CSDS 233 - Final Study Guide

Ethan Ho - erh101@case.edu

December 8, 2023

Contents

1 Priority Queues **4**
- 1.1 Terminology & Concept . . . 4
- 1.2 Relation to Heaps . . . 4

2 Heaps **4**
- 2.1 Terminology & Concept . . . 4
  - 2.1.1 Max-at-Top vs. Min-at-Top . . . 5
- 2.2 Methods . . . 6
  - 2.2.1 Removing the Maximum Element . . . 6
  - 2.2.2 Inserting . . . 6
  - 2.2.3 Removing Arbitrary Element . . . 7
  - 2.2.4 Updating Arbitrary Element . . . 8
- 2.3 Building . . . 8
  - 2.3.1 Slow Method . . . 8
  - 2.3.2 Faster Method . . . 8
- 2.4 Implementation . . . 9
- 2.5 Heap sort . . . 10

3 Hash Tables **10**
- 3.1 Terminology & Concept . . . 10
- 3.2 Chaining . . . 12
- 3.3 Open Addressing . . . 12
  - 3.3.1 Linear Probing . . . 12
  - 3.3.2 Quadratic Probing . . . 14
  - 3.3.3 Double Hashing . . . 14
- 3.4 Removed Flag . . . 15
- 3.5 Infinite Probing . . . 17
- 3.6 Rehashing . . . 17
- 3.7 String Hashing . . . 18
  - 3.7.1 Horner's Method . . . 18

4 Sorting Algorithms **18**
- 4.1 Selection Sort . . . 19
- 4.2 Insertion Sort . . . 20
- 4.3 Bubble Sort . . . 22
- 4.4 Shell Sort . . . 24
- 4.5 Quick Sort . . . 24
- 4.6 Merge Sort . . . 28
- 4.7 Bucket Sort . . . 31

5 Graphs **32**
- 5.1 Terminology & Concept . . . 32
  - 5.1.1 Weighted vs. Unweighted . . . 33
  - 5.1.2 Directed vs. Undirected . . . 34

5.1.3 Connected vs. Disconnected 34
5.1.4 Trees vs. Graphs 35
5.2 Spanning Trees 35
5.3 Adjacency Matrices 36
5.4 Implementation 36
5.5 Traversals 38
5.5.1 Breadth-First Traversal 38
5.5.2 Depth-First Traversal 39
5.6 Algorithms 40
5.6.1 Dijkstra's Algorithm 40
5.6.2 Prim's Algorithm 43

1 Priority Queues

1.1 Terminology & Concept

Recall queues from earlier in the semester. They use a FIFO (first-in-first-out) approach, with methods such as *insert*, *remove*, and *peek*.

A priority queue is similar to a normal queue, except the order that elements are dequeued in a priority queue is dependent on a priority. The higher the priority, the earlier the element is removed.

1.2 Relation to Heaps

Priority queues are closely related to heaps. We can use heaps to implement the priority queue paradigm. We will see in following sections how this is achieved.

2 Heaps

2.1 Terminology & Concept

A heap represents a special type of tree. More specifically, a heap is a complete, binary tree, where every level is filled from left to right and each node has a maximum of 2 children.

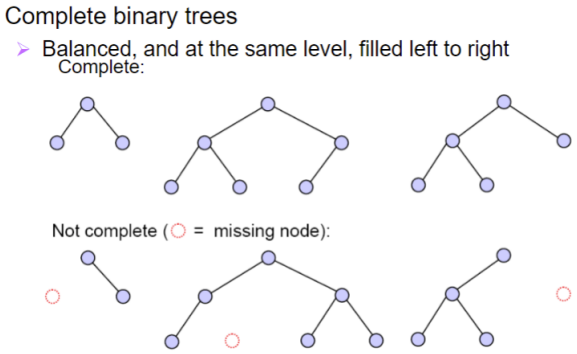


Figure 1: Differences between complete and incomplete binary trees.

We can represent complete binary trees in an array by simply writing down the nodes in level-order traversal.

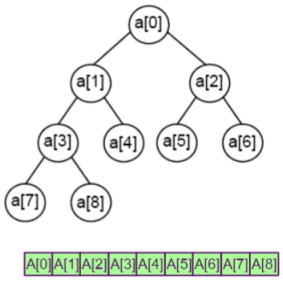


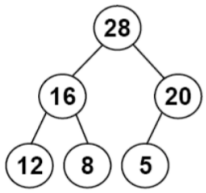
Figure 2: Using level-order traversal to put a tree into an array.

Assume that we have an array called a. Then, we can figure out nodes relative to each other...

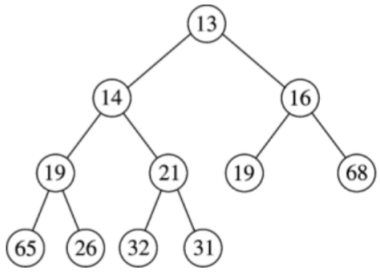
- The left child of node $a[i]$ is $a[2 * i + 1]$.
- The right child of node $a[i]$ is $a[2 * i + 2]$.
- The parent of node $a[i]$ is $a[(i - 1)/2]$, when $i \neq 0$.

2.1.1 Max-at-Top vs. Min-at-Top

In a max-at-top heap, a parent node is always greater than its children.



In a min-at-top heap, a parent node is always less than all of its children.



Note that we will mostly be working with max-at-top heaps for this chapter.

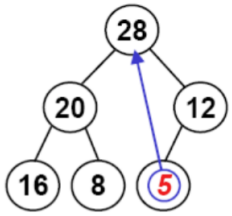
2.2 Methods

2.2.1 Removing the Maximum Element

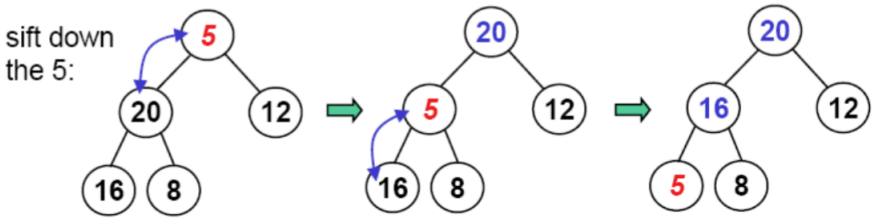
Removing the maximum element from a (max-on-top) heap is a multi-step process:

1. Replace the root of the heap with the *last* value of the heap.
2. This replacing value is most likely out of place, meaning it has children that are greater than it. This is where we call a method called "siftDown" on it, where we do the following:
   (a) Check to see if the current value is less than or equal to one of its children.
   (b) If so, choose the larger of the two children and swap it with the current value. We have now "sifted down" the current value by 1 level.
   (c) Continue this process until the current value is greater than its children (or it reaches the bottom of the heap).

Let's say we have the following heap, and we want to remove the maximum element:



We simply just replace it with the last value (5) and sift down 5 as necessary.



2.2.2 Inserting

Inserting an element has the opposite process:

1. Insert the new element into the last spot of the heap.

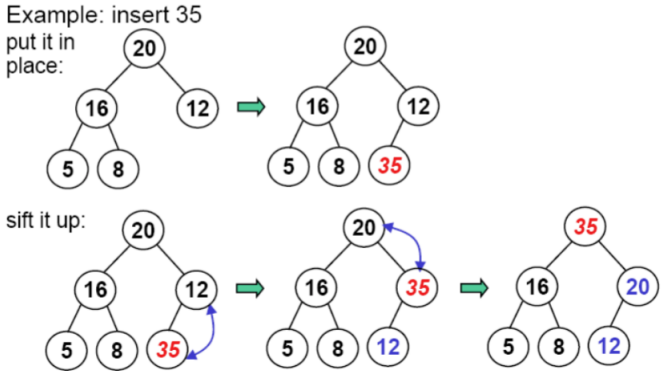
2. This new, inserted value now needs to be appropriately sifted up, as it may be greater than its parents. We call a method called "siftUp", where we do the following:

   (a) Check to see if the current value is greater than its parent.

   (b) If so, swap the parent with the current value. We have now "sifted up" the current value by 1 level.

   (c) Continue this process until the current value is less than its parent (or it reaches the top of the heap).

Here is an example of a heap insertion. Notice how 35 is submitted to the *end* of the heap and is sifted up accordingly.



2.2.3 Removing Arbitrary Element

Removing an arbitrary element in a heap (not necessarily the max) is more complicated than removing from the max or inserting into the heap. This is because if we delete an arbitrary element and replace it with the last element in the heap (like we've been doing), we initially don't know whether to $siftDown$ or $siftUp$. To combat this, we simply compare the replacing item with the item being replaced. If the new item is greater than the old item, we call siftUp. If the new item is less than or equal tothe old item, we call siftDown.

1. Compare the to-be-replaced item with the replacing item (last item in the heap).

   (a) If the new item is greater than the old item, replace the old item and call "siftUp" on the new item.

   (b) If the new item is less than or equal to the old item, replace the old item and call "siftDown" on the new item.

2.2.4 Updating Arbitrary Element

Updating an arbitrary element is almost exactly the same as removing the element. When you update an element, compare its previous value to its current value. If its new value is greater than its old value, call $siftUp$. If its new value is less than or equal to its old value, call $siftDown$. Follow the general rules for removing an arbitrary element and apply them to updating.

2.3 Building

2.3.1 Slow Method

The intuitive and naïve way to build a heap from random data would be to simply call the *insert* method for every single element in the set of data. This building algorithm looks like this:

```
public void buildHeap(T[ ] array, int size )
{
   < initialize the heap here ...>
   for( int i = 0; i < size; i++ )
      insert(array[i]);
}
```

However, visualizing a heap, note that it is pointless to call $siftDown$ on the bottom level nodes, since they are not able to go down any further. The "faster method" below describes an optimized version of the slower algorithm.

