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) {
```

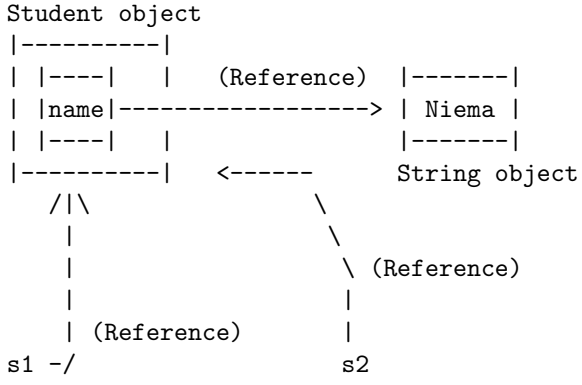
```
    numStudents++;
}
```

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:

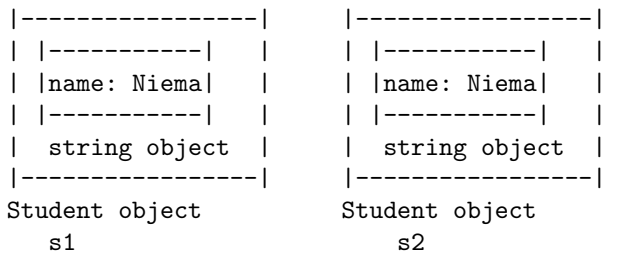


It also follows that `s2` is a reference to the same object that `s1` is referring to.

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`.

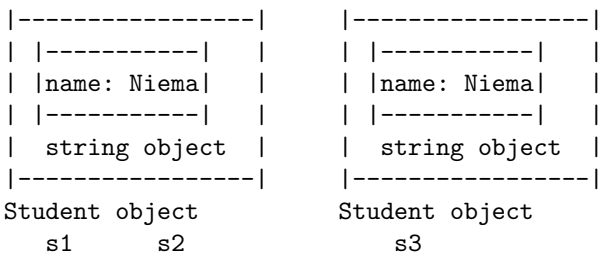
In other words, in Java, `s1` and `s2` are both references to the same object; in C++, `s1` *is* the object and `s2` is *another* object.

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:



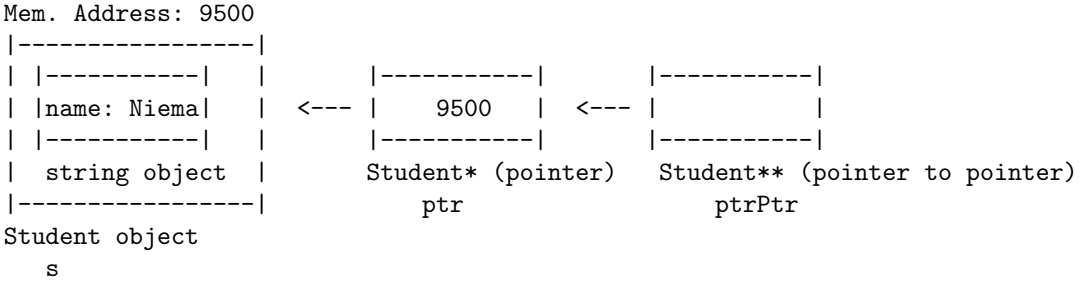
Here, `s2` can be seen as *another* way to call `s1` (think of `s2` as another name for `s1`). `s3` would be a copy of `s1`.

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.

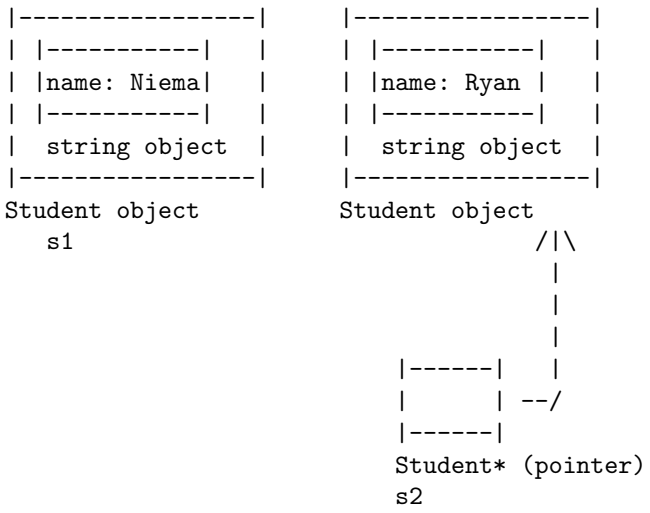
   ```
   // ptr->x is the same thing as (*ptr).x
   ptr->name;
   ```

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:

```
|-----------------------------| |-----------------------------|
|         |----|              | |         |----|              |
|   curr: | 0  |              | |   curr: | 3  |              |
|         |----|              | |         |----|              |
|           int               | |           int               |
|-----------------------------| |-----------------------------|
            itr                               end
```

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:

```
|-----------------------------| |-----------------------------|
|         |----|              | |         |----|              |
|   curr: | 1  |              | |   curr: | 3  |              |
|         |----|              | |         |----|              |
|           int               | |           int               |
|-----------------------------| |-----------------------------|
            itr                               end
```

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:

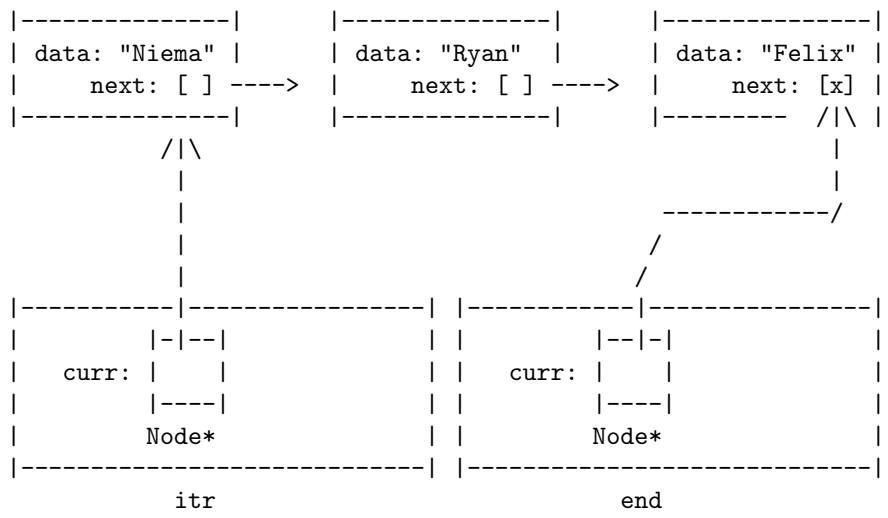
```
LinkedList<string> names;
// populate with data

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:

```
|---------------|      |---------------|      |---------------|
| data: "Niema" |      | data: "Ryan"  |      | data: "Felix" |
|     next: [ ] ----> |     next: [ ] ----> |     next: [x] |
|---------------|      |---------------|      |---------------|
```

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.

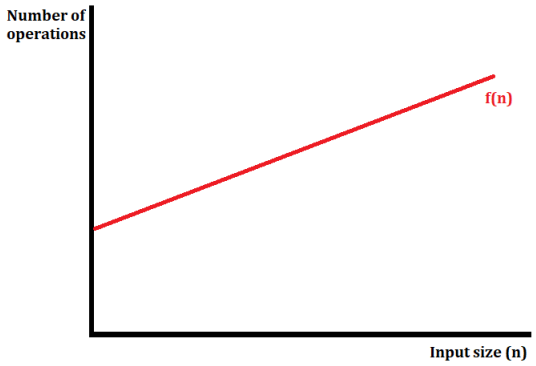
That being said, we want to know how fast an algorithm is. The best way to do so is by figuring out the performance in terms of number of operations with respect to the input size n (instead of the amount of time).

3.1 Notation of Complexity

Consider the following notations:

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

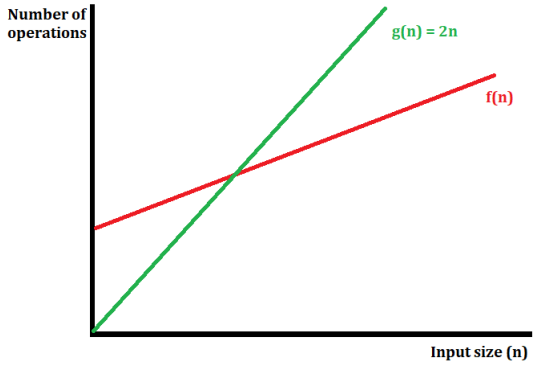
Consider the following graph:



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 $n \to \infty$.

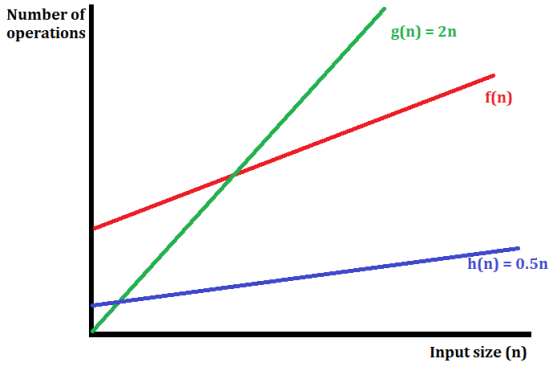
  Consider the following graph:



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 $\theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$. Mathematically:

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

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

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

  Therefore, we can say $f(n)$ is $\theta(n)$.

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.

3.2 Finding Big-O Time Complexity

Given some algorithm, how do we find the Big-O time complexity of it?

1) Determine $f(n)$, or the number of operations our algorithm performs to solve an input of size n.

2) Drop all lower terms of n. In other words, we're only interested in the highest term of n.

3) Drop the constant coefficient.

3.2.1 Example: Grades

Consider the following easy example.

- Input: List of n students, like so:

```
|---------|---------|---------|
| Niema   | Ryan    | Felix   |
| A+      | A       | A       |
|---------|---------|---------|
```

- Algorithm:

```
Print length of the list (n)
For each student x in the list:
    Print x's name
    Print x's grade
```

- Output:

```
3
Niema   A+
Ryan    A
Felix   A
```

Here, we note a few things:

- Regardless of what the length of the list is, printing the length of the list n is a constant time operation. So, this is exactly 1 operation.
- For one student, we print the student's name and grade. Both of these are 1 operation each, for a total of 2 operations. So, for each student, we need to do 2 operations. Therefore, we need to do $2n$ operations for the loop.
- So, the number of operations that we need to do is:

$$f(n) = 2n + 1$$

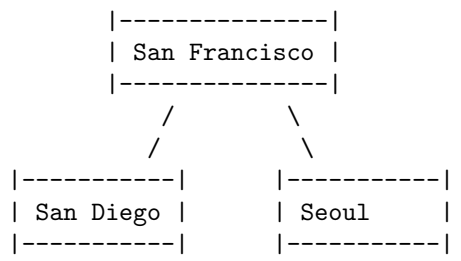
- Of course, we need to drop the lowest terms and the coefficient of the remaining term:

$$\boxed{f(n) = n}$$

3.2.2 Example: Flight Network

Consider the following medium example.

- Input: Flight network of n cities, like so:



  Here, we have a direct flight from San Diego to San Francisco and a direct flight from Seoul to San Francisco. We do not have a direct flight from San Diego to Seoul.

- Algorithm:

```
// Header line = sets up some file
Print header line.
For each unique pair of cities:
    Print the city names
    Print whether or not a direct flight exists
```

- Output:

```
city1, city2, direct
San Diego, San Francisco, yes
San Francisco, Seoul, yes
San Diego, Seoul, no
```

Here, we note a few things again:

- When printing the header line, we need to print three things: `city 1`, `city 2`, and `direct`. Thus, there are 3 operations that are involved.
- For one pair of cities, we need to print three things: the first city name in the pair; the second city name in the pair; and whether or not there is a direct flight between the two cities.

  So, for each pair of cities, there are three operations. We also know that there are $\binom{n}{2}$ ways to get every possible pair of cities. Therefore, we need to do $3\binom{n}{2} = 3\frac{n(n-1)}{2} = 3\left(\frac{n^2}{2} - \frac{n}{2}\right)$ operations for the loop.

- So, the number of operations that we need to do is:

$$f(n) = \frac{3n^2}{2} - \frac{3n}{2} + 3$$

- And, finally, we can drop the lower terms and the coefficient of the remaining term.

$$f(n) = n^2$$

3.2.3 Example: Loops

Consider the following hard example.

- Algorithm:

```
1:  void foo(unsigned int n) {
2:      unsigned int count = 0;
3:      while (n > 0) {
4:          for (int i = 0; i < n; ++i) {
5:              cout << ++count << endl;
6:          }
7:          n /= 2;
8:      }
9:  }
```

And, as usual, the notes:

- This is considered a hard example because the inner loop depends on the outer loop. In the previous examples, we only had to consider one loop and whatever was in this loop.
- As usual, in the `count` declaration (line 2), we only have one operation.
- Before we discuss the two loops, let's first consider the following notes:
  - Let's assume that the print statement is one operation.
  - Whatever `n` is in the while loop, we will iterate from 0 to `n` in the inner for-loop.
  - The division operator (line 7) is considered to be one operation.

  We now consider the actual loops:
  - Whatever n currently is in the while loop, we're going to iterate n times (0 to $n-1$) in the for loop.
  - After the for loop, we do one additional operation for division.

  We just need to figure out how many times the inner loop is iterating overall across all iterations of the while loop, that'll be our answer. So, whatever `n` currently is, the inner loop will be iterating `n` times. In this sense, we only really need to consider all cases of `n` that the inner loop will encounter.

  In our first iteration, the inner loop will iterate n times. The next iteration will iterate $\frac{n}{2}$ times. The next iteration will iterate $\frac{n}{4}$ times, and so on. Essentially, for the inner loop:

$$\overbrace{n}^{\text{1st iteration}} + \underbrace{\frac{n}{2}}_{\text{2nd iteration}} + \overbrace{\frac{n}{4}}^{\text{3rd iteration}} + \underbrace{\frac{n}{8}}_{\text{4th iteration}} + \cdots + 4 + 2 + 1$$

  If we pull out n, we have:

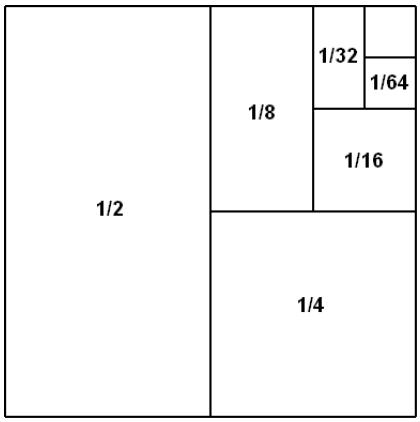
$$n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots\right)$$

  Now, we need to figure out what the sum of everything in the parenthesis. Effectively, we note that we're working with:

$$n\left(1 + \sum_{n=1}^{\infty}\left(\frac{1}{2}\right)^n\right) = 2$$

  Where the summation was evaluated due to the infinite geometric sum formula $S_\infty = \frac{a_1}{1-r}$.

