Lab Assignment-8

ROLL: 2005535 | NAME: SAHIL SINGH | DATE: 14/09/21

QUES 1: [A] WAP to implement a stack which will support three additional operations in addition to push and pop.

a) peekLowestElement - return the lowest element in the stack without removing it from the stack

b) peekHighestElement - return the highest element in the stack without removing it from the stack

c) peekMiddleElement - return the (size/2+1) th element in the stack without removing it from the stack.

SOLUTION:

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

#define INT_MAX 2147483647
#define INT_MIN -2147483648

typedef struct Stack
{
    int data;
    struct Stack *link;
} Stack;

void push(Stack **, int);
int pop(Stack **);
int isEmpty(Stack *);
void display(Stack *);
int peek_lowest(Stack *);
int peek_highest(Stack *);
int peek_middle(Stack *, int *, int);

int main()
{
    Stack *stack = NULL;
    int choice;
    do
    {
        int val;
        printf("1) Insert in stack\n2) Display\n3) Delete top\n");
        printf("4) Peek lowest\n5) Peek highest\n6) Peek middle\n7) Exit\n->: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            printf("Enter value: ");
            scanf("%d", &val);
```

```
            push(&stack, val);
            break;
        case 2:
            printf("\ntop->");
            display(stack);
            break;
        case 3:
            printf("\nDeleted element: %d\n", pop(&stack));
            break;
        case 4:
            printf("\nLowest value: %d\n", peek_lowest(stack));
            break;
        case 5:
            printf("\nHighest value: %d\n", peek_highest(stack));
            break;
        case 6:
            printf("\nMiddle element: %d\n",
                   peek_middle(stack, &val, 0));
            break;
        default:
            printf("\nExiting...\n");
        }
        printf("----------------------------\n");
    } while (choice >= 1 && choice <= 6);
    return 0;
}

void push(Stack **stack, int num)
{
    Stack *temp = (Stack *)malloc(sizeof(Stack));
    temp->data = num;
    temp->link = *stack;
    *stack = temp;
}

int pop(Stack **stack)
{
    if (isEmpty(*stack))
    {
        printf("\nUnderflow!");
        return -9999999;
    }
    Stack *temp = (*stack);
    *stack = (*stack)->link;
    int val = temp->data;
    free(temp);
    return val;
}

int isEmpty(Stack *stack)
```

```c
{
    if (!stack)
        return 1;
    return 0;
}

void display(Stack *stack)
{
    if (isEmpty(stack))
    {
        printf("\b\b \n");
        return;
    }
    int temp = pop(&stack);
    printf("%d->", temp);
    display(stack);
    push(&stack, temp);
}

int peek_lowest(Stack *stack)
{
    if (!stack)
        return INT_MAX;

    int val = pop(&stack);
    int lowest = peek_lowest(stack);
    push(&stack, val);
    if (lowest > val)
        return val;
    return lowest;
}

int peek_highest(Stack *stack)
{
    if (!stack)
        return INT_MIN;

    int val = pop(&stack);
    int highest = peek_highest(stack);
    push(&stack, val);
    if (highest < val)
        return val;
    return highest;
}

int peek_middle(Stack *stack, int *length, int position)
{
    if (!stack)
    {
        *length = position;
```

```
        return INT_MIN;
    }
    int val = pop(&stack);
    int next = peek_middle(stack, length, position + 1);
    push(&stack, val);

    if (*length / 2 == position)
        return val;
    return next;
}
```

OUTPUT:

```
1) Insert in stack
2) Display
3) Delete top
4) Peek lowest
5) Peek highest
6) Peek middle
7) Exit
->: 1
Enter value: 10
----------------------------
1) Insert in stack
2) Display
3) Delete top
4) Peek lowest
5) Peek highest
6) Peek middle
7) Exit
->: 1
Enter value: 20
----------------------------
1) Insert in stack
2) Display
3) Delete top
4) Peek lowest
5) Peek highest
6) Peek middle
7) Exit
->: 1
Enter value: 30
----------------------------
1) Insert in stack
2) Display
3) Delete top
4) Peek lowest
5) Peek highest
6) Peek middle
7) Exit
```

