CS 600 Advanced Algorithms

Homework 2 Solutions

Contents

1 [C-2.8] Index-based list with $\mathcal{O}(1)$ inserts and removals at the ends

We keep track of the array's size, capacity, rank 0 index and rank n index. When adding or removing an element from either end, we use the appropriate index pointer to get to that element and update. We handle for possible wrap arounds using the capacity. Note that the array index of an element at rank k is simply $k + rank_0 \mod capacity$. The pseudocode is shown below,

Algorithm Insert at rank 0

Input: An array A and an element to insert e

if $size = capacity$ **then**
    Overflow Error
$A[rank_0] \leftarrow e$
if $rank_0 > 0$ **then**
    $rank_0 \leftarrow rank_0 - 1$
else
    $rank_0 \leftarrow capacity - 1$
$size \leftarrow size + 1$

Algorithm Remove element at rank 0

Input: An array A
Output: The element e removed from A

if $size = 0$ **then**
    List Empty Error
$rank_0 \leftarrow (rank_0 + 1) \mod capacity$
$e \leftarrow A[rank_0]$
$size \leftarrow size - 1$ **return** e

Algorithm Insert element at the end

Input: An array A and an element e

if $size = capacity$ **then**
    Overflow Error
$rank_n \leftarrow (rank_n + 1) \mod capacity$
$A[rank_n] \leftarrow e$
$size \leftarrow size + 1$

Algorithm Remove element at the end

Input: An array A
Output: An element e removed from A

if $size = 0$ **then**
    List Empty Error
$e \leftarrow A[rank_n]$
if $rank_n = 0$ **then**
    $rank_n \leftarrow capacity - 1$
else
    $rank_n \leftarrow rank_n - 1$
$size \leftarrow size - 1$ **return** e

Algorithm Get an element

Input: An array A, and a non-negative index i
Output: The i^{th} element of the array

if $size = 0$ **then**
    List Empty Error
return $A[(i + rank_0) \mod capacity]$

2 [C-2.20] Level Order Traversal of a Binary Tree

We can use a queue to perform a level order traversal of a binary tree in linear time. We start by adding the root of the tree to the queue. To traverse, we first dequeue a node v, and then enqueue its left and right child if it is an internal node. We repeat this until the queue is empty, in which case we would have traversed through the entire binary tree.

Algorithm Level Order Traversal of a Binary Tree

Input: A Binary Tree, T with a level numbering function
Output: A sequence of nodes with their level numbers

$Q.enqueue(T.root())$
while Q is not empty **do**
    $v \leftarrow Q.dequeue()$
    if $T.isInternal(v)$ **then**
        $Q.enqueue(T.leftChild(v))$
        $Q.enqueue(T.rightChild(v))$

We access each node exactly once. The running time of this algorithm is $\mathcal{O}(n)$.

3 [A-2.2] Lowest Common Ancestor

We recurse down the tree from its root and look for a subtree that contains both employees in it as children.

Algorithm LCA(T, x, y)

Input: A binary tree T and two nodes x and y.
Output: The lowest common ancestor of x and y.

if $T.isExternal(T.root)$ **then return** $T.root$
if $T.isRoot(x)$ **or** $T.isRoot(y)$ **then return** $T.root$

$left \leftarrow$ LCA$(T.root.leftChild(), x, y)$
$right \leftarrow$ LCA$(T.root.rightChild(), x, y)$

if $T.isExternal(left)$ **then return** $right$
if $T.isExternal(right)$ **then return** $left$
return $T.root$

This algorithm runs in $\mathcal{O}(h)$ where h is the height of tree since in the worst case, we would need to recurse all the way down the tree to find the LCA of x and y.

4 [R-3.6] Find element with smallest key in a binary search tree

Here we need to find the element at the left most internal node of the tree. We start at the root and traverse the left children until we reach an external node.

Algorithm Element with smallest key in a BST

Input: A binary search tree, T
Output: The node containing the smallest element in T

$v \leftarrow T.root.leftChild()$
while $T.isInternal(v)$ **do**
    $v \leftarrow T.leftChild(v)$
return v

The running time of this algorithm is $\mathcal{O}(h)$ where h is the height of the tree.

5 [C-3.3] The findAllElements(k) operation

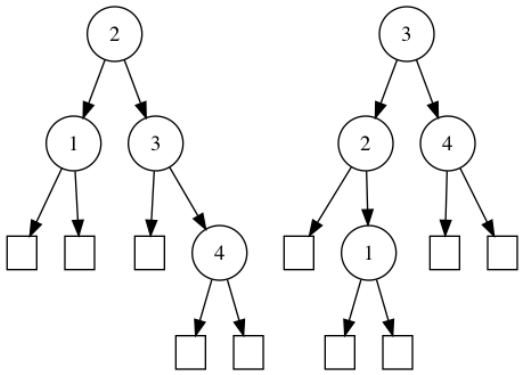
We start by performing a binary search on the ordered array to find the element k. This can be done because our array is ordered. We start with the midpoint of the array and recurse to the left if the element we are looking at is larger than k or recurse to the right if it is smaller. Through this, we can find k is $\mathcal{O}(\log n)$ time. Once we find k, we scan the array until we find an element that is not k. If k appears s times in the array, we perform $s-1$ comparisons. Therefore, the running time of our algorithm is $\mathcal{O}(\log n + s)$.

