Stacks

Kristiāns Vinters

Fall 2023

Introduction

I solved the stack/calculator assignment in Go. I used Go because I want to become more familiar with it. Source code and benchmark data is available on GitHub[1].

Implementation

I structured the program into three sub-packages: `solver`, `stacks`, `token`. The main function in `cmd/stack/main.go` only creates a stack from the `stacks` package and passes it and a string RPN expression to `solver.Solve()`.

`stacks`

I utilize Go 1.18 generics to define generic stacks, but for the calculator I only allow a `float32` stack. The `stacks` package implements the `StaticStack` and `DynamicStack` and defines a `Stack` interface. The `Stack` interface defines generic stack methods, which are implemented by the specific stack structs.

```go
type Stack[T any] interface {
    Push(val T) error
    Pop() (T, error)
    Empty() bool
}
```

Both stack structs are defined as follows with only the type name being different:

```go
// or type DynamicStack[T any] struct { ...
type StaticStack[T any] struct {
    size int
```

[1]https://github.com/Phanty133/id1021/tree/master/stack

```go
    ip   int
    data []T
}
```

For both stack types, most of the method functionality is shared. If the stack is not empty, the `Push` method increments the instruction pointer and sets the value at the pointer to the passed value. The `Pop` method returns the value at the internal pointer and decrements the pointer. If the user attempts to pop an empty stack, the method returns an underflow error. The `Empty` method returns true if the internal pointer is -1.

`StaticStack`

A static stack is created with a constructor function, which expects a predefined fixed-length data array, because Go, as far as I am aware, does not support run-time fixed-length array allocation (run-time arrays are handled through slices). If the user attempts to push to a full stack, the `Push` method returns an overflow error.

```go
func NewStaticStack[T any](dataArr []T) *StaticStack[T] {
    return &StaticStack[T]{
        size: cap(dataArr),
        ip:   -1,
        data: dataArr,
    }
}
```

`DynamicStack`

A dynamic stack is created with a constructor function, which expects the initial size of the stack. The constructor allocates a slice of the specified size and returns a pointer to the stack struct. Although I use a slice, I perform under/overflow checks and reallocation manually.

```go
func NewDynamicStack[T any](dataArr []T) *DynamicStack[T] {
    return &DynamicStack[T]{
        size: initialSize,
        ip:   -1,
        data: make([]T, initialSize),
    }
}
```

To implement dynamic resizing, I defined a `Reallocate` method, which handles size changes and data copying:

```
func (stack *DynamicStack[T]) Reallocate(newSize int) {
    newData := make([]T, newSize)
    copy(newData, stack.data)
    stack.data = newData
    stack.size = newSize
}
```

Whenever I push to the stack, I perform a check for whether the stack is full. If it is, I double the size:

```
if stack.ip == stack.size - 1 {
    stack.Reallocate(stack.size * 2)
}
```

When I pop from the stack, I check whether the stack is less than a quarter full. If it is, I halve the size:

```
if stack.ip < stack.size / 4 {
    stack.Reallocate(stack.size / 2)
}
```

I decided to do it this way to avoid frequent resizing. By checking for quarter capacity and resizing only by half, I leave a buffer which should handle most minor push/pops.

`tokens`

The `token` package defines a `TokenType` enum for representing the operator type. The package also defines a token type-function map, which is used to map the operator type to the operator function.

```
var tokenFuncMap = map[TokenType]func(val1 float32, val2 float32) float32{
    ADD: func(val1 float32, val2 float32) float32 { return val1 + val2 },
    ...
}
```

`solver`

The `solver` package contains calculator functions which use the other two packages. The calculator is structured around an `Expression` struct.

```
type Expression[StackType stacks.Stack[float32]] struct {
    numStack StackType
    ops []token.Token
}
```

The package defines two internal `Expression` methods for parsing the string expression and solving it: `PopulateExpression` and `ParseExpression`. The `PopulateExpression` takes in a normalized[2] string expression and populates the value stack and operator array with values and operators, respectively.

The `ParseExpression` method takes in a populated `Expression` and solves it. It iterates over the operator array and applies the operator function to the top two values in the value stack. The result is pushed back onto the value stack. The method returns the final value in the value stack.

