untitled napkin thing

Samuel Hsiang

July 2015

Some cheesy dedication thing to fill out once I'm done.

Contents

Preface

You might have heard of Evan Chen's Napkin, a resource for olympiad math people that serves as a jumping point into higher mathematics. (In fact, I'm using Evan's template right now. Thanks Evan!) The Wikipedia articles on higher mathematics are just so dense in vocabulary and deter many smart young students from learning them before they are formally taught in a course in college. Evan's Napkin aims to provide that background necessary to leap right in.

I feel the same way about computer science. For most, the ease of the AP Computer Science test means that the AP coursework is often inadequate in teaching the simplest data structures, algorithms, and big ideas necessary to approach even silver USACO problems. On the other hand, even the best reference books, like Sedgewick, are far too dense and unapproachable for someone who just wants to sit down and learn something interesting. The road, for many, stalls here until college.

Everyone should be able to learn the simplest data structures in Java or C++ standard libraries reasonably quickly, and someone with problem-solving experience can easily jump right into understanding algorithms and more advanced data structures. So here goes.

A Few Important Notes...

1. I'm assuming some fluency in C-style syntax. If this is your first time seeing code, please look somewhere else for now. I may or may not add an introduction to object-oriented programming later.

2. It is essential that you understand the motivations and the complexities behind everything we cover. I feel that this is not stressed at all in AP Computer Science and lost under the heavy details of rigorous published works.

3. I'm avoiding what I call the heavy details because they don't focus on the math behind the computer science and lose the bigger picture. My goal is for every mathematician or programmer, after working through this, to be able to code short scripts to solve problems. Once you understand how things work, you can then move on to those details which are necessary for building larger projects. The heavy details become meaningless as languages develop or become phased out. The math and ideas behind the data structures and algorithms will last a lifetime.

4. You really should actually code up each data structure with its most important functions or algorithm as you learn them. I truly believe the only way to build a solid foundation is to code. Do not become reliant on using the standard library (`java.util`, for instance) without understanding how the tool you are using works.

5. This by no means represents everything you will ever have to know for competitive computer science. Expect this to grow as I learn more.

Chapter 1

Fundamentals

1.1 Comparable

The interface `Comparable` is how Java can impose an ordering on any object. Implementing `Comparable` requires support to the method `int compareTo(Object o)`. If `a.compareTo(b) < 0`, then `a` should come before `b` in the ordering. If `a.compareTo(b) > 0`, then `a` should come after `b` in the ordering. If `a.compareTo(b)==0`, then `a` is considered to be equal to `b`. `a.compareTo(b) == 0` should be equivalent to `a.equals(b)`. For example, consider the following `class MyPair`:

```
class MyPair implements Comparable {
    int x;
    int y;
    // sort by x-coordinate first
    public int compareTo(Object o) {
        MyPair c = (MyPair) o;
        if(x < c.x) return -1;
        if(x > c.x) return 1;
        // if x-coordinates equal, compare y
        if(y < c.x) return -1;
        if(y > c.x) return 1;
        return 0; // equal
    }
}
```

We can also clean this up a bit using generics:

```
class MyPair implements Comparable<MyPair> {
    int x;
    int y;
    public int compareTo(MyPair c) {
        if(x < c.x) return -1;
        if(x > c.x) return 1;
        if(y < c.x) return -1;
        if(y > c.x) return 1;
        return 0;
    }
}
```

Note that no casting is required anymore.

1.2   Sorts

1.2.1   Merge Sort

The idea behind merge sort is simple. If we are given two equally sized sorted stacks, we could easily combine them into one sorted stack by comparing the top of the two stacks only, since the smallest element remaining must be one of those two.

From here, we divide and conquer. Simply cut the array in half, recurse on each half, and merge the two back together. This runs in $O(n \log(n))$ time.

1.2.2   Quicksort

Quicksort also uses a divide and conquer strategy to achieve an $O(n \log(n))$ amortized bound. We take a random element from the array, move anything less than it to the left of the array, anything right of it to the right of the array, and recurse.

This is implemented in `Arrays.sort()` and `Collections.sort()`.

1.3   Binary Search

Before we begin studying data structures, it is necessary to first understand general search techniques. Given a list, it is straightforward to simply check every element in the list in a linear search.

One way we can make this faster is by imposing an ordering on the list. Then inserting elements becomes slower because we can no longer just pop the element to the the end of the list. However, if we are asked to find an element, we know roughly where in the list to search because we can compare the element to other elements in the list.

Consider an array of sorted elements. The array supports $O(1)$ access to any element in the array. We'll take a look at the middle element in the array. Depending on whether or not the element to be searched is larger, smaller, or equal to the middle element, we can guarantee we can eliminate half of the array. Then finding an element is an $O(\log n)$ operation on a sorted array. This can be done either iteratively or recursively.

Chapter 2

Java Standard Library Data Structures

The purpose of this chapter is to provide an overview on how the most basic and useful data structures work. The implementations of most higher-level languages already coded these for us, but it is important to know how each data structure works rather than blindly use the standard library.

More technical explanations of all of these can be found, mostly under the package `java.util`, in the Java API.

I strongly believe that Java is better than C++ for beginning programmers. It forces people into good coding habits, and though the lack of pointers initially frustrated me, it really does make learning general concepts liked LinkedLists much easier, as the intricacies of the C++ pointer no longer distract from the larger idea.

2.1   Generics

You may or may not have seen these used before:

```
ArrayList<Integer> al = new ArrayList<Integer>();
```

This means that `al` is an ArrayList of Integers. We can only add Integers into the ArrayList, and anything removed from the ArrayList is guaranteed to be an Integer. I don't think the beginning programmer needs to know how to necessarily code a class that supports generics. However, you are going to use the standard library, so you should know how to use a class that does support generics. I think standard classes are relatively straightforward to use but can be annoying to actually implement.

**For simplicity, all data structures will store Strings.**

2.2   List

A List is a collection of objects with an ordering. Users of a list have control over where in the list each object is and can access a specific element by its index, like in an array.

2.2.1   ArrayList

What is nice about an array? We can access or change any element we want in the array in $O(1)$ time. The problem is that an array has fixed length. It's not easy to append an element to the end of an array.

The fix to this is pretty simple. Why not just make a bigger array, and copy everything over to the new array? Then there's more room at the end to add a new element. This is the idea behind an ArrayList. If the backbone array runs out of space, we create a new array with double the size and keep going as if nothing happened.

"a"	"b"	null
0	1	2

"a"	"b"	"c"
0	1	2

"a"	"b"	"c"	null	null	null
0	1	2	3	4	5

"a"	"b"	"c"	"d"	null	null
0	1	2	3	4	5
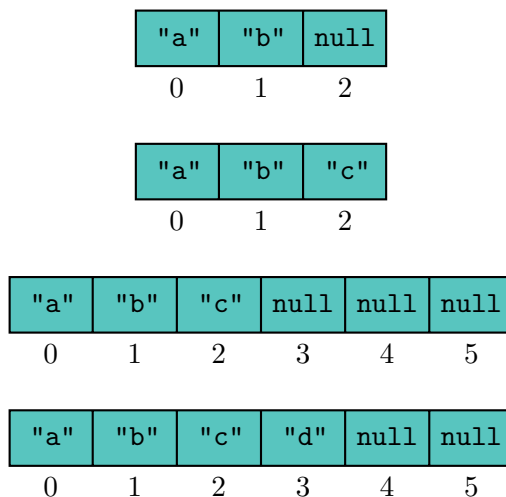
We see that there is still room in the array to add `"c"`, but to add more elements to the list, we must use a new array with double the length.

It's important to note that any given insertion to the ArrayList is either $O(n)$ or $O(1)$, but there is only one $O(n)$ insertion for every $O(n)$ $O(1)$ insertions, so we still average out to constant time.

- `boolean add(String s)` – add an element to the end of the list. (By convention, this returns `true` if the addition was successful, and `false` otherwise.)

- `void add(int i, String s)` – shift everything from position `i` onward down by one, and add `s` at position `i`.

- `boolean contains(Object o)` – return `true` if `o` is stored in the ArrayList, and `false` otherwise. Remember to cast `o` to a String.

- `String get(int i)` – return the element stored at index `i`.

- `boolean isEmpty()` – return `true` if `size() == 0`.

- `String remove(int i)` – remove and return the element at index i from the list. (Why is this annoying?)

- `String set(int i, String s)` – replace the element stored at index `i` with `s`, and return the element that was originally at position `i`.

- `int size()` – note that this is not just the length of the backbone array.

The `get` and `set` functions are very nice. They are easy to code and run in constant time. These are the bread and butter of any array. Adding at the end of the ArrayList is nice as well. `contains` is a pain, as it is $O(n)$, and adding to and removing from early in the list are more annoying.

If this is your first time seeing an ArrayList, I would suggest coding up your own. It should be relatively straightforward.

2.2.2   LinkedList

Arrays are nice for accessing, say, the seventh element in the list. We extend this to an ArrayList to implement adding and removing elements to and from the end of the list nicely. Removing elements from the beginning of the list, however, is cumbersome.

The LinkedList attempts to remedy this. It trades $O(1)$ access to any element in the list for an easier way to remove elements from either end of the list easily. Consider a chain of paper clips:

It's easy to add or remove more paper clips from either end of the chain, and from any given paper clip, it's easy to access the paper clip directly previous or next to it in the chain. If we needed the seventh paper clip in the chain, we'd need to manually count, an $O(n)$ operation. However, if we then needed to remove that paper clip from the chain, it wouldn't be that hard, assuming we kept a finger on the seventh paper clip.

The standard library implements a cyclical doubly-linked list, with a dummy head. It looks something like this:
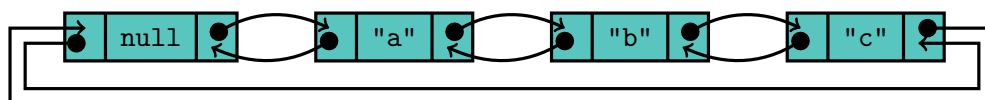
We see that each node maintains a pointer to its next neighbor and its previous neighbor, in addition to containing the String it stores. We can store this data in a class like the following:

```
class ListNode {
    ListNode prev, next;
    String s;
}
```

If we were to insert an element after a ListNode `a`, it is necessary to update all pointers:

```
ListNode b = new ListNode();
b.prev = a;
b.next = a.next;
b.next.prev = b;
a.next = b;
```

Since the LinkedList is symmetric, inserting an element before a node is also easy. To add something to the end of the list, simply add it before the dummy head. From here it should not be too hard to implement all the important functions of a LinkedList.

- `boolean add(String s)` – add to the end.

- `void add(int i, String s)` – this is linear. Don't forget about 0-indexing.

- `void addFirst()`

- `boolean contains(Object o)`

- `String get(int i)`

- `String getFirst()`

- `String getLast()`

- `boolean isEmpty()`

- `String remove()` – remove the last element.

- `String remove(int i)`

- `String removeFirst()`

- `String set(int i, String s)`

- `int size()`

With a LinkedList implemented, two other data structures immediately follow.

2.2.3   Stack

A stack is is literally a stack. If we have a stack of papers, we can *push* things on the top and *pop* things off the top. Sometimes we *peek* at what's on the top but don't actually remove anything. We never do anything with what's on the bottom. This is called *LIFO*: Last In, First Out.

- `String push(String s)` – pushes the item on the top of the stack, and returns the same element.

- `String pop()` – removes the item at the top of the stack and returns it.

- `String peek()` – returns the top element but does not remove it.

Java implements a Stack using an ArrayList-like structure. This works just as well, and is faster in practice, but I prefer the LinkedList structure as a mathematical concept as it is more elegant and more easily customizable.

2.2.4   Queue

A queue is like a lunch line. We *add* things to the end and *poll* things from the front. Sometimes we *peek* at the front but don't actually remove anything. The first person in line gets served first. This is called *FIFO*: First In, First Out.

- `boolean add(String s)`

- `String poll()` – removes the item at the front of the queue and returns it. Same thing as `remove()`.

- `String peek()`

In Java, `Queue` is an interface. This means that we cannot instantiate a `Queue`, so the following statement is illegal.

```
Queue<String> q = new Queue<String>();
```

Instead, we must do something like this:

```
Queue<String> q = new Queue<LinkedList>();
```

This is legal because LinkedList implements Queue. Queue, however, does not extend List, so it is somewhat untruthful to place Queue under List in this book. However, I do want to stress that the LinkedList is the standard FIFO queue.

2.3   PriorityQueue

Quite often a FIFO queue is not always desirable. Maybe the String I want to remove at every given point is the one that is lexicographically least. a PriorityQueue is a Queue that allows us to do this using what is known as a *heap*.

A min heap is a tree such that every node is smaller than or equal to all of its children. Pictured is a complete binary min heap, which will be of most use to us.
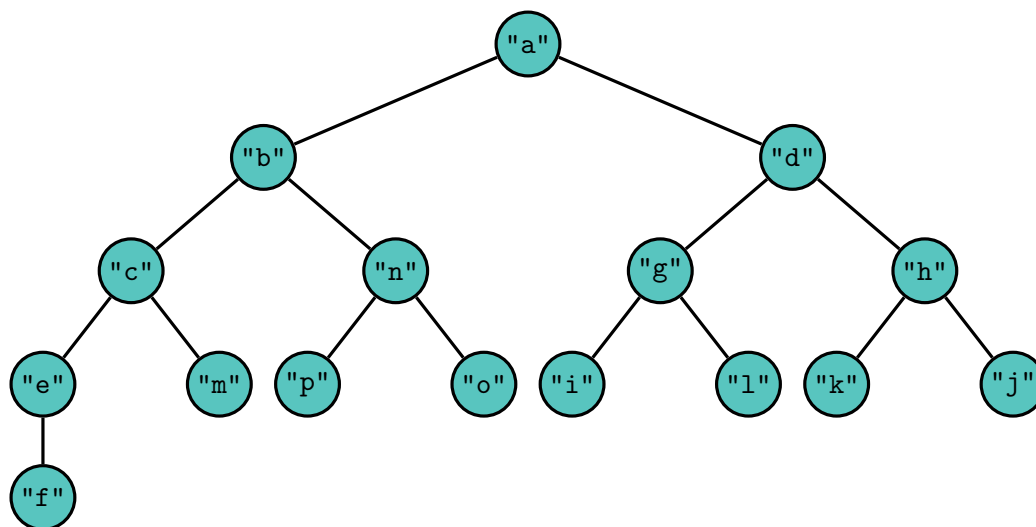
We see that the root of the tree will always be the top element. It is tempting to use a container class with a pointer to its left and its right child. However, we have a much nicer way to store *complete* binary trees with an array. Consider the following numbering of the nodes:
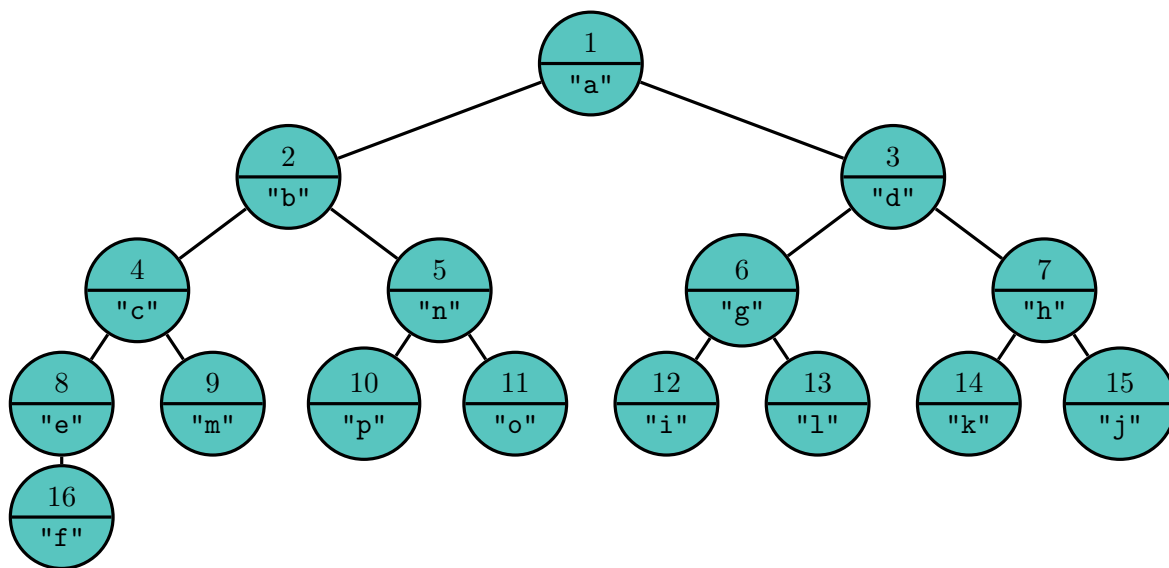
We see that every number from 1 to 16 is used, and for every node, if the index associated with it is i, the left child is $2i$, and the right child is $2i + 1$. This leads to a very natural implementation of the tree in an array:

null	"a"	"b"	"d"	"c"	"n"	"g"	"h"	"e"	"m"	"p"	"o"	"i"	"l"	"k"	"j"	"f"
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

