WHAT IS DATA STRUCTURE?

"The way information is organized in the memory of a computer is called a **data structure"**.

(OR)

A data structure is a way of organizing data that considers not only the items stored, but also their relationship to each other. Advance knowledge about the relationship between data items allows designing of efficient algorithms for the manipulation of data.

Definition of data structures

- Many algorithms require that we use a proper representation of data to achieve efficiency.
- This representation and the operations that are allowed for it are called data structures.
- Each data structure allows insertion, access, deletion etc.

Why do we need data structures?

- Data structures allow us to achieve an important goal: component reuse

- Once each data structure has been implemented once, it can be used over and over again in various applications.

Common data structures are

- Stacks • Queues • Lists
- Trees • Graphs • Tables

Classification of data Structure:

Based on how the data items or operated it will classified into

1. **Primitive Data Stucture :** is one the data items are operated closest to the machine level instruction.

   Eg : int, char and double.

2. **Non-Primitive Data Structure :** is one that data items are not operated closest to machine level instruction.

2.1. **Linear Data Structure :** In which the data items are stored in sequence order.

Eg: Arrays, Lists, Stacks and Queues.

2.2. **Non Linear Data Structure :** In which the order of data items is not presence.

Eg : Trees, Graphs.

Linear Data Structure

1. List
    a. Array
        i. One Dimensional
        ii. Multi-Dimensional
        iii. Dynamic Array
        iv. Matrix
            1. Sparse Matr
    b. Linked List
        i. Single Linked List
        ii. Double Linked Lis
        iii. Circular Linked Li
    c. Ordered List
        i. Stack
        ii. Queue
            1. Circular Qu
            2. Priority Que
        iii. Deque
2. Dictionary (Associative Array)
    a. Hash Table

Non-Linear Data Structures

1. Graph
    a. Adjacency List
    b. Adjacency Matrix
    c. Spanning Tree
2. Tree
    a. M-Way Tree
        i. B-Tree
            1. 2-3-4 Tree
            2. B+ Tree
    b. Binary Tree
        i. Binary Search Tree
        ii. Self-Balancing Binary Search Tree
            1. AVL Tree
            2. Red-Black Tree
            3. Splay Tree
        iii. Heap
            1. Min Heap
            2. Max Heap
            3. Binary Heap
        iv. Parse Tree

Operations performed on any linear s

1. Traversal – Processing each element in the list
2. Search – Finding the location of the element with a given value.
3. Insertion – Adding a new element to the list.
4. Deletion – Removing an element from the list.
5. Sorting – Arranging the elements in some type of order.
6. Merging – Combining two lists into a single list.

An example of several common data s **Characteristics of Data Structures**

Data Structure	Advantages	Disadvantages
Array	Quick inserts Fast access if index known	Slow search Slow deletes Fixed size
Ordered Array	Faster search than unsorted array	Slow inserts Slow deletes Fixed size
Stack	Last-in, first-out acces	Slow access to other items
Queue	First-in, first-out access	Slow access to other items

Linked List	Quick inserts Quick deletes	Slow search
Binary Tree	Quick search Quick inserts Quick deletes *(If the tree remains balanced)*	Deletion algorithm is complex
Red-Black Tree	Quick search Quick inserts Quick deletes *(Tree always remains balanced)*	Complex to implement
2-3-4 Tree	Quick search Quick inserts Quick deletes *(Tree always remains balanced)* *(Similar trees good for disk storage)*	Complex to implement
Hash Table	Very fast access if key is known Quick inserts	Slow deletes Access slow if key is not known Inefficient memory usage
Heap	Quick inserts Quick deletes Access to largest item	Slow access to other items
Graph	Best models real-world situations	Some algorithms are slow and very complex

Abstract Data Types

Abstract data type (**ADT**) is a specification of a set of data and the set of operations that can be performed on the data.

Examples

- Associative array
- Set
- Stack
- Queue
- Tree

Uses of ADT: -

1. It helps to efficiently develop well designed program
2. Facilitates the decomposition of the complex task of developing a software system into a number of simpler subtasks
3. Helps to reduce the number of things the programmer has to keep in mind at any time
4. Breaking down a complex task into a number of earlier subtasks also simplifies testing and debugging

Algorithm:

Definition: An *algorithm* is a finite set of instructions which, if followed, accomplish a particular task. In addition every algorithm must satisfy the following criteria:

1. input: there are zero or more quantities which are externally supplied;
2. output: at least one quantity is produced;
3. definiteness: each instruction must be clear and unambiguous;
4. finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Linear Data Structures

A data structure is said to be *linear* if its elements form a sequence or a linear list.

Examples:

Arrays
Linked Lists
Stacks, Queues

Arrays

Arrays

- The very common linear structure is array. Since arrays are usually easy to traverse, search and sort, they are frequently used to store relatively permanent collections of data.
- An array is a list of a finite number n of *homogeneous* data elements (i.e., data elements of the same type) such that:
    a) The elements of the array are referenced respectively by an *index* consisting of n consecutive numbers.
    b) The elements of the array are stored respectively in successive memory locations.

Opearations of Array

- Two basic operations in an array are *storing* and *retrieving (extraction)*

Storing: A value is stored in an element of the array with the statement of the form,

Data[i] = X ; Where I is the valid index in the array
And X is the element

Extraction : Refers to getting the value of an element stored in an array.

X = Data [i], Where I is the valid index of the array and X is the element.

Array Representation

- ✓ The number ***n*** of elements is called the *length* or *size* of the array. If not explicitly stated we will assume that the index starts from 0 and end with n-1.
- ✓ In general, the length (range) or the number of data elements of the array can be obtained from the index by the formula,

  Length = UB – LB + 1
- ✓ Where UB is the largest index, called the Upper Bound, and LB is the smallest index, called Lower Bound, of the array.
- ✓ If LB = 0 and UB = 4 then the length is,

  Length = 4 – 0 + 1 = 5
- ✓ The elements of an array A may be denoted by the subscript notation (or bracket notation),

  A[0], A[1], A[2], … , A[N]
- ✓ The number K in A[K] is called a *subscript* or an *index* and A[K] is called a *subscripted variable*.
- ✓ Subscripts allow any element of A to be referenced by its relative position in A.
- ✓ If each element in the array is referenced by a single subscript, it is called single dimensional array.
- ✓ In other words, the number of subscripts gives the dimension of that array.

Two-dimensional Arrays

- ✓ A two-dimensional *mXn* array A is a collection of m*n data elements such that each element is specified by a pair of integers (such as I, J), called subscripts, with the property that,

  0 ≤ I < m and 0 ≤ J < n
- ✓ The element of A with first subscript ***i*** and second subscript ***j*** will be denoted by,

  A[i,j] or A[i][j] *(c language)*
- ✓ Two-dimensional arrays are called *matrices* in mathematics and *tables* in business applications; hence two-dimensional arrays are sometimes are called *matrix arrays.*
- ✓ There is a standard way of drawing a two-dimensional *mXn* array A where the elements of A form a rectangular array with m rows and n columns and where the element A[i][j] appears in ***row i*** and ***column j***.
- ✓ A *row* is a horizontal list of elements, and a *column* is a vertical list of elements.

  Example:

  Columns

		0	1	2
	0	A[0][0]	A[0][1]	A[0][2]
Rows	1	A[1][0]	A[1][1]	A[1][2]
	2	A[2][0]	A[2][1]	A[2][2]

- ✓ The two-dimensional array will be represented in memory by a block of m*n sequential memory locations.
- ✓ Specifically, the programming languages will store the array either
    1. Column by column, i.e. ***column-major order,*** or
    2. Row by row, i.e. ***row-major order.***

Abstract Data Types (ADT)

- ❖ The ADT consists of a set of definitions that allow programmers to use the functions while hiding the implementation. This generalization of operations with unspecified implementations is known as abstraction.
- ❖ An ADT is a data declaration packaged together with the operations that are meaningful on the data type.
    1. Declaration of Data
    2. Declaration of Operations

An array is a collection of memory locations which allows storing homogeneous elements. It is an example for linear data structure.

An array lets you declare and work with a collection of values of the same type (homogeneous). For example, you might want to create a collection of five integers. One way to do it would be to declare five integers directly:

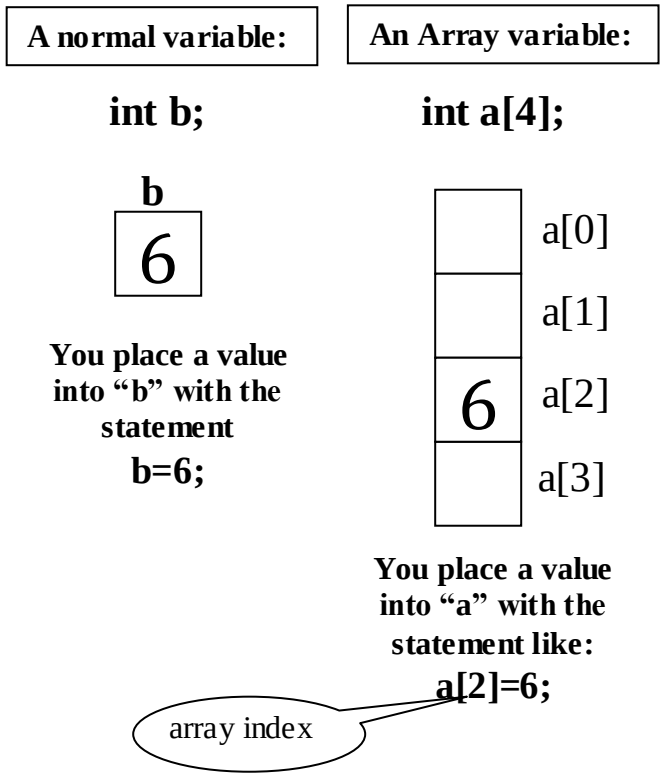
```
int a, b, c, d, e;
```

Suppose you need to fined average of 100 numbers. What will you do? You have to declare 100 variables. For example:

```
int a, b, c, d, e, f, g, h, i, j, k,
l, m, n... etc.,
```

An easier way is to declare an array of 100 integers:

```
int a[100];
```



The General Syntax is:

```
datatype array_name [size];
```

Subscript

Example:

```
int a[5];
```

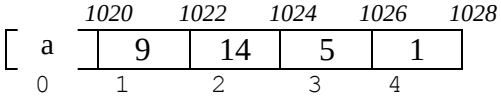
The five separate integers inside this array are accessed by an **index**. All arrays start at index zero and go to n-1 in C. Thus, `int a[5];` contains five elements. For example:

```
a[0] = 12;
a[1] = 9;
a[2] = 14;
```

```
a[3] = 5;
a[4] = 1;
```

Note: The array name will hold the address of the first element. It is called as BASE ADDRESS of that array. The base address can't be modified during execution, because it is static. It means that the increment /decrement operation would not work on the base address.

Consider the first element is stored in the address of 1020. It will look like this,



a[0] means a + 0 ➔ 1020 + 0 ➔ 1020 (locates the 1020)
a[1] means a + 1 ➔ 1020 + 1 * size of datatype ➔ 1020 + 2 ➔ 1022 [for 'int' size is 2 byte]
a[2] means a + 2 ➔ 1020 + 2 * size of datatype ➔ 1020 + 4 ➔ 1024
a[3] means a + 3 ➔ 1020 + 3 * size of datatype ➔ 1020 + 6 ➔ 1026
a[4] means a + 4 ➔ 1020 + 4 * size of datatype ➔ 1020 + 8 ➔ 1028
Array indexing helps to manipulate the index using a for loop. Because of that retrieval of element from an array is very easy. For example, the following code ***initializes all of the values in the array to 0:***

```
int a[5]; /* Array declaration */
int i;

/* Initializing Array Elements to
0 */
for (i=0; i<5; i++)
    a[i] = 0;

/* print array */
printf("Elements in the array
are…\n");
for (i=0; i < 5; i++)
      printf("%d\n",a[i]);
```

Note : (*mathematics*) A matrix most of whose entries are zeros.

