DATA STRUCTURES

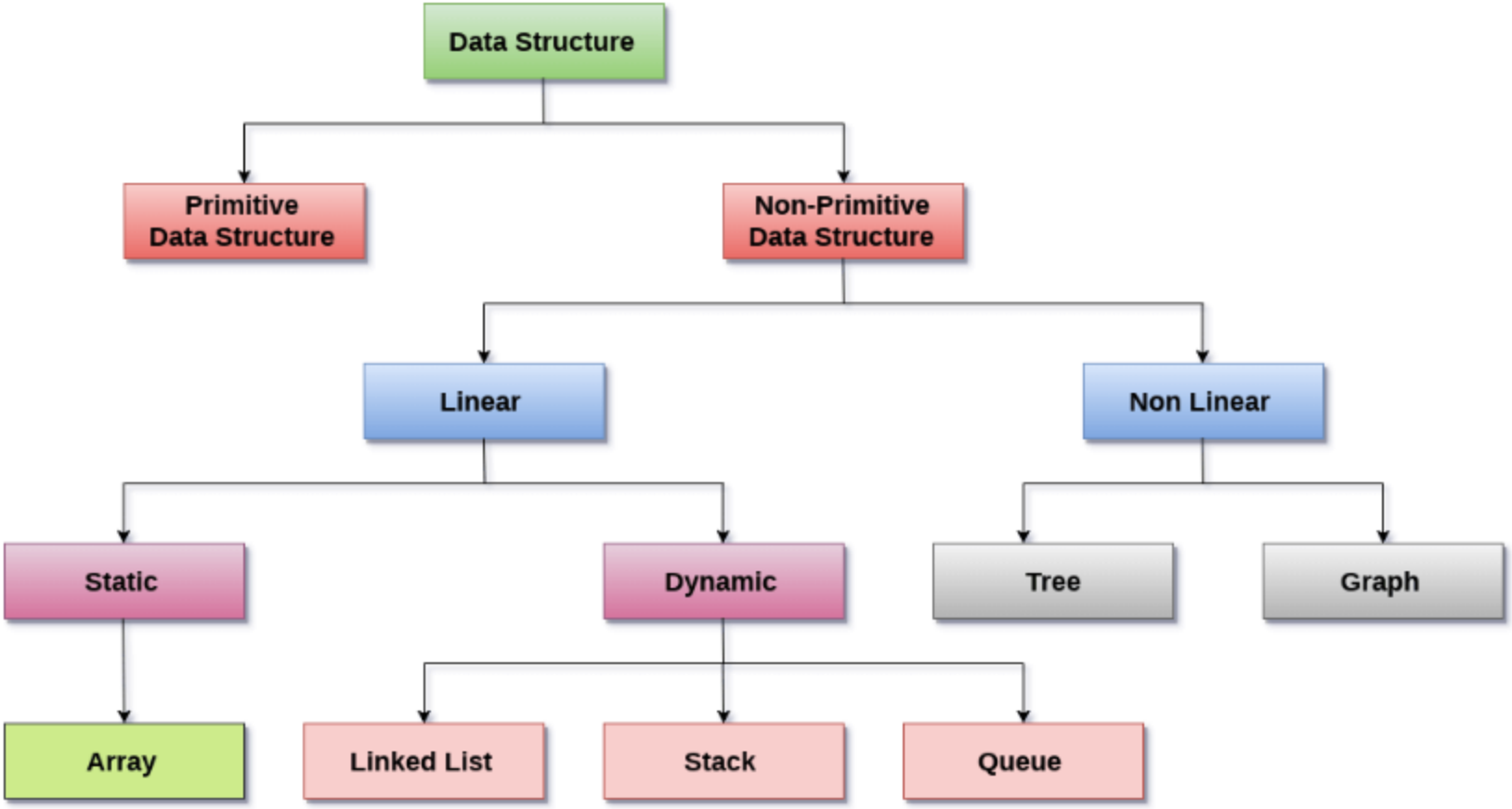
UNIT-1

Dr. SELVA KUMAR S
ASSOCIATE PROFESSOR
B.M.S. COLLEGE OF ENGINEERING

INTRODUCTION

- In computer science, a **data structure** is a data organization, management, and storage format that enables efficient access and modification.
- Data structures serve as the basis for abstract data types (ADT). The ADT defines the logical form of the data type. The data structure implements the physical form of the data type.
- Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.

