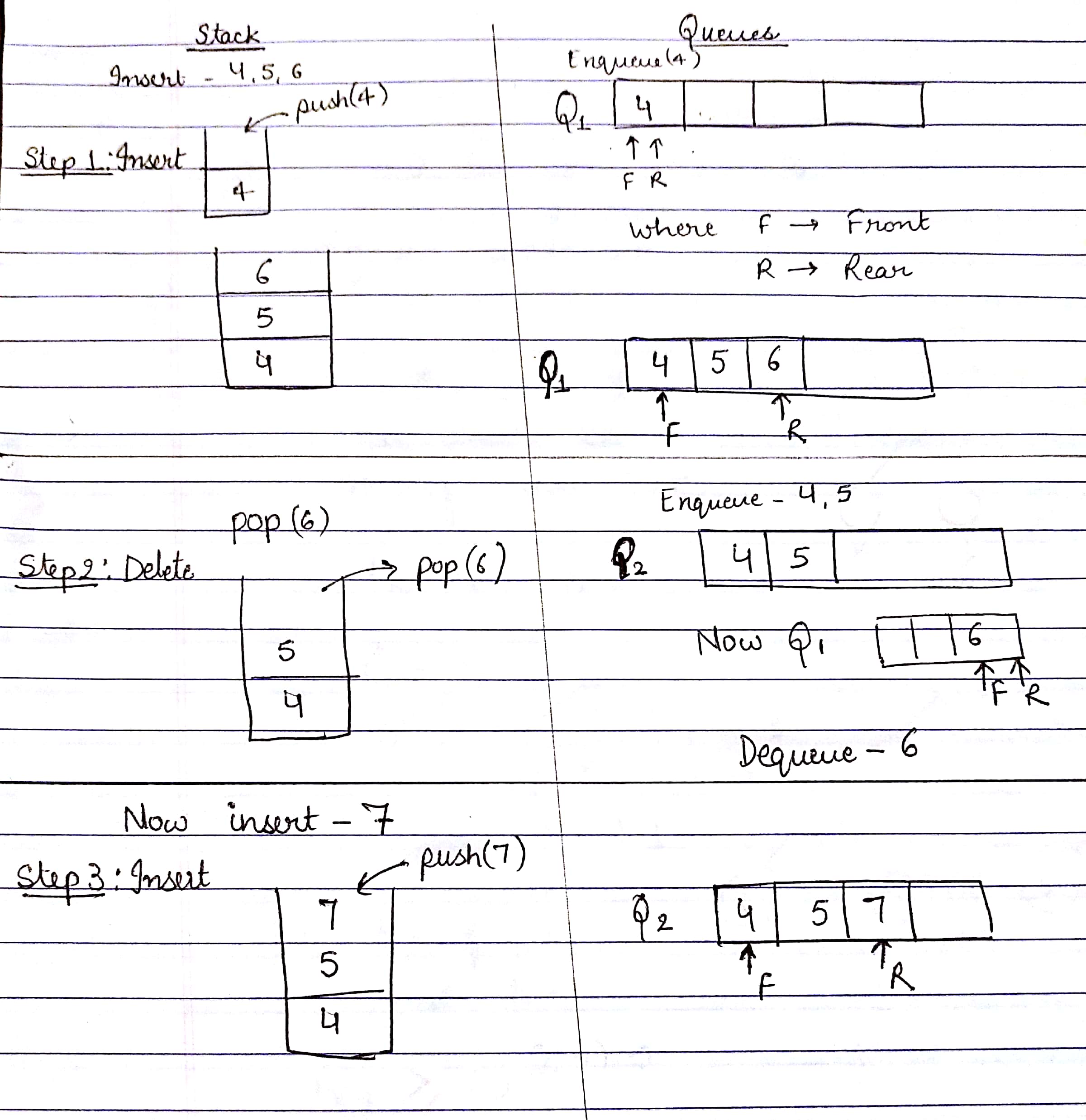
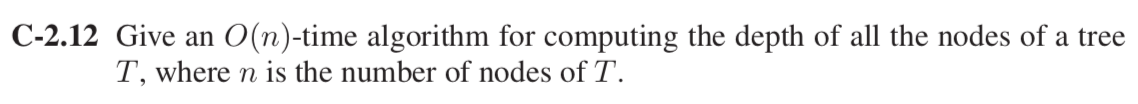


**Solution:** We can implement stack using two queues by using this approach.

Lets take 2 queues Q1 and Q2 for push() operation check Q1 is empty if it is empty start enqueue elements(lets say 4,5,6). Inserting an element is a easy task. Now we have to pop() the element and according to the example **6** should be pop first. For this we will take Q2 and start enqueuing elements from Q1, we will be continuing enqueue elements till one element is left in the Q1. Now only **6** is left in Q1 and we dequeue it. Now we have to insert element **7** than we can enqueue this element in the Q2. Similarly we can perform push() and pop() operations for stack.

