Samuel Jackson – 2520998J

ADS2 - Assessed Exercise 2

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

Important to note: Unless stated otherwise log n refers to log\_2 n, not log\_10 n.

Part 1 :

1. 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).

1. 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.
* int size (not used for a binary search tree).

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.

1. 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)**.
2. **Do some drawings here.**

Part 2:

1. I chose to use the array-based heap implementation for my solution.

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

Here is the pseudocode for the algorithm: