

Design and Analysis of Data Structures

Niema Moshiri
Liz Izhikevich

Contents

1	Introduction and Review	1
1.1	Welcome!	1
1.2	Tick Tock, Tick Tock	2
1.3	Classes of Computational Complexity	6
1.4	The Fuss of C++	12
1.5	Random Numbers	30
1.6	Bit-by-Bit	33
1.7	The Terminal-ator	38
1.8	Git, the “Undo” Button of Software Development	43
2	Introductory Data Structures	47
2.1	Array Lists	47
2.2	Linked Lists	53
2.3	Skip Lists	59
2.4	Circular Arrays	67
2.5	Abstract Data Types	75
2.6	Deques	76
2.7	Queues	79
2.8	Stacks	81
2.9	And the Iterators Gonna Iterate-ate-ate	84
3	Tree Structures	91
3.1	Lost in a Forest of Trees	91
3.2	Heaps	99
3.3	Binary Search Trees	105
3.4	BST Average-Case Time Complexity	112
3.5	Randomized Search Trees	117
3.6	AVL Trees	125
3.7	Red-Black Trees	136
3.8	B-Trees	145
3.9	B+ Trees	156

4	Introduction to Graphs	167
4.1	Introduction to Graphs	167
4.2	Graph Representations	173
4.3	Algorithms on Graphs: Breadth First Search	177
4.4	Algorithms on Graphs: Depth First Search	181
4.5	Dijkstra's Algorithm	185
4.6	Minimum Spanning Trees	190
4.7	Disjoint Sets	197
5	Hashing	209
5.1	The Unquenched Need for Speed	209
5.2	Hash Functions	210
5.3	Introduction to Hash Tables	215
5.4	Probability of Collisions	221
5.5	Collision Resolution: Open Addressing	227
5.6	Collision Resolution: Closed Addressing	236
5.7	Collision Resolution: Cuckoo Hashing	241
5.8	Hash Maps	245
6	Implementing a Lexicon	253
6.1	Creating a Lexicon	253
6.2	Using Linked Lists	254
6.3	Using Arrays	257
6.4	Using Binary Search Trees	260
6.5	Using Hash Tables and Hash Maps	262
6.6	Using Multiway Tries	266
6.7	Using Ternary Search Trees	274
7	Coding and Compression	285
7.1	Return of the (Coding) Trees	285
7.2	Entropy and Information Theory	286
7.3	Honey, I Shrunk the File	290
7.4	Bitwise I/O	293
8	Summary	301
8.1	ArrayList (Dynamic Array)	301
8.1.1	Unsorted ArrayList	301
8.1.2	Sorted ArrayList	302
8.2	Linked List	304
8.3	Skip List	306
8.4	Heap	308
8.5	Binary Search Tree	309
8.5.1	Regular Binary Search Tree	309
8.5.2	Randomized Search Tree	310
8.5.3	AVL Tree	311
8.5.4	Red-Black Tree	311

8.6	B-Tree	313
8.7	B+ Tree	315
8.8	Hashing	317
8.8.1	Linear Probing	318
8.8.2	Double Hashing	318
8.8.3	Random Hashing	319
8.8.4	Separate Chaining	320
8.8.5	Cuckoo Hashing	320
8.9	Multiway Trie	321
8.10	Ternary Search Tree	323
8.11	Disjoint Set (Up-Tree)	324
8.12	Graph Representation	326
8.12.1	Adjacency Matrix	326
8.12.2	Adjacency List	326
8.13	Graph Traversal	328
8.13.1	Breadth First Search (BFS)	328
8.13.2	Depth First Search (DFS)	328
8.13.3	Dijkstra's Algorithm	329
8.14	Minimum Spanning Tree	330
8.14.1	Prim's Algorithm	330
8.14.2	Kruskal's Algorithm	330

Chapter 1

Introduction and Review

1.1 Welcome!

This is a textbook companion to the Massive Open Online Course (MOOC), *Data Structures: An Active Learning Approach*, which has various activities embedded throughout to help stimulate your learning and improve your understanding of the materials we will cover. While this textbook contains all **STOP and Think** questions, which will help you reflect on the material, and all **Exercise Breaks**, which will test your knowledge and understanding of the concepts discussed, we recommend utilizing the MOOC for all **Code Challenges**, which will allow you to actually implement some of the algorithms we will cover.

About the Authors

Niema Moshiri (B.S.'15 UC San Diego) is a Ph.D. candidate in the Bioinformatics and Systems Biology (BISB) program at the University of California, San Diego. His educational background is in Bioinformatics, which spans Computer Science (with a focus on Algorithm Design) and Biology (with a focus on Molecular Biology). He is co-advised by Siavash Mirarab and Pavel Pevzner, with whom he performs research in Computational Biology with a focus on HIV phylogenetics and epidemiology.

Liz Izhikevich (B.S.'17 UC San Diego, M.S.'18 UC San Diego) is an NSF and Stanford Graduate Fellow in the Computer Science Department at Stanford University. She has spent years working with and developing the content of multiple advanced data structures courses. Her research interests lie at the intersection of serverless distributed systems and security.

Acknowledgements

We would like to thank Christine Alvarado for providing the motivation for the creation of this textbook by incorporating it into her Advanced Data Structures course at UC San Diego, as well as for reviewing the content and providing

useful feedback for revisions. We would also like to thank Paul Kube, whose detailed lecture slides served as a great reference for many of the topics discussed (especially the lesson walking through the proof for the Binary Search Tree average-case time complexity). Lastly, we would like to thank Phillip Compeau and Pavel Pevzner, whose textbook (*Bioinformatics Algorithms*) served as inspiration for writing style and instructional approach.

1.2 Tick Tock, Tick Tock

During World War II, Cambridge mathematics alumnus Alan Turing, the “Father of Computer Science,” was recruited by British intelligence to crack the German naval Enigma, a rotor cipher machine that encrypted messages in a way that was previously impossible to decrypt by the Allied forces. In 2014, a film titled *The Imitation Game* was released to pay homage to Turing and his critical work in the field of cryptography during World War II. In the film, Turing’s leader, Commander Denniston, angrily critiques the Enigma-decrypting machine for not working, to which Turing replies:

“It works... It was just... Still working.”

In the world of mathematics, a solution is typically either correct or incorrect, and all correct solutions are typically equally “good.” In the world of algorithms, however, this is certainly not the case: even if two algorithms produce identical solutions for a given problem, there is the possibility that one algorithm is “better” than the other. In the aforementioned film, the initial version of Turing’s Enigma-cracking machine would indeed produce the correct solution if given enough time to run, but because the Nazis would change the encryption cipher every 24 hours, the machine would only be useful if it were able to complete the decryption process far quicker than this. However, at the end of the movie, Turing updates the machine to omit possible solutions based on knowledge of the unchanging greeting appearing at the end of each message. After this update, the machine is able to successfully find a solution fast enough to be useful to the British intelligence agency.

What Turing experienced can be described by **time complexity**, a way of quantifying the execution time of an algorithm as a function of its input size by effectively counting the number of elementary operations performed by the algorithm. Essentially, the time complexity of a given algorithm tells us roughly how fast the algorithm performs as the input size grows.

There are three main notations to describe time complexity: **Big- \mathcal{O}** (“Big- \mathcal{O} ”), **Big- Ω** (“Big- $\mathcal{O}\!\!\!\Omega$ ”), and **Big- Θ** (“Big- Θ ”). **Big- \mathcal{O}** notation provides an *upper-bound* on the number of operations performed, **Big- Ω** provides a *lower-bound*, and **Big- Θ** provides both an *upper-bound and a lower-bound*. In other words, **Big- Θ** implies *both* **Big- \mathcal{O}** and **Big- Ω** . For our purposes, we will only consider **Big- \mathcal{O}** notation because, in general, we only care about upper-bounds on our algorithms.

Let $f(n)$ and $g(n)$ be two functions defined for the real numbers. One would write

$$f(n) = \mathcal{O}(g(n))$$

if and only if there exists some constant c and some real n_0 such that the absolute value of $f(n)$ is at most c multiplied by the absolute value of $g(n)$ for all values of n greater than or equal to n_0 . In other words, such that

$$|f(n)| \leq c |g(n)| \text{ for all } n \geq n_0.$$

Also, note that, for all three notations, we drop all lower functions when we write out the complexity. Mathematically, say we have two functions $g(n)$ and $h(n)$, where

$$|g(n)| \geq |h(n)| \text{ for all } n \geq n_0.$$

If we have a function $f(n)$ that is $\mathcal{O}(g(n) + h(n))$, we would just write that $f(n)$ is $\mathcal{O}(g(n))$ to simplify. Note that we only care about time complexities with regard to large values of n , so when you think of algorithm performance in terms of any of these three notations, you should really only think about how the algorithm scales as n approaches infinity.

For example, say we have a list of n numbers and, for each number, we want to print out the number and its opposite. Assuming the “print” operation is a single operation and the “opposite” operation is also a single operation, we will be performing 3 operations for each number (print itself, return the opposite of it, and print the opposite). If we have n elements and we will be performing exactly 3 operations on each element, we will be performing exactly $3n$ operations. Therefore, we would say that this algorithm is $\mathcal{O}(n)$, because there exists some constant (3) for which the number of operations performed by our algorithm is always bounded by the function $g(n) = n$.

Note that, although this algorithm is $\mathcal{O}(n)$, it is also technically $\mathcal{O}(n^2)$ and any other larger function of n , because technically, these are all functions that are upper-bounds on the function $f(n) = 3n$. However, when we describe algorithms, we always want to describe them using the tightest possible upper-bound.

Here’s another example: say I have an algorithm that, given a list of n students, prints out 5 header lines (independent of n) and then prints the n students’ names and grades on separate lines. This algorithm will perform $2n + 5$ operations total: 2 operations per student (print name, print grade) and 5 operations independent of the number of students (print 5 header lines). The time complexity of this algorithm in Big- \mathcal{O} notation would be $\mathcal{O}(2n + 5)$, which we would simplify to $\mathcal{O}(n)$ because we drop the constant ($2n$ becomes n) and drop all lower terms ($5 < n$ as n becomes large).

Exercise Break

What is the tightest upper-bound for the time complexity of the following segment of C++ code (in Big- \mathcal{O} notation)?

```

1 void print_info(vector<int> a) {
2     int n = a.size(); float avg = 0.0;
3     for(int i = 0; i < n; i++)
4         avg += a[i];
5     avg /= n;
6 }
```

The following is an equivalent Python implementation (where **a** is a list of numbers):

```

1 def print_info(a):
2     n = len(a); avg = 0.0
3     for i in range(n):
4         avg += a[i]
5     avg /= n
```

Exercise Break

What is the tightest upper-bound for the time complexity of the following segment of C++ code (in Big- \mathcal{O} notation)?

```

1 void dist(vector<int> a) {
2     int n = a.size();
3     for(int i = 0; i < n-1; i++)
4         for(int j = i+1; j < n; j++)
5             cout << a[j] - a[i] << endl;
```

The following is an equivalent Python implementation (where **a** is a list of numbers):

```

1 def dist(a):
2     n = len(a)
3     for i in range(n-1):
4         for j in range(i+1, n):
5             print(a[j] - a[i])
```

Exercise Break

What is the tightest upper-bound for the time complexity of the following segment of C++ code (in Big- \mathcal{O} notation)?

```

1 void tricky( int n ) {
2     int op = 0;
3     while( n > 0 ) {
4         for( int i = 0; i < n; i++ )
5             op++;
6         n /= 2;
7     }
8 }
```

The following is an equivalent Python implementation (where n is an integer):

```

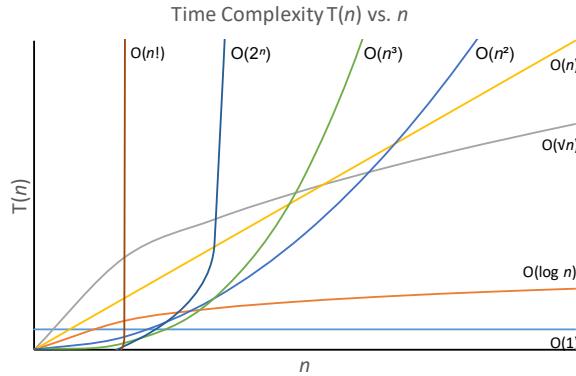
1 def tricky( n ):
2     op = 0
3     while n > 0:
4         for i in range(n):
5             op += 1
6         n = int(n/2)
```

Hint: Does the sum representing the number of operations performed converge to some function of n ?

When we discuss algorithms colloquially, we don't always explicitly say "Big- \mathcal{O} of n " or "Big- \mathcal{O} of n -cubed," etc. Instead, we typically use more colloquial terms to refer to Big- \mathcal{O} time complexities. The following are some of the most common of these colloquial terms (along with their corresponding Big- \mathcal{O} time complexities):

Colloquial Term	Big- \mathcal{O} Complexity
Constant Time	$\mathcal{O}(1)$
Logarithmic Time	$\mathcal{O}(\log n)$
Polynomial Time	$\mathcal{O}(n^k)$ (for any constant k)
Exponential Time	$\mathcal{O}(k^n)$ (for any constant k)
Factorial Time	$\mathcal{O}(n!)$

The following is a plot of various common time complexities.



A **data structure**, as implied by the name, is a particular structured way of storing data in a computer so that it can be used efficiently. Throughout this text, we will discuss various important data structures, both simple and complex, that are used by Computer Scientists every day. Specifically, we will cover what these data structures are, how they function (i.e., the algorithms that will be running behind-the-scenes for insertions, deletions, and look-ups), and when to appropriately use each one.

When dealing with data storage, in addition to worrying about time complexity, we will also be worrying about **space complexity**, which measures the amount of working storage needed for an input of size n . Just like time complexity, space complexity is described using Big- \mathcal{O} notation.

For example, say we have n cities and we want to store information about the amount of gas and time needed to go from city i to city j (which is not necessarily equivalent to the amount of gas and time needed to go from city j to city i). For a given city i , for each of the n cities, we need to represent 2 numbers: the amount of gas needed to go from city i to each of the n cities (including itself), and the amount of time needed to go from city i to each of the n cities (including itself). Therefore, we need to store $n \times n \times 2 = 2n^2$ numbers. Therefore, the space complexity is $\mathcal{O}(n^2)$.

The lasting goal of this text is to provide you with a solid and diverse mental toolbox of data structures so that, when you work on your own projects in the future, you will be able to recognize the appropriate data structure to use. Much like how a brain surgeon probably shouldn't use a butter knife to perform a surgery (even though it could technically work), you should always use the appropriate data structure, even if a less appropriate one would "technically work." In the following chapters, you will begin learning about the data structures covered in this text.

1.3 Classes of Computational Complexity

Now that we have the ability to determine algorithm time complexity in our arsenal of skills, we have found a way to take an arbitrary algorithm and describe its performance as we scale its input size. Note that, however, algorithms are simply *solutions to computational problems*. A single computational problem can have numerous algorithms as solutions. As a simple example, say our computational problem is the following: "Given a vector containing n integers, return the largest element." The following is one algorithmic solution:

```

1 int getMax1( vector<int> vec) {
2     int max = vec[0];
3     for( int i : vec)
4         if( i > max)
5             max = i ;
6     return max;
7 }
```

The following is an equivalent Python implementation:

```

1 def getMax1(vec):
2     m = vec[0]
3     for i in vec:
4         if i > m:
5             m = i
6     return m

```

The following is a second algorithmic solution that is equally correct, but that is clearly less efficient (and more convoluted):

```

1 int getMax2(vector<int> vec) {
2     for(int i : vec) {
3         bool best = true;
4         for(int j : vec)
5             if(i < j)
6                 best = False;
7         if(best)
8             return i;
9     }
10 }

```

The following is an equivalent Python implementation:

```

1 def getMax2(vec):
2     for i in vec:
3         best = True
4         for j in vec:
5             if i < j:
6                 best = False
7         if best:
8             return i

```

As we can deduce, the first algorithm has a time complexity of $\mathcal{O}(n)$, whereas the second algorithm has a time complexity of $\mathcal{O}(n^2)$. We were able to describe the two *algorithms* (`getMax1` and `getMax2`), but is there some way we can describe the *computational problem* itself? In this section, we will be discussing the classes of computational problems: **P**, **NP**, **NP-Hard**, and **NP-Complete**.

First, we will discuss the **P** class of computational problems. Any computational problem that can be *solved* in *polynomial* time (or better, of course) is considered a member of **P**. In other words, given a computational problem, if there exists a polynomial time algorithm that solves the problem—where we would need to formally prove that the algorithm does indeed always provide an optimal solution to the problem—we would classify the problem as class **P**.

For example, let's define the “Oldest Person Problem” as follows:

- **Input:** A list of people *population*, represented as (Name, Age) pairs
- **Output:** The name of the oldest person in *population* (if there is a tie, return the names of all such people)

Somewhat intuitively, we can come up with the following algorithm to solve the “Oldest Person Problem”:

```

1 OldestPerson( population ):
2     oldestAge = -1
3     oldestNames = empty list of strings
4     for each person in population:
5         if person.age > oldestAge:
6             oldestAge = person.age
7             clear oldestNames
8             add person.name to oldestNames
9         else if person.age == oldestAge:
10            add person.name to oldestNames
11
12     return oldestNames

```

As you hopefully inferred, if *population* has n individuals, this algorithm has a worst-case time complexity of $\mathcal{O}(n)$, which is polynomial time. Therefore, since we have found a polynomial-time solution to the “Oldest Person Problem,” we can say that the “Oldest Person Problem” belongs in problem class **P**.

Next, we will discuss the **NP** (Nondeterministic Polynomial time) class of computational problems. Any computational problem where, given a proposed answer to the problem, one can *verify* the answer for correctness in polynomial time is considered a member of **NP**. Note, however, that although we need to be able to *verify* an answer in polynomial time, we do not necessarily need to be able to *compute* a correct answer in polynomial time (i.e., it is not necessary for there to exist a polynomial-time algorithm that solves the problem optimally).

We mentioned previously that **P** is the class of computational problems that can be *solved* in polynomial time. If **NP** is the class of computational problems that can be *verified* in polynomial time (whether or not you can solve them in polynomial time), it should be clear that **P** is a subset of **NP**: if we can *solve* a problem in polynomial time, then we can certainly *verify* a proposed answer to the problem in polynomial time (we can just solve it in polynomial time and then compare the proposed answer to our answer).

Let’s discuss a problem that is a member of **NP** (i.e., a proposed answer can be *verified* in polynomial time) but is not a member of **P** (i.e., no polynomial-time solution exists at the moment). The “Subset Sum Problem” is defined as follows:

- **Input:** A set of integers *set*
- **Output:** A non-empty subset of *set* whose sum is 0

You can try to think of a polynomial-time algorithm that solves the “Subset Sum Problem,” but we can assure that your efforts will likely be futile (because if you *did* find such an algorithm, you would have solved one of the *Millennium Prize Problems* and would receive a \$1,000,000 prize). However, if we were to give you a set of integers *set* and a proposed answer *subset*, you would be able to check the correctness of *subset* in polynomial time:

```

1 CheckSubset(set , subset):
2     // verify subset is valid non-empty subset of set
3     if subset is empty:
4         return False
5     for each element of subset:
6         if element does not appear in set:
7             return False
8
9     // verify that elements of subset add up to 0
10    sum = 0
11    for each element of subset:
12        sum = sum + element
13    if sum == 0:
14        return True
15    else:
16        return False

```

Exercise Break

Which of the following statements are true?

- All problems in **P** can be **verified** in polynomial time
- All problems in **P** can be **solved** in polynomial time
- All problems in **NP** can be **verified** in polynomial time
- All problems in **NP** can be **solved** in polynomial time

The third class of computational problems we will discuss is **NP-Hard** (Nondeterministic Polynomial-time **hard**). Describing **NP-Hard** problems is a bit trickier because we want to avoid going too deep in the topic; but basically, a problem can be considered **NP-Hard** if it is at *least* as hard as the hardest problems in **NP**. More precisely, a problem H is **NP-Hard** when every problem L in **NP** can be “reduced,” or transformed, to problem H in polynomial time. As a result, if someone were to find a polynomial-time algorithm to solve any **NP-Hard** problem, this would give polynomial-time algorithms for all problems in **NP**. The “Subset Sum Problem” described previously is an example of an **NP-Hard** problem. There’s no “clean” way to come to this realization on your own without taking a complexity theory course, so take our word that it is (an explanation would go well out of the scope of this text).

The last class of computational problems we will discuss is **NP-Complete**, which is simply the intersection between **NP** and **NP-Hard**. In other words, an **NP-Hard** problem is considered **NP-Complete** if it can be verified in polynomial time (i.e., it is also in **NP**). One interesting **NP-Complete** problem is the “Boolean Satisfiability Problem,” which is the basis of modern encryption. When we encrypt sensitive data, we feel safe because, without knowing the encryption key, a malicious person would need to solve the “Boolean Satisfiability

Problem” to be able to forcefully decrypt our data, which is unfeasible because of the hardness of the problem.

STOP and Think

The “Boolean Satisfiability Problem” is the basis of modern encryption, and it asks the following question: “Given some arbitrary Boolean formula, does there exist a set of values for the variables in the formula such that the formula is satisfied?” For example, if we were given the Boolean formula $x \text{ AND NOT } y$, does there exist a set of values for x and y that satisfies the formula? In this small example, we can see there does indeed a solution that satisfies this formula ($x = \text{TRUE}$ and $y = \text{FALSE}$), but is there way of algorithmically determining if a solution exists for some arbitrary Boolean formula?

The “Boolean Satisfiability Problem” is **NP-Complete**, meaning it is both **NP-Hard** and **NP**. If someone were to find a polynomial-time solution to any **NP-Hard** problem (not necessarily the “Boolean Satisfiability Problem”), what would be the repercussions to data encryption, if any?

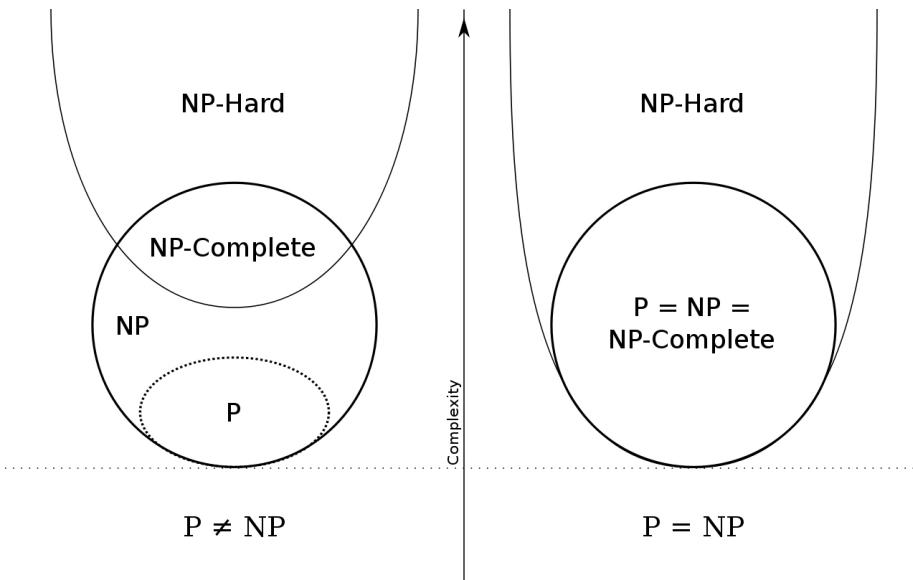
Exercise Break

Which of the following statements are true?

- All problems in **NP-Complete** can be **verified** in polynomial time
- All problems in **NP-Complete** can be **solved** in polynomial time
- All problems in **NP-Hard** can be **verified** in polynomial time
- All problems in **NP-Hard** can be **solved** in polynomial time

When we introduced **P** and **NP**, we mentioned that **P** is a subset of **NP**. What if, however, **P** wasn’t just a subset of **NP**? What if **P** and **NP** were actually equal sets? This question of whether or not **P** equals **NP** is known as the “**P** vs. **NP** Problem,” which is a major unsolved problem in the field of computer science. Informally, it asks whether every problem whose solution can be quickly verified by a computer can also be quickly solved by a computer.

Formally, if it can be mathematically proven that **P** = **NP**, then it would imply *all* problems that can be verified in polynomial time can also be solved in polynomial time. Thus, if there is no known algorithmic solution to a problem that can be verified in polynomial time, we can conclude that a solution *must* exist, and humans just have not found the solution yet. On the other side, if it can be mathematically proven that **P** ≠ **NP**, then it would imply that there indeed exists a set of problems that can be verified in polynomial time but that cannot be solved in polynomial time. If we run into a problem in **NP** for which we cannot find a polynomial time solution, it very well could be that no such solution exists. The following is a diagram representing the relationship between **P**, **NP**, **NP-Hard**, and **NP-Complete** under both possible scenarios regarding the equality (or lack thereof) of **P** and **NP**.



Exercise Break

Which of the following statements are known to be true?

- All problems in P are also in NP
- All problems in NP are also in P
- All problems in $NP\text{-Complete}$ are also in NP
- All problems in NP are also in $NP\text{-Complete}$
- All problems in $NP\text{-Complete}$ are also in $NP\text{-Hard}$
- All problems in $NP\text{-Hard}$ are also in $NP\text{-Complete}$
- $P = NP$
- $P \neq NP$

As computer scientists, our entire lives are focused on solving computational problems. Even if you choose to tackle problems in other fields (e.g. Biology, Economics, Neuroscience, etc.), you will take the original non-computational problem and model it as a formal computational problem formulation. Your goal will then be to solve the computational problem, and the solution you receive will help give you insight about the original non-computational problem.

As such, it is important to be aware of how to classify computational problems, and understanding the different classes of computational problems will help you predict how fruitful it would be to try to solve the problems you encounter. For example, say you work as an analyst for a sales company, and your

boss hands you a list of cities and the distances between each pair of cities, and he asks you to find a route that visits each city exactly once and returns to the origin city, but such that the route's length is less than a given distance. You might spend hours, days, or even years trying to find a polynomial-time solution to the problem, but had you taken a step back and thought about the problem, you would have realized that this problem is exactly the decision version of the “Traveling Salesman Problem,” which is **NP-Complete**. Thus, you would have immediately realized that, although possible, it is extremely unlikely that you would be able to find a polynomial-time solution to the problem because of its **NP-Complete** status.

In general, when you face a computational problem, if you can deduce that the problem is not in class **P** (and you are unable to simplify the problem to make it part of class **P**), you typically are forced to choose between one of two options:

- If the input size you are dealing with is small enough, a non-polynomial-time solution may work fine for you
- If the input size is too large for a non-polynomial-time solution, you can try to create a polynomial-time “heuristic” (i.e., an algorithm that isn’t guaranteed to give the globally optimal solution, but that does a pretty good job coming close to optimality, hopefully “good enough” for your purposes)

We hope that, now that you have learned about the classes of computational problems, you will be able to think about the problem before blindly jumping in to try to solve it to hopefully save yourself some time and effort.

1.4 The Fuss of C++

In his book titled *C++ for Java Programmers*, Mark Allen Weiss describes the different mentality between C++ and Java:

“C++’s main priority is getting correct programs to run as fast as it can; incorrect programs are on their own. Java’s main priority is not allowing incorrect programs to run; hopefully correct programs run reasonably fast, and the language makes it easier to generate correct programs by restricting some bad programming constructs.”

As Weiss hints at, if you are transitioning from Java to C++, you must beware that C++ is not designed to help you write correct code. The safety checks built into Java slow down the execution of your code, but the lack of these safety checks in C++ means that, even if your code’s logic is incorrect, you may still be allowed to execute it which can make debugging tricky.

STOP and Think

Based on the descriptions above, why would we use C++ instead of Java to implement data structures in this course?

In this chapter we assume you are familiar with programming in Java and C. The purpose of this chapter is to provide you with a very basic introduction to C++. As mentioned before, since we are learning about data structures and thus want to place an emphasis on speed, everything in this text will be done in C++. If you are already comfortable with C++, you don't need much preparation, if any, language-wise. However, if you are *not* already comfortable with C++, this text assumes that you are at least comfortable with Java. As such, this section highlights the key differences one would need to know to transition from Java to C++.

We will first discuss the differences between **data types** in C++ and Java, which are actually quite similar.

Number Representation Like Java, C++ has the `int` and `double` types. However, with `ints`, the number of bytes an `int` variable takes in C++ depends on the machine, whereas in Java, an `int` variable takes exactly 4 bytes, no matter what hardware is used. For the purposes of this class, however, this difference is negligible, as most modern machines use 4-byte `ints`.

```
1 int a; // this variable can range from -2^31 to +2^31 - 1
2 long b; // this variable can range from -2^63 to +2^63 - 1
3 char c; // this variable can range from -2^7 to +2^7 - 1
```

Unsigned Data Types In C++, you can specify unsigned data types, whereas Java does not allow unsigned types. Recall that the first bit of an `int`, `long`, `double`, or even `char` is the “sign bit”: it represents if the stored value is positive or negative. This is why, in both languages, a regular `int` (which contains 4 bytes, or $4 \times 8 = 32$ bits) ranges from -2^{31} to $+2^{31} - 1$ (31 bits to represent the magnitude of the number, and 1 bit to represent the sign). In C++, if you specify a variable to be an `unsigned int` instead of a regular `int`, its value will be able to range from 0 to $+2^{32} - 1$.

```
1 unsigned int a; // this variable can range from 0 to +2^32 - 1
2 unsigned long b; // this variable can range from 0 to +2^64 - 1
3 unsigned char c; // this variable can range from 0 to +2^8 - 1
```

Booleans The equivalent of the Java `boolean` data type is simply `bool`. The usage of a C++ `bool` is the same as the usage of a Java `boolean`.

```
1 bool havingFun = true;
```

Strings In C++, the `string` type is very similar to Java’s `String` class. However, there are a few key differences. First, Java `Strings` are immutable. On the other hand, one can modify C++ `strings`. Next, the substring command in C++ is `substr`, where `s.substr(i,n)` returns the substring of `s` that starts at position `i` and is of length `n`. Also, in Java, `String` objects can be concatenated with any other object (and the other non-`String` object will automatically be converted to a `String`), but in C++, `string` objects can only be concatenated with other `string` objects (the conversion is not automatically done for you).

```
1 string message = "Hi, Niema!";
2 string name = message.substr(4,5); // name == "Niema"
```

Comparing Objects To compare objects in C++, you simply use the relational operators `== != < <= > >=`. In Java, you only use the relational operators to compare primitives and you use `.equals()` to compare objects. But in C++, you can do something special. You can “overload” the relational operators, meaning you can specify custom methods that get called when the relational operators are applied to objects. Thus in C++, you use relational operators to compare *everything* (including non-primitives). We’ll talk about operator-overloading in more detail later.

```
1 string name1 = "Niema";
2 string name2 = "Liz";
3 bool sameName = (name1 == name2); // sameName is 0, or false
```

Next, we will discuss the differences between **variables** in Java and C++. Note that variables in C++ can actually be much more complex than what we cover here, but the more complicated details are out of the scope of this text. This is just to teach you the basics.

Variable Safety In Java, if you declare a local variable but do not initialize it before you try to use it, the compiler will throw an error. However, in C++, if you declare a local variable but do not initialize it before you try to use it, the compiler will NOT throw an error! It will simply use whatever random “garbage data” happened to be at that location in memory! This can cause a LOT of headache when debugging your code, so always remember to initialize your variables in C++! Also, in Java, if you attempt to “demote” a variable (i.e., store it in a smaller datatype) without explicitly casting it as the lower datatype, the compiler will throw an error. However, in C++, the compiler will not complain! The following is perfectly valid C++ code in which we do both: we declare two variables and use them without initializing them, and we then demote to a smaller datatype without explicitly typecasting.

```

1 int harry; // dummy variable 1
2 int lloyd; // dummy variable 2
3 bool dumbAndDumber = (harry + lloyd); // C++ will allow this!

```

Global Variables In Java, all variables must be declared either within a class or within a method. In C++, however, variables can be declared outside of functions and classes. These “global variables” can be accessed from any function in a program, which makes them difficult to manage, so try to avoid using them.

```

1 bool dummy = true;
2 class DummyClass {
3     // some stuff here
4 };
5 int main() {
6     cout << dummy; // this is valid
7 }

```

Constant Variables In Java, a variable can be made so that it cannot be reassigned via the `final` keyword. In C++, the equivalent keyword is `const`, though there are subtle differences. In Java, `final` simply prevents the variable from being reassigned. If the data itself is mutable, it can still be changed. In C++, `const` will prevent that data from being changed. The C++ `const` keyword can be a bit tricky, and we’ll discuss it in more detail shortly.

```

1 int main() {
2     const string S = "Harry"; // S cannot be reassigned
3                             // "Harry" cannot be modified
4     S = "Barry"; // This is not allowed!
5                             // But if S were not const it would be OK
6     S[0] = 'L'; // This is not allowed!
7                             // But if S were not const it would be OK
8 }

```

We will now discuss the differences between classes in Java and C++. To exemplify the differences, we will write a `Student` class in both Java and C++:

```

1 class Student { // Java
2     public static int numStudents = 0;      // declare + define
3     private String name;                  // declare
4     public Student(String n) { /*CODE*/ } // declare + define
5     public void setName(String n) { /*CODE*/ } // declare + define
6     public String getName() { /*CODE*/ } // declare + define
7 }

```

```

1 class Student { // C++
2     public:
3         static int numStudents;           // declare
4         Student(string n);            // declare
5         void setName(string n);        // declare
6         string getName() const;       // declare
7
8     private:
9         string name;                  // declare
10    };
11 int Student::numStudents = 0;          // define
12 Student::Student(string n) { /*CODE*/ } // define
13 void Student::setName(string n) { /*CODE*/ } // define
14 string Student::getName() const { /*CODE*/ } // define

```

- In Java, each individual item must be declared as `public` or `private`, but in C++, we have `public` and `private sections`, started by the keywords `public` and `private`, respectively.
- In Java, we actually fill out the methods within the class, but in C++, we only declare the methods, and the actual implementations are listed separately outside of the class (prefixed by the class name, with the `::` operator separating the class name and the method name)
- In C++, accessor (or “getter”) methods are tagged with the keyword `const`, which prevents the method from modifying instance variables
- In C++, there is a semicolon after the class’s closing bracket

Also, note that, in C++, for the sake of convenience, you can initialize non-static variables of the object using a **member initializer list** when you define the constructor. Note that, because static variables cannot be initialized in constructors in C++, you can *only* use the member initializer list to initialize non-static variables of an object. Also note that any instance variables that are declared `const` can be initialized *only* in an initializer list; if you try to set their value in the body of the constructor, the compiler will complain. The following is an example of the syntax for the same `Student` class described above (but with the setter and getter methods omitted for the sake of simplification):

```

1 class Student { // C++
2     public:
3         static int numStudents; // declare static var
4         Student(string n);   // declare constructor
5     private:
6         string name;         // declare instance var
7     };
8 int Student::numStudents = 0; // define static var
9 // define constructor using member initializer list
10 Student::Student(string n) : name(n) {
11     numStudents++;
12 }

```

To finish off the comparison of classes in C++ versus in Java, we want to introduce the notion of .cpp and .h files in C++. In Java, you write *all* of your code (declarations *and* definitions) in a .java file, which must have a class defined (where the class must have the same name as the filename of the .java file, minus the file extension).

In C++, however, you can have your code split between a .cpp file (known as a “source file,” and sometimes using the extension .cc) and a .h file (known as a “header file”). In the .h file, you will *declare* your classes and functions, but you will not actually define them. Then, in the .cpp file, you will actually fill in the bodies of your functions.

It might seem inconvenient to have a single series of logic split between two files, but the reason behind this is to be able to distribute your header file freely so that people can have a map of how to use your product (i.e., they will have access to the declarations of all of your classes and functions, so they will know *how* they would be able to use your full code) without any fear of anyone stealing your implementation (because all of your actual code is in the source .cpp file). It also makes the compilation process simpler and faster.

The following is an example of how we would split up the previously described Student class into a .h and .cpp file:

```

1 // Student.h
2 class Student {
3     public:
4         static int numStudents; // declare static var
5         Student(string n); // declare constructor
6     private:
7         string name; // declare instance var
8 }

1 // Student.cpp
2 int Student::numStudents = 0; // define static var
3 // define constructor using member initializer list
4 Student::Student(string n) : name(n) {
5     numStudents++;
6 }
```

Next, we are going to discuss three important, and related, concepts—**references**, **pointers**, and **memory management**—and how each concept is used (or not used) in Java and C++.

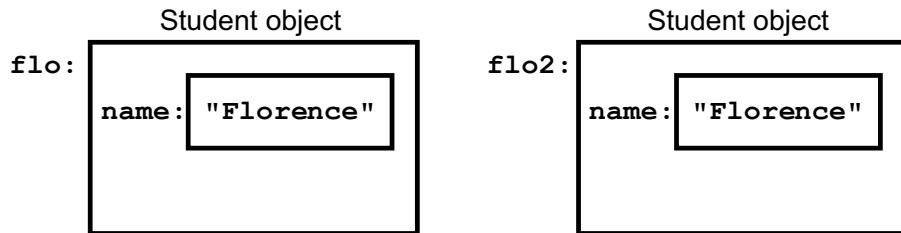
Objects vs. Primitives in Java In Java, the rules for what is stored in a variable are simple: all object variables store *object references* while primitive type variables store values directly. In Java, assignment is done by value, so when you assign a primitive variable’s value to another variable, the actual value is copied; when it is an object variable, the reference is copied and you get two references to the same object.

Objects and Primitives in C++ In C++, on the other hand, there is no such distinction between object and primitive variables. By default, *all* variables, both primitives and objects, actually hold *values*, NOT object references. C++, like Java, does assignment by value. However, this means that when one object variable is assigned to another, a copy of the entire object is made (like calling `clone` in Java). Let's look at an example:

```

1 // Creates Student object with name "Florence"
2 // Stores as variable 'flo'
3 // Note we do NOT use the keyword 'new' to create the object
4 Student flo("Florence");
5
6 // flo2 stores a copy of the Student, with the same name
7 Student flo2 = flo;

```



As illustrated above, by default, C++ stores variable values directly, no matter their type, and assignments are done by copying the data. This includes passing parameters to a function. But what if you want to have two ways to access the same data and avoid this copying-the-data behavior? C++ provides two different concepts that give you related but subtly different ways to do this: **references** and **pointers**.

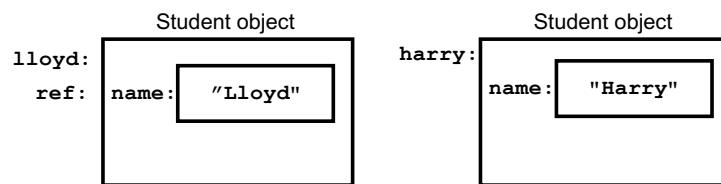
References in C++ C++ references are NOT the same as Java references. Although they are related, how they are used and their syntax is pretty different, so it is best if you simply think of them as different concepts. References in C++ are simply aliases for existing variables. When you define a reference variable in C++, the variable is treated exactly as another name as the variable you set it to. Thus, when you modify the reference variable, you modify the original variable as well without needing to do anything special. The syntax for creating a reference variable in C++ is to place an & after the type name in the variable declaration. If T is some type, then T& is the syntax to declare a reference to a T variable. Reference declarations must be combined with assignments except in the case of function parameters (discussed further shortly). Let's look at an example:

```

1 // creates Student object with name "Lloyd"
2 // stores as variable 'lloyd'
3 Student lloyd("Lloyd");
4
5 // creates reference to 'lloyd' called 'ref'
6 Student & ref = lloyd;
7
8 // creates Student object with name "Harry"
9 // stores as variable 'harry'
10 Student harry("Harry");

```

The following picture shows what the objects and variables look like so far:

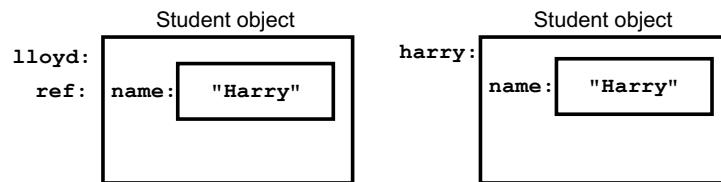


Now, we can execute the following lines:

```

1 // 'ref' is now a copy of 'harry',
2 // so 'lloyd' is ALSO now a copy of 'harry',
3 ref = harry;
4
5 // this would print "Harry"
6 cout << lloyd.getName();

```



Note that the whole `Student` object was copied, and it replaced the `Student` object that was formerly stored in `ref/lloyd`.

There are two main uses for C++ references: parameter passing and aliasing long variable names. In many cases, it's extremely useful not to make copies of objects when they are passed to functions, either because you want the function to be able to modify the data in the object, or because you want to avoid wasting time and space with the copy.

Note: This is an overly simple explanation of references. In particular, C++11 has a notion of lvalue references (which are what we showed above) and rvalue references. Don't worry about these subtleties if you don't want to. Throughout this book we'll use only "classic" (lvalue) references like what we described above. Though, if you want to learn more, there are certainly plenty of websites that would be happy to explain this in more detail!

Pointers in C++ Pointers in C++ are actually quite similar to references in Java. They are variables that store the *memory address* of some data, as opposed to storing the data directly. That is, their *value* is a memory address.

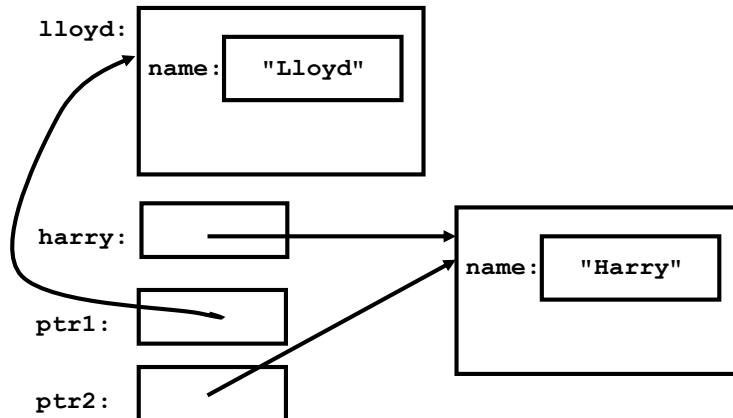
If T is some type, then T* is the syntax to declare a pointer to a T variable. A pointer variable can be initialized either with NULL, with the value of another pointer variable, with the memory address of another variable, or with a call to new (which will be discussed shortly). The memory address of a variable can be attained by placing the & symbol before the variable name. To access the object to which a pointer points to, you must “dereference” the pointer by placing the * symbol before the variable name. To access an instance variable of the object to which a pointer points, you can either dereference the pointer and access the instance variable using the . symbol as normal, or you can use the -> operator on the pointer directly (which is the more conventional way).

```

1 Student lloyd("Lloyd"); // Student object
2 Student* harry = new Student("Harry"); // Student pointer
3 Student* ptr1 = &lloyd; // ptr1 stores address of lloyd
4 // Student pointer pointing to same object as harry
5 Student* ptr2 = harry;
6 cout << (*ptr1).getName(); // prints "Lloyd"
7 cout << ptr2->getName(); // prints "Harry"

```

The memory diagram for the code above is shown. Note that arrows are used to denote memory addresses in the diagram:



As we can see, each box represents a memory location with some data stored in it. Some of the locations have labels, but the **Student** object named Harry does not have a label directly. It can only be accessed via the pointer stored in either harry or ptr2. The data stored in each box depends on the type of the variable. lloyd stores a **Student** object, while harry, ptr1, and ptr2 all store memory addresses.

Beware of the nomenclature used with pointers. The nomenclature goes as

follows: we can either say that a pointer *points* to an object, or we can say that a pointer *stores the address* of an object. For example, in the code above, look at the line where we initialize `ptr2`. After doing the assignment `ptr2 = harry`, we can either say “`ptr2` *points* to the `Student` object named Harry,” or we can say “`ptr2` *stores the address* of the `Student` object named Harry.” However, it would be inaccurate to say that “`ptr2` *points to the address* of the `Student` object named Harry.” The reason why we’re bringing attention to this seemingly meaningless formality is because, in C++, you can actually have a pointer that points to another pointer! For example, the following lines of code are perfectly valid, and the notion of a “pointer to a pointer” can actually be useful in many contexts:

```
1 Student lloyd("Lloyd"); // Student object
2 Student* dumb = &lloyd; // Student pointer with address of lloyd
3 // Student pointer pointer to store address of dumb
4 Student** dumber = &dumb;
```

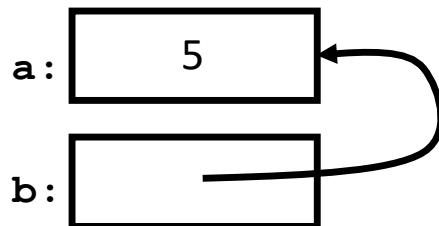
STOP and Think

Can you draw the memory diagram for the above code?

We have shown several examples of using pointers to point to objects, but you can also use pointers to point to primitive data, and to modify that data! For example, consider the following code:

```
1 int a = 5; // a is an int-type variable
2 int* b = &a; // b is a pointer to int storing the address of a
```

The diagram for the code above looks like this:



Now, you can use `b` to access, and *change*, `a`.

```
1 *b = 42; // This will change the value that's stored in a!
2 cout << a; // This will print 42
```

Memory Management In Java, there is very little that you, the programmer, must do to manage how memory is created or destroyed. You allocate new

memory for objects using the `new` keyword, and that memory is allocated on the heap and automatically reclaimed when the object is no longer reachable (you might have heard this being referred to as Java having a “garbage collector”). Unfortunately, in C++, it’s not so simple. In a nutshell, memory can be allocated in two ways: *automatically* and *dynamically* (it’s technically a little more complicated than this, but for our purposes, this will do). The default memory allocation is **automatic**, and it occurs whenever data is created unless you specify otherwise. For example, the following code uses automatic memory allocation:

```
1 int a = 5; // create space for int at compile time on stack
2 // create space for Student at compile time on stack
3 Student harry("Harry");
```

Automatic memory allocation is nice because, just like its name implies, the memory will also be *reclaimed* automatically when the variables go out of scope. Note that automatically allocated variables are *usually* created on the stack, and thus, you don’t have to worry too much about them.

On the other hand, **dynamic** memory allocation is done at run time and this memory will persist until *you* explicitly delete it. For example:

```
1 int* a = new int(5); // create new int on heap
2 // create new Student on heap
3 Student* harry = new Student("Harry");
```

The keyword `new` tells C++ to allocate the memory on the heap and not to delete that memory allocation until the programmer explicitly requests it, *even if all of the variables that point to that memory location go out of scope!* So, you, the programmer, must explicitly ask for all of your dynamically allocated memory to be deleted using the keyword `delete`. If you do not, you will have what’s called a *memory leak*. For example:

```
1 int* a = new int(5); // create new int on heap
2 // create new Student on heap
3 Student* harry = new Student("Harry");
4 // Some more code
5 delete a; // done with a, so delete it
6 delete harry; // done with harry, so delete it
7 // Some more code
```

By default, `delete` will free all the memory directly associated with the data stored in the variable. However, if the object points to other dynamically created data, the default behavior will *not* delete this nested data and you will again have a memory leak, as in the following example:

```

1 class Student { // C++
2     public:
3         Student(string n);
4         void setName(string n);
5         string getName() const;
6     private:
7         string* name; // name is now a pointer
8     };
9
10 Student::Student(string n) { // constructor
11     // name is allocated dynamically
12     // (just for illustration, not a good idea)
13     name = new string(n);
14 }
15
16 int doNothing(string aName) {
17     // Dynamically allocate Sammy
18     Student* s = new Student("Sammy");
19
20     // Some code here
21
22     delete s; // Delete memory allocated to sammy
23     // Because Student does not have a destructor defined,
24     // this will NOT delete the Student's name field!
25 }
```

Fortunately, when you call `delete` on an object, C++ will invoke the object's destructor. If it is a class you defined, you must supply a destructor for the object if the object itself has any dynamically created data. We can fix the code above by adding a destructor to the `Student` class:

```

1 class Student { // C++
2     public:
3         Student(string n);
4         ~Student(); // Declare the destructor
5         void setName(string n);
6         string getName() const;
7     private:
8         string* name; // name is now a pointer
9     };
10
11 Student::Student(string n) { // constructor
12     name = new string(n); // name is allocated dynamically
13 }
14
15 Student::~Student() { // destructor
16     delete name;
17 }
18
19 int doNothing(string aName) {
20     Student* s = new Student("Sammy"); // Dynamically allocate s
21     delete s; // This will call the destructor for Student
22 }
```

One final note about the code above is that there was NO NEED to create

either the `Student` object or the name within the `Student` object dynamically. We just did this for illustration. Dynamic allocation is useful when you need your objects to have life beyond the scope in which they are initially declared such as in linked data structures like Linked Lists and Trees.

Memory management in C++ can be challenging, but by keeping these two rules in mind, you'll be fine:

- If you can use automatic memory allocation, you probably want to
- If you cannot, make sure you have a call to `delete` for every call to `new`

Exercise Break

What would be output by the execution of the following lines of code? **Hint:** Draw the memory diagrams.

```

1 Student a("Lloyd");
2 Student & b = a;
3 Student * c = new Student("Harry");
4 b = *c;
5 Student d = a;
6 cout << d.getName();

```

Previously, we briefly mentioned the `const` keyword, but what does it actually mean? In general, `const` implies that something will not be allowed to be changed. Of course, this sounds vague, and the reason why we're keeping the general definition vague is because the actual restrictions that come about from the `const` keyword depend on where it is placed with respect to the variable it is modifying.

For variables in general, you can place the `const` keyword before or after the type, and the `const` keyword makes it so that the variable and the data it stores **can never be modified after initialization**. The following is an example using `int` objects:

```

1 const int a = 5; // 'a' cannot be modified after this
2 // it will always have a value of 5
3 int const b = 6; // 'b' cannot be modified after this
4 // it will always have a value of 6

```

If a `const` variable stores a mutable object, that object similarly may not be mutated (nor can the variable be reassigned).

With **pointers**, the `const` keyword becomes a bit trickier. The following are the different places we can place the `const` keyword and a description of the result:

```

1 int a = 5; // create a regular int
2 int b = 6; // create a regular int
3 // we can change what ptr1 points to,
4 // but we can't modify the actual data pointed to
5 const int * ptr1 = &a;
6 int const * ptr2 = &a; // ptr2 is equivalent to ptr1
7 // we can modify the data pointed to by ptr3,
8 // but we can't change what ptr3 points to
9 int * const ptr3 = &a;
10 // we can't change what ptr2 points to
11 // AND we can't modify the actual object itself
12 const int * const ptr4 = &a;
13
14 ptr1 = &b; // valid: I CAN change what ptr1 points to
15 *ptr1 = 7; // NOT valid: I CAN'T modify the data pointed to
16 *ptr3 = 7; // valid: I CAN modify the data pointed to
17 ptr3 = &b; // NOT valid: I CAN'T change what ptr3 points to
18 ptr4 = &b; // NOT valid: I CAN'T change what ptr4 points to
19 *ptr4 = 7; // NOT valid: I can't modify the data pointed to

```

With **references**, the **const** keyword isn't too complicated. Basically, it prevents modifying the data being referenced via the **const** reference. The following are some examples with explanations:

```

1 int a = 5; // create a regular int
2 const int b = 6; // create a const int
3 const int & ref1 = a; // create a const reference to 'a'
4 // (can't modify the value of a using ref1)
5 // This is OK. Even though a is allowed to change, it's OK to
6 // have a reference that does not allow you to change it because
7 // nothing unexpected will happen.
8 int const & ref2 = a; // equivalent to ref1
9 ref1 = 7; // NOT valid: ref1 can't modify the data
10 const int & ref3 = b; // valid: can have const ref to const data
11 int & ref4 = b; // NOT valid: non-const ref to const data
12 // ref4 might change the data but b says it shouldn't be changed,
13 // which is unexpected
14 int & const ref5 = a; // invalid syntax (const comes before &)

```

As mentioned previously, **functions** can be **const** as well. By making a function **const**, you are enforcing that the function can *only* modify *local* variables, or variables whose scope only exists *within* the function. You cannot modify anything that exists outside of the **const** function (in particular, instance variables).

Global Functions In Java, every function must either be an instance method or a static function of a class. C++ supports both of these cases, but it also supports functions that are not part of any class, called “global functions.” Typically, every functional C++ program starts with the global function **main**:

```

1 int main() {
2     // CODE
3 }
```

Notice that this main method has a return value (`int`), whereas the main methods of Java are `void`. By convention, a C++ main method returns 0 if it completed successfully, or some non-zero integer otherwise.

Passing Parameters C++ has two parameter-passing mechanisms: *call by value* (as in Java) and *call by reference*. When an object is passed by value, since C++ objects are not references to objects, the function receives a copy of the actual argument. As a result, the function cannot modify the original, and a lot of time and space might be wasted making a copy of the object. If we want to be able to modify the original object, we must pass the object by reference, which we can do by adding an & after the parameter type as follows:

```

1 // Successfully swaps the values of the argument variables
2 void swap(int & a, int & b) {
3     int temp = a;
4     a = b;
5     b = temp;
6 }
```

Note that, even if we do not intend to modify the original data, there are still times where we would want to pass by reference, and in these cases it's best to pass by constant reference. For example, in the world of Bioinformatics, programs often represent genomes as string objects and perform various algorithms on these strings. The human genome is roughly 3.3 billion letters long, so a string object containing the entire human genome would be roughly 3.3 GB large. Even if we have no intention of modifying this object in whatever method we will pass it in, we want to pass by reference so that we do not accidentally create a copy of this absurdly large object. If we truly do not intend to modify it, we pass it by constant reference, so we don't end up modifying it accidentally.

```

1 double gcContent(const string & genome) {
2     // CODE that does not modify genome
3 }
```

Finally, prior to C++11, passing by `const` reference was the only way to pass rvalues (values that aren't named variables, like the result of a sum, the return value of a function, or a literal value) to functions without passing them by value, because rvalues by their very transient nature are not allowed to be modified. Even though C++11 introduced the concept of rvalue reference (mainly) for passing rvalues to functions by reference, you will still see functions that use constant reference to allow passing of rvalues by reference. In short, in C++, you always use call by reference when a function needs to modify a parameter, and you still might want to use call by reference in other situations

as well.

Vectors The C++ vector has the best features of the array and `ArrayList` in Java. A C++ `vector` has convenient elemental access (using the familiar [] operator) and can grow dynamically. If `T` is some type, then `vector<T>` is a dynamic array of elements of type `T`. A vector also has the convenient “push” and “pop” functions of a stack (a data type that will be further discussed later in this text), where you can add an element to the back of the `vector` using the `push_back` function, and you can remove the last element of the vector using the `pop_back` function.

```

1 vector<int> a;           // a vector that is initially empty
2 vector<int> a(100);     // a vector initially containing 100 elements
3 a.push_back(0);          // add 0 to the end of a
4 a.pop_back();            // remove the last element of a
5 cout << a[10];          // output the element at index 10 of a

```

Vector Indexing Regarding indexing, the valid indices are between 0 and `a.size()-1` (just like in Java). However, unlike in Java, there is no runtime check for legal indices, so accessing an illegal index could cause you to access garbage data without even realizing it.

Memory Like in a Java `array`, the elements stored in a C++ `vector` are contiguous in memory (i.e., the elements all show up right after one another). Regarding memory storage, like all other C++ objects, a `vector` is a value, meaning the elements of the `vector` are values. If one `vector` is assigned to another, all elements are copied (unlike in Java, where you would simply have another reference to the same `array` object).

```

1 // a vector that initially contains 100 elements
2 vector<int> a(100);
3
4 // b is now a copy of a, so all of a's elements are copied
5 vector<int> b = a;

```

Lastly, we will discuss `input` and `output` in C++, which for basic I/O is much simpler than in Java.

IO Keywords/Operators/Functions In C++, the standard output, standard input, and standard error streams are represented by the `cout`, `cin`, and `cerr` objects, respectively. Also, newline characters can be output using the `endl` keyword. You use the `<<` operator to write to standard output (or to any `ostream`, for that matter):

```

1 cout << "Hello, world!" << endl << "My name is Lloyd!" << endl;

```

and you use the `>>` operator to read from standard input (or any `istream`, for that matter):

```

1 // declare the variable to hold the input
2 int n;
3
4 // prompt user
5 cout << "Please enter n: ";
6
7 // read user input and store in variable n
8 cin >> n;

```

The `getline` method reads an entire line of input (from any `istream`):

```

1 // declare the variable to hold the input
2 string userInput;
3
4 // read a single line from cin and store it in userInput
5 getline(cin, userInput);

```

End of Input If the end of input has been reached, or if something could not be read correctly, the stream is set to a failed state, which you can test for using the `fail` method:

```

1 int n;
2 cin >> n;
3 if(cin.fail()) {
4     cerr << "Bad input!" << endl;
5 }

```

At this point, you should hopefully be comfortable with C++ and the nuances between it and Java, at least to the point where you will be able to recognize issues with C++ code. If you feel that you are not yet at this point, please re-read this section and search the internet for any concepts you do not fully grasp.

Now that we have reviewed the main basic syntax and variable differences between C++ and Java, we will now discuss how **generic programming** is implemented in C++. With data structures, recall that we do not always know what data types we will be storing, but whatever data structures we implement should work exactly the same if we store `ints`, `longs`, `floats`, `strings`, etc. In Java, we could use “generics,” such as in the following example:

```

1 class Node<Data> { // Java Node class (generic type "Data")
2     public static final Data data;
3     public Node(Data d) {
4         data = d;
5     }
6 }

```

```

1 // a.data is a variable of type Student
2 Node<Student> a = new Node<Student>(exampleStudent);
3
4 // b.data is a variable of type String
5 Node<String> b = new Node<String>(exampleString);

```

In C++, the equivalent of this is the use of **templates**.

```

1 template<typename Data> // generic type "Data"
2 class Node { // C++ Node class (can now use type "Data")
3     public:
4         Data const data;
5         Node (const Data & d) : data(d) {}
6 }

```

```

1 // a.data is a variable of type Student
2 Node<Student> a(exampleStudent);
3
4 // b.data is a variable of type string
5 Node<string> b(exampleString);

```

Exercise Break

Given the following C++ `BSTNode` class, how would you create a pointer to a `BSTNode` with `int` data?

```

1 template<typename Data>
2 class BSTNode {
3     public:
4         BSTNode<Data>* left;
5         BSTNode<Data>* right;
6         BSTNode<Data>* parent;
7         Data const data;
8
9         BSTNode( const Data & d ) : data(d) {
10             left = right = parent = 0;
11         }
12 }

```

- `BSTNode* nodePtr;`
- `BSTNode<int> nodePtr;`
- `BSTNode<int>* nodePtr;`

In summation, C++ is a very powerful, but very tricky, programming language. There are very few safety checks because the main priority of the language is speed, and it is up to you to ensure that your code is correct and safe. In Java, if your code throws a runtime error, the Java Runtime Environment

gives you very detailed information about where in your code the error occurred and what the error was. In C++, however, there is a very good chance that your runtime error will simply say **SEGMENTATION FAULT**. Many Java programmers who transition to C++ are frustrated with the lack of detail in C++ runtime errors in comparison to Java runtime errors, so if you fall into this boat, do not feel discouraged! Be sure to read up on how to debug your code using **gdb** as it will certainly come in handy when you work on the programming assignments.

1.5 Random Numbers

The phenomenon of *randomness* has had numerous applications throughout history, including being used in dice games (as early as the 3,000s BCE in Mesopotamia), coin-flips (which originally were interpreted as the expression of divine will and were later used in games and decision-making), the shuffling of playing cards (which were used in divination as well as in games), and countless other applications.

To Computer Scientists, *randomness* is essential to a special class of algorithms known as **Randomized Algorithms**: algorithms that employ a degree of randomness as part of their logic. Randomized Algorithms can be broken down into two classes of algorithms: **Las Vegas Algorithms**, which use the random input to reduce the expected running time or memory usage but are guaranteed to terminate with a correct result, and **Monte Carlo Algorithms**, which have a chance of producing an incorrect result (but hopefully perform “pretty well” on average).

In the context of data structures specifically, *randomness* is essential to a special class of data structures known as **Randomized Data Structures**, which are data structures that incorporate some form of random input to determine their structure and data organization, which in turn effects their performance in finding, inserting, and removing elements.

For all of the applications listed above, we used the vague term *randomness*, but we can be more concrete and specify that these are all applications of **random number generation**. For example, rolling a k -sided die can be considered equivalent to generating a random number from 1 to k , flipping a coin can be considered equivalent to generating a random number that can only be 0 or 1, etc. Even with Randomized Algorithms and Randomized Data Structures, any applied randomness can be broken down into randomly generating numbers. In this section, we will discuss computational methods of random number generation.

It might seem strange, but it turns out that it can be extremely difficult to achieve *true* random number generation. The method by which we generate *true* random numbers is by measuring some physical phenomenon that is expected to be random and then compensating for possible biases in the measurement process. Example sources of physical randomness include measuring atmospheric noise, thermal noise, and other external electromagnetic and quantum phenomena: things that demonstrate natural *entropy* (or disorder).

The speed at which entropy can be harvested from natural sources is dependent on the underlying physical phenomena being measured, which is typically significantly slower than the speed of a computer processor. As a result, the computer processes that measure these sources of entropy are said to be “blocking”: they have to slow down (or halt entirely) until enough entropy is harvested to meet the demand. As a result, *true* random number generation is typically very slow.

It turns out, however, that there exist computational algorithms that can produce long sequences of seemingly random results, which are in fact completely determined by a shorter initial value, known as a **seed**. Basically, you create an instance of a random number, and you *seed* it with some number, which is then used to generate the sequence of “random” numbers. This type of a random number generator is said to be **pseudo-random** because it *seems* to be random for all intents and purposes, but given the seed, it’s fully deterministic.

For example, let’s say that yesterday, we seeded some random number generator using the integer 42 as our seed. The sequence of numbers we received as we queried the random number generator *seemed* to be randomly distributed. However, let’s say that today, we seed the random number generator, again with the integer 42 as our seed. The sequence of numbers we receive today will be *completely identical* to the sequence of numbers we received yesterday! Hence, the random number generator is *pseudo-random*. Because these algorithms for generating pseudo-random numbers are not bound by any physical measurements, they are *much faster* to generate than *true* random number generation.

STOP and Think

We mentioned that generating *true* random numbers is very slow, but generating *pseudo*-random numbers based off of some seed number is very fast. Is there some way we can merge the two approaches to get reasonably good randomness that is fast to generate?

It turns out that we can incorporate both approaches of random number generation to generate a sequence of random numbers that are “random enough” for practical uses, but in a fashion that is much faster than generating a sequence of true random numbers. We can take a pseudo-random number generator and *seed* it with a true random number! The result is a sequence of random numbers that can be generated quickly, but that changes upon each execution of our program.

For example, let’s say that we have a program that outputs a sequence of random numbers using a pseudo-random number generator, but seeded with some natural true random number (e.g. thermal noise in the computer circuitry). If we ran the program yesterday, it would have measured some amount of thermal noise (let’s call it x), then it would have seeded the pseudo-random number generator with x , and then it would have output a sequence of random numbers. If we then run the same program today, it would measure some amount of

thermal noise that would be *different* from yesterday's thermal noise (let's call it y), then it would seed the pseudo-random number generator with y (instead of with x), and it would generate an entirely different sequence of random numbers! Also, even though we still have to generate a true random number, which is slow, because we only have to generate a *single* true random number—and not an entire sequence of them—it is actually a fast enough operation.

In most programming languages, the built-in random number generator will generate a single random integer at a time, where the random integer will typically range between 0 and some language-specific maximum value. In C++, the random number generator is the `rand()` function, which generates a random integer between 0 and `RAND_MAX`, inclusive (the exact value of `RAND_MAX` depends on the exact C++ Standard Template Library, or STL, that is on your system). Before we call `rand()`, however, we must first seed it by calling the `srand()` function (which stands for `seed rand`) and passing our seed integer as a parameter. It is common to use the current system time as a seed.

```
1 srand(time(NULL)); // seed using system time
2 int number = rand(); // random number from 0 through RANDMAX
```

What if we wanted more control over the range of numbers from which we sample? For example, what if we want to generate a random integer in the range from 0 through 99? It turns out that we can do this fairly trivially using the modulo operator! Recall that the **modulo** operator (%) returns the remainder of a division operation. Specifically, if we call $a \% b$, we are computing the remainder of a divided by b . If we wanted to generate a number from 0 through 99, we could do the following:

```
1 int number = rand(); // random number from 0 through RANDMAX
2 number = number % 100; // map to range 0 through 99
```

What about if we wanted to generate a random number from 1 to 100?

```
1 int number = rand(); // random number from 0 through RANDMAX
2 number = number % 100; // map to 0-99
3 number = number + 1; // add 1 to map to 1-100
```

What about if we had C++ vector called `myVec`, and we wanted to choose a random element from it?

```
1 vector<string> myVec;
2 // add stuff to myVec
3 int index = rand() % myVec.size(); // randomly choose an index
4 string element = myVec[index]; // grab the element
```

The possibilities are endless! All of the examples above were sampling from the **Uniform Distribution** (i.e., every outcome was equally likely), but we can become even more clever and use statistics knowledge to sample from other

distributions (e.g. Gaussian, Exponential, Poisson, etc.) by performing some combination of operations using techniques like the ones above. In short, *any* random sampling from *any* distribution can be broken down into sampling from a Uniform Distribution.

Exercise Break

Using the C++ random number generator `rand()`, design a fair coin that outputs either 1 or 0 with equal probability.

Exercise Break

Using the C++ random number generator `rand()`, design a fair k -sided die that outputs one of the numbers from 1 through k , inclusive, with equal probability.

Exercise Break

Using the C++ random number generator `rand()`, design a biased coin that outputs 1 with probability p and 0 with probability $1-p$.

We hope that you are now comfortable with random number generation in general as well as in C++ specifically. Also, we hope you now understand the distinction between *true* random numbers (truly random, but can only be generated by harvesting natural entropy, which is slow) and *pseudo-random* numbers (seemingly random, but deterministic upon the seed, but can be generated very quickly). Also, remember that you can combine these two random number generation approaches by using a true random number to *seed* a pseudo-random number generator.

No matter what problem you are trying to solve, if you need to generate a random number in some certain range, or if you need to sample from some distribution, or if you need to do something possibly even more fancy, rest assured that any of these tasks can be broken down into sampling random integers. Later in this text, we will come back to random number generation as it applies to the various Randomized Data Structures we will be discussing.

1.6 Bit-by-Bit

In the digital age, people are typically comfortable with the notion of “bits” as being “1s and 0s” as well as with the notion of “bytes” being the fundamental unit of memory on a computer, but the connection between the two is not always obvious.

A **bit** is the basic unit of information in computing and can have only one of two values. We typically represent these two values as either a 0 or a 1, but they can be interpreted as logical values (true/false, yes/no), algebraic signs (+/-), activation states (on/off), or any other two-valued attribute.

A **byte** is a unit of digital information, and it is just a sequence of some number of bits. The size of the byte has historically been hardware-dependent, and no definitive standards existed that mandated the size, but for the purposes of this course as well as almost all computer applications, you can assume that a byte is specifically a sequence of **8 bits**. Note that, with modern computers, a byte is the smallest unit that can be stored. In other words, a file can be 1 byte, 2 bytes, 3 bytes, etc., but a file *cannot* be 1.5 bytes. In other words, a file's size must be a discrete number of bytes.

Exercise Break

How many distinct symbols could be represented with 1 byte?

Exercise Break

How many distinct symbols could be represented with 4 bytes?

As was mentioned previously, 1 byte is the smallest unit of storing memory in modern computers. Thus, every single file on a computer is simply a sequence of bytes, and by extension, a sequence of 8-bit sequences. Yes, *every* file on a computer is just a sequence of 8-bit chunks. Text, images, videos, audio, literally *all* filetypes.

As one can infer, if everything is represented as some sequence of bytes, there must be some mapping that is done to represent useful information as a sequence of 1s and 0s. **ASCII**, abbreviated from **American Standard Code for Information Interchange**, is a character-encoding scheme where each possible byte is mapped to a specific “symbol” (I say “symbol” in quotes because not all ASCII characters are meaningful to humans). A mapping of each of the possible bytes to their corresponding ASCII characters can be found at [ASCII Table](#).

If we think of decimal numbers in terms of their binary representations (i.e., as a sequence of bits), we can perform various bitwise operations on them. Before talking about bitwise operations, though, let's first (re-)familiarize ourselves with how to think of numbers in binary.

Recall from elementary school that, in the decimal system, numbers are organized into columns: the rightmost column is the “ones” column, then to the left of it is the “tens” column, then the “hundreds” column, etc. We then learned that “**decimal**” meant “**base 10**” and that the rightmost column of a decimal number actually represents 10^0 , the column to the left of it represents 10^1 , then 10^2 , etc. For example, the number 729 can be thought of as $(10^2 \times 7) + (10^1 \times 2) + (10^0 \times 9)$.

Binary numbers work in the same way, but instead of being “base 10” (which is the case for “decimal”), they are “**base 2**.” Thus, the rightmost column represents 2^0 , the column to the left of it represents 2^1 , then 2^2 , etc. For example, the number 101 can be thought of as $(2^2 \times 1) + (2^1 \times 0) + (2^0 \times 1)$. In other words, 101 in binary is equal to 5 in decimal.

Exercise Break

Convert the binary number 101010 to decimal.

Binary addition and subtraction work just like decimal addition and subtraction: align the digits of the two numbers and simply add column by column, carrying the 1 if a given column overflows. In decimal addition, we carry the one when a given column wraps around from a value of 9 to a value of 0 (because in decimal, the valid digits are 0-9, so incrementing 9 wraps around to 0). Identically, in binary addition, we carry the one when a given column wraps around from a value of 1 to a value of 0 (because in binary, the valid digits are 0-1, so incrementing 1 wraps around to 0).

The following is an example of the addition of two binary numbers:

a:	0	1	0	1
b:	0	1	1	0
<hr/>				
a+b:	1	0	1	1

Note that the right column's addition was simply $1 + 0 = 1$. Then, the next column to the left was simply $0 + 1 = 1$. Then, the next column was $1 + 1 = 10$, so we put a value of 0 in that column and carried the 1. Lastly, the leftmost column was simply $0 + 0 = 0$, plus the 1 we carried over, so we put a value of 1.

Exercise Break

Add the binary numbers 0101 and 0101.

Aside from addition, there are many **bitwise operations** one can do on numbers. These bitwise operations treat the numbers as their binary and perform the relevant operation bit-by-bit. The following bitwise operations will be explained using single-bit examples, but when performed on numbers with more than one bit, you simply go through the numbers' bits column-by-column, performing the single-bit examples independently on each column independently. It will help to follow the logic of the operations by thinking of the bits in terms of 1 = TRUE and 0 = FALSE.

Bitwise AND (&)	1 & 1 = 1	0 & 1 = 0	1 & 0 = 0	0 & 0 = 0
Bitwise OR ()	1 1 = 1	0 1 = 1	1 0 = 1	0 0 = 0
Bitwise XOR (^)	1 ^ 1 = 0	0 ^ 1 = 1	1 ^ 0 = 1	0 ^ 0 = 0
Bitwise NOT (~)	$\sim 1 = 0$		$\sim 0 = 1$	

In addition to these single-bit operations, there are bit-shifting operations that can be done on binary numbers. The **left bit-shift operator** (`<<`) shifts each bit of the binary number left by the specified number of columns, and the **right bit-shift operator** (`>>`) shifts each bit of the binary number right by the specified number of columns. For example, we shift two 8-bit numbers:

- `00001000 << 2 = 00100000`
- `00001000 >> 2 = 00000010`

When bit-shifting, one should imagine the 1s as “information” and the 0s as “empty space.” If a 1 gets “pushed over the edge,” it is simply lost (or “cleared”). Also, as can be seen in the example above, when we shift left, the columns on the right side of the number are filled with “empty space” (0s), and when we shift right, the columns on the left side of the number are also filled with “empty space” (0s).

The following are some examples of performing the previously described bitwise operators. Note that, even though the numbers are initially represented as decimal numbers, the operations treat them as their binary representations (because, in reality, the numbers are represented in binary on the computer and are simply displayed to us in decimal, or hex, or whatever representation we choose).

<code>unsigned char a = 5, b = 67;</code>								
a:	0	0	0	0	0	1	0	1
b:	0	1	0	0	0	0	1	1
Bitwise AND: a & b:	0	0	0	0	0	0	0	1
Bitwise OR: a b:	0	1	0	0	0	1	1	1
Bitwise XOR: a ^ b:	0	1	0	0	0	1	1	0
Bitwise NOT: ~a:	1	1	1	1	1	0	1	0
Left Shift: a << 2:	0	0	0	1	0	1	0	0
Right Shift: a >> 2:	0	0	0	0	0	0	0	1

Just like with any other type of math, bitwise operations can only really be learned through practice. As such, we will be testing your skills in the next few exercises.

Exercise Break

What is the result of the bitwise expression `7 | (125 << 3)`? Assume the numbers are unsigned 8-bit numbers.

Note that, previously, we specified the datatype `unsigned char` to represent a byte in C++. In many programming languages (such as in C++), datatypes can be either *signed* or *unsigned*. In all of the previous examples in this section, we exclusively dealt with *unsigned* values: the smallest value is 0, and all other values are positive. For example, in C++, an `unsigned int` is 4 bytes (i.e., 32 bits), meaning the smallest possible value is 00000000 00000000 00000000 00000000 (which has a value of 0), and the largest possible value is 11111111 11111111 11111111 11111111 (which has a value of $2^{32} - 1$). In other words, with an *unsigned* datatype, all of the bits are used to represent *magnitude* of the number.

In a *signed* datatype, one bit is reserved to represent the *sign* of the number (typically, 0 = positive and 1 = negative), and the remaining bits represent the *magnitude* of the number. Typically, the “sign bit” is the left-most bit of the number (as it is in C++). Because one of the bits is reserved to represent the *sign* of the number, if a given signed datatype has a size of n bits, the smallest value it can hold is -2^{n-1} and the largest value it can hold is $2^{n-1} - 1$. For example, in C++, a `signed int` is 4 bytes (i.e., 32 bits, just like an `unsigned int`), but because the leftmost bit represents the sign of the number, the remaining 31 bits are used to represent magnitude, meaning the smallest value that can be represented is -2^{31} and the largest value that can be represented is $2^{31} - 1$.

Depending on the programming language, performing bitwise operations on a signed datatype performs the given operation on the $n-1$ *magnitude* bits without modifying the *sign* bit (such that the sign of the number does not change). As a result, programming languages that have this feature often have signed *and* unsigned versions of each of the bitwise operations. In general, be wary when performing bitwise operations on a signed datatype, and be sure to reference the relevant documentation to see how the bitwise operations work on signed datatypes in the language you are using.

It is out of the scope of this text, but if you are interested in learning how to reverse the sign of a given number in binary, you can look into **Two’s Complement**, the bitwise operation that does so.

Exercise Break

What is the **largest** integer a C++ `unsigned char` can represent, given that it has a size of 1 byte (i.e., 8 bits)?

Exercise Break

What is the **largest** integer a C++ `signed char` can represent, given that it has a size of 1 byte (i.e., 8 bits)?

Exercise Break

What is the **smallest** integer a C++ `signed char` can represent, given that it has a size of 1 byte (i.e., 8 bits)?

1.7 The Terminal-ator

When non-Unix users see the Unix Command Line (i.e., the Terminal), they often immediately think of 80's hacker movies. Although we must admit that using your computer's terminal for regular computer science tasks isn't quite as intense as "hacking into the mainframe" of some evil corporation, the terminal is still quite powerful. In this day and age, we have become used to the "point-and-click" (or even simply "touch") nature of modern operating systems, but most large-scale computational tasks are done remotely via some compute server. Consequently, knowing how to use the Unix command line properly is vital.

In this section, you will learn some of the basics of how to navigate the Unix command line, and in addition to basic navigation, you will be introduced to some of the more powerful Unix tools built into most Unix-based operating systems. Beware that we will only be touching the surface in this course, and as with most computer-related things, one can only become truly comfortable using the terminal with practice, practice, practice.

Unfortunately, it is quite impossible to truly teach you how to use the Unix command line from absolutely no experience via a textbook, and you will instead want to use an online course to do so. Our personal recommendation is the interactive self-paced Codecademy course called Learn the Command Line. Work through the Codecademy lessons (the resources they make available for free will suffice for our purposes, but you have the option of buying a subscription to have access to their paid resources if you want extra practice), and when you are finished (or if you are already comfortable using the Unix command line), continue onward! Note that you can solve all of the following challenges directly within the accompanying Stepik text.

Exercise Break

To warm up, we'll start with something simple: create a file called `message.txt` in the working directory containing the text "`Hello, world!`". You may want to use `vim`, a command line text editor, to create the file. Use the Interactive `vim` Tutorial to learn how to use it if you are unfamiliar with it.

Exercise Break

Now something a bit more challenging: create a directory called `files` in the working directory, and within that directory, create 3 files named `file1.txt`, `file2.txt`, and `file3.txt` (the content of these files is irrelevant).

Exercise Break

Given a FASTA file `example.fa` in the working directory, perform the following tasks:

1. Count the number of sequences in `example.fa`, and save the result into a file in the working directory called `1_num_seqs.txt` (**Hint:** you can `grep` for the '`>`' symbol and then use `wc` to count the number of lines)
2. In each sequence (i.e., in each DNA string), the “seed” is defined as characters 2-8 of the sequence (inclusive, 1-based counting). Find the unique “seed” sequences in `example.fa`, and save the result into a file in the working directory called `2_unique_seeds.txt` (**Hint:** you will want to exclude identifiers, which you can do via `grep -v` for the '`>`' symbol to isolate the sequences, then use `cut` to extract the “seed” sequences, and then use `sort` and `uniq` to remove duplicate lines)

It might be useful to read the Wikipedia page about the FASTA format if you are unsure about how `example.fa` is structured.

In general, the vast majority of your time with the Unix command line will be basic navigation, file manipulation, and program execution. If you don’t remember all of the fancier Unix commands off the top of your head, that’s perfectly fine. You can always search Google if you have a specific problem you need to solve, and StackOverflow is always your friend on that front.

If you are still not comfortable with the Unix command line, as we mentioned before, the best way to learn is to practice. For more practice, we suggest installing a Linux distribution onto a virtual machine via VirtualBox. If you end up really enjoying Linux and want to use it as one of your main Operating Systems on your computer (as opposed to through a virtual machine on top of your main Operating System), you might even want to think about dual-booting your main Operating System with Ubuntu (or whatever Linux distribution you prefer).

Earlier in this text, we reviewed/introduced some basic C++ syntax and usage. Further, throughout this text, we will have C++ coding challenges embedded within the lessons in order to test your understanding of the data structures and algorithms we will be discussing. For these coding challenges, you will be performing all of your programming directly within Stepik in order to avoid issues regarding setting up your environment. However, as a Computer Scientist, you will likely be developing and running your own code from the command line, either locally on your own computer or remotely on some compute cluster. As such, it is important for you to know how to compile and run code from the Unix command line. In the second half of this lesson, we will be learning how to write, compile, and execute C++ code directly from the command line.

