




# Expression Conversion using Stack

- 
- ▶ Infix to Postfix
  - ▶ Infix to Prefix
  - ▶ Postfix to Infix
  - ▶ Prefix to Infix
  - ▶ Prefix to Postfix
  - ▶ Postfix to Prefix

- ▶ Operands : Variables or Constants (A,b,5 ,6 etc.)
- ▶ Operators : +, -, \*, /, ^
- ▶ Infix : Operator **between** operands ie, A+B
- ▶ Postfix : Operator **after** operands ie, AB+
- ▶ Prefix : Operator before operands ie, +AB

Why Conversion??

- ❑ Computers find **postfix and prefix easier** to evaluate.
- ❑ Removes the need for parentheses.
- ❑ Stack-based algorithms are efficient.

Operator	Precedence	Associativity
$\wedge$	Highest	Right
$*$ , $/$	Medium	Left
$+$ , $-$	Lowest	Left



## ► Infix to Postfix Algorithm

- Initialize **empty stack** for operators.
- Scan the infix expression from **left to right**.
- If **operand**, add to postfix output.
- If **operator**:
  - Pop from stack to output until you find an operator with lower precedence or a left parenthesis.
  - Push current operator.
- If '(' push to stack
- If ')' pop until '(' is found
- At end, pop all remaining operators to output.

Expression : **A+B\*C**

Output: A

Push +

Output: AB

Push \*

Output: ABC

Pop \*,+

**Output: ABC\*+**

## ► Infix to Prefix Algorithm

- Reverse the infix expression
- Swap '(' with ')' in the reversed string
- Convert the reversed infix to postfix using **Postfix algorithm**.
- Reverse the postfix result → final prefix.

Example:  $A+B*C$

Reverse:  $C*B+A$

Convert to postfix :  $CB*A+$

Reverse postfix :  $+A*BC \rightarrow$  prefix

## ► Postfix to Infix Algorithm

- Initialize **empty stack**.
- Scan postfix left to right:
  - If operand  $\rightarrow$  push to stack
  - If operator  $\rightarrow$  pop to operands, combine as (op1 operator op2), push to stack
- End: Top of stack = final infix expression.

Expression: ABC\*+

Push A, Push B, Push C

See \*  $\rightarrow$  Pop C, B  $\rightarrow$  (B\*C)  $\rightarrow$  Push

See +  $\rightarrow$  Pop (B\*C), A  $\rightarrow$  (A+(B\*C))



## ► Prefix to Infix Algorithm

- Initialize **empty stack**.
- Scan prefix right to left:
  - If operand  $\rightarrow$  push to stack
  - If operator  $\rightarrow$  pop to operands, combine as (op1 operator op2), push to stack
- End: Top of stack = final infix expression.

Expression: +A\*BC

Push C, Push B

See \*  $\rightarrow$  Pop B, C  $\rightarrow$  (B\*C)  $\rightarrow$  Push

Push A

See +  $\rightarrow$  Pop (B\*C), A  $\rightarrow$  (A+(B\*C))

## ► Postfix to Prefix Algorithm

- Initialize an **empty stack**.
- Scan postfix expression from left to right:
- If operand- $\rightarrow$  push onto the stack
- If Operator
  - Pop the top two elements (**op2 then op1**)
  - Form new string: **operator op1 op2**
  - Push this new string back onto the stack.
- At the end, the stack's top element is the **prefix expression**.

Expression: ABC\*+

Push A, Push B, Push C

See \*  $\rightarrow$  Pop C, B  $\rightarrow$  \*BC  $\rightarrow$  Push

See +  $\rightarrow$  Pop \*BC, A  $\rightarrow$  +A\*BC  $\rightarrow$  Push

Output: +A\*BC

## ► Prefix to Postfix Algorithm

- Initialize an **empty stack**.
- Scan postfix expression from right to left:
- If operand- $\rightarrow$  push onto the stack
- If Operator
  - Pop the top two elements (**op2 then op1**)
  - Form new string: **op1 op2 operator**
  - Push this new string back onto the stack.
- At the end, the stack's top element is the **postfix expression**.

Expression: +A\*BC

Push C, Push B

See \*  $\rightarrow$  Pop B, C  $\rightarrow$  BC\*  $\rightarrow$  Push

See +  $\rightarrow$  Pop A, BC\*  $\rightarrow$  ABC\*+  $\rightarrow$  Push

Output: ABC\*+

# Recursion

- ▶ The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.
- ▶ **Steps to Implement Recursion**
  - ❑ **Step1 - Define a base case:** Identify the simplest (or base) case for which the solution is known. This is the stopping condition for the recursion, as it prevents the function from infinitely calling itself.
  - ❑ **Step2 - Define a recursive case:** Define the problem in terms of smaller subproblems. Break the problem down into smaller versions of itself, and call the function recursively to solve each subproblem.
  - ❑ **Step3 - Ensure the recursion terminates:** Make sure that the recursive function eventually reaches the base case, and does not enter an infinite loop.
  - ❑ **Step4 - Combine the solutions:** Combine the solutions of the subproblems to solve the original problem.

