S.No: 29	Exp. Name: ***Stack Implementation using arrays***	Date:2024-10-27

Aim:

Write a program that implements the **Stack** Data structure using Arrays

The operations performed on the Stack are

1. Push
2. Pop
3. Peek

Input format:

- The first line of input contains the number of operations to be performed on the stack
- The next lines contain the integers separated by space in which the first integer indicates the operation to be performed and the second integer contains the element to be pushed.
- **1 --->** indicates the Push
- **2 --->** indicates the Pop
- **3 --->** indicates the Peek

Output format:

- Display every element after performing the peek operation
- Display the stack after performing all operations at the end.

Sample Test Case:

Input:

6
1 2
1 3
1 4
2
3
1 5

Output:

3
5 3 2

Explanation:

1 2 ---> 2 will be pushed
1 3 ---> 3 will be pushed
1 4 ---> 4 will be pushed
2 ---> Last element(4) is popped
3 ---> peek operation is performed which results in printing the top element of the stack i.e 3
1 5 ---> 5 will be pushed
6 Operations are completed
Therefore one peek operation is performed so the output is 3 and the final stack **5 3 2**

Note:

- If the stack is empty and when pop and peek are performed first, proceed to the next operation without displaying and modifying anything in the stack.
- If the stack is empty and nothing is pushed to the stack, Print Empty at the end.

Source Code:

CTC17094.c

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int sarr[MAX];
int top = -1;
int pval = 0;
void push(int value){
   if(top==MAX-1)
      return;
   top++;
   sarr[top]=value;
}
void pop(){
   if(top==-1)
      return;
   top--;
}
void peek(){
   if(top==-1)
      return;
   else
      pval=sarr[top];
}
void display(){
   if(top==-1)
   printf("Empty\n");
   for(int i=top;i>=0;i--){
      if(i!=0)
         printf("%d ",sarr[i]);
      else
         printf("%d\n",sarr[i]);
   }
}
int main(){
   int n,task,value,flag=0;
   scanf("%d",&n);
   for(int i=0;i<n;i++){
      scanf("%d",&task);
      if(task==1){
         scanf("%d",&value);
         push(value);
      }
      else if(task==2){
         pop();
      }
      else if(task==3){
         if(top!=-1){
            peek();
            flag++;
         }
      }
   }
   if(flag>0){
```

```
        printf("%d\n",pval);
    }
    display();
}
```


Execution Results - All test cases have succeeded!

Test Case - 1
User Output
6
1 2
1 3
1 4
2
3
1 5
3
5 3 2

Test Case - 2
User Output
5
1 44
1 55
2
2
3
Empty

S.No: 30	Exp. Name: ***Stack Implementation using Linked lists***	Date:2024-10-27

Aim:

Write a program that implements the **Stack** Data structure using Linked lists

The operations performed on the Stack are

1. Push
2. Pop
3. Peek

Input format:

- The first line of input contains the number of operations to be performed on the stack
- The next lines contain the integers separated by space in which the first integer indicates the operation to be performed and the second integer contains the element to be pushed.
- **1 --->** indicates the Push
- **2 --->** indicates the Pop
- **3 --->** indicates the Peek

Output format:

- Display every element after performing the peek operation
- Display the stack after performing all operations at the end.

Sample Test Case:

Input:

6
1 2
1 3
1 4
2
3
1 5

Output:

3
5->3->2

Explanation:

1 2 ---> 2 will be pushed
1 3 ---> 3 will be pushed
1 4 ---> 4 will be pushed
2 ---> Last element(4) is popped
3 ---> peek operation is performed which results in printing the top element of the stack i.e 3
1 5 ---> 5 will be pushed
6 Operations are completed
Therefore one peek operation is performed so the output is 3 and the final stack **5->3->2**

Note:

- If the stack is empty and when pop and peek are performed first, proceed to the next operation without displaying and modifying anything in the stack.
- If the stack is empty and nothing is pushed to the stack, Print Empty at the end.

Source Code:

CTC17095.c

