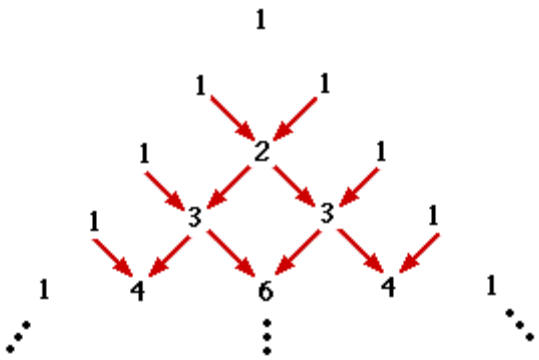
Programming and Data Structures Course - Jan to Mar 2015

Programming Assignments

Assignment 1: Pascal's triangle

You must have heard of Pascal's triangle. In this programming assignment, your program is required to compute the sum of a given set of elements of the Triangle. Most importantly, your program should have a **recursive function** to calculate the values in the triangle.

As you know, the values in the Pascal's Triangle are called Binomial Coefficients. At the top tip of the arrangement is number 1, which makes up the **zeroth** row. The first row contains two 1's **. In general, in each row, the first and the last numbers are 1.** To find any number in the *i* th row, add the two numbers immediately above it in *(i-1)*th row, as shown in the diagram below.



INPUT & OUTPUT:

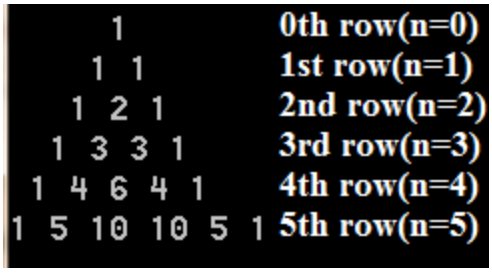
The input consists of multiple lines.
The **first line** contains a number *n* which indicates that the number of rows in the Pascal's

triangle will be *n+1* (Note that rows of Pascal's triangle are indexed starting with *0* at the top and the elements in a row are also indexed starting with a *0*).

The **second line** contains a number *m* which indicates the number of ***transactions*** to be performed on the Pascal's triangle. Each transaction is given in a separate line. A transaction is a space separated list of integers. The first integer in each list indicates the row number, say R, and the rest of the integers in the list indicate the indices of values in row R. For each transaction, you have to compute the sum of given coefficients in the given row R.

Example: Input will be given in the following format:
5
3
3 1 2
5 1 1 1 4
4 2 3 2

- Here, the First line contains number 5. So there will be 6 rows in the corresponding Pascal's triangle as shown below:
-



- The Second line of the input indicates that we must perform 3 transactions.
-
- The Third line contains a space separated list i.e, 3 1 2 - the output for this line should be the sum of 1st coefficient and 2nd coefficient in the 3rd row of the triangle (thus output for this transaction should be ${}^3C_1+{}^3C_2 = 6$).
- Similarly we can compute the output for the other two transactions listed in lines 4 and 5.

The output must be a space separated sequence of integers terminated by a new-line. For each transaction there must be 1 integer in the output list that corresponds to the sum of specified coefficients in that transaction. If the transaction has an invalid input, i.e, when any

of the indices are out of bound or the row number is greater than *n*, then the output must be a -1.

Thus output for the given input would, thus, be:
6 20 16

Sample Input :
6
4
7 1 2 2 3
5 0 1 1 4
4 2 3 2
6 4 5 3 2 1

Sample Output :
-1 16 16 62

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int pascal(int,int);
void space(int,int);

int main()
{
      int n,i,j,m;
      int iToken, iPascal, iRowNum, iRowSum, iNumOps;
      int flag;
      scanf("%d",&n);
      scanf("%d",&m);
      int sum[m];
      for(i=0;i<m;i++){
            sum[i]= 0;
      }
      for(i=0;i<m;i++){
          scanf("%d",&iRowNum);
          scanf("%d",&iNumOps);
          flag=0;
```

```
          if(iRowNum > n){
            flag=1;
          }
          iRowSum = 0;
          for (j=0;j<iNumOps;j++){
            scanf("%d",&iToken);
            if (iToken > iRowNum) flag=1;
            iPascal = pascal(iRowNum,iToken);
            iRowSum = iRowSum + iPascal;
          }
          if (flag==1) sum[i]= -1;
          else sum[i]=iRowSum;
      }
      for(i=0;i<m;i++){
           printf("%d ",sum[i]);
      }
      return 0;
}


int pascal(int row,int column)
{
      if(column==0 || row==column)
            return 1;
      else if(column>row)
            return 0;
      else
            return (pascal(row-1,column-1)+pascal(row-1,column));
}
```

Assignment 2: Polynomial Multiplication

This programming exercise is to give you exposure to the advantage of maintaining linked lists. You should use the doubly linked list studied in the Week 3, first lecture. The exercise is univariate polynomial multiplication. Recall that a univariate polynomial is a polynomial over a single variable, and in the following discussion when we say polynomial, we mean a polynomial in a single variable. You are given two polynomials – the first one, M1(x), is called the multiplicand, and the second one, M2(x), is called the multiplier. Your program has to output the product of M1(x) and M2(x). You could use the functions implemented in the lectures to maintain your lists corresponding to the polynomials. Below, you find a formal description of the inputs and outputs.

INPUT & OUTPUT:

The input will be four lines in the format specified below:

Line 1: Number of terms in multiplicand
e.g: 2

Line 2: Coefficients and exponents in the multiplicand in the format ***coeff 1 exp1 coeff2 exp2***
e.g: 2 2 1 0 which represents 2x^2+1

Line 3: Number of terms in multiplier.
e.g: 3

Line 4: Coefficients and exponents in the multiplier in the format ***coeff 1 exp1 coeff2 exp2 ....***
e.g: -23 12 11 10 -6 41 which represents -6x^41-23x^12+11x^10

Each line will have multiple sets of **coeff exp** pairs, each corresponding to a non zero coefficient in the corresponding polynomial. The output has to be a single polynomial in 'x' that is the product polynomial. Note that, in the input, for each exponent, there may be multiple terms with the same exponent. However you should print a single term for each exponent while you print the output. Also, if the resulting product is zero, you have to explicitly print a zero.

The output has to be displayed as in the sample output shown with decreasing value of exponent.

Sample Input:
2
2 1 1 0

2
3 2 2 0

Sample Output:
6x^3+3x^2+4x+2

Solution:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct container {
      int value;
      int index;
      struct container *next;
} node;


void printPoly(node *p)
{
      int atleastOne = 0;
      while(p)
      {
            if(p->value != 0)
            {
                  atleastOne = 1;
                  if(p->value == 1)
                  {
                        if(p->index == 0)
                              printf("%d", p->value);
                  }
                  else
                        printf("%d", p->value);
                  if(p->index > 1)
                        printf("x^%d", p->index);
                  else if(p->index == 1)
                        printf("x");
                  while(p->next && p->next->value == 0)
                        p = p->next;
                  if(p->next != NULL && p->next->value >= 0)
                        printf("+");
```

```
            }
            p = p->next;
      }
      if(atleastOne == 0)
            printf("0");
      printf("\n");
}

int isLast(node * list, node * position)
{
if (position->next == NULL) return 1;
return 0;
}

node * after(node * list, node * position)
{
if (list == NULL || isLast(list,position) ) return NULL;

/* if list is empty or if position in the list is last, then
t is invalid, as there is no successor */

while (list != position)
      list = list->next;

/* skip till you reach position, and then return the next element */
return list->next;
}

void insertAfter(node * list, node * position, int value, int index)
/* note that we are not returning anything */
{
node * following;
node * new_node; /* to store the address of a new node */

if (position == NULL) return;

/* the query is now meaningful, let us allocate space for the new
value */

new_node = (node *) malloc(sizeof(node));
new_node->value = value;
new_node->index = index;
following = after(list,position);
```

```
if (following == NULL)
        {position->next = new_node;
     new_node->next = NULL;
         return ;
        }
/* Insert after position, which is the last position, and list is
longer */
new_node->next = following;
position->next = new_node;

/* note that new_node has been inserted after position and the list
