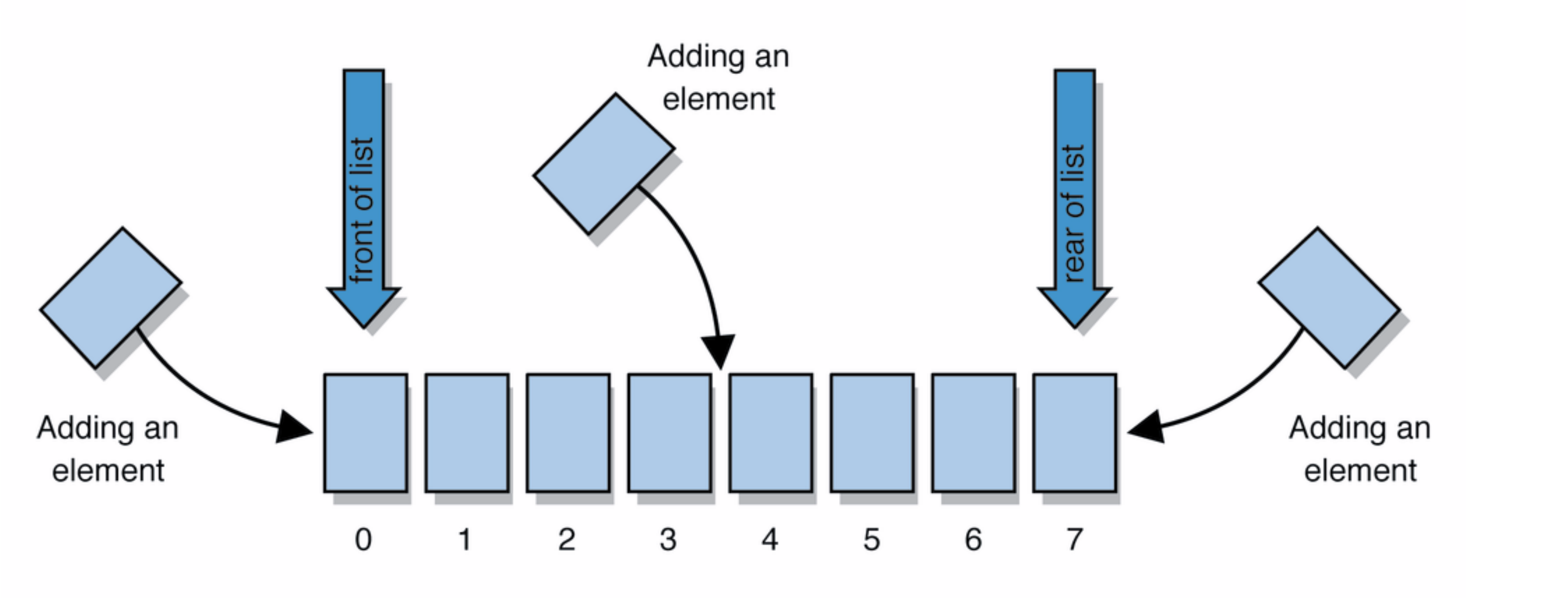
COMP251: DATA STRUCTURES & ALGORITHMS

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

Stack

List (Recap)

- A collection storing an ordered sequence of elements
  - Each element is accessible by a 0-based **index**
  - A list has a **size** (number of elements that have been added)
  - Elements can be added to the front, back, or elsewhere



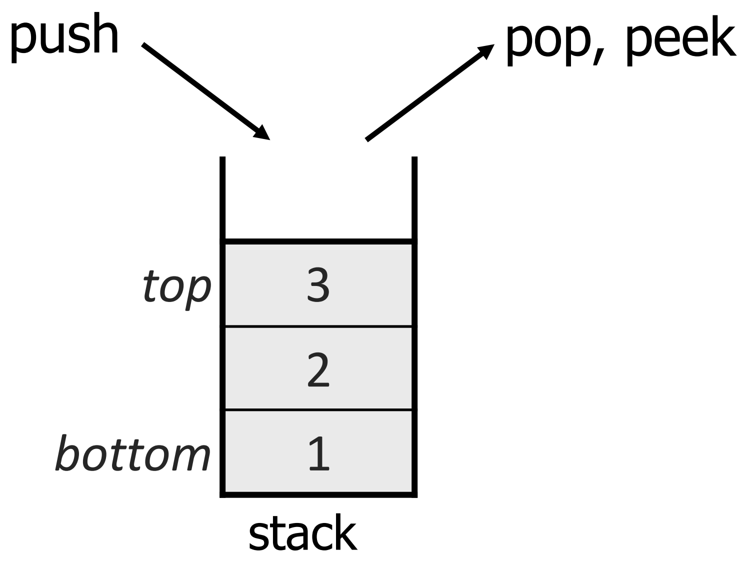
ADT List operations (Recap)

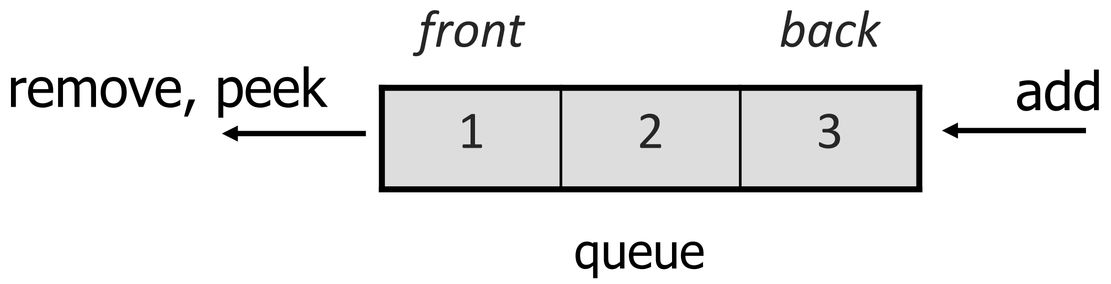
add(**value**)	appends value at end of list
add(**index**, **value**)	inserts given value just before the given index, shifting subsequent values to the right
clear()	removes all elements of the list
indexOf(**value**)	returns first index where given value is found in list (-1 if not found)
get(**index**)	returns the value at given index
remove(**index**)	removes/returns value at given index, shifting subsequent values to the left
set(**index**, **value**)	replaces value at given index with given value
size()	returns the number of elements in list
contains(**value**)	returns true if given value is found somewhere in this list

Stack and Queue

- Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.

- Two specialty collections:
  - **stack:** Retrieves elements in the reverse of the order they were added.
  - **queue:** Retrieves elements in the same order they were added.

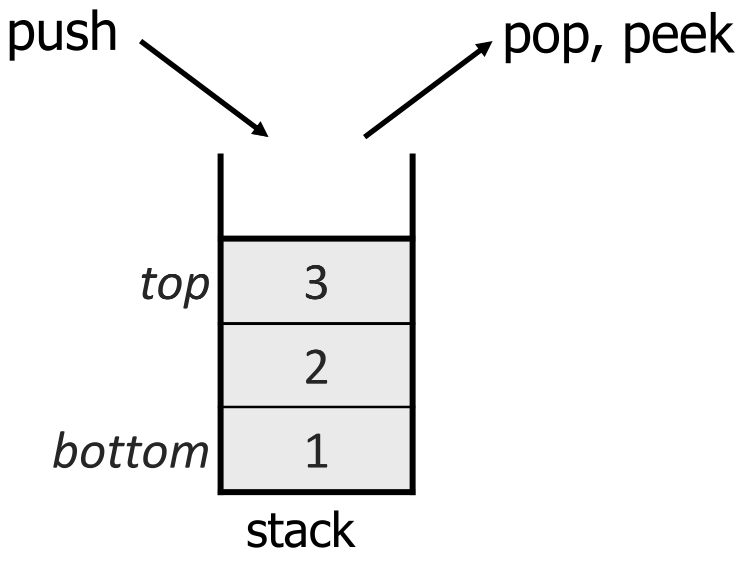




ADT Stack

- Insertion and deletion only examine the last element added
  - we call it **top** of stack
  - the other end is called the bottom
- Access to other items is not allowed
- A LIFO (Last In First Out) collection or data structure

`push(`**value**`)`	places given value on top of stack
`pop()`	removes top value from stack and returns it
`peek()`	returns top value from stack without removing it
`size()`	returns number of elements in stack
`isEmpty()`	returns `true` if stack has no elements

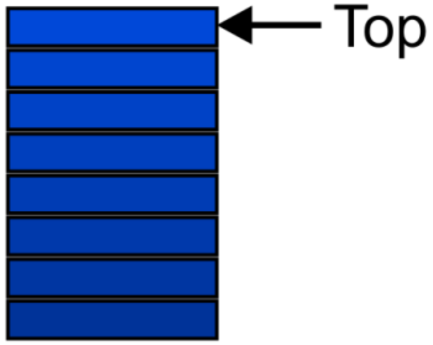


ADT Stack

last-in–first-out (LIFO) list

Graphically, we may view these operations as follows:

- `peek():` returns the last item pushed to the stack without removing it. It is also called `top()` in some text books.

