ASSINGMENT NO.	**8**
TITLE	To write a program for implementing optimal binary search tree
PROBLEM STATEMENT /DEFINITION	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 and implement optimal Binary search tree. 2. To understand procedure to create weight balanced tree.
OUTCOME	• Implement OBST (Weight balance tree) using Object Oriented features.
S/W PACKAGES AND HARDWARE APPARATUS USED	• 64-bit Open source Linux or its derivative. • Open Source C++ Programming tool like G++/GCC.
REFERENCES	Data structures in C++ by Horowitz, Sahni.
INSTRUCTIONS FOR WRITING JOURNAL	1. Date 2. Assignment no. 3. Problem definition 4. Learning objective 5. Learning Outcome 6. Concepts related Theory 7. Algorithm/Pseudo code 8. Test cases 9. Conclusion/Analysis

Prerequisites:

- Binary tree and its implementation
- Concept of weight balanced tree data structure

Learning Objectives:

- To understand concept of weight balanced tree data structure.
- To understand procedure to create weight balanced tree.
- To implement OBST Tree

Learning Outcomes:
After successful completion of this assignment, students will be able to
- Implement OBST using Object Oriented features.
- Analyze working of various operations on OBST Tree.

Concepts related Theory:

OBST:-The Binary search Tree is having minimum average cost of searching.

Example -Consider input sequence of 3 identifier (do, if, while) , what is its Optimal Binary Search Tree.

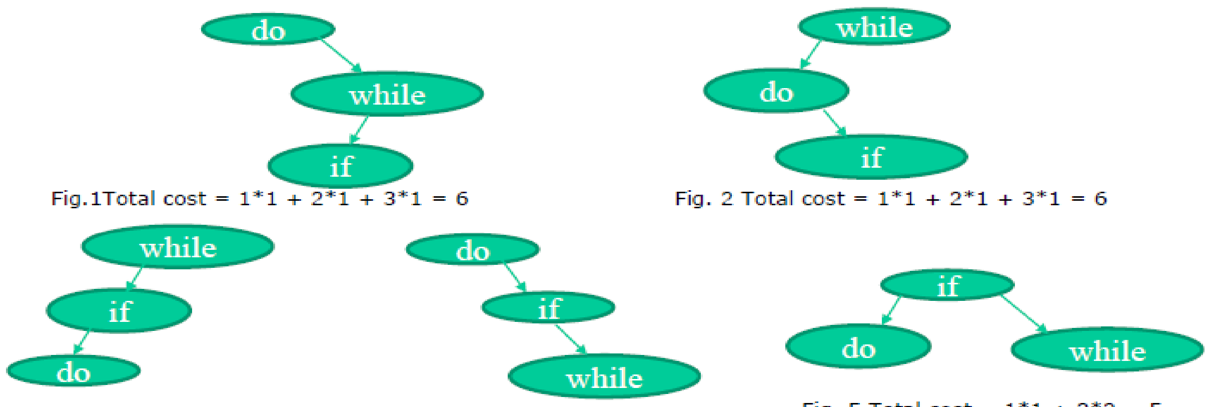

Fig.1Total cost = 1*1 + 2*1 + 3*1 = 6

Fig. 2 Total cost = 1*1 + 2*1 + 3*1 = 6

Fig. 3
Total cost = 1*1 + 2*1 +3*1 = 6

Fig. 4
Total cost = 1*1 + 2*1 +3*1 = 6

Fig. 5 Total cost = 1*1 + 2*2 = 5

Optimal Binary Search Tree extends the concept of Binary search tree. Binary Search Tree (BST) is a nonlinear data structure which is used in many scientific applications for reducing the search time. In BST, left child is smaller than root and right child is greater than root. This arrangement simplifies the search procedure.

Optimal Binary Search Tree (OBST) is very useful in dictionary search. The probability of searching is different for different words. OBST has great application in translation. If we translate the book from English to German, equivalent words are searched from English to German dictionary and replaced in translation. Words are searched same as in binary search tree order.

