Ashcon Abae

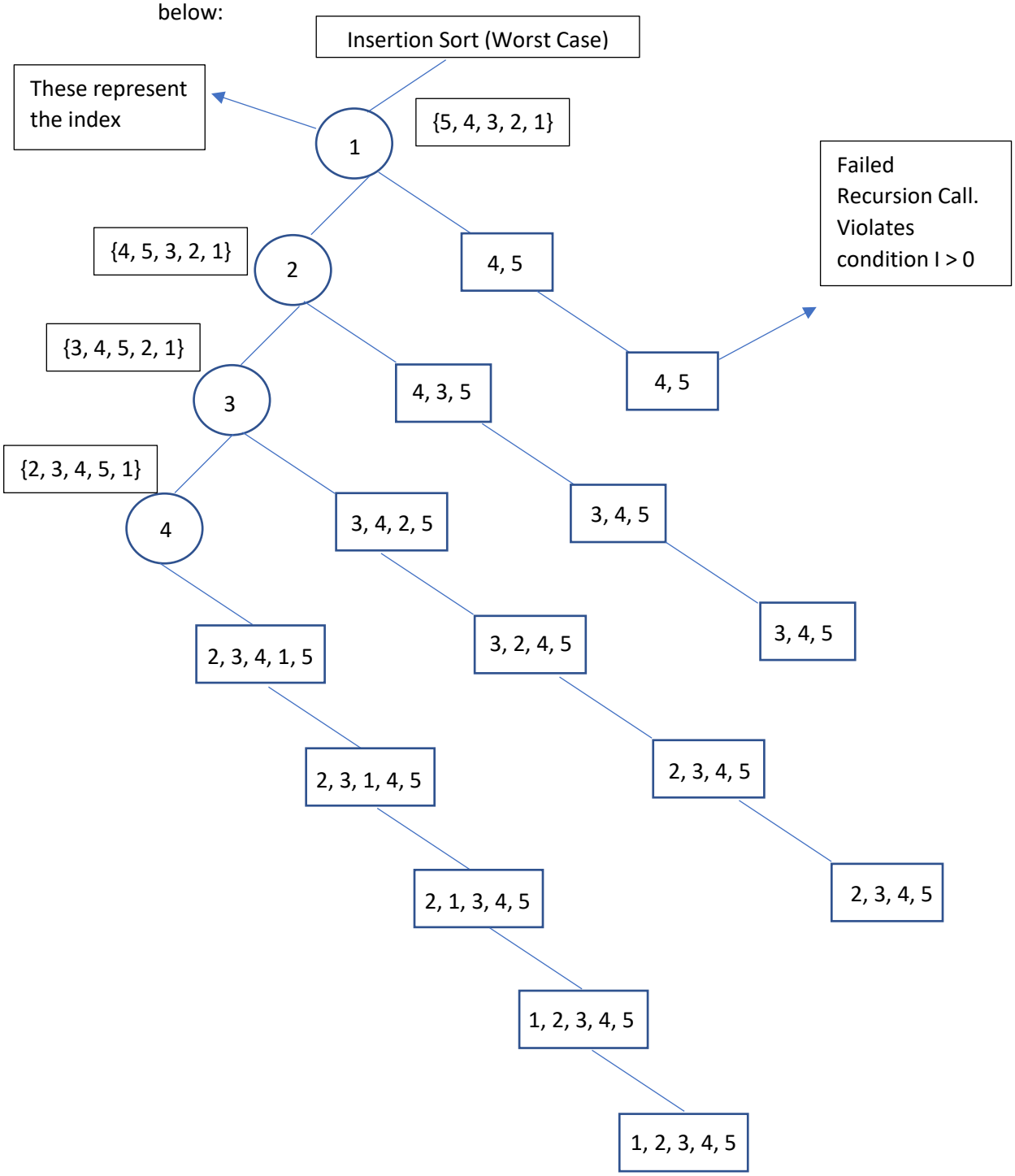
Design & Analysis of Computer Algorithms