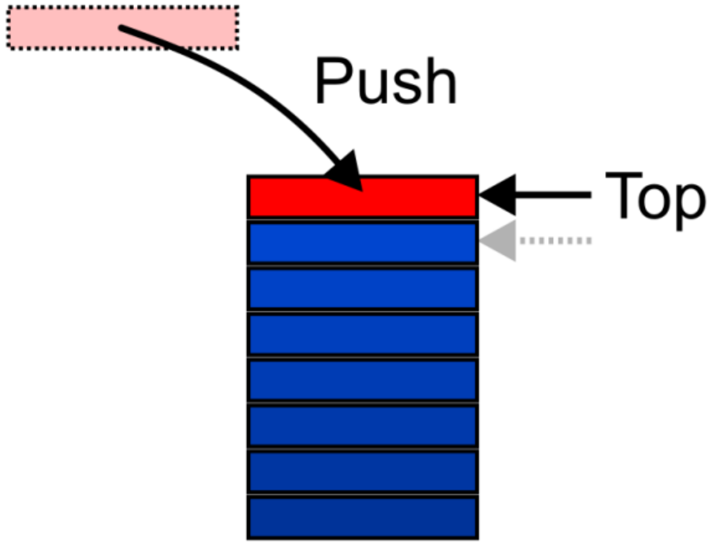


ADT Stack

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.

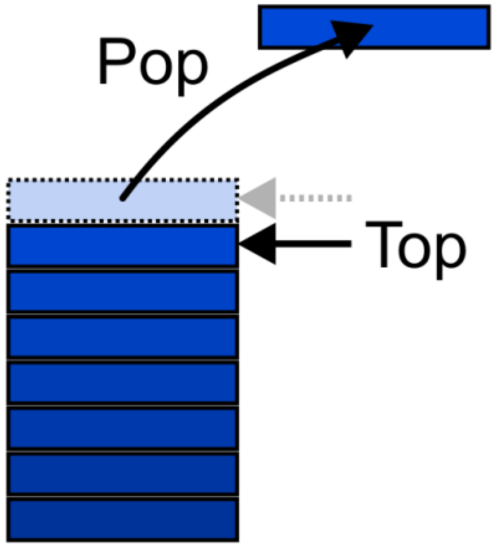


ADT Stack

last-in–first-out (LIFO) list

Graphically, we may view these operations as follows:

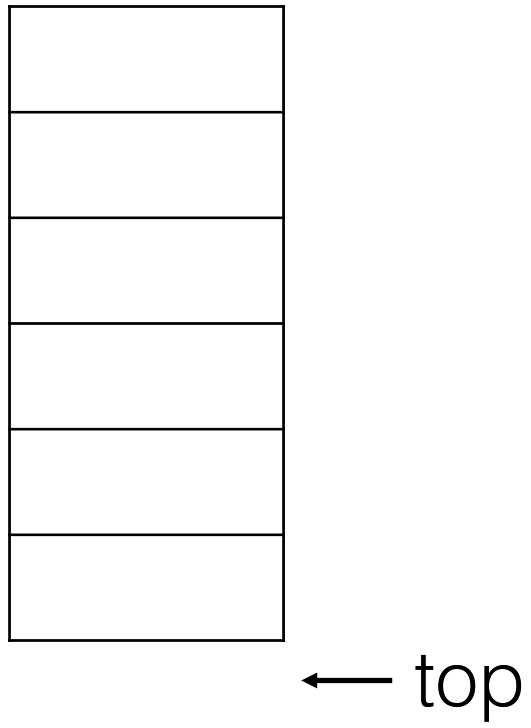
–`pop()`: remove the last item pushed (inserted) to the stack and return it.



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

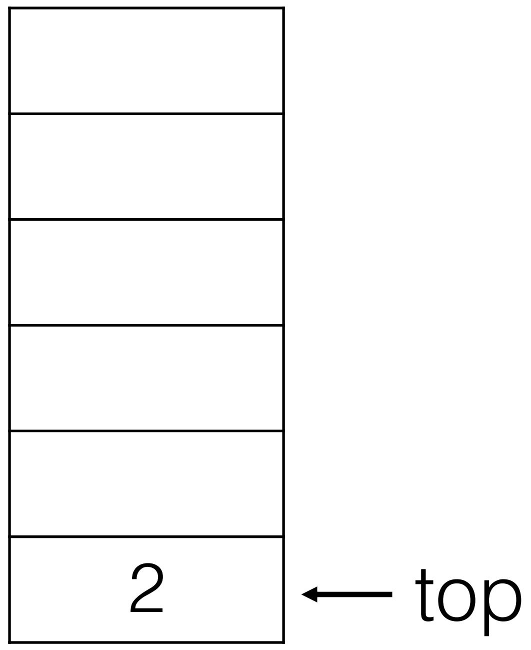
Example

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



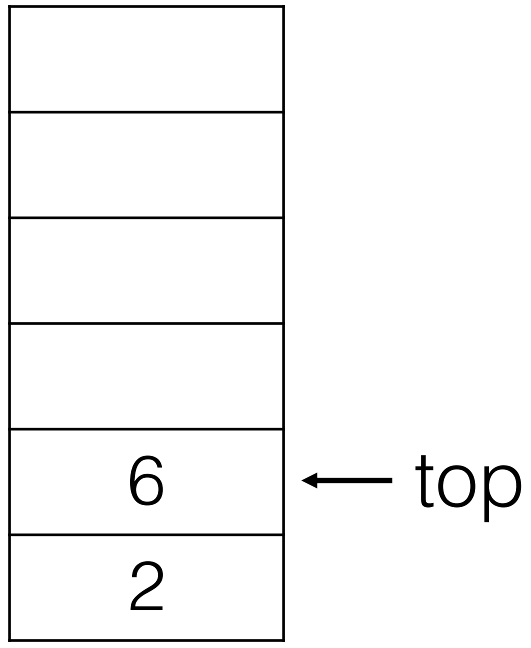
Example

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



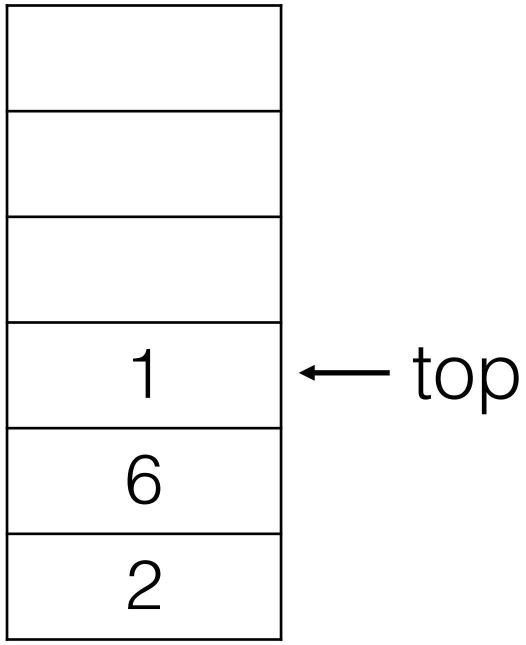
Example

- 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)



1

6

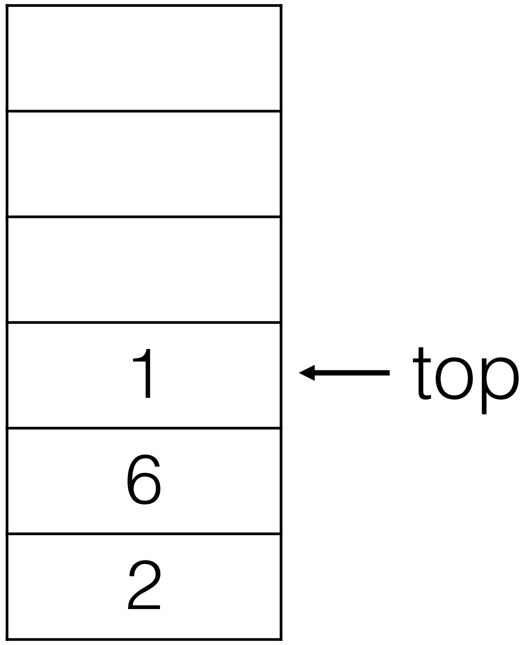
2

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)
  - peek()   //just return value 1



