HP35

Fatima Mohammad Ali

Fall 2023

Introduction

In this assignment, a calculator that calculates mathematical expressions using *Reverse Polish Notation* was coded with the implementation of *stacks*. Since two types of stacks, i.e. *dynamic* and *static*, were used, the two types were later compared using benchmarks to estimate the cost of each stack.

The Calculator

The design of the calculator is based on defining an expression and performing its desired arithmetic operations. To define an RPN expression, an array of *items* was created in a `Calculator` class. The array holds elements of the object `Item`, which is a two-element object that holds a type (of type `ItemType`) and a value (of type `int`). The `Item` class also defines methods that return type and/or value when called, e.g.:

```
//returns addition operation
public static Item Add(){
    return new Item(ItemType.ADD, 0);
}
```

The `Calculator` class has a constructor, a `run` method that is called in `main`, and a `step` method that will perform the desired operation by checking the `type()` of the next `Item` in the item array to switch to its corresponding case. The `step` function includes the cases ADD, SUB, MUL, and DIV, which respectively perform stack procedure by popping the two previous integers from the stack and pushing the result (such as popping x and y respectively and pushing (x + y) at once for the ADD case). For the VALUE case, one push of the next Item value is done. Besides complementary code snippets that were added to `Item` and `step()`, all coding concerning the calculator design was given in the assignment ”A Calculator” by Johan Montelius (Fall 2023).

Stack

The stacks used in this assignment were implemented in two classes, `Static` and `Dynamic`. Each class extends the abstract class `Stack` which itself includes two abstract methods `push()` and `pop()`, a *stack* (`int[]`), a *size* of the array (`int`), and a *stack pointer* (`int`). The *push* operation will push an item at the top of the stack, while the *pop* operation will retrieve the value at the top of the stack. The size is determined when creating the array, and the stack pointer is used to navigate the stack.

Static stack implementation

The `Static` stack has a fixed size when created and can not allocate more items during run-time, meaning it can reach full capacity. `Static` has a constructor given a size (`int size`, initialized to desired size) and a stack pointer (`top = 0`). The class has a `push` method that pushes a given value (`int val`) at the top of the stack, as follows:

```
public void push(int val){
    stack[top] = val;   //pushes value in the stack
    top++;  //goes to the next index
    //if the stack pointer is at the maximum index, print the message
    if(top == size)
        System.out.println("Stack is full");
}
```

There is also a `pop` method that removes and returns the item at the top of the stack:

```
public int pop(){
    top--;  //go to the previous index
    //if the stack pointer is less than 0, print the message
    if(top < 0){
        System.out.println("Stack is empty");
        return 0;
    }
    else
        return stack[top]; //pop the element
}
```

Dynamic Stack Implementation

The `Dynamic` stack is able to resize during run-time, which makes it more flexible than the static stack.

The dynamic stack starts with a set size (either given or standard is 4) defined as follows:

```
public class Dynamic extends Stack{
    final int SIZE;

    public Dynamic(){
        SIZE = 4;
        stack = new int[SIZE];  //stack is given size 4
        top = 0;    //stack pointer initialized to 0
        size = SIZE;
    }
```

The extension in the `push` method is done by copying all array elements to a new, bigger array (double the size of the old array):

```
    public void push(int val){

        if(top == size){ //if stack pointer is at the maximum index
            int[] temparr = new int [2*size]; //new bigger array

            for(int i = 0; i < size; i++)
                temparr[i] = stack[i];  //copy elements to new array

            size = 2*size;  //set new size
            stack = temparr; //old array points to the new array
        }
        stack[top] = val;   //push given value in array
        top++;  //go to next index
    }
```

If the stack gets unnecessarily big, it could be downsized by detecting the position of the stack pointer. For this assignment, it was done so that if the stack has doubled and the stack pointer is still at one quarter(1/4) of the stack, then the array size will be halved (so the stack pointer is able to push one quarter freely until the stack is full again). This would be more efficient than to constantly double the size to half it later after popping some elements. The procedure was done in the following `pop()` method:

```
    public int pop(){
        top--;  //go to previous index
        if(top < 0)
            System.out.println("Stack is empty");
        else if(top < (size/4) && size > SIZE){ //&& min 4 items in stack
            int[] temparr = new int [size/2]; //new half sized array

            for(int i = 0; i < temparr.length; i++)
                temparr[i] = stack[i];

            size = size/2; //set new size
            stack = temparr;
        }
        return stack[top];  //pop item
    }
```

Benchmarks

A benchmark was used to compare the cost of each stack by pushing and popping a given amount (n) of elements in a loop of 10 000 times and taking the minimum value of 10 such tries, as seen in table 1.

n	static	dynamic	ratio
50	0.09	0.45	4.99
100	0.35	0.86	2.47
200	0.63	1.75	2.76
400	1.24	3.12	2.51
800	2.46	6.32	2.57
1600	4.92	12.91	2.62
3200	9.76	29.74	3.05

Table 1: *n* gives the number of pushes and pops, *static* and *dynamic* give the minimum timestamp of 10 tries by running the push and pops 10 000 times (in microseconds), and *ratio* is the ratio between dynamic/static.

As seen in the table, the dynamic stack costs almost more than double as much as the static one. The time complexity for the static stack is constant since it has a fixed size regardless of the amount of pushes and pops. The time complexity for the dynamic stack however, is constant until it has to double in size and copy all elements to a new array, then it becomes constant again until next extension. Hence, even though the dynamic stack is more flexible than the static one by resizing, it is more costly in time.