Data Structure Lab

MCA- 207

SELF LEARNING MATERIAL

DIRECTORATE

OF DISTANCE EDUCATION

SWAMI VIVEKANAND SUBHARTI UNIVERSITY

MEERUT – 250 005,

UTTAR PRADESH (INDIA)

SLM Module Developed By :

Author:

Reviewed by :

Assessed by:

Study Material Assessment Committee, as per the SVSU ordinance No. VI (2)

Published by: Gayatri Sales

Typeset at: Micron Computers **Printed at:** Gayatri Sales, Meerut.

DATA STRUCTURE LAB

Write Program in C or C++ for following:

• Sorting programs: Bubble sort, Merge sort, Insertion sort, Selection sort, and Quick sort.

• Searching programs: Linear Search, Binary Search.

• Array implementation of Stack, Queue, Circular Queue, Linked List.

• Implementation of Stack, Queue, Circular Queue, Linked List using dynamic memory allocation.

• Implementation of Binary tree.

• Program for Tree Traversals (preorder, inorder, postorder).

• Program for graph traversal (BFS, DFS).

• Program for minimum cost spanning tree, shortest path.

Unit -I

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, Artifical 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 arrise the following problems:

Processor speed:

To handle very large amout 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 perticular 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 herierchy 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 classfied 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

Elementary Data Organization

In this article, we'll take a look about all the basic terms used in data structure. These terms are the building blocks of data structure.

1. Data

Data can be defined as a representation of facts, concepts or instructions in a formalized manner suitable for communication, interpretation or processing by human or electric machines. Data is represented with the help of characters like alphabets (A-Z,a-z), digits (0-9) and special characters (+,-,*,<, >, =, ^ etc.).

2. Data Item (Field)

A set of characters which are used together to represent a specific data element e.g. name of a student in a class is represented by the data item, say NAME.
The data items can be classified into two types depending on the usage:

1. **Elementary Data Items:**

These data items can't be further sub-divided.
For e.g. ROLL NUMBER.

2. **Group Data Items:**

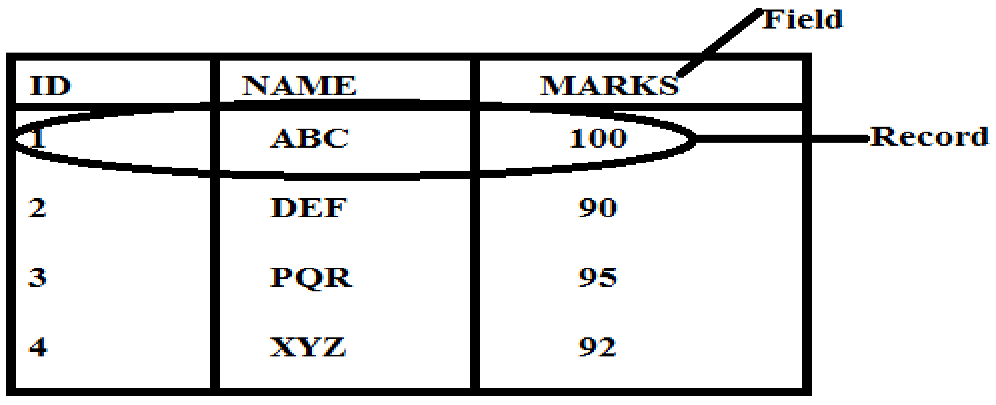
These data items can be further sub-divided into elementary data items.
For example: DATE may be divided into Days, Months and Years.

3. Record

Record is a collection of related data items. E.g. a student record for a student's contains data fields such as name, age, sex, class etc.

4. File (Data File)

File is collection of logically related records. E.g. a payroll file might consist of the employee pay records for a company.



ID	NAME	MARKS
1	ABC	100
2	DEF	90
3	PQR	95
4	XYZ	92

Student File

5. Entity:

An entity is a person, place, thing, event or concept about which information recorded.

6. Attribute:

Attributes gives the characteristics or properties of the entity.

7. Data Value:

A data value is the actual data or information contained in each attribute.

Example:

- Entity : Employee
- Attributes: Id, Name, Age
- Value: 1, ABC, 18

8.Entity Set

Entities with similar attributes form an entity set. For example, All the employees in an organization form an entity set.

9. Primary and Secondary Keys:

Primary Key: A field or collection of fields in a record which identifies a record uniquely is called a primary key or simply key. In a student file: ID is primary key.
Secondary Key: A field in a record which identifies the records but not uniquely is called a secondary key. For Example, NAME and MARKS are the secondary keys in Student file.

Data Structure operations

Data Structure is the way of storing data in computer’s memory so that it can be used easily and efficiently. There are different data-structures used for the storage of data. It can also be define as a mathematical or logical model of a particular organization of data items. The representation of particular data structure in the main memory of a computer is called as storage structure. **For Examples:** Array, Stack, Queue, Tree, Graph, etc.

Operations on different Data Structure:

There are different types of operations that can be performed for the manipulation of data in every data structure. Some operations

are explained and illustrated below:

- **Traversing:**

  Traversing a Data Structure means to visit the element stored in it. This can be done with any type of DS.
  Below is the program to illustrate traversal in an array:

```
// C++ program to traversal in an array
#include <iostream>
using namespace std;

// Driver Code
int main()
{
   // Initialise array
   int arr[] = { 1, 2, 3, 4 };

   // size of array
   int N = sizeof(arr) / sizeof(arr[0]);

   // Traverse the element of arr[]
   for (int i = 0; i < N; i++) {

      // Print the element
      cout << arr[i] << ' ';
   }

   return 0;
}
```

- **Output:**
- 1 2 3 4

Searching: Searching means to find a particular element in the given data-structure. It is considered as successful when the required element is found. Searching is the operation which we can performed on data-structures like array, linked-list, tree, graph, etc.

- Below is the program to illustrate searching an element in an array:

```
// C++ program to searching in an array
#include <iostream>
using namespace std;
```

```
// Function that finds element K in the

// array

void findElement(int arr[], int N, int K)
{

    // Traverse the element of arr[]
    // to find element K
    for (int i = 0; i < N; i++) {

        // If Element is present then
        // print the index and return
        if (arr[i] == K) {
            cout << "Element found!";
            return;
        }
    }

    cout << "Element Not found!";
}

// Driver Code
int main()
{
    // Initialise array
    int arr[] = { 1, 2, 3, 4 };

    // Element to be found
    int K = 3;

    // size of array
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function Call
    findElement(arr, N, K);
    return 0;
}
```

- **Output:**
- Element found!

Insertion: It is the operation which we apply on all the data-structures. Insertion means to add an element in the given data structure. The operation of insertion is successful when the required element is added to the required data-structure. It is unsuccessful in

some cases when the size of the data structure is full and when there is no space in the data-structure to add any additional element. The insertion has the same name as an insertion in the data-structure as an array, linked-list, graph, tree. In stack, this operation is called Push. In the queue, this operation is called Enqueue.
Below is the program to illustrate insertion in stack:

```
// C++ program for insertion in array
#include <iostream>
using namespace std;

// Function to print the array element
void printArray(int arr[], int N)
{
   // Traverse the element of arr[]
   for (int i = 0; i < N; i++) {

      // Print the element
      cout << arr[i] << ' ';
   }
}

// Driver Code
int main()
{
   // Initialise array
   int arr[4];

   // size of array
   int N = 4;

   // Insert elements in array
   for (int i = 1; i < 5; i++) {
      arr[i - 1] = i;
   }

   // Print array element
   printArray(arr, N);
   return 0;
}
```

Output:

```
1 2 3 4
```

Deletion: It is the operation which we apply on all the data-structures. Deletion means to delete an element in the given data structure. The operation of deletion is successful when the required element is deleted from the data structure. The deletion has the same name as a deletion in the data-structure as an array, linked-list, graph, tree, etc. In stack, this operation is called Pop. In Queue this operation is called Dequeue. Below is the program to illustrate dequeue in Queue:

```
// C++ program for insertion in array
#include <bits/stdc++.h>
using namespace std;

// Function to print the element in stack
void printStack(stack<int> St)
{
   // Traverse the stack
   while (!St.empty()) {

      // Print top element
      cout << St.top() << ' ';

      // Pop top element
      St.pop();
   }
}

// Driver Code
int main()
{
   // Initialise stack
   stack<int> St;

   // Insert Element in stack
   St.push(4);
   St.push(3);
   St.push(2);
   St.push(1);

   // Print elements before pop
   // operation on stack
   printStack(St);

   cout << endl;

   // Pop the top element
   St.pop();

   // Print elements after pop
```

```
    // operation on stack
    printStack(St);
    return 0;
}
```

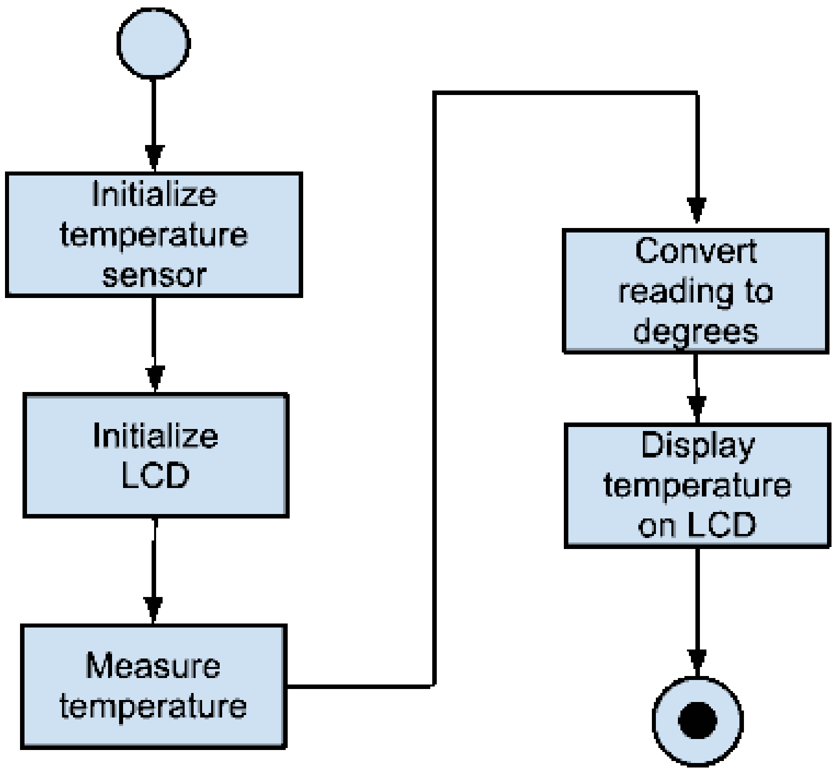
Output:

```
1 2 3 4

2 3 4
```

Algorithm Complexity and Time-Space trade-off

An algorithm is a well defined list of steps for solving a particular problem. Starting from an initial state, an algorithm states the instructions needed to describe a computation that proceeds through a well-defined series of successive states, eventually terminating in a final ending state. There may be more than one algorithm to solve a particular problem.

Given below is an algorithm gives the required steps required by a temperature sensor to measure and display temperature on an attached LED screen.


Initialize temperature sensor
Initialize LCD
Measure temperature
Convert reading to degrees
Display temperature on LCD

Complexity And Space-Time Tradeoff

The complexity of an algorithm is the function which gives the running time and/ or space in term of input size. In simple words, the complexity of an algorithm refers to how fast or slow a particular algorithm performs. We define complexity as a numerical function T(n) - time versus the input size n.

Further, each of our algorithm will involve a particular data structure. Accordingly we may not always be able to use the most efficient algorithm, since the choice of data structure depends on many things including the type of data and the frequency with which various data operations are applied.

How To Find The Complexity Of An Algorithm?

Let us understand this with the help of an example. Suppose we are implementing an algorithm that helps us to search for an record amongst a list of records. We can have the following three cases which relate to the relative success our algorithm can achieve with respect to time:

- Best Case: The record we are trying to search is the first record of the list. If f(n) is the function which gives the running time and/ or storage space requirement of the algorithm in terms of the size n of the input data, this particular case of the algorithm will produce a complexity C(n)=1 for our algorithm f(n) as the algorithm will run only 1 time until it finds the desired record.

- Worst Case: The record we are trying to search is the last record of the list. If f(n) is the function which gives the running time and/ or storage space requirement of the algorithm in terms of the size n of the input data, this particular case of the algorithm will produce a complexity C(n)=n for our algorithm f(n) as the algorithm will run n times until it finds the desired record.

- Average Case: The record we are trying to search can be any record in the list. In this case we do not know at which position it might be. Hence we take an average of all the possible times our algorithm may run. Hence assuming for n data, we have a probability of finding any one of them is 1/n. Multiplying each of these with the number of times our algorithm might run for finding each of them and then taking a sum of all those multiples, we can obtain the complexity C(n) for our algorithm f(n) in case of an average case as following:

$$C(n) = 1.\frac{1}{n} + 2.\frac{1}{n} + \ldots + n.\frac{1}{n}$$

$$C(n) = (1 + 2 + \ldots + n).\frac{1}{n}$$

$$C(n) = \frac{n(n+1)}{2}.\frac{1}{n} = \frac{n+1}{2}$$

Hence in this way, we can find the complexity of an algorithm.

The time and space it uses are two major measures of the efficiency of an algorithm. Sometimes the choice of data structure involves a space-time tradeoff; by increasing the amount of space for storing the data one may be able to reduce the time needed for processing the data or vice versa. Hence let us look over them in detail:

- Space Complexity: It is also known as memory requirement. The space complexity of an algorithm is the amount of memory it needs to run to completion. We would usually want our algorithm to take the least possible memory for operation, however in more powerful machines more resources are usually allocated for the operation in order to reduce the time taken.

- Time Complexity: It is also known as performance requirement. Time Complexity is calculated of referred in instances when we may be interested to know in advance whether the program will provide a satisfactory real time response or not. There may be several possible solutions to a problem with different time requirements or with different time complexity. Time complexity is heavily taken care of in cases when an algorithm needs to be modeled to be run on even the least powerful machines.

How To Balance The Two Then?

The best algorithm to solve a given problem is one that requires less space in memory and takes less time to complete its execution. But in practice it is not always possible to achieve both these objectives. As we know there may be more then one approach to solve a particular problem. One approach may take more space but takes less time to complete its execution while the other approach may take less space but takes more time to complete its execution. We may have to sacrifice one at the cost of the other. If space is our constraint, then we have to choose a program that requires less space at the cost of more execution time. On the other hand if time is our constraint then we have to choose a program that takes less time to complete its execution at the cost of more space.

And with that, we finish our today's lesson. I hope you learnt something with that. Stay tuned for more and take care :)

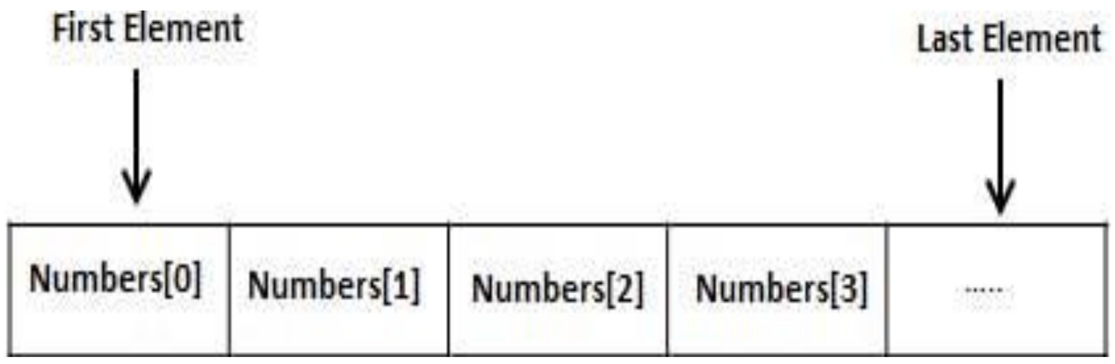
Arrays:

Array Definition

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you 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.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type arrayName [ arraySize ];
```

This is called a single-dimensional array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

```
double balance[10];
```

Here balance is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

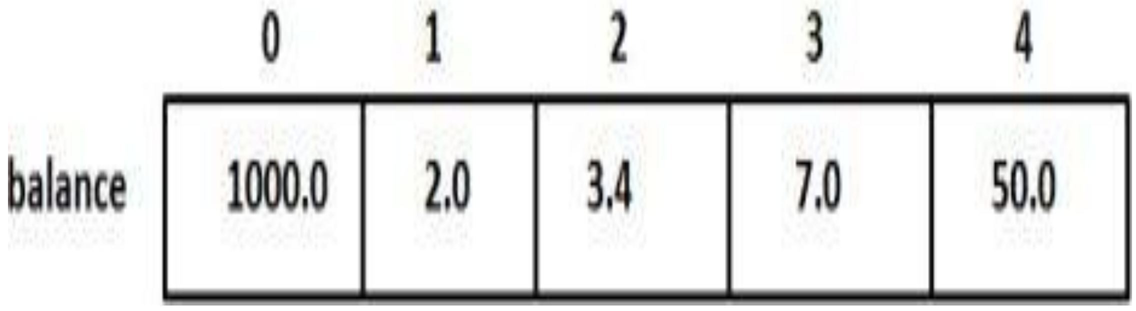
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

balance[4] = 50.0;

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above −



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays –

```
#include<stdio.h>

int main (){

int n[10];/* n is an array of 10 integers */
int i,j;

/* initialize elements of array n to 0 */
for( i =0; i <10; i++){
     n[ i ]= i +100;/* set element at location i to i + 100 */
}

/* output each array element's value */
for(j =0; j <10; j++){
     printf("Element[%d] = %d\n", j, n[j]);
}

return0;
}
```

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

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Arrays in Detail

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer −

Sr.No.	**Concept & Description**
1	Multi-dimensional arrays C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	Passing arrays to functions You can pass to the function a pointer to an array by specifying the array's name without an index.

3	Return array from a function C allows a function to return an array.
4	Pointer to an array You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

Representation and Analysis

In this section we will see another representation of multidimensional arrays. Here we will see the Array of Arrays representation. In this form, we have an array, that is holding the starting addresses of multiple arrays. The representation will be look like this.

This is a two-dimensional array x of size [7 x 8]. Each row is represented as a single onedimensional array. The initial array is holding the addresses of these single arrays. They are array of addresses, so we can say that, it is an array of pointers. Each pointer is holding addresses of another arrays.

create this kind of array, we can use the new keyword like below −

```
int [][] x = new int[7][8];
```

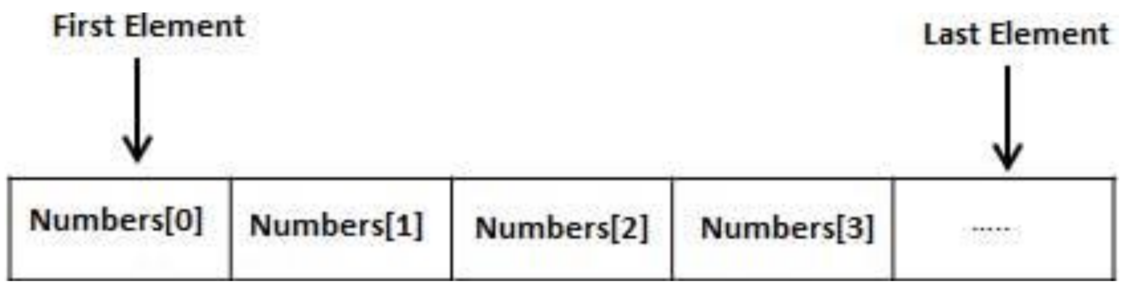
To retrieve an element present at position x[i, j], it will find the address using x[i] at first, then move to jth index in that array.

Single and Multidimensional Arrays

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you 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.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type arrayName [ arraySize ];
```

This is called a single-dimensional array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

```
double balance[10];
```

Here balance is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

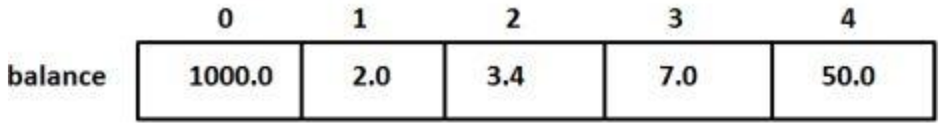
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above −



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays –

```
#include<stdio.h>

int main (){

int n[10];/* n is an array of 10 integers */
int i,j;

/* initialize elements of array n to 0 */
for( i =0; i <10; i++){
      n[ i ]= i +100;/* set element at location i to i + 100 */
}
/* output each array element's value */
for(j =0; j <10; j++){
      printf("Element[%d] = %d\n", j, n[j]);
}

return0;
}
```

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

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Arrays in Detail

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer −

Sr.No.	Concept & Description
1	Multi-dimensional arrays C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	Passing arrays to functions You can pass to the function a pointer to an array by specifying the array's name without an index.
3	Return array from a function C allows a function to return an array.
4	Pointer to an array You can generate a pointer to the first element of an array by simply

	specifying the array name, without any index.

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration −

type name[size1][size2]...[sizeN];

For example, the following declaration creates a three dimensional integer array −

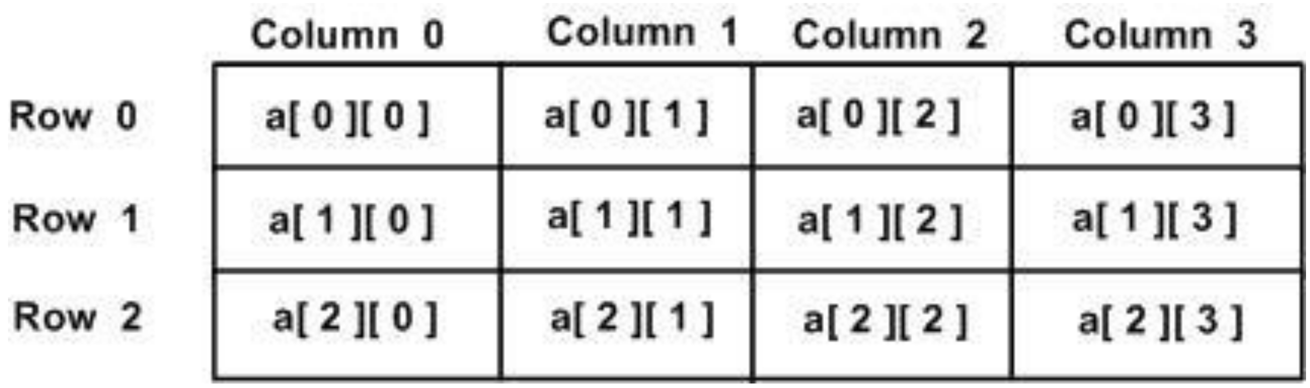
int threedim[5][10][4];

Two-dimensional Arrays

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows −

type arrayName [x][y];

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array **a**, which contains three rows and four columns can be shown as follows −



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4]={
{0,1,2,3},/*  initializers for row indexed by 0 */
{4,5,6,7},/*  initializers for row indexed by 1 */
{8,9,10,11}/*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example −

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example −

```
int val = a[2][3];
```

The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure. Let us check the following program where we have used a nested loop to handle a two-dimensional array −

```

#include<stdio.h>

int main (){

/* an array with 5 rows and 2 columns*/
int a[5][2]={{0,0},{1,2},{2,4},{3,6},{4,8}};
int i, j;

/* output each array element's value */
for( i =0; i <5; i++){

for( j =0; j <2; j++){
        printf("a[%d][%d] = %d\n", i,j, a[i][j]);
}
}

return0;
}
```

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

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

address calculation

An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the subscript). The subscript is an ordinal number which is used to identify an element of the array.

Calculating the Address of 1D Array Elements

- The array name is a symbolic reference to the address of the first byte of the array. When we use the array name, we are actually referring to the first byte of the array.

- The subscript or the index represents the offset from the beginning of the array to the element being referenced. That is, with just the array name and the index, C can calculate the address of any element in the array.

- Since an array stores all its data elements in consecutive memory locations, storing just the base address, that is the address of the first element in the array, is sufficient. The address of other data elements can simply be calculated using the base address.
- The formula to perform this calculation is,

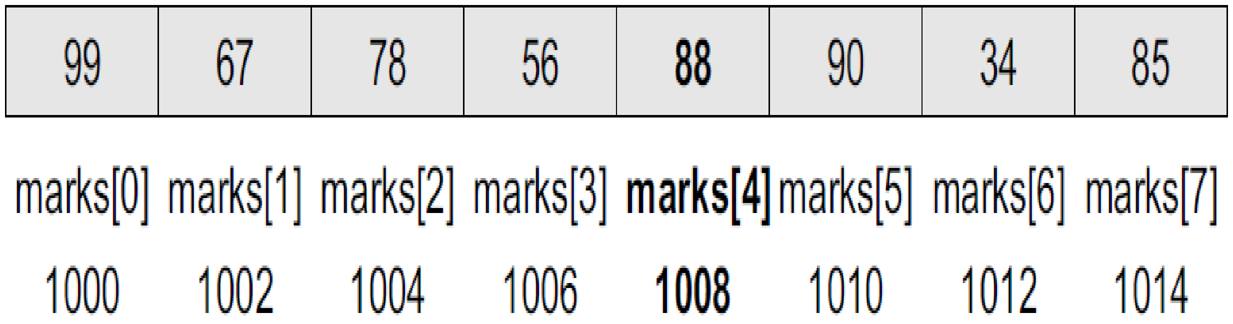
Address of A[k] = Base_Address(A) + w(k – Lower_Bound)

Here, A is the array, k is the index of the element of which we have to calculate the address, Base_Address is the base address of the array A, and w is the size of one element in memory, for example, size of int is 2 bytes.

Address calculation in 1d array : Example

Example 1 : Given an array int marks[] = {99,67,78,56,88,90,34,85}, calculate the address of marks[4] if the base address = 1000.

Solution :



We know that storing an integer value requires 2 bytes, therefore, its size is 2 bytes.

Address of A[k] = Base_Address(A) + w(k – Lower_Bound)

marks[4] = 1000 + 2(4 – 0)
= 1000 + 2(4)
= 1008 **[Ans]**

Example 2 : Base address of an array **B[1300 : 1900]** as 1020 and size of each element of array is 2 bytes in the memory. Find the address of **B[1700].**

Solution :

The given values are: Base_Address = 1020, Lower_Bound = 1300, w = 2, k = 1700

Address of A[k] = Base_Address(A) + w(k – Lower_Bound)

B[1700] = 1020 + 2 * (1700 – 1300)
= 1020 + 2 * 400
= 1020 + 800
= 1820 **[Ans]**

Calculating the Length of an Array

The length of an array is given by the number of elements stored in it. The general formula to calculate the length of an array is

Length = upper_bound – lower_bound + 1

where upper_bound is the index of the last element and lower_bound is the index of the first element in the array.

application of arrays

- Arrays are used to implement mathematical vectors and matrices, as well as other kinds of rectangular tables. Many databases, small and large, consist of one-dimensional arrays whose elements are records.

- Arrays are used to implement other data structures, such as lists, heaps, hash tables, deques, queues and stacks.

- 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. The array may contain subroutine pointers(or relative subroutine numbers that can be acted upon by SWITCH statements) that direct the path of the execution.

Character String in C

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

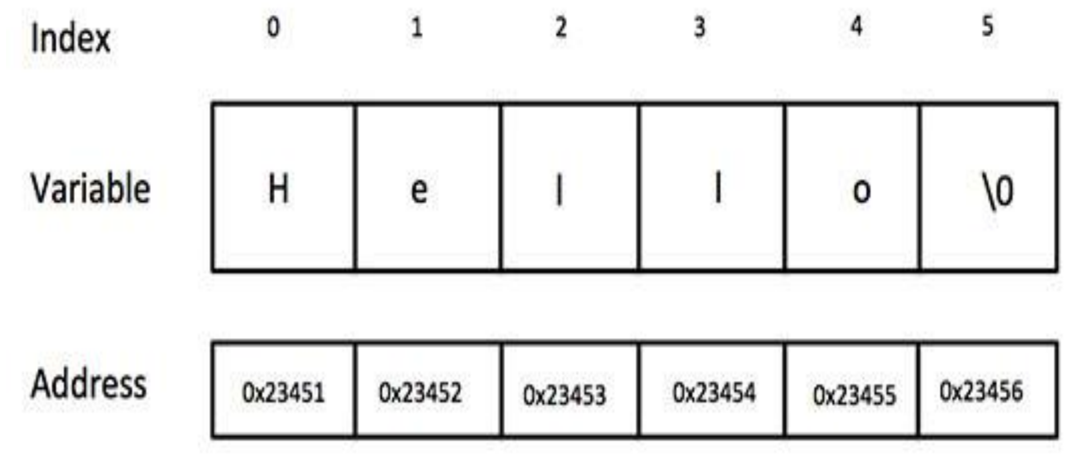
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows −

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C/C++ −



Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

```
#include <stdio.h>

int main () {

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
printf("Greeting message: %s\n", greeting );
return 0;
}
```

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

```
Greeting message: Hello
```

C supports a wide range of functions that manipulate null-terminated strings –

Sr.No.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);**

	Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions –

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

int main () {

char str1[12] = "Hello";
char str2[12] = "World";
char str3[12];
int  len ;

/* copy str1 into str3 */
strcpy(str3, str1);
printf("strcpy( str3, str1) :  %s\n", str3 );

/* concatenates str1 and str2 */
strcat( str1, str2);
printf("strcat( str1, str2):   %s\n", str1 );

/* total lenghth of str1 after concatenation */
len = strlen(str1);
printf("strlen(str1) :  %d\n", len );
return 0;
}
```

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

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

Character string operation

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

include this library to your program to use some of the inbuilt string manipulation functions present in the library.

there are about 20-22 inbuilt string manipulating functions present in this library.
The most common functions used from the library are:
1. strlen("name of string")
2. strcpy(dest, source)
3. strcmp(string1, string2)
4. strstr(str1, str2)

strlen(string)

Declaration and usage of strlen() function

It is one of the most useful functions used from this library, whenever we need the length of the string say "str1"
we write it as

```
int len;
```

```
len = strlen(str1);
```

len will be the length of the string "str1".(it will include all the spaces and characters in the string except the NULL('\0') character.

strcpy(dest, source)

This function copies the string "source" to string "dest".
Declaration

```
strcpy( str2, str1 );
```

Return value
This returns a pointer to the destination string dest.

Example
Below code shows the usage of strcpy() function.

```
#include<stdio.h>
#include<string.h>
int main( )
{
   char str1[ ] = "I love programming" ;
   char str2[20]= "" ;
   printf ( "source string(i.e str1) = %s\n", str1 ) ;
   printf ( "dest string(i.e str2) = %s\n", str2) ;
   strcpy ( str2, str1 ) ;
   printf ( "target( i.e str2) string after strcpy( ) = %s\n", str2 ) ;
   return 0;
}
```

Output

```
source string(i.e str1) = I love programming
dest string(i.e str2) =
target( i.e str2) string after strcpy( ) =I love programming
```

it is also easy to copy one string to another via a for loop, as

```
int i,len;
len=strlen(str1);
for( i=0 ; i < len ; i++ ){
     str2[i]=str1[i];
}
//last index in the string was len-1
//now outside for loop value of i=len.
str2[i]='\0';    // don't ever forget to terminate the string with '\0'.
```

either way is good to copy but using strcpy is a good habit to have.

strcmp(str1, str2)

This function is used to compare two strings "str1" and "str2". this function returns zero("0") if the two compared strings are equal, else some none zero integer.
Declaration and usage of strcmp() function

```
strcmp( str1 , str2 );    //declaration
//using this function to check given two strings are equal or not.
if( strcmp( str1, str2 )==0){
        printf("string %s and string %s are equal\n",str1,str2);
}
else
```

```
      printf("string %s and string %s are not equal\n",str1,str2);
```

Return value

- if Return value if < 0 then it indicates str1 is less than str2
- if Return value if > 0 then it indicates str2 is less than str1
- if Return value if = 0 then it indicates str1 is equal to str2

Normal approach for comparing two strings is

```
void compare( char str1[], char str2[]){
int l1,l2,i;
l1=strlen(str1);
l2=strlen(str2);
if(l1!=l2){
    printf("string %s and string %s are not equal\n",str1,str);
}
else{
       for( i = 0; i < l1 ; i++){   // l1 or l2 anyone as both are equal.
              if(str1[i]!=str2[i]){
                  printf("string %s and string %s are not equal\n",str1,str);
                  break;
              }
      }
      if(i==l1){      //if the for loop has ran for the whole l1 lenth.
             printf("string %s and string %s are equal\n",str1,str);
      }
   }
}
```

you can yourself see that the first comparing approach is always better and time saving too.

strstr(str1, str2)

This library function finds the first occurrence of the substring str2 in the string str1. The terminating '\0' character is not compared.

Declaration

```
 strstr( str1, str2);
 - str1: the main string to be scanned.
 - str2: the small string to be searched in str1.

// let say str2="speed";
//function can also be written as
     strstr(str1, "speed" );
```

his function is very useful in checking whether str2 is a substring of str1 or not.

Return value

This function returns a pointer to the first occurrence in str1 of any of the entire sequence of characters specified in str2, or a NULL pointer if the sequence is not present in str1.

Example
Below code shows the usage of strstr() function.

```
#include<stdio.h>
#include<string.h>
int main ()
{
  char str1[55] ="This is a test string for testing";
  char str2[20]="test";

  char *p;
  p = strstr (str1, str2);
  if(p)
  {
    printf("string found\n" );  //i.e str2 is a substring of str1.
    printf ("First occurrence of string \"test\" in \"%s\" is"\
           " \"%s\"",str1, p);
  }
  else printf("string not found\n" );  // str2 is not a substring of str1.
   return 0;
```

```
}
```

Output

```
string found
```

```
First occurrence of string “test” in “This is a test string for testing” is “test string for testing”
```

Above explained two functions **strcpy()** and **strcmp()** can also be extended to n characters in the following way:

```
strncpy(dest, source,n)
```

Copies up to n characters from the string pointed to, by source to dest.

```
strncmp(str1, str2, n)
```

Compares at most the first n bytes of str1 and str2.

Array as Parameters

An **array** is a collection of similar type of data, data could be value or address. When we pass an array as a parameter then it splits into the **pointer** to its first element. We can say that if I shall pass the array of character as a parameter then it would be split into the pointer to the character. So, if a function parameter declared as T arr[] or T arr[n] is treated as T *arr..

In C language, it is easy to work on the 1D array as compared to a multidimensional array. In this article, I will explain a few ways to pass the array as parameters. Here I will also explain the few methods to passing 2d array to function.

Parameters as a pointer

We know that array split into the pointer of its first element, so here I am creating a function whose parameters are an integer pointer.

```
#include <stdio.h>

//Size of the created array

#define ARRAY_SIZE 5

//Function to read array element

voidReadArray(int *paData)

{

int index = 0;

for(index= 0; index < ARRAY_SIZE; ++index)

{

printf("%d\n",paData[index]);

}

}

int main(int argc, char *argv[])

{

//Create an array

int aiData[ARRAY_SIZE] = {1,2,3,4,5};

//Pass array as a parameter

ReadArray(aiData);

return0;
}
```

Parameters as a sized array

One of the simple ways to pass the array as a parameter declared the function with prototype same as the array that will pass to the function.

```
#include <stdio.h>

//Size of the created array

#define ARRAY_SIZE 8

voidReadArray(int acData[ARRAY_SIZE])

{

int index = 0;

for(index= 0; index < ARRAY_SIZE; ++index)

{

printf("%d\n",acData[index]);

}

}

int main(int argc, char *argv[])

{

//Create an array

int aiData[ARRAY_SIZE] = {1,2,3,4,5,6,7,8};

//Pass array as a parameter

ReadArray(aiData);

return0;

}
```

Parameters as an unsized array

When we pass the 1D array as a parameter then don’t need to specify the **size of the array.** It behaves like T *, where T is the type of the array.

```
#include <stdio.h>

//Size of the created array

#define ARRAY_SIZE 8

voidReadArray(int acData[])

{

int index = 0;

for(index= 0; index < ARRAY_SIZE; ++index)

{

printf("%d\n",acData[index]);

}

}

int main(int argc, char *argv[])

{

//Create an array

int aiData[ARRAY_SIZE] = {1,2,3,4,5,6,7,8};

//Pass array as a parameter

ReadArray(aiData);

return0;

}
```

}

Ways to passing a 2D array as a parameter to the function

Similar to the 1D array we can pass the 2D array as a parameter to the function. It is important to remember that when we pass a 2D array as a parameter decays into a pointer to an array, not a pointer to a pointer.

Passing 2d array to function in c using pointers

The first element of the multi-dimensional array is another array, so here, when we will pass a 2D array then it would be split into a pointer to the array.

For example,

If int aiData[3][3], is a 2D array of integers, it would be split into a pointer to the array of 3 integers (int (*)[3]).

```
#include <stdio.h>

//Size of the created array

#define ARRAY_ROW 3

#define ARRAY_COL 3

voidReadArray(int(*piData)[ARRAY_COL])

{

int iRow = 0;

int iCol = 0;

for(iRow = 0; iRow < ARRAY_ROW; ++iRow)

{

for(iCol = 0; iCol < ARRAY_COL; ++iCol)

{

printf("%d\n", piData[iRow][iCol]);

}

}
}

int main(int argc, char *argv[])

{

//Create an 2D array

int aiData[ARRAY_ROW][ARRAY_COL] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

//Pass array as a parameter

ReadArray(aiData);

return0;

}
```

Passing 2d array to function with row and column

Which prototype of the function should be the same as the passing array. In other word, we can say that if int aiData[3][3] is a 2D array, the function prototype should be similar to the 2D array.

```
#include <stdio.h>

//Size of the created array

#define ARRAY_ROW 3

#define ARRAY_COL 3

voidReadArray(int aiData[ARRAY_ROW][ARRAY_COL])

{

int iRow = 0;

int iCol = 0;

for(iRow = 0; iRow < ARRAY_ROW; ++iRow)

{

for(iCol = 0; iCol < ARRAY_COL; ++iCol)
{

printf("%d\n", aiData[iRow][iCol]);

}

}

}

int main(int argc, char *argv[])

{

//Create an 2D array

int aiData[ARRAY_ROW][ARRAY_COL] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

//Pass array as a parameter

ReadArray(aiData);

return0;
```

Passing 2d array to function omitting the row

In C language, the elements of the 2d array are stored row by row, there is no much more importance of the row when we are passing a 2d array to function. But it needs to remember that we have to specify the size of the column because it is used in jumping from row to row in memory.

```
#include <stdio.h>

//Size of the created array

#define ARRAY_ROW 3

#define ARRAY_COL 3

voidReadArray(int aiData[][ARRAY_COL])

{

int iRow = 0;

int iCol = 0;
for(iRow = 0; iRow < ARRAY_ROW; ++iRow)

{

for(iCol = 0; iCol < ARRAY_COL; ++iCol)

{

printf("%d\n", aiData[iRow][iCol]);

}

}

}

int main(int argc, char *argv[])

{

//Create an 2D array

int aiData[ARRAY_ROW][ARRAY_COL] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

//Pass array as a parameter

ReadArray(aiData);

return0;

}
```

Passing 2d array to a function, using the pointer to a 2D array

If int aiData[3][3] is a 2D array of integers, then &aiData would be pointer the 2d array that has 3 rows and 3 columns.

