`A`, `B`) implies (`B`, `A`). In other words, it is possible to travel both from `A` to `B` and from `B` to `A`. The following depicts an undirected graph. Notice that the edges do not have arrows.



There are several useful concepts in connection with undirected graphs:

- **Degree**: The number of edges associated with a vertex. The degree of vertex `A` is 3.
- **Order**: The number of vertices in the graph. The order of the above graph is 5.
- **Isolated Vertex**: A vertex without an associated edge such as `C`. Here the degree is zero.
- **Leaf Vertex**: A vertex with one edge and the degree is 1. `D` is a leaf vertex.
- **Path**: A sequence of edges connecting two vertices. A path from `B` to `D` is `B`, `A`, `D`.

A directed graph consists of edges with arrows. This is because each edge has a direction. In other words, edge (`A`, `B`) does not imply there is an edge (`B`, `A`). The following depicts a directed graph.



There are several useful concepts associated with directed graphs:

- **In-degree**: The number of edges pointed to a given vertex. The in-degree of vertex `B` is 1.
- **Out-degree**: The number of edges pointed away from a vertex. The out-degree of vertex `B` is 2.
- **Order**: This is exactly the same as an undirected graph. Here the order is 9.
- **Source Vertex**: A vertex with an in-degree of zero. `A` is the only source vertex in the above graph.
- **Sink Vertex**: A vertex with an out-degree of zero. `I` is the only sink vertex in the above graph.
- **Path**: This works the same for directed graphs as for undirected graphs.

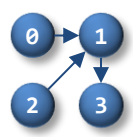
Note that a singly-linked list is a special form of a directed graph. Each vertex (called a node) has an in-degree of zero or one and an out-degree of zero or one. A singly-linked list also has the constraint that any two vertices are connected with exactly one path. A doubly-liked list is an undirected graph.

A tree is also a special form of graph, one where any two vertices are connected by exactly one path. Notice that this definition is similar to that of a linked list. This is because a linked list is a special type of tree. Both linked lists and trees have an indegree of one. The difference between a linked list and a tree, however, is that the verticies of a tree can have an outdegree of greater than one.

From the standpoint of an abstract data type, a graph has the following operations:

Name	Example	Description
Create	`Graph g(10);`	Create a new graph with 10 verticies.
Assignment	`g1 = g2;`	Make a copy of a graph.
Empty	`f = g.empty();`	Returns `true` if a graph has no verticies, `false` otherwise.
Size	`n = g.size();`	Returns the number of verticies currently in a graph.
Capacity	`c = g.capacity();`	Returns the number of possible edges in a graph.
Clear	`g.clear();`	Remove all edges from a graph.
Add	`g.add(v1, v2);`	Create an edge between v1 and v2.
Is Edge	`f = g.isEdge(v1, v2);`	Returns true if there is an edge between v1 and v2.
Find Edges	`s = g.findEdges(v);`	Find all the edges connected to a given vertex.
Find Path	`l = g.findPath(v1, v2);`	Returns a list of vertices representing the path between v1 and v2.

Aside from the standard container interfaces, the graph has an `add()`, `isEdge()`, `findEdge()`, and `findPath()` method. The graph can be represented with the following figure:

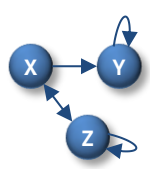


In this example, there are four vertices so the order is 4. There are three edges so the size is three. The edges are (`0`, `1`), (`1`, `3`), and (`2`, `1`). Thus the in-degree of `1` is two (from nodes `0` and `2`) and the out-degree is one (to node `3`). There is a path from `0` to `3`, but there is no path from `3` to `0`.

Another way to represent the graph is with an adjacency-matrix. This consists of two matricies, the first representing the edges (called *adj*) and the second representing the names of the verticies (called *data*).

$$adj = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \qquad data = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

In the adjacency-matrix, the rows represent when an edge originates from and the columns represent where an edge terminates. In this example, we can see that an edge originates in X and terminates in Y because the first row and second column has a '1' in it. This above adjacency matrix corresponds to the following:



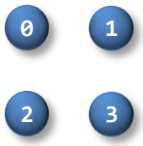
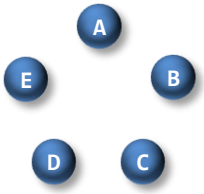
Notice how the `Graph` data structure does not store arbitrary data as the other containers do. Instead, it represents whether edges exist between vertices.

Create

The graph creation process requires the client to specify the order or number of vertices.

```
Graph :: Graph(int numNodes);
```

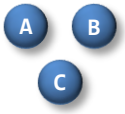
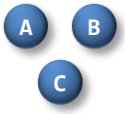
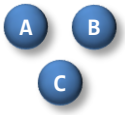
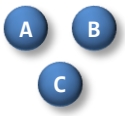
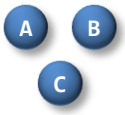
Notice how the client does not specify a template.

	Code	Description	State
1	`Graph g1(4);`	There is no default constructor. The client must specify the order (number of verticies) upon creation.	
2	`Graph g2(5);`	This time there are five vertices in the graph. Note that we do not specify the naming convention of the vertices at graph creation time. The vertices are named with letters in this example for illustration purposes only.	

Standard Container Interfaces

The first thing to remember is that a graph is not a container. Therefore it does not make much sens to use the standard container interfaces on it. Nevertheless, a few are commonly employed:

```
int Graph :: size() const;
int Graph :: capacity() const;
bool Graph :: empty() const;
void Graph :: clear();
```

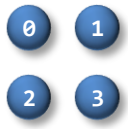
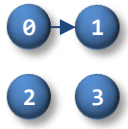
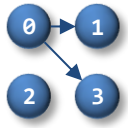
	Code	Description	State
1	`Graph g(3);`	Create a graph of three verticies. Initially there are no edges of course.	
2	`assert(g.size() == 3);`	There are three verticies in this graph. Remember that `size()` returns the number of verticies rather than edges.	
3	`assert(g.capacity() == 9);`	Even though this graph has no edges, there are 9 edges possible. They are: { (A, A), (A, B), (A, C), (B, A), (B, B), (B, C), (C, A), (C, B,), (C, C) }. The `capacity()` method returns the number of possible edges.	
4	`assert(g.empty() == false);`	This is a silly method because it always returns false. We always create a graph with at least one vertex so the size is never zero.	
5	`g.clear();`	This removes all edges. Note that most containers report that they are empty after `clear()`. Not so with a graph!	

Add

The setter of the `Graph` class is the `add()` method. This method adds an edge to a graph by specifying two vertices.

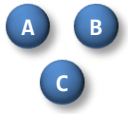
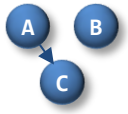
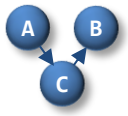
```
void Graph :: add(const Vertex & v1, const Vertex & v2);
```

It is possible to add an edge to the graph between any vertices.

	Code	Description	State
1	`Graph g(4);`	A graph with four verticies. There are no edges initially.	
2	`g.add(Vertex(0), Vertex(1));`	Add an edge between verticies `0` and `1`.	
3	`g.add(Vertex(0), Vertex(3));`	Now there is a second edge in our graph.	

These vertices can be either of the `Vertex` class or any derived class. The client can specify a unique naming convention for the vertex by defining a new derived class. For example, the vertices can be named {0, 1, 2, … } or {A, B, C, ….} or even {"Salt Lake City", "Denver", "Orlando"… }. In each case, the client-provided derived class needs to identify how to convert from the textual name of a vertex into an index which the Graph class uses.

For example, consider a `Vertex` derivation where the labels are letters. We will call this `LVertex`. Now internally, all vertices are stored as a number. This means our `LVertex` class will map 0 to A, 1 to B, and so on.

	Code	Description	State
1	`Graph g(3);`	A graph with three verticies. There are no edges initially. Notice how we do not need to specify the flavor of Vertex we are using.	
2	`g.add(LVertex('A'), LVertex('C'));`	Add an edge between verticies `A` and `C`.	
3	`g.add(LVertex('C'), LVertex('B'));`	A second edge will be added between verticies `C` and `B`.	

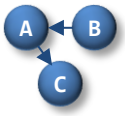
Is Edge

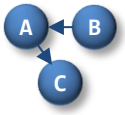
One of the main getters of the Graph class is the `isEdge()` method which indicates whether an edge exists between two vertices.

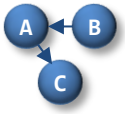
```
bool Graph :: isEdge(const Vertex & v1, const Vertex & v2);
```

As with the `add()` method, either the Vertex or any derived class can be used as the input parameter.

	Code	Description	State

1	`Graph g(3);`
	`g.add(LVertex('A'), LVertex('C'));`
	`g.add(LVertex('B'), LVertex('A'));`
		Two edges are created. One between `A` and `C`, and a second between `B` and `A`.	

2	`if (g.isEdge(LVertex('A'), LVertex('C')))` `  cout << "Edge!";`	There is an edge between A and C. This means the following will be displayed: `Edge!`	

3	`if (g.isEdge(LVertex('A'),`
	`  LVertex('B')))`
	`  cout << "Edge!";`
		There is no edge between `A` and `B`. This is because we have a directed graph. Note that there is an edge between `B` and `A`.	

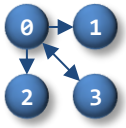
Find Edges

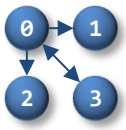
It is possible to retrieve a set of all the edges relating to a given vertex.

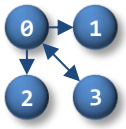
```
set <Vertex> Graph :: findEdges(const Vertex & v);
```

Since the output is a set, there is no guarantee of the order of the output.

	Code	Description	State

1	`Graph g(4);`
	`g.add(Vertex(0), Vertex(1));`
	`g.add(Vertex(0), Vertex(2));`
	`g.add(Vertex(0), Vertex(3));`
	`g.add(Vertex(3), Vertex(0));`
		In our four vertex graph, we have four edges. Notice how there is an edge both between `0` and `3`, as well as one between `3` and `0`.	

2	`set <Vertex> s;`
	`s = g.findEdges(Vertex(0));`
	`cout << s.size();`
		`findEdges()` will find all the edges that lead from `0`. The set `s` will consist of the vertex `1`, `2`, and `3`. Thus the size of `s` is:
		`3`
		This corresponds to the outdegree of `3`.
			

3	`s = g.findEdges(Vertex(1));` `cout << s.size();`	There is no edge that leads from `1` even though there is one edge that leads into `1`. Thus the outdegree is `0`: `0`	

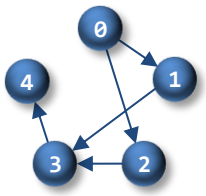
Find Path

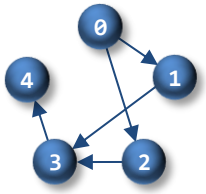
The `findPath()` method returns a list of nodes leading from one vertex to another.

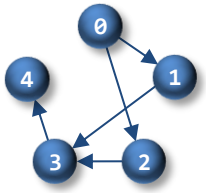
```
list <Vertex> Graph :: findPath(const Vertex & v1, const Vertex & v2);
```

This method guarantees to find *a* shortest path, but not *the* shortest path. This means that if one path from `A` to `B` consists of three edges and another consists of four, the path with three edges will be returned. However, if there are multiple paths with three edges, there is no guarantee which will be returned.

	Code	Description	State

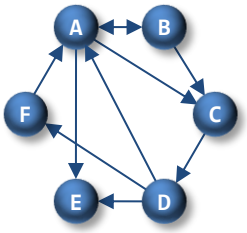
1	`Graph g(5);`
	`g.add(Vertex(0), Vertex(1));`
	`g.add(Vertex(0), Vertex(2));`
	`g.add(Vertex(1), Vertex(3));`
	`g.add(Vertex(2), Vertex(3));`
	`g.add(Vertex(3), Vertex(4));`
		In our five vertex graph, there is no direct path from 0 to 4.	

2	`list <Vertex> l;`
	`l = g.findPath(Vertex(0), Vertex(4));`
	`cout << l.size();`
		`findPath()` could either return the path {`0`, *`1`*, `3`, `4`} or it could find the path {`0`, *`2`*, `3`, `4`}. Both of these paths utilize three edges and four vertices.
		`4`
			

3	`l = g.findPath(Vertex(4), Vertex(0));` `cout << l.size();`	Notice that vertex `0` is a source and vertex `4` is a sink. This is because the in-degree of `0` is zero and the out-degree of `4` is also zero. Therefore there can be no path between `4` and `0`. In this case, `findPath()` will return an empty list. `0`	

Implementation

There are four main ways to implement a graph: as a collection of interconnected nodes, a set of pairs indicating edges, an adjacency-list, or an adjacency-matrix. We will consider each of these implementations with a simple six-node graph:

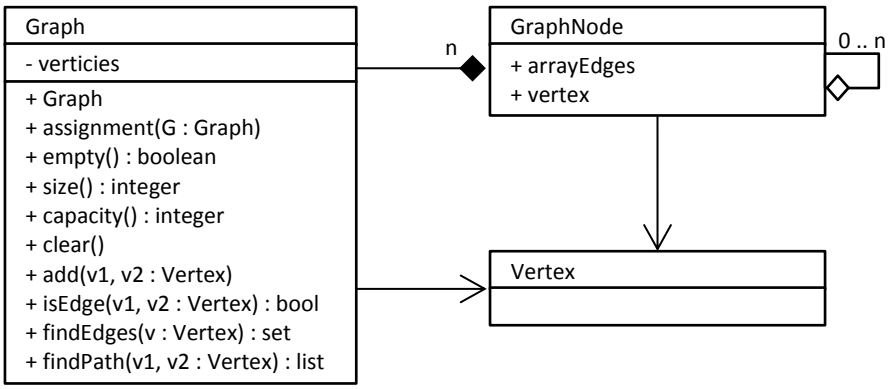


In the following examples, we are attempting to find the path from `A` to `F`.

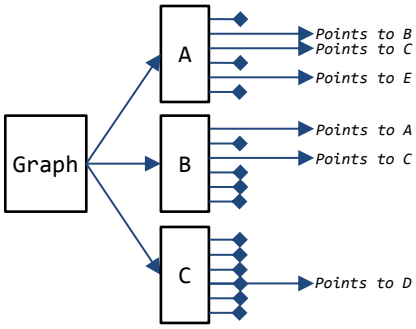
```
path = g.findPath(VertexLetter('A'), VertexLetter('F'));
```

Interconnected Nodes

Perhaps the most obvious option is to represent a graph as a collection of interconnected nodes. Each node can point to other nodes with pointers much like a linked list or a tree. Recall that there is one pointer in a singly linked list node, two pointers in a doubly linked list node, and three pointers in a binary tree node. Since there can be as many edges for a given vertex as there are vertices in the graph, a much larger number of pointers is needed. In fact, an array of pointers is needed.



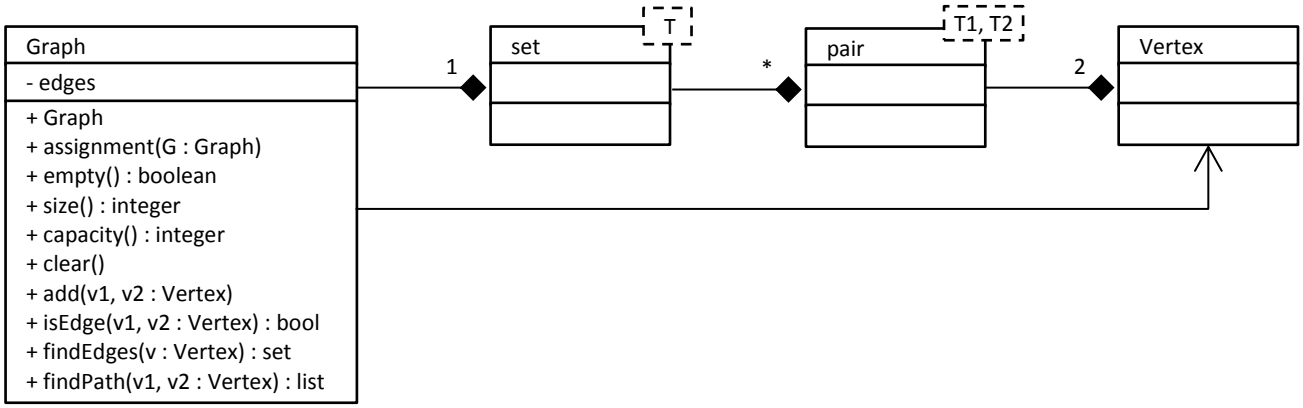
On the surface, this seems like a viable solution. However, problems surface immediately. In a linked list or a tree, every node is connected to the collection with a pointer. No such guarantee exists with a graph; a leaf node may have no adjacent vertices. The second problem is that the out-degree of a given vertex is limited only by the order (number of vertices in the graph). A graph with a thousand vertices would require 999 pointers per node, making the size of a node O(n) and the size of the graph $O(n^2)$.



Notice that the above figure only depicts three of the nodes. The other three were omitted for space reasons.

Set of Edges

Another implementation would be to represent a graph as a set of pairs of vertices. Each pair in the set would constitute one edge.

