Student: Dakin Werneburg
Section 6381

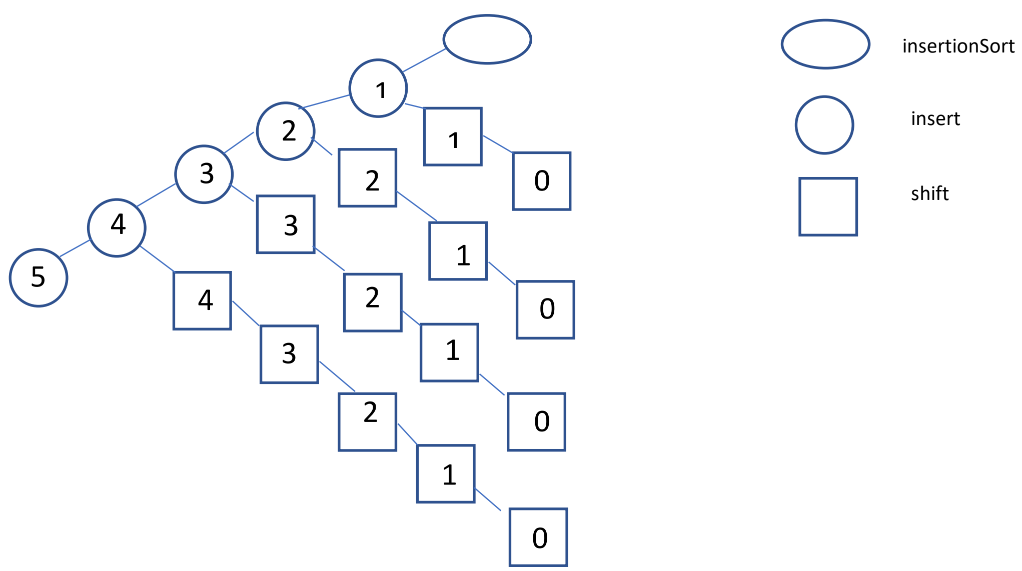
Instructor: Susan Furtney
Date: February 3, 2019

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:

Assuming the array is sorted, there would only be one activation of shift() for each activation of insert() because it would reach shift() base case each time. This would result in linear time complexity O(n) for a best case.

Student: Dakin Werneburg
Section 6381

Instructor: Susan Furtney
Date: February 3, 2019

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

Solution:

#of nodes = $\frac{n^2+3n}{2}$

$$\sum_{i=1}^{n} i + 1 = \frac{n(n+1)}{2} + n = \frac{n^2+3n}{2}$$

Big-Θ for execution time: $\Theta(n^2)$

Height of Tree = n+2

Big-Θ for memory: $\Theta(n)$

Student: Dakin Werneburg Instructor: Susan Furtney
Section 6381 Date: February 3, 2019

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.

Solution:

```
public class SortedPriorityQueue<T extends Comparable> {

	private ArrayList<T> queue;

	public SortedPriorityQueue(){
		queue = new ArrayList<T>();
	}

	public void add(T element){
		int n = queue.size();
		while(n>0 && element.compareTo(queue.get(n-1))<0){
			n--;
		}
		queue.add(n, element);
	}

	public T remove(){
		return queue.remove(queue.size()-1);

	}
}
```

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.

Solution:

In the sort algorithm, there is an instantiation of the SortedPriorityQueue. This is O(1). There are two For-Loops which need to execute at least n times. The first one will need to call add method of the SortedPriorityQueue which consists of an assignment to a variable n. This is O(1). It then needs to find the correct spot to add the element. We start by searching from the end which should have the highest value; however, in the worst case, it will need to search the entire array. This will be O(n). Then to add the element it will need to copy the array and insert the element which is O(n). Therefore adding each element for the sort algorithm will be $O(n^2)$, one O(n) for the add for-loop in sort() and O(n) for the queue.add() in SortedPriorityQueue.

The second For-Loop of the sort() algorithm, need to run n times to remove each element in the queue. However, to remove the element is just O(1) in the remove method of SortedPriorityQueue because it just removing the last element because its already in sorted order.

Therefore, the total time complexity is $O(n^2)$. This is less efficient that the java priority queue.