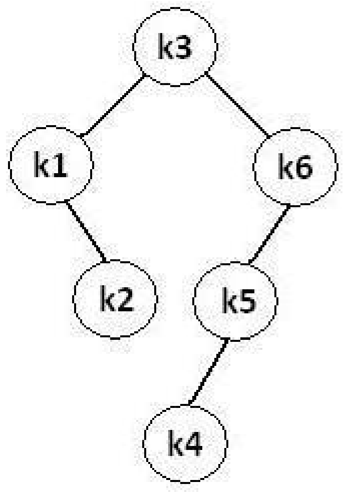
OPTIMAL BINARY SEARCH TREES

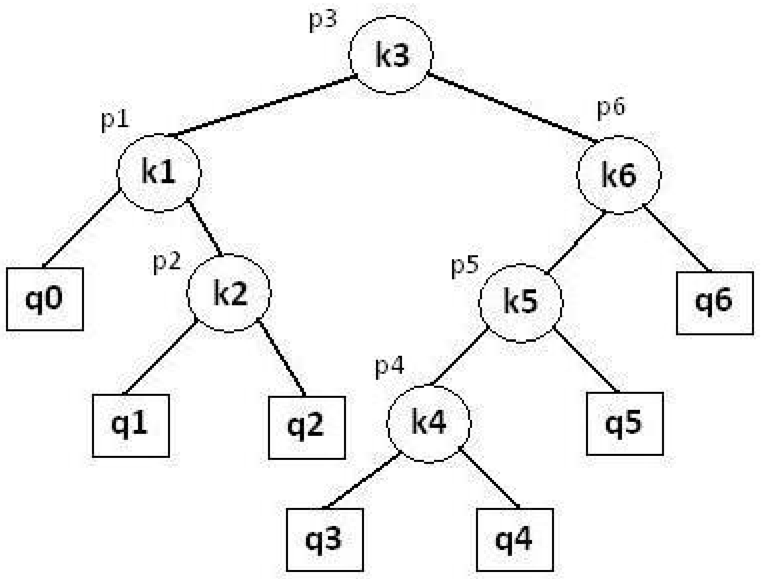
1. PREPARATION BEFORE LAB
DATA STRUCTURES

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 k_1, k_2, … , 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 $k_1 < k_2 < … < k_n$. 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 k_1 can be interpreted as representing all key values that are not stored and are less than k_1. Similarly, the terminal node in the extended tree that is the right successor of k_n, represents all key values not stored in the tree that are greater than k_n. The terminal node that is successed between k_i and k_{i-1} in an inorder traversal represents all key values not stored that lie between k_i and k_{i-1}.

EXAMPLE:

In the extended tree in the above figure if the possible key values are 0, 1, 2, 3, …, 100 then the terminal node labeled q_0 represents the missing key values 0, 1 and 2 if k_1=3. The terminal node labeled q_3 represents the key values between k_3 and k_4. If k_3=17 and k_4=21 then the terminal node labeled q_3 represents the missing key values 18, 19 and 20. If k_6 is 90 then the terminal node q_6 represents the missing key values 91 through 100.

An obvious way to find an optimal binary search tree is to generate each possible binary search tree for the keys, calculate the weighted path length, and keep that tree with the smallest weighted path length. This search through all possible solutions is not feasible, since the number of such trees grows exponentially with “n”.

An alternative would be a recursive algorithm. Consider the characteristics of any optimal tree. Of course it has a root and two subtrees. Both subtrees must themselves be optimal binary search trees with respect to their keys and weights. First, any subtree of any binary search tree must be a binary search tree. Second, the subtrees must also be optimal.

Since there are "n" possible keys as candidates for the root of the optimal tree, the recursive solution must try them all. For each candidate key as root, all keys less than that key must appear in its left subtree while all keys greater than it must appear in its right subtree. Stating the recursive algorithm based on these observations requires some notations:

- OBST(i, j) denotes the optimal binary search tree containing the keys ki, ki+1, …, kj;
- $W_{i,j}$ denotes the weight matrix for OBST(i, j)
  $W_{i,j}$ can be defined using the following formula:
$$W_{i,j} = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k$$
- $C_{i,j}$, $0 \le i \le j \le n$ denotes the cost matrix for OBST(i, j)
  $C_{i,j}$ can be defined recursively, in the following manner:
  $C_{i,i} = W_{i,j}$
  $C_{i,j} = W_{i,j} + \min_{i<k\le j}(C_{i,k-1} + C_{k,j})$
- $R_{i,j}$, $0 \le i \le j \le n$ denotes the root matrix for OBST(i, j)
  Assigning the notation $R_{i,j}$ to the value of k for which we obtain a minimum in the above relations, the optimal binary search tree is OBST(0, n) and each subtree OBST(i, j) has the root $k_{R_{ij}}$ and as subtrees the trees denoted by OBST(i, k-1) and OBST(k, j).

