Alexander Shah
Homework 7: More on Trees and Sorting
EN.605.202.81 Section 84

1.

A right in-threaded binary tree has pointers for leaves to successor nodes. This can be implemented in arrays as follows.

```
Define TreeNode {
        DataType Data
        TreeNode Left, Right
}

Define treeArray as Array
Define RThreadArray as Array

Method MakeTree(index, DataType item):
        TreeNode Temp = new TreeNode
        Temp.Data = item
        Temp.Left = null //no left child
        Temp.Right = null //no right child
        RThreadArray[index] = false //no rthread
        treeArray[index] = Temp

Method SetLeft(parentIndex, DataType item):
        leftIndex = 2 * parentIndex + 1
        if (treeArray[parentIndex] == null) or (leftIndex >= Length of treeArray)
                // handle error
        else if (treeArray[leftIndex] != null)
                // handle error, parentIndex can't have existing left child
        else
                //make new left and thread
                MakeTree(leftIndex, item)
                treeArray[leftIndex].Right = parentIndex
                RThreadArray[leftIndex] = true

Method SetRight(parentIndex, DataType item):
        rightIndex = 2 * parentIndex + 2
        if (treeArray[parentIndex] == null) or (rightIndex >= Length of treeArray)
                // handle error
        else if (!RThreadArray[parentIndex])
                // handle error, parentIndex can't have existing right child
        else
                MakeTree(rightIndex, item)

                //copy IOS from parent to right, add rthread
```

```
treeArray[rightIndex].Right = treeArray[parentIndex].Right
RThreadArray[rightIndex] = true

//update parent right and remove rthread
treeArray[parentIndex].Right = rightIndex
RThreadArray[parentIndex] = false
```

2.

We can traverse tree by traversing the sequential array based on how we assigned indexes to children nodes.

```
Method IOT(idx):
    while (idx != null) and (idx < len(treeArray))
        // find left
        while ((2 * idx + 1 < len(treeArray)) and (treeArray[2 * idx + 1] != null))
            idx = 2 * idx + 1

            //show or do something with the data
            print(treeArray[idx].Data)

            // find right
            if (RThreadArray[idx] == true) //find thread
                idx = treeArray[idx].Right
            else
                //no thread
                idx = 2 * idx + 2
```

3.

```
Define A as Array
Define Count as Array
Define Output as Array

n = len(A)

//for each element
for i from 0 to n:
   current_small = A[i]
   //compare to all other elements
   for j from 0 to n:
      if A[j] < current_small:
         //to count up difference to smallest element
         Count[i] += 1
   //Stores element at A in its sorted place, the number of spaces away from smallest is its index
```

```
    Output[Count[i]] = A[i]
return Output

//Example
A=[2,4,3,5,6]
Count = [0,2,1,3,4]
Output = [2,3,4,5,6]
```

The Output array needs to be size n, the same size as the unique number of elements in the array. If there are repeated values they will be overwritten.

4.

This method executes in O(n^2) time as it requires comparing every element against every other element (in a double nested loop) and requires O(n) space to store multiple copies of n size arrays. The order to data does not affect the speed or space requirements for the algorithm as it still needs to do just as many comparisons and make just as many arrays.

5.

We can compare each element to the next element and check for min and max at each step while iterating through the array. This would be accomplished by iterating through the array by twos. The value at A[i] will be compared to the next value A[i+1], and based on which is larger we would then check whether our min and max should be updated. By iterating by twos, we cut the search in half. So overall there should be n/2 comparisons, instead of the naive approach checking every value for min and max which would take n time, or iterating through the array to find min then max, which would require 2n.

e.g.

```
Define max_val as Integer
Define min_val as Integer
max_val = min_val = A[0]
i=0

while i < n-1:
    if A[i] < A[i+1]:
        max_val = max(max_val, A[i+1])
        min_val = min(min_val, A[i])
    else:
        max_val = max(max_val, A[i])
        min_val = min(min_val, A[i + 1])
    i+=2
```