12/19/2022

Splay Tree

Quiz 3 Equivalent

Name: K.M. TAHLIL MAHFUZ FARUK

Student ID: 200042158

Course: CSE 4303(Data Structure)

K.M. Tahlil Mahfuz Faruk
ISLAMIC UNIVERSITY OF TECHNOLOGY

Splay Tree:

A Splay tree is a special kind of binary search tree. It is roughly height balanced. It is self-balanced binary search tree. It has splay property. That means if we search a particular node, splay property makes the node the root of the binary search tree maintaining all the properties of a binary search tree.

Splaying is applied using left rotation and right rotation aka. Zig and Zag rotation. It is same as the rotation of AVL tree.

BIT Standard Operations:

A binary indexed tree can have the following standard operations.

- Insert
- Left Rotate(Zig rotation)
- Right Rotate(Zag rotation)
- Splay
- Search
- Preorder
- Inorder
- Postorder

Advantages:

- ❖ Searching can be done more efficiently than BST.
- ❖ Each searching roughly balances it height.
- ❖ The nodes closer to the root can be searched more efficiently.
- ❖ Performance improves.

Disadvantages:

- ❖ Splay tree can have linear height after the splaying property also.
- ❖ Linear height will have O(n) time complexity, it doesn’t improve the searching time.
- ❖ It will take a lot of rotation during the operations of a splay tree.

Sample code of a basic implementation of Splay Tree:

```
#include<bits/stdc++.h>
using namespace std;

class node
{
    public:

    int data;
    node* right;
    node* left;

    node()
    {
        right=NULL;
        left=NULL;
    }
    node(int x)
    {
        right=NULL;
        left=NULL;
        data=x;
    }
};

class SplayTree{
    public:
    node* root=NULL;

    node* rightRotate(node* x){
        node* y =x->left;
        x->left=y->right;
        y->right=x;
        return y;
    }
    node* leftRotate(node* x){
        node* y =x->right;
        x->right=y->left;
        y->left=x;
        return y;
    }

    node* splay(node *r,int key)
    {
```

```
        if(r==NULL || r->data==key) return r;
        if (r->data>key)
        {
            if(r->left == NULL) return r;
            if(r->left->data > key)
            {
                r->left->left=splay(r->left->left, key);
                r=rightRotate(r);
            }
            else if(r->left->data < key)
            {
                r->left->right=splay(r->left->right,key);
                if(r->left->right!=NULL)r->left=leftRotate(r->left);
            }
            if(r->left==NULL){
                return r;
            }
            else return rightRotate(r);
        }
        else
        {
            if(r->right==NULL) return r;
            if(r->right->data>key)
            {
                r->right->left=splay(r->right->left,key);
                if (r->right->left!=NULL)r->right=rightRotate(r->right);
            }
            else if(r->right->data<key)
            {
                r->right->right=splay(r->right->right,key);
                r=leftRotate(r);
            }
            if(r->right==NULL){
                return r;
            }
            else return leftRotate(r);
        }
    }

    node* insert(node *r,int k)
    {
        if (r == NULL)return new node(k);
        r = splay(r, k);
        if (r->data == k) return r;
        node *newnode = new node(k);
```

```
    if (r->data > k)
    {
        newnode->right = r;
        newnode->left = r->left;
        r->left = NULL;
    }
    else
    {
        newnode->left = r;
        newnode->right = r->right;
        r->right = NULL;
    }
    return newnode;
}

string search(node* r,int key)
{
    node* t=splay(r,key);
    root=t;
    if(t->data==key){return "True";}
    else return "False";
}

node* delete_key(node* r,int key)
{
    node *temp;
    if (!r) return NULL;
    r = splay(r, key);
    if (key != r->data) return r;
    if (!r->left)
    {
        temp = r;
        r = r->right;
    }
    else
    {
        temp = r;
        r = splay(r->left, key);
        r->right = temp->right;
    }
    delete temp;
    return r;
}
void preorder(node *t)
{
```

```
        if(t == NULL)return;
        cout<<" "<<t->data<<" ";
        preorder(t->left);
        preorder(t->right);
    }
    void inorder(node *t)
    {
        if(t == NULL)return;
        preorder(t->left);
        cout<<" "<<t->data<<" ";
        preorder(t->right);
    }
    void postorder(node *t)
    {
        if(t == NULL)return;
        preorder(t->left);
        preorder(t->right);
        cout<<" "<<t->data<<" ";
    }
};


signed main()
{
    SplayTree st;
    int n;cin>>n;
    vector<int>v(n);
    for(auto &e:v){
        cin>>e;
        st.root=st.insert(st.root,e);
    }
    st.preorder(st.root);
    st.root=st.delete_key(st.root,2);
    st.preorder(st.root);
    cout<<st.search(st.root,2)<<endl; //False
    cout<<st.search(st.root,10)<<endl;//True
    cout<<st.search(st.root,122)<<endl;//False

    return 0;
}
```

Complexity Analysis:

Time Complexity

Operation	Best case	Worst Case
Insertion	O(logn)	O(n)
Deletion	O(logn)	O(n)
Searching	O(logn)	O(n)
Splay	O(logn)	O(n)

Space Complexity

Space Complexity	**O(N)**

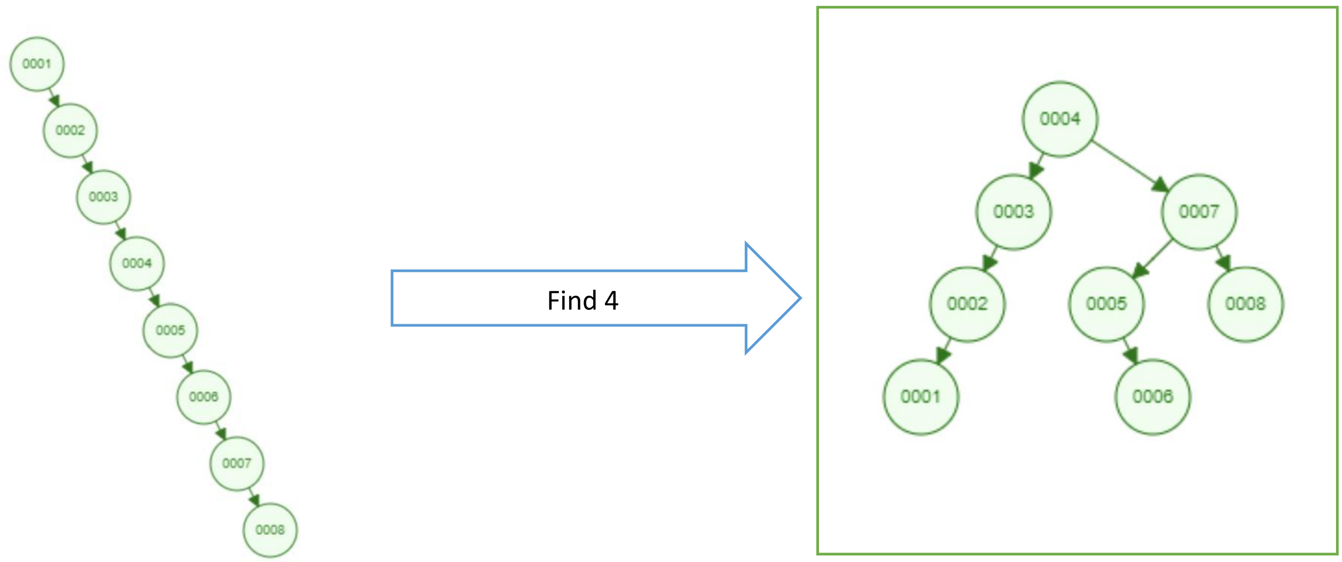
Time complexity will differ in terms of best cases and worst cases.

Best Case Complexity: O(logN)

Worst Case Complexity: O(N)

Use cases:

Suppose we have a set of numbers that is {1,2,3,4,5,6,7,8}. If we insert it in the splay tree then it will look like this-



At first the BST had a O(N) complexity for searching as it has linear complexity. But using splay tree this complexity can be optimized.

As we can see in searching the height becomes O(N) to O(logN) in terms of splay tree search. Now if we search 5 it will take o(logN) time complexity. This demonstration implies that in each search the closer node to the root becomes more efficient to search.

So, in those cases of BST where searching could take O(N) that can be optimized to O(logN) using splay tree.