Chapter 2

Stack and Recursion

Stack:

- Stack is a linear data structure in which an element may be inserted or deleted only at one end. i.e. The elements are removed from a stack in the reverse order of that in which they were inserted into the stack.
- Stack uses a variable called **top** which points topmost element in the stack.
- Top is incremented while pushing(inserting) an element into the stack and decremented while popping(deleting) an element from the stack.
- A stack follows the principle of Last-In-First-Out(LIFO) system.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- **PUSH** and **POP** are two terms used for insertion and deletion operation in stack respectively.

Example;

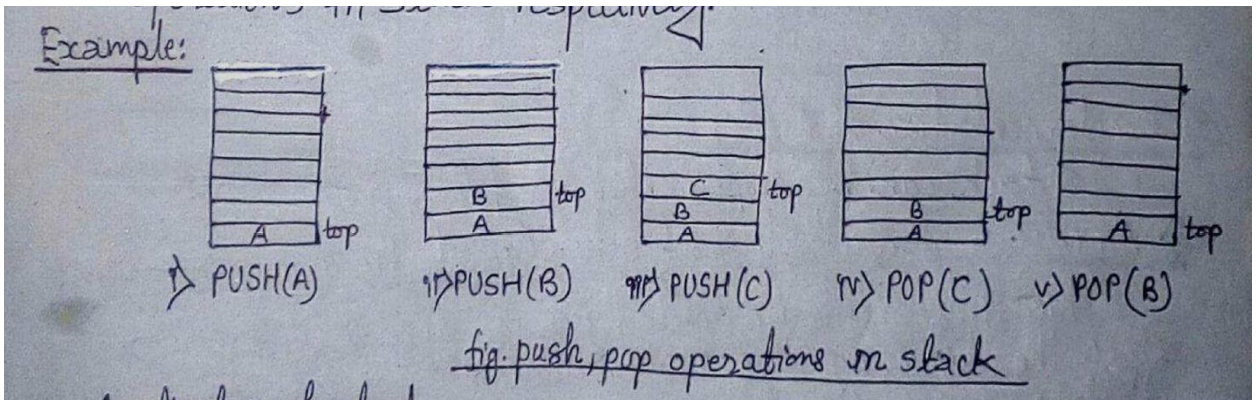


fig: push, pop operations on stack

Application of Stack:

- To evaluate the postfix and prefix expression.
- To keep the page-visited history in web browser.
- To perform the undo sequence in a text editor.
- Used in recursion.
- To memory management.

Stack ADT:

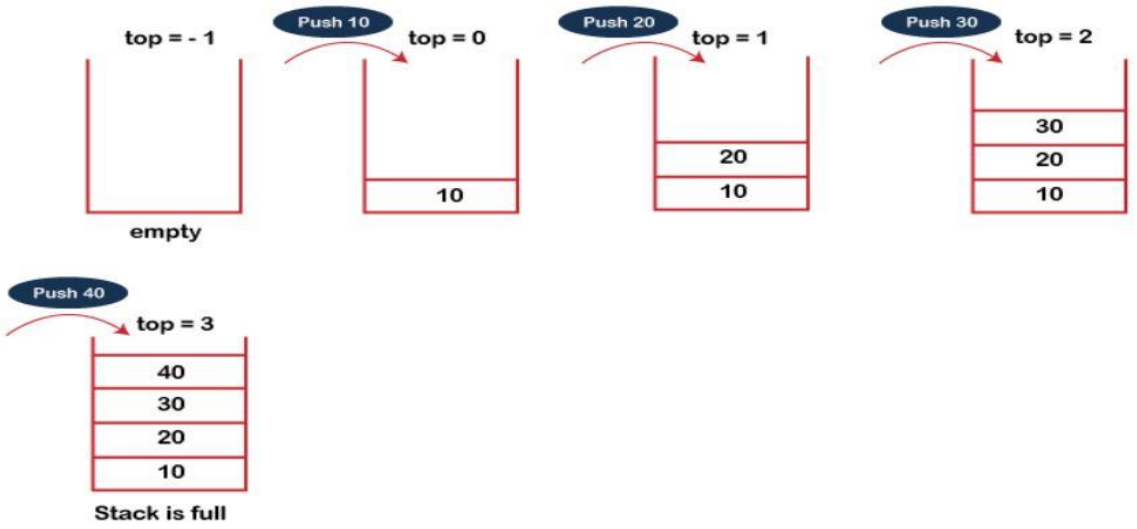
In stack ADT(Abstract Data Type) we discuss about the operations that can be performed on stack.