In an introductory programming course, or as a beginner programmer, you are likely to write code that is quite simple and that can be written cleanly in a single file. For example, say we have a file called `HelloWorld.cpp` that consists of the following:

```

1 #include <iostream>
2 int main() {
3     std::cout << "Hello , world!" << std::endl;
4 }
```

We could compile our code using `g++`, a C++ compiler, and execute it, all from the command line:

```

1 $ g++ HelloWorld.cpp # compile our program
2 $ ./a.out           # run our program
3 Hello , world!      # our code's output
```

The default filename for the executable file was `a.out`, which isn't very descriptive. To specify a filename, we can use the `-o` flag:

```

1 $ g++ -o hello_world HelloWorld.cpp # compile our program
2 $ ./hello_world                   # run our program
3 Hello , world!                  # our code's output
```

Exercise Break

Perform the following tasks:

1. Write a program `PrintNums.cpp` that prints the numbers 1-10 to standard output (`std::cout`), one number per line
2. Use `g++` to compile `PrintNums.cpp` and create an output executable file called `print_nums`

The expected output to standard output (`cout`) from running `print_nums` should look like the following:

```

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

You may want to use `vim`, a command line text editor, to write your code.

The simple `g++` command we just learned is great if your code is simple and fits in just a handful of files, but as you become a more seasoned programmer, you will begin embarking on more and more complex projects. When working on a large and robust project, good practice dictates that the program should be modularized into smaller individual components, each typically in its own

file, for the sake of organization and cleanliness. Although this makes the actual programming and organization easier, it makes compiling a bit more complex: the more components a program has, the more things we need to specify to `g++` for compilation.

As a result, developers of complex projects will include a **Makefile**, which is a file that contains all of the information needed to compile the code. Then, as a user, we simply download the project and call the `make` command, which parses the **Makefile** for us and performs the compilation on its own. The basic syntax for a **Makefile** is as follows (by [tab], we mean the tab character):

```
1 target: dependencies
2 [tab]system command
```

For example, we could compile the previous example using the following **Makefile**:

```
1 all:
2         g++ -o hello_world HelloWorld.cpp
```

Note that the whitespace above is a **tab character**, *not* multiple spaces. In a **Makefile**, you **must** use tab characters for indentation. To compile our code, assuming we have `HelloWorld.cpp` and **Makefile** in the same directory, we can go into the directory and do this:

```
1 $ make           # run the make command to compile our code
2 $ ./hello_world # run our program
3 Hello, world!   # our code's output
```

In the example above, our target is called `all`, which will be the default target for this **Makefile**. The `make` program will use this target if no other one is specified. However, note that, in general, the default target for **Makefiles** is whichever target was specified *first*. We also see that there are no dependencies for target `all`, so `make` safely executes the system commands specified.

Depending on the project, if it has been modularized enough, it is useful to use different targets because, if you modify a single file in the project, `make` won't recompile *everything*: it will *only* recompile the portions that were modified. Also, you can add custom targets for doing common tasks, such as removing all files created during the compilation process. For example, say our `HelloWorld.cpp` code depended on another file, `PrintWords.cpp`, we might create the following **Makefile**:

```

1 all: hello_world          # default all target
2 hello_world: HelloWorld.o PrintWords.o # hello_world target
3         g++ -o hello_world HelloWorld.o PrintWords.o
4 HelloWorld.o: HelloWorld.cpp      # HelloWorld.o target
5         g++ -c HelloWorld.cpp
6 PrintWords.o: PrintWords.cpp     # PrintWords.o target
7         g++ -c PrintWords.cpp
8 clean:                   # clean target
9 rm *o hello_world

```

We can then compile and run our code as follows:

```

1 $ make      # compile our code
2 $ ./hello_world # run our program
3 Hello, world! # our code's output
4 $ make clean   # remove all files resulting from compilation

```

You can get pretty fancy when you write a **Makefile**! So as your projects become more and more complex, be sure to write a **Makefile** that is robust enough to handle your project with ease.

Exercise Break

Perform the following tasks:

1. Write a program `PrintNums.cpp` that prints the numbers 1-10 to standard output (`std::cout`), one number per line
2. Write a **Makefile** such that your code can be compiled with just the `make` command. It should create an output executable file called `print_nums`
3. Compile your code using the `make` command

The expected output to standard output (`cout`) from running `print_nums` should look like the following:

```

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

```

You may want to use `vim`, a command line text editor, to write your code.

As we mentioned before, being fluent in using the Unix command line is an essential quality to have, especially in this day and age. Some examples of tasks that require usage of the Unix command line include the following:

- Running large computational tasks on a remote compute cluster
- Developing, compiling, and executing code
- Automating repetitive tasks for convenience
- So much more!

Just like with most things, the only way to become comfortable with the Unix command line is practice, practice, practice. In time, you'll get the hang of things!

1.8 Git, the “Undo” Button of Software Development

“That feature was working last night, but then I tried to fix this other feature, and it broke everything! [insert curse word] I need to at least get it back to a working state!”

—Every computer science student, at some point in their career

Whether we are talking about a piece of code for a programming assignment, or about your final essay after you made incoherent changes to it at 3:30 AM, or about your now unusable car after your friend Tim insisted he could install a new sound system because he's an electrical engineer and “a car is just a big circuit,” we have all wished at some point in our lives that we could revert something to a previous unbroken state.

Luckily for us, there exists a solution to two of the three scenarios described above! **Version Control** is a system that records changes to a file or set of files over time so that you can recall specific versions later. We will be focusing on one specific version control system, **git**, which is one of the main version control systems used by software developers. Git may not be able to fix your car, but at least your grades will be able to flourish!

There are three main types of version control systems (VCSs): *local*, *centralized*, and *distributed*.

A **local** VCS is essentially a more sophisticated way of saving different versions of a file locally (think `Final_Essay_rough.doc`, `Final_Essay_final.doc`, `Final_Essay_final_final.doc`, `Final_Essay_final_seriously.doc`, etc.). In practice, a local VCS does not actually save all of these different versions of the same file. Instead, for each revision of your file, it keeps a “patch” (i.e., a list of edits) with respect to a previous version. Thus, to re-create what a file looked like at any point in time, the local VCF simply adds up all of the patches until that time point.

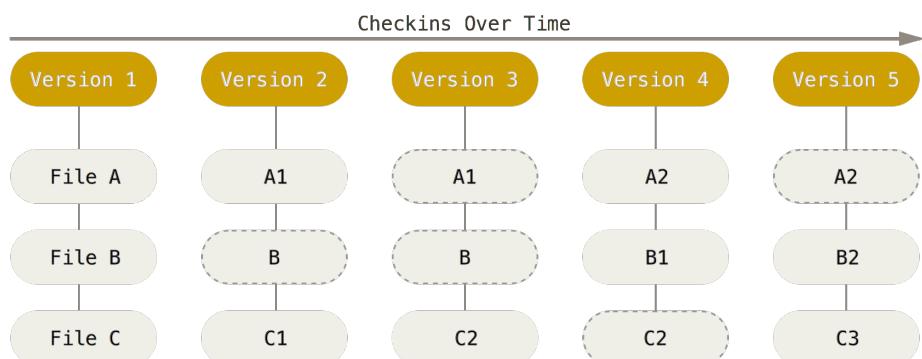
A local VCS solves the issue of keeping a personal set of revisions to a file, but as we know, most real-world projects are not one-person jobs: they typically consist of collaborative efforts of an entire team. A **centralized** VCS solves this need to have version control when dealing with revisions from multiple

individuals. Centralized VCSs have a single server that contains all of the versioned files. Individual developers check out files from that central place when they want to make changes. This works quite well, but it has some flaws. One large flaw being: What if the centralized server goes down, even temporarily? During that downtime, nobody can collaborate at all, nor can they save any changes to anything they were working on. Also, if the server goes down permanently (e.g. the hard drive crashes), everything except for what the developers might have had checked out locally would be lost forever.

Consequently, we have the **distributed** VCS, such as Git. Distributed VCS clients don't just check out the latest snapshot of the files in the project: they fully mirror the repository locally. In other words, every clone is a full backup of all the data: if the centralized server dies, any local client repository can simply be copied back to the server to restore it.

Git is a relatively unique distributed VCS in that it treats versions as “snapshots” of the repository filesystem, not as a list of file-based changes. From the official Git website:

*“Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a **stream of snapshots**. ”*



In general, the Git workflow is as follows:

1. Do a **clone** (clones the remote repository to your local machine)
2. Modify the files in your working directory
3. Stage the files (adds snapshots of them to your staging area)
4. Do a **commit** (takes the files as they are in the staging area and stores that snapshot permanently to your Git directory)

5. Do a **push** (upload the changes you made locally to the remote repository)

For more extensive information about Git, be sure to read the official Pro Git book. In general, the extent of Git you will need to use for programming assignments in a data structures course or for a personal project will be as follows:

1. **Create** the Git repository
2. **Clone** the Git repository locally (`git clone <repo_url> <folder_name>`)
3. **Add** any files you need synchronized in the repository (`git add <files>`)
4. **Commit** any changes you make (`git commit -m <commit_message>`)
5. **Push** the changes to the repository (`git push`)

Of course, this introduction to Git was extremely brief, but in reality, the bulk of your day-to-day Git usage will be pulling, adding, committing, and pushing. The more complex features are typically needed once in a blue moon, in which case you can simply Google the issue you’re having and find an extensive StackOverflow post answering your exact question. However, if you still feel as though you want more practice with Git, Codecademy has an excellent interactive self-paced free online course called Learn Git.

Chapter 2

Introductory Data Structures

2.1 Array Lists

The first data structure that will be covered in this text is one of the fundamental data structures of computer science: the **array**. An array is a homogeneous data structure: all elements are of the same type (int, string, etc). Also, the elements of an array are stored in adjacent memory locations. The following is an example of an array, where the number inside each cell is the index of that cell, using 0-based indexing (i.e., the first element is at index 0, the second element is at index 1, etc.).

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Because each cell has the same type (and thus the same size), and because the cells are adjacent in memory, it is possible to quickly calculate the address of any array cell, given the address of the first cell.

Say we allocate memory for an array of n elements (the total number of cells of the array must be defined beforehand), where the elements of the array are of a type that has a size of b bytes (e.g. a C++ `int` has a size of 4 bytes), and the resulting array is allocated starting at memory address x . Using 0-based indexing, the element at index $i = 0$ is at memory address x , the element at index $i = 1$ is at memory address $x + b$, and the element at index i is at memory address $x + bi$. The following is the same example array, where the number inside each cell is the index of that cell, and the number below each cell is the memory address at which that cell begins.

0	1	2	3	4	5	6	7	8	9
$ x$	$ x+b$	$ x+2b$	$ x+3b$	$ x+4b$	$ x+5b$	$ x+6b$	$ x+7b$	$ x+8b$	$ x+9b$

Because of this phenomenon of being able to find the memory address of any i -th element in constant time (and thus being able to access any i -th element in constant time), we say that arrays have **random access**. In other words, we can access any specific element we want very quickly: in $\mathcal{O}(1)$ time.

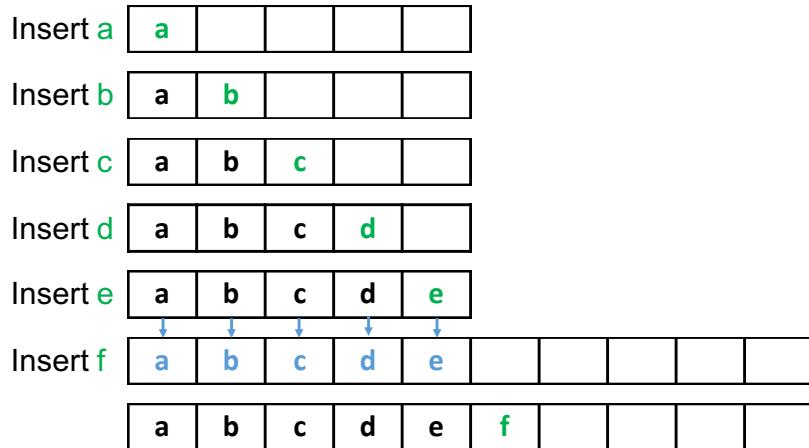
Exercise Break

You create an array of integers (assume each integer is exactly 4 bytes) in memory, and the beginning of the array (i.e., the start of the very first cell of the array) happens to be at memory address 1,000 (in decimal, not binary). What is the memory address of the start of cell 6 of the array, assuming 0-based indexing (i.e., cell 0 is the first cell of the array)?

For our purposes from this point on, we will think of arrays specifically in the context of using them to store contiguous lists of elements: an **ArrayList**. In other words, we'll assume that there are no empty cells between elements in the array (even though an array in general has no such restriction). As a result, we will assume that a user can only add elements to indices between 0 and n (inclusive), where n is the number of total elements that exist in the list prior to the new insertion (i.e., a user can only add elements to the *front* or *back* of the array).

As can be inferred, we as programmers don't always know exactly how many elements we want to insert into an ArrayList beforehand. To combat this, many programming languages implement ArrayLists as "dynamic": they allocate some default "large" amount of memory initially, insert elements into this initial array, and once the array is full, they create a new larger array (typically twice as large as the old array), copy all elements from the old array into the new array, and then replace any references to the old array with references to the new array. In C++, the "dynamic array" is the **vector** data structure.

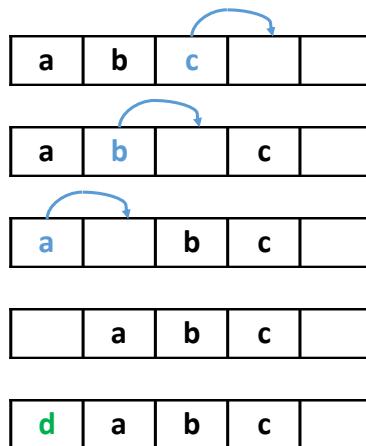
The following is an example of this in action, where we add the letters a-f to an ArrayList backed by an array initialized with 5 cells, where each insertion occurs at the end of the list (notice how we allocate a new array of twice the old array's length and copy all elements from the old array into the new array when we want to insert 'f'):



STOP and Think

Array structures (e.g. the array, or the Java `ArrayList`, or the C++ `vector`, etc.) require that all elements be the same size. However, array structures can contain strings, which can be different lengths (and thus different sizes in memory). How is this possible?

Based on the previous example, it should be clear that inserting at the end of an Array List for which the backing array is not full, assuming I know how many elements are currently in it, is constant time. However, what if I want to insert at the beginning of an Array List? Unfortunately, because of the rigid structure of an array (which was vital to give us random access), I need to move a potentially large number of elements out of the way to make room for the element I want to insert. The following is an example of inserting **d** to the beginning of an Array List with 3 elements previously in it:



As can be seen, even though the best-case time complexity for insertion into an array is $\mathcal{O}(1)$ (which we saw previously), the worst-case time complexity for insertion into an Array List is $\mathcal{O}(n)$ because we could potentially have to move all n elements in the array (or as in the case we saw previously, we may have to allocate an entirely new array and copy all n elements into this new array).

Formally, the following is the pseudocode of insertion into an Array List. Recall that we assume that all elements in the array must be contiguous. Also, note that the pseudocode uses 0-based indexing.

```

1 // insert element into array
2 // return True on success or False on failure
3 insert(element, index):
4     // perform the required safety checks before insertion
5     if index < 0 or index > n:    // invalid indices
6         return False
7
8     if n == array.length:        // if array is full
9         newArray = empty array of length 2*array.length
10        for i from 0 to n-1:    // copy over all elements
11            newArray[i] = array[i]
12        array = newArray        // replace old array with new
13
14    // perform the insertion algorithm
15    if index == n:             // insertion at end of array
16        array[index] = element // perform insertion
17
18    else:                      // general insertion
19        for i from n-1 to index: // make space for new element
20            array[i+1] = array[i]
21        array[index] = element // perform insertion
22
23    n = n+1                  // increment number of elements
24    return True

```

Also, the following is the pseudocode of finding an element in an arbitrary array (i.e., the elements are in no particular order). Again, the pseudocode is using 0-based indexing.

```

1 // returns True if element exists in array, otherwise False
2 find(element):
3     for i from 0 to n-1:        // iterate through all n elements
4         if array[i] == element: // if we match, return True
5             return True
6     return False                // element not found

```

Exercise Break

What is the worst-case time complexity for an “insert” operation in an arbitrary Array List (i.e., we know nothing about the elements it contains)?

Exercise Break

What is the worst-case time complexity for a “find” operation in an arbitrary Array List (i.e., we are searching for a given element and we know nothing about the elements the Array List already contains)?

As you may have inferred, the worst-case time complexity to find an element in an arbitrary Array List is $\mathcal{O}(n)$ because there is no known order to the elements, so we have to individually check each of the n elements.

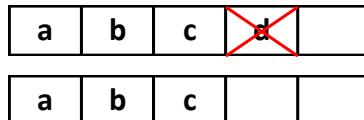
However, what if our Array List happened to be sorted? Could we exploit that, along with the feature of random access, to speed things up? We will introduce an algorithm called **Binary Search**, which allows us to exploit random access in order to obtain a worst-case time complexity of $\mathcal{O}(\log n)$ for searching in a **sorted** Array List. The basic idea is as follows: because we have random access, compare the element we’re searching for against the middle element of the array. If our element is less than the middle element, our element must exist on the left half of the array (if at all), so repeat the search on the left half of the array since there is no longer any point to search in the right half of the array. If our element is larger than the middle element, our element must exist on the right half of the array (if at all), so repeat on the right half of the array. If our element is equal to the middle element, we have successfully found our element.

```

1 // perform binary search to find element in a sorted array
2 BinarySearch(array, element):
3     L = 0 and R = n-1           // initialize left and right
4
5     loop infinitely:
6         if L > R:             // left > right: we failed
7             return False
8         M = the floor of (L+R)/2 // compute middle index
9
10        if element == array[M]: // we found our element
11            return True
12        if element > array[M]: // recurse right
13            L = M + 1
14        if element < array[M]: // recurse left
15            R = M - 1

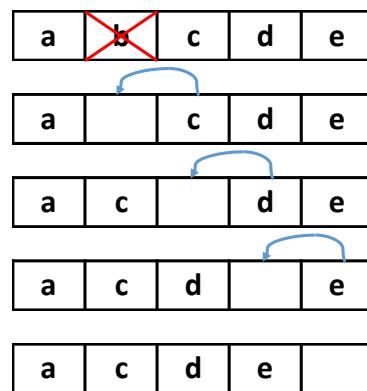
```

Just like with insertion, removal at the end of our list of elements is very fast: we simply remove the element at the n -th index of our backing array, which is a constant-time operation. The following is an example of removing the last element from an Array List:



However, also just like with insertion, removal at the beginning of our list of

elements is very slow: we remove the element at index 0 of our backing array, which is a constant-time operation, but we then need to move all of the elements left one slot, which is an $\mathcal{O}(n)$ operation overall. In the following example, instead of removing from the very beginning of the array, we remove the second element, and we see that we still have a worst-case $\mathcal{O}(n)$ operation when we have to move all of the elements left. Note that this “leftward move” requirement comes from our restriction that all elements in our array must be contiguous (which is necessary for us to have random access to any of the elements).



STOP and Think

When we remove from the very beginning of the backing array of an Array List, even before we move the remaining elements to the left, the remaining elements are all still contiguous, so our restriction is satisfied. Can we do something clever with our implementation to avoid having to perform this move operation when we remove from the very beginning of our list?

Exercise Break

You are given an Array List with 100 elements, and you want to remove the element at index 7 (0-based indexing). How many “left-shift” operations will you have to perform on the backing array?

Formally, the following is the pseudocode of removal from an Array List. Recall that we assume that all elements in the array must be contiguous. Also, note that the pseudocode uses 0-based indexing.

```

1 // removes element at position "index" in the array
2 remove(index):
3     if index < 0 or index >= n: // invalid indices
4         return False
5     clear array[index]
6     if index < n-1:
7         for i from index to n-2: // shift all elements left
8             array[i] = array[i+1]
9             clear array[n-1]           // not technically necessary
10            n = n-1                  // decrement number of elements
11            return True

```

STOP and Think

Why is it not actually necessary to “clear” the element in the last position of the Array List during a “remove” operation?

In summation, Array Lists are great if we know exactly how many elements we want and if the data is already sorted, as finding an element in a sorted Array List is $\mathcal{O}(\log n)$ in the worst case and accessing a specific element is $\mathcal{O}(1)$. However, inserting into an Array List is $\mathcal{O}(n)$ in the worst case and finding an element in a non-sorted Array List is $\mathcal{O}(n)$. Also, if we don’t know exactly how many elements we want to store, we would need to allocate extra space in order to avoid having to rebuild the array over and over again, which would waste some space.

In general, all data structures have applications in which they excel as well as cases in which they fail, and it is up to the programmer to keep in mind these trade-offs when choosing what data structures to use.

2.2 Linked Lists

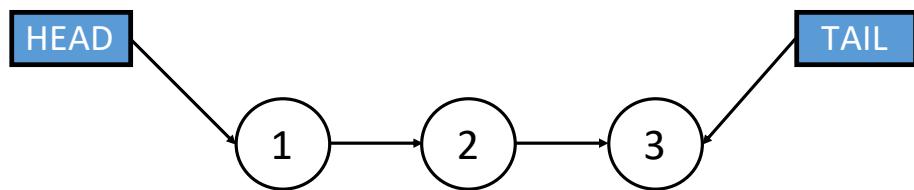
Recall from the previous section that Array Lists are excellent if we know exactly how many elements we want to store, but if we don’t, they can be problematic either in terms of time complexity (rebuilding the backing array as we need to grow it) or in terms of space complexity (allocating more space than we need just in case).

We will now introduce a data structure called a **Linked List**, which was developed in 1955 by Allen Newell, Cliff Shaw, and Herbert A. Simon at RAND Corporation. The Linked List is a dynamically-allocated data structures, meaning it grows dynamically in memory on its own very time-efficiently (as opposed to an Array List, for which we needed to explicitly reallocate memory as the backing array fills, which is a very time-costly operation). When analyzing various aspects of the Linked List, try to keep the Array List in mind because both are “list” structures, so they can effectively do the same things, but they each have their respective pros and cons.

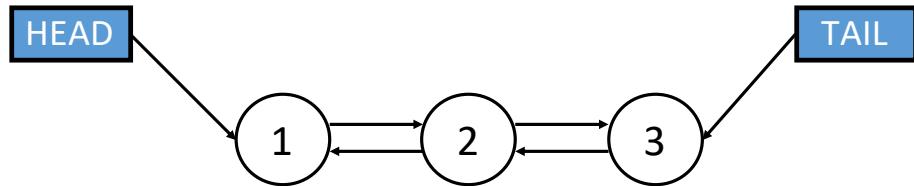
A Linked List is a data structure composed of **nodes**: containers that each hold a single element. These nodes are “linked” to one another via pointers.

Typically, we maintain one global ***head*** pointer (i.e., a pointer to the first node in the Linked List) and one global ***tail*** pointer (i.e., a pointer to the last node in a Linked List). These are the only two nodes to which we have direct access, and all other nodes can only be accessed by traversing pointers starting at either the head or the tail node.

In a **Singly-Linked List**, each node only has a pointer pointing to the node directly after it (for internal nodes) or a null pointer (for the tail node). As a result, we can only traverse a Singly-Linked List in one direction.



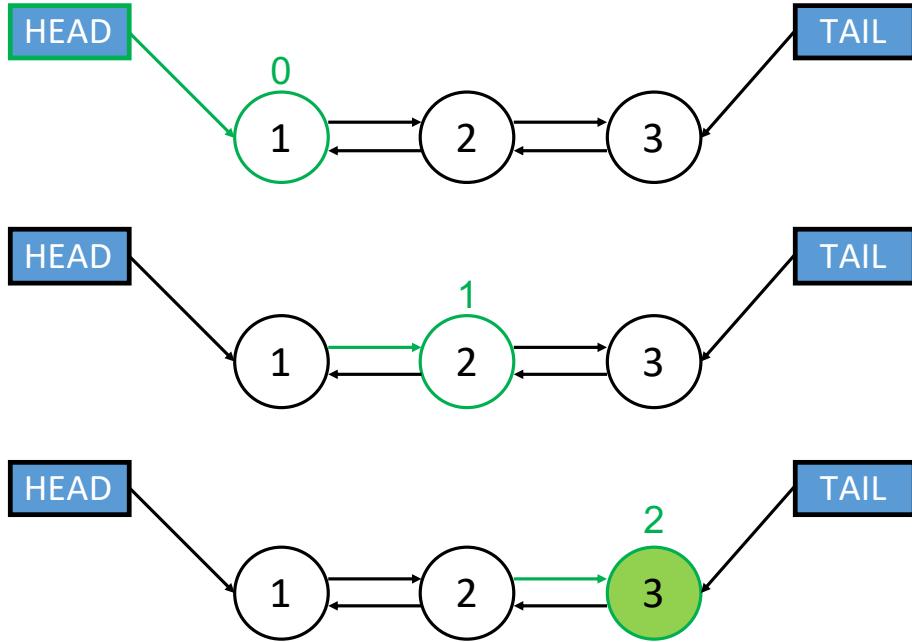
In a **Doubly-Linked List**, each node has two pointers: one pointing to the node directly after it and one pointing to the node directly before it (of course, the head and the tail nodes only have a single pointer each). As a result, we can traverse a Doubly-Linked List in both directions.



Exercise Break

If I only have direct access to the *head* or *tail* pointer in a Linked List, and I can only access the other elements by following the nodes' pointers, what is worst-case time complexity of finding an element in a Linked List with n elements?

As you may have inferred previously, to find an arbitrary element in a Linked List, because we don't have direct access to the internal nodes (we only have direct access to *head* and *tail*), we have to iterate through all n elements in the worst case. The following is an example Linked List, in which we will find the element at index $i = 2$ (where indexing starts at $i = 0$):



STOP and Think

Notice that when we look for an element in the middle of the Linked List, we start at the *head* and step forward. This works fine if the index is towards the beginning of the list, but what about when the index of interest is large (i.e., closer to n)? Can we speed things up?

As mentioned previously, to find an element at a given index in a Linked List, we simply start at the head node and follow forward pointers until we have reached the desired node. Note that, if we are looking for the i -th element of a Linked List, this is a $\mathcal{O}(i)$ operation, whereas in an array, accessing the i -th element was a $\mathcal{O}(1)$ operation (because of “random access”). This is one main drawback of a Linked List: even if we know exactly what index we want to access, because the data is not stored contiguously in memory, we need to slowly iterate through the elements one-by-one until we reach the node we want.

The following is pseudocode for the “find” operation of a Linked List:

```
1 // returns True if element exists in Linked List, otherwise False
2 find(element):
3     current = head                  // start at the head node
4     while current is not NULL:
5         if current.data == element:
6             return True
7         current = current.next      // follow forward pointer
8     return False                    // failed to find element
```

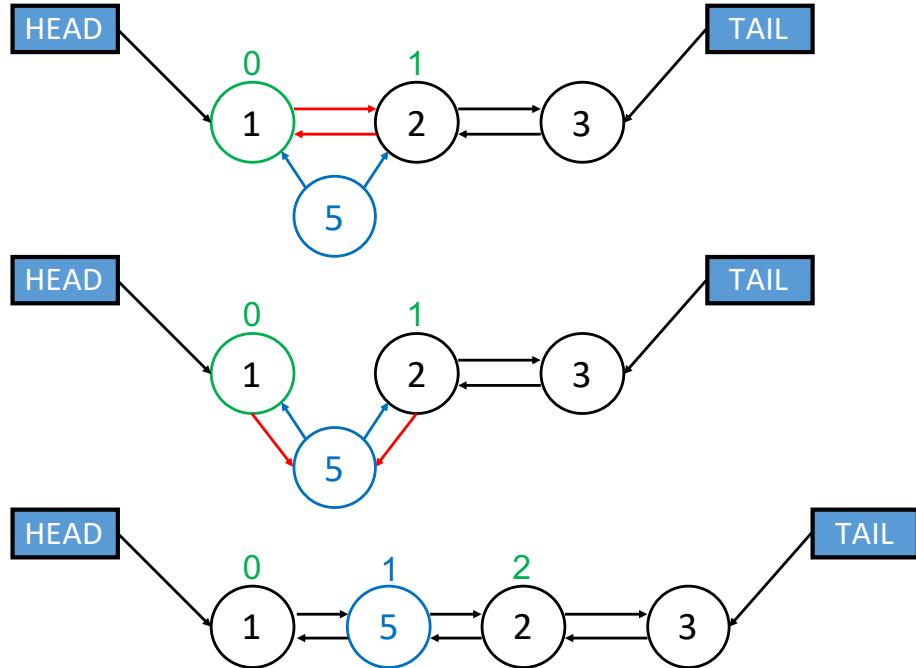
If we want to find what element is at a given “index” of the Linked List, we can perform a similar algorithm, where we start at the head node and iterate through the nodes via their forward pointers until we find the element we desire. Note that, in the following pseudocode, we use 0-based indexing.

```

1 // returns element at index of Linked List (or NULL if invalid)
2 find(index):
3     if index < 0 or index >= n: // invalid indices
4         return NULL
5     curr = head                // start at head
6     repeat index times:       // move forward index times
7         curr = curr.next
8     return curr

```

The “insert” algorithm is almost identical to the “find” algorithm: you first execute the “find” algorithm just like before, but once you find the insertion site, you rearrange pointers to fit the new node in its rightful spot. The following is an example of inserting the number 5 to index 1 of the Linked List (using 0-based counting):



Notice how we do a regular “find” operation to the index directly before index i , then we point the new node’s “next” pointer to the node that was previously at index i (and the new node’s “prev” pointer to the node that is directly before index i in the case of a Doubly-Linked List, as above), and then

we point the “next” pointer of the node before the insertion site to point to the new node (and the “prev” pointer of the node previously at index i to point to the new node in the case of a Doubly-Linked List). Because the structure of a Linked List is only based on pointers, the insertion algorithm is complete simply after changing those pointers.

The following is pseudocode for the “insert” operation of a Linked List (with added corner cases for the first and last indices):

```

1 // inserts newnode at index of Linked List
2 insert(newnode, index):
3     if index == 0:                      // beginning of list
4         newnode.next = head
5         head.prev = newnode
6         head = newnode
7     else if index == size:              // end of list
8         newnode.prev = tail
9         tail.next = newnode
10        tail = newnode
11    else:                            // general case
12        curr = head
13        repeat index-1 times:        // move curr to directly
14            curr = curr.next          // before insertion site
15        newnode.next = curr.next    // update the pointers
16        newnode.prev = curr
17        curr.next = newnode
18        newnode.next.prev = newnode
19    size = size + 1                  // increment size

```

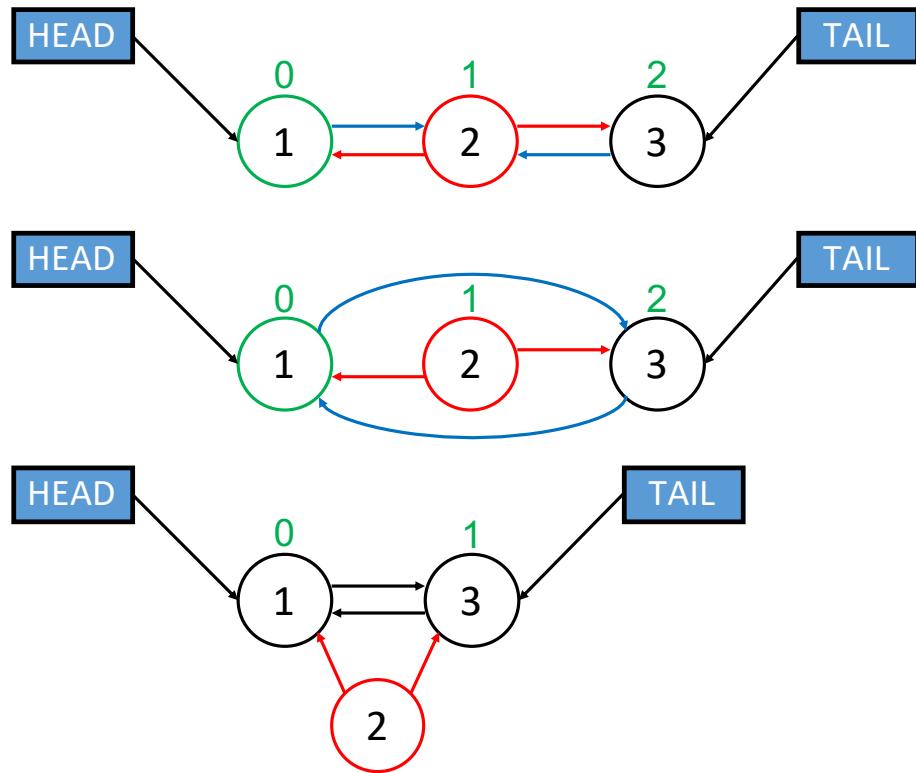
The “remove” algorithm is also almost identical to the “find” algorithm: you first execute the “find” algorithm just like before, but once you find the insertion site, you rearrange pointers to remove the node of interest. The following is pseudocode for the “remove” operation of a Linked List (with added corner cases for the first and last indices):

```

1 // removes the element at index of Linked List
2 remove(index):
3     if index == 0:                      // beginning of list
4         head = head.next
5         head.prev = NULL
6     else if index == n:                // end of list
7         tail = tail.prev
8         tail.next = NULL
9     else:                            // general case
10        curr = head
11        repeat index-1 times:        // move curr to directly
12            curr = curr.next          // before removal site
13        curr.next = curr.next.next // update the pointers
14        curr.next.prev = curr
15    n = n - 1                        // decrement n

```

The following is an example of removing the element at index 1 of the Linked List (using 0-based counting):



STOP and Think

Notice that the node we removed still exists in this diagram. Not considering memory management (i.e., only thinking in terms of data structure functionality), is this an issue?

In summation, Linked Lists are great (constant-time) when we add or remove elements from the beginning or the end of the list, but finding elements in a Linked List (even one in which elements are sorted) cannot be optimized like it can in an Array List, so we are stuck with $\mathcal{O}(n)$ “find” operations. Also, recall that, with Array Lists, we needed to allocate extra space to avoid having to recreate the backing array repeatedly, but because of the dynamic allocation of memory for new nodes in a Linked List, we have no wasted memory here.

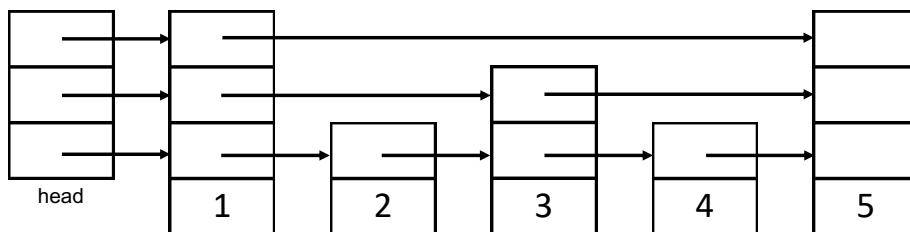
Array-based and Linked data structures are two basic starting points for many more complicated data structures. Because one strategy does not entirely dominate the other (i.e., they both have their pros and cons), you must analyze each situation to see which approach would be better.

2.3 Skip Lists

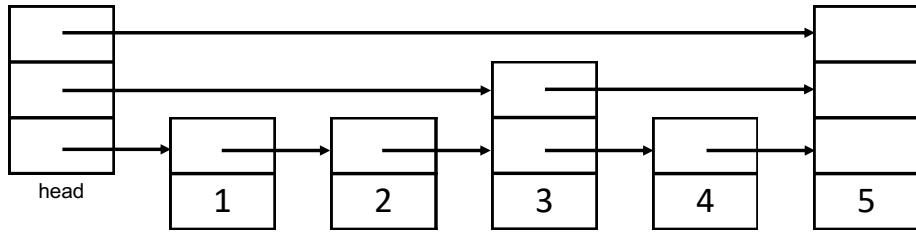
We have thus far discussed two basic data structures: the Array List and the Linked List. We saw that we could obtain worst-case $\mathcal{O}(\log n)$ find operations with an Array List if we kept it sorted and used binary search, but insert and remove operations would always be $\mathcal{O}(n)$. Also, to avoid having to keep rebuilding the backing array, we had to allocate extra space, which was wasteful. We also saw that, with a Linked List, we could obtain worst-case $\mathcal{O}(1)$ insert and remove operations to the front or back of the structure, but finding elements would be $\mathcal{O}(n)$, even if we were to keep the elements sorted. This is because Linked Lists lack the random access property that Array Lists have. Also, because each node in a Linked List is created on-the-fly, we don't have to waste extra space like we did with the Array List.

Is there any way for us to somehow reap the benefits of both data structures? In 1989, computer scientist William Pugh invented the **Skip List**, a data structure that expands on the Linked List and uses extra forward pointers with some random number generation to simulate the binary search algorithm achievable in Array Lists.

A Skip List is effectively the same as a Linked List, except every node in the Skip List has *multiple layers*, where each layer of a node is a forward pointer. For our purposes, we will denote the number of layers a node reaches as its *height*. The very bottom layer is exactly a regular Linked List, and each higher layer acts as an “express lane” for the layers below. Also, the elements in a Skip List must be **sorted**. The sorted property of the elements in a Skip List will let us perform a find algorithm that is functionally similar to binary search. The following is an example of a Skip List:



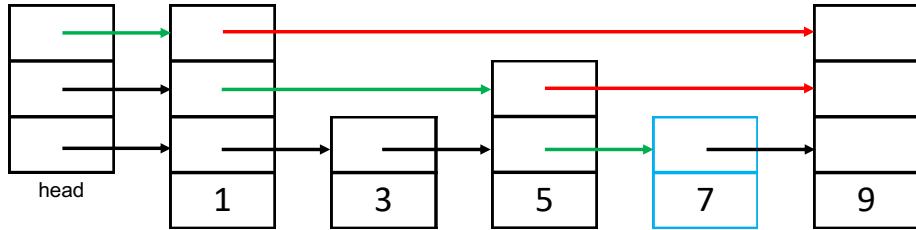
Just like with a Linked List, we have a *head*. However, because Skip Lists have *multiple layers*, the head also has multiple layers. Specifically, for each layer i , the i -th pointer in head points to the first node that has a height of i . In the example above, the first node (1) happens to reach every layer, so each pointer in head points to 1, but this does not have to be the case. Take the following example:



Now, the three pointers in *head* point to three different nodes. For our purposes, we will number the bottom layer as layer 0, the next layer as layer 1, etc. In this example, *head*'s pointer in layer 0 points to node 1, its pointer in layer 1 points to node 3, and its pointer in layer 2 points to node 5. Also, nodes 1, 2, and 4 have heights of 0, node 3 has a height of 1, and node 5 has a height of 2.

To **find** an element *e* in a Skip List, we start our list traversal at *head*, and we start at the highest layer. When we are on a given layer *i*, we traverse (i.e., actually move forward on) the forward pointers on layer *i* until *just before* we reach a node that is larger than *e* or until there are no more forward pointers on level *i*. Once we reach this point, we move down one layer and continue the search. If *e* exists in our Skip List, we will eventually find it (because the bottom layer is a regular Linked List, so we would step through the elements one-by-one until we find *e*). Otherwise, if we reach a point where we want to move down one layer but we're already on layer 0, *e* does not exist in the Skip List.

The following is an example in which we attempt to find the element 7. Red arrows denote pointers that we *could* have traversed, but that would have taken us to a node that was too big (so we instead chose to go down 1 level), and green arrows denote pointers we actually took. We define both pointers that we *could* have traversed and pointers that we *actually took* as “followed.”



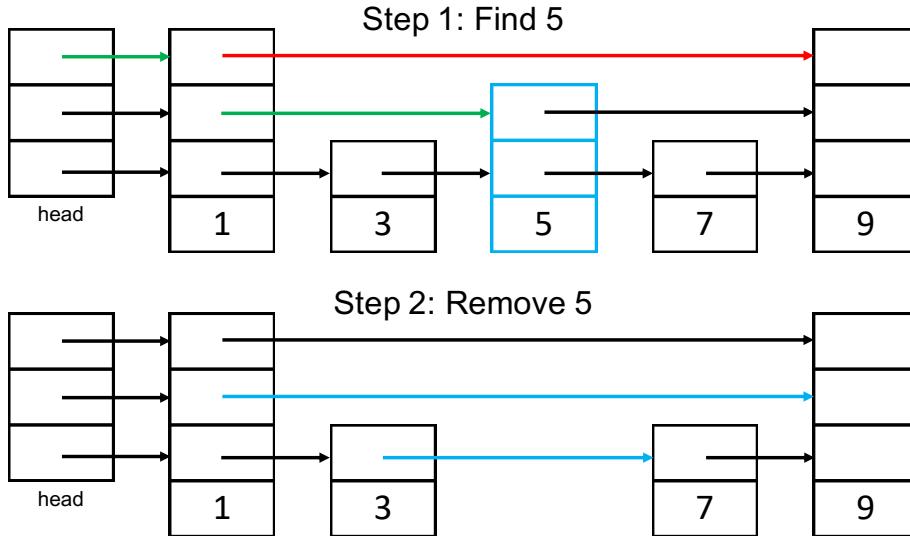
The following is formal pseudocode to describe the “find” algorithm. In the pseudocode, **head** is *head* and **head.height** is the highest layer in *head* (which is the highest layer in the Skip List, by definition). Also, for a given node *c*, *c.next* is a list of forward-pointers, where *c.next[i]* is the forward-pointer at layer *i*.

```

1 // returns True if element exists in Skip List , otherwise False
2 find(element):
3     c = head
4     L = head.height
5     while L >= 0:           // can't go negative
6         if c.key == element: // we found element
7             return True
8         if c.next[L] is NULL or c.next[L].key > element:
9             L = L - 1          // drop one layer
10        else:
11            c = c.next[L]
12    return False           // we failed on every layer

```

To **remove** an element from a Skip List, we simply perform the “find” algorithm to find the node we wish to remove, which we will call *node*. Then, for each layer *i* that *node* reaches, whatever is pointing to *node* on layer *i* should instead point to whatever *node* points to on layer *i*. The following is an example in which we remove 5 from the given Skip List. In the “Step 1: Find” figure, red arrows denote pointers that we *could* have traversed, but that would have taken us to a node that was too big (so we instead chose to go down 1 level), and green arrows denote pointers we actually took. In the “Step 2: Remove” figure, blue arrows denote pointers that were updated to actually perform the removal.



The following is formal pseudocode to describe the “remove” algorithm. In the pseudocode, *head* is *head* and *head.height* is highest layer in *head* (which is the highest layer in the Skip List, by definition). Also, for a given node *c*, *c.next* is a list of forward-pointers, where *c.next[i]* is the forward-pointer at layer *i*, and *c.prev* is a list of reverse-pointers, where *c.prev[i]* is a pointer

to the node that points to c on layer i .

```

1 // removes element if it exists in the list
2 remove(element):
3     c = head
4     L = head.height
5     while L >= 0:           // can't go negative
6         if c.next[L] is NULL or c.next[L].key > element:
7             L = L - 1        // drop one layer
8         else:
9             c = c.next[L]
10            if c.key == element: // we found element, so break
11                break
12            for i from 0 to L:    // fix pointers
13                c.prev[i].next[i] = c.next[i]
```

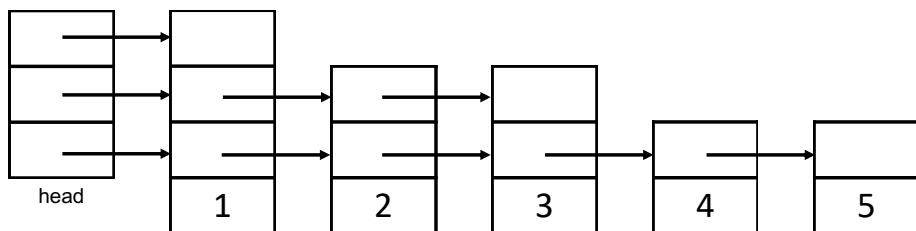
Exercise Break

What is the worst-case time complexity to find or remove elements from a Skip List?

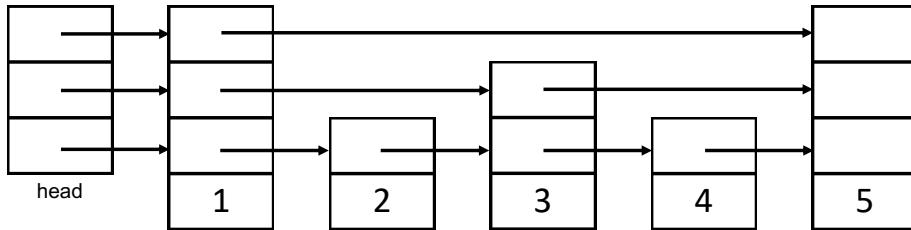
Exercise Break

Assuming the heights of the nodes in a Skip List are optimally-distributed (i.e., each “jump” allows you to traverse half of the remainder of the list), what is the time complexity to find or remove elements from a Skip List?

Hopefully the previous questions made you think about how the distribution of *heights* in a Skip List affects the *performance* of the Skip List. To emphasize this thought, look at the following Skip List, and notice how using it is no faster than using an ordinary Linked List because we have to potentially iterate over all n elements in the worst case:



For example, finding node 5 would require following 5 pointers (i.e., 5 node traversals) and finding node 3 would require following 3 pointers (i.e., 3 node traversals). However, this next example of a Skip List has a better distribution of heights, meaning we check less elements to find the one we want:

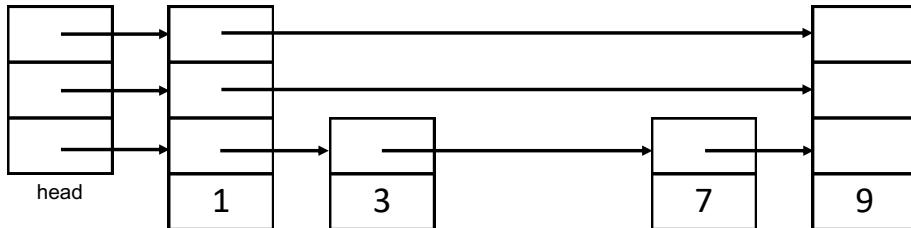


For example, finding node 5 would require following 2 pointers (i.e., 2 node traversals) and finding node 3 would require following 3 pointers (i.e., 2 node traversals). Remember, the amount of pointers we end up following is not always equivalent to the amount of nodes we end up traversing. This is because we sometimes need to “follow” a pointer to check to see if we should even traverse to that node in the first place.

Clearly, the distribution of heights has a significant impact on the performance of a Skip List. With the best distribution of heights, the Skip List “find” algorithm effectively performs binary search, resulting in a $\mathcal{O}(\log n)$ time complexity. Thus, we must ask ourselves the following question: How can we design our Skip List such that heights are automatically “well-distributed”?

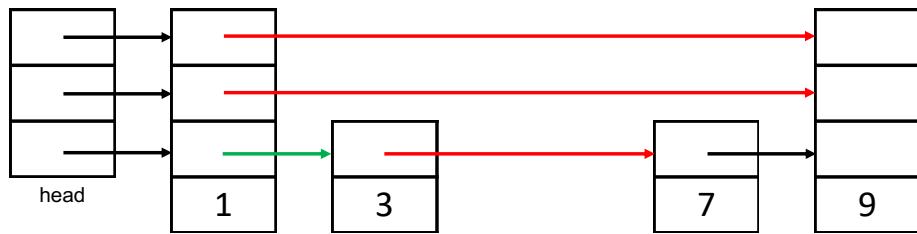
The task of optimizing the distribution of heights in a Skip List falls under the *insert* operation. We do so by first performing the regular “find” algorithm to find where we will insert our new element. Then, we determine our new node’s height. By definition, the new node must have a height of at least 0 (i.e., the 0-th layer). So how many layers higher should we build the new node? To answer this question, we will play a simple coin-flip game (we will shortly explain what this game formally represents): starting at our base height of 0, we flip a coin, where the coin’s probability of heads is p . If we flip heads, we increase our height by 1. If we flip tails, we stop playing the game and keep our current height.

Say, for example, we wish to insert 5 into the following Skip List:

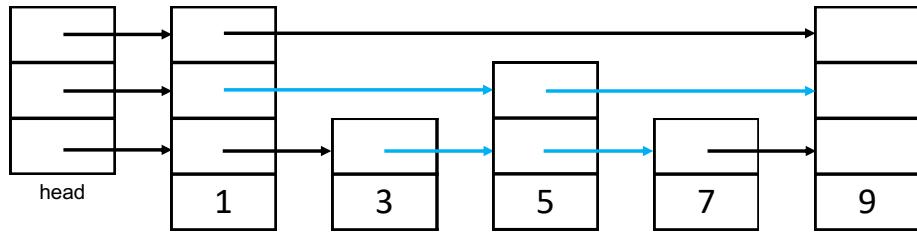


First, we perform the regular “find” algorithm to find the insertion site for 5. In the following figure, red arrows denote pointers that we *could* have traversed, but that would have taken us to a node that was too big (so we instead chose to go down 1 level), and green arrows denote pointers we actually took. As you hopefully inferred, the red arrows are the only ones that we might have to fix

upon insertion.



Now that we have found our insertion site, we must determine the height of our new node. We know that the coin must have a height of at least 0, so we start our height at 0. Then, we flip a coin where the probability of heads is p . Let's say we flipped heads (with probability p): we now increase our height from 0 to 1. Then, we flip the coin again. This time, let's say we flipped tails (with probability $1 - p$): we stop playing the game and keep our height of 1. We perform the insertion with this new node by simply updating two pointers: one on layer 0 and one on layer 1.



The following is formal pseudocode to describe the “insert” algorithm. In the pseudocode, `head` is *head* and `head.height` is highest layer in *head* (which is the highest layer in the Skip List, by definition). Also, for a given node *c*, `c.next` is a list of forward-pointers, where `c.next[i]` is the forward-pointer at layer *i*. Lastly, note that the probability of coin-flip success, p , must be a variable defined in the Skip List itself (we refer to it as *p*).

```

1 // inserts element if it doesn't exist in the list
2 insert(element):
3     c = head
4     L = head.height
5     toFix = empty list of nodes of length head.height + 1
6     while L >= 0: // can't go negative
7         if c.next[L] is NULL or c.next[L].key > element:
8             toFix[L] = current // might have to fix pointer here
9             L = L - 1 // drop one layer
10        else:
11            c = c.next[L]
12            if c.key == element: // if we found element,
13                return // exit (no duplicates)
14
15 // if we reached here, we can perform the insertion
16 newNode = new node containing element (height = 0)
17 while newNode.height < head.height:
18     result = result of coin flip with probability p of heads
19     if result is heads:
20         newNode.height = newNode.height + 1
21     else: // tails, so keep current height
22         break
23     for i from 0 to newNode.height: // fix pointers
24         newNode.next[i] = toFix[i].next[i]
25         toFix[i].next[i] = newNode

```

For those of you who are interested in probability and statistics, it turns out that we don't have to do multiple coin-flips: we can determine the height of a new node in a single trial. A coin-flip is a **Bernoulli distribution** (or a **binary distribution**), which is just the formal name for a probability distribution in which we only have two possible outcomes: *success* and *failure* (which we often call *heads* and *tails*, *yes* and *no*, etc.). We say that p is the probability of *success*, and we say that $q = 1 - p$ is the probability of *failure*.

What we described above, where we perform multiple coin-flips until we get our first tails, is synonymous to saying "Sample from a Bernoulli distribution until you see the first failure." It turns out that this statement is actually a probability distribution in itself: the **Geometric distribution**. The Geometric distribution tells us, given a Bernoulli distribution with probability p of success, what is the probability that our k -th trial (i.e., our k -th coin-flip) results in the *last failure in a row before the first success*? Formally, for a Geometric random variable X , we say that $Pr(X = k) = (1 - p)^k p$: we need k failures (each with probability $1 - p$), and then we need one success (with probability p).

But wait! The Geometric distribution tells us about the number of flips until the first *success*, but we want the number of flips until the first *failure*! It turns out that we can trivially modify what we just said to make the general definition of a Geometric distribution work for us. Recall that *success* and *failure* are just two events, where the probability of *success* is p and the probability of *failure* is $q = 1 - p$. If we simply swap them when we plug them into the Geometric distribution, we will have swapped *success* and *failure*, so the resulting Geometric distribution would describe the number of trials until the

first *failure*. Formally, $\Pr(X = k) = p^k(1 - p)$: we need k successes (each with probability p), and then we need one failure (with probability $1 - p$).

Therefore, if we were to sample from a Geometric distribution following $\Pr(X = k) = p^k(1 - p)$, we would get a random value for k , the number of coin-flips *before* the first *failure*, which is exactly our new node's height!

If this probability and statistics detour didn't make much sense to you, that's perfectly fine. Performing separate individual coin-flips, each with probability p of success, vs. performing a single trial from the Geometric distribution above are both completely mathematically equivalent. It's just an interesting connection we wanted to make for those of you who were curious.

Exercise Break

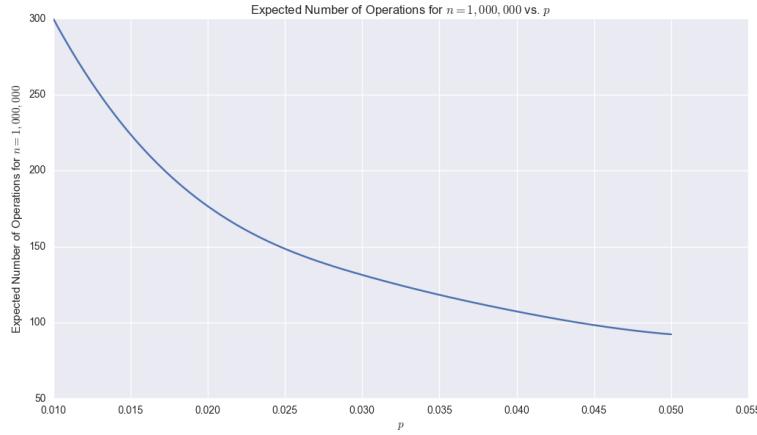
To determine the heights of new nodes, you use a coin with a probability of success of $p = 0.3$. What is the probability that a new node will have a height of 0?

Exercise Break

To determine the heights of new nodes, you use a coin with a probability of success of $p = 0.3$. What is the probability that a new node will have a height of 2?

Of course, as we mentioned before, the *worst-case* time complexity to find, insert, or remove an element in a Skip List is $\mathcal{O}(n)$: if the heights of the nodes in the list are distributed poorly, we will effectively have a regular Linked List, and we will have to iterate through the elements of the list one-by-one. However, it can be formally proven (via a proof that is a bit out-of-scope) that, with “smart choices” for p and the maximum height of any given node, the expected number of comparisons that must be done for a “find” operation is $1 + \frac{1}{p} \log_{\frac{1}{p}} n + \frac{1}{1-p}$, meaning that the **average-case** time complexity of a Skip List is $\mathcal{O}(\log n)$.

The term “smart choices” is vague, so we will attempt to be a bit more specific. In the paragraph above, we mentioned that the expected number of comparisons that must be done for a “find” operation is $1 + \frac{1}{p} \log_{\frac{1}{p}} n + \frac{1}{1-p}$. However, as p increases, the amount of space we need to use to store pointers also increases. Therefore, when we pick a value for p to use in our Skip List, assuming we have a ballpark idea of how large n will be, we should choose the smallest value of p that results in a reasonable expected number of operations. For example, the following is a plot of $y = \frac{1}{p} \log_{\frac{1}{p}} n$ for $n = 1,000,000$:



The curve begins to flatten at around $p = 0.025$, so that might be a good value to pick for p . Once we've chosen a value for p , it can be formally proven (via a proof that is a bit out-of-scope) that, for good performance, the maximum height of the Skip List should be no smaller than $\log_{\frac{1}{p}} n$.

Exercise Break

Imagine you are implementing a Skip List, and you chose a value for $p = 0.1$. If you are expecting that you will be inserting roughly $n = 1,000$ elements, what should you pick as your Skip List's maximum height?

In summation, we have now learned about the Skip List, a data structure based off of the Linked List that uses random number generation to mimic the binary search algorithm of a sorted Array List in order to achieve an **average-case** time complexity of $\mathcal{O}(\log n)$. Also, we learned that, in order to achieve this average-case time complexity, we need to be smart about choosing a value of p : if p is too small or too big, we will effectively just have a regular Linked List, and as p grows, the memory usage of our Skip List grows with it. Once we have chosen a value of p , we should choose the maximum height of our Skip List to be roughly $\log_{\frac{1}{p}} n$.

In this section, we discussed a modified Linked List data structure that mimicked some properties of an Array List, and in the next section, we will do the reverse: we will discuss a modified Array List data structure that mimics some properties of a Linked List: the Circular Array.

2.4 Circular Arrays

Imagine we have a set of elements we want to store in some data structure. The elements don't necessarily have a particular order (they might, but we don't

know in advance). In the previous sections of the text, we learned about two data structures we could theoretically use to store our elements: Array Lists and Linked Lists.

We learned that Array Lists are great if we want to be able to randomly access elements *anywhere* in my dataset. In terms of inserting elements, however, they do fine if we're always adding to the end of the list. If we want to insert elements at the beginning of the list however, we have to waste time moving other elements to make room for the insertion. Linked Lists, on the other hand, are great at inserting elements to the beginning or end of the list. If we want to access elements in the middle however, we have to waste time iterating through the elements until we reach the desired element.

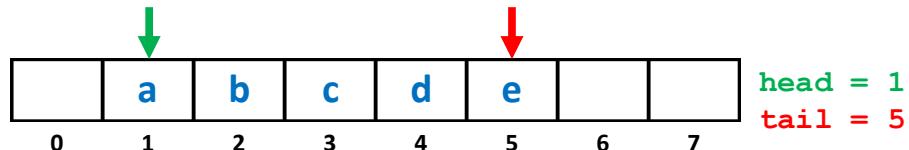
Is there any way we can create a new data structure that allows us to enjoy some of the perks of both data structures? In this section, we will discuss the **Circular Array**, our attempt at reaping the benefits of both Array Lists and Linked Lists.

At the lowest level, a Circular Array is really just a regular Array List with a clever implementation: we will try to mimic the implementation of a Linked List. Recall that a Linked List has a *head* pointer and a *tail* pointer, and when we want to insert an element to the beginning or end of a Linked List, all we do is update a few pointers (which is a constant-time operation).

What if, similar to *head* and *tail* pointers in a Linked List, we take our Array List and use *head* and *tail* indices? The first element would be the element at the *head* index, and the last element would be the element at the *tail* index. As we add to the end of the Circular Array, we can simply increment *tail*, and likewise, as we add to the front of the Circular Array, we can simply decrement *head*. As we increment *tail*, if we ever go out of bounds (i.e., *tail* becomes equal to the size of the array), we can simply wrap around to index 0. Likewise, as we're decrementing *head*, if we ever go out of bounds (i.e., *head* becomes -1), we can simply wrap around to the last index of the array (i.e., *size*-1).

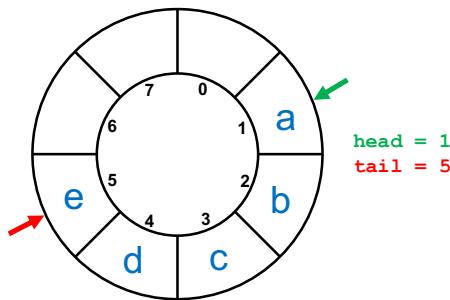
Note that, just like with an Array List, the elements of our Circular Array will be contiguous in the backing array. The reason for this is that we will only allow users to insert to the *head* or the *tail* of the Circular Array.

The following is an example of a Circular Array containing 5 elements but with a capacity of 8 elements. The backing array's indices are written below the backing array in black, and the *head* and *tail* indices have been written and are also drawn onto the figure with arrows.



When adding/removing elements, we only really care about elements' locations with respect to the *head* and *tail* indices, not with respect to the actual

indices in the backing array. Because of this, as well as because of the fact that the “wrap around” performed by both *head* and *tail* is a constant-time operation, many people prefer to picture the Circular Array as exactly that: a circular array. The following is a circularized representation of the same example shown above.



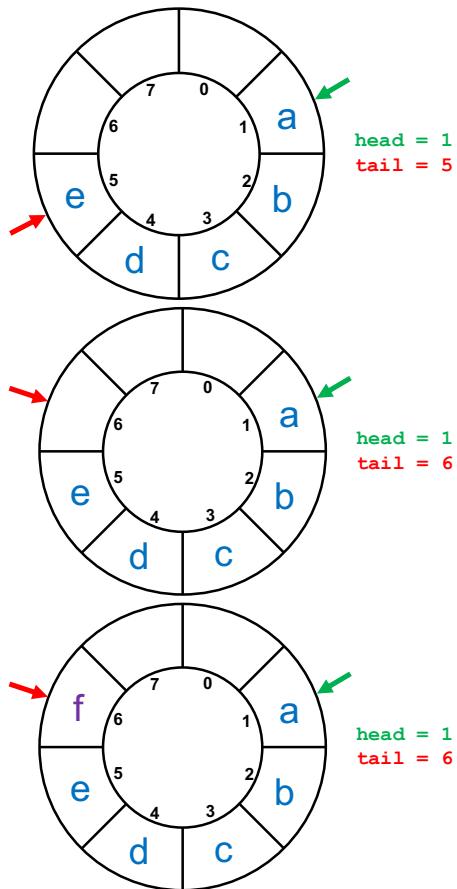
You can choose either representation (linear or circular) to visualize the Circular Array in your own mind: both representations are equally valid. The linear representation is closer to the implementation “truth,” whereas the circular representation better highlights the purpose of the structure.

Exercise Break

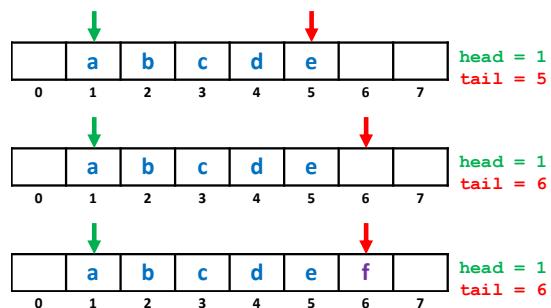
You are given an arbitrary Circular Array with a capacity of 8 that contains 3 elements. Which of the following pairs of *head* and *tail* indices could be valid? Assume indices are 0-based (i.e., the first index of the backing array is 0 and the last index of the backing array is 7).

- *head* = 0 and *tail* = 6
- *head* = 5 and *tail* = 7
- *head* = 6 and *tail* = 0
- *head* = 0 and *tail* = 2
- *head* = 3 and *tail* = 7
- *head* = 7 and *tail* = 1
- *head* = 1 and *tail* = 7

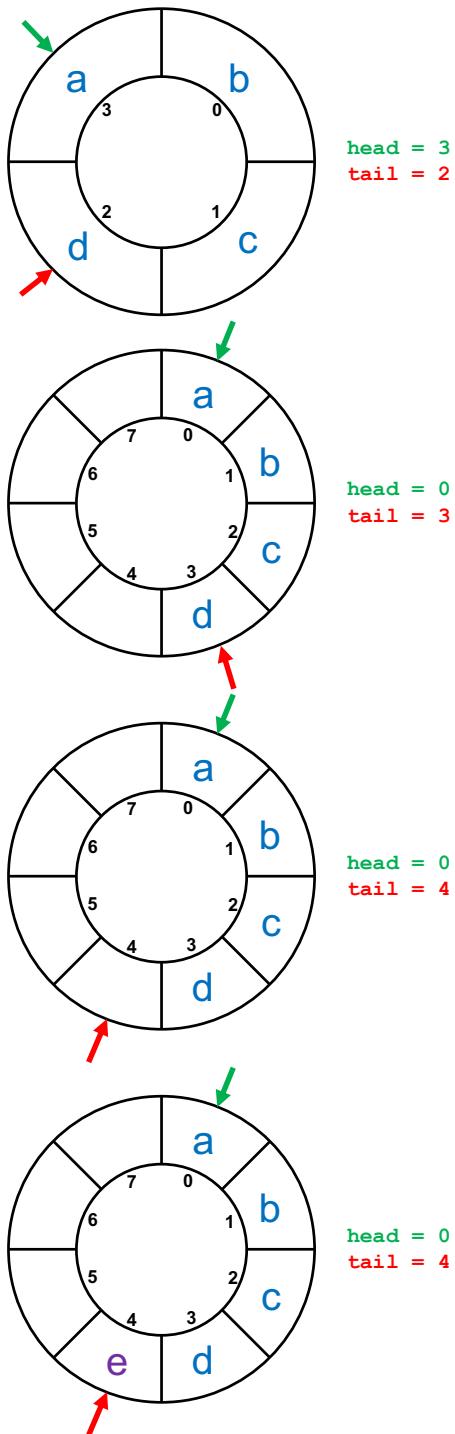
The following is an example of insertion at the end of a Circular Array with some elements already in it:



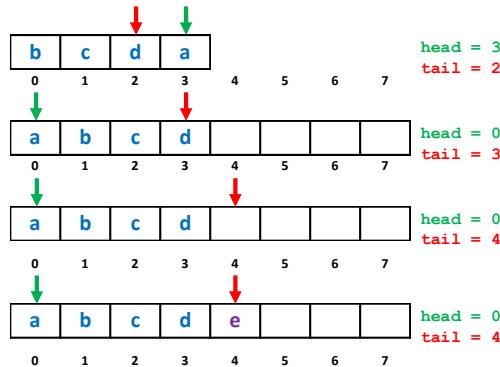
The following is the same example, but in the linear representation:



Note that, just like with Array Lists, if the backing array becomes full, we can create a new backing array (typically of twice the size) and simply copy all elements from the old array into the new one. The following is an example:



The following is the same example, but in the linear representation:



We briefly discussed the insertion algorithm at a higher level, but let's formalize it. In the following pseudocode, assume our Circular Array has a backing array named `array` (which has length `array.length`), an integer `n` that keeps track of the number of elements added to the Circular Array, and two indices `head` and `tail`. For our purposes, let's assume you can only add elements to the front or back of the Circular Array to keep the problem relatively simple.

```

1 // inserts element at the front of the Circular Array
2 insertFront(element):
3     if n == array.length:          // if array is full,
4         newArray = empty array of length 2*array.length
5         for i from 0 to n-1:       // copy all elements
6             newArray[i] = array[(head+i)%array.length]
7         array = newArray           // replace array with newArray
8         head = 0 and tail = n-1 // fix head and tail indices
9         // insertion algorithm
10        head = head - 1         // decrement head index
11        if head == -1:          // if out of bounds, wrap around
12            head = array.length-1
13        array[head] = element    // perform insertion
14        n = n + 1                // increment size



---


1 // inserts element at the back of the Circular Array
2 insertBack(element):
3     if n == array.length:          // if array is full,
4         newArray = empty array of length 2*array.length
5         for i from 0 to n-1:       // copy all elements
6             newArray[i] = array[(head+i)%array.length]
7         array = newArray           // replace array with newArray
8         head = 0 and tail = n-1 // fix head and tail indices
9         // insertion algorithm
10        tail = tail + 1          // increment tail index
11        if tail == array.length: // if out of bounds, wrap around
12            tail = 0
13        array[tail] = element    // perform insertion
14        n = n + 1                // increment size

```

STOP and Think

How could we generalize this idea to allow for insertion into the middle of the Circular Array?

Exercise Break

What is the worst-case time complexity for an “insert” operation at the front or back of a Circular Array?

Exercise Break

What is the worst-case time complexity for an “insert” operation at the front or back of a Circular Array, given that the backing array is not full?

Exercise Break

What is the worst-case time complexity for an “remove” operation at the front or back of a Circular Array?

Removal at the front or back of a Circular Array is fairly trivial. To remove from the front of a Circular Array, simply “erase” the element at the *head* index, and then increment the *head* index (wrapping around, if need be). To remove from the back of a Circular Array, simply “erase” the element at the *tail* index, and then decrement the *tail* index (wrapping around, if need be).

```
1 // removes the element at the front of the Circular Array
2 removeFront():
3     erase array[head]
4     head = head + 1
5     if head == array.length:
6         head = 0
7     n = n - 1
```

```
1 // removes the element at the back of the Circular Array
2 removeBack():
3     erase array[tail]
4     tail = tail - 1
5     if tail == -1:
6         tail = array.length - 1
7     n = n - 1
```

STOP and Think

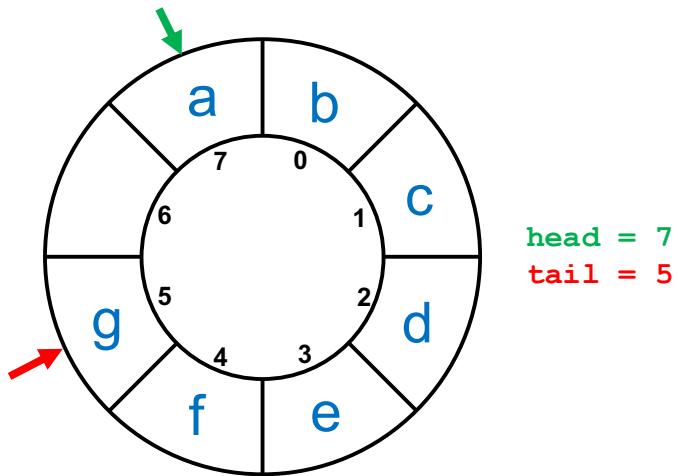
Is it necessary for us to perform the “erase” steps in these algorithms? What would happen if we didn’t?

So far, we’ve only looked at adding or removing elements from the front or back of a Circular Array, which is no better than adding or removing elements

from the front or back of a Linked List. So what was the point of all this added complexity? Recall that, with a Linked List, if we want to access an element somewhere in the middle of the list, even if we know which index of our list we want to access, we need to perform an $\mathcal{O}(n)$ iteration to follow pointers and reach our destination. However, with a Circular Array, the backing structure is an Array List, meaning we have random access to any element given that we know which index of the backing array we need to query.

If we want to access the element of a Circular Array at index i , where i is with respect to the *head* index (i.e., *head* is $i = 0$, irrespective of what index it is in the backing array), we can simply access the element at index $(\text{head} + i) \% \text{array.length}$ in the backing array. By modding by the backing array's length, we ensure that the index we get from the operation $(\text{head} + i)$ wraps around to a valid index if it exceeds the boundaries of the array.

The following is an example of a Circular Array where the head index wrapped around. Notice that, if we consider *head* to be “ $i = 0$ ” of our elements (even though it's index 7 of the backing array), we can access any i -th element of our list via the formula above. For example, the element at $i = 2$ of our list is at index $(7+2)\%8 = 9\%8 = 1$ of the backing array, which corresponds to c.



STOP and Think

Under what condition(s) would we be safe to omit the mod operation in the formula above? In other words, under what condition(s) would the formula $\text{head} + i$ be valid?

Previously, we discussed Array Lists and Linked Lists, and now, we've also introduced and analyzed the Circular Array, a data structure that is based on an Array List, but that attempts to mimic the efficient front/back insertion/removal properties of a Linked List. In the next sections, we will explore some

real-life applications in which we only care about adding/removing elements from the front/back of a list, which will give purpose to our exploration of Linked Lists and Circular Arrays.

2.5 Abstract Data Types

As programmers, we are often given a task in non-technical terms, and it is up to us to figure out how to actually implement the higher-level idea. How do we choose what tools to use to perform this task? Typically, the job at hand has certain requirements, so we scour the documentation of our favorite language (or alternatively just search on Google and pick the first StackOverflow result, who are we kidding) and pick something that has the functions that satisfy our job's requirements.

For example, say we want to store the grades of numerous students in a class in a way that we can query the data structure with a student's name and have it return the student's grade. Clearly, some type of a "map" data structure, which maps "keys" (student names) to "values" (grades), would do the trick. If we're not too worried about performance, we don't necessarily care about *how* the data structure executes these tasks: we just care that it gets the job done.

What we are describing, a model for data types where the data type is defined by its *behavior* from the point of view of a *user* of the data (i.e., by what functions the user claims it needs to have) is an **Abstract Data Type (ADT)**.

We just introduced a term, "abstract data type," but this course is about "data structures," so what's the difference? The two terms sound so similar, but they actually have very different implications. As I mentioned, an Abstract Data Type is defined by what functions it should be able to perform, but it does not at all depend on how it actually goes about doing those functions (i.e., it is not implementation-specific). A **Data Structure**, on the other hand, is a concrete representation of data: it consists of all of the nuts and bolts behind how the data are represented (how they are stored in memory, algorithms for performing various operations, etc.).

For example, say a user wants us to design a "set" container to contain integers. The user decide that this container should have the following functions:

- An "insert" function that will return True upon insertion of a new element, or False if the element already exists
- A "find" function that will return True if an element exists, otherwise False
- A "union" function that will add all of the elements from another "set" into this "set"

The user has successfully described what functionality the "set" container will have, without any implementation-specific details (i.e., *how* those three

functions described will work). Consequently, this “set” container is an Abstract Data Type. It is now up to us, the programmers, to figure out a way to *actually* implement this Abstract Data Type.

Can we figure out the time complexities of any of the three functions described in the “set” container above? Unfortunately, no! An Abstract Data Type *only* describes functionality, not implementation, the time complexities of these functions completely depends on how the programmer chooses to *implement* the “set” (i.e., what Data Structure we choose to use to back this container).

Exercise Break

Which of the following statements are true about an Abstract Data Type?

- Any implementations of an Abstract Data Type have a strict set of functions they must support
- An Abstract Data Type is designed from the perspective of a user, not an implementer
- An Abstract Data Type contains details on how it should be implemented

In short, an Abstract Data Type simply *describes* a set of features, and based on the features we wish to have, we need to choose an appropriate Data Structure to use as the backbone to implement the ADT.

For example, what if a user wanted an ADT to store a list of songs? When we listen to music, do we always necessarily want to iterate through the songs in the order in which they appear? Typically, we like having the ability to choose a specific song, which could appear somewhere in the middle of the list of songs. As a result, it might make sense for us, the programmers, to use an array-based structure to implement this ADT because having random access could prove useful. However, we very well could use a Linked List to implement this ADT instead! The functionality would still be correct, but it might not be as fast as using an array-based structure.

In the next sections, we will discuss some fundamental Abstract Data Types and will discuss different approaches we can use to implement them (as well as the trade-offs of each approach).

2.6 Deques

In this day and age, we take web browsing for granted. The plethora of information available at our fingertips is extensive, and with web browsers, we can conveniently explore this information (or more realistically, watch funny cat videos on YouTube or find out which movie character we are via a BuzzFeed survey). As we’re browsing, a feature we typically heavily depend on is the

browser history, which allows us to scroll back and forward through pages we've recently viewed.

We can think of the browser history as a list of web-pages. As we visit new pages, they get added to the end of the list. If we go back a few pages and then go another route, we remove the pages that were previously at the end of the list. Also, if we browse for too long and this list becomes too large, web-pages from the beginning of the list are removed to save space.

To summarize, we've just listed a set of features: inserting, removing, and looking at elements from the front and back of some sort of list. What we have just described is the first Abstract Data Type (ADT) we will explore: the **Double-Ended Queue**, or **Dequeue/Deque** (pronounced “deck”).

More formally, the Deque ADT is defined by the following set of functions:

- **addFront(element)**: Add `element` to the front of the Deque
- **addBack(element)**: Add `element` to the back of the Deque
- **peekFront()**: Look at the `element` at the front of the Deque
- **peekBack()**: Look at the `element` at the back of the Deque
- **removeFront()**: Remove the `element` at the front of the Deque
- **removeBack()**: Remove the `element` at the back of the Deque

STOP and Think

Do these functions remind us of any data structures we've learned about?

The following is an example adding and removing elements in a Deque. Note that we make no assumptions about the implementation specifics of the Deque because the Deque is an ADT. In the example, the “front” of the Deque is the left side, and the “back” of the Deque is the right side.

Initialize deque	DEQUE: <empty>
AddBack A	DEQUE: A
AddBack B	DEQUE: A B
AddFront C	DEQUE: C A B
RemoveBack	DEQUE: C A
RemoveFront	DEQUE: A
AddBack D	DEQUE: A D
RemoveBack	DEQUE: A
RemoveFront	DEQUE: <empty>

Based on what we have learned so far in this text, there are two approaches we could take that are quite good for implementing a Deque: using a Linked List or using a Circular Array. As you should recall, both data structures have good performance when accessing/modifying elements at their front/back, so how can we make a decision about which of the two we would want to use to implement a Deque?

If we were to choose a Linked List (a *Doubly*-Linked List, specifically), because we would have direct access to both the *head* and *tail* nodes and because inserting/removing elements at the front and back of a Doubly-Linked List can be reduced to constant-time pointer rearrangements, we would be guaranteed $\mathcal{O}(1)$ time for all six of the Deque operations described previously, no matter what. However, if we were to want access to the middle elements of our Deque, a Doubly-Linked List would require a $\mathcal{O}(n)$ “find” operation, even if we know exactly which index in our Deque we want to access.

If we were to choose a Circular Array, because we have direct access to both the *head* and *tail* indices and because the backing array has constant-time access to any element, the “find” and “remove” operations of the Deque are guaranteed to have $\mathcal{O}(1)$ time. However, even though inserting into the Circular Array usually has $\mathcal{O}(1)$ time, recall that, when the backing array is completely full, we need to create a new backing array and copy all of the elements over, which results in a $\mathcal{O}(n)$ worst-case time complexity. Also, if we were to want access to the middle elements of our Deque, if we knew exactly which index in our Deque we wanted to access, the random access property of an array would allow us $\mathcal{O}(1)$ time access.

Exercise Break

If we only care about adding/removing/viewing elements in the front or back of a Deque (and not at all in the middle), which of the two implementation approaches we discussed would be the better choice?

In summation, assuming we don’t care too much about access to the elements in the middle of the Deque (which is typically the case), our best option is to use a Linked List to implement our Deque. In doing so, we can achieve $\mathcal{O}(1)$ time for all six of the Deque functions we described previously without wasting any extra memory leaving space for future insertions (which we do with an Array List).

We already discussed how Deques can be used to implement browser history functions of web browsers, but it turns out that Deques can also be useful in implementing *other* Abstract Data Types. In the next sections, we will discuss two other very useful ADTs that we can use a Deque to implement.

2.7 Queues

The next Abstract Data Type we will discuss is the **Queue**. If you've ever gone grocery shopping or waited in any type of a line (which is actually called a "queue" colloquially in British English), you've experienced a Queue. Just like a grocery store line, we add elements to the back of the Queue and we remove elements from the front of the Queue. In other words, the *first* element to come out of the Queue is the *first* element that went into the Queue. Because of this, the Queue is considered a "**First In, First Out**" (**FIFO**) data type.

Formally, a Queue is defined by the following functions:

- **enqueue(element)**: Add **element** to the back of the Queue
- **peek()**: Look at the element at the front of the Queue
- **dequeue()**: Remove the element at the front of the Queue

STOP and Think

Do these functions remind us of an Abstract Data Type we've learned about?

As you should have hopefully inferred, we can use a Deque to implement a Queue: if we implement a Queue using a Deque as our backing structure (where the Deque would have its own backing structure of either a Doubly-Linked List or a Circular Array, because as you should recall, a Deque is also an ADT), we can simply reuse the functions of a Deque to implement our Queue. For example, say we had the following Queue class in C++:

```

1  class Queue {
2      private:
3          Deque deque;
4
5      public:
6          bool enqueue(Data element);
7          Data peek();
8          void dequeue();
9          int size();
10 };

```

We could extremely trivially implement the Queue functions as follows:

```

1 bool Queue::enqueue(Data element) {
2     return deque.addBack(element);
3 }
4 Data Queue::peek() {
5     return deque.peekFront();
6 }
7 void Queue::dequeue() {
8     deque.removeFront();
9 }
10 int Queue::size() {
11     return deque.size();
12 }
```

The equivalent implementation in Python would be the following:

```

1 class Queue:
2     deque = Deque()
3     def enqueue(self, element):
4         return deque.addBack(element)
5     def peek(self):
6         return deque.peekFront()
7     def dequeue(self):
8         deque.removeFront()
9     def __len__(self):
10        return len(deque)
```

Of course, as we mentioned, the Deque itself would have some backing data structure as well, but if we use our Deque implementation to back our Queue, the Queue becomes extremely easy to implement.

Watch Out! Notice that, in our implementation of a Queue, the `dequeue()` function has a `void` return type, meaning it *removes* the element on the front of the Queue, but it does not *return* its value to us. This is purely an implementation-level detail, and in some languages (e.g. Java), the `dequeue()` function *removes and returns* the top element, but in other languages (e.g. C++), the `dequeue()` function *only removes* the front element without returning it, just like in our implementation.

STOP and Think

In our implementation of a Queue, we chose to use the `addBack()`, `peekFront()`, and `removeFront()` functions of the backing Deque. Could we have chosen `addFront()`, `peekBack()`, and `removeBack()` instead? Why or why not?

The following is an example in which we enqueue and dequeue elements in a Queue. Note that we make no assumptions about the implementation specifics of the Queue (i.e., we don't use a Linked List nor a Circular Array to represent the Queue) because the Queue is an ADT.

Initialize queue	QUEUE: <empty>
Enqueue N	QUEUE: N
Enqueue M	QUEUE: N M
Enqueue Q	QUEUE: N M Q
Dequeue	QUEUE: M Q
Enqueue D	QUEUE: M Q D
Dequeue	QUEUE: Q D
Dequeue	QUEUE: D
Dequeue	QUEUE: <empty>

Exercise Break

What is the worst-case time complexity of adding an element into a Queue, given that we are using a Deque as our backing structure in our implementation and given that we are using a **Circular Array** as the backing structure of our Deque?

Exercise Break

What is the worst-case time complexity of adding an element into a Queue, given that we are using a Deque as our backing structure in our implementation and given that we are using a **Doubly-Linked List** as the backing structure of our Deque?

Despite its simplicity, the Queue is a very powerful ADT because of the numerous applications to which it can be applied, such as the following:

- Organizing people waiting for their turn (e.g. for a ride, for an event, for a cash register, etc.)
- Keeping track of tasks that need to be completed (in real life, in a computer scheduler, etc.)
- “Graph” exploration via the “BFS” algorithm (which will be covered in the “Graphs” section of this text)

In the next section, we will discuss another simple, yet powerful, ADT that can also be implemented using a Deque.

2.8 Stacks

The next Abstract Data Type we will discuss is the **Stack**. If you have ever “stacked” up the dishes after dinner to prepare for washing, you have (potentially unknowingly) used a Stack. When you are washing the dishes after dinner,

you add unwashed dishes to the top of the stack of dishes, and when you want to wash the next dish, you take one off from the top of the stack of dishes. With a Stack, the *first* element to come out is the *last* one to have gone in. Because of this, the Stack is considered a “**Last In, First Out**” (**LIFO**) data type.

Formally, a Stack is defined by the following functions:

- `push(element)`: Add `element` to the top of the Stack
- `top()`: Look at the element at the top of the Stack
- `pop()`: Remove the element at the top of the Stack

STOP and Think

Do these functions remind us of an Abstract Data Type we’ve learned about?

Just like with the Queue, we can use a Deque to implement a Stack: if we implement a Stack with a Deque as our backing structure (where the Deque would have its own backing structure of either a Doubly Linked List or a Circular Array, because as you should recall, a Deque is an ADT), we can again simply re-use the functions of a Deque to implement our Stack. For example, say we had the following Stack class in C++:

```

1 class Stack {
2     private:
3         Deque deque;
4
5     public:
6         bool push(Data element);
7         Data top();
8         void pop();
9         int size();
10    };

```

We could very trivially implement the Stack functions as follows:

```

1 bool Stack::push(Data element) {
2     return deque.addBack(element);
3 }
4 Data Stack::top() {
5     return deque.peekBack();
6 }
7 void Stack::pop() {
8     deque.removeBack();
9 }
10 int Stack::size() {
11     return deque.size();
12 }

```

The equivalent Python class would be the following:

```

1  class Stack:
2      deque = Deque()
3      def push(self, element):
4          return deque.addBack(element)
5      def top(self):
6          return deque.peekBack()
7      def pop(self):
8          deque.removeBack()
9      def __len__(self):
10         return len(deque)

```

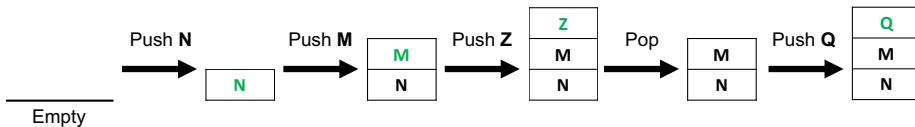
Of course, as we mentioned, the Deque itself would have some backing data structure as well, but if we use our Deque implementation to back our Stack, the Stack becomes extremely easy to implement.

Watch Out! Notice that, in our implementation of a Stack, the `pop()` function has a `void` return type, meaning it *removes* the element on the top of the Stack, but it does not *return* its value to us. This is purely an implementation-level detail, and in some languages (e.g. Java), the `pop()` function *removes and returns* the top element, but in other languages (e.g. C++), the `pop()` function *only removes* the top element without returning it, just like in our implementation.

STOP and Think

In our implementation of a Stack, we chose to use the `addBack()`, `peekBack()`, and `removeBack()` functions of the backing Deque. Could we have chosen `addFront()`, `peekFront()`, and `removeFront()` instead? Why or why not?

The following is an example in which we push and pop elements in a Stack. Note that we make no assumptions about the implementation specifics of the Stack (i.e., we don't use a Linked List nor a Circular Array to represent the Stack) because the Stack is an ADT.



Exercise Break

What is the worst-case time complexity of adding ***n* elements** (not 1 element) into a Stack, given that we are using a Deque as our backing structure in our implementation and given that we are using a Doubly Linked List as the backing structure of our Deque?

Exercise Break

Say I have a stack of integers `s`, and I run the following code:

```

1 for (int i = 0; i < 10; i++) {
2     s.push(i);
3 }
4 s.pop();
5 s.push(100);
6 s.push(200);
7 s.pop();
8 s.pop();
9 s.pop();

```

The equivalent Python code would be the following:

```

1 for i in range(10):
2     s.push(i)
3 s.pop()
4 s.push(100)
5 s.push(200)
6 s.pop()
7 s.pop()
8 s.pop()

```

What is the number at the top of the stack after running this code? In other words, what would be returned if I call `s.top()`?

Despite its simplicity, the Stack is a very powerful ADT because of the numerous applications to which it can be applied, such as the following:

- Storing student exams that need to be graded
- Keeping track of the evaluation of smaller expressions within a large complex mathematical expression
- “Graph” exploration via the “Depth-First Search” algorithm (which will be covered in the “Graphs” section of this text)

This concludes our discussions about the introductory Data Structures and Abstract Data Types we will be covering in this text. We hope that the knowledge you acquired will help you implement them as well as to give you some intuition to help you understand more advanced data structures that we will encounter in the later sections of the text.

2.9 And the Iterators Gonna Iterate-ate-ate

As you may have noticed by now, although many data structures have similar functionality (e.g. insert, remove, find, etc.) and similar purpose (i.e., to store data in a structured fashion), they are implemented in fundamentally different ways. The differences in their implementations are essentially the causes of their

differing costs/benefits. However, as data structure wizards, we want to code our data structures in a fashion such that the user doesn't have to worry about the implementation details. For example, take the following segment of code:

```
1 for (auto element : data) {  
2     cout << element << endl;  
3 }
```

What data structure is `data`? Is it a `set`? `unordered_set`? `vector`? It could be any of these, or maybe something else altogether! Nonetheless, we are able to iterate through the elements of `data` in a clean fashion. In this section, we will be covering the C++ Standard Template Library (STL), with our attention focused on iterators, the mystical entities that make this functionality possible.

The **iterator pattern** of Object-Oriented design permits access to the data in some data structure in sequential order, without exposing the container's underlying representation. Basically, it allows us to iterate (hence the name "iterator") through the elements of a data structure in a uniform and simple manner.

You may be familiar with the **for-each loop** (shown above), where we iterate through the elements of a data structure in a clean fashion: the programming language actually uses an iterator to perform the for-each loop. Note that iterators exist in numerous languages, not just C++.

You may recall that Java has many standard data structures (e.g. lists, sets, maps, etc.) already implemented and ready to use in the Collections API in the `java.util` package. The Java Collections API defines an `Iterator` interface that can be used to traverse collections (each collection is responsible for defining its own iterator class), which only has three methods: `next()`, `hasNext()`, and `remove()`.

C++ has a package with similar functionality to Java's Collections API: the **C++ Standard Template Library (STL)**. With regard to iterators, which are what we are focusing on in this section, the specifics of C++ iterators are slightly different in comparison to Java iterators. The syntax is messy to describe verbally, so the best way to teach how to use iterators is to provide you with a step-by-step example:

```

1 vector<string> c;
2 /* populate the set with data */
3
4 // get an iterator pointing to c's first element
5 vector<string>::iterator itr = c.begin();
6 // get an iterator pointing past c's last element
7 vector<string>::iterator end = c.end();
8
9 // loop over the elements (overloaded !=)
10 while(itr != end) {
11     // dereference the iterator to get element
12     cout << *itr << endl;
13     // point to next element (overloaded ++ pre-increment)
14     ++itr;
15 }
```

- `*itr` returns a reference to the container object that `itr` is pointing to
- `itr1 == itr2` returns `true` if `itr1` and `itr2` refer to the same position in the same container (`false` otherwise)
- `itr1 != itr2` is equivalent to `!(itr1 == itr2)`
- `c.begin()` returns an iterator positioned at the first item in container `c`
- `c.end()` returns an iterator positioned *after* the last item in container `c` (typically `NULL` or some equivalent)

Exercise Break

Which of the following statements about C++ iterators are true?

- Iterators only exist in the C-family of languages
- Iterators are used behind-the-scenes of for-each loops
- Iterators can be used to cleanly, easily, and uniformly iterate through the elements of a data structure

As mentioned before, if our collection has an iterator class implemented (which all of the C++ STL data structures do), we can easily iterate over the elements of our collection using a for-each loop. The syntax is as follows:

```

1 void printElements(vector<string> & c) {
2     for(string s : c) {          // "for-each" string 's' in 'c':
3         cout << s << endl;      // print 's'
4     }
5 }
```

You may have noticed that the above code functions identically to the example code we showed you when we first introduced C++ iterator syntax. In addition to looking similar and resulting in identical output, the two methods

are actually functionally identical. As we mentioned, for-each loops use iterators behind the scenes. When we call the for-each loop, C++ actually creates an iterator object pointing to the first element, keeps incrementing this iterator (performing the contents of our loop's body on each element), and stops the moment it reaches the end. It simply hides these details from us to keep our code looking clean.

Exercise Break

What will be output by the following code?

