Introduction to Data Structure

Data

* Data is basically a fact or entity. Data are raw facts whereas information is processed data.

* Data refers to value or set of values
e.g:- Marks obtained by the students

* Data can be classified into two types - numerical data and alphanumerical data. These two data types specify the nature of the data items and it is used certain operations. The numerical data type can be any numbers with decimal or without decimal point, for example, integers (10, 20, 30) and floating point numbers (10.25, 16.5 etc). The alphanumerical data can be single character or group of characters, for example character constants ('A', 'B', 'C', etc) and string constants ("Simon", "Arun", "Arjun" etc)

When a programmer reads any one such type of data for processing, first we need to store into computer memory. The process of storing data elements into computer memory is called data representation. To process this data, data must be organized in a particular fashion. An organization means structuring of data.

Data structure - Definitions

Data structure deals with the study of how data is organized in the computer's ~~main~~ memory and how it maintains logical relationship between individual elements of data and also, how efficiently the data can be retrieved and manipulated

(or)

A data structure is a Systamatic way of storing data in a computer memory and the associated method for retrieving the data.

Data Structure - classification

Data structure is broadly classified into two categories.

1. primitive data structures
2. Non-primitive data structures [ADT]

* primitive data structures.

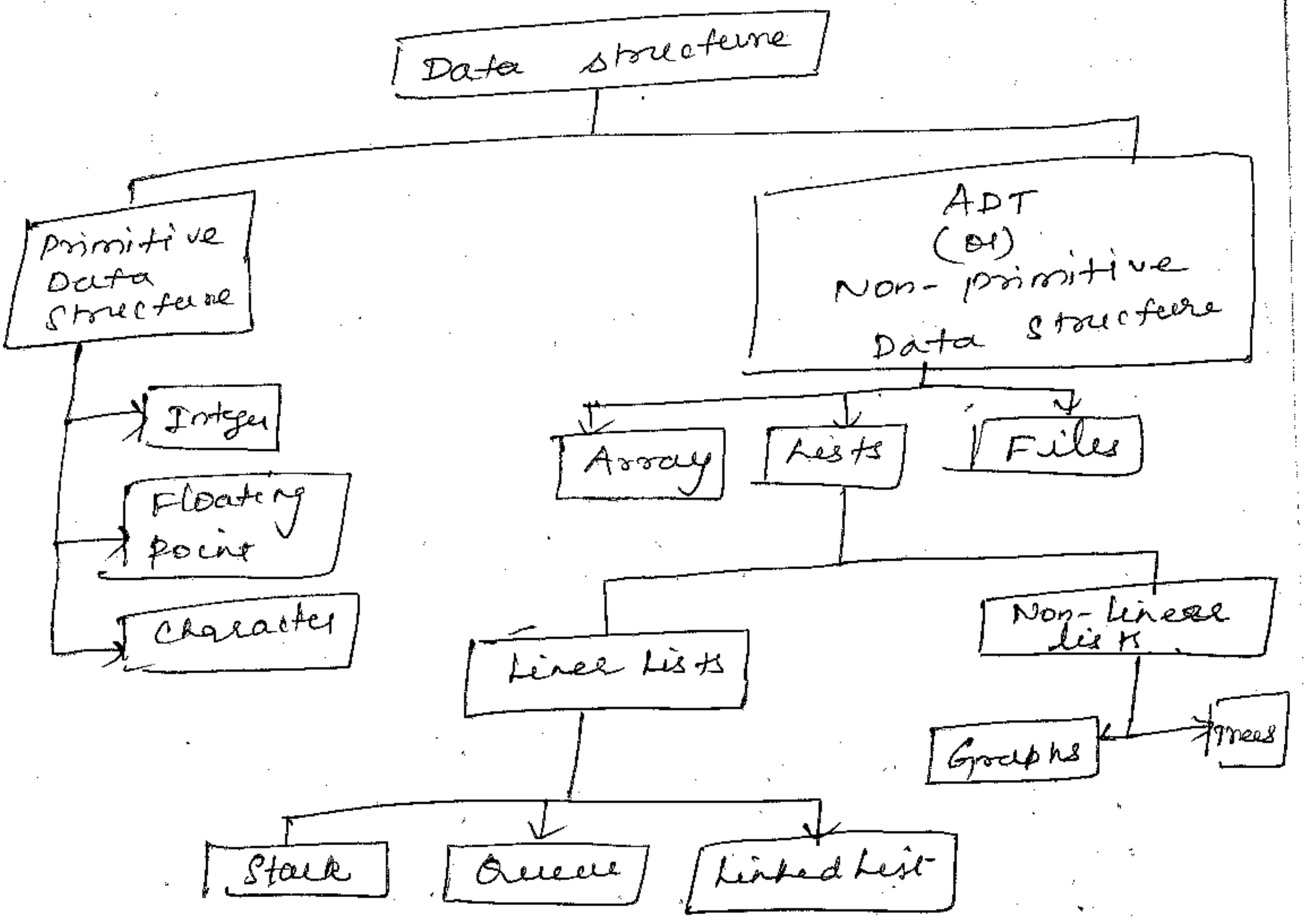
These data structures are basic structures and are manipulated / operated directly by machine instructions.

The integers, floating-point numbers, characte Constants, string constants etc are some of the primitive data structures. In c language, these primitive data structures are defined Using data types such as int, float, char and double. Representations of these primitive data types are already specified in the C compiler.

Abstract Data types [ADT] or Non-primitive Data types

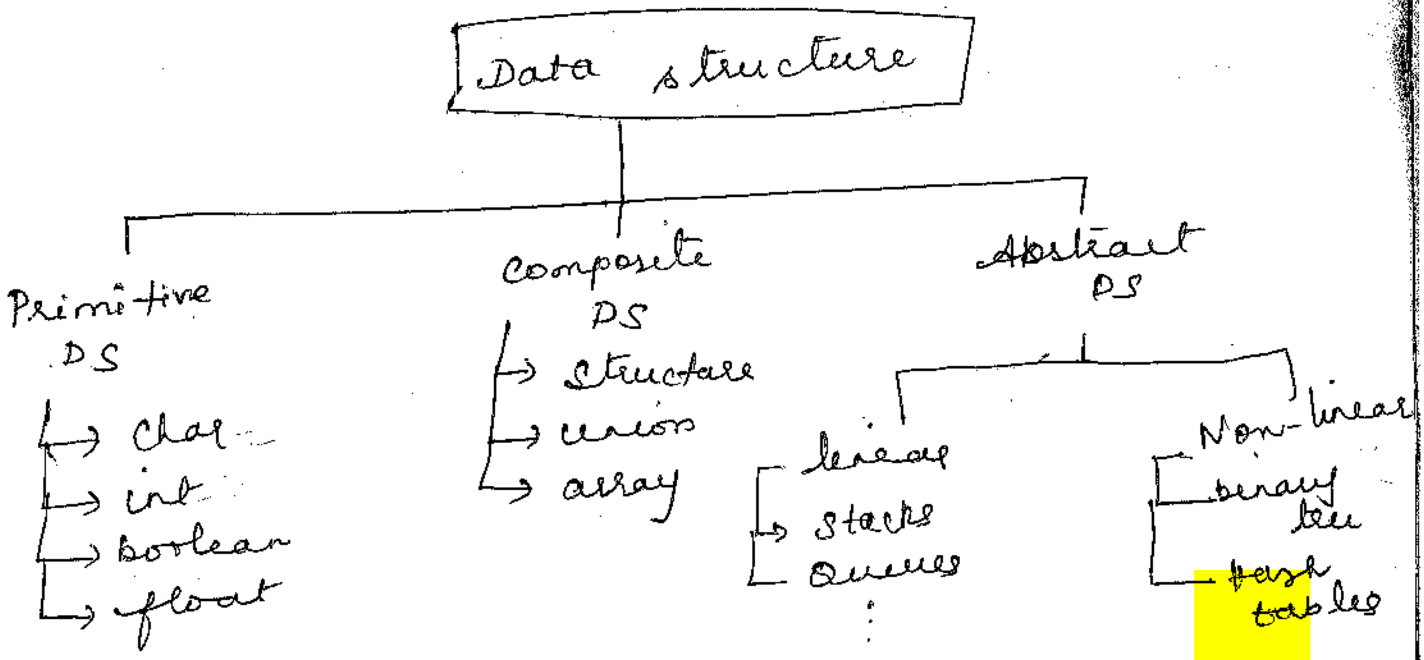
An abstract data type (ADT) is a Specification of a data type in a formal way without regard to any particular implementation or programming Language.

