Aashray Shrestha

CS 3310 - Data and File Structures
Instructor : Dr.Ajay Gupta, Western Michigan University
TA : Ms.Rajani Pingili

I do not give permission to share my answer with the class.

EXTRA CREDIT : SOFTWARE LIFE CYCLE REPORT - HW 4

PHASE 1 : SPECIFICATION

Suppose we are interested in dynamically maintaining a set S of integers, which is initially empty, while supporting the following three main operations:

1. ***add(v)***: Adds value v to set S.

2. ***getMedian()****:* Returns the current median value of the set. For a set with even cardinality, we define the median as the average of the two most central values. For a set with odd cardinality, median is the middle element when considering data in a sorted manner. (In other words, a median is the element in the data set which separates the higher half of the data sample from the lower half; OR half the elements are smaller and half the elements are larger than the median).

3. ***deleteMedian():*** deletes upper median of the set for even cardinality, otherwise deletes median if it's an odd cardinality.

   We will store each element of the set in one of the two priority queues:
   - **A min-oriented priority queue**, Q+, of all elements greater than or equal to the current median value.
   - **A max-oriented priority queue**, Q−, of all elements less than the current median value.

   Additional Specifications have been implemented in the program such as :
   - **Calculating average time for the different operations** *add(v)*, *getMedian()*, and *deleteMedian()*
   - **Implement *sortUsingMedians()*** which returns a sorted array of length n, where n is the current cardinality of the set S
   - **Use of *mergeSort()*** to return a sorted array of length n and *sortUsingMedians()* as mentioned above

Aashray Shrestha

PHASE 2: DESIGN

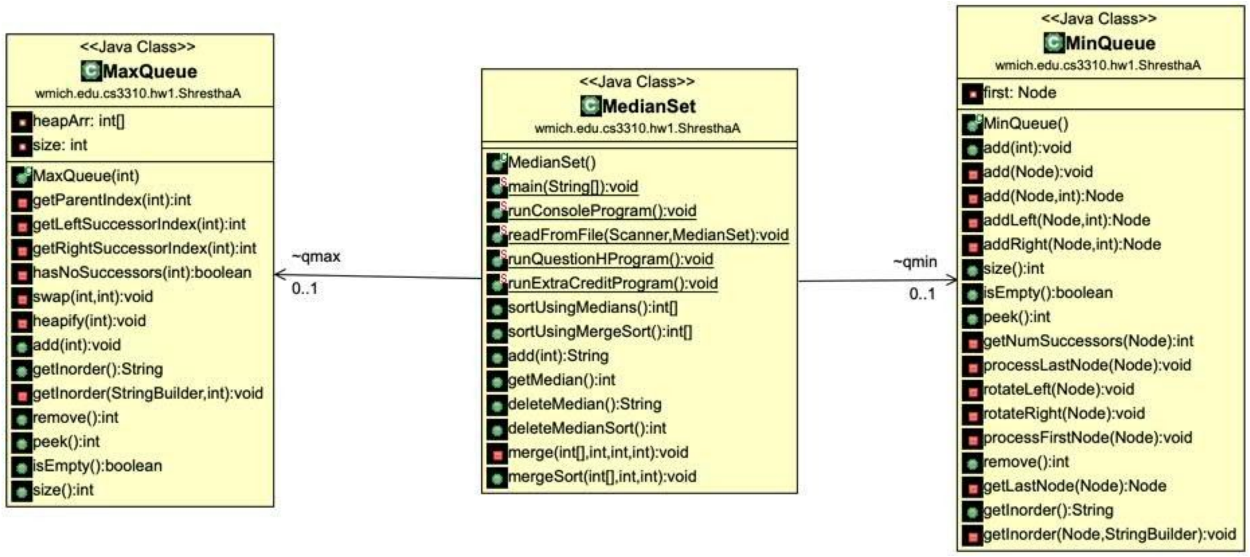


Fig : UML Diagram Built on Eclipse

1. MedianSet Class

MedianSet class acts as the main class and the driver for the program.

The MedianSet class instantiates the MaxQueue and MinQueue Classes to run the program.

```
MinQueue qmin = new MinQueue();
MaxQueue qmax = new MaxQueue ();
```

The MedianSet class consists of the following methods :

a. **public static void** main(String[] args) **throws** IOException {

  - function call the runConsoleProgram function

  }

b. **public static void** runConsoleProgram() **throws** IOException {

  - Create an instance of the MedianSet class to use the methods in a more efficient way
  - Show the list of options for the user to look through and run the program

    1 : Run Manually

      This option has further options to look into

        R - Read from File

        A - Add

        C - Check Median

        D - Delete Median

        E - Exit Program

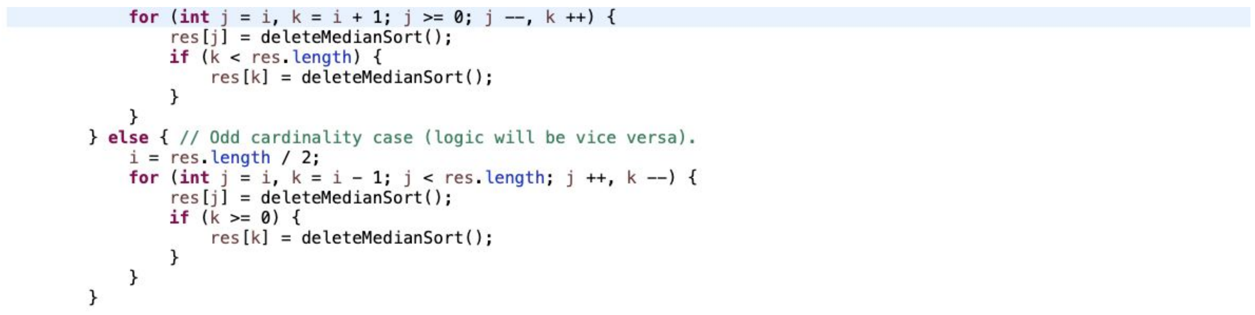
    2 : Extra Credit Problem for the Average Time using different Sorting

    3 : Average Time taken for the 3 operations : add, deleteMedian, getMedian

```
                    }
c. public static void readFromFile(Scanner sc, MedianSet ms) throws IOException{
           ask user for name of the file
if exists () {     Read the entire file and have the operations
                       loop (endOfFile){
                              Check the instruction in the file separated by commas
                                 if “g”
                                        get the median value, or display an error if not present
                                                  display the time taken for the instruction to process
                                if “d”
                                        delete the median value, or display error if not present
                                                  display the time taken for the instruction to process
                                if “a”
                                        add the value to the given data set,  and print the
                                                  Inorder traversal of Q-
                                                  Inorder traversal of Q+
                                                  display the time taken for the instruction to process
                              } - end if
                    } - end loop
                    } else {
                              display Error -  “File Not Found”
                    }
}
d. public static void runQuestionHProgram() throws IOException {
          // The method is for measuring the average time for the 3 methods
                    Use of ArrayList to store the time for each of the operation
                              if “g”
                                        get the median value, or display an error if not present
                                        store the time to the predefined .add method of the ArrayList
                              if “d”
                                        delete the median value, or display error if not present
                                        store the time to the predefined .add method of the ArrayList
                              if “a”
                                        add the value to the given data set,  and print the
                                                  Inorder traversal of Q-
                                                  Inorder traversal of Q+
                                        store the time to the predefined .add method of the ArrayList
                                        Calculate the Average Times and Print it out!
                    } end if
```

```
    }

e. public static void runExtraCreditProgram() throws IOException {
//Compare the sortUsingMedians and sortUsingMergeSort to calculate time complexity
        Read the hw4input.txt file
        Create an items array as : String items [] = input.split(“,”);
                for ( String item : items) // items =  array of the input from the txt file
                        if “a”                  // Only add because the task is to sort arrays
                        add it to the items array
                        } end if
                        Call SortUsingMedian function and calculate time

                for ( String item : items) // items =  array of the input from the txt file
                        if “a”                  // Only add because the task is to sort arrays
                        add it to the items array
                        } end if
                        Call sortUsingMergeSort function and calculate time
                }
f. public int[] sortUsingMedians()
        The method that sorts values in two queues and returns them as an array.
                int res [] = new int [qmax.size() + qmin.size()]
                                                    //Create an empty array with defined length
                         i      // declare a variable to be the median index
                if (res.length () % 2 == 0 ) // Even length
                        i = res.length / 2  + 1 ; // index for median
```

