1. Stack

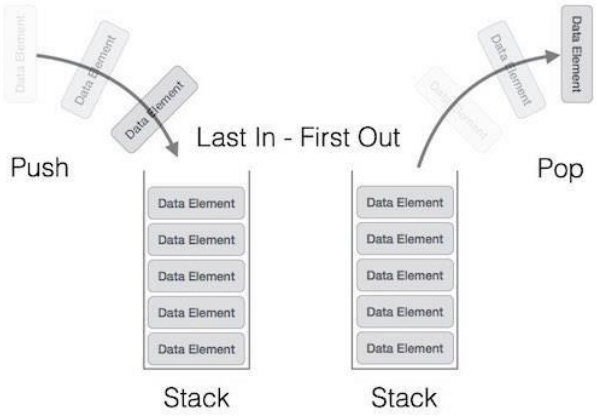
1.1 Definition and Stack Operations

. A stack is an Abstract Data Type (ADT), that is popularly used in most programming languages. The stack follows the LIFO (Last in - First out) structure where the last element inserted would be the first element deleted

Stack Representation

A Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations on Stacks

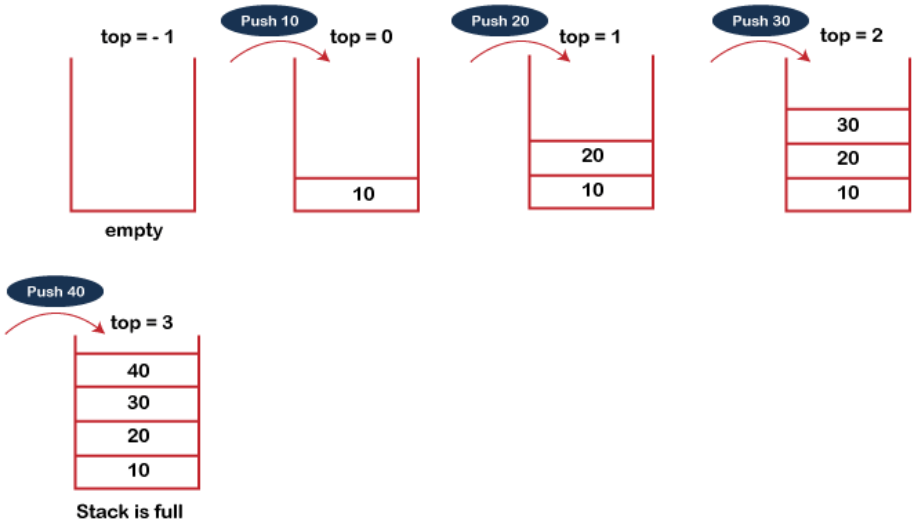
Stack operations usually are performed for initialization, usage and, de-initialization of the stack ADT.

The most fundamental operations in the stack ADT include: push(), pop(), peek(), isFull(), isEmpty().

Stack uses pointers that always point to the topmost element within the stack, hence called as the **top** pointer.

Insertion: push()

push() is an operation that inserts elements into the stack. The following is an algorithm that describes the push() operation in a simpler way.

