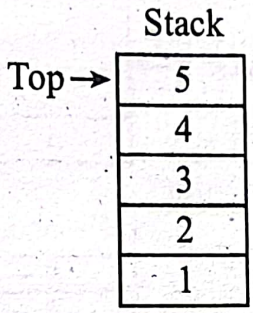
Chapter – 2

STACK AND RECURSION

1. Stack

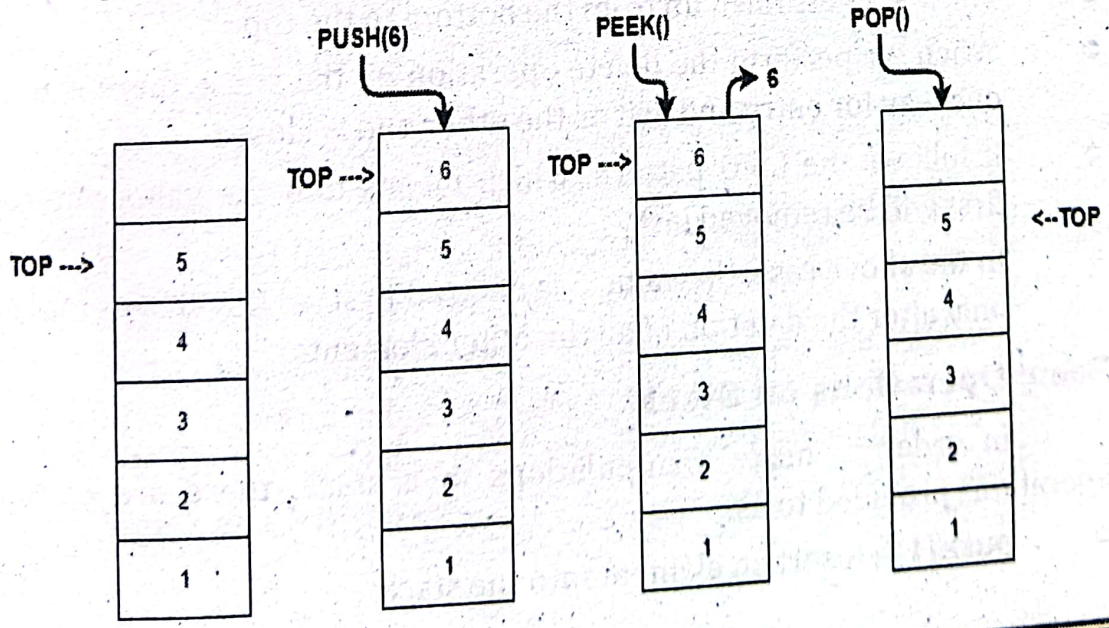
1.1 Definition and Stack Operation

- A stack is a linear data structure in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack.
- A stack is an abstract data type (ADT) which is used to store data in a linear fashion. A stack only has a single end (which is stack's top) through which we can insert or delete data from it.
- A Stack is a data structure following the LIFO(Last In, First Out) principle.



If you have trouble visualising stacks, just assume a stack of books.

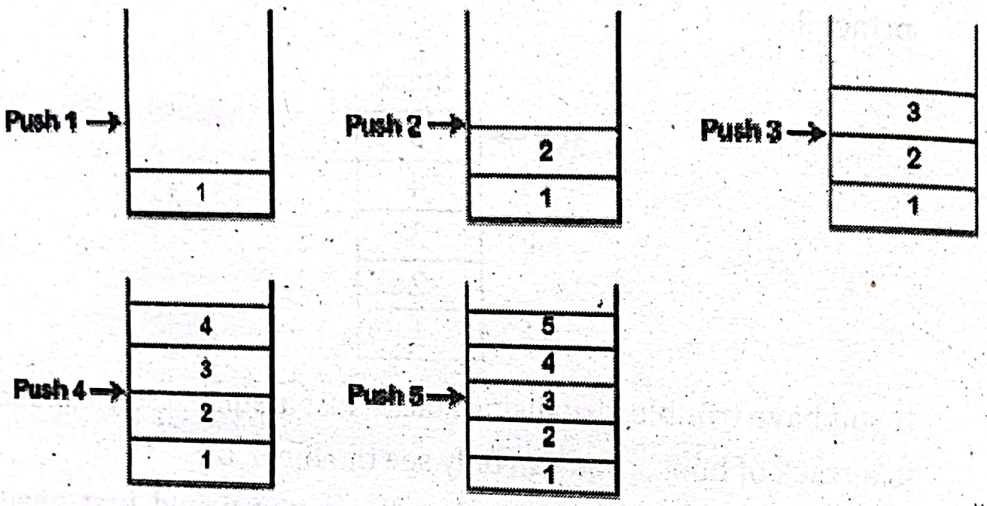
- In a stack of books, you can only see the top book
- If you want to access any other book, you would first need to remove the books on top of it
- The bottom-most book in the stack was put first and can only be removed at the last after all books on top of it have been removed.



- It is called a stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a predefined 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.

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 first, so it will be removed only after the deletion of all the other elements.

Basic Operations on Stack

In order to make manipulations in a stack, there are certain operations provided to us.

- **push()** to insert an element into the stack

- **pop()** to remove an element from the stack
- **top()** Returns the top element of the stack.
- **isEmpty()** returns true if stack is empty else false.
- **size()** returns the size of stack.
- **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.

1.2 Stack as ADT and its Array Implementation

Stack as ADT

[2013 Fall]

A stack of elements of type T is a finite sequence of elements of T together with the operations

- **CreateEmptyStack(S)**: Create or make stack *S* be an empty stack
- **Push(S, x)**: Insert *x* at one end of the stack, called its top
- **Top(S)**: If stack *S* is not empty; then retrieve the element at its top
- **Pop(S):** If stack *S* is not empty; then delete the element at its top
- **IsFull(S):** Determine if *S* is full or not. Return true if *S* is full stack; return false otherwise
- **IsEmpty(S):** Determine if *S* is empty or not. Return true if *S* is an empty stack; return false otherwise.

Thus by using a stack we can perform above operations thus a stack acts as an ADT. Here, all the operation works like a black box that it only deals with what operations are performed, hiding the details of how the operation is performed.

So, we can consider stack as ADT.

Primitive operations of stack

[2014 Spring]

The following operations can be performed on a stack:

1. **PUSH() operation:**
   - The push operation is used to add (or push or insert) elements in a Stack.
   - When we add an item to a stack, we say that we push it onto the stack.
   - The last item put into the stack is at the top.

Algorithm for push operation on stack

In an algorithm of an array ST[N] is used for the implementation of stack and variable 'Top' keeps track of the position of the stack. A value 'x' is added to the stack if it is not already full.

Step I: check if(top>=N-1)

{

Display "stack is full !!!!!" // **This is called overflow condition in stack.**

exit

}

Step II: set top=top+1;

Step III: set ST[top]=x;

Step IV: stop.

2. **POP operation:**

The pop operation is used to remove or delete the top element from the stack. we remove an item, we say that we pop an item from the stack. When an item pops, it is always the top item which is removed.

Algorithm for pop operation on stack

In an algorithm of an array ST[N] is used for the implementation of stack and variable 'Top' keeps track of the position of the stack. A value 'x' is added to the stack if it is not already full.

Step I: check if(top==-1)

{

Display "stack is empty !!!!!" // **This is called overflow condition in stack.**

exit;

}

Step II: set x=ST[top];

Step III: set top=top-1;

Step IV: stop.

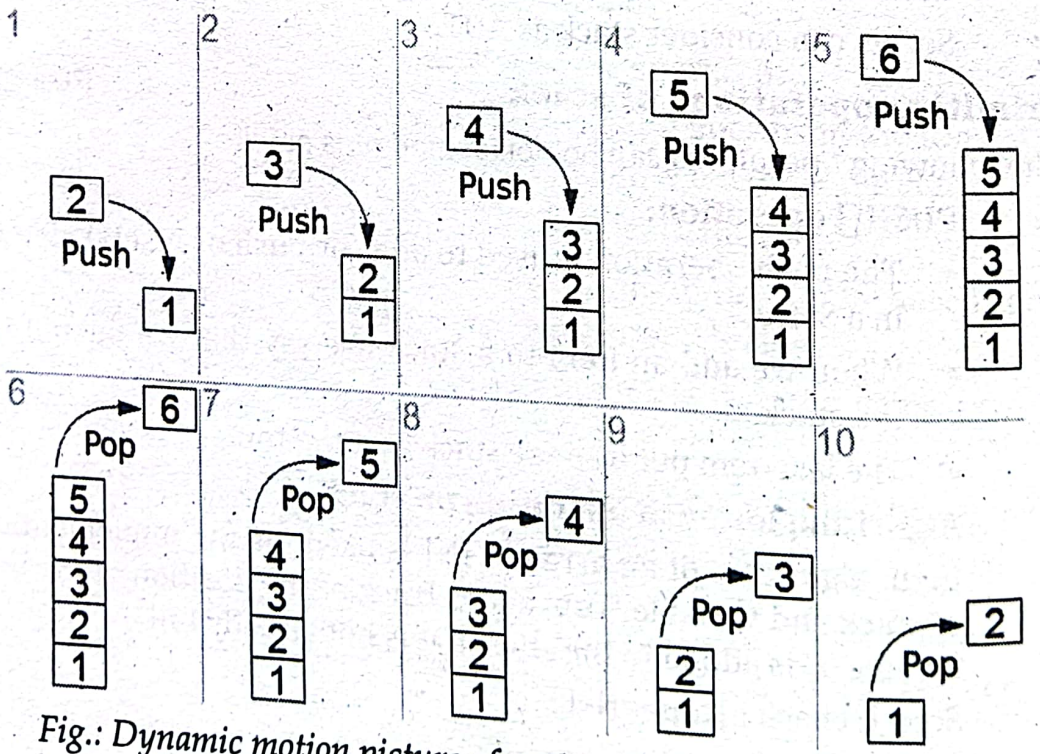


Fig.: Dynamic motion picture of push and pop operation on Stack

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)

- Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.
  1. Increment the variable Top so that it can now refer to the next memory location.
  2. Add an element at the position of the incremented top. This is referred to as adding a new element at the top of the stack.
- Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Deletion of an element from a stack (Pop operation)

- Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack.
- The top most element of the stack is stored in another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.
- The underflow condition occurs when we try to delete an element from an already empty stack.

