Crash Course Coding Companion

Samuel Hsiang
Thomas Jefferson High School for Science and Technology
samuel.c.hsiang@gmail.com

in collaboration with

Alexander Wei
Phillips Exeter Academy

August 12, 2015

`https://www.dropbox.com/s/z2lur71042pjaet/guide.pdf?dl=0`

`https://github.com/alwayswimmin/cs_guide`

For Rachel

Acknowledgements

I am truly thankful for all my school teachers and USACO instructors, who gave me the knowledge which I wish to pass on through this text. I am grateful for Shankar and Ross, Evan, and Dr. Osborne, who separately made me believe such a task was possible. I must credit Valerie, for she was the one who drove me to really dive into and come to love computer science. Alex and Shwetark were extremely helpful with writing this text, as they provided input and helped me find problems to supplement my work. Finally, I'd like to thank all my friends and avid programmers in the TJ community. This is for all of you, and I hope you find it useful in your studies.

Contents

List of Algorithms

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.[1] 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 too dense and unapproachable for someone who just wants to sit down and learn something interesting.[2] The road, for many, stalls here until college. Everyone should be able to learn the simplest data structures in Java or C++ standard libraries, and someone with problem-solving experience can easily jump right into understanding algorithms and more advanced data structures.

A few important notes, before we begin.

- I'm assuming some fluency in C-style syntax. If this is your first time seeing code, please look somewhere else for now.
- 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. 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.
- It is recommended 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.

[1] In fact, I'm using Evan's template right now. Thanks Evan!

[2] Sedgewick, notably, is getting better. Check out his online companion to *Algorithms, 4th Edition.*

Chapter 1

Fundamentals

1.1 Input and Output

The first part of solving any programming contest problem is reading the input correctly. In this section, we'll briefly go over input and output in Java using `java.util.Scanner` and `java.io.PrintWriter`. There are two scenarios that you should be familiar with: `stdin`/`stdout` and file I/O. You encounter the former when you enter input and see output for a program run in the commandline. You encounter the latter when you have two files (for example `input.txt` and `output.txt`) that you read from and write to.

When using `stdin`/`stdout`, we read input from `System.in` using a Scanner and output our results using `System.out.println()`. `Scanner.nextInt()` and `Scanner.next()` read in integers and strings, respectively. `System.out.println()` prints its argument and adds a newline at the end. (If we don't want the newline, we can use `System.out.print()`.) Here's an example of a main method that takes two integers and outputs their sum:

```
public static void main(String args[]){
  // hint: you should write "import java.util.*;" at the top of your code.
  Scanner sc = new Scanner(System.in);
  int x = sc.nextInt();
  int y = sc.nextInt();
  System.out.println(x + y);
}
```

File I/O is a touch more complicated. For our Scanner, we have to make a new File object and use it in the constructor. We do the same for our PrintWriter. However, PrintWriter also comes with a couple more usage notes. First, we should include `throws IOException` after our main method, since Java requires that we acknowledge the possibility of an IOException. (We bravely assume that our file open will succeed!) After we finish printing, we must also close the PrintWriter to make sure that everything gets written. Here's a snippet showing how Scanner and PrintWriter work with files:

```java
public static void main(String args[]) throws IOException {
  // hint: for file I/O, you should also have "import java.io.*;"
  Scanner sc = new Scanner(new File("input.txt"));
  int x = sc.nextInt();
  int y = sc.nextInt();
  PrintWriter pw = new PrintWriter(new File("output.txt"));
  pw.println(x + y);
  pw.close();
}
```

Although more efficient methods of I/O exist, such as BufferedReader and BufferedWriter, what's covered here should be sufficient for now. (It's possible to read in 10^5 integers with Scanner in a fraction of a second.)

1.2 Complexity

Before we start computing contests, we should understand what the problems on these contests are trying to test. One class of problems, called *implementation problems*, assesses your ability write code quickly and accurately. These problems are only common in easier contests, since they usually don't involve too much thinking or creativity—you just have to implement what's written in the problem statement. Most competitive programming problems ask you to come up with a clever algorithm instead, testing speed and memory efficiency.

To analyze the efficiency of algorithms, computer scientists use a concept called *complexity*. Complexity is measured as a function of the input size; an algorithm could require $3n$, $n^4/3$ or even $2^n + n^2$ steps to finish calculating for an input of size n. To express complexity, we use something called "big-O notation." Essentially, we write the number of steps it takes for an algorithm to finish inside a pair of parentheses with an O in front, like this: $O(\#$ of steps$)$. However, we drop any constant factors and lower order terms from the expression.[1] I'll explain why in a moment; let's look at some examples for now.

Suppose we have three programs that require $3n$, $n^4/3$ and $2^n + n^2$ steps to finish, respectively. The complexity of the first program is $O(n)$ because we don't care about the constant factor 3 on the $3n$. The complexity of the second program is $O(n^4)$; again, we drop the constant. For the last program, we write its complexity as $O(2^n)$ because n^2 is a lower order term.

As to why we drop the constants and the lower order terms, consider the first two programs from above. When $n = 300$, the first program takes 900 steps, while the second program takes $2,700,000,000$ steps. The second program is much slower, despite a smaller constant factor. Meanwhile, if we had a third program that runs in $5n$ steps, it would still only take $1,500$ steps to finish. Constant factors become pretty much irrelevant when we're comparing functions that grow at different rates. The same can be said about lower order terms: n^2 gets dwarfed by 2^n even when $n = 10$.

Thus in programming contests, we usually want a program with the correct complexity, without worrying about too much constant factors. Complexity will be the difference between whether a program gets **accepted** or **time limit exceeded**. As a rule of thumb, a modern processor can do

[1] Unfortunately, this isn't entirely accurate. Saying an algorithm is $O(f(n))$ actually means it takes *at most* $c \cdot f(n)$ steps to finish, for a sufficiently large constant c.

around 10^8 computations each second. When you plug the maximum possible n into the complexity of your algorithm, it should never be much more than that.

We've focused on time and haven't talked much about memory so far, but memory does gets tested. The amount of memory a program uses as a function of n is called its *space complexity*, as opposed to the *time complexity* we discussed earlier. Space complexity however, shows up much less frequently than time complexity. One of the reasons is that you usually run out of time before you can exceed the memory limit.

1.3 Sorting

To further explore the concept of complexity, we will use sorting algorithms as a case study. Sorting is just as it sounds—we're given a collection of objects, and we want to sort them into a predefined order.

1.3.1 Insertion Sort

Insertion sort builds up a sorted list by inserting new elements one at a time. Inserting an element into a sorted list takes time proportional to the length of the list, so the runtime of this algorithm is $1+2+3+\cdots+n$, which is $O(n^2)$. One way to think about this is to iterate i from 1 to n, and let the first i elements be our sorted list. To insert the $(i+1)$th element, we just swap it with the largest, the second largest, and so on, until it's greater than than the next largest element. Then we have a sorted list in the first $(i+1)$ entries of the array. Insertion sort, despite being slower than merge sort and quicksort, is still useful because of its efficiency on small inputs. Many implementations of merge sort and quicksort actually use insertion sort once the problem size gets small.

Around how long is the longest list that you can sort with insertion sort in less than a second?

1.3.2 Merge Sort

The idea behind merge sort is the following observation: If we are given two sorted lists of length $n/2$, we only need n comparisons to merge them into one sorted list of length n. So we can divide and conquer. We do this by cutting the array in half, sorting each half recursively, and merging the two halves back together. Because our recursion goes $\log_2 n$ levels deep, this algorithm runs in $O(n \log n)$. Try to work out the details of this yourself. (You can in fact prove that $O(n \log n)$ comparisons is optimal for sorting algorithms, one of the few problems in computer science that has a non-trivial lower bound.)

Around how long is the longest list that you can sort with merge sort in less than a second?

1.3.3 Quicksort

Quicksort also uses a divide and conquer strategy to run in $O(n \log n)$ on average.[2] We first take a random element from the array, called the *pivot*. Then, we move anything less than the pivot to the left of the array, anything greater than the pivot to the right of the array, and recurse on the

[2] A bound "on average" is also called an *amortized bound*.

two halves that we just created. We say that quicksort runs in $O(n \log n)$ only on average because there are cases that can make quicksort run in $O(n^2)$. What would happen if we chose the smallest element of the array as the pivot each time?

Quicksort was previously implemented by `Arrays.sort()` and `Collections.sort()`, but Java now uses dual-pivot quicksort and timsort.

1.3.4 Comparable

We'll finish our section on sorting by taking a look at how Java sorts objects. In `java.util.Arrays`, there is a built-in sort method, `Arrays.sort()`, which takes an array and sorts it. If we want to sort an array of our own objects, we have to first implement the `Comparable` interface. This is how Java orders a set of objects. Implementing `Comparable` requires support of the method `int compareTo(Object o)`. If `a.compareTo(b) < 0`, then `a` comes before `b`; if `a.compareTo(b) > 0`, then `a` comes after `b`; if `a.compareTo(b) == 0`, then `a` is considered equal to `b`. (`a.compareTo(b) == 0` should be equivalent to `a.equals(b)`.) For example, consider the 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
    }
}
```

If you know generics (Section 2.1), we can clean this up a bit. Note that no casting is required anymore.

```
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;
    }
}
```

1.4 Binary Search

Before we begin studying data structures in Chapter 2, it is necessary to first understand some search techniques. Suppose we're given a list and we want to check if it contains some element. We

can do this directly by looking at every element of 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 to search because we can compare our element to other elements in the list. In fact, we only need to look at $O(\log n)$ elements to check if the queried element exists.

Consider a sorted array that supports $O(1)$ access to any element. We can compare our queried element to the middle element of the array. Depending on whether our element is larger, smaller, or equal to the middle element, we'll be able to eliminate at least half of the array. Thus finding an element in a sorted array with *binary search* is an $O(\log n)$ operation. We can implemented this either iteratively or recursively.

Chapter 2

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 in a language's API. For Java, this is 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

In general, a data structure can store any kind of data, ranging from integers to strings to other data structures. We therefore want to implement data structures that can hold any and all kinds of information. When we use a data structure, however, we might want our structure to store only one kind of information: only strings, for example, or only integers. We use *generics* to specify to an external structure that we only want it to store a particular kind of information.

```
1 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. We can write `Integer i = al.get(0)` without any need to cast to an Integer.

I don't think the beginning programmer needs to know how to necessarily code a class that supports generics, since each language has its own complex set of rules governing generics. However, we use the standard library extensively in any coding environment, so it is necessary to use a class that does support generics. I think standard classes are relatively straightforward to use but can be annoying to actually implement.

When examining a language's API or explanations of implemented functions in this chapter, the characters `E`, `V`, and `K` represent generic classes. For example, when we set `al = new ArrayList<Integer>()`, `E` represents `Integer`. Otherwise, `E` simply means any object.

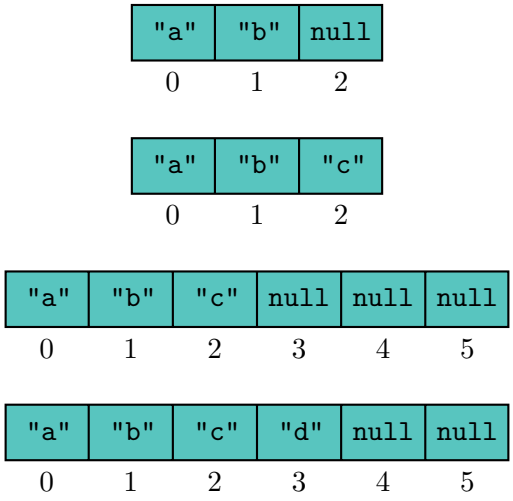
2.2 List

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 Size-Adjustable Array

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. If the backbone array runs out of space, we create a new array with double the size and keep going as if nothing happened.



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 structure 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.

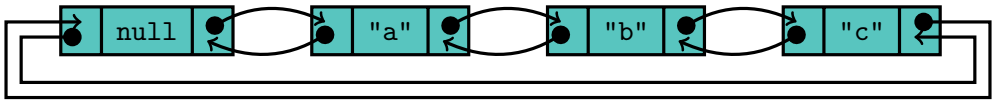
The LinkedList attempts to remedy this. It trades $O(1)$ access to any element in the list for an easier way to remove elements from either end of the list easily. Consider a chain of paper clips:

[1]

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:

[1] `http://img.thrfun.com/img/078/156/paper_clip_chain_s1.jpg`



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.

2.3 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.

2.4 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.

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

Instead, we must do something like this:

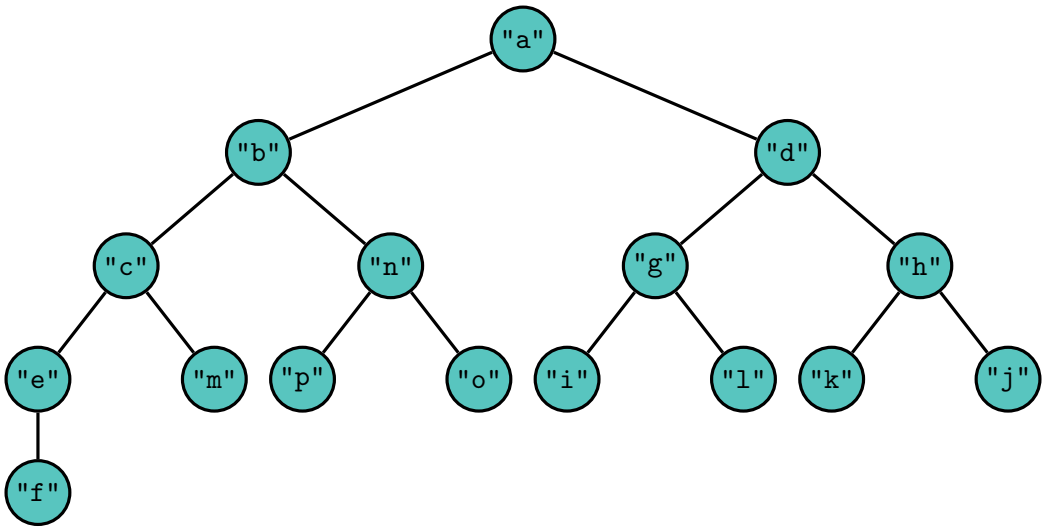
```
1 Queue<String> q = new Queue<LinkedList>();
```

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.

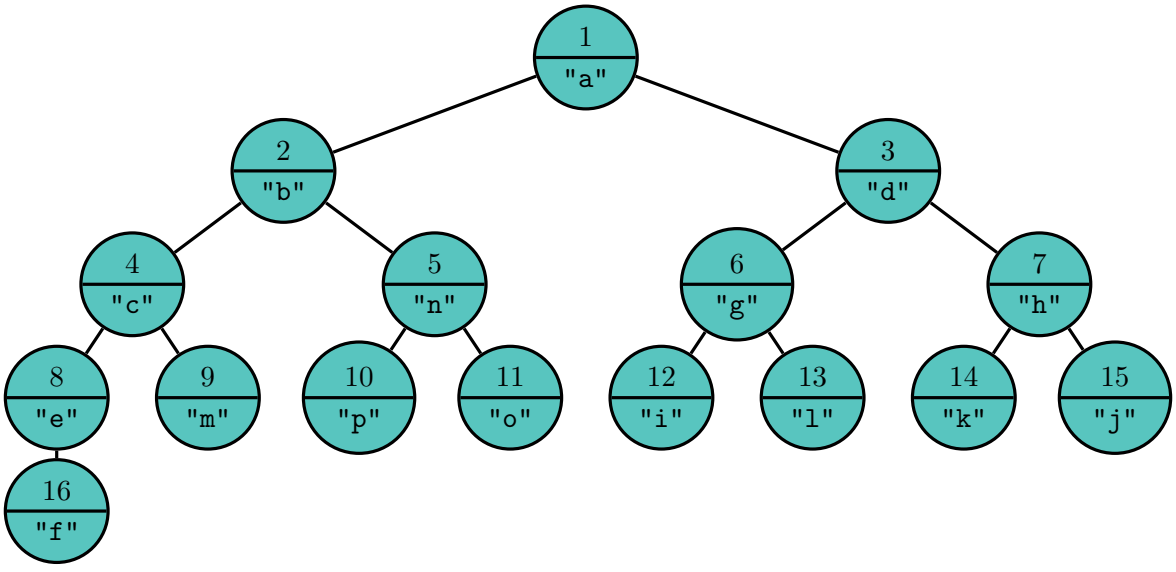
2.5 PriorityQueue

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.



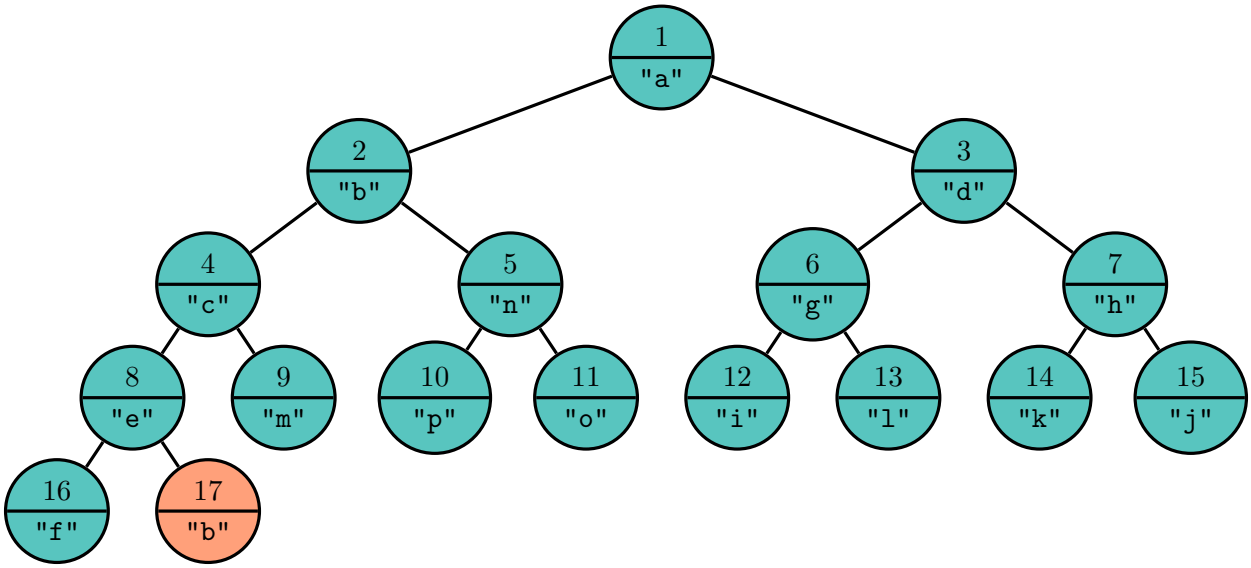
We see that the root of the tree will always be the top element. It is tempting to use a container class with a pointer to its left and its right child. However, we have a much nicer way to store *complete* binary trees with an array. Consider the following numbering of the nodes:



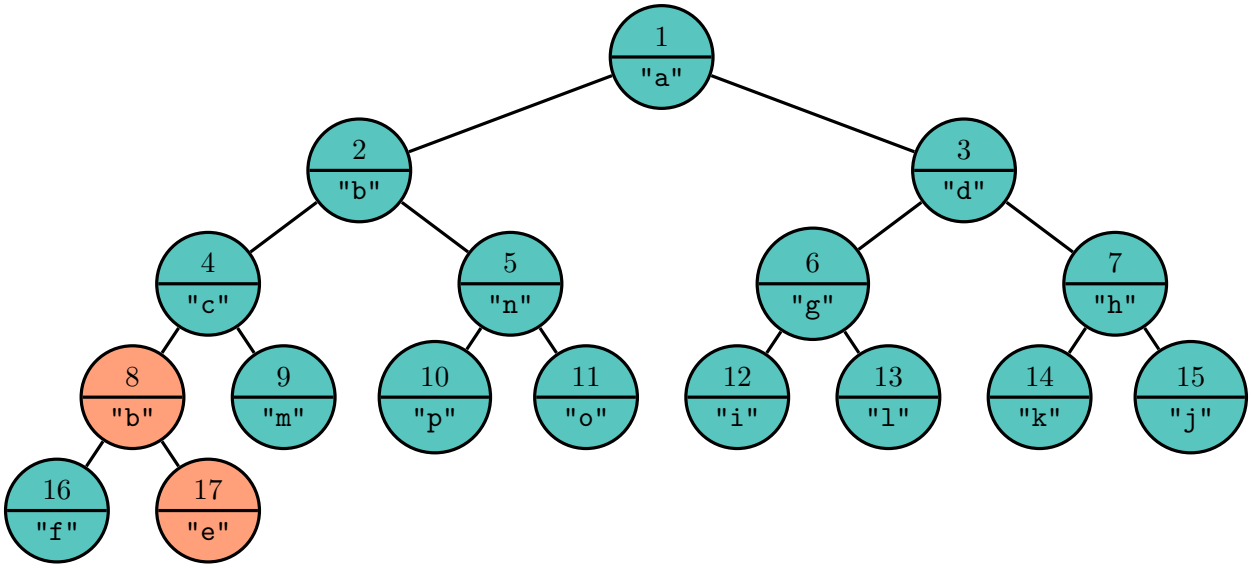
We see that every number from 1 to 16 is used, and for every node, if the index associated with it is i, the left child is $2i$, and the right child is $2i + 1$. This leads to a very natural implementation of the tree in an array:

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

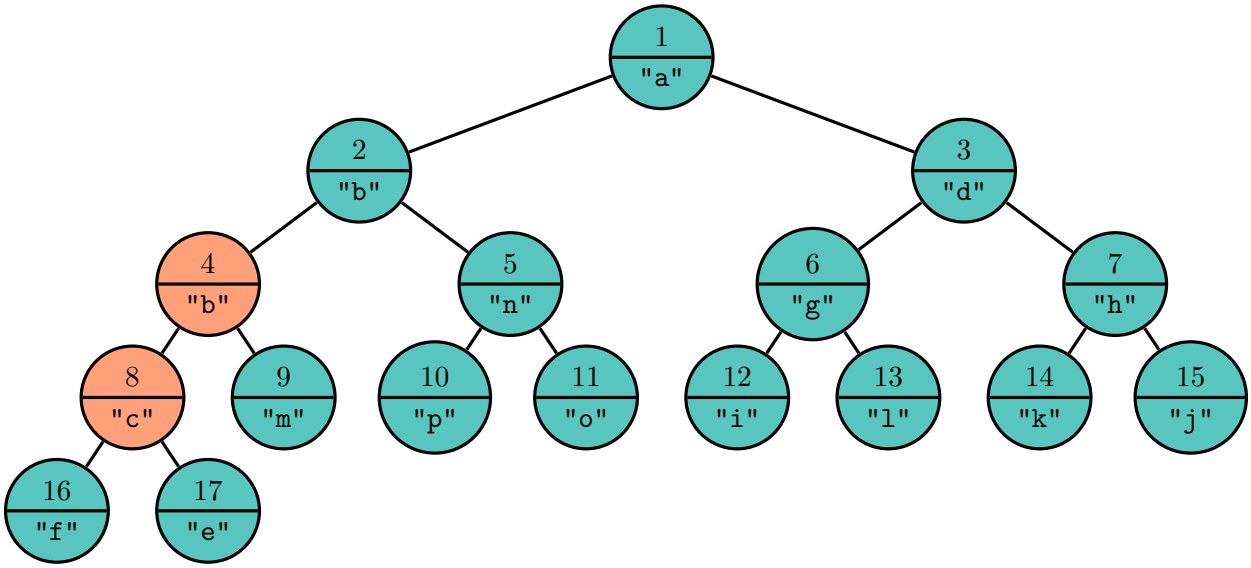
How do we add elements to our heap, while maintaining the heap qualities? Well, let's just add it to the very end and see what we get. Suppose we are to add `"b"` to the tree.



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.

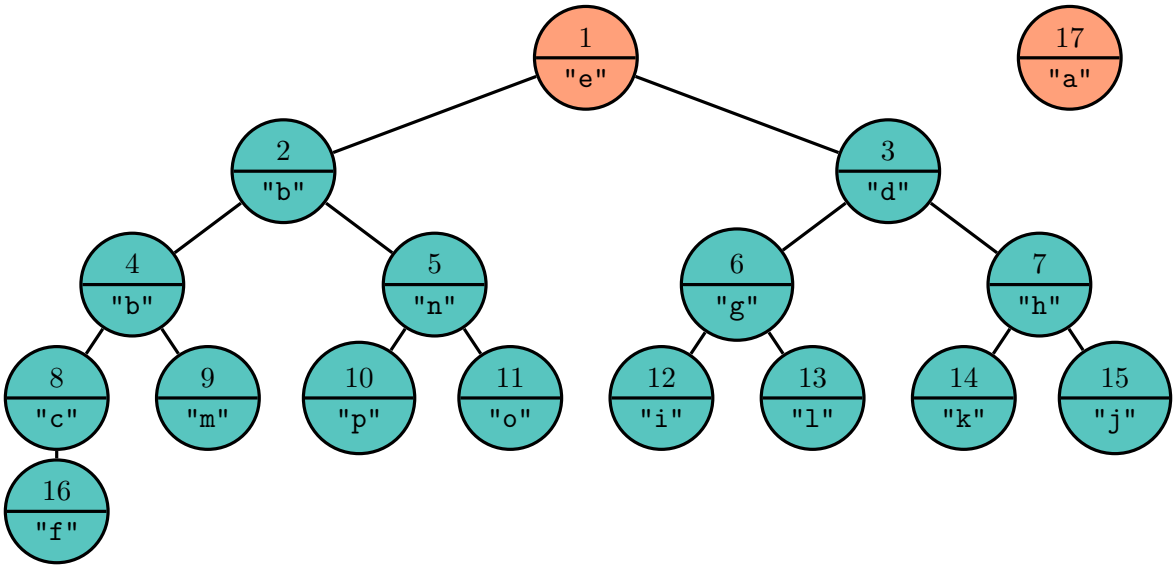


One more swap...

