COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

Computer Information System
University of Fraser Valley

* Some slides from “Algorithms and Data Structures”
by Douglas Wilhelm Harder

Stack

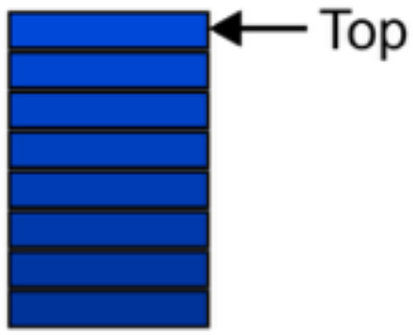
Abstract Stack

- An Abstract Stack (Stack ADT) is an abstract data type which emphasizes specific operations:
  - `push(Object obj)`: Inserted objects are ***pushed onto*** the stack
  - `top()`: The ***top*** of the stack is the most recently object pushed onto the stack
  - `pop()`: When an object is ***popped*** from the stack, the current ***top*** is erased
- Other properties of abstract stack:
  - Uses a explicit linear ordering
  - Insertions and removals are performed individually (you cannot define a concatenation (like what we implemented for lists) method for the class of stack

Abstract Stack

Also called a ***last-in–first-out*** (LIFO) list

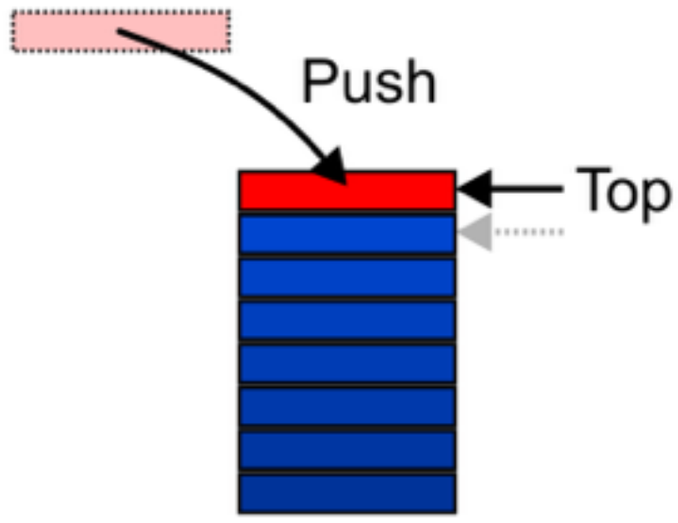
- Graphically, we may view these operations as follows:
- `top()`: returns the last item pushed to the stack. It is also called `peek()` in some text books.



Abstract Stack

Also called a *last-in–first-out* (LIFO) list

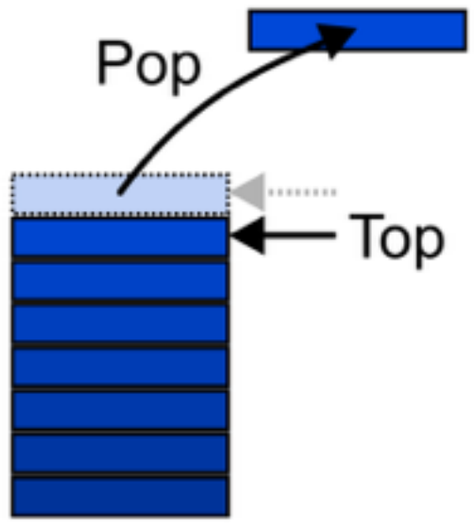
- Graphically, we may view these operations as follows:
- `push(Object obj)`: insert the given object (`obj`) to the top of the stack.



Abstract Stack

Also called a *last-in–first-out* (LIFO) list

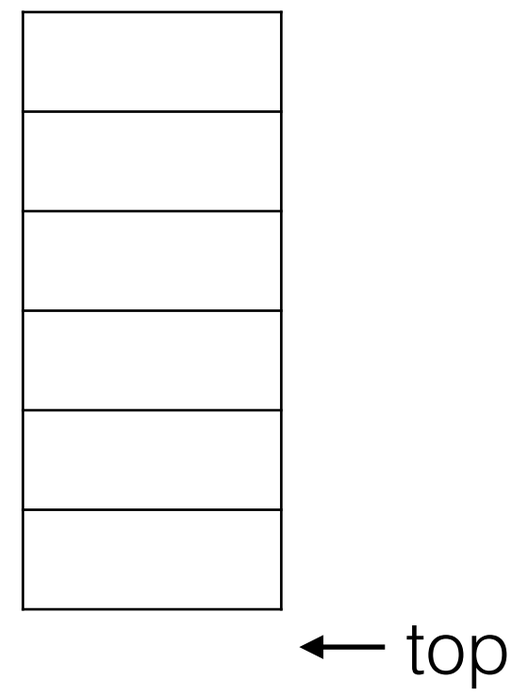
- Graphically, we may view these operations as follows:
- `pop()`: remove the last item pushed (inserted) to the stack and return it.



It is an undefined operation to call either pop or top on an empty stack (you should throw an exception).

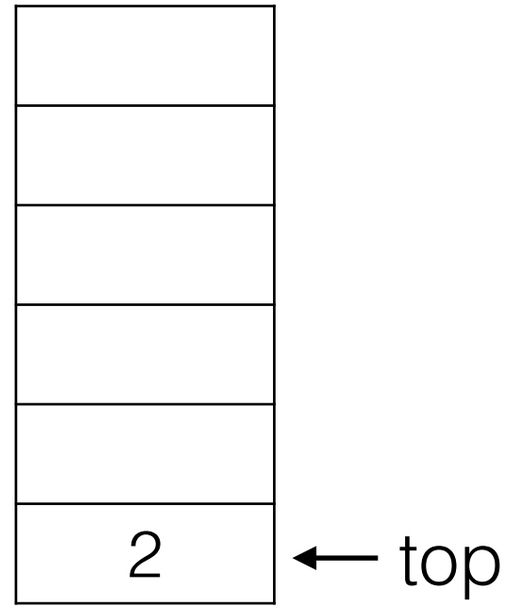
Example

- Assume we have a stack for integer values. The stack is empty at the beginning. Then we apply a set of operations.