Push() and pop() operation showing Fixed sized array implementation using c++ programming.

```
#include <iostream.h>
#include<conio.h>,
Class Stack
{
Private:
    int top;
    int a[10];  // Array implementation using stack
Public:
        Stack
        {
                top=-1;
        }
```

```
void push(int data)  // stack as ADT using push operation.
        {
                if(a==9)
                {
                        cout<<"stack is full!!!"<<endl;
                }
                else
                {
                        top++;
        a[top]=data; // keeping data to stack top array.

                }
        }

        void pop() // Stack as ADT using pop operation
        {
                if(a==-1)
                {
                        cout<<"stack is empty!!!"<<endl;
                }
                else
                {
                cout<<"popped item is"<<a[top]<<endl;
                        top--;
                }
        }

        void display()
        {
                for(int i=0; i<=top;i++)
                {
                        cout<<a[i]<<endl;
                }
        }
};
void main()
{
```

```
    Stack s;

    s.push(10); // it pushes 10 to the empty stack making the
    stack top from -1 to 0
        s.push(20); // it push 20 to stack top
            s.push(30);// it pushes 30 to the stack top
                s.push(40); // it pushed 40 to the
    stack top

    s.display(); //it display the stack data

    s.pop(); //it deletes the stack top data in LIFO order
     s.pop(); //it deletes the stack top data in LIFO order
     getch();
  }
```

Types of Stacks:

- **Fixed Size Stack:** As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.
- **Dynamic Size Stack:** A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

Advantages of Stacks:

- **Simplicity and Efficiency:** Stacks are simple to implement and use, making them efficient for certain operations.
- **Memory Management:** Stacks are essential for managing memory in function calls and local variable storage.
- **Undo and Redo Operations:** Stacks facilitate easy implementation of undo and redo features in applications.
- **Algorithmic Applications:** Stacks are fundamental in various algorithms like depth-first search and backtracking.
- **Syntax Parsing:** Used in parsing and evaluating expressions, making them crucial in compilers and interpreters.
- **Browser Navigation:** Enables efficient implementation of forward and backward navigation in web browsers.

Disadvantages of Stacks:

- **Limited Access:** Access to elements is limited to the top of the stack, restricting flexibility in certain scenarios.
- **Fixed Size (in some implementations):** Some stack implementations have a fixed size, which can lead to overflow issues if not managed carefully.
- **No Random Access:** Lack of random access to elements makes it unsuitable for situations where direct access to any element is necessary.
- **Potential for Stack Overflow:** In certain situations, if not properly managed, a stack can lead to a stack overflow error, especially in recursive algorithms.
- **Not Suitable for All Data Structures:** Stacks are not suitable for all types of data structures or scenarios, limiting their applicability.
- **Complexity in Undo/Redo Tracking:** While useful for undo and redo operations, tracking changes and managing a stack of states can become complex in large-scale applications.

Application of stack in computing world [2013 Fall/2015 Spring/2017 Fall]

- **Infix to Postfix/Prefix Conversion:** Stacks are used for converting mathematical expressions from infix to postfix/prefix notation.
- **Undo-Redo Features:** Found in editors, Photoshop, and similar applications for reverting and redoing actions.
- **Web Browsers Navigation Stacks** facilitate forward and backward navigation features in web browsers.
- **Algorithmic Problems:** Utilised in algorithms such as Tower of Hanoi, tree traversals, stock span, and histogram problems.
- **Backtracking:** Essential for problems like Knight-Tour, N-Queen, maze navigation, and game strategies.
- **Graph Algorithms:** Applied in Topological Sorting and Strongly Connected Components.
- **Memory Management:** Used as the primary tool for managing memory allocations in computer programs.
- **String Reversal:** Stacks efficiently reverse the order of characters in a string.
- **Function Call Implementation:** Helps manage function calls in computers, ensuring the last-called function is completed first.
- **Undo/Redo in Text Editors:** Stacks play a crucial role in implementing undo and redo operations in text editors.

Application of stack in Non-computing world

[2013 Fall/2015 Spring/2017 Fall]

- **Plate Stacking:** Used in plate dispensers where plates are stacked, and the top plate is accessible.
- **Book Piles:** Represents stacks of books where the top book is easily reachable.
- **LIFO Systems:** Various systems employing Last In, First Out (LIFO) principles, such as lifeguard tubes or firewood stacking.
- **Tray Stacking:** Trays in cafeterias or food services are often stacked using a last-in, first-out approach.
- **Push-Down Dispensers:** Seen in napkin dispensers, tissues, or disposable cup dispensers.
- **Token Systems:** Tickets or tokens in a dispenser, ensuring the first one dispensed is the first used.
- **Cash Register Operations:** Represents the order in which cash or receipts are processed in a cash register.
- **Library Book Returns:** Books returned to a library are often stacked in a last-in, first-out manner until shelved.

1.3 Expression Evaluation: Infix and Postfix [2014 Spring]

What is infix?

- **Infix:** The typical mathematical form of expression that we encounter generally is known as infix notation. In infix form, an operator is written in between two operands.

  Example: An expression in the form of A * (B + C) / D is in infix form. This expression can be simply decoded as: "Add B and C, then multiply the result by A, and then divide it by D for the final answer."

What is Prefix?

- **Prefix:** In prefix expression, an operator is written before its operands. This notation is also known as "Polish notation".

  Example: The above expression can be written in the prefix form as / * A + B C D. This type of expression cannot be simply decoded as infix expressions.

What is Postfix?

- **Postfix:** In postfix expression, an operator is written after its operands. This notation is also known as "Reverse Polish notation".

  Example: The above expression can be written in the postfix form as A B C + * D /. This type of expression cannot be simply decoded as infix expressions.

Refer to the table below to understand these expressions with some examples:

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B–C	–+ABC	AB+C–
(A+B)*C–D	–*+ABCD	AB+C*D–

Prefix and postfix notions are methods of writing mathematical expressions without parentheses. Let's see the infix, postfix and prefix conversion.

Operator precedence

Operator	Name
(), {}, []	parenthesis
$, ^	Exponents
/, *	Division and Multiplication
+, -	Addition and Subtraction

⬆ As going to top it has higher precedence than the lower one.

Converting infix to prefix and postfix without using stack.

A$B+C–D+E/F/(G+H)

A$B+C-D+E/F/(G+H) To prefix	A$B+C-D+E/F/(G+H) To postfix
= A$B + C - D + E/F/ +GH	= A$B + C - D + E/F/ GH+
= $AB + C - D + /EF / +GH	= AB$ + C - D + EF/ / GH+
= $AB + C - D + //EF+GH	= AB$ + C - D + EF/GH+/
= +$ABC - D + //EF+GH	= AB$ + C - DEF/GH+/+
= -+$ABCD + //EF+GH	= AB$ + CDEF/GH+/+-
= + – +ABCD //EF+GH	= AB$CDEF/GH+/+ – +

1.4 Expression Conversion: Infix to Postfix and Postfix to Infix.

Algorithm for evaluating a postfix expression using stack

Step I: while (not end of the expression)
{

Step II: scan an element from an expression.

Step III: check if(element is operand)
{
Push the element to the stack &
Go to step II.
}

Step IV: check if(element is operator)

{

Pop operand 1 from stack

Pop operand 2 from stack

& perform the desired operation between the popped operands

Push the result into the stack &

Go to step II

}

}

Step VI: Pop the result from the stack and push it.

Step VII: stop/end.

Example 1: Evaluate the following postfix expression using stack.

623+-382/+*2$3+

Input	Operation	Stack status
6	Push 6	6
2	Push 2	6,2
3	Push 3	6,2,3
+	Pop 3	6,2
	Pop 2	6
	Push (2+3=5) Addition	6,5
–	Pop 5	6
	Pop 6	#(empty stack)
	Push (6–5=1) Subtraction	1
3	Push 3	1,3
8	Push 8	1,3,8
2	Push 2	1,3,8,2
/	Pop 2	1,3,8
	Pop 8	1,3
	Push (8/2=4) Division	1,3,4
+	Pop 4	1,3
	Pop 3	1
	Push (3+4=7) Addition	1,7
*	Pop 7	1
	Pop 1	# (empty)
	Push (1*7=7) Multiplication	7

Input	Operation	Stack status
2	Push 2	7,2
$	Pop 2	7
	Pop 7	# (empty stack)
	Push (7$2=49) Exponent	49
3	Push 3	49,3
+	Pop 3	49
	Pop 49	#(empty stack)
	Push (49+3=52) Addition	52
	Pop 52	#(empty stack)

∴ The result that is evaluated using the stack is 52.