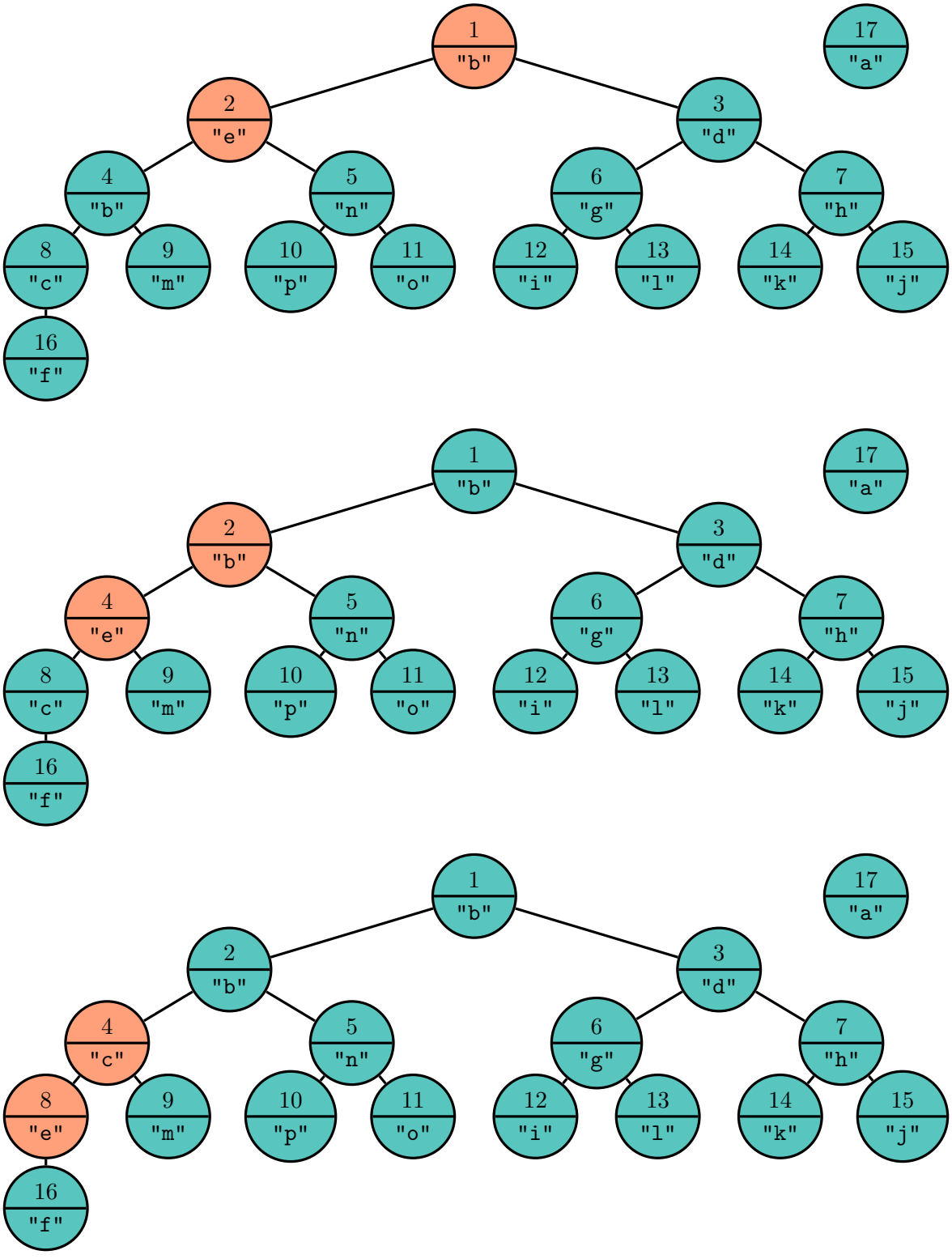


And now we have the heap property restored. As the tree has depth at most $\log n$, this process is $O(\log n)$.

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!

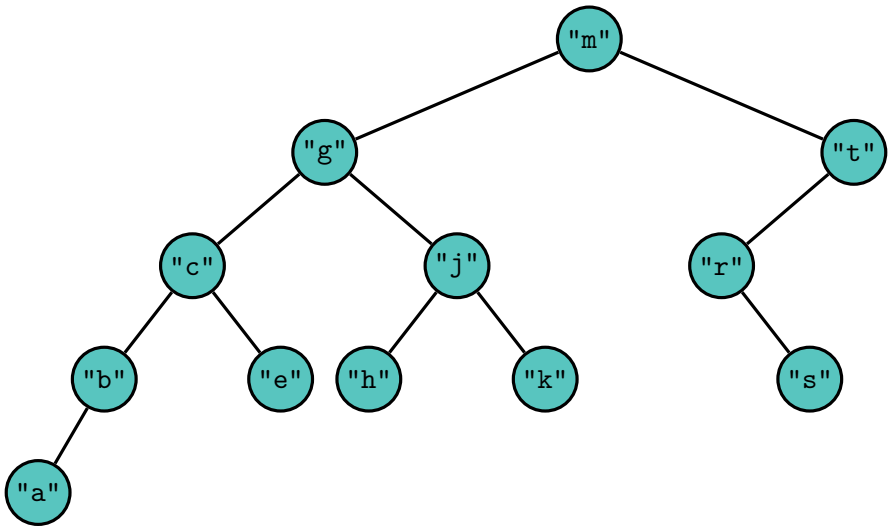
- `boolean add(String s)`
- `boolean isEmpty()`
- `String poll()`
- `String peek()`
- `int size()`

2.6 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.

2.6.1 TreeSet

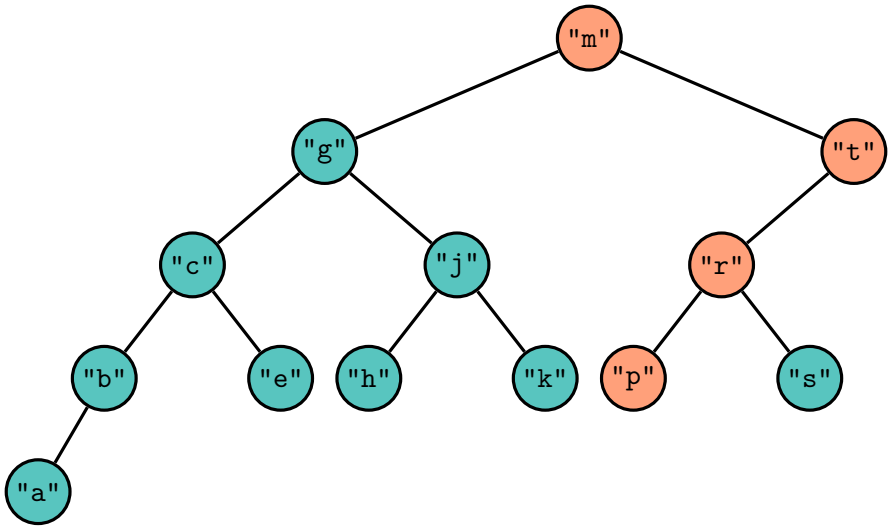
A TreeSet is Java's implementation of a *binary search tree* (BST). A binary search tree is a tree where every node is greater than every node in its left subtree and less than every node in its right subtree. As with a heap, to use a BST, we need to impose some kind of ordering on the elements stored.



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.

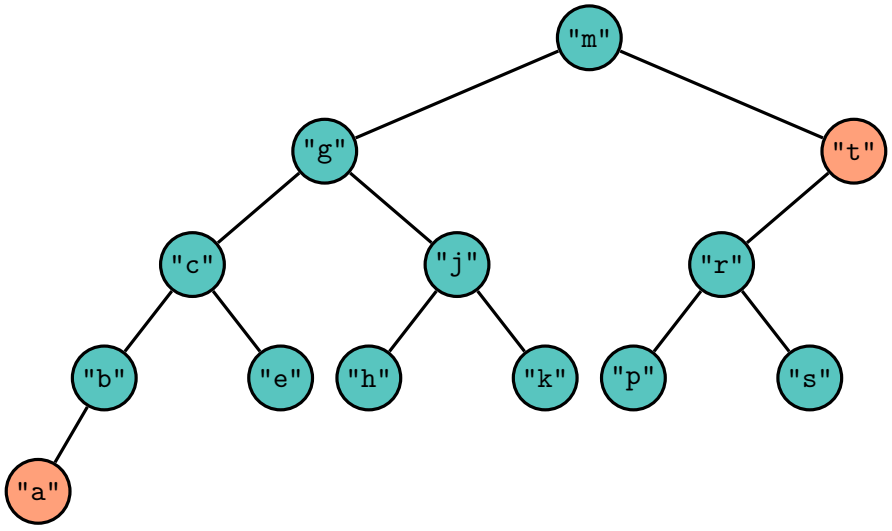
Given the name of the tree, searching for an element within the tree is quite natural, and similar to a binary search. Compare the element to be searched for with the current node. If they are equal, we are done; otherwise, search the appropriate left or right subtree. As with most structures and algorithms 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:

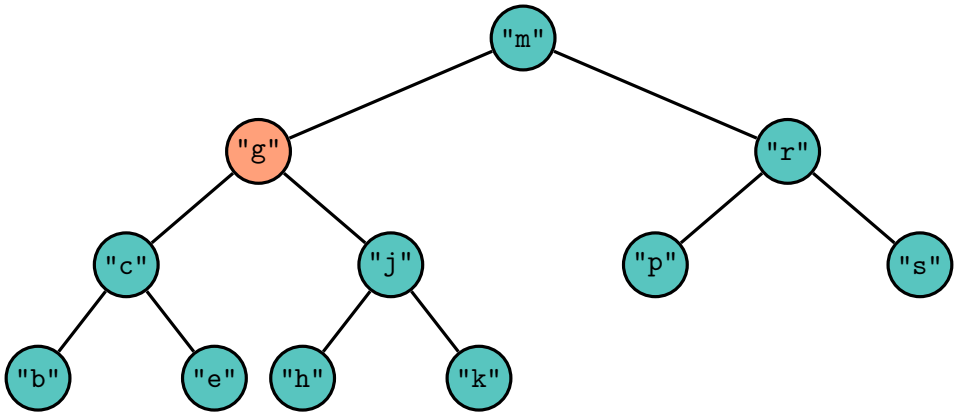


Deleting an element is the annoying part. Unfortunately, there's not much we can do besides casework.

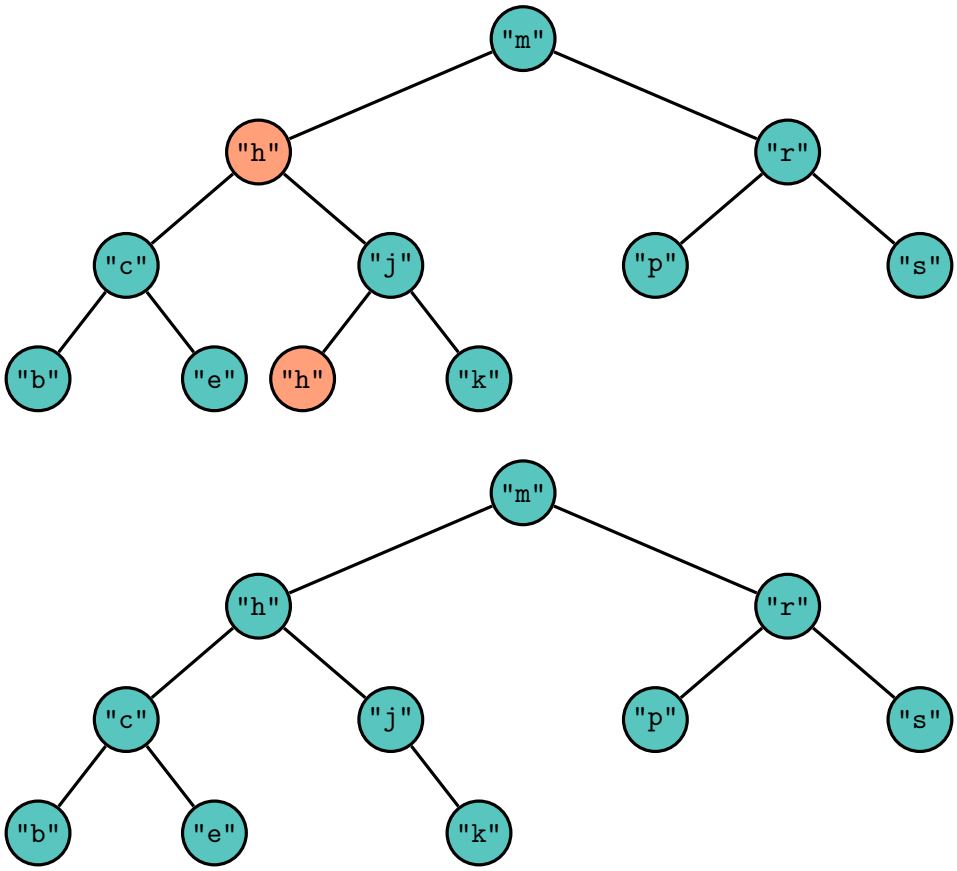
Removing a leaf, like `"a"`, from the tree is very easy. Removing a node with only once child, like `"t"`, is also relatively straightforward.



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.



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.

A red-black tree is a self-balancing BST that guarantees $O(\log n)$ operations by making sure the height of the tree grows logarithmically. 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. How exactly a red-black tree works, together with some more balanced binary search trees which are useful on the competitive scene, are covered in a later chapter.

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.

2.6.2 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:

```
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.

```
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:

```
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()`

2.7 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 Strings.

```
Map<String, String> email = new TreeMap<String, String>();
email.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.

- `String get(Object k)` – return the value assigned to `k`.
- `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`.

2.8 BigInteger

BigInteger is in `java.math` for times when int and long just aren't large enough. The way BigInteger works is it stores a number as an array of ints. Each value in the array represents a digit in some very large base. Addition and subtraction can be done in the standard way. Generally multiplying two BigIntegers is not necessary on contests, but it can be sped up using Karatsuba or the FFT.

2.9 C++ Analogs

- `ArrayList` – `vector`
- `LinkedList` – `list` or `deque`
- `Stack` – `stack`
- `Queue` – `queue`
- `PriorityQueue` – `priority_queue`, but note that `priority_queue` pops *max* element first
- `TreeSet` – `set`

- `HashSet` – `unordered_set`, C++11
- `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 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.

3.1.1 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.

3.1.2 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.

3.1.3 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.

3.2 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.

3.3 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.

3.4 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 both time and 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 ← 0
    for j ≡ i + 1, N do
        if value(i) > value(j) then
            x ← PROCESS(j)
            if x > max then
                max ← x
    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 ← 0
    for j ≡ i + 1, N do
        if value(i) > value(j) then
            x ← PROCESS(j)
            if x > max then
                max ← x
    dp(i) ← 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 ≡ N, 1 do                                   ▷ i goes in reverse
    max ← 0
    for j ≡ i + 1, N do
        if value(i) > value(j) then
            if dp(j) > max then
                max ← dp(j)
    dp(i) ← 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!

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

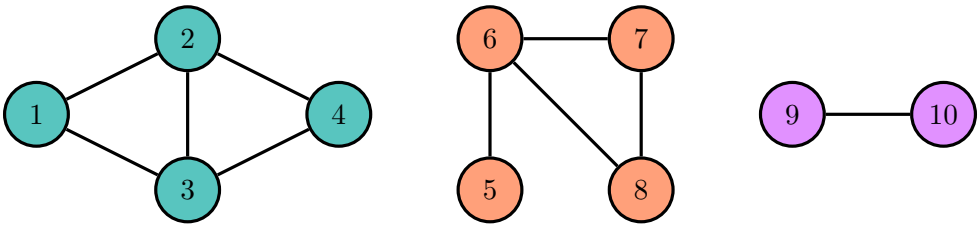
Chapter 4

Graph Algorithms

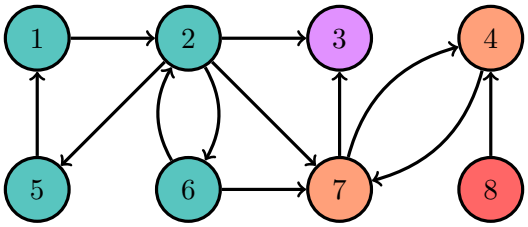
In this chapter we explore some of the most famous results in graph theory.

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 and won't be covered in this chapter.

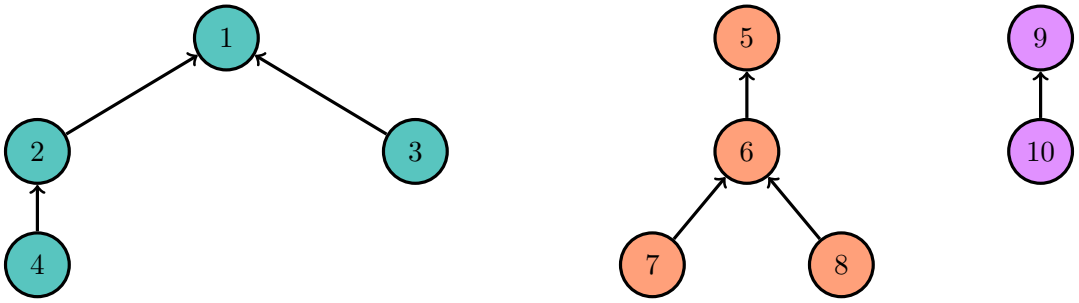
4.1.1 Flood Fill

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 (Disjoint Set Union)

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.

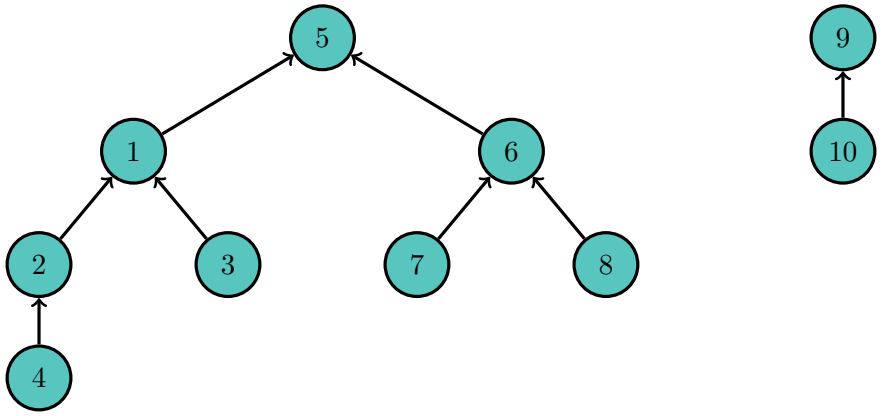


The idea of a pointer can easily be stored within an array.

-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:

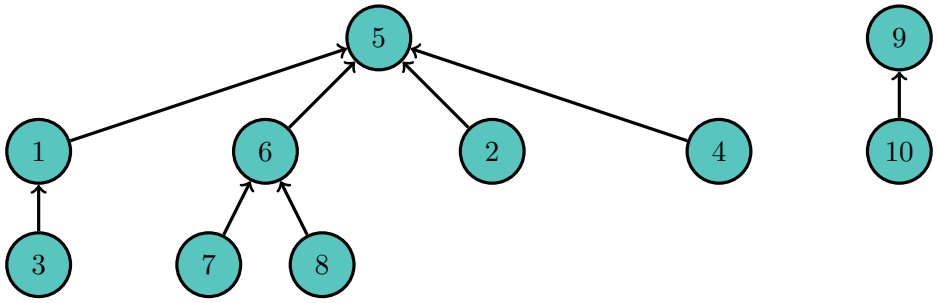


A problem quickly arises – the $find$ operation threatens to become linear. There are two simple things we can do to optimize this.

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) ← FIND(parent(v))
    return parent(v)
function UNION(u, v)
    uRoot ← FIND(u)
    vRoot ← FIND(v)
    if uRoot = vRoot then
        return
    if size(uRoot) < size(vRoot) then
        parent(uRoot) ← vRoot
        size(vRoot) ← size(uRoot) + size(vRoot)
    else
        parent(vRoot) ← uRoot
        size(uRoot) ← size(uRoot) + size(vRoot)
```

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 $E \approx V$.

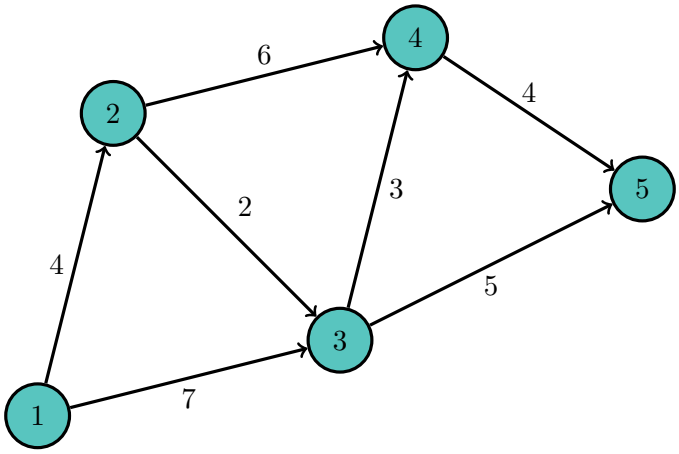
For every vertex v in the graph, we keep track of the shortest known distance $dist(v)$ from the source to v, a boolean $visited(v)$ to keep track of which nodes we "visited," and a pointer to the previous node in the shortest known path $prev(v)$ so that we can trace the shortest path once the algorithm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

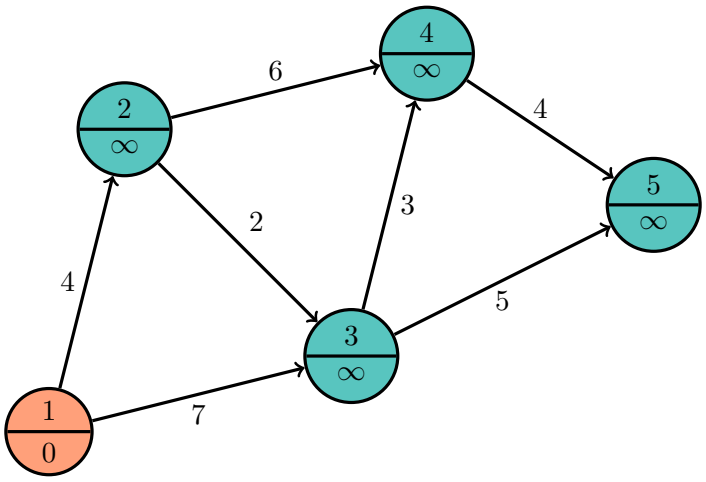
The $O(V^2)$ implementation immediately follows.

Algorithm 2 Dijkstra

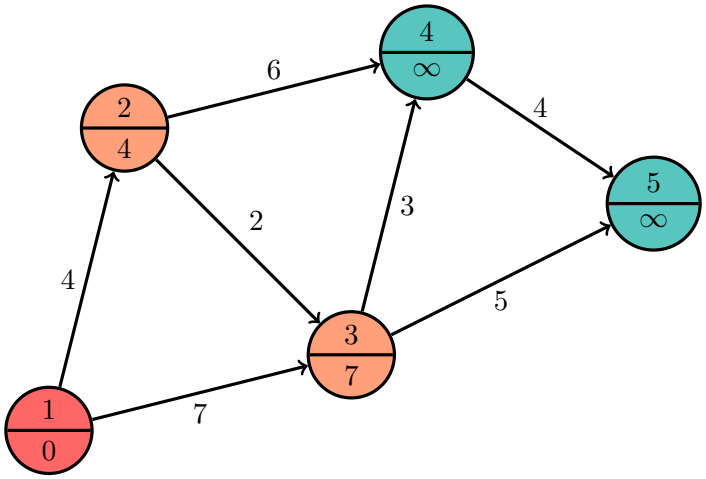
```
for all vertices v do
    dist(v) ← ∞
    visited(v) ← 0
    prev(v) ← −1
dist(src) ← 0
while ∃v s.t. visited(v) = 0 do
    v ≡ v s.t. visited(v) = 0 with min dist(v)
    visited(v) ← 1
    for all neighbors u of v do
        if visited(u) = 0 then
            alt ← dist(v) + weight(v, u)
            if alt < dist(u) then
                dist(u) ← alt
                prev(u) ← v
```

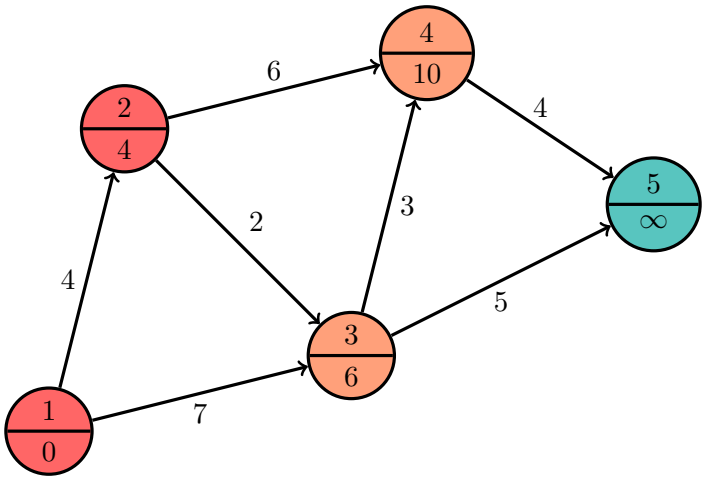


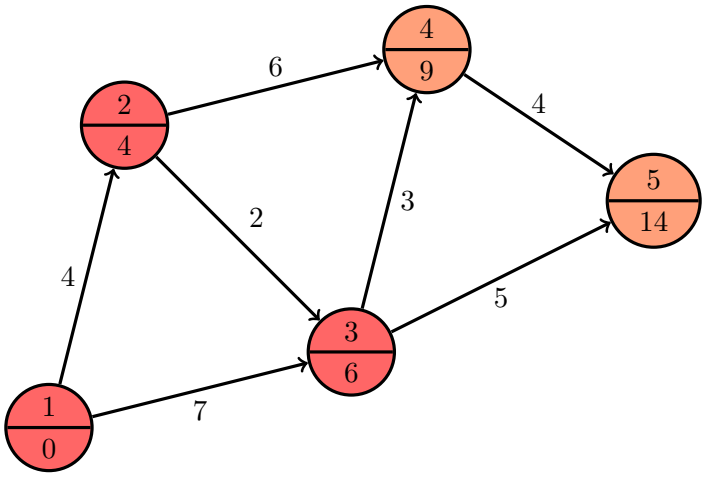
Let's run Dijkstra's algorithm on the above graph with vertex 1 as the source. We first set all the distances besides the source to be ∞.

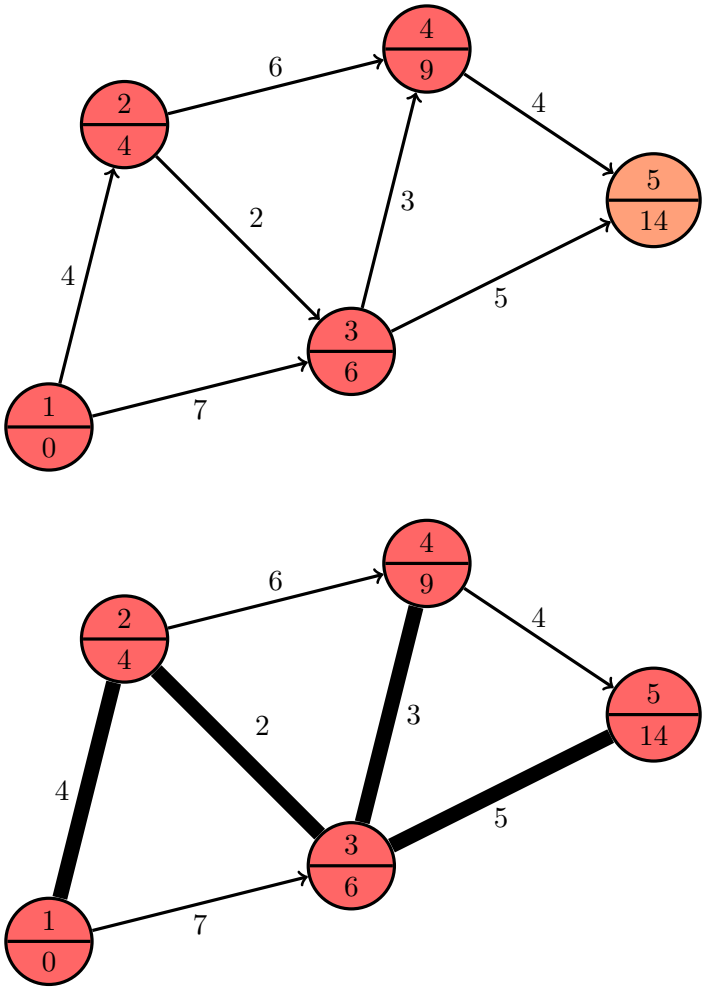


Now, we continue choosing the closest unvisited node, mark it as visited, and and update its neighbors.









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. The main problem with using the standard library heap is having repeated vertices in the heap. We could just ignore this problem and discard visited vertices as they come out of the heap. Alternatively, we could choose never to have repeated vertices in the heap. To do this, 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. Either way, we achieve $O(E \log V)$, as we do $E + V$ updates to our heap, each costing $O(V)$.

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) ← 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.

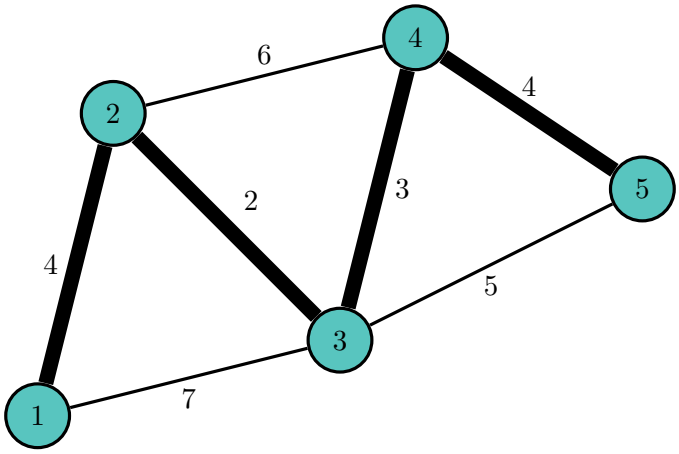
The idea here is the shortest path, assuming no negative cycles, has length at most $V - 1$.

Algorithm 4 Bellman-Ford