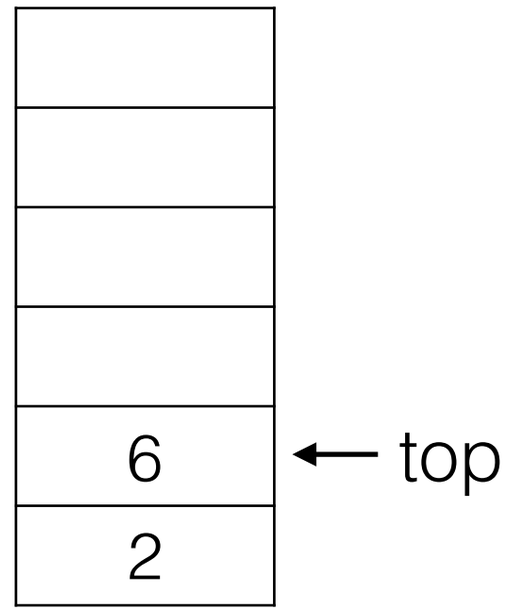
Example

- Assume we have a stack for integer values. The stack is empty at the beginning. Then we apply a set of operations.
  - push(2)



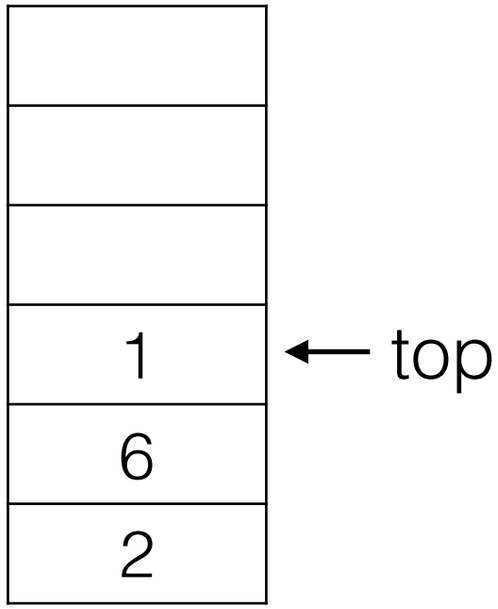
Example

- Assume we have a stack for integer values. The stack is empty at the beginning. Then we apply a set of operations.
  - push(2)
  - push(6)



Example

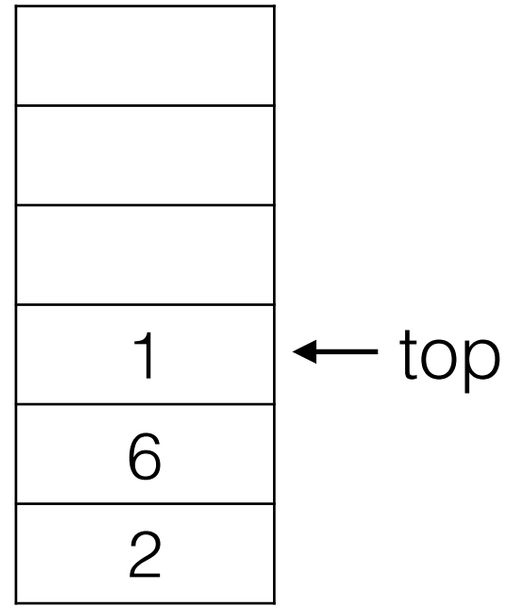
- Assume we have a stack for integer values. The stack is empty at the beginning. Then we apply a set of operations.
  - push(2)
  - push(6)
  - push(1)



Example

- Assume we have a stack for integer values. The stack is empty at the beginning. Then we apply a set of operations.

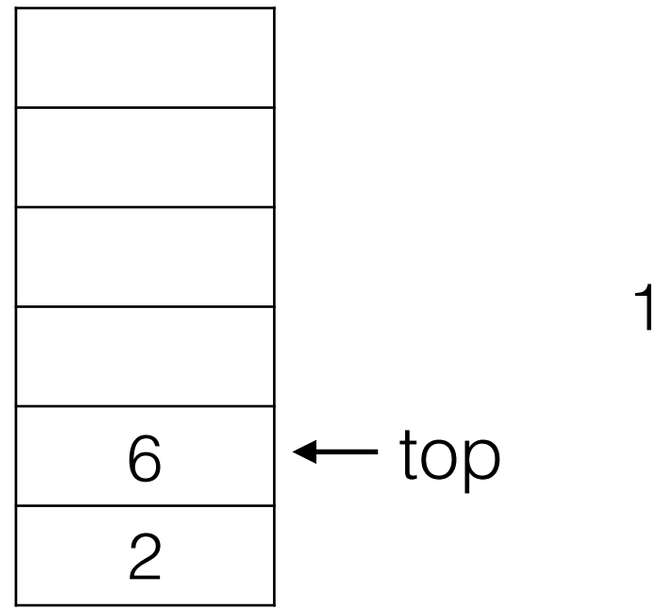
  - push(2)
  - push(6)
  - push(1)
  - top() //just return value 1



Example

- Assume we have a stack for integer values. The stack is empty at the beginning. Then we apply a set of operations.

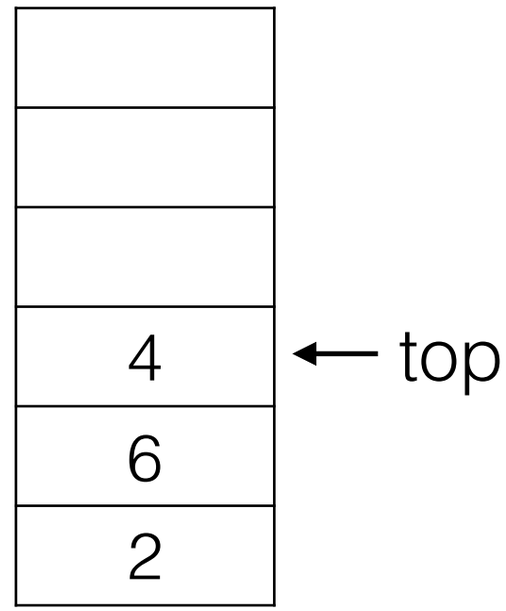
  - push(2)
  - push(6)
  - push(1)
  - top()
  - pop() //remove 1 and return it



Example

- Assume we have a stack for integer values. The stack is empty at the beginning. Then we apply a set of operations.

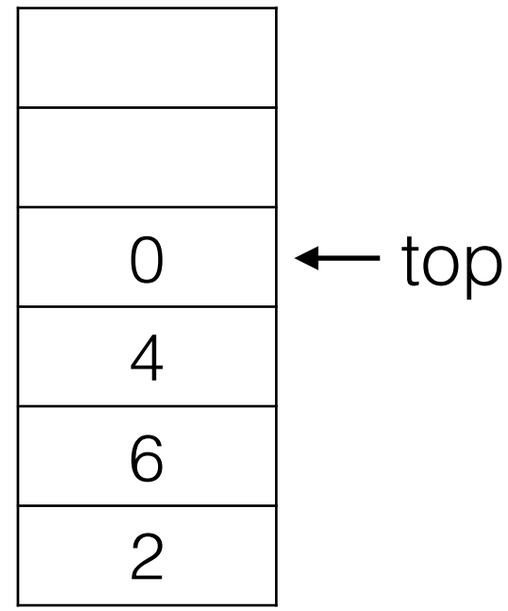
  - push(2)
  - push(6)
  - push(1)
  - top()
  - pop()
  - push(4)



Example

- Assume we have a stack for integer values. The stack is empty at the beginning. Then we apply a set of operations.

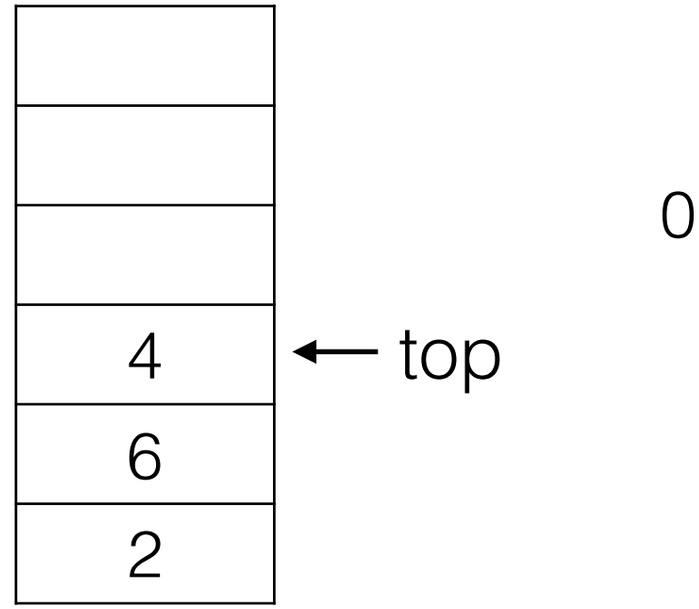
  - push(2)
  - push(6)
  - push(1)
  - top()
  - pop()
  - push(4)
  - push(0)



Example

- Assume we have a stack for integer values. The stack is empty at the beginning. Then we apply a set of operations.

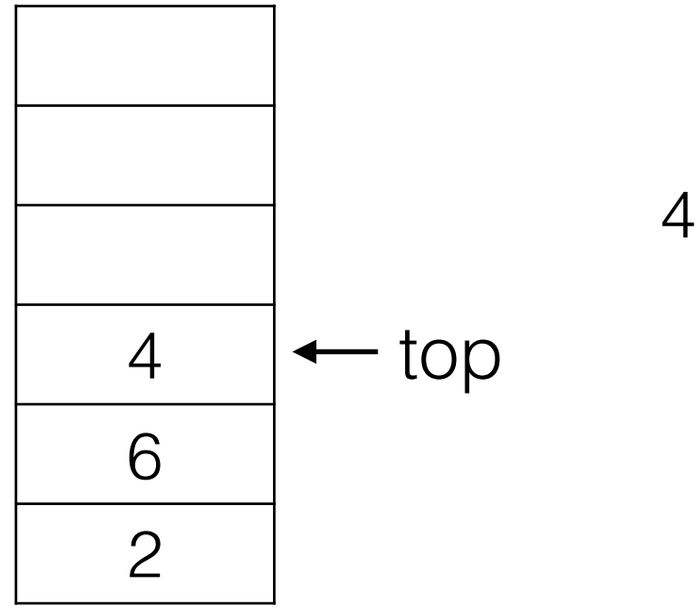
- push(2)
- push(6)
- push(1)
- top()
- pop()
- push(4)
- push(0)
- pop()



Example

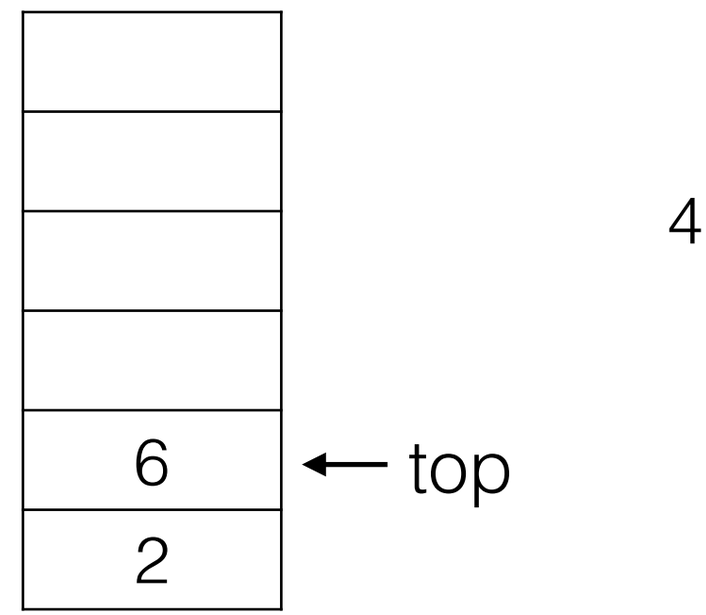
- Assume we have a stack for integer values. The stack is empty at the beginning. Then we apply a set of operations.

