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).

```
Zig-Zag (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

Zag-Zig (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.

Implementation:

```
// The code is adopted from http://goo.gl/SDH9hH
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left, *right;
};

/* Helper function that allocates a new node with the gi
    NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(stru
```

```
    node->key   = key;
    node->left  = node->right  = NULL;
    return (node);
}

// A utility function to right rotate subtree rooted wit
// See the diagram given above.
struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

// A utility function to left rotate subtree rooted with
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

// This function brings the key at root if key is presen
// If key is not present, then it brings the last access
// root.  This function modifies the tree and returns th
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at roo
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
        // Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)
        {
            // First recursively bring the key as root o
            root->left->left = splay(root->left->left, k

            // Do first rotation for root, second rotati
            root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left
```

```
        {
            // First recursively bring the key as root o
            root->left->right = splay(root->left->right,

            // Do first rotation for root->left
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }

        // Do second rotation for root
        return (root->left == NULL)? root: rightRotate(r
    }
    else // Key lies in right subtree
    {
        // Key is not in tree, we are done
        if (root->right == NULL) return root;

        // Zag-Zig (Right Left)
        if (root->right->key > key)
        {
            // Bring the key as root of right-left
            root->right->left = splay(root->right->left,

            // Do first rotation for root->right
            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key)// Zag-Zag (Righ
        {
            // Bring the key as root of right-right and 
            root->right->right = splay(root->right->righ
            root = leftRotate(root);
        }

        // Do second rotation for root
        return (root->right == NULL)? root: leftRotate(r
    }
}

// The search function for Splay tree.  Note that this f
// returns the new root of Splay Tree.  If key is presen
// then, it is moved to root.
struct node *search(struct node *root, int key)
{
    return splay(root, key);
}

// A utility function to print preorder traversal of the
// The function also prints height of every node
void preOrder(struct node *root)
```

```
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Drier program to test above function*/
int main()
{
    struct node *root = newNode(100);
    root->left = newNode(50);
    root->right = newNode(200);
    root->left->left = newNode(40);
    root->left->left->left = newNode(30);
    root->left->left->left->left = newNode(20);

    root = search(root, 20);
    printf("Preorder traversal of the modified Splay tre
    preOrder(root);
    return 0;
}
```

Output:

```
Preorder traversal of the modified Splay tree is
20 50 30 40 100 200
```

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
(Source: http://www.cs.berkeley.edu/~jrs/61b/lec/36)

We will soon be discussing insert and delete operations on splay trees.

References:

http://www.cs.berkeley.edu/~jrs/61b/lec/36
http://www.cs.cornell.edu/courses/cs3110/2009fa/recitations/rec-splay.html
http://courses.cs.washington.edu/courses/cse326/01au/lectures/SplayTrees.ppt