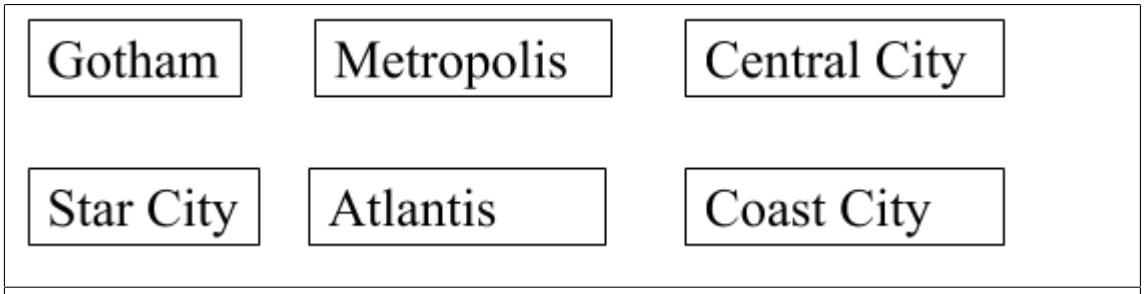
Practice Midterm 2

You have 2 hours to complete this exam. This exam is meant solely for practice and topics that are not in this exam may be covered while topics in it may not be covered. This exam is out of 100 points. Every line may have at most one statement (including closing brackets).

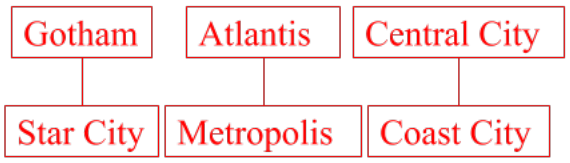
Problem	Points
1	12
2	5
3	6
4	7
5	8
6	12
Total	50

1 Flash Union

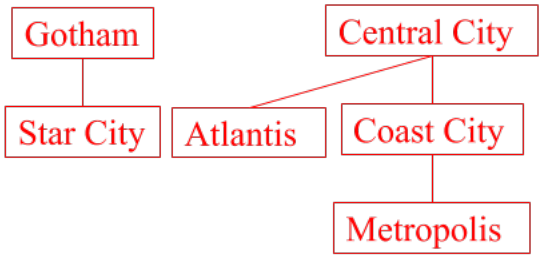
(a) Beary Allen is a fast UC Berkeley student wants to connect the world. He has a set of cities and wants to connect cities and see if they are connected in the fastest way possible. To do this, he will us the fastest Quick Union Data Structure that we have learned. Draw the Quick Union Data Structure in box that corresponds to how the Data Structure looks after all the calls in that box. Note that calls occurs sequentially. This means the Quick Union Data Structure in Box 2 is the starting Structure for Box 3. If there is a tie, break it by making the left tree corresponding to the left item the root.(5 pts)



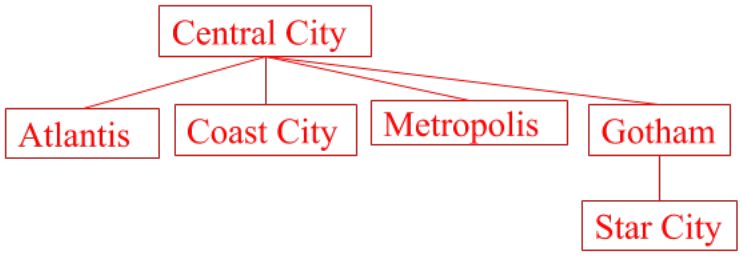
connect(Gotham, Star City); connect(Atlantis, Metropolis); connect(Central City, Coast City);



connect(Metropolis, Central City);



connect(Metropolis, Star City);



2 Dat-uhhh Structures

(a) How many nodes would be in the left subtree of a complete binary search tree where height (h) > 0 and the root starts at a height of 0. Note, if you want to use height in your calculations, you must use the height of the **full binary search tree rooted at the original root**, not subtrees.

Solution:
$2^h - 1$
Explanation:
The amount of nodes in a perfect Binary Search Tree is 2h+1 - 1 which can be seen via observation. Since the tree is perfect, there will be the same amount of elements to the left and right of the root. This means each time you go down one level in the tree, you "throw away" half the items you are looking at. When looking at the left subtree, we throw away half the elements as well as the root which means that we would have 2h -1 to look at in the left subtree

(b) How many nodes would be in the right subtree of the right subtree in a full binary search tree (so the tree rooted at the right child of the root's right child). The same assumptions should be made as in the previous problem.