- push(2)
- push(6)
- push(1)
- top()
- pop()
- push(4)
- push(0)
- pop()
- top()



Example

- Assume we have a stack for integer values. The stack is empty at the beginning. Then we apply a set of operations.
  - push(2)
  - push(6)
  - push(1)
  - top()
  - pop()
  - push(4)
  - push(0)
  - pop()
  - top()
  - pop()



Applications

Numerous applications:

- Parsing code:
  - Matching parenthesis
  - Matching XML tags (e.g., XHTML)
- Tracking function calls (stack frames)
- Dealing with undo/redo operations
- Assembly language

The stack is a very simple data structure

- Given any problem, if it is possible to use a stack, this significantly simplifies the solution

Implementations

We will look at two implementations of stacks:

The optimal asymptotic run time of any algorithm is $\Theta(1)$

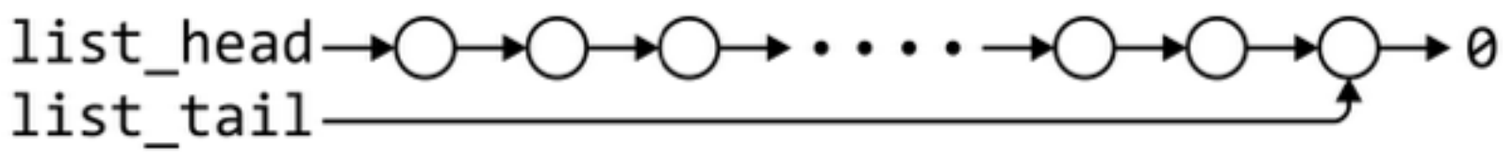
- The run time of the algorithm is independent of the number of objects being stored in the container
- We will always attempt to achieve this lower bound

We will look at

- Singly linked lists
- One-ended arrays

Linked-List Implementation

Operations at the ***front*** of a singly linked list are all $\Theta(1)$



	Front/1st	**End/n^{th}**
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

The desired behaviour of an Abstract Stack may be reproduced by performing all operations at the front

`SList` Definition

The definition of singly linked list class (`SList`) from lab2 is:

```
public class SList {
   private SListNode head;
   private SListNode tail;
   private int size;
  //methods:
   public boolean isEmpty();
   public int length();
   public void insertFront(Object obj);
   public void insertEnd(Object obj);
   public Object nth(int position);
   public Object removeEnd();
   public Object removeFront();
}
```

Stack Class (linked list)

The stack class using a singly linked list has a single private member variable:

we use an object of the SList

```
public class LinkListStack {
  // *** fields ***
   private SList list;  // an object of SList

  //*** methods ***
  // constructor
    public LinkListStack() { ... }
  // add items
    public void push(Object obj) { ... }
  // remove items
    public Object pop() throws EmptyStackException { ... }
  // other methods
    public Object top() throws EmptyStackException { ... }
    public int length() { ... }
    public boolean isEmpty() { ... }
}
```

Stack Class (linked list)

Try to write a constructor:

```
public class LinkListStack {
  // *** fields ***
   private SList list;

  //*** methods ***
  // constructor
    public LinkListStack() { ... }

    .
    .
    .
}
```

Stack Class (linked list)

Try to write a constructor:

```
public class LinkListStack {
  // *** fields ***
   private SList list;

  //*** methods ***
  // constructor
    public LinkListStack() { ... }

     .
     .
     .
}
```

```
public LinkListStack() {
        list = new SList();
}
```

Stack Class (linked list)

The empty and push functions just call the appropriate functions of the **`SList`** class

```
// push an item
  public void push(Object obj) { . . . }
```

Stack Class (linked list)

The empty and push functions just call the appropriate functions of the **`SList`** class

```
// push an item
  public void push(Object obj) { . . . }
```

```
public void push(Object obj) {
        list.insertFront(obj);
}
```

Stack Class (linked list)

The top and pop functions, however, must check the boundary case:

```
public Object pop() throws EmptyStackException {
   if (list.isEmpty()) throw new EmptyStackException();
   return list.removeFront();
}
```

```
public Object top() throws EmptyStackException {
   if (list.isEmpty()) throw new EmptyStackException();
   return list.nth(1);
}
```

Stack Class (linked list)

- `SList` (singly linked list) has two ends. And we have access to both ends (`head` and `tail`).
- Note that it is up to us as the designers of the Stack class to decide which end of the linked-list corresponds to the top of the stack.
- Why did we use the front end?

Stack Class (Array based)

- We could choose:
  - always add items at the beginning of the array
  - always add items at the end of the array.

Stack Class (Array based)

- We could choose:
  - always add items at the beginning of the array
  - always add items at the end of the array.
- However, it is clearly not a good idea to add items at the beginning of the array since that requires moving all existing items; i.e., that choice would make push and pop to be $\Theta(n)$ (where n is the number of items in the stack).
- If we add items at the end of the array, then both push and pop will be $\Theta(1)$ (except when the array is full).

