Homework 2 Solution-Komal Wavhal

Table of Contents

Chapter 2

2.5.13

Describe how to implement a stack using two queues. What is the running time of the `push()` and `pop()` methods in this case?

Solution:

To implement a stack using 2 queues, we can use 1 queue to store the elements and the other one to use if for the reversing the order while performing the pop() operation.

- Implement 2 queues Q1 and Q2 for the implementation of push() operation
- Firstly, check if Q1 is empty, if it is, begin adding elements to it, such as 0,1,2
- For the pop() operation, remove the element according to the stack's LIFO, which mean – the element '2' should be popped first. To get this, we take the Q2 and start transferring elements from Q1 to Q2 and continue this until only 1 element remains in Q1 which is going to be '2'
- Now, only one element '2' is left in Q1, we dequeue it.
- When it's time to insert a new element, such as 3, we enqueue it in Q2
- Both the enqueue and dequeue operations take constant time, O(1). Therefore, the time complexity for both push() and pop() operations will also be O(1).

2.5.20

Give an $O(n)$-time algorithm for computing the depth of all the nodes of a tree T, where n is the number of nodes of T.

Solution:

We can use a Depth-First Search (DFS) approach to compute the depth of all nodes in the tree. The depth of a node is defined as the number of edges from the root node to the node.

The idea is to recursively traverse the tree, starting from the root, and compute the depth for each node by adding 1 to the depth of its parent.

- If the current node is the root, the depth is '0'
- For each child of the current node, we recursively compute its depth as the depth of the current node + 1

Pseudocode:

```
depth_Of_Tree_T(T,v, depth):
    if T.isRoot(v) then
        depth ← 0
    else
        for each child in T.children(v):
            depth_Of_Tree_T(T, child, depth + 1)
Compute_Depth_Of_All_Nodes(T):
    Root ← T.getRoot()
    Depth_Of_Tree_T(T, root, 0)
```

- The algorithm runs in O(n), where n is the number of nodes in the tree.

2.5.32

Suppose you work for a company, iPuritan.com, that has strict rules for when two employees, x and y, may date one another, requiring approval from their lowest-level common supervisor. The employees at iPuritan.com are organized in a tree, T, such that each node in T corresponds to an employee and each employee, z, is considered a supervisor for all of the employees in the subtree of T rooted at z (including z itself). The lowest-level common supervisor for x and y is the employee lowest in the organizational chart, T, that is a supervisor for both x and y. Thus, to find a lowest-level common supervisor for the two employees, x and y, you need to find the ***lowest common ancestor*** (LCA) between the two nodes for x and y, which is the lowest node in T that has both x and y as descendants (where we allow a node to be a descendant of itself). Given the nodes corresponding to the two employees x and y, describe an efficient algorithm for finding the supervisor who may approve whether x and y may date each other, that is, the LCA of x and y in T. What is the running time of your method?

Solution:

To determine the lowest common ancestor (LCA) of two nodes x and y in a tree T, begin by considering the root node and start traversing the tree. If either x or y matches the current node, then the current node is the LCA. If neither matches, recursively apply the LCA algorithm to the left and right subtrees. If x and y are found in the left and right subtrees, respectively, then the current

node is the LCA. If both x and y are located in the left subtree, the LCA is in the left subtree; similarly, if both are in the right subtree, the LCA is in the right subtree.

- We traverse the tree recursively, exploring both the left and right subtrees.
- When both subtrees return non-null nodes, the current node is the LCA. If one of the subtrees returns null, we propagate the non-null node up the recursion stack.

Pseudocode

LowestCommonAncestor(root, x, y):

    If root is null or root is x or root is y:

        Return root

    left ← LowestCommonAncestor(root.left, x, y)

    right ← LowestCommonAncestor(root.right, x, y)

    if left is not null and right is not null:

        return root

    return left if left is not null else right

- Time complexity for this will be O(n) where n is number of nodes in the tree.