```

1 vector<char> data;
2 data.push_back('H');
3 data.push_back('e');
4 data.push_back('l');
5 data.push_back('l');
6 data.push_back('o');
7 data.push_back('!');
8
9 for (auto element : data) {
10     cout << element;
11 }
```

Of course, being the hard-working and thorough computer scientists that we are, if our goal in this course is to implement data structures, we will want to implement the iterator pattern for our data structures so that users can easily iterate through the container's elements! If we want to implement the iterator pattern, we will need to implement the following functions/operators:

- *Functions in the Data Structure Class*
 - `begin()`: returns an `iterator` object “pointing to” the first element of the container
 - `end()`: returns an `iterator` object “pointing just past” the last element of the container
- *Operators in the Data Structure's Iterator Class*
 - `==` (equal): returns `true` (1) if the two `iterator` objects are pointing to the same element, otherwise `false` (0)
 - `!=` (not equal): returns `false` (0) if the two `iterator` objects are pointing to the same element, otherwise `true` (1)
 - `*` (dereference): returns a reference to the data item contained in the element the `iterator` object is pointing to
 - `++` (pre-increment): causes the `iterator` object to point to the next element of the container and returns the pointer to the object AFTER the increment

- **`++`** (post-increment): causes the `iterator` object to point to the next element of the container, but returns the pointer to the object BEFORE the increment

The following is an example of implementing an iterator for a Linked List. Note that we have omitted the actual Linked List functions (e.g. find, insert, remove, etc.) for the sake of keeping the example clean and focused on iterators.

```

1  using namespace std; // to omit std::
2  class Node { // Helper Node class
3      public:
4          int value; Node* next;
5  }; class LinkedList { // Linked List class
6      public:
7          Node* root; // root node
8          class iter : public iterator<forward_iterator_tag, int> {
9              public:
10                 friend class LinkedList;
11                 Node* curr; // element being pointed to
12                 // typedefs needed for C++ STL support:
13                 typedef int value_type;typedef int& reference;
14                 typedef int* pointer;typedef int difference_type;
15                 typedef forward_iterator_tag iterator_category;
16                 iter(Node* x=0):curr(x){} // constructor
17                 bool operator==(const iter& x) const { // ==
18                     return curr == x.curr;
19                 }
20                 bool operator!=(const iter& x) const { // !=
21                     return curr != x.curr;
22                 }
23                 reference operator*() const { // dereference
24                     return curr->value;
25                 }
26                 iter& operator++() { // pre-increment
27                     curr=curr->next; return *this;
28                 }
29                 iter operator++(int) { // post-increment
30                     iter tmp(curr); curr=curr->next; return tmp;
31                 }
32             };
33             iter begin() { return iterator(root); } // point to first
34             iter end() { return iterator(NULL); } // point AFTER last
35         };

```

Thus, if we were to want to iterate over the elements of this Linked List, we could easily do the following:

```

1  for(auto it = l.begin(); it != l.end(); it++) {
2      cout << *it << endl;
3  }

```

In this example, because we were creating an iterator for a Linked List, which is a structure built around pointers to nodes when implemented in C++,

the iterator objects had `Node` pointers (i.e., `Node*`) to keep track of the element to which they pointed. The following is an example of implementing an iterator for an Array List. Note that we have omitted the actual Array List functions (e.g. find, insert, remove, etc.) for the sake of keeping the example clean and focused on iterators.

```

1  using namespace std; // to omit std::
2  class ArrayList { // Array List class
3      public:
4          int arr[10]; int size;
5          class iter : public iterator<forward_iterator_tag,int> {
6              public:
7                  friend class ArrayList;
8                  int* curr; // element being pointing to
9                  // typedefs needed for C++ STL support
10                 typedef int value_type;typedef int& reference;
11                 typedef int* pointer;typedef int difference_type;
12                 typedef forward_iterator_tag iterator_category;
13                 iter(int* x=0):curr(x){} // constructor
14                 bool operator==(const iterator& x) const { // ==
15                     return curr == x.curr;
16                 }
17                 bool operator!=(const iterator& x) const { // !=
18                     return curr != x.curr;
19                 }
20                 reference operator*() const { // dereference
21                     return *curr;
22                 }
23                 iterator& operator++() { // pre-increment
24                     curr++; return *this;
25                 }
26                 iterator operator++(int) { // post-increment
27                     iter tmp(curr); curr++; return tmp;
28                 }
29             };
30             iter begin() { // point to first
31                 return iter(&arr[0]);
32             }
33             iter end() { // point AFTER last
34                 return iter(&arr[size]);
35             }
36         };

```

Thus, if we were to want to iterate over the elements of this Array List, we could easily do the following:

```

1  for(auto it = l.begin(); it != l.end(); it++) {
2      cout << *it << endl;
3  }

```

As can be seen, similar to a Linked List, we still use pointers to point to our elements when implementing an iterator for an Array List. Note that in the Array List iterator, iterators point directly to elements in the array (i.e., the

actual integers we are storing). However, in the Linked List iterator, iterators pointed to nodes and we had to use the nodes to access the elements we were storing. This is a direct consequence of a Linked List property: memory is allocated as nodes are created dynamically. Consequently, nodes end up all over the place in memory, and thus we need to use the nodes' forward pointers to iterate over them. With an Array List, however, because the slots of the array are all allocated at once and are contiguous in memory, we can simply move across cells of the array.

We started this chapter by learning *about* different data structures, without worrying too much about actual implementation. Then, we learned about some specifics regarding implementation of the data structures themselves. Now, we have learned about a key feature that we can implement into our data structures: iterators. Iterators are powerful entities that allow the user to iterate over the elements of *any* data structure in a clean and uniform syntax, without knowing how the data structure actually works on the inside.

Of course, as you noticed in the last portions of this section, iterators can be a *pain* to implement from scratch! Lucky for us, in practically all programming languages, some pretty smart people have already gone through and implemented all of the fundamental data structures *with* iterators! In practice, you will always want to use some existing implementation of a data structure (and its iterator implementation). In C++ specifically, you want to use data structures already implemented in the STL, which are implemented extremely efficiently.

Nevertheless, it is important to know *how* to go about implementing things from scratch, as you never know what you will see in life! Perhaps you will enter an extremely niche field of research, and data structures you need have not yet been implemented in the language of your preference, or perhaps you are secretly a genius who will one day design a groundbreaking super efficient data structure all on your own, for which you will of course give credit to the authors of this amazing online textbook that sparked your interest in data structures in the first place. Either way, you may one day find yourself needing to know the good practices of data structure implementation, and the iterator pattern is a key feature you will want to implement.

Chapter 3

Tree Structures

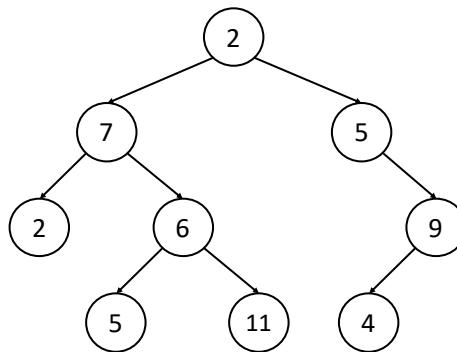
3.1 Lost in a Forest of Trees

In the previous chapter, one of the introductory data structures we covered was the Linked List, which, as you recall, was a set of **nodes** that were strung together via links (**edges**). We had direct access to the “head” node (and the “tail” node as well, if we so chose), but in order to access nodes in the middle of the list, we had to start at the “head” (or “tail”) node and traverse edges until we reached the node of interest.

The general structure of “a set of nodes and edges” is called a **Graph**, with a Linked List being an extremely simple example of a graph. Specifically, we can think of a Singly-Linked List as a graph in which every node (except the “tail” node) has a single “forward” edge, and we can think of a Doubly-Linked List as a graph in which every node (except the “head” and “tail” nodes) has exactly two edges: one “forward” edge and one “back” edge.

The Linked List was a very simple graph structure, so in this chapter, we’ll introduce a slightly more complex graph structure: the **Tree** structure.

Formally, a **graph** is, by definition, a collection of **nodes** (or **vertices**) and **edges** connecting these nodes. A **tree** is defined as a graph without any **undirected cycles** (i.e., cycles that would come about if we replaced directed edges with undirected edges) nor any **unconnected parts**. The following is an example of a valid tree:

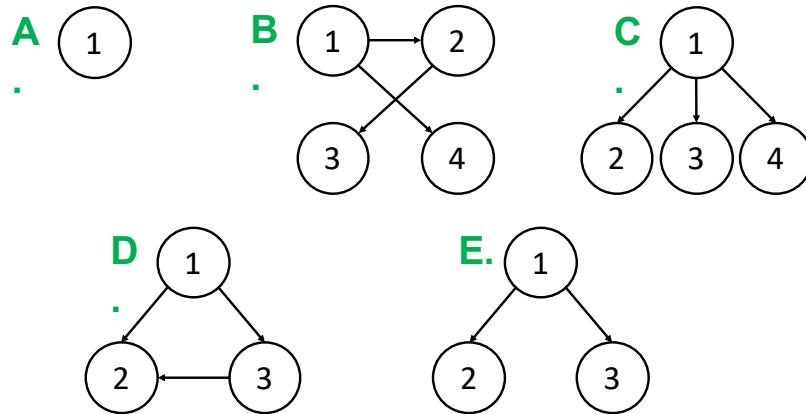


Note that, even though we drew this tree with directed edges, it is perfectly valid for a tree to have undirected edges. Even though the tree above looks like what we would expect of a tree, there are some special “weird” cases that one might not expect are valid trees:

- a tree with no nodes is a valid tree called the “null” (or “empty”) tree
- a tree with a single node (and no edges) is also a valid tree

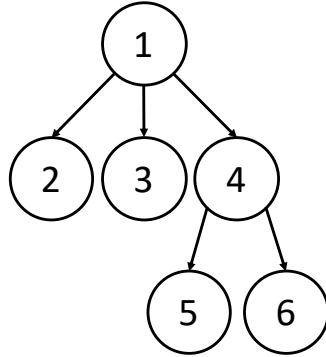
Exercise Break

Which of the following are valid trees?

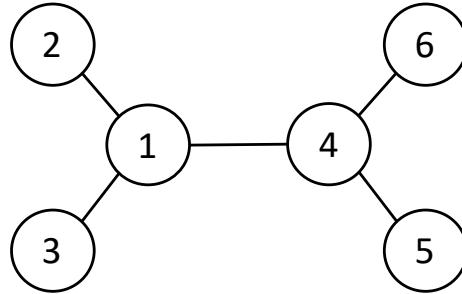


There are two classes of trees: *rooted* and *unrooted*. In a **rooted** tree, a given node can have a single **parent** node above it and can have any number of **children** nodes below it. Just like with a family tree, all nodes that appear along the path going upward from a given node are considered that node’s **ancestors**, and all nodes that appear along any path going downward from a given node are considered that node’s **descendants**. There is a single node at the top of the tree that does not have a **parent**, which we call the **root**, and there

can be any number of nodes at the bottom of the tree that have no *children*, which we call **leaves**. All nodes that have *children* are called **internal nodes**. The following is an example of a *rooted* tree, where the *root* is 1, the *leaves* are $\{2, 3, 5, 6\}$, and the *internal nodes* are $\{1, 4\}$:



In an **unrooted** tree, there is no notion of *parents* or *children*. Instead, a given node has **neighbors**. Any nodes with just a single *neighbor* is considered a **leaf** and any node with more than one neighbor is considered an **internal node**. The following is an example of an *unrooted* tree, where the *leaves* are $\{2, 3, 5, 6\}$, and the *internal nodes* are $\{1, 4\}$:

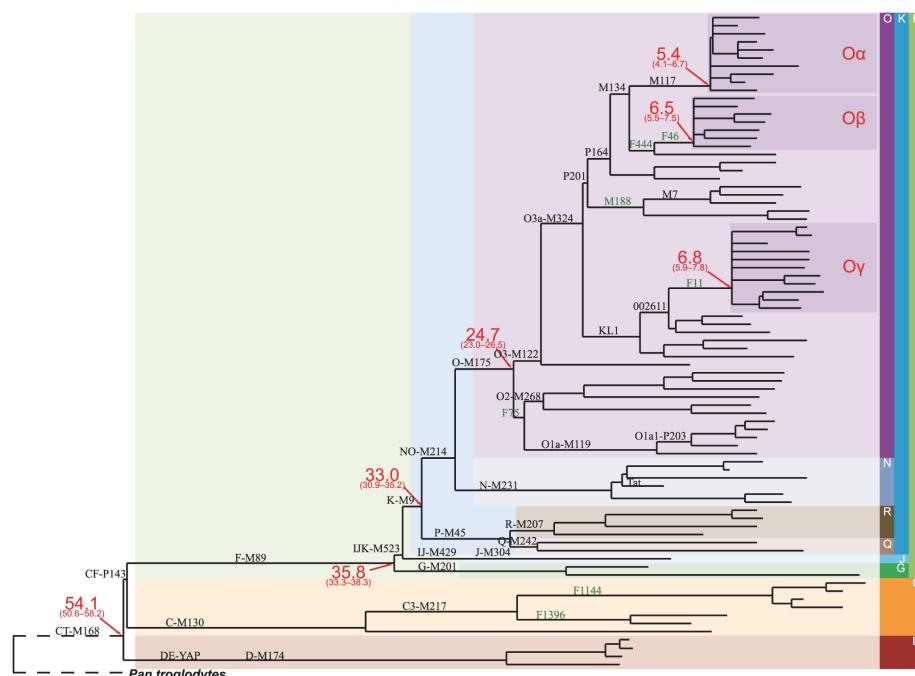


STOP and Think

If I were to make the directed edges of the first tree into undirected edges, would it become equivalent to the second tree?

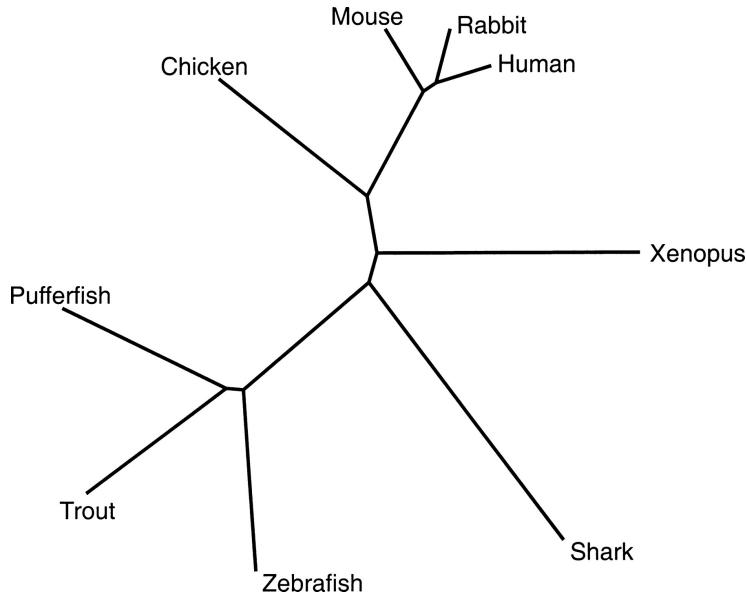
Rooted trees have an implied hierarchical structure: an arbitrary node may have *children* and *descendants* below it, and it may have a *parent* and *ancestors* above it. In some contexts, we might want to interpret this hierarchy as “top-down” (i.e., we care about relationships from root to leaves) or as “bottom-up” (i.e., we care about relationships from leaves to root), but either way, the underlying tree structure is the same, despite how we choose to interpret the hierarchy it can represent.

For example, in Bioinformatics, we can study a paternal lineage of humans by creating an evolutionary tree from the Y chromosomes of a large set of individuals (because the Y chromosome is a single DNA sequence that is passed down from father to son). In this case, the tree represents time in the “top-down” direction: traversing the tree from *root* to *leaves* takes you forward in time. As a result, the *root* node is the most recent common ancestor of the individuals we chose to sequence. The following is an example of such a tree:



Unrooted trees do not necessarily have an implied hierarchical structure. In some contexts, we might want to interpret an “inside-out” hierarchy (i.e., we care about relationships from internal nodes to leaves) or an “outside-in” hierarchy (i.e., we care about relationships from leaves to internal nodes). In unrooted trees, instead of thinking of nodes as “parents” or “children,” it is common to simply think of nodes as “neighbors.” Also, even if we don’t impose any sense of directionality onto the tree, distances between nodes can provide some information of “closeness” or “relatedness” between them.

For example, in Bioinformatics, if we want to construct an evolutionary tree from different organisms, if we don’t know what the true ancestor was, we have no way of picking a node to be the root. However, the unrooted tree still provides us relational information: leaves closer together on the tree are more similar biologically, and leaves farther apart on the tree are more different biologically. The following is an example of such a tree:



For a graph T to be considered a tree, it must follow all of the following constraints (which can be proven to be equivalent constraints):

- T is connected and has no undirected cycles (i.e., if we made T 's directed edges undirected, there would be no cycles)
- T is acyclic, and a simple cycle is formed if any edge is added to T
- T is connected, but is not connected if any single edge is removed from T
- There exists a unique simple path connecting any two vertices in T

If T has n vertices (where n is a finite number), then the above statements are equivalent to the following two conditions:

- T is connected and has $n-1$ edges
- T has no simple cycles and has $n-1$ edges

However, we run into a bit of a problem with the 0-node graph, which we previously called the “empty” tree. On one hand, as a graph, it violates some of the constraints listed above and thus should not be considered a tree. On the other hand, we explicitly called it a valid tree data structure previously because it represents a tree data structure containing 0 elements, which is valid. What is the correct verdict? For the purposes of this text, since we are focusing on data structures and not on graph algorithms specifically, let's explicitly handle this edge case and just say that a 0-node graph is a valid tree, despite the fact that it fails some of the above conditions.

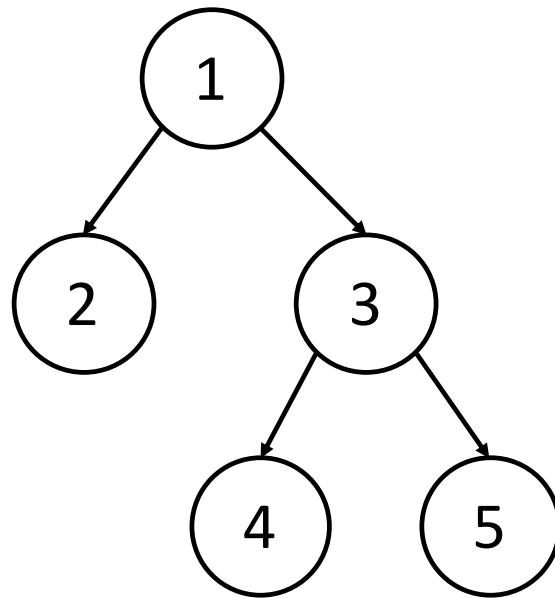
STOP and Think

Which of the above conditions are violated by the empty tree?

In this text, we will pay special attention to **rooted binary trees**, which are rooted trees with the following two restrictions:

- All nodes except the root have a parent
- All nodes have either 0, 1, or 2 children

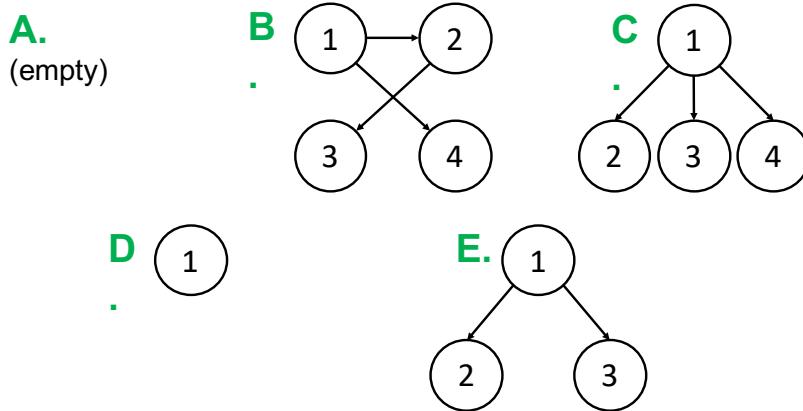
Many of the example trees used previously are binary trees, but an example binary tree has been reproduced:



Note that *any* graph that follows the above restrictions is considered a valid rooted binary tree. Again, for our purposes, we will consider the empty tree to be a valid rooted binary tree. Also, we specify “rooted” binary tree here because there do exist “unrooted” binary trees, but they are out of the scope of this text, so if we use the term “binary tree” at any point later in this text without specifying “rooted” or “unrooted,” we mean “rooted.”

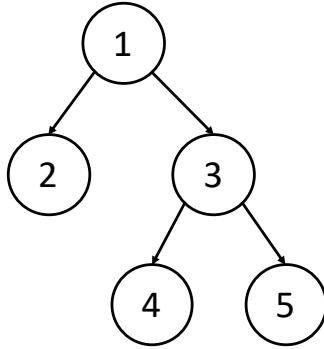
Exercise Break

Which of the following are valid rooted binary trees?



With rooted binary trees (and rooted trees in general), we typically only maintain a pointer to the root because all other nodes in the tree can be accessed via some traversal starting at the root. It can be useful to keep pointers to the leaves of the tree as well, but these pointers can be harder to maintain because the leaves of a rooted tree change rapidly as we insert elements into the tree.

Because we typically only keep track of the root node, to traverse all of the nodes in a rooted binary tree, there are four traversal algorithms: **pre-order**, **in-order**, **post-order**, and **level-order**. In all four, the verb “visit” simply denotes whatever action we perform when we are at a given node u (whether it be printing u ’s label, or modifying u in some way, or incrementing some global count, etc.).



In a **pre-order** traversal, we first visit the current node, then we recurse on the left child (if one exists), and then we recurse on the right child (if one exists). Put simply, **VLR** (**Visit-Left-Right**). In the example above, a pre-order traversal starting at the root would visit the nodes in the following order:
1 2 3 4 5

In an **in-order** traversal, we first recurse on the left child (if one exists), then we visit the current node, and then we recurse on the right child (if one

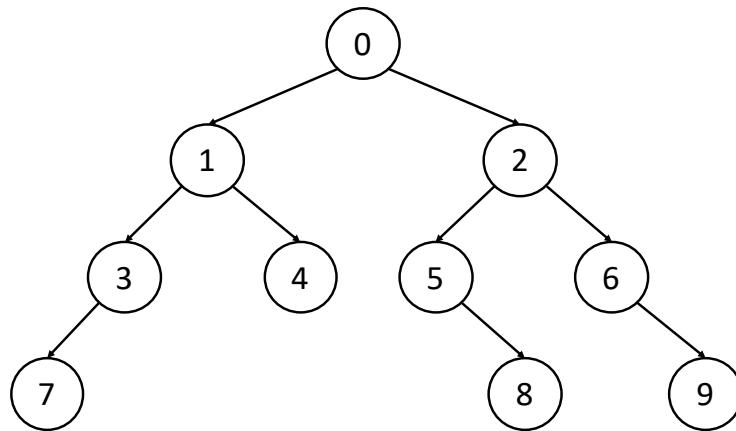
exists). Put simply, **LVR** (**L**evel-**V**isit-**R**ight). In the example above, an in-order traversal starting at the root would visit the nodes in the following order: 2 1 4 3 5

In a **post-order** traversal, we first recurse on the left child (if one exists), then we recurse on the right child (if one exists), and then we visit the current node. Put simply, **LRV** (**L**evel-**R**ight-**V**isit). In the example above, a post-order traversal starting at the root would visit the nodes in the following order: 2 4 5 3 1

In a **level-order** traversal, we visit nodes level-by-level (where the root is the “first level,” its children are the “second level,” etc.), and on a given level, we visit nodes left-to-right. In the example above, a level-order traversal starting at the root would visit the nodes in the following order: 1 2 3 4 5

Exercise Break

In what order will the nodes of the following tree be visited in a pre-order traversal? In an in-order traversal? In a post-order traversal? In a level-order traversal?



Exercise Break

What is the worst-case time complexity of performing a pre-order, in-order, post-order, or level-order traversal on a rooted binary tree with n nodes?

Now that you have learned about trees in general, with a focus on rooted binary trees, you are ready to learn about the various other specific tree structures covered in this text. The bulk of these trees will be rooted binary trees, so don’t forget the properties and algorithms you learned in this section!

3.2 Heaps

In the previous chapter, we discussed the Queue ADT, which can be thought of like a grocery store line: the first person to enter the line (queue) is the first person to come out of the line (queue), or so we would hope... However, what if, instead of a grocery store, we were talking about an emergency room? Clearly there is some logic regarding the order in which we see patients, but does an emergency room line follow a queue?

Say we have an emergency room line containing 10 individuals, all with pretty minor injuries (e.g. cuts/bruises). Clearly, the order in which these 10 patients are treated doesn't really matter, so perhaps the doctor can see them in the order in which they arrived. Suddenly, someone rushes through the door with a bullet wound that needs to be treated immediately. Should this individual wait until all 10 minor-injury patients are treated before he can be seen?

Clearly, we need an ADT that is similar to a Queue, but which can order elements based on some *priority* as opposed to blindly following the “First In First Out” (FIFO) ordering. The ADT that solves this exact problem is called the **Priority Queue**, which is considered a “**Highest Priority In, First Out**” (HPIFO) data type. In other words, the *highest priority* element currently in the Priority Queue is the first one to be removed.

Recall from when we introduced the previous ADTs that an ADT is defined by a set of functions. The Priority Queue ADT can be formally defined by the following set of functions:

- `insert(element)`: Add `element` to the Priority Queue
- `peek()`: Look at the highest priority element in the Priority Queue
- `pop()`: Remove the highest priority element from the Priority Queue

Although we are free to implement a Priority Queue any way we want (because it is an ADT), you'll soon realize that the data structures we've learned about so far make it a bit difficult to guarantee a good worst-case time complexity for *both* insertion and removal. We could theoretically use a *sorted Linked List* to back our Priority Queue, which would result in $\mathcal{O}(1)$ peeking and removing (we would have direct access to the highest priority element), but insertion would be $\mathcal{O}(n)$ to guarantee that our list remains sorted (we would have to scan through the sorted list to find the correct insertion site). Likewise, we could theoretically use an *unsorted Linked List* to back our Priority Queue, which would result in $\mathcal{O}(1)$ insertion (we could just insert at the head or tail of the list), but peeking and removing would be $\mathcal{O}(n)$ (we would have to scan through the unsorted list to find the highest priority element). If you were to implement this using a sorted or unsorted Array, the worst-case time complexities would be the same as for a sorted or unsorted Linked List.

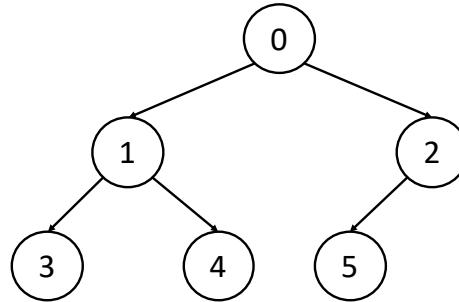
It turns out that there is a specific data structure that developers typically use to implement the Priority Queue ADT that guarantees $\mathcal{O}(1)$ peeking and

$\mathcal{O}(\log n)$ insertion and removal in the worst case. In this section of the text, we will explore the first of our tree structures: the **Heap**.

A heap is a tree that satisfies the **Heap Property**: For all nodes A and B , if node A is the parent of node B , then node A has *higher priority* (or equal priority) than node B . In this text, we will only discuss **binary heaps** (i.e., heaps that are binary trees), but this binary tree constraint is not required of heaps in general. The binary heap, has three constraints (two of which should hopefully be obvious):

- **Binary Tree Property:** All nodes in the tree must have either 0, 1, or 2 children
- **Heap Property** (explained above)
- **Shape Property:** A heap is a *complete* tree. In other words, all levels of the tree, except possibly the bottom one, are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right

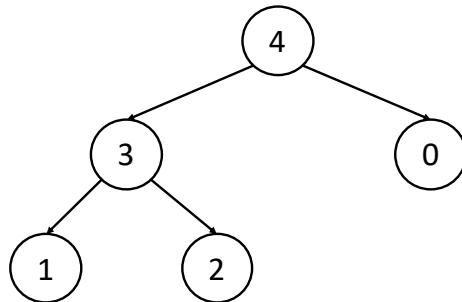
The following is an example of a heap (higher priority is given to smaller elements):



We mentioned the term “priority,” but what does it formally mean in the context of elements in a tree? The concept of *priority* can be more easily understood by looking at the two types of heaps: the **min-heap** and the **max-heap**.

A **min-heap** is a heap where every node is *smaller* than (or equal to) all of its children (or has no children). In other words, a node A is said to have *higher priority* than a node B if $A < B$ (i.e., when comparing two nodes, the *smaller* of the two has higher priority). The example above is an example of a min-heap.

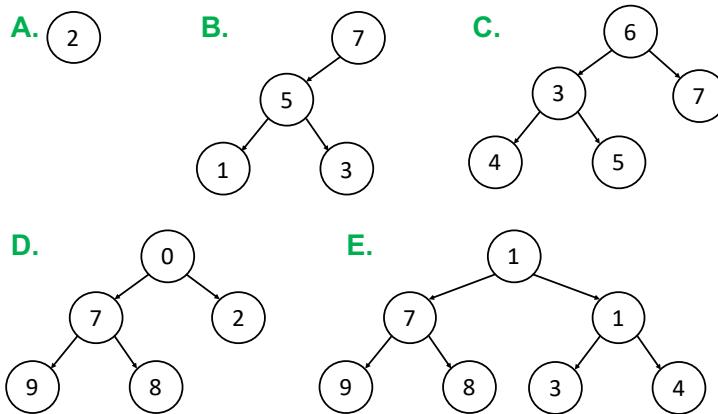
A **max-heap** is a heap where every node is *larger* than (or equal to) all of its children (or has no children). In other words, a node A is said to have higher priority than a node B if $A > B$ (i.e., when comparing two nodes, the *larger* of the two has higher priority). The following is an example of a max-heap:



Note: We hinted at this above by sneaking in “or equal to” when describing min-heaps and max-heaps, but to be explicit, note that elements of a heap are interpreted as *priorities*, and as a result, it makes sense for us to **allow duplicate priorities** in our heap. Going back to the initial hospital example that motivated this discussion in the first place, what if we had *two* victims with bullet wounds, both of whom entered the hospital at the same time? The choice of which patient to see first is arbitrary: both patients have equally high priorities. As a result, in a heap, if you encounter duplicate priorities, simply break ties arbitrarily.

Exercise Break

Which of the following are valid binary heaps?



Given the structure of a heap (specifically, given the *Heap Property*), we are guaranteed that any given element has a higher (or equal) priority than all of its children. As an extension, it should hopefully be clear that any given element must also have a higher priority than all of its *descendants*, not just its direct children. Thus, it should be intuitive that the root, which is the ancestor of all other elements in the tree (i.e., all other elements of the tree are descendants of the root) must be the highest-priority element in the heap.

Thus, the "algorithm" to peek at the highest-priority element in a heap is extremely trivial:

```

1 peek():
2     return the root element

```

Since we can assume that our implementation of a heap will have direct access to the root element, the peek operation is $\mathcal{O}(1)$ in the worst case.

The first non-trivial heap algorithm we will discuss is the **insertion** operation, "push." First, recall that one of the constraints of a heap is that it must be a full tree. Given this constraint, the first step of the heap insertion algorithm is quite intuitive: simply insert the new element in the next open slot of the tree (the new node will be a leaf in the bottom level of the tree, by definition of "next open slot"). In doing so, we have maintained the *Shape Property*, so the only constraint that might be violated is the *Heap Property*: the new element might potentially have higher priority than its parent, which would make the heap invalid.

Thus, the next (and last) step of the insertion algorithm should be intuitive as well: we must fix the (potentially) disrupted *Heap Property* of our tree. To do so, we must **bubble up** the newly-inserted element: if the new element has a higher priority than its parent, swap it with its parent; now, if the new element has higher priority than the new parent, swap; repeat until it has reached its correct place (either it has lower priority than its parent, or it is the root of the tree).

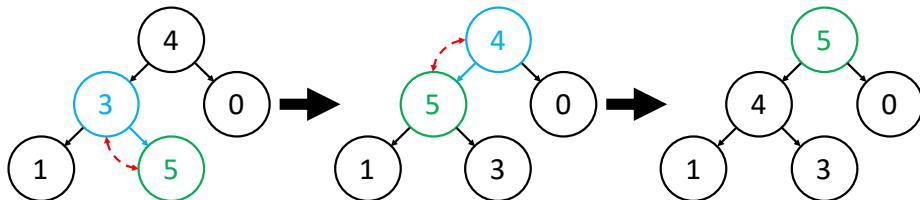
Formally, the pseudocode of heap insertion is as follows:

```

1 insert(e):
2     place e in next open slot (as defined by a "full tree")
3     while e has a parent and e.priority > e.parent.priority:
4         swap e and e.parent

```

The following is a visualization of the heap insertion algorithm, where we insert 5 into the existing max-heap:



Because of the *Shape Property* of a heap, we are guaranteed to have a perfectly balanced binary tree, which means that we are guaranteed to have $\mathcal{O}(\log n)$ levels in our tree. If we have L levels in our tree, in the worst case, we do $L - 1$ comparisons for the bubble up process. As a result, if $L = \mathcal{O}(\log n)$ and we do $\mathcal{O}(L)$ comparisons for the insertion algorithm, the overall insertion algorithm is $\mathcal{O}(\log n)$. Note that, in the above algorithm, we assumed that we

have direct (i.e., constant-time) access to the “next open slot” of the tree. We will soon see that this is easily achievable when implementing the heap.

The second heap algorithm we will discuss is the **removal** operation, “pop.” Just like the insertion algorithm, it is actually quite simple when we use the constraints of a heap to derive it.

First, recall that the root element is by definition the highest-priority element in the heap. Thus, since we always remove the highest-priority element of a heap, the first step is to simply remove the root. However, since we need a valid tree structure, we need some other node to become the new root. In order to maintain the *Shape Property* of the heap, we can simply place the “last” element of the heap (i.e., the rightmost element of the bottom row of the heap) as the new root.

By making the “last” element of the heap the new root, we most likely have violated the *Heap Property*: the new root might have lower priority than its children, which would make the heap invalid. Thus, the next (and last) step of the “pop” algorithm should be intuitive as well: we must fix the (potentially) disrupted *Heap Property* of our tree. To do so, we must **trickle down** the newly-placed root: if the root has lower priority than its children, swap it with its highest-priority child; now, if it still has lower priority than its children in this new position, swap with its highest-priority child again; repeat until it has reached its correct place (either it has higher priority than all of its children, or it is a leaf of the tree).

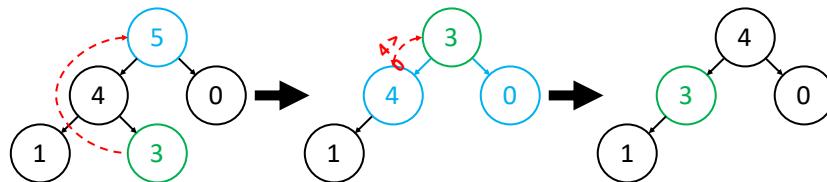
Formally, the pseudocode of heap removal is as follows:

```

1  pop():
2      replace root with rightmost element of bottom level ("curr")
3      while curr not leaf and curr.priority < any of its children:
4          swap curr and its highest-priority child

```

The following is a visualization of the heap “pop” algorithm, where we pop from the existing max-heap:



Because of the *Shape Property* of a heap, we are guaranteed to have a perfectly balanced binary tree, which means that we are guaranteed to have $\mathcal{O}(\log n)$ levels in our tree. If we have L levels in our tree, in the worst case, we do $L - 1$ comparisons for the trickle down process. As a result, if $L = \mathcal{O}(\log n)$ and we do $\mathcal{O}(L)$ comparisons for the insertion algorithm, the overall pop algorithm is $\mathcal{O}(\log n)$.

STOP and Think

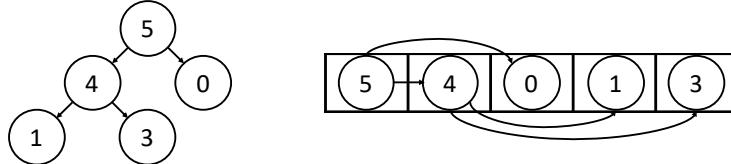
When trickling down, why do we have to swap with the highest-priority child specifically?

Exercise Break

Which property (or properties) of a binary heap give rise to the $\mathcal{O}(\log n)$ worst-case time complexity of the insertion algorithm? The pop algorithm?

Although we like visualizing a binary heap as a tree structure with nodes and edges, when it comes to implementation, by being a little clever, it turns out that we can store a binary heap as an array. Technically, any binary tree can be stored as an array, and by storing a binary tree as an array, we can avoid the overhead of wasted memory from parent and child pointers. Thus, given an index in the array, we can find the element's parent and children using basic arithmetic. An added bonus with binary heaps over unbalanced binary trees is that, since a binary heap is by definition a complete tree, there is no wasted space in the array: all elements will be stored perfectly contiguously.

The following is a diagram showing how we can represent an example binary max-heap as an array, where the tree representation is shown to the left and the array representation (with the edges maintained for the sake of clarity) is shown to the right:



Notice that, in the array representation, we can access the parent, left child, or right child of any given element in constant time using simple arithmetic. For an element at index i of the array, using 0-based indexing,

- its **parent** is at index $\lfloor \frac{i-1}{2} \rfloor$
- its **left child** is at index $2i + 1$
- its **right child** is at index $2i + 2$

Also, recall that we previously assumed constant-time access to the “next open slot” of the heap. When a heap with n elements is implemented as an array, the heap’s “next open slot” is at index n (using 0-based indexing).

Exercise Break

Say we have a heap represented as an array, using 0-based indexing, and we are looking at the element at index 101. At what index would we find its parent?

Exercise Break

Which of the following statements are true?

- The element at index i of the array representation of a binary heap must have a higher priority than all elements at indices larger than i
- By representing a heap as an array, we save on the memory costs from pointers in the tree representation
- If I were to iterate through the elements of the array representation of a binary heap, I would be traversing the corresponding tree representation in an *in-order* traversal
- Given an index in the array representation of a binary heap, I can find the parent, left child, and right child of the element at that index in constant time using arithmetic

We have now very efficiently solved our initial problem of implementing a queue-like data type that can take into account a sense of ordering (based on “priority”) of the elements it contains. We first described the Priority Queue ADT, which defined the functions we needed in such a data type.

Then, we dove into implementation specifics by discussing the heap data structure, which is almost always used to implement a Priority Queue. Because of the constraints of the heap data structure, we are able to guarantee a worst-case time complexity of $\mathcal{O}(\log n)$ for both inserting and popping elements, as well as a $\mathcal{O}(1)$ worst-case time complexity for peeking at the highest-priority element.

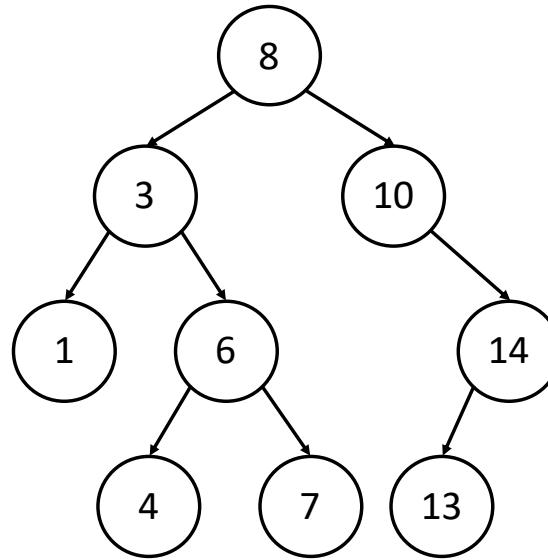
In the next sections of this chapter, we will continue to explore other types of binary trees, and it will hopefully become clear why we would want to have so many different data structures and ADTs in our arsenal of tools.

3.3 Binary Search Trees

In the previous section, we were motivated by the goal of storing a set of elements in such a manner that we could quickly access the *highest priority* element in the set, which led us to the heap data structure (and its use to implement the Priority Queue ADT). However, although we obtained fast access to the highest priority element, we had no way of efficiently looking at any other element. What if, instead, we want a data structure that could store a set of elements such that we could find any arbitrary element fairly quickly?

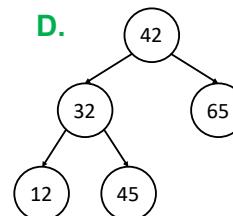
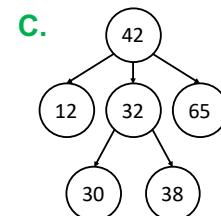
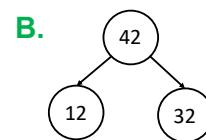
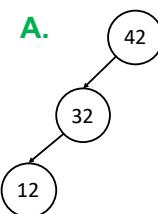
The second type of binary tree we will discuss is the **Binary Search Tree (BST)**, which is a rooted binary tree (i.e., there is a single “root” node, and all nodes have either 0, 1, or 2 children) in which any given node is larger than all nodes in its left subtree and smaller than all nodes in its right subtree. As can be inferred, a BST can only store elements if there exists some way of comparing them (e.g. strings, characters, numbers, etc.): there must be some inherent

order between elements. We will be learning about the following functions of a BST: *find*, *size*, *clear*, *insert*, *empty*, *successor*, and the *iterator* pattern. The following is an example of a typical valid Binary Search Tree:



Exercise Break

Which of the following are valid Binary Search Trees?



Exercise Break

What is the space complexity of a Binary Search Tree with n elements?

The following is pseudocode for the basic BST functions. They should be somewhat intuitive given the structure of a BST. In these examples, we assume that a BST object has two global variables: the *root node* (`root`), and an integer keeping track of the *size* of the BST (`size`). Note that pseudocode only represents the logic behind the algorithms (not actual implementation details), so you do not need to worry about language-specific things (e.g. syntax, memory management) until actual implementation.

```

1 find(e): // returns True if e exists in BST, otherwise False
2     current = root           // start at the root
3     while curr != e:
4         if e < curr:        // traverse left
5             c = c.leftChild
6         else if e > curr:   // traverse right
7             c = c.rightChild
8         if current == NULL: // no such child, failure
9             return False
10        return True          // curr == e, so we found e


---


1 insert(e): // inserts e into BST and returns True on success
2     if BST is empty:
3         root = element; return True
4     curr = root           // start at root
5     while curr != e:
6         if e < curr:
7             if curr.leftChild == NULL: // no left child exists
8                 curr.leftChild = e; return True
9             else:                  // left child exists,
10                curr = curr.leftChild // so traverse left
11         else if e > curr:
12             if curr.rightChild == NULL: // no right child exists
13                 curr.rightChild = e; return True
14             else:                  // right child exists,
15                 curr = curr.rightChild // so traverse right
16     return False // curr == e, can't have duplicate elements


---


1 size(): // returns the number of elements in BST
2     return size


---


1 clear(): // clears BST
2     root = NULL; size = 0

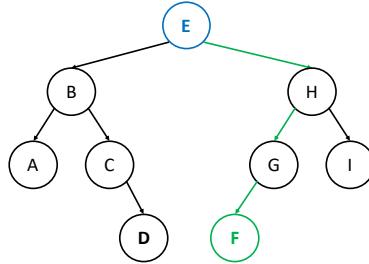

---


1 empty(): // returns True if BST is empty, otherwise False
2     if size == 0:
3         return True
4     else:
5         return False

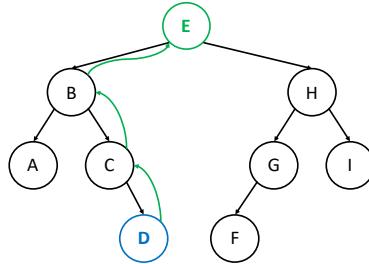
```

The **successor** function of a Binary Search Tree (or of a node, really) finds the “successor”—the next largest node—of a given node in the BST. We can break this function down into two cases:

Case 1 If node u has a right child, we can somewhat trivially find u 's successor. Recall that, by definition, a node is smaller than all of the nodes in its right subtree. Therefore, if u has a right child, u 's successor must be in its right subtree. Note that u 's successor is not necessarily its right child! Specifically, if u has a right child, u 's successor must be the smallest node in its right subtree. Thus, to find u 's successor, we can traverse to u 's right child and then traverse as far left as possible (What does this repeated left traversal do?). In the following example, we will find E's successor:



Case 2 If node u does not have a right child, we have to be a bit more clever. Starting at u , we can traverse up the tree. The moment we encounter a node that is the left child of its parent, the parent must be u 's successor. If no such node exists, then u has no successor. In the following example (slightly different than the above example), we will find D's successor:



```

1  successor(u): // returns u's successor, otherwise NULL
2      if u.rightChild != NULL: // Case 1: u has a right child
3          curr = u.rightChild
4          while curr.leftChild != NULL: // find smallest node
5              curr = curr.leftChild // in right subtree
6          return curr
7      else: // Case 2: u doesn't have a right child
8          curr = u
9          while curr.parent != NULL: // traverse up until curr
10             if curr == curr.parent.leftChild:
11                 return curr.parent // is parent's left child
12             else:
13                 curr = curr.parent
14         return NULL // no successor exists

```

Now that we have determined how to find a given node's successor efficiently, we can perform two important BST functions.

The first function we will describe is the **remove** function. To remove a node u from a BST, we must consider three cases. First, if u has no children, we can simply delete u (i.e., have u 's parent no longer point to u). Second, if u has a single child, have u 's parent point to u 's child instead of to u (Why does this work?). Third, if u has two children, replace u with u 's successor and remove u 's successor from the tree (Why does this work?).

```

1  remove(e): // removes e if exists in BST, returns True on success
2      curr = root                      // start at the root
3      while curr != e:
4          if e < curr:                  // traverse left
5              curr = curr.leftChild
6          else if e > curr:           // traverse right
7              curr = curr.rightChild
8          if curr == NULL:           // no such child , so failure
9              return False
10         // we only reach here if curr == e: found e
11         if curr has no children: // Case 1
12             remove the edge from curr.parent to curr
13         else if curr has 1 child: // Case 2
14             have curr.parent point to curr s child instead of curr
15         else:                   // Case 3
16             s = curr s successor
17             if s is its parent's left child:
18                 s.parent.leftChild = s.rightChild
19             else:
20                 s.parent.rightChild = s.rightChild
21             replace curr's value with s's value

```

The second function we will describe is the **in-order traversal** function. An in-order traversal of a BST starts at the root and, for any given node, traverses left, then “visits” the current node, and then traverses right. However, what if we want to iterate through the elements of a BST in sorted order with the freedom of starting at any node that we want? Doing a full-fledged in-order traversal starting at the root would be inefficient. Instead, we can simply start at whichever node we want and repeatedly call the successor function until we reach a point where no more successors exist.

```

1  inOrder(): // in-order traversal over elements of BST
2      curr = the left-most element of BST
3      while curr != NULL:
4          output curr
5          curr = successor(curr)

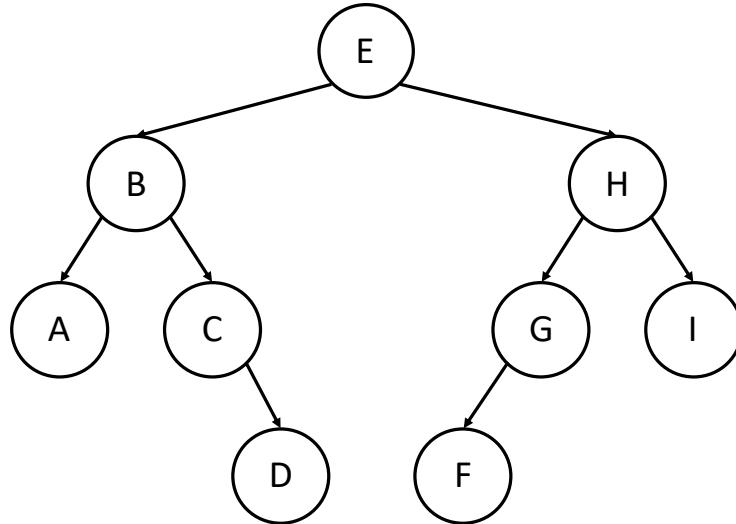
```

To be able to analyze the performance of a Binary Search Tree, we must first define some terms to describe the tree's shape. A tree's **height** is the longest distance from the root of the tree to a leaf, where we define “distance” in this case as the number of edges (not nodes) in a path. A tree with just a root has

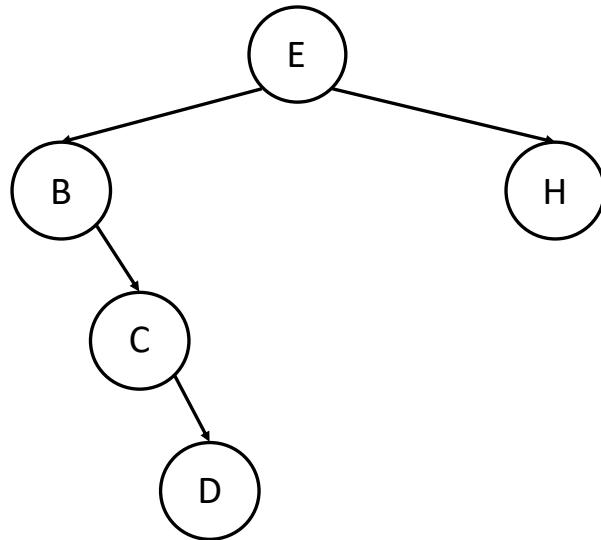
a height of 0, a tree with a root and one child has a height of 1, etc. For the sake of consistency, we say that an empty tree has a height of -1.

Notice that the worst-case time complexity of inserting, finding, and removing elements of a BST is proportional to its *height*: in the tree-traversal step of any of the three aforementioned operations (i.e., traversing left or right to either find an element or to find the insertion site for a new element), we perform a constant-time comparison operation to see if we have to continue left or right, or to see if we are finished. Recall that we perform this constant-time comparison operation once for each level of the BST, so as the number of levels in a BST grows (i.e., the *height* of the BST grows), the number of comparison operations we must perform grows proportionally.

A **balanced** binary tree is one where most leaves are equidistant from the root and most internal nodes have two children. A *perfectly balanced* binary tree is one where *all* leaves are equidistant from the root, and *all* internal nodes have two children. The term “balanced” can be generalized to any k -ary tree (e.g. binary, ternary, 4-ary, etc.): a balanced k -ary tree is one where many (or most) internal nodes have exactly k children and all leaves are roughly equidistant from the root. The following is an example of a balanced BST:



An **unbalanced** binary tree is one where many internal nodes have exactly one child and/or leaves are not equidistant from the root. A *perfectly unbalanced* binary tree is one where *all* internal nodes have exactly one child. The term “unbalanced” can be generalized to any k -ary tree: an unbalanced k -ary tree is one where many internal nodes have less than k children (the more extreme deviation from k children, the more unbalanced the tree). The following is an example of an unbalanced BST:



It should be clear that, with regard to balance, the two extreme shapes a tree can have are *perfectly balanced* and *perfectly unbalanced*. If a tree is *perfectly unbalanced*, each internal node has exactly one child, so the tree would have a height of $n - 1$ (we would have n nodes down one path, and that path will have $n - 1$ edges: one edge between each pair of nodes along the path).

If a binary tree is *perfectly balanced*, each internal node has exactly two children (or k children, if we generalize to k -ary trees) and all leaves are exactly equidistant from the root, so a *perfectly balanced* binary tree with n elements has a height of $\log_2(n + 1) - 1$. This equation looks complicated, but the $\log_2(n)$ portion, which is the important part, has some simple intuition behind it: every time you add another level to the bottom of a perfectly balanced binary tree, you are roughly doubling the number of elements in the tree. If a tree has one node, adding another row adds 2 nodes. If a tree has 3 nodes, adding another row adds 4 nodes. If a tree has 7 nodes, adding another row adds 8 nodes. Since the height of a perfectly balanced binary tree effectively tells you “how many doublings” occurred with respect to a one-node perfectly balanced binary tree, the number of nodes in the tree would be roughly $n = 2^h$ (where h is the height of the tree), so the height of the tree would be roughly $h = \log_2(n)$.

STOP and Think

What previously-discussed data structure does a binary tree transform into when it becomes perfectly unbalanced?

Exercise Break

What is the worst-case time complexity of the find operation of a Binary Search Tree with n elements?

Exercise Break

What is the worst-case time complexity of the find operation of a *balanced* Binary Search Tree with n elements?

As you might have noticed, although Binary Search Trees are relatively efficient when they are balanced, they can become unbalanced fairly easily depending on the order in which the elements are inserted.

STOP and Think

If we were to insert elements into a BST in sorted order (i.e., insert the smallest element, then insert the next smallest, etc.), what would the resulting tree look like?

We found out that a Binary Search Tree performs pretty badly in the worst case, but that it performs pretty well when it is well-balanced. How does the BST perform on average (i.e., what is its average-case time complexity)? Although this is a seemingly simple question, it requires formal mathematical exploration to answer, which we will dive into in the next section.

3.4 BST Average-Case Time Complexity

As we mentioned previously, the worst-case time complexity of a Binary Search Tree is quite poor (linear time), but when the Binary Search Tree is well-balanced, the time complexity becomes quite good (logarithmic time). On average, what do we expect to see?

Notice that we can define a Binary Search Tree by the insertion order of its elements. If we were to look at all possible BSTs with n elements, or equivalently, look at all possible insertion orders, it turns out that the average-case time complexity of a successful “find” operation in a binary search tree with n elements is actually $\mathcal{O}(\log n)$. So how can we prove it? To do so (with a formal mathematical proof), we will make two assumptions:

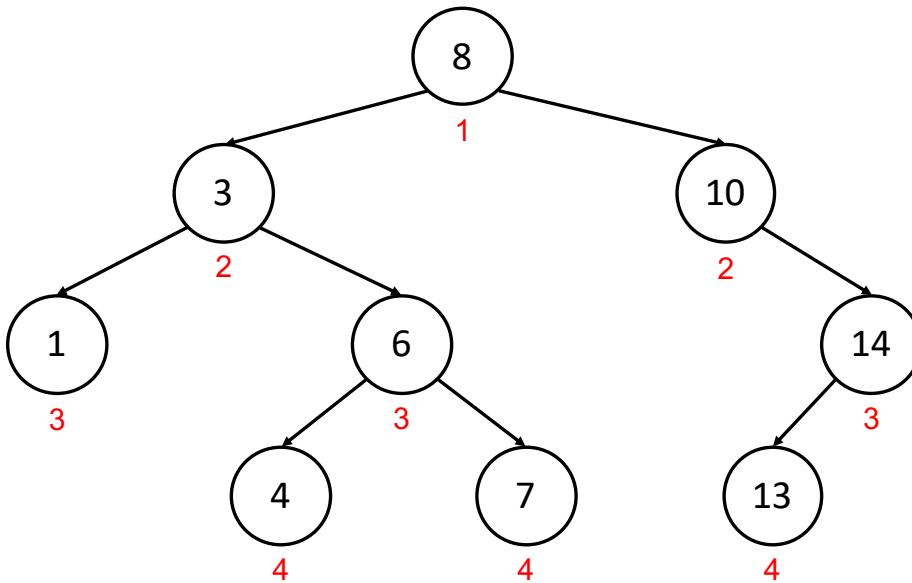
1. All n elements are equally likely to be searched for
2. All $n!$ possible insertion orders are equally likely

The first step of the proof is to simply change around nomenclature (easy enough, right?). Recall that “time complexity” is really just a metric of the number of operations it takes to execute some algorithm. Therefore, “average-case time complexity” is really just “average-case *number of operations*” (in Big- \mathcal{O} notation, as usual). Also, recall from statistics that the “average” of some distribution is simply the “expected value” (a computable value) of the distribution. In our case, our “distribution” is “number of operations,” so finding the “average number of operations” is equivalent to computing the “expected

number of operations.” In other words, “finding the average-case time complexity” to perform a successful “find” operation in a BST is simply **“computing the expected number of operations”** needed to perform a successful “find” operation in a BST.

“Number of operations” is a value we can compute during the execution of our “find” algorithm in a specific case (by simply counting the number of operations executed), but it’s a bit of an abstract term. We want to do a formal mathematical proof, so we need to use values that can be derived from the tree itself. Recall that the “find” algorithm starts at the root and traverses left or right, performing a “single comparison” (it really does 3 comparisons in the worst case, but even if it does 3 comparisons at each node, that’s still $\mathcal{O}(1)$ comparisons at each node) until it finds the node of interest. Therefore, it performs $\mathcal{O}(1)$ comparisons at each node on the path from the root to the node.

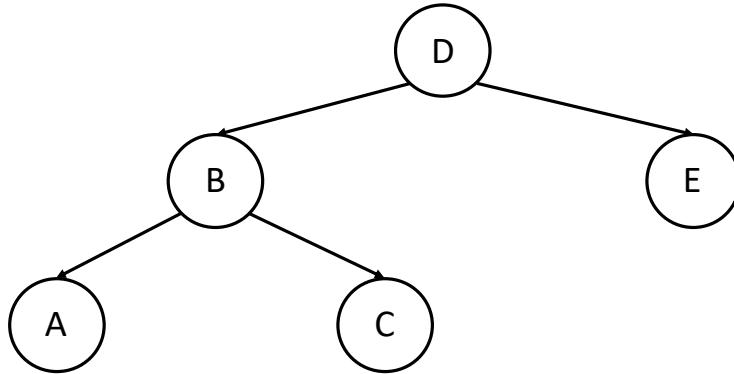
Let’s slap on a formal definition: for a node i in a BST, define the **depth** of node i , d_i , to be the number of nodes along the path from the root of the BST to i . The depth of the root is 1, the depths of the root’s children are 2, etc. The following is a BST with the depth of each node labeled below it in red:



With this definition, we can revise the value we are trying to compute: we are now **“computing the expected depth”** of a BST with n nodes.

Exercise Break

Using the same numbering scheme as discussed previously (where the root has a depth of 1), determine the depth of each node in the following tree:



Recall from statistics that the expected value of a discrete random variable X is simply $\sum_{i=1}^n p_i X_i$, where p_i is the probability that outcome X_i occurs. As we mentioned before, our discrete random variable is “depth.” In a specific BST j with n nodes, the probability of searching for a node i can be denoted as p_{ji} , and the depth of node i can be denoted as d_{ji} . Therefore, “computing the expected depth” of a specific BST j is simply computing $E_j(d) = \sum_{i=1}^n p_{ji} d_{ji}$

Remember those two assumptions we made at the beginning? The first assumption was that “all n elements are equally likely to be searched for.” Mathematically, this means $p_1 = p_2 = \dots = p_n = \frac{1}{n}$, so $E_j(d) = \sum_{i=1}^n p_{ji} d_{ji} = \sum_{i=1}^n \frac{1}{n} d_{ji} = \frac{1}{n} \sum_{i=1}^n d_{ji}$.

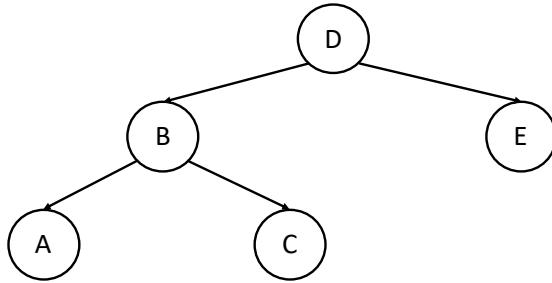
Now, let’s introduce yet another formal definition: let the **total depth** of a specific BST j with n nodes be denoted as $D_j(n) = \sum_{i=1}^n d_{ji}$. Thus, $E_j(d) = \frac{1}{n} D_j(n)$. However, this is for a specific BST j , but we need to generalize for any arbitrary BST!

Let $D(n)$ denote the **expected total depth** among ALL BSTs with n nodes. If we can solve for $D(n)$, then our answer, the expected depth of a BST with n nodes, will simply be $\frac{1}{n} D(n)$ (again because of assumption 1). However, we just introduced a new term! How will we simplify our answer?

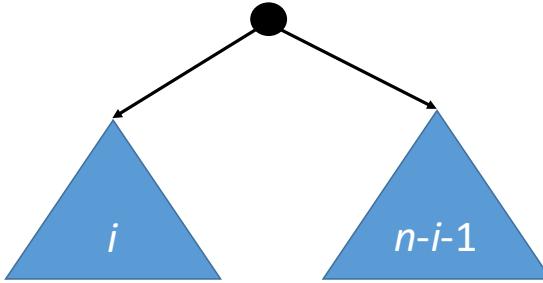
Each BST j is simply the result of insertion order j (because a BST can be defined by the order in which we inserted its elements). For the first insertion, we can insert any of our n elements. For the second insertion, we can insert any of the $n - 1$ remaining elements. By this logic, there are $n \times (n - 1) \times (n - 2) \times \dots = n!$ possible insertion orders. Based on our second assumption, all insertion orders are equally likely. Therefore, technically, we could rewrite $D(n) = \frac{1}{n!} \sum_{j=1}^n D_j(n)$. But wait... Does this mean we have to enumerate all $n!$ possible BSTs with n nodes and compute the total depth of each? This naive approach sounds far too inefficient! Let’s take a step back and rethink this.

Exercise Break

What is the total depth of the following tree?



Recall that, if we can solve $D(n)$, our answer, the expected depth of a BST with n nodes, will simply be $\frac{1}{n}D(n)$. Instead of brute-forcing the solution, let's define it as a recurrence relation. Define $D(n|i)$ as the expected total depth of a BST with n nodes given that there are i nodes in its left subtree (and thus $n - i - 1$ nodes in its right subtree). The following is a general example of such a BST:



If we want to rewrite $D(n|i)$ in terms of $D(i)$ and $D(n - i - 1)$, we can note that the resulting tree will have all of the depths of the left subtree, all of the depths of the right subtree, and a depth for the root. However, note that every node in the left subtree and every node in the right subtree will now have 1 greater depth, because each node in the two subtrees now has a new ancestor. Thus, $D(n|i) = D(i) + D(n - i - 1) + i + (n - i - 1) + 1 = D(i) - D(n - i - 1) + n$. Note that i can be at minimum 0 (because you can have at minimum 0 nodes in your left subtree) and at most $n - 1$ (because we have a root node, so at most, the other $n - 1$ nodes can be in the left subtree).

Let $P_n(i)$ denote the probability that a BST with n nodes has exactly i nodes in its left subtree (i.e., the probability of having the tree shown above). With basic probability theory, we can see that $D(n) = \sum_{i=0}^{n-1} D(n|i)P_n(i)$. Recall, however, that by definition of a BST, the left subtree of the root contains all of the elements that are smaller than the root. Therefore, if there are i elements to the left of the root, the root must have been the $(i + 1)$ -th smallest element of the dataset. Also recall that, also by definition of a BST, the root is simply the first element inserted into the BST. Therefore, $P_n(i)$ is simply the probability that, out of n elements total, we happened to insert the $(i + 1)$ -th smallest element first. Because of assumption 2, which says that all insertion

orders are equally probable (and thus all of the n elements are equally likely to be inserted first), $P_n(i) = \frac{1}{n}$, so $D(n) = \sum_{i=0}^{n-1} \frac{1}{n} D(n|i) = \frac{1}{n} \sum_{i=0}^{n-1} D(n|i) = \frac{1}{n} \sum_{i=0}^{n-1} [D(i) + D(n-i-1) + n]$.

$$\text{Therefore, } D(n) = \frac{1}{n} \sum_{i=0}^{n-1} D(i) + \frac{1}{n} \sum_{i=0}^{n-1} D(n-i-1) + n.$$

Exercise Break

Recall that, based on one of our two initial assumptions, we derived that the probability that a BST with n nodes has exactly i nodes in its left subtree is $P_n(i) = \frac{1}{n}$. Which of our two initial assumptions was this based on?

In the derived formula $D(n) = \frac{1}{n} \sum_{i=0}^{n-1} D(i) + \frac{1}{n} \sum_{i=0}^{n-1} D(n-i-1) + n$, note that the two sums are actually exactly the same, just counting in opposite directions: $\sum_{i=0}^{n-1} D(i) = D(0) + \dots + D(n-1)$ and $\sum_{i=0}^{n-1} D(i) = D(n-1) + \dots + D(0)$.

- Therefore, $D(n) = \frac{2}{n} \sum_{i=0}^{n-1} D(i) + n$
- Multiply by n : $nD(n) = 2 \sum_{i=0}^{n-1} D(i) + n^2$
- Substitute $n-1$ for n : $(n-1)D(n-1) = 2 \sum_{i=0}^{n-2} D(i) + (n-1)^2$
- Subtract that equation from the one before it: $nD(n) - (n-1)D(n-1) = 2D(n-1) + n^2 - (n-1)^2$
- Collecting terms results in the final recurrence relation: $nD(n) = (n+1)D(n-1) + 2n - 1$
- To solve this recurrence relation, divide by $n(n+1)$ to get $\frac{D(n)}{n+1} = \frac{D(n-1)}{n} + \frac{2n-1}{n(n+1)}$
- This telescopes nicely down to $n=1$: $\frac{D(1)}{2} = \frac{D(0)}{1} + \frac{2-1}{(1)(2)}$, so $D(1) = 1$ because $D(0) = 0$ (a tree with 0 nodes by definition has a total depth of 0)
- Collecting terms, we get: $\frac{D(n)}{n+1} = \sum_{i=1}^n \frac{2i-1}{i(i+1)} = \sum_{i=1}^n \frac{2}{i+1} - \sum_{i=1}^n \frac{1}{i(i+1)}$
- To simplify further, we can prove this identity for the second term (by induction): $\sum_{i=1}^n \frac{1}{i(i+1)} = \frac{n}{n+1}$
- Therefore: $\sum_{i=1}^n \frac{2}{i+1} = 2 \sum_{i=1}^n \frac{1}{i} + \frac{2}{n+1} - 2 = 2 \sum_{i=1}^n \frac{1}{i} - \frac{2n}{n+1}$
- Thus, the final closed form solution is: $D(n) = 2(n+1) \sum_{i=1}^n \frac{1}{i} - 3n$
- We can approximate $D(n)$: $\sum_{i=1}^n \frac{1}{i} \approx \int_{x=1}^n \frac{1}{x} dx = \ln(n) - \ln(1) = \ln(n)$

Therefore, the average-case number of comparisons for a successful “find” operation is approximately $2\frac{n+1}{n} \ln(n) - 3 \approx 1.386 \log_2(n)$ for large values of n . Thus, since 1.386 is just a constant, we have formally proven that, in the average case, given those two assumptions we made initially, a successful “find” operation in a Binary Search Tree is indeed $\mathcal{O}(\log n)$.

As can be seen, the average-case time complexity of a “find” operation in a Binary Search Tree with n elements is $\mathcal{O}(\log n)$. However, keep in mind that this proof depended on two key assumptions:

1. All n elements are equally likely to be searched for
2. All $n!$ possible insertion orders are equally likely

It turns out that, unfortunately for us, real-life data do not usually follow these two assumptions, and as a result, in practice, a regular Binary Search Tree does not fare very well. However, is there a way we can be a bit more creative with our binary search tree to improve its average-case (or hopefully even worst-case) time complexity? In the next sections of this text, we will discuss three “self-balancing” tree structures that come about from clever modifications to the typical Binary Search Tree that will improve the performance we experience in practice: the Randomized Search Tree (RST), AVL Tree, and Red-Black Tree.

3.5 Randomized Search Trees

In the previous section of this chapter, we were able to formally prove that, after making two assumptions about randomness, the average-case time complexity of BST insertion, removal, and finding is $\mathcal{O}(\log n)$. However, we also concluded that, in practice, those two assumptions we made were pretty unrealistic. Can we salvage all of the hard work we just did? Is there some clever way for us to simulate the same random distribution of tree topologies? If, in practice, we are somehow able to successfully simulate the same exact randomness in tree structure that we assumed in our proof, we would be able to actually experience this average-case time complexity.

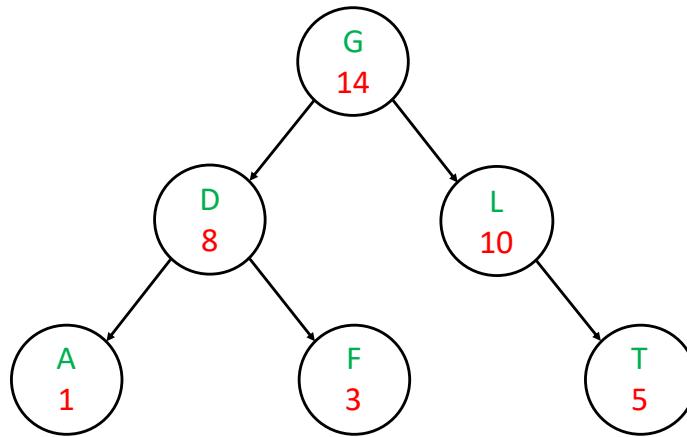
It turns out that, by tying in the *Heap Property* (from the previous section on Heaps) with some random number generation to our Binary Search Trees, we can actually achieve this simulation, and as a result, the $\mathcal{O}(\log n)$ average-case time complexity. The data structure we will be discussing in this section is the **Randomized Search Tree (RST)**.

The Randomized Search Tree is a special type of Binary Search Tree called a **Treap** (“Tree” + “Heap”). Formally, a Treap is a binary tree in which nodes contain two items, a *key* and a *priority*, and which must obey the following two restrictions:

1. The tree must follow the **Binary Search Tree properties** with respect to its **keys**

2. The tree must follow the ***Heap Property*** with respect to its **priorities**

The following is an example of a valid Treap where keys are letters, with alphabetic ordering, and priorities are integers, with numeric ordering (i.e., for the purpose of this lesson, the higher integer value will always have higher priority):



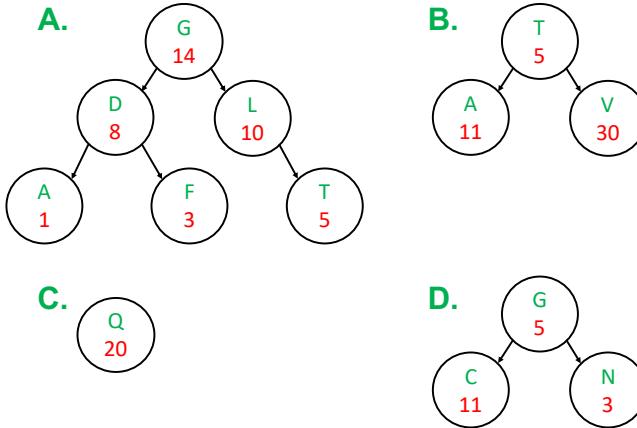
Given a set of $(key, priority)$ pairs, where all keys are unique, we can easily construct a valid Treap containing the given pairs:

- Start with an empty Binary Search Tree
- Insert the $(key, priority)$ pairs in decreasing order of *priority*, using the regular Binary Search Tree insertion algorithm with respect to the *keys*
- The resulting BST is a Treap: the BST ordering of the keys is enforced by the BST insertion algorithm, and the *Heap Property* of the priorities is enforced by inserting pairs in descending order of priorities

However, this trivial algorithm to construct a Treap relies heavily on knowing in advance *all* of the elements so that we can sort them and insert them all at once. Instead of making this unrealistic assumption, let's explore algorithms that allow for more realistic dynamic use of the structure.

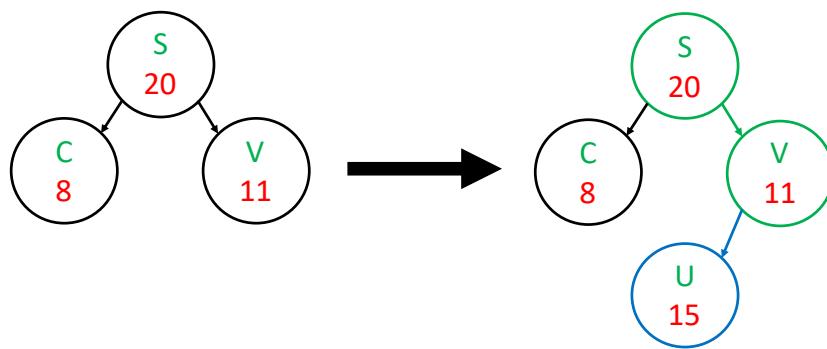
Exercise Break

Which of the following are valid Treaps?



Because a Treap is just a type of Binary Search Tree, the algorithm to find a key in a Treap is completely identical to the “find” algorithm of a typical BST: start at the root node, and then traverse left or right down the tree until you either find the desired element (or fall off the tree if the desired element doesn’t exist). The Binary Search Tree “find” algorithm is extensively described in the Binary Search Tree section of this chapter, so if you’re having trouble remembering it, be sure to reread that section.

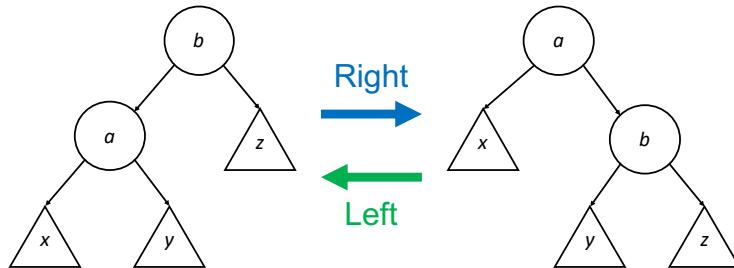
Inserting a new *(key, priority)* element into a Treap is a bit trickier. The first step is to insert the *(key, priority)* pair using the typical BST insertion algorithm (using the *key* of the pair). After the BST insertion, the resulting tree will be valid with respect to *key* ordering, but because we ignored the *priority* of the pair, there is a strong chance that the resulting tree will violate the *Heap Property*. To clarify this realization, the following is an example of a Treap just after the BST insertion algorithm, where the element inserted is (U, 15):



Notice how the Binary Search Tree properties are maintained with respect to the *keys*, but we have now violated the *Heap Property* with respect to the *priorities* (15 is larger than 11). Recall from the Heap section of the text that

we typically fix the *Heap Property* by bubbling up the newly-inserted element. Is there some way for us to bubble up the new element to fix the *Heap Property*, but without breaking the Binary Search Tree properties?

It turns out that we actually are able to “bubble up” a given node without destroying the Binary Search Tree properties of the tree via an operation called an **AVL Rotation**. This operation was originally designed for the “AVL Tree” data structure, which we will cover extensively in a later section of the text, but we will borrow the operation for use with our Treaps. AVL Rotations can be done in two directions: **right** or **left**. The following is a diagram generalizing both right and left AVL Rotations. In the diagram, the triangles represent arbitrary subtrees of any shape: they can be empty, small, large, etc. The circles are the “important” nodes upon which we are performing the rotation.



Formally, we can describe AVL Rotations with the following pseudocode:

```

1  AVLRight(b): // right AVL rotation on node b
2      a = left child of b
3      y = right child of a (or NULL if no right child)
4      p = parent of b (or NULL if no parent)
5      if p is not NULL and b is the right child of p:
6          make a the right child of p
7      otherwise, if p is not NULL and b is the left child of p:
8          make a the left child of p
9      make y the left child of b
10     make b the right child of a

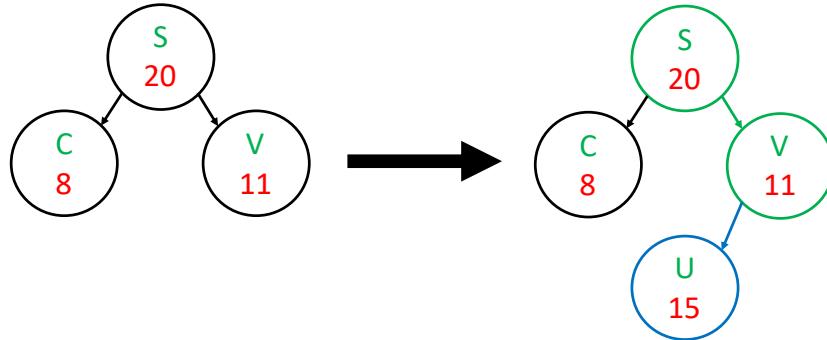
```

```

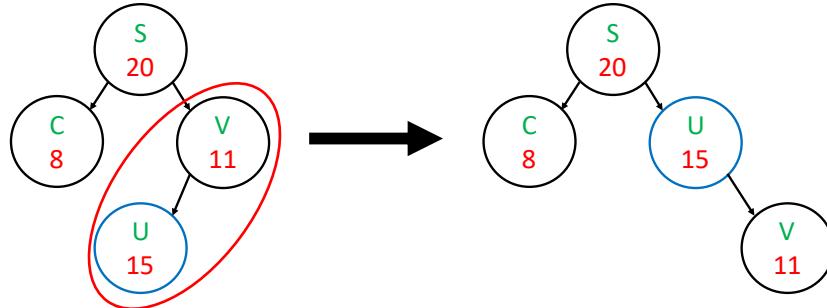
1  AVLLeft(a): // left AVL rotation on node a
2      b = right child of a
3      y = left child of b (or NULL if no left child)
4      p = parent of a (or NULL if no parent)
5      if p is not NULL and a is the right child of p:
6          make b the right child of p
7      otherwise, if p is not NULL and a is the left child of p:
8          make b the left child of p
9      make y the right child of a
10     make a the left child of b

```

Recall that we attempted to insert the element (U, 15) into a Treap, which disrupted the *Heap Property* of the tree (with respect to priorities):



Now, we will use AVL Rotations to bubble up the new element (in order to fix the *Heap Property*) without disrupting the Binary Search Tree properties:



As you can see, we succeeded! We were able to bubble up the node that was violating the *Heap Property* in a clever fashion such that we maintained the Binary Search Tree properties. The first step of the Treap insertion algorithm was the regular Binary Search Tree insertion algorithm, which is $\mathcal{O}(h)$, where h is the height of the tree. Then, the bubble up AVL rotation step performs a single rotation (which is a constant-time operation) at each level of the tree on which it operates, so in the worst case, it would perform $\mathcal{O}(h)$ AVL rotations, resulting in a worst-case time complexity of $\mathcal{O}(h)$ to perform the AVL rotation step. As a result, the overall time complexity of the Treap insertion is $\mathcal{O}(h)$.

The following is formal pseudocode for each of the three Treap operations: find, insert, and remove. Note that we did not explicitly discuss Treap removal, but just like with Treap insertion, the first step is to simply perform BST removal, and if we have broken the *Heap Property*, fix it with AVL rotations (to either bubble up or trickle down the node in violation). If you're unsure of any of the BST algorithms referenced in any of the following pseudocode, reread the BST section of this text to refresh your memory.

