Name : Deshmukh Mehmood Rehan

MIS No. : 612303050

SY Computer Science – Div 1

Question 1: *Two stacks in one array:*

Create a data structure twoStacks that represents two stacks. Implementation of

twoStacks should use only one array, i.e., both stacks should use the same array for

storing elements. Following functions must be supported by twoStacks.

push1(int x) –> pushes x to first stack

push2(int x) –> pushes x to second stack

pop1() –> pops an element from first stack and return the popped element

pop2() –> pops an element from second stack and return the popped element

Code:

header.h : This File includes the declarations of structures and function prototypes

```
/*
1. Two stacks in one array:
Create a data structure twoStacks that represents two stacks. Implementation
of
twoStacks should use only one array, i.e., both stacks should use the same
array for
storing elements. Following functions must be supported by twoStacks.
push1(int x) -> pushes x to first stack
push2(int x) -> pushes x to second stack
pop1() -> pops an element from first stack and return the popped element
pop2() -> pops an element from second stack and return the popped element
*/

/* Including the necessary libraries */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define SIZE 20 /* we will define the size here which can be changed later on
but its necessary as we are using an array to implement the 2 stacks */
```

```c
/* This is our structure for 2 Stacks in an array
Basically my approach is simple:
    1. For the first Stack we push from the start of the array incrementing
the top1
    2. For the second stack we push from the end of the array in reverse order
decrementing the top2
    3. The full condition will occur when top1 + 1 == top2

This is more space efficient compared to my other idea where i had decided to
use half the stack for stack1 and other half for stack 2

*/
typedef struct Stacks{
    int *array; /* pointer to the array of elements */
    int top1; /* top of stack1 */
    int top2; /* top of stack2 */
} Stacks;


/* These are all the functions we can define for our Stacks */
void init(Stacks *s); /* Intializes the Stacks */
int isEmpty1(Stacks s); /* Checks if the first Stack is empty */
int isEmpty2(Stacks s); /* Checks if the second Stack is empty */
int isFull(Stacks s); /* Check whether the stacks are full */
void push1(Stacks *s, int x); /* push element to first stack */
void push2(Stacks *s, int x); /* push element to the second stack */
int pop1(Stacks *s); /* pop element from first stack */
int pop2(Stacks *s); /* pop element from second stack */
void display1(Stacks s); /* display the first stack */
void display2(Stacks s); /* display the second stack */
```

logic.c : contains the definition of all the functions declared in the header file along with some helper functions

```c
#include "header.h" /* including the header file */

/*
To initialize the Stacks all we need to do is:
    1. malloc memory for the array
    2. set top1 as -1
    3. set top2 as SIZE


    if malloc fails return
```

```c
*/
void init(Stacks *s){
    s->array = (int *)malloc(SIZE * sizeof(int));
    if(!s->array) return;

    s->top1 = -1;
    s->top2 = SIZE;
}

/*
if top1 is still at -1, it means the stack1 is empty
*/
int isEmpty1(Stacks s){
    return s.top1 == -1;
}

/*
if the top2 is still at SIZE, it means that stack2 is empty
*/
int isEmpty2(Stacks s){
    return s.top2 == SIZE;
}

/*
The isFull condition will be when all the elements of the malloced array are
full, this happens when top1 + 1 == top2
*/
int isFull(Stacks s){
    return s.top1 + 1 == s.top2;
}

/*
To push element to stack1 we increment top and place the element at top1 index

if the stacks are full we simply return
*/

void push1(Stacks *s, int x){
    if(isFull(*s)) return;

    s->array[++s->top1] = x;
}

/*
To push element to stack2 we decrement top and place the element at at top2
index

if the stacks are full we simply return
```

```c
*/

void push2(Stacks *s, int x){
    if(isFull(*s)) return;

    s->array[--s->top2] = x;
}
/*
to pop element from stack1 we simply return the element at top1 index and then
decrement top1

if stack1 is empty we return INT_MIN
*/
int pop1(Stacks *s){
    if(isEmpty1(*s)) return INT_MIN;

    return s->array[s->top1--];
}

/*
to pop element from stack2 we simply return the element at top2 index and then
increment top2

if stack2 is empty we return INT_MIN
*/
int pop2(Stacks *s){
    if(isEmpty2(*s)) return INT_MIN;

    return s->array[s->top2++];
}

/*
to display stack1 we simply traverse from 0th index to top1 and print each
element

we also handle the case when stack1 is empty
*/
void display1(Stacks s){
    if(isEmpty1(s)) return;

    int i;
    for(i = 0; i <= s.top1; i++){
        printf("%d | ", s.array[i]);
    }
    printf("\b\b\b   \n");

    return;
}
```