1 ⟵ top

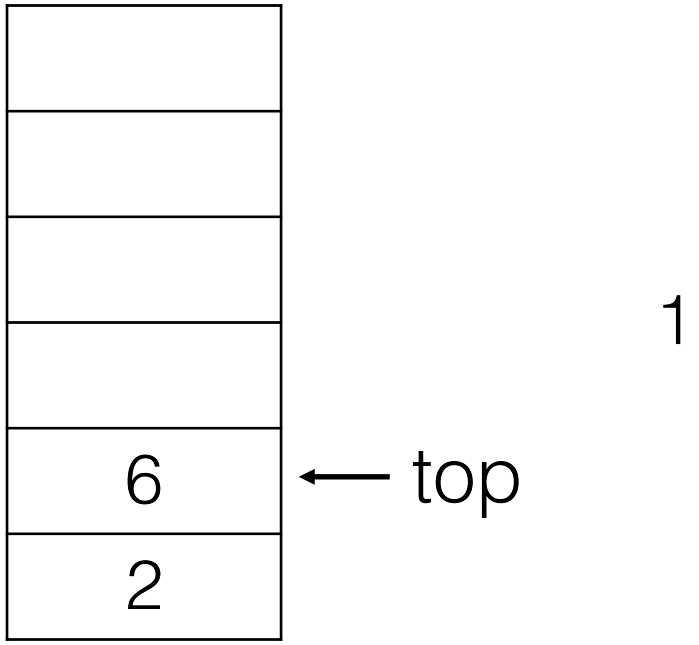
6

2

Example

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

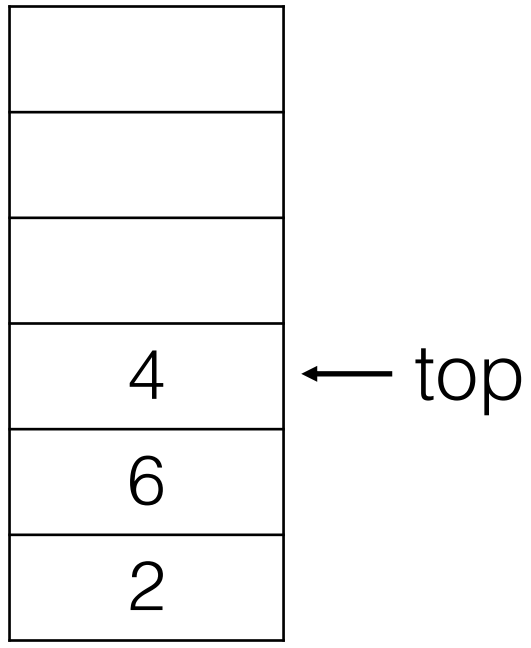
  - push(2)
  - push(6)
  - push(1)
  - peek()
  - pop()   //remove 1 and return it



Example

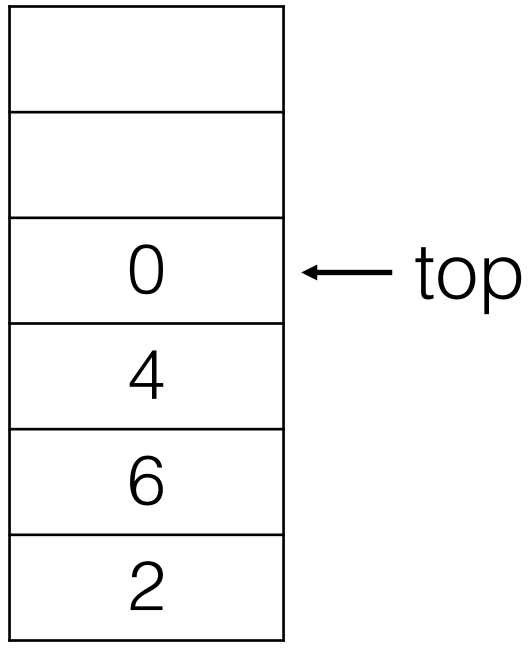
- Assume we have a stack for integer values. The stack is empty at the beginning. Then we apply a set of operations.

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

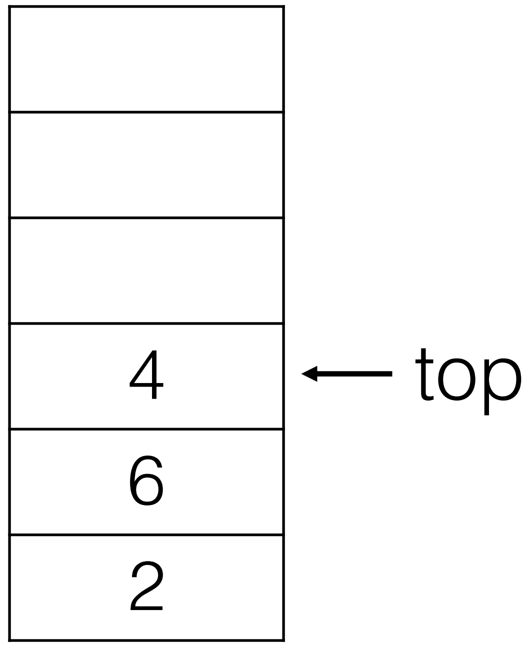


0
4
6
2

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)
  - push(1)
  - peek()
  - pop()
  - push(4)
  - push(0)
  - pop()



4 ← top

6

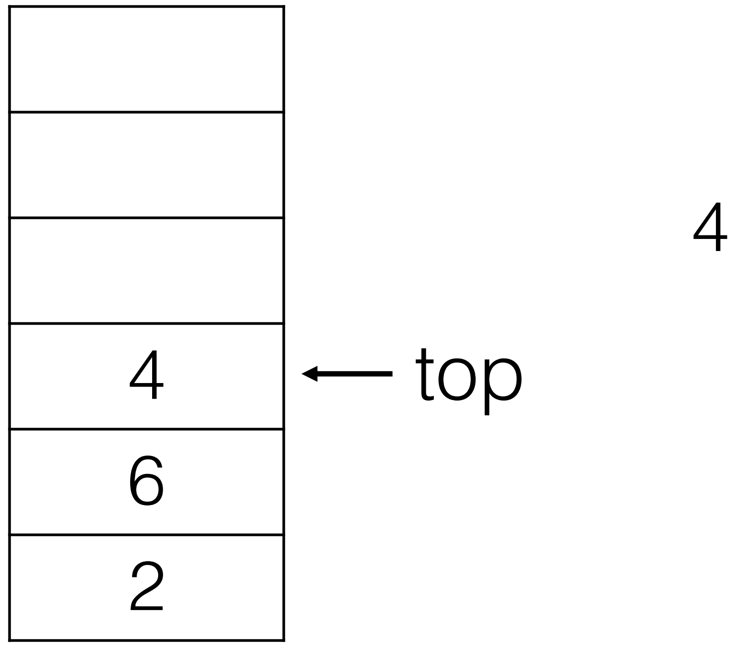
2

0

Example

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

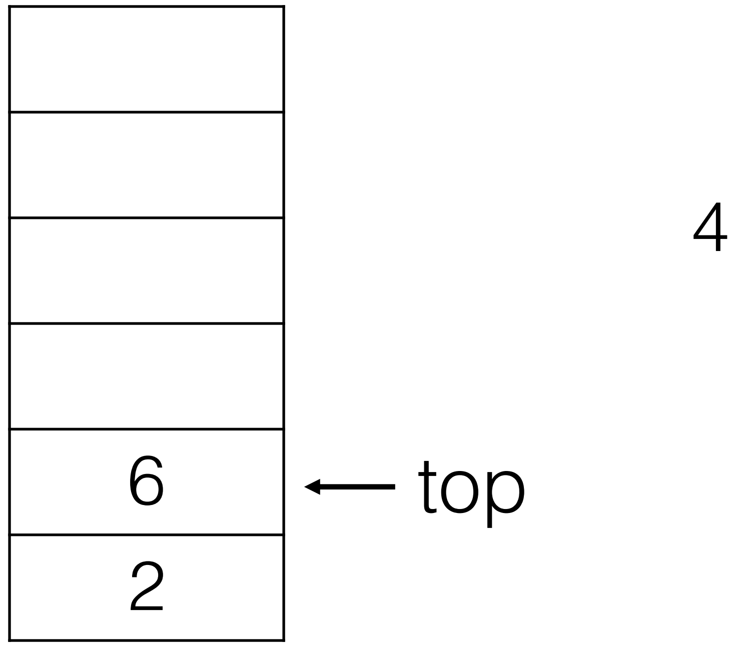
  - push(2)
  - push(6)
  - push(1)
  - peek()
  - pop()
  - push(4)
  - push(0)
  - pop()
  - peek()