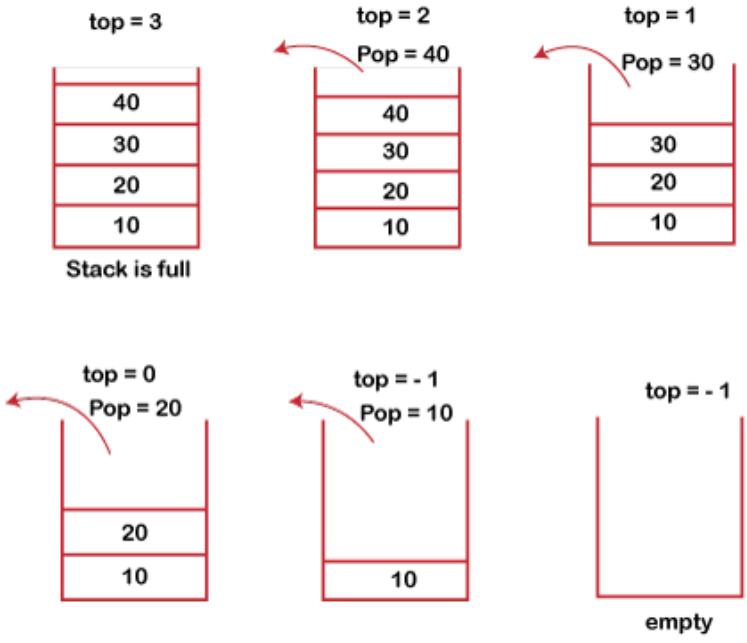


Algorithm

1 − Checks if the stack is full.
2 − If the stack is full, produces an error and exit.
3 − If the stack is not full, increments top to point next empty space.
4 − Adds data element to the stack location, where top is pointing.
5 − Returns success.

Deletion: pop()

pop() is a data manipulation operation which removes elements from the stack. The following pseudo code describes the pop() operation in a simpler way.



Algorithm
1 − Checks if the stack is empty.
2 − If the stack is empty, produces an error and exit.
3 − If the stack is not empty, accesses the data element at which top is pointing.
4 − Decreases the value of top by 1.
5 − Returns success.

peek()

The *peek()* is an operation retrieves the topmost element within the stack, without deleting it. This operation is used to check the status of the stack with the help of the top pointer.

Algorithm
1. START
2. return the element at the top of the stack
3. END

isFull()

isFull() operation checks whether the stack is full. This operation is used to check the status of the stack with the help of top pointer.

Algorithm
1. START
2. If the size of the stack is equal to the top position of the stack, the stack is full. Return 1.
3. Otherwise, return 0.
4. END

isEmpty()

The *isEmpty()* operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Algorithm
1. START
2. If the top value is -1, the stack is empty. Return 1.
3. Otherwise, return 0.
4. END

Applications

- Function call management in programming languages.
- Expression evaluation and parsing.
- Undo/Redo operations in applications.
- Backtracking algorithms.
- Depth-first search (DFS) algorithms.

- Used in recursion
- Balancing symbols, such as parentheses, in code.
- Conversion of Postfix to Infix expression
- Conversion of Infix to Postfix expression

1.2 Stack ADT and its Array Implementation

Stack ADT

Defination: A list of data items than can only be accessed at one end, called the top of the stack.

Operations: stack: Creates an empty stack

Push: Inserts an element at the top.

Pop: Deletes the top element

Empty/Full: Check the status of the stack

St

- Array Implementation (static implementation)
- Linked List Implementation (dynamic implementation)

Array Implementation of Stack using C++:

```
#include <iostream>

using namespace std;

const int MAX_SIZE = 100;

class Stack {
private:
    int data[MAX_SIZE];
    int top;

public:
    Stack() {
        top = -1;
    }

    void push(int item) {
        if (isFull()) {
            cout << "Stack Overflow! Cannot push item." << endl;
            return;
```

```
        }

        data[++top] = item;
    }

    int pop() {
        if (isEmpty()) {
            cout << "Stack Underflow! Cannot pop item." << endl;
            return -1;  // Returning a special value to indicate error
        }

        return data[top--];
    }

    int peek() {
        if (isEmpty()) {
            cout << "Stack is empty." << endl;
            return -1;  // Returning a special value to indicate error
        }

        return data[top];
    }

    bool isEmpty() {
        return (top == -1);
    }

    bool isFull() {
        return (top == MAX_SIZE - 1);
    }
};

int main() {
    Stack stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);

    cout << "Top element: " << stack.peek() << endl;

    cout << "Popped element: " << stack.pop() << endl;
    cout << "Popped element: " << stack.pop() << endl;

    cout << "Top element: " << stack.peek() << endl;
```

```
    return 0;
}
```

In this implementation, the stack is implemented using a fixed-size array data[] to store the elements. The class Stack has member functions push(), pop(), peek(), isEmpty(), and isFull() which perform similar operations as described earlier. The maximum size of the stack is defined by the constant MAX_SIZE.

