Splay Tree

UWU/CST/16/026

December 27, 2018

Introduction

Splay tree is a self-balancing binary search tree where we adjust the tree structure after every access in order to move the accessed element to the top.

This is done by repeated performing a series of rotations to the element and its neighbour.

It is optimizes so that elements that have been recently accessed are quick to access again.

This makes the recently accessed item to be accessible in O(**??**) time if accessed again.

Splay tree operations

It support all of the typical binary tree operations such as insertion, search, and deletion.

The sub operation splaying tree that makes all of the operations possible.

Splaying

It is what keeps the splay tree roughly balanced. To splay a node, splaying steps are repeated until it moves to the top of the tree.

There are three possibilities to perform splaying:

- The node's parent is the root.

  We only need to perform one rotation to make it the root.

  This is referred as "zig" or "zag" case.
- The node is the left child of a right child (or the right child of a left child).

  We need to perform two rotations.

  This is referred as "zig-zag" or "zag-zig" case.
- The node is the left child of a left child (or the right child of a right child).

  There is also two rotations.

  It is referred as "zig-zig" or "zag-zag" case.

Search

The search operation does the standard BST search, in addition to search, the found node also splays.

If the search is successful, then the node that is found is moved to top and becomes the new root. Else the last node accessed prior to reaching the NULL is splayed and becomes new root.

Search operation not only brings the searched key to root, but also balances the binary search tree.

There are following cases for node being accessed.

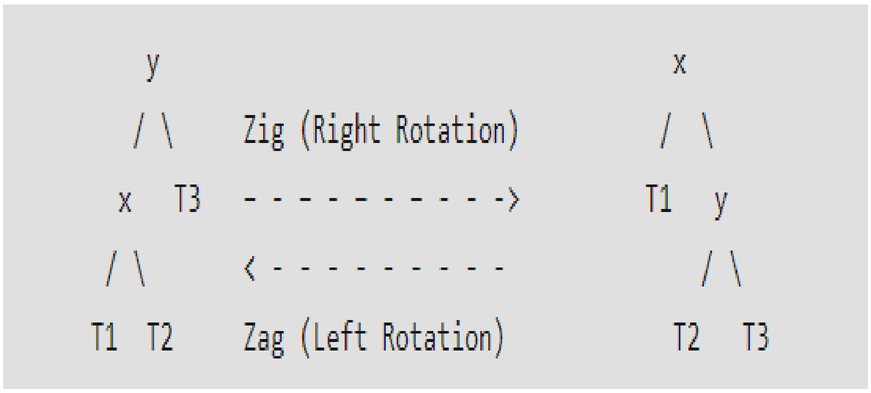
- Node is root

  We can simply return the root without doing anything else as the accessed node is already root

- Zig: Node is child of root

  The node has no grandparent.

  Node is either a left child of root (do right rotation) or right child of root (do left rotation).



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

- Node has both parent and grandparent

  Zig-Zig and Zag-Zag

  Node is left child of parent and parent is also left child of grandparent (do two right rotations) or node is right child of parent and parent is also right child of grandparent (do 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
```

  Zig-Zig and Zag-Zag

  Node is left child of parent and parent is right child of grandparent (do left rotation followed by right rotation) or node is right child of parent and parent is left child of grandparent (do 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
```

Insert

To insert a node, find its appropriate location at the bottom of the tree using binary search and perform splaying for that node to bring top of the tree.

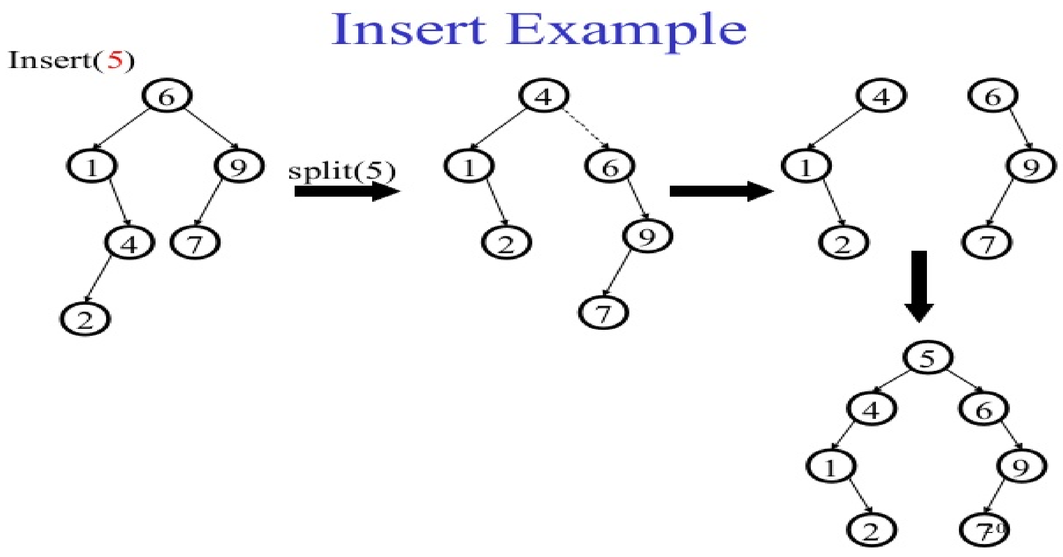
Then split the tree into two seperated trees, then insert the new node as the new root and merge the seperated trees with the new root.

