Data Structures and Algorithms (w/ Java)

Muchang Bahng

Spring 2023

When a program is build and run, we must worry about two computational overheads:

1. the runtime of the program, which is determined by the efficiency of the algorithm itself
2. the memory overhead of the program, which is determined by the types of data structures used.

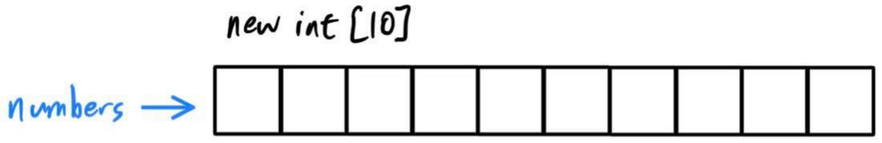
When we want to optimize these programs, we therefore want to look at their data structures and algorithms.

Note that Java has the following **primitive types**: `int`, `long`, `float`, `double`, `boolean`, `char`. Everything else is a **reference type**.

0.1 References and Memory Allocation

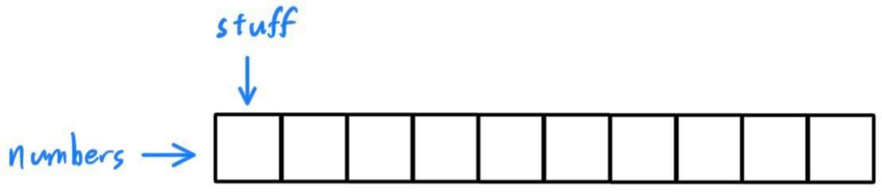
We can think of our RAM as storing our memory. Whenever we call the `new` keyword, we allocate new memory for whatever object we are storing. Say that an `int` stores 4 bytes each, and we create an array of ten integers.

```
int[] numbers = new int[10];
```



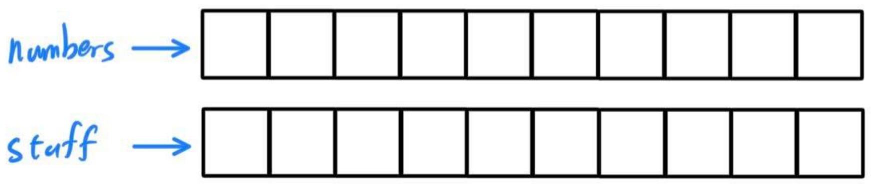
This array would take up 40 bytes of memory, and to access this part of the memory, we create a variable `numbers` that really just stores the location of the object in memory. Since variables are just references to memory locations, we can have multiple variables referencing the same object. Note that we did not use the `new` keyword here.

```
int[] stuff = numbers;
```



Therefore, both `numbers` and `stuff` points to the same array, and whatever we do with one is done to the other. If we wanted to create two distinct arrays in memory, we can do this:

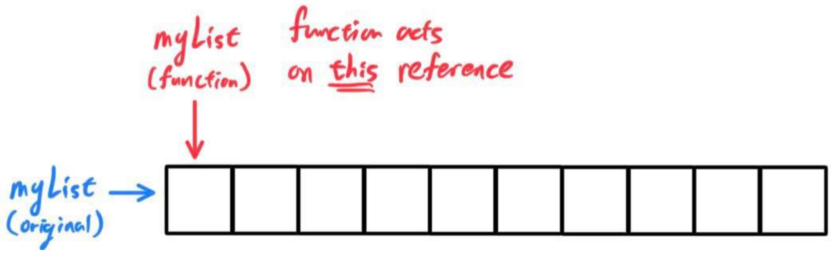
```
int[] numbers = new int[10];
int[] stuff = new int[10];
```



which would take up 80 bytes of memory. Therefore, we must distinguish the *actual object in memory* and the *variable that references the object.* Say that the variable is `myList` and the object that it references to is `ArrayList<String> Object`. Then, we will denote it shorthand as

$$\texttt{myList} \mapsto \texttt{ArrayList<String> Object}$$

This is all simple enough, but it gets a bit more confusing when we talk about how functions act. When we have some function `func` and input variable, say `myList` $\mapsto$ `ArrayList<String> Object`, the function create a *copy of the reference* in the backend, also named `myList` (confusingly) and acts on this copy. Note now that we are working with two references.

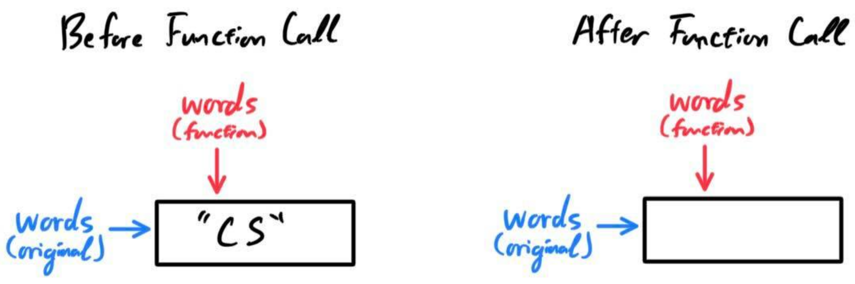


Therefore, the function is allowed to modify `ArrayList<String> Object` through the copied reference variable.

Example 0.1 (Modifying Referred Object). Given the function

```
public static void removeFront(List<String> words) {
    words.remove(0); // this 'words' is a copy of the reference
}

List<String> words = new LinkedList<>();
words.add("CS");
removeFront(words);
System.out.println(words); // prints [] (empty)
```

if `words` references an array of 1 million elements, then `removeFront` will not copy the entire array and allocate new memory, but it will just act on the original array.

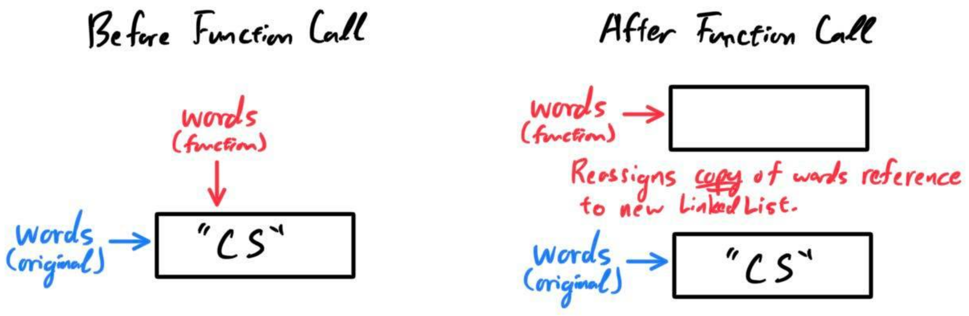


However, it cannot modify the reference of the original variable, and so you cannot "lose" a reference inside a method.

Example 0.2. Given the function

```
public static void tryBreakReference(List<String> words) {
    words = new LinkedList<>(); // this 'words' is a copy of the reference
}

List<String> words = new LinkedList<>();
words.add("CS");
tryBreakReference(words);
System.out.println(words); // Still prints ["CS"]
```

This does not affect the object that the original `words` is referencing to since the function takes only the *copied reference variable* and changes its reference to the new LinkedList.

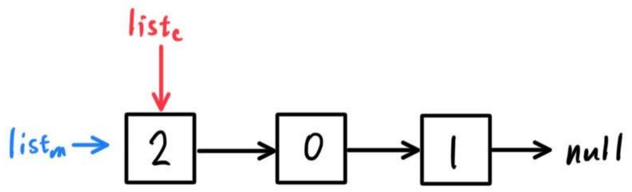


The default value for an uninitialized (no memory allocated by call to `new`) object is `null`. If you try to call any methods to a null object, you will get a **null pointer exception error**.

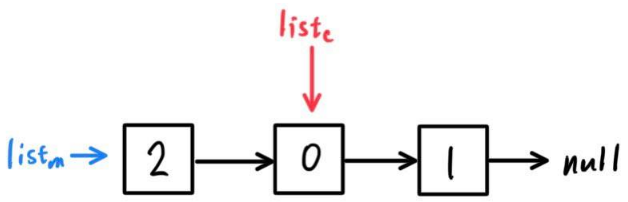
Example 0.3. Let's go through an exercise. Given the code

```
public static ListNode foo(ListNode list) {
    list = list.next;
    list.next = null;
    return list;
}

public static void main(String[] args) {
    ListNode list = new ListNode(2,  new ListNode(0, new ListNode(1)));
    ListNode ret  = foo(list);
    printList(ret);     // prints 0
    printList(list);    // prints 2, 0
}
```

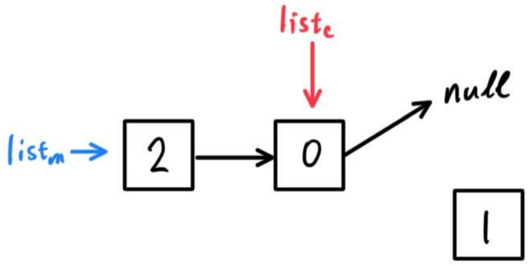
Let us examine the behavior of it. Note that we allocate memory for `list`, which we will denote by the reference $\texttt{list}_m$ and when we input it into `foo`, we get a copy of the reference, denoted $\texttt{list}_c$.



`foo` takes $\texttt{list}_c$ and has it reference the next value, which is 0. Therefore, $\texttt{list}_m$ is $2, 0, 1$ and $\texttt{list}_c$ is $0, 1$.



Now, we set the next value of `list`$_c$ to null, and since this modifies the LinkedList object, we have both `list`$_m$ is $2, 0$ and `list`$_c$ is 0.



