



2nd Class

2018-2019

Data Structures

هياكل البيانات

أستاذ المادة : م. علاء عبد الحسين

Data structure:

A collection of data elements whose organization is characterized by accessing operation that are used to store and retrieve the individual are elements.

Algorithm:- is a finite set of instructions which, if followed, accomplish a particular task. In our course, it is not enough to specify the form of data structure, but we must give also the algorithm of operation to access these data structures. Any algorithm must satisfy the following criteria.

- 1- Input.
- 2- Output.
- 3- Definiteness.
- 4- Finiteness.
- 5- Effectiveness.

How to choose the suitable data structure

For each set of, data there are different methods to organize these data in a particular data structure. To choose the suitable data structure, we must use the following criteria.

- 1- Data size and the required memory.
- 2- The dynamic nature of the data.
- 3- The required time to obtain any data element from the data structure.
- 4- The programming approach and the algorithm that will be used to manipulate these.

Built – In Data Structures

Programming language usually provide some data structures that are built into the language, for instance, C x C++ provide structures and arrays of various dimensions.

Arrays in C

Array is the first data structure that is built in the C language so it can be considered as a data type in the language. The simplest form of array is a one-dimensional array that may be defined abstractly as a finite ordered set of homogenous elements.

By "finite" we mean that there is a specific number of elements in the array. This number may be large or small, but it must exist.

By "ordered" we mean that the elements of the array are arranged so that there is a zeroth, first, second, third, and so forth.

By "homogeneous" we mean that all the elements in the array must be of the same type. For example an array may contain all integers or all characters but may not contain both.

Necessary to develop a method of ordering its elements in a linear fashion and transforming a two.

Dimensional reference to the linear representation. One method of representing a two-dimensional array in memory is the row-major representation. Under this representation the first row of the array occupies the first set of memory locations reserved for the array, the second row occupies the next set, and so forth.

Let us suppose that a two-dimensional integer array is stored in row-major sequence as in the figure below:

			Base (a) ←
		a {0} {0}	
		a {0} {1}	
		a {0} {2}	
		a {0} {3}	
		a {0} {4}	
Row 0	{		
		a {1} {0}	
		a {1} {1}	
		a {1} {2}	
		a {1} {3}	
		a {1} {4}	
Row 1	{		
		a {2} {0}	
		a {2} {1}	
		a {2} {2}	
		a {2} {3}	
		a {2} {4}	
Row 2	{		

And let we suppose that, for an array are, base (a) is the address of the first element of the array. That is, if ar is declared by

int ar {r1} {r2};

where r1 and r2 are the range of the first and the second dimension, base (ar0 is the address of ar {0} {0}).We also assume that esize is the size of each element in the array. Let us calculate the address of an arbitrary element.

Ar {j1} {j2}

Since the elements is in row j1,its address can be calculated by computing the address the first element of row j1 and adding the quantity j2*esize . but to reach the quantity j2 *esize >But reach the first elements of row j1 (the elemnt cu [j1] [o] .it is necessary to pass thug j1Complew rews, each of which Contains r2 elements ,So the address od the first elements of row j1 is at base (ar) +j1*r2*seize .

Therefore the address of $a[r_1][r_2]$ is at $\text{base}(a) + (r_1 * r_2 + r_2) * \text{esize}$.

Example

If we have array declared as :

$a[3][5]$

So, $r_1=2$ and $r_2=4$ and the address of $a[0][0]$ is $\text{base}(a)$, let us suppose that each element of the array requires a single unit of storage, so $\text{esize} = 1$. Then the location of $a[2][4]$ can be computed by :

$$\text{base}[a] + (r_1 * r_2 + r_2) * \text{esize}$$

$$\text{base}[a] + (2 * 5 + 4) * 1$$

$$\text{base}[a] + 14$$

Multidimensional arrays

C also allows arrays with more than two dimension, for example, a three – dimensional array

`int b[3][2][4];`

Or Six dimensional array:

`int C[7][k][3][5][8][2];`

Implementing One – Dimensional Arrays

A one dimensional array can be implemented easily the C declaration

```
Int b[100];
```

Reserves 100 successive memory locations, each large enough to contain single integer. The address of the first location is called the base address of the array b and is denoted by $\text{base}(b)$. Suppose that the size of each individual element of the array is esize . Then reference to the element $b[0]$ is to the element at location $\text{base}(b)$, reference to $b[1]$ is to the element at $\text{base}(b) + \text{esize}$, reference to $b[2]$ is the element $\text{base}(b) + 2 * \text{esize}$.

In general, inference to $b[j]$ is to the element location $\text{base}(b) + j * \text{esize}$.

In fact, in the C language an array variable is implemented as a pointer variable. The type of the variable b in the above declaration is "Pointer to an integer" or int^* .

An asterisk does not appear in the declaration because the brackets automatically imply that the variable is pointer.

If we have the declaration :-

```
Int * b;
```

Since b is pointer to an integer, $*(b + j)$ is the value of j th integer following the integer location b .

For example :-

The C declaration

```
Int a [100];
```

Specifies an array of 100 integers. The smallest element of an array's index is called its lower bound and in C is always 0, and the highest element is upper bound.

If lower is the lower bound of an array and upper the upper bound , the number of elements in the array , called its range , is given by upper – lower + 1 .

For example , in the array, a, declared previously, lower bound is 0 , the upper bound is 99, and the range is 100.

Using One – Dimensional Arrays

A one dimensional array is used when it is necessary to Keep a large number of items in memory and reference all the items in uniform manner.

Suppose we wish to read 100 integers, find their average , and determine by how much each integer deviates from that average.

```
# define NUMELTS 100

Void main ( )
{
int num [NUMELTS];
int j ;
int total ;
Float avg ;
Float diff ;
Total = 0 ;
For (j= 0 ; j < NUMELTS , j ++ )
{
scanf ("% d ", &num [ j ] ) ;
Total += num [j];
}
Avg=(float) total / NUMELTS ; printf ("Average difference" ) ;
For (j = 0; j < NUMELTS ; j ++ )
{
```

```

diff= num [i ] avg ;
printf ("In % d % £ " ; num [i ] , diff ) ;
}

```

Print f ("In average is : % £" ; avg) ;

$b[i]$, the elements at location $base(b) + i * esize$, is equivalent to the element pointed to by $b + i$, which is $* (b + i)$

Two – Dimension Arrays

An element of two dimensional array is accessed by specifying two indices : a row number and column number .

	Column 0	Col 1	Col 2	Col 3	Col 4
row 0					
row 1					
row 2					

Two-dimensional array

The number of rows or Columns is called the range of the dimension. In the array a, the range of the first dimension is 3 and the Second is 5.

Array Representation in the Memory

Although it is convenient for the programmer to think of the elements of a two – dimensional array as being organized in a two – dimensional table, the hardware of most Computers has no Such Localities. As array must be stored in the memory of a computer and the memory is usually linear. To implement a two dimensional array, it is.

Structures

The second built in Data structure is structure. A structure is a group of variables under one name, in which each variable is identified by its own identifier, each of which is known as a member of structure.

For example, consider the following declaration.

Struct:

```
Chair    first {10};  
Chair    midnit;  
Chair    last {20};  
}Sname, ename;
```

This declaration creates two structures variable, `sname` and `ename`, each of contains three members: `first`, `midnit`, and `last`.

Alternatively, we can assign a tag to the structures and then declare the variables by means of the tag. For example, consider the following declaration that accomplishes the same thing as the declaration just given.

Struct name type:

```
Chair    first {10};  
Chair    midnit;  
Chair    last {20};  
};
```

Struct name type `sname`, `ename`;

This definition creates a structure tag `nametype` containing three members. Once a structure tag has been defined, variables `sname` and `ename` may be declared.

