Assignment No 1(D)

Title:

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?

Objectives:

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

Learning Objectives:

- ✓ To understand concept of OBST.
- ✓ To understand concept & features like extended binary search tree.

Learning Outcome:

- ✓ Define class for Extended binary search tree using Object Oriented features.
- ✓ Analyze working of functions.

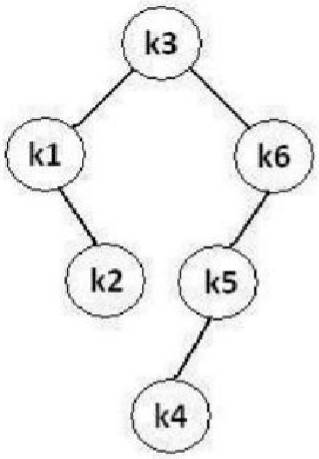
Software Required: g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

Theory:

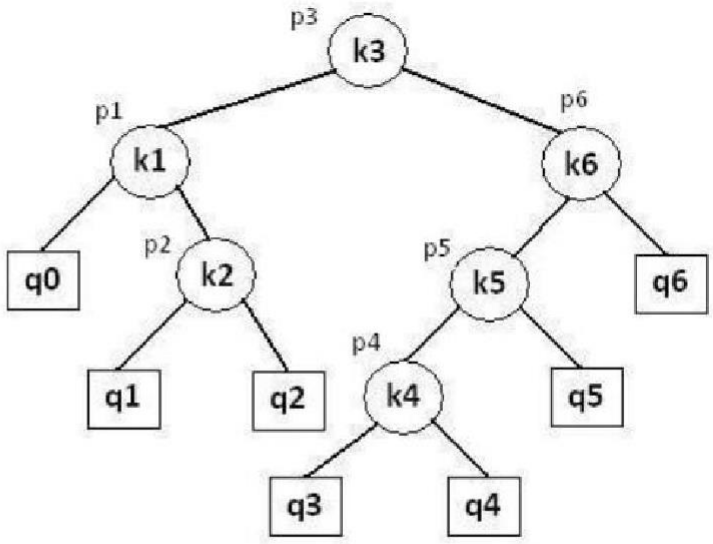
An optimal binary search tree is a binary search tree for which the nodes are arranged on levels such that the tree cost is minimum.

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:

- 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;
- The round nodes represent internal nodes; these are the actual keys stored in the tree;
- 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.
- If the user searches a particular key in the tree, 2 cases can occur:
- 1 – the key is found, so the corresponding weight „p" is incremented;
- 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.

ALGORITHMS

We have the following procedure for determining R(i, j) and C(i, j) with 0 <= i <= j <= n:

PROCEDURE COMPUTE_ROOT(n, p, q; R, C)

begin

for i = 0 to n do

C (i, i) ←0

W(i,i)←q(i) for m = 0 to n do
for i = 0 to (n – m) doj ←i + m
W (i, j) ←W (i, j – 1) + p (j) + q (j)
*find C (i, j) and R (i, j) which minimize thetree cost
end
The following function builds an optimal binary search tree
FUNCTION CONSTRUCT(R, i, j)
begin
*build a new internal node N labeled (i, j)k ←R (i, j)
f i = k then
*build a new leaf node N‟ labeled (i, i)else
*N‟ ←CONSTRUCT(R, i, k)
*N‟ is the left child of node Nif k = (j – 1)
then
*build a new leaf node N‟‟‟ labeled (j, j)else
*N‟‟‟ ←CONSTRUCT(R, k + 1, j)
*N‟‟‟ is the right child of node Nreturn N
end

COMPLEXITY ANALYSIS:

The algorithm requires O (n2) time and O (n2) storage. Therefore, as „n‟ increases it will run outof storage even before it runs out of time. The storage needed can be reduced by almost half by implementing the two-dimensional arrays as one-dimensional arrays.

Conclusion: This program gives us the knowledge OBST, Extended binary search tree.

Program :

```
#include<iostream>

using namespace std;

#define SIZE 10
class OBST
{
int p[SIZE];
int q[SIZE];
int a[SIZE];
int w[SIZE][SIZE];
//'r[i][j]'
int c[SIZE][SIZE];
int r[SIZE][SIZE];
int n;

public:

void 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];
        }
        for(i=1;i<=n;i++)
        {
                cout<<"\n p["<<i<<"]";
                cin>>p[i];
        }
        for(i=0;i<=n;i++)
        {
                cout<<"\n q["<<i<<"]";
                cin>>q[i];
        }
}

int 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   build_OBST()
{
   int i,j,k,l,m;
   for(i=0;i<n;i++)
   {
            //initialize
            w[i][i]=q[i];
            r[i][i]=c[i][i]=0;
            //Optimal  trees  with  one  node
            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;
   //Find optimal trees with 'm' nodes
   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;
            }
   }
}
/* This function builds the tree from the tables made by the OBST function */void
build_tree()
{
    int i,j,k;
    int queue[20],front=-1,rear=-1;
    cout<<"The Optimal Binary Search Tree For the Given Node
    Is…\n";cout<<"\n The Root of this OBST is ::"<<r[0][n];
    cout<<"\nThe Cost of this OBST is::"<<c[0][n];
    cout<<"\n\n\t NODE \t LEFT CHILD \t RIGHT
    CHILD ";cout<<"\n";
    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<<"\t\t"<<r[i][k-1];
                        queue[++rear]=i;
                        queue[++rear]=k-1;
                }
                else

                        cout<<"\t\t";
                if(r[k][j]!=0)
                {



                }
                Else
    cout<<"\t"<<r[k][j]; queue[++rear]=k;
    queue[++rear]=j;


    cout<"\t

}//end of whilecout<<"\n";
                }
                }
                /

                int main()
                {
                OBST obj;
                obj.get_data();
                obj.build_OBST();
                obj.build_tree();
                return 0;
                }
```