**OBST(i, j) will involve the weights qi-1, pi, qi, …, pj, qj.*

All possible optimal subtrees are not required. Those that are consist of sequences of keys that are immediate successors of the smallest key in the subtree, successors in the sorted order for the keys.

The bottom-up approach generates all the smallest required optimal subtrees first, then all next smallest, and so on until the final solution involving all the weights is found. Since the algorithm requires access to each subtree's weighted path length, these weighted path lengths must also be retained to avoid their recalculation. They will be stored in the weight matrix 'W'. Finally, the root of each subtree must also be stored for reference in the root matrix 'R'.

Algorithms in Pseudocode

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) do
             j ← i + m
             W (i, j) ← W (i, j – 1) + p (j) + q (j)
             *find C (i, j) and R (i, j) which minimize the tree 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)

     if 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 N
     if 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 N
     return N
end
```

Example of Running the Algorithm

Find the optimal binary search tree for N = 6, having keys k_1 … k_6 and weights $p_1 = 10$, $p_2 = 3$, $p_3 = 9$, $p_4 = 2$, $p_5 = 0$, $p_6 = 10$; $q_0 = 5$, $q_1 = 6$, $q_2 = 4$, $q_3 = 4$, $q_4 = 3$, $q_5 = 8$, $q_6 = 0$. The following figure shows the arrays as they would appear after the initialization and their final disposition.

Initial array values:

R	0	1	2	3	4	5	6
0		1					
1			2				
2				3			
3					4		
4						5	
5							6
6							

W	0	1	2	3	4	5	6
0	5	21	28	41	46	54	64
1		6	13	26	31	39	49
2			4	17	22	30	40
3				4	9	17	27
4					3	11	21
5						8	18
6							0

C	0	1	2	3	4	5	6
0							
1							
2							
3							
4							
5							
6							

The values of the weight matrix have been computed according to the formulas previously stated, as follows:

W (0, 0) = q0 = 5
W (1, 1) = q1 = 6
W (2, 2) = q2 = 4
W (3, 3) = q3 = 4
W (4, 4) = q4 = 3
W (5, 5) = q5 = 8
W (6, 6) = q6 = 0

W (0, 1) = q0 + q1 + p1 = 5 + 6 + 10 = 21
W (0, 2) = W (0, 1) + q2 + p2 = 21 + 4 + 3 = 28
W (0, 3) = W (0, 2) + q3 + p3 = 28 + 4 + 9 = 41
W (0, 4) = W (0, 3) + q4 + p4 = 41 + 3 + 2 = 46
W (0, 5) = W (0, 4) + q5 + p5 = 46 + 8 + 0 = 54
W (0, 6) = W (0, 5) + q6 + p6 = 54 + 0 + 10 = 64
W (1, 2) = W (1, 1) + q2 + p2 = 6 + 4 + 3 = 13

--- and so on ---
until we reach:
W (5, 6) = q5 + q6 + p6 = 18

The elements of the cost matrix are afterwards computed following a pattern of lines that are parallel with the main diagonal.

C (0, 0) = W (0, 0) = 5
C (1, 1) = W (1, 1) = 6
C (2, 2) = W (2, 2) = 4
C (3, 3) = W (3, 3) = 4
C (4, 4) = W (4, 4) = 3
C (5, 5) = W (5, 5) = 8
C (6, 6) = W (6, 6) = 0

C	0	1	2	3	4	5	6
0	5						
1		6					
2			4				
3				4			
4					3		
5						8	
6							0

C (0, 1) = W (0, 1) + (C (0, 0) + C (**1**, 1)) = 21 + 5 + 6 = 32
C (1, 2) = W (0, 1) + (C (1, 1) + C (**2**, 2)) = 13 + 6 + 4 = 23
C (2, 3) = W (0, 1) + (C (2, 2) + C (**3**, 3)) = 17 + 4 + 4 = 25
C (3, 4) = W (0, 1) + (C (3, 3) + C (**4**, 4)) = 9 + 4 + 3 = 16
C (4, 5) = W (0, 1) + (C (4, 4) + C (**5**, 5)) = 11 + 3 + 8 = 22
C (5, 6) = W (0, 1) + (C (5, 5) + C (**6**, 6)) = 18 + 8 + 0 = 26

**The bolded numbers represent the elements added in the root matrix.*

C	0	1	2	3	4	5	6
0	5	32					
1		6	23				
2			4	25			
3				4	16		
4					3	22	
5						8	26
6							0

R	0	1	2	3	4	5	6
0		1					
1			2				
2				3			
3					4		
4						5	
5							6
6							

C (0, 2) = W (0, 2) + min (C (0, 0) + C (**1**, 2), C (0, 1) + C (2, 2)) = 28 + min (**28**, 36) = 56
C (1, 3) = W (1, 3) + min (C (1, 1) + C (2, 3), C (1, 2) + C (**3**, 3)) = 26 + min (31, **27**) = 53
C (2, 4) = W (2, 4) + min (C (2, 2) + C (**3**, 4), C (2, 3) + C (4, 4)) = 22 + min (**20**, 28) = 42
C (3, 5) = W (3, 5) + min (C (3, 3) + C (4, 5), C (3, 4) + C (**5**, 5)) = 17 + min (26, **24**) = 41
C (4, 6) = W (4, 6) + min (C (4, 4) + C (5, 6), C (4, 5) + C (**6**, 6)) = 21 + min (29, **22**) = 43

C	0	1	2	3	4	5	6
0	5	32	56				
1		6	23	53			
2			4	25	42		
3				4	16	41	
4					3	22	43
5						8	26
6							0

R	0	1	2	3	4	5	6
0		1	1				
1			2	3			
2				3	3		
3					4	5	
4						5	6
5							6
6							

Final array values:

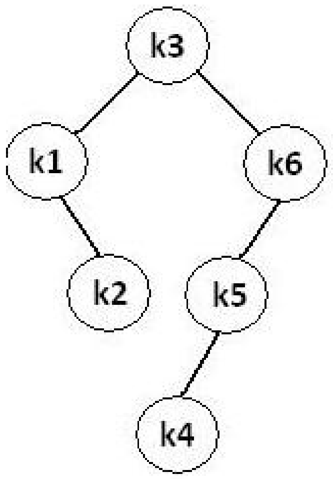
C	0	1	2	3	4	5	6
0	5	32	56	98	118	151	188
1		6	23	53	70	103	140
2			4	25	42	75	108
3				4	16	41	68
4					3	22	43
5						8	26
6							0

R	0	1	2	3	4	5	6
0	0	1	1	2	3	3	3
1		0	2	3	3	3	3
2			0	3	3	3	4
3				0	4	5	6
4					0	5	6
5						0	6
6							0

The resulting optimal tree is shown in the bellow figure and has a weighted path length of 188.

Computing the node positions in the tree:
- The root of the optimal tree is R(0, 6) = k3;
- The root of the left subtree is R(0, 2) = k1;
- The root of the right subtree is R(3, 6) = k6;
- The root of the right subtree of k1 is R(1, 2) = k2
- The root of the left subtree of k6 is R(3, 5) = k5
- The root of the left subtree of k5 is R(3, 4) = k4

Thus, the optimal binary search tree obtained will have the following structure:



COMPLEXITY ANALYSIS

The algorithm requires O (n^2) time and O (n^2) storage.
Therefore, as ‘n’ increases it will run out of 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.

2. Sample coding

