W2_pxb410

Parv

February 2022

1 Question 1

Provide the corresponding expressions for the following notations. For each list what traversal was used to generate the expression.

1. prefix (i.e., functional notation) (3 P.)
2. infix (i.e., conventional) (3 P.)
3. postfix (i.e., reverse Polish) (3 P.)

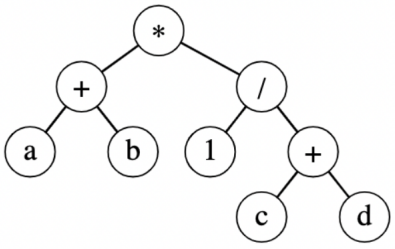


Figure 1: Binary Tree 1

1.1 Answer a

In **pre-order** traversal for prefix notation, the operator is printed first then it's left and right operands respectively. The final answer after the complete traversal will be: ***+ab/1+cd**

1.2 Answer b

In **in-order** traversal for infix notation, the left operand is printed first then the operator and at last the right operand. The final answer after the complete traversal will be: **(a+b)*(1/(c+d))**

1.3 Answer c

In **post-order** traversal for post-fix notation, the left operand is printed first then the right operand and finally the operator. This final output is also known as reverse polish notation: **ab+1cd+/***

2 Question 2

Examine the code below for insertion and deletion in a binary tree (this is from the textbook, figures 4.22 and 4.25). Write a minimal set of test cases that you would need to test these methods. For each case, explain your rationale behind it. Your explanations should refer to the specific conditions and line numbers that are tested. (10 P.)
Given code:

```
/**
 * Internal method to insert into a subtree.
 *
 * @param x the item to insert.
 * @param t the node that roots the subtree.
 * @return the new root of the subtree.
 */
private BinaryNode<AnyType> insert(AnyType x, BinaryNode<AnyType> t)
    {
    if (t == null)
        return new BinaryNode<>(x, null, null);
    int compareResult = x.compareTo(t.element);
    if (compareResult < 0)
        t.left = insert(x, t.left);
    else if (compareResult > 0)
        t.right = insert(x, t.right);
    else
        ; // Duplicate; do nothing
    return t;
}

/**
 * Internal method to remove from a subtree.
 *
 * @param x the item to remove.
 * @param t the node that roots the subtree.
 * @return the new root of the subtree.
 */
private BinaryNode<AnyType> remove(AnyType x, BinaryNode<AnyType> t)
    {
    if (t == null)
        return t; // Item not found; do nothing
```

```
        int compareResult = x.compareTo(t.element);
        if (compareResult < 0)
            t.left = remove(x, t.left);
        else if (compareResult > 0)
            t.right = remove(x, t.right);
        else if (t.left != null && t.right != null) // Two children
        {
            t.element = findMin(t.right).element;
            t.right = remove(t.element, t.right);
        } else
            t = (t.left != null) ? t.left : t.right;
        return t;
    }
```

2.1 Answer a: test cases for insertion method

- The first thing that we do in the insert method is if the parameter node is null it returns a new binary node with that element. So we have to test that first. For example, letfirst insert 100. 100 element is in the parent node.
- The second test is the left child test. Let's insert 50 and test will that 50 should be inserted in the left node.
- Same, goes with the right hand child. If we have 120, it should be the element of the right child node.
- A feature of the bst tree is that all the elements, in the bst are unique. So if we try to add an element that has already been added it shouldn't add it to the tree and instead just return the node where the element is added
- Next text is adding elements in the subtree. The code should follow a recursive approach. If we add 25 it should first go the left hand side of the 100(parent node) then go down to the node which has 50 in i and then become then left hand child of 25.

2.2 Answer b: test cases for deletion method

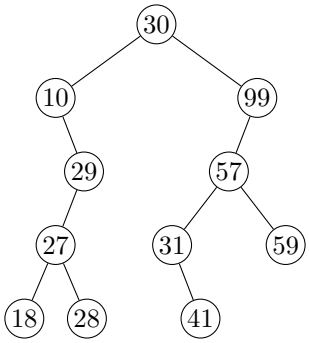
- if we have an empty tree with no node and we call the remove method(100, t) it should just simply return t.
- When we try to remove an element that doesn't exist it should again return 3.
- When removing a leaf node, we should just remove the point to the leaf node from it's parent node.
- If the node to be deleted has only one child. Then the child should be coped to the node and then node should be deleted.