```
->: 2

top->30->20->10 >
----------------------------
1) Insert in stack
2) Display
3) Delete top
4) Peek lowest
5) Peek highest
6) Peek middle
7) Exit
->: 4

Lowest value: 10
----------------------------
1) Insert in stack
2) Display
3) Delete top
4) Peek lowest
5) Peek highest
6) Peek middle
7) Exit
->: 5

Highest value: 30
----------------------------
1) Insert in stack
2) Display
3) Delete top
4) Peek lowest
5) Peek highest
6) Peek middle
7) Exit
->: 6

Middle element: 20
----------------------------
1) Insert in stack
2) Display
3) Delete top
4) Peek lowest
5) Peek highest
6) Peek middle
7) Exit
->: 3

Deleted element: 30
----------------------------
1) Insert in stack
2) Display
```

```
3) Delete top
4) Peek lowest
5) Peek highest
6) Peek middle
7) Exit
->: 2

top->20->10 >
----------------------------
1) Insert in stack
2) Display
3) Delete top
4) Peek lowest
5) Peek highest
6) Peek middle
7) Exit
->: 7

Exiting...
----------------------------
```

QUES 2: [B] Write a program to evaluate a postfix expression using stack.

SOLUTION:

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

typedef struct Stack
{
    int data;
    struct Stack *link;
} Stack;

void push(Stack **, int);
void pop(Stack **);
int top(Stack *);
int operation(char, int, int);
int EvoPostfix(char[]);

int main()
{
    Stack *stack = NULL;
    char str[100];
    scanf(" %[^\n]s", str);
    printf("= %d\n", EvoPostfix(str));
    return 0;
}

void push(Stack **stack, int num)
```

```
{
    Stack *temp = (Stack *)malloc(sizeof(Stack));
    temp->data = num;
    temp->link = *stack;
    *stack = temp;
}

void pop(Stack **stack)
{
    if (!*stack)
        return;
    Stack *temp = *stack;
    *stack = (*stack)->link;
    free(temp);
}

int top(Stack *stack)
{
    if (stack == NULL)
        return 0;
    return stack->data;
}

int operation(char ch, int op1, int op2)
{
    switch (ch)
    {
    case '+':
        return op1 + op2;
    case '-':
        return op1 - op2;
    case '*':
        return op1 * op2;
    case '/':
        return op1 / op2;
    case '%':
        return op1 % op2;
    }
    //Exponent
    int res = 1;
    for (int i = 0; i < op2; i++)
        res *= op1;
    return res;
}

int EvoPostfix(char arr[])
{
    Stack *stack = NULL;
    int op1;
    int op2;
```

```
    for (int i = 0; arr[i] != '\0'; i++)
    {
        if (arr[i] >= '0' && arr[i] <= '9')
        {
            int num = 0;
            while (arr[i] != ' ')
            {
                num *= 10;
                num += (int)arr[i++] - 48;
            }
            push(&stack, num);
            continue;
        }
        else if (arr[i] == '*' || arr[i] == '/' || arr[i] == '%')
        {
            op2 = top(stack);
            pop(&stack);
            op1 = top(stack);
            pop(&stack);
            push(&stack, operation(arr[i], op1, op2));
        }
        else if (arr[i] == '+' || arr[i] == '-' || arr[i] == '^')
        {
            op2 = top(stack);
            pop(&stack);
            op1 = top(stack);
            pop(&stack);
            push(&stack, operation(arr[i], op1, op2));
        }
    }
    return top(stack);
}
```

OUTPUT:

```
5 3 2 * + 7 9 / 4 * 2 / - 6 - 2 +
= 7
```

QUES 3: [C] Write a program to evaluate a prefix expression using stack.