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)
  - peek()
  - pop()
  - push(4)
  - push(0)
  - pop()
  - peek()
  - pop()



Implementations

We will look at two implementations of stacks:

- Array based
- Linked lists

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

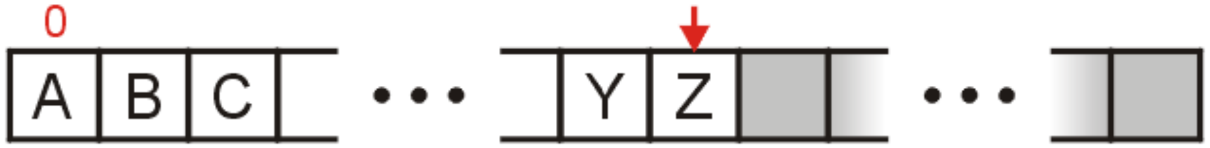
- The run time of the algorithm is independent of the number of objects being stored in the container
- We will always attempt to achieve this lower bound

Stack: Array Implementation

- If an array is used to implement a stack, we could choose:
  - always add items at the beginning of the array
  - always add items at the end of the array
- Note that we are trying to optimize push and pop as stacks are usually assumed to be extremely fast
- What is a good index for the top item?
  - Consider top to be at position 0?
  - Consider top to be at position n-1?

Stack: Array Implementation

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



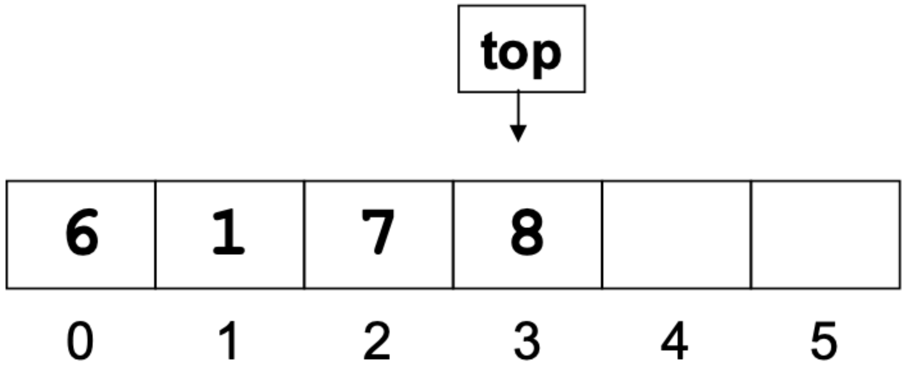
	Front/1st	End/nth
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(n)$	$\Theta(1)$
Erase	$\Theta(n)$	$\Theta(1)$

Stack: Array Implementation

- If an array is used to implement a stack, we could choose:
    - always add items at the beginning of the array
    - always add items at the end of the array
- Note that we are trying to optimize push and pop as stacks are usually assumed to be extremely fast
- What is a good index for the top item?
    - Consider top to be at position 0? $\Theta(n)$ to insert or remove
    - Consider top to be at position n-1? $\Theta(1)$ to insert or remove

Stack: Array Implementation

the index of top is: (the number of items in stack) - 1



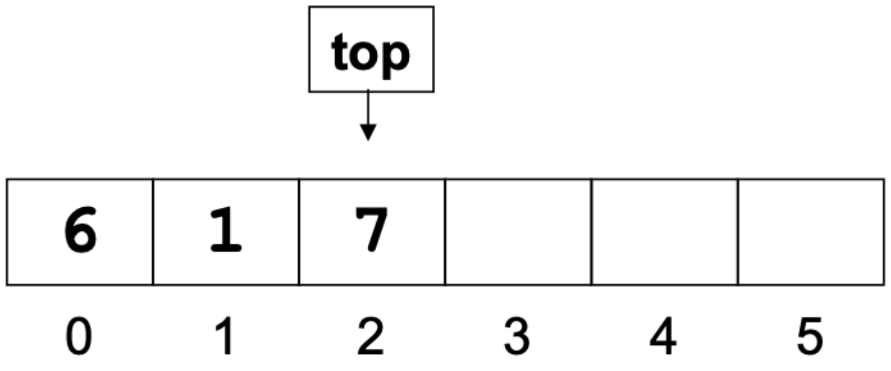
index of **top** is

<u>current size – 1</u>

```
//Java Code
Stack st = new Stack();
st.push(6); //top = 0
st.push(1); //top = 1
st.push(7); //top = 2
st.push(8); //top = 3
```

Stack: Array Implementation

the index of top is: (the number of items in stack) - 1



top

6	**1**	**7**			
0	1	2	3	4	5

index of **top** is

current size – 1

```
//Java Code
Stack st = new Stack();
st.push(6); //top = 0
st.push(1); //top = 1
st.push(7); //top = 2
st.push(8); //top = 3
st.pop(); //top = 2
```

Stack: Array Implementation

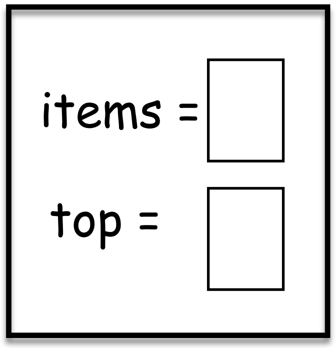
We need to store an array and a field to keep the index of top:

```
public class ArrayBasedStack<AnyType> {
    private static int MAX_SIZE = 1000;  // initial array size
    private AnyType[] items;            // array to keep items in the stack
    private int top;                   // indicates the index of top in top stack
                                       // which is equal to the number of items in the stack-1

    public ArrayBasedStack(){
        items = (AnyType[]) (new Object[MAX_SIZE]);
        top = -1;
    }
    public int size() { return top+1; }
    public boolean isEmpty() {  return top < 0; }
    private void expand() { [. . .]}
    public void push(AnyType x) {
        if (size() == MAX_SIZE) expand();
        items[++top] = x;
    }
    public AnyType peek() throws EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException();
        return items[top];
    }
    public AnyType pop() throws EmptyStackException {
        if (isEmpty())
            throw new EmptyStackException();
        return items[top--];
    }
```

Constructor

The class is only storing the address of the array

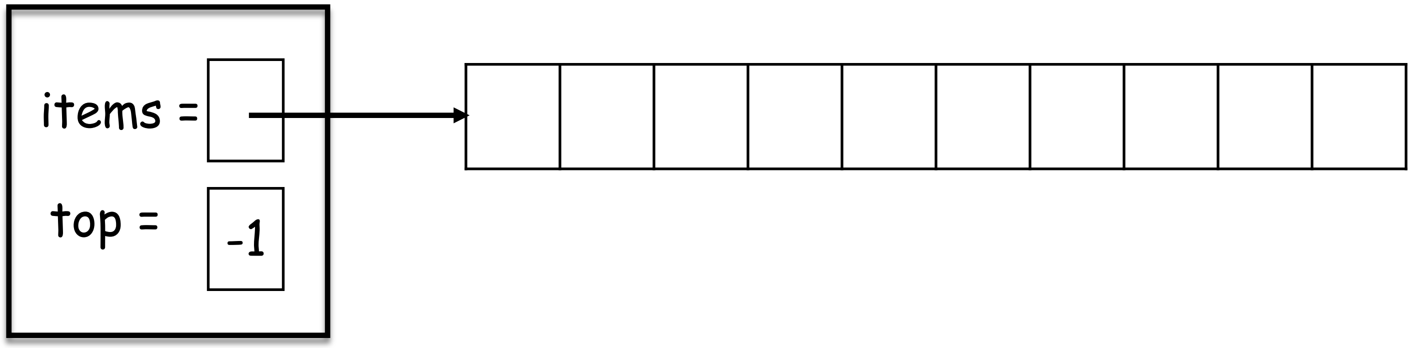
- We must allocate memory for the array and initialize the index top