```
for all vertices v do
    dist(v) ← ∞
    prev(v) =← −1
dist(src) ← 0
for i ≡ 1, V − 1 do
    for all edges (u, v) do
        if dist(u) + weight(u, v) < dist(v) then
            dist(v) ← dist(u) + weight(u, v)
            prev(v) ← 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) ← ∞
    visited(v) ← 0
    prev(v) ← −1
dist(src) ← 0
while ∃v s.t. visited(v) = 0 do
    v ≡ v s.t. visited(v) = 0 with min dist(v)
    visited(v) ← 1
    for all neighbors u of v do
        if visited(u) = 0 then
            if weight(v, u) < dist(u) then
                dist(u) ← weight(v, u)
                prev(u) ← 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 FIND(u) ≠ FIND(v) then
        add (u, v) to spanning tree
        UNION(u, v)
```

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

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.

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

Complex Ideas and Data Structures

Here we build on previous material to introduce more complex ideas that are useful for solving USACO Gold problems and beyond.

6.1 Dynamic Programming over Subsets ($n2^n$ DP)

We've already covered how dynamic programming can turn exponential solutions into polynomial solutions, but it can also help turn factorial solutions into exponential. Problems where the bound on n is 20, for example, signal that an exponential solution is the one required. 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 cute bijection from the subsets of $\{0, 1, 2, \ldots, N-1\}$ to the integers from 0 to $2^N - 1$. That is, the subset $\{0, 2, 5, 7\}$ maps to $2^0 + 2^2 + 2^5 + 2^7 = 165$ in the bijection. 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** ▷ i represents the subset S
  $dp(i) \leftarrow -1$
  for all $j \in S$ **do** ▷ $j \in S$ satisfy `i & (1 << j) != 0`
    $alt \leftarrow \min(dp(i - 2^j), strength(j) - \sum_{k \in S \setminus \{j\}} weight(k))$
    if $dp(i) < alt$ **then**
      $dp(i) \leftarrow alt$

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

6.2 $\sqrt{n}$ Bucketing

$\sqrt{n}$ bucketing is a relatively straightforward idea – given n elements $\{x_i\}_{i=1}^n$ 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:

- $update(i, z)$ – increment the value of x_i by z
- $query(i, j)$ – return $\sum_{k=i}^{j} x_k$.

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

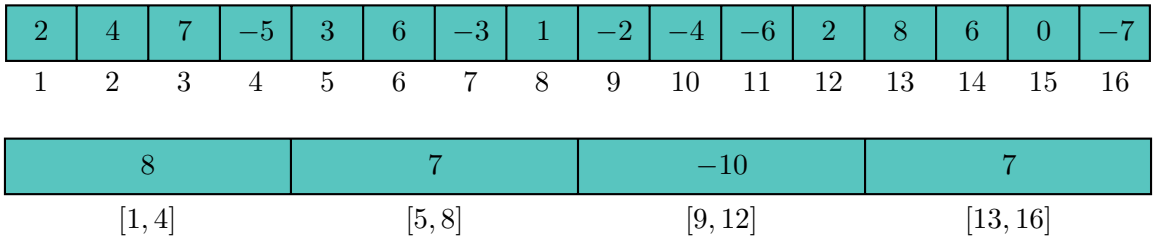
Another natural approach would be to store in a separate array the sum of the first i terms in the sequence for every index i, or store the *prefix sums*.

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

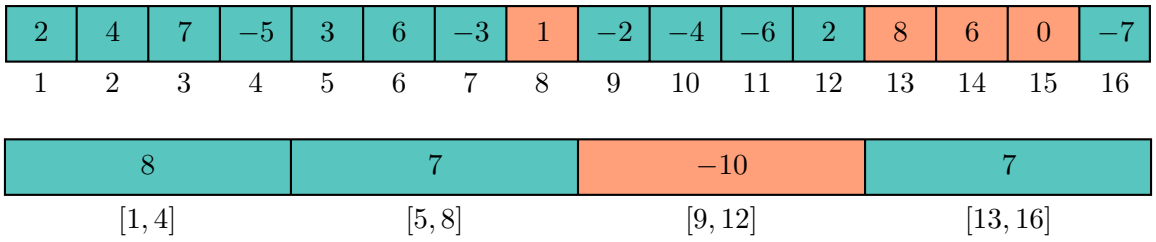
Now *query* becomes an $O(1)$ operation, as we can simply subtract two elements in the array to answer a query. Unfortunately, *update* becomes $O(n)$, as changing the value of an element in the beginning of the sequence forces us to change almost all the values in the prefix sum array.

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.



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 $query(7, 15)$.



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.

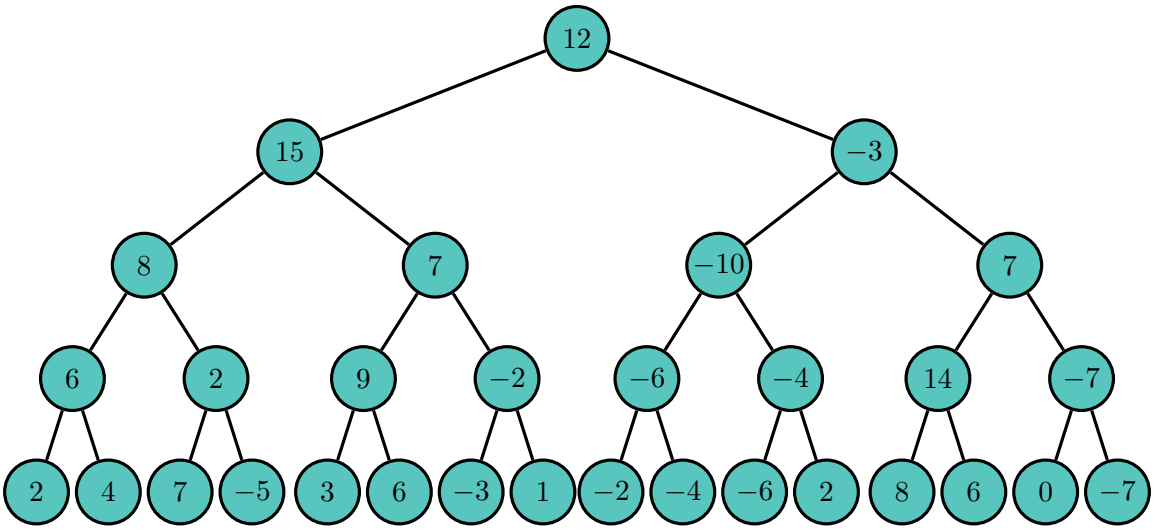
It is often the case that $O(\sqrt{n})$ bounds can be improved to $O(\log n)$ using more complex data structures like segment trees and more complex ideas like 2^n jump pointers, both of which are covered in this chapter. These are, however, more complicated to implement and as such are often comparable 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.

6.3 Segment Tree

For our range sum query problem, it turns out that we can do as well as $O(\log n)$ with a *segment tree*, also known as a *range tree* or *augmented 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.

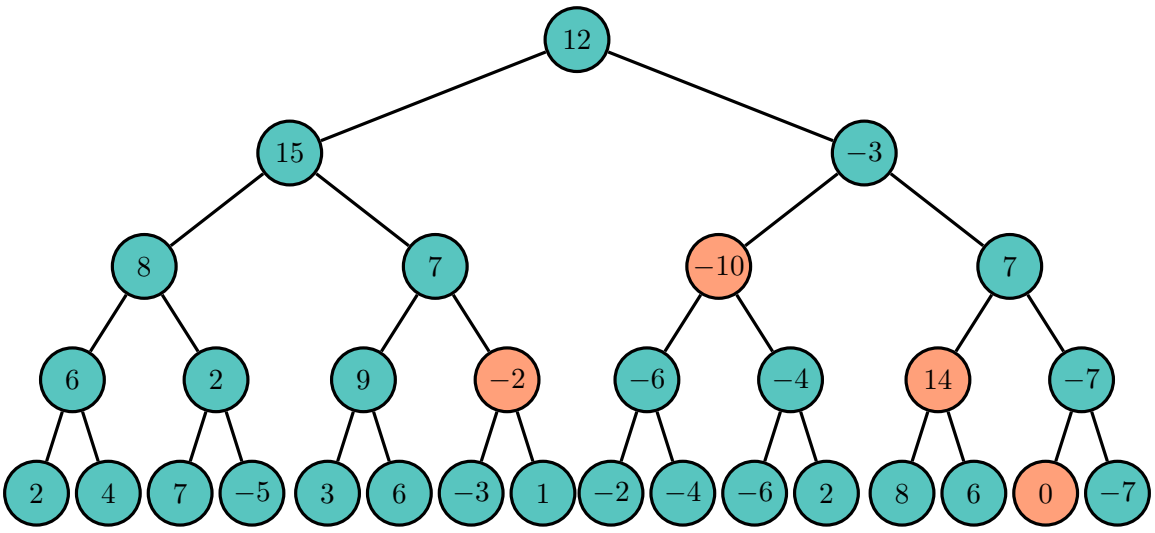
As the name "tree" suggests, we draw inspiration from a binary structure. Let's build a tree on top of the array, where each node keeps track of the sum of the numbers associated with its children. Every node keeps track of the *range* it is associated with and the *value* of the sum of all elements on that range. For example, the root of the tree is responsible for the sum of all elements in the range $[1, n]$, its left child is responsible for the range $\left[1, \left\lfloor \frac{1+n}{2} \right\rfloor\right]$ and its right child is responsible for $\left[\left\lfloor \frac{1+n}{2} \right\rfloor + 1, n\right]$.

In general, for the vertex responsible for the range $[l, r]$, its left child holds the sum for $\left[l, \left\lfloor \frac{l+r}{2} \right\rfloor\right]$ and its right child $\left[\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r\right]$. As we go down to the tree, eventually we'll have nodes with ranges $[l, l]$ that represent a single element in the original list. These, of course, will not have any children.



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

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 $[7, 15]$.

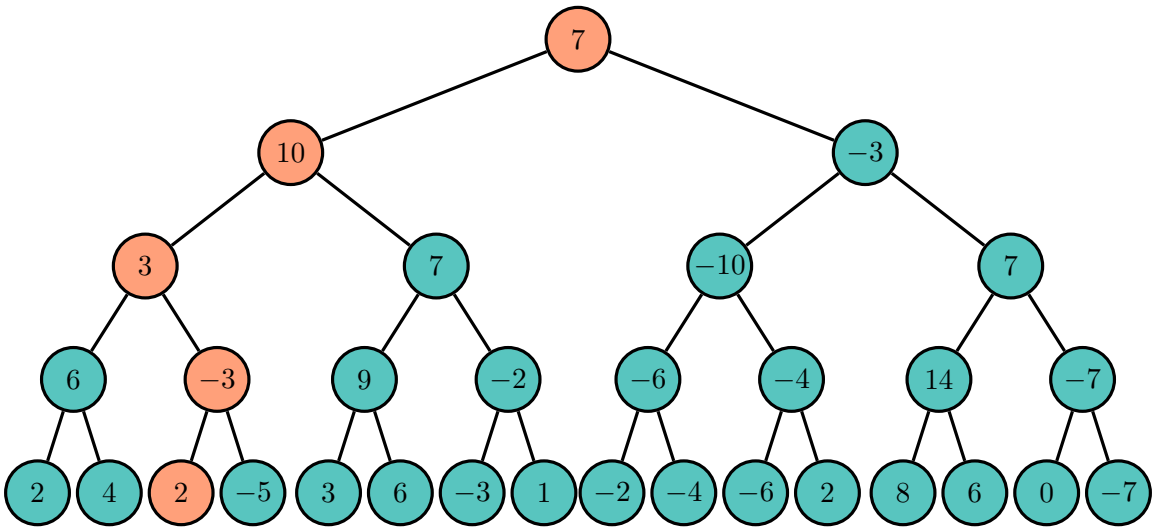


-2 represents the sum $x_7 + x_8$, -10 the sum $x_9 + x_{10} + x_{11} + x_{12}$, 14 the sum $x_{13} + x_{14}$, and 0 represents the single element x_{15}. It seems we always want to take the *largest* segments that stay within the range $[7, 15]$. But how do we know exactly which segments these are?

We handle queries using a recursive call, starting at the root of the tree. We then proceed as follows: If the current node's interval is completely disjoint from the queried interval, we return 0. If the current node's interval is completely contained within the queried interval, we return the sum associated with that node. Otherwise, we pass the query on to the node's two children. Note that this process is $O(\log n)$ because each level in the tree can have at most two highlighted nodes.

function QUERY(range $[l, r]$, range $[a, b]$) ▷ at node $[l, r]$, want $\sum_{i=a}^{b} x_i$
  if $r < a$ or $b < l$ **then** ▷ $[l, r] \cap [a, b] = \emptyset$
    return 0
  if $a \le l$ and $r \le b$ **then** ▷ $[l, r] \subseteq [a, b]$
    return $sum(l, r)$
  return QUERY($\left[l, \left\lfloor \frac{l+r}{2} \right\rfloor\right], [a, b]$) + QUERY($\left[\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r\right], [a, b]$)

Segment trees also handle modifications well. If we want to change the third element to 2, then we have to update the highlighted nodes in the following diagram. We can implement this the same way we implement queries. Starting from the root, we update each modified node's children before recomputing the value stored at that node. The complexity is $O(\log n)$; we change the value of one node in each level of the tree.



function UPDATE(range $[l, r]$, p, k) ▷ $x_p \leftarrow x_p + k$
  if $r < p$ or $p < l$ **then** ▷ $p \notin [l, r]$
    return
  if $l = r$ **then** ▷ leaf node
    $sum(l, r) \leftarrow k$
    return
  UPDATE($\left[l, \left\lfloor \frac{l+r}{2} \right\rfloor\right]$, p, k)
  UPDATE($\left[\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r\right]$, p, k)
  $sum(l, r) \leftarrow sum(l, \left\lfloor \frac{l+r}{2} \right\rfloor) + sum(\left\lfloor \frac{l+r}{2} \right\rfloor + 1, r)$

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. However, if we want a tree of size n, we can always round up to the nearest power of 2, which is at most $2n$. A perfectly balanced segment

tree of size n equalling a power of 2 requires $2n-1$ nodes representing our segments, so $4n$ bounds the memory we'll need.

Now all we need is a way for us to easily store the sum values associated with each node. We can clearly use a structure with two child pointers, but we can also exploit the fact that the segment tree structure is a balanced binary search tree. We can store the tree like we store a heap. The root is labeled 1, and for each node with label i, the left child is labeled $2i$ and the right child $2i+1$. Here's some sample code.