How do we add elements to our heap, while maintaining the heap qualities? Well, let's just add it to the very end and see what we get. Suppose we are to add "b" to the tree.
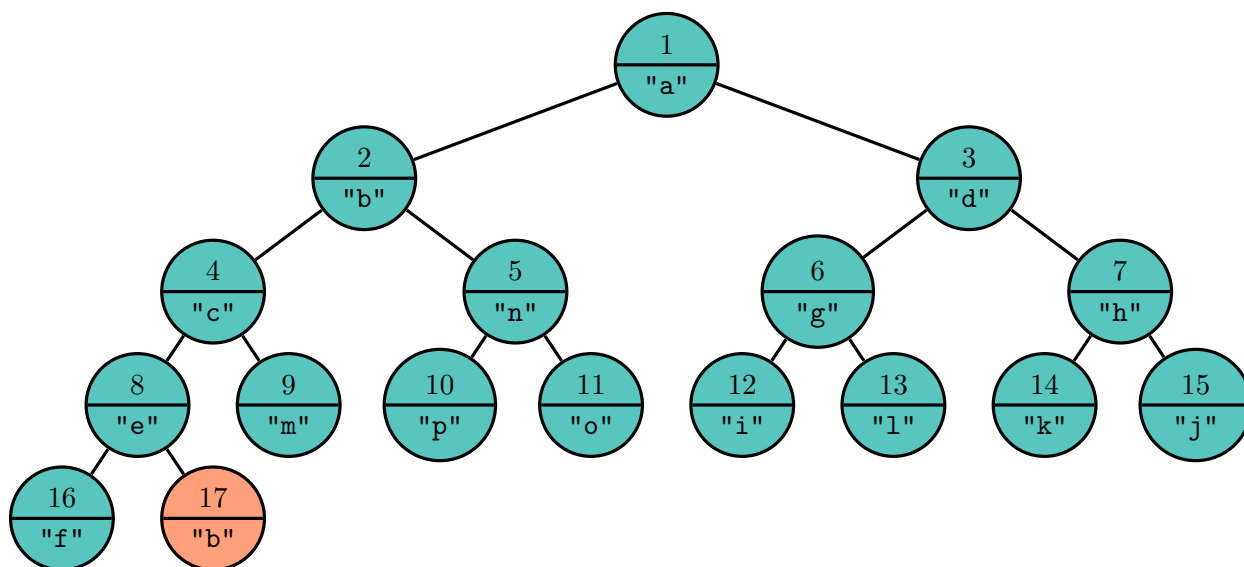
Well, "b" comes before "e" in the alphabet, so let's swap the nodes. We are guaranteed that "b" should come before the other child (in this case, "f") by the transitive property.
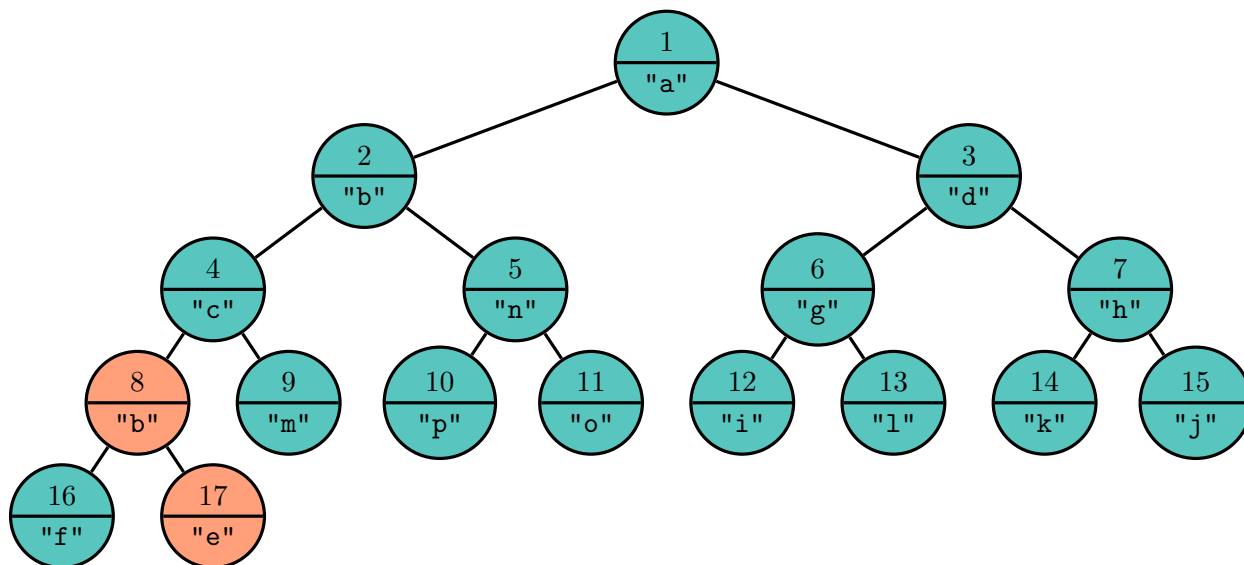
One more swap...
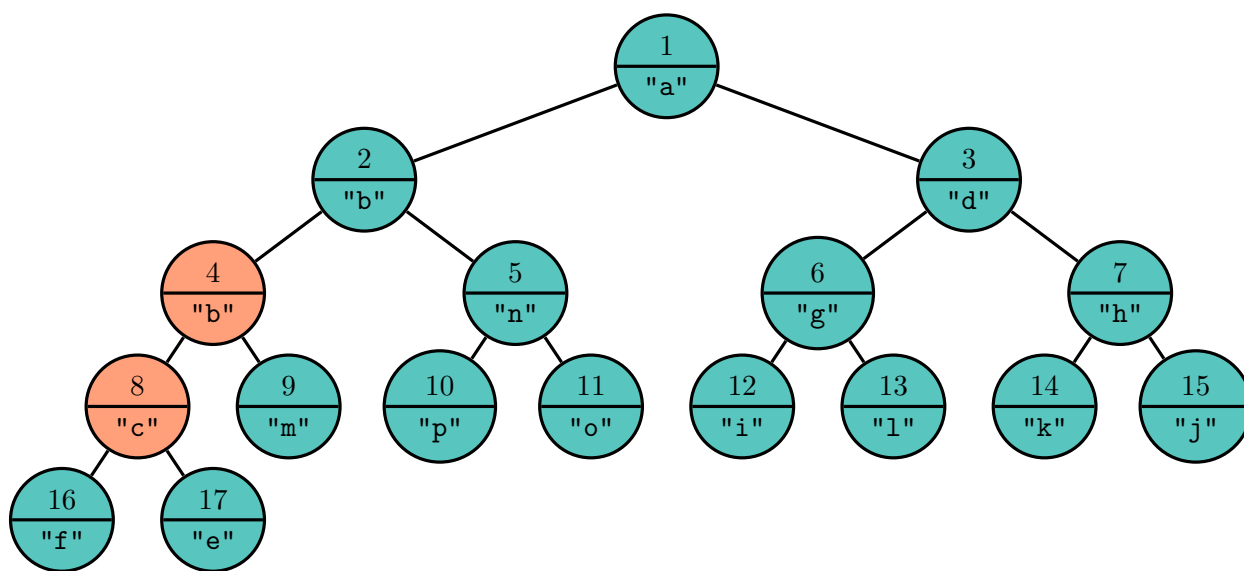
And now we have the heap property restored. As the tree has depth at most $\log n$, this process is $O(\log n)$.

To remove the root from the heap, we replace the root with the last leaf:
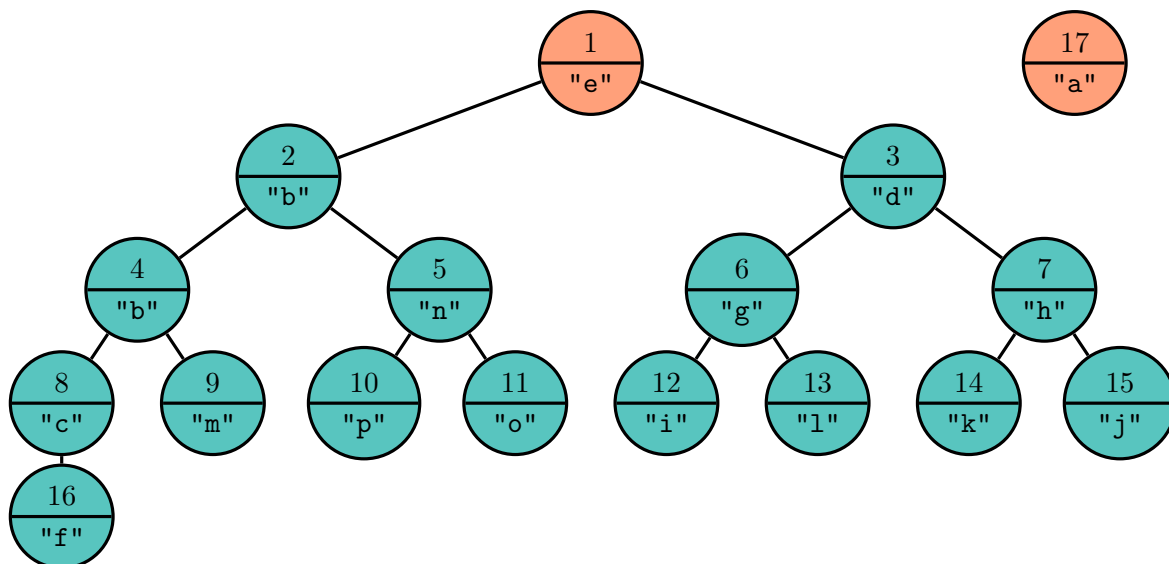
We perform a series of swaps to restore the heap property. We always want to choose the smaller child to swap until the heap property is satisfied.
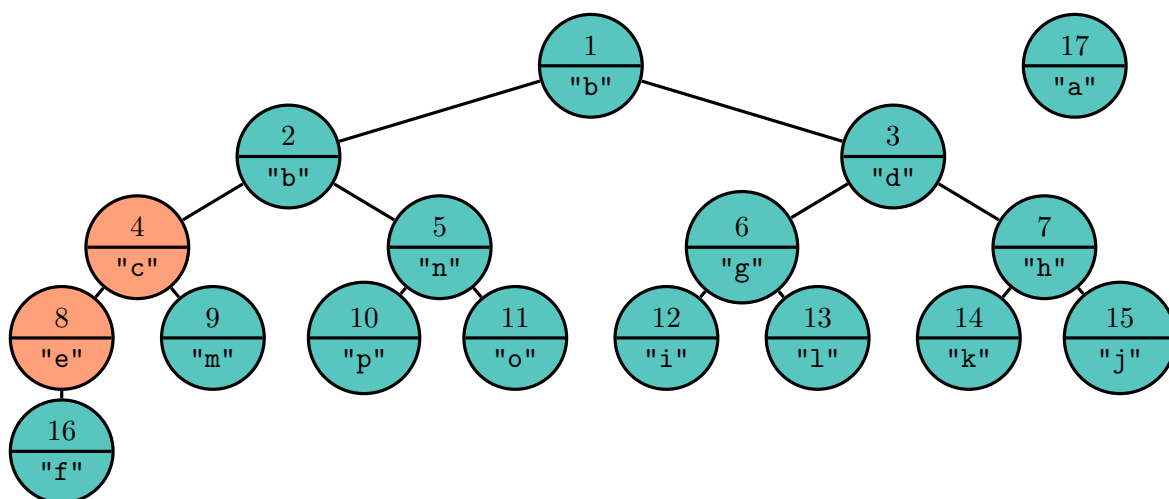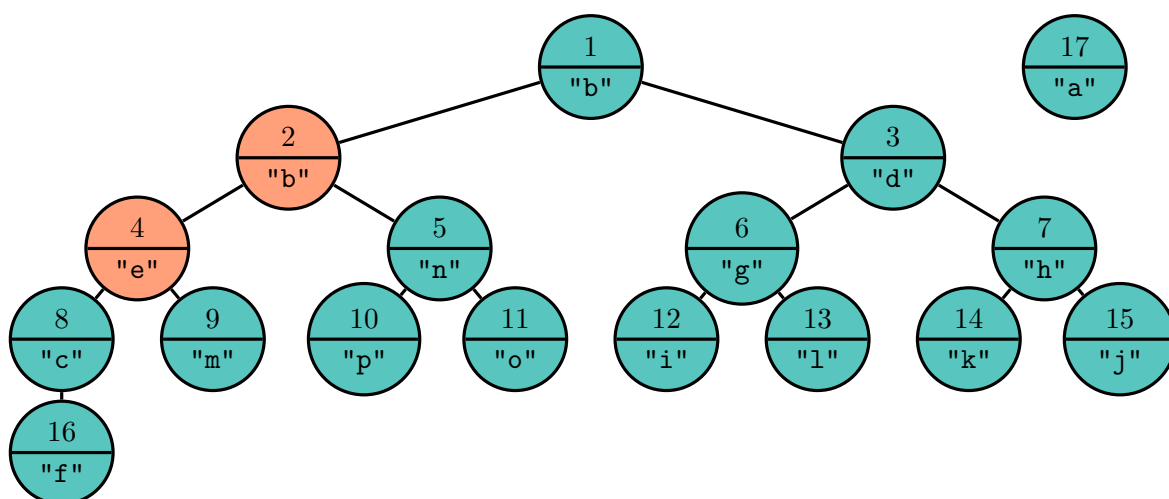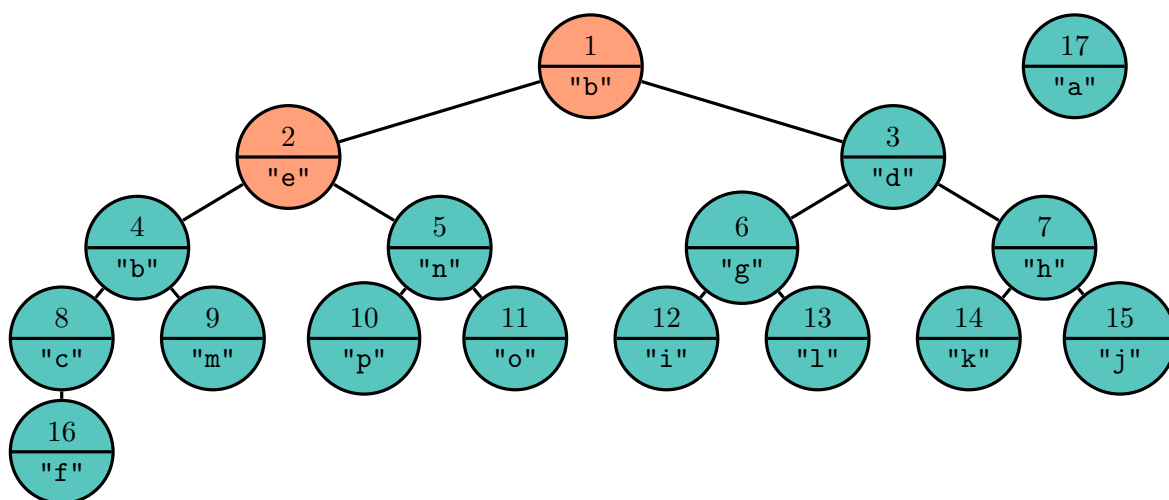
And we are done. Once again, this takes at most $\log(N)$ swaps. This idea can be extended to removing or changing the value of any node we'd like from a tree – this is particularly useful for Dijkstra later.

