Data Structure Semester Project

M Mujtaba Bawani k163863

Mustafa Irfan k163886

SPLAY TREE

Data Structure – Binary Search Tree

**Why to use Splay tree?**

In a binary search tree, a lot of time is required to access last inserted node and its time complexity is O(n), and BST is not a balanced tree. AVL tree solves the problem of balancing, its time complexity is O(log n). The AVL tree requires extra storage for height field in each node, and for every node it becomes complicated to calculate balance factor. Still, a lot of time is required o access recently inserted element in the tree.

Splay tree has the benefit of fast access to recently inserted element, as the last inserted element becomes root node of the tree.

**Time complexity:**

All splay trees operations such as Search, Insert and Delete takes O(log n) time complexity.

In Insertion, first new element is inserted with the same operation as of binary search tree, than it performs splay. In splay the newly inserted node comes at the top through rotations.

In Searching, it searches like BST, than it splays, that is the element searched comes at the top of the tree or it becomes root, following rotations.

In deletion, it searches that element by BST method, then it splays, that is the element to be deleted becomes root node, then it is deleted. After deletion the left child of the root node becomes the root node of the tree.

**Space complexity:**

The space complexity of splay tree is O(n), as it does not take much space to store elements. No extra memory is required.

**Comparison with BST**

In splay tree’s insertion, some properties are same as BST that is left sub tree contains only nodes with keys or data less than the node’s key, and the right sub tree contains only nodes with keys greater than the node’s key, and the left and right sub tree’s also have above properties. But after inserting the splay tree call splay function recursively until the newly inserted node becomes the root node of the tree.

The same is the case in search operation that is first it searches an element as it is searched in a binary search tree, then it is splayed, hence the element searched becomes the root node of the tree, following splay rotations.

In deletion, first the element to be deleted is searched as it is searched in binary search tree. Later, it is splayed, that is it is moved to top of the tree and it becomes root node. Then this node is deleted with the following cases:

* If the root is null, the root is returned, else splay the given node if it is present it becomes the root node.
* If the given node to be deleted is present, delete the root node, check for left sub tree’s root node if it is null, return root node of right sub tree, else splay rightmost node of left sub tree and after splaying make root node of right sub tree as right child of left sub tree.

However, in binary search tree, splaying was not performed in all insertion, searching and deletion.

**Opertaions In Splay Tree:-**

**1. Void insert(object x)**

Temp ← root

Par ← null

While (temp != null)

par ← temp

If (temp.key.compareTo(x) < 0 ) temp ← temp.left

else temp ← temp.right

Temp ← new Node (x)

Temp.parent ← par

If (par = null) root ← temp

Else if (par.key.compareTo(x) < 0 ) par ← par.right

Else par ← par.left

Splay(temp) //after inserting splay it move it on to the root of tree

**2. Node delete(object x)**

If (root = null) return null //tree is empty

If (search(x) = null) return null //x is not present in tree

//now the object to be deleted is on root of the tree

Temp ← root

If (root.isLeaf())

root ← null

return temp

If (root.isEither())

If (root.left != null)

root.left.parent ← null

root ← root.left

Else

root.right.parent ← null

root ← root.right

return temp

//root is full node so now we will split the tree into left subtree and right subtree

leftTree ← root.left

rightTree ← root.right

Right.parent ← null

leftTree.parent ← null

T ← predecessor (leftTree)

Splay (T) //will move predecessor of root on the top

Root.right ← rightTree

rightTree.parent ← root

Return temp

**3. Node search(object x)**

Temp ← root

If (temp = null) return null //empty tree

If (temp.key = data) return temp

While (temp != null)

If (temp.key > data)

temp ← temp.left

else if (temp.key. < data )

temp ← temp.right

else

return splay (temp) //when found splay it move it on to the root of tree

Return null //not found

**Experimental analysis of SplayTree:-**

Test Case 1

run: 230,155,156,325,214, 182,19,325,455,218,293,69,120,349,300,450,382,9,38,345

Inserting 230

Node{key=230}

Inserting 155

Case1(a):- ZigRight

Node{key=155} Node{key=230}

Inserting 156

Case3(b):- ZigZagLeft

Node{key=156} Node{key=155} Node{key=230}

Inserting 325

Case2(b):- ZigZigLeft

Node{key=325} Node{key=230} Node{key=156} Node{key=155}

Inserting 214

Case3(a):- ZigZagRight

Case1(a):- ZigRight

Node{key=214} Node{key=156} Node{key=325} Node{key=155} Node{key=230}

Inserting 182

Case3(a):- ZigZagRight

Node{key=182} Node{key=156} Node{key=214} Node{key=155} Node{key=325} Node{key=230}

Inserting 19

Case2(a):- ZigZigRight

Case1(a):- ZigRight

Node{key=19} Node{key=182} Node{key=155} Node{key=214} Node{key=156} Node{key=325} Node{key=230}

Inserting 355

Case2(b):- ZigZigLeft

Case2(b):- ZigZigLeft

Node{key=355} Node{key=182} Node{key=19} Node{key=325} Node{key=155} Node{key=214} Node{key=156} Node{key=230}

Inserting 425

Case1(b):- ZigLeft

Node{key=425} Node{key=355} Node{key=182} Node{key=19} Node{key=325} Node{key=155} Node{key=214} Node{key=156} Node{key=230}

Inserting 218

Case3(b):- ZigZagLeft

Case3(b):- ZigZagLeft

Case2(a):- ZigZigRight

Node{key=218} Node{key=182} Node{key=355} Node{key=19} Node{key=214} Node{key=325} Node{key=425} Node{key=155} Node{key=230} Node{key=156}

Inserting 293

Case3(a):- ZigZagRight

Case3(b):- ZigZagLeft

Node{key=293} Node{key=218} Node{key=355} Node{key=182} Node{key=230} Node{key=325} Node{key=425} Node{key=19} Node{key=214} Node{key=155} Node{key=156}

Inserting 69

Case3(b):- ZigZagLeft

Case2(a):- ZigZigRight

Case1(a):- ZigRight

Node{key=69} Node{key=19} Node{key=293} Node{key=182} Node{key=355} Node{key=155} Node{key=218} Node{key=325} Node{key=425} Node{key=156} Node{key=214} Node{key=230}

Inserting 120

Case2(a):- ZigZigRight

Case3(b):- ZigZagLeft

Node{key=120} Node{key=69} Node{key=293} Node{key=19} Node{key=155} Node{key=355} Node{key=182} Node{key=325} Node{key=425} Node{key=156} Node{key=218} Node{key=214} Node{key=230}

Inserting 349

Case3(a):- ZigZagRight

Case2(b):- ZigZigLeft

Node{key=349} Node{key=293} Node{key=355} Node{key=120} Node{key=325} Node{key=425} Node{key=69} Node{key=155} Node{key=19} Node{key=182} Node{key=156} Node{key=218} Node{key=214} Node{key=230}

Inserting 300

Case3(b):- ZigZagLeft

Case1(a):- ZigRight

Node{key=300} Node{key=293} Node{key=349} Node{key=120} Node{key=325} Node{key=355} Node{key=69} Node{key=155} Node{key=425} Node{key=19} Node{key=182} Node{key=156} Node{key=218} Node{key=214} Node{key=230}

Inserting 450

Case2(b):- ZigZigLeft

Case2(b):- ZigZigLeft

Node{key=450} Node{key=349} Node{key=300} Node{key=425} Node{key=293} Node{key=325} Node{key=355} Node{key=120} Node{key=69} Node{key=155} Node{key=19} Node{key=182} Node{key=156} Node{key=218} Node{key=214} Node{key=230}

Inserting 382

Case3(a):- ZigZagRight

Case3(a):- ZigZagRight

Node{key=382} Node{key=349} Node{key=450} Node{key=300} Node{key=355} Node{key=425} Node{key=293} Node{key=325} Node{key=120} Node{key=69} Node{key=155} Node{key=19} Node{key=182} Node{key=156} Node{key=218} Node{key=214} Node{key=230}

Inserting 9

Case2(a):- ZigZigRight

Case2(a):- ZigZigRight

Case2(a):- ZigZigRight

Case1(a):- ZigRight

Node{key=9} Node{key=382} Node{key=300} Node{key=450} Node{key=120} Node{key=349} Node{key=425} Node{key=19} Node{key=293} Node{key=325} Node{key=355} Node{key=69} Node{key=155} Node{key=182} Node{key=156} Node{key=218} Node{key=214} Node{key=230}

Inserting 38

Case3(b):- ZigZagLeft

Case2(a):- ZigZigRight

Case3(b):- ZigZagLeft

Node{key=38} Node{key=9} Node{key=382} Node{key=19} Node{key=120} Node{key=450} Node{key=69} Node{key=300} Node{key=425} Node{key=293} Node{key=349} Node{key=155} Node{key=325} Node{key=355} Node{key=182} Node{key=156} Node{key=218} Node{key=214} Node{key=230}

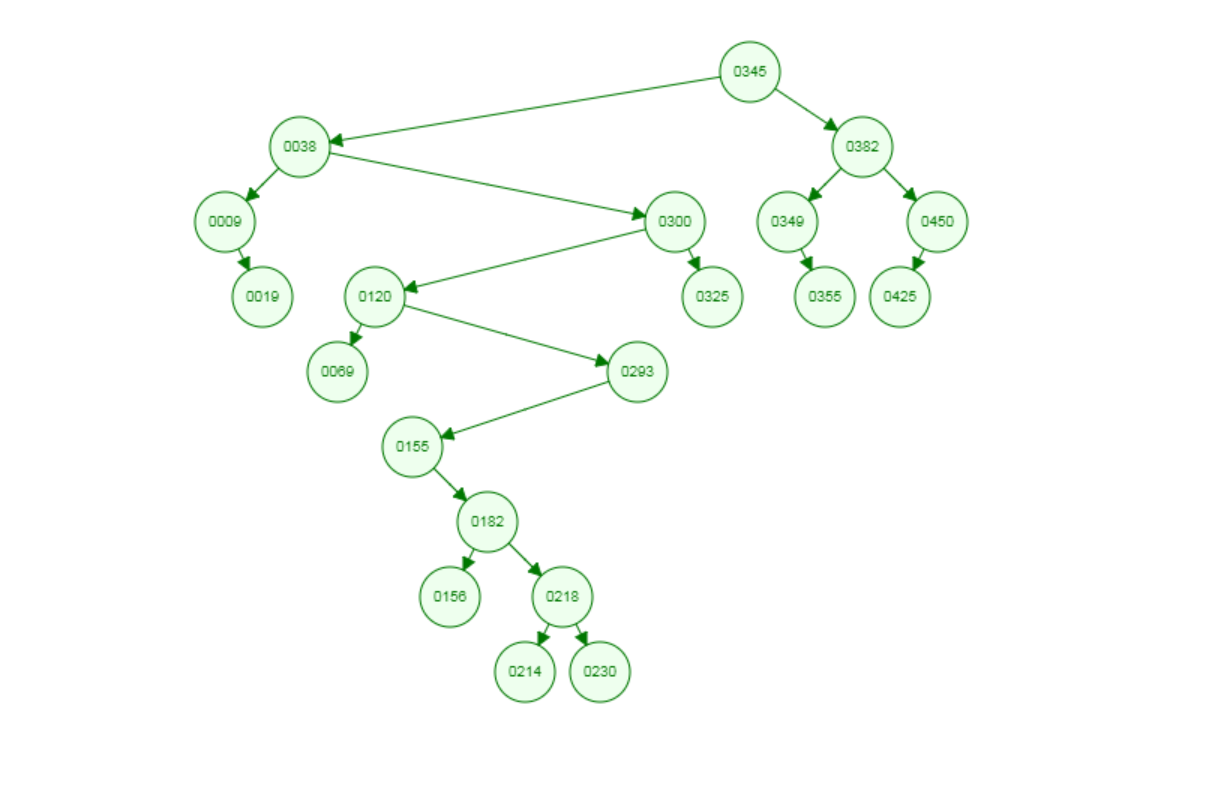
Inserting 345

Case3(a):- ZigZagRight

Case2(b):- ZigZigLeft

Case3(b):- ZigZagLeft

Node{key=345} Node{key=38} Node{key=382} Node{key=9} Node{key=300} Node{key=349} Node{key=450} Node{key=19} Node{key=120} Node{key=325} Node{key=355} Node{key=425} Node{key=69} Node{key=293} Node{key=155} Node{key=182} Node{key=156} Node{key=218} Node{key=214} Node{key=230}

****

**Comparison Analysis:-**

**INSERTION**

**Deletion:-**

**Applications:**

Splay trees can be used in following cases:

* When queries are highly biased. When a set of queries favors a specific element. It is done when query that is done frequently would appear at the top of the tree that is the same case in Splay trees that in splay operation the node becomes root node of the tree. It makes easy to find the data searched recently, since it is on top of the tree.
* It can also be used in Network routers. As the network router receives network packets at a high rate from incoming connections, so the router needs a big table that is used to lookup an IP address to find out which outgoing connection to use. The splay tree data structure helps when router needs to find the IP address that has been used once and it is likely to be used again and maybe many times.
* Storing caches from frequent activity serves a purpose of splay trees, and it’s logic can be implemented behind cache.
* Garbage collection is the systematic recovery of computer storage pool that is being used by a program when that program no longer needs the storage. The Garbage collection feature uses splay approach to access those least frequently searched nodes and delete them. This is done after storing data into tree data structure, then finding tree’s height, and then all left and right nodes at height h-1 are pointed to null.

**Summary:-**

1. Can be shown that any M consecutive operations starting from an empty tree take at most O(M log(N))
   * All splay tree operations run in amortized O(log n) time
2. Splay trees are tree structures that:
   * Are not perfectly balanced all the time
   * Data most recently accessed is near the root.
3. Implements most-recently used (MRU) logic
4. Splaying can be done top-down; better because:
   * only one pass
   * no recursion pointers necessary
5. Splay trees are *very* effective search trees
   * relatively simple: no extra fields required
   * excellent **locality** properties:

frequently accessed keys are cheap to find (near top of tree)

infrequently accessed keys stay out of the way (near bottom of tree)