```
#include <stdio.h>
#include <stdlib.h>
int peekval =0;
struct stack{
   int data;
   struct stack *next;
}*top =NULL;

void push (int value){
   struct stack *ptr=malloc(sizeof(struct stack));
   ptr->data=value;
   ptr->next = top;
   top =ptr;
}
void pop(){
   if(top==NULL)
      return;
   struct stack *tmp=top;
   top = top->next;
   free(tmp);
}
void peek(){
   if(top==NULL){
      return;
   }
   peekval=top->data;
}

void display(){
   struct stack *tmp=top;
   if(top==NULL){
      printf("Empty\n");
   }
   else{
      while(tmp->next!=NULL){
         printf("%d->",tmp->data);
         tmp=tmp->next;
      }
      printf("%d\n",tmp->data);
   }
}
int main(){
   int n,task,value,flag=0;
   scanf("%d",&n);
   for(int i=0;i<n;i++){
      scanf("%d",&task);
      if(task==1){
         scanf("%d",&value);
         push(value);
      }
      else if(task==2)
      pop();
      else if(task==3)
```

```
        if(top!=NULL){
        peek();
        flag++;
        }
    }
    if(flag>0){
    printf("%d\n",peekval);
    }
    display();
}
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output
6
1 2
1 3
1 4
2
3
1 5
3
5->3->2

Test Case - 2
User Output
5
1 44
1 55
2
2
3
Empty

Test Case - 3
User Output
7
1 7
1 8
2
1 9
3
1 10
2
9
9->7

S.No: 31	Exp. Name: ***Convert infix expression to equivalent postfix expression.***	Date:2024-11-13

Aim:

Function Rules:

Fill the missing logic in **function** `InfixToPostfix` with **return type** `char *` and **parameters** as listed below:

- `char *expr`

You are given a string **expr** which contains an infix expression. Your need to convert the infix expression to an equivalent postfix expression.

Complete the function **InfixToPostfix** with the following parameter:
$\quad$**expr:** Infix expression as an input in the form of string

The function will return:
$\quad$**string:** Postfix expression in the form of string.

Constraints:
0 < len(expr) <= 10^5

Sample test case:
Input: (a+b/c)
Output: abc/+

Important Instruction : Sample test case is for explanatory purpose but to run your custom test case on the terminal follow the input layout as mentioned in the command line arguments.

Source Code:

CTC9758.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAX 100
int pre(char c){
   if(c=='+'||c=='-')return 1;
   if(c =='*'||c=='/')return 2;
   if(c == '^')return 3;
   return 0;
}
int isOperator(char ch){
   return ch=='+'||ch =='-'||ch=='*'||ch=='/'||ch=='^';
}

char * InfixToPostfix(char *expr) {
   char *postfix = (char *)malloc(MAX* sizeof(char));
   char stack[MAX];
   int top =-1;
   int k=0;
   for(int i =0;expr[i];i++){
```

```
      if(isdigit(expr[i])||isalpha(expr[i])){
         postfix[k++]=expr[i];
      }
      else if(expr[i]=='('){
         stack[++top]=expr[i];
      }
      else if(expr[i]== ')'){
         while (top!=-1&& stack[top] != '(')
            postfix[k++]=stack[top--];
         top--;
      }
      else if(isOperator(expr[i])){
         while(top != -1 && pre(stack[top]) >= pre(expr[i])){
            if(expr[i]=='^'&& stack[top ]=='^')break;
            postfix[k++]=stack[top--];
         }
         stack[++top]=expr[i];
      }
   }
   while(top!=-1){
      postfix[k++]=stack[top--];
   }
   postfix[k]='\0';
   return postfix;
}

int main(int argc, char *argv[]) {
   char *expr = argv[1];
   printf("%s\n", InfixToPostfix(expr));
   return 0;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output
abc/+

Test Case - 2
User Output
abc/d*+ef^-

S.No: 32	Exp. Name: ***Balanced Parentheses***	Date:2024-11-13

Aim:

Function Rules:

Fill the missing logic in **function** `BalancedBrackets` with **return type** `char *` and **parameters** as listed below:

- `char *str`

You are given a string **str** of some set of braces**.** Your task is to check whether the given parentheses in the string are balanced or not.