Constructor

The class is only storing the address of the array

- We must allocate memory for the array and initialize the index top



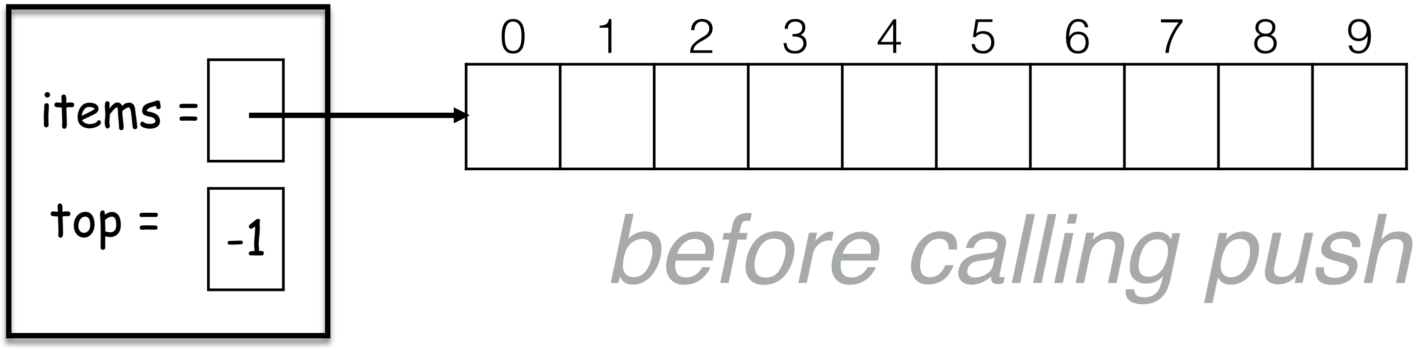
```
public ArrayBasedStack() {
   items = (AnyType[]) new Object[MAX_SIZE];
   top = -1;
}
```

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(AnyType obj) { ... }
```



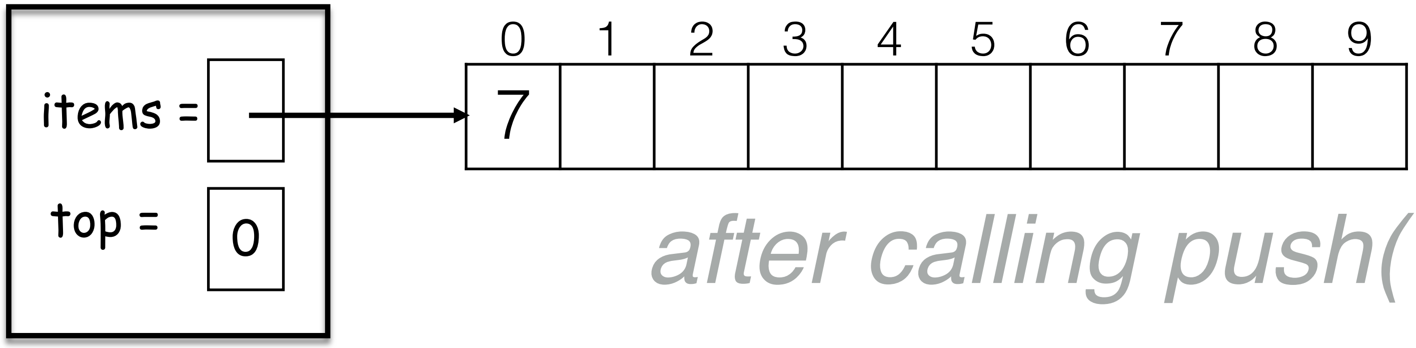
before calling push(7)

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(AnyType obj) { ... }
```



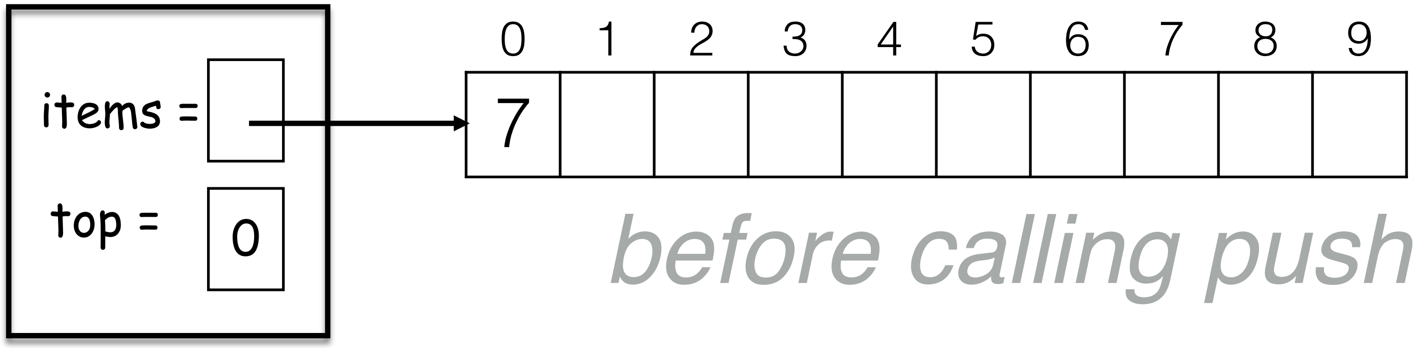
after calling push(7)

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(AnyType obj) { ... }
```



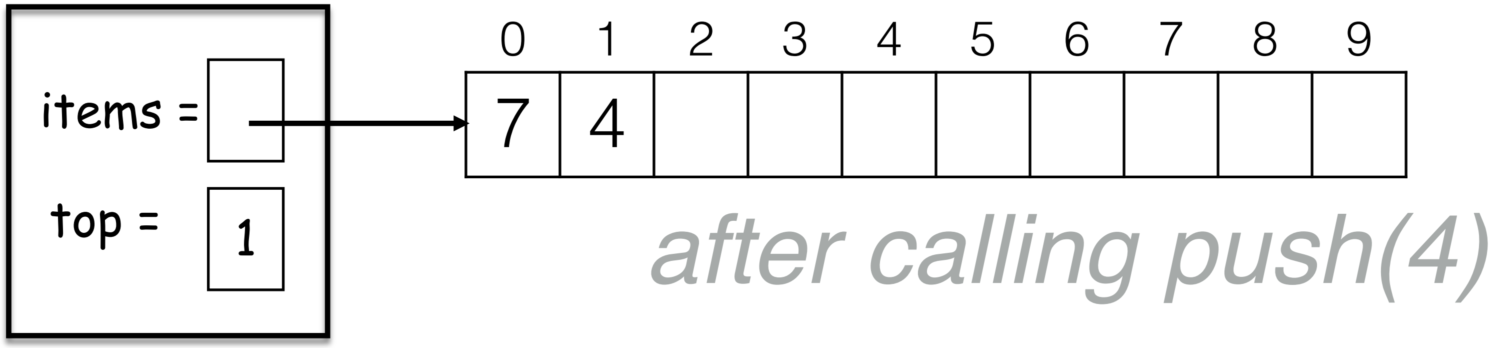
before calling push(4)

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(AnyType obj) { ... }
```



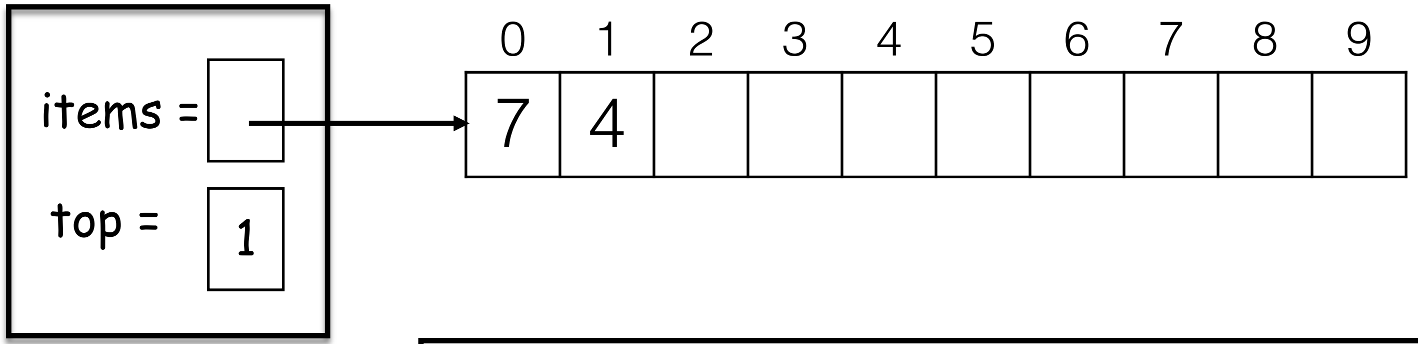
after calling push(4)