Example 2: Evaluate the following postfix expression using stack.

ABC+*CBA-+* Where, A=1, B=2 and C=3

123+*321-+*

Input	Operation	Stack status
1	Push 1	1
2	Push 2	1,2
3	Push 3	1,2,3
+	Pop 3	1,2
	Pop 2	1
	push (2+3=5)Addition	1,5
*	Pop 5	1
	Pop 1	# (empty stack)
	Push (1*5=5)Multiplication	5
3	Push 3	5,3
2	Push 2	5,3,2
1	Push 1	5,3,2,1
-	Pop 1	5,3,2
	Pop 2	5,3
	Push (2-1=1)Subtraction	5,3,1
+	Pop 1	5,3
	Pop 3	5
	Push (3+1=4) Addition	5,4
*	Pop 4	5

Input	Operation	Stack status
	Pop 5	#(empty stack)
	Push (5*4=20) Multiplication	20
	Pop 20	#(empty stack)

∴ The result that is evaluated using the stack is 20.

Algorithm for converting infix to a postfix expression using stack [2014 Fall]

Step I: while (not end of the expression)
{
Step II: scan an element from an expression.
Step III: check if(element is operand)
{
Push the element to the stack &
Go to step II.
}
Step IV: check if(element is operator)
{
while(priority(element)<=priority(stack top) & (stack top)!= " (")
{
Pop an element from the stack and
print it
}
Push the element to the stack and
Go to step II.
}
Step V: if (element= " (")
{
Push elements on stack
Go to step II.
}
Step VI: if (element= ") ")
{
Pop and print element from stack until " (" is encountered
& than pop single " (" from the stack
}
Go to step II.
}

Step VII: while(stack is not empty)

{

Pop an element from the stack and print it.

}

Step VIII: stop/end.

Example: Convert the following infix expression to a postfix expression using stack.

A+(B/C-(D*E$F)+G)*H

Input	Postfix expression (Print)	Stack status
A	A	
+	A	+
(	A	+(
B	AB	+(
/	AB	+(/
C	ABC	+(/
-	ABC/	+([/ POPPED]
	ABC/	+(-
	ABC/	+(-
(	ABC/	+(-(
D	ABC/D	+(-(
*	ABC/D	+(-(*
E	ABC/DE	+(-(*
$	ABC/DE	+(-(*$
F	ABC/DEF	+(-(*$
)	ABC/DEF$*	+(-([Pop $ and * and print]
	ABC/DEF$*	+(- [pop single " ("]
+	ABC/DEF$* -	+(+
	ABC/DEF$* -	+(+
G	ABC/DEF$* - G	+(+ [POP +]
)	ABC/DEF$* - G+	+ [Pop single "("]
*	ABC/DEF$* - G+	+*
H	ABC/DEF$* - G+H	+* [pop + and *]
	ABC/DEF$* - G+H*+	# (empty stack)

∴ The result that is converted using stack is ABC/DEF$* - G+H*+

Example: Convert the following infix expression to a postfix expression using stack.

((A+B)-C*D/E)$*(H-I)*F+G

- operator>operand
- So, it cant be converted to the postfix expression

Advantages of postfix over infix expression [2018 Fall]

- Any formula can be expressed without parenthesis.
- It is very convenient for evaluating formulas on a computer with stacks.
- Postfix expression doesn't have the operator precedence.
- Postfix is slightly easier to evaluate.
- It reflects the order in which operations are performed.
- You need to worry about the left and right associativity.
- Postfix expression is simple to execute as a comparison to the infix expression it requires more operation to execute.
- In the postfix expression the overhead of brackets is not there while in the infix expression the overhead of brackets is there.
- The precedence of the operator has not affected the postfix expression while in the infix operator precedence is important.

Converting postfix to infix expression using stack.

[2017 Fall]

abc-+de-fg-h+/*

Expression	Stack
abc-+de-fg-h+/*	
-+de-fg-h+/*	a,b,c
+de-fg-h+/*	a, (b – c)
de-fg-h+/*	(a + b – c)
-fg-h+/*	(a + b – c), d, e
fg-h+/*	(a + b – c), (d – e)
-h+/*	(a + b – c), (d – e), f, g
h+/*	(a + b – c), (d – e), (f – g)
+/*	(a + b – c), (d – e), (f – g), h
/*	(a + b – c), (d – e), (f – g + h)
*	(a + b – c), (d – e)/(f – g + h)
# (empty)	(a + b – c)*(d – e)/(f – g + h)

∴ The result that is converted using stack is

(a+b-c)*(d-e)/(f-g+h)

2. Recursion

2.1 Recursion: A Problem Solving Technique

- Recursion is a process in which the function calls itself indirectly or directly in order to solve the problem.
- The function that performs the process of recursion is called a recursive function. There are certain problems that can be solved pretty easily with the help of a recursive algorithm.
- Recursion is a problem-solving technique in which a function calls itself in order to break down a complex problem into simpler, more manageable subproblems.
- The most common real-life application of recursion is when you are calculating how much money you have in a box filled with Rs. 100 notes.
- If there are too many notes, then you might just ask your friend to do the same work by dividing the entire stack into two.
- Once you both are done with counting, you will just add up both the results in order to get the total amount.
- Recursion and the concepts behind it are the heart and the power of computer systems. Computers are really good at **automating things** for us.
- Recursion can do one thing very well: repeating the same function with a slight change.

A recursive function has the capability to continue as an infinite loop. There are two properties that have to be defined for any recursive function to prevent it from going into an infinite loop. They are:

1. **Base criteria:** There has to be one predefined base condition. Whenever this base criterion is fulfilled, the function will stop calling itself.
2. **Progressive approach:** The recursive calls should consist of a progressive approach. Whenever a recursive call is made to the function, it should be reaching near the base condition.

Recursion is often referred to as a problem-solving technique in real-world examples because it provides an elegant and natural way to address problems that exhibit a recursive structure or can be decomposed into smaller, similar subproblems. Let's explore some real-world examples to illustrate why recursion is considered a valuable problem-solving technique:

File System Traversal:

- **Problem:** Navigating through a file system to find all files and subdirectories.

- **Recursive Solution:**
  - Start at a directory.
  - For each item in the directory, if it's a file, process it; if it's a directory, apply the same process recursively.
- **Why Recursion:**
  - Directories contain files and subdirectories, and the same process is applied to each subdirectory, mirroring the recursive structure.

Web Page Navigation:

- **Problem:** Traversing a website's hierarchy to extract information.
- **Recursive Solution:**
  - Start at the main page.
  - For each link on the page, if it leads to another page, apply the same process recursively.
- **Why Recursion:**
  - The structure of a website often mirrors a tree-like structure, with pages linking to other pages, creating a natural recursive pattern.

Organisational Structure:

- **Problem:** Understanding the hierarchy of an organisation.
  - **Recursive Solution:**
    - Start with the CEO.
    - For each department, apply the same process recursively to understand the hierarchy within that department.
  - **Why Recursion:**
    - Organisations often have nested structures, with departments containing teams, and teams containing sub-teams, creating a recursive hierarchy.

Mathematical Induction:

- **Problem:** Proving mathematical theorems by establishing a base case and showing that if it holds for smaller instances, it holds for larger instances.
- **Recursive Solution:**
  - Prove the theorem for a base case.
  - Prove that if the theorem holds for a given case, it holds for the next case.
- **Why Recursion:**
  - Mathematical induction follows a recursive pattern, ensuring that the theorem holds for all cases.

In these examples, recursion is called a problem-solving technique because it helps break down complex problems into simpler, more manageable subproblems, allowing for a more elegant and intuitive solution. The recursive approach often aligns with the natural structure of the problems encountered in the real world, making it a powerful tool for problem-solving in various domains.

2.2 Principle of Recursion

There are two properties that have to be defined for any recursive function to prevent it from going into an infinite loop. They are:

- **Base criteria:** There has to be one predefined base condition. Whenever this base criterion is fulfilled, the function will stop calling itself.
- **Progressive approach:** The recursive calls should consist of a progressive approach. Whenever a recursive call is made to the function, it should be reaching near the base condition.
- The principle of recursion involves solving a problem by breaking it down into smaller instances of the same problem. In a recursive approach, a function calls itself with a smaller input, and this process continues until a base case is reached.
- The solution to the base case is then combined to obtain the solution to the original problem. Recursion is a powerful concept in computer science and mathematics, providing a concise way to express repetitive structures or problems.

The Three Laws of Recursion

Like the robots of Asimov, all recursive algorithms must obey three important laws:

1. A recursive algorithm must call itself, recursively.
2. A recursive algorithm must have a **base case.**
3. A recursive algorithm must change its state and move toward the base case.

A base case is the condition that allows the algorithm to stop recursion.

- A base case is typically a problem that is small enough to solve directly.
- In the factorial algorithm the base case is n=1.

We must arrange for a change of state that moves the algorithm toward the base case.

- A change of state means that some data that the algorithm is using is modified.

- Usually the data that represents our problem gets smaller in some way.
- In the factorial n decreases.

2.3 Recursive Algorithm

- A recursive algorithm is an algorithm that calls itself in order to solve a problem.
- It is a programming or computational approach where a function or method calls itself with a smaller or simpler input, making use of the same logic to solve the problem iteratively.

