CSE 100 Notes

Advanced Data Structures

Fall 2021

Taught by Professor Niema Moshiri

Table of Contents

1 A Brief Introduction

In this course, we will primarily be building off of our prior knowledge of data structures (CSE 12). In particular, we will:

- Analyze data structures for both time and space complexity.
- Describe the strengths and weaknesses of a data structure.
- Implement complex data structures correctly and efficiently.

1.1 Data Structures vs. Abstract Data Types

When talking about data, we often hear about data structures and abstract data types.

Data Structures (DS)	**Abstract Data Type** (ADT)
Data structures are collections that contain: • Data values. • Relationships among the data. • Operations applied to the data. It also describes how the data are organized and how tasks are performed. So, a data structure defines every single detail about anything relating to the data.	Abstract data types are defined primarily by its behavior from the view of the user. So, not necessarily how the operations are done, but rather what operations it must have from a completely abstract point of view. Specifically, it describes only what needs to be done, not how it's done.

Consider the `ArrayList` (DS) vs. the `List` (ADT).

- A `List` will most likely have the following operations:
  - `add`: Adds an element to the list.
  - `find`: Does an element exist in the list?
  - `remove`: Remove an element from the list.
  - `size`: How many elements are in this list?
  - `ordered`: Each element should be ordered in the way we added it. For example, if we added `5`, and *then* added `3`, and *then* added `10`, our list should look like: `[5, 3, 10]`.

  Of course, as an abstract data type, `List` isn't going to define how these operations work. It just lists all operations that any implementing data structure must have. In other words, we can think of `List`, or any abstract data type, as a *blueprint* for future data structures.
- An `ArrayList` is simply an array that is expandable. It is internally backed by an array. So, we can perform the following operations:
  - We can `add` an element to the `ArrayList`. In this case, we add the element to the next available slot in the array, expanding the array if necessary.
  - We can `find` an element in the `ArrayList`. In this case, we can search through each slot of the array until we find the array or we reach the end of the array.
  - We can `remove` an element from the `ArrayList`. In this case, we can simply move every element after the specified element back one slot.
  - We can get the `size` of the `ArrayList`. In this case, this is as simple as seeing how many elements are in this `ArrayList`.
  - And, we know that the `ArrayList` is `ordered`. In this case, this is already done via the `add` and `remove` methods.

Notice how `ArrayList` specifies how each operation defined by `List` works. In this sense, we say that `ArrayList` essentially implements `List` because we need to define *how* the tasks defined by `List` are performed.

So, the key takeaways are:

- An abstract data type (in our case, `List`) specifies what needs to be done without specifying how it's done.
- A data structure (in our case, `ArrayList`) actually defines **how** the data is organized, how the different operations are performed, and how exactly everything is represented.

2 Introduction to C++

Here, we will talk about C++, the programming language that we will use in this course.

2.1 Data Types

First, we'll compare the data types in Java and C++.

Data Type	Java	C++
`byte`	1 byte	1 byte
`short`	2 bytes	2 bytes
`int`	4 bytes	4 bytes
`long`	8 bytes	8 bytes
`long long`		16 bytes
`float`	4 bytes	`4 bytes`
`double`	8 bytes	`8 bytes`
`boolean`	Usually 1 byte	
`bool`		Usually 1 byte
`char`	2 bytes	1 byte

It should be mentioned that:

- In Java, you can only have signed data types.
- In C++, you can have both signed and unsigned data types.
- `boolean` (Java) and `bool` (C++) are effectively the same thing: they represent either `true` or `false`.

2.2 Strings

There are some major differences between strings in Java and C++, which we will discuss below.

2.2.1 Representation

In Java, strings are represented by the `String` class. In C++, strings are represented by the `string` type.

2.2.2 Mutability

Strings in Java are immutable. The moment you create a string, you won't be able to modify them. The only way to change a string variable is by creating a new string and reassigning them.

In C++, strings are actually mutable. You can modify strings in-place.

2.2.3 Concatenation

In Java, you can concatenate any type to a string. For example, the following is valid:

```
String a = "this is a string" + 123;
```

In C++, you can only concatenate strings with other strings. So, if you wanted to convert an integer (or any other type) to a string, you would have to *first* convert that integer to a string (or use a string stream).

