CSE 100 Notes

Advanced Data Structures

A "Condensed" Study Guide

Fall 2021
Taught by Professor Niema Moshiri

Table of Contents

1 Introduction to C++

In this section, we will talk very briefly about C++. We will purposely be omitting:

- Templates
- Vectors
- Input/Output
- Some examples.

1.1 Data Types

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

Remark: In C++, you can have both signed and unsigned data types.

1.2 Strings

Some quick facts about C++ strings in general:

- Strings are represented by the `string` type.
- Strings are actually mutable. You can modify strings in-place.
- 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).
- We can take the substring using the `substr` method. The method signature is:

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

  Keep in mind that C++'s `substr` method takes in a **length** for the second parameter.

1.3 Comparing Non-Primitive Objects

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. A common example of this can be found with *iterators*.

1.4 Variables

Here, we will give a brief overview of variables.

1.4.1 Initialization

Variable initialization in C++ 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.

1.4.2 Narrowing

In C++, we are able to *freely* cast a "higher" variable type to a "lower" one. 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.

1.4.3 Declaration

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.

1.5 Classes

Now, we will very briefly go over classes in C++.

1.5.1 Class Declaration

A typical class in C++ looks 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 */ }
```

A few observations can be made about how classes are laid out.

- 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.
- In C++, we can "declare" the methods and the constructor, and then outside of the class we can implement the methods.

1.5.2 Constructor Shortcut

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;
}
```

This is equivalent to the following:

```
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**, and can be used to directly initialize the instance variables.

1.6 Memory Management

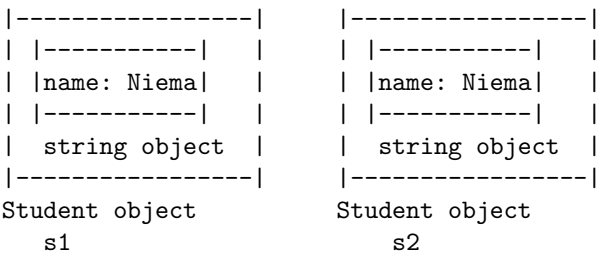
Here, we will talk about memory management in C++.

1.6.1 Class Copies

Suppose we have the following C++ code:

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

Here, `s1` creates a new `Student` instance. `s2` copies the `s1` instance entirely; in other words, it creates a new, separate object with the same properties as `s1`. The memory diagram for these are shown below:

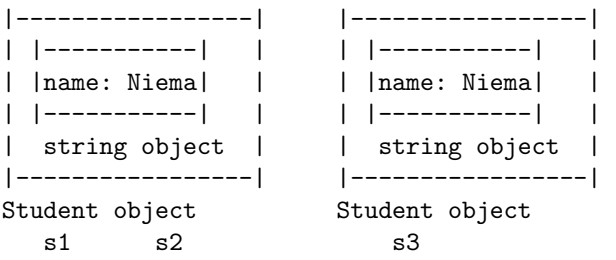


1.6.2 References

Consider the following C++ code:

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

The associated memory diagram would look like:



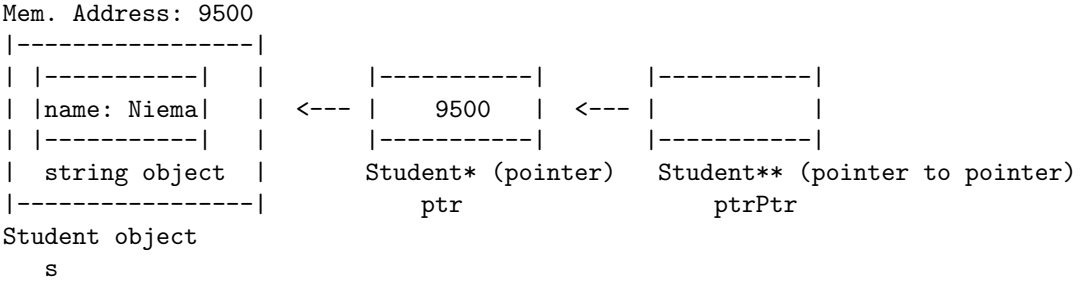
Here, `s2` can be seen as *another* way to call `s1` (think of `s2` as another name for `s1`). `s3` would be a copy of `s1`.

1.6.3 Pointers

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;
```