Data structure classificaton



Advantage of Data structures

- Efficiency
- Reusability
- Abstraction

Operations on Data Structures

- Traversing
- Insertion
- Deletion
- Searching
- Sorting
- Merging

Basic concepts: Structures

- WAP for the below given scenario:
- A university wants to automate their admission process. Students are admitted based on the marks scored in a qualifying exam.
- A student is identified by student id, age and marks in qualifying exam. Data are valid, if:
- Age is greater than 20
- Marks is between 0 and 100 (both inclusive)
- A student qualifies for admission, if
- Age and marks are valid and
- Marks is 65 or more
- Write a program to represent the students seeking admission in the university.

```
#define SIZE 100
   struct student
   {
   int id;
   int age;
   int marks;
   };

void accept_input( struct student s1);
{
  printf("enter the id");
  scanf("%d",&s1.id);
  printf("enter the age");
  scanf("%d",&s1.age);

  printf("enter the marks");
  scanf("%d",&s1.marks);

}

void display_output( struct student s1)
{
  printf("sudent details are:\n");
  printf("ID - %d\t" s1.id);
  printf("ID - %d\t" s1.age);
  printf("ID - %d\n" s1.marks);
}

int main()
{
  struct student s[SIZE];
  int no_of_students,i;

  printf("Please enter no. of students")
  scanf("%d",&no_of_students);
  for(i=0;i<no_of_students;i++)
  {
    accept_input(s[i]);
  }

for(i=0;i<no_of_students;i++)
  {
    display_output(s[i]);
  }

   return 0;
}
```

Basic concepts: Pointers

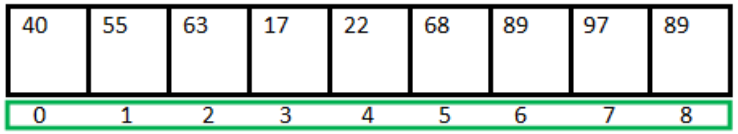
- WAP to swap two numbers using functions

```c
#include <stdio.h>

// function to swap the two numbers
void swap(int *x,int *y)
{
    int t;
     t   = *x;
    *x   = *y;
    *y   =  t;
}
```

```c
int main()
{
    int num1,num2;

    printf("Enter value of num1: ");
    scanf("%d",&num1);
    printf("Enter value of num2: ");
    scanf("%d",&num2);

    //displaying numbers before swapping
    printf("Before Swapping: num1 is: %d, num2 is: %d\n",num1,num2);

    //calling the user defined function swap()
    swap(&num1,&num2);

    //displaying numbers after swapping
    printf("After  Swapping: num1 is: %d, num2 is: %d\n",num1,num2);

    return 0;
}
```

Basic concepts: Dynamic memory allocation

Since C is a structured language, it has some fixed rules for programming. One of it includes changing the size of an array. An array is collection of items stored at continuous memory locations.



Array Length = 9
First Index = 0
Last Index = 8

As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example,

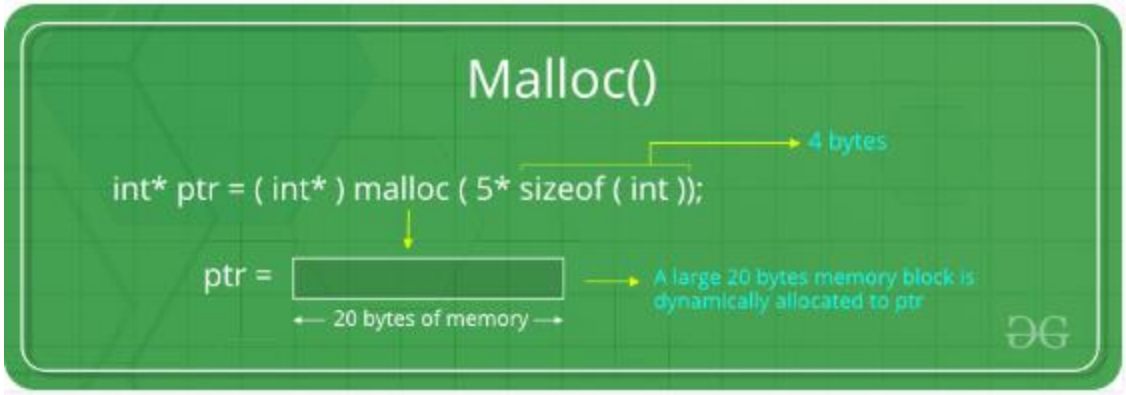
- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred to as **Dynamic Memory Allocation in C**.

Dynamic memory allocation

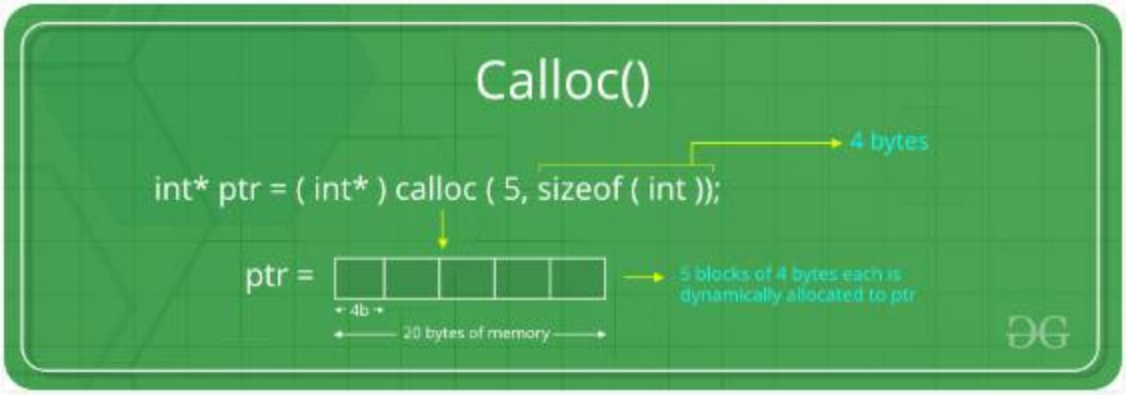
- **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
- C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:
- malloc()
- calloc()
- free()
- realloc()