For a better visualization of the summation, we note that the summation can be represented by[1]:



- So, the number of operations is:
$$f(n) = 2n$$
- Taking out the constant, we have:
$$f(n) = n$$

  So, this algorithm runs in $O(n)$ time.

Remark: It's important to not jump straight to conclusions. Most people (when they saw this algorithm) would have assumed an $O(n^2)$ or $O(n \log(n)$ algorithm.

[1]Taken from Wikipedia

3.3 Common Big-O Time Complexity

It's good to know of some Big-O time complexities.

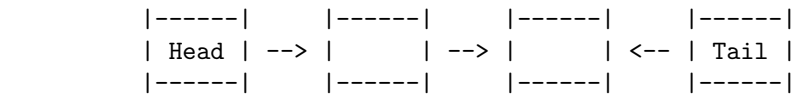
Big-O	Common Name	Notes
$O(1)$	Constant	The time complexity does not depend on the input size n.
$O(\log n)$	Logarithmic	If the input size is doubled, the number of operations is increased by a constant. One common example is binary search: if we have 8 elements, it would take 3 operations. Doubling the number of elements would result in 4 operations. Also, it does not matter what base the logarithmic function is.
$O(n)$	Linear	Your algorithm scales with the number of elements linearly. For example, twice as many elements roughly means twice as slow.
$O(n \log n)$		
$O(n^2)$	Quadratic	If the input size is doubled, we'll have quadruple the amount of elements.
$O(n^3)$	Cubic	Similarly to quadratic or linear, if the input size is doubled, the number of operations are multiplied by 8.
$O(n^a)$	Polynomial	For some constant a, this is known as polynomial time. All of the Big-O time complexities above are considered polynomial. Anything that is upper-bounded by $O(n^a)$ is called polynomial.
$O(k^n)$	Exponential	For some constant k.
$O(n!)$	Factorial	

Remark: Anything algorithm that runs in polynomial time is considered "good." Exponential and factorial time complexities are considered "bad."

3.4 Space Complexity

We can also describes algorithms by space complexity – how much space does an algorithm need for some input size n? Just like time complexity, we often use Big-O notation.

Consider a singly linked list:



Suppose it takes k bytes to store a node. If we have n nodes, it would take roughly $k \cdot n$ bytes to represent all of these nodes. We will always have one head and one tail nodes (regardless of how many inputs we have), so these remain constant. Thus, the space complexity for a singly linked list is:

$$f(n) = kn + 2c$$

Where:

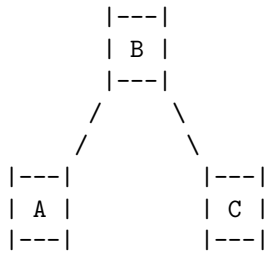
- c is a constant that represents the number of bytes needed to store the head and tail pointer.
- k is a constant that represents the number of bytes needed to store a node.

4 Trees

Before we can talk about trees, we need to talk about graphs.

4.1 Graphs

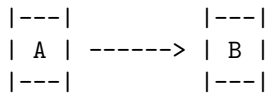
A graph is a collection of nodes and edges. For instance, here is a simple graph:



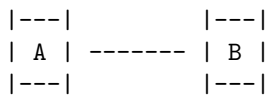
We have nodes A, B, and C, that have some connection to each other. We also have edges, or links, that connect their nodes.

There are two types of edges.

- A directed edge, where we can go from one node to another node, but not the other way around. In other words, we can think of a directed edge as an *one-way street*.



- An undirected edge, where we can go from one node to another node and vice versa. In other words, we can think of an undirected edge as a *two-way street*.



With that said, we should observe that the simple graph that we drew above (containing nodes A, B, and C) could represent a linked list. In particular:

- If the edges were undirected, we would have a doubly linked list.
- If the edges were directed (directional), we would have a singly linked list.

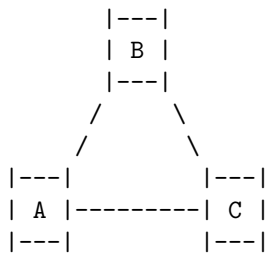
So, a linked list is essentially a chain of nodes in sequence. We have n nodes and $n - 1$ edges.

4.2 What are Trees?

A tree is a graph with two properties:

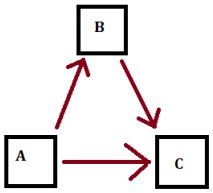
- It has no undirected cycles.

  In graphs, we could theoretically have something like:



This is known as a **cycle** (specifically, an undirected cycle). Essentially, we can go from A to B, B to C, and then back to A from C.

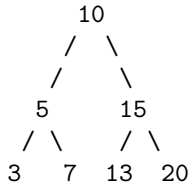
We can also have something like this:



This is not a cycle because we cannot start at A and end up back at A (and the same applies with B). This is because we'll always end up stuck at C. And, if we start at C, we're stuck at C. That being said, if we converted each of the directed edges of this graph into undirected edges, we get an undirected cycle which is not allowed.

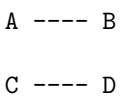
- The nodes must be connected.

Consider the following graph:



This is a graph; it has nodes and it has edges. There are no undirected cycles. Finally, all the nodes are connected (all nodes are connected to the other nodes in this tree in some way). Therefore, this is a tree.

Consider the following graph:



This is not a tree because nodes A and B are not connected to C and D.

4.3 Special Cases of Valid Trees

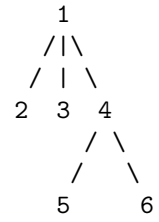
There are a few cases of valid trees that we should discuss.

- The empty ("null") tree. This tree has 0 nodes and 0 edges.
- A tree containing a single node. This has 1 node and 0 edges.

4.4 Rooted vs. Unrooted Trees

Now, we talk briefly about rooted vs. unrooted trees.

- A **rooted** tree is a tree with a hierarchical structure (there is some sense of direction from top to bottom). This looks something like:

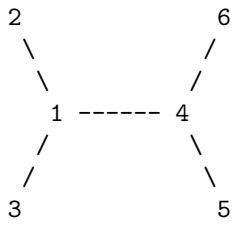


  With a rooted tree, we can consider some definitions.

  - For some node, the **parent** of a node is the node that is directly connected above said node.
  - For some node, the **child** (or **children**) of a node is/are the node(s) that are directly connected below said node.
  - The **root** node is the node at the very top (and thus doesn't have a parent node). In the example above, node 1 is the root node.
  - A node is considered to be a **leaf** node if it doesn't have any children. Nodes 2, 3, 5, 6 are considered leaves.
  - A node is considered to be an **internal** node if it does have children. Nodes 1, 4 are considered internal nodes.

  For example, consider node 4. This node's parent is node 1. This node has two children: node 5 and node 6.

- An **unrooted** tree is one where there is not a top-to-bottom hierarchical structure, but more of an inside-outward structure. This looks something like:



With an unrooted tree, we now have the following definitions:

- The **neighbors** of a node are nodes that are directly connected to said node. For example, node 1 has three neighbor (2, 3, 4). Node 4 also has three neighbor (1, 5, 6).
- A node is considered to be a **leaf** node if it has one neighbor.
- A node is considered to be an **internal** node if it has more than one neighbor.

4.5 Rooted Binary Trees

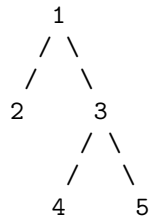
There are a lot of data structures that involve rooted binary trees. So, let's talk about them.

First and foremost, it's rooted (hence the name). This means:

- There is a root node.
- All of the edges have a downward hierarchical relationship.

Trees, in general, do not need to be binary. Any internal node can have any arbitrary number of children. *However*, for a **binary tree**, any node must have either 0, 1, or 2 child/children nodes.

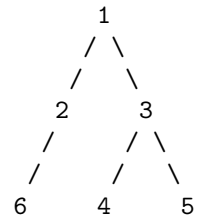
To emphasize this point, consider the binary tree:



Here, we see that:

- Nodes 2, 4, 5 have 0 children.
- Node 1 has 2 children.
- Node 3 has 2 children.

This is a perfect binary tree because every node that's internal has exactly 2 children and every leaf node has exactly 0 children. *However*, a binary tree can support single-child relationships. For example, let's consider the same binary tree, with an additional node:



Here, we see that:

- Nodes 4, 5, 6 have 0 children.
- Node 1 has 2 children.
- Node 3 has 2 children.
- Node 2 has 1 child.

And this is still a valid binary tree (even though 2 only has one child.)

4.6 Tree Traversals

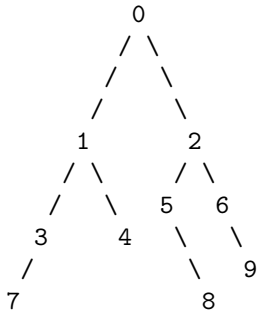
If we store our data in some tree data structure, we need a tree traversal algorithm in order to iterate through all nodes (and our data). In this class, we'll talk about the following tree traversal algorithms:

- **Preorder:** Visit, Left, Right.
- **In-Order:** Left, Visit, Right.
- **Postorder:** Left, Right, Visit.
- **Level-Order:** 1st Level (Left to Right), 2nd Level (Left to Right), ...

We should note that preorder, in-order, and postorder traversals are examples of **depth first search** (DFS) whereas level-order traversal is an example of **breadth first search** (BFS). Regardless of the tree traversal algorithm we use, we will always start at the root.

4.6.1 Preorder Traversal (V, L, R)

Consider the following tree:



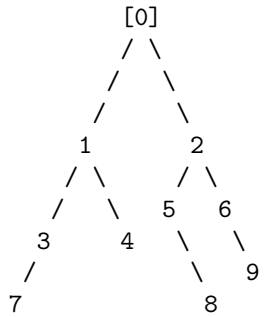
In a preorder traversal, we are guaranteed that for any given node, its ancestors were visited before the node itself. So, in the tree above, if we visit node 1, then we are guaranteed that node 0, its parent, has already been visited. If we visit node 3, then we are guaranteed that node 1 and node 0 have already been visited.

Preorder traversal has three steps:

- Visit the current node.
- Recurse to the left node.
- Recurse to the right node.

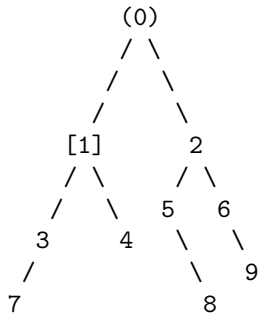
Going back to the example tree, let's run through the preorder traversal algorithm. Note that, in the tree, I'll denote `[n]` as saying that we are at node n and `(n)` as saying that we have already visited node n.

- We start at the root node (0).



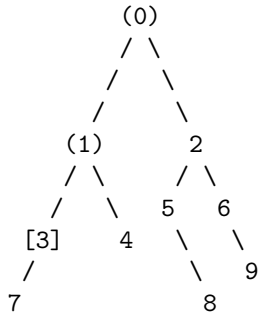

```
Result: 0
```

- Now, we traverse to the left node (1).



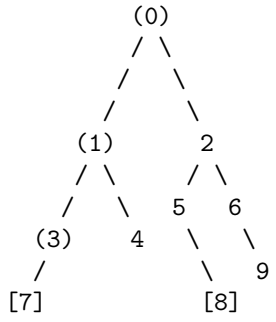

```
Result: 0, 1
```

- Now, we traverse to the left node (3).



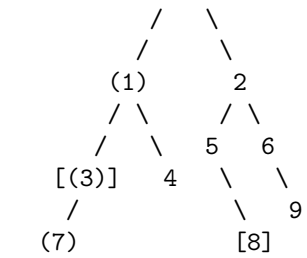

```
Result: 0, 1, 3
```

- Now, we traverse to the left node (7).

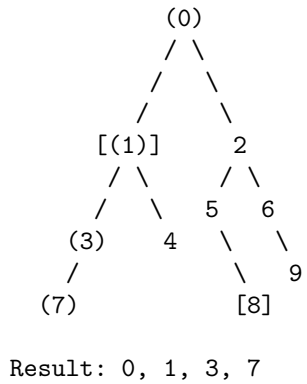



```
Result: 0, 1, 3, 7
```

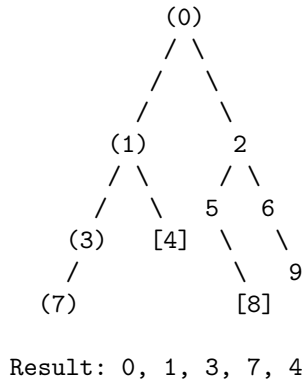
- Now, we would traverse to the left node. However, there's nothing there! So, we're done traversing left. Here, we can try traversing right. However, once again, there is nothing to traverse to. So, we're done with node 7. Let's move back to node 3.



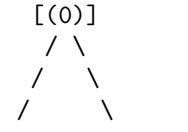
- Now that we're back at node 3, let's try traversing right. However, there is no right node. So, we're done with 3 and we move back up to node 1.

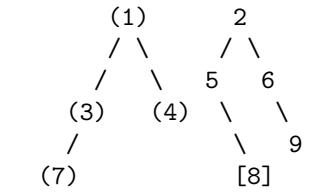


- Now that we're back at node 1, let's try traversing right. In this case, we are able to, so we traverse to the right node (4).



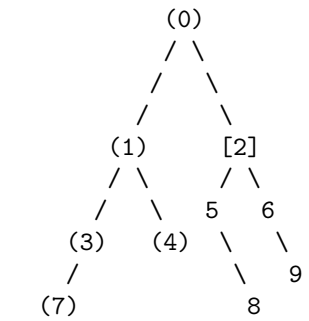
- At node 4, there is no left or right node. So, we're done with node 4. and we can move back up to node 1. Since we've already visited node 1 and its children, we can move back up to the root node, node 0.





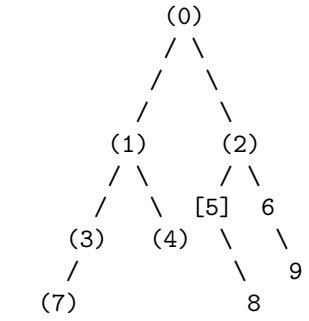

```
Result: 0, 1, 3, 7, 4
```

- Now that we're at node 0, we can try to traverse right (since that's the only operation we can do). Since there are right nodes, we traverse to the right node (2).



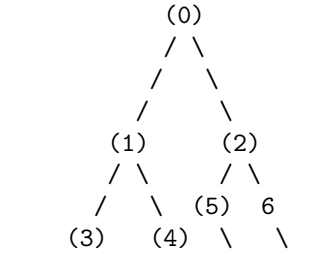

```
Result: 0, 1, 3, 7, 4, 2
```

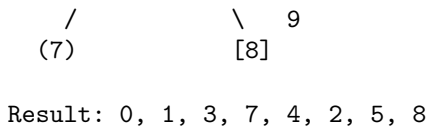
- Node 2 has a left and right child. Of course, we're going to traverse to the left node (5).




```
Result: 0, 1, 3, 7, 4, 2, 5
```

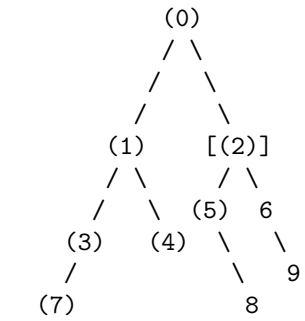
- Node 5 does not have a left child, but node 5 does have a right child, so we traverse to the right node (8).