10 November 2019

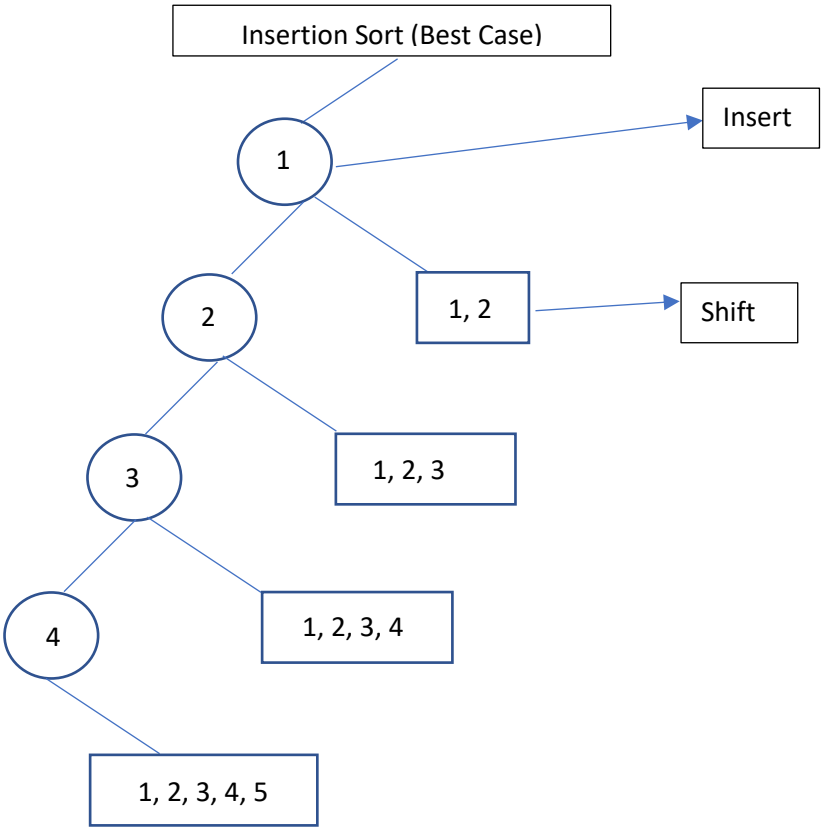
Homework 3

1.

a. We will consider the worst-case recursion tree for insertionSort with array of length 5 below:



b. In the recursion tree above, the circular nodes represent a call to insert while the rectangular nodes represent a call to shift. The worst case in insertion sort is when the array is sorted in reverse order. The best case for insertion sort would be when the array is already sorted. Hence, there won't be numerous calls to shift; just one call to check the elements of the array are sorted.
Thus, our best case insertion sort recursion tree with array length of 5 would be:



2. The following formula counts the number of nodes in the insertion sort recursion tree from problem 1:

$$
\begin{aligned}
&1 + \sum_{i=3}^{n+1} i \\
&= \left(\sum_{i=1}^{n+1} i\right) - 2 \\
&= \frac{(n+1)(n+2)}{2} - 1 \\
&= \frac{n^2 + 3n + 2}{2} - 1 \\
&= \frac{n^2 + 3n}{2}
\end{aligned}
$$

So for our array of size 5 from problem 1, the number of nodes in our tree would be $\frac{5^2+3(5)}{2} = \frac{25+15}{2} = \frac{40}{2} = 20$. As we discussed in problem 1, worst case for insertion sort is when the array is sorted in reverse order. The time complexity for this worst case is $O(n^2)$. The best case for insertion sort is when the array is already sorted. The time complexity for this best case is O(n). Our time complexity can be represented as $\Theta(n^2)$.
The formula for expressing the height of our recursion tree would simply be n – 1. That is, the memory for this algorithm can be expressed as $\Theta(n)$

3. The following java code implements a priority queue using a sorted list with the built-in ArrayList class:

```
import java.util.*;
import java.util.ArrayList;

public class SortedPriorityQueue<T extends Comparable<T>>
{
	//our queue representation
	public ArrayList<T> spq;

	//our constructor simply initializes the spq arraylist
	public SortedPriorityQueue()
	{
		spq = new ArrayList<T>();
	}

	//adding an element to the sorted priority queue
	public void add(T item)
	{
		if(spq.isEmpty())
			spq.add(item);
		else
		{
			//adding before current first object if necessary
			if(item.compareTo(spq.get(0)) < 0)
			spq.add(0, item);
			else
			{
				//sanity check and to see if item needs to be added at end of
				queue
				boolean added = false;

				for(int i = 0; i < spq.size() - 1; i++)
				{
```

```
                                        if(item.compareTo(spq.get(i)) >= 0 &&
                                        item.compareTo(spq.get(i + 1)) < 0)
                                        {
                                                spq.add(i + 1, item);
                                                added = true;
                                                break;
                                        }
                                }

                                if(!added)
                                {
                                        spq.add(item);
                                }
                        }
                }
        }

//removing and returning last element from the sorted priority queue if not empty, null
otherwise
public T remove()
{
        if(!spq.isEmpty())
        {
                if(spq.size() > 1)
                {
                        return spq.remove(spq.size() – 1);
                }
                else
                        return spq.remove(0);
        }
        return null;
}

//helper functions
public boolean isEmpty()
{
        return spq.isEmpty();
}

@Override
public String toString()
{
        return spq.toString();
}
```

}

4. The provided sort algorithm calls add() n times then proceeds to call remove() n times. Therefore, the running time of the provided sort algorithm would be (n * (time of add()) + (n * (time of remove()). The running time of the add method I created is n because our queue is fully traversed each time traversal takes place. The running time of the remove method I created is constant because it is simply removing the last element. Therefore, the total running time of the provided sort algorithm is:

$$O\big((n * n) + (n * 1)\big) = O(n^2)$$

Thus, this implementation is less efficient than the one that uses the Java Priority Queue.