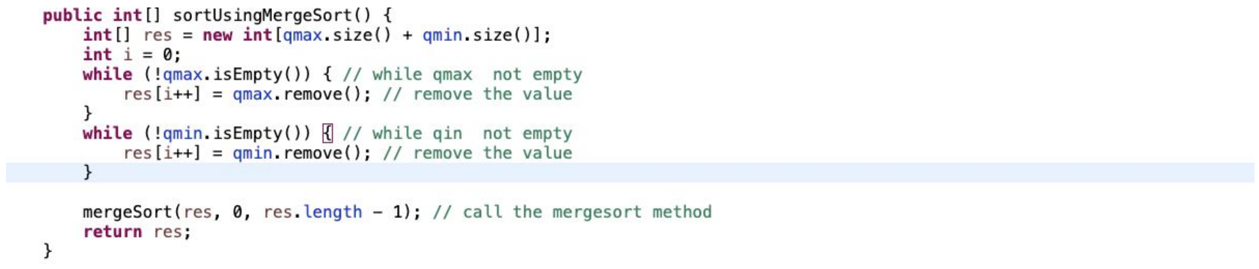
Here we need to loop through the two indexes (j and k) to define places where the results of deleteMedian operations will be put. In **case** of even cardinality, index j should go down,whereas index k should go up. Every time we perform deleteMedian operation, we get two already sorted values (two queues, two halves or array) in ascending order, but every first value will be going down, and every second value will be going up.



```
            for (int j = i, k = i + 1; j >= 0; j --, k ++) {
                res[j] = deleteMedianSort();
                if (k < res.length) {
                    res[k] = deleteMedianSort();
                }
            }
        } else { // Odd cardinality case (logic will be vice versa).
            i = res.length / 2;
            for (int j = i, k = i - 1; j < res.length; j ++, k --) {
                res[j] = deleteMedianSort();
                if (k >= 0) {
                    res[k] = deleteMedianSort();
                }
            }
        }
```

g. **public int**[] sortUsingMergeSort()

Method to perform sort the array using merge sort.First, it needs to fill out the array with values from the queues, second sort this array using merge sort and return sorted array.

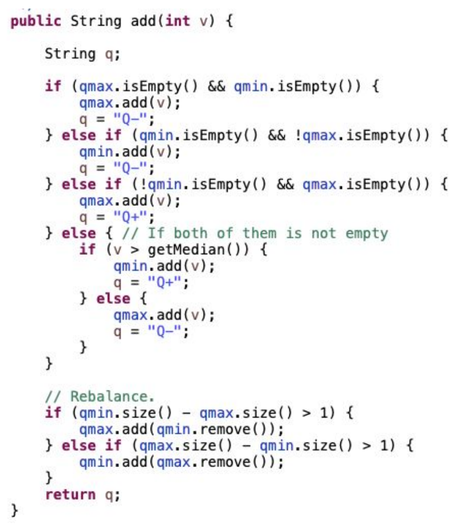


```
    public int[] sortUsingMergeSort() {
        int[] res = new int[qmax.size() + qmin.size()];
        int i = 0;
        while (!qmax.isEmpty()) { // while qmax  not empty
            res[i++] = qmax.remove(); // remove the value
        }
        while (!qmin.isEmpty()) { // while qin  not empty
            res[i++] = qmin.remove(); // remove the value
        }

        mergeSort(res, 0, res.length - 1); // call the mergesort method
        return res;
    }
```

}

h. **public** String add(**int** v) {

Method to add the value to the set.



```
    public String add(int v) {

        String q;

        if (qmax.isEmpty() && qmin.isEmpty()) {
            qmax.add(v);
            q = "Q-";
        } else if (qmin.isEmpty() && !qmax.isEmpty()) {
            qmin.add(v);
            q = "Q-";
        } else if (!qmin.isEmpty() && qmax.isEmpty()) {
            qmax.add(v);
            q = "Q+";
        } else { // If both of them is not empty
            if (v > getMedian()) {
                qmin.add(v);
                q = "Q+";
            } else {
                qmax.add(v);
                q = "Q-";
            }
        }

        // Rebalance.
        if (qmin.size() - qmax.size() > 1) {
            qmax.add(qmin.remove());
        } else if (qmax.size() - qmin.size() > 1) {
            qmin.add(qmax.remove());
        }
        return q;
    }
```

i. **public int** getMedian() {

Required method to return the median from the set.

```
        if (qmax.isEmpty() && qmin.isEmpty()) {
            return -1; // If no values present yet.
        }

        if (qmax.size() == qmin.size()) {
            return (qmax.peek() + qmin.peek()) / 2;
        } else if (qmax.size() > qmin.size()) {
            return qmax.peek();
        } else {
            return qmin.peek();
        }
```

}

j. **public** String deleteMedian() {

Required method to delete the median.

```
        if (qmax.isEmpty() && qmin.isEmpty()) {
            return "no value present";
        }
        if (qmax.isEmpty() && !qmin.isEmpty()) {
            return "median " + qmin.remove() + " is deleted from Q+";
        }
        if (!qmax.isEmpty() && qmin.isEmpty()) {
            return "lower median " + qmax.remove() + " is deleted from Q-";
        }
        String q;

        if ((qmax.size() + qmin.size()) % 2 == 0) { // If even cardinality.
            int m = qmin.remove(); // Delete upper median.
            q = "upper median " + m + " is deleted from Q+";
        } else { // Otherwise, delete median.
            int m = qmax.size() > qmin.size() ? qmax.remove() : qmin.remove();
            q = "median " + m + " is deleted from Q-";
        }
        return q;
```

}

k. **public int** deleteMedianSort(){

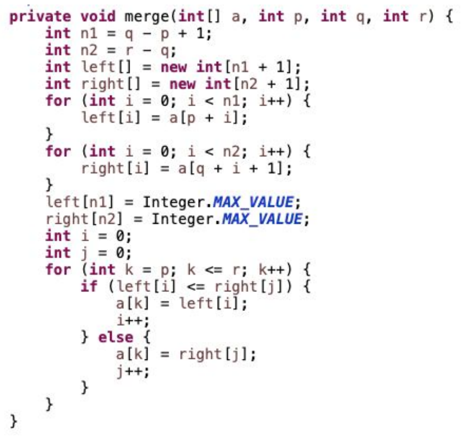
The same function as deleteMedian, but returns only integer value (previously deleted).
Required for sortUsingMedians() operation.

```
        if (qmax.isEmpty() && qmin.isEmpty()) {
            return -1;
        }
        if (qmax.isEmpty() && !qmin.isEmpty()) {
            return qmin.remove();
        }
        if (!qmax.isEmpty() && qmin.isEmpty()) {
            return qmax.remove();
        }

        if ((qmax.size() + qmin.size()) % 2 == 0) { // If even cardinality.
            int m = qmax.remove();
            return m;
        } else {
            int m = qmin.remove();
            return m;
        }
```

}

l. **private void** merge(**int**[] a, **int** p, **int** q, **int** r) {

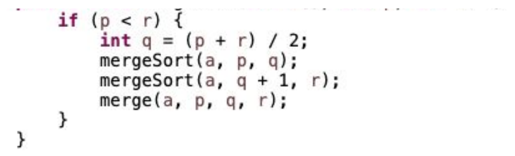
Helper operation to merge two sorted arrays.



```
private void merge(int[] a, int p, int q, int r) {
    int n1 = q - p + 1;
    int n2 = r - q;
    int left[] = new int[n1 + 1];
    int right[] = new int[n2 + 1];
    for (int i = 0; i < n1; i++) {
        left[i] = a[p + i];
    }
    for (int i = 0; i < n2; i++) {
        right[i] = a[q + i + 1];
    }
    left[n1] = Integer.MAX_VALUE;
    right[n2] = Integer.MAX_VALUE;
    int i = 0;
    int j = 0;
    for (int k = p; k <= r; k++) {
        if (left[i] <= right[j]) {
            a[k] = left[i];
            i++;
        } else {
            a[k] = right[j];
            j++;
        }
    }
}
```

m. **public void** mergeSort(**int** a[], **int** p, **int** r){

Performs a merge sort of array a.



```
    if (p < r) {
        int q = (p + r) / 2;
        mergeSort(a, p, q);
        mergeSort(a, q + 1, r);
        merge(a, p, q, r);
    }
}
```

}

2. MaxQueue Class

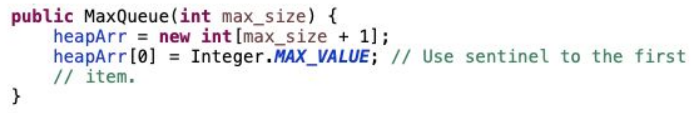
A max-oriented priority queue, Q−, of all elements less than the current median value.

