Stacks II (Chapter 5)

Eleni Mangina

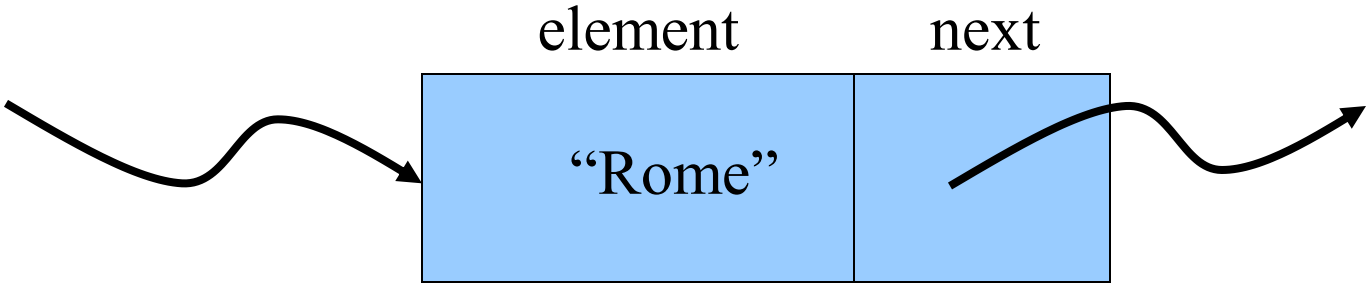
Room B2.05
School of Computer Science and Informatics
University College Dublin, Ireland

Array-Based Stacks

- Array-based implementations => Finite Capacity.
  - Memory (Mis-)management issues.
  - Application Logic issues.
- Possible Solution: Use an “extendable array”
  - Creating a new larger array and copying the existing values into it.
  - Running time – O(n) => Pop / Push running times become O(n)
- Are there any other implementation strategies?
- YES! We can use the **Linked List** data structure.

Linked Lists

- In a Linked List, the objects are stored in **nodes**.
  - Each node also maintains a reference to the next node in the list (this is the **link**).
  - The link of the last node in the list is set to null.



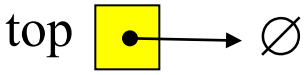
Null means “end of list”

- We also store references to “key” nodes / entry points.
  - These provide us with a way of accessing the list

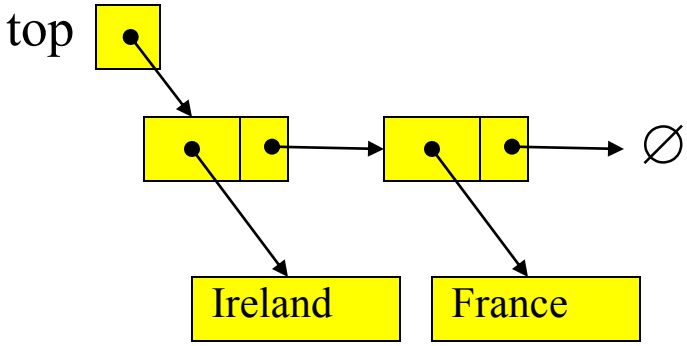
Link-Based Stack

- Auxiliary Data Structure: **Node**
  - A reference to the object being stored in the stack
  - A link to the next node in the stack (this is the link).

- Key Nodes / Data:
  - The “top” node of the stack
  - The link of the bottom node in a non-empty stack is set to null.
  - Need to keep track of the “size” of the stack



Empty Stack



Stack of size 2

Class Node

```
public class Node<E> {

    // Instance variables:
    private E element;
    private Node<E> next;

    /** Creates a node with null references to its element and next node. */
    public Node() { this(null, null); }

    /** Creates a node with the give n element and next node. */
    public Node(E e, Node<E> n) { element = e; next = n; }

    // Accessor methods:
    public E getElement() { return element; }
    public Node<E> getNext() { return next; }

    // Modifier methods:
    public void setElement(E newElem) { element = newElem; }
    public void setNext(Node<E> newNext) { next = newNext; } }
```

