Duelist Card

Data Structure Module 1 (Stack)

Iffa Amalia Sabrina [5025221077]

Problem Summary

All problems on the planet TC are solved through card games. TCHolders are residents of TC who have unique abilities. Faiz challenges Denta, a TCHolder from Duelist City, to a card game. The game has specific orders: Push, Pop, Top, Max, and Min. Denta will be the winner if he successfully executes all of the orders. However, Denta has trouble remembering the orders, so he uses his ability to seeks assistance from someone on Earth. As Denta's chosen partner on Earth, we must assist him by using a stack program.

In the first line of input it contains of a number which also the number of commands (n). Then the next n lines are followed by the commands according to the decryption. The output of this program is the result of the command specified in the description.

Definition

Stack is a dynamic data structure that adheres to the Last In First Out (LIFO) principle. According to this principle, when a function to delete a node is called, the last element inserted into a Stack will be the first element deleted. One way to visualize a Stack is to imagine it as a stack of plates, with each new plate being placed at the top of the stack.

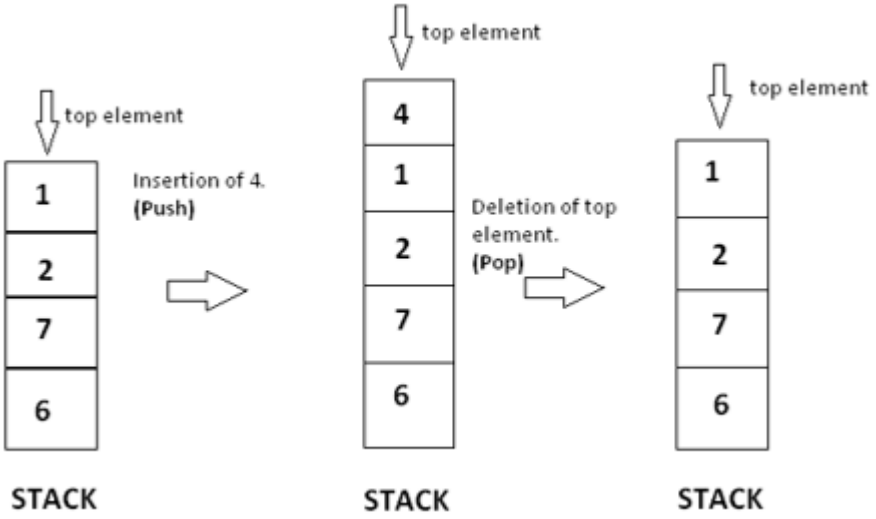


Figure 1. Stack Operation

Basic Operation

- `isEmpty` - determines whether or not the given Stack is empty.
- `size` - to get the size data of the Stack.
- `push` - to add new data at the top of the list.
- `pop` - removes a data from top.
- `top/peek` - to get the data at the top.
- `max` - to get the biggest data in the node.
- `min` - to get the smallest data in the node.

Solution

In this program, we are asking to create a stack program that can solve the problem. The first thing we need to do is copy and paste the entire stack program from GitHub or we can make it by our own by understanding the logic of stack program from GitHub. Second, we have to modify the structure of the node by adding new integers, Max and Min, which function to find the maximum and minimum values in the values that will later be added to the list later. Third, we need to modify the Push function so that the Max and Min node values are the same as the value (the value added to the list). If value is greater than list → head → Max, the node max value becomes value in Push function; conversely, to find minimum value in Push function.

Sample Input 0

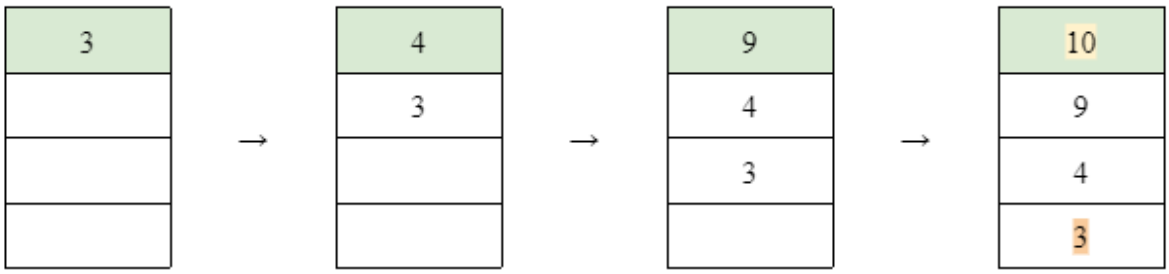
```
7
push 3
push 4
push 9
push 10
top
min
max
```

