Sample Solution for Problem Set 6

Data Structures and Algorithms, Fall 2021

November 5, 2021

Contents

1 Problem 1

(a)

omitted.

(b)

Algorithm

We may assume the preorder and the postorder are valid. For a given preorder $Pre[l_r..r_r]$ and postorder $Post[l_o..r_o]$ of a full binary tree, the key of the root of this tree must be $Pre[l_r] = Post[r_o]$. Then,

- If $l_r = r_r$, which means this tree has only one node, directly return.
- If $l_r < r_r$, search for the index idx of the the element $Pre[l_r + 1]$ in the postorder $Post[l_o..r_o]$, recursively build the left subtree using preorder $Pre[l_r + 1..idx + 1]$ and postorder $Post[l_r..idx]$ and build the right subtree using preorder $Pre[idx + 2..r_r]$ and postorder $Post[idx + 1..r_o - 1]$.

Algorithm 1 Construct Full Tree

```
function ConstructFullTree(Pre, Post, l_r, r_r, l_o, r_o)
    root = NewNode();
    root.key = Pr[l_r];
    if l_r == r_r then
        return root;
    for i = l_o to r_o do
        if Po[i] == Pr[l_r + 1] then
            idx = i;
            break;
    root.left = ConstructFullTree(Pre, Post, l_r + 1, idx + 1, l_o, idx);
    root.right = ConstructFullTree(Pre, Post, idx + 2, r_r, idx + 1, r_o - 1);
    return root;
```

Correctness proof We prove by induction:

- (base case) when $size = l_r - r_r + 1 = 1$, the preorder and postorder both have the same single element. The algorithm construct a new node, set it value to that element and return it. The preorder and the postorder of this node matches the given preorder and postorder.
- (inductive hypothesis) when $size \leq l_r - r_r + 1 = 2k - 1$, the algorithm correctly constructs a tree that matches the given preorder and postorder.
- (inductive step) when $size = l_r - r_r + 1 = 2k + 1$, The root of the tree is $Pre[l_r]$. Assume the index of the key of the left child in the postorder traversal is idx. The preorder and the postorder of the left subtree is $Pre[l_r + 1..idx + 1]$ and $Post[l_o, idx]$. The preorder and the postorder of the right subtree is $Pre[idx + 2, r_r]$ and $Post[idx + 1, r_o - 1]$. By the inductive hypothesis, we can construct the left subtree and right subtree and link it to the root. The algorithm returns a tree that matches the given preorder and postorder.

Time complexity The algorithm takes $O(n^2)$ time in the worst case.

(c)

It is not possible to construct a general binary tree from preorder and postorder traversals. You can proof by given two different trees that has the same preorder and postorder.

2 Problem 2

- Lemma: At most $n-1$ right rotations suffice to transform the tree into a right-going chain.
- Proof of lemma:
  - I.B: For $n=1$, no rotation is required.
  - I.H: For $1 \le n \le k$, at most $n-1$ right rotations suffice to transform the tree into a right-going chain.
  - I.S: For $n = k+1$, assume the size of the left subtree is m ($1 \le m \le k$), then the size of the right subtree is $k-m$. From **I.H**, we can take at most

    $$t = (m-1) + \max\{0, (k-m-1)\} \le k-1$$

    right rotations to transform them into right-going chains. Then take 1 right rotation at root, the whole tree transform into a right-going chain, also. $t+1 \le n-1$.
- Proof of the original proposition:
  - We can use a stack to record the sequence of nodes that have been taken right rotations, take left rotations in reverse order can restore the original binary search tree.
  - Assume that we need to transform T_1 into T_2. Transform T_1 and T_2 to the right chain, record the right rotation order of T_2. Take these left rotations to T_1, which is a right-going chain in this moment.

