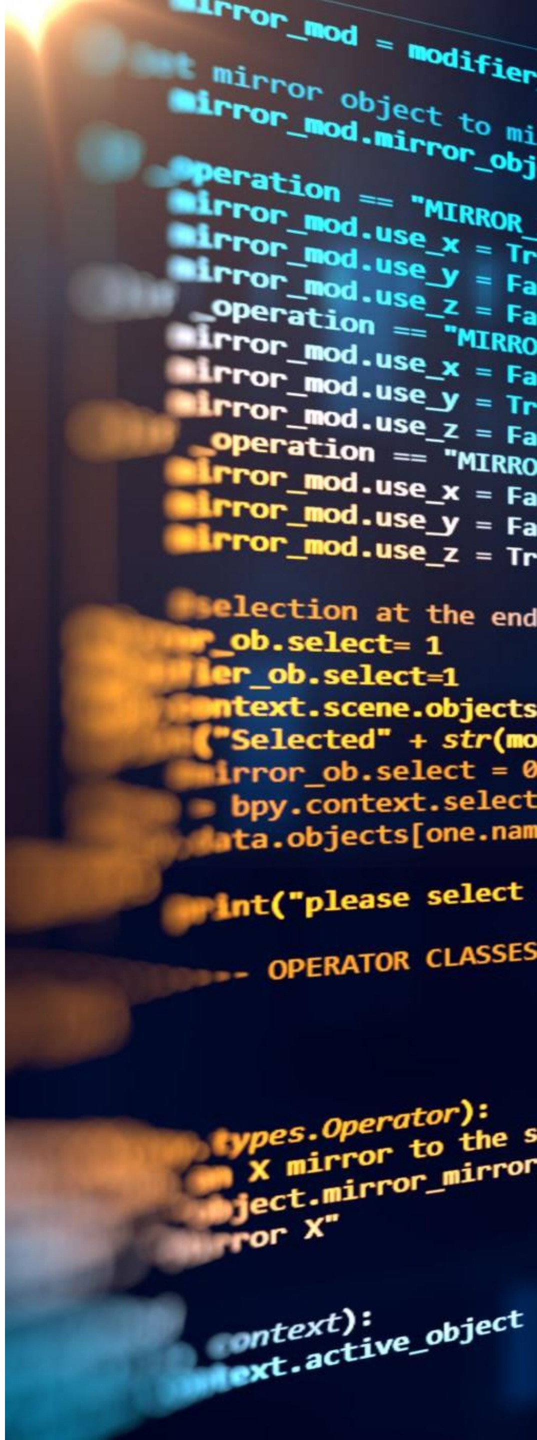
Please use the following QR code to check in and record your attendance.

CS 1027
Fundamentals of Computer Science II

Stack ADT (cont.)

Ahmed Ibrahim



Lecture Recap

Abstract Data Types (ADTs)

- ADTs encapsulate data and operations on a collection of data elements, providing a clear interface while hiding implementation details.
- ADTs focus on "**what**" operations can be performed rather than "**how**" they are executed.
- **Examples**: **Stacks**, **queues**, **lists**, and **trees** are commonly used ADTs in programming, each with specific operations associated with it.

Lecture Recap

Implementing ADTs in Java Using Interfaces

- **Java Interfaces**: Interfaces are used to define ADTs by specifying method **signatures** without implementation.
- Example: A BagADT interface might include methods like add, remove, and count, each outlining operations on the collection while leaving the implementation details open.