Remember to implement your heap in an array or ArrayList!

- `boolean add(String s)`

- `boolean isEmpty()`

- `String poll()`

- `String peek()`

- `int size()`

2.4 Set

A Set is a collection of objects with no duplicate elements. Set is a Java interface. Note that the data structures discussed in this section can be extended to become multisets, but Java implementations of these explicitly disallow multiplicity.

2.4.1 TreeSet

Binary Search Tree (BST)

A TreeSet is Java's implementation of a *binary search tree* (BST). A binary search tree is a tree where every node is greater than every node in its left subtree and less than every node in its right subtree. As with a heap, to use a BST, we need to impose some kind of ordering on the elements stored.
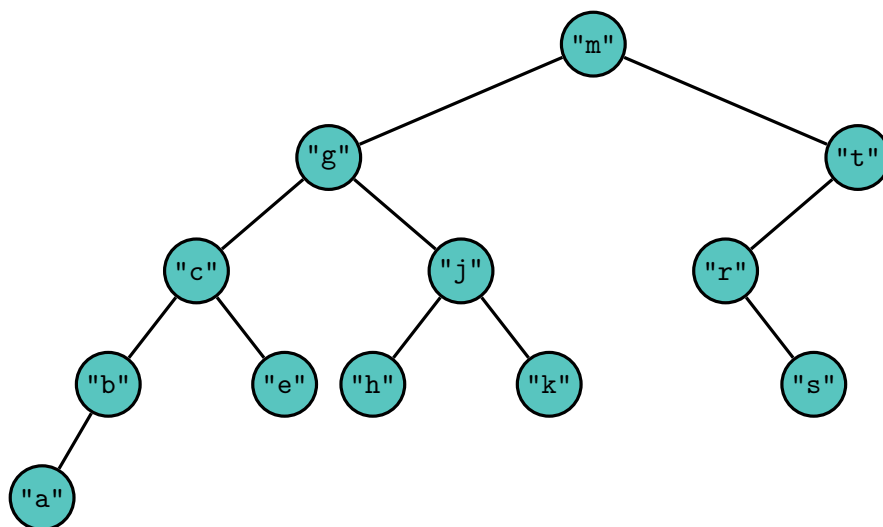
The tree need not be complete. Because it is not complete, there is no way to nicely bound the size of the array we would need if we were to use the same storage method as with the heap. Thus, we are forced to use a TreeNode, with left and right pointers. This is also problematic when

determining guarantees on time complexities later, but the ways to solve this problem are pretty complicated so we'll ignore them for now.

Given the name of the tree, searching for an element within the tree is quite natural, and similar to a binary search. Compare the element to be searched for with the current node. If they are equal, we are done; otherwise, search the appropriate left or right subtree. As with most structures and algorithms with a binary search structure, this operation lends itself nicely to recursion. If the tree is reasonably nice, we expect to complete this in $O(\log n)$ time, but searching can be as bad as linear if the tree looks like a LinkedList.

Adding an element is also natural. As our tree represents a set, it will not contain the same element twice. We trace down until we hit a null pointer, and add the element in the appropriate spot. Let's add a `"p"` to the BST:
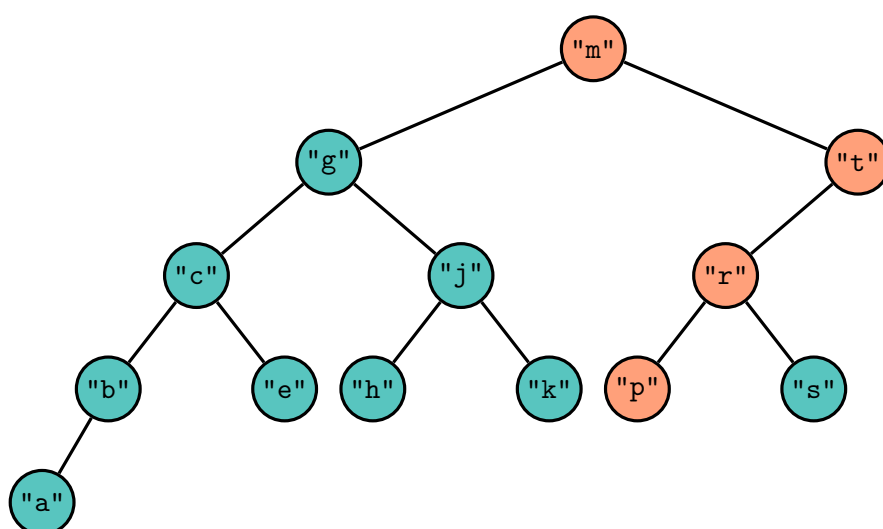
Deleting an element is the annoying part. Unfortunately, there's not much we can do besides casework.
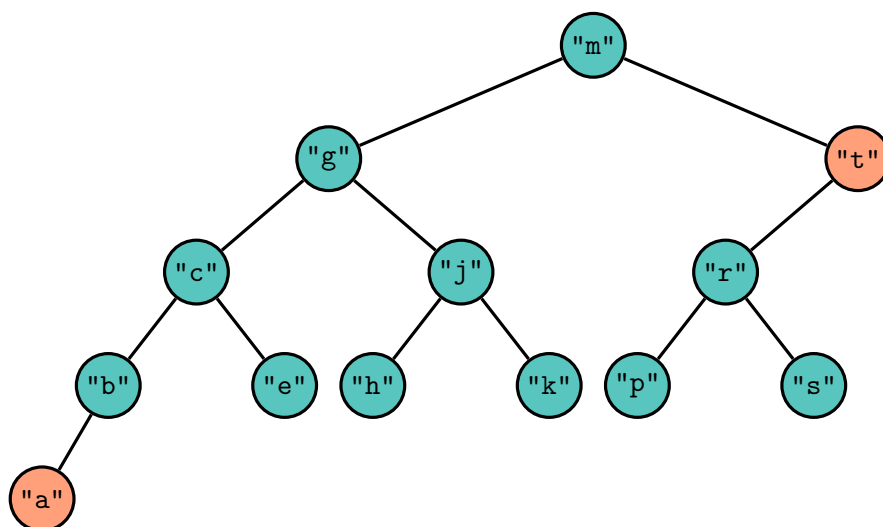
Removing a leaf, like `"a"`, from the tree is very easy. Removing a node with only once child, like `"t"`, is also relatively straightforward.
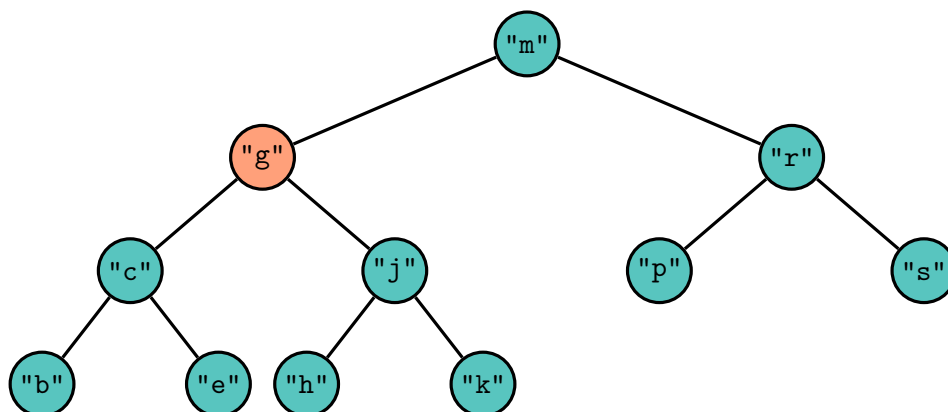
Now, removing an element with two children is tricky. We'll try to remove `"g"`. Consider the least element in the right subtree of `"g"`, which in this case is `"h"`. We find `"h"` by always choosing the left child on the right subtree until we cannot go any further. This must be the least element.

Note that `"h"` has either no children or only one child, and that nodes like these are easy to remove. We then change the value of the node containing `"g"` to `"h"`, which is legal since `"h"` is the least element, and remove `"h"` from the right subtree, and we are done.
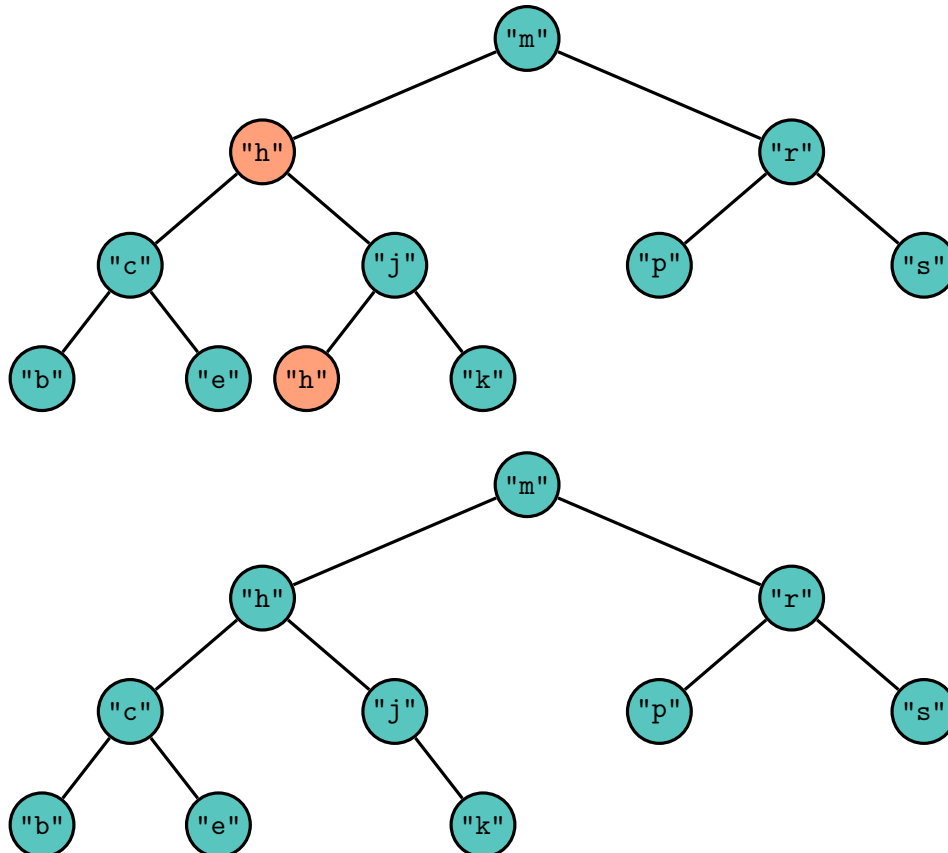
Here are some notable functions TreeSet implements. You don't need to implement everything – `add()`, `remove()`, and `contains()` are the most important.

- `boolean add(String s)`

- `String ceiling(String s)` – the least element in the set greater than or equal to `s`, or `null` if there is no such element.

- `boolean contains(Object o)`

- `String first()` – the least element in the set.

- `String floor(String s)` – the greatest element in the set less than or equal to `s`, or `null` if there is no such element.

- `String higher(String s)` – the least element in the set strictly greater than `s`, or `null` if there is no such element.

- `boolean isEmpty()`

- `String last()` – the greatest element in the set.

- `String lower(String s)` – the greatest element in the set strictly less than `s`, or `null` if there is no such element.

- `boolean remove(Object o)`

- `int size()`

Since the TreeSet is ordered, iterating over a TreeSet will also be in order.

Red-Black Tree

A standard BST has $O(\log n)$ operations if the tree is "nice", but each operation can be $O(n)$ in the worst case. We need to find a way to automatically balance the BST such that we avoid linear time complexities.

The red-black tree is a self-balancing BST that guarantees $O(\log n)$ operations. It is implemented in Java's TreeSet, so while the BST I described above does not guarantee nice time bounds, Java's implementation does.

I don't think learning exactly how a red-black tree works is particularly useful for the beginning programmer. Feel free to skip this section and come back to it later.

2.4.2   HashSet

A HashSet is a way for us to store objects when we do not require a natural ordering on the set. As with the TreeSet, we want to be able to check whether an element is in our set or not quickly. We do this with the help of a *hash function*. Every Java Object supports the `hashCode()` function. We

usually want to map an object with an integer hash, so that we can store the values in an array. For example, let us define the following hash for Strings:

```java
public int hashCode () {
    int hash = 0;
    for(int k = 0; k < length (); k++) {
        hash *= 31;
        hash += (int) (charAt(k));
    }
    return hash;
}
```

`a.equals(b)` should imply `a.hashCode() == b.hashCode()`. This function produces the same result as the actual `hashCode()` function in the String class. However, this is not quite what we want for our HashSet, because in the end we wish to be able to store the objects in some kind of array. This hash not only returns integers that can be very large, they can also be negative, and thus are not suitable as array indices. The natural way to fix this is to take the hash modulo the size of the array we want.

```java
String[] table = new String[10007];
int index(String s) {
    int i = s.hashCode() % table.length;
    if(i >= 0)
        return i;
    return i + table.length;
}
```

We chose the number 10007 because it is a prime number, and primes are generally nice when taking a number modulo something else, as integers modulo a prime form a field. Remember that a negative number `%` another number is not necessarily positive, so we need to be a little careful.

From here, adding an element to the HashSet and checking if an element is contained both seem straightforward:

```java
boolean add (String s) {
    table[index(s)] = s;
    return true;
}
boolean contains(Object o) {
    int i = index((String) o);
    return table[i] != null && table[i].equals(o);
}
```

`null` is always annoying to deal with, and will have to be handled separately.

However, one problem quickly arises. Two Strings may map to the same index in the array. We call this a *collision*. The easiest way to handle a collision is by *chaining*. We change the hash table to store a LinkedList instead of a single element in the event of a collision. Java once implemented this method of resolving collisions, but recently changed the LinkedList to a BST in Java 8.

- `boolean add(String s)`

- `boolean contains(Object o)`

- `boolean isEmpty()`

- `boolean remove(Object o)`

- `int size()`

2.5   Map

A Map is simply an extension of a Set. It stores a mapping that takes a key to a value. Map is a
Java Interface. Generics for Maps therefore have two arguments. Consider the following Map from
Strings to Integers.

```
Map<String, String> email = new TreeMap<String, String>();
m.put("Samuel Hsiang", "samuel.c.hsiang@gmail.com");
```

The keys of a Map form a Set, though the Values need not be unique.

The TreeMap is the Map variant of the TreeSet; similarly, the HashMap is the Map variant of
the HashSet.

All useful Set functions have a Map counterpart. The following additional functions are of use.

- `Set<String> keySet()` – returns the set of keys.

- `Integer put(String k, String v)` – assigns the value `k` to the key `v`, and returns the old
  value assigned to `k`.

2.6   BigInteger

BigInteger is in `java.math` for times when int and long just aren't large enough.

2.7   C++ Analogs

For transferring to STL.

- `ArrayList` – `vector`

- `LinkedList` – no explicit equivalent, but see `queue`

- `Stack` – `stack`

- `Queue` – `queue`

- `PriorityQueue` – `priority_queue`, but note that `priority_queue` pops *max* element first

- `TreeSet` – `set`

- `HashSet` – `unordered_set`, C++11

23

- `TreeMap` – `map`

- `HashMap` – `unordered_map`, C++11

- `BigInteger` – no equivalent

Chapter 3

Big Ideas

In this chapter we'll discuss some general ideas for solving problems. Starting in this chapter I'm going to shift from language-specific terms, like PriorityQueue and TreeMap, to more general terms, like binary heap and binary search tree. Algorithms I present will no longer be in the form of concrete Java code but rather in a more abstract pseudocode.

3.1 Search Techniques

There are many different general strategies for finding the answer to a problem.

3.1.1 Complete Search

Sometimes the best way is simply to try everything. This could be the intended solution (check the complexity of a complete search and compare it against the time limits), or we may be just coding a brute force to squeeze out a few extra points from an intractable problem at the end of a contest. Either way, the order in which we search can make or break the code.

Suppose we wanted to solve the game of chess. The placement of the pieces on the board and a toggle for whether black or white is to move represents a game state. The set of legal moves maps this game state to one set of states and another set of states to this game state.

We can therefore think of the different states as vertices in a very large directed graph. Methods we use for solving chess, or any other problem, are identical to ways of traversing graphs, especially trees.