2.3.2 Faster Method

Knowing that calling $siftDown$ on the bottom level nodes is pointless, we can start calling $siftDown$ on the last element that is a parent.

1. Start position = (numItems - 2)/2. This is the last parent of the heap. We want to start calling $siftDown$ from here.
2. $shiftDown$ from this position.
3. Decrement the position by 1.
4. Repeat steps 2 and 3 until the position is 0.

```
public void buildHeap( )
{
   for( int i = (numItems - 2)/2; i >= 0; i--) //more efficient than
        last algorithm because unncessary siftDown calls aren't made
      siftDown( i );
}
```

2.4 Implementation

```
public class Heap<T extends Comparable<T>> {
   private T[] items;
   private int maxItems;
   private int numItems;
   public Heap(int maxSize) {
      items = (T[])new Comparable[maxSize];
      maxItems = maxSize;
      numItems = 0;
   }

   public void insert(T item) {
      if (numItems == maxItems) {
         // code to grow the array goes here...
      }
      Items[numItems] = item;
      numItems++;
      siftUp(numItems-1);
   }

   public T removeMax() {
      T toRemove = items[0];
      items[0] = items[numItems-1];
      numItems--;
      siftDown(0);
      return toRemove;
   }

   public T update(int i, T item) {
      T oldItem = items[i];
      items[i] = item;
      if (item.compareTo(oldItem) == 1) //if new item is greater than
          old item, sift up
         siftUp(i);
      else //if not, sift down
         siftDown(i);
      return oldItem;
   }

   public T delete(int i) { //very similar to update
      T toDelete = items[i];
      items[i] = item[numItems - 1];
      numItems--;
      if (toDelete.compareTo(items[i]) == -1)
         siftUp(i);
      else
         siftDown(i);
      return toDelete;
```

```
  }

  private void siftDown(int i) { // Input: the node to sift
     T toSift = items[i];
     int parent = i;
     int child = 2 * parent + 1; // Child to compare with; start with
         left child
     while (child < numItems) {
        // If the right child is bigger than the left one, use the
            right child instead.
        if (child +1 < numItems && // if the right child exists
        items[child].compareTo(items[child + 1]) < 0) // ... and is
            bigger than the left child
           child = child + 1; // take the right child
        if (toSift.compareTo(items[child]) >= 0)
           break;
        // Sift down one level in the tree.
        items[parent] = items[child];
        items[child] = toSift;
        parent = child;
        child = 2 * parent + 1;
     }
     items[parent] = toSift;
  }

  //note that there is no implementation for siftUp, but it would be
      very similar to siftDown, just going in the opposite direction
}
```

2.5 Heap sort

Heap sort is a two step process:

1. Turn the array of data into a max-at-top heap
2. Remove the current maximum value and place it after the end of the heap. Let the heap reconfigure itself. Repeat this until the heap is of size 1. You now have a sorted array.

Heap sort is a bit complicated to explain and visualize, so please visit lecture 13 to see a full visualization of it.

3 Hash Tables

3.1 Terminology & Concept

So far in this course, data structures have had linear or logarithmic time operations. What if we could do better? What if there was a data structure that

allowed for constant time operations? Introducing the hash table!

Hash tables take advantage of the array's constant time indexing. When you have an integer array *arr* (let's say of size 10), you can get the 4th element in the array by simply calling $arr[3]$. This is a constant time operation. However, the issue is that the 4th index has no relation to the data that you're storing in that slot. If you wanted to search for a specific integer in your array, you'd have to loop through the entire array to find it! This search becomes a linear time operation, which is less ideal than a constant time operation.

The hash table solution takes data and *turns it* into an index. Now there's a relationship between the index in the array and the data, allowing for a constant time search, insert, and removal. Let's take a simple example.

Let's say you're a TA in charge of grading a class of 40 students. We can represent each student as an object with a *grade* field, which is an integer from 0 to 100, representing their percentage grade. For the sake of simplicity, let's assume that all grades are integers and no two students have the same grade. How could you use hash tables to effectively store students? You could use their *grade* as the index! Create an array of size 101 and when you want to insert a student into array, the index at which you put the student is equal to their grade in the class. For example, if student Leonardo has a 98 in the class, his student object would be put in slot 98 of the hash table. We can then retrieve Leonardo's student object almost instantly by just finding his grade and indexing the hash table with it. This process of mapping a piece of data to an index for a hash table is called *hashing*, and these types of processes run in methods called *hashing functions*.

As mentioned, a hash function is a function that takes in some sort of data and turns it into an index to be used for the hash table. Some examples include using the number of letters of a word or using a student's grade in the class to index the data.

Hash functions (in the context of this class) frequently use remainders (%). Let's modify our grade example to see this. Let's say, due to standards enforced by our organization, we are only allowed to make the hash table of size 50. How can we now hash our students' grades to fit in a hash table of size 50? We can't simply use their raw grades this time because some students may have grades that are greater than 50. The solution? Use the remainder function! Our hash function can simply be $grade\%50$. This will guarantee that all indices will fall within the range of $[0, 50)$, just as we want. Again, for the sake of simplicity, ignore any potential "collisions" (if you don't understand what this means, don't worry we'll get to it). To fit large integer values into smaller hash table sizes, we can simply use $value\%size$. This is not always the ideal solution. Sometimes there are better hashing functions. However, this is a commonly used hashing function that you should be familiar for this course.

3.2 Chaining

What happens when two pieces of data share the same index after the hash function? For example, if we are hashing words by their number of letters, "Apple" and "Mouse" would be indexed to the same slot in our hash table.

We have multiple solutions to solve this issue, and this section will explain the first solution: chaining!

The idea is this: In each slot of the hash table, we store a linked list. Whenever an element is added to the hash table, it gets added to the *linked list* at its hashed index.

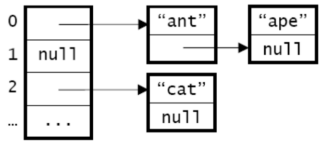


Figure 3: Chaining - Each index has a linked list. Index 0 has 2 elements in its linked list, "ant" and "ape". Index 2 has 1 element in its linked list, "cat".

There exists an alternative implementation of chaining where instead of using linked lists, we use arrays.

The major disadvantage with the chaining approach is that creating linked lists for each slot creates large memory overhead. Along with this, if we have many elements mapped to the same index, the associated linked list can become very long, making the overall data structure less efficient.

3.3 Open Addressing

Open addressing is a different approach to solving collisions. Instead of using linked lists, open addressing searches for an open slot to place the element in. We have three different types of open addressing: linear probing, quadratic probing, and double hashing.

3.3.1 Linear Probing

In linear probing, if we map data to a specific index, and that index happens to be full, we simply increment the index by 1 until we find the first open slot. For example, let's say we had the following hash table:

0	
1	apple
2	
3	cat
4	
5	

Now we want to insert 'kite'. We run 'kite' through our hashing function and we get an index of 3. However, we already have a value in slot 3: 'cat'. So, we increment the index up until we find the next open address: 4.

0	
1	apple
2	
3	cat
4	kite
5	

Let's say we want to insert 'whale'. We run 'whale' through our hashing function and get an index of 1. However, we already have a value in slot 1: 'apple'. So, we increment the index up until we find the next open address: 2.

0	
1	apple
2	whale
3	cat
4	kite
5	

Finally, let's say we want to insert 'chair', which maps to an index of 1. Since slot 1 is full, we increment the index up until we find the next open address: 5.

0	
1	apple
2	whale
3	cat
4	kite
5	chair

Note that, for when the probe reaches the end of the hash table, simply reset the probe index back to the beginning of the table.

3.3.2 Quadratic Probing

Quadratic probing is similar to linear probing, except instead of incrementing by 1 from the previous index, we increment by the quadratic sequence (1, 4, 9, ...) from the *originally determined* index (from the hashing function).
For example, let's say we start with the hash table

0	
1	apple
2	
3	cat
4	
5	

We want to insert 'kite', which has a hashed index of 3. However, we already have a value in slot 3: 'cat'. So, we use quadratic probing to find an empty slot! We add 1 to the index (4), which happens to be an open slot!

0	
1	apple
2	
3	cat
4	kite
5	

Now we want to insert 'whale', which we also received to be a hash index of 3. First we try adding 1 to the index (4), which is already full. Then we try adding 4 (the next value in the quadratic sequence) to 3 (notice we add it to 3, the originally hashed index, not 4). Since this exceeds the array size, we go back to the beginning and see that we end up at 1, which is also filled. Now we add 9 to 3, which lands us at index 0, which is not filled.

0	whale
1	apple
2	
3	cat
4	kite
5	

3.3.3 Double Hashing

Double hashing is our third option for open addressing. In double hashing, we use *two* hash functions we call $h1$ and $h2$. The process for double hashing is as follows:

1. Apply hashing function $h1$ to the data for an initial index.
   (a) If the index is empty, put the data at that slot. You're done.
   (b) If the index is *not* empty, then go to step 2.
2. Let $h1$ be the current index.
3. Add the index obtained from hashing the data in $h2$ onto the current index. Let this be the new current index.
   (a) If the current index is empty, put the data at that slot. You're done.
   (b) If the index is *not* empty, then go back to step 3.

To summarize this process, $h1$ picks an initial slot. If it's full, then we add the value obtained from $h2$ over and over to our current index until we find an open address.

For example, let's say we have the following hash table.

