❖ Stack

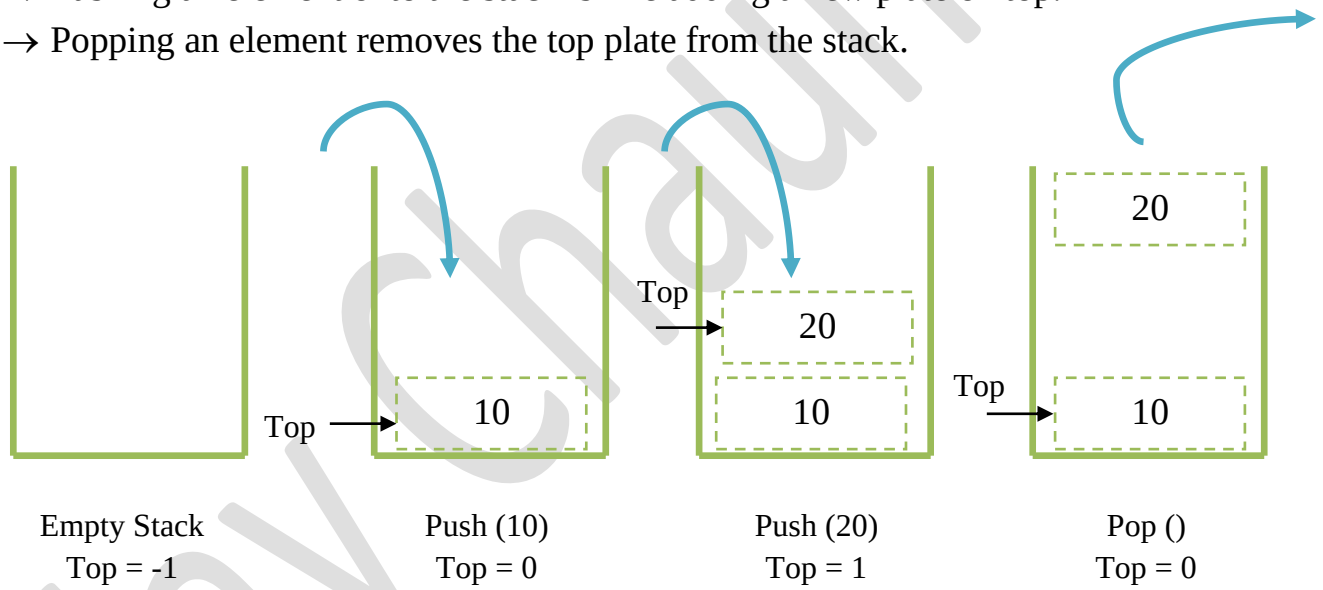
→ A Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.

→ It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

→ It behaves like a stack of plates, where the last plate added is the first one to be removed. Think of it this way:

→ Pushing an element onto the stack is like adding a new plate on top.

→ Popping an element removes the top plate from the stack.



➢ **Some key points related to stack**

→ It is called as stack because it behaves like a real-world stack, piles of books, etc.

→ A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.

→ It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

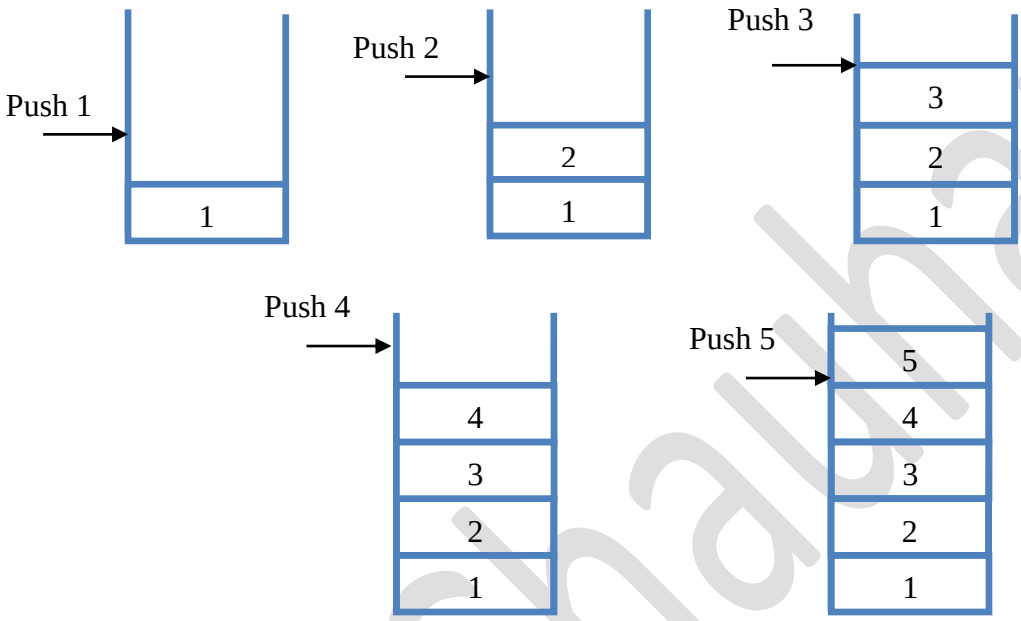
→ Elements in a stack follow a vertically linear arrangement.