In the main() function, we create a Stack object, push some elements onto it, and then pop and peek the elements to demonstrate the basic operations of a stack.

1.3 Expression Evaluation: Infix and Postfix

An expression is defined as the number of operands or data items combined with several operators. One of the applications of the stack is to evaluate the expression. We can represent the expression in three types:

1. Infix expression
2. Postfix expression
3. Prefix expression

Infix Expressions

The usual expressions which we encounter are infix expressions.

For example,

(A + B) * C / D - E

Postfix Expressions

In postfix expressions, the operators are written after the operands as shown below:

A B C + * D /

Prefix Expressions

Here, the operators are written before the operands. An example is,

/ * A + B C D

Precedence of Operators

In our school days, we read about BODMAS or PEMDAS. What exactly were those?

They were acronyms to remember the precedence of operators while solving an expression, but what is precedence?

The precedence of operators gives us an order in which operators are evaluated in any expression.

Computers have a similar idea of precedence, too, when it comes to operators. It is as follows:

- Exponential (^) (Highest precedence Note: ^ = $)
- Multiplication and division (* /) (Next precedence)
- Addition and subtraction (+ –) (Least precedence)

- **Evaluation of Infix expression :**

Infix Expression: 2 * (5 * (3 + 6)) / 5 - 2

Character	Action	Operand Stack	Operator Stack
2	Push to the operand stack	2	
*	Push to the operator stack	2	*
(	Push to the operator stack	2	(*
5	Push to the operand stack	5 2	(*
*	Push to the operator stack	5 2	* (*
(	Push to the operator stack	2 1	(* (*
3	Push to the operand stack	3 5 2	(* (*
+	Push to the operator stack	3 2 1	+ (* (*
6	Push to the operand stack	6 3 5 2	+ (* (*
)	Pop 6 and 3	5 2	+ (* (*
	Pop +	5 2	(* (*
	6 + 3 = 9, push to operand stack	9 5 2	(* (*
	Pop (	9 5 2	* (*
)	Pop 9 and 5	2	* (*
	Pop *	2	(*
	9 * 5 = 45, push to operand stack	45 2	(*
	Pop (	45 2	*
/	Push to the operator stack	45 2	/ *
5	Push to the operand stack	5 45 2	/ *
-	Pop 5 and 45	2	/ *
	Pop /	2	*
	45/5 = 9, push to the operand stack	9 2	*
	Pop 9 and 2		*
	Pop *		
	9 * 2 = 18, push to operand stack	18	
	Push - to the operator stack	18	-
2	Push to the operand stack	2 18	-
	Pop 2 and 18		-
	Pop -		
	18 - 2 = 16, push to operand stack	16	

- The final answer (here 16) is stored in the stack and is popped at the end.

- **Evaluation of Postfix expression :**

Postfix expression: 2 3 1 * + 9 –

SN	Character Scanned	Operand Stack	Operation Performed
1	2	2	Since 2 is operand, push it into the Operand Stack
2	3	2 3	Since 3 is operand, push it into the Operand Stack
3	1	2 3 1	Since 1 is operand, push it into the Operand Stack
4	*	2 3	Pop value1 = 1, Pop value2 = 3, Compute: 3 * 1 = 3, push 3 into the Operand stack.
5	+	5	Pop value1 = 3, Pop value2 = 2, Compute: 2 + 3 = 5, push 5 into the Operand stack.
6	9	5 9	Since 9 is operand, push it into the Operand Stack
7	-	**-4**	Pop value1 = 9, Pop value2 = 5, Compute: 5 - 9 = -4, push -4 into the Operand stack.