0	
1	apple
2	
3	cat
4	kite
5	

We want to add 'mouse'. We apply our first hashing function to it $h1$ and it turns out to be 4. This index is occupied by 'kite'. We then look at our second hashing function. Let's actually give the hashing function a function this time, and say that $h2$ returns the number of characters in the string. Since 'mouse' has 5 characters, we will increment by 5 until we find an open address. After the first addition 4 + 5 (with respect to overflow), we see that we land at index 3, which is also occupied. We then add the value from $h2$ once again, 3 + 5 (with respect to overflow), we see that we now land at empty index 2. We can then place 'mouse' at index 2.

0	
1	apple
2	mouse
3	cat
4	kite
5	

3.4 Removed Flag

There is a massive issue with the open addressing solution to collisions. Examine the following scenario:

Let's say we have this hash table:

0	
1	apple
2	
3	
4	
5	

Now we want to add 'orange' to our hash table. We run the data through the function, and the hashing function indicates that it should be at index 1. However, since 'apple' is already at index 1, we need a way to handle collisions. Let's use linear probing for the sake of simplicity. After probing, our table should look like this:

0	
1	apple
2	orange
3	
4	
5	

Great! Now let's say we want to remove apple from the table. All we need to do is hash 'apple', which is 2, and remove it at index 2.

0	
1	
2	orange
3	
4	
5	

Oh no! There's a problem! What if we wanted to search the table to see if we had 'orange'? We call the hashing function, which gives us 2 for orange, and we check that slot, but there's nothing there! 'apple' used to be there, but it's no longer there, and we don't know where orange is. From this implementation, we'd think orange doesn't exist in our table at all! Let's fix that.

We can solve this issue by introducing a flag. If we are searching for an item by hash in the table, and the slot has the 'removed' flag, we know that there *used* to be some value there, meaning we should continue searching for our intended value. We can set this up by initializing an object for all slots of the hash table. This object keeps track of the 'status' of the slot, indicating whether an object has been removed or not.

```
public class HashTable {
  private class Entry {
```

```
    private String key;
    private String etymology;
    private boolean removed;
  }
  private Entry[] table;
  private int tableSize;
```

3.5 Infinite Probing

We've solved one issue with open addressing, but there are still more. What happens if the probe runs infinitely? In other words, what do we do if the probe always hits a value, no matter how long we try to probe, even if there may still be empty addresses? We need a precaution to prevent probes from running to long.

The precaution we take is keeping track of the number of iterations, and stopping it if the number of iterations exceeds a limit. Mathematically, it makes sense to cut off probing after the number of iterations has reached the size of the hash table, because at that point it will just start repeating values. In essence, when the number of iterations reaches the size of the table, we simply indicate that the operation cannot be performed, whether it be inserting or removing.

```
private int probe(String key) {
  int i = h1(key); // first hash function
  int j = h2(key); // second hash function
  int iterations = 0;
  // keep probing until we get an empty or removed position
  while (table[i] != null && table[i].removed==false) {
    i = (i + j) \% tableSize;
    iterations++;
    if (iterations >= tableSize) return -1; //return -1 (error value)
        if number of iterations exceeds the size of the table
  }
  return i;
}
```

3.6 Rehashing

Over time, we will see that a hash table's efficiency will decrease. This is due to two major reasons.

1. In the context of open addressing, after some time of inserting and removing, the hash table will end up having many 'removed' flags. Recall that search probes that encounter 'removed' flags must continue searching. The amount of 'removed' flags can significantly increase the time complexity

of the search method, defeating the purpose of implementing a hash table (the supposed "fast operation data structure") in the first place.

2. As *load factor* λ (the ratio of number of keys to table size) increases, the difficulty of finding open spots to insert into scales with it. A hash table with a large load factor (meaning many of the slots in the hash table are already full) can significantly increase the time complexity of the insert method, again, defeating the purpose of implementing a hash table in the first place.

In both cases, we will use rehashing to solve our issues.

For the first case, the presumption is that there are many 'removed' flags, but the load factor itself is not too high. The solution for this is to simply create another hash table (of the same size), and move every entry to the new table.

For the second case, we need to somehow decrease the load factor without removing elements. The solution for this is to create another hash table, roughly double the size of the initial one, and move all entries to this table.

3.7 String Hashing

What hash functions can we use for strings? A good hash function for strings needs to meet the following requirements:

1. Full table size utilization
2. Uniform key mapping throughout table
3. Efficient to compute

3.7.1 Horner's Method

Please just see lecture 16 for a mathematical breakdown of this.

```
int hash = s.charAt(0);
for (int i = 1; i < s.length(); i++)
  hash = hash * b + s.charAt(i);
```

4 Sorting Algorithms

This section will outline the various sorting algorithms covered in this course. For each algorithm, I will provide the*

- Concept of algorithm

- Example
- Implementation (Java code)
- Time complexity analysis (Big O notation)

*Two notes: (1) I have not completed everything for shell sort yet. (2) I did not include an example or implementation for bucket sort as Prof. Erman briefly touched on it in class. His slides do not even include any code.

4.1 Selection Sort

Concept
You can think of selection sort as "for each element in the array, find the nth smallest and move it to position n". Here is the general process of selection sort given a supposedly unsorted array integers of size n. Let's call the array arr.

1. Let variable $i = 0$.
2. Loop through arr using i, meaning for each iteration, i will increment by 1. In each iteration, the following will happen.
   (a) From i to the end of arr, loop through to find the smallest element. Let's call the position of this smallest element k.
   (b) Swap the elements of i and k.
   (c) Now for elements 0 through i, the array is sorted.
3. Once i reaches the end of the loop, the array should be sorted.

Example
$arr = [2, 5, 7, 3, 4, 1, 9]$ Goal: Sort arr in ascending order using selection sort.

Let's use the steps from the subsection above to complete the goal.

First, we let $i = 0$. Then, from position i to the length of arr, we search for the *smallest* element. In this case, the smallest element from 0 to the length of arr is 1, which is stored in the 5th index of arr (this is our k from the step-by-step procedure). We then swap the elements in i and k, so we swap the elements in 0th and 5th indices (elements 2 and 1). Now we have the following array.

$[1, 5, 7, 3, 4, 2, 9]$

Now, we repeat the process! This time, $i = 1$. We search for the smallest element from 1 to the length of the array (2 in index 5), and swap the elements:

$[1, 2, 7, 3, 4, 5, 9]$

Notice that all we're doing is finding the smallest element from i to the length of arr and swapping them. Continuing the pattern:

$\rightarrow [1, 2, 3, 7, 4, 5, 9]$
$\rightarrow [1, 2, 3, 4, 7, 5, 9]$
$\rightarrow [1, 2, 3, 4, 5, 7, 9]$
$\rightarrow [1, 2, 3, 4, 5, 7, 9]$

Java Code

```
static void selectionSort(int[] arr) {
   for (int i = 0; i < arr.length - 1; i++) {
      int j = findMin(arr, i); //find the minimum value with i as the
          lower bound and set as j
      swap(arr, i, j); //call helper method that swaps the elements in
          positions i and j
   }
}

static int findMin(int[] arr, int lower) {
   int curIndexMin = lower; //placeholder variable for the minimum value
   for (int i = lower + 1; i <= arr.length - 1; i++) {
      if (arr[i] < arr[curIndexMin]) {
         curIndexMin = i;
      }
   }
   return curIndexMin;
}
```

Time Complexity Analysis
Selection sort is considered an $O(n^2)$ algorithm. This is because for each element in the array (n), we loop again with respect to the size of n. This means that the complexity of the method scales with $n * n = n^2$. Thus, selection sort's time complexity expressed in O-notation is $O(n^2)$.

4.2 Insertion Sort

Concept
For insertion sort, you start with a sorted sub-array of size 1 and increase it until it is size n, thus sorting the entire array.

1. Let variable $i = 1$. The assumption is that any value in the subarray from $[0, i)$ is sorted. We see that when $i = 1$, the only element in range $[0, i]$ is the first element of the array. Since this is a subarray of size 1, the subarray is inherently sorted.

2. Loop through arr using i, meaning for each iteration, i will increment by

1. In each iteration, the following will happen.

   (a) Examine the element at place i, call this element k. Since we know the values from $[0, i)$ are sorted, all we need to do is push the value k back until the element before k is $\leq k$ and the element after k is $\geq k$. We use a while loop to do this, swapping k with its previous element until it is $\geq$ than the element before it.

   (b) The above process sorts the subarray $[0, i+1)$, meaning we can then safely increment i, making the new subarray $[0, i)$ sorted.

3. Once i reaches the end of the loop, the entire array should be sorted.

Example
$arr = [2, 1, 7, 5, 3, 4, 9]$ Goal: Sort arr in ascending order using insertion sort.

Let's use the steps from the subsection above to complete the goal.

First, we let $i = 1$. Notice how the subarray $[0, i)$ is already sorted, as the only value in $[0, i)$ is 2. This array of size 1 is already sorted. Now, look at the element at i, which happens to also be 1. We then *push back* this value k until the element before it is less than equal to k (we push back by swapping). Now we have the following array.

$[1, 2, 7, 5, 3, 4, 9]$

Notice how now the range $[0, i + 1)$ is sorted! This means that we can safely increment i. Now $i = 2$ and the subarray from $[0, i)$ is sorted. We then repeat this process.

Because $i = 2$, we push back the value k until the element before it is less than or equal to k. In this case, the element is 7, and the element right before it (2) is already less than 7.

