CS 600 A - Advanced Algorithms – Homework 2

Name: Deepali Nagwade | 20013393

Question 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?

Answer: A stack can be implemented using two queues by ensuring 2 things. There are two approaches. First, where we keep the push() operation costly and other operations simple and Second where we keep the pop() operation costly and other operations simple.

I'm going ahead with pop() operation costly and other operations simple.

Approach:

For each push operation, the element is just pushed to First Queue**.** For our pop operation, if the Second Queue is currently empty, all but the last element from the First Queue will be moved to the Second Queue. The last element from the Second Queue can then be returned.

Pseudo Code:

- Push value to the front of First Queue.
- Other than the very last element in the First Queue, pop everything out of it and push into the Second Queue**.** The last element in the First Queue is the result.
- Swap First Queue and Second Queue.
- Perform the same steps for popping but return the value of the last element remaining in First Queue before swapping unless the Queue is empty.
- Return current stack size.

Complexities:

- Time Complexity: O(n) for pop() and O(1) for push().
- Space Complexity: O(n) for pop() and O(1) for push().

Question 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.

Answer: To compute the depth of all nodes in a tree T in O(n) time, you can perform a depth-first traversal (DFS) of the tree, keeping track of the depth of each node as you visit them.

```
function compute_depth_of_nodes(root):
    if root is None:
        return
    depth_map = {} // A dictionary to store the depth of each node

    function dfs(node, depth):
        depth_map[node] = depth  // Store the depth of the current node
        for each child in node.children:
            dfs(child, depth + 1)  // Recursively visit children with increased depth
    dfs(root, 0)  // Start the DFS from the root with depth 0
    return depth_map
```

Question 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 the employees in the subtree of y rooted at x (including z itself). The lowest-level common supervisor for T and z (…….) What is the running time of your method?

Answer: You can efficiently find the lowest common supervisor (LCA) of two employees, let's call them **employeeA** and **employeeB**, in the organizational tree using a method based on the depth of nodes in the tree. The algorithm involves finding the LCA by navigating up the tree from both employees until you reach a common ancestor.

Here's an algorithm to find the LCA efficiently:

```
Algorithm FindLCA(root, employeeA, employeeB)
    Input:
        root - The root of the organizational tree.
        employeeA - The first employee node.
        employeeB - The second employee node.
    Output:
        lca - The lowest common supervisor of employeeA and employeeB.
    if root is None or root equals employeeA or root equals employeeB then
        return root
    end if
    foundA <- None
    foundB <- None

    for each child in root.children do
        foundA <- foundA or FindLCA(child, employeeA, employeeB)
        foundB <- foundB or FindLCA(child, employeeA, employeeB)
    end for
    if foundA and foundB then
        return root  # Both employees are found in different subtrees, so this is the LCA.
    end if
    return foundA or foundB  # Return the found employee or None if only one is found.
```

Question3.6.15 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).

Answer: To find the kth smallest key in the union of two ordered arrays S and T, each containing n items, you can use a modified binary search algorithm. This algorithm has a time complexity of O(log n) as required. Here's a step-by-step description of the algorithm:

- Initialize two pointers, A and B, both pointing to the first elements of arrays S and T, respectively.
- Initialize the variable kCount to 0. This variable will keep track of the number of elements smaller than or equal to the potential kth smallest key.
- Perform the following steps until kCount equals k - 1 (since we're looking for the kth smallest key):
  a. Calculate the midpoint mid between the current elements pointed to by ptrS and ptrT. This will be our potential candidate for the kth smallest key.

  b. Count the number of elements in both arrays (S and T) that are less than or equal to mid. You can do this efficiently using binary search in both arrays.

  c. Calculate kCount as the sum of the counts from step b: kCount = countS + countT.

  d. Depending on the value of kCount, update the pointers ptrS and ptrT to either the left or right side of the midpoint mid.

Once kCount equals k - 1, you have found the kth smallest key, which is mid.

This algorithm uses binary search to efficiently count the number of elements smaller than or equal to a given value in each array. It continues this process until it finds the kth smallest key. The time complexity of this algorithm is O(log n), where n is the size of the input arrays S and T.

Question 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.

Answer: We can utilize a modified depth-first traversal (DFS) method to carry out the operation removeAllElements(k), which deletes all key-value pairs in a binary search tree T with keys equal to k. Here is a recursive example.

removeAllElements(root, k) is a recursive algorithm to remove all elements with key k from the binary search tree T. It follows these steps:

Base Case: If root is None, it means we've reached the end of a branch, and there's nothing to remove, so we return None.