3.6.15

Let S and T be two ordered arrays, each with n items. Describe an $O(\log\ n)$-time algorithm for finding the kth smallest key in the union of the keys from S and T (assuming no duplicates).

Solution:

To find the k-th smallest key in the union of the keys from two sorted arrays S and T, where both arrays have n elements, we can use a binary search approach. The idea is to reduce the size of the problem in each step, similar to how binary search works.

- Consider the middle element of both arrays S and T
- Compare the middle elements of the two arrays.
- Depending on the comparison, we discard half of the arrays in each step.
- Recursively search in the remaining array until we find the k-th element

```
findKthSmallest(S, T, k):
    if length of S is 0 then
        return T[k]
    if length of T is 0 then
        return S[k}
    midS ← length of S // 2
    midT ← length of T // 2

    if midS + midT is < k:
        if S[midS] is > T[midT] then
            return findKthSmallest(S, T[midT  + 1:], k – midT – 1)
        else
            return findKthSmallest(S[midS + 1:], T, k – midS -1)
    else
        if S[midS] is > T[midT] then
```

```
            return findKthSmallest(S[:midS], T, k)
        else
            return findKthSmallest(S, T[:midT], k)
```

- Initially if either of the arrays is empty, we return the k-th element of the other array, as the two arrays are sorted.
- If the sum of the middle indices midS + midT is less than k, we know that the k-th smallest element must be in the remaining part of the arrays after the current middle elements.
- We recursively call the function, adjusting the arrays and the value of k.
- The time complexity is O(log n), where n is the size of each array.

3.6.19

Describe how to perform an operation `removeAllElements`(k), which removes all key-value pairs in a binary search tree T that have a key equal to k, and show that this method runs in time $O(h + s)$, where h is the height of T and s is the number of items returned.

Solution:

To remove all nodes with the key k in a Binary Search Tree (BST), we can perform a recursive search to locate the nodes with the matching key and remove them

- Start by traversing the tree, starting from the root node. At each node, compare the current node's key with the target key k
- If the current node's key is equal to k, we remove that node and continue searching in both the left and right subtrees to remove any additional occurrences of k.
- If the key is less than k, we continue searching in the right subtree. If the key is greater than k, we continue searching in the left subtree.

- The recursion continues until all nodes with the key k are removed or no more nodes with key k are found.

Pseudocode

```
removeAllElements(rootNode, k):
    if rootNode is null then
        return rootNode
    while rootNode is not null and rootNode.key is k:
        rootNode ← remove(rootNode)
    if rootNode is not null and rootNode.key is < k:
        rootNode.rightChild ← removeAllElements(rootNode.rightChild, k)
    else if rootNode is not null and rootNode.key is > k:
        rootNode.leftChild ← removeAllElements(rootNode.leftChild, k)
    return rootNode
```

- Finding the node with the key k takes O(h) time, where h is the height of the tree. The removal of all occurrences of k involves traversing the tree recursively, and this will take O(s). So, total time complexity is O(h+s).

3.6.26

Suppose you are asked to automate the prescription fulfillment system for a pharmacy, MailDrugs. When an order comes in, it is given as a sequence of requests, "x_1 ml of drug y_1," "x_2 ml of drug y_2," "x_3 ml of drug y_3," and so on, where $x_1 < x_2 < x_3 < \cdots < x_k$. MailDrugs has a practically unlimited supply of n distinctly sized empty drug bottles, each specified by its capacity in milliliters (such 150 ml or 325 ml). To process a drug order, as specified above, you need to match each request, "x_i ml of drug y_i," with the size of the smallest bottle in the inventory than can hold x_i milliliters. Describe how to process such a drug order of k requests so that it can be fulfilled in $O(k \log(n/k))$ time, assuming the bottle sizes are stored in an array, T, ordered by their capacities in milliliters.