Depth-First Search

Depth-first search is the most simple of the searches. A depth-first search called on a vertex v in the search tree recursively calls itself on each of the children of v. To go back to the chess example, a DFS would begin with one possible first move, like a3, and test *every possible move sequence* beginning with a3 before moving on to a second possible first move, like a4.

A quick note on recursive processes: The *run-time stack* keeps track of our position and data that go out of scope when we jump into a new function. This stack allows us to call functions within

functions safely. Recursion is dangerous in contest programming because the run-time stack is slow and can easily balloon in size, crashing the program. For this reason do not hesitate to use your own stack in an iterative process in practice, though I often describe processes as recursive as they are easier to understand, code, and debug in this form.

Now it is painfully obvious that strictly using a DFS is not preferable in computer chess. The computer would certainly never finish testing every move sequence beginning with a3 and therefore would not consider more standard, and likely better, moves like e4.

Breadth-First Search

Breadth-first search traverses the search tree level by level. We keep track of a queue that stores each level's state. At each step we pop off the first element in the queue and add the states that it can reach to the end of the queue. In this way, we explore those states only after we explore all the states associated with a lower level. In the chess example, we would store all 20 possible first moves in the queue, and for each first move, we add all possible second moves to the end of the queue.

This means that we search all possible states of a certain depth at roughly the same time, but we need to store the data associated with each of these states, and this can be quite costly in terms of memory.

BFS is the better choice over DFS when we are asked for the "smallest" answer, which is often associated with the lowest level.

Depth-First Search with Iterative Deepening

Depth-first search with iterative deepening is a DFS that only searches up to a certain level in the search tree before stopping and heading back to lower levels. If it turns out the search did not find anything in the first N levels, we broaden the search to the first $N + 1$ levels with another DFS, hence iterative deepening. This method of searching is slower than a BFS, but maintains the nice BFS property of finding the "smallest" answer first while shedding the memory harness that holds the BFS back.

DFSID is the closest to what we do when analyzing an actual chess game. We make a move in our minds and test its performance against whatever the opponent might make as his move, tracing the game around 3-4 moves deep before testing another possible move.

3.1.2   Greedy Algorithm

The main problem with any of the searches described above is they are exponential in nature. A *greedy algorithm* is one that takes the "best" possible option at each step, essentially disregarding any other possible option. To find the best solution for $n = 4$, we consider only the best solution for $n = 3$ and no other possible solution. The approach of maximizing at each step clearly does not always work, as the locally optimal choice is not necessarily globally optimal. Always moving in the direction of a target, for example, fails if there is a wall in the way. However, if the greedy algorithm can solve a problem, the code runs very quickly.

One problem that cannot be solve using the greedy approach is the integer knapsack problem. It's important to be able to catch an incorrect greedy algorithm, and one classic example involves

finding the most efficient way to make change. Given a sequence of coin denominations (d_i, where $1 \leq i \leq n$), with $d_1 = 1$, find the smallest number of coins necessary that sum to some value V. The greedy approach takes the most valuable coin that doesn't overshoot V and adds it to our set until we achieve V. One counterexample is $v_1 = 1$, $v_2 = 3$, $v_3 = 4$ and $V = 6$. The greedy algorithm chooses $4 + 1 + 1$, which is worse than $3 + 3$. Note however that greedy algorithm still always works for some sets of coin values, like United States currency. Don't let the fact that the greedy algorithm works for the most obvious example fool you.

The greedy algorithm combined with ideas from dynamic programming constitute *best-first search*, a category of searches that includes the Dijkstra algorithm for shortest paths.

1. (USACO Training) There is a long list of stalls, some of which need to be covered with boards. You can use up to N ($1 \leq N \leq 50$) boards, each of which may cover any number of consecutive stalls. Cover all the necessary stalls, while covering as few total stalls as possible.

2. (IOI 1996) You are given a three-valued (1, 2, or 3) sequence of length up to 1000. Find a minimum set of exchanges to put the sequence in sorted order.

3. (Samir Khuller) We are given N jobs, each of which requires one unit of time to complete. The ith job opens at some time t_i and must be completed by some deadline d_i, where $t_i, d_i \in \mathbb{Z}$. Given that only one job can be completed at a time, determine if all N can be completed by their deadlines.

3.1.3   Binary Searching on the Answer

Suppose that the problem we need to solve is finding the minimum number M such that some property holds. That is, the property holds for any $x \geq M$ but does not hold for $x < M$. Perhaps the best approach we have so far for finding this M is simply trying all the numbers from 1 to M. However, this linear search is clearly inefficient.

The nature of this problem should remind you of some other search technique. Oftentimes, with problems of this property, it is easy to check whether some condition holds for some given x. In this case, it is much easier to binary search on M rather than find it some other way.

1. (Matthew Savage) Ashley's journey through Unova is just beginning, and she has just picked her first Pokémon! Unfortunately, not knowing much about them, she picked a Snivy, and a particularly stubborn and unfriendly one at that.

    Being Ashley, she decides to try to win over the Snivy in the only way she knows how – baked goods.

    Ashley knows r ($0 \leq r \leq 1000$) recipes for Poképuffs. Each recipe r_i has a deliciousness rating d_i ($0 \leq d_i \leq 1000$) and requires some combination of the I ($0 \leq I \leq 1000$) available ingredients. (More specifically, the recipe r_i uses the quantity I_{ij} ($0 \leq I_{ij} \leq 1000$) of ingredient I_j.)

    Ashley has some amount of each ingredient I_j on hand A_j ($0 \leq A_j \leq 10^9$) and can buy more from the nearby store for a price of c_j ($1 \leq c_j \leq 10^9$) dollars per unit using the M ($0 \leq M \leq 10^{12}$) dollars she currently has.

Of course, Ashley also has limited supplies and therefore can only produce Poképuffs from a single recipe. However, she can make as many as she wants, in integer increments.

We define "total deliciousness" (D) to be the sum of the deliciousnesses of the individual Poképuffs that Ashley has baked.

Ashley wants to have the best chance possible with Snivy, and therefore would like to know - what is the maximum possible deliciousness ($\max(D)$) that she can produce?

Note: there is a "just do it" solution that is faster than the binary search by a log factor. It is also *much* more annoying to code; so annoying that I was unable to debug my "just do it" solution in the actual contest environment. I included this problem as an exercise to demonstrate how easy binary searching on the answer can be.

2. (CF 287B) Vova, the Ultimate Thule new shaman, wants to build a pipeline. As there are exactly n ($1 \leq n \leq 10^{18}$) houses in Ultimate Thule, Vova wants the city to have exactly n pipes, each such pipe should be connected to the water supply. A pipe can be connected to the water supply if there's water flowing out of it. Initially Vova has only one pipe with flowing water. Besides, Vova has several splitters.

A splitter is a construction that consists of one input (it can be connected to a water pipe) and x output pipes. When a splitter is connected to a water pipe, water flows from each output pipe. You can assume that the output pipes are ordinary pipes. For example, you can connect water supply to such pipe if there's water flowing out from it. At most one splitter can be connected to any water pipe.

Vova has one splitter of each kind: with $2, 3, 4, \ldots, k$ ($2 \leq k \leq 10^9$) outputs. Help Vova use the minimum number of splitters to build the required pipeline or otherwise state that it's impossible.

Vova needs the pipeline to have exactly n pipes with flowing out water. Note that some of those pipes can be the output pipes of the splitters.

3. (IOI 2009) Mecho the bear has found a little treasure - the bees' secret honeypot, which is full of honey! He was happily eating his newfound treasure until suddenly one bee saw him and sounded the bee alarm. He knows that at this very moment hordes of bees will emerge from their hives and start spreading around trying to catch him. He knows he has to leave the honeypot and go home quickly, but the honey is so sweet that Mecho doesn't want to leave too soon. Help Mecho determine the latest possible moment when he can leave.

Mecho's forest is represented by a square grid of N by N ($1 \leq N \leq 800$) unit cells, whose sides are parallel to the north-south and east-west directions. Each cell is occupied by a tree, by a patch of grass, by a hive or by Mecho's home. Two cells are considered adjacent if one of them is immediately to the north, south, east or west of the other (but not on a diagonal). Mecho is a clumsy bear, so every time he makes a step, it has to be to an adjacent cell. Mecho can only walk on grass and cannot go through trees or hives, and he can make at most S ($1 \leq S \leq 1000$) steps per minute. At the moment when the bee alarm is sounded, Mecho is in the grassy cell containing the honeypot, and the bees are in every cell containing a hive (there may be more than one hive in the forest). During each minute from this time onwards, the following events happen in the following order:

- If Mecho is still eating honey, he decides whether to keep eating or to leave. If he continues eating, he does not move for the whole minute. Otherwise, he leaves immediately and

takes up to S steps through the forest as described above. Mecho cannot take any of the honey with him, so once he has moved he cannot eat honey again.

- After Mecho is done eating or moving for the whole minute, the bees spread one unit further across the grid, moving only into the grassy cells. Specifically, the swarm of bees spreads into every grassy cell that is adjacent to any cell already containing bees. Furthermore, once a cell contains bees it will always contain bees (that is, the swarm does not move, but it grows).

In other words, the bees spread as follows: When the bee alarm is sounded, the bees only occupy the cells where the hives are located. At the end of the first minute, they occupy all grassy cells adjacent to hives (and still the hives themselves). At the end of the second minute, they additionally occupy all grassy cells adjacent to grassy cells adjacent to hives, and so on. Given enough time, the bees will end up simultaneously occupying all grassy cells in the forest that are within their reach.

Neither Mecho nor the bees can go outside the forest. Also, note that according to the rules above, Mecho will always eat honey for an integer number of minutes.

The bees catch Mecho if at any point in time Mecho finds himself in a cell occupied by bees.

Write a program that, given a map of the forest, determines the largest number of minutes that Mecho can continue eating honey at his initial location, while still being able to get to his home before any of the bees catch him.

3.2   Dynamic Programming

The idea behind dynamic programming is to avoid doing the same thing twice. Two nodes in the search tree described in the complete search techniques might very well represent the same state. For example, two different sets of initial moves could result in the same chessboard configuration, and if we already calculated who has the winning or losing position in that configuration, we ought to remember that fact somehow so we don't need to calculate it again. Of course, however, actually keeping track of every possible game state is intractable by memory constraints.

Here's a much more reasonable problem. Given a sequence of $N \leq 10,000$ integers, what is the maximum decreasing subsequence? A subsequence does not have to consist of consecutive terms in the original sequence.

6	9	8	4	7	5
1	2	3	4	5	6

The natural complete search approach would be to use recursion, or DFS. When we process an element in the list, we recursively process all elements that come after it and choose the one that gives the maximum subsequence.

function PROCESS(i)
    $max \leftarrow 0$
    for $j \equiv i + 1, N$ **do**
        if $value(i) > value(j)$ **then**

$$x \leftarrow \text{PROCESS(j)}$$
$$\textbf{if } x > max \textbf{ then}$$
$$max \leftarrow x$$
$$\textbf{return } max + 1$$

However, this algorithm is exponential. In the worst case, it is $O(2^N)$. We notice a lot of repetition: processing the 9 in the list above, for example, requires finding the longest subsequences beginning with 8, 4, 7, and 5, while processing 8 requires finding subsequences for 4, 7, and 5. It seems silly to do the same task twice, so we'll keep track of the length of the longest subsequence in a separate array.

function PROCESS(i)
    if i has already been processed **then**
        return $dp(i)$
    $max \leftarrow 0$
    for $j \equiv i + 1, N$ **do**
        if $value(i) > value(j)$ **then**
            $x \leftarrow$ PROCESS(j)
            if $x > max$ **then**
                $max \leftarrow x$
    $dp(i) \leftarrow max + 1$
    return $max + 1$

This reduces the complexity of the algorithm to $O(n^2)$. Note that to process an index, we must process first all later indices. This imposes a natural ordering in which to process the indices: in reverse. This idea lends itself to a nice iterative solution.

for $i \equiv N, 1$ **do** $\triangleright$ i goes in reverse
    $max \leftarrow 0$
    for $j \equiv i + 1, N$ **do**
        if $value(i) > value(j)$ **then**
            if $dp(j) > max$ **then**
                $max \leftarrow dp(j)$
    $dp(i) \leftarrow max + 1$

The answer to the original problem is then the maximum value of $dp(i)$ for all i. For this specific problem, it's relatively easy to speed up the algorithm to $O(\log n)$ by replacing the linear search with something else.

The integer knapsack problem is another example where dynamic programming may be useful. *Knapsack problems* are a family of problems with the following form:

We are given a list of K objects each assigned an availability, a size, and a value. We have a total amount of "space" available in our knapsack and need to find the set of objects from our list that maximizes the total the value of objects in the set such that the total size does not exceed the space and the number of times we take one particular object does not exceed its availability.

Dynamic programming yields a straightforward $O(NK)$ solution. See if you can find it. Note, however, if N is very large, this solution is no longer practical. In general, the knapsack problem is NP-complete, so don't think dynamic programming works on everything!

3.2.1 Dynamic Programming over Subsets

Consider the following problem:

(USACO December 2014, guard) Farmer John and his herd are playing frisbee. Bessie throws the frisbee down the field, but it's going straight to Mark the field hand on the other team! Mark has height H ($1 \leq H \leq 1,000,000,000$), but there are N cows on Bessie's team gathered around Mark ($2 \leq N \leq 20$). They can only catch the frisbee if they can stack up to be at least as high as Mark. Each of the N cows has a height, weight, and strength. A cow's strength indicates the maximum amount of total weight of the cows that can be stacked above her.

Given these constraints, Bessie wants to know if it is possible for her team to build a tall enough stack to catch the frisbee, and if so, what is the maximum safety factor of such a stack. The safety factor of a stack is the amount of weight that can be added to the top of the stack without exceeding any cow's strength.

We can try the $O(N!)$ brute force, trying every permutation of cows possible. However, this is far too slow. $N \leq 20$ hints at an exponential solution, so we think of trying every possible subset of the cows. Given a subset S of cows, the height reached is the same, so perhaps we sort the subset by strength, and put the strongest cow on the bottom. We see that this greedy approach fails: suppose that the first cow has weight 1 and strength 3 and the second cow has weight 4 and strength 2. Greedy would tell us to put the first cow on the bottom, but this fails, while putting the second cow on the bottom succeeds.

When greedy fails, the next strategy we look at is dynamic programming. To decide whether S is stable, we have to find whether there exists a cow j in S that can support the weight of all the other cows in S. But how do we know whether the set $S \setminus \{j\}$ is stable? This is where dynamic programming comes in.

This leads to a $O(N2^N)$ solution. This seems like a pain to code iteratively, but there is a nice fact about subsets: there is a bijection from the subsets of $\{0, 1, 2, \ldots, N-1\}$ to the integers from 0 to $2^N - 1$. We call this technique *masking*. We require all the subsets of S to be processed before S is processed, but that property is also handled by our bijection, since subtracting a power of 2 from a number decreases it. With a little knowledge of bit operators, this can be handled easily.

> **for** $i \leftarrow 0, 2^N - 1$ **do** $\triangleright$ i represents the subset S
> $\quad dp(i) \leftarrow -1$
> $\quad$ **for all** $j \in S$ **do** $\triangleright$ `i & (1 << j) != 0`
> $\quad\quad alt \leftarrow \min(dp(i - 2^j), strength(j) - \sum_{k \in S \setminus \{j\}} weight(k))$
> $\quad\quad$ **if** $dp(i) < alt$ **then**
> $\quad\quad\quad dp(i) \leftarrow alt$

`&` is the bitwise and function, while `<<` is the left shift operator.

Brian Dean compiled some standard dynamic programming problems with animations and analyses. Practice dynamic programming here: `http://people.cs.clemson.edu/~bcdean/dp_practice/`

Chapter 4

Graph Algorithms

In this chapter we explore some famous graph theory results.

