Splay Tree | Set 1 (Search)

Difficulty Level : Medium • Last Updated : 15 May, 2022

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).

, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

```
             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 right child of parent and parent is left child of grand parent (Left Rotation followed by right rotation) OR node is left child of its parent and parent is right child of grand parent (Right Rotation followed by left rotation).

```
Zag-Zig (Left Right Case):
       G                        G                            X
       / \                   / \
     T2   T3               T1   T2

Zig-Zag (Right Left Case):
  G                          G                            X
 /  \                       /  \                         /   \
T1    P     rightRotate(P)  T1    X     leftRotate(G)     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:

```
    public:
    int key;
    node *left, *right;
};

/* Helper function that allocates
a new node with the given key and
    NULL left and right pointers. */
node* newNode(int key)
{
    node* Node = new node();
    Node->key = key;
```

```
// See the diagram given above.
node *rightRotate(node *x)
{
    node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

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

// This function brings the key at
// root if key is present in tree.
// If key is not present, then it
// brings the last accessed item at
// root. This function modifies the
// tree and returns the new root
node *splay(node *root, int key)
{
    // Base cases: root is NULL or
    {
        // 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 of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root,
            // second rotation is done after else
            root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring
            // the key as root of left-right
            root->left->right = splay(root->left->right, key);

            // 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(root);
    }
    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, key);

            // Do first rotation for root->right
            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key)// Zag-Zag (Right Right)
        {
            // Bring the key as root of
        return (root->right == NULL)? root: leftRotate(root);
    }
}

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

// A utility function to print
// preorder traversal of the tree.
// The function also prints height of every node
void preOrder(node *root)
{
    if (root != NULL)
    {
        cout<<root->key<<" ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Driver code*/
int main()
{
    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);
    cout << "Preorder traversal of the modified Splay tree is \n";
    preOrder(root);
    return 0;
}

// This code is contributed by rathbhupendra
```

C

```
{
    int key;
    struct node *left, *right;
};

/* Helper function that allocates a new node with the given key and
    NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key   = key;
    node->left  = node->right  = NULL;
    return (node);
}

// A utility function to right rotate subtree rooted with y
// 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 x
// 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 present in tree.
// If key is not present, then it brings the last accessed item at
// root.  This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is done after else
            root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);

            // 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(root);
    }
    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, key);

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

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

// The search function for Splay tree. Note that this function
// A utility function to print preorder traversal of the tree.
// 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);
    }
}

/* Driver 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 tree is \n");
    preOrder(root);
    return 0;
}
```

Java

```
// Java implementation for above approach
class GFG
{

// An AVL tree node
static class node
{

    int key;
    node left, right;
};

/* Helper function that allocates
a new node with the given key and
```

```
        return (Node);
    }

    // A utility function to right
    // rotate subtree rooted with y
    // See the diagram given above.
    static node rightRotate(node x)
    {
        node y = x.left;
        x.left = y.right;
        y.right = x;
        return y;
    }

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

    // This function brings the key at
    // root if key is present in tree.
    // If key is not present, then it
    // brings the last accessed item at
    // root. This function modifies the
    // tree and returns the new root
    static node splay(node root, int key)
    {
        // Base cases: root is null or
        // key is present at root
        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)
            {
                root = rightRotate(root);
            }
            else if (root.left.key < key) // Zig-Zag (Left Right)
            {
                // First recursively bring
                // the key as root of left-right
                root.left.right = splay(root.left.right, key);

                // 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(root);
        }
        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, key);

                // Do first rotation for root.right
                if (root.right.left != null)
                    root.right = rightRotate(root.right);
            }
            else if (root.right.key < key)// Zag-Zag (Right Right)
            {
                // Bring the key as root of
                // right-right and do first rotation
                root.right.right = splay(root.right.right, key);
                root = leftRotate(root);
            }

            // Do second rotation for root
            return (root.right == null) ?
                                root : leftRotate(root);
        }
    }

    // The search function for Splay tree
```

```
}

// A utility function to print
// preorder traversal of the tree.
// The function also prints height of every node
static void preOrder(node root)
{
    if (root != null)
    {
        System.out.print(root.key + " ");
        preOrder(root.left);
        preOrder(root.right);
    }
}

// Driver code
public static void main(String[] args)
{
    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);
    System.out.print("Preorder traversal of the" +
                     " modified Splay tree is \n");
    preOrder(root);
}
}

// This code is contributed by 29AjayKumar
```

