Binary Search Trees-II

ESO207

Indian Institute of Technology, Kanpur

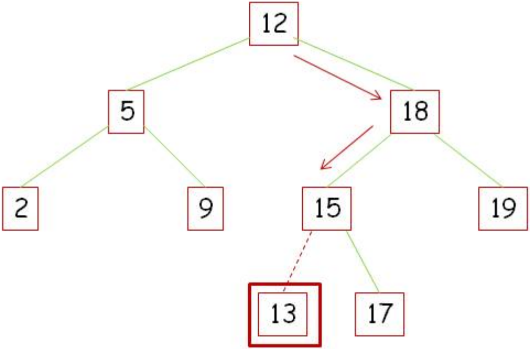
Summary

- In this part, we will study two basic operations on ***binary search tree***.
- **Insert** into a binary search tree, and,
- **Delete** a node from a binary search tree.
- These operations cause the dynamic set represented by a binary search tree to change.
- The data structure must be modified so that the binary-search-tree property continues to hold.

Idea behind Insertion

- Suppose we wish to insert a node x with key k in a given binary search tree T.
- The goal is to insert x somewhere in the tree so that the *bst* property remains good. The insertion finds a “safe leaf position” and inserts the node there.
- If the given key is less than the key at the root, we move to the left-child, otherwise, we move to the right child.
- This step is repeatedly performed until we “are ready to drop off ” the tree. The node is inserted in this very position.

Example



insert node with key 13 in tree

Pseudo-code

- Pseudo code for insertion BSTINSERT(T, z)– it inserts a node z in a binary search tree T.
- Procedure maintains a trailing node, namely, the parent node y of the current node x.
- The current node x moves down the tree:
  1. if $z.key < x.key$ then set $x = x.left$.
  2. if $z.key \geq x.key$ then set $x = x.right$.
- Keep track of the trailing node y: the parent of x.
- This ensures that when current node x becomes NIL(i.e., we fall off the tree) then the node is inserted as a child of the trailing node y.

Pseudo-code: insertion in binary search tree

```
BSTInsert(T, z) // insert node z into binary search tree T
    x = T.root
    y = NIL
    while x ≠ NIL
        y = x
        if z.key < x.key
            x = x.left
        else
            x = x.right
    if y == NIL   //Tree T is empty
        T.root = z
    if z.key < y.key
        y.left = z   //insert z as y's left child
    else
        y.right = z   //insert z as y's right child
```

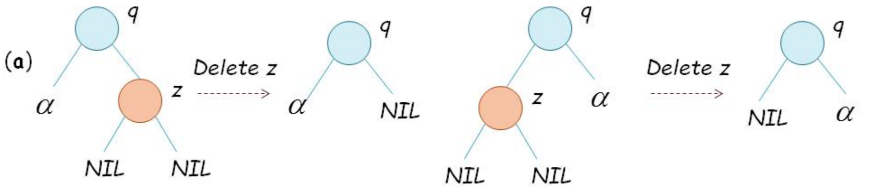
Time complexity of insertion

- As with other operations, we proceed from the root to some leaf, and then make $O(1)$ operations to insert the node.
- Hence, time complexity is $O(h)$.

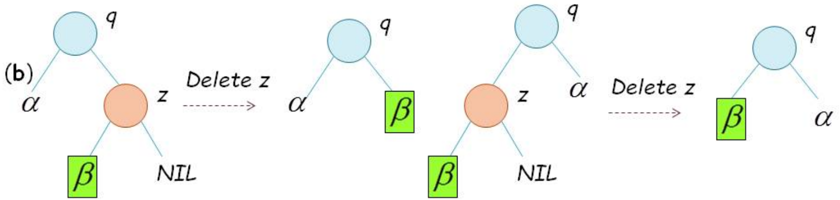
Deletion

- ***Task***: Delete a node z from a binary search tree T.
- These can be placed into three cases–the third case is a bit tricky.
  1. Case 1: If z is a leaf (i.e., no children) then z is removed and its parent is suitably modified to replace z with NIL as its child.
  2. Case 2: If z has only one child, then this child replaces z. For example, if z has parent q and child w. Then, w replaces z as the corresponding child of q.

Deletion: Cases 1 and 2



(**a**) Deleted node z is a leaf node. (**a.1**) z is a right child of its parent q, and (**a.2**) z is a left child of its parent q.



(b) Deleted node z has one child. z is replaced by this child.

Deletion: Case 3

- Case 3: z has two children.
- General idea: Replace z by its successor node, say y.
- There are two sub-cases.