```

1  find(key):
2      perform BST find based on key

```

```

1 insert(key, priority):
2     // Step 1: BST Insertion Algorithm
3     n = (key, priority)
4     perform BST insertion based on key
5
6     // Step 2: Fix Heap Property via Bubble Up
7     while n is not root and n.priority > n.parent.priority:
8         if n is left child of n.parent:
9             perform right AVL rotation on n.parent
10        else:
11            perform left AVL rotation on n.parent

```

```

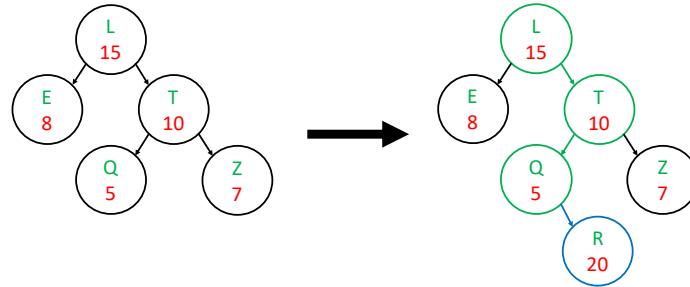
1 remove(key):
2     // Step 1: BST Removal Algorithm
3     perform BST removal based on key
4     n = node that was moved as a result of BST removal
5     if Heap Property is not violated:
6         return
7
8     // Step 2: Fix Heap Property if necessary
9     // Bubble Up if necessary
10    while n is not root and n.priority > n.parent.priority:
11        if n is left child of n.parent:
12            perform right AVL rotation on n.parent
13        else:
14            perform left AVL rotation on n.parent
15
16     // Otherwise, Trickle Down if necessary
17     while n is not a leaf:
18         if n.leftChild.priority < n.rightChild.priority:
19             lowerPriority = n.leftChild.priority
20         else:
21             lowerPriority = n.rightChild.priority
22         if n.priority < lowerPriority:
23             if n.leftChild.priority == lowerPriority:
24                 perform left AVL rotation on n
25             else:
26                 perform right AVL rotation on n
27         else:
28             break

```

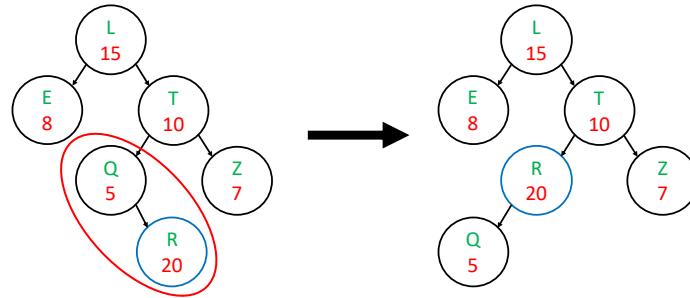
Exercise Break

Previously, we described the time complexity of insertion into a Treap with respect to the tree's height, h . With respect to the number of elements in the tree, n , what is the worst-case time complexity of Treap insertion?

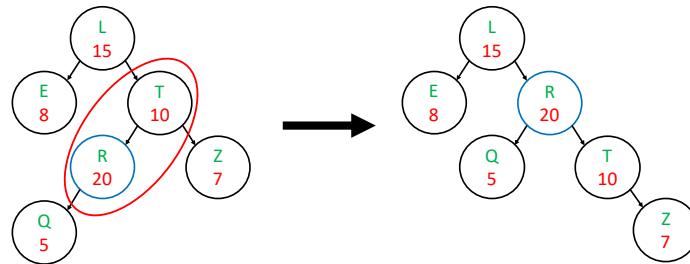
Hopefully the question about worst-case time complexity jolted your memory a bit: remember that we are jumping through all of these hoops in order to obtain a $\mathcal{O}(\log n)$ *average*-case time complexity, but the *worst*-case time complexity remains unchanged at $\mathcal{O}(n)$ if the tree is very unbalanced. Nevertheless, the following is a more complex example of insertion into a Treap. In the following example, the element being inserted is (R, 20):



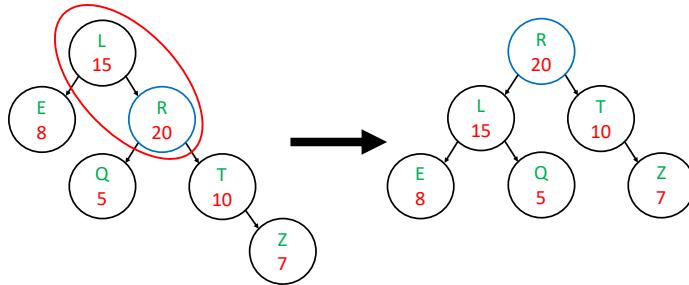
Now that we've done the naive Binary Search Tree insertion algorithm, we need to bubble up to fix the *Heap Property*. First, we see that the new element's priority, 20, is larger than its parent's priority, 5, so we need to perform an AVL rotation (a left rotation) to fix this:



Next, we see that the new element's priority, 20, is larger than its parent's priority, 10, so we need to again perform an AVL rotation (a right rotation) to fix this:



Finally, we see that the new element's priority, 20, is larger than its parent's priority, 15, so we need to perform one final AVL rotation (a left rotation) to fix this:



Again, success! We were able to insert our new $(\text{key}, \text{priority})$ element into our Treap in a manner that maintained the Binary Search Tree properties (with respect to *keys*) as well as the *Heap Property* (with respect to *priorities*). Also, hopefully you noticed something interesting: after the initial naive Binary Search Tree insertion step of the Treap insertion algorithm, the resulting tree was quite out of balance, but by chance, because of the new element's *priority*, the bubble-up step of the Treap insertion algorithm actually improved our tree's balance! This peculiar phenomenon motivates the data structure we introduced at the beginning of this section: the Randomized Search Tree.

A **Randomized Search Tree** is a Treap where, instead of us supplying both a key and a priority for insertion, we only supply a key, and the priority for the new $(\text{key}, \text{priority})$ pair is randomly generated for us. Given that a Randomized Search Tree is just a Treap, the pseudocode for its find, insert, and remove operations should seem very trivial:

```

1  find(key):
2      perform BST find based on key


---


1  insert(key):
2      priority = randomly generated integer
3      perform Treap insert based on (key, priority)


---


1  remove(key):
2      perform Treap remove based on key

```

Recall that we began this exploration of Treaps and Randomized Search Trees because we wanted to obtain the $\mathcal{O}(\log n)$ average-case Binary Search Tree time complexity we formally proved in the previous section of this text without having to make the two unrealistic assumptions we were forced to make in the proof. It turns out that, via a formal proof that is beyond the scope of this text, by simply randomly generating a priority for each key upon its insertion into our tree, we are able to in-practice (i.e., with real data) simulate the exact same random distribution of tree structures that we needed, meaning we have actually succeeded in achieving an average-case time complexity of $\mathcal{O}(\log n)$!

You might be thinking “Wait... That’s it?,” and you’re absolutely right: that’s it. To reiterate in the event that it didn’t quite register, we have simulated

the random tree topologies needed to achieve the $\mathcal{O}(\log n)$ average-case time complexity of a BST by simply creating an RST, which is a Treap in which the *keys* are the same as we would be inserting into a BST, but the *priorities* are randomly generated.

Exercise Break

With respect to the number of elements in the tree, n , what is the average-case time complexity of the Randomized Search Tree find operation?

Exercise Break

With respect to the number of elements in the tree, n , what is the *worst*-case time complexity of the Randomized Search Tree find operation?

Throughout this lesson, we were able to design a new data structure, the Randomized Search Tree, that is able to exploit the functionality of a Treap with random number generation in order to achieve the theoretical $\mathcal{O}(\log n)$ average-case time complexity *in practice* for the Binary Search Tree operations.

However, hopefully you noticed (either on your own, or via the trick questions we placed throughout this section) that, although we've obtained an *average*-case time complexity of $\mathcal{O}(\log n)$ by clever trickery, we still face our existing *worst*-case time complexity $\mathcal{O}(n)$. Is there any way we can be even *more* clever and bump the worst-case time complexity down to $\mathcal{O}(\log n)$ as well? In the next section, we will discuss the first of two tree structures that actually achieves this feat: the AVL Tree.

3.6 AVL Trees

We learned about the Binary Search Tree, which we formally proved has an *average*-case time complexity of $\mathcal{O}(\log n)$ if we were to make unrealistic assumptions. We then discussed an improved data structure, the Randomized Search Tree, which is actually able to obtain this $\mathcal{O}(\log n)$ *average*-case time complexity without the need for unrealistic assumptions. Nevertheless, in the *worst* case, both structures have an $\mathcal{O}(n)$ time complexity. Can we squeeze even more speed out of our trees?

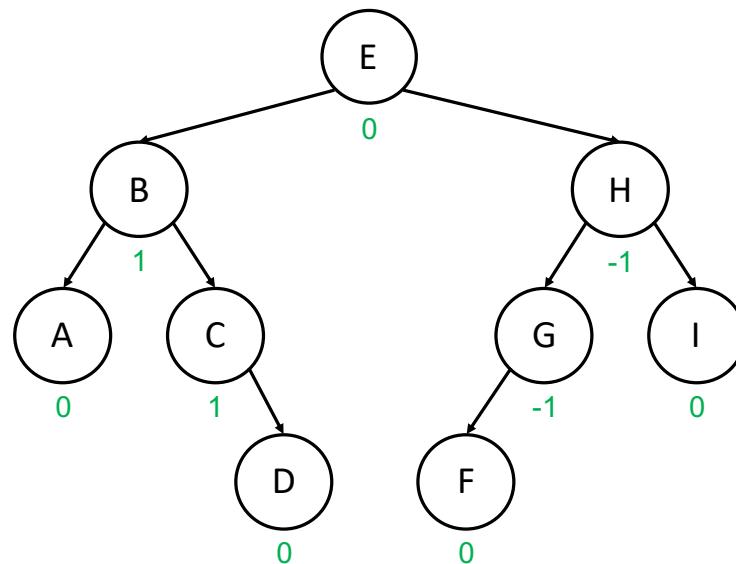
In 1962, two Soviet computer scientists, Georgy Adelson-Velsky and Evgenii Landis, forever changed the landscape of Binary Search Tree structures when they created a revolutionary self-balancing tree that achieves a *worst*-case time complexity of $\mathcal{O}(\log n)$. They named this new data structure after themselves, the Adelson-Velsky and Landis tree, or as we know it, the **AVL Tree**.

An AVL Tree is a Binary Search Tree in which, for all nodes in the tree, the *heights* of the two child subtrees of the node differ by at most one. If, at any time, the two subtree heights differ by more than one, rebalancing is done to

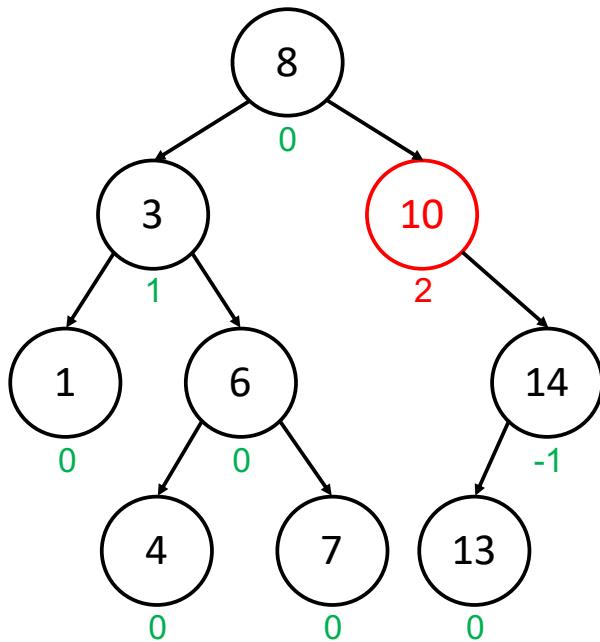
restore this property. In order to analyze the AVL Tree, we will first provide some formal definitions:

- The **balance factor** of a node u is equal to the height of u 's right subtree minus the height of u 's left subtree
- A Binary Search Tree is an AVL Tree if, for all nodes u in the tree, the balance factor of u is either -1, 0, or 1

The following is an example of an AVL Tree, where each node has been labeled with its balance factor. Note that all nodes have a balance factor of -1, 0, or 1.

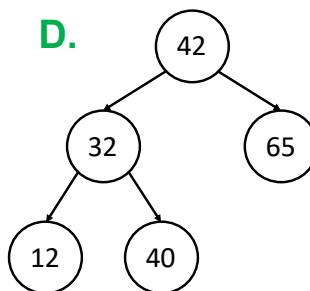
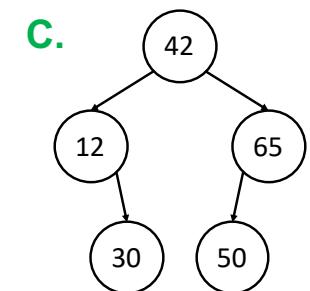
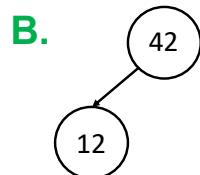
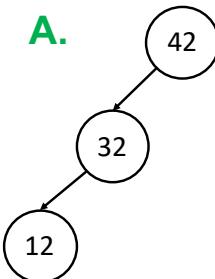


The following, however, is a Binary Search Tree that *seems* pretty balanced, but it is in fact **not** an AVL Tree: there exists a node (node 10) has an invalid balance factor (2).



Exercise Break

Which of the following trees are valid AVL Trees?



It should be intuitive that the AVL Tree restriction on balance factors keeps the tree *pretty* balanced, but as wary Computer Scientists, we have been trained to trust nobody. Can we formally prove that the height of an AVL Tree is $\mathcal{O}(\log n)$ in the worst-case? Luckily for us, this proof is *far* simpler than the Binary Search Tree average-case proof we covered earlier in this text.

First, let us define N_h as the minimum number of nodes that can form an AVL Tree of height h . It turns out that we can come up with a somewhat intuitive recursive definition for N_h . Specifically, because an AVL Tree with N_h nodes must have a height of h (by definition), the root must have a child that has height $h - 1$. To minimize the total number of nodes in the tree, this subtree would have N_{h-1} nodes.

By the property of an AVL Tree, because the root can only have a balance factor of -1, 0, or 1, the other child has a minimum height of $h - 2$. Thus, by constructing an AVL Tree where the root's left subtree contains N_{h-1} nodes and the root's right subtree contains N_{h-2} nodes, we will have constructed the AVL Tree of height h with the least number of nodes possible. In other words, we have defined a recurrence relation: $N_h = N_{h-1} + N_{h-2} + 1$

Now, we can define base cases. When we defined *height* in the Binary Search Tree section of this text, we defined the height of an empty tree as -1, the height of a one-node tree as 0, etc. For the purposes of this recurrence, however, let's temporarily change our definition such that an empty tree has a height of 0, a one-node tree has a height of 1, etc. This change in numbering only adds 1 to the height of any given tree (which does not affect Big- \mathcal{O} time complexity), but it simplifies the math in our proof. Using this numbering, we can define two trivial base cases. $N_1 = 1$ because a tree of height $h = 1$ is by definition a one-node tree. $N_2 = 2$ because the smallest number of nodes to have a tree with a height of 2 is 2: a root and a single child.

Now that we have defined a recurrence relation ($N_h = N_{h-1} + N_{h-2} + 1$) and two base cases ($N_1 = 1$ and $N_2 = 2$), we can reduce:

- $N_h = N_{h-1} + N_{h-2} + 1$
- $N_{h-1} = N_{h-2} + N_{h-3} + 1$
- $N_h = (N_{h-2} + N_{h-3} + 1) + N_{h-2} + 1$
- $N_h > 2N_{h-2}$
- $N_h > 2^{\frac{h}{2}}$
- $\log N_h > \log 2^{\frac{h}{2}}$
- $2 \log N_h > h$
- $h = \mathcal{O}(\log N_h)$

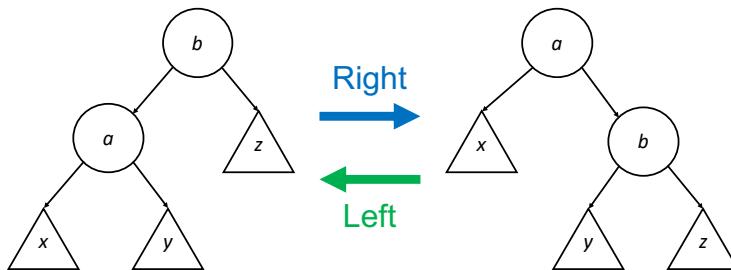
We have now formally proven that an AVL Tree containing n nodes has a height that is indeed $\mathcal{O}(\log n)$, even in the worst-case. As a result, we have formally proven that an AVL Tree has a *worst*-case time complexity of $\mathcal{O}(\log n)$.

We have formally proven that, given an AVL Tree, the worst-case time complexity for finding, inserting, or removing an element is $\mathcal{O}(\log n)$ because the height of the tree is at worst $\mathcal{O}(\log n)$. Now, let's discuss the algorithms for these three operations that are able to *Maintain* the AVL Tree properties without worsening the time complexity.

Finding an element in an AVL Tree is trivial: it is simply the Binary Search Tree “find” algorithm, which was discussed in detail in the Binary Search Tree section of this text.

Before discussing the insertion and removal algorithms of an AVL Tree, we want to first refresh your memory with regard to a fundamental operation known as the **AVL Rotation**. AVL rotations were introduced previously in the Randomized Search Tree section of this text, but since they are somewhat non-trivial and are so important to AVL Trees (hence the name “AVL” rotation), we thought it would be best to review them.

Recall that AVL Rotations can be done in two directions: **right** or **left**. The following is a diagram generalizing both right and left AVL Rotations. In the diagram, the triangles represent arbitrary subtrees of any shape: they can be empty, small, large, etc. The circles are the “important” nodes upon which we are performing the rotation.



Note that the y subtree was the right child of node a but now becomes the *left* child of node b ! Formally, we can describe AVL Rotations with the following pseudocode:

```

1  AVLRight(b): // right AVL rotation on node b
2      a = left child of b
3      y = right child of a (or NULL if no right child)
4      p = parent of b (or NULL if no parent)
5      if p is not NULL and b is the right child of p:
6          make a the right child of p
7      otherwise, if p is not NULL and b is the left child of p:
8          make a the left child of p
9      make y the left child of b
10     make b the right child of a

```

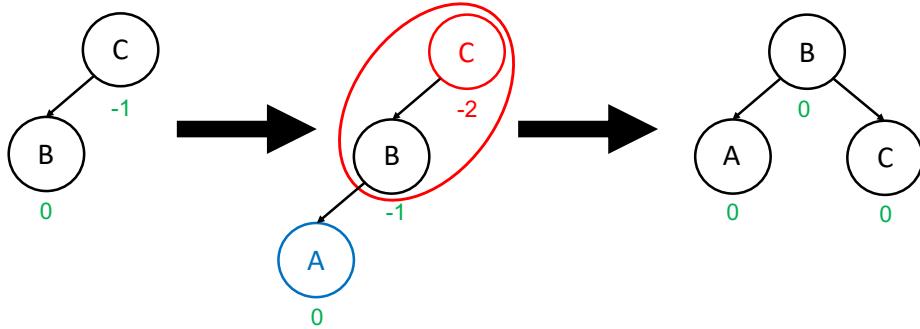
```

1  AVLLeft(a): // left AVL rotation on node a
2      b = right child of a
3      y = left child of b (or NULL if no left child)
4      p = parent of a (or NULL if no parent)
5      if p is not NULL and a is the right child of p:
6          make b the right child of p
7      otherwise, if p is not NULL and a is the left child of p:
8          make b the left child of p
9      make y the right child of a
10     make a the left child of b

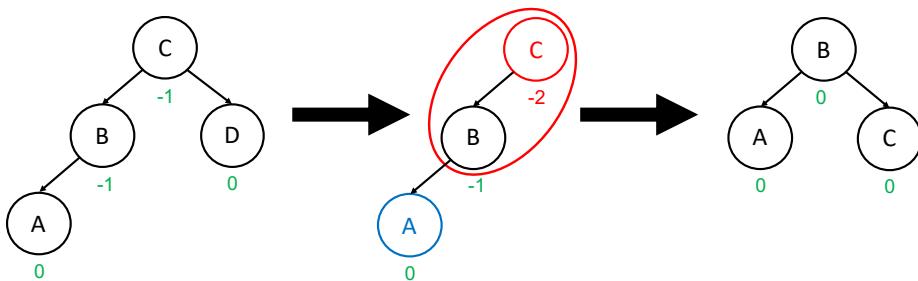
```

The AVL Tree insertion and removal algorithms rely heavily on the Binary Search Tree insertion and removal algorithms, so if you're unsure about either of them, be sure to reread the BST section of this text to refresh your memory.

To insert an element into an AVL Tree, first perform the regular BST insertion algorithm. Then, starting at the newly-inserted node, traverse up the tree and update the balance factor of each of the new node's ancestors. For any nodes that are now “out of balance” (i.e., their balance factor is now less than -1 or greater than 1), perform AVL rotations to fix the balance. The following is an example of an insertion into an AVL Tree, where the element A is inserted:



To remove an element from an AVL Tree, first perform the regular BST removal algorithm. Then, starting at the parent of whatever node was actually removed from the tree, traverse up the tree and update each node's balance factor. For any nodes that are now “out of balance,” perform AVLs rotation to fix the balance. The following is an example of a removal from an AVL Tree, where the element D is removed:

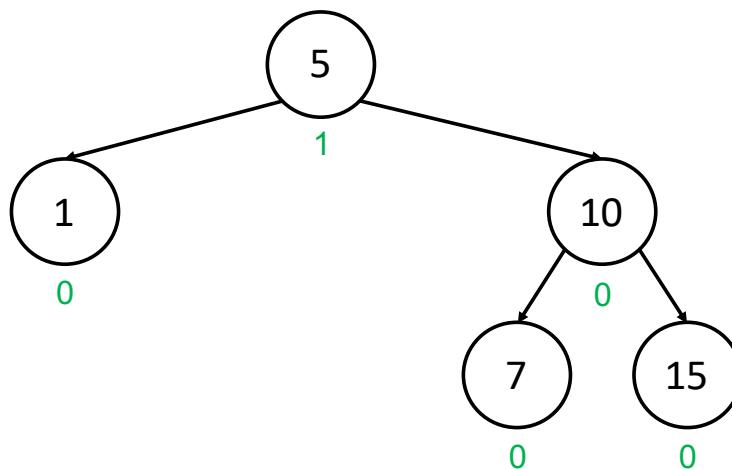


As can be seen, the AVL Tree rebalances itself after an insertion or a removal by simply traversing back up the tree and performing AVL rotations along the way to fix any cases where a node has an invalid balance factor. Because of this phenomenon where the tree balances itself without any user guidance, we call AVL Trees “self-balancing.”

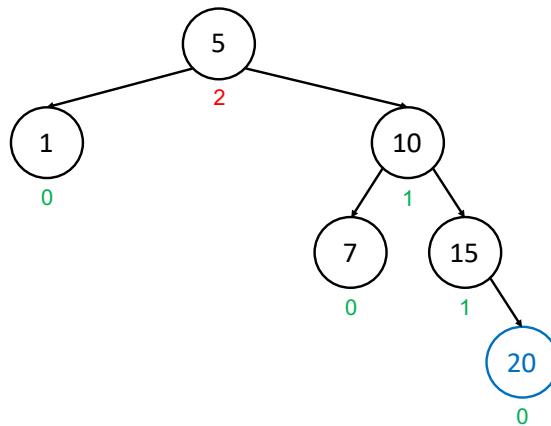
STOP and Think

Can you think of an example where a single AVL rotation will fail to fix the balance of the tree?

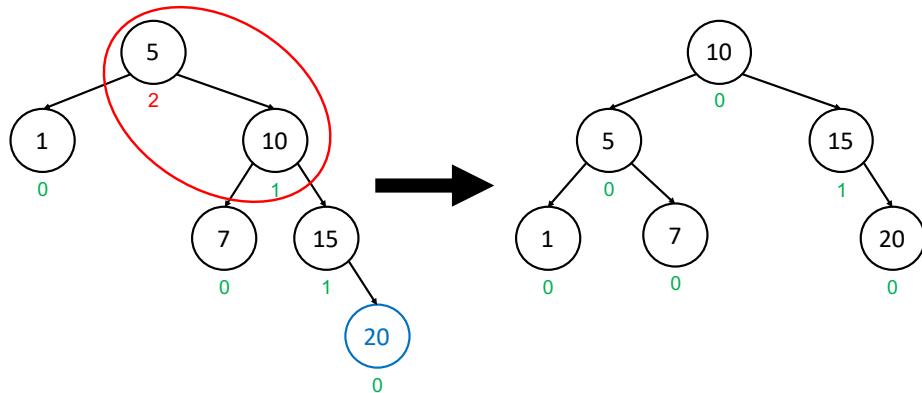
The following is a more complex example in which we insert 20 into the following AVL Tree:



Following the traditional Binary Search Tree insertion algorithm, we would get the following tree (note that we have updated balance factors as well):

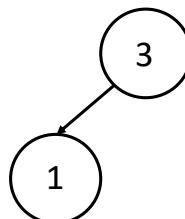


As you can see, the root node is now out of balance, and as a result, we need to fix its balance using an AVL rotation. Specifically, we need to rotate left at the root, meaning 10 will become the new root, 7 will become the right child of 10, and 5 will become the left child of 10:



Exercise Break

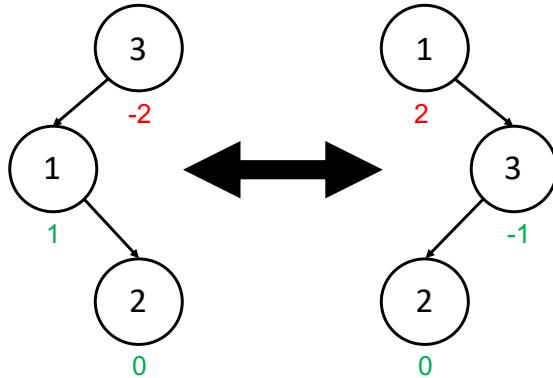
If we were to insert 2 into the following AVL Tree, which node would go out of balance?



STOP and Think

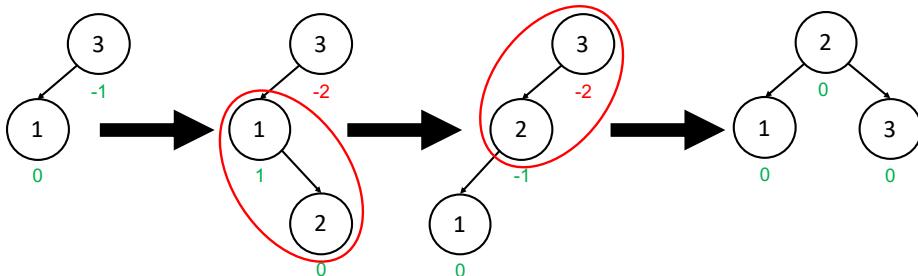
Will we be able to fix this tree the same way we fixed the trees previously?

As you may have noticed in the previous exercise, the simple AVL rotation we learned fails to fix the balance of the tree:



To combat this problem, we will introduce the **double rotation**, which is simply a series of two AVL rotations that will be able to fix the tree's balance. The issue comes about when we have a “kink” in the shape with respect to what exactly went out of balance. Contrast the above tree with the trees we saw previously, which were also out of balance, but which looked like a “straight line” as opposed to a “kink.”

The solution to our problem (a double rotation) is actually quite intuitive: we have a method that works on nodes that are in a straight line (a single rotation), so our solution to kinks is to first perform one rotation to *transform* our problematic kink *into* the straight line case, and then we can simply perform another rotation to solve the straight line case. The following is the correct way to rebalance this tree (where 2 is the node we inserted into the tree):



Formally, we can describe a double rotation using the following pseudocode, which relies on the previous AVL rotation pseudocode (a denotes the node at

the top of the kink):

```

1 DoubleAVLLeftKink(a): // Perform AVL rotation on a left kink: <
2     b = left child of a
3     c = right child of b
4     AVLLeft(b)
5     AVLRight(a)

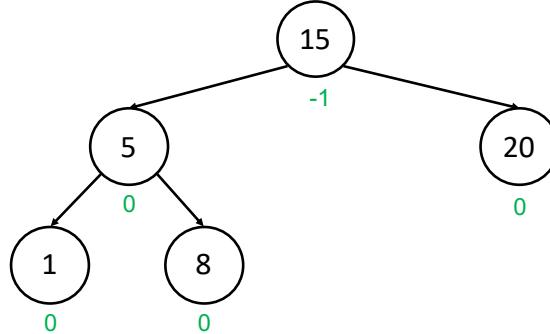
```

```

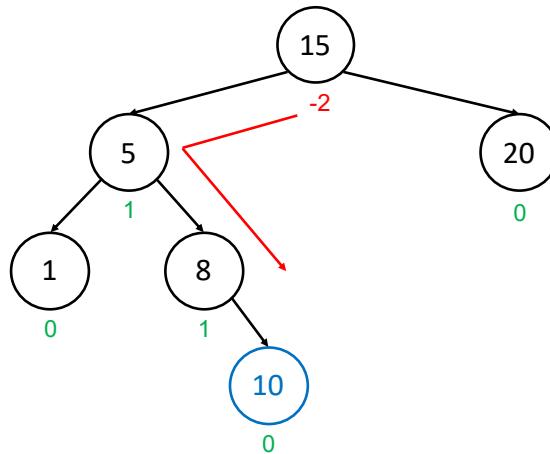
1 DoubleAVLRightKink(a): // Perform AVL rotation on a right kink: >
2     b = right child of a
3     c = left child of b
4     AVLRight(b)
5     AVLLeft(a)

```

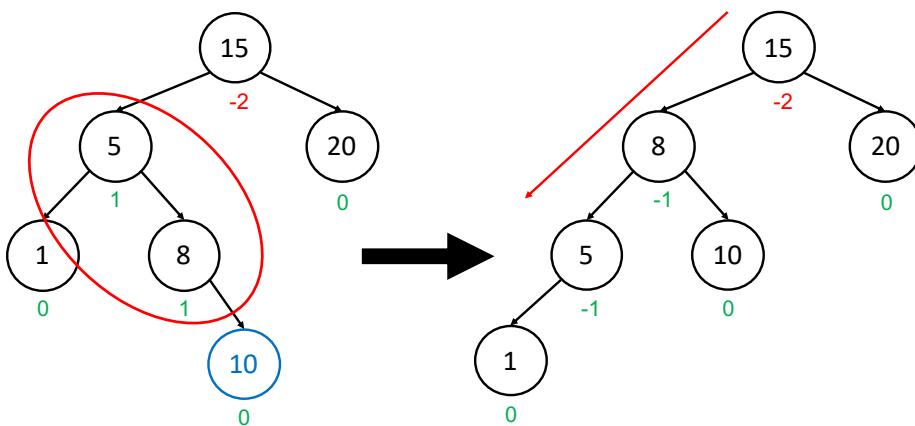
The following is a more complex example in which we insert 10 into the following AVL Tree:



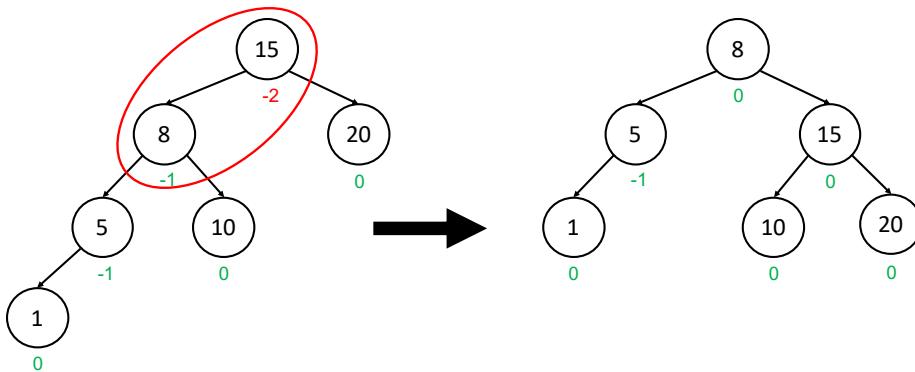
Following the traditional Binary Search Tree insertion algorithm, we would get the following tree (note that we have updated balance factors as well):



As you can see, the root node is now out of balance, and as a result, we need to fix its balance. However, as we highlighted via the bent red arrow above, this time, our issue is in a “kink” shape (not a “straight line” shape, as it was in the previous complex example). As a result, a single AVL rotation will no longer suffice, and we are forced to perform a double rotation. The first rotation will be a left rotation on node 5 in order to transform this “kink” shape into a “straight line” shape:

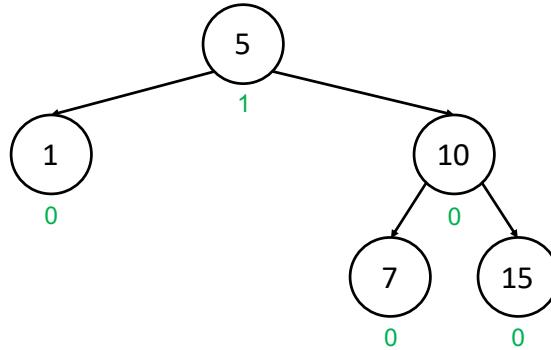


Now, we have successfully transformed our problematic “kink” shape into the “straight line” shape we already know how to fix. Thus, we can perform a right rotation on the root to fix our tree:



Exercise Break

If we were to insert 6 into the following AVL Tree, after performing the required rotation(s) to fix the tree’s balance, which node would be the new root?



We have now discussed the AVL Tree, which is the first ever Binary Search Tree structure able to rebalance itself automatically in order to guarantee a **worst-case** time complexity of $\mathcal{O}(\log n)$ for finding, inserting, and removing elements. Given that a perfectly balanced Binary Search Tree has a worst-case $\mathcal{O}(\log n)$ time complexity, you might be thinking “Wow, our new data structure is pretty darn fast! We can just call it a day and move on with our lives.” However, it is known fact that a Computer Scientist’s hunger for speed is insatiable, and in our hearts, we want to go *even faster!*

Because of the realization above about perfectly balanced trees, it is clear that, given that we are dealing with Binary Search Trees, we cannot improve the worst-case *time complexity* further than $\mathcal{O}(\log n)$. However, note that, in the worst case, an AVL Tree must perform roughly $2 \log(n)$ operations when inserting or removing an element: it must perform roughly $\log(n)$ operations to traverse *down* the tree, and it must then perform roughly $\log(n)$ operations to traverse back *up* the tree in order to restructure the tree to maintain balance. In the next section, we will discuss the Red-Black Tree, which guarantees the same $\mathcal{O}(\log n)$ worst-case time complexity, but which only needs to perform a *single* pass through the tree (as opposed to the two passes of an AVL Tree), resulting in an even *faster* in-practice run-time for insertion and removal.

3.7 Red-Black Trees

In the previous section, we learned about the AVL Tree, which is a special self-balancing Binary Search Tree that guarantees a worst-case $\mathcal{O}(\log n)$ time complexity for finding, inserting, and removing elements. However, the AVL Tree required to make two passes through the tree for inserting or removing elements: one pass down the tree to actually perform the insertion or removal, and then another pass up the tree to maintain the tree’s balance. Can we somehow avoid this second pass to speed things up a bit further?

In 1972, Rudolf Bayer, a German computer scientist, invented a data structure called the **2-4 Tree** in which the length of any path from the root to a leaf was guaranteed to be equal, meaning that the tree was guaranteed to be perfectly balanced. However, the 2-4 Tree was not a Binary Search Tree.

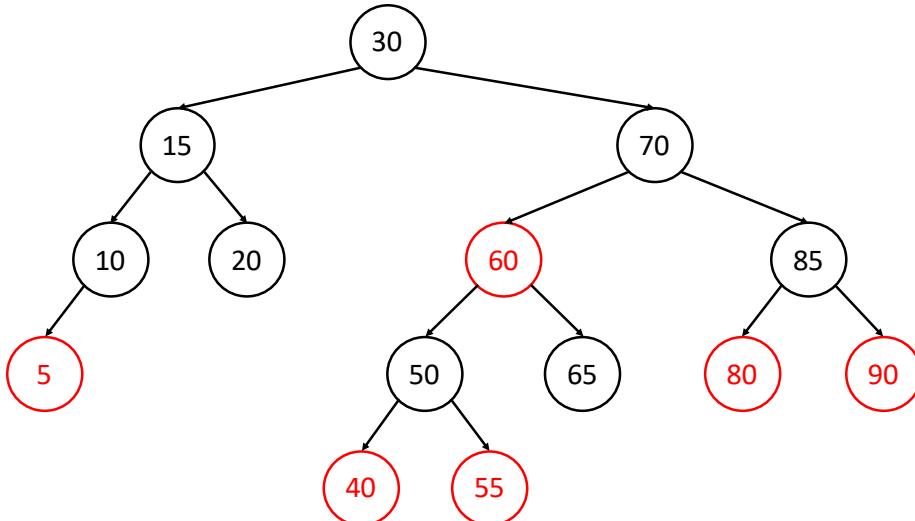
In 1978, Leonidas J. Guibas and Robert Sedgewick, two computer scientists who were Ph.D. students advised by the famous computer scientist Donald Knuth at Stanford University, derived the **Red-Black Tree** from the 2-4 Tree. Some say that the color red was chosen because it was the best-looking color produced by the color laser printer available to the authors, and others say it was chosen because red and black were the colors of pens available to them to draw the trees.

In this section, we will discuss the Red-Black Tree, which is a self-balancing Binary Search Tree, resulting in a worst-case $\mathcal{O}(\log n)$ time complexity, that is able to insert and remove elements by doing just a *single* pass through the tree, unlike an AVL Tree, which requires *two* passes through the tree.

A Red-Black Tree is a Binary Search Tree in which the following four properties must hold:

1. All nodes must be “colored” either **red** or **black**
2. The root of the tree must be **black**
3. If a node is **red**, all of its children must be **black** (i.e., we can't have a red node with a red child)
4. For any given node u , every possible path from u to a “null reference” (i.e., an empty left or right child) must contain the same number of **black** nodes

Also, not quite a property but more of a definition, “null references” are colored **black**. In other words, if some node has a right child but no left child, we assume this “phantom” left child is colored black. The following is an example of a Red-Black Tree:



Exercise Break

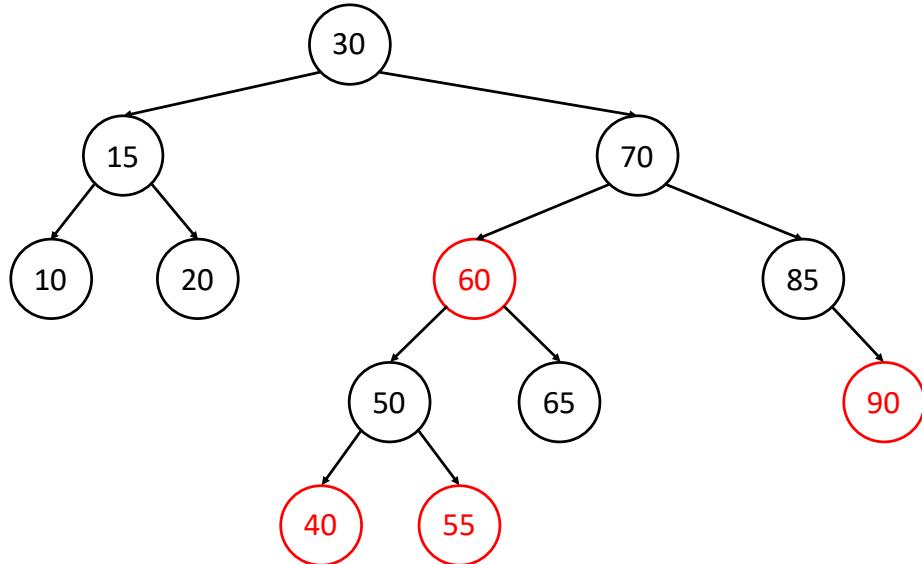
How many black nodes are in the tree above? Include null references (i.e., “phantom” children) in your count. Remember, both red and black nodes can have null references!

Exercise Break

In the same tree above, if we were to ignore node colors, is the tree also an AVL Tree?

Exercise Break

The following tree is a slightly modified version of previous example tree. Which of the following statements are true?



- The tree is not a valid AVL Tree because node 30 is out of balance
- The tree is not a valid AVL Tree because node 70 is out of balance
- The tree is a valid Red-Black Tree
- The tree is a valid AVL Tree
- The tree is not a valid AVL Tree because node 85 is out of balance

As you hopefully noticed, although *some* Red-Black Trees are *also* AVL Trees, *not all* Red-Black Trees are AVL Trees. Specifically, based on the example in the previous exercise, AVL Trees are actually *more balanced* than Red-Black

Trees, as the AVL Tree balance restrictions are stricter than the Red-Black Tree balance restrictions. Nevertheless, we can formally prove that Red-Black Trees do indeed also have a $\mathcal{O}(\log n)$ worst-case time complexity.

First, let's define $bh(x)$ to be the *black height* of node x . In other words, $bh(x)$ is the number of black nodes on the path from x to a leaf, excluding itself. We claim that any subtree rooted at x has at least $2^{bh(x)} - 1$ internal nodes, which we can prove by induction.

Our base case is when $bh(x) = 0$, which only occurs when x is a leaf. The subtree rooted at x only contains node x , meaning the subtree has 0 internal nodes (because x is a leaf, and is thus not an internal node by definition). We see that our claim holds true on the base case: $2^{bh(x)} - 1 = 2^0 - 1 = 1 - 1 = 0$ internal nodes in the subtree rooted on x .

Now, let's assume this claim holds true for all trees with a black height less than $bh(x)$. If x is black, then both subtrees of x have a black height of $bh(x) - 1$. If x is red, then both subtrees of x have a black height of $bh(x)$. Therefore, the number of internal nodes in any subtree of x is $n \geq 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 \geq 2^{bh(x)} - 1$.

Now, let's define h to be the height of our tree. At least half of the nodes on any path from the root to a leaf must be black if we ignore the root (because we are not allowed to have two red nodes in a row, so in the worst case, we'll have a black-red-black-red-black-... pattern along the path). Therefore, $bh(x) \geq \frac{h}{2}$ and $n \geq 2^{\frac{h}{2}} - 1$, so $n + 1 \geq 2^{\frac{h}{2}}$. This implies that $\log(n + 1) \geq \frac{h}{2}$, so $h \leq 2 \log(n + 1)$. In other words, we have formally proven that the height of a Red-Black Tree is $\mathcal{O}(\log n)$.

Exercise Break

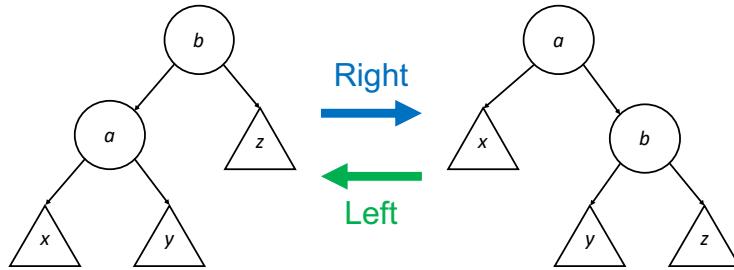
Which of the following statements are true?

- Red-Black Trees and AVL Trees have the same worst-case find, insert, and remove Big- \mathcal{O} time complexities
- Red-Black Trees and AVL Trees have the same average-case find, insert, and remove Big- \mathcal{O} time complexities
- All Red-Black Trees are AVL Trees
- The height of a Red-Black Tree can be larger than the height of an AVL Tree with the same elements
- The height of a Red-Black Tree is at most twice the height of an AVL Tree with the same elements
- Some, but not all, Red-Black Trees are AVL Trees

The insertion and removal algorithms for the Red-Black Tree are a bit tricky because there are so many properties we need to keep track of. With the AVL Tree, we had one restructuring tool at our disposal: AVL rotations. Now, we

have two tools at our disposal: **AVL rotations** and **node recoloring** (i.e., changing a black node's color to red, or a red node's color to black).

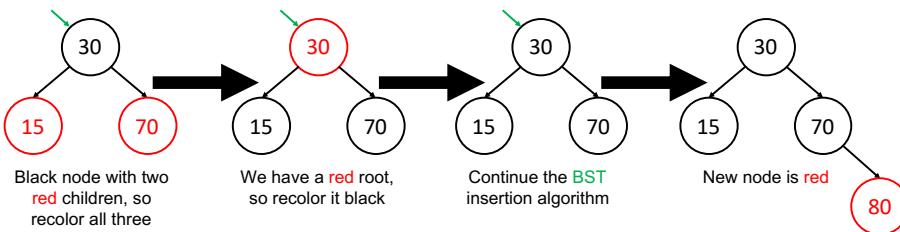
Before continuing with the Red-Black Tree algorithms, we want to first refresh your memory with regard to AVL Rotations because they are somewhat non-trivial. Recall that AVL Rotations can be done in two directions: **right** or **left**. The following is a diagram generalizing both right and left AVL Rotations. In the diagram, the triangles represent arbitrary subtrees of any shape: they can be empty, small, large, etc. The circles are the “important” nodes upon which we are performing the rotation.



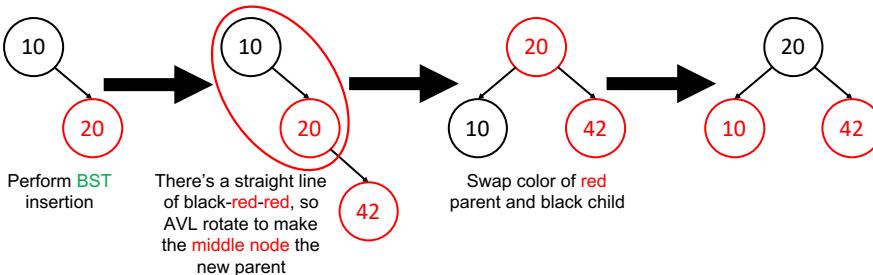
Before even going into the steps of the insertion algorithm, recall that the root of a Red-Black Tree *must* be colored black. As such, if we ever have a red root after an insertion, assuming it has no red children (which it shouldn't, assuming we did the insertion algorithm correctly), we can simply recolor the root black. Keep this in the back of your mind as you read the following paragraphs.

Also, our default color for newly-inserted nodes is always red. The motivation behind this is that, if we were to automatically color a newly-inserted node black, we would probably break the “same number of black nodes along paths to null references” restriction (Property 4) of the Red-Black Tree, and we would need to do potentially complicated restructuring operations to fix this property. By making newly-inserted nodes red, the number of black nodes in the tree is unchanged, thus maintaining the “black nodes along paths to null references” restriction (Property 4). Also keep this in the back of your mind as you read the following paragraphs.

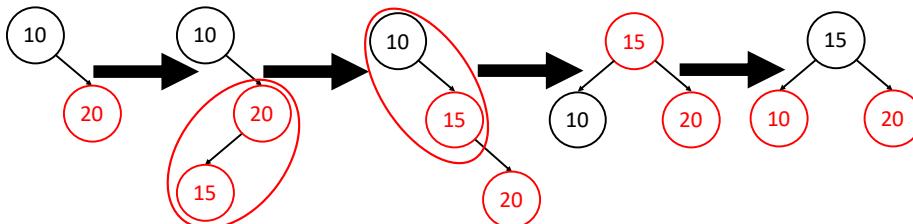
The first step to inserting an element from a Red-Black Tree is to simply perform the regular Binary Search Tree insertion algorithm, with the newly-inserted node being colored red. If this new node is the first one in the tree (i.e., it is the root), simply recolor it black. Otherwise, during your traversal down the tree in the Binary Search Tree insertion algorithm, if you ever run into a black node with two red children, recolor all three nodes (i.e., make the black node red, and make its two red children black). The following is a simple example of this process, where we insert the number 80 into the following tree:



In the example above, the newly-inserted node (80) happened to be the child of a black node, meaning we didn't violate the "red nodes cannot have red children" property by chance. However, what if we weren't so lucky, and the newly-inserted red node became the child of a red node? It turns out that we can fairly simply fix this problem using AVL rotations and recoloring. The following is an example of this process, where we insert the number 42 into the following tree:



That was pretty easy, right? We're hoping you're a bit skeptical because of what you remember seeing in the previous AVL Tree section. Specifically, this single AVL rotation was the solution because our nodes of interest were in a straight line (i.e., the black-red-red path was a *straight line*), but what about if they were in a *kink* shape (i.e., the black-red-red path had a kink)? Just like in the AVL Tree section, our simple solution is to just perform a first AVL rotation to *transform* our kink shape *into* a straight line, which we're comfortable with dealing with, and then just dealing with the straight line using a second AVL rotation. In other words, in the case of a kink, we need to perform a **double rotation**. The following is an example of this process, where we insert the number 15 into the same tree as before:



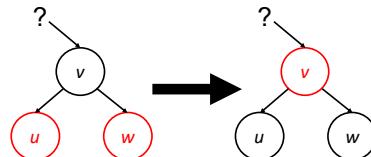
And that's it! That is the entirety of Red-Black Tree insertion. It would be a good idea to look at the “before” and “after” trees for each of the possible cases described above and verify in your own mind that the Red-Black Tree properties we mentioned at the beginning of this section remain unviolated after the new insertion.

To summarize, there are a handful of cases we must consider when inserting into a Red-Black Tree. In all of the following cases, the new node will be denoted as u . Also, a question mark denotes the rest of the tree outside of what is explicitly shown with nodes.

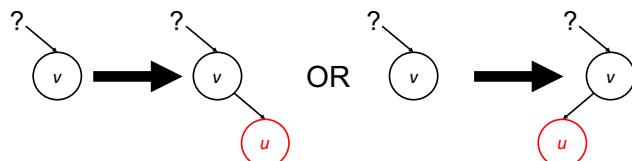
The most trivial case: if the tree is empty, insert the new node as the root and color it black:



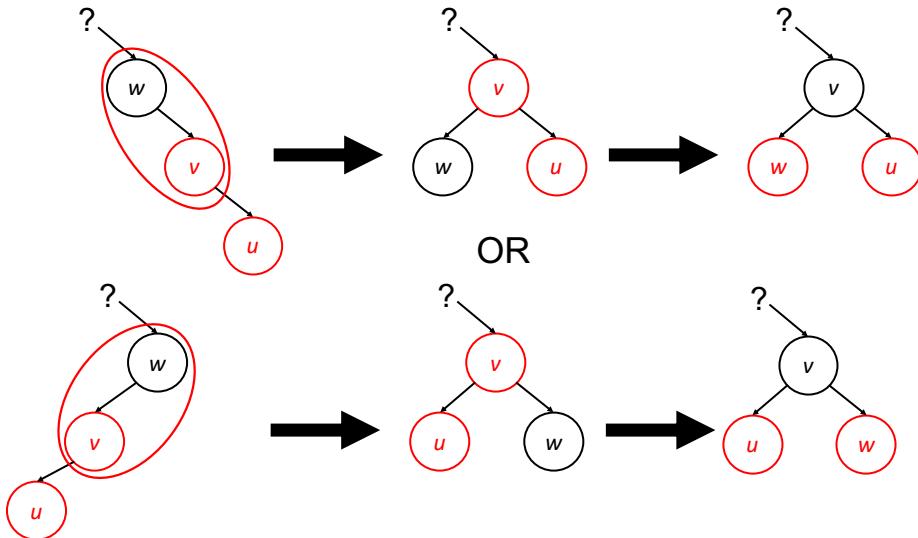
If the tree is not empty, insert the new node via the Binary Search Tree insertion algorithm, and color the new node red. While you're performing the Binary Search Tree insertion algorithm, if you run into a black node with two red children, recolor all three (and if the parent was the root, recolor it back to black):



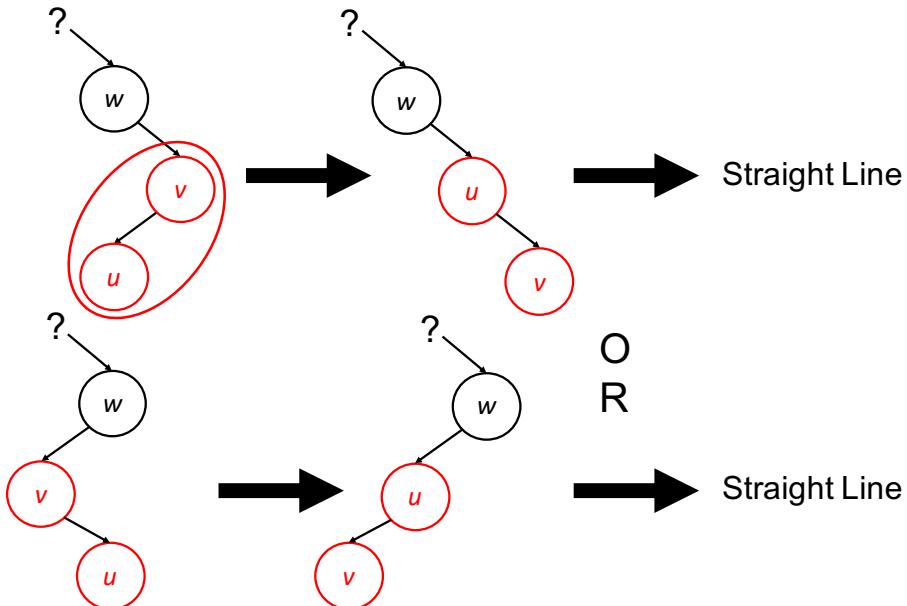
If the newly-inserted node is the child of a black node, we are done:



If the new node is the child of a red node, if the nodes are in a straight line, perform a single rotation and a recoloring:



If the new node is the child of a red node, if the nodes are in a kink, perform a single rotation to transform them into a straight line, and then perform the straight line method above:



We hope that you may be a bit confused at this point. We started this entire discussion with the motivation that we would be deviating from the AVL Tree, which is always pretty close to the optimal height of a Binary Search Tree (i.e.,

around $\log n$) and towards the Red-Black Tree for the sole intention of getting even better speed in-practice, yet we just formally proved that a Red-Black Tree can potentially be so out-of-balance that it can hit roughly twice the optimal height of a Binary Search Tree (i.e., around $2 \log n$), which would be *slower* in practice!

Our response to you is: fair enough. You are indeed correct that, given a Red-Black Tree and an AVL Tree containing the same elements, the AVL Tree will probably be able to perform slightly faster *find* operations in practice. Specifically, in the absolute worst case (where a Red-Black Tree has roughly twice the height of a corresponding AVL Tree), the Red-Black Tree will take roughly twice as long to find an element in comparison to the corresponding AVL Tree (around $2 \log n$ vs. around $\log n$ operations). If this logic wasn't clear, just note that a find operation on any Binary Search Tree takes steps proportional to the height of the tree in the worst-case because we simply traverse down the tree, so since the AVL Tree has roughly half the height of the Red-Black Tree in the absolute worst case, it performs roughly half the operations to perform a find.

However, what about *insert* and *remove* operations? Using the same exact logic, if a Red-Black Tree is extremely out of balance, it will take roughly the same amount of time to remove or insert an element in comparison to the corresponding AVL Tree (around $2 \log n$ vs. around $2 \log n$ operations). However, if a Red-Black Tree is pretty balanced, it will take roughly *half* the time to remove or insert an element in comparison to the corresponding AVL Tree (around $\log n$ vs. around $2 \log n$ operations). If this logic wasn't clear, just note that an insertion or a removal operation on a Red-Black Tree only takes one pass down the tree, whereas an insertion or removal operation on an AVL Tree takes two passes (one down, and one back up). So, if the two trees are roughly equally balanced, the Red-Black Tree performs the insertion or removal using roughly half as many operations.

In short, we typically see AVL Trees perform better with find operations, whereas we typically see Red-Black Trees perform better with insert and remove operations. In practice, most data structures we use will be updated frequently, as we rarely know all of the elements we need in advance. As a result, because insertions and removals are actually very frequent in practice, it turns out that Red-Black Trees are the Binary Search Tree of choice for ordered data structures in many programming languages (e.g. the C++ `map` or `set`).

With this, we have concluded our discussion regarding the various types of Binary Search Trees. It was quite a lengthy discussion, so the following is a brief recap of what we covered.

We first introduced the **Binary Search Tree**, a binary tree in which, for any node u , all nodes in u 's left subtree are smaller than u and all nodes in u 's right subtree are larger than u . With this property, for balanced trees, we would be able to achieve a $\mathcal{O}(\log n)$ time complexity for finding, inserting, and removing elements. We were able to formally prove that, although the *worst-case* time complexity of the three operations of a Binary Search Tree is $\mathcal{O}(n)$, under some assumptions of randomness, the *average-case* time complexity is

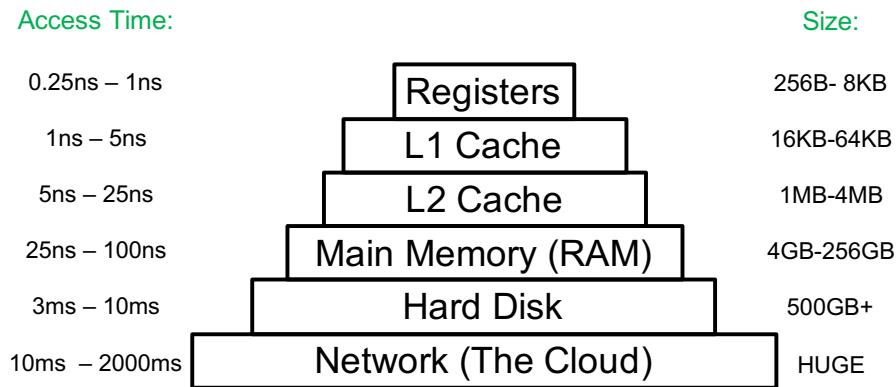
$\mathcal{O}(\log n)$. However, we also noted that the assumptions of randomness we made were somewhat unreasonable with real data.

Then, to remedy the unreasonable nature of our assumptions we made when we proved the $\mathcal{O}(\log n)$ average-case time complexity of a Binary Search Tree, we introduced the **Randomized Search Tree**, a special kind of **Treap** in which *keys* are the elements we would have inserted into a regular Binary Search Tree and *priorities* are randomly-generated integers. Even though we didn't explicitly go over the formal proof in this text, it can be proven that a Randomized Search Tree does indeed simulate the same random distribution of tree shapes that came about from the assumptions of randomness we made in our formal proof for the $\mathcal{O}(\log n)$ average-case time complexity. However, even though we explored a data structure that could actually obtain the *average*-case $\mathcal{O}(\log n)$ time complexity we wanted, we were still stuck with the original $\mathcal{O}(n)$ *worst*-case time complexity.

To speed things up even further, we introduced the **AVL Tree** and the **Red-Black Tree**, two self-balancing Binary Search Tree structures that were guaranteed to have a *worst*-case time complexity of $\mathcal{O}(\log n)$. The balance restrictions of the AVL Tree are stricter than those of the Red-Black Tree, so even though the two data structures have the same *time complexities*, in practice, AVL Trees are typically faster with find operations (because they are forced to be more balanced, so traversing down the tree without modifying it is faster) and Red-Black Trees are typically faster with insertion and removal operations (because they are not required to be as balanced, so they perform less operations to maintain balance after modification).

3.8 B-Trees

As we have been introducing different tree structures and analyzing their implementations, we have actually been making an assumption that you probably overlooked as being negligible: we have been assuming that all node accesses in memory take the same amount of time. In other words, as we traverse an arbitrary tree structure and visit a node, we assume that the time it takes to “visit” a node is the same for every node. In practice, is this assumption accurate? It turns out that, unfortunately, this assumption can’t possibly be accurate based on the way memory is organized in our computers. As you might already know (and if you don’t, we’ll bring you up-to-speed), memory in a computer is based on a hierarchical system as shown:



The basic idea behind the hierarchy is that the top layers of memory—colloquially referred to as the “closest” memory—take the shortest time to access data from. Why? Partly because they are much smaller and therefore naturally take less time to traverse to find the data.

Going back to tree traversals, consequently, if there happens to be a pointer to a node that we need to traverse that points to memory in an L2 cache, then it will take longer to traverse to that node than it would take to traverse to a node that is located in an L1 cache. Also, since pointers can point practically anywhere, this situation happens more often than you may think. How does the CPU (the Central Processing Unit) generally determine which data is placed where in memory **and** how can we take advantage of that knowledge to speed up our tree traversal *in practice*?

Note: It is a common to ask: “Why don’t we just (somehow) fit our *entire* tree structure into a fast section of memory—such as an L1 cache—to ensure that we have fast constant accesses across the entire tree?” The problem with doing this is that the sections of memory that are fast (such as the caches) are actually quite small in size. Consequently, as our data structures grow in size, they will not be able to fit in such a small section of memory. You might subsequently wonder: “Well, why don’t we just make that fast section of memory bigger?” Unfortunately, the bigger we make the memory, the slower it will be, and we don’t want slower memory!

As it turns out, the CPU determines which data to put in which sections of memory largely based on *spatial* locality. In other words, if we are accessing data that is close to other data, the CPU will also load the close data into an L1 cache because it predicts that we might need to access the close data in the near future. For example, take a look at the following for-loop:

```

1 for (int i = 0; i < 10; i++) {
2     a[i] = 0; // CPU loads a[0], but also a[1], etc. in advance
3 }
```

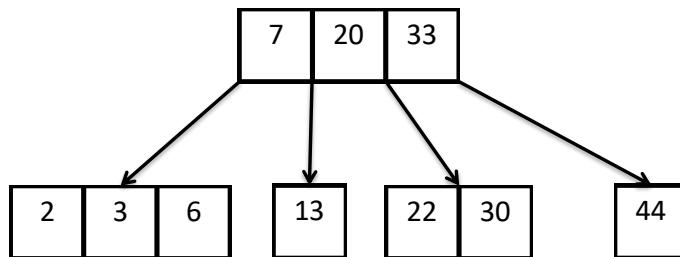
The CPU is going to try to load the entire array **a** into close memory (the

L1 cache) because it predicts that we will probably need to access other sections of the array soon after (and what a coincidence, it's right)!

So, by knowing how the CPU determines which data is placed where in memory, what can this further tell us about accessing memory in tree structures? Well, it tells us that, if a node holds multiple pieces of data (e.g. an employee object that contains `string name`, `int age`, etc.), then by loading one piece of data from the node (e.g. the employee's name), the CPU will automatically load the rest of the data (e.g. the employee's age, etc.) from the same node. This implies that accessing other data that is *inside* the node structure is expected to be faster than simply traversing to another node.

How can we take advantage of the information we just learned about spatial locality of memory to not just have a tree structure be fast *theoretically*, but also be fast *in practice*? The answer is to minimize the amount of node traversals we have to do to find data (i.e., make the trees as short as possible) **and** store as much data as possible close together (i.e., have each node store multiple keys). Why? By minimizing the amount of node traversals, we minimize the risk of having to access a section of memory that takes longer to access. By having a node store multiple keys, we are able to trick the CPU into loading more than one key at once into a fast section of memory (the L1 cache), thereby making it faster to find another key within the same node because it will already be in the L1 cache (as opposed to somewhere else in Main Memory).

The data structure that does the above is called a **B-Tree**. A B-Tree is a self-balancing tree data structure that generally allows for a node to have more than two children (to keep the tree wide and therefore from growing in height) and keeps multiple inserted keys in one node. We usually define a B-Tree to have “order b ,” where b is the minimum number of children any node is allowed to have and $2b$ is the maximum number of children any node is allowed to have. The following is an example of a B-Tree with $b = 2$:



In the B-Tree above, each integer is considered a separate key (i.e., each integer would have had its own node in a BST). Also, just like in other trees, every key in the left subtree is smaller than the current key and every key in the right subtree is greater (e.g. starting at key 7, keys 2, 3, and 6 are all smaller and key 13 is greater). Also note that, since the maximum number of children is $2 \times 2 = 4$, a node can therefore only store up to 3 keys (because pointers to children are stored *in between* keys and on the edges). More generally, we say

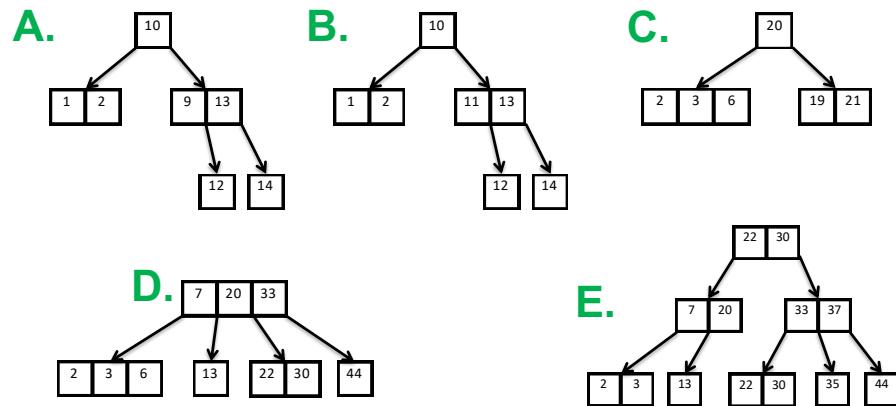
that a node can store up to $2b - 1$ keys.

Formally, we define a B-Tree of order b to be a tree that satisfies these properties:

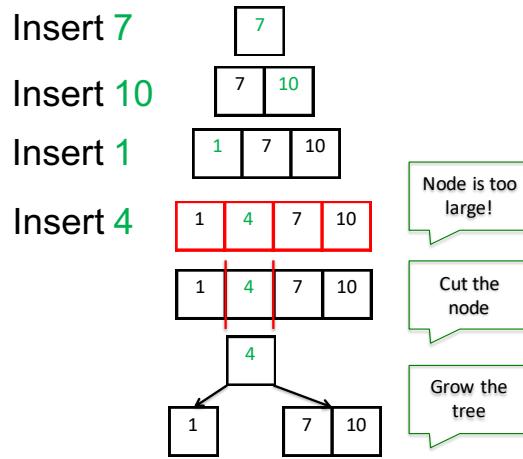
1. Every node has at most $2b$ children
2. Every internal node (except root) has at least b children
3. The root has at least two children if it is not a leaf node
4. An internal node with k children contains $k - 1$ keys
5. **All leaves appear in the same level** (We will later see how this is enforced during the insert operation)

Exercise Break

Which of the following B-Trees are valid for any arbitrary order b ? **Note:** Make sure to check that *all* B-Tree properties are satisfied for any valid tree.

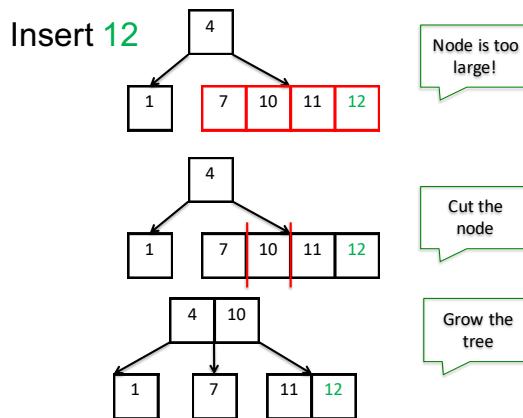


How do we insert new elements into a B-Tree? The insert operation for a B-Tree is a slightly intricate matter since, as we mentioned earlier, this is a *self-balancing* tree. Moreover, we have the requirement that *all* leaves *must* be on the **same** level of the tree. How do we achieve this? The secret to having the leaves constantly be on the same level is to have the tree grow *upward* instead of down from the root, which is an approach unlike any of the tree structures we have discussed thus far. Let's look at an example for a B-Tree in which $b = 2$:

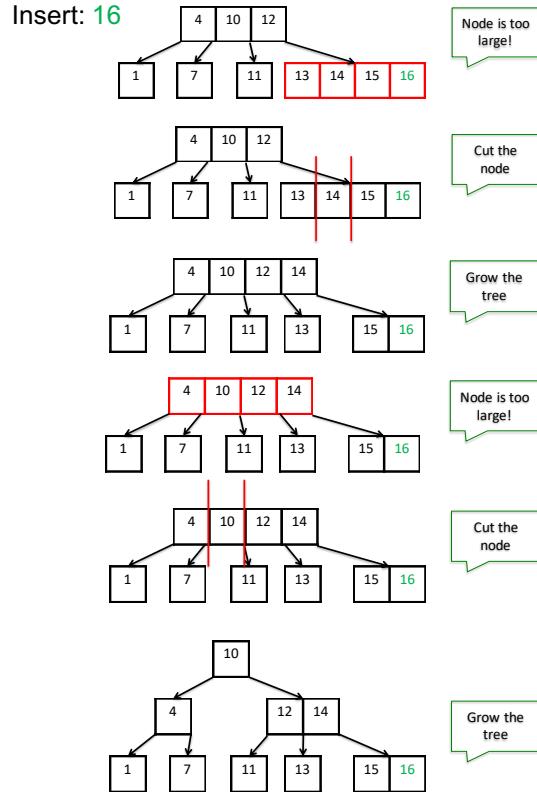


Notice how, every time we insert, we “shuffle” over the keys to make sure that the keys in our node stay properly sorted. Also, notice that, in the first 3 insertions, there was just one node and that node was the root. However, in the last insertion, where our node grew too big, we cut the node and chose the largest key of the first half of the node to create a new root (i.e., the tree grew *upward*). By doing so, we have now kept all leaves on the same level!

If we continue to insert keys into the previous example, we will see that the insertion and re-ordering process generally stays the same:



Notice how, in the example above, since the root was not full, we allowed key 10 to become a part of the root. But what happens when the root becomes full *and* we need to grow upward? The same exact steps! The keys just “bubble up” the tree like so:



The following is pseudocode more formally describing the B-Tree insertion algorithm. Note that the real code to implement insert is usually *very* long because of the shuffling of keys and pointer manipulations that need to take place.