Lecture Recap

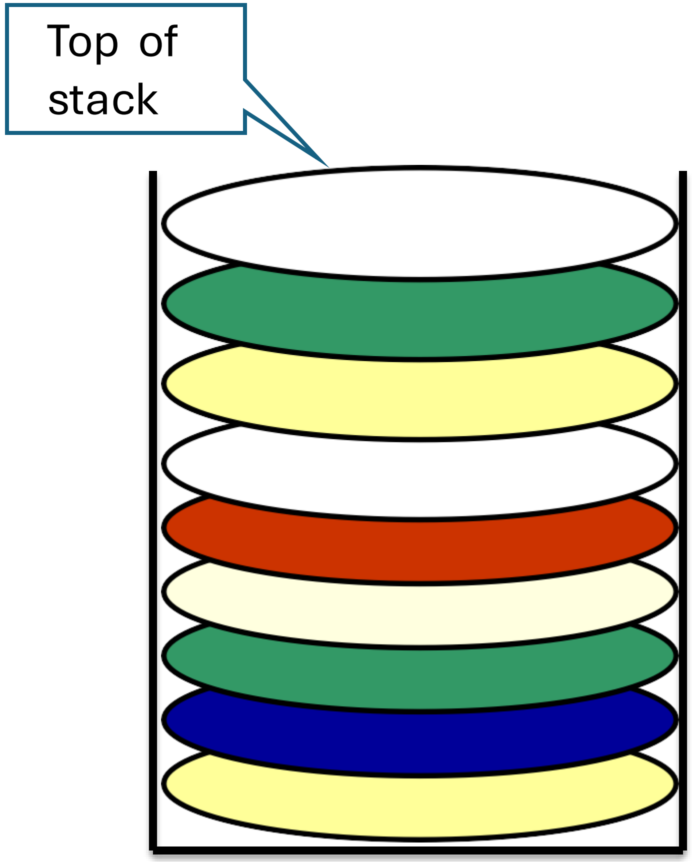
The Stack ADT

- A stack is a collection organized to restrict access to the last item added (LIFO).

Core Operations:

- **Push**: Adds an element to the top.
- **Pop**: Removes and returns the top element.
- **Peek**: Returns the top element without removing it.

Applications: Commonly used in scenarios like undo functionality, expression evaluation, and parsing.



Lecture Recap

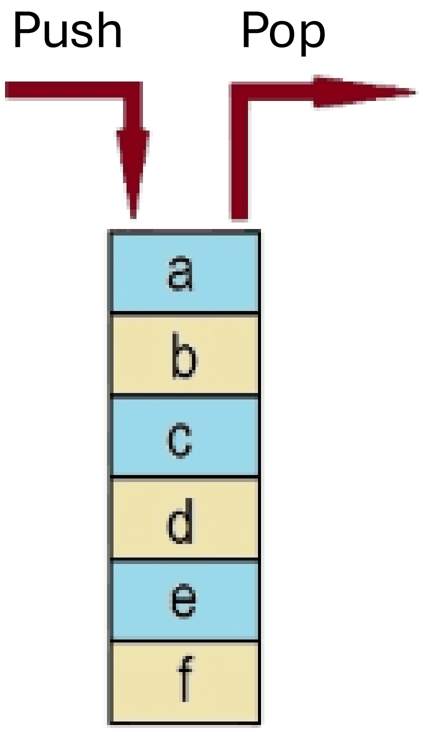
- Implementing a Stack with Arrays
  - ArrayStack Implementation:
    - It uses an array to hold stack elements with a top variable pointing to the last element.
    - Demonstrates memory handling in Java with generics and the use of casting for arrays of generic types.
  - Provided Code/Algorithms:
    - Initialization, push, pop, peek, size, and isEmpty methods.

Stack ADT With Linked List

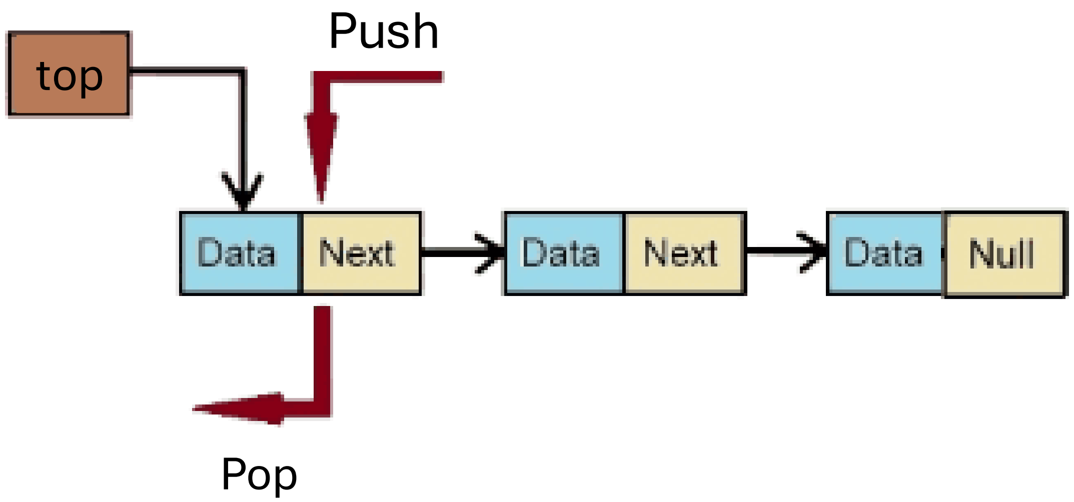
Another Stack Implementation

- We will now examine implementing a linked list of the Stack ADT.
- In this version, the stack's data items are stored in the nodes of a **linked list**.
- This linked list-based implementation will use the <u>same Stack ADT interface</u> as the array-based version but with a different <u>underlying data structure</u>.

```
public interface StackADT<T>
{
    public void push (T dataItem); // Adds one  // data item to the
    top of this stack

    public T pop( ); // Removes and returns the
    // top data item of this stack

    public T peek( ); // Returns the top data
    // item of this stack

    public boolean isEmpty( ); // Returns true
    // if this stack is empty

    public int size( ); // Returns the number of data items in this
    stack

    public String toString( ); // Returns a
    // string representation of this stack

}
```



