Gina Cody Shool of Computer Science and Software Engineering
Concordia University
COMP 352: Data Structure and Algorithms

Student: Duc Nguyen

Contents

1 Double-stack array

NOTE: All the Java implementation of this question is in the folder Implementation. This is out of scope for this assignment, but it will help anyone reading this get a better understanding.

1.1 Case 1:

Fairness in space allocation to the two stacks is required. In that sense, if Stack1 for instance use all its allocated space, while Stack 2 still has some space; insertion into Stack 1 cannot be made, even though there are still some empty elements in the array.

A. Description

In this scenario, the array could be implemented by allocating the range from starting index to the maximum capacity index to be the reserved space for Stack 1 (left-to-right order). While stack 2 takes place from the last index of the array, goes left until it reaches its maximum capacity (right-to-left order).

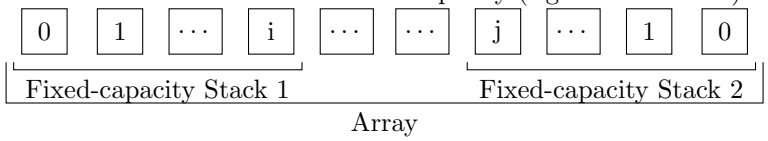


Illustration of the implementation in this case. The index is of stack that it belongs to, not the "real" index of the array.

Global variables:

- array: the array itself
- arrayLength: The length of the array
- headStack1(i): The index of the last element in Stack 1. (starts from -1 when the stack is empty)
- stack1Capacity: the fixed given capacity of Stack 1.
- headStack2(j): The index of the last element in Stack 2. (starts from arrayLength when the stack is empty)
- stack2Capacity: the fixed given capacity of Stack 2.

B. Algorithms for push(), pop(), size(), isEmpty(), is Full()

Algorithm 1 Push Element to Stack 1

```
1: procedure PushStack1(element)
2:     if isStack1Full() = false then
3:         headStack1 ← headStack1 + 1
4:         array[headStack1] ← element
5:     else
6:         throw FullStackException
```

Algorithm 2 Push Element to Stack 2

```
1: procedure PushStack2(Element)
2:     if isStack2Full() = false then
3:         headStack2 ← headStack2 − 1
4:         array[headStack2] ← element
5:     else
6:         throw FullStackException
```

Algorithm 3 Pop Element out of Stack 1. Return the popped element

```
1: procedure popStack1()
2:     if isStack1Empty() = false then
3:         temp ← array[headStack1]
4:         array[headStack1] ← null
5:         headStack1 ← headStack1 - 1
6:         return temp
7:     else
8:         throw EmptyStackException
```

Algorithm 4 Pop Element out of Stack 2. Return the popped element

```
1: procedure popStack2()
2:     if isStack2Empty() = false then
3:         temp ← array[headStack2]
4:         array[headStack2] ← null
5:         headStack2 ← headStack2 + 1
6:         return temp
7:     else
8:         throw EmptyStackException
```

Algorithm 5 Return an integer indicates the current size of Stack 1

```
procedure sizeStack1()
    return headStack1 + 1
```

Algorithm 6 Return an integer indicates the current size of Stack 2

```
procedure sizeStack2()
    return arrayLength - headStack2
```

Algorithm 7 Return a boolean value indicates whether or not Stack 1 is empty

```
procedure isStack1Empty()
    if headStack1 = −1 then
        return true
    else
        return false
```

Algorithm 8 Return a boolean value indicates whether or not Stack 2 is empty

```
procedure isStack2Empty()
    if headStack2 = arrayLength then
        return true
    else
        return false
```

Algorithm 9 Return a boolean value indicates whether or not Stack 1 is full

```
procedure isStack1Full()
    if sizeStack1() = stack1Capacity then
        return true
    else
        return false
```

Algorithm 10 Return a boolean value indicates whether or not Stack 2 is full

```
procedure isStack2Full()
    if sizeStack2() = stack2Capacity then
        return true
    else
        return false
```

C. Big-O complexity

Since every algorithm performs with a constant time:

- Push() - $O(1)$
- Pop() - $O(1)$
- Size() - $O(1)$
- isEmpty() - $O(1)$
- isFull() - $O(1)$

D. Big-$\Omega()$ complexity

- Push() - $\Omega(1)$
- Pop() - $\Omega(1)$
- Size() - $\Omega(1)$
- isEmpty() - $\Omega(1)$
- isFull() - $\Omega(1)$

1.2 Case 2:

Space is critical; so you should use all available elements in the array if needed. In other words, the two stacks may not finally get the same exact amount of allocation, as one of them may consume more elements (if many push() operations are performed for instance into that stack first).

