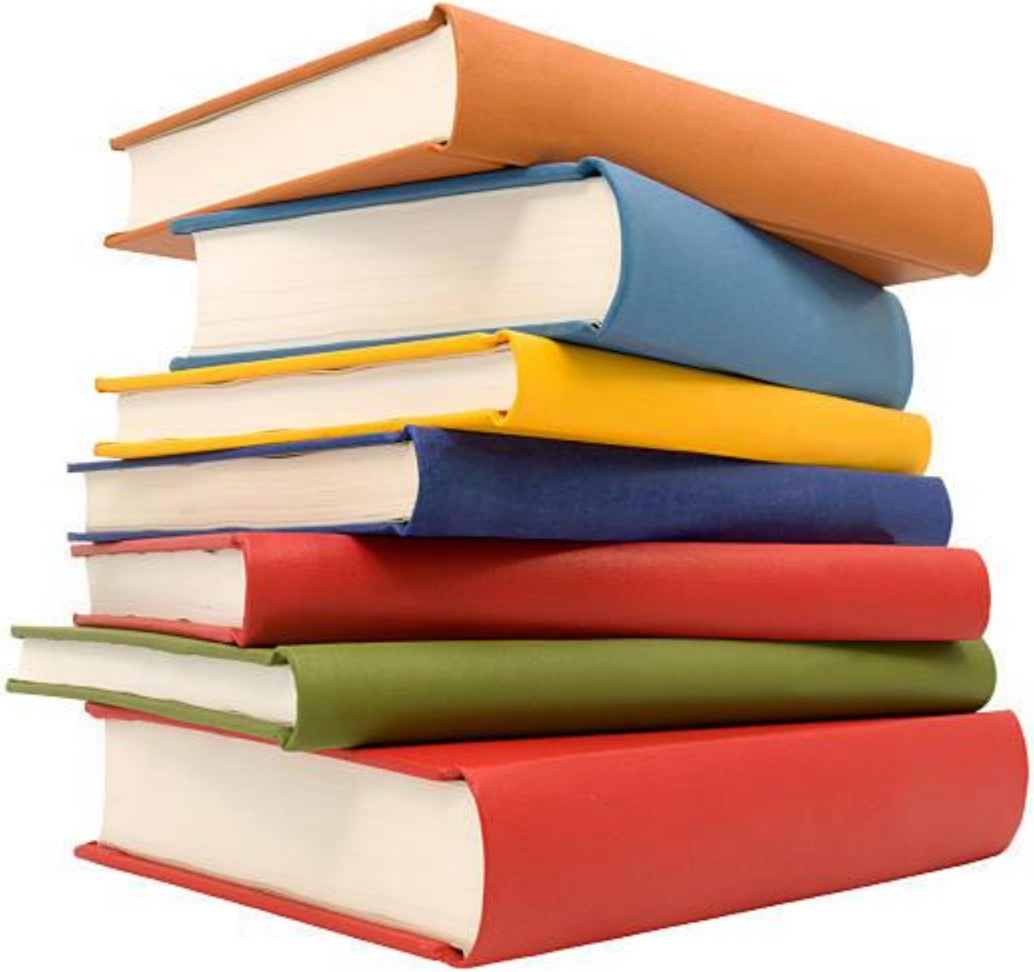