$[1, 2, 7, 5, 3, 4, 9]$

This means that we can safely increment i and move onto the next value, since the subarray from $[0, i + 1)$ is already sorted. Continuing the pattern:

$\rightarrow [1, 2, 5, 7, 3, 4, 9]$
$\rightarrow [1, 2, 3, 5, 7, 4, 9]$
$\rightarrow [1, 2, 3, 4, 5, 7, 9]$
$\rightarrow [1, 2, 3, 4, 5, 7, 9]$

Java Code

```
static void insertionSort(int[] arr) {
  for (int i = 1; i < arr.length; i++) {
```

```
      int toInsert = arr[i];
      for (j = i; j > 0 && toInsert < arr[j - 1]; j--) { //push back
          (swap) the 'toInsert' value until the value before it is less
          than or equal to it
         arr[j] = arr[j - 1];
      }
      arr[j] = toInsert;
   }
}
```

Time Complexity Analysis
Insertion sort is considered an $O(n^2)$ algorithm, as its nested for-loops make insertion sort run close to n^2 with random data. However, insertion sort is much faster than selection sort for nearly sorted data. Why? Remember how we had to "push back" the value k (or toInsert in the pseudocode) until the value before it was less than or equal to the previous value? If our array is already nearly sorted, we would usually not need to push k back that as much as we would have to with random data. This leads to less complexity and a higher runtime.

4.3 Bubble Sort

Concept
Bubble sort can be pictured using bubbles (hence the name). Essentially, the largest value of a subarray "floats" (like a bubble) to the end of a subarray. This subarray starts as the entire array (meaning the largest value of the array will float to the end of the array), and decreases in size by 1. This means that on the second iteration, the *second* largest element of the array will "float" to the *second-to-last* spot of the array. This process continues until all elements have been put in their proper position.

1. Let variable $i =$ the length of the array -1.
2. Iterate i to 0 by decrementing by 1 for each iteration. Within each iteration...
   (a) Let variable $j = 0$.
   (b) Loop j to i, pushing up the largest value all the way to i.
3. Once $i = 0$, the entire array should be sorted.

Example
$arr = [2, 1, 7, 9, 5, 3, 4]$ Goal: Sort arr in ascending order using bubble sort.

Let's use the steps from the subsection above to complete the goal.

First, we let $i = 6$, because arr.length $-1 = 6$. Remember, in bubble sort, we push the largest value of the subarray to the top. In this case, our subarray is $[0, 7)$. We start our nested loop where $j = 0$, and we swap the value in j with the value in $j + 1$ if the value in j is larger than it:

$arr = [2, 1, 7, 9, 5, 3, 4]$

$j = 0$ Since the value at j (2) is larger than the value at $j + 1$ (1), we swap the two (swap 2 and 1).
$arr = [1, 2, 7, 9, 5, 3, 4]$ Swap 2 and 1; increment j.

$j = 1$ Since the value at j (2) is not larger than the value at $j + 1$ (7), we don't do anything except increment j.
$arr = [1, 2, 7, 9, 5, 3, 4]$ Increment j.

$j = 2$ Since the value at j (7) is not larger than the value at $j + 1$ (9), we don't do anything except increment j.
$arr = [1, 2, 7, 9, 5, 3, 4]$ Increment j.

$j = 3$ Since the value at j (9) is larger than the value at $j + 1$ (5), we swap the two (swap 9 and 5).
$arr = [1, 2, 7, 5, 9, 3, 4]$ Swap 9 and 5; increment j.

Continuing this process until $j = i$... $[1, 2, 7, 5, 3, 9, 4]$
$[1, 2, 7, 5, 3, 4, \mathbf{9}]$

Now notice how the largest value (9) "floated" to the end of the array. We now need to repeat this process for the rest of the subarrays.

$[1, 2, 5, 3, 4, \mathbf{7}, 9]$
$[1, 2, 3, 4, \mathbf{5}, 7, 9]$
$[1, 2, 3, \mathbf{4}, 5, 7, 9]$
$[1, 2, \mathbf{3}, 4, 5, 7, 9]$
$[1, \mathbf{2}, 3, 4, 5, 7, 9]$
$[\mathbf{1}, 2, 3, 4, 5, 7, 9]$

Java Code

```
static void bubbleSort(int[] arr) {
  for (int i = arr.length - 1; i > 0; i--) {
    for (int j = 0; j < i; j++) { //bubbling up
      if (arr[j] < arr[j + 1]) {
        swap(arr, j, j+1) //call a constant time method to swap the
            two elements
      }
    }
```

```
    }
}
```

Time Complexity Analysis
Bubble sort is considered an $O(n^2)$ algorithm, as the nested for-loop structure cause the algorithm's time to scale by a factor of about n^2.

4.4 Shell Sort

not rn i'll do it later

4.5 Quick Sort

Concept
Quick sort uses the strategy of *divide and conquer* to sort arrays more efficiently than some of the previously mentioned sorting algorithms. Given an array, a pivot is chosen (see *Choosing a Pivot* below). A pivot is simply an element in the array. We then arrange the array so that all of the values to the *left* of the pivot are less than the pivot, and all of the values to the *right* of the pivot are greater than the pivot. We now have two subarrays where every value in the left subarray is less than or equal to every value in the right subarray. For each of these two subarrays, we call this same process again. Quick sort is a recursive algorithm.

1. Pick a pivot in the given subarray (at the beginning, this subarray will be the whole array).
2. Call the "partition" method, which arranges all values *less than* the pivot to the left of the pivot and all values *greater than* the pivot to the right of the pivot.
3. Restart this process (go back to step 1) with each subarray.

Choosing a Pivot: There are many ways we can choose a pivot for QuickSort. Some of the common choices include:

- Pick the middle element: This is generally a good choice, especially if the array is sorted (or nearly sorted). As quick sort works most efficiently when a partition *evenly* divides the subarray, picking the middle element for a nearly sorted array is always a good idea.
- Pick the first or last element: This is generally not a good choice. It's especially bad if the array is nearly sorted, as it heavily nullifies quick sort's divide and conquer advantage.
- Random pivot: Good for random data.

- Median of three elements: Good for maximizing the probability of a good pivot being chosen.

Understanding Partitioning: Partitioning is (arguably) the most tricky concept of quick sort, so I will try to break it down here.

We know that partitioning is the process of organizing all values less than or equal the pivot to the *left* of the pivot and all values greater than or equal the pivot to the *right* of the pivot, but how is this done internally? Let's tackle this with an example:

Say we have the following array:

$[2, 7, 5, 4, 1, 3, 6]$

For the sake of simplicity, let's say our choosing pivot method is to use the middle element as the pivot. In this case, 4 is the middle element. We want to organize the array such that all elements to the left of 4 are less than or equal 4 and all elements to the right of 4 are greater than or equal 4.

We start out by identifying two variables, one (i) before the front of the array, and another (j) after the end of the list. In other words, $i = -1$ and $j = 7$. Then, we keep incrementing i until we find a value that is *greater* than or equal the pivot. We know that this found value is out of place (because we want all values to the left of the pivot to be less than or equal the pivot value). Below, the bold indicates the current i.

$[\mathbf{2}, 7, 5, 4, 1, 3, 6]$ *Not greater than or equal to 4... let's move on*
$[2, \mathbf{7}, 5, 4, 1, 3, 6]$ *Aha!* $7 > 4$*! We wait here.*

We've found an i value that's greater than or equal to the pivot! Great. Now we do something similar for j, except in this case we keep *decrementing* j until we find a value that is *less* than or equal the pivot. We know that this found value is out of place (because we want all values to the right of the pivot to be greater than or equal the pivot value). Below, the bold indicates the current j.

$[2, 7, 5, 4, 1, 3, \mathbf{6}]$ *Not less than or equal to 4... let's move on*
$[2, 7, 5, 4, 1, \mathbf{3}, 6]$ *Yay!* $3 < 4$*! We wait here.*

Now that we've found i and j values that are *both* out of place, we simply swap them! Note that you *must* ensure that i has not passed j, meaning $i < j$. If they have passed each other, then do not swap, just terminate the loop because it's an indication that the subarray has already been partitioned. We will see this in action later.

$[2, \mathbf{3}, 5, 4, 1, \mathbf{7}, 6]$ *Swapped 7 and 3.*

Okay! Let's continue. Move i up until it reaches a value greater than or equal the pivot.

$[2, 3, \mathbf{5}, 4, 1, 7, 6]$ *Found one!* $5 > 4$.

Now move j down until it reaches a value less than or equal the pivot.

$[2, 3, 5, 4, \mathbf{1}, 7, 6]$ *Found one!* $1 < 4$.

First, we check to make sure i has not passed j. Since $i = 2$ and $j = 4$ and $2 < 4$, we are still good to swap. We swap:

$[2, 3, \mathbf{1}, 4, \mathbf{5}, 7, 6]$ *Swapped 5 and 1.*

Continue the process:

$[2, 3, 1, \mathbf{4}, 5, 7, 6]$ *value at i is now 4 and we stop here because* $4 \geq 4$

$[2, 3, 1, \mathbf{4}, 5, 7, 6]$ *value at j is also 4 and we stop here because* $4 \leq 4$

Do we swap? No! $i = 3$ and $j = 3$. $i \not< j$, therefore we simply return the index j to be used as the splitter of the subarray. Our partitioned array ends up being:

$[2, 3, 1, 4, 5, 7, 6]$

Notice how every value to the left of our pivot (4) is less than or equal to it, and every value to the right of our pivot is greater than or equal to it. We have now partitioned our subarray!

Example
$arr = [2, 5, 7, 3, 4, 1, 9]$ Goal: Sort arr in ascending order using quick sort.

Let's use the steps from the subsection above to complete the goal.