```
#include <stdio.h>
#include<stdlib.h>
#define NMAX 20
```

```
typedef struct OBST
{
        int KEY;
        struct OBST *left, *right;
}
OBST;

int C[NMAX][NMAX]; //cost matrix
int W[NMAX][NMAX]; //weight matrix
int R[NMAX][NMAX]; //root matrix
int q[NMAX]; //unsuccesful searches
int p[NMAX]; //frequencies
int NUMBER_OF_KEYS; //number of keys in the tree
int KEYS[NMAX];
OBST *ROOT;

void COMPUTE_W_C_R()
{
     int x, min;
     int i, j, k, h, m;

       //Construct weight matrix W
     for(i = 0; i <= NUMBER_OF_KEYS; i++)
     {
           W[i][i] = q[i];
           for(j = i + 1; j <= NUMBER_OF_KEYS; j++)
                 W[i][j] = W[i][j-1] + p[j] + q[j];
     }

       //Construct cost matrix C and root matrix R
     for(i = 0; i <= NUMBER_OF_KEYS; i++)
           C[i][i] = W[i][i];
     for(i = 0; i <= NUMBER_OF_KEYS - 1; i++)
     {
           j = i + 1;
           C[i][j] = C[i][i] + C[j][j] + W[i][j];
           R[i][j] = j;
     }
     for(h = 2; h <= NUMBER_OF_KEYS; h++)
           for(i = 0; i <= NUMBER_OF_KEYS - h; i++)
           {
                 j = i + h;
                 m = R[i][j-1];
                 min = C[i][m-1] + C[m][j];
                 for(k = m+1; k <= R[i+1][j]; k++)
                 {
                       x = C[i][k-1] + C[k][j];
                       if(x < min)
                       {
                            m = k;
                            min = x;
                       }
                 }
                 C[i][j] = W[i][j] + min;
                 R[i][j] = m;
           }
```