Solution:
$2^{h-1} - 1$
Explanation:
This can be seen by drawing a basic binary search tree such as the one above. It probably would have been better to use a larger one than the one provided as, in the tree above, there was only 1 node in the right subtree of the right subtree. The logic follows from the prior problem just extended to the next level.

(c) What is the criteria that a heap must fulfill so that it can be a min heap and max heap at the same time.

Solution:
Infinitely many
Explanation:
The elements can all be the same

(d) Can red-black trees have only black links (so no red links/nodes)? When is this the case?

Solution:
Yes, it is possible
Explanation:
it is possible for red black trees to only have black links. Think about it in terms of 2-3 trees. Red links in Left Leaning Red Black Trees connect 2 nodes that are part of the same 2 node in a 2-3 Tree. For there to be no red links, there must be no 2 nodes in the 2-3 tree meaning there are only "1" nodes. If this is the case, both the red black tree and 2-3 tree would look like normal Binary Search Trees. An interesting observation that can be made is that, for this to happen, the tree must have 2n-1elements, where n is some integer. This is because, based off our balancing techniques, we attempt to get as close to a log(n) height as possible. At time where there are 2n-1elements, it is possible to reach exactly log(n) height, this is done making the tree a perfect Binary Search Tree (complete with leaves on the same level). Try it out on a 2-3 tree or red black tree for verification.

(e) What is the maximum amount of children a "QuadTree" that works in 3 dimensions can have?

Solution:
8
Explanation:
2^3. We do this since we have 2 possible ways to go for 3 dimensions.

(f) What modification would you have to do to a K-D Tree to account for 3 dimensional nodes?

Solution:
We would need to alternate every 3 levels instead of every 2. So if our dimensions are x,y,z. We would traverse to left/right based on if x value is smaller or larger, the y value on the second level, and the z value on the third (order doesn't matter but must do all 3 before repeating).
Explanation:
See above

(g) What is wrong with the following code snippet? Assume that it compiles properly, there is a constructor, and a hashcode method that has a hashcode function works well (spreads everything well).

```
1 Public class Human{
2         int legs;
3         int arms;
4         @Override
5         public boolean equals(Human no) {
6                 return legs == no.legs && arms == no.arms;
7         }
8         ....
9 }
```

Solution:
The main problem with this code is that the equals does not actually override any methods! The signature is wrong. Normally, the equals takes in an Object not a Human. This will cause errors when a non "Human" is inputted.
Explanation:
See above

3 Asymptotics

Write the Asymptotic runtime of the following function. Use the tightest bounds possible. Each problem is worth 4 points.

```
1  public static void mango(int N){
2          if(N<=1){
3                  return 1;
4          }
5     for int(i = 0; i < N / 2 ; i ++){
6          System.out.println('you');
7      }
8      mango(N - 1); mango(N-2);\\lie
9  }
```

Solution:
$O(N * 2^N)$
Explanation:
We must realize that there are N layers. Additionally, the amount of work done per node is N. The nodes per layer is 2^i ,where i is the current layer, because of the 2 recursive calls. The summation would look as follows:

$$\sum_{i=0}^{N} N * 2^i = N(2 + 4 + 8 + ...2^n = 2N2^n = O(N * 2^n)$$

```
1  public static void stowb(int N){
2          if(N <= 1){
3                  return 1;
4          }
5          for(int i = 0; i < (N*1000)/N ; i ++){
6                  System.out.println("erry");
7          }
8          stowb(N/4);
9  }
```

Solution:
$\Theta(log(n))$
Explanation:
The amount of work done in the for loop is constant (though a very large constant), so we disregard it and just consider it constant time. The amount of layers would be log4N. Since we have constant work on each layer, 1 node per layer, and $log_4 N$ layers, the runtime is $O(log(n))$.

```
1  public static void grongrula(int N){
2          if(N<=1){
3                  return N;
4          }
5          for(int i = 0 ; i < N; i++){
6                  for (int j = 0 ; j < i; j++){
```

```
7                               System.out.print("Huh?");
8                       }
9               }
10              grongrula(N/2)
11      }
```