Malloc()




```
// Dynamically allocate memory using malloc()
ptr = (int*)malloc(n * sizeof(int));
```

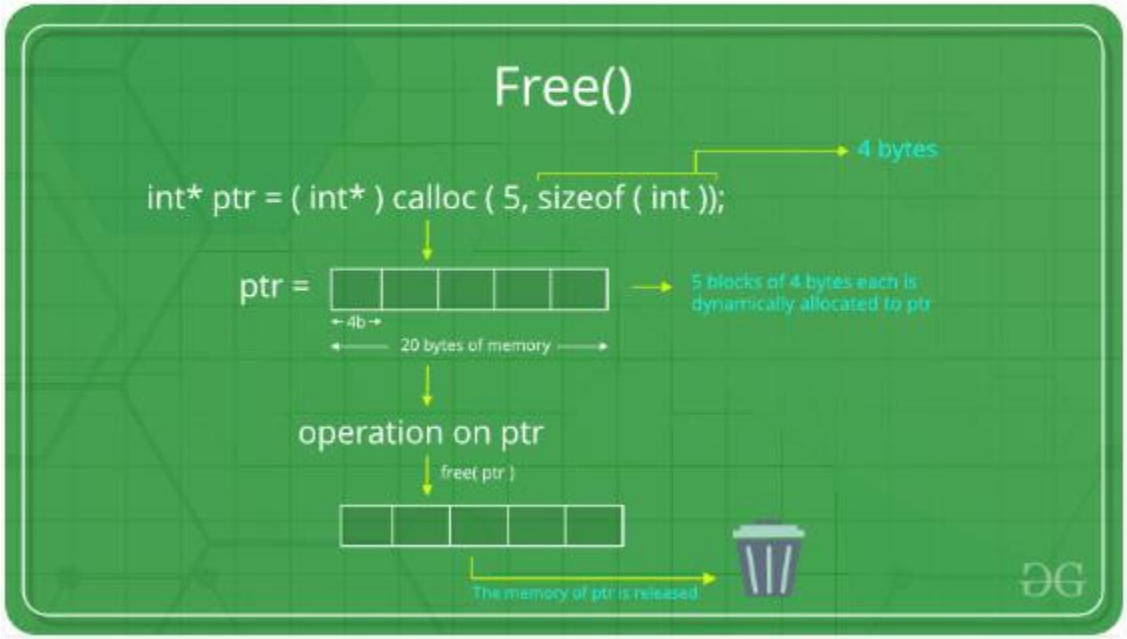
Calloc()



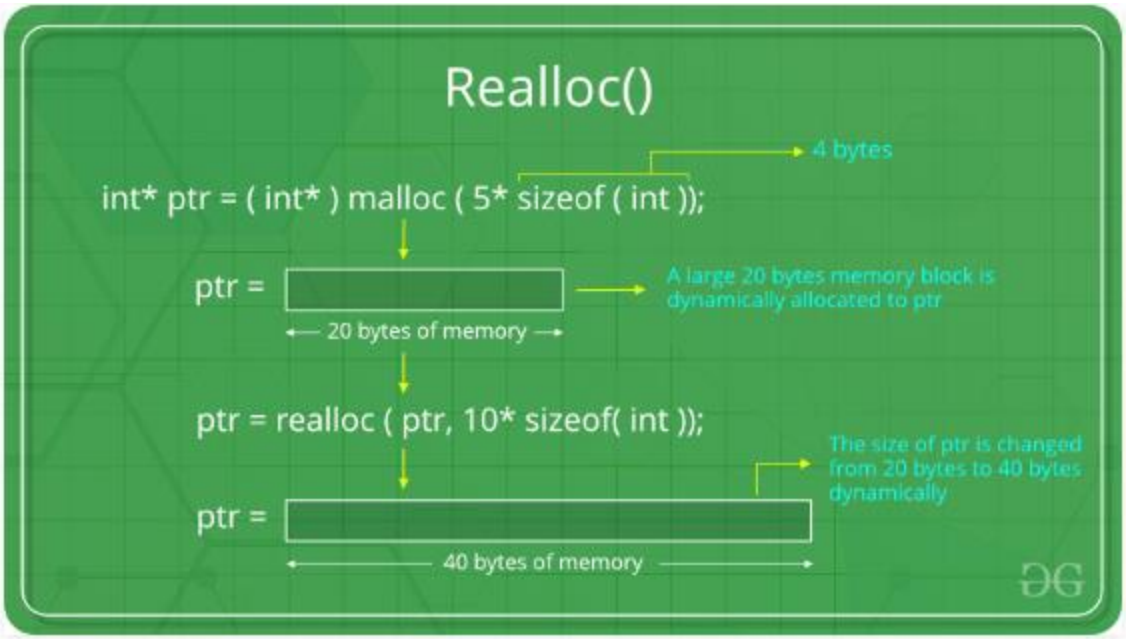

```
// Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));
```