Time Complexity: All enqueue operations will take O(1) time, dequeue operations will also take O(1) time. So, total running time complexity of push() and pop() operations will take O(1).





**Solution:** Based on the algorithm **depth** given in the text book, we can find the depth of

all nodes of a tree T.

If the node is the root node then the depth of that node is 0. After that we can check the

depth for all the nodes using this concept that depth of child is 1 plus depth of the parent.

**Algorithm**

Algorithm depthAllNodes(T, v):

**if** T.isRoot(v) **then**

**return** 0

**else**

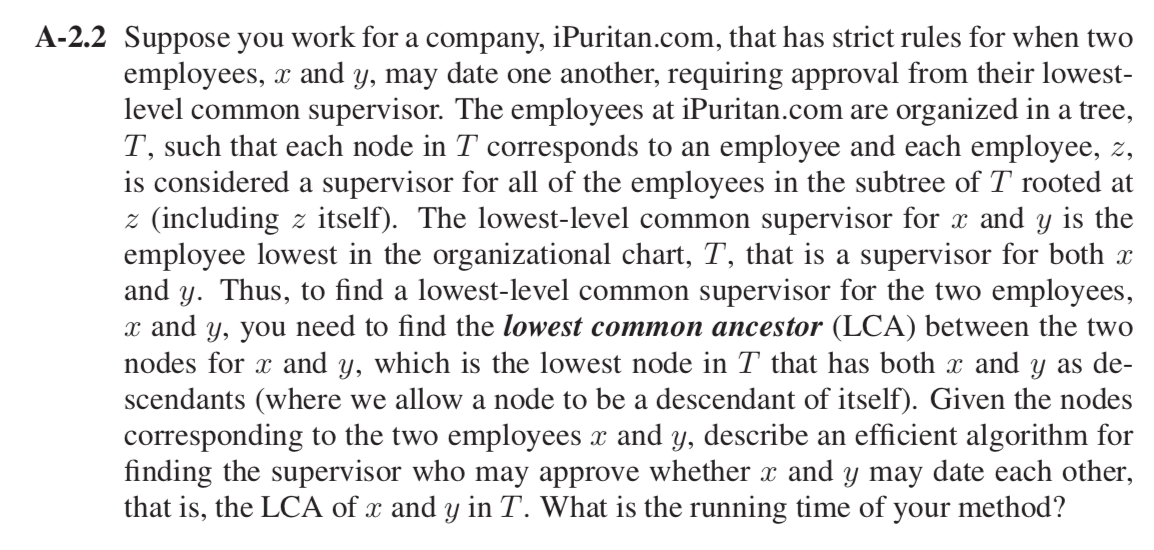
d = 0

**for** each child w of v **do**

**d =** 1 + depth(T, T.parent(v))

**return** d

**Time Complexity**: This algorithm will run in O(n) time. If-else statement run in constant time and for loop will run *n* times for *n* nodes as given number of nodes in a tree.



**Solution:** To find the lowest common ancestor between two nodes *x, y* of a tree *T*, First consider root node and start traversing it. If given values *x, y* of a node matches then root is the lowest common ancestor. If *x, y* doesn’t match, call recursively for lowest common ancestor algorithm for the left subtree and right subtree. If *x, y* present as the left child and right child then the parent node is lca, if *x, y* present in the left subtree then lca is from left subtree vice versa with right subtree.

Algorithm lca(T,x,y)

**if** (isRoot *null*)

**return** *null*

**if** (isRoot = x or isRoot =y)

**return** root

left ← lca(root.leftChild, x, y)

right ← lca(root.rightChild, x, y)

**if**(left and right)

**return** root

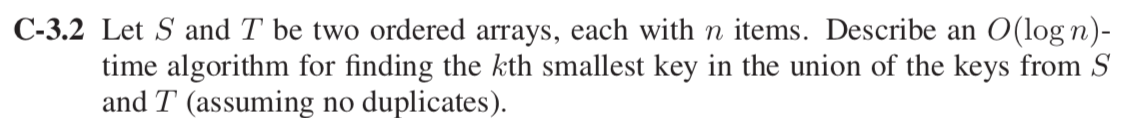
**else** **if**(left)

**return** left

**else**

**return** right

**Time Complexity**: Running time of this algorithm is O(n) because it is traversing the tree from root to all the nodes. Where n is the number of nodes present in the tree.



**Solution:** To find *k*th smallest key in the union of keys from S and T where S and T are ordered arrays, each with n items.

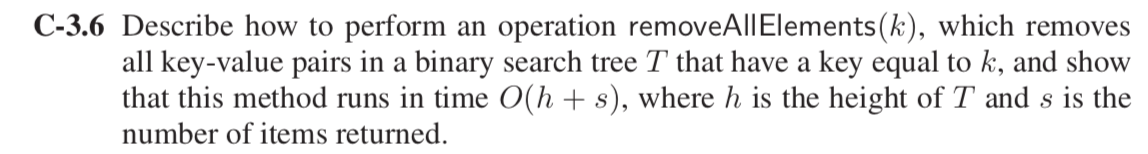
Firstly, examine the *k/2* element in the array list S. Now analyze largest element in the T which is less than *k/2* by binary search. Now adding the indices of these 2 elements:

* if sum of them is equal i.e. *k* then take maximum of two elements.
* If sum > k, binary search is performed to the right of S.
* If sum < k, binary search is performed to the left of S.

Now same operations are performed on T based on the largest element but less than the current element in S.

Now calculating total time complexity for the process, performing binary search for two arrays S and T will take O(log n) and O(log n) respectively.

That is *O*(log2 *n*). After solving this will give O(log n) running time complexity for the whole process.



**Solution:** To remove all thenodes from a binary search tree. Firstly, perform post order traversal on the tree and recursively call left subtree and right subtree and free the nodes respectively.

**Algorithm** removeAllElements(T, k)

***Input:*** A search key k for node of a binary search tree T.

***Output:*** Empty binary search tree

**if** T(k, T.root()) is *null*

**return** *null*

**else**

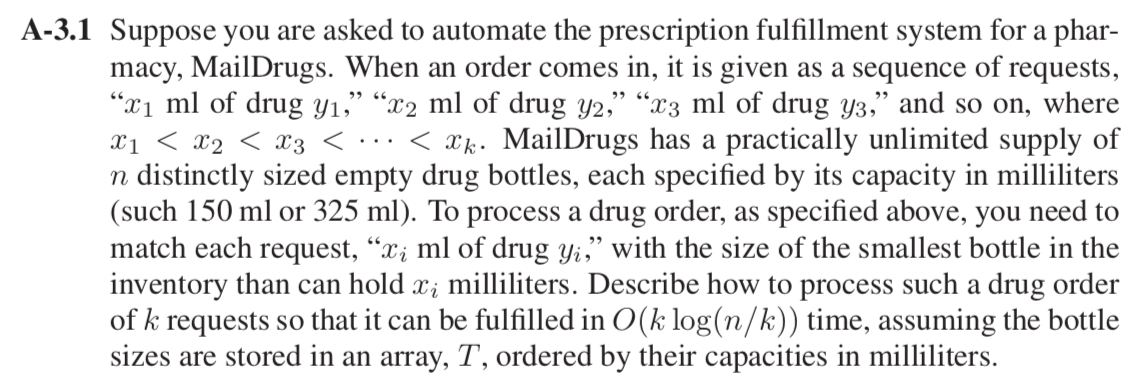
removeAllElements( binaryPostorder(T, T.leftChild(k)))

removeAllElements(binaryPostorder(T, T.rightChild(k)))

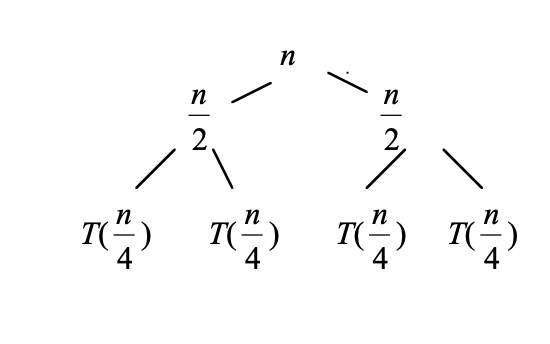
perform the “free” action for key (k) node

// (binaryPostorder algorithm is defined in our textbook)

**Time Complexity:** The number of nodes visited in a tree is proportional to height *h* of T. In binary search tree, remove method runs in O(*h*) time. To remove *s* elements from the binary search tree the algorithm will take O (*h + s*) time.



**Solution:** Appling divide and conquer method (Binary search) to store a drug in the smallest bottle in the inventory hold *xi* millimeters. As sorted in an array T by capacities the smallest bottle will be find in the left whereas the largest in the right.



The above approach will give the recurrence relation:

T(n) = T(n/2) + c

Solving this recurrence relation using iteration method

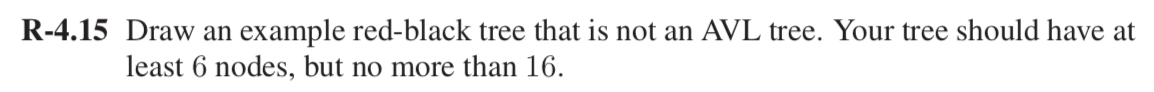
T(n) = (c + c + T(n/4))

After solving will give

T(n) = k\*c + T(n/2k)

T(n) = c log n

For n requests the time complexity of the above method is *c log n* but we have to process drug order of k requests will change the time complexity to O(*k log n).* Since the order is already sorted for xi it will take less time to search for k requests. After every xi  which changes the complexity to O(*k log n/k).*



**Solution:**

NILNI

NIL

NIL

NIL

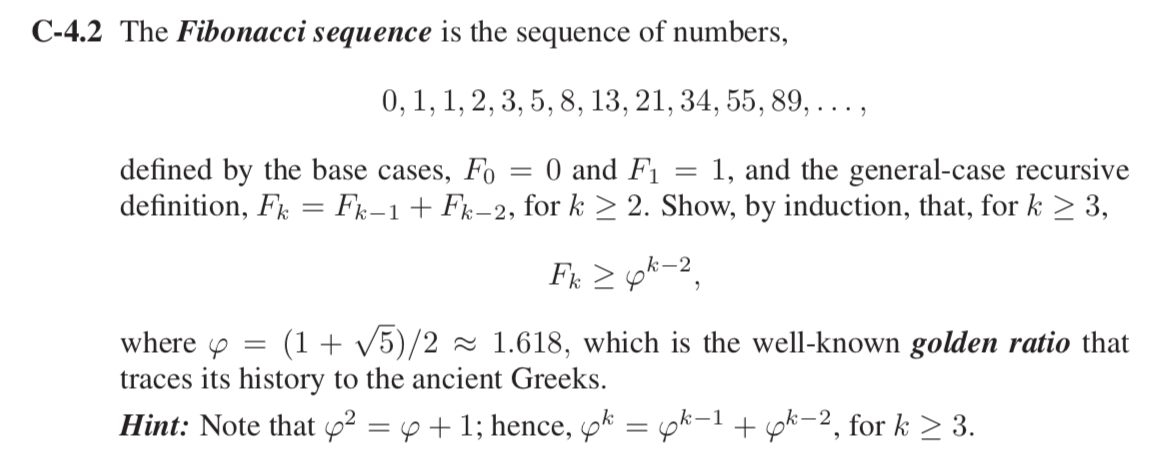
NIL

NIL

NIL

Above tree is a red-black tree consists of 6 nodes. It is not an AVL tree.

According to the property of AVL tree, for every internal node, v, in T, the heights of the children of v may differ by at most 1.



**Solution: Given**: F0 =0 and F1 = 1

And Fk = Fk−1 + Fk−2, for k ≥ 2

**Proof:** Fk ≥ φk−2,

Base Case: for k=2

F2 = F1 + F0= 1

Fk ≥ φk−2 => φ0 = 1

Therefore, true for k = 2

Base Case: for k=2

F3 = F2 + F1= 2

Fk ≥ φk−2 => φ

Fk > φ

Therefore, true for k = 3

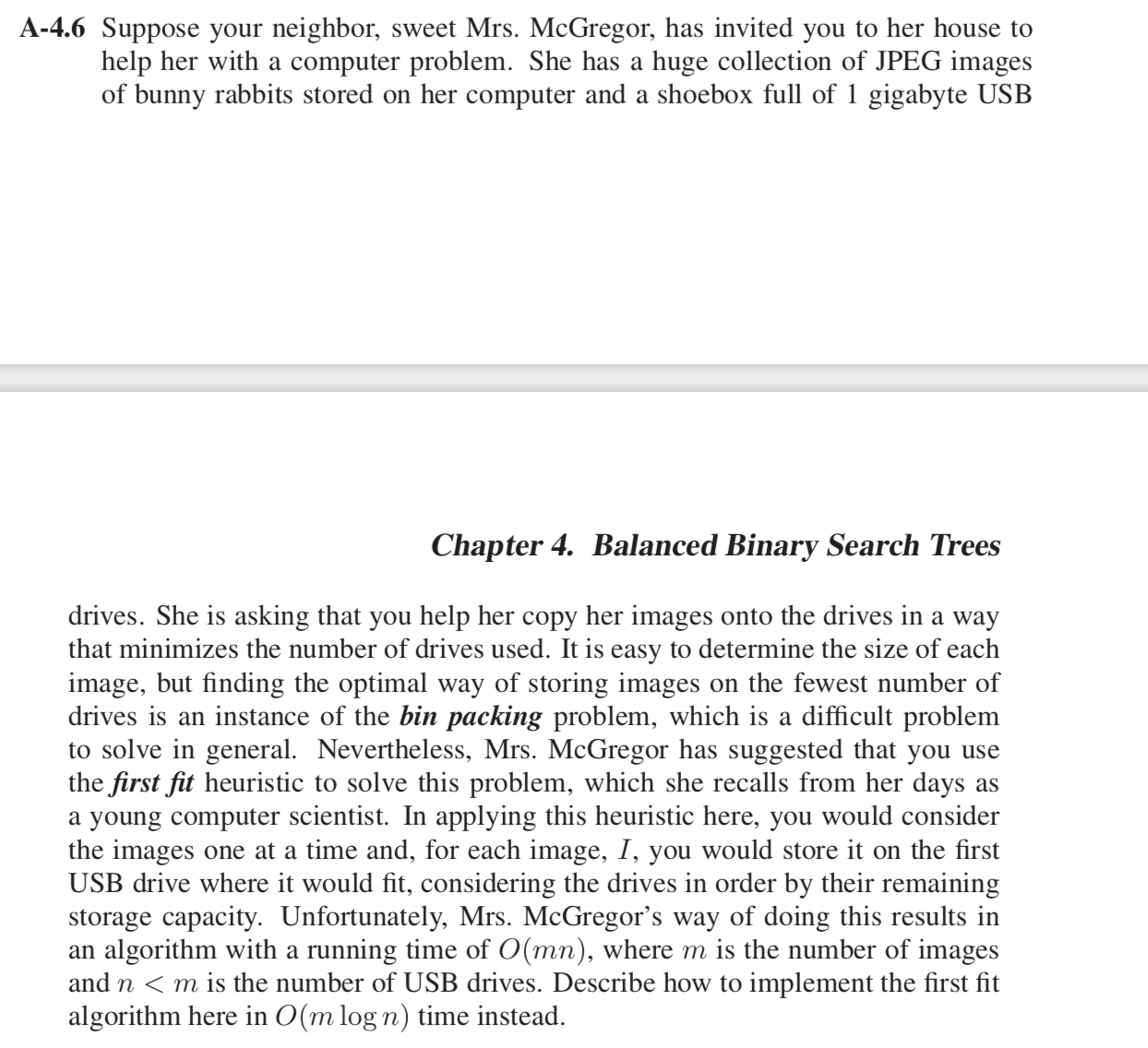
Now check for k-1

Fk−1 = Fk-2 + Fk−3 > φk−4 + φk−5 == = (Given: φ2 =φ+1 )

Fk−1 ≥ φk−3 (Assumption true for k-1)

So, it will be true for Fk ≥ φk−2

Hence Proved



**Solution:** Bin Packing Problem: objects of different volumes must be packed into a finite number of bins or containers each of volume *V* in a way that minimizes the number of bins used.

First fit heuristic: The algorithm processes the items in arbitrary order. For each item, it attempts to place the item in the first bin that can accommodate the item. If no bin is found, it opens a new bin and puts the item within the new bin.

With the help of balancing search trees (AVL tree) we can minimize the running time complexity of storing the images into the hard drives from *O(mn) to O(m log n).*

Where m is the number of images and n < m is the number of USB drives.

While performing insertion operation in AVL tree takes *O(log n)* for n items. And also checking inserting images in an order to check all the m drives if any space left in the previous bins (according to First fit) takes *m* running time. So total running time complexity for first fit is *O(m log n).*