Solution:

- Since the bottle sizes are sorted in the array T, we can use binary search to quickly find the smallest bottle that can hold each requested amount of drug. We perform this search for each request and update the search space for the next request.
- We divide the array into smaller subarrays as we progress with each request. The array is already sorted, so once a suitable bottle for a request is found, we adjust the search space by narrowing down the range in T to search only the remaining possible bottles for the next request.
- The time complexity for searching the appropriate bottle for each request is O(log n), and since we make k such requests, the total complexity will be O(klog (n/k)) as the remaining search space shrinks with each request.

Pseudocode

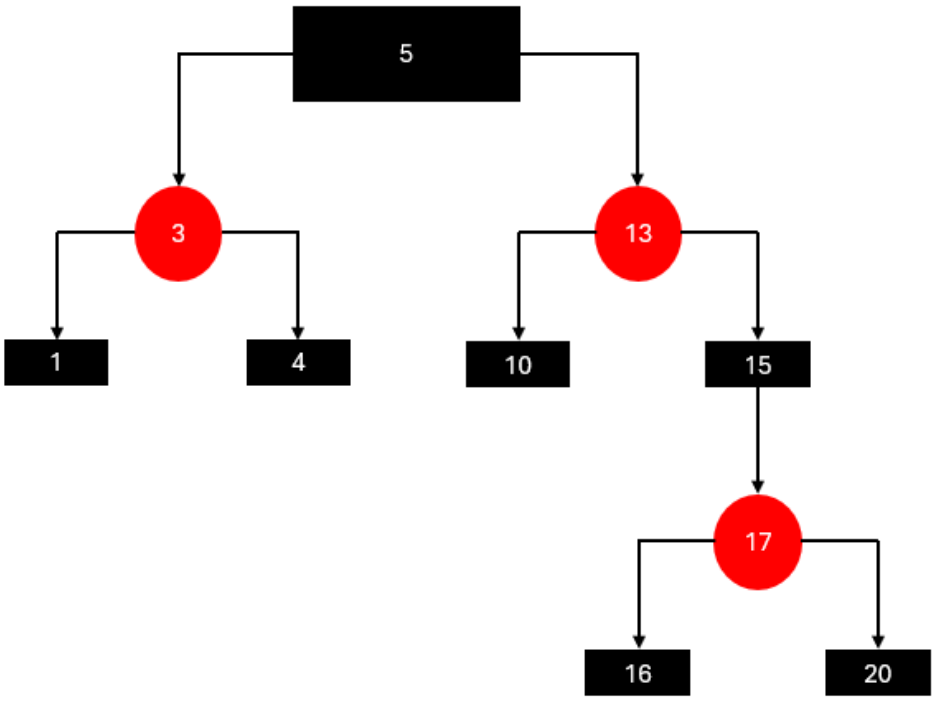
```
ProcessDrugOrder(requests, bottleSizes):
    matchedBottles ← []
    start ← 0
    end ← length of bottleSizes – 1
    for requests in requests:
        matchedBottleIndex ← BinarySearch(request, bottleSizes, start, end)
        matchedBottles.append(bottleSozes[matchedBottleIndex])
    if request is < requests[length of requests //2] then
        end ← matchedBottleIndex – 1
    else
        start ← matchedBottleIndex + 1
    return matchedBottles
```

```
BinarySearch(request, array, start, end)
    while start is < or equal to end:
        mid ← (start + end) // 2
        if array[mid] is request then
            return mid
        else if array[mid] is < request then
            start ← mid + 1
        else
            end ← end – 1
    return start
```

- We initialize the start and end pointers to cover the entire array of bottle sizes. For each request, we perform a binary search to find the smallest bottle that can satisfy the request.
- The BinarySearch function returns the index of the smallest bottle in the array T that can hold the requested amount. After finding a suitable bottle, we update the search space (start and end) for the next request based on whether the current request is smaller or larger than the midpoint of the array.
- The time complexity for processing all k requests is O(klog(n/k))

4.7.15

Draw an example red-black tree that is not an AVL tree. Your tree should have at least 6 nodes, but no more than 16.