```

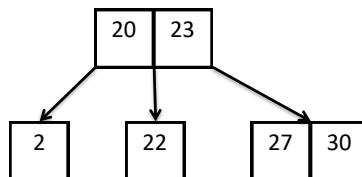
1  insert(key): // return true upon success
2      if key is already in tree:
3          return False // no duplicates allowed
4      node = leaf node into which key must be inserted
5      insert key into node (maintain sorted order)
6      allocate space for children
7
8      // while new key doesn't fit in the node, grow tree
9      while node.size >= 2*b:
10         left ,right = result of splitting node in half
11         parent = parent of node
12         remove largest key from left , and
13         insert it into parent (maintain sorted order)
14         make left and right new children of parent
15         node = parent
16     return True
  
```

What is the worst-case time complexity for a B-Tree insert operation? In each insertion, we traverse the entire height of the tree to insert the key in one of the leaves, which is $\mathcal{O}(\log_b n)$. In the worst case, during insertion, we need to re-sort the keys within each node to make sure the ordering property is kept; this is a $\mathcal{O}(b)$ operation. We also need to keep in mind that, in the worst case, the inserting node overflows, so we need to traverse all the way back up the tree to fix the overflowing nodes and re-assign pointers; this is a $\mathcal{O}(\log_b n)$ operation. Consequently, the worst-case time complexity for a B-Tree insert operation simplifies to $\mathcal{O}(b \log_b n)$.

Also, since we have been analyzing the performance of a B-Tree with respect to memory, we will also choose to analyze the worst-case time complexity with respect to memory (i.e., how much memory the CPU needs to access in order to insert a key). In the worst case, for each node, we need to read all the keys the node contains. Because the maximum number of keys we can store in a node is $2b - 1$, the number of keys the CPU needs to read in a node becomes $\mathcal{O}(b)$. As mentioned before, the tree has a depth of $\mathcal{O}(\log_b n)$, making the overall worst-case time complexity $\mathcal{O}(b \log_b n)$.

Exercise Break

Which of the following key insertion orders could have built the following B-Tree with $b = 2$?



- 2, 22, 27, 23, 30, 20
- 20, 2, 22, 23, 27, 30
- 23, 2, 20, 22, 30, 27
- 27, 30, 20, 23, 2, 22

How would we go about finding a key in a B-Tree? Finding a key is actually very simple and similar to performing binary search in a given node. The following is pseudocode describing the find algorithm, and we would call this recursive function by passing in the root of the B-Tree:

```

1 // all find operations should use binary search
2 find(key, node): // return true upon success
3     if node is empty:
4         return False
5     if key is in node:
6         return True
7     if key is smaller than node[0]: // node[0] is smallest
8         if node[0] has leftChild:
9             return find(key, node[0].leftChild)
10        else: // key can't exist in the tree
11            return False
12    nextNode = largest element of node smaller than key
13    if nextNode has rightChild:
14        return find(key, nextNode.rightChild)
15    else: // key can't exist in this tree
16        return False

```

What is the worst-case time complexity of the find operation? In the worst case, we have to traverse the entire height of the balanced tree, which is $\mathcal{O}(\log_b n)$. Within each node, we can do a binary search to find the element, which is $\mathcal{O}(\log_2 b)$. Thus, the worst-case time to find an element simplifies to $\mathcal{O}(\log n)$. However, what is the worst-case time complexity of the find operation with respect to memory access? At each level of the tree, we need to read in the entire node of size $\mathcal{O}(b)$. Consequently, the worst-case time to find an element is equal to $\mathcal{O}(b \log_b n)$.

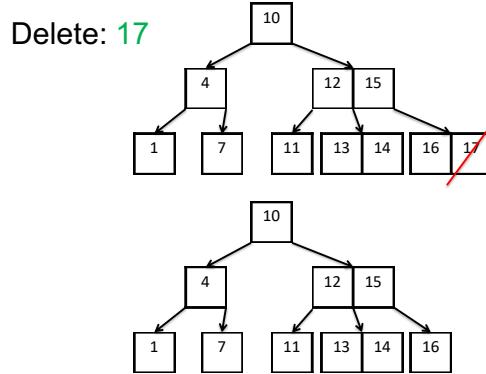
How do we delete a key from a B-Tree? If we're lazy and don't care about memory management, then we can do a "lazy deletion" in which we just mark the key as "deleted" and just make sure that we check whether a key is marked as deleted or not in the find operation. However, the whole motivation behind B-Trees is to gain fast access to elements, partly by minimizing node traversals. Thus, if we start piling up "deleted keys," our nodes will fill up much more quickly and we will have to create unnecessary new levels of nodes. By repeatedly doing so, our find and insert operations will take longer (because the height of our tree will be larger), meaning we will essentially lose the efficiency we just spent so much time trying to obtain via clever design. Consequently, most programmers choose to invest their time into implementing a more *proper* delete method.

How do we implement a proper delete method? The delete operation has 4 different cases we need to consider:

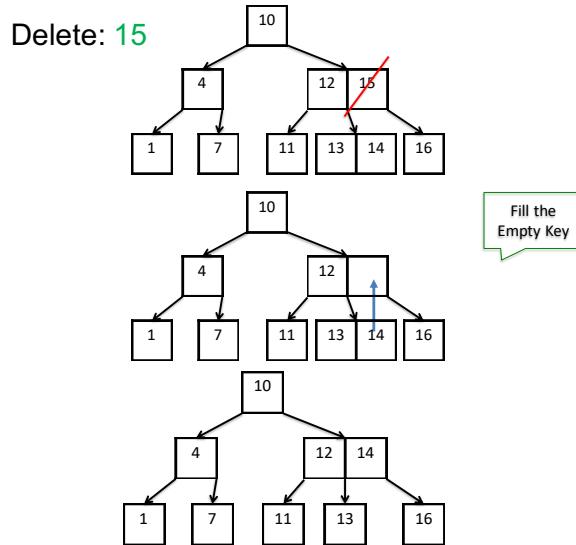
1. Delete leaf-key – no underflow
2. Delete non-leaf key – no underflow
3. Delete leaf-key – underflow, and "rich sibling"
4. Delete leaf-key – underflow, and "poor sibling"

Note: "Underflow" is defined as the violation of the "a non-leaf node with k children contains $k - 1$ keys" property.

Let's start with the easiest example, **Case 1: Delete a key at a leaf – no underflow**. Within this first case, no properties are violated if we just remove the key:



Case 2: Delete non-leaf key – no underflow. Within this second case, we face the problem of re-assigning empty key “slots” because there are pointers to leaves that need to be kept (this is different than in case 1 where we just deleted the key “slot”).



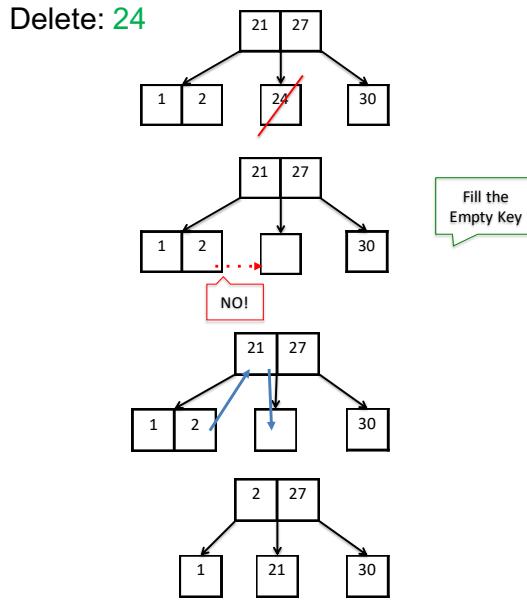
When filling the empty key, we have two different options for choosing which key we want to use as a replacement:

1. The largest key from the left subtree
2. The smallest key from the right subtree

Note: If the node from which we took a key now also has an empty “slot” that needs to be filled (i.e., it wasn’t a leaf), just repeat the same steps above recursively until we hit a leaf node!

In the next two cases of the delete operation, we now face the problem of violating the “a non-leaf node with k children contains $k - 1$ keys” property (i.e., we face an “underflow”) and thus have to restructure the tree to avoid the violation. In other words, we can’t just get rid of the “slot” as we did in the previous cases because that would lead to an internal node without enough children. In both of the following cases, we restructure the tree by taking a key from the parent node. However, the difference between the following two cases is determined by whether we also *take* a key from the immediate sibling or *merge* with a key from the immediate sibling.

Case 3: Delete leaf-key underflow, and “rich sibling”

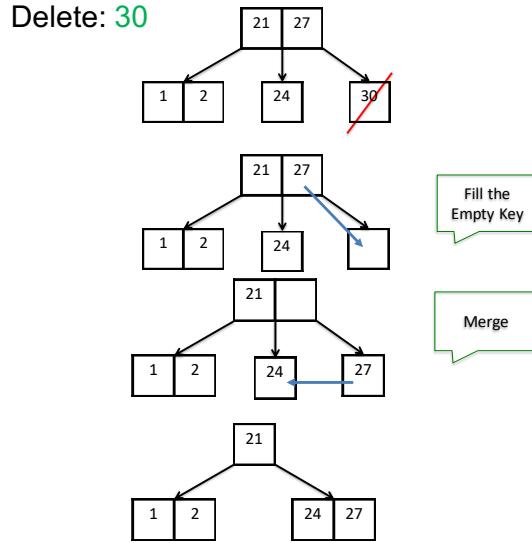


Note that, in the case above, the sibling was “rich” enough to provide a key to the parent in order for the parent to be able to give a key to the empty node.

STOP and Think

Why is it not okay to just move the sibling’s largest key to the empty node (i.e., what becomes violated)?

But what happens if the empty node doesn’t have an immediate sibling that can provide a key without causing more underflow?

Case 4: Delete leaf-key underflow, and “poor sibling”


What is the worst-case time complexity of the delete operation for a B-Tree? It turns out that it is the same as the insert operation, $\mathcal{O}(b \log_b n)$. Why? In the worst case, we need to traverse over all the keys in a node ($\mathcal{O}(b)$) in all levels of the tree ($\mathcal{O}(\log_b n)$), similarly to insert.

Note: Some programmers choose to voluntarily violate the “underflow” conditions described above in order to simplify the deletion process. However, by doing so, we risk having a taller tree than necessary, thereby making it less efficient. Also note that the real code to implement the delete operation is *extremely* long (much longer than the code to implement the insert operation) because of the excessive shuffling of keys and pointer manipulation that needs to take place.

Exercise Break

Which of the following statements regarding B-Trees are true?

- In theory, a B-Tree has the same worst-case time complexity as a regular self-balancing tree for the find operation.
- The order of insertion will not affect the structure of a B-Tree since they are self-balancing.
- A B-Tree will always have all of its leaves on the same level. There are absolutely no exceptions.
- A B-Tree achieves its optimization by keeping multiple keys in one node and making the tree as tall as possible.

As B-Trees grow large, we start to expect that the CPU will need to access the Hard Disk (the slowest section of memory in our computer) every now and then because there is no way that the rest of the tree can fit in the closer and faster sections of memory. Because it is known that the Hard Disk is designed in block-like structures (to help the CPU traverse memory more efficiently), a single node in a B-Tree is specifically designed to fit in one “Hard Disk block” to ensure that the CPU copies over *all* the data from a single node in one access.

Moreover, since B-Trees have such a good system in place for efficiently reading stored data from the Hard Disk, it is very common for B-Trees to explicitly store their data directly on the Hard Disk so we can save the B-Tree and have the ability to access it later without having to re-build it from scratch. Note that, if we do this, then the worst case number of Hard Disk reads that we will need to make is equal to the height of the B-Tree!

Databases and filesystems are common for storing large amounts of relational data on the Hard Disk. Consequently, the optimizations done behind B-Trees are extremely relevant for these systems as well. Therefore, in the next lesson, we will look at a variant of the B-Tree, called a B+ Tree, that organizes data slightly differently to help better implement systems such as databases.

3.9 B+ Trees

As we have now seen, the B-Tree is an exceptionally optimized way to store and read data on the Hard Disk. Consequently, when saving a large dataset on the Hard Disk, it is only natural to want to use the B-Tree structure to take advantage of all of its perks (i.e., it keeps our data sorted and lets us query fairly fast in practice).

A common way to store a large dataset is to use a relational database. If we wanted to store a relational database using the B-Tree structure, we would store each data *entry* (commonly stored as a row in a database, with multiple data fields stored as columns) as a single key. However, there are a couple of aspects to working with databases that make the B-Tree structure not so ideal for this purpose:

1. More often than not, database entries are actually really big (each row can have tens of columns). Also, we often only want to query a single database entry at a time. Consequently, in the worst case, if we wanted to find *one* database entry in a B-Tree, we would need to traverse $\mathcal{O}(\log_b n)$ database entries to reach it. Since our databases can consist of *millions* of large rows, traversing $\mathcal{O}(\log_b n)$ entries can be a huge waste of time to only look up *one* database entry!
2. Another common task typically performed on databases is to query all the entries at once (e.g. to output everything currently being stored). If we wanted to query all the keys in a B-Tree, we would have to traverse *up* and *down* *all* the nodes in all levels of the tree (i.e. execute a pre-order traversal), thereby having to figure out exactly *where* the pointer

to the node is located in memory (L1 cache, Main Memory, Hard Disk...) each time we go both *down* and *up* the tree. Potentially having to figure out where a node is located *twice* thus adds a large time overhead for an operation that definitely shouldn't need it.

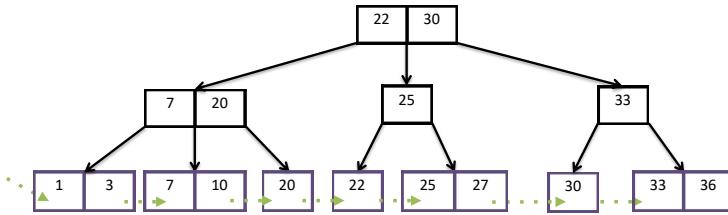
How do we take into consideration the concerns described previously to transform the B-Tree structure into a more useful structure to store large data sets?

1. If we are worried about having to traverse nodes filled with other database entries before reaching the *single* database entry we queried, then why don't we store the actual full database entries *only* at the leaves? In order to still be able to traverse the tree, however, our keys above the leaves would be kept for storing *only* the single field we used to *sort* the data records. That way, we get to traverse *smaller* nodes to reach the query. By having less data to traverse, we would increase our chances that the smaller key nodes are stored *closer to each other* in memory, thereby making it more likely that they are in a *closer section of memory*, such as the L1 Cache (remember, this is because the CPU determines which data to put in closer sections of memory largely based on *spatial* locality). By having the keys be in a closer section of memory, it will thus take less time to access the queried database entry.

2. Now that we have all the full data records guaranteed to be at the leaves, let's take advantage of this situation to create an easier way to read all the data entries *without* having to traverse up and down the tree; let's link together all the leaf nodes (i.e., implement the leaf nodes as a Linked List)! That way, we can just do a linear scan of our leaf nodes to read *all* of our database entries.

The implementation of the two ideas above in the B-Tree structure produces a modified structure called the **B+ Tree**. The following is an example of B+ Tree with the modifications we discussed. Note that the internal nodes outlined in black are filled with *keys* (i.e., the integers we use to sort the tree), and the nodes outlined in purple are the leaves filled with *all* the data records. Consequently, you will notice that some integers are repeated twice within the B+ Tree (e.g. there are two instances of the key 7: one is a *key* and another is a *data record*).

Also, note that we are using integer values to sort the tree *and* we are storing the integer values themselves. As a result, the *keys* are equal to the *data records*. However, if we were inserting large database entries, then the data records in the purple boxes would contain more information other than just the integer value.



STOP and Think

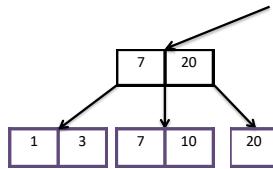
Can you see how the purple data records are sorted in the B+ Tree above?

Note: Hopefully you have noticed that the B+ Tree that we introduced *technically* violates one of the fundamental properties of a tree: the green Linked List arrows cause undirected cycles to appear in the “tree.” As a result, this presents a bit of a conundrum because, on the one hand, the B+ Tree that is used *in practice* to *actually* implement databases really does use the Linked List structure at the bottom. On the other hand, *in theory*, this is no longer a tree and is simply a graph (a *directed acyclic graph*, to be exact). What are we going to do about this? Well, since we have agreed to talk about B+ Trees, for this lesson and for the purpose of this text, we are going to omit the Linked List structure at the leaves in order to keep a B+ Tree a valid tree. Do keep in mind, however, that in practice, it is common for people to still refer to using a “B+ Tree” structure to implement their database *with* the use of the Linked List leaf structure.

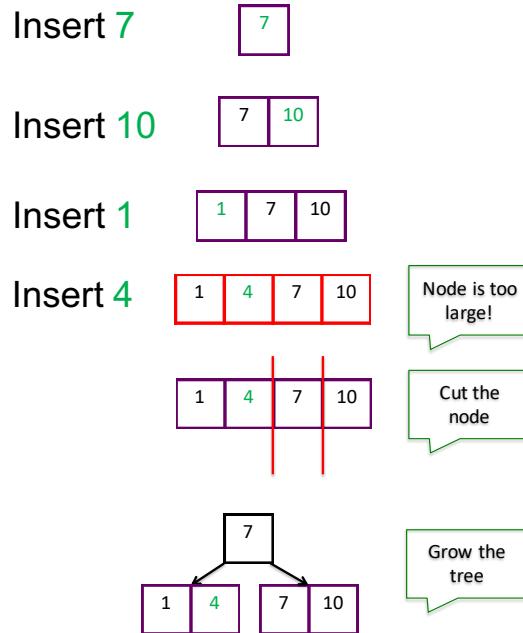
Formally, we define a B+ Tree by the values M and L , where M is equal to the maximum number of children a given node can have, and L is equal to the maximum number of data records stored in a leaf node. A B+ Tree of order M is a tree that satisfies the following properties:

1. Every node has at most M children
2. Every internal node (except the root) has at least $\lceil \frac{M}{2} \rceil$ children
3. The root has at least two children if it is not a leaf
4. An internal node with k children contains $k - 1$ keys
5. All leaves appear on the same level of the tree
6. Internal nodes contain only search keys (i.e., no data records)
7. The smallest data record between search keys x and y equals x

Note that the major difference between a B+ Tree and a B-Tree is expressed in properties 6 and 7 (the first 5 properties are shared between both). Also, note how property 7 is expressed in the following subtree, which was taken from the B+ Tree previously. For example, the *data record* 7 is stored between the *keys* 7 and 20. The *data record* 20 is stored between the *keys* 20 and NULL.



How does the insert operation of a B+ Tree work? It is actually algorithmically quite similar to the B-Tree insert operation (unsurprisingly). Just a reminder: within the B-Tree insert operation, we always inserted keys into the leaves. Once a leaf node would overflow, we would cut the overflowing node and grow the tree *upward*. The same idea applies to a B+ Tree. Let's look at an example of a B+ Tree in which $M = 3$ and $L = 3$ (i.e., the leaves can have a maximum of 3 data records and the internal nodes can have a maximum of 2 keys and 3 child pointers):



Note the two main differences between the B+ Tree insertion that happened above and the B-Tree insertion we studied previously:

1. The node was cut around the data record 7: the *smallest* data record of the *second* half of the node. This differs from what happened in the B-Tree insertion, in which we cut around the data record 4: the *largest* data record of the *first* half of the node.
2. We initially inserted elements into data record nodes (the purple nodes). Consequently, when we grew the tree, none of our data records moved up

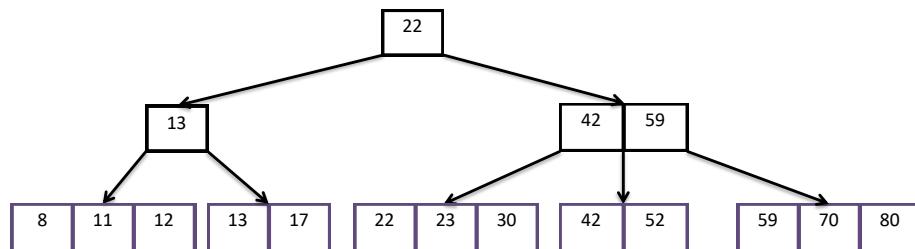
a level. Instead, we created a key node (the black node) containing the key 7 and used it to grow the tree upward. As a result, we see two instances of the integer 7 (one is the *key* and the other is the actual *data record*).

STOP and Think

Why did we choose to cut around the the *smallest* data record of the *second* half of the node? **Hint:** Look back at properties of a B+ Tree from before.

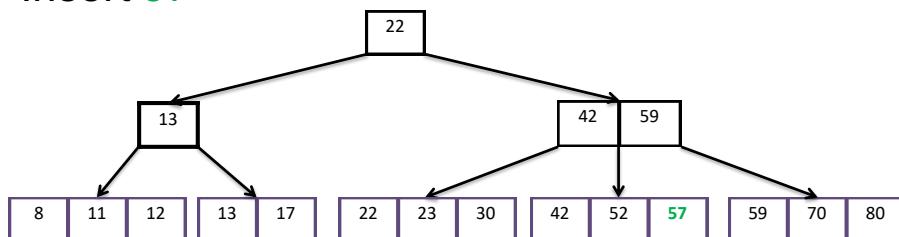
Fun Fact: A B+ Tree with $M = 3$ and $L = 3$ is commonly referred to as a “2-3 Tree.” This is because, as mentioned above, the internal nodes can have a maximum of **2** keys and the leaves can have a maximum of **3** data records.

Since we have already practiced using a very similar B-Tree insert operation with a good number of fairly simple trees in the previous lesson, let’s skip ahead and try a slightly more complicated B+ Tree structure, in which $M = 3$ and $L = 3$:

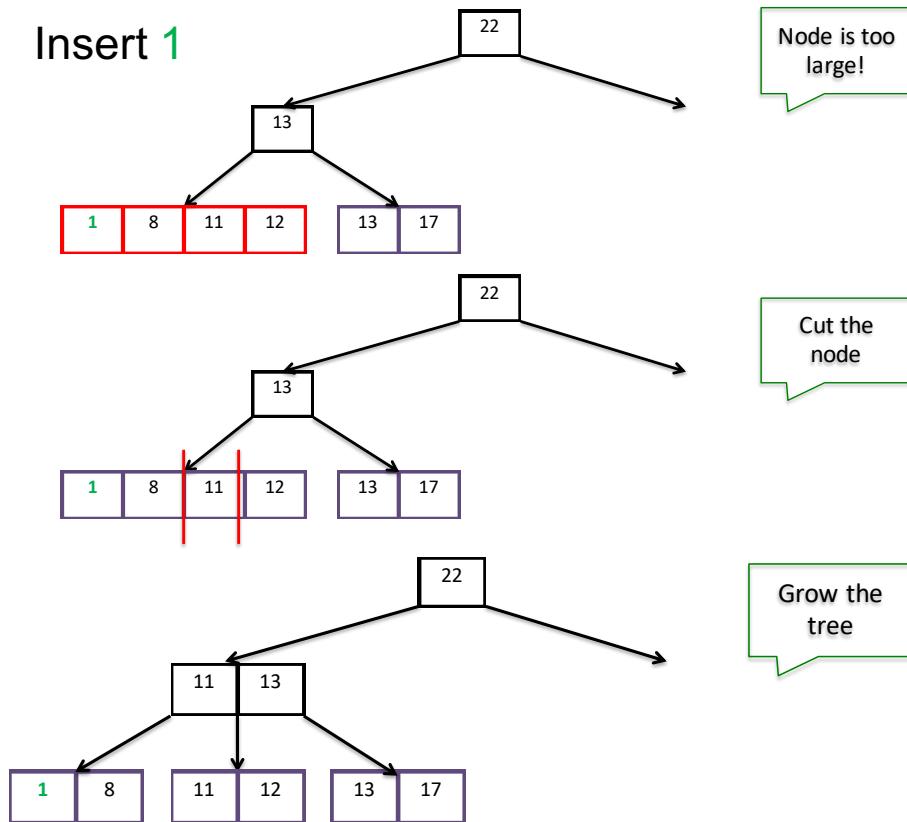


What happens when we attempt to insert the data record 57?

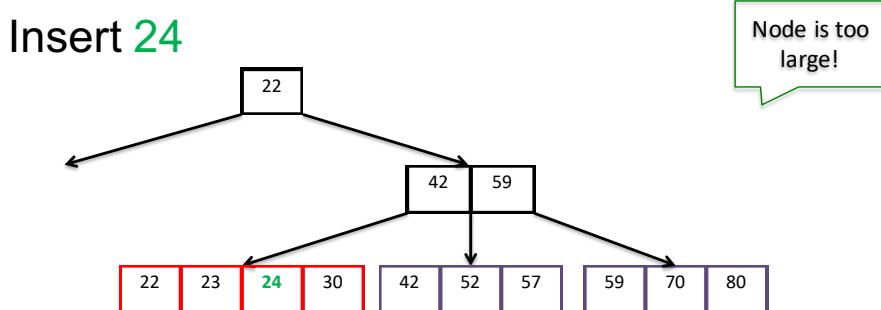
Insert 57

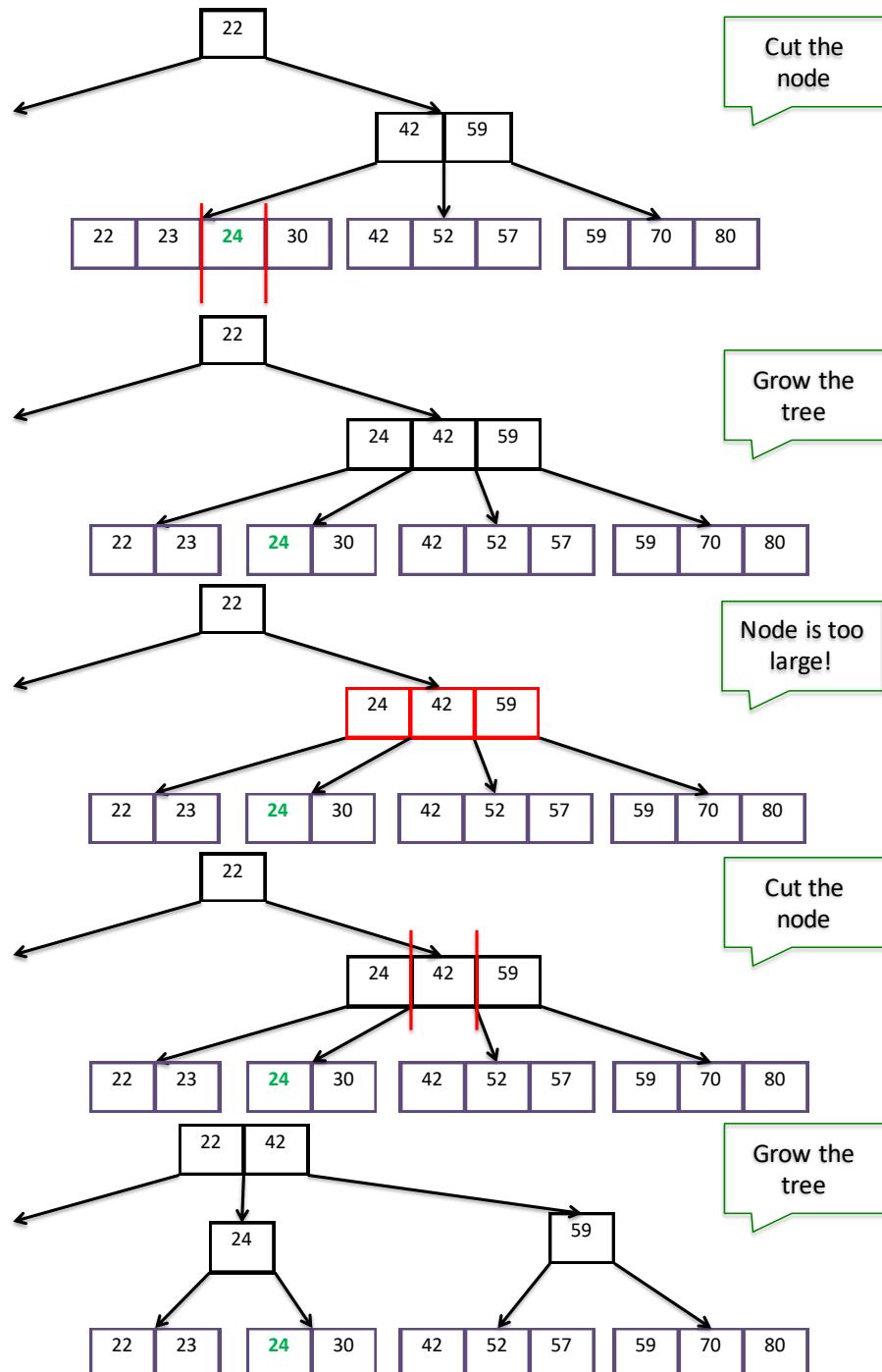


Note how no adjustments need to be made (because no *data record* nodes nor *key* nodes face an overflow) and data record 57 can be safely inserted. Does the same thing happen if we attempt to insert data record 1 into the root’s left subtree?



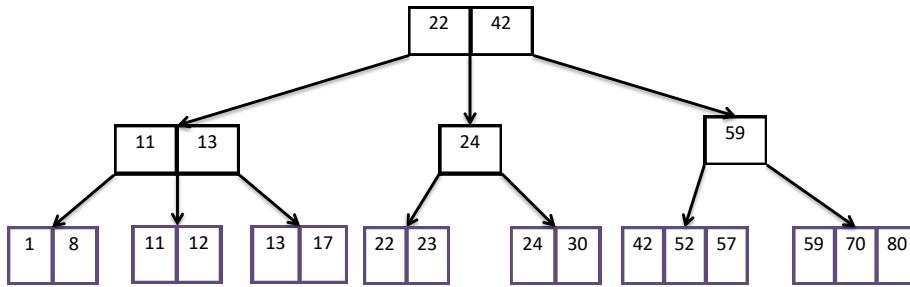
Because the inserting leaf node faced an overflow of data records, we had to resort to growing a new key node. What happens when we attempt to insert data record 24 into the root's right subtree?





Note: When growing upward directly from the leaves, we grow a *key* (i.e., we do not move the actual data record up a level)! For example, when “cutting” data record 24, a duplicate 24 appeared in the tree (i.e., the key). However, when growing upward from an internal node (i.e., a key), we do actually cut the key itself and move it up a level (i.e., a duplicate does not appear). For example, when “cutting” key 42, we actually moved the key itself up to the root.

Thus, we end up with the following monstrosity. Pay attention to how property 7—“Smallest data record between search keys x and y equals x ”—still holds after all the insertions we have made:



The following is pseudocode more formally describing the B+ Tree insertion algorithm. Note that the real code to implement insert is usually *very* long because of the shuffling of keys and pointer manipulations that need to take place.

```

1  insert(key, data): // return True upon success
2      if data is already in tree:
3          return False // no duplicates allowed
4      node = the leaf node into which data must be inserted
5      insert (key, data) into node (maintain sorted order)
6      allocate space for children
7
8      if node.size <= L: // node is not too large
9          return True // no need to fix
10
11     // while the node is too large, grow the tree
12     l,r = split node in half // if odd, r gets extra
13     p = parent of node
14     duplicate r's smallest key and insert into p (sorted)
15     make l and r new children of p
16     node = p
17     while node.size > M-1:
18         l,r = split node in half // if odd, r gets extra
19         p = parent of node
20         remove r's smallest key and insert into p (sorted)
21         make l and r new children of p
22         node = p
23     return True
  
```

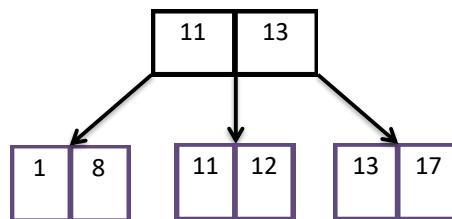
Note: We pass in a *(key, (data record)* pair into the insert method in order to provide our B+ Tree (*both* a *(means of sorting* the data record (the key) *(and* the data record itself.

Note: Duplicates of a data record are not allowed. However, different data records *can* share the same key. If different data records happen to share the same key, we can create a Linked List at the location of the previous key and append the new data record to that Linked List. Do not worry if this doesn't make much sense to you; we will explore this concept in more depth when we discuss Separate Chaining in the Hashing chapter. Consequently, for our current purposes, we will assume that all keys inserted into a B+ Tree are unique.

What is the worst-case time complexity for a B+ Tree insert operation? Hopefully it is intuitive to you that the B+ Tree insertion algorithm has the same worst-case time complexity as a B-Tree insert operation because, in both cases, we need to potentially traverse the entire height of the tree and reshuffle data records/keys at each level to maintain the sorted property. However, because a B+ Tree is expressed in terms of the variables M and L (not b), the worst-case time complexity for a B+ Tree insertion becomes $\mathcal{O}(M \log_M n + L)$. Why? In the worst case, we need to traverse over each of the M keys in an internal node, each of the L data records in a leaf node, and each of the $\mathcal{O}(\log_M n)$ levels of the tree. The worst-case time complexity with regards to *memory accesses* is also $\mathcal{O}(M \log_M n + L)$.

Exercise Break

What is the value of the *new key* node that will be generated after inserting the data record 9 into the following B+ Tree, in which $M = 3$ and $L = 2$ (i.e., the leaves can have a maximum of 2 data records and the internal nodes can have a maximum of 2 keys/3 child pointers)? What is the value of the *new root* node?



It should also hopefully be intuitive that finding a key in a B+ Tree is practically identical to a B-Tree's find algorithm! The following is pseudocode describing the find algorithm, and we would call this recursive function by passing in the root of the B+ Tree:

```

1 // all find operations should use binary search
2 find(key, data, node): // return True upon success
3     if node is empty:
4         return False
5     if node is leaf and data is in node:
6         return True
7     if key is smaller than node[0]:
8         if node[0] has leftChild:
9             return(key, data, node[0].leftChild)
10        else:
11            return False
12    nextNode = largest element of node <= key
13    if nextNode has rightChild:
14        return(key, data, node[0].rightChild)
15    else:
16        return False

```

STOP and Think

Why do we pass in *both* the key *and* the data record as a pair? In other words, why couldn't we just search for the data record by itself?

What is the worst-case time complexity of the find operation of a B+ Tree? Considering that the algorithm above is practically identical to the B-Tree find algorithm, it should hopefully be intuitive that the time complexity will be very similar. In the worst case, we have to traverse the entire height of the balanced tree, which is $\mathcal{O}(\log_M n)$. Within each internal key node, we can do a binary search to find the element, which is $\mathcal{O}(\log_2 M)$. Within each leaf node, we can do a binary search to find the element, which is $\mathcal{O}(\log_2 L)$. Thus, the worst-case time complexity to find an element simplifies to $\mathcal{O}(\log_2 n + \log_2 L)$.

However, what is the worst-case time complexity of the find operation with respect to *memory access*? At each level of the tree except the leaves, we need to read in the entire node of size $\mathcal{O}(M)$. At the level of the leaves, we need to read in the entire node of size $\mathcal{O}(L)$. Consequently, the worst-case time to find an element with respect to memory access is $\mathcal{O}(M \log_M n + L)$.

As you have probably guessed by now, deleting a data record in a B+ Tree is also very similar to deleting a key in a B-Tree. The major difference is that, because data records are *only* stored in the leaves, we will always be deleting from the leaves. However, problems start to arise when we need to deal with an existing key in the B+ Tree that all of a sudden may or may not have its respective data record in the tree (depending on whether or not it was deleted). Consequently, a lot of merging of leaf nodes and key redistributions begin to happen. Therefore, we will omit formal discussion of the deletion algorithm for a B+ Tree in this text.

It is important to note, however, that the worst-case time complexity of the delete operation for a B+ Tree turns out to be the same as the insert operation, $\mathcal{O}(M \log_M n + L)$. Why? In the worst case, we need to traverse over all M keys in an internal node, all L data records in a leaf node, and all $\mathcal{O}(\log_M n)$ levels

of the tree, similarly to insert.

Exercise Break

Which of the following statements regarding B+ Trees are true?

- The only way the height of a B+ Tree can increase is if a new root is created.
- A data record can appear twice in a B+ Tree.
- A key can appear twice in a B+ Tree.

As we have now seen, the B+ Tree is actually very similar to the B-Tree. Both are self-balancing trees that have logarithmic insertion, find, and delete operations. However, because of the two changes that we made to the B-Tree—storing data records *only* at the leaves and implementing the leaves as a Linked List (something that we admittedly ignored for the sake of keeping the B+ Tree a valid tree—the B+ Tree actually becomes extremely useful to store large data sets. It is actually so useful that it *really is* used to implement common database languages and file systems!

For example, relational database systems such as IBM DB2, Informix, Microsoft SQL Server, Oracle 8, Sybase ASE, and SQLite support the use of a B+ Tree to efficiently index into data entries. Also, your very own computer's operating system is likely using B+ Trees to organize data on your hard drive! NTFS, the file system used by all modern Windows operating systems, uses B+ Trees for directory indexing, and ext4, a file system very commonly used in a wide variety of Linux distributions, uses *extent trees* (a modified version of the B+ Tree) for file extent indexing.

This concludes our discussion of tree structures, but as you may have noticed, in this section, we had already started to deviate from the traditional definition of a “tree” by introducing the Linked List structure at the leaves of a B+ Tree that violate the formal “tree” definition. In the next chapter, we will finally expand our vision and generalize the notion of “nodes” and “edges” without the restrictions we placed on our data structures in this chapter. Specifically, in the next chapter, we will discuss graphs.

Chapter 4

Introduction to Graphs

4.1 Introduction to Graphs

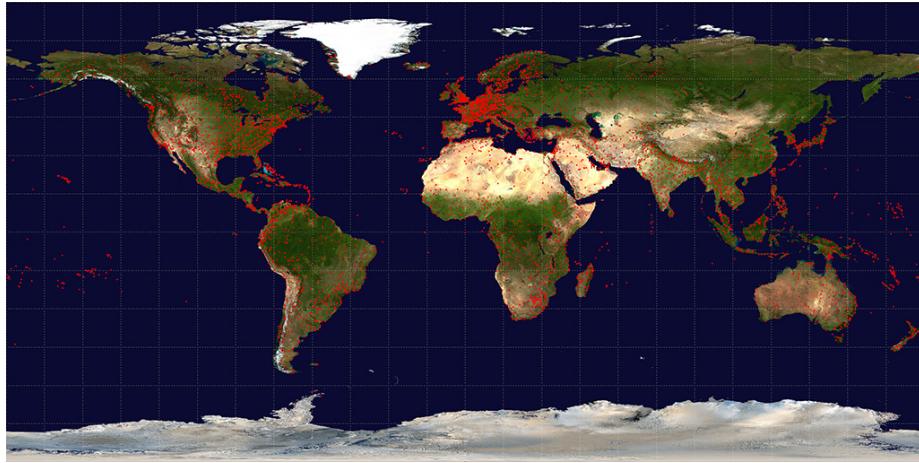
Long ago, before computers were staple items in all households, people had to use physical maps to navigate the streets of the world. As computers became more and more commonplace, services like MapQuest and later Google Maps were developed, which allowed individuals to obtain and print out turn-by-turn directions to reach their destinations. However, the moment you veered off the exact path laid out for you, it was often difficult to figure out how to get back on track. Technology advanced even further, leading to the development of portable GPS devices, which allowed for turn-by-turn directions that could be generated in real-time, and if you veered off the initial path, they would be able to generate updated directions for you on the fly. Now, we've reached a point where even portable GPS devices are obsolete, as most people are able to navigate themselves using their cell phones.

Clearly, navigation systems are essential for transportation, but how do these systems actually work under the hood? You enter a starting location and a destination, and you are magically given turn-by-turn directions for the shortest path connecting these two locations; so what's really going on behind the scenes? In this chapter, we will discuss **Graphs**, the data structures that allow for our beloved navigation systems to function.

Let's take our original idea about navigation and generalize it somewhat: let's imagine that we're given a set of "locations" (which we will refer to as **nodes**) and a set of "connections" between these locations (which we will refer to as **edges**). If we are looking at the original navigation idea, the nodes would be building addresses as well as street intersections, and the edges would be streets connecting the nodes. If we are looking instead at flights to various places in the world, where we want to find the shortest series of flights that will get us from one airport to another, the nodes would be airports, and the edges would be flights that connect airports. If we are instead looking at data packets that we need to send between ports in a network, the nodes would be

ports and the edges would be network connections between ports. With this generalization of nodes and edges, the applications of our navigation idea are endless.

The following images were obtained from OpenFlights. The following is an image of the globe, where airports are denoted as red dots:



As we mentioned previously, we can think of our airports as nodes, and we can think of flights as edges that connect pairs of nodes (airports). The following is the same image of the globe, but now with flights overlain as green curves:



It should hopefully be clear now that, if we are clever with how we represent our data as a set of nodes and a set of edges, both the navigation problem and the flight problem mentioned previously can be transformed into the same computational problem: given a node *start* and a node *end*, what is the shortest path (i.e., series of edges) to take us from *start* to *end*?

Formally, a graph consists of the following:

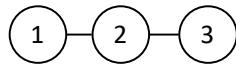
- A set of elements called **nodes** (or **vertices**)
- A set of connections between pairs of nodes called **edges**

In general, graphs do not *need* to have a sense of sequential or hierarchical order. So, unlike in trees, nodes in a graph will not necessarily have a “parent,” “successor,” etc. As a result, a graph can take up many different shapes and forms. A graph can be any of the following:

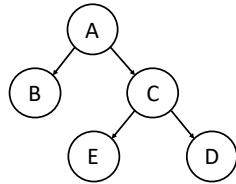
- An **unstructured** set, such as a **disconnected** group of nodes:



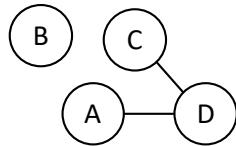
- A **sequential** set—where order matters—of **connected** nodes and edges:



- A **hierarchical** set—where rank exists (i.e. “parents,” “successors”, etc.)—of **connected** nodes and edges:

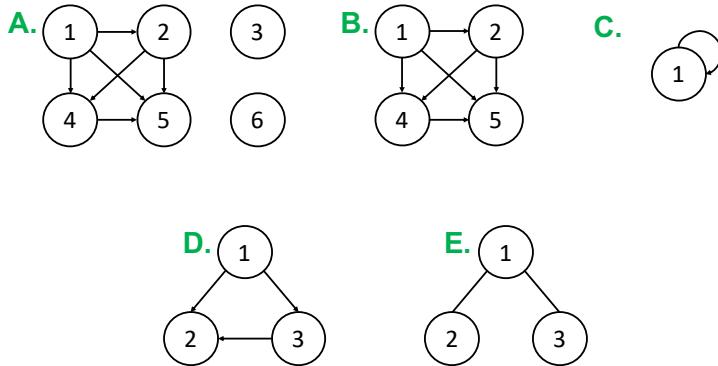


- A **structured** set of both **connected** and **disconnected** nodes and edges:

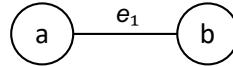


Exercise Break

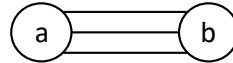
Which of the following would be defined as a graph?



The example graphs shown previously are quite simple, but in reality, graphs can be incredibly massive, to the point that drawing out a picture of nodes and edges will no longer be feasible. As a result, we need a formal mathematical definition of a “graph.” Specifically, we choose to formally represent a graph $G = (V, E)$, where V is the set of vertices in G and E is the set of edges in G . Formally, we represent edge e as a pair (v, w) such that both v and w are vertices. For example, e_1 in the following graph would be written as (a, b) .



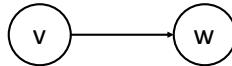
For an arbitrary graph G , the set V follows the intuitive notion of a “set” in that each vertex in V is unique. However, note that the set E is not necessarily as intuitive: there can exist multiple equivalent edges connecting the same two vertices v and w . An example of such a graph, known as a multigraph, can be found in the following figure, in which we have three edges connecting vertices a and b :



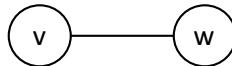
However, for the purpose of this text, we will ignore the possibility of multigraphs and operate under the assumption that we can have at most one edge connecting two given vertices. It is also useful to know that, when talking about the sizes of graphs, $|V|$ represents the total number of vertices and $|E|$ represents the total number of edges. In the first example graph above, $|V|= 2$ (because there are two vertices: a and b) and $|E| = 1$ (because there is only a single edge, one that connects a and b).

Based on the behavior of a graph’s edges, a graph is often classified as being either *directed* or *undirected*. A **directed** graph is a graph in which each edge has a direction associated with it. For example, in a directed graph, $e = (v, w)$ would mean that there is a connection from node v to w but not necessarily

a connection from node w to v . As a result, (v, w) is considered an *ordered pair* (i.e., the edge will always be leaving the first node and entering the second node).



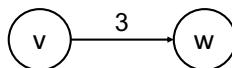
An **undirected** graph is a graph in which each edge has no particular direction associated with it. For example, in an undirected graph, $e = (v, w)$ would mean that there is a connection from node v to w and from w to v . As a result, (v, w) is considered an *unordered pair* (i.e., the order in which we write the vertices doesn't matter because we are guaranteed that edge e connects both v to w and w to v).



STOP and Think

How could you transform an *undirected* graph into a *directed* graph?

A **weighted** graph is a graph in which each edge has a “weight” (or “cost”) associated with it. As a result, we formally represent an edge $e = (v, w, c)$, where c is the “edge weight” (or “cost”). In the following example, the edge can be formally defined as $(v, w, 3)$.



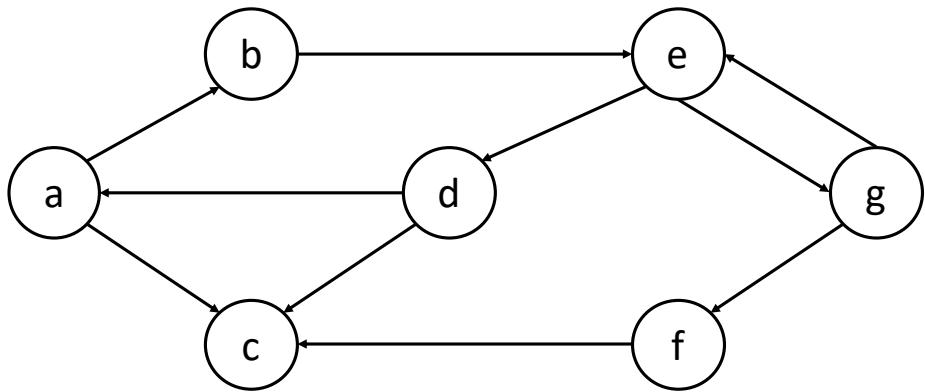
It is important to note that a “weight” can be any number (positive, negative, decimal, etc.). A weighted graph can also be either directed or undirected. An **unweighted** graph is a graph in which each edge doesn't have a “weight” associated with it. To “transform” an unweighted graph into a weighted graph, it is thus common to think of each edge as just having a weight of 1.

STOP and Think

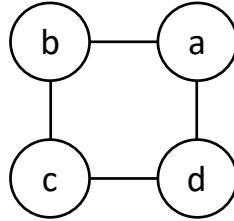
Could we transform an unweighted graph into a weighted graph by giving each edge a weight of 2 (as opposed to the previously mentioned weight of 1)?

Exercise Break

List all of the edges that appear in the following *directed unweighted* graph.



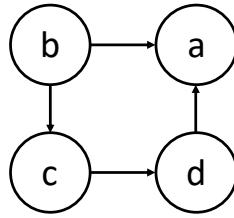
A **cycle** in a graph is a valid path that starts and ends at the same node. For example, in the following graph, one cycle would be defined by the path $a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$.



It is important to note that, just because a directed graph may have an “undirected cycle” (i.e., a cycle that is formed by removing direction from the directed edges in a graph), there need not be a cycle in the directed graph.

STOP and Think

Does the following graph contain a cycle?



We explored graphs and discussed some of their properties, and based on the previous exercises, we saw that it can be easy to interpret graphs when we can see them in their entirety. Also, we saw that we can solve some interesting real-world problems by simply exploring a graph. However, what happens when

the number of nodes and/or edges becomes absurdly large, to the point that we can no longer just “eyeball” solutions?

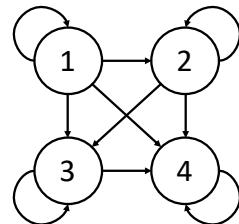
Because real-world graphs can blow up in size, before we can even begin to worry about graph exploration algorithms, we need to think about how we can represent graphs on a computer so that any graph exploration algorithms we come up with can be implemented and executed on these real-world datasets. In the next section, we will discuss various methods to represent graphs and we will analyze the trade-offs of each type of representation.

4.2 Graph Representations

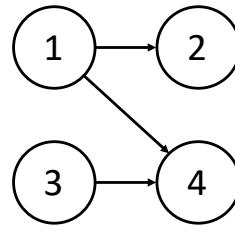
As you saw in the last section, graphs can come in many shapes and sizes. Consequently, it is a good idea to consider the properties of the graph you are dealing with to make the best decision on how to go about representing it. One good way to quantify the shape and size of a graph is to compare the number of edges in the graph, $|E|$, to the number of vertices in the graph, $|V|$.

Suppose a graph has $|V|$ vertices. How many possible edges could the graph have? We know that, at minimum, the graph could just look like a “forest” of isolated vertices, and as a result, it would have no edges ($|E| = 0$). Can we calculate an upper-bound on the number of edges? For our purposes, as mentioned in the previous section of the text, we will disallow “multigraphs” (i.e., we are disallowing “parallel edges”: multiple edges with the same start and end node). Suppose that our first vertex is connected to every other vertex in the graph (including itself). That means that we would have $|V|$ edges coming from that single vertex. Now, suppose we had the same for all vertices in the graph, not just the first one. All $|V|$ of our vertices would have exactly $|V|$ edges (one to each vertex, including itself), resulting in $|E| = |V| \times |V| = |V|^2$ edges.

We like to use the word **dense** to describe graphs with many edges (close to our upper-bound of $|V|^2$ edges for our purposes), such as the following one:

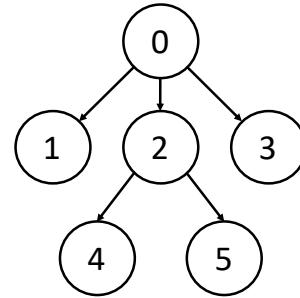


We like to use the word **sparse** to describe graphs with few edges (significantly fewer than $|V|^2$ edges for our purposes), such as the following one:



One way to represent a graph is by using an **adjacency matrix**. The following is an example graph with its corresponding adjacency matrix, M :

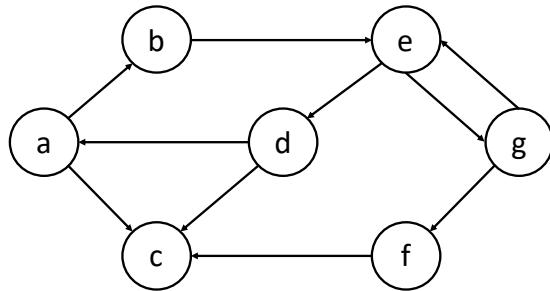
	0	1	2	3	4	5
0	0	1	1	1	0	0
1	0	0	0	0	0	0
2	0	0	0	0	1	1
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0



- Row i represents all of the edges *coming from* vertex i . Column j represents all of the edges *going to* vertex j
- A value of 1 in a given cell $M(i,j)$ means that there exists an edge from vertex i to vertex j
 - For example, the 1 in the top row, located in cell $M(0,1)$, means that there exists an edge *from* vertex 0 *to* vertex 1
 - See if you can match the rest of the non-zero entries in the matrix with the edges in the graph

The following is a more complex example of an adjacency matrix with its corresponding graph. See if you can match the non-zero entries in the matrix with the edges in the graph:

	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	0	0	0	0	1	0	0
c	0	0	0	0	0	0	0
d	1	0	1	0	0	0	0
e	0	0	0	1	0	0	1
f	0	0	1	0	0	0	0
g	0	0	0	0	1	1	0



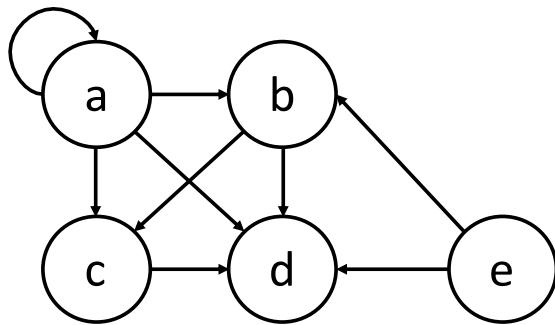
Notice that the *space complexity* (i.e., the amount of space that this matrix takes up) is $\mathcal{O}(|V|^2)$. As a result, this representation is not ideal for sparse graphs (i.e., graphs with significantly fewer than $|V|^2$ edges) because we would expect to have a lot of 0s (which do not really give us any information) and therefore a lot of wasted space.

STOP and Think

How would the adjacency matrix of an *undirected* graph look like?

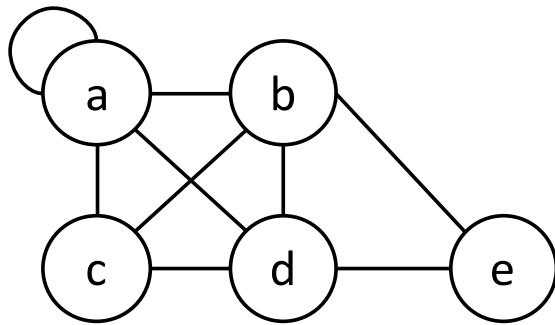
Exercise Break

Construct an adjacency matrix for the following *directed* graph.



Exercise Break

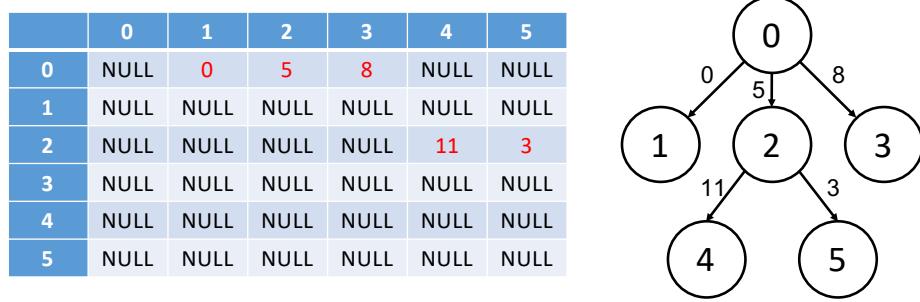
Construct an adjacency matrix for the following *undirected* graph.



How would we go about using an adjacency matrix for a weighted graph (a graph where each edge has a cost associated with)? You might have actually guessed it! We can replace the 1s in the matrix with the actual costs of the edges. The only other thing we would also need to worry about is how to define a “no edge.” At first glance, it might be tempting to just assume that we can

always keep a 0 value as `NULL`. However, in general, it is perfectly valid for edges to have a weight of 0 and thus it is up to the person implementing the matrix to make the appropriate decision as to which `NULL` value to choose.

The following is an example where we took the graph and adjacency matrix from a previous example and added some costs to all the edges.

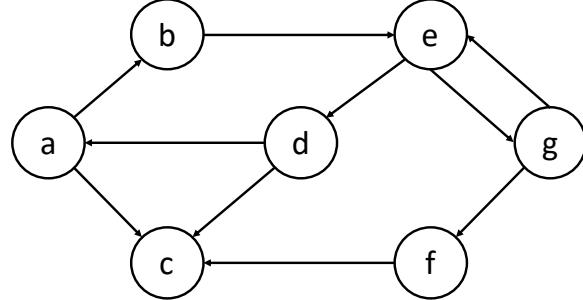


As we saw previously, an adjacency matrix is not ideal for graphs with significantly fewer than $|V|^2$ edges because of the wasted space used up by unneeded 0s. As a result, we turn to an alternative data structure to represent *sparse* graphs: the **adjacency list**. We will use the following example to describe how an adjacency list works.

```

a: {b, c}
b: {e}
c: {}
d: {a, c}
e: {d, g}
f: {c}
g: {e, f}

```



- The i -th “row” is a list representing the edges coming from vertex i
- The j -th entry in a given row i corresponds to the j -th edge coming out of vertex i

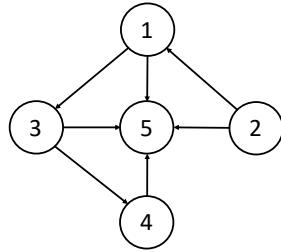
As you can probably see, if $|E|$ is relatively small, the adjacency list has a much smaller storage requirement than an equivalent adjacency matrix. Since we are storing a single list for each of our $|V|$ vertices (and each list has some constant-space overhead) and each of our $|E|$ edges is represented by a single element in one of the lists (so each edge is only accounted for once), our space complexity becomes $\mathcal{O}(|V| + |E|)$.

It is important to note that, as mentioned before, $|E|$ can be as large as $|V|^2$. If this is the case, then the space complexity for storing a dense graph

becomes $|V| + |V|^2$ and therefore will take up more space than an adjacency matrix (which would have only taken up $|V|^2$). As a result, we can conclude that an adjacency list is not ideal for dense graphs.

Exercise Break

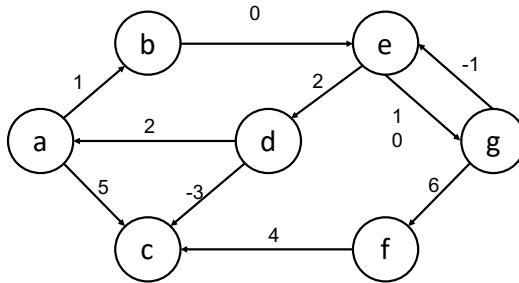
Construct an adjacency list for the following graph.



How would we go about making an adjacency list for a *weighted* graph? This might be a bit less intuitive than it was for the adjacency matrix since we no longer have a space where we could just write in the weight of each edge. As a result, one of the ways to represent a weighted graph would be to *make* space to store the weights of each edge. For example, we could store each entry of our adjacency list as a pair (v, c) , where v would be the destination vertex (just as before) and c would now be the cost of the edge. Take a look at the following example:

```

a: {(b,1), (c,5)}
b: {(e,0)}
c: {}
d: {(a,2), (c,-3)}
e: {(d,2), (g,10)}
f: {(c,4)}
g: {(e,-1), (f,6)}
    
```



Now that we have learned about different ways to store different types of graphs, it is time to discover how we can go about using these representations to not just *store* a graph, but to also be able to *efficiently traverse* it.

4.3 Algorithms on Graphs: Breadth First Search

Now that we have learned about the basics of graphs and graph representation, we can finally return to the original real-world problems we had discussed. The first real-world problem was the “navigation problem”: given a starting location

start and a destination location *end*, can we find the shortest driving path from *start* to *end*? The second real-world problem was the “airport problem”: given a starting airport *start* and a destination airport *end*, can we find the shortest series of flights from *start* to *end*? The third real-world problem was the “network routing problem”: given a starting port *start* in a network and given a destination port *end* in a network, can we find the shortest path in the network from *start* to *end*?

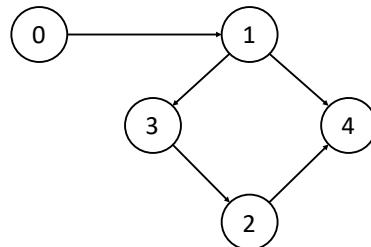
It should be clear by now that all three of these problems, as well as all other real-world problems that are similar in nature, are actually reduced to the exact same computational problem when they are represented as graphs: given a starting node *start* and a destination node *end*, what is the shortest path from *start* to *end*? In the next sections, we will discuss various graph algorithms that solve this “shortest path problem.”

Fun Fact: In many cases, you will often only be interested in finding the shortest path from a particular vertex to another *single* particular vertex. However, as of today, there is no known algorithm which can run more efficiently by only dealing with a particular single destination vertex as opposed to just finding the shortest path from one to *all* other vertices. As a result, the best optimization that we can do is to include an “early out” option and hope that it happens early enough in the computation (i.e. return from the algorithm once it has computed the shortest path from the source to the destination, but not necessarily all the rest of the vertices).

STOP and Think

Why does including an “early out” option **not** improve the worst-case time complexity of a shortest path algorithm (or any algorithm, really)?

One algorithm to explore all vertices in a graph is called **Breadth First Search (BFS)**. The idea behind BFS is to explore all vertices reachable from the current vertex before moving onto the next. We will use the following graph as an example.



Suppose we start at 0. BFS will then choose to “visit” 1 because it is immediately reachable from 0. From 1, BFS will choose to “visit” 3 and 4 (not simultaneously, but in some arbitrary order) because they are both immediately

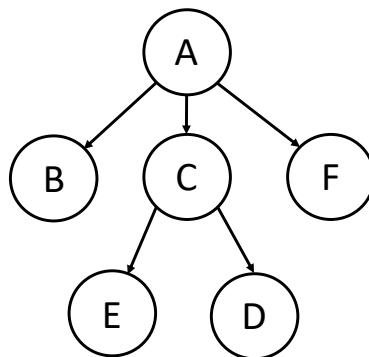
reachable from 1. After, BFS will “visit” 2 as it reachable from 3. The rough algorithm for BFS can be thought of as the following:

1. Begin at the starting node s . It has distance 0 from itself.
2. Consider nodes adjacent to s . They have distance 1. Mark them as visited.
3. Then consider nodes that have not yet been visited that are adjacent to those at distance 1. They have distance 2. Mark them as visited.
4. Repeat until all nodes reachable from s are visited.

Breadth First Search can be difficult to understand without being able to actually see the algorithm in action. As a result, we have created a visualization that shows the order in which each vertex is explored, which can be found at <https://goo.gl/6M4ZLn>.

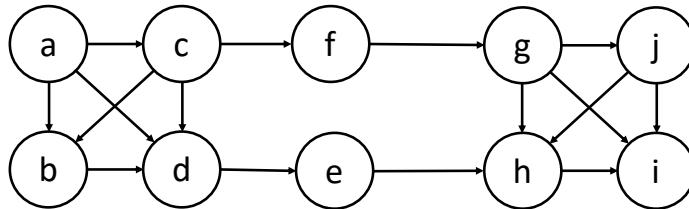
Exercise Break

In what order would Breadth First Search explore the vertices in the following graph when starting with vertex A ? Break ties in order-of-exploration by giving priority to the the vertex that contains the alphabetically smaller letter.



Exercise Break

In what order would Breadth First Search explore the vertices in the following graph when starting with vertex a ? Break ties in order-of-exploration by giving priority to the the vertex that contains the alphabetically smaller letter.



In terms of the actual implementation of Breadth First Search, we take advantage of the **queue** ADT to determine the order in which to explore vertices. The intuition behind choosing a queue for BFS stems from a queue's *first-in first-out* property. With this property, we guarantee that we will explore all the vertices enqueued (put on the queue) first, before moving on to the vertices adjacent to the next dequeued vertex.

```

1  BFSShortestPath(u,v):
2      q = empty queue
3      add (0,u) to q // (0,u) = (length from u, current vertex)
4      while q is not empty:
5          (length,curr) = q.dequeue()
6          if curr == v: // found v
7              return length
8          for all edges (curr,w): // explore all neighbors
9              if w has not yet been visited:
10                 add (length+1,w) to q
11      return "FAIL" // no path exists from u to v

```

We have created a visualization of how the Breadth First Search algorithm takes advantage of the queue ADT to find the shortest path between two nodes, which can be found at <https://goo.gl/ZGHUCe>.

Let's analyze the time complexity of the BFS algorithm. We have to initialize certain properties for each vertex, which is $\mathcal{O}(|V|)$ overall. Then, the while-loop potentially iterates over all $\mathcal{O}(|E|)$ edges, and within the loop, we must iterate over all edges leaving the current vertex. Theoretically, we can imagine a graph in which all $\mathcal{O}(|E|)$ edges in the graph leave the node from which we begin BFS. Thus, at a glance, it may seem as though the overall time complexity for BFS is $\mathcal{O}(|V| + |E|^2)$. However, if you pay closer attention, you will notice that, throughout the entirety of the BFS algorithm, we explore all $\mathcal{O}(|E|)$ exactly once (in the worst case), meaning these $\mathcal{O}(|E|)$ edges are simply spread across the outer-loop and the inner-loop of the pseudocode.

Consequently, because we initialize all $\mathcal{O}(|V|)$ vertices and traverse all $\mathcal{O}(|E|)$ edges exactly once, we can conclude that the tightest worst-case time complexity for Breadth First Search is $\mathcal{O}(|V| + |E|)$

STOP and Think

Say we have a dense graph (i.e., roughly $|V|^2$ edges) with 1,000,000,000 nodes on which we wish to perform BFS. If each edge in the queue could be represented by even just one byte, roughly how much space would the BFS queue require at its peak?

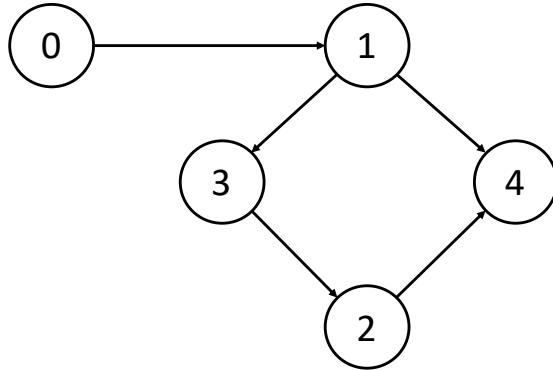
As can be seen, Breadth First Search is an excellent algorithm for traversing a graph, and it can be used to solve many problems in weighted and unweighted graph problems. With regard to the example we started with, it should hopefully be clear that BFS allows us to find the shortest path (i.e., least edges) from one

node to any other node in a graph.

However, hopefully the previous lasting question piqued your curiosity: we mentioned earlier in this chapter that graphs can get quite massive when working with real data, so what happens if the memory requirements of BFS become too great for modern computers to be able to handle memory-wise? Is there an alternative graph traversal algorithm that might be able to save the day?

4.4 Algorithms on Graphs: Depth First Search

Another algorithm to explore all vertices in a graph is called **Depth First Search (DFS)**. The idea behind DFS is to explore all the vertices going down from the current vertex before moving on to the next neighboring vertex. Let's look at the same graph we looked at when we introduced BFS in the last lesson:



Suppose we start at 0. DFS will then choose to “visit” 1 because it is immediately reachable from 0. From 1, DFS can choose to “visit” either 3 or 4 because they are *both* immediately reachable from 1. Suppose DFS chooses to visit 3; after, DFS will “visit” 2 as it is reachable from 3. DFS will finish by visiting 4 because it is immediately reachable from 2. Notice that the DFS algorithm chooses to explore 4 last, as opposed to 2 like in BFS.

It is important to note that DFS could have chosen to explore 4 instead of 3 ($0 \rightarrow 1 \rightarrow 4$). You may notice that, in this case, 4 would not have any more immediately reachable vertex to explore, yet we would still have some unexplored vertices left. As a result, DFS would choose to explore an “unvisited” vertex reachable from the most recent explored vertex (1). In this case, DFS would then choose 3, as it is the next immediately reachable vertex from 1.

The rough algorithm for DFS can be thought of as the following:

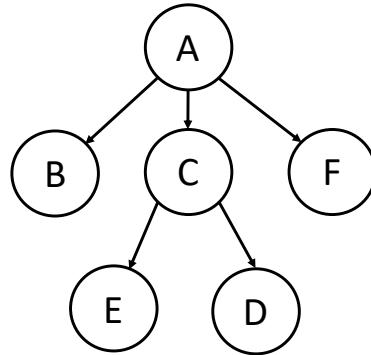
1. Start at s . It has distance 0 from itself.
2. Consider a node adjacent to s . Call it t . It has distance 1. Mark it as visited.

3. Then consider a node adjacent to t that has not yet been visited. It has distance 2. Mark it as visited.
4. Repeat until all nodes reachable from s are visited.

Depth First Search can be difficult to understand without being able to actually see the algorithm in action. As a result, we have created a visualization that shows the order in which each vertex is explored, which can be found at <https://goo.gl/SwCfx9>.

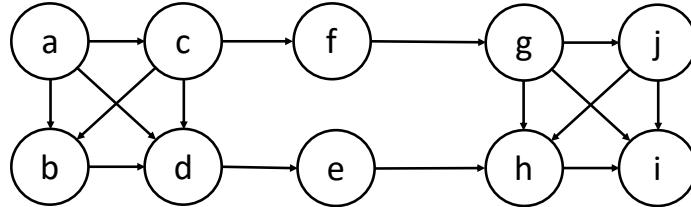
Exercise Break

In what order would Depth First Search explore the vertices in the following graph when starting with vertex A ? Break ties in order-of-exploration by giving priority to the vertex that contains the alphabetically smaller letter.



Exercise Break

In what order would Depth First Search explore the vertices in the following graph when starting with vertex a ? Break ties in order-of-exploration by giving priority to the vertex that contains the alphabetically smaller letter.



Just like how BFS took advantage of the *queue* ADT to explore the vertices in a particular order, Depth First Search takes advantage of the **stack** ADT. The intuition behind choosing a stack for DFS stems from a stack's *first-in last-out* property. With this property, we guarantee that we will explore all

the vertices pushed (put on the stack) most recently first, before moving on to vertices that were adjacent to a vertex from before.

```

1  DFSShortestPath(u,v):
2      s = empty stack
3      add (0,u) to s // (0,u) = (length from u, current vertex)
4      while s is not empty:
5          (length,curr) = s.pop()
6          if curr == v: // found v
7              return length
8          for all edges (curr,w): // explore all neighbors
9              if w has not yet been visited:
10                 add (length+1,w) to q
11      return "FAIL" // no path exists from u to v

```

Notice that the only change in the algorithm from the BFS algorithm we provided earlier is that all mentions of “queue” have been turned into “stack.” Isn’t it crazy how the replacement of one ADT can create an entirely new algorithm? Also, it is cool to note that the nature of Depth First Search’s exploration makes implementing the algorithm recursively absurdly simple. Take a look at this pseudocode:

```

1  DFSRecursion(s): // s is the starting vertex
2      mark s as explored
3      for each unexplored neighbor v of s:
4          DFSRecursion(v)

```

Notice that declaring/using a stack is not needed since the recursive nature of the algorithm provides the stack implementation for us.

STOP and Think

It is important to note that there does not exist such a simple recursive pseudocode for Breath First Search; why might this be the case?

We have created a visualization of how the Depth First Search algorithm takes advantage of the stack ADT to find the shortest path between two nodes, which can be found at <https://goo.gl/Z7RWXb>.

Exercise Break

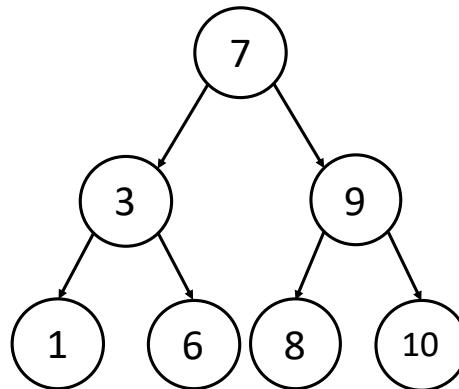
As we have seen by now, DFS seems to behave very similarly to BFS. Consequently, how do their time complexities compare?

Hopefully you should remember that the only difference in the pseudocode for BFS and DFS was their use of a queue and stack, respectively. As a result, your intuition should tell you that we only need to compare the run times for those two ADTs to see if we expect their run times to differ or stay the same.

Assuming optimal implementation, what is the worst-case time complexity of the three operations of a stack: top, pop, and push?

As you hopefully realized, the worst-case time complexities of both a queue and a stack (assuming optimal implementation) are the same. As a result, we can intuitively come to the correct conclusion that both DFS and BFS have a worst-case time complexity of $\mathcal{O}(|V| + |E|)$. Notice, however, that, though their worst-case time complexities are the same, their memory management is totally different! In other words, to traverse the graph, BFS had to store all of a vertex's neighbors in each step in a queue to later be able to traverse those neighbors. On the other hand, if you take a look at the recursive implementation of DFS, DFS only needed to store the previous neighbor to explore the next one. As a result, for certain graph structures, DFS can be more memory efficient.

For example, take a look at the following graph, which also happens to be a Binary Search Tree:



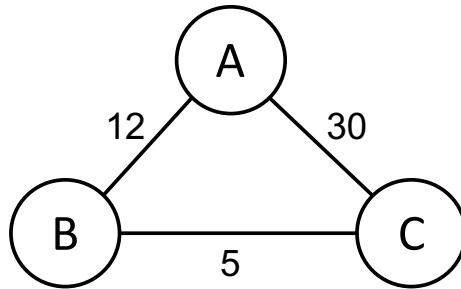
For DFS, there will be a certain time in the algorithm where it will need to store in memory a single strand of vertices going *down* the tree (e.g. vertices 7, 3, and 1 will be stored in memory all at once). For BFS, there will be a certain time in the algorithm where it will need to store in memory a single strand of vertices going *across* the tree (e.g. vertices 1, 6, 8, and 10 will be stored in memory all at once). Even within this simple example, we already see that BFS is beginning to need more space to perform. Now imagine if the tree grows larger: the width of the tree will begin to grow exponentially larger than the height of the tree and as a result, BFS will start to require an exponentially more amount of memory! As a result, in practice, BFS can begin to slow down in performance because of all the extra memory accesses it will need to perform.

Now that we have seen two very solid algorithms that are used to traverse graphs, we will begin to take a look at their potential algorithmic drawbacks in the next section.

4.5 Dijkstra's Algorithm

By now, we have encountered two different algorithms that can find a shortest path between the vertices in a graph. However, you may have noticed we have been operating under the assumption that the graphs being traversed were *unweighted* (i.e., all edge weights were the same). What happens when we try to find the shortest path between two vertices in a *weighted* graph using BFS? Note that we define the *shortest weighted path* as the path that has the smallest *total path weight*, where *total path weight* is simply the sum of all edge weights in the path.

For example, let's perform BFS on the following graph:



Suppose we are trying to find the shortest path from vertex *A* to *C*. By starting at vertex *A*, BFS will immediately traverse the edge with weight 30, thereby discovering vertex *C* and returning the path *A* → *C* to be the shortest path. Yet, if we look closely, we find that, counter-intuitively, the *longer* path in terms of number of edges actually produces the *shortest weighted* path (i.e., *A* → *B* → *C* has a total path weight of $12 + 5 = 17$, which is less than 30).

Consequently, we need to look to a new algorithm to help us find an accurate weighted shortest path: specifically, we will look at **Dijkstra's Algorithm**. The idea behind Dijkstra's Algorithm is that we are constantly looking for the lowest-weight paths and seeing which vertices are discovered along the way. As a result, the algorithm explores neighbors similarly to how the Breadth First Search algorithm would, except that it prioritizes which neighbors it searches for next by calculating which neighbor lies on the path with the lowest overall path weight.

Take the simple example from above. Suppose we start at vertex *C* and want to find the shortest path from *C* to all other vertices. Dijkstra's Algorithm would run roughly like this:

1. We would take note that the distance from vertex *C* to vertex *C* is 0 (by definition, the shortest distance from a node to itself must be 0). As a result, we would mark that we have found the shortest path to vertex *C* (which is just *C*).
2. We would traverse the next “shortest” path (i.e., the path with the small-

est total weight) that we could find from C . In this case, our options are a path of weight 5 ($C \rightarrow B$) or a path of weight 30 ($C \rightarrow A$); we choose the path with the smaller weight of 5 ($C \rightarrow B$).

3. We would take note that we have arrived at vertex B , which implies that the shortest distance from vertex C to vertex B is the total weight of the path we took, which is 5. As a result, we would mark that we have found the shortest path to vertex B (which is $C \rightarrow B$).
4. We would traverse the next “shortest” path that we could find from C . Our options are now the path we forewent earlier ($C \rightarrow A$, with weight 30) and the path extending from B ($C \rightarrow B \rightarrow A$, with weight $5 + 12 = 17$). We choose the path with the smaller weight of 17 ($C \rightarrow B \rightarrow A$).
5. We would take note that we have arrived at vertex A , which implies that the shortest distance from vertex C to vertex A is the total weight of the path we took, which is 17. As a result, we would mark that we have found the shortest path to vertex A (which is $C \rightarrow B \rightarrow A$).
6. We would take note that all vertices have been marked as found, and as a result, all shortest paths have been found!

Up until now, we’ve been saying at a higher level that Dijkstra’s Algorithm needs to just

1. Search for the next shortest path that exists in the graph, and
2. See which vertex it discovers by taking that shortest path.

However, how do we get an algorithm to just “search” for the next shortest path? Unfortunately, a computer can’t just visually scan the graph all at once like we’ve been doing. To solve this problem, Dijkstra’s Algorithm takes advantage of the *Priority Queue* ADT to store the possible paths and uses the total weights of the paths to determine priority.

How do we go about storing the possible paths? The intuitive approach would be to store the paths themselves in the priority queue (e.g. $(A \rightarrow B \rightarrow C)$, $(A \rightarrow B \rightarrow D)$, etc.). However, when the priority queue tries to order the paths, it would need to repeatedly recompute the total path weights to perform its priority comparisons. Thus, to save time, it might seem easiest to store all the paths with their respective total path costs (e.g. $(1, A \rightarrow B \rightarrow C)$, $(3, A \rightarrow B \rightarrow D)$, etc.). However, as our graphs grow larger, this can lead to wasting a lot of space (for example, notice how the path $A \rightarrow B$ was repeated twice in the example above). To combat this waste of space, we instead make the realization that, when following a path, it is enough to know only the *single* immediate vertex that is to be explored at the end of the path at each step. As a result, rather than inserting *entire paths* with their respective costs in the priority queue, we store tuples consisting of (*cost*, *vertex to explore*).

For our purposes, a vertex object needs to contain at least these 3 fields:

- *Distance*: The shortest path that was discovered to get to this vertex
- *Previous*: The vertex right before it, so we can keep track of the path and overall structure of the graph
- *Done*: A boolean field to determine if a shortest path has already been found to this vertex

It is important to note that, since we are storing vertices that are immediately reachable from the current vertex in the priority queue, the same vertex can in fact appear in the priority queue more than once if there exists more than one path to reach it.

Here is a glimpse at how the pseudocode for Dijkstra's Algorithm looks:

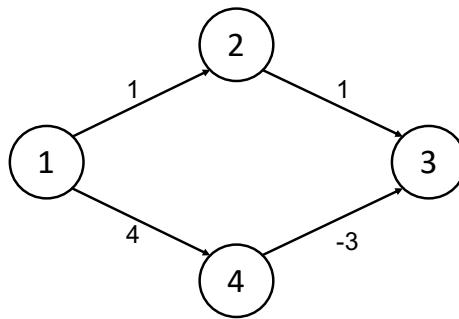
```