4.1 Connected Components

A *connected component* of an undirected graph is a subgraph such that, for any two vertices in the component, there exists a path from one to the other. The diagram illustrates three connected components of a graph, where each vertex is colored togethwer with its associated component.
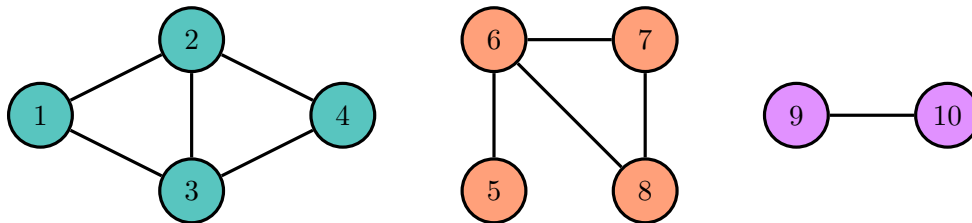
A *strongly connected component* of a directed graph is a subgraph such that every vertex in the component can be reached from any other vertex in the component.
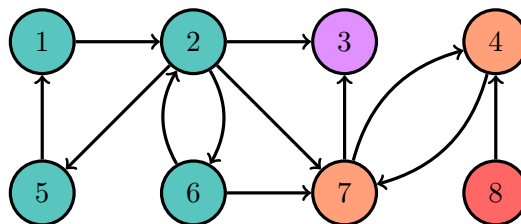
Finding the connected components of an undirected graph is a straightforward problem, while finding the strongly connected components of a directed graph is more complicated.

4.1.1   Flood Fill

Really any kind of search method solves the undirected graph connected components problem. We could use recursion with a depth-first search. To avoid using the run-time stack, we could use a queue to perform a breadth-first search. Both of these run in $O(E + V)$ time. I would recommend in general to use the BFS.

4.1.2   Union-Find

The union-find data structure is another way for us to solve the connected components problem. Union-find is unique from the other search techniques in that it can process input as it is presented, edge by edge. This also means it is possible to add more edges at the end, therefore changing the graph, while still running quickly. An algorithm that works like this is an *online algorithm*, while an algorithm that requires all the input data presented at the beginning is an *offline algorithm*.

A natural idea for solving the connected components problem is for each vertex to maintain a pointer to another vertex it's connected to, forming a *forest*, or collection of trees. To check whether two elements are in the same component, simply trace the tree up to the root by jumping up each pointer.
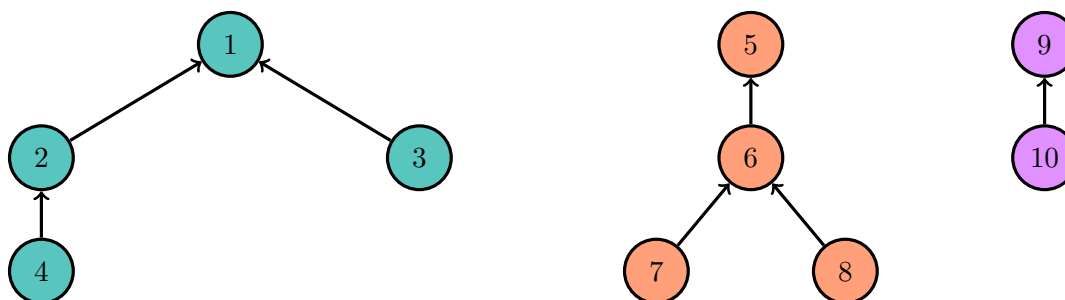
The idea of a pointer can easily be stored within an array.

-1	1	1	2	-1	5	6	6	-1	9
1	2	3	4	5	6	7	8	9	10

We want to support two operations: $find(v)$, which returns the root of the tree containing v, and $union(u, v)$, which merges the components containing u and v. This second operation is easy given the first; simply set the pointer of $find(u)$ to be $find(v)$.

$union(4, 6)$, unoptimized:
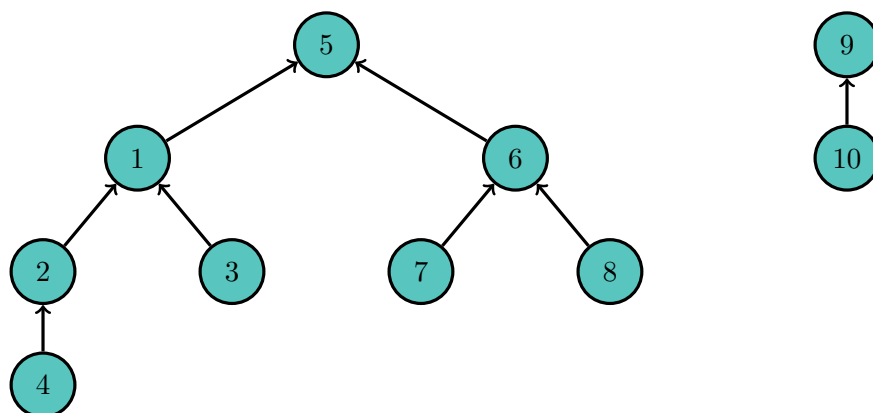
A problem quickly arises – the $find$ operation threatens to become linear. There are two simple things we can do to optimize this.

The first is to always add the shorter tree to the taller tree, as we want to minimize the maximum height. An easy heuristic for the height of the tree is simply the number of elements in that tree. We can keep track of the size of the tree with a second array. This heuristic is obviously not perfect, as a larger tree can be shorter than a smaller tree, but it turns out with our second optimization that this problem doesn't matter.

The second fix is to simply assign the pointer associated with v to be $find(v)$ at the end of the $find$ operation. We can design $find(v)$ to recursively call $find$ on the pointer associated with v, so this fix sets pointers associated with nodes along the entire chain from v to $find(v)$ to be $find(v)$. These two optimizations combined make the $union$ and $find$ operations $O(\alpha(V))$, where $\alpha(n)$ is the inverse Ackermann function, and for all practical values of n, $\alpha(n) < 5$.

$find(4)$, optimized:
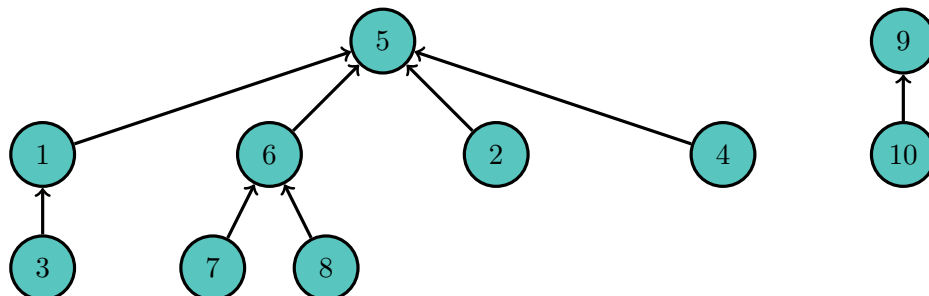
Algorithm 1 Union-Find

    function FIND(v)
        if v is the root **then**
            return v
        $parent(v) \leftarrow$ FIND($parent(v)$)
        return $parent(v)$
    function UNION(u, v)
        $uRoot \leftarrow$ FIND(u)
        $vRoot \leftarrow$ FIND(v)
        if $uRoot = vRoot$ **then**
            return
        if $size(uRoot) < size(vRoot)$ **then**
            $parent(uRoot) \leftarrow vRoot$
            $size(vRoot) \leftarrow size(uRoot) + size(vRoot)$
        else
            $parent(vRoot) \leftarrow uRoot$
            $size(uRoot) \leftarrow size(uRoot) + size(vRoot)$

---

1. (USACO Open 2008, nabor) Farmer John has N ($1 \leq N \leq 100,000$) cows who group themselves into "Cow Neighborhoods". Each cow is at a unique rectilinear coordinate, on a pasture whose x and y coordinates are in the range $1 \ldots 1,000,000,000$. Two cows are neighbors if at least one of two criteria is met: (1) If the cows are no further than some integer Manhattan distance C ($1 \leq C \leq 1,000,000,000$) apart. (2) If cow A and B are both neighbors of cow Z, then cow A is a neighbor of cow B. Given the locations of the cows and the distance C, determine the number of neighborhoods and the number of cows in the largest neighborhood.

4.1.3 Tarjan

Tarjan's algorithm for strongly connected components

Strongly connected components are a reasonably advanced topic, and they are explicitly banned on IOI. Feel free to skip this section for now.

4.2 Shortest Path

A classic. Assign nonnegative weights to each of the edges, where the weight of the edge (u, v) represents the distance from u to v. This graph can be either directed or undirected.

4.2.1 Dijkstra

Dijkstra's algorithm solves the single-source shortest path problem. From any vertex, we can compute the shortest path to each of the remaining vertices in the graph. The two formulations of Dijkstra's algorithm run in $O(V^2)$ or $O(E \log V)$ time, whichever one suits us better. Note that it

is possible to do better than $O(E \log V)$ using a Fibonacci heap. The former works nicely on dense graphs, as $E \approx V^2$, while the latter works better on sparse graphs, as $E \approx V$.

For every vertex v in the graph, we keep track of the shortest known distance $dist(v)$ from the source to v, a boolean $visited(v)$ to keep track of which nodes we "visited," and a pointer to the previous node in the shortest known path $prev(v)$ so that we can trace the shortest path once the algorithm finishes.

Dijkstra iteratively "visits" the next nearest vertex, updating the distances to that vertex's neighbors if necessary. Therefore, at any step, we have the first however-many nearest vertices to the source, which we call "visited" and for which the shortest path is known. We also have the shortest path to all the remaining vertices that stays within the "visited" vertices besides for the very last edge, if such a path exists. We claim that the known distance to the closest vertex that has not yet been visited is the shortest distance. We can then "visit" that vertex. It shouldn't be hard to prove that this algorithm indeed calculates the shortest path.

The $O(V^2)$ implementation immediately follows.

Algorithm 2 Dijkstra

> **for all** vertices v **do**
>> $dist(v) \leftarrow \infty$
>> $visited(v) \leftarrow 0$
>> $prev(v) \leftarrow -1$
>
> $dist(src) \leftarrow 0$
> **while** $\exists v$ s.t. $visited(v) = 0$ **do**
>> $v \equiv v$ s.t. $visited(v) = 0$ with min $dist(v)$
>> $visited(v) \leftarrow 1$
>> **for all** neighbors u of v **do**
>>> **if** $visited(u) = 0$ **then**
>>>> $alt \leftarrow dist(v) + weight(v, u)$
>>>> **if** $alt < dist(u)$ **then**
>>>>> $dist(u) \leftarrow alt$
>>>>> $prev(u) \leftarrow v$
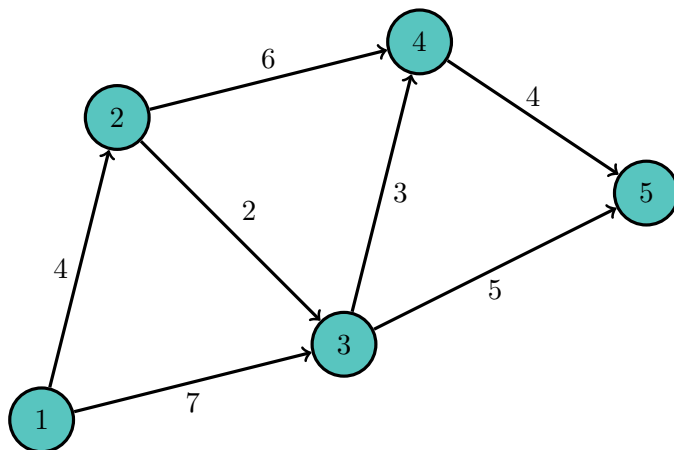
Let's run Dijkstra's algorithm on the above graph with vertex 1 as the source. We first set all the distances besides the source to be ∞.
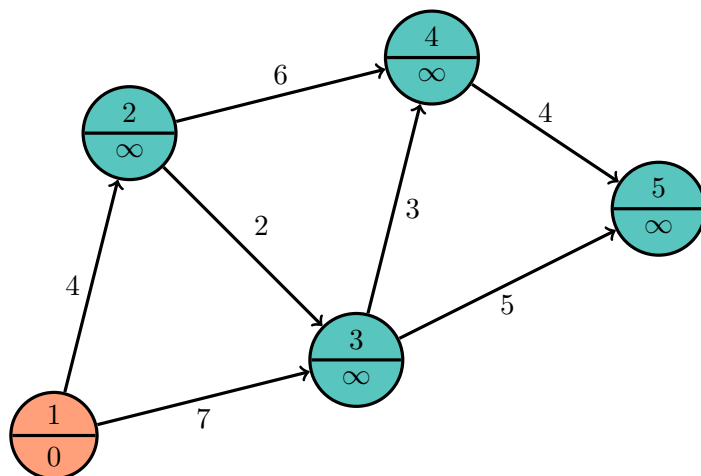
Now, we continue choosing the closest unvisited node, mark it as visited, and and update its neighbors.
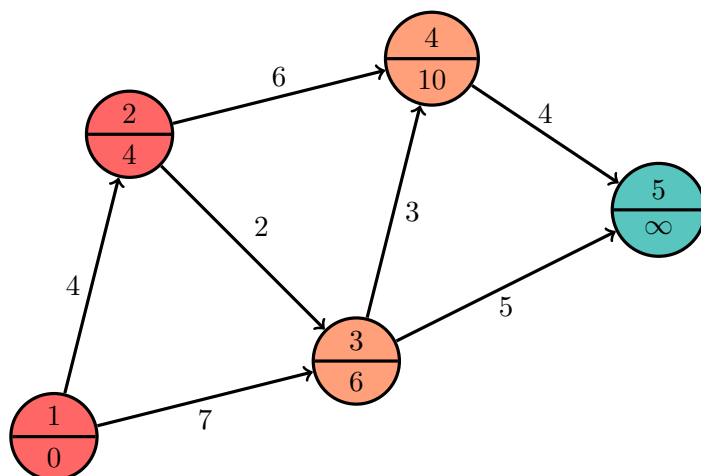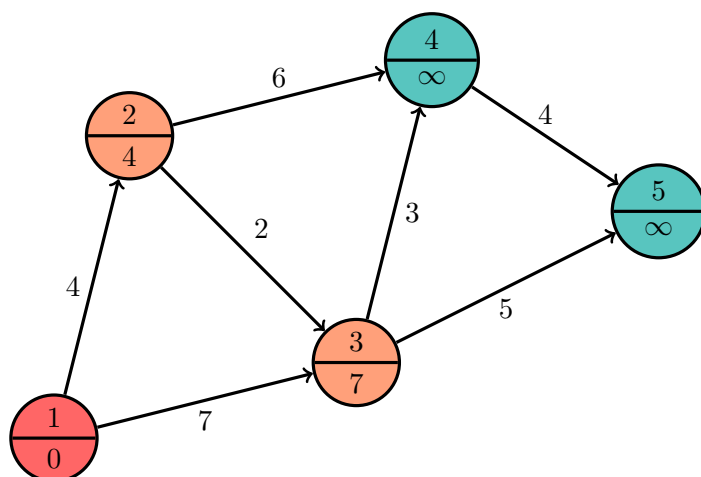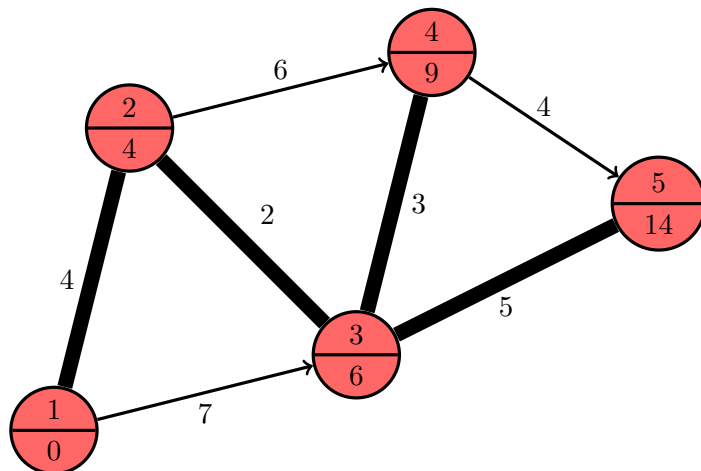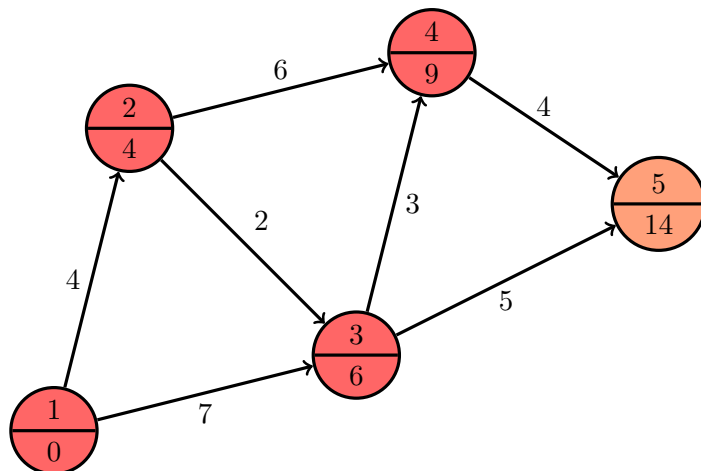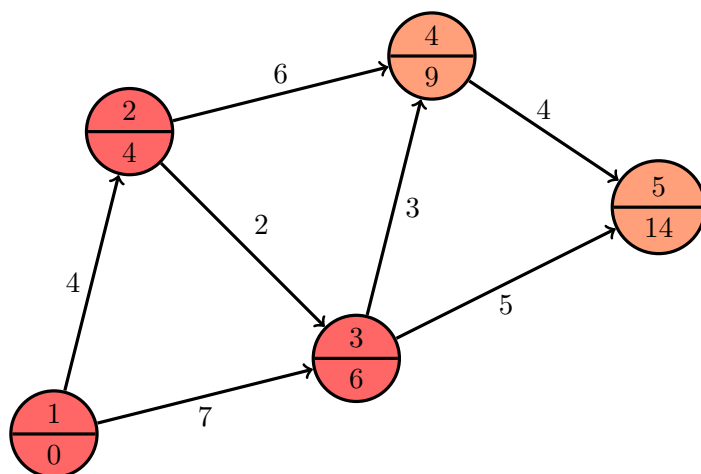
The slow part of the $O(V^2)$ formulation is the linear search for the vertex v with the minimum $dist(v)$. We happen to have a data structure that resolves this problem – a binary heap. However, we can't simply use the standard library heap, as we need to be able to change the value of the distances once they are already in the heap, or *decrease-key*. This is a pretty simple function to add,

however, if you have a heap already coded. Once we have this implemented, we achieve $O(E \log V)$, as we do $E + V$ updates to our heap, each costing $O(V)$.