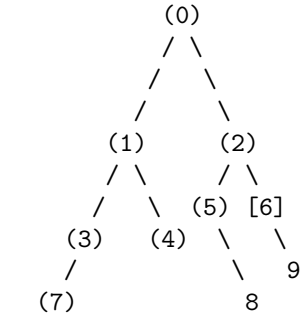
Result: 0, 1, 3, 7, 4, 2, 5, 8

- At node 8, we cannot traverse left or right. So, we're done and we traverse back to node 5; since we've done all operations possible on node 5 (we visited it, we tried to traverse left, and we tried to traverse right), we go back to node 2.



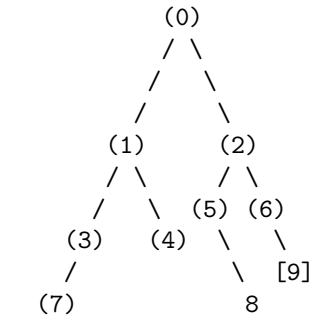
Result: 0, 1, 3, 7, 4, 2, 5, 8

- At node 2, we can only traverse to the right node. Since node 2 has a right child, we can traverse to the right node (6).



Result: 0, 1, 3, 7, 4, 2, 5, 8, 6

- At node 6, we cannot traverse to the left node since there is no left child. But, there is a right child so we traverse to the right node (9).



```
Result: 0, 1, 3, 7, 4, 2, 5, 8, 6, 9
```

- At this point, we traverse back to node 6. Since we're done with node 6, we traverse back to node 2. Since we're done with node 2, we traverse back to node 0 (root node). Since we're done with node 0, we're done! Our final result is:

```
0, 1, 3, 7, 4, 2, 5, 8, 6, 9
```

4.6.2 In-order Traversal (L, V, R)

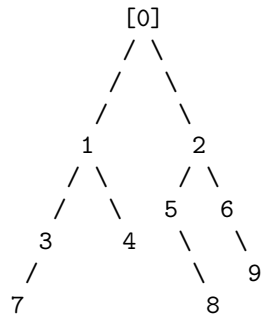
This algorithm works like so:

- Traverse down the left subtree.
- Once that's done, visit the current node.
- Then traverse down the right subtree.

Unlike the other traversal algorithms, an in-order traversal only really makes sense in the context of a binary tree. **Note that**, in the tree, I'll denote `[n]` as saying that we are at node n and `(n)` as saying that we *have been* at node n (but not necessarily visited it yet).

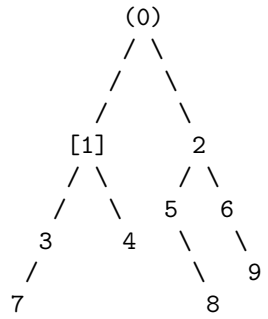
Consider the same tree example from the previous example.

- We start at node 0, the root node.



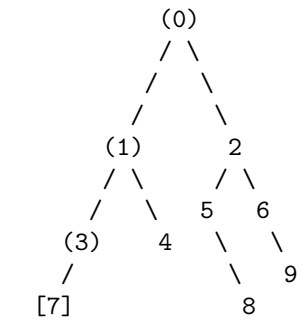

```
Result:
```

- Because of in-order traversal, we immediately traverse to the left node. In this case, we go to node 1.



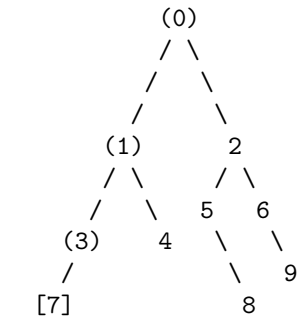

```
Result:
```

- Once again, we go to the left node. For the sake of saving space, we're going to condense two steps into one. First, we traverse node 3, and then we traverse node 7.



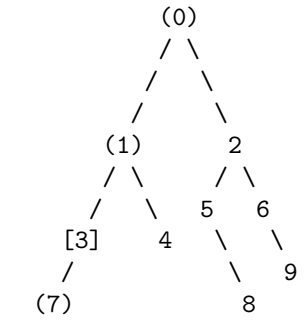

```
Result:
```

- We see that node 7 doesn't have a left child node! So, we *visit* this node.



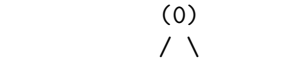

```
Result: 7
```

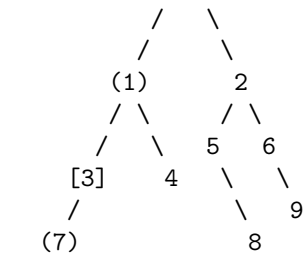
Afterwards, we attempt to traverse to the right child node. However, there is no right child node associated with node 7, so we go back to the previous node (since we're done processing node 7).




```
Result: 7
```

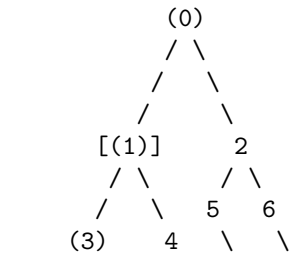
- Now that we're back at node 3, we can formally *visit* it (since we're done going to the left child node).





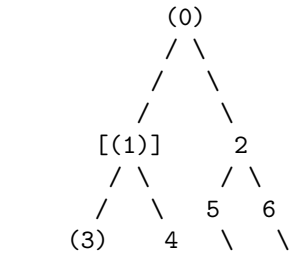

```
Result: 7, 3
```

After this, we can go back to the previous node: node 1.



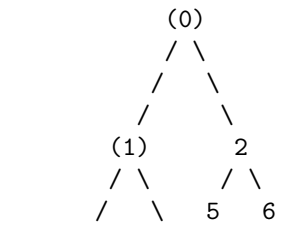

```
Result: 7, 3
```

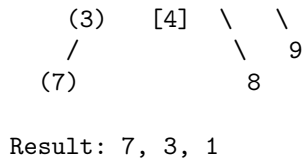
- Since we've already visited node 1's left child node, we can now visit node 1 itself.




```
Result: 7, 3, 1
```

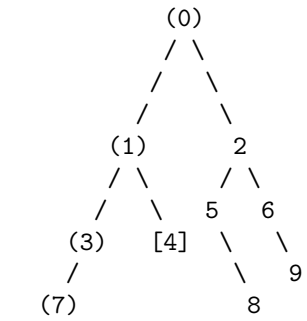
Now, we attempt to traverse to node 1's right child node. Because node 1 *does* have a right child node, we can traverse to it, and so we traverse to node 4.





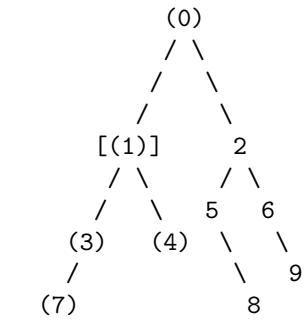

```
Result: 7, 3, 1
```

- From node 4, we try to traverse to this node's left child. However, node 4 doesn't have a left child. So, we can actually visit node 4.



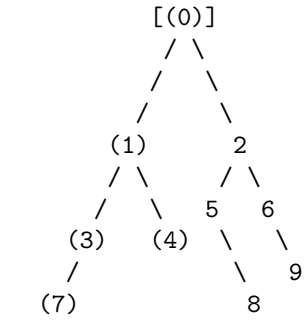

```
Result: 7, 3, 1, 4
```

Once we visit node 4, we try to traverse to this node's right child. Again, this node doesn't have a right child, so we traverse back to the parent node.



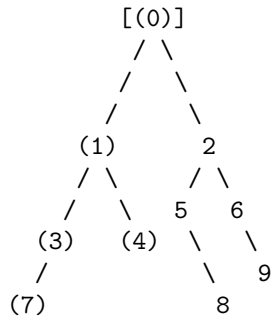

```
Result: 7, 3, 1, 4
```

- We're back at node 1, expect there's nothing to do at node 1 (since we're done with all possible operations). So, we go back to node 0 (the root node).



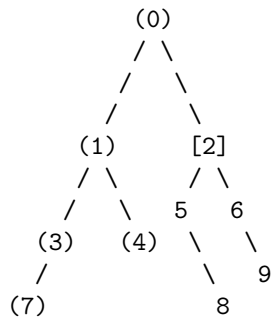
```
Result: 7, 3, 1, 4
```

Because we're done visiting the left child of the root node, we can now visit the node itself.




```
Result: 7, 3, 1, 4, 0
```

At this point, we can traverse to the root node's right neighbor.




```
Result: 7, 3, 1, 4, 0
```

- For the sake of conciseness, I'll omit the remaining steps. However, the result of in-order traversal is:

```
Result: 7, 3, 1, 4, 0, 5, 8, 2, 6, 9
```

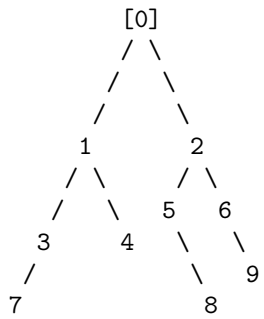
4.6.3 Postorder Traversal (L, R, V)

In postorder traversal, I'm guaranteed that, before I visit any of a node, that I have visited that node's descendents. The idea is as follows:

- Start by visiting the left nodes.
- Then, visit the right nodes.
- After those nodes are all visited, then visit the current

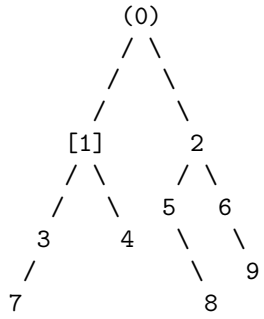
As usual, we're going to stick with the binary tree that we've used. **Note that**, in the tree, I'll denote `[n]` as saying that we are at node n and `(n)` as saying that we *have been* at node n (but not necessarily visited it yet).

- Begin at the root node as always.



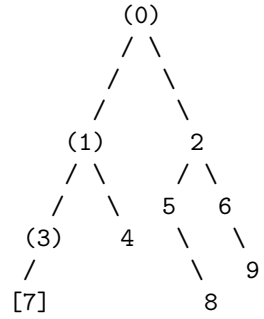
`Result:`

- Let's now traverse to node 1, or node 0's left child node.



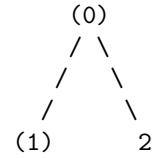
`Result:`

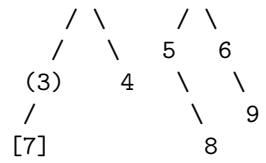
- From node 1, we can traverse to node 3 and then node 7.



`Result:`

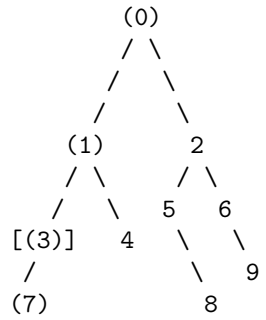
At this point, since node 7 doesn't have any child nodes, we cannot traverse to node 7's left or right child nodes. Therefore, we can visit node 7.





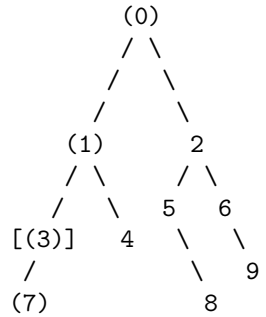
```
Result: 7
```

- Now, we can traverse back to node 3.



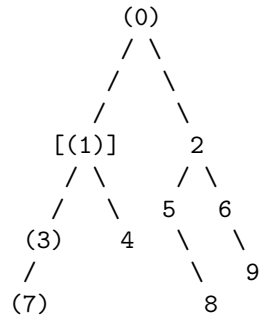
```
Result: 7
```

At node 3, we've already traversed to the left child node (7). Since node 3 doesn't have a right child node, we cannot traverse to that node. Therefore, we can visit node 3:



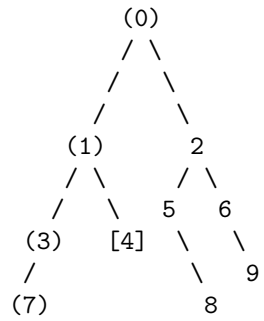
```
Result: 7, 3
```

- Now, we can traverse to node 1.



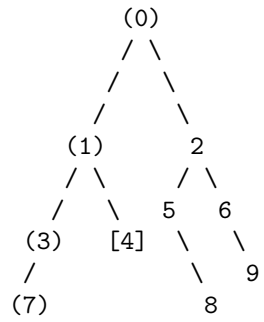
```
Result: 7, 3
```

At node 1, we now need to visit the right child node, so we do that.



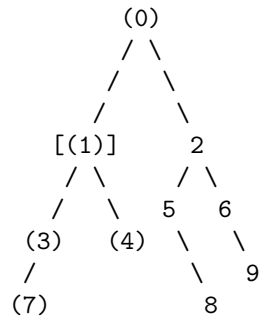

```
Result: 7, 3
```

Since node 4 doesn't have a left or right child node, we cannot traverse to the left or right child node. So, we visit node 4.



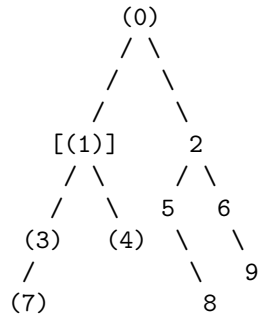

```
Result: 7, 3, 4
```

- Now, we can traverse back to node 1.



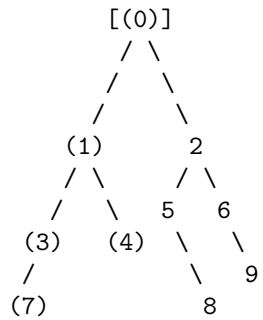

```
Result: 7, 3, 4
```

Since we already traversed to node 1's left and right child nodes (and thus node 1's left and right subtrees), we can visit node 1.



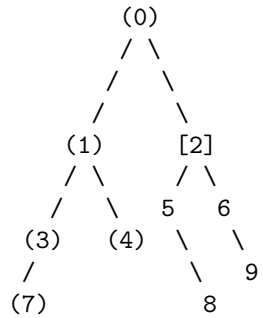

```
Result: 7, 3, 4, 1
```

- Now, we can traverse back to node 0.




```
Result: 7, 3, 4, 1
```

Now that we're at node 0, we need to traverse to the right child node. So, we do that.




```
Result: 7, 3, 4, 1
```

- We've now traversed to node 2. At this point, we need to traverse to node 2's left and right child nodes. To save some steps, we'll do it all in one go. Namely:
  - We traverse to node 5, and then node 8 (since node 5 doesn't have a left child node).
  - Because node 8 doesn't have a left or right child node, we visit node 8 and go back to node 5.
  - Because we've visited node 5's left and right child nodes (noting that node 5 doesn't even have a left child node), we can visit node 5 and then go back to node 2.
  - Now that we're at node 2, we can traverse to node 6, and then back to node 9.