C#

```
// C# implementation for above approach
using System;

class GFG
{

// An AVL tree node
public class node
{
a new node with the given key and
null left and right pointers. */
static node newNode(int key)
{
    node Node = new node();
    Node.key = key;
    Node.left = Node.right = null;
    return (Node);
}

// A utility function to right
// rotate subtree rooted with y
// See the diagram given above.
static node rightRotate(node x)
{
    node y = x.left;
    x.left = y.right;
    y.right = x;
    return y;
}

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

// This function brings the key at
// root if key is present in tree.
// If key is not present, then it
// brings the last accessed item at
// root. This function modifies the
// tree and returns the new root
static node splay(node root, int key)
{
    // Base cases: root is null or
    // key is present at root
    if (root == null || root.key == key)
        return root;

    // Key lies in left subtree
    if (root.key > key)
```

```
            {
                // First recursively bring the
                // key as root of left-left
                root.left.left = splay(root.left.left, key);

                // Do first rotation for root,
                // second rotation is done after else
                root = rightRotate(root);
            }
            else if (root.left.key < key) // Zig-Zag (Left Right)
            {
                // First recursively bring
                // the key as root of left-right
                root.left.right = splay(root.left.right, key);

                // 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(root);
        }
        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, key);

                // Do first rotation for root.right
                if (root.right.left != null)
                    root.right = rightRotate(root.right);
            }
            else if (root.right.key < key)// Zag-Zag (Right Right)
            {
                // Bring the key as root of
                // right-right and do first rotation
                root.right.right = splay(root.right.right, key);
                root = leftRotate(root);
            }
```

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

// A utility function to print
// preorder traversal of the tree.
// The function also prints height of every node
static void preOrder(node root)
{
    if (root != null)
    {
        Console.Write(root.key + " ");
        preOrder(root.left);
        preOrder(root.right);
    }
}

// Driver code
public static void Main(String[] args)
{
    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);
    Console.Write("Preorder traversal of the" +
                  " modified Splay tree is \n");
    preOrder(root);
}
}

// This code is contributed by 29AjayKumar
```

vascript

```
<script>
// Javascript implementation for above approach
```

```
    null left and right pointers. */
    constructor(key)
    {
        this.key = key;
        this.left = this.right = null;
    }
}

// A utility function to right
// rotate subtree rooted with y
// See the diagram given above.
function rightRotate(x)
{
    let y = x.left;
    x.left = y.right;
    y.right = x;
    return y;
}

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

// This function brings the key at
// root if key is present in tree.
// If key is not present, then it
// brings the last accessed item at
// root. This function modifies the
// tree and returns the new root
function splay(root,key)
{

    // Base cases: root is null or
    // key is present at root
    if (root == null || root.key == key)
        return root;

    // Key lies in left subtree
    if (root.key > key)
    {
            // First recursively bring the
            // key as root of left-left
            root.left.left = splay(root.left.left, key);

            // Do first rotation for root,
            // second rotation is done after else
            root = rightRotate(root);
        }
        else if (root.left.key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring
            // the key as root of left-right
            root.left.right = splay(root.left.right, key);

            // 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(root);
    }
    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, key);

            // Do first rotation for root.right
            if (root.right.left != null)
                root.right = rightRotate(root.right);
        }
        else if (root.right.key < key)// Zag-Zag (Right Right)
        {
            // Bring the key as root of
            // right-right and do first rotation
            root.right.right = splay(root.right.right, key);
            root = leftRotate(root);
        }

        // Do second rotation for roo
```

```
// Note that this function returns the
// new root of Splay Tree. If key is
// present in tree then, it is moved to root.
function search(root,key)
{
     return splay(root, key);
}

// A utility function to print
// preorder traversal of the tree.
// The function also prints height of every node
function preOrder(root)
{
    if (root != null)
    {
        document.write(root.key + " ");
        preOrder(root.left);
        preOrder(root.right);
    }
}

