HP35 Pocket Calculator using RPN

Muhammad Shahwaiz Abbasi

VT 2026

Introduction

This report presents the second assignment for the Data Structures and Algorithms (ID1021) course.

We will create a version of the HP35 calculator that can calculate mathematical expressions written in *reverse Polish notation (RPN)*. RPN is a postfix notation where the operator comes after the operands. For example, instead of writing (4 + 5) * 6, we write 4 5 + 6 *. The good thing is that Reverse Polish notation removes the need for parentheses.

The objective of this assignment is to understand how stacks work and how to use stacks to make a simpler *version* of a calculator.

Stack

There will be two types of stacks that we will be implementing in this assigment:

1. **A static stack:** Fixed Size and risk of overflow.
2. **A dynamic stack:** Adjustable size with flexible memory usage.

Stacks only hold integers instead of any other data types. A stack is a data structure that allows two basic operations:

1. **Push**: Adds an integer to the top of the stack.
2. **Pop**: Removes and returns the integer currently at the top.

How does a Stack work? A stack works with Last In First Out (LIFO), meaning that the last integer added to the stack is the first one to be removed. Stacks only keep track of the last index or in other words it can only access the top of the stack. To get access to the lower one, we have to first pop the top one.

Static Stack

A static stack has a fixed size that is declared when the stack is created and cannot be changed afterwards. Once the stack is full no more elements can be added. If you attempt to add more integers to a full stack then stack overflow can occur.

Dynamic Stack

A dynamic stack differs from a static stack in that its size can be changed during runtime. This stack can grow and shrink when the elements are added or removed which can reduce the risk of stack overflow and stack underflow. However, it requires additional memory which makes implementation complex.

Implementing Stacks

Let us start with the blueprint for the stack data structure.

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

typedef struct stack {
    int top;
    int size;
    int *array;
} stack;
```

This blueprint provides three different pieces of information. The first one is top which is the index where the next value will be inserted. `Size` is the total capacity of the stack. And last but not least we have a pointer that points to an array where the data is stored.

Static Stack Implementation

```
stack *new_stack(int size) {
    if (size < 1) {
        fprintf(stderr, "Stack size has to be at least 1\n");
        return 0;
    }
    int *array = (int*)malloc(size*sizeof(int));
    stack *stk = (stack*)malloc(sizeof(stack));
    stk->top = 0;
    stk->size = size;
```

```
    stk->array = array;
    return stk;
}
```

We start with a function named `new_stack` that takes a `size` named argument. We check if the size is less than 1. If true, we print a message saying that the size must be at least 1 to create a stack. After that we use `malloc` to access some memory for point array and the point stk. Since the data is stored in byte we multiply the number of size by `sizeof(int)` to get the total space. We initialize the stack by setting top to 0 as the stack is empty, saving the size and pointing the array pointer to the newly allocated memory.

```
void push(stack *stk, int val) {
    if (stk->top >= stk->size) {
        fprintf(stderr, "Stack overflow\n");
        return;
    }
    stk->array[stk->top] = val;
    stk->top+=1;
}
```

This is a `Push` function which adds integers to the stack and then increments the top. If the stack is full the message will be printed saying *Stack overflow*

```
int pop(stack *stk) {
    if (stk->top <= 0) {
        fprintf(stderr, "Stack underflow\n");
        return 0;
    }
    stk->top-=1;
    int val = stk->array[stk->top];
    return val;
}
```

Above code is a `Pop` function. It first decrements the top and then returns the value at the top of the stack. And if the top is 0 or less, a message will be printed saying *Stack underflow*

Dynamic Stack Implementation

As mentioned earlier in this report, we said that dynamic stack differs from static stacks in terms of size. Where the dynamic can easily adjust size based on the number of elements.

```
void push(stack *stk, int val) {
    if (stk->top == stk->size) {
        int size = stk->size * 2;
        int *copy=(int*)malloc( size * sizeof(int));
        for (int i = 0; i < stk->size; i++) {
            copy[i] = stk->array[i];
        }
        free(stk->array);
        stk->array = copy;
        stk->size = size;
    }
```

The push function checks if the top is equal to the current size. If true, then it doubles the size of the stack and creates a copy of the existing elements. It then updates the new size and points the array pointer to the new larger array.

```
int pop(stack *stk) {
     if (stk->top <= 0) {
        fprintf(stderr, "Stack underflow\n");
        return 0;
    }
    stk->top-=1;
    int val = stk->array[stk->top];
        if (stk->top <= stk->size / 4 && stk->size > 4) {
        int size = stk->size / 2;
        int *copy = (int*)malloc(size * sizeof(int));
        for (int i = 0; i < stk->top; i++) {
            copy[i] = stk->array[i];
        }
```

If the top is 0 or less, a message will be printed saying *Stack underflow*. The Pop function decrements the top and gets the element at the top of the stack. After that it checks if the top is one quarter of the size **AND** it has more than 4 elements then it shrinks the size of the stack by half and copies the existing elements. It then updates the new size and points the array pointer to the new smaller array.

THE CALCULATOR (HP35)

The final task for this assignment was to make a simple version of a calculator that uses the Reverse Polish Notation. We imported the whole code from the dynamic stack and wrote a new main to get user input for calculations.

```
stack *stk = new_stack(10);
```

```
printf("HP-35 pocket calculator\n");
int n = 10;
char *buffer = malloc(n);
bool run = true;
while(run) {
    printf(" > ");
    fgets(buffer, n, stdin);
    if (strcmp(buffer, "\n") == 0) {
        run = false;
    } else if (strcmp(buffer, "+\n") == 0) {
        int a = pop(stk);
        int b = pop(stk);
        push(stk, a + b);
    }
    ... // Substraction
    ... // Multiplication
    ... // Divison
    }
    printf("the result is: %d\n\n", pop(stk));
```

The code above is the *main* function which initializes a stack with a capacity of 10 elements and allocates 10 bytes of memory for user input. Then the program enters a while loop and asks the user for input until the loop stops. If the user enters without typing anything, the program exits and the element at the top is displayed. The calculator follows the Last In First Out (LIFO) method. It uses an if-else statement to determine the operation the user wanted. Then the calculator pops the top two elements from the stack to do the calculations like addition, subtraction, multiplications and division.

But for subtraction and division, the order is reversed, so the second value becomes first and the first value becomes second. For example, if we calculate `10 - 5` , the user inputs `10 5 -`. Here 5 is the first value popped because it is at the top of the stack while 10 is the second value popped. Therefore, the second value must be the first value to have the correct mathematical result. Additionally, if you attempt to divide by zero, a message will be displayed saying that division by zero is not possible.

At last, the assignment asked us for the result for `4 2 3 * 4 + 4 * + 2 -` and using the calculator, I got the answer **42**.