```
final int SZ = 1 << 17; // some sufficiently large power of 2
int[] sum = new int[2 * SZ]; // sum[1] contains sum for [1, SZ], and so on
int query(int i, int l, int r, int a, int b) {
    // i.e. query(1, 1, SZ, 7, 15)
    if(r < a || b < l) return 0;
    if(a <= l && r <= b) return tree[i];
    int m = (l + r) / 2;
    int ql = query(2 * i, l, m, a, b);
    int qr = query(2 * i + 1, m + 1, r, a, b);
    return ql + qr;
}
void update(int i, int l, int r, int p, int k) {
    // i.e. update(1, 1, SZ, 3, 2)
    if(r < p || p < l) return;
    if(l == r){
        sum[i] = k;
        return;
    }
    int m = (l + r) / 2;
    update(2 * i, l, m, p, k);
    update(2 * i + 1, m + 1, r, p, k);
    sum[i] = sum[2 * i] + sum[2 * i + 1];
}
```

Mathematically, what allows the segment tree to work on the addition operation lies in the fact that addition is an associative operation. This means that we can support a wide variety of other kinds of range queries, so long as the operation is associative. For example, we can also support range minimum queries and *gcd* and *lcm* queries. We can even combine different types of information stored at a node. One situation that requires this is maintaining the maximum prefix sum.

For our simple range sum query problem, we don't need the nice, completely balanced structure present when the number of elements is a nice power of two. However, it is necessary if we want to force the array *sum*[] to have the same nice properties as an actual heap so we can perform nice iterative operations on our tree, as previously, all tree operations were recursive. It is also necessary if we need the numbers representing the indices in our tree to have special properties, as in the Fenwick tree.

6.3.1 Lazy Propagation

It is often the case that in addition to performing range queries, we need to perform *range updates*. (Before, we only had to implement point updates.) One extension of our sum problem would require the following two functions:

- *update*(a, b, k) – increment the value of x_i by k for all $i \in [a, b]$
- *query*(a, b) – return $\sum_{i=a}^{b} x_i$.

Some Motivation: $\sqrt{n}$ Blocking

Let's go back to our $\sqrt{n}$ blocking solution and see what changes we can make, and hopefully we can extend this idea back to our segment tree. If we're looking for an $O(\sqrt{n})$ implementation for *update*, we clearly can't perform point updates for all values in the range. The way we sped up *query* was by keeping track of an extra set of data, the sum of all the elements in a bucket, which we used when *the entire bucket was in the query range*.

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

Can we do something similar for *update*? The case we need to wory about is when an entire bucket is included in the update range. Again, we don't want to touch the original array a at all, since that makes the operation linear. Instead, whenever we update an entire bucket, we track the information about the update separately. Thus we store a value for each bucket indicating the amount by which we've incremented that entire bucket.

With this in mind, highlighted are the elements we'll need for *update*$(4, 14, 3)$.

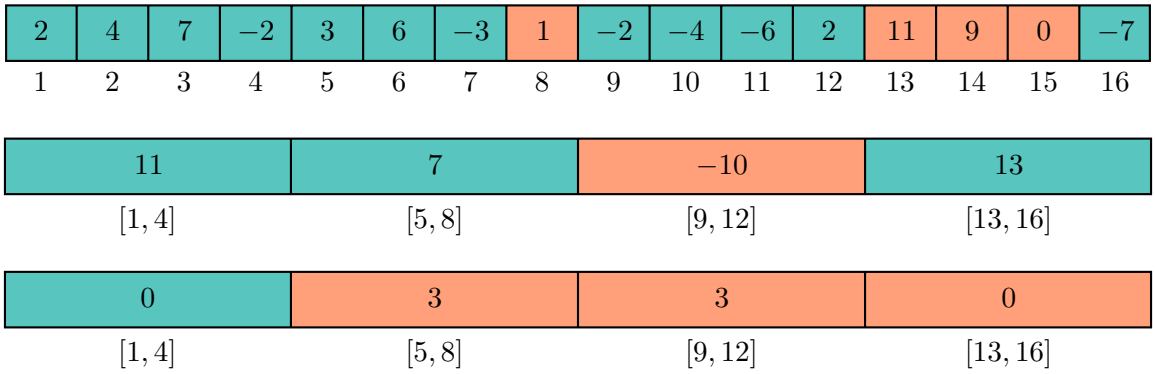
2	4	7	−2	3	6	−3	1	−2	−4	−6	2	11	9	0	−7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

11	7	−10	13
$[1, 4]$	$[5, 8]$	$[9, 12]$	$[13, 16]$

0	3	3	0
$[1, 4]$	$[5, 8]$	$[9, 12]$	$[13, 16]$

To reiterate, we are not storing the actual values of the elements or buckets in the arrays, where they were stored when we solved the original formulation of the problem. Despite this fact, we can still calculate the value of any element or bucket. a_i is equal to the sum of the $\left\lceil \frac{i}{\sqrt{n}} \right\rceil$th value stored in the third array and the ith value stored in the first array. The sum of any given bucket can be calculated similarly. However, we must remember to adjust for bucket size. In the example, there are four elements per bucket, so we have to add $4 \cdot 3 = 12$ to get the correct sum for an updated bucket. Because of all this, we can query a range exactly like we did without range updates.

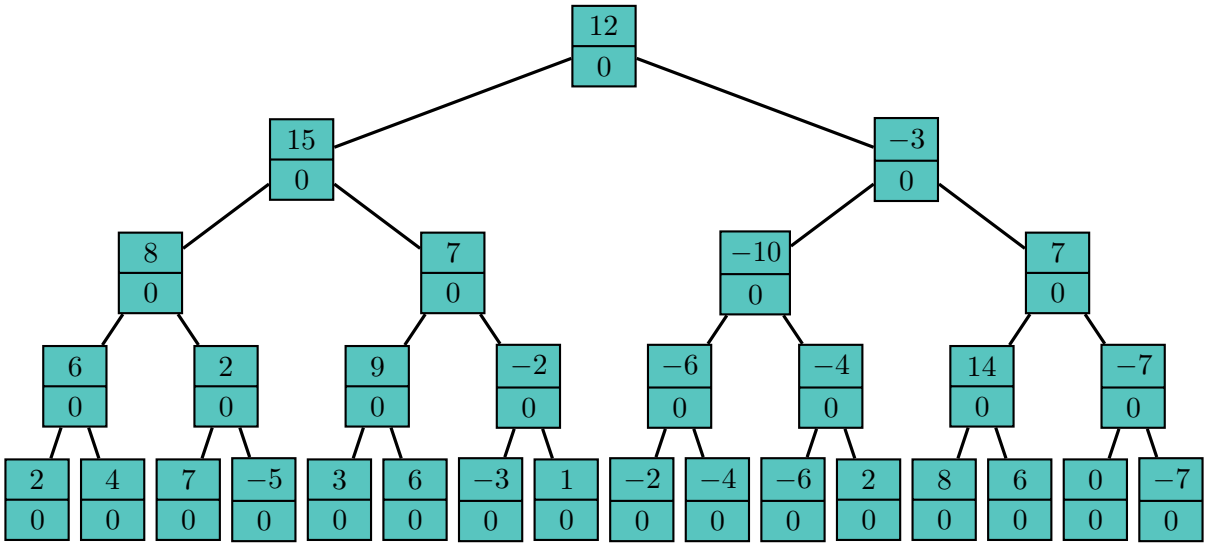
Highlighted are the values necessary for *query*$(7, 15)$.



Thus we have achieved an $O(\sqrt{n})$ solution for both range updates and and range queries.

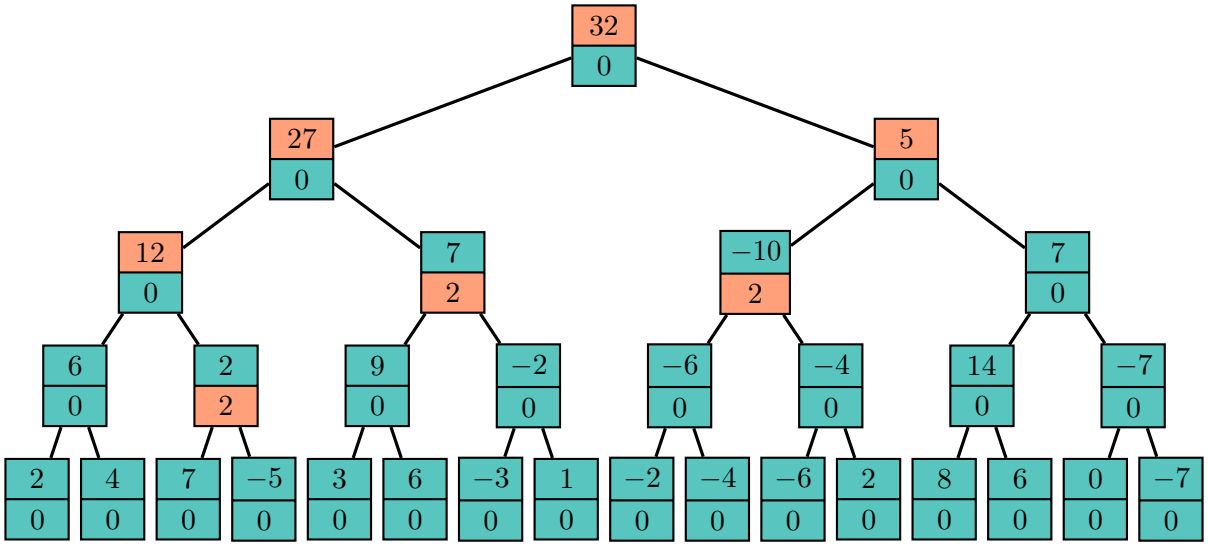
Lazy Propagation on a Segment Tree

Motivated by how we fixed our bucketing solution, let's try adding a similar extra piece of information to our segment tree to try to get an $O(\log n)$ solution. Call this extra value the "lazy" value.



Once again, if the entire range associated with a node is contained within the update interval, we'll just make a note of it on that particular node and not update that node or any of its children. We'll call such a node "lazy."

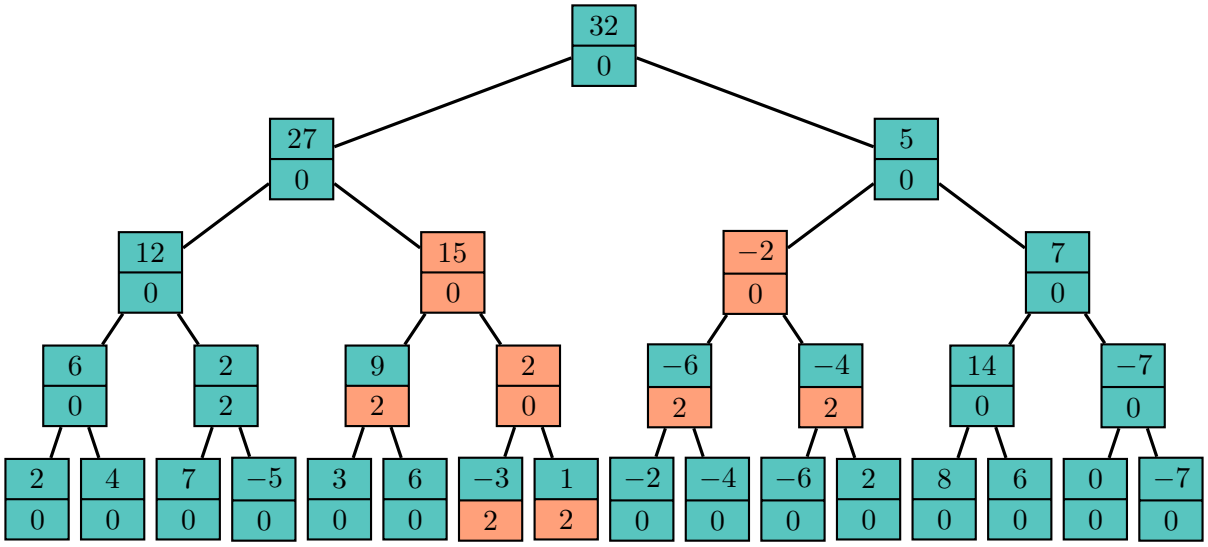
Here's the status of the tree after $update(3, 12, 2)$.



When a node is lazy, it indicates that the sum numbers of every node in its subtree are no longer accurate. In particular, if a node is lazy, the sum number it holds is not equal to the sum of the values in its leaves. This means that whenever we need to access any node in the subtree of that node, we'll need to update some values.

Suppose we encounter a lazy node while traversing down the tree. In order to get accurate sum values in that node's subtree, we need to apply the changes indicated by its lazy value. Thus we update the node's sum value, incrementing by the lazy value once for each leaf in the node's subtree. In addition, we have to propagate the lazy value to the node's children. We do this by incrementing each child's lazy value by our current node's lazy value. And finally, we need to set the lazy value to 0, to indicate that there's nothing left to update. When we implement a segment tree, all of this is usually encapsulated within a "push" function.

Let's illustrate querying with an example: $query(7, 13)$. To answer this, we need to access the nodes for the ranges $[7, 8]$, $[9, 12]$, and $[13, 13]$. The node for $[13, 13]$ is up-to-date and stores the correct sum. However, the other two nodes do not. (The node for $[7, 8]$ is in the subtree of the node for $[5, 8]$, which is lazy.) Thus as we recurse, we push our lazy values whenever we encounter them. Highlighted are the nodes we'll need to update for the query. Notice how $[5, 8]$ and $[9, 12]$ simply pass their lazy numbers to their children, where they'll update themselves when necessary.



The complexity of querying remains the same, even when propagating lazy values. We have to push at most once for each node we encounter, so our runtime is multiplied only by a constant factor. Thus, like normal segment tree queries, lazy segment tree queries also run in $O(\log n)$.

While we're thinking about complexity, we can also ask ourselves why it *doesn't* take $O(n)$ time to update $O(n)$ nodes. With lazy propagation, we take advantage of two things. The first is that on each query, we access very few nodes, so as long as the nodes we access are up-to-date, we're all set. The second is that we can combine updates while they're still being propagated, that the update operation is associative. This allows us to update only when it's absolutely necessary—the rest of the time, we can be lazy. (The next time someone tells you you're being lazy, you can say it's a good thing.)

Like normal segment trees, lazily propagating segment trees can handle a diverse set of range updates and range queries. We can support an update, where instead of *incrementing* each element on a range *by* a certain value, we *set* each element *to* that value. There is no difference between the two as point updates, but they are very different operations when applied as range updates. Sometimes, we can even have more than one range update on a single tree. When implementing this, however, it is important to be careful when pushing lazy values—composing different operations can become quite complicated.

Below is my implementation of a segment tree supporting range sum and range increment. Note that it includes one detail that we left out in our development of lazy propagation above: how we update a node whose range partially intersects the updated range. One way to do this is to calculate the length of the intersection and update directly. However, this does not work well for queries that are not as nice as incrementing, such as setting an entire range to a value. Instead, we first update the children of this node. Then we push the lazy values off the children so their sum values are accurate. This allows us to recalculate the sum value of the parent like we do for a non-lazy segtree. I have this as my `pull` function below.

```
final int SZ = 1 << 17;
int[] sum = new int[2 * SZ];
int[] lazy = new int[2 * SZ];

void push(int i, int l, int r){
  if(lazy[i] != 0){
    sum[i] += (r - l + 1) * lazy[i];
    if(l < r){
      lazy[2 * i] += lazy[i];
      lazy[2 * i + 1] += lazy[i];
    }
    lazy[i] = 0;
  }
}

void pull(int i, int l, int r){
  int m = (l + r) / 2;
  push(2 * i, l, m);
  push(2 * i + 1, m + 1, r);
  sum[i] = sum[2 * i] + sum[2 * i + 1];
}

int query(int i, int l, int r, int a, int b) {
  push(i, l, r);
  if(r < a || b < l) return 0;
  if(a <= l && r <= b){
    return sum[i];
  }
  int m = (l + r) / 2;
  int ql = query(2 * i, l, m, a, b);
  int qr = query(2 * i + 1, m + 1, r, a, b);
  return ql + qr;
}