// Driver code
let root = new Node(100);
    root.left = new Node(50);
    root.right = new Node(200);
    root.left.left = new Node(40);
    root.left.left.left = new Node(30);
    root.left.left.left.left = new Node(20);

    root = search(root, 20);
    document.write("Preorder traversal of the" +
                     " modified Splay tree is <br>");
    preOrder(root);

// This code is contributed by rag2127
</script>
```

Output:

```
Preorder traversal of the modified Splay tree is
[illegible]0 50 30 40 100 200
```

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, For 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)
See Splay Tree | Set 2 (Insert) for splay tree insertion.

Advantages of Splay Trees:

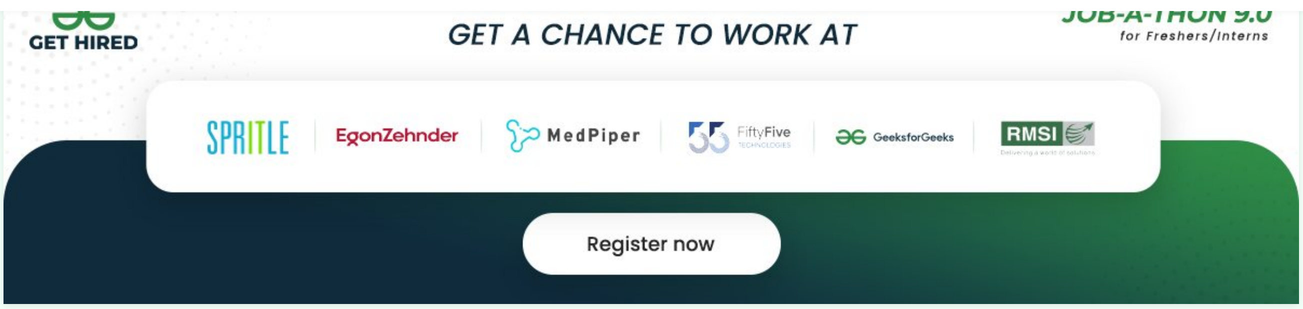
- Useful for implementing caches and garbage collection algorithms.
- Require less space as there is no balance information is required.
- Splay trees provide good performance with nodes containing identical keys.

Disadvantages of Splay Trees:

- The height of a splay tree can be linear when accessing elements in non decreasing order.
- The performance of a splay tree will be worse than a balanced simple binary search tree in case of uniform access.

References:

ttp://www.cs.berkeley.edu/~jrs/61b/lec/36
tp://www.cs.cornell.edu/courses/cs3110/2009fa/recitations/rec-splay.html
http://courses.cs.washington.edu/cour ses/ se326/01au/lectures/SplayTrees.ppt



Previous Next

RECOMMENDED ARTICLES

Page : 1 2 3

01 Splay Tree | Set 2 (Insert)
16, Jan 14

02 Splay Tree | Set 3 (Delete)
13, Jun 17

03 K Dimensional Tree | Set 1 (Search and Insert)
31, Oct 14

04 m-Way Search Tree | Set-2 | Insertion and Deletion
02, May 19

05 Convert a Generic Tree(N-array Tree) to Binary Tree
29, Oct 20

06 Ternary Search Tree
13, Jan 13

07 How to handle duplicates in Binary Search Tree?
11, May 15

08 Treap (A Randomized Binary Search Tree)
22, Oct 15

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Medium

Easy Normal Medium Hard Expert

Improved By : Ujjwal Mishra, rathbhupendra, nidhi_biet, 29AjayKumar, rag2127, tcostell, sagar0719kumar, guptavivek0503, adhamasharkawy

Article Tags : Self-Balancing-BST, splay-tree, Advanced Data Structure

Improve Article Report Issue

Start Your Coding Journey Now! Login Register

In Media SDE Cheat Sheet
Contact Us Machine learning
Privacy Policy CS Subjects
Copyright Policy Video Tutorials

News

Top News
Technology
Work & Career
Business
Finance
Lifestyle

Languages

Python
Java
CPP
Golang
C#
SQL

Web Development

Web Tutorials
Django Tutorial
HTML
CSS
JavaScript
Bootstrap

Contribute

Write an Article
Improve an Article
Pick Topics to Write
Write Interview Experience
Internships
Video Internship

▲