# Govt. Polytechnic for Women, Sirsa

# E-Contents

## SUBJECT: DATA STRUCTURES USING C

## BRANCH: Computer Engineering

## SEM: 4th



**By:- Hitesh Kumar**

**UNIT-1**

**Fundamental Notations**

## 1.1 Problem solving Concepts: Top Down and Bottom up Design

Programming refers to the method of creating a sequence of instructions to enable the computer to perform a task. It is done by developing logic and then writing instructions in a programming language. A program can be written using various programming practices available. A **programming practice** refers to the way of writing a program and is used along with coding style guidelines. Some of the commonly used programming practices include top-down programming, bottom-up programming and structured programming,

### 1.1.1 Top Down Programming

Top-down programming focuses on the use of modules. It is therefore also known as modular programming. The program is broken up into small modules so that it is easy to trace a particular segment of code in the software program. The modules at the top level are those that perform general tasks and proceed to other modules to perform a particular task. Each module is based on the functionality of its functions and procedures. In this approach, programming begins from the top level of hierarchy and progresses towards the lower levels. The implementation of modules starts with the main module. After the implementation of the main module, the subordinate modules are implemented and the process follows in this way. In top-down programming, there is a risk of implementing data structures as the modules are dependent on each other and they nave to share one or more functions and procedures. In this way, the functions and procedures are globally visible. In addition to modules, the top-down programming uses sequences and the nested levels of commands.

### 1.1.2 Bottom-up Programming

Bottom-up programming refers to the style of programming where an application is constructed with the description of modules. The description begins at the bottom of the hierarchy of modules and progresses through higher levels until it reaches the top. Bottom-up programming is just the opposite of top-down programming. Here, the program modules are more general and reusable than top-down programming.

It is easier to construct functions in bottom-up manner. This is because bottom-up programming requires a way of passing complicated arguments between functions.

### 1.1.3 Structured Programming

Structured programming is concerned with the structures used in a computer program. Generally, structures of computer program comprise decisions, sequences, and loops. The **decision structures** are used for conditional execution of statements (for example, 'if

statement). The **sequence structures** are used for the sequentially executed statements. The **loop structures** are used for performing some repetitive tasks in the program.

Structured programming forces a logical structure in the program to be written in an efficient and understandable manner. The purpose of structured programming is to make the software code easy to modify when required. Structured programming focuses on reducing the following statements from the program.

1. 'GOTO' statements.
2. 'Break' or 'Continue' outside the loops.
3. Multiple exit points to a function, procedure, or subroutine. For example, multiple 'Return' statements should not be used.
4. Multiple entry points to a function, procedure, or a subroutine.

Structured programming generally makes use of top-down design because program structure is divided into separate subsections. A defined function or set of similar functions is kept separately. Due to this separation of functions, they are easily loaded in the memory. In addition, these functions can be reused in one or more programs. Each module is tested individually. After testing, they are integrated with other modules to achieve an overall program structure. Note that a key characteristic of a structured statement is the presence of single entry and single exit point. This characteristic implies that during execution, a structured statement starts from one defined point and terminates at another defined point.

## 1.2 Data Type, Variable and Constants

### 1.2.1 Data and Data Item

Data are simply collection of facts and figures. Data are values or set of values. A data item refers to a single unit of values. Data items that are divided into sub items are group items;those that are not are called elementary items. For example, a student's name may be divided into three sub items – [first name, middle name and last name] but the ID of a student would normally be treated as a single item. In the above example ( ID, Age, Gender, First, Middle, Last, Street, Area ) are elementary data items, whereas (Name, Address ) are group data items.

### 1.2.2 Data Type

Data type is a classification identifying one of various types of data, such as floating-point, integer, or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; and the way values of that type can be stored. It is of two types: Primitive and non-primitive data type. Primitive data type is the basic data type that is provided by the programming language with built-in support. This data type is native to the language and is supported by machine directly while non-primitive data type is derived from primitive data type. For example- array, structure etc.

### 1.2.3 Variable

It is a symbolic name given to some known or unknown quantity or information, for the purposeof allowing the name to be used independently of the information it represents. A variable name in computer source code is usually associated with a data storage location and thus also its contents and these may change during the course of program execution.

### 1.2.4 Record

Collection of related data items is known as record. The elements of records are usually called fields or members. Records are distinguished from arrays by the fact that their number of fields is typically fixed, each field has a name, and that each field may have a different type.

### 1.2.5 Constant

A constant is a value or an identifier whose value cannot be altered in a program. For example: 1, 2.5, "C programming is easy", etc.
As mentioned, an identifier also can be defined as a constant.

```
const double PI = 3.14
```

Here, PI is a constant. Basically what it means is that, PI and 3.14 is same for this program.

Below are the different types of constants you can use in C.

**1. Integer constants**

An integer constant is a numeric constant (associated with number) without any fractional or exponential part. There are three types of integer constants in C programming:

- decimal constant(base 10)
- octal constant(base 8)
- hexadecimal constant(base 16)

For example:

```
Decimal constants: 0, -9, 22 etc

Octal constants: 021, 077, 033 etc

Hexadecimal constants: 0x7f, 0x2a, 0x521 etc
```

In C programming, octal constant starts with a 0 and hexadecimal constant starts with a 0x.

**2. Floating-point constants**

A floating point constant is a numeric constant that has either a fractional form or an exponent form. For example:

```
-2.0
```

```
0.0000234

-0.22E-5
```

**Note:** E-5 = $10^{-5}$

**3. Character constants**

A character constant is a constant which uses single quotation around characters. For example: 'a', 'l', 'm', 'F'

**4. Escape Sequences**

Sometimes, it is necessary to use characters which cannot be typed or has special meaning in C programming. For example: newline (enter), tab, question mark etc. In order to use these characters, escape sequence is used.

## 1.3 Pointers

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

There are a few important operations, like **(a)** define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations −

```
#include<stdio.h>

int main (){

intvar=20;/* actual variable declaration */

int*ip;/* pointer variable declaration */

ip=&var;/* store address of var in pointer variable*/

printf("Address of var variable: %x\n",&var);

/* address stored in pointer variable */

printf("Address stored in ip variable: %x\n",ip);

/* access the value using the pointer */

printf("Value of *ip variable: %d\n",*ip);

return0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

## 1.4 DATA STRUCTURE

In computer science, a data structure is a particular way of storing and organizing data in a computer's memory so that it can be used efficiently. Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a data structure. The choice of a particular data model depends on the two considerations first; it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data whenever necessary.

### 1.4.1 Need of data structure

- It gives different level of organization data.
- It tells how data can be stored and accessed in its elementary level.

- Provide operation on group of data, such as adding an item, looking up highest priority item.
- Provide a means to manage huge amount of data efficiently.
- Provide fast searching and sorting of data.

### 1.4.2 Selecting a data structure

Selection of suitable data structure involve following steps –

- Analyze the problem to determine the resource constraints a solution must meet.
- Determine basic operation that must be supported. Quantify resource constraint for each operation
- Select the data structure that best meets these requirements.

Each data structure has cost and benefits. Rarely is one data structure better than other in all situations. A data structure require :

- Space for each item it stores
- Time to perform each basic operation
- Programming effort.

Each problem has constraints on available time and space. Best data structure for the task requires careful analysis of problem characteristics.

### 1.4.3 Type of data structure

#### 1.4.3.1 Static data structure

A data structure whose organizational characteristics are invariant throughout its lifetime. Such structures are well supported by high-level languages and familiar examples are arrays and records. The prime features of static structures are

(a) None of the structural information need be stored explicitly within the elements – it is often held in a distinct logical/physical header;

(b) The elements of an allocated structure are physically contiguous, held in a single segment of memory;

(c) All descriptive information, other than the physical location of the allocated structure, is determined by the structure definition;

(d) Relationships between elements do not change during the lifetime of the structure.

#### 1.4.3.2 Dynamic data structure

A data structure whose organizational characteristics may change during its lifetime. The adaptability afforded by such structures, e.g. linked lists, is often at the expense of decreased efficiency in accessing elements of the structure. Two main features distinguish dynamic structures from static data structures. Firstly, it is no longer possible to infer all structural information from a header; each data element will have to contain information relating it logically to other elements of the structure. Secondly, using a single block of contiguous storage is often not appropriate, and hence it is necessary to provide some storage management scheme at run-time.

#### 1.4.3.3 Linear Data Structure

A data structure is said to be linear if its elements form any sequence. There are basically two ways of representing such linear structure in memory.

**a)** One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays**.**
**b)** The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists**.**
The common examples of linear data structure are arrays, queues, stacks and linked lists.

#### 1.4.3.4 Non-linear Data Structure

This structure is mainly used to represent data containing a hierarchical relationship between elements. E.g. graphs, family trees and table of contents.

## 1.5 BRIEF DESCRIPTION OF DATA STRUCTURES

### 1.5.1 Array

The simplest type of data structure is a linear (or one dimensional) array. A list of a finite number *n* of similar data referenced respectively by a set of *n* consecutive numbers, usually 1, 2,3 . . . . . *n*. if we choose the name **A** for the array, then the elements of **A** are denoted by subscript notation **A** 1, **A** 2, **A** 3 . . . . **A** nor by the parenthesis notation
A (1), A (2), A (3) . . . . . A (n)
or by the bracket notation
A [1], A [2], A [3] . . . . . A [n]

**Example:**

A linear array **A[8]** consisting of numbers is pictured in following figure.

### 1.5.2 Linked List

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. Each node is divided into two parts:

- The first part contains the information of the element/node
- The second part contains the address of the next node (link /next pointer field) in the list.

There is a special pointer Start/List contains the address of first node in the list. If this special pointer contains null, means that List is empty.

**Example:**

### 1.5.3 Tree

Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a rooted tree graph or, simply, a tree.

### 1.5.4 Graph

Data sometimes contains a relationship between pairs of elements which is not necessarily hierarchical in nature, e.g. an airline flights only between the cities connected by lines. This data structure is called Graph.

### 1.5.5 Queue

A queue, also called FIFO system, is a linear list in which deletions can take place only at one end of the list, the Font of the list and insertion can take place only at the other end Rear.

### 1.5.6 Stack

It is an ordered group of homogeneous items of elements. Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack). The last element to be added is the first to be removed (LIFO: Last In, First Out).

## 1.6 DATA STRUCTURES OPERATIONS

The data appearing in our data structures are processed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely in the frequency with which specific operations are performed.

The following four operations play a major role in this text:

**Traversing:** accessing each record/node exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called "visiting" the record.)

**Searching:** Finding the location of the desired node with a given key value, or finding the locations of all such nodes which satisfy one or more conditions.

**Inserting:** Adding a new node/record to the structure.

**Deleting:** Removing a node/record from the structure.

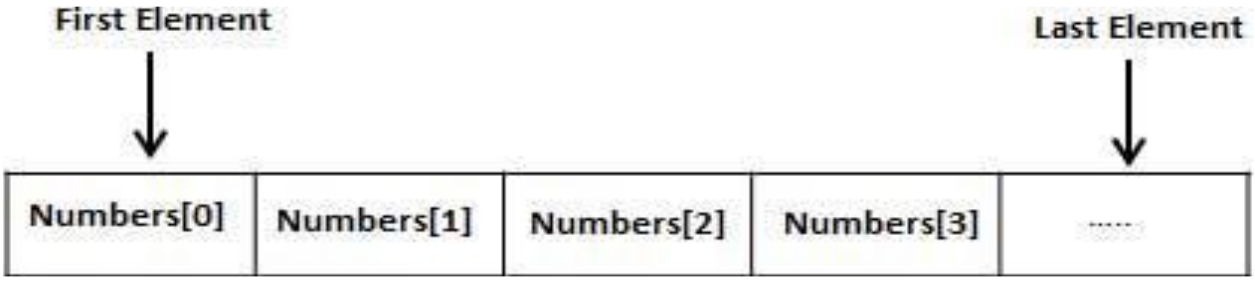
**Sorting:** Arranging elements in either ascending or descending order.

# UNIT-2

## Arrays

### 2.1 Concept of Arrays

Array is a data structure which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Instead of declaring individual variables, such as number0, number1, ..., and number99, we declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.
All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.




The array may be categorized as :

- One dimensional array
- Two dimensional array
- Multidimensional array

### 2.1.1 One Dimensional Array

A one-dimensional array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.

As an example consider the C declaration

`intanArrayName[10];`

Syntax :datatype Arrayname [sizeofArray];

In the given example the array can contain 10 elements of any value available to the `int` type. In C, the array element indices are 0-9 in this case. For example, the expressions `anArrayName[0]` and `anArrayName[9]` are the first and last elements respectively.

One Dimensional Arrays can be initialized as follows:

**Examples –**

```
// A character array in C
char arr1[] = {'g', 'e', 'e', 'k', 's'};


// An Integer array in C
```

```
int arr2[] = {10, 20, 30, 40, 50};


// Item at i'th index in array is typically accessed
// as "arr[i]".  For example arr1[0] gives us 'g'
// and arr2[3] gives us 40.
```

As we know now 1-d array are linear array in which elements are stored in the successive memory locations. The element at which the first element is stored in memory is called its **base address**. Now consider the following example :

int arr[5];

| Element | 34 | 78 | 98 | 45 | 56 |
|---|---|---|---|---|---|
| Memory Address | arr[0] = 100 | arr[1] = ? | arr[2] = ? | arr[3] = ? | arr[4] = ? |

Here we have defined an array of five elements of integer type whose first element is at base address 100. i.e, the element arr[0] is stored at base address 100. Now for calculating the starting address of the next element i.e. of a[1], we can use the following formula :

**Base Address (B)+ No. of bytes occupied by element (C) * index of the element (i)**

/* Here C is constant integer and vary according to the data type of the array, for e.g. for integer the value of C will be 2 bytes, since an integer occupies 2 bytes of memory. */

Now, we can calculate the starting address of second element of the array as :

arr[1] = 100 + 2 * 1 = 102/*Thus starting address of second element of array is 102 */

Similarly other addresses can be calculated in the same manner as :

arr[2] = 100 + 2 * 2 = 104

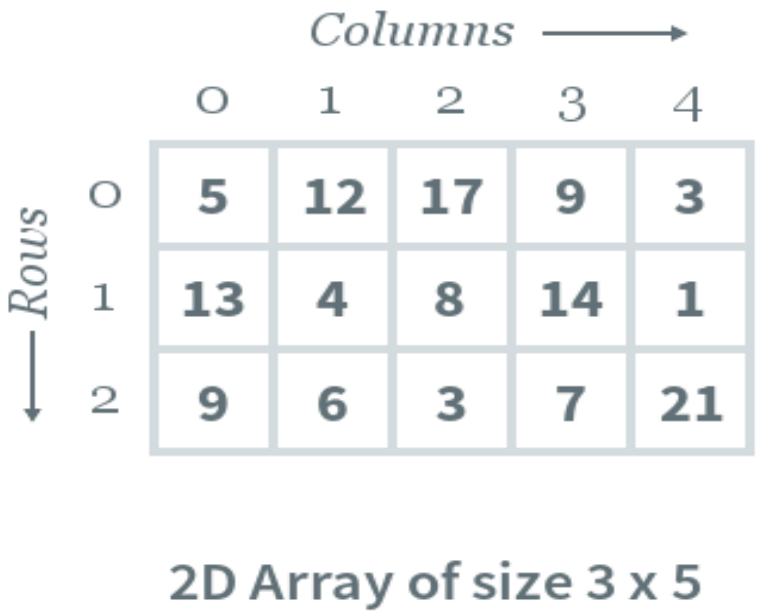
arr[3] = 100 + 2 * 3 = 106

arr[4] = 100 + 2 * 4 = 108

### 2.1.2 Two Dimensional Array

An array of one dimensional arrays is known as 2-D array. The two dimensional (2D) array in C programming is also known as matrix. A matrix can be represented as a table of rows and columns.

Consider following 2-D array, which is of the size 3×5. For an array of size N×M, the rows and columns are numbered from 0 to N−1 and columns are numbered from 0 to M−1, respectively. Any element of the array can be accessed by arr[i][j] where $0 \le i < N$ and $0 \le j < M$. For example, in the following array, the value stored at arr[1][3] is 14.




2D Array of size 3 x 5

**2D array declaration:**

To declare a 2D array, you must specify the following:
**Row-size:** Defines the number of rows
**Column-size:** Defines the number of columns
**Type of array:** Defines the type of elements to be stored in the array i.e. either a number, character, or other such datatype. A sample form of declaration is as follows:

```
typearr[row_size][column_size]
```

A sample C array is declared as follows:

```
intarr[3][5];
```

**2D array initialization:**

An array can either be initialized during or after declaration. The format of initializing an array during declaration is as follows:

```
typearr[row_size][column_size]={{elements},{elements}...}
```

An example is given below:

```
intarr[3][5]={{5,12,17,9,3},{13,4,8,14,1},{9,6,3,7,21}};
```

Initializing an array after declaration can be done by assigning values to each cell of 2D array, as follows.

```
typearr[row_size][column_size]
arr[i][j]=14
```

An example of initializing an array after declaration by assigning values to each cell of a 2D array is as follows:

```
intarr[3][5];
arr[0][0]=5;
arr[1][3]=14;
```

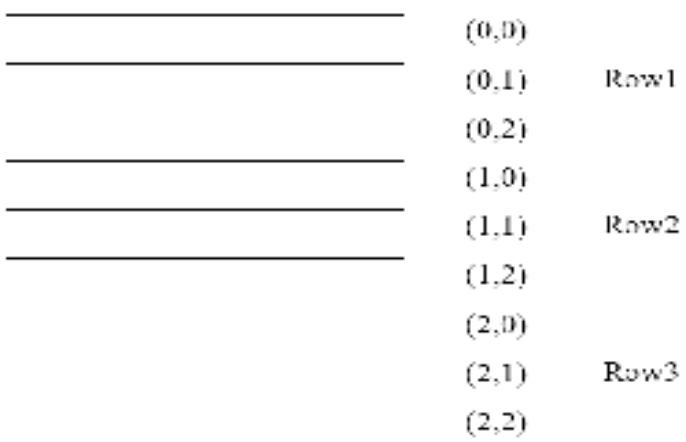
**Processing 2D arrays:**

The most basic form of processing is to loop over the array and print all its elements, which can be done as follows:

```
typearr[row_size][column_size]={{elements},{elements}...}
for i from0 to row_size
for j from0 to column_size
printarr[i][j]
```

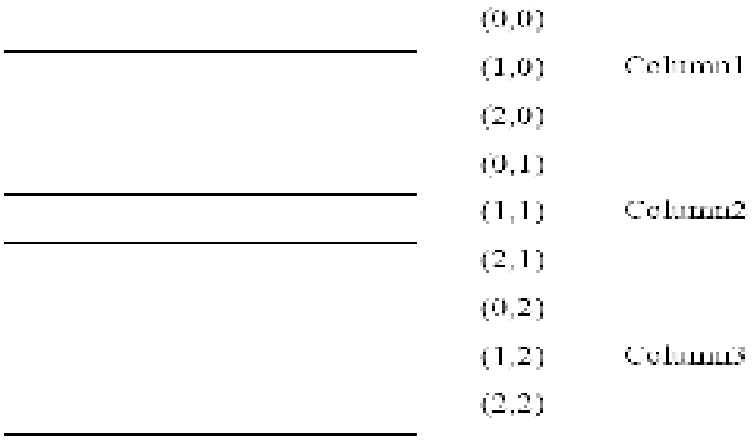
**Representation of two dimensional arrays in memory**

A two dimensional 'm x n' Array A is the collection of m X n elements. Programming language stores the two dimensional array in one dimensional memory in either of two ways-
**Row Major Order:** First row of the array occupies the first set of memory locations reserved for the array; Second row occupies the next set, and so forth.




To determine element address A[i,j]:
Location ( A[ i,j ] ) =Base Address + ( N x ( I - 1 ) ) + ( j - 1 )
For example:
Given an array [1…5,1…7] of integers. Calculate address of element T[4,6], where BA=900.
Sol) I = 4 , J = 6
M= 5 , N= 7
Location (T [4,6]) = BA + (7 x (4-1)) + (6-1)
= 900+ (7 x 3) +5
= 900+ 21+5
= 926

**Column Major Order:**Order elements of first column stored linearly and then comes elements of next column




To determine element address A[i,j]:
Location ( A[ i,j ] ) =Base Address + ( M x ( j - 1 ) ) + ( i - 1 )
For example:
Given an array [1…6,1…8] of integers. Calculate address element T[5,7], where BA=300
Sol) I = 5 , J = 7
M= 6 , N= 8

Location (T [4,6]) = BA + (6 x (7-1)) + (5-1)
= 300+ (6 x 6) +4

= 300+ 36+4

= 340

**2.2 Operations on array**

**a) Traversing:** means to visit all the elements of the array in an operation is called traversing.

**b) Insertion:** means to put values into an array

**c) Deletion / Remove:** to delete a value from an array.

**d) Sorting:** Re-arrangement of values in an array in a specific order (Ascending or Descending) is called sorting.

**e) Searching:** The process of finding the location of a particular element in an array is called searching.

**a) Traversing in Linear Array:**

It means processing or visiting each element in the array exactly once; Let **'A'** is an array stored in the computer's memory. If we want to display the contents of ***'A'***, it has to be traversed i.e. by accessing and processing each element of ***'A'*** exactly once.

**Algorithm:** (Traverse a Linear Array) Here **LA** is a Linear array with lower boundary **LB** and upper boundary **UB**. This algorithm traverses **LA** applying an operation Process to each element of **LA**.

1. [Initialize counter.] Set K=LB.
2. Repeat Steps 3 and 4 while K≤UB.
3. [Visit element.] Apply PROCESS to LA[K].
4. [Increase counter.] Set k=K+1.

   [End of Step 2 loop.]
5. Exit.

The alternate algorithm for traversing (using for loop) is :

**Algorithm:** (Traverse a Linear Array) This algorithm traverse a linear array **LA** with lower bound **LB** and upper bound **UB**.

1. Repeat for K=LB to UB
   Apply PROCESS to LA[K].
   [End of loop].
2. Exit.

**This program will traverse each element of the array to calculate the sum and then calculate& print the average of the following array of integers.**
( 4, 3, 7, -1, 7, 2, 0, 4, 2, 13)

**b)Insertion:**

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Algorithm

Let **LA** be a Linear Array (unordered) with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the $K^{th}$ position of LA −

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include<stdio.h>


main(){

int LA[]={1,3,5,7,8};

int item =10, k =3, n =5;

int i =0, j = n;

printf("The original array elements are :\n");

for(i =0; i<n; i++){

printf("LA[%d] = %d \n", i, LA[i]);

}

   n = n +1;

while( j >= k){

LA[j+1]= LA[j];

      j = j -1;

}
```

```
  LA[k]= item;
        printf("The array elements after insertion :\n");
for(i =0; i<n; i++){
printf("LA[%d] = %d \n", i, LA[i]);
}
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

**c) Deletion Operation**

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the $K^{th}$ position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include<stdio.h>
main(){
int LA[]={1,3,5,7,8};
int k =3, n =5;
int i, j;
printf("The original array elements are :\n");
for(i =0; i<n; i++){
printf("LA[%d] = %d \n", i, LA[i]);
}
   j = k;
while( j < n){
LA[j-1]= LA[j];
      j = j +1;
}
   n = n -1;
printf("The array elements after deletion :\n");
for(i =0; i<n; i++){
printf("LA[%d] = %d \n", i, LA[i]);
}
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
```

```
LA[2] = 7
LA[3] = 8
```

**d) Search Operation**

You can perform a search for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include<stdio.h>

main(){

int LA[]={1,3,5,7,8};

int item =5, n =5;

int i =0, j =0;

printf("The original array elements are :\n");

for(i =0; i<n; i++){

printf("LA[%d] = %d \n", i, LA[i]);

}

while( j < n){

if( LA[j]== item ){

break;

}

   j = j +1;

}
```

```
printf("Found element %d at position %d\n", item, j+1);

}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3
```

**e) Update Operation**

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to update an element available at the $K^{th}$ position of LA.

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include<stdio.h>

main(){

int LA[]={1,3,5,7,8};

int k =3, n =5, item =10;

int i, j;

printf("The original array elements are :\n");

for(i =0; i<n; i++){

printf("LA[%d] = %d \n", i, LA[i]);

}

LA[k-1]= item;

printf("The array elements after updation :\n");
```

```
for(i =0; i<n; i++){

printf("LA[%d] = %d \n", i, LA[i]);

}

}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

**f) Sorting:**

**Sorting** an array is the ordering the array elements in ***ascending*** (increasing from min to max) or ***descending***(decreasing from max to min) order.

**Bubble Sort:**
The technique we use is called *"Bubble Sort"* because the bigger value gradually bubbles theirway up to the top of array like air bubble rising in water, while the small values sink to thebottom of array. This technique is to make several passes through the array. On each pass,successive pairs of elements are compared. If a pair is in increasing order (or the values areidentical), we leave the values as they are. If a pair is in decreasing order, their values areswapped in the array.

| Pass = 1 | Pass = 2 | Pass = 3 | Pass=4 |
|---|---|---|---|
| **2 1** 5 7 4 3 | **1 2** 5 4 3 7 | **1 2** 4 3 5 7 | **1 2** 3 4 5 7 |
| 1 **2 5** 7 4 3 | 1 **2 5** 4 3 7 | 1 **2 4** 3 5 7 | 1 **2 3** 4 5 7 |
| 1 2 **5 7** 4 3 | 1 2 **5 4** 3 7 | 1 2 **4 3** 5 7 | 1 2 3 4 5 7 |
| 1 2 5 **7 4** 3 | 1 2 4 **5 3** 7 | 1 2 3 4 5 7 | |
| 1 2 5 4 **7 3** | 1 2 4 3 5 7 | | |
| 1 2 5 4 3 7 | | | |

- Underlined pairs show the comparisons. For each pass there are size-1 comparisons.
- Total number of comparisons= **(size-1)²**

**Algorithm:** (Bubble Sort) BUBBLE (DATA, N)
Here DATA is an Array with N elements. This algorithm sorts the elements in DATA.

1. for pass=1 to N-1.
2. for (i=0; i<= N-Pass; i++)
3. If DATA[i]>DATA[i+1], then:
   Interchange DATA[i] and DATA[i+1].
   [End of If Structure.]
   [End of inner loop.]
   [End of Step 1 outer loop.]
4. Exit.

## 2.3 Applications of array

Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of (or include) one-dimensional arrays whose elements are records. Arrays are used to implement other data structures, such as heaps, hash tables, deques, queues, stacks, strings, and VLists. One or more large arrays are sometimes used to emulate in-program dynamic memory allocation, particularly memory pool allocation. Historically, this has sometimes been the only way to allocate "dynamic memory" portably. Arrays can be used to determine partial or complete control flow in programs, as a compact alternative to (otherwise repetitive) multiple IF statements. They are known in this context as control tables and are used in conjunction with a purpose built interpreter whose control flow is altered according to values contained in the array.

## 2.4 Sparse matrix

Matrix with maximum zero entries is termed as sparse matrix. It can be represented as:

- Lower triangular matrix: It has non-zero entries on or below diagonal.
- Upper Triangular matrix: It has non-zero entries on or above diagonal.
- Tri-diagonal matrix: It has non-zero entries on diagonal and at the places immediately above or below diagonal.

When a sparse matrix is represented with 2-dimensional array, we waste lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of matrix are filled with zero. That means, totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times.

**Sparse Matrix Representation**

A sparse matrix can be represented by using TWO representations, those are as follows...

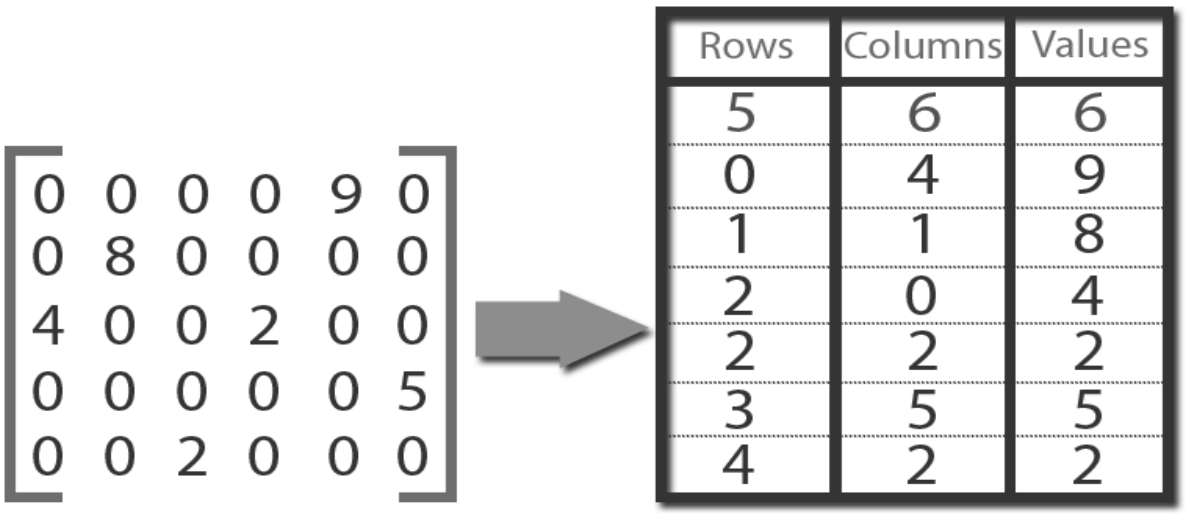
Triplet Representation

Linked Representation

**Triplet Representation**

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores total rows, total columns and total non-zero values in the matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...

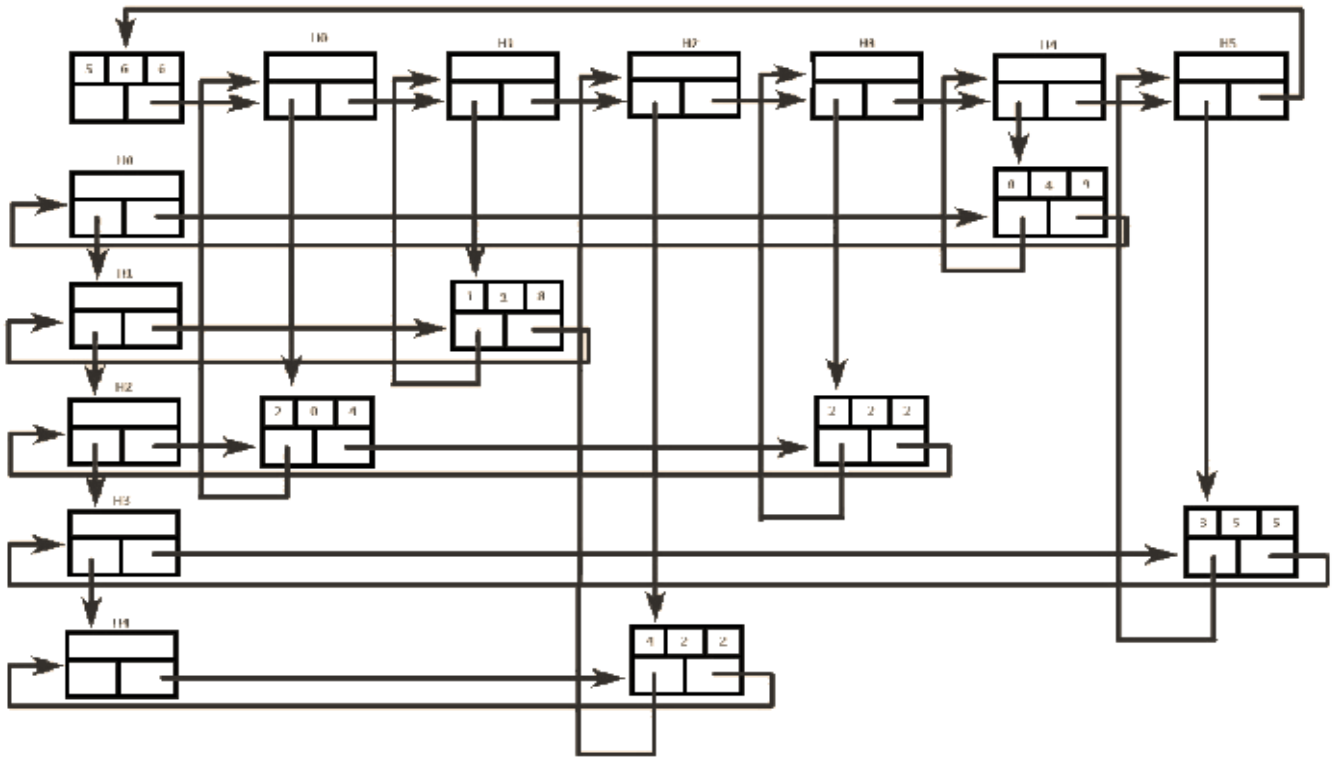



In above example matrix, there are only 6 non-zero elements ( those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

**Linked Representation**

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely header node and element node. Header node consists of three fields and element node consists of five fields as shown in the image...


Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below image...



In above representation, H0, H1,...,H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

In this representation, in each row and column, the last node right field points to its respective header node.

# UNIT-3

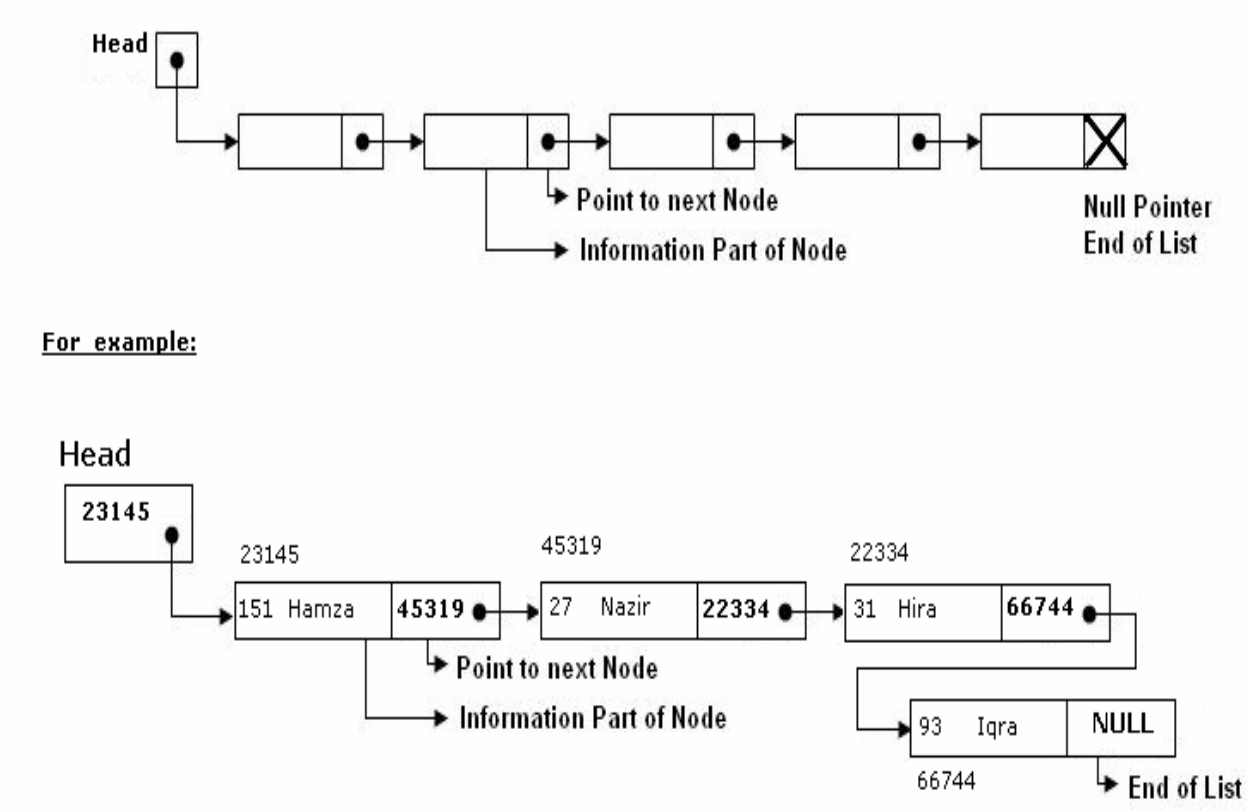
## Linked Lists

### 3.1 Introduction

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**"**. Each node is divided into two parts.

- The first part contains the information of the element.
- The second part called the link field contains the address of the next node in the list.

To see this more clearly lets look at an example:

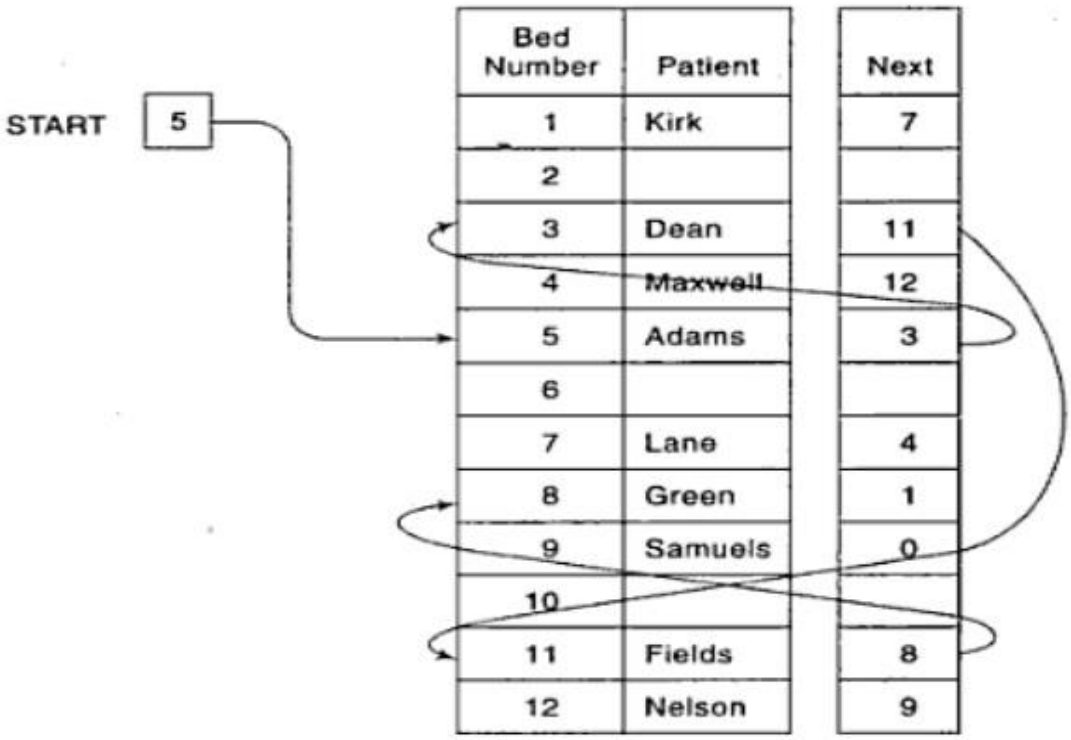



The ***Head*** is a special pointer variable which contains the address of the first node of the list. If there is no node available in the list then ***Head*** contains ***NULL***value that means, List is empty.The left part of the each node represents the information part of the node, which may contain an entire record of data (e.g. ID, name, marks, age etc). the right part represents pointer/link to the next node. The next pointer of the last node is ***null*** pointer signal the end of the list.

### 3.2 Representation of Linked list in memory

Linked list is maintained in memory by two linear arrays: **INFO** and **LINK** such that **INFO [K]** and **LINK [K]** contain respectively the information part and the next pointer field of node of **LIST**. LIST also requires a variable name such as **START** which contains the location of

the beginning of the list and the next pointer denoted by **NULL** which indicates the end of the **LIST**.




### 3.3 Operations on Linked List

There are several operations associated with linked list i.e.

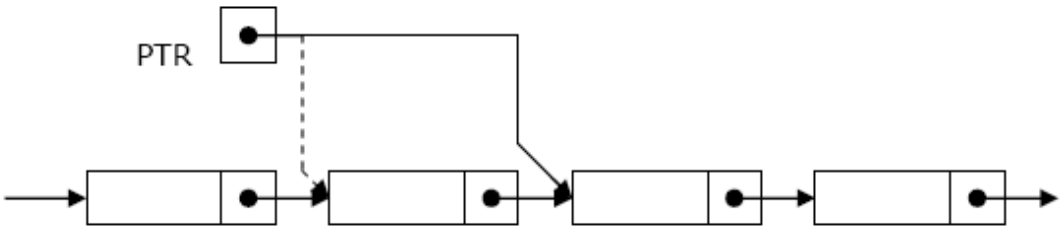
**a) Traversing a Linked List**

Suppose we want to traverse LIST in order to process each node exactly once. The traversing algorithm uses a pointer variable PTR which points to the node that is currently being processed.

Accordingly, PTR->NEXT points to the next node to be processed so,

PTR=HEAD [ Moves the pointer to the first node of the list]

PTR=PTR->NEXT [ Moves the pointer to the next node in the list.]



**Algorithm:** (Traversing a Linked List) Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of list. The variable PTR point to the node currently being processed.

1. Set PTR=HEAD. [Initializes pointer PTR.]
2. Repeat Steps 3 and 4 while PTR!=NULL.
3. Apply PROCESS to PTR-> INFO.
4. Set PTR= PTR-> NEXT [PTR now points to the next node.]
   [End of Step 2 loop.]
5. Exit.

**b) Searching a Linked List:**

Let list be a linked list in the memory and a specific ITEM of information is given to search. IfITEM is actually a key value and we are searching through a LIST for the record containing ITEM, then ITEM can appear only once in the LIST. Search for wanted ITEM in List can be performed by traversing the list using a pointer variable PTR and comparing ITEM with the contents PTR->INFO of each node, one by one of list.

**Algorithm:** SEARCH(INFO, NEXT, HEAD, ITEM, PREV, CURR, SCAN)
LIST is a linked list in the memory. This algorithm finds the location LOC of the node where ITEM first appear in LIST, otherwise sets LOC=NULL.

1. Set PTR=HEAD.
2. Repeat Step 3 and 4 while PTR≠NULL:
3. if ITEM = PTR->INFO then:
   Set LOC=PTR, and return. [Search is successful.]
   [End of If structure.]
4. Set PTR=PTR->NEXT
   [End of Step 2 loop.]
5. Set LOC=NULL, and return. [Search is unsuccessful.]
6. Exit.

**Searching in sorted list**

Algorithm: **SRCHSL (INFO, LINK, START, ITEM, LOC)**

LIST is sorted list (Sorted in ascending order) in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST or sets LOC=NULL.

1. Set PTR:= START

2. Repeat while PTR ≠ NULL
If ITEM >INFO[PTR], then:
Set PTR := LINK[PTR]
Else If ITEM = INFO[PTR], then:
Set LOC := PTR
Return
Else Set LOC:= NULL
Return
[End of If structure]
[End of step 2 Loop]

3. Set LOC:= NULL

4. Return

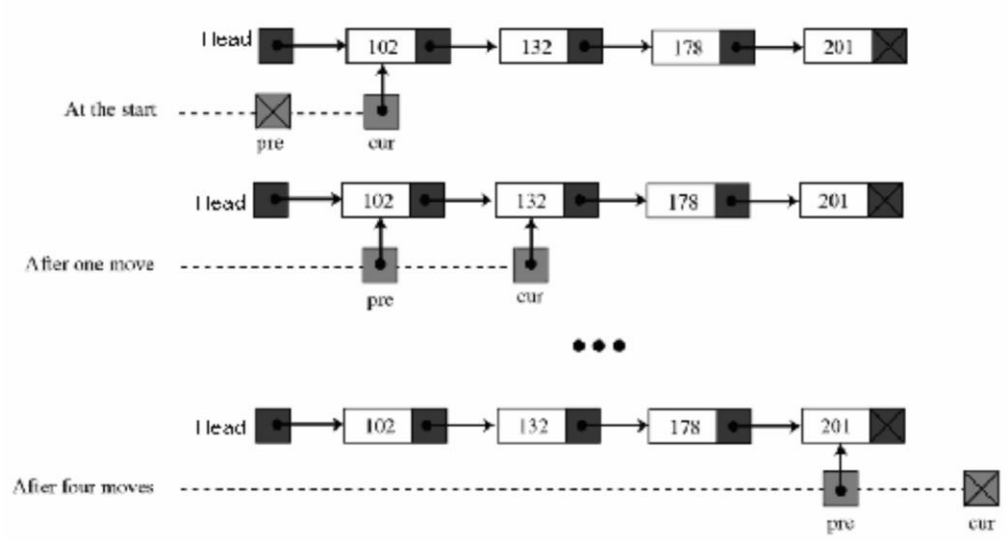
**Search Linked List for insertion and deletion of Nodes:**

Both insertion and deletion operations need searching the linked list.

- To add a new node, we must identify the logical predecessor (address of previous node) where the new node is to be inserting.
- To delete a node, we must identify the location (addresses) of the node to be deleted and its logical predecessor (previous node).

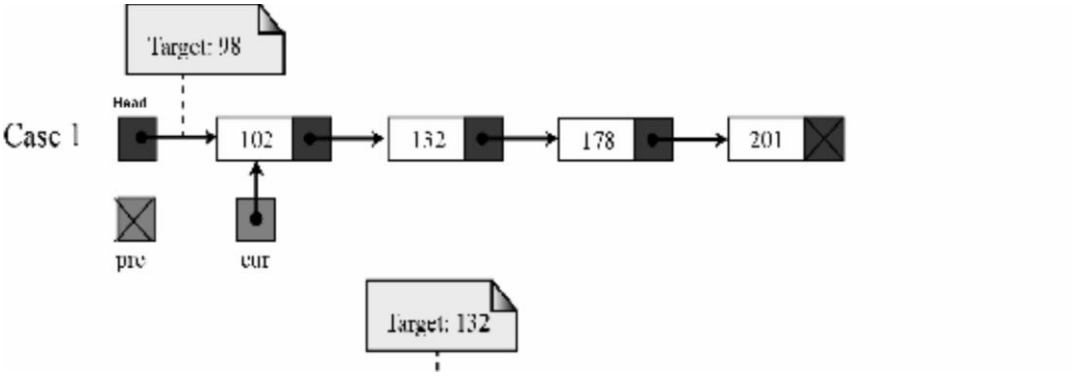
**Basic Search Concept**

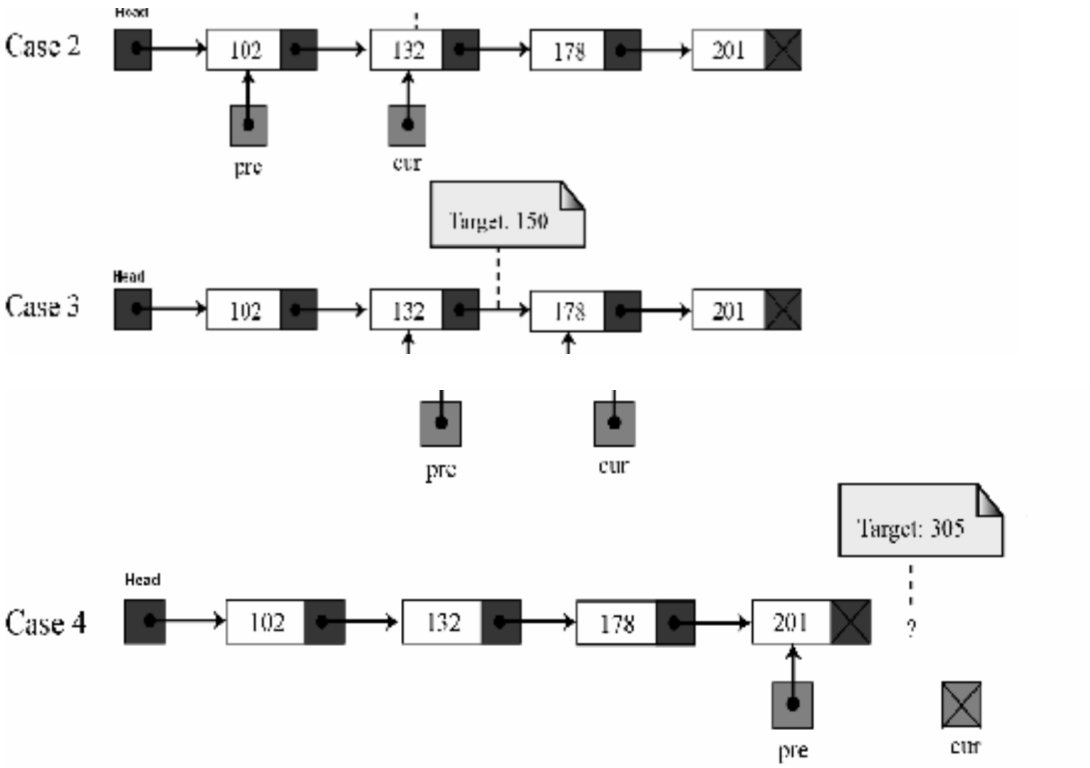
Assume there is a sorted linked list and we wish that after each insertion/deletion this list should always be sorted. Given a target value, the search attempts to locate the requested node in the linked list. Since nodes in a linked list have no names, we use two pointers, **pre** (for previous) and**cur** (for current) nodes. At the beginning of the search, the **pre** pointer is **null** and the **cur** pointer points to the first node (**Head**). The search algorithm moves the two pointers together towards the end of the list. Following Figure shows the movement of these two pointers through the list in an extreme case scenario: when the target value is larger than any value in the list.




Moving of ***pre*** and ***cur*** pointers in searching a linked list

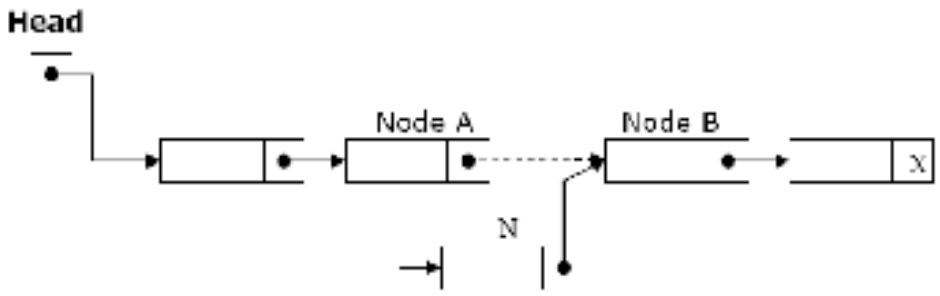
Values of ***pre*** and ***cur*** pointers in different cases





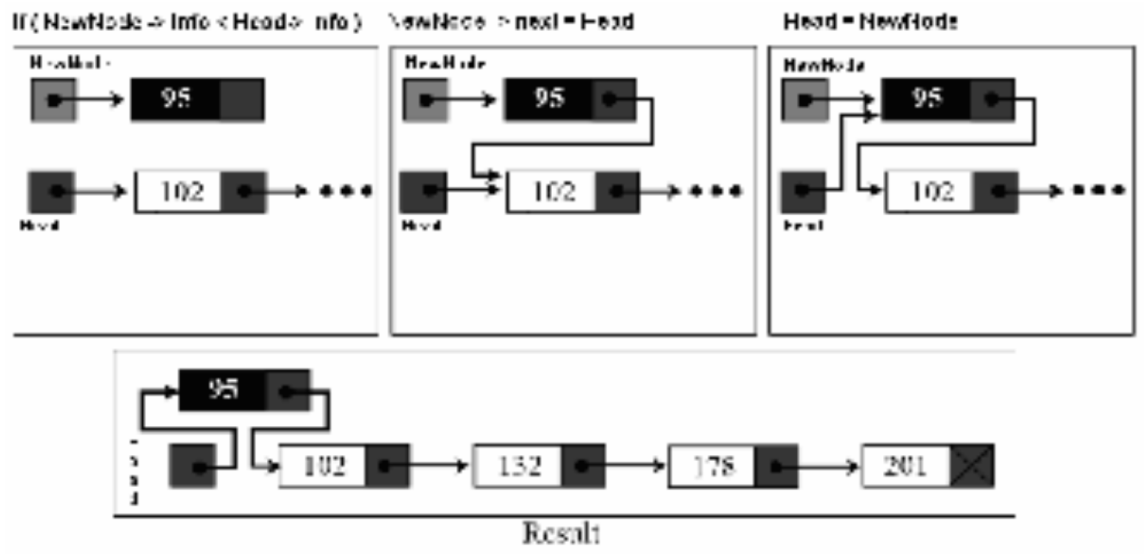

**c) Insertion into a Linked List:**

If a node N is to be inserted into the list between nodes **A** and **B** in a linked list named LIST. It Can be shown as:




**Inserting at the Beginning of a List:**

If the linked list is sorted list and new node has the least low value already stored in the list i.e. (***if New->info < Head->info)*** then new node is inserted at the beginning / Top of the list.




**Algorithm:** INSFIRST (INFO, LINK,START,AVAIL,ITEM)
This algorithm inserts ITEM as the first node in the list
Step 1: [OVERFLOW ?] If AVAIL=NULL, then
Write: OVERFLOW
Return
Step 2: [Remove first node from AVAIL list ]
Set NEW:=AVAIL and AVAIL:=LINK[AVAIL].
Step 3: Set INFO[NEW]:=ITEM [Copies new data into new node]
Step 4: Set LINK[NEW]:= START
[New node now points to original first node]
Step 5: Set START:=NEW [Changes START so it points to new node]
Step 6: Return

**Inserting after a given node**

**Algorithm:** INSLOC(INFO, LINK,START,AVAIL, LOC, ITEM)
This algorithm inserts ITEM so that ITEM follows the node with location LOC or inserts ITEM as the first node when LOC =NULL
Step 1: [OVERFLOW] If AVAIL=NULL, then:
Write: OVERFLOW
Return
Step 2: [Remove first node from AVAIL list]
Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]
Step 3: Set INFO[NEW]:= ITEM [Copies new data into new node]
Step 4: If LOC=NULL, then:
Set LINK[NEW]:=START and START:=NEW
Else:
Set LINK[NEW]:=LINK[LOC] and LINK[LOC]:= NEW
[End of If structure]
Step 5: Return

**Inserting a new node in list:**

The following algorithm inserts an ITEM into LIST.

**Algorithm:** INSERT( ITEM)
[This algorithm add newnodes at any position (Top, in Middle and at End) in the List ]

1. Create a **NewNode** node in memory
2. Set **NewNode ->** INFO =ITEM. [Copies new data into INFO of new node.]
3. Set **NewNode ->** NEXT = NULL. [Copies NULL in NEXT of new node.]
4. If **HEAD**=NULL, then **HEAD**=**NewNode** and return. [Add first node in list]
5. if **NewNode->** INFO < **HEAD**->INFO
   then Set **NewNode**->NEXT=**HEAD** and **HEAD**=**NewNode** and return
   [Add node on top of existing list]
6. PrevNode = NULL, CurrNode=NULL;
7. for(CurrNode =HEAD; CurrNode != NULL; CurrNode = CurrNode ->NEXT)
   { if(NewNode->INFO <= CurrNode ->INFO)
   {
   break the loop
   }
   PrevNode = CurrNode;
   } [ end of loop ]
   [Insert after PREV node (in middle or at end) of the list]
8. Set **NewNode**->NEXT = PrevNode->NEXT and
9. Set PrevNode->NEXT= **NewNode**.
10. Exit.

**d) Delete a node from list:**

The following algorithm deletes a node from any position in the LIST.

**Algorithm:** DELETE(ITEM)
LIST is a linked list in the memory. This algorithm deletes the node where ITEM first appear in LIST, otherwise it writes "NOT FOUND"

1. if **Head** =NULL then write: "Empty List" and return [Check for Empty List]
2. if ITEM = **Head** -> info then: [ Top node is to delete ]
   Set **Head** = **Head** -> next and return
3. Set PrevNode = NULL, CurrNode=NULL.
4. for(CurrNode =HEAD; CurrNode != NULL; CurrNode = CurrNode ->NEXT)
   { if (ITEM = CurrNode ->INFO ) then:
   {
   break the loop
   }
   Set PrevNode = CurrNode;
   } [ end of loop ]
5. if(CurrNode = NULL) then write : Item not found in the list and return
6. [delete the current node from the list]
   Set PrevNode ->NEXT = CurrNode->NEXT
7. Exit.

**e) Concatenating two linear linked lists**

**Algorithm:** Concatenate(INFO,LINK,START1,START2)
This algorithm concatenates two linked lists with start
pointers START1 and START2
Step 1: Set PTR:=START1
Step 2: Repeat while LINK[PTR]≠NULL:
Set PTR:=LINK[PTR]
[End of Step 2 Loop]
Step 3: Set LINK[PTR]:=START2
Step 4: Return

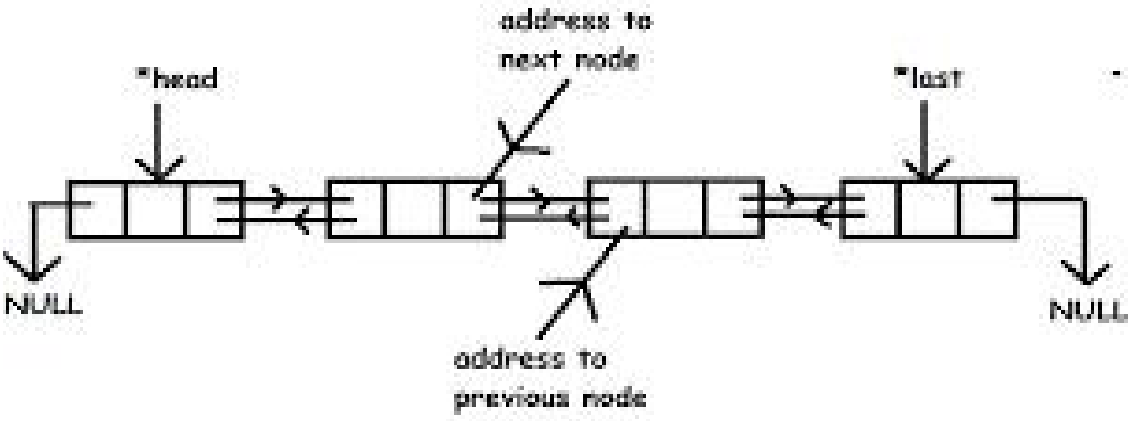
**// A Program that exercise the operations on Liked List**