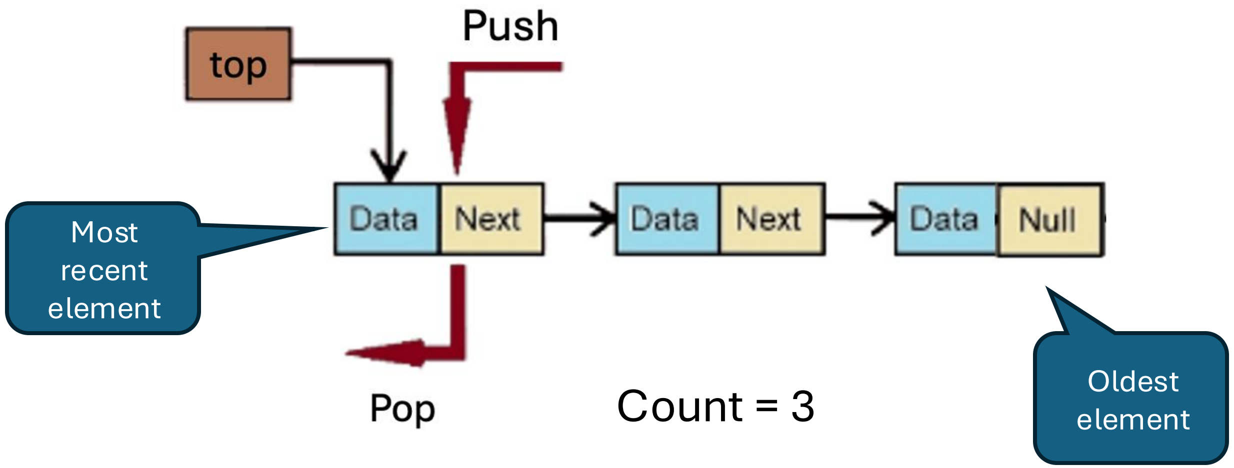
Array



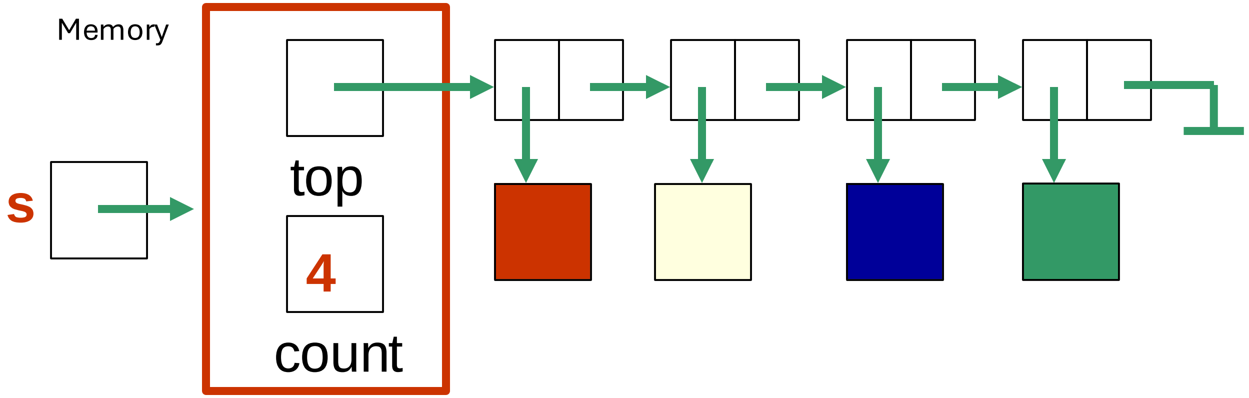
Linked List

Linked List Implementation of Stack

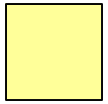
- Each node in the linked list contains a data item, with the top of the stack represented by the first node of the list.
- A reference to this first node (which we'll call top) refers to the entire linked list.
- Additionally, we'll maintain a variable, count, to track the number of elements in the stack.