0.2 Arrays

Let us take a look at the most basic data structure in Java: the array. They are mutable, homogeneous (can only store one type), and fixed in size. We can print their outputs by converting them to a string, as such with the static method:

```
System.out.println(Arrays.toString(arr));
```

0.3 Classes and Objects

Just like in other object oriented languages, Java uses classes, and everything must be used in a class. Here, we have a class with static and dynamic variables, static and dynamic methods, and constructors.

```
public class Point {
  public static String creator = "Muchang"; // static variable

  public int x; // dynamic variable
  public int y; // dynamic variable

  public Point (int x, int y) { // constructor
    this.x = x;
    this.y = y;
  }

  public Point (Point p) { // constructor
    this.x = p.x;
    this.y = p.y;
  }

  public double distanceFrom(Point other) { // dynamic method
    return Math.sqrt((x - other.x)*(x - other.x) + (y - other.y)*(y - other.y));
  }

  public static void greet() { // static method
    System.out.println("I am a point!");
  }
}
```

0.3.1 Public vs Private Variables and Attributes

Note that every single variable and method had the `public` keyword, which allows users to read/modify the variables and run the methods, *even outside the class it is defined in.* If we switched them to `private`, then we could still access them within the code, but we would not be able to read/modify them elsewhere. This is particularly useful for when we are dealing with sensitive information, and if we do want to grant users the power to read/modify them, we can use separate public getter and setter methods.

```
public class Point {
  private String name;
  public int x;
  private int y;

  public Point (String name, int x, int y) {
    this.name = name;
    this.x = x;
    this.y = y;
  }

  public getName() {    // getter method
    return this.name;
  }

  public setName(String newName) {  // setter method
    this.name = newName;
  }

  private void increment() {
    this.x ++;
    this.y ++;
  }

  public static void main(String[] args) {
    Point p = new Point(1, 2);
    System.out.println(p.x);
    System.out.println(p.y); // still accessible since within the class
    p.increment(); // still runnable since within the class
    System.out.println(p.x);
    System.out.println(p.y);
  }
}
```

0.3.2 Inheritance

0.3.3 .equals() method

Generally for objects, we should use the `.equals()` method. It must be implemented for the given class. If it is not implemented, then the `.equals()` just checks memory locations, and so calling `p = new Point(0.0, 0.0)` and `q = new Point(0.0, 0.0)` and comparing them with `p.equals(q)` would return false. When we also create an array of a certain object, we must create the actual objects in the array by calling new. For example, we don't even create the Point objects in an initialized array.

```
Point[] pointArray = new Point[5];  // creates the array and allocates memory
```

```
System.out.println(pointArray[0].x) // but didn't actually create the points, so error
```

We could also store the same references to the variables.

```
ArrayList<Point> myPoints = new ArrayList<>();
Point p = new Point(2.0, 2.0);
myPoints.add(p);
p.x = 3.4;
myPoints.add(p);

// This creates the ArrayList of form (p, p), which both reference
// the same Point object, so both of form (3.4, 2.0), (3.4, 2.0).
```

So, we should call new for every Point object.

```
ArrayList<Point> myPoints = new ArrayList<>();
for (int i = 0; i < 10; i++) {
    myPoints.add(new Point(0.0, 0.0));
}
Point p = new Point(0.0, 0.0);
System.out.println(pointList.contains(p));
```

The final line can print either true or false:

1. It prints false if there is no .equals() method implemented in the Point class. `.contains()` uses the `.equals()` method on every element of the ArrayList by running a for loop, so it will use the default implementation of checking references.
2. If `.equals()` is implemented as before, it will check the values of x and y, so it will print true.

1 Collections Interface

Interfaces in Java specify functionality by specifying what methods exist. At almost the top, we have the **collections** interface, which represents "a group of stuff." It is divided into 3 main subinterfaces: Lists, Sets, and Maps (along with many others), which add more functionality. So, an interface represents the functionality of whatever we will create, while the implementation is the actual concrete class.

1.1 List Interface

We want to define a list that has the following methods given some list `lst`.

1. `Object y = lst.get(0)` outputs the 0th element.
2. `boolean y = lst.contains(elem)` checks if `elem`
3. `lst.add(Object elem)` adds the element to the end of the list.
4. `lst.remove(Object elem)` removes the element.
5. `lst.size()` returns size of the list.

Definition 1.1 (ArrayList)**.** An ArrayList just implements an array in the backend but with some extra systematic way to dynamically grow. If we add to an array, we either have space and can do it, or we don't and can't. If we add to an ArrayList, we can

1. simply add to the first open position if there's space left, $O(1)$
2. we grow the size of the ArrayList by creating a new larger array, copying everything, and then adding to the first open position. (linear time $O(n)$), since we have to add all the elements to the new array.

Starting with a length 1 array, if we add N elements one at a time and when full, create a new array that is

1. is twice as large (geometric growth: 1, 2, 4, 8, 16, ...). Then, we must copy at sizes 1, 2, 4, 8, ... and the total values copied looks like

$$1 + 2 + 4 + \ldots + (N/4) + (N/2) = N - 1$$

This is what the Java.util.ArrayList implements, and you can see the performance of adding is $O(N)$.

2. has 1 more position (arithmetic growth: 1, 2, 3, ...). Then, we must copy at sizes 1, 2, 3, 4, ... and the total values copied looks like

$$1 + 2 + 3 + \ldots + (N - 1) = N(N - 1)/2$$

If the arithmetic pattern is adding, say 1,000,000 elements, then we are wasteful of memory.

This geometric growth is a good tradeoff between performance and memory usage. It never uses more than twice the memory of an array in order to store it. Furthermore, the runtime of a geometric growth pattern is amortized constant time, which means that it is constant when averaged over a long time. This is because the vast majority of these operations are constant time, with a few add operations which require resizing to be longer. But these few ones happen less and less frequently that when averaged over a long period, we can treat it as constant.

One thing to note is that while adding to the end of an ArrayList can be efficient, adding to the front is not since it must shift the entire Array, even if there is space left.

Theorem 1.1 (ArrayList Runtime Complexity)**.** *The following are true for ArrayList `lst`:*

1. *Getting and Contains*
   (a) *`lst.get(int index)` is $O(1)$.*
   (b) *Getting every element is $O(n)$*
   (c) *`lst.contains(Object elem)` is $O(n)$*
2. *Adding*
   (a) *`lst.add(Object elem)` is amortized $O(1)$.*
   (b) *`lst.add(0, Object elem)` is $O(n)$.*
   (c) *`lst.add(int index, Object elem)` is on average $O(n)$.*
3. *Removing*
   (a) *`lst.remove(0)` is $O(n)$*
   (b) *`lst.remove(int index)` is $O(n)$*
   (c) *`lst.remove(lst.size() - 1)` is $O(1)$*
   (d) *`lst.remove(Object elem)`*

Proof. Listed.

1. Getting the element at index `index` requires us to just look at the same index in the underlying array, which is $O(1)$.
2. We loop through each element of the ArrayList and call `.equals(elem)` at each step, which results in $O(1)$.
3. Since the geometric growth of the ArrayList happens exponentially less frequently, it averages out to be $O(1)$, so amortized.

4. Adding at a specific index requires $O(n)$ since we create a new ArrayList and copy over all the elements with the added element.
5. Removing an object requires us to shift the indices of the remaining elements by 1, so this is $O(n)$.

■

Example 1.1 (String)**.** The string type is just an ArrayList of characters. It has the following attributes and methods. Let `x = "I love CS201"`

1. `int y = x.length()` outputs the length and is $O(1)$
2. `char y = x.charAt(0)` outputs a character and is $O(1)$
3. `String y = x.substring(0, 4)`
4. `boolean y = x.equals("I love CS201")`
5. `String y = x + "!!"`
6. `String[] y = x.split(" ")`
7. `String y = String.join(" ", words)`

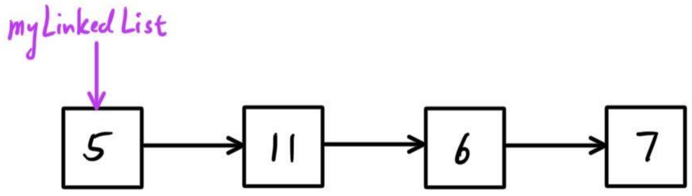
Definition 1.2 (Linked List)**.** A linked list contains a sequence of nodes that each contain an object for its element and a reference to the next node. More specifically, it can be divided up into 3 parts:

1. The variable which points to the *first* node. This can be confusing since this variable, which represents an *entire list* is just a pointer to the first node.
2. A sequence of nodes containing the element and a reference to the next node.
3. The final node containing the element and a reference to `null`.

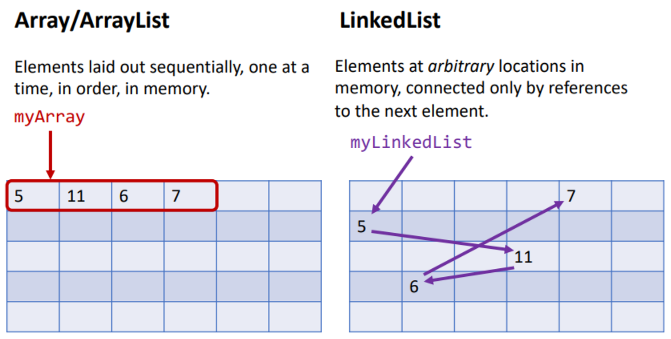
We can implement these functionalities in the `ListNode` class, which are used to build a linked list of integers.

```
public class ListNode {
    int info;           // value i.e. element
    ListNode next;      // reference to next ListNode

    ListNode(int x) {
        info = x;
    }
    ListNode(int x, ListNode node) {
        info = x;
        next = node;
    }
}
```

The following diagram represents a linked list.



But in reality, the elements are all located random in memory and can only be found by references.