ADT is a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.



Data structure classification.

Our aim is study the ADT data structures and its operations. Each data structure categorized into different chapters. Each chapter contains data structure's description, operations, display of data structures contents with examples using C language.


Data structure
Primitive DS
Char
int
boolean
float
Composite DS
Structure
union
array
Abstract DS
linear
stacks
Queues
Non-linear
binary tree
hash tables

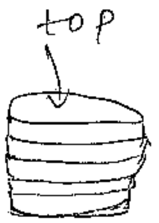
A stack is an ordered list in which insertions (also called pushes & adds) and deletions (also called pops and removes) are made at one end called the top.

Stacks

A stack is a linear list in which all additions and deletions are restricted to one end, called the top. If you insert a data series into a stack and then remove it, the order of the data is reversed.

Data input as {5, 10, 15, 20} is removed as {20, 15, 10, 5}. This reversing attribute is why stacks are known as the last in-first out [LIFO] data structure.

Eg



stacks of coins

stacks of books.

Definition

A stack is a last in-first out (LIFO) data structure in which all insertions and deletions are restricted to one end, called the top.

eg ~~As~~ when you visit a website and browse through the links, a stack of visited pages is stored by the browser. When you press the backbutton ~~last~~ most recently visited page will be displayed.

Basic Stack operations

The three basic stack operations are

- * push
- * POP
- * Stack top

*push operation

→ Push is used to insert data into the stack. It adds an item at the top of the stack. After the push, the new item becomes the top.

→ The only potential problem with this simple operation is that we must ensure that there is room for the new item. If there is not enough room, the stack is is an overflow state and the item cannot be added

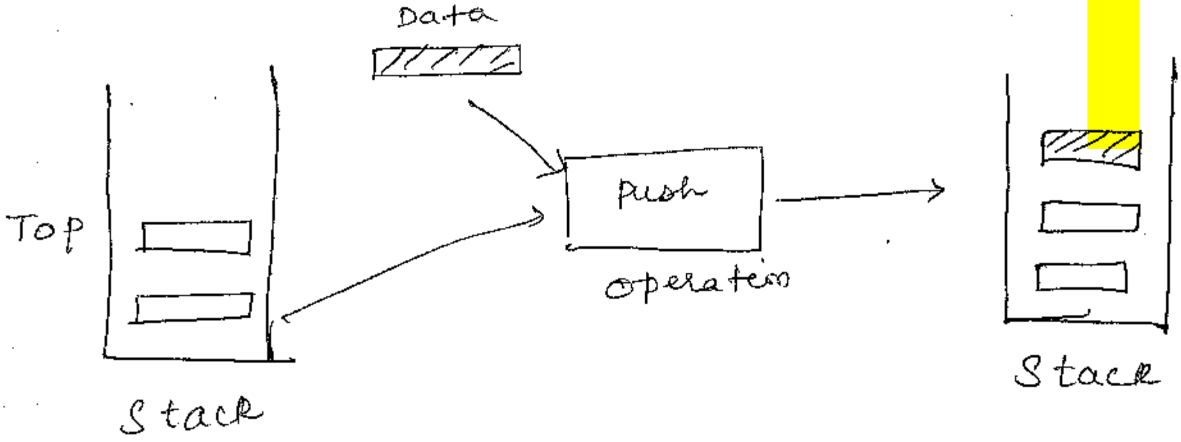


Fig: Push operation

Pop Operation

When we pop a stack, we remove the item at the top of the stack and return it to the user. Because we have removed the top item, the next older item in the stack becomes the top. when the last item in the stack is deleted, the stack must be set to its empty state. If pop is called when the stack is empty, it is in an underflow state.

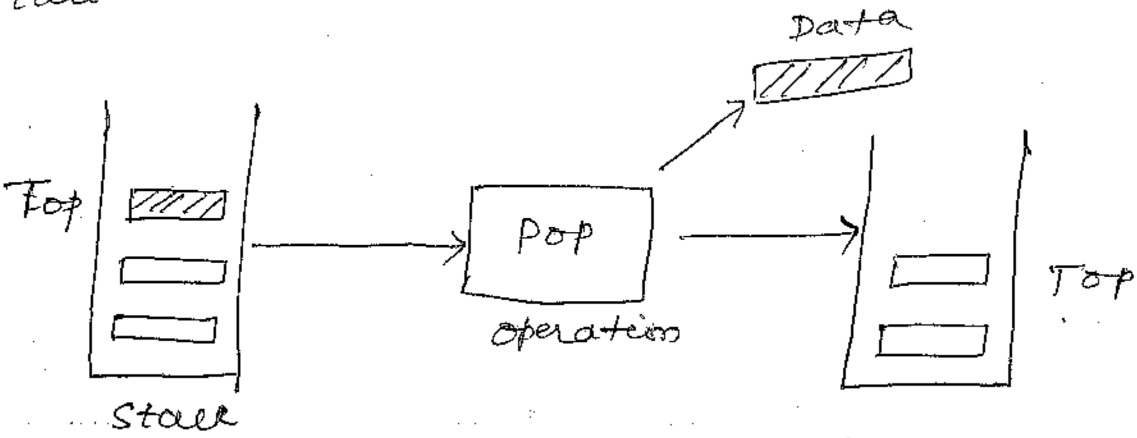
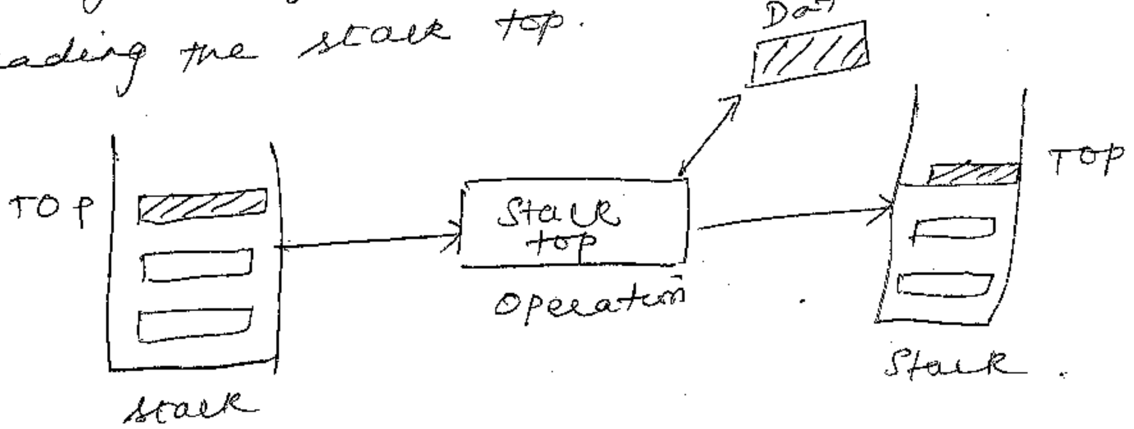


Fig: Pop Stack operation

Stack Top

The third stack operation is stack top. Stack top copies the item at the top of the stack, that is, it returns the data in the top element to the user but does not delete it. you might think of this operation as reading the stack top.



Implementation of Stack Using an array

```
#include <stdio.h>
#include < conio.h>
# define max 100

int Stack[max], ~~top=0~~; top= -1;

void main ()
{
      int ch;
      clrscr();
      do
       {
         printf ("\n Stack");
         Printf ("\n 1. push \n 2. pop \n 3. Display \n");
         Printf ("\n 4. Exit");
         Printf ("\n Enter your choice :");
         Scanf ("%d", &ch);

         Switch (ch)
          {
               case 1 : push();
                        break;

               case 2 : Pop();
                        break;
```

```
            case 3 : display()
                        break;
            case 4 : exit();
            default : Printf("\n Enter a Validchoice")
        }
    } while (1);
}

void push()
{  if (top==max-1
      Printf("\n Overflow");
   else
   {  int element;
      printf("Enter Element : \n");
      scanf("%d", & element);
      Printf("\n Element(%d) has been pushed
                    at %d", element, top);
      top++;
      Stack [top] = element;

      ~~top++;~~
   }
}
```

```
void pop()
{   if (top == -1)
    Printf ("underflow \n");
    else
    {
       top--;
       Printf ("Element has been popped out!");
    }
}

void display()
 {
    if(top==-1)
       Printf ("Stack is Empty!!");
       else
       {
         int i;
for(i=top; i>=0; i--) (or) for(i=0; i<=~~max~~top; i++)
             printf ("%d", Stack[i]);

       }
  }
```

* Stacks structures are usually implemented using arrays or linked list.