  - At node 9, there is no left or right child node so we can visit node 9 and go back to node 6.
  - At node 6, because there's no left child node and the right child node has already been visited, we can visit node 6 and go back to node 2.
  - Since we've visited all of node 2's left and right child nodes, we can visit node 2 and go back to node 0, the root node.
  - Since we've visited all of node 0's left and right child nodes, we can visit node 0. Thus, we're done.

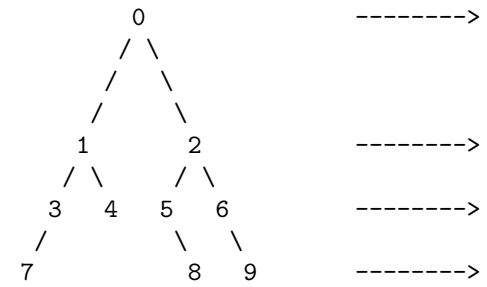
- The final result, then, is:

```
Result: 7, 3, 4, 1, 8, 5, 9, 6, 2, 0
```

4.6.4 Level-Order Traversal

Unlike the other three algorithm, level-order traversal is an example of breadth first search.

The idea is relatively simple. We're traversing with respect to distance away from the root. In this sense, we're traversing like so:



So, essentially, if we traversed using level-order traversal, our result would be:

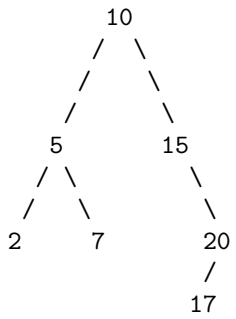
```
Result: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

5 Binary Search Trees

A binary search tree is a special type of binary tree with the following properties:

- It must be a rooted binary tree (can only have 0, 1, or 2 children).
- Every node is larger than all nodes in its left subtree.
- Every node is smaller than all nodes in its right subtree.

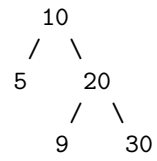
Consider the following tree:



This is a valid binary search tree because:

- It's rooted.
- Each node has 0, 1, or 2 children.
- For any given node, its left node's value are smaller than the given node's value. For example, for node 5, $2 < 5$. Another example is for node 10, where $2 < 5 < 7 < 10$.
- For any given node, its right node's values are bigger than the given node's value. For example, for node 5, $5 < 7$. Another example is for node 10, where $10 < 15 < 17 < 20$.

Consider the following tree:



This is not a binary search tree. While node 20's left and right child nodes meet the criteria ($9 < 20$ and $20 < 30$), node 9 is on the right of node 10 and we know that $10 < 9$ is false.

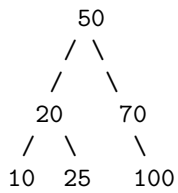
5.1 BST Find Algorithm

Denote `query` to be the query (the element we want to find) and `current` to be the current node (the node that we're at). The binary search tree find algorithm works as follows

1. Start at the root.
2. If `query == current`, success!
3. Otherwise, if `query > current`, traverse right and go back to step 2.
4. Otherwise, if `query < current`, traverse left and go back to step 2.

Remark: If we try to traverse left or right but no such child exists, then the element doesn't exist.

Consider the following BST:



Let's suppose we tried to look for `20` (so `query = 20`). We start at node 50 (so `current = 50`).

- Since `current = (50 != 20) = query`, we need to check the child nodes.
- Since `query = (20 < 50) = current`, we traverse to the left child. Thus, `current = 20`.
- Since `current = (20 == 20) = query`, we're done.

Let's now suppose we tried to look for `25`. Once again, `query = 25` and `current = 50`.

- Since `current = (50 != 25) = query`, we need to check the child nodes.
- Since `query = (25 < 50) = current`, we traverse to the left child. Thus, `current = 20`.
- Since `current = (20 != 25) = query`, we need to check the child nodes.

- Since `query = (25 > 20) = current`, we need to check the right child. Thus, `current = 25`.
- Since `current = (25 == 25) = query`, we're done.

Finally, let's suppose we tried to find `60`. Once again, `query = 60` and `current = 50`.

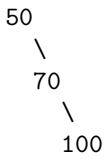
- Since `current = (50 != 60) = query`, we need to check the child nodes.
- Since `query = (60 > 50) = current`, we go to the right child. Thus, `current = 70`.
- Since `current = (70 != 60) = query`, we need to check the child nodes.
- Since `query = (60 < 70) = current`, we go to the left child. Since 70 does not have a left child, the element is not found.

5.2 BST Insert Algorithm

The binary search tree insert algorithm works as follows

1. Perform `find` operation, starting at the root.
2. If `find` succeeds, there is a duplicate element so we don't insert.
3. If `find` doesn't succeed, insert the new element at the site of failure.

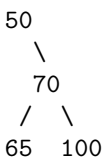
Let's consider a very simple binary search tree:



Let's suppose we tried to insert `100`. Since `100` exists in the binary search tree, we don't need to add this element.

Let's suppose we tried to insert `65`. Since `65` does not exist in the binary search tree, we can add it to the binary search tree. To be concrete:

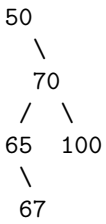
- We start at `50` (root element).
- Since `current != 65` but `current < 65`, we traverse to the right child. Now, `current = 70`.
- Since `current != 65` but `current > 65`, we traverse to the left child.
- Since there is nothing to traverse to (node 70 doesn't havea left child), we append `65`, like so:



As a final example, suppose we tried to insert `67`. The algorithm will run like so:

- We start at `50` (root).
- Since `current != 67` but `current < 67`, we traverse to the right child. Now, `current = 70`.
- Since `current != 67` but `current > 67`, we traverse to the left child. So, `current = 65`.

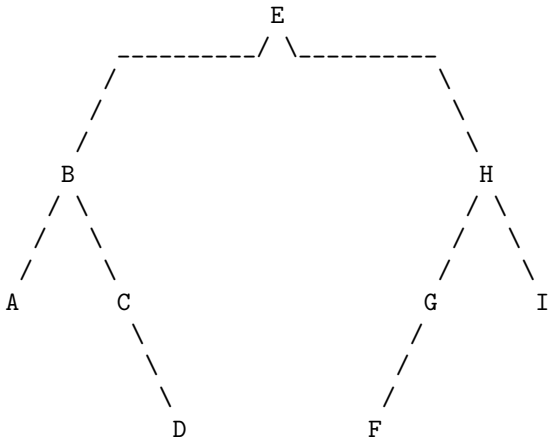
- Since `current != 65`, but `current < 67`, we traverse to the right child.
- But, since there's nothing to traverse to (node 65 doesn't have a right child), we append `67` like so:



5.3 BST Successor Algorithm

What is a node successor? Given some node U, the successor of node U is the next largest node. In other words, it's the node that is immediately larger than node U.

Consider the following binary search tree[2]:



The successors are as follows:

- The successor of A is B.
- The successor of B is C.
- The successor of C is D.
- The successor of D is E.
- The successor of E is F.
- The successor of F is G.
- The successor of G is H.
- The successor of H is I.

If we had an efficient algorithm to determine the successor of a node, then we can implement an efficient iterator that would iterate over our binary search tree in increasing order of size.

How do we find the successor of a given node? The algorithm is as follows:

[2]The tree is a bit oversized due to the way text is formatted, don't worry about that.

- If the node has a right child, traverse right once, then all the way left.

  Consider the following examples:

  - If we wanted to find the successor of E, we would traverse right once ($E \rightarrow \boxed{H}$) and then traverse all the way left ($H \rightarrow G \rightarrow \boxed{F}$).
  - If we wanted to find the successor of B, we would traverse right once ($B \rightarrow C$) and then traverse all the way left. Since C doesn't have any left child nodes, $\boxed{C}$ is the successor.

- Otherwise, traverse up the tree. The first time the current node is its parent's left child, the parent is our successor.

  Consider the following examples:

  - Suppose we wanted to find the successor of D. We note that D doesn't have a right child, so we cannot do the first step of this algorithm and must go to this step of the algorithm.
    - First, we note that D is C's *right* child. So, the left child condition isn't met. So, we go up one.
    - We note that C is B's right child again, so the left child condition isn't met.
    - We note that B is E's *left* child. So, the left child condition is met. Therefore, E is the successor of D.
  - Suppose we wanted to find the successor of A. We note that A doesn't have a right child, so we need to do this step of the algorithm.
    - First, we note that A is B's left child. So, the left child condition is met; thus, B is the successor of A.

5.4 BST Remove Algorithm

As usual, we begin by running the find algorithm. However, if we find the node to delete, we need to consider three cases.

1. No Children: Just delete the node.
2. One Child: Just directly connect my child to my parent.
3. Two Children: Replace my value with my successor's value, and remove me.

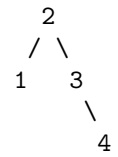
5.4.1 Case 1: No Children

Consider the following binary search tree:

If we wanted to remove `3`, we can just delete it since it has no children. We can set the parent (`2`)'s right child node to `nullptr`.

5.4.2 Case 2: One Child

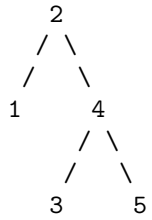
Consider the following binary search tree:



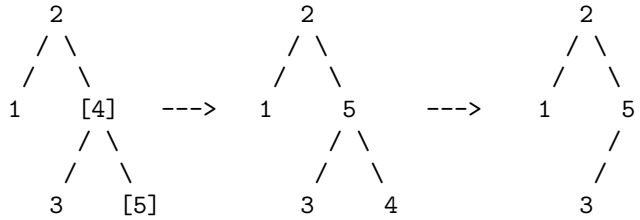
Suppose we wanted to remove 3, we can simply link 2 with 4, like so:

5.4.3 Case 3: Two Children

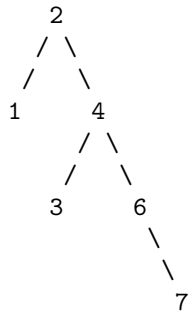
Consider the following binary search tree:



Suppose we wanted to remove 4. To do so, we need to find node 4's successor value, put the successor's value in the node's place, and then remove the node itself. We know that node 4's successor is 5, so we swap their values and then delete the node containing the value that we wanted to remove (in this case, it's the node that we swapped withthe successor). A visualization is shown below[3]:

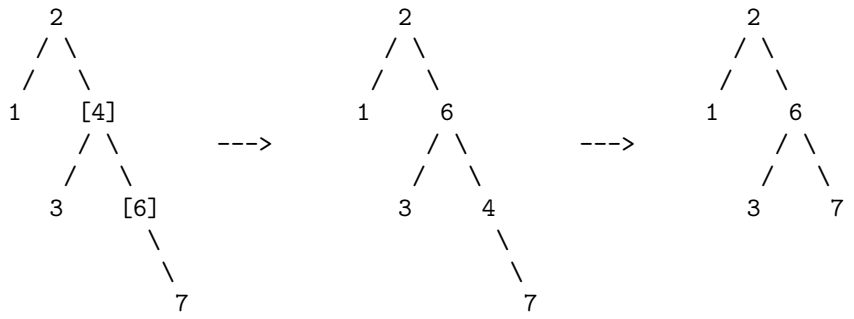


But, what if we had a more complex example? Suppose we had to deal with this binary search tree:



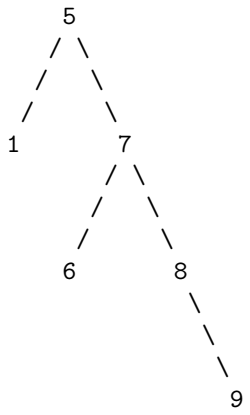
Suppose we wanted to remove 4. 4's successor is 6. We begin by swapping the two values like usual. Then, we can remove the node where 4 is at right now and attach the now-deleted node's child node to node 6.

[3] I put [] to emphasize the two nodes being swapped. It serves no other purpose.



5.5 Height of a Node and Tree

Consider this tree (which we'll use for some examples to supplement the definitions):



Then, we say that:

- The **height** of a **node** is the longest distance (number of edges) from said node to a leaf.

  In the above tree:

  - The distance from node 9 to a leaf is 0 (no edges).
  - The distance from node 8 to a leaf is 1 (1 edge).
  - The distance from node 7 to a leaf is 2. Although node 7 can reach two different leaf nodes (6, 9), we are only interested in the farthest leaf node (9).
  - The distance from node 5 to a leaf is 3. Although node 5 can reach three different leaf nodes (1, 6, 9), we are only interested in the farthest leaf node (9).

- The **height** of a **tree** is the height of the root of the tree.

  The root of the tree is node 5, so we say that the height of the tree is 3 (since we want to find the distance from the root node to the farthest leaf node).

5.6 Tree Balance

We can think of tree balance as a metric of how tall a tree is with respect to the number of nodes it has. In particular, for some n, we can think of tree balancing as a spectrum between perfectly unbalanced and perfectly balanced.

Consider $n = 7$ (a tree with 7 nodes).

In a perfectly unbalanced tree, we have a height of **6**. In a perfectly balanced tree, we have a height of **2**. Both trees are binary search trees.

Basically, a perfectly unbalanced binary search tree is like a linked list.

5.7 Time Complexity

Part of evaluating the time complexity of any BST is figuring out the tree's shape; whether the tree is balanced or unbalanced will make a difference.

For the `find`, `insert`, and `delete` operations, the worst-case runtime is as follows:

Tree Type	Worst Case Big-O	Why?
Perfectly Unbalanced Tree	$\boxed{O(n)}$	For a perfectly unbalanced tree, if we have n nodes, then a perfectly unbalanced tree will have a height of $n-1$. The worst case would occur if we had to traverse over all the edges of the tree.
Perfectly Balanced Tree	$O(\log(n))$	The reason why this is $O(\log(n))$ – more specifically, $O(\log_2(n+1)-1)$ – is because even if we double the number of nodes in a tree, the tree's height would only grow by 1.

Remark: $O(\log(n))$ is not actually the worst case; this is actually a very nice case simply because this assumes that a tree is perfectly balanced. In other words, *if* the tree was perfectly balanced, the worst-case runtime would be $O(\log(n))$; however, because any given tree will probably not be perfect, we cannot make that assumption. So, the worst-case runtime for any given binary search tree is actually $O(n)$.

5.7.1 Find Algorithm: Best vs. Worst vs. Average Case

We should note that:

- The **best** case scenario is if the query is the root.
- The **worst** case scenario is if we need to work with a perfectly unbalanced tree and the query is not found.
- The **average** case scenario is the theoretical expected value over all trees and queries.

For n elements as $n \to \infty$:

Case	Runtime	Remark(s)
Best	$O(1)$	It doesn't matter if the tree is perfectly balanced or unbalanced if the root node is the right node.
Worst	$O(n)$	If the tree is perfectly unbalanced and the value was either not found or is the last node in the tree (i.e. a leaf node).

The average case is a bit more complicated. In particular, we need to assume the following:

1. All n elements are equally likely to be searched for.

   If we had a binary search tree with the elements $1, 2, 3$, then:

   $$P(Q = 1) = P(Q = 2) = P(Q = 3) = \frac{1}{n} = \frac{1}{3}$$

   This is saying that the probability that our query is `1` is the same as the probability that our query is `2` which is the same as saying that the probability that our query is `3`, or $\frac{1}{3}$. This holds for n elements.

2. All $n!$ possible insertion orders are equally likely.

   If we had the elements $1, 2, 3$, there are $3! = 6$ possible insertion orders:

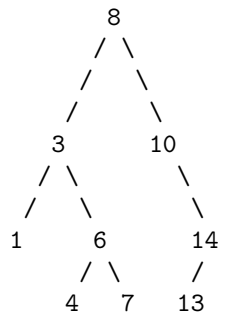
   - 123
   - 132
   - 213
   - 231
   - 312
   - 321

   However, `213` and `231` gave us the same tree structure. So, there are 5 unique tree structures for 6 possible insertion orders.

5.7.2 Depth of a Node

The **depth of a node** is the number of nodes in the path from that node to the root.

For example, consider the following binary search tree:



We can say that:

- The number of nodes from the root node to itself is **1**: itself.
- The number of nodes from node 10 to the root node is **2**.

- The number of nodes from node 14 to the root node is **14**.
- The number of nodes from node 13 to the root node is **4**.
- The number of nodes from node 3 to the root node is **2**.
- The number of nodes from node 6 to the root node is **3**.
- The number of nodes from node 4 to the root node is **4**.
- The number of nodes from node 7 to the root node is **4**.
- The number of nodes from node 1 to the root node is **3**.

The **average case time complexity** is the expected number of operations to find a query.

Suppose one operation is one comparison and we were looking for the number `3`. This would require **2** comparisons, or 2 operations. The average case time complexity is the expected number of operations for every node in this tree. This is equivalent to the expected depth.

Generally speaking, the number of comparisons to find a node is equal to the depth of that node.

6 Treaps and Randomized Search Trees

Here, we will talk about treaps and randomized search trees, along with a concept called *AVL rotations*.

6.1 Treap

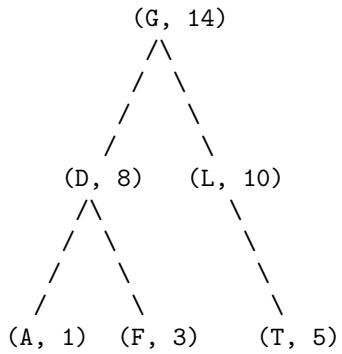
A **treap** is a special tree data structure. The name comes from how this is a *tree* data structure that makes use of a heap. In particular, a treap stores a `(key, priority)` pair.

How does it make use of both a tree and a heap?

- **BST** properties with respect to *keys*.
  - Larger than all keys in the left subtree.
  - Smaller than all keys in the right subtree.

  In other words, if we ignored the priorities, it would hold the binary search tree properties.
- **Heap** properties with respect to *priorities*.
  - Larger than all priorities below.

Consider the following valid treap, where the *keys* are the letters and the *priorities* are the numbers:



With respect to the binary search tree properties:

- We know that $D < G$.
- We know that $A < D$ and $D < F$.
- We know that $G < L$.
- We know that $L < T$.

So, it does fulfill the BST properties. With respect to the keys:

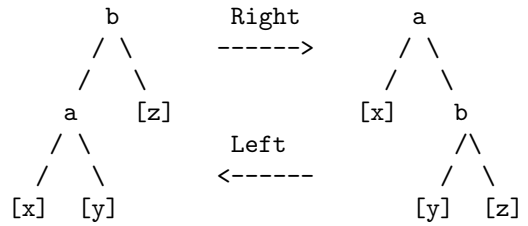
- We know that 14 is larger than all other nodes (its children); it's also at the top of the tree.
- We know that 8 is larger than its children nodes ($1 < 3 < 8$).
- We know that 10 is larger than its child node ($10 > 5$).

So, it does fulfill the Heap properties.

6.2 AVL Rotations

Sometimes, we want to be able to restructure a binary search tree without damaging its binary search tree properties. Here, we make use of something called **AVL Rotations**.

Consider the following two trees (here, `[a]` means that a is a potential subtree):



Remark: We can say that, by doing a right AVL rotation, we are rotating a and b clockwise. By doing a left AVL rotation, we are rotating a and b counterclockwise.

Regarding the left tree:

- b is the parent node.
- a is a child node with two children.
- z is a right subtree. It could either be one node, an entire subtree, or a `nullptr`.
- x is a left subtree (with respect to a). It could either be one node, an entire subtree, or empty.
- y is a right subtree (with respect to a). It could either be one node, an entire subtree, or empty.

Given the left tree, we can do a **right AVL rotation**, where:

- We make a the parent.
- We make b become its right child.
- x remains the left child of a.
- z remains the right child of b.
- We make y the left child of b.

More concretely, regarding the left tree:

- We know that a is the left child of b. Therefore, $a < b$.
- We know that z is the right child of b. That means everything in the z subtree is greater than b. Therefore, because $b > a$, it follows that everything in the z subtree is greater than a.
- The x subtree is a left descendent of a, so all nodes in x is less than a and is therefore less than b.
- The y subtree is a right descendent of a, so all nodes in y is greater than a.

Essentially:

```
b > a       a < b
b < z       a < z
b > x       a < y
b > y       a > x
```

In this particular AVL rotation:

- x (and the edge going into x) and z (and the edge going into z) remain unchanged. Essentially, this means that x remains a left child of a and z remains a right child of $b¿$
- Specifically, the only thing that we are changing is are b, a, and y, and their edge relationships.

Why can we make these changes?

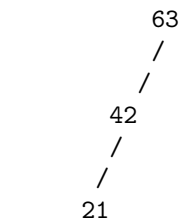
- We know that $a < b$ so making a the root node and b a right child node is valid.
- We know that $b > y$ and $a < y$ (or $a < y < b$), so making y the left child of b (where b is the right child of a) still maintains the $a < y < b$ property.

When making these changes, the key thing to note is that we are still maintaining BST properties.

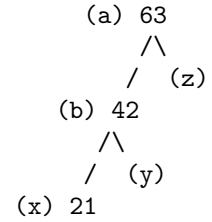
Of course, the same idea applies when doing a **left AVL rotation**.

6.2.1 Example 1: Right AVL Rotation

Consider the following binary search tree:



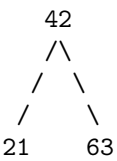
Suppose we wanted to do a right AVL rotation on nodes `63` and `42`. Let's begin by labeling these nodes:



Where:

- We will rotate `42` and `63` (nodes a and b).
- Nodes y and z are `nullptr`.

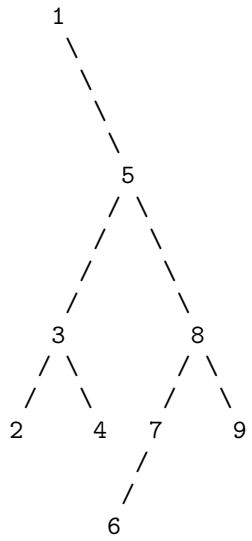
When doing a right rotation, a becomes the new parent and b becomes the new right child for a. That is:



Which is the result.

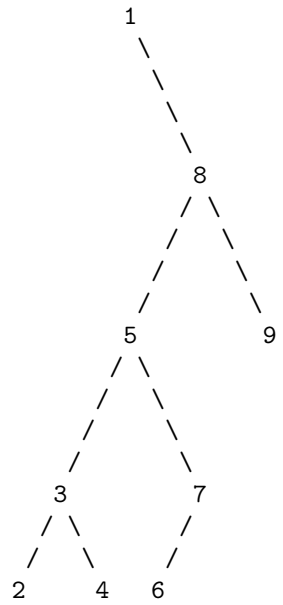
6.2.2 Example 2: Left AVL Rotation

Suppose we have the following binary search tree:



Let's suppose that we wanted to perform a left AVL rotation on nodes `5` and `8`. Then:

- `1` should remain unchanged.
- `8` would be the right child of `1`.
- We would rotate `5` counterclockwise, thus it will be the left child of `8`.
- The left subtree of `5` (in the original tree) and the right subtree of `8` (in the original tree) remain unchanged. So, we append those subtrees on their respective parent nodes.
- The left subtree of `8` (in the original tree) now becomes the right child of `5`.



6.3 Treap Insertion

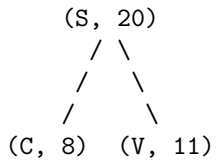
To insert a new key/priority pair:

(1) We first insert this pair via the BST insertion algorithm with respect to the keys.

(2) We use AVL rotations to "bubble up" to fix Heap with respect to priorities.

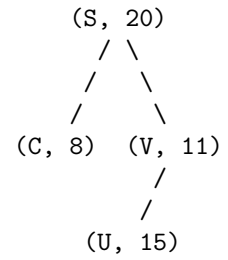
6.3.1 Example 1: Simple Treap Insertion

Recall that the letter represents the key and the number represents the priority. Consider the following treap:

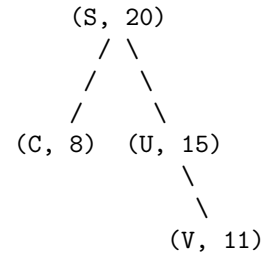


And let's suppose we wanted to insert $(U, 15)$.

(1) We first insert this pair like how we would insert any pair by using the BST insertion algorithm. The treap would look something like:

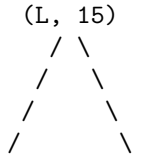


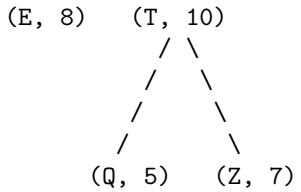
(2) By inserting this pair, though, we have violated the heap properties. So, we're going to do a **right AVL rotation** on U and V so that U becomes the new parent. So:



6.3.2 Example 2: Slightly Harder Treap Insertion

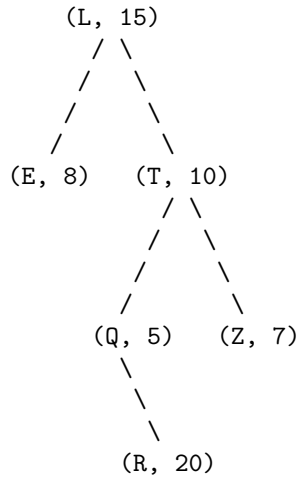
Recall that the letter represents the key and the number represents the priority. Consider the following treap:



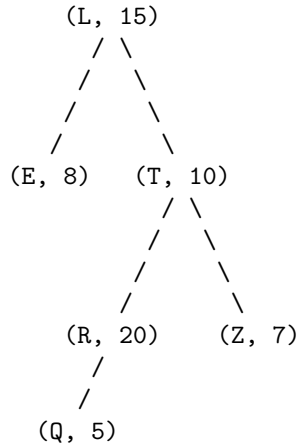


And let's suppose we wanted to insert $(R, 20)$.

(1) We first insert this pair like how we would insert any pair by using the BST insertion algorithm. The treap would look something like:



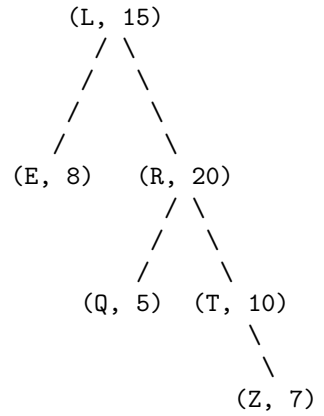
(2) By inserting this pair, though, we have violated the heap properties. So, we're going to do a **left AVL rotation** on Q and R so that R becomes the parent and Q the child.



The changes we made are:

- All we did was made R the parent.
- Because $Q < R$, Q is now the left child of R.

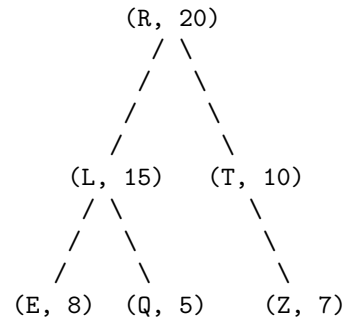
(3) However, in this position, we are still violating the heap properties. So, we need to do another a **right AVL rotation** on R and T.



The changes we made are:

- R is now the parent.
- T is the right child.
- R keeps the left child.

(4) Once again, we are still not done since we are still violating the heap properties. So, we need to yet another **left AVL rotation** on R and L so that R becomes the parent and L the child.



The changes we made are:

- R is the parent.
- L became the left child.
- R's prior left child (Q) became the L's new right child.
- Everything else remains unchanged.

6.4 Randomized Search Trees (RSTs)

A randomized search tree is simply a **treap** where:

- We represent the elements as *keys* (thus, maintaining BST properties).
- We randomly generate priorities (maintaining heap properties).

In other words, we don't assign a priority level to each element, but a random priority level is assigned to each element for us.

Because of the randomness of the priorities, we hope that (on average) we get a better balanced tree.

6.4.1 Example 1: Sorted Numbers

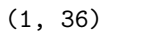
Suppose we wanted to insert the following integers into a binary search tree in this order:

```
1, 2, 3, 4, 5, 6, 7
```

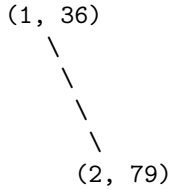
If we put this into a regular binary search tree, we will get a perfectly unbalanced binary search tree, which is terrible if we care about performance.

Let's now use a *randomized* search tree (a treap) and see if this changes anything.

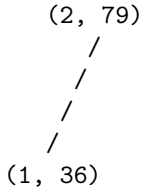
- Suppose we wanted to insert `1`. Suppose the randomly generated priority is `36`. So:



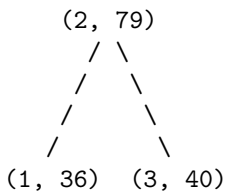
- Now, suppose we wanted to insert `2`. Suppose the randomly generated priority is `79`. So:



  Afer doing a left AVT (so we can maintain heap properties):

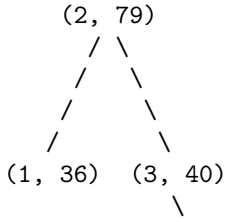


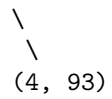
- Now, suppose we wanted to insert `3`. Suppose the randomly generated priority is `40`. So:



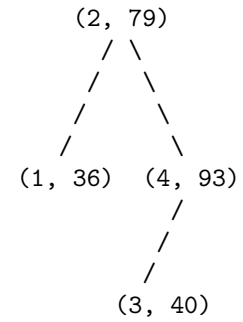
  Since the heap property is valid, we don't need to change anything.

- Now, suppose we wanted to insert `4`. Suppose the randonly generated priority is `93`. Then:

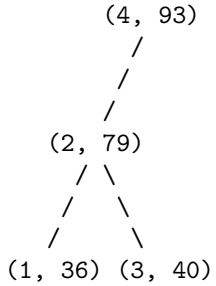




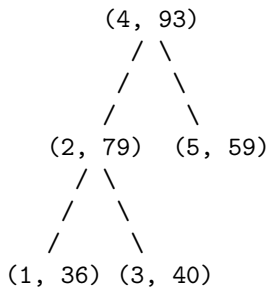
Since the heap properties are violated, we now need to do a left AVL rotation on `4` and `3`. This gives us:



Since the heap properties are violated, we (again) need to do another left AVL rotation on `4` and `2`. This gives us:

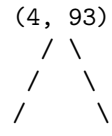


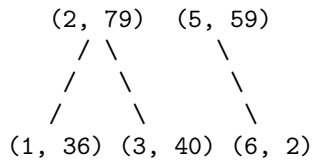
- Now, suppose we wanted to insert `5`. Suppose the randonly generated priority is `59`. Then:



The heap properties are satisfied so no additional changes are needed.

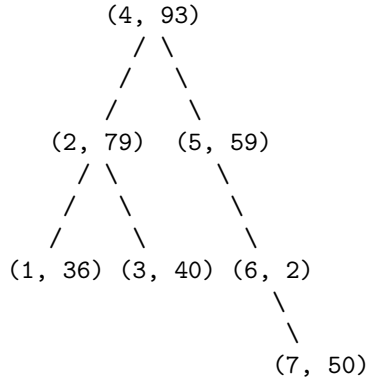
- Now, suppose we wanted to insert `6`. Suppose the randonly generated priority is `2`. Then:



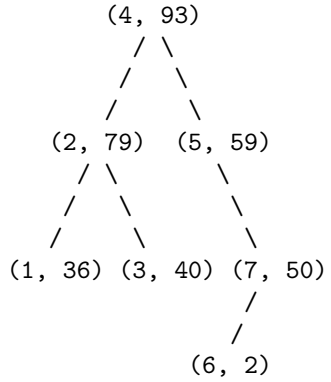


The heap properties are satisfied so no additional changes are needed.

- Now, suppose we wanted to insert `7`. Suppose the randomly generated priority is `50`. Then:



Since the heap properties are not satisfied, we need to do a left AVL rotation on `7` and `6`. So:



The heap properties are now satisifed, so no additional changes are needed.

At this point, we should note that this *randomized search tree* looks significantly more balanced when compared to a normal binary search tree (which would have been a perfectly unbalanced binary search tree, or a linked list).

Again, a randomized search tree still has $O(n)$ worst case time complexity because our key/priority pairs could have been $(1, 7)$, $(2, 6)$, $(3, 5)$, $(4, 4)$, $(5, 3)$, and so on. So, we could still get unlucky with the randomly generated numbers.

7 AVL Trees

We know that a binary search tree has an average time complexity of $O(\log n)$ and worst time complexity of $O(n)$. Of course, even the average time complexity that we found was derived from some unrealistic assumptions. To mitigate this, we introduced the randomized search tree, which is an extension of a binary search tree that makes the assumptions more reasonable. That being said, the worst case time complexity of a randomized search tree is $O(n)$. What if we wanted the worst time complexity of $O(\log n)$?

7.1 Introduction to AVL Trees

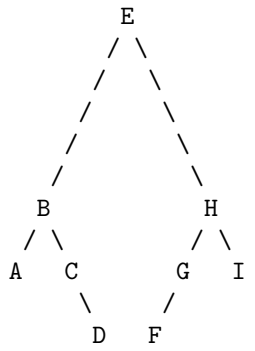
AVL trees are another extension of binary search trees. The only difference, of course, is that it achieves a $O(\log n)$ worst-case for tbe find, insert, and remove operations.

Before we talk about an AVL tree, we need to talk about some concepts.

- Balance Factor (BH): If R is the height of the right subtree and L is the height of the left subtree, then:
$$\text{BH} = R - L$$
Recall: The **height** of a **node** is the longest distance (number of edges) from said node to a leaf.
- AVL Tree: A binary search tree in which every node has a balance factor of -1, 0, or 1.

7.1.1 Example 1: AVL Tree

Consider the following binary search tree:



We note that:

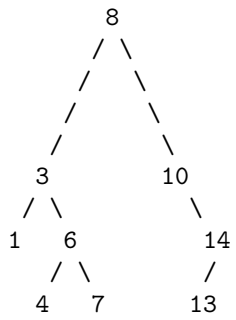
- Nodes A, D, F, and I have a balance factor of 0 because their left and right height is 0 (they have no subtrees).
- Node C has a height of 0 on the left (no left child) and a height of 1 on the right (right subtree with 1 node: D). So:
$$BH_C = 1 - 0 = 1$$
- Node G has a height of 1 on the left (left subtree with 1 node: F) and a height of 0 on the right (no right child), so:
$$BH_G = 0 - 1 = -1$$
- Node H has a height of 2 on the left (left subtree with 2 nodes: G and F) and a height of 1 on the right (right subtree with 1 node: I). So:
$$BH_H = 1 - 2 = -1$$

- Node B has a height of 1 on the left (left subtree with 1 node: A) and a height of 2 on the right (right subtree with 2 nodes: C and D). So:
$$BH_B = 2 - 1 = 1$$
- Node E has a height of 3 on the left (left subtree with height 3: B, C, D) and a height of 3 on the right (right subtree with height 3: H, G, F). So:
$$BH_E = 3 - 3 = 0$$
Again, remember that the height is the longest distance from one node to a leaf. So, for the left subtree of E, this would be 3 since B, C, and D form the longest distance from E to a leaf.

Because every node has a balance factor of -1, 0, or 1, this is an AVL tree.

7.1.2 Example 2: Non-AVL Tree

Consider the following binary search tree:



We note that:

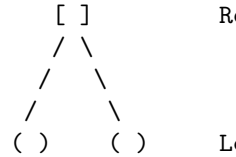
- Nodes `1`, `4`, `7`, and `13` all have balance factor of 0 since they don't have any subtree.
- Node `14` has a left height of 1 and right height of 0, so its height is -1.
- Node `6` has a left and right height of 1, so its height is 0.
- Node `3` has a left height of 1 and right height of 2, so its height of 1.
- Node `10` has a left height of 0 and a right height of 2, so its height is 2.

Here, we note that Node `10` violates the property of an AVL tree (its balance factor is not -1, 0, or 1). So, this tree is **not** an AVL tree.

7.2 Proof of AVL Tree Worst-Case Time Complexity

We said that worst-case time complexity to find an element in an AVL tree is $O(\log n)$. We need to prove that this is actually the case.

Proof. Denote N_h to be the minimum number of nodes that can form an AVL tree with height h. Consider, for instance, the following tree:



`Root Node`

`Left / Right Subtree`
`Each subtree can be empty, have one node,`
`or many nodes.`

If this AVL tree has height h, we want to minimize the number of nodes that can form this tree. This tree would have, in its entirety, N_h nodes. Suppose we picked an arbitrary subtree, say, the right subtree on the right. Then, this subtree would have a height of $h-1$ and would thus have N_{h-1} nodes. By the properties of an AVL tree, we know that the root node can have a balance factor of -1, 0, or 1. The left subtree (in its worst case scenario) would have a height of $h-2$; this is because the left height plus the root node would give us a height of $h-2+1=h-1$, and the right height plus the root node would give us a height of $h-1+1=h$. So, the balance factor of the root node would be:

$$h-(h-1)=h-h+1=1$$

Since the left subtree has a height of $h-2$, it follows that the left subtree has N_{h-2} nodes.

We can define a recurrence relation representing the total number of nodes in this tree like so:

$$\overbrace{N_h}^{\text{Total number of nodes}} = \underbrace{N_{h-1}}_{\substack{\text{Number of nodes}\\ \text{in right subtree}}} + \overbrace{N_{h-2}}^{\substack{\text{Number of nodes}\\ \text{in left subtree}}} + \underbrace{1}_{\text{Root node}}$$

For this proof only, let's assume that the height of a node is determined by the number of nodes as opposed to the number of edges. For instance, a tree with one node would have a height of 1 and a tree with no nodes would have a height of 0. Then, we know that:

$$N_1=1$$

$$N_2=N_1+N_0+1=1+1=2$$

Then, we can do:

$$N_{h-1}=N_{h-2}+N_{h-3}+1$$

$$N_h=(N_{h-2}+N_{h-3}+1)+N_{h-2}+1=2N_{h-2}+N_{h-3}+2$$

We know that:

$$N_h>2N_{h-2}$$

This is because when we had a tree with N_h nodes, one of the subtrees had N_{h-1} nodes and the other had N_{h-2} nodes. By definition, H_{n-2} is smaller than H_{n-1}. Since, $H_n=N_{h-2}+N_{h-1}+1$, by definition H_n is greater than $2N_{h-2}$. Therefore:

$$N_h>2^{\frac{h}{2}} \implies \log N_h>\log 2^{\frac{h}{2}} \implies 2\log N_h>h$$

And, therefore:

$$\boxed{h \text{ is } O(\log N_h)}$$

□

7.3 AVL Tree Insertion

We now discuss AVL tree insertions. We won't discuss the AVL tree `find` algorithm simply because the `find` algorithm is equivalent to that of a normal binary search tree's `find` algorithm.

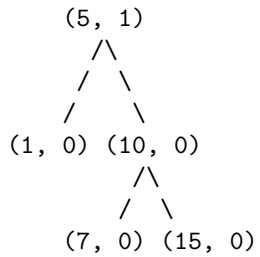
The insertion algorithm for an AVL tree is slightly more complicated than a normal binary search tree. Specifically, we need to do the following:

- Insert the element like you would with a regular BST insertion.
- After this is done, update balance factors across the tree.

- If any balance factors were broken as a result of this insertion, fix broken balance factors using AVL rotations.

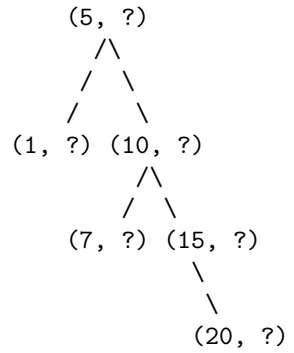
7.3.1 Example 1: Insertion

Consider the following AVL tree, where each entry is formatted like (Element, Balance Factor):

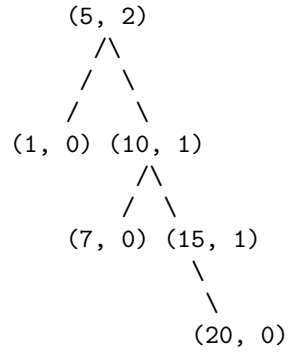


If we wanted to insert `20`, we would do the following:

(1) Insert the element like you would with a normal BST.



(2) We now need to update the balance factors for each node.



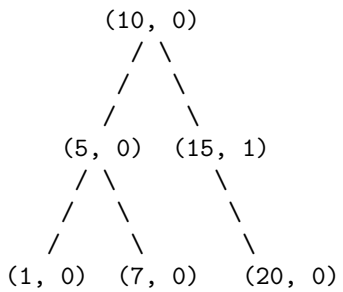