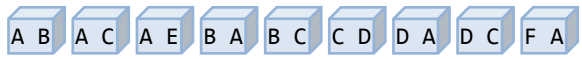


Clearly this would be much more space efficient than the node implementation. Since every edge would constitute only two vertices, the size of the graph would be proportional to the number of edges. Unfortunately, there are other performance implications. For example, consider what would be required in order to find if an edge exists between two vertices. It would be necessary to iterate through the entire list. In other words, it would be O(n). For this reason, the set implementation is not commonly used for the graph unless the size (number of edges) is very small.

The mathematical way to represent a graph as a set of edges is the following:

$$g = \left\{ \begin{array}{c} (A,B),(A,C),(A,E), \\ (B,A),(B,C), \\ (C,D), \\ (D,A),(D,C), \\ (F,A) \end{array} \right\}$$

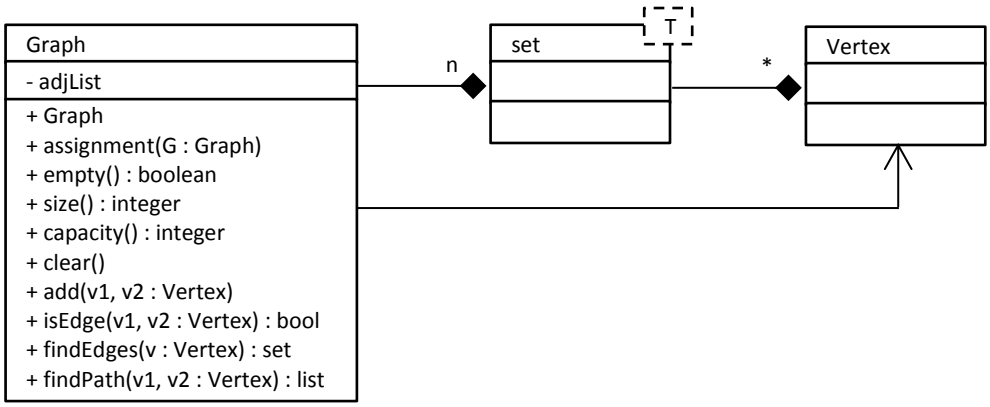
The same graph can be represented using our ADT figure:



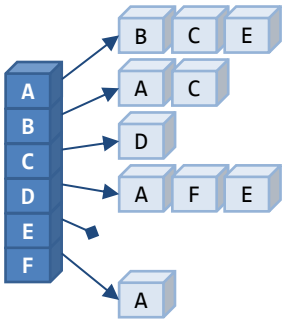
There are many applications where this is the best implementation. Consider, for example, a bus company like Greyhound. There might be a thousand or more individual stops (nodes) that are accessible through their service. Most of these stops are only reachable by a single route. This means that their graph is sparse: a large number of nodes and a small number of edges. For applications like these, the set implementation can be optimal.

Array of Sets

A third implementation is an array of sets. This common implementation is a hybrid between the node implementation and the set implementation. Each vertex is given a slot in an array. Attached to each slot is a set or list of nodes adjacent to the vertex corresponding to the slot. For example, consider our 6-node graph. Vertex '`A`' will be assigned to slot `0`, vertex '`B`' will be assigned to slot `1` and so on. The next part of the adjacency list implementation is how each edge is represented. Since the slot in the array indicates the "source" component of an edge, only a "destination" vertex is required to complete the edge description. Thus a list or set of vertices attached to each slot in the array will complete the edge description.



As long as the out-degree of the nodes is not too great, this can be an efficient solution for most applications. A graphical representation of an array of lists for our six-vertex graph is the following:

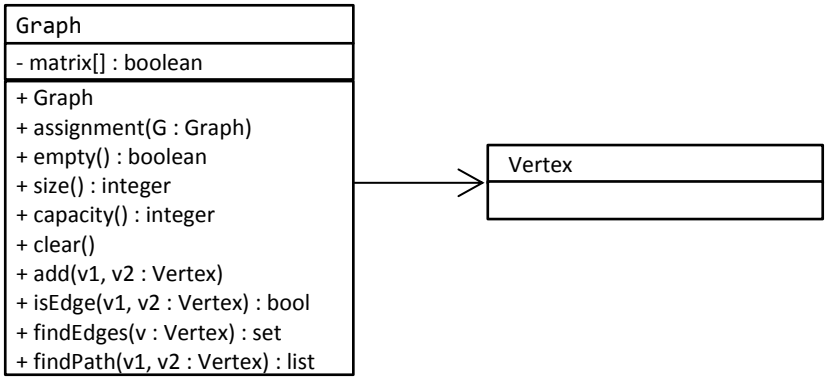


Notice how there are three elements in the set associated with vertex `A`. This means that the out-degree of `A` is three. The edges are (`A`, `B`), (`A`, `C`), and (`A`, `E`).

Notice how there is no set associated with vertex `E`. This is because `E` is a sink vertex and therefore has an out-degree of zero. We can either represent this with a `NULL` pointer or an empty set.

Grid

The final implementation is called a grid. Recall from the array of set method that each slot in the array is the "source" component of the edge. The grid extends this by representing the graph as a two-dimensional array of `bool`s.



The horizontal dimension corresponds to the "source" component of the edge in much the same way as the slots in the array of lists implementation. In the grid, the vertical dimension corresponds to the "destination" component. If a given slot in the grid has a "`true`" value, then there exists an edge from the vertex corresponding to the horizontal index of the slot and the vertex corresponding to the vertical index of the slot. Note that the source and destination are flipped from the adjacency-matrix.

Source

Destination	A	B	C	D	E	F
A		T		T		T
B	T					
C	T	T				
D			T			
E	T			T		
F				T		

From this grid, it is possible to see all the vertices reachable directly from `A` by scanning down the A column. Here vertex `B`, `C`, and `E` are directly reachable. Thus (`A`, `B`), (`A`, `C`), and (`A`, `E`) are edges.

It is also possible to see which edges lead into a given vertex. For example, say one wished to know the list of vertices with an edge leading into `A`. By scanning across the `A` row, one can see that `B`, `D`, and `F` all have edges leading directly into `A`. Therefore (`B`, `A`), (`D`, `A`), and (`F`, `A`) are edges.

Notice how there are no '`T`'s in the `E` column. This means that there are no edges leading out of `E` so `E` must be a sink. A quick scan of the matrix reveals that `E` is the only sink. Notice how every row has at least one '`T`'. This means that there are no sources in this graph. Any vertex with no '`T`'s in the row must be a source.

The three most common implementations of a graph are a grid (when the number of edges is high), a set of edges (when the number of edges is low), and an array of sets (when the number of edges is proportional to the number of verticies). There are advantages to each, depending on the properties of the graph and how it is utilized. For this reason, implementation details of all three will be presented below.

Create

To create a set of edges, it is only necessary to instantiate an empty set which is accomplished by set's default constructor. This is O(1) in time and space.

```
Graph.Graph(n)
  s
```

To create an array of sets object, it is necessary to allocate an array of nodes where each element contains an empty set.

```
Graph.Graph(n)
  array ← NEW set[n]
```

Observe how this takes O(n) space. Each edge added to the array of sets will increase the amount of utilized space, bringing our total space allocation to O(n + m) where n is the number of vertices and m is the number of edges.

To create a grid object, it is necessary to allocate the grid of `bool`s and assign each to `false`. In many languages, newly allocated arrays or blocks of allocated memory are initialized to `false` automatically. Just because it happens automatically does not mean it is without cost! C++ requires the programmer to explicitly initialize memory. In both cases, the cost of clearing the memory is $O(n^2)$:

```
Graph.Graph(n)
  matrix ← NEW Bool[n x n]
  FOR i ← 0 .. n²
    grid[i] ← FALSE
```

Thus both the space requirements and algorithmic complexity of the grid is $O(n^2)$.

Add

Adding an edge to a set of edges involves inserting the edge into the set. This is O(log n) of the number of edges assuming our set is implemented as a BST:

```
Graph. add(v1, v2)
  s.insert(Edge(v1,v2))
```

Adding an edge to an array of sets graph involves indexing into the appropriate slot in the array and then adding a vertex to the corresponding set.

```
Graph. add(v1, v2)
  grid[v1].insert(v2)
```

Notice how finding the appropriate slot in the array happens in constant time (O(1)). Adding a node to the set depends on the implementation of set. If the set is implemented with a sorted array, then O(m) is the cost where `m` is the out-degree of the vertex. If the set is implemented with a BST, then the cost is O(log m). In both cases, the space requirements are increased by the size of one vertex.

Adding an edge to a grid occurs in a completely different way. It involves indexing into the appropriate slot in the matrix and setting that slot to `true`.

```
Graph.add(v1, v2)
  grid[v1, v2] ← TRUE
```

Indexing into an array occurs in constant time (O(1)), as does setting a slot to `true`. Thus the overall performance is O(1) and there is no change in space requirements.

Is Edge

To determine whether an edge exists in a set of edges, it is necessary to construct the required edge and see if it exists. This is O(log n) where n is the number of edges in the graph:

```
Graph.isEdge(v1, v2)
  RETURN s.find(Edge(v1,v2)) ≠ NULL
```

To determine whether an edge exists between two vertices in an array of sets, it is necessary to index into the appropriate slot in the array and query the set for the existence of the destination node.

```
Graph.isEdge(v1, v2)
  RETURN array[v1].find(v2) ≠ NULL
```

Indexing into the appropriate slot in the array can be accomplished in constant time. However, `set.find()` is accomplished in O(log m) where m is the out-degree of the vertex. Thus the overall cost of the function is O(log m).

To determine the existence of an edge with the grid implementation, it is only necessary to index into the appropriate slot in the memory and return the Boolean value.

```
Graph.isEdge(v1, v2)
  RETURN grid[v1, v2]
```

Because array indexing occurs in constant time, the overall cost of this method is O(1).

Find Edges

It is common to find all the outbound edges from a given vertex. This is very expensive with the set implementation. We can either search for every possible edges leading from a given vertex, or iterate through all the edges. The first implementation is O(n log m) where `n` is the number of verticies and `m` is the number of deges. The second is O(m). Which do you choose? Depending on the size of `n` and `m`.

With the array of sets implementation, this requires simply returning the corresponding set corresponding to a given edge.

```
Graph.findEdges(v1)
  RETURN array[v1]
```

The cost of this algorithm depends completely on whether the array of set needs to be copied. If it does, the cost is O(m) where m is the out-degree. Otherwise, it is O(1).

The grid version of the function requires more work. This involves traversing one row of the grid and adding a node if the slot in the grid contains the `true` value.

```
Graph.findEdges(v1)
  FOR v2 ← 0 … size()
    IF grid[v1, v2] = TRUE
      s.insert(v2)
  RETURN s
```

It may appear that the cost of this function is O(n) where n is the number of vertices in the graph. However, inserting an element into a set does not happen in constant time. Using the array implementation of the set, insert takes O(m) where m is the out-degree. This makes the overall cost of the function $O(n + m^2)$ because each slot in a row needs to be visited (O(n)) and the cost of adding m elements to the set is $O(n^2)$. Using the BST implementation of the set, insertion is O(log m) and the overall cost is O(n + m log m).

Find Path

The process of finding a path between two vertices is essentially a search problem. There are two main approaches: a depth-first search and a breadth-first search. The depth-first approach involves following the "first" edge from the source vertex until a dead-end is discovered or the destination vertex is found. From that point, the algorithm backtracks to the previous decision point (the edge with an out-degree of greater than 1) and the process continues. Only after all the possibilities are exhausted is the shortest path returned. While this algorithm works, it has two shortcomings. The first is that there is no way to determine if a cycle is found. In other words, if A has an edge to B and B has an edge to A, then this depth-first algorithm could lead to an infinite loop. The second problem is that many paths may pass through a given node. Each successive iteration does not leverage the work done by previous iterations, yielding a large amount of redundant searching.

The breadth-first search does not suffer from these problems. Imagine starting with vertex A. The length of the path from A to A is zero because we are already there. We will assign the value 0 to vertex A because the shortest path to A has zero edges. Next, imagine all the vertices that A points to. Each of these vertices will be assigned to the value of 1 unless a vertex already has a value assigned less than 1. This is because it only takes one edge to get to these vertices. Now, from all the vertices with the value 1, we produce a set of all the vertices they point to. Again, some may already have a value associated with them. We skip them because the shortest path to those vertices is not through the path we are identifying at this point. The rest of the vertices will inherit the value 2. This continues until the destination vertex is found. Here, the length of the shortest path to this vertex is the number assigned to the vertex. Note that if we reach a point where all of the vertices reachable from a given step of the algorithm already have a value, then we have already "solved" the shortest path for them. This means the algorithm is done and there is no path from the source vertex to the destination vertex. We call this breadth-first approach the "shortest path algorithm."

```
Graph.shortestPath( source, destination )
  distance ← 0

  toVisit.push_back(source)
  FOR i ← 0 … size()
    distances[i] ← -1

  WHILE !toVisit.empty() and distances[destination] = -1

    v ← toVisit.front()
    toVisit.pop_front()

    IF distances[v] > distance
      distance++

    s ← findEdges(v)
    FOR it ← s.begin() … s.end()
       IF distances[it] = -1
         distances[it] ← distance + 1
         predecessor[it] ← v
         toVisit.push_back(it)
       distance++

  IF distances[destination] = -1
     RETURN ERROR
  path.push_back(destination)
  FOR i ← 1 … distance
     path.push_back(predecessor[path[i – 1]]
  RETURN path
```

Application

Two examples: finding the path between two cities and finding the prerequisites for a given course.

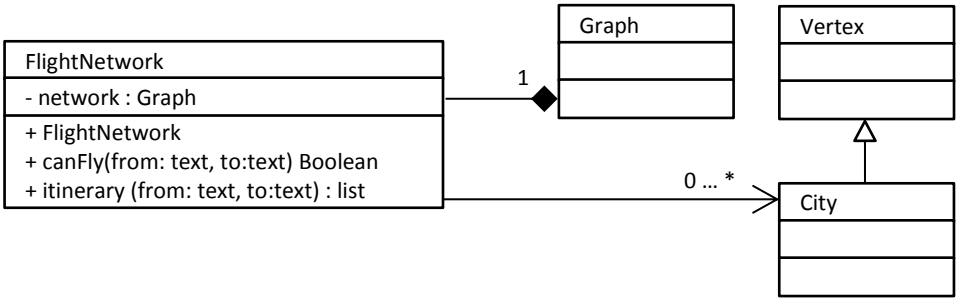
Flights

Owning a small airline, I wish to organize our flight network into a graph. This will not only help our expansion efforts, but it will simplify our registration process.

The first thing that is needed is to define the notion of a city as a vertex in a graph. To accomplish this, we will need a `City` class that that is derived off of the `Vertex` class.

```
const char * CITIES[] =
{
     "New York",
     "Boston",
... code removed for brevity ...
};

class City : public Vertex
{
... code removed for brevity ...
};
```

Now we will build a new class that captures our flight information. Our new `FlightNetwork` class will contain exactly one `Graph` (through "composition"). It will also use (through "association") a `City` as a local variable and as a parameter to a function. This `City` will be derived (through "inheritance") a `Vertex`. The UML class diagram of these classes is presented below.



Next, we will build a graph containing all of the flights our fledgling airline offers. In this case, a flight is an edge in the graph. All this is done in the constructor.

```
FlightNetwork :: FlightNetwork()
{
   network.add(City("SLC"), City("Denver"));
   …
}
```

Our two remaining interfaces are trivial. One will inform the user if a flight exists between two cities:

```
bool FlightNetwork :: canFly(const char * from, const char * to)
{
   // convert the string cities into verticies and make the inquiry
   return network.isEdge(City(from), City(to));
}
```

Notice how this function converts the client's city names (which are c-strings) into vertices that are needed by the `Graph` class.