Sample Output 0

```
10
3
10
```

Sample Input-Output 0 Explanation

This program (sample input-output 0) will contain 7 commands. The first command is “push 3”, which means that the program will add 3 to the node. The second command is “push 4”, which means that the program will add 4 to the node and above 3. The third command is “push 9”, which means that the program will add 9 to the node and above 4. The fourth command is “push 10”, which means that the program will add 10 to the node and above 9. The fifth command is “top” which instructs the program to print the data at the top of the node, which in this case is 10. As a result, the first line of output will be 10 as the top of the node. The sixth command is “min” which instructs the program to print the smallest or the minimum data of the node, which in this case is 3. As the smallest data of the node, the second line of output will be 3. The seventh command is “max” which instructs the program to print the largest or the maximum data of the node, which in this case is 10. As a result, the third line of the output will be 3 as the node's largest data.



Green: Top of the node
Yellow: The maximum or the biggest number of the node
Orange: The minimum or the lowest number of the node

Figure 2. Sample Input-Output 0 Explanation

The Codes of the Program

Header

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define max 10
```

In the first few lines, we include necessary header files: `stdio.h`, `stdlib.h`, and `limits.h`. These header files provide functions and definitions required for input/output operations, dynamic memory allocation, and data type limits. Using the `#define` preprocessor directive, we define a constant max with a value of 10. This constant represents the maximum size of the command array.

Stack

```
struct Stack {
    int* elements;
    int top;
    int capacity;
};
```

Next, we define a structure called `Stack` to represents our stack. The structure called `Stack` contains three fields:

- `elements` - a pointer to an integer that will hold or store the elements of the stack.
- `top` - an integer that keeps track of the index of the top element in the stack.

- `capacity` - an integer that represents the maximum number of elements the stack can hold.

Creating a Stack

```
struct Stack* createStack(int capacity) {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->elements = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}
```

After defining a structure called `Stack`, to create a stack with given capacity, we defined `createStack` that takes an integer `capacity` as a parameter. The function allocates memory for the stack structure and the elements array. It initializes the top index to -1, indicating an empty stack, and returns a pointer to a `Stack` structure.

- Inside the function, we allocate memory for the stack structure using `malloc`.
- We initialize the `capacity` and `top` members of the stack structure.
- We allocate memory for the elements array using `malloc` based on the given capacity.
- Finally, we return the pointer to the created stack.

Checking if the Stack is Empty or Full

```
int isEmpty(struct Stack* stack) {
    return stack->top==-1;
}
int isFull(struct Stack* stack) {
    return stack->top==stack->capacity - 1;
}
```

Next, we define two functions `isEmpty` and `isFull` to check whether the stack is empty or full, respectively. Both functions take a pointer to a `Stack` structure as a parameter and return an integer value.

- The `isEmpty` function checks if the `top` member of the stack is -1, indicating an empty stack. If the stack is empty then it returns 1 (true), otherwise it returns 0 (false).

- The `isFull` function checks if the `top` member of the stack is equal to the `capacity - 1`, indicating a full stack. If the stack is full then it returns 1 (true), otherwise it return 0 (false).

Three Stack Operation

```
void push(struct Stack* stack, int x) {
    if (isFull(stack)) {
        printf("Stack Overflow\n");
        return;
    }
    stack->elements[++stack->top] = x;
}
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack Underflow\n");
        return INT_MIN;
    }
    return stack->elements[stack->top--];
}
int top(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty\n");
        return INT_MIN;
    }
    return stack->elements[stack->top];
}
```

After defining `isEmpty` and `isFull` to check whether the stack is empty or full, we define three stack operations: `push`, `pop`, and `top`.

- The `push` function takes a pointer to a `Stack` structure and an integer value `x` as parameters.
  - It first checks if the stack is full using the `isFull` function.
  - If the stack is full, it prints "Stack Overflow" and returns.
  - Otherwise, it increments the `top` member of the stack and assigns the value `x` to the corresponding element in the `elements` array.
- The `pop` function takes a pointer to a `Stack` structure as a parameter and returns an integer value.
  - It first checks if the stack is empty using the `isEmpty` function.
  - If the stack is empty, it prints "Stack Underflow" and returns `INT_MIN`, which represents an invalid value.

    - Otherwise, it retrieves the top element from the `elements` array, decrements the `top` member of the stack, and returns the element.
- The `top` function takes a pointer to a `Stack` structure as a parameter and returns the top element of the stack.
    - It first checks if the stack is empty using the `isEmpty` function.
    - If the stack is empty, it prints "Stack is empty" and returns `INT_MIN`.
    - Otherwise, it retrieves the top element from the `elements` array and returns it.

Finding the Minimum and Maximum Values in the Stack