Advantages:

- Reduces memory access time, because all the elements are stored sequentially. By incrementing the index, it is possible to access all the elements in an array.
- Reduces no. of variables in a program.
- Easy to use for the programmers.

Disadvantages:

- Wastage of memory space is possible. *For example: Storing only 10 elements in a 100 size array. Here, remaining 90 elements space is waste because these spaces can't be used by other programs till this program completes its execution.*
- Storing heterogeneous elements are not possible.
- Array bound checking is not available in 'C'. So, manually we have to do that.

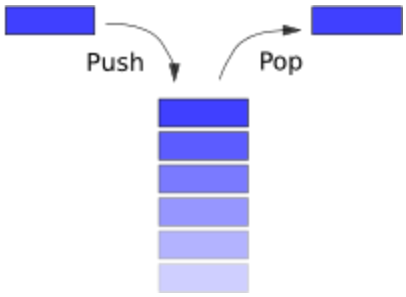
STACK :

"A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top".* stacks are sometimes referred to as Last In First Out (LIFO) lists

Stacks have some useful terminology associated with them:

- **Push** To add an element to the stack
- **Pop** To remove an element from the stock
- **Peek** To look at elements in the stack without removing them
- **LIFO** Refers to the last in, first out behavior of the stack
- **FILO** Equivalent to LIFO

stack (data structure)



Simple representation of a stack

Given a stack S=(a[1],a[2],.....a[n]) then we say that a1 is the bottom most element and element a[i]) is on top of element a[i-1], 1<i<=n.

Implementation of stack :

1. array (static memory).
2. linked list (dynamic memory)

The operations of stack is

1. PUSH operations
2. POP operations
3. PEEK operations

The Stack ADT

A stack S is an abstract data type (ADT) supporting the following three methods:

push(n) : Inserts the item ***n*** at the top of stack

pop() : Removes the top element from the stack and returns that top element. An error occurs if the stack is empty.

peek(): Returns the top element and an error occurs if the stack is empty.

1. Adding an element into a stack. (called PUSH operations)

Adding element into the TOP of the stack is called PUSH operation.

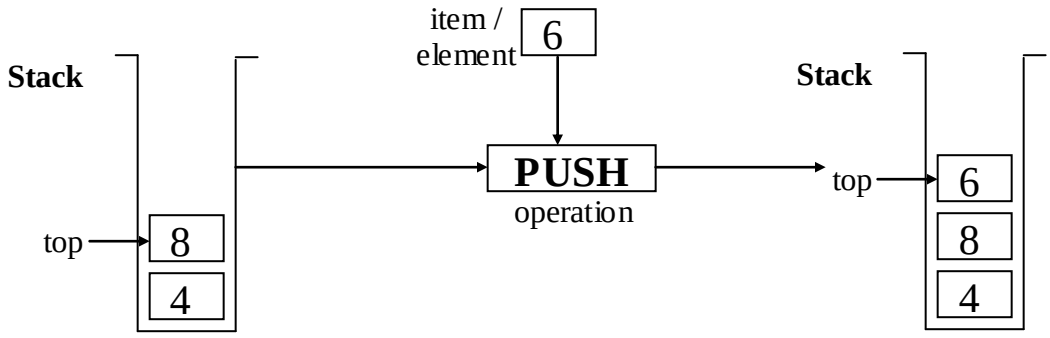
Check conditions :

TOP = N , then **STACK FULL**

where N is maximum size of the stack.

Adding into stack (PUSH algorithm)

```
procedure add(item : items);
{add item to the global stack stack ;   top is the current top of stack
and n is its maximum size}
begin
   if top = n then stackfull;
   top := top+1;
   stack(top) := item;
end: {of add}
```



Implementation in C using array:

/* here, the variables stack, top and size are global variables */

void push (int item)

```
{
        if (top == size-1)
                printf("Stack is Overflow");
        else
        {
                top = top + 1;
                stack[top] = item;
        }
}
```

2. Deleting an element from a stack. (called POP operations)

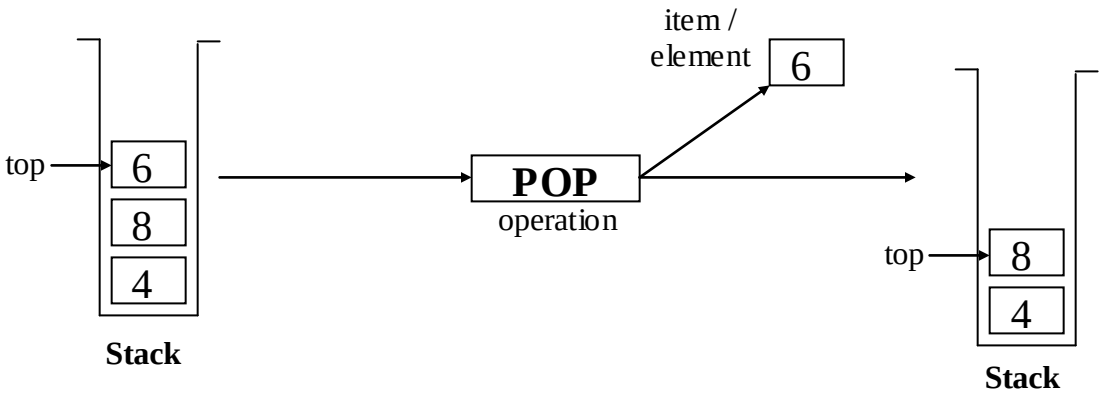
Deleting or Removing element from the TOP of the stack is called POP operations.

Check Condition:

TOP = 0, then **STACK EMPTY**

Deletion in stack (POP Operation)

```
procedure delete(var item : items);
{remove top element from the stack stack and put it in the item}
begin
   if top = 0 then stackempty;
   item := stack(top);
   top := top-1;
end; {of delete}
```



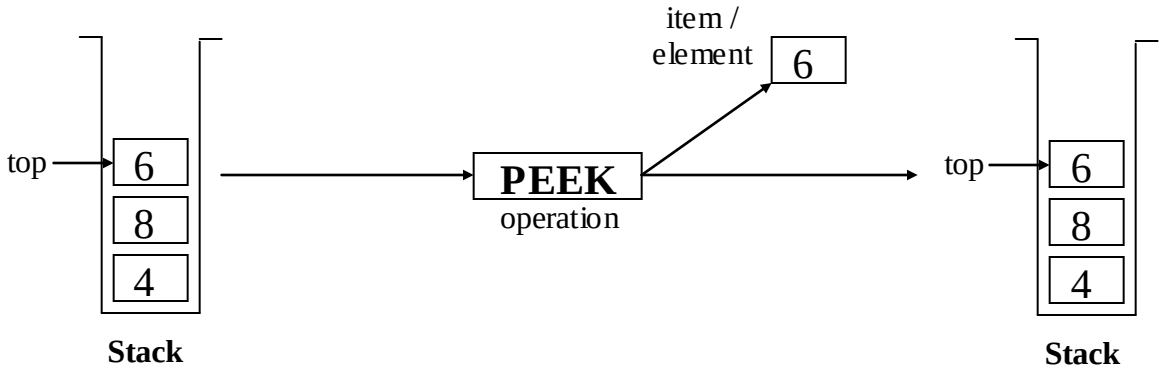
Implementation in C using array:

```
/* here, the variables stack, and top are global variables */
 int pop ( )
{
        if (top == -1)
        {
                printf("Stack is Underflow");
                return (0);
```

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

3. Peek Operation:

- ✓ Returns the item at the top of the stack but does not delete it.
- ✓ This can also result in ***underflow*** if the stack is empty.



Algorithm:

PEEK(STACK, TOP)

```
BEGIN
    /* Check, Stack is empty? */
    if (TOP == -1) then
        print "Underflow" and return 0.
    else
        item = STACK[TOP]/ * stores the top element into a local variable
    */
        return item          / * returns the top element to the user */
END
```

Implementation in C using array:

```
/* here, the variables stack, and top are global variables */
 int pop ( )
{
        if (top == -1)
        {
                printf("Stack is Underflow");
                return (0);
        }
        else
        {
```

```
            return (stack[top]);
        }
}
```

Applications of Stack

1. It is very useful to evaluate arithmetic expressions. (Postfix Expressions)
2. Infix to Postfix Transformation
3. It is useful during the execution of recursive programs
4. A Stack is useful for designing the compiler in operating system to store local variables inside a function block.
5. A stack (memory stack) can be used in function calls including recursion.
6. Reversing Data
7. Reverse a List
8. Convert Decimal to Binary
9. Parsing – It is a logic that breaks into independent pieces for further processing
10. Backtracking

Note :

1. Infix notation A+(B*C)

equivalent Postfix notation ABC*+

2. Infix notation (A+B)*C

Equivalent Postfix notation AB+C*

Expression evaluation and syntax parsing

Calculators employing reverse Polish notation (also known as **postfix notation**)use a stack structure to hold values.

Expressions can be represented in prefix, postfix or infix notations. Conversion from one form of the expression to another form needs a stack. Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code. Most of the programming languages are context-free languages allowing them to be parsed with stack based machines. Note that natural languages are context sensitive languages and stacks alone are not enough to interpret their meaning.

Infix, Prefix and Postfix Notation

We are accustomed to write arithmetic expressions with the operation between the two operands: **a+b** or **c/d**. If we write **a+b*c**, however, we have to apply precedence rules to avoid the ambiguous evaluation (add first or multiply first?).

There's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated.

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c		
40 - 3 * 5 + 1		

Examples of use: (application of stack)
Arithmetic Expressions: Polish Notation

- An arithmetic expression will have operands and operators.
- Operator precedence listed below:

  Highest : ($)
  Next Highest : (*) and (/)
  Lowest : (+) and (-)

- For most common arithmetic operations, the operator symbol is placed in between its two operands. This is called ***infix notation***.

  *Example: A + B , E * F*
- Parentheses can be used to group the operations.

  *Example: (A + B) * C*
- Accordingly, the order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

- Polish notation refers to the notation in which the operator symbol is placed before its two operands. This is called ***prefix notation***.

  *Example: +AB, *EF*
- The fundamental property of polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression.
- Accordingly, one never needs parentheses when writing expressions in Polish notation.

- ***Reverse Polish Notation*** refers to the analogous notation in which the operator symbol is placed after its two operands. This is called ***postfix notation***.

  *Example: AB+, EF**
- Here also the parentheses are not needed to determine the order of the operations.

- *The computer usually evaluates an arithmetic expression written in infix notation in two steps,*
    1. *It converts the expression to **postfix notation**.*
    2. *It evaluates the postfix expression.*
- *In each step, the stack is the main tool that is used to accomplish the given task.*

(1)Question : (Postfix evaluation)

How to evaluate a mathematical expression using a stack The algorithm for Evaluating a postfix expression ?

- Initialise an empty stack
- While token remain in the input stream
    - Read next token
    - If token is a number, push it into the stack
    - Else, if token is an operator, pop top two tokens off the stack, apply the operator, and push the answer back into the stack
- Pop the answer off the stack.

Algorithm postfixexpression

Initialize a stack, opndstk to be empty.

```
{scan the input string reading one element at a time into symb }
While ( not end of input string )
        {
        Symb := next input character;

        If symb is an operand Then

                push (opndstk,symb)
        Else
                [symbol is an operator]
                {
                Opnd1:=pop(opndstk);
                Opnd2:=pop(opndnstk);
                Value := result of applying symb to opnd1 & opnd2
                Push(opndstk,value);
                }
Result := pop (opndstk);
```

Example:
*6 2 3 + - 3 8 2 / + * 2 $ 3 +*

Symbol	*Operand 1 (A)*	*Operand 2 (B)*	*Value (A ⊗ B)*	*STACK*
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3				1, 3
8				1, 3, 8
2				1, 3, 8, 2
/	8	2	/	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2				7, 2
$	7	2	49	49
3				49, 3
+	49	3	52	***52***

The Final value in the STACK is 52. This is the answer for the given expression.

(2) run time stack for function calls (write factorial number calculation procedure)

push local data and return address onto stack
return by popping off local data and then popping off address and returning to it
return value can be pushed onto stack before returning, popped off by caller

(3) expression parsing

e.g. matching brackets: [... (... (...) [...(...) ...] ...) ...]
push left ones, pop off and compare with right ones

4) INFIX TO POSTFIX CONVERSION

Infix expressions are often translated into postfix form in which the operators appear after their operands. **Steps:**

1. Initialize an empty stack.
2. Scan the Infix Expression from left to right.
3. If the scannned character is an operand, add it to the Postfix Expression.
4. If the scanned character is an operator and if the stack is empty, then push the character to stack.

5. If the scanned character is an operator and the stack is not empty, Then

(a) Compare the precedence of the character with the operator on the top of the stack.

(b) While operator at top of stack has higher precedence over the scanned character & stack
is not empty.
(i) POP the stack.
(ii) Add the Popped character to Postfix String.
(c) Push the scanned character to stack.

6. Repeat the steps 3-5 till all the characters
7. While stack is not empty,
(a) Add operator in top of stack
(b) Pop the stack.
8. Return the Postfix string.

Algorithm Infix to Postfix conversion (without parenthesis)

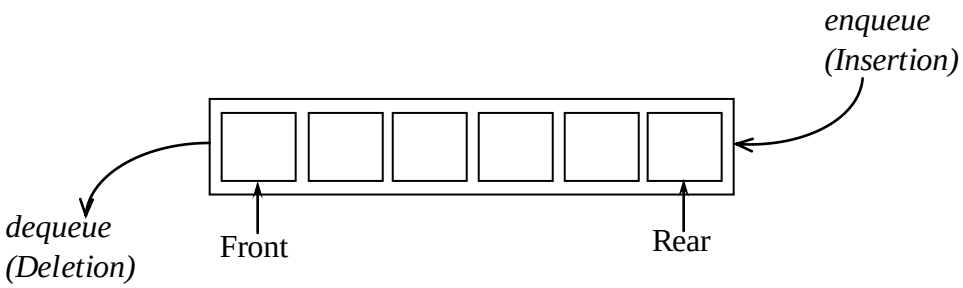
1. Opstk = the empty stack;
2. while (not end of input)
{
symb = next input character;

3. if (symb is an operand)
add symb to the Postfix String
4. else
{
5. While(! empty (opstk) && prec (stacktop (opstk), symb))
{
topsymb = pop (opstk)
add topsymb to the Postfix String;
} / * end of while */
Push(opstk, symb);
} /* end else */
6. } /* end while * /

7. While(! empty (opstk))
{
topsymb = pop (opstk)
add topsymb to the Postfix String
} /* end of while */
8. Return the Postfix String.

QUEUE :

"A queue is an ordered list in which all insertions at one end called REAR and deletions are made at another end called FRONT". *queue*s are sometimes referred to as First In First Out (FIFO) lists.



Example

1. The people waiting in line at a bank cash counter form a queue.
2. In computer, the jobs waiting in line to use the processor for execution. This queue is called ***Job Queue***.

Operations Of Queue

There are two basic queue operations. They are,

Enqueue – Inserts an item / element at the rear end of the queue. An error occurs if the queue is full.

Dequeue – Removes an item / element from the front end of the queue, and returns it to the user. An error occurs if the queue is empty.

1. Addition into a queue

```
procedure addq (item : items);
{add item to the queue q}
begin
   if rear=n then queuefull
   else begin
      rear :=rear+1;
      q[rear]:=item;
   end;
end;{of addq}
```

2. Deletion in a queue

```
procedure deleteq (var item : items);
{delete from the front of q and put into item}
begin
   if front = rear then queueempty
   else begin
      front := front+1
      item := q[front];
   end;
end
```

Uses of Queues (Application of queue)

Queues remember things in first-in-first-out (FIFO) order. Good for fair (first come first served) ordering of actions.

Examples of use: (Application of stack)

1• scheduling

processing of GUI events
printing request

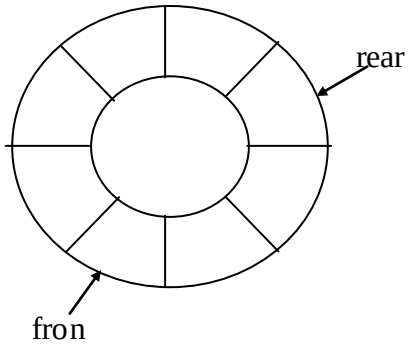
2• simulation

orders the events
models real life queues (e.g. supermarkets checkout, phone calls on hold)

Circular Queue :

Location of queue are viewed in a circular form. The first location is viewed after the last one.

Overflow occurs when all the locations are filled.



Algorithm Circular Queue Insert

```
Void  CQInsert ( int queue[ ], front, rear, item)
        {
        if ( front = = 0 )
                front = front +1;
        if ( ( ( rear = maxsize ) && ( front = = 1 ) )  || ( ( rear ! = 0 ) && ( front = rear
+1)))
         {
         printf( " queue overflow ");

         if( rear = = maxsize )
                rear = 1;
         else
```

```
            rear = rear + 1;
            q [ rear ] = item;
        }
    }
```

Algorithm Circular Queue Delete

```
int CQDelete ( queue [ ], front, rear )
    {
    if ( front = = 0 )
            printf ( " queue underflow ");
    else
      {
       item = queue [ front ];
       if(front = = rear )
            {
            front = 0; rear = 0;
            }
    else if ( front = = maxsize )
            {
            front = 1;
            }
    else
            front = front + 1;
     }
    return item;
   }
```

Priority Queue

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

Two types of queue are

1. Ascending Priority Queue
2. Descending Priority Queue

1. Ascending Priority Queue

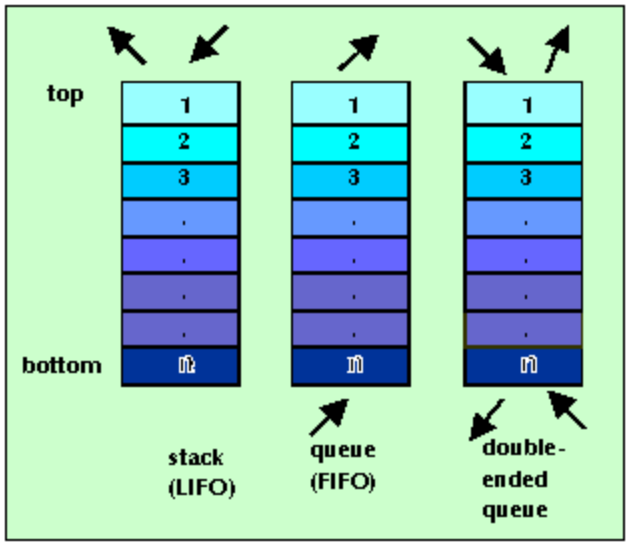
Collection of items into which item can be inserted arbitrarily & from which only the Smallest item can be removed.

2. **Descending Priority Queue**

Collection of items into which item can be inserted arbitrarily & from which only the Largest item can be removed.

Double Ended Queue

A **deque** (short for *double-ended queue*) is an abstract data structure for which elements can be added to or removed from the front or back(both end). This differs from a normal queue, where elements can only be added to one end and removed from the other. Both queues and stacks can be considered specializations of deques, and can be implemented using deques.



double-ended queue

Two types of Dqueue are

1. Input Restricted Dqueue
2. Ouput Restricted Dqueue.

1. Input Restricted Dqueue

Where the input (insertion) is restricted to the rear end and the deletions has the options either end

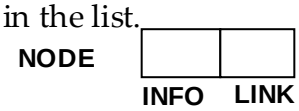
2. Ouput Restricted Dqueue.

Where the output (deletion) is restricted to the front end and the insertions has the option either end.

Example: *Timesharing system using the prototype of priority queue – programs of high priority are processed first and programs with the same priority form a standard queue.*

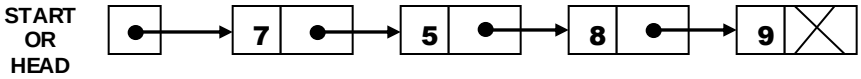
Linked List

- Some demerits of array, leads us to use linked list to store the list of items. They are,
    1. It is relatively expensive to insert and delete elements in an array.
    2. Array usually occupies a block of memory space, one cannot simply double or triple the size of an array when additional space is required. *(For this reason, arrays are called* ***"dense lists"*** *and are said to be* ***"static"*** *data structures.)*
- A ***linked list***, or ***one-way list***, is a linear collection of data elements, called ***nodes***, where the linear order is given by means of ***pointers***. That is, each node is divided into two parts:
    - ✓ The first part contains the information of the element i.e. INFO or DATA.
    - ✓ The second part contains the ***link field***, which contains the address of the next node in the list.



- The linked list consists of series of nodes, which are not necessarily adjacent in memory.
- A list is a ***dynamic data structure*** i.e. the number of nodes on a list may vary dramatically as elements are inserted and removed.
- The pointer of the last node contains a special value, called the ***null*** pointer, which is any invalid address. This ***null pointer*** signals the end of list.
- The list with no nodes on it is called the ***empty list*** or ***null list***.

Example*: The linked list with 4 nodes.*



Types of Linked Lists:

a) Linear Singly Linked List
b) Circular Linked List
c) Two-way or doubly linked lists
d) Circular doubly linked lists

Advantages of Linked List

1. Linked List is dynamic data structure; the size of a list can grow or shrink during the program execution. So, maximum size need not be known in advance.
2. The Linked List does not waste memory
3. It is not necessary to specify the size of the list, as in the case of arrays.
4. Linked List provides the flexibility in allowing the items to be rearranged.

What are the pitfalls encountered in single linked list?

1. A singly linked list allows traversal of the list in only one direction.
2. Deleting a node from a list requires keeping track of the previous node, that is, the node whose link points to the node to be deleted.
3. If the link in any node gets corrupted, the remaining nodes of the list become unusable.

Linearly-linked List

Is a collection of elements called Nodes. Each node consist of two fields, namely data field to hold the values and link(next) field points to the next node in the list.

It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes.

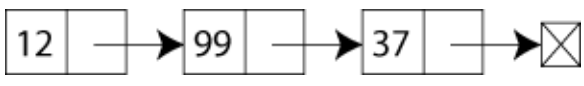
A linked list is a self-referential datatype (or) data structure because it contains a pointer or link to another data of the same type.

Linked lists permit insertion and removal of nodes at any point in the list in constant time, **but do not allow random access**.

Several different types of linked list exist: **singly-linked lists, doubly-linked lists, and circularly-linked lists.** One of the biggest advantages of linked lists is that nodes may have multiple pointers to other nodes, allowing the same nodes to simultaneously appear in different orders in several linked lists

Singly-linked list

The simplest kind of linked list is a **singly-linked list** (or **slist** for short), which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.

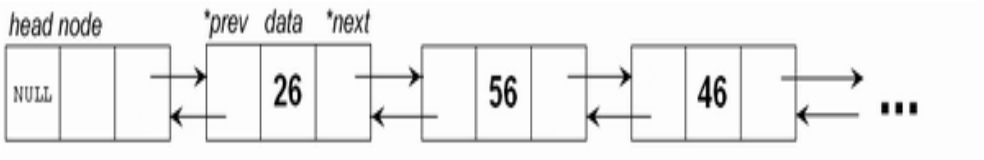


A singly linked list containing three integer values

Doubly-linked list

A more sophisticated kind of linked list is a **doubly-linked list** or **two-way linked list**. Each node has two links: one points to the previous node, or points to a null value or

empty list if it is the first node; and one points to the next, or points to a null value or empty list if it is the final node.



An example of a doubly linked list.