SOLUTION:

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

typedef struct Stack
{
    int data;
    struct Stack *link;
```

```c
} Stack;

void push(Stack **, int);
void pop(Stack **);
int top(Stack *);
int operation(char, int, int);
int EvoPrefix(char[]);

int main()
{
    char str[100];
    scanf(" %[^\n]s", str);
    printf("= %d\n", EvoPrefix(str));
    return 0;
}

void push(Stack **stack, int num)
{
    Stack *temp = (Stack *)malloc(sizeof(Stack));
    temp->data = num;
    temp->link = *stack;
    *stack = temp;
}

void pop(Stack **stack)
{
    if (!*stack)
        return;
    Stack *temp = *stack;
    *stack = (*stack)->link;
    free(temp);
}

int top(Stack *stack)
{
    if (!stack)
        return 0;
    return stack->data;
}

int operation(char ch, int op1, int op2)
{
    switch (ch)
    {
    case '+':
        return op1 + op2;
    case '-':
        return op1 - op2;
    case '*':
        return op1 * op2;
```

```
    case '/':
        return op1 / op2;
    case '%':
        return op1 % op2;
    }
    //Exponent
    int res = 1;
    for (int i = 0; i < op2; i++)
        res *= op1;
    return res;
}

int EvoPrefix(char arr[])
{
    Stack *stack = NULL;
    int op1;
    int op2;
    for (int i = strlen(arr) - 1; i >= 0; i--)
    {
        if (arr[i] >= '0' && arr[i] <= '9')
        {
            int num = 0;
            int rev = 0;
            while (arr[i] != ' ')
            {
                rev *= 10;
                rev += (int)arr[i--] - 48;
            }
            while (rev)
            {
                num *= 10;
                num += rev % 10;
                rev /= 10;
            }
            push(&stack, num);
            continue;
        }
        else if (arr[i] == '*' || arr[i] == '/' || arr[i] == '%')
        {
            op1 = top(stack);
            pop(&stack);
            op2 = top(stack);
            pop(&stack);
            push(&stack, operation(arr[i], op1, op2));
        }
        else if (arr[i] == '+' || arr[i] == '-' || arr[i] == '^')
        {
            op1 = top(stack);
            pop(&stack);
            op2 = top(stack);
```

```
            pop(&stack);
            push(&stack, operation(arr[i], op1, op2));
        }
    }
    return top(stack);
}
```

OUTPUT:

```
+ ^ 3 2 16
= 25
```

QUES 4: [D] Write a program to convert an infix expression into its equivalent postfix notation.

SOLUTION:

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

typedef struct Stack
{
    char data;
    struct Stack *link;
} Stack;

void push(Stack **, char);
void pop(Stack **);
char top(Stack *);
int isEmpty(Stack *);
int higherPrecedence(char, char);
char *InfixToPostfix(char[]);

int main()
{
    char str[100];
    printf("Infix: ");
    scanf(" %[^\n]s", str);
    char *arr = InfixToPostfix(str);
    printf("Postfix: %s\n", arr);
    free(arr);
    return 0;
}

void push(Stack **stack, char ch)
{
    Stack *temp = (Stack *)malloc(sizeof(Stack));
    temp->data = ch;
    temp->link = *stack;
```

```
    *stack = temp;
}

void pop(Stack **stack)
{
    if (!*stack)
        return;
    Stack *temp = *stack;
    *stack = (*stack)->link;
    free(temp);
}

char top(Stack *stack)
{
    if (!stack)
        return '\0';
    return stack->data;
}

int isEmpty(Stack *stack)
{
    if (!stack)
        return 1;
    return 0;
}

int higherPrecedence(char ch1, char ch2) //Returns 1 if stack operator has equal or
{                                        //higher precedence than scanned operator
    if (ch1 == '^')
        return 1;
    else if (ch1 == '*' && (ch2 == '+' || ch2 == '-
' || ch2 == '/' || ch2 == '%' || ch2 == '*'))
        return 1;
    else if (ch1 == '/' && (ch2 == '+' || ch2 == '-
' || ch2 == '/' || ch2 == '%' || ch2 == '*'))
        return 1;
    else if (ch1 == '%' && (ch2 == '+' || ch2 == '-
' || ch2 == '/' || ch2 == '%' || ch2 == '*'))
        return 1;
    else if (ch1 == '+' && (ch2 == '+' || ch2 == '-'))
        return 1;
    else if (ch1 == '-' && (ch2 == '+' || ch2 == '-'))
        return 1;
    return 0;
}

char *InfixToPostfix(char str[])
{
    Stack *stack = NULL;
    int j = 0;
```

```
    char *arr = (char *)malloc(strlen(str) * sizeof(char));
    for (int i = 0; str[i] != '\0'; i++)
    {
        if (str[i] == ')')
        {
            while (!isEmpty(stack) && (top(stack) != '('))
            {
                arr[j++] = top(stack);
                pop(&stack);
            }
            pop(&stack);
            continue;
        }
        if ((str[i] >= 'a' && str[i] <= 'z') || (str[i] >= 'A' && str[i] <= 'Z'))
        {
            arr[j++] = str[i];
            continue;
        }
        while (!isEmpty(stack) &&
               higherPrecedence(top(stack), str[i]))
        {
            arr[j++] = top(stack);
            pop(&stack);
        }
        push(&stack, str[i]);
    }
    while (!isEmpty(stack))
    {
        arr[j++] = top(stack);
        pop(&stack);
    }
    return arr;
}
```