Implementation Q-using array-based heaps.

private int[] heapArr; // Array to store heap

private int size; // Size of heap

Constructor : to instantiate MaxQueue and initialize it with the maximum size.

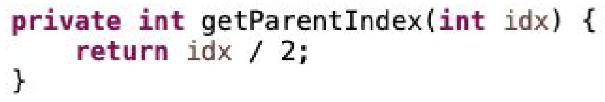


```
public MaxQueue(int max_size) {
    heapArr = new int[max_size + 1];
    heapArr[0] = Integer.MAX_VALUE; // Use sentinel to the first
    // item.
}
```

The **MaxQueue** class consists of the following methods :

a. private int getParentIndex(int idx) {

// Method to return parent index

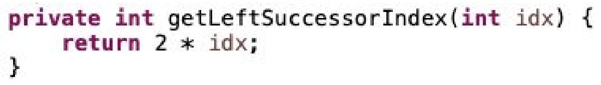


```
private int getParentIndex(int idx) {
    return idx / 2;
}
```

}

b. private int getLeftSuccessorIndex(int idx){

// Method to return index of left successor (child)

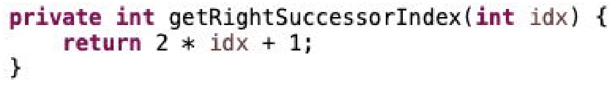


```
private int getLeftSuccessorIndex(int idx) {
    return 2 * idx;
}
```

}

c. private int getRightSuccessorIndex(int idx){

// Method to return index of right successor (child)

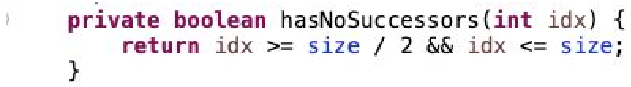


```
private int getRightSuccessorIndex(int idx) {
    return 2 * idx + 1;
}
```

}

d. private boolean hasNoSuccessors(int idx){

//Method that returns true if item under the given index has no successors

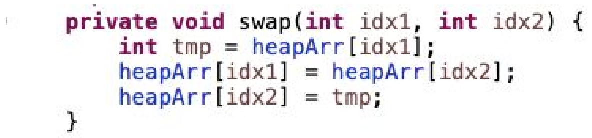


```
private boolean hasNoSuccessors(int idx) {
    return idx >= size / 2 && idx <= size;
}
```

}

e. private void swap(int idx1, int idx2) {

// Method to swap elements in an array under indexes idx1 and idx2. As a result, item under idx1 will be under idx2, and item under idx2 will be under idx1.

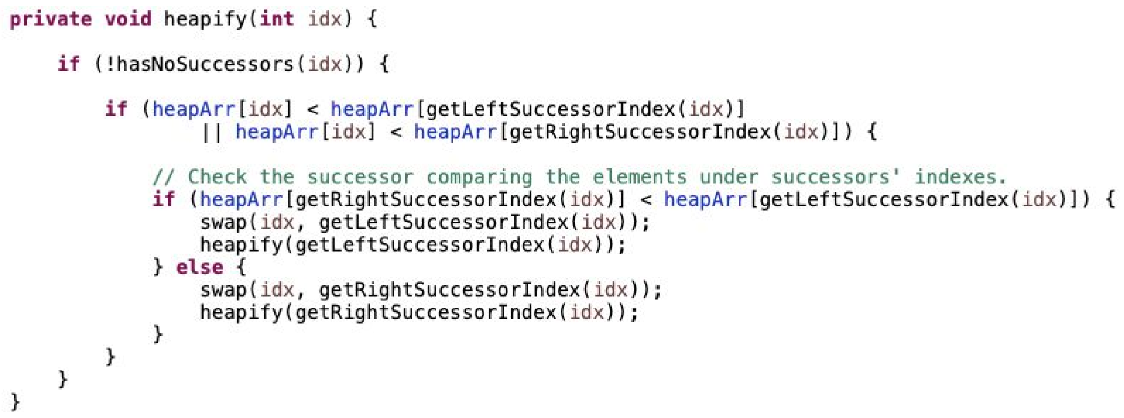


```
private void swap(int idx1, int idx2) {
    int tmp = heapArr[idx1];
    heapArr[idx1] = heapArr[idx2];
    heapArr[idx2] = tmp;
}
```

}

f. private void heapify(int idx) {

//Method to build the heap starting from the element under given idx. To perform this operation, the element under idx must have at least one successor (or child) since heap is built starting from this element as the root.

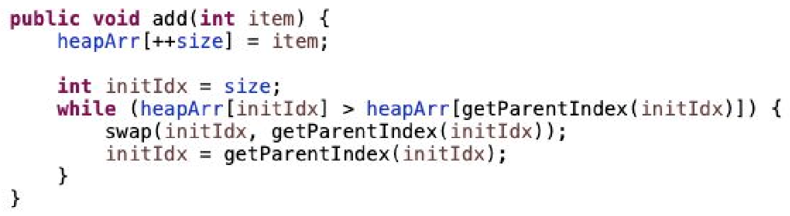


```
private void heapify(int idx) {

    if (!hasNoSuccessors(idx)) {

        if (heapArr[idx] < heapArr[getLeftSuccessorIndex(idx)]
                || heapArr[idx] < heapArr[getRightSuccessorIndex(idx)]) {

            // Check the successor comparing the elements under successors' indexes.
            if (heapArr[getRightSuccessorIndex(idx)] < heapArr[getLeftSuccessorIndex(idx)]) {
                swap(idx, getLeftSuccessorIndex(idx));
                heapify(getLeftSuccessorIndex(idx));
            } else {
                swap(idx, getRightSuccessorIndex(idx));
                heapify(getRightSuccessorIndex(idx));
            }
        }
    }
}
```

}

g. public void add(int item) {

// Method to add element to the heap

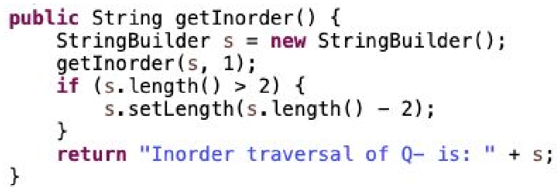


```
public void add(int item) {
    heapArr[++size] = item;

    int initIdx = size;
    while (heapArr[initIdx] > heapArr[getParentIndex(initIdx)]) {
        swap(initIdx, getParentIndex(initIdx));
        initIdx = getParentIndex(initIdx);
    }
}
```

}

h. public String getInorder() {

// Returns a string represented inorder heap traversal. Recursively calls itself.