```
#include<iostream.h>
#include <malloc.h>
#include <process.h>
struct node
{
int info;
struct node *next;
};
struct node *Head=NULL;
struct node *Prev,*Curr;
voidAddNode(int ITEM)
{
struct node *NewNode;
NewNode = new node;
// NewNode=(struct node*)malloc(sizeof(struct node));
NewNode->info=ITEM; NewNode->next=NULL;
if(Head==NULL) { Head=NewNode; return; }
if(NewNode->info < Head->info)
{ NewNode->next = Head; Head=NewNode; return;}
Prev=Curr=NULL;
for(Curr = Head ; Curr != NULL ; Curr = Curr ->next)
{
if(NewNode->info <Curr ->info) break;
elsePrev = Curr;
}
NewNode->next = Prev->next;
Prev->next = NewNode;
} // end of AddNode function
voidDeleteNode()
        { intinf;
if(Head==NULL){ cout<< "\n\n empty linked list\n"; return;}
cout<< "\n Put the info to delete: ";
cin>>inf;
if(inf == Head->info) // First / top node to delete
{ Head = Head->next; return;}
```

```
Prev = Curr = NULL;
for(Curr = Head ; Curr != NULL ; Curr = Curr ->next )
{
if(Curr ->info == inf) break;
Prev = Curr;
}
if(Curr == NULL)
cout<<inf<< " not found in list \n";
else
{ Prev->next = Curr->next; }
}// end of DeleteNode function
void Traverse()
{
for(Curr = Head; Curr != NULL ; Curr = Curr ->next )
cout<<Curr ->info<<"\t";
} // end of Traverse function
int main()
{ intinf, ch;
while(1)
{ cout<< " \n\n\n\n Linked List Operations\n\n";
cout<< " 1- Add Node \n 2- Delete Node \n";
cout<< " 3- Traverse List \n 4- exit\n";
cout<< "\n\n Your Choice: "; cin>>ch;
switch(ch)
{ case 1: cout<< "\n Put info/value to Add: ";
cin>>inf);
AddNode(inf);
break;
case 2: DeleteNode(); break;
case 3: cout<< "\n Linked List Values:\n";
Traverse(); break;
case 4: exit(0);
} // end of switch
} // end of while loop
return 0;
} // end of main ( ) function
```

### 3.4 Doubly Linked Lists

A doubly linked list is a list that contains links to next and previous nodes. Unlike singly linked lists where traversal is only one way, doubly linked lists allow traversals in both ways.




**Dynamic Implementation of doubly linked list**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
structnode
{
structnode *previous;
intdata;
structnode *next;
}*head, *last;
voidinsert_begning(intvalue)
{
structnode *var,*temp;
var=(structnode *)malloc(sizeof(structnode));
var->data=value;
if(head==NULL)
{
head=var;
head->previous=NULL;
head->next=NULL;
last=head;
}
else
{
temp=var;
temp->previous=NULL;
temp->next=head;
head->previous=temp;
head=temp;
}
}
voidinsert_end(intvalue)
{
structnode *var,*temp;
```

```
var=(structnode *)malloc(sizeof(structnode));
var->data=value;
if(head==NULL)
{
head=var;
head->previous=NULL;
head->next=NULL;
last=head;
}
else
{
last=head;
while(last!=NULL)
{
temp=last;
last=last->next;
}
last=var;
temp->next=last;
last->previous=temp;
last->next=NULL;
}
}
intinsert_after(intvalue, intloc)
{structnode *temp,*var,*temp1;
var=(structnode *)malloc(sizeof(structnode));
var->data=value;
if(head==NULL)
{
head=var;
head->previous=NULL;
head->next=NULL;
}
else
{
temp=head;
while(temp!=NULL && temp->data!=loc)
{
temp=temp->next;
}
if(temp==NULL)
{
printf("\n%d is not present in list ",loc);
}
else
{
temp1=temp->next;
temp->next=var;
var->previous=temp;
var->next=temp1;
```

```
temp1->previous=var;
}
}
last=head;
while(last->next!=NULL)
{
last=last->next;
}
}
intdelete_from_end()
{
structnode *temp;
temp=last;
if(temp->previous==NULL)
{
free(temp);
head=NULL;
last=NULL;
return0;
}
printf("\nData deleted from list is %d \n",last->data);
last=temp->previous;
last->next=NULL;
free(temp);
return0;
}
intdelete_from_middle(intvalue)
{
structnode *temp,*var,*t, *temp1;
temp=head;
while(temp!=NULL)
{
if(temp->data == value)
{
if(temp->previous==NULL)
{
free(temp);
head=NULL;
last=NULL;
return0;
}
else
{
var->next=temp1;
temp1->previous=var;
free(temp);
return0;
}
}
else
```

```
{
var=temp;
temp=temp->next;
temp1=temp->next;
}
}
printf("data deleted from list is %d",value);
}
voiddisplay()
{
structnode *temp;
temp=head;
if(temp==NULL)
{
printf("List is Empty");
}
while(temp!=NULL)
{
printf("-> %d ",temp->data);
temp=temp->next;
}
}
intmain()
{
intvalue, i, loc;
head=NULL;
printf("Select the choice of operation on link list");
printf("\n1.) insert at begning\n2.) insert at at\n3.) insert at middle");
printf("\n4.) delete from end\n5.) reverse the link list\n6.) display list\n7.)
exit
while(1)
{
printf("\n\nenter the choice of operation you want to do ");
scanf("%d",&i);
switch(i)
{
case1:
{
printf("enter the value you want to insert in node ");
scanf("%d",&value);
insert_begning(value);
display();
break;
}
case2:
{
printf("enter the value you want to insert in node at last ");
scanf("%d",&value);
insert_end(value);
display();
```

```
break;
}
case3:
{
printf("after which data you want to insert data ");
scanf("%d",&loc);
printf("enter the data you want to insert in list ");
scanf("%d",&value);
insert_after(value,loc);
display();
break;
}
case4:
{
delete_from_end();
display();
break;
}
case5:
{
printf("enter the value you want to delete");
scanf("%d",value);
delete_from_middle(value);
display();
break;
}
case6 :
{
display();
break;
}
case7 :
{
exit(0);
break;
}
}
}
printf("\n\n%d",last->data);
display();
getch();
}
```

### 3.5 Circular Linked List

A circular linked list is a linked list in which last element or node of the list points to first node. For non-empty circular linked list, there are no NULL pointers. The memory declarations for representing the circular linked lists are the same as for linear linked lists. All operations performed on linear linked lists can be easily extended to circular linked lists with following

exceptions:
• While inserting new node at the end of the list, its next pointer field is made to point to the first node.
• While testing for end of list, we compare the next pointer field with address of the first Node.Circular linked list is usually implemented using **header linked list**. Header linked list is a linked list which always contains a special node called the **header node**, at the beginning of the list. This header node usually contains vital information about the linked list such as number of nodes in lists, whether list is sorted or not etc. Circular header lists are frequently used instead of ordinary linked lists as manyoperations are much easier to state and implement using header listThis comes from the following two properties of circular header linked lists:
• The null pointer is not used, and hence all pointers contain valid addresses
• Every (ordinary) node has a predecessor, so the first node may not require a special case.

Algorithm: **(Traversing a circular header linked list)**

//This algorithm traverses a **circular header linked list**

START pointer storing the address of the header node.
Step 1: Set PTR:=LINK[START]
Step 2: Repeat while PTR≠START:
Apply PROCESS to INFO[PTR]
Set PTR:=LINK[PTR]
[End of Loop]
Step 3: Return

**Searching a circular header linked list**

Algorithm: SRCHHL(INFO,LINK,START,ITEM,LOC)

//This algorithm searches a circular header linked list

Step 1: Set PTR:=LINK[START]
Step 2: Repeat while INFO[PTR]≠ITEM and PTR≠START:
Set PTR:=LINK[PTR]
[End of Loop]
Step 3: If INFO[PTR]=ITEM, then:
Set LOC:=PTR
Else:
Set LOC:=NULL
[End of If structure]
Step 4: Return

**Deletion from a circular header linked list**

**Algorithm:** DELLOCHL(INFO,LINK,START,AVAIL,ITEM)
This algorithm deletes an item from a circular header linked list.
Step 1: CALL FINDBHL(INFO,LINK,START,ITEM,LOC,LOCP)

Step 2: If LOC=NULL, then:
Write: 'item not in the list'
Exit
Step 3: Set LINK[LOCP]:=LINK[LOC] [Node deleted]
Step 4: Set LINK[LOC]:=AVAIL and AVAIL:=LOC
[Memory retuned to Avail list]
Step 5: Return

**Searching in circular list**

Algorithm: FINDBHL(NFO,LINK,START,ITEM,LOC,LOCP)
This algorithm finds the location of the node to be deleted
and the location of the node preceding the node to be
deleted
Step 1: Set SAVE:=START and PTR:=LINK[START]
Step 2: Repeat while INFO[PTR]≠ITEM and PTR≠START
Set SAVE:=PTR and PTR:=LINK[PTR]
[End of Loop]
Step 3: If INFO[PTR]=ITEM, then:
Set LOC:=PTR and LOCP:=SAVE
Else:
Set LOC:=NULL and LOCP:=SAVE
[End of If Structure]
Step 4: Return

**Insertion in a circular header linked list**

Algorithm: INSRT(INFO,LINK,START,AVAIL,ITEM,LOC)
This algorithm inserts item in a circular header linked list
after the location LOC
Step 1:If AVAIL=NULL, then
Write: 'OVERFLOW'
Exit
Step 2: Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]
Step 3: Set INFO[NEW]:=ITEM
Step 4: Set LINK[NEW]:=LINK[LOC]
Set LINK[LOC]:=NEW
Step 5: Return

**Insertion in a sorted circular header linked list**

Algorithm: INSERT(INFO,LINK,START,AVAIL,ITEM)
This algorithm inserts an element in a sorted circular header
linked list
Step 1: CALL FINDA(INFO,LINK,START,ITEM,LOC)
Step 2: If AVAIL=NULL, then
Write: 'OVERFLOW'
Return
Step 3: Set NEW:=AVAIL and AVAIL:=LINK[AVAIL]
Step 4: Set INFO[NEW]:=ITEM

Step 5: Set LINK[NEW]:=LINK[LOC]
Set LINK[LOC]:=NEW
Step 6: Return
Algorithm: FINDA(INFO,LINK,ITEM,LOC,START)
This algorithm finds the location LOC after which to
insert
Step 1: Set PTR:=START
Step 2: Set SAVE:=PTR and PTR:=LINK[PTR]
Step 3: Repeat while PTR≠START
If INFO[PTR]>ITEM, then
Set LOC:=SAVE
Return
Set SAVE:=PTR and PTR:=LINK[PTR]
[End of Loop]
Step 4: Set LOC:=SAVE
Step 5: Return

### 3.6 Applications of Linked Lists:

- Linked Lists can be used to implement Stacks , Queues.
- Linked Lists can also be used to implement Graphs. (Adjacency list representation of Graph).
- Implementing Hash Tables :- Each Bucket of the hash table can itself be a linked list. (Open chain hashing).
- Undo functionality in Photoshop or Word . Linked list of states.
- A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term.
- However, for any polynomial operation , such as addition or multiplication of polynomials , linked list representation is more easier to deal with.
- Linked lists are useful for dynamic memory allocation.
- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running.
- All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.

### 3.7 Comparison of Linked List and Array

Comparison between array and linked list are summarized in following table –

# Difference between Array and Linked List

| S.No. | ARRAY | LINKED LIST |
|---|---|---|
| 1. | An array is a grouping of data elements of equivalent data type. | A linked list is a group of entities called a node. The node includes two segments: data and address. |
| 2. | It stores the data elements in a contiguous memory zone. | It stores elements randomly, or we can say anywhere in the memory zone. |
| 3. | In the case of an array, memory size is fixed, and it is not possible to change it during the run time. | In the linked list, the placement of elements is allocated during the run time. |
| 4. | The elements are not dependent on each other. | The data elements are dependent on each other. |
| 5. | The memory is assigned at compile time. | The memory is assigned at run time. |
| 6. | It is easier and faster to access the element in an array. | In a linked list, the process of accessing elements takes more time. |
| 7. | In the case of an array, memory utilization is ineffective. | In the case of the linked list, memory utilization is effective. |
| 8 | When it comes to executing any operation like insertion, deletion, array takes more time. | When it comes to executing any operation like insertion, deletion, the linked list takes less time. |

# Unit 4

## Stacks, Queues and Recursion

A stack is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the "top", and the opposite end is known as the "base".

**The stack abstract data type is an ordered collection of items where items are added to and removed from the top.** Operations on the Stack

Stack() creates a new, empty stack

- push(item) adds the given item to the top of the stack and returns nothing
- pop() removes and returns the top item from the stack
- peek() returns the top item from the stack but doesn't remove it (the stack isn't modified)
- is_empty() returns a boolean representing whether the stack is empty
- size() returns the number of items on the stack as an integer

**Implementation of a Stack**

There are two ways in which we can implement the stack data structure.

**1. Using an array**

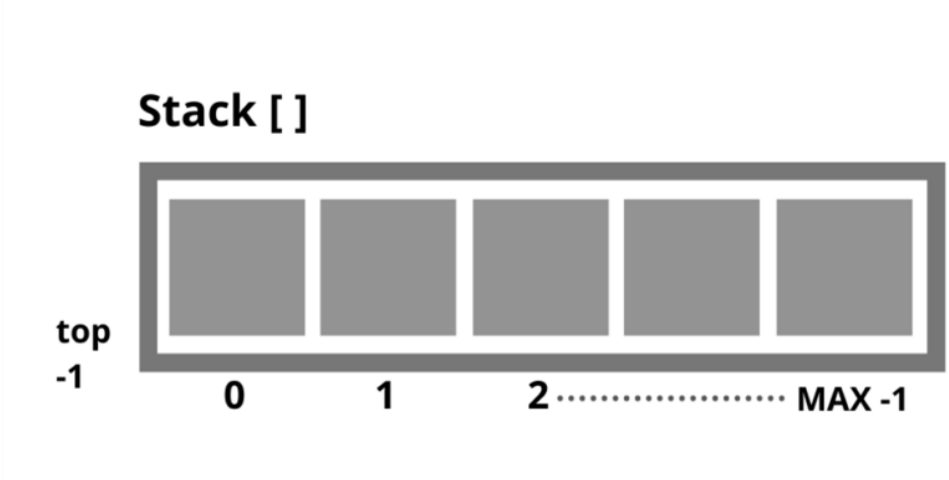
In this representation, we will be implementing the stack using an array whose index starts with 0.

Before inserting or deleting an element, there are two conditions that we have to check, the first is the overflow condition and the second is the underflow condition.

The overflow condition simply checks whether the stack is full or not. If it is already full, then you can't insert a new element on to the stack.

The underflow condition checks whether the stack is empty or not because if the stack is already empty and if we are trying to delete an element that does not even exist as the stack is already empty, we will definitely get an error.

These are some corner case conditions that we have to check to make sure that our program works perfectly in any situation without any errors.




**//Implementation of Push and POP operation**

```
void push (int item) {
   // MAX – 1 represents the top of the stack
   // Checks if stack full or not
   if (top == MAX - 1) {
      printf("Overflow! The Stack is already full.");
   }
   else {
      top = top + 1;
      Stack[top] = item;
   }
}
// Checking the Underflow condition & deleting the elements from the stack
int pop () {
   int temp;
   // Checks if stack is empty or not
   if (top == -1) {
      printf("Underflow! The stack is already empty.");
      return -1;
   }
else {
```

```
    temp = stack[top]; // Storing current value of stack in temp
    top = top – 1; // Decrementing the pointer
    }
return temp;
}
```

Here, the top pointer keeps track of the topmost location of the stack. If the value of the top pointer is -1 then it represents that the stack is empty. This point is also applicable to the linked list representation of stack.

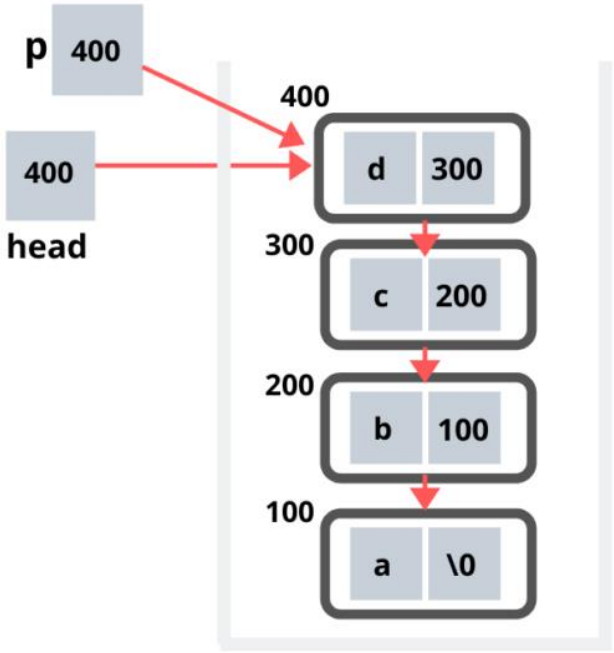
In the stack, when we pop an element, we are not actually deleting the element from the stack as we do in a linked list. Here, we are only decrementing the top pointer and the value inside that particular stack block will still be there.

**2. Using the linked list**

In this representation, we will be implementing the stack data structure using a linked list.

As we know, the linked list consists of various nodes and each node has two things. One is the data and the other is the pointer that points to the next node.

But, before inserting an element, we first have to create a new node and then add that node to a linked list. Here, also, we have to check the overflow and underflow conditions.