Complete the function **BalancedBrackets** with the following parameter(s):
$\quad$**string** Str: String that contains the parenthesis.

The function returns:
$\quad$A **string** that returns '**Balanced**' if the parenthesis of the input string are balanced or returns '**Not balanced**' if the parenthesis are not balanced.

Constraints:

- **1 ≤ length of each string str ≤ 10^2**

Note: Every string consists of (,), {, }, [, and] braces only.

Sample test case 1:
Input:
[{}]{}[]()

Output:
Balanced

Explanation:
The brackets in the first '**[{}]{}[]()**' are balanced, because all brackets are closed and all nested brackets are closed in order. So the output is **Balanced.**

Sample test case 2:
Input:
[{}

Output:
Not balanced

Explanation:
The brackets in the second string '**[{}**' are not balanced, because the nested bracket '{' was not closed before it's surrounding '[ ]', so the order was not respected. Then the output is **Not balanced.**
Instructions: To run your custom test cases strictly map your input and output layout with the visible test cases.

Source Code:

CTC15014.c

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

char * BalancedBrackets(char *str) {
   char open[]={'(','{','['};
   char close[]={')','}',']'};
   static char b[]="Balanced";
   static char u[]="Not balanced";
   char stack[500];
   int top = -1;
   for(int i=0;str[i]!='\0';i++){
      for(int j=0;j<3;j++){
         if(str[i]==open[j]){
            stack[++top]=open[j];
         }
         else if(str[i]==close[j]){
            if(stack[top]==open[j])
               top--;
            else{
               return u;
            }
         }
      }
   }
   if(top==-1)
      return b;
   return u;
}

int main(int argc, char *argv[]) {
   char *str = argv[1];
   printf("%s\n", BalancedBrackets(str));
   return 0;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output
Balanced

Test Case - 2
User Output
Not balanced

S.No: 33	Exp. Name: ***Reverse a Stack using Recursion***	Date:2024-10-28

ID: 2301331720023

Aim:

Your task is to design a program that reverse the order of elements in a stack using recursion, without utilizing any loops.

Input format:

- The first line should contain an integer **n** representing the number of elements in the stack.
- The second line should contain **n** space-separated integers representing the elements of the stack.

Output format:

- The output will display the elements of the reversed stack, also separated by space

Constraints:

1 <= size of the stack <= 10^4
-10^9 <= Each element of the stack <= 10^9

Example input :

8
2 9 -54 92 6 9 2 0

Example output:

0 2 9 6 92 -54 9 2

Source Code:

CTC30989.c

```
#include <stdio.h>
#include <stdlib.h>
struct node{
   int data;
   struct node *next;
}*top = NULL;

void push(int value){
   struct node *ptr;
   ptr = malloc(sizeof(struct node));
   ptr->data = value;
   ptr->next = top;
   top=ptr;
}
void display(){
   struct node* tmp;
   if(top==NULL)
   printf("stack is empty\n");
   tmp=top;
   while(tmp->next!=NULL){
      printf("%d ",tmp->data);
      tmp=tmp->next;
   }
   printf("%d\n",tmp->data);
}
```

```
int main(){
   int n,value;
   scanf("%d",&n);
   for(int i=0;i<n;i++){
      scanf("%d",&value);
      push(value);
   }
   display();
}
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output
8
2 9 -54 92 6 9 2 0
0 2 9 6 92 -54 9 2

Test Case - 2
User Output
10
-5 -6 79 373 9872 82 9 102 523 0
0 523 102 9 82 9872 373 79 -6 -5

S.No: 34	Exp. Name: ***Binary search Technique***	Date:2024-10-27

ID: 2301331720023

Aim:

Write a program to check whether the given element is present or not in the array of elements using the binary search Technique

Input format:

- The first line of input contains an integer N representing the no. of elements of the array
- The second line input contains the array of N integers separated by space
- The last line of input contains the key element to be searched

Output format:

- If the element is found, print the index.
- If the element is not found, print **Not found**.

Sample Test Case:

Input:

7
1 2 3 4 3 5 6
3

Output:

2

Source Code:

CTC17128.c

```
#include <stdio.h>
int main(){
   int n,k;
   scanf("%d",&n);
   int arr[n];
   for(int i=0;i<n;i++){
      scanf("%d",&arr[i]);
   }
   scanf("%d",&k);
   int i=0,j=n-1,index,flag=0;
   while(i<=j){
      int mid = (i+j)/2;
      if(arr[mid]==k){
         flag =1;
         index =mid;
         break;
      }
      else if(arr[mid]>k){
         j=mid-1;
      }
      else if(arr[mid]<k){
         i = mid+1;
      }
   }
   if(flag==1)
      printf("%d\n",index);
```

```
    else
        printf("Not found\n");
}
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output
7
1 2 3 4 3 5 6
3
2

Test Case - 2
User Output
10
1 2 3 4 5 6 7 8 9 19
20
Not found

ID: 2301331720023

S.No: 35	Exp. Name: ***Fibonacci Numbers***	Date:2024-10-28

Aim:

Function Rules:

Fill the missing logic in **function** `fibonacciNumbers` with **parameters** as listed below:

- `int N`
- `int *resultsArr`

Update the `resultsArr` with the resultant values of your function and `return` the **number of values**.

Problem:

You are given a positive integer **N**. Your job is to print the Fibonacci series of numbers as an array/list up to the given integer **N**.

Complete the function **fibonacciNumbers** with the following parameter:
integer N

Function will returns:
Array: an array of integers that is having all the fibonacci numbers upto N

Constraints:

0 < N <= 10^5

Sample Test Case

Input:

5

Output:

[0, 1, 1, 2, 3, 5]

Source Code:

CTC9250.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MaxArrSize 100000


int fibonacciNumbers(int N, int *resultsArr) {
   // Write code here
   if(N<0) return 0;
   int count =0;
   resultsArr[count++]=0;
   if(N>=1){
      resultsArr[count++]=1;
   }
   while(1){
      int next =resultsArr[count-1]+resultsArr[count-2];
      if(next>N) break;
      resultsArr[count++]=next;
```

```
    }
    return count;

}

void printArrayElements(int *resultsArr, int resultsArrLength) {
    int index;
    for(index = 0; index < resultsArrLength - 1; index++) {
        printf("%d,", resultsArr[index]);
    }
    printf("%d\n", resultsArr[index]);
}

int main(int argc, char *argv[]) {
    int N = atoi(argv[1]);
    int resultsArr[MaxArrSize];
    int resultsArrLength = fibonacciNumbers(N, resultsArr);
    printArrayElements(resultsArr, resultsArrLength);
    return 0;
}
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output
0,1,1,2,3,5

Test Case - 2
User Output
0,1,1,2,3,5,8,13,21,34,55,89

S.No: 36	Exp. Name: ***Tower of Hanoi Puzzle***	Date:2024-10-28

Aim:

Given a Towers of Hanoi puzzle with N discs initially placed on Peg A, you need to write a function to find the minimum number of movements required to solve the puzzle, moving all the discs from Peg A to Peg C, using Peg B as an intermediate peg.

Input Format:
The input consists of a single integer N, representing the number of discs.

Output Format:
The function should return an integer representing the minimum number of movements required to solve the puzzle.

Constraints:
1 ≤ N ≤ 20

Example:
Input:
N = 3
Output:
7

Explanation:
For a Towers of Hanoi puzzle with 3 discs, it takes a minimum of 7 movements to move all the discs from Peg A to Peg C using Peg B as an intermediate peg.

Note:
In the Towers of Hanoi puzzle, the number of movements required to solve the puzzle follows the pattern 2^N - 1, where N is the number of discs.

Instruction: To run your custom test cases strictly map your input and output layout with the visible test cases.

Source Code:

CTC17071.c

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

int count(int n){
   int x =(pow(2,n))-1;
   return x;
}
int main(){
   int n;
   scanf("%d",&n);
   int k = count(n);
   printf("%d",k);
}
```

Execution Results - All test cases have succeeded!

Test Case - 1
User Output
3
7

Test Case - 2
User Output
5
31