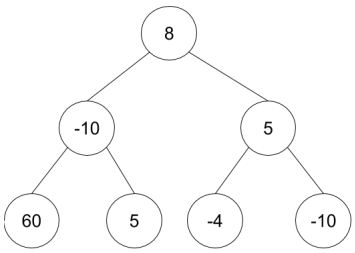
Solution:
$\Theta(N^2)$
Explanation:
The amount of work done in the second for loop is proportional to the work done in the first for loop- for the first iteration, this will be N^2; however, we have a recursive loop that we must take into account. The amount of work done per now is $(\frac{N}{2^i})^2$ or $(\frac{N}{4^i})^2$) where i is the current layer. The amount of layers is log(N) because we divide by 2 at each recursive call. This makes the summation formula:

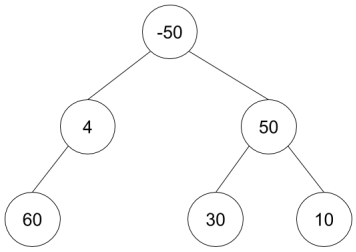
$$\sum_{i=0}^{log(n)} (\frac{N^2}{4^i}) = (\frac{N^2}{4^{log(N)}})$$

4 Arithmetic Trees

This problem deals with finding different types of paths inside of a Binary Tree. A path is defined as a set of nodes between any 2 nodes in the tree (This means the root may not be part of a path).

(a) To start off, we will attempt to find the maximum path sum in a Binary Tree. This means finding the path for which the sum is maximized. Your path must have at least one node but can have more. For example, the tree below on the left has a maximum path of 60+-10+8+5 = 63. On the right, we have a maximum path of 3+50+10 = 63





You are given 2 methods, a *pathsum* method, which returns the maximum path sum and a helper method *pathsumhelper* which aids in this endeavor.
Below is the api needed for a BSTNode

```
public class BSTNode{
    int val;
    BSTNode left, right;
    BSTNode(int d, BSTNode left, BSTNode right){
        val = d;
        this.left = left;
        this.right = right;
    }
}
```

Fill in the code below to finish finding the maxPathSum:

Solution:

```
public class BST {
    private int max_so_far = Integer.MIN_VALUE;
    public int pathSum(BSTNode t){
        maxPathSum(t);
        return max_so_far;
    }

    private int maxPathSum(BSTNode t){
        if (t == null) {
            return 0;
        }
        int leftSum = Math.max(maxPathSum(t.left),0);
        int rightSum = Math.max(maxPathSum(t.right),0);
        max_so_far = Math.max(t.val + leftSum + rightSum, max_so_far);
        return Math.max(t.val+leftSum, t.val+rightSum);
    }
}
```

Explanation:
See Above

(b) What is the runtime of your algorithm in part a in terms of N where N is the number of nodes in the Tree.

Solution:
$O(N)$
Explanation:
We only touch each node once

(c) Now say that instead of a normal Binary Tree, we were dealing with a **Binary Search Tree**. What optimization could we do to *MaxPathSum* to speed up our code? No need to write out code just write what you would do.

Solution:
In a Binary Search Tree, we can use the ordered property to our advantage. Once we see the node 0, or any negative number, we know not to traverse to the right anymore. This will not make an asymptotic difference, but could help your code run faster.
Explanation:
We only touch each node once

(d) What is the runtime of your algorithm in part c in terms of N where is the number of nodes in the Tree.

Solution:
$O(N)$
Explanation:
We only touch each node once. The optimization is not guaranteed to speed up our traversal at all.

(e) Say we wanted to find the maximum product path in a normal **Binary Tree** where the maximum product path is defined as a path where the product of the numbers of each node in the path is its maximum value. What would you need to account for? (this is an open ended question)

Solution:
We would have to account for the fact that negatives can flip the signs which means we would have to store the max and min at any given point since the negative signs can flip this.
Explanation:
See above

(f) What constraints would we need to have on the nodes of the input **Binary Tree** such that you could take the above algorithm for finding the max path sum and replace all operations for addition with their corresponding multiplication ones and have it function without any other logic changes.

Solution:
We must not have any negative numbers.
Explanation:
See above

(g) Finally, say we wanted to find the maximum product path in a **Binary Search Tree**. What optimization could we do to speed it up? (Hint this combines parts c and g).

