Homework 2 - Solution

C-2.5

We will use 2 queues, *Q1* and *Q2*, where *Q1* is used to store elements *Q2* is used for auxiliary data structure for maintaining FILO for stack.

To implement the stack using 2 queues, *Q1* and *Q2*, we will enqueue elements into *Q1* whenever a push call is made. This takes $O(1)$ time to complete.

For pop calls, we can dequeue all elements of *Q1* and enqueue them into *Q2* except for the last element which we set aside in a temporary variable. We then return the elements to *Q1* by dequeuing from *Q2* and enqueuing into *Q1*. The last element which we set aside in a temporary variable earlier is then returned as the result of the pop. Thus, performing a pop takes $O(n)$ time.

C-2.12

The depth of a node of a tree *T* can be obtained by incrementing the parent node's depth by 1 in a pre-order (or post-order) traversal . To determine the depth of each subtree of *T* we can thus use the recursion method. For determining the terminating condition for this recursive call, we can take into fact that if tree *T* doesn't have any subtree, the depth of it will be 0.

Algorithm computeDepth(binaryTree *node*, int *depth*):
Input: Root of binary tree(subtree) *node*, Current depth *depth*
Output: Compute the depth of all nodes

```
if node.isExternal(node) then
        return 0
if node.isExternal(node.leftChild()) and node.isExternal(node.rightChild()) then
        return 0
node.depth ← depth
computeDepth(node.leftChild(), depth+1)
computeDepth(node.rightChild(), depth+1)
```

We will call computeDepth(*root*,*0*) to compute the depth for all nodes of tree *T*. The algorithm visits each node only once and each visit takes constant time. Therefore, the running time for the algorithm is $O(n)$.

A-2.2

To find the least common ancestor, we will check with the root node, if any of them is *x* or *y* or *externalNode*, these will serve as the exit conditions for our recursive call. If we do not find either of the element as *root*, then we further divide it into two parts, left subtree and right subtree and repeat the same. Thus while recursive callbacks, the node where left subtree contains

one element and the right subtree contains another, it becomes the lowest common ancestor. Its pseudocode can be written as:

Algorithm lowestCommonAncestor(Node *root*, Node *x*, Node *y*)
Input: The *root*, *x* and *y* node elements of the binary tree
Output: The lowest common ancestor node element for nodes *x* and *y* from the binary tree

```
if root.isExternal(root) then
        return root
if (root.isRoot(x) or root.isRoot(y)) then
        return root
Node l ← lowestCommonAncestor(root.leftChild(), x, y)
Node r ← lowestCommonAncestor(root.rightChild(), x, y)
if (node.isExternal(l)) then
        return r
if (node.isExternal(r)) then
        return l
else
        return root
```

Here the recursive function is called for the no. of times equal to the height of the tree in worst case. Thus, the running time of the algorithm is O(*h*), where *h* is the height of the tree.

C-3.2

We will use a divide and conquer approach, which is used in binary search. In this approach we compare the middle elements of arrays *S* and *T*, let us call this *midS* and *midT* respectively. We then try to divide the search space of finding *k*th smallest key either by diving *S* or *T* based on the value of *k* and the values of *midS* and *midT*. In this way, we define a new sub problem with half the size of one of the arrays, thus the running time of this algorithm will be *O*(log *n*). Note that this algorithm works assuming there are no duplicate elements.

Algorithm findKthSmallest(int[] *S*, int[] *T*, int *k*)
Input: The sorted arrays *S* and *T* of size n, and the value of *k*
Output: The *k*th smallest key in the union of the keys from *S* and *T*

```
if (S.length = 0) then
        return T[k]
if (T.length = 0) then
        return S[k]
int midS ← S.length / 2
int midT ← T.length / 2
if (midS + midT < k) then
        if (S[midS] > T[midT]) then
                return findKthSmallest(S, T[midT+1,T.length], k-midT-1)
        else
```

```
                return findKthSmallest(S[midS+1,S.length], T, k-midS-1)
    else
        if (S[midS] > T[midT]) then
                return findKthSmallest(S[0,midS], T, k)
        else
                return findKthSmallest(S, T[0,midT], k)
```

C-3.6

In a binary search tree, a value less than a node goes to its left subtree and a value greater than or equal to node will go to the right subtree. Thus, we start by comparing the root to the value of *k* and at each level (height), we deal with only one child node based on the value of parent node, we go to either of its left or right child. Since we have a unique node to deal with at each level, the cost of traversing a tree will be *O(h)* where *h* is the height of the tree.

With *s* number of items to be deleted, it will take *O(s)* time to delete these in addition to finding the node. Thus the total running time for removeAllElements will be O(*h* + *s*).