```
#include <stdio.h>

//Size of the created array

#define ARRAY_ROW 3

#define ARRAY_COL 3
```

```
voidReadArray(int(*piData)[ARRAY_ROW][ARRAY_COL])

{

int iRow = 0;

int iCol = 0;

for(iRow = 0; iRow < ARRAY_ROW; ++iRow)

{

for(iCol = 0; iCol < ARRAY_COL; ++iCol)

{

printf("%d\n", (*piData)[iRow][iCol]);

}

}

}

int main(int argc, char *argv[])

{

//Create an 2D array

int aiData[ARRAY_ROW][ARRAY_COL] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

//Pass array as a parameter

ReadArray(&aiData);

return0;
```

Ordered List

In order to implement the ordered list, we must remember that the relative positions of the items are based on some underlying characteristic. The ordered list of integers given above (17, 26, 31, 54, 77, and 93) can be represented by a linked structure as shown in Figure 15. Again, the node and link structure is ideal for representing the relative positioning of the items.

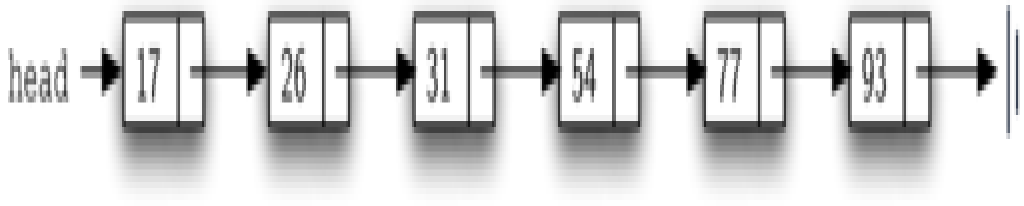


Figure 15: An Ordered Linked List

To implement the OrderedList class, we will use the same technique as seen previously with unordered lists. Once again, an empty list will be denoted by a head reference to None (see Listing 8).

Listing 8

```
classOrderedList:

def __init__(self):

    self.head = None
```

As we consider the operations for the ordered list, we should note that the isEmpty and size methods can be implemented the same as with unordered lists since they deal only with the number of nodes in the list without regard to the actual item values. Likewise, the remove method will work just fine since we still need to find the item and then link around the node to remove it. The two remaining methods, search and add, will require some modification.

The search of an unordered linked list required that we traverse the nodes one at a time until we either find the item we are looking for or run out of nodes (None). It turns out that the same approach would actually work with the ordered list and in fact in the case where we find the item it is exactly what we need. However, in the case where the item is not in the list, we can take advantage of the ordering to stop the search as soon as possible.

For example, Figure 16 shows the ordered linked list as a search is looking for the value 45. As we traverse, starting at the head of the list, we first compare against 17. Since 17 is not the item we are looking for, we move to the next node, in this case 26. Again, this is not what we want, so we move on to 31 and then on to 54. Now, at this point, something is different. Since 54 is not the item we are looking for, our former strategy would be to move forward. However, due to the fact that this is an ordered list, that will not be necessary. Once the value in the node becomes greater than the item we are searching for, the search can stop and return False. There is no way the item could exist further out in the linked list.

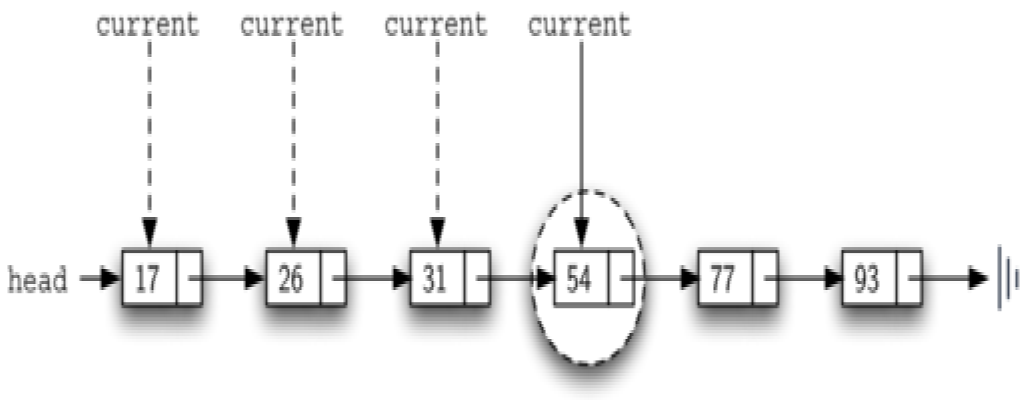


Figure 16: Searching an Ordered Linked List

Listing 9 shows the complete search method. It is easy to incorporate the new condition discussed above by adding another boolean variable, stop, and initializing it to False (line 4). While stop is False (not stop) we can continue to look forward in the list (line 5). If any node is ever discovered that contains data greater than the item we are looking for, we will set stop to True (lines 9–10). The remaining lines are identical to the unordered list search.

Listing 9

```
def search(self,item):

    current = self.head

    found = False

    stop = False

while current != Noneandnot found andnot stop:

if current.getData() == item:

            found = True

else:

if current.getData() > item:

                stop = True

else:

                current = current.getNext()
```

```
return found
```

The most significant method modification will take place in add. Recall that for unordered lists, the add method could simply place a new node at the head of the list. It was the easiest point of access. Unfortunately, this will no longer work with ordered lists. It is now necessary that we discover the specific place where a new item belongs in the existing ordered list.

Assume we have the ordered list consisting of 17, 26, 54, 77, and 93 and we want to add the value 31. The add method must decide that the new item belongs between 26 and 54. Figure 17 shows the setup that we need. As we explained earlier, we need to traverse the linked list looking for the place where the new node will be added. We know we have found that place when either we run out of nodes (current becomes None) or the value of the current node becomes greater than the item we wish to add. In our example, seeing the value 54 causes us to stop.

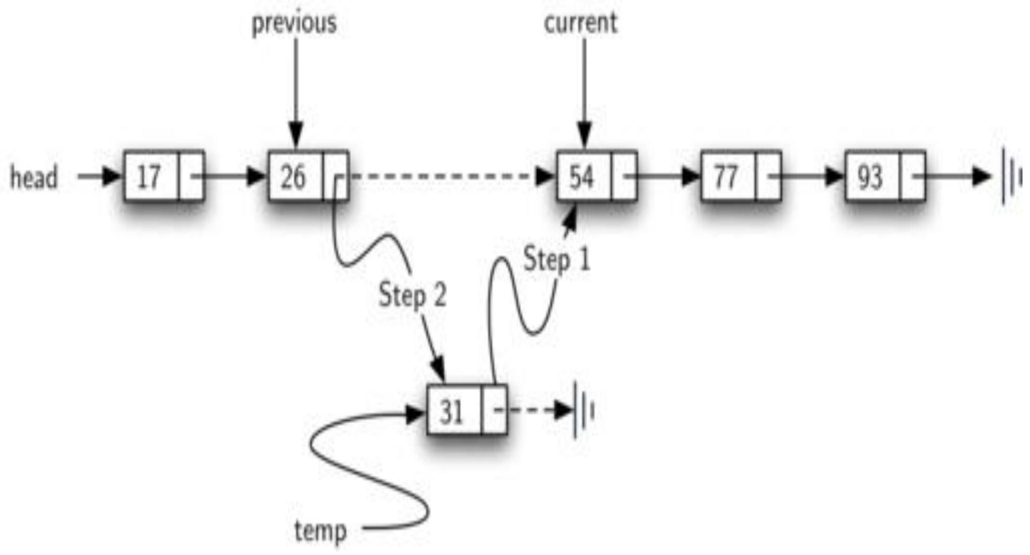


Figure 17: Adding an Item to an Ordered Linked List

As we saw with unordered lists, it is necessary to have an additional reference, again called previous, since current will not provide access to the node that must be modified. Listing 10 shows the complete add method. Lines 2–3 set up the two external references and lines 9–10 again allow previous to follow one node behind current every time through the iteration. The condition (line 5) allows the iteration to continue as long as there are more nodes and the value in the current node is not larger than the item. In either case, when the iteration fails, we have found the location for the new node.

The remainder of the method completes the two-step process shown in Figure 17. Once a new node has been created for the item, the only remaining question is whether the new node will be added at the beginning of the linked list or some place in the middle. Again, previous == None (line 13) can be used to provide the answer.

Listing 10

```
def add(self,item):
    current = self.head
    previous = None
    stop = False
while current != Noneandnot stop:
if current.getData() > item:
            stop = True
else:
            previous = current
            current = current.getNext()


    temp = Node(item)
if previous == None:
        temp.setNext(self.head)
        self.head = temp
else:
        temp.setNext(current)
        previous.setNext(temp)
```

The OrderedList class with methods discussed thus far can be found in ActiveCode 1. We leave the remaining methods as exercises. You should carefully consider whether the unordered implementations will work given that the list is now ordered.

Sparse Matrices

A sparse matrix is a matrix in which majority of the elements are 0. An example for this is given as follows.

The matrix given below contains 5 zeroes. Since the number of zeroes is more than half the elements of the matrix, it is a sparse matrix.

```
5 0 0
3 0 1
0 0 9
```

A program to implement a sparse matrix is as follows.

Example

```
#include<iostream>

usingnamespace std;

int main (){

  int a[10][10]={{0,0,9},{5,0,8},{7,0,0}};

  int i, j, count =0;

  int row =3, col =3;

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

    for(j =0; j < col;++j){

      if(a[i][j]==0)

      count++;

    }

  }

  cout<<"The matrix is:"<<endl;

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

    for(j =0; j < col;++j){

      cout<<a[i][j]<<" ";

    }

    cout<<endl;

  }

  cout<<"The number of zeros in the matrix are "<< count <<endl;
  if(count >((row * col)/2))

  cout<<"This is a sparse matrix"<<endl;

  else

  cout<<"This is not a sparse matrix"<<endl;

  return0;

}
```

Output

```
The matrix is:
0 0 9
5 0 8
7 0 0
The number of zeros in the matrix are 5
This is a sparse matrix
```

In the above program, a nested for loop is used to count the number of zeros in the matrix. This is demonstrated using the following code snippet.

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

  for(j =0; j < col;++j){

    if(a[i][j]==0)

    count++;

  }

}
```

After finding the number of zeros, the matrix is displayed using a nested for loop. This is shown below.

```
cout<<"The matrix is:"<<endl;

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

  for(j =0; j < col;++j){

    cout<<a[i][j]<<" ";

  }
  cout<<endl;

}
```

Finally, the number of zeroes are displayed. If the count of zeros is more than half the elements in the matrix, then it is displayed that the matrix is a sparse matrix otherwise the matrix is not a sparse matrix.

```
cout<<"The number of zeros in the matrix are "<< count <<endl;

if(count >((row * col)/2))

cout<<"This is a sparse matrix"<<endl;

else

cout<<"This is not a sparse matrix"<<endl;
```

Vectors

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array. Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

Certain functions associated with the vector are:

Iterators

1. begin() – Returns an iterator pointing to the first element in the vector
2. end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector
3. rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
4. rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

5. cbegin() – Returns a constant iterator pointing to the first element in the vector.

6. cend() – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.

7. crbegin() – Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element

8. crend() – Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

filter_none
edit
play_arrow
brightness_4

```
// C++ program to illustrate the
// iterators in vector
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Output of begin and end: ";
    for (auto i = g1.begin(); i != g1.end(); ++i)
        cout << *i << " ";

    cout << "\nOutput of cbegin and cend: ";
    for (auto i = g1.cbegin(); i != g1.cend(); ++i)
        cout << *i << " ";

    cout << "\nOutput of rbegin and rend: ";
    for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
        cout << *ir << " ";

    cout << "\nOutput of crbegin and crend : ";
    for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
        cout << *ir << " ";

    return 0;
}
```

Output:

```
Output of begin and end: 1 2 3 4 5
Output of cbegin and cend: 1 2 3 4 5
Output of rbegin and rend: 5 4 3 2 1
Output of crbegin and crend : 5 4 3 2 1
```

Capacity

1.
   size() – Returns the number of elements in the vector.
2. max_size() – Returns the maximum number of elements that the vector can hold.
3. capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
4. resize(n) – Resizes the container so that it contains ‘n’ elements.
5. empty() – Returns whether the container is empty.
6. shrink_to_fit() – Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
7. reserve() – Requests that the vector capacity be at least enough to contain n elements.

filter_none
edit
play_arrow
brightness_4

```
// C++ program to illustrate the
// capacity function in vector
#include <iostream>
#include <vector>

using namespace std;

int main()
{
   vector<int> g1;

   for (int i = 1; i <= 5; i++)
```

```
        g1.push_back(i);

    cout << "Size : " << g1.size();
    cout << "\nCapacity : " << g1.capacity();
    cout << "\nMax_Size : " << g1.max_size();

    // resizes the vector size to 4
    g1.resize(4);

    // prints the vector size after resize()
    cout << "\nSize : " << g1.size();

    // checks if the vector is empty or not
    if (g1.empty() == false)
        cout << "\nVector is not empty";
    else
        cout << "\nVector is empty";

    // Shrinks the vector
    g1.shrink_to_fit();
    cout << "\nVector elements are: ";
    for (auto it = g1.begin(); it != g1.end(); it++)
        cout << *it << " ";

    return 0;
}
```

Output:

```
Size : 5

Capacity : 8

Max_Size : 4611686018427387903

Size : 4

Vector is not empty

Vector elements are: 1 2 3 4
```

Element access:

1. reference operator [g] – Returns a reference to the element at position ‘g’ in the vector

2. at(g) – Returns a reference to the element at position ‘g’ in the vector

3. front() – Returns a reference to the first element in the vector
4. back() – Returns a reference to the last element in the vector
5. data() – Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

filter_none
edit
play_arrow
brightness_4

```
// C++ program to illustrate the
// element accesser in vector
#include <bits/stdc++.h>
using namespace std;

int main()
{
   vector<int> g1;

   for (int i = 1; i <= 10; i++)
      g1.push_back(i * 10);

   cout << "\nReference operator [g] : g1[2] = " << g1[2];

   cout << "\nat : g1.at(4) = " << g1.at(4);

   cout << "\nfront() : g1.front() = " << g1.front();

   cout << "\nback() : g1.back() = " << g1.back();

   // pointer to the first element
   int* pos = g1.data();

   cout << "\nThe first element is " << *pos;
   return 0;
}
```

Output:

```
Reference operator [g] : g1[2] = 30

at : g1.at(4) = 50

front() : g1.front() = 10

back() : g1.back() = 100
```

```
The first element is 10
```

Modifiers:

1. assign() – It assigns new value to the vector elements by replacing old ones
2. push_back() – It push the elements into a vector from the back
3. pop_back() – It is used to pop or remove elements from a vector from the back.
4. insert() – It inserts new elements before the element at the specified position
5. erase() – It is used to remove elements from a container from the specified position or range.
6. swap() – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.
7. clear() – It is used to remove all the elements of the vector container
8. emplace() – It extends the container by inserting new element at position
9. emplace_back() – It is used to insert a new element into the vector container, the new element is added to the end of the vector

.

filter_none
edit
play_arrow
brightness_4

```
// C++ program to illustrate the
// Modifiers in vector
#include <bits/stdc++.h>
#include <vector>
using namespace std;

int main()
{
    // Assign vector
    vector<int> v;

    // fill the array with 10 five times
    v.assign(5, 10);

    cout << "The vector elements are: ";
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";

    // inserts 15 to the last position
    v.push_back(15);
    int n = v.size();
    cout << "\nThe last element is: " << v[n - 1];

    // removes last element
    v.pop_back();

    // prints the vector
    cout << "\nThe vector elements are: ";
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";

    // inserts 5 at the beginning
    v.insert(v.begin(), 5);

    cout << "\nThe first element is: " << v[0];

    // removes the first element
    v.erase(v.begin());

    cout << "\nThe first element is: " << v[0];

    // inserts at the beginning
    v.emplace(v.begin(), 5);
    cout << "\nThe first element is: " << v[0];

    // Inserts 20 at the end
    v.emplace_back(20);
    n = v.size();
    cout << "\nThe last element is: " << v[n - 1];

    // erases the vector
    v.clear();
    cout << "\nVector size after erase(): " << v.size();

    // two vector to perform swap
    vector<int> v1, v2;
    v1.push_back(1);
    v1.push_back(2);
    v2.push_back(3);
    v2.push_back(4);

    cout << "\n\nVector 1: ";
```

```
    for (int i = 0; i < v1.size(); i++)
        cout << v1[i] << " ";

    cout << "\nVector 2: ";
    for (int i = 0; i < v2.size(); i++)
        cout << v2[i] << " ";

    // Swaps v1 and v2
    v1.swap(v2);

    cout << "\nAfter Swap \nVector 1: ";
    for (int i = 0; i < v1.size(); i++)
        cout << v1[i] << " ";

    cout << "\nVector 2: ";
    for (int i = 0; i < v2.size(); i++)
        cout << v2[i] << " ";
}
```

Output:

```
The vector elements are: 10 10 10 10 10

The last element is: 15

The vector elements are: 10 10 10 10 10

The first element is: 5

The first element is: 10

The first element is: 5

The last element is: 20

Vector size after erase(): 0


Vector 1: 1 2

Vector 2: 3 4

After Swap

Vector 1: 3 4

Vector 2: 1 2
```

Stacks:

Array Representation and Implementation of stack

We were discussed basic definitions of the Data structures and algorithms in the previous article. in this article, let’s dig deeper into the Data structure world, and especially, let’s get our hands dirty with a little bit for coding as well.

Objectives of this article:

1. Discuss Data types, built-in, and derived data types.
2. Introduce **Stack** Derived Data Structure
3. Implement and use Stack in Python {code} and NodeJs {code}
4. Introduce **Array** Derived Data Structure
5. Implement and use Stack in Python {code} and NodeJs {code}

Introduction to Data types

Data structures are made from one or more Data objects. data objects represent the data that we are going to store using carefully designed data structures. Data types are identified as primary ways of classifying several types of data in a data structure such as string, character, integer, etc. There are two major data types in the programming world, namely build-in data types and Derived data types.

Built-in Data Type

These are the basic data types that programming languages are supporting. Mainly known as the **primary data types** in a particular programming language.

Derived Data Type

These data types are implemented using one or more built-in (primary) data types. All data structures are developed based on such kind of derived data types. In this article, we will discuss Stack’s an Array's data structures with the implementation examples.

Stack Data Structure

The **stack** is an **Abstract data type** (ADT is a **type** for objects whose behavior is defined by a set of values and a set of operations) which is one of the main Data structures in programming languages, and for beginners, this is an easily understandable structure. **LIFO (L**ast **I**n **F**irst **O**ut) is the main specialty of a stack**.**

Like most of the data structures, the stack also represent real-world objects. For instance, a stack of coins, a stack of boxes, etc.

A stack can be implemented using an array, a list, a pointer, etc. When it comes to a stack, there is a set of functions defined to use the stack efficiently in the programming context.

stack operations (Image by author)

Python implementation

In python, we can use a list data type as a built-in data type to implement the stack data structure.

{code}Please find the attached codebase in this Github link.

NodeJs implementation

In NodeJs, we can implement a stack data structure using the Array data type.

{code} Please find the attached codebase in this Github link.

Push operation

Once you defined the stack class one of the main functionalities is push function. Here you will input an **item** to the top of the array.

Algorithm to implement
We can define an algorithm to implement the push operation in the stack class.

Step 1 − Checks if the stack is full(assume that the list is implemented based on a dynamic array) for the given size or not.Step 2 − If the stack is full, print an errorStep 3 − If the stack is not full for the given maximum size, increase the top by one and point the pointer to the next empty space.Step 4 − Add a new element to the new empty space which is in the top of the stackStep 5 − Return success.

Pop operation

The **pop** function will remove the topmost element from the stack, and the stack item count will be reduced by one. Even though it seems like the topmost element removed from the stack, still that **element will not be completely removed**, only the pointer will move to the below position.

Algorithm to implement
We can define an algorithm to implement the pop operation in the stack class.

Step 1 − Checks if the stack is empty by looking at the array lengthStep 2 − If the stack is empty, print an error, exitStep 3 − If the stack is not empty, get the element which is pointing at the top of the stack.Step 4 − Decreases the size of the stack by 1, automatically the pointer of the top most item will changed to the below item.Step 5 − Returns success.

Peek operation

The **peek** function will display the topmost element from the stack, and this operation will not remove the item from the stack like in the **pop** operation.

Algorithm to implement
We can define an algorithm to implement the peek operation in the stack class. the peek operation just returns the value at the top of the stack.

Step 1 − Checks if the stack is empty by looking at the array lengthStep 2 − If the stack is empty, print an error, exitStep 3 − If the stack is not empty, get the element which is pointing at the top of the stack.Step 4 − Returns success.

There are other functions like **isEmpty()**, **isFull(),** and **printStackItems()** which can be used as supportive functions that will help you to use the stack efficiently.

Stack implementation and all supportive functional implementations will be found in this codebase. stack data structure

Get your hands dirty with a stack data structure because we will use these data structures in the future when we are solving real-life problems in the algorithms section.

Array Data Structure

The array is one of the most used data structures when programmers implementing their algorithms. One specialty of an array is that the array container should be in **fixed size** and all the elements should be in the **same type**.

Array representation (Image By author)

Apart from all the elements in the array should be in the same data type, array always starts from the 0th element (**zero-indexed**), and the size of the array means, how many elements can be stored in itself.

In many data types, there are major operations(functionalities) that exist for the effective and efficient usage of that data type. In the array data type, there are five major operations exist.

Array Insertion

Add a new element to the given index is called the array insertion. We were able to implement inserting an element to the given index via python and nodejs programming language.

Implementation of the array insert operation in **nodejs**

Implementation of the array insert operation in **python**

Array Search

We can perform search operations on an array element based on either the **value** or an **index**.

Search by index means, return the corresponding array element to the given index, and the search by value means to return the corresponding value the given index in the array.

Implementation of the array insert operation in **nodejs**

Implementation of the array insert operation in **python**

Array Deletion

Since the array is a fixed size data structure, delete an element from the given position in an array is a little bit tricky. You should adjust the new array by reducing the size of the array while you are deleting the given element. Please refer following code samples which explain how to delete a given element from an array with basic programming techniques.

Implementation of the array insert operation in **python**

Implementation of the array insert operation in **Nodejs**

Array Update

Array update is quite an easy operation that you just need to traverse through the given array until you find the required element to update. Please follow the following code samples to get more familiar with array update functionality.

Implementation of the array update operation in **nodejs — code**

Implementation of the array update operation in **python — code**

Array Traversal

Array traversal is nothing but print all the array elements in a sequence. Since the array is a zero-indexed data type, we can start to print array elements by traversal through the array structure from the zero position.

Implementation of the array traversal operation in **nodejs — code**

Implementation of the array traversal operation in **python — code**

In conclusion, we have learned how to use the **Stack** and **Array** efficiently. Please find the following code segments to see how we can implement a stack and an array in python and nodejs.

You will only need node installed in your machine and you are good to go. (node installation link)

Start coding to get your hands dirty for the following lessons.

FYI: when you trying to implement data structure operations, always remember on "Algorithm to implement" and think in that direction. This approach will help you to get a deep understanding of the entire workflow of data structure usage.

Operations on Stacks:

Push & Pop

Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end and (top) an element is removed from that end only.

stack::push()

push() function is used to insert an element at the top of the stack. The element is added to the stack container and the size of the stack is increased by 1.

Syntax :

stackname.push(value)
Parameters :

The value of the element to be inserted is passed as the parameter.
Result :
Adds an element of value same as that of
the parameter passed at the top of the stack.

Examples:

```
Input :   mystack
        mystack.push(6);
Output :  6

Input :   mystack
        mystack.push(0);
        mystack.push(1);
Output :  0, 1
```

Errors and Exceptions

1. Shows error if the value passed doesn’t match the stack type.

2. Shows no exception throw guarantee if the parameter doesn’t throw any exception.

filter_none
edit
play_arrow
brightness_4

```
// CPP program to illustrate
// Implementation of push() function
#include <iostream>
#include <stack>
using namespace std;

int main()
{
   // Empty stack
   stack<int> mystack;
   mystack.push(0);
   mystack.push(1);
   mystack.push(2);
```

```
    // Printing content of stack
    while (!mystack.empty()) {
        cout << ' ' << mystack.top();
        mystack.pop();
    }
}
```
Output:

```
2 1 0
```

Note that output is printed on the basis of LIFO property

stack::pop()

pop() function is used to remove an element from the top of the stack(newest element in the stack). The element is removed to the stack container and the size of the stack is decreased by 1.

Syntax :

stackname.pop()
Parameters :
No parameters are passed.
Result :
Removes the newest element in the stack
or basically the top element.

Examples:

```
Input :   mystack = 0, 1, 2

          mystack.pop();

Output :  0, 1


Input :   mystack = 0, 1, 2, 3, 4, 5

          mystack.pop();

Output :  0, 1, 2, 3, 4
```

Errors and Exceptions

1. Shows error if a parameter is passed.

2. Shows no exception throw guarantee.

filter_none
edit

play_arrow
brightness_4

```
// CPP program to illustrate
// Implementation of pop() function
#include <iostream>
#include <stack>
using namespace std;

int main()
{
    stack<int> mystack;
    mystack.push(1);
    mystack.push(2);
    mystack.push(3);
    mystack.push(4);
    // Stack becomes 1, 2, 3, 4

    mystack.pop();
    mystack.pop();
    // Stack becomes 1, 2

    while (!mystack.empty()) {
        cout << ' ' << mystack.top();
        mystack.pop();
    }
}
```

Output:

```
2 1
```

Note that output is printed on the basis of LIFO property

Application :

Given a number of integers, add them to the stack and find the size of the stack without using size function.

Input : 5, 13, 0, 9, 4

Output: 5

Algorithm

1. Push the given elements to the stack container one by one.
2. Keep popping the elements of stack until it becomes empty, and increment the counter variable.
3. Print the counter variable.

filter_none
edit
play_arrow
brightness_4

```
// CPP program to illustrate
// Application of push() and pop() function
#include <iostream>
#include <stack>
using namespace std;

int main()
{
   int c = 0;
   // Empty stack
   stack<int> mystack;
   mystack.push(5);
   mystack.push(13);
   mystack.push(0);
   mystack.push(9);
   mystack.push(4);
   // stack becomes 5, 13, 0, 9, 4

   // Counting number of elements in queue
   while (!mystack.empty()) {
      mystack.pop();
      c++;
   }
   cout << c;
}
```

Output:

```
5
```

Array Representation of Stack

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false.

How to understand a stack practically?

There are many real-life examples of a stack. Consider the simple example of plates stacked over one another in a canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

Time Complexities of operations on stack:

push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

Applications of stack:

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- Backtracking is one of the algorithm designing technique .Some example of back tracking are Knight-Tour problem,N-Queen problem,find your way through maze and game like chess or checkers in all this problems we dive into someway if that way is not efficient we come back to the previous state and go into some another path. To get back from current state we need to store the previous state for that purpose we need stack.
- In Graph Algorithms like Topological Sorting and Strongly Connected Components
- In Memory management any modern computer uses stack as the primary-management for a running purpose.Each program that is running in a computer system has its own memory allocations

- String reversal is also a another application of stack.Here one by one each character get inserted into the stack.So the first character of string is on the bottom of the stack and the last element of string is on the top of stack. After Performing the pop operations on stack we get string in reverse order .

Implementation:

There are two ways to implement a stack:

- Using array
- Using linked list

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Implementing Stack using Arrays

- C++
- C
- Java
- Python
- C#

filter_none
edit
play_arrow
brightness_4

```
/* C++ program to implement basic stack
   operations */
#include <bits/stdc++.h>

using namespace std;

#define MAX 1000

class Stack {
    int top;

public:
    int a[MAX]; // Maximum size of Stack
    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};

bool Stack::push(int x)
{
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
        return false;
    }
    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}

int Stack::pop()
{
    if (top < 0) {
        cout << "Stack Underflow";
        return 0;
    }
    else {
        int x = a[top--];
        return x;
    }
}
int Stack::peek()
{
    if (top < 0) {
        cout << "Stack is Empty";
        return 0;
    }
    else {
        int x = a[top];
        return x;
    }
}

bool Stack::isEmpty()
{
    return (top < 0);
```

```
}

// Driver program to test above functions
int main()
{
    class Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << s.pop() << " Popped from stack\n";

    return 0;
}
```

Output :

```
10 pushed into stack

20 pushed into stack

30 pushed into stack

30 popped from stack
```

Pros: Easy to implement. Memory is saved as pointers are not involved.
Cons: It is not dynamic. It doesn’t grow and shrink depending on needs at runtime.

Implementing Stack using Linked List

- C++
- C
- Java
- Python
- C#

filter_none
edit
play_arrow
brightness_4

```
// C++ program for linked list implementation of stack
#include <bits/stdc++.h>
using namespace std;
```

```
// A structure to represent a stack
class StackNode {
public:
    int data;
    StackNode* next;
};

StackNode* newNode(int data)
{
    StackNode* stackNode = new StackNode();
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}

int isEmpty(StackNode* root)
{
    return !root;
}

void push(StackNode** root, int data)
{
    StackNode* stackNode = newNode(data);
    stackNode->next = *root;
    *root = stackNode;
    cout << data << " pushed to stack\n";
}

int pop(StackNode** root)
{
    if (isEmpty(*root))
        return INT_MIN;
    StackNode* temp = *root;
    *root = (*root)->next;
    int popped = temp->data;
    free(temp);

    return popped;
}

int peek(StackNode* root)
{
    if (isEmpty(root))
        return INT_MIN;
    return root->data;
}

// Driver code
int main()
{
    StackNode* root = NULL;

    push(&root, 10);
    push(&root, 20);
    push(&root, 30);

    cout << pop(&root) << " popped from stack\n";

    cout << "Top element is " << peek(root) << endl;

    return 0;
}

// This is code is contributed by rathbhupendra
```

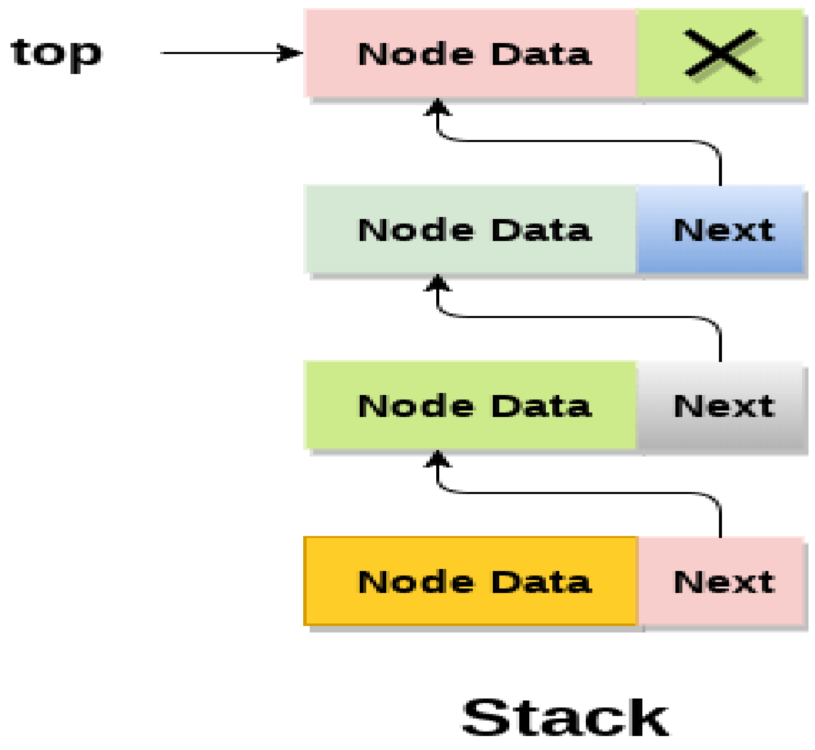
Output:

```
10 pushed to stack

20 pushed to stack

30 pushed to stack

30 popped from stack

Top element is 20
```

Linked Representation of Stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Adding a node to the stack (Push operation)

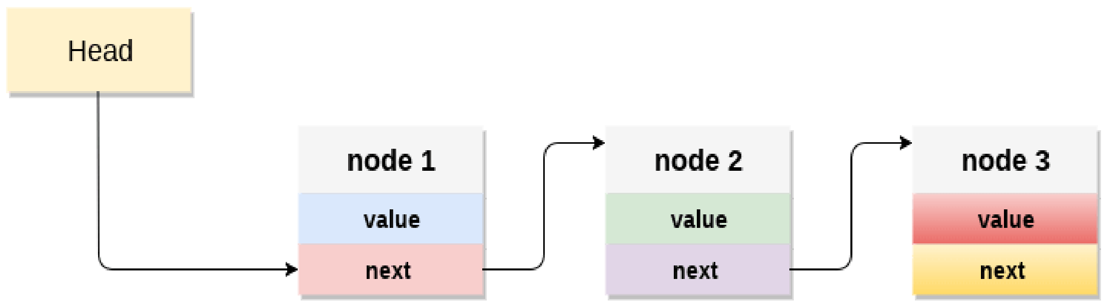
Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

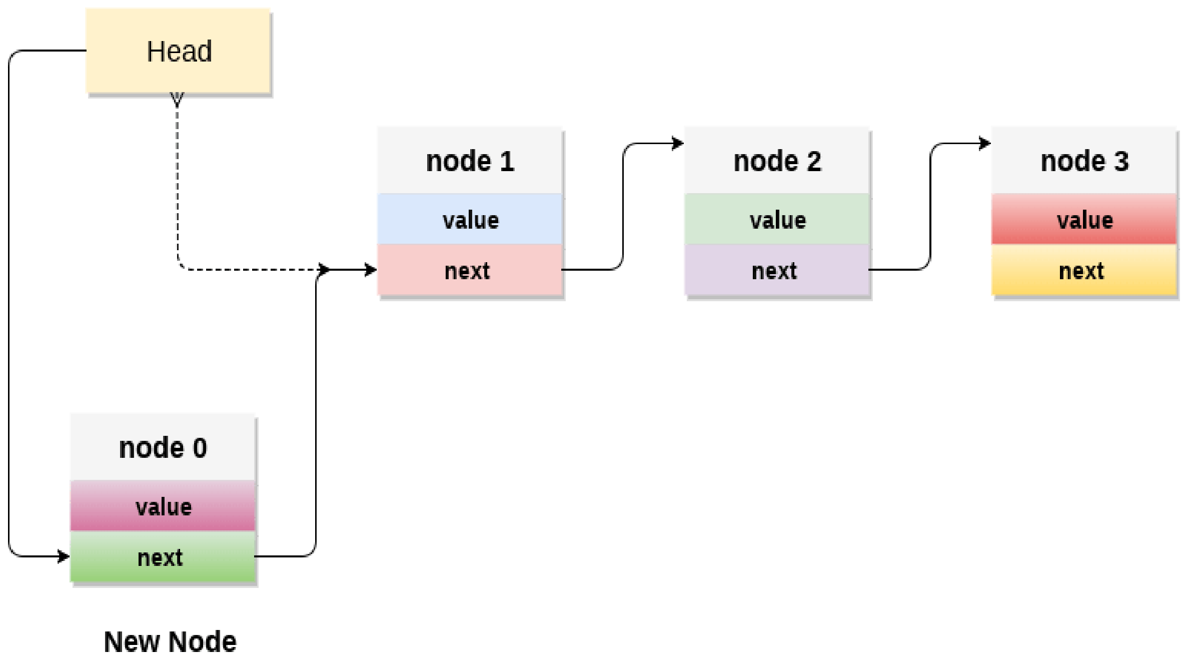
1. Create a node first and allocate memory to it.

2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.

3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : o(1)





C implementation :

```
void push ()
{
int val;
struct node *ptr =(struct node*)malloc(sizeof(struct node));
if(ptr == NULL)
{
printf("not able to push the element");
}
else
{
printf("Enter the value");
scanf("%d",&val);
if(head==NULL)
{
ptr->val = val;
ptr -> next = NULL;
head=ptr;
}
else
{
ptr->val = val;
ptr->next = head;
head=ptr;

}
printf("Item pushed");

}
}
```

Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

- **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
- **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

C implementation

```
void pop()
```

```
{
int item;
struct node *ptr;
if (head == NULL)
{
printf("Underflow");
}
else
{
item = head->val;
ptr = head;
head = head->next;
free(ptr);
printf("Item popped");

}
}
```

Display the nodes (Traversing)

Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

- Copy the head pointer into a temporary pointer.
- Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity : o(n)

C Implementation

```
void display()
{
int i;
struct node *ptr;
ptr=head;
if(ptr == NULL)
{
printf("Stack is empty\n");
}
else
{
printf("Printing Stack elements \n");
while(ptr!=NULL)
{
printf("%d\n",ptr->val);
ptr = ptr->next;
}
}
}
```

Menu Driven program in C implementing all the stack operations using linked list :

```
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{
int val;
struct node *next;
};
struct node *head;

void main ()
{
int choice=0;
printf("\n*********Stack operations using linked list*********\n");
printf("\n----------------------------------------------\n");
while(choice != 4)
{
printf("\n\nChose one from the below options...\n");
printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
printf("\n Enter your choice \n");
scanf("%d",&choice);
switch(choice)
{
case 1:
{
push();
break;
}
case 2:
{
pop();
break;
}
case 3:
{
display();
break;
}
case 4:
{
printf("Exiting....");
break;
}
default:
{
printf("Please Enter valid choice ");
}
};
}
}
void push ()
{
int val;
struct node *ptr = (struct node*)malloc(sizeof(struct node));
if(ptr == NULL)
{
printf("not able to push the element");
}
else
{
printf("Enter the value");
scanf("%d",&val);
if(head==NULL)
{
ptr->val = val;
ptr -> next = NULL;
head=ptr;
}
```

```
else
{
ptr->val = val;
ptr->next = head;
head=ptr;

}
printf("Item pushed");

}
}

void pop()
{
int item;
struct node *ptr;
if (head == NULL)
{
printf("Underflow");
}
else
{
item = head->val;
ptr = head;
head = head->next;
free(ptr);
printf("Item popped");

}
}
void display()
{
int i;
struct node *ptr;
ptr=head;
if(ptr == NULL)
{
printf("Stack is empty\n");
}
else
{
```

```
printf("Printing Stack elements \n");
while(ptr!=NULL)
{
printf("%d\n",ptr->val);
ptr = ptr->next;
}
}
}
```

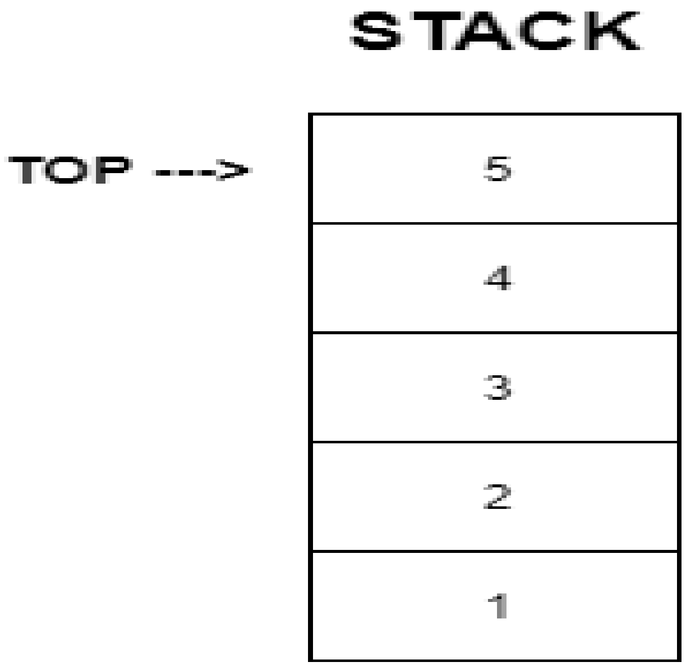
Operations Associated with Stacks

First, let us see the properties of data structures that we already do know and build-up our concepts towards the stack.

