Part A: Trees

For questions in this part, you will use the following values: `[6, 7, 8, 11, 12, 13, 21, 22, 23]`

Question 1 [**5 points**]

Draw a Min Heap (as a tree) containing the values above. It can be any valid Min Heap that contains exactly the above values.

Answer

Any binary tree with the exact 9 nodes given such that every parent is less than its children, and the tree is complete and filled left to right. Therefore level 1 should have 1 node, 2 should have 2, 3 should have 4, and 4 should have 2 nodes all the way to the left.

Point Breakdown

- **(+2 pt)** If the tree has the right structure (complete filled from left to right)
- **(+2 pt)** If every parent is smaller then both of it's children
- **(+1 pt)** If exactly the correct values are present
  Var A: 6,7,8,11,12,13,21,22,23
  Var B: 4,5,6,11,12,13,25,26,27
  Var C: 6,7,8,14,15,16,21,22,23
  Var D: 4,5,6,11,12,13,21,22,23

Question 2 [**5 points**]

Write out the ArrayList representation of the Heap you drew in the previous question.

Answer

An array containing the nodes from their previous answer read from left to right, top to bottom.

Point Breakdown

- **(-1 pt)** For each value that is not in the right place BASED ON THEIR HEAP DRAWING

Question 3 [**5 points**]

Draw a BST containing the values above with a **height** of 6. It can be any valid BST of height 6 that contains exactly the above values.

Answer

Any binary tree such that for every node smaller values are to its left and larger values are to its right. It must have a height of 6, meaning there are 6 edges from the root to the deepest leaf.

Point Breakdown

- **(+2 pt)** If the tree has the right structure (binary, and has a height of 6)
- **(+1 pt)** If the tree has 6 levels instead of a height of 6
- **(+2 pt)** If the ordering is correct (for EVERY node smaller values are to the left, larger to the right)
- **(+1 pt)** If exactly the correct values are present
  Var A: 6,7,8,11,12,13,21,22,23
  Var B: 4,5,6,11,12,13,25,26,27
  Var C: 6,7,8,14,15,16,21,22,23
  Var D: 4,5,6,11,12,13,21,22,23

Question 4 [**5 points**]

Draw an AVL tree containing the values above **and label each node with its balance factor**. It can be any valid AVL tree that contains exactly the above values.

Answer

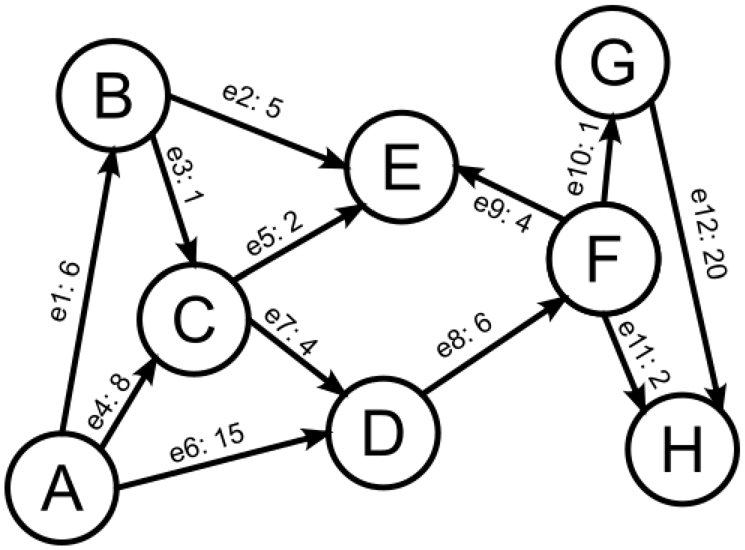
A valid BST that also meets AVL constraints. Since there are 9 nodes, there are not enough to reach a depth of 5. And there are too many to make a full tree of depth 3. So this tree must have a depth of 4 on one side a depth of 3 on the other.

Point Breakdown

- **(+1 pt)** If all nodes have a balance factor of +1 0 or -1
- **(+1 pt)** If all of the nodes have their balance factor correctly labeled (flipping sign is ok)
- **(+2 pt)** If the ordering is correct (for EVERY node smaller values are to the left, larger to the right)
- **(+1 pt)** If exactly the correct values are present
  Var A: 6,7,8,11,12,13,21,22,23
  Var B: 4,5,6,11,12,13,25,26,27
  Var C: 6,7,8,14,15,16,21,22,23
  Var D: 4,5,6,11,12,13,21,22,23

Part B: Graphs

For questions in this section use the following weighted, directed graph:



Note that each edge is labeled with both a name, and its weight.

Question 1 [**4 points**]

Write out the `Map` that would have been returned by the `computeOutgoingEdges` function from PA2 if it was run on the above graph. When writing the map you can write out each key-value pair as `<key>:<value>`. For example, `"A": "A really cool value"` would be the key-value pair mapping the string `"A"` to the string `"A really cool value"`.

