

# B.C.A study

## Unit -1: Introduction to Data Structure and it's Characteristics

### Introduction

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vitle role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible

### Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned

**Data:** Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

**Group Items:** Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

**Record:** Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

**File:** A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

**Attribute and Entity:** An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

**Field:** Field is a single elementary unit of information representing the attribute of an entity.

## Need of Data Structures

As applications are getting complexed and amount of data is increasing day by day, there may arise the following problems:

**Processor speed:** To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

**Data Search:** Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

**Multiple requests:** If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process

in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

## Advantages of Data Structures

**Efficiency:** Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

**Reusability:** Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

**Abstraction:** Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

## Data Structure Classification



**Linear Data Structures:** A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:  
**Arrays:** An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

**Linked List:** Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

**Stack:** Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: – piles of plates or deck of cards etc.

**Queue:** Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

**Non Linear Data Structures:** This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

**Trees:** Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.

**Graphs:** Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.

## Operations on data structure

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

**Example:** If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

## Array

# Definition

- Arrays are defined as the collection of similar type of data items stored at contiguous memory locations.
- Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.
- Array is the simplest data structure where each data element can be randomly accessed by using its index number.
- For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variable for the marks in different subject. instead of that, we can define an array which can store the marks in each subject at a the contiguous memory locations.

The array **marks[10]** defines the marks of the student in 10 different subjects where each subject marks are located at a particular subscript in the array i.e. **marks[0]** denotes the marks in first subject, **marks[1]** denotes the marks in 2nd subject and so on.

## Properties of the Array

1. Each element is of same data type and carries a same size i.e. int = 4 bytes.
2. Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
3. Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of data element.

for example, in C language, the syntax of declaring an array is like following:

1. `int arr[10]; char arr[10]; float arr[5]`

## Need of using Array

In computer programming, the most of the cases requires to store the large number of data of similar type. To store such amount of data, we need to define a large number of variables. It would be very difficult to remember names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

Following example illustrates, how array can be useful in writing code for a particular problem.

In the following example, we have marks of a student in six different subjects. The problem intends to calculate the average of all the marks of the student.

In order to illustrate the importance of array, we have created two programs, one is without using array and other involves the use of array to store marks.

#### Program without array:

```
1. #include <stdio.h>
2. void main ()
3. {
4.   int marks_1 = 56, marks_2 = 78, marks_3 = 88, marks_4 = 76, marks_5 = 56, marks_6 = 89;
5.   float avg = (marks_1 + marks_2 + marks_3 + marks_4 + marks_5 + marks_6) / 6 ;
6.   printf(avg);
7. }
```

#### Program by using array:

```
1. #include <stdio.h>
2. void main ()
3. {
4.   int marks[6] = {56,78,88,76,56,89};
5.   int i;
6.   float avg;
7.   for (i=0; i<6; i++ )
8.   {
9.     avg = avg + marks[i];
10.  }
11.  printf(avg);
12. }
```

## Advantages of Array

- Array provides the single name for the group of variables of the same type therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process, we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

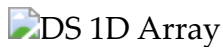
## Memory Allocation of the array

As we have mentioned, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of first element in the main memory. Each element of the array is represented by a proper indexing.

The indexing of the array can be defined in three ways.

1. 0 (zero – based indexing) : The first element of the array will be arr[0].
2. 1 (one – based indexing) : The first element of the array will be arr[1].
3. n (n – based indexing) : The first element of the array can reside at any random index number.

In the following image, we have shown the memory allocation of an array arr of size 5. The array follows 0-based indexing approach. The base address of the array is 100th byte. This will be the address of arr[0]. Here, the size of int is 4 bytes therefore each element will take 4 bytes in the memory.



In 0 based indexing, If the size of an array is n then the maximum index number, an element can have is **n-1**. However, it will be n if we use **1** based indexing.

## Accessing Elements of an array

To access any random element of an array we need the following information:

1. Base Address of the array.
2. Size of an element in bytes.
3. Which type of indexing, array follows.

Address of any element of a 1D array can be calculated by using the following formula:

1. Byte address of element A[i] = base address + size \* ( i – first index)

**Example :**

1. In an array, A[-10 ..... +2 ], Base address (BA) = 999, size of an element = 2 bytes,
2. find the location of A[-1].
3.  $L(A[-1]) = 999 + [(-1) - (-10)] \times 2$
4.     = 999 + 18
5.     = 1017

## 2D Array

2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding bulk of data at once which can be passed to any number of functions wherever required.

## How to declare 2D Array

The syntax of declaring two dimensional array is very much similar to that of a one dimensional array, given as follows.

```
1. int arr[max_rows][max_columns];
```

however, It produces the data structure which looks like following.



Above image shows the two dimensional array, the elements are organized in the form of rows and columns. First element of the first row is represented by `a[0][0]` where the number shown in the first index is the number of that row while the number shown in the second index is the number of the column.

## How do we access data in a 2D array

Due to the fact that the elements of 2D arrays can be random accessed. Similar to one dimensional arrays, we can access the individual cells in a 2D array by using the indices of the cells. There are two indices attached to a particular cell, one is its row number while the other is its column number.

However, we can store the value stored in any particular cell of a 2D array to some variable `x` by using the following syntax.

```
1. int x = a[i][j];
```

where `i` and `j` is the row and column number of the cell respectively.

We can assign each cell of a 2D array to 0 by using the following code:

```
1. for ( int i=0; i<n ;i++)  
2. {  
3.     for (int j=0; j<n; j++)  
4.     {  
5.         a[i][j] = 0;  
6.     }  
7. }
```



# Initializing 2D Arrays

We know that, when we declare and initialize one dimensional array in C programming simultaneously, we don't need to specify the size of the array. However this will not work with 2D arrays. We will have to define at least the second dimension of the array.

The syntax to declare and initialize the 2D array is given as follows.

```
1. int arr[2][2] = {0,1,2,3};
```


The number of elements that can be present in a 2D array will always be equal to (**number of rows \* number of columns**).

**Example :** Storing User's data into a 2D array and printing it.

## C Example :

```
1. #include <stdio.h>
2. void main ()
3. {
4.     int arr[3][3],i,j;
5.     for (i=0;i<3;i++)
6.     {
7.         for (j=0;j<3;j++)
8.         {
9.             printf("Enter a[%d][%d]: ",i,j);
10.            scanf("%d",&arr[i][j]);
11.        }
12.    }
13.    printf("\n printing the elements ....\n");
14.    for(i=0;i<3;i++)
15.    {
16.        printf("\n");
17.        for (j=0;j<3;j++)
18.        {
19.            printf("%d\t",arr[i][j]);
20.        }
21.    }
22. }
```

# Sparse matrix

 Example of a sparse matrix.

A **sparse matrix** is a one in which the majority of the values are zero. The proportion of zero elements to non-zero elements is referred to as the **sparsity** of the matrix. The opposite of a sparse matrix, in which the majority of its values are non-zero, is called a **dense matrix**.

Sparse matrices are used by scientists and engineers when solving partial differential equations. For example, a measurement of a matrix's sparsity can be useful when developing theories about the connectivity of computer network. When using large sparse matrices in a computer program, it is important to optimize the data structures and algorithm to take advantage of the fact that most of the values will be zero.

## Sparse matrix example

Here is an example of a  $4 \times 4$  matrix containing 12 zero values and 4 non-zero values, giving it a sparsity of 3:

```
[[5, 0, 0, 0],  
 [0, 11, 0, 0],  
 [0, 0, 25, 0],  
 [0, 0, 0, 7]]
```

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value)**.

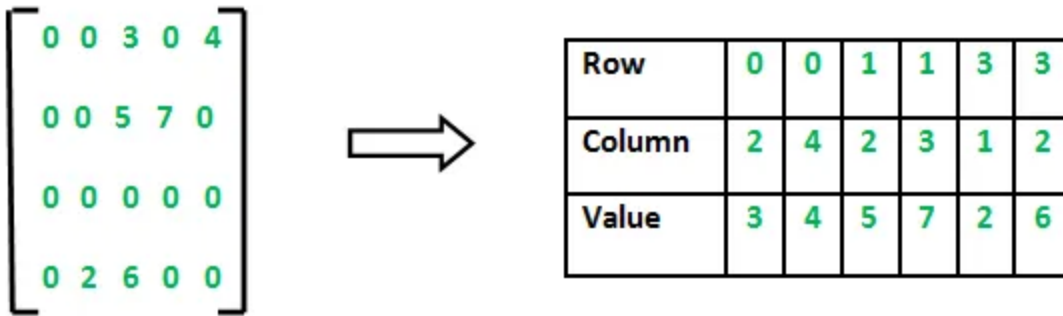
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

### Method 1: Using Arrays

2D array is used to represent a sparse matrix in which there are three rows named as

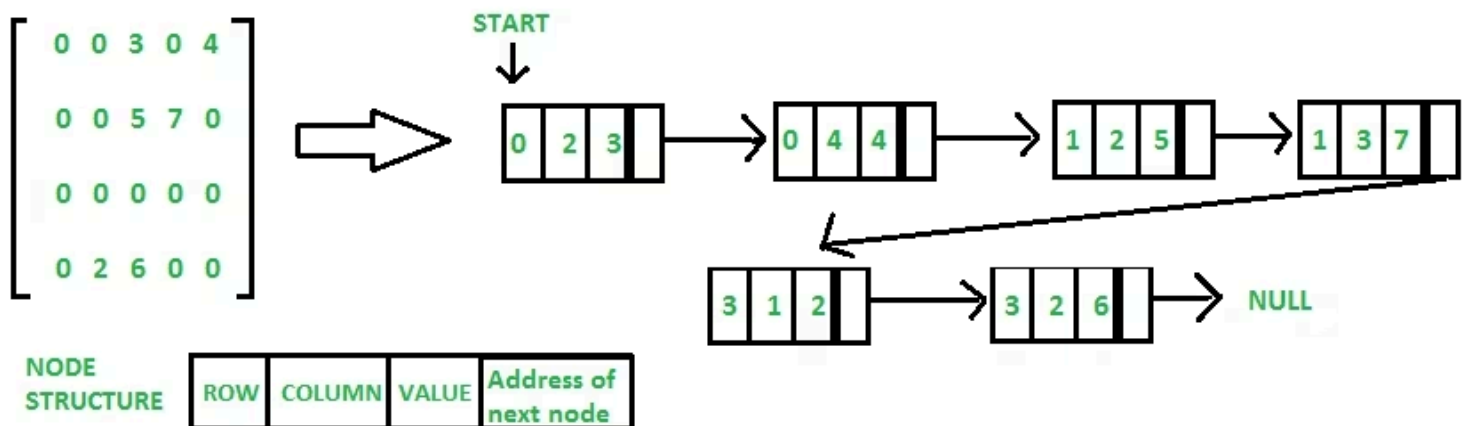
- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)



## Method 2: Using Linked Lists

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node



(<https://media.geeksforgeeks.org/wp-content/uploads/Sparse-Matrix-Linked-List-22.png>)

### • Upper and Lower Triangular Matrices:

– Definition: An upper triangular matrix is a square matrix in which all entries below the main diagonal are zero (only nonzero entries are found above the main diagonal – in the upper triangle). A lower triangular matrix is a square matrix in which all entries above the main diagonal are zero (only nonzero entries are found below the main diagonal – in the lower triangle). See the picture below.

– Notation: An upper triangular matrix is typically denoted with U and a lower triangular matrix is typically denoted with L.

– Properties:



[A WordPress.com Website.](#)

# B.C.A study

## Unit -2 : Stacks and Queues

### Stacks and Queues

An array is a *random access* data structure, where each element can be accessed directly and in constant time. A typical illustration of random access is a book – each page of the book can be open independently of others. Random access is critical to many algorithms, for example binary search.

A linked list is a *sequential access* data structure, where each element can be accessed only in particular order. A typical illustration of sequential access is a roll of paper or tape – all prior material must be unrolled in order to get to data you want.

In this note we consider a subcase of sequential data structures, so-called *limited access* data structures.

### Stacks

A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack. A stack is a limited access data structure – elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

A stack is a **recursive** data structure. Here is a structural definition of a Stack:

## Applications

- The simplest application of a stack is to reverse a word. You push a given word to stack – letter by letter – and then pop letters from the stack.
- Another application is an “undo” mechanism in text editors; this operation is accomplished by keeping all text changes in a stack. **Backtracking.** This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze – how do you find a way from an entrance to an exit? Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.
- Language processing:
  - space for parameters and local variables is created internally using a stack.
  - compiler’s syntax check for matching braces is implemented by using stack.
  - support for recursion

## Implementation

In the standard library of classes, the data type stack is an *adapter* class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, an ArrayList, a linked list, or any other collection. Regardless of the type of the underlying data structure, a Stack must implement the same functionality. This is achieved by providing a unique interface: `public interface StackInterface<AnyType> { public void push(AnyType e); public AnyType pop(); public AnyType peek(); public boolean isEmpty(); }` The following picture demonstrates the idea of implementation *by composition*.

Another implementation requirement (in addition to the above interface) is that all stack operations must run in **constant time  $O(1)$** . Constant time means that there is some constant  $k$  such that an operation takes  $k$  nanoseconds of computational time regardless of the stack size.

## Array-based implementation

In an array-based implementation we maintain the following fields: an array  $A$  of a default size ( $\geq 1$ ), the variable *top* that refers to the top element in the stack and the *capacity* that refers to the array size. The variable *top* changes from  $-1$  to  $\text{capacity} - 1$ . We say that a stack is empty when  $\text{top} = -1$ , and

the stack is full when  $top = capacity - 1$ .

In a fixed-size stack abstraction, the capacity stays unchanged, therefore when *top* reaches *capacity*, the stack object throws an exception. See [ArrayStack.java \(https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/code/ArrayStack.java\)](https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/code/ArrayStack.java) for a complete implementation of the stack class.

In a dynamic stack abstraction when *top* reaches *capacity*, we double up the stack size.

## Linked List-based implementation

Linked List-based implementation provides the best (from the efficiency point of view) dynamic stack implementation.

See [ListStack.java \(https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/code/ListStack.java\)](https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/code/ListStack.java) for a complete implementation of the stack class.

## Queues

A queue is a container of objects (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle. An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed **enqueue** and **dequeue**. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

## Implementation

In the standard library of classes, the data type queue is an *adapter* class, meaning that a queue is built on top of other data structures. The underlying structure for a queue could be an array, a Vector, an ArrayList, a LinkedList, or any other collection. Regardless of the type of the underlying data structure,

a queue must implement the same functionality. This is achieved by providing a unique interface.

```
interface QueueInterface<AnyType> { public boolean isEmpty(); public AnyType getFront(); public AnyType dequeue(); public void enqueue(AnyType e); public void clear(); }
```

Each of the above basic operations must run at constant time  $O(1)$ . The following picture demonstrates the idea of implementation by composition.

## Circular Queue

Given an array  $A$  of a default size ( $\geq 1$ ) with two references *back* and *front*, originally set to -1 and 0 respectively. Each time we insert (enqueue) a new item, we increase the back index; when we remove (dequeue) an item – we increase the front index. Here is a picture that illustrates the model after a few steps:

As you see from the picture, the queue logically moves in the array from left to right. After several moves *back* reaches the end, leaving no space for adding new elements

However, there is a free space before the front index. We shall use that space for enqueueing new items, i.e. the next entry will be stored at index 0, then 1, until *front*. Such a model is called a **wrap around queue** or a **circular queue**

Finally, when *back* reaches *front*, the queue is full. There are two choices to handle a full queue: a) throw an exception; b) double the array size.

The circular queue implementation is done by using the modulo operator (denoted %), which is computed by taking the remainder of division (for example,  $8\%5$  is 3). By using the modulo operator, we can view the queue as a circular array, where the “wrapped around” can be simulated as “back % array\_size”. In addition to the back and front indexes, we maintain another index: *cur* – for counting the number of elements in a queue. Having this index simplifies a logic of implementation.

# Arithmetic Expression Evaluation

An important application of stacks is in parsing. For example, a compiler must parse arithmetic expressions written using infix notation:  $1 + ((2 + 3) * 4 + 5) * 6$ . We break the problem of parsing infix expressions into two stages. First, we convert from infix to a different representation called postfix. Then we parse the postfix expression, which is a somewhat easier problem than directly parsing infix.

**Converting from Infix to Postfix.** Typically, we deal with expressions in infix notation  $2 + 5$  where the operators (e.g.  $+$ ,  $*$ ) are written between the operands (e.g. 2 and 5). Writing the operators after the operands gives a postfix expression 2 and 5 are called operands, and the  $+$  is operator. The above arithmetic expression is called infix, since the operator is in between operands. The expression  $2\ 5\ +$  Writing the operators before the operands gives a prefix expression  $+2\ 5$ . Suppose you want to compute the cost of your shopping trip. To do so, you add a list of numbers and multiply them by the local sales tax (7.25%):  $70 + 150 * 1.0725$ . Depending on the calculator, the answer would be either 235.95 or 230.875. To avoid this confusion we shall use a postfix notation  $70\ 150\ +\ 1.0725\ *$ . Postfix has the nice property that parentheses are unnecessary.

Now, we describe how to convert from infix to postfix.

1. Read in the tokens one at a time
2. If a token is an integer, write it into the output
3. If a token is an operator, push it to the stack, if the stack is empty. If the stack is not empty, you pop entries with higher or equal priority and only then you push that token to the stack.
4. If a token is a left parentheses '(', push it to the stack
5. If a token is a right parentheses ')', you pop entries until you meet '('.
6. When you finish reading the string, you pop up all tokens which are left there.
7. Arithmetic precedence is in increasing order:  $+$ ,  $-$ ,  $*$ ,  $/$ ;

Example. Suppose we have an infix expression:  $2 + (4 + 3 * 2 + 1) / 3$ . We read the string by characters. '2' – send to the output. '+' – push on the stack. '(' – push on the stack. '4' – send to the output. '+' – push on the stack. '3' – send to the output. '\*' – push on the stack. '2' – send to the output.

**Evaluating a Postfix Expression.** We describe how to parse and evaluate a postfix expression.

1. We read the tokens in one at a time.
2. If it is an integer, push it on the stack
3. If it is a binary operator, pop the top two elements from the stack, apply the operator, and push the result back on the stack.

...

[A WordPress.com Website.](https://bcastudyguide.com/unit-2-stacks-and-queues/)

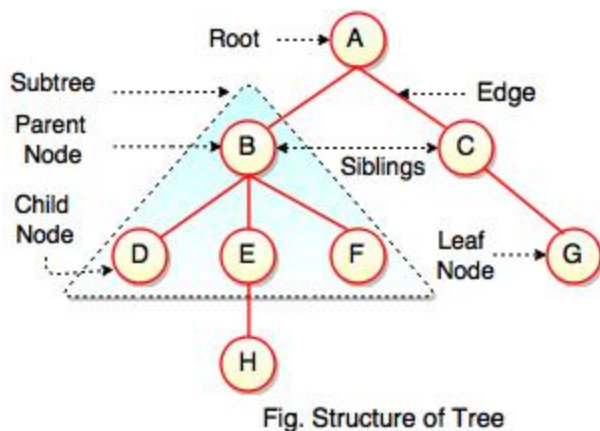


# B.C.A study

## Unit- 4 :Trees

### What are trees?

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- Tree is one of the most powerful and advanced data structures.
- It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- It represents the nodes connected by edges.



The above figure represents structure of a tree. Tree has 2 subtrees.

A is a parent of B and C.

B is called a child of A and also parent of D, E, F.

Tree is a collection of elements called Nodes, where each node can have arbitrary number of children.

In the above figure, D, F, H, G are **leaves**. B and C are **siblings**. Each node excluding a root is connected by a direct edge from exactly one other node  
parent → children.

## Levels of a node

Levels of a node represents the number of connections between the node and the root. It represents generation of a node. If the root node is at level 0, its next node is at level 1, its grand child is at level 2 and so on. Levels of a node can be shown as follows:

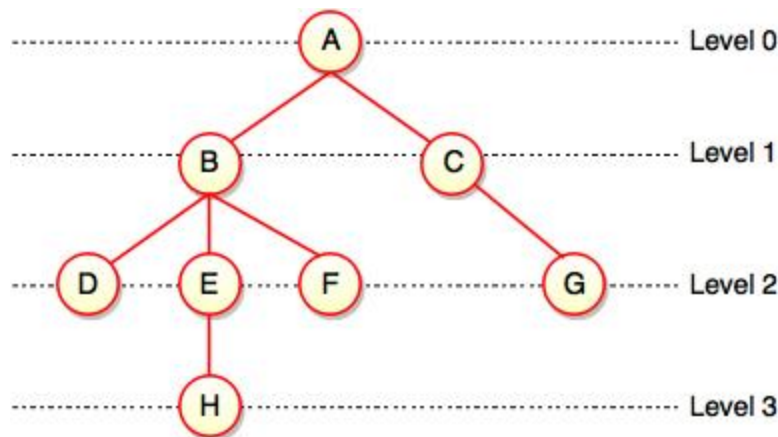


Fig. Levels of Tree

## Binary Tree Traversal

Binary tree traversing is a process of accessing every node of the tree and exactly once. A tree is defined in a recursive manner. Binary tree traversal also defined recursively.

**There are three techniques of traversal:**

1. Preorder Traversal
2. Postorder Traversal
3. Inorder Traversal

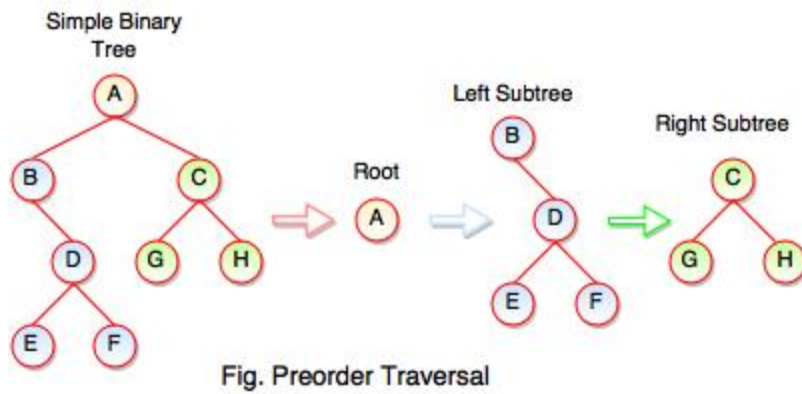
### 1. Preorder Traversal

**Algorithm for preorder traversal**

**Step 1 :** Start from the Root.

**Step 2 :** Then, go to the Left Subtree.

**Step 3 :** Then, go to the Right Subtree.



The above figure represents how preorder traversal actually works.

**Following steps can be defined the flow of preorder traversal:**

**Step 1 :** A + B (B + Preorder on D (D + Preorder on E and F)) + C (C + Preorder on G and H)

**Step 2 :** A + B + D (E + F) + C (G + H)

**Step 3 :** A + B + D + E + F + C + G + H

**Preorder Traversal :** A B C D E F G H

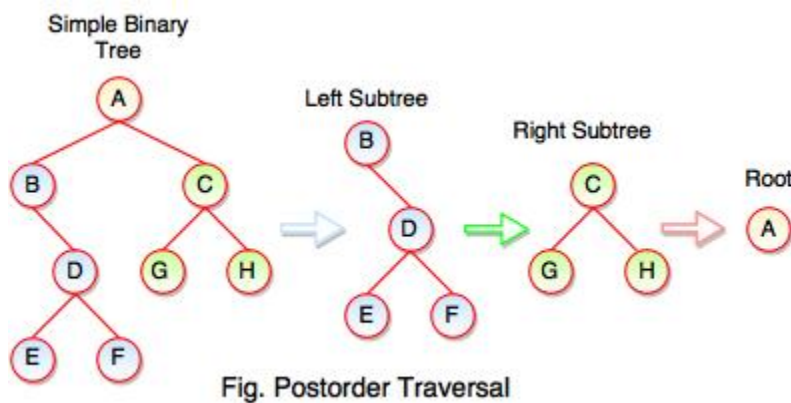
## 2. Postorder Traversal

**Algorithm for postorder traversal**

**Step 1 :** Start from the Left Subtree (Last Leaf).

**Step 2 :** Then, go to the Right Subtree.

**Step 3 :** Then, go to the Root.



The above figure represents how postorder traversal actually works.

**Following steps can be defined the flow of postorder traversal:**

**Step 1 :** As we know, preorder traversal starts from left subtree (last leaf) ((Postorder on E + Postorder on F) + D + B )) + ((Postorder on G + Postorder on H) + C) + (Root A)

**Step 2 :** (E + F) + D + B + (G + H) + C + A

**Step 3 :** E + F + D + B + G + H + C + A

**Postorder Traversal : E F D B G H C A**

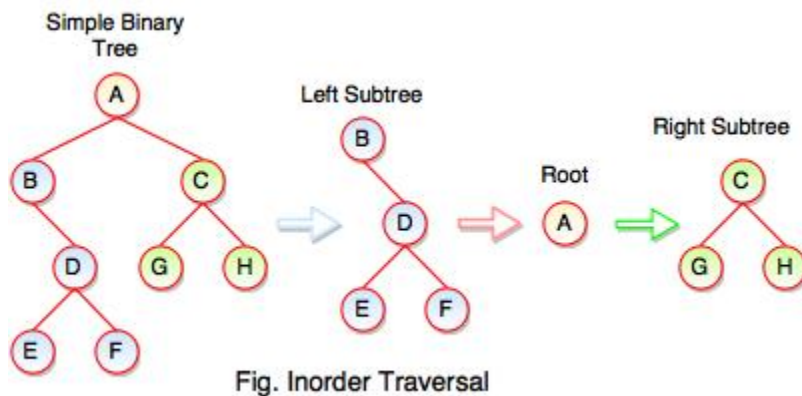
### 3. Inorder Traversal

**Algorithm for inorder traversal**

**Step 1 :** Start from the Left Subtree.

**Step 2 :** Then, visit the Root.

**Step 3 :** Then, go to the Right Subtree.



The above figure represents how inorder traversal actually works.

**Following steps can be defined the flow of inorder traversal:**

**Step 1 :** B + (Inorder on E) + D + (Inorder on F) + (Root A ) + (Inorder on G) + C (Inorder on H)

**Step 2 :** B + (E) + D + (F) + A + G + C + H

**Step 3 :** B + E + D + F + A + G + C + H

**Inorder Traversal : B E D F A G C H**

**Binary Tree**

Binary tree is a special type of data structure. In binary tree, every node can have a maximum of 2 children, which are known as **Left child** and **Right Child**. It is a method of placing and locating the records in a database, especially when all the data is known to be in random access memory (RAM).

### Definition:

“A tree in which every node can have maximum of two children is called as Binary Tree.”

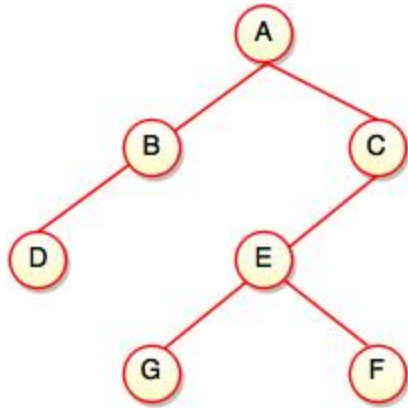


Fig. Binary Tree

The above tree represents binary tree in which node A has two children B and C. Each children have one child namely D and E respectively.

### Representation of Binary Tree using Array

Binary tree using array represents a node which is numbered sequentially level by level from left to right. Even empty nodes are numbered.

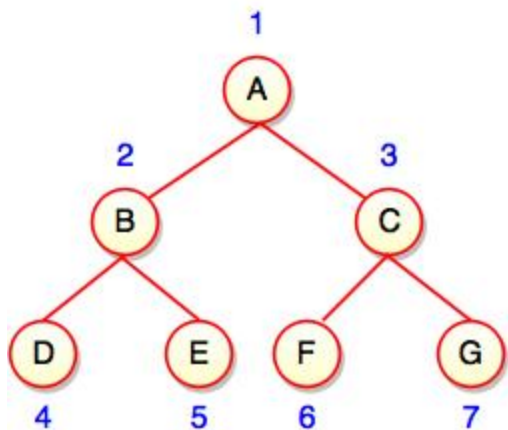


Fig. Binary Tree using Array

Array index is a value in tree nodes and array value gives to the parent node of that particular index or node. Value of the root node index is always -1 as there is no parent for root. When the data item of the tree is sorted in an array, the number appearing against the node will work as indexes of the node in an

array.

0	1	2	3	4	5	6	7
7	A	B	C	D	E	F	G

Fig. Location Number of an Array in a Tree

Location number of an array is used to store the size of the tree. The first index of an array that is '0', stores the total number of nodes. All nodes are numbered from left to right level by level from top to bottom. In a tree, each node having an index  $i$  is put into the array as its  $i$ th element.

The above figure shows how a binary tree is represented as an array. Value '7' is the total number of nodes. If any node does not have any of its child, null value is stored at the corresponding index of the array.

## Binary Search Tree

- Binary search tree is a binary tree which has special property called BST.
- BST property is given as follows:

**For all nodes A and B,**

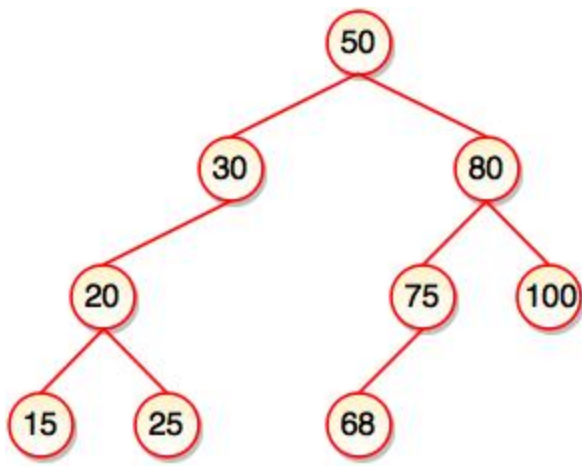
- If B belongs to the left subtree of A, the key at B is less than the key at A.
- If B belongs to the right subtree of A, the key at B is greater than the key at A.

**Each node has following attributes:**

- Parent (P), left, right which are pointers to the parent (P), left child and right child respectively.
- Key defines a key which is stored at the node.

**Definition:**

"Binary Search Tree is a binary tree where each node contains only smaller values in its left subtree and only larger values in its right subtree."



**Fig. Binary Search Tree**

- The above tree represents binary search tree (BST) where left subtree of every node contains smaller values and right subtree of every node contains larger value.
- Binary Search Tree (BST) is used to enhance the performance of binary tree.
- It focuses on the search operation in binary tree.

**Note:** Every binary search tree is a binary tree, but all the binary trees need not to be binary search trees.

## Binary Search Tree Operations

Following are the operations performed on binary search tree:

### 1. Insert Operation

- Insert operation is performed with  $O(\log n)$  time complexity in a binary search tree.
- Insert operation starts from the root node. It is used whenever an element is to be inserted.

**The following algorithm shows the insert operation in binary search tree:**

**Step 1:** Create a new node with a value and set its left and right to NULL.

**Step 2:** Check whether the tree is empty or not.

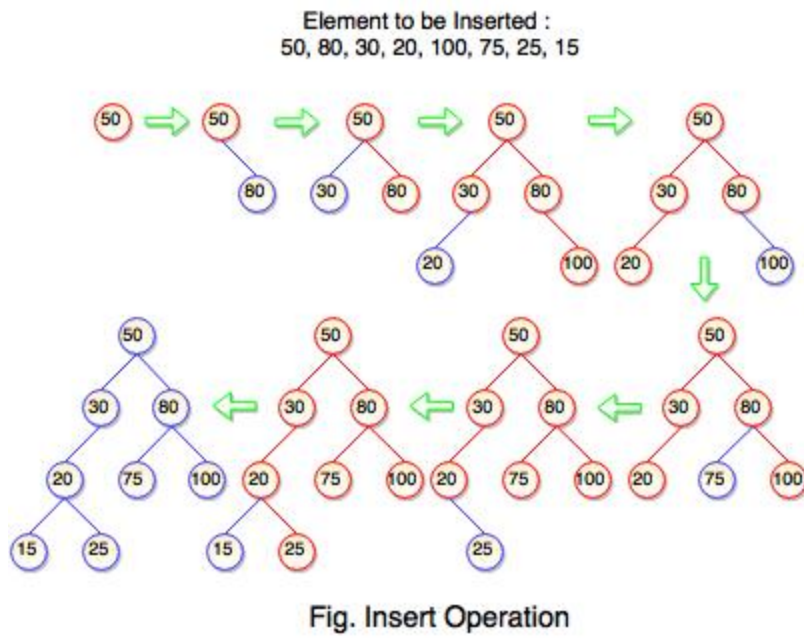
**Step 3:** If the tree is empty, set the root to a new node.

**Step 4:** If the tree is not empty, check whether a value of new node is smaller or larger than the node (here it is a root node).

**Step 5:** If a new node is smaller than or equal to the node, move to its left child.

**Step 6:** If a new node is larger than the node, move to its right child.

**Step 7:** Repeat the process until we reach to a leaf node.



The above tree is constructed a binary search tree by inserting the above elements {50, 80, 30, 20, 100, 75, 25, 15}. The diagram represents how the sequence of numbers or elements are inserted into a binary search tree.

## 2. Search Operation

- Search operation is performed with  $O(\log n)$  time complexity in a binary search tree.
- This operation starts from the root node. It is used whenever an element is to be searched.

**The following algorithm shows the search operation in binary search tree:**

**Step 1:** Read the element from the user .

**Step 2:** Compare this element with the value of root node in a tree.

**Step 3:** If element and value are matching, display “Node is Found” and terminate the function.

**Step 4:** If element and value are not matching, check whether an element is smaller or larger than a node value.

**Step 5:** If an element is smaller, continue the search operation in left subtree.

**Step 6:** If an element is larger, continue the search operation in right subtree.

**Step 7:** Repeat the same process until we found the exact element.

**Step 8:** If an element with search value is found, display “Element is found” and terminate the function.



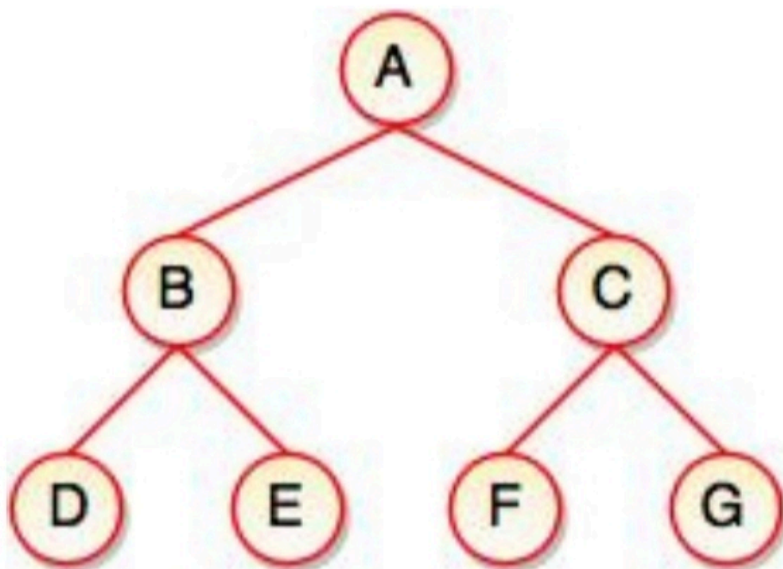
**Step 9:** If we reach to a leaf node and the search value is not match to a leaf node, display “Element is not found” and terminate the function.

**There are four types of binary tree:**

1. Full Binary Tree
2. Complete Binary Tree
3. Skewed Binary Tree
4. Extended Binary Tree

## 1. Full Binary Tree

- If each node of binary tree has either two children or no child at all, is said to be a **Full Binary Tree**.
- Full binary tree is also called as **Strictly Binary Tree**.



**Fig. Full Binary Tree**

- Every node in the tree has either 0 or 2 children.
- Full binary tree is used to represent mathematical expressions.



## 2. Complete Binary Tree

- If all levels of tree are completely filled except the last level and the last level has all keys as left as possible, is said to be a **Complete Binary Tree**.
- Complete binary tree is also called as **Perfect Binary Tree**.

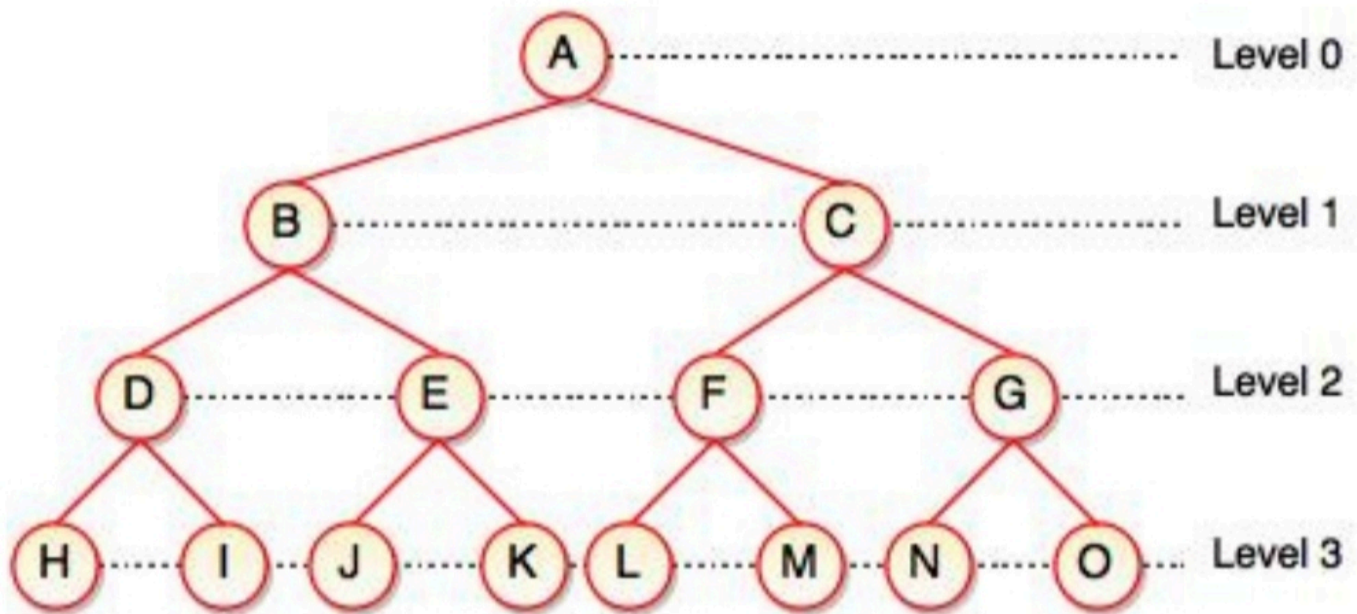


Fig. Complete Binary Tree

- In a complete binary tree, every internal node has exactly two children and all leaf nodes are at same level.
- For example, at Level 2, there must be  $2^2 = 4$  nodes

and at Level 3 there must be  $2^3 = 8$  nodes.

### 3. Skewed Binary Tree

- If a tree which is dominated by left child node or right child node, is said to be a **Skewed Binary Tree**.
- In a skewed binary tree, all nodes except one have only one child node. The remaining node has no child.

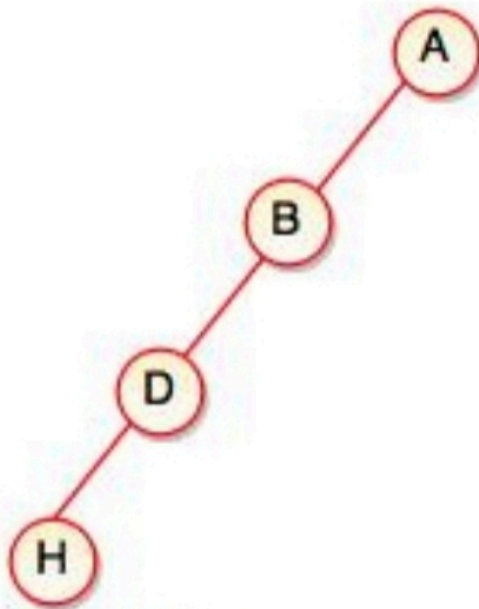


Fig. Left Skewed Binary Tree

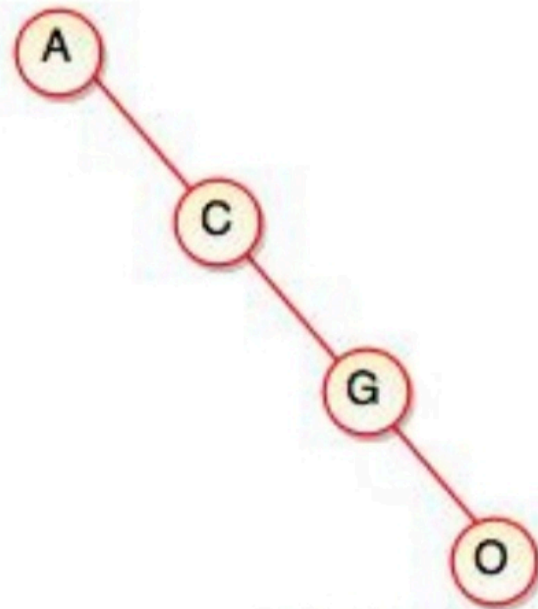
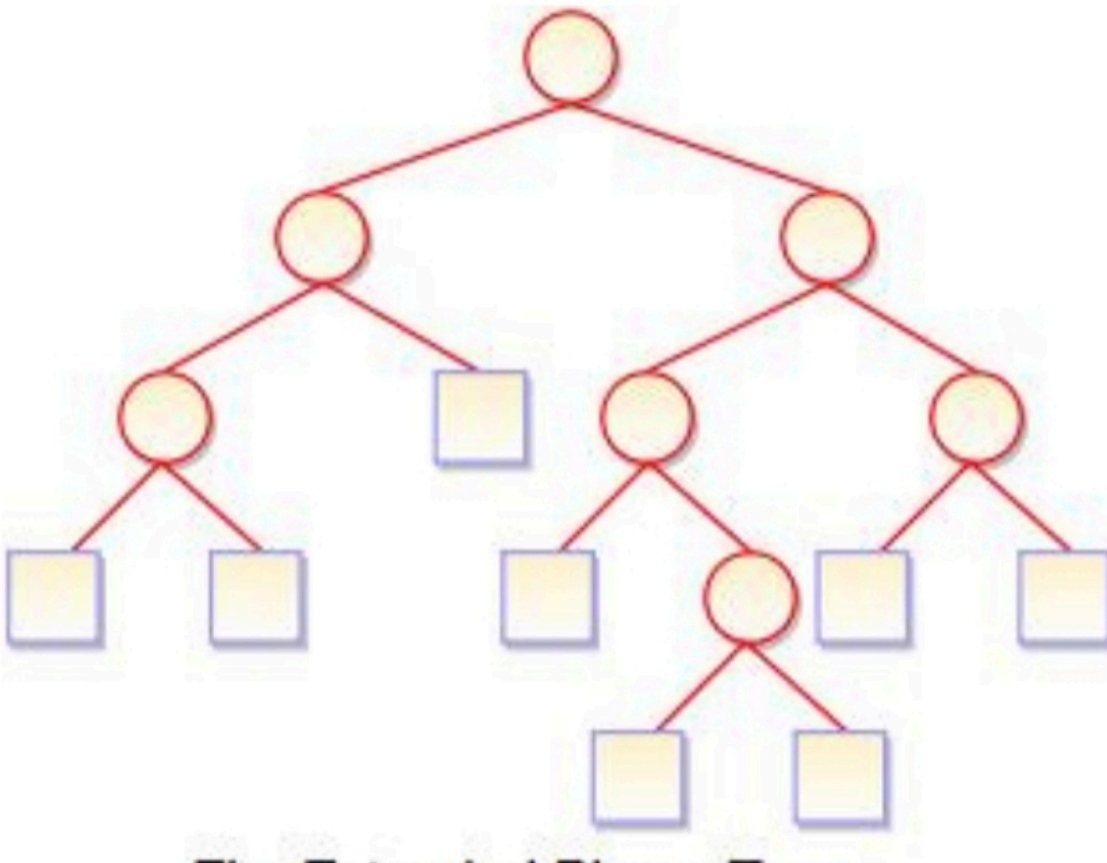


Fig. Right Skewed Binary Tree

- In a left skewed tree, most of the nodes have the left child without corresponding right child.
- In a right skewed tree, most of the nodes have the right child without corresponding left child.

## 4. Extended Binary Tree

- Extended binary tree consists of replacing every null subtree of the original tree with special nodes.
  - Empty circle represents internal node and filled circle represents external node.
  - The nodes from the original tree are internal nodes and the special nodes are external nodes.
  - Every internal node in the extended binary tree has exactly two children and every external node is a leaf.
- It displays the result which is a **complete binary tree**.



## Fig. Extended Binary Tree



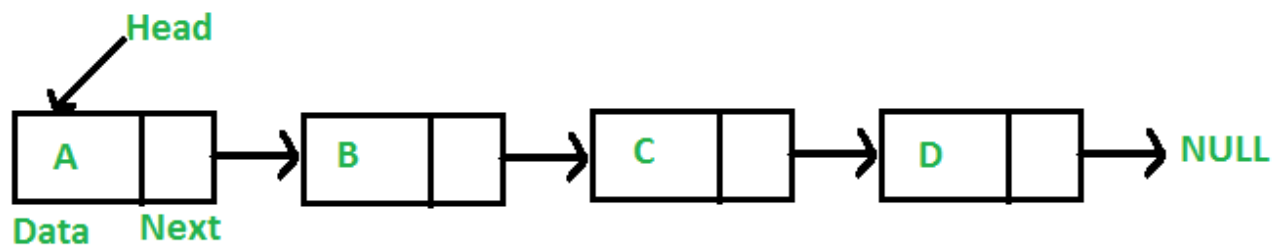
[A WordPress.com Website.](#)



# B.C.A study

## Unit-3:List

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

### Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

- 1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
- 2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

For example, in a system, if we maintain a sorted list of IDs in an array `id[]`.

`id[] = [1000, 1010, 1050, 2000, 2040]`.

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in `id[]`, everything after 1010 has to be moved.

### Advantages over arrays

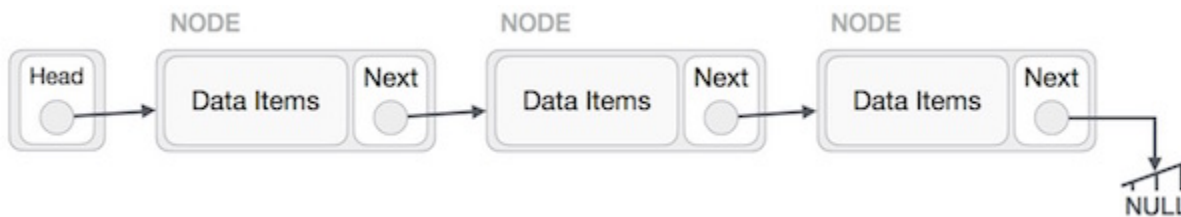
- 1) Dynamic size
- 2) Ease of insertion/deletion

**Drawbacks:**

- 1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation. Read about it [here \(https://www.geeksforgeeks.org/binary-search-on-singly-linked-list/\)](https://www.geeksforgeeks.org/binary-search-on-singly-linked-list/).
- 2) Extra memory space for a pointer is required with each element of the list.
- 3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

## Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

## Types of Linked List

Following are the various types of linked list.

- **Simple Linked List**
- **Doubly Linked List**
- **Circular Linked List**



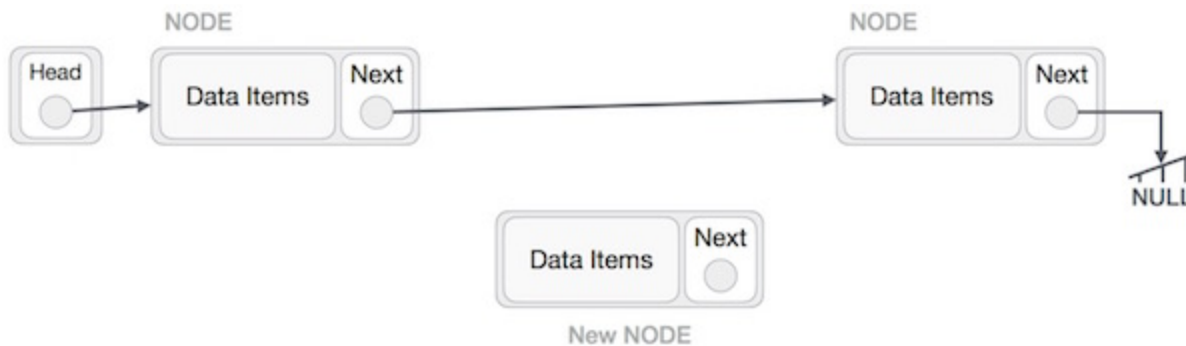
# Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

## Insertion Operation

Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



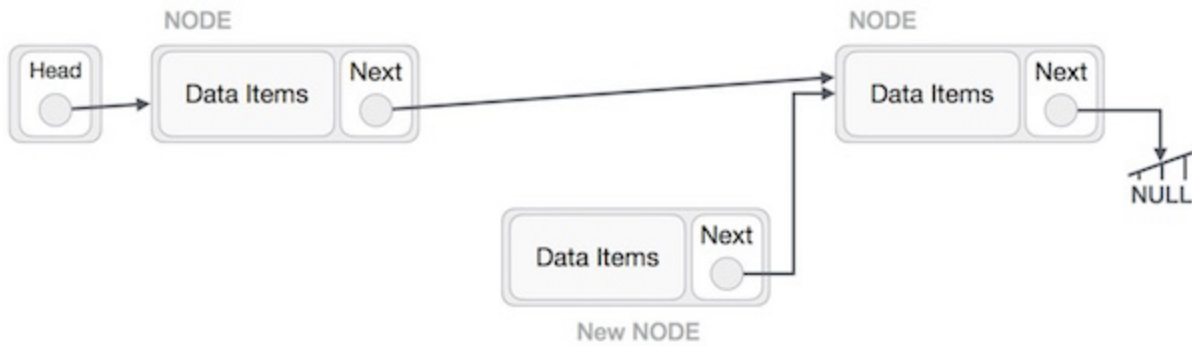
Imagine that we are inserting a node **B**(NewNode), between **A** (LeftNode) and **C**(RightNode). Then point B.next to C –

```

NewNode.next -> RightNode;

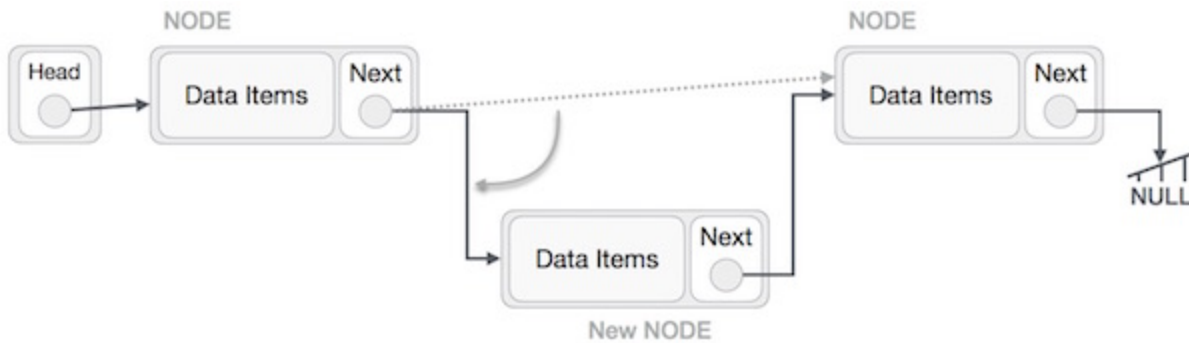
```

It should look like this –

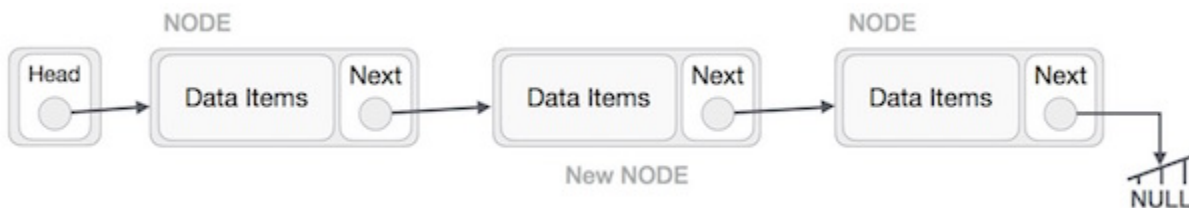


Now, the next node at the left should point to the new node.

`LeftNode.next -> NewNode;`



This will put the new node in the middle of the two. The new list should look like this –



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

// A simple C program to introduce //

```
#include <stdio.h> #include <stdlib.h> structNode { intdata; structNode* next; };
```

```
intmain() { structNode* head = NULL; structNode* second = NULL; structNode* third = NULL;
```

```

head = (structNode*)malloc(sizeof(structNode)); second =
(structNode*)malloc(sizeof(structNode)); third =
(structNode*)malloc(sizeof(structNode));

head->data = 1; head->next = second;

second->data = 2; second->next = third

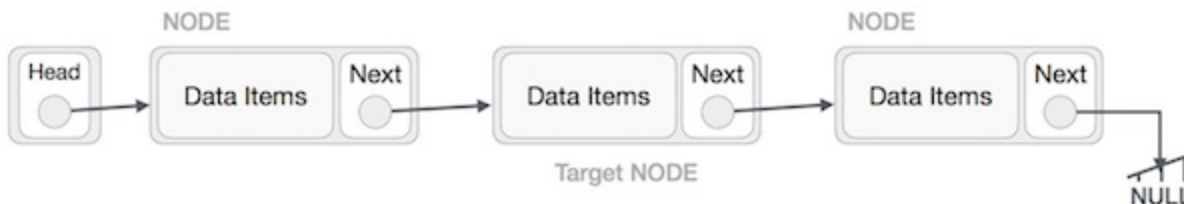
third->data = 3; third->next = NULL;

return 0; }

```

## Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



The left (previous) node of the target node now should point to the next node of the target node –

```
LeftNode.next -> TargetNode.next;
```

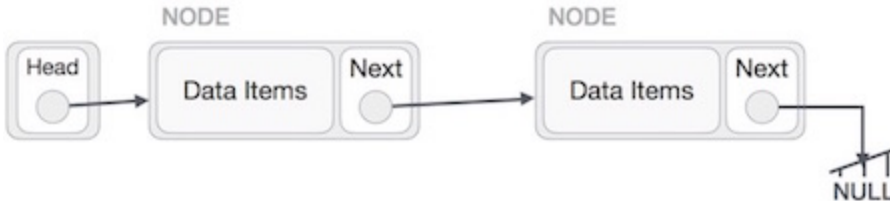


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next -> NULL;
```



We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



```
// A complete working C program to demonstrate deletion in singly // linked list
#include <stdio.h> #include <stdlib.h> structNode { intdata; structNode *next; };

void push(structNode** head_ref, intnew_data) { structNode* new_node =
(structNode*)malloc(sizeof(structNode)); new_node->data = new_data; new_node->next
= (*head_ref); (*head_ref) = new_node; }

void deleteNode(structNode **head_ref, intkey) { structNode* temp = *head_ref,
*prev; if(temp != NULL && temp->data == key) { *head_ref = temp->next;
free(temp); return; }

while(temp != NULL && temp->data != key) { prev = temp; temp = temp->next; }

if(temp == NULL) return; prev->next = temp->next; free(temp); }

void printList(structNode *node) { while(node != NULL) { printf(" %d ", node->data);
node=node->next; } }

int main() { structNode* head = NULL; push(&head, 7); push(&head, 1); push(&head,
3); push(&head, 2);

puts("Created Linked List: "); printList(head); deleteNode(&head, 1); puts("\nLinked
List after Deletion of 1: "); printList(head); return0;

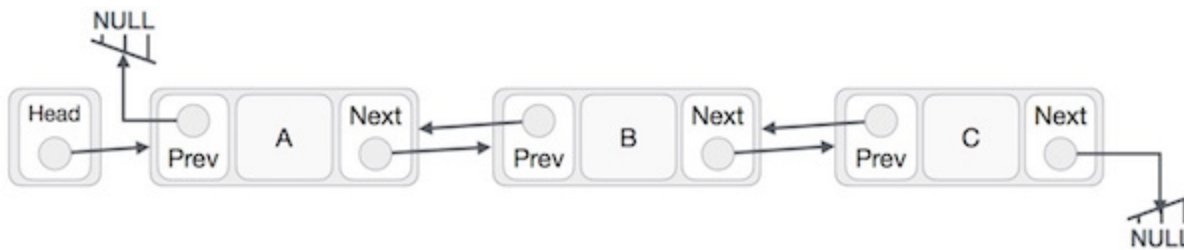
}
```

## Doubly Linked List

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **Prev** – Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** – A Linked List contains the connection link to the first link called First and to the last link called Last.

## Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

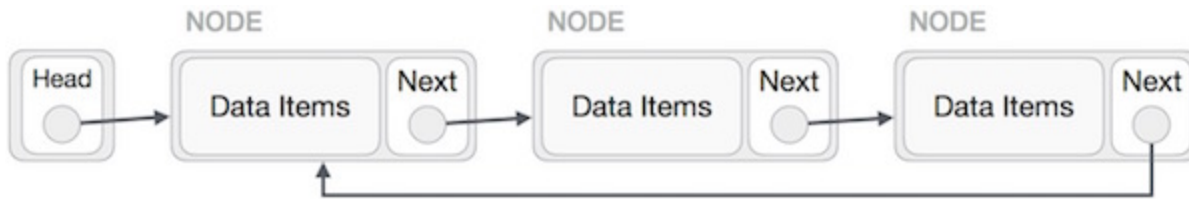
- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

## Circular Linked List

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

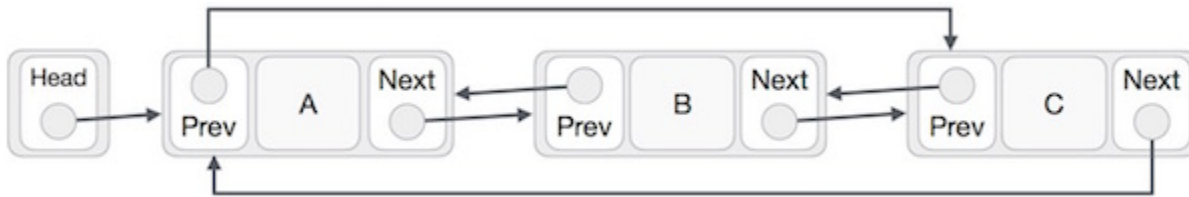
## Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



## Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



As per the above illustration, following are the important points to be considered.

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

## Header Linked List in C

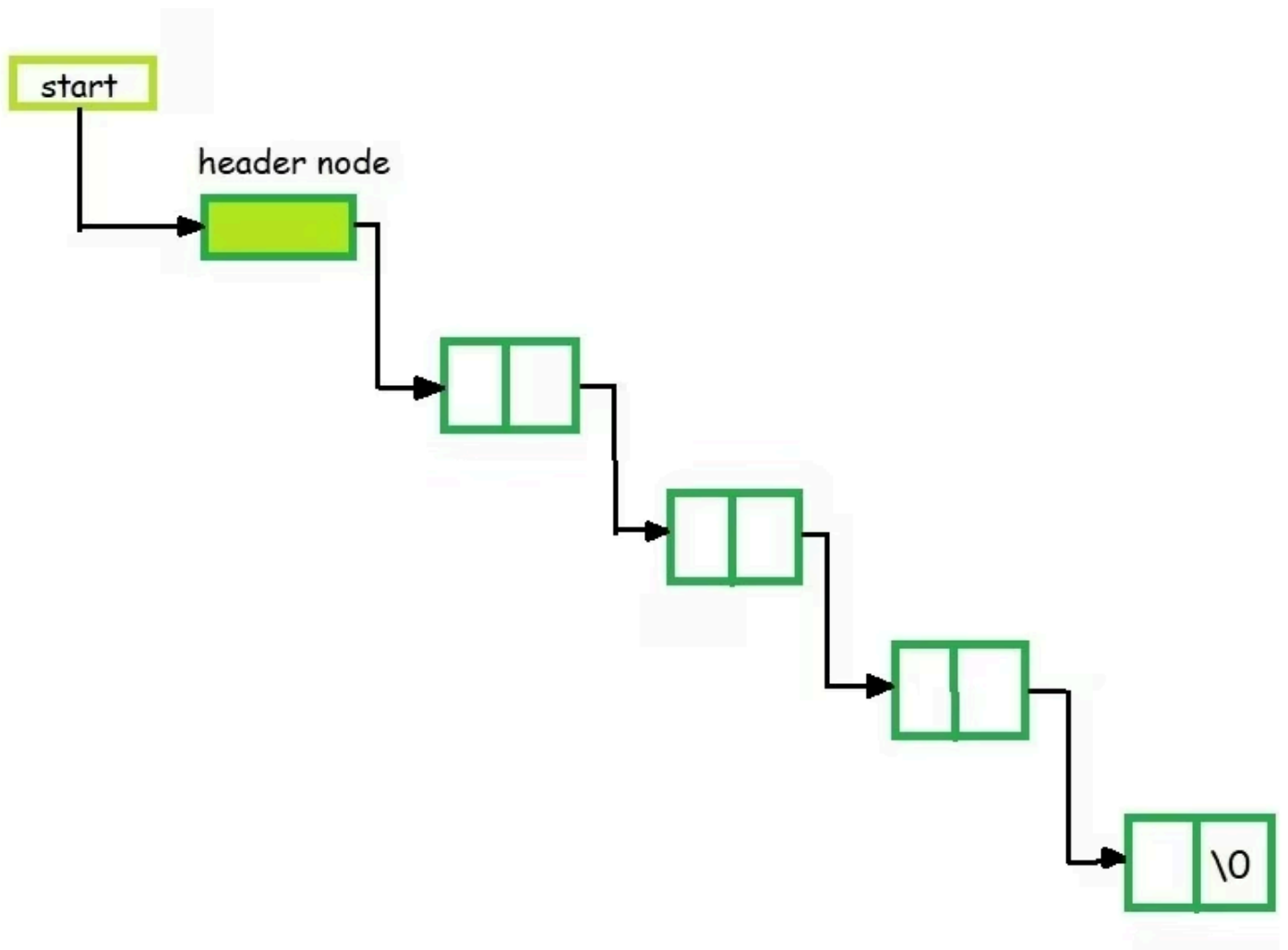
A *header node* is a special node that is found at the *beginning* of the list. A list that contains this type of node, is called the header-linked list. This type of list is useful when information other than that found in each node is needed.

For example, suppose there is an application in which the number of items in a list is often calculated. Usually, a list is always traversed to find the length of the list. However, if the current length is maintained in an additional header node that information can be easily obtained.

### Types of Header Linked List

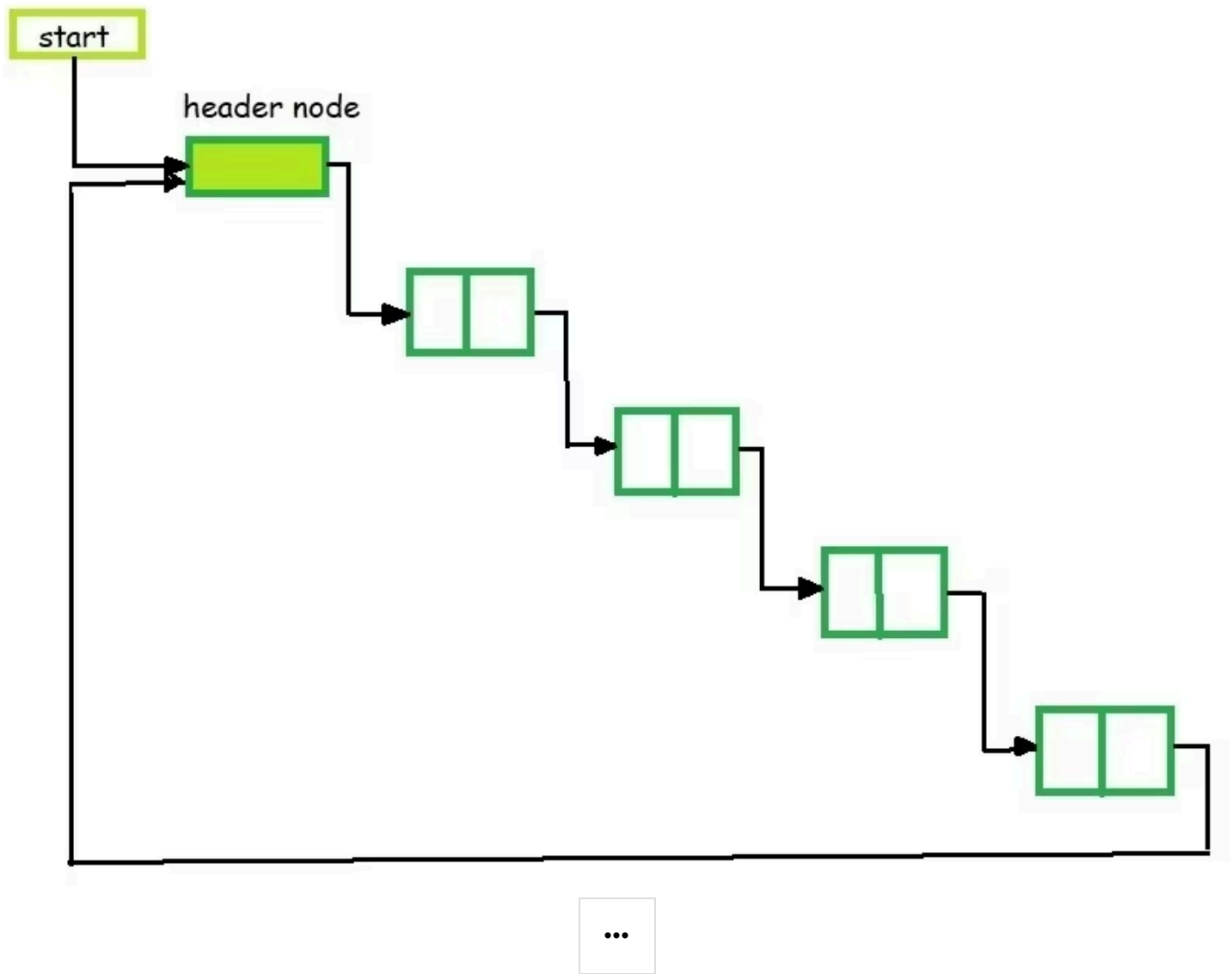
#### 1. Grounded Header Linked List

It is a list whose *last node* contains the *NULL* pointer. In the header linked list the **start** pointer always points to the header node. **start -> next = NULL** indicates that the grounded header linked list is *empty*. The operations that are possible on this type of linked list are *Insertion, Deletion, and Traversing*.



### 1. Circular Header Linked List

A list in which *last node* points back to the *header node* is called circular linked list. The chains do not indicate first or last nodes. In this case, external pointers provide a frame of reference because last node of a circular linked list does **not contain** the **NULL** pointer. The possible operations on this type of linked list are *Insertion, Deletion and Traversing*.



A WordPress.com Website.



# B.C.A study

## Unit-5:B-Tree

### Introduction to B TREE and its operations, **invention of B- tree**

A **B tree** is designed to store sorted data and allows search, insertion and deletion operation to be performed in logarithmic time. As In multiway search tree, there are so many nodes which have left subtree but no right subtree. Similarly, they have right subtree but no left subtree. As is known, access time in the tree is totally dependent on the level of the tree. So our aim is to minimize the access time which can be through balance tree only.

For balancing the tree each node should contain  $n/2$  keys. So the B tree of order  $n$  can be defined as:

1. All leaf nodes should be at same level.
2. All leaf nodes can contain maximum  $n-1$  keys.
3. The root has at least two children.
4. The maximum number of children should be  $n$  and each node can contain  $k$  keys. Where,  $k \leq n-1$ .
5. Each node has at least  $n/2$  and maximum  $n$  nonempty children.
6. Keys in the non-leaf node will divide the left and right sub-tree where the value of left subtree keys will be less and value of right subtree keys will be more than that particular key.

### Operations performed on B Tree

1. **Insertion in B-Tree**
2. **Deletion from B-Tree**

## 1) Insertion in B-Tree

The insertion of a key in a B tree requires the first traversal in B-tree. Through the traversal, it is easy to find that key which needs to be inserted is already existed or not. There are basically two cases for inserting the key that are:

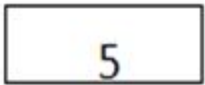
1. Node is not full
2. Node is already full

If the leaf node in which the key is to be inserted is not full, then the insertion is done in the node.

If the node were to be full then insert the key in order into existing set of keys in the node, split the node at its median into two nodes at the same level, pushing the median element up by one level.

**Let us take a list of keys and create a B-Tree: 5,9,3,7,1,2,8,6,0,4**

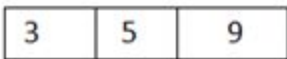
### 1) Insert 5



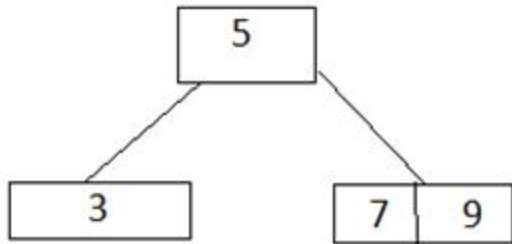
**2) Insert 9:** B-tree insert simply calls B tree insert non-full, putting 9 to the right of 5.



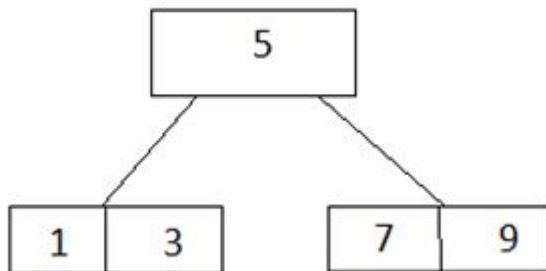
**3) Insert 3:** Again B-tree insert non-full is called



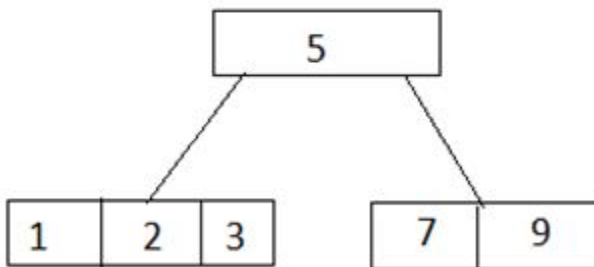
**4) Insert 7:** Tree is full. We allocate a new empty node, make it the root, split a former root, and then pull 5 into a new root.



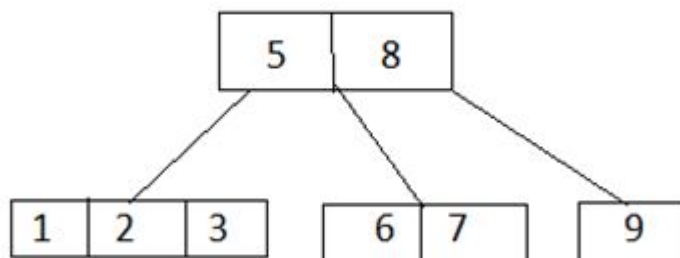
5) **Insert 1:** It goes with 3



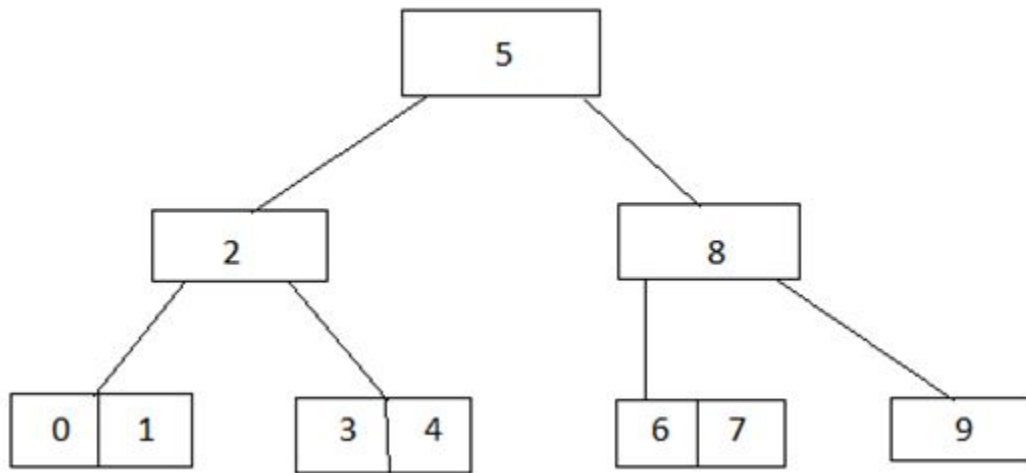
6) **Insert 2:** It goes with 3



7) **Insert 8, 6:** As firstly 8 goes with the 9 and then 6 would go with 7, 8, 9 but that node is full. So we split it bring its middle child into the root.



**8) Insert 0, 4:** 0 would go with the 1, 2, and 3 which are full, so we split it sending the middle child up to the root. Now it would be nice to just stick 4 in with 3, but the B-tree algorithm requires us to split the full root. Now we can insert 4 assured that future insertion will work.



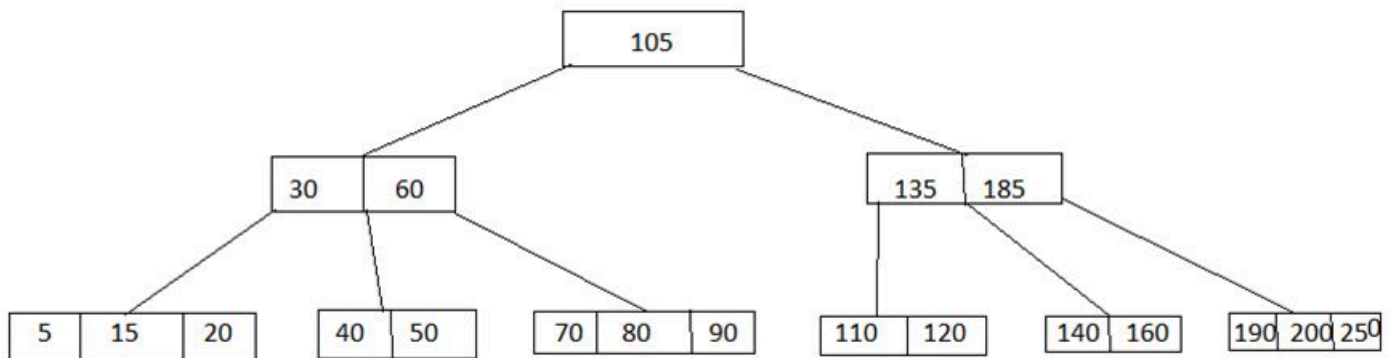
## 2) Deletion from B Tree

Let us take a B-tree of order 5,

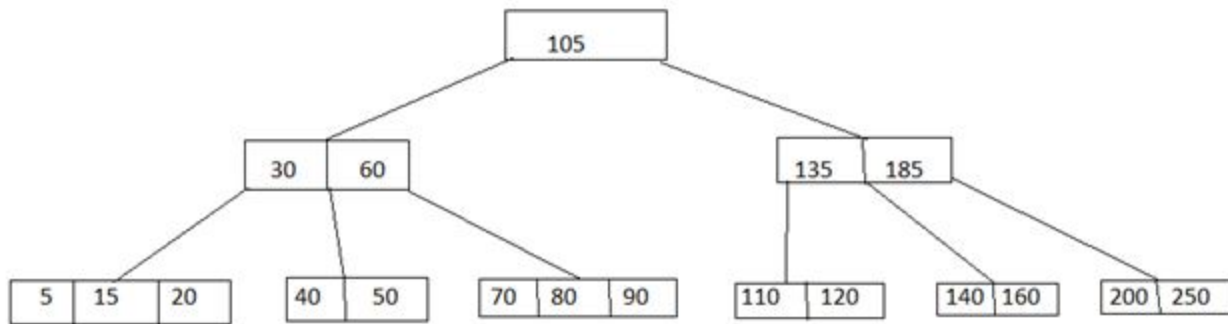
Deletion of the key also requires the first traversal in B tree, after reaching on a particular node two cases may be occurred that are:

1. Node is leaf node
2. Node is non leaf node

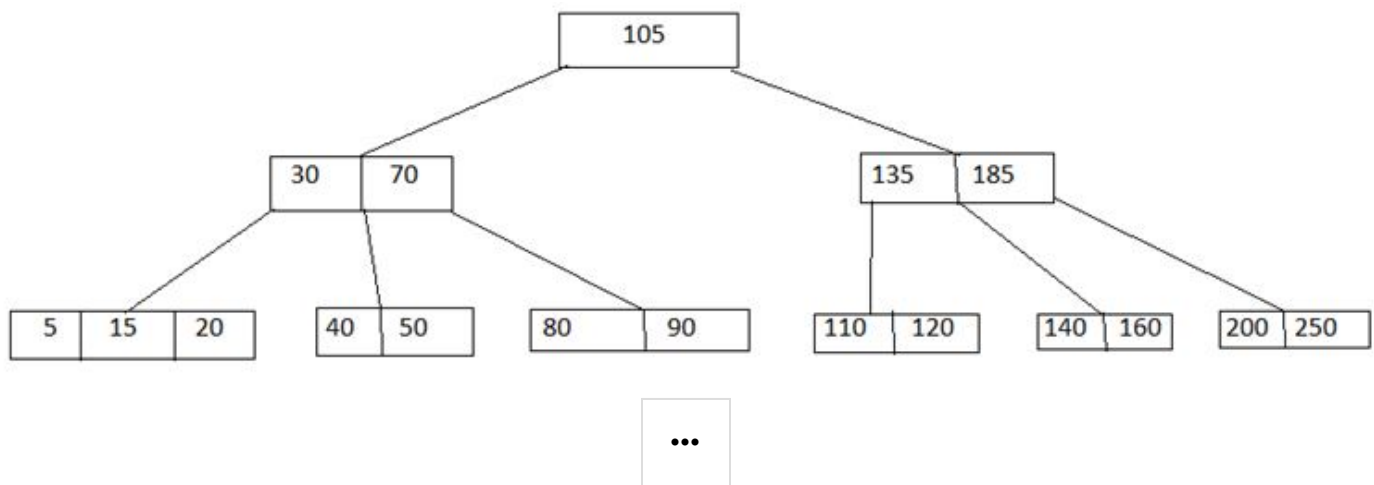
**Example: Let us take a B tree of order 5**



**1) Delete 190:** Here 190 is in leaf node, so delete it from only leaf node.



**2) Delete 60:** Here 60 is in non leaf node. So first it will be deleted from the node and then the element of the right child will come in that node.



[A WordPress.com Website.](https://bcastudyguide.com/unit-5-b-tree/)

