Static and Dynamic Stack Implementation for Reversed Polish Notation Calculator in Java

Alexander Alvarez

Fall 2024

Introduction

A calculator using reversed polish notation will be implemented in Java using static and dynamic stack arrays. This report will go over the creation of these stacks and how to implement them to use reversed polish notation for calculation.

The general behaviour of these stacks are to be described as well as how they were implemented.

Reversed polish notation

The reversed polish notation (RPN) is a way of writing mathematical expressions by writing the operand last. The use case of RPN for this report is the fact that calculation becomes more simple by using a stack, and we don't need any parentheses.

The stack

The word "stack" in Java refers to an array that is used to store elements. A stack consists of

- The stack array
- A stack pointer
- A size variable
- Two methods named `push()` and `pop()`

The stack pointer will point at an element at the top of the stack. Two methods `push()` and `pop()` are responsible for adding and removing elements in the stack and will also increment the stack pointer as they do. The stack array is used as the ”stack” and its size is defined by the size variable. However, the size variable is either static or dynamic depending on wether the stack is static or dynamic.

The reason for using RPN for calculations with a stack is due to the simplicity as previously mentioned. This is due to how the `push()` and `pop()` methods works on the stack. The `push()` operation will add an element to the top of the stack and will increment the stack pointer to point at this element as the top element. The `pop()` operation removes the top element and returns the previous element as the top element in the stack by incrementing the stack pointer again. This works well with RPN as we can write or push all the elements we want to work with onto the stack and then pop them out for the calculation.

StaticStack

The main difference between the static and dynamic stack is that the array size of the static stack does not change but is defined when the stack is created. The stack pointer is referred to as `top` in the following code.

```
public StaticStack(int size){
  stack = new int[size];
  top = 0;
}
```

A fixed stack size enables something called ”Stack Overflow” where pushing more elements than there is room for in the stack leads to an error. A solution to this can be to return a specific value when the stack pointer reaches the length of the stack, or an exception can be thrown. I made it so that the `push()` method will throw `ArrayIndexOutOfBoundsException` when the method detects that there will be a stack overflow.

```
public void push(int val) throws ArrayInde....{
  if(top == (stack.length)) throw new Ar...("Stack Overflow");

  stack[top++] = val;
}
```

Another error will appear if we try to pop an element when the stack is already empty. The `pop()` method will try to return a value at index -1 if the method is already empty. This is the same error as the previous due to the method trying to access an element at an index that is out of bounds for the array size. I used the same solution to solve this error.

```
public int pop()throws ArrayInde...{
  if(top == 0) throw new ArrayI...("Stack is empty");

  return stack[--top];
}
```

This is basically it for the static stack. The static stack is stored in the class `StaticStack` to be used in our calculator later.

DynamicStack

If the static stack has an array size that is constant and is defined when the stack is created, then the dynamic stack is simply the opposite.

We will have a default size so that the stack size does not become unnecessarily small when it changes.

```
public DynamicStack(){
  size = 4;
  stack = new int[size];
  top = 0;
}
```

A dynamic stack size does not generate an error when we try to push more elements than there is room for in the stack. The stack will instead increase in size to fit more elements. This is done by creating a new, bigger stack and copying over all the elements from the previous stack to the new stack when the stack pointer reaches the size limit instead of throwing an exception.

The size variable will be updated to the new size which is chose to be double the size of the stack.

```
public void push(int value){
  if(top == (size)){
    int copy[] = new int[size*2];
    for(int i = 0; i < size; i++) copy[i] = stack[i];
    stack = copy;
    size = stack.length;
  }

  stack[top++] = value;
}
```

The `pop()` method will generate the same "out of bounce" error for the dynamic stack if we try to pop an element in an already empty stack.

The `pop()` method in the dynamic stack serves another function which is to decrease the stack size if the bigger stack size is not being utilized. This is done in a similar way to how the `push()` method increases the stack size. The `pop()` method creates a new, smaller stack and copies over all the elements from the previous stack to the new stack when the stack pointer points to a certain index.

The size variable will be updated to the new size which is chose to be half the size of the stack. We leave some margins for the method to work with so that the size does not immediately decrease when the stack pointer points at half the stack size. This is done by simply making less than half the stack size the point of execution. We also don't want to decrease the stack size beyond the default size.

```
public int pop()throws ArrayInde...{
  if(top == 0) throw new ArrayI...("Stack is empty");

  if(((top + 1) < (size / 2)) && (size != 4)){
    int copy[] = new int[size/2];
    for(int i = 0; i < copy.length; i++) copy[i] = stack[i];
    stack = copy;
    size = stack.length;
  }

  return stack[--top];
}
```

The dynamic stack is very similar to the static stack with the difference being how they manage their sizes. The dynamic stack is stored in the class `DynamicStack` to later be used in our calculator.

The Calculator

The calculator itself is not too difficult to make. We create our stack and the reader for the terminal inputs. The integer inputs is pushed onto the stack until the reader detects an operation. If an operation is inserted, the the program will switch to the corresponding case to perform the calculation.

The respective operation cases are almost identical to eachother, difference being the operand in the respective `for` loops. The calculation stores the top element of the stack on a variable and performs the corresponding operation on this variable with each of the remaining elements in the stack. The result is then pushed onto the stack to be printed out or to be used in further operations.

The elements are in the wrong order in the case for the subraction operation. The problem is solved by multiplying the result with -1 before pushing it to the stack.

```
while(run){
  String input = br.readLine();
  switch (input){
    case "+":
      int sum = stack.pop();
      for(int i = stack.top; i > 0; i--){
        sum += stack.pop();
      }
      stack.push(sum);
      break;

    .
    .
    .

    case "":
      run = false;
      break;

    default:
      Integer nr = Integer.parseInt(input);
      stack.push(nr);
      break;
  }
}
```

Running the code and inserting the given input of ”$423 * 4 + 4 * +2-$” we get 110 as the output result. The result is the same for both the static and dynamic stack, provided that the static stack is of atleast length 3.

Conclusion

Both the static and dynamic stack was successfully implement using RPN. The output result is the same solution as my solution solved on paper, so I take it as the calculator being successful.

My biggest issue was implementing generic and abstract classes for the stacks, but i skipped this since it was not mandatory according to the assignment description.