Stack benchmarks

To benchmark the stack implementations, I did 2000 runs, each of which consisted of a 1000 iterations of N pushes and N pops (sequentially as listed). The results for each run were then written to a CSV file. Code used for benchmarking is given in appendix A. I benchmarked both implementations for $\forall N \in \{100, 500, 1000, 2000, 5000\}$. Unsurprisingly, the static stack outperformed the dynamic stack in all cases, as per Fig. 1. Results for all N are given in appendix B. The speedup ratio of static over dynamic doesn't appear to change significantly over N (Fig. 2), averaging out at about 1.5.

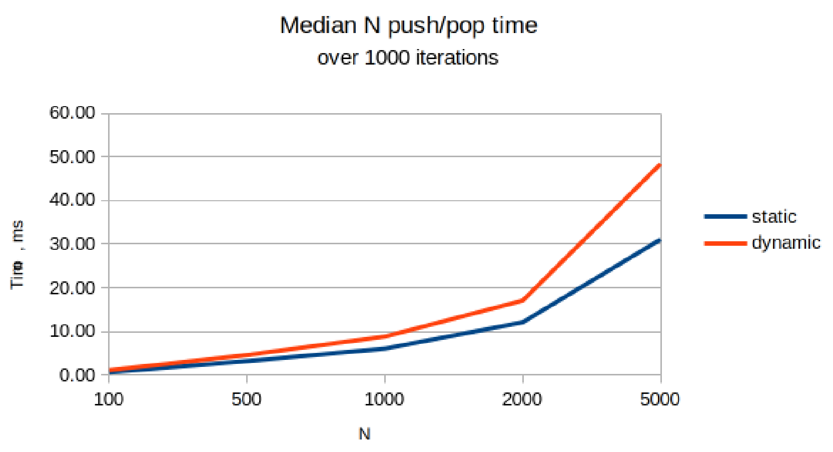

Figure 1: Median times for both implementations

[2]All multi-spaces replaced with single spaces according to the regular expression `/\s{2,}/`.

N	Speedup ratio
100	1.8
500	1.4
1000	1.5
2000	1.4
5000	1.6

Figure 2: $Ratio = \frac{Med(DynamicTimes)}{Med(StaticTimes)}$

An interesting observation is that the static stack run times are more consistent than the dynamic stack run times for all N. This is evident in the run time histograms (Fig. 3). The static stack times are more consistent most likely because the stack is pre-allocated and the run time is not affected by the size of the stack. On the other hand, the dynamic stack times are more inconsistent because the stack is dynamically reallocated, thus likely more susceptible to background processes.