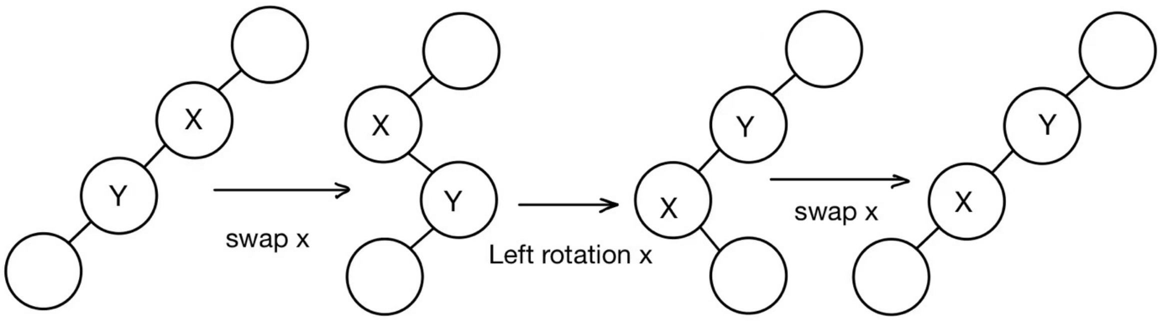
3 Problem 3

Algorithm

For a given binary tree, apply $O(n)$ rotations to transform it to a chain that each non-leaf nodes have only left child.

If we can use $O(1)$ operations to swap any two nearby nodes in the chain without affecting other nodes, we can apply a $O(n^2)$ bubble-sort-like algorithm to the chain such that each non-leaf node's key is larger than its left child's key. Such a chain is a BST.

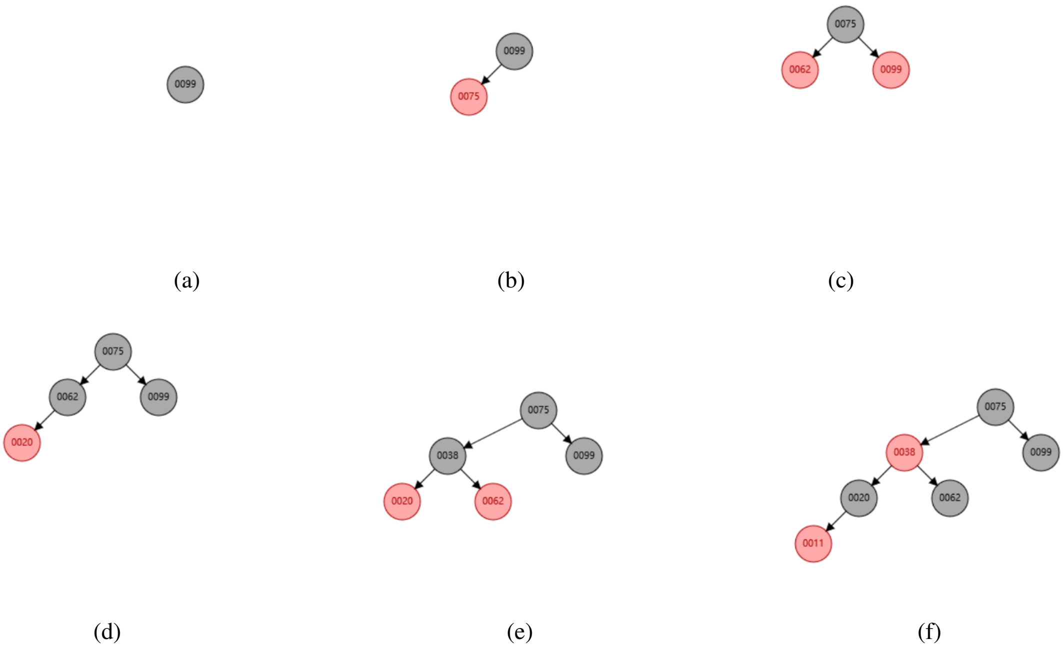
The swap operation can be done as follows:



Then we get an algorithm using at most $O(n^2)$ operations to transform an arbitrary n-node binary tree with distinct node values into a binary search tree.