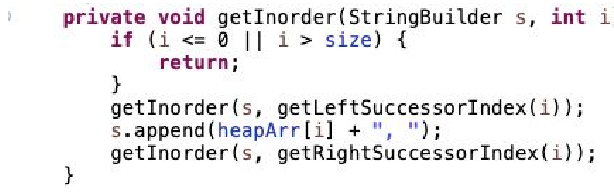


```
public String getInorder() {
    StringBuilder s = new StringBuilder();
    getInorder(s, 1);
    if (s.length() > 2) {
        s.setLength(s.length() - 2);
    }
    return "Inorder traversal of Q- is: " + s;
}
```

}

i. private void getInorder(StringBuilder s, int i){

//Method to store inorder traversal using recursion. Inorder traversal assumes to visit left child, then visit node, then visit right child.

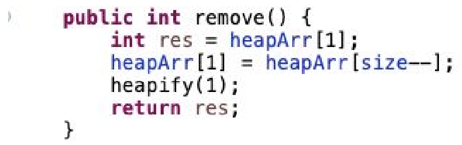


```
private void getInorder(StringBuilder s, int i) {
    if (i <= 0 || i > size) {
        return;
    }
    getInorder(s, getLeftSuccessorIndex(i));
    s.append(heapArr[i] + ", ");
    getInorder(s, getRightSuccessorIndex(i));
}
```

}

j. public int remove() {

// Removes top element (maximum element) from the heap because it is maxheap. Maximum element is always stored as the first element



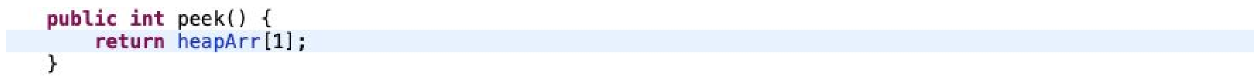
```
public int remove() {
    int res = heapArr[1];
    heapArr[1] = heapArr[size--];
    heapify(1);
    return res;
}
```

}

k. public int peek(){

Aashray Shrestha

// Returns maximum element of the heap

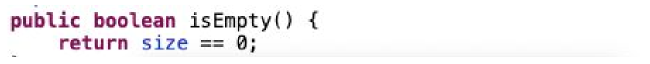


```java
public int peek() {
    return heapArr[1];
}
```

}

l. public boolean isEmpty(){

// Returns true if this heap is empty, otherwise false

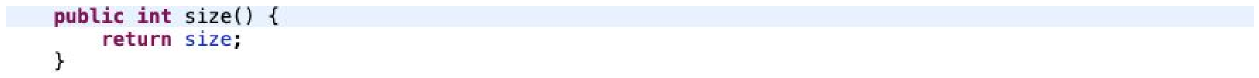


```java
public boolean isEmpty() {
    return size == 0;
```

}

m. public int size() {

// Returns the number of elements in the heap



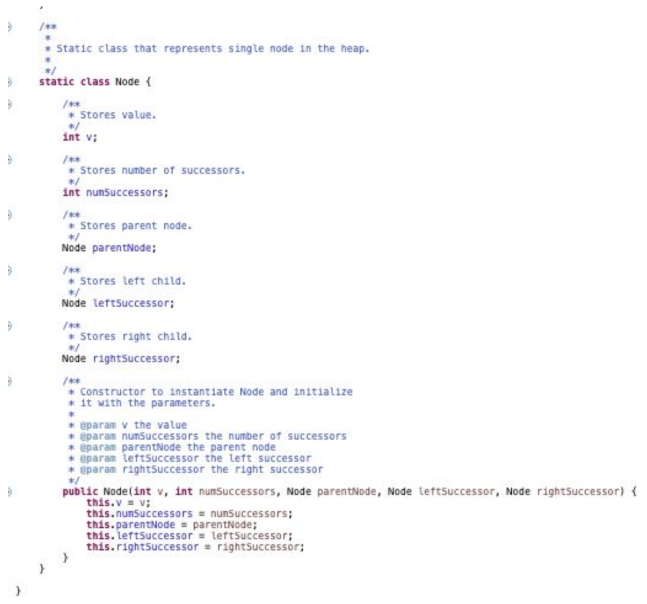
```java
public int size() {
    return size;
}
```

}

3. MinQueue Class

A min-oriented priority queue, Q+, of all elements greater than or equal to the current median value. Implementation Q+ using size-balanced Linked-List based heaps.

It has the following Node class



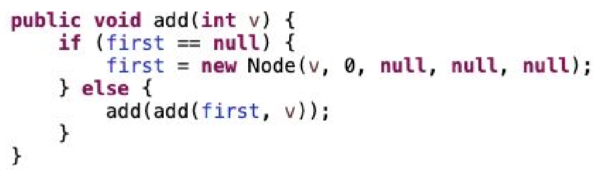
```java
/**
 *
 * Static class that represents single node in the heap.
 *
 */
static class Node {

    /**
     * Stores value.
     */
    int v;

    /**
     * Stores number of successors.
     */
    int numSuccessors;

    /**
     * Stores parent node.
     */
    Node parentNode;

    /**
     * Stores left child.
     */
    Node leftSuccessor;

    /**
     * Stores right child.
     */
    Node rightSuccessor;

    /**
     * Constructor to instantiate Node and initialize
     * it with the parameters.
     *
     * @param v the value
     * @param numSuccessors the number of successors
     * @param parentNode the parent node
     * @param leftSuccessor the left successor
     * @param rightSuccessor the right successor
     */
    public Node(int v, int numSuccessors, Node parentNode, Node leftSuccessor, Node rightSuccessor) {
        this.v = v;
        this.numSuccessors = numSuccessors;
        this.parentNode = parentNode;
        this.leftSuccessor = leftSuccessor;
        this.rightSuccessor = rightSuccessor;
    }
}
}
```

private Node first; // private field

The **MaxQueue** class consists of the following methods :

a. **public void** add(**int** v) {

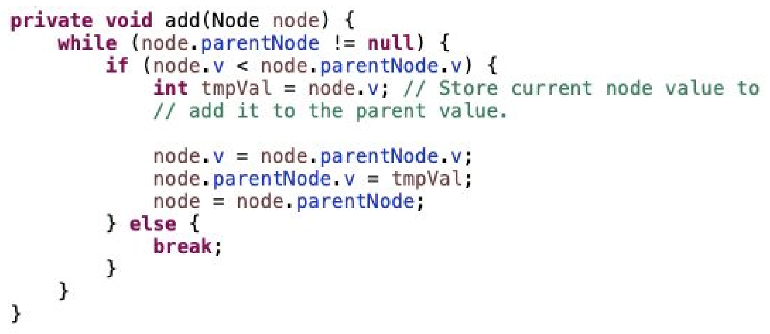
```
    // Adds value to the heap.
        if first node == null
            add new node
        else
            add as new node
    }
```



```
public void add(int v) {
    if (first == null) {
        first = new Node(v, 0, null, null, null);
    } else {
        add(add(first, v));
    }
}
```

b. **private void** add(Node node) {

//method to iterate over the nodes until parent of the current node is not null, and update every node.