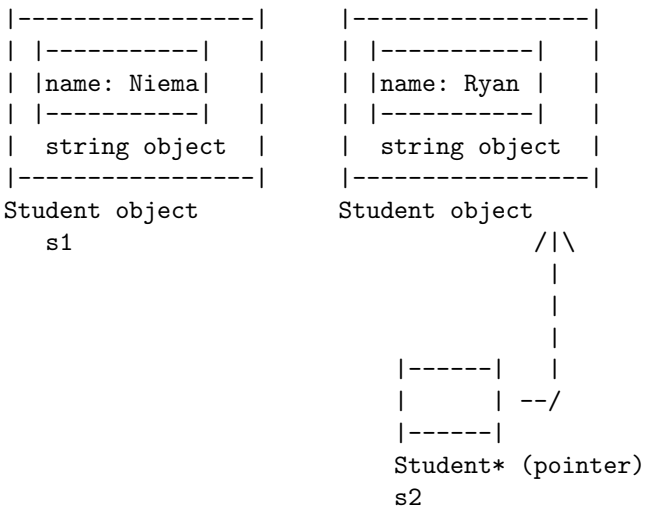
2. Arrow dereferencing.

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

1.6.4 Stack/Heap and Dynamic Memory Management

```
// Same thing as saying
// Student s1(``Niema'');
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**.

The variable, `s2`, is **not** allocated on the heap. However, the object that it is pointing to **is** allocated on the heap.

1.7 Constant Keyword

Now, we will talk about the `const` keyword and its significance.

1.7.1 The Basics

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

1.7.2 `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.

1.7.3 `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).

1.7.4 `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.

1.8 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 */
}
```

1.8.1 Passing by Value

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).

1.8.2 Passing by Reference

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.

1.9 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**.

1.9.1 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`.

1.9.2 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:

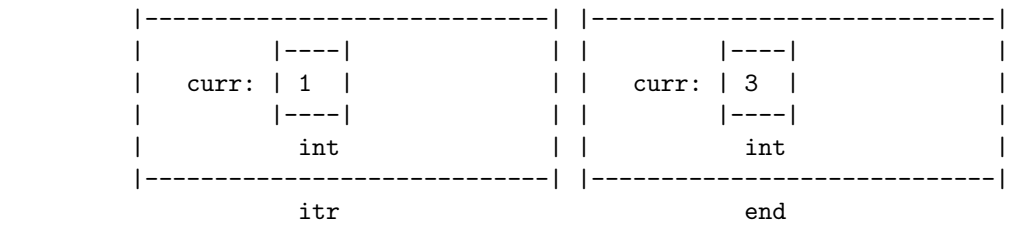
```
    0       1       2           // Index
['‘Niema’', '‘Ryan’', '‘Felix’']      // names array
```

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:



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`.

1.9.3 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()
```

2 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.

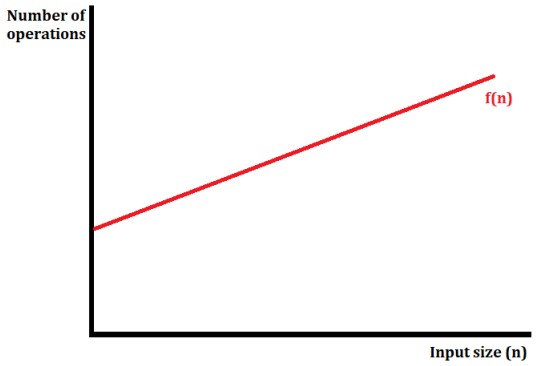
That being said, we want to know how fast an algorithm is. The best way to do so is by figuring out the performance in terms of number of operations with respect to the input size n (instead of the amount of time).

2.1 Notation of Complexity

Consider the following notations:

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

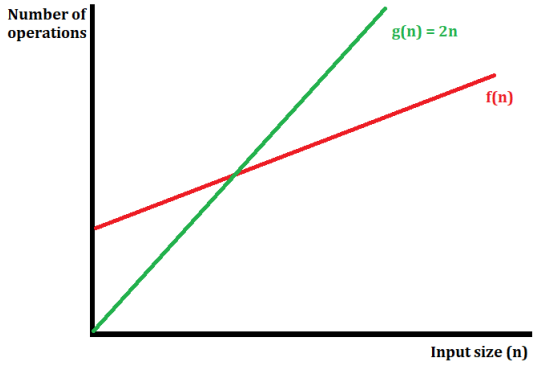
Consider the following graph:



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 $n \to \infty$.

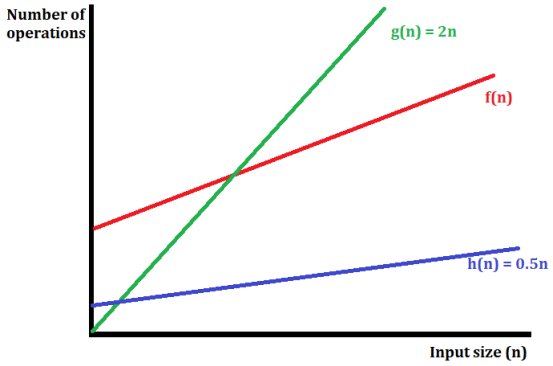
  Consider the following graph:



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.

2.2 Common Big-O Time Complexity

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

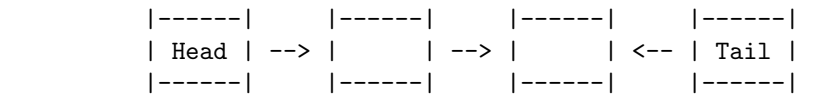
Big-O	Common Name	Notes
$O(1)$	Constant	The time complexity does not depend on the input size n.
$O(\log n)$	Logarithmic	If the input size is doubled, the number of operations is increased by a constant. One common example is binary search: if we have 8 elements, it would take 3 operations. Doubling 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."

2.3 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:

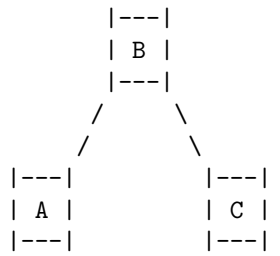
- c is a constant that represents the number of bytes needed to store the head and tail pointer.
- k is a constant that represents the number of bytes needed to store a node.

3 Trees

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

3.1 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.

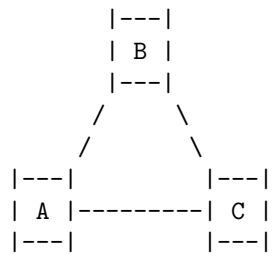
So, a linked list is essentially a chain of nodes in sequence. It has n nodes and $n-1$ edges.

3.2 What are Trees?

A tree is a graph with two properties:

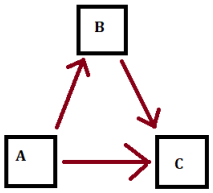
- It has no undirected cycles.

  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:

```
A ---- B

C ---- D
```

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

3.3 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.

3.4 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). 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).
  - A node is considered to be a **leaf** node if it doesn't have any children.
  - A node is considered to be an **internal** node if it does have children.
- 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:

```
2             6
 \           /
  \         /
   1 ------ 4
  /         \
 /           \
3             5
```

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.

3.5 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.

3.6 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 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.

4.1 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.

4.2 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 (stop here).
3. If `find` doesn't succeed, insert the new element at the site of failure.

4.3 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. 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:

- If the node has a right child, traverse right once, then all the way left.
- Otherwise, traverse up the tree. The first time the current node is its parent's left child, the parent is our successor.

4.4 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.

4.5 Height of a Node and Tree

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

4.6 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.

4.7 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 $O(n)$.

4.7.1 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.

4.7.2 Depth of a Node

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

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

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.

5 Treaps and Randomized Search Trees

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

5.1 Treap

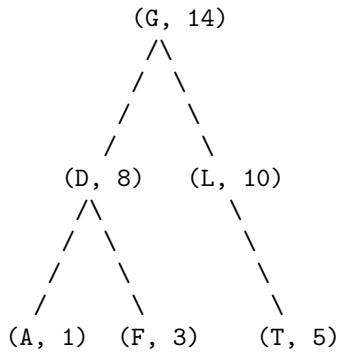
A **treap** is a special tree data structure. The name comes from how this is a *tree* data structure that makes use of a heap. In particular, a treap stores a `(key, priority)` pair.

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:

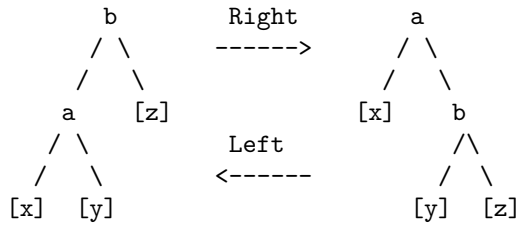
- We know that 14 is larger than all other nodes (its children); it's also at the top of the tree.
- We know that 8 is larger than its children nodes ($1 < 3 < 8$).
- We know that 10 is larger than its child node ($10 > 5$).

So, it does fulfill the Heap properties.

5.2 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**.

As a final note, we also consider *kinks*, which require two AVL rotations. In particular, if the nodes are in a kink shape, like so:

```
  a           a       Out-of-balance node.
 /             \
b               b     Heavy child node.
 \             /
  c           c       Heavy grandchild node.
```

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.

5.3 Treap Insertion

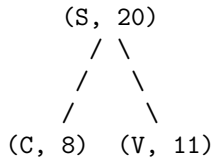
To insert a new key/priority pair:

(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.

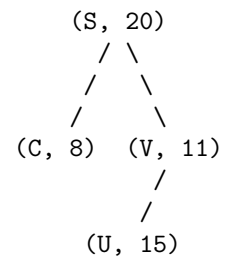
5.3.1 Example 1: Simple Treap Insertion

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

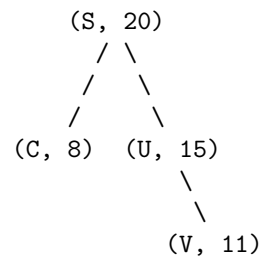


And let's suppose we wanted to insert $(U, 15)$.

(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:

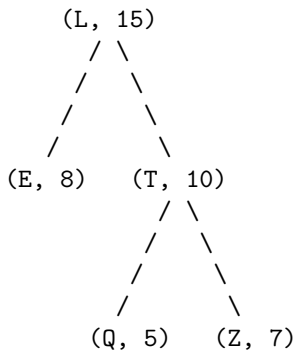


(2) By inserting this pair, though, we have violated the heap properties. So, we're going to do a **right AVL rotation** on U and V so that U becomes the new parent. So:



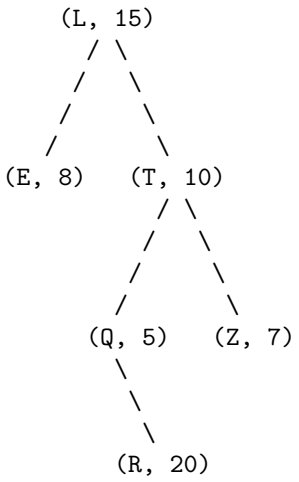
5.3.2 Example 2: Slightly Harder Treap Insertion

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

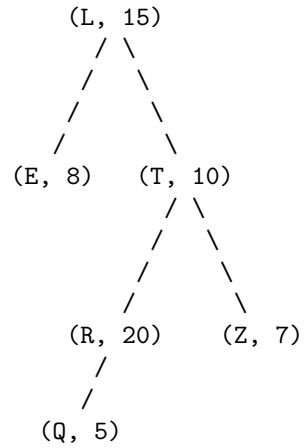


And let's suppose we wanted to insert $(R, 20)$.

(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:



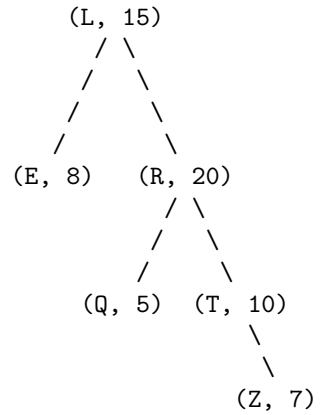
(2) By inserting this pair, though, we have violated the heap properties. So, we're going to do a **left AVL rotation** on Q and R so that R becomes the parent and Q the child.



The changes we made are:

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

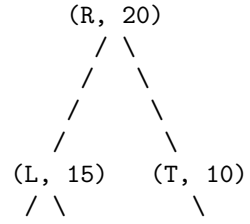
(3) However, in this position, we are still violating the heap properties. So, we need to do another a **right AVL rotation** on R and T.

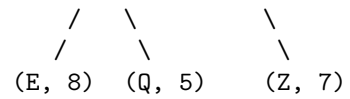


The changes we made are:

- 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.

5.4 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.

Again, a randomized search tree still has $O(n)$ worst case time complexity because our key/priority pairs could have been formed in a straight linear fashion, and so on. So, we could still get unlucky with the randomly generated numbers.

6 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 $O(\log n)$?

6.1 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.

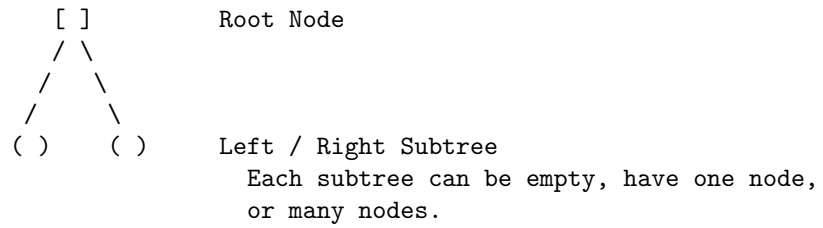
Before we talk about an AVL tree, we need to talk about some concepts.

- 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.

6.2 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:



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:

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

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:

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

□

6.3 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 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.

7.1 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.

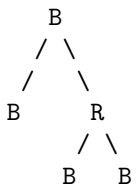
7.2 Red-Black Trees vs. AVL Trees

A valid red-black tree is not necessarily a valid AVL tree.

7.3 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 $O(\log n)$. □

7.4 Red-Black Tree Insertion

We need to consider a few cases.

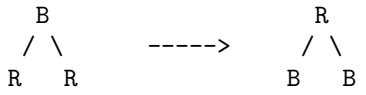
7.4.1 Insertion Case 1: Empty Tree

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

7.4.2 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:



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 If the new node that we are about to insert is the child of a black node, then we're done.
  - Case 2: Child of Red Node, Straight Line
    Here, we need to:
    1. Insert the node, like normal.
    2. Perform a single AVL rotation on the upper two nodes.
    3. Recolor the nodes.
  - Case 3: Child of Red Node, Kink
    Here, we need to:
    1. Rotate to make straight line.
    2. Perform straight line insertion case.

8 Set and Map ADTs

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

8.1 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.

8.2 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.

8.3 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.

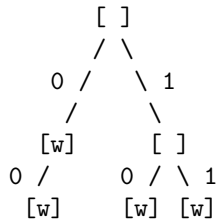
- Sorted Linked List: $O(n)$ find/remove/insert, can iterate in sorted order.