**Stack Implementation Using Linked List**

```
struct node {
  int i;
  struct node *link;
};
// Inserting an element
void push (int item) {
  //Creating a single node
  struct node *p = (struct node *) malloc(sizeOf(struct node));
  // Checking for Overflow condition
  if (p == NULL) {
    printf("Error of malloc.");
    return;
  }
  p -> data = item;
  p -> link = head;
  head = p;
}
// Deleting an element
void pop () {
```

```
  int item;
  struct node *p;
  // Checking the Underflow condition
  if (head == NULL) {
    printf("Underflow");
    return -1;
  }

  item = head -> i;
  p = head;
  head = head -> next;
  free(p);
}
```

**Applications of Stack**

- Backtracking: This is a process when you need to access the most recent data element in a series of elements.
- Depth first Search can be implemented.
- The function call mechanism.
- Simulation of Recursive calls: The compiler uses one such data structure called stack for implementing normal as well as recursive function calls.
- Parsing: Syntax analysis of compiler uses stack in parsing the program.
- Web browsers store the addresses of recently visited sites on a stack.
- The undo-mechanism in an editor.
- Expression Evaluation: How a stack can be used for checking on syntax of an expression.
    - **Infix expression:** It is the one, where the binary operator comes between the operands. e. g., A + B * C.
    - **Postfix expression:** Here, the binary operator comes after the operands. e.g., ABC * +

- **Prefix expression:** Here, the binary operator proceeds the operands. e.g.,+ A * BC
- This prefix expression is equivalent to A + (B * C) infix expression. Prefix notation is also known as Polish notation. Postfix notation is also known as suffix or Reverse Polish notation.

- **Reversing a List:** First push all the elements of string in stack and then pop elements.
- **Expression conversion:** Infix to Postfix, Infix to Prefix, Postfix to Infix, and Prefix to Infix
- Implementation of **Towers of Hanoi**

**2: Notations**

There are 3 notations which are mostly used in computer system :

- INFIX notation : The notation in which the operator is between the operands.
- PREFIX notation : The notation in which the operator is before the operands.
- POSTFIX notation : The notation in which the operator is after the operands.

Example :

Consider an expression as a*b+c

- INFIX notation : a*b+c
- PREFIX notation : +*abc
- POSTFIX notation : ab*c+
- **Converting INFIX notation into POSTFIX notation :**
- For converting infix notation to postfix notation , we use operator stack as data structure. If an operator comes then print it , but if an operator comes then push it in the stack . Now if another operator with high/same priority comes into stack then pop the previous operator and push the new one. But if lower priority operator comes then push it also.
- Example :
- A+(B*C-(D/E-F)*G)*H

| **Stack** | **Input** | **Output** |
|---|---|---|
| Empty | A+(B*C-(D/E-F)*G)*H | - |
| Empty | +(B*C-(D/E-F)*G)*H | A |
| + | (B*C-(D/E-F)*G)*H | A |
| +( | B*C-(D/E-F)*G)*H | A |
| +( | *C-(D/E-F)*G)*H | AB |

| | | |
|---|---|---|
| +(* | C-(D/E-F)*G)*H | AB |
| +(* | -(D/E-F)*G)*H | ABC |
| +(- | (D/E-F)*G)*H | ABC* |
| +(-( | D/E-F)*G)*H | ABC* |
| +(-( | /E-F)*G)*H | ABC*D |
| +(-(/ | E-F)*G)*H | ABC*D |
| +(-(/ | -F)*G)*H | ABC*DE |
| +(-(- | F)*G)*H | ABC*DE/ |
| +(-(- | F)*G)*H | ABC*DE/ |
| +(-(- | )*G)*H | ABC*DE/F |
| +(- | *G)*H | ABC*DE/F- |
| +(-* | G)*H | ABC*DE/F- |
| +(-* | )*H | ABC*DE/F-G |
| + | *H | ABC*DE/F-G*- |
| +* | H | ABC*DE/F-G*- |
| +* | End | ABC*DE/F-G*-H |
| Empty | End | ABC*DE/F-G*-H*+ |

**Evaluating POSTFIX expression :**

For evaluating a postfix expression , we use operand stack as data structure. If an operand comes then push it . Keep pushing the operands until an operator come. When an operator come , then pop two operands , perform that operation and push back the result. Repeat this until the final result is known.

Example: Evaluate the following postfix notation of expression : **456*+**

| **Step** | **Input Symbol** | **Operation** | **Stack** | **Calculation** |
|---|---|---|---|---|
| 1. | 4 | Push | 4 | |
| 2. | 5 | Push | 4,5 | |
| 3. | 6 | Push | 4,5,6 | |
| 4. | * | Pop (2 elements & Evaluate) | 4 | 5*6 = 30 |
| 5. | | Push result (30) | 4,30 | |
| 6. | + | Pop (2 elements & Evaluate) | Empty | 4 + 30 = 34 |
| 7. | | Push result (34) | 34 | |
| 8. | | No-more elements (pop) | Empty | 34 (Result) |

**Recursion:**

Recursion may be defined as a programming technique/method of defining a function in terms of its own definition. For example:

the Fibonacci series: 0,1,1,2,3,5,8…….

f (n) = f(n-1) + f(n-2)

**f(0) = f(1) = 1 is the base case**

and Factorial Function: n! = n*(n-1)!

Factorial(n) = n*Factorial(n-1)

**Factorial(0)=Factorial(1)=1 is the base case**

In programming recursion is a function which calls to the same function. In other words, a recursive function is one that calls itself.

It is a Problem solving technique which uses Divide-and-Conquer method to solve a problem by breaking into smaller problems, solving sub-problems recursively and finally assemble sub-solutions.

**Advantages of Recursion:**

- Recursion leads to solutions that are
  - compact
  - simple
  - easy to understand
  - easy to prove correct
- Recursion emphasizes thinking about a problem at a high level of abstraction

**Disadvantages of Recursion:**

- Recursive calls can result in a an infinite loop of calls if base-case( in order to stop recursive call) is not defined properly.
- It has more execution overheads as compared to iterations.

**Rules/ Requirements to write a recursive function:**

- Each recursive call should be on a smaller instance of the same problem, that is, a smaller subproblem.
- The recursive calls must eventually reach a base case, which is solved without further recursion.

**Algorithm to find factorial of a number is given below:**

**Step 1:** Start
**Step 2:** Read number *n*
**Step 3:** Call factorial(n)
**Step 4:** Print factorial f
**Step 5:** End

Algorithm factorial(n)
**Step 1:** If n==1 then return 1
**Step 2:** Else
f=n*factorial(n-1)
**Step 3:** Return f
Step 4: End

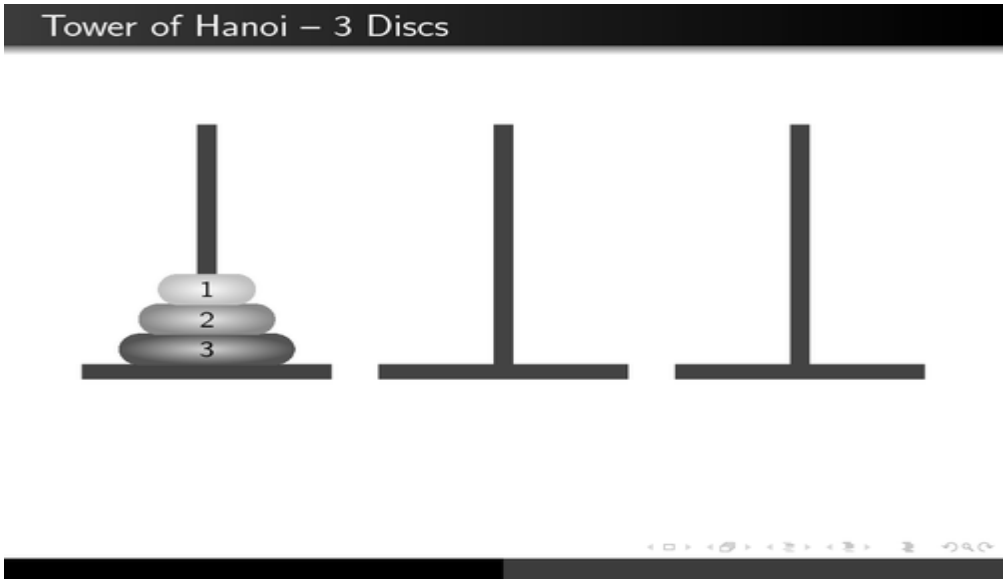
# Difference between Recursion and Iteration

The following table highlights all the important differences between recursion and iteration −

| Recursion | Iteration |
|---|---|
| Recursion uses the selection structure. | Iteration uses the repetition structure. |
| Infinite recursion occurs if the step in recursion doesn't reduce the problem to a smaller problem. It also becomes infinite recursion if it doesn't convert on a specific condition. This specific condition is known as the base case. | An infinite loop occurs when the condition in the loop doesn't become False ever. |
| The system crashes when infinite recursion is encountered. | Iteration uses the CPU cycles again and again when an infinite loop occurs. |
| Recursion terminates when the base case is met. | Iteration terminates when the condition in the loop fails. |
| Recursion is slower than iteration since it has the overhead of maintaining and updating the stack. | Iteration is quick in comparison to recursion. It doesn't utilize the stack. |
| Recursion uses more memory in comparison to iteration. | Iteration uses less memory in comparison to recursion. |
| Recursion reduces the size of the code. | Iteration increases the size of the code. |

**Tower of Hanoi Problem:**

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted –




These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. The number of disks may increase upto n, but the no of towers remains the same i.e 3.

The mission is to move all the disks from source tower to some another tower without violating the sequence in which they were placed originally.. A few rules to be followed for Tower of Hanoi are −

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Tower of Hanoi puzzle with n disks can be solved in minimum $\mathbf{2^n-1}$ steps

Algorithm

To write an algorithm for Tower of Hanoi, let us mark three towers with name, **source**, **destination** and **intermediate**(only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination towers.

If we have 2 disks −

- First, we move the smaller (top) disk to intermediate tower.
- Then, we move the larger (bottom) disk to destination towers.
- And finally, we move the smaller disk from towers to destination towers.

- The algorithm for Tower of Hanoi with more than two disks divides the stack of disks in two parts. The largest disk ($n^{th}$ disk) is in one part and all other (n-1) disks are in the second part.

- Our ultimate aim is to move disk **n** from source to destination and then put all other (n-1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

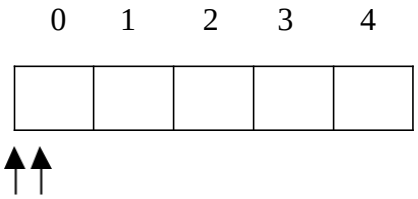
```
Algorithm Hanoi(disk, source, inter, dest)

Step 1: IF disk is equal 1, THEN
            move disk from source to destination  Exit
Step2: Hanoi(disk - 1, source, destination, intermediate)
             move disk from source to destination
Step 3:Hanoi (disk - 1, intermediate, source, destination)
 END
```

**Introduction to queue:**

A queue is a data structure that is best described as "first in, first out". A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. A real world example of a queue is people waiting in line at the bank. As each person enters the bank, he or she is "enqueued" at the back of the line. When a teller becomes available, they are "dequeued" at the front of the line.

**Representation of a Queue using Array:**

Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.

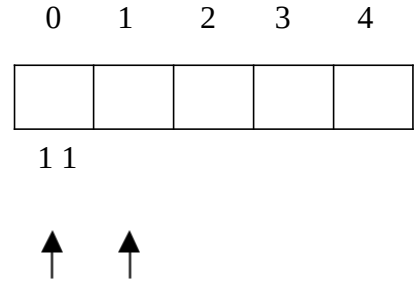



Q u e u e E mp t y

FRONT = REAR = 0

F R

Now, insert 11 to the queue. Then queue status will be:




REAR=REAR+1=1

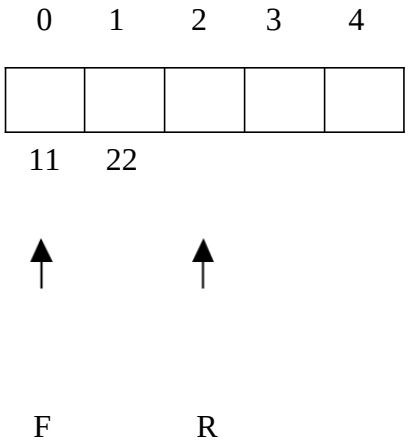
FRONT = 0

F R

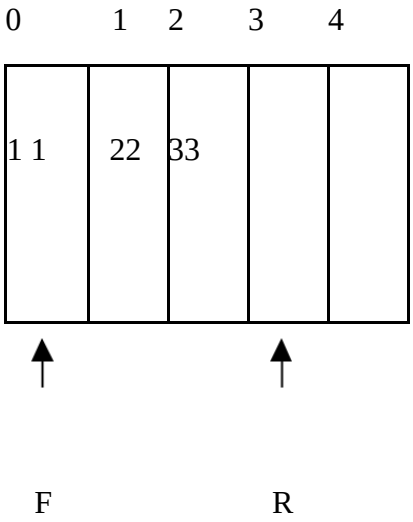
Next, insert 22 to the queue. Then the queue status is:

REAR=REAR+1=2

FRONT = 0



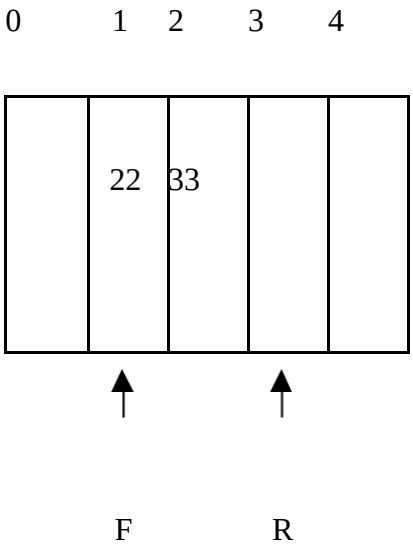

Again insert another element 33 to the queue. The status of the queue is:




REAR=REAR+1=3

FRONT = 0

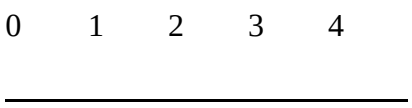
Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



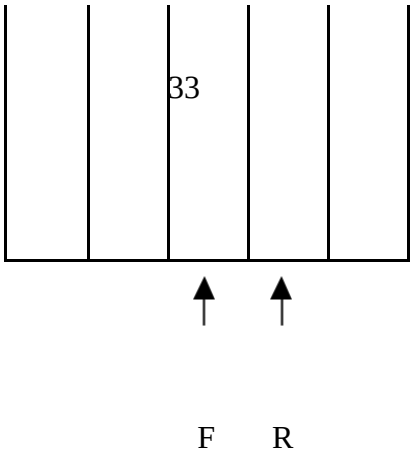

REAR = 3

FRONT = FRONT + 1 = 1

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:

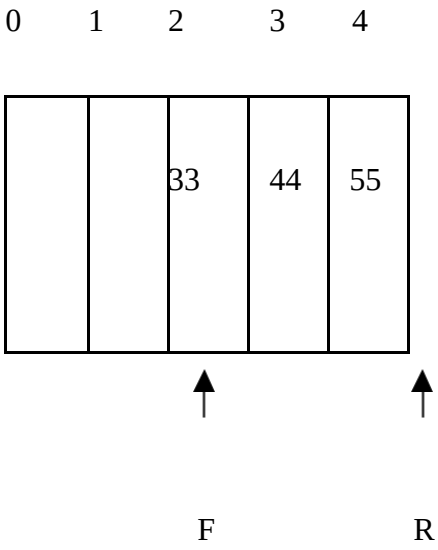



REAR = 3




FRONT = FRONT + 1 = 2

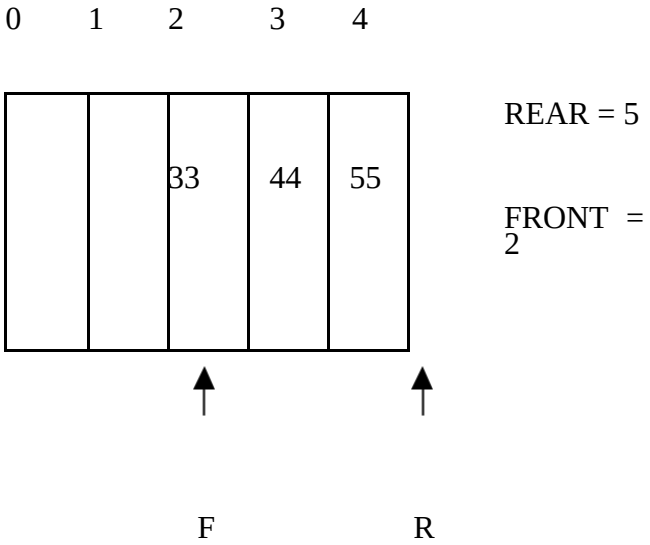
Now, insert new elements 44 and 55 into the queue. The queue status is:



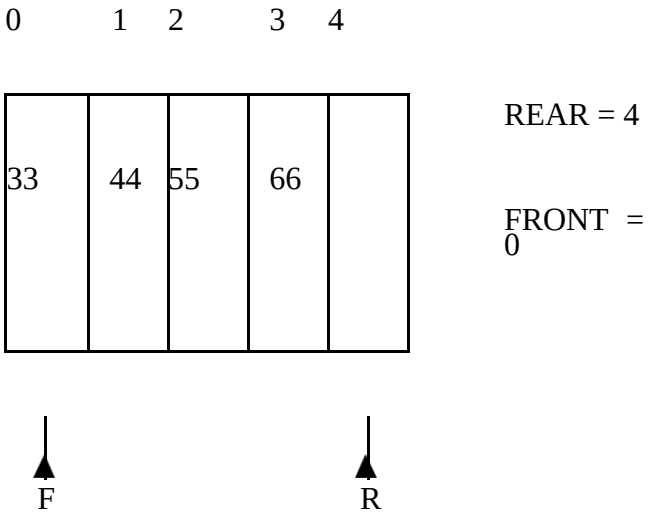

REAR = 5

FRONT = 2

Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:




Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To over come this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:




This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

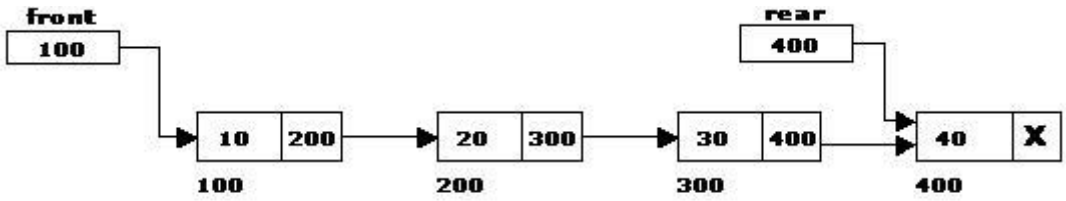
**Procedure for Queue operations using array:**

In order to create a queue we require a one dimensional array Q(1:n) and two variables *front* and *rear*. The conventions we shall adopt for these two variables are that *front* is always 1 less than the actual front of the queue and rear always points to the last element in the queue. Thus, front = rear if and only if there are no elements in the queue. The initial condition then is front = rear = 0.

The various queue operations to perform creation, deletion and display the elements in a queue are as follows:

1. insertQ(): inserts an element at the end of queue Q.
2. deleteQ(): deletes the first element of Q.
3. displayQ(): displays the elements in the queue.

**Linked List Implementation of Queue:** We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers front and rear for our linked queue implementation.



**Source Code:**

```
front = 0

rear = 0

mymax = 3

# Function to create a stack. It initializes size of
stack as 0 def createQueue():

   queue  =  []
   return
   queue
 # Stack is empty when stack size is 0

def isEmpty(queue):

   return len(queue) == 0

# Function to add an item to stack. It increases size
by 1 def enqueue(queue,item):

   queue.append(item)

# Function to remove an item from stack. It decreases
size by 1 def dequeue(queue):

   if      (isEmpty(queue)):
      return    "Queue    is
      empty"

   item=queue[
   0]          del
   queue[0]
   return item

 # Driver   program   to   test   above
functions queue = createQueue()

while           True:
   print("1
   Enqueue")
   print("2
   Dequeue")
   print("3
   Display")
   print("4 Quit")
```

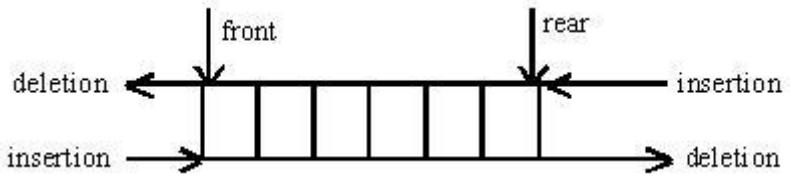
```
ch=int(input("Enter choice"))
if(ch==1):
   if(rear < mymax):
      item=input("enter item")
      enqueue(queue, item)
      rear = rear + 1
   else:
      print("Queue is full")
elif(ch==2):
   print(dequeue(queue))
elif(ch==3):
   print(queue)
else:
   break
```

**Applications of Queues:**

1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

**DEQUE(Double Ended Queue):**

A **double-ended queue** (**dequeue**, often abbreviated to **deque**, pronounced *deck*) generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).It is also often called a **head-tail linked list.** Like an ordinary queue, a double-ended queue is a data structure it supports the following operations: enq_front, enq_back, deq_front, deq_back, and empty. Dequeue can be behave like a queue by using only enq_front and deq_front , and behaves like a stack by using only enq_front and deq_rear.
The DeQueue is represented as follows.




DEQUE can be represented in two ways they are

1) Input restricted DEQUE(IRD)
2) output restricted DEQUE(ORD)

The output restricted DEQUE allows deletions from only one end and input restricted DEQUE allow insertions at only one end. The DEQUE can be constructed in two ways they are

1) Using array

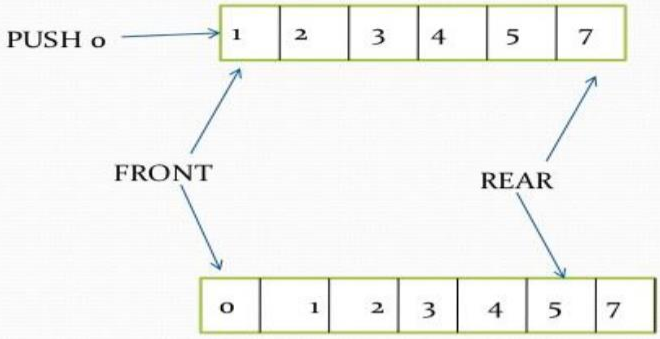
2)Using linked list

**Operations in DEQUE**

1. Insert element at back
2. Insert element at front
3. Remove element at front
4. Remove element at back

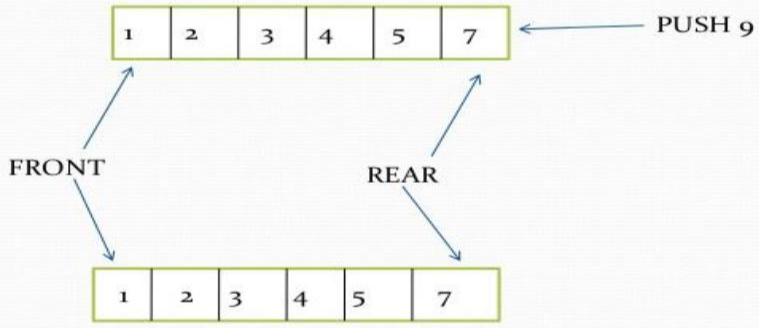
## Insert_front

- insert_front() is a operation used to push an element into the front of the *Deque*.



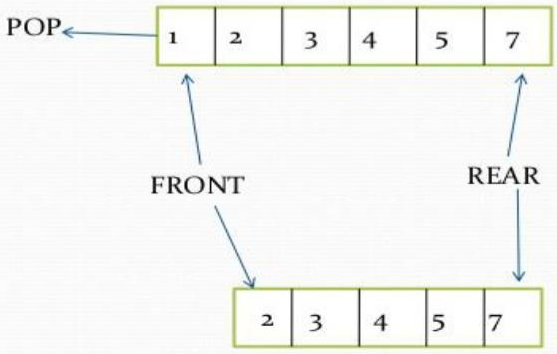

## Insert_back

- insert_back() is a operation used to push an element at the back of a *Deque*.



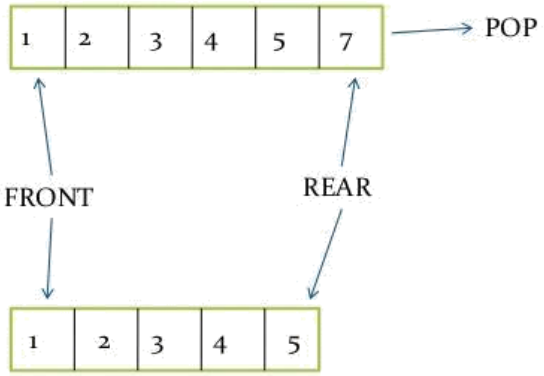

## Remove_front

- remove_front() is a operation used to pop an element on front of the *Deque*.



# Remove_back

• remove_front() is a operation used to pop an element on front of the *Deque*.