## ► Comparison of Recursive and Iterative Approaches

Approach	Complexity	Memory Usage
Iterative Approach	$O(n)$	$O(1)$
Recursive Approach	$O(n)$	$O(n)$

## ► Need For Recursion

- ❑ Recursion helps in logic building. Recursive thinking helps in solving complex problems by breaking them into smaller subproblems.
- ❑ Recursive solutions work as a basis for Dynamic Programming and Divide and Conquer algorithms.

## ► Example 1: Sum of Natural Numbers

```
int sum(int n) {  
    int res = 0;  
    for (int i = 1; i <= n; ++i) {  
        res += i;  
    }  
    return res;  
}
```

Iterative

If (n==1)  
 return 1;

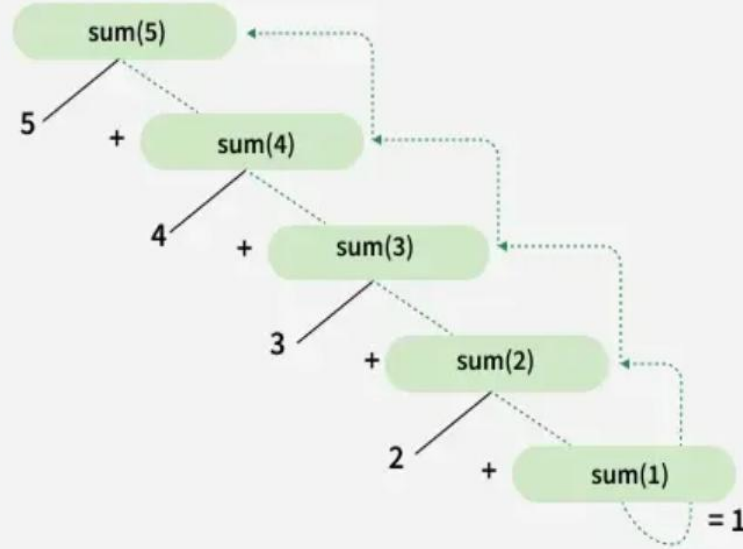
$\text{sum}(n) = n + \text{sum}(n-1)$

Recursive

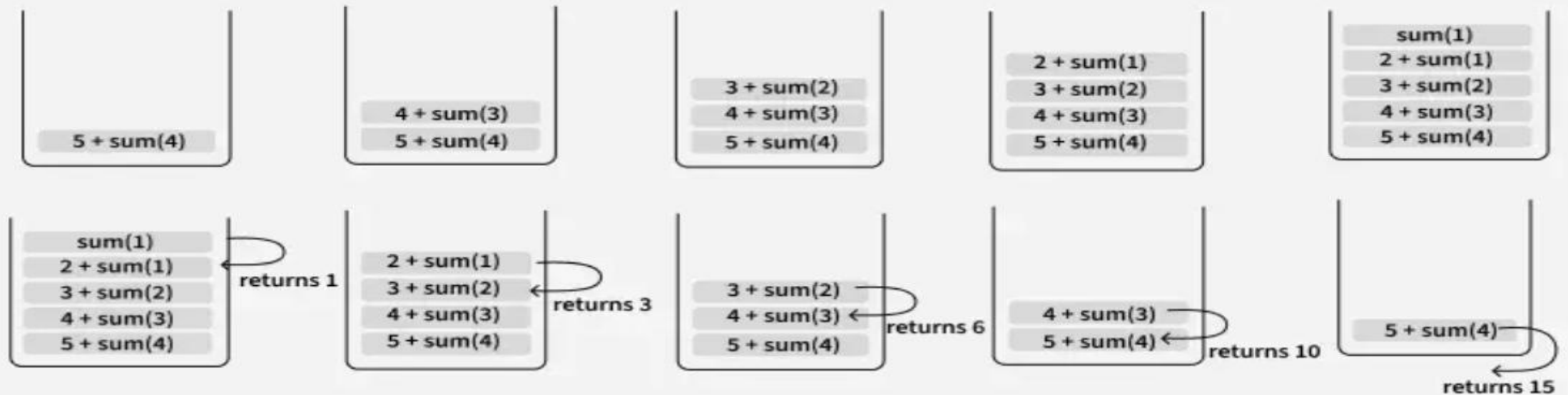


$\text{sum}(5) = 5 + \text{sum}(4)$   
 $\text{sum}(4) = 4 + \text{sum}(3)$   
 $\text{sum}(3) = 3 + \text{sum}(2)$   
 $\text{sum}(2) = 2 + \text{sum}(1)$