1. (USACO Training Pages, butter) Farmer John owns a collection of pastures with weighted edges between some pairs of locations. Each pasture is inhabited by a cow, and the cows wish to all congregate at one of the pastures. Find the pasture at which the cows should meet in order to minimize combined travel distance.

2. (USACO February 2012, relocate) FJ is moving! He is trying to find the best place to build a new farm so as to minimize his daily travel time.

   The region to which FJ plans to move has N towns $(1 \le N \le 10,000)$. There are M bi-directional roads $(1 \le M \le 50,000)$ connecting certain pairs of towns. All towns are reachable from each-other via some combination of roads. FJ needs your help selecting the best town as the home for his new farm.

   There are markets in K of the towns $(1 \le K \le 5)$ that FJ wants to visit every day. In particular, every day he plans to leave his new farm, visit the K towns with markets, and then return to his farm. FJ can visit the markets in any order he wishes. When selecting a town in which to build his new farm, FJ wants to choose only from the $N - K$ towns that do not have markets, since housing prices are lower in those towns.

   Please help FJ compute the minimum distance he will need to travel during his daily schedule, if he builds his farm in an optimal location and chooses his travel schedule to the markets as smartly as possible.

3. (USACO December 2012, mroute) Farmer John's farm has an outdated network of M pipes $(1 \le M \le 500)$ for pumping milk from the barn to his milk storage tank. He wants to remove and update most of these over the next year, but he wants to leave exactly one path worth of pipes intact, so that he can still pump milk from the barn to the storage tank.

   The pipe network is described by N junction points $(1 \le N \le 500)$, each of which can serve as the endpoint of a set of pipes. Junction point 1 is the barn, and junction point N is the storage tank. Each of the M bi-directional pipes runs between a pair of junction points, and has an associated latency (the amount of time it takes milk to reach one end of the pipe from the other) and capacity (the amount of milk per unit time that can be pumped through the pipe in steady state). Multiple pipes can connect between the same pair of junction points.

   For a path of pipes connecting from the barn to the tank, the latency of the path is the sum of the latencies of the pipes along the path, and the capacity of the path is the minimum of the capacities of the pipes along the path (since this is the "bottleneck" constraining the overall rate at which milk can be pumped through the path). If FJ wants to send a total of X units of milk through a path of pipes with latency L and capacity C, the time this takes is therefore $L + \frac{X}{C}$.

   Given the structure of FJ's pipe network, please help him select a single path from the barn to the storage tank that will allow him to pump X units of milk in a minimum amount of total time.

4. (IOI 1999, Traffic Lights) In the city of Dingilville the traffic is arranged in an unusual way. There are junctions and roads connecting the junctions. There is at most one road between

any two different junctions. There is no road connecting a junction to itself. Travel time for a road is the same for both directions. At every junction there is a single traffic light that is either blue or purple at any moment. The color of each light alternates periodically: blue for certain duration and then purple for another duration. Traffic is permitted to travel down the road between any two junctions, if and only if the lights at both junctions are the same color at the moment of departing from one junction for the other. If a vehicle arrives at a junction just at the moment the lights switch it must consider the new colors of lights. Vehicles are allowed to wait at the junctions. You are given the city map which shows

- the travel times for all roads (integers),

- the durations of the two colors at each junction (integers)

- the initial color of the light and the remaining time (integer) for this color to change at each junction.

Your task is to find a path which takes the minimum time from a given source junction to a given destination junction for a vehicle when the traffic starts. In case more than one such path exists you are required to report only one of them.

4.2.2   Floyd-Warshall

Dijkstra is nice when we are dealing with edges with nonnegative weights and are looking for the distances from one vertex to all the others. Floyd-Warshall solves the shortest path problem for all pairs of vertices in $O(V^3)$ time, which is faster than V single-source Dijkstra runs on a dense graph. Floyd-Warshall works even if some edge weights are negative but not if the graph has a negative cycle.

Algorithm 3 Floyd-Warshall
---
    for all vertices v **do**
        $dist(v, v) = 0$
    for all edges (u, v) **do**
        $dist(u, v) = weight(u, v)$
    for all vertices k **do**
        for all vertices i **do**
            for all vertices j **do**
                if $dist(i, j) > dist(i, k) + dist(k, j)$ **then**
                    $dist(i, j) \leftarrow dist(i, k) + dist(k, j)$

4.2.3   Bellman-Ford

Bellman-Ford is a single-source $O(VE)$ shortest path algorithm that works when edge weights can be negative. It is preferable to Floyd-Warshall when the graph is sparse and we only need the answer for one source. Like Floyd-Warshall, the algorithm fails if the graph contains a negative cycle, but the algorithm is still useful for detecting negative cycles.

The idea here is the shortest path, assuming no negative cycles, has length at most $V - 1$.

Algorithm 4 Bellman-Ford

---
    for all vertices v **do**
        $dist(v) \leftarrow \infty$
        $prev(v) =\leftarrow -1$
    $dist(src) \leftarrow 0$
    for $i \equiv 1, V - 1$ **do**
        for all edges (u, v) **do**
            if $dist(u) + weight(u, v) < dist(v)$ **then**
                $dist(v) \leftarrow dist(u) + weight(u, v)$
                $prev(v) \leftarrow u$
    for all edges (u, v) **do**   ▷ check for negative cycles
        if $dist(u) + weight(u, v) < dist(v)$ **then**
            negative cycle detected

4.3   Minimum Spanning Tree

Consider a connected, undirected graph. A *spanning tree* is a subgraph that is a tree and contains every vertex in the original graph. A *minimum spanning tree* is a spanning tree such that the sum of the edge weights of the tree is minimized. Finding the minimum spanning tree uses many of the same ideas discussed earlier.
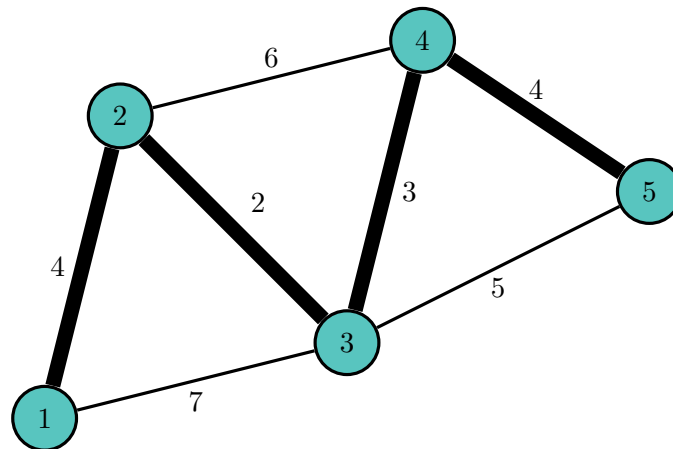
4.3.1   Prim

Prim's algorithm for finding the minimum spanning tree is very similar to Dijkstra's algorithm for finding the shortest path. Like Dijkstra, it iteratively adds a new vertex at a time to build a tree. The only difference is $dist(v)$ stores the shortest distance from *any* visited node instead of the source.

Algorithm 5 Prim

> **for all** vertices v **do**
> > $dist(v) \leftarrow \infty$
> > $visited(v) \leftarrow 0$
> > $prev(v) \leftarrow -1$
>
> $dist(src) \leftarrow 0$
> **while** $\exists v$ s.t. $visited(v) = 0$ **do**
> > $v \equiv v$ s.t. $visited(v) = 0$ with min $dist(v)$
> > $visited(v) \leftarrow 1$
> > **for all** neighbors u of v **do**
> > > **if** $visited(u) = 0$ **then**
> > > > **if** $weight(v, u) < dist(u)$ **then**
> > > > > $dist(u) \leftarrow weight(v, u)$
> > > > > $prev(u) \leftarrow v$

The proof of correctness is left as an exercise. The complexity of this algorithm depends on how the minimum unvisited vertex is calculated. Using the same approaches as Dijkstra, we can achieve $O(V^2)$ or $O(E \log V)$.

4.3.2   Kruskal

While Prim greedily adds vertices to the tree, Kruskal's algorithm greedily adds edges. It iterates over all the edges, sorted by weight. We need to watch out for adding a cycle, breaking the tree structure, which means we need to keep track of each vertex's connected component. If an edge connects two vertices from the same connected component, we don't want to add it to our tree. However, we have a union-find algorithm that works perfectly for this.

Algorithm 6 Kruskal

> **for all** edges (u, v) in sorted order **do**
> > **if** $\textsc{Find}(u) \neq \textsc{Find}(v)$ **then**
> > > add (u, v) to spanning tree
> > > $\textsc{Union}(u, v)$

This algorithm requires a sort of the edges and thus has complexity $O(E \log E) = O(E \log V)$.

1. (SPOJ, INVENT) Given tree with N $(1 \le N \le 15,000)$ vertices, find the minimum possible weight of a complete graph (a graph where every pair of vertices is connected) such that the given tree is its unique minimum spanning tree.

4.4   Eulerian Tour

An *Eulerian tour* of a graph is a path that traverses every edge exactly once. If the tour ends exactly where it started, it is called an *Eulerian circuit*. A graph has an Eulerian circuit if it is connected

and every vertex has even degree. A graph has an Eulerian path if it is connected and all vertices but exactly two have even degrees. The mathematical proofs for these graph properties hinge on the idea that removing a cycle from the graph maintains the Eulerian property. We construct an Eulerian tour by appealing to this idea.

Algorithm 7 Eulerian Tour

---

   function FINDTOUR(v)
      while v has a neighbor u **do**
         delete edge (v, u)
         FINDTOUR(u)   ▷ FINDTOUR(u) must trace a circuit back to v
      add v to tour

It is not preferable to use the run-time stack; we can use our own stack if necessary.

If the graph contains an Eulerian circuit, we call this function on any vertex we like. If it contains an Eulerian path, we call this function on one of the vertices with odd degree.

1. (USACO Training, Airplane Hopping) Given a collection of cities, along with the flights between those cities, determine if there is a sequence of flights such that you take every flight exactly once, and end up at the place you started. (In other words, find an Eulerian circuit on a directed graph.)

2. (USACO Training, Cows on Parade) Farmer John has two types of cows: black Angus and white Jerseys. While marching 19 of their cows to market the other day, John's wife Farmeress Joanne, noticed that all 16 possibilities of four successive black and white cows (e.g., *bbbb*, *bbbw*, *bbwb*, *bbww*, ..., *wwww*) were present. Of course, some of the combinations overlapped others.

   Given N ($2 \leq N \leq 15$), find the minimum length sequence of cows such that every combination of N successive black and white cows occurs in that sequence.

   (The answer is not hard to guess, but use Eulerian circuits to prove that it is correct.)

4.5   Maximum Flow

We are given a directed connected graph with weighted edges, a source node, and a sink node. Each edge weight represents the maximum capacity of that edge in the "flow." For every node besides the source and the sink node, total flow in is equal to total flow out. Our goal is to maximize total flow from source to sink.

A natural approach to "solving" this problem would be to use the greedy algorithm:

Find a path from the source to the sink with the maximum flow; that is, the path with the largest capacity of the minimum of the edges along this path. "Send flow" along this path from the source to the sink by decreasing each edge along this path by the capacity of the path. We repeat, and this is guaranteed to terminate since on any given move, we "remove" an edge.

What is wrong with our greedy approach? Consider the following graph:
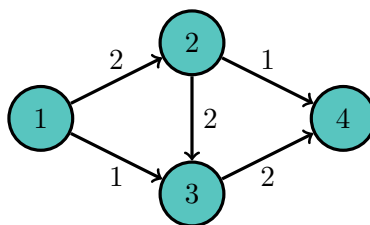
The max flow from vertex 1 to vertex 4 is 3, but greedy gives only 2.

4.5.1   Ford-Fulkerson

Find a path from the source to the sink in which all the edges have positive weight. Find the max flow across this path, which is the minimum weight of any edge on this particular path. Call this value *cap*. Then subtract *cap* from the weight of every edge along the path and *increment the reverse edge* (the edge connecting the same two vertices but running in the opposite direction) by *cap*.

The difference between this algorithm and the greedy approach from earlier is that the paths we now allow may run along a reverse path, essentially undoing any suboptimal flow from earlier. These more general paths are called *augmenting paths*.

This algorithm is guaranteed to terminate for graphs with integral weights. However, it is not guaranteed to terminate for graphs with non-integral capacities. For graphs with integral weights, the run time is $O(Ef)$, where f is the maximum flow and E is the number of edges. In practice, run time tends to be much faster since the algorithm still greedily adds the most flow it can at each step.

4.5.2   Edmonds-Karp

Edmonds-Karp is a slight refinement of Ford-Fulkerson.

4.5.3   Push-Relabel

4.5.4   Max-Flow Min-Cut

Chapter 5

Computational Geometry

For actual geometry problems, and not graph theory problems hiding in the plane.

I'm too lazy to actually write this section right now so here are some useful links from the USACO Training Pages.

`https://www.dropbox.com/s/nqzk63bjby1iaq9/Computational%20Geometry.pdf?dl=0`

`https://www.dropbox.com/s/ykf65dk6sefb6zk/2-D%20Convex%20Hull.pdf?dl=0`

These essentially cover anything I would want to say in this chapter anyway, so I'll likely fill this chapter out last.

5.1 Basic Tools

Cross Product

Dot Product

$\tan^{-1}$, `atan2`

5.2 Formulas

5.2.1 Area

5.2.2 Distance

5.2.3 Configuration

5.2.4 Intersection

5.3 Convex Hull

Chapter 6

Strings

6.1 Rabin-Karp

6.2 Knuth-Morris-Pratt

`https://activities.tjhsst.edu/sct/lectures/1415/stringmatching_10_3_14.pdf`

6.3 Aho-Corasick

6.4 Trie

6.5 Suffix Tree

6.6 Suffix Array

Chapter 7

More Stuff

Here are some more tools to stuff in the toolbox. This chapter introduces more complex ideas that are useful for solving USACO Gold problems and beyond.

7.1 $\sqrt{n}$ Bucketing

$\sqrt{n}$ bucketing is a relatively straightforward idea – given n elements $\{a_i\}_{i=1}^n$ in a sequence, we group them into $\sqrt{n}$ equal-sized buckets. The motivation for arranging elements like this is to support an operation called a *range query*.

