IT5003 Tutorial 5

Nov 2019

Q1a

- Given preorder and inorder traversal, get back the original tree.
- Preorder – Root, left, right
  - So for preorder, root is first item.
- Inorder – Left, Root, Right
  - Leftmost node is first item.
  - Since every item (except for the root), is followed by its parent,
    - 5, 7, 4, 28, 9 is left subtree, 18, 16, 17 is right subtree
- Recursively do this again and again.
  - Ie from the preorder traversal,
    - 7, 5, 28, 4, 9 for left subtree => 7 is root of left subtree
    - 18, 17, 16 for right subtree => 18 root of right subtree

```
        8
       / \
      7   18
     / \    \
    5  28    17
       / \   /
      4   9 16
```

Q1b

- Use the idea from q1a. A recursive implementation like most other tree algorithms (as need to traverse everything)
- Important assumption – all items are distinct. (if they are not, can somehow mark them, eg append an alphabet)
- Base case – empty tree. (How to tell?)
- Recursive case
  - Get the root of the tree (first item of preorder). Let this node be X
  - Get the index of X in the inorder traversal. Let this be xIdx
    - inOrder[0:xIdx] gives the inorder of left subtree of X
    - inOrder[xIdx+1:] gives the inorder right subtree of X
    - preOrder[1:xIdx+1] is the preorder of left subtree of X
    - preOrder[xIdx+1:] gives the preorder of the right subtree of X
  - X.left = constructTree(…?) (assume our constructTree function works! ☺)
  - X.right = constructTree(…?)

Q1c

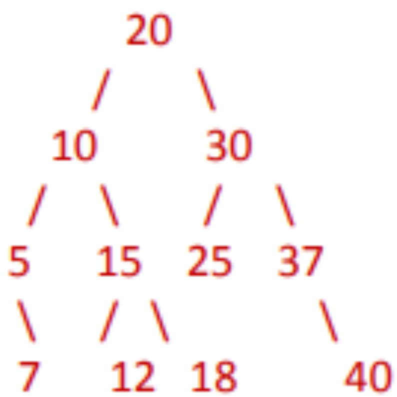
- If you think its correct, state your reasoning.
- If you think its wrong, give a counterexample.

- False, there is a simple counterexample of a very simple tree with two nodes!

Q2

- Similar idea to a preorder traversal. But instead of printing out the tree, we write the values to a new tree.
- It is a recursive formulation, like most tree algorithms
- Base case – Empty tree (how to tell?)
- Recursive step.
  - Let the current item be the root of the current subtree.
    - root = TreeNode(T.item)
  - Assume our flipTree algorithm is correct. Hence,
    - flipTree(T.leftT) gives the flipped left subtree
    - Similarly for flipTree(T.rightT)
  - Hence,
    - root.leftT = flipTree(…)?
    - root.rightT = flipTree(…)?

Q3A



Q3b

- Pre-order: 20, 10, 5, 7, 15, 12, 18, 30, 25, 37, 40 5
  - Root, Left, Right
- In-order: 5, 7, 10, 12, 15, 18, 20, 25, 30, 37, 40
  - Left, Root, Right
- Post-order: 7, 5, 12, 18, 15, 10, 25, 40, 37, 30, 20
  - Left, Right, Root
- Level-order: 20, 10, 30, 5, 15, 25, 37, 7, 12, 18, 40

Q3c – i

Delete 30:
According to the lecture notes, we move the smallest node (37) in the right subtree to the position of the deleted node.

```
        20
       /  \
     10    37
    / \   / \
   5  15 25  40
    \ / \
    7 12 18
```

Delete 10:

```
        20
       /  \
     12    37
    / \   / \
   5  15 25  40
    \   \
    7   18
```

Q3c - ii

Delete 15:

```
        20
       /  \
     12    37
    / \   / \
   5  18 25  40
    \
    7
```

The new tree is not a complete binary tree because 7 is the right child of 5.

Q4 – Building bBst from sorted list.

- Given sorted L, build a sorted list using function buildBSTfromSortedList
- Again, tree algorithms are usually recursive.
- Base case. If tree is empty (L == ?), return None

- Recursive case
  - We want a balanced tree. i.e. given a node X, 0<=size(X.left) – size(X.right) <=1
    - This is because since we allow the left subtree to contain more nodes in the case we cannot split evenly
  - Therefore, let the middle item L[len(L)//2] be the node. Let rIdx = len(L)//2
    - root=TreeNode(L[rIdx])
    - Assume our recursive solution works.
      - Then, buildBSTfromSortedList(L[:rIdx]) gives us the left subtree. Similar reasoning for right.
  - Therefore, similar reasoning from previous questions,
    - root.left = TreeNode(L[:rIdx])
    - root.right = TreeNode(L[:rIdx+1])
- Conveniently, L[:rIdx+1] will never go out of bounds in Python.
  - Eg. If x = [4,3,1], x[9] will result in an error but x[9:] will give [].