Samuel Jackson – 2520998J

ADS2 - Assessed Exercise 2

How to run the code:
All the necessary code is in the following java files: ArrayQueue.java, BSTQueue.java, BSTConstantQueue.java, TestingDS.java.

All the code can be run in TestingDS.java.
Create an object of the respective type, all classes contain insert(int key), min() and extract_min().
ArrayQueue (array-based heap) also contains insert(int[] keysArray) if desired.

ArrayQueue has print method printArray().

BSTQueue and BSTConstantQueue have printTree() methods.

Important to note: Unless stated otherwise log n refers to $\log_2$ n, not $\log_{10}$ n.

Part 1 :

a) A min-heap was used for the implementation of the array-based queue.

A min-heap is a data structure where the parent node must be smaller/equal to the child nodes. This is a more appropriate heap data structure than max-heap for a min-priority queue because it allows us to find the minimum in constant time (by returning the head of the heap).

There are four primary operations in the min-heap data structure:

- min() : returning the minimum key in the data structure.
- extract_min() : removing and returning the minimum key in the data structure.
- insert(int x) : adding an element of integer x to the data structure, return void.
- heapify(int index) : method to ensure that the data structure adheres to the heap properties, return void.

min() is returning the head of the heap, leading to a constant time, **O(1)**, operation because it is an array access, which only has to occur one time.

heapify(int index) is a function which assumes that the data structure is satisfying the heap property, but the root may be breaking it. Therefore, in the worst-case, it must traverse down the height of the tree in case the element at the top is the largest element in the array (worst-case for min-heap). The height of the array is log n, meaning **heapify is O(log n)**.

extract_min() is an operation of O(log n). It is a result of finding min, O(1), and deleting the element is O(1) but upon deletion from the head, you must ensure that the heap property remains, hence requiring heapify, a O(log n) operation. O(1) + O(1) + O(log n) = O(log n).

insert(int x) is an operation which adds an element to the end of an array (left-most part of the heap-tree). Insert traverses up the tree, confirming that the newly inserted element is in the right position (smaller than parent but larger than child). In the worst- case, new element is the smallest in the tree, then insert will traverse the height of the tree, log n. Therefore, **insert is O(log n)**. Insert also contains heapify, O(log n), to ensure heap property is maintained. Hence, O(log n) + O(log n) = O(log n).

b) A binary search tree was used for the implementation for the min-priority queue.

The binary tree class has a nested class, Node, and three primary operations. The nested class has the following fields:

- Node left (left child).
- Node right (right child).
- int key.

The three primary operations are:

- insert(int key) : Insert a node with this.key = key into the tree, return void.
- min() : return the value of the smallest node in the tree.
- extract_min() : return the value of the smallest node, and remove the node.

A min-priority queue was simple because a binary search tree has properties which can be abused in order to find the minimum first. A binary search tree must satisfy the following properties:

- The left child of a node must be smaller (left subtree must be smaller than node).
- The right child of a node must be greater (right subtree must be greater than node).

Abusing these properties, we can create the priority queue.

insert(int key) is an operation of O(n). Insertion has to make sure that it is placing itself in the tree in a place which ensures that binary search tree property of left/right subtrees being less/greater than parent respectively. In the worst case, on a right-skewed tree, where consequently inserted element is greater than the next, this is traversing all the elements, leading to a O(n) runtime. It is symmetric for a left-skewed tree.

min() is an abuse of the binary search tree property where you look for the left-most node in the tree and return its key. In the case of a left-skewed tree, **this is a running time of O(n)**.

extract_min() is a running time of O(n) in the case of a left-skewed tree. Deletion is simple and a O(1) operation but finding the element takes some time.

c) In the case where the BST is self-balancing (assuming AVL tree), we ensure that the max height of the tree is log n. This means, on insert, we only have to traverse log n elements, **leading to O(log n)**. It is similar for min as it only has to traverse log n elements of the left-subtree, **leading to O(log n)** too. Consequently, from min, **extract_min is O(log n)**.

d) My implementation of this extension sacrifices the time complexity of insert, in order to ensure the constant time complexity of extract_min.

