Structure & Performance Of Static & Dynamic Stacks

William Frid

Fall 2023

Introduction

A stack is a memory mechanism based on the Last In, First Out (LIFO) principle. It is commonly used in lower-level languages such as Java, C, and Assembly to store data temporarily before high-level languages became more common. On top of this, there are static (fixed-sized) stacks and dynamic (flexible-size) stacks. Both have their own advantages and disadvantages. Such as flexibility, speed, and usability.

This report discusses the structure of a static and dynamic stack that will be used in a traditional HP35 calculator software. The calculator software will be using reversed Polish notation since this method of handing data takes great advantage of the stack and is thus ideal for the purpose of exploring which type of stack should be used in certain cases, and how well they perform. The focus will start on the code, followed by a benchmark that will show how these two types of stacks perform in practice.

Static Stack

To create a static stack, a class called *StaticStack* was made that extends the class *Stack*, which holds a constructor, pointer (of type integer), and an empty array(the actual stack). *Note that the array is the stack itself and thus both words will be used throughout the report. StaticStack* adds the methods `pop()`, `push()`, and its own constructor which has an integer parameter that's used for passing an argument to the superclass constructor to set the array length (stack size).

The methods `pop()` and `push()` in the subclass *StaticStack* do three things; check pointer location to prevent *Array Index Out Of Bounds Exception* from being thrown because of from too much popping or pushing, increase/decrease pointer to always keep in on top, and add/read a value to return from the stack. The method `push()` can be seen below this paragraph. Note that the method `pop()` wasn't included due to its similarity.

```
if (p == this.arr.length - 1) {
    throw new ArithmeticException("Stack overflow accu...");
}
arr[p++] = val;
```

For the pointer to work correctly, it needs to point to the top of the stack. When the stack is initialized or empty, the pointer holds the index to the first integer of the stack (index 0). Upon pushing, it increments by one after the integer has been inserted into the stack. Upon popping, the pointer is first decreased by one, followed by the value is read, and returned by the method. Note how the order or changing the pointer value differs based on the action. This can be done differently, but can easily become confusing for anyone reading the code. The pointer could for instance always point on the latest inserted value. But then it would have to be negative or have extra logic to handle an empty array.

Dynamic Stack

The dynamic stack was created as a class called *DynamicStack*, it was built on the class *StaticStack* (error handling and pointer logic was copied). What differs is the support for a growing and shrinking stack. This is accomplished by adding three more methods, `growStack()`, `shrinkStack()`, and `copyArrayContent()`. The methods `growStack()` and `shrinkStack()` both have the main purpose of creating new arrays, and both call upon the method `copyArrayContent()` (the `...Stack()`) methods will be discussed in later subsections). `copyArrayContent()` is required since arrays in Java are of a fixed length, and hence changing the size requires copying the old array data to a new array (which is created in the `...Stack()`) methods). In other words, it moves the stack content onto a new stack with a different size. This is simply done with a for-loop as seen below.

```
for (int i = 0; i < p; i++) {
    newArr[i] = oldArr[i];
}
this.arr = newArr;
```

The for-loop is based on the pointer since the pointer will always be smaller than `newArr` and `oldArr` and thus always copy just the data under the pointer in the stack and leave out data that has already been popped.

Dynamic Stack: Pushing

Upon push, the method `push()` first checks if the stack is full. If so, it calls upon `growStack()`, which uses the variable `growthSpeed` and generates a

new larger array of a fitting size. This is shown below as well as how the `growthSpeed` increases and `shrinkSpeed` resets (more on this under section Dynamic Stack: Popping).

```
shrinkSpeed = 0;
int[] newArr = new int[this.arr.length + ++growthSpeed * 2];
copyArrayContent(this.arr, newArr);
```

This means that the growth accelerates each time the size is increased (2, 4, 6, 8, 10,... etc.). Other formulas could also be used, such as exponential acceleration. The idea behind this is that values are generally added in groups, and thus an accelerating growth of the stack leads to less need to create new stacks and by that, better performance.