A. Description

The ADT can be implemented by the same way as the previous case. However, in this case, it keeps pushing new element into both stacks without a fixed limit for the capacity. A stack is only full when the array is full of elements.

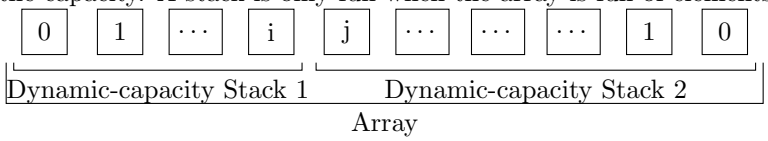


Illustration of the implementation in this case. The index is of stack that it belongs to, not the "real" index of the array.

Global variables:

- array: the array itself
- arrayLength: The length of the array
- headStack1: The index of the last element in Stack 1. (starts from -1 when the stack is empty)
- headStack2: The index of the last element in Stack 2. (starts from arrayLength when the stack is empty)

B. Algorithms for push(), pop(), size(), isEmpty(), is Full()

Algorithm 11 Push Element to Stack 1

```
1: procedure PushStack1(element)
2:     if isFull() = false then
3:         headStack1 ← headStack1 + 1
4:         array[headStack1] ← element
5:     else
6:         throw FullStackException
```

Algorithm 12 Push Element to Stack 2

```
1: procedure PushStack2(Element)
2:     if isFull() = false then
3:         headStack2 ← headStack2 − 1
4:         array[headStack2] ← element
5:     else
6:         throw FullStackException
```

Algorithm 13 Pop Element out of Stack 1. Return the popped element

```
1: procedure popStack1()
2:     if isStack1Empty() = false then
3:         temp ← array[headStack1]
4:         array[headStack1] ← null
5:         headStack1 ← headStack1 - 1
6:         return temp
7:     else
8:         throw EmptyStackException
```

Algorithm 14 Pop Element out of Stack 2. Return the popped element

```
1: procedure popStack2()
2:     if isStack2Empty() = false then
3:         temp ← array[headStack2]
4:         array[headStack2] ← null
5:         headStack2 ← headStack2 + 1
6:         return temp
7:     else
8:         throw EmptyStackException
```

Algorithm 15 Return an integer indicates the size of Stack 1

```
procedure sizeStack1()
    return headStack1 + 1
```

Algorithm 16 Return an integer indicates the size of Stack 2

```
procedure sizeStack2()
    return arrayLength - headStack2
```

Algorithm 17 Return a boolean value indicates whether or not Stack 1 is empty

```
procedure isStack1Empty()
    if headStack1 = −1 then
        return true
    else
        return false
```

Algorithm 18 Return a boolean value indicates whether or not Stack 2 is empty

```
procedure isStack2Empty()
    if headStack2 = arrayLength then
        return true
    else
        return false
```

When the size of both stack is dynamic, the only time a stack is full is when the array is full.

Algorithm 19 Return a boolean value indicates whether or not a stack is full

```
procedure isFull()
    if headStack1 + 1 = headStack2 then
        return true
    else
        return false
```

C. Big-O complexity

Since there was no loop, or recursive calls in every implemented method, and every algorithm performs in a constant amount of time:

- Push() - $O(1)$

- Pop() - $O(1)$
- Size() - $O(1)$
- isEmpty() - $O(1)$
- isFull() - $O(1)$

D. Big-Ωcomplexity

- Push() - $\Omega(1)$
- Pop() - $\Omega(1)$
- Size() - $\Omega(1)$
- isEmpty() - $\Omega(1)$
- isFull() - $\Omega(1)$

1.3 Concerning the possibility of a 3-stack array

- **Case 1:** It is possible to implement a three-stack array. The third stack will lie between the first stack, and the second stack. In this case, the complexity will not change significantly since every stack has reserved its own space inside the array.