Stack Applications

Stack applications can be classified into four broad categories

1) reversing data

2) Parsing data → unmatched parentheses used in compiler.

infix to postfix transformation and evaluation → 3) Postponing data Usage and

4) backtracking steps. → computer Gaming, decision analysis and expert systems

The following applications are discussed ~~f~~ is detail

* Recursion

→ Sum of n numbers

→ factorial of given number

→ Fibonacci Series

→ GCD

→ Tower of Hanoi

* Conversion and evaluation of infix, prefix and postfix

Recursion

In general, there are two approaches to writing repetitive algorithms one uses iteration and Other uses recursion

definition

(Recursion is a repetitive process in which an algorithm calls itself.)

* Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first.

* Recursion is a technique that solves a problem by solving a smaller problem of the same type

* A procedure that is defined is terms of itself.

* In recursion approach the function calls itself until the condition is met. It is ~~Sl~~ slower than iteration, which means it uses more memory than iteration.

* Recursion makes code smaller and clean.

Recursion and Stacks

* Most Compilers implement recursion using stacks

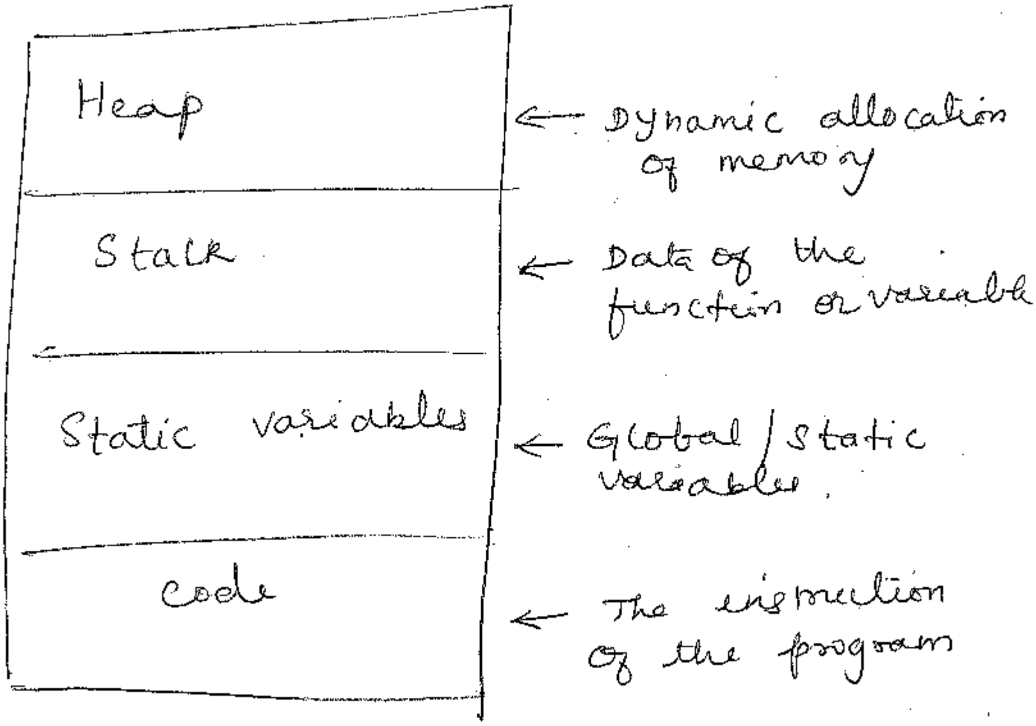
* When a method is called the compiler pushes the arguments to the method and the return address on the. stack and then transfers the control to the method

* When the method returns, it pops these Values Off the stack.

* The arguments disappear and the control returns to the return address.

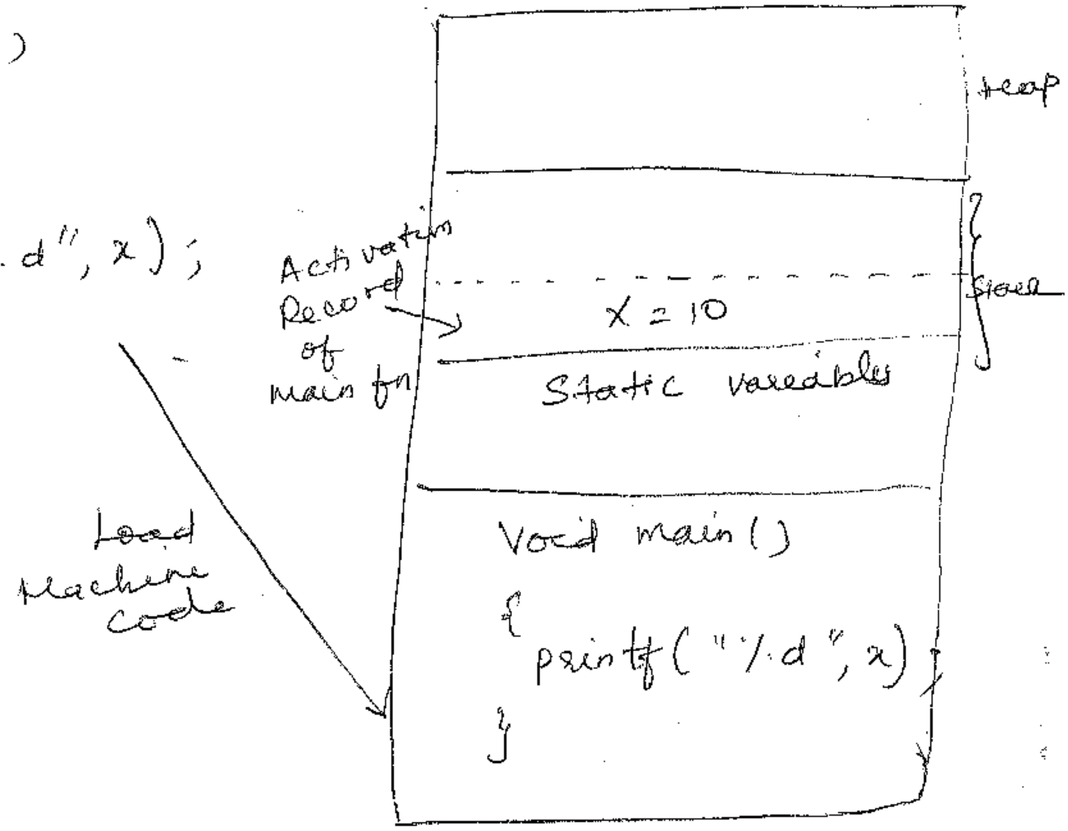
Main Memory

```
Void main ( )
{
   int a, b, c ;   ]
   float x;        ]  Data
   char c;         ]

   Printf ("Enter 3 numbers");                  ]
   Scanf ("%d %d %d", &a, &b, &c);              ]  Instruction

   - - - -
   - - - -

}
```


Heap
Stack
Static variables
code
← Dynamic allocation of memory
← Data of the function or variable
← Global / Static variables.
← The instruction of the program

Loading of program

```
void main()
 {
  int x=10;
  Printf("%d", x);

 }
```



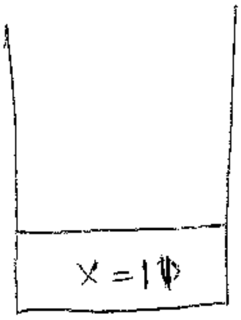
```
Void main()
{
 printf("%d", x)
}
```

Activation Record is created the moment the function is called. For every function activation record is created.