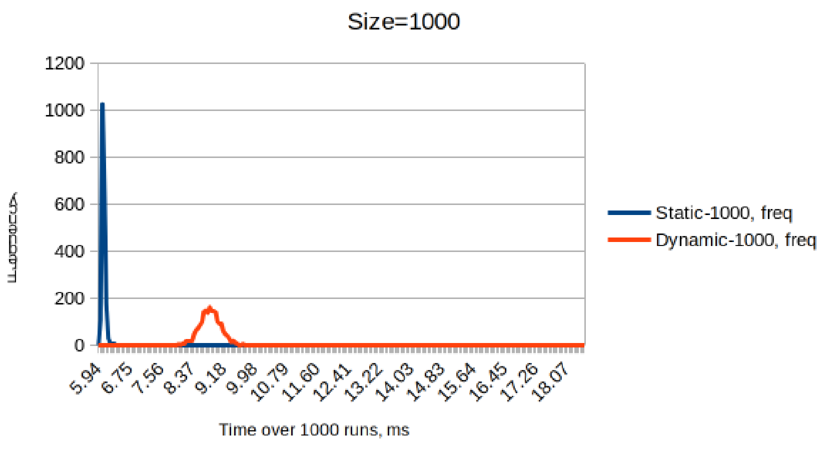

Figure 3: N=1000 run time histogram, all N histograms in appendix C

Appendices

A Benchmark code

```
func StackBench[StackType stacks.Stack[int]](
    tag string,
    stack StackType,
    runs int,
    runIters int,
```

```go
    stackIters int
) {
    outFile, err := os.Create(fmt.Sprintf("stack_%s.csv", tag))

    if err != nil {
        fmt.Println(err)
        return
    }

    defer outFile.Close()

    runTimes := make([]time.Duration, runs)

    for run := 0; run < runs; run++ {
        runStart := time.Now()

        for iter := 0; iter < runIters; iter++ {
            for i := 0; i < stackIters; i++ {
                stack.Push(i)
            }

            for i := 0; i < stackIters; i++ {
                stack.Pop()
            }
        }

        runTimes[run] = time.Since(runStart)
    }

    writer := csv.NewWriter(outFile)
    defer writer.Flush()

    for _, time := range runTimes {
        writer.Write([]string{fmt.Sprintf("%d", time.Microseconds())})
    }
}
```

B Benchmark results

Impl.	Min, ms	Max, ms	Mean, ms	Median, ms	Std.dev, ms
Static	0.60	0.91	0.62	0.62	0.01
Dynamic	0.9	2.7	1.1	1.1	0.16

Figure 4: N=100

Impl.	Min, ms	Max, ms	Mean, ms	Median, ms	Std.dev, ms
Static	3.02	5.17	3.20	3.20	0.14
Dynamic	4.0	6.6	4.6	4.6	0.24

Figure 5: Results for N=500

Impl.	Min, ms	Max, ms	Mean, ms	Median, ms	Std.dev, ms
Static	5.94	7.28	6.05	6.04	0.07
Dynamic	8.0	18.6	8.8	8.8	0.38

Figure 6: N=1000

Impl.	Min, ms	Max, ms	Mean, ms	Median, ms	Std.dev, ms
Static	11.95	15.43	12.18	12.08	0.26
Dynamic	15.8	21.8	17.1	17.1	0.46

Figure 7: N=2000

Impl.	Min, ms	Max, ms	Mean, ms	Median, ms	Std.dev, ms
Static	30.14	52.54	31.55	31.08	1.87
Dynamic	45.4	75.5	50.3	48.4	4.08