Stack Class (Array based)

For arrays, all operations at the back are $\Theta(1)$

	Front/1^{st}	End/n^{th}
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(n)$	$\Theta(1)$
Erase	$\Theta(n)$	$\Theta(1)$

Stack Class (Array based)

We need to store an array and a field to keep the size:

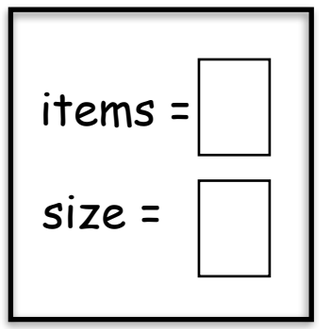
```
public class Stack {
  // *** fields ***
   private static final int INITSIZE = 1000;  // initial array size
   private Object[] items;          // array to keep items in the stack
   private int size;          // the number of items in the stack

  //*** methods ***
  // constructor
   public Stack() { ... }
  // add items
   public void push(Object obj) { ... }
  // remove items
   public Object pop() throws EmptyStackException { ... }
  // other methods
   public Object top() throws EmptyStackException { ... }
   public int length() { ... }
   public boolean isEmpty() { ... }
}
```

Constructor

The class is only storing the address of the array

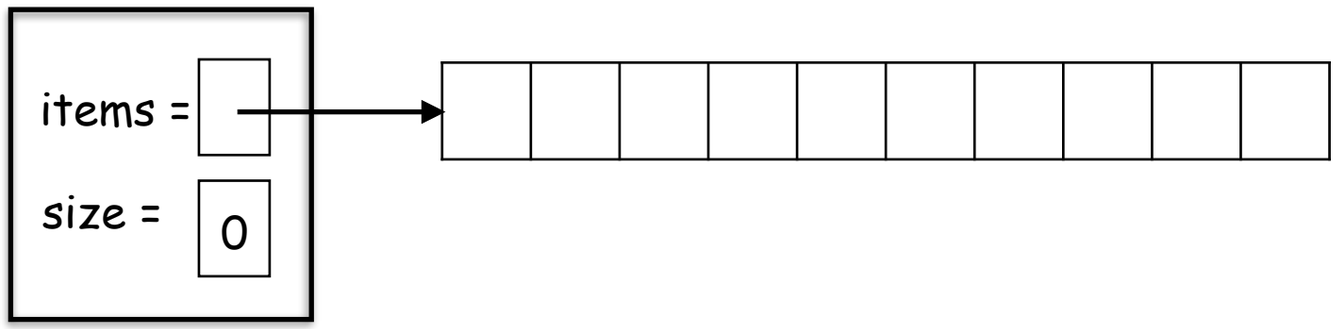
- We must allocate memory for the array and initialize the size



Constructor

The class is only storing the address of the array

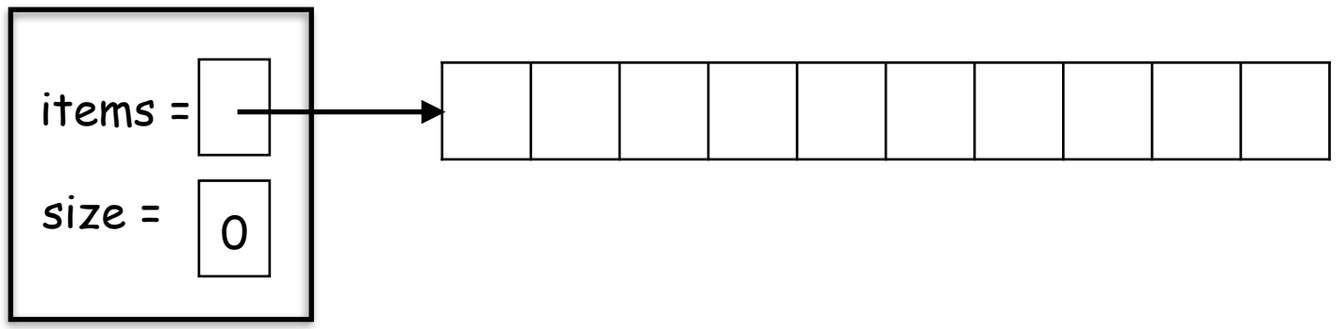
- We must allocate memory for the array and initialize the size and top item



Constructor

The class is only storing the address of the array

- We must allocate memory for the array and initialize the size and top item



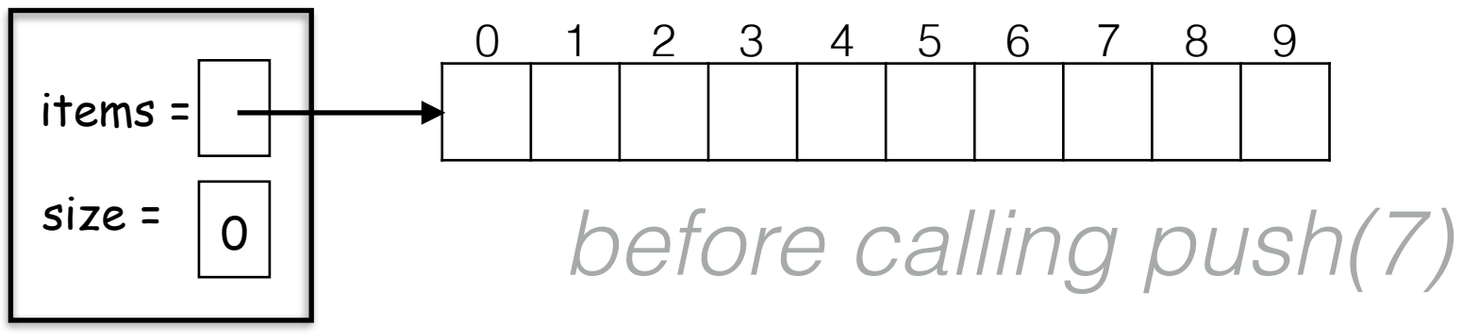

```
public Stack() {
   items = new Object[INITSIZE];
   size = 0;
}
```

Push

Push an object to the top of stack

- Note that in array implementation top is on the back of array.

```
// add items
public void push(Object obj) { ... }
```