Dynamic Stack: Popping

Popping is carried out similarly to pushing but with a key difference: It doesn't always trigger and its speed decreases when needed. This is to prevent unnecessary recreation since we don't want to shrink it to the base minimum each time. The method `pop()` shrinks the stack by calling `shrinkStack()` if the pointer is located in the lower half of the stack as seen in the logic below:

```
if (p + 1 < this.arr.length / 2) {
    this.shrinkStack();
}
```

When `shrinkStack()` is called, it works similarly to `growStack()`, and uses an accelerator called `shrinkSpeed`. It also has additional logic to prevent it from making the new array too small or equal to the current one. In addition, it can slow down if needed (see the if-statement in the code below). This fixes the issue with a high a `shrinkSpeed` when the size becomes too small.

```
growthSpeed = 0;
int t = this.arr.length;
int newSize = t - ++shrinkSpeed * 2;
if (newSize == t || newSize < p + 1 || newSize < SIZE) {
        if (shrinkSpeed > 0) {
                shrinkSpeed--;
        }
        return;
}
int[] newArr = new int[newSize];
copyArrayContent(this.arr, newArr);
```

Summarizing the pushing and popping, we get that the stack changes in size at an accelerating and sometimes decelerating speed. Growing twice in a row leads to a growth of $2 + 4 = 6$ in size. And shrinking three times in a row leads a a shrinking of $2 + 4 + 6 = 12$ in size. *Note that it doesn't always shrink because of the limitations.* This behavior can be compared to a person running back and forth to a wall. Upon leaving the wall, the person can run freely but needs to slow down eventually when running back to the wall.

Benchmarking

Each action a software does takes time. For example, creating a variable and printing to a console where some require more resources than others. This means that since the dynamic stack builds on more complex code than the static one, we know that it's going to be slower in theory because of the need to create new arrays and fill those with old data. This is compared to the static stack that only needs to pop and push thanks to its fixed size. The speed also comes down to the stack's initial size and what hardware the software runs on. Using a benchmark, the speed difference can be proven.

The first benchmark run (benchmark one) tested the push and pop max speed (based on 10 000 runs). It's simply built as it's based on two for-loops as seen below alongside its results in table 1. This benchmark used a static stack with a size fitted to the specific test, while the dynamic stack started with size 4 and later got to grow and shrink as needed.

```
for (int i = 0; i < testSize; i++) {
        stack.push(i);
}
for (int i = 0; i < testSize; i++) {
        stack.pop();
}
```

Size	Static(µs)	Dynamic(µs)	Diff.(µs)	Ratio(Dyn./Stat.)
1000	1	16	15	20
2000	8	44	36	6
3000	12	82	70	7
4000	16	120	104	7
5000	20	165	145	8
6000	24	209	184	9
7000	28	253	225	9
8000	32	304	272	10
9000	36	381	345	11
10000	40	458	418	11

Table 1: Benchmark one results.

The second benchmark (that builds on the HP35 calculator) performed $2*x-1$ pushes, $2x+1$ pops, and $x-1$ additions 10 000 times and calculated the average duration. This was done with a static stack of the size of $2*x-1$ and a dynamic stack with an initial size of 4. The results are illustrated in table 2.

x	Static(µs)	Dynamic(µs)	Diff.(µs)	Ratio(Dyn./Stat.)
10	0.312	0.834	0.522	2.67
100	1.52	4.61	3.1	2.85
1000	5.41	43.9	38.4	8.11
5000	46.1	245.0	199.0	5.31
10000	91.4	650.0	558.0	7.11

Table 2: Benchmark two results.

Discussion

The benchmark results in table 1 and table 2 prove that a static stack outperforms a dynamic one no matter the size. If we look at the two `Ratio(Dyn./Stat.)` columns, we can see that the static stack varies between 7-20 times faster than the dynamic one.

However dynamic stacks have other pros: They can for instance make development easier since they can vary in size and thus avoid stack overflow. They can also save memory at certain times, unlike a static stack of a fixed size (and therefore in some cases always takes up a lot of memory). On the other hand, using a static stack can prevent memory fragmentation since there is no risk of accessing the wrong address when reading/writing data. In the end, it comes down to the application and hardware and what type

of stack should be used. As well as the developer's preferences as this can prevent errors in the code.

Factors that can affect the benchmark results (but not change the size order based on stack type) are the algorithm for determining when to grow and shrink the stack, as well as how much it grows/shrinks (finding the optimal algorithm for this is demanding and will not be done for the report). But worth exploring would be exponential and logarithmic functions. On top of this, it is important to keep in mind that benchmark one looks for the best speed to find more "correct" results since this value will be the one with the least interference. Benchmark two on the other hand includes all results as it calculates an average. This benchmark acts as a more real-life scenario benchmark and can give worse results due to the interference that might arise during the run.