HP35-Calculator Using Stacks

Aditya Gupta

September 2022

Introduction

In this assignment, we will be implementing the HP-35 calculator in Java. This calculator uses reverse polish notation. How the input is structured with this notation lends itself well to implement it using the stack data structure.
We implement the calculator first using a static stack which has a fixed size, followed up by making the same calculator using a growable array-based stack or dynamic stack. In the dynamic stack with the increase in the input size, the program allocates more memory for the input values pushed into the stack.
After we manage to implement the calculator using the above implementations, we compare the efficiency of the two data structures against each other through benchmarking similar to how we did this in the previous assignment by taking the use of Java's `System.nanoTime()` function. After we have done this we move to the final question using our implementation of the HP-35 calculator where we calculate the last digit of the personal number using the first 9 digits of the personal number.

Static Stack

Implementation

We have an internal array called `list` which stores all the integer values that we have obtained from our input and that we are going to be storing inside our stack. Other than this we have an integer that stores the current pointer value for our stack. We initialise the array to the Maximum capacity we want the stack to be able to have. We initialize the current position of the pointer variable to 0 (above the top of the stack), but this can be implemented in another way where for an empty stack this pointer is set to a value of -1(top of the stack); this can make it easier to check if our stack is empty and have some other potential benefits.

```
public class Static_Stack {

    int [] list;
```

```
int pointer;

Static_Stack(int length){
    this.list = new int[length];
    this.pointer = 0;
    //This could also have been initialized to -1 depending on
    //one's implementation of the stack data structure
}
:
:
```

Since this is a static stack which has been implemented using a fixed-sized array, we understand that if we try to push more items into the stack than its maximum capacity we will observe a `Stack Overflow`. This means we tried to overload the internal array more than its maximum capacity and thus we not only run out of space but also go out of the array's bounds. In a similar fashion if one tries to pop out more values than which were pushed onto the stack, to begin with, or if somebody tries to pop from an empty stack we will observe a `Stack Underflow`. This means we are trying to obtain values from the stack when there were none, to begin with, or in other words, values that never existed.

- **The push() function**

```
public boolean push(int i) throws ArrayIndexOutOfBoundsException{
        if(pointer < list.length){
            this.list[pointer++] = i;
            return true;
        }
        else{
            throw new ArrayIndexOutOfBoundsException("Stack Overflow");
        }
    }
```

In the push() function, we get the value to be pushed onto the stack as an argument and depending on the value of the `pointer` variable. If the value of the pointer is less than the maximum capacity of the array, then we add the argument value onto the stack. In case the pointer variable's value exceeds the maximum capacity of the stack, there would be a Stack Overflow (as mentioned earlier), thus we have to throw an `ArrayIndexOutOfBounds` Exception whenever something like this happens.

- **The pop() function**

```
public int pop() {
    if(pointer -1 < 0){
        throw new ArrayIndexOutOfBoundsException("Stack Underflow");
    }
    else{
        return list[pointer--];
    }
}
```

In the pop() function, we return the value that the pointer variable currently points to. We can do this while the stack is not empty which we check using the pointer variable's value. If the pointer points to 0 (meaning the stack is empty based on our implementation) and the pop function is called, it will result in a Stack Underflow. To counter this we throw an `ArrayIndexOutOfBounds` exception whenever something like this happens.

Dynamic Stack

The Dynamic stack is an extension of the implementation of the static stack but rather than throwing `ArrayIndexOutOfBounds` Exception every time we experience a stack overflow, we create a new array of twice the size of the original internal array for the static stack and copy all the stack elements and replace the original array with the new array. Thus we never run out of space in the stack as we can always allocate more memory to accommodate all the new values that have been pushed onto this stack. We increase the size to twice the size instead of adding a constant amount of more space as this means that if our stack size ever increases we don't end up allocating too much memory while also allocating enough at once for us to not keep having to copy elements to new arrays very often. The dynamic stack can also reduce its size depending on the usage of space. If the original size of the stack is somewhat large and most of it remains unused, it's better to reduce the amount of memory that has been allocated for this purpose so if there are any other tasks that might need it can use it instead. We can do this by only shrinking the stack if a certain number of push function calls have already been made so that the stack doesn't start decreasing without the user getting an opportunity to enter any elements, to begin with. Also if we shrink the array whenever there are less than a certain amount of elements, then the execution time for the calculator implemented using the dynamic stack would be sluggishly slow because of continuously having to copy elements into a different array. Thus we keep track of the number of push operations done while also checking for the position of the stack pointer for whenever we need to shrink the size of our stack.

The two functions used to make the stack Dynamic are:

- **The arrayExpander() function**

```
public int[] arrayExpander(int [] list){
    int [] newList = new int[list.length * 2];
    for (int i = 0; i < list.length; i++) {
        newList[i] = list[i];
    }
    return newList;
}
```

Whenever the pointer variable's value becomes equal to the size of the list i.e. whenever there would have been a stack overflow in case of the static stack, we call the arrayExpander() function to replace the original internal array for the stack with one that is twice as large and has all the elements of the previous array as they have been copied to it as can be seen in the code above.

- **The arrayShrink() function**

```
public void arrayShrink(int [] list){
if(pointer<=(list.length)/4){
    int [] halfList = new int[list.length/2];
    for (int i = 0; i < pointer + 1; i++) {
        halfList[i] = list[i];
    }
    this.list = halfList;
}
}
```

Whenever the pop() function is called we check if we have used the push operation at least the same number of times the size of the internal stack array. If this condition meets then we check the position of the stack pointer and if it is lesser than one fourth of the value the length of the internal array for the stack and also the length of the internal array is greater than 4.

Benchmarking the stacks

We compare the amount of time taken for the static stack and the dynamic stack to perform the push and pop operations and from that we get the following data tabulated below:

Case	No. of Iterations	Time for Static Stack	Time for Dynamic Stack
1	10	13.2 us	21.0 us
2	100	9.7 us	17.5 us
3	1000	3.4 us	6.9 us
4	2000	2.9 us	5.3 us
5	10000	1.9 us	3.0 us

This is benchmarking the calculator we made before but with the two different implementations (dynamic and static). The static stack is set to a size of 8 and the dynamic stack is set to initialize the dynamic stack's size to 4 and enter 8 elements in each. We notice that the dynamic stack takes more time than the static stack. This is most probably because of the extra time taken to increase the size of the stack whenever it runs out of space. We notice that for more iterations the time taken for both implementations becomes lower and lower and also becomes very similar.

Calculating the last digit of the personal number

The last part of the assignment was to calculate the last digit of my personal number using the weird multiplication formula given inside the assignment, given by:

$$10 - ((y_1 *' + y_2 *' 1 + m_1 *' 2 + ...) mod_{10})$$

I implemented the weird multiplication operator using two new enum constants called W-STAR and MOD operators. Implementation in code:

```
case W_STAR:{
            int y = stack.pop();
            int x = stack.pop();
            stack.push(x*y%10 + x*y/10);
            break;
        }
case MOD:{
            int x = stack.pop();
            stack.push(x%10);
            break;
        }
```

Using the above formula I was able to find my personal number's last digit when I inserted the formula correctly according to the reverse polish notation. The last digit of my personal number is 8 and it comes out correctly after many difficulties in trying to set the formula in the right format.