**CMSC 451 Homework 3**

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;

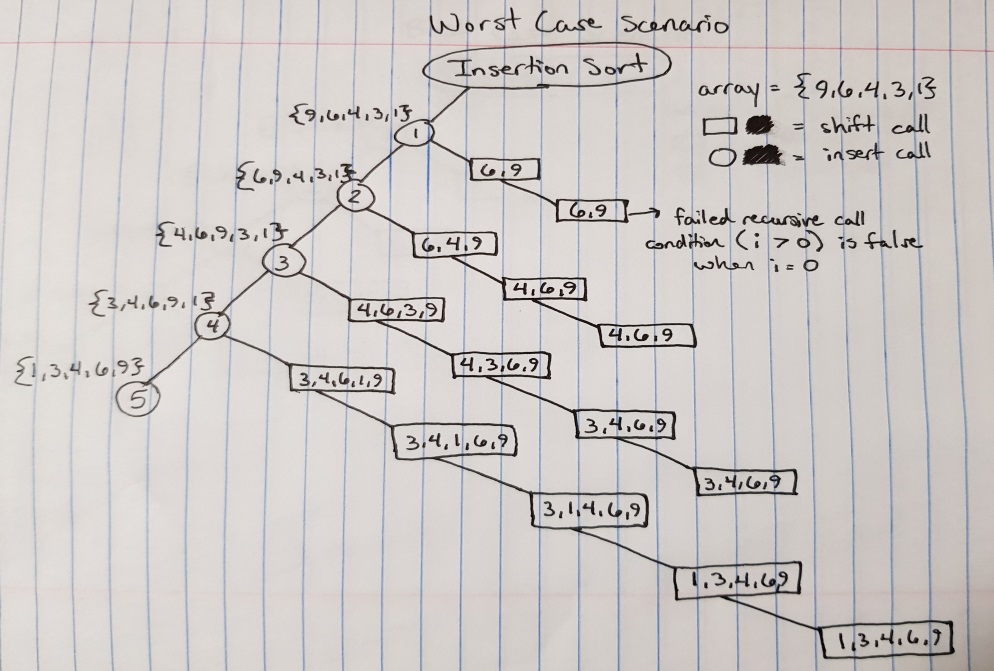
}

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.

**SOLUTION**

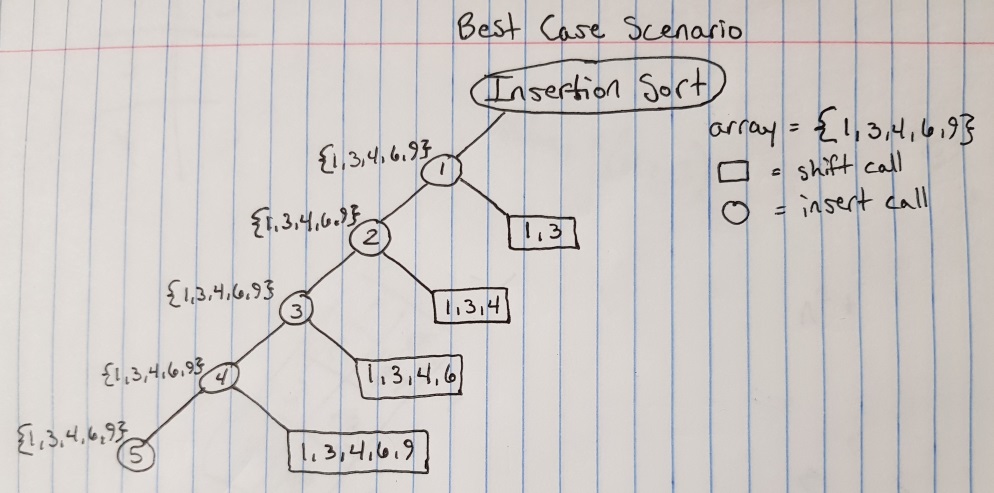
Since insertion sort works by comparing the current element to all the previous elements in an array and shifting them right until an element with a lesser value is found; the worst case would occur when the array is in descending order. In such a scenario, for every insert call the array would have to shift all the pervious elements before inserting the current element in proper order.

**Worst Case Recursion Tree**



When the insertion sort function is called, the first element to be sorted is actually the second element in the array. Since 6 < 9, the value 9 is shifted right and value 6 is inserted in its place. The shift function is called again, but this time it fails because i is now equal to 0, when it needs to be greater than 0 to meet the conditions of the shift function. Thus, there is always at least one shift function call made where i is equal to 0. In the best case scenario, the array would already be sorted in ascending order and so only one shift call would be needed to check the order for every insert call.

**Best Case Recursion Tree**



1. 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-ϴ for execution time? Determine a formula that expresses the height of the tree. What is the Big-ϴ for memory?

**SOLUTION**

**Formula for Number of Nodes**

Worst Case

*Let k = n - 1*

Best Case

**Execution Time:**

Worst Case

Average Case

Best Case

**Formula for Height of Tree**

Worst Case

Best Case

**Efficiency of Memory Utilization**

1. 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.