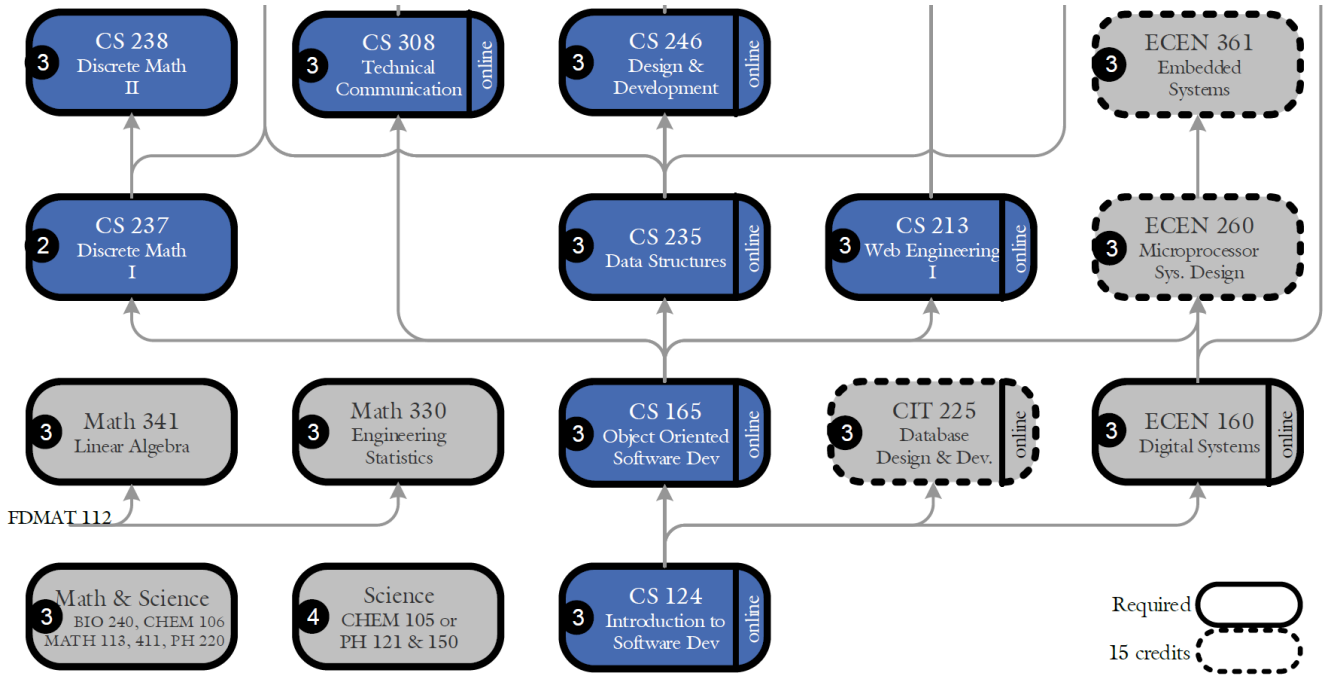
The second method will produce a list of legs that will constitute the itinerary between two cities.

```
list <string> FlightNetwork :: itinerary(const char * from, const char * to)
{
   list <string> route;

   // find the path. This is in verticies, not the format the user expects
   list <Vertex> path = flights.findPath(City(from), City(to));

   // return an empty path if there are less than 2 cities
   if (path.size() < 2)
      return route;

   // iterate through the list of verticies
   list <Vertex> :: iterator itFrom = path.begin();
   list <Vertex> :: iterator itTo   = itFrom + 1;
   while (itTo != path.end())
   {
      // create a string representing one leg
      string leg;
      leg = itFrom->getText();
      leg += " to ";
      leg += itTo->getText();

      // add the leg into the route we are building
      route.push_back(leg);

      // advance the iterators
      itFrom = itTo;
      itTo++;
   }
   return route;
}
```

As with the `FlightNetwork :: canFly()` function, most of the work done here is to translate the client's request into a format the `Graph` can understand. The rest is to translate the `Graph`'s output into the format the client requires. All the work is done in the `Graph :: findPath()` method.

Course Dependency

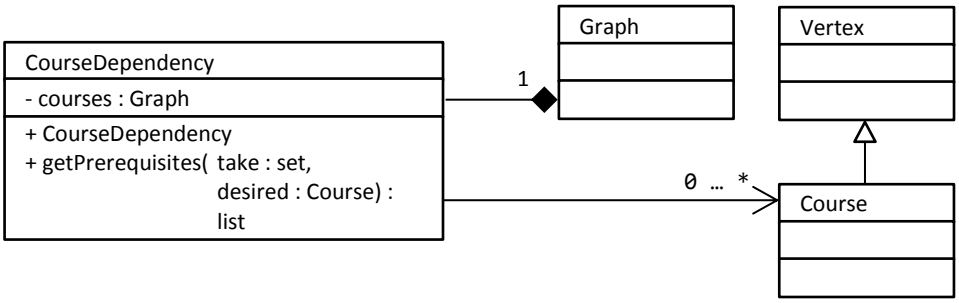
In the next example, we would like to capture the courses in a given Computer Science (CS) program. These courses are connected by prerequisites. For example, one can only take CS 165 after one as taken CS 124. Some of the courses in a given CS program are the following:



The first step is to create a class representing a course. Of course, this class will derive off of the `Vertex` class. The trick is that we want to this class to be generic, one that would work with any group of courses. To do this, we will add a `newCourse()` method which will add a course to the collection if it is not currently specified. For all instances to share the same course collection, we will need to make it static.

```
class Course : public Vertex
{
public:
   int newCourse(const char * course);
private:
   static vector <string> courseList;
};
```

Our challenge is to create a class that not only stores the prerequisites, but also to informs what courses a student would need to take in order to enroll in a given course. This can be represented with the following UML class diagram:



The constructor will populate the graph with all of the prerequisites. Because we want to use this same class for many departments, we will want to read the prerequisites from a file. The file format we will use is:

```
CS165 CS124 |
CS213 CS165 |
CS235 CS165 |
CS237 CS165 |
CS238 CS237 |
CS246 CS235 |
CS306 CS235 CS237 |
CS308 CS165 |
CS312 CS235 |
CS313 CS213 CS246 |
CS345 ECEN324 |
CS364 CS308 CS246 |
CS398 CS246 |
CS416 CS246 CS308
CS432 CS246 CS308
CS450 CS246 |
CS460 CS246 CS235 |
CS470 CS308
CS480 CS306 |
CS499 CS364 |
```

Each line in this file represents the prerequisites for a given course. The first item on a line is the course for which we are determining the prerequisites, and every subsequent item represents the prerequisites. Thus CS124 is a prerequisite for CS165. Both CS235 and CS237 are prerequisites for CS306.

From here, we define a non-default constructor. Note that the filename of the prerequisites file is a required parameter.

```
CourseDependency :: CourseDependency(const char * filename)
{
   ifstream fin(filename);
   if (fin.fail())
      return;

   // read each line of the file. The first is a vertex, the class from which
   // we are to derive the prerequisites. If the read text is not a recognized
   // classname, then the extraction operator will set the fail bit.
   while (fin >> vFrom)  // read the first vertex, the class
   {
      // now read the list of prerequisites. We will continue until the read text
      // is not recognized as a valid class name. The "|" will trigger this fail.
      while (fin >> vTo)
         courses.add(vFrom, vTo);

      // clear the error state which came from the "|"
      fin.clear();

      // skip over the invalid "|"
      fin.ignore();
   }
   fin.close();
}
```

The final method we need to implement is the function enumerating the minimal number of courses a student needs to take to be able to take a given course. In other words, it will find the minimal prerequisite chain. The input parameters will be the set of courses the student has already completed and the desired course. The output will be the list of courses to take.

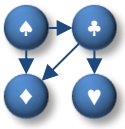
```
list <Course> CourseDependency :: getPrerequisites(const set <Course> & taken,
                                                   const Course & desired)
{
   list <Course> shortest;                     // the shortest known prerequisite list

   // loop through the set of courses taken to see
   // which will afford the shortest path to the desired course
   set  <Course> :: iterator it;
   for (it = taken.begin(); it != taken.end(); ++it)
   {
      list <Course> current;                   // the current prerequisite

      // what is the shortest path from this previously-taken class
      //   to the desired course?
      current = dependencies.findPath(*it, desired);
      if (current.size() != 0 &&               // a path was found
          (shortest.size() == 0 ||             // no previous path was found
           current.size() < shortest.size()))
         shortest = current;                   // this path is the current shortest
   }

   return shortest;
}
```

Notice how the `Graph :: findPath()` method does all of the heavy lifting here.

Examples

Example 13.1

Consider the following graph:



Please answer the following questions:

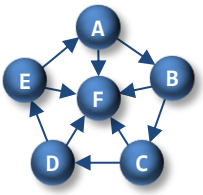
- What is the in-degree of ♥?
  There is only edge into ♥ and that is ♣→♥. Thus the in-degree is 1

- What is the out-degree of ♣?
  There are two edges out of ♣: ♣→♦ and ♣→♥. Thus the out-degree is 2.

- Name all the source and sink verticies.
  Only ♠ has an in-degree of zero so it is the only source.
  Both ♦ and ♥ have and out-degree of zero so they are the sinks.

Example 13.2

Consider the following graph:



Find all the paths between the following verticies:

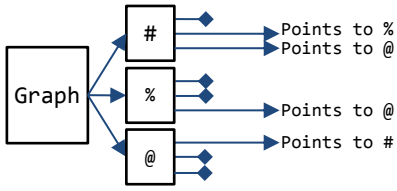
- A→C:
  A can only go to F and B. No nodes lead out of F so it is a sink. This leaves B which can only go to F and C. Again, F is a sink so it is a dead-end. Thus the only path from A to C is: {A, B, C}. However, from C we can also go to D and then E and then back to A. Thus another path is: {A, B, C, D, E, A, B, C}. In fact, we can go around the loop as often as we like to yield an infinite number of paths.

- F→A:
  No nodes lead out of F so there are no paths.

- D→F:
  D leads directly to F so one path is {D, F}. D also leads to E which leads to F. This is a second path {D, E, F}. E also leads to A which leads to F. This gives us {D, E, A, F}. A also leads to B which leads to F. A fourth path is thus {D, E, A, B, F}. B also leads to C which leads to F. A fifth path is {D, E, A, B, C, F}. Note that C leads to D. This means we can add the sequence {D, E, A, B, C} to all the previous paths. In fact, there are an infinite number of paths because we can go around the loop as often as we like.

Example 13.3

Draw the graph represented by the following interconnected nodes:

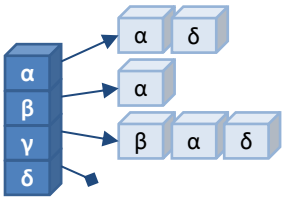


Answer:

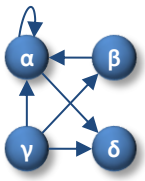
We should double-check our answers. The out-degree of # is two. It should point to both % and @. The out-degree of % is one and it should only point to @. Finally, the out-degree of @ is one and it should only point to #.

Example 13.4

Draw the graph represented by the following adjacency-list:



Answer:



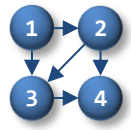
In the adjacency-list, the array of nodes on the left represent the "front" and the list on the right represent the "to." Thus, we can tell by inspection that δ is a sink because it does not point to anything. Looking at the graph reveals this to be true. We can also tell that γ is a source because it does not appear anywhere on the list on the right. Because noone points to it, it cannot be a destination from any other node. An inspection of the graph reveals this to be true.

Example 13.5

Draw the graph represented by the following set of edges:

$$g = \{ (1,2), (1,3), (2,3), (2,4), (3,4) \}$$

Answer:



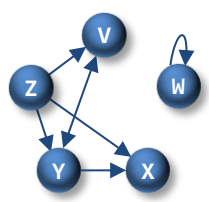
We should double-check our answers. There are two edges leading out of 1: the edge (1, 2) and (1, 3). Notice that there are no edges leading into 1 (the second item in the pair of coordinates corresponding to an edge). Thus node 1 is a source. Notice there are two edges leading into 4: the edge (2, 4) and (3, 4). There are no edges leading out of 4 (the first item in the pair of coordinates corresponding to an edge). Thus node 4 is a sink. Finally, notice how there are 5 elements in the set and there are five edges on the graph.

Example 13.6

Draw the graph represented by the following adjacency matrix:

$$adj = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \end{bmatrix} \qquad data = \begin{bmatrix} V \\ W \\ X \\ Y \\ Z \end{bmatrix}$$

Answer:



Recall that the source verticies are along the side and the destination verticies are across the top. Thus the first row represents all the verticies that V points to (and V only points to Y). Also, the first column represents who all points to V (and both Z and Y pont to V).

We should double-check our answers. From looking at the adjacency matrix, we see that the third row has nothing but 0's in it. This indicates that the third vertex (X) does not point to any other verticies. In other words, X is a sink. When we look at the graph, we can see that the out-degree of X is in fact zero.

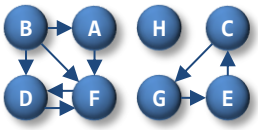
From looking at the adjacency matrix, we can see that the last column contains nothing but 0's. This means that no verticies point into Z. Looking at the graph, we can see that the in-degree of Z is zero.

Notice the second row of the graph, representing the out-degree of vertex W. Apparently the only vertex that W points to is W. Notice the second column of the graph, representing the in-degree of vertex W. Apparently the only vertex that points to W is the vertex W. By looking at the graph, we can see that W is isolated, the only vertex pointing to it is itself. So sad!

Exercises

Exercise 13.1

Consider the following graph:



Enumerate all the paths from A to C.

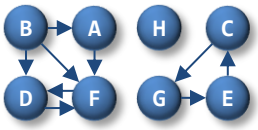
Answer:

_____

Please see page 356 for a hint.

Exercise 13.2

Consider the following graph:



Enumerate all the paths from B to D.

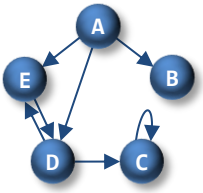
Answer:

_____

Please see page 356 for a hint.

Exercise 13.3

Consider the following graph:



Enumerate all the paths from A to C.

Answer:

_____

Please see page 35646 for a hint.

Exercise 13.4

Consider the graph represented by the following adjacency matrix:

$$adj = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad data = \begin{bmatrix} A \\ B \\ C \\ D \\ E \end{bmatrix}$$

Enumerate all the nodes that A points to.

Answer:

_____

Please see page 358 for a hint.

Exercise 13.5

Consider the graph represented by the following adjacency matrix:

$$adj = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad data = \begin{bmatrix} A \\ B \\ C \\ D \\ E \end{bmatrix}$$

Enumerate all the nodes that point to D.

Answer:

_____

Please see page 358 for a hint.

Exercise 13.6

Consider the graph represented by the following adjacency matrix:

$$adj = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \qquad data = \begin{bmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \end{bmatrix}$$

Enumerate all the nodes that points to B.

Answer:

_____

Please see page 358 for a hint.

Exercise 13.7

Consider the graph represented by the following adjacency matrix:

$$adj = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \qquad data = \begin{bmatrix} A \\ B \\ C \\ D \\ E \\ F \\ G \end{bmatrix}$$

Enumerate all the nodes that point to F.

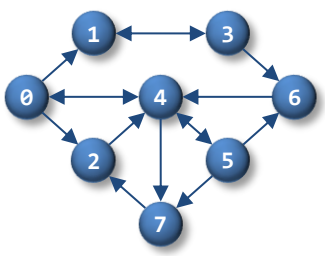
Answer:

_____

Please see page 358 for a hint.

Exercise 13.8

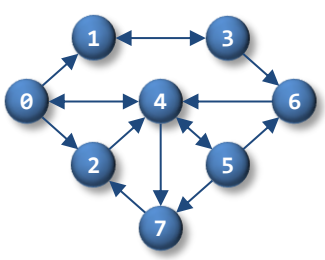
Represent the following graph as a set of edges:



Please see page 345 for a hint.

Exercise 13.9

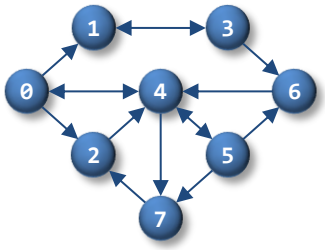
Represent the following graph as an array of sets:



*Please see page **Error! Bookmark not defined.** for a hint.*

Exercise 13.10

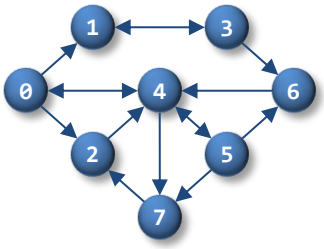
Represent the following graph as a grid:



*Please see page **Error! Bookmark not defined.** for a hint.*

Exercise 13.11

With the following graph, find the shortest path from vertex 0 to vertex 6:



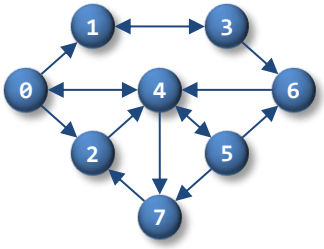
	0	1	2	3	4	5	6	7
vertex								
distance								
predecessor								

toVisit								

Please see page 350 for a hint.

Exercise 13.12

With the following graph, find the shortest path from vertex 7 to vertex 3:



	0	1	2	3	4	5	6	7
vertex								
distance								
predecessor								

toVisit								

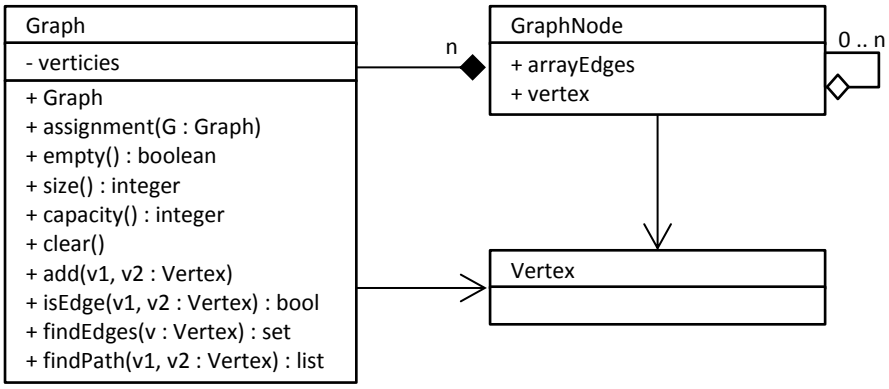
Please see page 350 for a hint.