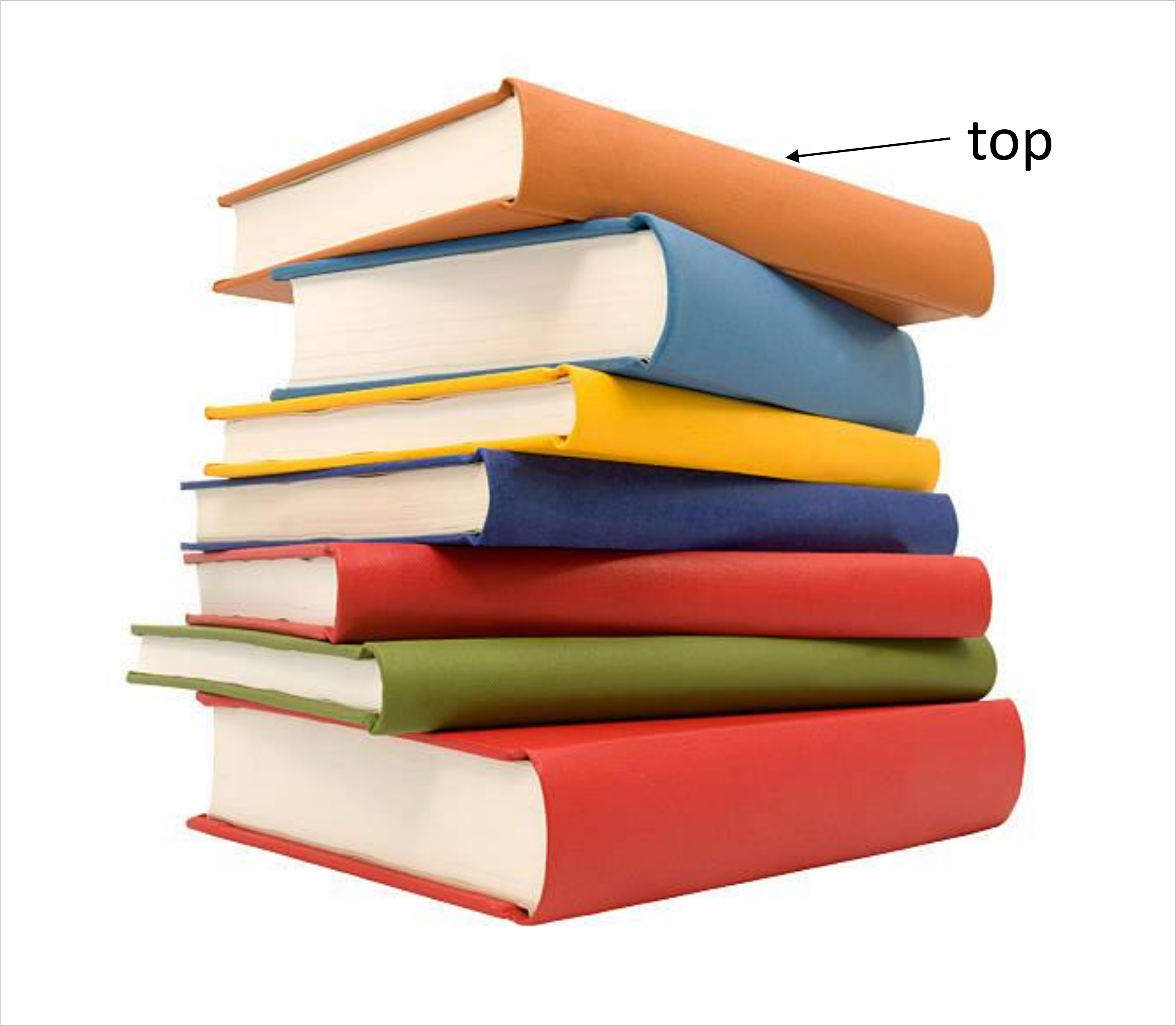
Stack

A special kind of List



What is a Stack?

- The Stack is a List where all insertions, removals, and retrievals take place at one end called the top.
- The Stack behaves in a Last-In, First-Out (LIFO) manner.



Primary methods (IStack)

```
// Place an item on the top of a stack (Insert)
public void Push(T item)

// Remove the top item of a stack (Remove)
public void Pop()

// Return the top item of a stack (Retrieve)
public T Top()
```

Unlike List, position is NOT passed as a parameter

Supporting methods (IContainer)

```
// Resets the stack to empty
public void MakeEmpty()

// Returns true if the stack is empty; false otherwise
public bool Empty()

// Returns the number of items in the stack
public int Size()
```

Generic Stack class

```
class Stack<T> : IStack<T>, IContainer<T>
{
    ...
}
```

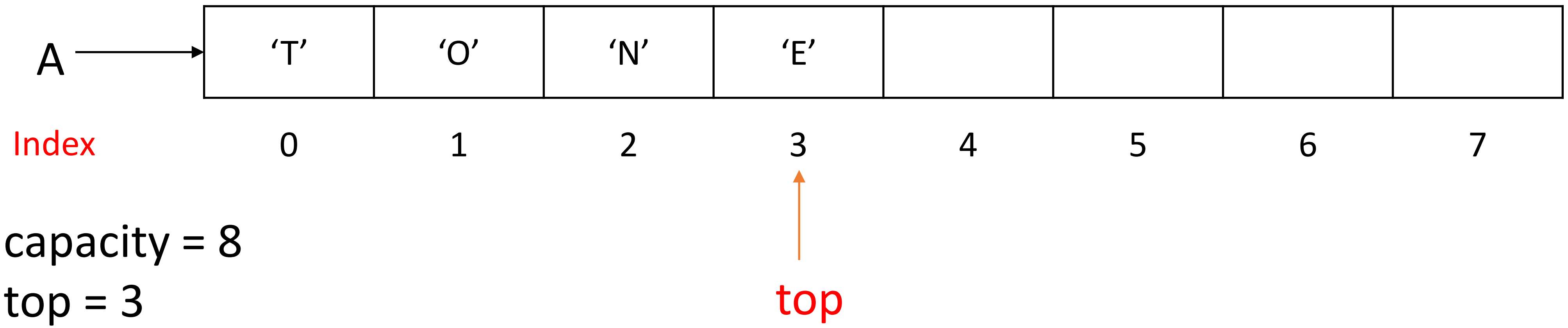
Note: In the code, IStack inherits IContainer.

Data structures

- Linear array ←
- Singly linked list

Linear array

```
private int capacity;  // Maximum capacity of the stack
private int top;       // Index of the top item in the stack
private T[] A;         // Linear array of items (Generic)
```

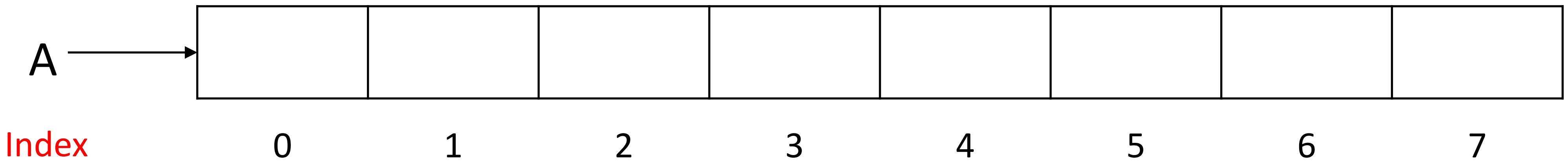


capacity = 8

top = 3

Constructor

- Basic Strategy
  - Create an array with a capacity of 8 (by default) and set top to -1.
  - A = new T [8] ;



capacity = 8

top = -1

Push an item onto the top of the Stack

- Basic Strategy
  - If stack is full (top+1 == capacity) then double the capacity of the stack (later).
  - Increase top by 1 and place the item at A[top].

```
public void Push(T item)
{
    if (top + 1 == capacity)
    {
        DoubleCapacity();
    }
    A[++top] = item;
}
```

Push ‘R’ onto the Stack

Before

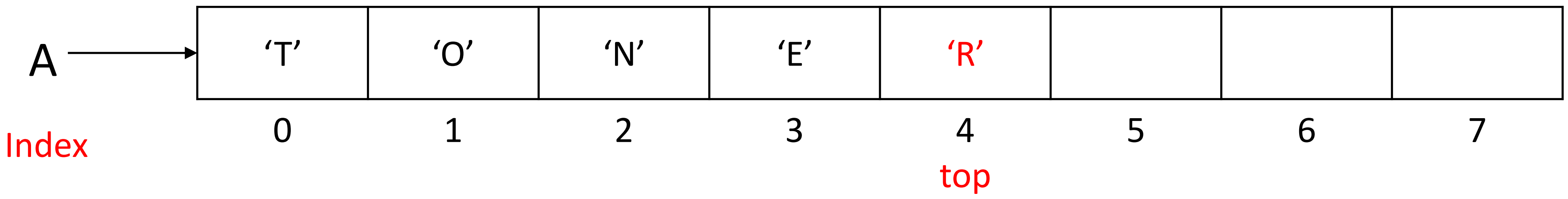
A →	‘T’	‘O’	‘N’	‘E’				
Index	0	1	2	3	4	5	6	7
				top				