For maximum program clarity, it is recommended that a tag be declared for each structure and variables than be declare using thee tag.

An alternative to using a structure tag is to use the typedef definition. For example:

Typedef struct {

```
    Chair    first {10};  
    Chair    midnit;  
    Chair    last {20};  
};
```

We can then declare:

Sname, ename,

Once a variable has been declared as a structure, each member within that variable may be accessed by specifying the variable name the item's member identifier, thus, the statement.

```
Printf ("%5", sname.first);
```

Can be used to print the first name in the structure sname, and the statement.

```
Ename. Midnit = "m"
```

Can be used to set the middle initial in the structure ename to the letter.

A member of structure may be declared to be another structure.

Struct addrtype {

```
    Chair    straddr {40};  
    Chair    city {10};  
    Chair    stafe {3};  
}
```

We may declare anew structure tag nmadtype:

Atruct nmadtype {

```
struct    nametype name;  
struct    addrtype adress;  
};
```

Implementing structures

Suppose that under a certain C implementation an integer is represented by four bytes, a float number by eight, and an array of ten characters by ten bytes. Then the declaration.

```
struct  M {  
int      X ;  
Gloat    y;  
Char      z{10};  
};
```

Then this structure requires 22 bytes to be implemented in the memory.

Applications of stacks

- 1- Arithmetic Expression Validity Consider a mathematical expression that includes Several Sets of nested parentheses I for example;

$$7 - ((X * ((X + y) / (J - 3)) + y) / (4 - 2.5))$$

We want ensure that the parentheses are nickel correctly ; that is we want to check that :

- 1- There are an equal number of right and left parentheses.
- 2- Every right parenthesis is preceded by a matching left parenthesis.

Expressions such as

((A+B) or A+B(

Violate condition 1 , and expressions such as

)A+B(-C or (A+B)) – (C+D

Violate condition 2

To Solue this problem , think of each left parenthesis as opening a scope and each right parenthesis as Closing a scope .

To change the problem slightly , let us assume that there are three different types of Scope delimiters exist These types are indicated by pureness ((and)), brackets ([aol]), and braces([oal]) . A scope enler must be of the Same type as its scope openerThus , string such as :

(A+B] , ([A + B]) , [A-(b]

Are invalid.

A stack may be used to Keep track of the types of Scope encrusted . Whenever a scope opener is encounkrel , it is pushed onto the stack . when never.

- If the stack is empty , the scope ender dose not have a matching opener and the string is invalid.
- If , however , the stack is nonempty , we pop the stack and check whether the popped item corresponds to the scope ender . If a match occurs , we continue . If it does not , the string is invalid .

figure below , shows the state of the stack after reading in parts of the string :

[illegible]

[
[
(
(

{xt(y-[a+b]*c-[(d+e)])} / (h-(j-(k-[

(

--

{x+ (y-[a+b]*c-[(d+e)])} / (h-(j-(k-[l - {x +(y-[a+b]*c-[(d+e)])} / (h-(j-(k-[l -n

Infix, Postfix , and prefix Expressions

Consider the sum of A and B. we think of applying the operator t" to the operands A and B and write the Sum as A+B . This particular representation is called ifix . There are two alternate notations for expressing the Sam of Aand Busing the symbols A, Band t . These are

T A B prefix

A B + postfix

In prefix notation the operator precedes the two operands , in postfix notation the operator follows the two operands , and in infix notation the operator

Is between the two operands.

The evaluation of the expression $A+B*c$, as written in standard infix notation , require Knowledge of which of the two operations + or * is to be performed first.

In the case of + and * we Know that multiplication is to be done before addition . Thus $A+B*c$ is interpreted as $A+(B*C)$.

For the Same example , if we want need to write the expression as $(A+B)*C$.Therefore , Infix notation may require parentheses to specify a dosinal or of operations.

Using postfix and prefix notation , the notation, the need for parentheses is eliminated because the operator is placed directly after (before) the two operands to which it applied .

To convert infix expression to postfix forms ,we use the following algorithm :

- 1- Completely parenthesize the infix expression .
- 2- Move each operator to the space hold by its corresponding right parenthesis.
- 3- Remove all parentheses.

We can apply this algorithm to the expression :

$$A / B \$ C + D * E - A * C$$

$$(((A / (B \$ C)) + (D * E)) - (A * C))$$

$$ABC\$ / DE*+AC*-$$

The conversion algorithm for infix to prefix specify that , after compactly parenthesizing the infix expression with order of priority , we move each operator to its corresponding left parant .

And after that , delimiting all parentheses

For example :-

$(((A / (B \$ C)) + (D * E)) - (A * C))$

$(((A / (B \$ C)) + (D * E)) - (A * C))$

- + / A\$BC *DE*AC

In the above example , we have consider five binary operations : addition , subtraction , , multiplication division, and exponentiation . The first four are available in C and C⁺⁺ and are dented by the usual operators + , - , * and / > The fifth , expunction is denotaol by \$. The value of the expression A \$ B is a raised to the B power , So that 3\$ 2 is of For these binary operators the following is the order of precedence (highest to lowest) :

Exponentiation

Multiplication / division

Addition / Subtraction

When unparethesize operators of the Same precedence are scanned , the order is assumeal to be left to right except in the case of the exponentiation , where the order is assumed to be trom right to left . Thus A+B+C means (A+B) +C, whereas A \$ B \$C means A \$ (B\$C) .

Evaluating a postfix Expression

Stacks can be used to evaluate the different expression notation in sedition to converting from one expression notation to another as we will see later.

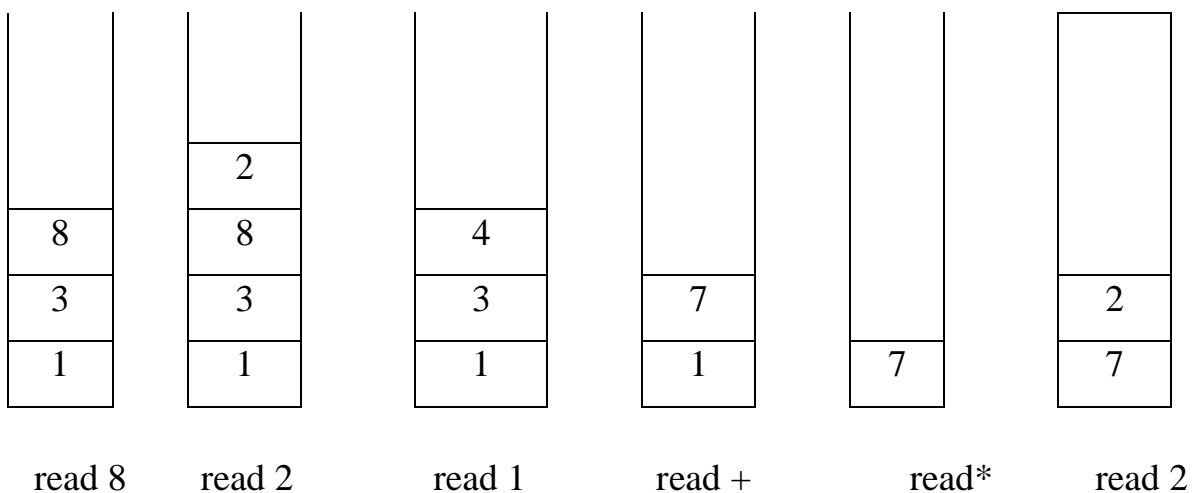
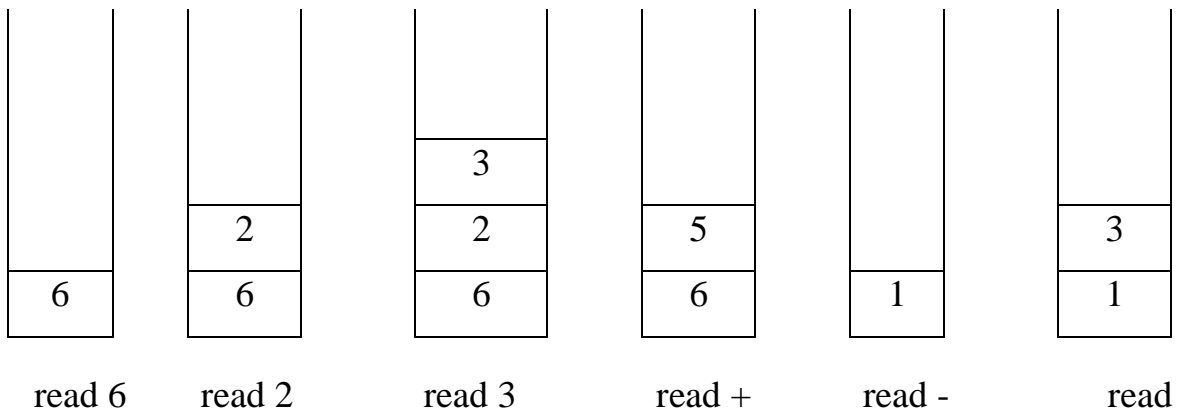
Here, we consider evaluating postfix expression using stack .