Problems

Problem 13.1

Create a class to capture the notion of a graph. This will consist of a collection of vertices and edges. The graph does not need to store anything in a vertex, or in an edge, but just keep track of whether there is an edge connecting one vertex to another.

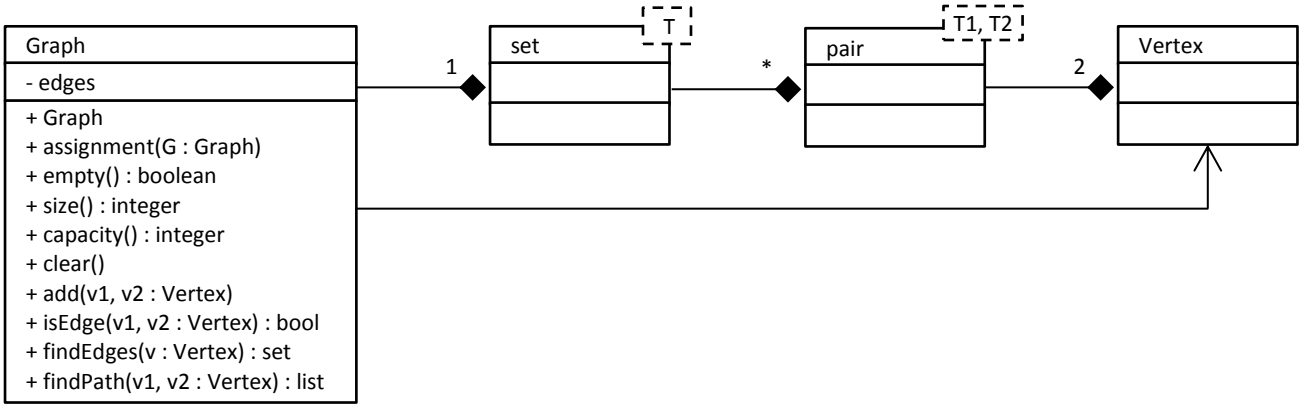
To implement this, use the interconnected node design:



Please see page 344 for a hint.

Problem 13.2

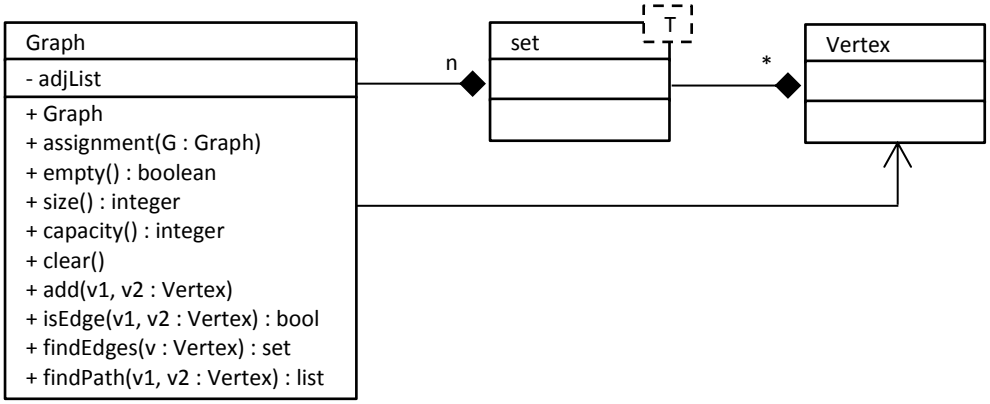
Same problem as 13.1, except use the set of edges design:



Please see page 345 for a hint.

Problem 13.3

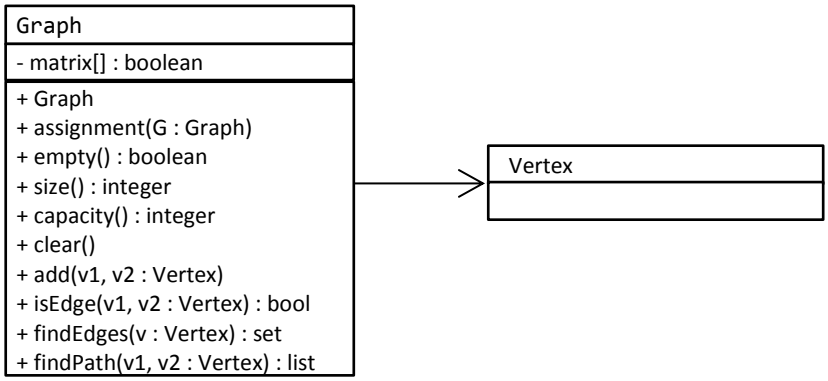
Same problem as 13.1 and 13.2, except use the array of sets design:



Please see page ***Error! Bookmark not defined.*** *for a hint.*

Problem 13.4

Same problem as 13.1, 13.2, and 13.3, except use the grid design:



Please see page ***Error! Bookmark not defined.*** *for a hint.*

Problem 13.5

Using the `Graph` class from Problem 13.1, 13.2, 13.3, or 13.4, implement a maze. This will read the edges from a file and represent them on the screen using ASCII-art. For example, consider the following file:

```
5 5
a1 a2  a2 b2   b2 b1
a2 a3  a3 b3   b3 c3   c3 c2   c2 c1   c1 d1   d1 e1   e1 e2   e2 e3   e3 d3   d3 d2
a3 a4  a4 b4   b4 b5   b5 c5   c5 c4   c4 d4   d4 e4   e4 e5   e5 d5
a4 a5
```

This file has a 5x5 maze (the.first two numbers in the file). There is an edge between a1 and a2. If this file is opened, the following would be the output:

```
What is the filename? maze5x5.txt
+  +--+--+--+--+
|  |  |       |
+  +  +  +--+  +
|     |  |  |  |
+  +--+  +  +  +
|        |     |
+  +--+--+--+--+
|     |       |
+  +  +  +--+  +
|  |     |     |
+--+--+--+--+  +
```

Problem 13.6

Using the maze program from Problem 13.5, write an algorithm to solve the maze. This is accomplished by using the `Graph::findPath()`. You will have to modify the maze display code to illustrate the path through the maze:

```
+  +--+--+--+--+
|##|  |       |
+  +  +  +--+  +
|##   |  |  |  |
+  +--+  +  +  +
|##      |     |
+  +--+--+--+--+
|## ##|## ## ##|
+  +  +  +--+  +
|  |## ##|   ##|
+--+--+--+--+  +
```

Please see page 350 for a hint.

APPENDIX A : ADT PROPERTIES

The following is a list of the properties of the ADTs.

ADT	Description	Access	Order	Expandable	Insertion
Vector	Works like an array	Random	Client specified	Yes	Client
Stack	First-in-Last-out	One end	By order	Yes	End
Queue	First-in-First-out	End	By order	Yes	End
Deque	Access either end	Either end	By order	Yes	Ends
Set	Collection of unique elements	By search only	Implementation	Yes	Blind
List	Ordered collection	Sequential	Client specified	Yes	Client
Map	Associative array	Index by key	Implementation	Yes	Blind
Hash	Collection of elements	Index by hash value	Implementation	Yes	Blind
Graph	Vertices and Edges	Existence of edge only	Implementation	No	None

The following is a list of the properties of the sorts.

Sort	Description	Best	Worst	Average	Space
Bubble	Often the slowest of the common sorting algorithms	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection	A very large number of comparisons are required	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion	A very large number of assignments are required	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Binary	Allocating the nodes is expensive	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(n)$
Merge	Does not require the entire array to be in memory	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick	Generally the fastest of the common sorting algorithms	$O(n)$	$O(n^2)$	$O(n \log n)$	$O(1)$
Heap	Guaranteed to always be fast	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

APPENDIX B : ADT PERFORMANCE

Insertion

Adding an element to the data structure.

ADT	Best	Typical	Worst	Discussion
Vector	O(1)	O(1)	O(n)	Reallocation in the worst case
Stack	O(1)	O(1)	O(n)	Reallocation in the worst case
Queue	O(1)	O(1)	O(n)	Reallocation in the worst case
Deque (array)	O(1)	O(1)	O(n)	Reallocation in the worst case
Deque (linked list)	O(1)	O(1)	O(1)	Adding a node always takes the same time
Set (array)	O(1)	O(n)	O(n)	Shifting element is the biggest expense
Set (BST)	O(1)	O(log n)	O(log n)	Finding the appropriate spot takes longest
List	O(1)	O(1)	O(1)	Adding a node always takes the same time
Map	O(1)	O(log n)	O(log n)	Finding the appropriate spot takes longest
Hash	O(1)	O(1)	O(n)	Poor hash function is the worst case
Graph (adj-list)	O(1)	O(1)	O(log n)	High out degree is the worst case
Graph (adj-matrix)	O(1)	O(1)	O(1)	Adding an edge is always fast

Removal

Remove an element from the data structure.

ADT	Best	Typical	Worst	Discussion
Vector	O(1)	O(1)	O(1)	Removes lots remain empty
Stack	O(1)	O(1)	O(1)	Abandoning a slot requires no time
Queue	O(1)	O(1)	O(1)	Abandoning a slot requires no time
Deque (array)	O(1)	O(1)	O(1)	Abandoning a slot requires no time
Deque (linked list)	O(1)	O(1)	O(1)	Deleting a node always takes same time
Set (array)	O(1)	O(n)	O(n)	Shifting element is the biggest expense
Set (BST)	O(log n)	O(log n)	O(log n)	Finding the appropriate spot takes longest
List	O(1)	O(1)	O(1)	Deleting a node always takes the same time
Map	O(log n)	O(log n)	O(log n)	Finding the appropriate spot takes longest
Hash	O(1)	O(1)	O(n)	Poor hash function is the worst case
Graph (adj-list)	O(1)	O(1)	O(log n)	High out degree is the worst case
Graph (adj-matrix)	O(1)	O(1)	O(1)	Clearing an edge is always fast

Retrieval

Extract one element from the data structure.

ADT	Best	Typical	Worst	Discussion
Vector	O(1)	O(1)	O(1)	All elements are equally easy to retrieve
Stack	O(1)	O(1)	O(1)	Getting the top element is always the same
Queue	O(1)	O(1)	O(1)	Getting the head element is always the same
Deque (array)	O(1)	O(1)	O(1)	Getting head or tail is the same
Deque (linked list)	O(1)	O(1)	O(1)	Getting head or tail is the same
Set (array)	O(1)	O(log n)	O(log n)	Finding the element takes the longest
Set (BST)	O(1)	O(log n)	O(log n)	Finding the element takes the longest
List	O(1)	O(n)	O(n)	Iterating through the list takes the longest
Map	O(1)	O(log n)	O(log n)	Finding the element takes the longest
Hash	O(1)	O(1)	O(n)	Poor hash function is the worst case
Graph (adj-list)	O(1)	O(1)	O(log n)	High out degree is the worst case
Graph (adj-matrix)	O(1)	O(1)	O(1)	All edges are equally easy to query

APPENDIX C : PSEUDOCODE

Abstract data types can be implemented in just about any programming language. It is therefore useful to describe their algorithms in pseudocode.

Pseudocode keywords

There are seven classes of keywords: receive, send, math, remember, compare, repeat, and call functions.

Receive

A computer can receive information from a variety of input sources, such as keyboard, mouse, a network, a sensor, or wherever.

Keyword	Description	Example
READ	Receive information from a file	READ studentGrade
GET	Receive input from a user, typically from the keyboard	GET income

Send

A computer can send information to the console, display it graphically, write to a file, or operate on a device.

Keyword	Description	Example
PRINT	When sending data to a permanent output device like a printer	PRINT full student name
WRITE	When writing data to a file	WRITE record
PUT	When sending data to the screen	PUT instructions
PROMPT	Just like PUT except always preceding a GET instruction	PROMPT for user name

Arithmetic

Most processors have the built-in capability to perform the following operations:

Keyword	Description	Example
()	Parentheses are used to override the default order of operations	c ← (f – 32) * 5/9
*** x** **/ ÷ mod div**	Any mathematical convention can be used	numWeeks ← days div 7
+ -	Addition / subtraction	SET count ← count + 1
|| √ π	Common mathematical values or operations	PUT √10
≤ ≠	Common comparison operations	IF grade ≥ 60

Remember

A computer can assign a value to a variable or a memory location.

Keyword	Description	Example
SET **=** **←**	Assign a value to a variable	SET answer ← 42

Compare

A computer can compare two values and select one of two alternate actions

Keyword	Description	Example
IF **ELSE**	For two possible outcomes	IF income / 10 ≤ tithingPaid PUT full tithe message ELSE PUT scripture from Malachi
SWITCH CASE	For multiple possible outcomes	SWITCH option CASE 1 PUT great choice! CASE 2 PUT could be better CASE 3 PUT please reconsider

Repeat

A computer can repeat a group of actions.

Keyword	Description	Example
WHILE	When repeating through the same code more than once	WHILE studentGrade < 60 takeClass()
FOR	When counting	FOR count ← 1 to 10 by 2s PUT count

Functions

A computer can call a function and pass parameters between functions.

Usage	Conventions	Example
Declaring	Functions are named. Input parameters are enumerated. Statements are indented. RETURN values. END.	computeTithing(income) SET tithing ← income / 10 RETURN tithing END
Calling	Call by name Specify parameters	PUT computeTithing(income)

APPENDIX D : UML CLASS DIAGRAM

Abstract data types are commonly built upon classes if such constructs are available in your high-level language of choice. Since the structure of these classes and their relationship with other classes can be non-trivial, it is often useful to describe them with an object-oriented modelling language such as UML class diagrams.

Single Class Description

A single class can be described in a UML class diagram with a one column three row table:

ClassName
+ publicVariable : integer # protectedVariable : integer - privateVariable : integer
+ publicMethod() : void # protectedMethod() : void - privateMethod() : void

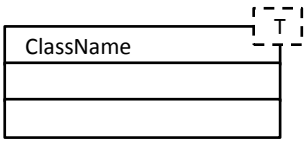
The top row refers to the class or structure name. This will be the type of the object if an instance of the class is instantiated. The second row refers to the member variables of the class or structure. In this case, there are three member variables. The third row refers to the member functions or methods of the class. If this UML class diagram described a structure lacking methods, then only two rows would be present.

Notice the `+`, `#`, and `-` symbols before the variables and methods. These are access modifiers. The minus sign (`-`) indicates the variable or method is private, indicating it is only accessible from a method within the class itself. The hash mark (`#`) indicates the variable or method is protected. This means it is accessible only to methods within the class or from methods within derived classes. The plus (`+`) indicates the variable or method is public, namely the client can directly access it from the object or from any associated method.

The above class would be defined in C++ as the following:

```
class ClassName
{
public:
   void publicMethod();
   int publicVariable;

protected:
   void protectedMethod();
   int protectedVariable;

private:
   void privateMethod();
   int privateVariable;
};
```

It is also possible to specify that a class is a template. This is accomplished by placing the template type in a box at the upper-right corner of the class diagram. Here the methods and member variables are omitted for the sake of brevity.



Class Relations

UML class diagrams also allow for the description of how classes relate to each other. These are accomplished with arrows connecting single class descriptions. There are six flavors of relations available in class diagrams:

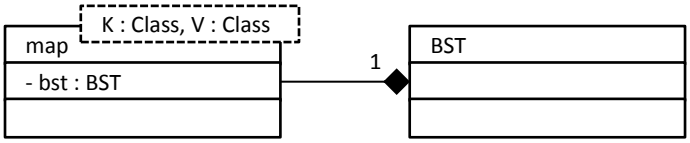
Notation	Name	Description
——◆	Composition	One class is a member variable of another
——◇	Aggregation	One class is a member variable reference of another
——▷	Inheritance	One class derives off of a base class
——⊕	Nested-class	One class is a nested-class of another
——>	Association	One class is a local variable in a method of another
- - - - ->	Dependency	One class is otherwise related to another

Composition

Composition is a type of "has-a" relation where an object from one class is part of another. When an object of the enclosing class is created, an object of the composition class is created. When the enclosing class object is destroyed, the corresponding object in the composition class is also destroyed. An example of this relation is the BST class being part of the Map class:

```
class map               // enclosing class
{
... code removed for brevity ...
private:
   BST bst;             // composition class
... code removed for brevity ...
};
```

Notice how the `bst` object is created when the `map` is instantiated. Also, when `map` falls out of scope, then the `bst` object will be destroyed. This can be represented with the following UML class diagram:



Notice the number next to the composition line. This number specifies the number of instances of the `BST` class that are contained in the `map` class. In this case, only one. You could instead specify a range (1..5), zero or one (0/1), or an unbounded number of instances (*). As a general rule, any class that is a member variable of another class is probably composition. We usually specify the number of instances next to the arrow (1 in the above example).