Circularly-linked list

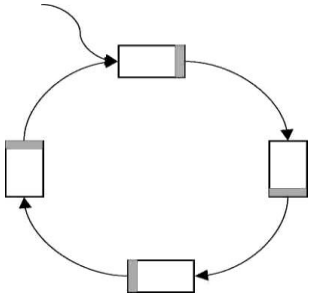
In a **circularly-linked list**, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, you begin at any node and follow the list in either direction until you return to the original node. Viewed another way, circularly-linked lists can be seen as having no beginning or end. This type of list is most useful for managing buffers for data ingest, and in cases where you have one object in a list and wish to see all other objects in the list.

The pointer pointing to the whole list is usually called the end pointer.

Singly-circularly-linked list

In a **singly-circularly-linked list**, each node has one link, similar to an ordinary *singly-linked list*, except that the next link of the last node points back to the first node.

As in a singly-linked list, new nodes can only be efficiently inserted after a node we already have a reference to. For this reason, it's usual to retain a reference to only the last element in a singly-circularly-linked list, as this allows quick insertion at the beginning, and also allows access to the first node through the last node's next pointer.



- Note that there is no **NULL** terminating pointer
- Choice of **head** node is arbitrary
- A **tail** pointer serves no purpose
- *What purpose(s) does the* **head** *pointer serve?*

Doubly-circularly-linked list

In a **doubly-circularly-linked list**, each node has two links, similar to a *doubly-linked list*, except that the previous link of the first node points to the last node and the next link of the last node points to the first node. As in doubly-linked lists, insertions and removals can be done at any point with access to any nearby node.

Sentinel nodes

Linked lists sometimes have a special *dummy* or *sentinel node* at the beginning and/or at the end of the list, which is not used to store data.

Basic Operations on Linked Lists

1. Insertion
    a. At first
    b. At last
    c. At a given location (At middle)
2. Deletion
    a. First Node
    b. Last Node
    c. Node in given location or having given data item

Initial Condition

HEAD = NULL;
/* Address of the first node in the list is stored in HEAD. Initially there is no node in the list. So, HEAD is initialized to NULL (No address) */

What are the Applications of linked list?

- To implement of Stack, Queue, Tree, Graph etc.,
- Used by the Memory Manager
- To maintain Free-Storage List

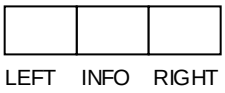
Doubly Linked Lists (or) Two – Way Lists

There are some problems in using the Single linked list. They are

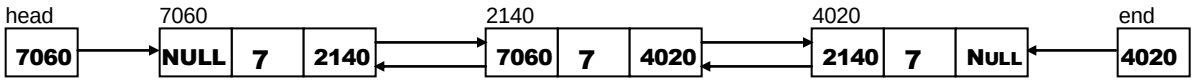
1. A singly linked list allows traversal of the list in only one direction. (Forward only)
2. Deleting a node from a list requires keeping track of the previous node, that is, the node whose link points to the node to be deleted.

These major drawbacks can be avoided by using the double linked list. The doubly linked list is a ***linear collection*** of data elements, called ***nodes***, where each node is divided into three parts. They are:

1. *A pointer field LEFT which contains the address of the preceding node in the list*
2. *An information field INFO which contains the data of the Node*
3. *A pointer field RIGHT which contains the address of the next node in the list*



Example:



Linked lists vs. arrays

	Array	Linked list
Indexing	O(1)	O(*n*)
Inserting / Deleting at end	O(1)	O(1)
Inserting / Deleting in middle (with iterator)	O(*n*)	O(1)
Persistent	No	Singly yes
Locality	Great	Bad

Array	Linked list
Static memory	Dynamic memory
Insertion and deletion required to modify the existing element location	Insertion and deletion are made easy.
Elements stored as contiguous memory as on block.	Element stored as Non-contiguous memory as pointers
Accessing element is fast	Accessing element is slow

SINGLY LINKED LISTS

1. Insertion of a Node in the Beginning of a List

Step 1 : Allocate memory for a node and assign its address to the variable ' New'
Step 2 : Assign the element in the data field of the new node.
Step 3 : Make the next field of the new node as the beginning of the existing list.
Step 4 : Make the new node as the Head of the list after insertion.

Algorithm InsertBegin (Head, Elt)
[Adding the element **elt** in the beginning of the list pointed by Head]

1. new ← getnode (NODE)
2. data (new) ← elt
3. next (new) ← Head
4. Head ← new
5. return Head

Insertion of a Node at the End of a Singly Linked List

Step 1 : Allocate memory for a node and assign its address to the variable ' New'
Step 2 : Assign the element in the data field of the new node.
Step 3 : Make the next field of the new node as NULL This is because the new node
will be the end of the resultant list.
Step 4 : If the existing list is empty, call this new node as the list. Else, get the address of the last node in the list by traversing from the beginning pointer.
Step 5: Make the next field of the last node point to the new node.

Algorithm InsertEnd (Head, Elt)
[Adding the element **elt** at the end of the list]

1. new ← getnode (NODE)
2. data (new) ← elt
3. next (new) ← NULL
4. if (Head == NULL) Then
   Head ← new
   Return Head
   Else
   Temp ← Head
5. While (next (temp) # NULL)
   temp ← next (temp)
6. next (temp) ← new
7. return Head.

Applications of linked lists

Linked lists are used as a building block for many other data structures, such as stacks, queues and their variations.

1. Polynomial ADT:

A polynomial can be represented with primitive data structures. For example, a polynomial represented as $a_k x^k$ $a_{k-1} x^{k-1}$ + ... + a_0 can be represented as a linked list. Each

node is a structure with two values: a_i and i. Thus, the length of the list will be k. The first node will have (a_k, k), the second node will have (a_{k-1}, k-1) etc. The last node will be (a_0, 0).
The polynomial $3x^9 + 7x^3 + 5$ can be represented in a list as follows: (3,9) --> (7,3) --> (5,0) where each pair of integers represent a node, and the arrow represents a link to its neighbouring node.
Derivatives of polynomials can be easily computed by proceeding node by node. In our previous example the list after computing the derivative would represented as follows: (27,8) --> (21,2). The specific polynomial ADT will define various operations, such as multiplication, addition, subtraction, derivative, integration etc. A polynomial ADT can be useful for symbolic computation as well.

2. Large Integer ADT:

Large integers can also be implemented with primitive data structures. To conform to our previous example, consider a large integer represented as a linked list. If we represent the integer as successive powers of 10, where the power of 10 increments by 3 and the coefficent is a three digit number, we can make computations on such numbers easier. For example, we can represent a very large number as follows:
$513(10^6) + 899(10^3) + 722(10^0)$.
Using this notation, the number can be represented as follows:
(513) --> (899) --> (722).
The first number represents the coefficient of the 10^6 term, the next number represents the coefficient of the 10^3 term and so on. The arrows represent links to adjacent nodes. The specific ADT will define operations on this representation, such as addition, subtraction, multiplication, division, comparison, copy etc.

An array allocates memory for all its elements lumped together as one block of memory. In contrast, a linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets is overall structure by using pointers to connect all its nodes together like the links in a chain.
Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field which is a pointer used to link one node to the next node.

Each node is allocated in the heap with a call to malloc(), so the node memory continues to exist until it is explicitly deallocated with a call to free(). The front of the list is a pointer to the first node. Here is what a list containing the numbers 1, 2, and 3 might look like...

malloc() malloc() is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype for malloc() and other heap functions are in `stdlib.h`. The argument to malloc() is the integer size of the block in bytes. Unlike local ("stack") variables, heap memory is not automatically deallocated when the creating function exits. malloc() returns NULL if it cannot fulfill the request. (extra for experts) You may check for the NULL case with assert() if you wish just to be safe. Most modern programming systems will throw an exception or do some other

automatic error handling in their memory allocator, so it is becoming less common that source code needs to explicitly check for allocation failures.
free() free() is the opposite of malloc(). Call free() on a block of heap memory to indicate to the system that you are done with it. The argument to free() is a pointer to a block of memory in the heap — a pointer which some time earlier was obtained via a call to malloc().

Sequential list: contiguous cells, indexed by location

Linked list: noncontiguous cells linked by pointers, implicitly indexed by number of links away from head

Both also contain
location of first element (head)
length or end-of-list marker
Examples of end-of-list marker:
'\0' for C strings (sequential list)
NULL for C linked list

Representation

Sequential list: contiguous cells, indexed by location **Linked list:** noncontiguous cells linked by pointers, implicitly indexed by number of links away from head
Both also contain
location of first element (head)
length or end-of-list marker
Examples of end-of-list marker:
'\0' for C strings (sequential list)
NULL for C linked list

Sequential List implementation

In C: typically an array with an int for the last used index.
A problem: must reserve memory for the list to grow into.
limits length of list to reserved length
reserved memory unusable for other purposes
Main *advantage* of sequential list: fast access to element by index.

Linked List implementation

In C: typically individual cells dynamically allocated containing a pointer to the next cell.

Advantages:

space used adapts to size • usually results in better space usage than sequential despite storing a pointer in each cell

speed improvements for some operations
Disadvantages:

• speed reductions for some operations

Time efficiency 1

In the following, let n be the length of the list.
Initialize to empty list
Sequential and Linked O(1)
For sequential lists this really depends on the complexity of memory allocation, a complex subject in itself. For linked lists, memory allocation time can affect the performance of the other operations.
Select ith element
Sequential O(1) Linked O(i)

Time efficiency 2
Determine length
Sequential and Linked
O(1) if recorded
O(n) for marker – then linked takes longer following pointers and is more likely to leave the cache
Traverse
Sequential and Linked O(n)
Linked takes longer for reasons above, but may be
insignificant compared to processing done on the elements.
Search

Sequential ordered O(log n)
Sequential unordered, Linked O(n)
Linked takes longer for reasons above.
Ordering can improve linked on average, since we can more
quickly detect that an element isn't in the list.

Time efficiency 3
Changes
These depend of course on how we locate where to make the change. The following describe the additional cost.
Delete or insert element
Sequential O(n * size of an element) We must move elements over! Linked O(1)
Replace element (unordered)
Sequential and Linked O(1)
Replace data in an element
Sequential and Linked O(1)

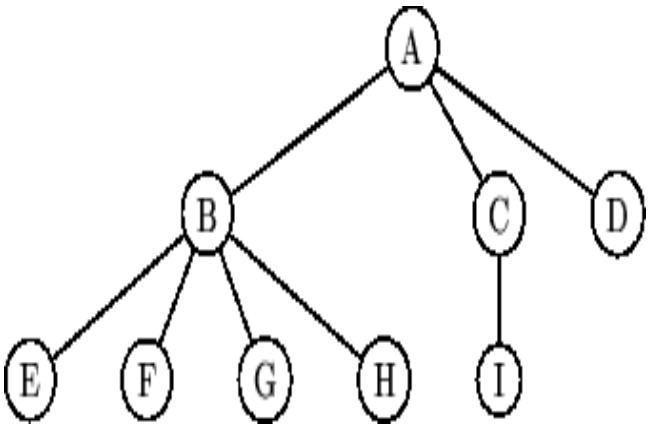
Variants
There are other list implementations. For example:
Doubly Linked list

allows efficient backwards traversal
takes longer to insert and delete (but the same complexity)
takes more space for the extra pointer (unless we use the for trick to save space at the cost of time)
Circular list (head and tail linked)

Trees

The ADT tree

A **tree** is a finite set of elements or **nodes**. If the set is non-empty, one of the nodes is distinguished as the **root** node, while the remaining (possibly empty) set of nodes are grouped into subsets, each of which is itself a tree. This hierarchical relationship is described by referring to each such subtree as a **child** of the root, while the root is referred to as the **parent** of each subtree. If a tree consists of a single node, that node is called a **leaf** node.



A simple tree.