- **Array:** Its a random-access container, meaning any element of this container can be accessed instantly
- **Linked List:** It's a sequential-access container, meaning that elements of this data structure can only be accessed sequentially

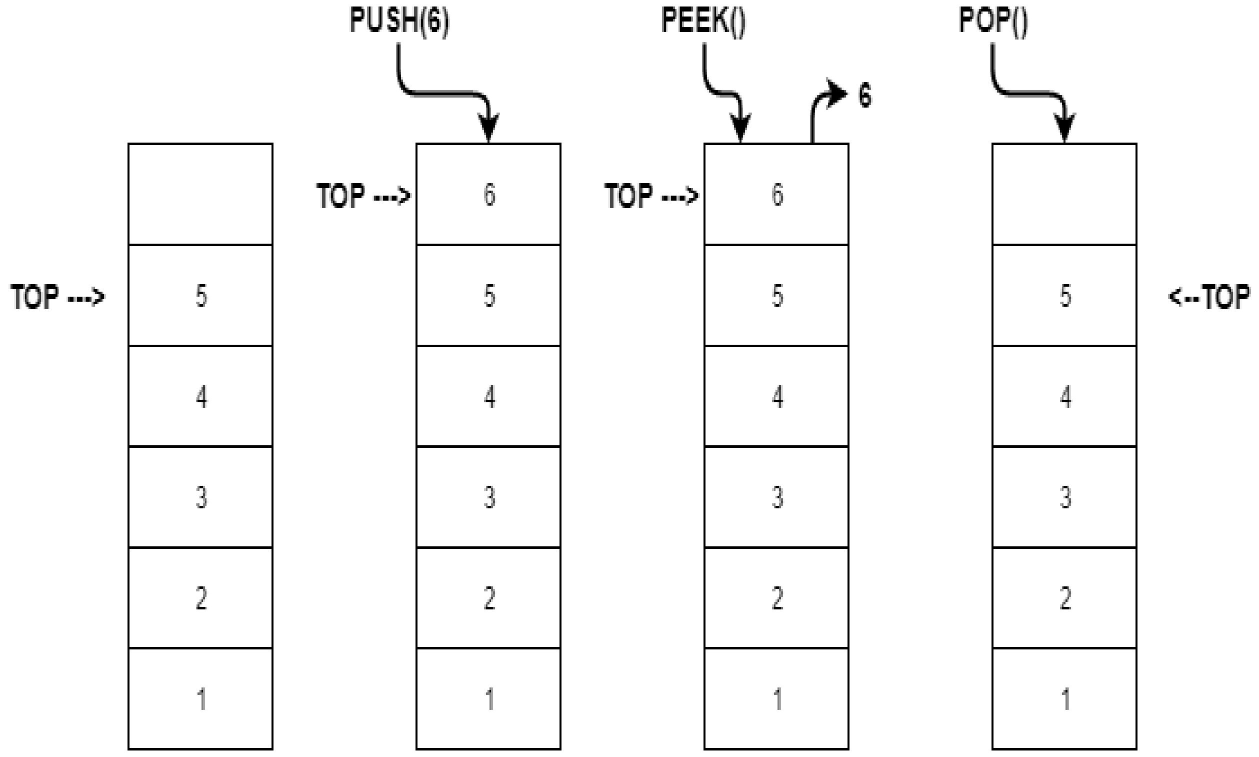
→ Following a similar definition, a **stack** is a container where only the top element can be accessed or operated upon.

A **Stack** is a data structure following the LIFO(Last In, First Out) principle.



If you have trouble visualizing stacks, just assume a stack of books.

- In a stack of books, you can only see the top book
- If you want to access any other book, you would first need to remove the books on top of it
- The bottom-most book in the stack was put first and can only be removed at the last after all books on top of it have been removed.



PUSH Operation

Push operation refers to inserting an element in the stack. Since there's only one position at which the new element can be inserted — Top of the stack, the new element is inserted at the top of the stack.

POP Operation

Pop operation refers to the removal of an element. Again, since we only have access to the element at the top of the stack, there's only one element that we can remove. We just remove the top of the stack. **Note:** We can also choose to return the value of the popped element back, its completely at the choice of the programmer to implement this.

PEEK Operation

Peek operation allows the user to see the element on the top of the stack. The stack is not modified in any manner in this operation.

isEmpty: Check if stack is empty or not

To prevent performing operations on an empty stack, the programmer is required to internally maintain the size of the stack which will be updated during push and pop operations accordingly. isEmpty() conventionally returns a boolean value: True if size is 0, else False.

Stack Implementation

As we've learned before, Stack is a very useful concept that a good programmer could use to his/her benefit. But can you implement it in your code yet? It's not that difficult once you think about it, let's walk through its properties and implement them in code.

You should remember one very important thing though →

All operations in the stack must be of **O(1)** time complexity

We shall be implementing stack in two different ways by changing the underlying container: **Array** and **Linked List.**

1. Array Implementation

An array is one of the simplest containers offering random access to users based on indexes. But can we access any element of the stack at any given time? **No**. That's why we need to set an index as **top** and then access only the element at index **top.**

```
int stack[10]
int top = -1
```

Here, 10 is a pre-defined capacity of the stack. We can throw a stack overflow error if a user tries to exceed this capacity.

★ The default value for the **top** is -1, denoting that the stack is empty.

Do we need to store any other parameter for the stack? current size, perhaps? **No.** We may need to store the **capacity** of the stack but we don't need to store the current size. We can know the current size of stack by looking at the value of the **top. (How?)**

Let us wrap this group of data members in a **class**

```
classStack{
int arr[]
int capacity
int top
}
```

Let us also create a constructor which initializes **capacity** and **top**

```
Stack(int cap)
{
   capacity = cap
   top = -1
}
```

★ You are also required to allocate memory to **arr** according to the language you use to implement it.

→Now, we need to implement the operations that we generally perform on stacks.

PUSH Operation

What changes are made to the stack when a new element is pushed?

- A new element is inserted on top
- The value of top increases by 1

▹ What if the stack is filled to its capacity?

We shall check if the stack if full before inserting a new element and throw an error if it is.

Now, let us implement this simply

```
voidpush(int item)
{
if ( top == capacity - 1 )
        print( "Stack overflow!" )
else
    {
        arr[top+1] = item
        top = top + 1
    }
}
```

POP Operation

Let's try one more: Pop().

What changes are made to the stack internally for a pop operation?

- The top element is removed
- The value of **top** is decreased by 1

▹ Can you think of an exception in this case like the one above of stack being full?
Ans: The stack can be empty when the pop operation is called

Let’s try implementing it now

```
voidpop()
{
if ( isEmpty() == True )
      print( "Stack is empty!" )
else
      top = top - 1
}
```

★ Is decreasing the value of **top** same as deleting the top element? (**Think!**)

Peek and isEmpty Operation

Peek and isEmpty are quite simple to implement. We need to steer clear of exceptions though.

```
intpeek()
{
if ( isEmpty() == True )
   {
      print( "Stack is empty!" )
return -1
   }
else
return arr[top]
}
bool isEmpty()
{
if ( top == -1 )
return True
else
return False
}
```

2. Linked List Implementation

Before implementing stack with a linked list, let us first try to visualize a stack as a linked list. There are two ends of a linked list: **head** and **tail.**

Which end do you think should represent the top of the stack? (**Think!**)

The top of the stack should be represented by **head** because otherwise, we would not be able to implement the operations of the stack in O(1) time complexity.

Let us assume that we have used class for linked list

```
classListNode{
int val
   ListNode next
}
```

What should an empty stack look like if implemented with a linked list? **Ans:** Its head will point to NULL

```
ListNode head = NULL
```

Is there any benefit to implementing a stack as linked list compared to arrays? **Ans:** We do not need to mention the size of the stack beforehand.

→ Although, if you want to implement a limit to prevent excess use, you may need to encapsulate the **class ListNode** inside some other class along with a data member **capacity**.

```
classStack{
int capacity
classListNode{
int val
      ListNode next
   }
}
```

We shall be using just using **class ListNode** below for simplicity.

→Let us move towards stack operations

PUSH Operation

The same properties hold as above with an added benefit that we need not worry about the stack being full

```
voidpush(int item)
{
  ListNode temp = ListNode(item)
  temp.next = head
  head = temp
}
```

POP Operation

Properties of linked list relaxed us from checking if the stack is full in push operation. Do they provide any relaxation for exceptions in pop operation? **No.** We still need to check if the stack is empty.

Since the top is represented by the **head** in this implementation. How do we delete the first element in a linked list?

Simple, we make the second element as the head.

```
voidpop()
{
if ( head == NULL )
     print ( "Stack is empty!" )
else
     head = head.next
}
```

★ You may be required to deallocate the popped node to avoid memory leak according to your programming language's conventions.

Peek and isEmpty Operation

The implementation of these two operations is pretty simple and straight-forward in the linked list too.

```
intpeek()
{
if ( head == NULL )
```

```
   {
       print ( "Stack is empty!" )
return -1
   }
else
return head.val
}
bool isEmpty()
{
if ( head == NULL )
return True
else
return False
}
```

Augmentations in Stack

You can augment the stack data structure according to your needs. You can implement some extra operations like:-

- isFull(): tells you if the stack if filled to its capacity
- The pop operation could return the element being deleted

★ A quite interesting augmentation that has been asked in many interviews asks you to implement a **MinStack** which holds all properties of a regular stack but also returns the minimum value in the stack. Since all stack operations are expected to be executed in constant time, you need to return the minimum element in the stack in O(1) time complexity.

Applications of Stack Data Structure

- An "undo" mechanism in text editors
- Forward and backward feature in web browsers
- Check for balanced parentheses in an expression
- Expression evaluation and syntax parsing
- **Backtracking**. This is a process when you need to access the most recent data element in a series of elements. Think of a 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.

- We use a stack for the Iterative implementation of several recursive programs like tree traversals, DFS traversal in a graph, etc.
- For solving several problems in algorithms, we use a stack as the principle data structure with which they organize their information.
- Memory management: Any modern computer environment uses a stack as the primary memory management model for a running program.

Application of stack:

Conversion of Infix to Prefix and Postfix Expressions

After a long time, I'm going to work on Expression Evaluation, so I'm writing this blog post to revisit and explain it to myself.

The expressions we (human beings) write are called infix expressions as the operators come in between the operands to denote the expression's execution flow.

Let's consider the following expression.

A + B, this is an infix expression because the operator "+" comes between operands "A" and "B".

To evaluate expressions manually infix notation is helpful as it is easily understandable by the human brain.

But infix expressions are hard to parse in a computer program hence it will be difficult to evaluate expressions using infix notation. To reduce the complexity of expression evaluation Prefix or Postfix expressions are used in the computer programs.

Let's see what is Postfix expressions:

In Postfix expressions, operators come after the operands. Below are an infix and respective Postfix expressions.

A + B → A B +

As mentioned in the above example, the Postfix expression has the operator after the operands.

To begin conversion of Infix to Postfix expression, first, we should know about operator precedence.

Operator Precedence:

Precedence of the operators takes a crucial place while evaluating expressions.

The top operator in the table has the highest precedence. As per the precedence, the operators will be pushed to the stack.

Let’s see an example of the infix to Postfix conversion, we will start with a simple one,

Infix expression: **A + B**

If we encounter an operand we will write in the expression string, if we encounter an operator we will push it to an operator stack.

So we have two elements,

1. An empty expression string
2. An empty operator stack

In the expression first, we are encountering “A”, as it is an operand we will add it to the expression string. So now the two elements look like below,

1. Expression string: A

2. Operator Stack:

The second token we are encountering is an operator “+”, so we will push it to the operator stack.

1. Expression string: A
2. Operator Stack: **+**

The third token is an operand “B”, so we will add it to the expression string.

1. Expression string: A B
2. Operator Stack: +

Thus we processed all the tokens in the given expression, now we need to pop out the remaining tokens from the stack and have to add it to the expression string.

Pop the operator “+” from the stack and add it to the expression string which already has “**A B**” in it.

So now the output becomes “**A B +**” which is the Postfix notation for the given infix expression “**A + B**”.

Second Example:

Let’s convert a little complex expression with parentheses. Below is the given infix expression,

((A + B) — C * (D / E)) + F

The given expression has parentheses to denote the precedence. So let’s start with the conversion with two empty elements respectively,

1. An empty expression string

2. An empty operator stack

The first token to encounter is an open parenthesis, add it to the operator stack.

1. Expression string:
2. Operator Stack: **(**
3. Remaining expression: **(A + B) - C * (D / E)) + F**

The second token to encounter is again an open parenthesis, add it to the stack.

1. Expression string:
2. Operator Stack: **((**
3. Remaining expression: **A + B) - C * (D / E)) + F**

Next token un the expression is an operand “A”, so add it to the expression string.

1. Expression string: **A**
2. Operator Stack: **((**
3. Remaining expression: **+ B) - C * (D / E)) + F**

Afterward, we have an operator “+”, so add it to the stack.

1. Expression string: **A**
2. Operator Stack: **((+**
3. Remaining expression: **B) - C * (D / E)) + F**

Then we have an operand, so add it to the expression string.

1. Expression string: **A B**

2. Operator Stack: **((+**
3. Remaining expression: **) - C * (D / E)) + F**

Next token in the given infix expression is a close parenthesis, as we encountered a close parenthesis we should pop the expressions from the stack and add it to the expression string until an open parenthesis popped from the stack.

1. Expression string: **A B +**
2. Operator Stack: **(**
3. Remaining expression: **- C * (D / E)) + F**

Notice here we didn't push the close parenthesis to the stack, instead, we pooped out the operator "+" and added it to the expression string and pooped out one open parenthesis from the stack as well.

Next, we are encountering with an operator "-", so push it to the stack.

1. Expression string: **A B +**
2. Operator Stack: **(-**
3. Remaining expression: **C * (D / E)) + F**

Next is an operand "C", so add it to the expression string,

1. Expression string: **A B + C**
2. Operator Stack: **(-**
3. Remaining expression: *** (D / E)) + F**

Next is an operator "*", so push it to the stack.

1. Expression string: **A B + C**

2. Operator Stack: **(- ***
3. Remaining expression: **(D / E)) + F**

Next is an open parenthesis, so add it to the stack.

1. Expression string: **A B + C**
2. Operator Stack: **(- * (**
3. Remaining expression: **D / E)) + F**

Next is an operand “D”, so add it to the expression string.

1. Expression string: **A B + C D**
2. Operator Stack: **(- * (**
3. Remaining expression: **/ E)) + F**

Next we encounter an operator “/”, so push it to the stack.

1. Expression string: **A B + C D**
2. Operator Stack: **(- * (/**
3. Remaining expression: **E)) + F**

Then an oprand “E”, add it to the expression string.

1. Expression string: **A B + C D E**
2. Operator Stack: **(- * (/**
3. Remaining expression: **)) + F**

Then a close parenthesis, as we saw earlier, we should not push it to the stack instead we should pop all the operators from the stack and add it to the expression string until we encounter an open parenthesis. Then pop the open parenthesis from the stack but don't add it to the expression string.

1. Expression string: **A B + C D E /**
2. Operator Stack: **(- ***
3. Remaining expression: **) + F**

Next token is again a close paranthesis, so we will pop all the operators and add them to the expression string until we reach the open parenthesis and we will pop the open parenthesis as well from the operator stack.

1. Expression string: **A B + C D E / * -**
2. Operator Stack:
3. Remaining expression: **+ F**

Next token is an operator "+", so push it to the stack.

1. Expression string: **A B + C D E / * -**
2. Operator Stack: **+**
3. Remaining expression: **F**

Next token is an operand, "F". Add it to the expression string.

1. Expression string: **A B + C D E / * - F**
2. Operator Stack: **+**
3. Remaining expression:

As we processed the whole infix expression, now the operator stack has to be cleared by popping out each remaining operator and adding them to the expression string.

Here we have the operator "+" on the stack, so we will pop out the operator "+" from the stack and will add it to the expression string. So the resultant Postfix expression would look like below,

Final Postfix expression: **A B + C D E / * - F +**

Evaluation of postfix expression using stack

As discussed in Infix To Postfix Conversion Using Stack, the compiler finds it convenient to evaluate an expression in its postfix form. The virtues of postfix form include elimination of parentheses which signify priority of evaluation and the elimination of the need to observe rules of hierarchy, precedence and associativity during evaluation of the expression.

As **Postfix expression** is without parenthesis and can be evaluated as two operands and an operator at a time, this becomes easier for the compiler and the computer to handle.

Evaluation rule of a Postfix Expression states:

1. While reading the expression from left to right, push the element in the stack if it is an operand.

2. Pop the two operands from the stack, if the element is an operator and then evaluate it.

3. Push back the result of the evaluation. Repeat it till the end of the expression.

Algorithm

1) Add) to postfix expression.

2) Read postfix expression Left to Right until) encountered

3) If operand is encountered, push it onto Stack
   [End If]

4) If operator is encountered, Pop two elements

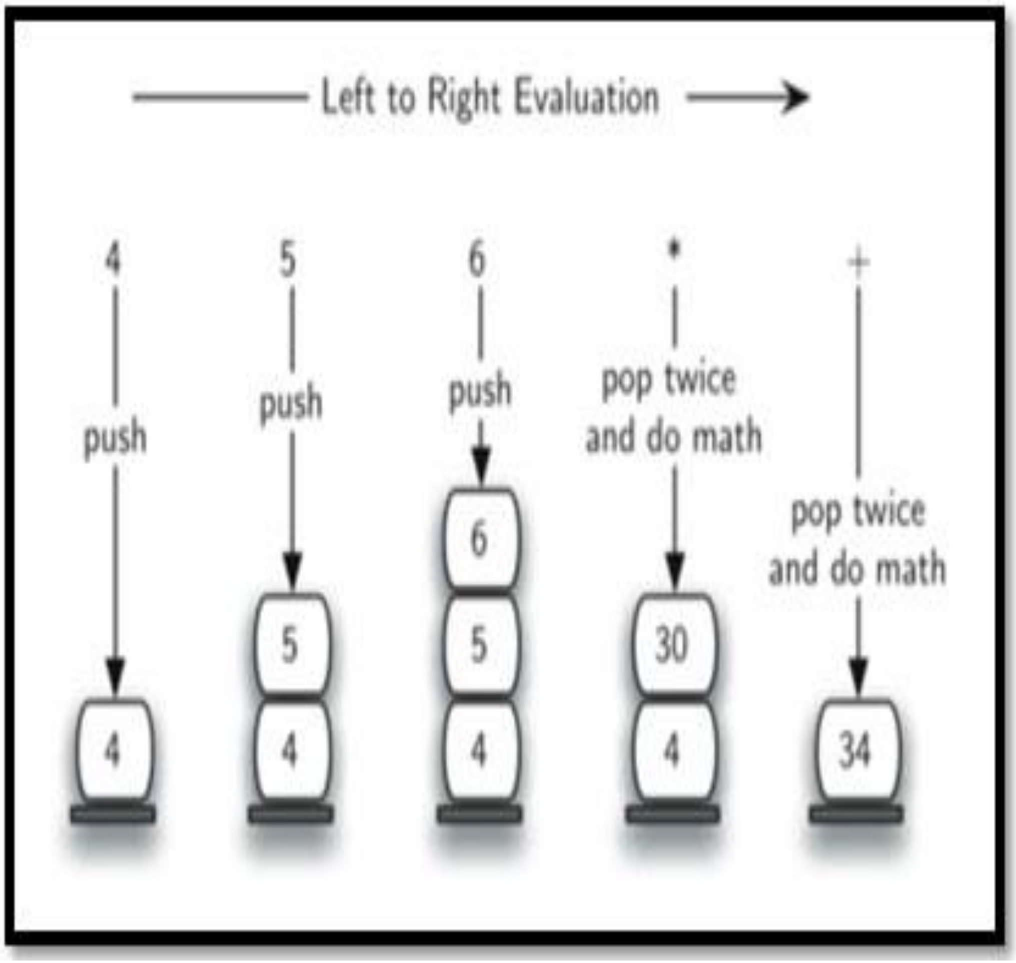
i) A -> Top element
ii) B-> Next to Top element
iii) Evaluate B operator A
push B operator A onto Stack

5) Set result = pop

6) END

Let's see an example to better understand the algorithm:

Expression: 456*+



Result: 34

Evaluation of Postfix Expressions Using Stack

```
/* This program is for evaluation of postfix expression
* This program assume that there are only four operators
* (*, /, +, -) in an expression and operand is single digit only
* Further this program does not do any error handling e.g.
* it does not check that entered postfix expression is valid
* or not.
* */

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

#define MAXSTACK 100 /* for max size of stack */
#define POSTFIXSIZE 100 /* define max number of charcters in postfix expression */

/* declare stack and its top pointer to be used during postfix expression
evaluation*/
int stack[MAXSTACK];
int top = -1; /* because array index in C begins at 0 */
/* can be do this initialization somewhere else */

/* define push operation */
void push(int item)
{

if (top >= MAXSTACK - 1) {
        printf("stack over flow");
return;
    }
else {
        top = top + 1;
        stack[top] = item;
    }
}

/* define pop operation */
int pop()
{
int item;
if (top < 0) {
        printf("stack under flow");
    }
else {
        item = stack[top];
        top = top - 1;
return item;
    }
}

/* define function that is used to input postfix expression and to evaluate it */
void EvalPostfix(char postfix[])
```

```
{

int i;
char ch;
int val;
int A, B;

    /* evaluate postfix expression */
for (i = 0; postfix[i] != ')'; i++) {
        ch = postfix[i];
if (isdigit(ch)) {
            /* we saw an operand,push the digit onto stack
ch - '0' is used for getting digit rather than ASCII code of digit */
            push(ch - '0');
        }
elseif (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            /* we saw an operator
* pop top element A and next-to-top elemnet B
* from stack and compute B operator A
*/
            A = pop();
            B = pop();

switch (ch) /* ch is an operator */
            {
case '*':
                val = B * A;
break;

case '/':
                val = B / A;
break;

case '+':
                val = B + A;
break;

case '-':
                val = B - A;
break;
            }

            /* push the value obtained above onto the stack */
            push(val);
        }
    }
    printf(" \n Result of expression evaluation : %d \n", pop());
}

int main()
{

int i;

    /* declare character array to store postfix expression */
char postfix[POSTFIXSIZE];
    printf("ASSUMPTION: There are only four operators(*, /, +, -) in an expression and
operand is single digit only.\n");
    printf(" \nEnter postfix expression,\npress right parenthesis ')' for end expression : ");

    /* take input of postfix expression from user */

for (i = 0; i <= POSTFIXSIZE - 1; i++) {
        scanf("%c", &postfix[i]);

if (postfix[i] == ')') /* is there any way to eliminate this if */
        {
break;
        } /* and break statement */
    }

    /* call function to evaluate postfix expression */

    EvalPostfix(postfix);

return 0;
}
```

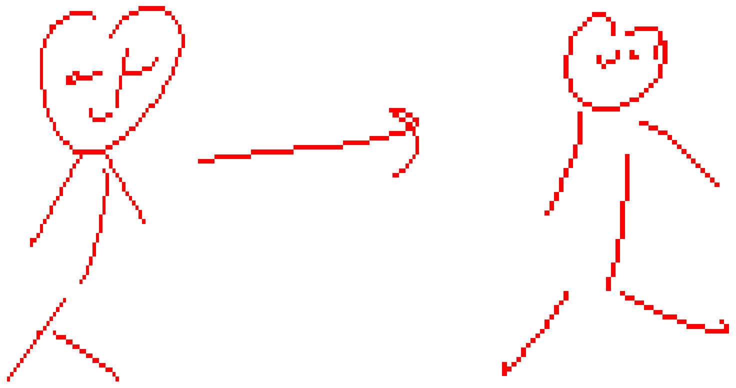
Output

```
First Run:
Enter postfix expression, press right parenthesis ')' for end expression : 456*+)
Result of expression evaluation : 34


Second Run:
Enter postfix expression, press right parenthesis ')' for end expression: 12345*+*+)
Result of expression evaluation: 47
```

Recursion:

Recursive definition and processes

Many of the processes in nature are recursive. Imagine a process that starts with an equilateral triangle and replace the middle 1/3rd of each line segment by another equilateral triangle. If we continue this process again and again then the shape begin to show more like a snowflake. This is actually called a Koch snowflake
You can see a nice demonstration of this process at Wikipedia.
Computer language rules are defined recursively. For example, a language can specify the rule for defining a variable as follows.



alpha = {a,b,…,z};
var = alpha | alpha {var}

This specifies that a variable in this language must be a finite combination of alpha characters.

Recursion is based on Mathematical notion of induction, where an inductive proof can be built by proving the theorem for n=1, then by showing in more general, that if an assumption that a theorem holds for n = r implies it holds for n = r+1, then theorem holds for all n. The induction is a powerful mathematical tool for proving many interesting theorems. For example, we can prove that, for any n >= 1,
1 + 2 + …. + n = (n/2)(n+1)
We prove this by assuming if the case for (n-1) holds true then that implies that the case for n holds as well.

Definition

A process can be recursive in nature and a Formal Definition that defines a recursive function can be given as follows.

A function that calls itself directly (or indirectly) to solve a smaller version of its task until a final call which does not require a self-call is a recursive function.

In any recursive process it is important to have a terminal condition (which we call the base condition) to assure that the recursive process will end at some point. There are many examples of recursion that we encounter everyday. A computer file system can be recursively examined by looking at folders, sub-folders etc. In other words, to find all files in a directory, we look at its sub-folders, sub-folders of sub-folders etc. The process can terminate when there are no more sub-folders to look at. A pseudo algorithm/function that examines all files under a particular directory can be written as:

```
function list(dir D) {
  for all files in D
      list files
  for all folders $D_i$ in the D
      list($D_i$)
```

The important thing to understand about recursion is that, it is the same algorithm we keep applying until we find a base case or termination condition. Recursion is a way to solve problems using a strategy call divide and conquer.

Divide and conquer approach is very common in programming. As we are dealing with large complex programs, it is always to better to think of a simple case first, then try to generalize it for more general cases. For example, if one is trying to solve a problem that involve n things, then it may be better to think how to solve the problem for n = 1. Often this is a trivial, but an important case. Now see if the solution to n = 2 problem can be obtained by using the solution to n = 1 case. We need to prove in general that we may be able to find the solution to n problem using the solution to n-1 problem. More strongly, we can find the solution to n problem using solutions to all the problems up to n-1. This is the basis for recursion.

Thinking Recursively

Let us start with a motivating example. Suppose you are given the task to find the least number of coins to give change for an amount less than 99 cents. A grocery clerk will probably do this without even thinking that he is applying a recursive algorithm in the process. Certainly our approach is to find the largest coin less than the amount and start with that coin. Once the largest coin is used we can look at the balance and apply the same algorithm to find the next largest coin less than the balance. Here is an example.

Assume we have to give change for 63 cents. So we do the following. In each step we choose the largest coin that is less than the balance. But the algorithm at each step is the same. Eventually, the balance goes to zero. Note that this may not work with all coin denominations. But it does work with US coins.

63 – 25 = 38 (1 coin)
38 – 25 = 13 (1 coin)
13 – 10 = 3 (1 coin)
3 – 1 = 2 (1 coin)
2 – 1 = 1 (1 coin)

1 – 1 = 0 (1 coin)

So we need 6 coins to give change. If we need to write this as a formal algorithm, we would say

```
numCoins(b) = 1 + numCoins(b-25)   if  b >= 25
numCoins(b) = 1 + numCoins(b-10)   if  25 > b >= 10
numCoins(b) = 1 + numCoins(b-5)   if  10 > b >= 5
numCoins(b) = 1 + numCoins(b-1)   if  5 > b >= 1
numCoins(0) = 0     { base case}
```

We note that solution to b problem can be found if we know the solution to b-25 etc.

Recursion is a powerful tool. Some algorithms required to solve complex problems can be recursive in nature. Recursive solutions are simple and elegant and mathematically sound.

Here is a recursive function that prints the digits of a number in decimal form.

```
public static void printDecimal(long n) {
    if (n > = 10)  printDecimal(n/10);
    System.out.print((char)('0'+(n%10)));
}
```

You can see the recursive nature of this Java function as it calls it-self during the process.

More Examples

You will encounter many problems in life that can be solved recursively. You need to have faith in your solution and that you can prove, if you can assume the solution to sub problem(s), then you can find the solution to the original problem. An interesting example of recursion is solving a Maze using recursion. Suppose you need to traverse a maze starting from one end and finding the exit from another end. It is computationally impossible to find all paths from beginning to the end. A pseudo algorithm for finding a recursive solution is as follows.

```
function MazeSolver(Maze M, currentCell){
    if (currentCell == destination)  return true;
    else { currentCell = visited;
            if (currentCell+1 is defined)  MazeSolver(M, currentCell + 1);
            if (currentCell-1 is defined)  MazeSolver(M, currentCell - 1);
            if (currentCell+M.numColumns is defined)  MazeSolver(M, currentCell +
M.numColumns);
            if (currentCell-M.numColumns is defined)  MazeSolver(M, currentCell +
M.numColumns);
```

```
        }
}
```

Another interesting example of an elegant recursive solution is the **Tower of Hanoi** problem. In this problem, you are given 3 pegs, named origin, destination, and intermediate and n disks that are stacked from largest to the smallest. Your goal is to move all n disks from origin to destination using intermediate under the following assumptions. You can move only one disk at a time and you cannot place a larger disk on the top of a smaller disk. The problem seems simple enough to solve if the number of disks is small. For example, if n = 3, then we can easily find a solution that involves 7 steps. But for general n, the problem seems difficult to solve. Not so, if you are willing to look at the problem like this. Suppose you consider the following. You know how to solve or move (n-1) disks from origin to intermediate using destination. Then you move the largest disk (one that is called n) from origin to destination. Now you move the rest of the disks (n-1) from intermediate to destination using the origin. An implementation of this can be found in code examples.

Tail Recursion

Recursive calls can occur at any point of the algorithm. For example, if we consider a for loop, that runs from 0 to n-1, then we know that the loop body is executed repeatedly with different values of n. Similarly, If the recursive call occurs at the end of the function, called tail recursion, the result is similar to a loop. That is, function executes all the statements before recursive call before jumping into the next recursive call.

Example

```
public void foo(int n) {

    if (n == 1) return;

    else { System.out.println(n); foo(n-1);}

}
```

Question: What is the output if foo(5) is called?

Head Recursion

If the recursive call occurs at the beginning of the function, called head recursion, the function saves the state of the program before jumping into the next function call. That is, function waits to evaluate statements until the exit condition is reached. The state of the program is saved in a stack (we will learn more about stacks later in the course)

Example

```
void  foo(int n) {
    If  (n==0) return;
   else  { foo(n-1);
              System.out.println(n);
            }
}
```

Question: What is the output if foo(5) is called?

recursion in C

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion(){
   recursion();/* function calls itself */
}

int main(){
   recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Number Factorial

The following example calculates the factorial of a given number using a recursive function −

```
#include<stdio.h>

unsignedlonglongint factorial(unsignedint i){

if(i <=1){
return1;
}
return i * factorial(i -1);
}
```

```
int  main(){
int i =12;
   printf("Factorial of %d is %d\n", i, factorial(i));
return0;
}
```

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

```
Factorial of 12 is 479001600
```

Fibonacci Series

The following example generates the Fibonacci series for a given number using a recursive function −

```
#include<stdio.h>

int fibonacci(int i){

if(i ==0){
return0;
}

if(i ==1){
return1;
}
return fibonacci(i-1)+ fibonacci(i-2);
}

int  main(){

int i;

for(i =0; i <10; i++){
      printf("%d\t\n", fibonacci(i));
}

return0;
}
```

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

```
0
1
1
2
3
5
```

```
8
13
21
34
```

example of recursion

```
#include<stdio.h>
int fibonacci(int);
void main ()
{
int n,f;
printf("Enter the value of n?");
scanf("%d",&n);
f = fibonacci(n);
printf("%d",f);
}
int fibonacci (int n)
{
if (n==0)
{
return 0;
}
else if (n == 1)
{
return 1;
}
else
{
return fibonacci(n-1)+fibonacci(n-2);
}
}
```

Output

```
Enter the value of n?12
144
```

Tower of Hanoi Problem

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.

2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

3. No disk may be placed on top of a smaller disk.

Approach :

Take an example for 2 disks :

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1 : Shift first disk from 'A' to 'B'.

Step 2 : Shift second disk from 'A' to 'C'.

Step 3 : Shift first disk from 'B' to 'C'.

The pattern here is :

Shift 'n-1' disks from 'A' to 'B'.

Shift last disk from 'A' to 'C'.

Shift 'n-1' disks from 'B' to 'C'.

Image illustration for 3 disks :

Examples:

Input : 2

Output : Disk 1 moved from A to B

Disk 2 moved from A to C

```
         Disk 1 moved from B to C

Input : 3
Output : Disk 1 moved from A to C
         Disk 2 moved from A to B
         Disk 1 moved from C to B
         Disk 3 moved from A to C
         Disk 1 moved from B to A
         Disk 2 moved from B to C
         Disk 1 moved from A to C
```

```
// C++ recursive function to
// solve tower of hanoi puzzle
#include <bits/stdc++.h>
usingnamespacestd;

voidtowerOfHanoi(intn, charfrom_rod,
                    charto_rod, charaux_rod)
{
    if(n == 1)
    {
        cout << "Move disk 1 from rod "<< from_rod <<
                            " to rod "<< to_rod<<endl;
        return;
    }
    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
    cout << "Move disk "<< n << " from rod "<< from_rod <<
                                 " to rod "<< to_rod << endl;
    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

// Driver code
intmain()
{
    intn = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
    return0;
}

// This is code is contributed by rathbhupendra
```

C

```
#include <stdio.h>

// C recursive function to solve tower of hanoi puzzle
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
    return 0;
}
```

Java

```
// Java recursive program to solve tower of hanoi puzzle

class GFG
{
    // Java recursive function to solve tower of hanoi puzzle
    static void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
    {
        if (n == 1)
        {
            System.out.println("Move disk 1 from rod " +  from_rod + " to rod " + to_rod);
            return;

        }

        towerOfHanoi(n-1, from_rod, aux_rod, to_rod);

        System.out.println("Move disk " + n + " from rod " +  from_rod + " to rod " + to_rod);

        towerOfHanoi(n-1, aux_rod, to_rod, from_rod);

    }


    //  Driver method

    public static void main(String args[])

    {

        int n = 4; // Number of disks

        towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods

    }

}
```

Output:

```
Tower of Hanoi Solution for 4 disks:

A: [4, 3, 2, 1] B: [] C: []

Move disk from rod A to rod B
A: [4, 3, 2] B: [1] C: []

Move disk from rod A to rod C
A: [4, 3] B: [1] C: [2]

Move disk from rod B to rod C
A: [4, 3] B: [] C: [2, 1]

Move disk from rod A to rod B
A: [4] B: [3] C: [2, 1]
```

```
Move disk from rod C to rod A
A: [4, 1] B: [3] C: [2]

Move disk from rod C to rod B
A: [4, 1] B: [3, 2] C: []

Move disk from rod A to rod B
A: [4] B: [3, 2, 1] C: []

Move disk from rod A to rod C
A: [] B: [3, 2, 1] C: [4]

Move disk from rod B to rod C
A: [] B: [3, 2] C: [4, 1]

Move disk from rod B to rod A
A: [2] B: [3] C: [4, 1]

Move disk from rod C to rod A
A: [2, 1] B: [3] C: [4]

Move disk from rod B to rod C
A: [2, 1] B: [] C: [4, 3]

Move disk from rod A to rod B
A: [2] B: [1] C: [4, 3]

Move disk from rod A to rod C
A: [] B: [1] C: [4, 3, 2]

Move disk from rod B to rod C
A: [] B: [] C: [4, 3, 2, 1]
```

Related Articles

- Recursive Functions
- Iterative solution to TOH puzzle
- Quiz on Recursion

https://www.youtube.com/watch?v=YstLjLCGmgg

References:

http://en.wikipedia.org/wiki/Tower_of_Hanoi

This article is contributed by **Rohit Thapliyal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

simulating recursion

When you are using a high-level language like Java, the stack operations described in the previous section are carried out automatically, and you can remain blissfully ignorant of the details. In part, the attractiveness of recursion lies in the fact that much of this complexity is hidden away, which makes it possible for you to concentrate on the algorithm itself.

At the same time, it is possible—though tedious—to perform the stack operations explicitly and thereby simulate the recursive operation. If you are using an older language that does not support recursion (such as FORTRAN), defining an explicit control stack is often the only way to code a recursive algorithm. And even though there is no need to simulate recursion in modern languages like Java, doing so can sometimes provide a more precise understanding of the underlying mechanics.

At one level, the simplest way to simulate recursion is to model the stack in much the same way that the underlying machine does, pushing the values of each argument individually prior to a method call and popping those values off when the method returns. Such a strategy, however, does not take advantage of Java's data structuring capabilities. Since Java makes it possible to represent structures that are more complex than individual stack entries, it seems preferable to simulate recursion in Java at a somewhat higher level of abstraction. For instance, one strategy that seems particularly appropriate is to define ...

Some computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function **α** either calls itself directly or calls a function **β** that in turn calls the original function **α**. The function **α** is called recursive function.

Example − a function calling itself.

```
intfunction(intvalue){
```

```
if(value<1)
return;
function(value-1);

   printf("%d ",value);
}
```

Example − a function that calls another function which in turn calls it again.

```
int function1(int value1){
if(value1 <1)
return;
   function2(value1 -1);
   printf("%d ",value1);
}
int function2(int value2){
   function1(value2);
}
```

Properties

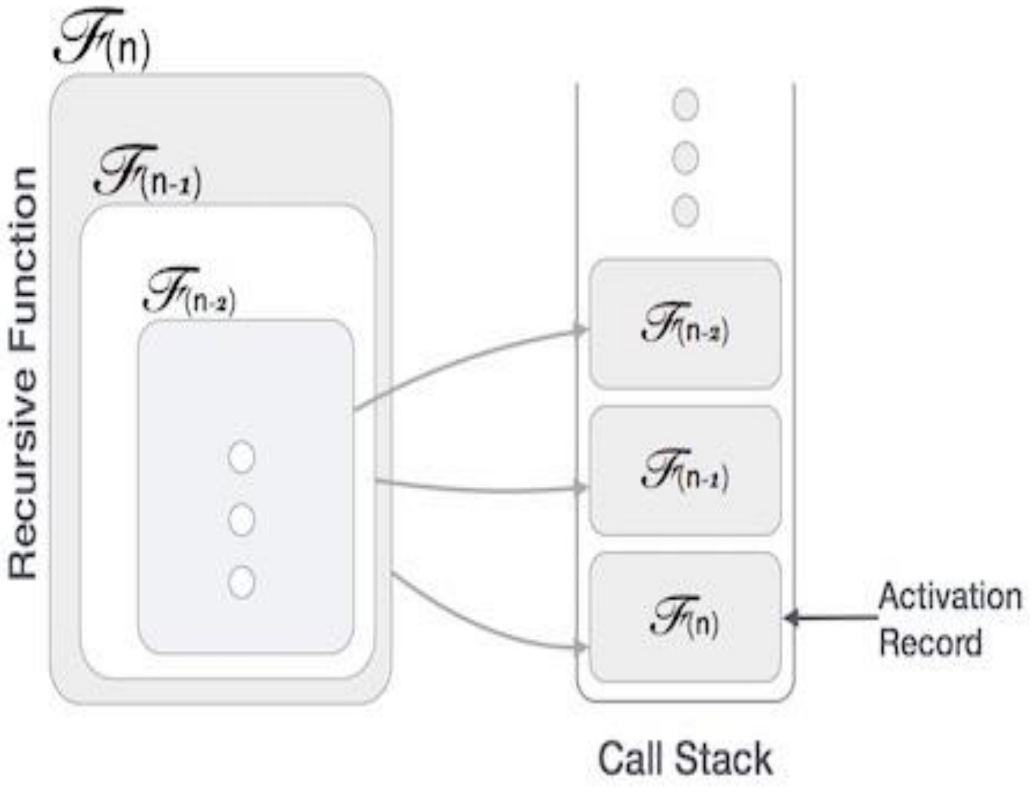
A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

- **Base criteria −** There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach −** The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Implementation

Many programming languages implement recursion by means of **stacks**. Generally, whenever a function (**caller**) calls another function (**callee**) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee.

This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.



This activation record keeps the information about local variables, formal parameters, return address and all information passed to the caller function.

Analysis of Recursion

One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

Time Complexity

In case of iterations, we take number of iterations to count the time complexity. Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made. A call made to a function is O(1), hence the (n) number of times a recursive call is made makes the recursive function O(n).

Space Complexity

Space complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.

Backtracking

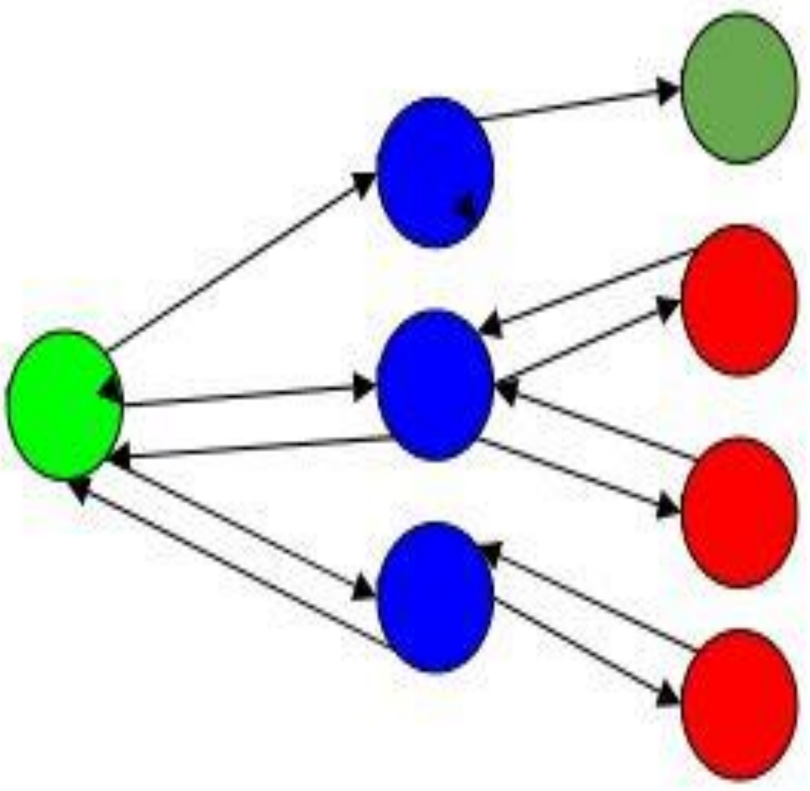
Backtracking is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.

Backtracking algorithm is applied to some specific types of problems,

- Decision problem used to find a feasible solution of the problem.
- Optimisation problem used to find the best solution that can be applied.
- Enumeration problem used to find the set of all feasible solutions of the problem.

In backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.

Example,



Here,

Green is the start point, blue is the intermediate point, red are points with no feasible solution, dark green is end solution.

Here, when the algorithm propagates to an end to check if it is a solution or not, if it is then returns the solution otherwise backtracks to the point one step behind it to find track to the next point to find solution.

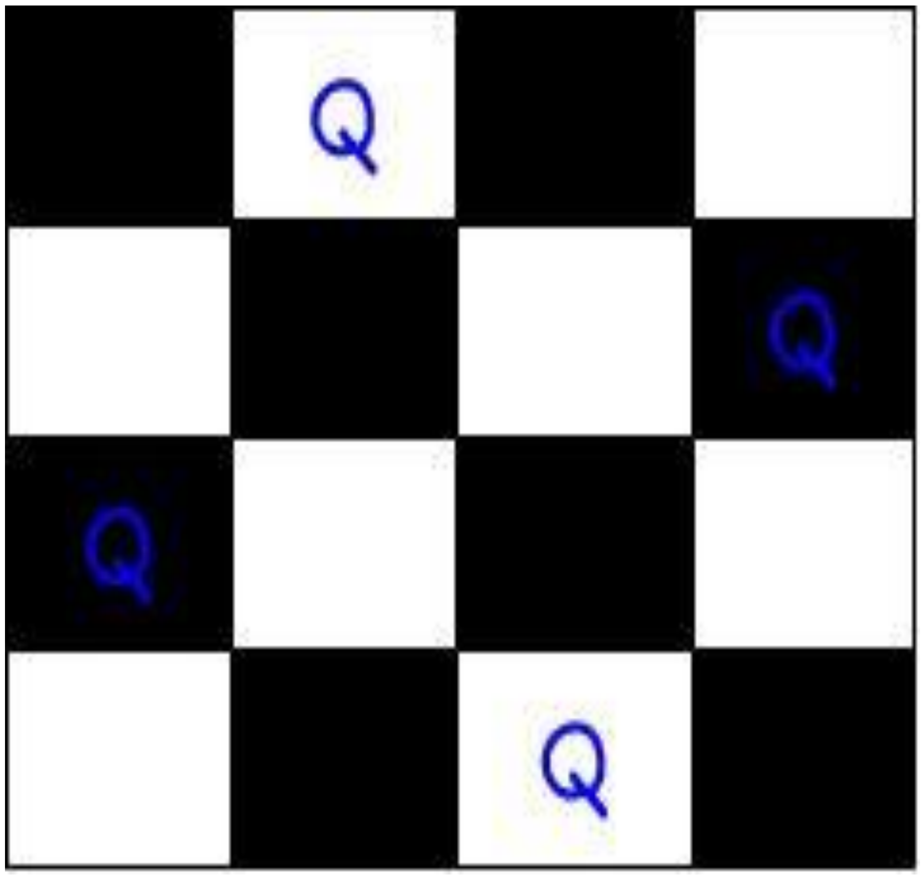
Algorithm

```
Step 1 − if current_position is goal, return success
Step 2 − else,
Step 3 − if current_position is an end point, return failed.
Step 4 − else, if current_position is not end point, explore and repeat above steps.
```

Let’s use this backtracking problem to find the solution to **N-Queen Problem**.

In N-Queen problem, we are given an NxN chessboard and we have to place n queens on the board in such a way that no two queens attack each other. A queen will attack another queen if it is placed in horizontal, vertical or diagonal points in its way. Here, we will do 4-Queen problem.

Here, the solution is −



Here, the binary output for n queen problem with 1's as queens to the positions are placed.

```
{0 , 1 , 0 , 0}
{0 , 0 , 0 , 1}
{1 , 0 , 0 , 0}
{0 , 0 , 1 , 0}
```

For solving n queens problem, we will try placing queen into different positions of one row. And checks if it clashes with other queens. If current positioning of queens if there are any two queens attacking each other. If they are attacking, we will backtrack to previous location of the queen and change its positions. And check clash of queen again.

Algorithm

```
Step 1 − Start from 1st position in the array.

Step 2 − Place queens in the board and check. Do,
   Step 2.1 − After placing the queen, mark the position as a part of the solution and then recursively check if this will lead to a solution.
   Step 2.2 − Now, if placing the queen doesn't lead to a solution and trackback and go to step (a) and place queens to other rows.
   Step 2.3 − If placing queen returns a lead to solution return TRUE.
Step 3 − If all queens are placed return TRUE.

Step 4 − If all rows are tried and no solution is found, return FALSE.
```

recursive algorithms

Recursion is a powerful problem solving tool. In this lesson we consider few well-known recursive algorithms. We present them first, since it is easy to understand why they are recursive. Recursive definitions are in fact mathematical definitions that can be directly translated into code and also prove the correctness. Let us start with Binary Search Algorithm.

Binary Search Algorithm

Looking up a word in the telephone directory? Here is the perfect algorithm. Find the middle of the directory. If the word you are searching is alphabetically higher than the middle word, then focus only on the right half of the book, else focus on the left half of the book. Who knows, you may be lucky and find the word in the middle. If you don't find the word, then focus on half the directory and continue the same process. The key here is that every search effort reduces the search area by a half. So if we are dealing with a data set of size 10^{20} (that is about a million records), the word can be found or we

can determine that the word is not in the directory in no more than 20 comparisons. Only assumption you are making here is that directory is a sorted list. Here is an algorithm for searching a dictionary (or directory)
Algorithm:

```
Search(dictionary)
{
  if (Dictionary has only 1 page)
    Sequentially search page for word
  else
  {
    Open the dictionary to the middle page
    Determine which half of the dictionary the word is in
    if (The word is in the first half)
      Search(first half of dictionary)   // ignore second half
    else
      Search(second half of dictionary   // ignore first half
  }
}
```

Note that the problem presents all characteristics of a recursive algorithm. It does have a terminal case to end recursion and it does express the solution to larger problem using solution to one or smaller sub problems. One of the important assumptions we made about the binary search here is that list is sorted. Binary search cannot be performed if the list is not sorted. Binary search is recursive (it can be implemented iteratively as well) and its complexity is O(log n) compared to the complexity of linear search that is of O(n).

Factorial

Factorial is an important mathematical function. A recursive definition of factorial is as follows.

$$n! = \begin{cases} 1 & \text{if } n \leq 1 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

The above definition can be directly translated into code as follows. Note that we have a base case as well as a recursive case.

```
main()
{
  inti = 3;              // 1
  cout<< f(i) <<endl;    // 2
}
int f(int a1)
{
   if (a1 <= 1)              // 3
      return 1;              // 4
   else                      // 5
      return a1 * f(a1 - 1);  // 6
}
```

Non-recursive version:

```
int fact(int n)
{
   int i;
   int prod = 1;
   for (i = 1; i <= n; i++)
      prod *= i;
   return prod;
}
```

The Fibonacci Sequence

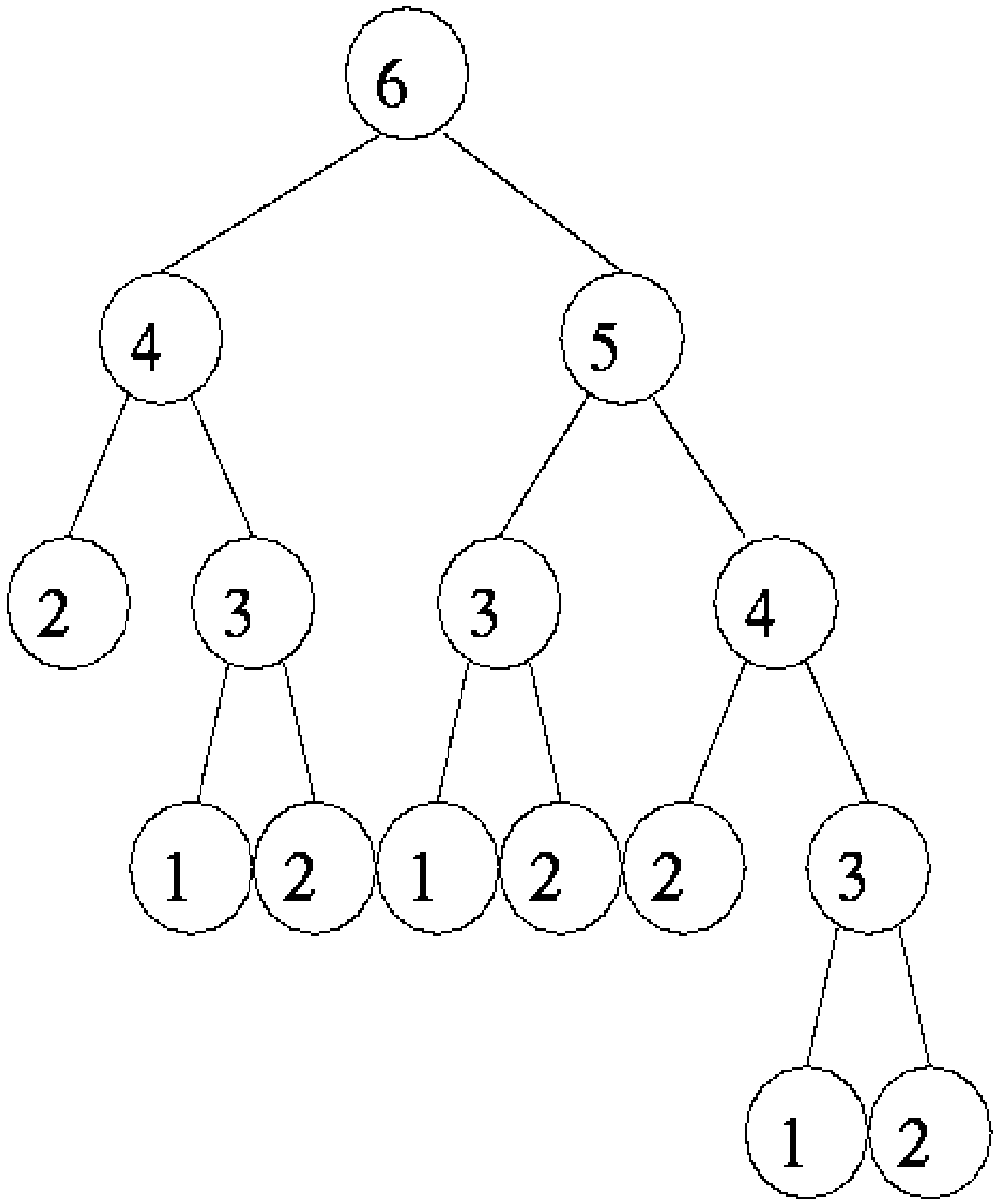
Fibonacci Definition:

$$\mathbf{Fib}(n) = \begin{cases} 1 & \text{if } n \leq 2 \\ \mathbf{Fib}(n-1) + \mathbf{Fib}(n-2) & \text{otherwise} \end{cases}$$

Consider:

```
int fib(intval)
{
   if (val<= 2)
      return 1;
   else
      return fib(val - 1) + fib(val - 2);
}
```

Call graph for fib(6):



Non-recursive version:

```
int fib(intval)
{
  int current = 1;
  int old = 1;
  int older = 1;
  val -=2;
  while (val> 0)
  {
    current = old + older;
```

```
      older = old;
      old = current;
      --val;
   }
   return current;
}
```

Greatest Common Divisor

Definition:

$$\gcd(a,b) = \begin{cases} b & \text{if } b \text{ divides } a \\ \gcd(b, \text{remainder}(a,b)) & \text{otherwise} \end{cases}$$

```
intgcd(int a, int b)
{
   int remainder = a % b;
   if (remaider == 0)
      return b;
   else
      return gcd(b, remainder);
}
```

principles of recursion

The recursion is a process by which a function calls itself. We use recursion to solve bigger problem into smaller sub-problems. One thing we have to keep in mind, that if each sub-problem is following same kind of patterns, then only we can use the recursive approach.

A recursive function has two different parts. The base case and the recursive case. The base case is used to terminate the task of recurring. If base case is not defined, then the function will recur infinite number of times (Theoretically).

In computer program, when we call one function, the value of the program counter is stored into the internal stack before jumping into the function area. After completing the task, it pops out the address and assign it into the program counter, then resume the task. During recursive call, it will store the address multiple times, and jumps into the next function call statement. If one base case is not defined, it will recur again and again, and store address into stack. If the stack has no space anymore, it will raise an error as “Internal Stack Overflow”.

One example of recursive call is finding the factorial of a number. We can see that the factorial of a number n = n! is same as the n * (n-1)!, again it is same as n * (n - 1) * (n -

2)!. So if the factorial is a function, then it will be called again and again, but the argument is decreased by 1. When the argument is 1 or 0, it will return 1. This could be the base case of the recursion.

Example

```
#include<iostream>

usingnamespace std;

long fact(long n){

   if(n <=1)

   return1;

   return n * fact(n-1);

}

main(){

   cout <<"Factorial of 6: "<< fact(6);

}
```

Output

```
Factorialof6:720
```

tail recursion

Here we will see what is tail recursion. The tail recursion is basically using the recursive function as the last statement of the function. So when nothing is left to do after coming back from the recursive call, that is called tail recursion. We will see one example of tail recursion.

Example

```
#include<iostream>

usingnamespace std;

void printN(int n){

   if(n <0){
    return;
  }
  cout << n <<" ";
  printN(n -1);
}
int main(){
  printN(10);
}
```

Output

```
10 9 8 7 6 5 4 3 2 1 0
```

The tail recursion is better than non-tail recursion. As there is no task left after the recursive call, it will be easier for the compiler to optimize the code. When one function is called, its address is stored inside the stack. So if it is tail recursion, then storing addresses into stack is not needed.

We can use factorial using recursion, but the function is not tail recursive. The value of fact(n-1) is used inside the fact(n).

```
long fact(int n){
  if(n <=1)
    return1;
  n * fact(n-1);
}
```

We can make it tail recursive, by adding some other parameters. This is like below −

```
long fact(long n,long a){
  if(n ==0)
    return a;
  return fact(n-1, a*n);
}
```

removal of recursion

Each recursive function can be transformed into an equivalent non-recursive function, subject to using iterative constructs and possibly extra data structures for simulating the call stack. Depending on programming language and form of recursion, different algorithms can be applied.
Consider the following recursive formulation of factorial:

```
publicstaticlongfactorial(intn){
        if(n<=1)
                return1;
        else
                returnn*factorial(n-1);
}
```

There are various iterative forumulations of factorial. There are different approaches towards deriving an iterative formulation from a recursive. The following example illustrates how single recursion can be transformed into iteration while a stack is used to keep track of variables for each call frame.

```
// Abstract program pointer
publicenumPointer{
        calling,returning
};

publicstaticlongfactorial(intn){
        intr=0;// Result
        Stacks=newStack();// Prepare call stack
        Pointerp=Pointer.calling;
        do
                switch(p){
                casecalling:
                        if(n<=1){
                                // Return from base case
                                r=1;
```

```
                                        p=Pointer.returning;
                              }else{
                                        // Recursive call
                                        s.push(n);// Backup call frame
                                        n=n-1;// Compute argument
                              }
                              break;
                    casereturning:
                              // Return from recursive call
                              n=(int)s.peek();// Restore variables
                              s.pop();// Destroy call frame
                              r=n*r;// Use recursive result
                              break;
                    }
          while(!(s.isEmpty()));
          returnr;
}
```

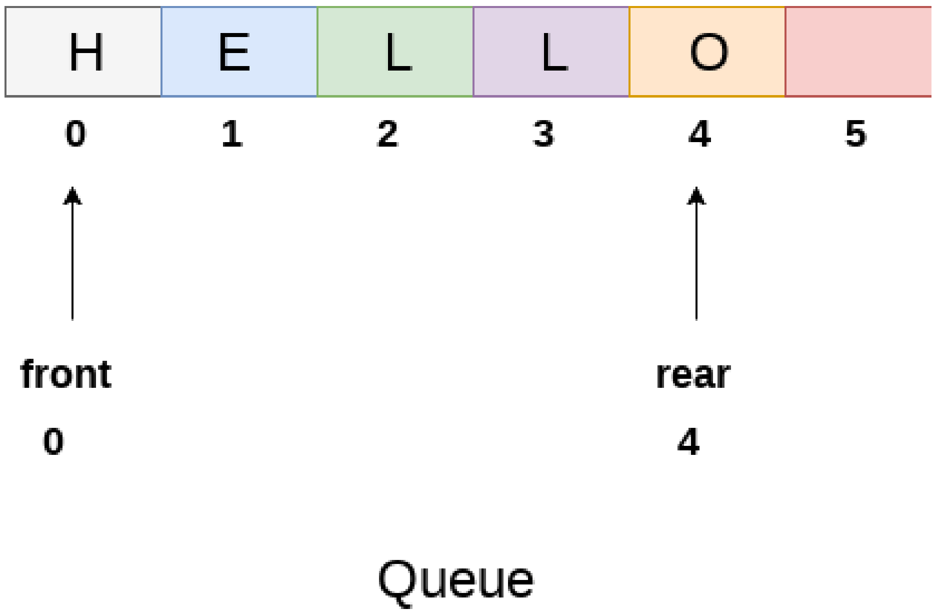
The original function only uses one local variable n for the argument of the function. Thus, the simulated call stack maintains ints, indeed. If a function had additional local variable declarations, then these would need to be pushed together in one record. The iterative formulation loops as long as the call stack is not empty while the loop's body is executed at least once for the base case. The loop can be in two modes: calling or returning. In calling mode, the parts of the function for the base case or prior to a recursive call are evaluated. In returning mode, the parts of the function past a recursive call are evaluated. In this sense, the mode and the switch statement on it can be seen as the abstract model of a program pointer: the pointer points to the beginning of a function or the remaining operations past the recursive call.

Unit - II

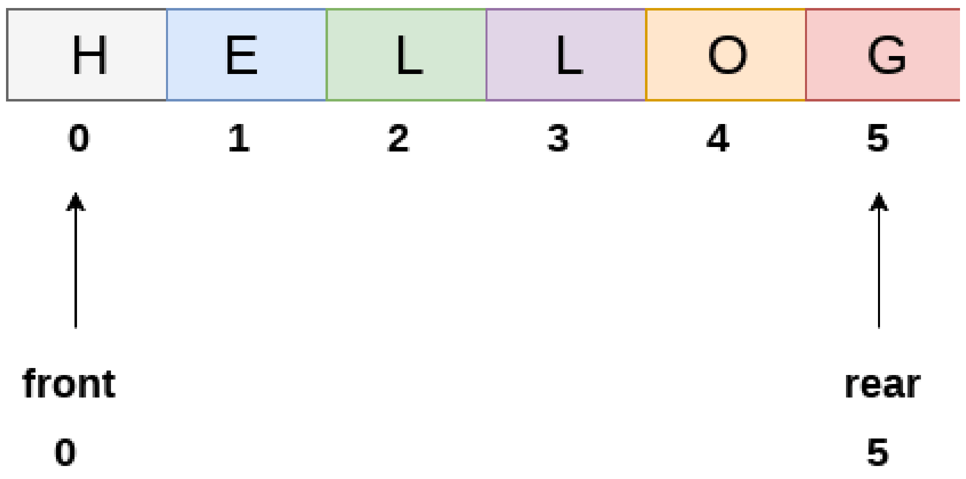
Queues:

Array and linked representation and implementation of queues

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.

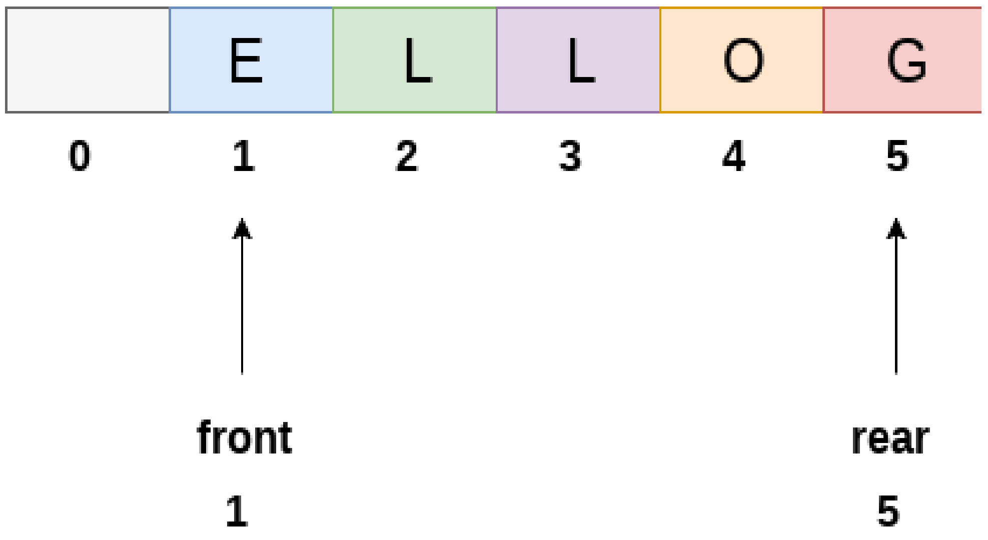


The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

```
o Step 1: IF REAR = MAX - 1
  Write OVERFLOW
  Go to step
  [END OF IF]
o Step 2: IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
o Step 3: Set QUEUE[REAR] = NUM
o Step 4: EXIT
```

C Function

```c
void insert (int queue[], int max, int front, int rear, int item)
{
if (rear + 1 == max)
{
printf("overflow");
}
else
{
if(front == -1 && rear == -1)
{
front = 0;
rear = 0;
}
else
{
rear = rear + 1;
}
queue[rear]=item;
}
}
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

```
o Step 1: IF FRONT = -1 or FRONT > REAR
          Write UNDERFLOW
          ELSE
          SET VAL = QUEUE[FRONT]
          SET FRONT = FRONT + 1
          [END OF IF]
o Step 2: EXIT
```

C Function

```
int delete (int queue[], int max, int front, int rear)
{
int y;
if (front == -1 || front > rear)

{
printf("underflow");
}
else
{
y = queue[front];
if(front == rear)
{
front = rear = -1;
else
front = front + 1;
}
return y;
}
}
```

Menu driven program to implement queue using array

```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
int choice;
while(choice != 4)
{
printf("\n*************************Main Menu*****************************\n");
printf("\n==========================================================
========\n");
printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
printf("\nEnter your choice ?");
scanf("%d",&choice);
switch(choice)
{
case 1:
insert();
break;
case 2:
delete();
break;
case 3:
display();
break;
case 4:
exit(0);
break;
default:
printf("\nEnter valid choice??\n");
}
}
}
void insert()
{
int item;
printf("\nEnter the element\n");
scanf("\n%d",&item);
if(rear == maxsize-1)
{
printf("\nOVERFLOW\n");
return;
}
if(front == -1 && rear == -1)
{
front = 0;
rear = 0;
}
else
{
rear = rear+1;
}
queue[rear] = item;
printf("\nValue inserted ");

}
void delete()
{
int item;
if (front == -1 || front > rear)
{
printf("\nUNDERFLOW\n");
return;

}
else
{
item = queue[front];
if(front == rear)
```

```
{
front = -1;
rear = -1 ;
}
else
{
front = front + 1;
}
printf("\nvalue deleted ");
}


}

void display()
{
int i;
if(rear == -1)
{
printf("\nEmpty queue\n");
}
else
{   printf("\nprinting values .....\n");
for(i=front;i<=rear;i++)
{
printf("\n%d\n",queue[i]);
}
}
}
```

Output:

```
************Main Menu**************

==============================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit
```

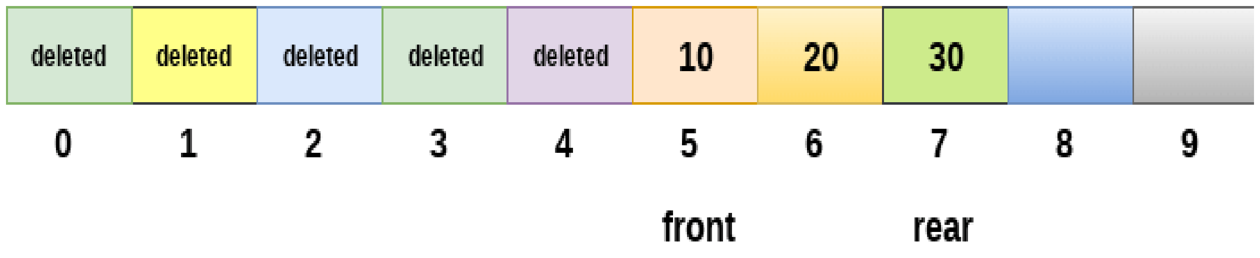
```
Enter your choice ?1

Enter the element
123

Value inserted

*************Main Menu**************

===============================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
90

Value inserted

*************Main Menu**************

====================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?2

value deleted

*************Main Menu**************
===============================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3
```

```
printing values .....

90

*************Main Menu**************

==============================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?4
```

Drawback of array implementation

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

- **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



limitation of array representation of queue

The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

- **Deciding the array size**

On of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

Linked List implementation of Queue

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

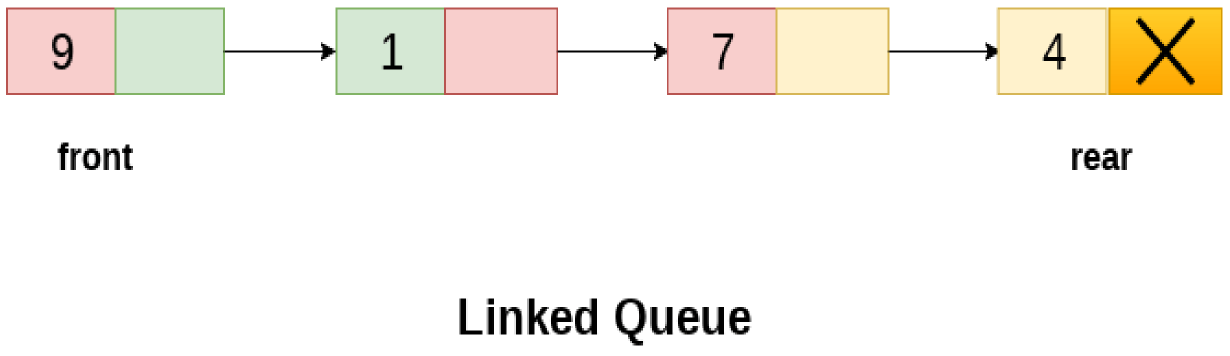
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

```
Ptr = (struct node *) malloc (sizeof(struct node));
```

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
ptr -> data = item;
if(front == NULL)
{
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
}
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

```
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
```

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

- **Step 1:** Allocate the space for the new node PTR
- **Step 2:** SET PTR -> DATA = VAL
- **Step 3:** IF FRONT = NULL
  SET FRONT = REAR = PTR
  SET FRONT -> NEXT = REAR -> NEXT = NULL
  ELSE
  SET REAR -> NEXT = PTR
  SET REAR = PTR
  SET REAR -> NEXT = NULL
  [END OF IF]
- **Step 4:** END

C Function

```
void insert(struct node *ptr, int item; )
{


ptr = (struct node *) malloc (sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW\n");
return;
}
else
{
ptr -> data = item;
if(front == NULL)
{
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
}
else
{
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
}
}
}
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

```
ptr = front;
front = front -> next;
free(ptr);
```

The algorithm and C function is given as follows.

Algorithm

```
○ Step 1: IF FRONT = NULL
          Write " Underflow "
          Go to Step 5
          [END OF IF]
○ Step 2: SET PTR = FRONT
○ Step 3: SET FRONT = FRONT -> NEXT
○ Step 4: FREE PTR
○ Step 5: END
```

C Function

```
void delete (struct node *ptr)
{
if(front == NULL)
{
printf("\nUNDERFLOW\n");
return;
}
else
{
ptr = front;
front = front -> next;
free(ptr);
}
}
```

Menu-Driven Program implementing all the operations on Linked Queue

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
int choice;
while(choice != 4)
{
printf("\n***********************Main Menu***************************\n");
printf("\n============================================================
========\n");
printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
printf("\nEnter your choice ?");
scanf("%d",& choice);
switch(choice)
{
case 1:
insert();
break;
case 2:
delete();
break;
case 3:
display();
break;
case 4:
exit(0);
break;
default:
printf("\nEnter valid choice??\n");
}
}
}
void insert()
{
struct node *ptr;
int item;

ptr = (struct node *) malloc (sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW\n");
return;
}
else
{
printf("\nEnter value?\n");
scanf("%d",&item);
ptr -> data = item;
if(front == NULL)
{
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
}
else
{
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
}
}
```

```
}
void delete ()
{
struct node *ptr;
if(front == NULL)
{
printf("\nUNDERFLOW\n");
return;
}
else
{
ptr = front;
front = front -> next;
free(ptr);
}
}
void display()
{
struct node *ptr;
ptr = front;
if(front == NULL)
{
printf("\nEmpty queue\n");
}
else
{   printf("\nprinting values .....\n");
while(ptr != NULL)
{
printf("\n%d\n",ptr -> data);
ptr = ptr -> next;
}
}
}
```

Output:

```
***********Main Menu**********

==============================

1.insert an element
2.Delete an element
```

```
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
123

***********Main Menu**********

==============================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
90

***********Main Menu**********

==============================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

123

90

***********Main Menu**********

==============================

1.insert an element
2.Delete an element
3.Display the queue
```

```
4.Exit

Enter your choice ?2

***********Main Menu**********

==============================
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

90

***********Main Menu**********

==============================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?4
```

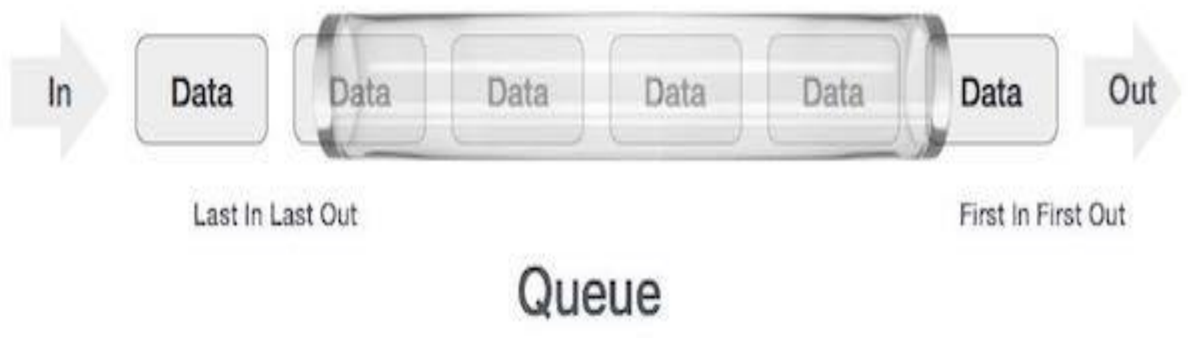
Operations on Queue:

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek
   return queue[front]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek(){
return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull

if rear equals to MAXSIZE
returntrue
else
returnfalse
   endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull(){
if(rear == MAXSIZE -1)
returntrue;
else
returnfalse;
}
```

isempty()

Algorithm of isempty() function −

Algorithm

```
begin procedure isempty
```

```
if front is less than MIN  OR front is greater than rear
returntrue
else
returnfalse
   endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

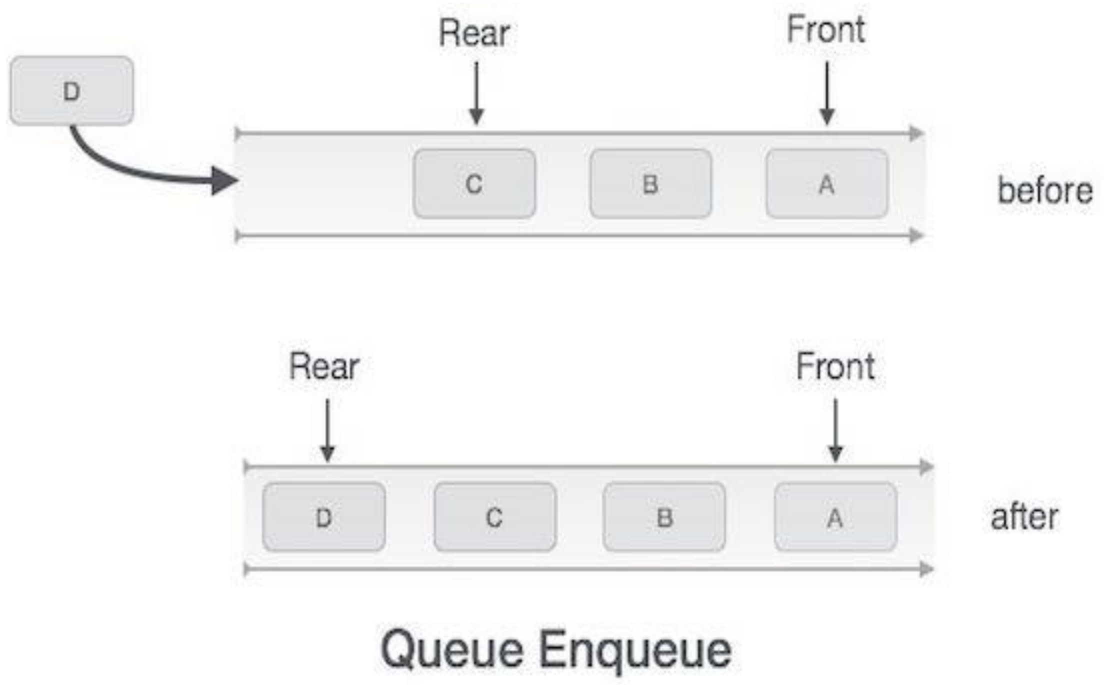
```
bool isempty(){
if(front <0|| front > rear)
returntrue;
else
returnfalse;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)

if queue is full
return overflow
   endif

   rear ← rear +1
   queue[rear]← data
returntrue

end procedure
```

Implementation of enqueue() in C programming language −

Example

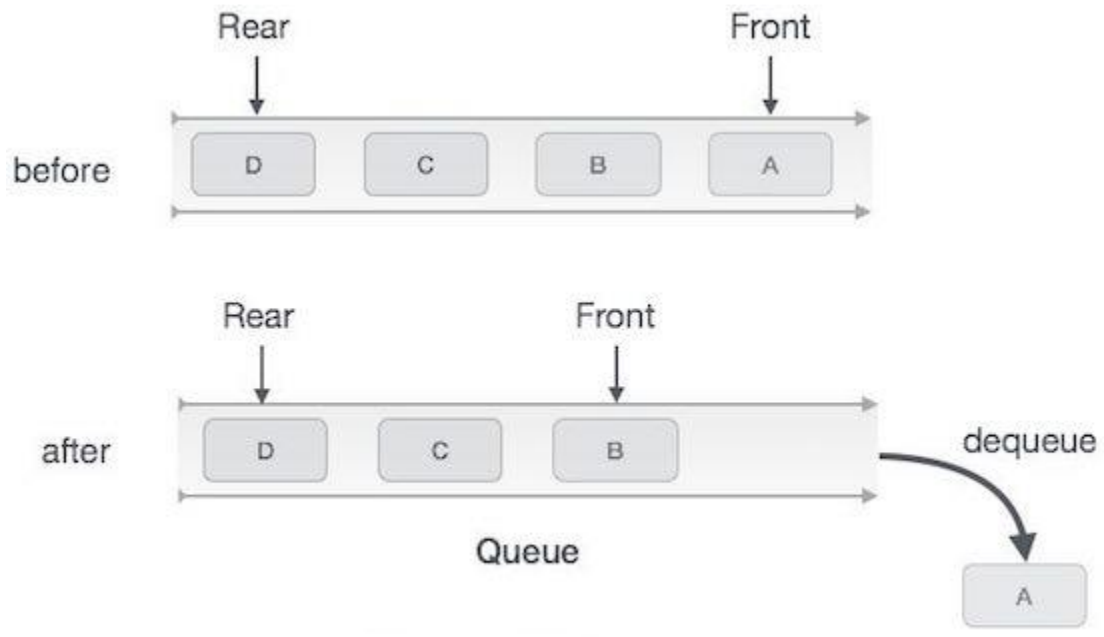
```
int enqueue(int data)
if(isfull())
return0;

   rear = rear +1;
   queue[rear]= data;
```

```
return1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue

if queue is empty
return underflow
endif

   data = queue[front]
   front ← front +1
returntrue
```

```
end procedure
```

Implementation of dequeue() in C programming language −

Example

```
int dequeue(){
if(isempty())
return0;

int data = queue[front];
   front = front +1;

return data;
}
```

Create

```
#include< stdio.h >
int main()
{

    FILE *fp;   /* file pointer*/
char fName[20];

    printf("Enter file name to create :");
    scanf("%s",fName);

    /*creating (open) a file, in “w”: write mode*/
    fp=fopen(fName,"w");
    /*check file created or not*/
if(fp==NULL)
    {
        printf("File does not created!!!");
        exit(0); /*exit from program*/
    }

    printf("File created successfully.");
return 0;
}
```

Output

```
    Run 1:
    Enter file name to create : file1.txt
```

```
File created successfully.

Run 2:
Enter file name to create : d:/file1.txt
File created successfully.

“file will be created in d: drive”.

Run 3:
Run 1:
Enter file name to create : h:/file1.txt
File does not created!!!
```

Add

A program to add two numbers takes to numbers and does their mathematical sum and gives it to another variable that stores its sum.

Example Code

```
#include<stdio.h>

int main(void){

   int a =545;

   int b =123;

   printf("The first number is %d and the second number is %d \n", a , b);

   int sum = a + b;

   printf("The sum of two numbers is %d", sum);

   return0;

}
```

Output

```
The first number is 545 and the second number is 123
The sum of two numbers is 668
```

Delete

The cast-expression argument must be a pointer to a block of memory previously allocated for an object created with the new operator. The **delete** operator has a result of type **void** and therefore does not return a value. For example:

C++Copy

```
CDialog* MyDialog = new CDialog;
// use MyDialog
delete MyDialog;
```

Using **delete** on a pointer to an object not allocated with **new** gives unpredictable results. You can, however, use **delete** on a pointer with the value 0. This provision means that, when **new** returns 0 on failure, deleting the result of a failed **new** operation is harmless. For more information, see The new and delete Operators.

The **new** and **delete** operators can also be used for built-in types, including arrays. If pointer refers to an array, place empty brackets ([]) before pointer:

C++Copy

```
int* set = newint[100];
//use set[]
delete [] set;
```

Using the **delete** operator on an object deallocates its memory. A program that dereferences a pointer after the object is deleted can have unpredictable results or crash.

When **delete** is used to deallocate memory for a C++ class object, the object's destructor is called before the object's memory is deallocated (if the object has a destructor).

If the operand to the **delete** operator is a modifiable l-value, its value is undefined after the object is deleted.

If the /sdl (Enable additional security checks) compiler option is specified, the operand to the **delete** operator is set to an invalid value after the object is deleted.

Using delete

There are two syntactic variants for the delete operator: one for single objects and the other for arrays of objects. The following code fragment shows how they differ:

C++Copy

```
// expre_Using_delete.cpp
structUDType
{
};

intmain()
{
// Allocate a user-defined object, UDObject, and an object
//  of type double on the free store using the
//  new operator.
   UDType *UDObject = new UDType;
double *dObject = newdouble;
// Delete the two objects.
delete UDObject;
delete dObject;
// Allocate an array of user-defined objects on the
// free store using the new operator.
   UDType (*UDArr)[7] = new UDType[5][7];
// Use the array syntax to delete the array of objects.
delete [] UDArr;
}
```

The following two cases produce undefined results: using the array form of delete (delete []) on an object, and using the nonarray form of delete on an array.

Example

For examples of using **delete**, see new operator.

How delete works

The delete operator invokes the function **operator delete**.

For objects not of class type (class, struct, or union), the global delete operator is invoked. For objects of class type, the name of the deallocation function is resolved in global scope if the delete expression begins with the unary scope resolution operator (::). Otherwise, the delete operator invokes the destructor for an object prior to deallocating memory (if the pointer is not null). The delete operator can be defined on a per-class basis; if there is no such definition for a given class, the global operator delete is invoked. If the delete expression is used to deallocate a class object whose static type has a virtual destructor, the deallocation function is resolved through the virtual destructor of the dynamic type of the object.

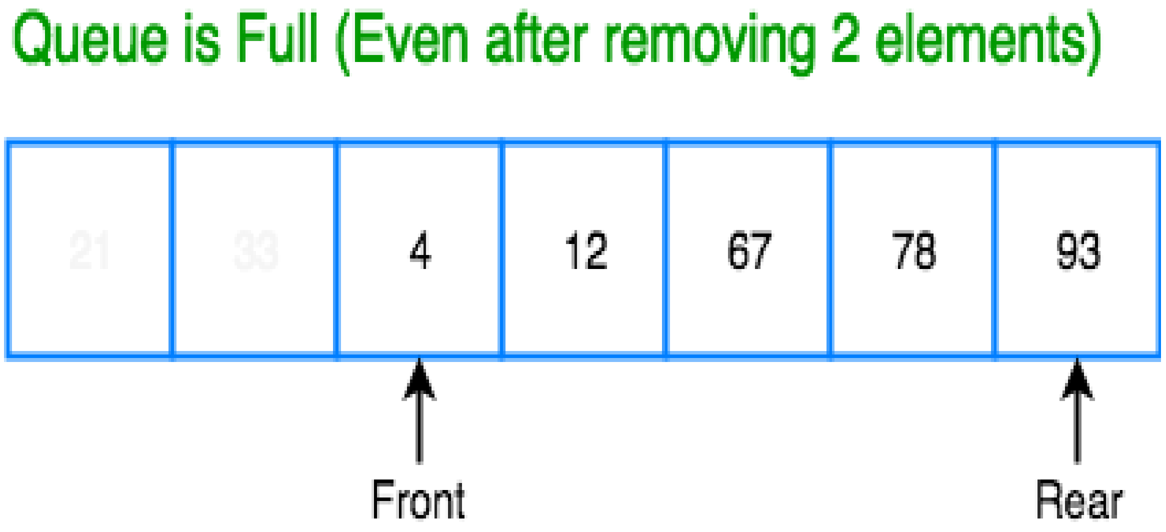
Full and Empty:

Circular queue

Before we start to learn about Circular queue, we should first understand, why we need a circular queue, when we already have linear queue data structure.

In a Linear queue, once the queue is completely full, it's not possible to insert more elements. Even if we dequeue the queue to remove some of the elements, until the queue is reset, no new elements can be inserted. You must be wondering why?

When we **dequeue** any element to remove it from the queue, we are actually moving the **front** of the queue forward, thereby reducing the overall size of the queue. And we cannot insert new elements, because the **rear** pointer is still at the end of the queue.

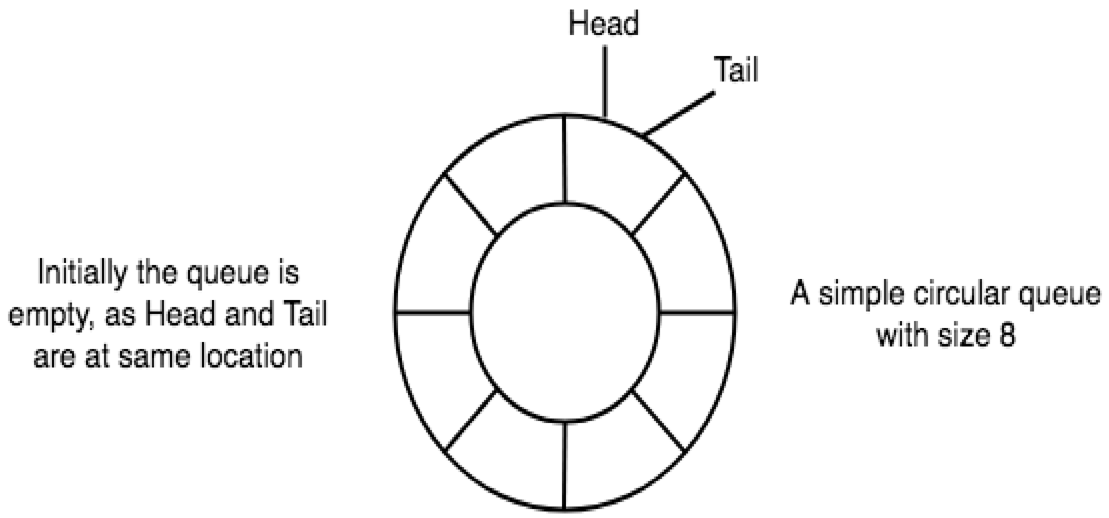


The only way is to reset the linear queue, for a fresh start.

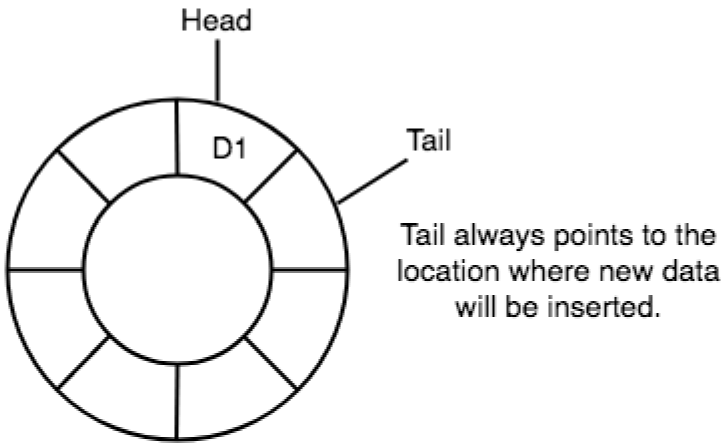
Circular Queue is also a linear data structure, which follows the principle of **FIFO**(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

Basic features of Circular Queue

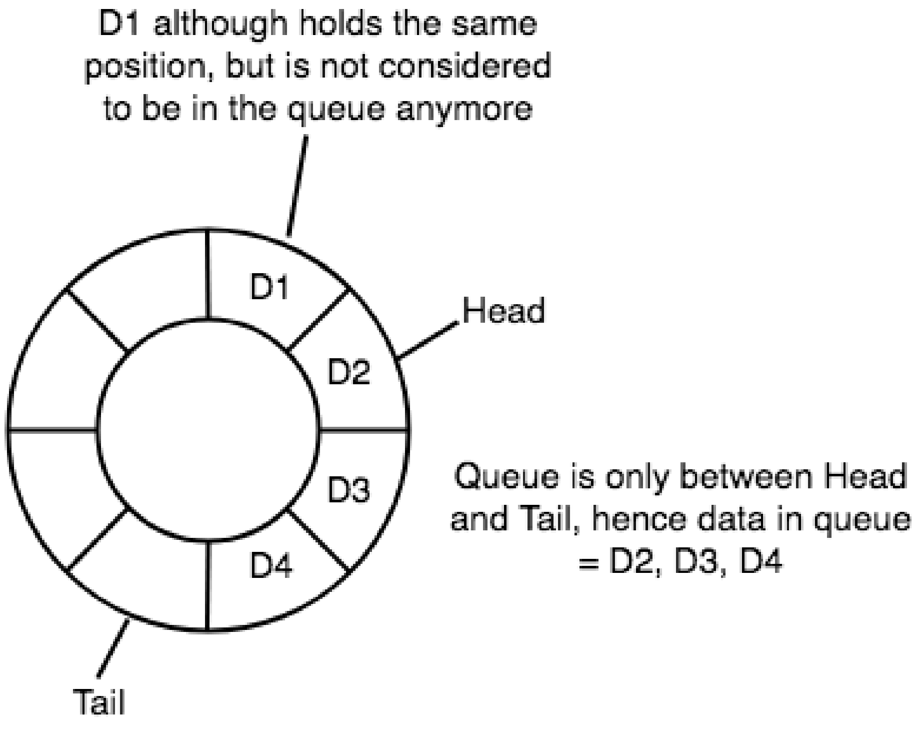
1. In case of a circular queue, head pointer will always point to the front of the queue, and tail pointer will always point to the end of the queue.
2. Initially, the head and the tail pointers will be pointing to the same location, this would mean that the queue is empty.


Head
Tail
Initially the queue is empty, as Head and Tail are at same location
A simple circular queue with size 8

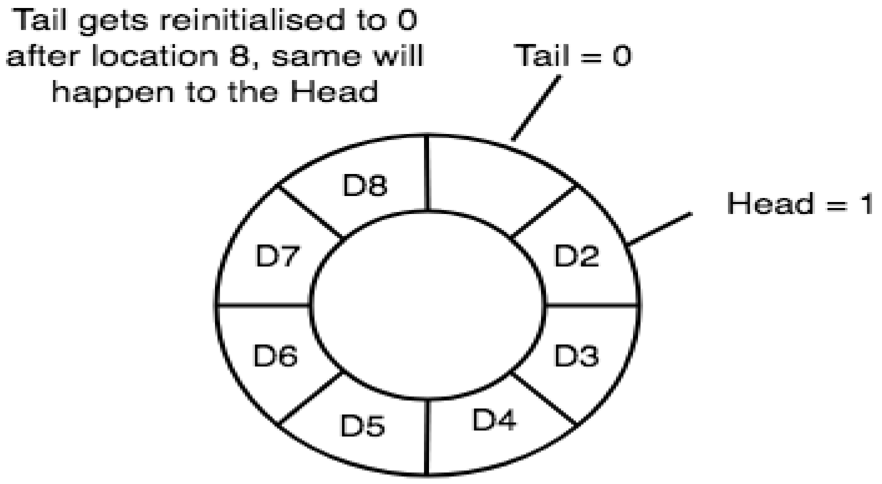
3. New data is always added to the location pointed by the tail pointer, and once the data is added, tail pointer is incremented to point to the next available location.


Head
D1
Tail
Tail always points to the location where new data will be inserted.

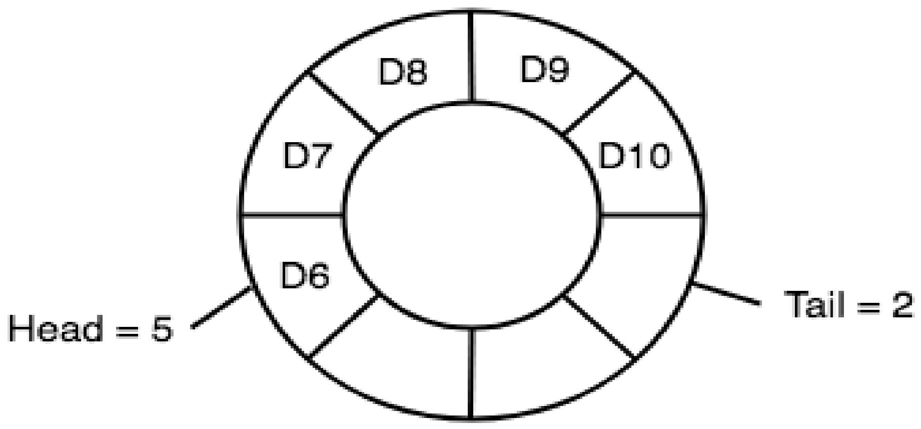
1. In a circular queue, data is not actually removed from the queue. Only the head pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between head and tail, hence the data left outside is not a part of the queue anymore, hence removed.


D1 although holds the same position, but is not considered to be in the queue anymore
D1
Head
D2
D3
D4
Queue is only between Head and Tail, hence data in queue = D2, D3, D4
Tail

2. The head and the tail pointer will get reinitialised to **0** every time they reach the end of the queue.


Tail gets reinitialised to 0 after location 8, same will happen to the Head
Tail = 0
Head = 1
D8
D7
D2
D6
D3
D5
D4

3. Also, the head and the tail pointers can cross each other. In other words, head pointer can be greater than the tail. Sounds odd? This will happen when we dequeue the queue a couple of times and the tail pointer gets reinitialised upon reaching the end of the queue.



In such a situation the value of the Head pointer will be greater than the Tail pointer

Going Round and Round

Another very important point is keeping the value of the tail and the head pointer within the maximum queue size.

In the diagrams above the queue has a size of 8, hence, the value of tail and head pointers will always be between 0 and 7.

This can be controlled either by checking everytime whether tail or head have reached the maxSize and then setting the value 0 or, we have a better way, which is, for a value x if we divide it by 8, the remainder will never be greater than 8, it will always be between 0 and 0, which is exactly what we want.

So the formula to increment the head and tail pointers to make them **go round and round** over and again will be, head = (head+1) % maxSize or tail = (tail+1) % maxSize

Application of Circular Queue

Below we have some common real-world examples where circular queues are used:

1. Computer controlled **Traffic Signal System** uses circular queue.
2. CPU scheduling and Memory management.

Implementation of Circular Queue

Below we have the implementation of a circular queue:

1. Initialize the queue, with size of the queue defined (maxSize), and head and tail pointers.
2. enqueue: Check if the number of elements is equal to maxSize - 1:
   - If **Yes**, then return **Queue is full**.
   - If **No**, then add the new data element to the location of tail pointer and increment the tail pointer.
3. dequeue: Check if the number of elements in the queue is zero:
   - If **Yes**, then return **Queue is empty**.
   - If **No**, then increment the head pointer.
4. Finding the size:
   - If, **tail >= head**, size = (tail - head) + 1
   - But if, **head > tail**, then size = maxSize - (head - tail) + 1

```
/* Below program is written in C++ language */


#include<iostream>


using namespace std;


#define SIZE 10


class CircularQueue
{
   int a[SIZE];
   int rear;//same as tail
   int front;//same as head
   public:
CircularQueue()
{
       rear = front =-1;
}

// function to check if queue is full
   bool isFull()
{
if(front ==0&& rear == SIZE -1)
{
returntrue;
}
if(front == rear +1)
{
returntrue;
}
returnfalse;
}

// function to check if queue is empty
   bool isEmpty()
{
if(front ==-1)
{
returntrue;
}
else
{
returnfalse;
}
}

//declaring enqueue, dequeue, display and size functions
    void enqueue(int x);
    int dequeue();
    void display();
    int size();
};

// function enqueue - to add data to queue
void CircularQueue ::enqueue(int x)
{
if(isFull())
{
        cout <<"Queue is full";
}
else
{
if(front ==-1)
{
            front =0;
}
```

```
        rear =(rear +1)% SIZE;// going round and round concept
// inserting the element
        a[rear]= x;
        cout << endl <<"Inserted "<< x << endl;
}
}

// function dequeue - to remove data from queue
int CircularQueue ::dequeue()
{
    int y;

if(isEmpty())
{
        cout <<"Queue is empty"<< endl;
}
else
{
        y = a[front];
if(front == rear)
{
// only one element in queue, reset queue after removal
            front =-1;
            rear =-1;
}
else
{
```

```
            front =(front+1)% SIZE;
}
return(y);
}
}

void CircularQueue ::display()
{
/* Function to display status of Circular Queue */
    int i;
if(isEmpty())
{
        cout << endl <<"Empty Queue"<< endl;
}
else
{
        cout << endl <<"Front -> "<< front;
        cout << endl <<"Elements -> ";
for(i = front; i != rear; i=(i+1)% SIZE)
{
            cout << a[i]<<"\t";
}
        cout << a[i];
        cout << endl <<"Rear -> "<< rear;
}
}
```

```
int CircularQueue ::size()
{
if(rear >= front)
{
return(rear - front)+1;
}
else
{
return(SIZE -(front - rear)+1);
}
}

// the main function
int main()
{
   CircularQueue cq;
   cq.enqueue(10);
   cq.enqueue(100);
   cq.enqueue(1000);

   cout << endl <<"Size of queue: "<< cq.size();

   cout << endl <<"Removed element: "<< cq.dequeue();

   cq.display();

return0;
}
```

```
Inserted 10

Inserted 100

Inserted 1000

Size of queue: 3

Removed element: 10

Front -> 1

Elements -> 100      1000

Rear -> 2
```

Deque

A **deque**, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure. Figure 1 shows a deque of Python data objects.

It is important to note that even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures. It is up to you to make consistent use of the addition and removal operations.

The Deque Abstract Data Type

The deque abstract data type is defined by the following structure and operations. A deque is structured, as described above, as an ordered collection of items where items are added and removed from either end, either front or rear. The deque operations are given below.

- Deque() creates a new deque that is empty. It needs no parameters and returns an empty deque.
- addFront(item) adds a new item to the front of the deque. It needs the item and returns nothing.
- addRear(item) adds a new item to the rear of the deque. It needs the item and returns nothing.
- removeFront() removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
- removeRear() removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
- isEmpty() tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the deque. It needs no parameters and returns an integer.

As an example, if we assume that d is a deque that has been created and is currently empty, then Table {dequeoperations} shows the results of a sequence of deque operations. Note that the contents in front are listed on the right. It is very important to keep track of the front and the rear as you move items in and out of the collection as things can get a bit confusing.

Table 1: Examples of Deque Operations

Deque Operation	Deque Contents	Return Value
d.isEmpty()	[]	True
d.addRear(4)	[4]	
d.addRear('dog')	['dog',4,]	

Table 1: Examples of Deque Operations

Deque Operation	Deque Contents	Return Value
d.addFront('cat')	['dog',4,'cat']	
d.addFront(True)	['dog',4,'cat',True]	
d.size()	['dog',4,'cat',True]	4
d.isEmpty()	['dog',4,'cat',True]	False
d.addRear(8.4)	[8.4,'dog',4,'cat',True]	
d.removeRear()	['dog',4,'cat',True]	8.4
d.removeFront()	['dog',4,'cat']	True

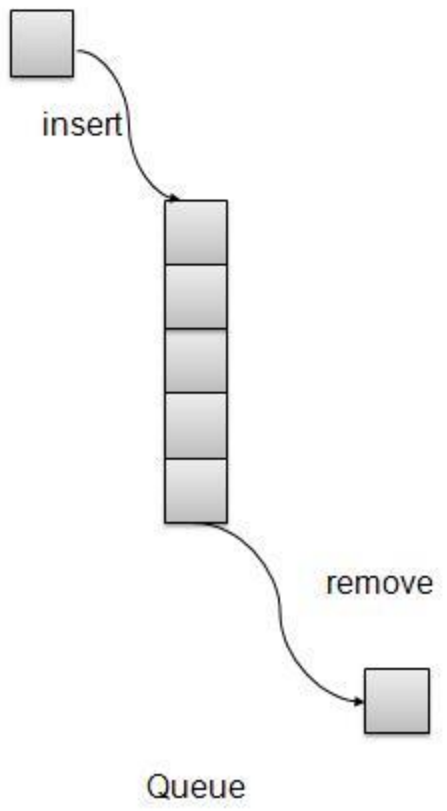
Priority Queue

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are ordered by key value so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

- **insert / enqueue −** add an item to the rear of the queue.
- **remove / dequeue −** remove an item from the front of the queue.

Priority Queue Representation

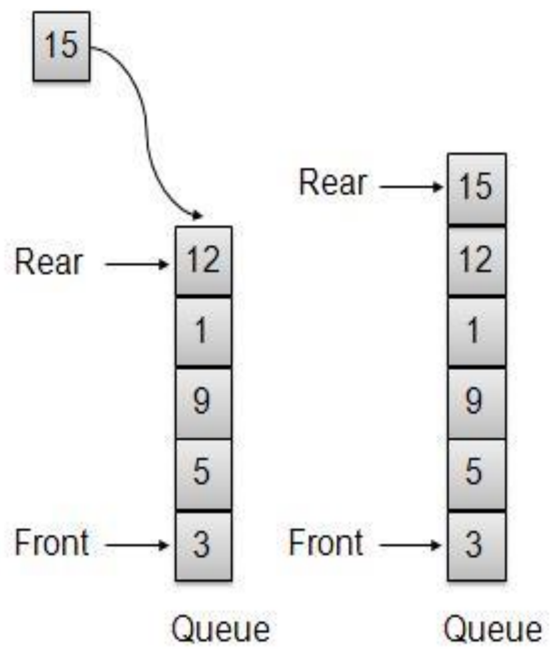


We're going to implement Queue using array in this article. There is few more operations supported by queue which are following.

- **Peek −** get the element at front of the queue.
- **isFull −** check if queue is full.
- **isEmpty −** check if queue is empty.

Insert / Enqueue Operation

Whenever an element is inserted into queue, priority queue inserts the item according to its order. Here we're assuming that data with high value has low priority.

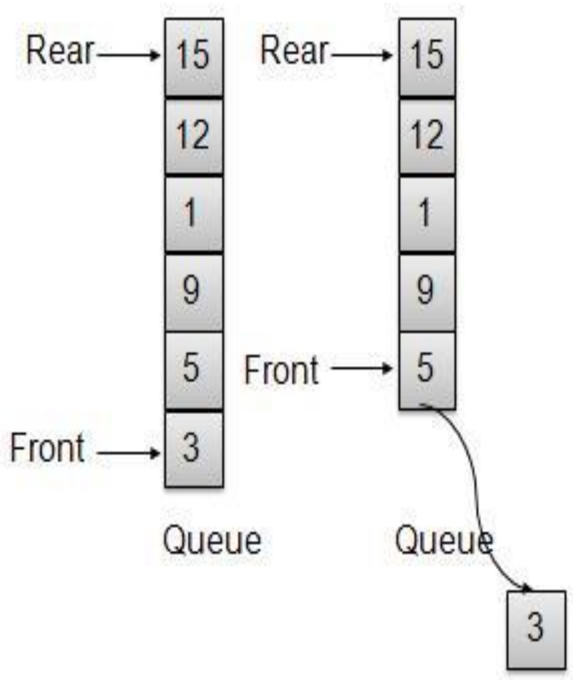


One item inserted at rear end

```
void insert(int data){
int i =0;

if(!isFull()){
// if queue is empty, insert the data

if(itemCount ==0){
        intArray[itemCount++]= data;
}else{
// start from the right end of the queue
for(i = itemCount -1; i >=0; i--){
// if data is larger, shift existing item to right end
if(data > intArray[i]){
              intArray[i+1]= intArray[i];
}else{
break;
}
}
// insert the data
        intArray[i+1]= data;
        itemCount++;
}
}
}
```

Remove / Dequeue Operation

Whenever an element is to be removed from queue, queue get the element using item count. Once element is removed. Item count is reduced by one.



One Item removed from front

```
int removeData(){
return intArray[--itemCount];
}
```

Demo Program

PriorityQueueDemo.c

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<stdbool.h>
#define MAX 6
```

```
int intArray[MAX];
int itemCount =0;

int peek(){
return intArray[itemCount -1];
}

bool isEmpty(){
return itemCount ==0;
}

bool isFull(){
return itemCount == MAX;
}

int size(){
return itemCount;
}

void insert(int data){
int i =0;

if(!isFull()){
// if queue is empty, insert the data
if(itemCount ==0){
        intArray[itemCount++]= data;
}else{
// start from the right end of the queue

for(i = itemCount -1; i >=0; i--){
// if data is larger, shift existing item to right end
if(data > intArray[i]){
                intArray[i+1]= intArray[i];
}else{
break;
}
}

// insert the data
        intArray[i+1]= data;
        itemCount++;
}
}
}

int removeData(){
```

```
return intArray[--itemCount];
}

int main(){
/* insert 5 items */
   insert(3);
   insert(5);
   insert(9);
   insert(1);
   insert(12);

// ------------------
// index : 0  1 2 3 4
// ------------------
// queue : 12 9 5 3 1
   insert(15);

// ---------------------
// index : 0  1 2 3 4  5
// ---------------------
// queue : 15 12 9 5 3 1

if(isFull()){
      printf("Queue is full!\n");
}

// remove one item
int num = removeData();
   printf("Element removed: %d\n",num);

// ---------------------
// index : 0  1  2 3 4
// ---------------------
// queue : 15 12 9 5 3

// insert more items
   insert(16);

// ----------------------
// index :  0  1 2 3 4  5
// ----------------------
// queue : 16 15 12 9 5 3

// As queue is full, elements will not be inserted.
   insert(17);
   insert(18);
```

```
// ----------------------
// index : 0   1  2 3 4 5
// ----------------------
// queue : 16 15 12 9 5 3
   printf("Element at front: %d\n",peek());

   printf("----------------------\n");
   printf("index : 5 4 3 2  1  0\n");
   printf("----------------------\n");
   printf("Queue:  ");

while(!isEmpty()){
int n = removeData();
     printf("%d ",n);
}
}
```

If we compile and run the above program then it would produce following result −

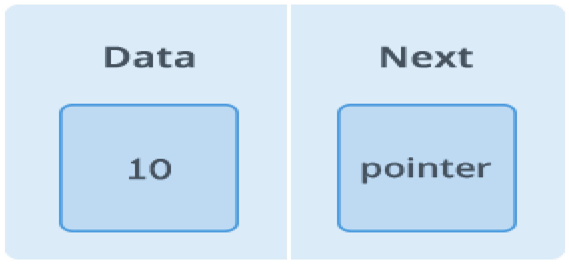
```
Queue is full!
Element removed: 1
Element at front: 3
----------------------
index : 5 4 3 2 1 0
----------------------
Queue: 3 5 9 12 15 16
```

Linked list:

Representation and Implementation of Singly Linked Lists

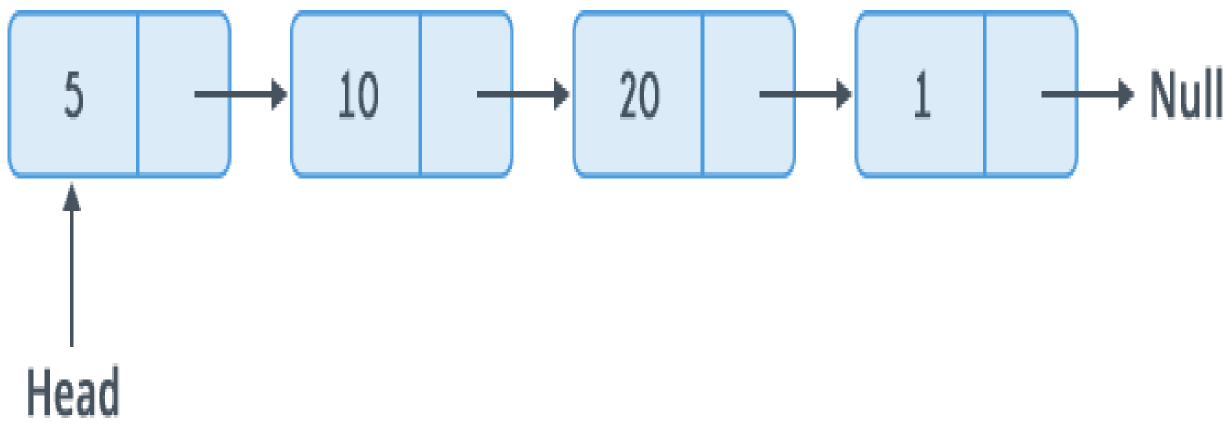
A **linked list** is a way to store a collection of elements. Like an array these can be character or integers. Each element in a linked list is stored in the form of a **node**.

Node:



A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.

Linked List:



A linked list is formed when many such nodes are linked together to form a chain. Each node points to the next node present in the order. The first node is always used as a reference to traverse the list and is called **HEAD**. The last node points to **NULL**.

Declaring a Linked list :

In C language, a linked list can be implemented using structure and pointers .

```
structLinkedList{
int data;
structLinkedList*next;
};
```

The above definition is used to create every node in the list. The **data** field stores the element and the **next** is a pointer to store the address of the next node.

Noticed something unusual with next?

In place of a data type, **struct LinkedList** is written before next. That's because its a **self-referencing pointer**. It means a pointer that points to whatever it is a part of. Here **next** is a part of a node and it will point to the next node.

Creating a Node:

Let's define a data type of struct LinkedListto make code cleaner.

```
typedefstructLinkedList*node;//Define node as pointer of data type struct LinkedList
```

```
node createNode(){
    node temp;// declare a node
    temp =(node)malloc(sizeof(structLinkedList));// allocate memory using malloc()
    temp->next= NULL;// make next point to NULL
return temp;//return the new node
}
```

typedef is used to define a data type in C.

malloc() is used to dynamically allocate a single block of memory in C, it is available in the header file stdlib.h.

sizeof() is used to determine size in bytes of an element in C. Here it is used to determine size of each node and sent as a parameter to malloc.

The above code will create a node with data as value and next pointing to NULL.

Let's see how to **add a node to the linked list**:

```
node addNode(node head,int value){
    node temp,p;// declare two nodes temp and p
    temp = createNode();//createNode will return a new node with data = value and next
pointing to NULL.
    temp->data = value;// add element's value to data part of node
if(head == NULL){
        head = temp;//when linked list is empty
}
else{
        p  = head;//assign head to p
while(p->next!= NULL){
            p = p->next;//traverse the list until p is the last node.The last node always points
to NULL.
}
        p->next= temp;//Point the previous last node to the new node created.
}
return head;
}
```

Here the new node will always be added after the last node. This is known as **inserting a node at the rear end**.

Food for thought

This type of linked list is known as **simple or singly linked list**. A simple linked list can be traversed in only one direction from **head** to the last node.

The last node is checked by the condition :

```
p->next= NULL;
```

Here -> is used to access **next** sub element of node p. **NULL** denotes no node exists after the current node , i.e. its the end of the list.

Traversing the list:

The linked list can be traversed in a while loop by using the **head** node as a starting reference:

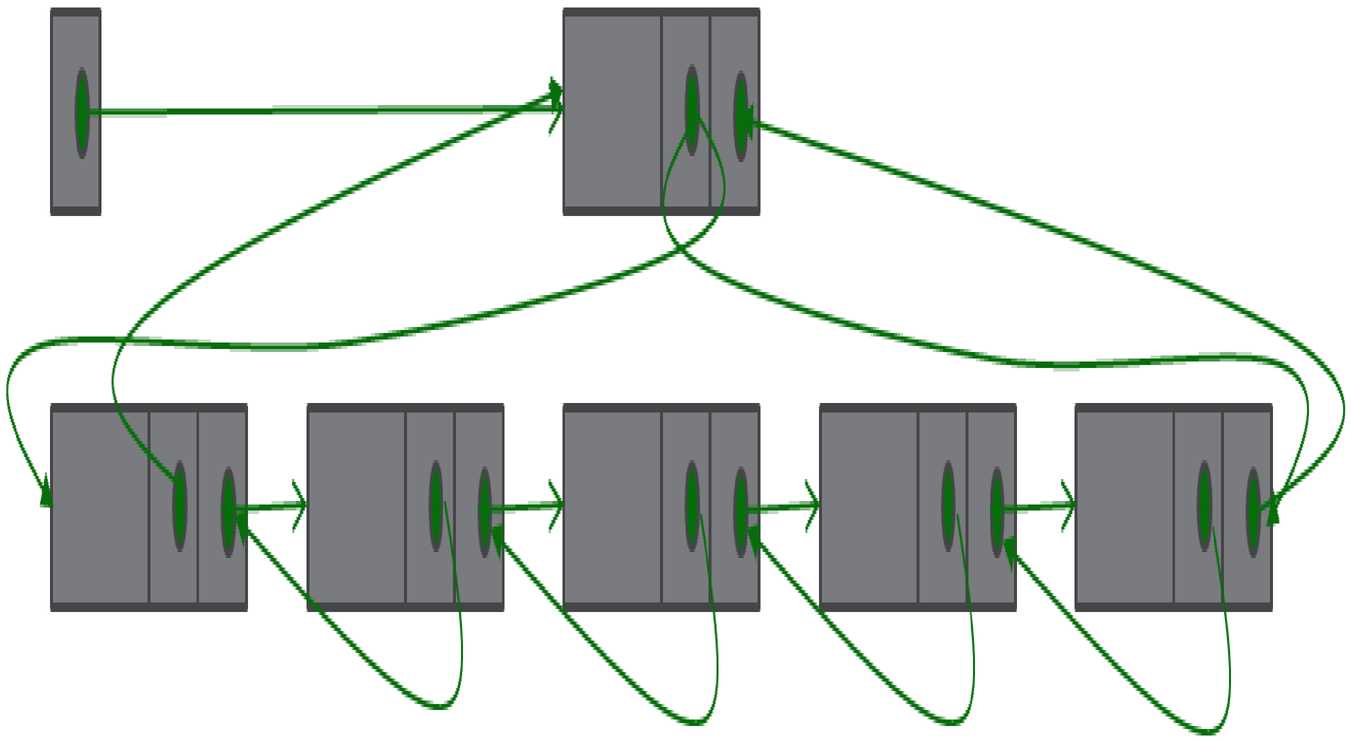
```
node p;
p = head;
while(p != NULL){
    p = p->next;
}
```

Two-way Header List

A two way list is a linear collection of data elements, called nodes, where each node N is divided into three parts:- information field, Forward link- which points to the next node and Backward link-which points to the previous node.

Why Two Way List is important?

The importance of a two way list is that, a two way list and a circular header list may be combined into a two way circular header list as the figure below. The list is circular because the two end nodes point back to the header node.



Circular Two Way List

Traversing and Searching of Linked List

A linear list can be traversed in two ways

- In order traversal
- Reverse order traversal

In order Traversal

To traverse the linear linked list, we walk the list using the pointers, and process each element until we reach the last element.

.cf { font-family: Lucida Console; font-size: 9pt; color: black; background: white; }
.cl { margin: 0px; }
.cb1 { color: green; }
.cb2 { color: blue; }
.cb3 { color: maroon; }

```
void traverseinorder(node *head)



{

while(head!=NULL)

 {

 printf("%dn",head->info);

 head=head->next;

 }

}
```

Reverse Order Traversal

To traverse the linear linked list in reverse order, we walk the list until we reach the last element. The last element is processed first, then the second last and so on and finally the first element of the list.

To implement this we can use either a stack (a last-in-first-out or LIFO data structure) or recursion. Here is the recursive version:

```
void traverseinreverseorder(node *head)

{

if(head->next!=NULL)

 {

 traverseinreverseorder (head->next);

 printf("%dn",head->info);
 }

}
```

Searching an Element

In a linear linked list, only linear searching is possible. This is one of the limitations of the linked list: we cannot directly approach any element other than head.

Depending on whether the list is sorted or unsorted, a suitable searching method can be used.

List is Unsorted

We traverse the list from the beginning, and compare each element of the list with the given element (say “item”) to be searched.

```
node *searchunsortedlist(node *head, int item)

{

while((head!=NULL) &&(head->info!=item))

 head=head->next;

return head;

}
```

List is Sorted

If the list is sorted in ascending order, we traverse the list from the beginning and compare each element of the list with the item to be searched. If a match occurs, the location of the element is returned. If we reach an element that has a value greater than “item”, or we move past the end of the list, we return NULL.

```
node *searchsortedlist(node *head, int item)
```

```
{

while(head != NULL)

 {

if(head->info == item)
return head;
elseif (item < head->info)
return NULL;
&nsp; else

 head = head->next;

 }

return NULL;

}
```

Overflow and Underflow

Overflow

Overflow occurs when we assign such a value to a variable which is more than the maximum permissible value.

Underflow

Underflow occurs when we assign such a value to a variable which is less than the minimum permissible value.

JVM does not throw any exception in case Overflow or underflow occurs, it simply changes the value. Its programmer responsibility to check the possibility of an overflow/underflow condition and act accordingly.

Example (Overflow)

Consider the case of int variable, it is of 32 bit and any value which is more than Integer.MAX_VALUE (2147483647) is rolled over. For example, Integer.MAX_VALUE + 1 returns -2147483648 (Integer.MIN_VALUE).

As int data type is 32 bit in Java, any value that surpasses 32 bits gets rolled over. In numerical terms, it means that after incrementing 1 on Integer.MAX_VALUE

(2147483647), the returned value will be -2147483648. In fact, you don't need to remember these values and the constants Integer.MIN_VALUE and Integer.MAX_VALUE can be used.

Underflow of int

Underflow is the opposite of overflow. While we reach the upper limit in case of overflow, we reach the lower limit in case of underflow. Thus after decrementing 1 from Integer.MIN_VALUE, we reach Integer.MAX_VALUE. Here we have rolled over from the lowest value of int to the maximum value.

For non-integer based data types, the overflow and underflow result in INFINITY and ZERO values.

Insertion and deletion to/from Linked Lists

While (ptr<>NULL) repeat steps 3 to 4. Have another way to solve this solution? A pointer ptr is being used to visit the various nodes in the list. It may be noted in the above algorithm that if the item ‘X’ is found then the search stops. Here we’ll see how to write C program to insert a new node or element into a linked list at all four possible positions: At the front of the list; At the end of the list; Before a specified node; After a specified node; Here is the data structure that represents a node (or element) of the linked list. Find – Finds any node in the list. STEPS:[CHECK IF THE FIRST NODE IS THE DESIRED ONE]. This operation is similar to traveling the list. Head-> [3,1000]-> [43,1001]-> [21,1002] In the example, the number 43 is present at location 1000 and the address is present at in the previous node. A delete operation involves the following two steps: a)search the list for the node which is to be deleted. DELETING A NODE FROM A LINKED LIST. If such a node is found then ptr points to the selected node and back points to immediate previous node in the list. The Linked List is being pointed by a pointer First at the beginning. The number of nodes in the list is also counted during the traverse. 1.If First=NULL then {print “List empty” STOP}; 3.ptr=First; {point ptr to the 1st node}, 6.ptr=NEXT(ptr) [shift ptr to the next node]. We just need to make a few adjustments in the node links. C Program To Implement Queue using Linked List Data Structure. In this algorithm a node with data value equal to ‘VAL’. Let LIST be a pointer to a linked list. [check if the first node is the desired one]. CREATE---In this algorithm a Linked List of nodes is created. Contribute your code (and comments) through Disqus. Write a C Program to implement singly linked list operations. insert_end()]insert_pos() delete_begin() delete_end() delete_pos() These functions are called by the menu-driven main function. The list is pointed by pointer first, the last node of the list points to NULL., indicating the end of the list. Here’s simple Menu Driven Program to to implement singly linked list operations like Creation, Insertion, Deletion, Display, Count, Add Node, Delete Node, Search, Reverse, etc. It may be noted here that the search operation had an upper hand over the insert and delete algorithms for linked lists. In the main function, we take input from the user based on what operation the user wants to do in the program. In a singly linked list, each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list. A delete operation involves the following two steps: a)search the list for the node which is to be deleted. In this algorithm a node with data value equal to ‘VAL’. In this algorithm a node X is inserted in the list after a node with data part equal to ‘VAL’. A algorithm for the deletion of a node from a linked list is given below: DELETE: Let List be a pointer to a linked list. Single linked list operations written using C program. TRAVEL: In this algorithm a linked list, pointed by first, is traversed. Let us assume that a linked list of N number of nodes is to be created. In the above algorithm , step 6 is worth noting i.e ptr=NEXT(ptr). This is how a linked list is represented. first one is data and second field is link that refers to the second node. Inserting a new element into a singly linked list at beginning is quite simple. A pointer ptr travels the list in such a way that each visited node is checked for data part equal to ‘VAL’. If such a node is found then ptr points to the selected node and back point to immediate previous node in the list. Algorithm to delete first node from singly linked list in C. Steps to delete first node from singly linked list. Let List be a pointer to a linked list. Many a times, it is required to traverse whole of a linked list. Insertion in singly linked list at beginning . A pointer ptr is being used to visit the various nodes in the list. Single linked list operations written using C program. We will proceed further by taking the linked list we made in the previous article. We have discussed Linked List Introduction and Linked List Insertion in previous posts on a singly linked list. The node X is inserted before the selected node. b)delete the node. While traversing the data part of each vivited node is compared with an item ‘x’. To delete a node from the linked list, we need to do the following steps. Next: Write a program in C to insert a new node at the middle of Singly Linked List. The program implemented insert, delete, merge, print and quit menu operations. Previous: Write a program in C to insert a new node at the beginning of a Singly Linked List. A delete operation involves the following two steps: a)search the list for the node which is to be deleted. How to delete first node from singly linked list in C language. 2.ptr=First;   [point ptr to the 1st node], 5.ptr=NEXT (ptr); [shift ptr to the next node]. Simple Singly Linked List C Programs Using functions,C Example Programs,Insert,Delete,Display,Count,functions,Singly Linked List Using functions,Singly Linked List Program in C, Data Structures and Algorithm Linked List Programs Using functions in c … If the item is found then the search stops otherwise the process continues til the end of the list(i.e NULL) is encountered. It may be noted in the above algorithm that in step 3 a function exit() has been used. Linked lists in C (Singly linked list) Linked list traversal using while loop and recursion; Concatenating two linked lists in C; Make sure that you are familiar with the concepts explained in the article(s) mentioned above before proceeding further. This step means that the pointer Ptr should be shifted to the node which is being pointed by NEXT(ptr); Search is an operation in which an item is searched in a linked list. in C Programming Language. An algorithm for search operation is given below: In this algorithm a linked list, pointed by first, is traversed. A variable I is being used as a counter to count the number of nodes in the created list. Various linked list operations: Traverse, Insert and Deletion. Rekha Setia is a passionate blogger of Extra Computer Notes. In this algorithm a node with data value equal to 'VAL'. A algorithm for the deletion of a node from a linked list is given below: Let List be a pointer to a linked list. The traverse stops when a NULL is encountered. Learn How To Implement Queue using Linked List in C Programming Language. if you have any ideas or any request me @ Google+, Creation,Insertion ,Deletion algorithms of a Linked List. Two pointers ptr and back travel the list in such a way that each visited node is checked for data equal to 'VAL'. Write a C program to create a singly linked list of n nodes and delete the first node or beginning node of the linked list. Insertion in Singly linked list Singly linked list has two field. There are the following steps which need to be followed in order to inser a new node in the list at beginning. A algorithm for the deletion of a node from a linked list is given below: DELETE: Let List be a pointer to a linked list. The program is given below that will perform insertion, deletion and display a singly linked list. Basic operations of a singly-linked list are: Insert – Inserts a new element at the end of the list. 1.first=new node;{create the 1st node of the list pointed by first}; 4.Far a First;   [point Far to the First], 10.Far=X;[shift the pointer to the last node of the list]. Deleted from the list. In this tutorial, you will learn different operations on a linked list.

Insertion and deletion Algorithms

A queue is an abstract data structure that contains a collection of elements. Queue implements the FIFO mechanism i.e the element that is inserted first is also deleted first.

Queue cane be one linear data structure. But it may create some problem if we implement queue using array. Sometimes by using some consecutive insert and delete operation, the front and rear position will change. In that moment, it will look like the queue has no space to insert elements into it. Even if there are some free spaces, that will not be used due to some logical problems. To overcome this problem, we will use the circular queue data structure.

A circular queue is a type of queue in which the last position is connected to the first position to make a circle.

Algorithm

insert(queue, key) −

```
begin
   if front = 0 and rear = n – 1, or front = rear + 1, then queue is full, and return
   otherwise
   if front = -1, then front = 0 and rear = 0
   else
      if rear = n – 1, then, rear = 0, else rear := rear + 1
   queue[rear] = key
end
```

delete(queue) −

```
begin
   if front = -1 then queue is empty, and return
   otherwise
   item := queue[front]
   if front = rear, then front and rear will be -1
   else
      if front = n – 1, then front := 0 else front := front + 1
end
```

Example

```
#include<iostream>

usingnamespace std;

int cqueue[5];

int front =-1, rear =-1, n=5;

void insertCQ(int val){

   if((front ==0&& rear == n-1)||(front == rear+1)){

      cout<<"Queue Overflow \n";

      return;

   }

   if(front ==-1){

      front =0;
```

```
        rear =0;
    }
    else{
        if(rear == n -1)
            rear =0;
        else
            rear = rear +1;
    }
    cqueue[rear]= val ;
}
void deleteCQ(){
    if(front ==-1){
        cout<<"Queue Underflow\n";
        return;
    }
    cout<<"Element deleted from queue is : "<<cqueue[front]<<endl;
    if(front == rear){
        front =-1;
        rear =-1;
    }
    else{
        if(front == n -1)
            front =0;
        else
            front = front +1;
    }
```

```
}
void displayCQ(){
   int f = front, r = rear;
   if(front ==-1){
      cout<<"Queue is empty"<<endl;
      return;
   }
   cout<<"Queue elements are :\n";
   if(f <= r){
      while(f <= r){
         cout<<cqueue[f]<<" ";
         f++;
      }
   }
   else{
      while(f <= n -1){
         cout<<cqueue[f]<<" ";
         f++;
      }
      f =0;
      while(f <= r){
         cout<<cqueue[f]<<" ";
         f++;
      }
   }
   cout<<endl;
```

```
}
int main(){
   int ch, val;
   cout<<"1)Insert\n";
   cout<<"2)Delete\n";
   cout<<"3)Display\n";
   cout<<"4)Exit\n";
   do{
      cout<<"Enter choice : "<<endl;
      cin>>ch;
      switch(ch){
         case1:
            cout<<"Input for insertion: "<<endl;
            cin>>val;
            insertCQ(val);
         break;
         case2:
            deleteCQ();
         break;
         case3:
            displayCQ();
         break;
         case4:
            cout<<"Exit\n";
         break;
            default: cout<<"Incorrect!\n";
      }
   }while(ch !=4);
      return0;
}
```

Output

```
1)Insert
2)Delete
3)Display
4)Exit
Enter choice :
1
Input for insertion:
10
Enter choice :
1
Input for insertion:
20
Enter choice :
1
Input for insertion:
30
Enter choice :
1
Input for insertion:
40
Enter choice :
1
Input for insertion:
50
Enter choice :
3
Queue elements are :
10 20 30 40 50
Enter choice :
2
Element deleted from queue is : 10
Enter choice :
2
Element deleted from queue is : 20
Enter choice :
3
Queue elements are :
```

```
30 40 50
Enter choice :
4
Exit
```

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.

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.
- **Insert Last −** Adds an element at the end of the list.
- **Delete Last −** Deletes an element from the end of the list.
- **Insert After −** Adds an element after an item of the list.
- **Delete −** Deletes an element from the list using the key.
- **Display forward −** Displays the complete list in a forward manner.
- **Display backward −** Displays the complete list in a backward manner.

Insertion Operation

Following code demonstrates the insertion operation at the beginning of a doubly linked list.

Example

```
//insert link at the first location
void insertFirst(int key,int data){

//create a link
struct node *link =(struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;

if(isEmpty()){
//make it the last link
last= link;
}else{
//update first prev link
      head->prev = link;
}

//point it to old first link
   link->next= head;

//point first to new first link
   head = link;
}
```

Deletion Operation

Following code demonstrates the deletion operation at the beginning of a doubly linked list.

Example

```
//delete first item
struct node* deleteFirst(){

//save reference to first link
struct node *tempLink = head;

//if only one link
if(head->next== NULL){
last= NULL;
}else{
      head->next->prev = NULL;
}

   head = head->next;

//return the deleted link
return tempLink;
}
```

Insertion at the End of an Operation

Following code demonstrates the insertion operation at the last position of a doubly linked list.

Example

```
//insert link at the last location
void insertLast(int key,int data){

//create a link
struct node *link =(struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;

if(isEmpty()){
//make it the last link
last= link;
}else{
```

```
//make link a new last link
last->next= link;

//mark old last node as prev of new link
     link->prev =last;
}

//point last to new last node
last= link;
}
```

Linked List in Array

Both Arrays and Linked List can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.

Key Differences Between Array and Linked List

1. An array is the data structure that contains a collection of similar type data elements whereas the Linked list is considered as a non-primitive data structure contains a collection of unordered linked elements known as nodes.

2. In the array the elements belong to indexes, i.e., if you want to get into the fourth element you have to write the variable name with its index or location within the square bracket while in a linked list though, you have to start from the head and work your way through until you get to the fourth element.

3. Accessing an element in an array is fast, while Linked list takes linear time, so it is quite a bit slower.

4. Operations like insertion and deletion in arrays consume a lot of time. On the other hand, the performance of these operations in Linked lists are fast.

5. Arrays are of fixed size. In contrast, Linked lists are dynamic and flexible and can expand and contract its size.

6. In an array, memory is assigned during compile time while in a Linked list it is allocated during execution or runtime.

7. Elements are stored consecutively in arrays whereas it is stored randomly in Linked lists.

8. The requirement of memory is less due to actual data being stored within the index in the array. As against, there is a need for more memory in Linked Lists due to storage of additional next and previous referencing elements.

9. In addition memory utilization is inefficient in the array. Conversely, memory utilization is efficient in the linked list.

Following are the points in favour of Linked Lists.

(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, and in practical uses, the upper limit is rarely reached.

(2) Inserting a new element in an array of elements is expensive because a room has to be created for the new elements and to create room existing elements have to be shifted.

For example, suppose 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.

So Linked list provides the following two advantages over arrays

1) Dynamic size
2) Ease of insertion/deletion

Linked lists have following drawbacks:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do a binary search with linked lists.

2) Extra memory space for a pointer is required with each element of the list.

3) Arrays have better cache locality that can make a pretty big difference in performance.

Polynomial representation and addition

Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

Example:

Input:

1st number = $5x^2 + 4x^1 + 2x^0$
2nd number = $-5x^1 - 5x^0$

Output:

$5x^2-1x^1-3x^0$

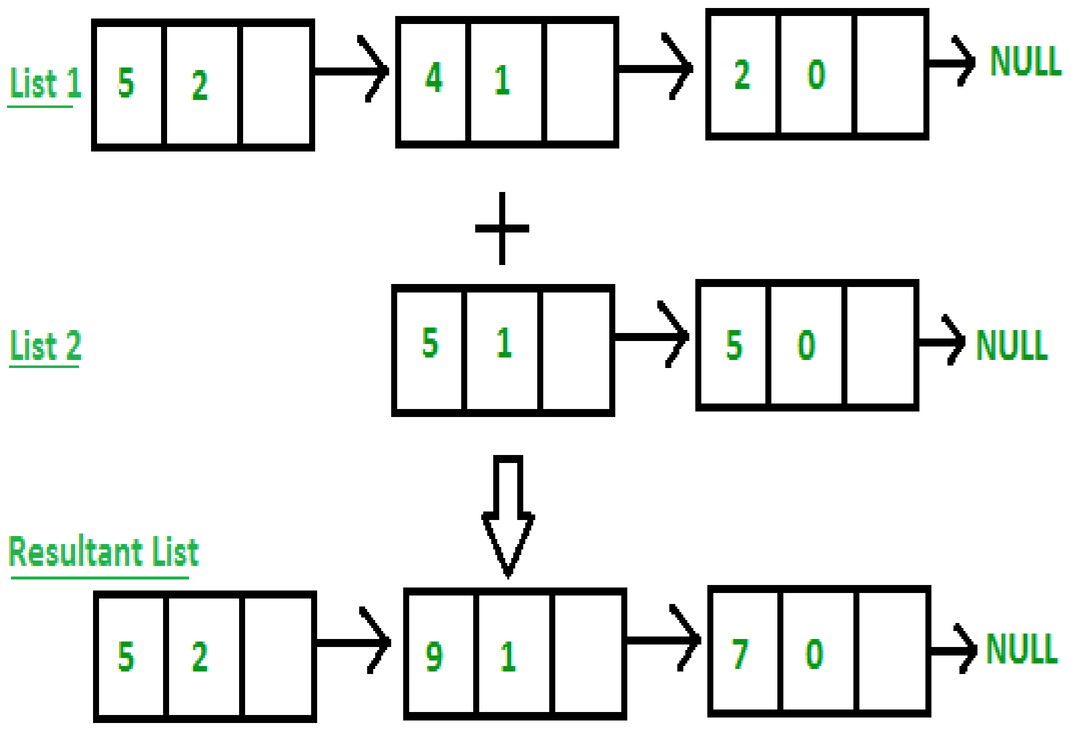
Input:

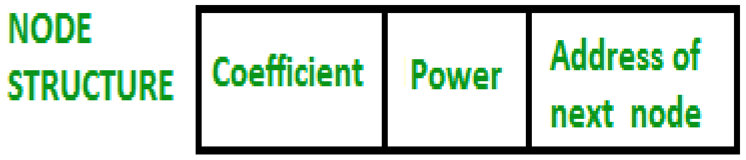
1st number = $5x^3 + 4x^2 + 2x^0$

2nd number = 5x^1 - 5x^0

Output:

$5x^3 + 4x^2 + 5x^1 - 3x^0$





Recommended: Please solve it on "PRACTICE" first, before moving on to the solution.

CPP

filter_none

edit
play_arrow
brightness_4

```
// C++ program for addition of two polynomials

// using Linked Lists

#include <bits/stdc++.h>

usingnamespacestd;


// Node structure containing power and coefficient of

// variable

structNode {

    intcoeff;

    intpow;

    structNode* next;

};


// Function to create new node

voidcreate_node(intx, inty, structNode** temp)

{

    structNode *r, *z;

    z = *temp;

    if(z == NULL) {

        r = (structNode*)malloc(sizeof(structNode));

        r->coeff = x;

        r->pow= y;

        *temp = r;
        r->next = (structNode*)malloc(sizeof(structNode));
        r = r->next;
        r->next = NULL;
    }
    else{
        r->coeff = x;
        r->pow= y;
        r->next = (structNode*)malloc(sizeof(structNode));
        r = r->next;
        r->next = NULL;
    }
}

// Function Adding two polynomial numbers
voidpolyadd(structNode* poly1, structNode* poly2,
             structNode* poly)
{
    while(poly1->next && poly2->next) {
        // If power of 1st polynomial is greater then 2nd,
        // then store 1st as it is and move its pointer
        if(poly1->pow> poly2->pow) {
            poly->pow= poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
        }
```

```
}

    // If power of 2nd polynomial is greater then 1st,
    // then store 2nd as it is and move its pointer
    elseif(poly1->pow< poly2->pow) {
        poly->pow= poly2->pow;
        poly->coeff = poly2->coeff;
        poly2 = poly2->next;
    }

    // If power of both polynomial numbers is same then
    // add their coefficients
    else{
        poly->pow= poly1->pow;
        poly->coeff = poly1->coeff + poly2->coeff;
        poly1 = poly1->next;
        poly2 = poly2->next;
    }

    // Dynamically create new node
    poly->next
        = (structNode*)malloc(sizeof(structNode));
    poly = poly->next;
    poly->next = NULL;
}
while(poly1->next || poly2->next) {
        if(poly1->next) {
            poly->pow= poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
        }
        if(poly2->next) {
            poly->pow= poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }
        poly->next
            = (structNode*)malloc(sizeof(structNode));
        poly = poly->next;
        poly->next = NULL;
    }
}

// Display Linked list
voidshow(structNode* node)
{
    while(node->next != NULL) {
        printf("%dx^%d", node->coeff, node->pow);
        node = node->next;
        if(node->coeff >= 0) {
            if(node->next != NULL)
```

```
                printf("+");
        }
    }
}

// Driver code
intmain()
{
    structNode *poly1 = NULL, *poly2 = NULL, *poly = NULL;

    // Create first list of 5x^2 + 4x^1 + 2x^0
    create_node(5, 2, &poly1);
    create_node(4, 1, &poly1);
    create_node(2, 0, &poly1);

    // Create second list of -5x^1 - 5x^0
    create_node(-5, 1, &poly2);
    create_node(-5, 0, &poly2);

    printf("1st Number: ");
    show(poly1);

    printf("\n2nd Number: ");
    show(poly2);
    poly = (structNode*)malloc(sizeof(structNode));


    // Function add two polynomial numbers

    polyadd(poly1, poly2, poly);


    // Display resultant List

    printf("\nAdded polynomial: ");

    show(poly);


    return0;

}
```

Output

```
1st Number: 5x^2+4x^1+2x^0

2nd Number: -5x^1-5x^0

Added polynomial: 5x^2-1x^1-3x^0
```

Generalized linked list

In this section we will see the generalized lists. The generalized list can be defined as below −

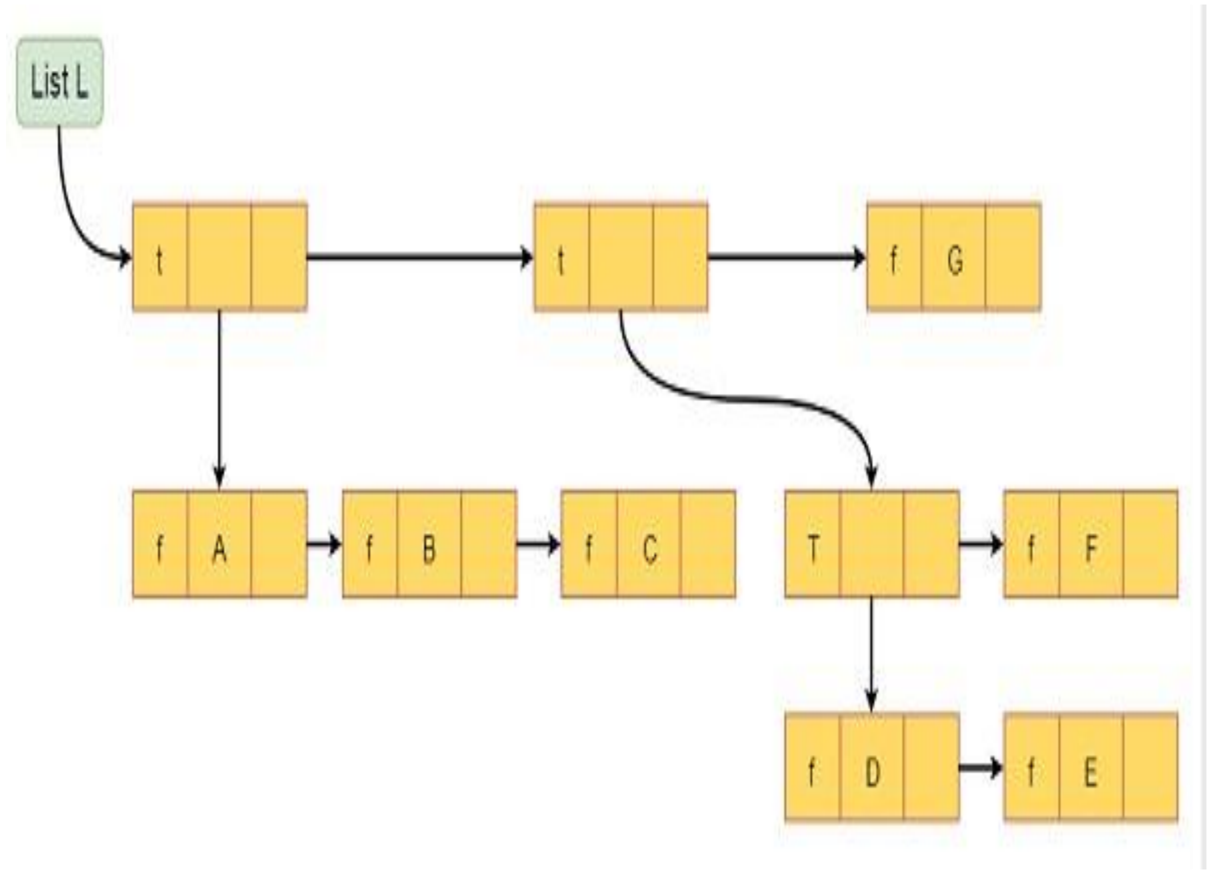
A generalized list L is a finite sequence of n elements (n ≥ 0). The element ei is either an atom (single element) or another generalized list. The elements ei that are not atoms, they will be sub-list of L. Suppose L is ((A, B, C), ((D, E), F), G). Here L has three elements sub-list (A, B, C), sub-list ((D, E), F), and atom G. Again sub-list ((D, E), F) has two elements one sub-list (D, E) and atom F.

In C++, we can define the Generalized list structure like below −

```
class GeneralizedListNode{
  private:
    GeneralizedListNode *next;
    bool tag;
```

```
      union{
         char data;
         GeneralizedListNode *down;
      };
};
```

So if the tag is true, then element represented by the node is a sub-list. The down points to the first node in the sub-list. If tag is false, the element is atom. The next pointer points to the next element in the list. The list will be look like this.



Garbage Collection and Compaction

Garbage collection (GC) is a dynamic approach to automatic memory management and heap allocation that processes and identifies dead memory blocks and reallocates storage for reuse. The primary purpose of garbage collection is to reduce memory leaks.

GC implementation requires three primary approaches, as follows:

- Mark-and-sweep - In process when memory runs out, the GC locates all accessible memory and then reclaims available memory.
- Reference counting - Allocated objects contain a reference count of the referencing number. When the memory count is zero, the object is garbage and is then destroyed. The freed memory returns to the memory heap.
- Copy collection - There are two memory partitions. If the first partition is full, the GC locates all accessible data structures and copies them to the second partition, compacting memory after GC process and allowing continuous free memory.

Some programming languages and platforms with built-in GC (e.g., Java, Lisp, C# and .Net) self-manage memory leaks, allowing for more efficient programming.

Techopedia explains Garbage Collection (GC)

Garbage collection's dynamic approach to automatic heap allocation addresses common and costly errors that often result in real world program defects when undetected.

Because they are difficult to identify and repair, allocation errors are costly. Thus, garbage collection is considered by many to be an essential language feature that makes the programmer’s job easier with lower manual heap allocation management. However, GC is not perfect, and the following drawbacks should be considered:

- When freeing memory, GC consumes computing resources.
- The GC process is unpredictable, resulting in scattered session delays.
- When unused object references are not manually disposed, GC causes logical memory leaks.
- GC does not always know when to process within virtual memory environments of modern desktop computers.
- The GC process interacts poorly with cache and virtual memory systems, resulting in performance-tuning difficulties.

Unit - III

Trees:

Basic terminology

Tree data structure may be defined as-

Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive definition.

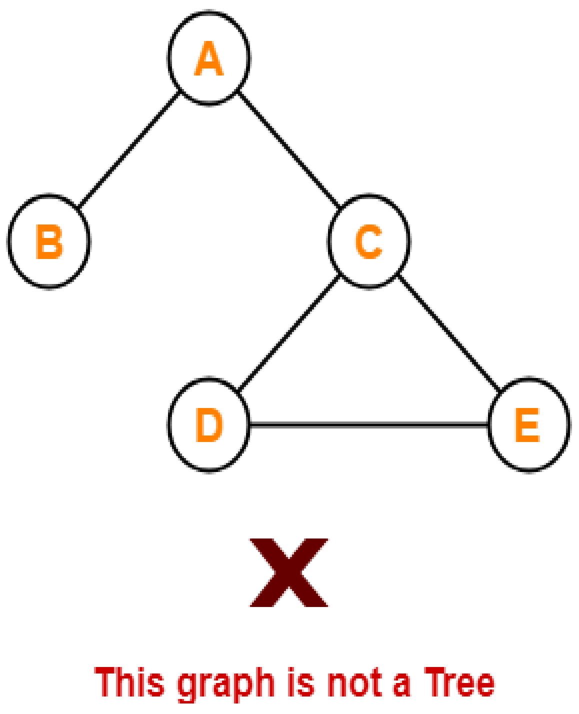
OR

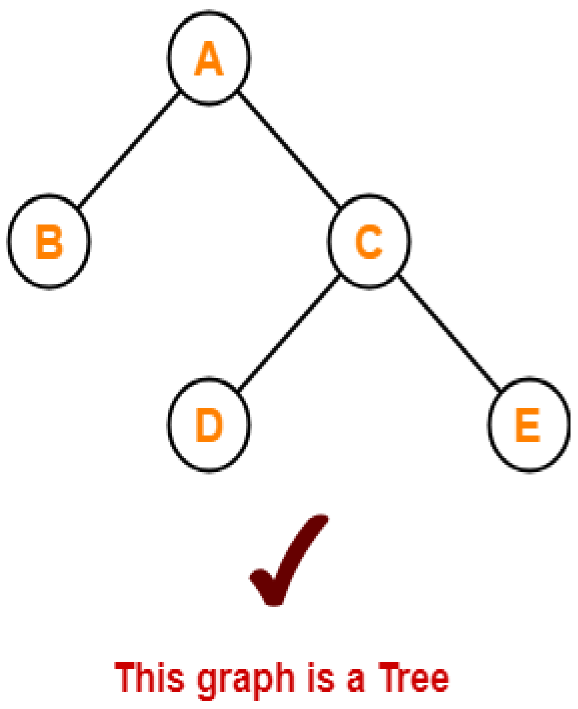
A tree is a connected graph without any circuits.

OR

If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.

Example-





Properties-

The important properties of tree data structure are-

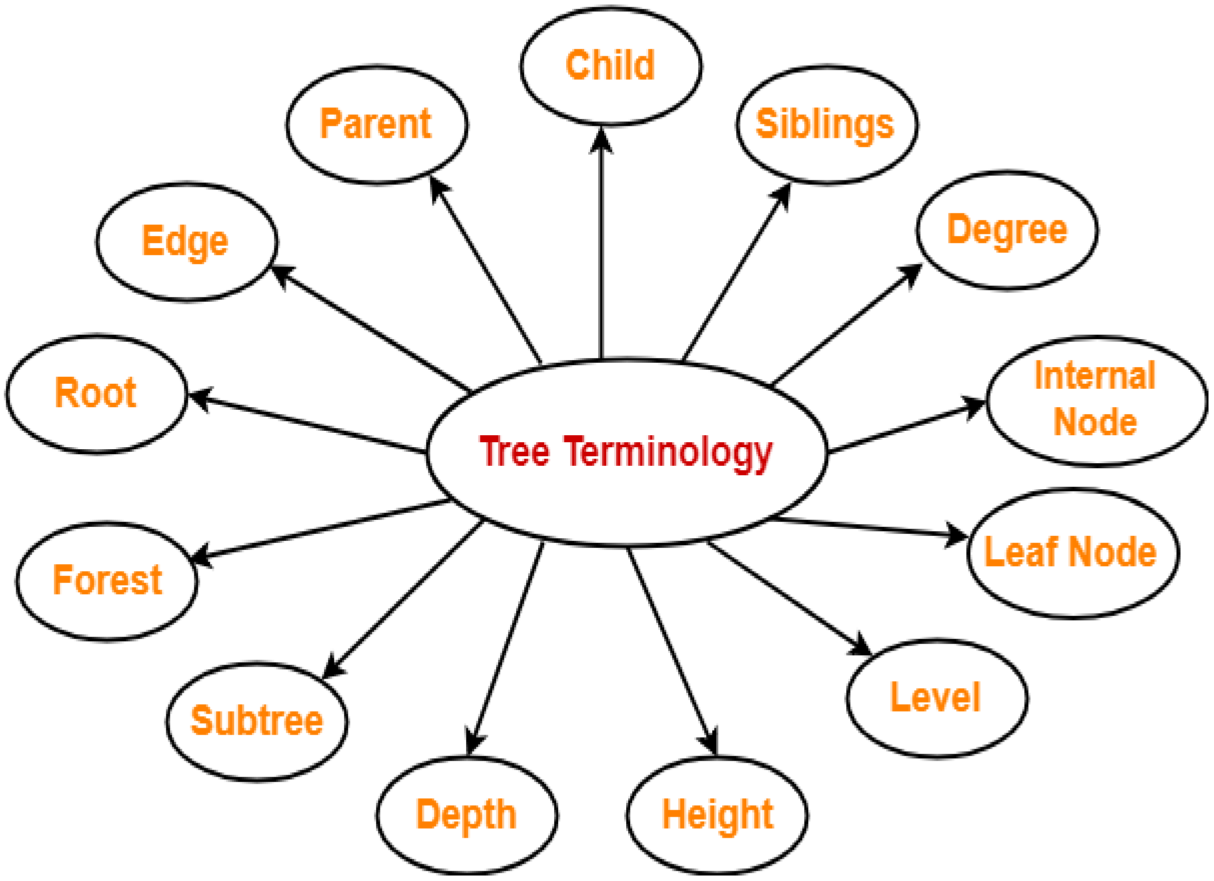
- There is one and only one path between every pair of vertices in a tree.

- A tree with n vertices has exactly (n-1) edges.
- A graph is a tree if and only if it is minimally connected.
- Any connected graph with n vertices and (n-1) edges is a tree.

To gain better understanding about Tree Data Structure,

Tree Terminology-

The important terms related to tree data structure are-

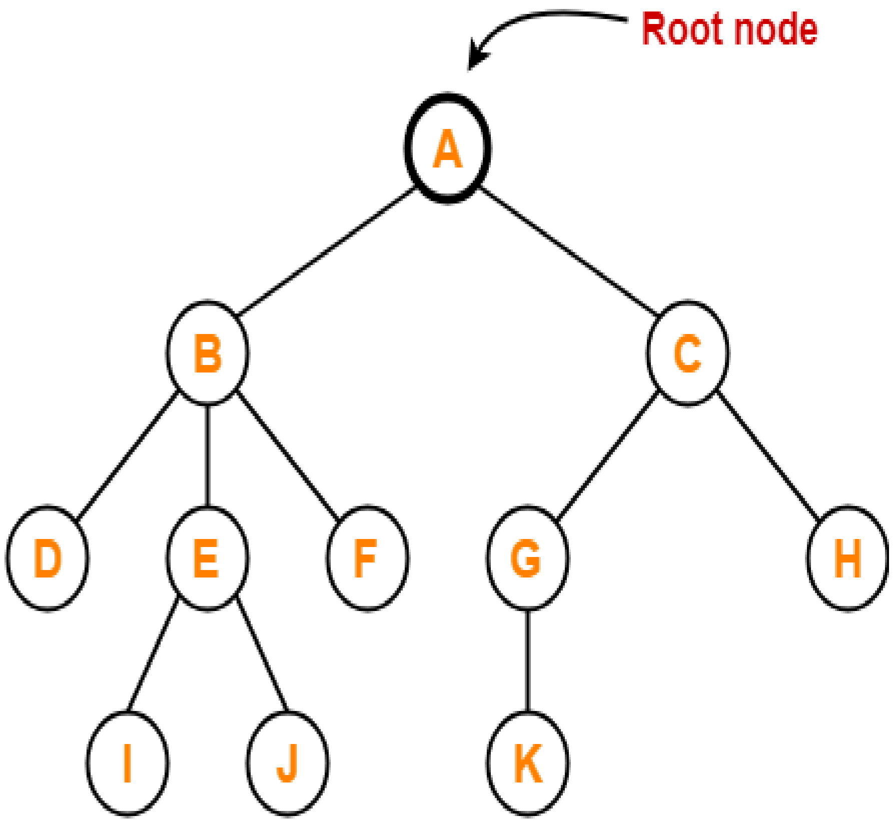


1. Root-

- The first node from where the tree originates is called as a **root node**.
- In any tree, there must be only one root node.

- We can never have multiple root nodes in a tree data structure.

Example-

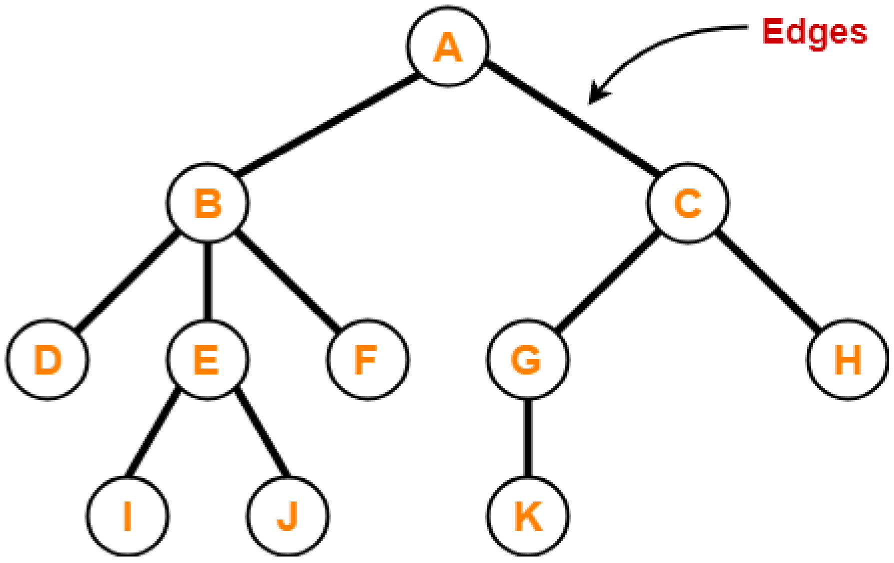


Here, node A is the only root node.

2. Edge-

- The connecting link between any two nodes is called as an **edge**.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.

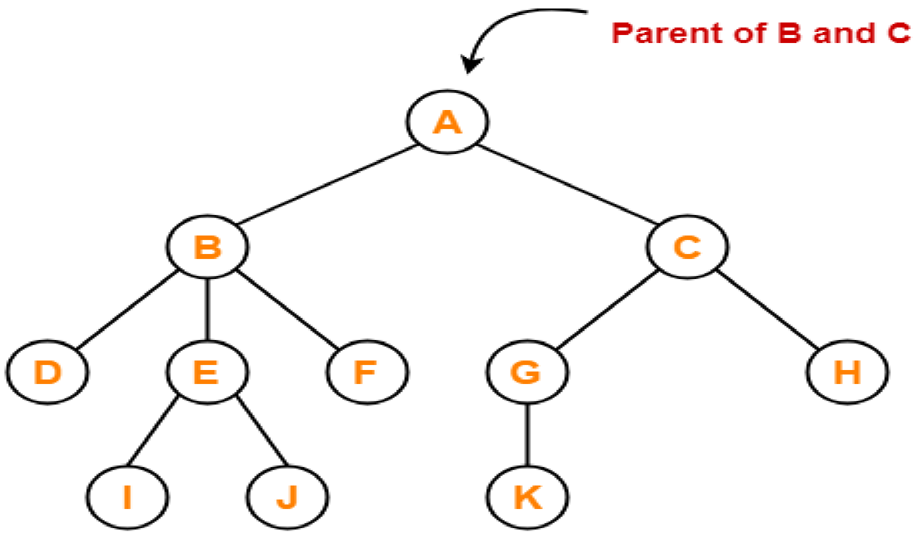
Example-



3. Parent-

- The node which has a branch from it to any other node is called as a **parent node**.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.

Example-



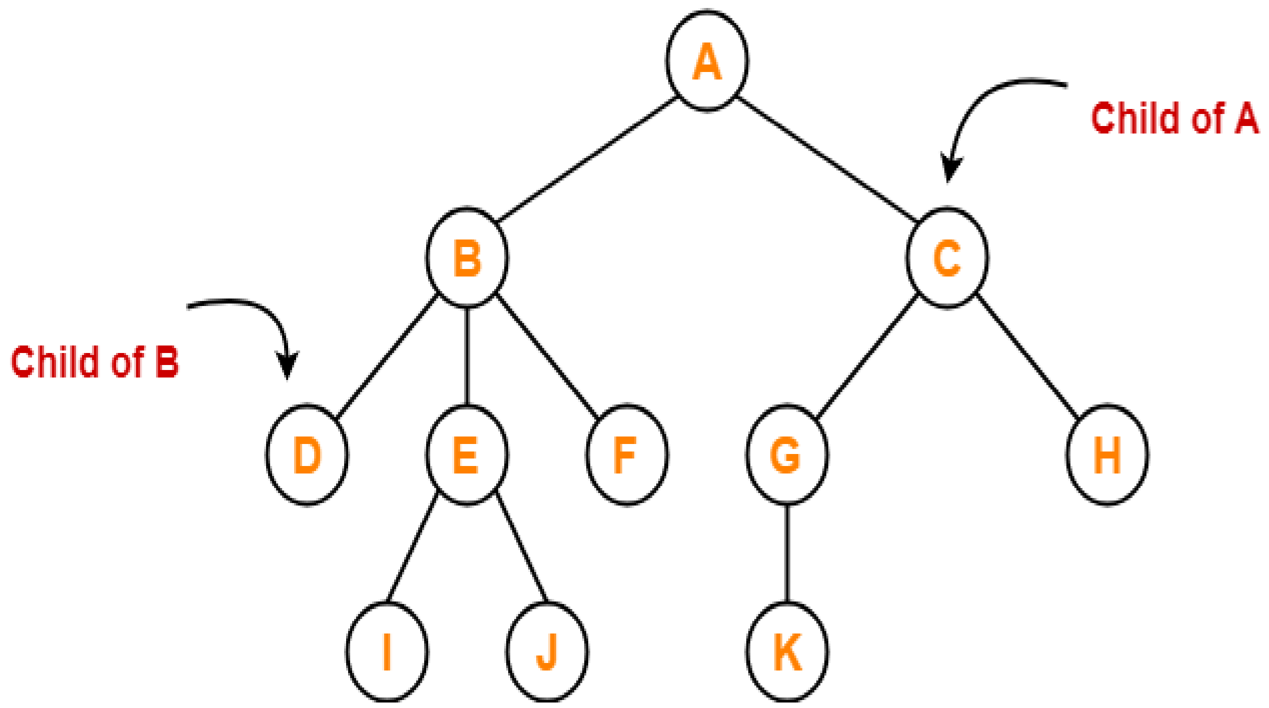
Here,

- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

4. Child-

- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.

Example-



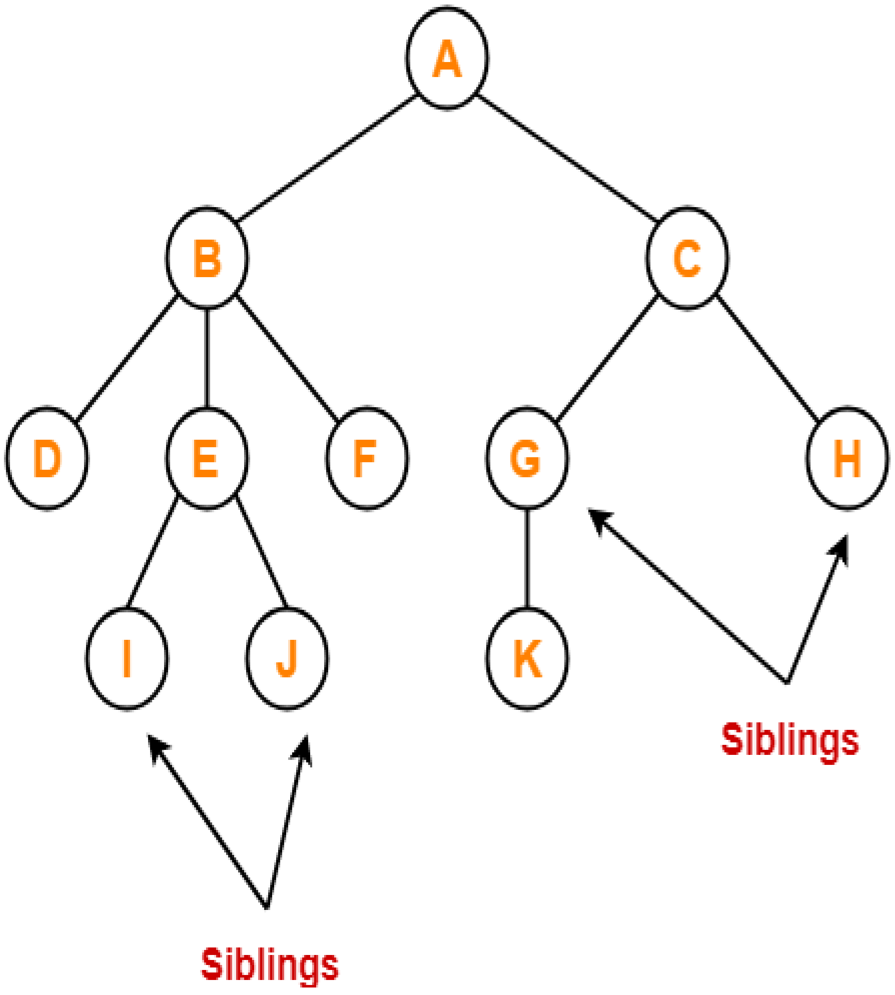
Here,

- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

5. Siblings-

- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.

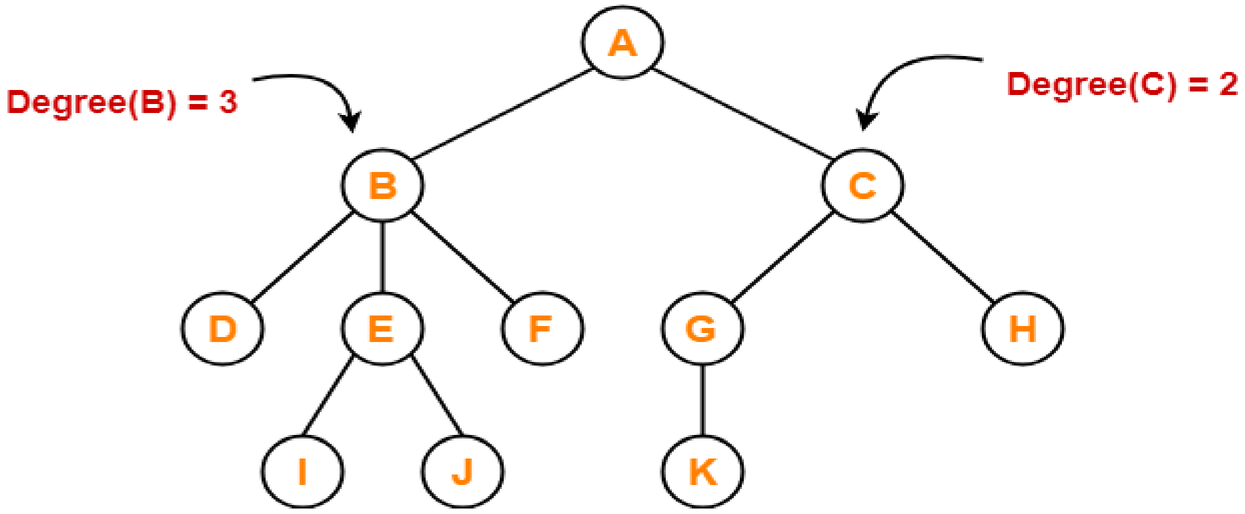
Example-



6. Degree-

- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.

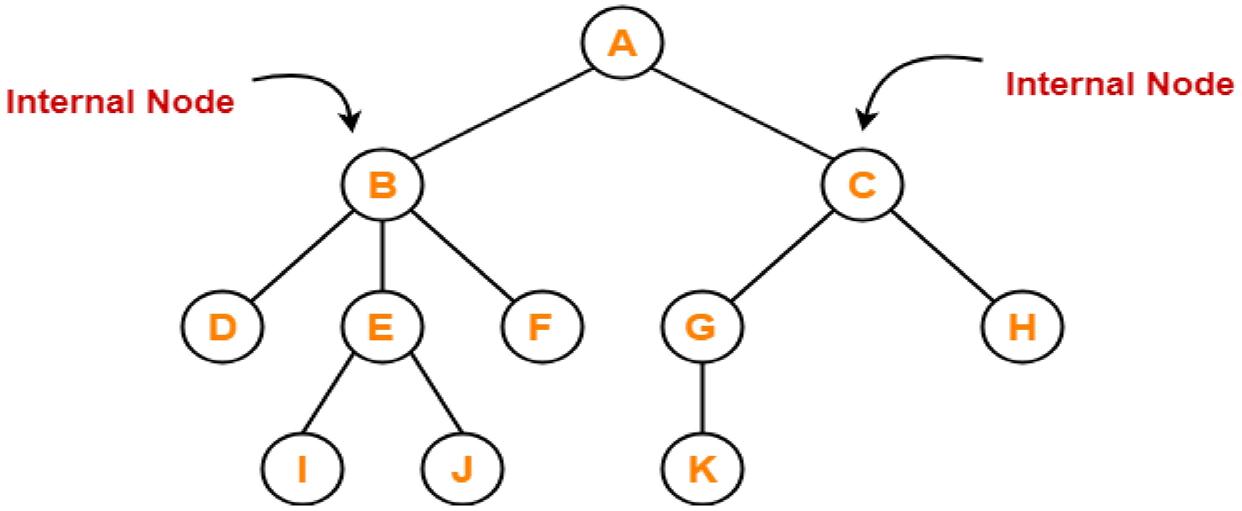
Example-



. Internal Node-

- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.

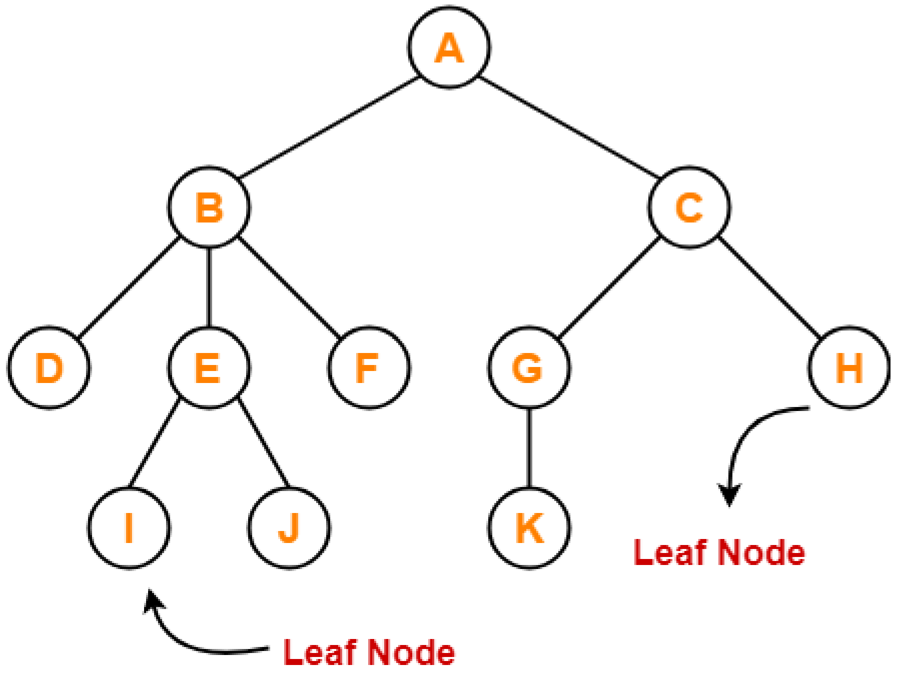
Example-



8. Leaf Node-

- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.

Example-

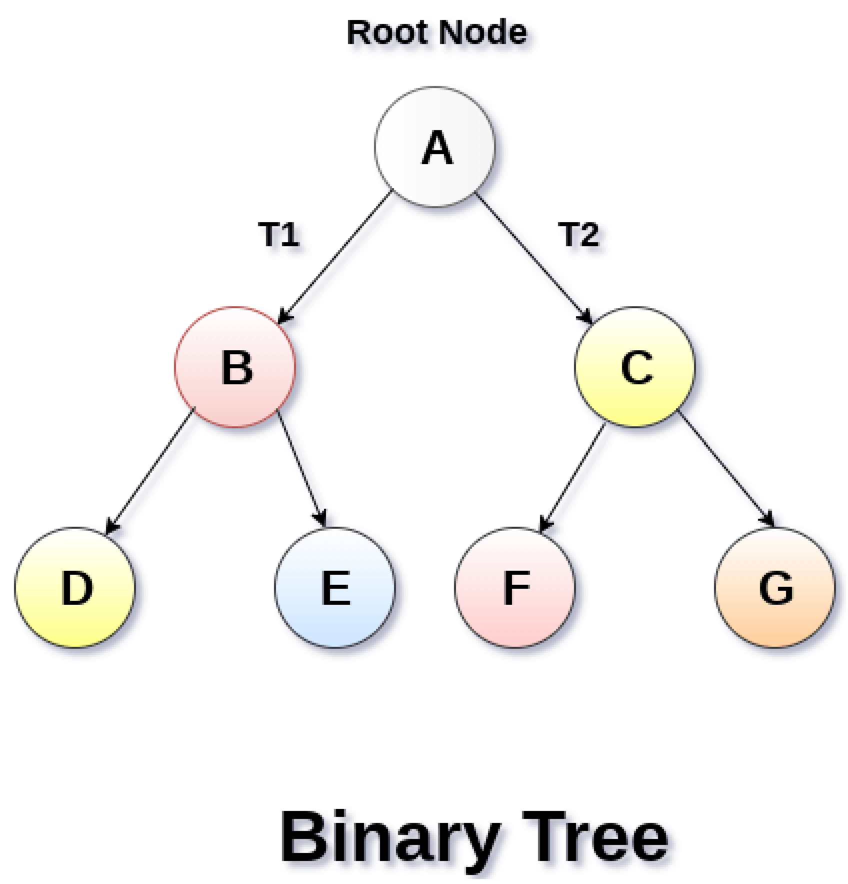


Binary Trees

Binary Tree is a special type of generic tree in which, each node can have at most two children. Binary tree is generally partitioned into three disjoint subsets.

1. Root of the node
2. left sub-tree which is also a binary tree.
3. Right binary sub-tree

A binary Tree is shown in the following image.

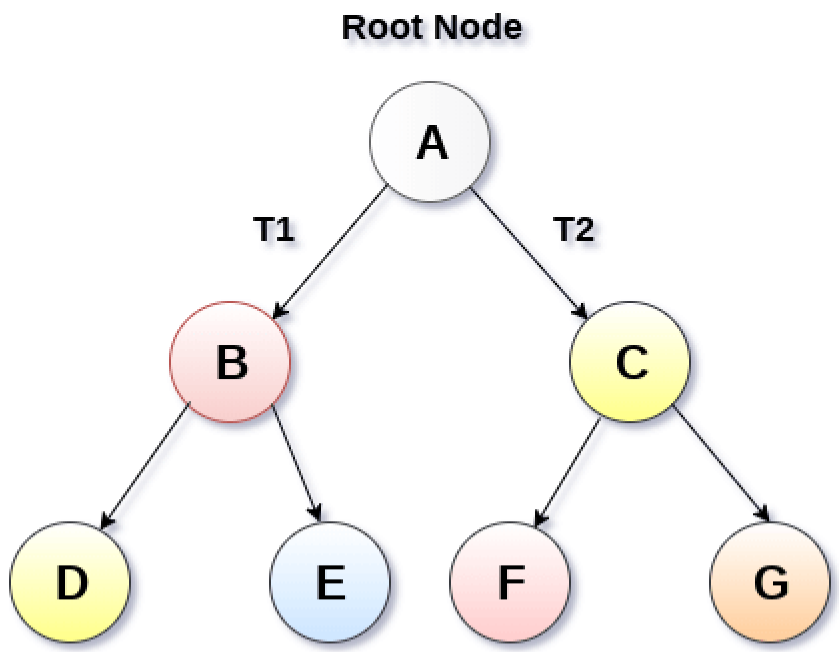


Types of Binary Tree

1. Strictly Binary Tree

In Strictly Binary Tree, every non-leaf node contain non-empty left and right sub-trees. In other words, the degree of every non-leaf node will always be 2. A strictly binary tree with n leaves, will have (2n - 1) nodes.

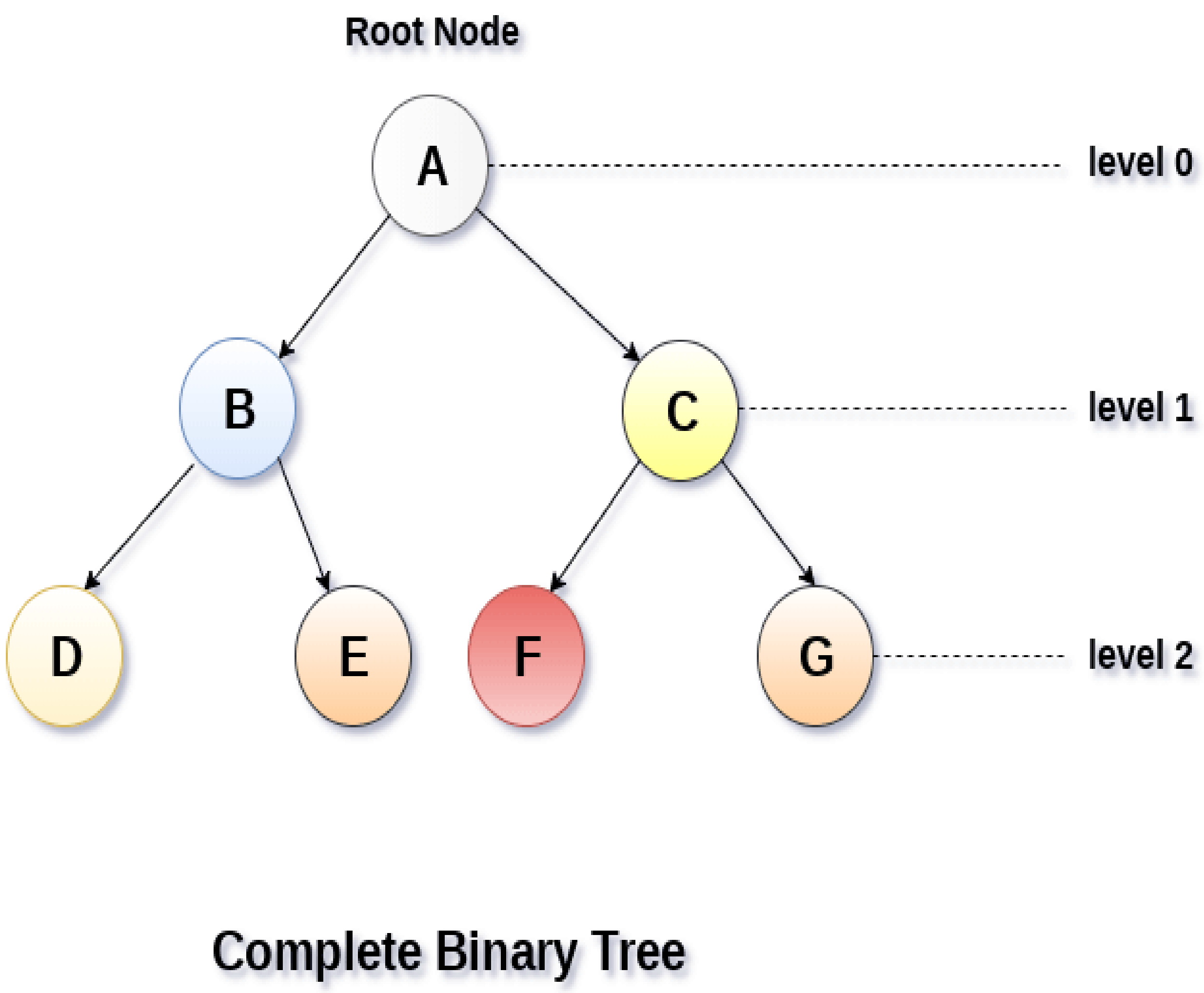
A strictly binary tree is shown in the following figure.



Binary Tree

2. Complete Binary Tree

A Binary Tree is said to be a complete binary tree if all of the leaves are located at the same level d. A complete binary tree is a binary tree that contains exactly 2^l nodes at each level between level 0 and d. The total number of nodes in a complete binary tree with depth d is 2^{d+1}-1 where leaf nodes are 2^{d} while non-leaf nodes are 2^{d}-1.



Binary Tree Traversal

SN	Traversal	Description
1	Pre-order Traversal	Traverse the root first then traverse into the left sub-tree and right sub-tree respectively. This procedure will be applied to each sub-tree of the tree recursively.
2	In-order	Traverse the left sub-tree first, and then traverse the root and the right sub-tree respectively. This procedure will be applied to each

	Traversal	sub-tree of the tree recursively.
3	Post-order Traversal	Traverse the left sub-tree and then traverse the right sub-tree and root respectively. This procedure will be applied to each sub-tree of the tree recursively.

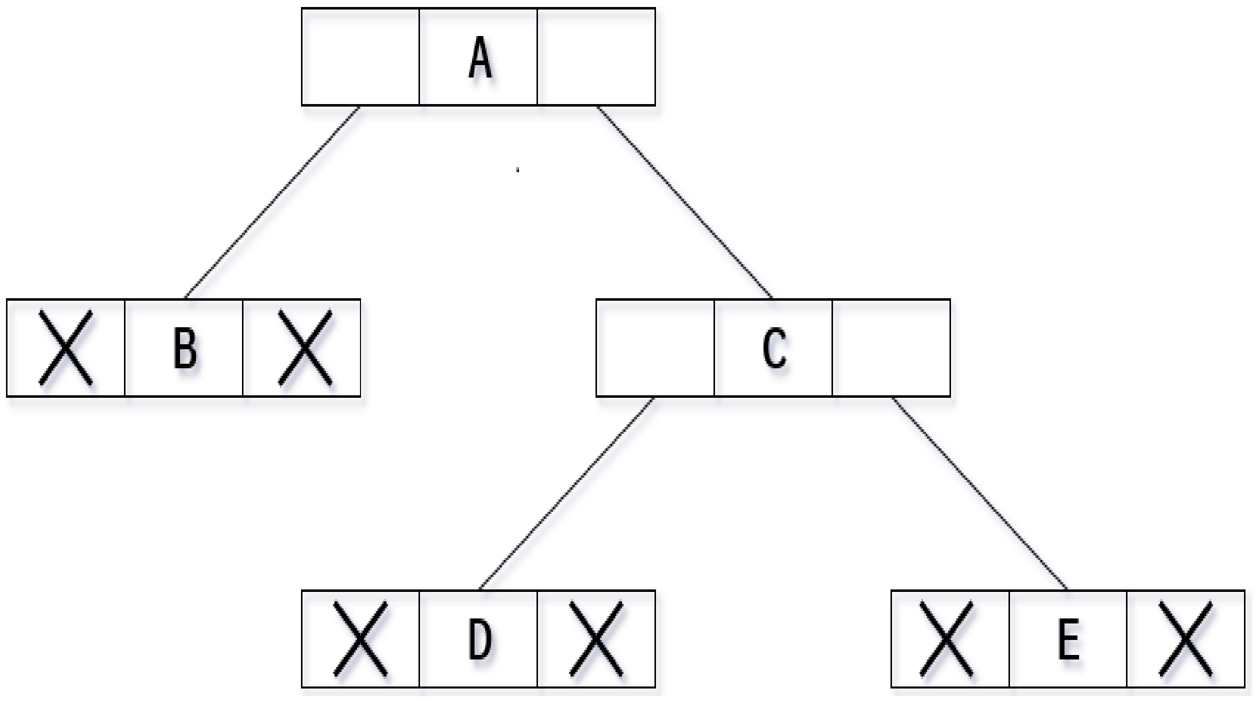
Binary Tree representation

There are two types of representation of a binary tree:

1. Linked Representation

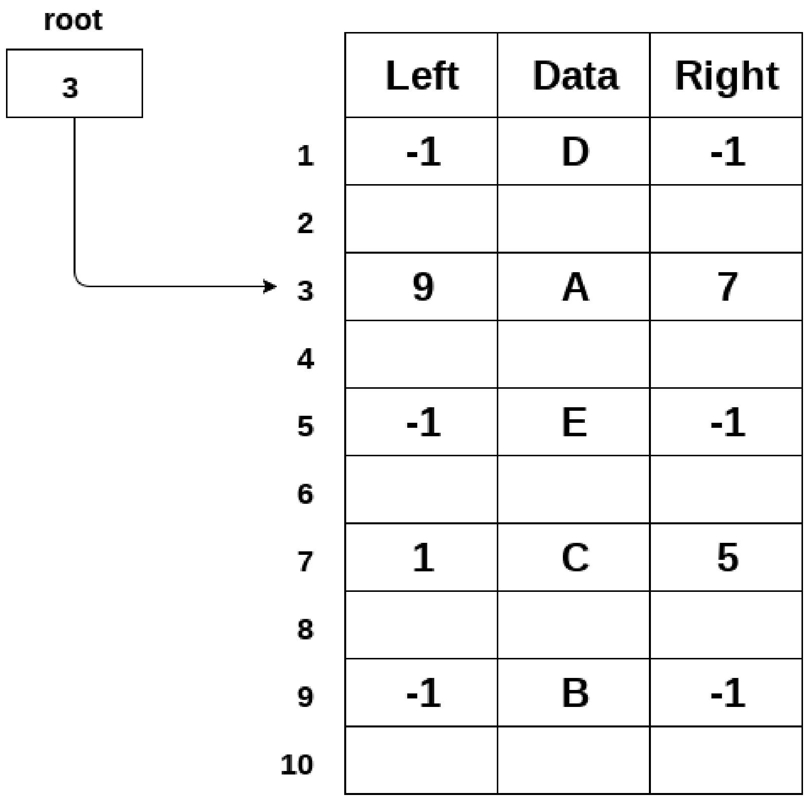
In this representation, the binary tree is stored in the memory, in the form of a linked list where the number of nodes are stored at non-contiguous memory locations and linked together by inheriting parent child relationship like a tree. every node contains three parts : pointer to the left node, data element and pointer to the right node. Each binary tree has a root pointer which points to the root node of the binary tree. In an empty binary tree, the root pointer will point to null.

Consider the binary tree given in the figure below.



In the above figure, a tree is seen as the collection of nodes where each node contains three parts : left pointer, data element and right pointer. Left pointer stores the address of the left child while the right pointer stores the address of the right child. The leaf node contains **null** in its left and right pointers.

The following image shows about how the memory will be allocated for the binary tree by using linked representation. There is a special pointer maintained in the memory which points to the root node of the tree. Every node in the tree contains the address of its left and right child. Leaf node contains null in its left and right pointers.

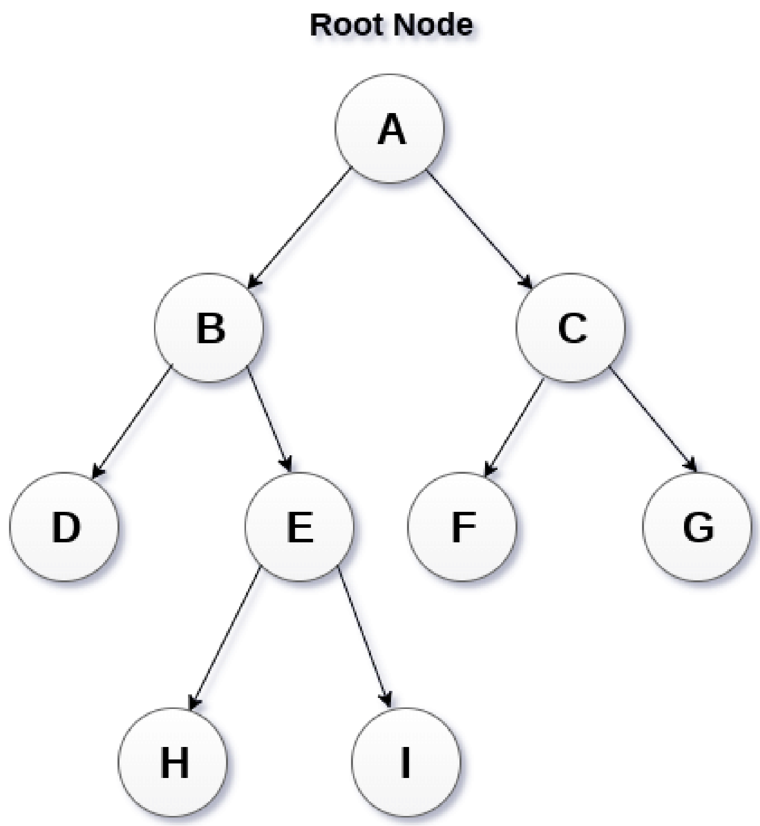


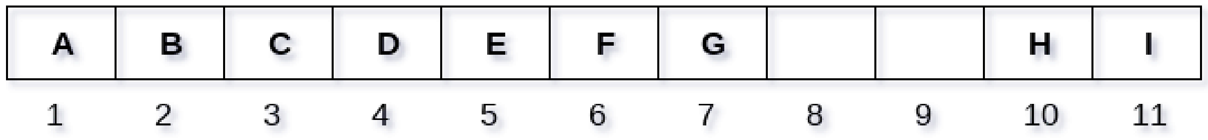
	Left	Data	Right
1	-1	D	-1
2			
3	9	A	7
4			
5	-1	E	-1
6			
7	1	C	5
8			
9	-1	B	-1
10			

Memory Allocation of Binary Tree using linked Representation

2. Sequential Representation

This is the simplest memory allocation technique to store the tree elements but it is an inefficient technique since it requires a lot of space to store the tree elements. A binary tree is shown in the following figure along with its memory allocation.





A	B	C	D	E	F	G			H	I
1	2	3	4	5	6	7	8	9	10	11

Sequential Representation of Binary Tree

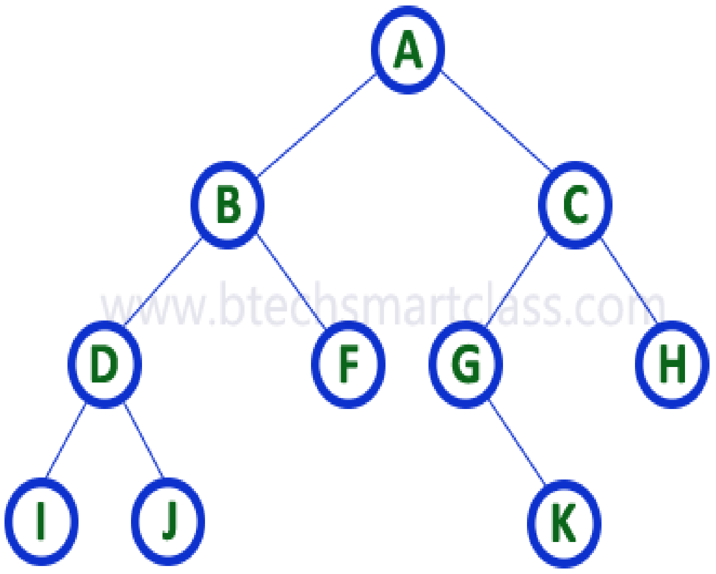
In this representation, an array is used to store the tree elements. Size of the array will be equal to the number of nodes present in the tree. The root node of the tree will be present at the 1st index of the array. If a node is stored at ith index then its left and right children will be stored at 2i and 2i+1 location. If the 1st index of the array i.e. tree[1] is 0, it means that the tree is empty.

Binary tree representation

A binary tree data structure is represented using two methods. Those methods are as follows...

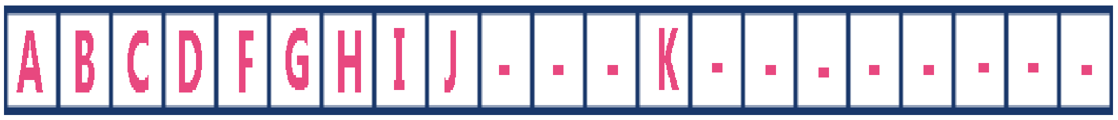
1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...



1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.
Consider the above example of a binary tree and it is represented as follows...

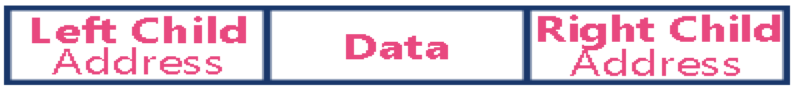


To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of **2n + 1**.

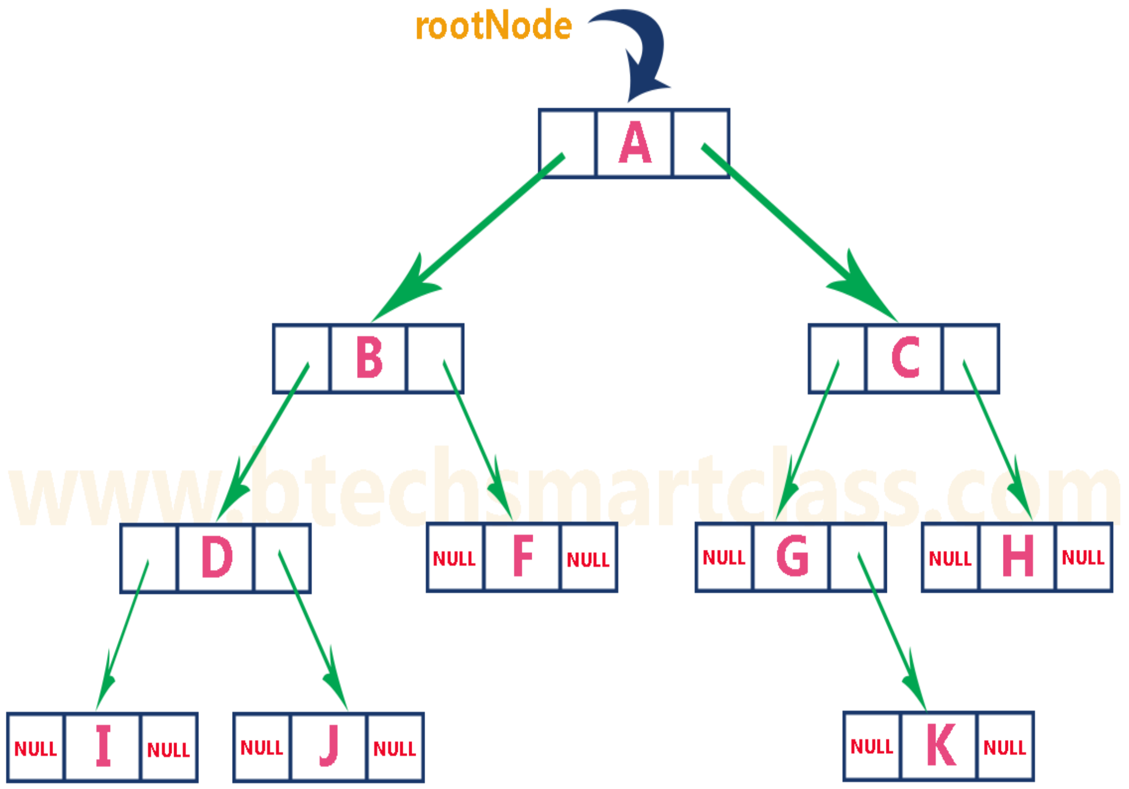
2. Linked List Representation of Binary Tree

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...



The above example of the binary tree represented using Linked list representation is shown as follows...



Algebraic Expressions

Algebra is an interesting and enjoyable branch of mathematics in which numbers, shapes and letters are used to express problems. Whether you are learning algebra in school or you are examining a certain test, you will notice that almost all mathematical problems are represented in words.

Therefore, the need of translating written word problems into algebraic expressions arises when we need to solve them.

Most of the algebraic word problems consist of real-life short stories or cases. Others are simple phrases such as the description of a math problem. Well, in this article we will learn how to write algebraic expressions from simple word problems, then advance to lightly complex word problems.

What is an Algebraic Expression?

Many people interchangeably use algebraic expression and algebraic equations unaware that these terms are totally different.