Examples of Composition include:

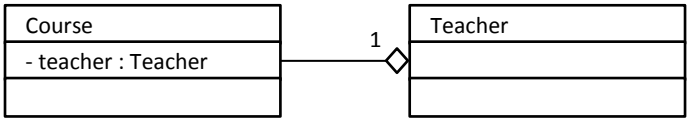
- The Deque example Chapter 06. Here one `deque` class is composed of a collection of `Nodes`.
- The Asteroids example in Chapter 07. Here one `Game` class contains one `Ship` and a collection of entities.
- The Large Numbers example also in Chapter 07. Here a `WholeNumber` is composed of a single `list`, each element of which consists of a three digit number.
- The implementation of a BST in Chapter 09. Here one `BST` is composed of many `BNodes`.
- The implementation of a Map in Chapter 10. Here a `map` is composed of one `BST`.

Aggregation

Aggregation is similar to composition with one important difference: destruction of the enclosing class does not necessarily result in destruction of the aggregation class. For example, consider a class representing a section in a college course. This class will contain a reference to the teacher. Notice that the teacher object can be part of many courses. Also, notice that the course does not destroy the teacher when the course is completed. That would be most unfortunate.

```
class Course            // enclosing class
{
... code removed for brevity ...
private:
  Teacher * teacher;    // aggregation class
... code removed for brevity ...
};
```

An example of a UML class diagram is the following:



As with composition, aggregation also may include a number next to the arrow to indicate the number of instances of the class are referenced by the other. In this case, there is one and only one reference to a `Teacher` from the `Course` class.

When a class is a pointer member variable to another class, the aggregation relationship is a possibility. For example, if the `Course` class contains a reference to `Teacher` but does not create or destroy the `Teacher` object, then the relationship is aggregation. However, if the `Course` class were to allocate the `Teacher` object and then destroy it in the destructor, this would be composition.

Examples of Aggregation include:

- The implementation of a Node in Chapter 06. Here one `Node` can refer to zero, one, or two `Nodes`. Destroying one `Node` does not necessitate creation or destruction of the neighboring `Nodes`.
- The implementation of a List Iterator in Chapter 07. Here one `iterator` refers to a `Node` in the linked list. Destroying the `iterator` does not destroy the `Node`.
- The implementation of a Binary Node in Chapter 08. Similar to a linked list, a `BNode` can refer to zero, one, two, or three `BNodes`.
- The implementation of a BST Iterator in Chapter 09. Similar to the `iterator` for a `list`, the `iterator` to a `BST` refers to a `BNode` but does not manage it. Destroying the instance of the `BST iterator` does not result in the destruction of the `BNode`.

Inheritance

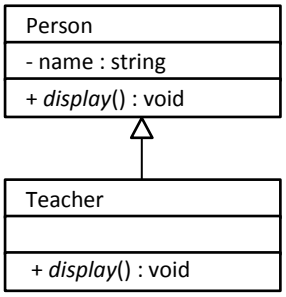
Inheritance relations are "is-a" where one class is a type or instance of another class. For example, the base class might be person.

```
class Person
{
public:
   virtual void display() const
   {
      std::cout << name;
   }
protected:
   std::string name;
};
```

A derived class may be a teacher.

```
class Teacher : public Person
{
public:
   virtual void display() const
   {
      std::cout << "Professor " << name;
   }
};
```

The UML class diagram representing this relation is the following:



Note how virtual functions are italic and pure virtual functions have the `=0` notation.

We use the same inheritance arrow for simple inheritance, polymorphism, and abstract classes. If inheritance is used, then there will be no virtual functions so none of the methods will be in italic. If polymorphism is used, there should be at least one italic or virtual function in the base-class. Finally, if the base class is an abstract class meaning there is at least one function that is a pure virtual function, then there must be the `=0` notation next to the pure virtual function.

There is one example of inheritance presented in this textbook:

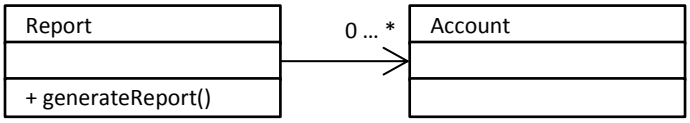
- In Chapter 13 (Graph), the `Vertex` class is a base class. From this, we can create many derived classes, such as a vertex of letters (`LVertex`), a vertex of cities (`City`), and a vertex of courses (`Course`)

Association

Association is when two classes are aware of each other but do not contain an instance of each other. This means one class cannot contain a member variable that is the second class. Instead, they usually have a local variable in a method that is then freed/destroyed when the method finishes execution. For example, consider an `Account` class that represents an individual's bank account. Next, consider a `Report` class that generates a report for the bank manager. The `Report` class does not contain an `Account` and an `Account` does not contain a `Report`. Therefore this is not an example of composition or aggregation. However, there is still a relation because the `Report` has to be aware of several `Accounts` to work properly.

```
class Report
{
... code removed for brevity ...
public:
   void generateReport()
   {
      list <Account> accounts;
      getDelinquentAccounts(accounts);
      displayResults(accounts);
   }
};
```

In this example, one `Report` is associated with multiple `Accounts`. Also, while `Report` is aware of `Account`, `Account` is not aware of `Report`. Finally, notice how the relationship between `Report` and `Account` is maintained only for the life of the `generateReport()` function. These are the defining characteristics of the Association relation.



As with composition and aggregation, you can specify the number of instances of one class that are associated with the other class. In this example, there can be anywhere between zero and a large number of `Accounts` associated with `Report`.

If one class is a local variable in a method of another class, then Association is probably the type of relationship.

There is one example of association presented in this textbook:

- In Chapter 13 (Graph), the `Graph` class does not contain a `Vertex`. However, it can accept a `Vertex` as a parameter. Because parameters are the same as local variables, this is an example of Association.

Nested-Class

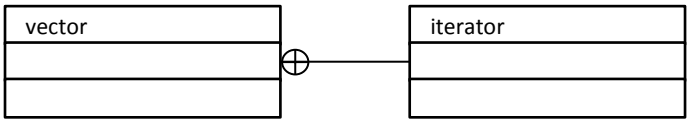
A nested-class is when one class is defined within another class. For example, an iterator is a nested-class of a container class.

```
{
   // first an object of the vector class
   vector <int> v;

   // next, we will create a vector iterator. Note that it is defined with the ::
   vector <int> :: iterator it;
}
```

An example of a class definition for vector and iterator is the following:

```
template <class T>
class vector
{
public:
   class iterator;
... code removed for brevity ...
};

template <class T>
class vector <T> :: iterator
{
... code removed for brevity ...
};
```

In this example, `iterator` is a public nested class. The full name of the class is "`vector <T> :: iterator`". The UML class diagram would be:



Please see "Appendix H. Review: Nested Class" for details how to implement a nested class.

Examples of nested classes include:

- Every instance of an iterator presented in this text. This includes the vector iterator in Chapter 01, the set iterator in Chapter 05, the list iterator in Chapter 07, the BST iterator in Chapter 09, and the Map iterator in Chapter 10.
- The implementation of List in Chapter 07. Here the `Node` class is moved to within the `list`.
- The implementation of BST in Chapter 09. Here the `BNode` class is moved to within the `BST`.
- The implementation of Map in Chapter 10. Here the `BST` class is moved within the `map`. Note that the `BST` also has a `BNode` as a nested class.

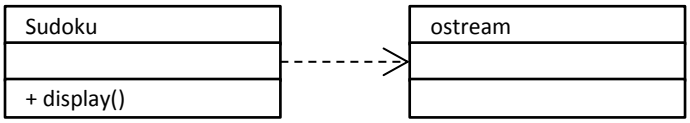
Dependency

A dependency relation exists between two classes when changes to one class affect another. Notice that composition, aggregation, inheritance, and association can all be classified as dependencies. We use the dependency relation only when the others cannot describe the relation adequately.

For example, consider the standard output stream object called `cout`. If one were to change the behavior of `cout`, it would certainly alter the output of any function or method using it. In the following example, we are using `cout` to display a game:

```
class Sudoku
{
public:
   void display()
   {
      cout << "Board…";
      …
   }
};
```

This is not an example of composition because the Sudoku class does not contain an instance of `cout`. It is not an example of aggregation because Sudoku does not maintain a reference to `cout`. It is not inheritance because Sudoku is not a derived class. Finally, it is not association because `cout` is not used as a local variable. Thus we use the default relation: dependency.

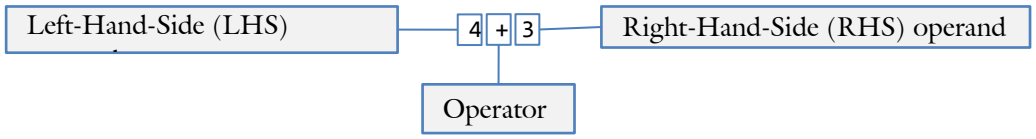


There is one example of dependency presented in this textbook:

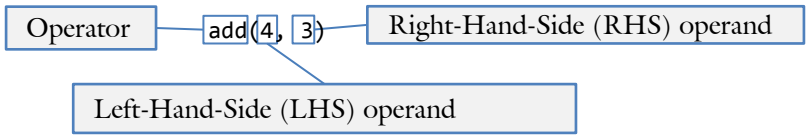
- The implementation of Map in Chapter 10. Here we use a `pair` to convert two template types (`K` and `V`) into one (`T`) for `BST`.

APPENDIX E : OPERATOR OVERLOADING

Containers in the STL use operator overloading. **Operator overloading** is the process of using a more convenient and human-readable notation for calling a function (such as "4 + 3")…



… than the functional notation common in programming languages (such as "`add(4, 3)`").



In other words, operator overloading does not add any power to C++; there is nothing you can do with operator overloading that cannot be done with traditional functions. Instead, operator overloading makes it easier and more convenient for you and others to use your code. Containers commonly use the assignment operator, the square bracket operator, the dereference operator, the comparision operators, and the increment operators.

Assignment

Conceptually the assignment (=) operator works the same as the copy constructor with the exception that a new object is not created. Instead, an exact copy of the object on the left-hand-side is made to the object on the right-hand side.

Prototype

The prototype for the assignment operator defined with the `Complex` class on the right-hand-side is:

The name of the function is "`Complex :: operator = ()`". We can call it with "`c1.operator=(c2)`" or "`c1 = c2`"

```
Complex & Complex :: operator = (const Complex & rhs);
```

Return the left-hand-side by-reference so we don't make a copy of `this`

The left-hand-side is `this` because we are overloading a member

The right-hand-side is an object constant by-reference so we don't change or make a copy of `rhs`

Implementation

The following is the implementation of the assignment operator for the `Complex` class:

```
Complex & Complex :: operator = (const Complex & rhs)
{
   this->r = rhs.r;                              // the left-hand-side of the operator
   this->i = rhs.i;                              //   comes in as "this"
   return *this;                                 // return *this because "this" is
}                                                //    a pointer to a Complex
```

The most common mistake is to forget to return `*this`, thereby making it impossible to chain the assignment operator.

Square bracket

The square bracket operator (`[]`), also known as the array index operator, is commonly used to retrieve an item from a collection. The interesting thing about this operator, however, is that the parameter does not need to be an integer.

Using the operator

The square bracket operator is commonly used with containers (classes storing data). To demonstrate, we will start with the simple container class `Array`:

```
class Array
{
... code removed for brevity ...
   private:
      double * data;                    // where the data for the array is stored
      int      sizeArray;               // the number of items currently in the array
};
```

We can use the square-bracket operator both as an accessor and as a mutator:

```
{
   Array array(10);                     // initial capacity of 10
   array[0] = 9.9;                      // change the 1st item by setting it to 9.9
   cout << array[0] << endl;            // access the 1st item, the value 9.9
}
```

The right-hand-side of the square bracket operator traditionally is an integer, corresponding to the index of the item in the container. Note, however, that this can be any data type.

The left-hand-side is the class itself, represented as `this`. If you wish the square-bracket operator to function only as a getter, make `this` constant by making the method itself constant.

The return value is the data being accessed from the container. If the square bracket operator is to function as just a getter, either return by-value or a constant by-reference. If the square bracket operator is to function as both a getter and a setter, then the function must return by-reference.

Prototype

The prototype for the square-bracket operator defined with the `Array` class on the right-hand-side is:

The name of the function is "`Complex :: operator [] ()`". We can call it with "`c.operator[](7)`" or "`c[7]`"

```
double & Array :: operator [] (int index);
```

Return value by-reference so we this operator can be both a getter and a setter

The left-hand-side is `this` because we are overloading a member

The right-hand-side is typically an index. Since we usually use a built-in data type, there is no need to make it

Implementation

The following is the implementation of the square-bracket operator for the `Array` class:

```
double & Array :: operator [] (int index) throw(bool)
{
   if (index < 0 || index >= sizeArray)
      throw false;
   return data[index];
}
```

Prefix increment & decrement

The prefix increment and decrement operators (`++x` and `--x`) are much like the add-onto operators except there is no right-hand-side. This means that they are not truly infix operators and they take only one parameter.

When to use

While the obvious uses for the increment and decrement operators are for adding or subtracting one, other metaphors apply. For example, ++ could mean "move ahead" or "next turn" or even "redo."

Using the operator

The prefix increment and decrement operators update the object before the expression is evaluated. This means that if we display the results of `++c`, we expect to see the new value instead of the old:

```
{
   Complex c(3.0, 7.0);            // (3.0 + 7.0i)
   cout << ++c << endl;            // displays "4.0 + 7.0i"
   cout <<   c << endl;            // since c changes, this also displays "4.0 + 7.0i"
 }
```

Since the increment and decrement operators are unary operators (only one operand), there is only one parameter. This parameter, the right-hand-side, must be by-reference because it changes.

The return value is the newly updated right-hand-side. This means it must be returned by-reference.

Prototype

The prototype for the prefix increment operator defined with the `Complex` class is the following:

The name of the function is "`operator ++ ()`". We can call it with "`operator++(c)`" or "`++c`"

```
Complex & operator ++ (Complex & rhs);
```

Return the right-hand-side by-reference because we return the newly updated

The right-hand-side is the object to be changed so it must be by-reference

Implementation

The following is the implementation of the prefix increment operator for the `Complex` class:

```
inline Complex & operator ++ (Complex & rhs)
{
   return rhs += 1.0;              // prefix ++ is exactly the same as += 1
}
```

It is very common to define the prefix-increment operator in terms of the add-onto operator. Just make sure that the add-onto operator is defined first.

Comparision operators

The comparisons operators such as equivalence (`==`) and greater-than (`>`) take two parameters and return a Boolean value.

Using the operator

In algebra, the notation for comparing two complex numbers is the following:

```
(3 + 7i) ≠ (2 + 9i)
```

Notice how the expression is either true or it isn't. To represent this in C++, we need two `Complex` objects and the comparison operators:

```
{
   Complex c1(3.0, 7.0);            // (3.0 + 7.0i)
   Complex c2(2.0, 9.0);            // (2.0 + 9.0i)
   if (c1 == c2)
      cout << "Same!\n";
   if (c1 != c2)
      cout << "Different!\n";
}
```

Both the left-hand-side and the right-hand-side are `Complex` objects. Since we don't want to needlessly make a copy of `lhs` or `rhs`, both are passed by-reference. Notice that we do not expect either `c1` or `c2` to change. This means that both `lhs` and `rhs` must be constant as well.

The return value of the comparison operator is a Boolean value.

Prototype

The prototype for the equivalence operator defined with the `Complex` class is the following:

The name of the function is "`operator == ()`". We can call it with "`operator==(c1, c2)`" or "`c1 == c2`"

```
bool operator == (const Complex & lhs, const Complex & rhs);
```

Return `true` or `false`, depending on whether the expression is true.

Both the left-hand-side and the right-hand-side are pass-by-reference so we don't make a copy. They are also constant so we don't change `lhs` or `rhs`

Implementation

The following is the implementation of the equivalence operator for the `Complex` class:

```
inline bool operator == (const Complex & lhs, const Complex & rhs)
{
   return (lhs.getReal()      == rhs.getReal() &&
           lhs.getImaginary() == rhs.getImaginary());
}
```

APPENDIX F : TEMPLATES

Abstract data types should be able to work with any underlying data type. This, in C++, requires the use of templates. Templates are mechanisms allowing one function or class to compile into many. For example, the following function template `display()` will be compiled into two separate functions, one for the `string` data type and one for the `double`.

```
template <class T>
void display(const T & data)
{
   cout << "The data is: "
        << data << endl;
}

int main()
{
   display(string("Text"));
   display(3.14159);
   return 0;
}
```

```
void display(const string & data)
{
   cout << "The data is: "
        << data << endl;
}

void display(const double & data)
{
   cout << "The data is: "
        << data << endl;
}

int main()
{
   display(string("Text"));
   display(3.14159);
   return 0;
}
```

The same works with classes:

```
template <class T>
class Store
{
   public:
      Store(const T & t) : t(t) {}
      T get() const { return t;  }
   private:
      T t;
};

int main()
{
   Store <double> storeNum(3.14149);
   cout << storeNum.get() << endl;

   string text("Text");
   Score <string> storeText(text);
   cout << storeText.get() << endl;
   return 0;
}
```

```
class Store
{
   public:
      Store(const double & t) : t(t) {}
      double get() const { return t;  }
   private:
      double t;
};

class Store
{
   public:
      Store(const string & t) : t(t) {}
      string get() const { return t;  }
   private:
      string t;
};

int main()
{
   Store <double> storeNum(3.14149);
   cout << storeNum.get() << endl;

   string text("Text");
   Score <string> storeText(text);
   cout << storeText.get() << endl;
   return 0;
}
```

Function template

There are two aspects to the syntax of a function template: the definition of the template data type (called the **template prefix**) and the use of the template data type in the function (called the **type parameter**).