There are no more elements to scan, we return the top element from stack and it is **-4** (which is the value of the given expression)

1.4 Expression Conversion

Infix to Postfix

Converting an infix expression to postfix notation (also known as Reverse Polish Notation) involves rearranging the expression so that operators are written after their operands. Here are the steps to convert infix to postfix:

1. Initialize an empty stack to store operators temporarily.
2. Scan the infix expression from left to right.

3. For each element in the infix expression:
    - If the element is an operand (number or variable), output it directly.
    - If the element is an opening parenthesis '(', push it onto the stack.
    - If the element is a closing parenthesis ')', pop operators from the stack and output them until an opening parenthesis is encountered. Discard the opening parenthesis.
    - If the element is an operator (+, -, *, /, etc.), compare its precedence with the operator at the top of the stack:
        - If the stack is empty or contains an opening parenthesis '(', push the operator onto the stack.
        - If the current operator has higher precedence than the operator at the top of the stack, push it onto the stack.
        - If the current operator has equal or lower precedence than the operator at the top of the stack, pop the operator from the stack and output it. Repeat this step until the stack is empty, an opening parenthesis is encountered, or the current operator has higher precedence than the operator at the top of the stack. Then push the current operator onto the stack.
4. After scanning the entire infix expression, pop any remaining operators from the stack and output them.
5. The output will be the equivalent postfix expression.

Infix expression: A * (B + C * D) + E

SN	**Scanned token**	**operator stack (opstack)**	**postfix string (output list)**
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +

10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			**A B C D * + * E +**

Postfix to Infix

Converting postfix notation (also known as Reverse Polish Notation) to infix notation involves converting an expression with operators written after their operands to an expression where operators are written between their operands. Here are the steps to convert postfix to infix:

1. Initialize an empty stack to store intermediate results.
2. Scan the postfix expression from left to right.
3. For each element in the postfix expression:
    - If the element is an operand (number or variable), push it onto the stack.
    - If the element is an operator (+, -, *, /, etc.), pop the top two elements from the stack (let's call them `operand1` and `operand2`).
    - Construct a string by concatenating `operand2`, the operator, and `operand1`, surrounded by parentheses.
    - Push the constructed string onto the stack.
4. After scanning the entire postfix expression, the final infix expression will be at the top of the stack.
5. Pop the infix expression from the stack and display it.

Postfix : AB*C+

Character Scanned	**Stack**
A	A
B	A B
*	A*B
C	A*B C
+	(A*B)+C

2. Recursion

2.1 Recursion - A problem Solving Technique

Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller subproblems. It involves solving a problem by reducing it to a simpler version of the same problem.

The process of recursion generally follows these steps:

1. **Define the base case**: The base case is the simplest version of the problem that can be directly solved without further recursion. It acts as the termination condition for the recursion. When the base case is reached, the recursion stops, and the result is returned.
2. **Identify the recursive case**: The recursive case refers to the part of the function where it calls itself with modified parameters. It represents the breakdown of the original problem into smaller subproblems of the same type.
3. **Modify the parameters**: In each recursive call, the parameters or inputs to the function should be modified in a way that brings the problem closer to the base case. This ensures that the recursive calls eventually reach the base case and terminate.
4. **Recur**: Within the function, make a recursive call to itself, passing the modified parameters. This step allows the function to solve the smaller subproblem and move closer to the base case.
5. **Combine the results**: As the recursion unwinds and the base case is reached, the results from each recursive call are combined or processed to obtain the final solution. This combination step may involve aggregating results, performing calculations, or any other required operation.

It's important to ensure that the recursive calls eventually reach the base case to avoid infinite recursion. Without proper termination conditions, the recursion can continue indefinitely, leading to stack overflow errors.

Recursion is often used to solve problems that exhibit repetitive or self-referential structures. It provides an elegant and intuitive way to solve such problems by dividing them into smaller, more manageable subproblems. Recursive solutions can be concise, expressive, and reflect the natural structure of the problem at hand.

2.2 Principle of Recursion

The principle of recursion is based on solving a problem by breaking it down into smaller, simpler instances of the same problem. It involves defining a function or algorithm that calls itself with modified parameters, working towards a base case that defines when the recursion should stop.

The principle of recursion typically involves the following elements:

1. Base case: A base case is a condition that defines when the recursion should terminate. It represents the simplest version of the problem that can be directly solved without further recursion. When the base case is reached, the recursion stops, and the results are returned or used to build the solution.
2. Recursive case: The recursive case refers to the part of the function or algorithm that calls itself with modified parameters. It represents the breakdown of the original problem into smaller subproblems of the same type. By calling itself, the function solves these subproblems, moving closer to the base case.
3. Progress towards the base case: In each recursive call, the parameters or input to the function should be modified in a way that brings the problem closer to the base case. This ensures that the recursive calls eventually reach the base case and terminate.
4. Combination of results: As the recursion unwinds and the base case is reached, the results from each recursive call are combined or processed to obtain the final solution. This combination step may involve aggregating results, performing calculations, or any other required operation.

By following the principle of recursion, complex problems can be broken down into simpler instances, making them easier to solve. It often mirrors the natural structure of the problem and can lead to concise and elegant solutions. However, it is important to carefully define the base case and ensure that the recursive calls progress towards it, avoiding infinite recursion and ensuring termination.

2.3 Recursive Algorithms

2.3.1 Greatest Common Divisor

```
#include <iostream>
using namespace std;

int gcd(int a, int b) {
   if (b == 0) {
      return a; // Base case: when b becomes zero, a is the GCD
   } else {
      return gcd(b, a % b); // Recursive step: recursively call gcd with b and the remainder of a
divided by b
   }
}

int main() {
   int num1=63, num2=42;

   int result = gcd(num1, num2);
   cout << "GCD: " << result << endl;

   return 0;
}
```

2.3.2 Sum of Natural Numbers

```
#include <iostream>

using namespace std;

int sumOfNaturalNumbers(int n) {
    if (n == 0)
        return 0;
    else
        return n + sumOfNaturalNumbers(n - 1);
}

int main() {
    int number;
    cout << "Enter a positive integer: ";
    cin >> number;

    int sum = sumOfNaturalNumbers(number);
    cout << "The sum of natural numbers up to " << number << " is: " << sum << endl;

    return 0;
}
```

2.3.3 Factorial of a Positive integer

```
#include<iostream>
using namespace std;

int add(int n);

int main() {
    int n;

    cout << "Enter a positive integer: ";
    cin >> n;
    cout << "Sum =  " << add(n);
    return 0;
}
int add(int n) {
    if(n != 0)
        return n + add(n - 1);
    return 0;
}
```

2.3.4 Fibonacci Series

```
#include<iostream>
    using namespace std;

  void printFibonacci(int n){
    static int n1=0, n2=1, n3;
    if(n>0){
        n3 = n1 + n2;
        n1 = n2;
        n2 = n3;
        cout<<n3<<" ";
        printFibonacci(n-1);
    }
  }

  int main(){
    int n;
    cout<<"Enter the number of elements: ";
    cin>>n;

    cout<<"Fibonacci Series: ";
    cout<<"0 "<<"1 ";

    printFibonacci(n-2);  //n-2 because 2 numbers are already printed
    return 0;
  }
```

2.3.5 Tower of Hanoi

```
#include<iostream>
using namespace std;

//tower of HANOI function implementation
void TOH(int n,char Sour, char Aux,char Des)
{
        if(n==1)
        {
                cout<<"Move Disk "<<n<<" from "<<Sour<<" to "<<Des<<endl;
                return;
        }

        TOH(n-1,Sour,Des,Aux);
        cout<<"Move Disk "<<n<<" from "<<Sour<<" to "<<Des<<endl;
        TOH(n-1,Aux,Sour,Des);
}
```

```
//main program
int main()
{
	int n=3; //number of disks

	//calling the TOH
	TOH(n,'A','B','C');

	return 0;
}
```

2.4 Recursion and Stack

Recursion and the stack are two related concepts that often go hand in hand, but they serve different purposes and have distinct characteristics. Let's explore the differences between recursion and the stack:

Recursion:

- Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller subproblems.
- It involves solving a problem by reducing it to a simpler version of the same problem.
- Recursive functions have a base case that defines when the recursion should stop and one or more recursive cases that call the function again with modified parameters.
- Recursion can be an elegant and intuitive way to solve certain problems, especially those that exhibit repetitive or self-referential structures.
- Recursive algorithms often mirror the structure of the problem, making the code easier to understand and implement.
- However, recursive solutions may have performance implications, as they can consume more memory and potentially result in stack overflow if the recursion depth becomes too large.

Stack:

- A stack is an abstract data type that follows the Last-In-First-Out (LIFO) principle. It is a linear data structure in which elements are added or removed from the top.
- The stack data structure can be implemented using an array or a linked list.
- It has two primary operations: push (adding an element to the top) and pop (removing the top element).
- The stack is widely used in programming to manage function calls, particularly in recursive scenarios.
- Each time a function is called, the function call is pushed onto the stack, along with its local variables and the program's execution context.
- When a function completes its execution, it is popped from the stack, and the program returns to the point where the function was called.
- The stack provides the necessary mechanism for maintaining the order and state of function calls during recursion.

In summary, recursion is a programming technique that involves a function calling itself to solve a problem by breaking it down into smaller subproblems. The stack, on the other hand, is a data structure that facilitates the management of function calls, including maintaining their order and state. Recursion often relies on the stack to keep track of the function calls and their associated data.

2.5 Recursion Vs Iteration

Recursion	Iteration
Recursion is the technique of defining anything in terms of itself.	It is a process of executing a statement or set of statements repeatedly, until some specific condition is satisfied.
Stack is used to store local variables when the function is called.	Stack is not used.
Recursion terminates when the base case is met.	Iteration terminates when the condition in the loop fails.
Recursion uses more memory in comparison to iteration.	iteration uses less memory in comparison to recursion.
The system crashes when infinite recursion is encountered.	Iteration uses the CPU cycles again and again when an infinite loop occurs.

2.6 Recursive Data Structure

A recursive data structure is a type of data structure that can contain references to other instances of the same type. In other words, it is a self-referential data structure.

The defining characteristic of a recursive data structure is that it can be defined in terms of itself. This allows for the creation of complex structures by combining simpler instances of the same structure.

For example, one common recursive data structure is the linked list. A linked list consists of nodes, where each node contains a value and a reference to the next node in the list. The last node in the list typically has a reference to null, indicating the end of the list. The structure is recursive because each node contains a reference to another node of the same type (or null to terminate the list).

Recursive data structures are often used when dealing with hierarchical or nested data, such as trees or graphs. They provide a flexible and dynamic way to represent complex relationships and can be processed using recursive algorithms that traverse and manipulate the structure.

Recursive data structures are needed because they provide a powerful and flexible way to represent and process complex data relationships. Here are a few reasons why recursive data structures are useful:

1. Hierarchical and nested data: Recursive data structures are well-suited for representing hierarchical or nested data, where elements can contain sub-elements of the same type. For example, a file system can be represented as a tree structure, with directories containing files and sub-directories.
2. Dynamic structure: Recursive data structures allow for the creation of dynamic data structures that can grow or shrink as needed. New elements can be added or removed by simply creating or deleting references to other instances of the same type. This flexibility is particularly useful when dealing with data that changes over time or in response to user actions.
3. Recursive algorithms: Recursive data structures naturally lend themselves to recursive algorithms. Recursive algorithms are based on the concept of solving a problem by breaking it down into smaller subproblems of the same type. These algorithms often mirror the structure of the data, making them easier to understand and implement.
4. Memory efficiency: Recursive data structures can be memory-efficient because they allow for the reuse of the same structure at different levels. Instead of duplicating the entire structure for each level, only references to existing instances need to be stored, resulting in less memory consumption.
5. Code reusability: Recursive data structures promote code reusability. Once a data structure is defined recursively, it can be used to represent and process similar data in various contexts. This can save development time and effort by leveraging existing code and algorithms.