As an example , consider the postfix expansion

6 2 3 + - 3 8 2 / + * 2 \$ 3 +

To evaluate such expression, we repeatedly read characters from the postfix expression. If the character read is an operand, push the value associated with it onto the stack. If it is an operator, pop two values from the stack,

Apply the operator to them, and push the result back onto the stack



49

Read \$

3
49

Read3

52

Read+

Queues

A queue is an ordered collection of items from which items may be deleted at one end (collect the front of the queue) and into which items may be inserted at the other end (called the nearer of the queue). The figure illustrates a queue containing there elements A, B and C. A is at the front of the queue and C is at the rear.

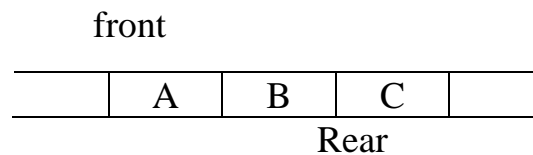


Figure -1-

In figure -2- an element has been deleted from the queue. Since may be deleted only from the front of the queue, A is removed and B is now at the front.

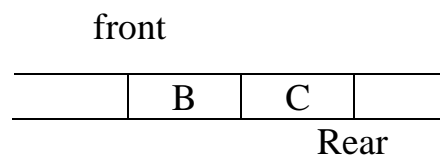


Figure -2-

In figure -3- , when items D and E are inserted , they may be inserted at the rear of the queue.



Figure -3-

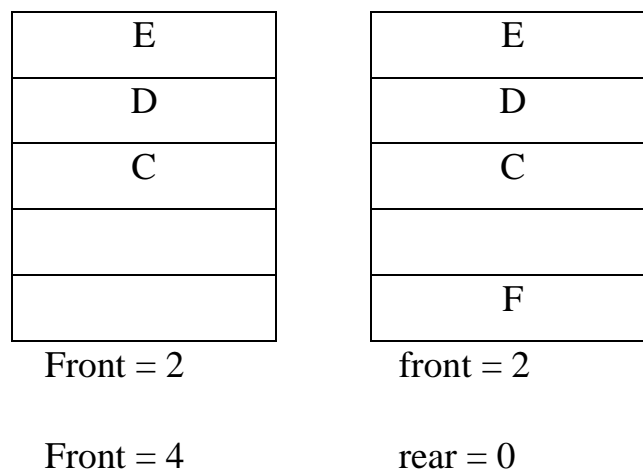
Initially, the queue is empty. Then H, B, and C have been inserted, then tow items have been deleted into two new items D and E have been inserted. Although there are

two spaces in the queue array but, we cannot insert a new element to this queue, because of the overflow case is notified when

$q.\text{rear} == \text{maxqueue} - 1$

A more efficient queue representation is to treat the array of the queue as a circuited rather than a straight line. That means even if the last element of the array is occupied, anew value can be inserted in the first element of the array as long as that element is empty.

Unfortunately, it is difficult under this representation to determine when the queue is empty. The condition $q.\text{rear} < q.\text{front}$ is no longer valid as a test for the empty queue.



One may to solve this problem is to establish the conversation that the value of front is the array index immediately preceding the first element of the queue rather than the index of the first element itself. Since rear is the index of last element of the queue, the condition $q.\text{front} == q.\text{rear}$.

The first element inserted into a queue is the first element to be removed. For this reason a queue is sometimes called a fifo (first – in first – out) list.

Three primitive operations can be applied to a queue. The operation insert (q,x) inserts item x at the rear of the queue q. the operation

X = remove (a) deletes the front element from the queue q and sets x to its contents. The third operation, empty (a), returns false or true depending on whether or not the queue contains any element.

The insert operation can always be performed since there is no limit to the number of elements a queue may contain, the remove operation, however, can be applied only if the queue is nonempty; there is no way to remove an element from a queue containing no element. The result of an illegal attempt is called underflow. The empty operation is, of course, always application.

Implies that the queue is empty. A queue of integers may therefore be declared and initialized by:

```
# define queue {

    int  items [MAXCUEUE];

    int  front , rear;

};

Struct queue q;

q. front = q. rear = MAXCUEUE - 1;
```

this solution leads a new problem that we cannot distinguish between full and empty queue.



In figure (a) removing the last element, leaving the queue empty. In figure (b) adding an element to the last free slot in the queue leaving the queue full. The value of front and rear in the two situation are identical.

One solution is to scarifies one element of the array and to allow a queue to grow only as large as one less than the size of the array.

Implementation of queues

The queue can be represented using an array to hold the elements of the queue and to use two variables, front and rear, to hold the positions within the array of the first and last elements of the queue.

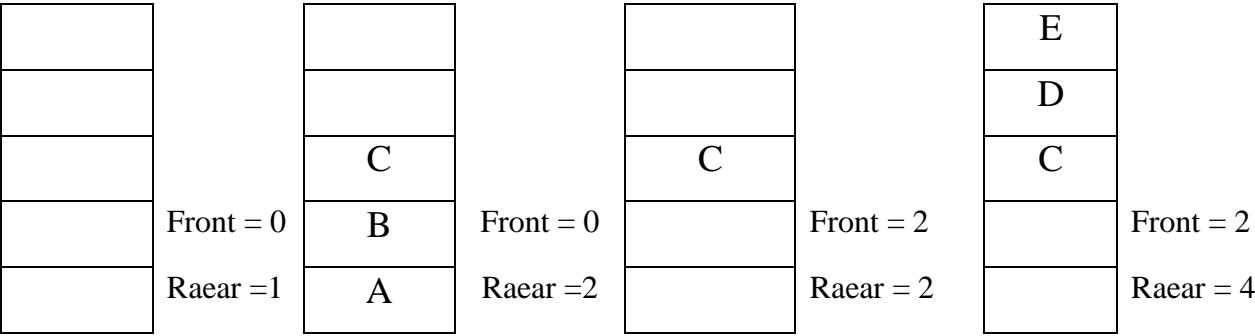
Of course using an array, to hold queue introduce the possibility of overflow, if the queue should grow larger than the size of the array.

```
# define MAXCUEUE 100
```

```
Struct queue {  
    int items { MAXCUEUE}  
    int front, rear;  
    { q;
```

Initially, q. rear is set to -1 , and q. front is set to 0. The queue is empty whenever $q. rear < q. front$, and the number of elements in the queue at any time is equal to the value of $q. rear - q. front + 1$.