Jay Chauhan

➢ Working of Stack

→ Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

→ Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



→ Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

→ When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered Last, so it will be removed First.

- Stacks adhere strictly to the Last-In-First-Out (LIFO) principle for insertion and deletion of elements.
- Envision a vertical stack with the top portion being the active end for additions and removals.
- Initially, when a stack is created, it is marked as empty with the top pointer set to a sent initial value like -1
- During insertion (push operation), the element is placed at the current top position after incrementing the top pointer.

Jay Chauhan

- This push process continues until the stack's predetermined capacity is reached, after which insertions aren't possible.
- For removal (pop operation), the element at the current top is retrieved and returned.
- Simultaneously, the top pointer is decremented, removing that topmost element from the stack.
- A pop attempt on an empty stack triggers an underflow condition, indicating an erroneous operation.
- The stack grows upwards during pushes and shrinks downwards during pops, maintaining the LIFO order.

❖ Stack Operations

→ The following operations can be performed on a stack:

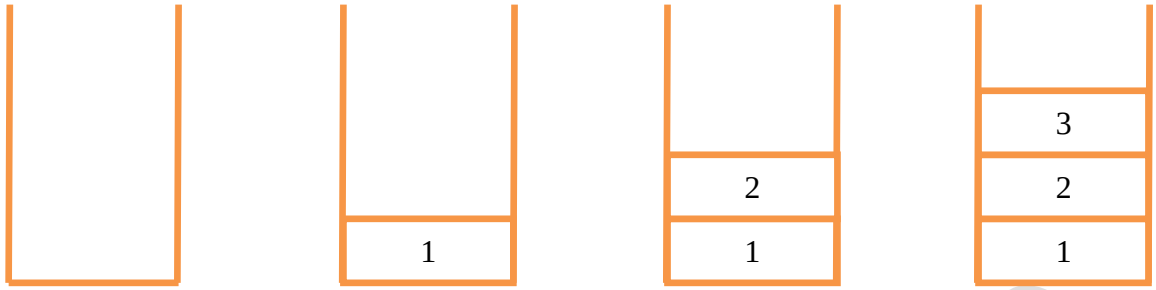
- **push()** - Inserts an element into the stack. If the stack is full, an overflow condition occurs.
- **pop()** - Removes an element from the stack. If the stack is empty, an underflow condition occurs.
- **peep()** - Retrieves the element at the ith position from the **top** of the stack.
- **change()** - Modifies the element at the ith position from the **top** of the stack.
- **display()** - Prints all elements in the stack.

✓ **Key Note:** All operations in the stack are performed from the **TOP** of the stack.

❖ PUSH Operation

→ Before inserting an element into a stack, we must check whether the stack is full.
→ If the stack is full, inserting an element will cause an overflow condition.
→ Initially, when a stack is created, the top is set to -1, indicating that the stack is empty.
→ When pushing a new element:
→ First, increment the top pointer (top = top + 1).
→ Then, insert the new element at this updated top position.
→ Elements are inserted sequentially until the stack reaches its maximum size.

Jay Chauhan



Empty Stack
Top = -1

Push (1)
Top = 0

Push (2)
Top = 1

Push (3)
Top = 2

➢ PUSH Algorithm

Procedure PUSH (S, TOP, X)

→ This procedure inserts an element X to the top of a stack which is represented by a vector S containing N elements with a pointer TOP denoting the top element in the stack.

STEP 1: [Check for stack overflow?]
If TOP >= N
then Write ("STACK Overflow")
Return