Basic Tree Concepts

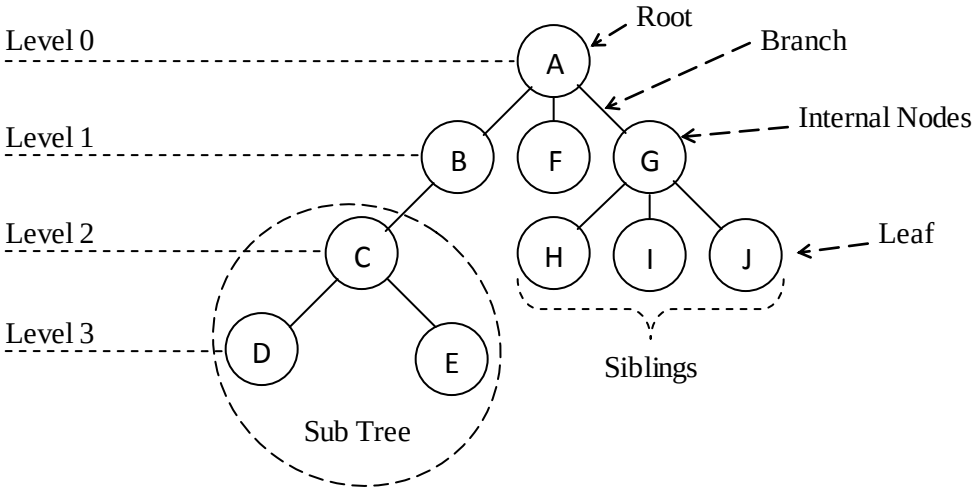
- A tree consists of a finite set of elements, called ***'nodes'***, and a finite set of directed lines, called ***'branches'***, that connect the nodes.

- The number of branches associated with a node is the ***degree of the node***.

    - When the branch is directed toward a node, it is an ***indegree branch***; when the branch is directed away from the node, it is an ***outdegree branch***.
    - *The sum of indegree and outdegree branches is the degree of the node.*
    - The indegree of the root is by definition is zero.

- A ***leaf*** is any node with an outdegree of zero.

- A node that is not a root or a leaf is known as an ***internal node***.
- A node is a ***parent*** if it has ***successor*** nodes – that is, if it has an outdegree greater than zero.
- Conversely, a node with a **predecessor** is a ***child***. A child node has an indegree of one.
- Two or more nodes with the same parent are ***siblings***.

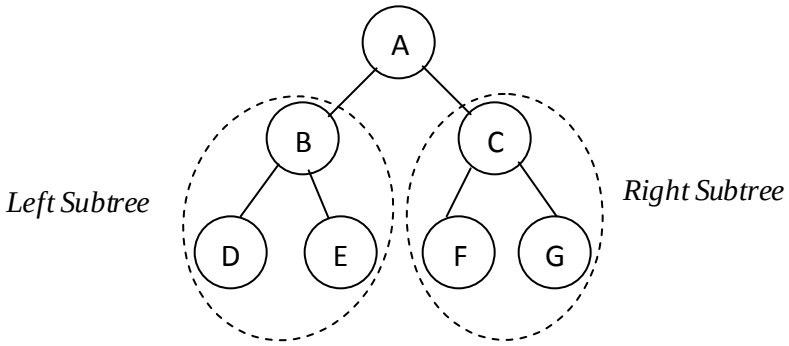


Parents	:	A, B, C, G
Children	:	B, F, G, C, H, I, J, D, E
Siblings	:	{ B, F, G }, { D, E }, { H, I, J }
Leaves	:	F, D, E, H, I, J
Length	:	4

- A ***path*** is a sequence of nodes in which each node is adjacent to the next one.
- Every node in the tree can be reached by following a ***unique path*** starting from the root.
- The ***level*** of a node is its distance from the root. Because the root has a zero distance from itself, the root is at level 0. The children of the root are at the level 1.
- The ***height*** or ***length*** of the tree is the level of the leaf in the longest path from the root plus 1. By definition, the height of an empty tree is -1.
- A tree may be divided into ***subtrees***. A ***subtree*** is any connected structure below the root.
- The first node in a subtree is known as the ***root of the subtree*** and is used to name the subtree.

Binary Trees

- A ***binary tree*** is a tree in which no node can have more than two subtrees.
- These subtrees are designated as the ***left subtree*** and ***right subtree***.
- Each subtree is a binary tree itself.



- The ***height of the binary trees*** can be mathematically predicted. The maximum height of the binary tree which has N nodes,

$$H_{max} = N$$

- A tree with a maximum height is rare. It occurs when the entire tree is built in one direction. The ***minimum height of the tree***, H_{min} is determined by,

$$Hmin = \lfloor Log_2 N \rfloor + 1$$

- Given a height of the binary tree, H, the minimum and maximum number of nodes in the tree are given as,

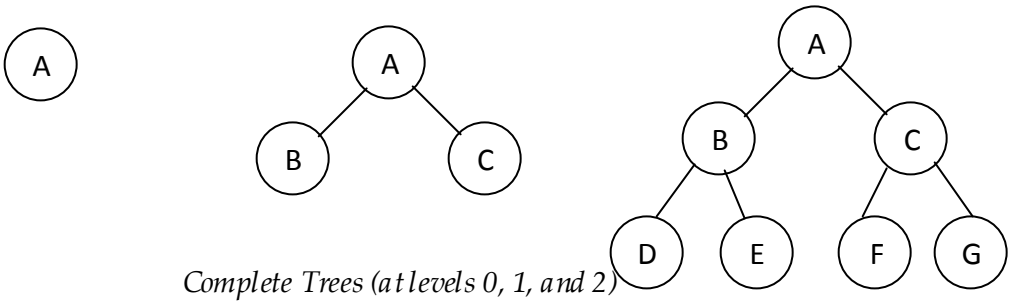
$$N_{min} = H \quad \textbf{and,}$$

$$N_{max} = 2^H - 1$$

- If the height of the tree is less, then it is easier to locate any desired node in the tree.
- To determine whether tree is balanced, the ***balance factor*** should be calculated.
- If H_L represents the height of the left subtree and H_R represents the height of the right subtree then ***Balance factor***,

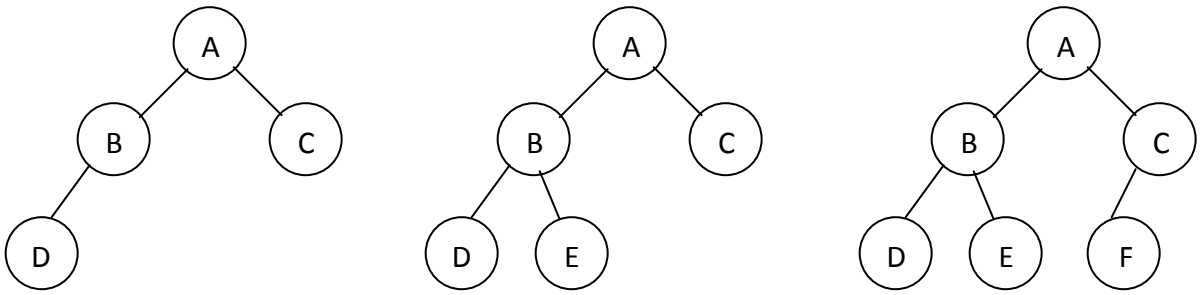
$$B = H_L - H_R$$

- A tree is balanced if its balance factor is ***0*** and its subtrees are also balanced.
- A binary tree is balanced if the *height of its subtrees differs by no more than one and its subtrees are also balanced*.
- A ***complete tree*** has the maximum number of entries for its height.
- The maximum number is reached when the least level is full. *The maximum number is reached when the last level is full*.



Complete Trees (at levels 0, 1, and 2)

- A tree is considered ***nearly complete*** if it has the minimum height for its nodes and all nodes in the last level are found on the left.



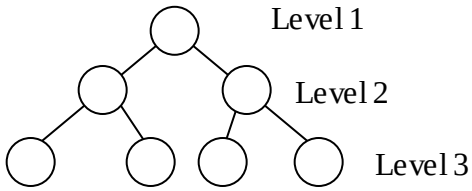
Nearly Complete Trees (at level 2)

BINARY TREE REPRESENTATION

A **binary tree** is a tree which is either empty, or one in which every node:

- has no children; or
- has just a left child; or
- has just a right child; or
- has both a left and a right child.

A Complete binary tree of depth K is a binary tree of depth K having 2k – 1 nodes.



A very simple representation for such binary tree results from sequentially numbering the nodes, starting with nodes on level 1 then those on level 2 and so on. Nodes on any level are numbered from left to right as shown in the above picture. This numbering scheme gives us the definition of a complete binary tree. A binary tree with n nodes and of depth K is complete if its nodes correspond to the nodes which are numbered one to n in the full binary tree of depth K.

Array Representation:

Each node contains **info, left, right** and **father** fields. The left, right and father fields of a node point to the node's left son, right son and father respectively.

Using the array implementation, we may declare,

```
#define NUMNODES 100
struct nodetype
        {
        int info;
        int left;
        int right;
        int father;
        };
struct nodetype node[NUMNODES];
```

This representation is called linked array representation.

Under this representation,

info(p) would be implemented by reference **node[p].info,**
left(p) would be implemented by reference **node[p].left,**
right(p) would be implemented by reference **node[p].right,**
father(p) would be implemented by reference **node[p].father**
respectively.

The operations,

isleft(p) can be implemented in terms of the operation **left(p)**
isright(p) can be implemented in terms of the operation **right(p)**

Example: -

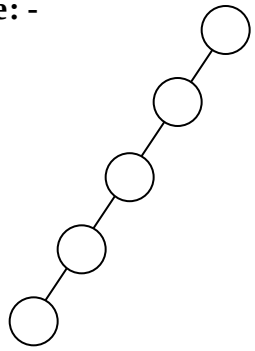


Fig (a)

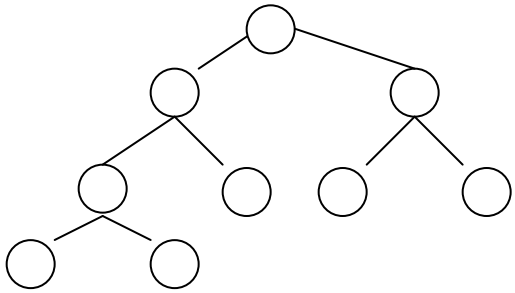


Fig (b)

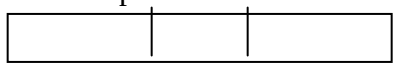
The above trees can be represented in memory sequentially as follows

A
B
-
C
-
-
-
D
-
E

The above representation appears to be good for complete binary trees and wasteful for many other binary trees. In addition, the insertion or deletion of nodes from the middle of a tree requires the insertion of many nodes to reflect the change in level number of these nodes.

Linked Representation: -

The problems of sequential representation can be easily overcome through the use of a linked representation. Each node will have three fields LCHILD, DATA and RCHILD as represented below



LCHILD DATA RCHILD

T

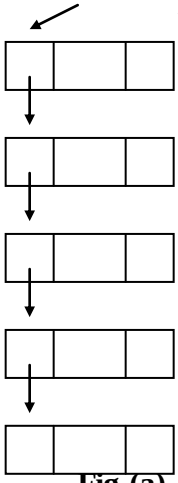


Fig (a)

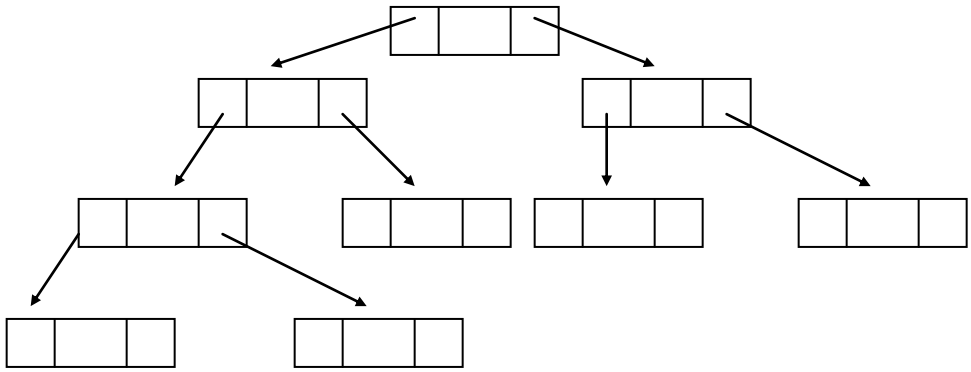


Fig (b)

In most applications it is adequate. But this structure make it difficult to determine the parent of a node since this leads only to the forward movement of the links.