Let us examine what might happen under this representation, the following figure illustrates an array of five elements used to represent q queue.

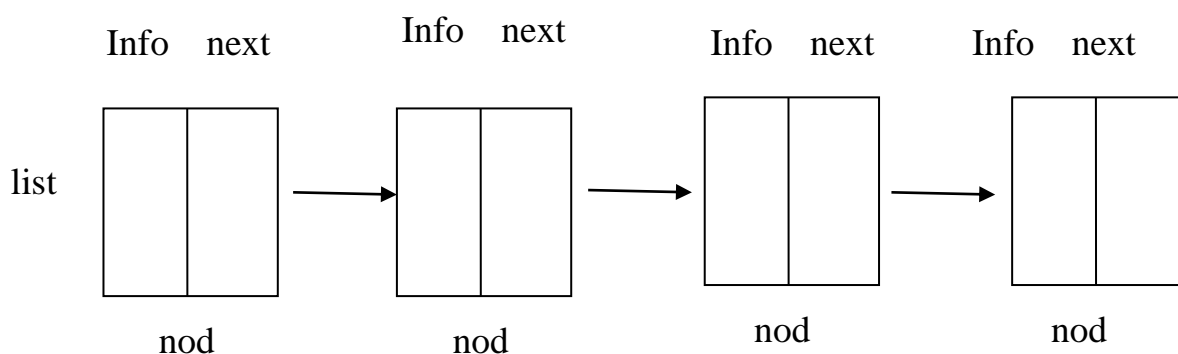


Linked List

The drawback of using sequential storage to represent stacks and queues is that affixed amount of storage remains allocated to the stack or queue even when the structure is actually using a smaller amount or possibly no storage at all. Further, no more than that fixed amount of storage may be allocated, thus introducing the possibility of overflow.

In Sequential representation, the items of a stack or queue are implicitly ordered by Sequential order of storage. Thus if $q.items[x]$ represents an element of a queue the next element will be $q.items[x+1]$.

Suppose that the items of a stack or a queue were explicitly ordered, that is each item contained within itself the address of the next item. Such an explicit ordering gives rise to a data structure, which is known as linked list.



Linear Linked

Each item in the list is called node and contains two fields, an information field and a next address field.

Information field holds an actual element on the list the next field contains the address of the next element in the list. Such an address, which is used to access a particular node is a pointer.

The entire linked list is accessed from an external pointer list that points to the first node in the list the next field of the last node contains a special value, known as

null. The list with no node is known as null list or empty (the external pointer is unitized to null)

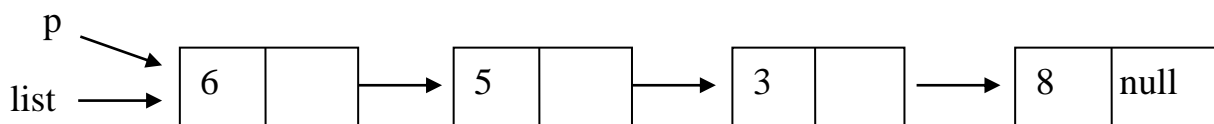
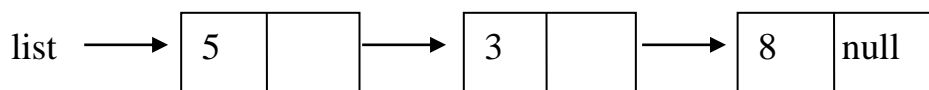
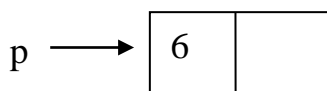
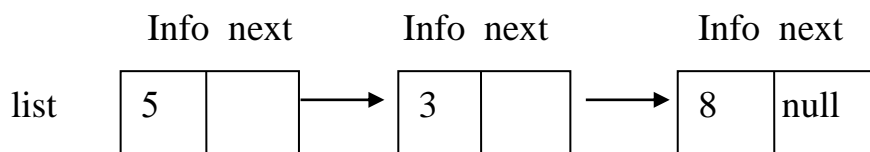
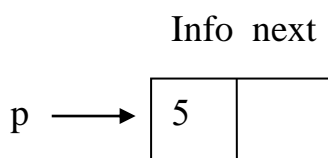
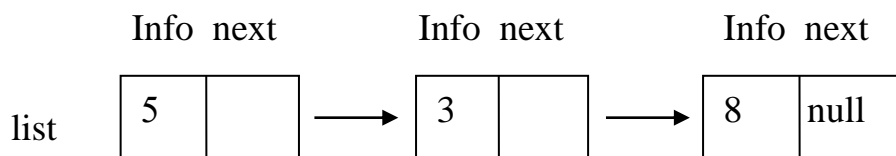
If p is a pointer to a node, node (p) refers to the node pointed to by p,

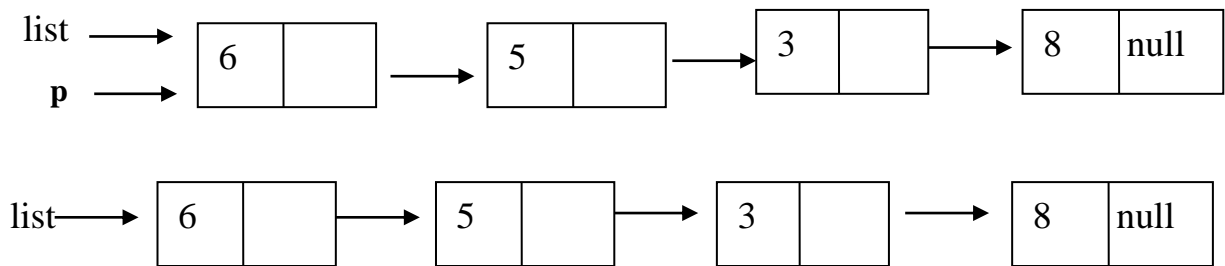
Info (p) refers to the information portion of the node. Next (p) refers to the address of the next node and is therefore a pointer.

Inserting and removing Nodes from a list

A list is a dynamic data structure. The number of nodes on a list may vary as elements are inserted and removed.

Desire to add the integer 6 to the front of that list. The next figure shows the steps of adding a new item to the list.





The first step is to obtain an empty node and set the contents of a variable named `p` to the address of that node . let us assume that the a mechanism to do that is :-

```
P = getnode ( )
```

Next step is to insert the integer 6 into the info portion of the newly allocated node :-

```
Info (p) = 6
```

The next step is to set the next field of the node since the node is to be added to the front of the list, the node that follows should be the current first node on the list :-

```
Next (P ) = list ;
```

Since `list` is the external pointer , its value must be modified to the address of the new first node of the list .

```
List = p ;
```

Putting all steps together , we have an algorithm for adding anew value to the front of a list :-