OUTPUT:

```
Infix: a+(b*c-(d/e^f)*g)*h
Postfix: abc*def^/g*-h*+
```

QUES 5: [E] Write a program to convert an infix expression into its equivalent prefix notation.

SOLUTION:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct Stack
{
    char data;
```

```c
    struct Stack *link;
} Stack;
void push(Stack **, char);
void pop(Stack **);
char top(Stack *);
int isEmpty(Stack *);
int higherPrecedence(char, char);
char *InfixToPrefix(char[]);
int main()
{
    char str[100];
    printf("Infix: ");
    scanf(" %[^\n]s", str);
    char *arr = InfixToPrefix(str);
    printf("Prefix: %s\n", arr);
    free(arr);
    return 0;
}
void push(Stack **stack, char ch)
{
    Stack *temp = (Stack *)malloc(sizeof(Stack));
    temp->data = ch;
    temp->link = *stack;
    *stack = temp;
}
void pop(Stack **stack)
{
    if (!*stack)
        return;
    Stack *temp = *stack;
    *stack = (*stack)->link;
    free(temp);
}
char top(Stack *stack)
{
    if (!stack)
        return '\0';
    return stack->data;
}
int isEmpty(Stack *stack)
{
    if (!stack)
        return 1;
    return 0;
}
int higherPrecedence(char ch1, char ch2) //Returns 1 if stack operator has equal or
{                                        //higher precedence than scanned operator
    if (ch1 == '^')
        return 1;
```

```
    else if (ch1 == '*' && (ch2 == '+' || ch2 == '-
' || ch2 == '/' || ch2 == '%' || ch2 == '*'))
        return 1;
    else if (ch1 == '/' && (ch2 == '+' || ch2 == '-
' || ch2 == '/' || ch2 == '%' || ch2 == '*'))
        return 1;
    else if (ch1 == '%' && (ch2 == '+' || ch2 == '-
' || ch2 == '/' || ch2 == '%' || ch2 == '*'))
        return 1;
    else if (ch1 == '+' && (ch2 == '+' || ch2 == '-'))
        return 1;
    else if (ch1 == '-' && (ch2 == '+' || ch2 == '-'))
        return 1;
    return 0;
}
char *InfixToPrefix(char str[])
{
    Stack *stack = NULL;
    int j = 0;
    char *arr = (char *)malloc(strlen(str) * sizeof(char));
    for (int i = strlen(str) - 1; i >= 0; i--)
    {
        if (str[i] == '(')
        { //evaluate all expressions after closing parentheses
            while (!isEmpty(stack) && (top(stack) != ')'))
            {
                arr[j++] = top(stack);
                pop(&stack);
            }
            pop(&stack);
            continue;
        }
        if ((str[i] >= 'a' && str[i] <= 'z') || (str[i] >= 'A' && str[i] <= 'Z'))
        {
            arr[j++] = str[i];
            continue;
        }
        //operator input in stack
        while (!isEmpty(stack) &&
               higherPrecedence(top(stack), str[i]))
        {
            arr[j++] = top(stack);
            pop(&stack);
        }
        push(&stack, str[i]);
    }
    while (!isEmpty(stack))
    {
        if (top(stack) != ')')
            arr[j++] = top(stack);
```

