Assignment 2 HP35

Mahad Ahmed

Spring Fall 2022

Introduction

In this assignment I have implemented a calculator that can calculate mathematical expressions described in *reverse Polish notation*. Reverse Polish notation is simply put writing a mathematical expression with the operand last. For example instead of writing the usual way *3+2* I instead write *32+*. This is the way the old HP35 calculator which was used in the 1970s worked. To calculate an operation I need to implement something called a *stack*. A stack is a data structure that have two basic operations *push* and *pop*. Push is when I push something onto the stack i.e store something. When doing the pop operation it will remove the item on the top of the stack and return it's value. *LIFO*, Last In First Out is the principle that this stack is working on.

For the first task I have implemented a static stack and for the second task a dynamic stack.

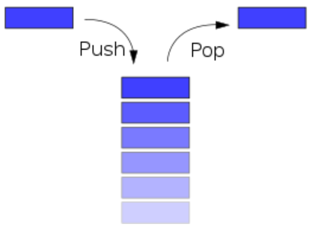

Figure 1: An image of stack and it's operations

Task 1, Static stack implementation

In this first task I implemented a static stack. Meaning this stack can not allocate any more space then what is demanded. It is fixed. A dynamic stack can allocate more space for more items as needed. Hence why it is dynamic.

It was fairly simple to implement this stack when you understood how it should work. I implemented namely with 2 basic functions. A push function doing what I explained in the introduction. Also a pop. Fairly simple in it's execution.

Stack pointer/Push

The implementation of the static stack was quite simple but some problems did occur. Of them was the stack pointer. Because the stack pointer always needs to point to the top of the stack, but when the stack was empty the pointer would point to the position above the top of the stack. Because I set the stack pointer to the value 0. To solve this problem I set the pointer to -1 and and incremented it before pushing a new value. By doing this the stack pointer will always point to the element on the top most position in the stack.

```
// 1. Push operation
 public void push (int value){
    if (pointer == items.length - 1)
      System.out.println("stackoverflow");
    else
      items[++pointer] = value;
  }
```

Pop

The pop method was pretty simple to do because of the stack pointer always pointing to the top most element in the stack. I just took it out, saved the value and return it.

```
// 2. Pop operation
  public int pop (){
    if (pointer == -1 ){
      System.out.println("stackNOflow");
      return 0;
    }
    // 3.
    int x = items[pointer];
    items[pointer--] = 0;
    return x;
  }
```

Task 2, Dynamic stack implementation

The difference as mentioned in task 1 between a static and a dynamic stack is that a dynamic stack can handle overflow. By that I mean if a overflow of items happens the dynamic stack can just allocate more space by creating a new and bigger array and copying the items over to the new array. But the static stack can not handle it and will result in a overflowed stack. I have implemented the dynamic static as shown below. But something else I did was to shrink the stack if I have extended it and don't need the space anymore. If I just let it be and keep extending it, it would take up much unnecessary space in our memory. Memory efficiency is key not just time efficiency.

Push

Much for the code of this method could be reused from the static stack push method. But instead of "StackOverFlow" when the stack is full I implemented that it could change size to fit in more values if I wanted. So every time a stackOverFlow would happen in a static stack I instead double the size of the current array. Another solution is to square the length of the array, but that would get big very fast and probably get out of control. For that reason I chose to just double the array.

```java
public void push (int value){
   if (pointer == items.length - 1){
     int[] newItems = new int [items.length * 2];
     sizeCounter = newItems.length;
     for (int i = 0; i <= pointer; i++)
       {
         newItems[i] = this.items[i];
       }
     this.items = newItems;
   }

   items[++pointer] = value;
 }
```

Pop

The changes I made to the pop function was I wanted to see if I could make it smaller. This was done by checking the stack pointer index. If the pointer was index was loIr than half the length of the array, it means that I made a new array and copied my items over to the new array. But I didn't want keep changing the size of the array every time, so it only halves the array if the elements are less than half of the array. By doing this I skip making a new stack every time I push or pop at the top.

```
  public int pop (){
    if (pointer == -1 ){
      System.out.println("stacknoflow");
      return 0;
    }
    if (pointer < ((sizeCounter/2) - 1)){
      int[] newItems = new int [(items.length / 2)];
      sizeCounter = newItems.length;
      for (int i = 0; i <= pointer; i++)
        {
          newItems[i] = this.items[i];
        }
      this.items = newItems;
    }

    int x = items[pointer];
    items[pointer--] = 0;
    return x;
  }
}
```

Benchmarks

Benchmark as shown below, is quite similar in result but enough to make a difference. The computer can also make a difference on which values you get on the benchmark.

Stack	**Run time(ms)**
Static	0,5
Dynamic	0.65

Table 1: Benchmark of the static and dynamic stacks

The dynamic stack is a little bit sloIr than it's static counterpart. The reason for this is obvious because the dynamic stack keeps resizing.

Calculating last digit

To calculate the last digit as described in the task file, I had to implement two cases: a *MOD* case that popped the stack. After which it operated modulus 10 on it and pushed it afterwards.

The second case I wanted to separate the numbers. As in if I get the ansIr 14, I made into 1 and 4. This is done checking if / 10 resulted in the value 1. This then means it's bigger than 10. If it's true then I just take the number modulus 10 and add 1. If it's not the case then I only return the number.

```
case NEWMUL: {
int y = stack.pop();
int x = stack.pop();
if (((y * x) / 10) == 1)
{
stack.push((1 + (y * x) % 10));
}
else
{
stack.push(x * y);
}
}
```