Figure 8: N=5000

C Run time histograms

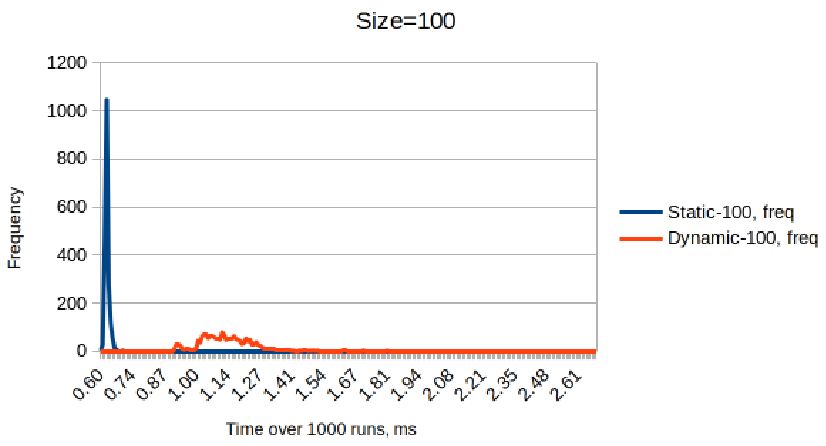

Figure 9: N=100

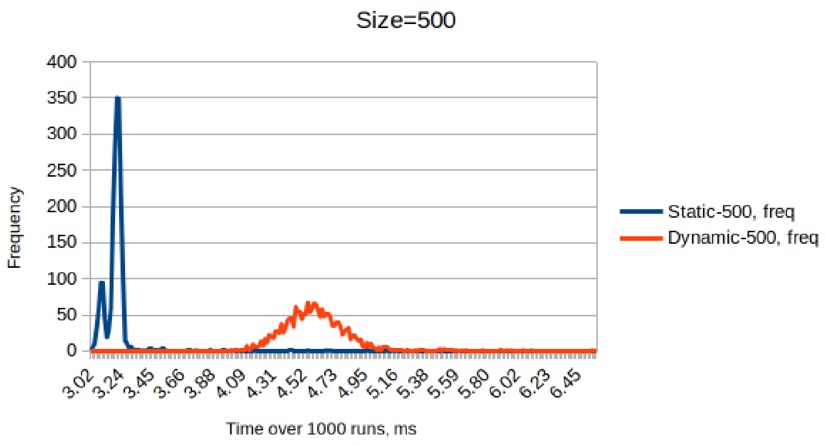

Figure 10: N=500

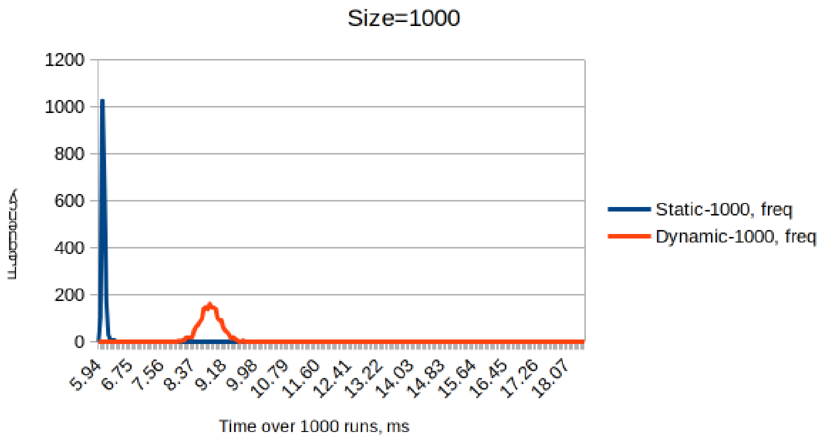

Figure 11: N=1000

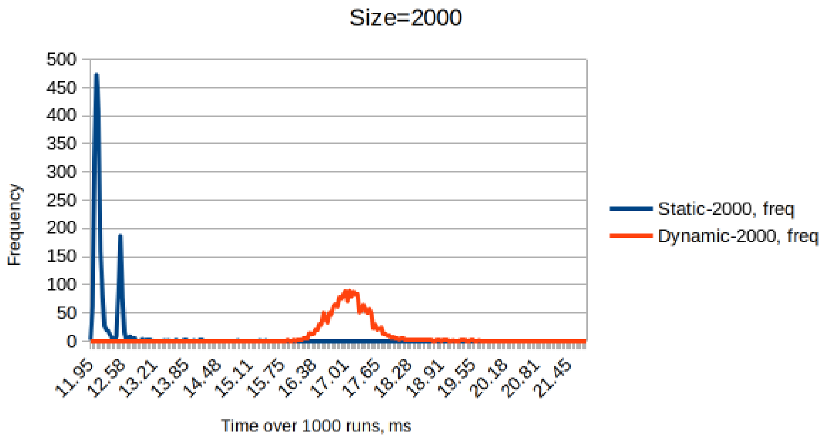

Figure 12: N=2000

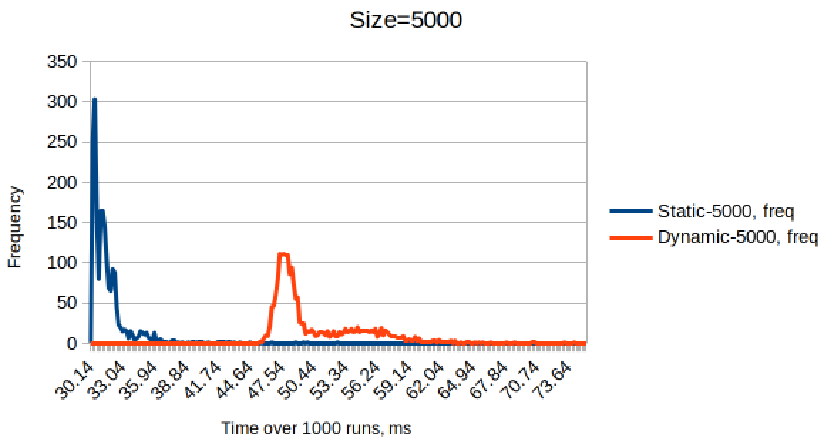

Figure 13: N=5000