- If the node has two children, then we should find the inorder successor of the node. We copy the content of the inorder successor to node and then delete the inorder successor. An inorder predecessor can also be used.

3 Question 3

3.1 Answer 3.a

Tree 1: Binary Search Tree after inserting all the elements



3.2 Answer 3.b & c

The balance factor is calculated by the formula:

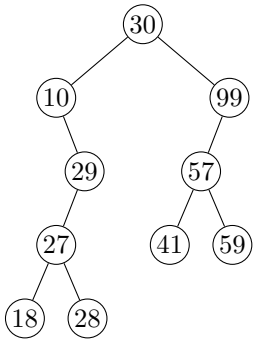
$$heightofleftsubtree - heightofrightsubtree$$

The balance after the insertion of 57 is: **2-2 = 0**

Similarly, the balance factor after the insertion of 18 is **4 -2 = 0**

3.3 Answer 3.c

Tree 2: Removing node 31

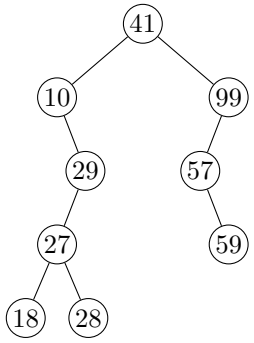


3.4 Answer 3.d

Deleting the parent node ncan be done on two ways: We can find the parent node's inorder successor and then delete the inorder successor and copy the contents of the inorder successor to the node. In this case the parent's node inroder successor is 41

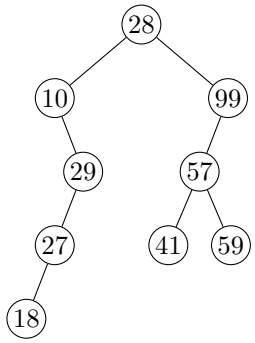
1st way:

Tree 3: After removing the parent node, 41 takes it place



The other way is to find the inorder predecessor which is usually the right-most element in the binary search tree on the left sub-tree. In this case it's 28.

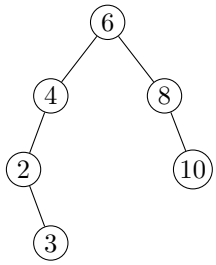
Tree 4: Removing node 30 and replacing it with 28.



4 Question 4

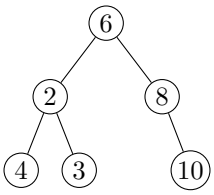
4.1 Answer

Tree 5: Adding 3 into the tree

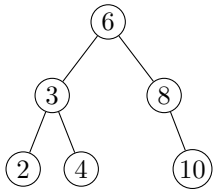


In this tree at node 4 there is an imbalance as the balance factor is **2**. This means the node left subtree has more elements then its' right subtree(which is empty). In this case we will have to perform some rotations to balance the node 4. It is important to note that the the rest of the nodes are balanced. Node 6 has a balance factor of **1**. Node 2 and 8 have a balance factor of **-1**. And, the leaf nodes 3 and 10 have a balance factor of 0.

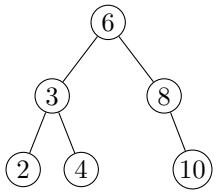
Tree 6: First left rotation on the 2 and 3



Tree 6: Second right rotation on the 2



Final tree after balancing node 4



5 Question 5

For adding 7, we simply add it to the leaf node of F because there is a place for the element to be stored in the leaf node F.

Similarly we can put 100 in the leaf node of V. 100 ¡ 97 and hence it will be in the sight leaf node of 97.