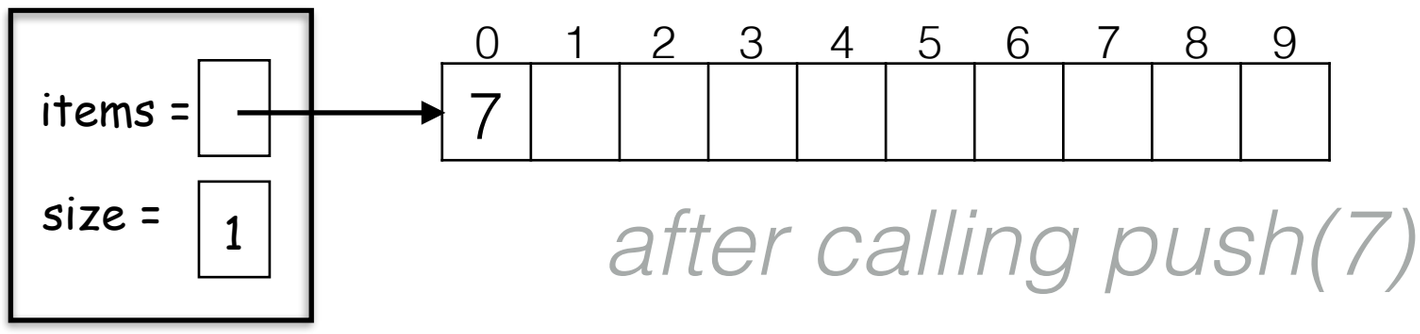
before calling push(7)

Push

Push an object to the top of stack

- Note that in array implementation top is on the back of array.

```
// add items
public void push(Object obj) { ... }
```



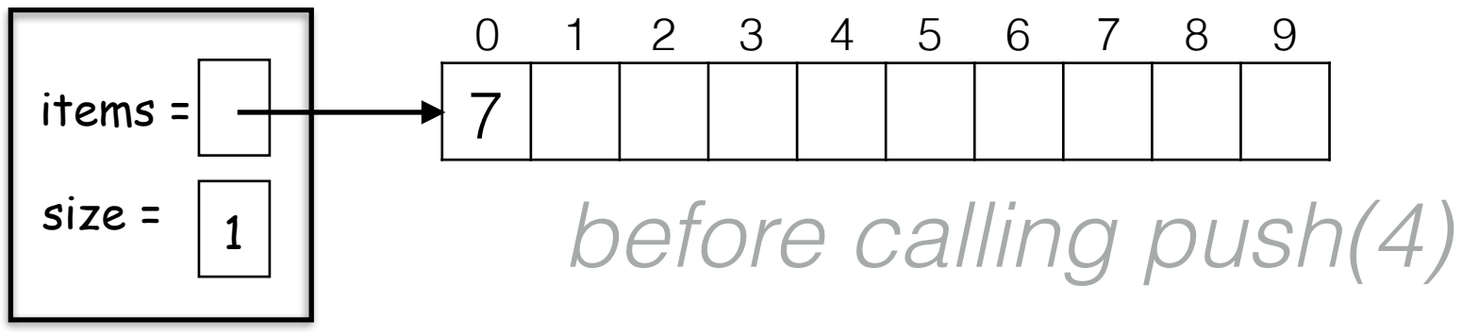
after calling push(7)

Push

Push an object to the top of stack

- Note that in array implementation top is on the back of array.

```
// add items
public void push(Object obj) { ... }
```



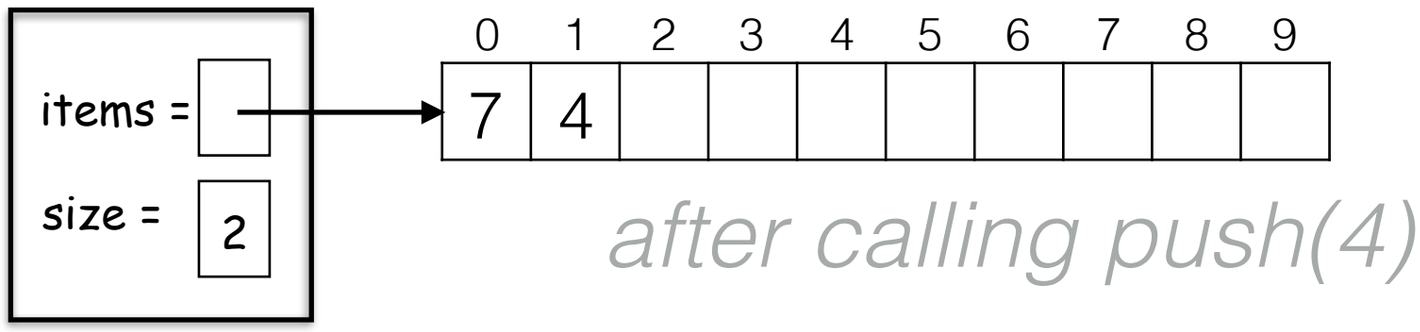
before calling push(4)

Push

Push an object to the top of stack

- Note that in array implementation top is on the back of array.

```
// add items
public void push(Object obj) { ... }
```



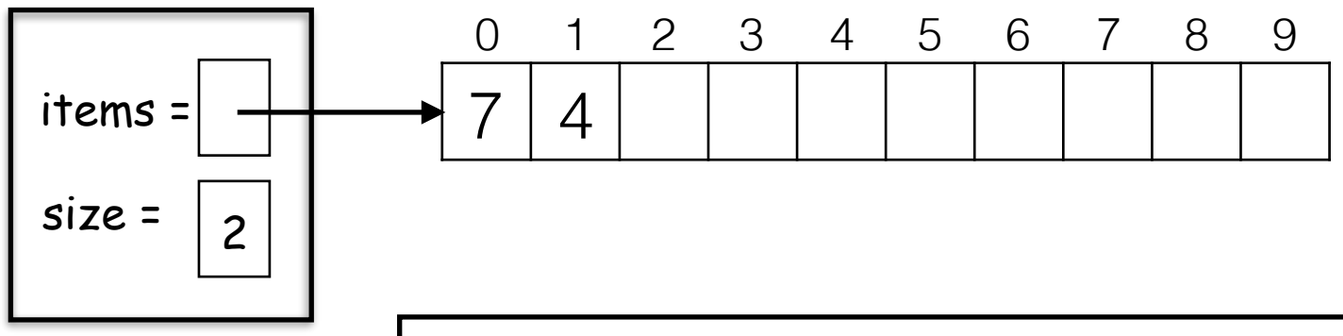
after calling push(4)