2.7 Types of Recursion

1. **Direct Recursion**: In direct recursion, a function directly calls itself within its own body.

```
void directRecursion(int n) {
    if (n <= 0)
        return;

    cout << n << endl;
    directRecursion(n - 1);
}
```

2. **Indirect Recursion**: In indirect recursion, a function calls another function(s), which in turn calls the original function or calls other functions that ultimately lead back to the original function.

```
void indirectRecursionA(int n);

void indirectRecursionB(int n) {
    if (n <= 0)
```

```
        return;

    cout << n << endl;
    indirectRecursionA(n - 1);
}

void indirectRecursionA(int n) {
    if (n <= 0)
        return;

    cout << n << endl;
    indirectRecursionB(n - 1);
}
```

3. **Tail Recursion**: In tail recursion, the recursive call is the last operation performed within a function. The result of the recursive call is directly returned without any further processing.

```
int tailRecursion(int n, int result = 0) {
    if (n == 0)
        return result;

    return tailRecursion(n - 1, result + n);
}
```

4. **Non-tail Recursion**: In non-tail recursion, there are additional operations to be performed after the recursive call before returning.

```
void nonTailRecursion(int n) {
    if (n <= 0)
        return;

    nonTailRecursion(n - 1);
    cout << n << endl;
}
```

5. **Nested Recursion**: In nested recursion, a recursive function is called within the recursive call as an argument.

```
int nestedRecursion(int n) {
    if (n <= 0)
        return 0;

    cout << n << endl;
    return nestedRecursion(nestedRecursion(n - 1));
}
```

2.8 Applications of Recursion

Recursion is a powerful technique that finds applications in various areas of computer science and problem-solving. Here are some common applications of recursion:

1. Mathematical Calculations: Recursion is often used to solve mathematical problems that can be defined recursively. Examples include calculating factorial, Fibonacci sequence, exponentiation, and permutations/combinations.
2. Tree and Graph Traversals: Recursive algorithms are frequently employed for traversing and manipulating tree and graph structures. They can be used for tasks such as searching, inserting, deleting, and modifying nodes in a tree or graph.
3. Backtracking: Backtracking algorithms often utilize recursion to explore all possible solutions to a problem by trying different choices and undoing incorrect choices when necessary. Examples include solving puzzles like Sudoku or the N-Queens problem.
4. File System Operations: Recursive algorithms are commonly used to traverse and perform operations on hierarchical file systems. For instance, listing all files in a directory and its subdirectories, calculating the total size of a directory, or searching for specific files.
5. Divide and Conquer Algorithms: Many divide and conquer algorithms utilize recursion to break down a problem into smaller subproblems, solve them recursively, and combine their results. Examples include merge sort, quick sort, and binary search.
6. Parsing and Syntax Analysis: Recursive descent parsing is a technique commonly used in compiler design and syntax analysis. It involves recursive functions that parse and analyze the structure of a programming language's grammar.
7. Dynamic Data Structures: Recursive data structures, such as linked lists, trees, and graphs, often require recursive algorithms for their construction, manipulation, and traversal.
8. Fractals and Graphics: Recursion finds applications in generating complex and self-similar patterns, such as fractals. Fractal generation algorithms use recursive calls to create intricate graphical representations.

These are just a few examples of how recursion is applied in various domains. Recursion provides an elegant and intuitive approach to solving problems that exhibit repetitive or self-referential structures, making it a fundamental concept in computer science and programming.