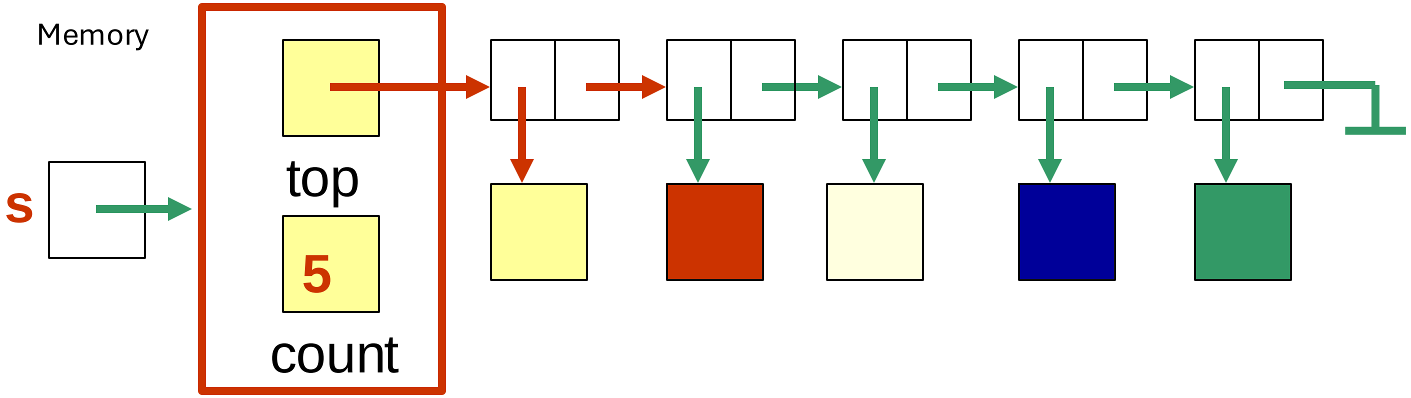


A stack s with 4 elements

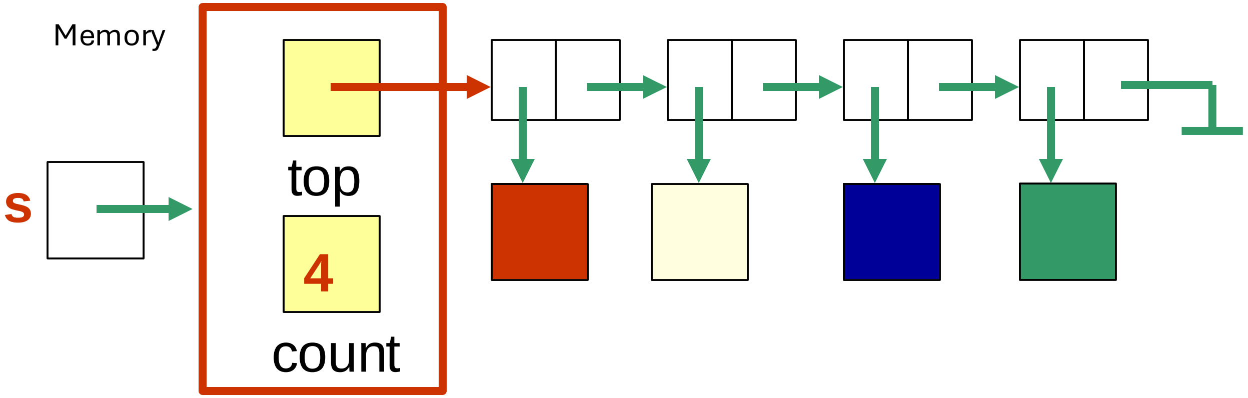


After pushing a fifth element

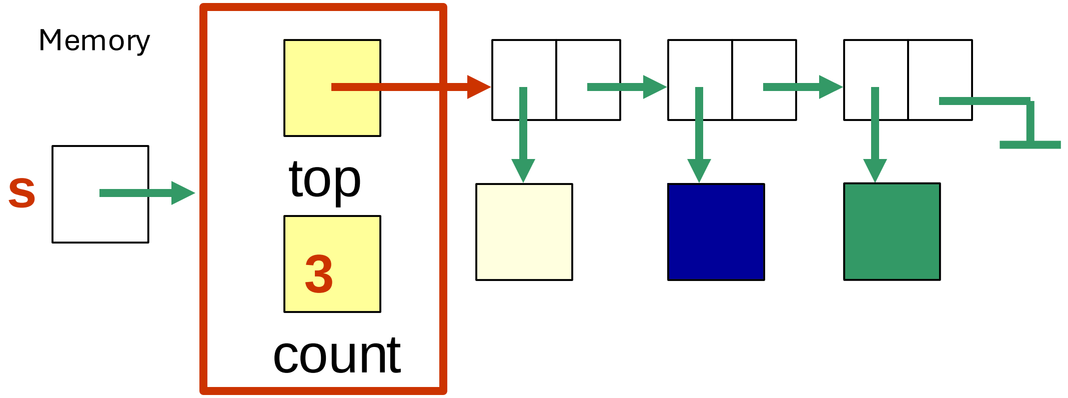




After popping an element

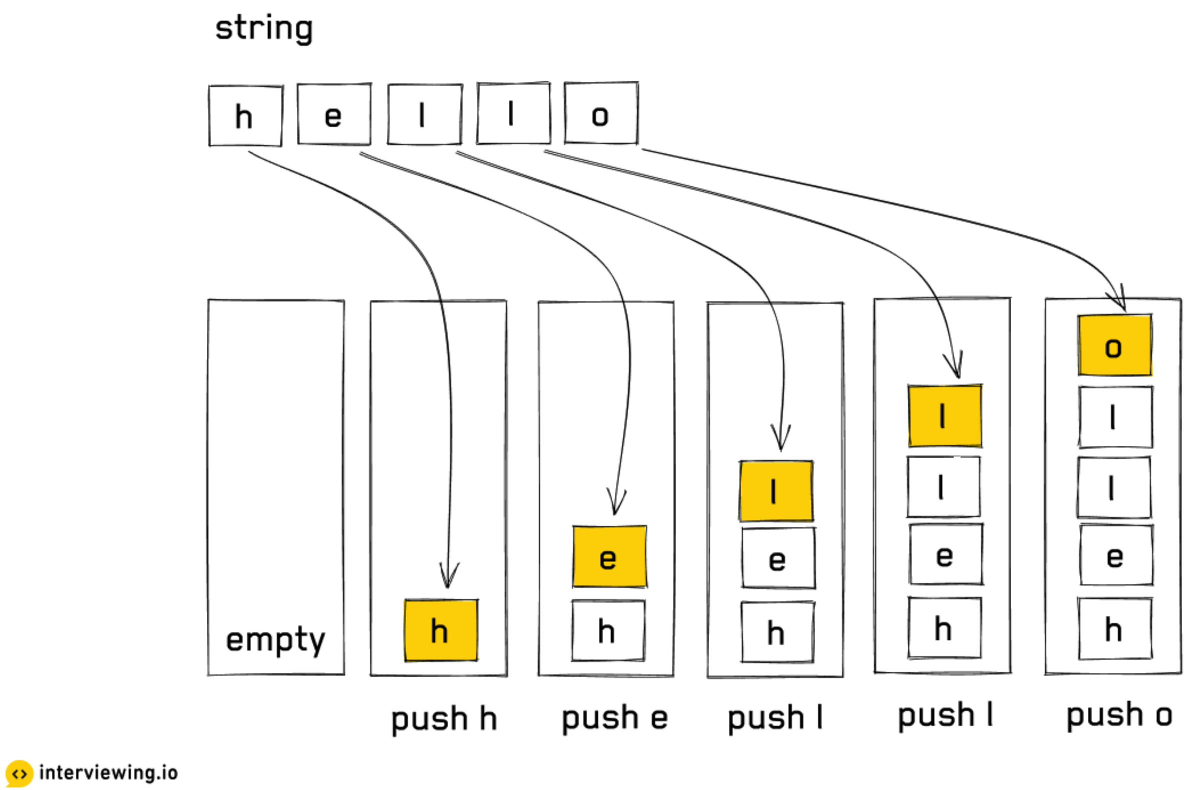


After popping another element

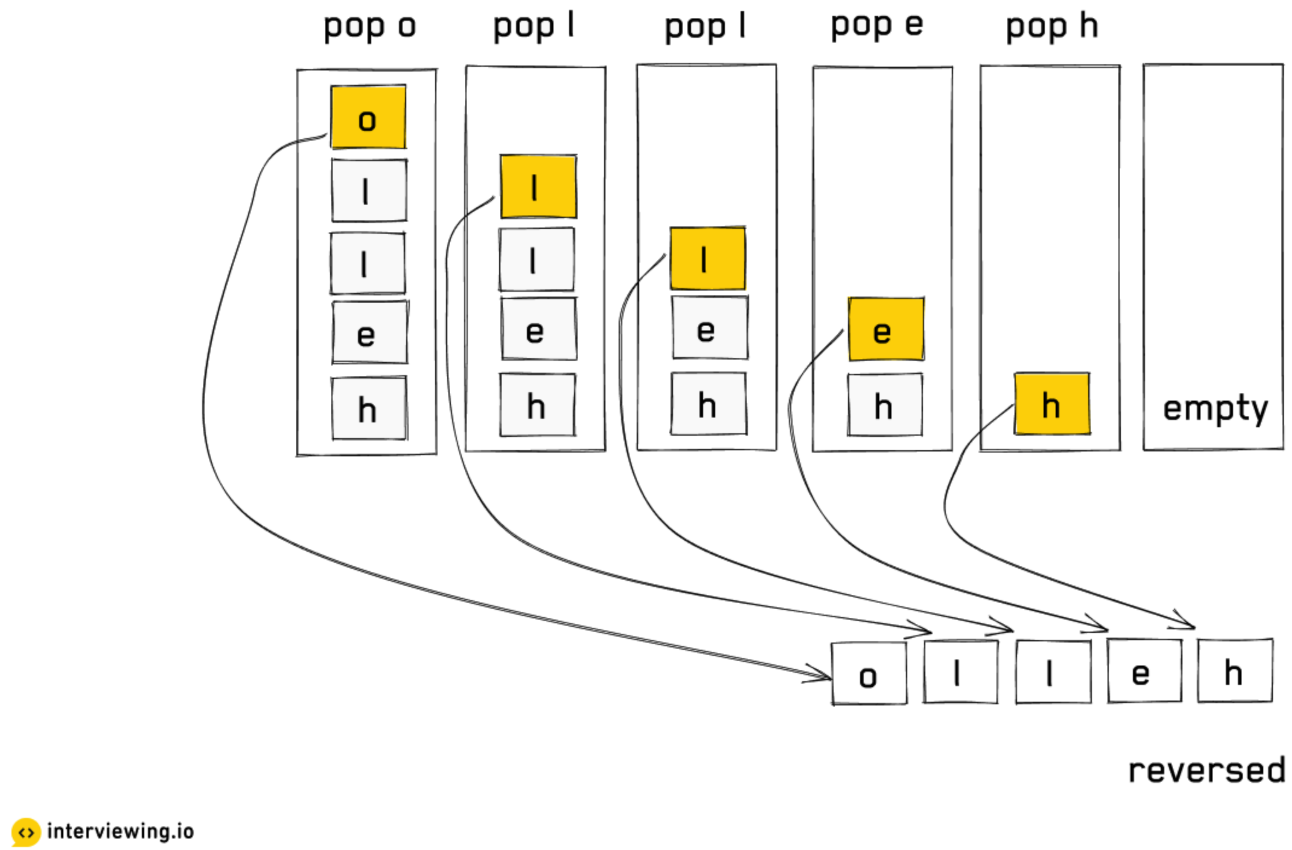


Applications of Stack

Reverse A String /1



Reverse A String /2



Pseudocode to Reverse a String Using a Stack

1. Initialize an empty stack.
2. Push characters onto the stack:
   1. For each character c in the input string: Push c onto the stack.
3. Pop characters from the stack:
   1. Initialize an empty string reversed_string.
   2. While the stack is not empty:
      1. Pop the top character from the stack.
      2. Append this character to reversed_string.
4. Output reversed_string.

Question!

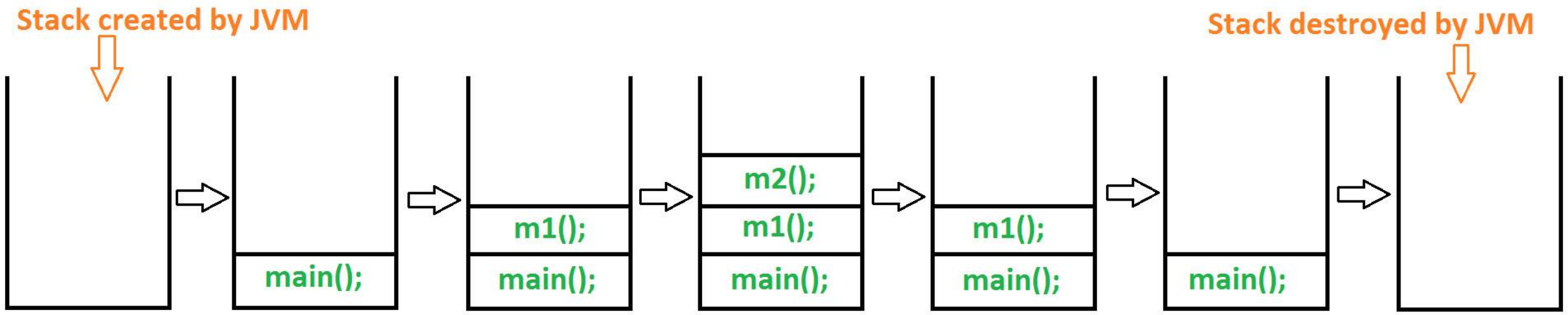
Consider the given pseudocode to reverse a string using a stack, and answer the following:

a) Suppose we want to reverse each word in a sentence individually (e.g., "Hello World" becomes "World Hello"). Modify the pseudocode to handle this case.

b) Discuss any additional complexity or edge cases that may arise in this modified version, such as handling multiple spaces or special characters.

Call Stack

```
public class HaveAnIdeaAboutSomething{
   public static void main(String ... args){ m1(); }
public static void m1(){ m2(); }
public static void m2(){ System.out.println("I Like 7"); }
}
```



Source: https://www.linkedin.com/pulse/java-runtime-stack-mechanism-jrsm-youssef-najeh/

Question!

Consider the following Java code:

```java
public class RecursiveExample {
  public static void main(String[] args) {
   int result = recursiveMethod(4);
   System.out.println("Result: " + result);
}
public static int recursiveMethod(int n) {
  if (n <= 1) {return n;} else {
   int temp = recursiveMethod(n - 1);
   int result = n + temp;
   return result;}
   }
}
```

a) Draw the call stack at the point when recursiveMethod(2) is called for the first time. Show the value of **n** and any intermediate variables (e.g., temp, result) at each stack level.

b) Describe what would happen if this code were modified to remove the base case (if (n <= 1) return n;). Explain how this change would impact the call stack and why it could lead to a stack overflow error.

Source: https://www.linkedin.com/pulse/java-runtime-stack-mechanism-jrsm-youssef-najeh/

Another Stack Implementation

- Infix and Postfix notations are two different but equivalent notations for writing algebraic expressions.

Infix	Postfix	Notes
A * B + C / D	A B * C D / +	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	divide C by D, add B, multiply by A

Infix Notation

- The operator is placed between the operands while writing an arithmetic expression using infix notation.
- For example, A+B; here, the plus operator is placed between the two operands A and B.
- Although writing expressions using infix notation is easy, <u>computers find it difficult to parse</u> as they need a lot of information to evaluate the expression.
- So, computers work more efficiently with expressions written using **prefix** notation.

Postfix Notation

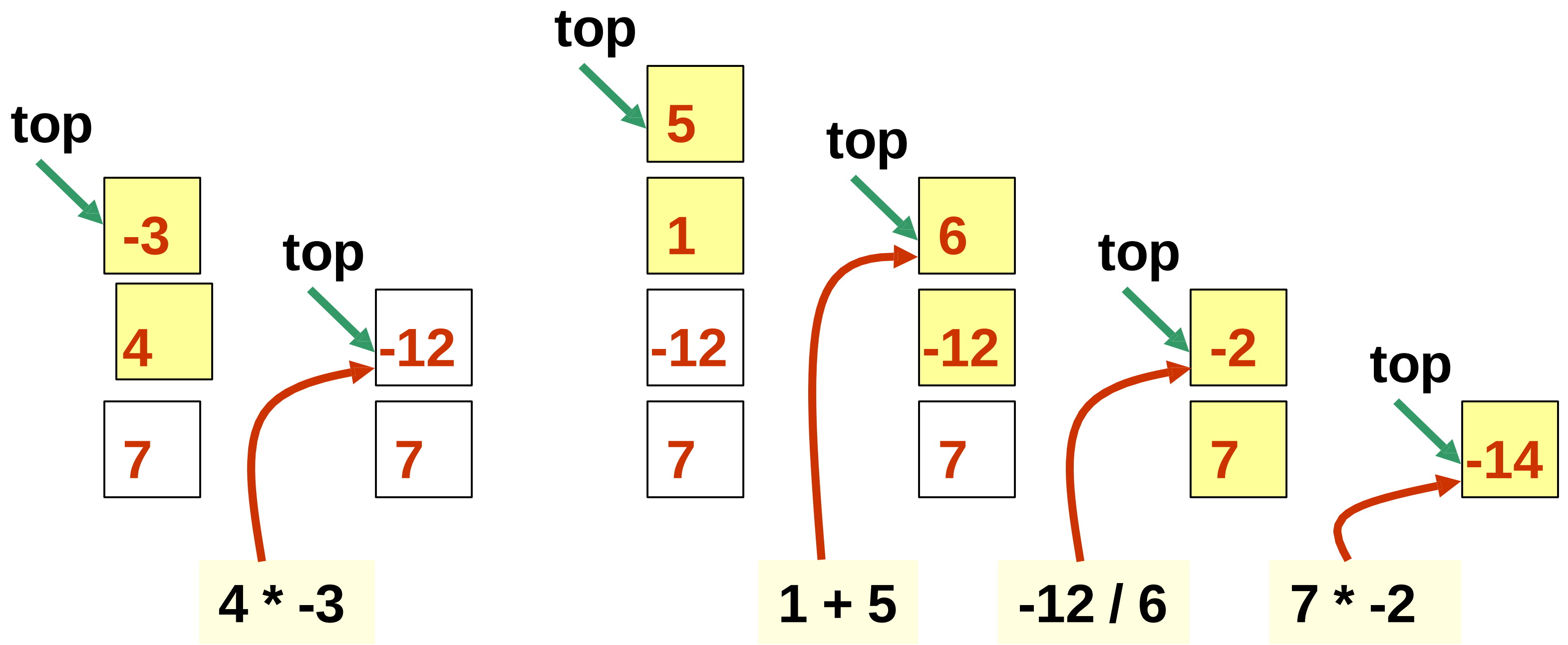
- A Polish mathematician gave Postfix notation. His aim was to develop a **parenthesis-free** prefix notation (also known as Polish notation).
- In postfix notation, the operator is placed after the operands. For example, if an expression is written as A+B in **infix** notation, the same expression can be written as AB+ in **postfix** notation.
- The order of postfix expression evaluation is always from **left** to **right**.

Evaluating A Postfix Expression

- Algorithm to evaluate a postfix expression:
  - Scan from left to right, determining if the next term is an operator or operand
  - If it is an operand, push it on the stack
  - If it is an operator, pop the stack twice to get the two operands, perform the operation, and push the result back onto the stack.
- Try the algorithm in the following example; ultimately, the stack will contain a single value.

Using a Stack to Evaluate a Postfix Expression

Evaluation of 7 4 -3 * 1 5 + / *



The result is the only item on the stack at the end of the evaluation.

Stack Applications: Undo and Redo Functionality

- Two stacks can provide undo and redo functionality for text editors or graphic design software applications.
- **Approach**:
  - Use one stack (undoStack) to store each action performed by the user.
  - When an action is undone, pop from undoStack and push it onto a redoStack.
  - When a redo is performed, pop from redoStack and push it back to undoStack.
- Many interactive applications, including text editors and drawing tools, use this approach to allow users to undo and redo their actions.

Stack Applications: Sorting a Stack

- Two stacks can be used to sort elements within a stack.
- **How to do that**
  - Use one stack as the main stack and the other as a temporary holding stack.
  - Transfer elements from the main stack to the temporary stack, keeping them sorted as you do so.
- This is useful in applications where sorting needs to be done using only stacks, such as certain restricted algorithms or memory-limited embedded systems.

Stack Applications: Backtracking

- Backtracking in **maze-solving** or **Pathfinding**
- Two stacks can aid in implementing backtracking algorithms for problems like maze solving.
- **How to do that**
  - Use one stack to track the current path as you explore.
  - Use a second stack to store alternate paths or checkpoints, allowing you to backtrack effectively.
- This is useful in real-time pathfinding applications, games, or solving puzzles like mazes where alternative paths must be tracked.

Stack Applications: Queue Simulation

1. Initialize a stack pathStack to keep track of the current path.

2. Start at the maze entry point and push it onto pathStack.

3. While pathStack is not empty:

   a. Set currentPosition to the top of pathStack.

   b. If currentPosition is the exit point:

      - Solution is found. Print the path from pathStack and exit.

   c. Otherwise:

      - Find all possible moves (up, down, left, right) from currentPosition.

      - Filter moves to find the first valid and unexplored path.

   d. If a valid move is found:

      - Mark the currentPosition as visited to avoid revisiting it.

      - Push the new position onto pathStack.

   e. If no valid moves are available (dead-end):

      - Backtrack by popping the top of pathStack to return to the previous position.

4. If pathStack is empty and exit is not found, return "No solution exists".

Stack Applications: Queue Simulation

- To simulate a queue using two stacks, we can use the following approach:

1. **Define Two Stacks**:
    1. **Stack1** (inStack): Used to store elements as they are enqueued.
    2. **Stack2** (outStack): Used to retrieve elements in the correct FIFO order during dequeuing.
2. **Enqueue Operation**:
    1. Push the new element onto inStack.
    2. This keeps inStack in LIFO order, as usual for a stack.
3. **Dequeue Operation**:
    1. If outStack is empty, move all elements from inStack to outStack (this reverses the order of elements).
    2. Pop the top element from outStack, which will now be the front of the queue.
    3. If outStack is not empty, simply pop the top element from outStack.
    4. This ensures that elements are dequeued in FIFO order.


Thank you