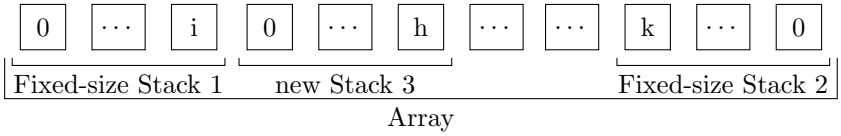


Illustration of one possible implementation in this case. The index is of stack that it belongs to, not the "real" index of the array.

- **Case 2:** It is possible to implement a three-stack array. To optimize all space inside the array, the new third stack starts right after the end of the first stack (after headStack1) , but it must end before the end of stack 2 (before headStack2). This leads to a significant change in time complexity since the stack 3 will have to shift all its elements every time there is a new push or pop operation in stack 1.

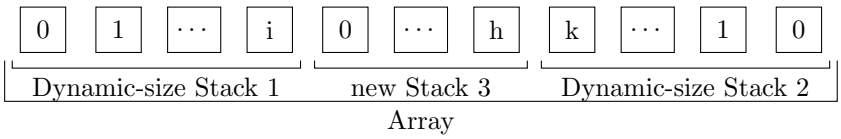


Illustration of one possible implementation in this case. The index is of stack that it belongs to, not the "real" index of the array.

2 Antique dealer

2.1 Pseudo code of max() function in a stack based array

Variables:

- *stack:* the stack itself implemented in the array
- *headStack:* the last index of the stack inside the array.

Algorithm 20 Return the maximum value of the inventory

```
1: procedure MAX()
2:     max ← stack[0]
3:     for k ← 1 to headStack do
4:         if stack[k] > max then
5:             max ← stack[k]
6:     return max
```

2.2 Big-O complexity

Since the algorithm has to loop through every existing element inside the stack, the complexity of the algorithm is O(n).

2.3 Possible to have <u>all</u> 3 methods (push(), pop(), and max()) be designed to have O(1)?

It is possible to have all 3 methods push(), pop() and max() to have O(1). It could be implemented by using a singly linked list. Each *node* inside the linked list would contain a variable *Max* to keep track of the max value each time a new element is pushed in, along with *Next* serves as a poiner, links to the next node. In addition, the top of the stack is being implemented at the *Head* of the linked list.

Demonstration

Algorithm 21 Push new element into the inventory

```
1: procedure PUSH(NEWVALUE)
2:     temp ← Node(newValue)          ▷ Create a new Node with newValue
3:     if head ≠ null then
4:         temp.max ← maxOf(head.max, newValue)
5:         temp.next ← head
6:         head ← temp
7:     else                                  ▷ Stack is currently empty
8:         temp.max ← newValue
9:         head ← temp
```

Algorithm 22 Pop and return the last element

```
1: procedure POP()
2:     if head = null then                ▷ The stack is currently empty
3:         return null
4:     else
5:         temp ← head
6:         head ← head.next
7:         return temp
```

Algorithm 23 Return the max value in the inventory

```
1: procedure MAX()
2:     if head = null then return 0       ▷ The stack currently is empty
3:     else
4:         return head.max
```

Note: All the algorithms here were implemented in the folder *Implementation* to give readers better understanding.

3 Evaluate complexity's relationship

a. $f(n) = \log^3 n$ $g(n) = \sqrt{n}$
b. $f(n) = n\sqrt{n} + \log n$ $g(n) = \log n^2$
c. $f(n) = n$ $g(n) = \log^2 n$
d. $f(n) = \sqrt{n}$ $g(n) = 2^(\sqrt{\log n})$
e. $f(n) = 2^n$ $g(n) = 3^n$

a. $f(n) = \log^3 n$ is of power 3 while $\sqrt{n}$ is a function of square root. Thus, the growth rate of f(n) is slower than g(n). $\longrightarrow f(n)$ **is of** $O(g(n))$

b. $f(n) = n\sqrt{n} + \log n$ has a greater growth rate when comparing with growth rate of $g(n) = \log n^2$. $\longrightarrow f(n)$ **is of** $\Omega(g(n))$

c. $f(n) = n$ is a function with linear growth rate while $g(n) = \log^2 n$ is a function of log. The linear growth is faster than log function. Thus, $f(n)$ **is of** $\Omega(g(n))$