4 Problem 4

4.1 a

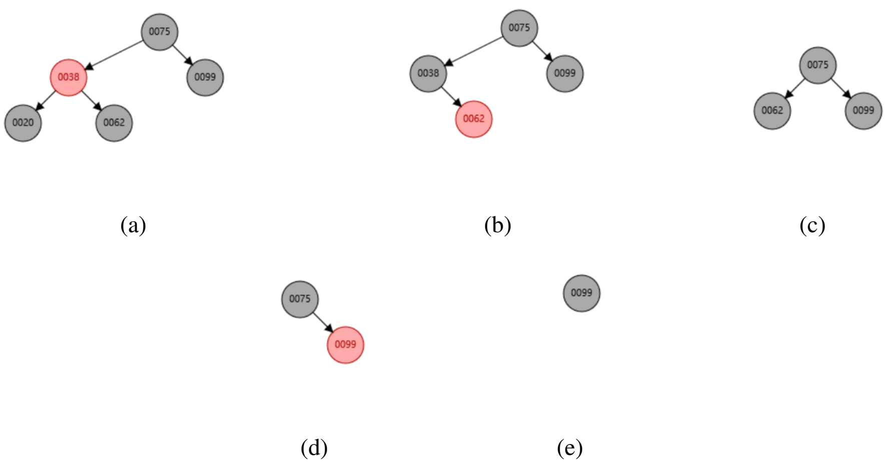


(a) (b) (c)

(d) (e) (f)

Figure 1: Insert

4.2 b



(a) (b) (c)

(d) (e)

Figure 2: Delete

5 Problem 5

(a)

Let T_h be the minimum number of vertices that an AVL tree with height h has. It can be seen that $T_n \geq T_{n-1} + T_{n-2} + 1$, which shows that there exists some constant $C > 1$, so that $T_n \geq \exp(Cn)$ when n is large enough.

(b)

Algorithm

Algorithm 2 Balance(x)

if $x.left.h > x.right.h + 1$ **then**
  if $x.left.left.h < x.left.right.h$ **then**
    $Left - Rotate(x.left)$
  $Right - Rotate(x)$
if $x.right.h > x.left.h + 1$ **then**
  if $x.right.right.h < x.right.left.h$ **then**
    $Right - Rotate(x.right)$
  $Left - Rotate(x)$

Correctness

WLOG, we will only prove correctness when $x.left.h > x.right.h + 1$.

- If $x.left.left.h < x.left.right.h$, it can be seen that $x.left.right.h = x.left.left.h + 1$ from our basic assumption. Hence, We may perform $Left - Rotate$ operation on $x.left$, which maintains balance property for all vertices in the left subtree of x, while negating the fact that $x.left.left.h < x.left.right.h$.
- After first operation, it is guaranteed that $x.left.left.h \geq x.left.right.h$. We may perform $Right - Rotate$ operation on vertice x, achieving the goal.

(c)

(d)

We only needs to check it performs $O(1)$ rotations. It follows from the fact that Balance operation will reduce the height of unbalanced subtree by exactly 1.

Remark

We assume that Rotation operation will maintain information such as the height of subtree and so on.

Algorithm 3 Insert(x, z)

```
if $x == NIL$ then
    $z.h = 0$ return;
if $x.key < z.key$ then
    $Insert(x.right, z)$
    if $x.right == NIL$ then
        $x.right = z$
else
    $Insert(x.left, z)$
    if $x.left == NIL$ then
        $x.left = z$
$left\_h = -1, right\_h = -1$
if $x.left \neq NIL$ then
    $left\_h = x.left.h$
if $x.right \neq NIL$ then
    $right\_h = x.right.h$
$x.h = \max(left\_h, right\_h) + 1$
$Balance(x)$
```

6 Problem 6

6.1 a

First we need to show that the expected height of a randomly built binary search tree on n distinct keys is $O(\lg n)$. Refer to CLRS Theorem 12.4 for detailed proof.
Each execution of the algorithm corresponds to a path from the root to a leaf node in Q_1 and Q_2, so the complexity is

$$\begin{aligned} O\left(\lg n_1 + \lg n_2\right) &= O\left(\lg n_1 n_2\right) \\ &= O\left(\lg \left(\frac{n_1 + n_2}{2}\right)^2\right) \\ &= O\left(\lg(n_1 + n_2)\right) \\ &= O\left(\lg n\right) \end{aligned} \tag{1}$$

6.2 b

- MakeQueue(): trivial.
- FindMin(Q): return $Q.val$.
- DeleteMin(Q): return $Meld(Q.left, Q.right)$.
- Insert(Q, x): create a meldable priority queue Q_1 with a single node, whose value is x. return $Meld(Q, Q_1)$.
- DecreaseKey(Q, x, y): detach the subtree rooted at x from Q. Set $x.val = y$. Then $meld(x, Q)$.
- Delete(Q, x): replace x with $meld(x.left, x.right)$.