- Unsorted ArrayList: $O(n)$ find/remove, amortized $O(1)$ insert.
- Sorted ArrayList: $O(\log n)$ find (binary search), $O(n)$ remove/insert, can iterate sorted.
- Self-Balancing BST: $O(\log n)$ find/insert/remove, can iterate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

9 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.

9.1 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. In other words, we can choose to *stop* at that node (and get the resulting word) or we can continue until we hit another word node.

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:

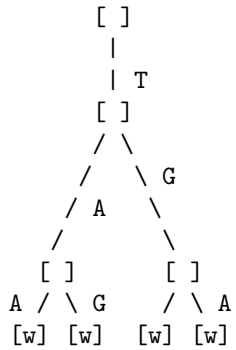
```
0, 00, 10, 11
 ^   ^   ^
(Word Nodes)
```

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

9.2 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:



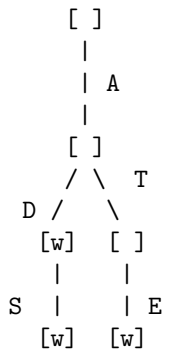
Where the DNA alphabet is:

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

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`.

9.3 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).

10 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.

10.1 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.

10.2 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.

10.3 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.

10.4 TST Time Complexity

The time complexity for a TST is:

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

11 Appendix

Here's some useful information that might be helpful.

11.1 Lecture and Stepik Links

These all link to the corresponding lectures on Edstem and Stepik. You may need to be logged in.

Num.	Lecture Title	Edstem Link	Stepik Link
1	Course Introduction	Click Here	N/A
2	C++ Review	Click Here	Click Here
3	C++ Iterators	Click Here	Click Here
4	Time and Space Complexity	Click Here	Click Here
5	Trees	Click Here	Click Here
6	Binary Search Trees	Click Here	BSTs/BST Avg. Time-Complex
7	Treaps and Randomized Search Trees	Click Here	Heaps/RSTs
8	AVL Trees	Click Here	Click Here
9	Red-Black Trees	Click Here	Click Here
10	Set and Map ADTs	Click Here	Lexicon/LL/Array/BSTs/HT/HM
11	Multiway Trie	Click Here	Click Here
12	Ternary Search Tree	Click Here	Click Here

11.2 Visualization Links

Click https://www.cs.usfca.edu/ galles/visualization/Algorithms.html to see all visualizations. Some important ones:

Data Structure	Link
Binary Search Tree	Click Here
AVL Trees	Click Here
Red-Black Trees	Click Here
Trie	Click Here
Ternary Search Tree	Click Here

11.3 Time/Space Complexities

Here are some time/space complexities, courtesy of Wikipedia.

11.3.1 Binary Search Tree

Algorithm	Average	Worst
Search/Find	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

Space Complexity:

- Average: $O(n)$
- Worst: $O(n)$

11.3.2 Treap/RST

Algorithm	Average	Worst
Search/Find	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

Space Complexity:

- Average: $O(n)$
- Worst: $O(n)$

11.3.3 AVL Tree

Algorithm	Average	Worst
Search/Find	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

Space Complexity:

- Average: $O(n)$
- Worst: $O(n)$

11.3.4 Red-Black Tree

Algorithm	Average	Worst
Search/Find	$O(\log n)$	$O(\log n)$
Insert	Amortized $O(1)$	$O(\log n)$
Delete	Amortized $O(1)$	$O(\log n)$

Remark: The below quote is from Stepik.

> The balance restrictions of the AVL Tree are stricter than those of the Red-Black Tree, so even though the two data structures have the same time complexities, in practice, AVL Trees are typically faster with find operations (because they are forced to be more balanced, so traversing down the tree without modifying it is faster) and Red-Black Trees are typically faster with insertion and removal operations (because they are not required to be as balanced, so they perform less operations to maintain balance after modification).

Space Complexity:

- Average: $O(n)$
- Worst: $O(n)$

11.3.5 Multiway Trie

Algorithm	Worst
Search/Find	$O(k)$
Insert	$O(k)$
Delete	$O(k)$

Here, k is the length of the word.

Space Complexity:

- Worst: $O(\Sigma^k)$, where Σ is the size of the alphabet and k is the size of a word.

11.3.6 Ternary Search Tree

Algorithm	Average	Worst
Search/Find	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

Space Complexity:

- Average: $O(n)$
- Worst: $O(n)$