In order to make min() and extract_min() constant time then I had to know the smallest node in the tree beforehand. My approach was to create an array of Node objects and add elements upon insert() and include a tail which contained the index of the last element in the array.

To make these changes, I added an attribute to the Node class “p” which was a pointer to the parent of the node.

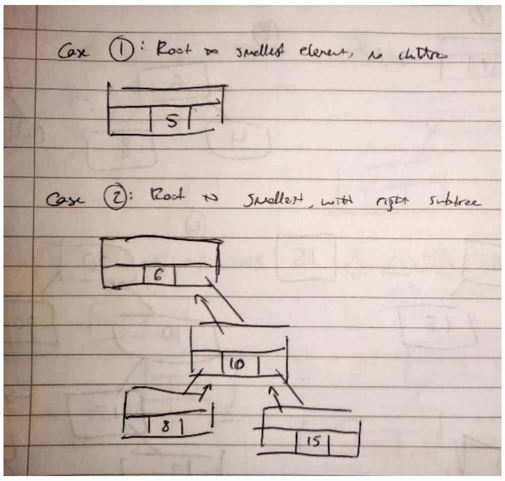
Insert would assign the parent node during the process, ensuring that all nodes had a parent (except root, pointing to null).

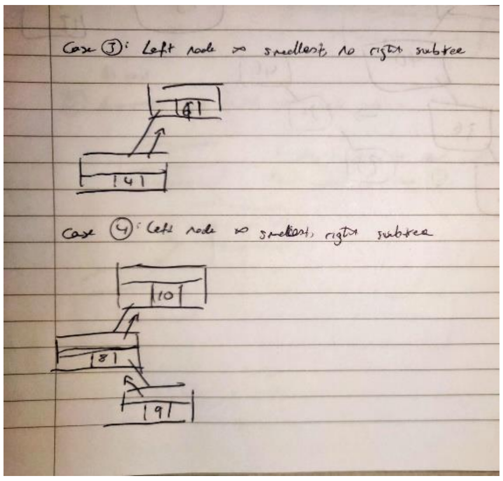
Insert would also add Node elements into the new Node array, whilst checking if resizing is necessary. The array was then reverse insertion sorted with the smallest element at the end of the array.

Min() would peek at the tail of the array (looking at the smallest node), therefore it is only an array access hence O(1).

An important problem was making sure that I was able to know the next smallest node when the smallest is removed. The solution that I implemented with an array was useful for this because I just moved the tail pointer down by one index and insertion sort guarantees the node at the new tail index is the smallest in the tree.

The final portion of this problem was ensuring that it was removed from the tree correctly. In order to deal with this, I utilised the parent attribute. There was 4 distinct cases to deal with:


Case ①: Root is smallest element, no children
5
Case ②: Root is smallest, with right subtree
6
10
8
15

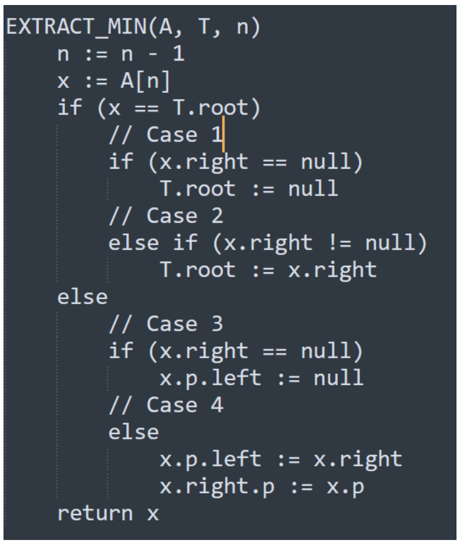

Case ③: Left node is smallest, no right subtree
8
4
Case ④: Left node is smallest, right subtree
10
8
9

The key in removing the elements from the tree was to make sure that no node was pointing to the element. To deal with each case, independently, was relatively simple.