Remark: We only need to update the balance factor of all the nodes *along the path* from the new node to the root node.

(3) Notice how node `5` has a balance factor of `2`. This is a problem; we now need to fix it using AVL rotations. So, we'll perform a **left AVL rotation** on nodes `5` and `10`. How did we figure this out?

- Node `5` is the node that is out of balance.
- Node `10` is the *heavier* child; it's the child whose subtree is contributing to the invalid balance factor that `5` has.

We perform a left AVL rotation because 5 is the bad node and 10 is the heavier node; it is also the right child. So, by rotating the two nodes counterclockwise, we can drop node 5 and bump node 10, balancing the tree.

Doing the left AVL rotation yields the tree:



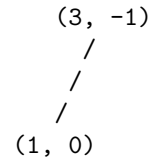
We got these values because:

- Nodes 20, 7, and 1 are leaf nodes.
- Node 5 has left height 1 and right height 1, so it has an overall height of 0.
- Node 15 has left height 0 and right height 1, so it has an overall height of 1.
- Node 10 has left height 2 and right height 2, so it has an overall height of 0.

That being said, we now have an AVL tree.

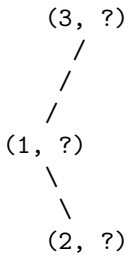
7.3.2 Example 2: Insertion

Consider the following AVL tree, where each entry is formatted like (Element, Balance Factor):

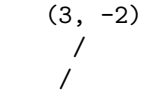


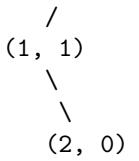
If we wanted to insert 2, we would do the following:

(1) Insert the element like you would with a normal BST.

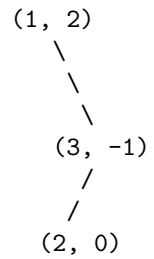


(2) We now need to update the balance factors for each node.

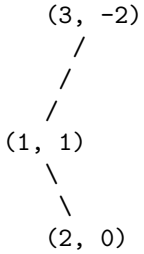




(3) Since `3` has a load factor of `-2`, we need to fix this. To do so, we perform a right AVL rotation on nodes `1` and `3`.



However, this is also invalid. To fix this, we perform a left AVL rotation on nodes `1` and `3`:

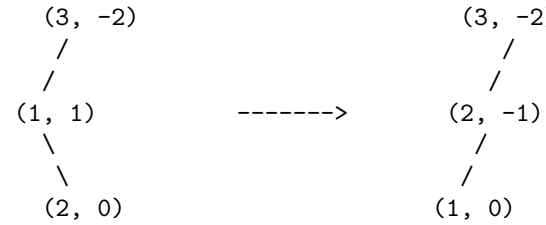


Notice how we basically went back and forth here. This is a problem. What happened?

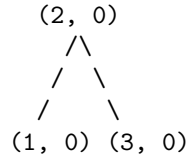
- Well, we need to keep track of all three nodes, not just nodes `1` and `3`.
- In the prior examples, the out-of-balance node, its heavy child, and the grandchild formed a *straight* line. That is:

  So, we could just do a right AVL rotation for this structure. Likewise, if we had the straight lines (in the opposite direction), we could just do a left AVL rotation.
- In this example, though, we have the out-of-balance node, its child, but the other child being in a different direction; that is, it formed a *kink* shape like so:

So, what we need to do is convert this kink shape/structure into a straight line. So, we will have to do a double rotation. We will begin by doing a left AVL rotation with nodes `1` and `2` (the child and the grandchild):



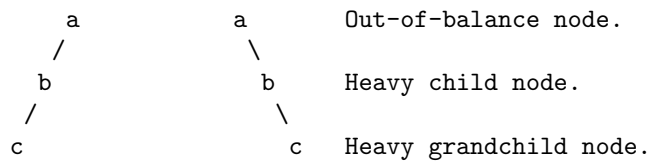
Now, we can do a right AVL rotation with nodes `2` and `3`:



Which is an AVL tree. This behavior can be replicated even if the tree is flipped across.

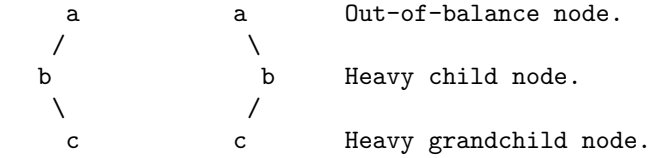
7.3.3 AVL Tree Insertion Summary

- If the nodes are in a straight line, like so:



Then, we can do one AVL rotation on a and b.

- If the nodes are in a kink shape, like so:



Then, we need to do an AVL rotation on b and c first, and then a and c. In other words, we rotate the child and grandchild nodes first, and then the parent and its new child node next.

8 Red-Black Trees

The red-black tree is a slightly more complicated self-balancing binary search tree. Here, we will talk more about what it is and why we have it.

8.1 Properties

Here are some properties of a red-black tree.

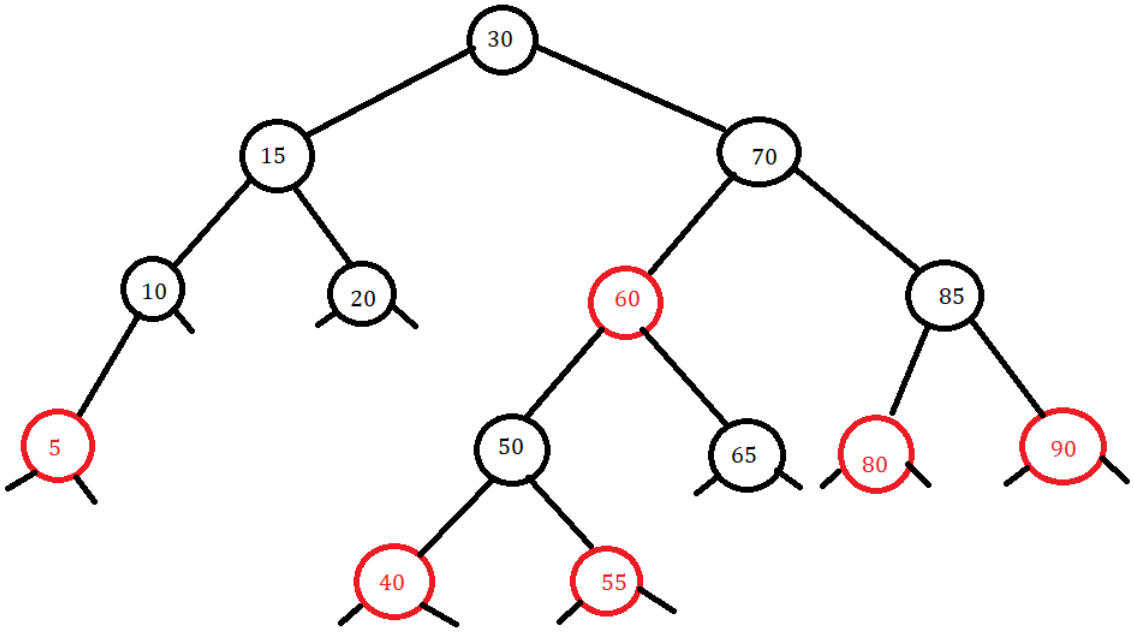
(1) All nodes must either be **black** or **red**.

(2) The root must be **black**.

(3) If a node is **red**, all of its children must be **black**. You cannot a **red** node with a **red** child.

(4) For every node u, every possible path from u to a `null` reference must have the same number of **black** nodes. A `null` reference is **black**. Essentially, for every single node in the tree, we should be able to take any arbitrary path to get to a `null` reference and you should hit the same exact number of black nodes for every single path that could be taken from that given node.

Consider the following tree:



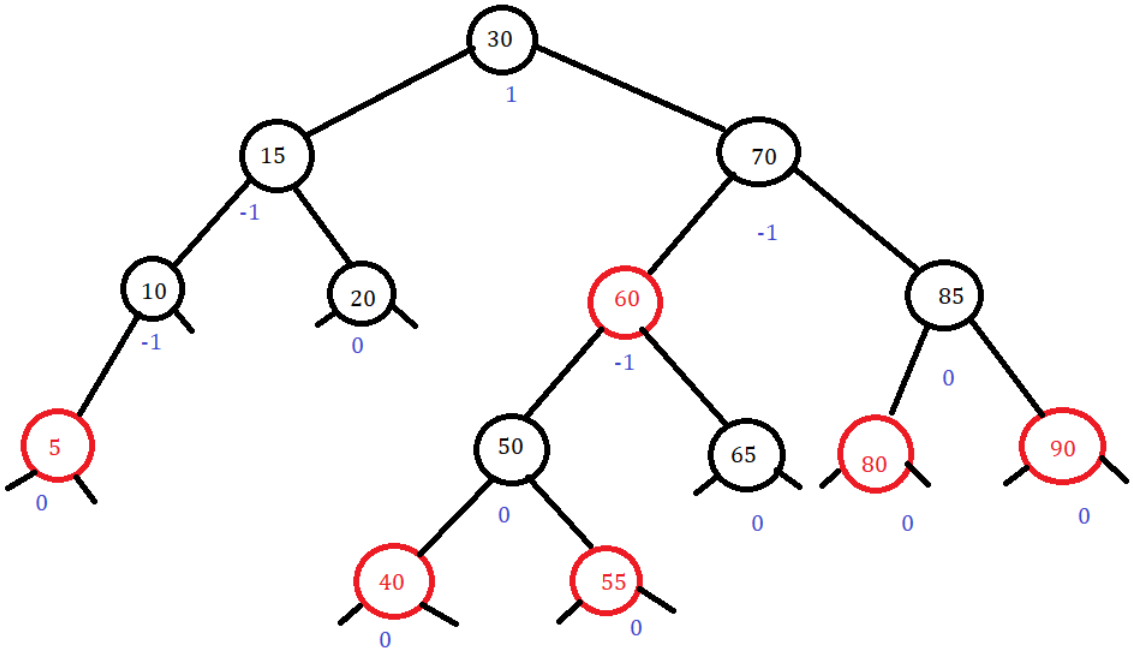
- The first property is satisfied; this can be seen above.
- The second property is satisfied since the root node (`30`) is, indeed, black.
- The third property is satisfied. In particular, we notice that node `60` has two children that are black; the leaf nodes all have no children.
- The fourth property is satisifed. Let's consider some examples:
  - Node `5` has no black nodes. It also has 0 possible ways to go from said node to a `null` reference.
  - Node `10` has 1 black node (itself) to a `null` reference. There is also one possible way to go to a `null` reference.
  - Node `15` has 2 black nodes (15, 10 or 15, 20) to either `null` reference and 2 possible ways to go to a `null` reference.

– Node `30` has 3 black nodes, or 5 different paths (30, 15, 20 or 30, 15, 10 or 30, 70, 85 or 30, 70, 50 or 30, 70, 65) to the `null` reference.

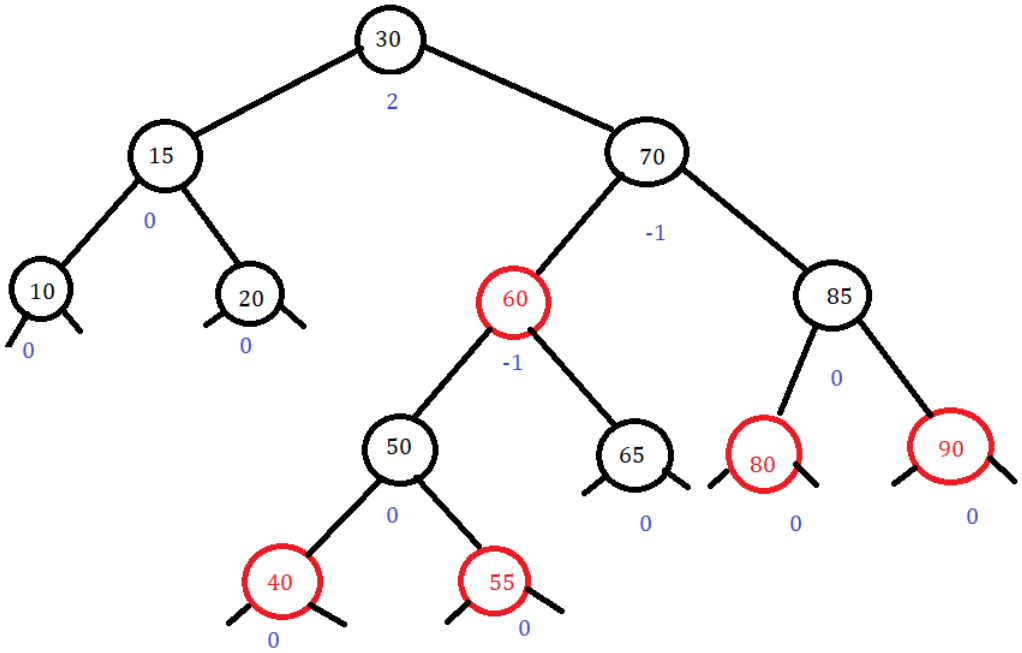
So, this is a red-black tree.

8.2 Red-Black Trees vs. AVL Trees

Recall the same red-black tree from above. However, is that red-black tree a valid AVL tree? Well:



From this, it is indeed a valid AVL tree. However, is this the case for all red-black tree? Consider the same tree as above, but with the only difference being that we removed Node `5`. This would look like:



This is still a red-black tree because it satisfies all the properties. In particular, we note that:

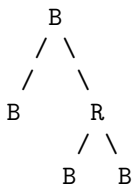
- $30 \to 15 \to 10$
- $30 \to 15 \to 20$
- $30 \to 70 \to 50$
- $30 \to 70 \to 65$
- $30 \to 70 \to 85$

And it follows that this is still a red-black tree. **However**, it is not a valid AVL tree.

8.3 Proof of Red-Black Tree Worst-Case Time Complexity

We said that the worst-case time complexity to find an element in a red-black tree is $O(\log n)$. We now need to prove that this is the case.

Proof. Denote $bh(x)$ to be the number of black nodes from x to a leaf node (excluding itself). Consider the following red-black tree:



Here, we note that:

- $bh(\text{root}) = 1$. We exclude the root node when counting the number of nodes, so there is only one other black node.
- $bh(\text{leaf}) = 0$. We cannot count the leaf node itself since we cannot count the initial node.
- $bh(R) = 1$. There are two possible paths from it to a leaf, and on each path there is exactly one black node.

Our first claim is: any subtree rooted at x has at least $2^{bh(x)} - 1$ internal nodes. To prove this, we will use induction.

- Base Case: Consider $bh(x) = 0$. This happens when x is a leaf node. So, $2^0 - 1 = 1 - 1 = 0$. This works because the black height of a leaf is 0, and the subtree rooted at a leaf has 0 internal nodes because that subtree only contains that leaf itself.
- Inductive Step: Let's assume that this claim holds true if the black height is less than $bh(x)$. We now want to show taht this is the case for $bh(x)$. To do so, we need to consider several scenarios (note that x is the root node of the subtree):
  - If x is black and both of its children are black, then it follows that x has $bh(x)$ black height and both child nodes have $bh(x) - 1$ black height.
  - If x is black but it has at least one red child, then the red child would have a black height of $bh(x)$ given that x has a black height of $bh(x)$. This is because we do not even count the red node when counting the black height. Whatever number of black nodes we passed from x to a leaf (excluding x), we must have passed through the same exact number of nodes through the red child node.

– If x is red, then we know that its children must be black (by definition). If x has a black height of $bh(x)$, then both of x's child nodes will have a black height of $bh(x) - 1$ (note that we need to exclude the black node itself when we are calculating its black height).

Basically, the number of internal nodes in any possible subtree of x is:

$$\underbrace{2^{bh(x)-1} - 1}_{\text{From one child}} + \overbrace{2^{bh(x)-1} - 1}^{\text{From one child}} + \underbrace{1}_{x \text{ itself}} \geq 2^{bh(x)} - 1$$

It follows that the claim holds true even in the generalization.

Now, denote h to be the height of the tree. In general, at least half of the nodes on any path from the root to the leaf must be black. We are guaranteed that:

$$bh(x) \geq \frac{h}{2}$$

$$n \geq 2^{\frac{h}{2}} - 1$$

Therefore:

$$n + 1 \geq 2^{\frac{h}{2}}$$

This implies that:

$$\log(n+1) \geq \frac{h}{2}$$

So, we get that:

$$h \leq 2\log(n+1)$$

In other words, h is $O(\log n)$. □

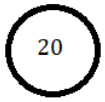
8.4 Red-Black Tree Insertion

We need to consider a few cases.

8.4.1 Insertion Case 1: Empty Tree

For this, we insert the new node as the root. Then, color that node **black**.

If we insert `20` into the red-black tree, then we are left with the tree:



8.4.2 Insertion Case 2: Non-Empty

We need to do the following:

- Perform regular BST insertion. If you ever see a black node with 2 red children, recolor all three (make the parent red and the children black). In other words:

```
  B                R
 / \      ----->   / \
R   R             B   B
```

If the parent is the root, color it black.

- Color the new node **red**.
- Potentially fix the tree for red-black tree properties. This is where we need to consider the potential cases.
  - Case 1: Child of Black Node

    Consider the following tree:

```
...          (This is part of a tree)
  \
   B
```

    If the new node that we are about to insert is the child of a black node, then we're done. This is because the new node will be red by default, so:

```
...          (This is part of a tree)
  \
   B
  /
 R
```

    Here, we note that the new node is on the left. However, if the new node is on the right, we're still done since we didn't violate anything.
  - Case 2: Child of Red Node, Straight Line

    Here, we need to:

    1. Insert
    2. Single Rotation
    3. Recolor

    So, consider the following tree:

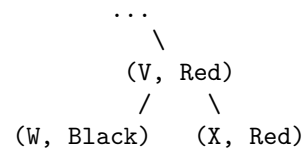
```
...
  \
 (W, Black)
    \
   (V, Red)
```

    If we added a node X, then it would look like[4]:

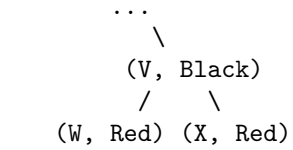
```
...
  \
 (W, Black)
    \
   (V, Red)
      \
   (X, Red)
```

    We now need to perform a left AVL rotation on W and V, like so:

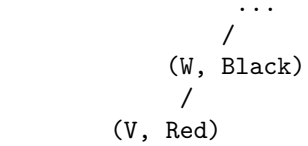
[4]Note that the letters are some variables.



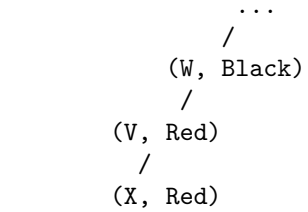
Now, we need to recolor those nodes.



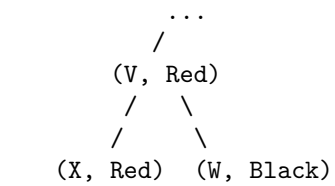
Likewise, if we had the following tree:



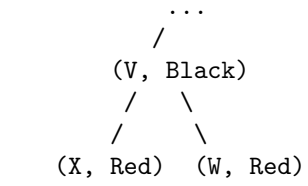
And we added a variable X, it would look like:



We would perform a right AVL rotation on W and V, like so:



And then to recolor these nodes:



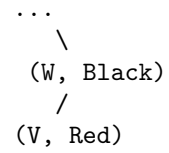
– Case 3: Child of Red Node, Kink

Here, we need to:

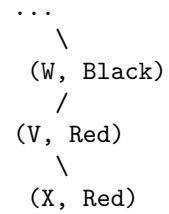
1. Rotate to make straight line.

2. Perform straight line insertion case.

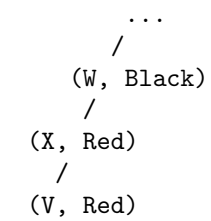
So, consider the following tree:



And suppose we wanted to add X, which will be a right child of V:

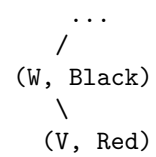


Here, we have a kink-shape. So, we need to perform a left AVL rotation on V and X:

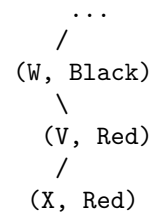


Now, we can do the straight-line fix (from the previous part). First, we perform a right AVL rotation on X and W and then update the colors.

Now, consider the following tree:



And suppose we wanted to add X, which will be a left child of V:



We will perform a right AVL rotation on V and X:

```
(W, Black)
   \
  (X, Red)
     \
    (V, Red)
```

And then we can do the straight-line fix. This involves performing a left AVL rotation on W and X followed by updating the colors.

9 Set and Map ADTs

Now, we'll talk more about set and map ADTs (abstract data types).

9.1 The Set ADT

A set abstract data type is one that stores multiple elements (keys), like an array. It has the following operations:

- `find(x)`: `true` if `x` exists in this set and `false` otherwise.
- `insert(x)`: Add `x` to the set. So, whether or not `x` was in the set, this operation will make sure `x` is in said set.
- `remove(x)`: Removes `x` from the set.

Notice how we have not discussed any implementation details yet, or what to do if any of these fail (aside from `find`). What if we call `insert` with an element that already exists in the set? Or, what if `remove` is called with an element that doesn't exist in the set?

In other words, simply think of a set as a bag of items that has some number of unique elements; you can check if the element exists, insert an element, remove an element, and so on.

9.2 The Map ADT

A map abstract data type is one that stores multiple (key, value) pairs. It has the following operations:

- `get(k)`: Returns the value associated with key `k` if `k` exists in the map.
- `put(k, v)`: Maps the key `k` to the value `v`.
- `remove(k)`: Removes the key `k` and its value from the map.

Once again, we have not discussed any implementation details; that is left to whatever ultimately implements this abstract data type. For instance, if we called `get` on an key `k` that doesn't exist, then should the method return `nullptr`? Or throw an error? How about `put`? What if the key `k` already exists in the map? Should we update the value? Or should we throw an error?

For instance, suppose we have a map called `students`. Our key could be the student names and the value could be the grades. It would look roughly like:

```
names (k)       grades (v)
-------------------------
Niema      ->   A+
Felix      ->   A
Ryan       ->   A
```

So, with respect to the *keys*, this is essentially a set. In other words, we can think of a map as a set with an associated value.

9.3 Implementing the Set and Map ADT

How can we implement the set and map ADT? We will only briefly discuss how we can implement the set ADT since we can easily transform a set ADT implementation to a map ADT implementation (by storing the key *and* the value as opposed to just the key). The converse is also true; that is, we can transform a map ADT implementation to a set ADT implementation by using dummy values for the values.

- Unsorted Linked List: $O(n)$ find/remove, $O(1)$ insert.

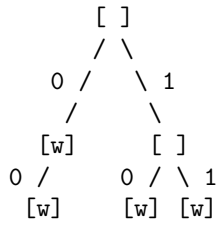
- Sorted Linked List: $O(n)$ find/remove/insert, can iterate in sorted order.
- Unsorted ArrayList: $O(n)$ find/remove, amortized $O(1)$ insert.
- Sorted ArrayList: $O(\log n)$ find (binary search), $O(n)$ remove/insert, can iterate sorted.
- Self-Balancing BST: $O(\log n)$ find/insert/remove, can iterate sorted.
- Hash Table: $O(1)$ expected, need to perform $O(k)$ hash where k is a constant representing the length of the key.

10 Multiway Tries

In many tree structures, we store elements in the **nodes** of a tree. Each node represents a single element. A *trie* is a bit different, though.

10.1 Trie

A **trie** is a tree structure in which elements are represented by **paths**. Consider the following *binary trie* (a binary trie is a trie that is a binary tree):



Here, we don't even look at the nodes (value-wise). Rather, consider the path from the root node to the left-most leaf node. Here, we note that this path from the root to that leaf node via the edges forms a string of `00`. Nodes that are denoted `[w]` are word nodes; think of them as separators.

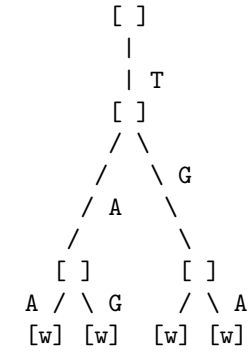
If we take the path to the left-most node and then to the right path followed by the left path, and then the right path fully, we note that this forms the words:

```
0, 00, 10, 11
 ^   ^   ^
(Word Nodes)

0
0 -> 0
1 -> 0
1 -> 1
```

10.2 Multiway Tries

A **multiway trie** is a trie in which nodes can have more than **two children**.

Consider the following multiway trie over the DNA alphabet:



Where the DNA alphabet is:

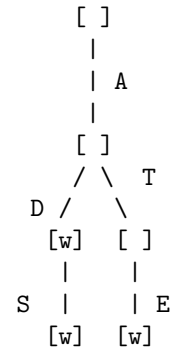
$$\{A, C, G, T\}$$

The word nodes are denoted by `[w]`. We note that the words that can be represented are:

- TAA

- TAG
- TGA

Now, consider the following multiway trie over the English alphabet:



The words that are stored are as follows:

- AD
- ADS
- ATE

So, does the word `AT` exist in this multiway try?

- The answer is no. While there is a path that consist of `AT`, this does not stop at a word node. So, you have `ATE`, but not `AT`.

10.3 MWT Insertion, Finding, and Removing

Insertion works like so:

- Start at the root.
- For each letter in the word that we are inserting:
  - Check if the current node has a child edge labeled by that letter.
  - If it does not, create new child edge labeled by letter.
  - Traverse down to the node that the end of that child edge.
- Once we finish that word, we mark the current node as a word node.

Finding works like so:

- Start at the root.
- For each letter in the word that we are inserting:
  - Check if the current node has a child edge labeled by that letter.
  - If it does not, the word does not exist and we can return.
  - Traverse down to the node that the end of that child edge.
- Check if the node is a word node. If it is, the word is found. Otherwise, it is not found.

Deleting works like so:

- Start at the root.

- For each letter in the word that we are inserting:
  - Check if the current node has a child edge labeled by that letter.
  - If it does not, the word does not exist and we can return.
  - Traverse down to the node that the end of that child edge.
- Check if the node is a word node. If it is, unmark it as a word node (so that it doesn't form a word).

11 Ternary Search Trees

A ternary search tree serves a similar purpose to a multiway trie. However, performance-wise, there are differences.

- BST: $O(k \log n)$, memory efficient.
- MWT: $O(k)$, memory inefficient.
- TST: Somewhere in between. It stores words similarly to a MWT, but with less wasted speed.

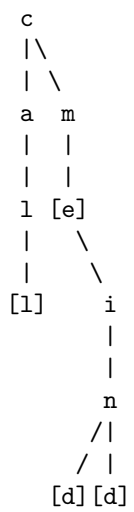
One notable difference between MWTs and TSTs is how we use **nodes** to store a letter.

11.1 TST Find Algorithm

The find algorithm for a ternary search tree works like so:

- Start at the root node. Denote N as the node that we are at and N_l as the letter at this node.
- For each letter l in the query:
  - If $l > N_l$, then traverse to the right child node.
  - Else, if $l < N_l$, then traverse to the left child node.
  - Else, we do the following:
    - If l is the last letter of the query and N is a word node, then we found the word.
    - Else, traverse to the middle child.
- At this point, the word wasn't found, so it doesn't exist.

For instance, consider the following ternary search tree:



Let's suppose I wanted to find `mind`.

- We start at the root node `c`, and focus on the first letter of the query `m`. Since `m > c`, we traverse right.
- Now, we're at the node `m`. We are still focusing on the first letter `m`. Since `m == m`, we traverse down.
- Now, we're at the node `e`. We are now focusing on the second letter of the query `i`. Since `i > e`, we traverse right.
- Now, we're at the node `i`. We are still now focusing on the second letter of the query `i`. Since `i == i`, we traverse down.

- Now, we're at the node `n`. We are now focusing on the third letter of the query `n`. Since `n == n`, we traverse down.
- We are now at the node `d`. We are focusing on the the fourth letter `d`. But, since `d == d` and this is the last letter of the query, we found it.

11.2 TST Insert Algorithm

The idea behind this algorithm is very similar in nature to the TST find algorithm. The notable differences are:

- If we need to traverse to a child that doesn't exist, simply create it and traverse.
- Make the last word in the traversal a word node.

11.3 TST Remove Algorithm

Again, the idea behind this algorithm is very similar in nature to the TST find algorithm. The only difference is:

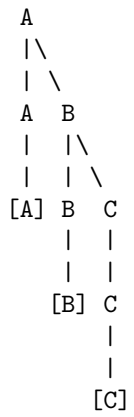
- Make the last node in the traversal not a word node.

11.4 TST Time Complexity

The time complexity for a TST is:

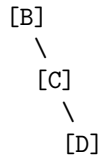
- $O(n)$ worst case.
- $O(\log n)$ average case.

For instance, we can get the worst case if we inserted the words `AAA`, `BBB`, and `CCC` (in sorted order). The tree would look like:



Which is essentially just a linked list.

One thing to note is that a TST works extremely well if the words all have very similar prefixes. Suppose we have the words `AAA`, `AAB`, `AAC`, and `AAD`. Then, our TST would look like:


[B]
[C]
[D]

12 Hashing, Hash Tables, Hash Maps, and Collisions

Now, we're going to talk about hash functions, hash tables and maps, and collisions.

12.1 Hash Functions

A hash function **takes in** an object x and **returns** an integer representation of x.

There are a few properties to consider:

- **Property of Equality:** If x is equal to y, then $h(x)$ must equal $h(y)$.
- **Property of Inequality:** If x is not equal to y, then it would be nice (but not necessary) if $h(x)$ was not equal to $h(y)$.

Remark: The only property that matters is the property of equality. In other words, a valid hash function *must* satisfy the property of equality, but not necessarily the property of inequality. *In other words*, a valid hash function is one that satisfies the property of equality, but a good hash function is one that also satisfies the property of inequality.

12.1.1 Example 1: Trivial Hash Function

Consider the hash function, where `s` is a string:

```
h(s):
    return 0
```

For any word, we will always get back `0`. This is a **valid** hash function since it always satisfies the property of equality. However, this isn't very good because two different strings will give the same hash value.

12.1.2 Example 2: Good Hash Function

Consider the hash function, where `s` is a string:

```
h(s):
    out = 0
    for c in s:
        out += ASCII value of c
    return out
```

This is a **valid** hash function since it always satisfies the property of equality. It's also better than the previous hash function since it does differentiate between some strings. However, if we gave the string $x_1x_2x_3$ and $x_2x_3x_1$ where x_1, x_2, and x_3 are arbitrary characters, we would get the same hash value. So, there's room for improvement.

12.1.3 Example 3: Better Hash Function

Consider the hash function, where `s` is a string:

```
h(s):
    out = 0
    for c in s:
        out *= 31
        out += ASCII value of c
    return out
```

This is a **valid** hash function since it always satisfies the property of equality. This is also significantly better than the previous hash function because two different words with the same letters (shuffled around) will, in general, give us a different hash value.

12.1.4 Example 4: Invalid Hash Function

```
h(s):
    return a random integer
```

This is an **invalid** hash function since it doesn't always satisfy the property of equality.

12.1.5 Example 5: Invalid Hash Function

```
h(s):
    return current time
```

This is an **invalid** hash function since it doesn't always satisfy the property of equality.

12.2 Hash Tables

A **hash table** is essentially an array that can hold values. As the name implies, we make use of a hash function to determine where we put the values.

Suppose we have the following hash table:

```
[_, _, _, _, _]     Array
 0  1  2  3  4      Index
```

And suppose A has hash value $h(A) = 65$, C has hash value $h(C) = 67$, and G has hash value $h(G) = 71$. We can take the hash value and mod it by the length of the array to get the appropriate index.

For example, to insert A to our hash table, take its hash ($h(A) = 65$) and mod it by 5 (the length of the array). This gives us 0, so:

```
[A, _, _, _, _]     Array
 0  1  2  3  4      Index
```

Now, to insert C to our hash table, take its hash (67) and mod it by 5. This gives us 2, so:

```
[A, _, C, _, _]     Array
 0  1  2  3  4      Index
```

To see if an element exists, we can use the hash function to get the hash, mod it by the length of the array to get the index, and then go to that index of the array and check if an element exists and if it's equal to our target element. We can use this same process to remove elements as well.

As a result, one can say that this implements the set ADT.

12.3 Hash Maps

A hash map, like a hash table, is an array that can hold values. A hash map is like a map; instead of inserting individual elements, we can insert a key with a corresponding value. Instead of checking if an element exists, we can retrieve or remove an element associated with a key.

A hash map has some array behind the scene. For example:

```
[_, _, _, _, _]     Array
 0  1  2  3  4      Index
```

Let's say this hash map will map student names to student grades. Suppose the hash values for `Niema` and `Ryan` are as follows:

```
h(Niema) % 5 = 2
h(Ryan) % 5 = 0
h(Felix) % 5 = 3
```

Suppose we want to insert `Niema` with value `A+`. Then:

```
[_, _, Niema (A+), _, _]     Array
 0  1  2           3  4      Index
```

Suppose we now want to insert `Ryan` with value `A`. Then:

```
[Ryan (A), _, Niema (A+), _, _]     Array
 0         1  2           3  4      Index
```

Now, let's suppose we wanted to know `Ryan`'s grade. Take the hash of `Ryan`, mod it by the length of the array, and get the corresponding value. In our case, we would get `A`.

Now, suppose we wanted to know `Felix`'s grade. Then, hashing `Felix` and modding it by the length of the array gives us `3`, but there's nothing at index 3 of the backing array. So, `Felix` must not exist.