Array/ArrayList
Elements laid out sequentially, one at a time, in order, in memory.
myArray
5
11
6
7
LinkedList
Elements at *arbitrary* locations in memory, connected only by references to the next element.
myLinkedList
7
5
11
6

To print everything in a linked list, we just loop over the nodes as long as the nodes are not null.

```
public static void printList(ListNode list) {
    while (list != null) {          // common conditional for traversing
        System.out.println(list.info);
        list = list.next;
    }
}
```

Theorem 1.2 (Linked List Runtime Complexity). *The following are true for a basic LinkedList* `lst`*:*

1. *Getting and contains*
   (a) `lst.get(int index)` *is $O(n)$ on average (unless you get the first index, which is fast).*
   (b) *Getting every element in the list is $O(n^2)$.*
   (c) `lst.contains(Object elem)` *is $O(n)$*
2. *Adding*
   (a) *Start:* `lst.add(0, Object elem)` *is $O(1)$*
   (b) *Middle:* `lst.add(int index, Object elem)` *is on average $O(n)$.*
   (c) *End:* `lst.add(Object elem)` *is $O(n)$*
3. *Removing*
   (a) *Start:* `lst.remove(0)` *or* `lst.remove()` *is $O(1)$*
   (b) *Middle:* `lst.remove(int index)` *is on average $O(n)$*
   (c) *End:* `lst.remove(lst.size() - 1)` *is $O(n)$*

Proof. Listed.

1. We must traverse from the beginning of the linked list, and so it is $O(n)$. If we just pay attention to the first (or last, for doubly-linked list) element, then this is just $O(1)$.
2. Getting every element is just looping an $O(n)$ operation n times, so $O(n^2)$.
3. You need to iterate through each element and call `.equals(elem)`, so it is $O(n)$.
4. We can simply take the reference
5. Adding

∎

Even though our basic LinkedList solves the problem of adding in the beginning, in order to add in the middle or end, we must get to that position (which is $O(n)$ time) before we are able to utilize our $O(1)$ add. This is quite inefficient, especially when we do repeated adding, so we should keep track of certain "markers" that indicate where our current node is. **Iterators** do this naturally, so we would like to implement some current notion of position.Below we implement a new linked list (of integers).

Definition 1.3 (Iterator)**.** An **iterator** is a Java interface that has the two methods:

1. `.hasnext()` checks if there is an element after the current one.
2. `.next()` prints out the next element.

We want to implement iterators to any collections or whatever custom class if we want to be able to use enhanced for loops over them.

Definition 1.4 (DIYLinkedList)**.** Note the following:

1. Adding to the end (`add`) and to the front (`addtoFront`) are both $O(1)$, since we always have access to the dynamic attributes `first` and `last`.

```
public class DIYLinkedList implements Iterable<Integer> {
    private class ListNode {
        int value;
        ListNode next;
        public ListNode(int value) {
            this.value = value;
        }
        public ListNode(int value, ListNode next) {
            this.value = value;
            this.next = next;
        }
    }

    private ListNode first;
    private ListNode last;
    private int size;

    public DIYLinkedList() {
        size = 0;
    }

    public int size() {
        return size;
    }

    public int get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        current = first;
        for (int i = 0; i < index; i++) {
            current = current.next;
        }
        return current.value;
    }

    public void add(int elem) {
        // add to end
        if (last == null) {
            last = new ListNode(elem);
            first = last;
        }
        else {
            last.next = new ListNode(elem);
            last = last.next;
        }
        size++;
    }

    public void addToFront(int element) {
        // add to front
        first = new ListNode(element, first);
        size++;
    }

    private class DIYListIterator implements Iterator<Integer> {
        ListNode current = first;

        @Override
        public boolean hasNext() {
            return current != null;
        }

        @Override
        public Integer next() {
            int value = current.value;
            current = current.next;
            return value;
        }
    }

    @Override
    public Iterator<Integer> iterator() {
        return new DIYListIterator();
    }
}
```

Theorem 1.3 (Appending). *Appending two ListNodes (of size n and m) is $O(n)$ time.*

```
public static ListNode append(ListNode listA, ListNode listB) {
    ListNode first = listA;
    while (listA.next != null) {
        listA = listA.next;
    }
    listA.next = listB;

    return first;
}
```

Theorem 1.4 (Reversing). *When we reverse a linked list, we want to work with it one step at a time by establishing a **loop invariant**, which is just some condition that we want to be true every iteration. In this case, our invariant is "after k iterations, `rev` points to the reverse of the first k nodes."*

```
public ListNode reverse(ListNode front) {
    ListNode rev = null;
    ListNode list = front;
    while (list != null) {
        ListNode temp = list.next;
        list.next = rev;
        rev = list;
        list = temp;
    }
    return rev;
}
```

Example 1.2. Here are three reversing examples, in increasing difficulty:

1. If `front` is a ListNode with `front.next == null`, then `reverse(front)` will return

   $$\texttt{reverse(front)} \mapsto \texttt{front} \mapsto \texttt{null}$$

2. If we have a linked list `list` $\mapsto 1 \mapsto 2 \mapsto 3 \mapsto$ `null`, then `reverse(list.next)` will return

   $$\texttt{reverse(list.next)} \mapsto 3 \mapsto 2 \mapsto \texttt{null}$$

3. If we have a linked list `list` $\mapsto 1 \mapsto 2 \mapsto 3 \mapsto$ `null`, then after running `reverse(list.next)`, the original list variable will be

   $$\texttt{list} \mapsto 1 \mapsto 2 \mapsto \texttt{null}$$

   This is because after the method call, we have $3 \mapsto 2 \mapsto$ `null`, but the 1 still points to 2! Therefore, the original list, which points to 1, will point to 2, which points to `null`.

1.2 Set Interface

Sets are collections that are unordered and store unique elements. We want to define a set that has the following methods given some set `st`.

1. `boolean y = st.contains(elem)` checks if `elem` is in set.
2. `st.add(Object elem)` adds element and returns false if already there.
3. `st.remove(Object elem)` removes element.
4. `st.size()` returns size of the list.

We can loop over a set not with a regular for loop, but with an enhanced for loop (since sets implement iterables but are not ordered by index).

Theorem 1.5. *We can also convert between lists and sets by taking an empty ArrayList and using the .addAll() method.*

```
List<String> myList = new ArrayList<>();
myList.addAll(mySet);
```

Definition 1.5 (HashSet)**.** The HashSet implementation offers constant time performacne for basic operations (add, remove, contains, size), under some assumptions. To count unique words in an Array of Strings, using a HashSet is much much faster than using ArrayLists, since the ArrayList code uses the contains function, which itself is linear.

1.3 Map Interface

Maps pair keys with values, like a dictionary. We want to define a map that has the following methods given some map `mp`.

1. `boolean y = mp.containsKey(k)` checks if `k` is a key in the map.
2. `mp.get(Object k)` returns the value associated with key `k`.
3. `mp.put(Object k, Object v` adds the key value pair `k`, `v`.

We can also loop over a map with an enhanced for loop.

Definition 1.6 (Hash Tables)**.** A hash table is an array of key value pairs. But rather than adding to positions in order from 0, 1, 2, ..., we will calculate the hash of the key, which would return an int that specifies where we store this key-value pair. So to store, `<"ok", 8>`, we will calculate `hash("ok") == 5` and store it in the 5th index.

```
0
1
2 <"hi", 5>
3
4
5 <"ok", 8>
6
7
```

We can immediately see how this makes search easier, since if we want to find the value associated with the key "ok", then we can calculate the hash of it to find the index and look it up on the array. Java implements this with the `.hashCode()` method on the key. More specifically, to get this index, we get the hash, we calculate `Math.abs(key.hashCode()) % list.size()` (remember to take the modulus to get the index between 0 and the list size).

1. To `put(key, value)`, we compute `hash(key)` and add it in that index. If there is already a key-value pair there, then update the value.
2. To `get(key)`, we compute `hash(key)` and retrieve it from the index.
3. To `containsKey(key)`, we compute `hash(key)` and check if the key exists at the index hash position of the list.

So running `get(key)` on a HashMap looks up position `hash(key)` in the hash table and returns the value there. Immediately, we see that if `hash` is not injective (which it isn't), then we can run into collisions. This is solved using chaining or bucketing. Bucketing basically takes each index in the array and stores not just one key-value pair, but a list of key-value pairs. So basically, when we want to search for the value of a key, we compute the index of it with `hash(key)`, which would return a list of key-value pairs. Java would iterate through these keys and call the `.equals()` method to compare them. This can be a problem if you are dealing with keys that are custom built classes.

1. This means that if the key types are something that you've custom built, then you must define the `.equals()` methods in them! You must override and implement `.equals()`.
2. We must also custom implement the `.hashCode()` method.

This is optional, but we can override the `toString()` method to print something we want. Obviously, if we create a custom `hashCode()` method that trivially maps to 0, then we would just have one giant list in the bucket at index 0, which is no more efficient than a list search. So, we should ideally assume that given N pairs with M buckets, our hashing function is built so that the probability of two random (unequal keys) hash to the same bucket is 1/M. Note that this hash function is completely deterministic. We should talk about runtime/memory tradeoff. Given N pairs and M buckets (with SUHA):

1. $N >> M$ means too many pairs in too few buckets, so runtime inefficient
2. $M >> N$ means too many buckets for too few pairs, so memory wasteful
3. M slightly larger than N is the sweet spot.

To maintain an ideal ratio, we basically create a new larger table (with geometric resizing) and rehash/copy everything until we reach it.

2 Algorithms

2.1 Iteration vs Recursion

There are two general paradigms in which we can code algorithms. So far, we have taken the iterative approach, which works with functions that simply loop over a collection.

Example 2.1 (Iteration)**.** The following is an iterative method to count all the elements in a LinkedList.

```
public int countIter(ListNode list) {
    int total = 0;
    while (list != null) {
        total++;
        list = list.next;
    }
    return total;
}
```

A recursive approach of constructing the same function requires us to do two steps:

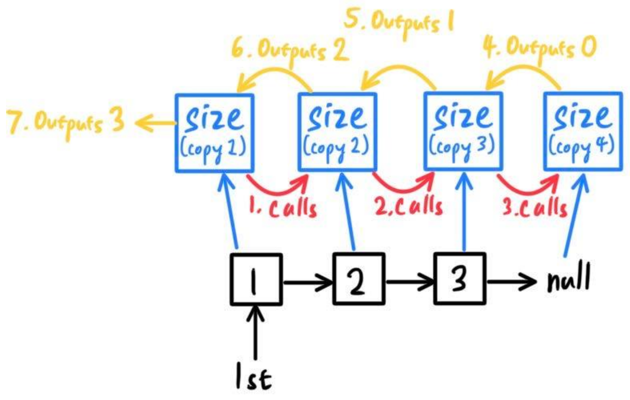
1. Consider the base case, like in an induction proof.
2. Assuming that the function can be solved for some subset of the input (what Fain refers to as ”the oracle”), determine what we should do with the result of the recursive call.