6 [A-3.2] $\mathcal{O}(n)$ time algorithm for constructing a binary search tree from a sorted array

We start by taking the element at the midpoint of the sorted array and place it at the root of the BST. We recurse into the left and right subarrays created by this midpoint; each time placing the midpoint of the left subarray in the left child of the root, and the midpoint of the right subarray in the right child of the root. For each subarray we perform a constant-time operation to place its midpoint in the BST. Further, we go perform this operation on each element of the ordered array exactly once. This conversion takes $\mathcal{O}(n)$ time. Note that the tree constructed is balanced and allows for $\mathcal{O}(\log n)$ search.

7 [R-4.4] Order of insertion into an AVL Tree

Consider the array $[1, 2, 3, 4]$. If we insert these elements into an AVL tree from the left we would have the first tree shown below. If we did it from the right we would have the second tree shown below.



This example contradicts the statement.

8 [R-4.7] Minimum number of nodes in a red-black tree of height 8

8.1 Wrong Answer 1

Proof of *Theorem 4.3* states that $h \leq 2\log(n+1)+1$ where n is the number of internal nodes and h is the height of the tree. Given $h = 8$, we have $8 \leq 2\log(n+1)+1$. Simplifying, we get $n + 1 \geq 2^{3.5} = 11.31$. Therefore $n \geq 11$. At least one internal node has two external nodes, thus the number of nodes is more than 13.

However, this answer does not say that we can bulid a red-black tree of height 8 with 13 nodes. It takes a lot more nodes.

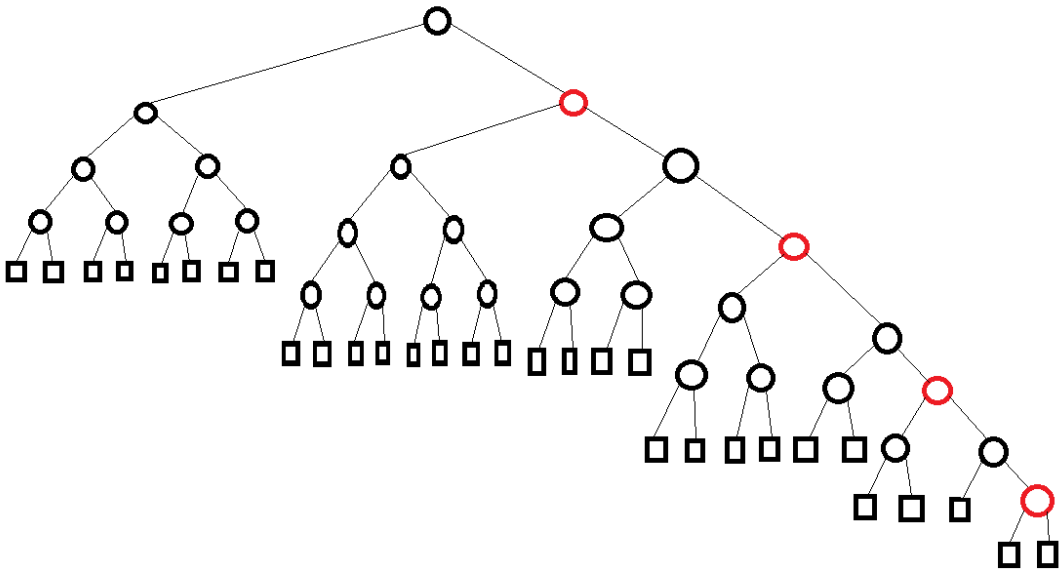
8.2 Wrong Answer 2

The smallest number of nodes that are possible in a red-black tree is $2^h - 1$, where h is the height of the tree assuming all nodes are black. Minimum number of nodes for a red-black tree of height 8 is therefore, $2^8 - 1 = 255$.

This doesn't take into account the red nodes in the tree. The best part about red-black trees are the guarantees we get by coloring the tree. Otherwise, it is just another binary search tree.

8.3 Correct Answer

Build the red-black tree of height 8 with the goal of having the minimum number of nodes.



The above red-blakc tree does not violate any rule and has a height of 8 with 61 nodes. The idea is to create a tree of the largest height with the least number of nodes. This can be achieved by having a black depth of $\frac{8}{2} = 4$ for leaves and alternating black and red along the longest subtree.

9 [A-4.4] Efficiently managing a set of key value pairs

We are required to compute a listing off all pairs in S in $\mathcal{O}(n)$ time. A listing of all the people in a given zip code should be computed in $\mathcal{O}(\log n + s)$ time, where s is the number of people in the zip code. Further, removals and insertions must run in $\mathcal{O}(\log n)$ time.

One way to achieve this is to store the elements of S in a balanced binary search tree T (for us, either an AVL tree or a red-black tree) ordered by a combination of zip code and name. In other words, the keys are pairs of zip codes and names, and can be lexicographically ordered. During insertion we first go by zip code and then break potential ties alphabetically. By the nature of balanced binary search trees, we are assured that insertions and removals run in $\mathcal{O}(\log n)$ time.

A full listing of the elements in S would take $\mathcal{O}(n)$ time. This can be acheived by performing an inorder traversal of the tree T we construct.

To list out all the people belonging to a certain zip code k, we perform a standard search on T until we find k and then list out all the nodes in subtrees that have k as their zip code. This operation can be performed in $\mathcal{O}(\log n + s)$ time where s is the number of people that belong to zip code k.