For the sake of simplicity, let's use the method of picking the middle element as our pivot. In this case, the pivot would be 3. Using the method explained in *Understanding Partitioning*, we can create the following array:

$[2, 1, 3, 7, 4, 5, 9]$

Now we call the quick sort algorithm *again* twice on each of the created subarrays. In otherwords, we call the quick sort method on the range $[0, 3)$ and $[3, 7)$. After this call, our array looks like

$[1, 2, 3, 4, 7, 5, 9]$ *Assumed that 1 and 4 were the pivots of their respective subarrays*

Now we have the subarrays $[0, 1)$, $[1, 3)$, $[3, 4)$, and $[4, 7)$. Again, we call the quick sort algorithm for each of these subarrays. Continuing the sort, our array looks like

$[1, 2, 3, 4, 5, 7, 9]$ *Need to call again for the created subarray* $[5, 7)$.
$[1, 2, 3, 4, 5, 7, 9]$ *Quick sort actually calls for subarrays of size 1, but I will not show that because it would just be a copy and paste of the same line many times.*

Java Code

```
static void quickSort(int[] arr) {
   myQuickSort(arr, 0, arr.length - 1); //call myQuickSort with first
       and last as the first and last values of the array
}

static void myQuickSort(int[] arr, int first, int last) {
   if (first >= last) return; //base case
   int split = partition(arr, first, last); //call partitioning to split
       the subarray; split is our "middle value"
   //call myQuickSort on the resulting two subarrays
   myQuickSort(arr, first, split);
   myQuickSort(arr, split + 1, last);
}

static int partition(int[] arr, int first, int last) {
   int pivot = arr[(first + last)/2]; //picking the middle element as
       the pivot
   int i = first - 1;
   int j = last + 1;
   while (true) {
      do { //increment i until element at i is greater than or equal to
          pivot
         i++;
      } while (arr[i] < pivot);
      do { //decrement j until element at j is less than or equal to
          pivot
         j--;
      } while (arr[j] > pivot);
      if (i < j) //ensure that i and j have not passed each other
         swap(arr, i, j); //call constant time method that swaps
             elements at i and j
      else
         return j;
   }
}
```

Time Complexity Analysis
Quick sort is considered an $O(n \log(n))$ algorithm. Theoretically, quick sort *could* run at $O(n^2)$, if the worst pivot is chosen for every iteration of quick sort. However, since this is astronomically unlikely for large values of n, a more accurate complexity for this algorithm is $O(n \log(n))$.

4.6 Merge Sort

Concept
Like quick sort, merge sort also takes advantage of the divide and conquer strategy to efficiently sort sets of data. There are two major parts in merge sort: (1) split the subarrays into halves recursively, until all subarrays are of size 1, and (2) merge these subarrays to create sorted subarrays that will be merged into larger sorted subarrays.

1. Recursively split the array into halves recursively, until all subarrays are of size 1.
2. Merge these subarrays into sorted subarrays. These sorted subarrays will then be merged to create even larger sorted subarrays.

Understanding Merging: The merging portion of merge sort is (arguably) the most complicated part of merge sort. I will demonstrate this using an example.

Say we have the following array:

$[2, 4, 5, 7, 1, 3, 8, 9]$

Note that when we merge an array, we *assume* that each of the two halves are sorted already. Also, note that this array might be a subarray of a larger array. For the sake of simplicity, I've ignored this concept. We can see that these two halves are sorted already. Elements $[0, 4)$ are sorted and elements $[4, 8)$ are sorted. We need to merge these two subarrays into one large sorted subarray.

To achieve this merge, we have a few variables that we were either given or we create.

- temp: This is an array of the same data type and of the same size as the given array. We will use this array to temporarily store values.
- i: This is a cursor variable at the *start* of the left subarray.
- j: This is a cursor variable at the *start* of the right subarray.
- k: This is another cursor variable that is equal to the start of the left array, however it will be used to index the temp array.

In this case, temp is an integer array of size 8. i is 0 and points to value 2 in our array. j is 4 and points to value 1 in our array. k is also 0.

We then begin a while-loop, which only runs when i and j haven't reached the ends of their respective subarrays. Obviously, since i and j are at the start of their respective subarrays, we do not need to worry about this right now. First, we check to see if the value at i (2) is less than or equal to the value at j (1). In this case, it is not. Therefore, we save the value at j into the temp array at spot k. In other words, we simply move the smaller value (between the values at i and j) to the temp array. We then increment k. Values at i and j are bolded.

$[\mathbf{2}, 4, 5, 7, \mathbf{1}, 3, 8, 9]$ *Because 1 is the smaller value, we move it to temp.*
temp = $[1]$ *We now increment j and k.*

Now we repeat the process. This time, since the value at i (2) is less than or equal to the value at j (3), we move 2 into temp at $k = 1$ and increment i and k.
$[\mathbf{2}, 4, 5, 7, 1, \mathbf{3}, 8, 9]$ *Because 2 is the smaller value, we move it to temp.*
temp = $[1, 2]$ *We now increment i and k.*

Continuing this process:

$[2, \mathbf{4}, 5, 7, 1, \mathbf{3}, 8, 9]$ *Because 3 is the samller value, we move it to temp.*
temp = $[1, 2, 3]$ *We now increment j and k.*

$[2, \mathbf{4}, 5, 7, 1, 3, \mathbf{8}, 9]$ *Because 4 is the samller value, we move it to temp.*
temp = $[1, 2, 3, 4]$ *We now increment i and k.*

$[2, 4, \mathbf{5}, 7, 1, 3, \mathbf{8}, 9]$ *Because 5 is the samller value, we move it to temp.*
temp = $[1, 2, 3, 4, 5]$ *We now increment i and k.*

$[2, 4, 5, \mathbf{7}, 1, 3, \mathbf{8}, 9]$ *Because 7 is the samller value, we move it to temp.*
temp = $[1, 2, 3, 4, 5, 7]$ *We now increment i and k.*

Notice that i has reached the end of its respective subarray. Because we've reached the end, we know that the rest of the right subarray $\geq$ at j can simply be added to the end of the temp array.

temp = $[1, 2, 3, 4, 5, 7, 8, 9]$ *Add the rest (8 and 9) from the right subarray, since we've reached the end of the left subarray.*

We now have a completely sorted temp array. The last thing we need to do is copy everything from temp into the actual subarray.

$[1, 2, 3, 4, 5, 7, 8, 9]$

This is how merging subarrays into larger sorted subarray works. To summarize, we make an auxiliary array called temp where we simply hand pick the lowest values in order from the left and right subarrays, move them to the temp array, and then copy everything from the temp array to the main array afterwards.

Example
$arr = [2, 5, 7, 3, 4, 1, 9, 0]$ Goal: Sort arr in ascending order using quick sort.

Let's use the steps from the subsection above to complete the goal.

First, we recursively split the array into halves until we have arrays of size 1. Note that in code, we don't actually create new arrays for the splitting, we just use pointers to indicate where splits happen. Visualizing them as subarrays helps with understanding how splitting and merging work.

$[2, 5, 7, 3, 4, 1, 9, 0]$
$[2, 5, 7, 3][4, 1, 9, 0]$
$[2, 5][7, 3][4, 1][9, 0]$
$[2][5][7][3][4][1][9][0]$ *Size 1 arrays!*

Now, using the merging technique explained in *Understanding Merging*, we create sorted subarrays from smaller subarrays.

$[2, 5][3, 7][1, 4][0, 9]$ *Notice how each subarray is sorted!*
$[2, 3, 5, 7][0, 1, 4, 9]$ *Same with these subarrays!*
$[0, 1, 2, 3, 4, 5, 7, 9]$

Java Code

```
static void mergeSort(int[] arr) {
   int[] temp = new int[arr.length]; //create temp array to be used and
       indexed
   myMergeSort(arr, temp, 0, arr.length - 1); //call merge sort with
       front and back of array as constraints
}

static void myMergeSort(int[] arr, int[] temp, int start, int end) {
   if (start >= end) return; //base case
   int middle = (start + end)/2; //split point

   //recursively split the subarray
   myMergeSort(arr, temp, start, middle); //call merge sort on left side
   myMergeSort(arr, temp, middle + 1, end); //call merge sort on right
       side

   merge(arr, temp, start, middle, middle+1, end); //merge the subarrays
}
```

```
static void merge(int[] arr, int[] temp, int leftStart, int leftEnd, int
    rightStart, int rightEnd) {
   int i = leftStart;
   int j = rightStart;
   int k = leftStart; //index for temp
   while (i <= leftEnd && j <= rightEnd) { //while i and j have not
       reached the end of their subarrays...
      if (arr[i] <= arr[j]) { //find the minimum value at i and j; which
          ever is lower, move to temp and increment the respective
          cursor along with k
         temp[k] = arr[i];
         i++;
      } else {
         temp[k] = arr[j];
         j++;
      }
   }
   //if j reached the end of its subarray first, add the rest of what's
       on the left subarray
   while (i <= leftEnd) {
      temp[k] = arr[i];
      i++;
      k++;
   }
   // if i reached the end of its subarray first, add the rest of what's
       on the right subarray
   while (j <= rightEnd) {
      temp[k] = arr[j];
      j++;
      k++;
   }
   //copy everything from the temp array to the original array
   for (i = leftStart; i <= leftEnd; i++)
      arr[i] = temp[i];
}
```

Time Complexity Analysis
Merge sort is considered an $O(n \log(n))$ algorithm. Note that it does take up more space than quick sort because it uses temporary arrays to store data, while quick sort sorts in-place.

4.7 Bucket Sort

Unlike the other sorting algorithms, I will not be writing too extensively on this sort, as Prof. Erman brushed over this concept quickly and did not even include any code for it. Also, make sure you have a good intuition on hash tables before you read this.