**Applications of DEQUE:**

1. The A-Steal algorithm implements task scheduling for several processors (multiprocessor scheduling).
2. The processor gets the first element from the deque.
3. When one of the processor completes execution of its own threads it can steal a thread from another processor.
4. It gets the last element from the deque of another processor and executes it.

**Circular Queue:**

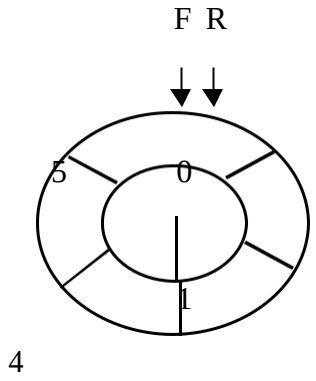
Circular queue is a linear data structure. It follows FIFO principle. In circular queue the last node is connected back to the first node to make a circle.

- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as “Ring buffer”.
- Items can inserted and deleted from a queue in O(1) time.

1. Using single linked list
2. Using double linked list
3. Using arrays

**Representation of Circular Queue:**

Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



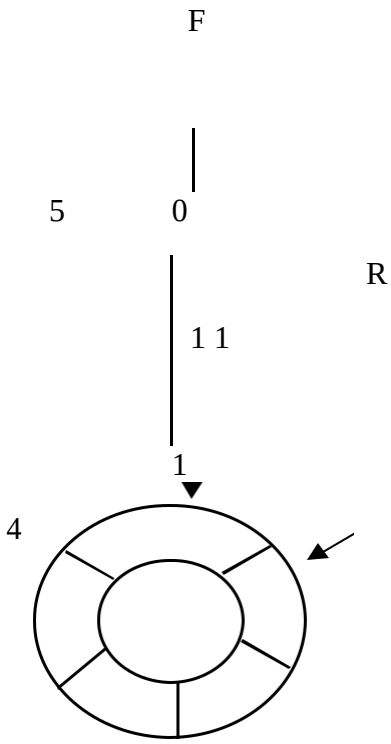

Q u e u e E
mp t y

MAX=6

FRONT = REAR = 0

COUNT = 0

Now, insert 11 to the circular queue. Then circular queue status will be:




FRONT = 0

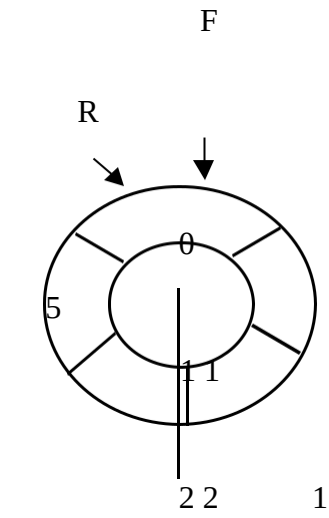
REAR =(REAR +1) %6=1

COUNT = 1

3 2

Circular Queue

Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:




FRONT = 0, REAR = 5

REAR =REAR %6 =5

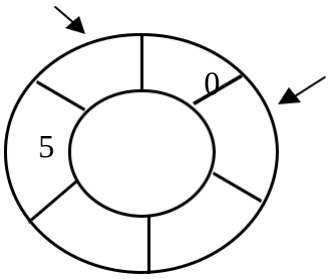
COUNT = 5

3 2

Circular Queue

Now, delete an element. The element deleted is the element at the front of the circular queue. So, 11 is deleted. The circular queue status is as follows:

R




F

FRONT = (FRONT + 1) % 6 =
1 REAR = 5

2 2 1

COUNT = COUNT - 1 = 4

4 5 5

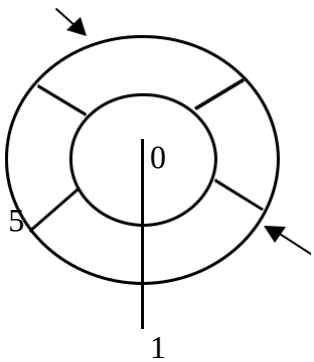
44 33

3 2

Circular Queue

Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:

R

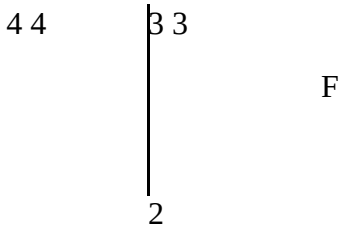



FRONT = (FRONT + 1) % 6
= 2

4 5 5

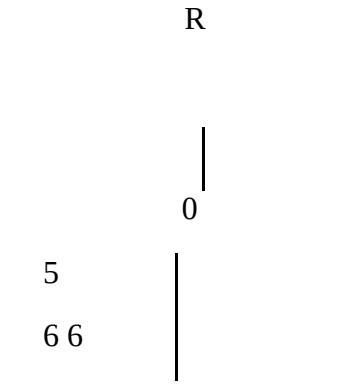
REAR = 5

COUNT = COUNT - 1 = 3




C irc u lar Q u e u
e

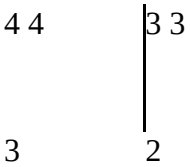
Again, insert another element 66 to the circular queue. The status of the circular queue is:



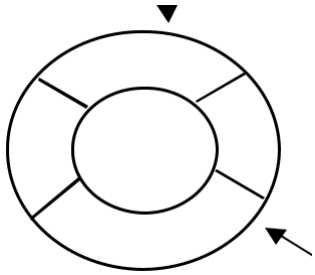

FRONT = 2

REAR =(REAR +1) %6=0

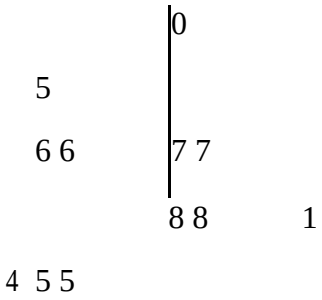
COUNT = COUNT + 1 = 4



C irc u lar Q u e
u e



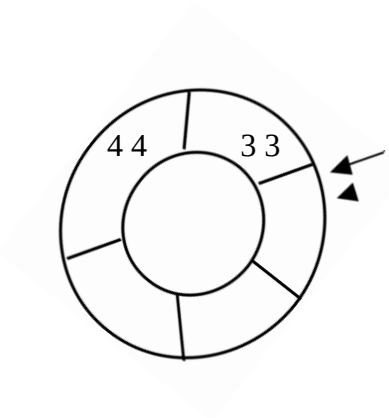
Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:




FRONT = 2, REAR = 2

REAR=REAR%6=2

COUNT = 6




C irc u lar Q u e u e

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

## Difference Between Stack and Queue

| **Parameter** | **Stack Data Structure** | **Queue Data Structure** |
|---|---|---|
| Basics | It is a linear data structure. The objects are removed or inserted at the same end. | It is also a linear data structure. The objects are removed and inserted from two different ends. |
| Working Principle | It follows the Last In, First Out (LIFO) principle. It means that the last inserted element gets deleted at first. | It follows the First In, First Out (FIFO) principle. It means that the first added element gets removed first from the list. |
| Pointers | It has only one pointer- the **top**. This pointer indicates the address of the topmost element or the last inserted one of the stack. | It uses two pointers (in a simple queue) for reading and writing data from both the ends- the **front** and the **rear**. The rear one indicates the address of the last inserted element, whereas the front pointer indicates the address of the first inserted element in a queue. |
| Operations | Stack<br>uses **push** and **pop** as two of its operations. The pop operation functions to remove the element from the | Queue<br>uses **enqueue** and **dequeue** as two of its operations. The dequeue operation deletes the elements from the queue, and |

| | | |
|---|---|---|
| | list, while the push operation functions to insert the element in a list. | the enqueue operation inserts the elements in a queue. |
| Structure | Insertion and deletion of elements take place from one end only. It is called the top. | It uses two ends- front and rear. Insertion uses the rear end, and deletion uses the front end. |
| Full Condition Examination | When top== max-1, it means that the stack is full. | When rear==max-1, it means that the queue is full. |
| Empty Condition Examination | When top==-1, it indicates that the stack is empty. | When front = rear+1 or front== -1, it indicates that the queue is empty. |
| Variants | A Stack data structure does not have any types. | A Queue data structure has three types- circular queue, priority queue, and double-ended queue. |
| Visualization | You can visualize the Stack as a vertical collection. | You can visualize a Queue as a horizontal collection. |
| Implementation | The implementation is simpler in a Stack. | The implementation is comparatively more complex in a Queue than a stack. |

## Unit 5

# Trees: Introduction

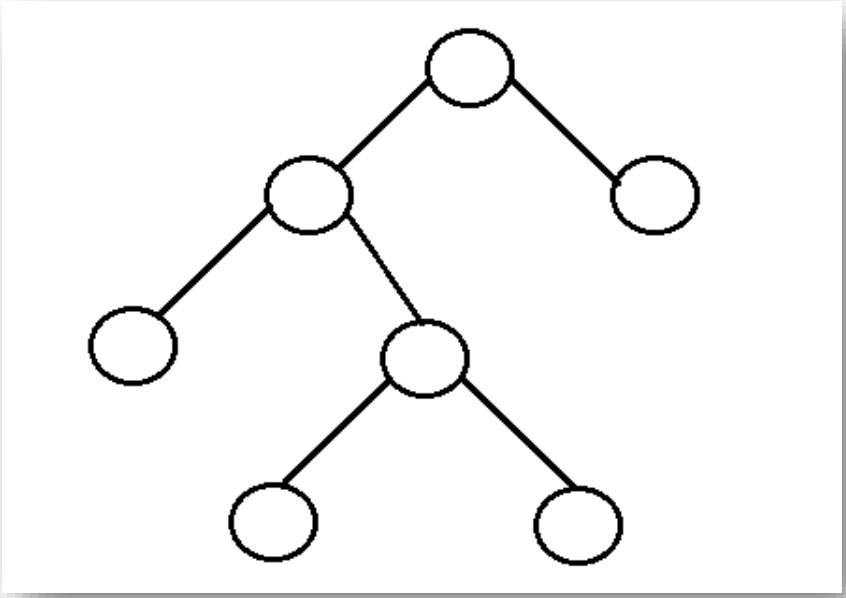
**Binary Tree**

A binary tree is an important class of a tree data structure in which a node can have at most two children. Child node in a binary tree on the left is termed as 'left child node' and node in the right is termed as the 'right child node.'

A binary tree may also be defined as follows:

- A binary tree is either an empty tree
- Or a binary tree consists of a node called the root node, a left subtree and a right subtree, both of which will act as a binary tree once again

.



**Binary Tree Representation in C:** A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL.
A Tree node contains following parts.
1. Data
2. Pointer to left child
3. Pointer to right child

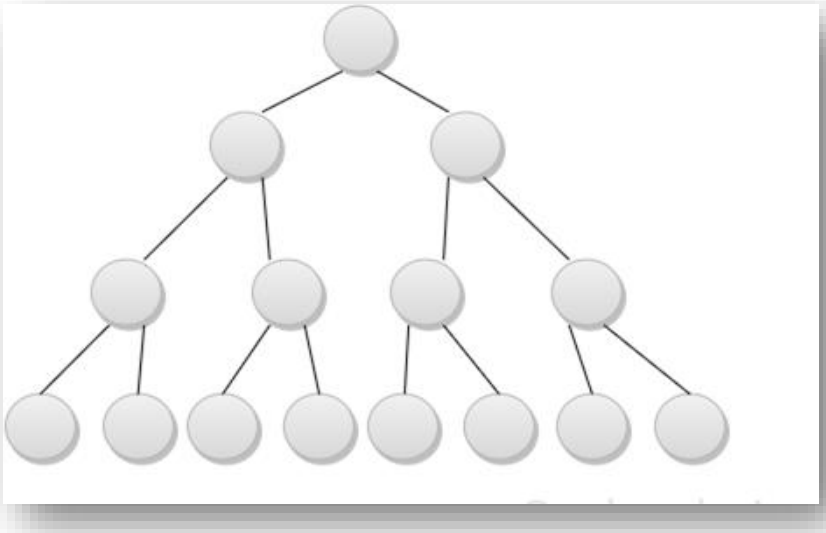
In C, we can represent a tree node using structures. For example a tree node with an integer data can be defined as:

```
struct node
{
  int data;
  struct node *left;
  struct node *right;
};
```

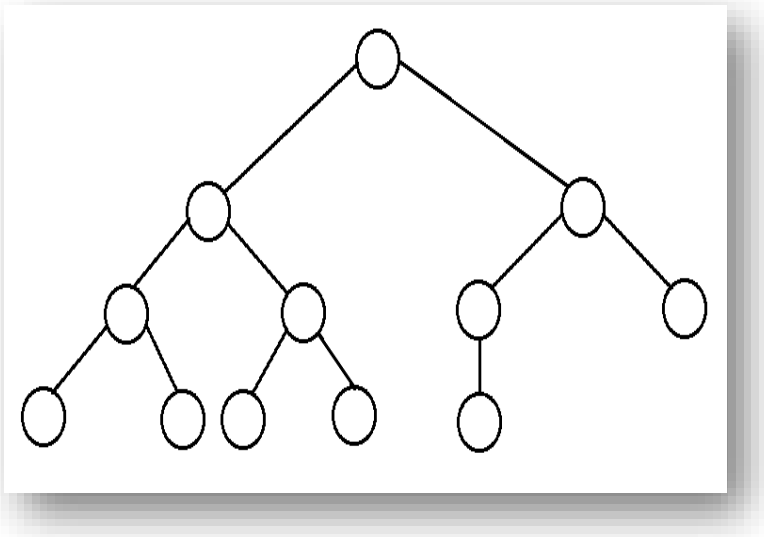
**Types of Binary Trees:**

Following are the types of Binary trees:

1. **Full Binary Tree:** A **full binary tree** is a tree in which all the nodes other than the leaves has exact two children. It is also called as proper or 2-tree.



2. **Complete Binary Tree:** A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible.

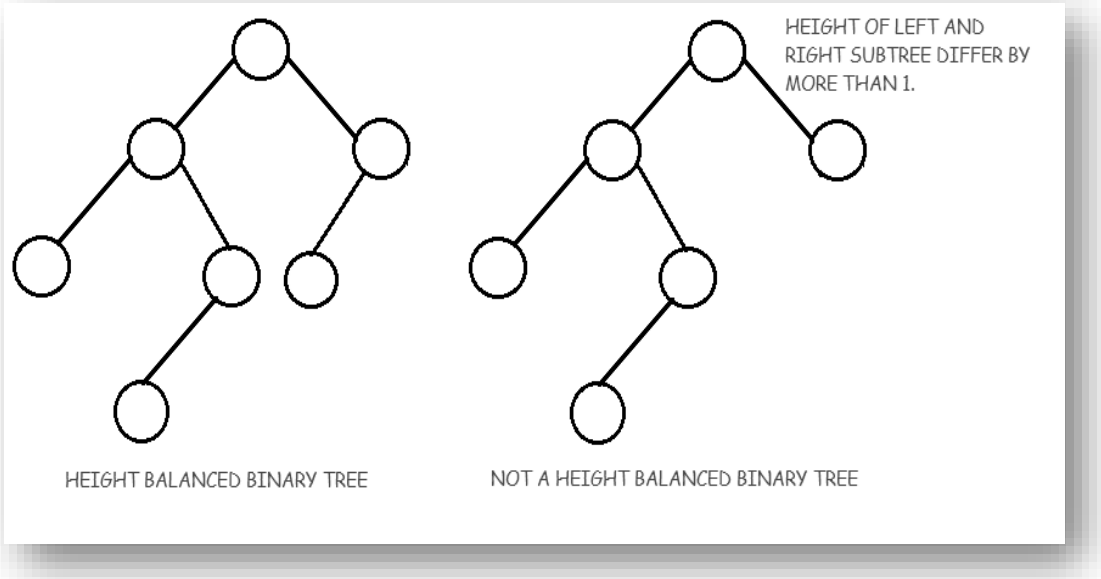


3. **Balanced binary tree:** A binary tree is height balanced if it satisfies the following conditions:

1) The left and right subtrees' heights differ by at most one, AND
2) The left subtree is balanced, AND
3) The right subtree is balanced

An empty tree is height balanced.

- The height of a balanced binary tree is O(*Log n*) where *n* is number of nodes.

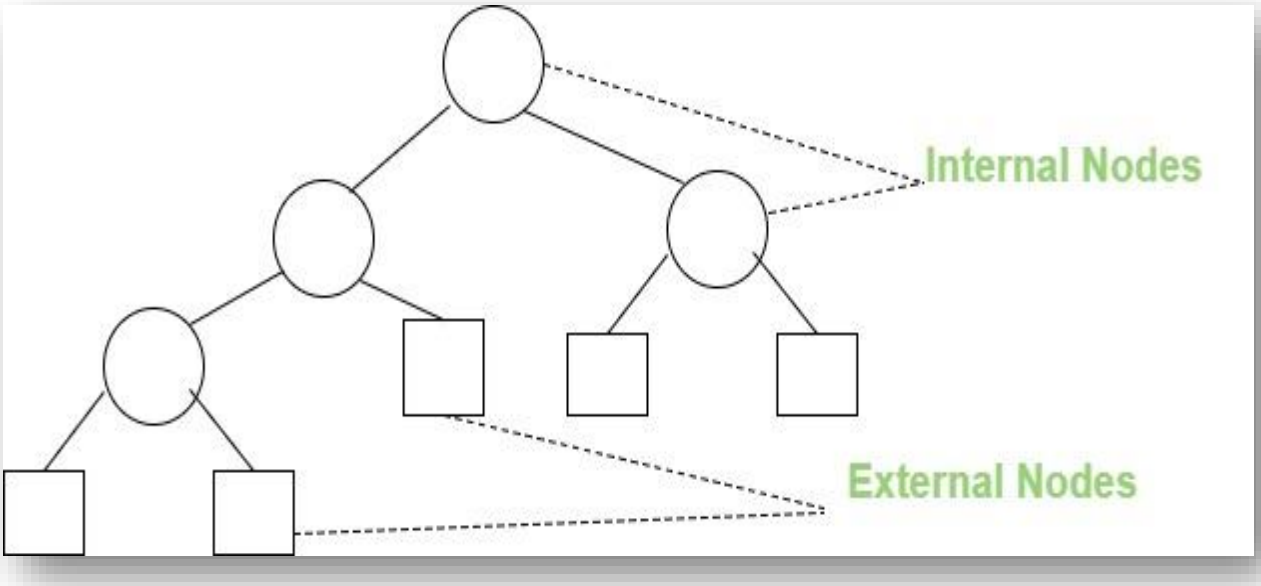


4. **Extended Binary Tree:** A binary tree can be converted into an extended binary tree by adding new nodes to its leaf nodes and to the nodes that have only one child. These new nodes are added in such a way that all the nodes in the resultant tree have either zero or two children. The nodes with two children are called Internal node and the nodes with zero child are called External node.

   The Internal node are represented by circle and External nodes are represented by box.

   Also External node path is always greater than one by Internal node path i.e $[N_E] = [N_i] + 1$

   The original tree contains only internal nodes(Circles) and External Nodes(Squares) are the nodes added to make it complete.




**Representation of Binary Tree in Memory:**

Let T be a Binary Tree. There are two ways of representing T in the memory as follow:

1. Sequential Representation of Binary Tree.
2. Linked Representation of Binary Tree.

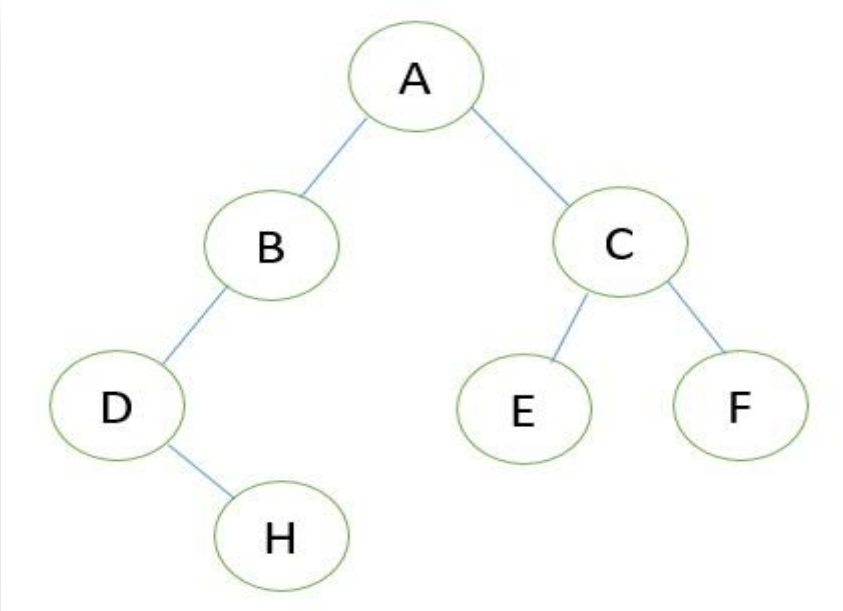
**Sequential Representation of Binary Tree:**

In sequential representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree. This representation uses only a linear array TREE as follows:

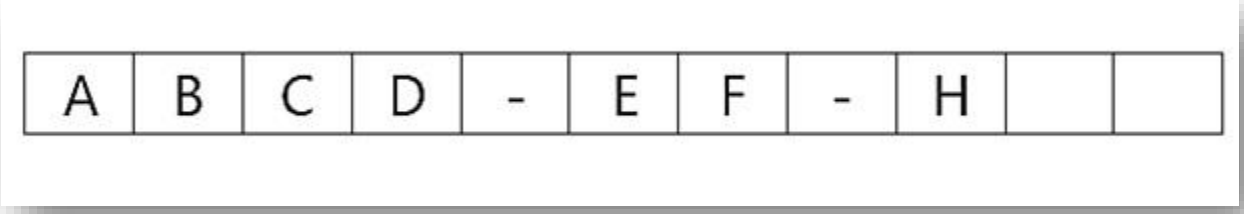
1. The root N of T is stored in TREE [1].
2. If a node occupies TREE [k] then its left child is stored in TREE [2 * k] and its right child is stored into TREE [2 * k + 1].

For Example:

Consider the following Tree




**Its sequential representation is as follow:**



**Linked Representation of Binary Tree:**

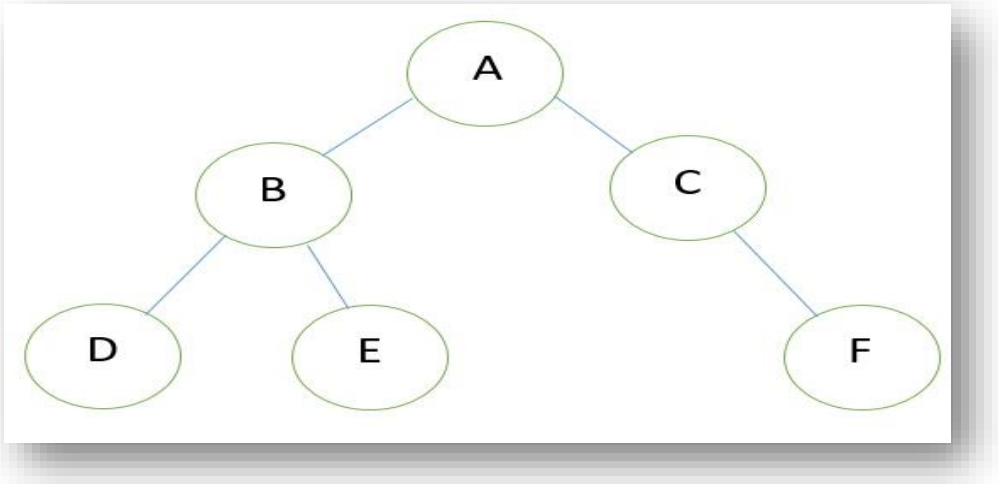
We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.
In this linked list representation, a node has the following structure...


In this representation of binary tree root will contain the location of the root R of T. If any one of the subtree is empty, then the corresponding pointer will contain the null value if the tree T itself is empty, the ROOT will contain the null value.

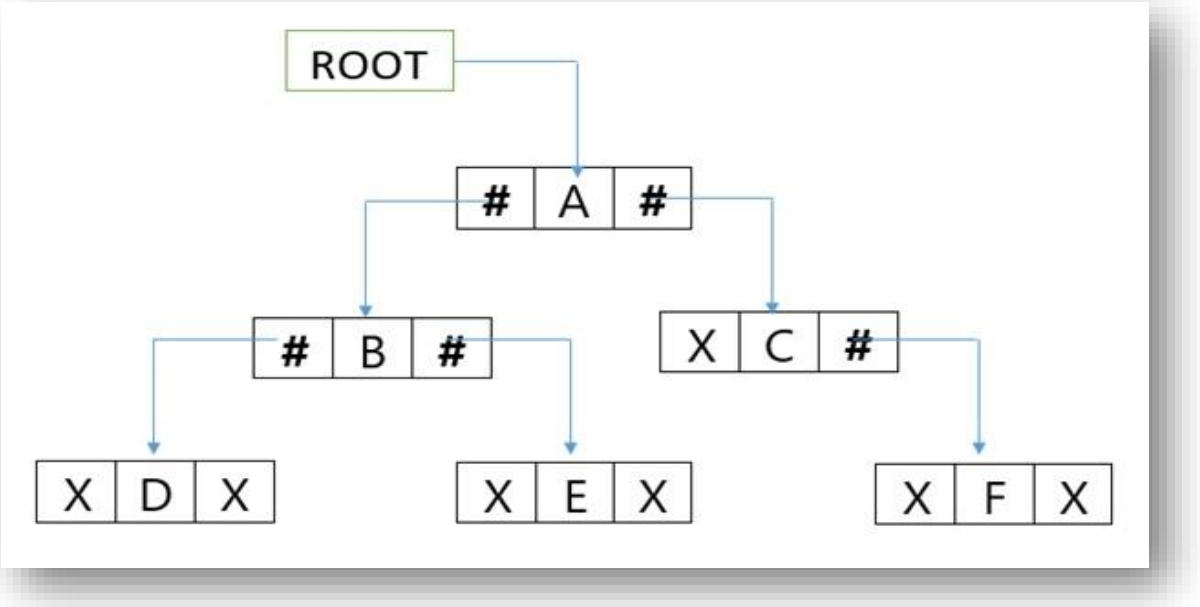
Example

Consider the binary tree T given below. In the linked representation of T, each node is represented with three fields, and that the empty subtree is pictured by using x for null entries.