An important thing to note is that the method does not call itself. It calls an identical clone, with its own state, which makes up what we call the **call stack**. Each local call gets its own call frame, with its own local variables, etc., and invoking the method does not resume until invoked method returns, a condition we call **eager evaluation**.

Example 2.2 (Recursion)**.** The following is a recursive method to count all the elements in a linked list.

```
public int size(ListNode list) {
    if (list == null) return 0;
    return 1 + size(list.next);
}
```

Doing this on, say a linked list `lst` $\mapsto 1 \mapsto 2 \mapsto 3 \mapsto$ `null` gives us the following diagram:



Note that we must ensure that every recursive call gets closer to the base case, or this may never end. Speaking of runtime, let us state a method to compute the runtime complexity of recursive algorithms.

Theorem 2.1 (Runtime Complexity of Recursive Algorithms)**.** *To compute the runtime complexity, we must consider two things:*

1. *At what rate are we approaching the base case?*
2. *How long each recursion takes.*

Once we know these two, we can simply multiply them.

Example 2.3 (Reverse)**.** We can reverse a LinkedList with the following recursive algorithm.

```
public static ListNode reverse(ListNode list) {
    if (list == null || list.next == null) {
        return list;
    }
    ListNode reversedLast = list.next;
    ListNode reversedFirst = reverse(list.next);
    reversedLast.next = list;
    list.next = null;
    return reversedList;
}
```

Example 2.4. The following algorithm

```
public static ListNode rec(ListNode list) {
    if (list == null || list.next == null) {
        return list;
    }
    ListNode after = rec(list.next);
    if (list.info <= after.info) {
        list.next = after;
        return list;
    }
    return after;
}
```

2.2 Sorting

As a start, we can sort only ordered things, so we will be talking about arrays and lists (both ArrayLists and LinkedLists). The `Java.util` implementations sort from least to greatest, sorts in place (i.e. mutates the array rather than creating a new one), and is stable (i.e. does not reorder elements if not needed).

1. `Arrays.sort(int[] x)` is used to sort arrays and is $O(n \log n)$
2. `Collections.sort(List<> x)` is used to sort lists and is $O(n \log n)$.

By definition, sorting requires some sort of order $\leq$ defined on a set, and this order can be implemented through a comparable interface, which has a `a.compareTo(b)` method, which returns a negative integer for $\leq$, 0 for $=$, and a positive integer for $\geq$.

Example 2.5. Since strings implement this interface, we can compare them lexicographically, which is the **natural order** for String objects.

1. `"a".compareTo("b")` returns -1
2. `"b".compareTo("b")` returns 0

3. `"b".compareTo("a")` returns 1

Example 2.6. We can create a custom class of Blobs and compare them by their names.

```
public class Blob implements Comparable<Blob> {
    String name;
    String color;
    int size;

    @Override
    public int compareTo(Blob other) {
        return this.name.compareTo(other.name);
    }
}
```

Therefore, after putting them all into an array, we can call the `Arrays.sort` method to sort them with our custom `compareTo` operator.

```
List<Blob> myBlobs = new ArrayList<>();
myBlobs.add(new Blob("bo", "blue" 4);
myBlobs.add(new Blob("al", "red", 2);
myBlobs.add(new Blob("cj", "green", 1);
myBlobs.add(new Blob("di", "red", 4);

System.out.println(myBlobs);
// [("bo", "blue" 4), ("al", "red", 2), ("cj", "green", 1), ("di", "red", 4)]

Collections.sort(myBlobs);
System.out.println(myBlobs);
// [("al", "red", 2), ("bo", "blue" 4), ("cj", "green", 1), ("di", "red", 4)]
```

If we want to make multiple custom sorting systems that is not the natural order, we have to define a separate helper class implementing a `comparator` interface.

```
public class Blob implements Comparable<Blob> {
    String name;
    String color;
    int size;

    @Override
    public int compareTo(Blob other) {
        return this.name.compareTo(other.name);
    }

    public class BlobComparator implements Comparator<Blob> {
        @Override
        public int compare(Blob a, Blob b) {
            int sizeDiff = a.size - b.size;
            if (sizeDiff != 0) {
                return (-1) * sizeDiff;
            }
            return a.compareTo(b);
        }
    }

    public static void main(String[] args) {
        \\ assume myBlobs already defined

        System.out.println(myBlobs);
        // [("bo", "blue" 4), ("al", "red", 2), ("cj", "green", 1), ("di", "red", 4)]

        Collections.sort(myBlobs, new BlobComparator()); // custom sorting
        System.out.println(myBlobs);
        // [("bo", "blue" 4), ("di", "red", 4), ("al", "red", 2), ("cj", "green", 1)]
    }
}
```

In summary, comparables allow you to define an natural ordering, while comparators allow you to define other custom orderings. Furthermore, when comparing two `a` and `b`, comparables are methods on the specific object `a.compareTo(b)`, while comparators are methods on the `Comparator` object `c.compare(a, b)`.

Theorem 2.2 (Comparator Shorthands). *Here are some comparator shorthands:*

1. *To create a comparator that compares according to the natural order, we just do*

```
Comparator<String> c = Comparator.naturalOrder();
c.compare("a", "b") \\ -1
c.reversed().compare("a", "b"); \\ 1
```

2. *To create a comparator that compares according to the length, we just do*

```
Comparator<String> c = Comparator.comparing(String::length);
c.compare("this", "is") \\ 1
c.compare("is", "it") \\ 0
```

3. *If we want to compare according to the length and then the natural order, then we just do*

```
Arrays.sort(arr, Comparator.
    comparing(String::length).
        thenComparing(Comparator.naturalOrder()));
```

Definition 2.1 (Selection Sort). The **selection sort** algorithm is an iterative algorithm with the loop invariant that "on iteration i, the first i elements are the smallest i elements in sorted order." On iteration i, we must find the smallest element from index i onward and swap that with the element at index i.

```
public static void selectSort(int[] ar) {
    for (int i = 0; i < ar.length; i++) {
        int minDex = i;
        for (int j=i+1; j < ar.length; j++) {
            if (ar[j] < ar[minDex]) {
                minDex = j;
            }
        }
        int temp = ar[i];
        ar[i] = ar[minDex];
        ar[minDex] = temp;
    }
}
```

This is $O(n^2)$.

Definition 2.2 (MergeSort Algorithm)**.** An improved version of this algorithm uses a recursive method, which does the steps

1. Take an array split it into two halves.
2. Sort the first half and then sort the second half.
3. Merge the two halves so that the combined total is sorted.

and has the base case that sorting an array of length 1 is just itself. To do this, we must describe the helper function `merge`, which will merge two sorted arrays into a bigger sorted array. We basically want to loop through each indices of each array and add the smaller element to the new bigger array until we've exhausted all elements in one of the arrays. Then, we just copy the rest of the elements in the other array over.

Furthermore, we can create a wrapper function `mergeSort`, which helps us initialize the parameters to the recursive call, allowing for more convenience.

```
public static void mergeSort(int[] ar) {
    mergeHelper(ar, 0, ar.length);
}

public static void merge(int[] ar, int l, int mid, int r) {
    int[] sorted = new int[r - l];
    int sDex = 0; int lDex = l; int rDex = mid;
    while (lDex < mid && rDex < r) {
        if (ar[lDex] <= ar[rDex]) {
            sorted[sDex] = ar[lDex];
            lDex++;
        }
        else {
            sorted[sDex] = ar[rDex];
            rDex++;
        }
        sDex++;
    }
    if(lDex == mid) {System.arraycopy(ar, rDex, sorted, sDex, r - rDDex); }
    else {System.arraycopy(ar, lDex, sorted, sDex, mid - lDex); }
    System.arraycopy(sorted, 0, ar, l, r - l);
}

public static void mergeHelper(int[] ar, int l, int r) {
    int diff = r - l;
    if (diff < 2) {return;}     // base case if 0 or 1 elements
    int mid = l + diff/2
    mergeHelper(ar, l, mid);    // recursively sort 1st half
    mergeHelper(ar, mid, r);    // recursively sort 2nd half
    merge(ar, l, mid, r);       // merge the 2 sorted parts
}
```

There will be $O(\log n)$ levels of recursion, and for each recursion we will have to run the `merge` function, which is linear in the number of elements we are sorting ($O(n)$), so the total time complexity of this algorithm is $O(n \log n)$. We can also determine the **recurrence relation** of this algorithm as

$$T(N) = T(N/2) + T(N/2) + O(N) = O(N \log N)$$

2.3 Searching

Definition 2.3 (Binary Search)**.** Given that we have a sorted list (this is important!), we can search for the index of an element in $O(\log n)$ time. We want the loop invariant "if the target is in the array/list, it is in the range [low, high].” Let us have a list of N elements, and at every step, we either

1. get our desired element and its index, or
2. cut down our search space by half

The code can be a bit more general by implementing a generic type `T`.

```
public static <T> int binarySearch(List<T> list, T target, Comparator<T> comp) {
    int low = 0;
    int high = list.size() - 1;
    while (low <= high) {
        int mid = (low + high)/2;   // rounds down since integer division
        T midval = list.get(mid);

        int cmp = comp.compare(midval, target);
        if (cmp < 0)
            low = mid + 1;
        else if (cmp > 0)
            high = mid - 1;
        else
            return mid;         // target found
    }
    return -1;      // target not found
}
```

3 Trees

3.1 Stacks, Queues, and Priority Queues

Definition 3.1 (Stacks)**.** A **stack** is an abstract data structure represented as a **Last-In-First-Out (LIFO) list**, which implements the following methods given stack `st`, which we can initialize with

```
    Stack<String> st = new Stack<>();
    for (String s : strs) { st.push(s); }
    while (! st.isEmpty()){ System.out.println(st.pop()); }
```

1. `st.add(Object element)` adds to the top of the stack, which is $O(1)$
2. `st.pop()` removes the element that is at the top of the stack, which is $O(1)$, and returns whatever is popped out.