Using the linked implementation, we may declare,

```
stuct nodetype
        {
        int info;
        struct nodetype *left;
        struct nodetype *right;
        struct nodetype *father;
        };
typedef struct nodetype *NODEPTR;
```

This representation is called **dynamic node representation.** Under this representation,

info(p) would be implemented by reference **p→info,**
left(p) would be implemented by reference **p→left,**
right(p) would be implemented by reference **p→right**,
father(p) would be implemented by reference **p→father**.

PRIMITIVE OPERATION ON BINARY TREES

(1) maketree() function

Which allocates a node and sets it as the root of a single-node binary tree, may be written as follows;

```
NODEPTR maketree(x)
int x;
{
NODEPTR p;

p = getnode();          /* getnode() function get a available node */
p→info = x;
p→left = NULL;
p→right=NULL;
return(p);
}
```

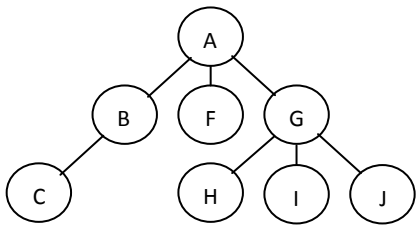
(2) setleft(p,x) function Which sets a node with contents x as the left son of node(p)

```
setleft(p,x)
NODEPTR p;
int x;
{
if(p = = NULL)
        printf("insertion not made");
else if ( p→left ! = NULL)
        printf("invalid insertion ");
else
        p→left = maketree (x);
}
```

Conversion of a General Tree to Binary Tree

General Tree:

- A General Tree is a tree in which each node can have an unlimited out degree.
- Each node may have as many children as is necessary to satisfy its requirements. *Example: Directory Structure*



- It is considered easy to represent binary trees in programs than it is to represent general trees. So, the general trees can be represented in binary tree format.

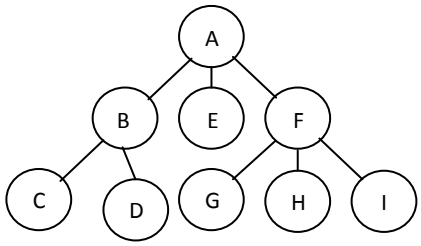
Changing general tree to Binary tree:

- The binary tree format can be adopted by changing the meaning of the left and right pointers. There are two relationships in binary tree,
  - Parent to child
  - Sibling to sibling

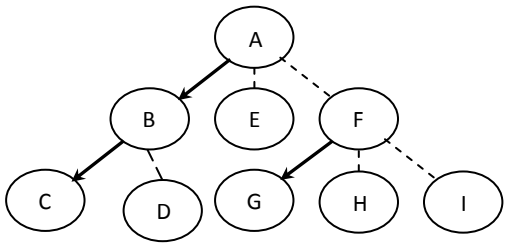
Using these relationships, the general tree can be implemented as binary tree.

Algorithm

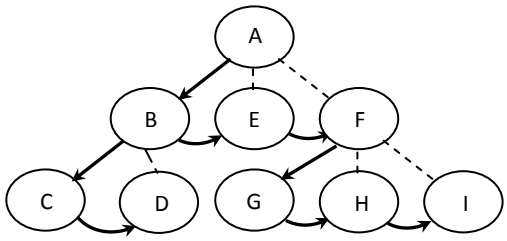
1. Identify the branch from the parent to its first or leftmost child. These branches from each parent become left pointers in the binary tree
2. Connect siblings, starting with the leftmost child, using a branch for each sibling to its right sibling.
3. Remove all unconnected branches from the parent to its children



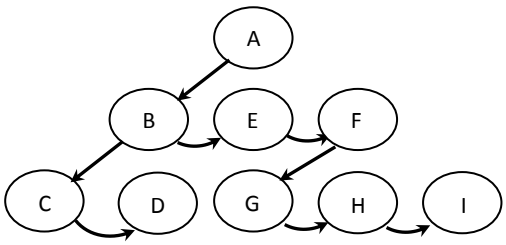
(a) General Tree



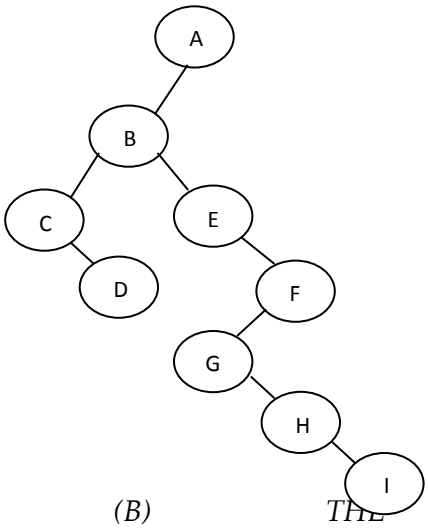
Step 1: Identify all leftmost children



Step 2: Connect Siblings



Step 3: Delete unneeded branches



(B) THE RESULTING BINARY TREE

BINARY TREE TRAVERSALS

- A binary tree traversal requires that each node of the tree be processed once and only once in a predetermined sequence.
- The two general approaches to the traversal sequence are,
  - ***Depth first traversal***
  - ***Breadth first traversal***

- In depth first traversal, the processing proceeds along a path from the root through one child to the most distant descendent of that first child before processing a second child. *In other words, in the depth first traversal, all the descendants of a child are processed before going to the next child*.
- In a breadth-first traversal, the processing proceeds horizontally form the root to all its children, then to its children's children, and so forth until all nodes have been processed. *In other words, in breadth traversal, each level is completely processed before the next level is started*.

Depth-First Traversal

There are basically three ways of binary tree traversals. They are :

1. **Pre Order Traversal**
2. **In Order Traversal**
3. **Post Order Traversal**

In C, each node is defined as a structure of the following form:

```
struct node
        {
        int  info;
        struct node *lchild;
        struct node *rchild;
        }
typedef struct node NODE;
```

Binary Tree Traversals (Recursive procedure)

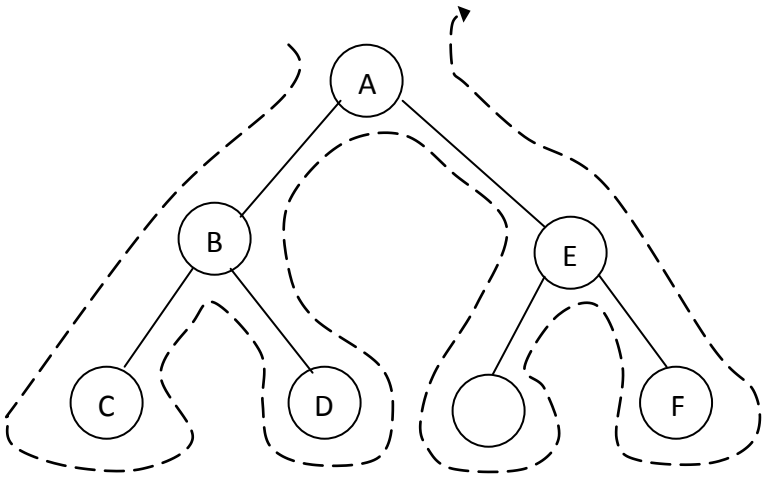
1. Inorder Traversal

Steps : 1. Traverse left subtree in inorder
2. Process root node
3. Traverse right subtree in inorder

Algorithm	C Coding
NODE * T)	void inorder_traversal (
Algorithm inoder traversal (Bin-Tree T)	{ if(T ! = NULL)

```
Begin
If ( not empty (T) ) then
Begin
        Inorder_traversal ( left subtree ( T ) )
        Print ( info ( T ) )   /  * process node */
        Inorder_traversal ( right subtree ( T ) )
End
End
```

```
{
inorder_traversal(T->lchild);
printf("%d \t ", T->info);
inorder_traversal(T->rchild);
}
```



The Output is : C → B → D →A →E → F

2. Preorder Traversal

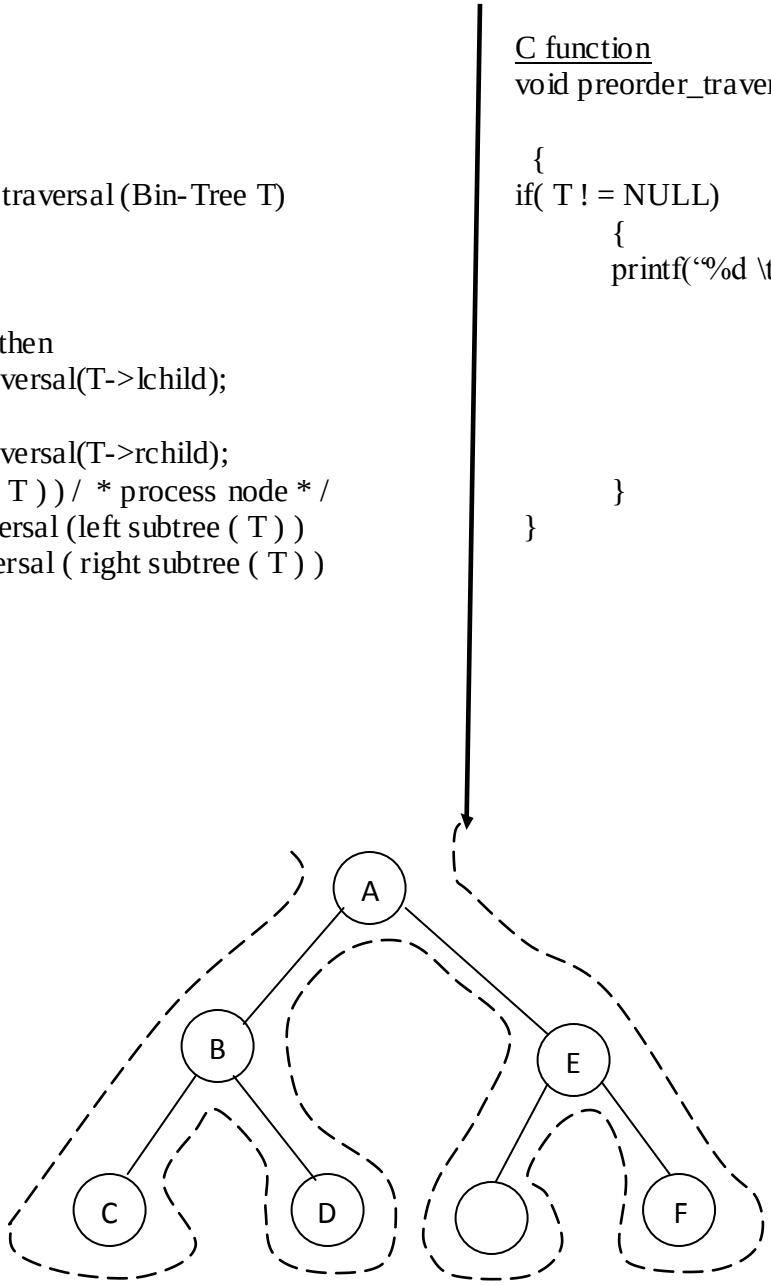
Steps : 1. Process root node
2. Traverse left subtree in preorder
3. Traverse right subtree in preorder

Algorithm

```
NODE * T)

Algorithm preorder traversal (Bin-Tree T)

Begin
>info);
If ( not empty (T) ) then
        preorder_traversal(T->lchild);
Begin
        preorder_traversal(T->rchild);
        Print ( info ( T ) ) /  * process node * /
        Preoder traversal (left subtree ( T ) )
        Inorder traversal ( right subtree ( T ) )
End
End
```

C function

```
void preorder_traversal (

  {
if( T ! = NULL)
        {
        printf("%d \t", T-

        }
 }
```