Since I have not provided too much information about this, feel free to check out the following links for some visualizations and explanations of bucket sort:

- `https://www.youtube.com/watch?v=VuXbEb5ywrU` - If you're limited on time, just watch this one. It's just over 2 minutes and does a good enough job explaining it.
- `https://www.youtube.com/watch?v=ROnYubH_spM` - This one is a little more in depth and contains Java code.

Concept
Let's say we have an integer array of size n. We use the concept of hash tables to sort this array. Since we know hash tables have constant time operations, we can simply use the value of any integer k in the array to hash itself into a hash table of size M. The hash function used will keep *general relative order* of the integer. To prevent collisions, the hash table uses the method of chaining. When integer k is added to any of these linked lists (called "buckets"), the bucket is sorted using insertion sort, as the intuition is that not many elements will be added to that bucket. Because the hash table keeps a general relative order of the integers, a sorted list can then be retrieved from the hash table in linear time.

Time Complexity Analysis
Bucket sort is considered an $O(M + N)$ algorithm, where M is the number of linked lists (buckets) in the created hash table. Note that since bucket sort also uses an additional data structure, it takes up much more space than previously discussed sorting algorithms. Additionally, note that if the data is evenly distributed, the time complexity of bucket sort reduces to $O(n)$, whereas if all values end up landing in the same bucket, it's essentially insertion sort $O(n^2)$.

5 Graphs

5.1 Terminology & Concept

The graph is a data structure consisting of nodes (also called vertices) and edges (also called arcs). An edge forms a path between two vertices.

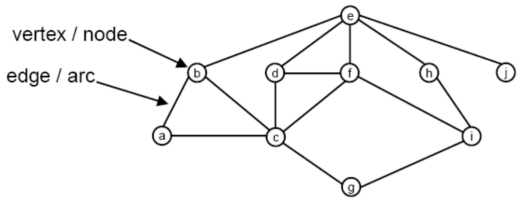


Figure 4: Circles represent nodes, lines represent edges

5.1.1 Weighted vs. Unweighted

Graphs can be weighted. Weighted graphs have predetermined costs associated with traveling along an edge. These costs are denoted by a number. The higher the number, the higher the cost.

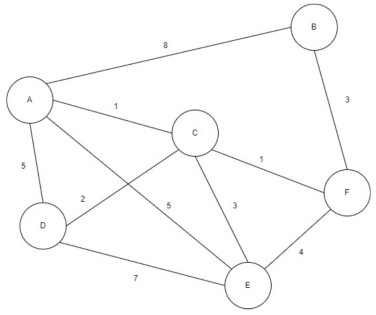


Figure 5: Weighted graph

Graphs can also be unweighted. This means that there is an equal cost for traveling along any edge in the graph.

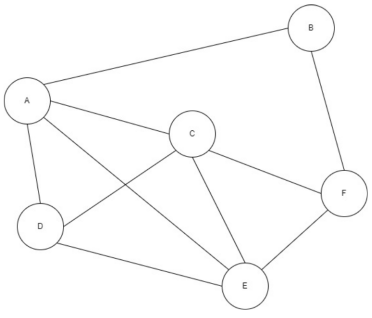


Figure 6: Unweighted graph

5.1.2 Directed vs. Undirected

Graphs can be directed. In diagram, directed edges are usually indicated through arrows. The arrows indicate that traversal can only occur in the direction the arrows are pointing.

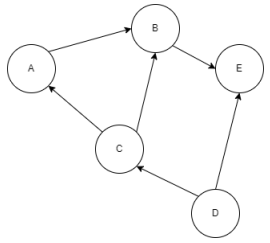


Figure 7: Directed graph - traffic can move from A to B, but not from B to A

Graphs can also be undirected. Traffic can flow either way. Edges are typically indicated using normal lines.

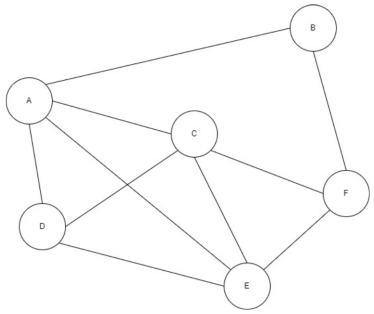


Figure 8: Undirected graph - traffic can move from A to B and from B to A

5.1.3 Connected vs. Disconnected

Graphs can be connected. Connected graphs imply that there exists a path between any two vertices.

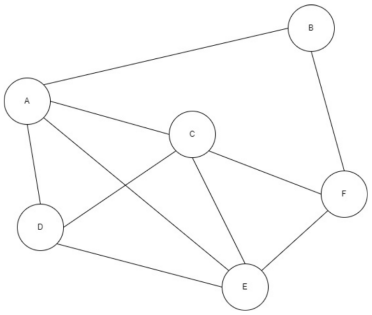


Figure 9: Connected graph

Graphs can be disconnected. Disconnected graphs have disjoint sets of complete graphs.

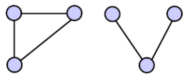


Figure 10: Disconnected graph

5.1.4 Trees vs. Graphs

A tree is a *type* of graph, as a tree is also consisted of a set of nodes connected by a set of vertices. Trees are also inherently undirected and connected. However, note that trees are *acyclic*, meaning that they have no cycles in their paths. This is what separates a tree from a "traditional" graph.

5.2 Spanning Trees

A spanning tree is a tree that is a subset of a connected graph that contains (1) all vertices of the graph and (2) a subset of the edges that form a tree.

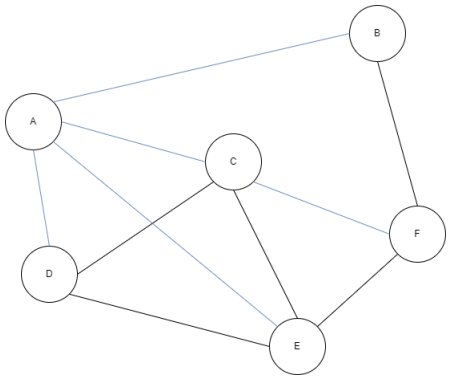
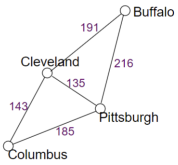


Figure 11: Minimum spanning tree (highlighted edges) - reaches all vertices of the graph and contains a subset of the edges to form a tree

5.3 Adjacency Matrices

An adjacency matrix simply lists the vertices that each vertex is connected to and their associated costs.



	Buf	Cle	Pitt	Col
Buffalo	X	191	216	?
Cleveland	191	X	135	143
Pittsburgh	216	135	X	185
Columbus	?	143	185	X

Figure 12: Adjacency matrix for graph to the left - For each row, the *cost* needed to traverse a neighbor is recorded in its respective column.

The "?" symbol is used to indicate any column vertex that the analyzed (row) vertex is *not* connected to. The "X" symbol is used to indicate that the row vertex is the same as the column vertex.

Note that adjacency matrices can also be represented as linked lists, which each list item holds the identity and cost of an adjacent node. In code, we denote an adjacency list of a specific vertex as a list of edges, represented by edge objects (see below).

5.4 Implementation

```
public class Graph {
   class Vertex {
      private String id;
```

```
      private LinkedList <Edge> edges; //adjacency list

      //useful for future methods
      private boolean encountered;
      private boolean done;
      private Vertex parent;
      private double cost;

      //more Vertex methods below...
   }

   class Edge {
      private int endNode;
      private double cost;

      //more Edge methods below...
   }

   private Vertex[] vertices;
   private int numVertices;
   private int maxNum;

   public Graph(int maximum) { //graph constructor
      vertices = new Vertex[maximum];
      numVertices = 0;
      maxNum = maximum;
   }

   public int addNode(String id) {
      vertices[numVertices] = new Vertex(id);

      numVertices++;
      return numVertices-1; //return node index
   }

   public void addEdge(int i, int j, double cost) {
      Edge newEdge = new Edge();
      newEdge.endNode = j;

      vertices[i].edge.add(newEdge);

      newEdge = new Edge();
      newEdge.endNode = i;

      vertices[j].edges.add(newEdge);
   }
}
```

5.5 Traversals

Traversing a graph refers to the concept of visiting every vertex in the graph by using the graph's edges (paths). Note that a traversal can only occur on connected graphs. In this course, we have studied two different types of traversals.

5.5.1 Breadth-First Traversal

Concept
The general process of breadth-first traversal is as follows:

1. Begin at the starting vertex.
2. Visit all of its adjacent (neighboring) vertices.
3. Visit all of the starting vertex's unvisited neighbors neighbors (unvisited vertices 2 edges away from the starting vertex).
4. Visit all unvisited vertices 3 edges away from the starting vertex.
5. etc...

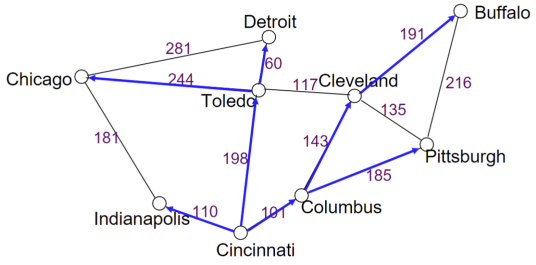


Figure 13: Breadth first traversal starting from Cincinnati - prioritize visiting all neighbors before moving to "deeper depths"

Remember level-order traversal of trees? This is very analogous to breadth-first traversal on graphs! There are some minor differences, but the general idea holds up!