STEP 2: [Increment TOP]
TOP ← TOP + 1

STEP 3: [Insert Element]
S[TOP] ← X

STEP 4: [Finished]
Return

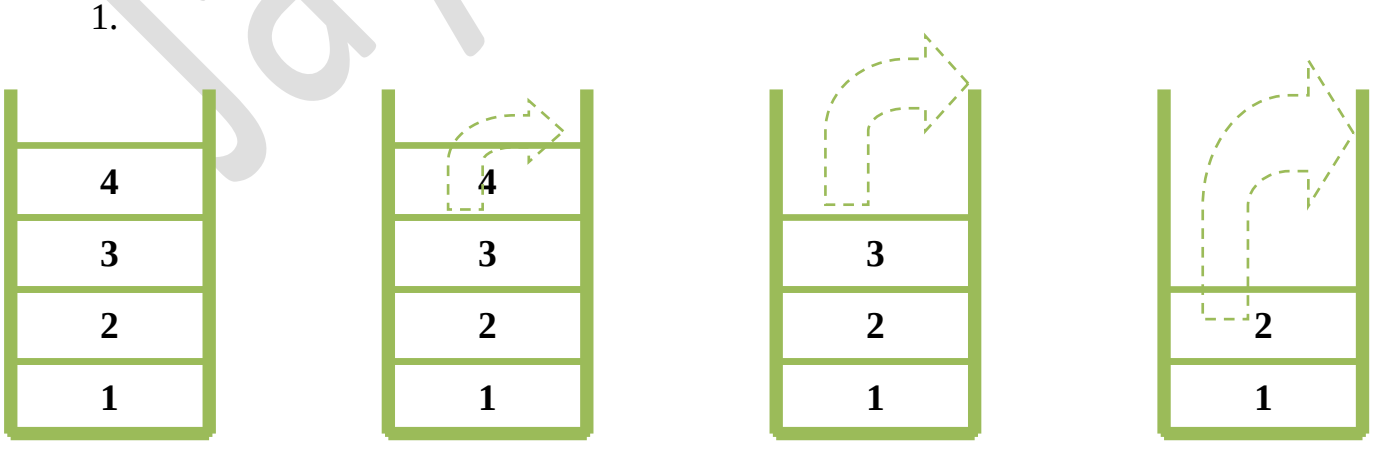
Jay Chauhan

Push Function :

```
void push(int x)
{
   if (top >= n -1 )
   {
      System.out.println("overflow");
   }
   else
   {
      top++;
      a[top] = x;
      System.out.println("Element entered successfully");
   }
}
```

❖ POP Operation

- Before deleting the element from the stack, we need to check whether the stack is empty or not.
- If we try to delete the element from the empty stack, then the underflow condition occurs.
- If the stack is not empty, we first access the element which is pointed by the top.
- Once the pop operation is performed, the top is decremented by 1, i.e., top = top - 1.



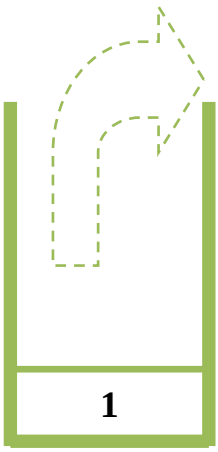
Stack is Full
Top = 3

Pop 4
Top = 2

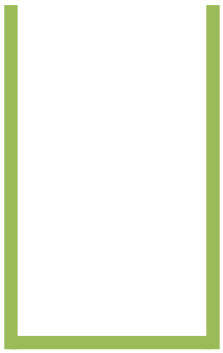
Pop 3
Top = 1

Pop 2
Top = 0

Jay Chauhan



Pop 1
Top = -1



Empty Stack
Top = -1

- **POP Algorithm**

Function POP(S, TOP)

→This method removes the top element from the stack which is represented by a vector S and returns this element. TOP is a pointer to the top element of the stack.

Step 1: [Check for the Underflow on stack]
If TOP = 0
then Write ("STACK UNDERFLOW ON POP")
take action in response to underflow
Exit

Step 2: [Decrement top Pointer]
TOP ← TOP - 1

Step 3: [Returns pointer top element of stack]
Return(S[TOP + 1])

Jay Chauhan

POP Function :

```
void pop()
{
      if (top == -1)
        {
         System.out.println("Underflow");
       }
        else
        {
         System.out.println("Popped element: " + a[top]);
         top--;
      }
   }
```

❖ Peep Operation

- **Check if the stack is empty**
  - If the stack is empty, an **underflow condition** occurs, meaning there are no elements to peek.
- **Access the element at a given position**
  - The element is accessed without removing it from the stack.
- **Formula to get the element at position pos from the top:**

element=stack[top−I+1]

- top refers to the index of the top element.
- I is the position from the top (1-based index).

➢ PEEP Algorithm

Function PEEP(S, TOP, I)

→ In this method the S is a vector (consisting the N elements) representing a sequential allocated stack, and a pointer TOP denoting the top element of the Stack, this method returns the value of the Ith element from the TOP of the stack. The element is not deleted by this method.

Jay Chauhan

Step 1: [Check for the Stack Underflow]
If TOP-I+1 ≤ 0
then Write("Stack Underflow On Peep")
take action in response to underflow
Exit

Step 2: [Return Ith Element from the top of the Stack]
Return(S [TOP - I + 1])

PEEP Function :

```
void peep(int I)
{
   if (top - I + 1 <=-1)
       {
       System.out.println("Underflow");
       }
else
    {
       System.out.println("Element at position " + i + " from top: " + a[top - i + 1]);
    }
}
```

❖ Change Operation

→ The **change operation** in a stack is used to modify an element at a specific position from the top without affecting other elements.

- It does **not remove or add** elements, only updates an existing value.
- The position is counted from the **top of the stack** (1-based index).
- If the given position is **invalid** (out of range), an **underflow condition** occurs.

Formula for Accessing the Position:

stack[top−I+1]=X

Where:

- top is the index of the **top element**.
- I is the **position from the top**.
- X is the **updated value**.

