CSE4303: DATA STRUCTURES

ASSIGNMENT ON
SPLAY TREE

NAME: Chowdhury Ashfaq
STUDENT ID: 200042123
PROGRAM: BSc. In SWE
DEPARTMENT: CSE
SUBMISSION DATE: 20/12/2022

What is a Splay Tree?

A roughly Balanced Binary Indexed Tree which can access recently accessed element quickly is Splay Tree.
Splay Tree has a special operation Splay which ensures that the element accessed recently remains near the root. As they remain near the root so the elements accessed recently or most frequently can be accessed in almost constant time. This is the speciality of a splay tree and it is used in many important functions of a machine.

What are the operations we can perform on a Standard Splay Tree?

The operations that can be performed on a Splay Tree are almost similar to a Binary Search Tree. But here we have one extra operation. The operations are:

1. Searching.
2. Insertion.
3. Deletion.
4. Splaying.

Advantages of Splay Tree

Splay Tree is a self-balancing Binary Search Tree. AVL tree, Red Black Tree are also self-balancing binary search trees. So why'd we use Splay Tree instead of using them? The reasons are:

1. Better performance as frequently accessed node will be nearer to the root.
2. Requires least time for searching in practical applications.
3. Very easy to implement.
4. No extra information is stored.
5. Easy to understand.

Disadvantages of Splay Tree

No data structure comes without any disadvantage. There are a few disadvantage of Splay Tree which are given below:

1. The tree isn't strictly balanced which means in worst case scenario the height of the tree can be O(n). So, accessing can take O(n) time which is a very rare case.
2. If the access pattern is random then it's wasteful to use Splay tree as it would increase cost.

- ## Sample Code of a basic implementation of Splay Tree

The implementation of Binary Indexed Tree in C++ have been given below:

```cpp
#include <iostream>
using namespace std;


class node
{
public:
    int key;
    node* left, * right;
};

node* newNode(int key)
{
    node* Node = new node();
    Node->key = key;
    Node->left = Node->right = NULL;
    return (Node);
}

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* root, int key)
{
    if (root == NULL || root->key == key)
        return root;

    if (root->key > key)
    {
        if (root->left == NULL) return root;

        if (root->left->key > key)
        {
            root->left->left = splay(root->left->left, key);
```

```
            root = rightRotate(root);
        }
        else if (root->left->key < key)
        {
            root->left->right = splay(root->left->right, key);

            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }

        return (root->left == NULL) ? root : rightRotate(root);
    }
    else
    {
        if (root->right == NULL) return root;

        if (root->right->key > key)
        {
            root->right->left = splay(root->right->left, key);

            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key)
        {
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }

        return (root->right == NULL) ? root : leftRotate(root);
    }
}

node* search(node* root, int key)
{
    return splay(root, key);
}

void preOrder(node* root)
{
    if (root != NULL)
    {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

- ## Complexity Analysis

For searching, in best case Splay Tree would take O(1) or constant time. On average case the access time is O(log n) which is like AVL Tree and RB Tree. Worst case scenario would take O(n) time when the tree is skewed.

For insertion and Deletion the average case time complexity is O(log n) and worst case time complexity is O(n).

- ## Use cases

Splay Tree has a few use cases which are:

1. In caches and GCC compilers Splay tree is used.

A problem where Splay Tree can be used is:

A tree has M vertices. An integer value is associated with each node. Q queries are given. A query can be in either of the two forms T s v or W s v.
A query of the form T s v means the order of all the values on the path between vertices s and v should be reversed.
A query of the form W s v means the sum of all the values on the path between vertices s and v should be the output.