1  dijkstra(s):
2      // perform initialization step
3      pq = empty priority queue
4      for each vertex v:
5          v.dist = infinity; v.prev = NULL; v.done = False
6
7      // perform the traversal
8      enqueue (0,s) onto pq
9      while pq is not empty:
10         dequeue (weight,v) from pq
11         if v.done == False: // min path not discovered yet
12             v.done = True
13             for each edge (v,w,d):
14                 c = v.dist + d // total distance to w through v
15                 if c < w.dist: // smaller-weight path found
16                     w.prev = v; w.dist = c
17                     enqueue (c,w) onto pq

```

As you can see above, Dijkstra's Algorithm runs as long as the priority queue is not empty. Moreover, the algorithm never goes back to update a vertex's fields once it is dequeued the first time. This implies that Dijkstra's Algorithm guarantees that, for each vertex, the first time an instance of it is removed from the priority queue, a shortest path to it has been found. Why is this true? All other vertices discovered later in the exploration of Dijkstra's Algorithm must have at least as great a total path cost to them as the vertex that was just dequeued (if it had a smaller total path cost, then that vertex would have been discovered earlier). Therefore, we couldn't possibly achieve a smaller cost pathway by considering the vertices that appear later in the priority queue. This type of algorithmic approach, where we commit to a decision and never look back, is called a "greedy" approach.

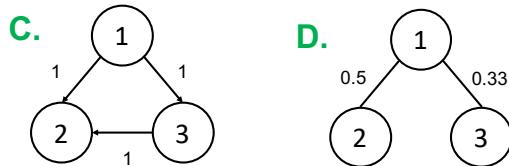
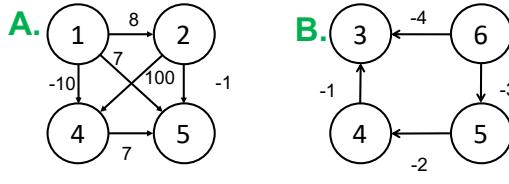
All this time, we have been making one major assumption: no negative weight edges can exist in the graph! If negative weights existed, then the following assumption that all future paths will have a larger weight than the current path is no longer valid. For example, take a look at the following graph:



If Dijkstra's Algorithm starts at vertex 1, it would discover vertex 2 first (since it has a total path cost of 1) and then vertex 3 (since it has a total path cost of $1 + 1 = 2$) and lastly vertex 4 (since it has a total path cost of 4). However, the shortest path from vertex 1 to vertex 3 actually has a total path cost of 1! How? If you take the path $1 \rightarrow 4 \rightarrow 3$, you would end up with a path cost of $4 + (-3) = 1$.

Exercise Break

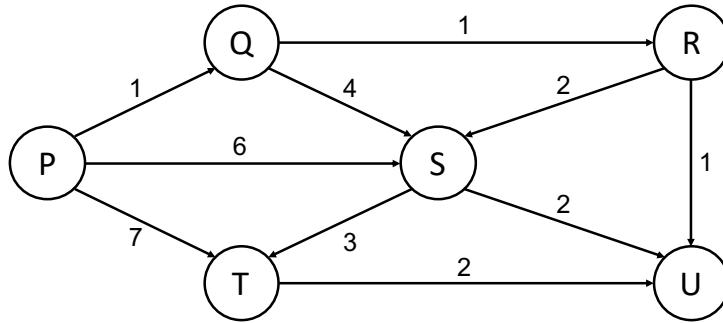
On which of the following graphs will Dijksta's Algorithm successfully find the shortest path starting from any vertex?



We have created a visualization of how Dijkstra's Algorithm takes advantage of the priority queue ADT to find the shortest paths in a graph, which can be found at <https://goo.gl/rzb9nq>.

Exercise Break

In what order would Dijkstra's Algorithm label the vertices in the following graph as “done” when starting at vertex P ?



As we've seen, Dijkstra's Algorithm is able to solve the shortest path problem in weighted graphs, something that BFS can't always do. However, what cost do we face when running this more capable algorithm? In other words, what is the worst-case time complexity of Djikstra's Algorithm? Let's look at the major steps in the algorithm to figure this out:

- Initializing all the fields in a vertex (distance, previous, done): there are $|V|$ such elements, so we have a worst-case time complexity of $\mathcal{O}(|V|)$ here
- Inserting and deleting elements from the Priority Queue:
 - Each edge of the graph (suppose it is implemented as an adjacency list) can cause a single insertion and deletion from the priority queue, so we will be doing at worst $\mathcal{O}(|E|)$ priority queue insertions and deletions
 - The insertion and deletion of a well-implemented priority queue with N elements is $\mathcal{O}(\log N)$. In Dijkstra's algorithm, our “ N ” is simply $|E|$ in the worst case (because we will have added all $|E|$ potential elements into the priority queue), and therefore, for each element we insert and delete from the priority queue, we have a worst-case time complexity of $\mathcal{O}(\log|E|)$. Thus, for all of the $|E|$ elements, we have a worst-case time complexity of $\mathcal{O}(|E|\log|E|)$

Taking into account all of the time complexities, the algorithm has an overall worst-case time complexity of $\mathcal{O}(|V| + |E|\log|E|)$.

Exercise Break

Which of the following are valid ways to modify an unweighted graph to have Dijkstra's Algorithm be able to accurately produce the shortest paths from one vertex to all other vertices?

- We can assign a cost of 1 to each edge.
- We can assign any cost to any edge and Dijkstra's will inevitably produce the shortest path.

- No valid modifications can be made; this exercise break is a trick question.
- No modifications need to be made; the algorithm already takes into account unweighted edges.
- We can assign a cost of -1 to each edge.
- We can assign a cost of $\frac{1}{2}$ to each edge.

4.6 Minimum Spanning Trees

Imagine you are the engineer responsible for setting up the intranet of UC San Diego: given a set of locations V and a set of costs E associated with connecting pairs of locations with cable, your task is to determine the networking layout that minimizes the overall cost of a network where there exists a path *from* each location *to* each location. If your design has even a single pair of locations that cannot reach one another, this can be problematic in the future because individuals at either location will not be able to send each other files. If your design does not minimize the overall cost of such a network, you will have wasted valuable money that could have been spent elsewhere. As such, these two requirements are quite strict.

Notice that we can trivially transform the Network Design Problem above into a formal computational problem that looks quite similar to other problems we solved previously in this chapter. We can represent UC San Diego as a **graph** in which the locations are represented as **nodes** (or **vertices**) and the pairwise location connection possibilities are represented as **undirected weighted edges**, where the weight associated with an edge is simply the cost of the cables needed to connect the two locations connected by the edge.

With this graph representation, the Network Design Problem transforms into the formal computational problem: given a graph G , defined by a set of vertices V and a set of edges E , return a collection of edges such that the following two constraints are maintained:

- For each pair of vertices in the graph, there exists a path, constructed only from edges in the returned collection, that connects the two vertices in the pair
- The sum of the weights of the edges in the returned collection is minimized

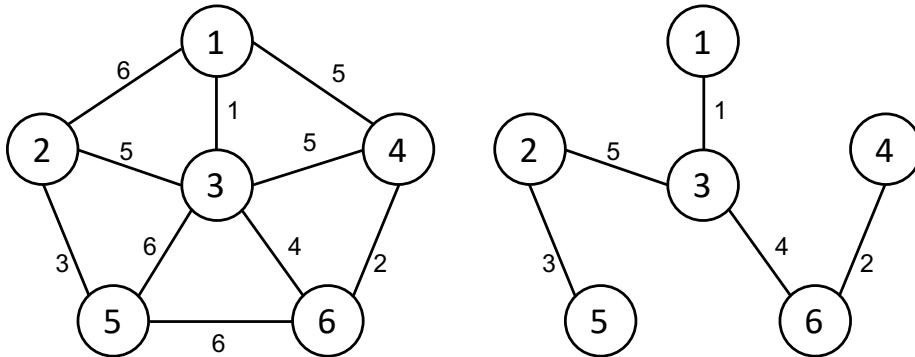
Notice that, since it would be redundant to have multiple paths connecting a single pair of vertices (if the paths are equal weight, you don't need to keep them all; if not, you would only want to keep the shortest-weight path), the subgraph that we return must not contain any cycles. In other words, our subgraph will be a *tree*. Specifically, we call a tree that hits all nodes in a graph G as a **Spanning Tree** of G . Because we want the Spanning Tree with the smallest overall cost, we want to find a **Minimum Spanning Tree (MST)** of G .

In this section, we will discuss two algorithms for finding Minimum Spanning Trees: *Prim's Algorithm* and *Kruskal's Algorithm*.

A *spanning tree* is a tree (what a surprise) that *spans* across the entire graph (i.e., for any pair of nodes u and v in the graph G , there exists some path along the edges in the MST that connects u and v). More formally, a spanning tree for an undirected graph G is an undirected graph that

- contains all the vertices of G ,
- contains a subset of the edges of G ,
- has no cycles, and
- is connected (this implies that only connected graphs can have spanning trees).

The following is an example of a graph with its spanning tree:



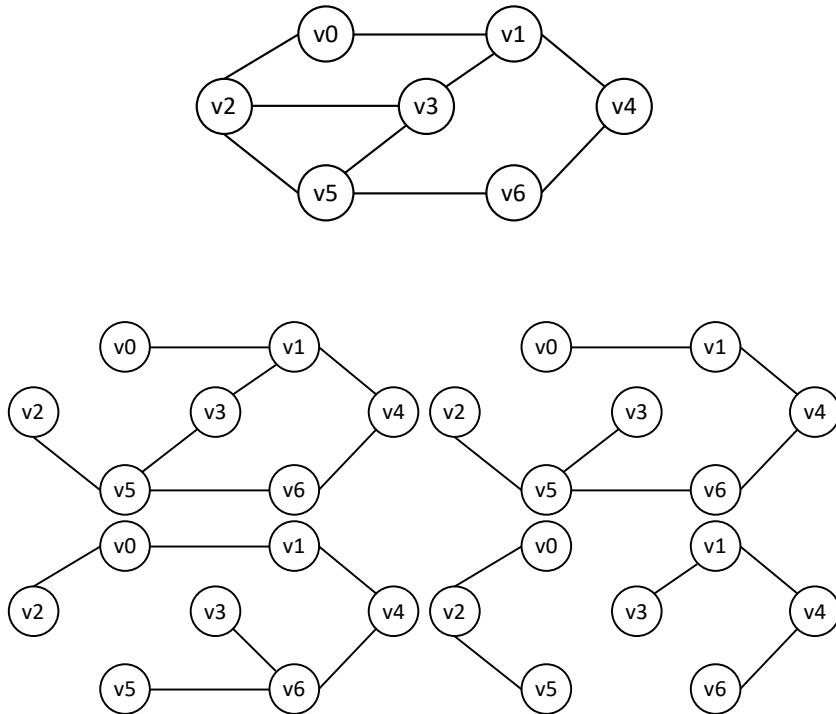
Note that a spanning tree with N vertices will always have $N - 1$ edges, just like any other tree!

STOP and Think

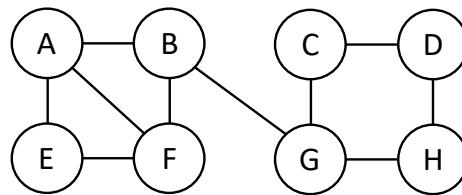
Can a graph have more than one spanning tree that fulfills the four spanning tree properties mentioned above?

Exercise Break

Which of the following are valid spanning trees of the following graph?

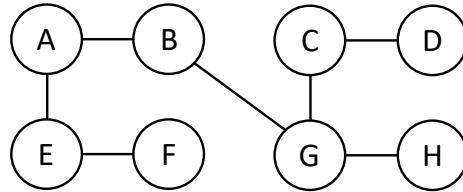


How do we go about algorithmically finding spanning trees? This problem is actually very easy! All we need to do is run a single-source shortest-path algorithm, such as Breadth First Search. We can pick any vertex to run the shortest-path algorithm on. Once the algorithm is finished executing, all the “previous” fields of the vertex would lead us through a path that would create a spanning tree. For example, take the following graph:



Arbitrarily starting at vertex E and breaking ties by giving priority to the smallest letter of the alphabet, BFS would uncover the vertices in the following order: E (previous NULL) $\rightarrow A$ (previous E) $\rightarrow F$ (previous E) $\rightarrow B$ (previous A) $\rightarrow G$ (previous B) $\rightarrow C$ (previous G) $\rightarrow H$ (previous G) $\rightarrow D$ (previous C).

Following the discovered pathway above, we would thus get an assembled spanning tree that looks like this:



So why is BFS able to produce a valid spanning tree? The answer lies behind the idea that BFS is in charge of finding a single shortest path from one vertex to all other vertices and will therefore produce no cycles in its output—otherwise, there would be more than one path to get to a vertex—and thus a tree that covers all vertices.

It is important to note that, in weighted graphs, we can also run BFS to find an *arbitrary* spanning tree. However, in weighted graphs, we often care about actually factoring the weights of the edges (cost of the paths). For example, in our original Networking Problem, an *arbitrary* spanning tree would translate into any network that successfully connects all of the computers at UCSD, but a *minimum* spanning tree will specifically translate into a valid network that has the *lowest cost*. Consequently, it can be useful to find a **Minimum Spanning Tree (MST)** for a graph: “the least-cost version” of the graph that is still connected. Formally, an MST is a spanning tree that minimizes the total sum of edge weights.

Now that we are factoring in *minimum total weight* in our algorithm, we should remember that BFS didn’t always work well in these cases, which is why we originally introduced Dijkstra’s Algorithm. In this case, we will also use a concept from Dijkstra’s Algorithm to achieve what BFS couldn’t. We will introduce two different algorithms that can efficiently produce Minimum Spanning Trees: *Prim’s* and *Kruskal’s*.

In general,

- **Prim’s Algorithm** starts with a single vertex from the original graph and builds the MST by iteratively adding the least costly edges that stem from it (very similar to Dijkstra’s Algorithm)
- **Kruskal’s Algorithm** starts with a forest of vertices without any edges and builds the MST by iteratively adding the least costly edges from the entire graph

As mentioned, **Prim’s Algorithm** starts with a specified starting vertex. While the growing Minimum Spanning Tree does not connect *all* vertices, Prim’s Algorithm adds the edge that has the least cost from any vertex in the spanning tree that has been built so far. More formally,

1. Let V be the set of all vertices, S be the empty set, and A be the empty set. Choose any vertex r to be the root of our spanning tree; set $S = \{r\}$ and $V = V - \{r\}$ (i.e., remove r from set V).

2. Find the least weight edge, e , such that one endpoint, v_1 , is in S and another, v_2 , is in $V \setminus S$ (i.e., an edge that stems from the spanning tree being built so far, but doesn't create any cycles).
 - Set $A = A + \{e\}$, $S = S + \{v_2\}$, and $V = V - \{v_2\}$ (i.e., Add that edge to A , add its other vertex endpoint to S , and remove that endpoint from the set of all “untouched” vertices, V).
3. If $V \setminus S = \emptyset$ (i.e., once S contains all vertices), then output (S, A) as the two sets of vertices and edges that define our MST. Otherwise, our MST clearly doesn't cover all vertices and so we must repeat step 2 above.

We have created a visualization of how Prim's Algorithm chooses which vertices and edges to use to build the Minimum Spanning Tree, which can be found at <https://goo.gl/C9Q8UH>.

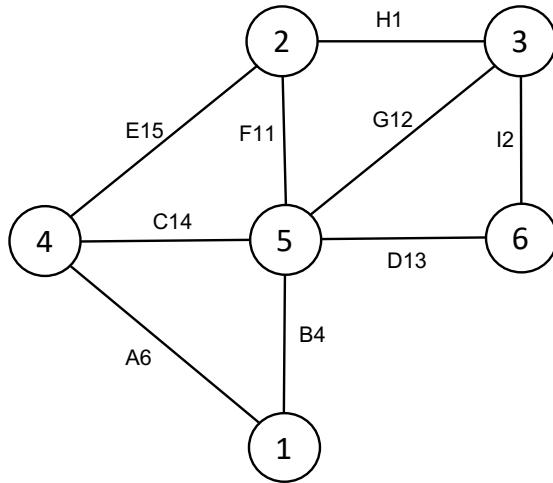
Implementation-wise, Prim's Algorithm achieves the steps described previously by using a priority queue of edges sorted by edge weights. However, unlike Dijkstra's Algorithm, which had used the *total* path edge weights explored so far to the priority queue for comparison, Prim's Algorithm uses *single* edge weights. Here is the pseudocode for Prim's Algorithm:

1. Initialize all the vertex fields of the graph—just like Dijkstra's Algorithm: Set all “done” fields to false. Pick a start vertex r . Set its “prev” field to -1 and its “done” field to true. Add all the edges adjacent to r as a tuple that includes the starting vertex v , ending vertex w , and cost c , into the priority queue.
2. Is the priority queue empty? Done!
3. Else: Dequeue the top element (v, w, c) from the priority queue (i.e., remove the smallest-cost edge).
4. Is the “done” field of vertex w marked true? If so, then this edge connects two vertices already connected in the MST and we therefore cannot use it. Go to step 2.
5. Else: Mark the “done” field of vertex w true, and set the “prev” field of w to indicate v .
6. Add all the edges adjacent to w into the priority queue.
7. Go to step 2.

We have created a visualization of how Prim's Algorithm takes advantage of the priority queue ADT to choose which vertices and edges to use to build the Minimum Spanning Tree, which can be found at <https://goo.gl/XaoM4e>.

Exercise Break

For the following graph, what is the order of the edges that an MST would be built by Prim's Algorithm, starting at Vertex 1? **Note:** Each edge is labeled with its name, followed by its weight (i.e., A6 means edge A has a weight of 6).



Kruskal's Algorithm is very similar to Prim's Algorithm. However, the major difference is that Kruskal's Algorithm starts with a “forest of nodes” (basically the original graph except with no edges) and iteratively adds back the edges based on which edges have the lowest weight. More formally, for a weighted undirected graph $G = (V, E)$:

1. Let E be the set of all edges and A be the empty set.
2. Find the least weight edge, e .
 - If $A = A + \{e\}$ does not create a cycle; set $E = E - \{e\}$ and $A = A + \{e\}$. In other words, if adding e to A does not create a cycle, add e to A (i.e., add the edge to the growing MST) and remove e from E .
 - Else set $E = E - \{e\}$ (this edge creates a cycle and we should just ignore it)
3. If $E = \emptyset$ (i.e., once we have checked all the edges), then output (V, A) as the two sets of vertices and edges that define our MST. Otherwise, we must repeat step 2 above.

STOP and Think

In Prim's Algorithm, we say that an edge e would create a cycle in our growing MST if both vertices of e were already in our growing MST. However, in

Kruskal's Algorithm, even if both vertices of an edge e are already in our work-in-progress MST, e won't necessarily create a cycle. Why is this?

We have created a visualization of how Kruskal's Algorithm chooses which edges to use to build the Minimum Spanning Tree, which can be found at <https://goo.gl/keBuj9>.

STOP and Think

Did Kruskal's Algorithm output the same MST as Prim's Algorithm?

Implementation-wise, here is the pseudocode for Kruskal's Algorithm on a weighted undirected graph $G = (V, E)$:

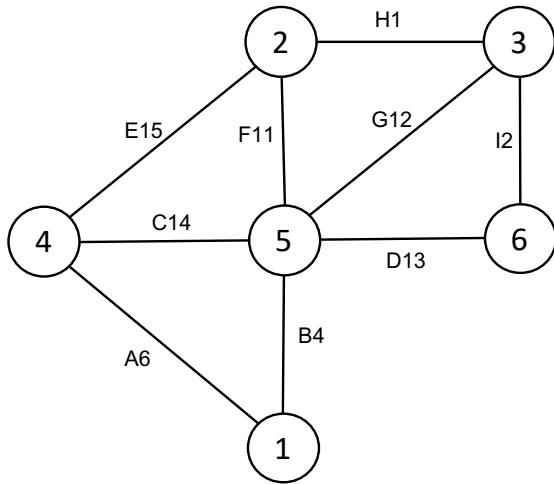
1. Create a forest of vertices
2. Create a priority queue containing all the edges, ordered by edge weight
3. While fewer than $|V| - 1$ edges have been added back to the forest:
 - (a) Deque the smallest-weight edge (v, w, c) from the priority queue
 - (b) If v and w already belong to the same tree in the forest: go to 3a (adding this edge would create a cycle)
 - (c) Else: Join those vertices with that edge and continue

We have created a visualization of how Kruskal's Algorithm takes advantage of the priority queue to choose which vertices and edges to use to build the Minimum Spanning Tree, which can be found at <https://goo.gl/bQAc3d>.

It turns out that, if Kruskal's Algorithm is implemented efficiently, it has a worst-case Big-O time complexity that is the same as Prim's Algorithm: $\mathcal{O}(|V| + |E|\log|E|)$.

Exercise Break

For the following graph, what is the order of the edges that an MST would be built by Kruskal's Algorithm, starting at Vertex 1? **Note:** Each edge is labeled with its name, followed by its weight (i.e., A6 means edge A has a weight of 6).



Exercise Break

In a dense graph, which MST algorithm do we expect to perform better when implemented efficiently?

Exercise Break

True or False: Just like in Dijkstra's Algorithm, both Prim's Algorithm and Kruskal's Algorithm require the edges to be non-negative.

We started this discussion by trying to come up with a good way to design the cabling of UC San Diego's intranet, and we were able to transform the original problem into the task of finding a Minimum Spanning Tree in an undirected weighted graph. Then, throughout this section, we explored two algorithms, Kruskal's Algorithm and Prim's Algorithm, both of which solve this computational problem efficiently in $\mathcal{O}(|V| + |E| \log |E|)$ time. We saw that Prim's Algorithm was implemented very similarly to Dijkstra's Algorithm, yet it didn't have all of the same restrictions (e.g. Kruskal's Algorithm and Prim's Algorithm can easily work with negative edges).

STOP and Think

For a graph with all unique edge weights, both Kruskal's and Prim's Algorithm will always return the same exact MST. Why might this be the case?

4.7 Disjoint Sets

In the hit TV show *Chuck*, Sarah Walker is a secret agent working for the CIA, and she is assigned the task of spying on computer geek Chuck Bartowski.

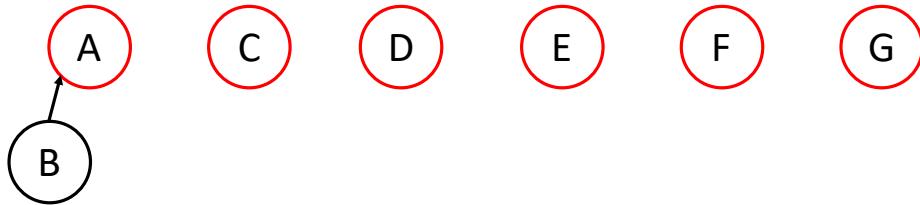
She wants to make sure Chuck isn't secretly working for some evil agency, so specifically, her task is to find out if there is some "connection" between Chuck and a known villain, either direct or indirect. To do so, she must look at Chuck's own social circle, and then look at the social circles of each individual in Chuck's social circle, and then look at the social circles of each individual in *those* people's social circles, etc. As one might expect, this task blows up in size quite quickly, so Sarah would be better off writing an efficient program to solve the problem for her. How might she formulate this CIA task as a formal computational problem?

We can represent Chuck's social network (and the social networks of those around him) using a *graph*: nodes (V) are individuals, and edges (E) represent individuals having a direct social connection to one another. Sarah's goal is to come up with a way of being able to query a person's name and determine if the person is connected to Chuck, either directly or indirectly. If she were to approach this as a naive graph traversal algorithm, she would need to use BFS each time she performs a query, which is $\mathcal{O}(|V| + |E|)$. Luckily, Sarah did well in her university Data Structures course and, as a result, she had the Computer Science skills needed to solve the seemingly complex problem quite efficiently.

In this section, we will discuss the **Disjoint Set** ADT, which will help Sarah greatly speed up her surveillance problem. The Disjoint Set is an Abstract Data Type that is optimized to support two particular operations:

- **Union:** Given two elements u and v , merge the sets to which they belong
- **Find:** Given an element e , return the set to which it belongs

As a result, a Disjoint Set is also commonly referred to as a "Union-Find" data type. Disjoint Sets can be very efficiently implemented using the **Up-Tree** data structure, which in essence is simply a graph in which all nodes have a single "parent" pointer. Nodes that do not have a parent (i.e., the roots of their respective trees) are called **sentinel nodes**, and each sentinel node represents a single set. In the following example, there are 6 sets, each represented by its respective sentinel node (shown in red): A , C , D , E , F , and G .



A disjoint set often starts out as a forest of single-node sets. We begin connecting vertices together by calling the union operation on two vertices at a time, such as what you see in the figure above. By doing so, we slowly begin creating a structure that looks like an upside-down tree (i.e., a tree where all pointers point up *to the root* instead of the usual down *from the root*), hence

the name “Up-Tree.” We will explore the union operation in more depth later in this section.

The algorithm behind the “find” operation of a Disjoint Set implemented as an Up-Tree should hopefully be intuitive. Recall that the sentinel nodes represent (or really name) the sets. As a result, given an element, finding the set to which the element belongs is actually the exact same problem as finding the sentinel node that is the ancestor of the element. As a result, to find an element’s set, you can simply start at the element and traverse “parent” pointers up the tree until you reach the element’s sentinel node (i.e., the ancestor that has no parent). Then, simply return the sentinel node, because the sentinel node “is” the element’s set.

Because the “find” algorithm traverses up the tree of the given element, it should hopefully be intuitive that the larger the height of the tree an element is in, the slower the “find” operation is on that element. As a result, our goal should always be to try to minimize our tree heights as much as possible, as this will translate into faster “find” operations.

In the meantime, how do we go about implementing this “forest of nodes”? One option would be to implement it as an adjacency list or an adjacency matrix, just like we went about implementing the graphs in the previous lessons. Note, however, that nodes in an Up-Tree will always only need to store an edge to a single other node, its parent, as opposed to multiple adjacent vertices. As a result, we can take advantage of a simple array to store the information that we need.

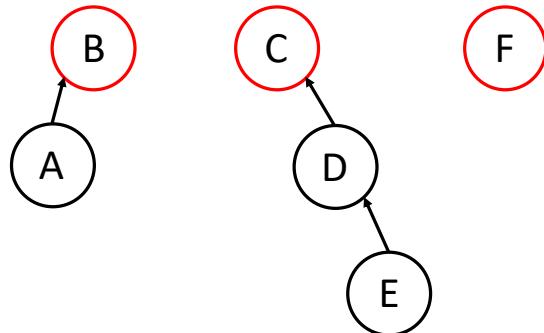
The following is an array that represents the Up-Tree in the previous example:

A	B	C	D	E	F	G
-1	0	-1	-1	-1	-1	-1

Every index of the array corresponds to a particular vertex (e.g. index 0 corresponds to vertex A, index 3 corresponds to vertex D, etc.). The content of each cell of the array corresponds to the corresponding vertex’s parent. For example, the index corresponding to vertex B—index 1—has a 0 in it because index 0 corresponds to vertex A and vertex A is the parent of vertex B. Also, the index corresponding to vertex F—index 5—has a -1 in it because an index of -1 corresponds to **NULL**, and since vertex F doesn’t have a parent, its parent is **NULL**.

Exercise Break

Fill in the missing elements in the array for the corresponding following Up-Tree.

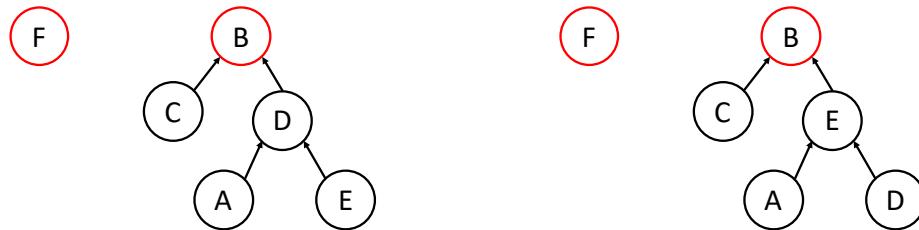


A	B	C	D	E	F
?	?	-1	?	3	-1

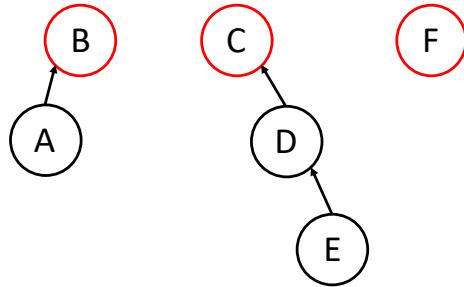
Exercise Break

Select the corresponding disjoint set for the following array:

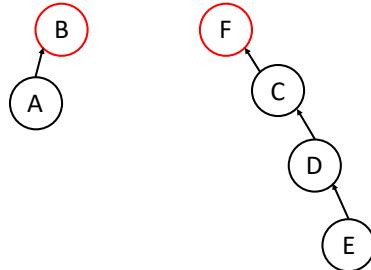
A	B	C	D	E	F
5	-1	4	4	5	-1



We mentioned earlier that elements in the disjoint set are connected through the **union** operation (we care about connecting elements in our set in order to be able to show a relationship between the elements, just like in a general graph). How exactly does the union operation work? Formally, the purpose of the union operation is to join two sets. This implies that, after we have unioned two different sets, the two sets should end up with the **same** sentinel node. Consequently, a lot of freedom actually surrounds the concept behind the union operation because there are so many ways to merge two sets of vertices. For example, suppose we start with the following disjoint set:



Let's say we want to union(F, E) (i.e., merge the set vertex F is in with the set vertex E is in). First, we must find the set that contains vertex F by calling "find" on vertex F . This would return vertex F since the sentinel node of vertex F is vertex F . Second, we must find the set that contains vertex E by calling "find" on vertex E . This would return vertex C since the sentinel node of vertex E is vertex C . Lastly, we must find a way to merge both sets. An intuitive way might be to union the sets by connecting vertex C to vertex F like so:

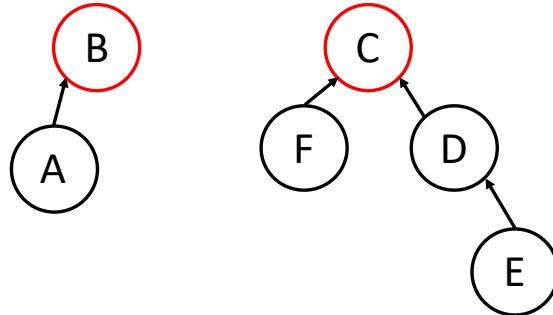


In the example above, we basically just stacked the vertices and literally connected the sentinel node of vertex E (vertex C) to the sentinel node of vertex F (vertex F). Thus, vertex F becomes the parent and vertex C becomes the child. Notice that we specifically connected the sentinel nodes of the two elements on which we called union, not necessarily the elements themselves! This is extremely important because, otherwise, we might be disrupting the structure of our Up-Tree. Now, the important question to ask is: Do vertex F and vertex E end up returning the same sentinel node? The answer in this case

is obviously yes; if we traverse up from both vertex E and vertex F , the root is vertex F .

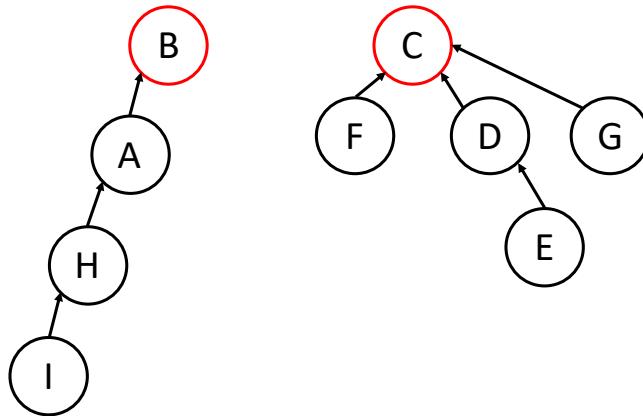
In the previous example, we arbitrarily chose which sentinel nodes we wanted to use as the parent and child when merging. However, as mentioned before, our overarching goal while building an Up-Tree is to minimize its height because a smaller tree height corresponds to a faster “find” operation. Consequently, instead of making the choice arbitrarily, we can perform a **Union-by-Size**. In this insertion tactic, the sentinel node of the *smaller* set (i.e., the set with less elements) gets attached to the sentinel node of the *larger* set. Thus, the sentinel node of the smaller set becomes the child and the sentinel node of the larger set becomes the parent. Ties can be broken arbitrarily.

The following is the result of performing the exact same operation, $\text{union}(F, E)$, by using the Union-by-Size approach. Note that, in the original disjoint set above, the sentinel node of F (node F) has 1 element in its set (node F), and the sentinel node of E (node C) has 3 elements in its set (node C , node D , and node E). As a result, the sentinel node of the *larger* set (node C) is made the *parent*, and the sentinel node of the *smaller* set (F) is made the *child*.

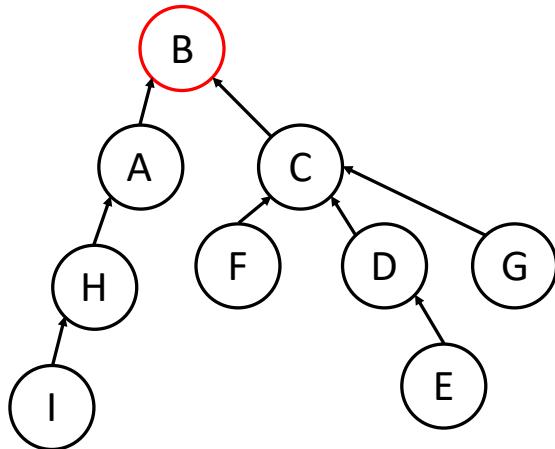


Under Union-by-Size, the only way for the height of a tree resulting from the union of two sets to be *larger* than the heights of both of the two initial sets is if the sets have exactly the same height. As a result, the “worst case” for Union-by-Size is considered to be the situation in which each merge had to merge sets with equal size. For example, say we have a dataset of n elements, each in their own set initially. To invoke the worst case, we can perform $\frac{n}{2}$ union operations to form $\frac{n}{2}$ sets, each with exactly 2 elements. Then we can perform $\frac{n}{4}$ union operations to form $\frac{n}{4}$ sets, each with exactly 4 elements. If we keep doing this until we are left with one set with n elements, that one set will have the shape of a balanced binary tree. As a result, the worst-case time complexity of a “find” operation in Union-by-Size is the same as the worst-case height of a balanced binary tree: $\mathcal{O}(\log n)$.

Along side the Union-by-Size method, there is also a **Union-by-Height** method. In the Union-by-Height method, the sentinel node of the *shorter* set (i.e., smaller height) gets attached to the sentinel node of the *taller* set. Again, ties can be broken arbitrarily. Let’s say we start with the following disjoint set:



The following is the result of performing $\text{union}(A, C)$ by using the Union-by-Height approach. The sentinel node of node A (node B) has a height of 3 and the sentinel node of node C (node C) has a height of 2. As a result, the sentinel node of the *taller* set (node B) is made the *parent* and the sentinel node of the *shorter* set (node C) is made the *child*.



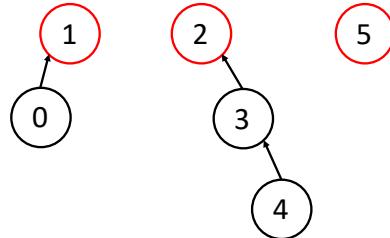
Just like Union-by-Size, the Union-by-Height method results in a worst-case time complexity of $\mathcal{O}(\log n)$ for the “find” operation.

STOP and Think

If we used Union-by-Size instead of Union-by-Height on the example above, would the resulting tree be better, worse, or just as good as the one produced by the Union-by-Height method?

Exercise Break

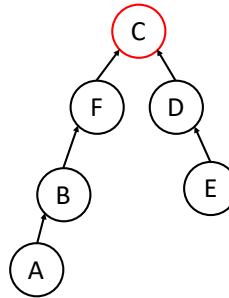
Fill in the array that would result in performing union(0,3) using the Union-by-Height method on the following disjoint set.



0	1	2	3	4	5
?	?	?	?	?	?

As we've seen, choosing the “smarter” union operations (Union-by-Height or Union-by-Size) can lead to big improvements when it comes to finding particular vertices. Furthermore, we can actually do *even better* by making a slight optimization to the “find” operation: adding **path compression**. As mentioned previously, the original implementation of the “find” operation involves starting at the queried node, traversing its tree all the way up to the sentinel node, and then returning the sentinel node. This algorithm, assuming we used Union-by-Height or Union-by-Size to construct our Up-Tree, has a worst-case time complexity of $\mathcal{O}(\log n)$. If we want to use the “find” operation again to search for another vertex's sentinel node, the operation will again take $\mathcal{O}(\log n)$ time (as expected).

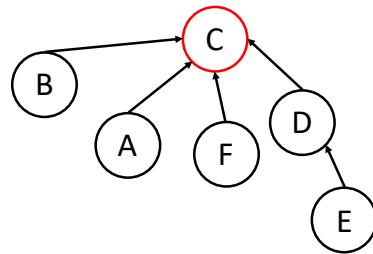
So how can we go about improving the time complexity of the “find” operation even more? The answer lies in taking advantage of all of the information we obtain during the “find” algorithm: not just the sentinel node of interest. Take a look at the example following disjoint set:



Suppose we want to perform $\text{find}(A)$. We know that the goal of “find” is to return the sentinel node of vertex A (which is vertex C in this case). Notice, however, that every vertex on our exploration path (vertex A , vertex B , and

vertex F) have vertex C as their sentinel nodes, not just vertex A ! As a result, it is unfortunate that, in the process of finding vertex A , we have gained the information that vertex B and vertex F also share the same root (which would save us time later if we wanted to find(B) or find(F)), yet we have no way of easily and efficiently memorizing this information... Or do we?

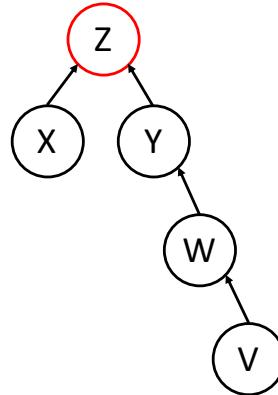
It turns out that we actually do! The solution is as follows: As we traverse the tree to return the sentinel node, we re-attach each vertex along the way directly to the root. This *extremely* simple, yet quite powerful, technique is called **path compression**. For example, using path compression on the disjoint set above, calling find(A) would yield the following disjoint set once the operation finishes:



Now, if I call find(B) on the new disjoint set above, this operation will be constant time! This is because we guarantee that since the vertex is attached directly to the root, we will only need one operation to return the root. You are now allowed to be mind-blown.

Exercise Break

Which vertices will be connected directly to the sentinel node (vertex Z) as a result of performing find(V) using path compression on the following disjoint set?



An Up-Tree with an implementation of path compression is an example of a *self-adjusting structure*. In a self-adjusting structure, an operation like the “find” operation occasionally incurs a high cost because it does extra work to modify the structure of the data (such as the reattachment of vertices directly to the root in the example of path compression we saw previously). However, the hope is that subsequent similar operations will be made *much* more efficient. Examples of other self-adjusting structures are splay trees, self-adjusting lists, skew heaps, etc. (just in case you’re curious to see what else is out there).

So how do we figure out if the strategy of self-adjustment is actually worth it in the long run? The answer lies in **amortized cost analysis**. In regular algorithmic analysis, we usually look at the time or space cost of doing a single operation in either the best case, average case, or worst case. However, if we were to do so for an operation that involved something like path compression, we would not get the tightest complexity because we already know that, based on how path compression was designed, the first time we use it is essentially guaranteed to take longer than subsequent times. Amortized cost analysis considers the time or space cost of doing a *sequence of operations* (as opposed to a single operation) because the total cost of the entire sequence of operations might be less with the extra initial work than without!

We will not go into the details of how to actually perform amortized cost analysis because the math can get quite intense. Nonetheless, a mathematical function that is common in amortized cost analysis is $\log^*(N)$ (read: “log star of N ” and also known as the “single variable inverse Ackerman function”), which is equal to the number of times you can take the log base-2 of N until you get a number less than or equal to 1. For example,

- $\log^*(2) = \log^*(2^1) = 1$
- $\log^*(4) = \log^*(2^2) = 2$ (note that $4 = 2^2$)
- $\log^*(16) = \log^*(2^4) = 3$ (note that $16 = 2^{2^2}$)
- $\log^*(65536) = \log^*(2^{16}) = 4$ (note that $65536 = 2^{2^{2^2}}$)
- $\log^*(2^{65536}) = 5$ (note that $2^{65536} = 2^{2^{2^2}}$ is a huge number)

The reason we use this particular function is because it is able to take into consideration both aspects of a self-adjusting operation:

- The first non-constant time part (log base-2 in this case) in which the structure is re-adjusting itself
- The second constant time part (less than or equal to 1 in this case) in which the structure has finished re-adjusting and is now reaping the benefits of the self-adjustment

It can be shown that, with **Union-by-Size** or **Union-by-Height**, using **path compression** when implementing “find” performs any combination of up

to $N - 1$ “union” operations and M “find” operations and therefore yields a worst-case time complexity of $\mathcal{O}(N + M \log^*(N))$ over all operations. This is much better than old-fashioned logarithmic time because $\log^*(N)$ grows much slower than $\log(N)$, and for all practical purposes, $\log^*(N)$ is never more than 5 (so it can be considered practically constant time for reasonable values of N). Therefore, for each individual find or union operation, the worst-case time complexity becomes almost constant (i.e., almost $\mathcal{O}(1)$).

We have now introduced the Disjoint Set ADT, which allows us to create sets of elements and to very efficiently perform union and find operations on these sets. You may have noticed (hopefully triggered by the **STOP and Think** question) that Union-by-Height always produces an Up-Tree that is always either just as good, or many times even better, than the tree produced by Union-by-Size. However, it turns out that, in practice, people typically choose to use Union-by-Size when they implement their Up-Trees. The reason why they choose Union-by-Size over Union-by-Height is that there is a *huge* savings when it comes to path compression; one side effect of path compression is that the height of the tree and the subtrees within it change very frequently. As a result, maintaining accurate information about the *heights* of sets (which is required for Union-by-Height) becomes quite difficult because of path compression, whereas maintaining accurate information about the *sizes* of sets (which is required for Union-by-Size) is extremely easy (constant-time, actually). As a result, though Union-by-Size may get *slightly* worse results than Union-by-Height, the difference is so small in comparison to the savings from path compression that it is considered negligible.

Going back to the example that we had initially introduced in the lesson, by using a Disjoint Set implemented using an Up-Tree, Sarah can very efficiently solve her problem! In her case, elements of the Disjoint Set would be the people she notices, and as she sees individuals u and v “connect,” she can simply perform $\text{union}(u, v)$. When she wants to check if some suspect w is connected in some way to Chuck, she can perform $\text{find}(\text{“Chuck”})$ and $\text{find}(w)$ and compare the sets they return for equality.

Chapter 5

Hashing

5.1 The Unquenched Need for Speed

Throughout this text, three basic operations for storing data have been on our minds: *find*, *insert*, and *remove*. We have discussed various data structures that can be used to implement these three functions, and we have analyzed their time complexities in order to describe their performance:

- In an *unsorted* Array List and a Linked List, the *worst*-case time complexity to find an element is $\mathcal{O}(n)$
- In a Randomized Search Tree and a well-structured Skip List, the *average*-case time complexity to find an element is $\mathcal{O}(\log n)$
- In a *sorted* Array List and a balanced Binary Search Tree, the *worst*-case time complexity to find an element is $\mathcal{O}(\log n)$

$\mathcal{O}(\log n)$ is pretty fast, but as we have said before, us computer scientists can never be satisfied: we want even *faster* data structures. With an array, if we knew the specific index we wanted to access, we could theoretically access our element of interest in $\mathcal{O}(1)$ time. Formally, if we were looking for a key k in an array a **and** if we had a way of knowing that key k would be at index i , we could find k with a single $\mathcal{O}(1)$ array access operation: $a[i]$.

Hashing is a way of making the idea above a reality. Given an element k , hashing would help us figure out where we would expect k to appear in an array. In this chapter, we will discuss good design methods that allow us to achieve an average-case time complexity of $\mathcal{O}(1)$ for finding, inserting, and removing elements. Specifically, the data structure we will discuss is the **Hash Table**.

If you do a simple Google search for “hashing definition computer science,” you may notice that you will get a variety of definitions. For example,

- “*Hashing is the transformation of a string of characters into a usually shorter fixed-length value or key that represents the original string*” (Search-SQLServer),

- “Hashing is one way to enable security during the process of message transmission when the message is intended for a particular recipient only... Hashing is also a method of sorting key values in a database table in an efficient manner” (techopedia),
- “Hashing involves applying a hashing algorithm to a data item, known as the hashing key, to create a hash value” (WikiBooks),
- and, perhaps our favorite, “Hashing is one of the great ideas of computing and every programmer should know something about it” (I Programmer).

Technically, all the definitions above are correct in their own contexts. However, for our purposes, we will stick most closely with the third definition. For our purposes, we will describe hashing as follows:

- Given a key k , use a **hash function** to compute a number (called a **hash value**) that represents k (we will cover hash functions and their design in more detail in the next section of the text). This process of using a hash function to compute a hash value for some key k is called **hashing**
- Then, we can use the hash value of k to compute an index in some array in which we will store k , which is the idea behind a **Hash Table**

There are many design choices that need to be made in order to design a fast Hash Table, so throughout this chapter, we will discuss the following design topics:

- Designing a **good hash function**
- Deciding on the **size of the Hash Table**
- Deciding on the **collision resolution strategy** (i.e., what should be done when multiple keys map to the same location in a Hash Table)

These topics (as well as the notion of what a Hash Table really is) are probably unclear so far, so we will chip away at these topics one-by-one. In the next section, we will discuss the first topic: hash functions.

5.2 Hash Functions

We started our discussion of hashing with the idea that, given some key k , we could come up with a function that could output an integer that “represents” k . Formally, we call this function a **hash function**, which we will denote as $h(k)$ throughout this section.

Hopefully, our use of the word “represents” upset you a bit: what does the word “represent” actually mean mathematically? Formally, a hash function can be described by the following two properties, the first of which being a *requirement* of *all* valid hash functions and the second being a feature that would be “nice to have” if at all possible, but not necessary:

1. **Property of Equality:** Given two keys k and l , if k and l are equal, $h(k)$ **must equal** $h(l)$. In other words, if two keys are equal, they **must** have the same hash value
2. **Property of Inequality:** Given two keys k and l , if k and l are *not* equal, it would be nice if $h(k)$ was not equal to $h(l)$. In other words, if two keys are *not* equal, it would be nice (but not necessary!) for them to have different hash values

A hash function is formally called *perfect* (i.e., a **perfect hash function**) if it is guaranteed to return different hash values for different keys (in other words, if it is guaranteed to have *both* the Property of Equality *and* the Property of Inequality).

For our purposes with regard to Data Structures, a “good” hash function means that different keys will map to different indices in our array, making our lives easier. Formally, we say that a “good” hash function minimizes *collisions* in a Hash Table, even if collisions are still theoretically possible, which simply translates to faster performance. The mechanism by which collisions worsen the performance of a Hash Table, as well as how we deal with them, will be discussed extensively later in this chapter, but for now, trust us.

In this text, we are introducing hash functions with the intent of using them to implement a Hash Table, but there are many real-world applications in which we use “good” hash functions outside the realm of Data Structures! One very practical example is checking the validity of a large file after you download it. Let’s say you want to install a new Operating System on your computer, where the installation file for an Operating System can sometimes reach a few gigabytes in size, or let’s say you want to stream some movie (legally, of course), or perhaps you want to download a video game to play (again, legally, of course). In all of these examples (which are quite different at a glance), the computational task is the same: there is a large file hosted on some remote server, and you want to download it to your computer. However, because network transfer protocols don’t have many provisions for ensuring data integrity, packets can become corrupted, so larger files are more likely to become corrupted at some point in their transfer. As a result, you want some way to check your large file for validity before you try to use it.

Ideally, you might want to do a basic Unix command (e.g. `diff`) to check your downloaded file against the original for equality, but you can’t check for equality unless you actually had the original file locally, which you don’t! That was the whole reason why you were downloading it in the first place! Instead, many people who host these large files will *also* hash the file and post the hash value, which is just an integer (typically represented as a hexadecimal number) that you can easily copy (or just read) without any worries of corruption. You can then download the large file, compute a hash value from it, and just visually compare your file’s hash value against the original one listed online:

- If your hash value is *different*, the file you downloaded does not match the original (because of the Property of Equality, which is a **must** for a hash

function to be valid), so something must have gotten corrupted during the download process

- If your hash value *matches* the one online, even though there is *technically* still a chance that your file is different than the original (because the Property of Inequality is simply a “nice feature,” but not a requirement), the hash functions used by these hashing tools are very good, so you can assume that the chance that your file being corrupted is negligible

If you are curious to learn more about using hashing to check a file for validity, be sure to check out the main hash functions used for this task today: MD5, SHA-1, and CRC (where CRC32 is typically the most popular out of the types of CRC).

Let’s imagine we are designing a hash function for keys that are ASCII characters (e.g. the `unsigned char` primitive data type of C++). The following is a C++ implementation of a hash function that is *valid*, because keys that are of equal value will have the same hash value, and *good* (*perfect*, actually), because keys of unequal value are guaranteed to have different hash values:

```
1 unsigned int hashValue(unsigned char key) {
2     return (unsigned int)key; // cast key to unsigned int
3 }
```

Because each of the C++ primitive data types that are less than or equal to 4 bytes in size (`bool`, `char`, `unsigned char`, `int`, `unsigned int`, etc.) are stored in memory in a way that can be interpreted as an `unsigned int`, simply casting them as an `unsigned int` would result in a perfect hashing. Also, because primitive data types in most, if not all, languages can be cast to an integer in $\mathcal{O}(1)$ time, this perfect hash function has a worst-case time complexity of $\mathcal{O}(1)$.

Let’s imagine now that we are designing a hash function for keys that are strings. The following is a C++ implementation of a hash function that is *valid*, because keys that are of equal value will have the same hash value, and *decent*, because keys with different values will *tend* to have different hash values:

```
1 unsigned int hashValue(string key) {
2     unsigned int val = 0;
3     for(int i = 0; i < key.length(); i++) {
4         val += (unsigned int)(key[i]);
5     }
6     return val;
7 }
```

This hash function is pretty good because keys with different lengths or characters will probably return different hash values. If we didn’t take into account all of the characters of the string, we would have more collisions than we would want. For example, if we wrote some hash function that only looks at the first character of a string, even though the hash function would be $\mathcal{O}(1)$, it would always return the same hash value for strings that had the same first

character (e.g. “Hello” and “Hashing” have the same first character, so a hash function that only checks the first character of a string *must* return the same hash value for these two very different strings). As a result, for a string (or list, or collection, etc.) of length k , a good hash function must iterate over **all k characters** (or elements).

However, in the function above, even though we iterate over all k characters of a given string, we can still have a lot of collisions. For example, using the hash function above, because we simply add up the ASCII values of each character, the strings “Hello” and “eHlol” will have the same hash value because, even though their letters are jumbled, they contain the same exact letters, so the overall sum will be the same. Formally, if the arithmetic of our hash function is commutative, the order of the letters will not affect the results of the math, potentially resulting in unnecessary collisions.

Therefore, a good hash function for strings of length k (or generally, lists/containers/etc. with k elements) must have a time complexity of $\mathcal{O}(k)$ and must perform arithmetic that is **non-commutative**. If you are interested in learning about clever hash functions for strings, Ozan Yigit at York University wrote an excellent brief article.

Let’s now imagine that we are designing a hash function for keys of some arbitrary datatype **Data** (it doesn’t matter what **Data** is: the point we will make will be clear regardless of datatype). The following is a C++ implementation of a hash function that is *valid*, because keys that are of equal value will have the same hash value, but that is pretty terrible, because we will have a *lot* of collisions:

```

1 unsigned int hashValue(Data key) {
2     return 0;
3 }
```

It should hopefully be obvious why this is such a bad hash function: no matter what *key* is, it will always have a hash value of 0. In other words, the hash value of *key* tells us absolutely nothing useful about *key* and can’t really be used to do anything.

Let’s again imagine that we are designing a hash function for keys of the same arbitrary datatype **Data**. The following is a C++ implementation of a hash function that is *not valid*, because keys that are of equal value will *not necessarily* have the same hash value:

```

1 unsigned int hashValue(Data key) {
2     return (unsigned int)rand(); // return a random integer
3 }
```

Even if two keys are identical, if I compute their hash values separately, because my hash function simply generates a random number each time it’s called, they will almost certainly have different hash values. Recall that, for a hash function to be *valid*, the Property of Equality *must* hold, but the fact that

equal keys can have different hash values violates this property.

Exercise Break

Which of the following hash functions are valid?

```
1 unsigned int hashCode(unsigned int key) {
2     return key % 100;
3 }
```

```
1 unsigned int hashCode(unsigned int key) {
2     return 100;
3 }
```

```
1 unsigned int hashCode(unsigned int key) {
2     return ((13*key - 20) % 23) + 5;
3 }
```

```
1 unsigned int hashCode(unsigned int key) {
2     return rand(); // random integer
3 }
```

```
1 unsigned int hashCode(unsigned int key) {
2     return key % (unsigned int)time(NULL);
3 }
```

```
1 unsigned int hashCode(unsigned int key) {
2     return key;
3 }
```

```
1 unsigned int hashCode(unsigned int key) {
2     return (unsigned int)time(NULL);
3 }
```

```
1 unsigned int hashCode(unsigned int key) {
2     return key % rand();
3 }
```

We've now discussed the properties of a hash function as well as what makes a hash function good or bad. However, it is important to note that, for most things you will do in practice, excellent hash functions will have already been defined; you typically only have to worry about writing your own hash function if you design your own object. Nevertheless, we can now assume that, given a datatype we wish to store, we have some (hopefully good) hash function that can return a hash value for an object of that type. We can now tie this back to our original idea of using hash values in order to determine indices to use to store elements in an array. Say we have an array of length m and we want to insert an object key into the array. We can now perform two hashing steps:

1. Call $h(key)$, where h is the hash function for the type of object key is, and save the result as $hashValue$
2. Perform a second “hashing” by modding $hashValue$ by m to get a valid index in the array (i.e., $index = hashValue \% m$)

By performing these two steps, given an arbitrary object key , we are able to successfully compute a valid index in our array in which we can store key . Again, as we mentioned before, if our hash function is not *perfect* (i.e., if we sometimes hash *different* objects to the *same* hash value), or if two different hash values result in the same value when modded by m , we will run into *collisions*, which we will discuss extensively later in this chapter.

Exercise Break

Say you have an array of length 5. Also, you have a primary hash function that simply returns the integer it was given (i.e., $h(k) = k$), and you have a secondary hash function that computes an index in your array by modding the result of the primary hash function by the length of the array (i.e., $i = H(k) = h(k)\%5$). For each of the following integers, compute the index of your array to which they hash: 12, 17, 22, 27, and 32

Thus far, we have seen examples of hash functions that were good (or even *perfect*, like $h(k)$ in the previous exercise). Nevertheless, picking an unfortunate size for our array or choosing a bad indexing hash function (like $H(k)$ in the previous exercise) can still result in numerous collisions. Also, we saw that, if a given object key has k elements (e.g. a string with k characters, or a list with k numbers), a good hash function for key will incorporate each of key 's k elements into key 's hash value, meaning a good hash function would be $\mathcal{O}(k)$. Also, to avoid collisions that come about when comparing objects containing the same elements but in a different order, we need the arithmetic we perform to be **non-commutative**.

Thus far, we have discussed hashing and the motivation for Hash Tables, and we have defined what a hash function is as well as what makes one good or bad, but we still have yet to actually formally discuss Hash Tables themselves. In the next section, we will finally shed light on this seemingly mystical data structure that achieves $\mathcal{O}(1)$ average-case finds, insertions, and removals.

5.3 Introduction to Hash Tables

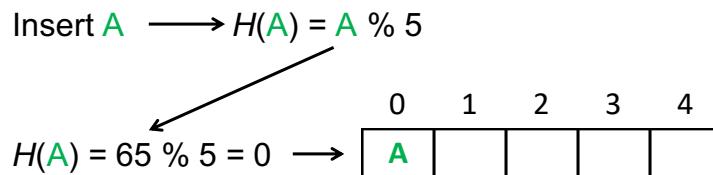
We started this discussion about hashing with the motivation of achieving $\mathcal{O}(1)$ find, insert, and remove operations on average. We then discussed hash functions, which allow us to take any object of some arbitrary type and effectively “convert” it into an integer, which we can then use to index into an array. This hand-wavy description of the ability to index into an array is the basic idea of the Hash Table data structure, which we will now formally discuss.

DISCLAIMER: Whenever we say “ $\mathcal{O}(1)$ ” or “constant time” with regard to the average-case time complexity of a Hash Table, this is **ignoring the time complexity of the hash function**. For primitive data types, hash functions are constant time, but for types that are collections of other types (e.g. lists, strings, etc.), good hash functions iterate over *all* the elements in the collection. For example, a good hash function for strings of length k would iterate over all k characters in the string, making it $\mathcal{O}(k)$. Thus, mapping a string of length k to an index in an array is in reality $\mathcal{O}(k)$ overall: we first perform a $\mathcal{O}(k)$ operation to compute a hash value for the string, and we then perform a $\mathcal{O}(1)$ operation to map this hash value to an index in the array.

Nevertheless, people say that Hash Tables in general have an average-case time complexity of $\mathcal{O}(1)$ because, unlike any of the other generic data structures (i.e., data structures that can store *any* datatype) we have seen thus far, the average-case performance of a Hash Table is *independent* of the number of elements it stores. For this reason, from here on out, we will omit the time complexity of computing a hash value when describing the time complexity of a Hash Table, which is the norm in the realm of Computer Science. Nevertheless, please keep this disclaimer in mind whenever thinking about Hash Tables.

A **Hash Table** can be thought as a collection of items that, on average, can be retrieved really fast. A Hash Table is implemented by an array, and more formally, we define a Hash Table to be an array of size M (which we call the Hash Table’s *capacity*) that stores keys k by using a hash function to compute an *index* in the array at which to store k .

For example, suppose we have a Hash Table with a capacity of $M = 5$ that stores letters as keys. We would choose a hash function that takes in the ASCII values of letters and maps them to indices (e.g. ‘A’ \rightarrow index 0, ‘B’ \rightarrow index 1, etc.). Since $M = 5$, we need to make sure that any characters we insert into the Hash Table will always map to a valid index (a number between 0 and 4, inclusive). As a result, we choose the hash function to be $H(k) = k \% M$. Inserting the letter ‘A’ into our Hash Table would go through this process:

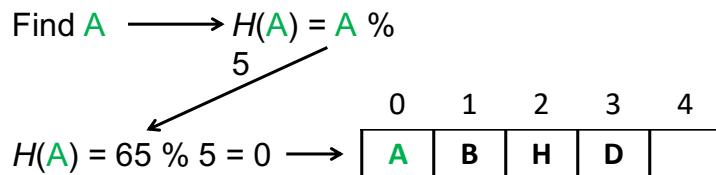


The rest of our insertions would look like this:

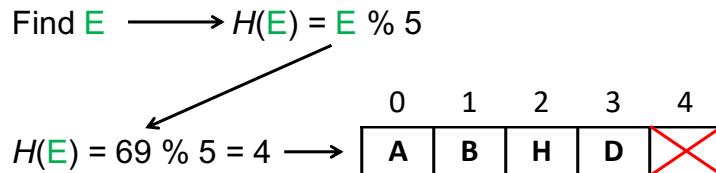
Insert B	A	B			
Insert D	A	B		D	
Insert H	A	B	H	D	

Note: Some of you may have noticed that there is redundancy in our mapping. For example, if we were to attempt to insert the letter 'A' and then the letter 'F', 'F' would have mapped to the slot already occupied by 'A'. More formally, we call this a *collision*. We will discuss how to handle collisions in-depth later on, but for now, just know that they are the cause of any slowness in a Hash Table, so we want to design our Hash Table in such a way that minimizes them.

Hopefully you found the previous example to be pretty cool: we have now found a way to take a key of *any* type and have it map to a valid index in an array, all thanks to our hash function. Moreover, we are now able to find the key in constant time because we know exactly where we expect it to be, also because of our hash function! Finding the letter 'A' in our Hash Table (which should be successful) would go through the following process (note how similar it is to insert):



Similarly, attempting to find the letter 'E' in our Hash Table (which should fail) would be just as easy:



Note that we didn't have to resort to linearly searching or even using some optimized search algorithm such as binary search to find the key. Instead, we were able to find the key in constant time. We got lucky with this simple example, but it turns out that we can formally prove that, if we design our Hash Table intelligently, the *average-case* time complexity to find an element is $\mathcal{O}(1)$ (we will later see why the *worst-case* time complexity is *not* constant-time).

Exercise Break

Which index will the character 'a' hash to in a Hash Table with $M = 7$, with $H(k) = k\%M$ (where we use key k 's ASCII value in the mod operation), and where we are using 0-based indexing?

In general, a Hash Table needs to have the following set of functions:

- `insert(key)`
- `find(key)`
- `remove(key)`
- `hashFunction(key)`: to produce a hash value for a key to use to map to a valid index
- `key_equality(key1, key2)`: to check if two keys are equal

The following is pseudocode for the “insert” operation of a Hash Table. Note that this is a simplified version of the “insert” operation as we do not allow a key to be inserted if another key is already at its index (i.e., if we have a *collision*). We will discuss later how to handle collisions, but for now, we will disallow them. In the following pseudocode, `arr` is the array that is backing the Hash Table, and `H` is the hash function that maps `key` to a valid index.

```

1  insert(key):
2      index = H(key)
3      if arr[index] is empty:
4          arr[index] = key

```

The following is pseudocode for the “find” operation of a Hash Table. Note that this “find” operation is also simplified, as it is based upon the simplified premise of the “insert” function above (i.e., no collisions allowed). In the following pseudocode, `arr` is the array that is backing the Hash Table, and `H` is the hash function that maps `key` to a valid index.

```

1  find(key):
2      index = H(key)
3      return arr[index] == key

```

Exercise Break

If you were to insert the `string` objects `"Orange"`, `"Apple"`, `"Cranberry"`, `"Banana"`, and `"Strawberry"` (in that order) into an initially empty Hash Table with capacity $M = 5$ using the following hash function $H(k)$, at what index will each `string` be inserted? The insert algorithm you will use is the one mentioned previously, in which we don’t allow insertion if there is a collision.

```

1  int H(string k) {
2      char letter = k.at(0); // first character
3      return letter % M;
4  }

```

So far, we have seen that inserting and finding an item can be done in constant time (under the assumptions we made earlier), which is extremely useful. However, what if we wanted to iterate over the items we have inserted in the Hash Table in sorted order?

STOP and Think

Can you think of a good (i.e., efficient) way to print out the keys of the following Hash Table in alphabetical order, where the hash function is defined to be $H(k) = k\%M$?

Insert F	F				
Insert B	F	B			
Insert N	F	B		N	
Final:	F	B		N	

Hopefully you didn't spend too much time on the **STOP and Think** question, because the answer is that there really is unfortunately *no* good way of printing the elements of a Hash Table in sorted order. Why? Because keys are **not** sorted in this data structure! This is very different from a lot of the data structures that we have discussed so far (such as Binary Search Trees) in which we maintained an ordering property at all times. So, in the previous example, if we were to iterate through the keys in our Hash Table, we would expect the output to be "F B N."

What if we were to insert the numbers 1-4 in an arbitrary Hash Table? Could we then expect the keys to be in sorted order? The answer is no! Why? Because we have no clue exactly what hash value the underlying hash function is producing in order to map the key to an index. For example, if $H(k) = 2^k\%M$, then we would expect to see the following:

Insert 1			1		
Insert 2			1		2
Insert 3			1	3	2
Insert 4		4	1	3	2

You might be wondering why we would even be using a hash function as complicated as $H(k) = 2^k \% M$, and you might suspect that we're just being unrealistic for the sake of argument. However, it turns out that using a hash function more complicated than just modding by M to perform the index mapping helps randomize the indices to which keys map, which helps us reduce collisions on average.

In C++, a Hash Table is called an `unordered_set`. The name should be pretty intuitive because we are storing a *set* of keys, and the set is *unordered*. The following is example C++ code to actually use a Hash Table:

```

1 #include<unordered_set> // include C++ Hash Table
2 #include<string>
3 #include<iostream>
4 using namespace std;
5
6 int main() {
7     unordered_set<string> animals = {"Giraffe", "Bear", "Toucan"};
8 }
```

If we wanted to easily iterate over the keys we stored, we could use an iterator. For example, we could add this to the end of `main()`:

```

1 for(auto it = animals.begin(); it != animals.end(); ++it) {
2     cout << *it << endl;
3 }
```

Equivalently, we could have used a for-each loop. For example, we could add this to the end of `main()`:

```

1 for(auto s : animals) {
2     cout << s << endl;
3 }
```

Note: If you were to iterate over the elements of a Hash Table and they *happened* to be iterated over in sorted order, know that it is purely by chance. There is absolutely no guarantee that the elements will be sorted, and on the contrary, because of the fancy hash functions used to help minimize collisions, they will actually most likely not be.

Note: If we wanted to insert a key that was an instance of a custom class, we would need to overload C++'s hash function and key equality methods in order to have the `unordered_set` be able to properly hash the custom class.

So far, all of our insertions have been working smoothly in constant time because we have avoided discussing what happens when a key indexes to a slot that is already occupied (i.e., when we encounter a *collision*). However, this is arguably the most important part of implementing an optimized Hash Table. Why? More often than not, we are mapping from a large space of possible keys (e.g. the numbers 1 to 1,000,000 for a Hash Table storing yearly household

incomes, a large number of strings representing every possible human name for a Hash Table storing Google employee information, etc.) to a much smaller space: the slots of the array that is backing the Hash Table. Since computers do not have unlimited memory, a Hash Table's capacity is in fact *much much* smaller than the vast set of possible keys. As a result, we will inevitably encounter *collisions*: two different keys indexing to the same Hash Table location.

Earlier in this lesson, our pseudocode for the insert operation of a Hash Table simply refused to insert a key if a collision were to occur. In practice, however, we would be facing a terrible data structure if we couldn't insert all the values we wanted. As a result, we need to invest time into thinking about the best possible ways to *avoid* collisions in the first place, as well as *what to do* if a collision were to occur. For the time being, we'll keep the topic of *resolving* collisions on hold just a bit longer, but in the next lesson, we will use probability theory to discuss the causes of collisions in more depth, and we will think about how to choose an optimal **capacity** and a good indexing hash function for a Hash Table to help us *avoid* collisions.

5.4 Probability of Collisions

Up until this point, we have discussed Hash Tables and how they work: you have some array of length M , and to insert a key k into the array, you hash k to some index in the array and perform the insertion. However, even though we addressed the fact that *collisions* can occur (i.e., multiple keys can hash to the same index in the array), we chose to effectively ignore them. We emphasized that collisions are the cause of slowness in a Hash Table, but aside from just repeatedly stating that collisions are bad for performance, we just glossed over them.

In this section, we will begin to chip away at collisions by focusing on *why they occur*, and as a result, *how* we can go about *avoiding* (or at least *minimizing*) them. We still won't be discussing *why they cause slowness* in a Hash Table nor how to go about *resolving* them, but these equally important topics will be extensively discussed soon. For now, just remember that collisions are bad for performance, and because of this, we want to design our Hash Table in such a way that we minimize them.

Thus far, we have witnessed two components of a Hash Table that we are free to choose:

- A Hash Table is backed by an array, so we have the power to choose the **array's size** (i.e., the Hash Table's **capacity**)
- A Hash Table computes indices from keys, so we have the power to choose the hash function that does this index mapping

To figure out how we can optimally design these two parameters of a Hash Table, we will take a journey through the realm of probability theory to see how we can probabilistically minimize collisions. Formally, we will try to find a

way to compute the *expected number* of collisions, and we will then attempt to minimize this value.

To analyze the probability of collisions, we will use this basic identity to simplify our computation: $P(A) = 1 - P(A')$, i.e., the probability of some event A occurring equals 1 minus the probability of its complement, A' , occurring. The basic idea surrounding this identity is that, at the end of the day, there is a 100% chance that an event either will or will not occur (i.e., $P(A) + P(A') = 1$ for some event A). Consequently, we will say that, for a Hash Table with M slots and N keys, the probability of seeing at least one collision is $P_{N,M}(\geq 1 \text{ collision}) = 1 - P_{N,M}(\text{no collision})$. For our purposes, we will assume that all slots of the Hash Table are equally likely to be chosen for an insertion of some arbitrary key. Let's apply this equation to calculate the probability of encountering a collision.

For example, suppose we have a Hash Table where $M = 1$ (i.e., our Hash Table has 1 slot) and $N = 2$ (i.e., we have 2 keys to insert). We know, this doesn't seem like the smartest decision, but suspend your disbelief for the sake of argument.

1. What is the probability that our *first* key does not face collision upon insertion in our not-so-smart Hash Table? The answer is clearly 100%: all the slots in the Hash Table are empty, meaning there is nothing for our first key to collide with.
2. Now, given that the first key did not face a collision, what is the probability that our *second* key does not face collision upon insertion in our not-so-smart Hash Table? The answer is 0%: all the slots in the Hash Table are now full, so it is impossible for us to find an empty slot in which to insert our second key.

Now, if we want to calculate $P_{2,1}(\geq 1 \text{ collision})$, we first need to figure out how to calculate $P_{2,1}(\text{no collision})$. As you may have noticed, there are multiple events we need to take into account (the first *and* second key not facing a collision upon insertion) in order to figure out the *overall* probability of a collision not happening, and that these events are *conditional*: the outcome of earlier events affects the probabilities of later events. To refresh your memory about conditional probabilities, if we have two events A and B (dependent or independent: doesn't matter), we can compute the probability of both events occurring like so: $P(A \text{ and } B) = P(A)P(B|A) = P(B)P(A|B)$

Hopefully you can see that we are dealing with *conditional probabilities*: when we attempt to calculate the probability that the i -th key doesn't face a collision, we assume that each of the previous $i-1$ keys did not face collisions when they were inserted. Formally:

$$\begin{aligned} P_{N,M}(\text{no collision}) &= P_{N,M}(1^{\text{st}} \text{ key no collision}) \\ &\quad \times P_{N,M}(2^{\text{nd}} \text{ key no collision} | 1^{\text{st}} \text{ key no collision}) \dots \end{aligned}$$

Therefore, $P_{N,M}(\text{no collision}) = 1 \times \frac{M-1}{M} \times \frac{M-2}{M} \dots \frac{M-N+1}{M}$

We can therefore conclude that $P_{2,1}(\geq 1 \text{ collision}) = 1 - P_{N,M}(\text{no collision}) = 1 - (1 \times 0) = 1 = 100\%$

Let's see what happens when we increase the size of our Hash Table to $M = 3$ and we wish to insert $N = 3$ keys.

1. What is the probability that our *first* key does *not* face collision upon insertion in this Hash Table? 100%, because the entire backing array is empty, so there is nothing for our first key to collide with
2. Now, given that the first key did not face collision, what is the probability that our *second* key does *not* face collision upon insertion this Hash Table? $\frac{2}{3}$ or 66.67%, because one of the three slots of the Hash Table is already full with the first key, so the second key can settle in any of the remaining 2 spots to avoid collision
3. Now, given that *neither* of the first two keys faced collisions, what is the probability that our *third* key does *not* face collision upon insertion in this Hash Table? $\frac{1}{3}$ or 33.33%, because two of the three slots of the Hash Table are already full, so the third key can only settle in the remaining 1 spot to avoid collision

Therefore, $P_{3,3}(\geq 1 \text{ collision}) = 1 - P_{3,3}(\text{no collision}) = 1 - (1 \times \frac{2}{3} \times \frac{1}{3}) = 1 - \frac{2}{9} = \frac{7}{9} \approx 77.8\%$! This is definitely a bit concerning. We expected 3 keys to be inserted and thus created 3 slots, so why is probability of seeing a collision so high? Should we have created *more* than 3 slots?

Exercise Break

Suppose you have a Hash Table that has 500 slots. It currently holds 99 keys, all in different locations in the Hash Table (i.e., there are no collisions thus far). What is the probability that the next key you insert will cause a collision?

Exercise Break

Calculate $P_{3,5}(\geq 1 \text{ collision})$.

Referring back to the earlier example, we had 3 keys to insert, so we chose to create a Hash Table with a capacity of 3 in order to fit all the keys. However, we calculated an alarmingly high collision probability of 77.8% with that construction. What exactly is going on? This is actually a pretty crazy phenomenon and it is illustrated best by the **Birthday Paradox**. The goal of the Birthday Paradox is to figure out how likely is it that at least 2 people share a birthday given a pool of N people. For example, suppose there are 365 slots in a Hash Table: $M = 365$ (because there are 365 days in a non-leap year). Using the equations from before, we see the following for various values of N individuals:

- For $N = 10$, $P_{N,M}(\geq 1 \text{ collision}) = 12\%$
- For $N = 20$, $P_{N,M}(\geq 1 \text{ collision}) = 41\%$
- For $N = 30$, $P_{N,M}(\geq 1 \text{ collision}) = 71\%$
- For $N = 40$, $P_{N,M}(\geq 1 \text{ collision}) = 89\%$
- For $N = 50$, $P_{N,M}(\geq 1 \text{ collision}) = 97\%$
- For $N = 60$, $P_{N,M}(\geq 1 \text{ collision}) => 99\%$

So, among 60 randomly-selected people, it is almost certain that at least one pair of them will have the same birthday. That is crazy! The fraction $\frac{60}{365}$ is a mere 16.44%, which is how full our Hash Table of birthdays needed to be in order to have almost guaranteed a collision. How can we use this intuition to help us build a Hash Table that minimizes the number of collisions?

We prefaced this discussion about probability with the intention of selecting a capacity for our Hash Table that would help us *avoid* collisions in the average case. Mathematically, this translates into the following question: “What should be the capacity of our Hash Table in order to keep the *expected* (i.e., average-case) number of collisions relatively small?” Let’s see how we can go about computing the **expected total number of collisions** for *any* arbitrary Hash Table.

Suppose we are throwing N cookies into M glasses of milk. The first cookie lands in some glass of milk. The remaining $N - 1$ cookies have a probability $\frac{1}{M}$ of landing in the *same* glass of milk as the first cookie, so the average number of collisions with the *first* cookie will be $\frac{N-1}{M}$ (i.e., $\frac{1}{M}$ added $N - 1$ times to account for the same probability for each remaining cookie left to throw).

Now, let’s say that the second cookie lands in some glass of milk. The remaining $N - 2$ cookies each have probability $\frac{1}{M}$ of landing in the same glass of milk as the second cookie, so the number of collisions with the second cookie will be $\frac{N-2}{M}$, etc.

So, the **expected total number of collisions** is $\sum_{i=1}^{N-1} \frac{i}{M} = \frac{N(N-1)}{2M}$

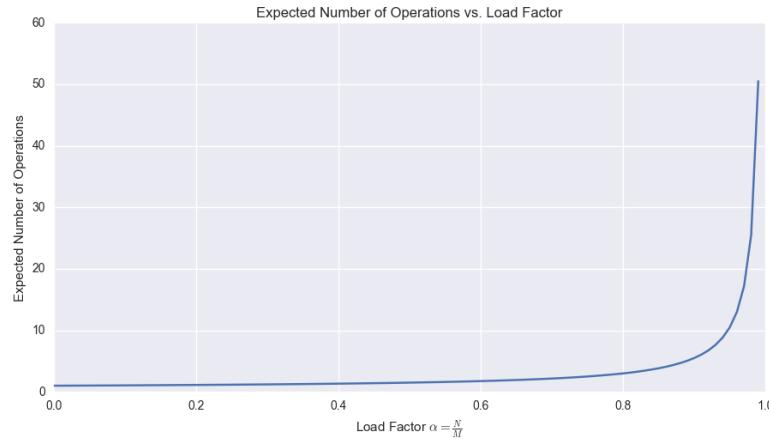
If we want there to be 1 collision on average (which translates into the $\mathcal{O}(1)$ average-case time complexity we desire), we can see that the expected number of collisions will be 1 when $\frac{N(N-1)}{2M} = 1$, which, for a large M , implies $N = \sqrt{2M}$. In other words, if we will be inserting N elements into our Hash Table, in order to keep the make the expected number of collisions equal to 1, we need our Hash Table to have a capacity $M = \mathcal{O}(N^2)$.

We hope that the underlying logic is somewhat intuitive: the more extra space you have, the lower the expected number of collisions. As a result, we also expect a better average-case performance. However, as you might have inferred, this is extremely wasteful space-wise! In practice, making the expected number of collisions exactly 1 is a bit overkill. It turns out that, by some proof that is a bit out-of-scope for this text, we can formulate the expected number

of operations a Hash Table will perform as a function of its **load factor** $\alpha = \frac{N}{M}$:

$$\text{Expected number of operations} = \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right)$$

If we were to plot this function, we would get the following curve:



Notice that the expected number of operations stays pretty flat for quite some time, but when it reaches a load factor somewhere around 0.75, it begins to jump up drastically. By general “rule of thumb,” it has been shown that we can experience $\mathcal{O}(1)$ performance on average when $\alpha = \frac{N}{M} \approx 0.75 \rightarrow M \approx 1.3N$. As a result, if we have a prior estimate of the magnitude of N , the number of elements we wish to insert, we would want to allocate a backing array of size approximately $1.3N$. Also, throughout the lifetime of a specific instance of our Hash Table, if we ever see that the load factor is approaching 0.75, we should consider resizing our array (typically to twice the prior capacity) to keep it low.

With regard to resizing the backing array, however, you should note that, because one step of our indexing process was to mod by M in order to guarantee valid indices, we would need to re-hash (i.e., reinsert) all of the elements from the old array into the new one from scratch, which is a $\mathcal{O}(n)$ operation. In order to avoid performing this slow resizing operation too often, we want to make sure to allocate a reasonably large array right off the bat.

We have now seen why, based on probabilistic analysis, it is important to use a Hash Table that has a capacity larger than the amount of keys we expect to insert in order to avoid collisions. However, it is important to note that we assumed that our keys were always equally likely to be hashed to each of our M indices; is this always a fair assumption?

Not *really*. Take for example the pair of hash functions $h(k) = 3k$ and $H(k) = h(k)\%M$, where k is the key, M is the capacity of the Hash Table, and $H(k)$ returns the index. As you can see, $h(k)$ only returns hash values that are multiples of 3. This shouldn’t be a problem, unless the capacity of the Hash

Table, M , happens to be a multiple of 3 as well (e.g. 6). If that happens, then it will never be possible to have $H(k)$ produce an index that isn't also a multiple of 3 because of the nature of the mod operation (e.g. $3(1)\%6 = \text{index } 3$, $3(2)\%6 = \text{index } 0$, $3(3)\%6 = \text{index } 3$, etc.). Consequently, there will be some indices in the Hash Table that will always be empty (indices 1, 2, 4, and 5 in this example). As a result, by having the keys want to map to the same indices, we face a largely unequal distribution of probabilities across our Hash Table, and as a result, an increase in collisions.

What's the solution? We avoid getting trapped in the problem described above by always choosing the **capacity of our Hash Table to be a prime number**. By definition, a prime number is a natural number that has no positive divisors other than 1 and itself. Consequently, modding by a prime number will guarantee that there are no factors, other than 1 and the prime number itself, that will cause the mod function to never return some index values. For example, using the pair of hash functions above with $M = 7$, we get the following equal distribution of hash values: $3(1)\%7 = \text{index } 3$, $3(2)\%7 = \text{index } 6$, $3(3)\%7 = \text{index } 2$, $3(4)\%7 = \text{index } 5$, $3(5)\%7 = \text{index } 1$, $3(6)\%7 = \text{index } 4$, $3(7)\%7 = \text{index } 0$, etc.

Thus, when we choose a capacity for our Hash Table, even when we are resizing the Hash Table, we always want to round the capacity to the next nearest prime number in order to ensure that that $H(k)$ doesn't yield an unequal distribution. *However*, it is very important to note that while prime numbers smooth over a number of flaws, they don't necessarily solve the problem of unequal distributions. In cases where there is still not a good distribution, the *true* problem lies in a poorly developed hash function that isn't well distributed in the first place.

What does all of this analysis tell us at the end of the day? All of the scary math we trudged through gave us formal proof for the following Hash Table design suggestions:

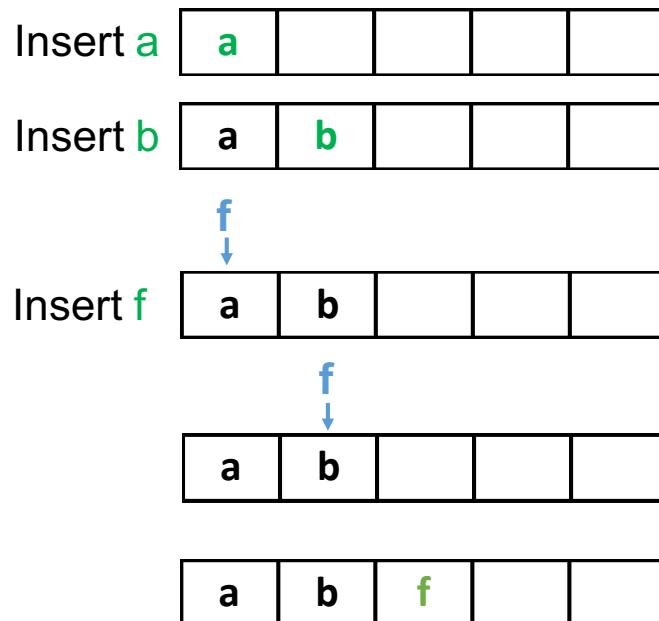
- To be able to maintain an average-case time complexity of $\mathcal{O}(1)$, we need our Hash Table's backing array to have free space
- Specifically, if we expect to be inserting N keys into our Hash Table, we should allocate an array roughly of size $M = 1.3N$
- If we end up inserting more keys than we expected, or if we didn't have a good estimate of N , we should make sure that our load factor $\alpha = \frac{N}{M}$ never exceeds 0.75 or so
- Our analysis was based on the assumption that every slot of our array is equally likely to be chosen by our indexing hash function. As a result, we need a hash function that, on average, spreads keys across the array randomly
- In order to help spread keys across the array randomly, the size of our Hash Table's backing array should be prime

The analysis itself illustrated that, as the size of our data grows, collisions become inevitable. We have now discussed how to try to *avoid* collisions quite extensively, but we are still not equipped to *handle* collisions should they occur. In the next few sections, we will finally discuss various **collision resolution strategies**: how our Hash Table should react when different keys map to the same index.

5.5 Collision Resolution: Open Addressing

As we mentioned repeatedly thus far, we have acknowledged that collisions can happen, and even further, we have proven that they are statistically impossible to fully avoid, but we simply glossed over what to do should we actually encounter one. In this section, we will discuss the first of our **collision resolution strategies** (i.e., what should be done to successfully perform an insert/find/remove operation if we were to run into a collision): **Linear Probing**. The idea behind this strategy is extremely simple: if an object *key* maps to an index that is already occupied, simply shift over and try the next available index.

Here is an intuitive example in which the hash function is defined as $H(k) = (k + 3)\%m$, where k is the ASCII value of *key* and m is the size of the backing array (we add 3 to k to have the letter 'a' hash to index 0 for simplicity):



STOP and Think

Suppose you are inserting a new key into a Hash Table that is already full. What do you expect the algorithm to do?

Exercise Break

What is the probability of a collision for inserting a **new** arbitrary key in the Hash Table above (*after* we inserted the key 'f')?

How do we go about actually implementing this simple collision resolution strategy? Here is the pseudocode to implement Linear Probing when inserting a key k into a Hash Table of capacity M with a backing array called arr , using an indexing hash function $H(k)$:

```

1  insert_LinearProbe(k):
2      index = H(k)
3      Loop infinitely:
4          if arr[index] == k:           // duplicate
5              return
6          else if arr[index] == NULL: // empty slot
7              arr[index] = k; return
8          else:                      // collision
9              index = (index + 1) % M
10         if index == H(k):        // went full circle (full)
11             enlarge arr and rehash all existing elements
12         index = H(k)

```

Notice that the core of the Linear Probing algorithm is defined by this equation: `index = (index + 1) % M`

By obeying the equation above, we are ensuring that our key is inserted strictly within an index (or more formally, an address) *inside* our Hash Table. Consequently, we call this a **closed hashing** collision resolution strategy (i.e., we will insert the actual key only in an address bounded by the realms of our Hash Table). Moreover, since we are inserting the keys themselves into the calculated addresses, we like to say that Linear Probing is an **open addressing** collision resolution strategy (i.e., the keys are open to move to an address other than the address to which they initially hashed).

Note: You will most likely encounter people using the terms **closed hashing** and **open addressing** interchangeably since they arguably describe the same method of collision resolution.

STOP and Think

How would we go about *finding* a key using Linear Probing?

Exercise Break

Suppose we have an empty Hash Table, where $H(k) = k\%M$ and $M = 7$. After inserting the keys 31, 77, and 708 into our Hash Table (in that order), which index will the key 49 end up hashing to using the collision resolution strategy of Linear Probing?

Exercise Break

Using the collision resolution strategy of Linear Probing, fill in the missing elements in the Hash Table, where $H(k) = k\%M$ and $M = 5$, after inserting all of the following elements (in order): 5, 10, 11, 12, and 13

Exercise Break

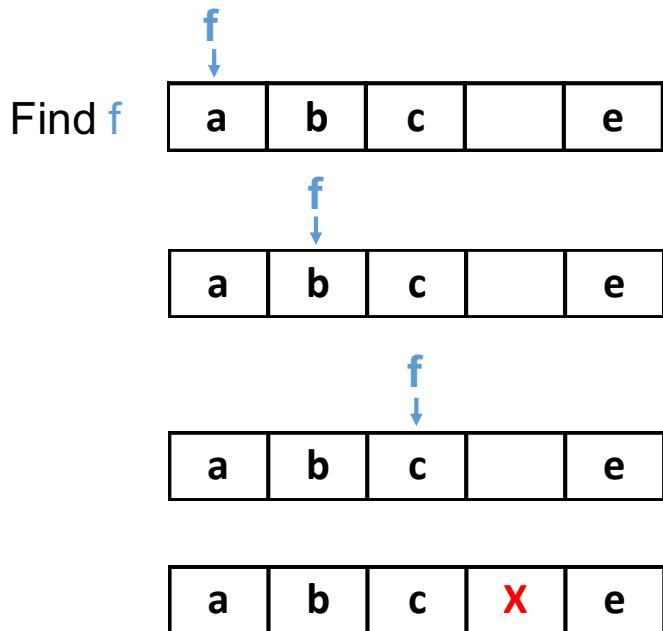
Sort the following keys in the order that they must have been inserted into the following Hash Table based on the following facts:

- Linear Probing was used as the collision resolution strategy
- $H(k) = k\%M$, where k is the ASCII value of the key and M is the size of the backing array

0	1	2	3	4
R		H	C	D

Hint: Feel free to use the chart located at ASCII Table.

We've discussed how to insert elements using Linear Probing, but how do we go about deleting elements? In order to answer this question, we first need to take a look at how the *find* algorithm works in a Hash Table using Linear Probing. In general, in order to not have to search the *entire* Hash Table for an element, the find algorithm will search for an element until it finds an empty slot in Hash Table, at which it will terminate its search. For example, using the same hash function as we did previously, $H(k) = (k + 3)\%m$, we would find an element like so:



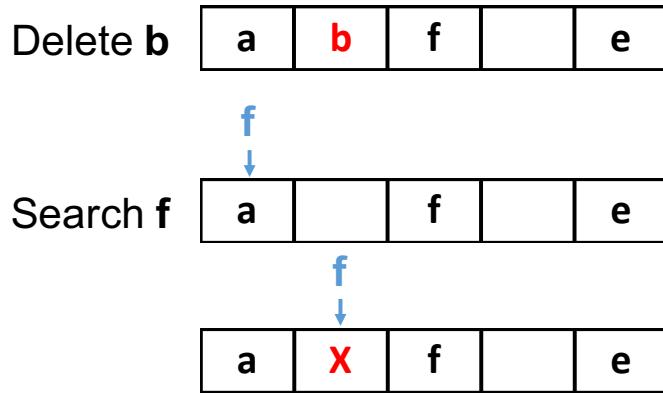
STOP and Think

Why does this always work?

Now that we know how the find operation works, let's look at how we might go about deleting an element in a Hash Table that uses Linear Probing. Intuitively, you might think that we should delete the key by simply emptying the slot. Would that work? Suppose we have the following Hash Table:

a	b	f		e
---	---	---	--	---

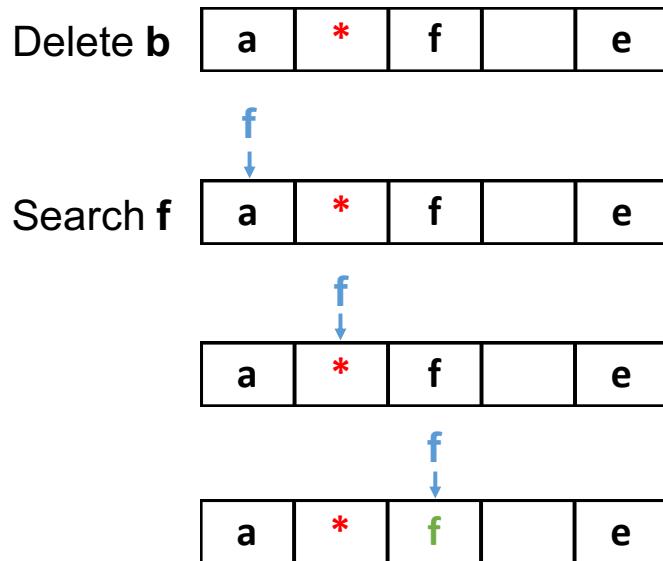
Now suppose we use the deletion method described above to delete 'b' and then search for 'f'. Let's see what happens:



As we see in the example above, the key 'f' would not have been found, which is clearly a problem.

Note: You might be thinking “Why don’t we just edit the find method so that it doesn’t stop when a slot is empty?.” This would work, but if we were to do so, our find operations would become *significantly* less efficient. In practice, the find operation is used much more frequently than the remove operation, and as a result, it makes more sense to keep the find operation as optimized as possible.

Consequently, the solution to this conundrum is to use a “deleted” flag. In other words, we need to specify to the Hash Table that, even though there is in fact no element in that slot anymore, we still need to look beyond it since we have *deleted* from the slot.



Now that we have discussed a proper remove algorithm, it is important to note that we face the problem of having these “deleted” flags piling up, which would make our find algorithm less and less efficient. Unfortunately, the only solution to this problem is to regenerate a new Hash Table and reinsert all valid (i.e., not deleted) keys into the new Hash Table.

As you hopefully agree, Linear Probing is a fairly simple and straightforward solution to collision resolution. What’s the catch? We had originally introduced Hash Tables because we wanted to achieve extremely fast find, insert, and remove operations. Unfortunately, Linear Probing slows us down extensively. As you might have noticed while doing the previous exercises, Linear Probing can resort to having to *linearly scan* the Hash Table to find an open slot to insert a key. As a result, our *worst-case* time complexity for an insert operation becomes $\mathcal{O}(N)$. The same goes for attempting to *find* a key in the Hash Table: in the worst-case scenario, we must linearly scan the entire Hash Table to see if the key exists. However, with a table that is not very full, we can in fact achieve an *average-case* $\mathcal{O}(1)$ time complexity for our operations.

Another negative quality about Linear Probing that you may have noticed is that it results in clusters, or “clumps,” of keys in the Hash Table. It turns out that clumps are not just “bad luck”: probabilistically, they are actually *more likely* to appear than not! To demonstrate this, we will use the following empty Hash Table:

0	1	2	3	4

If we were to insert a new key, it would have a $\frac{1}{5}$ chance of landing in any of the 5 slots. Let’s say that, by chance, it landed in slot 0:

0	1	2	3	4
X				

Now, if we were to insert a new key again, how likely is it to land in any of the four remaining slots? Intuitively, you might say that it would have a $\frac{1}{4}$ chance of landing in any of the 4 remaining slots, but unfortunately, this is not correct. Remember: even though it can’t be inserted in slot 0, it can still *index* to slot 0! For slots 2, 3, and 4, each has a $\frac{1}{5}$ chance of being filled by our new key (if the key indexes to 2, 3, or 4, respectively). What about slot 1? If the new key indexes to slot 1, it will be inserted into slot 1. However, remember,

if the element indexes to slot 0, because of Linear Probing, we would shift over and insert it into the next open slot, which is slot 1. As a result, there are two ways for the new key to land in slot 1, making the probability of it landing in slot 1 become $\frac{2}{5}$, which is twice as likely as any of the other three open slots! Because this probability is twice as large as the others, let's say we ended up inserting into slot 1:

0	1	2	3	4
X	X			

Now, what if we were to insert another key? Just as before, slots 3 and 4 each have a $\frac{1}{5}$ chance of being filled (if we were to index to 3 or 4, respectively). To fill slot 2, we could index to 2 with $\frac{1}{5}$ chance, but what would happen if we were to index to 1? Again, because of Linear Probing, if we index to 1, we would see the slot is filled, and we would shift over to slot 2 to perform the insertion. Likewise, if we index to 0, we would see the slot is filled and shift over to slot 1, see the slot is filled and shift over to slot 2, and *only then* perform the insertion. In other words, the probability of landing in slot 2 is $\frac{3}{5}$, which is **three times as likely** as any of the other slots!

We explained earlier *why* clumps are bad (because we potentially have to do a linear scan across the clump, which is slow), but now we've actually shown that clumps are *more likely to appear and grow* than not! What can we do to combat this issue?

Intuitively, you might think that, instead of probing one slot over during each step of our scan, we could just choose a larger “skip”: why not jump *two* slots over? Or *three* slots over? This way, the elements would be more spaced out, so we would have solved the issue of clumps, right? Unfortunately, it's not that simple, and the following example will demonstrate why simply changing the skip from 1 to some larger value doesn't actually change anything. Let's start with the following Hash Table that contains a single element, but this time, let's use a skip of 3:

0	1	2	3	4
X				

What would happen if we were to try to insert a new element? Like before, it has a $\frac{1}{5}$ chance of indexing to 0, 1, 2, 3, or 4. If it happens to land in slots

1, 2, 3, or 4 (each with $\frac{1}{5}$ probability), we would simply perform the insertion. However, if it were to land in slot 0 (with probability $\frac{1}{5}$), we would have a collision. Before, in traditional Linear Probing, we had a skip of 1, so we shifted over to slot 1, so the probability of inserting in slot 1 was elevated (to $\frac{2}{5}$). Now, if we index to 0, slot 3 has an elevated probability of insertion (to $\frac{2}{5}$, like before). In other words, by using a skip of 3, we have the exact same predicament as before: one slot has a higher probability of insertion than the others! All we've done is *changed* which slot it is that has a higher probability!

0	1	2	3	4
X			X	

Even though it might *appear* as though the clumps have disappeared because there's more space between filled slots, it turns out that the clumps still do in fact exist: they are just harder to see. The reason *why* clumps tend to grow is the exact probabilistic issue we saw previously: with the initial indexing, all slots are equally likely to be filled, but upon collision, we deterministically increase the probability of filling certain slots, which are the slots that expand a clump. Instead, we need to think of a clever way to somehow *evenly distribute* the insertion probabilities over the open slots, but in a way that is deterministic such that we would still be able to find the key if we were to search for it in the future.

The solution to avoid keys having a higher probability to insert themselves into certain locations—thereby avoiding the creation of clumps—is pretty simple: designate a *different* offset for each particular key. Once we do this, we ensure that we have an *equal* probability for a key to hash to any slot. For example, suppose we start with the following Hash Table:

0	1	2	3	4
X				

Our initial problem in Linear Probing was that slot 1 had an unequal probability of getting filled (a $\frac{2}{5}$ chance). This was because slot 1 had two ways of getting filled: if key k initially hashed to index 1 *or* index 0, thereby guaranteeing that the key k would also hash to index 1. By designating a *different* offset for each particular key, however, we no longer have the guarantee that just because key k initially hashed to index 0, it will also hash to index 1. As

a result, there is no guarantee for a *particular* area in the Hash Table to face a higher probability for keys to hash there and we thus consequently eliminate the inevitable creation of clumps.

We do what is described above using a technique called Double Hashing. The concept behind the collision resolution strategy of **Double Hashing** is to use two hash functions: $H_1(k)$ to calculate the hashing index and $H_2(k)$ to calculate the *offset* in the probing sequence. More formally, $H_2(k)$ should return an integer value between 1 and $M - 1$, where M is the size of the Hash Table. You can think of Linear Probing as originally having $H_2(k) = 1$ (i.e., we always moved the key 1 index away from its original location). A common choice in Double Hashing is to set $H_2(K) = 1 + \frac{K}{M} \% (M - 1)$.

The following is the pseudocode to implement Double Hashing when inserting a key k into a Hash Table of capacity M with a backing array called `arr`, using a hash function $H(k)$. Note that the pseudocode for Double Hashing is extremely similar to that of Linear Probing:

```

1  insert_DoubleHash(k):
2      index = H1(k) ; offset = H2(k)
3      Loop infinitely:
4          if arr[index] == k:           // duplicate
5              return
6          else if arr[index] == NULL: // empty slot
7              arr[index] = k; return
8          else:                      // collision
9              index = (index + offset) % M
10         if index == H(k):          // went full circle (full)
11             throw an exception OR enlarge table

```

STOP and Think

Is Double Hashing an *open addressing* collision resolution strategy?

Another collision resolution strategy that solves Linear Probing's clumping problem is called **Random Hashing**. The idea behind Random Hashing is that we use a pseudorandom number generator **seeded by the key** to produce a **sequence** of hash values. Once an individual hash value is returned, the algorithm just mods it by the capacity of the Hash Table, M , to produce a valid index that the key can map to. If there is a collision, the algorithm just chooses the next hash value in the pseudorandomly-produced sequence of hash values.

The following is pseudocode for implementing Random Hashing when inserting a key k into a Hash Table of capacity M with a backing array called `arr`:

```

1  insert_RandomHash(k):
2      RNG = new Pseudorandom Number Generator seeded with k
3      nextNumber = next pseudorandom number generated by RNG
4      index = nextNumber % M
5      Loop infinitely:
6          if arr[index] == k: // duplicate
7              return
8          else if arr[index] == NULL: // empty slot
9              arr[index] = k; return
10         else:
11             nextNumber = next number generated by RNG
12             index = nextNumber % M
13             if all M locations have been probed: // full
14                 throw an exception OR enlarge table

```

An important nuance of Random Hashing is that we must seed the pseudorandom number generator by the key. Why? Because we need to make sure that our hash function is deterministic (i.e., that it will *always* produce the same hash value sequence for the same key). Therefore, the only way we can do that is to guarantee to always use the same seed: the key we are currently hashing. In practice, Random Hashing is considered to work just as well as Double Hashing. However, *very good* pseudo random generation can be an inefficient procedure, and as a result, it is more common to just stick with Double Hashing.

By merely introducing a second offset hash function, Double Hashing turns out to outperform Linear Probing in practice because we no longer face the inevitable fast deterioration of performance resulting from clumps of keys dominating the Hash Table. By using a deterministic second hash function that depends upon the key being inserted, we are able to get a more uniform distribution of the keys while still having the ability to figure out where we expect to find the key. The same applies for Random Hashing. However, the remove algorithm for Double Hashing and for Random Hashing still turn out to be as wasteful as for Linear Probing, so we still have to worry about reinserting elements into the Hash Table periodically to clean up the “delete” flags. Also, in all the open-addressing methods discussed so far, the probability of *future* collisions increase each time an inserting key faces a collision.

Consequently, in the next lesson, we will explore an alternative collision resolution strategy called Separate Chaining, which vastly simplifies the process of deleting elements, and more importantly, doesn’t necessarily require the probability of *future* collisions to increase each time an inserting key faces a collision in our Hash Table.

5.6 Collision Resolution: Closed Addressing

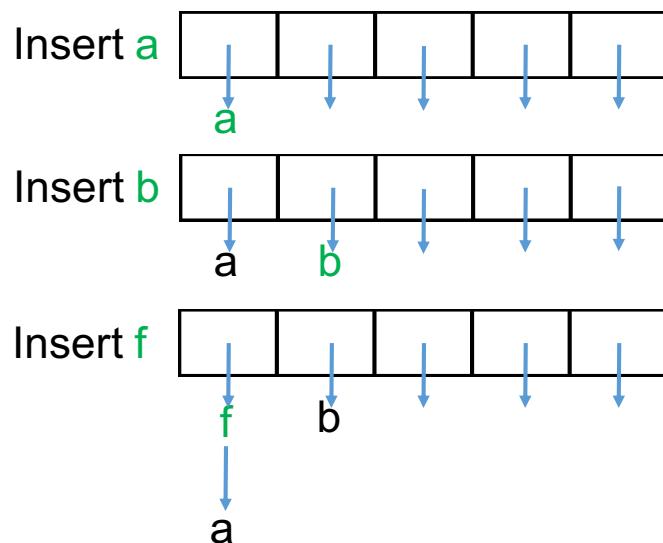
In the previous section, we discussed our first attempt at handling collisions using a technique called Linear Probing where, if a key maps to a slot in our Hash Table that is already occupied by a different key, we simply slide over and try again; if that slot is also full, we slide over again; etc. On *average*, assuming we’ve designed our Hash Table intelligently, we experience $\mathcal{O}(1)$ find, insert, and

remove operations with Linear Probing. However, in the *worst* case, we could theoretically have to iterate over all N elements to find a key of interest, making our three operations $\mathcal{O}(N)$. Also, we saw that keys tend to form “clumps” that make probing excessively slow, and these clumps are probabilistically favored to grow. We introduced Double Hashing and Random Hashing, which helped solve our issue of clump formation and helped speed things up a bit.

Recall that, if we try to insert a key and *don't* encounter a collision, the probability of subsequent collisions must increase, regardless of how we choose to handle collisions (because we have now filled a previously-unfilled slot of our Hash Table). If we try to insert a key and we *do* encounter a collision, with Linear Probing, we take a performance hit on two accounts: we have to linearly scan until we find an open slot for insertion (so we take a hit now), and because we've now filled a previously-unfilled slot of our Hash Table, we have yet again increased the probability of subsequent collisions (so we will take a hit in a future insertion).

Upon a collision, the current insertion will take a performance hit, no matter which collision resolution strategy we choose. In other words, the performance hit we take now is inevitable. However, in this section, we will discuss another collision resolution strategy **Separate Chaining**, in which collisions *do not* increase the probability of future collisions, so we avoid taking the additional performance hit in the future. The basic idea behind Separate Chaining is to keep pointers to Linked Lists as the keys in our Hash Table.

Here is an intuitive example using Separate Chaining in which the hash function, the same we used in the previous section, is defined as $H(k) = (k + 3)\%m$, where k is the ASCII value of the key and m is the size of the backing array (we add 3 to k to have the letter 'a' hash to index 0 for simplicity):



STOP and Think

Can the Hash Table above ever get full?

Exercise Break

What is the probability of a collision for inserting a **new** arbitrary key in the Hash Table above (*after* we inserted the key 'f')?

STOP and Think

Is the probability larger or smaller than it was when we used Linear Probing?

How do we go about implementing this collision resolution strategy? The following is pseudocode to implement Separate Chaining when inserting a key k into a Hash Table of capacity M with a backing array called arr , using a hash function $H(k)$:

```

1  insert_SeparateChaining(k):
2      index = H(k)
3      if Linked List in arr[index] does not contain k:
4          insert k into Linked List at arr[index]
5      if n/m > loadFactorThreshold:
6          resize and recopy elements

```

Notice that the core of the Separate Chaining algorithm is defined by this line: `insert k into Linked List at arr[index]`

As you might have noticed, we have been pretty vague as to which implementation of a Linked List (Singly-Linked, Doubly-Linked, etc.) to use because it is really up to the person implementing the Hash Table to decide exactly which Linked List implementation to choose based on his or her needs. Just like we have been discussing throughout this entire text, we can get vastly different time complexities based on which backing implementation we use and *how* we chose to use it.

A common implementation choice is to use a Singly-Linked List with just a head pointer. By choosing that implementation, we can consequently add new elements to the *front* of our Linked List (i.e., traverse the entire Linked List and then reassign the next pointer of the last element), which would maintain a worst-case constant time complexity. However, in doing so, we would potentially accrue duplicate elements, the consequences of which we will discuss shortly.

By obeying the algorithm above, we are ensuring that our key is inserted strictly within the *single* index (or more formally, "address") to which it originally hashed. Consequently, we call this a **closed addressing** collision resolution strategy (i.e. the key *must* be located in the original address). Moreover, since we are now allowing multiple keys to be at a single index, we like to say that Separate Chaining is an **open hashing** collision resolution strategy (i.e., the keys do not necessarily need to be physically inside the Hash Table itself).

Note: You will most likely encounter people using the terms **open hashing** and **closed addressing** interchangeably since they arguably describe the same method of collision resolution.

Exercise Break

Consider a Hash Table with 100 slots. Collisions are resolved using Separate Chaining. Assuming that there is an equal probability of mapping to any index of the Hash Table, what is the probability that the first 3 slots (index 0, index 1, and index 2) are unfilled after the first 3 insertions?

We briefly mentioned previously that, by not checking for duplicates in the *insert* method, we could get a worst-case *constant* time complexity. So how does checking for duplicates even work in the first place? We can check for duplicate insertions using two different methods:

1. Check for duplicates during the *insert* operation by linearly scanning the corresponding Linked List *before* inserting a new element (and simply not inserting it if it already exists in the Linked List). In this case, during deletion, we can simply stop searching the Linked List for the element the first time we find it.
2. Check for duplicates during the *delete* operation by linearly scanning the *entire* corresponding Linked List, even if we find the element. In this case, during insertion, we can simply insert at the head of the Linked List.

Note that, if we were to forgo checking for duplicates in the *insert* operation, we could end up with multiple copies of the same element stored in the Linked List at a given position, which would potentially slow down *find* and *delete* operations significantly. For example, imagine the following pseudocode:

```

1 t = empty hash table (linear probing)
2 iterate 100 times:
3     insert 42 into t
4 insert 84 into t // assume 84 hashes to same slot as 42

```

If we were to assume that 42 and 84 hash to the same slot in the backing array, if we were to try to find 84, we would have to iterate over a Linked List containing 101 elements before finding it even though the Hash Table only contains 2 elements! Thus, it may be preferable to handle checking for duplicates in the *insert* operation to avoid this.

Exercise Break

Consider a Hash Table in which collisions are resolved using Separate Chaining. Select the tightest worst-case time complexity for an insert function that uses a Doubly-Linked List (i.e., there are head and tail pointers, and you can traverse

the list in either direction) and inserts keys to the end of the Linked Lists. This insert function does not allow duplicates.

How would we go about implementing the remove and find functions for a Hash Table that uses Separate Chaining? The algorithm is actually quite simple: hash to the correct index of the Hash Table, and then search for the item in the respective Linked List. Note that this remove algorithm is *much* easier than the remove algorithm we had to use in Linear Probing.

It is also important to note that Separate Chaining does not necessarily *have* to use Linked Lists! A Hash Table can have slots that point to AVL Trees, Red-Black Trees, etc., in which case the worst-case time complexity to find an item would definitely be faster. However, the reason why we use Linked Lists is that we do not expect the worst-case time complexity of finding an item in a Hash Table slot to make a huge difference. Why? Because if it did, that would mean that the rest of our Hash Table is performing poorly and that would imply that we should probably investigate why that might be the case and fix it (perhaps the hash function isn't good or the load factor has become too high).

So what are the advantages and disadvantages of Separate Chaining? In general, the average-case performance is considered much better than Linear Probing and Double Hashing as the amount of keys approaches, and even *exceeds*, M , the capacity of the Hash Table (though this proof is beyond the scope of the text). This is because the probability of *future* collisions does not increase each time an inserting key faces a collision with the use of Separate Chaining: in the first **Exercise Break** of this section *and* the previous section we saw that inserting the key 'f' into a Hash Table that uses Separate Chaining kept the probability of a new collision at 0.4 as opposed to increasing it to 0.6 as with Linear Probing. It is also important to note that we could never have exceeded M in Linear Probing or Double Hashing (not that we would want to in the first place) without having to resize the backing array and reinsert the elements from scratch.

A disadvantage for Separate Chaining, however, is that we are now dealing with a bunch of pointers. As a result, we lose some optimization in regards to memory for two reasons:

1. We require extra storage for pointers (when storing primitive data types).
2. All the data in our Hash Table is no longer huddled near a single memory location (since pointers can point to memory anywhere), so this poor locality causes poor cache performance (i.e., it often takes the computer longer to find data that isn't located near previously accessed data)

As we have now seen, Separate Chaining is a closed addressing collision resolution strategy that takes advantage of other data structures (such as a linked list) to store multiple keys in one Hash Table slot. By storing *multiple* keys in *one* Hash Table slot, we ensure that the probability of *future* collisions does not increase each time an inserting key faces a collision—something that we

saw happen in the open addressing methods—thereby giving us more desirable performance.

Nonetheless, we will continue to explore other collision resolution strategies in the next section, where we will look at a collision resolution strategy called Cuckoo Hashing that not only resolves collisions, but that takes extra measures to avoid collisions in the first place.

5.7 Collision Resolution: Cuckoo Hashing

The final collision resolution strategy that we will discuss is called **Cuckoo Hashing**. Cuckoo Hashing—and its weird name—comes from the concept of actual Cuckoo chicks pushing each other out of their nests in order to have more space to live. In all of the previous open addressing collision resolution strategies we have discussed thus far, if an inserting key collided with a key already in the Hash Table, the existing key would remain untouched and the inserting key would take responsibility to find a new place. In Cuckoo Hashing, however, an arguably more aggressive and opposite approach takes place: if an inserting key collides with a key already in the Hash Table, the inserting key pushes out the key in its way and takes its place. The displaced key then hashes to a new location (ouch...).

More formally, Cuckoo Hashing is defined as having two hash functions, $H_1(k)$ and $H_2(k)$, both of which return a position in the Hash Table. As a result, one key has strictly two different hashing locations, where $H_1(k)$ is the first location that a key always maps to (but doesn't necessarily always stay at). We have created a visualization of a simplified version of Cuckoo Hashing, which uses only one Hash Table instead of two (soon to be explained), which can be found at <https://goo.gl/ZFkhQL>.

Cuckoo Hashing was invented with the use of two Hash Tables simultaneously in order to decrease the probability of collisions in each table. Formally, the hash function $H_1(k)$ hashes keys *exclusively* to the *first* Hash Table T_1 , and the hash function $H_2(k)$ hashes keys *exclusively* to the *second* Hash Table T_2 . A key k starts by hashing to T_1 , and if another arbitrary key j collides with key k at some point in the future, key k then hashes to T_2 . However, a key can also get kicked out of T_2 , in which case it hashes back to T_1 and potentially kicks out another key. We admit that this may sound confusing, so we have created another visualization to show how the keys jump around between the two tables, which can be found at <https://goo.gl/cTpMYh>.

How do we go about actually implementing this seemingly complex collision resolution strategy? Here is pseudocode to implement Cuckoo Hashing when inserting a key k , with two tables $t1$ and $t2$, both of capacity M , with backing arrays arr1 and arr2 , respectively, using a hash function $H1(k)$ and $H2(k)$, respectively:

```

1  insert_CuckooHash(k):
2      index1 = H1(k); index2 = H2(k)
3      if arr1[index1] == k or arr2[index2] == k: // duplicate
4          return False
5      current = k
6      while looping less than MAX times: // MAX is commonly 10
7          oldValue = arr1[H1(current)] // save key to displace
8          arr1[H1(current)] = current // insert new key
9          if oldValue == NULL: // if empty slot,
10             return True // success
11         current = oldValue // reinsert
12         oldValue = arr2[H2(current)] // save key to displace
13         arr2[H2(current)] = current // insert the new key
14         if oldValue == NULL: // if empty slot,
15             return True // success
16         current = oldValue // repeat (displaced key)
17     // loop ended: insertion failed (need to rehash table)
18     // rehash commonly done by introducing new hash functions
19     return False

```

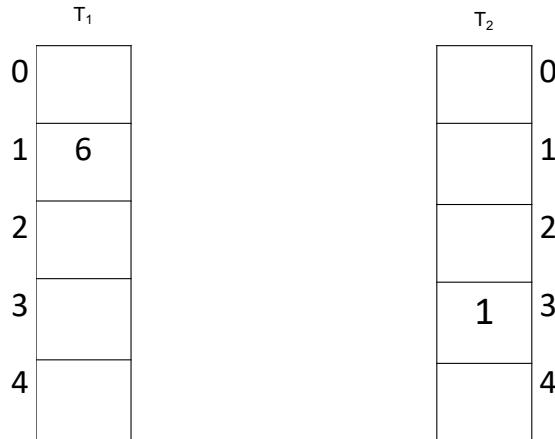
STOP and Think

Is Cuckoo Hashing considered to be an open addressing collision resolution strategy?

Exercise Break

Based on the following Hash Tables and following hash functions, determine the corresponding index to which each of the keys will hash using Cuckoo Hashing *after* key 11 is inserted.

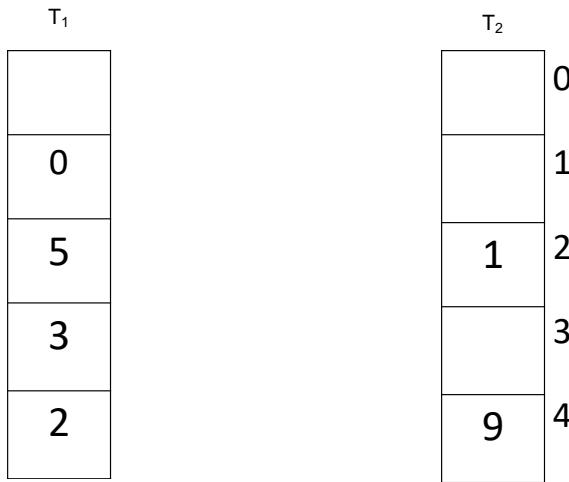
$$\begin{aligned}H_1(k) &= k \% M \\H_2(k) &= (3^k) \% M\end{aligned}$$



Exercise Break

Based on the following Hash Tables and hash functions, determine the corresponding index to which each of the keys will hash using Cuckoo Hashing *after* key 7 is inserted.

$$\begin{aligned} H_1(k) &= (2^k) \% M \\ H_2(k) &= (k + 1) \% M \end{aligned}$$

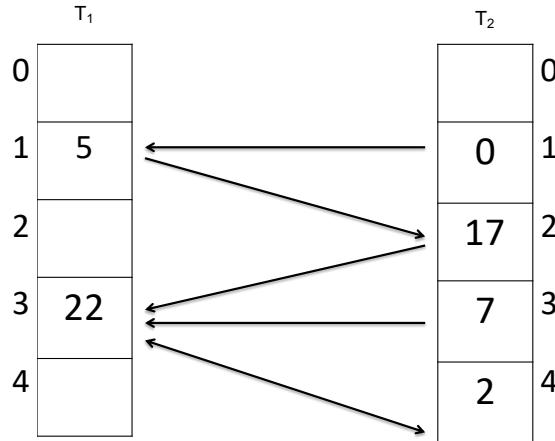


In the previous examples, we saw that the bulk of the algorithm ran inside a while-loop that was bounded by a MAX limit. Why do we even have a limit in the first place? You may have noticed that, in the second visualization, we showed that inserting a key started to take longer because different keys started bouncing back and forth between places. For example, inserting the integer key 17 took 4 iterations of hashing. As both tables begin to fill up, the probability of collisions increases. *However*, unlike the other collision resolution strategies we discussed in which a key k could end up trying every single Hash Table slot until the entire Hash Table was full (e.g. Linear Probing, Double Hashing, Random Hashing) a key k in Cuckoo Hashing only has two different locations that it can map to (index 1 = $H_1(k)$ and index 2 = $H_2(k)$). Consequently, if both Hash Table locations for *all* keys are full, then the algorithm faces an infinite cycle. We have created a visualization to exemplify this, which can be found at <https://goo.gl/SF5WhJ>.

Specifically, the reason why we were in an infinite cycle in the previous visualization is that there were no empty slots in either T_1 or T_2 in which a key could potentially land, like so:

$$H_1(k) = (k + 1) \% M$$

$$H_2(k) = (2^k) \% M$$



Consequently, unless Cuckoo Hashing has a limit as to how many times it attempts to move a key, the algorithm will never stop. As seen in the pseudocode previously, when a key causes an infinite loop and the insertion fails, we must then resort to rehashing. Rehashing is generally done by introducing two new hash functions and reinserting all the elements.

Note: It is important to make sure that the second hash function used returns *different* indices for keys that originally hashed to the same index. This is because, if a key collides with another key in the first Hash Table, we want to make sure that it will not collide with the same key again in the second Hash Table. Otherwise, we risk hitting a cycle the moment we insert two keys that hash to the same first index.

STOP and Think

Let k be a key and M be the capacity of the Hash Table. Why would the pair of hash functions $H_1(k) = k \% M$ and $H_2(k) = (k + 3) \% M$ not be considered good?

By making the sacrifice of only allowing each key to hash to strictly two different locations (thereby potentially causing the cycle in the first place), we end up getting a reward of a **worst-case constant** time complexity for two of our major operations! Specifically:

- For the **find** operation: if the key is not in either index $1 = H_1(k)$ or index $2 = H_2(k)$, then it is not in the table; this is a constant time operation
- For the **delete** operation: if the key exists in our table, we know that it is either in index $1 = H_1(k)$ or index $2 = H_2(k)$, and all we have to do is remove the key from its current index; this is a constant time operation

This is unique to Cuckoo Hashing because, in all of the other collision resolution strategies we have discussed thus far, we could only guarantee an *average-case* constant time complexity with respect to those two operations because, in the worst case, we had to traverse the entire Hash Table.

For the “insert” operation in Cuckoo Hashing, however, we only get an *average-case* constant time complexity because, in the worst case, we would have to rehash the entire table, which has an $\mathcal{O}(n)$ time complexity. However, we can prove through amortized cost analysis (which is beyond the scope of this text) that, on average, we can finish an insertion before the algorithm is forced to terminate and rehash.

Fun Fact: A lot of proofs about cycles in Cuckoo Hashing are solved by converting the keys within the two Hash Tables to nodes and their two possible hashing locations to edges to create a graph theory problem!

Note: Cuckoo Hashing is not necessarily restricted to using just two Hash Tables; it is not uncommon to use more than two Hash Tables, and generally, for d Hash Tables, each Hash Table would have a capacity of $\frac{M}{d}$, where M is the calculated capacity of a single Hash Table (i.e., the capacity that we would have calculated had we decided to use a different collision resolution strategy that required only one Hash Table).

Exercise Break

Which of the following statements about Cuckoo Hashing are true? Assume that we are using only two Hash Tables in our implementation of Cuckoo Hashing.

- Cuckoo Hashing has a worst-case unbounded time complexity with regards to the insert operation.
- Cuckoo Hashing is able to rehash in constant time, which is why it has an average case constant time complexity.
- In Cuckoo Hashing, each key is not able to hash to more than 2 locations.
- In Cuckoo Hashing, an inserting key never needs to use the second hash function when it is first inserted.

Hopefully by now you are convinced that Cuckoo Hashing is by far the most optimized open addressing collision resolution strategy that we have discussed in terms of worst-case time complexities. Specifically, it provides us a guaranteed *worst-case constant* time complexity in the “find” and “remove” operations that the previous strategies were unable to guarantee.

5.8 Hash Maps

Thus far in the text, we have only discussed *storing keys* in a data structure, and we have discussed numerous data structures that can be used to find, insert, and remove keys (as well as their respective trade-offs). Then, throughout

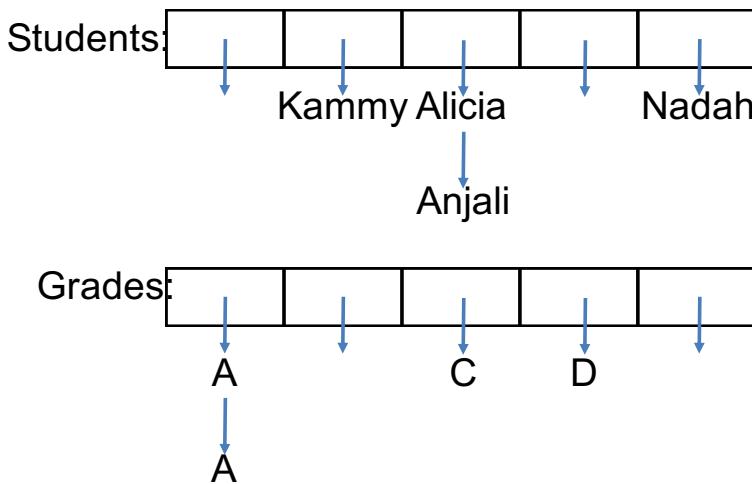
this chapter, we introduced the Hash Table, a data structure that, on average, performs *extremely fast* (i.e., constant-time) find, insert, and remove operations.

However, what if we wanted to push further than simply storing *keys*? For example, if we were to teach a class and wanted to store the students' names, we could represent them as strings and store them in a Hash Table, which would allow us to see if a student is enrolled in our class in constant time. However, what if we wanted to *also* store the students' grades? In other words, we can already query a student's name against our Hash Table and receive "true" or "false" if the student is in our table or not, but what if we want to instead return the student's grade?

The functionality we have described, in which we query a *key* and receive a *value*, is defined in the **Map** ADT, which we will formally describe very soon. After discussing the Map ADT, we will introduce a data structure that utilizes the techniques we used in Hash Tables to *implement* the Map ADT: the **Hash Map**.

Sad Fact: Most people do not know the difference between a Hash Table and a Hash Map, and as a result, they use the terms interchangeably, so beware! As you will learn, though a Hash Map is built on the same *premise* as a Hash Table, it is a bit different and arguably more convenient in day-to-day programming.

As we mentioned, our goal is to have some efficient way to store students and their grades such that we could query our data structure with a student's name and then have it return to us their grade. We could, of course, store the student names and grades in separate Hash Tables, but how would we know which student has which grade?



Consequently, we want to find a way to be able to store a student's name *with* his or her letter grade. This is where the **Map** Abstract Data Type comes into play! The Map ADT allows us to *map* keys to their corresponding values.

The Map ADT is often called an *associative array* because it gives us the benefit of being able to *associatively* cluster our data.

Formally, the Map ADT is defined by the following set of functions:

- **put(key, value)**: perform the insertion, and return the previous *value* if overwriting, otherwise `NULL`
- **has(key)**: return true if *key* is in the Map, otherwise return false
- **remove(key)**: remove the *(key, value)* pair associated with *key*, and return *value* upon success or `NULL` on failure
- **size()**: return the number of *(key, value)* pairs currently stored in the Map
- **isEmpty()**: return true if the Map does not contain any *(key, value)* pairs, otherwise return false

The Map ADT can theoretically be implemented in a multitude of ways. For example, we could implement it as a Binary Search Tree: we would store two items inside each node, the *key* and the *value*, and would keep the Binary Search Tree ordering property based on just *keys*.

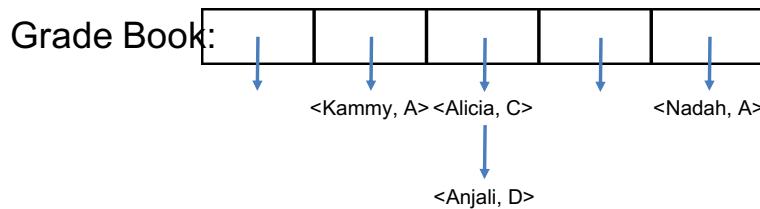
However, if we didn't care so much about the sorting property but rather wanted faster *put* and *has* operations (e.g. constant time), then the Map ADT could also be implemented effectively as a Hash Table: we refer to this implementation as a **Hash Map**.

- **insert(key, value)**: perform the insertion, and return the previous *value* if overwriting, otherwise `NULL`
- **find(key)**: return the *value* associated with the *key*
- **remove(key)**: remove the *(key, value)* pair associated with *key*, and return *value* upon success or `NULL` on failure
- **hashFunction(key)**: return a hash value for *key*, which will then be used to map to an index of the backing array
- **key_equality(key1, key2)**: return true if *key1* is equal to *key2*, otherwise return false
- **size()**: return the number of *(key, value)* pairs currently stored in the Hash Map
- **isEmpty()**: return true if the Hash Map does not contain any *(key, value)* pairs, otherwise return false

Just like a Hash Table, a Hash Map uses a hash function for the purpose of being able to access the addresses of the tuples inserted. Consequently, in a Hash Map, keys must be hashable and have an associated equality test to be

able to check for uniqueness. In other words, to use a custom class type as a key, one would have to overload the hash and equality member functions.

When we find, insert, or remove $(key, (value))$ pairs in a Hash Map, we do everything *exactly* like we did with a Hash Table, but with respect to the *key*. For example, in the Hash Map insertion algorithm, because we are given a $(key, value)$ pair, we hash only the *key* but store the *key* and the *value* together. The following is an example of a Hash Map that contains multiple $(key, value)$ pairs:



When we want to find elements, we perform the exact same “find” algorithm as we did with a Hash Table, but again with respect to the *key* (which is why our “find” function only had *key* as a parameter, not *value*). Once we find the $(key, value)$ pair, we simply return the *value*. For example, in the example Hash Map above, if we want to perform the “find” algorithm on “Kammy”, we perform the regular Hash Table “find” algorithm on “Kammy”. When we find the pair that has “Kammy” as its *key*, we return the *value* (in this case, ‘A’).

Just like with finding elements, if want to remove elements from our Hash Map, we perform the Hash Table “remove” algorithm with respect to the *key* (which is why our “remove” function only had *key* as a parameter, not *value*), and once we find the $(key, value)$ pair, we simply remove the pair.

Let’s look at the pseudocode for the following Hash Map operations. Note that a Hash Map can be implemented using a Hash Table with any of the collision resolution strategies we discussed previously in this chapter. In all of the following pseudocode, the Hash Map is backed by an array `arr`, and for a $(key, value)$ pair `pair`, `pair.key` indicates the *key* of `pair` and `pair.value` indicates the *value* of `pair`.

In the `insert` operation’s pseudocode, we ignore collisions (i.e., each key maps to a unique index in the backing array) because the actual insertion algorithm would depend on which collision resolution strategy you choose to implement.

```

1  insert(key ,value):
2      index = hashFunction(key)
3      returnVal = NULL
4      if arr [index] .key == key: // should return old value
5          returnVal = arr [index] .value
6      arr [index] = <key ,value>
7      return returnVal

```

With respect to insertion, originally, in a Hash Table, if a key that was being inserted already existed, we would abort the insertion. In a Hash Map, however, attempting to insert a key that already exists will *not* abort the insertion. Instead, it will result in the original value being overwritten by the new one.

The pseudocode for the **find** operation of a Hash Map is provided. Note that this “find” algorithm returns the *value* associated with *key*, as opposed to a boolean value as it did in the Hash Table implementation.

```

1  find(key):
2      index = hashFunction(key)
3      if arr[index].key == key:
4          return arr[index].value
5      else:
6          return NULL

```

The pseudocode for the **remove** operation of a Hash Map is provided. Just like with the pseudocode for the insertion algorithm above, in the following pseudocode, we ignore collisions (i.e., each key maps to a unique index in the backing array) because the actual remove algorithm would depend on which collision resolution strategy you choose to implement.

```

1  remove(key):
2      index = hashFunction(key)
3      returnVal = NULL
4      if arr[index].key == key: // should return old value
5          returnVal = arr[index].value
6      delete arr[index]
7      return returnVal

```

Exercise Break

Which of the following statements regarding the following code are true using the previous pseudocode?

```

1  hash_map<string , string> groceryList = {
2      { "Fruit" , "Bananas" },
3      { "Cereal" , "Frosted Flakes" },
4      { "Vegetable" , "Cucumbers" },
5      { "Juice" , "Cranberry" }
6  };
7  groceryList.insert({ "Ice Cream" , "Chocolate Chip" });
8  groceryList.insert({ "Juice" , "Carrot" });

```

- `Grocery_List["Juice"]` will return "Cranberry"
- `Grocery_List["Cereal"]` will return "Frosted Flakes"
- `Grocery_List["Juice"]` will return "Carrot"
- `Grocery_List["Juice"]` will return "Carrot, Cranberry"

- The tuple containing the key "Fruit" will appear before the tuple containing the key "Ice Cream" if we were to iterate over the elements of the `unordered_map`

In practice, however, we realize that you will more often than not be using the built-in implementation of a Hash Map as opposed to implementing it from scratch, so how do we use C++'s Hash Map implementation? In C++, the implementation of a Hash Map is the `unordered_map`, and it is implemented using the Separate Chaining collision resolution strategy. Just to remind you, in C++, the implementation of a Hash Table is the `unordered_set`. Going all the way back to the initial goal of implementing a grade book system, the C++ code to use a Hash Map would be the following:

```
1 unordered_map<string, string> gradeBook = {
2     { "Kammy", "A" },
3     { "Alicia", "C" },
4     { "Anjali", "D" },
5     { "Nadah", "A" }
6 };
```

If we wanted to add a new student to our grade book, we would do the following:

```
1 gradeBook.insert({ "Bob", "B" });
```

If we wanted to check Nadah's grade in our grade book, we would do the following:

```
1 cout << gradeBook[ "Nadah" ] << endl;
```

Although we have mentioned many times that there is no particular ordering property when it comes to a Hash Map (as well as a Hash Table), we can still iterate through the inserted objects using a for-each loop like so:

```
1 for (auto student : gradeBook) {
2     auto key = student.first;
3     auto value = student.second;
4     cout << key << ":" << value << endl;
5 }
```

Note: C++ deviates slightly from the traditional Map ADT with respect to insertion. When inserting a duplicate element in the C++ `unordered_map`, the original value is not replaced. On the other hand, Python's implementation of the Map ADT (the `dict`) does in fact replace the original value in a duplicate insertion.

It is also important to note that, in practice, we often use a Hash Map to implement “one-to-many” relationships. For example, suppose we want to

implement an “office desk” system in which each desk drawer has a different label: “pens,” “pencils,” “personal papers,” “official documents,” etc. Inside each particular drawer, we expect to find office items related to the label. In the “pens” drawer, we might expect to find our favorite black fountain pen, a red pen for correcting documents, and that pen we “borrowed” from our friend months ago.

How would we use a Hash Map to implement this system? Well, the *drawer labels* would be considered the *keys*, and the *drawers* with the objects inside them would be considered the corresponding *values*. The C++ code to implement this system would be the following:

```
1 unordered_map<string, vector<OfficeSupply>> desk = {
2     { "pens", {favPen, redPen, stolenPen} },
3     { "personal papers", {personalNote} }
4 };
```

In the Hash Map above, we are using *keys* of type `string` and *values* of type `vector<OfficeSupply>` (where `OfficeSupply` is a custom class we created). Note that the *values* inserted into the Hash Map are **NOT** `OfficeSupply` objects, but `vectors` of `OfficeSupply` objects.

If we now wanted to add a printed schedule to the “personal papers” drawer of our desk, we would do the following:

```
1 desk[ "personal papers" ].push_back( schedule );
```

Note: If we wanted to calculate the worst-case time complexity of finding an office supply in our desk, we would now need to take into account the time it takes to find an element in an unsorted `vector` (which is an Array List) containing n `OfficeSupply` objects, which would be $\mathcal{O}(n)$. If we wanted to ensure constant-time access across `OfficeSupply` objects, we could also use a Hash Table instead of a `vector` (yes, we are saying that you can use `unordered_set` objects as values in your `unordered_map`).

To easily output which pens we have in our desk, we could use a for-each loop like so:

```
1 for (auto pen : desk[ "pens" ]) {
2     cout << pen << endl;
3 }
```

Exercise Break

Which of the following statements regarding the Hash Map data structure are true?

- *Values* being inserted into a Hash Map need to be hashable.

- A *key* doesn't always need to be specified when inserting into a Hash Map, as the insert method will randomly assign a key.
- Inserting a $(key, value)$ pair with a *key* that already exists in the Hash Map will cause an error, as all *keys* must be unique.
- *Keys* being inserted into a Hash Map need to be hashable.

We began this chapter with the motivation of obtaining *even faster* find, insert, and remove operations than we had seen earlier in the text, which led us to the Hash Table, a data structure with $\mathcal{O}(1)$ find, insert, and remove operations in the *average* case. In the process of learning about the Hash Table, we discussed various properties and design choices (both in the Hash Table itself as well as in hash functions for objects we may want to store) that can help ensure that we actually experience the constant-time performance on average.

We then decided we wanted even *more* than simply *storing* elements: we decided we wanted to be able to *map* objects to other objects (map *keys* to *values*, specifically), which led us to the Map ADT. Using our prior knowledge of Hash Tables, we progressed to the Hash Map, an extremely fast implementation of the Map ADT.

In practice, the Hash Table and the Hash Map are arguably two of the most useful data structures you will encounter in daily programming: the Hash Table allows us to store our data and the Hash Map allows us to easily cluster our data, both with great performance.

Chapter 6

Implementing a Lexicon

6.1 Creating a Lexicon

From the dawn of written language, mankind has made efforts to keep a record of all words in a given language. We should all hopefully be familiar with “lexicons” (*lexicon* is a fancy word for “a list of words”) and “dictionaries” (a list of words with their definitions). The oldest known dictionaries are thought to have been created in roughly 2,300 BCE. These dictionaries were Akkadian Empire cuneiform tablets with translations of words between the Sumerian and Akkadian languages. Ancient dictionaries have been found for many languages around the world, including the Chinese, Greek, Sanskrit, Hindvi, and Arabic languages.

The earliest dictionaries in the English language were glossaries of French, Spanish, or Latin words along with definitions of the foreign words in English. The English word *dictionary* was invented by John of Garland in 1220 when he wrote a book, *Dictionarius*, to help with Latin *diction*. Throughout the centuries, numerous other English dictionaries were created; however, it wasn’t until Samuel Johnson wrote *A Dictionary of the English Language* in 1755 that a reliable English dictionary was deemed to have been produced.

Now, in the 21st century, dictionaries stored as printed text are becoming less and less popular. When asked about printed dictionaries, a pre-teen cousin of one of the authors had never even used one in the entirety of his life. Instead, online dictionaries, such as those used by services like Google Search or your phone’s autocorrect feature, have become the tool of choice. These digital representations of dictionaries are typically able to perform lookups extremely quickly: after searching for a word on Google Search, its definition (and synonyms, and etymology, and statistics about its usage, etc.) are pulled up in approximately half a second.

As computer scientists, we can think of the **lexicon** as an Abstract Data Type defined by the following functions:

- **find(word)**: Find word in the lexicon

- `insert(word)`: Insert word into the lexicon
- `remove(word)`: Remove word from the lexicon

These three functions should hopefully sound annoyingly familiar by now, assuming you read the other chapters of the text! We have defined an Abstract Data Type, and we are now tasked with choosing a Data Structure to use to implement it. In this chapter, we will discuss various possible implementation approaches, focusing on their respective pros and cons.

Throughout this chapter, because languages remain largely unchanging, we will assume that “find” operations are significantly more frequent than both “insert” and “remove” operations. Also, for the same reason, we will assume that we know the size of the lexicon (i.e., the number of words we will be putting into it) before its initial construction, which is quite unlike the applications we’ve dealt with in the past.

6.2 Using Linked Lists

The first Data Structure we will discuss for implementation of a lexicon is the **Linked List**. To refresh your memory, the following are some important details regarding Linked Lists:

- We can choose to use a *Singly*-Linked List, in which nodes only have *forward* pointers and there is only one *head* pointer, or we can choose to a *Doubly*-Linked List, in which nodes have both *forward* and *reverse* pointers and there is both a *head* and a *tail* pointer
- To traverse through the elements of a Linked List, we start at one end and follow pointers until we reach our desired position in the list
- Whichever type of Linked List we choose to use, we will have a $\mathcal{O}(n)$ worst-case time complexity for “find” and “remove” operations (because we need to iterate through all n elements in the worst case to find our element of interest)
- Whichever type of Linked List we choose to use, we will have a $\mathcal{O}(1)$ worst-case time complexity to insert elements at either the front or the back of the list (assuming we have both a *head* and a *tail* pointer)
- If we want to keep our list in *alphabetical order*, whichever type of Linked List we choose to use, we will have a $\mathcal{O}(n)$ worst-case time complexity to insert elements into the list (because we need to iterate through all n elements in the worst case to find the right position to perform the actual insertion)

Now that we have reviewed the properties of the Linked List, we can begin to discuss how to actually use it to implement the three lexicon functions we

previously described. The following is pseudocode to implement the three operations of a lexicon using a Linked List. In all three of the following functions, the backing Linked List is denoted as `linkedList`.

```
1 find(word):
2     return linkedList.find(word) function

1 insertUnsorted(word):
2     linkedList.insertFront(word)

1 insertSorted(word):
2     linkedList.sortedInsert(word)

1 remove(word):
2     linkedList.remove(word)
```

STOP and Think

In `insertUnsorted(word)`, we used the `insertFront` function of our Linked List. Could we have instead used the `insertBack` function?

Exercise Break

What is the **worst-case** time complexity of the `find` function defined in the previous pseudocode?

Exercise Break

What is the **worst-case** time complexity of the `insertUnsorted` function defined in the previous pseudocode?

Exercise Break

What is the **worst-case** time complexity of the `insertSorted` function defined in the previous pseudocode?

Exercise Break

What is the **worst-case** time complexity of the `remove` function defined in the previous pseudocode?

In C++, the `list` container is implemented as a Doubly-Linked List, meaning we can create a `Lexicon` class like the following:

```

1 class Lexicon {
2     public:
3         list<string> ll;
4         bool find(string word) {
5             return find(ll.begin(), ll.end(), word) != ll.end();
6         }
7         void insert(string word) {
8             if (!(this->find(word))) {
9                 ll.push_back(word);
10            }
11        }
12        void remove(string word) {
13            ll.remove(word);
14        }
15    };

```

In Python, the `deque` collection is implemented as a Doubly-Linked List, meaning we can create a `Lexicon` class like the following:

```

1 class Lexicon:
2     ll = collections.deque()
3     def find(self, word):
4         return word in self.ll
5     def insert(self, word):
6         if not self.find(word):
7             self.ll.append(word)
8     def remove(self, word):
9         self.ll.remove(word)

```

As you might have inferred, the Linked List might not be the best choice for implementing the lexicon ADT we described. To use a Linked List, we have two implementation choices.

We can choose to keep the elements **unsorted**. If we were to do so, the worst-case time complexity of *find* and *remove* operations would be $\mathcal{O}(n)$, and that of *insertion* operation would be $\mathcal{O}(1)$. However, if we were to iterate over the elements of the list, the elements would not be in any meaningful order.

Alternatively, we can choose to keep the elements **sorted**. If we were to do so, the worst-case time complexity of the *find* and *remove* operations would still be $\mathcal{O}(n)$, but now, the insert operation would also be $\mathcal{O}(n)$. However, if we were to iterate over the elements of the list, the elements *would* be in a meaningful order: they would be in *alphabetical order*. Also, we could choose if we wanted to iterate in *ascending* alphabetical order or in *descending* alphabetical order by simply choosing the end of the Linked List, *head* or *tail*, from which we wanted to begin our iteration.

In terms of memory efficiency, a Linked List has exactly one node for each word, meaning the space complexity is $\mathcal{O}(n)$, which is as good as we can get if we want to store all n elements.

In general, we're hoping that we have instilled enough intuition in you such that you shudder slightly every time you hear $\mathcal{O}(n)$ with regard to data structures, as we have explored so many different data structures that allow us to do

better. Also, in this case specifically, recall that we explicitly mentioned that “find” was going to be our most frequently used operation by far. Therefore, even though we could potentially have $\mathcal{O}(1)$ “insert” operations, the fact that “find” operations are $\mathcal{O}(n)$ in this implementation approach should make you frustrated, or even angry, with us for wasting your precious time. In the next section, we will discuss a slightly better approach for implementing our lexicon ADT: the Array.

6.3 Using Arrays

The second Data Structure we will discuss for implementation of a lexicon is the **Array**. To refresh your memory, the following are some important details regarding Arrays:

- Because all of the slots of an Array are allocated adjacent to one another in memory, and because every slot of an Array is exactly the same size, we can find the exact memory address of any slot of the Array in $\mathcal{O}(1)$ time if we know the index (i.e., we have **random access**)
- Arrays cannot simply be “resized,” so if we were to want to insert new elements into an Array that is full, we would need to allocate a new Array of large enough size, and we would then need to copy over all elements from the old Array into the new one, which is a $\mathcal{O}(n)$ operation
- If an Array is **unsorted**, to find an element, we potentially need to iterate over all n elements, making this a $\mathcal{O}(n)$ operation in the worst case
- If an Array is **sorted**, to find an element, because of *random access*, we can perform a **Binary Search** algorithm to find the element in $\mathcal{O}(\log n)$ in the worst case

Recall that we mentioned that we will very rarely be modifying our list of words. Thus, even though the time complexity to insert or remove elements in the worst case is $\mathcal{O}(n)$, because this occurs so infrequently, this slowness is effectively negligible for our purposes. Now that we have reviewed the properties of the Array, we can begin to discuss how to actually use it to implement the three lexicon functions we previously described. The following is pseudocode to implement the three operations of a lexicon using an Array. Recall that, because “find” is our most used operation by far, we want to be able to use Binary Search to find elements, meaning we definitely want to keep our Array **sorted** and keep all elements in the Array adjacent to one another (i.e., no gaps between filled cells). In all of the following three functions, the backing Array is denoted as **array**.

```

1  find (word):
2      return array.binarySearch (word)

```

```
1 insert(word):
2     array.sortedInsert(word)
```

```
1 remove(word):
2     array.remove(word)
```

STOP and Think

By making sure we do sorted insertions, we are making our insertion algorithm slower. Why would we do this instead of just allocating extra space and always adding new elements to the end of the Array?

Exercise Break

What is the **worst-case** time complexity of the `find` function defined in the previous pseudocode?

Exercise Break

What is the **worst-case** time complexity of the `insert` function defined in the previous pseudocode?

Exercise Break

What is the **worst-case** time complexity of the `remove` function defined in the previous pseudocode?

In C++, the `vector` container is implemented as a Dynamic Array, meaning we can create a `Lexicon` class like the following:

```

1  class Lexicon {
2      public:
3          vector<string> arr;
4          bool find(string word) {
5              return binary_search(arr.begin(), arr.end(), word);
6          }
7          void insert(string word) {
8              for(int i = 0; i < arr.size(); ++i) {
9                  if(word == arr[i]) { return; }
10                 else if(word < arr[i]) {
11                     arr.insert(arr.begin() + i, word); return;
12                 }
13             }
14             arr.push_back(word);
15         }
16         void remove(string word) {
17             for(int i = 0; i < arr.size(); ++i) {
18                 if(word == arr[i]) {
19                     arr.erase(arr.begin() + i); return;
20                 }
21             }
22         }
23     };

```

In Python, the `list` collection is implemented as a Dynamic Array, meaning we can create a `Lexicon` class like the following:

```

1  class Lexicon:
2      arr = list()
3      def find(self, word):
4          i = bisect.bisect_left(self.arr, word)
5          return i < len(self.arr) and self.arr[i] == word
6      def insert(self, word):
7          if not self.find(word):
8              bisect.insort_left(self.arr, word)
9      def remove(self, word):
10         self.arr.remove(word)

```

As you can see, we improved our performance by using an Array to implement our lexicon instead of a Linked List. We wanted to be able to exploit the **Binary Search** algorithm, so we forced ourselves to keep our Array **sorted**.

By keeping the elements in our Array sorted, we are able to use Binary Search to find elements, which, as you should hopefully recall, has a worst-case time complexity of $\mathcal{O}(\log n)$. However, because we need to keep our elements sorted to be able to do Binary Search, the time complexity of inserting and removing elements is $\mathcal{O}(n)$ in the worst case. This is because, even if we do a Binary Search to find the insertion point, we might have to move over $\mathcal{O}(n)$ elements to make room for our new element in the worst case.

Also, by keeping the elements in our Array sorted, if we were to iterate over the elements of the list, the elements *would* be in a meaningful order: they would be in *alphabetical order*. Also, we could choose if we wanted to iterate in *ascending* alphabetical order or in *descending* alphabetical order by simply

choosing the end of the Array from which we wanted to begin our iteration.

In terms of memory efficiency, with Dynamic Arrays, as we grow the Array, we typically double its size, meaning at any point in time, we will have allocated at most $2n$ slots, giving this approach a space complexity of $\mathcal{O}(n)$, which is as good as we can get if we want to store all n elements.

We started off by saying that insert and remove operations are pretty uncommon in this specific application, so even though both operations are $\mathcal{O}(n)$ in this implementation, we find them acceptable. However, our motivation in this entire text is speed, so like always, we want to ask ourselves: can we go even *faster*? In the next section, we will discuss an even better approach for implementing our lexicon ADT: the Binary Search Tree.

6.4 Using Binary Search Trees

Exercise Break

The third Data Structure we will discuss for implementation of a lexicon is the **Binary Search Tree**. Given that we will be doing significantly more “find” operations than “insert” or “remove” operations, which type of a Binary Search Tree would be the optimal choice for us *in practice*?

- Regular Binary Search Tree
- Randomized Search Tree
- AVL Tree
- Red-Black Tree

In general, if we want to choose a Binary Search Tree from the ones we discussed in this text, it should be clear that the only viable contenders are the AVL Tree and the Red-Black Tree because they are guaranteed to have a $\mathcal{O}(\log n)$ *worst-case* time complexity for all three operations (whereas the Regular Binary Search Tree and Randomized Search Tree are $\mathcal{O}(n)$ in the worst case). Note that we would prefer to use an **AVL Tree** since they have a stricter balancing requirement, which translates into faster “find” operations in practice.

Now that we have refreshed our memory regarding the time complexity of self-balancing Binary Search Trees, we can begin to discuss how to actually use them to implement the three lexicon functions we previously described. The following is pseudocode to implement the three operations of a lexicon using a Binary Search Tree. Again, we would choose to use an AVL Tree because of their self-balancing properties. In all three of the following functions, the backing Binary Search Tree is denoted as `tree`.

```

1  find(word):
2      return tree.find(word)

```

```
1 insert(word):
2     tree.insert(word)
```

```
1 remove(word):
2     tree.remove(word)
```

STOP and Think

By storing our words in a Binary Search Tree, is there some efficient way for us to iterate through the words in alphabetical order?

Exercise Break

What is the **worst-case** time complexity of the `find` function defined in the previous pseudocode?

Exercise Break

What is the **worst-case** time complexity of the `insert` function defined in the previous pseudocode?

Exercise Break

What is the **worst-case** time complexity of the `remove` function defined in the previous pseudocode?

In C++, the `set` container is (typically) implemented as a Red-Black Tree. We say “typically” because the implementation of the set container depends on the specific compiler/library, but most (almost all) use Red-Black Trees. Thus, we can create a `Lexicon` class like the following:

```
1 class Lexicon {
2     public:
3         set<string> tree;
4         bool find(string word) {
5             return tree.find(word) != tree.end();
6         }
7         void insert(string word) {
8             tree.insert(word);
9         }
10        void remove(string word) {
11            tree.erase(word);
12        }
13    };
```

Unfortunately, there is no built-in Binary Search Tree implementation in Python. However, feel free to attempt to implement your own!

As you can see, we improved our performance even further by using a self-balancing Binary Search Tree to implement our lexicon. By doing so, the worst-case time complexity of finding, inserting, and removing elements is $\mathcal{O}(\log n)$. Also, because we are using a Binary Search Tree to store our words, if we were to perform an *in-order traversal* on the tree, we would iterate over the elements in a meaningful order: they would be in *alphabetical order*. Also, we could choose if we wanted to iterate in *ascending* alphabetical order or in *descending* alphabetical order by simply changing the order in which we recurse during the in-order traversal:

```

1 ascendingInOrder(node):
2     ascendingInOrder(node.leftChild)      // Recurse on left child
3     output node.word                  // Visit current node
4     ascendingInOrder(node.rightChild)    // Recurse on right child

```

```

1 descendingInOrder(node):
2     descendingInOrder(node.rightChild)   // Recurse on right child
3     output node.word                  // Visit current node
4     descendingInOrder(node.leftChild)    // Recurse on left child

```

In terms of memory efficiency, a Binary Search Tree has exactly one node for each word, meaning the space complexity is $\mathcal{O}(n)$, which is as good as we can get if we want to store all n elements.

Of course, yet again, it is impossible to satisfy a computer scientist. So, like always, we want to ask ourselves: can we go even *faster*? Note that, up until now, every approach we described had a time complexity that depended on n , the number of elements in the data structure. In other words, as we insert more and more words into our data structure, the three operations we described take longer and longer. In the next section, we will discuss another good approach for implementing our lexicon: the Hash Table (as well as the Hash Map to implement a dictionary).

6.5 Using Hash Tables and Hash Maps

The fourth Data Structure we will discuss for implementation of a lexicon is the **Hash Table**. To refresh your memory, the following are some important details regarding Hash Tables and Hash Maps that we can use to implement a lexicon:

- To insert an element into a Hash Table, we use a **hash function** to compute the **hash value** (an integer) representing the element. We then mod the hash value by the size of the Hash Table's backing array to get an index for our element
- Ignoring the time it takes to compute hash values, the *average*-case time complexity of finding, inserting, and removing elements in a well-implemented Hash Table is $\mathcal{O}(1)$

- The time complexity to compute the hash value of a string of length k (assuming our hash function is good) is $\mathcal{O}(k)$
- A **Hash Map** is effectively just an extension of a Hash Table, where we do all of the Hash Table operations on the *key*, but when we actually perform the insertion, we insert the $(key, value)$ pair. For our purposes of making a lexicon, it could make sense to have *keys* be words and *values* be definitions
- It is impossible for us to iterate through the elements of an *arbitrary* Hash Table in a meaningful order
- For our purposes, we will assume collisions are resolved using Separate Chaining using a Linked List (the standard)

Now that we have reviewed the properties of the Hash Table, we can begin to discuss how to actually use it to implement the three lexicon functions we previously described. The following is pseudocode to implement the three operations of a lexicon using a Hash Table. In all three of the following functions, the backing Hash Table is denoted as `hashTable`.

```

1 find(word):
2     return hashTable.find(word)


---


1 insert(word):
2     hashTable.insert(word)


---


1 remove(word):
2     hashTable.remove(word)

```

STOP and Think

Does our choice in collision resolution strategy have any effect on the performance of our lexicon, ignoring the time it takes to compute an element's hash value?

Exercise Break

What is the **worst-case** time complexity of the `find` function defined in the previous pseudocode, ignoring the time it takes to compute an element's hash value?

Exercise Break

What is the **average-case** time complexity of the `find` function defined in the previous pseudocode, ignoring the time it takes to compute an element's hash value?

Exercise Break

What is the **worst-case** time complexity of the `insert` function defined in the previous pseudocode, ignoring the time it takes to compute an element's hash value?

Exercise Break

What is the **average-case** time complexity of the `insert` function defined in the previous pseudocode, ignoring the time it takes to compute an element's hash value?

Exercise Break

What is the **worst-case** time complexity of the `remove` function defined in the previous pseudocode, ignoring the time it takes to compute an element's hash value?

Exercise Break

What is the **average-case** time complexity of the `remove` function defined in the previous pseudocode, ignoring the time it takes to compute an element's hash value?

In C++, the `unordered_set` container is implemented as a Hash Table, meaning we can create a `Lexicon` class like the following:

```

1  class Lexicon {
2  public:
3      unordered_set<string> hashTable;
4      bool find(string word) {
5          return hashTable.find(word) != hashTable.end();
6      }
7      void insert(string word) {
8          hashTable.insert(word);
9      }
10     void remove(string word) {
11         hashTable.erase(word);
12     }
13 };

```

In Python, the `set` collection is implemented as a Hash Table, meaning we can create a `Lexicon` class like the following:

```

1 class Lexicon:
2     hashTable = set()
3     def find(self, word):
4         return word in self.hashTable
5     def insert(self, word):
6         self.hashTable.add(word)
7     def remove(self, word):
8         self.hashTable.remove(word)

```

Recall that the only difference between a *lexicon* and a *dictionary* is that a *lexicon* is just a list of words, whereas a *dictionary* is a list of words *with their definitions*. In C++, the `unordered_map` container is implemented as a Hash Map, meaning we can create a `Dictionary` class like the following:

```

1 class Dictionary {
2     public:
3         unordered_map<string, string> hashMap;
4         string find(string word) {
5             return hashMap[word];
6         }
7         void insert(string word, string def) {
8             hashMap[word] = def;
9         }
10        void remove(string word) {
11            hashMap.erase(word);
12        }
13    };

```

In Python, the `dict` collection is implemented as a Hash Map, meaning we can create a `Dictionary` class like the following:

```

1 class Dictionary:
2     hashMap = dict()
3     def find(self, word):
4         return self.hashMap[word]
5     def insert(self, word, definition):
6         self.hashMap[word] = definition
7     def remove(self, word):
8         del self.hashMap[word]

```

As you can see, we further improved our *average-case* performance by using a Hash Table to implement our lexicon. We have to perform a $\mathcal{O}(k)$ operation on each word in our lexicon (where k is the length of the word) in order to compute the word's hash value, so we say that our average-case time complexity is $\mathcal{O}(1)$ if we don't take into account the hash function, otherwise $\mathcal{O}(k)$.

However, keep in mind that the order in which elements are stored in a Hash Table's backing array does not necessarily have any meaning. In other words, it is impossible for us to iterate through the elements of an *arbitrary* Hash Table in a meaningful (e.g. alphabetical) order.

In terms of memory efficiency, recall that, with a Hash Table, we try to maintain the **load factor** (i.e., the percentage full the backing array is at any

given moment) relatively low. Specifically, 70% is typically a good number. Formally, because we will have allocated approximately $m = \frac{n}{0.7}$ slots to store n elements and because $\frac{1}{0.7}$ is a constant, our space complexity is $\mathcal{O}(n)$. However, even though the space usage scales linearly, note that we will always be wasting approximately 30% of the space we allocate, which can be a pretty big cost as our dataset gets large (30% of a big number is a smaller, but still big, number).

Also, note that, by simply using a Hash Map instead of a Hash Table, we were able to create a *dictionary* that stores *(word, definition)* pairs as opposed to a *lexicon* that only stores *words*. As a result, we have the added functionality of being able to find a word’s definition in addition to simply seeing if a word exists, but without worsening our performance. It should be noted that we could have theoretically made somewhat trivial modifications to any of the previous (and to any of the upcoming) implementation strategies to store a *dictionary* instead of a *lexicon* (i.e., simply store *(word, definition)* pairs in our data structure and performing all searching algorithms using just the *word*), but we would have to overload various C++ specific functions—such as the comparison function—to be able to use existing C++ data structure implementations.

Because we have to compute the hash values of our strings, we can safely say that the time complexity of this implementation is $\mathcal{O}(k)$ in the *average* case. Is there some way to make this $\mathcal{O}(k)$ time complexity a *worst-case* time complexity? Also, can we somehow gain the ability to iterate through our elements in alphabetical order? In the next section, we will discuss the Multiway Trie, a data structure that allows us to obtain both.

6.6 Using Multiway Tries

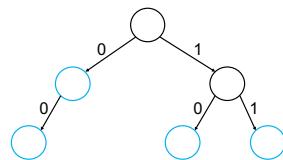
With regard to implementing our three lexicon functions, we’ve figured out how to obtain a $\mathcal{O}(\log n)$ *worst-case* time complexity—where n is the number of words in the lexicon—by using a balanced Binary Search Tree. We’ve also figured out how to obtain a $\mathcal{O}(k)$ *average-case* time complexity—where k is the length of the longest word in the lexicon—by using a Hash Table ($\mathcal{O}(k)$) to compute the hash value of a word, and then $\mathcal{O}(1)$ to actually perform the operation). Can we do even better?

In this section, we will discuss the **Multiway Trie**, a data structure designed for the exact purpose of storing a set of words. It was first described by R. de la Briandais in 1959, but the term *trie* was coined two years later by Edward Fredkin, originating from the word *retrieval* (as they were used to *retrieve* words). As such, *trie* was originally pronounced “tree” (as in the middle syllable of *retrieval*), but it is now typically pronounced “try” in order to avoid confusion with the word *tree*.

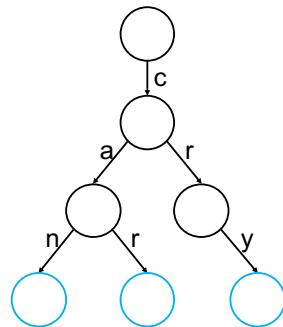
The **Trie** is a tree structure in which the elements that are being stored are *not* represented by the value of a single node. Instead, elements stored in a trie are denoted by the concatenation of the labels on the path from the root to the node representing the corresponding element. Nodes that represent keys are labeled as “word nodes” in some way (for our purposes, we will color them

blue). In this section, the tries that we will discuss will have **edge labels**, and it is the concatenation of these edge labels that spell out the words we will store.

The following is an example of a trie. Specifically, it is an example of a **Binary Trie**: a trie that is a binary tree. The words stored in the following trie are denoted by the prefixes leading up to all blue nodes: 0, 00, 10, and 11. Note that 01 is not in this trie because there is no valid path from the root labeled by 01. Also note that 1 is not in this trie because, although there is a valid path from the root labeled by 1, the node that we end up at is not a “word node.”

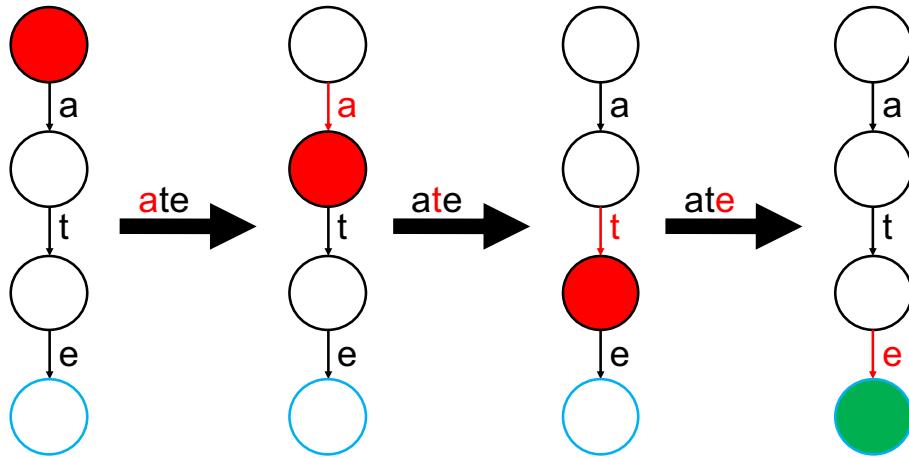


Of course, as humans, we don’t speak in binary, so it would be more helpful for our data structure to be able to represent words in our own alphabet. Instead of having only two edges coming out of each node (labeled with either a 1 or a 0), we can expand our alphabet to any alphabet we choose! A trie in which nodes can have more than two edges are known as Multiway Tries (MWT). For our purposes, we will use Σ to denote our alphabet. For example, for binary, $\Sigma = \{0, 1\}$. For the DNA alphabet, $\Sigma = \{A, C, G, T\}$. For the English alphabet, $\Sigma = \{a, \dots, z\}$. The following is a Multiway Trie with $\Sigma = \{a, \dots, z\}$ containing the following words: *can*, *car*, and *cry*.

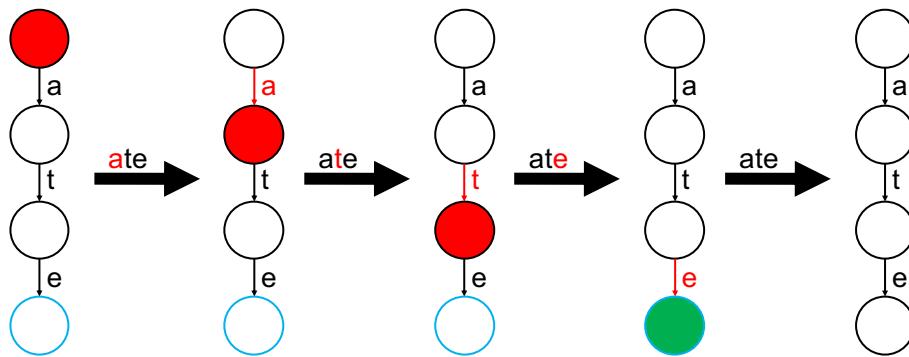


The algorithms behind the insert, remove, and find operations of a Multiway Trie are actually very simple. To **find** a word, simply start at the root and, for each letter of the word, follow the corresponding edge of the current node. If the edge you need to traverse does not exist, the word does not exist in the trie. Also, even if you are able to traverse all of the required edges, if the node you land on is not labeled as a “word node,” the word does not exist in the trie. A word *only* exists in the trie if you are able to traverse all of the required edges and the node you reach at the end of the traversal is labeled as a “word node.”

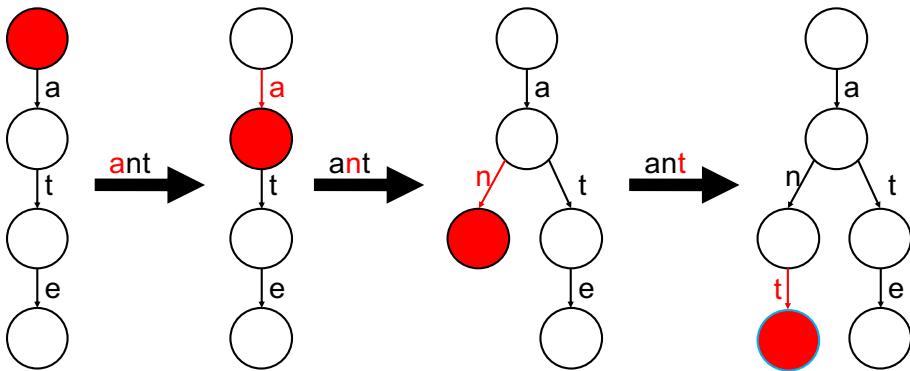
The following is a simple example, where $\Sigma = \{a, \dots, z\}$, on which we perform the find operation on the word *ate*. Note that, even though there exists a valid path from the root to the word *at* (which is a prefix of *ate*), the word *at* does not appear in our Multiway Trie because the node we reach after following the path labeled by *at* is not labeled as a “word node.”



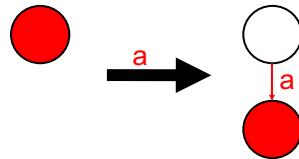
To **remove** a word, simply follow the find algorithm. If the find algorithm fails, the word does not exist in the trie, meaning nothing has to be done. If the find algorithm succeeds, simply remove the “word node” label from the node at which the find algorithm terminates. In the following example, we remove the word *ate* from the trie above by simply traversing the path spelled by *ate* and removing the “word node” label on the final node on the path:



To **insert** a word, simply attempt to follow the find algorithm. If, at any point, the edge we need to traverse does not exist, simply create the edge (and node to which it should point), and continue. In the following example, we add the word *ant* to the given trie:

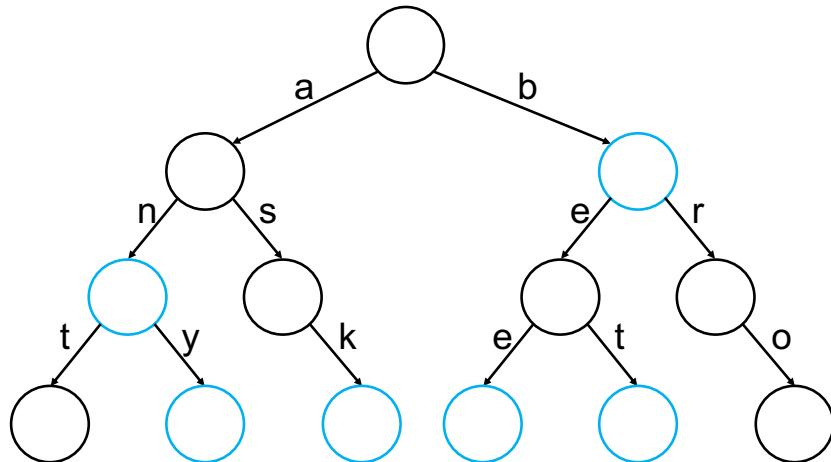


We want to emphasize the fact that, in a Multiway Trie, letters label *edges* of the trie, *not* nodes! Although this fact is quite clear given the diagrams, students often make mistakes when implementing Multiway Tries in practice. Specifically, note that an “empty” Multiway Trie (i.e., a Multiway Trie with no keys) is a **single-node tree with no edges**. Then, if I were to insert even a single-letter key (*a* in the following example), I would create a *second* node, and *a* would label the edge connecting my root node and this new node:



Exercise Break

List all of the words are contained in the following Multiway Trie.



The following is formal pseudocode for the three operations of a Multiway Trie that we mentioned previously. In the pseudocode, we denote the Multiway Trie's root node as the variable `root`.

```

1 find(word):
2     curr = root
3     for each character c in word:
4         if curr does not have an outgoing edge labeled by c:
5             return False
6         else:
7             curr = child of curr along edge labeled by c
8         if curr is a word-node:
9             return True
10        else:
11            return False

```

```

1 remove(word):
2     curr = root
3     for each character c in word:
4         if curr does not have an outgoing edge labeled by c:
5             return
6         else:
7             curr = child of curr along edge labeled by c
8         if curr is a word-node:
9             remove the word-node label of curr

```

```

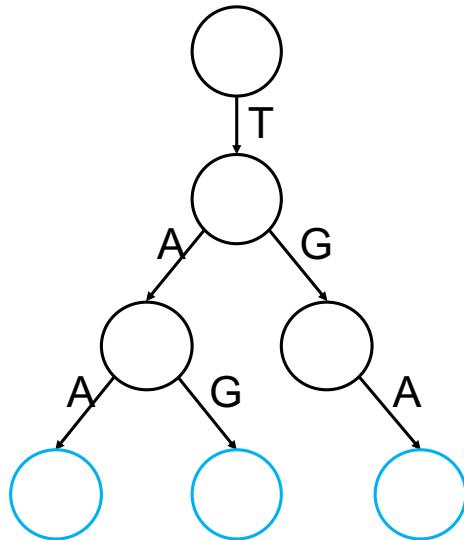
1 insert(word):
2     curr = root
3     for each character c in word:
4         if curr does not have an outgoing edge labeled by c:
5             create a new child of curr with the edge labeled by c
6             curr = child of curr along edge labeled by c
7         if curr is not a word-node:
8             label curr as a word-node

```

Exercise Break

Assume that, if you are given a node u and a character c , you can access the outgoing edge of u labeled by c (or determine that no such edge exists) in $\mathcal{O}(1)$ time. Given this assumption, what is the worst-case time complexity of the find, insert, and remove algorithms described previously with respect to n , the number of nodes in the trie, and k , the length of the longest word in the trie?

Abstractly, when we draw Multiway Tries, for any given node, we only depict the outgoing edges that we actually see, and we completely omit any edges that we don't observe. For example, the following is an example of a Multiway Trie with $\Sigma = \{A, C, G, T\}$ which stores the words TGA, TAA, and TAG (the three STOP codons of protein translation as they appear in DNA, for those who are interested):



Based on this representation, intuitively, we might think that each node object should have a list of edges. Recall that our motivation for discussing a Multiway Trie was to achieve a worst-case time complexity of $\mathcal{O}(k)$, where k is the length of the longest word, to find, insert, and remove words. This means that each individual edge traversal should be $\mathcal{O}(1)$ (and we do k $\mathcal{O}(1)$ edge traversals, one for each letter of our word, resulting in a $\mathcal{O}(k)$ time complexity overall). However, if we were to store the edges as a list, we would unfortunately have to perform a $\mathcal{O}(|\Sigma|)$ search operation to find a given edge, where $|\Sigma|$ is the size of our alphabet.

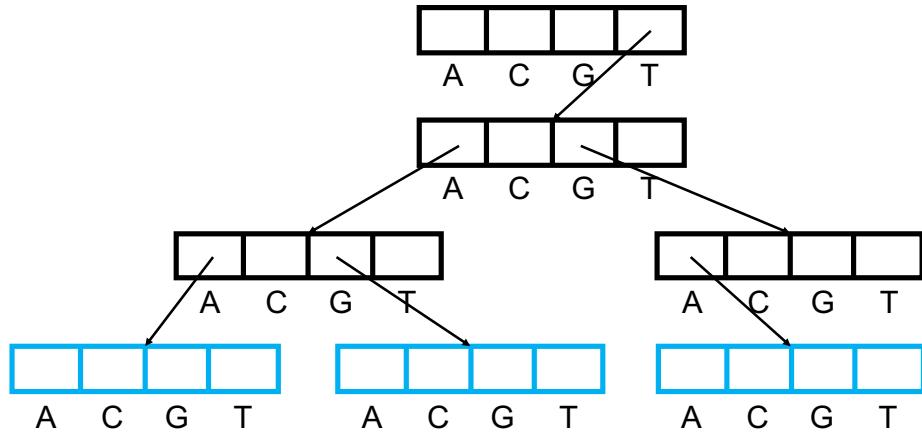
To combat this and maintain the $\mathcal{O}(1)$ edge traversals, we instead allocate space for *all* of the edges that can come out of a given node right off the bat in the form of an array. We fill in the slots for edges that we use, and we leave the slots for unused edges empty. This way, if we're at some node u and if we're given some character c , we can find the edge coming out of u that is labeled by c in $\mathcal{O}(1)$ time by simply going to the corresponding slot of the array of edges. This $\mathcal{O}(1)$ time complexity to find an edge given a character c assumes that we have a way of mapping c to an index in our array of edges in $\mathcal{O}(1)$ time, which is a safe assumption. For example, if $\Sigma = \{a, \dots, z\}$, in C++, we could do something like this:

```

1 // map 'a' to 0, 'b' to 1, ..., 'z' to 25 in O(1) time
2 int index(char c) {
3     return (int)c - (int)'a';
4 }
  
```

The following is a diagram representing the same Multiway Trie as above, but in a fashion more similar to the actual implementation. Note that, in the example, just like before, we are using the DNA alphabet for the sake of

simplicity (i.e., $\Sigma = \{A, C, G, T\}$), not the English alphabet.



Of course, the more implementation-like representation of Multiway Tries is a bit harder to read, which is why we draw them the more abstract way (as opposed to a bunch of arrays of edges). We will continue to draw them using the abstract way, but be sure to never forget what they actually look like behind-the-scenes.

Because of the clean structure of a Multiway Trie, we can iterate through the elements in the trie in sorted order by performing a *pre-order traversal* on the trie. The following is the pseudocode to recursively output all words in a Multiway Trie in ascending or descending alphabetical order (we would call either function on the root):

```

1  ascendingPreOrder(node):
2      if node is a word-node:
3          output the word labeled by path from root to node
4      for each child of node (in ascending order):
5          ascendingPreOrder(child)

```

```

1  descendingPreOrder(node):
2      if node is a word-node:
3          output the word labeled by path from root to node
4      for each child of node (in descending order):
5          descendingPreOrder(child)

```

We can use this recursive pre-order traversal technique to provide another useful function to our Multiway Trie: auto-complete. So, if we were given a prefix and we wanted to output *all* the words in our Multiway Trie that start with this prefix, we can traverse down the trie along the path labeled by the prefix, and we can then call the recursive pre-order traversal function on the node we reached.

Unfortunately, C++ does not natively have a Multiway Trie implementation

for us to use in our Lexicon class implementation. However, it is fairly simple to implement, and you can practice doing so in the accompanying online Stepik text! Nevertheless, imagine we have the following Multiway Trie implementation:

```

1 class MultiwayTrie {
2     public:
3         bool find(string word);
4         void insert(string word);
5         void remove(string word);
6 };

```

We could then create a Lexicon class like the following:

```

1 class Lexicon {
2     public:
3         MultiwayTrie mwt;
4         bool find(string word) {
5             return mwt.find(word);
6         }
7         void insert(string word) {
8             mwt.insert(word);
9         }
10        void remove(string word) {
11            mwt.remove(word);
12        }
13 };

```

The equivalent Python classes are the following:

```

1 class MultiwayTrie:
2     def find(word):
3         ...
4     def insert(word):
5         ...
6     def remove(word):
7         ...