Remember that this is just a list and so anything we can do with a stack we can do with a list. What makes the stack so useful is the way the list is implemented. We can literally imagine the elements of this list as ”stack.” If you want to remove something from the stack, of course you have to remove the top element.

Definition 3.2 (Queue)**.** A **queue** is an abstract data structure represented as a **First-In-First-Out (FIFO) list**, which implements the following methods given queue `q`, which we can initialize with

```
    Queue<String> q = new LinkedList<>();
    for (String s : strs) { q.add(s); }
    while (! q.isEmpty()) { System.out.println(q.remove()); }
```

Note that LinkedList implements the Queue interface.

1. `q.add(Object element)` adds to the top of the queue, referred to as **enqueue**.
2. `q.remove()` removes the first element in the queue, referred to as **dequeue**.

This is just like how a queue works. Whatever has been waiting in the queue the longest is the one that is removed first.

For now, we will abstractly think that a priority queue acts as a sorted list (though it is actually implemented as a binary heap).

Definition 3.3 (Priority Queues)**.** A priority queue simply adds things according to their priority. Every time we add an element, it looks at where the element should go to keep the list sorted. If we want to dequeue, then we just remove the first element of the list.

```
PriorityQueue<String> pq = new PriorityQueue<>();
pq.add("ac");
pq.add("c");
pq.add("bdf");
while(! pq.isEmpty()) { System.out.println(pq.remove()); }
// ac   bdf  c
```

But depending on the comparator what we use, the priority queue sorts it in a different manner.

```
PriorityQueue<String> pq = new PriorityQueue<>(Comparator.comparing(String::length));
pq.add("ac");
pq.add("c");
pq.add("bdf");
while(! pq.isEmpty()) { System.out.println(pq.remove()); }
// c   ac   bdf
```

1. `pq.add(Object element)` is $O(\log(N))$
2. `pq.remove()` is $O(\log(N))$
3. `pq.peek()` returns the minimal element and is $O(1)$
4. `pq.size()` returns number of elements and is $O(1)$

3.2 Binary Trees

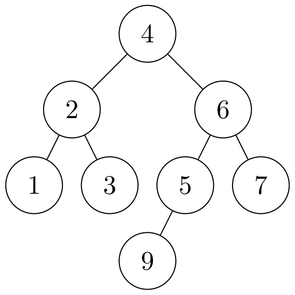
Let us compare the HashSet/Map and the TreeSet/Map. The purpose of Hashing is to "find" and add elements quickly.

1. This means that add, contains, put, and get are all amortized $O(1)$ (under Simple Uniform Hashing Assumption). The TreeSet/Map have all operations add, contains, put, get are $O(\log(N))$, which is slower, but is not amortized.
2. Trees are sorted, while Hashes are not, and so we can get a range of Tree values in sorted order efficiently, but not for Hashes.

A Node for trees is represented with the following class.

```
public class TreeNode {
    TreeNode left;
    TreeNode right;
    String info;

    TreeNode (String s) {
        info = s;
    }

    TreeNode(String s, TreeNode llink, TreeNode rlink) {
        info = s;
        left = llink;
        right = rlink;
    }
}
```

A tree looks pictorially like this:



Some terms:

1. The root of the tree is the top node, which is 4
2. The leaf of the tree are nodes that do not have a left nor right subchild.
3. A path is any path from one node to another node. A simple path is a path that doesn't cross the same edge twice
4. The height of a node is the length of the longest downward path to a leaf from that node.
5. The depth of a node is the number of edges from the root to the node.

3.2.1 Algorithms for Binary Tree

Theorem 3.1 (Print All Nodes in A Binary Tree)**.** *Here are three ways to recursively traverse a tree. The difference is in where the nonrecursive part is. Let us have a binary tree from above.*

1. *This tells us to print everything on the left of the node, then print the node, and then print everything on the right.*

```
void inOrder(TreeNode t) {
    if (t != null) {
        inOrder(t.left);
        System.out.println(t.info);
        inOrder(t.right);
    }
}
// 1, 2, 3, 4, 9, 5, 6, 7
```

2. *This tells us to print the node itself first, then print all the ones on the left, and then print all the ones on the right.*

```
void preOrder(TreeNode t) {
    if (t != null) {
        System.out.println(t.info);
        preOrder(t.left);
        preOrder(t.right);
    }
}
// 4, 2, 1, 3, 6, 5, 9, 7
```

3. *This tells us to print all the nodes on the left, then all ones on the right, and then the node itself.*

```
void postOrder(TreeNode t) {
    if (t != null) {
        postOrder(t.left);
        postOrder(t.right);
        System.out.println(t.info);
    }
}
// 1, 3, 2, 9, 5, 7, 6
```

Theorem 3.2 (Storing All Nodes in a List). *Now if we want to store them all in a list, then this recursive strategy will not work, since if we create a list inside the function body, then we will have a bunch of lists floating around in memory. Therefore, we want to initialize a list outside of the entire function, and store that entire thing within a wrapper function. The* `inOrder` *takes in also a reference to a list that it will be adding to.*

```
public ArrayList<String> visit(TreeNode root) {
    ArrayList<String> list = new ArrayList<>();
    inOrder(root, list);
    return list;
}

private void inOrder(TreeNode root, ArrayList<String> list) {
    if (root != null) {
        inOrder(root.left, list);
        list.add(root.info);
        inOrder(root.right, list);
    }
}
```

Definition 3.4 (Finding Height of Node). The height of a node is the longest downward path to a leaf from that node, so its height would be the maximum of the two heights of its children. A null node would have height -1, which is our base case.

```
public int getHeight(TreeNode root) {
    if (root == null) { return -1; }
    return 1 + Math.max(getHeight(root.left), getHeight(root.right));
}
```

Definition 3.5 (Finding Depth of Node). The depth is quite hard to find recursively, but if we have a reference to the parent, then we can write

```
int depth(TreeNode node) {
    if (node == null) {
        return -1;
    } else {
        return 1 + depth(node.parent);
    }
}
```

3.2.2 Binary Search Tree

Definition 3.6 (Binary Search Tree). A binary tree is a **binary search tree** if for every node, the left subtree values are all less than the node's value, and the right subtree values are all greater than the node's value. That is, the nodes are in order, and if we called `inOrder(root)` on the tree, then we would get a sorted list, which allows for efficient search.

We could then search recursively with the following:

```
public boolean contains(TreeNode tree, String target) {
    if (tree == null) return false;
    int result = target.compareTo(tree.info);
    if (result == 0) return true;
    if (result < 0) return contains(tree.left, target);
    return contains(tree.right, target);
}
```

or iteratively with

```
public static boolean contains(TreeNode node, String target) {
    while (node != null) {
        int comp = node.info.compareTo(target);
        if(comp == 0) return true;
        else if (comp > 0) node = node.left;
        else node = node.right;
    }
    return false;
}
```

Adding elements to a binary search tree is also very similar. But note that the order in which we add elements to the binary search tree will matter, since it can either make the tree **balanced** or **unbalanced**. In a balanced case, contains/add will be $O(\log(N))$, while in an unbalanced case, we will have $O(N)$.

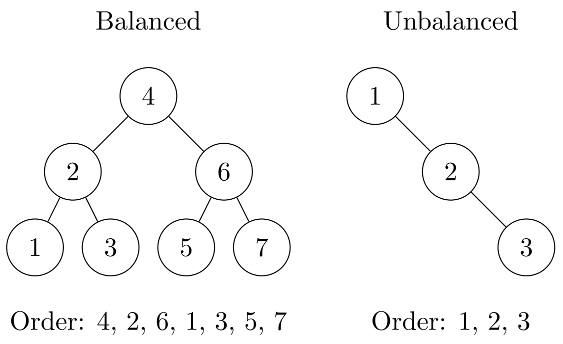


Figure 1: Comparison of balanced and unbalanced binary trees

3.3 Greedy Algorithms, Huffman

A **greedy algorithm** is an approach for solving a problem by selecting the best option available at the moment. They are useful because they may be optimal, they may not be optimal but work very well in practice, and they are easier to implement for starting out. It turns out that many large-scale machine learning models, like neural nets, are optimized using greedy algorithms (e.g. gradient descent).

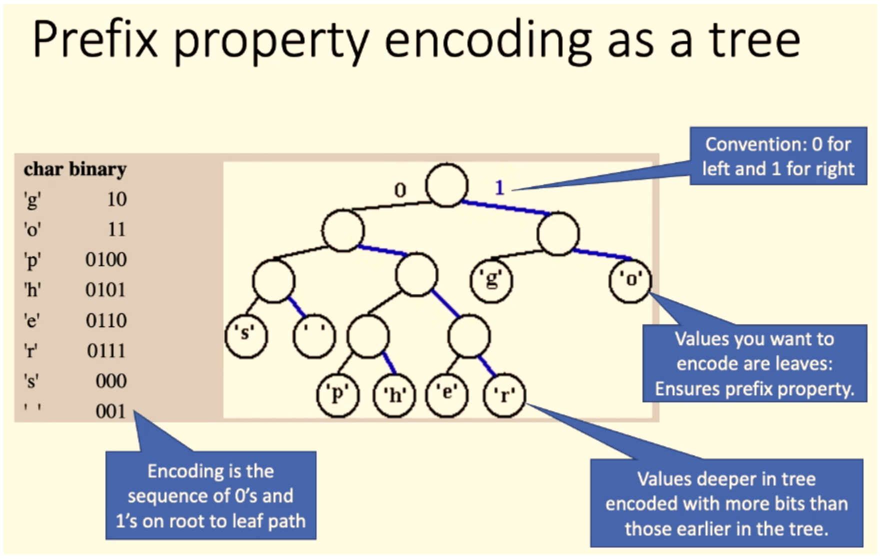
Example 3.1. In the Knapsack problem, we have \$10 and want to buy things at a market that maximizes our value. Say that we have an \$1 apple with value 2, a \$1 banana with value 1, and a \$10 pizza with value 10. Then, a greedy algorithm will make sure to first buy all things with the highest value-to-cost ratio.

Now, let's talk about **Huffman encoding**. We already know about the ASCII encoding that encodes characters in 7 bits, for a total of 2^7 possibilities. The extended ASCII uses 8 bits, but all of these things use something called **fixed-length encoding** which uses a constant number of bits to encode any character. To compress something, we want to use **variable-length encoding**.

To decode something, the mapping from the characters to the bits must be injective, so we can define an inverse over its image. It turns out that if we an encoding for 3 characters, say

$$a \mapsto 1, b \mapsto 10, c \mapsto 11$$

then decoding 1011 is ambiguous since 1 is a prefix of the encoding for c. Therefore, we do not want one encoding to be the prefix of another. It turns out that we can avoid this conflict by encoding everything as a binary tree and setting all encodings as leaf nodes.


Prefix property encoding as a tree
Convention: 0 for left and 1 for right
0
1
's'
' '
'p'
'h'
'e'
'r'
'g'
'o'
Values you want to encode are leaves: Ensures prefix property.
Encoding is the sequence of 0's and 1's on root to leaf path
Values deeper in tree encoded with more bits than those earlier in the tree.

char	binary
'g'	10
'o'	11
'p'	0100
'h'	0101
'e'	0110
'r'	0111
's'	000
' '	001

This makes sense since an encoding will be a prefix of another if and only if it is a parent of another. Furthermore, a greater depth of a character in the tree corresponds to a longer encoding. So, Huffman encoding tries to convert shorter characters to longer leaves and less recurrent characters into longer encodings. To decode a string of bits using a tree, we read the bit at a time to traverse left or right edge. When we reach a leaf, we decode the character and restart at root.

Now, we describe the greedy algorithm for building an optimal variable length encoding tree.

1. We take the document and compute the frequencies of all characters that we want to encode. We want to less frequent characters to be lower on the tree, and so we will build the tree up from the leaves.
2. We iteratively choose the lowest weight nodes to connect up to a new node with weight = sum of children.

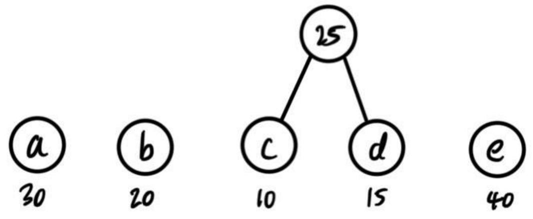
We implement this using a priority queue.

Example 3.2. Let us go do an example of where we have the characters and frequencies

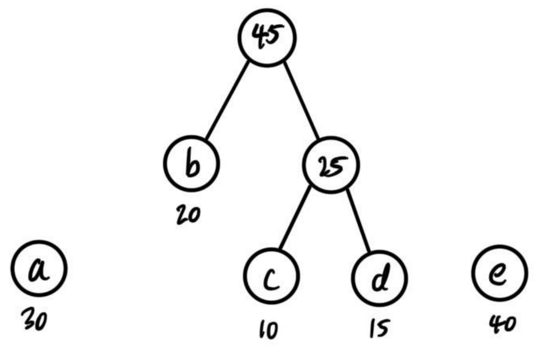
$$a \mapsto 30,\ b \mapsto 20,\ c \mapsto 10,\ d \mapsto 15,\ e \mapsto 40$$

Then, we have the following steps:

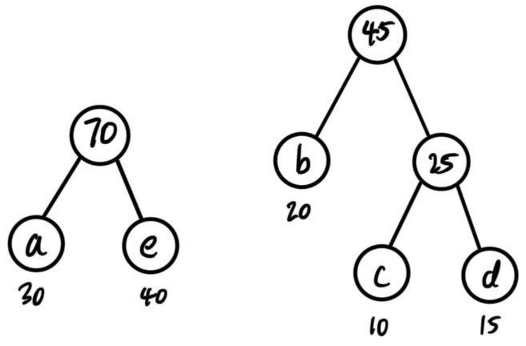
1. We write out them as leaf nodes with the values $30, 20, 10, 15, 40$. We take the smallest of the frequencies and sum them up: $10 + 15 = 25$.



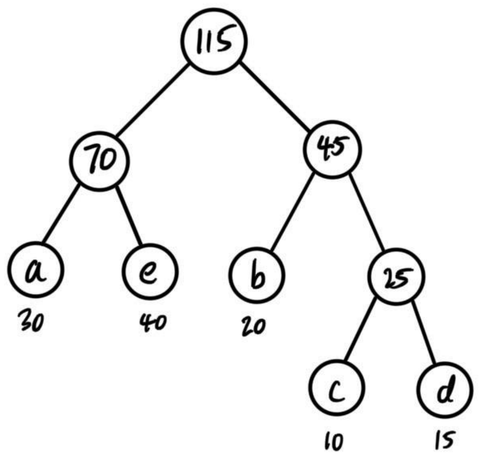
2. We have the values $30, 20, 25, 40$. We sum the smallest two frequencies: $20 + 25 = 45$.



3. We have the values $30, 45, 40$. We sum the smallest two frequencies: $30 + 40 = 70$.



4. We have the values $70, 45$. We sum them up to get the complete tree.



Theorem 3.3. *If we have a document of N total characters and M unique characters, the number of nodes in the Huffman tree, in complexity notation, is*

$$O(M)$$

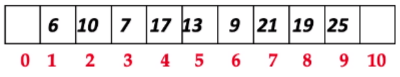
Clearly, this has nothing to do with N. Note that we have M leaf nodes, and in each iteration, we connect 2 nodes up to a parent. Therefore, the number of nodes to connect up decreases by 1 per iteration, and we create a new node per iteration. Since there are $M - 1$ iterations, we add one node, so there will be $M + M - 1 = O(n)$ nodes in the binary tree.

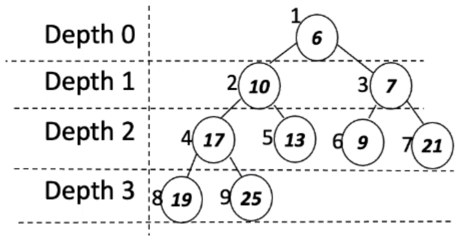
3.4 Binary Heaps

Definition 3.7 (Binary Heap)**.** A **binary heap** is a binary tree satisfying the following structural invariants:

1. Maintain the **heap property** that every node is less than or equal to its successors, and
2. The **shape property** that the tree is complete (full except perhaps last level, in which case it should be filled from left to right.

We should conceptually think of a binary heap as an underlying binary heap, but it is actually usually implemented with an array, and we can create a map from the heap to the array with the following indices.

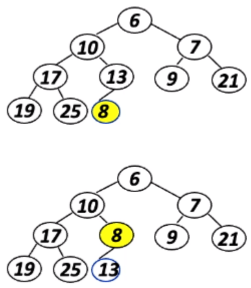


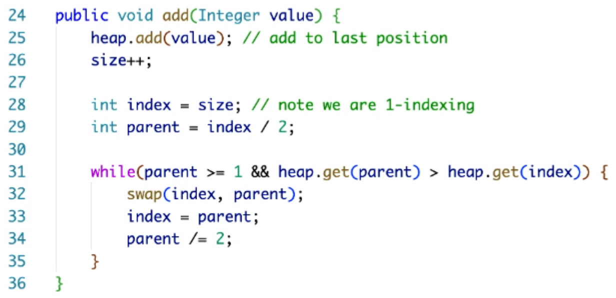


When 1-indexing, for node with index k, the left child is index $2k$, the right child is index $2k + 1$, and the parent is index $k/2$ (where this is integer division).

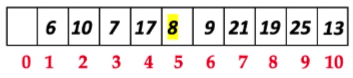
Implementing peek is easy, since we just return the first index, but it can be quite tricky to maintain this invariant after an arbitrary sequence of add/remove operations.

1. To add values to a heap, we add to the first open position in the last level of the tree (to maintain the shape property), and then swap with the parent is the heap property is violated. If we are swapping with the parent at most $\log(N)$ times, then the add property has $O(\log(N))$ complexity.





```java
public void add(Integer value) {
    heap.add(value); // add to last position
    size++;

    int index = size; // note we are 1-indexing
    int parent = index / 2;

    while(parent >= 1 && heap.get(parent) > heap.get(index)) {
        swap(index, parent);
        index = parent;
        parent /= 2;
    }
}
```



2. We remove the first (minimal) value, we first replace the root with the last node in the heap, and while the heap property is violated, we swap with the smaller child. There are two choices, the left or right child, in which we can swap. But we must always swap with the **smaller child**, since we swapped with the bigger child, then this bigger child would be larger than the smaller one, violating the heap property. Since a complete binary tree always has height $O(\log(N))$, remove also "traverses" one root-leaf path, and so its runtime complexity is $O(\log(N))$, too.

3. The decreaseKey operation just takes an arbitrary node and decreases its value to some other integer. In this case, it wouldn't violate the shape property, and to restore the heap property, we just put swap it with its parent if the new value is smaller than its parent, making this operation $O(\log(N))$.