Algorithm removeAllElements(*root, k*)
Input: A tree *T* with its root as *root*, and the key to be removed *k*
Output: Updated Tree *T*

```
    if root.isExternal(root) then
        return root
    while root.key = k then
        root ← remove(root)
    if root.key < k then
        return removeAllElements(root.rightChild(), k)
    else if root.key > k then
        return removeAllElements(root.leftChild(), k)
    return root
```

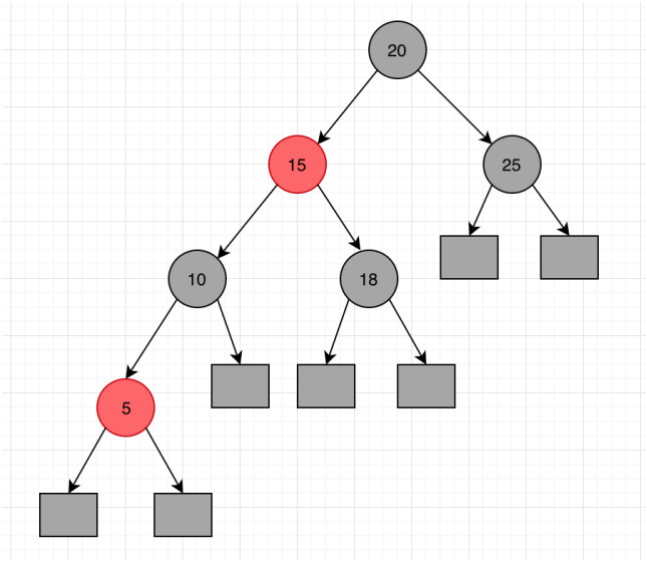
A-3.1

When a drug request comes in, specified as $\{x_1, x_2, \ldots, x_k\}$, the main idea is to coordinate *k* binary searches in *T* so as to avoid wasted comparisons.

There are several ways to do this, but one way is as follows:
First, do a binary search in *T* to locate the successor of $x_{k/2}$, which is at, say, index *i*.
Then, recursively search in subarray of *T* with indices less than *i*, for the items $x_1, \ldots, x_{\left(\frac{k}{2}\right)-1}$, and recursively search in subarray of *T* with indices greater than *i*, for the items $x_{\left(\frac{k}{2}\right)+1}, \ldots, x_k$.
To see that this search runs in *O(k log(n/k))* time, note that total time we spend is maximized when the successor items are distributed as far apart as possible, that is, at every *(n/k)th* position.

In this case, the first search takes log n time, the two on either side take 2 log(n/2) time, the next level of 4 take 4 log(n/4) time, and so on, with the final ones taking (k/2) log(n/(k/2)) time, which all adds up to a value that is *O(k log(n/k))*.

R-4.15



This is a valid red-black tree:

- The root node is black.
- All external nodes are black.
- All the leaf nodes have the same black depth.

Note, that the black height and actual height in a red-black tree can differ by a factor of 2. Also, this is not a AVL tree because the depth difference between external nodes is 2, greater than 1.

C-4.2

Base Case:
$k = 2, then\ F_k\ =\ F_{k-1} + F_{k-2}$

Induction:
$F_2\ =\ F_1 + F_0 = 1$
$F_3\ =\ F_2 + F_1 = 1 + 1 = 2 \geq 1.618 = \varphi$

Induction Hypothesis:
Assume that, it is true for $k = j - 1$ for all j with $3 \leq\ j - 1 <\ j$ and have to show that F_j is true.
$F_j\ =\ F_{j-1} + F_{j-2}$ (from definition of F_j)

$\geq \varphi^{j-3} + \varphi^{j-4}$ (from induction hypothesis)
$= \varphi^{j-4}(\varphi + 1)$
$= \varphi^{j-4} * \varphi^2 = \varphi^{j-2}$
Therefore, $F_k \geq \varphi^{j-2}$ for $k \geq 3$.

A-4.6

This is a case of Bin Packing Problems: In this type of problems, we have to minimize the number of bins, where the items (here, images) arrive one at a time (in unknown order), each must be put in a bin (here, USB drives), before considering the next item.

First Fit Heuristic: In this, for each image, *I*, you would store it on the first USB drive where it would fit, considering the drives in order by their remaining storage capacity. These algorithms are used where items arrive at run time i.e. one at a time.

Here, we can reduce the running time from *O(mn)* to *O(m log n)* by using a balanced binary search tree (e.g. AVL trees, Red-black trees).

Algorithm:
We start by inserting the first USB as the root of a Balanced Binary *T* with the key 1GB as its remaining Capacity. We copy the first image *I* into this USB and reduce the key of the node to its remaining capacity *d*. We now maintain the tree *T* ordered by their remaining storage capacity. Next, for each image, *I*, we do a search in *T* to find the smallest drive, *d*, with a reserve capacity larger than the size of I. Then, we remove *d* from *T*, decrement its capacity by the size of *I*, and re-insert it back into *T* with this new capacity as its search key. If there is no capacity available, we insert a new USB (node) with capacity 1GB into the tree *T*.

Each of these operations takes $O(\log n)$ time, and repeating them for each image will implement the first fit algorithm. Thus, the entire running time of this implementation is $O(m \log n)$.

The given problem is NP-hard combinatorial optimization problem.