```
int findMin(struct Stack* stack) {
    int min_val = INT_MAX;
    for (int i = 0; i <= stack->top; i++) {
        if (stack->elements[i] < min_val) {
            min_val = stack->elements[i];
        }
    }
    return min_val;
}

int findMax(struct Stack* stack) {
    int max_val = INT_MIN;
    for (int i = 0; i <= stack->top; i++) {
        if (stack->elements[i] > max_val) {
            max_val = stack->elements[i];
        }
    }
    return max_val;
}
```

Next, to find the minimum and maximum values in the stack, we define two utility functions `findMin` and `findMax`. Both functions take a pointer to a `Stack` structure as a parameter and return an integer value.

- The `findMin` function initializes a variable `min_val` with the maximum possible integer value (`INT_MAX`).
- It then iterates over the elements in the stack using a loop and updates `min_val` if a smaller element is found.
- Finally, it returns the minimum value found in the stack.
- The `findMax` function is similar to `findMin` but finds the maximum value in the stack instead.

Main Function

```
int main() {
    int rounds;
    scanf("%d", &rounds);
    struct Stack* cards = createStack(rounds);
    int min_val = INT_MAX;
    int max_val = INT_MIN;
    while (rounds--) {
        char command[max];
        scanf("%s", command);
        if (strcmp(command, "push") == 0) {
            int x;
            scanf("%d", &x);
            push(cards, x);
            min_val = (x < min_val) ? x : min_val;
            max_val = (x > max_val) ? x : max_val;
        } else if (strcmp(command, "pop") == 0) {
            if (!isEmpty(cards)) {
                int top_val = pop(cards);
                if (top_val == min_val) {
                    min_val = findMin(cards);
                }
                if (top_val == max_val) {
                    max_val = findMax(cards);
                }
            }
        } else if (strcmp(command, "top") == 0) {
            if (!isEmpty(cards)) {
                printf("%d\n", top(cards));
            }
        } else if (strcmp(command, "min") == 0) {
            printf("%d\n", min_val);
        } else if (strcmp(command, "max") == 0) {
            printf("%d\n", max_val);
        }
    }
    free(cards->elements);
    free(cards);
    return 0;
}
```

In the `main` function, we start by reading the number of rounds of commands from the user using `scanf` and storing it in the `rounds` variable.

We create a stack named `cards` using the `createStack` function and passing the `rounds` value as the capacity. We also initialize `min_val` with the maximum possible integer value (`INT_MAX`) and `max_val` with the minimum possible integer value (`INT_MIN`).

We enter a loop that iterates `rounds` number of times.

- Inside the loop, we read a command from the user using `scanf` and store it in the `command` array.
- We then check the command using a series of `if` and `else if` statements.
- If the command is "push", we read an integer value `x` from the user using `scanf`, push it onto the `cards` stack using the `push` function, and update `min_val` and `max_val` accordingly.
- If the command is "pop", we check if the stack is not empty using `isEmpty`. If it's not empty, we pop the top element from the `cards` stack using the `pop` function and update `min_val` and `max_val` if necessary.
- If the command is "top", we check if the stack is not empty using `isEmpty`. If it's not empty, we print the top element of the `cards` stack using the `top` function.
- If the command is "min", we simply print the value of `min_val`.
- If the command is "max", we simply print the value of `max_val`.

After the loop ends, we free the memory allocated for the `elements` array of the `cards` stack and then free the memory allocated for the `cards` stack itself using the `free` function.

Finally, the `main` function returns 0, indicating successful execution of the program.

Conclusion

This code implements a stack data structure and provides functions for creating a stack, performing stack operations, and determining the minimum and maximum values in the stack. Well that being said, this code can be used to assist Denta so he could be the winner because he successfully executed all of the orders.