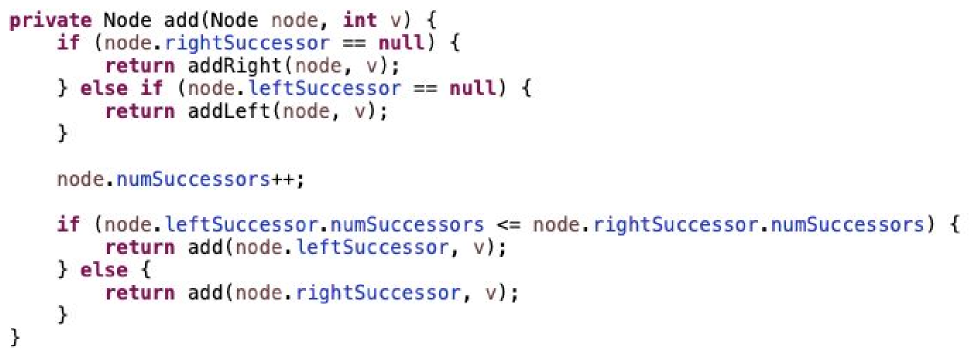


```
private void add(Node node) {
    while (node.parentNode != null) {
        if (node.v < node.parentNode.v) {
            int tmpVal = node.v; // Store current node value to
            // add it to the parent value.

            node.v = node.parentNode.v;
            node.parentNode.v = tmpVal;
            node = node.parentNode;
        } else {
            break;
        }
    }
}
```

}

c. **private** Node add(Node node, **int** v){

// Method to add the node recursively

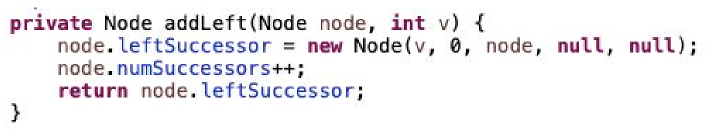


```
private Node add(Node node, int v) {
    if (node.rightSuccessor == null) {
        return addRight(node, v);
    } else if (node.leftSuccessor == null) {
        return addLeft(node, v);
    }

    node.numSuccessors++;

    if (node.leftSuccessor.numSuccessors <= node.rightSuccessor.numSuccessors) {
        return add(node.leftSuccessor, v);
    } else {
        return add(node.rightSuccessor, v);
    }
}
```

}

d. **private** Node addLeft(Node node, **int** v) {

// Creates left node and add it to the given node

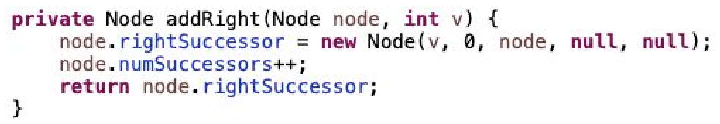


```
private Node addLeft(Node node, int v) {
    node.leftSuccessor = new Node(v, 0, node, null, null);
    node.numSuccessors++;
    return node.leftSuccessor;
}
```

}

e. **private** Node addRight(Node node, **int** v) {

// Creates right node and add it to the given node



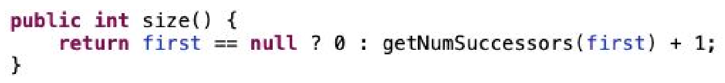
```
private Node addRight(Node node, int v) {
    node.rightSuccessor = new Node(v, 0, node, null, null);
    node.numSuccessors++;
    return node.rightSuccessor;
}
```

}

f. **public int** size(){

// * Returns size of the queue.

return 0 if first node is null, otherwise number of nodes

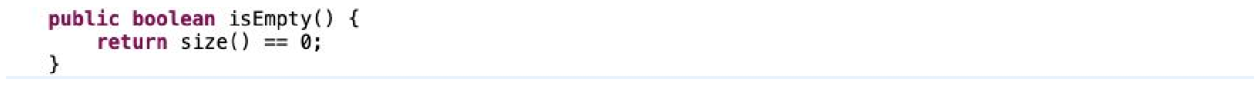


```
public int size() {
    return first == null ? 0 : getNumSuccessors(first) + 1;
}
```

}

g. **public boolean** isEmpty() {

// Returns true if queue is empty, otherwise false

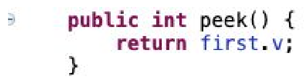


```
public boolean isEmpty() {
    return size() == 0;
}
```

}

h.**public int** peek() {

// Returns the first value of the queue

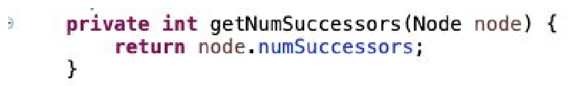


```
public int peek() {
    return first.v;
}
```

}

i. **private int** getNumSuccessors(Node node){

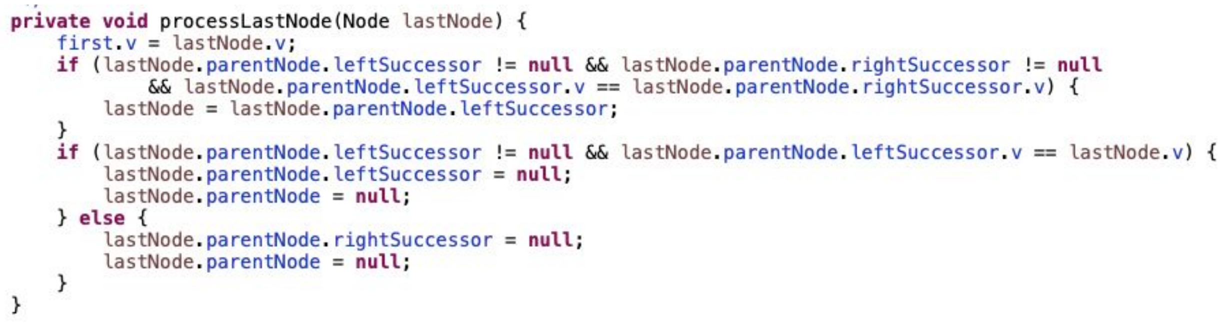
// method to return number of successors of the given node (or number of children).



```
private int getNumSuccessors(Node node) {
    return node.numSuccessors;
}
```

j. **private void** processLastNode(Node lastNode) {

// Method to process given last node. Since the first node needs to be deleted, we need to provide tree rotations.

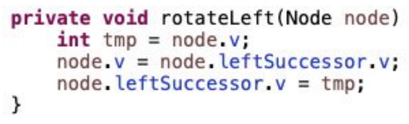


```
private void processLastNode(Node lastNode) {
    first.v = lastNode.v;
    if (lastNode.parentNode.leftSuccessor != null && lastNode.parentNode.rightSuccessor != null
            && lastNode.parentNode.leftSuccessor.v == lastNode.parentNode.rightSuccessor.v) {
        lastNode = lastNode.parentNode.leftSuccessor;
    }
    if (lastNode.parentNode.leftSuccessor != null && lastNode.parentNode.leftSuccessor.v == lastNode.v) {
        lastNode.parentNode.leftSuccessor = null;
        lastNode.parentNode = null;
    } else {
        lastNode.parentNode.rightSuccessor = null;
        lastNode.parentNode = null;
    }
}
```

}

k. **private void** rotateLeft(Node node){

// Method to perform left rotation

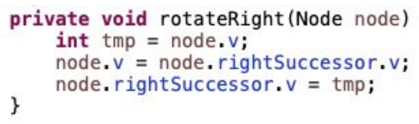


```
private void rotateLeft(Node node) {
    int tmp = node.v;
    node.v = node.leftSuccessor.v;
    node.leftSuccessor.v = tmp;
}
```

}

l. **private void** rotateRight(Node node) {

// Method to perform right rotation

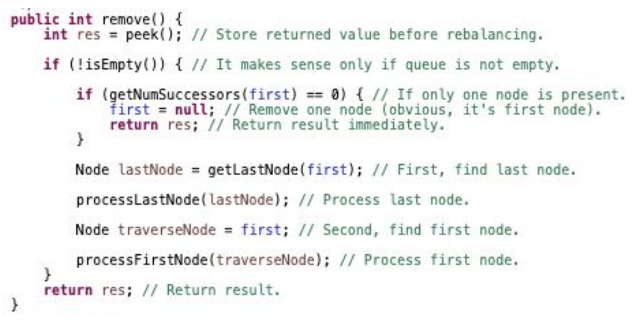


```
private void rotateRight(Node node) {
    int tmp = node.v;
    node.v = node.rightSuccessor.v;
    node.rightSuccessor.v = tmp;
}
```

}

m. **public int** remove() {

// Method to remove the first value of the queue. After removing operation the heap should be rebalanced. Rebalancing operation includes processing the last node, as well as the first node.