## Recursive Tree

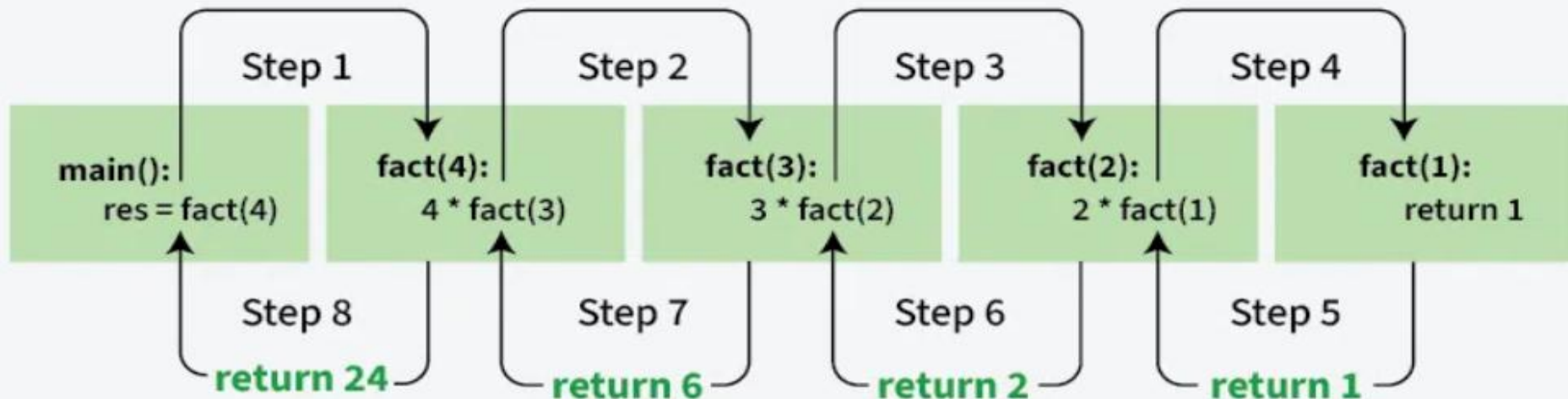


## Recursive Stack



## ► Example 2: Factorial of a Number

```
public class factorial{  
    public static int fact(int n) {  
        if (n == 0)  
            return 1;  
        return n * fact(n - 1); }  
    public static void main(String[] args) {  
        System.out.println("Factorial of 5 : " + fact(5));}}}
```





► Example 3: minimum index finder

```
static int minIndex(int arr[], int s, int e){
    int sml = Integer.MAX_VALUE;
    int minindex = ;
    for (int i = s; i < e; i++) {
        if (sml > arr[i]) {
            sml = arr[i];
            minindex = i;}}
    return minindex; }

static void fun2(int arr[], int start_index, int end_index)
{
    if (start_index >= end_index)
        return;
    int min_index;
    int temp;
    min_index = minIndex(arr, start_index, end_index);
    temp = arr[start_index];
    arr[start_index] = arr[min_index];
    arr[min_index] = temp;
    fun2(arr, start_index + 1, end_index);
}
```

► Answer it

```
static int fun1(int n)
{
    if (n == 1)
        return 0;
    else
        return 1 + fun1(n / 2);
}
```

---

```
static void fun2(int n)
{
    if(n == 0)
        return;
    fun2(n/2);
    System.out.println(n%2);
}
```

► Answer it

```
static int fun1 (int x, int y)
{
    if (x == 0)
        return y;
    else
        return fun1 (x - 1, x + y);
}
```

---

```
static void fun1 (int n)
{
    int i = 0;
    if (n > 1)
        fun1 (n - 1);
    for (i = 0; i < n; i++)
        System.out.print(" * ");
}
```

► Answer it

```
import java.io.*;
class Recursion {
    static void fun(int x)
    {
        if(x > 0)
        {
            fun(--x);
            System.out.print(x + " ");
            fun(--x);
        }
    }

    public static void main (String[] args)
    {
        int a = 4;
        fun(a);
    }
}
```

## ► Direct and Indirect recursion

- ❑ A function is called **direct recursive** if it calls itself directly during its execution. In other words, the function makes a recursive call to itself within its own body.
- ❑ An **indirect recursive function** is one that calls another function, and that other function, in turn, calls the original function either directly or through other functions. This creates a chain of recursive calls involving multiple functions, as opposed to direct recursion, where a function calls itself.

