Implement in Java the Min-priority Queue ADT defined above using

a) an array based binary heap similar to the one introduced in Lecture 8, [6]
b) a binary search tree. [6]

Observe that the ADT implementation operations should be in the form `q.insert(x)`, `q.min()`, etc. Explain in the report your implementation, noting the running time (using big-Oh notation) of each operation in both implementations.

Instruction on how to run the code:

Download each of MinPriorityQueueHeap.java, MinPriorityQueueBST.java, and TestQueue.java, which all belong in the same package, then run TestQueue.java to look at the implementations of each question that is printed out.

Q1(a):

In the worst case of the insert method in the array based binary heap implementation of the min priority queue ADT(abstract data type), the element that we insert has to be swapped at each recursion of moveDown() to swap the element at the end of the array with the element at the front of the array based heap in order to satisfy the min heap property. The swap method is also called in each loop of the while loop in moveUp() method that is called at each insert() call. As the total number of swaps in moveUp and move down methods is equal to log n, the worst case running time of insert and extractMin operations will be O(log n), while on the other hand the min() method will run at O(1) as it simply returns the element at the front of the array.

Q1(b):

In a normal binary search tree(BST) implementation of the min priority queue ADT, the running time of each of the operations - insert, min and extractMin - are O(h) where h is the height of the BST. The worst case running time of each of those operations will be when the BST is unbalanced, such as a tree with the min node (node with smallest key value) at the root and all the other nodes each at the right subtree of their parent node, and since unbalanced BST has a height of O(n), the worst case running time of the three operations will therefore be O(n).

c) What are the worst-case running times of the three ADT operations when the underlying BST is self-balancing? Briefly explain your answer. [3]

Q1(c):

When the BST is self-balancing, the tree will maintain the height, h, as small as possible, typically at log n, by a rotation - either left or right rotation, after each of the operations - insert, min and extractMin - has been called. As the worst case running time of each of those operations is O(h) where h is the height of the BST, the worst case running time will be O(log n) for each of the operations, such as in an AVL tree. There is also another extension of the BST,

as well as AVL tree, that makes the tree self-balancing as previously mentioned, which is the RB tree. Both min and extractMin operations in a RB tree run in O(log n) in the worst case, however the insert operation runs in O(1) in the worst case.

d) Implement and extension of BST that allows MIN and EXTRACT-MIN operations in O(1). Briefly describe your implementation in the report. Hint: maintain an extra pointer attribute in each node. [6]

Q1(d):

The extra pointer attribute, min, has been added as a field to the Node class of the MinPriorityQueueBST.java to denote the node with the minimum key value. Every time the insert() method is called, if either the min attribute of the root node is null, i.e. not yet set, or the min attribute of the root node points to a node that has a key is larger than the key attribute of the node that we are trying to insert, then we set the min attribute of root node to point to the node that we insert, as in this case the node that we insert will have the smallest key value.

As the min pointer is set each time we insert which takes O(1), instead of min() and extractMin() method we can call minConstant() and extractMinConstant() methods which simply gets the min pointer of the root node to return the node with minimum key, which takes O(1); instead of using a while loop to traverse down the BST to find and return the leftmost node in the tree, which would take O(n) running time.

Part 2

Implement an efficient algorithm in Java to solve the following problem:

> *You are given n ropes of different lengths (expressed as integers), and you are asked to connect them to form a single rope with the minimum cost. The cost of connecting two ropes is equal to the sum of their lengths.*

Given a sequence of rope lengths, the expected outputs are a sequence of rope connection operations and the total cost. Use one of your implementations of the Min-priority Queue ADT in your solution.

a) Give a brief description of your implementation, explaining why a priority queue is needed for an efficient algorithm. [8]
b) What is the output for this instance of the problem 4,8,3,1,6,9,12,7,2? [1]

Q2(a):

The lengths of the ropes which are picked first are included more than once in total cost. Therefore, to have the minimum total cost, we need to pick the smallest two ropes first and repeat this for the remaining ropes. For example, under the comment //2(a) in TestQueue.java, the values 4,3,2 and 6 are inserted into the heap by calling the buildMinHeap method, which implements the insert() method. The values 4,3,2 and 6 each represent the length of different

ropes. Then when we call getRopeMinCost method on this heap, it first picks the 2 ropes of smallest and second smallest rope length respectively, which are 2 and 3. It then adds these ropes together to create a new rope of length 5, which is printed out. Then this rope length value, 5, is added to the minimum total cost and is also inserted into the heap to print out the new heap which would now have ropes of length 4, 6 and 5 (as printed out). This is repeated until there is only one value left in the heap which is the length of the rope created last that is last added to the minimum total cost. The minimum total cost is then printed out afterwards which in this case is 29.

Since we pick two ropes of the first and second smallest rope length at each loop of the while loop until the heap(queue) contains only one rope length that is the aforementioned length of rope that is created last, we need to implement a min-priority queue for efficiency because after we call the buildMinHeap method on our heap to create a min-priority queue, we then know that right after the buildMinHeap method is called and also after we call insert or extractMin method the smallest rope length will always be at the start of the array-based binary heap that implements the min-priority queue, therefore it is more efficient as we always know where the smallest rope length is and thus we don't need to use a for loop to traverse through each value in the array to find the smallest rope length which will add another O(n) to the time-complexity.

Q2(b):

The output is min total cost = 153.