void update(int i, int l, int r, int a, int b, int k) {
  // push(i, l, r); // Necessary for non-commutative range updates.
  if(r < a || b < l) return;
  if(a <= l && r <= b){
    lazy[i] += k;
    return;
  }
  int m = (l + r) / 2;
  update(2 * i, l, m, a, b, k);
  update(2 * i + 1, m + 1, r, a, b, k);
  pull(i, l, r);
}
```

6.3.2 Fenwick Tree

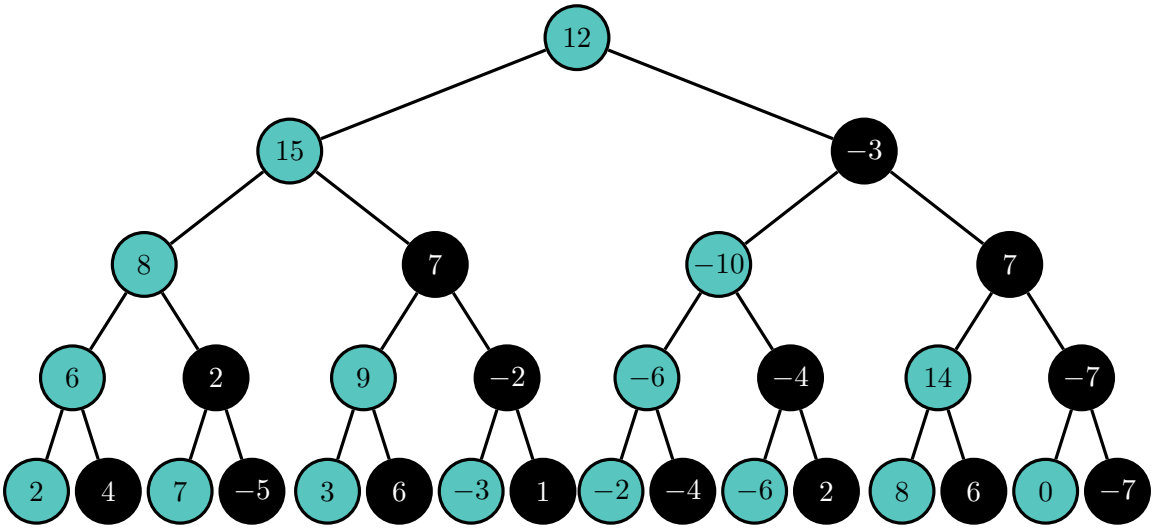
A *Fenwick tree*, or *binary indexed tree (BIT)*, is simply a faster and easier to code segment tree when the operator in question, in addition to being associative, has an inverse. Unfortunately, it's not at all intuitive, so bear with me at first and let the magic of the Fenwick tree reveal itself later.[1]

[1] In fact, it is so magical that Richard Peng hates it because it is too 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 ignore 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

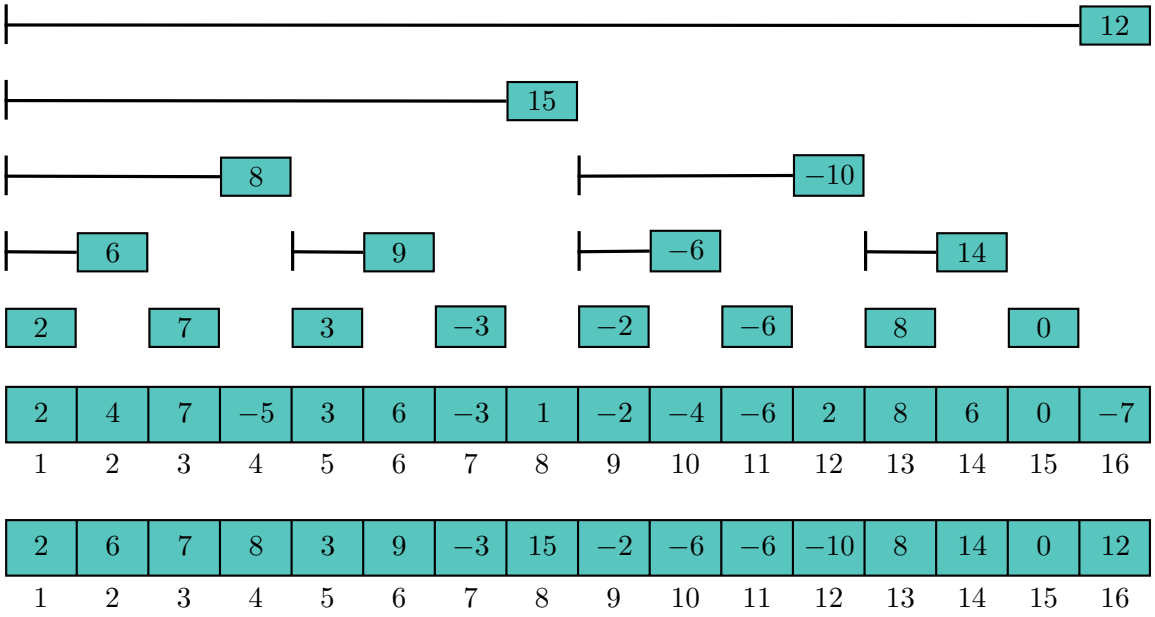
2	6	7	8	3	9	−3	15	−2	−6	−6	−10	8	14	0	12
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

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

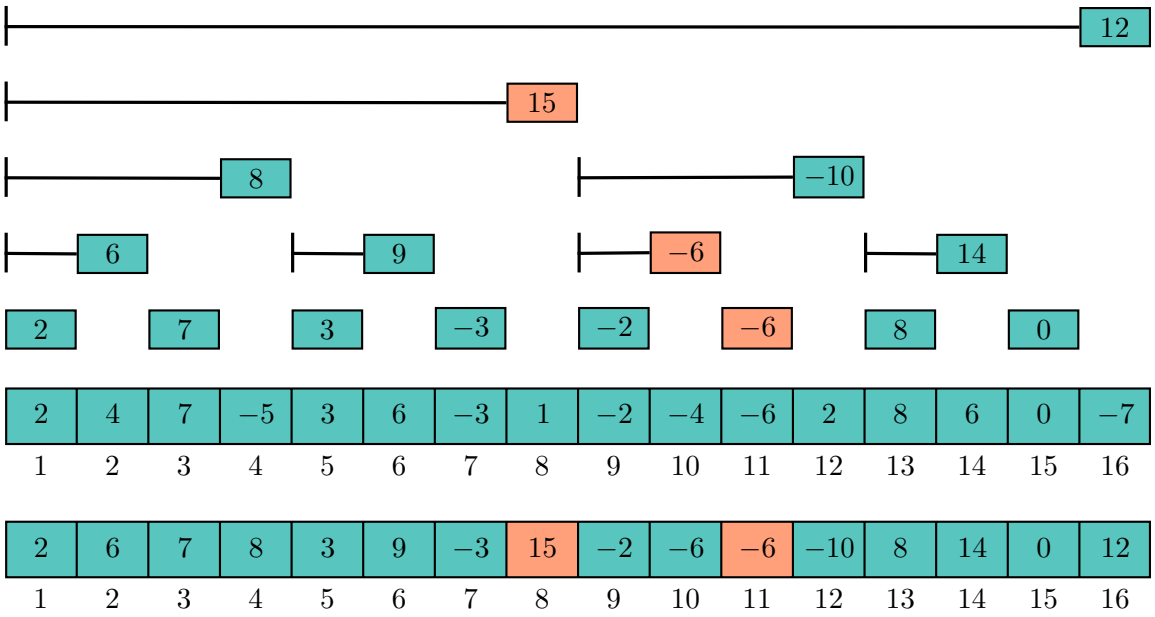
$$y_i = \sum_{k=i-2^{v_2(i)}+1}^{i} x_k,$$

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.



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 $\sum_{k=1}^{11} x_k$. Let's take a look at the diagram to see which elements we need.

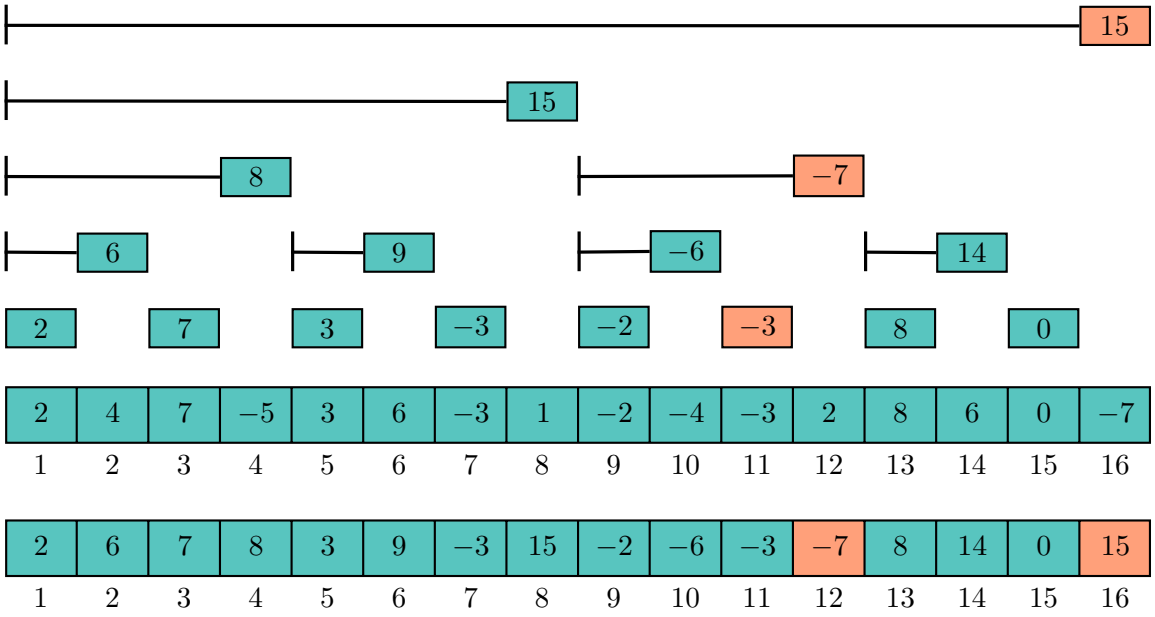


We see that the sum $\sum_{k=1}^{11} x_k = y_8 + y_{10} + y_{11}$. If we look at 11 in binary, we have $11 = 01011_2$. Let's see if we can find a pattern in these numbers in binary:

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

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 x_{11} from -6 to -3. Which numbers will we have to change?



We needed to increment the highlighted values, y_{11}, y_{12}, and y_{16}, by 3. Once again we'll look at 11, 12, and 16 in base 2.

$$\begin{aligned} 11 &= 01011_2, \\ 12 &= 01100_2 = 11 + 2^{v_2(11)}, \\ 16 &= 10000_2 = 12 + 2^{v_2(12)}. \end{aligned}$$

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 quickly it can be coded. The only tricky part is finding exactly what $2^{v_2(i)}$ is. 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. Note that here, values in the array remain 1-indexed, which is different from how we code segment trees.

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

6.4 Queue with Minimum Query

Suppose we wanted a list data structure with the following three operations:

- $add(x)$ – add x to the end of the list.
- $remove()$ – remove the first element in the list.
- $min()$ – return the minimum element in the list.

This is different from a heap since *remove* does not remove the minimum element. It's pretty easy to find a $O(\log n)$ solution using the data structures we already know. However, it is possible to build a data structure that can do any of these operations with complexity $O(1)$.

To solve this problem, we'll first solve an easier problem. Suppose instead of removing the first element of the list, we had remove the last element; in other words, we needed to build a stack with minimum query instead of a queue. This is a simple task; we'll just use a normal stack, but instead of storing single numbers, we'll store pairs. The pairs will each contain the number we're adding and the minimum element up to that point in the stack.

To build a queue given a stack with minimum query, we'll just have two stacks. When we add an element, we push it to the top of the first stack. When we remove an element, we take it off the top of the second stack. The minimum element in the queue is the smaller element between the minima of either stack.

This seems like it obviously doesn't work – one of the stacks keeps growing, while the other can only shrink. This is not a problem, however; when the second stack runs out of elements, we'll just pop off every element of the first stack and push each onto the second stack. This amounts to one $O(n)$ operation for every n $O(1)$ operations, which averages out to constant time.

6.5 Balanced Binary Search Tree

Recall that a normal binary search tree is not guaranteed to be balanced. Many data structures have been invented to self-balance the binary search tree.

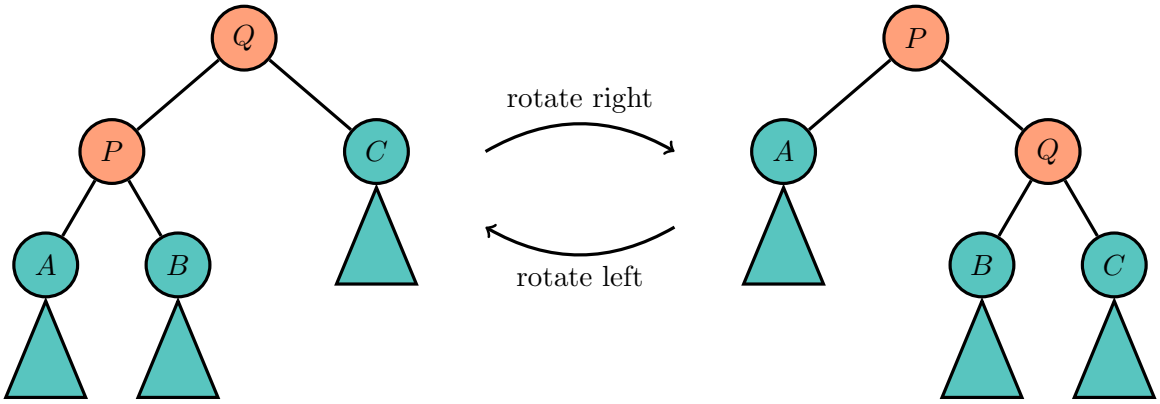
The *red-black tree* was an early development that represents the "just do it" approach, using casework to balance a tree. This results in a simple mathematical proof for the maximum depth of the tree. Unfortunately, coding these up are quite nasty do to the casework involved, so doing so is generally not preferred unless your name is Sreenath Are.

The *splay tree* is a data structure that guarantees amortized logarithmic performance; this means that any single operation is not necessarily linear, but over *any* sequence of length m for $m \geq n$, the performance for all m operations is $O(m \log n)$. Though the proof of the splay tree's complexity is necessarily more complicated than the red-black tree's complexity, as it requires amortized analysis, coding the splay tree is much easier than coding the red-black tree. In addition, the splay tree grants us more flexibility than the red-black tree provides, allowing us to build more complicated structures like link-cut trees using splay trees.

The *treap* is a probabilistic data structure that combines a binary search tree with a heap to create a BBST. Just as quicksort is not guaranteed to run in $O(n \log n)$, treaps are not guaranteed to have logarithmic depth. In practice, however, they almost always have performance that is $O(\log n)$. Treaps are especially useful in programming contests because they are the easiest BBST to implement.

6.5.1 Tree Rotation

The key idea behind all of these data structures is the use of the *tree rotation*, which is used to change the structure of the tree but will not change the order of the elements (inorder). Maintaining the order of the elements is important, of course, if we wish our tree to remain a binary search tree.

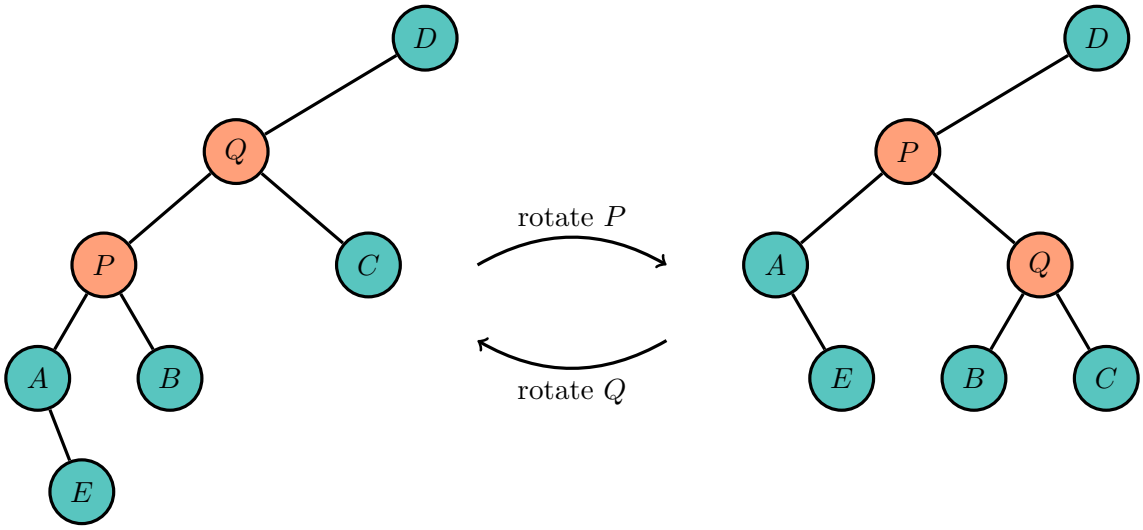


Here, the triangles represent subtrees, as A, B, and C could very well have children of our own, but they are not shown in the diagram. Note that the inorder ordering of the nodes has not been changed.

When we rotate right, we literally take the edge connecting P and Q and rotate it clockwise. Then P becomes the parent of Q where before P was the child of Q. However, this definition of

rotation is somewhat cumbersome, as we have different terms for the highly symmetrical rotating right and rotating left. The key characteristic a rotation is we move the lower node up one level. Thus, I prefer to think of tree rotation as whatever tree rotation, either left or right, will *rotate up* a node. In the diagram, rotating right is analogous to rotating P up, and rotating left is analogous to rotating Q up. Rotating a node up will change the tree such that its former parent is now its child.

The other notable change in the tree structure is the subtree associated with B passes between P and Q upon tree rotation. Finally, tree rotation can happen at any place in the tree, not just at the root. When we rotate P up, we must update the parent of Q to change its child to P.



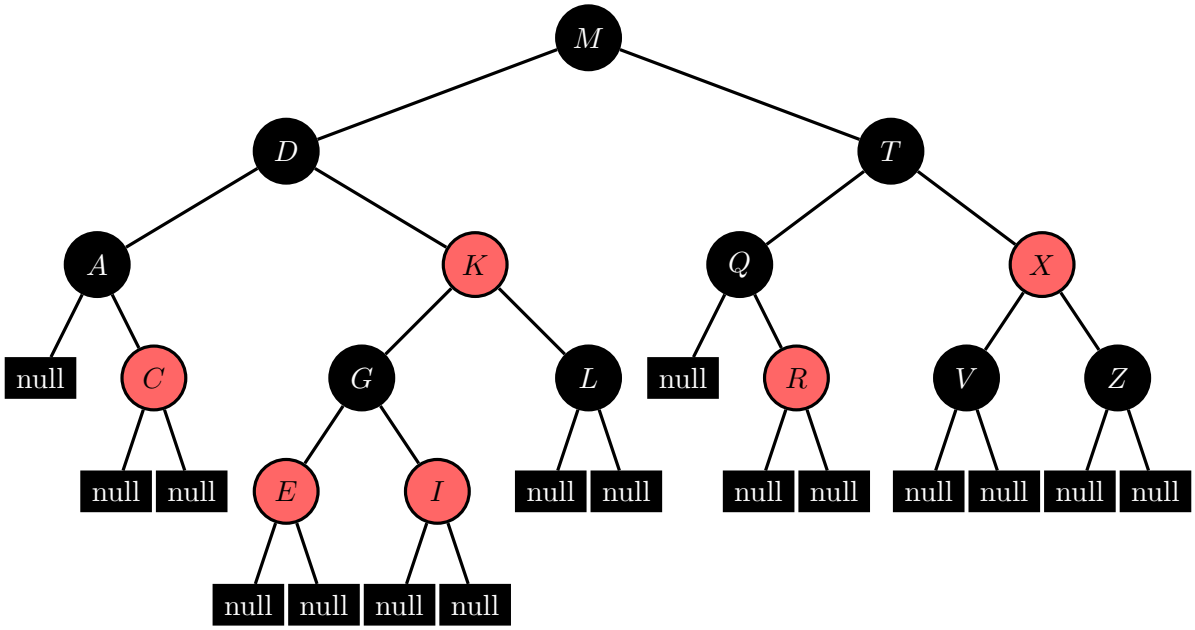
Note that in this example, rotating up P decreases the total height of the tree. We want to somehow systematically rotate up nodes to accomplish this. The following data structures provide such a system.

6.5.2 Red-Black Tree

As stated earlier, the red-black tree represents the "just do it" approach. We want to somehow bound the maximum length from the root to any leaf node by applying a constraint on the tree. In this case, our constraint is a coloring of the nodes (red or black) such certain properties are satisfied. For the red-black tree, the only nodes storing data are non-leaf nodes, and all leaf nodes store "null." The number of null nodes, then, is $n + 1$, where n is the number of non-null nodes. We do this so that any node storing useful information has two children.

We want our tree to satisfy the following properties:

1. The root is black.
2. All leaf nodes (null) are black.
3. Every red node has two black children. Consequently, a red node has a black parent.
4. Any path from the root to a null node contains the same number of black elements.



Note that every path from the root to a null node contains four black nodes.

The proof of $O(\log n)$ search follows immediately. The shortest possible path contains only black nodes, and the longest possible path contains black and red nodes alternating. Since the number of black nodes in both must be the same, any path is at most twice as long as any other path. As the number of nodes in the tree is $2n + 1$, the number of black nodes m in any path is then bounded below by $2^{2m} - 1 \geq 2n + 1$ and above by $2^m - 1 \leq 2n + 1$. Thus the height of the tree is on the order $O(\log n)$, and we are done.

Thus if our tree maintains its red-black coloring and satisfies the necessary properties, we can guarantee that our tree is balanced. We then consider the two ways we change the state of the tree, insertion and deletion. We can insert and delete in the normal way, but we might need to make changes to the tree after that to restore the red-black properties. We do this through a small number of color flips and tree rotations, which we can handle through casework.

Let's handle insertion first. When we insert a node, it takes the place of a black null leaf node. To maintain property 4, we must color the new node red, as it has two black null children. However, we may have violated some of the other properties, specifically 1 and 3.

We'll call the new node N, its parent P, its uncle (parent's sibling) U, and its grandparent G, if they exist.

We consider the following five cases.

1. N is the root node. That is, N is the first node added to the tree.

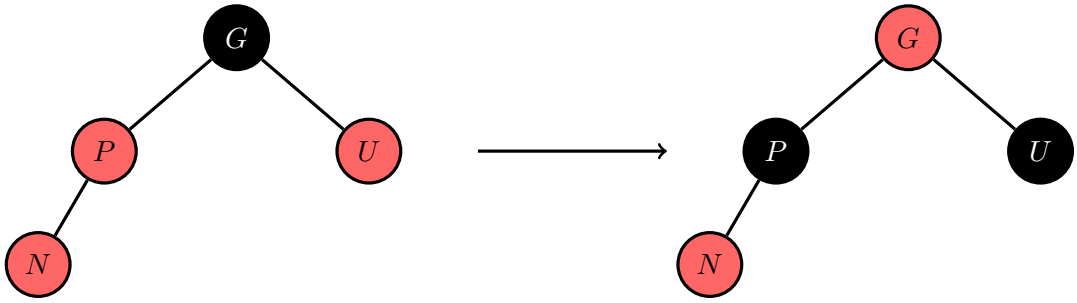
   It is easy to just change the color of N to red to restore property 1, and we are done.

2. P is black.

   Then property 3 is not violated, and we are done.

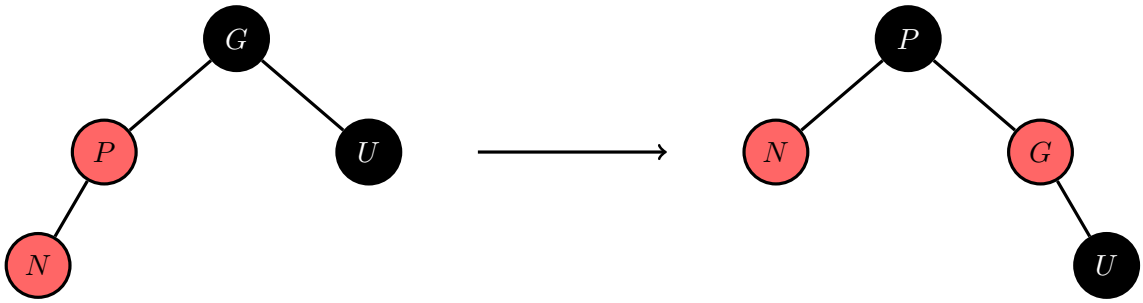
3. P is red, and U is red (G and U exist since P cannot be the root, as the root is black).

   As P and U are red, G is black. We simply color P and U black and G red to restore property 3.



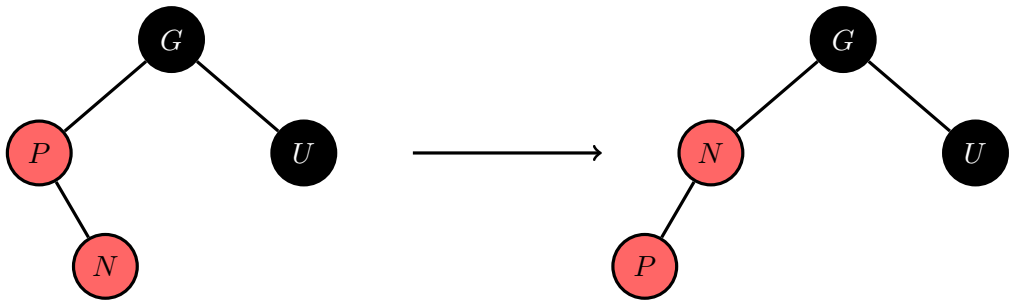
4. P is red, U is black, and N is on the same side of P as P is of G.

   G must be black. We rotate up P and color P black and G red.



5. P is red, U is black, and N is on the opposite side of P as P is of G.

   G must be black. We rotate up N, and this reduces to case 4.



Thus after we insert, we can always restructure the tree using a constant number of operations to restore the necessary red-black tree properties.

Now we need to work on deletion. Recall how deletion works on a normal binary search tree. If the node we need to replace has two children, we swap the node with the least element in its right subtree, which does not have two children. We then are able to remove that node more easily.

We can do the same thing with the red-black tree. If a node has two non-null children, we swap its value with the least non-null node in its right subtree and remove that node. Thus we reduce the deletion problem to the case where the node we need to remove has at least one null child.

Two cases are very easy.

1. The node we wish to remove is red.

   Then the node must have two null leaf nodes as children. Then we remove the node by replacing it and its children with a single null node.

2. The node is black and has a red child.

   We replace the node with its child and paint its child red.

The remaining case is the node black with two black null children. We'll first replace the node with a null node N. Then, all paths passing through N are one black node short compared to all other paths.

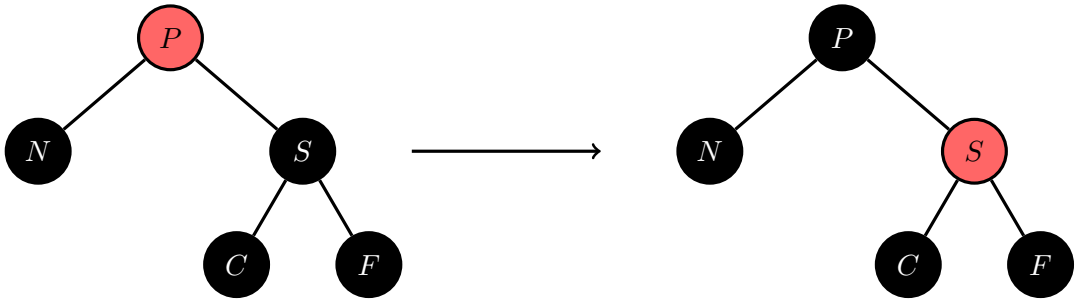
We denote the parent of N as P, its sibling S, and its sibling's children C and F, such that C is on the same side of S as N is of P, if they exist. That is, C is the "closer nephew" child, while F is the farther. We now describe a six-case balancing procedure on the black node N that fixes our problem.

1. N is the root.

   We are done, as every path possible must pass through N, so all paths are balanced.

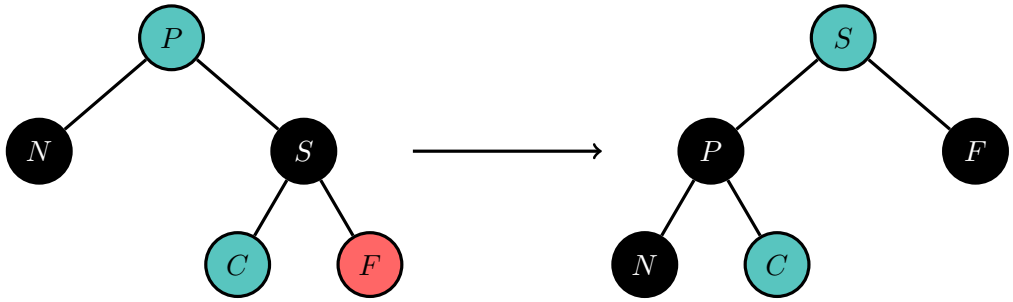
2. P is red and S, C, and F are black.

   Then we simply trade the colors of P and S. The number of black nodes for paths passing through S stays the same, but the number of black nodes for paths passing through N increases, so we are done.



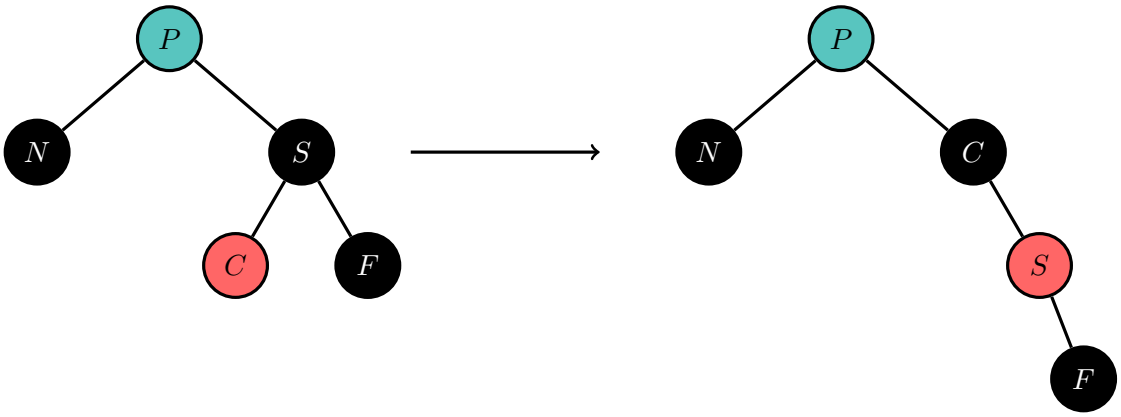
3. S is black and F is red.

   Then we color F black and rotate up S, before swapping the colors of P and S. Then paths passing through N gain a black vertex, while paths passing through C and F stay the same.



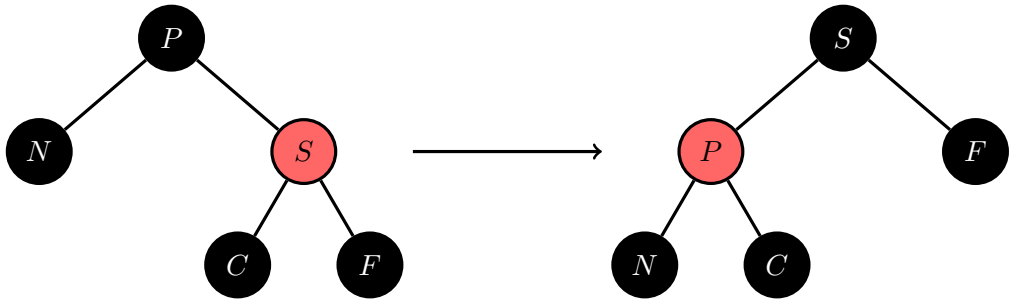
4. S is black, F is black, and C is red.

   Then we rotate up C and swap the colors of S and C. Then this reduces to case 3.



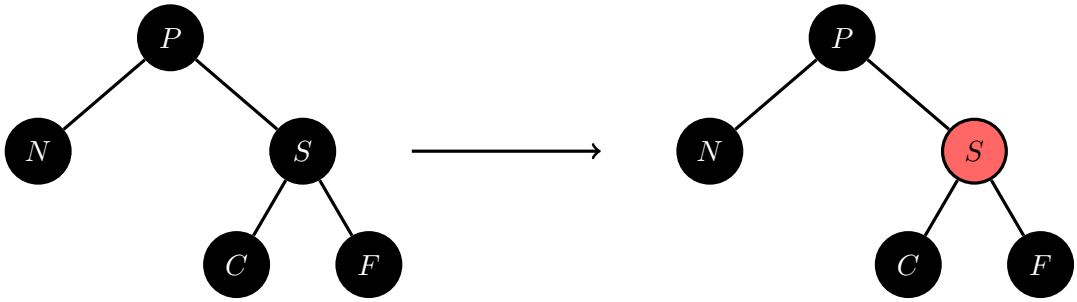
5. S is red.

   Then P, C, and F must be black. We rotate S up and swap the colors of P and S. C, the new sibling of N is black, so this then reduces to one of cases 2, 3, or 4.



6. P, S, C, and F are all black.

   Then we recolor S red. Now, all paths through P have the same number of black nodes, but each path through P has one less black node than each path not passing through P. Since P is black, and this procedure did not require that N be a leaf node, we can repeat this entire balancing procedure on P.



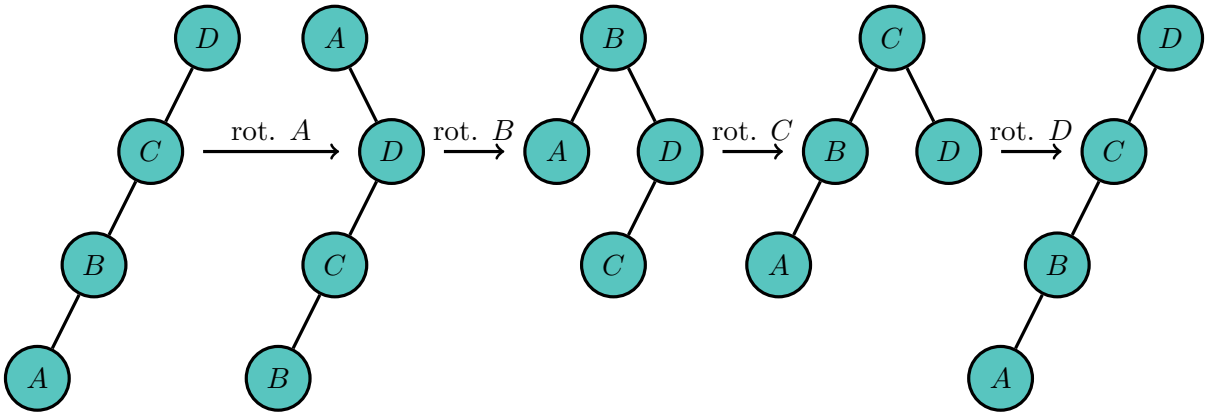
Unlike the balancing for insertions, the balancing for deletions has the potential to call itself on the parent node P. However, balancing following a deletion is still $O(\log n)$ worst case. It is possible prove that it is $O(1)$ amortized. Regardless, we now have a deletion algorithm that runs in $O(\log n)$ time.

Thus the red-black tree supports standard binary search tree operations, all in $O(\log n)$.

6.5.3 Splay Tree

Remember that rotating a node brings it one level closer to the root. The idea behind a splay tree is to apply a sequence of rotations that rotate a node all the way up to the root. The intuition for doing this is once we access a node, it is easy to perform other operations on it since it will simply be the root of the tree.

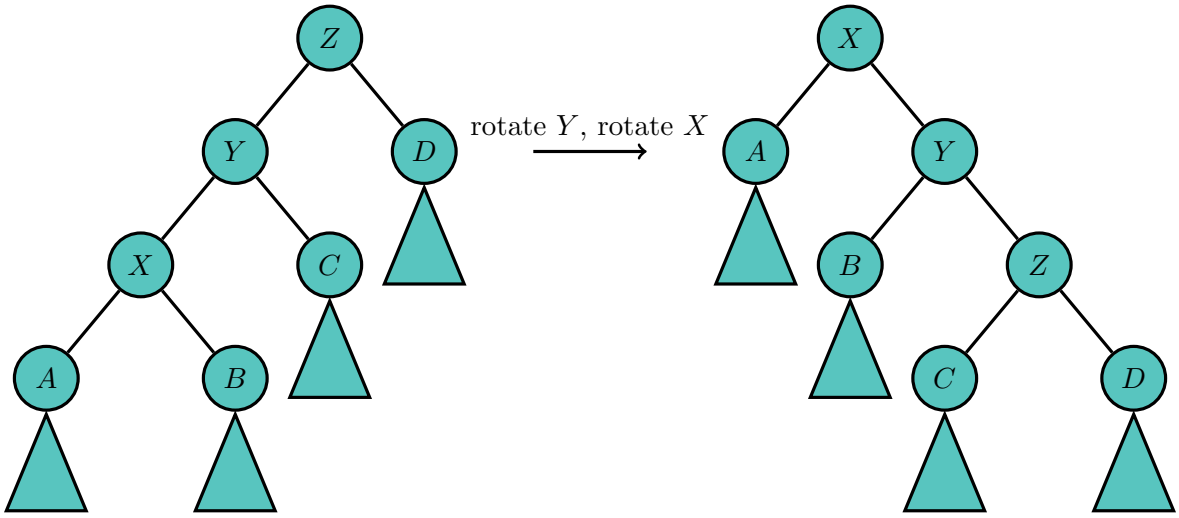
It is not at all clear that rotating nodes to the root will somehow improve the average performance of the tree. That's because rotating the normal way doesn't! To improve the performance of the tree, we expect that our balancing procedure tends to decrease the height of the tree. Unfortunately, simply repeatedly rotating a node up the tree doesn't exactly do that task. Consider a tree that looks liked a linked list, and rotating the nodes in the order A, B, C, D, as shown in the diagram. The height of the tree remains linear in magnitude, as at its smallest stage the tree is around $\frac{n}{2}$ in height, so accessing elements remains $O(n)$ on average.



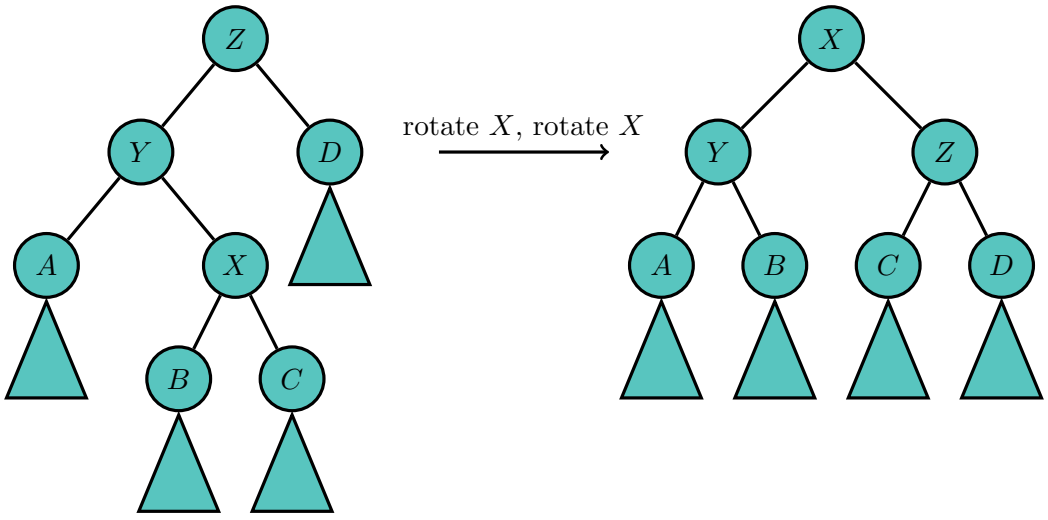
As discussed earlier, simply repeatedly applying the standard rotation clearly is not guaranteed to reduce the height of the tree on average. However, if we simply make one small change, magic happens. The trick of the splay tree is to rotate a node to the root in such a way that the tree has a

tendency to decrease in height. We'll use two compound rotations, and depending on the structure of the tree, we'll use a different one accordingly, to rotate a node to the root.

When a node is a left child and its parent is a left child, or the node is a right child and its parent is a right child, we first rotate up the parent, and then rotate up the node. This sequence of rotations is the only difference between splaying and rotating a node to the root using standard rotation.

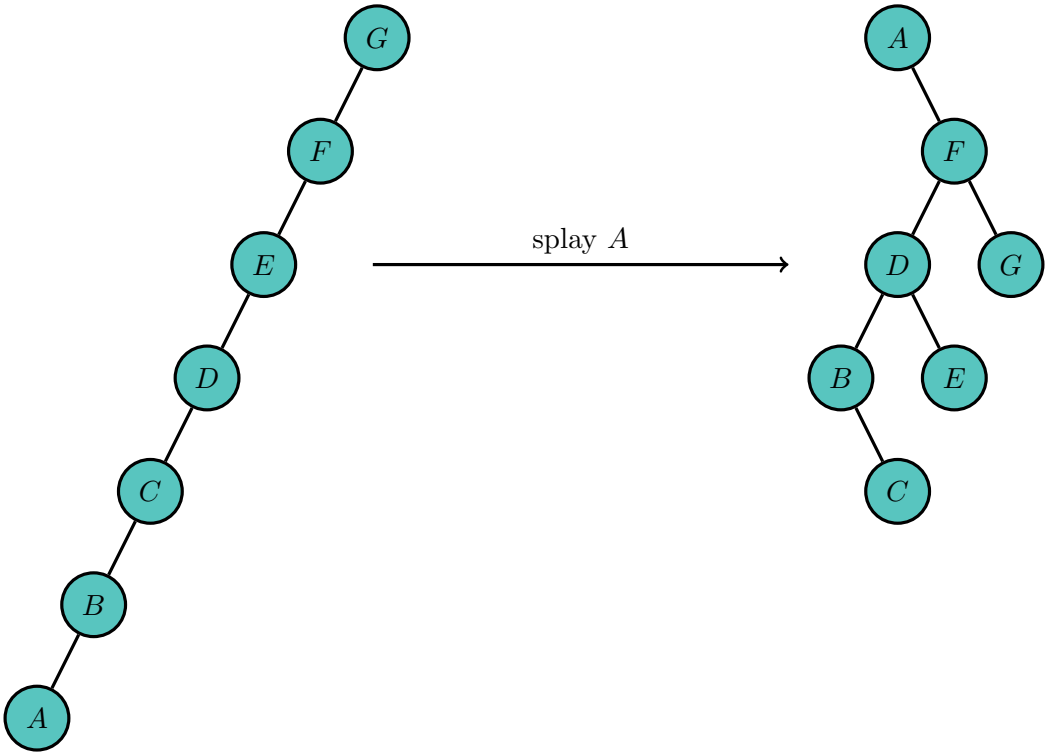


When a node is a left child and its parent is a right child, or the node is a right child and its parent is a left child, we rotate the node up twice, the normal way.



Finally, when a node's parent is the root, so it has no grandparent, we simply rotate up the normal way.

Rotating a node to the root in this way is called *splaying* the node. Thus, splaying a node brings it to the root of the tree. Splaying seems too simple to lead to an amortized $O(\log n)$ solution. To get a better idea of how splaying works, let's see what it does to a tree that looks like a linked list.



Splaying here cuts the height of the tree in half – a huge improvement. Splaying is not guaranteed to decrease the height of the tree, and it is possible for a sequence of splays to even result in a linked-list-like structure. However, given a number m at least equal to the maximum number of nodes n ever in the tree, *any* sequence of m splays runs in $O(m \log n)$. Splaying is then $O(\log n)$, amortized.[2]

Then, whenever we access a node, even in insertion and deletion, we splay that node to the top. This makes access, insert, and delete all $O(\log n)$ amortized.

Since splay trees need not satisfy a restrictive coloring, as red-black trees do, we have the freedom to completely change the structure of the tree at a whim. Recall how tedious the casework was to simply add or remove one node at a time in a red-black tree. Because the splay tree is simply a normal binary search tree that rotates nodes up upon access, we can detach an entire subtree from the tree and not worry about any properties of our tree no longer being satisfied.

For this reason, we define the *split* and *join* functions for splay trees.

Given two trees S and T, such that every element in S is less than every element in T, we join S and T into one tree by splaying the largest element in S, so that the root, which is now the largest element, has no right child. Then, we set its right child to the root of T, resulting in a single tree with elements from both S and T.

Given a tree and a value v, we split the tree in two by splaying the greatest element not greater than v to the root and detaching the right subtree from the rest. This results in two trees, one containing all elements at most v, and the other containing all elements greater than v.

[2]If you're curious to see the proof of this, a rather in depth explanation can be found here: `http://www.cs.cmu.edu/afs/cs/academic/class/15859-f05/www/documents/splay-trees.txt`

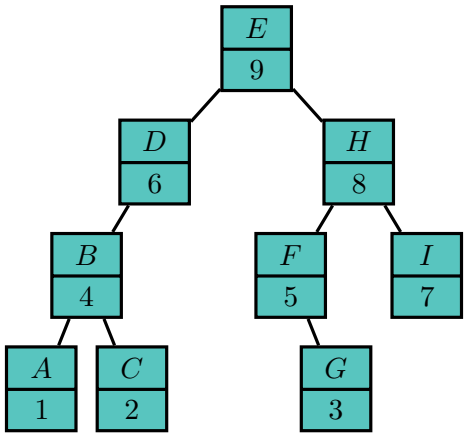
Both of these operations are $O(\log n)$ amortized. It is possible to implement insert and delete using split and join.

Since a binary search tree stores an ordered set, it is incredibly useful for storing a dynamic list of elements where the length can change and elements need to be added or removed from any point in the list, not just the beginning or the end. Splay trees are incredibly useful because of the split and join operations, as they allow us to remove consecutive elements from the list represented by our tree.

For example, to remove elements indexed in the range $[i, j]$, we split at index i and split at index $j+1$ to get three splay trees, one representing $[1, i-1]$, another $[i, j]$, and the last $[j+1, n]$. We can then merge the first and third trees together, to get two trees, one representing $[i, j]$ and the other representing the original list with $[i, j]$ removed. A similar procedure can insert a list within another list as as contiguous block. Both of these operations can be completed in $O(\log n)$ amortized using a splay tree.

6.5.4 Treap

A treap is a binary search tree smushed together with a heap. Each node is assigned, in addition to its key, a random *priority*. The priorities in a treap always satisfy the *heap property*: for any node, its priority is greater than the priority of either of its children. The diagram below illustrates a treap. On each node, the letter is the node's key and the number is the node's priority.



The randomness in each node's priority balances the treap. We can gain some intuition for why this works by looking at how a treap is built. Consider an array containing pairs of key and priority, sorted by key. To construct a treap from these values, we first take the element with the highest priority and set it as the root. We can then recurse on the two halves of the array split by the root, and construct the root's left and right subtrees in a similar way. Because the priority of each node is randomly chosen, we effectively choose a random element as the root each time we recurse.

Like quicksort, we can expect that the root doesn't always split the tree too unevenly and obtain a $O(\log n)$ bound on the expected depth of each node. Moreover, it is possible to show that the entire tree has a depth of $O(\log n)$ with high probability. This analysis means that a treap is as good complexity-wise as any other deterministic BBST for all practical purposes. (To do the first

part more rigorously, we can let $A_{i,j}$ be an indicator random variable representing whether the jth node is an ancestor of the ith node. What is the expected value of $\sum_{j=1}^{n} A_{i,j}$?)

In the remainder of this section, we'll go over the implementation of treaps. Our implementation will be quite different from the rotation-based implementations of splay trees and red-black trees described above. Intead of rotating, we will have two functions, `split` and `merge`, that will be used to define all other treap operations. Here are their definitions:

`split` takes a treap T and a key k and splits T into two treaps by k. Thus `split` outputs two treaps, L and R, where all the elements of L are less than k and all the elements of R are at least k. `split` has a complexity of $O(\log n)$.

`merge` does the opposite of `split`. It takes two treaps L and R, where each key in L is at most the smallest key in R, and turns them into one treap, T. `merge` also has a complexity of $O(\log n)$.

One way to think about these functions is to imagine treaps as a data structure that dynamically maintains sorted arrays. Then splitting is cutting an array into two, and merging is appending one array onto the end of another. With this perspective on `split` and `merge`, standard BST operations are not difficult to write. For example, we can insert a key k into a treap T by splitting T into L and R by k. Then we can create a new node n with key k, and merge L, n and R back into T. Other operations that we can implement are `erase`, `union` and `intersect`.

Writing `split` and `merge` is also relatively easy. With C++, we can use pointers and references to simplify our work.

To `split`, we recurse on T, while maintaing references to L and R. The root of T becomes the root of one of L and R, depending on its key. If the root of T becomes the root of R, we recurse on the left subtree of T and `split` into L any nodes that have key less than k. The other case, where the root of T becomes the root of L, works similarly.

To `merge`, we take the root of L and R with the higher priority, and set it as the root of T. If the root of L has higher priority, we `merge` R and the right subtree of L. Otherwise, we `merge` L and the left subtree of R. Like `split`, we can recurse down the relevant subtrees until we are done.

Here's a short C++ implementation of a treap with `split`, `merge` and `insert`:

```
struct node {
  node *l, *r;
  int k, p;
  node(int k) : l(0), r(0), k(k), p(rand()) {}
};

void split(node *t, node *&l, node *&r, int k){
  // Splits t into l and r.
  if(t == NULL) l = r = NULL;
  else if(k <= t -> k) split(t -> l, l, t -> l, k), r = t;
  else split(t -> r, t -> r, r, k), l = t;
}

void merge(node *&t, node *l, node *r){
  // Merges l and r into t.
  if(l == NULL || r == NULL) t = l ? l : r;
  else if(l -> p > r -> p) merge(l -> r, l -> r, r), t = l;
  else merge(r -> l, l, r -> l), t = r;
}

void insert(node *&t, int k){
  node *n = new node(k), *a;
  split(t, t, a, k);
  merge(t, t, n);
  merge(t, t, a);
}
```

We can also create an *implicitly keyed* treap, which functions as a dynamic array. Instead of storing keys, each node stores the size of its subtree. We can then define the *implicit key* of a node as its position in the in-order traversal of the treap, a value we can easily compute from subtree sizes. Modifying `split` to operate based on the implicit key allows us to rearrange parts of the array and insert new elements anywhere. Further augmentation allows us to perform range queries as well. And if we add lazy propagation, we can also support operations that modify ranges.

Here's an example of an implicitly keyed treap that supports reversal of subarrays. In `node`, `s` maintains the size of its subtree, and `f` is a flag that indicates reversal. We use `push` and `pull` to propagate data down and up the treap, respectively. Note the difference in the implementation of `split`.

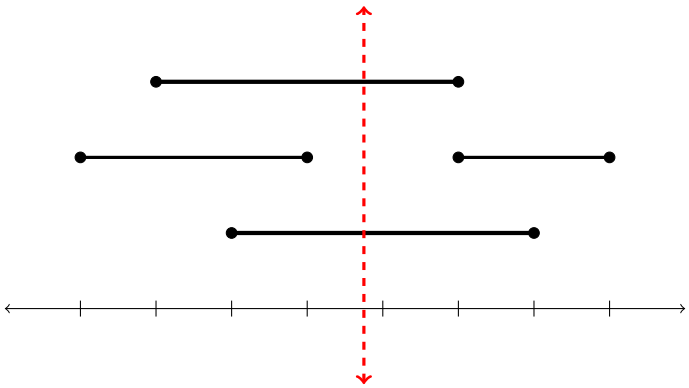
```
struct node {
  node *l, *r;
  int v, p, s, f;
  node(int v) : l(0), r(0), v(v), p(rand()), s(1), f(0) {}
};

int size(node *t){
  return t ? t -> s : 0;
}

void push(node *t){
  if(t == NULL) return;
  if(t -> f){
    swap(t -> l, t -> r);
    if(t -> l) t -> l -> f ^= 1;
    if(t -> r) t -> r -> f ^= 1;
    t -> f = 0;
  }
}

void pull(node *t){
  if(t == NULL) return;
  t -> s = size(t -> l) + size(t -> r) + 1;
}

void split(node *t, node *&l, node *&r, int k){
  push(t);
  if(t == NULL) l = r = NULL;
  else if(k <= size(t -> l)) split(t -> l, l, t -> l, k), r = t;
  else split(t -> r, t -> r, r, k - size(t -> l) - 1), l = t;
  pull(t);
}

void merge(node *&t, node *l, node *r){
  push(l), push(r);
  if(l == NULL || r == NULL) t = l ? l : r;
  else if(l -> p > r -> p) merge(l -> r, l -> r, r), t = l;
  else merge(r -> l, l, r -> l), t = r;
  pull(t);
}

void reverse(node *t, int l, int r){
  // Reverses the range [l,r].
  node *a, *b;
  split(t, t, b, r + 1);
  split(t, t, a, l);
  a -> f ^= 1;
  merge(t, t, a);
  merge(t, t, b);
}
```

6.6 Sweep Line

Oftentimes when trying to solve geometry problems, or combinatorial problems that we give a geometric interpretation, we have something that involves one or two dimensions. If this is difficult to deal with directly, we can use a sweep line to reduce the dimension of the problem.

Sweeplines are best explained with an example. Suppose we're given several intervals on the number line, and we want to find the length of their union. If we wanted to, we could use a segment tree with lazy propagation to insert the intervals and calculate the total length. However, a sweep line gives us a much easier approach. Imagine a vertical beam of light sweeping from $-\infty$ to ∞ on the number line. At any given time, we keep track of the number of intervals that are hit by this beam. This takes the problem from one dimension to zero dimensions. Our answer is then the length over which our beam hit a positive number of intervals.

Our implementation of a sweep line simulates this process. If we sort the endpoints of our intervals by their positions on the number line, we can simply increment our counter each time we encounter a "start" endpoint and decrement our counter each time we encounter an "end" endpoint. We can look at the regions between consecutive endpoints, and add to our result if our counter is positive in that region.



One other way to interpret sweep lines is to consider time as the dimension we sweep along. For the example above, this would mean that each interval appears and then disappears on our beam, existing only when the time is between its endpoints. Although time may not seem useful for our one dimensional problem, this type of thinking helps in higher dimensions.

Most sweep lines that you use won't be as simple as this. Sweepline problems usually involve reducing a two dimensional problem to a one dimensional problem and require maintaining data structures such as BBSTs or segment trees along that dimension. This technique also generalizes to higher dimensions—to solve a three dimensional problem, we can sweep along one dimension and use two dimensional data structures to maintain the other two dimensions.

To finish, let's go over another example, Cow Rectangles from USACO 2015 January:

`The locations of Farmer John's` N `cows` $(1 \leq N \leq 500)$ `are described by distinct points in the 2D plane. The cows belong to two different breeds: Holsteins and Guernseys. Farmer John wants to build a rectangular fence with sides parallel to the coordinate axes enclosing only Holsteins, with no Guernseys (a cow counts as enclosed`

`even if it is on the boundary of the fence). Among all such fences, Farmer John wants to build a fence enclosing the maximum number of Holsteins. And among all these fences, Farmer John wants to build a fence of minimum possible area. Please determine this area. A fence of zero width or height is allowable.`

The first observation we should make is that we want a Holstein on every side of the fence. Otherwise, we would be able to decrease the area of our fence without decreasing the number of Holsteins. Even with our observation, this problem still seems hard to deal with. We can make it easier by adding another constraint: Suppose we knew which Holstein was on the leftmost boundary of our fence. Since we added the constraint to the x-dimension, we naturally want to figure out where our rightmost Holstein is next. We can do this with a sweep line moving to the right.

The reason we want a sweep line here is because for any given rightmost cow, we want to know information about the Guernseys and Holsteins inbetween. With a sweep line, we can collect and maintain all this data by processing the cows from left to right. For example, whenever we see a Guernsey, we have to limit the y-coordinates that our Holsteins can take. And whenever we see a Holstein, we have to consider this Holstein as our rightmost cow and also store it in case we include it in a fence later on.

What remains is to find an appropriate data structure to track all this. A set data structure (STL set or Java TreeMap) turns out to be enough. We can insert the Holsteins, sorted by y-coordinate, and delete one whenever any rectangle bounded by that Holstein, the leftmost Holstein, and our sweep line includes a Guernsey. Thus we take $O(n \log n)$ time to sweep, giving us an $O(n^2 \log n)$ solution overall.

This type of analysis is pretty representative of sweep line problems. Whether you're given rectangles, line segments or even polygons, you want to think about how you can reduce the dimension and obtain a tractable problem. Note that sweep lines don't always have to move along some axis. Radial sweep lines (sweeping by rotating) and sweeping at an angle (rotating the plane by 45°) also work.

Chapter 7

Tree Algorithms

Up until now, we have only looked at algorithms that deal with general graphs. However, there is also much to be said about graphs with additional structure. In this section, we'll explore some problems and their solutions on trees. First, some definitions.

An undirected graph G is a *tree* if one of the following equivalent conditions holds:

- G is connected and has no cycles.
- G is connected with n vertices and $n - 1$ edges.
- There exists exactly one simple path between any two vertices of G.

A *rooted tree* is a tree where one vertex has been designated as the *root*. This gives each edge a natural direction – whether it leads towards or away from the root. In many problems, it is useful to arbitrarily designate a root. For example, one interpretation of a DFS on a tree is that we start from the root and traverse the tree downwards towards the leaves. (On trees, a *leaf* is a vertex with degree 1.)

We'll define a few more terms for rooted trees. An *ancestor* of a vertex v is another vertex a that lies on the path between v and the root. The *parent* of a vertex is its closest ancestor. The *depth* of a vertex is its distance from the root. We will use $depth(v)$ to denote the depth of a vertex v.

7.1 DFS on Trees

DFS is a very useful technique for collecting information about the structure of a tree. For example, we can compute in linear time the depth of each node, the size of each subtree or the diameter of the entire tree. This can be done by recursively computing results for each subtree, and then combining this data to obtain the final result. Sometimes, it takes more than one DFS to collect all the necessary information. Calculating node depth and subtree size via DFS is relatively straightforward. Below is a snippet of pseudocode computing these two sets of values.

```
function DFS(v, p)                          ▷ v is the current vertex, p is its parent
    sum(v) ← 1                              ▷ sum(v) is size of the subtree rooted at vertex v
    depth(v) ← depth(p) + 1                 ▷ depth(v) is the depth of vertex v
    for all vertices n adjacent to v do
        if n ≠ p then
            DFS(n, v)
            sum(v) ← sum(v) + sum(n)
```

Computing the diameter of a tree is a bit trickier. For a given tree, let r be its root, and let a be a vertex of maximum depth. It turns out that the diameter of the tree is the maximum distance between a and any other vertex in the tree. (Try to prove this!) Thus we can run two DFSes to compute the diameter, calculating the depth of each vertex with each pass.

Depth, subtree size and diameter are only a few of the values we can compute for trees. This style of DFS shows up often in problems and is also fundamental to many more complex tree algorithms.

7.2 Jump Pointers

Jump pointers are a technique for decomposing paths on rooted trees. The central idea is that we can always take a path and break it into $O(\log n)$ chunks of size 2^i. By augmenting these smaller chunks with data, we can use jump pointers to quickly answer a variety of queries, including level ancestor and lowest common ancestor.

Let's start with an example. The most fundamental application of jump pointers is to level ancestor queries—for any vertex v, we want to be able to find its kth ancestor in $O(\log n)$ time.To solve this problem, we precompute for each vertex a pointer to its 2^ith ancestor for all possible values of i. These are the "jump pointers" for which this technique is named. If our tree has n vertices, then each vertex has at most $\lfloor \log_2 n \rfloor$ pointers leading out from it. Thus we have at most $O(n \log n)$ jump pointers in total. We can compute these with a DP in $O(n \log n)$.

Using jump pointers, we can finish the level ancestor problem. Let 2^i be the largest power of 2 that is at most k. We use our jump pointers to obtain u, the 2^ith ancestor of v. If 2^i is not equal to k, we can jump up again from u. Continuing recursively, we'll finish in $O(\log n)$ iterations, since we reduce k by at least a factor of 2 each time.

Now we'll tackle lowest common ancestor queries. The lowest common ancestor (LCA), l, of two vertices u and v is the deepest vertex that is an ancestor of both u and v. LCA queries are especially useful because any path on a rooted tree can be broken into two shorter paths joined at the LCA of its endpoints.

The solution of the LCA problem with jump pointers consists of two steps. The first step involves bringing the two queried vertices to the same depth. If we are looking for the LCA of u and v, we can assume that $depth(u) > depth(v)$ without loss of generality. Let u' be the ancestor of u satisfying $depth(u') = depth(v)$. We can compute u' with a single level ancestor query in $O(\log n)$ time. If $u' = v$, then we are done.

Otherwise, if $u' \neq v$, we can find the LCA of u' and v by advancing them towards the root in increments of 2^i in a binary-search-like manner. If the 2^ith ancestors of u' and v are distinct, then the LCA of u' and v is equal to the LCA of the 2^ith ancestors of u' and v. Thus, iterating down

from the largest power of 2 less than n, we can move u' and v up the tree until they share a parent. This common parent is the LCA of u and v. Since jump pointers allow us to access the 2^ith ancestor of a vertex in $O(1)$, our algorithm runs in $O(\log n)$ time.

Algorithm 8 Jump Pointers, Level Ancestor and LCA