Recursion	Iteration
Terminates when the base case becomes true.	Terminates when the loop condition becomes false.
Logic is built in terms of smaller problems.	Logic is built using iterating over something.
Every recursive call needs extra space in the stack memory.	Every iteration does not require any extra space.
Smaller code size.	Larger code size.

## ► Common Applications of Recursion

- ❑ **Tree and Graph Traversal:** Used for systematically exploring nodes/vertices in data structures like trees and graphs.
- ❑ **Sorting Algorithms:** Algorithms like quicksort and merge sort divide data into subarrays, sort them recursively, and merge them.
- ❑ **Divide-and-Conquer Algorithms:** Algorithms like binary search break problems into smaller subproblems using recursion.
- ❑ **Fractal Generation:** Recursion helps generate fractal patterns, such as the Mandelbrot set, by repeatedly applying a recursive formula.
- ❑ **Backtracking Algorithms:** Used for problems requiring a sequence of decisions, where recursion explores all possible paths and backtracks when needed.
- ❑ **Memoization:** Involves caching results of recursive function calls to avoid recomputing expensive subproblems.

# Recursion Using Linked List

```
class Node {  
    int data;  
    Node next;
```

```
    Node(int new_data) {  
        data = new_data;  
        next = null;  
    }
```



```
public class ListRecursion {  
    static Node insertEnd(Node head, int data) {  
        if (head == null)  
            return new Node(data);  
        head.next = insertEnd(head.next, data);  
        return head; }  
    static void traverse(Node head) {  
        if (head == null) return;  
        System.out.print(head.data + " ");  
        traverse(head.next); }  
    public static void main(String[] args) {  
        Node head = null;  
        head = insertEnd(head, 1);  
        head = insertEnd(head, 2);  
        head = insertEnd(head, 3);  
        traverse(head); }}  

```



# Tail Recursion

```
static Node insertEnd(Node head, int data) {  
    if (head == null)  
        return new Node(data);  
    insertEndHelper(head, data);  
    return head; // head never changes  
}  
  
static void insertEndHelper(Node current, int data) {  
    if (current.next == null) {  
        current.next = new Node(data);  
        return;  
    }  
    insertEndHelper(current.next, data);  
}
```

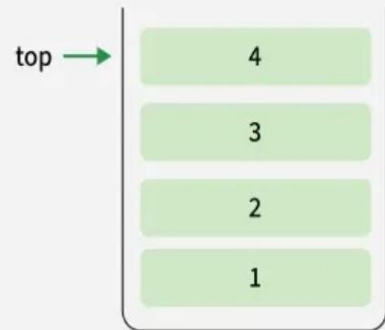
```
class Node {
    int data;
    Node next;
    Node(int x){
        data = x;
        next = null;}}
class LLRecurison{
    static Node reverseList(Node head){
        if (head == null || head.next == null)
            return head;
        Node revHead = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return revHead;
    }
}
```



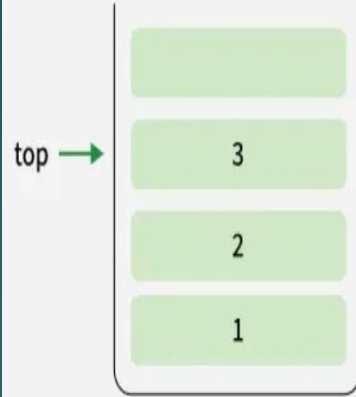


# Reverse a Stack using Recursion

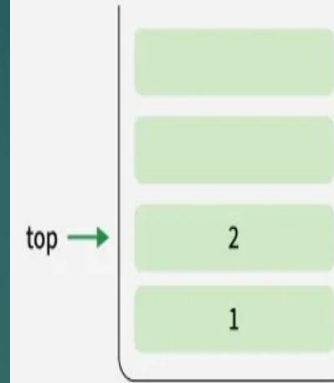
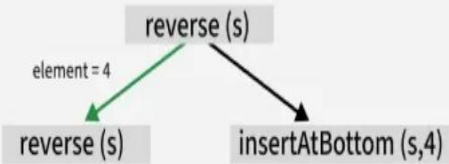
Consider the stack  $s$  with elements: 1 2 3 4



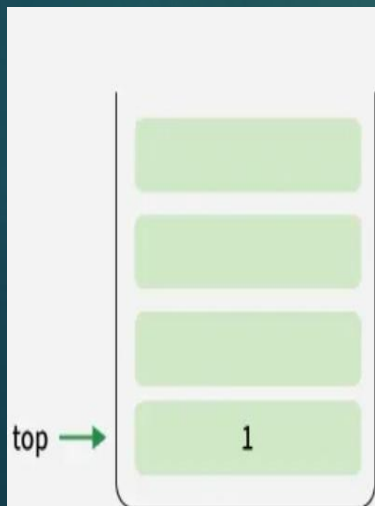
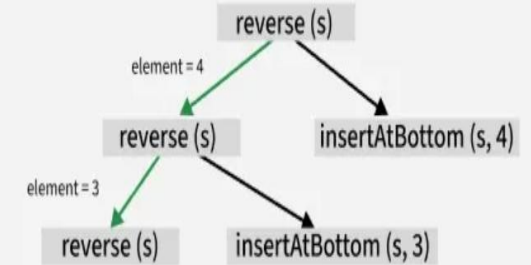
stack  $s$