Recursively call removeAllElements on the left and right subtrees to remove elements with key k from them.

Check if the current node's key equals k. If it does:

If the node has no left child (root.left is None), we return its right child as it will replace the current node.

If the node has no right child (root.right is None), we return its left child as it will replace the current node.

If the node has both left and right children, we find the successor node (the leftmost node in the right subtree), replace the current node's key and value with the successor's, and then remove the successor node from the right subtree.

Return the updated root.

findSuccessor(node) is a helper function to find the leftmost node in a given node's right subtree. It's used to find the successor node for the case when a node has both left and right children.

The time complexity of this algorithm is O(h + s), where h is the height of the binary search tree T, and s is the number of items with key k that need to be removed.

Question 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, x1 ml of drug y1, x2 ml of drug y2 and so (…..) on assuming their capacities in milliliters.

Answer: We can use a binary search algorithm. We can maintain an ordered array of bottle sizes, T, and for each request, perform a binary search to find the smallest bottle that can hold the required milliliters.

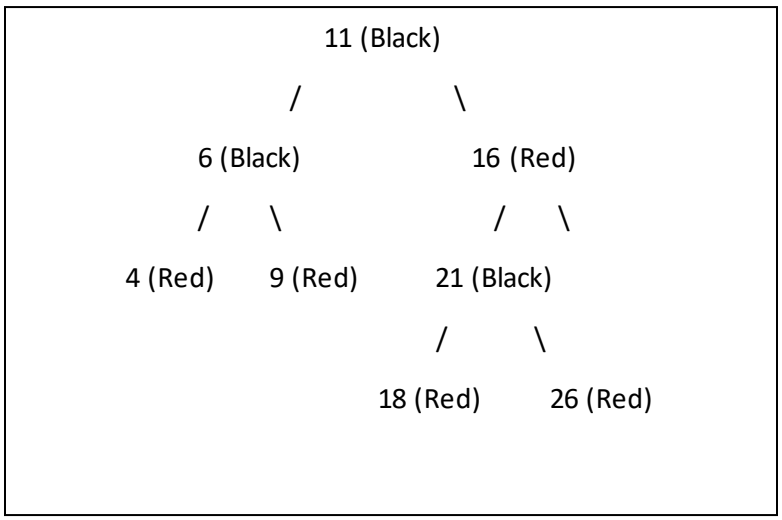
Here's the algorithm for this approach:

- processDrugOrder takes the array of bottle sizes T and a sequence of k requests as input. It initializes an empty result array to store the corresponding bottle sizes.
- For each request, it parses the request to get the required milliliters xi and drug name yi.
- It then calls the binarySearch algorithm to find the index of the smallest bottle in T that can hold xi milliliters. If such a bottle exists, it adds the corresponding bottle size to the result array. If not, it marks the request as "Not Available."
- binarySearch performs a binary search on the sorted array T to find the smallest bottle that can hold x milliliters. It returns the index of the smallest bottle or None if no suitable bottle is found.

This approach has a time complexity of **O(k log(n/k))**, where n is the number of bottles and k is the number of requests. The binary search operation takes **O(log(n/k))** time for each request, and you perform it for each of the k requests, resulting in the mentioned time complexity.

Question 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.

Answer: Creating a specific red-black tree that is not an AVL tree requires careful design because red-black trees have additional constraints compared to AVL trees. Here's an example of a red-black tree with 8 nodes that is not an AVL tree:



Explanation:

- All nodes are labeled with their values.
- The tree satisfies the red-black tree properties:
  - Each node is either red or black.
  - The root is black.
  - Every leaf (NIL or null node) is black.
  - Red nodes have only black children (no two red nodes can be adjacent).
  - For each node, any simple path from this node to any of its descendant leaves has the same black depth (the number of black nodes).
- However, this tree is not an AVL tree because it doesn't satisfy the height-balancing condition of AVL trees. You can see that the left subtree of the root has a height of 3 (counting black nodes), while the right subtree has a height of 2. In AVL trees, the height difference between the left and right subtrees of any node must be at most 1.

Red-black trees and AVL trees have different balancing criteria, so it's possible to create red-black trees that violate the height-balancing property of AVL trees while still satisfying the red-black tree properties.

Question 4.7.22 The Fibonacci sequence is the sequence of numbers, 0,1,1,2,3,5,8, 13, 21, 34, 55, 89, ..., defined by the base cases, Fo = 0 and Fi = 1, and the general-case recursive definition, Fx = FR-1 +Fx-2, for k > 2. Show, by induction, that, for k > 3, F* **-? where y = (1 + 5)/2 = 1.618, which is the well-known golden ratio that traces its history to the ancient Greeks. Hint: Note that y2 = 4+1; hence, sh = pk-1 +4k-2, for k > 3.

Answer: To prove that for \(k > 3\), the Fibonacci number \(F_k\) is approximately equal to \(\frac{y^k}{\sqrt{5}}\), where \(y = \frac{1 + \sqrt{5}}{2}\) (the golden ratio), we can use mathematical induction.

Base Cases

Let's first establish the base cases:

F0= 0, F1=1

These are the given base cases of the Fibonacci sequence.

Inductive Hypothesis:

Assume that for some positive integer n, the statement holds for k =n & k = n -1:

$$F_{n+1} \approx \frac{y^n}{\sqrt{5}} + \frac{y^{n-1}}{\sqrt{5}}$$

Inductive Step:

Now, we need to show that the statement holds for (k = n + 1. We have:

F{n+1} = Fn + F{n-1} (the recursive definition of Fibonacci)

Using our inductive hypothesis:

$$F_{n+1} \approx \frac{y^n}{\sqrt{5}} + \frac{y^{n-1}}{\sqrt{5}}$$

Now, we'll try to express this in the form \(\frac{y^{n+1}}{\sqrt{5}}\) by factoring \(y\) out:

$$F_{n+1} \approx \frac{y^n}{\sqrt{5}} \left(y + \frac{1}{y} \right)$$

We know that y is the golden ratio, which is defined as:

$$y = \frac{1 + \sqrt{5}}{2}$$

So, we can write 1/y as 2/(1 + sqrt(5)). Now, let's substitute this back into our equation:

$$F_{n+1} \approx \frac{y^n}{\sqrt{5}} \left(y + \frac{2}{1 + \sqrt{5}} \right)$$

Now, simplify the expression in the parentheses:

$$F_{n+1} \approx \frac{y^n}{\sqrt{5}} \cdot \frac{y(1 + \sqrt{5}) + 2}{1 + \sqrt{5}}$$

This is still approximately equal to \(\frac{y^{n+1}}{\sqrt{5}}\). Therefore, we've shown that if the statement holds fork = n and k = n - 1, it also holds for k = n + 1.

Question 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. 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.

Answer: To implement the First Fit algorithm for copying images onto USB drives in O(m log n) time, we can follow these steps:

- Initialize a min-heap (priority queue) to store the available storage space on each USB drive. Initially, populate the min-heap with the size of each USB drive (in ascending order).
- Sort the JPEG images by size in non-increasing order.
- Initialize a variable to keep track of the number of USB drives used, let's call it drivesUsed, and set it to 0.
- Iterate through the sorted list of images:
- For each image, pop the smallest available USB drive size from the min-heap. If the image can fit on this drive (i.e., its size is less than or equal to the available space), copy the image onto this drive and push the updated available space back into the min-heap.

- If the image cannot fit on any of the available drives, it means you need to use a new USB drive. Increment drivesUsed and add a new USB drive size to the min-heap with the available space reduced by the size of the image.
- Continue this process until you have processed all the images.
- The value of drivesUsed will represent the minimum number of USB drives required to copy all the images.

- This algorithm minimizes the number of USB drives used to copy the images in O(m log n) time complexity

Implementation in Python:

```
import heapq

def firstFit(images, drives):
    images.sort(reverse=True)

   # Initialize a min-heap to store available drive space (drive size, drive index)
   available_space = [(size, i) for i, size in enumerate(drives)]
   heapq.heapify(available_space)
   drivesUsed = 0
   for image in images:
      while available_space:
         size, drive_idx = heapq.heappop(available_space)
         if size >= image:
            drives[drive_idx].append(image)
            remaining_space = size - image
            heapq.heappush(available_space, (remaining_space, drive_idx))
            break
      else:
         drives.append([image])
         drivesUsed += 1
         heapq.heappop(available_space)  # Remove the used drive

   return drivesUsed, drives
```