1. Greatest Common Divisor

- The greatest common divisor (GCD) of two or more numbers is the greatest common factor number that divides them, exactly. It is also called the highest common factor (HCF).
- If a and b are two numbers then the greatest common divisor of both the numbers is denoted by gcd(a, b).
- To find the gcd of numbers, we need to list all the factors of the numbers and find the largest common factor.
- GCD is the greatest common factor of two or more numbers.
- A factor that is the highest among the numbers.

SOLVED EXAMPLES

Example 1: Find the greatest common divisor (or HCF) of 128 and 96.

Solution:

By the method of prime factorisation,

$128 = 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$

$96 = 2 \times 2 \times 2 \times 2 \times 2 \times 3$

$HCF\ (128, 96) = 2 \times 2 \times 2 \times 2 \times 2 = 32$

(OR)

By the method of division,

```
96) 128 (1
    96
   ----
    32) 96 (3
        96
       ----
         0
```

Hence, 32 is the HCF of 128 and 96.

(OR)

By Euclid's division algorithm,

128 = 96 x 1 + 32

96 = 32 x 3 + 0

Hence, the HCF of 128 and 96 is 32.

Example 2: Two rods are 22 m and 26 m long. The rods are to be cut into pieces of
Equal length. Find the maximum length of each piece.

Solution:

HCF or GCD is the required length of each piece.

22 = 2 x 11

26 = 2 x 13

HCF or the greatest common divisor = 2

Hence, the required maximum length of each piece is 2 m.

The greatest common factor of 3 numbers

We can find the greatest common factor/divisor of 3 three numbers by

the prime factorisation method as shown below.

Example 3: Find the greatest common factor of 24, 148 and 36.

Solution:

Prime factorisation of given numbers is

24 = 2 × 2 × 2 × 3

148 = 2 × 2 × 37

36 = 2 × 2 × 3 × 3

Greatest common factor = 2 × 2 = 4

Recursive Algorithm for GCD

Let us consider the two positive integers x and y.

```
GCD(x, y)
Begin
   if y = = 0 then
      return x;
   else
      Call: GCD(y, x%y);
   endif
End
```

WAP to find GCD of two positive integers using Recursion.

```
#include <stdio.h>
#include<conio.h>
```

```
    int gcd(int number1, int number2);
int main()
{
    int number1;
    printf("enter the first number");
    scanf("%d",&number1);

    int number2;
    printf("enter the second number to be inserted");
    scanf("%d",&number2);

    printf("GCD of %d and %d is %d", number1, number2,

}
int gcd(int number1, int number2)
    {
    if (number2==0)
    return number1;
    else
    return(number2, number1%number2);
    }
```

2. Sum of Natural Number

- The sum of natural numbers is the result of adding all the positive integers up to a given positive integer *n*. It can be calculated using the formula:

  $\text{sum} = \frac{n(n+1)}{2}$

  This formula is a concise way to find the sum of the first *n* natural numbers.
- **Example 1**: Let n = 5 Therefore, the sum of the first 5 natural numbers = 1 + 2 + 3 + 4 + 5 = 15.Thus, the output is 15.
- **Example 2**: Let n = 7 Therefore, the sum of the first 7 natural numbers = 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28. Thus, the output is 28.
- **Example 3**: Let n = 6 Therefore, the sum of the first 6 natural numbers = 1 + 2 + 3 + 4 + 5 + 6 = 21. Thus, the output is 21.

Recursive Algorithm for sum of n natural number

Let us consider a 'n' positive integer.

Algorithm findSum(n):

Input: An integer n

Output: The sum of integers from 1 to n

if n <= 1 then

return n

else

return n + findSum(n-1)

end if

End Algorithm

This algorithm recursively calculates the sum of integers from 1 to n. If n is 1 or less, it returns n itself. Otherwise, it adds n to the sum of integers from 1 to n-1.

WAP to find the sum of natural numbers using Recursion.

```
#include <stdio.h>
#include <conio.h>
int sum(int number);
int main() {
  int number;
  printf("Enter the number: ");
  scanf("%d", &number);
  printf("Sum of natural numbers up to %d is %d\n", number,
   sum(number));
  return 0;
}
int sum(int number) {
  if (number <= 1)
    return number;
  else
    return number + sum(number - 1);
}
```

3. Factorial of a positive integer

- Factorial of a positive integer (number) is the sum of multiplication of all the integers smaller than that positive integer.
- For example, factorial of 5! is 5 * 4 * 3 * 2 * 1 which equals 120.

Algorithm to find factorial of an integer:

Step 1: Start

Step 2: Read number n

Step 3: Call factorial (n)

Step 4: Print factorial f

Step 5: Stop

Factorial (n)

Step 1: If n==1 then return 1 //stopping condition (Base case)

Step 2: Else f= n* factorial (n-1)

Step 3: Return f

WAP to find the factorial of a number using Recursion.

```
#include<stdio.h>
#include<conio.h>
long int factorial(int number);
void main()
{
    int number;
    printf("Enter the number");
    scanf("%d", &number);

    printf("Factorial of a %d is %d\n", number, factorial (number));
    getch();
}
long int factorial(int number)
{
    if (number==0 || number==1)
      return 1;
    else
      return number*factorial(number-1);
}
```

4. Fibonacci sequence [2013 Fall]

A Fibonacci sequence is the sequence of integers in which each element in the sequence is the sum of the two previous elements.

The Fibonacci series starts from two numbers – F0 & F1. The initial values of F0 & F1 can be taken 0, 1 or 1, 1 respectively.

Fn = Fn-1 + Fn-2

E.g.

F8 = 0, 1, 1, 2, 3, 8, 13 or, F8 = 1, 1, 2, 3, 5, 8, 13, 21

Recursive algorithm to get Fibonacci sequence:

1. START
2. Input the non-negative integer 'n'
3. If (n==o || n==1)
   return n;
   else
   return fib(n-1)+fib(n-2);
4. Print, nth Fibonacci number
5. END

WAP to find the fibonacci sequence of a number using Recursion.

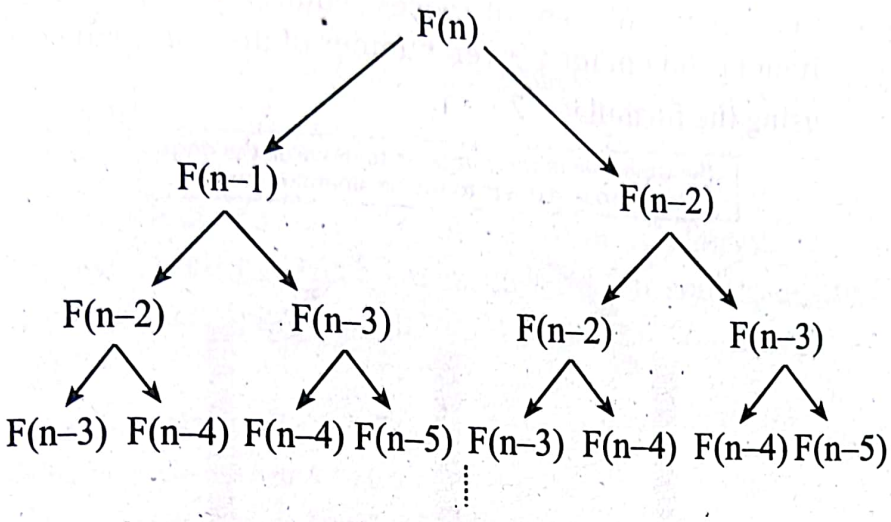
```
#include <stdio.h>
#include <conio.h>
int fibo(int number);
int main() {
  int number, i;
  printf("Enter how many terms you want to generate: ");
  scanf("%d", &number);
  printf("Fibonacci series up to %d terms:\n", number);
  for (i = 1; i <= number; i++) {
    printf("%d\t", fibo(i));
  }
  printf("\n");
  getch(); // Waiting for a key press before closing the console
   window
  return 0;
}
int fibo(int number) {
  if (number == 1)
    return 0;
  else if (number == 2)
    return 1;
  else
    return fibo(number - 1) + fibo(number - 2);
}
```

A Recursion Tree for the Fibonacci series.

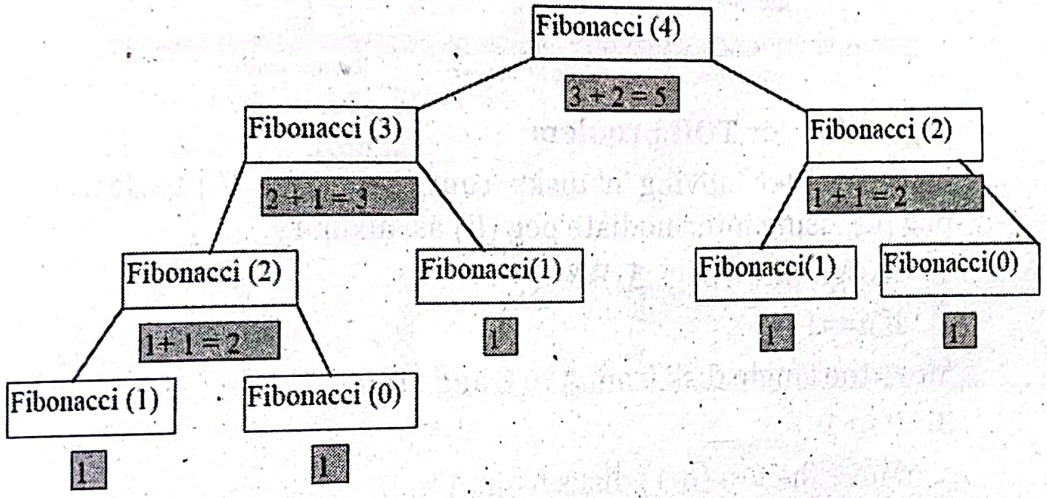
A recursion tree is a tree that is generated by tracing the execution of a recursive algorithm. A recursion tree shows the

relationship between calls to the algorithm. Each item in the tree represents a call to the algorithm.

Fibonacci Recursion Tree



Recursion tree using algorithm Fibonacci with N=4as:



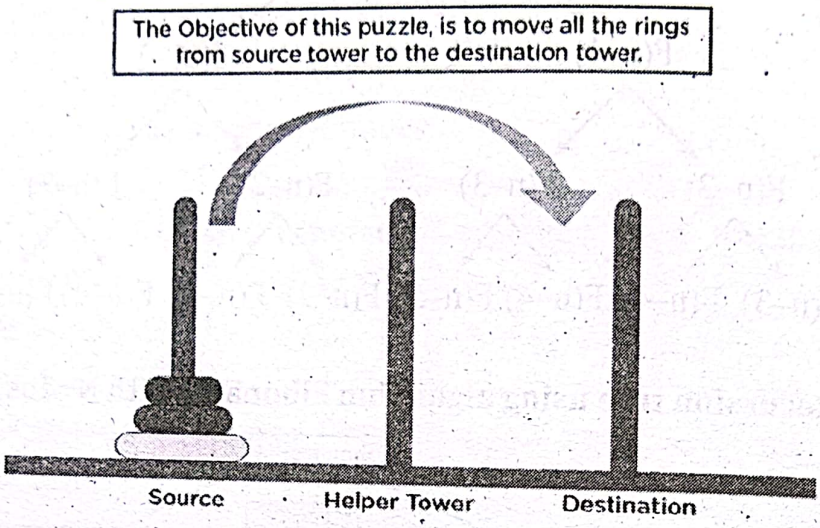
Each unshaded box shows a call to the algorithm Fibonacci with the input value of N in parentheses. Each shaded box shows the value that is returned from the call. Calls to algorithm Fibonacci are made until a call is made with input value one or zero. When a call is made with input value one or zero, a one is returned. When a call is made with N > 1, two calls are made to algorithm Fibonacci, and the value that is returned is the sum of the values from the two calls. The final number that is returned is 5. Thus the 4th number in the Fibonacci series is 5.

[2013 Fall]

5. Tower of Hanoi (TOH)

- Tower of Hanoi (TOH) is a mathematical puzzle which consists of three pegs named as origin, intermediate and destination and more than one disks.
- These disks are of different sizes and the smaller one sits over the larger one.

- In this problem we transfer all disks from origin peg to destination peg using intermediate peg for temporary storage and move only one disk at a time.
- The number of steps or moves required to solve the Tower of Hanoi problem for a given number of disks *n* can be calculated using the formula: $2^n - 1$



Algorithm for TOH problem:

Let's consider moving 'n' disks from source peg (A) to destination peg (C), using intermediate peg (B) as auxiliary.

1. Assign three pegs A, B & C
2. If n==1

Move the single disk from A to C and stop.

3. If n>1

a. Move the top (n-1) disks from A to B.
b. Move the remaining disks from A to C
c. Move the (n-1) disks from B to C
4. Terminate

WAP for finding the tower of hanoi of n disk using Recursion

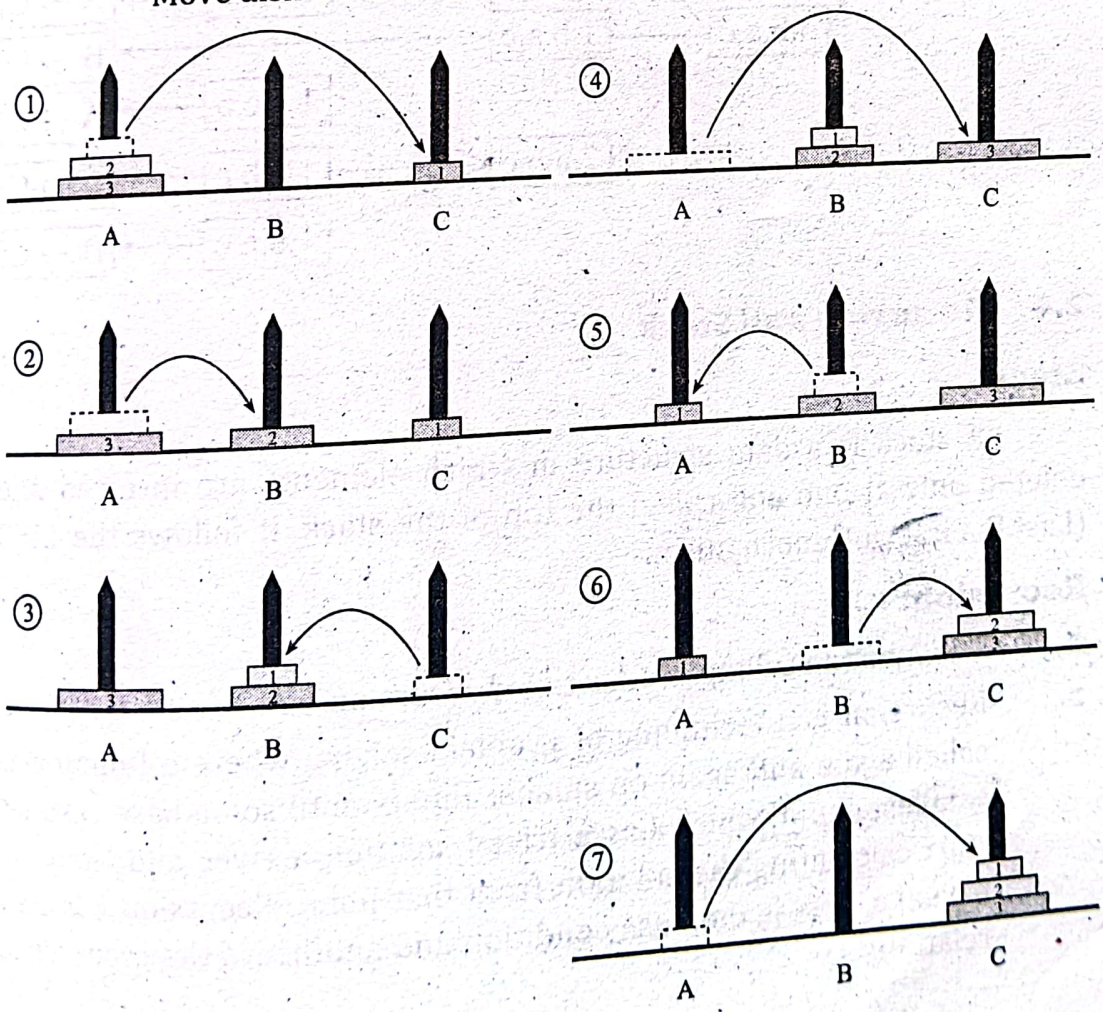
```
#include <stdio.h>
void towerOfHanoi(int n, char source, char auxiliary, char destination);
int main() {
  int n;

  printf("Enter the number of disks: ");
  scanf("%d", &n);
  printf("Steps to solve Tower of Hanoi with %d disks:\n", n);
  towerOfHanoi(n, 'A', 'B', 'C');
  return 0;
```

```
}
void towerOfHanoi(int n, char source, char auxiliary, char
    destination) {
  if (n == 1) {
    printf("Move disk 1 from %c to %c\n", source, destination);
    return;
  }
  towerOfHanoi(n - 1, source, destination, auxiliary);
  printf("Move disk %d from %c to %c\n", n, source, destination);
  towerOfHanoi(n - 1, auxiliary, source, destination);
}
```

Steps for solving 3 disk TOH

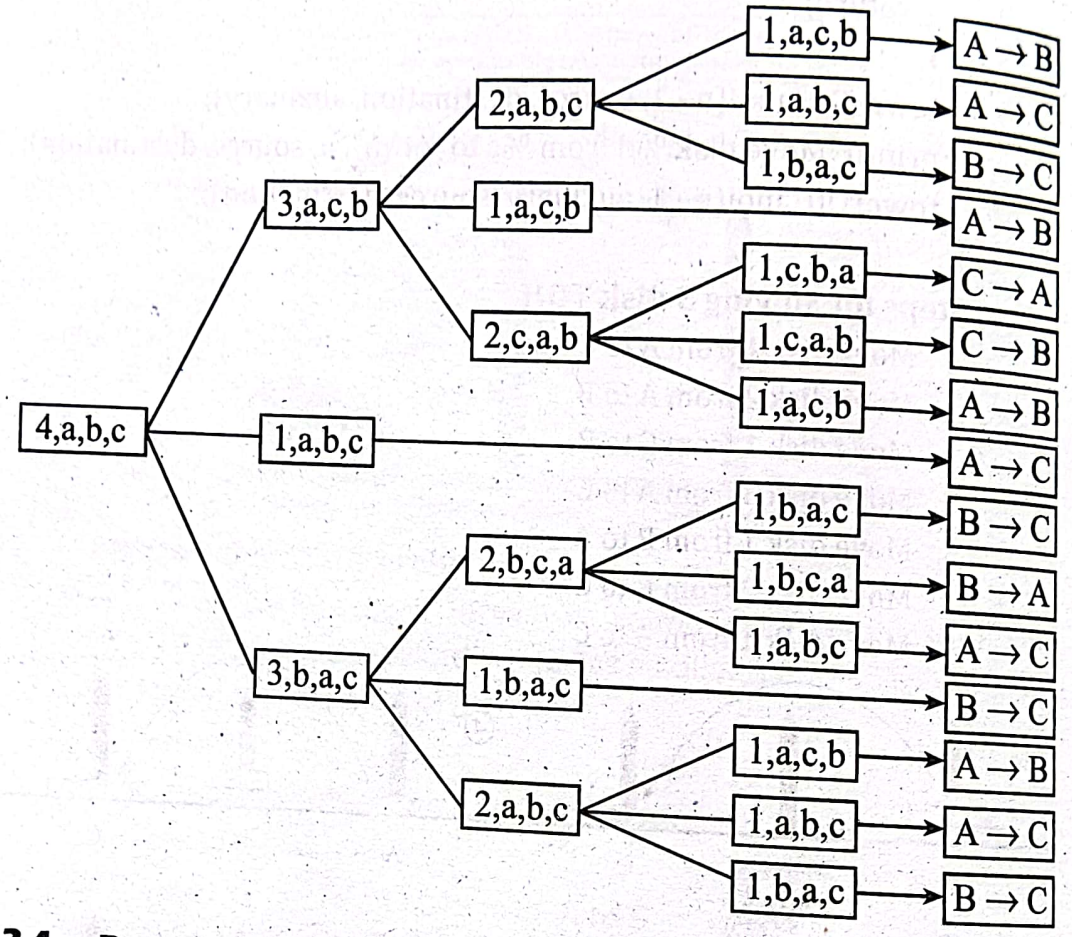
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C



Recursion Tree for TOH

1. Move Tower(N-1, BEG, END,AUX)
2. Move Tower(1, BEG, AUX, END) à(BEG à END)
3. Move Tower (N-1, AUX, BEG, END)

Recursion Tree when no. of disks are 4 as:



2.4 Recursion and Stack

Stack:

A stack is a data structure in which elements are inserted and deleted only at one end called the top of the stack. It follows the LIFO (Last In First Out) mechanism.

Recursion:

- The function calling itself is called recursion.
- Recursion is a technique of problem-solving where a function is called again and again on smaller inputs until some base case i.e. smallest input which has a trivial solution arrives and then we start calculating the solution from that point. Recursion has two parts i.e. first is the base condition and another is the recurrence relation.

- Let's understand them one by one using an example of the factorial of a number.

Recurrence:

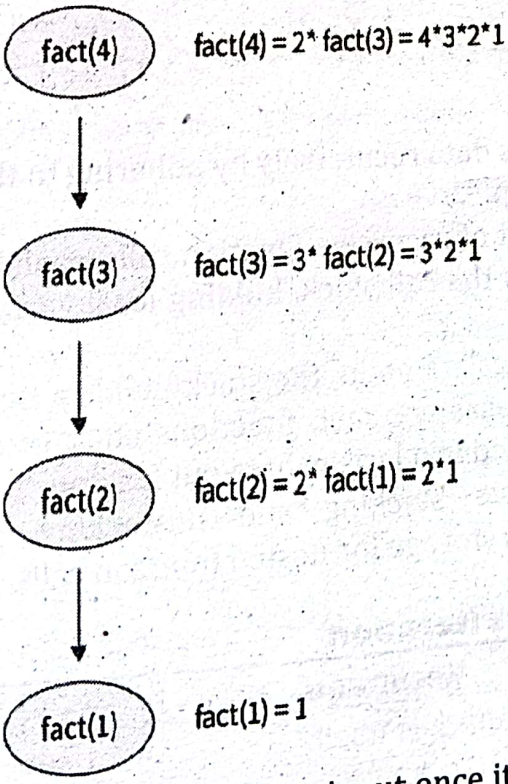
- Recurrence is the actual relationship between the same function on different sizes of inputs i.e. we generally compute the solution of larger input using smaller input.
- For example calculating the factorial of a number, in this problem let's say we need to calculate the factorial of a number N and we create a helper function say fact(N) which returns the factorial of a number N now we can see that the factorial of a number N using this function can also be represented as

  fact(N) = N * fact(N-1)

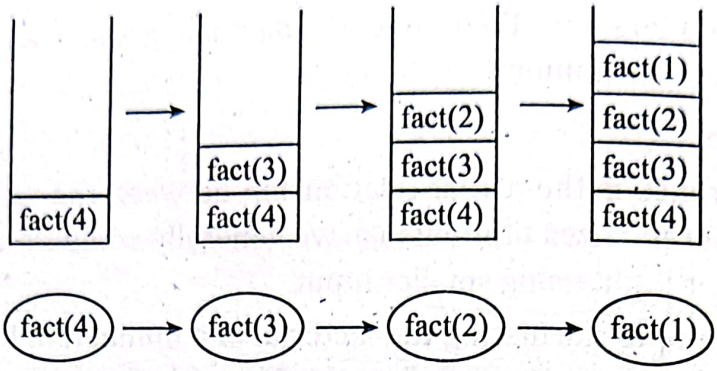
  The function fact(N) calls itself but with a smaller input the above equation is called recurrence relation.

Base condition:

- This is the condition where the input size given to the input is so small that the solution is very trivial to it. In the case of the above factorial problem we can see that the base condition is fact(1) i.e. on the input N=1 we know the solution is 1.

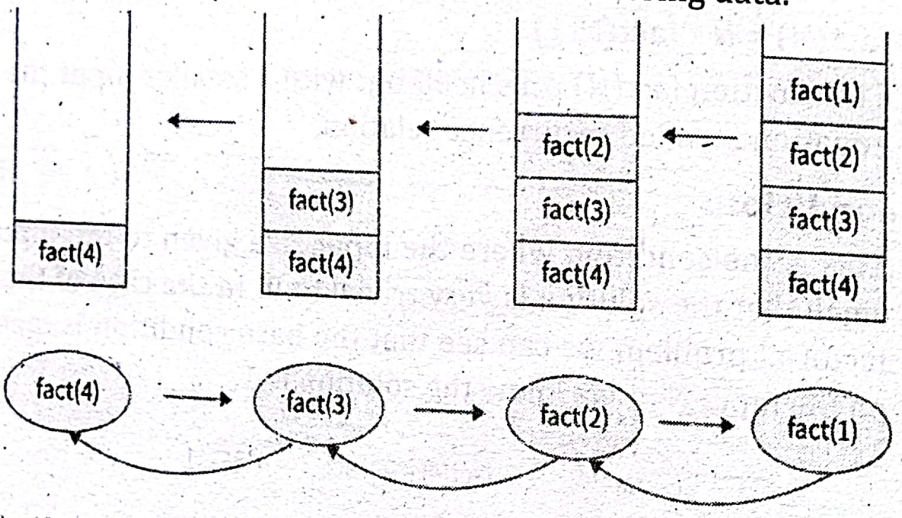


- Recursion backtracks to previous input once it finds the base case and the temporary function calls which are pending are stored in the stack data structure in the memory as follows.



- with each function call, the stack keeps filling until the base case arrives which is fact(1) = 1 in this case. After that, each function call is evaluated in the last in first out order.

Here is how Stack uses the Recursion for storing data.



fact(4) = 4* fact(3) = 4*6 = 24 fact(3) = 3* fact(2) = 3*2 = 6 fact(2) = 2* fact(1) = 2*1 = 2. BASE CASE : fact(1) = 1

- A stack stores data recursively by adhering to the Last In, First Out (LIFO) principle.
- In the context of recursive function calls, each function call adds a new frame to the call stack, holding local variables and execution information.
- As recursive calls unfold, the stack builds a nested structure, and when base cases are met, functions start to return, causing the stack to unwind in a last-in, first-out fashion.
- This recursive stacking and unstacking process efficiently manages data storage for nested function calls.

2.5 Recursion vs Iteration

[2013 Spring]

Property	Recursion	Iteration
Definition	Function calls itself.	A set of instruction 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.

Property	Recursion	Iteration
Usage	Used when code size need 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).
Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Does not use stack.
Overhead	Recursion possesses the overhead of repeated function calls.	No overhead of repeated function call.
Speed	Slow in execution.	Fast in execution.

Example: Factorial using recursion

```
#include <stdio.h>
int factorial_recursive(int n) {
  return (n == 0 || n == 1) ? 1 : n *
factorial_recursive(n - 1);
}
int main() {
  printf("Factorial using recursion:
%d\n", factorial_recursive(5));
  return 0;
}
```

Example: Factorial using iteration

```
#include <stdio.h>
int factorial_iterative(int n) {
  int result = 1;
  for (int i = 1; i <= n; ++i) result *=
i;
  return result;
}
int main() {
  printf("Factorial using iteration:
%d\n", factorial_iterative(5));
  return 0;
}
```

2.6 Recursive Data Structure

- A recursive data structure contains references to itself, such as a list or tree.
- These types of structures are dynamic data structures where the structure can theoretically grow to an infinite length.

i. Linked Lists:

- Elements are sequentially connected, where each element (node) points to the next one.
- Recursive operations often involve traversing the list by addressing one element at a time.

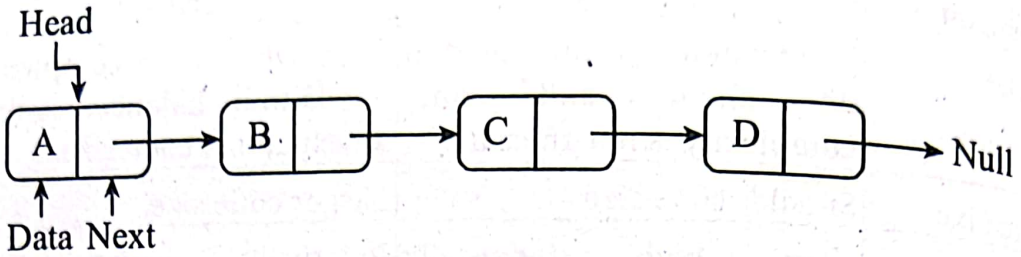


Fig: Linked List Representation

ii. Trees:

- Tree structures consist of nodes with parent-child relationships, forming a hierarchical arrangement.
- Recursive algorithms for trees commonly involve traversing nodes, such as in depth-first or breadth-first searches.

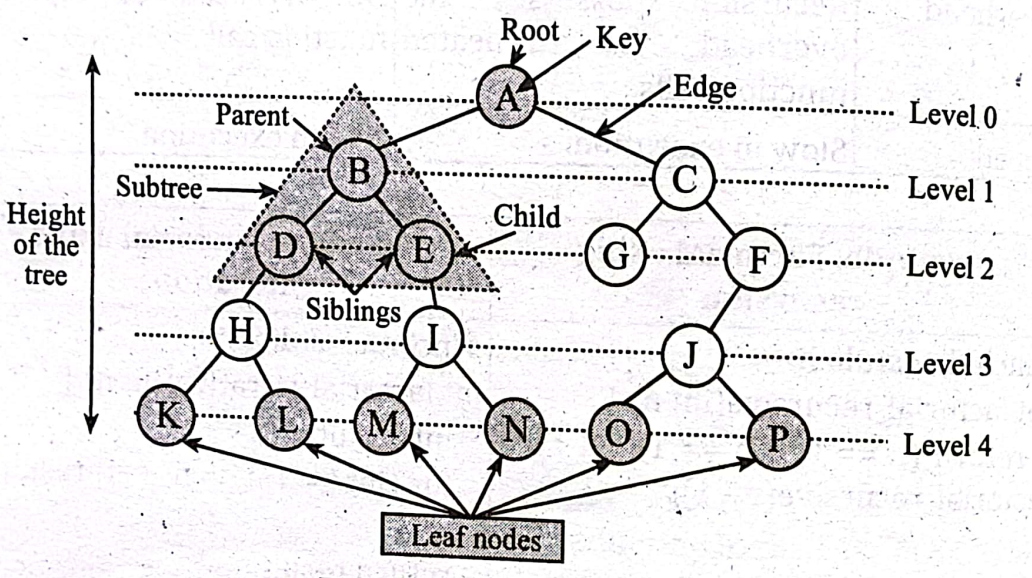


Fig.: Tree data structure

iii. Filesystems:

- Files and directories are organised in a hierarchical tree structure.
- Recursive operations in file systems often include tasks like traversing directories or copying entire directory structures.

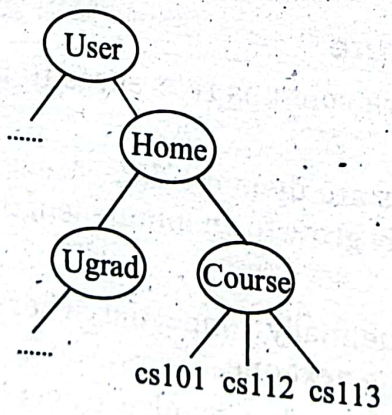


Fig.: File system

iv. Graph:

- A graph comprises nodes (vertices) and connections between them (edges).
- Recursive algorithms for graphs might focus on exploring paths, finding connected components, or traversing the graph in various ways.

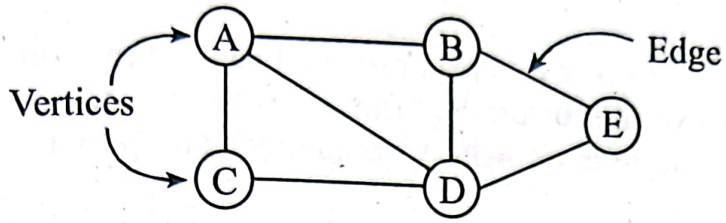


Fig.: Graph with edge and vertices

2.7 Types of Recursion

There are four different types of recursive algorithms, you will look at them one by one.

i. Direct Recursion

- A function is called direct recursive if it calls itself in its function body repeatedly.
- To better understand this definition, look at the structure of a direct recursive program.

```
int fun(int z){
 fun(z-1);  //Recursive call
}
```

- In this program, you have a method named fun that calls itself again in its function body. Thus, you can say that it is direct recursive.

ii. Indirect Recursion

- The recursion in which the function calls itself via another function is called indirect recursion.
- Now, look at the indirect recursive program structure.

```
int fun1(int z){     int fun2(int y){
 fun2(z-1);          fun1(y-2)
}                    }
```

- In this example, you can see that the function fun1 explicitly calls fun2, which is invoking fun1 again. Hence, you can say that this is an example of indirect recursion.

iii. Tailed Recursion

- A recursive function is said to be tail-recursive if the recursive call is the last execution done by the function. Let's try to understand this definition with the help of an example.

```
int fun(int z)
{
 printf("%d",z);
 fun(z-1);
 //Recursive call is last executed statement
}
```

- If you observe this program, you can see that the last line ADI will execute for method fun is a recursive call. And because of that, there is no need to remember any previous state of the program.

iv. Non-Tailed Recursion

A recursive function is said to be non-tail recursive if the recursion call is not the last thing done by the function. After returning back, there is something left to evaluate. Now, consider this example.

```
int fun(int z)
{
 fun(z-1);
 printf("%d",z);
 //Recursive call is not the last executed statement
}
```

- In this function, you can observe that there is another operation after the recursive call. Hence the ADI will have to memorise the previous state inside this method block. That is why this program can be considered non-tail recursive.

2.8 Application of a Recursion

1. **Algorithmic Problem Solving:** Many algorithms are naturally expressed using recursion. Examples include recursive implementations of sorting algorithms (e.g., quicksort, mergesort) and searching algorithms (e.g., binary search).
2. **Tree and Graph Traversal:** Recursive algorithms are commonly used for traversing trees and graphs. Depth-first and breadth-first searches are often implemented using recursion, making it easier to navigate complex structures.
3. **Dynamic Programming:** Recursive solutions are frequently used in dynamic programming, where a problem is broken down into smaller subproblems. Memoization or caching of results from recursive calls can be employed to optimise performance.
4. **Mathematical Calculations:** Recursion is employed in mathematical calculations, such as computing factorials, Fibonacci sequences, and solving problems related to combinatorics.

5. **File System Operations:** Operations on file systems, like directory traversal or searching for specific files, can be implemented using recursion. The hierarchical nature of directories makes recursion a natural fit for these tasks.
6. **Parsing and Syntax Analysis:** Recursive descent parsing is a technique often used in the implementation of parsers for programming languages. The grammar rules of a language are recursively applied to analyse and interpret the syntax of code.
7. **Fractals:** Generating fractal patterns, such as the Mandelbrot set, often involves recursion. Each part of the fractal is defined in terms of smaller copies of itself.
8. **Backtracking Algorithms:** Backtracking algorithms, used in problems like the N-Queens problem or the Sudoku solver, frequently employ recursion to explore different possibilities and backtrack when necessary.

Recursion simplifies the expression of certain algorithms and can lead to elegant and concise code when used appropriately. However, it's essential to be mindful of potential stack overflow issues in deep recursive calls and to consider iterative solutions for cases where recursion might be less efficient or not suitable.

Advantages of recursion

[2013 Spring]

1. The code may be easier to write.
2. To solve such problems which are naturally recursive such as the tower of Hanoi.
3. Reduce unnecessary calling of function.
4. Extremely useful when applying the same solution.
5. Recursion reduces the length of code.
6. It is very useful in solving the data structure problem.
7. Stacks evolutions and infix, prefix, postfix evaluations etc.

Disadvantages of recursion

[2013 Spring]

1. Recursive functions are generally slower than non-recursive functions.
2. It may require a lot of memory space to hold intermediate results on the system stacks.
3. Hard to analyse 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.

1. **Write the advantages of Postfix expression over the Infix expression while processing by computer system. Convert the given expression into Postfix expression showing the content of the stack at each step:**
(A+BC/D)+EF-(G*H+I-J) **[2021 Fall]**

⇒ **Advantages of Postfix Expression:**

1. **No Need for Parentheses:** Postfix expressions do not require parentheses to dictate the order of operations, making them simpler to evaluate.
2. **Efficient Evaluation:** Postfix expressions can be directly evaluated using a stack, without needing to consider operator precedence and associativity rules.
3. **Uniformity:** Each operator is applied to a fixed number of operands, leading to consistent and predictable evaluation.