- **Create Empty Stack(S):** Create stack S which is initially an empty stack.
- **push(S,x):** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop(S,x):** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **top(S):** If stack is not empty, then retrieve the element at the top.
- **isEmpty(S):** It determines whether the stack is empty or not.
- **isFull(S):** It determines whether the stack is full or not.
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

Implementation of Stack:

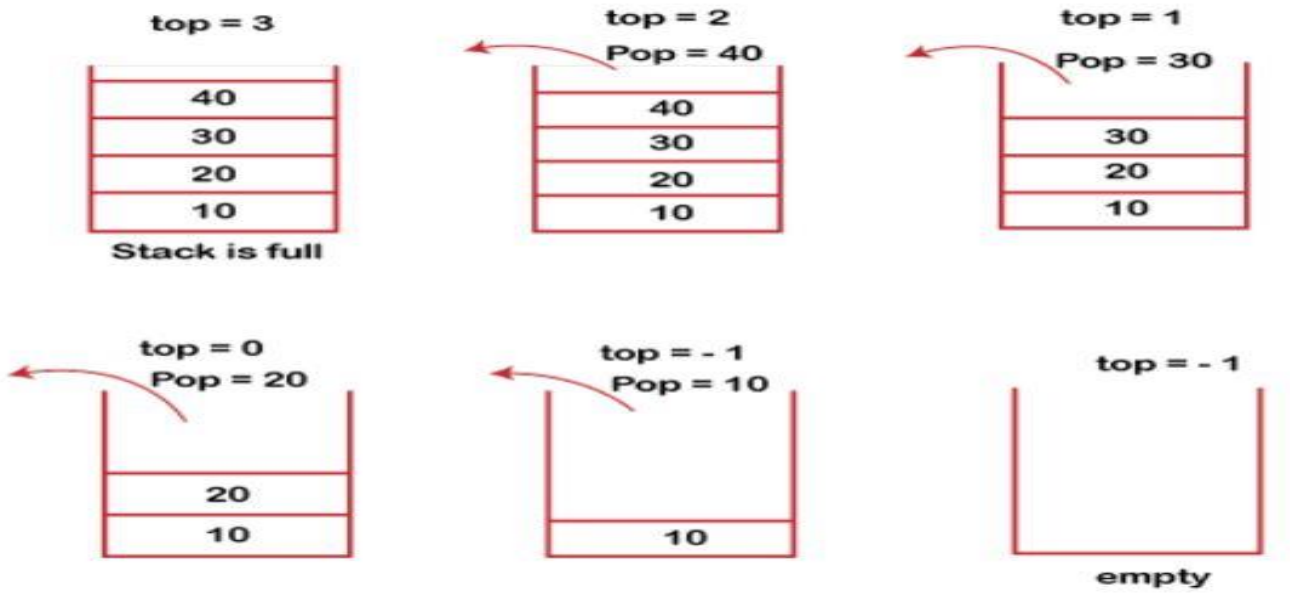
PUSH operation

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.



POP operation:

- Before deleting the element from the stack, we check whether the stack is empty.
- 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**.



Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Algorithm for push

PUSH:

Let stack [Max size] is an array for implementing the stack
1. [check for stack overflow?]
If top = maxsize-1, then print overflow & exit
2. set top = top + 1 (increase top by 1)
3. set stack [top] = item (inserts item in new top position)
4. exit.