Note that the order you choose a neighbor to examine doesn't really matter, as long as you prioritize visiting all neighbors first. For example, in the breadth-first traversal picture, if we start from Cincinnati, it is just as valid to visit Indianapolis, Columbus, Toledo (in that order) as it is to visit Columbus, Toledo, Indianapolis. They are all neighbors of the same "depth"; relative order within the same "level" does not matter.

Implementation
Pseudocode for breadth-first traversal is provided on lecture slides 22. I'm prioritizing putting down the code that Prof. Erman put actual Java for, not

pseudocode.

5.5.2 Depth-First Traversal

The general process of depth-first traversal is a follows:

1. Begin at the starting vertex.
2. Proceed as far as possible along a given neighbor path before "backtracking" and going along a new path.

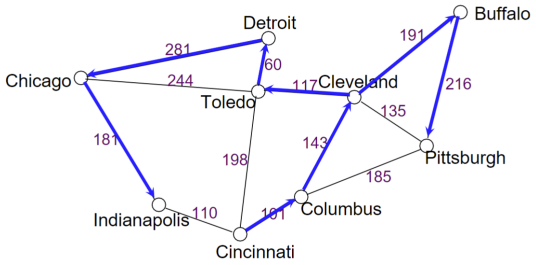


Figure 14: Depth-first traversal starting from Cincinnati - follow down a certain path until cannot anymore; backtrack and start along a new path from node closest to dead end node

Like in breadth-first traversal, the node that you choose to traverse to doesn't really matter, as long as it follows the depth-first traversal scheme. For example, in the depth-first traversal picture, instead of picking Columbus to visit first, they could have picked Toledo. Also notice the differences between this image and the breadth-first image. Depth-first traversed to Pittsburgh (or Indianapolis, depending on how you look at it), reached a dead end, and backtracked to another city from the node closest to the dead end.

Implementation

```
public void depthFirstTrav(int i, int parent) {
   System.out.println(vertices[i].id); //print out the id of the current
       vertex
   vertices[i].encountered = true; //indicate that the vertex has been
       visited already to not revisit it
   vertices[i].parent = parent;
   Iterator<Edge> edgeItr = edges.iterator(); //an iterator object to
       "loop" through the list of edges of the vertex (its adjacency
       matrix, essentially)
   while (edgeItr.hasNext()) {
      Edge curEdge = edgeItr.next()
```

```
      j = curEdge.endNode;
      if (vertices[j].encountered == false)
         myDepthFirstTrav(j, i);
   }
}
```

Please note that this code follows from the code shown in the implementation of graphs.

5.6 Algorithms

Both algorithms are written in pseudocode on the slides. I am not writing pseudocode on these sheets. Please visit lecture slides 23 for Dijkstra's algorithm pseudocode and lecture slides 24 for Prim's algorithm pseudocode.

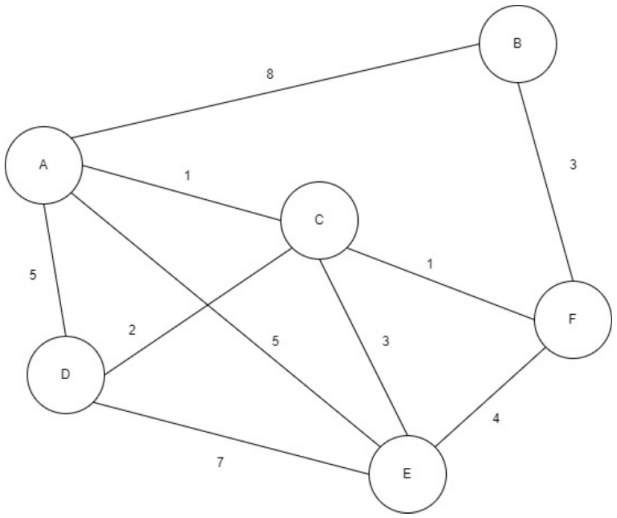
5.6.1 Dijkstra's Algorithm

Dijkstra's algorithm aims to find the shortest path from one vertex to all other vertices in the graph. This concept is a little more difficult, so I will go over it in more detail.

Concept
Dijkstra's algorithm follows this general procedure:

1. Begin at the starting node. It is your current node. Trivially, the shortest path to your current node from the starting node is 0. Take note of this.
2. Record the current node in a list of "already visited nodes".
3. Identify all neighbors of the current node. Take mental note of the costs to each of them and add each cost the cost it took to reach the node you're currently at.
4. For each calculated cost, if it is less than a shortest path already recorded to a node or there is no recorded path already, save it as the new shortest path to that node.
5. Select the node with the lowest calculated cost that hasn't been visited already. This is your new current node. Restart from step 2.
6. Once all nodes have been visited, the list should show the shortest paths to each node from the starting node.

Example
Let's start off with an example. Assume the following graph.



Let's say we want to find the shortest distance from node A to all other nodes. We can create a table with the following columns:

v	**vertexSet**	**A**	**B**	**C**	**D**	**E**	**F**

where v is the current vertex we're analyzing, *vertexSet* is a set of all of the vertices we've found the shortest path for, A, B, C, D, E, and F all represent the shortest path from A to that respective node.

Following our instructions from above, we first begin at the starting node, which is now our current node (A). Trivially, the shortest path to A from A is 0. Let's mark that down! Remember that the letter columns represent the distances of the shortest paths from A to that respective node. Let's also make sure to record A in the set of "already visited nodes" (*vertexSet*).

v	**vertexSet**	**A**	**B**	**C**	**D**	**E**	**F**
A	{A}	0					

Now, we identify all neighbors of A and record the cost to get there from A by adding the cost to A to the cost from A to X, where X is your destination. For example, the cost of B is the cost to get to A plus the cost to get from A to B, which is $0 + 8 = 8$. Since there is no shortest path indicated in slot B, let's mark it down! We also do the same for the rest of the letters. Note that since there is no direct path from A to F, we temporarily mark it as infinity, since we currently have not found any path to get to F from A.

v	**vertexSet**	**A**	**B**	**C**	**D**	**E**	**F**
A	{A}	0	8	1	5	5	∞

Following our directions, we then move to the node with the current shortest

path that has not already been analyzed. In this case, we move to node C, because it has the current shortest path (1) that has not been analyzed yet.

Why do we choose the node with the shortest path that has not yet been analyzed? This is because we are 100% *certain* that we have found the shortest path to this new node, in this case C. There is absolutely no way that C has a more optimal path from A. We know this because all other paths from A are *greater* than or equal to the cost from A to C, and under the assumption that all edges in the graph are positive, there will never be an instance where we find a path that costs less than this path we just found.

Because we now know that we've 100% found the shortest path from A to C, we can analyze C and put it in our vertexSet. Let's also update the any new optimal paths we find. Let's analyze each letter.

- A: For A, the optimal path is still (and will always be) 0.
- B: For B, there is no path from C to B, so therefore the most optimal path found is still 8.
- C: For C, we have already established that 1 is the optimal path.
- D: For D, we see that C *does* have a path to D. If we apply the formula of adding the cost to get to C plus the cost to get to C to D, we see that we have $1 + 2 = 3$. In other words, we are looking at the path from A to C to D and have found the total cost of this path to be 3. Now if we take a look at our chart, we currently have 5 listed as the most optimal path from A to D. Since we found a more optimal one, let's replace that value with our newfound cost (3).
- E: For E, we see a similar situation that we had to D. Our old cost to E from A was 5. However, from C, we see that our new shortest path is cost to C plus cost from C to E, which is $1 + 3 = 4$. Since $4 < 5$, we replace E's shortest path with 4.
- For F, we can see that C has opened a path to F that has a lower cost than ∞. Cost of C plus cost from C to F is $1 + 1 = 2$.

This leaves our new table to be...

v	**vertexSet**	**A**	**B**	**C**	**D**	**E**	**F**
A	{A}	0	8	1	5	5	∞
C	{A, C}	0	8	1	3	4	2

The next node to select should be F because it has the shortest path that has not yet been analyzed. Like last time, we know that we have found the most optimal route to F by the same logic. Continuing the pattern, the table should look like this at the end:

v	vertexSet	A	B	C	D	E	F
A	{A}	0	8	1	5	5	∞
C	{A, C}	0	8	1	3	4	2
F	{A, C, F}	0	5	1	3	4	2
D	{A, C, F, D}	0	5	1	3	4	2
E	{A, C, F, D, E}	0	5	1	3	4	2
C	{A, C, F, D, E, B}	0	5	1	3	4	2

After we have added every node to *vertexSet*, we know we've found the most optimal path from A to every other node in the graph.

5.6.2 Prim's Algorithm

Prim's algorithm aims to find a mininum spanning tree originating from a specified root node. A mininum spanning tree from a node is a spanning tree that has the most optimal cost possible (lowest possible cost tree spanning from specified node).

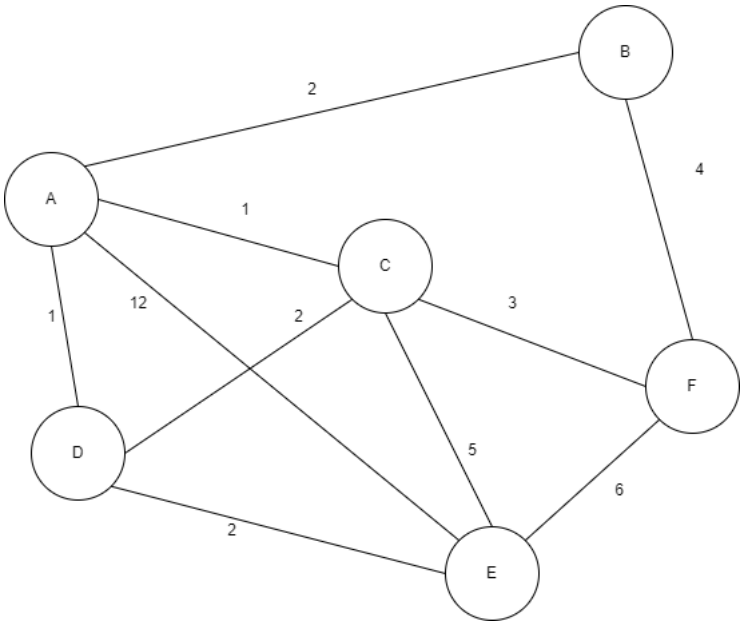
Assumptions

- The graph is connected.
- The graph is undirected.