2.2.4 Substring Method

In Java, we can take the substring of a string using the `substring` method. The method signature is:

```
String#substring(beginIndex, endIndex);
```

In C++, we can take the substring using the `substr` method. The method signature is:

```
string#substr(beginIndex, length);
```

An important distinction to make here is that Java's `substring` method takes in an **end index** for the second parameter, whereas C++'s `substr` method takes in a **length** for the second parameter.

2.3 Comparing Non-Primitive Objects

Suppose `a` and `b` are two non-primitive objects.

In Java, if we want to compare these two objects, we have to make use of the methods:

```
a.equals(b)
a.compareTo(b)
```

If we tried using the relational operators like `==` or `!=`, Java would compare the memory addresses of the two objects, which is often something that we aren't looking for.

In C++, even if `a` and `b` are objects, we can make use of the relational operators:

```
a == b      a != b
a < b       a <= b
a > b       a >= b
```

This is done through something called **operator overloading**, where we write a custom class and define how these operators should function.

2.4 Variables

Now, we will briefly discuss how variables function in both C++ and Java.

2.4.1 Initialization

In Java, variable initialization is **checked**. Consider the following code:

```
int fast;
int furious;
int fastFurious = fast + furious;
```

Because `fast` and `furious` aren't initialized, the Java compiler will throw a compilation error.

In C++, variable initialization is **not checked**. Consider the same code, which will compile:

```
int fast;
int furious;
int fastFurious = fast + furious;
```

Here, this would result in **undefined** behavior.

2.4.2 Narrowing

In Java, if we have a higher variable type and then try to cast this type to a smaller type, we would get a compilation error. Consider the following code:

```
int x = 40_000;
short y = x;
```

This code would result in a compilation error. If we didn't want a compilation error, we would have to explicitly *cast* the bigger variable type to the smaller type. The following Java code would compile just fine:

```
int x = 40_000;
short y = (short) x;
```

In C++, no compilation error would occur; that is, the following code would compile:

```
int x = 40_000;
short y = x;
```

What would actually happen is that `x` would get **truncated** when it is assigned to `y`, resulting in integer overflow.

2.4.3 Variable Declaration

In Java, variables **cannot** be declared outside of a class. The following Java code would result in a compile error:

```
// MyClass.java

int meaningOfLife = 42;
class MyClass {
    // some code
}
```

In order for this to compile, you have to put variable declarations inside the class space (as an instance variable) or in a method inside a class (as a local variable).

In C++, variables **can** be declared outside of a class. The following C++ code would compile completely fine:

```
// MyClass.cpp

int meaningOfLife = 42;
class MyClass {
    // some code
}
```

Here, `meaningOfLife` is a **global variable**. Anything in this file can access this variable. In general, it is considered poor practice to use global variables except in cases of constants.

2.5 Classes, Source Code, and Headers

Another thing that is important is the concept of classes (which leads to the topic of object-oriented programming). That being said, Java and C++ has some differences with regards to how classes function.

2.5.1 Class Declaration

There are some key differences in how methods and instance variables are laid out in Java and C++. In Java, a typical class would look like:

```
class Student {
    public static int numStudents = 0;
    private String name;

    public Student(String n) { /* Code */ }

    public void setName(String n) { /* Code */ }
    public String getName() { /* Code */ }
}
```

And in C++, a typical class would look like:

```
class Student {
    public:
        static int numStudents;

        Student(string n);

        void setName(string n);
        string getName() const;

    private:
        string name;
}

int Student::numStudents = 0;
Student::Student(string n) { /* Code */ }
void Student::setName(string n) { /* Code */ }
string Student::getName() const { /* Code */ }
```

There are several notable differences:

- **Modifiers:** In Java, if you want your method or instance variable to have an access modifier, you explicitly state the access modifier. In C++, you have a region for your access modifier. That is, there is a `public` region, `private` region, etc. Any methods or instance variables listed under these regions will take on that access modifier. For instance, `setName` is in the `public` region, so `setName` is public.
- **Implementation:** In Java, directly after declaring a method or constructor in a class, we need to provide the implementation code. In C++, we can "declare" the methods and the constructor, and then outside of the class we can implement the methods.

Now, consider the following C++ code:

```
class Point {
    private:
        int x;
        int y;

    public:
        Point(int i, int j);
}

Point::Point(int i, int j) {
```

```
    x = i;
    y = j;
}
```

Here, we're initializing the `x` and `y` instance variables directly from the constructor implementation. However, we can initialize these instance variables directly like so:

```
class Point {
    private:
        int x;
        int y;

    public:
        Point(int i, int j);
}

Point::Point(int i, int j) : x(i), y(j) {}
```

This is called the **member initializer list**.