Let's take a concrete example. Suppose we want to support two operations:

- *update*(i, x) – increment the value of a_i by x

- *query*(i, j) – return $\sum_{k=i}^{j} a_k$.

Suppose we simply stored the sequence in an array. *update* then becomes an $O(1)$ operation, but *query* is $O(n)$.

2	4	7	−5	3	6	−3	1	−2	−4	−6	2	8	6	0	−7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Another natural approach would be to store in a separate array the sum of the first i terms in the sequence for every index i, or store the *prefix sums*.

0	2	6	13	8	11	17	14	15	13	9	3	5	13	19	19	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Now *query* becomes an $O(1)$ operation, as we can simply subtract two elements in the array to answer a query. Unfortunately, *update* becomes $O(n)$, as changing the value of an element in the beginning of the sequence forces us to change almost all the values in the prefix sum array.

We can still use this idea, though... what we are looking for is some way to group values into sums such that we only need to change a small number of the sums to *update* and only require a small number of them to *query*.

This leads us directly to a $\sqrt{n}$ bucketing solution. Let's group the 16 elements into 4 groups.

2	4	7	−5	3	6	−3	1	−2	−4	−6	2	8	6	0	−7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

8	7	−10	7
$[1,4]$	$[5,8]$	$[9,12]$	$[13,16]$

We'll keep track of the total sum of each group. Now, if we want to update a value, we need to change only two values – the value of that element in the original array and the total sum of the bucket it is in. When we query a range, we'll take advantage of the sum of the bucket when we can. Highlighted are the numbers we'll need for *query*(7, 15).

2	4	7	−5	3	6	−3	1	−2	−4	−6	2	8	6	0	−7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

| 8	7	−10	7
$[1,4]$	$[5,8]$	$[9,12]$	$[13,16]$

Querying requires access to at most $\sqrt{n}$ bucket sums and $2(\sqrt{n}-1)$ individual values. Therefore we have $O(\sqrt{n})$ query and $O(1)$ update. We are able to improve $O(\sqrt{n})$ update to $O(1)$ because of nice properties of the + operator. This is not always the case for range queries: suppose, for instance, we needed to find the minimum element on a range.

It is often the case that $O(\sqrt{n})$ bounds can be improved to $O(\log n)$ using more complex data structures like segment trees and more complex ideas like 2^n jump pointers, both of which are covered in this chapter. These are, however, more complicated to implement and as such are often comparable in runtime in the contest environment. Steven Hao is notorious for using crude $\sqrt{n}$ bucketing algorithms to solve problems that should have required tighter algorithm complexities. $\sqrt{n}$ bucketing is a crude yet powerful idea; always keep it in the back of your mind.

7.2   Segment Tree

It turns out for our specific sum problem that we can do as good as $O(\log n)$ with a *segment tree*, or *range tree*, or *static BBST*. The essential idea is still the same – we want to group elements in some way that allows us to update and query efficiently.

As the name "tree" suggests, we draw inspiration from a binary structure. Let's build a tree on top of the array, where each node keeps track of the sum of the numbers associated with its children.
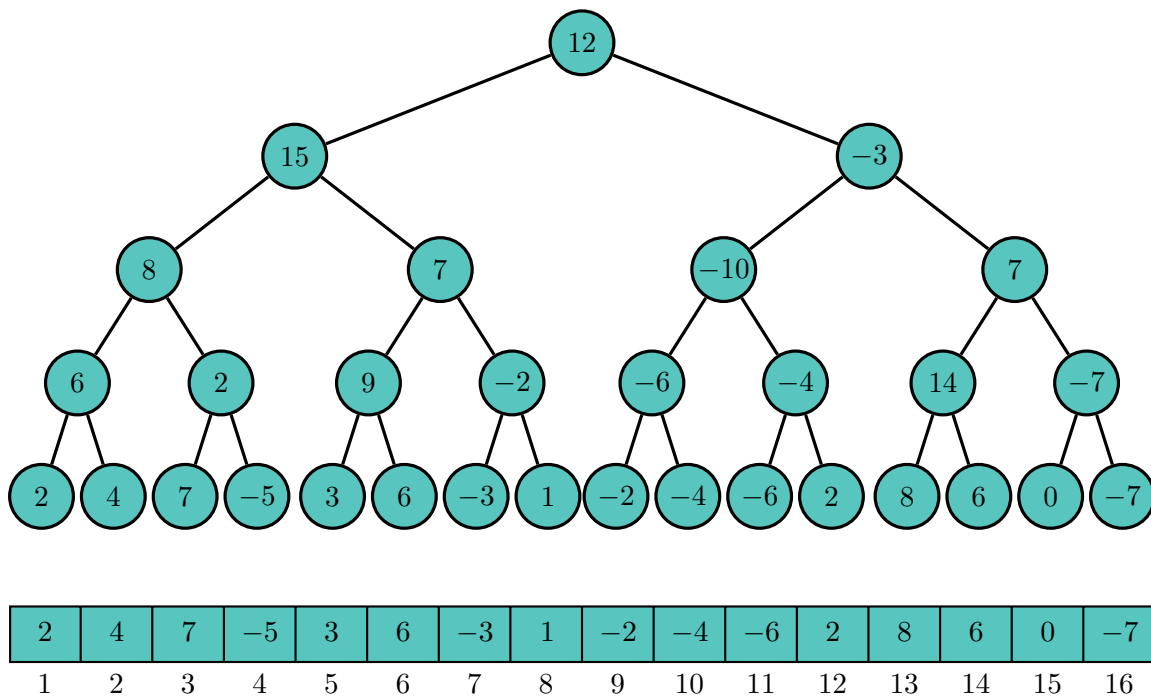
Highlighted are the nodes we'll need to access for $query(7, 15)$. Notice how the subtrees associated with each of these nodes neatly covers the entire range $[7, 15]$.
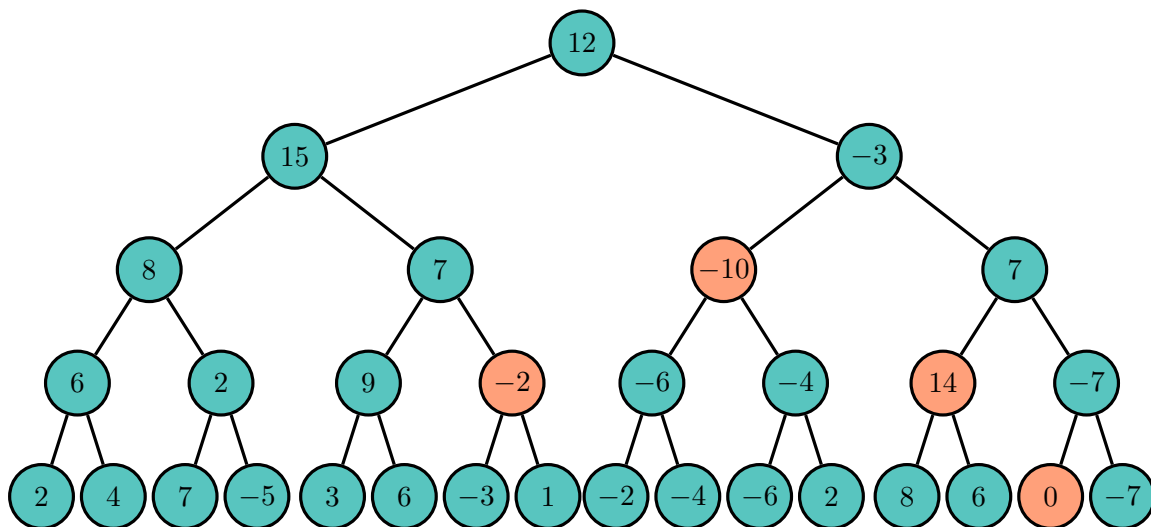
Again, the key idea is if the the interval associated with a node is completely contained within the interval we are querying, we simply return the sum associated with that node. Otherwise, we recurse on the two children. This process is $O(\log n)$ because each level in the tree can have at most two highlighted nodes.

If we wanted to change the third element to 2, we would have to update the highlighted nodes in the following diagram. This process is more straightforward than the querying.
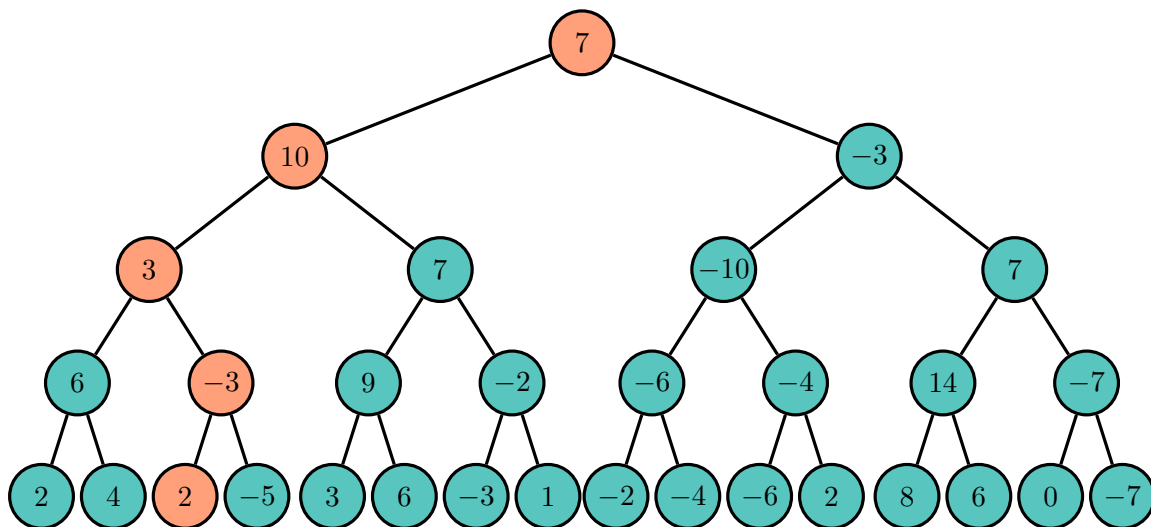
Updating is also $O(\log n)$ as we need to change the values of at most one node in each level in the tree.

I cheated with my example by using a nice power of two, $n = 16$, as the number of elements in the sequence. Of course, the size is not always this nice. One solution is to simply pad the back of the sequence with enough 0s and pretend that the number of elements in the sequence is actually a power of two. For the segment tree, this is not necessary – if the root vertex is associated with the range $[1, n]$, we can simply split this range into two, $\left[1, \left\lfloor \frac{n}{2} \right\rfloor\right]$ and $\left[\left\lfloor \frac{n}{2} \right\rfloor + 1, n\right]$. We then recursively apply this pattern except for when the lower bound of the range is equal to the upper bound. If $n = 12$, the resulting tree would have the following structure.
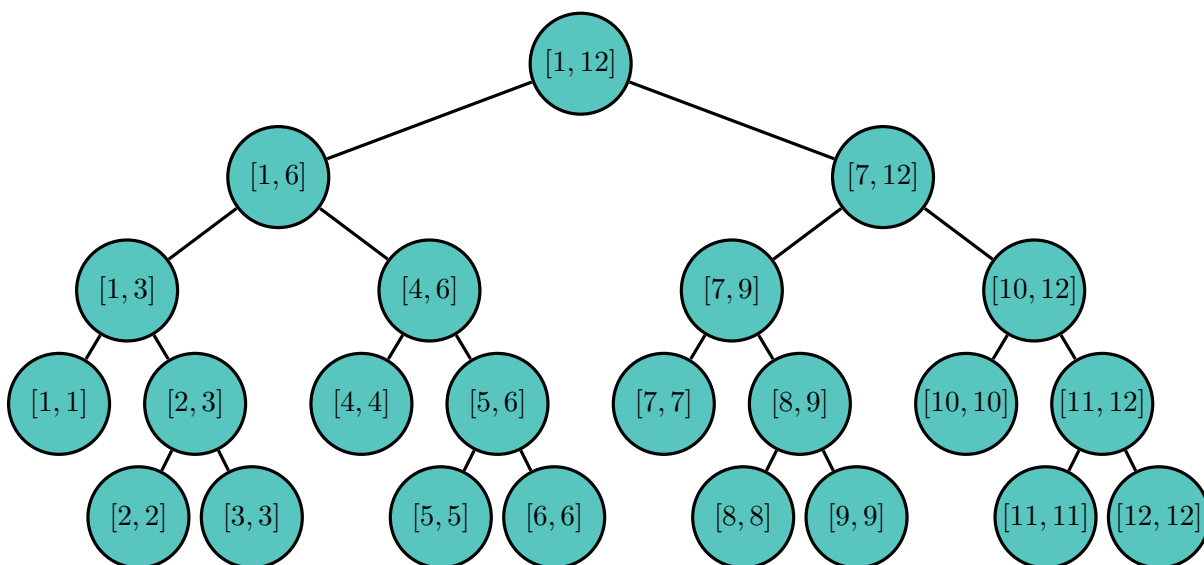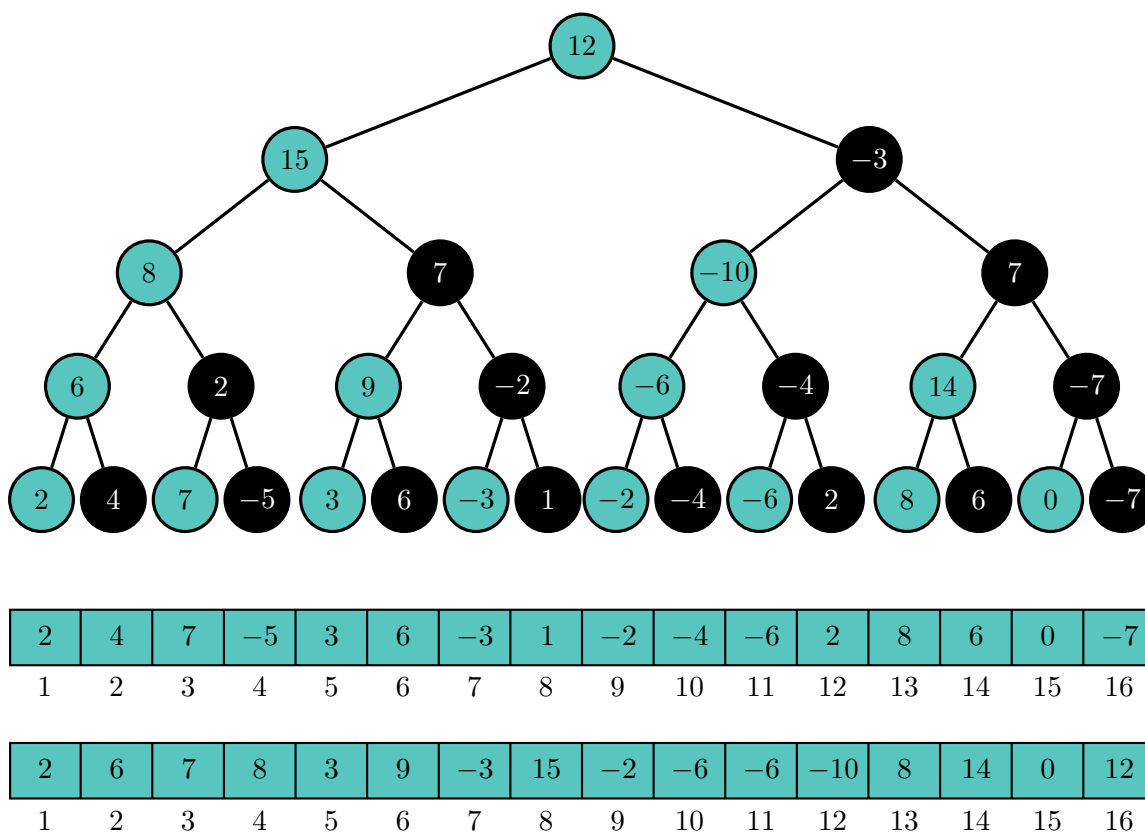

I prefer this approach over padding the sequence with zeros. We see that the segment tree structure does not have to resemble a complete tree at all. However, with this approach, it is still quite balanced, so we can store a segment tree within an array as we would a heap.