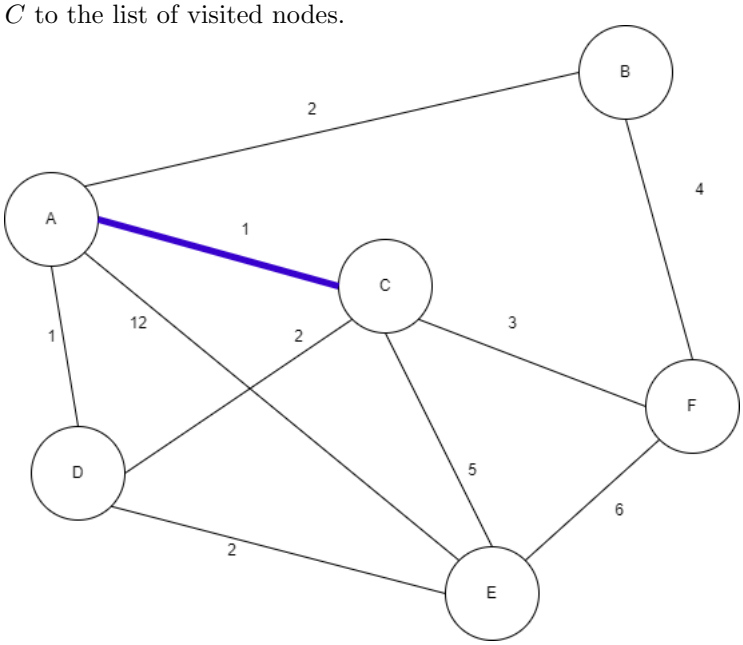
Concept
The general process of Prim's algorithm is as follows:

1. Begin at the starting node. Add this to a set of visited nodes.
2. For all nodes in the set of visited nodes, find which neighbor takes the least amount to get to, out of all the neighbors that you haven't been to before.
3. Add this found neighbor to set of visited nodes and mark down the edge that you traveled down to get to this neighbor.
4. Go back to step 2.
5. Repeat this step until all vertices have been added to the visited nodes set.

Example
Let's assume you have the graph below, and you're trying to find the minimum spanning tree from A.

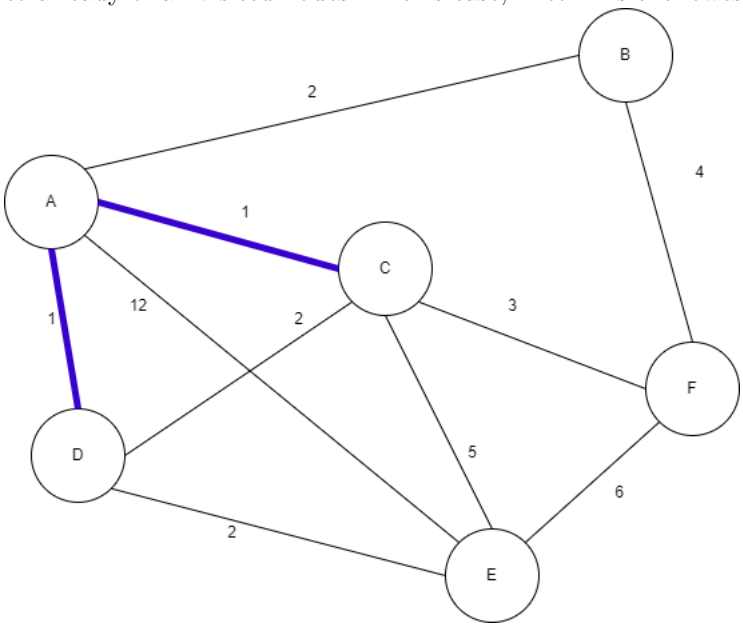


Begin at the starting node and add this to your set of visited nodes. Now, for all of the visited nodes (there's only one at this point), find the neighbor that you haven't visited with the lowest cost to get to. In this case, the neighbor that fits this requirement is along the path from A to C. Note that we could have also picked A to D, as the two edges are the same weight (1). Now we add C to the list of visited nodes.

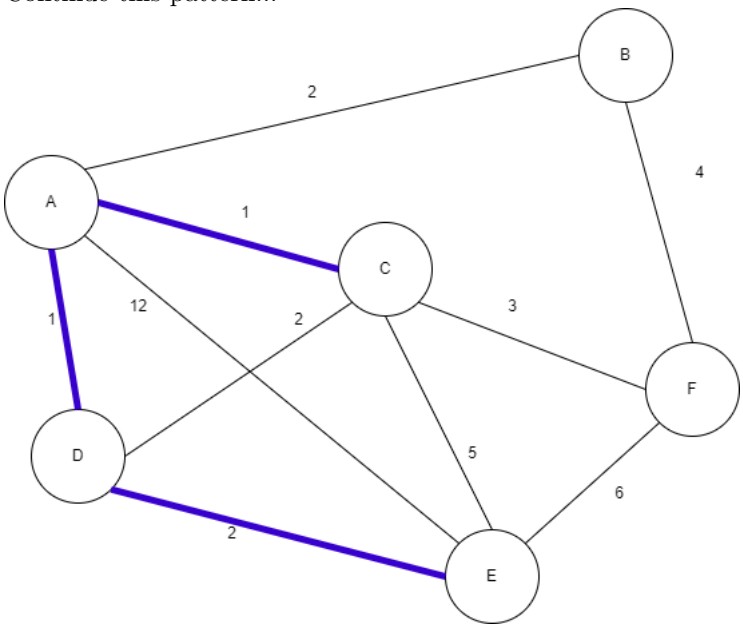


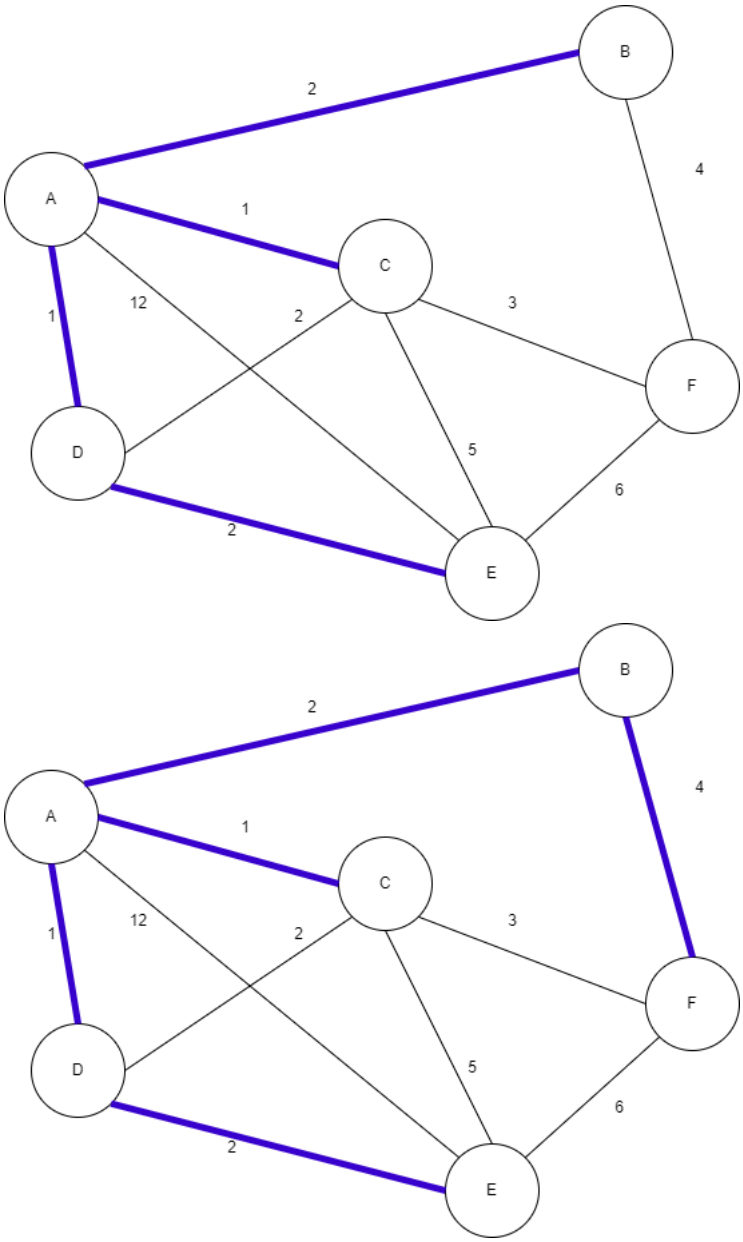
Repeating the process, examine the neighbor of lowest cost that we haven't been

to already of all visited nodes. In this case, A to D is the lowest at 1.



Continue this pattern...





This is our minimum spanning tree originating from A. There exists no spanning tree originating from a such that its total cost is lower than the spanning tree that we found, hence the "mininum" spanning tree.