# B.C.A study

## Unit-6:Sorting Technique

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

### 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?

We take an unsorted array for our example.



Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.



It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



These values are not in a sorted order.



So we swap them.



However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

## Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve 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.



This algorithm is not suitable for large data sets as its average and worst case complexities are of  $O(n^2)$ , where  $n$  is the number of items.

## How Selection Sort Works?

Consider the following depicted array as an example.



For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



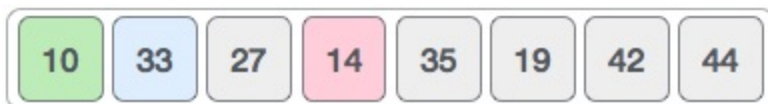
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process –



Now, let us learn some programming aspects of selection sort.

# Algorithm

- Step 1** - Set MIN to location 0
- Step 2** - Search the minimum element in the list
- Step 3** - Swap with value at location MIN
- Step 4** - Increment MIN to point to next element
- Step 5** - Repeat until list is sorted

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being  $O(n \log n)$ , it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

## How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following –



We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

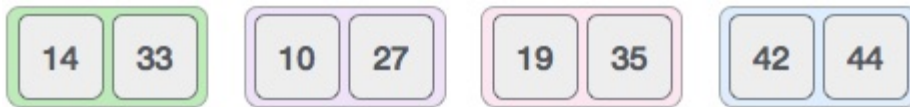


We further divide these arrays and we achieve atomic value which can no more be divided.



Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this –



Now we should learn some programming aspects of merge sorting.

## Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

- Step 1** – if it is only one element in the list it is already sorted, return.
- Step 2** – divide the list recursively into two halves until it can no more be divided.
- Step 3** – merge the smaller lists into new list in sorted order.

# What is Searching?

- Searching is the process of finding a given value position in a list of values.
- It decides whether a search key is present in the data or not.
- It is the algorithmic process of finding a particular item in a collection of items.
- It can be done on internal data structure or on external data structure.

## Searching Techniques

To search an element in a given array, it can be done in following ways:

1. Sequential Search
2. Binary Search

### 1. Sequential Search

- Sequential search is also called as Linear Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

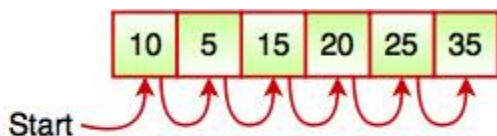


Fig. Sequential Search

The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

### Binary Search

- Binary Search is used for searching an element in a sorted array.

- It is a fast search algorithm with run-time complexity of  $O(\log n)$ .
- Binary search works on the principle of divide and conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.
- It is useful when there are large number of elements in an array.
- The above array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.

**For example**, if searching an element 25 in the 7-element array, following figure shows how binary search works:

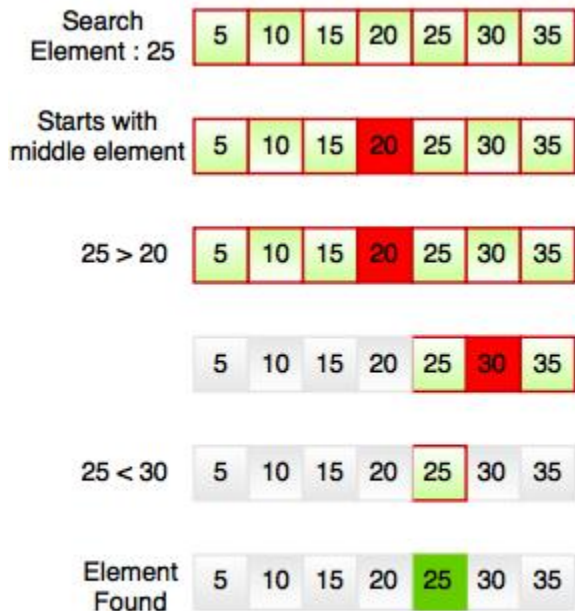


Fig. Working Structure of Binary Search

Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the left of the list. Repeat this, till you find an element.

## What is Hashing?

- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- Hashing is also known as **Hashing Algorithm** or **Message Digest Function**.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary search.
- Hashing allows to update and retrieve any data entry in a constant time  $O(1)$ .
- Constant time  $O(1)$  means the operation does not depend on the size of the data.

- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

## What is Hash Function?

- A fixed process converts a key to a hash key is known as a **Hash Function**.
- This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash**.
- Hash value represents the original string of characters, but it is normally smaller than the original.
- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

## What is Hash Table?

- Hash table or hash map is a data structure used to store key-value pairs.
- It is a collection of items stored to make it easy to find them later.
- It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
- It is an array of list where each list is known as bucket.
- It contains value based on the key.
- Hash table is used to implement the map interface and extends Dictionary class.
- Hash table is synchronized and contains only unique elements.
- The above figure shows the hash table with the size of  $n = 10$ . Each position of the hash table is called as **Slot**. In the above hash table, there are  $n$  slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.
- As we know the mapping between an item and the slot where item belongs in the hash table is called the hash function. The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to  $n-1$ .
- Suppose we have integer items {26, 70, 18, 31, 54, 93}. One common method of determining a hash key is the division method of hashing and the formula is :

Hash Key = Key Value % Number of Slots in the Table



- Division method or remainder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	$26 \% 10 = 6$	6
70	$70 \% 10 = 0$	0
18	$18 \% 10 = 8$	8
31	$31 \% 10 = 1$	1
54	$54 \% 10 = 4$	4
93	$93 \% 10 = 3$	3

0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

Fig. Hash Table

- After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure. In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by,  $\lambda = \text{No. of items} / \text{table size}$ . For example,  $\lambda = 6/10$ .



- It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.
- Constant amount of time  $O(1)$  is required to compute the hash value and index of the hash table at that location.



**[A WordPress.com Website.](#)**