```
function BuildJumpPointers(V, par)        ▷ Initially, par(0, v) holds the parent of vertex v.
    par(i, v) denotes the 2^i th ancestor of vertex v
    for all i from 1 to ⌊log_2 n⌋ do
        for all vertices v do
            par(i, v) ← par(i − 1, par(i − 1, v))
function LevelAncestor(u, k)
    for all i from ⌊log_2 n⌋ to 0 do
        if 2^i ≤ k then
            u ← par(i, u)
            k ← k − 2^i
    return u
function LCA(u, v)
    if depth(u) < depth(v) then
        swap(u, v)
    u ← LevelAncestor(u, depth(v) − depth(u))
    if u = v then
        return u
    for all i from ⌊log_2 n⌋ to 0 do
        if par(i, u) ≠ par(i, v) then
            u ← par(i, u)
            v ← par(i, v)
    return par(0, u)
```

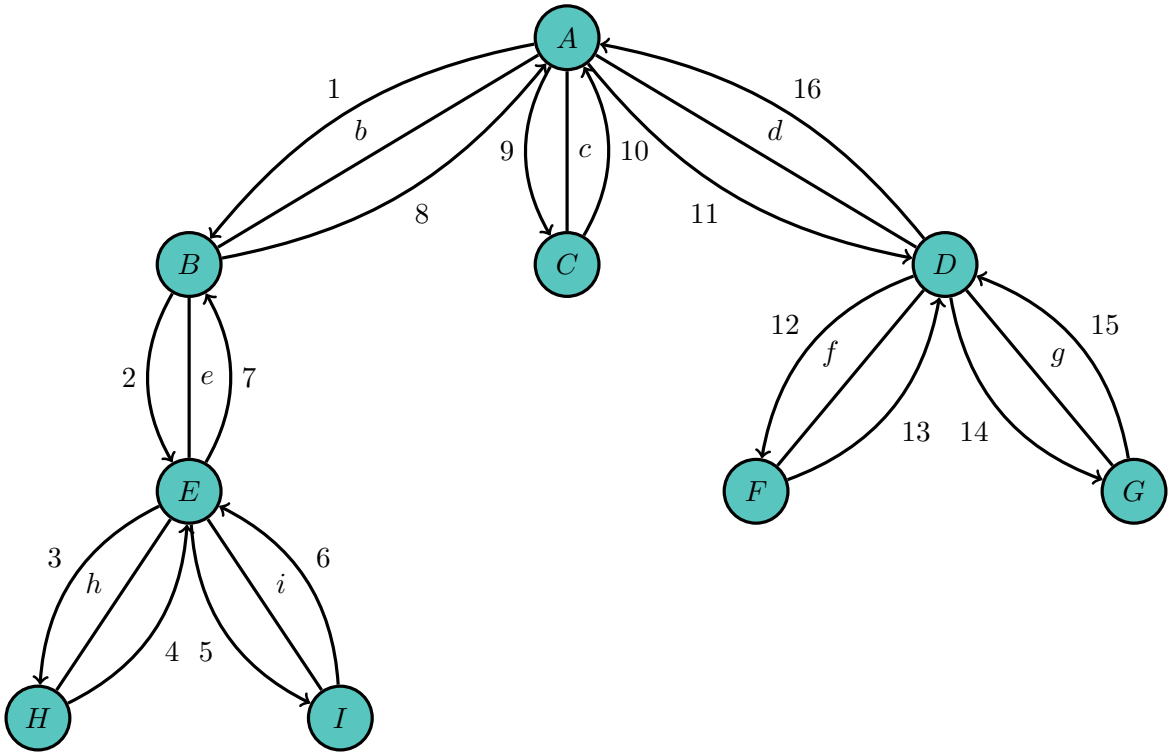
Level ancestor and LCA are the two basic tools to jump pointers. Applying these, we can quickly answer queries about paths, such as the length of the path between two vertices u and v. If we augment the jump pointers by storing additional information, we can also compute maximum weight or distance queries on weighted trees. Another important observation that we can make is that we can compute jump pointers on the fly, adding edges and answering queries online.

We now take some time to acknowledge a few limitations of jump pointers. Because each vertex of the tree is covered by $O(n)$ jump pointers on average (including the ones jumping over it), this structure cannot handle updates efficiently. Thus we usually apply jump pointers only if we know that the weights/values stored with the jump pointers will not change after being computed. For example, if we want to both answer maximum weight queries for paths and update edge weights, we should use heavy-light decomposition or link-cut trees to do so.

Overall, however, jump pointers are still a very flexible technique. With some creativity, these ideas can be used to provide elegant and easy-to-code solutions for problems that would otherwise require more complex data structures.

7.3 Euler Tour Technique

The *Euler tour technique* is a method of representing a tree (not necessarily binary) as a list. The idea is that a tree is uniquely determined by its DFS traversal order, going both down and up the tree. We'll keep track of the order in which we traverse the edges. Since each edge is traversed twice, each edge will appear in our list twice. We'll also name each edge by the child vertex in the parent-child pair.



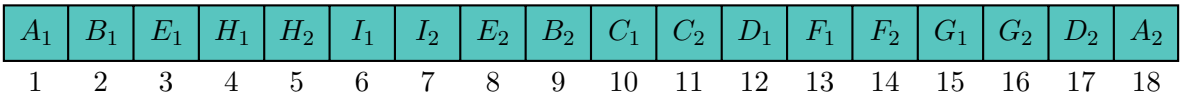
The edge traversal order is then described by the ordered list below.

b_1	e_1	h_1	h_2	i_1	i_2	e_2	b_2	c_1	c_2	d_1	f_1	f_2	g_1	g_2	d_2
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

We see a pattern in this list – the subtree of a node is contained between the two edges representing that node in the list, inclusive. For example, from the first e to the second e, the entire subtree of E is contained within the range $[2, 7]$.

7.3.1 Euler Tour Tree

The *Euler tour tree* uses this idea. Because we see the relationship between the list and the subtrees of the tree, we'll just have the list store the vertex names. To complete this pattern for the subtree of A, which is the entire tree, we'll simply add that letter to the beginning and end of the list.

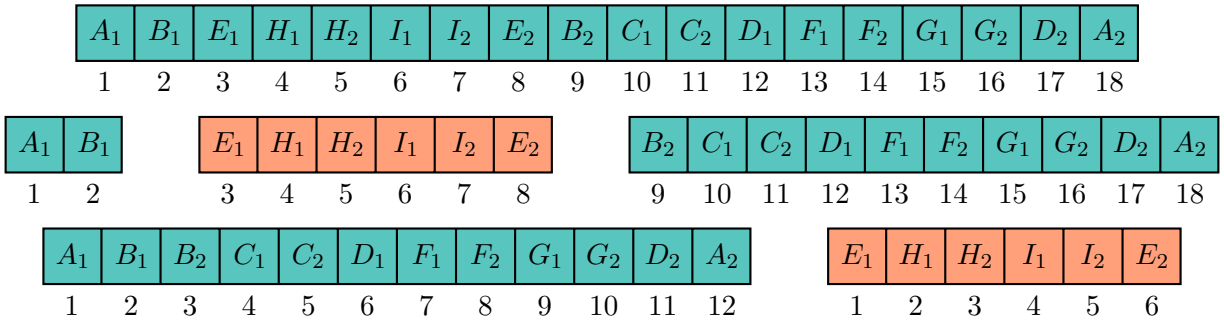


Now suppose we wanted to perform some operations on a forest, or collection of trees. The first operation is to *find* the root of the tree containing a vertex v. The second is to *link* the trees of two vertices u, v together by making v the parent of u. The third is to *cut* the subtree of v away from the rest of the tree by deleting the edge from v to its parent.

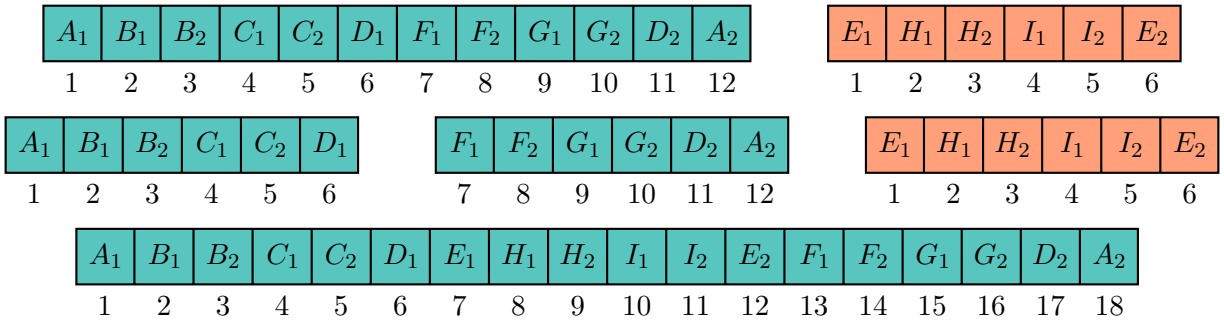
Note that, without the cut operation, this is simply union-find. However, because of the cut operation, we have to maintain the structure of the original forest, as otherwise we lose track of the actual parent of any given node.

We see that all operations are not simply on a single vertex but on an entire subtree. However, if we look at what each of the operations does to the Euler tour lists representing our forest, find returns the first element in the list, link places one ordered list within another, and cut removes a large contiguous section of one ordered list.

To cut E from the tree, we need to split the list in two:



Let's see what happens when we link E to D. We need to split the first list immediately after D_1.



We have a data structure that maintains a set of ordered elements that can split and merge quickly. Splay trees can maintain an ordered list, and the split and join operations on splay trees can easily implement link and cut. Each tree of our forest is then represented by a splay tree maintaining that tree's Euler tour list, and we can support each of the three necessary operations in $O(\log n)$ amortized time.

7.4 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.

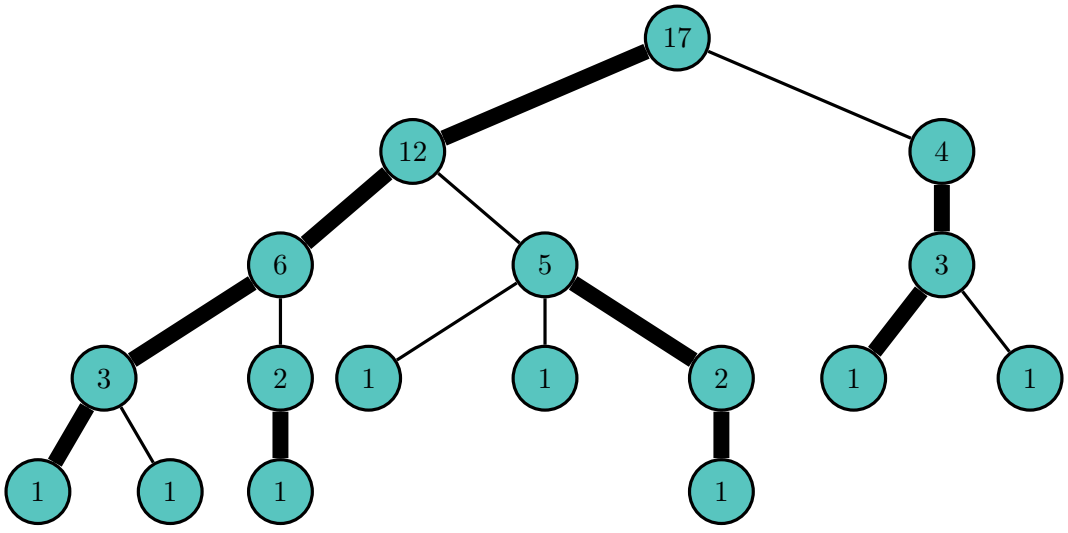
Certain problems can be solved very nicely in a list with segment trees and similar data structures. Heavy-light decomposition provides us with a way to exploit the fact that long "chains," which are present in unbalanced trees, might be slow in brute force but can be sped up with other data structures.

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 $\log n$.



7.5 Link-Cut Tree

Chapter 8

Strings

8.1 String Hashing

String hashing is a probabilistic technique for dealing with strings. The most common hashing algorithm used in programming contests is the *polynomial hash*, also known as the *Rabin-Karp hash*. This method allows for the fast comparison of strings and their substrings—$O(1)$ after linear time precomputation. Thus we can often use conceptually simpler hashing to replace trickier string algorithms, such as Knuth-Morris-Pratt and the Z-algorithm. In addition, adding binary search allows for the computation of the longest common prefix in $\log n$ time, useful in constructing suffix arrays.

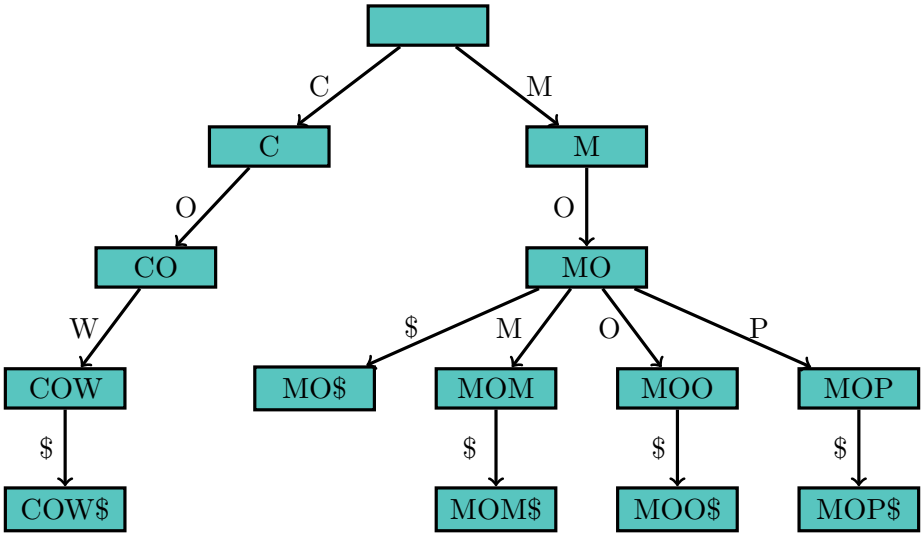
Jonathan Paulson gives a great introduction to polynomial hashing: `https://www.reddit.com/r/usaco/comments/39qrta/string_hashing/`

8.2 Knuth-Morris-Pratt

Knuth-Morris-Pratt is an easy-to-code linear time string matching algorithm. Using the "needle and haystack" analogy, for a given needle of length n and a haystack of length m, KMP takes $O(n)$ to preprocess the needle and $O(m)$ to search. Hariank and Corwin explain the algorithm well: `https://activities.tjhsst.edu/sct/lectures/1415/stringmatching_10_3_14.pdf`

8.3 Trie

A trie (from re*trie*val) is a data structure for storing strings that supports insertion and look-up in linear time. The trie maintains the strings in a rooted tree, where each vertex represents a prefix and each edge is labeled with a character. The prefix of a node n is the string of characters on the path from the root to n. (In particular, the prefix of the root is the empty string.) Every string that is stored in the tree is represented by a path starting from the root. Below is a picture of a trie storing "COW," "MO," "MOM," "MOO," and "MOP."



Insertion into a trie is straightfoward. We start at the root, and create edges and vertices as necessary until a path representing the string is formed. If a vertex already has an edge of the corect character leading out from it, we just travel down that edge. In order to identify the end of a string, we can append a "$" to every string before we insert. Searching is the same as insertion, except we terminate our search when we can't go any further instead of adding a new edge.

What we've seen so far doesn't make the trie seem particularly useful—string insertion and look up can be easily done in linear time with Rabin-Karp and a hash table as well. The advantage of the trie comes from its tree structure. Having common prefixes bundled together on the same path means we can compute compute information relating to prefixes easily. This allows for tree DPs that we wouldn't be able to do with hashing. (However, hashing does handle certain forms of prefix queries well, such as longest common prefix.)

Tries are also a building block for other string data structures, such as the Aho-Corasick automaton and the suffix tree. (In fact, tries can be viewed as a very simple form of string automaton.)

8.4 Suffix Array

For a string of length n, each index i $(0 \leq i < n)$ can represent the suffix that starts at that index. A *suffix array* is a list of these indices, sorted in lexicographically increasing order by suffix. Thus a suffix array is essentially a sorted list of the suffixes of a string. Below is the suffix array of "mississippi$."

Suffix Array	Suffix
11	$
10	i$
7	ippi$
4	issippi$
1	ississippi$
0	mississippi$
9	pi$
8	ppi$
6	sippi$
3	sissippi$
5	ssippi$
2	ssissippi$

At first, it might seem surprising that such a structure is useful—why do we care about suffixes at all? The crucial observation is that every substring of a string is a prefix of a suffix. Thus if we have something that does well with prefixes, such as hashing or a trie, we use this to compute information about substrings. A trie built from suffixes is known as a *suffix tree*, which we'll cover later. In this section, we'll go over what we can do with hashing and a suffix array.

First, let's figure out how we can construct a suffix array. The naive solution is to sort all n suffixes using $O(n)$ string comparison. Since sorting itself takes $O(n \log n)$ comparisons, we have an $O(n^2 \log n)$ algorithm. However, with hashing and binary search, we can lexicographically compare two strings in $O(\log n)$ time. We do this by binary searching for the longest common prefix and then comparing the next character. We can can compute the hash of any substring with a polynomial hash, so it's easy to compare if the prefixes of two suffixes (a.k.a two substrings) are equal. Now that we have $O(\log n)$ comparison, our algorithm runs in $O(n \log^2 n)$ overall.

Note that it is also possible to compute suffix arrays in $O(n \log n)$ and even $O(n)$, but these algorithms are much more involved. For the purposes of contests, an $O(n \log^2 n)$ suffix array algorithm should almost always be enough.

With a suffix array, we can check if any queried "needle" string exists using a binary search with hashing. We can even count the number of occurences by binary searching for the first and last macth. This works in $O(m + \log n \log m)$, where m is the length of the needle, because we need $O(\log m)$ operations for string comparison and $O(\log n)$ iterations of the binary search. Suffix arrays differ from KMP because KMP preprocesses the "needle," while suffix arrays preprocess the "haystack."

From the suffix array, we can also obtain another useful data structure called the *LCP array*. This array stores the longest common prefix between adjacent suffixes in the suffix array. We can use it to speed up pattern matching to $O(m + \log n)$. In addition, we can build a segment tree over the LCP array to answer queries, such as the LCP between two arbitrary suffixes.

To handle multiple strings with suffix arrays, we can either concatenate them with separator characters in between or separately compute their hashes.

Below is a short C++ code for computing suffix arrays. Look up lambda functions in C++ if you're not familiar with the syntax.

```
typedef unsigned long long ull;
// We use A as the exponent and M as the mod for our hash.
const ull M = 1000000007;
const ull A = 127;

void build(string S, int *sa){
  // Builds a suffix array for a string S in array sa.
  S.push_back('$');
  int N = S.size();
  ull P[N + 1], H[N + 1];
  P[0] = 1; // P stores the precomputed powers of A.
  H[0] = 0; // H stores the prefix hashes of S.
  for(int i = 0; i < N; i++){
    // Precomputing powers and hashes.
    P[i + 1] = A * P[i] % M;
    H[i + 1] = (A * H[i] + S[i]) % M;
    sa[i] = i;
  }
  auto f = [&](int a, int l){
    // Returns the hash of the substring starting at index a with length l.
    return (H[a + l] - H[a] * P[l] % M + M) % M;
  };
  auto comp = [&](int a, int b){
    // Compares the suffixes starting at indices a and b.
    int lo = 0, hi = min(N - a, N - b);
    while(lo + 1 < hi){
      int m = (lo + hi) / 2;
      (f(a, m) == f(b, m) ? lo : hi) = m;
    }
    return S[a + lo] < S[b + lo];
  };
  sort(sa, sa + N, comp);
}
```

8.5 Aho-Corasick

One way to think about the Aho-Corasick algorithm is KMP on a trie. This algorithm is used for matching multiple needles in a single haystack in linear time. To do this, we first add all strings to a trie that stores some extra information at its nodes. Upon inserting each string, we mark the last node visited as a endpoint node. Each node, in addition to storing its children and its status as an endpoint node, stores a *suffix link*. This link points to the longest proper suffix that is also a node on the trie. If necessary, each node can also store a *dictionary suffix link*, which points to the first endpoint node reachable by only following suffix links.

To build an Aho-Corasick automaton, we start with a trie of the needle strings. What we want to do is compute the suffix links. We simplify this by using an $O(nk)$ approach, where k is the alphabet size. For each node n, we compute an additional failure function, with one value corresponding to each letter of the alphabet. Suppose p is the prefix represented by n. Then the failure function of n for the letter α points to the node representing longest suffix of $p + \alpha$ in the trie.

We can compute all of this with a BFS. When we are at a node, we can find the suffix links of its children using the failure function of its own suffix link. We also have to calculate the node's

own failure function. For every letter that has a corresponding child, its failure function is equal to that child. Otherwise, its failure function is equal to the corresponding failure function of the node's suffix link. Take a moment to think through why this works.

To query, we can iterate through our haystack string character by character and make the corresponding moves in the automaton. We follow the failure function when no child exists. However, note that there isn't really a distinction between a normal trie edge and a failed edge anymore. To check for matches, we can look at the values we store at each node and/or follow dictionary suffix links. Here's an implementation of Aho-Corasick in C++ without dictionary suffix links:

```
const int SIGMA = 26;
const int MAXN = 100005;
// Each node is assigned an index where its data is stored.
// ch first stores the children of each node and later the failure function.
// val counts the number of strings ending at a given node.
// link stores suffix links.
int ch[MAXN][SIGMA], val[MAXN], link[MAXN], sz = 1;
// Array q is a hand-implemented queue.
// h and t point to the head and tail, respectively.
int q[MAXN], h, t;

void add(string s){
  // Adds a string to the trie.
  int p = 0; // p tracks our current position and starts at 0, the root.
  for(int i = 0; i < s.size(); i++){
    int c = s[i] - 'A';
    // ch[p][c] is 0 if null, so we have to allocate a new node/index.
    if(!ch[p][c]) ch[p][c] = sz++;
    p = ch[p][c];
  }
  val[p]++; // Updates endpoint marker.
}

void build(){
  // Computes all suffix links with a BFS, starting from the root.
  h = t = 0;
  q[t++] = 0;
  while(h < t){
    int v = q[h++], u = link[v];
    val[v] += val[u]; // Propagates endpoint marker.
    for(int c = 0; c < SIGMA; c++){
      if(ch[v][c]){
        // If child exists, we create a suffix link.
        // The node's failure function here is the child.
        // Also, note that we have a special case if v is the root.
        link[ch[v][c]] = v ? ch[u][c] : 0;
        q[t++] = ch[v][c];
      } else {
        // Otherwise, we need to figure out its failure function.
        // We do this by using the failure function of the node's suffix link.
        ch[v][c] = ch[u][c];
      }
    }
  }
}
```

8.6 Advanced Suffix Data Structures

I've yet to see these used in a contest, but they do exist.

8.6.1 Suffix Tree

8.6.2 Suffix Automaton

Chapter 9

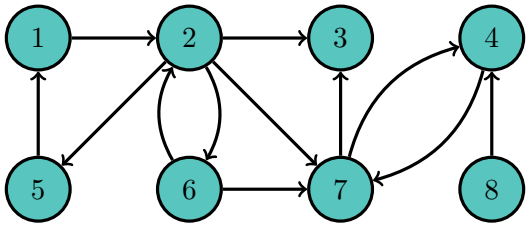
More Graph Algorithms

With the renewal of strongly connected components and network flow in the IOI syllabus, we overview some more graph algorithms that are standard but also noticeably more difficult than other algorithms presented in earlier chapters.

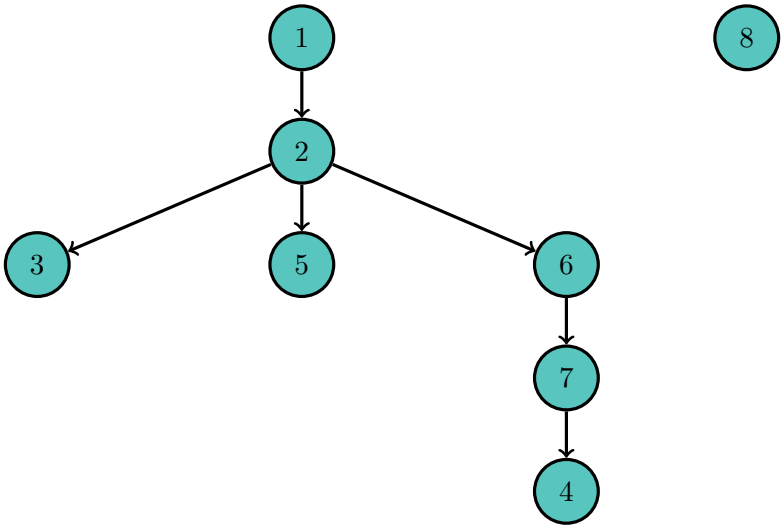
9.1 Strongly Connected Components

This section covers Tarjan's algorithm detects strongly connected components in a directed graph. We begin with the DFS traversal of the graph, building a forest (collection of trees). In a DFS, we only traverse each vertex once; this means when we encounter an edge to a vertex we already visited, we move on without calling the DFS on that node. Once we obtain the forest, we can split it into subgraphs that represent our strongly connected components.

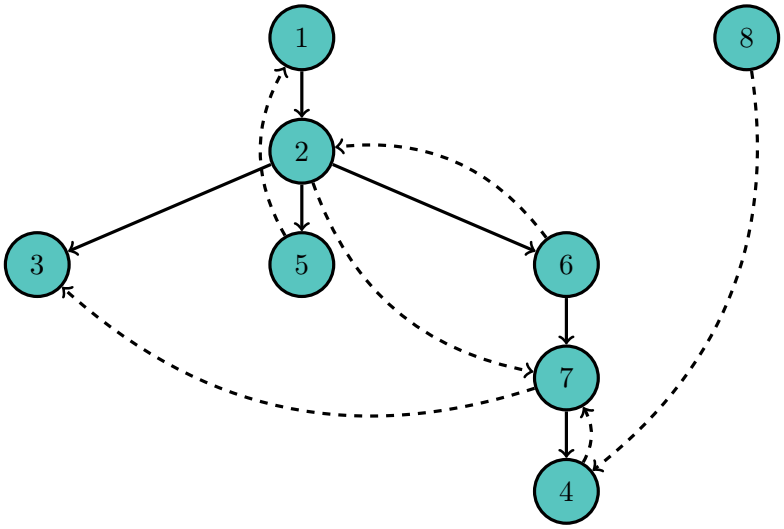
Let's work through how we can get the subgraphs representing the strongly connected components from our DFS. Here's the graph from the beginning of this section.



It doesn't matter from which node we begin our DFS or the order in which we choose children; in our example, we'll simply choose the node with the minimum label to traverse first. This will result in two trees in our forest, one rooted at 1 and the other rooted at 8.



Now this by itself is not very useful, as in order to find strongly connected components, we'll need the edges in the directed graph that aren't included in our tree. Here we add the other edges as dashed pointers.



Note that strongly connected components are represented by subtrees in the graph. We call the root of the subtree called the root of the strongly connected component.

One edge here is immediately useless. We already know that 2 can reach 7; 7 is in 2's subtree. The fact that there is an edge from 2 to 7 doesn't change anything. Then we have a crucial observation – the only possible useful extra edges are those that go up to a previous node in the subtree, like the edge from 5 to 1, or those that go "left" to a previously-visited vertex, either to a previous branch in the tree, like from 7 to 3, or to a previous subtree entirely, like from 8 to 4.

If a node v has an edge to a direct ancestor in the tree, that means we immediately have a cycle, and therefore the node, its ancestor, and every vertex along the way must be in the same strongly connected component.

Naturally, the "left" case is trickier. Suppose v has a left edge to a vertex u. We somehow need a way to find out if u has a path back to v. We know that u cannot go down the tree to v as v is not in the subtree of u by the way DFS constructed the tree. Therefore, we want to know whether u has a path back up to some common ancestor with v. However, again by the way DFS traverses the graph, the entire subtree of u has already been searched before the DFS reaches v. We want to exploit this fact with some kind of memoization.

If vertex v was the nth vertex visited in the DFS, we'll mark v with the label $order(v) = n$. We'll also keep track of the "least" vertex $link(v) = u$ that we know up to that point that v can visit, or the vertex u with the minimum $order(u)$ that v can reach so far.

As we're using a DFS, we'll use a stack S to keep track of nodes we've visited. In a normal DFS on a tree, once we finish exploring a vertex v, we pop off v from the stack. This will not be the case for us. A node remains on the stack iff it has a path to some node earlier in the stack.

This means as we explore the descendents of a vertex v, we'll know if v has a path back to a previous vertex. That is, if $link(v) < order(v)$, it stays on the stack. If $link(v) = order(v)$, we take it off the stack.

Now we describe Tarjan's algorithm. Here, num represents a global variable that indicates how many vertices have been visited so far.

Algorithm 9 Tarjan

```
function StrongConnect(vertex u)
    num ← num + 1                                   ▷ increment num
    order(u) ← num                                  ▷ set order(u) to smallest unused number
    link(u) ← order(u)                              ▷ least order(v) accessible is u itself
    push u on S
    for all neighbors v of u do
        if order(v) is undefined then               ▷ v has not been visited
            StrongConnect(v)
            link(u) ← min(link(u), link(v))
        else if v is on stack S then                ▷ v is in current component
            link(u) ← min(link(u), order(v))
    if link(u) = order(u) then                      ▷ u is root of component, create SCC
        create new strongly connected component
        repeat
            v ← top of S
            add v to strongly connected component
            pop top from S
        until u = v
function Tarjan(G(V, E))
    num ← 0
    initialize new empty stack S
    for all vertices v ∈ V do
        if order(v) is undefined then               ▷ v has not been visited
            StrongConnect(v)
```

This algorithm runs in $O(V + E)$.

9.2 Network Flow

We are given a directed graph with weighted edges, a source node, and a sink node. A *flow* is sent from the source to the sink. Each edge weight represents the maximum capacity of that edge. For every node besides the source and the sink node, total flow in is equal to total flow out. We can think of a flow network as a series of pipes through which water travels from an entrance to an exit in the network. The edge capacities represent pipe thickness. At any node, the total rate at which water enters the node must equal the total rate at which it exits the node, and along any path, the rate at which water flows is bottlenecked by the thinnest pipe.

More formally, for a graph $G(V, E)$, where $c(u, v)$ represents the capacity of the edge from u to v, the flow $f(u, v)$ from a source s to a sink t satisfies

$$\begin{aligned} f(u,v) &\leq c(u,v) \forall (u,v) \in E \\ f(u,v) &= -f(v,u) \forall (u,v) \in E \\ \sum_{v \in V} f(u,v) &= 0 \forall u \in V \setminus \{s,t\} \\ \sum_{v \in V} f(s,v) &= \sum_{v \in V} f(v,t) = |f|, \end{aligned}$$

where $|f|$ represents the total flow from the source to the sink.

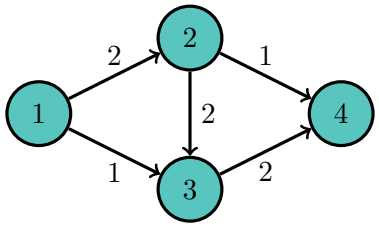
A *maximum flow* is one that maximizes the total flow $|f|$ from s to t, or in our example, maximizes the rate at which water can flow through our network.

We'll also define the residual capacity $c_f(u, v) = c(u, v) - f(u, v)$. Note that $c_f(u, v) \geq 0$ by the conditions imposed on f. The residual capacity of an edge represents how much capacity is left after a certain amount of flow has already been sent. We therefore have the residual graph $G_f(V, E_f)$, where E_f is the graph of residual edges, or all edges $(u, v) \in V^2$ satisfying $c_f(u, v) > 0$.

A natural approach to "solving" this problem would be to simply greedily add flow.

Find a path from the source to the sink in which all the edges have positive weight in the residual graph. Send flow along this path; that is, 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 residual capacity of every edge in the path. We repeat, and this is guaranteed to terminate since on any given move, we remove an edge from our residual graph.

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. This is because the best possible single path from the source to the sink may not be included the best possible overall flow.

9.2.1 Ford-Fulkerson

We somehow need a way to fix the inclusion of any suboptimal paths in our greedy approach, or to "send flow back" in case we sent it through a suboptimal path. We do this by introducing the *reverse edge* to our residual graph.

Find a path from the source to the sink in which all the edges have positive weight in the residual graph. 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 residual capacity of every edge along the path and *increment the residual capacity of the reverse edge* (the edge connecting the same two vertices but running in the opposite direction) by *cap*. We call this operation on the path *augmenting the path*. We simply choose an augmenting path until no such paths exist.

Algorithm 10 Ford-Fulkerson