Template prefix

In order to tell the compiler that a given function uses a template, it is necessary to use a template prefix:

```
template <class T>
```

The template prefix goes immediately before the function definition, stating that this function will have at least one parameter that uses an unknown data type.

Note that we can call our new data type "`T`" though any other valid C++ name will work. By convention we use "`T`" because data types should be TitleCased and because we don't know anything else about the data type to give it a more descriptive name.

There is one other aspect about the template prefix: it is possible for a function to accept two abstract data types corresponding to potentially dissimilar types. In this case, we can declare two templates in the prefix:

```
template <class T1, class T2>
```

Type parameter

Once a template prefix has been created, the programmer can then use the data type `T` the same as he would with any other data type in the program. This means that variables and parameters can be declared as type `T` rather than with a built-in data type (such as `int`, `float`, `bool`, etc.) or with a class (such as `Date`, `Point`, etc.).

```
{
   T variable;    // the variable is of type T
}
```

Template prefix with type parameter

Consider the standard `swap()` function:

```
void swap(float & lhs, float & rhs)
{
   float tmp = lhs;
   lhs = rhs;
   rhs = tmp;
}
```

The same function, using a template prefix and a type parameter, would be:

```
template <class T>
void swap(T & lhs, T & rhs)
{
   T tmp = lhs;
   lhs = rhs;
   rhs = tmp;
}
```

Since it is necessary for both the left-hand-side and the right-hand-side to be the same data type for the `swap()` function to work, only one type parameter is used for both the parameters.

Warning: all the type parameters represented in the template prefix must be represented in the function template parameters. If not, the compiler will not be able to tell which version of the template to use.

Class template

Defining a class template is exactly the same as defining any other class with the exception of the template prefix and the type parameter. There are, however, two additional quirks. First, the template prefix needs to go before every method definition. This can get tedious, but the C++ language demands it. Second, the full name of a template class includes the template designation. In other words, a template class `Store` is actually "`Store <T>`" because the template designation is an integral part of the class identity.

Template prefix

Just as a template prefix needed to go before the function definition of a template function, the template prefix needs to go before the class definition for a class template. Consider, for example, a class designed to store a value:

```
template <class T>
class Store
{
   public:
      Store(const T & t) : t(t) {}
      T get() const { return t;  }              // notice that methods defined inside
   private:                                     //    a class definition do not need
      T t;                                      //    the template prefix
};
```

The template prefix signifies that the data type `T` can be used anywhere in the class definition. This includes as a parameter to a member function, a local variable within a member function, and as a member variable.

Of course it is possible to define more than one data type in a template prefix:

```
template <class T1, class T2>
class StoreTwo
{
   public:
      Store(const T1 & t1, const T2 & t2) : t1(t1), t2(t2) {}
      T1 getOne() const { return t1; }
      T2 getTwo() const { return t2; }
   private:
      T1 t1;
      T2 t2;
};
```

Notice how the `getOne()` method returns a `T1` but does not take a `T1` as a parameter. This was impossible with function templates because the client cannot specify which version of the function to use. We do not have this constraint with class templates. The client specifies the version of the class at declaration time. See the section "Declaring an object from a class template" in two pages for details of how this works.

Type parameter

Once the template prefix has been defined, it can be used anywhere a standard data type can be used. In the case of class templates, this almost always means it is used to define member variables. However, it is also commonly used as parameters, return data types, and local variables.

```
template <class T>
class Store
{
   public:
      Store(const T & t) : t(t) {}          // type parameter used as a parameter
      T get() const { return t;  }          // type parameter used as a return type
   private:
      T t;                                  // type parameter used as a member variable
};
```

Class name

When a template is used with a class, the complete class name includes the template designation. Back to our previous example, there is no such thing as the "`Store`" class! Instead there is the "`Store<T>`" class.

To illustrate this point, consider the copy constructor. As you may recall from Chapter 2.4, the copy constructor takes a constant parameter of the same data type as the class. Since the data type of the class is `Store<T>` (not `Store`!), the copy-constructor must be:

```
template <class T>
class Store
{
   public:
      Store(const T & t) : t(t) {}
      Store(const Store<T> & t)            // notice the data type of the parameter
      {
         this->t = t.get();
      }
      T get() const { return t; }
      void set(const T & t);
   private:
      T t;
};
```

A common mistake is to forget the template designation in the class name.

Method definitions

The final aspect of defining a class template pertains to defining a method outside the class definition. Recall that methods defined inside a class definition are inline. Inline functions and methods should be trivial functions, consisting of just a few lines. Unlike normal non-template classes, template classes must be defined entirely in the header (`.h`) file. This includes the template class definition, inline method definitions, and non-inline definitions.

Every method definition needs to be preceded by the template prefix. Additionally, since method definitions include the class name, we need to include the template designation as well. Consider a simple `set()` method for our `Store<T>` class:

```
/***************************************
 * STORE<T> :: SET
 ***************************************/
template <class T>
void Store <T> :: set(const T & t)
{
   this->t = t;
}
```

Notice the template prefix immediately before the method definition. Also notice that the full class name of "`Store<T>`" must be used.

Nested classes

One final thought. When using templates with nested classes (see Appendix H for details about nested classes), it is necessary to include "typename" before the method definition. The most common need for this is when defining an iterator for a container:

```
template <class T>
typename array <T> :: iterator array <T> :: end ()
{
   return iterator (data + numCapacity);
}
```

APPENDIX G : ITERATORS

An **iterator** is a tool used to provide a standard way to iterate or traverse through a collection of values. They are designed to work the same regardless of the data type stored in the collection and regardless of how the collection is stored in memory. For example, an iterator used to loop through the items in a vector should work the same as an iterator used to loop through the items in a linked list.

Using iterators

For every container class, there is an associated iterator. In other words, we use a different iterator for a `string` object, a `vector`, and a `list` (the vector and the list will be discussed in Chapter 4.6).

Consider, for example, a `vector` of floating-point numbers containing the values `{1.2, 3.4, 5.6}`. A function displaying the values of the `vector` using an iterator is the following:

The data type of the iterator includes the data type of container we are iterating through:
`vector <float> :: iterator it`

The container itself indicates where the loop starts:
`values.begin()`

The container also indicates where the loop ends:
`values.end()`

Advance through the loop with the ++ operator:
`++it`

```
ostream & operator << (ostream & out, vector <float> & values)
{
   for (vector <float> :: iterator it = values.begin(); it != values.end(); ++it)
       out << *it << endl;
   return out;
}
```

Retrieve data in the container through the dereference

There are several things to observe about the syntax for iterators: the iterator data type, the process of initializing an iterator, the process of verifying the iterator is not beyond the end of the data, the process of advancing the iterator, and finally the process of retrieving data from the container.

Building iterators

Most ADTs need to have an iterator class enabling the client to visit every element contained therein. Fortunately, it is not difficult to build an iterator for a given container; one only needs to abstract the components of the non-iterator version of the loop. First, consider the loop to iterate through a string object:

```
for (const char * p = text.c_str(); p != text.buffer + test.size; p++)
   cout << *p << endl;
```

We would like this loop to work with an iterator:

```
for (StringIterator it = text.begin(); it != text.end(); it++)
   cout << *it << endl;
```

The first questions is: what will be the member variable for the `StringIterator` class? The obvious answer is: a pointer to a character, the same data type used for the non-iterator version of the FOR loop.

Start the loop

With our storage strategy found, we can define the `begin()` function in our `String` class:

```
class String
{
... code removed for brevity ...
      char * begin() { return buffer; }
... code removed for brevity ...
   private:
      char * buffer;
      int size;
};
```

Now we can define our `StringIterator` class to accept this data:

```
class StringIterator
{
   public:
      StringIterator(char * p) : p(p)        {              }
      StringIterator & operator = (char * p) { this->p = p; }
   private:
      char * p;
};
```

End the loop

How do we know we are at the end of the loop? The answer is when the pointer is referring to the null character. Therefore, our `String::end()` method should return a pointer to the null character:

```
class String
{
... code removed for brevity ...
    char * end() { return buffer + size; }
... code removed for brevity ...
};
```

Now we can define the not-equals operator in our `StringIterator` class to test this condition:

```
class StringIterator
{
... code removed for brevity ...
    bool operator != (const char * p) const { return this->p != p; }
... code removed for brevity ...
};
```

Advance

With the start of the loop defined and the end of the loop defined, we next need to define how to advance our iterator. We do not need to make any modifications to the `String` class to do this, `StringIterator` should be able to do it itself.

```
class StringIterator
{
... code removed for brevity ...
   StringIterator & operator ++ () { p++; return *this; }
... code removed for brevity ...
};
```

Retrieve data

The final step in the iterator definition process is to fetch the data referred to by the iterator. We use the dereference operator to do that. Since our member variable is a pointer to the data, the function is trivial:

```
class StringIterator
{
... code removed for brevity ...
    char & operator * () { return *p; }
... code removed for brevity ...
};
```

Observe how we return the character by-reference. This is important because the client of the iterator class needs to not only retrieve data from the string, but also set data to the string:

```
{
   StringIterator it = text.begin();
   *it = 'z';                                // use * to set data in text
   cout << *it << endl;                      // use * to retrieve data from text
}
```

The final string iterator class:

```
class StringIterator
{
public:
   StringIterator (                          ) : p(NULL)  {}
   StringIterator (char * p                  ) : p(NULL)  {}
   StringIterator (const StringIterator & rhs) : p(rhs.p) {}

   iterator & operator = (char *p)        { this->p = p;                            }
   bool operator == (const char *p) const { return this->p == p;                    }
   bool operator != (const char *p) const { return this->p != p;                    }

   iterator & operator ++ ()              { p++; return *this;                      }
   iterator operator(int postfix)         { iterator it(*this); p++; return it;     }
   char & operator * ()                   { return *p;                              }

private:
   char * p;
};
```

One final note. Iterators are commonly nested classes embedded in the container they are designed to iterate through. Thus the code would be something like:

```
namespace std
{
   class string
   {
   public:
... code removed for brevity ...
      class iterator
      {
... code removed for brevity ...
      };
... code removed for brevity ...
   }; // class string
}; // namespace
```

With this nested class (see Appendix H. Review: Nested Class) and namespace (see Appendix I. Review: Namespace) code, we will then declare a string iterator as:

```
{
   std::string::iterator it;
}
```

APPENDIX H : NESTED CLASS

In C++, it is possible to define a class within the definition of another class. This is called a nested class. Nested classes are simple in concept but some of the details can be a little counterintuitive. Before going into the details, the concept will be presented.

If one were to define a class outside a function, then the class would be said to have "global scope." This means that anyone can instantiate an object based on this definition. Of course this is the most common way to define classes.

It turns out that we can also define a class or structure within a function or method. Here the scope of the definition is limited to that function or method, meaning an object can be created from that definition only within the limited scope.

```
double computeTax(double income)
{
   struct Bracket
   {
      double high,
      double low,
      double rate,
      double fixedAmount
   };
   Bracket taxBrackets[] =
   {
```

... code removed for brevity ...

```
   };

   // compute tax bracket
```

... code removed for brevity ...

```
   return tax;
}
```

In the above example, we can only create a `Bracket` object from within the `computeTax()` function. It turns out, we can define classes within the scope of another class in much the same way. We call the outter class the "enclosing class" and we define the inner class a "nested class."

Private

One very common use of a nested class is to make the inner class `private` and thus only accessible to the enclosing class. This is particularly useful when the nested class only makes sense within the context of the enclosing class. For example, one may wish to put the linked list definition within the list class:

```
class list                // the enclosing class
{
```

... code removed for brevity ...

```
private:
   class Node             // since "node" is private, only "list" can refer to it
   {
```

... code removed for brevity ...

```
   };
   Node * pHead;          // here "list" can refer to the nested class "Node"
};
```

In this example, only the `list` class can refer directly to the `Node` class. This is desirable because we wish to hide the implementation details of `list` from the client. By using a nested class, there is no indication that the underlying data-structure of `list` is a linked list.

Public

A second use for sub-classes is when one class serves as a helper to the enclosing class. In this case, the nested class is public and therefore accessible to the client. A good example of this is an iterator:

```
template <class T>
class list
{
public:
   class iterator          // iterator is declared inside the list class
   {
... code removed for brevity ...
   };
};
```

In this example, one could define a list iterator with:

```
list <int> :: iterator it;
```

Notice how the scope resolution operator (the double-colon `::`) is used when defining a nested class.

Defining Nested Classes

A nested class can be defined completely within the enclosing class. This is usually done when the nested class is trivial, has a small number of methods, and when each method has a simple definition. When these conditions are not met, it is often convenient to define the nested class outside of the enclosing class.

```
template <class T>
class list
{
public:
   class iterator;             // forward declaration of the iterator class
... code removed for brevity ...
};

template <class T>
class list <T> :: iterator     // iterator is declared outside the list class
{
... code removed for brevity ...
};
```

The forward declaration of `iterator` in `list` indicates that it is a nested class. However, rather than cluttering the `list` definition, `iterator` is defined outside of `list`.

Defining Nested Methods

Whether a nested class is defined inside the enclosing class our outside, there are two options as to how to define a method. The first is to simply define the method inline as we do with methods in normal classes. The second is to define the method outside the class. Here, the syntax can be a bit more complicated. To show how this works, we will start with a simple nested class:

```
class Enclosing
{
public:
   class Nested
   {
   public:
      void method();
   };
};
```

We need to define `method()` because it is nothing but a prototype in `Nested`. When we do this, we need to specify the full name of `method()` in the definition:

```
void Enclosing :: Nested :: method()
{
   cout << "This is a method in a nested class\n";
}
```

Notice the first scope resolution operator `::` between `Enclosing` and `Nested` signifies that `Nested` is part of `Enclosing`. The second scope resolution operator signifies that `method()` is part of `Nested`. Therefore the complete name of the method is: `Enclosing :: Nested :: method()`.

Templates and Nested Classes

What happens when the enclosing class is a template? Consider the following class definition:

```
template <class T>
class TemplateEnclosing
{
public:
   class Nested;
};
```

First, we need to define the nested class. Notice how we put the template on the enclosing class but not on the nested class:

```
template <class T>
class TemplateEnclosing <T> :: Nested
{
public:
   void method();
};
```

Now we need to define the method.

```
template <class T>
void TemplateEnclosing <T> :: Nested :: method()
{
   cout << "This is a method in a nested class\n";
}
```

The complete name of the method is: `TemplateEnclosing <T> :: Nested :: method()`.

Typename

There is one final complication. When we return an object that is a template nested class (isn't that a mouthful!), then we need to remind the compiler that we are working with a template. We perform this reminder with the "`typename`" keyword. This can be very difficult to understand. Perhaps it is best explained by example.

Consider a template container called `array`. This will behave much like the built-in array type except it will have bounds checking on the `[]` operator and will have an iterator. We wish to use this new `array` type in the standard container fashion:

```
{
   // create an array of 10 floats
   array <float> numbers(10);

   // iterate through our numbers
   array <float> :: iterator it;
   for (it = numbers.begin(); it != numbers.end(); ++it)
      *it = 42.0;
}
```

This looks like it works much like every other container in the standard template library. The class definition of our array type would be something like this:

```
template <class T>
class array
{
public:
... code removed for brevity ...
   class iterator;
   iterator begin();
   iterator end();
... code removed for brevity ...
};
```

Of course much remains to be done with this `array` class, but we will focus on the `begin()` and `end()` method for now. Both of these methods return an `iterator`. The full type name of these iterators are `array <T> :: iterator`. We would expect to define these methods as the following:

```
template <class T>
array <T> :: iterator array <T> :: begin()        // this will produce an error
{
... code removed for brevity ...
}
```

Again, the return type of this method is `array <T> :: iterator` and the method name is `array <T> :: begin()`. Unfortunately, the compiler is not complete sure of what you meant by this statement. It needs to be reminded that `array` is a template and therefore "`array <T> :: iterator`" is a typename. We give the compiler that reminder by adding the `typename` keyword to the return type of the method:

```
template <class T>
typename array <T> :: iterator array <T> :: begin()
{
... code removed for brevity ...
}
```

Your big takeway is that you will need to add the `typename` keyword into the return type of any method returning a template's nested class. The compiler will give you a hint about this, but the hint will be followed by a few pages of useless compiler garbage.

APPENDIX I : NAMESPACE

When working with very large systems, it often becomes difficult to come up with unique names for all the functions and classes. As a result, names tend to be long and drawn out. Namespaces offer a solution to this problem, allowing for short and descriptive names that do not collide with the names of things in other parts of the program.

Perhaps the best analogy for namespaces is phone number area codes. My office number is 208 496 3487. However, you do not need to type the area code if you are calling from the same town as my office. In this case, you can simply use the 7-digit number 496 3487. This means, we have two versions of the number: the fully qualified number (208 496 3487) and the short or brief version (496 3487).

In C++, we can declare a namespace with the `namespace` keyword. Everything in the enclosing curly braces is part of that namespace. This could include functions, classes, or even global variables.

```
namespace custom
{
   class list
   {
... code removed for brevity ...
   };
};
```

If one were to declare a variable of type `list`, there are two options: either use the fully qualified names or use the brief version. To use the fully qualified name, use the namespace label followed by the scope resolution operator:

```
{
   custom::list l;             // use the fully qualified namespace;
}
```

Next, we can specify that we will be using the brief version of all members of the special namespace:

```
{
   using namespace custom:     // the brief version can be used by everything in custom
   list l;                     // this refers to custom::list rather than std::list
}
```

Finally, we could specify that we will be using the brief version of only one part of the `custom` namespace:

```
{
   using custom::list;         // only list can use the brief version
   list l;
}
```

Perhaps the most familiar use of namespaces is the standard namespace `std`. It is in this namespace that `cin` and `cout` reside. While we normally say "`using namespace std`," we don't have to.

```
#include <iostream>

int main()
{
   std::cout << "Hello world" << std::endl;   // use the fully qualified names.
}
```

All of the classes in the STL are in the `std` namespace. We can define our own version of these classes with the same name by putting them in a different namespace.

APPENDIX J : SEPARATE COMPILATION

Separate compilation is the process of designing a program with multiple files each of which is a manageable size. In the context of data structures, one could put a single class into each header file thereby making it easy and convenient to use it in a project.

Including Header Files

ADTs are almost always template classes and, as such, are completely defined in header files. In C++, header files end with the `.h` suffix. In order to include the code from a header file into a source file, it is necessary to use the `#include` pre-processor directive. This directive tells the pre-processor to copy-paste the code from the specified file directly into the source file. Consider the following line of code:

```
#include <vector>
```

This instructs the pre-processor to include a standard template library (`vector` in this case) into the program we are writing. We can also include files from our own directory using a similar syntax:

```
#include "vector.h"
```

There is no mystery what this line of code does. The above line of code is exactly the same as the programmer physically copying the code from `vector.h` and pasting it into the program at that point.

Notice that there is a slight difference between including the `vector` library and the `vector.h` header. In the former case, there are chevrons `<>` around the file name. This tells the pre-processor to look in the standard library directory for the file. In the latter case, there are double quotes `""` around the file name. This tells the pre-processor to look in the current directory for the file.

Avoiding Multiple Definitions

What would happen if one were to include the same header file twice? This would be the same as having two copies of the contents of the header files physically copy-pasted into a single source file. The result would be a compile error complaining about multiple definitions of the same class. How can this be avoided? There are two ways. The first is to put a special `#ifndef` around each header file:

```
#ifndef VECTOR_H
#define VECTOR_H

/**************************************
 * VECTOR
 * A custom version of std::vector
 **************************************/
namespace custom
{
   template <class T>
   class vector
   {
... code removed for brevity ...
   };
};

#endif // VECTOR_H
```

The first time this file is included, `VECTOR_H` is not defined in the code so it gets included. The second time, however, `VECTOR_H` is defined so the entire body of the header file is skipped. This mechanism ensures only one copy of a header file is ever included in the compilation of a source file. In other words, it enables

programmers to include header files without worrying about including the same file twice. This is called the `#include` guard idiom and it is the only mechanism described in the C++ standard to guard against multiple include errors.

A second way to accomplish this is to put a special compiler directive at the beginning of the file. This directive tells the compiler to skip the file once it has been compiled once.

```
#pragma once

/***************************************
 * VECTOR
 * A custom version of std::vector
 ***************************************/
namespace custom
{
   template <class T>
   class vector
   {
... code removed for brevity ...
   };
};
```

This mechanism is called the multiple-include optimization. Note that the `#pragma once` mechanism is not part of the C++ standard but is adopted by virtually all C++ compilers.

Makefile

A `makefile` is a mechanism to compile multiple source files. It accomplishes this by selectively compiling the object files of a project according to what has changed since the last compilation. The essence of `makefile` is this selection process.

A `makefile` is a file that contains the dependencies for a given project. In the `makefile`, the programmer indicates how to tell if a given file needs to be rebuilt, and how. Each dependency, called a rule, has three components:

- **Target**: the name of the file that might need to be rebuilt.
- **Dependency List**: the list of dependencies used to determine if target needs to be rebuilt.
- **Recipe**: instructions how to build the target.

The syntax for a recipe is:

```
<target> : <Dependency List>
        <Recipe>
```

Note that there must be a tab (not spaces) before recipe.

For example, we will need to re-build the file "`a.out`" if the source-file has changed (`vectorTest.cpp`) or if the header file has changed (`vector.h`). If any of these has changed, then we will need to rebuild the `a.out`.

```
a.out : vectorTest.cpp vector.h
        g++ vectorTest.cpp
```

One final thing to know about the `makefile` is that line comments start with the `#` rather than `//`. The complete `makefile` for the this simple program is:

```
# Rebuild a.out and generate a new TAR file
a.out : vectorTest.cpp vector.h
        g++ vectorTest.cpp
        tar -cf vector.tar vector.h vectorTest.cpp makefile
```

APPENDIX K : RECURSION

Recursion is the process of a function calling itself in the process of computing a result. At first this may seem counterintuitive: if a function cannot compute the answer without help, how can calling itself get any closer? It seems like the blind leading the blind! Fortunately, recursion does not work like that. This appendix will demonstrate how recursion works and why recursion has the same computational power as a traditional loop.

Two Parts

Every recursive function as two distinct parts: the end condition and progress step.

- The **end condition** is how the function knows that it has found the solution. Typically, the end condition is trivial. It is when the return value for a given input is obvious.
- The **progress step** represents the process of taking a large problem and breaking it into a slightly smaller pieces where these pieces are closer to the end condition.

To illustrate this, consider a function to add two numbers:

```
int add(int value1, int value2)
{
   return value1 + value2;
}
```

Now let's say, for the sake of argument, that we do not have the plus operator defined for all numbers. In fact, we only have the plus operator defined for the number one. This means our addition function would need to be written something like this:

```
int add(int value1, int value2)
{
   int sum = value1;
   while (value2 != 0)
   {
      sum++;
      value2--;
   }
   return sum;
}
```

Here we have turned `add(7, 3)` into `7 + 1 + 1 + 1`.

Now we will perform the same thing using recursion. The end condition is when `value2 == 0`. Since `x + 0 = x;` we can simply return `value1` when `value2 == 0`. The progress step will be to bring us closer to the end condition. We will say that `add(7, 3)` is the same thing as `1 + add(7, 2)`. Putting this all together, the recursive function will be:

```
int add(int value1, int value2)
{
   if (value2 == 0)
      return value1;                       // end condition
   else
      return 1 + add(value1, value2 - 1);  // progress step
}
```

Take a careful look at this code. There are two parts:

- `add(value1, 0) == value1`
- `add(value1, value2) == 1 + add(value1, value2 - 1)`

We will test this by adding `7` and `3`:

```
add(7,3) = 1 + add(7, 2)              // progress step
         = 1 + (1 + add(7, 1))        // progress step
         = 1 + (1 + (1 + add(7, 0)))  // progress step
         = 1 + (1 + (1 + 7))          // end condition
         = 1 + 1 + 1 + 7
         = 10
```

From this example, notice how our end condition tells us when to stop calling additional copies of `add()`. This end condition, it turns out, is the same end condition of our loop solution. Also, notice how the progress step brings us closer to the end condition. In this case, every time the function is executed, `value2` decreases by one which brings us closer to the end condition. Closer inspection reveals that the progress step is exactly the same as the decrement statement in the loop condition.

Every WHILE loop has two steps (the end condition and the progress step). Similarly, every recursion has two steps (the end condition and the progress step). Thus we can see that we can turn any WHILE loop into a recursive function and every recursive function can be turned into a WHILE loop. These two mechanisms have the same descriptive power.

Examples

To illustrate this point, we will present several examples. Each will be presented both as a WHILE loop and as a recursive function.

Display text

Consider a function to display the contents of a C-String on the screen. We will call this function `display()`. The WHILE loop solution may look something like this:

```
void display(char * text)
{
   while (*text)      // end condition
   {
      cout << *text;
      ++text;         // progress step
   }
}
```

We can see that we iterate through the c-string until the `NULL` character is reached. When this happens, we exit the loop and the function as well. This same function can be written recursively:

```
void display(char * text)
{
   if (*text)             // end condition
   {
      cout << *text;
      display(text + 1);  // progress step
   }
}
```

Here we can see that both functions display the contents of the c-string one character at a time and advance the pointer by one increment with every iteration. We will test the recursive function by calling `display()` with "`Hello`"

```
display({'H', 'e', 'l', 'l', 'o', '\0'}) = cout << 'H'; display({'e', 'l', 'l', 'o', '\0'})
                                           cout << 'H' << 'e'; display({'l', 'l', 'o', '\0'})
                                           cout << 'H' << 'e' << 'l'; display({'l', 'o', '\0'})
                                           cout << 'H' << 'e' << 'l' << 'l'; display({'o', '\0'})
                                           cout << 'H' << 'e' << 'l' << 'l' << 'o'; display({'\0'})
                                           cout << 'H' << 'e' << 'l' << 'l' << 'o';
```

Factorial

This next example will demonstrate how to compute factorial. As you may recall, computing a Factorial is the process of multiplying together all the numbers lower than the number. For example 5! (pronounced "five factorial") is 5 x 4 x 3 x 2 x 1.

We can compute factorial using a traditional loop:

```
int factorial(int num)
{
   int answer = 1;
   while (num > 1)                        // end condition
   {
      answer *= num;
      num--;                              // progress step
   }
   return answer;
}
```

We can do the same thing using recursion:

```
int factorial(int num)
{
   if (num == 1)                          // end condition
      return 1;
   else
      return factorial(num – 1) * num;    // progress step
}
```

Here we will compute 5!.

```
factorial(5) = factorial(4) * 5
             = (factorial(3) * 4) * 5
             = ((factorial(2) * 3) * 4) * 5
             = (((factorial(1) * 2) * 3) * 4) * 5
             = (((1 * 2) * 3) * 4) * 5
```

Binary Search

One final example. Say you have an array of n numbers. You wish to determine if a given value is present in this array. Because the array is sorted, we will chose to use a binary search. As you may recall, a binary search involves looping through an array where every iteration the possible values is decreased by a factor of two. With the first iteration, half the array is ruled out. With the second iteration, another half is ruled out. This continues until the element is found or the list of possible values is zero.

First, we will perform the binary search using a traditional WHILE loop. Note that my `iFirst` and `iLast` variables keep track of the bounds of the array in which the value may reside.

```
bool binarySearch(const int array[], int n, int value)
{
   int iFirst = 0;
   int iLast = n - 1;

   while (iFirst <= iLast)
   {
      int iMiddle = (iFirst + iLast) / 2;

      if (array[iMiddle] == value) // if the element in the middle
         return true;              //   is it, then exit

      if (value > array[iMiddle])  // if the searched value is
         iFirst = iMiddle + 1;     //   bigger, rule out front
      else                         // otherwise, rule out
         iLast = iMiddle - 1;      //   the back
   }
   // if it is not here, return false
   return false;
}
```

This loop can be converted into recursion:

```
bool binarySearch(const int array[], int n, int value)
{
    int iMiddle = n / 2;

    // end condition: the list is empty or the value is found
    if (n == 0)
       return false;
    if (array[iMiddle] == value)
       return true;

    // progress step
    if (value > array[iMiddle]
      return binarySearch(array + iMiddle + 1, n - iMiddle - 1, value);
   else
      return binarySearch(array, iMiddle, value);

}
```

Efficiency

One final thought about recursion. Though it has the power of loops and often provides a more elegant than loops, recursive solutions are almost always more expensive than loops. In the examples presented here, the loop version of the functions have one loop control variable that changed its value with every iteration of the loop. The recursive versions, on the other hand, create an extra copy of the control variable with every call to the recursive function. If a recursive function got called a thousand times, then there would be a thousand copies of the variable in memory at one point in time.

APPENDIX L : DEBUGGING

An essential skill in any programming context is debugging. Though there is some debate as to where the word "debugging" originated, many attribute it to the process of removing cockroaches from the original Eniac computer in the University of Pennsylvania engineering building basement. When Eniac malfunctioned, a technician would need to crawl through the enormous computer to remove the unfortunate cockroach whose corpse was shorting out an electrical circuit.

Debugging is especially important when working with data structures because the bugs are difficult to find and isolate. To help with this, three techniques will be presented: asserts, debug statements, debug code, and a tool called a debugger.

Asserts

An assert is similar to the "check engine" warning light in your car. It is a light that the driver never expects to see illuminated but, when it is, tells him or her to get the car fixed at the earliest possible time.

In C++, an assert appears as a function call taking a Boolean value as a parameter. If the parameter evaluates to true, then the program continues to function without interruption. If the parameter evaluates to false, then the program halts and presents the user with an error message. For example, consider the following code:

```
void displayMessage(const char * message)
{
   cout << message << endl;
}
```

What would happen if this function was called with a `NULL` pointer?

```
displayMessage(NULL);
```

The end result would be a crash! When the function attempts to dereference the `NULL` pointer, then a segmentation fault would result. The programmer can prepare for this bug by adding an assert. This will serve to inform the caller what is expected and notify the caller if the expectation is not met:

```
void displayMessage(const char * message)
{
   assert(message != NULL);
   cout << message << endl;
}
```

Using Asserts

Asserts are not built into the C++ language. They must be specifically included with the cassert library:

```
#include <cassert>
```

With this library included, it is possible to add asserts to any part of your code. Typically it is done three places:

1. At the beginning of a function. Here the purpose is to validate that the function is correctly called by checking each parameter. If invalid data is passed, there is no point in looking for the bug inside the function! Look at the caller instead.
2. Whenever a calculation is made. If a calculation is made or if data is returned from another function, add an assert or two to verify that the result is what you expect.
3. When returning out of a function. Again, you want to offer the caller a guarantee that the data being returned is valid.

Asserts are a wonderful tool because they automatically help you localize errors without incurring any performance costs. This is accomplished because asserts are automatically removed from the code before we sent the code to the customer. In other words, it is possible to compile a version of the code with all the assert code ignored. This is done with the `NDEBUG #include` or with the `-DNDEBUG` compiler switch.

For example, consider the following code:

```
#define NDEBUG
void function()
{
   assert(false);
}
```

Because we are sending false to the assert parameter, it appears that this assert will fire every time `function()` is called. However, notice the `NDEBUG` pre-compiler directive immediately before the function definition. This directive means that the asserts are effectively turned-off.

We can also turn off asserts through a command line parameter:

```
g++ file.cpp -DNDEBUG
```

The "`-NDEBUG`" statement does exactly the same thing as the "`#define NDEBUG`".

Building an Assert

The mystery of asserts can easily be reduced by building one from scratch. We almost never have to do this in practice because most languages have something equivalent to asserts built in.

It turns out that asserts are easy to build. There are only two special things you need to know about: `__FILE__` and `__LINE__`. The first is a special pre-compiler directive that allows the programmer to ask the compiler which file is currently being compiled. The second allows the programmer to ask which line of code is currently being compiled. For example, consider the following code fragment in "`test.cpp`":

```
{                                          // line 19
   cout << "file: " << __FILE__ << endl;   // line 20
   cout << "line: " << __LINE__ << endl;   // line 21
}                                          // line 22
```

If this program were to be run, the following output would result:

```
file: test.cpp
line: 21
```

Our first attempt at an assert would be the following:

```
void assert(bool test)
{
   // the assert only fires when the test evaluates to false
   if (!test)
   {
      // display the error message
      cerr << "Assertion failed " << __FILE__
           << " line "           << __LINE__ << endl;

      // quit the program
      exit(1);
   }
}
```

This works great up to a point. If `test` is `true`, we do nothing. If `test` is `false`, then we display an error message and quit. The problem is that we display the display the file name and line number of the `assert()` function, not that of the caller. We can get around this with a bit of trickery:

```
inline assertCode(bool test, const char * file, int line)
{
   if (!test)
   {
      cerr << "Assertion failed " << __FILE__
           << " line "           << __LINE__ << endl;
      exit(1);
   }
}
#define assert(test) assertCode(test, __FILE__, __LINE__)
```

Notice how `assert()` is now a macro. This means the `assertCode()` function call (including the `__FILE__` and `__LINE__` parameters) will be inserted in the client's code. Thus we will get the correct file name and line number.

There is one last detail that needs to be made. We want the assert to disappear if `NDEBUG` is defined. This is done with some more `#define` trickery, the macro will remove all instances of calls to assert. Thus the complete code is:

```
#ifdef NDEBUG
#define assert(test)
#else
inline assertCode(bool test, const char * file, int line)
{
   if (!test)
   {
      cerr << "Assertion failed " << __FILE__
           << " line "           << __LINE__ << endl;
      exit(1);
   }
}
#define assert(test) assertCode(test, __FILE__, __LINE__)
#endif // !NDEBUG
```

If the user defines `NDEBUG`, then all instances of `assert()` will be compiled away into nothing.