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(AnyType obj) { ... }
```



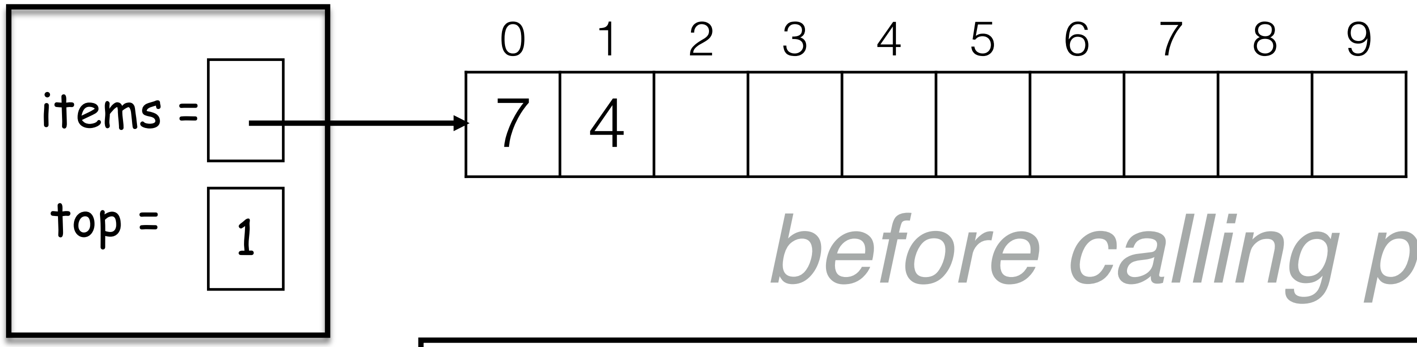
```
public void push(AnyType obj) {
    //if (size() == MAX_SIZE) expand();
    items[++top] = obj;
}
```

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(AnyType obj) { ... }
```



before calling push(6)

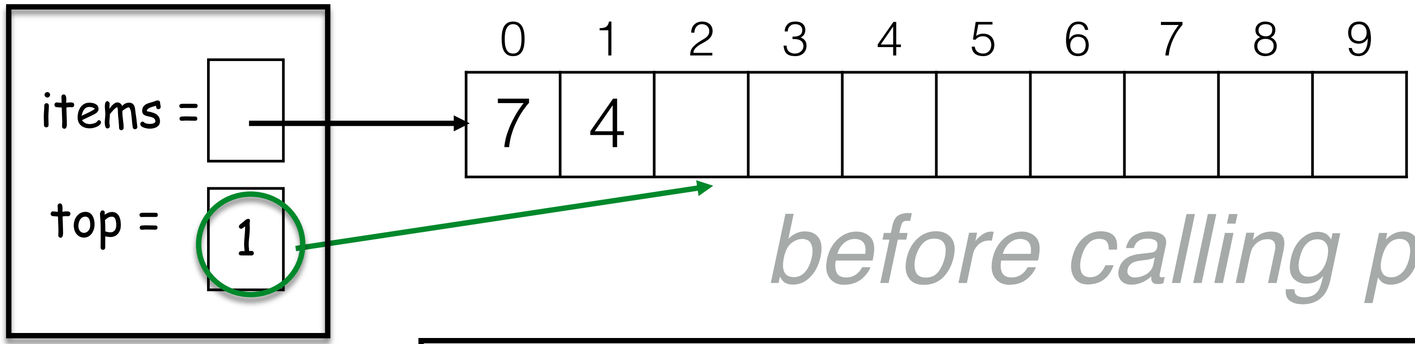
```
public void push(AnyType obj) {
    //if (size() == MAX_SIZE) expand();
    items[++top] = obj;
}
```

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(AnyType obj) { ... }
```



before calling push(6)

```
public void push(AnyType obj) {
    //if (size() == MAX_SIZE) expand();
    items[++top] = obj;
}
```

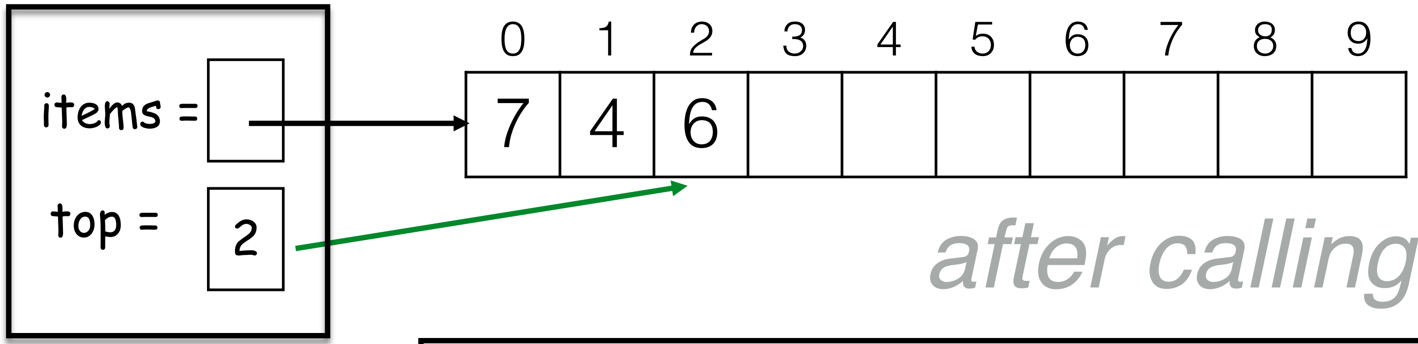
**Increase the top ,
then insert the item**

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(AnyType obj) { ... }
```



after calling push(6)

Now top of stack is index 2

```
public void push(AnyType obj) {
    //if (size() == MAX_SIZE) expand();
    items[++top] = obj;
}
```

peek

If there are n objects in the stack, the last is located at index $n - 1$ which is stored as `top`

peek

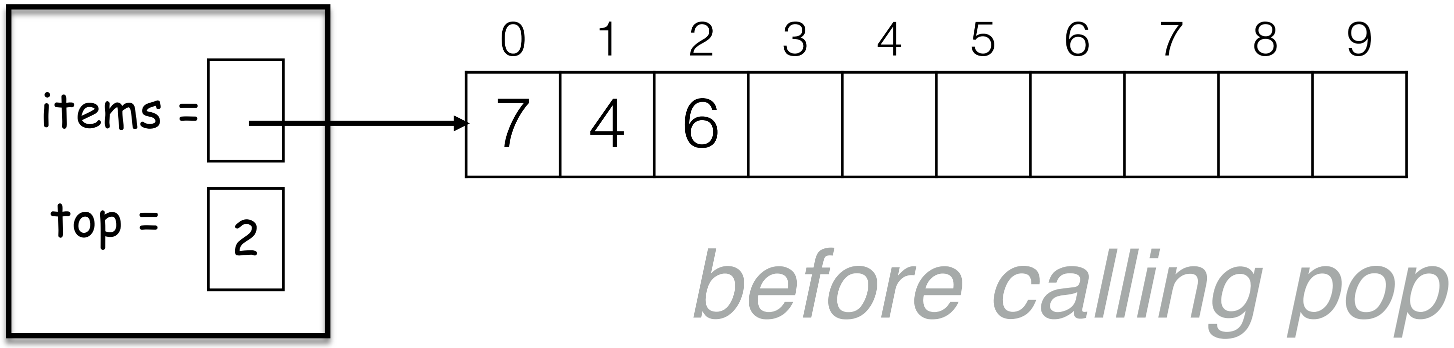
If there are n objects in the stack, the last is located at index $n - 1$ which is stored as `top`

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

Pop

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

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



items =

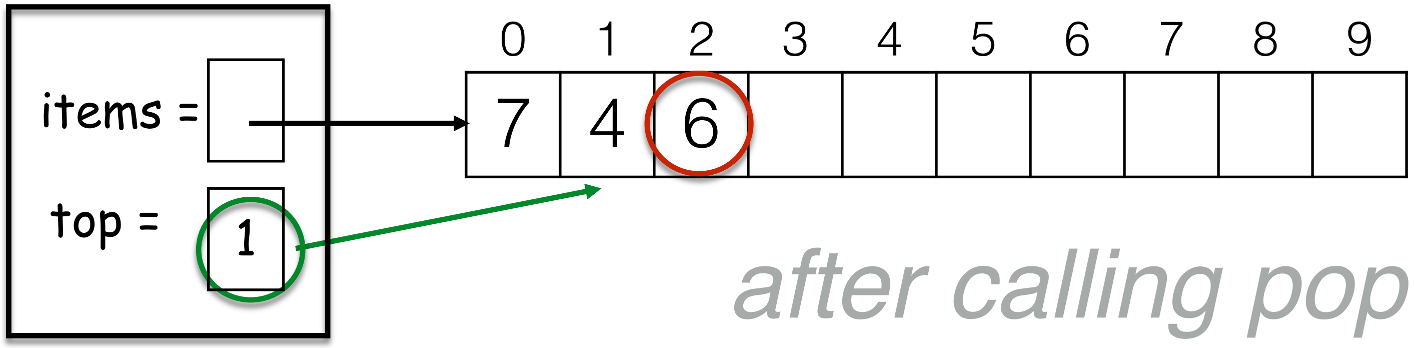
top = 2

0	1	2	3	4	5	6	7	8	9
7	4	6							

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 `top` is 1, which means that `items[1]` is the last item in the stack



`pop()`
should return 6

after calling pop

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

get the value at index top, then decrease top

isEmpty

The stack is empty if the stack size is zero:

```
public boolean isEmpty() {
    return top < 0;
}
```

The following is unnecessarily tedious:

- The `<` operator evaluates to either `true` or `false`

```
if ( top < 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 a few options:

- Increase the size of the array
- Throw an exception
- . . .

Array Capacity

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

- Add a method to class `ArrayBasedStack`, call it `expand()`
- Hint: similar to array implementation of List ADT

If we increase the array capacity, the question is:

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

- There is a huge discussion on array resizing, if you are interested.
- Here we doubled the size of array.

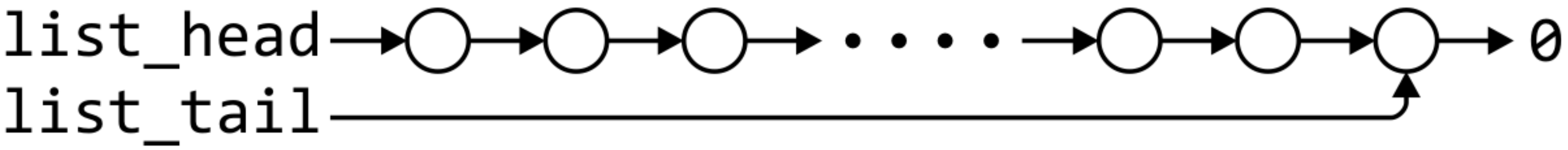
Stack: Array Implementation

- Easy to implement
- `push`, `pop` and `peek` can be implemented as $\Theta(1)$
- But the implementation is subject to the size of arrays
- If the maximum size of array is not known (or is much larger than expected) we need to use dynamic array
- Therefore occasionally push will take $\Theta(n)$

Stack:
Linked-List Implementation

Stack: 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 ADT Stack will be achieved by performing all operations at the front

Stack: Linked List Implementation

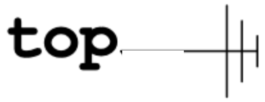
```
public class SListStack<AnyType> {
  private SListNode<AnyType> top;     // top of the stack.
  private int size;                   // Number of items in stack.

  public SListStack() {     // Here's how to create an empty stack.
      top = null;
      size = 0;
  }
  public int size() { return size; }
  public boolean isEmpty(){  return (size() == 0); }
  public void push(AnyType x) {
      top = new SListNode<AnyType>(x, top);
      size++;
  }
  public AnyType peek() throws EmptyStackException {
      if (isEmpty())
          throw new EmptyStackException();
      return top.item;
  }
  public AnyType pop() throws EmptyStackException {
    if (isEmpty())
        throw new EmptyStackException();
    AnyType item = top.item;
    top = top.next;
    size--;
    return item;
  }
```

push and pop at the head of list.

Stack: Linked List Implementation

push and pop at the front of linked list.

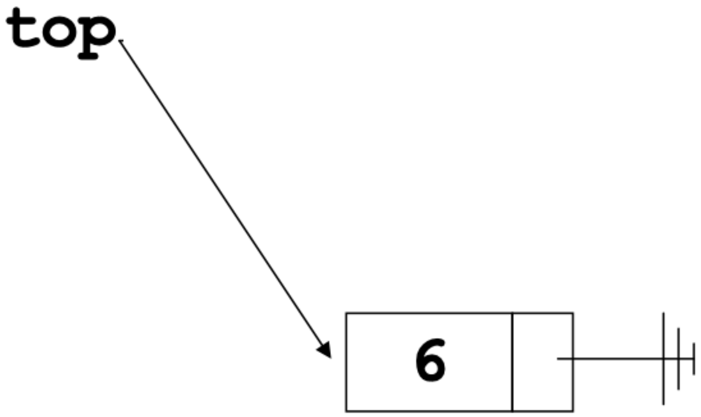
```
SListStack st = new SListStack();
```



Stack: Linked List Implementation

push and pop at the front of linked list.

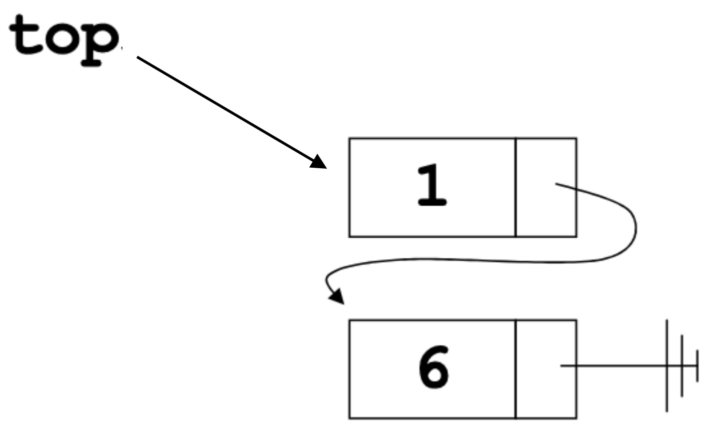
```
SListStack st = new SListStack();
st.push(6);
```



Stack: Linked List Implementation

push and pop at the front of linked list.

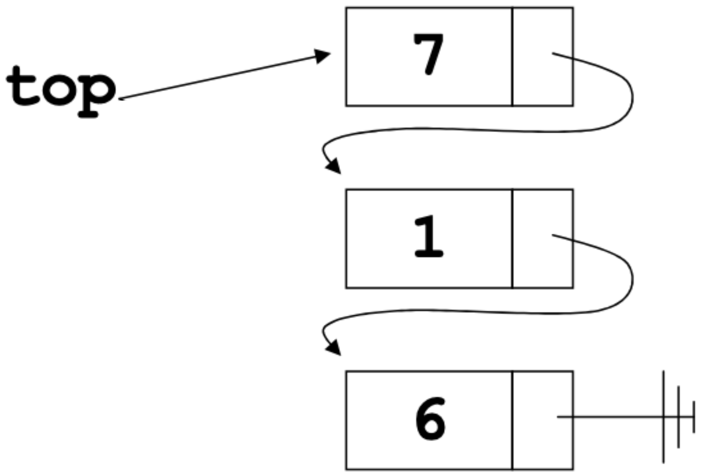
```
SListStack st = new SListStack();
st.push(6);
st.push(1);
```



Stack: Linked List Implementation

push and pop at the front of linked list.

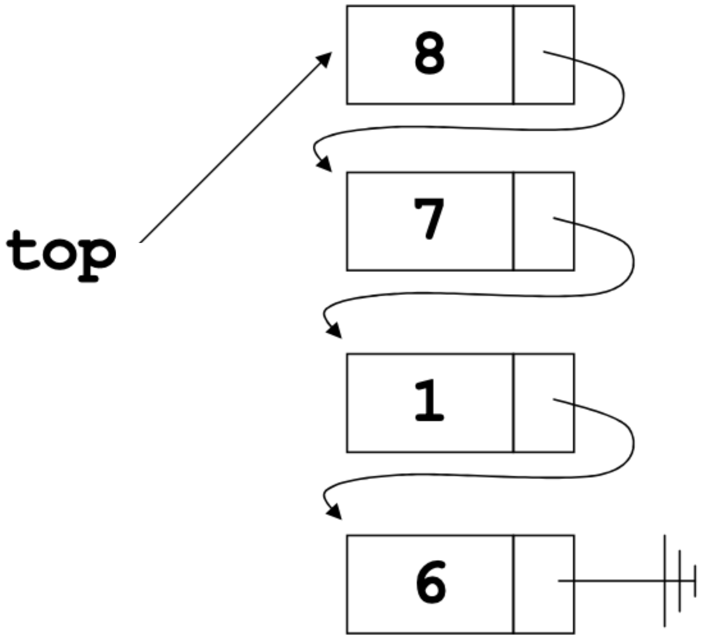
```
SListStack st = new SListStack();
st.push(6);
st.push(1);
st.push(7);
```



Stack: Linked List Implementation

push and pop at the front of linked list.

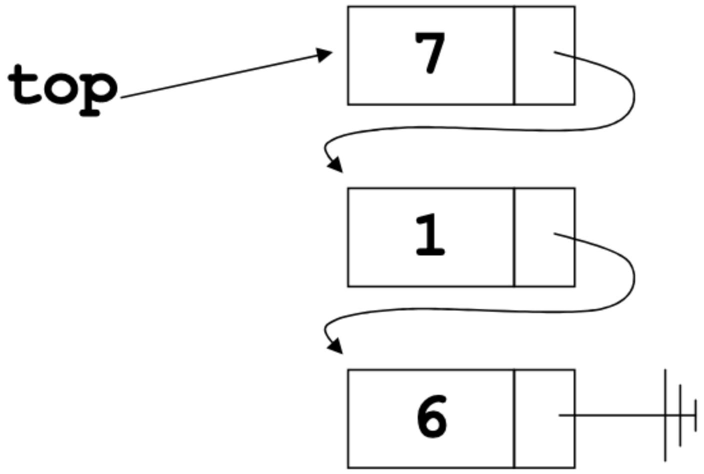
```
SListStack st = new SListStack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
```



Stack: Linked List Implementation

push and pop at the front of linked list.