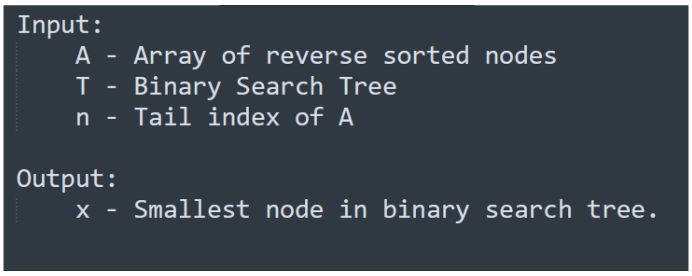
(Let x be the smallest node).

- Case 1: Set root to null.
- Case 2: Set root to the child of the x, and set new root's parent as null.
- Case 3: Set x's parent's left-child pointer to null. (Left child pointer of parent node as null).
- Case 4 (hardest case): Set the left-child pointer of x's parent to point towards the right child of x. Set the parent pointer of the right child of x to x's parent.

Compiling these cases, we obtain the pseudocode:



```
EXTRACT_MIN(A, T, n)
    n := n - 1
    x := A[n]
    if (x == T.root)
        // Case 1
        if (x.right == null)
            T.root := null
        // Case 2
        else if (x.right != null)
            T.root := x.right
    else
        // Case 3
        if (x.right == null)
            x.p.left := null
        // Case 4
        else
            x.p.left := x.right
            x.right.p := x.p
    return x
```



```
Input:
    A - Array of reverse sorted nodes
    T - Binary Search Tree
    n - Tail index of A

Output:
    x - Smallest node in binary search tree.
```

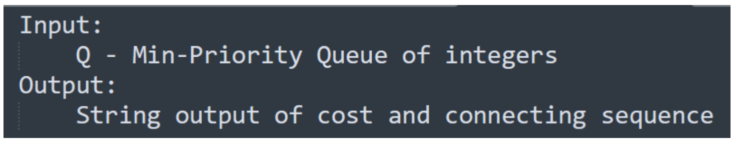
Since min() is only an array access, **it now has time complexity of O(1)**. Consequently, since min() is constant time, and there is only extract_min contains binary operations of O(1) then **extract_min also time complexity of O(1)**.

This comes at a sacrifice of insert() as insert now has to resize an array, O(n) so does not change the time complexity, but it also insertion sorts the array which is O(n^2) on the worst and average case. However, note that the worst-case would be if all the elements are sorted in ascending order. Worst-case cannot happen since elements are added incrementally and sorted as we go, avoiding the worst-case but maintaining the average case of O(n^2). This means that insert now has O(n^2),

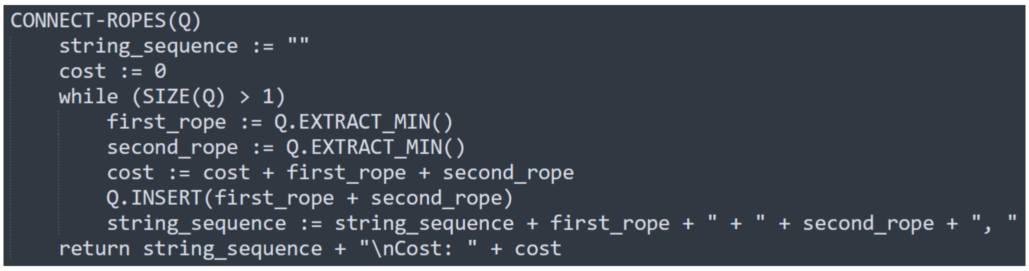
Part 2:

a) My solution to the problem is a simple algorithm. Extract the two minimum, join them, reinsert into queue.