```
            //Display weight matrix W
        printf("\nThe weight matrix W:\n");
              for(i = 0; i <= NUMBER_OF_KEYS; i++)
              {
                for(j = i; j <= NUMBER_OF_KEYS; j++)
                          printf("%d  ", W[i][j]);
                printf("\n");
              }

              //Display Cost matrix C
              printf("\nThe cost matrix C:\n");
              for(i = 0; i <= NUMBER_OF_KEYS; i++)
              {
                for(j = i; j <= NUMBER_OF_KEYS; j++)
                          printf("%d  ", C[i][j]);
                    printf("\n");
              }

              //Display root matrix R
              printf("\nThe root matrix R:\n");
              for(i = 0; i <= NUMBER_OF_KEYS; i++)
              {
                for(j = i; j <= NUMBER_OF_KEYS; j++)
                          printf("%d  ", R[i][j]);
                    printf("\n");
              }
}

//Construct the optimal binary search tree
OBST *CONSTRUCT_OBST(int i, int j)
{
      OBST *p;

      if(i == j)
              p = NULL;
      else
      {
          p = new OBST;
          p->KEY = KEYS[R[i][j]];
          p->left = CONSTRUCT_OBST(i, R[i][j] - 1); //left subtree
          p->right = CONSTRUCT_OBST(R[i][j], j); //right subtree
      }
      return p;
}

//Display the optimal binary search tree
void DISPLAY(OBST *ROOT, int nivel)
{
     int i;
     if(ROOT != 0)
     {
            DISPLAY(ROOT->right, nivel+1);
            for(i = 0; i <= nivel; i++)
                  printf("      ");
            printf("%d\n", ROOT->KEY);
            DISPLAY(ROOT->left, nivel + 1);
```

```
    }
}

void OPTIMAL_BINARY_SEARCH_TREE()
{
    float average_cost_per_weight;

    COMPUTE_W_C_R();
    printf("C[0] =  %d W[0] = %d\n", C[0][NUMBER_OF_KEYS],
W[0][NUMBER_OF_KEYS]);
    average_cost_per_weight =
C[0][NUMBER_OF_KEYS]/(float)W[0][NUMBER_OF_KEYS];
    printf("The cost per weight ratio is: %f\n", average_cost_per_weight);
    ROOT = CONSTRUCT_OBST(0, NUMBER_OF_KEYS);
}

int main()
{
    int i, k;

    printf("Input number of keys: ");
    scanf("%d", &NUMBER_OF_KEYS);

      for(i = 1; i <= NUMBER_OF_KEYS; i++)
      {
            printf("key[%d]= ",i);
            scanf("%d", &KEYS[i]);
            printf(" frequency = ");
            scanf("%d",&p[i]);
      }

      for(i = 0; i <= NUMBER_OF_KEYS; i++)
      {
            printf("q[%d] = ", i);
            scanf("%d",&q[i]);
      }

      while(1)
      {
            printf("1.Construct tree\n2.Display tree\n3.Exit\n");
            scanf("%d", &k);
            switch(k)
            {
                  case 1:
                        OPTIMAL_BINARY_SEARCH_TREE();
                        break;
            case 2:
                DISPLAY(ROOT, 0);
                break;
                  case 3:
                        exit(0);
                break;
            }
      }
    system("PAUSE");
}
```

3. Assignments

Problems that use the presented algorithms

- Write a program that creates a binary tree, whose node data structure contains an additional field called 'COUNTER'; use the tree search procedure which counts each searched element; after a certain number of searches, the 'COUNTER' fields are used as weights for the reconstruction of the tree under the form of an optimal tree. For each terminal node there will be used two additional nodes for storing the unsuccessful searches.
- Add a SEARCH () function to the sample algorithm in order to enable the user to search for a certain key in the constructed optimal binary search tree. If the key is found, increase its corresponding frequency 'p'; otherwise, if the search ends unsuccessfully, increase the corresponding 'q' value. Note that after the search, certain modifications may appear in the tree. Compute the matrices W, C and R again, in order to make sure the structure of the tree is correct.

Other functionalities

- Suppose that we are designing a program to simulate the storage and search in a dictionary. Words appear with different frequencies, however, and it may be the case that a frequently used word such as "the" appears far from the root while a rarely used word such as "conscientiousness" appears near the root. We want words that occur frequently in the text to be placed nearer to the root. Moreover, there may be words in the dictionary for which there is no definition. Organize an optimal binary search tree that simulates the storage and search of words in a dictionary.