Output is : A → B → C → D → E → F

3. Postorder Traversal

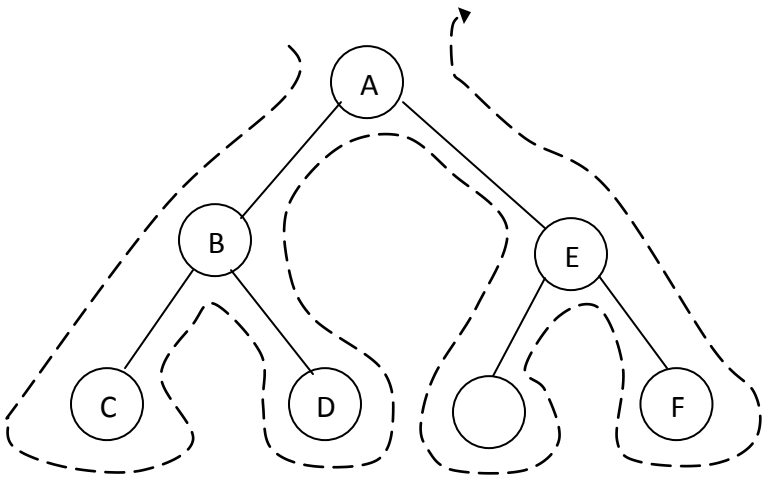
Steps : 1. Traverse left subtree in postorder
2. Traverse right subtree in postorder
3. process root node

Algorithm

```
Postorder Traversal
NODE * T)

Algorithm postorder traversal (Bin-Tree T)

Begin
        postorder_traversal(T->lchild);
If ( not empty (T) ) then
        postorder_traversal(T->rchile);
Begin
>info);
        Postorder_traversal ( left subtree ( T ) )
        Postorder_traversal ( right subtree( T))
        Print ( Info ( T ) ) / * process node */
End
End
```

C function

```
void postorder_traversal (

    {
    if( T ! = NULL)
            {

                    printf("%d \t", T-

            }
     }
```



The Output is : C → D → B → F → E → A

Non – Recursive algorithm: Inorder_Traversal

```
#define MAXSTACK 100

inorder_traversal (tree)
NODEPTR tree;
        {
        struct stack
                {
                int top;
                NODEPTR item[MAXSTACK];
                }s;
        NODEPTR  p;

s.top = -1;
p = tree;
do
        {/*  travel down left branches as far as possible, saving
           pointers to nodes passed */
        while(p!=NULL)
                {
                push(s,p);
                p = p → left;
                }
                /*  check if finished */

        if ( !empty (s) )
                {                                / * at this point the left subtree is empty */
                p=pop(s);
                printf("%d \n", p→info);                /* visit the root */
                p =p→right;                             /*  traverse right subtree */
                }
        }while( !empty (s) || p! = NULL );
}
```

Non – Recursive algorithm: Pre**order_Traversal**

```
#define MAXSTACK 100

preorder_traversal (tree)
{
NODEPTR tree;
        {
        struct stack
                {
                int top;
                NODEPTR item[MAXSTACK];
                }s;
        NODEPTR  p;

s.top = -1;
p = tree;
```

```
do
        {
        /*  travel down left branches as far as possible, saving
            pointers to nodes passed */
        if(p!=NULL)
                {
                printf("%d \n", p→info);                        /* visit the root */
                                if(p→right!=NULL)
                        push(s,p→right);                /* push the right subtree
                                                                on to the stack */
                p=p→left;
                }
        else
                p=pop(s);
        }while( ! empty(s) || p! = NULL )
}
```

Binary Search Tree

Binary tree that all elements in the left subtree of a node n are less than the contents of n, and all elements in the right subtree of n are greater than or equal to the contents of n.

Uses : used in sorting and searching

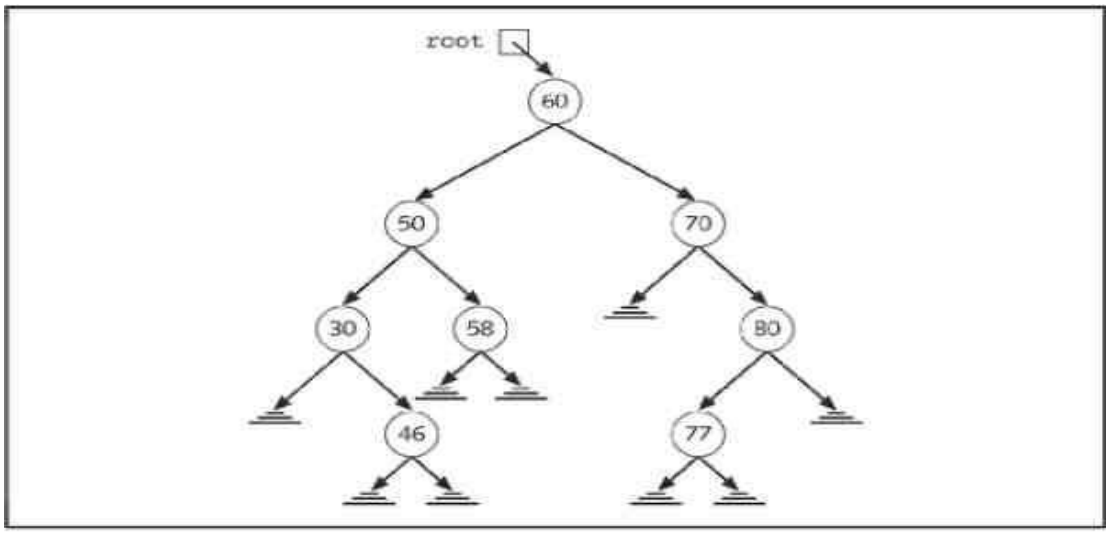


Figure Binary search tree

Applications of Binary Trees (find the duplication element from the list)

Binary tree is useful data structure when two-way decisions must be made at each point in a process. For example find the all duplicates in a list of numbers.

One way doing this is each number compare with it's precede it. However, it involves a large number of comparisons.

The number of comparisons can be reduced by using a binary tree.

Step 1: from root, each successive number in the list is then compared to the number in the root.
Step 2: If it matches, we have a duplicate.
Step 3: If it is smaller, we examine the left subtree.
Step 4: If it is larger, we examine the right subtree.
Step 5: If the subtree is empty, the number is not a duplicate and is placed into a new node at that passion in the tree.
Step 6: If the subtree is nonempty, we compare the number of the contents of root of the subtree and entire process is repeated till all node completed.

```
/*  read the first number and insert it into a sinlge-node binary tree */
        scanf("%d",&number);
        tree = maketree (number);
        while(there are numbers left in the input)
                {
                scanf("%d", &number);
                p = q = tree;
                while ( number ! = info(p) && q ! = NULL)
                        {
                        p = q;
                        if ( number < info (p) )
                                q = left (p);
                        else
                                q = right (p);
                        }
                if(number = = info(p) )
                        printf(" %d %s ", number, "is a duplicate");
                else if ( number < info(p) )
                        setleft( p, number );
                else
                        setright(p, number);
                }
```

(2) Application of Bianry Tree - (Sort the number in Ascending Order)

If a binary search tree is traversed in inorder(left,root,right) and the contents of each node are printed as the node is visited, the numbers are printed in ascending order.

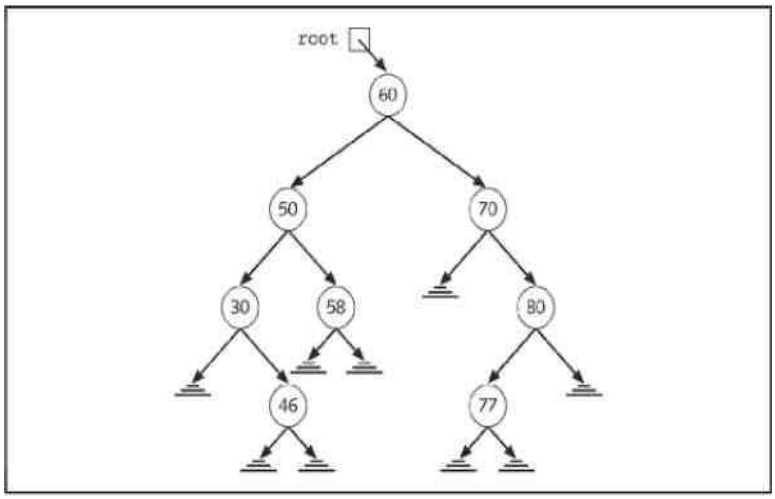


Figure Binary search tree

For convince the above binary search tree if it is traversed inorder manner the result order is,

30, 46, 50, 58, 60, 70, 77 and 80 is ascending order sequence.

(3) Application Binary Tree – (Expression Tree)

Representing an expression containing operands and binary operators by a strictly binary tree. The root of the strictly binary tree contains an operator that is to be applied to the results of evaluating the expressions represented by the left and right subtrees. A node representing an operator is a nonleaf, whereas a node representing an operand in a leaf.

BINARY SEARCH TREE OPERATIONS:

The basic operation on a binary search tree(BST) include, creating a BST, inserting an element into BST, deleting an element from BST, searching an element and printing element of BST in ascending order.

The ADT specification of a BST:

ADT BST

{

Create BST()	:	Create an empty BST;
Insert(elt)	:	Insert elt into the BST;
Search(elt,x)	:	Search for the presence of element elt and Set x=elt, return true if elt is found, else return false.
FindMin()	:	Find minimum element;
FindMax()	:	Find maximum element;
Ordered Output()	:	Output elements of BST in ascending order;
Delete(elt,x)	:	Delete elt and set x = elt;

}

Inserting an element into Binary Search Tree

Algorithm InsertBST(int elt, NODE *T)
[elt is the element to be inserted and T is the pointer to the root of the tree]

```
  If (T= = NULL) then
        Create a one-node tree and return
Else if (elt<key) then
        InsertBST(elt, T→lchild)
Else if(elt>key) then
        InsertBST(elt, T→rchild)
Else
        " element is already exist
return T
End
```

C coding to Insert element into a BST

```
struct node
        {
        int info;
        struct node *lchild;
        struct node *rchild;
        };
typedef stuct node NODE;
NODE *InsertBST(int elt, NODE *T)
        {
        if(T = = NULL)
                {
                T = (NODE *)malloc(sizeof(NODE));
                if ( T = = NULL)
                        printf ( " No memory error");
                else
                        {
                        t→info = elt;
                        t→lchild = NULL;
                        t→rchild = NULL;
                        }
                }
        else if ( elt < T→info)
                t→lchild = InsertBST(elt, t→lchild);
        else if ( elt > T→info)
                t→rchild = InsertBST( elt, t→rchild);
        return T;
        }
```

Searching an element in BST

Searching an element in BST is similar to insertion operation, but they only return the pointer to the node that contains the key value or if element is not, a NULL is return;

Searching start from the root of the tree;
If the search key value is less than that in root, then the search is left subtree;
If the search key value is greater than that in root, then the search is right subtree;
This searching should continue till the node with the search key value or null pointer(end of the branch) is reached.
In case null pointer(null left/righ chile) is reached, it is an indication of the absence of the node.

Algorithm SearchBST(int elt, NODE *T)
[elt is the element to be inserted and T is the pointer to the root of the tree]

```
1. If (T= = NULL) then
        Return NULL
2. If (elt<key) then                        /* elt is less than the key in root */
        return SearchBST(elt, T→lchild)
Else if(elt>key) then                       /* elt is greater than the key in root */
        return SerachBST(elt, T→rchild)
Else
        return T
End
NODE * SearchBST(int elt, NODE *T)
        {
        if(T = = NULL)
                return NULL;

         if ( elt < T→info)
                return SearchBST(elt, t→lchild);
        else if ( elt > T→info)
                return SearchBST( elt, t→rchild);
        else
                return T;
        }
```

Finding Minimum Element in a BST

Minimum element lies as the left most node in the left most branch starting from the root. To reach the node with minimum value, we need to traverse the tree from root along the left branch till we get a node with a null / empty left subtree.