**SOLUTION**

import java.util.ArrayList;

import java.util.Arrays;

import java.util.concurrent.ThreadLocalRandom;

public class SortedPriorityQueue<T> {

// Declare an object of ArrayList type.

private ArrayList<T> sortedList;

// Default Constructor for Sorted List

SortedPriorityQueue() {

sortedList = new ArrayList<T>();

}

// Adds Value to Sorted List

public void add(T value) {

// If Queue is Empty, Add Value

if (sortedList.size() == 0) {

sortedList.add(value);

// Else Find Ordered Position, Add Value,

// And Shift Following Values

} else {

T x = value;

for (int i = 0; i < sortedList.size(); i++) {

if (sortedList.get(i).toString().compareTo(x.toString()) < 0) {

x = sortedList.get(i);

sortedList.set(i,value);

value = x;

}

}

sortedList.add(x);

}

}

// Removes Last Index from List

public T remove() {

T x = sortedList.remove(sortedList.size() - 1);

return x;

}

// Given Algorithm from Question 4 for Sorting an Integer Array

static void sort(int[] array) {

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

for (int i = 0; i < array.length; i++) {

spq.add(array[i]);

}

for (int i = 0; i < array.length; i++) {

array[i] = spq.remove();

}

}

// Main Method

public static void main(String[] args) {

// Generates Test Array with Random Integers (0-9)

int[] array = new int[8];

for (int i = 0; i < array.length; i++) {

array[i] = ThreadLocalRandom.*current*().nextInt(0, 10);

}

// OUTPUT BEFORE SORT

System.***out***.println("Array BEFORE Sorting: ");

System.***out***.println(Arrays.*toString*(array)+"\n");

// OUTPUT AFTER SORT

*sort*(array);

System.***out***.println("Array AFTER Sorting: ");

System.***out***.println(Arrays.*toString*(array));

}

}

Output:

Array BEFORE Sorting:

[7, 8, 0, 5, 1, 9, 1, 2]

Array AFTER Sorting:

[0, 1, 1, 2, 5, 7, 8, 9]

1. 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.

**SOLUTION**

During execution, the add() method is called n times and the remove() method is also called n times. In the worst case scenario, when the add() method is called, the list has to be fully traversed to find the appropriate place for the value being added. As such, the execution time for the add() method is O(n). On the other hand, the remove() method simply removes the last element of the list so its execution time is constant O(1).

Therefore the execution time of this sorting algorithm is:

Since the overall efficiency of the Java priority queue is , and is more efficient than , this sorting algorithm is less efficient than the Java priority queue.

**Grading Rubric**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Problem** |  | **Meets** |  | **Does Not Meet** |  |
|  |  |  | **10 points** |  | **0 points** | |
|  |  |  |  |  |  | |
|  |  |  |  |  |  | |
|  | **Problem 1** |  | Recursion tree is drawn correctly (8) |  | Recursion tree is not drawn correctly | |
|  |  |  | (0) | |  |
|  |  |  |  |  |
|  |  |  |  |  |  | |
|  |  |  | Best case tree is described correctly (2) |  | Best case tree is not described | |
|  |  |  |  |  | correctly (0) | |
|  |  |  |  |  |  | |
|  |  |  | **10 points** |  | **0 points** | |
|  |  |  |  |  |  | |
|  |  |  |  |  |  | |
|  |  |  | Provided correct formula for number of |  | Did not provide correct formula for | |
|  |  |  | nodes in tree (3) |  | number of nodes in tree (0) | |
|  |  |  |  |  |  | |
|  | **Problem 2** |  | Provided correct Big-Theta for |  | Did not provide correct Big-Theta for | |
|  |  | execution time (2) |  | execution time (0) | |
|  |  |  |  |
|  |  |  |  |  |  | |
|  |  |  | Provided correct formula for tree |  | Did not provide correct formula for | |
|  |  |  | height (3) |  | tree height (0) | |
|  |  |  |  |  |  | |
|  |  |  | Provided correct Big-Theta for memory |  | Did not provide correct Big-Theta for | |
|  |  | (2) | |  | memory (0) | |
|  |  |  |  |  |  | |
|  |  |  | **10 points** |  | **0 points** | |
|  |  |  |  |  |  | |
|  |  |  |  |  |  | |
|  |  |  | Provided class correctly implements a |  | Provided class does not correctly | |
|  |  |  | priority queue (4) |  | implement a priority queue (0) | |
|  |  |  |  |  |  | |
|  |  |  | Provided class is generic (1) |  | Provided class is not generic (0) | |
|  |  |  |  |  |  | |
|  | **Problem 3** |  | Provided class uses an array list (1) |  | Provided class does not use an array | |
|  |  |  |  |  | list (0) | |
|  |  |  |  |  |  | |
|  |  |  | List in class is maintained in sorted |  | List in class is not maintained in sorted | |
|  |  |  | order (2) |  | order (0) | |
|  |  |  |  |  |  | |
|  |  |  | Implementation is most efficient (2) |  | Implementation is not most efficient | |
|  |  |  |  | (0) | |  |
|  |  |  |  |  |  | |
|  |  |  | **10 points** |  | **0 points** | |
|  |  |  |  |  |  | |
|  |  |  |  |  |  | |
|  | **Problem 4** |  | Provided correct worst case analysis (8) |  | Did not provide correct worst case | |
|  |  |  |  | analysis (0) | |
|  |  |  |  |  |
|  |  |  |  |  |  | |
|  |  |  | Provided correct efficiency comparison |  | Did not provide correct efficiency | |
|  |  |  | to Java priority queue (2) |  | comparison to Java priority queue (0) | |
|  |  |  |  |  |  |  |