Function call [Stack Usage in function call]

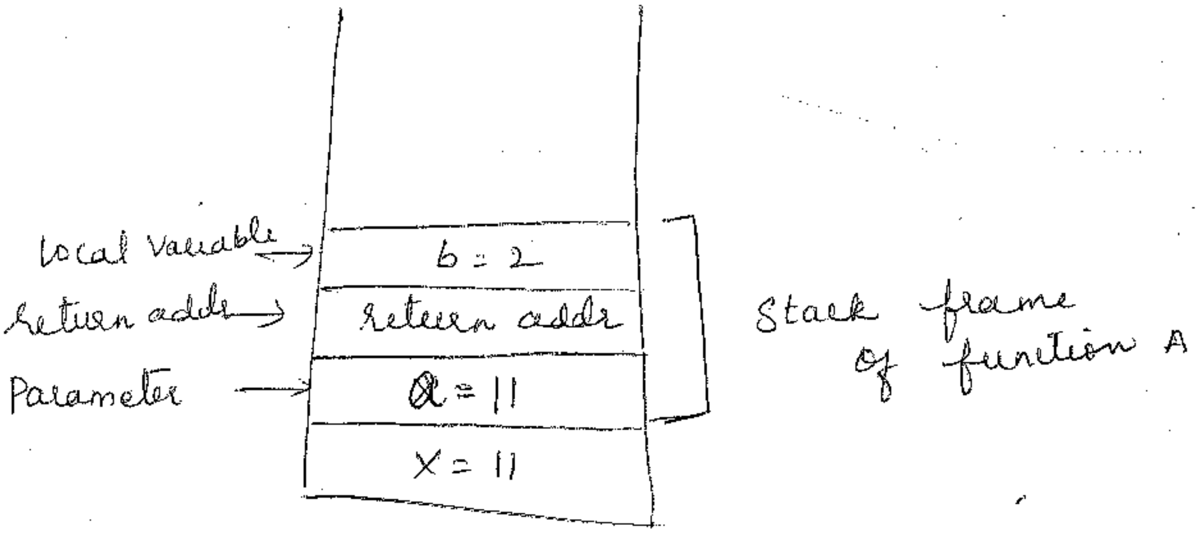
```
Void main ()
{
    int x = 10;
     x++;
     A(x);  ──── Functioncall ───┐
}                                │
                                 │
Void A(int a) <──────────────────┘
{  int b=2;
   printf ("%d",a);
}  Printf ("%d", b);
```

→ when main function is called, the local Variable X is pushed into a stack.



→ when main function calls function A, the following operations will be performed.

- * Push Parameters required by funition A
- * return address of main function
- * local Variables of function A



Once function A complites its ~~takk~~ task, it returns the control back to the called function. This is accomplised by pop the items till to obtain return address done by function A. Once the control is returned, the main function will clean up parameters of function (A).

Factorial of given number

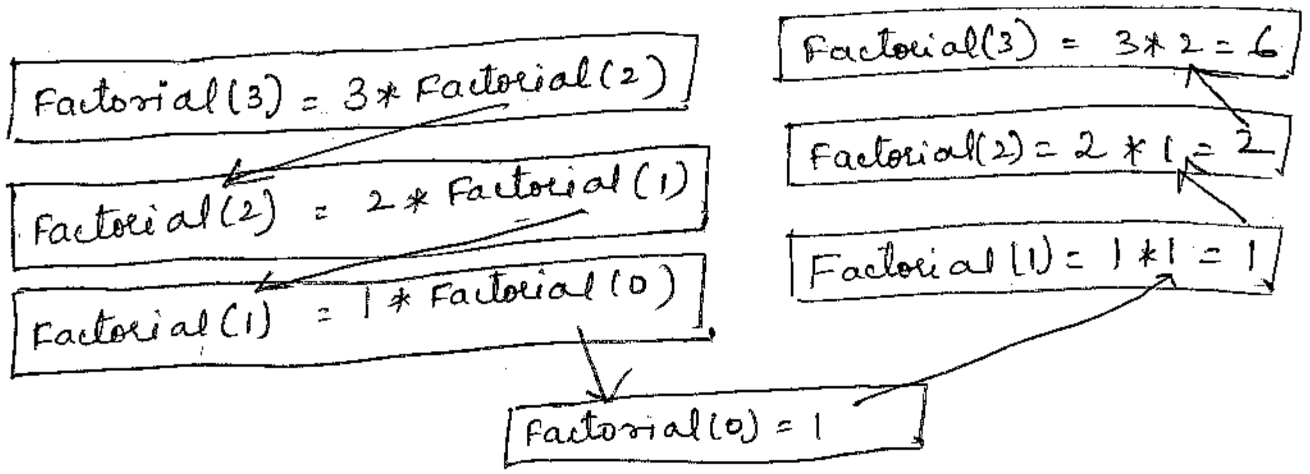
The factorial of a positive number is the product of the integral values from 1 to The number.

Factorial algorithm can be defined recursively as

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times (\text{Factorial}(n-1)) & \text{if } n>0 \end{cases}$$

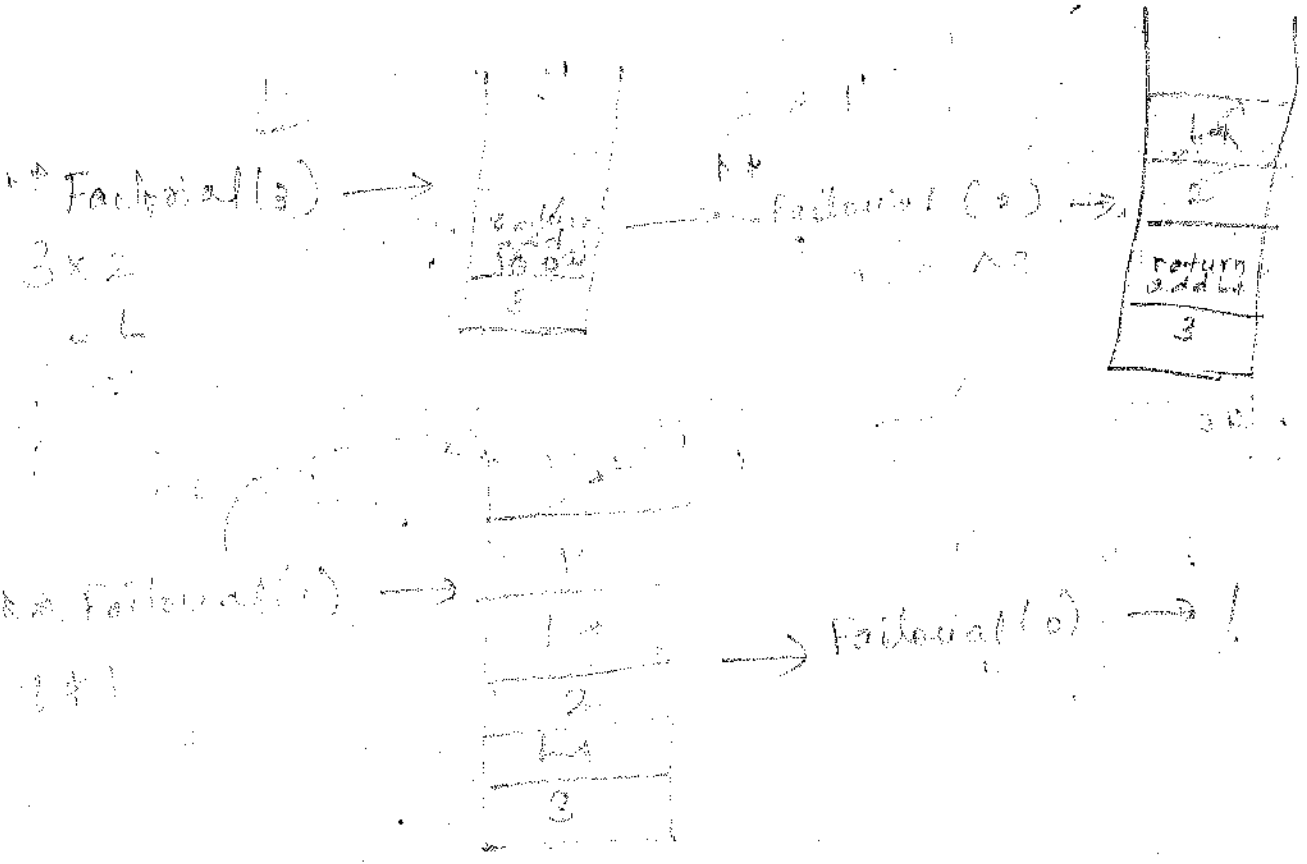
Algorithm {The factorial algorithm call itself, each time with different set of parameters}

```
Factorial (n)
{
  if (n==0)
      return 1;
  else
    return (n* Factorial(n-1))
}
```


Factorial(3) = 3* Factorial(2)
Factorial(2) = 2* Factorial(1)
Factorial(1) = 1* Factorial(0)
Factorial(0) = 1
Factorial(1) = 1*1 = 1
Factorial(2) = 2 * 1 = 2
Factorial(3) = 3*2 = 6

Recursive solution for a problem involves a two way journey. first we decompose the problem from the top to the bottom, then we solve it from the bottom to the top