**Binary Tree**



**Linked Representation of the Binary Tree**




**Binary Tree Traversal**

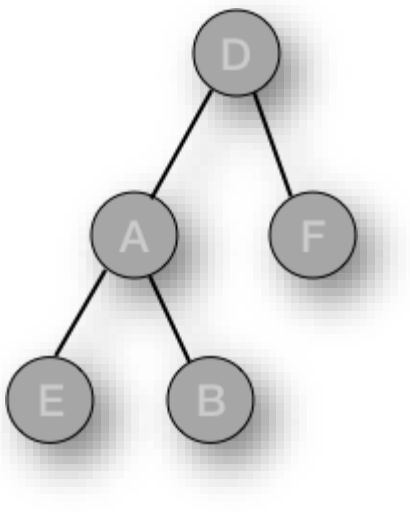
**Tree traversal** is the process of visiting each node in the tree exactly once. Visiting each node in a tree should be done in a systematic manner. There are basically three traversal techniques for a binary tree that are,

1. In-order traversal
2. Pre-order traversal
3. Post-order traversal

**In-order Traversal**

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. It is made clear that every node may represent a subtree itself.

For example, given below is a tree T




The In-order traversal for this tree is:

E-A-B-D-F

The recursive algorithm for In-order Traversal can be written as:

```
void inorder(struct node *root)
{
   if(root!=NULL) // checking if the root is not null
   {
      inorder(root->left_child); // visiting left child
      printf(" %c ", root->data); // printing data at root
      inorder(root->right_child);// visiting right child
   }
}
```

**Pre-order traversal:**

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

The pre-order traversal for tree T is:

D -A-E-B-F

The recursive algorithm for pre-order Traversal can be written as:

```
void preorder(struct node *root)
{
   if(root!=NULL) // checking if the root is not null
   {
      printf(" %c ", root->data); // printing data at root
      preorder(root->left_child); // visiting left child
      preorder(root->right_child); // visiting right child
   }
}
```

**Post-Order Traversal:**

In this traversal method, the root node is visited last. First we traverse the left subtree, then the right subtree and finally the root node.

The post-order traversal for tree T is:

E-B-A-F-D

The recursive algorithm for Post-order Traversal can be written as:

```
void postorder(struct node *root)
{
   if(root!=NULL) // checking if the root is not null
   {
      postorder(root->left_child); // visiting left child
      postorder(root->right_child); // visiting right child
      printf(" %c ", root->data); // printing data at root
```

```
    }

}
```

**Construct Tree from given Inorder and Preorder traversals**

Let us consider the below traversals:

Inorder sequence: D B E A F C
Preorder sequence: A B D E C F

We know that in a Preorder sequence, leftmost element is the root of the tree. So 'A' is root for given sequences. By searching 'A' in Inorder sequence, we can find out all elements on left side of 'A' i.e DBE are in left subtree and elements on right i.e FC are in right subtree. So we know below structure now.

```
          A
        /   \
       /     \
     D B E   F C
```

Now for the left subtree, DBE, search in preorder, B comes first. So B is the next root. Checking Inorder, D is to the left of B and E to the right of B.

For right subtree, first go to preorder sequence, C comes before F so C is root and from inorder traversal it is clear that F is to the left of C

Now, we can build the tree as:

```
      A
    /   \
   /     \
  B       C
 / \     /
/   \   /
D   E   F
```

The algorithm for building tree from preorder and Inorder traversal can be written as:

**Algorithm:buildTree**()

1) Pick an element from Preorder. Increment a Preorder Index Variable to pick next element in

next recursive call.
2) Create a new tree node tNode with the data as picked element.
3) Find the picked element's index in Inorder. Let the index be inIndex.
4) Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.
5) Call buildTree for elements after inIndex and make the built tree as right subtree of tNode.
6) return tNode.

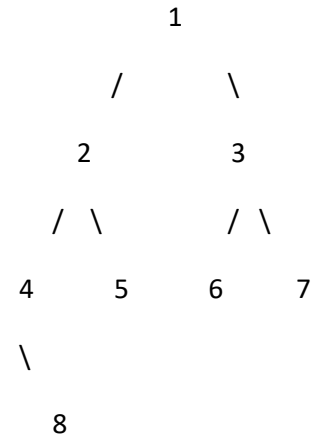
**Construct Tree from given Inorder and Postorder traversals:**

Let us consider the below traversals:

Inorder Sequence: 4, 8, 2, 5, 1, 6, 3, 7

Postorder Sequence: 8, 4, 5, 2, 6, 7, 3, 1

- Last element in the postorder sequence will be the root of the tree, here it is 1.
- Now search element 1 in inorder, suppose we find it at position i, we find it, make note of elements which are left to i (this will construct the left subtree) and elements which are right to i ( this will construct the right Subtree).
- Suppose in previous step, there are X number of elements which are left of 'i' (which will construct the left subtree), take first X elements from the postorder[] traversal, this will be the post order traversal for elements which are left to i. similarly if there are Y number of elements which are right of 'i' (which will construct the righ tsubtree), take next Y elements, after X elements from the postorder[] traversal, this will be the post order traversal for elements which are right to i
- From previous two steps construct the left and right subtree and link it to root.left and root.right respectively.



**A Binary Search Tree (BST)**

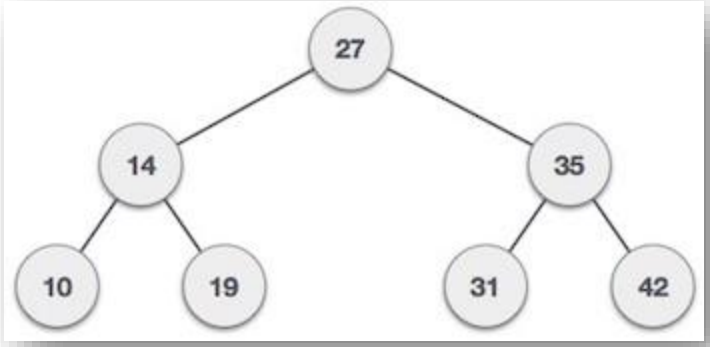
A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

For example, the tree given below is a binary search tree:




Here, the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

**Basic Operations**

Following are the basic operations of a tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Deletion**: Deletes an element from tree
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

**Searching in a BST:**

Searching in binary search tree starts with the root node. Following steps are performed to search a key in a BST:

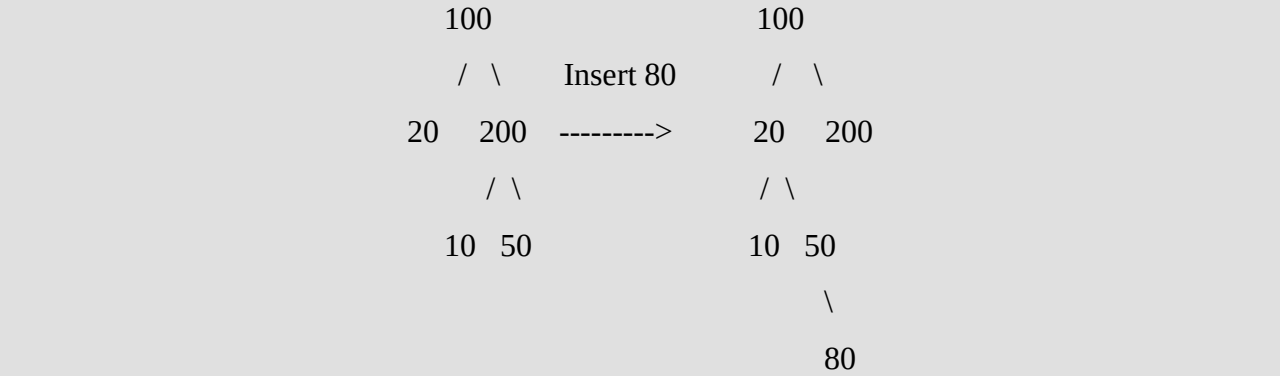
- Compare the given key with the root. If key matches root, the search is successful and return.
- If key is less than root, then search left sub tree recursively.
- If key is greater than root, then search right sub tree recursively.

E.g. If we want to search 31, following steps will be performed:

- Compare 31 with root i.e 27. 31>27, then search right sub tree.
- In right sub tree, 35 is the next node, 31<35. So, go to left sub tree of 35.
- In left sub tree, 31 is the next node. 31=31. So search is successful.

**Insertion in BST:**

A new element is always inserted at leaf. Whenever an element is to be inserted, start searching from the root node, then if element to be inserted is less than the value of root node, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.



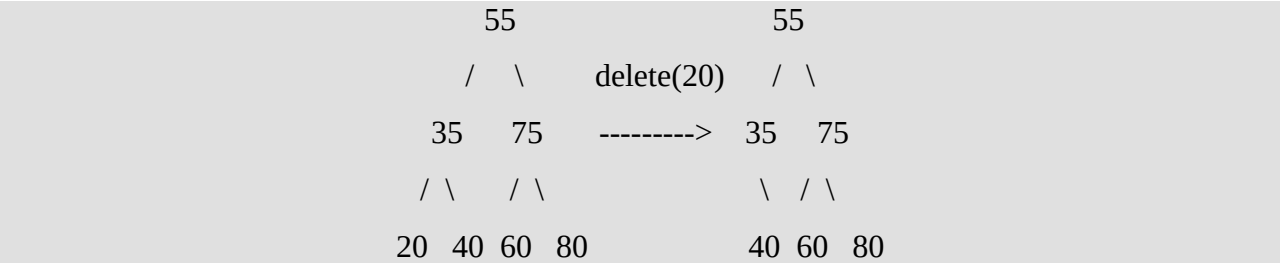

In the example above, we have to insert 80 in the existing tree. The first step is to find location to insert 80. Always keep in mind that the condition of BST should not be violated after insertion.

- Start with root(100). 100>80. So go to left subtree.
- In left subtree, 20 is the next node. As 20<80. So go the right of 20.
- In right subtree, 50 is the next node. As 50<80. So go the right of 50
- In the right of 50, no other node exists. So we have found location to insert 80(i.e to the right of 50.
- Insert 80.

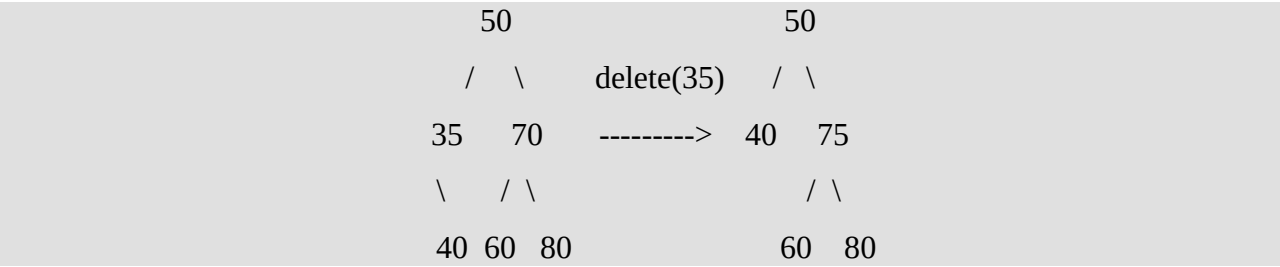
**Similarly you can practice insertion of 180.(will be to the left of 200)**

**Deletion of an element from BST:** When we delete a node, three possibilities arise. These are as follows:

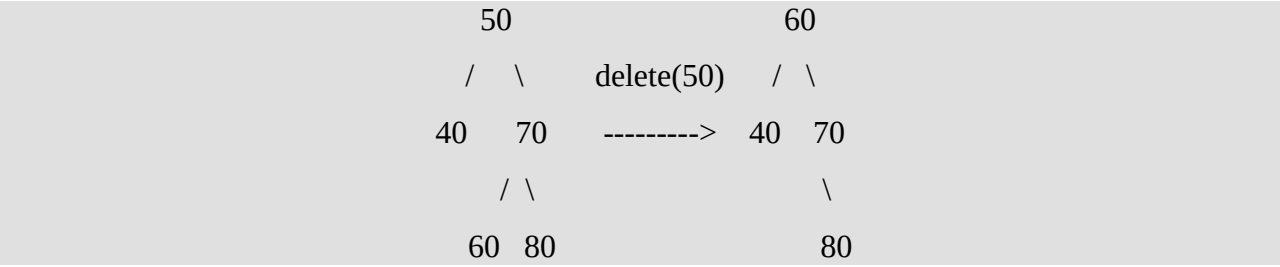
1. **Node to be deleted is leaf:** Simply remove from the tree. For example to delete 20 from the tree, simply remove it as it is leaf.




2. **Node to be deleted has only one child:** Copy the child to the node and delete the child. For example, to delete 35 from the tree, copy value of its only child(i.e 40) to it.




3. **Node to be deleted has two children**: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.




**BST Traversal:** A binary search tree can be traversed in same way as a simple binary tree (preorder, postorder and inorder).

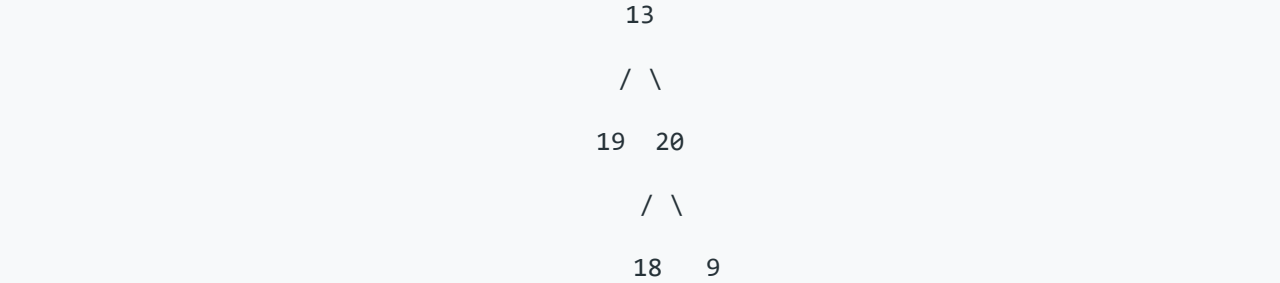
- Inorder traversal of a binary tree always gives a sorted list.
- While constructing a BST, we only need either a preorder or postorder traversal because after sorting the given sequence we automatically get inorder sequence.

**Balanced Binary Tree:**

A balanced binary tree is a binary tree structure in which the left and right subtrees of every node differ in height by no more than 1.

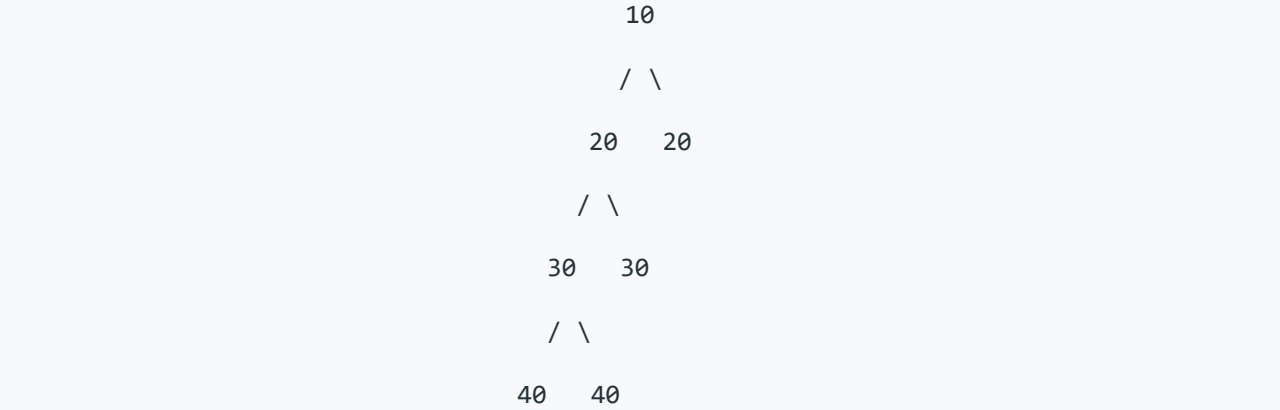
For example:

Given the following tree [13,19,20,null,null,18,9]:




It is a Balanced Binary tree because left and right subtrees ‘ height differ by 1 only.

Given the following tree [10,20,20,30,30,null,null,40,40]:




It is not a balanced binary tree because of height difference in left and right subtree is more than 1.

## Unit 6

## Sorting and Searching

### Searching

**Introduction**: Searching is a technique that finds whether a given element or value exists in the list or not. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:

- Linear Search or Sequential Search
- Binary Search

**Linear Search**: Linear Search Algorithm is the simplest search algorithm. In this search algorithm a sequential search is made over all the items one by one to search for the targeted item. Each item is checked in sequence until the match is found. If the match is found, particular item is returned otherwise the search continues till the end.

**Algorithm**

LinearSearch ( Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

**Example:**

Let us take an example of an array A[7]={9,2,4,6,3,7,11}. Array A has 7 items. Let us assume we are looking for 7 in the array. Targeted item=7.

Here, we have

A[7]={9,2,4,6,3,7,11}

X=7

At first, When i=0 (A[0]=9; X=7) not matched

i++ now, i=1 (A[1]=2; X=7) not matched

i++ now, i=2(A[2])=4; X=7)not matched

…….

i++ when, i=5(A[5]=7; X=7) Match Found

Hence, Element X=7 found at index 5.

**Binary Search:**

Binary search is a fast search algorithm. This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

**Algorithm**

```
Algorithm Binary search(B, M, S):
Step 1: [Initialize]
Low=1
High=M
Step 2: [Perform search]
Repeat thru step 4 while LOW<=HIGH
Step 3: [Obtain index of midpoint of interval]
MIDDLE=(LOW+HIGH)/2
Step 4: [Compare]
```

```
If S<B [MIDDLE]
Then HIGH=MIDDLE-1
Else if S>B [MIDDLE]
Then LOW=MIDDLE+1
Else Write('SUCCESSFULSEARCH')
Return (MIDDLE)
Step 5: [Unsuccessful search]
Write('UNSUCCESSFULSEARCH')
Return(0)
```

**Binary Search Example**

Let us take an example of an array A[16]={1,2,3,4,5,6,8,10,12,13,15,16,18,19,20,22}. The array is sorted and contains 16 items. We are looking for item 19 in this list.

Here, we have

A[16]={1,2,3,4,5,6,8,10,12,13,15,16,18,19,20,22}

X=19

low=0; high=15; mid=int((0+15)/2)=7

A[7]=10

As 19 is greater than the middle numbers that means we don't require to look for the targeted item in first sub array. Let us search in the second subarray A2={12,13,15,16,18,19,20,22}

low=8; high=15 in Array A2

Mid=int((8+15)/2)=11

A[11]=16

As 19 is greater than 16, we don't need to look in the first part of this subarray. A21={12,13,15 } and A22={18,19,20,22}

low=12; high=15; mid=int((12+15)2)=13

A[13]=19

Hence 19 is found and location 13 is returned.

**Difference between Linear and Binary Search**

| PARAMETER | LINEAR SEARCH | BINARY SEARCH |
|---|---|---|
| **Time Complexity** | The formula can be written as **O(N)** | **O(log 2 N)** is the formula that can be followed for this search |
| **Sequential** | Linear search is led by sequence; it starts from the first point and ends at the last point. | The binary search begins from the middle point. |
| **The most compelling case time** | The first Element serves to the most appropriate case time | Centre Element is the most relevant case time in this search. |
| **Prerequisites needed for an array** | Not needed | The array must be in sorted order |
| **The worst-case scenario for n elements** | N number of comparisons will be needed | The conclusion can be derived only after $\log_2 N$ comparisons |
| **Capable of being implemented on** | Linked lists and arrays | Incapable of being implemented directly on linked lists |
| **Algorithm type** | Iterative characteristics depicted | Characteristics of **"divide and conquer"** features are described. |
| **Utility** | Easy to decipher and apply. There is no need for ordered elements. | The algorithms are tricky to understand and apply. The elements have to be organized in the proper manner and order. |
| **Lines of Coding** | Less | More |

# Sorting

## Introduction

Sorting is a process of ordering or placing a list of elements in some kind of order. It is nothing but storage of data in sorted order. Sorting can be done in ascending and descending order. It arranges the data in a sequence which makes searching easier.

## Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques:.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

## Bubble Sort Algorithm

**Bubble Sort** is a simple algorithm which is used to sort a given set of n elements provided in the form of an array. Bubble Sort compares all elements one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

If we have total n elements, then we need to repeat this process for n-1 times.

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

## Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Let's consider an array with values {5, 1, 4, 2, 8}

Below, we have an explanation of how bubble sort will sort the given array.

**First Pass:**

( **5 1** 4 2 8 ) –> ( **1 5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.

( 1 **5 4** 2 8 ) –> ( 1 **4 5** 2 8 ), Swap since 5 > 4

( 1 4 **5 2** 8 ) –> ( 1 4 **2 5** 8 ), Swap since 5 > 2

( 1 4 2 **5 8** ) –> ( 1 4 2 **5 8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second Pass:**

( **1 4** 2 5 8 ) –> ( **1 4** 2 5 8 )

( 1 **4 2** 5 8 ) –> ( 1 **2 4** 5 8 ), Swap since 4 > 2

( 1 2 **4 5** 8 ) –> ( 1 2 **4 5** 8 )

( 1 2 4 **5 8** ) –> ( 1 2 4 **5 8** )

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

**Third Pass:**

( **1 2** 4 5 8 ) –> ( **1 2** 4 5 8 )

( 1 **2 4** 5 8 ) –> ( 1 **2 4** 5 8 )

( 1 2 **4 5** 8 ) –> ( 1 2 **4 5** 8 )

( 1 2 4 **5 8** ) –> ( 1 2 4 **5 8** )

```
ALGORITHM BUBBLE_SORT(A,N)
STEP1: SET i=0;j=0;
Step2 : For i=0 to N-1 Repeat Step 3
Step3: For j=0 to N-1
          If A[j]>A[j+1]

       Swap(A[j],A[j+1])

       Endif

Step 4: Return A

Step 5: End
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| i = 0 | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| | 0 | 5 | 3 | 1 | 9 | 8 | 2 | 4 | 7 |
| | 1 | 3 | 5 | 1 | 9 | 8 | 2 | 4 | 7 |
| | 2 | 3 | 1 | 5 | 9 | 8 | 2 | 4 | 7 |
| | 3 | 3 | 1 | 5 | 9 | 8 | 2 | 4 | 7 |
| | 4 | 3 | 1 | 5 | 8 | 9 | 2 | 4 | 7 |
| | 5 | 3 | 1 | 5 | 8 | 2 | 9 | 4 | 7 |
| | 6 | 3 | 1 | 5 | 8 | 2 | 4 | 9 | 7 |
| i =1 | 0 | 3 | 1 | 5 | 8 | 2 | 4 | 7 | 9 |
| | 1 | 1 | 3 | 5 | 8 | 2 | 4 | 7 | |
| | 2 | 1 | 3 | 5 | 8 | 2 | 4 | 7 | |
| | 3 | 1 | 3 | 5 | 8 | 2 | 4 | 7 | |
| | 4 | 1 | 3 | 5 | 2 | 8 | 4 | 7 | |
| | 5 | 1 | 3 | 5 | 2 | 4 | 8 | 7 | |
| i = 2 | 0 | 1 | 3 | 5 | 2 | 4 | 7 | 8 | |
| | 1 | 1 | 3 | 5 | 2 | 4 | 7 | | |
| | 2 | 1 | 3 | 5 | 2 | 4 | 7 | | |
| | 3 | 1 | 3 | 2 | 5 | 4 | 7 | | |
| | 4 | 1 | 3 | 2 | 4 | 5 | 7 | | |
| i = 3 | 0 | 1 | 3 | 2 | 4 | 5 | 7 | | |
| | 1 | 1 | 3 | 2 | 4 | 5 | | | |
| | 2 | 1 | 2 | 3 | 4 | 5 | | | |
| | 3 | 1 | 2 | 3 | 4 | 5 | | | |
| i = 4 | 0 | 1 | 2 | 3 | 4 | 5 | | | |
| | 1 | 1 | 2 | 3 | 4 | | | | |
| | 2 | 1 | 2 | 3 | 4 | | | | |
| i = 5 | 0 | 1 | 2 | 3 | 4 | | | | |
| | 1 | 1 | 2 | 3 | | | | | |
| i = 6 | 0 | 1 | 2 | 3 | | | | | |
| | | 1 | 2 | | | | | | |

**Optimized Bubble Sort Algorithm**

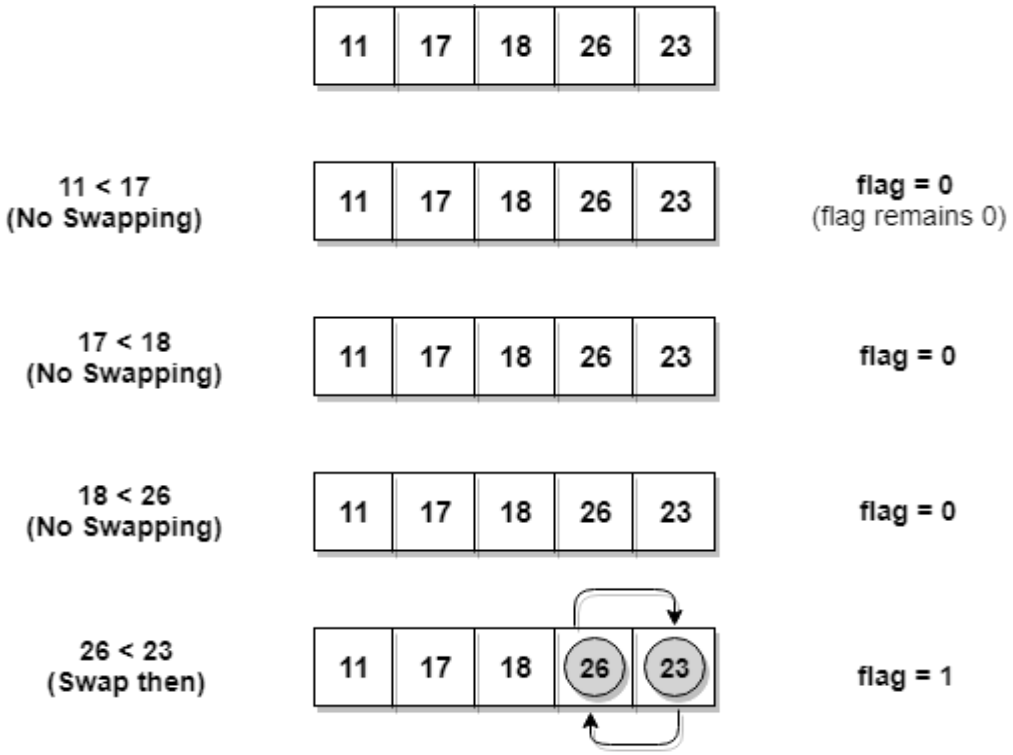
To optimize our bubble sort algorithm, we can introduce a flag to monitor whether elements are getting swapped inside the inner for loop.

Hence, in the inner for loop, we check whether swapping of elements is taking place or not, everytime.

If for a particular iteration, no swapping took place, it means the array has been sorted and we can jump out of the for loop, instead of executing all the iterations.

Let's consider an array with values {11, 17, 18, 26, 23}

Below, we have a pictorial representation of how the optimized bubble sort will sort the given array.




As we can see, in the first iteration, swapping took place, hence we updated our flag value to 1, as a result, the execution enters the for loop again. But in the second iteration, no swapping will occur, hence the value of flag will remain 0, and execution will break out of loop.

ALGORITHM FOR OPTIMISED BUBBLE SORT

```
Algorithm BUBBLE_OPTIM(A,N)

STEP1: SET i=0;j=0;

Step2 : For i=0 to N-1 Repeat Step 3 to 5

Step 3: Flag=0
Step4: For j=0 to N-1
          If A[j]>A[j+1]

         Swap(A[j],A[j+1])

        Flag=1

        Endif

Step 5: If Flag=0

            Return A

            Exit
```

Endif

Step6: Return A

Step 7: End

**Insertion Sort**

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration. This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

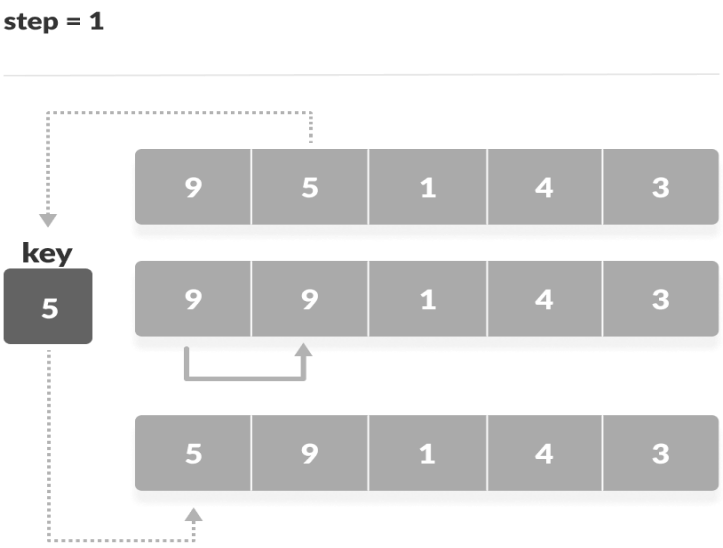
**How Insertion Sort Works?**

Suppose we need to sort the following array.


Initial array

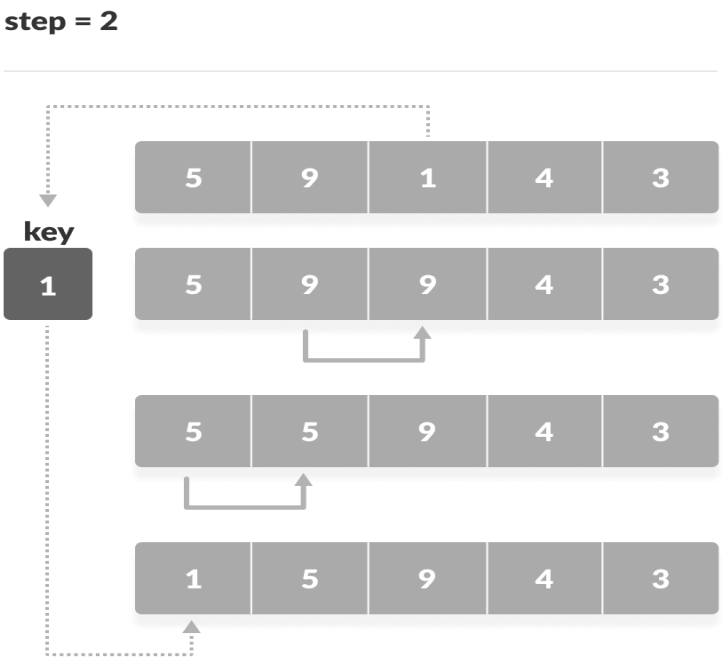
The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element



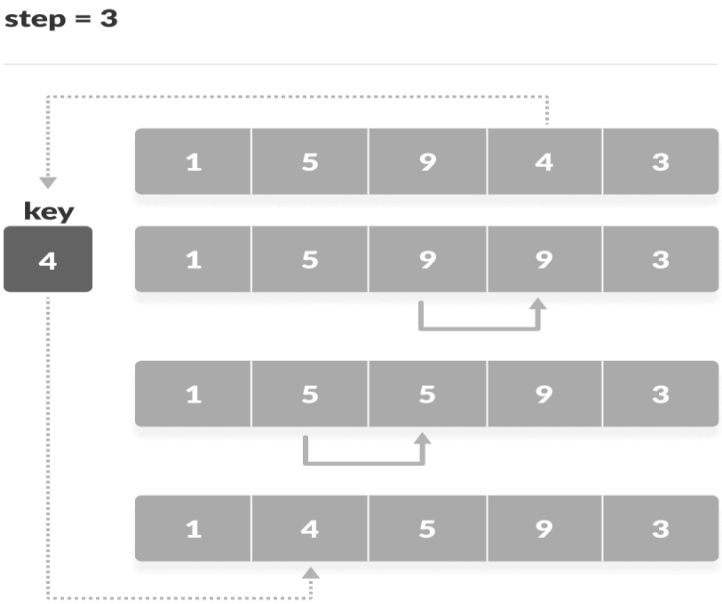

Now, the first two elements are sorted.

Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array

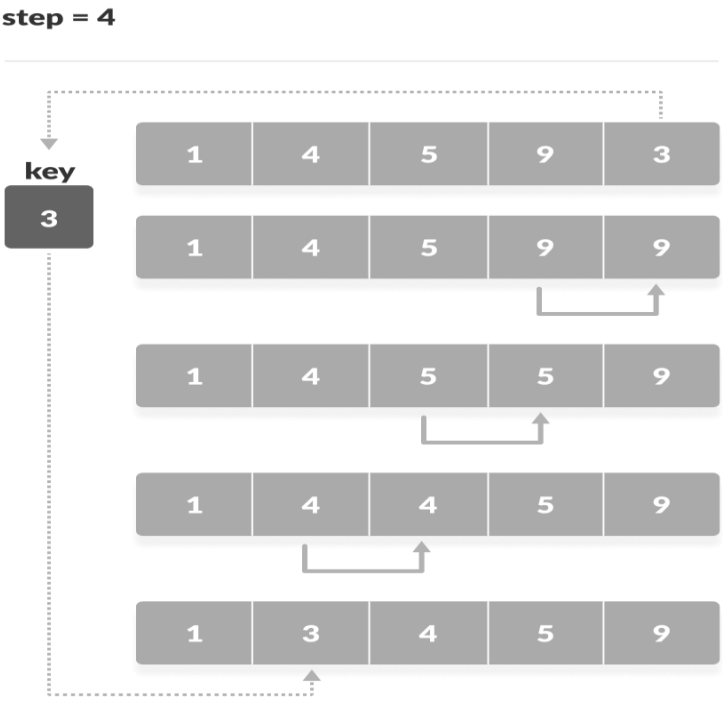



Place 1 at the beginning

Similarly, place every unsorted element at its correct position.




Place 4 behind 1




Place 3 behind 1 and the array is sorted

**Another Example:**

**12**, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 4 (last element of the array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12
**11, 12**, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13
**11, 12, 13**, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.
**5, 11, 12, 13**, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.
**5, 6, 11, 12, 13**

Algorithm For Insertion Sort

**Step 1** − If it is the first element, it is already sorted. return 1;
**Step 2** − Pick next element
**Step 3** − Compare with all elements in the sorted sub-list
**Step 4** − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
**Step 5** − Insert the value
**Step 6** − Repeat until list is sorted

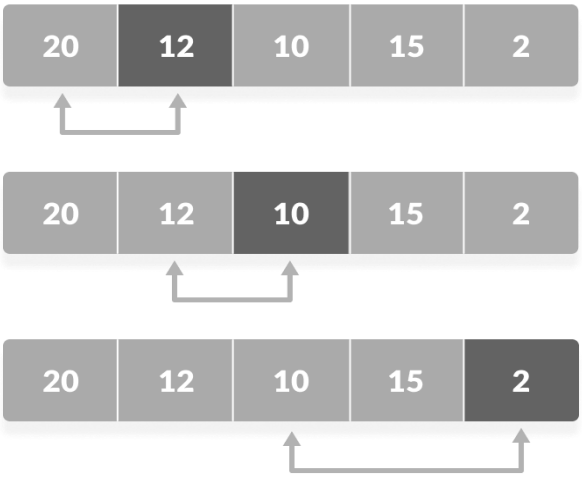
**Selection Sort:**

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list. The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

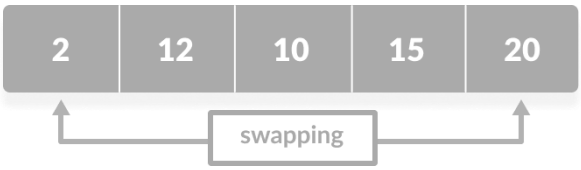
Set the first element as minimum.


Compare minimum with the second element. If second element is smaller than minimum, assign second element as minimum.

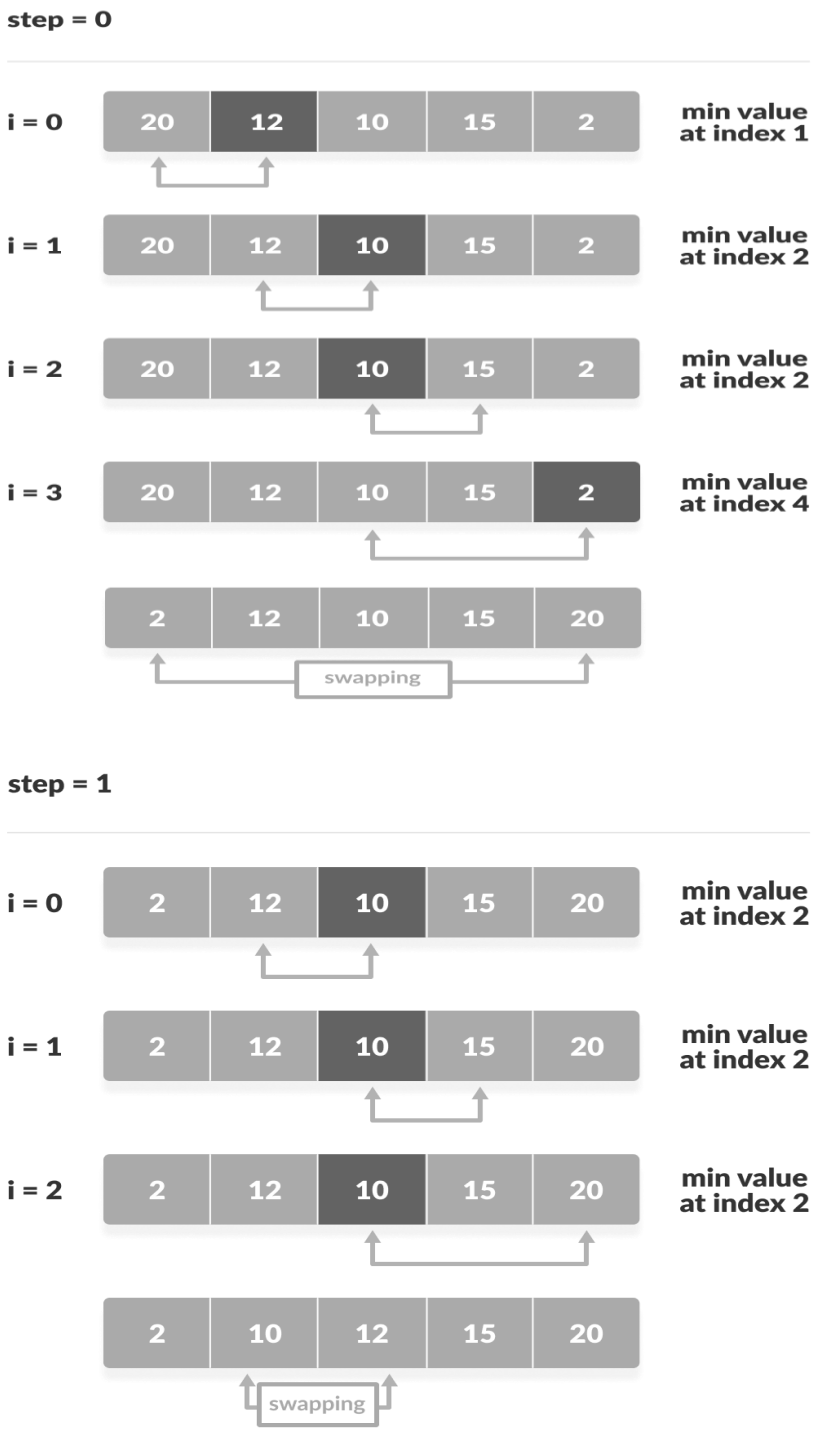
Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

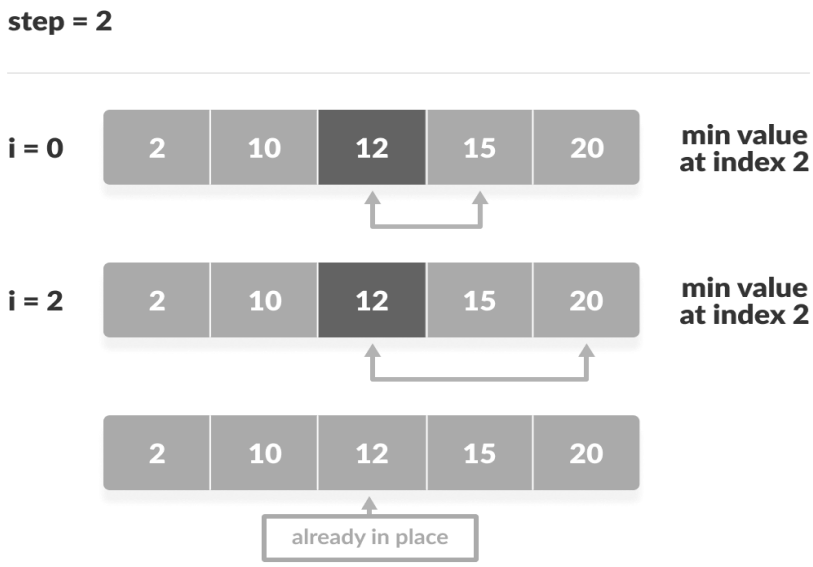



After each iteration, minimum is placed in the front of the unsorted list.



For each iteration, indexing starts from the first unsorted element.







```
Algorithm selectionsort(A,N)

  for i = 1 to n - 1
  /* set current element as minimum*/
    min = i

    /* check the element to be minimum */
    for j = i+1 to i
      if A[j] < A[min] then
        min = j;
      end if
    end for

    /* swap the minimum element with the current element*/
```

```
    if indexMin != i  then
      swap A[min] and A[i]
    end if
  end for

end
```

**Merge Sort:**

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into sorted list.

Here's how merge sort uses divide-and-conquer:

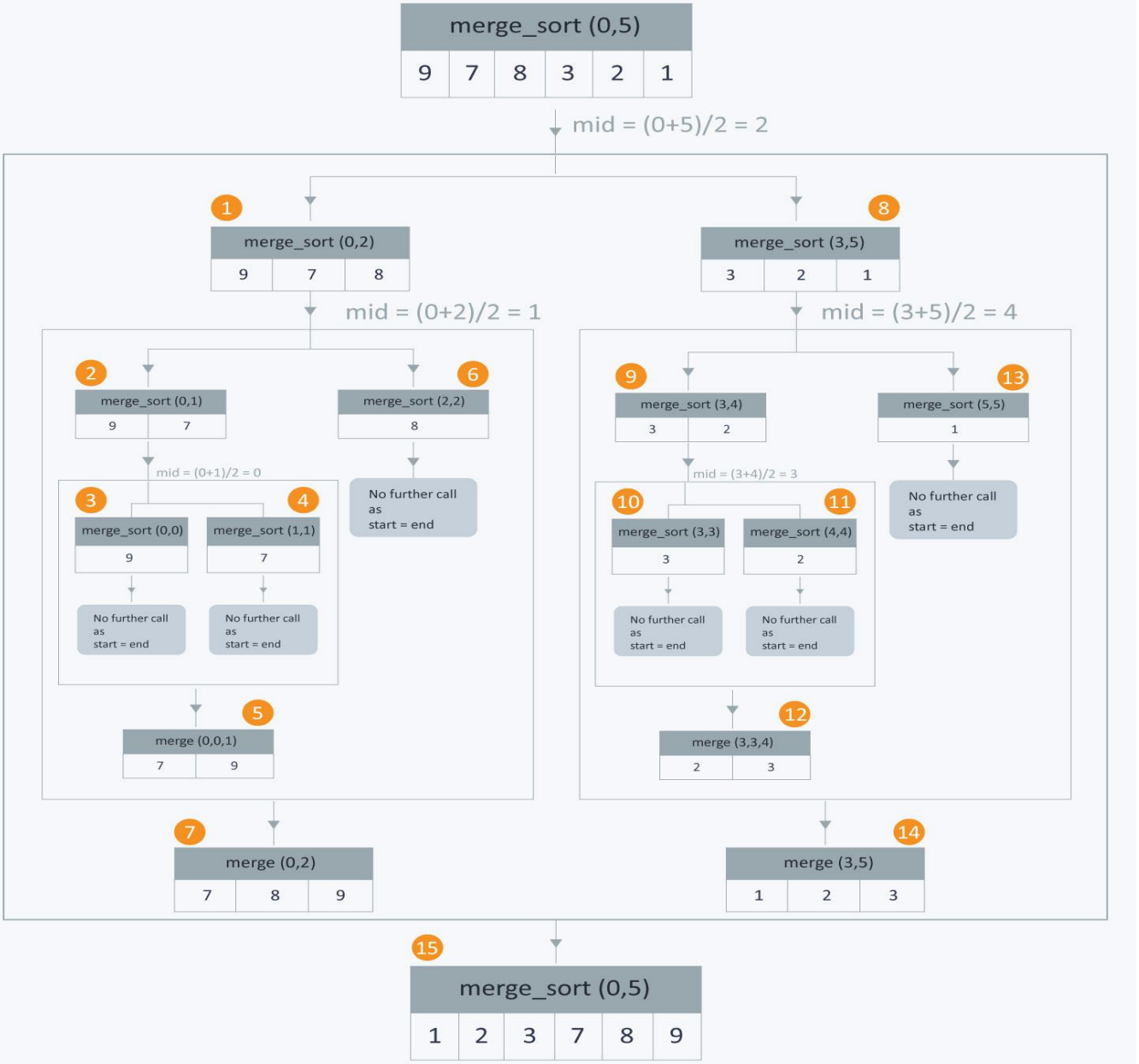
1. **Divide** by finding the number q of the position midway between p and r..
2. **Conquer** by recursively sorting the subarrays in each of the two subproblems created by the divide step. That is, recursively sort the subarray array[p..q] and recursively sort the subarray array[q+1..r].
3. **Combine** by merging the two sorted subarrays back into the single sorted subarray array[p..r].

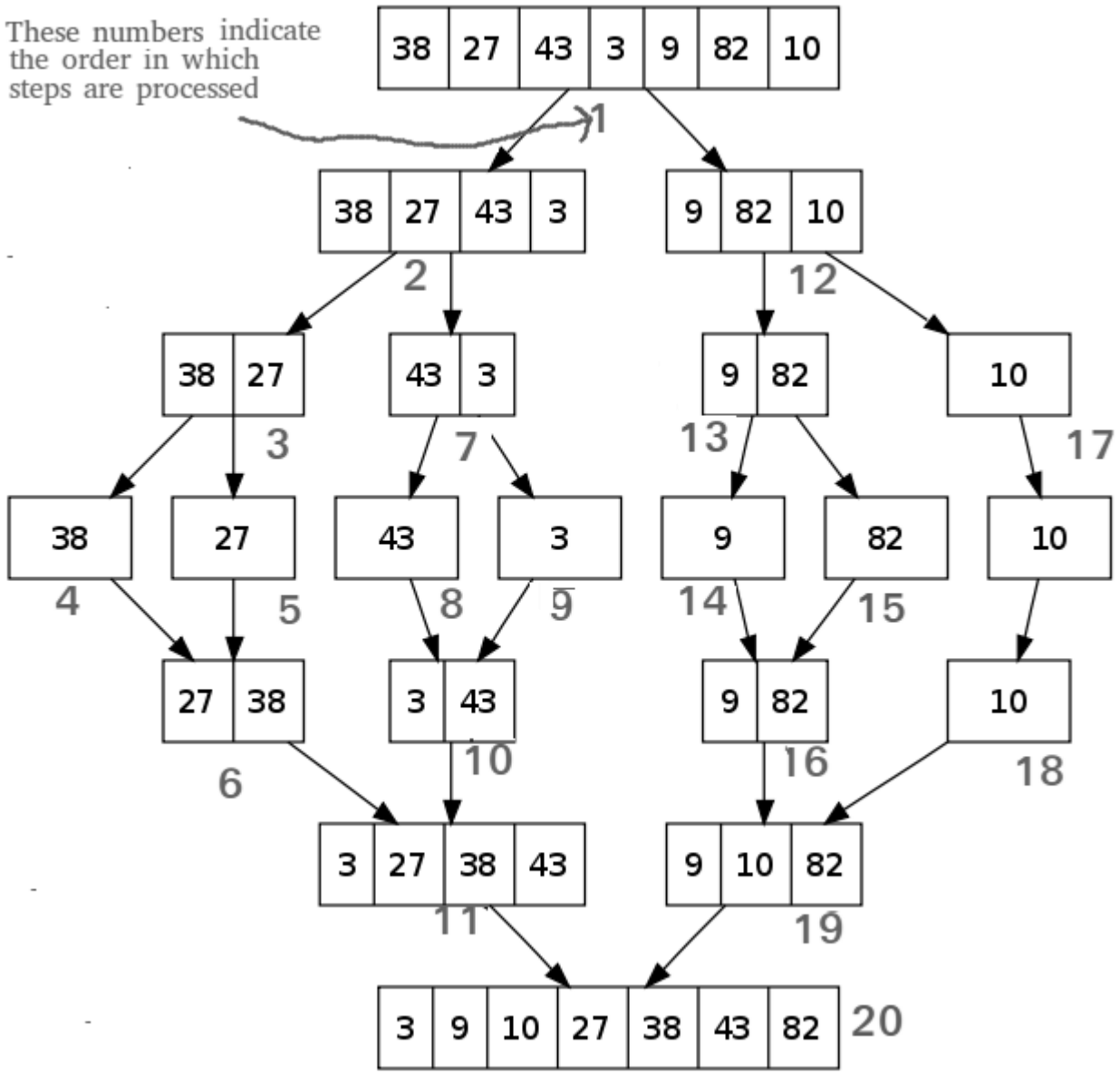
We need a base case. The base case is a subarray containing fewer than two elements, that is, when $p \geq r$, since a subarray with no elements or just one element is already sorted. So we'll divide-conquer-combine only when $p < r$

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