```
        pop(&stack);
    }
    for (int i = 0, j = strlen(arr) - 1; i <= strlen(arr) / 2; i++, j--)
    {
        char temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
    return arr;
}
```

OUTPUT:

```
Infix: a+(b*c-(d/e^f)*g)*h
Prefix: +a*-*bc*/d^efgh
```

QUES 6: [F] Two brackets are considered to be a matched pair if an opening bracket (i.e., (, [, or {) occurs to the left of a closing bracket (i.e.,),], or}) of the exact same type. There are three types of matched pairs of brackets: [], {}, and (). A matching pair of brackets is not balanced if the set of brackets it encloses are not matched. Write a program to determine whether the input sequence of brackets is balanced or not. If a string is balanced, it prints YES on a new line; otherwise, print NO on a new line.

Example: Input: {[()]} and Output: YES

Input: {[(])} and Output: NO

SOLUTION:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Stack
{
    char data;
    struct Stack *link;
} Stack;
void push(Stack **, char n);
void pop(Stack **);
char top(Stack *);
int isEmpty(Stack *);
int isBalanced(char str[]);
int main()
{
    char arr[100];
    printf("String: ");
    scanf("%s", arr);
    if (isBalanced(arr))
        printf("YES\n");
    else
        printf("NO\n");
    return 0;
```

```
}
void push(Stack **stack, char ch)
{
    Stack *temp = (Stack *)malloc(sizeof(Stack));
    temp->data = ch;
    temp->link = *stack;
    *stack = temp;
}
void pop(Stack **stack)
{
    if (!*stack)
        return;
    Stack *temp = *stack;
    *stack = (*stack)->link;
    free(temp);
}
char top(Stack *stack)
{
    if (!stack)
        return 'a';
    return stack->data;
}
int isEmpty(Stack *stack)
{
    if (!stack)
        return 1;
    return 0;
}
int isBalanced(char arr[])
{
    Stack *stack = NULL;
    for (int i = 0; arr[i] != '\0'; i++)
    {
        if (arr[i] == '(' || arr[i] == '[' || arr[i] == '{')
        {
            push(&stack, arr[i]);
            continue;
        }
        else if (arr[i] == ')' || arr[i] == '}' || arr[i] == ']')
        {
            if (isEmpty(stack))
                return 0;
        }
        if (top(stack) == '(' && arr[i] == ')')
            pop(&stack);
        else if (top(stack) == '{' && arr[i] == '}')
            pop(&stack);
        else if (top(stack) == '[' && arr[i] == ']')
            pop(&stack);
    }
```

```
    return isEmpty(stack);
}
```

OUTPUT:

```
String: {[(([]))]}
YES
```

QUES 7: [G] Write a program exhibiting Tower of Hanoi (recursive).

SOLUTION:

```c
#include <stdio.h>
void tower_of_hanoi(int, char, char, char);
int main()
{
    int noOfPegs;
    printf("Enter number of pegs: ");
    scanf("%d", &noOfPegs);
    char source = 'S';
    char aux = 'A';
    char dest = 'D';
    tower_of_hanoi(noOfPegs, source, dest, aux);
    return 0;
}
void tower_of_hanoi(int noOfPegs, char source, char dest, char aux)
{
    if (noOfPegs == 1)
    {
        printf("Peg %d: %c -> %c\n", noOfPegs, source, dest);
        return;
    }
    tower_of_hanoi(noOfPegs - 1, source, aux, dest);
    printf("Peg %d: %c -> %c\n", noOfPegs, source, dest);
    tower_of_hanoi(noOfPegs - 1, aux, dest, source);
}
```

SOLUTION:

```
Enter number of pegs: 4
Peg 1: S -> A
Peg 2: S -> D
Peg 1: A -> D
Peg 3: S -> A
Peg 1: D -> S
Peg 2: D -> A
Peg 1: S -> A
Peg 4: S -> D
Peg 1: A -> D
Peg 2: A -> S
```

```
Peg 1: D -> S
Peg 3: A -> D
Peg 1: S -> A
Peg 2: S -> D
Peg 1: A -> D
```