```
function AugmentPath(path p = {v_i}_{i=1}^m, where (v_i, v_{i+1}) ∈ E_f, v_1 = s, v_m = t)
    cap ← min_{i=1}^{m-1}(c_f(v_i, v_{i+1}))
    for i ≡ 1, m − 1 do
        f(v_i, v_{i+1}) ← f(v_i, v_{i+1}) + cap
        c_f(v_i, v_{i+1}) ← c_f(v_i, v_{i+1}) + cap
        f(v_{i+1}, v_i) ← f(v_{i+1}, v_i) − cap
        c_f(v_{i+1}, v_i) ← c_f(v_{i+1}, v_i) + cap          ▷ incrementing reverse edge
function MaxFlow(G(V, E), s, t ∈ V)
    for all (u, v) ∈ V^2 do
        f(u, v) ← 0
        c_f(u, v) ← c(u, v)
    |f| ← 0
    while ∃p = {v_i}_{i=1}^m, where (v_i, v_{i+1}) ∈ E_f, v_1 = s, v_m = t do
        cap ← min_{i=1}^{m-1}(c_f(v_i, v_{i+1}))
        |f| ← |f| + cap
        AugmentPath(p)
    return |f|
```

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 in our residual graph are called *augmenting paths*.

This algorithm is guaranteed to terminate for graphs with integral weights. Its performance is bounded by $O(Ef)$, where f is the maximum flow and E is the number of edges, as finding a path from s to t takes $O(E)$ and increments the total flow f by at least 1. The concept of removing edges can't be used to produce a stricter bound because while an edge in one direction may be removed from the residual graph, doing so creates an edge in the other direction.

In its crudest form, Ford-Fulkerson does not specify on which path to push flow if multiple paths exist. It simply states that as long as such a path exists, push flow onto it. In addition to being slow, Ford-Fulkerson, as it is stated, is not guaranteed to terminate for graphs with non-integral capacities. In fact, it might not even converge to the maximum flow for irrational capacities. However, these problems can be fixed by simply specifying how the algorithm chooses the next path on which to

push flow. Nonetheless, the Ford-Fulkerson algorithm is formulated beautifully mathematically and such is useful from a math perspective, as we will see with the Max-Flow Min-Cut Theorem.

9.2.2 Max-Flow Min-Cut Theorem

On a graph $G(V, E)$, an *s-t cut* $C = (S, T)$ splits the partitions V into S and T satisfying $s \in S$ and $t \in T$. The *cut-set* of C is the set of edges $\{(u, v) \in E : u \in S, v \in T\}$. The *capacity* of an s-t cut is given by

$$c(S, T) = \sum_{(u,v)\in S\times T} c(u, v).$$

A *minimum cut* is an s-t cut that minimizes $c(S, T)$.

A cut represents a set of edges that, once removed, separates s from t. A minimum cut is therefore a set of edges that does this but minimizes the total capacity of the edges necessary to disconnect s from t.

The max-flow min-cut theorem states that the maximum value of any s-t flow is equal to the minimum capacity of any s-t cut.

First, the capacity of any cut must be at least the total flow. This is true by contradiction. Any path from s to t has an edge in the cut-set, so therefore any flow from s to t is upper bounded by the capacity of the cut. Therefore, we need only construct one flow and one cut such that the capacity of the cut is equal to the flow.

We consider the residual graph G_f produced at the completion of the Ford-Fulkerson augmenting path algorithm.[1] Let the set S be all nodes reachable from s and T be $V \setminus S$. We wish to show that $C = (S, T)$ is a cut satisfying $|f| = c(S, T) = \sum_{(u,v)\in S\times T} c(u, v)$. This is true when the following is satisfied:

1. All edges $(u, v) \in S \times T$ are fully saturated by the flow. That is, $c_f(u, v) = 0$.
2. All reverse edges $(v, u) \in T \times S$ have zero flow. That is, $f(v, u) = 0$, or $c_f(v, u) = c(v, u)$.

The first condition is true by the way we constructed S and T, as if there existed a (u, v) where $c_f(u, v) > 0$, then v is accessible to s and ought to have been in S.

The second condition is true by the way the Ford-Fulkerson algorithm constructed reverse edges. If net flow was sent from v to u, then a reverse edge was constructed from u to v, so again, v is accessible to s, which is a contradiction.

Therefore, we have the flow

$$|f| = \sum_{(u,v)\in S\times T} c(u, v) - \sum_{(v,u)\in T\times S} 0 = c(S, T),$$

so we constructed a flow and a cut such that the flow $|f|$ is equal to the cut capacity $c(S, T)$, and we are done.

[1] If you're concerned that the Ford-Fulkerson algorithm will never terminate, there always exists a sequence of paths chosen such that it will. Edmonds-Karp is one example that always terminates.

9.2.3 Refinements of Ford-Fulkerson

As stated earlier, Ford-Fulkerson is limited by not specifying on which path to push flow. There are many algorithms that resolve this issue in different ways.

Edmonds-Karp

Edmonds-Karp fixes the problem by simply choosing the augmenting path of shortest unweighted length. This can be done easily using a BFS.

Algorithm 11 Edmonds-Karp

```
function ChoosePath(G_f(V, E_f), s, t ∈ V)                    ▷ BFS
    visited(v) denotes v has been added to queue
    prev(v) denotes vertex preceding v in BFS
    push s on queue Q
    visited(s) ← 1
    while Q is not empty do
        u ← top of Q
        for all neighbors v of u in G_f where visited(v) = 0 do
            push v on Q
            visited(v) ← 1
            prev(v) ← u
    pointer curr ← t
    while curr ≠ s do
        add curr to beginning of path p
        curr ← prev(curr)
    add s to beginning of p
    return p
function MaxFlow(G(V, E), s, t ∈ V)
    for all (u, v) ∈ V^2 do
        f(u, v) ← 0
        c_f(u, v) ← c(u, v)
    |f| ← 0
    while t can be reached from s do
        p ← ChoosePath(G_f, s, t)
        cap ← min_{i=1}^{m-1}(c_f(v_i, v_{i+1}))
        |f| ← |f| + cap
        AugmentPath(p)
    return |f|
```

The BFS is clearly $O(E)$. To complete our analysis, we must somehow bound the number of times we need to perform the BFS. To do this, we'll look at what pushing flow on a path does to our residual graph; in particular, how it affects our BFS traversal tree. Note that each vertex is on some level i in the BFS tree, characterized by the distance from the source s. For example, $L_0 = \{s\}$, L_1 contains all the neighbors of s, L_2 contains neighbors of neighbors not in L_0 or L_1, and so on.

We first claim that the level of any vertex in the graph is nondecreasing following an augment on a path p. If the augment saturates an edge, it may remove it from G_f, which cannot decrease the distance of any vertex from s. If the augment creates an edge $e = (u, v)$, that means we sent flow from v to u on the path p. Therefore, if v was originally level i, u must have been level $i + 1$. The level of u does not change by adding (u, v), and the level of v can either be i or $i + 2$, depending on whether edge (v, u) was deleted in the process. Either way, the level of all vertices is nondecreasing.

Now consider the bottleneck edge $e = (u, v)$ of an augmenting path p, where the level of u is i and the level of v is $i + 1$. The push operation deletes the edge e, but the level of v must stay at least $i + 1$. Now for the edge e to reappear in the graph G_f, flow must have been sent on the reverse edge $e' = (v, u)$ on some augmenting path p'. But on path p', u comes after v, which must be at least level $i + 1$. Therefore, u must be at least level i. But since the maximum level of a node that is connected to s is $V - 1$, an edge e can only be chosen as the bottleneck edge $\frac{V}{2}$ times, or $O(V)$.

There are E edges, each of which can be the bottleneck edge for $O(V)$ different augmenting paths, each of which takes $O(E)$ to process. Therefore, the Edmonds-Karp algorithm runs in $O(VE^2)$.

Dinic

Dinic's algorithm is another refinement of Ford-Fulkerson.

9.2.4 Push-Relabel

Unfortunately, even the much-improved bounds of the $O(VE^2)$ Edmonds-Karp are admittedly bad. While the push-relabel method for solving the max flow problem does not have the fastest theoretical bounds, two of its implementations have complexities $O(V^3)$ and $O(V^2\sqrt{E})$ and are among the fastest in practice.

Generic Push-Relabel

Ford-Fulkerson and its variants all deal with global augmentations of paths from s to t. Push-relabel takes a different perspective, introducing the concept of a *preflow* and a *height* to make local optimizations that ultimately result in the maximum flow.

A preflow maintains the same properties as a flow but modifies the conservation of flow condition. Instead of total flow in equalling total flow out, flow in must be at least, and therefore can exceed, flow out. We denote the difference between flow in and flow out as the *excess* $e(v)$.

$$\begin{aligned} f(u, v) &\leq c(u, v) \forall (u, v) \in E \\ f(u, v) &= -f(v, u) \forall (u, v) \in E \\ e(v) = \sum_{u \in V} f(u, v) &\geq 0 \forall v \in V \setminus \{s\} \\ e(s) &= \infty. \end{aligned}$$

The definitions of the residual capacity $c_f(u, v)$, edge set E_f, and graph G_f are the same as they were defined before, except with a preflow f instead of a normal flow.

We call a vertex $v \in V \setminus \{s, t\}$ *active* if $e(v) > 0$. Therefore, a vertex besides the source or sink is active if more flows into the vertex than flows out. s and t are *never* active. A preflow with no active vertices is simply a flow, at which point the excess of the sink $e(t)$ represents the value $|f|$ of the flow.

We can *push* flow from a node u to a node v by moving as much of the excess $e(u)$ to v as the capacity of the edge $c_f(u, v)$ will allow.

function PUSH(edge (u, v))
  $\delta \leftarrow \min(e(u), c_f(u, v))$ ▷ $c_f(u, v) = c(u, v) - f(u, v)$
  $f(u, v) \leftarrow f(u, v) + \delta$
  $f(v, u) \leftarrow f(v, y) - \delta$
  $e(u) \leftarrow e(u) - \delta$
  $e(v) \leftarrow e(v) + \delta$

The idea of the push-relabel algorithm is to first push as much preflow as possible through local optimizations in the direction the sink. When a node can no longer push flow to the sink, it pushes the excess back towards the source to turn the preflow into a flow.

However, the difficulty here lies in establishing this sense of "direction" from the source to the sink. Remember that we simply push preflow along a single edge in the graph at a time, not along a whole path. Moving flow from the source to the sink along a path that goes from the source to the sink is easy; moving flow from the source to the sink through local pushes without the knowledge of the graph structure as a whole is indeed a much harder problem.

To resolve this issue, we introduce a *label* to each of the nodes. The label $h(u)$ represents the "height" of u. In real life, water flows from higher to lower ground. We want s to represent that high ground and t to represent the low ground. As we push preflow from s to t, vertices along the way represent height values between those of s and t. However, eventually we have to push preflow back to handle both excesses in flow and suboptimal previous pushes, à la Ford-Fulkerson, but this contradicts the concept of height as we can't flow both downhill and uphill. Therefore, we'll need to be able to *relabel* a node, changing the height $h(u)$ to something that allows preflow to flow back towards s. We will relabel h in a systematic way that allows us to direct the preflow through the graph.

For this labeling to be useful for us, we'll need to impose some more constraints that must be satisfied no matter how we change the graph G_f or the height function h.

$$\begin{aligned} h(u) &\leq h(v) + 1 \forall (u, v) \in E_f \\ h(s) &= |V| \\ h(t) &= 0. \end{aligned}$$

What does this mean? For our algorithm, we can push preflow along the edge from u to v only if $c_f(u, v) > 0$ and $h(u) > h(v)$, so $h(u) = h(v) + 1$. We call such an edge $(u, v) \in E_f$ *admissible*. Furthermore, for all vertices v that can reach t in E_f, $h(v)$ represents the lower bound for the length of any unweighted path from v to t in G_f, and for all vertices that cannot reach t, then $h(v) - |V|$ is a lower bound for the unweighted distance from s to v.

t will always represent the lowest node, so $h(t) = 0$ is a natural constraint. We'll first set the preflow values of all vertices v that can be immediately reached from s to $c(s, v)$, saturating all the

out-edges of s. For any vertex v from which t can be reached, $h(v)$ represents the lower bound of the unweighted distance to t from v in the residual graph.

We want $h(s)$ to be a number large enough that will indicate that s has been disconnected to t, as we have already sent as much preflow possible from s in the direction of t by saturating all outgoing edges. Therefore, setting $h(s) = |V|$ is also natural. Since $h(s)$ represents the lower bound of the distance from s to t in G_f, and there are no paths from s to t in the residual graph, $|V|$ is a natural choice, since the longest possible path is $|V| - 1$.

Furthermore, we don't want any preflow sent back to s from a vertex v unless it is impossible to send any more preflow from v to t. If preflow is pushed from v to s, then $h(v) = |V| + 1$. If there existed a path v to t such that every edge is admissible, the path must have $|V| + 2$ vertices. This is true because for any two consecutive vertices v_i, v_{i+1} in the path, $h(v_i) = h(v_{i+1}) + 1$, but no path can have $|V| + 2$ distinct vertices.

This leads to the fact that the only nodes that can possibly continue to contribute to the final flow are active vertices v for which $h(v) < |V|$. A node with height at least $|V|$ does not have a valid path to t, and a node that is not active doesn't have any excess flow to push.

Now that I've explained the general idea behind the labeling constraints, it's time to actually describe what our relabeling process is. At first, the labels of all vertices besides the source start at 0. We only relabel a node v if it is active (therefore, it has excess flow it needs to push; $e(u) > 0$) but has no admissible out-edges in G_f (so it has no adjacent vertex on which it can push that excess flow). If a node has no admissible out-edges in G_f, every neighbor of u has a height label at least equal to $h(u)$. When we relabel a node, we always then *increase* the value of $h(u)$ to the least value where it can push flow onto another node.

function RELABEL(vertex u)
  $h(u) \leftarrow \min_{v|(u,v)\in E_f}(h(v)) + 1$ $\quad \triangleright\ (u,v) \in E_f \iff c_f(u,v) = c(u,v) - f(u,v) > 0$

Since we take the minimum height of all neighbors in the graph, we first try adjusting the height of u so that we can push flow from u to its neighbors that can possibly still reach t; that is, neighbors v satisfying $h(v) < |V|$. Once we try all these neighbors, we then increase the height of u to begin to push flow back towards s. We can always find such an edge, as any preflow pushed onto u must have also incremented the reverse edge from u back towards s.

Note that neither pushing on a admissible edge nor relabeling a vertex with no admissible out-edges changes the fact that h remains a valid labeling function.

The generic push-relabel algorithm simply pushes and relabels vertices until there are no active vertices and the preflow becomes a flow. This algorithm works because throughout the process, h remained a valid height function, and at the end, the preflow was converted into a flow. Since $h(s) = |V|$ and $h(t) = 0$, there is no augmenting path from s to t, so our flow is maximal.

Algorithm 12 Push-Relabel (Generic)

function PUSH(edge (u, v))
  if $e(u) > 0$ and $h(u) = h(v) + 1$ **then** ▷ push condition
    $\delta \leftarrow \min(e(u), c_f(u, v))$ ▷ $c_f(u, v) = c(u, v) - f(u, v)$
    $f(u, v) \leftarrow f(u, v) + \delta$
    $f(v, u) \leftarrow f(v, y) - \delta$
    $e(u) \leftarrow e(u) - \delta$
    $e(v) \leftarrow e(v) + \delta$
function RELABEL(vertex u)
  if $u \neq s, t$ and $e(u) > 0$ and $h(u) \leq h(v) \forall v | (u, v) \in E_f$ **then** ▷ relabel condition
    $h(u) \leftarrow \min_{v|(u,v)\in E_f}(h(v)) + 1$ ▷ $(u, v) \in E_f \iff c_f(u, v) = c(u, v) - f(u, v) > 0$
function MAXFLOW($G(V, E)$, $s, t \in V$)
  for all $v \in V \setminus \{s\}$ **do** ▷ initialize excess
    $e(v) \leftarrow 0$
  $e(s) \leftarrow \infty$
  for all $(u, v) \in V^2$ **do** ▷ initialize preflow
    $f(u, v) \leftarrow 0$
  for all neighbors $v \neq s$ of s **do** ▷ saturate edges to all neighbors of s
    $f(s, v) \leftarrow c(s, v)$
    $f(v, s) \leftarrow -c(s, v)$
    $e(v) \leftarrow c(s, v)$
  for all $v \in V \setminus \{s\}$ **do** ▷ preflow now has valid height function; initialize height
    $h(v) \leftarrow 0$
  $h(s) \leftarrow |V|$
  while we can still PUSH or RELABEL **do**
    PUSH or RELABEL
  return $e(t)$ ▷ $e(t) = |f|$

We can argue that this algorithm runs in $O(V^2E)$, which is already an improvement from Edmonds-Karp. However, just as Ford-Fulkerson could be sped up by specifying which augmenting paths to choose, we can do the same with the push-relabel algorithm, speeding it up by specifying a systematic method to choose an edge to push or a vertex to relabel.

Discharge

We first describe an auxiliary operation. For each vertex u, we'll need a way to visit in- and out-neighbors of u in a static cyclic order. This is easy with just a pointer; for vertex u, we'll call that pointer $curr(u)$. When the pointer passes through every element in the list of neighbors, we'll just reset it back to the first element.

```
function DISCHARGE(vertex u)
    while e(u) > 0 do                    ▷ perform an operation as long as u is active
        if curr(u) is at end of list of neighbors then
            RELABEL(u)
            reset curr(u)
        else
            if (u, curr(u)) is an admissible edge then
                PUSH((u, curr(u)))
            else
                move curr(u) to next neighbor of u
```

FIFO Selection

FIFO selection simply maintains a list of active vertices in a FIFO queue. We pop off the first vertex in the queue and discharge it, adding any newly-activated vertices to the end of the queue. This runs in $O(V^3)$.

Highest Label Selection

Highest label selection discharges the active vertex with the greatest height. This runs in $O(V^2\sqrt{E})$.

Improvements with Heuristics

Heuristics are meant to help relabel vertices in a smarter way. Bad relabelings are the slowest part of the algorithm, and improving the process can speed up max flow.

The *gap heuristic* takes advantage of "gaps" in the height function. Since a path of admissible edges consists of vertices whose heights decrease by exactly 1, the presence of a gap in height precludes the possibility of such a path. If there exists a value h' such that no vertex v exists such that $h(v) = h'$, then for every vertex v satisfying $h' < h(v) < |V|$, v has been disconnected from t, so we can immediately relabel $h(v) = |V| + 1$.

The *global relabeling heuristic* performs a backwards BFS from t every now and then to compute the heights of the vertices in the graph exactly.

Some dude on Codeforces[2] didn't have much luck improving performance with the global relabeling heuristic. I'd suggest sticking to the gap heuristic only.

9.2.5 Other Problems with Flow Solutions

Edge-Disjoint Paths

Bipartite Matching

[2] http://codeforces.com/blog/entry/14378

Chapter 10

Math

Algorithms here exhibit a different flavor than the graph theory, string, or geometry algorithms from earlier. This chapter is placed towards the end because material here doesn't really fit in any other section or the USACO canon, *not* because this material is particularly difficult.

10.1 Number Theory

In this section we explore ways for computers to quickly compute some useful mathematical quantities.

10.2 Combinatorial Games

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

10.3 Karatsuba

10.4 Matrices

10.5 Fast Fourier Transform

Chapter 11

Nonsense

Have fun.

11.1 Segment Tree Extensions

11.1.1 Fractional Cascading

11.1.2 Persistence

11.1.3 Higher Dimensions

11.2 DP Optimizations

11.3 Top Tree

Tree representation of a graph. Scary.

11.4 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 12

Problems

Problems, to be worked through as you progress through the text.

12.1 Bronze

USACO bronze is quite ad hoc. Knowing basic data structures in the standard library helps.

12.2 Silver

USACO silver tends to have the most standard problems. Silver tests knowledge of basic algorithms and coding ideas. Silver questions also sometimes require knowledge of the language, like familiarity with standard library data structures.

12.2.1 Complete Search

12.2.2 Greedy

12.2.3 Standard Dynamic Programming

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.

12.2.4 Standard Graph Theory

Flood Fill

Shortest Path

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 ($1 \leq N \leq 10,000$). There are M bi-directional roads ($1 \leq M \leq 50,000$) 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 ($1 \leq K \leq 5$) 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 ($1 \leq M \leq 500$) 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 ($1 \leq N \leq 500$), 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.

Minimal Spanning Tree

1. (USACO March 2014, irrigation) Due to a lack of rain, Farmer John wants to build an irrigation system to send water between his N fields ($1 \leq N \leq 2000$).

   Each field i is described by a distinct point (x_i, y_i) in the 2D plane, with 0 ¡= xi, yi ¡= 1000. The cost of building a water pipe between two fields i and j is equal to the squared Euclidean distance between them:

   $$(x_i - x_j)^2 + (y_i - y_j)^2$$

   FJ would like to build a minimum-cost system of pipes so that all of his fields are linked together – so that water in any field can follow a sequence of pipes to reach any other field.

   Unfortunately, the contractor who is helping FJ install his irrigation system refuses to install any pipe unless its cost (squared Euclidean length) is at least C ($1 \leq C \leq 1,000,000$).

   Please help FJ compute the minimum amount he will need pay to connect all his fields with a network of pipes.

2. (USACO February 2015, superbull) Bessie and her friends are playing hoofball in the annual Superbull championship, and Farmer John is in charge of making the tournament as exciting as possible. A total of N ($1 \leq N \leq 2000$) teams are playing in the Superbull. Each team is assigned a distinct integer team ID in the range $[1, 2^{30} - 1]$ to distinguish it from the other teams. The Superbull is an elimination tournament – after every game, Farmer John chooses which team to eliminate from the Superbull, and the eliminated team can no longer play in any more games. The Superbull ends when only one team remains.

   Farmer John notices a very unusual property about the scores in matches! In any game, the combined score of the two teams always ends up being the bitwise exclusive OR (XOR) of

the two team IDs. For example, if teams 12 and 20 were to play, then 24 points would be scored in that game, since $01100 XOR 10100 = 11000$.

Farmer John believes that the more points are scored in a game, the more exciting the game is. Because of this, he wants to choose a series of games to be played such that the total number of points scored in the Superbull is maximized. Please help Farmer John organize the matches.

3. (SPOJ, INVENT) Given tree with N ($1 \leq N \leq 15,000$) 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.

Union-Find

1. (USACO February 2013, tractor) One of Farmer John's fields is particularly hilly, and he wants to purchase a new tractor to drive around on it. The field is described by an $N \times N$ grid of non-negative integer elevations ($1 \leq N \leq 500$). A tractor capable of moving from one grid cell to an adjacent cell (one step north, east, south, or west) of height difference D costs exactly D units of money.

   FJ would like to pay enough for his tractor so that, starting from some grid cell in his field, he can successfully drive the tractor around to visit at least half the grid cells in the field (if the number of total cells in the field is odd, he wants to visit at least half the cells rounded up). Please help him compute the minimum cost necessary for buying a tractor capable of this task.

2. (CF 266E) There are n ($1 \leq n \leq 10^5$) employees working in company "X" (let's number them from 1 to n for convenience). Initially the employees didn't have any relationships among each other. On each of m ($1 \leq m \leq 10^5$) next days one of the following events took place:

   - either employee y became the boss of employee x (at that, employee x didn't have a boss before);
   - or employee x gets a packet of documents and signs them; then he gives the packet to his boss. The boss signs the documents and gives them to his boss and so on (the last person to sign the documents sends them to the archive);
   - or comes a request of type "determine whether employee x signs certain documents".

   Your task is to write a program that will, given the events, answer the queries of the described type. At that, it is guaranteed that throughout the whole working time the company didn't have cyclic dependencies.

Euler Tour

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.)

12.2.5 Easy Computational Geometry

12.3 Gold

USACO gold problems generally fall into two families; the first tests knowledge of more complex data structures not in the standard library, and the second tests cleverness with more nonstandard greedy or dynamic programming strategies.

12.3.1 More Dynamic Programming

12.3.2 Binary Search

1. (USACO March 2014, sabotage) Farmer John's arch-nemesis, Farmer Paul, has decided to sabotage Farmer John's milking equipment!

   The milking equipment consists of a row of N ($3 \leq N \leq 100,000$) milking machines, where the ith machine produces M_i units of milk ($1 \leq M_i \leq 10,000$). Farmer Paul plans to disconnect a contiguous block of these machines – from the ith machine up to the jth machine ($2 \leq i \leq j \leq N-1$); note that Farmer Paul does not want to disconnect either the first or the last machine, since this will make his plot too easy to discover. Farmer Paul's goal is to minimize the average milk production of the remaining machines. Farmer Paul plans to remove at least 1 cow, even if it would be better for him to avoid sabotage entirely.

   Fortunately, Farmer John has learned of Farmer Paul's evil plot, and he is wondering how bad his milk production will suffer if the plot succeeds. Please help Farmer John figure out the minimum average milk production of the remaining machines if Farmer Paul does succeed.

2. (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.

3. (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.

4. (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.

12.3.3 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.

12.3.4 More Standard Graph Theory

Maximum Flow and Variants

Strongly Connected Components

Other Graph Theory

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.

2. (CF 260C) Andrew plays a game called "Civilization". Dima helps him.

   The game has n ($1 \le n \le 3 \cdot 10^5$) cities and m ($0 \le m < n$) bidirectional roads. The cities are numbered from 1 to n. Between any pair of cities there either is a single (unique) path, or there is no path at all. A path is such a sequence of distinct cities v_1, v_2, ..., v_k, that there is a road between any contiguous cities v_i and v_{i+1} ($1 \le i < k$). The length of the described path equals to $k - 1$. We assume that two cities lie in the same region if and only if, there is a path connecting these two cities.

   During the game events of two types take place:

   - Andrew asks Dima about the length of the longest path in the region where city x lies.
   - Andrew asks Dima to merge the region where city x lies with the region where city y lies. If the cities lie in the same region, then no merging is needed. Otherwise, you need to merge the regions as follows: choose a city from the first region, a city from the second region and connect them by a road so as to minimize the length of the longest path in the resulting region. If there are multiple ways to do so, you are allowed to choose any of them.

   Dima finds it hard to execute Andrew's q ($1 \le q \le 3 \cdot 10^5$) queries, so he asks you to help him. Help Dima.

12.3.5 Standard Computational Geometry

12.3.6 Less Standard Problems

12.4 Beyond

12.4.1 Data Structure Nonsense

12.4.2 Other Nonsense