3.5 Red-Black Trees

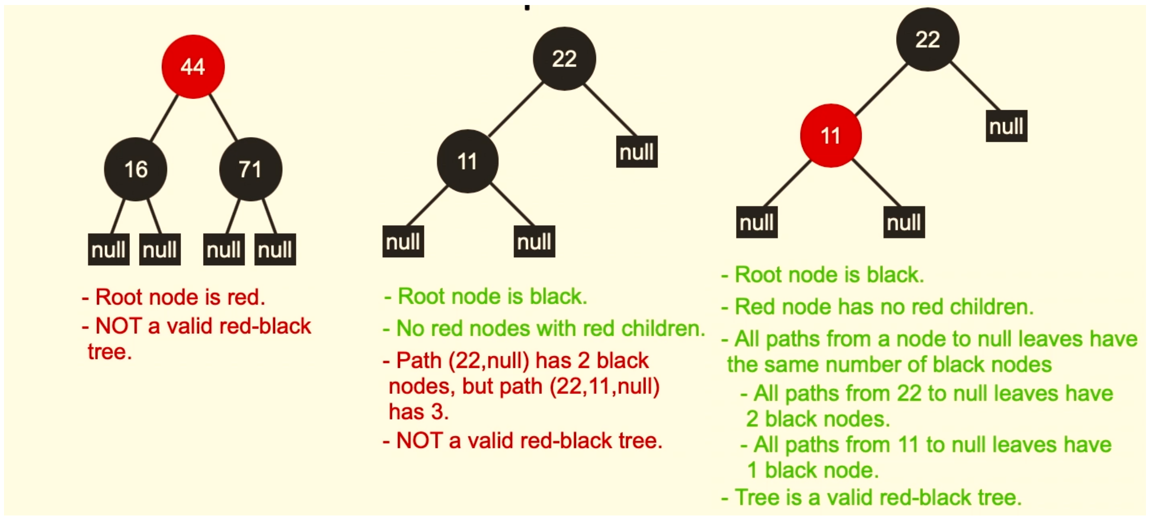
Now, we have learned how we can implement a priority queue using a binary heap. This is also possible to use a binary search tree, since it's easy to get the minimal element for adding and removing, but there are three things that make it difficult:

1. all elements must be unique
2. it is not array-based, and so uses more memory and higher constant factors on runtime
3. it is much harder to implement with guarantees that the tree will be balanced. This makes it difficult since if we want to search through a balanced BST, it is $O(\log(N))$, but if it turns out to be unbalanced, then it is $O(N)$.

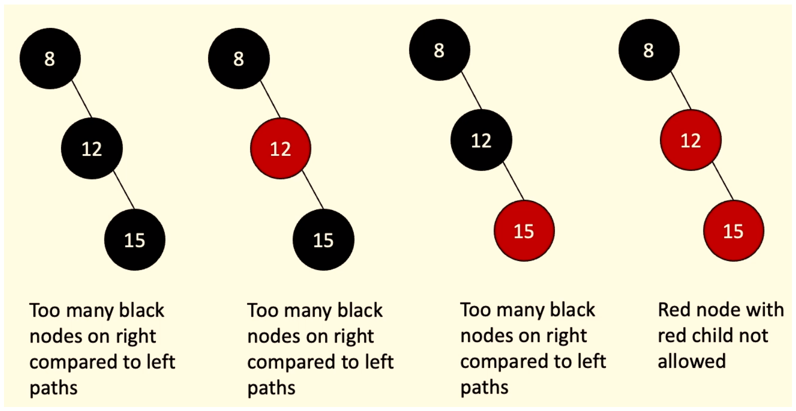
Therefore, while a balanced tree may be efficient on average, in the worst case the linear complexity is not tolerable. Therefore, we must implement a binary search tree that will do extra work to ensure that they are approximately balanced.

Definition 3.8 (Red-Black Tree)**. Red-Black Trees** are binary search trees that satisfy the following properties:

1. Every node is red or black
2. The root is black
3. A red node cannot have red children
4. From a given node, all paths to null descendants must have the same number of black nodes. (Null is considered to be a black node)
5. remember that it must be a binary search tree!


44
16
71
null
null
null
null
- Root node is red.
- NOT a valid red-black tree.
22
11
null
null
null
- Root node is black.
- No red nodes with red children.
- Path (22,null) has 2 black nodes, but path (22,11,null) has 3.
- NOT a valid red-black tree.
22
11
null
null
null
- Root node is black.
- Red node has no red children.
- All paths from a node to null leaves have the same number of black nodes
- All paths from 22 to null leaves have 2 black nodes.
- All paths from 11 to null leaves have 1 black node.
- Tree is a valid red-black tree.

Note that there are binary search trees that cannot be turned into a red-black tree.


8
12
15
Too many black nodes on right compared to left paths
8
12
15
Too many black nodes on right compared to left paths
8
12
15
Too many black nodes on right compared to left paths
8
12
15
Red node with red child not allowed

This is intentional because red-black tree properties guarantee approximate balance. If we can turn a binary search tree into a red-black tree, then it logically follows that the original BST was approximately balanced. Note that a red black tree does not make searching asymptotically faster in any way; it just takes care of the worst-case.

Remember that red black trees are also just binary search trees:

1. contains (search) method is the exact same thing as BST
2. The add method needs to be slightly modified, since after we add, we need to make sure that the resulting tree is a red-black tree. This is done in three steps:
   (a) Run the regular BST add
   (b) Color the new node red
   (c) Fix the tree to reestablish red-black tree properties. This is extremely complicated with different cases, but it all essentially uses some sort of recoloring and a (right or left) rotation of the tree.

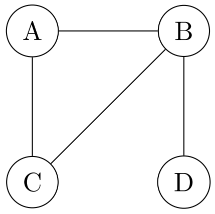
4 Graphs

Definition 4.1 (Graph)**.** A **graph** is a data structure for representing connections among items an dconsists of vertices connected by edges. It consists of a **vertex** (or node) and an **edge**. It can be directed or undirected. A **simple graph** there is at most one undirected edge between nodes (or 2 directed). Therefore, given that N is the number of vertices and M the number of edges, we have

$$M \leq N^2$$

for a simple graph. A **simple path** is a sequence of unique vertices where subsequent nodes are connected by edges, and the path doesn't repeat itself.

Definition 4.2 (Adjacency List/Matrix)**.** Given a graph, we can completely represent it with a list of adjacent vertices for each vertex or an adjacency matrix.



An adjacency list would look something like this

$$\begin{aligned} A &: B, C \\ B &: A, C, D \\ C &: A, B \\ D &: B \end{aligned}$$

and the adjacency matrix looks like this:

$$\begin{array}{c|cccc} & A & B & C & D \\ A & 0 & 1 & 1 & 0 \\ B & 1 & 0 & 1 & 1 \\ C & 1 & 1 & 0 & 0 \\ D & 0 & 1 & 0 & 0 \end{array}$$

Adjacency matrices can be memory wasteful if the graph is sparse, and for fast lookup of information in adjacency lists, we can implement a double hashing mechanism (`hashMap<Vertex, HashSet<Vertex>>`).

4.1 Depth First Search (DFS)

DFS basically traverses

Theorem 4.1 (Recursive DFS). *Now, in order to traverse this graph, we basically want to make an algorithm that starts at a node, prints it value, and then goes to all of its neighbors (which we can access through the adjacency list) to print them out. Thus, this is by nature recursive. We don't want the algorithm to loop around printing nodes infinitely often, so we must create a base case that tells the algorithm to not print out a node. It makes sense to create a set of visited nodes, which we can add to whenever we reach a new node. So, if we ever come onto a node that we have visited, we can just tell the function to do nothing.*

Now if we want to print out all the nodes of a general graph, we can do the following:

```
public static void dfs(char start) {
    if (!visited.contains(start)) {                // if already visited, backtrack
        visited.add(start);                         // else, visit this node
        System.out.println(start);                  // print it out
        for (char neighbor : aList.get(start)) {
            dfs(neighbor);                          // and explore its neighbors
        }
    }
}
```

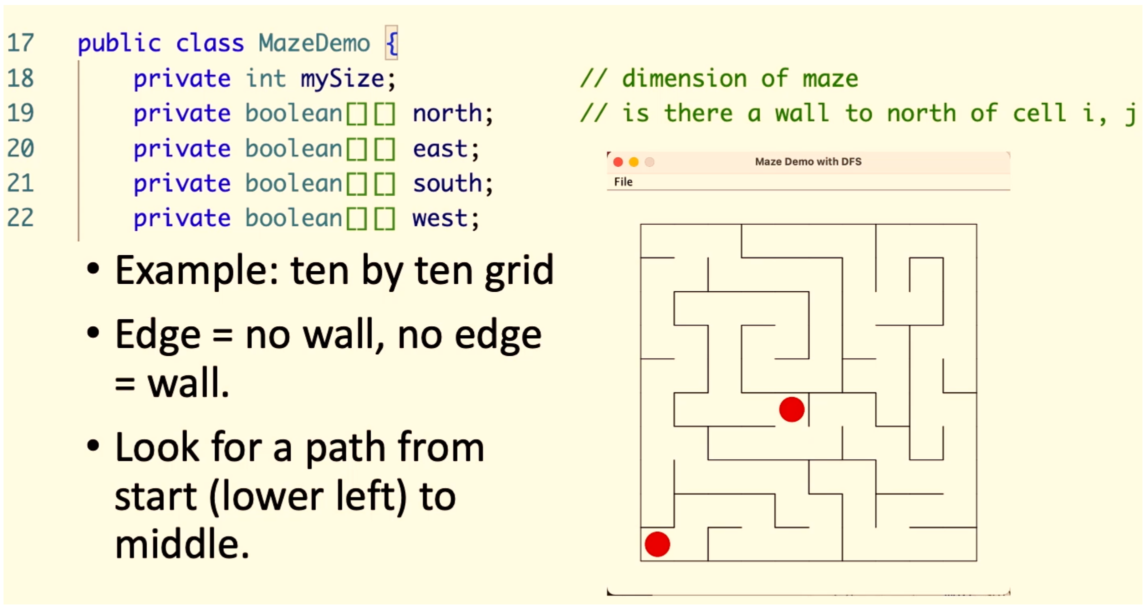
Theorem 4.2 (Iterative DFS). *Though recursion really makes this simple, we can construct an iterative approach that uses stacks. Note that in recursion, we are really making a call stack of different functions. We can be explicit about this by actually implementing a stack, which would store all the nodes that we have discovered, but not yet explored from)i.e. all the current nodes). At each iteration, we would pick a node to continue exploring, and since this is a DFS, we would want to implement a LIFO stack so that the last element we input in is the first thing that we should explore from, i.e. we always explore from the last node discovered.*

```
public static void dfs(char start) {
    Stack<Character> toExplore = new Stack<>();
    char current = start;
    toExplore.add(current);
    visited.add(current);

    while (!toExplore.isEmpty()) {
        current = toExplore.pop();                  // explore from most recently discovered node
        for (char neighbor : aList.get(current)) {  // look at all neighbors of current node
            if (!visited.contains(neighbor)) {      // if we haven't seen them before...
                previous.put(neighbor, current);    // note how we got here
                visited.add(neighbor);              // note that we have seen
```

```
                    toExplore.push(neighbor);          // mark to explore later
                }
            }
        }
    }
```