Push

Push an object to the top of stack

- Note that in array implementation top is on the back of array.

```
// add items
public void push(Object obj) { ... }
```




```
public void push(Object obj) {
    //if (items.length == size) expandArray();
    items[size] = obj;
    size++;
}
```

Push

Push an object to the top of stack

- Note that in array implementation top is on the back of array.

```
// add items
public void push(Object obj) { ... }
```

before calling push(6)

```
public void push(Object obj) {
    //if (items.length == size) expandArray();
    items[size] = obj;
    size++;
}
```

Push

Push an object to the top of stack

- Note that in array implementation top is on the back of array.

two invariants about size:

1. **always shows the number of items in stack**
2. **also refers to the index above the top of stack**

```
// add items
public void push(Object obj) { ... }
```

before calling push(6)

We do not need to use another variable for top!

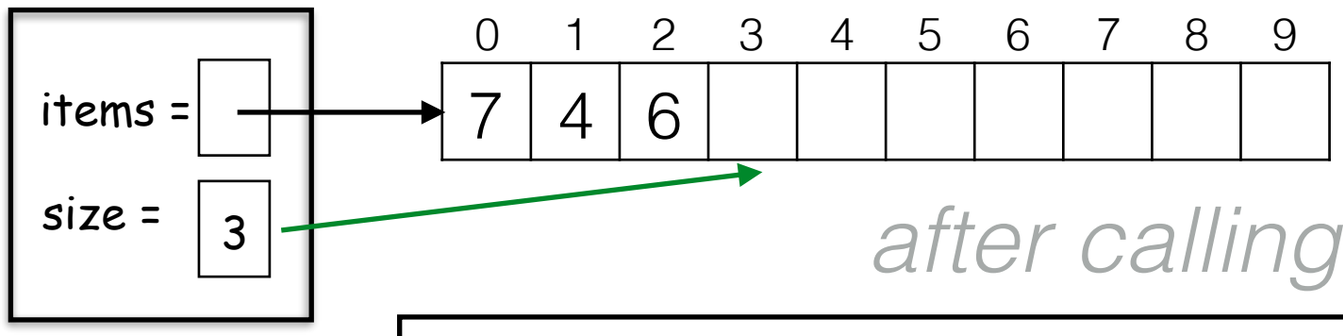
```
public void push(Object obj) {
    //if (items.length == size) expandArray();
    items[size] = obj;
    size++;
}
```

Push

Push an object to the top of stack

- Note that in array implementation top is on the back of array.

```
// add items
public void push(Object obj) { ... }
```



after calling push(6)

top of stack is index 3

```
public void push(Object obj) {
    //if (items.length == size) expandArray();
    items[size] = obj;
    size++;
}
```

Top

If there are n objects in the stack, the last is located at index $n - 1$

Top

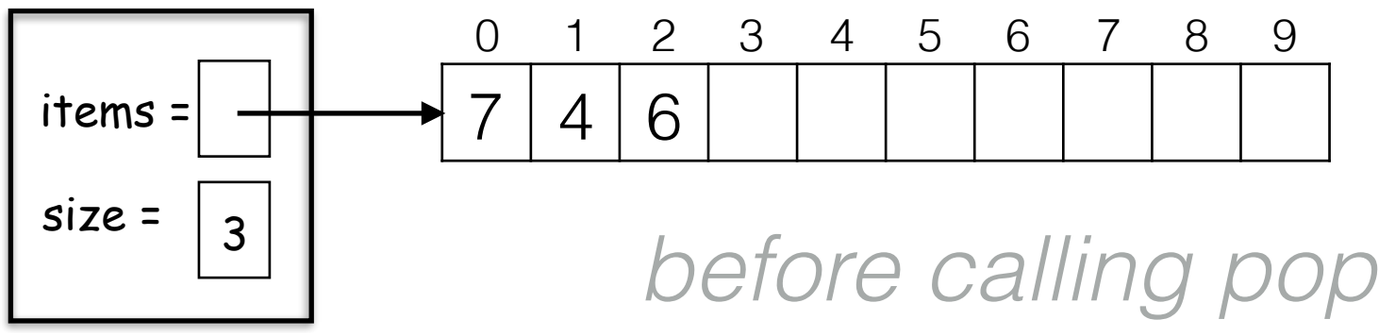
If there are n objects in the stack, the last is located at index $n - 1$

```
public Object top() throws EmptyStackException {
   if (size == 0) throw new EmptyStackException();
   return items[size-1];
}
```

Pop

The pop method needs to remove the top-of-stack item, and return it, as illustrated below.

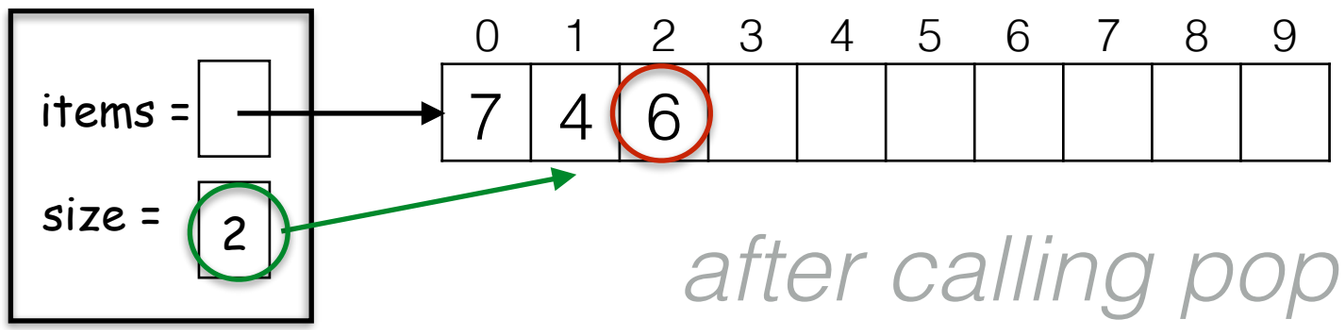
```
  // remove the item on top
public Object pop(Object obj) { ... }
```



before calling pop

Pop

Note that, in the picture, the value "6" is still in `items[2]`; however, that value is no longer in the stack because `size` is 2, which means that `items[1]` is the last item in the stack (or top)

