Stacks and HP-35

Jeffrey Chang

2nd September 2022

Introduction

For the course ID1021 - Algorithms and Data Structures, we are given tasks to submit each week for us to learn about the course content. This report concerns the second task of this course, an introductory assignment into simple stacks, namely, static and dynamic stacks and using these stacks to implement a basic calculator - the HP-35.

Background

The assignment content is about stacks as a data structure that stores and collects elements by using the two simple operations **PUSH** and **POP**. Due to the nature of how stacks store elements, it is also known LIFO (*Last In First Out*). Data which is *pushed* into and stored at the top of the stack is also the first one to be removed *popped* out and removed from the stack.

For this assignment, we are given the overall task to create the HP-35 calculator which utilises the stack data structure to calculate, meaning that inputting numbers and operations also work in a *LIFO* manner, for example, 2 + 1 would be 21+. Additionally, I am attempting to use C++ for the first time for this task to try out a new language.

For this specific assignment we were tasked to create a static and dynamic stack - meaning that one stack has a fixed array size whilst the other has, as its name suggests, a dynamic sized array which can change according to requirements.

The Tasks

Static Stack

The first task which we had was to create a static, fixed sized stack with a size of 4 and which only holds integers. The four questions which we had to consider, and are answered are in the following list:

- *Does the pointer point to the location above the top of the stack or does it point to the top of the stack?*

  For my code, the pointer points at the top of the stack and not the location above it. This can be seen in the following code snippet written in *C++*.

```
int top = -1
.
.
    top++; //adding to the next value, the pointer, so its at the top
    .
    .
//taking elements out of the stack (POP)
  if(top <= -1) //checking if the stack is empty
  cout <<"Stack Underflow"<<endl; //if true, then we have an underflow
  else
    cout <<"Number "<< stack[top] " is gone" <<endl;
    top--;
```

  The popped element is at the [**top**] place and following is the code top- - which checks the one below the element that just popped in the stack, therefore it is evident that the pointer points at the top of the stack as it goes with the pop and push functions, rather than being above the stack which perhaps would be done with an initialisation of top = 0.

- *What is the value of the pointer when the stack is empty?*

  When the stack is empty, the pointer is at its original value (or position), which, as can be seen from the previous code snippet, is -1. So whenever we attempt to pop the stack when there are no more elements, not only is there a stack underflow, but the pointer also has the value -1.

```
int top = -1
```

- *What should you do when a program tries to push a value on a full stack (stack overflow)?*

  If we try to push an element onto a full stack there should be an error message containing stack overflow, indicating that we have gone past the size and limit of the static stack. The following code is an example of how I did it.

```
//we have that n = size of array
//if the top is greater the n - 1,
//then there are more elements being pushed in than available in the stack
```

```
if(top >= n - 1)
  cout <<"Stack Overflow"<<endl;
  else
     top++;
     stack[top] = val;
```

So there should be a stack overflow warning if we try to push in more integers than the given size of the stack.

- *What should happen when someone pops an item from an empty stack?*

  Similarly to the previous question on stack overflow and that which I already mentioned on the empty pointer value, a stack underflow should occur if someone tries to pop an element from an already empty stack. So if the *top* has a value of -1 or less and we try to pop an element out of the stack an underflow will occur.

  ```
  if(top <= -1)
  cout <<"Stack Underflow"<<endl;
  else
     cout <<"The popped element is "<< stack[top] <<endl;
     top--;
  ```

This was the very basic implementation of a stack data structure that is also static in size. It is fundamentally important to understand how the stack works using basic sizes and a simple **PUSH** and **POP** function to truly understand what is going on, but once one gets how stacks work, implementing a basic static stack should not be too difficult.

Dynamic Stack

For the second part of the assignment, we are to actually implement the *Dynamic Stack* which can have a varying size according to the requirements set. On a personal note, I had to switch over to Java because of my current shallow knowledge of C++, and of course I translated the C++ static stack to Java as well (but I still wanted to present what I had done).

The Code

Following the instructions in the assignment paper, I filled in the following classes as shown:

```
public class Item
    public Item (ItemType type, int n)
        this.type = type
        this.value = n
```

```
    public ItemType type()
        return this.type
    public int value()

    //getting the three parts necessary for calculation
    public Item (ItemType itemType, int x)
            this.type = itemType;
            this.value = x;

public class Calculator
    public Calculator(Item[] expr)
            .
            .
        public int run()
            while (ip < expr.length)
            return stack.pop()
        public void step() //if ADD, then addition, if SUB, then subtraction...
                case ADD:
                    stack.push(x + y) //x & y values have integers as pointers
                    break             //that move down 3/4 steps in the stack
                case SUB, VALUE, MUL, DIV :
```

Benchmark and Discussion

This is of course a very shortened version of the entire Java file for the calculator to function properly with the most important parts included.

If we benchmark the Dynamic Stack data structure, using nanoTime, we can see that a static stack may perform faster for shorter/simpler tasks. This could be because the array size does not change so the program cannot do anything about that, however, the static stack is also more unsafe because once one pushes more elements into the stack than the size of the stack, there will be a stack overflow. This of course does not happen for dynamic stacks that can change its size. So the conclusion for the benchmarking is that the dynamic stack is safer as it does not crash but the static stack is faster if its for simple tasks which do not require larger sizes.

I also believe that, whilst we do not check for this, the dynamic stack may use more more memory, especially since I considered that it would be possible to solve this assignment with the use of *linkedlist* to solve this. But then, if we have a memory complexity requirement we could alternatively make use of the doubly linkedlist but then we would lose the time complexity part.

It appears that for the dynamic stack, as we increase the stack size, it also takes more time to push and pop all elements which makes sense considering the amount of extra things going on in the background - such

Stack Size	nanoTime in *microseconds*
10	10.5
100	75.6
1000	221.6
10 000	1261.3
100 000	4760.1

Table 1: Static Stack Benchmark - PUSH POP

Stack Size	nanoTime in *microseconds*
10	11.7
100	82.3
1000	411.6
10 000	2358.1
100 000	7932.4

Table 2: Dynamic Stack Benchmark - PUSH POP

as the increase and decreasing of the array size. However, this also means that it is more safe to use and no stack overflow will occur.

Implementation of Dynamic Array Size

Whilst the highlight of a dynamic stack is that it can increase its size whenever more elements are pushed in, it should also be able to decrease its size if there are too few elements to alleviate memory allocation and decrease the used memory for the stack. For an increase in stack size I simply multiplied the original stack by two which can be seen here:

```
//if stack overflow then double array size
Resize(maxSize * 2)
//transfer the array
 private void Resize{
        for (int i = 0; i < Array.length; i++)
            transferArray[i] = Array[i]
            Array = transferArray
        maxSize = newSize
```

The decrease this is so if a push were to come to insert another element, we dont have to allocate *more* memory again - so we avoid doing it twice.

```
    if (top < maxSize / 4){
        Resize(maxSize / 2);
        return pop();
```

Calculating My Last Digit

For the last part of this assignment we are to find and calculate our last personal number digit using the reversed Polish notation, which is what the HP-35 calculator uses. Following the equation given to us, and implementing the special mod and *´ notations we are given the following:

```
//Multiplication:
stack.push((y*x) / (10 + y*x % 10))
//Modulo 10:
stack.push(y % 10)
```

With the calculations done and inserting my personal number (010330-4192) into the equation we have:

10 - ((2*´0+1*´1+2*´0+1*´3+2*´3+1*´0+2*´4+1*´1+2*´9)mod10)

Which, in reverse Polish notation would be:

55+02*´11*´02*´31*´32*´01*´42*´11*´92*´++++++++m-

For my calculation and personal number, this yields 2, which is correct as it is my last digit.