Free()

```
free(ptr);
```



Realloc()




```
// Dynamically re-allocate memory using realloc()
ptr = realloc(ptr, n * sizeof(int));
```

Example

```
typedef struct {
    char * name;
    int age;
} person;
```

```
person * myperson = (person *) malloc(sizeof(person));
```

```
myperson->name = "John";
myperson->age = 27;
```

```
free(myperson);
```

Stack: Representation of stacks in C

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack.

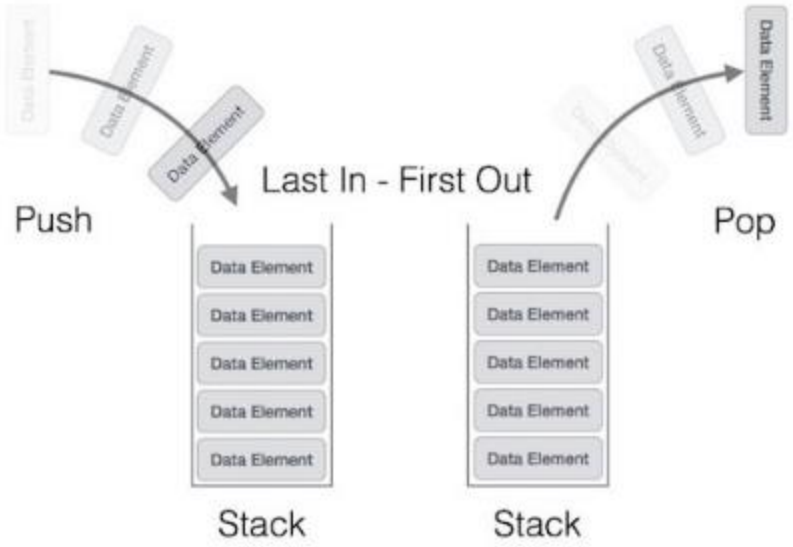
A real-world stack allows operations at one end only. Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first.




Stack Operations

- Basic operations:
- **push()** – Pushing (storing) an element on the stack.
- **pop()** – Removing (accessing) an element from the stack.



Stack Operations

- To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks:
- **top()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

Applications

- Function calls/Recursion
- Undo operations/Editors
- Balanced parentheses/ compiler
- Backtracking

Implementation of Stack

- Implementation of a stack using
  - Arrays
  - Linked List

Array Implementation

- Pseudo code:

```
int A[10]
top <-1 -1 //Empty stack
```

```
push(x)
{
 top<-top+1
 A[top]<-x
}
```

```
pop()
{
 top<- top - 1
}
```

```
top()
{
 return A[top]
}
```

```
IsEmpty()
{
 if(top == -1)
       return True;
else
      return False;
```

```
IsFull()
{
 if(top == SIZE)
    return True;
 else
    return False;
```

Array Implementation

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

#define SIZE 10

void push(int);
void pop();
void display();

int stack[SIZE], top = -1;

void main()
{
   int value, choice;
   clrscr();
   while(1){
      printf("\n\n***** MENU *****\n");
      printf("1. Push\n2. Pop\n3. Display\n4. Exit");
      printf("\nEnter your choice: ");
      scanf("%d",&choice);
      switch(choice){
         case 1: printf("Enter the value to be insert: ");
                 scanf("%d",&value);
                 push(value);
                 break;
         case 2: pop();
                 break;
         case 3: display();
                 break;
         case 4: exit(0);
         default: printf("\nWrong selection!!! Try again!!!");
      }
   }
}
```

```
void push(int value){
   if(top == SIZE-1)
      printf("\nStack is Full!!! Insertion is not possible!!!");
   else{
      top++;
      stack[top] = value;
      printf("\nInsertion success!!!");
   }
}

 void pop(){
    if(top == -1)
       printf("\nStack is Empty!!! Deletion is not possible!!!");
    else{
       printf("\nDeleted : %d", stack[top]);
       top--;
    }
 }

   void display(){
      if(top == -1)
         printf("\nStack is Empty!!!");
      else{
         int i;
         printf("\nStack elements are:\n");
         for(i=top; i>=0; i--)
            printf("%d\n",stack[i]);
      }
   }
```

Linked List Implementation

- Will be discussed in the Linked List chapter/Unit.

Example/application

- Reverse a string