Jay Chauhan

➢ Change Algorithm

Procedure CHANGE(S, TOP, I, X)

→In this procedure the vector 'S' represents a sequentially allocated stack and a pointer TOP denotes the top element of the stack. This procedure changes the value of the I^{th} element from the top of the stack to the value contained in X.

Step 1: [Check for the Stack Underflow]
If TOP-I+1 ≤ 0
then Write("Stack Underflow On CHANGE")
Return

Step 2: [Change Ith Element from the top of the Stack]
S [TOP – I + 1] ← X

Step 3: [Finished]
Return

CHNAGE Function :

```
void change(int i, int y)
{
      if (top - i + 1 <= -1)
      {
         System.out.println("Invalid position");
      }
      else {
         a[top - i + 1] = y;
         System.out.println("Element at position " + i + " changed to " + y);
      }
   }
```

DISPLAY Function :

```
void display() {
   if (top == -1) {
      System.out.println("Stack is empty");
   } else {
      for (int i = top; i >= 0; i--) {
         System.out.println(a[i]);
```

Jay Chauhan

```
    }
  }
}
```

Write a Java program to implement a stack using an array with the following operations:

1. **Push: Insert an element into the stack.**
2. **Pop: Remove the top element from the stack and display it.**
3. **Peep: Display an element at a given position from the top of the stack.**
4. **Change: Modify an element at a given position from the top of the stack.**
5. **Display: Show all elements of the stack.**

Constraints:

- **The program should accept the stack size from the user.**
- **The stack should handle overflow and underflow conditions.**
- **The peep and change operations should validate the given position.**

```
import java.util.Scanner;

 class stack1
 {
   int top = -1, N;
   int[] a;

   stack1()
   {
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter size:");
      N = sc.nextInt();
      a = new int[N];
   }

   void push(int x)
   {
      if (top == N - 1)
      {
```

Jay Chauhan

```
        System.out.println("Overflow");
    } else
    {
        top++;
        a[top] = x;
        System.out.println("Element entered");
    }
}

void pop()
{
    if (top == -1)
    {
        System.out.println("Underflow");
    } else
    {
        System.out.println("Popped element: " + a[top]);
        top--;
    }
}

void peep(int i)
{
    if (top - i + 1 <= -1)
    {
        System.out.println("Invalid position");
    } else
    {
      System.out.println("Element at position " + i + " from top: " + a[top - i + 1]);
    }
}

void change(int i, int y)
{
    if (top - i + 1 <= -1)
    {
```

Jay Chauhan

```
            System.out.println("Invalid position");
        } else
        {
            a[top - i + 1] = y;
            System.out.println("Element at position " + i + " changed to " + y);
        }
    }

    void display()
    {
        if (top == -1)
        {
            System.out.println("Stack is empty");
        } else
        {
            System.out.println("Stack elements:");
            for (int i = top; i >= 0; i--)
            {
                System.out.println(a[i]);
            }
        }
    }
}

class Main
{
    public static void main(String[] args)
    {
        stack1 s1 = new stack1();
        s1.push(10);
        s1.push(20);
        s1.push(30);
        s1.pop();
        s1.peep(3);
        s1.display();
        s1.change(2, 50); // Change the 2nd element from top to 50
```

Jay Chauhan

```
    s1.display();
  }
}
```

❖ Application of Stack

- **Balancing of Symbols**
- **String Reversal**
- **Undo/Redo Operations**
- **Recursion**
- **Depth First Search (DFS)**
- **Backtracking**
- **Expression Conversion**
- **Memory Management**

1. Balancing of Symbols

- Stack is used to check whether an expression has balanced parentheses {}, (), [].
- Each opening symbol is pushed onto the stack, and when a closing symbol appears, the stack is popped.
- If the stack is empty at the end, the expression is balanced; otherwise, it is not.

2. String Reversal

- A stack can be used to reverse a string.
- Each character of the string is pushed onto the stack.
- Since a stack follows the Last In, First Out (LIFO) principle, popping the characters results in the reversed string.

3. Undo/Redo

- Many applications, such as text editors, use stacks to implement the undo and redo functionality.
- When an operation is performed, it is pushed onto the stack.
- Undo operations pop the last action, while redo operations push it back.

4. Recursion

- Function calls in recursion are stored in a stack.
- Each recursive function call is pushed onto the stack until the base case is reached.
- Then, function calls are popped in reverse order.

Jay Chauhan

5. Depth First Search (DFS)

- In graph traversal, DFS uses a stack to explore nodes.
- A node is pushed onto the stack, and its adjacent nodes are explored by popping and pushing new nodes.

6. Backtracking

- Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

7. Expression Conversion

- Stack is used to convert infix expressions (e.g., A + B) to postfix (AB+) or prefix (+AB).
- Operators are pushed onto the stack and popped according to precedence rules.

8. Memory Management

- Stack memory is used to store function calls and local variables.
- Each function call pushes a new stack frame, and when the function completes, the stack frame is popped.