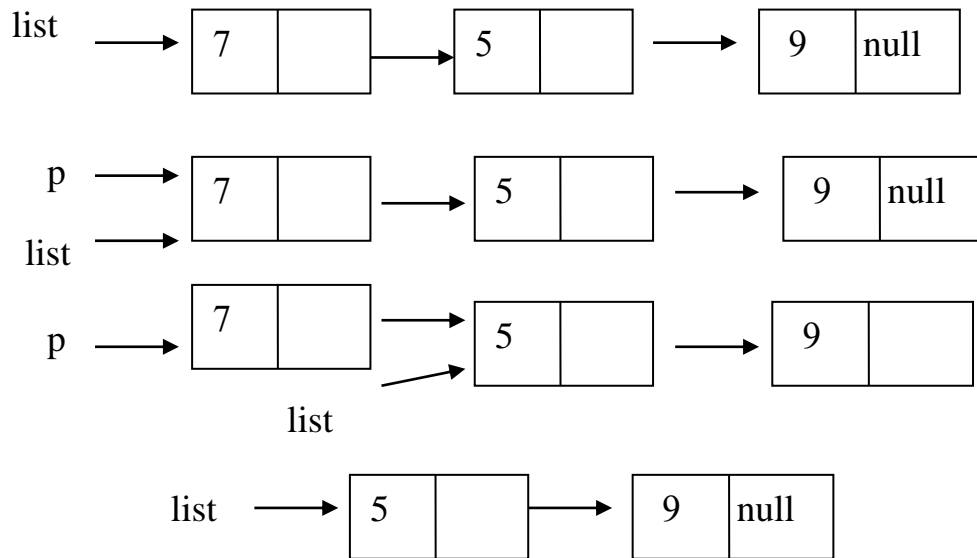
```
P = getnode ( ) :
```

```
Info (p) = 6 ;
```

```
Next (p) =list;
```

```
List = p;
```

The process of removing the hrstnode of anonempty list and storing the value of its info field into a variable `x` is as the next figure :-



To remove a node from the list, the following operations are performed :

`P = list ;`

`List = next (p) ;`

`X= info (p) ;`

`free node (p) ;`

The list step is for making node (p) available for reuse .

Allocating and Freeing Dynamic Variables In C a pointer variable to an integer can be created by the declaration

`int * P;`

Once a variable p has been declared as a pointer to a specific type of object it must be possible to dynamically create an object of that specific type and assign its address to p.

This may be done in C by calling the standard library function malloc (size)

Malloc dynamically allocates apportion of memory of size and returns a pointer to that portion

Consider the declarations

```
Int * pi;
```

```
Float * pr ;
```

The statements

```
Pi= (int * ) malloc (size of (int) ;
```

```
Pr = (float * ) malloc (Size of (float );
```

Dynamically create the integer variable * pi and the float variable *pr. These Variables are called dynamic variables.

The operator Size of returns the size, in bytes , of its operand.

malloc (Size of int) allocates storage for an integer, whereas malloc (Size of (float)) allocates storage for a floating – point number .

The malloc function return type is void *, So we need to Cast the pointer returned by that function to the required type . we should write .

```
pi= (int *) malloc (size of ) (int).
```

As an example of the use of the function malloc :-

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

```
Void main ( )
```

```
{
```

```
Int *p ,*q ;
```

```
Int x ;
```

```
P= (int * ) malloc (size of (int) ;
```

```
8p= 3;
```

```
q = p;
```

```

Cout << * p;
Cout << * q ;
X= 7;

*9=X ;
Cout << * p;
Cout << * q ;
P= (int *) malloc (size of (int ) ;
*p=5;
Cout << * p;
Cout << * q ;
Return ; ?

```

Implantation of Linked list using Dynamic Variables

Recall that a linked list consists of a set of nodes, each of which has two fields: an information field and appointer field to the next node in the list .

In addition , an external points to the first node in the list . the type of a pointer and anode is defined by :

```

Struct node {
int info ;
struct node  * next ;
} ;
Typedef struct node * NODEPTR ;

```

Perution on Linked lists

There are different types of operations can be implanted to deal with the linked list data structure, such as inserting anew item , removing an item , Search about itemetc.

Insert anew item to a linked list

1- Insert First : adding an item to the beginning of a linked list .

```
Void insfrst (NODEPTR *Plist , int x)
```

```
{  
NODEPTR P ;  
P= (NODEPTR ) malloc (size of (NODEPTR ) .  
p→ info = X ;  
p → next = *plist ;  
*plist = p ;  
Return;  
}
```

2- Insert End : adding an item to the end of a linked list :

```
Void insend (NODEPTR * plist , int X)
```

```
{  
NODEPTR p, q ;  
p= (NODEPTR ) malloc (sizeal (NODEPTR ) ;  
p→info =X ;  
p→next = NULL ;  
if (*plist =NULL )  
*plist = p;  
else {  
for (q=*plist ; q→next :NULL ; q=next ) ;  
q→ next = p ;  
}  
}
```

3- Place : adding an item to assorted linked list

```
Void place (NODEPTR *plist , int x)
```

```

{
NODEPTR p, q, w ;

q= NULL ;
for (p= * plist ; p != NULL X > p →info ; p = p want
q= p;
if (q== NULL )
in first (plist ,x);
else
{
W= ( NODEPTR ) malloc (sizeof (NODEPTR )

W→info = X ;
W→next =q →next ;
q→next =W ;
}
return ;
}
Reem of : removes the first node in a linked list removf (NODEPTR *plist )
{
Int X ;
NODEPTR P ;
if ( *plist == NULL )
{
Cout << "list is empty " ;
Exit (1) ;
}
P = plist ;
X= p→info ;

```

```
*plist = p → next ;
```

```
Free (p) ;
```

```
return (X) ;
```

```
}
```

4- Removl : removes the last node in a linked list
int removl (NODEPTR *plist)

```
{
```

```
Int X ;
```

```
NODEPTR p,q ;
```

```
if (*plist == NULL )
```

```
{
```

```
Count << "list is empty ";
```

```
Exit (1) ;
```

```
}
```

```
q= NULL ;
```

```
for (p=plist ; p→next := NULL ; p=p next
```

```
q= p;
```

```
X=P→ info ;
```

```
If (q == NULL )
```

```
*plist = NULL :
```

```
else
```

```
q→ next = NULL ;
```

```
free (p) ;
```

```
return (X) ;
```

```
}
```

-Search : afunction to seach about an itim into Search (NODEPTR *plist , int item)

```
NODEPTR p ;
```

```
For (p=*plist ; p!= NULL ++ I tem != p→info ; p= p →
```


If (p == NULL)

Return (o) ;

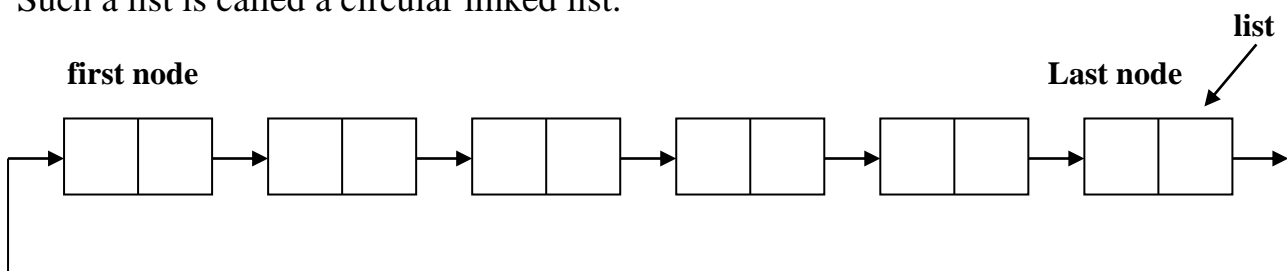
Else

Return (i) ;

Circular linked list

Given a pointer P to a node in a linear linked list, we cannot reach any of the nodes that precede the node that is pointed to by P. If a list is traversed, the external pointer to the list must be traversed to be able to reference the list again.

Suppose that a small change is made to the structure of a linear linked list, so that the next of the last node contains a pointer back to the first node rather than null. Such a list is called a circular linked list.