```
/*stack variables*/
int top=-1;
int item;
```

```
/*string declaration*/
char stack_string[MAX];
```

```
int main()
{
    char str[MAX];

    int i;

    printf("Input a string: ");
    scanf("%[^\n]s",str); /*read string with spaces*/
    /*gets(str);-can be used to read string with spaces*/

    for(i=0;i<strlen(str);i++)
        pushChar(str[i]);

    for(i=0;i<strlen(str);i++)
        str[i]=popChar();

    printf("Reversed String is: %s\n",str);

    return 0;
}
```

```
/*function definition of pushChar*/
void pushChar(char item)
{
    /*check for full*/
    if(isFull())
    {
        printf("\nStack is FULL !!!\n");
        return;
    }

    /*increase top and push item in stack*/
    top=top+1;
    stack_string[top]=item;
}
```

```
/*function definition of popChar*/
 char popChar()
{
    /*check for empty*/
    if(isEmpty())
    {
        printf("\nStack is EMPTY!!!\n");
        return 0;
    }

    /*pop item and decrease top*/
    item = stack_string[top];
    top=top-1;
    return item;
}
```

```
/*function definition of isFull*/
int isFull()
{
    if(top==MAX-1)
        return 1;
    else
        return 0;
}
```

```
/*function definition of isEmpty*/
int isEmpty()
{
    if(top==-1)
        return 1;
    else
        return 0;
}
```

Check for balanced parentheses

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SIZE 100

// Global variables for stack and top
char stack[MAX_SIZE];
int top = -1;

// Function to push a character onto the stack
void push(char data) {
    if (top == MAX_SIZE - 1) {
        printf("Overflow stack!\n");
        return;
    }
    top++;
    stack[top] = data;
}

// Function to pop a character from the stack
char pop() {
    if (top == -1) {
        printf("Empty stack!\n");
        return ' ';
    }
    char data = stack[top];
    top--;
    return data;
}
```

```
int is_matching_pair(char char1, char char2) {
    if (char1 == '(' && char2 == ')') {
        return 1;
    } else if (char1 == '[' && char2 == ']') {
        return 1;
    } else if (char1 == '{' && char2 == '}') {
        return 1;
    } else {
        return 0;
    }
}

// Function to check if the expression is balanced
int isBalanced(char* text) {
    int i;
    for (i = 0; i < strlen(text); i++) {
        if (text[i] == '(' || text[i] == '[' || text[i] == '{') {
            push(text[i]);
        } else if (text[i] == ')' || text[i] == ']' || text[i] == '}') {
            if (top == -1) {
                return 0; // If no opening bracket is present
            } else if (!is_matching_pair(pop(), text[i])) {
                return 0; // If closing bracket doesn't match the last opening bracket
            }
        }
    }
    if (top == -1) {
        return 1; // If the stack is empty, the expression is balanced
    } else {
        return 0; // If the stack is not empty, the expression is not balanced
    }
}
```

```
// Main function
int main() {
    char text[MAX_SIZE];
    printf("Input an expression in parentheses: ");
    scanf("%s", text);

    // Check if the expression is balanced or not
    if (isBalanced(text)) {
        printf("The expression is balanced.\n");
    } else {
        printf("The expression is not balanced.\n");
    }
    return 0;
}
```

Using Structure

```
#define MAX 100

struct stack {
  char stck[MAX];
  int top;
}s;

void push(char item) {
  if (s.top == (MAX - 1))
    printf("Stack is Full\n");

  else {
    s.top = s.top + 1;
    s.stck[s.top] = item;
  }
}

void pop() {
  if (s.top == -1)
    printf("Stack is Empty\n");

  else
    s.top = s.top - 1;
}
```

```
int checkPair(char val1,char val2){
    return (( val1=='(' && val2==')' )||( val1=='[' && val2==']' )||( val1=='{' && val2=='}' ));
}

int checkBalanced(char expr[], int len){

    for (int i = 0; i < len; i++)
    {
        if (expr[i] == '(' || expr[i] == '[' || expr[i] == '{')
        {
          push(expr[i]);
        }
        else
        {
            // exp = {{}}}
            // if you look closely above {{}} will be matched with pair, Thus, stack "Empty"
            //but an extra closing parenthesis like '}' will never be matched
            //so there is no point looking forward
        if (s.top == -1)
            return 0;
        else if(checkPair(s.stck[s.top],expr[i]))
        {
            pop();
            continue;
        }
        // will only come here if stack is not empty
        // pair wasn't found and it's some closing parenthesis
        //Example : {{}}(]
        return 0;
        }
    }
    return 1;
}
```

```
int main() {
  char exp[MAX] = "({})[]{}";
  int i = 0;
  s.top = -1;

  int len = strlen(exp);
  checkBalanced(exp, len)?printf("Balanced"): printf("Not Balanced");

  return 0;
}
```

