**1. Shown below is the code for the insertion sort consisting of two recursive methods that replace the two nested loops that would be used in its iterative counterpart:**

**void insertionSort(int array[]) { insert(array, 1); } void insert(int[] array, int i) { if (i < array.length) { int value = array[i]; int j = shift(array, value, i); array[j] = value; insert(array, i + 1); } } int shift(int[] array, int value, int i) { int insert = i; if (i > 0 && array[i - 1] > value) { array[i] = array[i - 1]; insert = shift(array, value, i - 1); } return insert; }**

**Draw the recursion tree for insertionSort when it is called for an array of length 5 with data that represents the worst case. Show the activations of insertionSort, insert and shift in the tree. Explain how the recursion tree would be different in the best case.**

Worst case input is

5,4,3,2,1

Insertion sort

4

1

0

5

Cell to shift

Cell to insert

1

0

1

2

2

0

1

3

0

1

2

3

4

2

3

Number in box, circle represents value of *i*

For best case:

1

2

1

2

3

3

4

4

5

**2. Refer back to the recursion tree you provided in the previous problem. Determine a formula that counts the numbers of nodes in that tree. What is Big- for execution time? Determine a formula that expresses the height of the tree. What is the Big- for memory?**

Ans: total nodes for except first and last node:

*3, 4, 5, 6 + 2 (first and last node)*

T = 2 +

T= 2 + (3+4+5+6…n+2)+1-1

= - 1

=

T = 0 ()

*Height = (n-1) +*

*At nth node branch at nth node*

*= 2n => h = 0 (n)*

*Space: 0 (n) for array*

**3. Provide a generic Java class named SortedPriorityQueue that implements a priority queue using a sorted list implemented with the Java ArrayList class. Make the implementation as efficient as possible.**

// import the required package

import java.io.\*;

import java.util.\*;

// generic class SortedPriorityQueue

public class SortedPriorityQueue<E>

{

// Declare the required variables

private ArrayList<E> SPQ;

// constructor

public SortedPriorityQueue()

{

SPQ = new ArrayList<E>();

}

// method to check list is empty

public boolean isEmpty()

{

return SPQ.size() == 0;

}

// method to add the value

public void add(E value)

{

SPQ.add(value);

}

// method to get the peek value

public E peek()

{

E min = SPQ.get(0);

for(int lp = 1; lp < SPQ.size(); lp++)

{

Comparable<E> qval = (Comparable<E>) SPQ.get(lp);

if(qval.compareTo(min) < 0) min = (E) qval;

}

return min;

}

// method to remove the values

public E remove()

{

E min = SPQ.get(0);

int minPos = 0;

for(int lp = 1; lp < SPQ.size(); lp++)

{

Comparable<E> qval = (Comparable<E>) SPQ.get(lp);

if(qval.compareTo(min) < 0)

{

min = (E) qval; minPos = lp;

}

}

SPQ.remove(minPos);

return min;

}

// main method to test the implementation of the SortedPriorityQueue

public static void main(String a[])

{

// object for the class

SortedPriorityQueue<Integer> spqi = new SortedPriorityQueue<Integer>();

// insert the values to the list

spqi.add(4);

spqi.add(7);

spqi.add(10);

spqi.add(12);

spqi.add(15);

// print the peek value

System.out.println("Peek:"+spqi.peek());

// remove the values from the queue

System.out.println("Remove:");

System.out.println(spqi.remove());

System.out.println( spqi.remove());

System.out.println(spqi.remove());

System.out.println("Peek"+spqi.peek());

}

}

**4. Consider the following sorting algorithm that uses the class you wrote in the previous problem: void sort(int[] array) { SortedPriorityQueue<Integer> queue = new SortedPriorityQueue(); for (int i = 0; i < array.length; i++) queue.add(array[i]); for (int i = 0; i < array.length; i++) array[i] = queue.remove(); }**

**Analyze its execution time efficiency in the worst case. In your analysis, you may ignore the possibility that the array list may overflow and need to be copied to a larger array. Indicate whether this implementation is more or less efficient than the one that uses the Java priority queue.**

The sorted priority queue is basically a binary heap, where the minimum element in the tree is stored at the root node and is accessible in O (1) time. However, once some element is inserted or deleted, we need to again heapify the whole tree which takes a time of log(n).

Hence the first for loop adds n elements to the queue. Each element requires log(n) effort to heapify the tree. Hence the complexity is O(nlogN)

Similarly, the second for loop, runs n times and fetches the minimum element every time and places it sequentially on the array. Fetching one element takes O(logn) time and doing it for n elements, again make the complexity as O(nlogN);

Hence the overall complexity of given program is O(nlogN)