HP35

Emanuel Paraschiv

Fall 2024

1 Introduction

The scope of this assignment is to implement a HP35 calculator capable of evaluating mathematical expressions described using reverse Polish notation. This task is designed to demonstrate the practical application of a stack data structure.

Implementing the stack

1.1 Static stack

A stack is a LIFO (Last-In, First-Out) data structure where the last element added is the first to be removed, using 2 principal methods: `push()` to add and `pop()` to remove elements. It's commonly implemented with arrays or linked lists, and in this experiment, we'll use arrays.

Implementing a static stack (with a fixed size) is a straightforward process. First, we have to create a `Stack` class along with a constructor that initializes the stack with a specified size. The `.push()` method is responsible for adding elements to the stack. It includes a check to determine if the stack has reached its maximum capacity. If the stack is full, a "Stack Overflow" exception is thrown to prevent adding more elements:

```
if(top==stack.length-1)
    throw new StackOverflowError("Stack overflow");
```

Otherwise we just increment the stack pointer to the next memory address and add the element there:

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

For `.pop()` the process is similar: first we check if the array is empty (which means that the stack pointer is null) and throw an exception if it is necessary,

```
if (top == -1) {
      throw new IllegalStateException("No elements to pop");
```

Otherwise we decrement the stack pointer and return the popped value.

```
int val = stack[top];
top--;
return val;
```

As expected, for an array of size smaller than the number of elements, when we try to push them and the array reaches its capacity an `StackOverflowError()` is thrown, and when we try to pop from an empty array, then an `IllegalStateException()` is thrown.

Also, as every `push()` operation is constant, the time complexity is constant, $O(1)$.

1.2 Dynamic stack

A dynamic stack can be more efficient than a static one. However, the implementation is crucial. If we increase the size of the array by just one element each time an overflow occurs, the performance would be significantly impaired:

- too much memory is allocated as we always have to create different arrays(can lead to memory fragmentation),
- the frequent resizing can lead to increased time complexity,
- the performance scaling is poor.

A better way of dealing with this problem is to double the size of the array every time we reach the maximum capacity. To implement this, every time we push an element we check if the stack pointer points to the last cell. If it is the case, we call a method named `changeSize(newSize)`:

```
 int[] newStack= new int[newSize];
// copy old elements
 for (int i = 0; i <= top; i++) {
     newStack[i]=stack[i];
 }
```

This method creates a new array with the user's input (`2*size` if too small or `size/2` if too large) to which all the elements in the stack are copied. After the copying, the method assigns this new stack to the old one.

Similarly, every time we pop an element, the `.pop()` method checks the size of the array, and if it is less than 1/4 used (but more than a min set value such as 4) then we are halving it:

```
if(top<size/4 && size>4)
   changeSize(size/2);
```

This way we can reclaim unused memory and improve the performance.

In the case of a dynamic stack, as we have to copy n elements to a new array every time we change the size, the time complexity becomes $O(n)$.

1.3 Abstract class

A better approach is to create a common interface for both stacks. This way we create a 'Stack' class which contains the attributes and methods of a stack. This blueprint is then implemented by both `Static` and `Dynamic` stacks.

After we do this, a great idea is to make the abstract class generic, to accept every kind of objects, which brings more flexibility.

```
public class Static<T> extends Stack<T>{}
public class Dynamic<T> extends Stack<T>{}
```

The `.push()`,`.pop()` and `.changeSize()` methods need to be changed in order to accept and work with objects instead of just integers:

```
@Override
public void push(T val) {
    ...
    stack[++top] = val;
...}
```

```
@Override
public T pop(){
    ...
    T val = stack[top--];
    ...}
```

In order to execute programs, I implemented a new `Main` class which only consists of one `main()` method to take care of the execution of the code.

2 The calculator

This calculator is an implementation of HP35, which leverages the stack structure to handle operations in Reverse Polish Notation (RPN).

By using a dynamic stack, the calculator efficiently handles a growing number of operands and operations without being constrained by a fixed size. The stack automatically resizes when it reaches its capacity, ensuring that the program can manage computations of varying lengths without running into overflow issues.

The implementation is the following:

The user inputs a computation to the terminal in the form of a string, written in RPN. Then the program iterates through each element:

- if we have an integer, the program will push it to the stack;

- if we have an operand (+ or - or * or /) then we will `pop()` the last 2 elements from the stack, will check for exceptions and if there are none, the result of the operation will be pushed to the top of the stack.

For example, this is the implementation for division (/):

```
case "/":
   int div1 = stack.pop();
   int div2 = stack.pop();
   if (div1 == 0) {
       // push them back
       stack.push(div2);
       stack.push(div1);
       throw new ArithmeticException("Can't divide by 0");
   }
   stack.push(div2 / div1);
   break;
```

For example, the result of this computation $(423 * 4 + 4 * +2-)$ is 42.

3 Conclusion

This kind of stack-based calculator is not just a theoretical concept but is practical in many fields, especially in compilers and interpreters. In those applications, expressions are often converted to a stack for efficient computation.

The efficiency and simplicity of the stack structure make it ideal for various use cases, including parsing, evaluating mathematical expressions, and managing function calls in recursive algorithms.