Using Structures and Pointers

```c
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

#define MAX 100

// Stack structure definition
typedef struct {
    char arr[MAX];
    int top;
} Stack;

// Initialize stack
void init(Stack *s) {
    s->top = -1;
}

// Check if stack is empty
bool isEmpty(Stack *s) {
    return s->top == -1;
}
```

```c
// Push element onto stack
void push(Stack *s, char c) {
    s->arr[++(s->top)] = c;
}

// Pop element from stack
char pop(Stack *s) {
    return s->arr[(s->top)--];
}

// Peek top element of stack
char peek(Stack *s) {
    return s->arr[s->top];
}
```

```
// Check if brackets are balanced
bool ispar(const char *s) {
    Stack stk;
    init(&stk);

    for (int i = 0; s[i] != '\0'; i++) {
        char ch = s[i];

        if (isEmpty(&stk)) {
            // Stack is empty, push the current bracket
            push(&stk, ch);
        } else if ((peek(&stk) == '(' && ch == ')') ||
                   (peek(&stk) == '{' && ch == '}') ||
                   (peek(&stk) == '[' && ch == ']')) {
            // Found a complete pair of brackets, pop it
            pop(&stk);
        } else {
            // Push current bracket onto stack
            push(&stk, ch);
        }
    }

    // If stack is empty, brackets are balanced
    return isEmpty(&stk);
}

int main() {
    const char *s = "{()}[]";

    // Function call to check for balanced brackets
    if (ispar(s)) {
        printf("true\n");
    } else {
        printf("false\n");
    }
    return 0;
}
```

Infix, Prefix and Postfix

- Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions.

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

Advantage of Postfix Expression over Infix Expression

- An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).
- Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

Evaluate the postfix expression

Pseudo code:

```
Evaluate_Postfix(exp)
{
 Create a Stack S
 for i<-0 to length(exp)-1
 {
  if exp[i] is operand
       push(exp[i])
 elseif (exp(i) is operator)
      {
         op2 <- pop()
         op1 <- pop()
         res <- compute(exp[i], op1, op2)
         pursh(res)
      }
 }
 return top of the stack
}
```

Example (Infix to Postfix)

- Infix: A*B+C*D-E
- AB*+C*D-E
- AB*+CD*-E
- AB*CD*+-E
- Postfix:AB*CD*+E-
-

Postfix Evaluation

```c
#include<stdio.h>
int stack[20];
int top = -1;

void push(int x)
{
    stack[++top] = x;
}

int pop()
{
    return stack[top--];
}
```

```c
int main()
{
    char exp[20];
    char *e;
    int n1,n2,n3,num;
    printf("Enter the expression :: ");
    scanf("%s",exp);
    e = exp;
```

```c
    while(*e != '\0')
    {
        if(isdigit(*e))
        {
            num = *e - 48;
            push(num);
        }
        else
        {
            n1 = pop();
            n2 = pop();
            switch(*e)
            {
            case '+':
            {
                n3 = n1 + n2;
                break;
            }
            case '-':
            {
                n3 = n2 - n1;
                break;
            }
            case '*':
            {
                n3 = n1 * n2;
                break;
            }
            case '/':
            {
                n3 = n2 / n1;
                break;
            }
            }
            push(n3);
        }
        e++;
    }
    printf("\nThe result of expression %s  =  %d\n\n",exp,pop());
```

Infix to Postfix Pseudo code

```
InfixtoPostfix(exp)
{
 create a Strack S
 for i<- 0 to lenght(exp)-1
 {
    if exp[i] is operand
       res <- res + exp[i]
    elseif exp[i] is operator
       while(!S.empty() && HasHigerPre(S.top(), exp[i])
             {
               res<-res+s.top()
               S.pop()
             }
          S.push(exp[i])
     elseif IsOpeningParentheses(exp[i])
          S.push(exp[i]
     elseif IsClosingParentheses(exp[i])
          {
            while(!S.empty() && !IsOpeningParentheses(S.top()))
                 {
                    res <- res + S.top()
                    S.pop()
                 }
                 S.pop()
          }
 }

  while(!S.empty())
   {
      res <- res+S.top()
           S.pop()
   }

return res
}
```

Algorithm

1. Push "(" onto Stack, and add ")" to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.
    [End of If]
6. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
    2. Remove the left Parenthesis.
    [End of If]
    [End of If]
7. END.

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where **^** is an exponential operator.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack , that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack , that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

C Code

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

#define SIZE 50

char stack[SIZE];
int top=-1;       /* Global declarations */

push(char elem)
{                          /* Function for PUSH operation */
    stack[++top]=elem;
}

char pop()
{                         /* Function for POP operation */
    return(stack[top--]);
}

int pr(char symbol)
{                  /* Function for precedence */

        if(symbol == '^')/* exponent operator, highest precedence*/
        {
                return(3);
        }
        else if(symbol == '*' || symbol == '/')
        {
                return(2);
        }
        else if(symbol == '+' || symbol == '-')          /* lowest precedence */
        {
                return(1);
        }
        else
        {
                return(0);
        }
}
```

```
  /* Main Program */
void main()
{
    char infix[50],postfix[50],ch,elem;
    int i=0,k=0;
    printf("Enter Infix Expression : ");
    scanf("%s",infix);
    push('#');
    while( (ch=infix[i++]) != '\0')
    {
        if( ch == '(') push(ch);
        else
            if(isalnum(ch)) postfix[k++]=ch;
            else
                if( ch == ')')
                {
                    while( stack[top] != '(')
                        postfix[k++]=pop();
                    elem=pop(); /* Remove ( */
                }
                else
                {       /* Operator */
                    while( pr(stack[top]) >= pr(ch) )
                        postfix[k++]=pop();
                    push(ch);
                }
    }
    while( stack[top] != '#')     /* Pop from stack till empty */
        postfix[k++]=pop();
    postfix[k]='\0';          /* Make postfix as valid string */
    printf("\nPostfix Expression =  %s\n",postfix);
}
```

Infix to Prefix

- Algorithm used
- Postfix
- Step 1: Add '')" to the end of the infix expression
- Step 2: Push(o nto the stack
- Step 3: Repeat until each character in the infix notation is scanned
  - IF a(is encountered, push it on the stack
  - IF an operand (whetheradigit oracharacter) is encountered, add it postfix expression.
  - IF a ")" is encountered, then
    a. Repeatedly pop from stack and add it to the postfix expression until a "(" is encountered.
    b. Discard the "(".That is, remove the(from stack and do not add it to the postfix expression
  - IF an operator O is encountered, then
    a. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence orahigher precedence than O
    b. Push the operator to the stack
- [END OF IF]
- Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
- Step 5: EXIT
- Prefix
- Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.
- Step 2: Obtain the postfix expression of the infix expression Step 1.
- Step 3: Reverse the postfix expression to get the prefix expression

Recursion

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.
- Using recursive algorithm, certain problems can be solved quite easily.
- Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Types of Recursion

- Direct recursion
- Indirect recursion
- Tail recursion
- Head recursion

Example

```
void printFun(int test)
{
   if (test < 1)
      return;
   else {
      printf(“%d”,test)
      printFun(test - 1); // statement 2
      printf(“%d”,test)
      return;
   }
}

int main()
{
   int test = 3;
   printFun(test);
}
```

Example: Find a Factorial of n

- factorial of n (n!) = 1 * 2 * 3 * 4 *... * n

```c
#include<stdio.h>
long int multiplyNumbers(int n);
int main() {
    int n;
    printf("Enter a positive integer: ");
    scanf("%d",&n);
    printf("Factorial of %d = %ld", n, multiplyNumbers(n));
    return 0;
}

long int multiplyNumbers(int n) {
    if (n>=1)
        return n*multiplyNumbers(n-1);
    else
        return 1;
}
```

Example: Fibonacci series

- 1 1 2 3 5 8 13 21 …. Fib(n) = Fib(n-1) + Fib(n-2)

```
//Fibonacci Series using Recursion
#include<stdio.h>
int fib(int n)
{
   if (n <= 1)                    // if(n==1 || n==2)
      return n;                   //      return 1
   return fib(n-1) + fib(n-2);
}

int main ()
{
  int n = 9;
  printf("%d", fib(n));
  getchar();
  return 0;
}
```

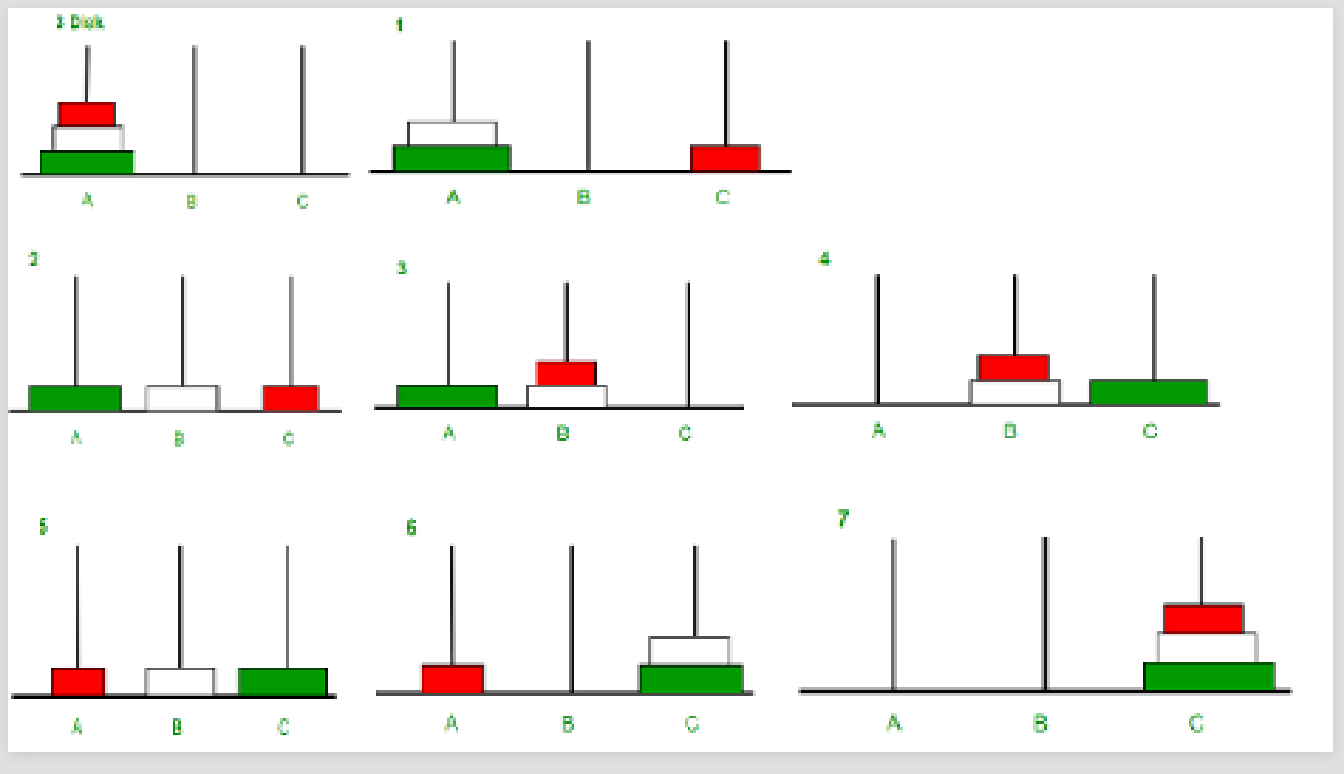
Example: Indirect Recursion

- Print odd number to n+1 and even number to n-1 (if n=10, print 2,1,4,3,6,5,8,7,10,9)

```
func odd(int n)
{
  if (n<=10)
   {
     printf("%d",n+1);
     n++;
     even(n);
   }
return;
}
```

```
func even(int n)
{
  if (n<=10)
   {
     printf("%d",n-1);
     n++;
     odd(n);
   }
return;
}
```

Example: Tower of Hanoi



If n=1

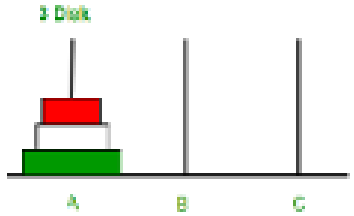
move disk from S to D

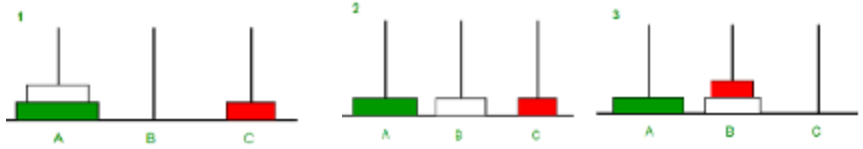
n>1

S1:move(n-1) disks from S to A using D

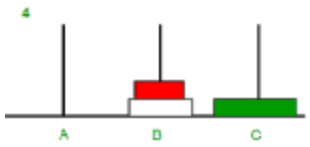
S2: move disk from S to D

S3: move(n-1) disks from A to D using S

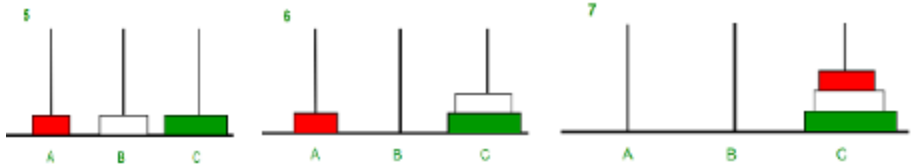




<-S1



<-S2



<-S3

```
Towerofhanoi(n,s,a,d)
{
    if(n==1)
  { print  “S ->D”
     return;
   }
  Towerofhanoi(n-1,s,d,a)
      printf “S->D”
  Towerofhanoi(n-1,a,s,d)
}
```

```c
/*
 * C program for Tower of Hanoi using Recursion
 */
#include <stdio.h>

void towers(int, char, char, char);

int main()
{
    int num;

    printf("Enter the number of disks : ");
    scanf("%d", &num);
    printf("The sequence of moves involved in the Tower of Hanoi are :\n");
    towers(num, 'A', 'C', 'B');
    return 0;
}
void towers(int num, char frompeg, char topeg, char auxpeg)
{
    if (num == 1)
    {
        printf("\n Move disk 1 from peg %c to peg %c", frompeg, topeg);
        return;
    }
    towers(num - 1, frompeg, auxpeg, topeg);
    printf("\n Move disk %d from peg %c to peg %c", num, frompeg, topeg);
    towers(num - 1, auxpeg, topeg, frompeg);
}
```

Trace the code

```
int sum(int n)
{
   if (n ==1)
     return 1;
   else
     return n+ sum(n-1);
}
```

Guess the output?

```
int f(int k, int n)
{
  if(n==k)
    return k;
  else
    if(n>k)
      return f(k, n-k);
    else
     return f(k-n, n)
}
```

How many times get() will be called before returned to main

```
void get(int n)
{
  if(n<1) return;
  get(n-1);
  get(n-3);
  printf("%d",n);
}
```