Binary search tree simply arranges the words in lexicographical order. Words like ‘the’, ‘is’, ‘there’ are very frequent words, whereas words like ‘xylophone’, ‘anthropology’ etc. appears rarely.

It is not a wise idea to keep less frequent words near root in binary search tree. Instead of storing words in binary search tree in lexicographical order, we shall arrange them according to their probabilities. This arrangement facilitates few searches for frequent words as they would be near the root. Such tree is called Optimal Binary Search Tree.

Consider the sequence of nkeys K = < k1, k2, k3, …, kn> of distinct probability in sorted order such that k1< k2< … <kn. Words between each pair of key lead to unsuccessful search, so for n keys, binary search tree contains n + 1 dummy keys di, representing unsuccessful searches.

Two different representation of BST with same five keys {k1, k2, k3, k4, k5} probability is shown in following figure

With n nodes, there exist (2n)!/((n + 1)! * n!) different binary search trees. An exhaustive search for optimal binary search tree leads to huge amount of time.

The goal is to construct a tree which minimizes the total search cost. Such tree is called optimal binary search tree. OBST does not claim minimum height. It is also not necessary that parent of sub tree has higher priority than its child.

Dynamic programming can help us to find such optima tree.

- We formulate the OBST with following observations
- Any sub tree in OBST contains keys in sorted order $k_i…k_j$, where $1 \le i \le j \le n$.
- Sub tree containing keys $k_i…k_j$ has leaves with dummy keys $d_{i-1}….d_j$.
- Suppose k_r is the root of sub tree containing keys $k_i…..k_j$. So, left sub tree of root k_r contains keys
  $k_i….k_{r-1}$ and right sub tree contain keys k_{r+1} to k_j. Recursively, optimal sub trees are constructed from the left and right sub trees of k_r.
- Let e[i, j] represents the expected cost of searching OBST. With n keys, our aim is to find and minimize e[1, n].
- Base case occurs when j = i – 1, because we just have the dummy key d_{i-1} for this case. Expected search cost for this case would be e[i, j] = e[i, i – 1] = q_{i-1}.
- For the case $j \ge i$, we have to select any key k_r from $k_i…k_j$ as a root of the tree.
- With k_r as a root key and sub tree $k_i…k_j$, sum of probability is defined as

$$w(i, j) = \sum_{m=i}^{j} p_m + \sum_{m=i-1}^{j} q_m$$

ADT:

```
class node{
int data;
node *lchild, *rchild;
friend class obst;
public:
node(int x)
{
data = x;
lchild= rchild= NULL;
}
};

class obst
{
class node *root;
public:
obst(){ root = null;}
void calculate_wt( double [], double [], int);
void create_tree(int,int)
};
```

```
Procedure calculate_wt(double *p, double *q, int n) {
  //p is array of probability of successful search, q is array of unsuccessful search, n is number of
  identifiers
            for (int i= 0 ; i< n ; i++)
            w[i][i] = q[i] ; r[i][i] = c[i][i] = 0; //calculate wts, costs of null tree and tree 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] = w[i][i+1];
            End for
            w[n][n] = q[n] ; r[n][n] = c[n][n] = 0;
            for (m = 2 ; m <= n ; m++) //calculate wt and cost of tree having more than one node
            for (int i= 0 ; i<= n-m ; i++)
            Min = 999; j= i+m;
            w[i][j] = w[i][j-1] +p[j] + q[j];
            for (int i1 = i+1 ; i1<j ; i1++)
            sum1 = c[i][i1-1] + c[i1][j];
            if (sum1<min) min = sum1; k= i1; end if
            End for
            C[i][j] = w[i][j] + c[i][k-1] + c[k][j]
            r[i][j] = k;
            End for
            Root = Create_tree(0,n)
            }
```

Review Questions:

1. What is OBST tree?
2. In an OBST tree, how the weights are found?
3. What is the difference between height balanced and weight balanced tree?