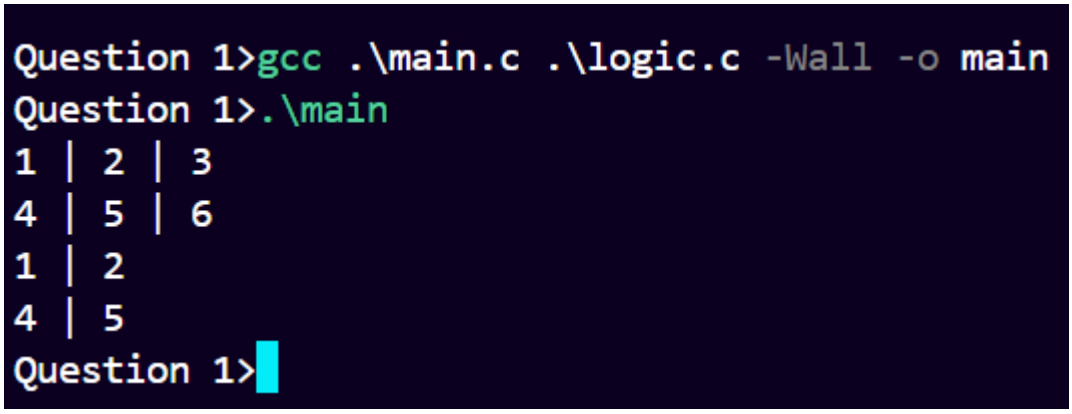
```c
/*
to display stack1 we simply traverse from last index to top2 in reverse order
and print each element

we also handle the case when stack2 is empty
*/
void display2(Stacks s){
    if(isEmpty2(s)) return;

    int i;
    for(i = SIZE - 1; i >= s.top2; i--){
        printf("%d | ", s.array[i]);
    }
    printf("\b\b\b   \n");

    return;
}
```

main.c : This contains the code to test the implementation

```c
#include "header.h"

int main(){
    Stacks s;
    init(&s); /* initialize the stacks */

    /* Push elements to stack1 */
    push1(&s, 1);
    push1(&s, 2);
    push1(&s, 3);

    /* Push elements to stack2 */
    push2(&s, 4);
    push2(&s, 5);
    push2(&s, 6);

    /* display the stacks */
    display1(s);
    display2(s);

    /* Pop elements from the stacks */
    pop1(&s);
    pop2(&s);

    /* Display the stacks again */
    display1(s);
    display2(s);
```

```
    return 0;
}
```

Output:



Question 2: *Check for balanced parentheses in an expression*

Given an expression string exp , write a program to examine whether the pairs and

the orders of “{“,”}”,”(“,”)”,”[“,”]” are correct in exp. For example, the program

should print true for exp = “[()]{}{[()()]()}” and false for exp = “[(])”

code:

```
/*
2. Check for balanced parentheses in an expression
Given an expression string exp , write a program to examine whether the pairs
and
the orders of “{“,”}”,”(“,”)”,”[“,”]” are correct in exp. For example, the
program
should print true for exp = “[()]{}{[()()]()}” and false for exp = “[(])”
*/


/* To solve this problem we will be using a character stack*/
#include "../Character Stack Implementation Using Singly LinkedList/logic.c"
#include <string.h>

/*Our approach for solving this problem is as follows:
1. We will traverse the entire expression string.
2. If we encounter an opening bracket we will push it onto the stack.
3. If we encounter a closing bracket we will check if the top of the stack the
closing bracket are pairs or not.
4. if they are not pairs or if the stack is empty then the expression is not
balanced.
5. If the stack is empty after traversing the entire expression then the
expression is balanced.
```

```c
*/

int isValid(char *exp); /*Function to check if the expression is balanced or
not*/

/* These are the helper functions */
int isOpening(char ch);  /* check if the character is an opening bracket */
int arePairs(char opening, char closing); /* check if the opening and closing
brackets are pairs */

int main(){
    char exp[] = "[()]{}{[()()]()}";
    printf("The expression %s is %s\n", exp, isValid(exp) ? "Balanced" : "Not
Balanced");

    char exp2[] = "[(])";
    printf("The expression %s is %s\n", exp2, isValid(exp2) ? "Balanced" :
"Not Balanced");

    return 0;
}

/*
returns 1 if the character is an opening bracket i.e. '(' or '{' or '['
*/

int isOpening(char ch){
    return ch == '(' || ch == '{' || ch == '[';
}

/*
returns 1 if the opening and closing brackets are pairs i.e. '(' and ')' or
'{' and '}' or '[' and ']' else returns 0
*/
int arePairs(char opening, char closing){
    if(opening == '(' && closing == ')') return 1;
    if(opening == '{' && closing == '}') return 1;
    if(opening == '[' && closing == ']') return 1;

    return 0;
}

/*
returns 1 if the expression is balanced else returns 0
we will use the approach mentioned above to solve this problem
*/
int isValid(char *exp){
    cStack s;
```

```
    cinit(&s);
    int i;
    for(i = 0; exp[i] != '\0'; i++){
        if(isOpening(exp[i])){
            cpush(&s, exp[i]);
        } else {
            if(cisEmpty(s) || !arePairs(cpeek(s), exp[i])){
                return 0;
            }
            cpop(&s);
        }
    }

    return cisEmpty(s);
}
```

Output:

```
Question 2>gcc .\main.c -Wall -o main
Question 2>.\main
The expression [()]{}{[()()]()} is Balanced
The expression [(]) is Not Balanced
Question 2>
```

Question 3: *Reverse a string using stack*

Given a string, reverse it using stack. For example "Data Structures" should be converted to "serutcurtS ataD".

code:

```
/* 3. Reverse a string using stack
Given a string, reverse it using stack. For example "Data Structures" should
be
converted to "serutcurtS ataD". */

/* To solve this problem we will be using a character stack*/
#include "../Stack Implementation Using Singly LinkedList/logic.c"
#include <string.h>

/*Our approach for solving this problem is as follows:
As Stack is a LIFO data structure, we can reverse a string using a stack.

1. We will traverse the entire string.
2. We will push each character onto the stack.
```

```c
3. After traversing the entire string we will pop each character from the
stack and store it in a new string.
4. The new string will be the reversed string.
*/
char *reverseString(char *str);

int main(){
    char str[] = "Data Structures";
    printf("The Original String is : %s\n", str);

    printf("The Reversed String is: %s\n", reverseString(str));
    return 0;
}

/*
We will solve this problem using the approach mentioned above.
*/
char *reverseString(char *str){
    int len = strlen(str);
    char *reversedStr = (char *)malloc(sizeof(char) * len);
    Stack s;
    init(&s);
    int i;
    for(i = 0; str[i] != '\0'; i++){
        push(&s, str[i]);
    }
    i = 0;
    while(!isEmpty(s)){
        reversedStr[i++] = pop(&s);
    }

    return reversedStr;
}
```

Output:

```
Question 3>gcc .\main.c -Wall -o main
Question 3>.\main
The Original String is : Data Structures
The Reversed String is: serutcurtS ataD
Question 3>
```

Question 4: *Convert a base 10 integer value to base 2*

code:

```c
/*
4 Convert a base 10 integer value to base 2
*/


/* To solve this problem we will be using an integer stack*/
#include "../Stack Implementation Using Singly LinkedList/logic.c"

/* Our approach for solving this problem is as follows:
We will use the property of the binary number system that a number can be
represented as a sum of powers of 2.
For example, 13 can be represented as 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 =
1101
So to convert a number to binary we will:
1. We will keep dividing the number by 2 and push the remainder onto the
stack.
2. After dividing the number by 2 we will pop the elements from the stack and
form the binary number.
*/
int decimalToBinary(int num);

int main(){
    int num;
    printf("Enter a base 10 number: ");
    scanf("%d", &num);

    printf("The base 2 representation of %d is: %d\n", num,
decimalToBinary(num));
    return 0;
}

/*
we will solve this problem using the approach mentioned above.
*/
int decimalToBinary(int num){
    Stack s;
    init(&s);
    while(num){
        push(&s, num % 2);
        num /= 2;
    }

    int binary = 0;
    while(!isEmpty(s)){
        binary = binary * 10 + pop(&s);
```

```
    }

    return binary;
}
```

Output:

```
Question 4>gcc .\main.c -Wall -o main
Question 4>.\main
Enter a base 10 number: 10
The base 2 representation of 10 is: 1010
Question 4>.\main
Enter a base 10 number: 12
The base 2 representation of 12 is: 1100
Question 4>
```