A circular linked list does not have a natural first or last node. We must, therefore, establish a first and last node by convention. One useful convention is to let the external pointer to the circular list point to the last node.

Implementation of circular linked lists

The type of a pointer and a node is defined in the same way as a linear linked list.

```
struct node {
    int info; struct node *next;
};

typedef struct node *NODEPTR;
```

- Insert first: a function to insert a new node to the beginning of a circular linked list:
 - Void insert (NODEPTR *plist, int x)
- ```
{
```

```

NODEPTR P;
P = (NODEPTR) malloc sizeof (NODEPTR);
P → info = x;
If (*plist == NULL)
*plist = P;
Else
P → next = P;
}

```

Void insert (NODEPTR \*plist , int x)

```

{
NODEPTR P;
P = (NODEPTR) malloc sizeof (NODEPTR);
P → info = x;
If (*plist == NULL)
{
*list = P;
}
Else
{
P → next = (*plist) → next
(*plist) → next = P;
*plist = P;
}
}

```

- Delete – first: a function to delete the first node in a circular linked list.

- Vint delF (NODEPTR \* plist)

```

{
If (*plist == NULL)
{
Cout << "the list is empty";
Exit (1) ; }
P = (*plist) → next;
X = P → info;
If (P == *plist)
*plist = NULL;
Else
(*plist) → next = P → next;
Free (P);
Return (x);
}

```

Delete – last: a function to delete the last n node I a circular linked list

```

Int delet-L (NODEPTR *plist)
{

```

```

NODEPTR P, a;
If (*plist == NULL)
{
 Cout << "the list is empty";
 Exity (1);
 Q = NULL;
 For (P = (*plist) → next; P != *plist; = P → next)
 X = P → info;
 If (q == NULL)
 *plist = NULL
 Else
 {
 q → next = P → next;
 *plist = q;
 }
 Free (P);
 Return (X);
}

```

## Recursive function to linear linked list

- Print: a recursive function to print the elements of linear linked list.

```

Void print (NODEPTR)

```

```

{
 If (P != NULL)
 {
 Cout << P → info;
 Print (P → next);
 }
}

```

Reprint: a recursive function to print the elements of a linear linked list in reverse order.

```

Void Rprint (NODEPTR P)

```

```

{
 If (P != NULL)
 {
 Rprint (P → next);
 }
}

```

```
Cout << P → info;
}
```

**Delete – value: a function to delete a specific value in a linear linked list.**

```
Void delete (NODEPTR *plist, int x)
```

```
{
NODEPTR P,q;
If (*plist == NULL)
{
Cout << "empty list";
Exit (1);
}
q= NULL;
```

```
for (P = *plist; P != NULL X X P → info != X; P = P → next
```

```
q = P;
if (q == NULL)
{ *plist = P → next; free (P);}
Else
If (P == NULL)
{
Cout << "the value is not found";
}
Else
q → next = P → next;
free(P);
}
```

```
}
```

Remix: a function to delete all nodes whom info field contains the value x.

```
Void remix (NODEPTR) *plist, int x)
```

```
{
```

```
NODEPTR P,q;
```

```
q = NULL; P = *plist;
```

```
while (P != NULL)
```

```
{
```

```
If (P → info == x)
```

```
{
```

```
P = P → next;
```

```
If (q == NULL)
```

```
{/ * remove first node of the list */
```

```
Free (*plist)
```

```
*plist = P;
```

```
}
```

```
Else
```

```
{
```

```
q → next = P = q → next;
```

```
}
```

```
Else
```

```
{ q = P;
```

```
P = P → next;
```

```
}
```

```
}
```

```
}
```

The function free is used to free storage of a dynamically allocated variable. The statement:

Free (P);

Makes any future reference to the variable \*P illegal. Calling free(P) makes the storage occupied by \*P a variable for reuse , if necessary.

To illustrate the use of free function, consider the following program:

```
include <stdlib.h>
```

```
Void main ()
```

```
{
```

```
int *p, *q;
```

```
P = (int X) malloc (sizeof (int));
```

```
*P= 5;
```

```
q = (int *) malloc (sizeof (int));
```

```
*q = 8;
```

```
Free (P);
```

```
P = q;
```

```
q = (int *) malloc (sizeof (int));
```

```
*q = 6;
```

```
Cout << *P;
```

```
Cout << *q;
```

```
Return;
```

```
}
```

## Recursion

Recursion is a programming tool that allows the user to write functions that calls themselves.

### Factorial function

Factorial function plays an important role in mathematics and statistics. Given a positive integer,  $n$ , factorial is defined as the product of all integers between  $n$  and 1. For example 5 factorial equals  $5 * 4 * 3 * 2 * 1 = 120$ . In mathematics, the exclamation mark (!) is often used to denote the factorial function. We may write the definition of this function as follows:

$$n! = 1 \text{ if } n = 0$$

$$n! = n * (n-1) * (n-2) * \dots * 1 \text{ if } n > 0$$

as an example for this definition:

$$0! = 1$$

$$1! = 1$$

$$2! = 2 * 1$$

$$3! = 3 * 2 * 1$$

$$4! = 4 * 3 * 2 * 1$$

.....

For any  $n > 0$  we see that  $n!$  equals  $(n-1)!$ . multiplying  $n$  by the product of all integers from  $n-1$  to 1 yields the product of all integers from  $n$  to 1. We may therefore define

$$0! = 1$$



$$1! = 1 * 0!$$

$$2! = 2 * 1!$$

$$3! = 3 * 2 !$$

$$u! = u * 3!$$

....

So we can write our mathematical notation as:

$$n! = 1 \quad \text{if } (n = 0)$$

$$n! = n (n-1) ! \quad \text{if } n > 0$$

this function may appear quite strange, since it once 0! Has been defined, defining 1! As 1 \* 0! Is not circular at all. Similarly, once 1! Has been defined, defining 2! As 2 \* 1! Is straight word.

Therefore , we must evaluate 3 ! Repeating this process , we have that

$$s = s * u!$$

$$u! = u * 3!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

The recursive algorithm to compute n ! may be directly translated into a C function as follows :

```
int fact (int n)
```

```
{
```

```
 int x, y ;
```

$$\}$$

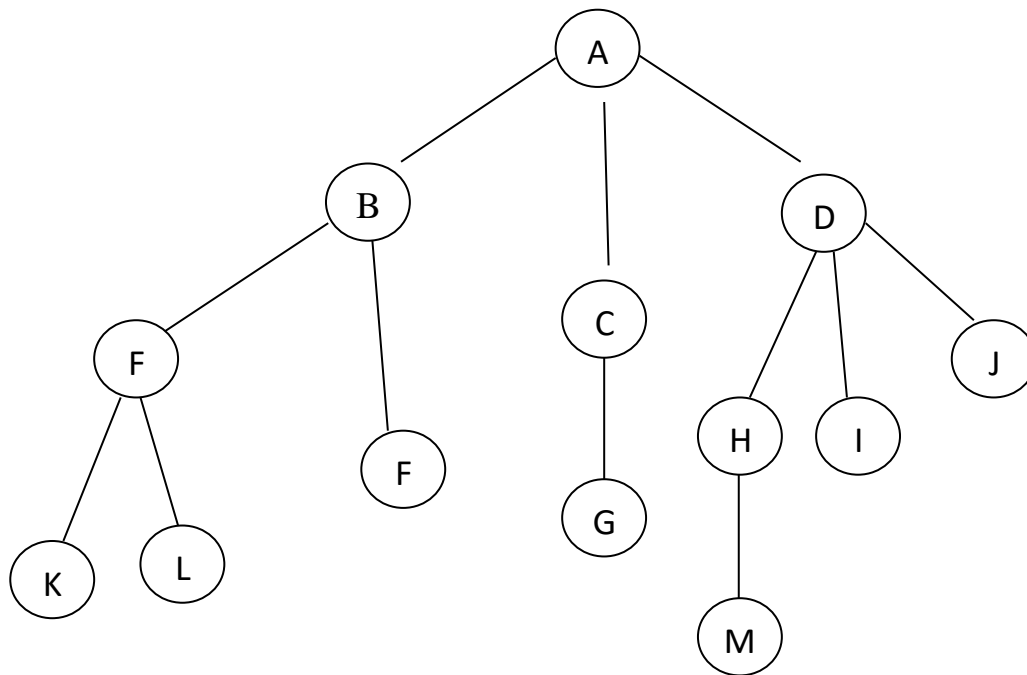
When the function returns, the stack is propel .