2.5.2 Source vs. Header Files

Consider the following class:

```
class Student {
    public:
        static int numStudents;
        Student(string n);

    private:
        string name;
}

int Student::numStudents = 0;
Student::Student(string n) : name(n) {
    numStudents++;
}
```

We can choose to break this up into two separate files; a **source** (usually `.cpp`) file and a **header** (usually `.h`) file. The header file contains the class and the method *declaration*; the source file contains the implementations for those methods. So, the above code can be written like so:

```
// The header file
// Student.h
class Student {
    public:
        static int numStudents;
        Student(string n);

    private:
        string name;
}

// The source file
// Student.cpp
int Student::numStudents = 0;
Student::Student(string n) : name(n) {
```

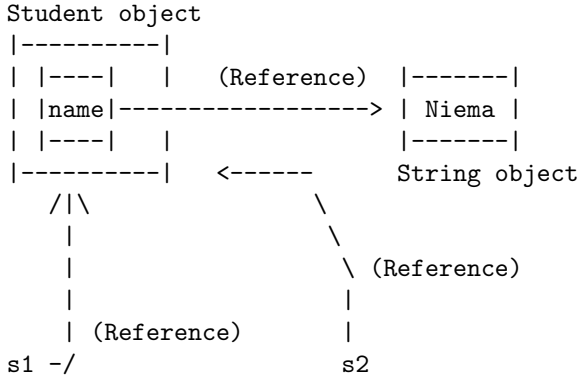
```
    numStudents++;
}
```

2.6 Memory Diagrams

Consider the following Java code:

```
Student s1 = new Student("Niema");
Student s2 = s1;
```

Here, `s1` is a *reference* to a `Student` object. This `Student` object contains a *reference* to a `string` object with the content `Niema`. That is:

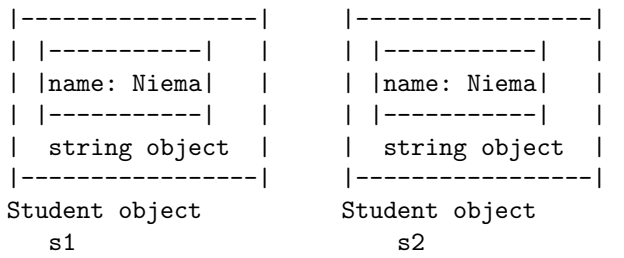


It also follows that `s2` is a reference to the same object that `s1` is referring to.

Now, consider the following C++ code:

```
Student s1("Niema");
Student s2 = s1;
```

Here, `s1` is a `Student` *object*. The `Student` object contains a `string` object with the content `Niema`. That is:



Additionally, when we assign `s1` to `s2`, we actually make a copy of said object. So, `s2` is its own object; it does not share a reference with `s1`.

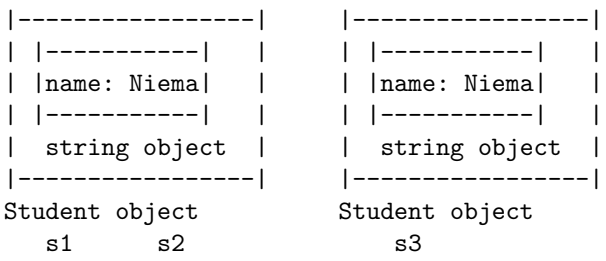
In other words, in Java, `s1` and `s2` are both references to the same object; in C++, `s1` *is* the object and `s2` is *another* object.

2.6.1 References

Consider the following C++ code:

```
Student s1 = Student("Niema");
Student & s2 = s1;
Student s3 = s2;
```

The memory diagram looks like this:



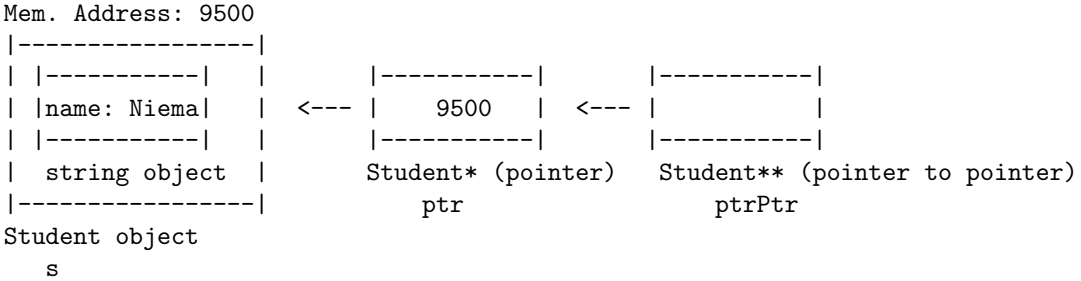
Here, `s2` can be seen as *another* way to call `s1` (think of `s2` as another name for `s1`). `s3` would be a copy of `s1`.

2.6.2 Pointers

Pointers are similar to Java references. Consider the following C++ code:

```
Student s = Student("Niema");
// * in this case means pointer
// & means memory address
// So, ptr stores a memory address to some object. In other words,
// it points to the object s.
Student* ptr = &s;
Student** ptrPtr = &ptr;
```

The memory diagram would look like:



If we wanted to access an object through a pointer, we can do this in several ways.

1. Deferencing a pointer.

   ```
   // * in this case dereferences the pointer
   // Think of the * as following the arrow
   (*ptr).name;
   ```

2. Arrow dereferencing.

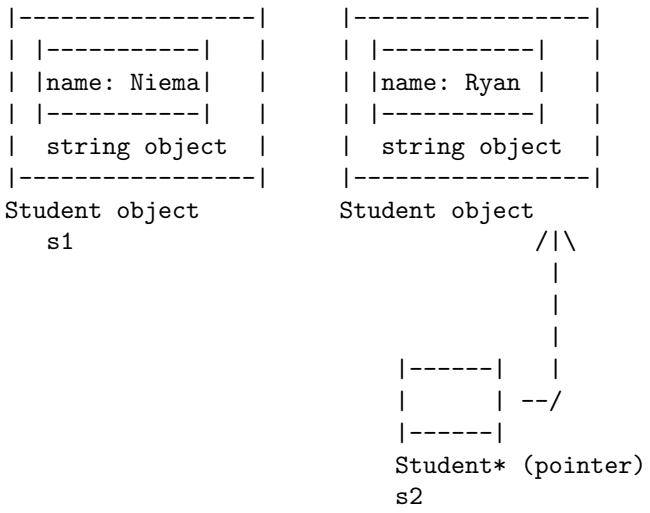
   ```
   // ptr->x is the same thing as (*ptr).x
   ptr->name;
   ```

2.6.3 Memory Management

Consider the following C++ code:

```
Student s1 = Student("Niema");
Student* s2 = new Student("Ryan");
```

The corresponding memory diagram is:



Here, `s1` is allocated on the *stack*; once the method returns, `s1` is automatically destroyed.

`s2` is allocated through the `new` keyword. This is known as dynamic memory allocation. So, `s2` is a pointer to the newly allocated memory. Because this object was created using the `new` keyword, we need to deallocate it ourselves. To do so, we need to explicitly call `delete` on this object:

```
delete s2;
```

`delete` takes in a memory address (i.e. pointer). This is very similar to `free` (in C). If we don't free this, we run into what is called a **memory leak**.

2.7 Constant Keyword

In C++, the `const` keyword means that the variable can never be reassigned. Consider the following:

```
const int a = 42;
int const b = 42;
```

If we tried reassigning `a` (e.g. `a = 41;`), we would get a compiler error.

The second line (`int const`) is identical to the first line.

2.7.1 `const` and Pointers

Consider the following C++ code:

```
int a = 42;                     // a
const int* ptr1 = &a;           // b
int const* ptr2 = &a;           // c
int* const ptr3 = &a;           // d
const int* const ptr4 = &a;     // e
```

- For lines (b) and (c), the pointer cannot modify the object that it is pointing to. But, we can reassign the pointer to point to a different object.
- For line (d), we cannot reassign the pointer to point to a different object. However, we can modify the object that the pointer is pointing to.

- For (e), we cannot reassign the pointer to point to a different object *or* modify the object that the pointer is pointing to.

In general:

```
const type* const varName = ...;
----------- -------------
   (a)            (b)
```

- Segment (A): The `const` next to `type*` means that we cannot modify the object or value behind the pointer.
- Segment (B): The `const` next to `varName` (the variable name) means that we cannot reassign the pointer to point to a different object or value.

2.7.2 `const` and References

Suppose we have the following C++ code:

```
int a = 42;
const int & ref1 = a;       // a
int const & ref2 = a;       // b
```

- In (a), the `const` means that we cannot modify the variable through the constant reference. So:

```
a = 21;         // Allowed.
ref1 = 20;      // Compile error!
```

- (b) is the same exact thing is (a).

2.7.3 `const` Functions

Recall the `Student` class from earlier:

```
class Student {
    public:
        Student(string n);
        string getName() const;

    private:
        string name;
}

Student::Student(string n) : name(n) {}
string Student::getName() const {
    return name;
}
```

What does the `const` in `getName()` do? Well, the `const` keyword after the function declaration means that the function cannot modify *this* object. So:

- You cannot do any assignments to instance variables.
- You can only call other `const` functions.

So, effectively, `const` after a function name means that we are guaranteeing that we aren't changing the object's state in any way.

2.8 Functions

In C++, we can have global functions (functions that are defined outside of classes). For instance, the main method (shown below) is a global function (and is required to be):

```
int main() {
    /* Do stuff */
}

class MyClass {
    /* Some code */
}
```

2.8.1 Passing by Value vs. Reference

In C++, you can pass parameters either by value or reference.

When passing by value, the function makes a **copy** of the values that you passed in. Some example code is shown below:

```
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

These copies are destroyed once the function returns (the stack frame is destroyed).

When passing by reference, the function takes in *references* to the variables. Some example code is shown below:

```
void swap(int & a, int & b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Effectively, whatever you change with the references will be reflected with the actual variables. So, in the above `swap` method, `a` and `b` will be updated after the function is done.

2.9 Vectors

A C++ `vector` is very similar in nature to Java's `ArrayList` class and arrays. Consider the following code, which demonstrates some common operations:

```
// Creates a new vector.
vector<int> a;
// Adds 42 to end of vector. Looks like: [42]
a.push_back(42);
// Adds 21 to end of vector. Looks like: [42, 21]
a.push_back(21);
// Removes 21 from vector. Looks like [42]
a.pop_back(); // returns 21
// We can access the first element (0th index).
a[0];
```

Like Java arrays or `ArrayList`, elements in a C++ vector are stored contiguously; that is, they are stored after the previous element.

We know that if we assign an object to another variable, the other variable will get a full copy of that object. The same applies to vectors; we can also create a copy of a vector simply be reassigning it:

```
vector<int> a;
a.push_back(42);
vector<int> b = a;
// a: [42]
// b: [42]
```

2.10 Input and Output

Consider the following code:

```
int n;
cout << "Enter a number: ";
cin >> n;

string message;
cout << "Enter a message: ";
getline(cin, message);

if (cin.fail()) {
    cerr << "Bad input!" << endl;
}
```

Here:

- `cin` represents standard input (`stdin`).
- `cout` represents standard output (`stdout`).
- `cerr` represents standard error (`stderr`).

In C++, we can use `istream` to handle input stream and `ostream` to handle output stream. `cin` is an example of an `istream`; `cout` is an example of an `ostream`.

We can make use of the overloaded `<<` and `>>` operators to write to standard output and read from standard input, respectively. So:

- `cout << "Enter a number"` effectively means to write this message to standard output.
- `cin >> n` effectively means to read from the standard input and store that input into `n`. We aren't necessarily restricted to `int`; we could use `long`, `double`, `string`, etc.
- We can also use `getline` to read from standard input and then store the result into a variable. In our example above, we called `getline(cin, message)`. `cin` is where we are reading the input from and `message` is the variable where we store the result of reading from `cin`.
- `endl` means `end` `l`ine and, in our use case here, writes a new line to standard error. In reality, we can use `endl` to write a newline to standard output or error.

2.11 Templates

Templates introduce the notion of *generic programming*. Consider the following code in Java:

```
class Node<Data> {
    public final Data data;
    public Node(Data d) {
        data = d;
    }
}

Node<String> a = new Node<String>(s);
Node<Integer> a = new Node<Integer>(s);
```

The generic type is `Data` (though you can rename it to whatever you want). We can use this type either as a parameter type or a return type. When creating a new object with a generic type, we simply put the type between the `<>` (like with the `Node` examples).

Consider the equivalent C++ example:

```
template<typename Data>
class Node {
    public:
        Data const data;
        Node(const Data & d) : data(d) {}
}

Node<string> a(s);
Node<int> b(n);
```

Here, we can use templates to achieve similar results (compared to the Java example). Functionality-wise, this is similar to Java.

2.12 Iterators

Consider the following C++ code:

```
for (string name : names) {
    cout << name << endl;
}
```

What is `names`?

- Is it a `vector`?
- Is it a `set`?
- Is it an `unordered_set`?
- Is it another collection that C++ has?

Well, it doesn't matter! Regardless of what collection we are using, how we use it doesn't matter when it comes to iterating over it. This functionality is made possible by something called **iterators**.

2.12.1 Iterating Over Arrays

Consider the following code:

```
void printInorder(int* p, int size) {
    for (int i = 0; i < size; ++i) {
        cout << *p << endl;
        ++p;
    }
}
```

The `*p` deferences the pointer, giving the value at the location that the pointer is pointing to.

The `++p` is an example of pointer arithmetic; this will add whatever the size of the type is to the pointer. In this case, this will make the pointer point to the memory address of the next element in the array.

Here, we know that `p` is (initially) a pointer to the first element in the array:

```
 0   4   8   12  16  20  24  28          Memory Address
 ------------------------------          (sizeof(int) = 4)
[10, 20, 25, 30, 46, 50, 55, 60]         Array
 ^
 p                                       Pointer
```

Dereferencing `p` (`*p`) gives us `10`.

When we do `++p`, we made the pointer point to the next memory address:

```
0   4   8   12  16  20  24  28          Memory Address
------------------------------
[10, 20, 25, 30, 46, 50, 55, 60]         Array
     ^
     p                                   Pointer
```

Dereferencing `p` (`*p`) gives us `20`.

2.12.2 Using Iterators

Consider the following C++ code:

```
vector<string> names;
// populate with data

vector<string>::iterator itr = names.begin();
vector<string>::iterator end = names.end();

while (itr != end) {
    cout << *itr << endl;
    ++itr;
}
```

Here, we note a few things.

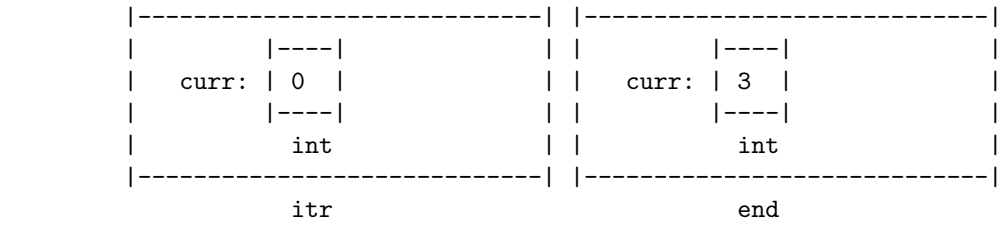
- `iterator` is simply a class that handles, well, iteration. So, `itr` and `end` are instances of the `iterator` class that is iterating over `names`.
- The `!=` operator (in `itr != end`) has been overloaded. This checks the `curr` property in the `iterator` class to see if it is equal (or, more specifically, not equal) to the `curr` property of the other index. In this case, `itr != end` is effectively comparing `itr.curr` with `itr.end`.

- The * dereferencing operator (in *itr) has also been overloaded. This operator has been overloaded to return whatever the value is at the curr index. So, in our case,*itr would return whatever value is at the specified curr index in the array that we are iterating through.
- The ++ operator (in ++itr) is also overloaded. This will increment the curr property in the iterator instance.

Suppose names has the following:

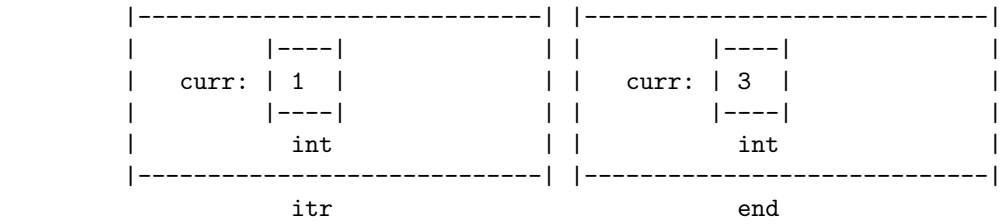
```
   0       1       2           // Index
["Niema", "Ryan", "Felix"]     // names array
```

Essentially, vector<string>::iterator will look something like:



Calling *itr will basically give us names[curr] (or, more specifically, names[0]). Comparing itr != end is basically the same as checking 0 != 3.

When we call ++itr, we now have:



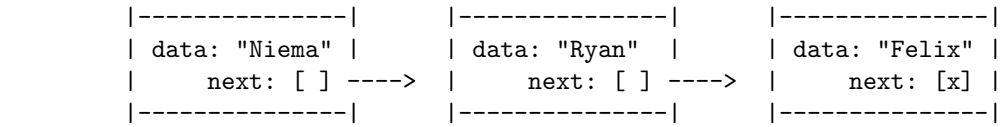
Calling *itr now will basically give us names[curr] (or, more specifically, names[1]). Comparing itr != end is basically the same as checking 1 != 3.

2.12.3 Linked List Iterator

Consider the following code, which is essentially the same code as the previous one:

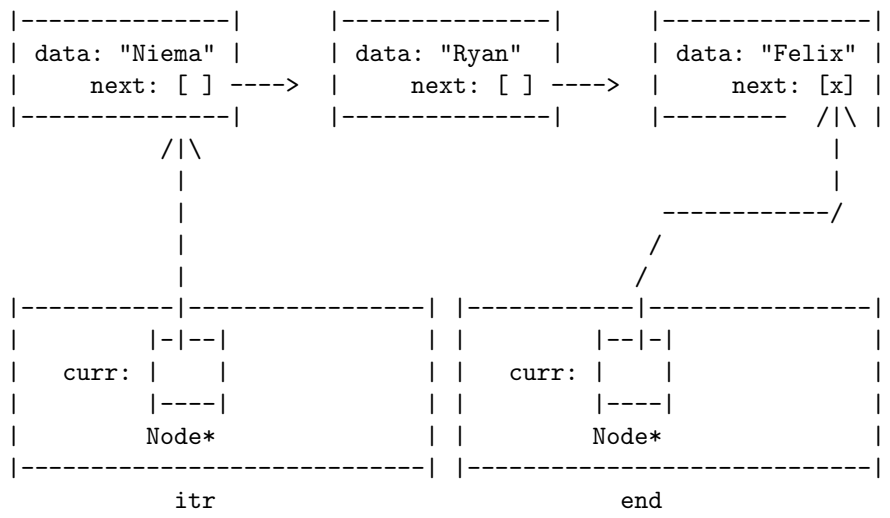
```
LinkedList<string> names;
// populate with data

LinkedList<string>::iterator itr = names.begin();
LinkedList<string>::iterator end = names.end();

while (itr != end) {
    cout << *itr << endl;
    ++itr;
}
```

The only difference is that we're now using a LinkedList instead of vector. However, the way the data is structured is very different. Suppose names has the following:



Here, `[x]` (in the `next` property of the last node) is a `nullptr`.

How does using nodes change our iterator? Well, `LinkedList<string>::iterator` will look something like:



Going back to the code:

```
while (itr != end) {
    cout << *itr << endl;
    ++itr;
}
```

It should be noted that:

- `!=` is once again overloaded to compare the values of the node's `data`.
- `*itr` is once again overloaded to return `data`. It would look like:

```
return curr->data;
```

- `++itr` is once again overloaded to make the iterator move to the next node. This would look like:

```
curr = curr->next;
```

2.12.4 Creating an Iterator Class

When creating data structures, we'll often need to create our own Iterator classes.

First, we'll talk about the operators associated with the iterator class:

- `==`: `true` if the iterators are pointing to the same item and `false` otherwise.
- `!=`: `true` if the iterators are pointing to the different item and `false` otherwise.
- `*` (dereference): Return a reference to the current data value.
- `++` (pre- and post-increment): Move the iterator to the next item.

And, we also need to talk about what functions are in the data structure class so we can make use of the iterator:

- `begin()`: Returns an iterator to the first element.
- `end()`: Returns an iterator to the element just after the last element (not the last element, but *after* the last element).

So, in the Linked List example:

```
[] -> [] -> []
^              ^
begin()        end()
```

And in any array-based structures:

```
[a, b, c, d, e]
 ^              ^
 begin()        end()
```

3 Time and Space Complexity

One of the key things computer scientists try to do is automate competitive tasks, and of course, that requires *performance*. So, that begs the question: how can we measure the performance of our program?

- How many hours does it take to run?
- Minutes?
- Nanoseconds?

These are all metrics of *human time*. However, a program has two aspects:

- The implementation.
- The algorithm behind that program.

While these different metrics of human time are good at measuring the actual implementation of a program, they don't do a good job describing how fast the *idea*, the algorithm itself, is. For instance, running the algorithm on two different devices, both which have wildly different hardware, will result in a significant difference in how fast your algorithm runs.

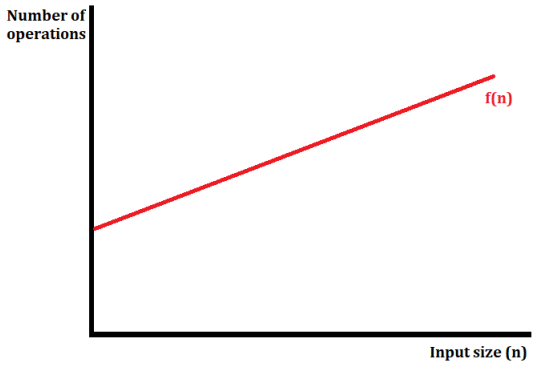
That being said, we want to know how fast an algorithm is. The best way to do so is by figuring out the performance in terms of number of operations with respect to the input size n (instead of the amount of time).

3.1 Notation of Complexity

Consider the following notations:

- Big-O: Upper bound.
- Big-Ω: Lower bound.
- Big-θ: Both upper and lower bound.

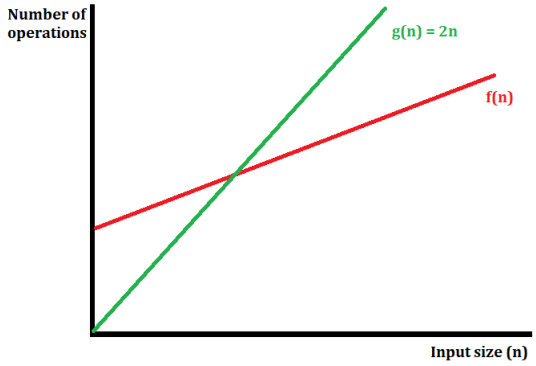
Consider the following graph:



Where $f(n)$ describes the number of operations of your algorithm for some n.

- We say that $f(n)$ is $O(g(n))$ if, for some constant a, we have $a * g(n) \geq f(n)$ as $n \to \infty$.

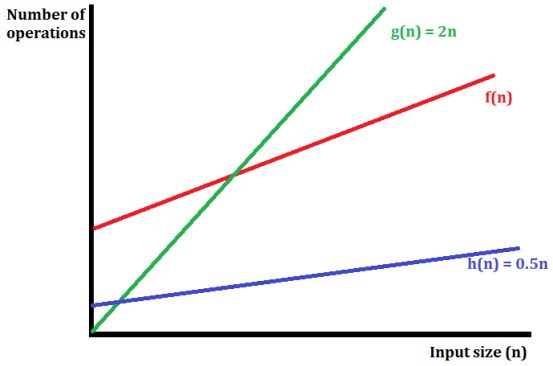
  Consider the following graph:



Here, we see that the intersection of the red and the green line occurs at some point, and that after that point the green line will always be greater than the red line. In other words, at that point, we can say that $f(n)$ will never be bigger than $g(n)$ beyond that point. Therefore, we say that $f(n)$ is $O(2n)$, or simply $O(n)$.

- Big-Ω works similarly. We say that $f(n)$ is $\Omega(g(n))$ if, for some constant b, $b * g(n) \leq f(n)$ as $n \to \infty$.

  Consider the following graph:



  Here, we see that the blue line h is strictly lower than the red line. In other words, $f(n)$ will never be smaller than $h(n)$. Therefore, we say that $f(n)$ is $\Omega(0.5n)$, or simply $\Omega(n)$.

- We say that $f(n)$ is $\Omega(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$. Mathematically:

$$b * g(n) \leq f(n) \leq ag(n)$$

  In the graphs above, we already found the b and a constants. So:

$$0.5n \leq f(n) \leq 2n$$

  Therefore, we cn say $f(n)$ is $\Omega(n)$.

Remarks:

- Your bigger or smaller functions do not need to be strictly (i.e. always) bigger or smaller than your $f(n)$. They just need to be strictly bigger or smaller beyond some n.
- We will almost always use Big-O.