Splay Tree | Set 1 (Search)

The worst case time complexity of Binary Search Tree (BST) operations like search, delete, insert is O(n). The worst case occurs when the tree is skewed. We can get the worst case time complexity as O(Logn) with AVL and Red-Black Trees.

Can we do better than AVL or Red-Black trees in practical situations?
Like AVL and Red-Black Trees, Splay tree is also self-balancing BST. The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in O(1) time if accessed again. The idea is to use locality of reference (In a typical application, 80% of the access are to 20% of the items). Imagine a situation where we have millions or billions of keys and only few of them are accessed frequently, which is very likely in many practical applications.

All splay tree operations run in O(log n) time on average, where n is the number of entries in the tree. Any single operation can take Theta(n) time in the worst case.

Search Operation
The search operation in Splay tree does the standard BST search, in addition to search, it also splays (move a node to the root). If the search is successful, then the node that is found is splayed and becomes the new root. Else the last node accessed prior to reaching the NULL is splayed and becomes the new root.

There are following cases for the node being accessed.

1) Node is root We simply return the root, don't do anything else as the accessed node is already root.

2) Zig: *Node is child of root* (the node has no grandparent). Node is either a left child of root (we do a right rotation) or node is a right child of its parent (we do a left rotation).
T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

```
            y                                     x
           / \     Zig (Right Rotation)          /  \
          x   T3   - - - - - - - - - ->         T1   y
         / \       < - - - - - - - - -              / \
        T1  T2     Zag (Left Rotation)            T2   T3
```

3) *Node has both parent and grandparent*. There can be following subcases.
……..**3.a) Zig-Zig and Zag-Zag** Node is left child of parent and parent is also left child of grand parent (Two right rotations) OR node is right child of its parent and parent is also right child of grand parent (Two Left Rotations).

```
Zig-Zig (Left Left Case):
       G                        P                           X
      / \                     /   \                        / \
     P   T4   rightRotate(G)  X     G     rightRotate(P)  T1   P
    / \       ============>  / \   / \    ============>       / \
```

```
   X  T3                 T1 T2 T3 T4                      T2  G
  / \                                                        / \
 T1 T2                                                      T3  T4
```

```
Zag-Zag (Right Right Case):
  G                           P                            X
 /  \                       /   \                         / \
T1   P     leftRotate(G)   G     X     leftRotate(P)     P   T4
    / \    ============>  / \   / \    ============>    / \
   T2   X                T1 T2 T3 T4                   G   T3
       / \                                            / \
      T3 T4                                          T1  T2
```

……..**3.b) Zig-Zag and Zag-Zig** Node is left child of parent and parent is right child of grand parent (Left Rotation followed by right rotation) OR node is right child of its parent and parent is left child of grand parent (Right Rotation followed by left rotation).

```
Zag-Zig (Left Right Case):
       G                        G                             X
      / \                     /   \                         /   \
     P   T4  leftRotate(P)   X     T4    rightRotate(G)    P     G
   /  \      ============>  / \          ============>   / \   /  \
  T1   X                   P  T3                        T1  T2 T3  T4
      / \                 / \
    T2  T3              T1   T2
```

```
Zig-Zag (Right Left Case):
  G                           G                           X
 /  \                        /  \                        /   \
T1   P    rightRotate(P)  T1    X     leftRotate(P)    G     P
    / \   =============>       / \    ============>   / \   / \
   X  T4                      T2   P                 T1  T2 T3  T4
  / \                             / \
 T2  T3                          T3  T4
```

Example:

```
         100                      100                          [20]
         /  \                    /   \                           \
       50   200                50    200                          50
      /          search(20)    /          search(20)             /  \
     40          ======>     [20]         ========>             30   100
    /            1. Zig-Zig     \         2. Zig-Zig             \     \
   30               at 40        30           at 100             40    200
  /                                \
[20]                               40
```

The important thing to note is, the search or splay operation not only brings the searched key to root, but also balances the BST. For example in above case, height of BST is reduced by 1.

Summary

1) Splay trees have excellent locality properties. Frequently accessed items are easy to find. Infrequent items are out of way.

2) All splay tree operations take O(Logn) time on average. Splay trees can be rigorously shown to run in O(log n) average time per operation, over any sequence of operations (assuming we start from an empty tree)

3) Splay trees are simpler compared to AVL and Red-Black Trees as no extra field is required in every tree node.

4) Unlike AVL tree, a splay tree can change even with read-only operations like search.

Applications of Splay Trees
Splay trees have become the most widely used basic data structure invented in the last 30 years, because they're the fastest type of balanced search tree for many applications.
Splay trees are used in Windows NT (in the virtual memory, networking, and file system code), the gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers, the most popular implementation of Unix malloc, Linux loadable kernel modules, and in much other software

Splay Tree | Set 2 (Insert)

The insert operation is similar to Binary Search Tree insert with additional steps to make sure that the newly inserted key becomes the new root.

Following are different cases to insert a key k in splay tree.

1) Root is NULL: We simply allocate a new node and return it as root.

2) Splay the given key k. If k is already present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.

3) If new root's key is same as k, don't do anything as k is already present.

4) Else allocate memory for new node and compare root's key with k.
…….**4.a)** If k is smaller than root's key, make root as right child of new node, copy left child of root as left child of new node and make left child of root as NULL.
…….**4.b)** If k is greater than root's key, make root as left child of new node, copy right child of root as right child of new node and make right child of root as NULL.

5) Return new node as new root of tree.

Example:

```
          100                    [20]                              25
          /  \                     \                              /  \
        50   200                    50                          20    50
       /           insert(25)      /  \       insert(25)            /  \
      40           ======>        30   100    ========>            30  100
     /           1. Splay(25)      \     \    2. insert 25          \    \
    30                             40    200                        40   200
   /
 [20]
```

Splay Tree | Set 3 (Delete)

Following are the different cases to delete a key **k** from splay tree.

1. If **Root is NULL:** We simply return the root.
2. Else Splay the given key k. If k is present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.
3. If new root's key is not same as **k**, then return the root as **k** is not present.
4. Else the key **k** is present.
    - Split the tree into two trees **Tree1** = root's left subtree and **Tree2** = root's right subtree and delete the root node.
    - Let the root's of **Tree1** and **Tree2** be **Root1** and **Root2** respectively.
    - If **Root1** is NULL: Return **Root2**.
    - Else, Splay the maximum node (node having the maximum value) of **Tree1**.
    - After the Splay procedure, make **Root2** as the right child of **Root1** and return **Root1**.