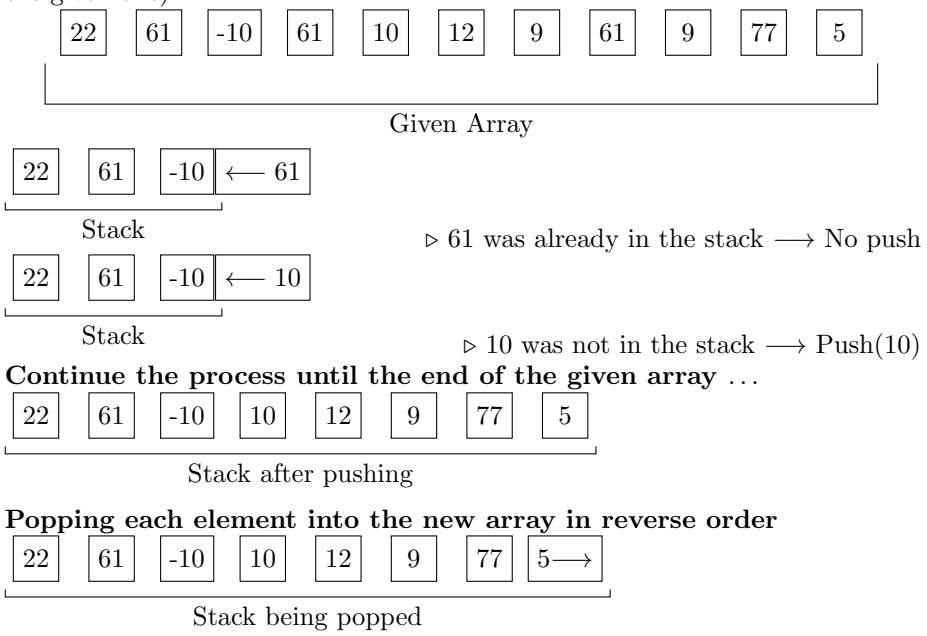
d. While $f(n) = \sqrt{n}$ is of square root function, the function $g(n) = 2^(\sqrt{\log n})$ is of exponential. However, the exponential of g(n) is a square root function. That leads to a slightly slower rate of growth in g(n). $\longrightarrow f(n)$ **is of** $\Omega(g(n))$

e. $f(n) = 2^n$ is an exponential function with base 2; while $g(n) = 3^n$ is an exponential function with base 3. Thus, $g(n) > f(n) \forall n \geq 1$. $\longrightarrow f(n)$ **is of O(g(n))**

4 Remove duplicate elements

4.1 Description

Create a stack firstly. Then make a loop through each element inside the given array. If the element was already pushed into the array, then the program will not re-push it again. In other word, the program only pushes the element which is not the duplication of any existing elements inside the Stack. After finish pushing elements into the Stack, create a new array which has the same size with the Stack. Then, the algorithm pops each element in the Stack into the array in reverse order (to guarantee the order in the new array is the same as the given one).



Algorithm 24 Receive input as an integer *array*, and return output as a new array which has no duplicate items.

```
1: procedure REMOVEDUPLICATE(INT[] ARRAY)
2:     stack ← new Stack of Integer
3:     for each element ∈ Array do          ▷ Pushing elements into the Stack
4:         if stack.contain(element) == false then
5:             stack.push(element)
6:     result ← new int[stack.size()]       ▷ Popping Stack elements to result
7:     for (i ← result.length − 1; i ≥ 0; i − −) do
8:         result[i] = stack.pop()
9:     return result
```

4.2 Big-O Complexity

Big-O complexity of this solution would be $O(n^2)$ since the algorithm has to loop through each element inside the array once **(n)**. Each element has to be compared to each existing element inside the stack **(n)** in order to make decision whether or not push this new element into the stack. Lastly, popping every element inside the stack to the new result array is of **O(n)**. Thus, it is $O(n * n + n) = O(n^2)$.

4.3 Big-Ω Complexity

Best case scenario in this situation is when the algorithm tries to determine should the element be pushed to Stack or not, the duplicate item is already at the front of the Stack; thus, there is no need to check other elements. Therefore, best case scenario is when the given array is full of duplicate items. As a result, the time complexity of the algorithm is of $\Omega(n)$.

4.4 Big-O *space* complexity

The worst case scenario for the memory is when there is no duplication in the given array. Then the size of the Stack is the same with the array. As a result, the algorithm space complexity is of **O(n)**