Stack Usage in recursive function



Return value will be placed in EAX register

GCD → Greatest Common Divesor

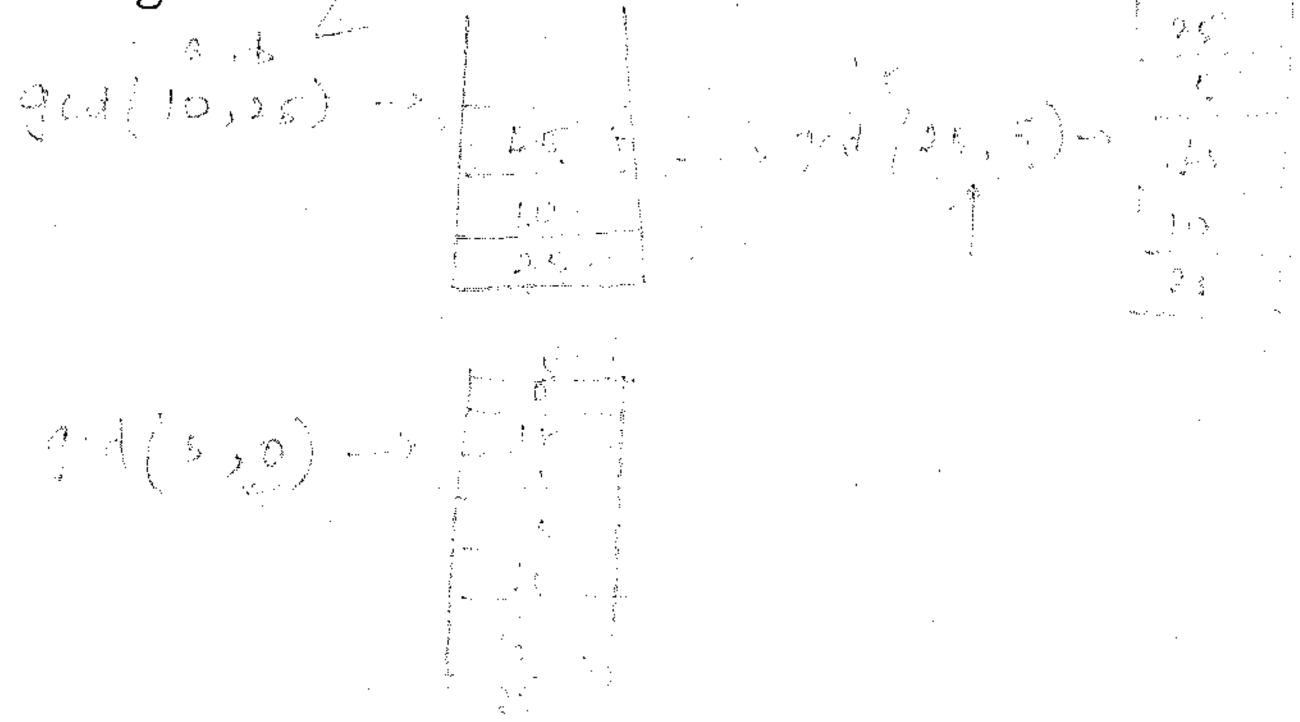
To determine the greatest common divisor between two non negative integers.

The recursive definition of GCD is

$$gcd(a,b) = \begin{cases} a & \text{if } b = 0 \\ b & \text{if } a = 0 \\ gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

Algorithm

```
int gcd (int a, int b)
{
   if (b==0)
      return a;
   if (a==0)
      return b;
   return gcd (b, a% b);
}
```



Fibonacci Numbers

In Fibonacci series, each number is, the sum of the previous 2 numbers.

0, 1, 1, 2, 3, 5, 8, 13, 21, 2*

To start a fibonacci series, we need to know the first 2 numbers.

Recursive definition of Fibonacci numbers is

$$\text{Fibonacci}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2) & \text{otherwise} \end{cases}$$

Algorithm

```
long fib (long num)
{
  if (num == 0 || num == 1)
    return num;
  return (fib(n-1) + fib(n-2));
}
```

Tower of Hanoi

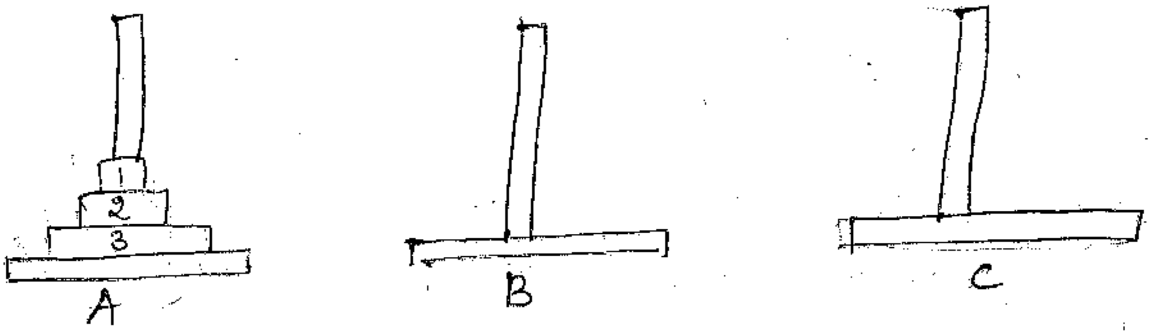
The Tower of Hanoi is a mathematical game or puzzle. It consists of three rods, and a number of disks of different sizes where can slide ~~only~~ onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules.

1. Only one disk can be moved at a time
2. 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.
3. No disk may be placed on Top of a smaller disk.

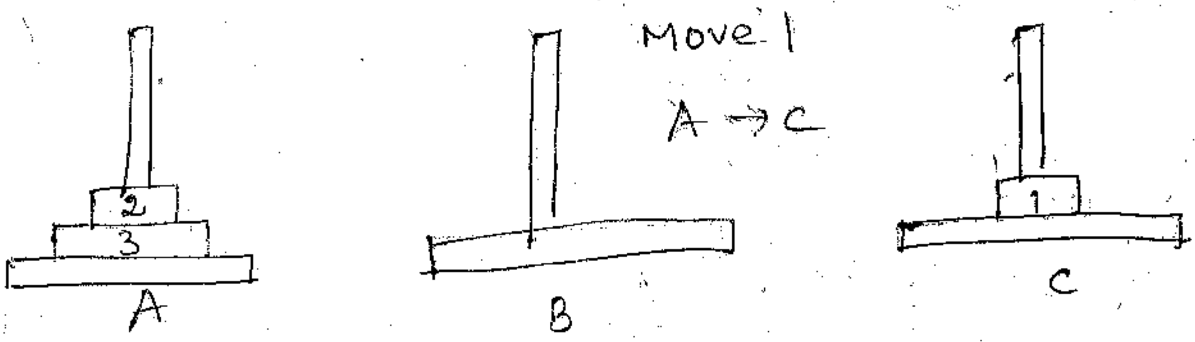
With three disks, the puzzle can be solved in seven moves. the minimum number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$, where n is the number of disks.