Diagram illustrating the state of the array  $y$  at different points in the execution of the `fact` program:

- Initial state:**  $y$  is a 4x3 array. The first column is labeled  $n \times y$  and the second column is labeled  $y = \text{fact}(u)$ . The third column is labeled  $\text{Cut} \ll \text{fact}(u);$ . The values are:  $y[0][0]=1, y[0][1]=0, y[0][2]=1$ ;  $y[1][0]=2, y[1][1]=1, y[1][2]=*$ ;  $y[2][0]=3, y[2][1]=2, y[2][2]=*$ ;  $y[3][0]=4, y[3][1]=3, y[3][2]=*$ .
- After  $y = \text{fact}(1)$ :**  $y[0][0]=2, y[0][1]=1, y[0][2]=1$ ;  $y[1][0]=3, y[1][1]=2, y[1][2]=*$ ;  $y[2][0]=4, y[2][1]=3, y[2][2]=*$ .
- After  $y = \text{fact}(2)$ :**  $y[0][0]=3, y[0][1]=2, y[0][2]=2$ ;  $y[1][0]=4, y[1][1]=3, y[1][2]=*$ .
- After  $y = \text{fact}(3)$ :**  $y[0][0]=4, y[0][1]=3, y[0][2]=6$ .
- After  $\text{Cut} \ll \text{fact}(u);$ :** The array  $y$  is empty.

# Tree

A tree is a data structure that has hierarchical relationships between its individual data items .



- The higher node A of the tree is the root .
- The nodes B, C and D which are directly Connected to the root node are the children of the root .
- The children of a given parent is the set of siblings .
- The link between a parent and its child is called branch .
- The root of the tree is the ancestor of all nodes in the tree.
- Each node may be the parent of any number of nodes in the tree.
- A node with no a subset of a tree which is itself a tree.
- The number of subtrees of a node is called its degree (the degree of A is 3) and the degree of the tree is the maximum degree of the nodes in the tree .
- The root of the tree has level 0 , and the level of any other node in the tree is one more than the level of its father .

- The depth of the tree is the maximum level of any leaf in the tree .

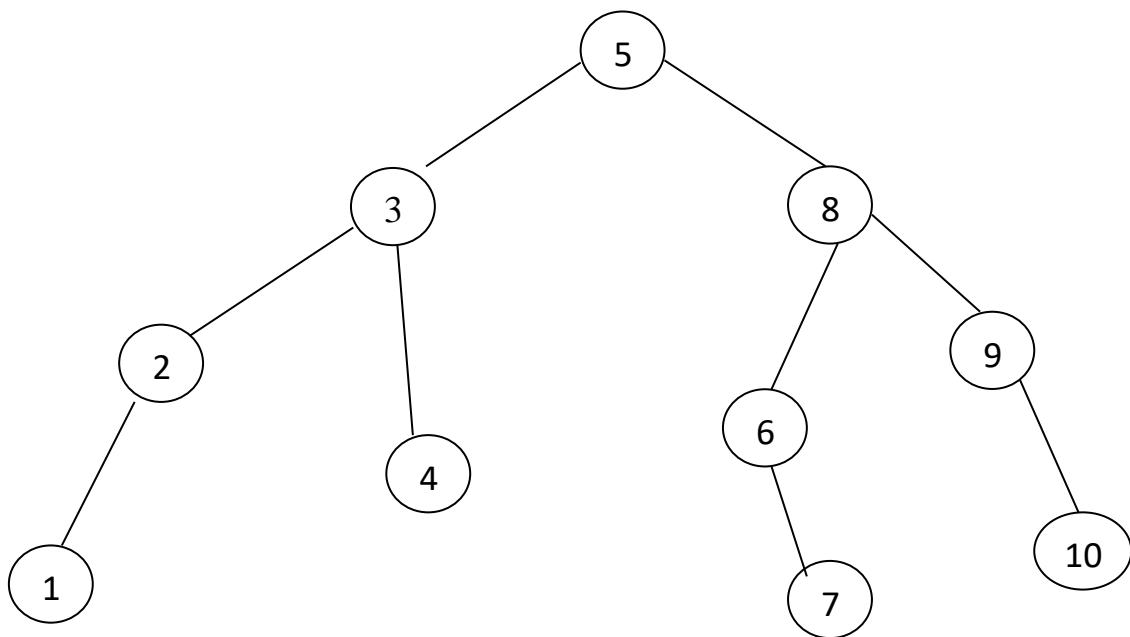
## Binary tree

A binary tree is the tree that is characterized by the fact that any node can have at most two branches (i.e) there is no node with degree greater than two ).

A binary tree may be empty or consist of a root and two disjoint binary trees called the left subtree and the right subtree .

## Binary Search trees

The binary search tree is a binary tree in which the left child , if there is , of any node contains a smaller value than does the parent node and the right child , if there is, contains a larger value than the parent node .

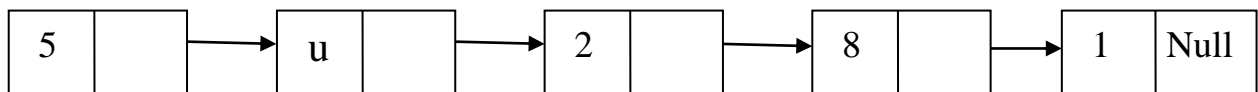


## Advantages of using Binary Search Trees

One of the drawbacks of using linked lists to store ordered informations , is the time it takes to search along the list to find any value .

For example , if we have the following linked list :

list



To find the value L , we must make a sequential traversal of all the nodes in the list .

The binary Search tree provides us with a structure that retains the flexibility of the linked list, while allowing quicker access to any node in the list .

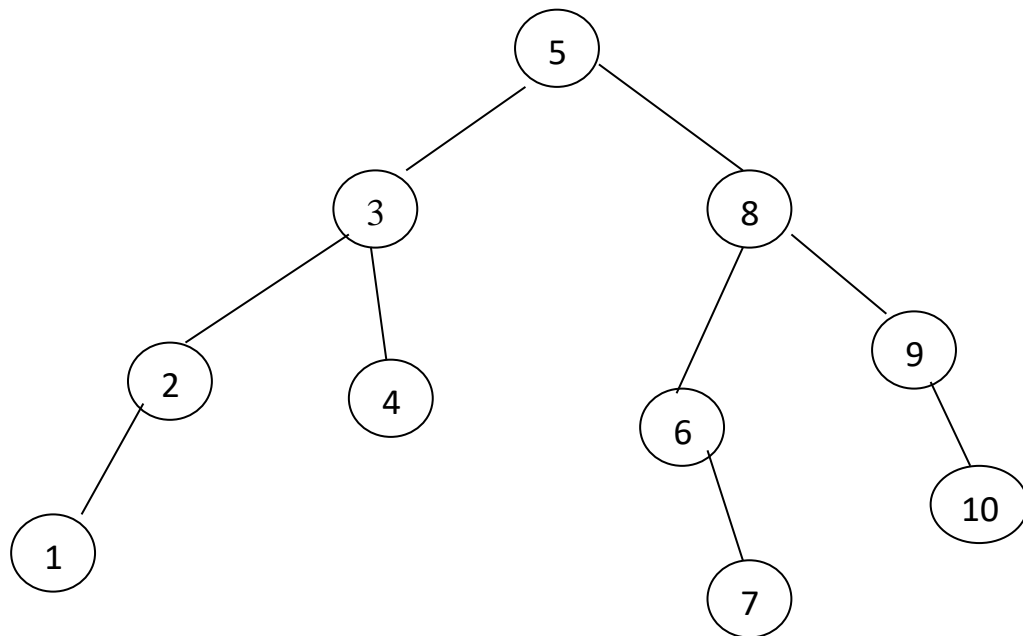
## Implementation of Binary Search tree

There are two methods to implement binary tree Structure :

- 1- Linear representation : which does not require the use of the pointers.
- 2- Linked representation : uses pointers.

## Linear Representation

The linear representation method of a binary Search tree uses a one – dimensional array of size  $(2^{d+1}-1)$  where  $s$  is the depth of the tree. In the following tree ,  $d = 3$  and this tree requires an array of size  $(2^{3+1}-1) = 15$  to be represented .



**Once the size of the array has been determined, the following method is used to represent the tree :**

- 1- Store the root in the 1<sup>st</sup> location of the array .
- 2- If a node is in the 1<sup>st</sup> location of the array, Store its left child at location  $(2n)$ , and its right child at location  $(2n+1)$  .

**The main advantages of this method are :**

- Its simplicity and the fact that, given a child node, its parent node can be determined immediately .

If a child node is at location  $N$  in the array, then its parent node can be determined immediately . If a child is at location  $N$  in the array, then its parent node is at location  $N/2$  .

|   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 5 | 3 | 8 | 2 | 4 | 6 | 9 | 1 | - | -  | -  | -  | 7  | -  | 10 |

### **The disadvantages of this method:**

- Insertion and deletion of a node causes Considerable data monument up and down the array , using an excessive amount of processing time , because insertions and deletions are the major data processing activities .
- Wasted memory location.

### **Linked Representation of a Binary Search Tree**

Because each node in a binary tree may have 2 children , anode in a linked representation has 2pointer fiebs , one for each child , and one or more data fields containing specific information about the node itself . when anode has no children the corresponding pointer fields are NULL.

As in a linear linked list, the first node in the tree is pointed to by an external pointer.

Operations on binary search trees

- Insert: add a new node to a binary search tree.
- Search: search for a node at the binary search tree.
- Delete: delete a node from the binary search tree.
- Print: prints all elements in the binary search tree.

Linked implementation of a binary search tree.

```
Struct node {

 Int key;

 Char *n;

 Struct node * lft;
```

```
Struct node *right;
```

```
}
```

```
Typedef struct node *NODEPTR;
```

-insertion function

```
Void insert (NODEPTR *root, int x, char *name)
```

```
{
```

```
 NODEPTR p,q, nwnode;
```

```
 nwnode = (NODEPTR) malloc (sizeof (NODEPTR));
```

```
 nwnode → key = x;
```

```
 nwnode → n = name;
```

```
 nwnode → left = NULL;
```

```
 nwnode → right = NULL;
```

```
 P = *root; q = NULL;
```

```
 While (P! = NULL)
```

```
 {
```

```
 Q = P;
```

```
 If (P → key > x)
```

```
 P = P → left;
```

```
 Else
```

```
 P = P → right;
```



```

}

If (q == NULL)

Root = newnode;
Else
If (q → key > x)
q → left = nwnode;
else
q → right = newnode;
}

```

#### - Search function

This search function return a pointer to the node that have a specific value.

NODEPTR search (NODEPTR \*root, int keyvalue)

```

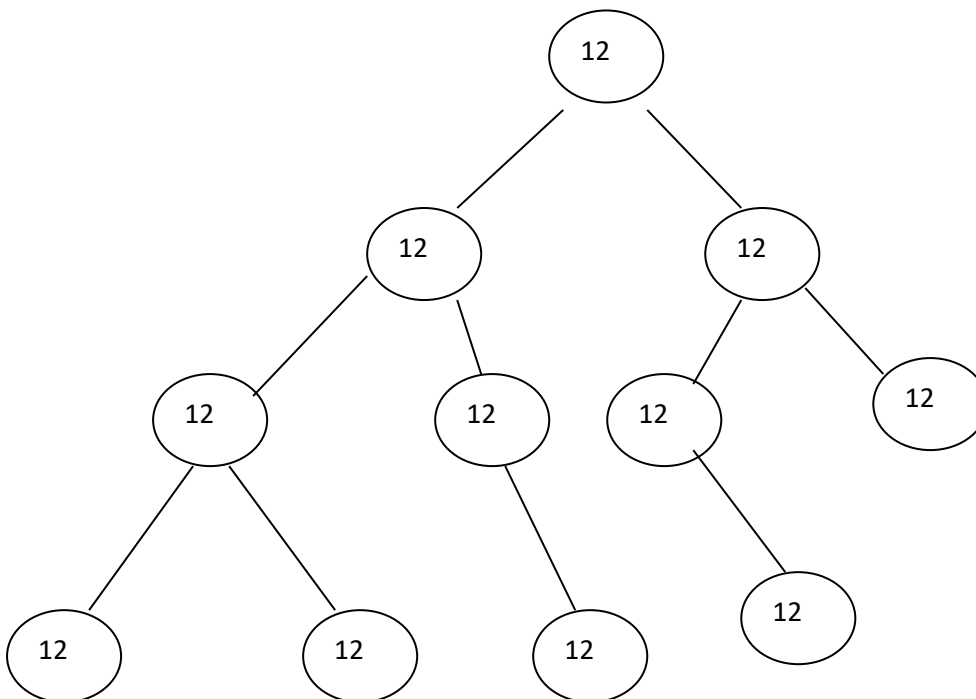
{
NODEPTR P;
Int valueintree = 0;
P = *root;
While (P != NULL && ! b Valueintree)
{
If (P → key = keyvalue)
Valueixtree = 1;
Else
If (P → key < keyvalue)
P = P → left;
Else
P = P → right
}
}

```

$$\}$$

To print the information of nodes within a tree we need to visit each of these nodes and then print its information. There are three different methods to visit the tree nodes, each of them prints nodes information in different order.

These method means traverse and print from the smallest to the largest values. We first need to print the roots left subtree, then we print the value I the root node finally print the value in the root's right subtree.



Void inder (NODEPTR P)

 $\{$

```

If (P → left);

Cout << P → key ;

Inorder (P → right);

}

```

## 2- Preorder print function (Root - left - right)

Preorder method traverse and print each node information before its left and right subtrees. Therefore, the information of the tree in the previous example are print as follows:

Preorder print: 12 7 5 2 6 10 11 25 15 18 45

Void preorder (NODEPTR P)

```

{
If (P != NULL)
{
Cout << P → key;
Preorder (P → left);
Preorder (P → right);
}
}

```

## 3- Postorder print function (left – right – root)

Postorder method travers and print each node information after its left and right subtree. The information of the same example of the previous two methods are print as follows:

Postorder = 26 5 11 10 7 18 15 45 25 12

Void postorder (NODEPTR P)

```

{
If (P != NULL)
{

```

```
Postorder (P \rightarrow left);
Postorder (P \rightarrow right);
Cout << P \rightarrow key;
}
}
```