Implementation of push algorithm in C language

```
void push(int val, int n) // n is the size of the stack
{
   if (top == n - 1)  // Check if the stack is full
   {
      printf("\nStack Overflow\n");
      return;  // Exit the function
   }
   else
   {
      top = top + 1;        // Increment the top pointer
      stack[top] = val;     // Add the new value to the stack
   }
}
```

C program for stack push implementation

```
#include <stdio.h>
#define MAX 5 // Define the maximum size of the stack

int stack[MAX]; // Array to hold stack elements
int top = -1;   // Initialize top of stack to -1

// Function to display the elements of the stack
void display() {
   if (top == -1) { // Check if the stack is empty
      printf("\nThe stack is empty.\n");
   } else {
```

```
        printf("\nCurrent stack elements are:\n");
        for (int i = top; i >= 0; i--) { // Display elements from top to bottom
            printf("%d\n", stack[i]);
        }
    }
}

// Function to push an element onto the stack
void push(int val) {
    if (top == MAX - 1) { // Check if the stack is full
        printf("\nStack Overflow! Cannot push %d onto the stack.\n", val);
    } else {
        top = top + 1;     // Increment the top pointer
        stack[top] = val;  // Add the new value to the stack
        printf("\n%d pushed onto the stack.\n", val);
        display(); // Automatically display the stack after pushing
    }
}

int main() {
    int value;

    printf("\nEnter values to push into the stack (Enter -1 to stop):\n");

    while (1) { // Infinite loop to take multiple inputs
        printf("\nEnter a value: ");
        scanf("%d", &value);

        if (value == -1) { // Exit condition
            printf("\nExiting program.\n");
            break;
        }

        push(value); // Push the value onto the stack
    }

    return 0;
}
```

Deletion of an element from a stack (Pop operation)

POP:-

1. check for the stack underflow

If Top <0 then

Print stack underflow and exit
Else
[Remove the top element]
Set item = stack [top]
2. Decrement the stack top.
3. Return the deleted item from the stack.
4. Exit.

Implementation of POP algorithm using C language

```
int pop()
{
   if (top == -1) // Check if the stack is empty
   {
      printf("Underflow\n");
      return -1; // Return a special error code
   }
   else
   {
      int item = stack[top]; // Access the top element
      top = top - 1;        // Decrement the top pointer
      return item;          // Return the popped item
   }
}
```

Visiting each element of the stack (Peek operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Implementation of Peek algorithm in C language

```
int peek()

{

   if (top == -1)  // Check if the stack is empty

   {

      printf("Stack Underflow\n");
```

```
        return -1; // Return a special error code
    }
    else
    {
        return stack[top]; // Return the top element without modifying the stack
    }
}
```

Expression evaluation

Expression evaluation is the process of interpreting and computing the value of an expression based on its syntax and the operations it specifies. An **expression** is a combination of operands (such as numbers, variables, or constants) and operators (like +, -, *, /, etc.) that defines a specific computation or logic.

- ***Infix Notation:*** Operators are written between the operands they operate on, E.g. A + B.
- ***Prefix Notation:*** Operators are written before the operands, E.g +AB
- ***Postfix Notation:*** Operators are written after operands. E.g, AB+

Operands: A,B,X, Y…
Operators: +,-,*,/,(,)

Operator Precedence

Operator	Precedence	Associativity
^	High	Right
*, /	Medium	Left
+, –	Low	Left

Expression Conversion:

1. **Infix to Postfix:**

Steps for Converting Infix to Postfix using a Stack:

1. **Initialize an empty stack** and an empty string (or list) for the postfix expression.
2. **Scan the infix expression from left to right**:
    - If the character is an **operand** (a number or variable like A, B, etc.), add it to the postfix expression.
    - If the character is an **operator** (like +, -, *, /, etc.), pop from the stack to the postfix expression until the top of the stack contains an operator of lower precedence or the stack is empty, then push the current operator onto the stack.
    - If the character is an **opening parenthesis (()**, push it onto the stack.
    - If the character is a **closing parenthesis ())**, pop from the stack to the postfix expression until you encounter an opening parenthesis (). Discard the pair of parentheses.
3. **After the entire infix expression has been scanned**, pop all the operators from the stack to the postfix expression.
4. **Return the postfix expression**.

1. Infix: A + B * C

- **Steps**:
    1. * has higher precedence than +, so evaluate B * C first.
    2. Then add A.
- **Postfix**: ABC*+

---

2. Infix: (A - B) * (C + D)

- **Steps**:
    1. Parentheses indicate precedence, so evaluate (A - B) and (C + D) first.
    2. Multiply the results.
- **Postfix**: AB-CD+*

---

3. Infix: A / (B + C)

- **Steps**:
    1. Parentheses indicate precedence, so evaluate (B + C) first.
    2. Divide A by the result.
- **Postfix**: ABC+/

---

4. Infix: A + B * (C - D ^ E)

- **Steps**:
    1. ^ has the highest precedence, so evaluate D ^ E first.
    2. Subtract D ^ E from C.
    3. Multiply the result by B.
    4. Add A to the result.
- **Postfix**: ABCDE^-*+

---

5. Infix: ((A + B) * C) / D

- **Steps**:
    1. Parentheses indicate precedence, so evaluate (A + B) first.
    2. Multiply the result by C.
    3. Divide the result by D.
- **Postfix**: AB+C*D/

Assignment:

- A * B + C / D – E
- (X + Y * Z) / (M - N)
- (P + Q) * (R + S - T)
- ((A + B) * C) - (D / E + F)
- (A + B) ^ (C - D * E)
- F * G + H $ (I - J / K)
- (L ^ M + N) * (O - P)
- Q + (R * S - T) $ (U / V)

2.Postfix to Infix:

Steps for Converting Postfix to Infix using a Stack:

Steps:

1. **Initialize a stack** to store operands (infix expressions).
2. **Traverse the postfix expression** from left to right for each symbol:
    - **If the symbol is an operand**, push it onto the stack.
    - **If the symbol is an operator**:

- Pop the top two operands from the stack. Let's call them operand2 and operand1 (in this order, as the stack is LIFO).
- Form a new string with the operator and the operands: (operand1 operator operand2).
- Push the newly formed string back onto the stack.

3. **At the end of traversal**, the stack will contain a single element, which is the desired infix expression.

1. **Postfix**: ABC*+
2. **Postfix**: AB-CD+*
3. **Postfix**: ABC+/
4. **Postfix:** ABCDE^-*+
5. **Postfix**: AB+C*D/

 # Convert infix to postfix

1. **(A + B) * (C - D) + E / F**
2. **A + (B * C - D) / (E + F) - G**
3. **(A + B * (C - D)) / (E + F) - G**
4. **(A + B) * (C + D) - (E / F + G) * H**

 # Convert Postfix to Infix

1. **ABC*+DEF*+G-***
2. **AB+CD*E+F-/**
3. **AB+CD*EF/+*G+**
4. **AB+C*DE*+F/**

Recursion:

The process in which a function calls itself directly or individually is called recursion. It is a powerful technique of writing a complicated algorithm in an easy way.

Recursion is a programming technique that involves a function calling itself directly or indirectly to break down a problem into smaller, more manageable problems.

Principles of Recursion:

Understanding recursion involves grasping its key components: the **base case**, **recursive case**, and **call stack**. These elements work together to solve problems efficiently, making recursion a valuable tool in a programmer's toolkit for tackling diverse computational challenges.

- Recursion breaks down a problem into smaller subproblems solved by function calling itself
- Base case is a condition that stops recursion and returns a value without further function calls
    - Prevents infinite recursion by providing a termination condition
    - Returns a value for the simplest subproblem (**factorial** of 1 is 1)
    - When **n == 0 or n == 1**, the function returns 1 directly. This prevents infinite recursion.
- Recursive case is where the function calls itself with a modified input moving towards the base case
    - Gradually reduces problem size until base case is reached (factorial of n is n times factorial of n-1)
- Recursive functions have a structure with base case condition check and recursive function call with modified input

1. Factorial of Positive Number:

```
#include <stdio.h>

long fact(long n) {
   if  (n == 1):
      return 1;
   return n * fact(n - 1);
}

int main() {
   printf("Factorial of 6: %ld\n", fact(6));
   return 0;
```

}

Recursive Calls (Going Down):

1. **`fact(6)`**: Calls `6 * fact(5)`.
2. **`fact(5)`**: Calls `5 * fact(4)`.
3. **`fact(4)`**: Calls `4 * fact(3)`.
4. **`fact(3)`**: Calls `3 * fact(2)`.
5. **`fact(2)`**: Calls `2 * fact(1)`.
6. **`fact(1)`**: Base case is reached. Returns `1`.

Return Values (Coming Back Up):

1. **`fact(2)`**: Returns `2 * 1 = 2`.
2. **`fact(3)`**: Returns `3 * 2 = 6`.
3. **`fact(4)`**: Returns `4 * 6 = 24`.
4. **`fact(5)`**: Returns `5 * 24 = 120`.
5. **`fact(6)`**: Returns `6 * 120 = 720`.

Recursive Algorithm:

1.Greatest Common Divisor:

Algorithm Steps:

1. **Input**: Two integers `a` and `b`.
2. **Check Base Case**:
   - If `b` equals 0, then `a` is the GCD. Return `a`.
3. **Recursive Step**:
   - Otherwise, calculate the remainder of `a` divided by `b` (`a % b`).
   - Call the function recursively with the parameters `b` and `a % b`.
4. **Repeat**:
   - Repeat steps 2 and 3 until `b` becomes 0.
5. **Output**:
   - The current value of `a` is the GCD.

#Program

```
#include <stdio.h>

// Function to calculate GCD using recursion
int gcd(int a, int b) {
    if (b == 0)
        return a;  // Base case: when b becomes 0, a is the GCD
    return gcd(b, a % b);  // Recursive case
}

int main() {
    int num1, num2;

    // Input two numbers
    printf("Enter two integers: ");
    scanf("%d %d", &num1, &num2);

    // Calculate and display the GCD
    printf("GCD of %d and %d is: %d\n", num1, num2, gcd(num1, num2));
    return 0;
}
```

2. Sum of Natural Number:

Algorithm Steps:

1. **Input**: Read an integer `N` (the number of natural numbers to sum).
2. **Base Case**:
   - If `N == 1`, the sum is `1`. Return `1`.
3. **Recursive Step**:
   - Otherwise, return `N + sum(N - 1)`, where `sum(N - 1)` is the sum of all natural numbers up to `N - 1`.
4. **Repeat**:
   - The recursion continues until the base case (`N == 1`) is reached.
5. **Output**:
   - The final result is the sum of the first `N` natural numbers.

#Program

```
#include <stdio.h>

int sum(int n) {
    if (n == 0)
        return 0;  // Base case
    return n + sum(n - 1);  // Recursive case
}

int main() {
    printf("Sum of first 5 numbers: %d\n", sum(5));
    return 0;
}
```

4. **Fibonacci Series**

Algorithm Steps for Fibonacci Series:

1. **Input**: Read an integer N (the number of terms in the Fibonacci series).
2. **Base Case**:
   - If N=0, return 0 (the first term of the Fibonacci series).
   - If N=1, return 1 (the second term of the Fibonacci series).
3. **Recursive Step**:
   - For N>1, compute: Fibonacci(N) = Fibonacci(N−1) + Fibonacci(N−2)
   - This adds the previous two terms of the series.
4. **Repeat**:
   - Continue calling the function recursively for N−1 and N−2 until the base cases (N=0 or N=1) are reached.
5. **Output**:
   - Combine the results of all recursive calls to generate the Fibonacci series up to N.

 # Program

```
#include <stdio.h>

// Function to calculate the nth Fibonacci number
int fibonacci(int n) {
    if (n == 0) // Base case 1: First Fibonacci number
        return 0;
    if (n == 1) // Base case 2: Second Fibonacci number
        return 1;
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursive call
}

int main() {
    int n, i;

    // Input: Number of terms to generate
    printf("Enter the number of terms: ");
    scanf("%d", &n);
```

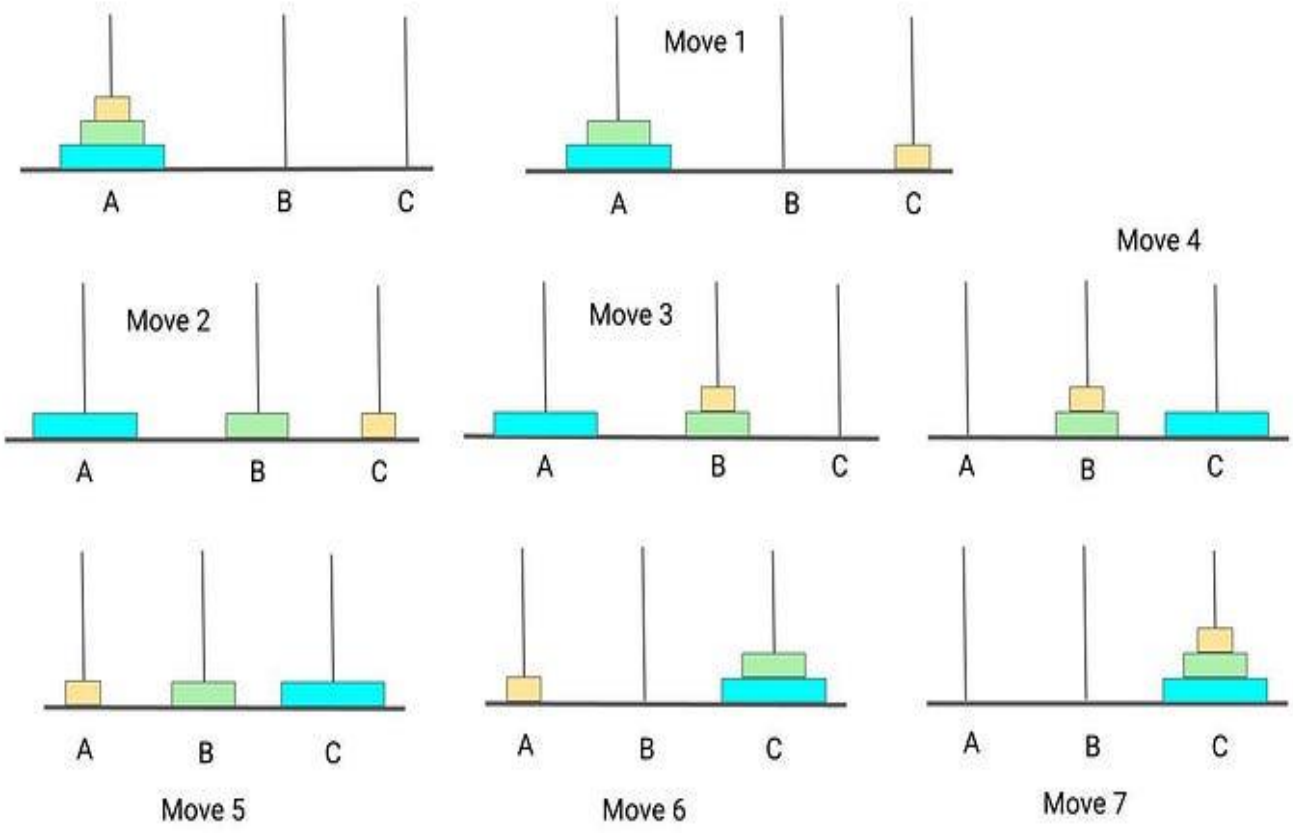
```
    // Output: Fibonacci series
    printf("Fibonacci series up to %d terms:\n", n);
    for (i = 0; i < n; i++) {
        printf("%d ", fibonacci(i));
    }

    printf("\n");
    return 0;
}
```

6. Tower of Hanoi:

The **Tower of Hanoi** is a classic recursive problem where the goal is to move nnn disks from a source peg to a destination peg, using an auxiliary peg, while following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be moved if it is the uppermost disk on a peg.
3. A larger disk cannot be placed on top of a smaller disk.



Program for Tower of Hanoi Algorithm

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.

Difference between Iteration and Recursion

The following table lists the major differences between iteration and recursion:

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.

Property	Recursion	Iteration
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.
Overhead	Possesses overhead of repeated function calls.	No overhead as there are no function calls in iteration.
Infinite Repetition	If the recursive function does not meet to a termination condition or the base case is not defined or is never reached then it leads to a stack overflow error and there is a chance that the an system may crash in infinite recursion.	If the control condition of the iteration statement never becomes false or the control variable does not reach the termination value, then it will cause infinite loop. On the infinite loop, it uses the CPU cycles again and again.

Recursive Data Structures:

Recursive data structures are structures where their definition or behavior relies on recursion, meaning they are defined in terms of smaller instances of themselves. These data structures often have a simple and elegant recursive definition and are commonly used in scenarios involving hierarchical or nested relationships.

1. Linked Lists

A **linked list** is a collection of nodes, where each node contains data and a reference (or pointer) to the next node in the list. It can be considered a recursive data structure because a list is often defined as a node pointing to another list.

2. Trees

A **tree** is a hierarchical data structure with a root node and child nodes. It's inherently recursive because each child node can be considered the root of a smaller subtree, which can recursively have its own children.