1 class Lexicon:
2     mwt = MultiwayTrie()
3     def find(word):
4         return self.mwt.find(word)
5     def insert(word):
6         return self.mwt.insert(word)
7     def remove(word):
8         return self.mwt.remove(word)

```

We previously discussed using a Hash Table to implement a lexicon, which would allow us to perform find, insert, and remove operations with an *average-case* time complexity of $\mathcal{O}(k)$, assuming we take into account the time it takes to compute the hash value of a string of length k (where k is the length of

the longest word in the dictionary). However, with this approach, it would be impossible for us to iterate through the words in the lexicon in any meaningful order.

In this section, we have now discussed using a Multiway Trie to implement a lexicon, which allows us to perform find, insert, and remove operations with a *worst-case* time complexity of $\mathcal{O}(k)$. Also, because of the structure of a Multiway Trie, we can easily and efficiently iterate through the words in the lexicon in alphabetical order (either ascending or descending order) by performing a *pre-order traversal*. We can also use this exact pre-order traversal technique to create *auto-complete* functionality for our lexicon.

However, Multiway Tries are *extremely* inefficient in terms of space usage: in order to have fast access to edges coming out of a given node u , we need to allocate space for each of the $|\Sigma|$ possible edges that could theoretically come out of u . If our trie is relatively dense, this space usage is tolerable, but if our trie is relatively sparse, there will be a *lot* of space wasted in terms of empty space for possible edges. In the next section, we will discuss a data structure that lets us obtain a time efficiency close to that of a Multiway Trie, but without the terrible space requirement: the Ternary Search Tree.

6.7 Using Ternary Search Trees

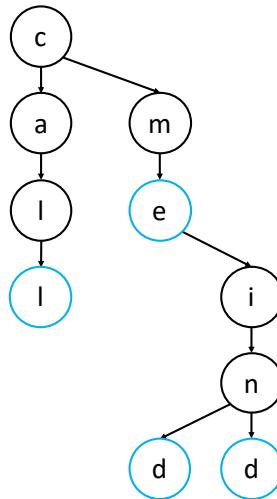
Thus far, we have discussed implementing a lexicon using a few different data structures and we have discussed their respective pros and cons. Recall that the “best” data structures we discussed for this task were the following, where n is the number of words in our lexicon and k is the length of the longest word:

- A *balanced Binary Search Tree* (such as an *AVL Tree* or a *Red-Black Tree*), which is the most space-efficient we can get, but has a *worst-case* time complexity of $\mathcal{O}(\log n)$. A Binary Search Tree has the added benefit of being able to iterate over the elements of the lexicon in alphabetical order
- A *Hash Table*, which is not quite as space-efficient as a Binary Search Tree (but not too bad), and which has an *average-case* time complexity of $\mathcal{O}(k)$ (when we take into account the time it takes to compute a hash value of a string of length k) and a *worst-case* time complexity of $\mathcal{O}(n)$. Unfortunately, the elements of a Hash Table are unordered, so there is no clean way of iterating through the elements of our lexicon in a meaningful order
- A *Multiway Trie*, which is the most *time*-efficient we can get in the worst case, $\mathcal{O}(k)$, but which is extremely inefficient *memory*-wise. A Multiway Trie has the added benefit of being able to iterate over the elements of the lexicon in alphabetical order as well as the ability to perform auto-complete by performing a simple pre-order traversal

In this section, we will discuss the **Ternary Search Tree**, which is a data structure that serves as a middle-ground between the Binary Search Tree and the Multiway Trie. The Ternary Search Tree is a type of trie, structured in a fashion similar to Binary Search Trees, that was first described in 1979 by Jon Bentley and James Saxe.

The **Trie** is a tree structure in which the elements that are being stored are *not* represented by the value of a single node. Instead, elements stored in a trie are denoted by the concatenation of the labels on the path from the root to the node representing the corresponding element. The Ternary Search Tree (TST) is a type of trie in which nodes are arranged in a manner similar to a Binary Search Tree, but with up to three children rather than the binary tree's limit of two.

Each node of a Ternary Search Tree stores a single character from our alphabet Σ and can have three children: a *middle child*, *left child*, and *right child*. Furthermore, just like in a Multiway Trie, nodes that represent keys are labeled as “word nodes” in some way (for our purposes, we will color them blue). Just like in a Binary Search Tree, for every node u , the *left child* of u must have a value *less* than u , and the *right child* of u must have a value *greater* than u . The *middle child* of u represents the next character in the current word. The following is an example of a Ternary Search Tree that contains the words *call*, *me*, *mind*, and *mid*:



If it is unclear to you *how* this example stores the words we listed above, as well as how to go about finding an arbitrary query word, that is perfectly fine. It will hopefully become more clear as we work through more examples together.

In a Multiway Trie, a word was defined as the concatenation of edge labels along the path from the root to a “word node.” In a Ternary Search Tree, the definition of a word is a bit more complicated: For a given “word node,” define

the path from the root to the “word node” as *path*, and define *S* as the set of all nodes in *path* that have a middle child also in *path*. The word represented by the “word node” is defined as the concatenation of the labels of each node in *S*, along with the label of the “word node” itself.

To **find** a word *key*, we start our tree traversal at the root of the Ternary Search Tree. Let’s denote the current node as *node* and the current letter of *key* as *letter*:

- If *letter* is less than *node*’s label: If *node* has a left child, traverse down to *node*’s left child. Otherwise, we have failed (*key* does not exist in this Ternary Search Tree)
- If *letter* is greater than *node*’s label: If *node* has a right child, traverse down to *node*’s right child. Otherwise, we have failed (*key* does not exist in this Ternary Search Tree)
- If *letter* is equal to *node*’s label: If *letter* is the last *letter* of *key* and if *node* is labeled as a “word node,” we have successfully found *key* in our Ternary Search Tree; if not, we have failed. Otherwise, if *node* has a middle child, traverse down to *node*’s middle child and set *letter* to the next character of *key*; if not, we have failed (*key* does not exist in this Ternary Search Tree)

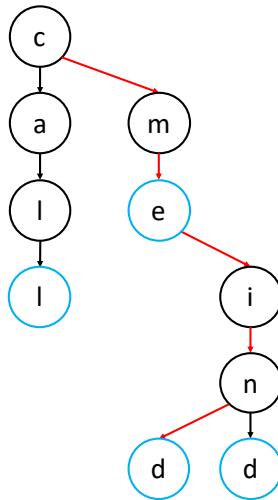
The following is formal pseudocode for the find algorithm of the Ternary Search Tree:

```

1  find(key):
2      node = root node of the TST; let = first letter of key
3      loop infinitely:
4          if let < node.label:
5              if node has left child:
6                  node = node.leftChild
7              else:
8                  return False
9              else if let > node.label:
10                 if node has right child:
11                     node = node.rightChild
12                 else:
13                     return False
14             else:
15                 if let is last letter of key and node is word-node:
16                     return True
17                 else:
18                     if node has middle child:
19                         node = node.midChild
20                         let = next letter of key
21                     else:
22                         return False

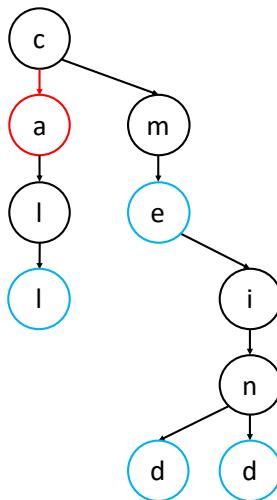
```

The following is the same example from before, and we will step through the process of finding the word *mid*:



1. We start with *node* as the root node ('c') and *letter* as the first letter of *mid* ('m')
2. *letter* ('m') is greater than the label of *node* ('c'), so set *node* to the right child of *node* ('m')
3. *letter* ('m') is equal to the label of *node* ('m'), so set *node* to the middle child of *node* ('e') and set *letter* to the next letter of *mid* ('i')
4. *letter* ('i') is greater than the label of *node* ('e'), so set *node* to the right child of *node* ('i')
5. *letter* ('i') is equal to the label of *node* ('i'), so set *node* to the middle child of *node* ('n') and set *letter* to the next letter of *mid* ('d')
6. *letter* ('d') is less than the label of *node* ('n'), so set *node* to the left child of *node* ('d')
7. *letter* ('d') is equal to the label of *node* ('d'), *letter* is already on the last letter of *mid* ('d'), and *node* is a "word node," so **success!**

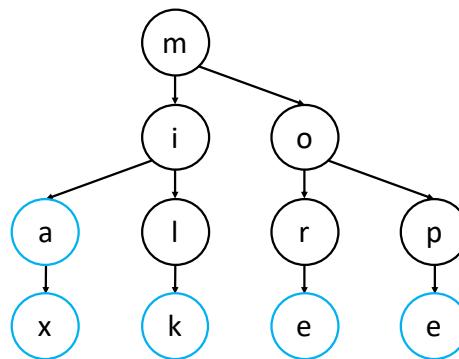
Using the same example as before, let's try finding the word *cme*, which might *seem* like it exists, but it actually doesn't:



- We start with *node* as the root node ('c') and *letter* as the first letter of *cme* ('c')
- *letter* ('c') is equal to the label of *node* ('c'), so set *node* to the middle child of *node* ('a') and set *letter* to the next letter of *cme* ('m')
- *letter* ('m') is greater than the label of *node* ('a'), but *node* does not have a right child, so we **failed**

Exercise Break

Which of the following words appear in the following Ternary Search Tree? *ma*, *max*, *mia*, *milk*, *mope*, *more*, *ore*, *pe*



The **remove** algorithm is extremely trivial once you understand the find algorithm. To remove a word *key* from a Ternary Search Tree, simply perform the find algorithm. If you successfully find *key*, simply remove the “word node”

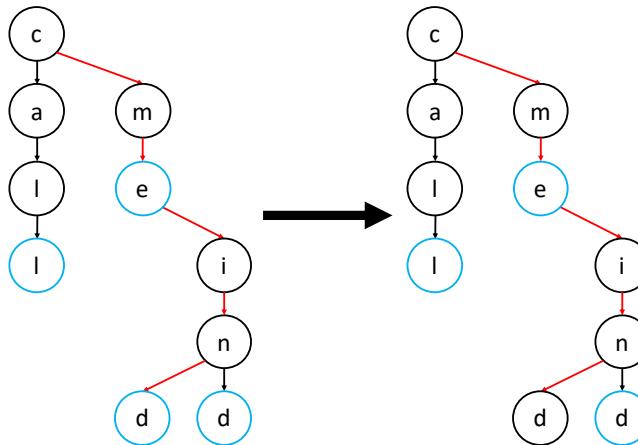
label from the node at which you end up. The following is formal pseudocode for the remove algorithm of the Ternary Search Tree:

```

1  remove(key):
2      node = root node of the TST; let = first letter of key
3      loop infinitely:
4          if let < node.label:
5              if node has left child:
6                  node = node.leftChild
7              else:
8                  return
9          else if let > node.label:
10             if node has right child:
11                 node = node.rightChild
12             else:
13                 return
14         else:
15             if let is last letter of key and node is word-node:
16                 remove word-node label from node; return
17             else:
18                 if node has middle child:
19                     node = node.midChild
20                     let = next letter of key
21                 else:
22                     return

```

The following is the initial example of a Ternary Search Tree, and we will demonstrate the process of removing the word mid:



1. We start with *node* as the root node ('c') and *letter* as the first letter of *mid* ('m')
2. *letter* ('m') is greater than the label of *node* ('c'), so set *node* to the right child of *node* ('m')
3. *letter* ('m') is equal to the label of *node* ('m'), so set *node* to the middle

child of *node* ('e') and set *letter* to the next letter of *mid* ('i')

4. *letter* ('i') is greater than the label of *node* ('e'), so set *node* to the right child of *node* ('i')
5. *letter* ('i') is equal to the label of *node* ('i'), so set *node* to the middle child of *node* ('n') and set *letter* to the next letter of *mid* ('d')
6. *letter* ('d') is less than the label of *node* ('n'), so set *node* to the left child of *node* ('d')
7. *letter* ('d') is equal to the label of *node* ('d'), *letter* is already on the last letter of *mid*, and *node* is a “word node,” so *mid* exists in the tree!
8. Remove the “word node” label from *node*

The **insert** algorithm also isn't too bad once you understand the find algorithm. To insert a word key into a Ternary Search Tree, perform the find algorithm:

- If you're able to legally traverse through the tree for every letter of *key* (which implies *key* is a prefix of another word in the tree), simply label the node at which you end up as a “word node”
- If you are performing the tree traversal and run into a case where you want to traverse left or right, but no such child exists, create a new left/right child labeled by the current letter of *key*, and then create middle children labeled by each of the remaining letters of *key*
- If you run into a case where you want to traverse down to a middle child, but no such child exists, simply create middle children labeled by each of the remaining letters of *key*

Note that, for the same reasons insertion order affected the shape of a Binary Search Tree, the order in which we insert keys into a Ternary Search Tree affects the shape of the tree. For example, just like in a Binary Search Tree, the root node is determined by the first element inserted into a Ternary Search Tree. The following is formal pseudocode for the insert algorithm of the Ternary Search Tree:

```

1  insert(key): // insert key into this TST
2      node = root node of the TST; let = first letter of key
3
4      loop infinitely:
5          if let < node.label:
6              if node has left child:
7                  node = node.leftChild
8              else:
9                  node.leftChild = new node labeled by let
10                 node = node.leftChild
11                 iterate let over remaining letters of key:
12                     node.midChild = new node labeled by let
13                     node = node.midChild
14                     label node as word-node
15             else if let > node.label:
16                 if node has right child:
17                     node = node.rightChild
18                 else:
19                     node.rightChild = new node labeled by let
20                     node = node.rightChild
21                     iterate let over remaining letters of key:
22                         node.midChild = new node labeled by let
23                         node = node.midChild
24                         label node as word-node
25             else:
26                 if let is the last letter of key:
27                     label node as word-node
28                 else:
29                     if node has a middle child:
30                         node = node.midChild
31                         let = next letter of key
32                     else:
33                         iterate let over remaining letters of key:
34                             node.midChild = new node labeled by let
35                             node = node.midChild
36                             label node as word-node

```

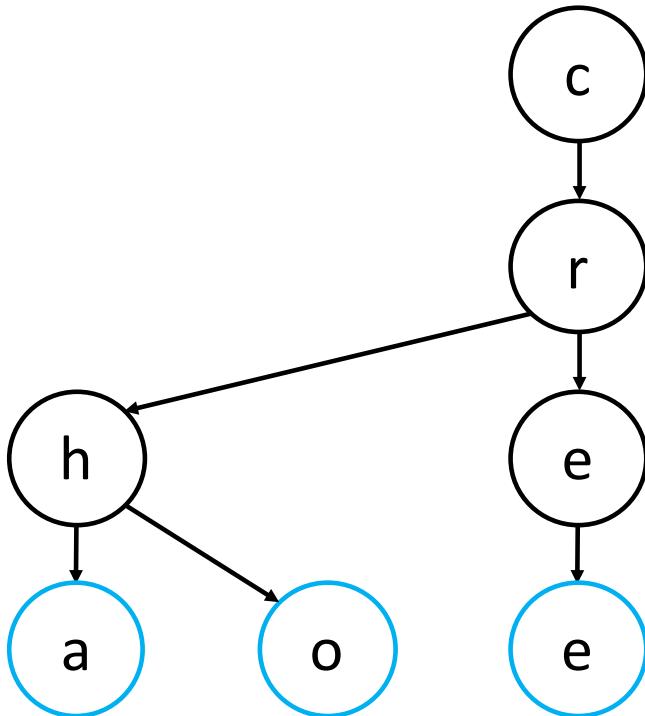
The following is the initial example of a Ternary Search Tree, and we will demonstrate the process of inserting the word *cabs*:

1. We start with *node* as the root node ('c') and *letter* as the first letter of *cabs* ('c')
2. *letter* ('c') is equal to the label of *node* ('c'), so set *node* to the middle child of *node* ('a') and set *letter* to the next letter of *cabs* ('a')
3. *letter* ('a') is equal to the label of *node* ('a'), so set *node* to the middle child of *node* ('l') and set *letter* to the next letter of *cabs* ('b')
4. *letter* ('b') is less than the label of *node* ('l'), but *node* does not have a left child:
5. Create a new node as the left child of *node*, and label the new node with *letter* ('b')

6. Set *node* to the left child of *node* ('b') and set *letter* to the next letter of *cabs* ('s')
7. Create a new node as the middle child of *node* ('s'), and label the new node with *letter* ('s')
8. Set *node* to the middle child of *node* ('s')
9. *letter* is already on the last letter of *cabs*, so label *node* as a “word node” and we’re done!

Exercise Break

The following Ternary Search Tree contains the words *cha*, *co*, and *cree*. In what order must they have been inserted?



Saying that the structure of a Ternary Search Tree is “clean” is a bit of a stretch, especially in comparison to the structure of a Multiway Trie, but the structure is ordered nonetheless because of the Binary Search Tree property that a Ternary Search Tree maintains. As a result, just like in a Multiway Trie, we can iterate through the elements of a Ternary Search Tree in sorted order by performing a in-order traversal on the trie. The following is the pseudocode to recursively output all words in a Ternary Search Tree in ascending or descending

alphabetical order (we would call either function on the root):

```

1 ascendingInOrder(node):
2     for left child of node (in ascending order):
3         ascendingInOrder(child)
4     if node is a word-node:
5         output the word labeled by path from root to node
6         for middle child of node (in ascending order):
7             ascendingInOrder(child)
8     for right child of node (in ascending order):
9         ascendingInOrder(child)

```

```

1 descendingInOrder(node):
2     for right child of node (in descending order):
3         descendingInOrder(child)
4         if node is a word-node:
5             output the word labeled by path from root to node
6     for middle child of node (in descending order):
7         descendingInOrder(child)
8     for left child of node (in descending order):
9         descendingInOrder(child)

```

We can use this recursive in-order traversal technique to provide another useful function to our Ternary Search Tree: auto-complete. If we were given a prefix and we wanted to output *all* words in our Ternary Search Tree that start with this prefix, we can traverse down the trie along the path labeled by the prefix, and we can then call the recursive in-order traversal function on the node we reached.

We have now discussed yet another data structure that can be used to implement a lexicon: the Ternary Search Tree. Because of the Binary Search Tree properties of the Ternary Search Tree, the *average*-case time complexity to find, insert, and remove elements is $\mathcal{O}(\log n)$ and the *worst*-case time complexity is $\mathcal{O}(n)$. Also, because inserting elements in a Ternary Search Tree is very similar to inserting elements in a Binary Search Tree, the structure of a Ternary Search Tree (i.e., the balance), and as a result, the performance of a Ternary Search Tree, largely depends on the order in which we insert elements. As a result, if the words we will be inserting are known in advance, it is common practice to randomly shuffle the words before inserting them to help improve the balance of the tree.

Because of the structure of a Ternary Search Tree, we can easily and efficiently iterate through the words in the lexicon in alphabetical order (either ascending or descending order) by performing a *in-order traversal*. We can also use this exact in-order traversal technique to create *auto-complete* functionality for our lexicon.

Ternary Search Trees are a bit slower than Multiway Tries, but they are *significantly* more space-efficient, and as a result, they are often chosen as the data structure of choice when people implement lexicons. In short, Ternary Search Trees give us a nice middle-ground between the *time*-efficiency of a Multiway Trie and the *space*-efficiency of a Binary Search Tree.

Chapter 7

Coding and Compression

7.1 Return of the (Coding) Trees

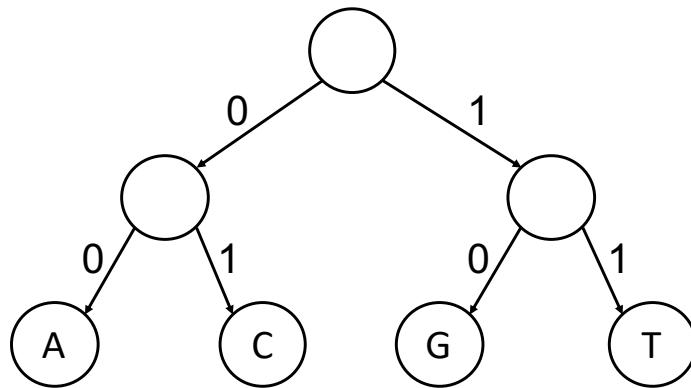
As we hinted at in a past discussion about Alan Turing and his work cracking Enigma, there is great use in being able to encode messages. Whether the goal is to encrypt secret messages to prevent unwanted eyes from viewing the content, or to compress files on a computer to reduce its filesize, it is intuitive that there are two key steps to encryption:

- Original Message → Coded Message (**Encryption**)
- Coded Message → Original Message (**Decryption**)

To encode a message, it is clear that we need some way of mapping words/symbols from the original message to their coded counterparts (and vice-versa). This brings about two vital questions:

- How do we **generate** this mapping?
- How do we **represent** this mapping?

We can represent any character-based code mapping using a tree, called a **coding tree**. In this tree, edges are labeled by “code” symbols, and the “non-coded” symbols label the leaves. A non-coded symbol (i.e., a leaf in the coding tree) is encoded by the “code” symbols labeling the edges in the path from the root to the non-coded symbol. For example, take the following simple coding tree:



In this tree, A is encoded by 00, C is encoded by 01, G is encoded by 10, and T is encoded by 11. Given an arbitrary string we wish to encode, we can simply replace each letter of the string with the corresponding code defined as the path from the code tree's root to the leaf corresponding to the given letter. Given an arbitrary encoded string, we can decode it by starting at the root and traversing the path defined by the encoded string; each time we hit a leaf, we simply output the corresponding letter, return to the root, and continue the traversal.

In this section, we simply wanted you to focus on how to encode or decode a message when given a mapping, and we wanted to demonstrate the fact that such a mapping could be represented using a coding tree. In the next section, we will cover the requirements and restrictions of such a mapping as well as how to go about creating one.

7.2 Entropy and Information Theory

To be able to understand the theoretical components behind **compression**, we must first dive into a key topic of information theory: **entropy**.

- Student A: “*Isn’t ‘entropy’ related to ‘chaos theory’? It seems irrelevant to information compression.*”
- Student B: “*I vaguely recall that word from Chemistry, but I only took the class to satisfy a major requirement so I wasn’t paying attention.*”
- Student C: “*I took biology courses for my General Science requirement because Bioinformatics is clearly the most interesting data science, so if this topic was only covered in Physics and Chemistry, it’s clearly useless to me.*”

Although Student C started off so strongly, unfortunately, all three of these students are wrong, and entropy is actually very important to Computer Science,

Mathematics, and numerous other fields. For our purposes, we will focus on the relationship between **entropy**, **data**, and **information**.

Information, at its core, is any propagation of cause-and-effect within a system. It is effectively just the *content of some message*: it tells us some detail(s) about some system(s). We can think of **data** as the raw unit of information. In other words, data can be thought of as the *representation* (or encoding) of *information* in some form that is better suited for processing or using (e.g. a measurement, a number, a series of symbol, etc.). A simple example of the relationship between data and information is the following: At the end of the quarter, Data Structures students receive a final percentage score in the class. A student's score in the class (a number) is a piece of *data*, and the *information* represented by this piece of data is the student's success in the course. If Bill tells us he received a score of 100%, the *data* we receive is "100%" (a number), and the *information* we extract from the data is "Bill did well in the Data Structures course."

Exercise Break

For each of the following items, choose which word (**data** or **information**) best describes the item.

- A list containing the quantity of each type of specialty pizza that was sold
- Which type of specialty pizza was the most popular
- A table containing survey results where individuals listed their age and their preferred programming language
- A positive correlation between age and preference of the Perl language

Now that we have formal definitions of the terms *data* and *information*, we can begin to tie in **entropy**, which is in general a measure of the disorder (i.e., non-uniformity) of a system. In the Physical Sciences, entropy measures disorder among molecules, but in Information Theory, entropy (**Shannon entropy** in this context) measures the unpredictability of information content. Specifically, *Shannon entropy* is the **expected value** (i.e., average) of the information contained in some data.

Say we flip a fair coin. The result of this flip contains 1 bit of information: either the coin landed on heads (1), or it landed on tails (0). However, if we were to flip a biased coin, where both sides are heads, the result of this flip contains 0 bits of information: we know before we even flip the coin that it *must* land on heads, so the actual outcome of the flip doesn't tell us anything we didn't already know. In general, if there are n possible outcomes of an event, that event has a value of $\lceil \log_2(n) \rceil$ bits of information, and it thus takes $\lceil \log_2(n) \rceil$ bits of memory to represent the outcomes of this event (which is why it takes $\lceil \log_2(n) \rceil$ bits to represent the numbers from 0 to $n - 1$ in binary).

Notice that we used the term "uniform." For our purposes, **uniformity** refers to the variation (or lack-thereof, specifically) among the symbols in our

message. For example, the sequence AAAA is **uniform** because only a single symbol appears (A). The sequence ACGT, however, is **non-uniform** (assuming an alphabet of only $\{A, C, G, T\}$) because there is no symbol in our alphabet that appears more frequently than the others. In general, the *more unique symbols* appear in our message, and the *more balanced the frequencies* of each unique symbol, the *higher* the entropy.

The topic of entropy gets pretty deep, but for the purposes of this text, the main take away is the following: **more uniform data** \rightarrow **less entropy** \rightarrow **less information stored in the data**. This simple relationship is the essence of data compression. Basically, if there are k bits of information in a message (for our purposes, let's think of a message as a binary string), but we are representing the message using n bits of memory (where $n > k$), there is clearly some room for improvement. This is the driving force behind data compression: can we think of a more clever way to encode our message so that the number of bits of memory used to represent it is equal to (or is at least closer to) the number of bits of information the message contains?

Exercise Break

Which of the following strings has the highest entropy? (Note that a calculator is not needed: you should be able to reason your way to an answer)

- AAAACGTAAAACGT
- ACGTACGTACGTACGT
- 1010101010101010

As mentioned before, we want to try to encode our messages so that the number of bits of *memory used to store the message* is equal to (or barely larger than) the number of bits of *information in the message*. A common example of this is the storage of DNA sequences on a computer. Recall that DNA sequences can be thought of as strings coming from an alphabet of four letters: **A**, **C**, **G**, and **T**. Typically, we store DNA sequences as regular text files, so each letter is represented by one ASCII symbol. For example, the sequence **ACGT**would be stored as:

$$\text{ACGT} \rightarrow \text{01000001 } \text{01000011 } \text{01000111 } \text{01010100} \text{ (4 bytes)}$$

However, since we know in advance that our alphabet only contains 4 possible letters, assuming perfect encoding, each letter can theoretically take $\log_2(4) = 2$ bits of memory to represent (as opposed to the 8 bits we are currently using per letter). If we simply map **A** \rightarrow **00**, **C** \rightarrow **01**, **G** \rightarrow **10**, and **T** \rightarrow **11**, we can now encode our DNA string as follows:

$$\text{ACGT} \rightarrow \text{00011011} \text{ (1 byte)}$$

This simple encoding method allowed us to achieve a guaranteed 4-fold compression, regardless of the actual sequence. However, note that we only partially analyzed the entropy of our message: we took into account that, out of all 256 possible bytes, we only use 4 to encode our message (**A**, **C**, **G**, and **T**), but what about the disorder among the letters we actually saw in the message? In this toy example, we saw all four letters in equal frequency (so there was *high disorder*). However, what if there is less disorder? Can we do better?

In the previous example, the string **ACGT** was represented by a single byte (**00011011**). However, what about the string **AAAAACGT**? Clearly we can represent it in two bytes (**00000000 00011011**), but now that we've seen the message, can we use the extra knowledge we've gained about the message in order to better encode it? Since **A** is the most frequent letter in our string, we can be clever and represent it using less than 2 bits, and in exchange, we can represent some other (less frequent) character using more than 2 bits. Although we will have a loss as far as compression goes when we encounter the longer-represented less-frequent character, we will have a gain when we encounter the shorter-represented more-frequent character. Let's map **A** → **0**, **C** → **10**, **G** → **110**, and **T** → **111**:

$$\text{AAAAACGT} \rightarrow \text{00000101 10111} \quad (1.625 \text{ bytes})$$

Of course, in memory, we cannot store fractions of bytes, so the number “1.625 bytes” is meaningless on its own, but what if we repeated the sequence **AAAAACGT** 1,000 times? With the previous naive approach, the resulting file would be 2,000 bytes, but with this approach, the resulting file would be 1,625 bytes, and both of these files are coming from a file that was originally 8,000 bytes (8 ASCII characters repeated 1,000 times). Note that, if we encode a message based on its character frequencies, we need to provide that vital character-frequency information to the recipient of our message so that the recipient can decode our message. To do so, in addition to the 1,625 bytes our compressed message occupies, we need to add enough information for the recipient to reconstruct the coding scheme to the beginning of our file, which causes slight overhead that results in a compressed file of just over 1,625 bytes. We will expand on this “overhead of added frequency information” shortly.

Basically, in addition to simply limiting our alphabet based on what characters we expect to see (which would get us down from 8,000 bytes to 2,000 bytes in this example), we can further improve our encoding by taking into account the frequencies at which each letter appears (which would get us down even lower than 2,000 bytes in this example). Unfortunately, not everybody cares about Biology, so these results will seem largely uninteresting if our compression technique only supports DNA sequences. Can we take this approach of looking at character frequencies and generalize it?

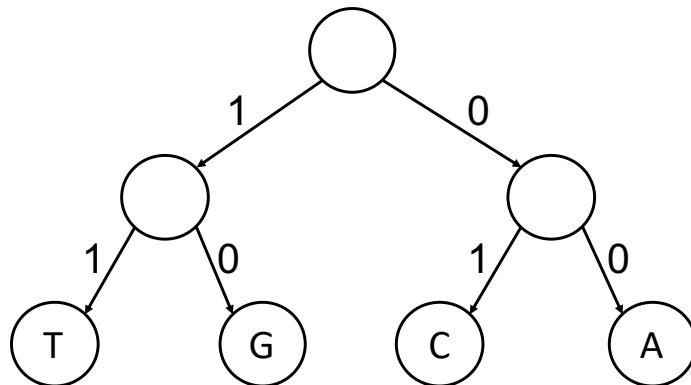
7.3 Honey, I Shrunk the File

In 1951, David A. Huffman and his classmates in an Information Theory course were given the choice of a term paper or a final exam. The topic of the term paper: Find the most efficient method of representing numbers, letters or other symbols using a binary code. After months of working on the problem without any success, just as he was about to give up on the enticing offer of skipping an engineering course's final exam, Huffman finally solved the problem, resulting in the birth of **Huffman Coding**, an algorithm that is still widely used today.

Say we have a set of n possible items and we want to represent them uniquely. We could construct a **balanced binary tree** of height $\lceil \log_2(n) \rceil$. Each item would be represented by a unique sequence of no more than $\lceil \log_2(n) \rceil$ bits (1 for right child, 0 for left child) that start at the root of the tree and continue to the leaf where that item is stored. Since data items are only in leaves, this gives a **prefix code**: no symbol is represented by a sequence that is a prefix of a sequence representing another symbol.

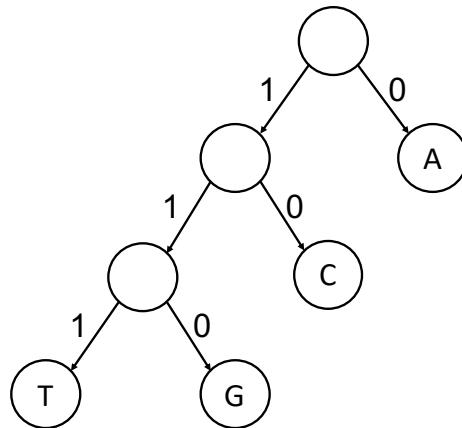
To be able to successfully encode and decode a message, our code *must* be a prefix code. For example, say $(\text{A}, \text{C}, \text{G}, \text{T})$ maps to $(0, 1, 10, 11)$ and I ask you to decode the string 110. This mapping is not a prefix code: $\text{C}(1)$ is a prefix of $\text{G}(10)$ and $\text{T}(11)$. Because of ambiguity arising from this fact, we are unable to exactly decode the message: $110 = \text{CCA}$, $010 = \text{CG}$, and $110 = \text{TA}$, so we fail to recover the original message.

Thus, it is clear that our code *must* be a prefix code, but in addition to the term “prefix code,” we also mentioned that we could theoretically construct a *balanced* binary tree. However, do we want our encoding tree to necessarily be perfectly balanced? We saw in the previous examples that a perfectly balanced binary tree $(00, 01, 10, 11)$ may not be optimum if the symbol frequencies vary. The following is the tree representing our initial symbol mapping:



When we used the unequal-length mapping $(0, 1, 10, 11)$, even though some (less frequent) symbols were encoded with more than $\lceil \log_2(n) \rceil$ bits, because the **more frequent** symbols were encoded with **less than** $\lceil \log_2(n) \rceil$ bits, we achieved significantly better compression. How do I know what the “best

possible case” is for compressing a given message? In general, the entropy (in “bits”) of a message is $\sum_i p_i \log \frac{1}{p_i}$ over all possible symbols i , where p_i is the frequency of symbol i . This entropy is the **absolute lower-bound** of how many bits we can use to store the message in memory. The following is the tree representing the improved symbol mapping, which happens to be the encoding tree that the Huffman algorithm would produce from our message:



Huffman’s algorithm can be broken down into three stages: **Tree Construction**, **Message Encoding**, and **Message Decoding**.

Exercise Break

Which of the following statements about Huffman compression are true?

- The Huffman algorithm takes into account symbol frequencies when deciding how to encode symbols
- The Huffman algorithm always generates a balanced binary tree to ensure $\lceil \log_2(n) \rceil$ symbol encoding length
- A header must be added to the compressed file to provide enough information so that the recipient can reconstruct the same coding tree that was used to encode the file

Huffman’s Algorithm: Tree Construction The algorithm creates the Huffman Tree in a **bottom-up** approach:

1. Compute the frequencies of all symbols that appear in the input
2. Start with a “forest” of single-node trees, one for each symbol that appeared in the input (ignore symbols with count 0, since they don’t appear at all in the input)

3. While there is more than 1 tree in the forest:
 - (a) Remove the two trees (T_1 and T_2) from the forest that have the lowest count contained in their roots
 - (b) Create a new node that will be the root of a new tree. This new tree will have T_1 and T_2 as left and right subtrees. The count in the root of this new tree will be the sum of the counts in the roots of T_1 and T_2 . Label the edge from this new root to T_1 as “1” and label the edge from this new root to T_2 as “0”
 - (c) Insert this new tree in the forest, and go back to 3
4. Return the one tree in the forest as the Huffman tree

We have created a visualization in which we construct a Huffman Tree from the string AAAAABBAHHBCBGCCC, which can be found at <https://goo.gl/Dv2fL8>.

Huffman’s Algorithm: Message Encoding Once we have a Huffman Tree, encoding the message is easy: For each symbol in the message, output the sequence of bits given by 1’s and 0’s on the path from the root of the tree to that symbol. A typical approach is to keep a pointer to each symbol’s node, and upon encountering a symbol in the message, go to that symbol’s node (which is a **leaf** in the tree) and follow the path to the root with recursive calls, and output the sequence of bits as the recursive calls pop from the runtime stack. We push and pop bits to/from a stack because a leaf’s encoding is the path from the *root* to the *leaf*, but in this traversal we described, we follow the path from *leaf* to *root*, meaning each leaf’s encoding will be *reversed!* So using a stack to push/pop helps us reverse the path labels in order to get the true encoding.

We have created a visualization in which we use the previously-constructed Huffman Tree to encode our string (AAAAABBAHHBCBGCCC), which can be found at <https://goo.gl/SAmH5W>.

Huffman’s Algorithm: Message Decoding If we have the Huffman Tree, decoding a message is easy:

- Start with the first bit in the coded message, and start with the root of the code tree
- As you read each bit, move to the left or right child of the current node, matching the bit just read with the label on the edge
- When you reach a leaf, output the symbol stored in the leaf, return to the root, and continue

We have created a visualization in which we use the previously-constructed Huffman Tree to decode our encoded string, which can be found at <https://goo.gl/4Bb8zf>.

Huffman Compression is just one approach to file compression, but many other methods exist as well. Even with Huffman Compression, clever techniques exist to further improve its compression abilities. For example, biologically, different segments of an organism's genome may have different symbol frequencies. Say, for example, that the first half of a genome is enriched in C's and G's, whereas the second half of the genome is enriched in A's and T's. If we were to split the genome into two pieces, Huffman Compress each piece independently of the other, and then merge these two encoded messages (including some indicator in the merged encoded message that tells us when one segment has finished and the other has started), we will be able to attain even better compression than if we were to Huffman Compress the entire genome in one go.

Also, note that Huffman Compression is a **lossless** compression algorithm, meaning we can compress a file, and when we decompress the resulting file, it will be completely identical to the original. However, we would be able to achieve better compression if we removed this lossless requirement. Such algorithms, in which the original file *cannot* be extracted from the compressed file, are called **lossy** compression algorithms. For example, when it comes to music, the lossless source file of a typical song 4 minutes in length will be approximately 25 MB. However, if we were to perform lossy compression on the file by encoding it using the MP3 audio compression algorithm, we can compress the file to somewhere in the range of 1 to 5 MB depending on what bitrate at which we choose to compress the file.

We live in an age of data, so file compression is vital to practically every part of our daily life. With better file compression, we could

- store more songs and movies on our mobile devices,
- speed up our internet browsing,
- archive important files in a feasible fashion,
- or do infinitely more cool things!

7.4 Bitwise I/O

We introduced the idea of **lossless data compression** with the goal of encoding data in such a way that we would be able to store data in smaller files on a computer with the ability to decode the encoded data perfectly, should we want to. Then, we discussed **Huffman Coding**, which allowed us to take a message and encode it in the fewest number of bits possible. However, now that we have this string of bits, how do we actually go about writing it to disk?

Recall that the smallest unit of data that can be written to disk is a *byte*, which is (for our purposes) a chunk of 8 *bits*. However, we want to be able to write our message to disk one bit at a time, which sounds impossible given what we just said. In this section, we will be discussing **Bitwise I/O**: the process of reading and writing data to and from disk *bit-wise* (as opposed to traditional

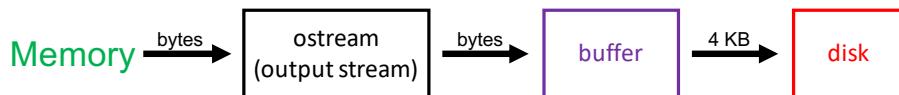
I/O, which is *byte*-wise). The discussion will be using C++ specifically, but the process is largely the same in other languages.

In regular bytewise I/O, we like to think that we write a single byte to disk at a time. However, note that accessing the disk is an *extremely* slow operation. We won't go into the computer architecture details that explain *why* this is the case, but take our word for now that it is. As a result, instead of writing every single byte to disk one at a time (which would be very slow), computers (actually, programming languages) have a "write buffer": we designate numerous bytes to be written, but instead of being written to disk immediately, they are stored in a buffer in memory (which is fast), and once the buffer is full, the *entire buffer* is written to disk at once. Likewise, for *reading* from disk, the same logic holds: reading individual bytes is extremely slow, so instead, we have a "read buffer" in memory to which we read *multiple* bytes from disk, and we pull individual bytes from this buffer. Typically, a default value for a disk buffer is 4 kilobytes (i.e., 4,096 bytes).

The basic workflow for **writing** *bytes* to disk is as follows:

- Write bytes to buffer, one byte at a time
- Once the buffer is full, write the entire buffer to disk and clear the buffer (i.e., "flush" the buffer)
- Repeat

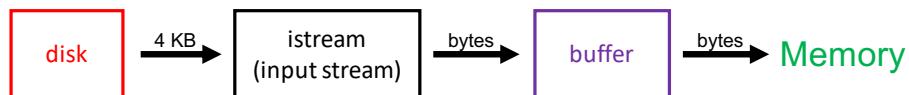
In most languages, we can create an "output stream" that handles filling the buffer, writing to disk, etc. for us on its own. The following is a typical workflow of **writing** some *bytewise* data from **memory** to **disk**:



Similarly, the basic workflow for **reading** *bytes* from disk is as follows:

- Read enough bytes from disk to fill the entire buffer (i.e., "fill" the buffer)
- Read bytes from buffer, one byte at a time, until the buffer is empty
- Repeat

Like with writing from disk, in most languages, one can create an "input stream" that handles reading from disk, filling the buffer, etc. The following is a typical workflow of **reading** some *bytewise* data from **disk** to **memory**:

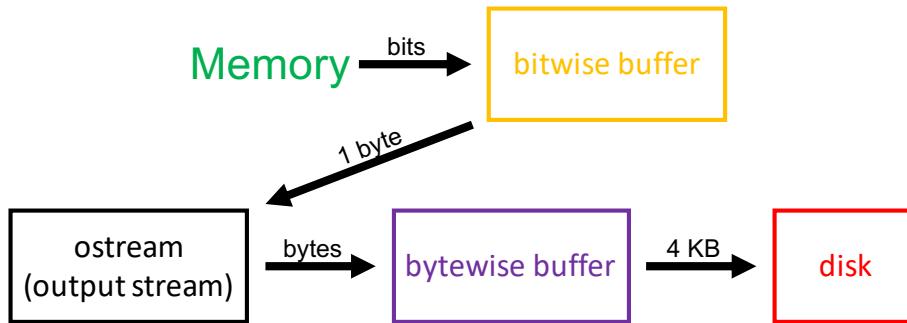


We've seen a workflow for writing *bytes* to disk (as well as for reading bytes from disk), but we want to be able to read/write *bits*. It turns out that we can use the same idea with a small layer on top to achieve this: we can build our own *bitwise* buffer that connects with the existing *bytewise* buffer. We will design our bitwise buffer to be 1 byte (i.e., 8 bits).

The basic workflow for **writing** *bits* to disk is as follows:

- Write bits to bitwise buffer, one bit at a time
- Once the bitwise buffer is full, write the bitwise buffer (which is 1 byte) to the bytewise buffer and clear the bitwise buffer (i.e., "flush" the bitwise buffer)
- Repeat until the bytewise buffer is full
- Once the bytewise buffer is full, write the entire bytewise buffer to disk and clear the bytewise buffer (i.e., "flush" the bytewise buffer)
- Repeat from the beginning

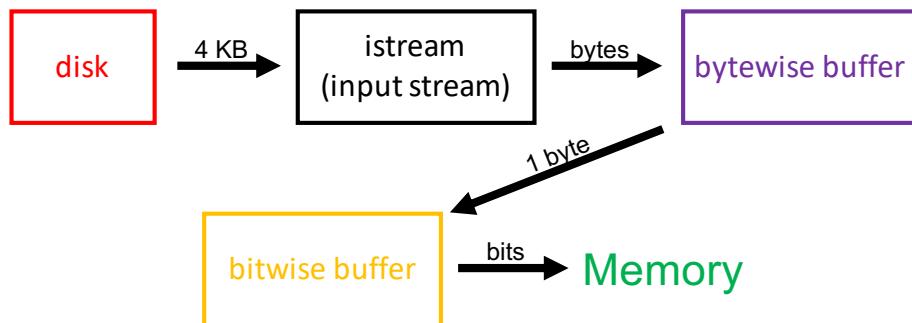
The following is a typical workflow of **writing** some *bitwise* data from **memory** to **disk**:



Similarly, the basic workflow for **reading** *bits* from disk is as follows:

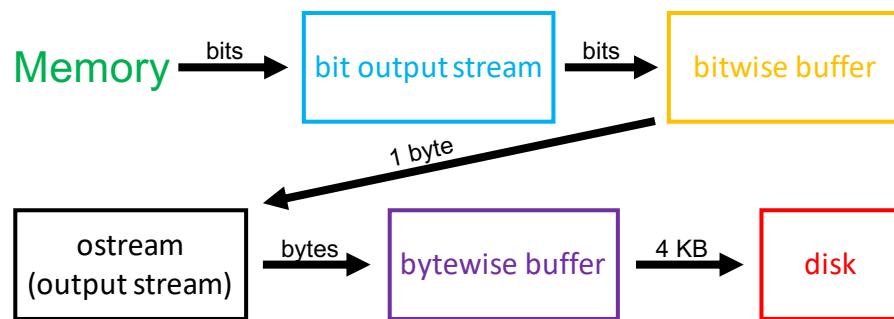
- Read enough bytes from disk to fill the entire bytewise buffer (i.e., "fill" the bytewise buffer)
- Read 1 byte from the bytewise buffer to fill the bitwise buffer (i.e., "fill" the bitwise buffer)
- Read bits from bitwise buffer, one bit at a time
- Once the bitwise buffer is empty, read another byte from the bytewise buffer to fill the bitwise buffer, repeat (i.e., "fill" the bitwise buffer)
- Once the bytewise buffer is empty, repeat from the beginning (i.e., "fill" the bytewise buffer)

The following is a typical workflow of **reading** some *bitwise* data from **disk** to **memory**:

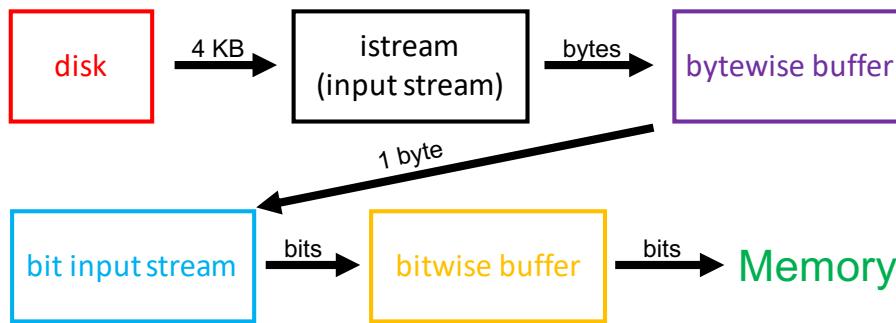


Programming languages include “input streams” and “output streams” so that we as programmers don’t have to worry with low-level details like handling bytewise buffers. With similar motivation, instead of us directly using a bitwise buffer, it would be better design if we were to design a “bit input stream” and a “bit output stream” to handle reading/writing the bitwise buffer for us automatically. Before we embark on the journey of actually designing the “bit input stream” and “bit output stream,” let’s reformulate our input and output workflows.

To write to disk, we will add a bit output stream to which we will write our bitwise data, and this bit output stream will feed into the regular output stream. The following is the updated workflow of **writing** some *bitwise* data from **memory** to **disk**:



Similarly, to read from disk, we will add a bit input stream from which we will read our bitwise data, and this bit input stream will feed from the regular input stream. The following is the updated workflow of **reading** some *bitwise* data from **disk** to **memory**:



Thus far, we have mentioned that we flush the bytewise buffer once it's been filled, which means that we write all of the bytes to disk. What about if we have run out of bytes we want to write, but our bytewise buffer is not full? We mentioned previously that a “good size” for a bytewise buffer is 4 KB (i.e., 4,096 bytes), but what if we only have 100 bytes we wish to write to disk? The solution is trivial: we just flush the bytewise buffer by writing whatever bytes we *do* have to disk. In the example above, instead of writing 4 KB, we only write 100 bytes.

However, with a bitwise buffer, things are a bit more tricky if we run out of bits to write before the bitwise buffer is full. The difficulty is caused by the fact that the smallest unit that can be written to disk is 1 *byte*, which is 8 *bits*. For example, say we want to write to disk a sequence of bits that is not a multiple of 8 (e.g. 111111111111):

- We would write the first 8-bit chunk (11111111), and 1111 would be left
- We would then try to write the remaining chunk (1111), but we are unable to do so because it is not a full byte

The solution is actually still quite simple: we can simply “pad” the last chunk of bits with 0s such that it reaches the full 8 bits we need to write to disk. Because bytes and bits are traditionally read from “left” to “right” (technically the way it is laid out in the computer architecture does not have “left” or “right” directionality, but instead “small offset” to “large offset”), we want our padding 0s to be on the right side of our final byte such that the bits remain contiguous. In the same example as above:

- We would write the first 8-bit chunk (11111111), and 1111 would be left
- We would then pad 1111 to become 11110000, and we would write this padded byte
- The result on disk would be: 11111111 11110000 (the space implies that the bits are two separate bytes)

This way, when we read from disk, we can simply read the bits from left to right. However, note that we have no way of distinguishing the padding

0s from “true” bits. For example, what if we were to write the bit sequence 111111111110000?

- We would write the first 8-bit chunk (11111111), and 11110000 would be left
- We would then write the remaining 8-bit chunk (11110000)
- The result on disk would be: 11111111 11110000 (the same two bytes written to disk as before)

As can be seen, because we cannot distinguish the padding 0s from “true” bits (as they’re indistinguishable on disk), we need to implement some clever way of distinguishing between them manually. For example, we could add a “header” to the beginning of the file that tells us how many bits we should be reading. For example, we could do the following in the previous example:

- 00001100 11111111 11110000
- The first byte (00001100) tells us how many bits we should expect to read ($00001100 = 12$)
- We would then read the next byte (11111111) and know that we have read 8 bits (so we have 4 bits left to read)
- We would then read the next byte (11110000) and know that we should only read the first 4 bits (1111)

It is important to note that this exact header implementation (“How many bits should we expect to read?”) is just *one* example. Thus, in whatever applications you may develop that require bitwise input and output, you should think of a clever header that would be optimal for your own purposes.

As we have hinted at, our end-goal is to create two classes, `BitOutputStream` and `BitInputStream`, that will handle bitwise output and input for us. Specifically, our `BitOutputStream` class should have a `writeBit()` function (write a single bit to the bitwise buffer) and a `flush()` function (write the bitwise buffer to the output stream, and clear the bitwise buffer), and our `BitInputStream` class should have a `readBit()` function (read a single bit from the bitwise buffer) and a `fill()` function (read one byte from the input stream to fill the bitwise buffer).

The following is an example of a potential `BitOutputStream` class. Some of the details have been omitted for you to try to figure out.

```

1 class BitOutputStream {
2     private:
3         unsigned char buf;    // bitwise buffer (one byte)
4         int nbits;           // number of bits in bitwise buffer
5         ostream & out;      // ref to bytewise output stream
6     public:
7         BitOutputStream(ostream& os) : out(os), buf(0), nbits(0) {}
8         void flush() {
9             out.put(buf);      // write bitwise buffer to ostream
10            out.flush();     // flush ostream
11            buf = 0;          // clear bitwise buffer
12            nbits = 0;        // no bits in bitwise buffer
13        }
14        void writeBit(unsigned int bit) {
15            if(nbits == 8) { // flush bitwise buffer if full
16                flush();
17            }
18            // write 'bit' to bitwise buffer (how?)
19            nbits++;          // one more bit in bitwise buffer
20        }
21    };
22

```

The following is an example of a potential `BitInputStream` class. Some of the details have been omitted for you to try to figure out.

```

1 class BitInputStream {
2     private:
3         unsigned char buf;    // bitwise buffer (one byte)
4         int nbits;           // # bits read from bitwise buffer
5         istream & in;        // ref to bytewise input stream
6     public:
7         BitInputStream(istream& is) : in(is), buf(0), nbits(8) {}
8         void fill() {
9             buf = in.get();    // read one byte from istream
10            nbits = 0;        // no bits read from bitwise buffer
11        }
12        unsigned int readBit() {
13            if(nbits == 8) { // fill bitwise buffer if empty
14                fill();
15            }
16            // read bit from bitwise buffer (how?)
17            unsigned int nextBit = ??;
18            nbits++;          // one more bit read
19            return nextBit;   // return bit we just read
20        }
21    };
22
23
24 };

```

STOP and Think

In our `BitOutputStream` class, we flush the C++ `ostream` in our `flush()` function, but this is optional, and it actually slows things down. Why would flushing the `ostream` here slow things down? Where might we want to flush the `ostream` instead to keep things as fast as possible?

As you hopefully noticed, we omitted some of the logic from `writeBit()` and `readBit()` for you to try to figure out on your own. Specifically, in both functions, we omitted bitwise operations that perform the relevant reading/writing of the bitwise buffer. In the following exercises, you will be solving some bitwise arithmetic challenges that will hopefully push you towards the bitwise operation logic necessary to complete `writeBit()` and `readBit()`.

Exercise Break

Given a byte (i.e., 8 bits) `byte` and a single bit `bit`, how would you set the **rightmost** bit of `byte` to `bit`? For example, if `byte` is 254 (i.e., `11111110`) and `bit` is 1, the result would be 255 (i.e., `11111111`).

Exercise Break

Given a byte (i.e., 8 bits) `byte` and a single bit `bit`, how would you set the **leftmost** bit of `byte` to `bit`? For example, if `byte` is 127 (i.e., `01111111`) and `bit` is 1, the result would be 255 (i.e., `11111111`).

Exercise Break

Given a byte (i.e., 8 bits) `byte`, a single bit `bit`, and an integer `n`, how would you set the **n-th bit from the right bit** (0-based counting) of `byte` to `bit`? For example, if `byte` is 85 (i.e., `01010101`), `n` is 3, and `bit` is 1, the result would be 93 (i.e., `01011101`).

Exercise Break

Given a byte (i.e., 8 bits) `byte`, how would you extract the **rightmost** bit of `byte`? For example, if `byte` is 255 (i.e., `11111111`), the result would be 1.

Exercise Break

Given a byte (i.e., 8 bits) `byte` and an integer `n`, how would you extract the **n-th bit from the right bit** (0-based counting) of `byte`? For example, if `byte` is 247 (i.e., `11110111`) and `n` is 3, the result would be 0.

Chapter 8

Summary

8.1 Array List (Dynamic Array)

Summary Description

- Array Lists (or Dynamic Arrays) are simply arrays wrapped in a container that handles automatically resizing the array when it becomes full
- As a result, if we know which index of the Array List we wish to access, we can do so in $\mathcal{O}(1)$ time because of the array's random access property
- Array Lists are typically implemented such that the elements are contiguous in the array (i.e., there are no empty slots in the array)

8.1.1 Unsorted Array List

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — We need to iterate over all n elements to find the query
- **Insert:** $\mathcal{O}(n)$ — If we insert at the front of the Array List, we need to move over each of the n elements
- **Remove:** $\mathcal{O}(n)$ — If we remove from the front of the Array List, we need to move over each of the n elements

Average-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — The average number of checks is $\frac{1+2+\dots+n}{n} = \frac{\sum_{i=1}^n i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$
- **Insert:** $\mathcal{O}(n)$ — The average number of moves is $\frac{n+(n-1)+\dots+1+0}{n} = \frac{\sum_{i=1}^n i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$

- **Remove:** $\mathcal{O}(n)$ — The average number of moves is $\frac{(n-1)+\dots+1+0}{n} = \frac{\sum_{i=1}^{n-1} i}{n} = \frac{n(n-1)}{2n} = \frac{n-1}{2}$

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — The query is the first element in the Array List
- **Insert:** $\mathcal{O}(1)$ — If we insert at the end of the Array List, we don't need to move any elements
- **Remove:** $\mathcal{O}(1)$ — If we remove from the end of the Array List, we don't need to move any elements

Space Complexity $\mathcal{O}(n)$ — The two extremes are *just before* resizing (completely full, so the array is of size n) or *just after* resizing (array is half full, so the array is of size $2n$)

8.1.2 Sorted Array List

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — We can perform Binary Search to find an element
- **Insert:** $\mathcal{O}(n)$ — If we insert at the front of the Array List, we need to move over each of the n elements
- **Remove:** $\mathcal{O}(n)$ — If we remove from the front of the Array List, we need to move over each of the n elements

Average-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — The derivation is too complex for a summary, but it's the average case of Binary Search
- **Insert:** $\mathcal{O}(n)$ — The average number of moves is $\frac{n+(n-1)+\dots+1+0}{n} = \frac{\sum_{i=1}^n i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$
- **Remove:** $\mathcal{O}(n)$ — The average number of moves is $\frac{(n-1)+\dots+1+0}{n} = \frac{\sum_{i=1}^{n-1} i}{n} = \frac{n(n-1)}{2n} = \frac{n-1}{2}$

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — The query is the first element in the Array List we check via Binary Search
- **Insert:** $\mathcal{O}(1)$ — If we insert at the end of the Array List, we don't need to move any elements
- **Remove:** $\mathcal{O}(1)$ — If we remove from the end of the Array List, we don't need to move any elements

Space Complexity $\mathcal{O}(n)$ — The two extremes are *just before* resizing (completely full, so the array is of size n) or *just after* resizing (array is half full, so the array is of size $2n$)

8.2 Linked List

Summary Description

- Linked Lists are composed of nodes connected to one another via pointers
- We typically only keep global access to two pointers: a *head* pointer (which points to the first node) and a *tail* pointer (which points to the last node)
- In a Singly-Linked List, each node maintains one pointer: a *forward* pointer that points to the next node in the list
- In a Doubly-Linked List, each node maintains two pointers: a *forward* pointer that points to the next node in the list, and a *previous* pointer that points to the previous node in the list
- Unlike an Array List, we do *not* have direct access to any nodes other than *head* and *tail*, meaning we need to iterate node-by-node one-by-one to access inner nodes
- Once we know where in the Linked List we want to insert or remove, the actual insertion/removal is $\mathcal{O}(1)$ because we just change pointers around
- As a result, in each of the cases (worst, average, and best), the time complexity of insert and remove are the same as the time complexity of find, because you call find and then perform a $\mathcal{O}(1)$ operation to perform the insertion/removal

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — In both Singly- and Doubly-Linked Lists, if our query is the middle element, we need to iterate over $\frac{n}{2}$ nodes
- **Insert:** $\mathcal{O}(n)$ — Perform find, which is $\mathcal{O}(n)$ in the worst case, and then perform $\mathcal{O}(1)$ pointer changes
- **Remove:** $\mathcal{O}(n)$ — Perform find, which is $\mathcal{O}(n)$ in the worst case, and then perform $\mathcal{O}(1)$ pointer changes

Average-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — The average number of checks in a Singly-Linked List is $\frac{1+2+\dots+n}{n} = \frac{\sum_{i=1}^n i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$, and in a Doubly-Linked List, if we know the index we want to access, it is $2\frac{\frac{n}{2}+1}{2} = \frac{n}{2} + 1$
- **Insert:** $\mathcal{O}(n)$ — Perform find, which is $\mathcal{O}(n)$ in the average case, and then perform $\mathcal{O}(1)$ pointer changes
- **Remove:** $\mathcal{O}(n)$ — Perform find, which is $\mathcal{O}(n)$ in the average case, and then perform $\mathcal{O}(1)$ pointer changes

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — Our query is the first element we check (or if we specify *head* or *tail* as our query)
- **Insert:** $\mathcal{O}(1)$ — Perform find, which is $\mathcal{O}(1)$ in the best case, and then perform $\mathcal{O}(1)$ pointer changes
- **Remove:** $\mathcal{O}(1)$ — Perform find, which is $\mathcal{O}(1)$ in the best case, and then perform $\mathcal{O}(1)$ pointer changes

Space Complexity $\mathcal{O}(n)$ — Each node contains either 1 or 2 pointers (for Singly- or Doubly-Linked, respectively) and the data, so $\mathcal{O}(1)$ space for each node, and we have exactly n nodes

8.3 Skip List

Summary Description

- Skip Lists are like Linked Lists, except every node has multiple *layers*, where each layer is a forward pointer
- We denote the number of layers a given node has as the node's *height*
- We typically choose a maximum height, h , that any node in our Skip List can have, where $h \ll n$ (the total number of nodes)
- We are able to “skip” over multiple nodes in our searches (unlike in Linked Lists, where we had to traverse through the nodes one-by-one), which allows us to mimic Binary Search when searching for an element in the list
- To determine the height of a new node, we repeatedly flip a weighted coin that has probability p to land on heads, and we keep adding layers to the new node until we encounter our first tails
- Just like with a Linked List, to insert or remove an element, we first run the regular find algorithm and then perform a single $\mathcal{O}(h)$ operation to fix pointers

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — If all of our nodes have the same height (low probability, but possible), we just have a regular Linked List
- **Insert:** $\mathcal{O}(n)$ — Perform find, which is $\mathcal{O}(n)$ in the worst case, and then perform a single $\mathcal{O}(h)$ pointer fix ($h \ll n$)
- **Remove:** $\mathcal{O}(n)$ — Perform find, which is $\mathcal{O}(n)$ in the worst case, and then perform a single $\mathcal{O}(h)$ pointer fix ($h \ll n$)

Average-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Insert:** $\mathcal{O}(n)$ — Perform find, which is $\mathcal{O}(\log n)$ in the average case, and then perform a single $\mathcal{O}(h)$ pointer fix ($h \ll n$)
- **Remove:** $\mathcal{O}(n)$ — Perform find, which is $\mathcal{O}(\log n)$ in the average case, and then perform a single $\mathcal{O}(h)$ pointer fix ($h \ll n$)

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — Our query is the first element we check
- **Insert:** $\mathcal{O}(h)$ — Perform find, which is $\mathcal{O}(1)$ in the best case, and then perform a single $\mathcal{O}(h)$ pointer fix ($h \ll n$)
- **Remove:** $\mathcal{O}(h)$ — Perform find, which is $\mathcal{O}(1)$ in the best case, and then perform a single $\mathcal{O}(h)$ pointer fix ($h \ll n$)

Space Complexity $\mathcal{O}(n \log n)$ — The formal proof is too complex for a summary

8.4 Heap

Summary Description

- A Heap is a complete binary tree that satisfies the *Heap Property*
- *Heap Property*: For all nodes A and B , if node A is the parent of node B , then node A has higher priority (or equal priority) than node B
- A *min*-heap is a heap in which every node is *smaller* than (or equal to) all of its children (or has no children)
- A *max*-heap is a heap where every node is *larger* than (or equal to) all of its children (or has no children)
- It is common to implement a Heap as an Array List because all of the data will be localized in memory (faster in practice)
- If a Heap is implemented as an Array List, the root is stored in index 0, the next node (in a level-order traversal) is at index 1, then at index 2, etc.
- If a Heap is implemented as an Array List, for a node u stored at index i , u 's parent is stored at index $\lfloor \frac{i-1}{2} \rfloor$, u 's left child is stored at index $2i + 1$, and u 's right child is stored at index $2i + 2$

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — We just return the root, which is a $\mathcal{O}(1)$ operation
- **Insert:** $\mathcal{O}(\log n)$ — Our new element has to bubble up the entire tree, which has a height of $\mathcal{O}(\log n)$
- **Remove:** $\mathcal{O}(\log n)$ — Our new root has to trickle down the entire tree, which has a height of $\mathcal{O}(\log n)$

Average-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — We just return the root, which is a $\mathcal{O}(1)$ operation
- **Insert:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Remove:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — We just return the root, which is a $\mathcal{O}(1)$ operation
- **Insert:** $\mathcal{O}(1)$ — Our new element doesn't have to bubble up at all
- **Remove:** $\mathcal{O}(1)$ — Our new root doesn't have to trickle down at all

Space Complexity $\mathcal{O}(n \log n)$ — We typically implement a Heap as an Array List

8.5 Binary Search Tree

Summary Description

- A Binary Search Tree is a binary tree in which any given node is larger than all nodes in its left subtree and smaller than all nodes in its right subtree
- A Randomized Search Tree is a Binary Search Tree + Heap (a “Treap”)
 - Each node has a *key* and a *priority*
 - The tree maintains the *Binary Search Tree Property* with respect to *keys*
 - The tree maintains the *Heap Property* with respect to *priorities*
- An AVL Tree is a self-balancing Binary Search Tree in which, for all nodes in the tree, the *heights* of the two child subtrees of the node differ by at most one
- A Red-Black Tree is a self-balancing Binary Search Tree in which
 - all nodes must be “colored” either red or black,
 - the root of the tree must be black,
 - red nodes can only have black children, and
 - every path from any node u to a null reference must contain the same number of black nodes
- AVL Trees are stricter than Red-Black Trees in terms of balance, so AVL Trees are typically faster for find operations
- Red-Black Trees only do one pass down the tree for inserting elements, whereas AVL trees need one pass down the tree and one pass back up, so Red-Black Trees are typically faster for insert operations

8.5.1 Regular Binary Search Tree

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — If we insert the elements in ascending/descending order, we get a Linked List
- **Insert:** $\mathcal{O}(n)$ — If we insert the elements in ascending/descending order, we get a Linked List
- **Remove:** $\mathcal{O}(\log n)$ — If we insert the elements in ascending/descending order, we get a Linked List

Average-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Insert:** $\mathcal{O}(\log n)$ — Effectively the same algorithm as find, with the actual insertion being a $\mathcal{O}(1)$ pointer rearrangement
- **Remove:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — The query is the root
- **Insert:** $\mathcal{O}(1)$ — The root only has one child and the node we are inserting becomes the root's other child
- **Remove:** $\mathcal{O}(1)$ — We are removing the root and the root only has one child

Space Complexity $\mathcal{O}(n)$ — Each node contains either 3 pointers (parent, left child, and right child) and the data, so $\mathcal{O}(1)$ space for each node, and we have exactly n nodes

8.5.2 Randomized Search Tree

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — If the elements are in ascending/descending order in terms of *both* keys *and* priorities, we get a Linked List
- **Insert:** $\mathcal{O}(n)$ — If the elements are in ascending/descending order in terms of *both* keys *and* priorities, we get a Linked List
- **Remove:** $\mathcal{O}(\log n)$ — If the elements are in ascending/descending order in terms of *both* keys *and* priorities, we get a Linked List

Average-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Insert:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Remove:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — The query is the root
- **Insert:** $\mathcal{O}(1)$ — The root only has one child and the node we are inserting becomes the root's other child
- **Remove:** $\mathcal{O}(1)$ — We are removing the root and the root only has one child

Space Complexity $\mathcal{O}(n)$ — Each node contains either 3 pointers (parent, left child, and right child) and the data, so $\mathcal{O}(1)$ space for each node, and we have exactly n nodes

8.5.3 AVL Tree

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — AVL Trees must be balanced by definition
- **Insert:** $\mathcal{O}(\log n)$ — AVL Trees must be balanced by definition, and the rebalancing is $\mathcal{O}(\log n)$
- **Remove:** $\mathcal{O}(\log n)$ — AVL Trees must be balanced by definition, and the rebalancing is $\mathcal{O}(\log n)$

Average-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Insert:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Remove:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — The query is the root
- **Insert:** $\mathcal{O}(\log n)$ — AVL Trees must be balanced, so we have to go down the entire $\mathcal{O}(\log n)$ height of the tree
- **Remove:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary

Space Complexity $\mathcal{O}(n)$ — Each node contains either 3 pointers (parent, left child, and right child) and the data, so $\mathcal{O}(1)$ space for each node, and we have exactly n nodes

8.5.4 Red-Black Tree

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — Red-Black Trees must be balanced (formal proof is too complex for a summary)
- **Insert:** $\mathcal{O}(\log n)$ — Red-Black Trees must be balanced (formal proof is too complex for a summary), so we have to go down the entire $\mathcal{O}(\log n)$ height of the tree, and the rebalancing occurs with $\mathcal{O}(1)$ cost during the insertion

- **Remove:** $\mathcal{O}(\log n)$ — Red-Black Trees must be balanced (formal proof is too complex for a summary), so we have to go down the entire $\mathcal{O}(\log n)$ height of the tree, and the rebalancing occurs with $\mathcal{O}(1)$ cost during the removal

Average-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Insert:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Remove:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — The query is the root
- **Insert:** $\mathcal{O}(\log n)$ — Red-Black Trees must be balanced (formal proof is too complex for a summary slide), so we have to go down the entire $\mathcal{O}(\log n)$ height of the tree, and the rebalancing occurs with $\mathcal{O}(1)$ cost during the insertion
- **Remove:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary

Space Complexity $\mathcal{O}(n)$ — Each node contains either 3 pointers (parent, left child, and right child) and the data, so $\mathcal{O}(1)$ space for each node, and we have exactly n nodes

8.6 B-Tree

Summary Description

- A B-Tree is a self-balancing tree in which
 - internal nodes must have at least b and at most $2b$ children (for some predefined b),
 - all leaves must be on the same level of the tree,
 - the elements within a given node must be in ascending order, and
 - child pointers appear between elements and on the edges of the node
- For two adjacent elements i and j in a B-Tree (where $i < j$, so i is to the left of j),
 - all elements down the subtree to the left of i must be smaller than i ,
 - all elements down the subtree between i and j must be greater than i and less than j , and
 - all elements down the subtree to the right of j must be greater than j

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — B-Trees must be balanced by definition
- **Insert:** $\mathcal{O}(b \log n)$ — B-Trees must be balanced by definition, and the re-balancing is $\mathcal{O}(b \log n)$ because, in the worst case, we need to shuffle around $\mathcal{O}(b)$ keys in a node at each level to keep the sorted order property
- **Remove:** $\mathcal{O}(b \log n)$ — B-Trees must be balanced by definition, and the re-balancing is $\mathcal{O}(b \log n)$ because, in the worst case, we need to shuffle around $\mathcal{O}(b)$ keys in a node at each level to keep the sorted order property

Average-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Insert:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Remove:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — The query is the middle element of the root (so Binary Search finds it first)
- **Insert:** $\mathcal{O}(\log n)$ — All insertions happen at the leaves. In the best case, no re-sorting or re-balancing is needed
- **Remove:** $\mathcal{O}(\log n)$ — Removing from any location will require the re-adjustment of child pointers *or* traversing the entire height of the tree. In the best case, no re-sorting or re-balancing is needed

Space Complexity $\mathcal{O}(n)$ — Every node must be at least $\frac{1}{3} \leq \frac{b-1}{2b-1} < \frac{1}{2}$ full, so in the worst case, every node is $\frac{1}{3}$ full, meaning we allocated $\mathcal{O}(3n)$ space, which is $\mathcal{O}(n)$

8.7 B+ Tree

Summary Description

- A B+ Tree can be viewed as a variant of the B-tree in which each internal node contains only keys and the leaves contain the actual data records
- M denotes the maximum number of children any node can have
- L denotes the maximum number of data records any node can have
- Every internal node except the root has at least $\lceil \frac{M}{2} \rceil$ children
- The root has at least 2 children (if it's not a leaf)
- An internal node with k children must contain $k - 1$ elements
- All leaves must be on the same level of the tree
- Internal nodes only contain “search keys” (no data records)
- The smallest data record between search keys i and j (where $i < j$) must equal i

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — B+ Trees must be balanced by definition
- **Insert:** $\mathcal{O}(M \log n + L)$ — B+ Trees must be balanced by definition, and the re-balancing is $\mathcal{O}(M \log n + L)$ because, in the worst case , we need to shuffle around $\mathcal{O}(M)$ keys in an internal node, $\mathcal{O}(L)$ data records in a leaf node, for $\mathcal{O}(\log n)$ levels of the tree in order to keep the sorted order property
- **Remove:** $\mathcal{O}(b \log n)$ — B+ Trees must be balanced by definition, and the re-balancing is $\mathcal{O}(M \log n + L)$ because, in the worst case , we need to shuffle around $\mathcal{O}(M)$ keys in an internal node, $\mathcal{O}(L)$ data records in a leaf node, for $\mathcal{O}(\log n)$ levels of the tree in order to keep the sorted order property

Average-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Insert:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Remove:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary

Best-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — All data records are stored at the leaves
- **Insert:** $\mathcal{O}(\log n)$ — All insertions happen at the leaves. In the best case, no re-sorting or re-balancing is needed
- **Remove:** $\mathcal{O}(\log n)$ — Removing from any location will require the re-adjustment of child pointers *or* traversing the entire height of the tree. In the best case, no re-sorting or re-balancing is needed

Space Complexity $\mathcal{O}(n)$ — Every node must be at least $\frac{\frac{M}{2}L}{ML} = \frac{1}{2} \left(\frac{ML}{ML} \right) = \frac{1}{2}$ full, so in the worst case, every node is $\frac{1}{2}$ full, meaning we allocated $\mathcal{O}(2n)$ space, which is $\mathcal{O}(n)$

8.8 Hashing

Summary Description

- A Hash Table is an array that, given a key key , computes a hash value from key (using a hash function) and then uses the hash value to compute an index at which to store key
- A Hash Map is the exact same thing as a Hash Table, except instead of storing just $keys$, we store $(key, value)$ pairs
- The capacity of a Hash Table should be prime (to help reduce collisions)
- When discussing the time complexities of Hash Tables/Maps, we typically ignore the time complexity of the hash function
- The load factor of a Hash Table, $\alpha = \frac{N}{M}$ (N = size of Hash Table and M = capacity of Hash Table), should remain below 0.75 to keep the Hash Table fast (a smaller load factor means better performance, but it also means more wasted space)
- For a hash function h to be *valid*, given two equal keys k and l , $h(k)$ must equal $h(l)$
- For a hash function h to be *good*, given two unequal keys k and l , $h(k)$ should ideally (but not necessarily) not equal $h(l)$
- A good hash function for a collection that stores k items (e.g. a string storing k characters, or a list storing k objects, etc.) should perform some non-commutative arithmetic that utilizes each of the k elements
- In Linear Probing (a form of Open Addressing), collisions are resolved by simply shifting over to the next available slot
- In Double Hashing (a form of Open Addressing), collisions are resolved in a way similar to Linear Probing, except instead of only shifting over one slot at a time, the Hash Table has a second hash function that it uses to determine the “skip” for the probe
- In Random Hashing (a form of Open Addressing), for a given key key , a pseudorandom number generator is seeded with key , and the possible indices are given by the sequence of numbers returned by the pseudorandom number generator
- In Separate Chaining (a form of Closed Addressing), each slot of the Hash Table is actually a data structure itself (typically a Linked List), and when a key hashes to a given index in the Hash Table, simply insert it into the data structure at that index

- In Cuckoo Hashing (a form of Open Addressing), the Hash Table has two hash functions, and in the case of a collision, the new key pushes the old key out of its slot, and the old key uses the other hash function to find a new slot

Space Complexity $\mathcal{O}(n)$ — Hash Tables typically have a capacity that is at most some constant multiplied by n (the constant is predetermined)

8.8.1 Linear Probing

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — If all the keys mapped to the same index, we would need to probe over all n elements
- **Insert:** $\mathcal{O}(n)$ — If all the keys mapped to the same index, we would need to probe over all n elements
- **Remove:** $\mathcal{O}(n)$ — If all the keys mapped to the same index, we would need to probe over all n elements

Average-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary
- **Insert:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary
- **Remove:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — No collisions
- **Insert:** $\mathcal{O}(1)$ — No collisions
- **Remove:** $\mathcal{O}(1)$ — No collisions

8.8.2 Double Hashing

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — If we are extremely unlucky, we may have to probe over all n elements
- **Insert:** $\mathcal{O}(n)$ — If we are extremely unlucky, we may have to probe over all n elements
- **Remove:** $\mathcal{O}(n)$ — If we are extremely unlucky, we may have to probe over all n elements

Average-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary
- **Insert:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary
- **Remove:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — No collisions
- **Insert:** $\mathcal{O}(1)$ — No collisions
- **Remove:** $\mathcal{O}(1)$ — No collisions

8.8.3 Random Hashing

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — If each number generated by our generator mapped to an occupied slot, we would need to generate n numbers
- **Insert:** $\mathcal{O}(n)$ — If each number generated by our generator mapped to an occupied slot, we would need to generate n numbers
- **Remove:** $\mathcal{O}(n)$ — If each number generated by our generator mapped to an occupied slot, we would need to generate n numbers

Average-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary (ignoring the time complexity of the pseudorandom number generator)
- **Insert:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary (ignoring the time complexity of the pseudorandom number generator)
- **Remove:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary (ignoring the time complexity of the pseudorandom number generator)

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — No collisions (ignoring the time complexity of the pseudorandom number generator)
- **Insert:** $\mathcal{O}(1)$ — No collisions (ignoring the time complexity of the pseudorandom number generator)
- **Remove:** $\mathcal{O}(1)$ — No collisions (ignoring the time complexity of the pseudorandom number generator)

8.8.4 Separate Chaining

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — If all the keys mapped to the same index (assuming Linked List)
- **Insert:** $\mathcal{O}(n)$ — If all the keys mapped to the same index (assuming Linked List) and we check for duplicates
- **Remove:** $\mathcal{O}(n)$ — If all the keys mapped to the same index (assuming Linked List)

Average-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary
- **Insert:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary
- **Remove:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — No collisions
- **Insert:** $\mathcal{O}(1)$ — No collisions
- **Remove:** $\mathcal{O}(1)$ — No collisions

8.8.5 Cuckoo Hashing

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — Keys can only map to two slots
- **Insert:** $\mathcal{O}(n)$ — If we run into a cycle and bound it by n (otherwise we could face an infinite loop), we rebuild the table in-place
- **Remove:** $\mathcal{O}(1)$ — Keys can only map to two slots

Average-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — Keys can only map to two slots
- **Insert:** $\mathcal{O}(1)$ — The formal proof is too complex for a summary
- **Remove:** $\mathcal{O}(1)$ — Keys can only map to two slots

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — Keys can only map to two slots
- **Insert:** $\mathcal{O}(1)$ — No collisions
- **Remove:** $\mathcal{O}(1)$ — Keys can only map to two slots

8.9 Multiway Trie

Summary Description

- A Multiway Trie is a tree structure in which, for some alphabet Σ ,
 - edges are labeled by a single character in Σ , and
 - nodes can have at most $|\Sigma|$ children
- For a given “word node” in a Multiway Trie, the key associated with the “word node” is defined by the concatenation of edge labels on the path from the root of the tree to the “word node”
- We use k to denote the length of the longest key in the Multiway Trie and n to denote the number of elements it contains

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(k)$ — Perform a $\mathcal{O}(1)$ traversal for each of the key’s k letters
- **Insert:** $\mathcal{O}(k)$ — Perform a $\mathcal{O}(1)$ traversal for each of the key’s k letters
- **Remove:** $\mathcal{O}(k)$ — Perform the find algorithm, and then remove the “word node” label from the resulting node

Average-Case Time Complexity

- **Find:** $\mathcal{O}(k)$ — If all keys are length k , then we do $\frac{k+k+\dots+k}{n} = \frac{nk}{k} = k$ on average, translating to $\mathcal{O}(k)$. If only one key is length k , even if all other keys are length 1, then we do $\frac{1+1+\dots+1+k}{n} = \frac{n1+k}{n} = 1$ for $n >> k$, translating to $\mathcal{O}(1)$. If the lengths of keys are distributed uniformly between 1 and k , then the expected length of a key is $\frac{k}{2}$, translating to $\mathcal{O}(k)$
- **Insert:** $\mathcal{O}(k)$ — Same proof as average-case find
- **Remove:** $\mathcal{O}(k)$ — Same proof as average-case find

Best-Case Time Complexity

- **Find:** $\mathcal{O}(k)$ — If all keys are length k (otherwise, it will simply be the length of the query string)
- **Insert:** $\mathcal{O}(k)$ — If all keys are length k (otherwise, it will simply be the length of the query string)
- **Remove:** $\mathcal{O}(k)$ — If all keys are length k (otherwise, it will simply be the length of the query string)

Space Complexity $\mathcal{O}(|\Sigma|^{k+1})$ — If we were to have every possible string of length k , the first level of our tree would have 1 node, the next level would have $|\Sigma|$ nodes, then $|\Sigma|^2$, etc., meaning our total space usage would be $(\sum_{i=1}^{k+1} |\Sigma|^i) = |\Sigma| \left(\frac{|\Sigma|^{k+1} - 1}{|\Sigma| - 1} \right) \approx |\Sigma|^{k+1}$

8.10 Ternary Search Tree

Summary Description

- A Ternary Search Tree is a trie in which each node has at most 3 children: a *middle*, *left*, and *right* child
- For every node u ,
 - the *left* child of u must have a value *less* than u ,
 - the *right* child of u must have a value *greater* than u , and
 - the *middle* child of u represents the next character in the current word
- For a given “word node,” define the path from the root to the “word node” as *path*, and define S as the set of all nodes in *path* that have a middle child also in *path*. The word represented by the “word node” is defined as the concatenation of the labels of each node in S , along with the label of the “word node” itself

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(n)$ — If we insert the elements in ascending/descending order, we effectively get a Linked List
- **Insert:** $\mathcal{O}(n)$ — If we insert the elements in ascending/descending order, we effectively get a Linked List
- **Remove:** $\mathcal{O}(n)$ — If we insert the elements in ascending/descending order, we effectively get a Linked List

Average-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Insert:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary
- **Remove:** $\mathcal{O}(\log n)$ — The formal proof is too complex for a summary

Best-Case Time Complexity

- **Find:** $\mathcal{O}(k)$ — If our query, whose length is k , was the first word that was inserted into the tree
- **Insert:** $\mathcal{O}(k)$ — If our new word, whose length is k , is a prefix of the first word that was inserted into the tree
- **Remove:** $\mathcal{O}(k)$ — If our query, whose length is k , was the first word that was inserted into the tree

Space Complexity $\mathcal{O}(n)$ — The formal proof is too complex for a summary

8.11 Disjoint Set (Up-Tree)

Summary Description

- The Disjoint Set ADT is defined by the “union” and “find” operations:
 - “union” merges two sets
 - “find” returns which set a given element is in
- We can implement the Disjoint Set ADT very efficiently using Up-Trees
- Under Union-by-Size (also known as Union-by-Rank), when you are choosing which sentinel node to make the parent of the other sentinel node, you choose the sentinel node whose set contains the most elements to be the parent
- Under Union-by-Height, when you are choosing which sentinel node to make the parent of the other sentinel node, you choose the sentinel node whose height is larger to be the parent
- Under Path Compression, any time you are performing the “find” operation to find a given element’s sentinel node, you keep track of every node you pass along the way, and once you find the sentinel node, directly connect all nodes you traversed directly to the sentinel node
- Even though Union-by-Height is slightly better than Union-by-Size, Path Compression gives us the biggest bang for your buck as far as speed-up goes, and because tree heights change frequently under Path Compression (thus making Union-by-Height difficult to perform), we typically choose to perform Union-by-Size
- In short, most Disjoint Sets are implemented as Up-Trees that perform Path Compression and Union-by-Size

Worst-Case Time Complexity

- **Find:** $\mathcal{O}(\log n)$ — Under Union-by-Size (the formal proof is too complex for a summary)
- **Union:** $\mathcal{O}(\log n)$ — Under Union-by-Size (the formal proof is too complex for a summary)

Amortized Time Complexity

- **Find:** almost $\mathcal{O}(1)$ — Technically the inverse Ackermann function, which is a small constant for all practical values of n (the formal proof is too complex for a summary)
- **Union:** almost $\mathcal{O}(1)$ — Technically the inverse Ackermann function, which is a small constant for all practical values of n (the formal proof is too complex for a summary)

Best-Case Time Complexity

- **Find:** $\mathcal{O}(1)$ — Our query is a sentinel node
- **Union:** $\mathcal{O}(1)$ — Our queries are both sentinel nodes

Space Complexity $\mathcal{O}(n)$ — Each element occupies exactly one node, and each node occupies $\mathcal{O}(1)$ space

8.12 Graph Representation

Summary Description

- A Graph is simply a set of nodes (or “vertices”) V and a set of edges E that connect them
- Edges can be either “directed” (i.e., an edge from u to v does not imply an edge from v to u), or “undirected” (i.e., an edge from u to v can also be traversed from v to u)
- Edges can be either “weighted” (i.e., there is some “cost” associated with the edge) or “unweighted”
- We call a graph “dense” if it has a relatively large number of edges, or “sparse” if it has a relatively small number of edges
- Graphs are typically represented as Adjacency Matrices or Adjacency Lists
- For our purposes in this text, we disallow “multigraphs” (i.e., we are disallowing “parallel edges”: multiple edges with the same start and end node), meaning our graphs have at most $|V|^2$ edges

8.12.1 Adjacency Matrix

Worst-Case Time Complexity

- We can **check if an edge exists** between two vertices u and v (and check its cost, if the graph is “weighted”) in $\mathcal{O}(1)$ time by simply looking at cell (u, v) of our Adjacency Matrix
- If we want to **iterate over all outgoing edges** of a given node u , we can do no better than $\mathcal{O}(|V|)$ time in the worst case because we would have to iterate over the entire u -th row of the Adjacency Matrix (which has $|V|$ columns)

Space Complexity $\mathcal{O}(|V|^2)$ — We must allocate the entire $|V| \times |V|$ matrix

8.12.2 Adjacency List

Worst-Case Time Complexity

- We can **check if an edge exists** between two vertices u and v (and check its cost, if the graph is “weighted”) by searching for node v in the list of edges in node u ’s slot in the Adjacency List, which would take $\mathcal{O}(|E|)$ time in the worst case (if all $|E|$ of our edges came out of node u)
- If we want to **iterate over all outgoing edges** of a given node u , because an Adjacency List has direct access to this list, we can do so in the least amount of time possible, which would be $\mathcal{O}(|E|)$ only in the event that all $|E|$ of our edges come out of node u

Space Complexity $\mathcal{O}(|V| + |E|)$ — We must allocate one slot for each of our $|V|$ vertices, and we place each of our $|E|$ edges in their corresponding slot

8.13 Graph Traversal

Summary Description

- In all graph traversal algorithms we discussed, we choose a specific vertex at which to begin our traversal
- For our purposes in this text, we disallow “multigraphs” (i.e., we are disallowing “parallel edges”: multiple edges with the same start and end node), meaning our graphs have at most $|V|^2$ edges

8.13.1 Breadth First Search (BFS)

Summary Description

- In Breadth First Search, we explore the starting vertex, then its neighbors, then their neighbors, etc. In other words, we explore the graph in layers spreading out from the starting vertex
- Breadth First Search can be easily implemented using a Queue to keep track of vertices to explore

Time Complexity $\mathcal{O}(|V| + |E|)$ — We have to potentially visit all $|V|$ vertices and traverse all $|E|$ edges, where each visit/traversal is $\mathcal{O}(1)$.

Space Complexity $\mathcal{O}(|V| + |E|)$ — We might theoretically have to keep track of every possible vertex and edge in the graph during our exploration. If we wanted to keep track of the entire current path of every vertex in our Queue, the space complexity would blow up.

8.13.2 Depth First Search (DFS)

Summary Description

- In Depth First Search, we explore the current path as far as possible before going back to explore other paths
- Depth First Search can be easily implemented using a Stack to keep track of vertices to explore
- Depth First Search can also be easily implemented recursively

Time Complexity $\mathcal{O}(|V| + |E|)$ — We have to potentially visit all $|V|$ vertices and traverse all $|E|$ edges, where each visit/traversal is $\mathcal{O}(1)$.

Space Complexity $\mathcal{O}(|V| + |E|)$ — We might theoretically have to keep track of every possible vertex and edge in the graph during our exploration. Because we are only exploring a single path at a time, even if we wanted to keep track of the entire current path, the space required to do so would only be $\mathcal{O}(|E|)$ because a single path can have at most $|E|$ edges.

8.13.3 Dijkstra's Algorithm

Summary Description

- In Dijkstra's Algorithm, we explore the shortest possible path at any given moment
- Dijkstra's Algorithm can be easily implemented using a Priority Queue, ordered by shortest distance from starting vertex, to keep track of vertices to explore

Time Complexity $\mathcal{O}(|V| + |E| \log|E|)$ — We must initialize each of our $|V|$ vertices, and in the worst case, we will insert (and remove) one element into a Priority Queue (ideally implemented as a Heap) for each of our $|E|$ edges.

Space Complexity $\mathcal{O}(|V| + |E|)$ — We might theoretically have to keep track of every possible vertex and edge in the graph during our exploration.

8.14 Minimum Spanning Tree

Summary Description

- Given a graph, a Spanning Tree is a tree that hits every node
- Given a graph, a Minimum Spanning Tree is a Spanning Tree that has the minimum overall cost (i.e., that minimizes the sum of all edge weights)
- We discussed two algorithms (Prim's and Kruskal's) that can find a Minimum Spanning Tree in a graph equally efficiently: worst-case time complexity of $\mathcal{O}(|V| + |E|\log|E|)$ and space complexity of $\mathcal{O}(|V| + |E|)$

8.14.1 Prim's Algorithm

Summary Description Start with a one-node tree, and repeatedly find a minimum-weight edge that connects a node in the tree to a node that is not in the tree and add the edge to the growing tree. Stop when all vertices are in the tree or you run out of edges.

Time Complexity $\mathcal{O}(|V| + |E|\log|E|)$ — We need to initialize all $|V|$ vertices, and we may need to add each of our $|E|$ edges to a Priority Queue (which should be implemented using a Heap).

8.14.2 Prim's Algorithm

Summary Description Start with a one-node tree, and repeatedly find a minimum-weight edge that connects a node in the tree to a node that is not in the tree and add the edge to the growing tree. Stop when all vertices are in the tree or you run out of edges.

Time Complexity $\mathcal{O}(|V| + |E|\log|E|)$ — We need to initialize all $|V|$ vertices, and we need to sort all $|E|$ edges. Note that the fastest algorithms to sort a list of n elements are $\mathcal{O}(n \log n)$.