Here is the pseudocode for the algorithm:



```
Input:
    Q - Min-Priority Queue of integers
Output:
    String output of cost and connecting sequence
```



```
CONNECT-ROPES(Q)
    string_sequence := ""
    cost := 0
    while (SIZE(Q) > 1)
        first_rope := Q.EXTRACT_MIN()
        second_rope := Q.EXTRACT_MIN()
        cost := cost + first_rope + second_rope
        Q.INSERT(first_rope + second_rope)
        string_sequence := string_sequence + first_rope + " + " + second_rope + ", "
    return string_sequence + "\nCost: " + cost
```

I chose to use an array-based heap implementation for the min-priority queue because the binary search tree would not allow for this algorithm to work whilst maintaining the properties of a binary search tree (duplicates would arise when reinserting and this would break the BST).

This algorithm is efficient, but it is important to recognise that the only interaction with the collection of integers is retrieving the minimum values. This is a critical point as to why the min-priority queue is more efficient.

Connect-ropes contains a while loop which will loop until there is only one item left in the collection, which is going to be $n - 1$ times. In each iteration of the while loop, the minimum must be obtained twice from the collection of elements. The difference shines here:

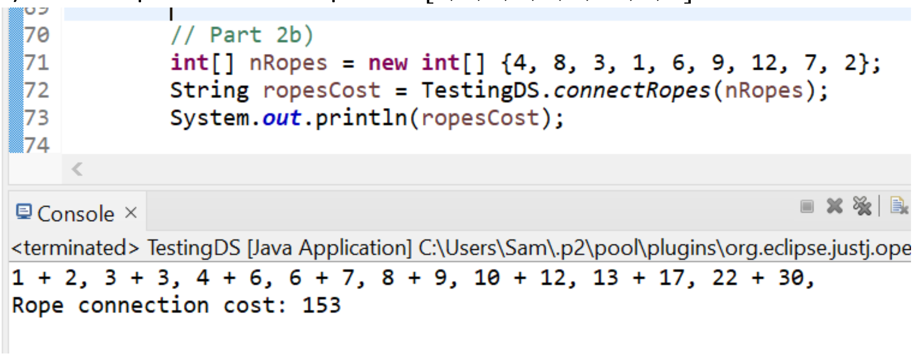
- Ordinary primitive array
    - Extracting minimum would have a time complexity of O(n), having to check each element.
    - Insert operation time:
        - Fixed size array: O(1).
        - Resizable array: Amortised O(1)
- Min-priority queue (heap-based)
    - Extracting minimum would have a time complexity of O(log n), having to traverse height of the heap.
    - Insert operation time: O(log n)

This means that connect-ropes with an ordinary primitive array would have a time complexity of O(n^2) whilst the min-priority queue would have a time complexity of O(n log n).

This is a fundamental difference which shows that a min-priority queue is more efficient. It is also significant to notice that there is also the insertion operation. There are constant time operations for extracting min but they tend to sacrifice the insert time complexity or have problems dealing with duplicates (Binary Search Trees).

For this reason, min-priority queue is the most efficient for this algorithm because it is focused on extracting the minimum from a collection of elements which is what a min-priority queue is designed to do in an efficient way, alongside dealing with insertion.

b) The output from the sequence: $[4, 8, 3, 1, 6, 9, 12, 7, 2]$ is:



```java
        // Part 2b)
        int[] nRopes = new int[] {4, 8, 3, 1, 6, 9, 12, 7, 2};
        String ropesCost = TestingDS.connectRopes(nRopes);
        System.out.println(ropesCost);
```

Console

```
<terminated> TestingDS [Java Application] C:\Users\Sam\.p2\pool\plugins\org.eclipse.justj.ope
1 + 2, 3 + 3, 4 + 6, 6 + 7, 8 + 9, 10 + 12, 13 + 17, 22 + 30,
Rope connection cost: 153
```