

Stack (I): Introduction and the Array-Based Implementation

CSCD 300 – Data Structures

Eastern Washington University

© Bojian Xu, Eastern Washington University. All rights reserved.

Goal

In this lecture, we will learn the basic concept of the *stack* data structure. Then we will demonstrate an array-based implementation of this data structure.

Outline

- 1 Web browser's "go back" functionality
- 2 The Stack data structure
- 3 An array-based implementation

Web browser's “go back” functionality

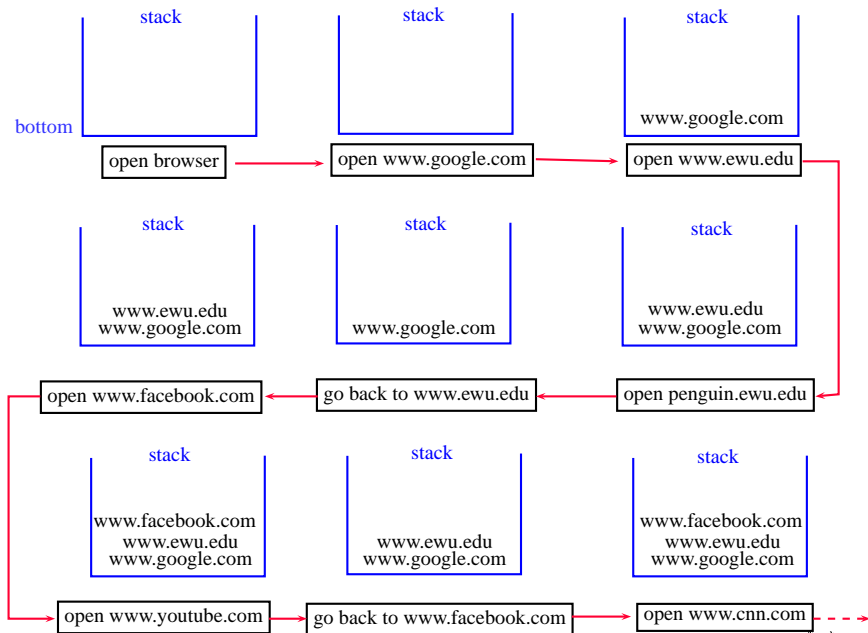
Most web browsers have the “go back” button or menu item that allow the users to go back to the previous URL. How is this functionality implemented ? It uses a data structure called *stack*.

A *stack* can be viewed as a container that stores items into a pile — the more recently received items are on the top of the older ones.

For the particular application of web browser:

- Whenever a webpage is opened, the previous page's URL is *pushed* into the stack and becomes the top item in the stack.
- Whenever the “go back” button is clicked, the top URL in the stack is *pop* out from the stack and is then opened.

An example follows ...



The *Stack* data structure

- A stack stores items in the order of **first-in-last-out**.
- A stack maintains its current **size**: the number of items being stored.
- A stack maintains the **top** reference which points to the top item.
- A stack supports the **pop** operation: extract the top item from the stack.
- A stack supports the **push** operation: save a new item into the stack and the new item's position is on the top of the stack.
- Plus a few other utility member fields and methods, depending on the needs from applications.

The above “**logical**” functionalities of the stack data structure can be implemented by using different “**physical**” data structures. We will discuss:

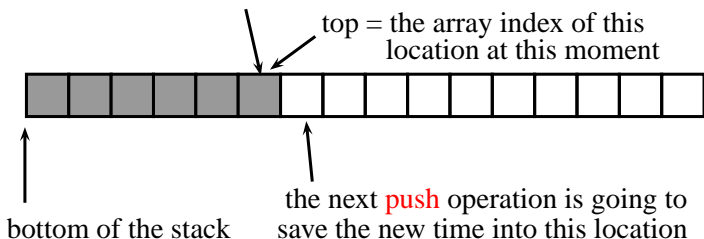
- an array-based implementation of the stack in this lecture.
- a singly linked list-based implementation of the stack in the next lecture.

An array-based implementation

- We allocate an array of some **fixed** capacity. Initially, every location of the array is not occupied.
- The **left** boundary of the array is viewed as the **bottom** of the stack.
- The **size** of the stack is the number of items being saved in the array, which is of course bounded by the array capacity.
- The **top** reference is the array index of the **rightmost** occupied array location.
- The **push** operation is to save the new item in the **leftmost** unoccupied array location, if such location exists.
- The **pop** operation is to **return and remove** the item saved in the **rightmost** occupied array location, if such location exists.


An example follows ...


the next **pop** operation is going to remove and return this item



stack size = 6 at this moment

array capacity = 16

 occupied array location

 empty array location

Pseudocode ¹

Initialization

```
Init an array of 'capacity' size;  
stack.size = 0;  
top = -1;
```

```
push(item)  
{  
    if(top == array.capacity-1)  
        report "stack is full";  
    return;  
    top ++;  
    array[top] = item;  
    stack.size ++;  
}
```

```
pop()  
{  
    if(top == -1)  
        report "empty stack";  
    return;  
    stack.size --;  
    top --  
    return array[top+1];  
}
```

See the attached Java code for the array-based implementation.

¹We use 0-based indexing.