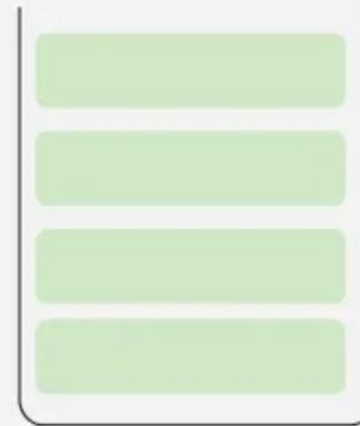
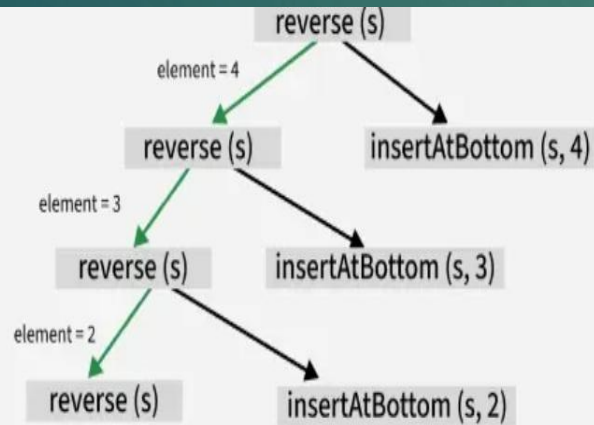
stack  $s$



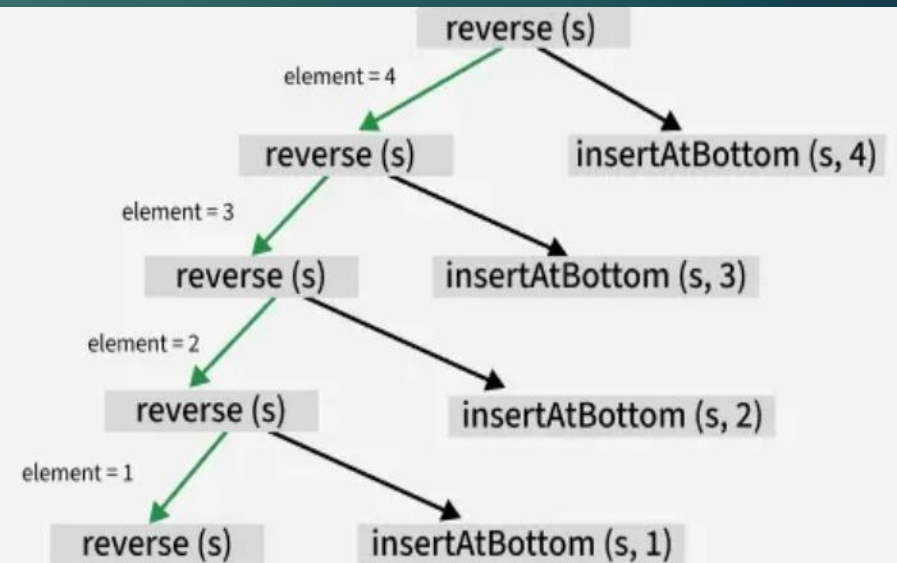
stack  $s$

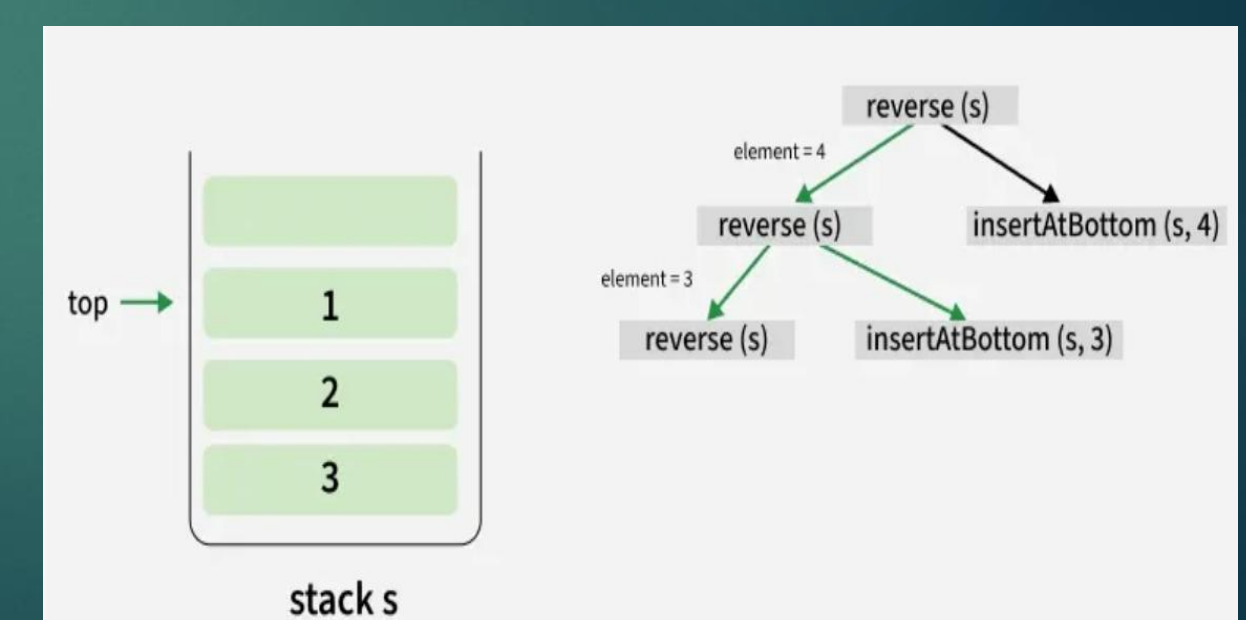
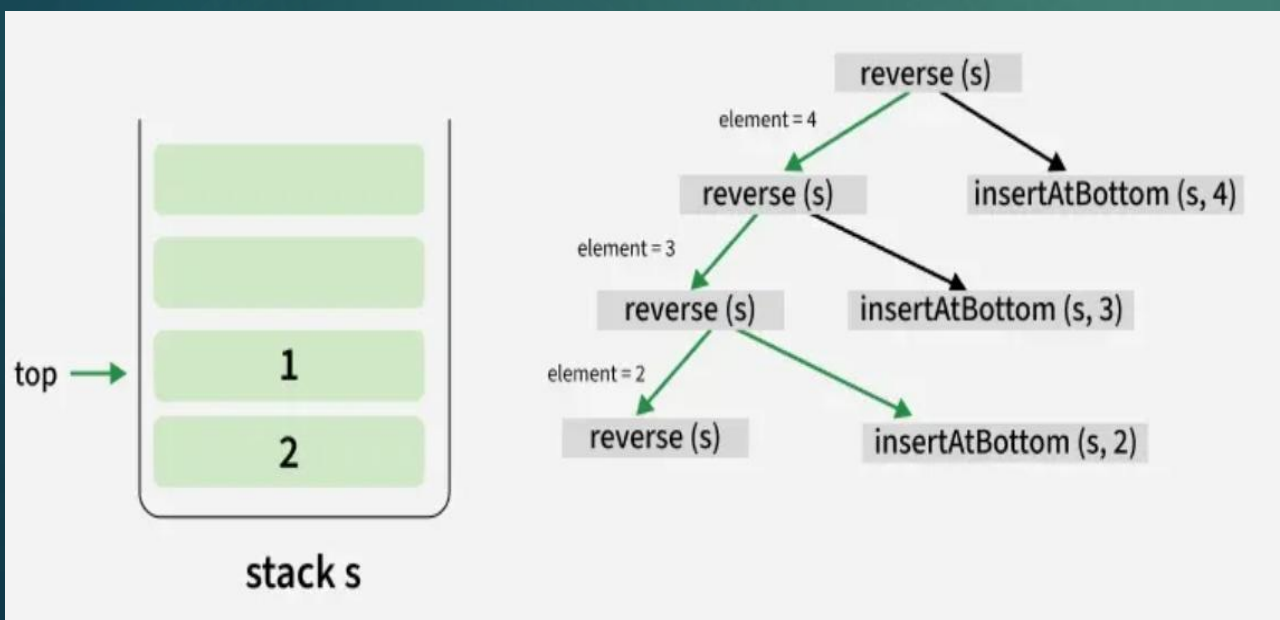
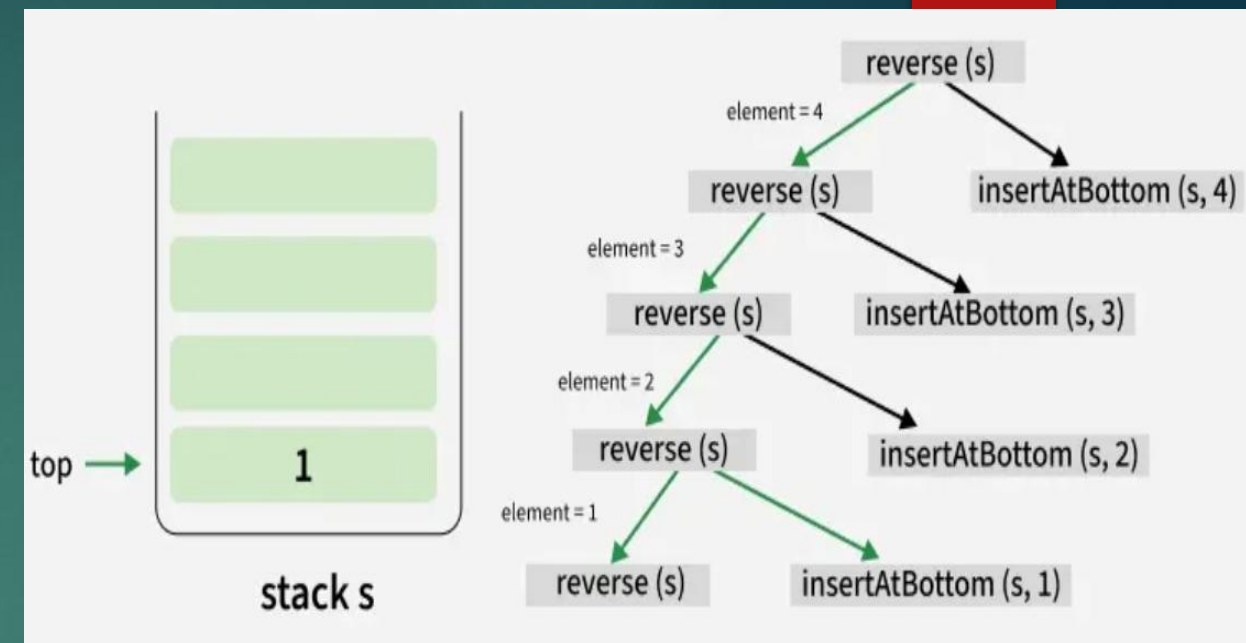
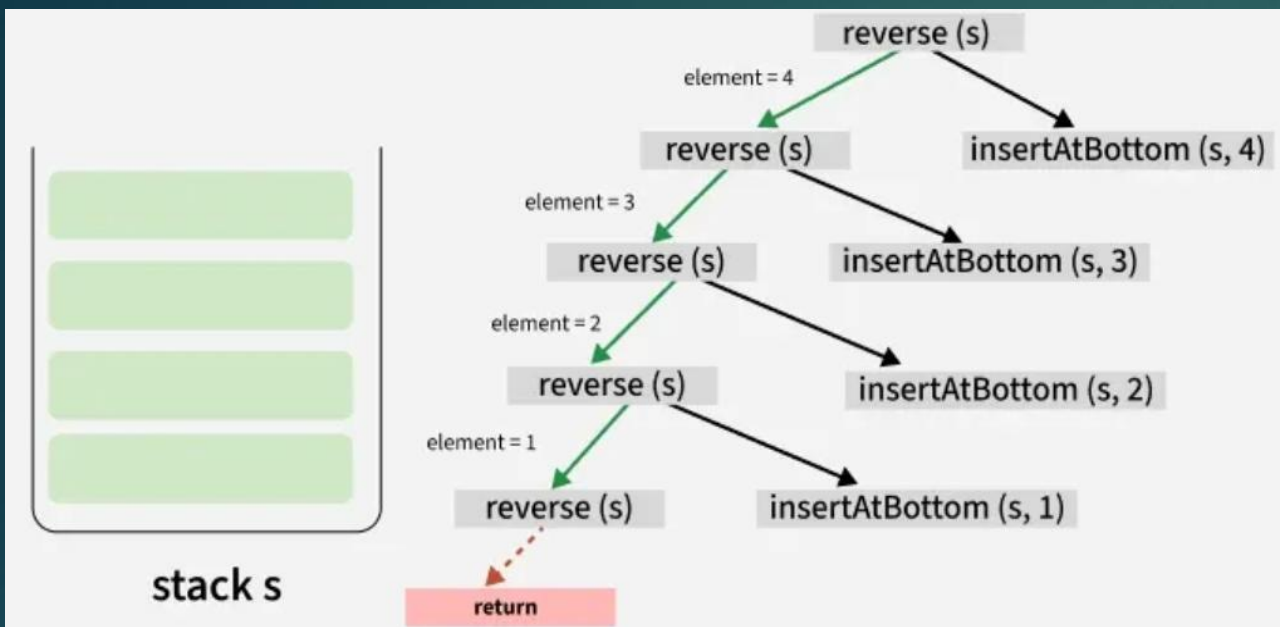