Debug Statements

When a program behaves differently than one would expect, it is often helpful to insert display statements for the purpose of learning what function has been called or the state of various variables. This can be accomplished as simply as inserting COUT statements:

```
template <class T>
T & array <T> :: operator [] (int index)
{
   std::cout << "Here!";
   return data[index];
}
```

Notice the `cout << "Here!";` statement. The purpose is to find out if the function `operator[]` got called in a given circumstance. Unfortunately, there are several problems with this approach:

1. If we run this program, we will probably get a bunch of "`Here!`"s displayed. It will make it difficult to tell what is going on.
2. Because there is no newline, the COUT buffer is not flushed. This means the last one or more words might not be displayed on the screen before the bug occurs.
3. The output will interfere with testBed or intermingle with the desired output.
4. It is difficult to remove all these statements when they are added.

We can solve all of these problems with the following:

```
template <class T>
T & array <T> :: operator [] (int index)
{
   Debug(std::cerr << "array[" << index << "]\n");
   Debug(std::cerr << "\t" << data[index] << "\n");

   return data[index];
}
```

There are several parts to this solution. First, we use cerr rather than cout. This means that output will be diverted to the "standard error stream" rather than the "standard output stream." Any program looking for a given output will not be confused by the extra output.

Second, `cerr` flushes the buffer with every output. While cout waits until a newline to actually display characters on the screen, `cerr` displays characters immediately. Thus it will never be the case that the program crashes with data still on the output buffer.

Third, the first display item will identify which function is being called. This makes it very easy to associate debug output with a given location in the code.

Finally, all the debug output statements are surrounded with a "`Debug()`" macro. This can be defined as:

```
#ifdef NDEBUG
#define Debug(statement)
#else // !NDEBUG
#define Debug(statement) statement
#endif // DEBUG
```

Thus we can effectively turn off all debug output with the `-DNDEBUG` compiler switch.

Debug Code

Sometimes it is helpful to write entire functions whose sole purpose is to help the programmer visualize what is going on. This is one step beyond the debug display statements we discussed earlier.

For example, imagine we have a class called `custom::array`. This is a container class much like the STL's `std::array`. In order to help visualize the state of the array at a given point in the execution, we will write the following function:

```
#ifndef NDEBUG
/*******************************************
 * ARRAY : DISPLAY
 * A debug utility to display the contents of the array
 *******************************************/
template <class T>
void array <T> :: display() const
{
   std::cerr << "array<T>::display()\n";
   std::cerr << "\tnum = " << num << "\n";
   for (int i = 0; i < num; i++)
      std::cerr << "\tdata[" << i << "] = " << data[i] << "\n";
}
#endif // NDEBUG
```

If the array had 5 elements, each containing the name of a dog, the output might be:

```
array<T>::display()
        num = 5
        data[0] = Fido
        data[1] = Buddy
        data[2] = Buster
        data[3] = Oscar
        data[4] = Princess
```

It is also useful to write a debug function to validate certain data structures. For example, consider a class called `Sudoku` which contains a 9x9 number puzzle. As the rules of Sudoku demand that all the cells have a value of 1-9 or the number 0 indicating that the square is empty, one might want to write a debug function to validate this:

```
Sudoku::validate()
{
   for (int row = 0; row < 9; row++)
      for (int col = 0; col < 9; col++)
         if (board[row][col] > 9 || board[row][col] < 0)
            cerr << "board[" << row << "][" << col << "] = "
                 << board[row][col] << endl;
}
```

The problem with this function, as with the problem with an earlier version of our do-it-yourself assert function, is that we do not tell the programmer the caller responsible for the invalid board. We can rectify this with some `#define` tricks:

```
#ifdef NDEBUG

#define validate()                    // compile this to nothing

#else

#define validate() validate(__LINE__) // call the debug function
Sudoku::validateCode(int line)
{
   for (int row = 0; row < 9; row++)
      for (int col = 0; col < 9; col++)
         if (board[row][col] > 9 || board[row][col] < 0)
            cerr << "board[" << row << "][" << col << "] = "
                 << board[row][col] <<
                 << " from line number " << line << endl;
}
#endif
```

Now, whenever one wishes to verify that the board is in a good state, one only needs to call the validate function. This will display the line number of the caller if something bad happens:

```
Sudoku::display()
{
   validate();                // this line number will be displayed
   …
}
```

Debugger

The final debugging method is actually a tool: a debugger. Debuggers are powerful tools enabling us to stop programs mid-execution and make inquiries as to its state. Most modern development tools such as Visual Studio come with powerful debuggers built in.

One of the oldest debuggers is the GNU Debugger (GDB). It is included in virtually all Unix and Linux distributions and is freely available on just about every platform. Debuggers like GDB work by inserting extra information called symbols into the executable so, when the program execution is paused, things like function and variable names can be recovered. To debug a program with GDB, we go through four steps: compile in the symbols, load the program, execute the program, and inquire as to the source of the problem.

1. Compile in the symbols

In order to insert these symbols into our programs, we need to add the –g debug switch:

```
g++ hello.cpp -g
```

Note that if there are several files compiled together (as is often the case with data structures), each .o needs to include the `-g` switch.

2. Load the program

Once the program is compiled, we load the executable into the debugger with:

```
gdb a.out
```

This launches GDB and loads the program but our program is not executing. From here, we get the GDB prompt:

```
(gdb)
```

Note that GDB has an entire command line language that is very rich. There are many things we can do at this point which are beyond the scope of this appendix.

3. Execute the program

From the GDB prompt, we can now execute the program with the "run" command:

```
(gdb) run
```

The program from this point will execute exactly the same as if we simply typed "`a.out`" from the command prompt.

4. Inquire

If a bug is encountered and the program crashes, we can inquire as to what the problem may have been. The most useful tool is called a "back trace:"

```
(gdb) bt
```

This program will list all the functions in the call stack that have been called, starting with the function where the crash occurred and working its way back to `main()`. Thus one can quickly find the location of a crash without having to write any extra debug code.

`<`	An operator used for ordering elements in many containers. The set and map use the less-than operator. Also, the various sort functions order elements through the less-than operator.	Chapter 05 Chapter 09 Chapter 11
2-3-4 tree	A non-binary self-balancing tree similar to a Red-Black tree.	Chapter 09
adjacency-list	An implementation of a graph consisting of an array where each element corresponds to a vertex in the graph. These elements are lists enumerating the vertices that can be reached from a given vertex.	Chapter 13
adjacency-matrix	An implementation of a graph consisting of a 2-dimensional array of Boolean values. Each column of the array corresponds to a source vertex and each row corresponds to a destination vertex. If a given square in the array has the value `true`, then an edge exists between the corresponding source and destination.	Chapter 13
ADT	An acronym for Abstract Data Type.	Chapter 01
ancestor node	A node that can be reached by following a succession of parent pointers.	Chapter 08
and	A Boolean operator corresponding to the mathematical notion of conjunction. It returns `true` only if both operands are `true` `b3 = b1 && b2;`	Chapter 00
arc	The connection between nodes in a tree. It is also the connection between two vertices in a graph.	Chapter 08 Chapter 13
balanced	A binary tree where the height of the left sub-tree differs from that of the right sub-tree by at most one for all nodes in the tree.	Chapter 08
binary node	A data structure consisting of a value and three pointers: left, right, and parent.	Chapter 08
binary tree	A collection of binary nodes connected by pointers	Chapter 08
`bool`	A data-type used to represent Boolean data. This is data that can be one of two values: `true` or `false`.	Chapter 00
BST	An acronym for Binary Search Tree, is a special type of tree where each the values in each node are ordered from left to right.	Chapter 09
bubble sort	The slowest of the common sorting algorithms, the bubble sort is characterized by searching for the largest item, then the next largest, and so on. The bubble sort is $O(n^2)$.	Chapter 11
bucket	A part of a hash table that may contain zero, one, or more than one element.	Chapter 12
call stack	The collection of functions that were called before the current function was executed where the first function in the stack is `main()`.	Chapter 09

canonical form	The standard way to represent something. For example, the canonical form of one half is ½ rather than 2/4.	Chapter 00
`capacity()`	A method returning the amount of available space in a container. `c = s.capacity();`	Chapter 01
chaining	A method of handling hash table collisions.	Chapter 12
child node	A binary node connected to a parent node through a parent pointer.	Chapter 08
class	Classes are custom data-types consisting of data (called member variables) and functions (called member functions or methods).	Chapter 00
collision	A condition occurring when more than one value maps to a given bucket in the hash table.	Chapter 12
complete	A binary tree where all the levels are filled except possibly for the final level.	Chapter 08
correctness	The measure of whether a function, data-type, or data structure operates the way it is intended. In other words, does it fulfill the stated or implied contract?	Introduction
descendant node	A node that can be reached by following a succession of parent pointers.	Chapter 08
directed arc	A one-way connection between nodes in a tree or graph. Directed arcs are commonly drawn with arrows whereas undirected arcs lack the arrow.	Chapter 08 Chapter 13
edge	On a graph, an edge is the connection between vertices. Other names are arc, line, branch, and link.	Chapter 13
element	One item in a collection. An abstract data type consists of a collection of elements.	Introduction
E notation	A way to represent a number similar to scientific notation. The number 1888 could be represented as 1.888×10^3 in scientific notation or `1.888e3` in E notation.	Chapter 00
`erase()`	A method in many ADTs that removes one element from the collection. `l.erase(it);`	Chapter 01
exponent	The component of the scientific notation, E notation, or floating point representation of a number referring to how many digits or bits are to the left of the decimal mark. For example, 1.888×10^3 has an exponent of 3.	Chapter 00
exponent bias	A way of representing numbers in binary similar to two's complement designed specifically for floating point numbers.	Chapter 00
`find()`	A method in many ADTs that locates a given element in the collection. `it = v.find("element");`	
graph	A data structure consisting of a collection of edges and vertices. The purpose of a graph is not to store arbitrary user information, but rather to represent the relationship between vertices and edges.	Chapter 13
hash	An abstract data type working much like a set but with better performance characteristics due to leveraging knowledge of the underlying base type.	Chapter 12

hash function	A function returning an index into a hash table from a given value.	Chapter 12
hash table	The most correct use of this term is in reference to the list of buckets in a hash that stores values. However, the term can also be used for the hash ADT itself.	Chapter 12
height	The length of the path from a given node to the root node.	Chapter 08
in-order successor	In a BST, the in-order successor is the node the follows the current node when performing an in-order traverse of the binary tree.	Chapter 09
`insert()`	A method in many ADTs that adds an element to the collection. `v.insert("element");`	Chapter 01
intersection	A binary operation returning the elements that are in both the left-hand-side and the right-hand-side. `intersection = s1 && s2;`	Chapter 05
isolated vertex	A vertex on a graph that has no edges.	Chapter 13
iterator	A class that facilities traversing through a container. `vector <int> :: iterator it;`	Chapter 01
leaf	A binary node with no child nodes. Also, an edge on a graph with only one edge connecting it to the rest of the graph.	Chapter 08 Chapter 13
level	The set of binary nodes in a given binary tree that have the same height.	Chapter 08
node	A data structure containing a single value. This could be used in a binary tree or in a linked list.	Chapter 06 Chapter 08
not	A Boolean operator that returns the opposite of a given value. `cout << !b;`	Chapter 00
O(1)	Algorithm complexity referring to constant performance. Execution time is unrelated to the size of the data-set.	Introduction
O(log n)	Algorithm complexity referring to logarithmic performance.	Introduction
O(n)	Algorithm complexity referring to linear performance. Twice as much data will take twice as much execution time.	Introduction
O(n log n)	Algorithm complexity equivalent to O(n) × O(log n).	Introduction
$O(n^2)$	Algorithm complexity where twice as much data will take four times as much execution time.	Introduction
object	Once instance of a class. Another way of saying this is that an object is a variable whose data-type is the class. The following example, we have a class called `Date`. The variable `today` is an object. `Date today;`	Chapter 00
operand	A parameter to an operator. For example the not (`!`) operator takes one operand while addition (`+`) takes two.	Chapter 00

operator	A function associated with a class or data-type. For example, the assignment operator (`=`) and the equivalence operator (`==`) are two operators associated with most data-types and classes.	Chapter 00
or	A Boolean operator corresponding to the mathematical notion of disjunction. It returns `false` only if both operands are `false`. `b3 = b1 || b2;`	Chapter 00
order	The number of vertices in a graph.	Chapter 13
out-degree	The number of edges pointed away from a vertex.	Chapter 13
mantissa	The component of the scientific notation, E notation, or floating point representation of a number referring to value of a number after the exponent has been removed. Another name for the mantissa is "significand." For example, 1.888×10^3 has a mantissa of 1.888.	Chapter 00
parent node	The binary node connected to at least one other node with a left or right pointer.	Chapter 08
path	The sequence of nodes or vertices one would visit to reach one node from another following pointers or edges.	Chapter 08 Chapter 13
perfect	A binary tree where all leafs are of the same height.	Chapter 08
perfect hash	A hash function that evenly maps elements into a hash table with no collisions and no empty buckets.	Chapter 12
performance	The measure of how long it takes for a given operation to be completed. This can be measured in CPU cycles (such as 12 cycles), in elapsed time (such as 54 ms), or in algorithmic complexity (such as O(n)).	Introduction
`push()`	A method which adds one element to the end of a stack. `s.push(10);`	Chapter 02
`push_back()`	A method which adds one element to the end of a container. `v.push_back(10);`	Chapter 01
recursion	The process of a function calling itself to compute a result. Recursion requires two steps: 1) a mechanism to know when the end is reached, and 2) a mechanism to advance towards the end condition.	Chapter 08
red-black tree	A BST that is self-balancing.	Chapter 09
root	The binary node having a `NULL` parent pointer and from which all other nodes can be reached.	Chapter 08
set	An ADT that stores a collection of values with no repeats.	Chapter 05
siblings	Two binary nodes sharing the same parent node.	Chapter 08
significand	The component of the scientific notation, E notation, or floating point representation of a number referring to value of a number after the exponent has been removed. Another name for the significand is "mantissa." For example, 1.888×10^3 has a significand of 1.888.	Chapter 00

sink	A vertex with an out-degree of zero. In other words, a vertex that points to no other vertex.	Chapter 13
size	The amount of space required to store a data-type or data structure in memory. This is usually measured in bytes.	Introduction
`size()`	A method associated with most containers which returns the number of elements currently stored therein. `cout << v.size();`	Chapter 01
source	A vertex with an in-degree of zero. In other words, a vertex that has no edges directed towards it.	Chapter 13
subtree	The collection of descendant nodes relative to a given reference node	Chapter 08
tree	A collection of connected binary nodes. A tree is a special form of graph where there is exactly on path between any two nodes.	Chapter 08
two's compliment	A way of representing integers in binary.	Chapter 00
union	A binary operation returning the elements that are either in the left-hand-side or the right-hand-side. `union = s1 || s2;`	Chapter 05
`vector`	A container similar to an array except it can resize and grow `vector <int> v;`	Chapter 01
vertex	A single named point on a graph. Other names include node, point, junction, and endpoint.	Chapter 13

APPENDIX N : INDEX

#include 379
[] square bracket 33
aggregation 228
and 13
arc 186
  directed 186
binary search tree 211
binary tree 186
bool 11
bucket 304
call stack 227
canonical form 22
chaining 311
char 19
child 186
class 26
collision 310
correctness 2
degree 325
deque 98
double 21, 23
E notation 22
exponent bias 22
Fibanocci 37
float 21, 23
function template 368
graph 325
hash 305
header file 379
height 186
in-degree 325
in-order predecessor 226
in-order successor 226
instance 26
int 14
integers 14
intersection 120
iterator 371
iterators 30
leaf 186
level 186
linear probing 310
linked list 140
list 163
long 14
long double 21, 23
long long 14
makefile 380
mantissa 22
map 247
namespace 378
nested class 374
node 139, 186
  ancestor 186
  binary 186
  descendant 186
not 13
O(1) 3, 4
O(log n) 3, 5
O(n log n) 3, 7
O(n) 3, 6
O(n2) 3, 8
object 26
open addressing 310
operator 363
  -- decrement 365
  != not equals 366
  [] square bracket 364
  ++ increment 365
  < less than 366
  <= less than or equal 366
  = assignment 363
  == equivalence 366
  > greater than 366
  >= greater than or equal 366
or 13
order 325
out-degree 325
parent 186
path 186, 325
perfect hash 315
performance 3
pseudocode 354
push_back 27
queue 77
recursion 227
root 186
set 116
short 14
sibling 186
significand 22
size 2
sort 267
  binary 279
  bubble 268
  heap 291
  insertion 275
  merge 282

quick....287
selection....272
stack....46
subtree....186
template prefix....368, 369
trailing significant field....22
tree....186
balanced....186
binary....186
complete....186
perfect....186
two's compliment....15
type parameter....368, 369
UML class diagram....356
union....120
vector....25
vertex....325
isolated....325
leaf....325
sink....325
source....325