The runtime complexity of this search is $O(N + M)$ because first, the while loop loops at most over the N nodes. The for loop may loop over M edges, but this is a bit pessemistic in bound. Rather, we can view it as looping over neighbors of each node at most exactly once, and so it considers every edge twice, meaning that the for loop will get called $2M$ times in the entire algorithm. So $N + 2M = O(N + M)$.

Example 4.1 (DFS in a Maze)**.** We can represent a grid graph, like a maze, with a two dimensional array that stores whether it is connected north, east, south, and west, where boolean of true represents that there is a wall, and false means there isn't a wall (so connected).



```
public class MazeDemo {
    private int mySize;              // dimension of maze
    private boolean[][] north;       // is there a wall to north of cell i, j
    private boolean[][] east;
    private boolean[][] south;
    private boolean[][] west;
```

- Example: ten by ten grid
- Edge = no wall, no edge = wall.
- Look for a path from start (lower left) to middle.

But remember that in a tree traversal, we recursively searched down and down until we hit a null node, in which case we backtrack up to look in another branch. For graphs, this is a bit more complicated, since we could go in loops. Therefore, we want to keep track of all the visited nodes to avoid infinite recursion. We have three base cases:

1. If we search off the grid, then this is not a valid path
2. If we already explored here, then we don't want to repeat it
3. If we reached the goal of the maze, then we output the length of the path.

The recursive case would take each node and recurse on its 4 adjacent neighbors, if they are connected. So, the entire function, which takes in a location (x, y) in the grid and the depth that it took to traverse to this point, would look like this:

```
private boolean[][] visited;

private int solveDFS(int x, int y, int depth) {
    if (x == 0 || y == 0 || x == mySize + 1 || y == mySize + 1) {
        return 0;
    }
    if (visited[x][y]) return 0;

    visited[x][y] = true;

    if (x == mySize/2 && y == mySize/2) {
        return depth;
    }

    if (!north[x][y]) { // if there is no wall above
        int d = solveDFS(x, y + 1, depth + 1);
        if (d > 0) return d;
    }
    if (!east[x][y]) {
        int d = solveDFS(x + 1, y, depth + 1);
        if (d > 0) return d;
    }
    if (!south[x][y]) {
        int d = solveDFS(x, y - 1, depth + 1);
        if (d > 0) return d;
    }
    if (!west[x][y]) {
        int d = solveDFS(x - 1, y, depth + 1);
        if (d > 0) return d;
    }
}
```

Note that this algorithm recurses on each of the N nodes 4 times (for each direction, and each recursive call is $O(1)$), so the complexity is $O(N)$.

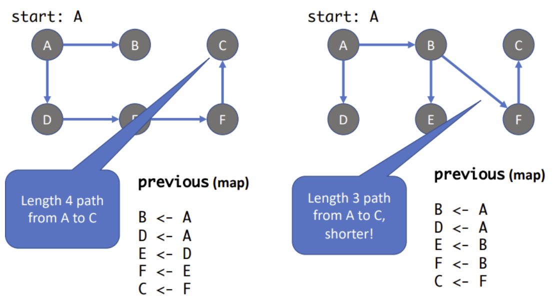
4.2 Breadth First Search (BFS)

Note that the main idea of DFS is to always explore a new adjacent vertex if possible, and if not, then backtrack to the most recent vertex adjacent to an unvisited vertex and continue. On the contrary, the main idea of BFS is to explore *all* your neighbors before you visit any of your neighbors' neighbors. It exhaustively searches for the closest regions of your search space before you look any further. Unlike DFS, which finds the some arbitrary path to a node, BFS finds the shortest (perhaps nonunique) path to a node.

Definition 4.3 (Iterative BFS)**.** This can be done simply by using a queue. Note that a queue is an interface, so we must use the Linked List implementation.

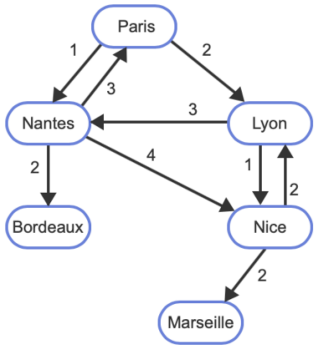
```
    public static void dfs(char start) {
        Queue<Character> toExplore = new LinkedList<>();
        char current = start;
        toExplore.add(current);
        visited.add(current);

        while (!toExplore.isEmpty()) {
            current = toExplore.pop();                  // explore from most recently discovered node
            for (char neighbor : aList.get(current)) {  // look at all neighbors of current node
                if (!visited.contains(neighbor)) {      // if we haven't seen them before...
                    previous.put(neighbor, current);    // note how we got here
                    visited.add(neighbor);              // note that we have seen
                    toExplore.push(neighbor);           // mark to explore later
                }
            }
        }
    }
```

Definition 4.4 (Search Trees)**.** Once we have traversed a graph using BFS or DFS, we can label the directed path that this traversal algorithm takes into a **search tree**. If we look at the search trees generated by DFS and BFS, we can see that the path from A to C is always shorter for BFS than for DFS.



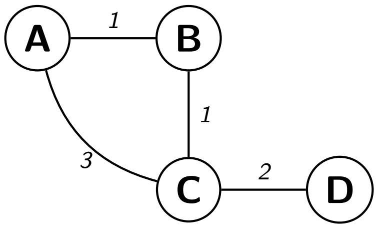
4.3 Shortest Paths and Dijkstra's Algorithm

Definition 4.5 (Weighted Graph)**.** A **weighted graph** just has some weights to each of the edges, which can represent multiple things, like distance, cost, or probabilities.



Remember that BFS gives the shortest paths in *unweighted graphs*. We can generalize BFS to account for weighted graphs, called Dijkstra's algorithm, which doesn't implement queues, but *priority queues*.

Theorem 4.3 (Dijkstra's Algorithm)**.** *The algorithm basically starts at a node, and explores first the closest nodes to the start node, called* `start`*. It does this by recording the shortest distance that it took to travel to this node from the start node. This is implemented in a* `Map` $<$ *Character, Integer* $>$*, which we will call* `dist` *for example. Furthermore, we want a* `PriorityQueue` *called* `exp` *that tells us which nodes that we need to explore next. Let us have a minimal example:*



1. *We start at a node A, which has distance* 0 *(since the distance from A to A is* 0*). So, A is* `current`*, and we have*

```
dist        exp
A -> 0      A
```

2. *We explore from A since that is the first in our Priority Queue. We want to look at all neighbors of the node (B, C) and compute their distances from* `start` *by adding the distances from* `start` *to* `current` *and* `current` *to the neighbor. So, we have*

```
dist        exp
A -> 0      B
B -> 1      C
C -> 3
```

3. *We explore from B, since that is first in our Priority Queue. We look at all neighbors of the node (A, C). The new distance to A is $d(A, B) + d(B, A) = 2$, so we do not update it. The new distance to C is $d(A, B) + d(B, C) = 2$ and is shorter than the current distance of 3, so we update it.*

```
dist        exp
A -> 0      C
B -> 1
C -> 2
```

4. *We explore from C, since that is first in our priority queue. We look at all neighbors of the node (A, B, D). The new distance to A is $d(A, C) + d(C, A) = 2 + 3 = 5$, so no update. The new distance to B is $d(A, C) + d(C, B) = 2 + 1 = 3$, so no update. D has not been explored before, so we we add its distance of $d(A, C) + d(C, D) = 2 + 2 = 4$ to the map and priority queue regardless.*

```
dist        exp
A -> 0      D
B -> 1
C -> 2
D -> 4
```

5. *We explore from D, since that is first in our priority queue. We look at all neighbors of the node (C). The new distance to C is $d(A, D) + d(D, C) = 4 + 2 = 6$, so no update. The priority queue is empty and we are done, with the shortest path to D being* `dist.get(D) = 4`.

```
dist        exp
A -> 0
B -> 1
C -> 2
D -> 4
```

The complete code would look like this, where `weight` correctly returns the weight of an edge.

```
public int Dijkstra(String start, String end, Map<String, List<String>> neighbors) {
    Map<String, Integer> dist = new HashMap<>();
    Comparator<String> c = (a, b) -> dist.get(a) - dist.get(b);
    PriorityQueue<String> exp = new PriorityQueue<>(c);

    dist.put(start, 0);
    exp.add(start);

    while (!exp.size() > 0) {
        String node = exp.remove();
        for (String adj : neighbors.get(node)) {
            int newDist = dist.get(node) + weight(node, adj);
            if (!dist.containsKey(adj) || newDist < dist.get(adj)) {
                dist.put(adj, newDist);
                exp.add(adj);
            }
        }
    }

    return dist.get(end);
}
```