capacity = 8

top = 3

Push ‘R’ onto the Stack

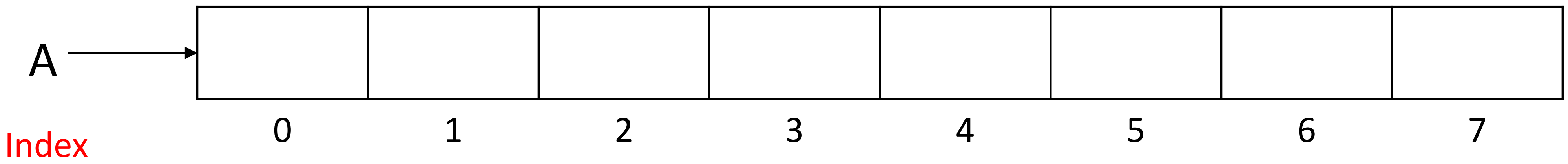
After



capacity = 8
top = 4

Push ‘T’ onto an empty Stack

Before

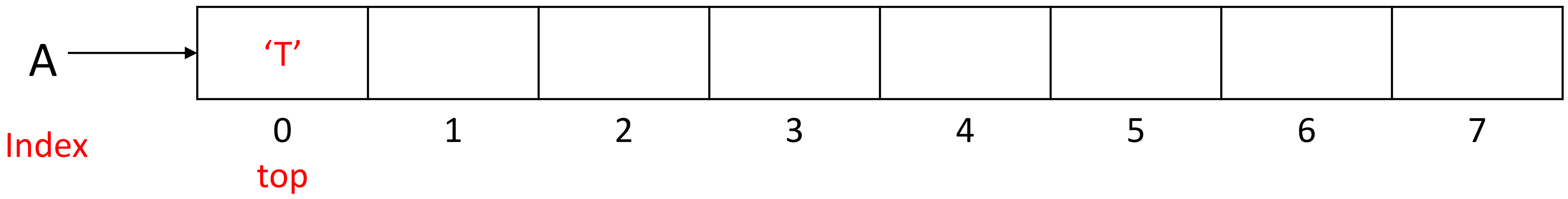


capacity = 8

top = -1

Push ‘T’ onto an empty Stack

After



capacity = 8

top = 0

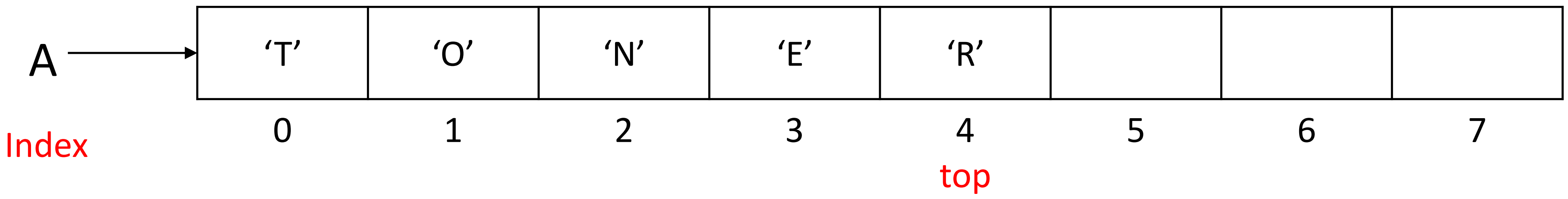
Pop an item off the Stack

- Basic Strategy
  - If the stack is empty, then throw an exception else decrease top by 1.

```
public void Pop()
{
    if (Empty())
        throw new InvalidOperationException("Stack is empty");
    else
        top--;
}
```