Intial State



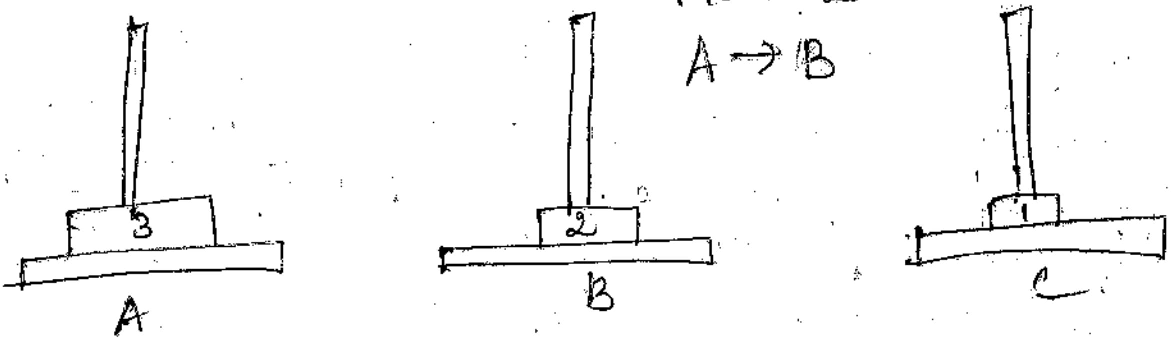
Move 1

A → C



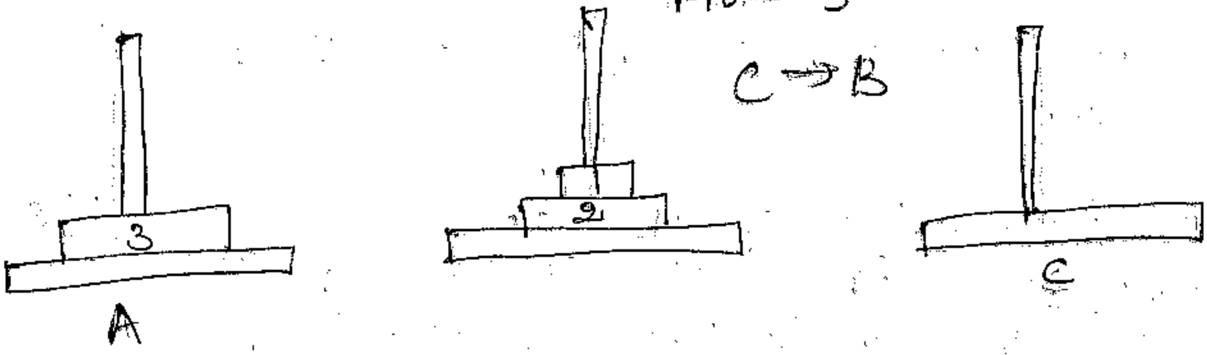
Move 2

A → B



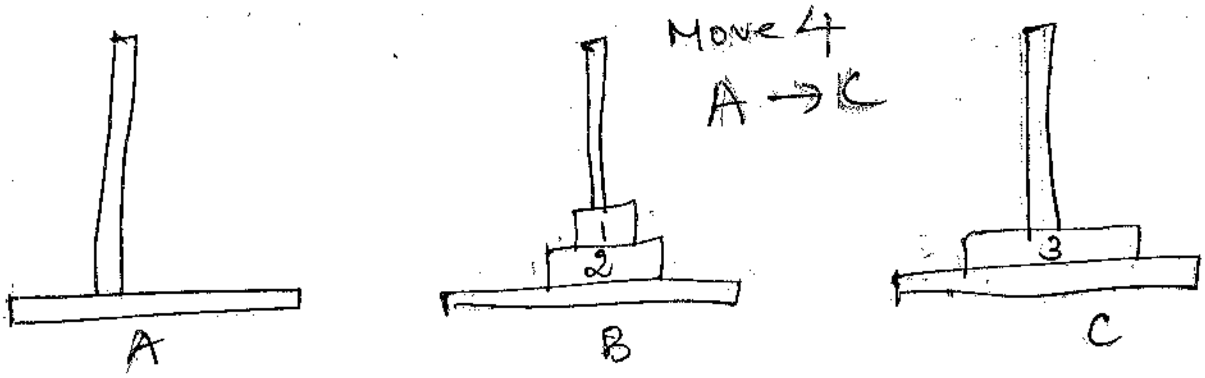
Move 3

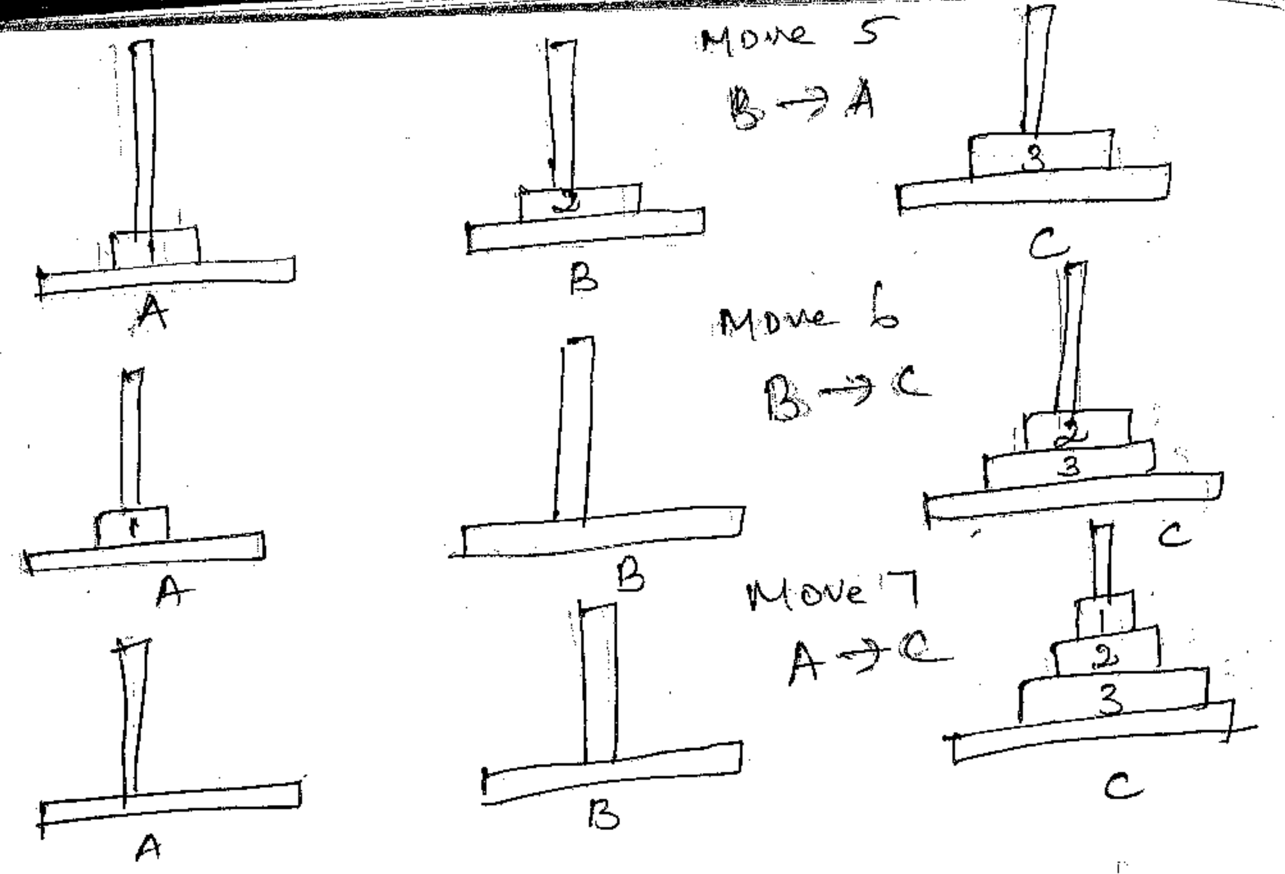
C → B



Move 4

A → C





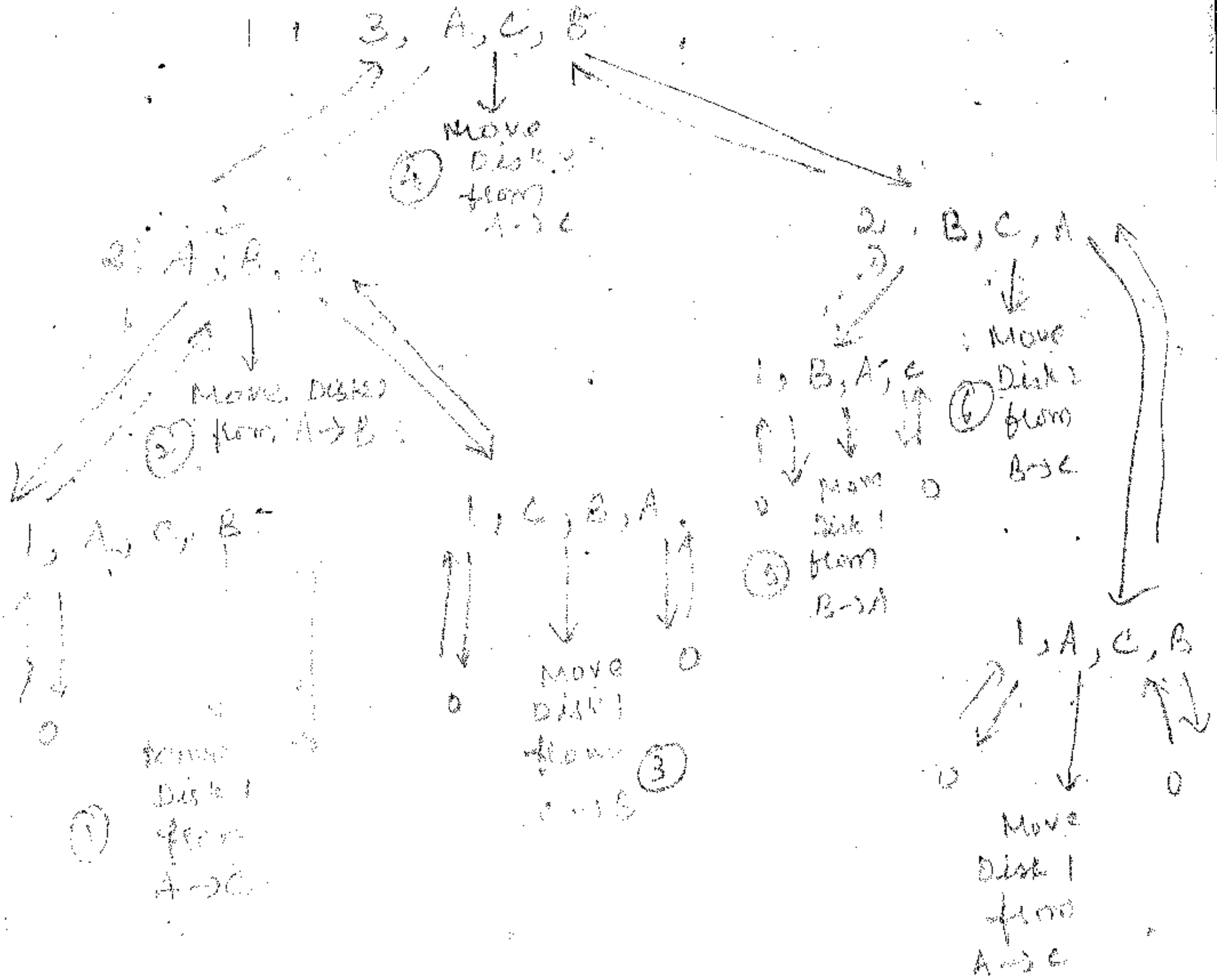
Move 5
B → A

Move 6
B → C

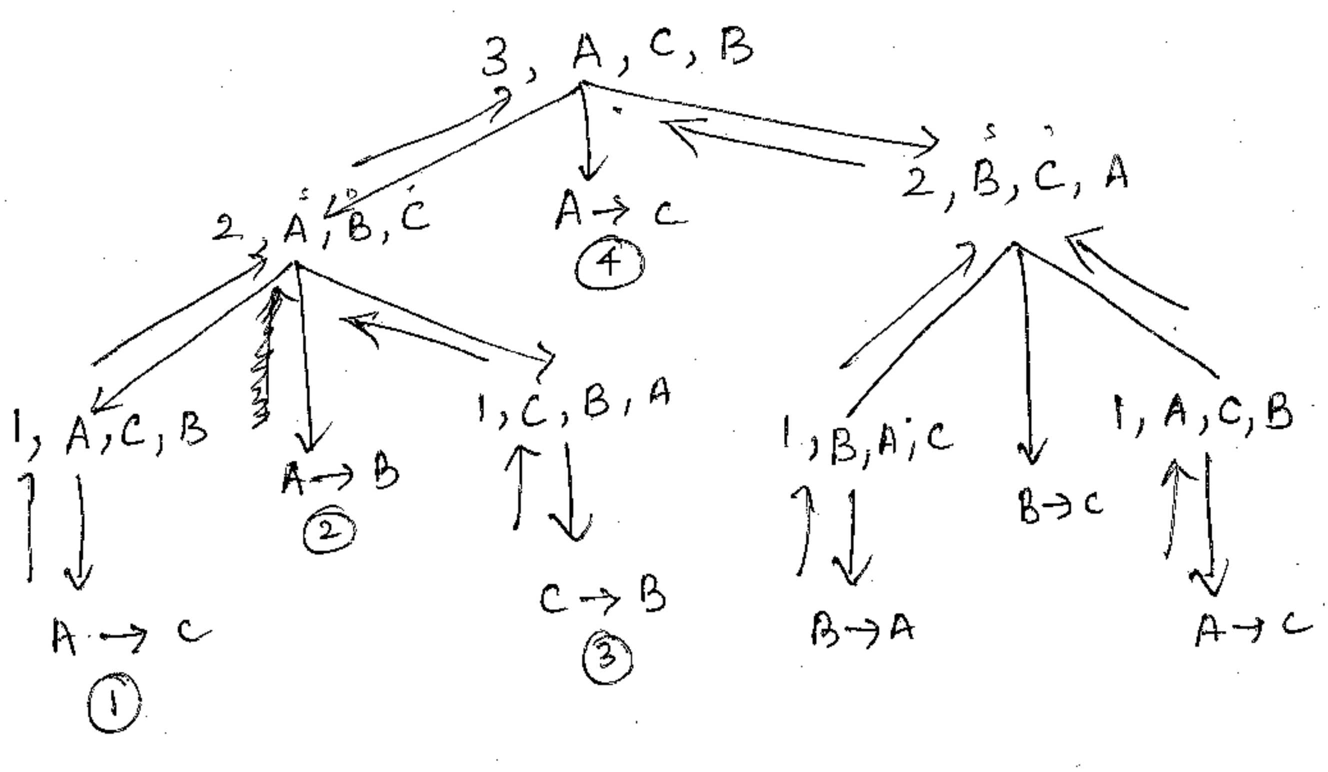
Move 7
A → C

Algorithm

```
void  TowerofHanoi (int n, char src, char dst,
                           char aux, int * moves)
  {
    if (n>0)
     {
       TowerofHanoi (n-1, Src, aux, dst, moves);
       Printf ("move disk %d from peg %c
                    to peg %c\n", n, src, dst);

       (*moves)++;
       Towerof.Hanoi (n-1, aux, dst, src, moves);

     }
  }
```

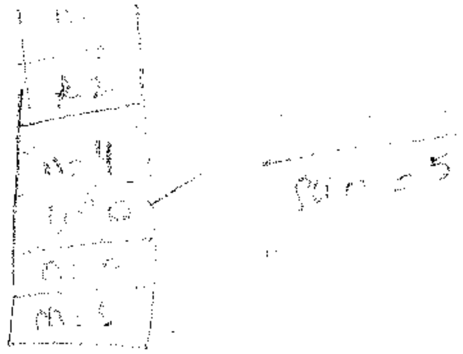

3, A, C, B
Move Disk 3 from A→C
4
2, A, B, C
Move Disk 2 from A→B
2
2, B, C, A
Move Disk 2 from B→C
6
1, A, C, B
0
Move Disk 1 from A→C
1
1, C, B, A
0
Move Disk 1 from C→B
3
0
1, B, A, C
0
from Disk 1 from B→A
5
0
1, A, C, B
0
Move Disk 1 from A→C
0

(Or)


3, A, C, B
2, A, B, C
A → C
4
2, B, C, A
1, A, C, B
A → B
2
1, C, B, A
1, B, A, C
B → C
1, A, C, B
A → C
1
C → B
3
B → A
A → C

Sum of n numbers.

```
main ( )          n: 5
{
w = getsum(n)
}
```



Algorithm

```
int getsum(n)
{
    static int sum=0;
      if (n>0)
       {
        sum = sum+n;
          getsum(n-1);
   }   }
   return sum;
}
```

5+4+3+2+1

Postponment

Often the logic of an application requires that the usage of data be deferred until some later point. A stack can be useful when the application requires that the use of data be posponed for a while

Infix, prefix and postfix expression

Consider the sum of A and B. we think of applying the operator "+" to be operands A and B and write the sum as A+B. This particular representation is called infix.

There are two alternate notations for expressing the sum of A and B using the symbols A, B and +. these are

+ AB prefix

AB + postfix

The prefixes "pre-", "post-," and "in-" refer to the relative position of the operator with respect to the two operands. In prefix notation the operator precedes the two operands, in postfix notation the operator follows the two operands and in ~~p~~ infix notation the operator is between the two operands.

Conversion of infix to postfix

One of the disadvantages of the infix notation is that we need to use parentheses to control the evaluation of the operators. We thus have an evaluation method that includes parentheses and two operator priority classes.

In the postfix and prefix notations, we do not need parentheses, each provides only one evaluation rule.

Although some high-level languages use infix notation, such expressions cannot be directly evaluated. Rather, they must be analyzed to determine the order in which the expressions are to be evaluated. A common evaluation technique is to convert the expressions to postfix notation before generating the code to evaluate them.

Rules for manual Converting infix to Postfix expressions

1. Fully parenthesize the expression using any explicit parentheses and the arithmetic precedence - multiply and divide before add and subtract.