Generic NodeStack Class

```
public class NodeStack<E> implements Stack<E> {

        protected Node<E> top; // reference to the head node

        protected int size; // number of elements in the stack

        public NodeStack() { // constructs an empty stack
                top = null; size = 0; }

        public int size() { return size; }

        public boolean isEmpty() { if (top == null) return true; return false; }

        public void push(E elem) { Node<E> v = new Node<E>(elem, top);
        // create and link-in a new node
        top = v; size++; }

        public E top() throws EmptyStackException {
                if (isEmpty()) throw new EmptyStackException("Stack is empty.");
                return top.getElement(); }

        public E pop() throws EmptyStackException {
                 if (isEmpty()) throw new EmptyStackException("Stack is empty.");
                E temp = top.getElement();
                top = top.getNext(); // link-out the former top node
                size--;
                return temp; } }
```

Link-Based Stack

Algorithm push(o):

Input: an object o

Output: none

node ← new Node(o)

node.next ← top

top ← node

size ← size + 1

Algorithm size():

Input: none

Output: count of objects on the stack

return size

Algorithm isEmpty():

Input: none

Output: true if the stack is empty, false otherwise

return size = 0

Algorithm pop():

Input: none

Output: the top object

e ← top.element

top.element ← null

top ← top.next

size ← size-1

return e

Algorithm top():

Input: none

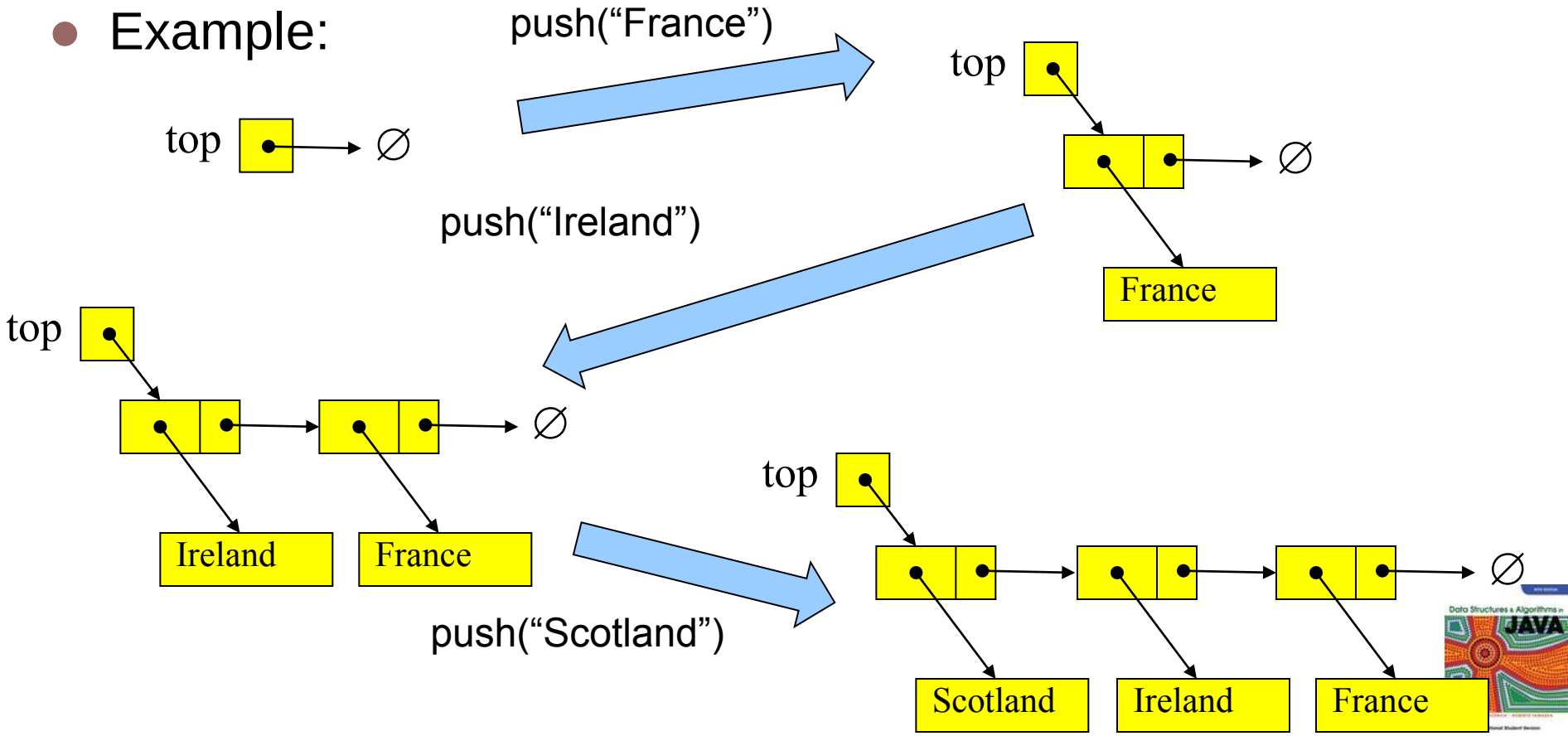
Output: the top object

return top.element

Link-Based Stacks: Dry Runs

- View operations as atomic
  - Show the state of the Linked List after each operation

- Example:



Link-Based Stacks: Impl.

- Class name: `LinkedStack`
  - Fields:
    - An inner class `Node` (see Worksheet)
    - An integer, size (number of objects in the stack)
    - A `Node` field, top (references top node in stack)
  - Constructors
    - Default Constructor (sets top to `null` and `size` to `0`)
  - Methods:
    - 1 per operation: methods names should match operation names
    - Implement methods based on pseudo code

- This is part of your next worksheet

Interlude: Inner Classes

- A class that is declared as part of another class.
  - The inner class is in effect part of implementation of the outer class.
- Why use one?
  - Logically groups code that is only used in one place
  - Increases readability and maintainability
- Example:

```
public class LinkedStack {
     public class Node {
          …
     }
     …
}
```

Link-Based Stacks: Analysis

- Operation Running Times:

Operation	Running Time	ABS Running Time
push(o)	O(1)	O(1)
pop()	O(1)	O(1)
top()	O(1)	O(1)
isEmpty()	O(1)	O(1)
size()	O(1)	O(1)

- Issues:
  - What happens if we pop from an empty stack?

- Which Implementation Strategy is better?

Reversing an Array Using a Stack

```
/** A nonrecursive generic method for
    reversing an array */
    public static <E> void reverse(E[] a) {
           Stack<E> S = new
    ArrayStack<E>(a.length);
    for (int i=0; i < a.length; i++)
          S.push(a[i]);
          for (int i=0; i < a.length; i++)
          a[i] = S.pop(); }
```

```
/** Tester routine for reversing arrays */
public static void main(String args[]) {
    Integer[] a = {4, 8, 15, 16, 23, 42};
     // autoboxing allows this
    String[] s = {"Jack", "Kate", "Hurley",
    "Jin", "Boone"};
    System.out.println("a = " +
    Arrays.toString(a));
    System.out.println("s = " +
    Arrays.toString(s));
    System.out.println("Reversing...");
    reverse(a); reverse(s);
    System.out.println("a = " +
    Arrays.toString(a));
    System.out.println("s = " +
    Arrays.toString(s)); }
```

Exercise:

Describe how to implement two stacks using one array. The total number of elements in both stacks is limited by the array length; all stack operations should run in O(1) time.

Solution:

Let us make the stacks (S_1 and S_2) grow from the beginning and the end of the array (A) in opposite directions. Let the indices T_1 and T_2 represent the tops of S_1 and S_2 correspondingly. S_1 occupies places $A[0...T_1]$, while S_2 occupies places $A[T_2...(n-1)]$. The size of S_1 is T_1+1; the size of S2 is $n-T_2+1$. Stack S_1 grows right while stack S_2 grows left. Then we can perform all the stack operations in constant time similarly to how it is done in the basic array implementation of stack except for some modifications to account for the fact that the second stack is growing in a different direction. Also to check whether anyone of these stacks is full, we check if $S_1.size() + S_2.size() \geq n$. In other words the stacks do not overlap if their total length does not exceed n.