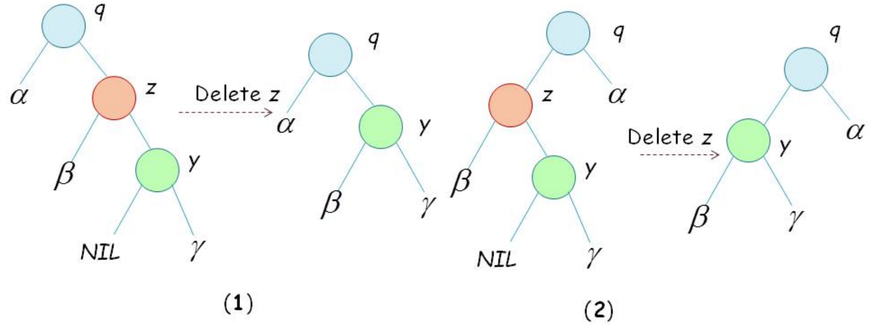
  Case 3.1 Successor node y of z is the right-child of z,

  Case 3.2 Successor node y of z is not the right child of z (but is further down in the sub-tree rooted at the right child of z).
- Rest of z's original right subtree becomes y's new right subtree, and,
- z's left subtree becomes y's new left subtree.

Deletion Case 3.1

- Call is to $Delete(T, z)$.
- Case 3.1: z has both children *and* the successor of z is the right child of y.
- Then, y has no left child.
- z is replaced by y (together with y's right sub-tree if one exists).

Deletion Case 3.1



y is the successor of z and is also the right child of z. Hence y does not have a left child.

The resulting tree after the deletion operation is shown. The node y replaces z as the corresponding child of q.

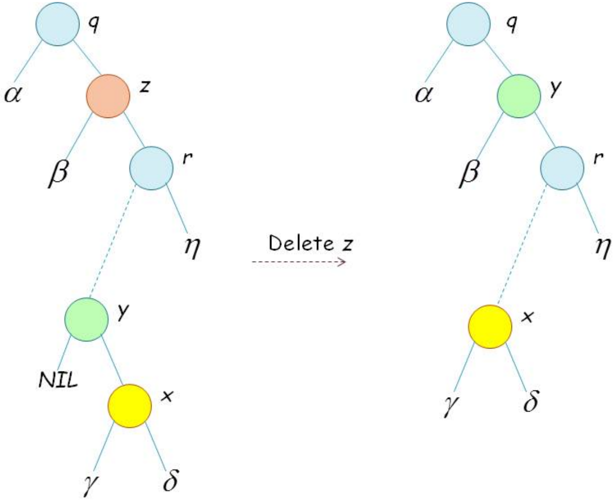
Deletion Case 3.2

- Call is to $Delete(T, z)$.
- Case 3.2: z has both children *and* the the successor node y of z is not the right child of z.
- y is located in the right-subtree of z.
- Let r be the right child of z.
- Then y is located in the left-subtree of r.
- The left child of y is NIL(y is a successor node of z in its right sub-tree).

Deletion Case 3.2

- We now perform a 2-step deletion, namely
  1. Delete y from its position, and
  2. Replace z by y.
- Deleting y from its position falls into either Case 1 (i.e., y is a leaf node) or Case 2 (i.e., y has one child) and is done accordingly.

Deletion: Case 3.2



(d) The subtree at y is replaced by the sub-tree at $\boldsymbol{x}$ and z is replaced by y.

Pseudo-code: Subroutine REPLACE

Procedure REPLACE replaces the node z by the node y in the tree T. Node z is now deleted.

```
REPLACE(T, z, y)
if z.p == NIL    // z is the root node
        T.root = y
elseif z == z.p.left  // z is left child of its parent
        z.p.left = y
                    // make y the left child of the parent node
elseif  z == z.p.right  // z is right child of its parent
        z.p.right = y
                   // make y the right child of the parent node
y.p = z.p
```

Pseudo-code for deletion

```
BST-DELETE(T, z)
    q = z.p
    if z.left == NIL  and z.right == NIL        // Case 1
        REPLACE(T, z, NIL)
    elseif z.left ≠ NIL  and z.right == NIL     // Case 2.1
        REPLACE(T, z, z.left)
    elseif z.left == NIL  and  z.right ≠ NIL    // Case 2.2
        REPLACE(T, z, z.right)
    y = SUCCESSOR(z)
    if y == z.right                     //Case 3.1
        y.left = z.left
        REPLACE(T, z, y)
    else                                //Case 3.2
        BST-DELETE(T, y)
        y.left = z.left
        y.right = z.right
        REPLACE(T, z, y)
```