AISSMS
INSTITUTE OF INFORMATION TECHNOLOGY
ADDING VALUE TO ENGINEERING

Department Of Computer Engineering

Data Structure And Algorithms Lab

Group-D

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AISSMS IOIT

SE COMPTER ENGINEERING

SUBMITTED BY

Kaustubh S Kabra
ERP No.- 34
Teams No.-20

2020 -2021

Experiment Number:-8

- **Aim:-** *Given sequence k = k1 <k2 < … <kn of n sorted keys, with a search probability pi for each key ki . Build the Binary search tree that has the least search cost given the access probability for each key.*

- **Objective:-**

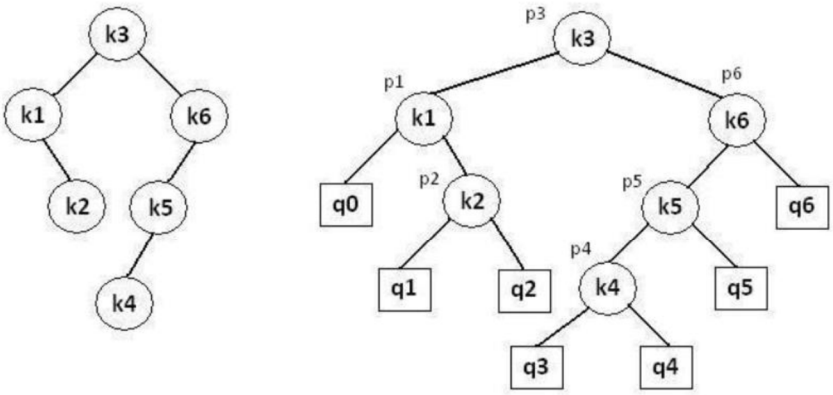
1. *To understand concept of OBST.*
2. *To understand concept & features like extended binary search tree.*

- **Theory:-**

An optimal binary search tree is a binary search tree for which the nodes are arranged on levels such that the tree cost is minimum.

For the purpose of a better presentation of optimal binary search trees, we will consider "extended binary search trees", which have the keys stored at their internal nodes. Suppose "n" keys k1, k2, … k n are stored at the internal nodes of a binary search tree. It is assumed that the keys are given in sorted order, so that k1< k2 < … < kn.

An extended binary search tree is obtained from the binary search tree by adding successor nodes to each of its terminal nodes as indicated in the following figure by squares:



Binary search tree

Extended binary search tree

In the extended tree:

1) *The squares represent terminal nodes. These terminal nodes represent unsuccessful searches of the tree for key values. The searches did not end successfully, that is, because they represent key values that are not actually stored in the tree;*
2) *The round nodes represent internal nodes; these are the actual keys stored in the tree;*
3) *Assuming that the relative frequency with which each key value is accessed is known, weights can be assigned to each node of the extended tree (p1 … p6). They represent the relative frequencies of searches terminating at each node, that is, they mark the successful searches.*
4) *If the user searches a particular key in the tree, 2 cases can occur:*
5) *1 – the key is found, so the corresponding weight „p" is incremented;*
6) *2 – the key is not found, so the corresponding „q" value is incremented.*

GENERALIZATION:

The terminal node in the extended tree that is the left successor of k1 can be interpreted as representing all key values that are not stored and are less than k1. Similarly, the terminal node in the extended tree that is the right successor of kn, represents all key values not stored in the tree that are greater than kn. The

terminal node that is successes between ki and ki-1 in an inorder traversal represent all key values not stored that lie between ki and ki - 1.

Algorithm:-

1) *Create a class for OBST and declare all the arrays required.*
2) *Initialize front and rear to -1.*
3) *Accept the number of nodes and the data to be present in each node from the user and add it to an array.*
4) *Accept the possibilities for successful and unsuccessful searches and store them in two different arrays.*
5) *Create a method to calculate the OBST and store the elements according to their order in a 2-D array.*
6) *Display the root node of the optimal binary search tree.*
7) *Display the left and right nodes of the OBST.*
8) *STOP*

Program:-

```
#include<iostream>

# define SIZE 10

using namespace std;

class Optimal

{

        private:

                int p[SIZE];

                int q[SIZE];
```

```
        int a[SIZE];
        int w [SIZE][SIZE];
        int c [SIZE][SIZE];
        int r [SIZE][SIZE];
        int n;
        int front, rear, queue[20];

    public:
        Optimal();
        void get_data();
        int Min_Value(int,int);
        void OBST();
        void build_tree();
};

Optimal::Optimal()
{
    front=rear=-1;
}

void Optimal::get_data()
{
    int i;
```

```
cout<<"\n Optimal Binary Search Tree \n";
cout<<"\n Enter the number of nodes: ";
cin>>n;
cout<<"\n Enter the data as .... \n";
for(i=1;i<=n;i++)
{
        cout<<"\n a["<<i<<"]";
        cin>>a[i];
}

cout<<"\n The probabilities for successful search .... \n";
for(i=1;i<=n;i++)
{
        cout<<"p["<<i<<"]";
        cin>>p[i];
}

cout<<"\n The probabilities for unsuccessful search .... \n";
for(i=0;i<=n;i++)
{
        cout<<"q["<<i<<"]";
        cin>>q[i];
}
```

```
}

int Optimal::Min_Value(int i, int j)
{
        int m,k;
        int minimum=32000;
        for(m=r[i][j-1];m<=r[i+1][j];m++)
        {
                if((c[i][m-1]+c[m][j])<minimum)
                {
                        minimum=c[i][m-1]+c[m][j];
                        k=m;
                }
        }
        return k;
}

void Optimal::OBST()
{
        int i,j,k,m;
        for(i=0;i<n;i++)
        {
                w[i][j]=q[i];
```

```
            r[i][i]=c[i][i]=0;
            w[i][i+1]=q[i]+q[i+1]+p[i+1];
            r[i][i+1]=i+1;
            c[i][i+1]=q[i]+q[i+1]+p[i+1];
        }
        w[n][n]=q[n];
        r[n][n]=c[n][n]=0;

        for(m=2;m<=n;m++)
        {
            for(i=0;i<=n-m;i++)
            {
                j=i+m;
                w[i][j]=w[i][j-1]+p[j]+q[j];
                k=Min_Value(i,j);
                c[i][j]=w[i][j]+c[i][k-1]+c[k][j];
                r[i][j]=k;
            }
        }
}

void Optimal::build_tree()
{
```

```
int i,j,k;
cout<<"The Optimal Binary Search Tree for the given node is: \n";
cout<<"\n The Root of this OBST is:: "<<r[0][n];
cout<<"\n The cost of this OBST is:: "<<c[0][n];
cout<<"\n\n\tNODE\tLEFT CHILD\tRIGHT CHILD";
cout<<"\n----------------------------------------------------"<<endl;
queue[++rear]=0;
queue[++rear]=n;
while(front!=rear)
{
        i=queue[++front];
        j=queue[++front];
        k=r[i][j];
        cout<<"\n\t"<<k;
        if(r[i][k-1]!=0)
        {
                cout<<"                 "<<r[i][k-1];
                queue[++rear]=i;
                queue[++rear]=k-1;
        }
        else
        {
                cout<<"                 -";
```

```
                }
                if(r[k][j]!=0)
                {
                        cout<<"                 "<<r[k][j];
                        queue[++rear]=k;
                        queue[++rear]=j;
                }
                else
                {
                        cout<<"                 -";
                }
        }
        cout<<endl;
}
int main()
{
        Optimal obj;
        obj.get_data();
        obj.OBST();
        obj.build_tree();
        return 0;
}
```

Output:-

Optimal Binary Search Tree

Enter the number of nodes: 4

Enter the data as ....

a[1] 1

a[2] 2

a[3] 3

a[4] 4

The probabilities for successful search ....

p[1] 3

p[2] 3

p[3] 1

p[4] 1

The probabilities for unsuccessful search ....

q[0] 2

q[1] 3

q[2] 1

q[3] 1

q[4] 1

The Optimal Binary Search Tree for the given node is:

The Root of this OBST is:: 2

The cost of this OBST is:: 32

NODE	*LEFT CHILD*	*RIGHT CHILD*
2	*1*	*3*
1	-	-
3	-	*4*
4	-	-

Analysis:-

Time Complexity: The algorithm requires O (n^3) time, since three nested for loops are used. Each of these loops takes on at most n values.

Space Complexity: The algorithm requires O(n^3) space.

Conclusion: – *Thus, we also have studied OBST and implemented it. We have built the Binary search tree that has the least search cost given the access probability for each key.*