Pop an item off the Stack

Before

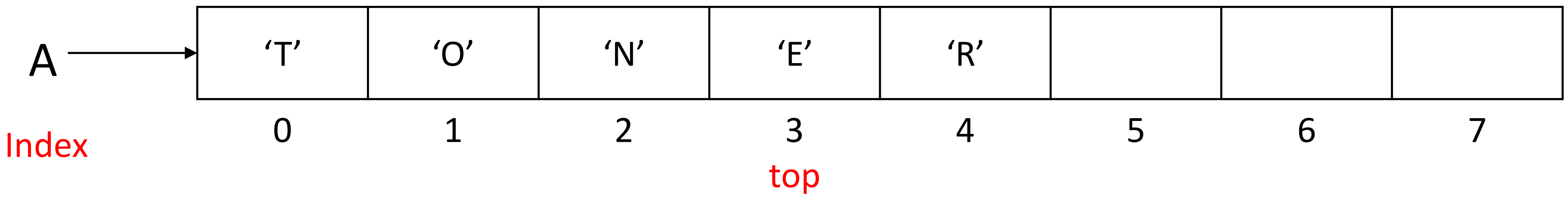


capacity = 8

top = 4

Pop an item off the Stack

After



capacity = 8

top = 3

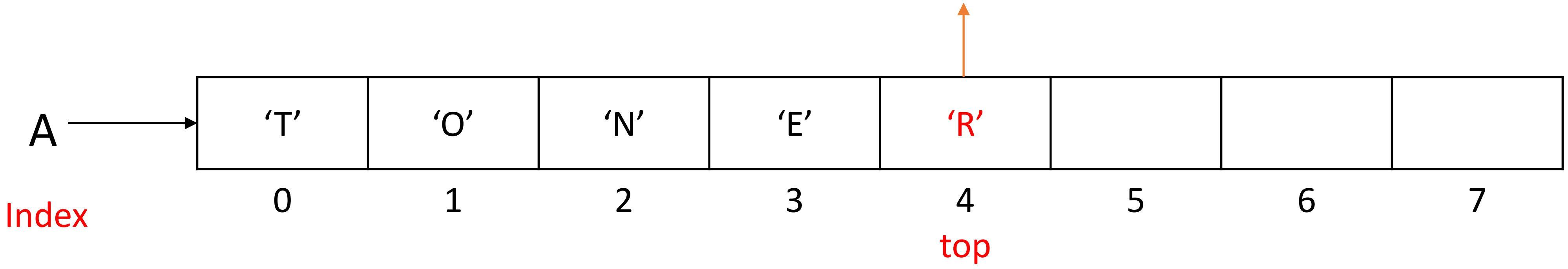
Although ‘R’ remains in the array, it is not accessible to the user and is overwritten with the next Push.

Retrieve the top item of a Stack

- Basic Strategy
  - If the stack is empty, then throw an exception else return A[top].

```
public T Top()
{
    if (Empty())
        throw new InvalidOperationException("Stack is empty");
    else
        return A[top];
}
```

Retrieve the top item of a Stack



capacity = 8

top = 4

Supporting methods

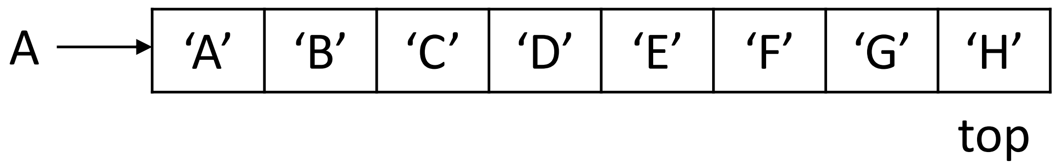
- MakeEmpty
  - Sets top to -1 (only)
  - O(1)
- Empty
  - Returns true if top is -1; false otherwise
  - O(1)
- Size
  - Returns top + 1
  - O(1)

Supporting methods

- DoubleCapacity
  - Doubles the capacity of the current Stack

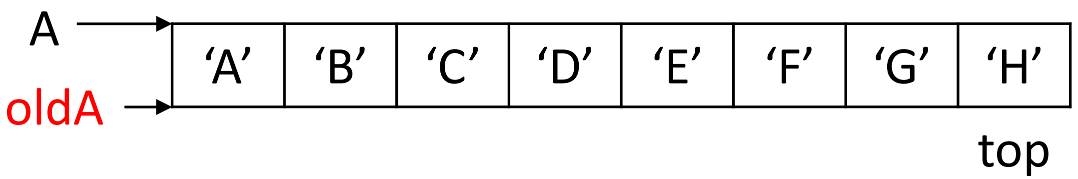
```
private void DoubleCapacity()
{
    int i;
    T[] oldA = A;

    capacity = 2 * capacity;
    A = new T[capacity];

    for (i = 0; i <= top; i++)
        A[i] = oldA[i];
}
```

A full stack A



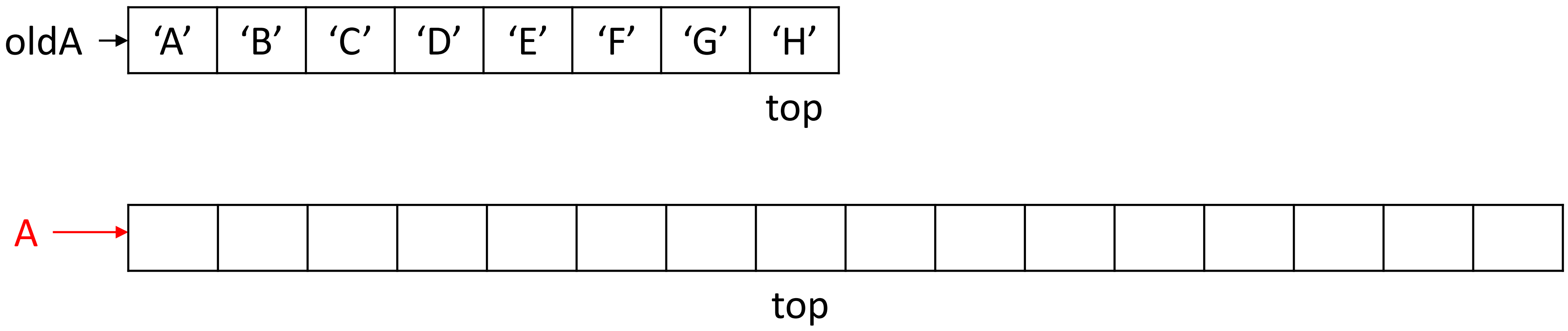
capacity = 8
top = 7

Set oldA to A



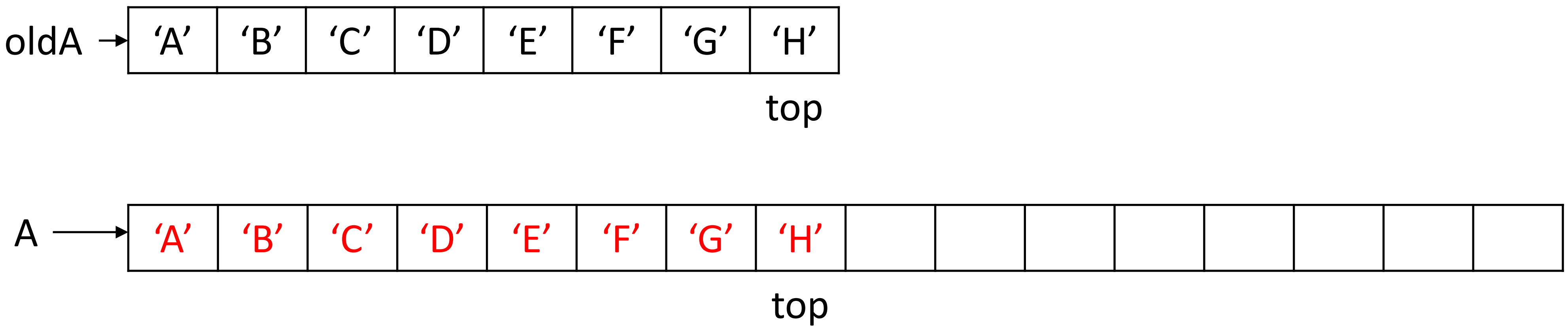
capacity = 16
top = 7

Set A to an array of twice the capacity



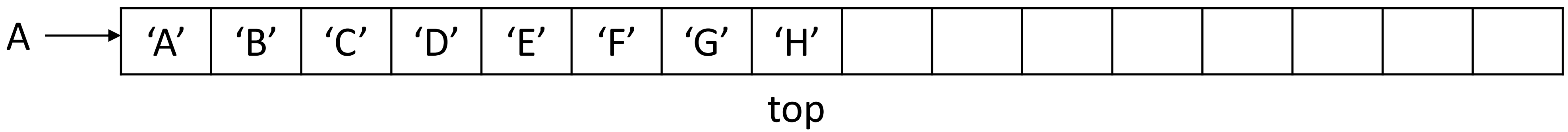
capacity = 16
top = 7

Copy items of oldA to A



capacity = 16
top = 7

Original stack A with twice the capacity



capacity = 16
top = 7

Data structures

- Linear array
- Singly linked list ←

Singly linked list

Data members

```
public class Stack<T> : IStack<T>
{
    ...                         // Node class
    private Node top;           // Reference to the top item
    private int count;          // Number of items in the stack
    ...
}
```

Node class (within Stack<T>)

```
private class Node
{
    public T Item       { get; set; }
    public Node Next    { get; set; }

    public Node()
    {
        Next = null;
    }
    public Node(T item, Node next)
    {
        Item = item;
        Next = next;
    }
}
```

Read/Write Properties

Constructor

- Basic strategy:
  - Set the top to null and count to 0.
  - Use the MakeEmpty method.

```
public Stack()
{
    MakeEmpty();
}
```

$$\left\{ \begin{array}{l} \text{top} \quad = \text{null} \\ \text{count} = 0 \end{array} \right.$$

Push an item onto the top of the Stack

- Basic Strategy
  - Create a new Node to store the item and set its Next field to top.
  - Set top to the new Node and increase count by 1.

```
public void Push(T item)
{
    top = new Node(item, top);
    count++;
}
```

Push ‘Y’ onto the top of an empty Stack

Before

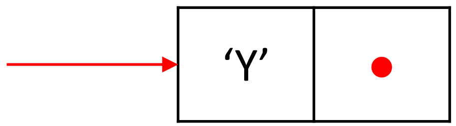
top •

count = 0

Push ‘Y’ onto the top of an empty Stack

After

top

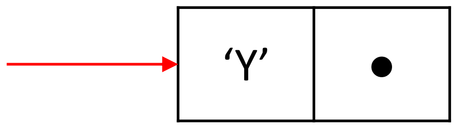


count = 1

Push ‘B’ onto the top of the Stack

Before

top

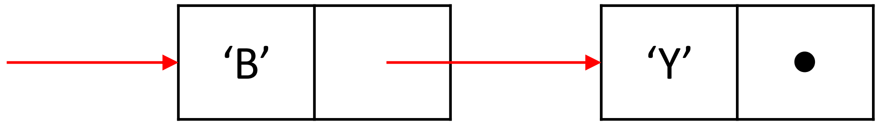


count = 1

Push ‘B’ onto the top of the Stack

After

top



count = 2

Like adding to the front of a List

Pop an item off the Stack

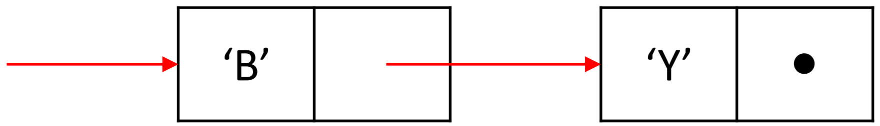
- Basic Strategy
  - If the Stack is empty, throw an exception else move top to the next Node and decrease count by 1.

```
public void Pop()
{
    if (Empty())
        throw new InvalidOperationException("Stack is empty");
    else {
        top = top.Next;
        count--;
    }
}
```