```
SListStack st = new SListStack();
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```



Stack: ADT List Implementation

- Assume that we have ADT List and we want to implement stack using the ADT List we already have

```
public Stack() {
   list = new List();
}
```

- push() and pop() can be done at either the beginning or end of the ADT List
    - at the beginning

```
public AnyType pop() {
   return list.remove(0);
}
```

```
public void push(AnyType obj) {
   list.add(0, obj);
}
```

Stack: ADT List Implementation

- Assume that we have ADT List and we want to implement stack using the ADT List we already have

```
public Stack() {
   list = new List();
}
```

- push() and pop() can be done at either the beginning or end of the ADT List
  - at the end

```
public AnyType pop() {
   return list.remove(list.size()-1);
}
```

```
public void push(AnyType obj) {
   list.add(list.size(), obj);
}
```

Stack: ADT List Implementation

- Assume that we have ADT List and we want to implement stack using ADT List we already have
- push() and pop() can be done at either the beginning or end of the ADT List
- Efficiency depends on implementation of ADT List
    - You should know which implementation you are using
    - Another reason to have different implementations for each ADT!

Summary

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

Array		Linked 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

Stack in Java Collection API

push(**value**)	Pushes the value onto the top of this stack.
empty()	Tests if this stack is empty
search(**value**)	returns the 1-based position where the given value is found in stack
peek()	returns the object at the top of this stack without removing it from the stack.
pop()	removes/returns value at the top of this stack
size()*	returns the number of elements in list
clear()*	removes all elements of the stack
toString()*	returns a string representation of the list such as "[3, 42, -7, 15]"

* these methods are inherited from Vector collection
Stack collection in Java API is an extension of Vector
Check the Java API tutorials for the full list of methods for vector:
https://docs.oracle.com/javase/10/docs/api/java/util/Vector.html

Check the Java API tutorials for the more details about Stack in JAVA API:
https://docs.oracle.com/javase/10/docs/api/java/util/Stack.html

Applications

Numerous applications:

- Parsing code:
  - Matching parenthesis
  - Checking balanced expressions
  - Evaluating arithmetic expressions
  - 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

Application: Parsing

Most parsing algorithms uses stacks

Examples includes:

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

Checking Balanced Strings

A stack can be used to check whether a string (e.g. a program) is balanced in terms of parentheses and braces,…

An example of balanced string:

```
abc{ d efg {} ijk l {m {n}} op} qr
```

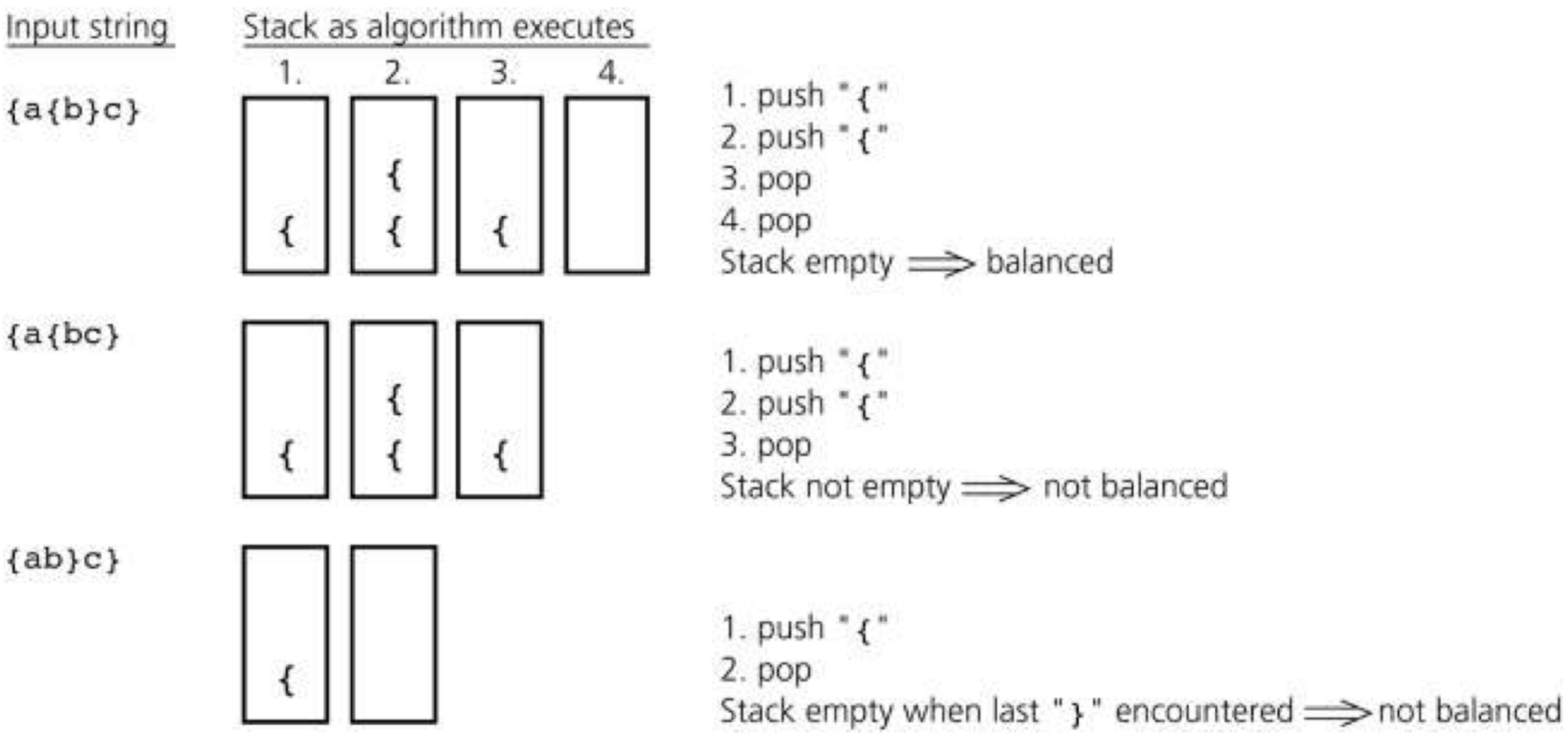
Examples of unbalanced strings:

```
abc{defg }} { ijk l {m {n}} op qr
abc{defg {} ijk l {m {n}} op}{ qr
```

Checking Balanced Strings

- Requirements for balanced strings:
    - Each time you encounter a close braces, “}” it matches an already open braces “{”
    - When you reach the end of string, you have matched all braces
- If you process the text from left to right, each time you see a close braces, “}”, it must be matched to the last seen, unmatched open braces “{”
- You can see that stack is a perfect ADT to solve this problem

Checking Balanced Strings



{a{b}c}

1. push "{"
2. push "{"
3. pop
4. pop

Stack empty ⟹ balanced

{a{bc}

1. push "{"
2. push "{"
3. pop

Stack not empty ⟹ not balanced

{ab}c}

1. push "{"
2. pop

Stack empty when last "}" encountered ⟹ not balanced

simply ignore the other characters

Checking Balanced Strings

```
public static boolean isBalanced(String s) {
    Stack<Character> stack = new Stack<Character>();

    for (int i = 0; i < s.length(); i++) {

        if (s.charAt(i) == '{')
            stack.push('{');

        else if (s.charAt(i) == '}') {
            if (stack.isEmpty())        return false;
            if (stack.pop() != '{')     return false;
        }
        // ignore all other characters
    }
    return stack.isEmpty();
}
```

using Java Stack API

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 (if not, there is an error)

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