There are different cases to insert a key:

- Root is null:

  We simply allocate new node and return it as root.
- Splay the given key. If the k is already present, becomes the new root. Else the last accessed leaf node becomes the new root
- If the root is same as the key, do not anything as key is already present.
- Else allocate the memory for new node and compares with root and key.
- If key is smaller than new root, then make the root as right child of new node and copy the left child of root as left child of new node and make the left child of root as NULL.
- If key is greater than new root, then make the root as left child of new node and copy the right child of root as right child of new node and make the right child of root as NULL.
- Return the root as new node.

Example:



Delete

The node to be deleted is first splayed for making it the root node. Then the node is deleted. We are left with two seperated trees that are joined with together.

There are some different cases to delete a key from tree:

- If the root is NULL:

  We simply return the root

- Splay the given key. If the k is already present, becomes the new root. Else the last accessed leaf node becomes the new root

- If the root is not same as the key, then return the key is not present.

- Else the key is present.

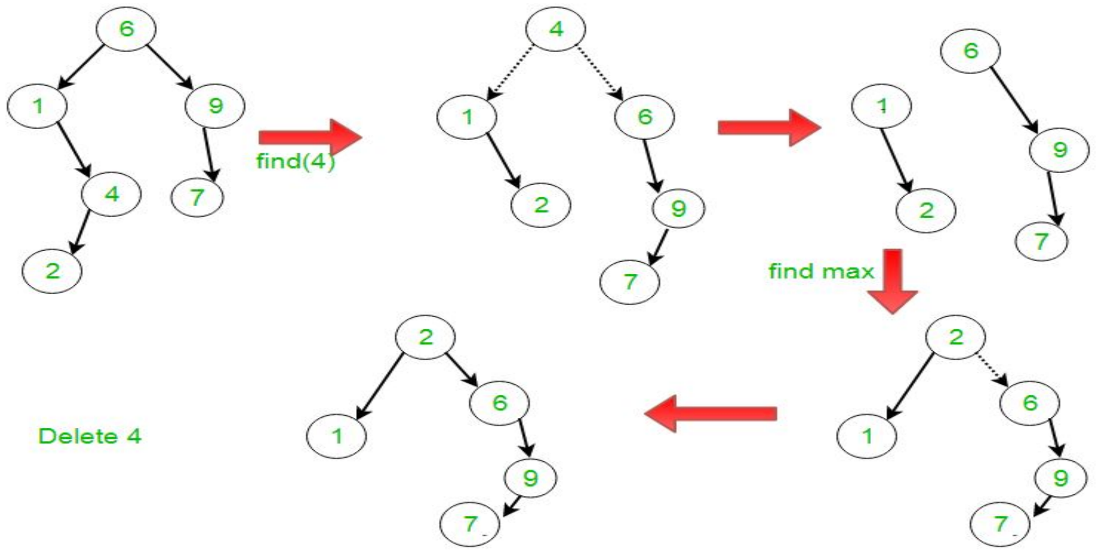
  Split the tree into two trees without root. Tree1 is the left sub tree and Tree2 is the right sub tree. Root1 and Root2 are the roots of the Tree1 and Tree2 respectively. Then delete the root node.

  If the Root1 is NULL, then return Root2.

  Else the splay the maximum node of Root1.

  After that, make the Root2 as the right child of Root1 and return Root1.

Example:



Advantages

- The splay tree's main advantage is that it keeps the most frequent nodes at the top of the tree. It is decreasing the time for subsequent queries. It can be accessed more quickly.
- Spay tree have low memory overhead. This makes them attractive for memory sensitive. It do not need to store any bookkeeping data.
- Average case performance is as efficient as other trees.
- The splay tree operations run in O(log n) time on average. Where n is the number of entries in the tree.

Application

- Network router

  A network router receives network packets at a high rate from incoming connection. It must quickly decide on which outgoing wire to send each packet, based on the IP address in the packet.
- Implementation of caches, memory allocators, garbage collectors, data compression, ropes (replacement of string used for long text strings), in Windows NT (in the virtual memory, networking, and file system code).
- Intrusion detection systems (IDS)

  They are essential part of security infrastructure. They are used to detect, identify and stop intruders.
- gcc compiler and GNU C++ library, the sed string editor, the most popular implementation of Unix malloc, Linux loadable kernel modules, and in much other sotftware.