2. change all infix notations in each parenthesis to postfix notation, starting from the innermost expressions. Conversion to postfix notation is done by moving the operator to the location of the expression's closing parenthesis.

3. Remove all parentheses.

eg 1 $\boxed{A + B * C}$

Step 1

$(A + (B * C))$

Step 2

$(A + (B C *))$

$(A (B C *) +)$

Step 3

$A B C * +$

eg2 $\boxed{(A+B)*C+D+E*F-G}$

Step 1

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

Step 2

$(((((AB+)C*)D+)\ (EF*)+)G-)$

Step 3

$AB+C*D+EF*+G-$

eg3

$\boxed{D-B+C}$

Step 1

$((D-B)+C)$

Step 2

$((DB-)+C)$
$((DB-)C+)$

Step 3

$DB-C+$

eg4 $\boxed{A * B + C * D}$

Step1 ((A*B) + (C*D))

Step2

((AB*) + (CD*))

((AB*) (CD*)+)

Step3

AB*CD*+

eg5 $\boxed{(A+B)*C-D*F+C}$

Step1

((((A+B)*C) - (D*F))+C)

Step2

((((AB+)C*) - (DF*))+C)

((((AB+)C*) (DF*)-)C+)

Step3 AB+C*DF*-C+

~~Algorithm~~ Steps to convert infix to Postfix

1. Print operands as they arrive.

2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack

3. If the incoming symbol is left parenthesis, push it on the stack.

4. If the incoming symbol is right parenthesis, pop the stack and print the operators until you see a left parenthesis.
[Discard the pair of parentheses]

5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack

6. If the incoming symbol has equal precedence with the top of the stack, Use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.

7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.

8. At the end Of the expression, pop and print all operators on the Stack (No parentheses Should remain)

Algorithm inToPostFix (formula)

```
createStack (Stack)
loop (for each character in formula)
    if(Character is Open parenthesis)
        PushStack (Stack, Character)
    elseif (character is close Parenthesis)
        popStack (Stack, Character)
        loop(character not open parenthesis)
            concatenate character to PostFixExpr
            PopStack (Stack, character)
        end loop
    else if (character is operator)
        StackTop (Stack, topToken)
        loop (not emptyStack (Stack)
        AND Priority(character) <= Priority (topToken))
        PopStack (Stack, tokenOut)
        concatenate tokenOut to PostFixExpr
        StackTop (Stack, topToken)
        endloop
        pushStack (Stack, token)
    else
```

```
        Concatenate token to postFixExpr
    endif
  end loop
  loop (not emptyStack (Stack) )
        popStack ( stack, character )
        concatenate token to postFixExpr
  end loop
     return postFix
end inToPostFix
```

eg

infix	stack	postfix
A + B * C - D/E		
+ B * C - D/E		A
B * C - D/E	+	A
* C - D/E	+	A B
C - D/E	* +	A B
- - D/E	* +	ABC
D/E	-	A B C * +
/ E	-	A B C * + D

/E	−	ABC*+D
E	/ −	ABC*+DE
	/ −	ABC*+DE/−

eg2

$\boxed{(A+B)*C - D*F+C}$

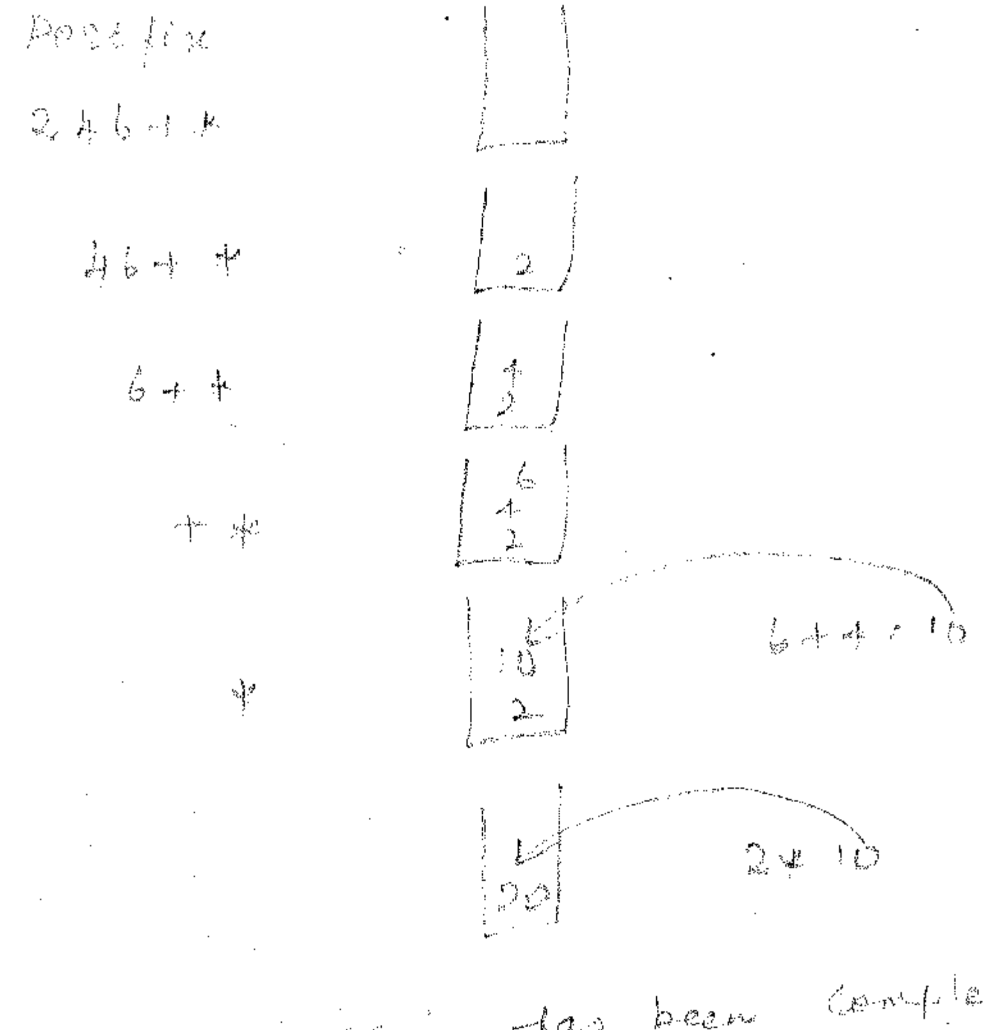
(A+B)*C − D*F+C	(	
A+B)*C − D*F+C	(	A
+B)*C − D*F+C	+ (	A
B)*C − D*F+C	+ (	A B
)*C − D*F+C		AB+
*C − D*F+C	*	AB+
C − D*F+C	*	AB+C

— D* F+C	—	AB+C*
D* F+C	—	AB+C*D
* F+C	* —	AB+C*D
F+C	* —	AB+C*DF
+C	+	AB+C*DF*—
C	+	AB+C*DF*—C+

Evaluation Postfix expression

In the postfix expression, the operands come before the operators. This means that we will have to postpone the Use of the operands this time, not the operators. We therefore put them into the stack. when we find ~~the~~ an operator, we pop the two operands at the top of the stack and perform the operation. We then push the value back in to the stack to be used later ~~[illegible]~~.

eg



When the expression has been completely evaluated, its value will be in the stack.

Algorithm

```
Algorithm postFixEvaluate (expr)
    createStack (stack)
    loop (for each character)
        if (character is operand)
            pushStack (stace, character)
        else
            PopStack (stack, oper2)
            popStack (stack, oper1)
            operator = character
            set value to calculate (oper1,
                                    operator, oper2)
            pushStack (stack, value)
        endif
    end loop
    popStack (stack, result)
    return (result)
end postFixEvaluate .
```

Iteration Vs Recursion techniques.

Recursion	iteration
1) Recursive function - is a function that is partially defined by itself	Iterative instruction - are Loop based repetitions of a process.
2) Recursion uses Selection Structure	Iteration uses repetition structure.
3) Infinite recursion occurs if the recursion step does not reduce the problem is a manner that converges on some Condition (base condition)	An infinite loop occurs with iteration if the Loop - condition test never becomes false.
4) Recursion terminates when a base case is recognized	Iteration terminates when the loop condition fails.
5) Recursion is usually slower than iteration due to overhead of maintaing stack	Iteration does not use stack so it is faster than recursion.

6) Recursion uses more memory than iteration.	Iteration consume less memory.
7) Infinite recursion can crash the system	infinite looping uses CPU cycles repeatedly.
8) Recursion makes code smaller	Iteration makes code longer.