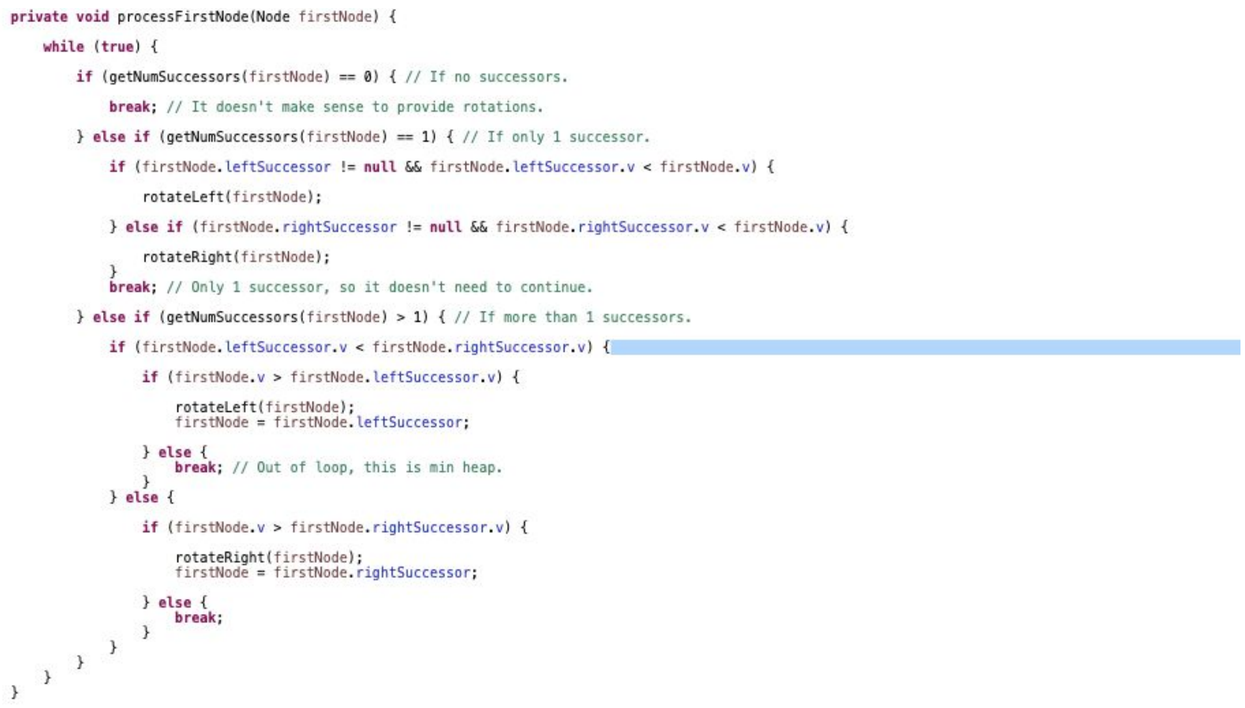


```
public int remove() {
    int res = peek(); // Store returned value before rebalancing.

    if (!isEmpty()) { // It makes sense only if queue is not empty.

        if (getNumSuccessors(first) == 0) { // If only one node is present.
            first = null; // Remove one node (obvious, it's first node).
            return res; // Return result immediately.
        }

        Node lastNode = getLastNode(first); // First, find last node.

        processLastNode(lastNode); // Process last node.

        Node traverseNode = first; // Second, find first node.

        processFirstNode(traverseNode); // Process first node.
    }
    return res; // Return result.
}
```

}

n.. **private void** processFirstNode(Node firstNode){

// method to process the first node. This operation requires actions to rotate subtress to build the min heap.

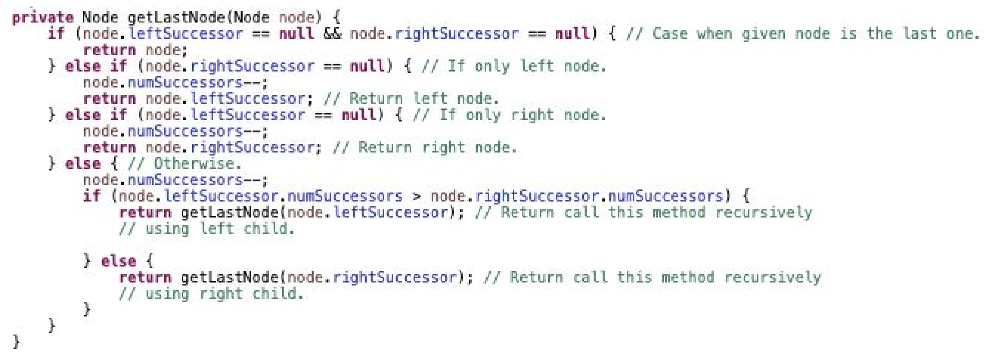


```
private void processFirstNode(Node firstNode) {

    while (true) {

        if (getNumSuccessors(firstNode) == 0) { // If no successors.

            break; // It doesn't make sense to provide rotations.

        } else if (getNumSuccessors(firstNode) == 1) { // If only 1 successor.

            if (firstNode.leftSuccessor != null && firstNode.leftSuccessor.v < firstNode.v) {

                rotateLeft(firstNode);

            } else if (firstNode.rightSuccessor != null && firstNode.rightSuccessor.v < firstNode.v) {

                rotateRight(firstNode);
            }
            break; // Only 1 successor, so it doesn't need to continue.

        } else if (getNumSuccessors(firstNode) > 1) { // If more than 1 successors.

            if (firstNode.leftSuccessor.v < firstNode.rightSuccessor.v) {

                if (firstNode.v > firstNode.leftSuccessor.v) {

                    rotateLeft(firstNode);
                    firstNode = firstNode.leftSuccessor;

                } else {
                    break; // Out of loop, this is min heap.
                }
            } else {

                if (firstNode.v > firstNode.rightSuccessor.v) {

                    rotateRight(firstNode);
                    firstNode = firstNode.rightSuccessor;

                } else {
                    break;
                }
            }
        }
    }
}
```

}

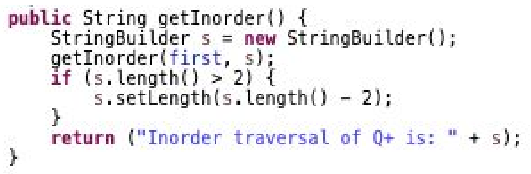
o. **private** Node getLastNode(Node node) {

// **Method to get the last node in the heap starting from the given node**.



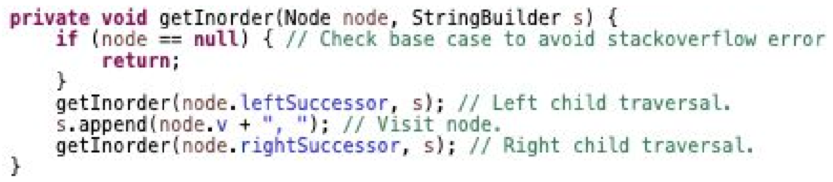
```
private Node getLastNode(Node node) {
    if (node.leftSuccessor == null && node.rightSuccessor == null) { // Case when given node is the last one.
        return node;
    } else if (node.rightSuccessor == null) { // If only left node.
        node.numSuccessors--;
        return node.leftSuccessor; // Return left node.
    } else if (node.leftSuccessor == null) { // If only right node.
        node.numSuccessors--;
        return node.rightSuccessor; // Return right node.
    } else { // Otherwise.
        node.numSuccessors--;
        if (node.leftSuccessor.numSuccessors > node.rightSuccessor.numSuccessors) {
            return getLastNode(node.leftSuccessor); // Return call this method recursively
            // using left child.

        } else {
            return getLastNode(node.rightSuccessor); // Return call this method recursively
            // using right child.
        }
    }
}
```

}
p. **public** String getInorder() {
// Method to return string representation of inorder traversal.


```
public String getInorder() {
    StringBuilder s = new StringBuilder();
    getInorder(first, s);
    if (s.length() > 2) {
        s.setLength(s.length() - 2);
    }
    return ("Inorder traversal of Q+ is: " + s);
}
```

}

q. **private void** getInorder(Node node, StringBuilder s) {
// Method to populate string builder with inorder traversal.


```
private void getInorder(Node node, StringBuilder s) {
    if (node == null) { // Check base case to avoid stackoverflow error
        return;
    }
    getInorder(node.leftSuccessor, s); // Left child traversal.
    s.append(node.v + ", "); // Visit node.
    getInorder(node.rightSuccessor, s); // Right child traversal.
}
```