Pop an item off the Stack

Before

top

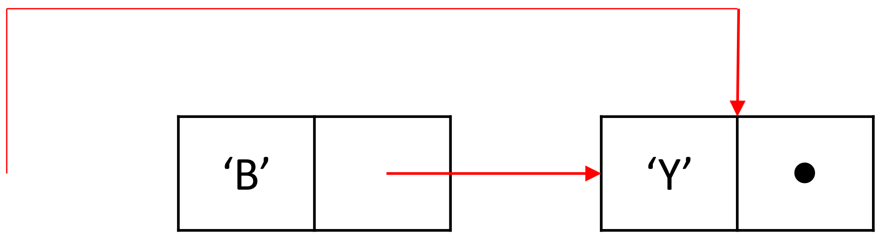


count = 2

Pop an item off the Stack

After

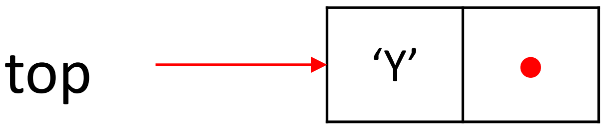
top



count = 1

Pop an item off the Stack

Before



count = 1

Pop an item off the Stack

After

top •

count = 0

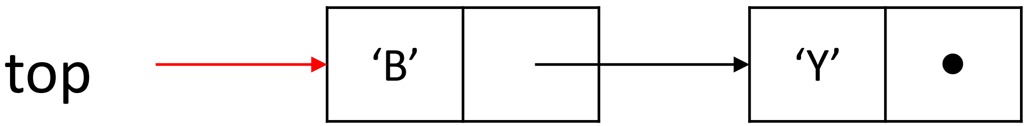
Retrieve the top item of a Stack

- Basic Strategy
  - If the Stack is empty, throw an exception else return the top item.

```
public T Top()
{
    if (Empty())
        throw new InvalidOperationException("Stack is empty“);
     else
         return top.Item;
}
```

Retrieve the top item of a Stack

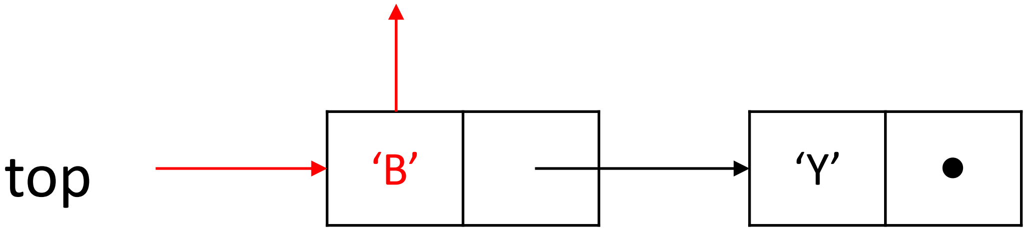
Before



count = 2

Retrieve the top item of a Stack

After



count = 2

Supporting methods

- MakeEmpty
  - Sets top to null and count to 0
  - O(1)
- Empty
  - Returns true if count is 0; false otherwise
  - O(1)
- Size
  - Returns count
  - O(1)

Worst-case time complexity

- The worst-case time complexity of all primary and supporting methods of a Stack (except one) is constant and expressed as O(1).

- Which one requires more time?

Exercises

1. Using an instance of stack, write a program that reads in a sequence of characters and prints them in reverse order.

2. Write a program that reads in a sequence of characters, and determines whether its parentheses, square brackets, and curly braces are “balanced”. For example, the sequence ([ ]) { } is balanced but ({]) } [is not.

3. Using a Stack, write an additional method for the singly linked list implementation of List to reverse the order of its items.

Exercises (con't)

4. A postfix expression is an arithmetic expression where the binary operator comes after its two operands. For example:

   a. (5 + 3) * 9 is expressed in postfix form as 5 3 + 9 *
   b. 8 / 4 / 2 is expressed in postfix form as 8 4 / 2 /
   c. 7 + (6 – 5) * 8 is expressed in postfix form as 7 6 5 – 8 * + (using BEDMAS)

   How can a stack be used to evaluate a given postfix expression? Notice that:

   a. A postfix expression does not require parentheses
   b. Given a postfix expression, the order of evaluation is unambiguous