Answer

A: [e1, e4, e6] or [B,C,D]
B: [e2, e3] or [C,E]
C: [e5, e7] or [E,D]
D: [e8] or [F]
F: [e9, e10, e11] or [E,G,H]
G: [e12] or [H]

Point Breakdown

- **(+4 pt)** If everything correct
- **(+2 pt)** If all of the information is correctly represented, but not in a map
- **(-1 pt)** For every edge (or vertex) that is in the wrong place (order doesn't matter)

Question 2 [**4 points**]

Which of the following traversal algorithms could possibly discover the path: `A -> C -> E`

Answer

A

- ☑ DFS
- ☑ BFS
- ☐ Djikstra's

B

- ☑ DFS
- ☐ BFS
- ☑ Djikstra's

C

- ☑ DFS
- ☑ BFS
- ☑ Djikstra's

D

- ☑ DFS
- ☐ BFS
- ☐ Djikstra's

Point Breakdown

- **(-2 pt)** For each checked box that shouldn't be
- **(-2 pt)** For each unchecked box tht should be
- **(0 pt)** NO credit for a blank answer

Question 3 [**4 points**]

Which of the following traversal algorithms could possibly discover the path: `A -> B -> C -> E`

Answer

A

- ☑ DFS
- ☐ BFS
- ☑ Djikstra's

B

- ☑ DFS
- ☑ BFS
- ☑ Djikstra's

C

- ☑ DFS
- ☐ BFS
- ☐ Djikstra's

D

- ☑ DFS
- ☑ BFS
- ☐ Djikstra's

Point Breakdown

- **(-2 pt)** For each checked box that shouldn't be
- **(-2 pt)** For each unchecked box tht should be
- **(0 pt)** NO credit for a blank answer

Question 4 [**4 points**]

Which of the following data structures could possibly be the TODO list in a traversal that discovered the path: `C -> D -> F -> G`

Answer

A
- ✓ Queue
- ✓ PriorityQueue
- ✓ Stack

B
- ☐ Queue
- ☐ PriorityQueue
- ✓ Stack

C
- ✓ Queue
- ☐ PriorityQueue
- ✓ Stack

D
- ☐ Queue
- ✓ PriorityQueue
- ✓ Stack

Point Breakdown

- **(-2 pt)** For each checked box that shouldn't be
- **(-2 pt)** For each unchecked box tht should be
- **(0 pt)** NO credit for a blank answer

Question 5 [**4 points**]

Which of the following data structures could possibly be the TODO list in a traversal that discovered the path: `A -> B -> C -> D -> F -> E`

Answer

A
- ☐ Queue
- ☐ PriorityQueue
- ✓ Stack

B
- ✓ Queue
- ☐ PriorityQueue
- ✓ Stack

C
- ☐ Queue
- ✓ PriorityQueue
- ✓ Stack

D
- ✓ Queue
- ✓ PriorityQueue
- ✓ Stack

Point Breakdown

- **(-2 pt)** For each checked box that shouldn't be
- **(-2 pt)** For each unchecked box tht should be
- **(0 pt)** NO credit for a blank answer

Part c: Priority Queue

In this part you will describe how to implement the `PriorityQueue` ADT using a Balanced BST as the data structure. For each function below, describe the algorithm you would use to implement it, and give the tight Big-O bound on the runtime **in terms of** n.

In your algorithms you may use any of the Balanced BST functions we described in class (`insert, find, remove`) to simplify your algorithm description. Any steps you take that do not use an existing function should be described in detail.

You may assume that elements of type `T` are comparable with the $<$ operator, and that the smaller values have the highest priority.

Question 1 [**10 points**]

Algorithm for `void PriorityQueue<T>.add(T elem)`:

Answer

Just call `insert(T)` on your balanced tree.
O(log n)

Point Breakdown

- **(+8 pt)** For a correct algorithm (either using insert, or describing the insert algorithm)
- **(+4 pt)** For an algorithm that is close, but either not quite correct, or doesn't run in log(n) time
- **(+2 pt)** For a runtime that matches THEIR algorithm (or for log n if they didn't give an algorithm)

Question 2 [**10 points**]

Algorithm for `T PriorityQueue<T>.peek()`:

Answer

The smallest item is all the way to the left so: Recursively (or iteratively) start at the root, and go to the left child until there is no left child. Return that value.
O(log n)

Point Breakdown

- **(+8 pt)** For a correct algorithm (move left until you can't anymore)
- **(+4 pt)** For an algorithm that is close, but either not quite correct, or doesn't run in log(n) time
- **(+2 pt)** For a runtime that matches THEIR algorithm (or for log n if they didn't give an algorithm)

Question 3 [**10 points**]

Algorithm for `T PriorityQueue<T>.remove()`:

Answer

Need to find and remove the smallest value.
To find, we will do the same as peek: start at the root and move left until we can't move left anymore. Let the value of that node be val.
Then we can simply call remove(val) on our balanced BST and return val.
O(log n)

Point Breakdown

- **(+8 pt)** For a correct algorithm (move left until you can't anymore, and then remove, or remove the node directly)
- **(+4 pt)** For an algorithm that is close, but either not quite correct, or doesn't run in log(n) time
- **(+2 pt)** For a runtime that matches THEIR algorithm (or for log n if they didn't give an algorithm)

Question 4 [**5 points**]

What would tight Big-O bound on the runtime of Djikstra's algorithm be if we use your implementation of `PriorityQueue` given above instead of using a `Heap`?