}

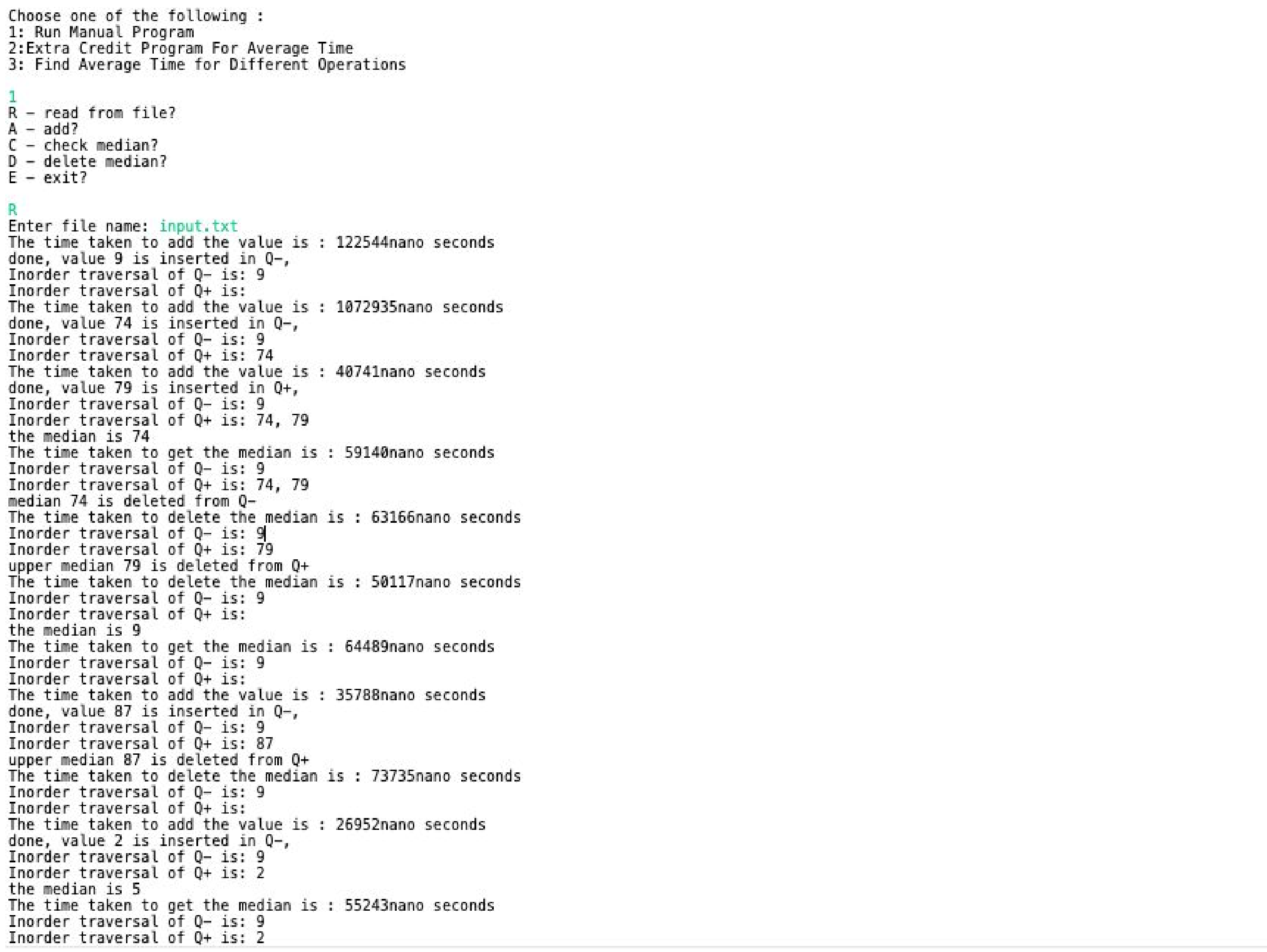
PHASE 3: RISK ANALYSIS
There is no risk to human health associated with running this program.

PHASE 4: VERIFICATION
The algorithms have been verified for all cases in the program. The code for LinkedLists and arrays has been mirrored, but separated to ensure that it runs the same, but utilizes the functions for the appropriate data types.

PHASE 5: CODING
The code has been attached in a zip file written in Java language.

Aashray Shrestha

PHASE 6: TESTING

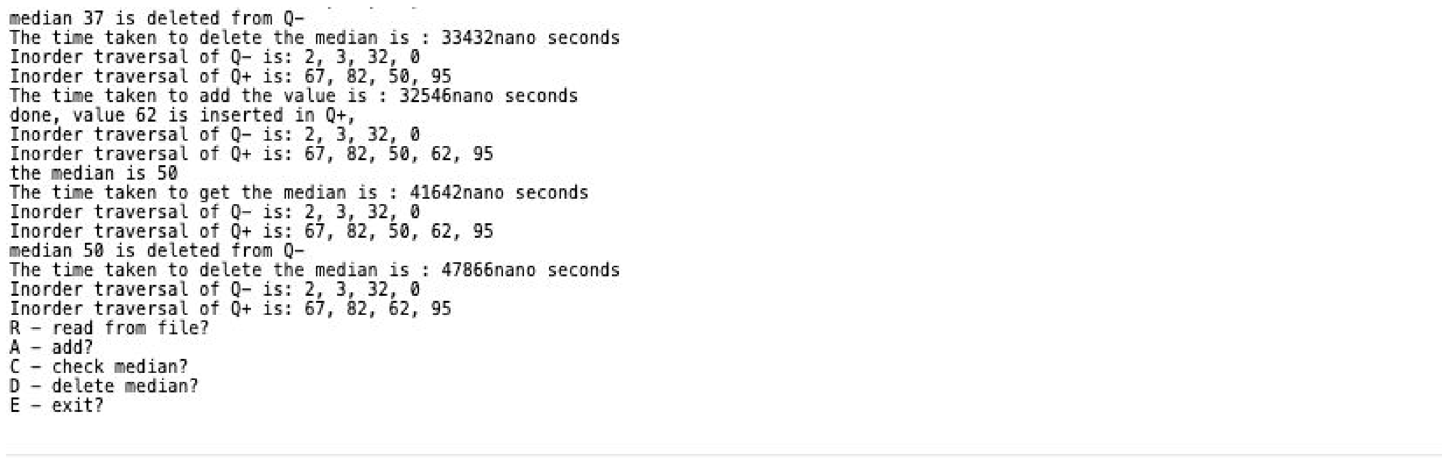


```
Choose one of the following :
1: Run Manual Program
2:Extra Credit Program For Average Time
3: Find Average Time for Different Operations

1
R - read from file?
A - add?
C - check median?
D - delete median?
E - exit?

R
Enter file name: input.txt
The time taken to add the value is : 122544nano seconds
done, value 9 is inserted in Q-,
Inorder traversal of Q- is: 9
Inorder traversal of Q+ is:
The time taken to add the value is : 1072935nano seconds
done, value 74 is inserted in Q-,
Inorder traversal of Q- is: 9
Inorder traversal of Q+ is: 74
The time taken to add the value is : 40741nano seconds
done, value 79 is inserted in Q+,
Inorder traversal of Q- is: 9
Inorder traversal of Q+ is: 74, 79
the median is 74
The time taken to get the median is : 59140nano seconds
Inorder traversal of Q- is: 9
Inorder traversal of Q+ is: 74, 79
median 74 is deleted from Q-
The time taken to delete the median is : 63166nano seconds
Inorder traversal of Q- is: 9
Inorder traversal of Q+ is: 79
upper median 79 is deleted from Q+
The time taken to delete the median is : 50117nano seconds
Inorder traversal of Q- is: 9
Inorder traversal of Q+ is:
the median is 9
The time taken to get the median is : 64489nano seconds
Inorder traversal of Q- is: 9
Inorder traversal of Q+ is:
The time taken to add the value is : 35788nano seconds
done, value 87 is inserted in Q-,
Inorder traversal of Q- is: 9
Inorder traversal of Q+ is: 87
upper median 87 is deleted from Q+
The time taken to delete the median is : 73735nano seconds
Inorder traversal of Q- is: 9
Inorder traversal of Q+ is:
The time taken to add the value is : 26952nano seconds
done, value 2 is inserted in Q-,
Inorder traversal of Q- is: 9
Inorder traversal of Q+ is: 2
the median is 5
The time taken to get the median is : 55243nano seconds
Inorder traversal of Q- is: 9
Inorder traversal of Q+ is: 2
```

▪

▪

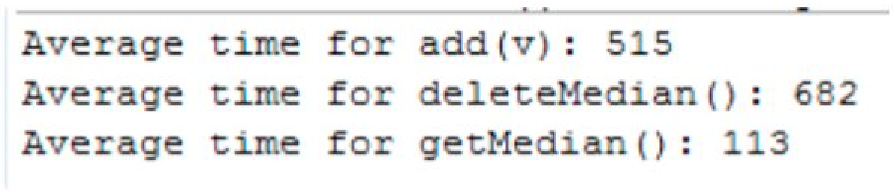
▪



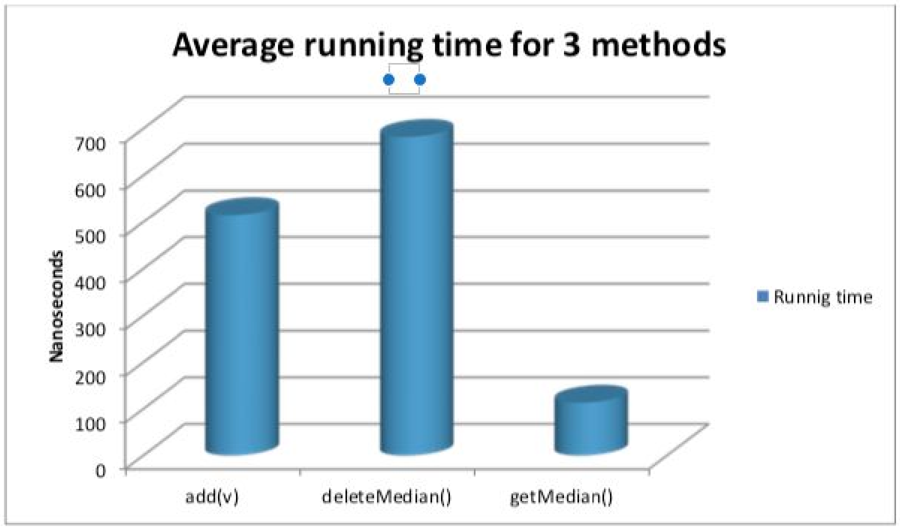
```
median 37 is deleted from Q-
The time taken to delete the median is : 33432nano seconds
Inorder traversal of Q- is: 2, 3, 32, 0
Inorder traversal of Q+ is: 67, 82, 50, 95
The time taken to add the value is : 32546nano seconds
done, value 62 is inserted in Q+,
Inorder traversal of Q- is: 2, 3, 32, 0
Inorder traversal of Q+ is: 67, 82, 50, 62, 95
the median is 50
The time taken to get the median is : 41642nano seconds
Inorder traversal of Q- is: 2, 3, 32, 0
Inorder traversal of Q+ is: 67, 82, 50, 62, 95
median 50 is deleted from Q-
The time taken to delete the median is : 47866nano seconds
Inorder traversal of Q- is: 2, 3, 32, 0
Inorder traversal of Q+ is: 67, 82, 62, 95
R - read from file?
A - add?
C - check median?
D - delete median?
E - exit?
```

```
A
ok, specify the value v to be inserted into S
3
done, value 3 is inserted in Q-,
Inorder traversal of Q- is: 2, 3, 3, 32, 0
Inorder traversal of Q+ is: 67, 82, 62, 95
R - read from file?
A - add?
C - check median?
D - delete median?
E - exit?

A
ok, specify the value v to be inserted into S
41
done, value 41 is inserted in Q+,
Inorder traversal of Q- is: 2, 3, 3, 32, 0
Inorder traversal of Q+ is: 67, 82, 41, 62, 95
R - read from file?
A - add?
C - check median?
D - delete median?
E - exit?

D
upper median 41 is deleted from Q+
Inorder traversal of Q- is: 2, 3, 3, 32, 0
Inorder traversal of Q+ is: 67, 82, 62, 95
R - read from file?
A - add?
C - check median?
D - delete median?
E - exit?

C
the median is 32
Inorder traversal of Q- is: 2, 3, 3, 32, 0
Inorder traversal of Q+ is: 67, 82, 62, 95
```

2. Average time in nanoseconds for the operations



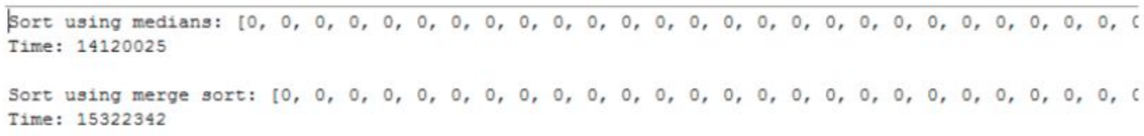
```
Average time for add(v): 515
Average time for deleteMedian(): 682
Average time for getMedian(): 113
```



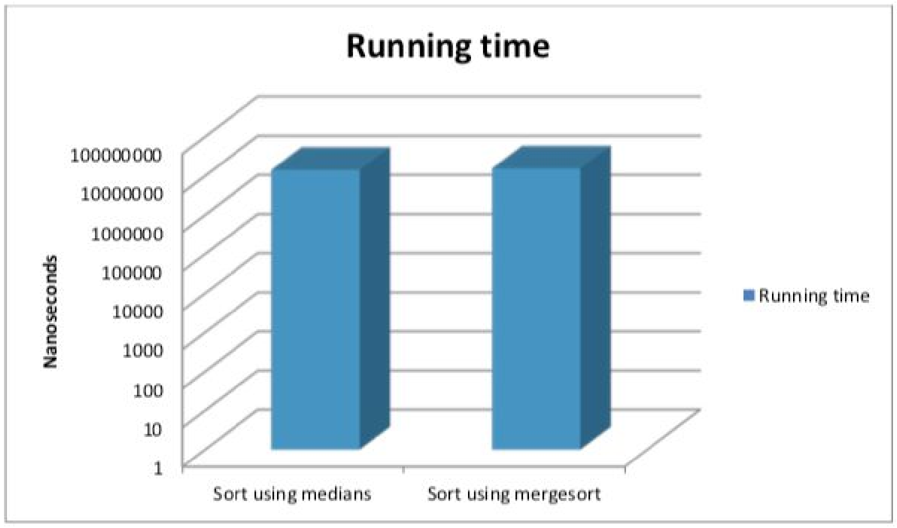
As we can see above, the results concur with the theoretical time complexities. Methods add(v) and deleteMedian() take O(log n), whereas getMedian() takes constant time (O(1)), so we see the expected results where time for getMedian() is much less than for other 2 methods.

3. Extra Credit Program For Average Time

Here is the result:



```
Sort using medians: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, C
Time: 14120025

Sort using merge sort: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, C
Time: 15322342
```



As you can see, the time values are almost the same, so this confirms the theoretical running times for both of algorithms (O(n log n).

Theoretical	**Empirical**
O (n log n)	O (n log n)

PHASE 7: REFINING THE PROGRAM
The program can be refined after receiving input and feedback from the grader.

PHASE 8: PRODUCTION
Attached with this is a zip file with the code in Java and my claims can be verified.

PHASE 9: MAINTENANCE
Program is complete and maintenance/improvement can be done after the feedback is received.

THEORETICAL VS EMPIRICAL COMPLEXITY ANALYSIS

I have attached a separate document named HW4Report.pdf which outlines the further documentation of the time complexity.