is longer */

return ;
/* return a pointer to the newly added node, if position is
well-defined */
}

node * last(node * list)
{
if (list == NULL) return list;
while(list->next != NULL)
     list = list->next;

/* skip till the last node is reached and then return it */
return list;
}

void insertLast( node * list, int value , int index)
{
insertAfter(list, last(list), value, index);
/* again, the calling functions have to be careful to check that
first is not NULL, else
this functions returns no information about the failure to insert
value */
return;
}


node * search(node * p, int ind)
{
     while(p)
     {
```

```
            if(ind < p->index)
            {
                  p = p->next;
            }
            else
                  return p;
      }
      return NULL;
}

node * getBefore(node *p, node *t)
{
      if(p == t)
            return NULL;
      while(p->next != t)
            p = p->next;
      return p;
}
node * readPoly(int size)
{
      node *p, *temp, *temp1;
      int i, val, ind;
      for(i = 0; i < size; i++)
      {
            scanf("%d %d", &val, &ind);
            if(i == 0)
            {
                  p = (node *) malloc(sizeof(node));
                  p->value = val;
                  p->index = ind;
                  p->next = NULL;
            }
            else
            {
                  temp = search(p, ind);
                  if(temp == NULL)
                        insertLast(p, val, ind);
                  else if(temp->index == ind)
                        temp->value += val;
                  else
                  {
                        temp1 = getBefore(p, temp);
                        if(temp1 == NULL)
```

```
                        {
                              p = (node *) malloc(sizeof(node));
                              p->value = val;
                              p->index = ind;
                              p->next = temp;
                        }
                        else
                        {
                              temp1->next = (node *)
malloc(sizeof(node));
                              temp1 = temp1->next;
                              temp1->value = val;
                              temp1->index = ind;
                              temp1->next = temp;
                        }
                  }
            }
      }
      return p;
}

node * mul(int val, int ind, node *p)
{
      node *temp, *new = NULL;
      while(p)
      {
            if(new == NULL)
            {
                  new = (node *) malloc(sizeof(node));
                  new->value = p->value * val;
                  new->index = p->index + ind;
                  new->next = NULL;
            }
            else
                  insertLast(new,p->value * val, p->index + ind);
            p = p->next;
      }
      return new;
}

node * sum(node *p1, node *p2)
{
      node *new = NULL;
```

```
int val, ind;
while(p1 && p2)
{
      if(p1->index == p2->index)
      {
            val = p1->value + p2->value;
            ind = p1->index;
            p1 = p1->next;
            p2 = p2->next;
      }
      else if(p1->index > p2->index)
      {
            val = p1->value;
            ind = p1->index;
            p1 = p1->next;
      }
      else
      {
            val = p2->value;
            ind = p2->index;
            p2 = p2->next;
      }
      if(new == NULL)
      {
            new = (node *) malloc(sizeof(node));
            new->value = val;
            new->index = ind;
            new->next = NULL;
      }
      else
            insertLast(new, val, ind);
}
while(p1)
{
      if(new == NULL)
      {
            new = (node *) malloc(sizeof(node));
            new->value = p1->value;
            new->index = p1->index;
            new->next = NULL;
      }
      else
            insertLast(new, p1->value, p1->index);
```

```
            p1 = p1->next;
      }
      while(p2)
      {
            if(new == NULL)
            {
                  new = (node *) malloc(sizeof(node));
                  new->value = p2->value;
                  new->index = p2->index;
                  new->next = NULL;
            }
            else
                  insertLast(new, p2->value, p2->index);
            p2 = p2->next;
      }
      return new;
}

int main()
{
      int s1, s2;
      int val, ind;
      int i, j;
      node *m1, *m2, *temp1, temp2, *res, *ans = NULL;
      scanf("%d", &s1);
      m1 = readPoly(s1);
      scanf("%d", &s2);
      m2 = readPoly(s2);
      temp1 = m1;
      while(temp1)
      {
            res = mul(temp1->value, temp1->index, m2);
            //multiply each term of m1 with all the terms of m2
            ans = sum(ans, res);
            //Maintain the intermediate sum in a polynomial
            temp1 = temp1->next;
      }
      printPoly(ans);
}
```

Assignment 3: Print strings from grid

In this assignment the input consists of a grid of size M X N. Each cell in the grid contains a lower case letter of the English alphabet. In a natural way, the cells are of two types: boundary cells and internal cells. Each internal cell in the grid has four neighbours in one of the left, right, top, down directions. A string of characters can be formed by starting at any cell and traversing the grid through the neighbours. You have to print all the possible strings subject to the following constraints:

**No two characters in a string can be same
**No two strings can be same in the final output
**The strings should be printed in alphabetically sorted order.

INPUT:

First line contains two integers M and N
Next M lines contains N space separated characters each

OUTPUT:

Print all possible strings in sorted order and obeying the above constraints.

INPUT SIZE:
1 <= M, N <= 20

SAMPLE INPUT:
2 2
a b
a c

SAMPLE OUTPUT:
a ab abc ac acb b ba bc bca c ca cb cba

Solution:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct list{
      char *str;
      struct list *next;
}List;

int presentInString(char ch, char *str)
{
      while(*str != '\0')
      {
            if(*str == ch)
                  return 1;
            str++;
      }
      return 0;
}

int presentInList(char *str, List *l)
{
      while(l)
      {
            if(strcmp(l->str, str) == 0)
                  return 1;
            l = l->next;
      }
      return 0;
}

List *sort(List *head)
{
      List *res = NULL;
      int len = 0;
      List *temp = head;
      while(temp)
      {
            len++;
            temp = temp->next;
```

```
	}
	for(int i = 0; i < len; i++)
	{
		List *max = head, *temp = head;
		while(temp)
		{
			if(strcmp(max->str, temp->str) <= 0)
				max = temp;
			temp = temp->next;
		}
		List *new = (List *)malloc(sizeof(List));
		new->str = (char *)malloc(strlen(max->str) + 1);
		char *chr = max->str;
		char *t = new->str;
		while(*chr)
		{
			*t = *chr;
			t++;
			chr++;
		}
		*t = '\0';
		max->str[0] = '\0';
		new->next = res;
		res = new;
	}
	return res;
}

void printStrings(int m, int n, char grid[][n], int i, int j, List
**head, char str[])
{
	int nexti, nextj;
	int nxt[4][2];			// nxt to get the indices of next
neighbors
	nxt[0][0] = 0;
	nxt[0][1] = 1;
	nxt[1][0] = 0;
	nxt[1][1] = -1;
	nxt[2][0] = 1;
	nxt[2][1] = 0;
	nxt[3][0] = -1;
	nxt[3][1] = 0;
```

```
	if(!presentInString(grid[i][j], str))
	{
		int len = strlen(str);
		str[len] = grid[i][j];
		str[len+1] = '\0';
		if(!presentInList(str, *head))
		{
			List *temp = (List *)malloc(sizeof(List));
			temp->str = (char *)malloc(strlen(str)+1);
			char *chr = str;
			char *t = temp->str;
			while(*chr)
			{
				*t = *chr;
				t++;
				chr++;
			}
			*t = '\0';
			temp->next = *head;
			*head = temp;
		}
	}
	for(int x = 0; x < 4; x++)
	{
		nexti = i + nxt[x][0];
		nextj = j + nxt[x][1];
		if(nexti >= 0 && nexti < m && nextj >= 0 && nextj < n &&
					!presentInString(grid[nexti][ne
				xtj], str))
			printStrings(m, n, grid, nexti, nextj, head, str);
	}
	str[strlen(str) - 1] = '\0';
}


int main()
{
	List *head = NULL;
	int m, n;
	scanf("%d %d\n", &m, &n);
	char grid[m][n];
	char str[100];
```

```
	str[0] = '\0';
	for(int i = 0; i < m; i++)
	{
		for(int j = 0; j < n; j++)
			scanf("%c ", &grid[i][j]);
	}
	for(int i = 0; i < m; i++)
	{
		for(int j = 0; j < n; j++)
			printStrings(m, n, grid, i, j, &head, str);
	}
	head = sort(head);
	List *temp = head;
	while(temp)
	{
		printf("%s ", temp->str);
		temp = temp->next;
	}
	return 0;
}
```

Assignment 4: Maintain a complete binary tree

A **complete binary tree** of depth D is defined as follows:
-- in every level except possibly the last level, is full.
-- in each level 'i', the nodes are labeled with values from 2^i upto 2^{i+1}-1, starting with the leftmost node which gets the least value, and the rightmost node gets the largest value.
-- if a node at level D-1 has at most one child, then all nodes at level D-1 with larger label have no children.

In this exercise, your program has to maintain a complete binary tree with integer values in every node. It must support the following functions:

** **Insert**: Insert a given key into the complete binary without changing the labels of any of the other nodes, and the tree must remain complete. After inserting, the program has to output the inorder traversal sequence of the new tree.

** **Remove**: Remove all occurrences of the given key from the tree. While Removing, follow the rules given below:
---- The key to be removed has to be replaced by the key that is in the node with the largest label in the tree. After replacement, the node with the largest label is deleted from the tree.
---- If there are multiple nodes in the tree with the same key, the input key in the node with the least label must be removed first.
---- After the removal is complete, the program must output the inorder traversal sequence of the new tree.

** **Query**: The program has to return the number of occurrences of the input key.

** **Size**: Given a key, your program has to return the number of nodes in the subtree rooted at the key value. To make things simpler, such a key will be unique. That is, there will not be multiple occurrences of such a key.

Input:

-- The first line gives the number of items in the initial tree.

-- The next line of the input gives the key values for the tree. You may use the approach described in the lectures to create the tree using the values provided. The values have to be inserted in the same order as is being input, filling each level one after the other from left to right.

-- The third line indicates the number of operations to be performed on the tree.

-- From fourth line onwards, each line will specify the operation to be performed along with key(s). The operations will use the following symbols:

* i : insert
* r : remove
* q : query
* s : size

Output:

-- The lines of output has to be in the exact same order as the input.

-- The first line of output should be the inorder traversal sequence of the constructed tree.

-- The result for each operation has to be displayed in a new line.

-- For insert and remove operations, the inorder traversal sequence of the new tree must be printed.

-- For query, it must return the number of occurrences of the key. If the key is absent, return 0.

-- For size, it has to display the number of nodes in the subtree rooted at the key value including the node itself. If the input value is in a leaf node, output 1. If the input value is in the root, return the size of the whole tree.

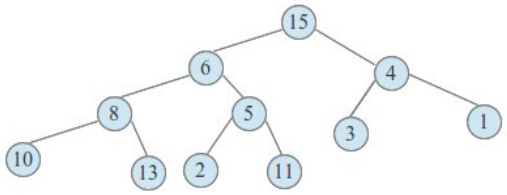
Sample Input 1:

```
11
15 6 4 8 5 3 1 10 13 2 11
4
i 9
r 8
q 4
```

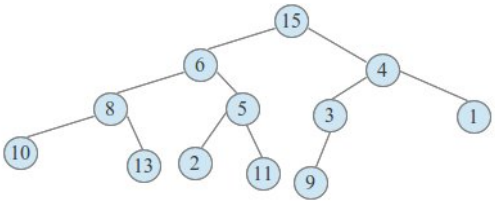
s 5

Sample Output 1:

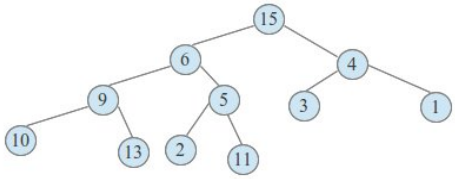
After construction :



After insert:



After remove:



10 8 13 6 2 5 11 15 3 4 1
10 8 13 6 2 5 11 15 9 3 4 1
10 9 13 6 2 5 11 15 3 4 1
1
3

Solution: (using linked lists, may also be done using arrays)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

typedef struct node{
    int data;
    int label;
    struct node * parent;
    struct node * rightChild;
    struct node * leftChild;
}Node;

typedef struct tree{
    Node *root;
    int size;
}BinaryTree;

BinaryTree *t;
Node *foundNode,*foundDataNode;
int maxData,flag;
int numOccur,countSubTree;

BinaryTree * createTree(){
    BinaryTree *t = (BinaryTree *)malloc(sizeof(BinaryTree));
    t->root = NULL;
    t->size = 0;
    return t;
}

Node * createNode(int data){
    Node *new1 = (Node *)malloc(sizeof(Node));
    new1->data = data;
    new1->label = t->size +1;
    new1->leftChild = NULL;
    new1->rightChild = NULL;
    new1->parent = NULL;
    return new1;
}
```

```
int size(BinaryTree *tr){
    return tr->size;
}

void setMax(Node *n){

    if(n == NULL)
          return;
     setMax(n->leftChild);
    if (n->label == t->size) {
          maxData = n->data;
     }
     setMax(n->rightChild);
}

void insert(Node *n, Node *parent, int position){

    if(position)
          parent->rightChild = n;
    else
          parent->leftChild = n;
    n->parent = parent;
     t->size = t->size + 1;
}

void findData (Node *temp, int iData){
     if (temp == NULL) return;
     findData (temp->leftChild,iData);
     if(temp->data == iData && foundDataNode == NULL){

          foundDataNode = temp;
     }
     findData (temp->rightChild,iData);
}

void deleteLastNode(Node *n, int key){
          if (n == NULL) return;
          deleteLastNode(n->leftChild,key);
          if (n->label == (int)(floor)(t->size/2) &&
n->rightChild!=NULL && flag == 0){
               flag = 1;
```

```
                n->rightChild = NULL;
                t->size = t->size - 1;
                setMax(t->root);
          }
          else if (n->label == (int)(floor)(t->size/2) &&
n->rightChild==NULL&& flag == 0){
                flag = 1;
                n->leftChild = NULL;
                t->size = t->size - 1;
                setMax(t->root);
          }
          deleteLastNode(n->rightChild,key);
}

void query(Node *n,int key){
    if(n == NULL)
          return;
     query(n->leftChild,key);
    if (n->data==key) numOccur++;
     query(n->rightChild,key);
}

void removeNode (Node* n, int key){
     int i;

     query(t->root,key);
     for (i=0;i<numOccur;i++){
          foundDataNode = NULL;
          findData(t->root,key);
          foundDataNode->data = maxData;
          flag=0;
          deleteLastNode(t->root,key);
     }
     numOccur = 0;
}


void countNodes(Node *temp){
     if (temp == NULL) return;
     countNodes (temp->leftChild);
     countSubTree++;
     countNodes (temp->rightChild);
}
```

```
void sizeTree(Node *n,int key){
      foundDataNode = NULL;
      findData(t->root,key);
      countSubTree = 0;
      countNodes(foundDataNode);
}

void inOrder(Node *root){
    if(root == NULL)
      return;
      inOrder(root->leftChild);
      printf("%d ", root->data);
      inOrder(root->rightChild);
}


void preOrder(Node *root){
    if(root == NULL)
      return;
    printf("%d ", root->data);
    preOrder(root->leftChild);
    preOrder(root->rightChild);
}


void postOrder(Node *root){
    if(root == NULL)
      return;
    postOrder(root->leftChild);
    postOrder(root->rightChild);
    printf("%d ", root->data);
}

void find (Node *temp,int iLabel){
      if (temp == NULL) return;
      find (temp->leftChild,iLabel);
      if(temp->label == iLabel && foundNode == NULL){
            foundNode = temp;
            return;
      }
      find (temp->rightChild,iLabel);
}
```

```
int main(){
     int num,m,i,j,array[MAX],ch,q;
     char *token;
     char *operation;
     char op;
    int iNum;
    Node *leftTemp, *rightTemp, *rootNode, *tempNode;


     scanf("%d ",&num);
     for(i=0;i<num;i++)
        scanf("%d ", &array[i]);
     t = createTree();
     foundNode = (Node *)malloc(sizeof(Node));
     for(i=0;i<num;i++){
          if(t->size==0){
                         rootNode = createNode(array[i]);
                         t->root = rootNode;
                         t->size = 1;
                    }
                    else if(t->size==1){
                         tempNode = createNode(array[i]);
                         tempNode->label = 2;
                         t->root->leftChild = tempNode;
                         t->size = 2;
                    }
                    else{
                         tempNode = createNode(array[i]);
                         maxData = tempNode->data;
                         if( (t->size % 2)==0){

                               foundNode = NULL;
                               find(t->root,t->size/2);
                               insert(tempNode,foundNode,1);
                               foundNode = NULL;

                         }
                         else{
                               find(t->root,(t->size + 1)/2);
                               insert(tempNode, foundNode, 0);
                               foundNode = NULL;
```

```
                    }
                }
      }
scanf("%d ",&m);
inOrder(t->root);

for(i=0;i<m;i++){
      scanf("%c ",&op);
      scanf("%d ",&iNum);

    switch(op)
    {
          case 'i' : //insert
                if(t->size==0){
                      rootNode = createNode(iNum);
                      t->root = rootNode;
                      t->size = 1;
                }
                else if(t->size==1){
                      tempNode = createNode(iNum);
                      tempNode->label = 2;
                      t->root->leftChild = tempNode;
                      t->size = 2;
                }
                else{
                      tempNode = createNode(iNum);
                      maxData = tempNode->data;
                      if( (t->size % 2)==0){

                            foundNode = NULL;
                            find(t->root,t->size/2);
                            insert(tempNode,foundNode,1);
                            foundNode = NULL;

                      }
                      else{
                            find(t->root,(t->size + 1)/2);
                            insert(tempNode, foundNode, 0);
                            foundNode = NULL;

                      }
                }
                inOrder(t->root);
```

```
                    //printf(" " );
                break;

            case 'r' :// delete
                removeNode(t->root,iNum);
                inOrder(t->root);
                //printf(" " );
                break;
            case 'q' ://query
                numOccur = 0;
                query(t->root,iNum);
                printf("%d ", numOccur);
                break;
            case 's' ://size
                sizeTree(t->root,iNum);
                printf("%d ", countSubTree);
                countSubTree = 0;
                break;
            default :
                printf("Invalid input\n" );
        }

    }

}
```

Assignment 5: Maintain a Binary Search Tree

This exercise is similar to the one in Unit 6. Here, your program has to maintain a Binary Search Tree or a BST, which has been taught in the lectures under Unit 7. All keys in the BST will be unique.

Your program must support the following functions:

** **Insert**: Insert a given key into the BST such that the tree remains a BST. That is, the node has to be inserted at the appropriate position so that a binary search can be done during lookup of a key. If the input key is already present in the tree, it must print the postorder sequence of the existing tree.
** **Remove**: Remove a given key from the tree. In case of a node having both children replace it with inorder successor.
** **Query**: The program has to return the depth of a node, given root is at depth 0.
** **Size**: Given a key, your program has to return the number of nodes in the subtree rooted at the key value.

Input:

** The first line gives the number of items in the initial tree.
** The next line of the input gives the key values for the tree. You may use the approach described in the lectures to create the tree using the values provided. The values have to be inserted in the same order as is being input.
** The third line indicates the number of operations to be performed on the tree.
** From fourth line onwards, each line will specify the operation to be performed along with key(s). The operations will use the following symbols:
------- i : insert
------- r : remove
------- q : query
------- s : size

Output:

** The lines of output has to be in the exact same order as the input.

** The first line of output should be the postorder traversal sequence of the constructed tree.
** The result for each operation has to be displayed in a new line.
** For insert and remove operations, the postorder traversal sequence of the new tree must be printed. If a duplicate key is found during insertion it must print the postorder sequence of the existing tree.
** For query, it must return the depth at which the key was found. If the key is absent, return -1.
** For size, it has to display the number of nodes in the subtree rooted at the key value including the node itself. If the input value is in a leaf node, output 1. If the input value is in the root, return the size of the whole tree.

Sample Input 1:

```
11
15 6 4 8 5 3 1 10 13 2 11
4
i 9
r 8
q 4
s 5
```

Sample Output 1:

```
2 1 3 5 4 11 13 10 8 6 15
2 1 3 5 4 9 11 13 10 8 6 15
2 1 3 5 4 9 11 13 10 6 15
2
1
```

Solution:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct bst{
```

```
      int data;
      struct bst * left;
      struct bst * right;
}BST;

BST * insert(BST *root, int val)
{
      if(!root)
      {
            BST *temp = (BST *)malloc(sizeof(BST));
            temp->data = val;
            temp->left = NULL;
            temp->right = NULL;
            return temp;
      }
      if(val < root->data)
            root->left = insert(root->left, val);
      else if(val > root->data)
            root->right = insert(root->right, val);
      return root;
}

void postOrder(BST *root){
      if(root)
      {
            postOrder(root->left);
            postOrder(root->right);
            printf("%d ", root->data);
      }
}

int size(BST * root){
      if(!root) return 0;
      return size(root->left) + size(root->right) + 1;
}

int findSize(BST *root, int val){
      while(root->data != val){
            if(val < root->data)
                  root = root->left;
            else
                  root = root->right;
      }
```

```
	return size(root);
}

int query(BST *root, int val)
{
	int count = 0;
	while(root && root->data != val)
	{
		if(val < root->data)
			root = root->left;
		else
			root = root->right;
		count++;
	}
	if(root)
		return count;
	return -1;
}

BST * succesor(BST *root)
{
	root = root->right;
	while(root->left)
		root = root->left;
	return root;
}

BST * delet(BST *root, int val){
	if(!root)
		return NULL;
	if(root->data == val){
		BST *temp;
		if(!root->left && !root->right)
		{
			free(root);
			temp = NULL;
		}
		else if(root->left && ! root->right)
		{
			temp = root->left;
			free(root);
		}
		else if(root->right && ! root->left)
```

```
            {
                temp = root->right;
                free(root);
            }
            else
            {
                BST *succ = succesor(root);
                root->data = succ->data;
                root->right = delet(root->right, succ->data);
                temp = root;
            }
            return temp;
        }
        if(root->data < val)
            root->right = delet(root->right, val);
        else
            root->left = delet(root->left, val);
        return root;
}

int main(){
        int n;
        BST *root = NULL;
        char c;
        scanf("%d", &n);
        for(int i = 0; i < n; i++)
        {
            int t;
            scanf("%d", &t);
            root = insert(root, t);
        }
        postOrder(root);
        printf("\n");
        scanf("%d", &n);
        for(int i = 0; i < n; i++)
        {
            char ch;
            scanf("%c", &ch);
            int t;
            scanf("%d\n", &t);
            switch(ch){
                case 'i':
                    root = insert(root, t);
```

```
                    postOrder(root);
                    printf("\n");
                    break;
               case 'r':
                    root = delet(root, t);
                    postOrder(root);
                    printf("\n");
                    break;
               case 'q':
                    printf("%d\n", query(root, t));
                    break;
               case 's':
                    printf("%d\n", findSize(root, t));
                    break;
          }
     }
     return 0;
}
```