Answer

$O(|V| + |E| log|E|)$

Point Breakdown

- **(+5 pt)** For an answer of $O(|V| + |E|X)$ where X is the worst runtime they gave for their above 3 answers
- **(+3 pt)** For an answer of $O(|V| + |E|log|E|)$ if it doesn't match their above algorithm
- **(-1 pt)** If they forgot the $|V|$

Part d: Data Structure Design

In this question you want to store information about your friends to perform various tasks efficiently. For each of the below scenarios you may assume that you are storing instances of the following class:

```
public class Friend {
  public String name;
  public Date birthday;
  public String favoriteColor;
}
```

Question 1 [**10 points**]

Whenever you throw a party for a subset of your friends you like to try to make sure that everyone invited knows at least a few other people at the party. How could you store `Friend` objects so that you could quickly find out how many other invitees each friend knows?
In your answer make sure you:

1. Name the ADT you would use
2. Name the specific data structure you would use to implement that ADT
3. Explain in at most a few sentences how this choice would help you efficiently solve the problem

Answer

ADT: Graph
DS: Adjacency List
Each friend is a vertex, and an edge exists between two friends if they know each other. With the adjacnecy list data structure, we can determine the number of edges connected to a given vertex in O(1) time, and iterate through them in O(deg(v)) time.

Point Breakdown

- **(+4 pt)** For Graph (would also accept Map if they describe what they did in PA2)
- **(+4 pt)** For Adj list
- **(+2 pt)** For a resonable explanation (half credit is something is missing, ie they don't relate the graph to the problem asked)
- **(+5 pt)** For a solution that is would correctly work but not as well

Question 2 [**10 points**]

You like to keep up with your friends birthdays so you never forget to wish them Happy Birthday! You'd like to store `Friend` objects such that for any given date you can quickly determine which friend, if any, has a birthday on that date. Keep in mind, you are very popular so you will also be frequently adding new `Friend` objects to this data structure.
In your answer make sure you:

1. Name the specific data structure you would use to store your `Friend` objects
2. Describe any extra information you need to give Java to have your data structure work correctly
3. Explain in at most a few sentences how this choice would help you efficiently solve the problem

Answer

A balanced BST, ie AVL or Red Black.
We would need a comparator so that the tree knows to order them based on their birthday.
If all Friends are stored in the tree by their birthday, then it only takes log(n) time to find a node for a particaular birthday, and we can also add new friends to the tree in only log(n) time.

Point Breakdown

- **(+4 pt)** For a balanced BST (or for stating a HashMap if their explanation is good)
- **(+2 pt)** For mentioning the need for a way to sort the elements (or for the need of a hash function)
- **(+4 pt)** For a resonable explanation (half credit if something is missing, ie they don't relate the ds to the problem asked)
- **(+5 pt)** For a solution that is would correctly work but not as well

Question 3 [**5 points**]

As everyone knows, once you get old enough to be a professor, you stop making new friends. Considering this information, is there another choice of data structure that would perform just as well as the one you selected for the previous question if you don't have to worry about adding new `Friend` object?

Answer

If no friends need to be added, then a SortedArray list would also allow us to lookup friends by birthday in O(log n) time.

Point Breakdown

- **(+5 pt)** If they give a solution that has at least as good of a runtime as the solution they gave on the prev question

Part e: Bonus

Question 1 [**5 points**]

If you have a Red-Black tree of height 7, what is the minimum number of nodes you would need to check to find the largest value in the tree? Include the root and the node containing the largest value in your count. We are looking for a specific number, not an asymptotic number of nodes. Explain your answer.

Answer

Answer: 4
The largest value in the tree will be all the way to the right, and therefore have an EmptyTree node as it's right child.
Since the tree is of height 7, then we know the deepest EmptyTree node is at depth 8. That means the shallowest possible EmptyTree node is at depth 4. That means the shallowest the biggest value could be is at depth 3. So you must check the root, and three more nodes, for a total of 4.

Point Breakdown

- **(+1 pt)** For an answer of 4
- **(+4 pt)** For a reasonable explanation that understands the depth constraint (they can still get these 4 points if their answer was off by one, like 3 or 4)