Solution:
Once we reach a number less than 1 we will never go to the left again.
Explanation:
See above

5 MashedMap

1. We have a HashMap, but a goon did something so that duplicates were not handled properly. This means that multiple versions of a key can exist in our HashMap. Luckily, your hashcode is still intact. In addition to this, the spread of the hashcode is pretty good and it distributes elements fairly evenly.

   Determine how long it would take in order to traverse your broken HashMap to find and delete ALL duplicates of ONE key. For example, how long would it take to find the key "macaroni". Provide the runtime (It is your choice what symbol to use) based off our prior assumptions. Give reasons for both runtimes. Correct runtimes with wrong explanations will be given 0 points.

   Solution:
   $\Omega(1)$
   $O(N)$
   Explanation:
   The Omega bound is find by the following reasoning:
   We have a good hashcode and a good spread, and since the key would be the same, it would always hash to the same bucket, this means that you would have to check very few elements.

   The O bound is found by the following reasoning:
   Technically this can happen if every single element in our HashMap is the same. It is not acceptable to say that the hashcode is bad.

2. You know for a fact that your HashMap class is perfect. You spent tireless hours in your 61B Lab making sure you had no flaws in your code of the HashMap. Assuming that your HashMap code is perfect how would this goon be able to ruin your duplicate handling given that he can only input items into your HashMap?

   Solution:
   A change in the equals method for the item being hashed would result in the duplicates not being handled. Your equals method normally goes through the bucket to check if the element is not working, but if your equals method does not work, it won't realize there is a duplicate.

   A NOT acceptable response would be that the hashcode function was screwed up as in the original problem it was stated that it remained intact.
   Explanation:
   See Above

3. Pretend that you did not remove the duplicates from the faulty HashMap and you wanted to keep track of how many times each key was inside of the HashMap. How would one do this? We would like our solution to be as fast as possible.

   Solution:
   The most intuitive way would be to create a HashMap where you insert elements as you go through the original HashSet. The element would be a node that has a the key and a value that is the count. Every time you found a duplicate, you would find the node with the corresponding and increment the value by 1. This would take $\Theta(1)$ because we would use the SAME hashcode that was provided to us earlier in the problem. The overall runtime as a result would be $\Theta(N)$
   Explanation:
   See Above

6 Riddle Me This

What president wears the biggest hat?

Solution:
The one with the largest head
Explanation:

7 Runtimes not Finished Call that Not Done Times

Write all runtimes in Theta if there is a tight bound, if there is not, write your runtime in terms of Omega and O. Assume nothing other than what is given in the problem.

a) Runtime of putting N presorted keys (with no duplicates) into a binary search tree.

Solution:
$\Theta(N^2)$
Explanation:
The items are presorted (either least to greatest or greatest to least) and this means that the Binary Search Tree will end up looking like a LinkedList

b) You have a minheap and you will perform deleteMin operations until the min-heap is empty. Whenever an element is deleted, it is added to a Binary Search Tree.

Solution:
$O(N^2)$
Explanation:
Deleting the min takes log(N) time and you perform that operation N times. The N^2 comes from the worst case runtime of inserting into a Binary Search Tree being O(N) time and that operation is performed N times ($N * N = N2$).$O(N^2 + NlogN) = O(N^2)$

c) Inserting N elements into a HashMap that all have the same key and different

Solution:
$\Theta(N)$
Explanation:
You insert N elements and since HashMaps allow for no duplicates, you will never have more than 1 element in your HashMap.

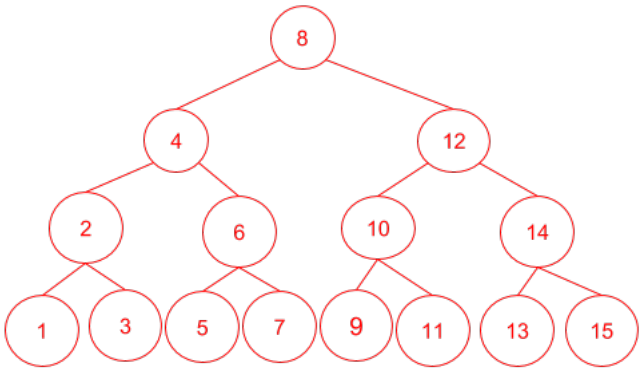
d) Given an array sorted from least to greatest (lowest index on left, highest index on right and no duplicates). Iterate through the array from left to right and at each index, insert the corresponding element into a stack. Finally, pop every element from the stack and insert it into a maxheap.

Solution:
$\Theta(N)$
Explanation:
The items are presorted and since it is put into the stack, the greatest elements are always above the smaller elements. As a result, no swimming will ever need to be done. We put N elements into a stack, pop off all N elements and insert them into a heap with no swimming so N insert operations. This results in 3N or $\Theta(N)$

8 Rotations and Flips

a) What series of inserts would result in a perfectly balanced binary search tree for the following input: 1, 2, 3, 4, 5, 6 , 7 , 8 , 9 , 10, 11, 12, 13, 14, 15. Draw the

Solution:
8, 12,4, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15

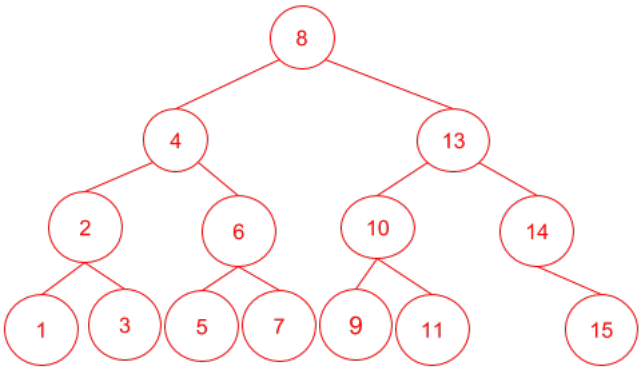


Explanation:
One way to make this problem a bit less overwhelming is to perform binary search. Find the middle element and then make it the root, then make its left child the middle element of the left partition and the right child the middle element of the right partition.

Another way this could be made a bit easier to approach is construct a 2-3 tree and see where the nodes get promoted to. It decreases the randomness and allows you to do it relatively quickly.

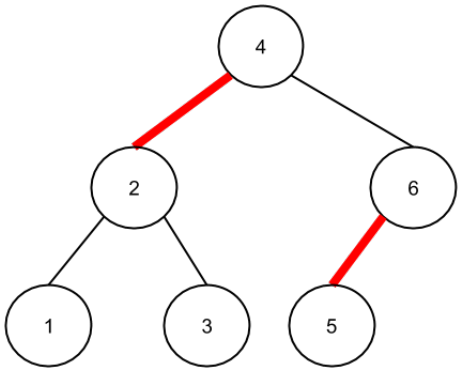
b) Delete the item 12 from the above Binary Search Tree and draw the results.

Solution:

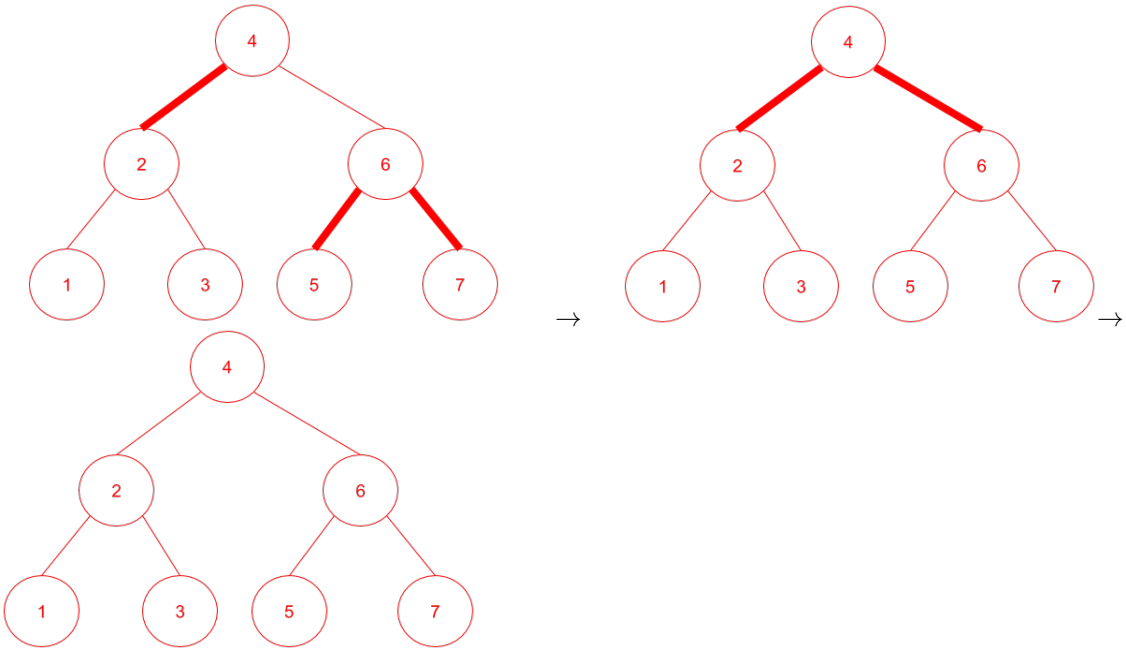


Explanation:
To do this, you want to do the "hibbard" deletion where you find the smallest item on the right side of the element (or largest on the left side) and then swap it with the item.

c) Insert 7 into the following red black tree. Show your steps using the red-black tree method for full credit. $\frac{1}{2}$ credit will be given for converting into a 2-3 tree and back. Denote red links with a dotted line.



Solution:



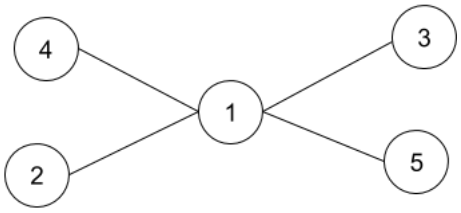
Explanation:

d) How many rotations that will occur if you insert 5.5 into the original red black tree?

Solution:
4
Explanation:
You insert 5.5 to the right of 5. Because red links cannot be on the right, we will rotate 5 to the left. Then we need to rotate 5.5 to the right since we would have to red links in a row. From there, we have a red right link so we do a color swap yielding a root with left and right red links. We do one more color flip and are ok then. You could also get this problem you simply do one more rotation then you get the above red black tree.

9 Graphical Blow to the Jaw

We have dealt with Breadth First Search in the class. Now we will go over another similar problem, bidirectional Breadth First Search. The jist of this problem is that we want to see if a path exists between 2 vertices in a graph. Instead of doing normal Breadth First Search, we choose to run 2 Breadth First Searches concurrently– one from the start vertex and one from the goal vertex. We then return the vertex that the searches intersect on. If the 2 vertices are not connected, we return -100. For example, in the below graph, we would return 1. Don't worry about accounting for odd lengths (they don't matter).



You will have to fill in 1 method to fill, *bidirbfs*, which takes in a source vertex s and a goal vertex g and returns the output of bidirectional BFS (the point at which the two BFS's meet). In addition to this, there is a helper method, *iteration* defined for you– cross it out if you choose not to use it.

Below are some methods for a graph that may be useful. Note all vertices are nonnegative consecutive integers in this problem (0,1,2,3.....N).

```
degree(int v); \\Returns the degree of some vertex v.
V(); \\Returns the number of vertices in a graph
E(); \\Returns the number of edges in the graph.
adj(v); \\Returns an iterable with all the vertices adjacent to v
addEdge(int v, int w); \\Adds an edge between vertices v and w
```

```
public class BiDirectionalBFS {
    Graph G;
    private class BFS {
        Queue<Integer> queue;
        boolean[] marked;

        BFS(int count, int source) {
            marked = new boolean[count];
            queue = new Queue<Integer>();
            queue.enqueue(source);
            marked[source] = true;
        }
    }
```

Solution:

```
    private int iteration(BFS curr, BFS other) {
        int v = curr.queue.dequeue();
        for (int w : G.adj(v)) {
            if (curr.marked[w] == false) {
                curr.marked[w] = true;
                curr.queue.enqueue(w);
            }
            if (other.marked[w] == true) {
                return w;
            }
        }
        return -1;
    }

    public int bidirectionalbfs(int s, int g) {
        BFS sbfs = new BFS(G.V(), s);
        BFS gbfs = new BFS(G.V(), g);
        while (!sbfs.queue.isEmpty()) {
            int v1 = iteration(sbfs, gbfs);
            int v2 = iteration(gbfs, sbfs);
            if (v1 != -1) {
                return v1;
            } else if (v2 != -1) {
                return v2;
            }
        }
        return -100;
    }
```

Explanation:
N/A

10 Josh's 61BBQ and Foot Massage

You are the Chef at a famous restaurant, *Josh's barbecue et Massage des Pieds*[1]. In your restaurant you have N possible menu items and you have M customers who come every day (you're a really good cook in this world). Each customer has a specific I.D number going from 1 to M.

You have a specific policy– once one customer has had one item from your menu, no one else can have it. To be fair, you have asked to customers to rank each of the N items with an integer from 1 to 100, where 1 is "61Bad" and 100 is "61Best". For every number between 1 and 100 a customer can rank $\frac{N}{100}$ items with that number; in other words there will be items that a customer ranks with the same number.

Every day, you will go through the customers in order of their I.D (from 1 to M) and create the order corresponding to the item which they rank the highest (as close to 100 as possible) that has not yet been picked. Once all menu items have been finished, you close your restaurant.

Your goal is design a system that does the following:

- Instantiating the preferences for all customers' preferences should take $O(MN)$ time
- Pick the item a customer should eat in $O(1)$ time
- After picking the item a customer should eat, it should be removed from everyone else's options in $O(M)$ time.

Use the below space to describe how you would use Data Structures we have learned in class (Arrays, LinkedLists, Stacks, Queues, Weighted Quick Union, Binary Search Tree, Priority Queues, Tries, Hashtables, 2-3 Trees, Red Black Trees, Quad Trees) to complete the problem.

[1]See the following link for more information on the restaurant https://tinyurl.com/JoshBBQ

Solution:
The setup is as follows:

- For each food item on the menu, create a wrapper class which has M Doubly LinkedList dummy elements (1 for each customer). The dummy element should also have some mapping back to the original food item.
- Create a Hashmap which contains a User Object.
- Inside each User Object create a Priority Queue of Doubly LinkedLists where $LinkedList_{10,5}$ corresponds to all the foods which customer number 10 ranked as 5.

The way we go through our algorithm is as follows:

1. Every day you iterate through the numbers $1...M$.
2. For the ith number in this list, you use your hashmap (or array) to get the ith user's priority queue.
3. In the Priority Queue you peek the top element and you remove the first node in the LinkedList. Call this node p. Thir process is $O(1)$ since getting from a Hashmap and peeking from a Priority Queue is $O(1)$
4. Using your mapping, you go back to the original food item for p and iterate over all the other dummy nodes that were created in it and remove these nodes from their corresponding LinkedLists in the other $1....i-1, i+1....m$ Priority Queues.
5. In addition to this, once a LinkedList is empty for some PriorityQueue, we want to pop it off the PrioirtyQueue since it no longer has any valid food choices. Because our Priority Queue only has 100 elements total, this is constant time. Removing all the nodes corresponding to a food only takes $O(M)$ time because removing a node from one LinkedList takes $O(1)$ time and popping from our Priority Queue is $O(1)$

Explanation:
N/A

11 Mini Algorithm Design

1. Give a brief description of an algorithm that you would use to get a minheap from a 2-3 Tree in $O(N)$ time where N is the amount of nodes in your tree.

   Solution:
   Perform an in order traversal of the 2-3 Tree. Modify the visit operation such that once you "reach a node" in the traversal and output it, you also input it into a min heap.
   Explanation:
   Performing an Inorder traversal of a Binary Search Tree will yield the items in sorted order. Because of the way we insert into a heap, when we add an element we will never need to swim up since we are inserting the items in sorted order. This means adding to the heap is a constant time operation for every insert, taking $O(N)$ time overall. Since the traversal is also $O(N)$, we have $O(N)$ runtime total.

2. We have a set of N words, in an alphabet with M characters, all of length L. However, we know that the $L - 10$ letters are all the same (the last $L - 10$ letters are all the same in the same order and we have knowledge of the first 10). How can we construct a Trie such that the following occurs

   - Query to see if any word exists in our Trie takes constant time.
   - Adding a word to our Trie, given it follows the same scheme as above takes constant time.

   Solution:
   Because we know that the strings are similar after the first 10 letters, we can simply cut off the Trie after the first 10 digits. Then when we look in our Trie, we do not have to traverse down an L length string, only 11 length. Similarly for insertion, we would only insert until the 10th letter.
   Explanation: