CS 600 Advanced Algorithms

Sri Dhanush Reddy Kondapalli

CWID: 10476150

Homework 2

1 C-2.5.13 Implement a stack using two queues. What is the running time of the push() and pop() methods in this case?

A stack may be implemented via the use of two queues. The first queue, Queue A, will handle the push() operation. When push() is called, the element is en-queued into Queue A. A counter variable for Queue A is then increased to track the addition of this new element. This operation takes O(n) time. In order to pop() from the stack, the second queue comes into play, which is known as Queue B. Queue B will be used to store all but one of the elements from Queue A, which are removed via the de-queue operation and en-queued into Queue B. After each individual de-queue, the aforementioned counter variable is decreased to track the current location before being increased once more after the successful en-queue into Queue B. This tracks a balancing to ensure that the correct number of elements are being transported.Once all but one of the elements from Queue A have made it into Queue B, then the final element of Queue A is de-queued. Queue B then becomes the new primary queue (in essence replacing Queue A), and the final de-queued element is returned. This methodology uses two queues, with one basically serving as a temporary container for the other. It operates in $O(n^2)$ time as $(n-1)*(n-1)+1$ is yielded from $n-1$ de-queues from Queue A, $n-1$ en-queues into Queue B, and the finalized return statement.

2 C-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.

Desciption: This algorithm calculates the depths of all nodes in a tree. It uses a dictionary passed as a parameter, depths, to store each node's depth. This algorithm is organized in a similar manner to that of a pre-order traversal algorithm, which is also defined in Chapter 2.4. Here, the "visit" action is replaced

with a setting of the dictionary's entry for that node to the current depth. This algorithm is originally called on the root of the tree with a depth of 0, and the depth is incremented during the recursive call as the children of v are visited (indicating an increase of depth by 1). This algorithm makes use of children(v), which is defined in Section 2.4 of the textbook and returns a set containing the children of node v.
Pseudocode:

```
Algorithm findAllDepths(T, v, d, depths):
    Input: Node v of depth d located in tree t and
           dictionary depths of key-value pair node:depth
    Output: Updates the depths dictionary with the
            depth of each node

    depths[v] <- d

    child_nodes <- T.children(v)

    for i <- 0 to child_nodes.size() do:
        findAllDepths(T, child_nodes[i], d + 1, depths)

    return depths
```

Running Time: This algorithm has a running time of O(n) as at a maximum it must visit every node of the tree.

3 A-2.5.32 Find the lowest common ancestor (LCA) between the two nodes

The Lowest Common Ancestor (LCA) problem may be solved by examining the parents of each of the two nodes all the way up to the root. This may be accomplished through some uses of the parent(v) function as described in Chapter 2.4 of the textbook. Each node x and y will get a list that corresponds to its collection of ancestors that leads all the way back to the root. The first parent of x or y will be stored in the first index of these lists, the second in the second and so on until the root is reached. Once both lists have been filled, the size of them will be compared. This is important as it has to do with the depth locations of x and y on the tree, and in order to ensure a proper LCA both lists must start off at the same point. Therefore, the size difference d of the two lists will be examined. The larger one will shrunken by having its first d elements removed (e.g., using List[d:] instead of List) to ensure an equal level of depth. From here, the heads of both lists will be removed and then compare to each-other to establish an LCA. In the worst case scenario, this will require an iteration throughout the entire (equal) length of both lists, resulting in a time of O(n).

4 C-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).

This algorithm uses the binary search model as a basis with some modifications. Due to this, it has a running time of $O(\log n)$. It operates based on using the midpoints of the two arrays. The midpoint will be taken by dividing n by 2, and then accessing the $\frac{n}{2}$ index of each array. The values of these two midpoints will be added. From here, a number of conditionals are evaluated:

- if $S[\frac{n}{2}] + T[\frac{n}{2}]k$:
  - if $S[\frac{n}{2}]$ is larger than $T[\frac{n}{2}]$, then recursively call the algorithm using the whole array S, a subarray of T starting at index $\frac{n}{2} + 1$, and a new k value of $k - \frac{n}{2} - 1$
  - if $T[\frac{n}{2}]$ is larger than $S[\frac{n}{2}]$, then recursively call the algorithm using a sub-array of S starting at index $\frac{n}{2} + 1$. the whole array T, and a new k value of $k - \frac{n}{2} + 1$
- if $S[\frac{n}{2}] + T[\frac{n}{2}] \geq k$:
  - if $S[\frac{n}{2}]$ is larger than $T[\frac{n}{2}]$, then recursively call the algorithm using a sub-array of S starting at index $\frac{n}{2} + 1$, the whole array T, and k
  - if $T[\frac{n}{2}]$ is larger than $S[\frac{n}{2}]$, then recursively call the algorithm using the whole array S, a subarray of T starting at index $\frac{n}{2} + 1$, and k

5 C-3.6.19 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 $O(h + s)$

This algorithm consists of two parts. The first of which is the finding of the nodes that have a key equal to k, as in order to delete nodes we must know which exact ones we are deleting. This functionality may be achieved by using an algorithm that is similar to the **TreeSearch()** method that is defined in Figure 3.2.4 of the textbook. Whereas the original **TreeSearch()** returns only a single node, this modified algorithm will instead be capable of returning a list of nodes. Due to the governing principle of binary search trees, nodes with

the same key values will be found adjacent to each other in direct parent-child relationships. This means that it will also have the same running time of the original **TreeSearch()** algorithm, O(h), as once one node with the desired key is found the entire cluster of matching-key nodes have been located. In the context of this problem, the length of the node array returned by this algorithm is s.

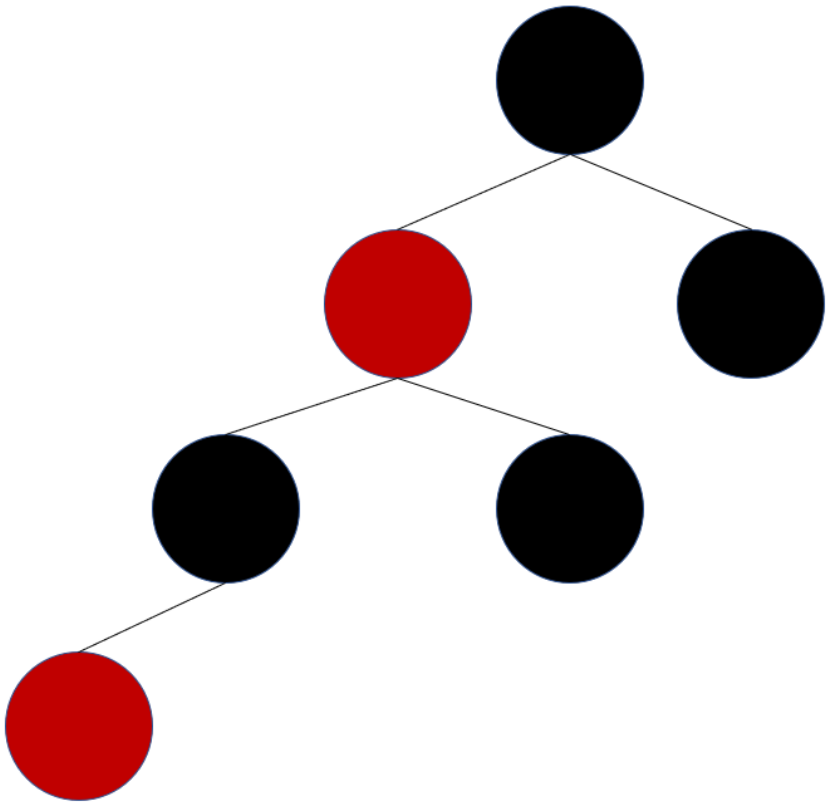
Using the the array of nodes that we previously found, we may now delete each of the nodes stored within. This is accomplished through the use of the **remove()** method defined in Section 3.2 of the textbook. This method has a running time of O(1) for every node visited, which in the worst case is proportional to the height of the tree O(h). Following the previous **TreeSearch()** reasoning, there are s nodes in parent-child relationships to be removed, meaning that the actual height of the sub-tree that **remove()** will be invoked upon is s. Therefore, in the worst case **s remove()** invocations will take O(s). This algorithm's total running time, $O(h+s)$, is derived by adding the time required to find all the matching nodes to the time required to actually remove them.

6 A-3.6.26 Order of k can be fulfilled in $O(k \log(\frac{n}{k}))$ time

This problem may be solved through the use of a binary search tree. The tree's key scheme would be based on the sizes of the bottles available. A search of this tree would therefore take $O(\log n)$ in the worst-case scenario. Due to the fact that all of the requests are ordered by increasing size, this helps improve the efficiency of the binary search. This is because every successfully found bottle helps narrow down the subtree of where subsequent bottles may be found (going in the increasing-size order means that the larger bottles will always be on the right of the tree when compared to the smaller ones. This therefore reduces the time needed to search the tree to be $O(\log \frac{n}{k})$, for as k increases the sub-tree becomes increasingly narrow. Repeat this process for all k requests, and the total running time is $O(k \log \frac{n}{k})$.

7 R-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.

An example of a tree that is a red-black tree but is not an AVL tree would be a tree wherein the difference in height between the left and right sub-trees is greater than 1. An example of this may be the following:



8 C-4.7.22 For Fibonacci sequence show, by induction, that, for $k \geq 3$, $F_k \geq \phi^{k-2}$

Base Case: For $n = 2$, $\phi^{2-2} = \phi^0 = 1$ and $F_2 = F_0 + F_1 = 0 + 1 = 1 \rightarrow 1 = 1\checkmark$
Inductive Hypothesis: There exists some $k_1, k_2 \geq 3$ such that
$F_k = F_{k_1} + F_{k_2} \geq \phi^{k_1-2} + \phi^{k_2-2}$.
Inductive Step:

$$F_{k_2+1} \geq \phi^{(k_2+1)-1} = \phi^{k_2} \rightarrow F_{k_1} + F_{k_2}$$
$$\phi^{k_2-1} = \phi^{k_1-2} + \phi^{k_2-2}$$

Substitution of the inductive hypothesis shows that $F_{k_1} \geq \phi^{k_1-2}$ and
$F_{k_2} \geq \phi^{k_2-2}$ holds true when $k > 2$ and ϕ evaluates to $(\frac{(1+\sqrt{5})}{2})$

9 A-4.7.47 Implement the first fit algorithm here in $O(m \log n)$ time

A self-balancing binary tree may be used to solve this problem. In this tree, the key scheme is determined by the remaining storage space of each USB drive. M images are iterated over. For each image I, a storage location is found by comparing I's size with the space remaining on the USBs (indicated via the keys of the tree). If a USB with enough space is found, then that image is assigned to that USB drive. However, if there exists no capable USB drive, then another node must be added into the tree. The cost for these 4 search and insertion processes as well as their associated re-balancing cost is $O(\log n)$. Multiplied by the total amount of images m, this yields and overall running time of $O(m \log n)$.