Algorithm FindMin(NODE * T)

1. If Tree is null then
   return NULL;
2. if lchild(Tree) is null then
   return tree
   else
   return FindMin(T→lchild)
3. End

```
NODE * FindMin( NODE *T )
        {
        if(T = = NULL)
                return NULL;

         if ( T→lchild = = NULL)
                return tree;
        else
                return FindMin(Tree→lchild);
        }
```

Finding Maximum Element in a BST

Maximum element lies as the right most node in the right most branch starting from the root. To reach the node with maximum value, we need to traverse the tree from root along the right branch till we get a node with a null / empty right subtree.
Algorithm FindMax(NODE * T)

1. If Tree is null then
   return NULL;
2. if rchild(Tree) is null then
   return tree
   else
   return FindMax(T→rchild)
3. End

```
NODE * FindMax( NODE *T )
        {
        if(T = = NULL)
                return NULL;

         if ( T→rchild = = NULL)
                return tree;
```

```
        else
                return FindMax(Tree→rchild);
        }
```

DELETING AN ELEMENT IN A BST

The node to be deleted can fall into any one of the following categories;

1. Node may not have any children (ie, it is a leaf node)
2. Node may have only on child (either left / right child)
3. Node may have two children (both left and right)

```
Algorithm DeleteBST(int elt, NODE * T)

1. If Tree is null then
        print "Element is not found"
2. If elt is less than info(Tree) then
        locate element in left subtree and delete it
    else if elt is greater than info(Tree) then
        locate element in right subtree and delete it
    else if (both left and right child are not NULL) then        /* node with two children */
        begin
                Locate minimum element in the right subtree
                Replace elt by this value
                Delete min element in right subtree and move the remaining tree as its
                right child
        end
   else
        if leftsubtree is Null then
                /*  has only right subtree or both subtree Null */
                replace node by its rchild
        else
                if right subtree is Null then
                        replace node by its left child\
        end

        free memory allocated to min node
        end
return Tree
End


NODE *DeleteBST(int elt, NODE * T)
{
NODE * minElt;

if(T = = NULL)
        printf(" element not found \n");
else if ( elt < T→info)
        T→lchild = DeleteBST(elt, T→lchild);
else if ( elt > T→info)
```

```
        T→rchild = DeleteBST(elt, T→rchild);
else
        if(T→lchild && T→ rchild)
                {
                minElt = FindMin(T→rchild);
                T→info = minElt→info;
                T→rchild = DeleteBST(T→info, T→rchild);
                }
        else
                {
                minElt = Tree;
                if (T→lchild = = NULL)
                        T = T→rchild;
                else if ( T→rchild = = NULL)
                        T = T→lchild;
                Free (minElt);
                }
return T;
}
```

THREADED BINARY TREE:

Definition: A **threaded** **binary tree** may be defined as follows:
A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node."

In binary trees, left and right child pointers of all leaf nodes(empty subtrees) are set to NULL while traversing the tree. Normally, a stack is maintained to hold the successor of the node in traversal. Replacing such NULL pointers with pointers to successor in traversal could eliminate the use of stack. These pointers are called as threads, and such a tree is called **threaded binary tree.**

If right child pointers nodes with empty right subtree, may be made to point its successor then the tree is called **right in-threaded binary trees**.

If left child pointers nodes with empty left subtree, may be made to point its successor then the tree is called l**eft in-threaded binary trees.**

Note : The last node in the tree not threaded. In example F is at rightmost and not threaded.

Implementing a right in-threaded binary tree:

To implement right in-threaded binary tree we need **info, left , right** and an extra logical field **rthread** is included within each node to indicate whether or not its right pointer is a thread.

lchild info rchild isthread

Thus a node is defined as follows :

```
struct nodetype
        {
        int info;
        struct nodetype * left;        /*  pointer to left son */
        struct nodetype *right;        /*  pointer to right son */
        int rthread;                   /*  rthread is TURE if right is NULL
        }                                  (or)  a non-NULL thread */
        typedef struc nodetype *NODEPTR;
```

We present a routine to implement inorder traversal of a right in-threaded binary tree;

```
intrav2(tree)
NODEPTR tree;
{
NODEPTR q, p;
p = tree;
        do
        {
        q=NULL;
        while(p!=NULL)                          / * traverse left branch */
                {
                q = p;
                p = p → left;
                }
        if(q ! = NULL)
                {
                printf("%d \n",q→info);
                p = q → right;

                while(q→rthread && p!=NULL)
                        {
                        printf("%d \n", p →info);
                        q = p;
                        p = p→right;
                        }
                }
        }while(q! = NULL);
}
intrav3(tree)
NODEPTR tree;
{
NODEPTR q, p;
q = NULL;
p = tree;
        do
        {
```

```
        while(p!=NULL)                                  / * traverse left branch */
                {
                q = p;
                p = p → left;
                }
        if(q ! = NULL)
                {
                printf("%d \n", q→info);
                p = q→right;
                }
        while( q ! = NULL && p = = NULL )
        {

                do
                {
                /*  node(q) has no right son. Back up until a left son or the tree root is
                        encountered */

                p = q;
                q = p→father;
                }while( ! isleft(p) && q ! = NULL);

        if ( q ! = NULL)
                {
                printf("%d \n",q→info);
                p = q→right;
                }
        }/* end while */
    } while (q ! = NULL);
}
```

Application of Graph : (path finding)

Create a graph with the cities as nodes and the roads as arcs. To find a path of length nr from node A to B, look for a node C such that an arc exists from A to C and a path of length nr-1 exists from C to B. If these conditions are satisfied for some node C, the desire path exists, else path doesn't exist. The algorithm uses an auxiliary recursive function findpath(k,a,b). This function return true if there is a path of length k from A to B and false otherewise. The algorithm for the program and function as follow:

```
scanf("%d", &n);                                    /* number of cities */
create n nodes and label them from 0 to n-1;
scanf("%d %d", &a,&b);                              /* seek path from a to b
*/
scanf("%d", &nr);                                   /* desired number of
roads to take */

while( scanf ("%d%d", &city1, &city2) != EOF)
        join(city1,city2);

if(findpath(nr,a,b))
        printf(" a path exists from %d to %d in %d steps", a, b, nr);
else
        printf(" no path exists from %d to %d in %d steps", a, b, nr);
```

The algorithm for the function findpath(k,a,b) follows:

```
        if( k =1 )
                / *  search for a path of length 1  */
                return ( adjacent (a,b) );
        /*  determine if there is a path through C  */

        for( c = 0 ; c < n ; ++c)
                if( adjacent ( a, c ) && findpath ( k-1, c, b) )
                        return ( TRUE );
        return ( FALSE );                       / *  assume no path exists */
```

C Representation of Graphs :

Suppose that the number of nodes in the graph is constant: that is , arcs may be added or deleted but nodes may not. A graph with 50 nodes could then be declared as follows:

```
#define MAXNODES 50
struct node
        {
        /*  information associated with each node */
        }

struct arc
```

```
        {
        int adj;  /*  information associated with each arc */
        };
    struct graph
        {
        struct node nodes[MAXNODES];
        struct arc arcs[MAXNODES]MAXNODES];
        };
        struct graph g;
```

The value of g.arcs[i][j].adj is either TRUE or FALSE depending on whether or not node j is adjacent to node i. The two-dimensional array g.arcs[][].adj is called an **adjacency matrix.**

Primitive Operations :

For existence of arcs is declared simply by,

```
int adj[MAXNODES][MAXNODES];
```

In effect, the graph is totally described by its adjacency matrix.

join(adj, node1, node2)

```
int adj [ ] [MAXNODES];
int node1, node2;
    {
    adj[node1][node2]=TRUE;     /*  add an arc from node1 to node2 */
    }
```

remv(adj, node1, node2)

```
int adj [ ] [MAXNODES];
int node1, node2;
    {
    adj[node1][node2]=FALSE;     /*  delete an arc from node1 to node2 */
    }
```

adjacent(adj, node1, node2)

```
int adj [ ] [MAXNODES];
int node1, node2;
    {
    return((adj[node1][node2]= = TRUE)?  TRUE :  FALSE);
    }
```

A weighted graph with a fixed number of nodes may be declared by

```
    Struct arc
        {
        int adj;
        int weight;
        };
        struct arc g[MAXNODES][MAXNODES];
```

The routine joinwt, which adds an arc from node1 to node2 with a given weight wt, may be coded as follows:

```
jointwt(g, node1, node2, wt)
struct arc g[ ] [MAXNODES];
int node1, node2,wt;
        {
        g[node1][node2].adj = TRUE;
        g[node1][node2].weight = wt;
        }
```

||| the routine remvwt is coded.

Transitive Closure:

Let us assume that a graph is completely described by its adjacency matrix, adj. consider the logical expression adj[i][k] && adj[k][j]. Its value is TRUE if and only if the values of both adj[i][k] and adj[k][j] are TRUE, which implies that there is an arc from node i to node k and an arc from node k to node j. ie, there is path of length 2 from i to j passing through k.

So, that the expression is,

(adj[i][0] && adj[0][j]) | | (adj[i][1] && adj[1][j]) | | . . . . . .

. . . | | (adj[i][MAXNODES-1] && adj[MAXNODES][j])

Consider an array adj2 such that adj2[i][j] is the value of the foregoing expression. adj2 is called the path matrix of length 2.

adj2[i][j] indicates whether or not there is a path of length 2 between i to j. adj2 is said to be the Boolean product of adj with itself.

|||by

Assume that path of length 3 or less exists between two nodes of a graph between i to j. It must be of length 1,2 or 3 is written as follows:

adj[i][j] | | adj2[i][j] || adj3[i][j]

Then, we wish to construct a matrix path of graph has node n such that path[i][j] is TRUE if and only if there is some path from node i to node j (of any length). Clearly,

Path[i][j] = = adj[i][j] | | adj2[i][j] | | . . . . | | adjn[i][j].

There must be another path from i to j of length less than or equal to n. Note that there are only n nodes in the graph, at least one node k must appear in the path twice. The path from i to j can be shortened by removing the cycle from k to k. This process is repeated until no two nodes in the path (except i and j)are equal and therefore the path is of length n or less. Such a path often called the transitive closure of the matrix adj.

We may write a C routine that accepts an adjacency matrix adj and computes its transitive closure path. This routine uses an auxiliary routine prod(a,b,c);

transclose(adj, path)

```
int adj [ ] [MAXNODES], path [ ] [ MAXNODES];
        {
        int i, j , k ;
        int newprod[MAXNODES], path [ ] [MAXNODES],
                        adjprod[MAXNODES][MAXNODES];
        for(i=0 ; i < MAXNODES; ++i)
                for(j=0 ; j < MAXNODES; ++j)
                        adjprod[i][j] = path[i][j] = ad[i][j];

        for(i=0 ; i < MAXNODES; ++i)
                {
                /* i represents the number of times adj has been multiplied by itself to
obtain adjprod.  At this point path represents all paths of length i or less */
                prod(adjprod, adj, newprod);
                for(j=0 ; j < MAXNODES; ++j)
                        for(k=0 ; k < MAXNODES; ++k)
                        path[j][k] = path[j][k] || newprod[j][k];
                for(j=0 ; j < MAXNODES; ++j)
                        for(k=0 ; k < MAXNODES; ++k)
                                adjprod[i][j] = newprod[j][k];
                }
        }
```

The routine prod may be written as follows:

```
        prod( a, b , c)
        int a[ ] [MAXNODES], b[ ] [MAXNODES], c[ ] [MAXNODES];
                {
                int i, j, k, val;
                for(i=0 ; i < MAXNODES; ++i)        /* pass through rows */
                        for(j=0 ; j < MAXNODES; ++j) /* pass through columns */
                        {
                        val = FALSE;
                                for(k=0 ; k < MAXNODES; ++k)
                                val = val || ( a[i][k] && b[k][j]);
                                c[i][j] = val;
                        }
                }
```

: