HP35 Calculator

Ayda Ghalkhanbaz

Fall 2023

Introduction

The main purpose of this assignment is learning about stacks and their structure using a HP35 calculator which is based on reversed polish notation algorithm. In this assignment two types of stacks are implemented: static and dynamic stack. Finally, some benchmarks were done in order to analyse these stacks.

The Calculator

This assignment was divided into three parts. The two first parts of this assignment were quite simple since the most part of the cod was already given. **Item** class only needed a constructor, getters, and some methods for creating objects of different types of mathematical operations. Also, the **calculator** was completed after adding some cases to handle operations such as: ADD,SUB,MUL, etc.

Stack

This part of assignment was the most important part. Two versions of stack were implemented. These stacks are the same in basis but have differences when it comes to their memory usage.

Static Stack

A static stack has a fixed size and in case of lack of capacity, we won't be able to resize the stack to push new items. In this task, a static stack of type int, and a stack pointer to keep track of the top of stack, was implemented. This stack has a constructor and two methods for push and pop items. In the code below, the `push()` method is shown. The logic behind this method is quite simple; if the stack is out of space, then the message "Stack Overflow!" will be shown to user and the running Java virtual machine will be terminated,

otherwise the item is pushed on the stack and the stack pointer points to the top of stack.

```
public void push(int element){
      if(sp == myArr.length){
          System.out.println("Stack Overflow!");
          System.exit(0);
      }
      myArr[sp] = element;
      sp++;
      // stack pointer points now to the top of stack
  }
```

The `pop()` method works the opposite way of `push()` method. At first, we check if the stack is empty; if so, then the message "Stack is already empty!" will be shown to user and we exit the program. But in case stack is not empty, we decrement the stack pointer and return the element which is now on the top of stack. The `pop()` method is shown below.

```
public int pop(){
      if(sp==0){
          System.out.println("Stack is already empty!");
          System.exit(0);
      }
      sp--;
      return myArr[sp];
  }
```

Dynamic Stack

A dynamic stack has a similar basis to static stack, the only difference is that a dynamic stack is flexible when it comes to the size of stack. It can easily increase/decrease its size without having "Stack Overflow" error. A dynamic stack can be initialized for instance with a size of 4 and during the program running increases its size if needed. In the implementation of `push()` method, which is shown below, we start with a stack of size 4 and a stack pointer pointing to the top of stack as well, and if we reach the top of stack then we create another stack with a doubled size and copy all

the items from the old stack to the new one. Finally, we can add the new element to the stack and update the stack pointer.

```
public void push(int element){
      if(sp == size){
         int[] newArr = new int[size * 2];
         for(int i = 0; i < size ;i++){
          newArr[i] = myArr[i];
         }
         size = size * 2;
         myArr = newArr;
         // myArr now points to the larger stack
      }
      myArr[sp] = element;
      sp++;
  }
```

The `pop()` method in dynamic stack has a similar logic to pop method in static stack. If we try to pop element from an empty stack, we get the message "Stack is already empty" and the program will be terminated. Otherwise, we decrement the stack pointer and return the element that stack pointer points to after decrementing. The only difference is if we have a stack with a large size which mostly is unused, we can free the memory by decreasing the size of stack. In the following code snippet, the implementation of `pop()` method is shown. The size of stack will decrease only when: 1. the stack is occupied to less than a quarter of its actual size and 2. we haven't reached the minimum size of stack yet (which is 4 in this case). If the conditions are met, then we halve the size of stack.

```
public int pop(){
       if(sp==0){
          System.out.println("Stack is already empty!");
          System.exit(0);
      }
      sp--;
      if(sp < size/4 && size > SIZE){
          int [] newArr = new int [size/2];
          size = size/2;
          for(int i= 0; i < sp; i++){
              newArr[i] = myArr[i];
          }
```

```
            myArr = newArr;
            // myArr now points to the larger stack
        }
        return myArr[sp];
    }
```

Benchmark

The result of benchmarking for the two stacks are presented in Table 1, showcasing the optimal time out of varying numbers of push and pop operations.

#n of Runs	Static	Dynamic	Ratio
10	0.35	0.92	2.60
20	0.25	1.75	6.96
50	0.62	0.65	1.04
100	0.51	2.09	4.12
200	0.86	3.11	3.63
400	1.36	4.20	3.08
800	2.41	8.37	3.48
1600	4.97	16.76	3.37
3200	10.22	32.18	3.15

Table 1: Benchmark results for static vs dynamic stacks in microseconds

As demonstrated in table above, the ratio is unstable primarily due to the smaller stack size, however after a while the dynamic stack becomes approximately 3 times slower than static stack.

Discussion

Comparing the time complexity of a static and dynamic stack leads to the conclusion that the static stack is faster since it has a fixed size and doesn't need to copy items from one stack to another with a smaller or larger size. Thus, a static stack has a time complexity of O(1), meaning it consistently takes a constant amount of time when pushing and popping elements from the stack. On the other hand, a dynamic stack is memory-efficient as it can adjust its length depending on the number of items on the stack. However, the process of copying items to a smaller or larger stack can be costly. During the copying process, a dynamic stack has a time complexity of O(n).

The more elements there are in the stack, the longer it takes to perform operations on the stack. Choosing between a static or dynamic stack involves a trade-off between achieving the most efficient memory usage or the shortest execution time for the program.