3. Graphs (Trees are a Special Case of Graphs)

A **graph** consists of nodes (vertices) and edges (connections between nodes). Graphs can also be recursive when defined using adjacency lists or adjacency matrices, where nodes reference other nodes, and recursion is useful for traversing them (e.g., DFS, BFS).

4. Trees and Graphs with Recursion in Traversals

In **tree** and **graph** traversals (in-order, pre-order, post-order, DFS, BFS), recursion is often used because the structure is hierarchical, and recursion elegantly follows the branching nature of trees or graphs.

For example, in tree traversal, the recursive function operates on subtrees, making it a classic example of recursion.

Types of Recursion:

1. Direct Recursion

- A function directly calls itself to solve smaller instances of the problem.
- This is typified by the factorial implementation where the methods call itself.

#Example

```
#include <stdio.h>

int fact(int n) {
```

```
    if (n == 1)
        return 1;  // Base case
    return n * fact(n - 1);  // Recursive case
}

int main() {
    printf("Factorial of 5: %d\n", fact(5));
    return 0;
}
```

---

2. Indirect Recursion

- A function calls another function, which in turn calls the original function (or another, forming a loop).
- Requires at least two functions to participate in the recursive process.
- This happens where one method, say method **A**, calls another method **B**, which then calls method **A**. This involves two or more methods that eventually create a circular call sequence.

```
#include <stdio.h>

void functionA(int n);
void functionB(int n);

void functionA(int n) {
    if (n > 0) {
        printf("A: %d\n", n);
        functionB(n - 1);  // A calls B
    }
}

void functionB(int n) {
    if (n > 0) {
        printf("B: %d\n", n);
        functionA(n - 1);  // B calls A
    }
```

```
}

int main() {
    functionA(3);  // Start with A
    return 0;
}
```

---

3. Tail Recursion

- The recursive call is the last operation in the function, with no further computation after it.
- The recursive call is the last statement.
- Efficient in terms of memory usage compared to non-tail recursion.

```
#include <stdio.h>

int func(int n) {

    if (n == 0)

        return;

else

        print("%d",n);

 return func(n-1);

}

int main() {

    func(3)

    return 0;

}
```

4. Non-Tail Recursion

- The recursive call is not the last operation; computation occurs after the recursive call returns.
- Requires additional stack space because pending operations are stored on the stack.
- Examples include Fibonacci and Tower of Hanoi, where results depend on multiple recursive calls.

```
#include <stdio.h>

int func(int n) {

    if (n == 0)

        return;

 return func(n-1);

 print("%d",n);

}

int main() {

    func(3)

    return 0;

}
```

```
#include <stdio.h>

int fibonacci(int n) {

    if (n <= 1)

        return n;  // Base case

    return fibonacci(n - 1) + fibonacci(n - 2);  // Recursive case

}
```

```
int main() {
    printf("Fibonacci of 5: %d\n", fibonacci(5));
    return 0;
}
```

Applications of Recursion:

Recursion is used in many fields of computer science and mathematics, which includes:

1. Data Structure Traversal

Recursion is ideal for traversing hierarchical or nested data structures:

- **Linked Lists** – Visiting or modifying each node.
- **Trees** – In-order, pre-order, post-order traversals.
- **Graphs** – Depth-First Search (DFS).

2. Mathematical Computations

Recursion helps solve problems that can be broken into smaller subproblems:

- **Factorial**: `n! = n × (n-1)!`
- **Fibonacci Sequence**: `F(n) = F(n-1) + F(n-2)`
- **Greatest Common Divisor (GCD)** using Euclid's algorithm.

3. Divide and Conquer Algorithms

Recursion is the backbone of many efficient algorithms:

- **Merge Sort**
- **Quick Sort**
- **Binary Search**

These algorithms divide the problem into smaller chunks, solve them recursively, and then combine the results.

4. Backtracking

Recursion is used to explore all possible solutions in problems like:

- **Sudoku Solver**
- **N-Queens Problem**
- **Maze Solving**
- **Generating all permutations/combinations**

5. Parsing and Expression Evaluation

Recursion is used in compilers and interpreters:

- Parsing mathematical expressions or programming language syntax.
- Evaluating nested expressions like `((2 + 3) * (5 - 1))`.