❖ Recursion

- Recursion is the process which comes into existence when a method calls a copy of itself to work on a smaller problem. Any method which calls itself is called a recursive method, and such method calls are called recursive calls.
- Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code; however, it is difficult to understand.
- Recursion cannot be applied to all the problems, but it is more useful for the tasks that can be defined in terms of similar subtasks. For example, recursion may be applied to sorting, searching, and traversal problems.
- Generally, iterative solutions are more efficient than recursion since method call is always overhead. Any problem that can be solved recursively can also be solved iteratively. However, some problems are best suited to be solved by recursion, for example, Tower of Hanoi, Fibonacci series, factorial finding, etc.

Jay Chauhan

Difference Between Recursion and Iteration: -

Property	Recursion	Iteration
Definition	Function calls itself.	A set of instructions repeatedly executed.
Application	For functions.	For loops.
Termination	Through base case, where there will be no function call.	When the termination condition for the iterator ceases to be satisfied.
Usage	Used when code size needs to be small, and time complexity is not an issue.	Used when time complexity needs to be balanced against an expanded code size.
Code Size	Smaller code size	Larger Code Size.
Time Complexity	Very high(generally exponential) time complexity.	Relatively lower time complexity(generally polynomial-logarithmic).
Space Complexity	The space complexity is higher than iterations.	Space complexity is lower.
Stack	Here the stack is used to store local variables when the function is called.	Stack is not used.
Speed	Execution is slow since it has the overhead of maintaining and updating the stack.	Normally, it is faster than recursion as it doesn't utilize the stack.
Memory	Recursion uses more memory as compared to iteration.	Iteration uses less memory as compared to recursion.

Example : Factorial of a Number

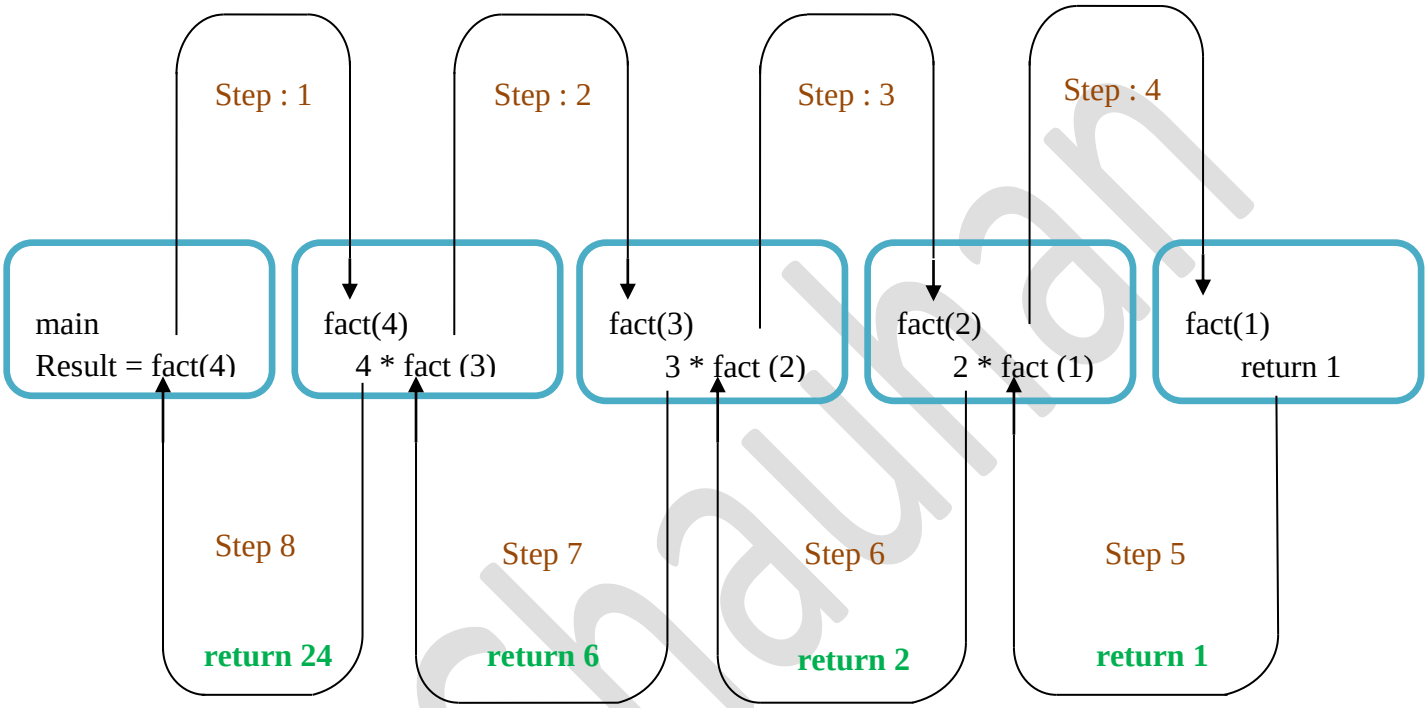
```
class Fact
{
   int fact(int n)
        {
      // BASE CONDITION
      if (n == 0)
         return 1;

      return n * fact(n - 1);
   }

   public static void main(String[] args) {
```

Jay Chauhan

```
    System.out.println("Result : " + fact(4));
  }
}
```



Recursive Method

→ A recursive method performs the tasks by dividing it into the subtasks. There is a termination condition defined in the method which is satisfied by some specific subtask. After this, the recursion stops and the final result is returned from the method.