7.2.1 Fenwick Tree

A *Fenwick tree*, or *binary indexed tree (BIT)*, is simply a faster and easier to code segment tree when the operator in question has an inverse. Unfortunately, it's not at all intuitive, so bear with me at first and let the magic of the Fenwick tree reveal itself later. In fact, it is so magical that Richard Peng hates it. This is a fair sentiment, because it is rather gimmicky. The key idea is to compress the data stored within a segment tree in a crazy way that ends up having a really slick implementation using some bit operation tricks.

As discussed earlier, the $+$ operator has an inverse, $-$. Therefore, there is an inherent redundancy, for example, in keeping track of the sum of the first $\frac{n}{2}$ elements, the sum of all n elements, and the sum of the last $\frac{n}{2}$ elements, as we do in the segment tree. If we are given only $\sum_{k=1}^{n/2} a_k$ and $\sum_{k=1}^{n} a_k$, we can find $\sum_{k=n/2+1}^{n} a_k$ easily using subtraction.

With this in mind, let's pretend every right child in the tree. We'll mark them as black in the diagram. After that, we'll write out the tree nodes in postfix traversal order, without writing anything whenever we encounter a black node.

2	4	7	−5	3	6	−3	1	−2	−4	−6	2	8	6	0	−7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

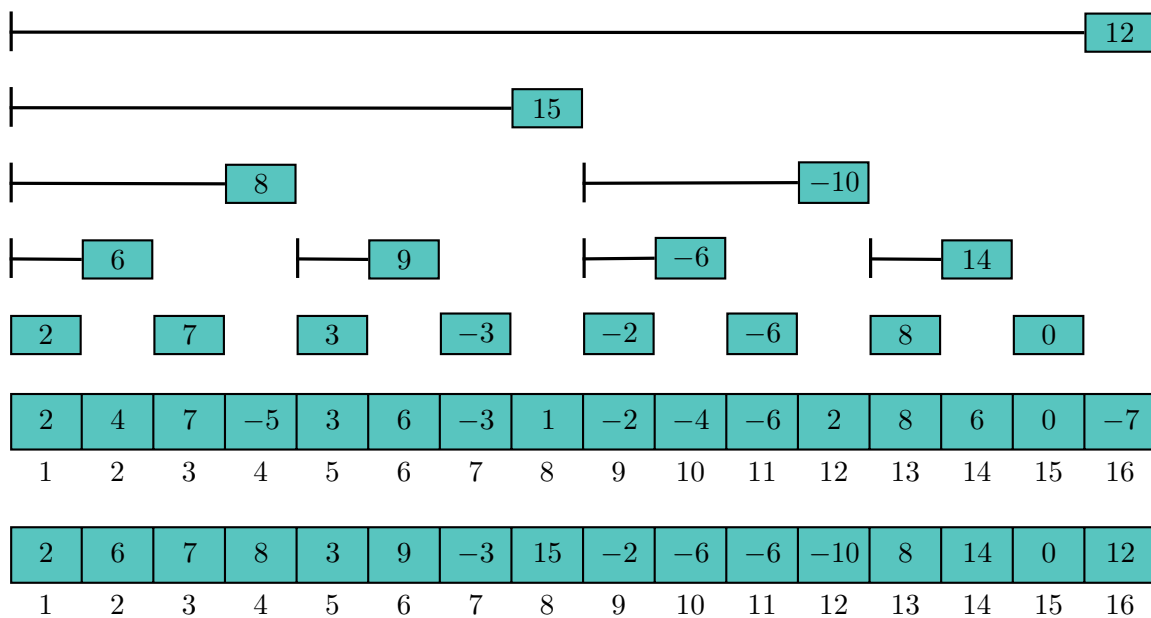| 2	6	7	8	3	9	−3	15	−2	−6	−6	−10	8	14	0	12
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Our Fenwick tree is simply this last array. This should be quite confusing – it is not at all clear why this array resembles a tree, and the numbers in the array make no sense whatsoever right now.

Notice that the final position of every unblackened node is just the rightmost black child in its subtree. This leads to the fact that the ith element in the Fenwick tree array is the sum

$$b_k = \sum_{k=i-2^{v_2(i)}+1}^{i} a_k,$$

where $2^{v_2(i)}$ is simply the greatest power of 2 that divides i. Let's look at a new diagram that hopefully will better illustrate this key property of the random array we just came up with.

2	4	7	−5	3	6	−3	1	−2	−4	−6	2	8	6	0	−7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

2	6	7	8	3	9	−3	15	−2	−6	−6	−10	8	14	0	12
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

All the framework is now in place. Now we need to find out how to query and update the Fenwick tree.

Suppose we wanted to find the sum $\sum_{k=1}^{1} 1a_k$. Let's take a look at the diagram to see which elements we need.
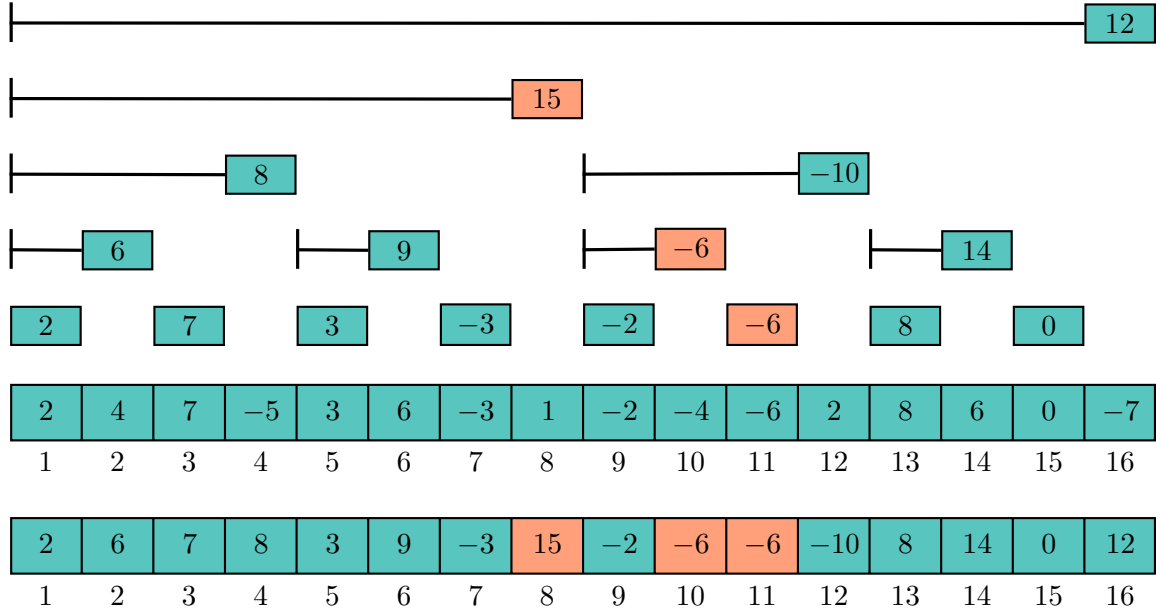
We see that the sum $\sum_{k=1}^{11} a_k = b_8 + b_{10} + b_{11}$. If we look at 11 in binary, we have $11 = 01011_2$. Let's see if we can find a pattern in these numbers in binary:

$$11 = 01011_2,$$
$$10 = 11 - 2^{v_2(11)} = 01010_2,$$
$$8 = 10 - 2^{v_2(10)} = 01000_2,$$
$$0 = 8 - 2^{v_2(8)} = 00000_2.$$

So, we can simply subtract $11 - 2^{v_2(11)} = 10 = 01010_2$, find the sum of the first 10 elements, and add b_{11} to that sum to get the sum of the first 11 elements. We see that repeating this process takes off the last 1 in the binary representation of the number i, and since there are at most $\log n + 1$ 1s in the binary representation $\forall i \in [1, n]$, the query operation is $O(\log n)$.

And now for the update operation. Suppose we want to change the value of a_{11} from -6 to -3. Which numbers will we have to change?
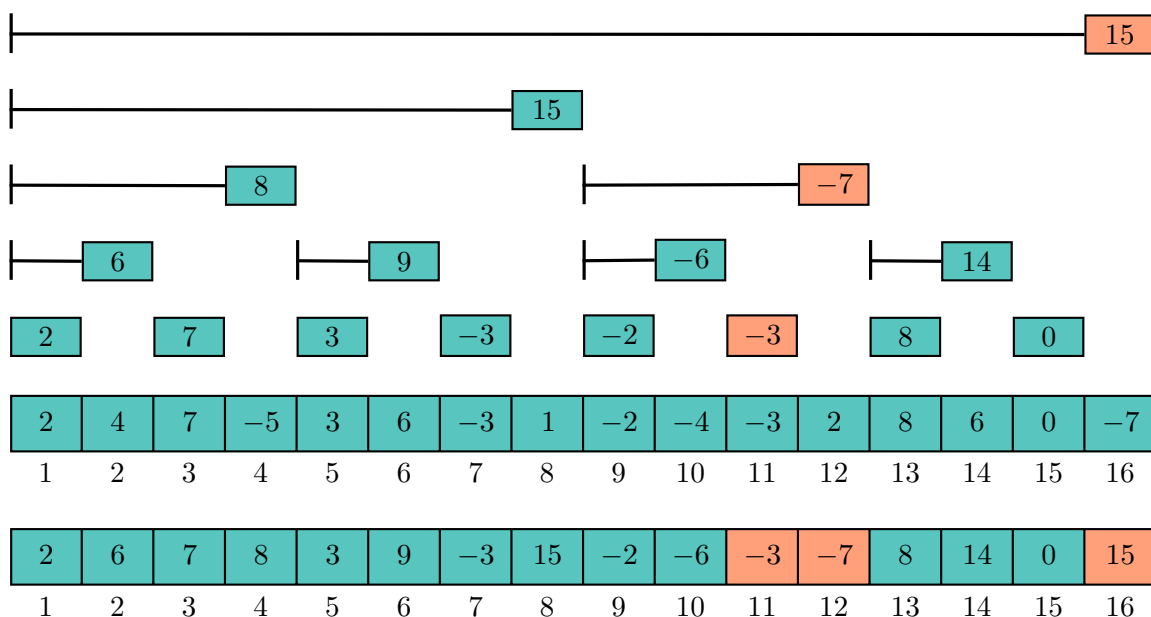
We needed to increment the highlighted values, b_{11}, b_{12}, and b_{16}, by 3. Once again we'll look at 11, 12, and 16 in base 2.

$$11 = 01011_2,$$
$$12 = 01100_2 = 11 + 2^{v_2(11)},$$
$$16 = 10000_2 = 12 + 2^{v_2(12)}.$$

It appears that instead of subtracting the largest dividing power of 2, we are adding. Once again this is an $O(\log n)$ operation.

The real magic in the Fenwick tree is how easy it is to code. The only tricky part is finding exactly what $2^{v_2(i)}$ is. But it turns out, by the way bits are arranged in negative numbers, this is just `i & -i`. With this in mind, here's all the code that's necessary to code a Fenwick tree.

```
int[] b = new int[MAXN]; // Fenwick tree stored as array
void update(int i, int x) {
    for( ; i < MAXN; i += i & -i)
        b[i] += x;
}
int prefixSum(int i) {
    int sum = 0;
    for( ; i > 0; i -= i & -i)
        sum += b[i];
    return sum;
}
int query(int i, int j) {
    return prefixSum(j) - prefixSum(i - 1);
}
```

7.2.2   Lazy Propagation

7.2.3   2D Segment Tree

7.2.4   3D Segment Tree

7.2.5   Persistent Segment Tree

1. (SPOJ, CPAIR) Given a sequence A_k of N ($1 \leq N \leq 100,000$) non-negative integers ($\forall k, A_k < 1000$), Answer Q ($1 \leq Q \leq 100,000$) queries, where each query consists of three integers, v, a, b. The answer is the number of pairs of integers i and j such that $1 \leq i \leq j \leq N$, $a \leq j - i + 1 \leq b$, and $A_k \geq v$ for every integer k between i, and j, inclusive.

7.3   Queue with Minimum Query

7.4   Balanced Binary Search Tree

Recall the red-black tree from earlier. These are pretty nasty to code and don't really have very nice properties, so on programming contests, we generally refer to other kinds of balanced binary search trees. Many problems which can be solved using splay trees and treaps can, however, still be solved using red-black trees, though doing so is generally not preferred unless your name is Sreenath Are.

7.4.1   Splay Tree

7.4.2   Treap

7.5   Euler Tour Technique

The *Euler tour technique* is a method of representing a tree as a list. The idea is a tree is uniquely determined by its DFS traversal order, both going down the tree and going up.

7.6 2^n Jump Pointers

7.7   Fractional Cascading

7.8   Heavy-Light Decomposition

Brute-force algorithms on trees are often fast when the tree is nearly complete, or has small depth. Unfortunately, brute force is far too slow when the tree becomes more and more unbalanced and approximates a linked list.

Certain problems can be solved very nicely in a list with segment trees and similar data structures. Heavy-light decomposition provides us with a way to exploit the fact that long "chains," which

are present in unbalanced trees, might be slow in brute force but can be sped up with other data structures.

Heavy-light decomposition provides us a way to dynamically find the lowest common ancestor as well as a way to prove time complexities about tree data structures, like link-cut trees.

Heavy-light decomposition is a coloring of all the edges in a binary tree either heavy or light. For each vertex v, let $s(v)$ denote the number of nodes in the subtree with v as its head. Then, if u, w are the children of v, $s(v) = s(u) + s(w) + 1$.

We see that the smaller child u or w must have less than half the subtree size as v, and so that child exhibits qualities similar to those of a completely balanced tree. We color the edge connecting v to its lesser child *light*. We color the other edge *heavy*.

We see that from any node, the number of light edges needed to reach the head of the tree is at most $\log n$, as each light edge doubles the subtree size. Then, the number of "heavy chains" along the upwards path from any node is also of the order $\log n$.
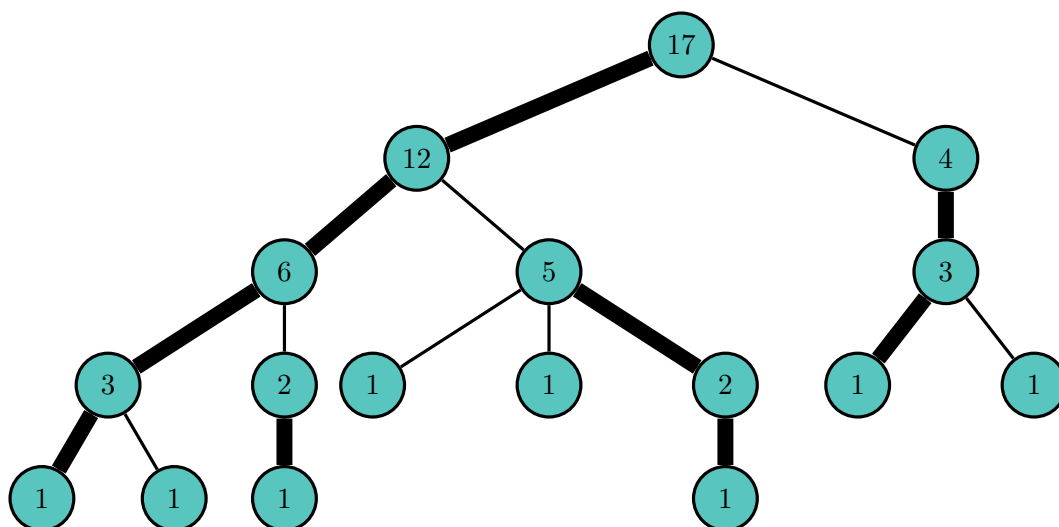
7.9   Link-Cut Tree

7.10   Top Tree

Tree representation of a graph. Scary.

7.11   Link-Cut Cactus

A *cactus* is a graph such that any two distinct cycles in the graph share at most one vertex. Graphs that are not trees are somehow hard to characterize; a cactus is a more general kind of graph that still has some nice properties. In particular, certain tree algorithms still run in polynomial time when modified on a cactus where no such generalization to all graphs exists.

Cacti are scary.

Chapter 8

Math

8.1 Games

`https://activities.tjhsst.edu/sct/lectures/1314/impartial_games_12_06_13.pdf`

8.2 Karatsuba

8.3 Matrices

8.4 Fast Fourier Transform

Beautiful.

8.4.1 Number Theoretic Transform

8.5 Primes