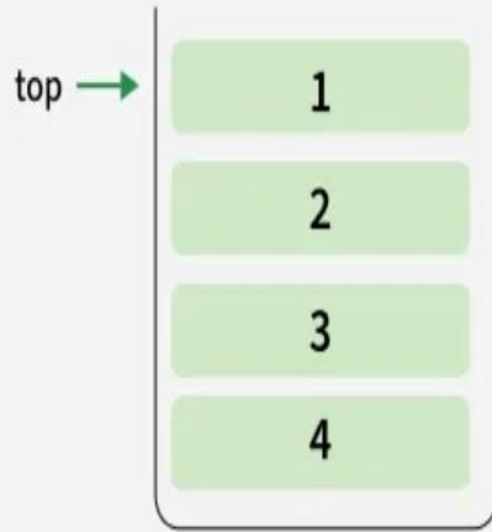
stack  $s$



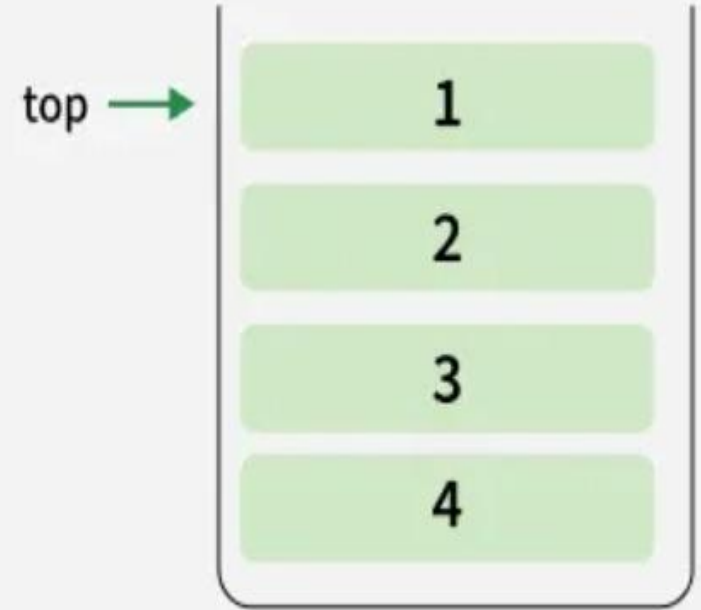
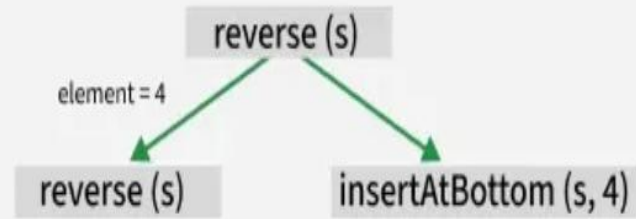
stack  $s$







stack s



Final stack s



```
import java.util.Stack;

public class StackRecurtion {
    public static void insertAtBottom(Stack<Integer> s, int x) {
        if (s.isEmpty()) {
            s.push(x);
        } else {
            int a = s.pop();
            insertAtBottom(s, x);
            s.push(a); } }

    public static void reverse(Stack<Integer> s) {
        if (!s.isEmpty()) {
            int x = s.pop();
            reverse(s);
            insertAtBottom(s, x);
        }
    }
}
```



```
public static void main(String[] args) {  
    Stack<Integer> s = new Stack<>();  
  
    // Pushing elements to the stack: 4 3 2 1 (bottom to top)  
    s.push(1);  
    s.push(2);  
    s.push(3);  
    s.push(4);  
  
    // Reversing the stack  
    reverse(s);  
  
    while (!s.isEmpty()) {  
        System.out.print(s.pop() + " ");  
    }  
}
```

# Sort a Stack using Recursion

```
import java.util.Stack;

public class Recursion {
    public static void sortedInsert(Stack<Integer> s, int x) {
        if (s.isEmpty() || x > s.peek()) {
            s.push(x);
            return;
        }
        int temp = s.pop();
        sortedInsert(s, x);
        s.push(temp);
    }

    public static void sort(Stack<Integer> s) {
        if (!s.isEmpty()) {
            int x = s.pop();
            sort(s);
            sortedInsert(s, x);
        }
    }
}
```

```
public static void main(String[] args) {  
    Stack<Integer> s = new Stack<>();  
    s.push(11);  
    s.push(2);  
    s.push(32);  
    s.push(3);  
    s.push(41);  
    sort(s);  
    while (!s.isEmpty()) {  
        System.out.print(s.pop() + " ");  
    }  
}
```