```
ALGORITHM  MERGE_SORT(A, low,high)
if(low<high)
       mid = low + (high-low)/2;
       MERGE_SORT (A,low,mid);
       MERGE_SORT (A,mid+1,high);
      MERGE(A,low,mid,high);
 Endif
End
```

# Merge Sort

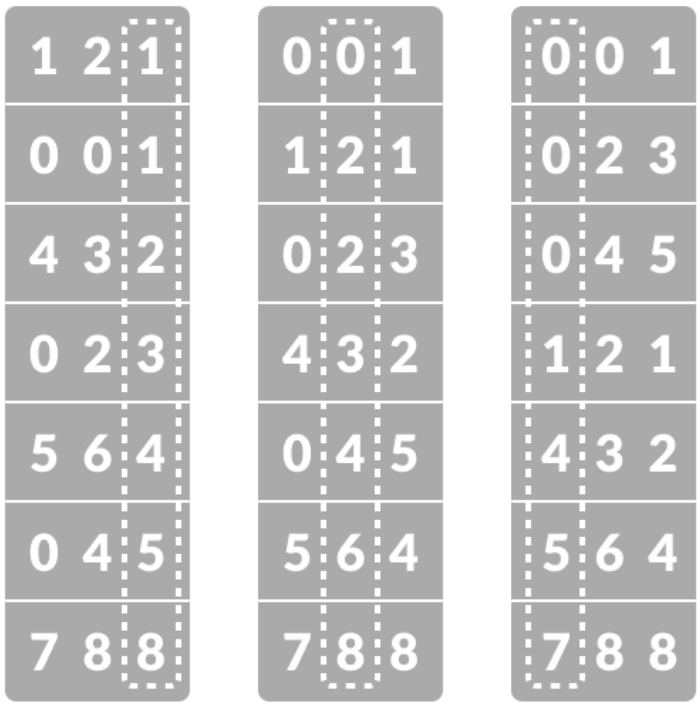

merge_sort (0,5)
9 7 8 3 2 1
mid = (0+5)/2 = 2
1
merge_sort (0,2)
9 7 8
mid = (0+2)/2 = 1
8
merge_sort (3,5)
3 2 1
mid = (3+5)/2 = 4
2
merge_sort (0,1)
9 7
mid = (0+1)/2 = 0
6
merge_sort (2,2)
8
No further call as start = end
3
merge_sort (0,0)
9
No further call as start = end
4
merge_sort (1,1)
7
No further call as start = end
5
merge (0,0,1)
7 9
9
merge_sort (3,4)
3 2
mid = (3+4)/2 = 3
13
merge_sort (5,5)
1
No further call as start = end
10
merge_sort (3,3)
3
No further call as start = end
11
merge_sort (4,4)
2
No further call as start = end
12
merge (3,3,4)
2 3
7
merge (0,2)
7 8 9
14
merge (3,5)
1 2 3
15
merge_sort (0,5)
1 2 3 7 8 9


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

**Radix Sort**

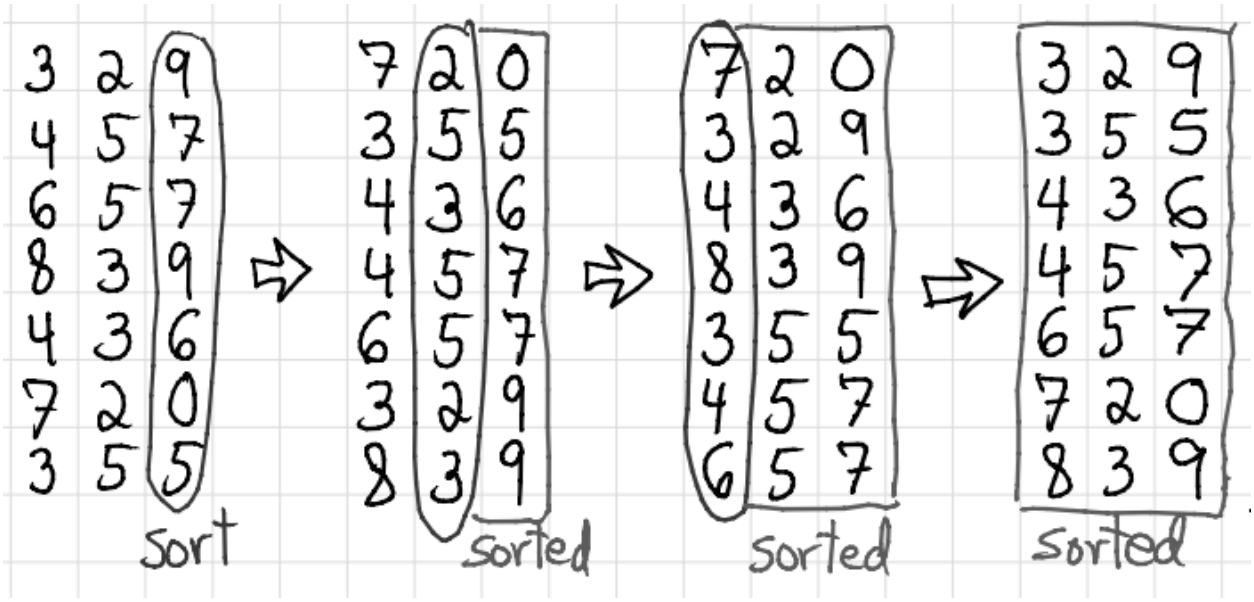
Radix sort is a sorting technique that sorts the elements by first grouping the individual digits of the same **place value**. Then, sort the elements according to their increasing/decreasing order.Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place.

Let the initial array be [121, 432, 564, 23, 1, 45, 788]. It is sorted according to radix sort as shown in the figure below.




sorting the integers according to units, tens and hundreds place digits

Example:



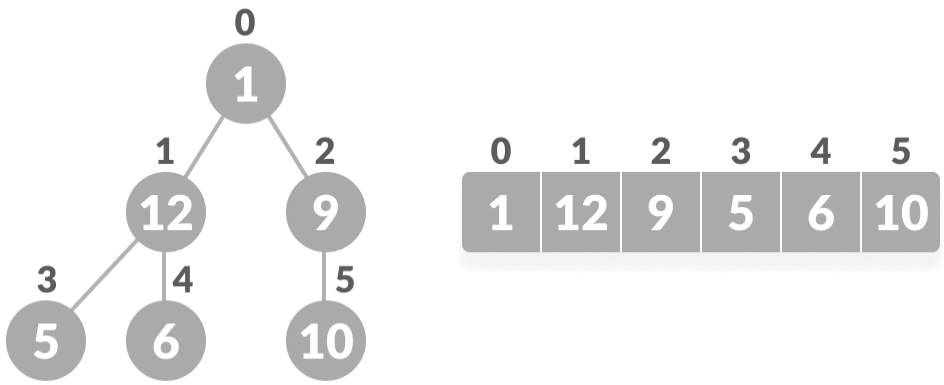

**Heap Sort:**

Heap Sort is a popular and efficient sorting algorithm which uses two types of data structures - arrays and trees. Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called a heap.

**Relationship between Array Indexes and Tree Elements**

In a complete binary tree parent node and children nodes are stored in particular pattern as:

If the index of any element in the array is i, the element in the index 2i+1 will become the left child and element in 2i+2 index will become the right child. Also, the parent of any element at index i is given by the lower bound of (i-1)/2.



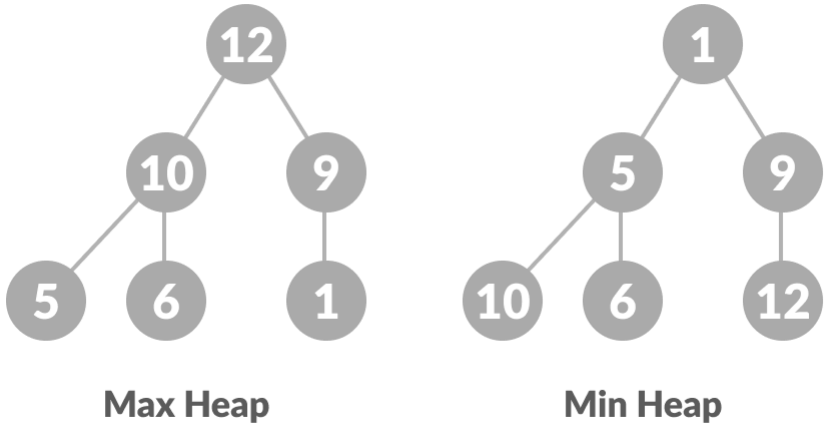

Relationship between array and heap indices

**Heap Data Structure**

Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if

it is a complete binary tree All nodes in the tree follow the property that they are greater than their children i.e. the largest element is at the root and both its children and smaller than the root and so on. Such a heap is called a **max-heap**. If instead, all nodes are smaller than their children, it is called a **min-heap**

The following example diagram shows Max-Heap and Min-Heap.




Max Heap and Min Heap

**Build max-heap**

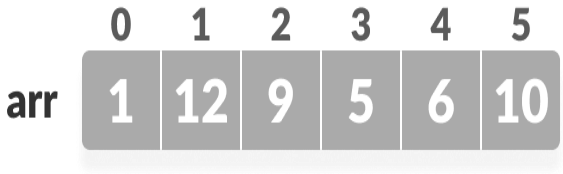
To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied to all the elements including the root element.

In the case of a complete tree, the first index of a non-leaf node is given by n/2 - 1. All other nodes after that are leaf-nodes and thus don't need to be heapified.

```
void heapify(int arr[], int n, int i) {
  // Find largest among root, left child and right child
  int largest = i;
  int left = 2 * i + 1;
  int right = 2 * i + 2;

  if (left < n && arr[left] > arr[largest])
    largest = left;

  if (right < n && arr[right] > arr[largest])
    largest = right;

    // Swap and continue heapifying if root is not largest
    if (largest != i) {
      swap(&arr[i], &arr[largest]);
      heapify(arr, n, largest);
  }
```

So, we can build a maximum heap as

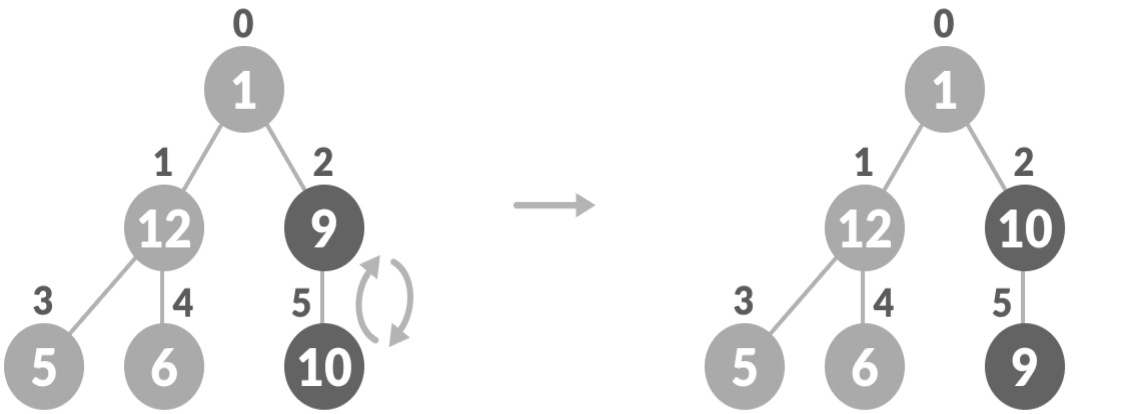
```
    // Build heap (rearrange array)

    for (int i = n / 2 - 1; i >= 0; i--)

      heapify(arr, n, i);
```



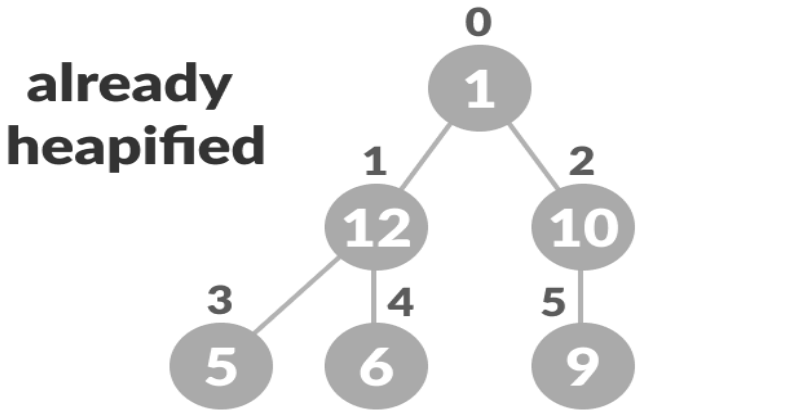

n = 6

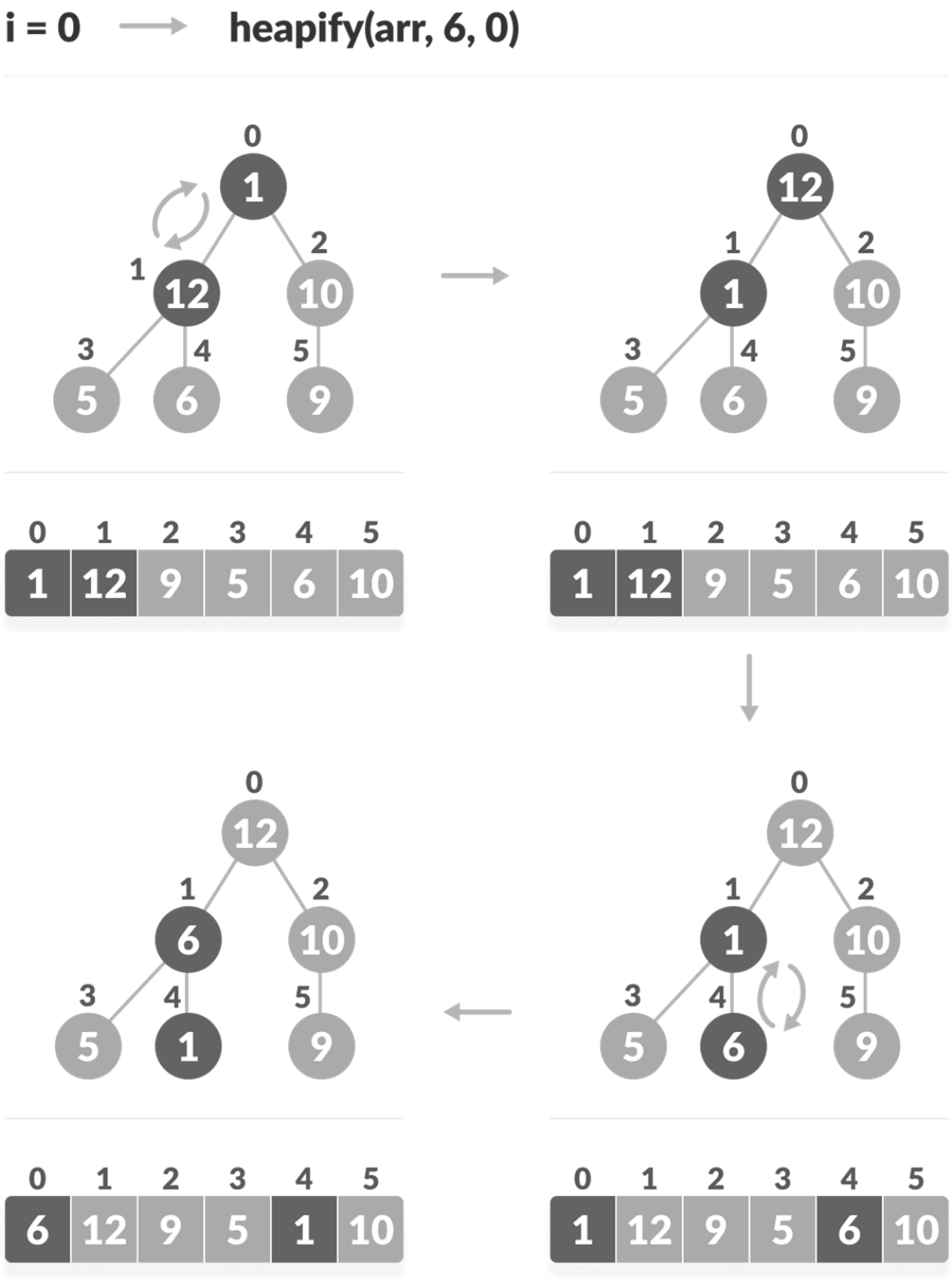
i = 6/2 - 1 = 2 # loop runs from 2 to 0

i = 2 ⟶ heapify(arr, 6, 2)




**i = 1** → **heapify(arr, 6, 1)**





i = 0
heapify(arr, 6, 0)
0 1 2 3 4 5
1 12 9 5 6 10
0 1 2 3 4 5
1 12 9 5 6 10
0 1 2 3 4 5
1 12 9 5 6 10
0 1 2 3 4 5
6 12 9 5 1 10

**Heap Sort**

Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.

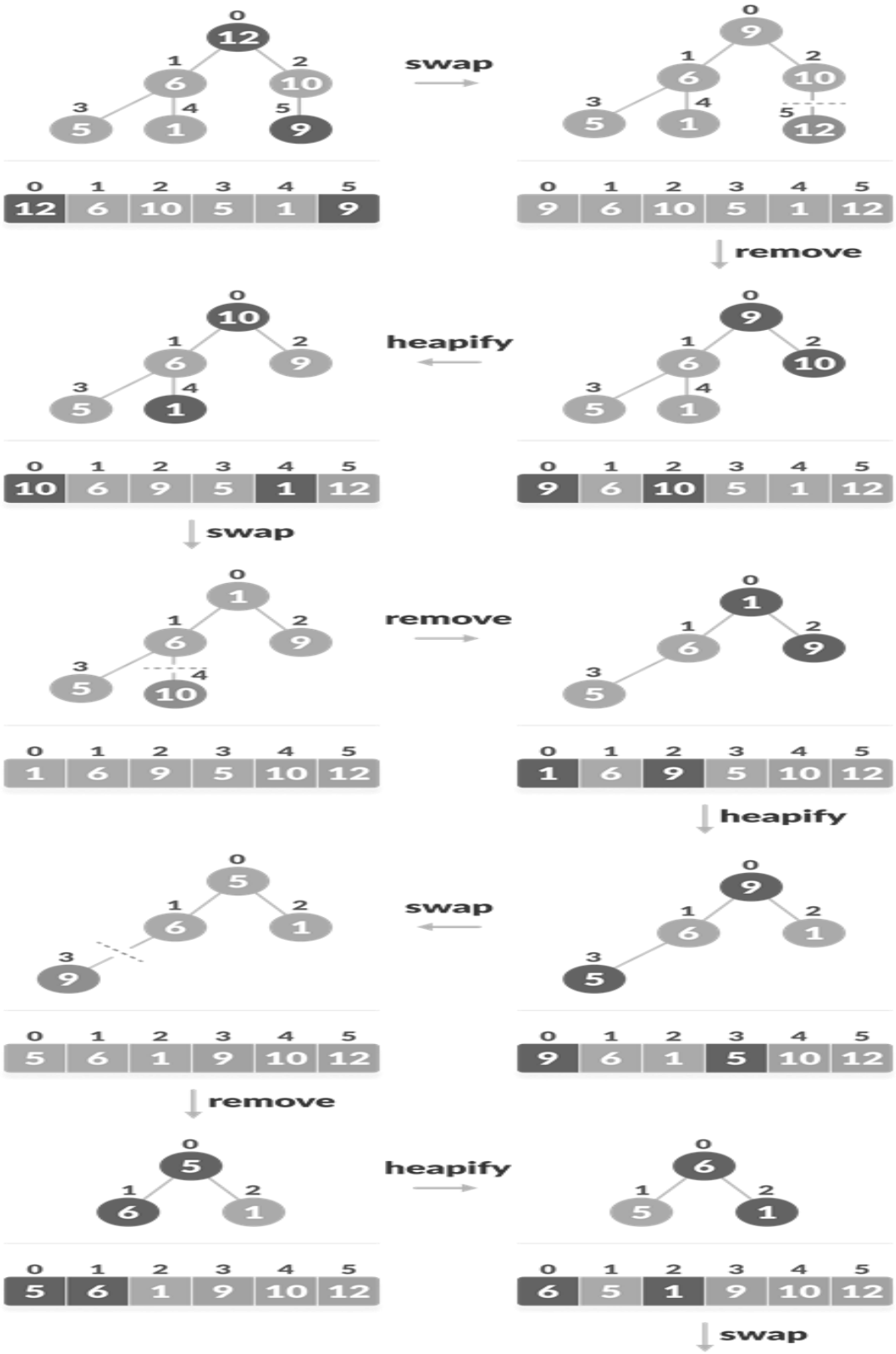
**Swap**: Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.

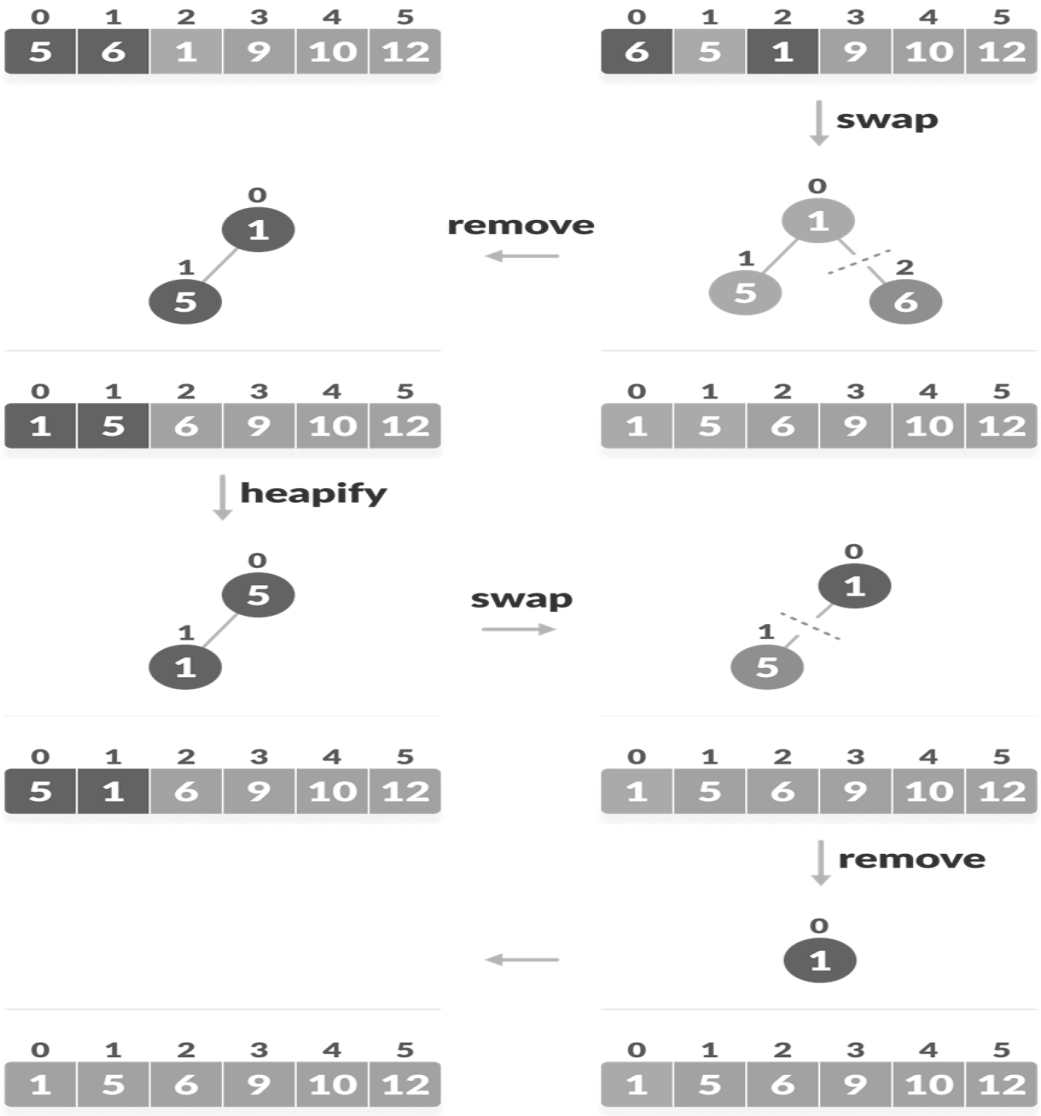
**Remove:** Reduce the size of the heap by 1.

**Heapify:** Heapify the root element again so that we have the highest element at root.

The process is repeated until all the items of the list are sorted.

```
// Heap sort
  for (int i = n - 1; i >= 0; i--) {
    swap(&arr[0], &arr[i]);


    // Heapify root element to get highest element at root again
    heapify(arr, i, 0);
  }
```


swap
remove
heapify
swap
remove
heapify
swap
remove
heapify
swap




Swap, Remove, and Heapify