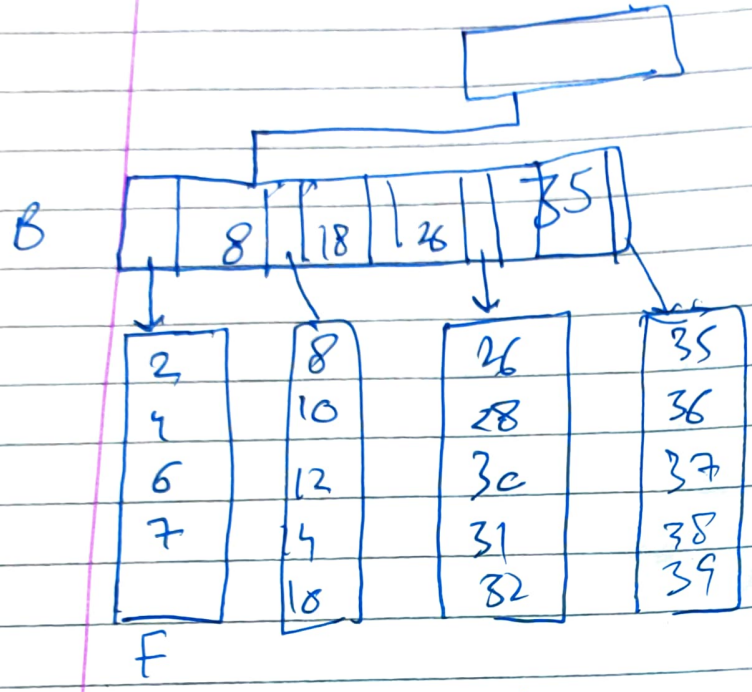
5.1 Answer

Since it invovled B-trees It drew it on paper as it was difficult to do it on latex

Q5

⇒ Adding 7

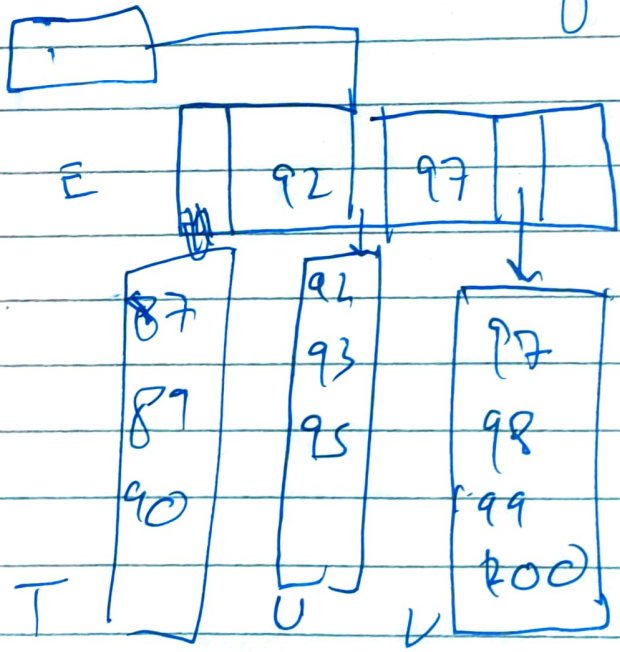
adding 7 is very straightforward. We just add it to the leaf node F because 7 < 8 and there is space available in F.



⇒ Adding 100

We add 100 to leaf node of V as 100 > 97

It will be on the right leaf node of 97

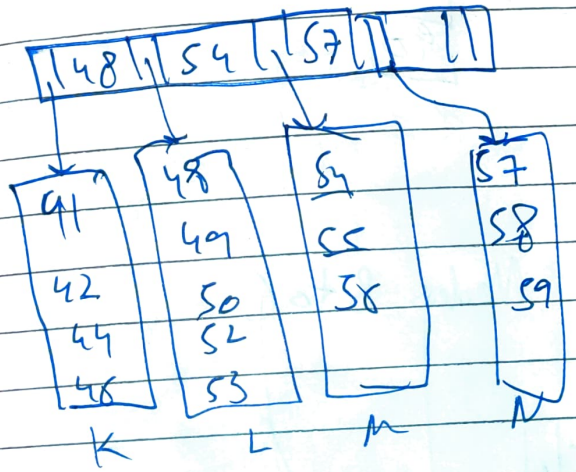


⇒ Removing 51

- Remove it from the leaf node of M. We update the interior node C and replace 51 by 52 which is the smallest element in the right subtree leaf M

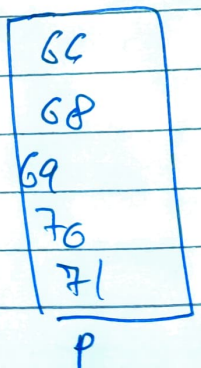
After removing 51 there are only 2 elements which is less than $\lceil 4/2 \rceil$.

In this scenario we can't even borrow from our left leaf node. So we merge L and M



⇒ Adding 71

We add it to the leaf node of P. It maintains the order

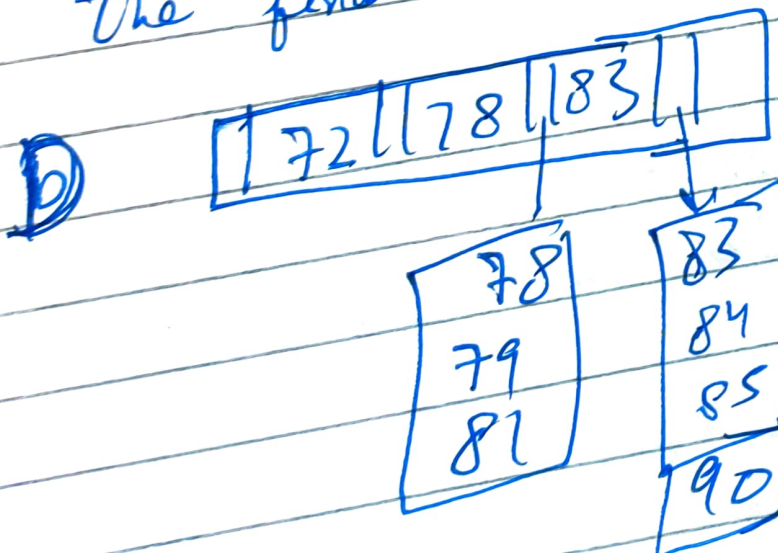


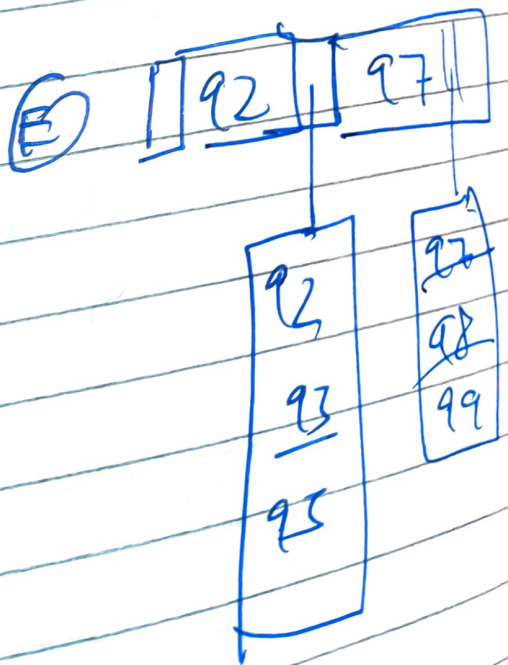
Removing 89

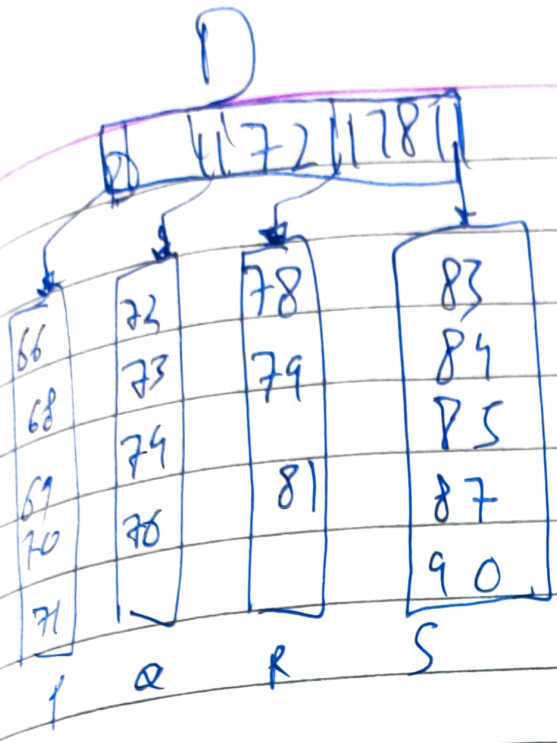
① Delete 89 from leaf node T elements reduce to below ½ and then we either borrow from left or merge with left node.

* We merge T with S.

The final node are

② 

③ 


D
72
78
66
68
69
70
71
73
73
74
76
78
79
81
83
84
85
87
90
P
Q
R
S


C
48
54
41
42
44
46
48
49
50
52
53
54
55
56
57
58
59
K
L
M
N


B
8
18
26
35
2
7
8
16
8
16
20
22
24
26
28
31
32
35
36
37
38
39
F
G
H
I
J