Conversion to Postfix Expression

Infix: (A+BC/D)+EF-(G*H+I-J)

Postfix: ABCD/+EF+GH*I+J-

Conversion Steps:

Step	Stack	Output
1	(	
2	(A	
3	(A+	
4	(A+BC	
5	(A+B(C/D	
6	(A+BCD/	
7	(A+BCD/+	
8	(A+BCD/+EF	
9	(A+BCD/+EF+	
10	(A+BCD/+EF+(GH	
11	(A+BCD/+EF+(GH*	
12	(A+BCD/+EF+(GH*+	
13	(A+BCD/+EF+(GH*+I	
14	(A+BCD/+EF+(GH*+I+	
15	(A+BCD/+EF+(GH*+I+J-	
16	(A+BCD/+EF+(GH*+I+J-	
17	(A+BCD/+EF+(GH*+I+J-	

Final Postfix: ABCD/+EF+GH*I+J-

2. **Explain how you use stack in the conversion of the following infix expression to a postfix expression using a stack: A+B-C/(DF)-HI.** [2023 Spring]

⇒ **Conversion of Infix to Postfix:**

1. Initial Expression: A+B-C/(DF)-HI
2. Stack: []

Steps:

1. Read 'A': Operand, add to output.
   - Output: A
   - Stack: []
2. Read '+': Operator, push to stack.
   - Output: A
   - Stack: [+]
3. Read 'B': Operand, add to output.
   - Output: AB
   - Stack: [+]
4. Read '-': Operator, pop and add to output until stack is empty or contains lower precedence.
   - Output: AB+
   - Stack: [-]
5. Read 'C': Operand, add to output.
   - Output: AB+C
   - Stack: [-]
6. Read '/': Operator, push to stack.
   - Output: AB+C
   - Stack: [- /]
7. Read '(': Push to stack.
   - Output: AB+C
   - Stack: [- / (]
8. Read 'D': Operand, add to output.
   - Output: AB+CD
   - Stack: [- / (]
9. Read 'F': Operand, add to output.
   - Output: AB+CDF
   - Stack: [- / (]
10. Read ')': Pop and add to output until '(' is found.
    - Output: AB+CDF /

- Stack: [-]

11. Read '-': Operator, pop and add to output until stack is empty or contains lower precedence.
    - Output: AB+CDF / -
    - Stack: []
12. Read 'H': Operand, add to output.
    - Output: AB+CDF / -H
    - Stack: []
13. Read 'I': Operand, add to output.
    - Output: AB+CDF / -HI
    - Stack: []

Final Postfix Expression: AB+CDF / -HI-

---

3. What are the advantages of postfix expression over infix expression? Write an algorithm to convert postfix expression to prefix expression. Convert the following postfix expression into prefix expression:
AB+CD* **[2021 Fall]**

⇒ **Advantages of Postfix Expression over Infix Expression**

- **No Parentheses Needed:** Postfix eliminates the need for parentheses, reducing complexity.
- **Easier Evaluation:** Postfix expressions can be evaluated using a stack without the need for operator precedence rules.

Algorithm to Convert Postfix to Prefix:

1. Initialize an empty stack.
2. Traverse the postfix expression from left to right.
3. For each character:
    - If the character is an operand, push it onto the stack.
    - If the character is an operator, pop the top two elements from the stack. Concatenate the operator with these two operands and push the resulting string back onto the stack.
4. The final element in the stack is the prefix expression.

Example:

- Postfix: AB+CD*
- Conversion Steps:
    1. Push A
    2. Push B
    3. Encounter +: Pop B and A, create +AB, push +AB

4. Push C
5. Push D
6. Encounter *: Pop D and C, create *CD, push *CD
7. Encounter +: Pop *CD and +AB, create + +ABCD

- **Prefix:** + +AB*CD

---

④ **In which conditions are recursive algorithms suitable? Explain with the problem of printing Fibonacci series.**

[2021 Fall]

⇒ **Conditions Suitable for Recursive Algorithms**

- **Conditions:**
  - Problems that can be divided into smaller sub-problems of the same type.
  - Problems with a clear base case and recursive case.
  - Problems where overlapping sub-problems or optimal substructure are present.

  Example: Fibonacci Series:
- **Recursive Algorithm:**

```
int fibonacci(int n) {
  if (n <= 1)
    return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

---

5. **How is a divide and conquer strategy used to recursively solve the Tower of Brahma (also called Tower of Hanoi)? Explain in detail. Also implement it using C or C++ code.**

[Spring 2023]

⇒ **Divide and Conquer Strategy:**

The Tower of Hanoi problem is solved using the divide and conquer strategy, which involves breaking down the problem into smaller subproblems and solving them recursively.

Steps:

1. Move the top n−1n-1n−1 disks from source to auxiliary rod.
2. Move the nth disk from source to destination rod.
3. Move the n−1n-1n−1 disks from auxiliary rod to destination rod.

Algorithm:

1. **Base Case:** If there is only one disk, move it from source to destination.

2. **Recursive Case:**
   - Move n−1n-1n−1 disks from source to auxiliary using destination as buffer.
   - Move the nth disk from source to destination.
   - Move n−1n-1n−1 disks from auxiliary to destination using source as buffer.

C++ Code Implementation:

```
#include <iostream>
using namespace std;
void towerOfHanoi(int n, char source, char auxiliary, char
    destination) {
  if (n == 1) {
    cout << "Move disk 1 from " << source << " to " << destination
    << endl;
    return;
  }
  towerOfHanoi(n-1, source, destination, auxiliary);
  cout << "Move disk " << n << " from " << source << " to " <<
   destination << endl;
  towerOfHanoi(n-1, auxiliary, source, destination);
}
int main() {
  int n = 3; // Number of disks
  towerOfHanoi(n, 'A', 'B', 'C');
  return 0;
}
```

Example: For n=3n = 3n=3:

1. Move 2 disks from A to B using C.
2. Move disk 3 from A to C.
3. Move 2 disks from B to C using A.

---

6. What is the principle of a stack? Explain any three applications of stack in computer science with examples. [2023 Spring]

⇒ **a. Principle of a Stack**

- **Definition:** A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, where the last element added to the stack is the first one to be removed.

- **Basic Operations:** Push (add an element), Pop (remove the top element), Peek (look at the top element without removing it).

b. Applications of Stack

1. **Function Call Management:**

   Example: In recursion, each function call is pushed onto the call stack. When a function returns, it is popped from the stack.

2. **Expression Evaluation:**

   Example: Converting infix expressions to postfix and evaluating postfix expressions using stacks.

3. **Syntax Parsing:**

   Example: Balancing parentheses in expressions and validating HTML/XML tags.

---

7. Explain how divide and conquer strategy is used to design an algorithm with a suitable example

⇒ **a. Explanation and Example**

Definition: A strategy that divides a problem into smaller subproblems, solves each subproblem recursively, and combines the solutions.

Example: Merge Sort, which divides the array into halves, sorts each half, and merges the sorted halves.

---

8. Why do you need to convert an infix expression to postfix? How do you use a stack to convert the following expression to postfix notation: (A+B)*C-(D^E)/F [2022 Fall]

⇒ **a. Importance of Conversion**

Operator Precedence: Postfix (or Reverse Polish Notation) does not require parentheses to define operator precedence, making it easier for computer algorithms to parse and evaluate expressions.

b. Conversion of (A+B)*C-(D^E)/F using Stack

*[**Note:** Using C++ to solve it mathematically, see algorithm and solve like earlier in syllabus covered.]*

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;
// Function to return precedence of operators
```

```
int precedence(char op) {
  if(op == '+'||op == '-') return 1;
  if(op == '*'||op == '/') return 2;
  if(op == '^') return 3;
  return 0;
}

// Function to convert infix to postfix
string infixToPostfix(string infix) {
  stack<char> s;
  string postfix;
  for(char& c : infix) {
    if(isalnum(c)) {
      postfix += c;
    } else if(c == '(') {
      s.push(c);
    } else if(c == ')') {
      while(!s.empty() && s.top() != '(') {
        postfix += s.top();
        s.pop();
      }
      s.pop(); // pop '('
    } else {
    while(!s.empty() && precedence(s.top()) >= precedence(c))
    {
        postfix += s.top();
        s.pop();
      }
      s.push(c);
    }
  }
  while(!s.empty()) {
    postfix += s.top();
    s.pop();
  }
  return postfix;
}
```

```
int main() {
    string infix = "(A+B)*C-(D^E)/F";
    cout << "Postfix: " << infixToPostfix(infix) << endl;
    return 0;
}
```

Output:

Postfix: AB+C*DE^F/-

---

8. What is the advantage of using recursive algorithms? Implement the recursive algorithms to solve the Tower of Hanoi problems using C or C++ code. [2022 Fall]

⇒ **Recursive Algorithms and Tower of Hanoi**

a. Advantage of Recursive Algorithms

- **Simplified Code:** Makes the implementation of complex algorithms more understandable and maintainable.
- **Natural Fit:** Ideal for problems that have a recursive structure, like tree traversals, backtracking problems, etc.

b. Tower of Hanoi Implementation in C++

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

void towerOfHanoi(int n, char from_rod, char to_rod, char
    aux_rod) {
  if (n == 1) {
    cout << "Move disk 1 from rod " << from_rod << " to rod "
    << to_rod << endl;
    return;
  }
  towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
  cout << "Move disk " << n << " from rod " << from_rod << " to
    rod " << to_rod << endl;
  towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

int main() {
  int n = 4; // Number of disks
  towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
  return 0;
}
```