4.7.22

The ***Fibonacci sequence*** is the sequence of numbers,

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots,$$

defined by the base cases, $F_0 = 0$ and $F_1 = 1$, and the general-case recursive definition, $F_k = F_{k-1} + F_{k-2}$, for $k \geq 2$. Show, by induction, that, for $k \geq 3$,

$$F_k \geq \phi^{k-2},$$

where $\phi = \left(1 + \sqrt{5}\right)/2 \approx 1.618$, which is the well-known ***golden ratio*** that traces its history to the ancient Greeks.

Hint: Note that $\phi^2 = \phi + 1$; hence, $\phi^k = \phi^{k-1} + \phi^{k-2}$, for $k \geq 3$.

Solution:

Base Case for k=2:

$F_2 = F_1 + F_0 = 1$

Since $F_k \geq \varphi^{k-2}$ we find that $F_2 = 1 \geq \varphi^0 = 1$

Therefore, k=2

Base Case for k=3:

$F_3 = F_2 + F_1 = 2$

$F_k \geq \varphi^{k-2} \quad => \quad \varphi^0 \quad = 1$

Therefore, k=3

Now check for k-1

$F_{k-1} = F_{k-2} + F_{k-3} > \varphi^{k-4} + \varphi^{k-5} = \varphi^{k-4}\left(1 + \frac{1}{\varphi}\right) = \varphi^{k-4}(\frac{\varphi+1}{\varphi}) = \varphi^{k-3}$ (Given: $\varphi 2 = \varphi + 1$)

$F_{k-1} \geq \varphi^{k-3}$ (Assumption true for k-1)

So, it will be true for $Fk \geq \varphi k-2$

Hence, we can say that, for k >= 3, $F_k >= \varphi^{k-2}$

Therefore, using mathematical induction, we conclude that for $k \geq 3$, $F_k \geq \varphi^{k-2}$.

4.7.47

Suppose your neighbor, sweet Mrs. McGregor, has invited you to her house to help her with a computer problem. She has a huge collection of JPEG images of bunny rabbits stored on her computer and a shoebox full of 1 gigabyte USB drives. She is asking that you help her copy her images onto the drives in a way that minimizes the number of drives used. It is easy to determine the size of each image, but finding the optimal way of storing images on the fewest number of drives is an instance of the ***bin packing*** problem, which is a difficult problem to solve in general. Nevertheless, Mrs. McGregor has suggested that you use the ***first fit*** heuristic to solve this problem, which she recalls from her days as a young computer scientist. In applying this heuristic here, you would consider the images one at a time and, 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. Unfortunately, Mrs. McGregor's way of doing this results in an algorithm with a running time of $O(mn)$, where m is the number of images and $n < m$ is the number of USB drives. Describe how to implement the first fit algorithm here in $O(m \log n)$ time instead.

Solution:

- The BST stores available USB drive capacities. Each node in the tree represents the remaining space in a USB drive. The BST allows efficient search, insertion, and deletion operations in O(log n) time.
- For each image I, search for the smallest available USB drive that can accommodate I. If found, insert the image into the drive and update the BST. If no suitable USB drive exists, allocate a new USB drive and insert it into the BST.

```
FirstFitOptimized(images, usbDrives):
    T ← BalancedBST()
    T.insert(1GB)
    for image in images:
        usbDrive ← T.findMinGreaterThan(image.size)
        if usbDrive is null then
            T.insert(1GB)
            usbDrive ← T.findMinGreaterThan(image, size)
        placeImageDrive(image, usbDrive)
```

```
            T.remove(usbDrive.capacity)
            usbDrive.capacity ← usbDrive.capacity – image.size
            if usbDrive.capacity is > 0 then
                  T.insert(usbDrive.capacity)
      return usbDrive
```

Total complexity for mmm images: O(m log n) which includes suitable USB drive and Inserting or updating the BST time complexities