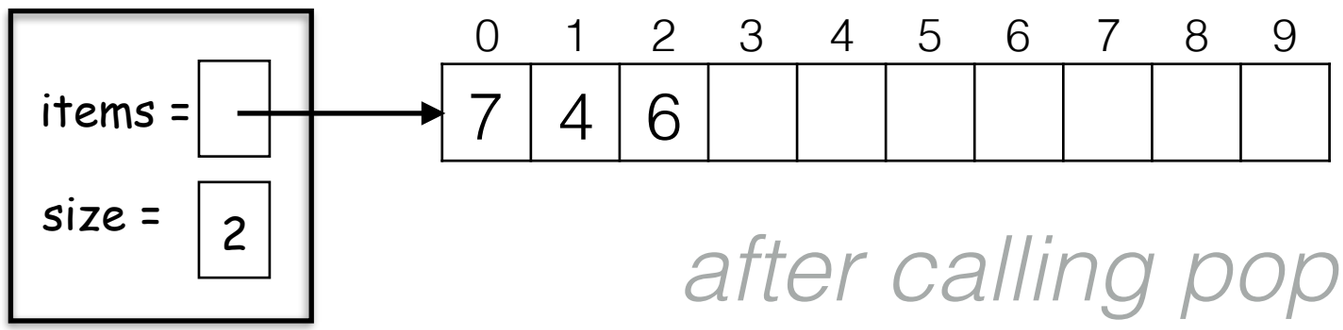


after calling pop

`pop()`
should return 6

Pop

The pop method needs to remove the top-of-stack item, and return it, as illustrated below.



after calling pop

```
public Object pop() throws EmptyStackException {
    if (size==0) throw new EmptyStackException();
    size--;
    return items[size];
}
```

Empty

The stack is empty if the stack size is zero:

```
public boolean isEmpty() {
    return size == 0;
}
```

The following is unnecessarily tedious:

- The == operator evaluates to either **`true`** or **`false`**

```
if ( size == 0 ) {
    return true;
} else {
    return false;
}
```

Exceptions

The case where the array is full is not an exception defined in the Abstract Stack

If the array is filled, we have five options:

- Increase the size of the array
- Throw an exception
- Ignore the element being pushed
- Replace the current top of the stack
- . . .

Array Capacity

If dynamic memory is available, the best option is to increase the array capacity (`INITSIZE` in Stack class)

- Add a method to class `Stack`, call it `expandArray()`
- Hint: Look at implementation of lists via array

If we increase the array capacity, the question is:

- How much?
- By a constant? `INITSIZE += c;`
- By a multiple? `INITSIZE *= c;`

- There is a huge discussion on array resizing, if you are interested.
- but here just double the size of array.

Summary

- We have discussed stack and two approaches for implementation:
  - Linked List
  - Array

Array		Link List	
pop	$\Theta(1)$	**pop**	$\Theta(1)$
push	$\Theta(1)$	**push**	$\Theta(1)$
top	$\Theta(1)$	**top**	$\Theta(1)$

- Very efficient data structure for some applications

Application: Parsing

Most parsing uses stacks

Examples includes:

- Matching tags in XHTML
- In C++, matching
  - parentheses `( ... )`
  - brackets, and `[ ... ]`
  - braces `{ ... }`

Parsing XHTML

The first example will demonstrate parsing XHTML

We will show how stacks may be used to parse an XHTML document

You will use XHTML (and more generally XML and other markup languages) in the workplace

Parsing XHTML

A ***markup language*** is a means of annotating a document to given context to the text

- The annotations give information about the structure or presentation of the text

The best known example is HTML, or HyperText Markup Language

- We will look at XHTML

Parsing XHTML

XHTML is made of nested

- *opening tags*, e.g., `<some_identifier>`, and
- matching *closing tags*, e.g., `</some_identifier>`

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the <i>browser</i>.</p></body>
</html>
```

Parsing XHTML

Nesting indicates that any closing tag must match the most recent opening tag

Strategy for parsing XHTML:

- read though the XHTML linearly
- place the opening tags in a stack
- when a closing tag is encountered, check that it matches what is on top of the stack and

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>			

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<head>		

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<head>	<title>	

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<head>	<title>	

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<head>		

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>		

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	<i>

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	<i>

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>	<p>	

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>	<body>		

Parsing XHTML

```
<html>
  <head><title>Hello</title></head>
  <body><p>This appears in the
  <i>browser</i>.</p></body>
</html>
```

<html>			

Parsing XHTML

We are finished with parsing, and the stack is empty

Possible errors:

- a closing tag which does not match the opening tag on top of the stack
- a closing tag when the stack is empty
- the stack is not empty at the end of the document

HTML

Old HTML required neither closing tags nor nesting

```
<html>
  <head><title>Hello</title></head>
  <body><p>This is a list of topics:
  <ol>                              <!-- para ends with start of list -->
    <li><i>veni                        <!-- implied </li> -->
    <li>vidi                        <!-- italics continues -->
    <li>vici</i>
</ol>                        <!-- end-of-file implies </body></html> -->
```

Parsers were therefore specific to HTML

– Results: ambiguities and inconsistencies

XML

XHTML is an implementation of XML

XML defines a class of general-purpose ***eXtensible Markup Languages*** designed for sharing information between systems

The same rules apply for any flavour of XML:

- opening and closing tags must match and be nested

Reading

- Chapter 16