→ The case at which the method doesn't recur is called the base case, whereas the instances where the method keeps calling itself to perform a subtask is called the recursive case. All the recursive methods can be written using this format.

➢ Advantages and Disadvantages of Recursion

- ***Advantages of Recursion***

1. The code may be easier to write.
2. Reduces unnecessary calling of methods.
3. Extremely useful when applying the same solution.

Jay Chauhan

4. Recursion reduces the length of the code.
5. It is very useful in solving data structure problems.
6. Stacks evolutions and infix, prefix, postfix evaluations, etc.

- ***Disadvantages of Recursion***

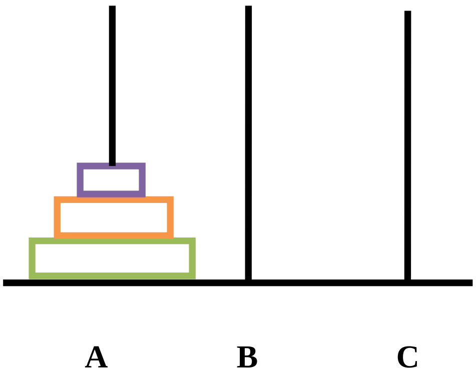
1. Recursive methods are generally slower than non-recursive methods.
2. It may require a lot of memory space to hold intermediate results on the system stack.
3. Hard to analyze or understand the code.
4. It is not more efficient in terms of space and time complexity.
5. The computer may run out of memory if the recursive calls are not properly checked.

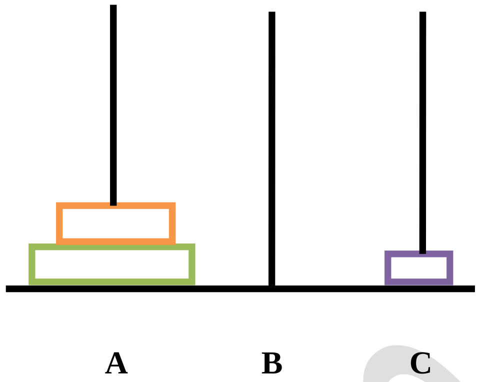
❖ Tower of Hanoi problem using recursion

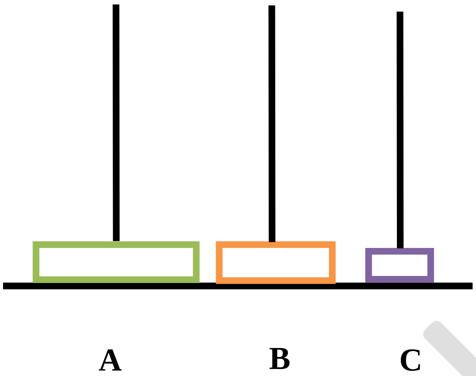
→ Tower of Hanoi is a mathematical puzzle where we have three rods (A, B, and C) and N disks. Initially, all the disks are stacked in decreasing value of diameter i.e., the smallest disk is placed on the top and they are on rod A. The objective of the puzzle is to move the entire stack to another rod (here considered C), obeying the following simple rules:

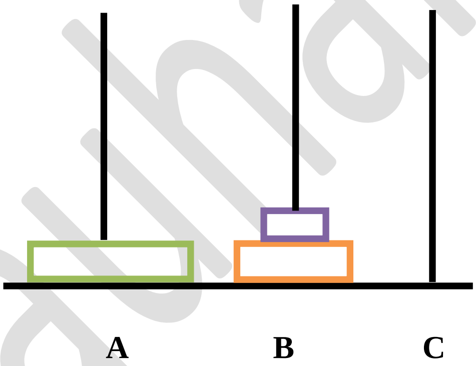
- **Only one disk can be moved at a time.**
- **Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.**
- **No disk may be placed on top of a smaller disk.**

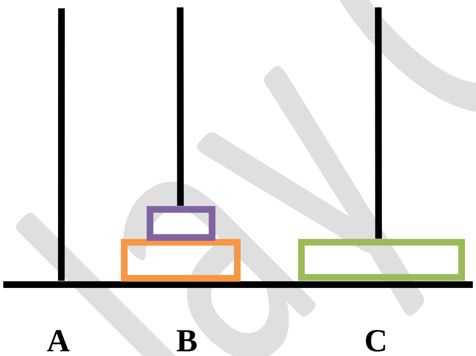
Jay Chauhan

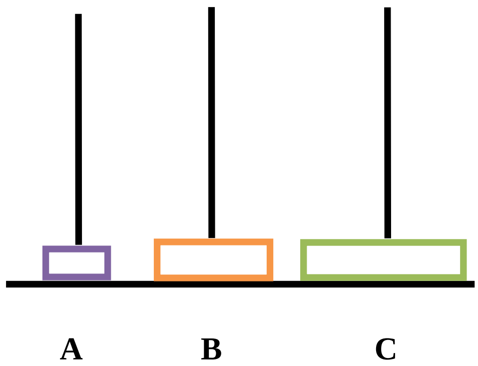


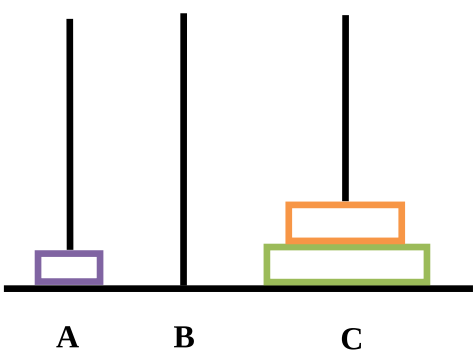


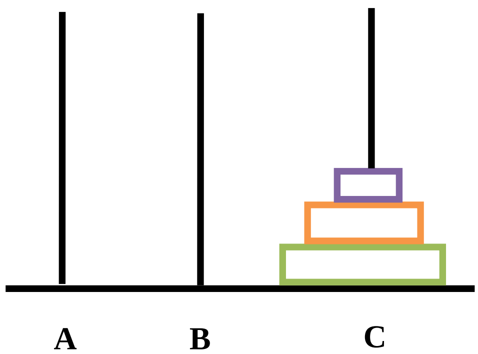












Jay Chauhan

JAVA PROGRAM FOR TOWER HANOI

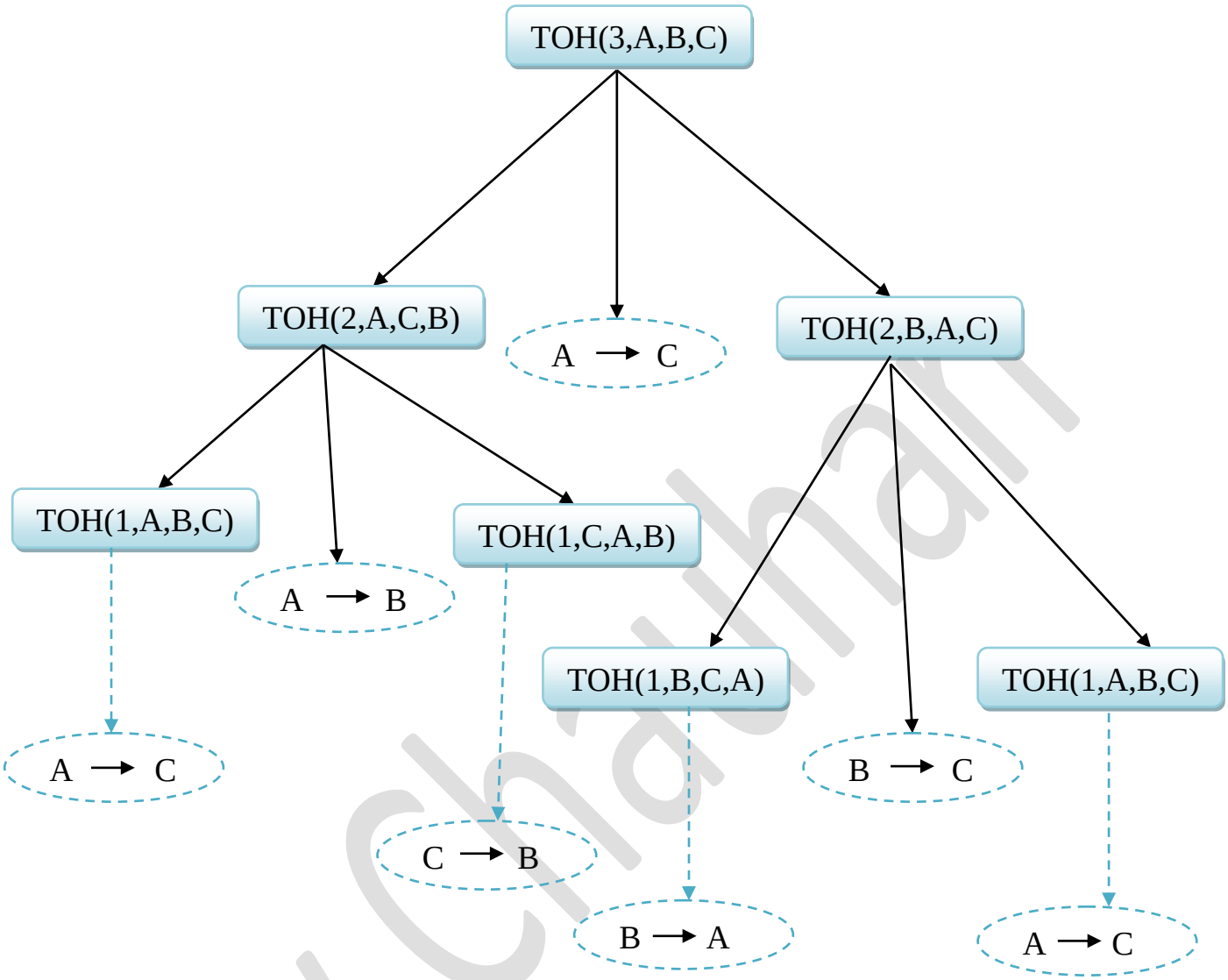
```
import java.util.Scanner;

class TOH {
   static void towerOfHanoi(int n, char A, char B, char C) {
      if (n == 0) {
         return;
      }
      towerOfHanoi(n - 1, A, C, B);  // Move (n-1) disks from A to B using C
      System.out.println("Move disk " + n + " from " + A + " to " + C);
      towerOfHanoi(n - 1, B, A, C);  // Move (n-1) disks from B to C using A
   }

   public static void main(String args[]) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter number of disks: ");
      int N = sc.nextInt();
      towerOfHanoi(N, 'A', 'B', 'C');  // Source = A, Auxiliary = B, Destination = C
   }

}
```

Jay Chauhan



❖ String Reversal Using Stack

Write an algorithm to reverse a string of characters using stack

Step 1: [Initialize Stack]
Top ← 1
S[TOP] ← '#'
Step 2: [Initialize Reverse]
Reverse ← ""
Step 3: [Get Next char]
NEXT ← NEXT CHAR (INPUT)
Step 4: [Insert into Stack & get next char]

Jay Chauhan

Repeat while NEXT ≠ '#'
Call PUSH(S,TOP,NEXT)
NEXT ← NEXT CHAR (INPUT)

Step 5: [Remove element from Stack & get the answer]

Repeat while S[TOP] ≠ '#'
TEMP ← POP(S,TOP)
REVERSE ← REVERSE ° TEMP

Step 6: [Print the answer]

WRITE (REVERSE)
RETURN

Example:

- String = DATA
- Append # at the end → DATA#
- Stack representation:

A
T
A
D

- Reversed output: ATAD

Jay Chauhan

JAVA PROGRAM OF STRING REVERSAL USING STACK

```
import java.util.Scanner;

class Stack
{
   int top = -1, i;
   char[] s;
   int n = 0;
   String rev = "";

   Stack(int a)
   {
      n = a;
      s = new char[a];
   }

   void push(char d)
   {
      top++;
      s[top] = d;
   }

   void pop()
   {
      top --;
      rev = rev + s[top+1];
   }
}

class Run
{
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter String: ");
      String str = sc.nextLine();
      Stack s1 = new Stack(str.length());
```

Jay Chauhan

```
    for(int i = 0; i< str.length(); i++)
    {
       s1.push(str.charAt(i));
    }

    for(int i = 0; i<str.length(); i++)
    {
       s1.pop();
    }
    System.out.println(s1.rev);
  }
}
```

Jay Chauhan

❖ Find Binary of The Given Decimal Number Using Stack

Write an algorithm to convert binary number to decimal number using stack.

Step 1: [Initialize Stack]

Top ← 1
S[TOP] ← '#'
ANS ← ""

Step 2: [Enter Element in Next]

NUM ← NEWDATA

Step 3: [Apply Division & add reminder in stack till until last step]

Repeat while NUM ≠ 0

REM ← NUM % 2

NUM ← NUM / 2

Call PUSH(S, TOP, REM)

Step 4: [Pop elements from Stack & concatenate them for answer]

Repeat while S[TOP] ≠ '#'

TEMP ← POP(S, TOP)

ANS ← ANS ° TEMP

Step 5: [Print the Answer]

- WRITE (ANS)

Jay Chauhan

JAVA PROGRAM TO FIND BINARY OF THE GIVEN DECIMAL NUMBER USING STACK

```
import java.util.Scanner;

class Stack
{
   int top = -1, n;
   int[] s;
   String bin = "";

   Stack(int a) {
      n = a;
      s = new int[a];
   }

   void push(int d) {
      top++;
      s[top] = d;
   }

   void pop() {
      bin = bin + s[top];
      top--;
   }
}

class DTOB {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter a decimal number: ");
      int num = sc.nextInt();

      int count = 0, temp = num;
      while (temp > 0) {  // Count the number of bits required
         temp = temp / 2;
         count++;
      }
```

Jay Chauhan

```
    Stack S1 = new Stack(count);

    while (num > 0) {  // Push binary digits to stack
       S1.push(num % 2);
       num = num / 2;
    }

    while (S1.top != -1) {  // Pop and construct the binary string
       S1.pop();
    }

    System.out.println("Binary Equivalent: " + S1.bin);
  }
}
```

Jay Chauhan