An algebraic is a mathematical phrase where two side of the phrase are connected by an equal sign (=). For example, $3x + 5 = 20$ is an algebraic equation where 20 represents the right-hand side (RHS) and $3x +5$ represents the left-hand side (LHS) of the equation.

On the other hand, an algebraic expression is a mathematical phrase where variables and constants are combined using the operational (+, -, × & ÷) symbols. An algebraic symbol lacks the equal (=) sign. For example, $10x + 63$ and $5x – 3$ are examples of algebraic expressions.

Let's take a review of the terminologies used in an algebraic expression:

- A variable is a letter whose value is unknown to us. For example, x is our variable in the expression: $10x + 63$.
- The coefficient is a numerical value used together with a variable. For example, 10 is the variable in the expression $10x + 63$.
- A constant is a term which has a definite value. In this case, 63 is the constant in an algebraic expression, $10x + 63$.

There are several types of algebraic expressions but the main type includes:

- Monomial algebraic expression

This is a type of expression having only one term for example, 2x, $5x^2$,3xy, etc.

- Binomial expression

An algebraic expression having two unlike terms, for example, 5y + 8, y+5, $6y^3 + 4$, etc.

- Polynomial expression

This is an algebraic expression with more than one term and with non -zero exponents of variables. An example of a polynomial expression is ab + b c + ca, etc.

Other types of algebraic expressions are:

- Numeric Expression:

A numerical expression only consists of numbers and operators. No variable is added in a numeric expression. Examples of numeric expressions are; 2+4, 5-1, 400+600, etc.

- Variable Expression:

This I an expression which contains variables alongside numbers, for example, 6x + y, 7xy+6, etc.

How to Solve Algebraic Expression?

The purpose of solving an algebraic expression in an equation is to find the unknown variable. When two expressions are equated, they form an equation and therefore, it becomes easier to solve for the unknown terms.

To solve an equation, place the variables on one side and the constants on the other side. The variables can be isolated by applying arithmetic operations like addition, subtraction, multiplication, division, square root, cube root etc.

An algebraic expression is always interchangeable. This implies that, you can rewrite the equation by interchanging the LHS and RHS.

Example 1

Calculate the value of x in the following equation

$5x + 10 = 50$

Solution

Given Equation as $5x + 10 = 50$

- Isolate the variables and the constants;
- You can keep the variable on the LHS and the constants on the RHS.

$5x = 50-10$

- Subtract the constants;

$5x = 40$

Divide both sides by the coefficient of the variable;

$x = 40/5 = 8$

Therefore, the value of x is 8.

Example 2

Find the value of the y when $5y + 45 = 100$

Solution

Isolate the variables from the constants;

$5y = 100 -45$

$5y = 55$

Divide both sides by the coefficient;

$y = 55/5$

$y= 11$

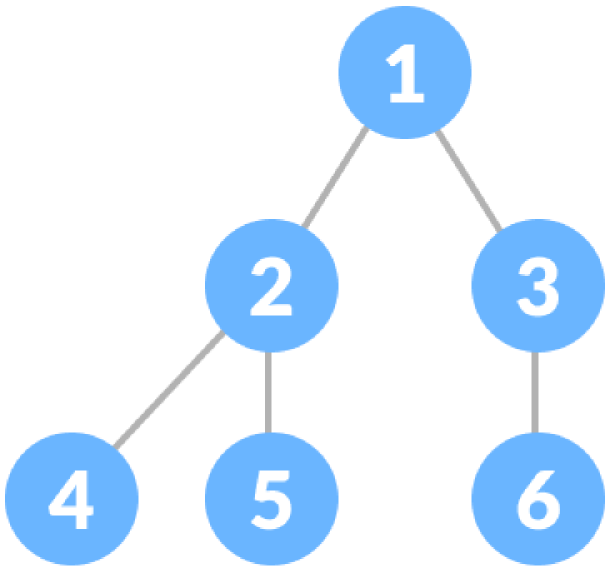
Complete Binary Tree

In this tutorial, you will learn about a complete binary tree and its different types. Also, you will find working examples of a complete binary tree in C, C++, Java and Python.

A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

A complete binary tree is just like a full binary tree, but with two major differences

1. All the leaf elements must lean towards the left.
2. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



Complete Binary Tree

Extended Binary Trees

- 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**.

Array and Linked Representation of Binary trees

Representing a tree with an array

You've seen two approaches to implementing a sequence data structure: either using an array, or using linked nodes. We extended our idea of linked nodes to implement a tree data structure. It turns out we can also use an array to represent a tree.

Here's how we implement a binary tree:

- The root of the tree will be in position 1 of the array (nothing is at position 0). We can define the position of every other node in the tree recursively:
- The left child of a node at position n is at position 2n.
- The right child of a node at position n is at position 2n + 1.
- The parent of a node at position n is at position n/2.

Self-test: ArrayBST representation

For each of the following self-tests, we will give you an instance of an ArrayBST. Match it to one of the four possible binary search trees below.

Tree 1	Tree 2	Tree 3	Tree 4

Self-test 1

Traversing Binary trees

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.

Example Tree

Depth First Traversals:

(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5
Please see this post for Breadth First Traversal.

Inorder Traversal (Practice):

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of In order

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.

Preorder Traversal (Practice):

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to

get prefix expression on of an expression tree. Please
see http://en.wikipedia.org/wiki/Polish_notation to know why prefix expressions are useful.
Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal (Practice):

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Uses of Postorder

Postorder traversal is used to delete the tree. Please see the question for deletion of tree for details. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see http://en.wikipedia.org/wiki/Reverse_Polish_notation to for the usage of postfix expression.
Example: Postorder traversal for the above given figure is 4 5 2 3 1.

- C++
- C
- Python
- Java
- C#

filter_none

edit
play_arrow
brightness_4

```
// C program for different tree traversals

#include <iostream>

usingnamespacestd;



/* A binary tree node has data, pointer to left child

and a pointer to right child */

structNode

{
    intdata;
    structNode* left, *right;
    Node(intdata)
    {
        this->data = data;
        left = right = NULL;
    }
};

/* Given a binary tree, print its nodes according to the
"bottom-up" postorder traversal. */
voidprintPostorder(structNode* node)
{
    if(node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);

    // now deal with the node
    cout << node->data << " ";
}
```

```
/* Given a binary tree, print its nodes in inorder*/
voidprintInorder(structNode* node)
{
    if(node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    cout << node->data << " ";

    /* now recur on right child */
    printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
voidprintPreorder(structNode* node)
{
    if(node == NULL)
        return;

    /* first print data of node */
    cout << node->data << " ";
    /* then recur on left sutree */
    printPreorder(node->left);

    /* now recur on right subtree */
    printPreorder(node->right);
}

/* Driver program to test above functions*/
intmain()
{
    structNode *root = newNode(1);
    root->left             = newNode(2);
    root->right         = newNode(3);
    root->left->left     = newNode(4);
    root->left->right = newNode(5);

    cout << "\nPreorder traversal of binary tree is \n";
    printPreorder(root);

    cout << "\nInorder traversal of binary tree is \n";
    printInorder(root);

    cout << "\nPostorder traversal of binary tree is \n";
    printPostorder(root);
    return0;
}
```

Output:

```
Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Postorder traversal of binary tree is
4 5 2 3 1
```

Threaded Binary trees

Inorder traversal of a Binary tree can either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

There are two types of threaded binary trees.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.

C representation of a Threaded Node
Following is C representation of a single threaded node.
filter_none
edit

play_arrow
brightness_4

```
structNode
{
   intdata;
   Node *left, *right;
   boolrightThread;
}
```

Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

Inorder Taversal using Threads

Following is C code for inorder traversal in a threaded binary tree.
filter_none
edit
play_arrow
brightness_4

```
// Utility function to find leftmost node in a tree rooted with n
structNode* leftMost(structNode *n)
{
   if(n == NULL)
      returnNULL;

   while(n->left != NULL)
      n = n->left;

   returnn;
}

// C code to do inorder traversal in a threaded binary tree
voidinOrder(structNode *root)
{
   structNode *cur = leftmost(root);
   while(cur != NULL)
   {
      printf("%d ", cur->data);

      // If this node is a thread node, then go to
      // inorder successor
      if(cur->rightThread)
         cur = cur->right;
      else// Else go to the leftmost child in right subtree
         cur = leftmost(cur->right);
   }
```

Traversing Threaded Binary trees

A threaded binary tree is a special kind of binary tree (a
tree in which each node has at most two children) that maintains a few extra
variables to allow cheap and fast **in-order traversal** of the tree. We will
explore the general structure of threaded binary trees, as well as
the Swift implementation of a fully functioning
threaded binary tree.

If you don't know what a tree is or what it is for, then
read this first.

In-order traversal

The main motivation behind using a threaded binary tree over a simpler and
smaller standard binary tree is to increase the speed of an in-order traversal
of the tree. An in-order traversal of a binary tree visits the nodes in the
order in which they are stored, which matches the underlying ordering of a
binary search tree. This means most threaded binary
trees are also binary search trees. The idea is to visit all the left children
of a node first, then visit the node itself, and then visit the right children
last.

An in-order traversal of any binary tree generally goes as follows (using Swift
syntax):

```
functraverse(n: Node?) {
if (n ==nil) { return
  } else {
traverse(n.left)
visit(n)
traverse(n.right)
  }
}
```

Where n is a a node in the tree (or nil), each node stores its children as
left and right, and "visiting" a node can mean performing any desired
action on it. We would call this function by passing to it the root of the
tree we wish to traverse.

While simple and understandable, this algorithm uses stack space proportional
to the height of the tree due to its recursive nature. If the tree has **n**
nodes, this usage can range anywhere from **O(log n)** for a fairly balanced
tree, to **O(n)** to a very unbalanced tree.

A threaded binary tree fixes this problem.

Huffman algorithm

Huffman coding is a lossless data compression algorithm. In this algorithm, a variable-length code is assigned to input different characters. The code length is related to how frequently characters are used. Most frequent characters have the smallest codes and longer codes for least frequent characters.

There are mainly two parts. First one to create a Huffman tree, and another one to traverse the tree to find codes.

For an example, consider some strings “YYYZXXYYX”, the frequency of character Y is larger than X and the character Z has the least frequency. So the length of the code for Y is smaller than X, and code for X will be smaller than Z.

Complexity for assigning the code for each character according to their frequency is O(n log n)

Input and Output

```
Input:

A stringwith different characters, say “ACCEBFFFFAAXXBLKE”

Output:

Codefor different characters:

Data: K,Frequency:1,Code:0000

Data: L,Frequency:1,Code:0001

Data: E,Frequency:2,Code:001

Data: F,Frequency:4,Code:01

Data: B,Frequency:2,Code:100

Data: C,Frequency:2,Code:101

Data: X,Frequency:2,Code:110

Data: A,Frequency:3,Code:111
```

Algorithm

huffmanCoding(string)

Input: A string with different characters.

Output: The codes for each individual characters.

```
Begin
   define a node with character, frequency, left and right child of the node for Huffman
tree.
   create a list ‘freq’ to store frequency of each character, initially, all are 0
   for each character c in the string do
      increase the frequency for character ch in freq list.
   done

   for all type of character ch do
      if the frequency of ch is non zero then
         add ch and its frequency as a node of priority queue Q.
   done

   while Q is not empty do
      remove item from Q and assign it to left child of node
      remove item from Q and assign to the right child of node
      traverse the node to find the assigned code
   done
End
```

traverseNode(n: node, code)

Input: The node n of the Huffman tree, and the code assigned from the previous call

Output: Code assigned with each character

```
if a left child of node n ≠φthen

   traverseNode(leftChild(n), code+’0’)     //traverse through the left child

   traverseNode(rightChild(n), code+’1’)    //traverse through the right child

else

   display the character and data of current node.
```

Example

```
#include

#include

#include

usingnamespace std;
```

```
struct node {
  int freq;
  char data;
  const node *child0,*child1;

  node(char d,int f =-1){//assign values in the node
    data = d;
    freq = f;
    child0 = NULL;
    child1 = NULL;
  }

  node(const node *c0,const node *c1){
    data =0;
    freq = c0->freq + c1->freq;
    child0=c0;
    child1=c1;
  }

  booloperator<(const node &a )const{//< operator performs to find priority in queue
    return freq >a.freq;
  }

  void traverse(string code ="")const{
    if(child0!=NULL){
        child0->traverse(code+'0');//add 0 with the code as left child

        child1->traverse(code+'1');//add 1 with the code as right child

    }else{

        cout <<"Data: "<< data<<", Frequency: "< qu;

  int frequency[256];


  for(int i =0; i<256; i++)

    frequency[i]=0;//clear all frequency


  for(int i =0; i1){

    node *c0 =new node(qu.top());//get left child and remove from queue

    qu.pop();

    node *c1 =new node(qu.top());//get right child and remove from queue

    qu.pop();

    qu.push(node(c0, c1));//add freq of two child and add again in the queue

  }
```

```
  cout << "The Huffman Code: "<
```

Output

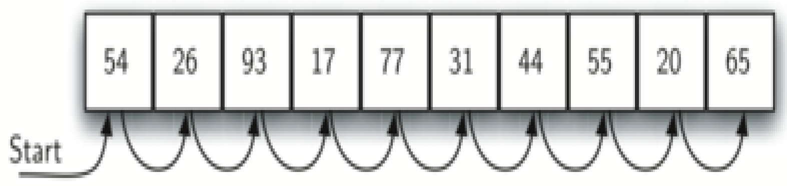
```
The Huffman Code:
Data: K, Frequency: 1, Code: 0000
Data: L, Frequency: 1, Code: 0001
Data: E, Frequency: 2, Code: 001
Data: F, Frequency: 4, Code: 01
Data: B, Frequency: 2, Code: 100
Data: C, Frequency: 2, Code: 101
Data: X, Frequency: 2, Code: 110
Data: A, Frequency: 3, Code: 111
```

Searching and Hashing:

Sequential search

When data items are stored in a collection such as a list, we say that they have a linear or sequential relationship. Each data item is stored in a position relative to the others. In Python lists, these relative positions are the index values of the individual items. Since these index values are ordered, it is possible for us to visit them in sequence. This process gives rise to our first searching technique, the **sequential search**.

The diagram below shows how this search works. Starting at the first item in the list, we simply move from item to item, following the underlying sequential ordering until we either find what we are looking for or run out of items. If we run out of items, we have discovered that the item we were searching for was not present.



Sequential search of a list of integers

The Python implementation for this algorithm is shown below. The function needs the list and the item we are looking for and returns a boolean value as to whether it is present. Remember in practice we would use the Python in operator for this purpose, so you can think of the below algorithm as what we would do if in were not provided for us.

```
def sequential_search(alist, item):
   position = 0

   while position < len(alist):
      if alist[position] == item:
         return True
      position = position + 1

   return False

testlist = [1, 2, 32, 8, 17, 19, 42, 13, 0]

sequential_search(testlist, 3)  # => False
sequential_search(testlist, 13)  # => True
```

Analysis of Sequential Search

To analyze searching algorithms, we need to decide on a basic unit of computation. Recall that this is typically the common step that must be repeated in order to solve the problem. For searching, it makes sense to count the number of comparisons performed. Each comparison may or may not discover the item we are looking for. In addition, we make another assumption here. The list of items is not ordered in any way. The items have been placed randomly into the list. In other words, the probability that the item we are looking for is in any particular position is exactly the same for each position of the list.

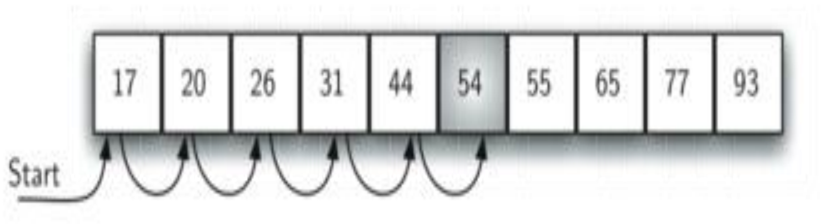
If the item is not in the list, the only way to know it is to compare it against every item present. If there are nn items, then the sequential search requires nn comparisons to discover that the item is not there. In the case where the item is in the list, the analysis is not so straightforward. There are actually three different scenarios that can occur. In the best case we will find the item in the first place we look, at the beginning of the list. We will need only one comparison. In the worst case, we will not discover the item until the very last comparison, the nth comparison.

What about the average case? On average, we will find the item about halfway into the list; that is, we will compare against \frac{n}{2}2n items. Recall, however, that as n gets large, the coefficients, no matter what they are, become insignificant in our approximation, so the complexity of the sequential search, is O(n)O(n):

Case	**Best Case**	**Worst Case**	**Average Case**
item is present	11	nn	\frac{n}{2}2n
item is not present	nn	nn	nn

We assumed earlier that the items in our collection had been randomly placed so that there is no relative order between the items. What would happen to the sequential search if the items were ordered in some way? Would we be able to gain any efficiency in our search technique?

Assume that the list of items was constructed so that the items were in ascending order, from low to high. If the item we are looking for is present in the list, the chance of it being in any one of the n positions is still the same as before. We will still have the same number of comparisons to find the item. However, if the item is not present there is a slight advantage. The diagram below shows this process as the algorithm looks for the item 50. Notice that items are still compared in sequence until 54. At this point, however, we know something extra. Not only is 54 not the item we are looking for, but no other elements beyond 54 can work either since the list is sorted.



Sequential search of an ordered list of integers

In this case, the algorithm does not have to continue looking through all of the items to report that the item was not found. It can stop immediately. The code below shows this variation of the sequential search function.

```
def ordered_sequential_search(alist, item):
   position = 0

   while position < len(alist):
      if alist[position] == item:
         return True

      if alist[position] > item:
         return False

      position = position + 1

   return False

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
ordered_sequential_search(testlist, 3)  # => False
ordered_sequential_search(testlist, 13)  # => True
```

The table below summarizes these results. Note that in the best case we might discover that the item is not in the list by looking at only one item. On average, we will know after looking through only \frac {n}{2}2n items. However, this technique is still O(n)O(n). In summary, a sequential search is improved by ordering the list only in the case where we do not find the item.

Case	**Best Case**	**Worst Case**	**Average Case**
item is present	11	nn	\frac{n}{2}2n
item is not present	nn	nn	\frac{n}{2}2n

binary search

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Binary search algorithm is given below.

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)

Step 1: [INITIALIZE] SET BEG = lower_bound
END = upper_bound, POS = - 1

Step 2: Repeat Steps 3 and 4 while BEG <=END

Step 3: SET MID = (BEG + END)/2

Step 4: IF A[MID] = VAL
SET POS = MID
PRINT POS
Go to Step 6
ELSE IF A[MID] > VAL
SET END = MID - 1
ELSE
SET BEG = MID + 1
[END OF IF]
[END OF LOOP]

Step 5: IF POS = -1
PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
[END OF IF]

Step 6: EXIT
```

Complexity

SN	Performance	Complexity

1	Worst case	O(log n)
2	Best case	O(1)
3	Average Case	O(log n)
4	Worst case space complexity	O(1)

Example

Let us consider an array arr = {1, 5, 7, 8, 13, 19, 20, 23, 29}. Find the location of the item 23 in the array.

In 1st step :

BEG = 0

END = 8ron

MID = 4

a[mid] = a[4] = 13 < 23, therefore

in Second step:

Beg = mid +1 = 5

End = 8

mid = 13/2 = 6

a[mid] = a[6] = 20 < 23, therefore;

in third step:

beg = mid + 1 = 7

End = 8

mid = 15/2 = 7

a[mid] = a[7]

a[7] = 23 = item;

}

therefore, set location = mid;

The location of the item will be 7.

Binary Search Program using Recursion

C program

```
#include<stdio.h>

int binarySearch(int[], int, int, int);

void main ()

{

    int arr[10] = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};

    int item, location=-1;

    printf("Enter the item which you want to search ");

    scanf("%d",&item);

    location = binarySearch(arr, 0, 9, item);

    if(location != -1)

    {

        printf("Item found at location %d",location);

    }

    else

    {

        printf("Item not found");

    }

}
```

```
int binarySearch(int a[], int beg, int end, int item)
{
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
        if(a[mid] == item)
        {
            return mid+1;
        }
        else if(a[mid] < item)
        {
            return binarySearch(a,mid+1,end,item);
        }
        else
        {
            return binarySearch(a,beg,mid-1,item);
        }

    }
    return -1;
}
```

Output:

```
Enter the item which you want to search
19
```

```
Item found at location 2
```

Java

```
import java.util.*;
public class BinarySearch {
public static void main(String[] args) {
    int[] arr = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
    int item, location = -1;
    System.out.println("Enter the item which you want to search");
    Scanner sc = new Scanner(System.in);
    item = sc.nextInt();
    location = binarySearch(arr,0,9,item);
    if(location != -1)
    System.out.println("the location of the item is "+location);
    else
        System.out.println("Item not found");
    }
public static int binarySearch(int[] a, int beg, int end, int item)
{
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
        if(a[mid] == item)
        {
            return mid+1;
        }
        else if(a[mid] < item)
        {
            return binarySearch(a,mid+1,end,item);
        }
        else
        {
            return binarySearch(a,beg,mid-1,item);
        }

    }
    return -1;
}
}
```

Output:

```
Enter the item which you want to search
45
the location of the item is 5
```

C#

```
using System;

public class LinearSearch
{
    public static void Main()
    {
```

```
    int[] arr = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
    int location=-1;
    Console.WriteLine("Enter the item which you want to search ");
    int item = Convert.ToInt32(Console.ReadLine());
    location = binarySearch(arr, 0, 9, item);
    if(location != -1)
    {
        Console.WriteLine("Item found at location "+ location);
    }
    else
    {
        Console.WriteLine("Item not found");
    }
}
public static int binarySearch(int[] a, int beg, int end, int item)
{
    int mid;
    if(end >= beg)
    {
        mid = (beg + end)/2;
        if(a[mid] == item)
        {
            return mid+1;
        }
        else if(a[mid] < item)
        {
            return binarySearch(a,mid+1,end,item);
        }
        else
        {
            return binarySearch(a,beg,mid-1,item);
        }

    }
    return -1;

    }
}
```

Output:

```
Enter the item which you want to search
20
Item found at location 3
```

Python

```
def binarySearch(arr,beg,end,item):
    if end >= beg:
        mid = int((beg+end)/2)
        if arr[mid] == item :
            return mid+1
        elif arr[mid] < item :
            return binarySearch(arr,mid+1,end,item)
        else:
            return binarySearch(arr,beg,mid-1,item)
    return -1



arr=[16, 19, 20, 23, 45, 56, 78, 90, 96, 100];
item = int(input("Enter the item which you want to search ?"))
location = -1;
location = binarySearch(arr,0,9,item);
if location != -1:
    print("Item found at location %d" %(location))
else:
    print("Item not found")
```

Output:

```
Enter the item which you want to search ?
96
Item found at location 9

Enter the item which you want to search ?
101
Item not found
```

Binary Search function using Iteration

```
int binarySearch(int a[], int beg, int end, int item)
{
    int mid;
    while(end >= beg)
    {
        mid = (beg + end)/2;
        if(a[mid] == item)
        {
            return mid+1;
        }
        else if(a[mid] < item)
        {
            beg = mid + 1;
        }
        else
        {
            end = mid - 1;
        }

    }
    return -1;
}
```

comparison and analysis

Describe comparative analysis as comparison analysis. Use comparison analysis to measure the financial relationships between variables over two or
more reporting periods. Businesses use comparative analysis as a way to identify their competitive positions and operating results over a defined period.
Larger organizations may often comprise the resources to
perform financial comparative analysis monthly or quarterly, but it is recommended to perform an annual financial comparison analysis at a minimum.

Financial Comparatives

Financial statements outline the financial comparatives, which are the variables defining operating activities, investing activities and financing activities for a company. Analysts assess company financial statements using percentages, ratios and amounts when making financial comparative analysis. This information is the business intelligence decision makers use for determining future business decisions. A financial comparison analysis may also be performed to determine company profitability and stability. For example, management of a new venture may make a financial comparison analysis periodically to evaluate company performance. Determining losses prematurely and redefining processes in a shorter period will favor compared to unforeseen annual losses.

Comparative Format

The comparative format for comparative analysis in accounting is a side by side view of the financial comparatives in the financial statements. Comparative analysis accounting identifies an organization’s financial performance. For example, income statements identify financial comparables such as company income, expenses, and profit over a period of time. A comparison analysis report identifies where a business meets or exceeds budgets. Potential lenders will also utilize this information to determine a company’s credit limit.

Comparative Analysis in Business

Financial statements play a pivotal role in comparative analysis in business. By analyzing financial comparatives, businesses are able to pinpoint significant trends and project future trends with the identification of considerable or abnormal changes. Business comparative analysis against others in their industry allows a company to evaluate industry results and gauge overall company performance. Different factors such as political events, economics changes, or industry changes influence the changes in trends. Companies may often document significant events in their financial statements that have a major influence on a change in trends.

Hash Table

Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.

Let a hash function H(x) maps the value at the index **x%10** in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.

Hash Functions

A hash function maps keys to small integers (buckets). An ideal hash function maps the keys to the integers in a random-like manner, so that bucket values are evenly distributed even if there are regularities in the input data.

This process can be divided into two steps:

- Map the key to an integer.
- Map the integer to a bucket.

Taking things that really aren't like integers (e.g. complex record structures) and mapping them to integers is icky. We won't discuss this. Instead, we will assume that our keys are either integers, things that can be treated as integers (e.g. characters, pointers) or 1D sequences of such things (lists of integers, strings of characters).

Simple hash functions

The following functions map a single integer key (k) to a small integer bucket value h(k). m is the size of the hash table (number of buckets).

Division method (Cormen) Choose a prime that isn't close to a power of 2. h(k) = k mod m. Works badly for many types of patterns in the input data.

Knuth Variant on Division h(k) = k(k+3) mod m. Supposedly works much better than the raw division method.

Multiplication Method (Cormen). Choose m to be a power of 2. Let A be some random-looking real number. Knuth suggests M = 0.5*(sqrt(5) - 1). Then do the following:

```
s = k*A
x = fractional part of s
h(k) = floor(m*x)
```

This seems to be the method that the theoreticians like.

To do this quickly with integer arithmetic, let w be the number of bits in a word (e.g. 32) and suppose m is 2^p. Then compute:

```
s = floor(A * 2^w)
x = k*s
h(k) = x >> (w-p)      // i.e. right shift x by (w-p) bits
                       // i.e. extract the p most significant
                       // bits from x
```

Hashing sequences of characters

The hash functions in this section take a sequence of integers k=k1,...,kn and produce a small integer bucket value h(k). m is the size of the hash table (number of buckets), which should be a prime number. The sequence of integers might be a list of integers or it might be an array of characters (a string).

The specific tuning of the following algorithms assumes that the integers are all, in fact, character codes. In C++, a character is a char variable which is an 8-bit integer. ASCII uses only 7 of these 8 bits. Of those 7, the common characters (alphabetic and number) use only the low-order 6 bits. And the first of those 6 bits primarily indicates the case of characters, which is relatively insignificant. So the following algorithms concentrate on preserving as much information as possible from the last 5 bits of each number, and make less use of the first 3 bits.

When using the following algorithms, the inputs ki **must** be unsigned integers. Feeding them signed integers may result in odd behavior.

For each of these algorithms, let h be the output value. Set h to 0. Walk down the sequence of integers, adding the integers one by one to h. The algorithms differ in exactly how to combine an integer ki with h. The final return value is h mod m.

CRC variant: Do a 5-bit left circular shift of h. Then XOR in ki. Specifically:

```
highorder = h & 0xf8000000    // extract high-order 5 bits from h
                              // 0xf8000000 is the hexadecimal representation
                              //   for the 32-bit number with the first five
                              //   bits = 1 and the other bits = 0
h = h << 5                    // shift h left by 5 bits
h = h ^ (highorder >> 27)     // move the highorder 5 bits to the low-order
                              //   end and XOR into h
h = h ^ ki                    // XOR h and ki
```

PJW hash (Aho, Sethi, and Ullman pp. 434-438): Left shift h by 4 bits. Add in ki. Move the top 4 bits of h to the bottom. Specifically:

```
// The top 4 bits of h are all zero
h = (h << 4) + ki             // shift h 4 bits left, add in ki
g = h & 0xf0000000            // get the top 4 bits of h
```

```
if (g != 0)                  // if the top 4 bits aren't zero,
   h = h ^ (g >> 24)           //   move them to the low end of h
   h = h ^ g
// The top 4 bits of h are again all zero
```

PJW and the CRC variant both work well and there's not much difference between them. We believe that the CRC variant is probably slightly better because

- It uses all 32 bits. PJW uses only 24 bits. This is probably not a major issue since the final value m will be much smaller than either.

- 5 bits is probably a better shift value than 4. Shifts of 3, 4, and 5 bits are all supposed to work OK.

- Combining values with XOR is probably slightly better than adding them. However, again, the difference is slight.

BUZ hash: Set up a function R that takes 8-bit character values and returns random numbers. This function can be precomputed and stored in an array. Then, to add each character ki to h, do a 1-bit left circular shift of h and then XOR in the random value for ki. That is:

```
highorder = h & 0x80000000    // extract high-order bit from h
h = h << 1                   // shift h left by 1 bit
h = h ^ (highorder >> 31)     // move them to the low-order end and
                            // XOR into h
h = h ^ R[ki]                // XOR h and the random value for ki
```

Rumor has it that you may have to run a second hash function on the output to make it random enough. Experimentally, this function produces good results, but is a bit slower than the CRC variant and PJW.

Collision Resolution Strategies

- Unless you are doing "perfect hashing" you have to have a collision resolution strategy, to deal with collisions in the table.
- The strategy has to permit find, insert, and delete operations that work correctly!
- Collision resolution strategies we will look at are:
    - Linear probing
    - Double hashing
    - Random hashing
    - Separate chaining

Hash Table Implementation

In this tutorial, you will learn what hash table is. Also, you will find working examples of hash table operations in C, C++, Java and Python.

Hash table is a data structure that represents data in the form of **key-value** pairs. Each key is mapped to a value in the hash table. The keys are used for indexing the values/data. A similar approach is applied by an associative array.

Data is represented in a key value pair with the help of keys as shown in the figure below. Each data is associated with a key. The key is an integer that point to the data.

1. Direct Address Table

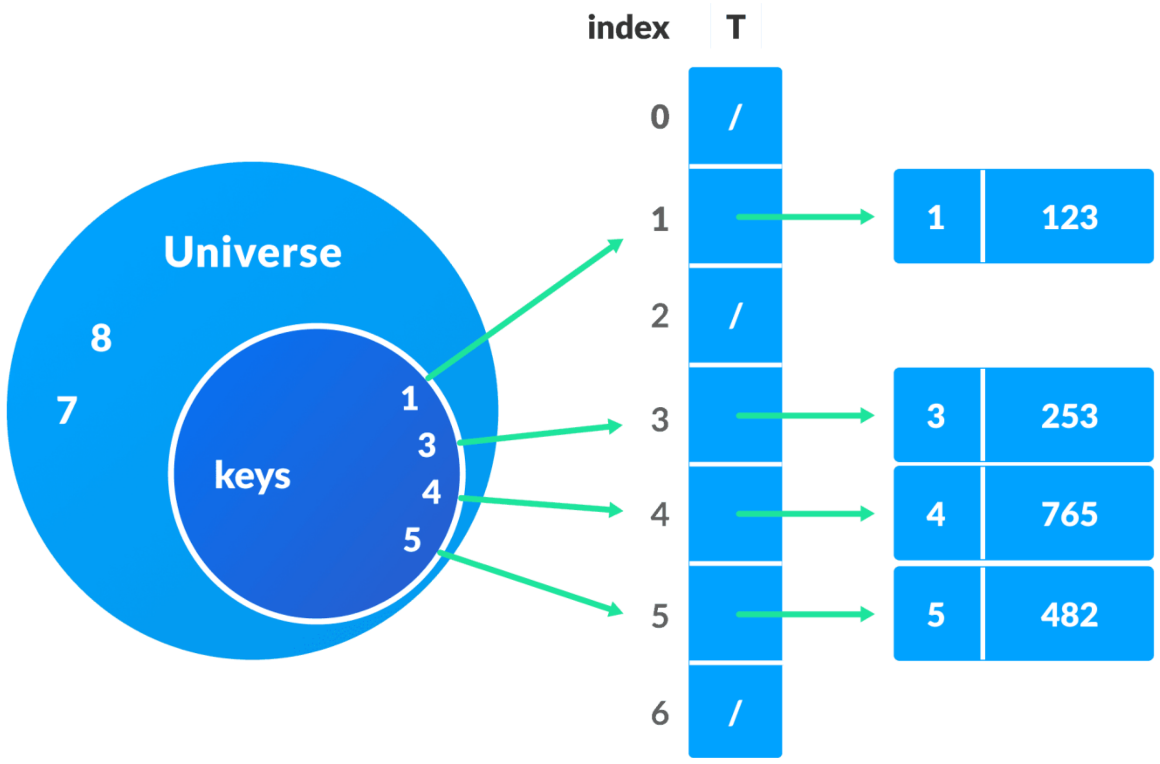
Direct address table is used when the amount of space used by the table is not a problem for the program. Here, we assume that

- the keys are small integers
- the number of keys is not too large, and
- no two data have the same key

A pool of integers is taken called universe `U = {0, 1, ……., n-1}`.

Each slot of a direct address table `T[0...n-1]` contains a pointer to the element that corresponds to the data.

The index of the array `T` is the key itself and the content of `T` is a pointer to the set `[key, element]`. If there is no element for a key then, it is left as `NULL`.


index
T
Universe
8
7
keys
1
3
4
5
0
1
2
3
4
5
6
/
/
/
1
123
3
253
4
765
5
482

Unit - IV

Sorting:

Insertion Sort

In this tutorial, you will learn how insertion sort works. Also, you will find working examples of insertion sort in C, C++, Java and Python.

Insertion sort works similarly as we sort cards in our hand in a card game.

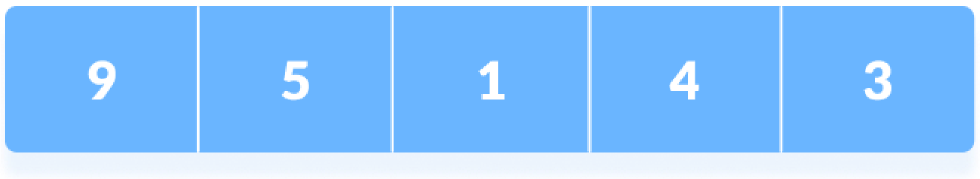
We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put at their right place.

A similar approach is used by insertion sort.

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

How Insertion Sort Works?

Suppose we need to sort the following array.

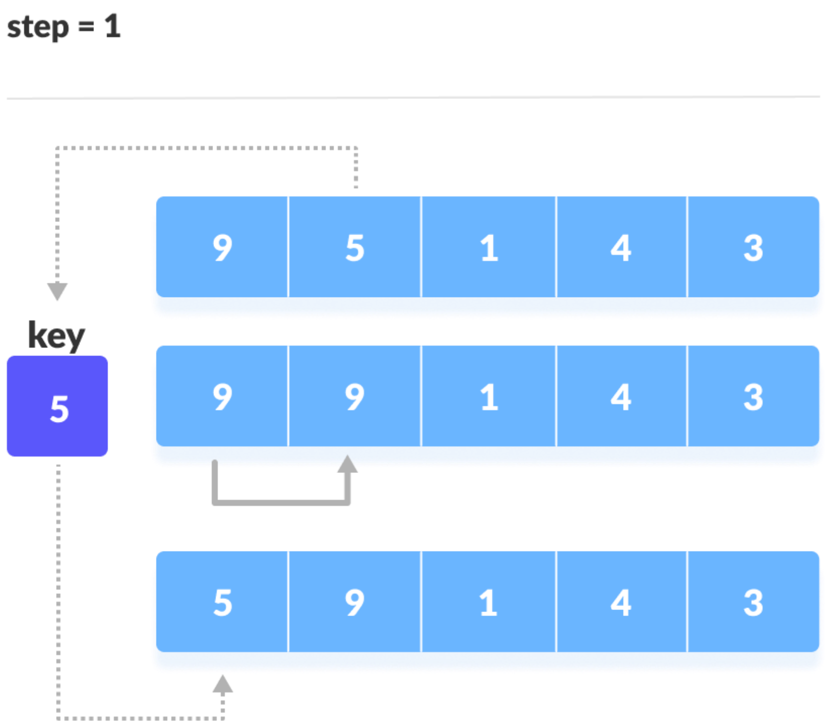


I

nitial array

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

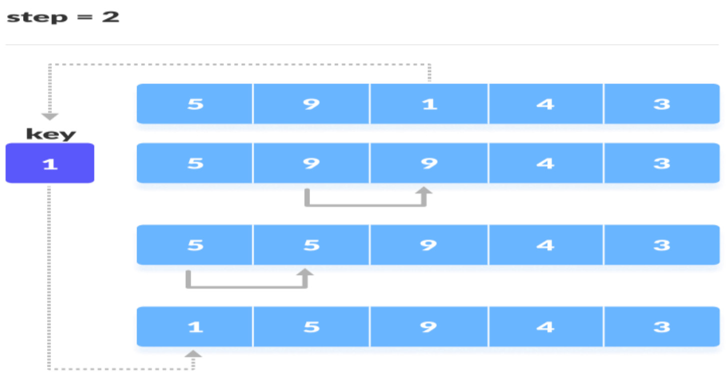
Compare `key` with the first element. If the first element is greater than `key`, then `key` is placed in front of the first element.



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

2. 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.



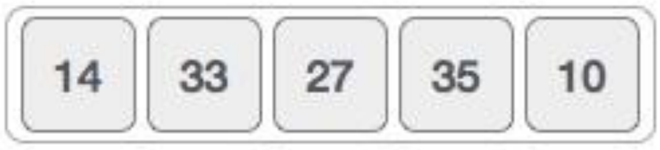
Place 1 at the beginning

Bubble Sorting

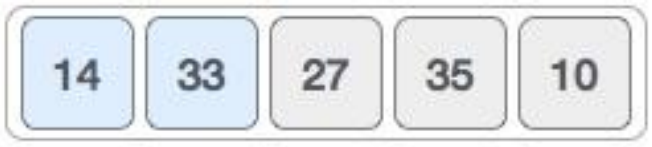
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

How Bubble Sort Works?

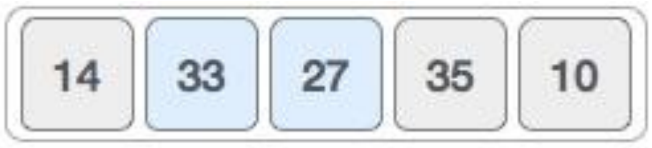
We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.



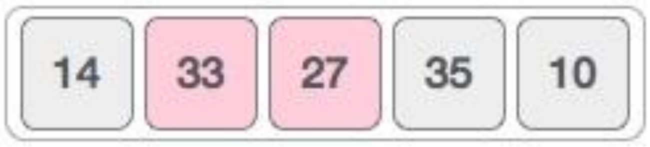
Bubble sort starts with very first two elements, comparing them to check which one is greater.



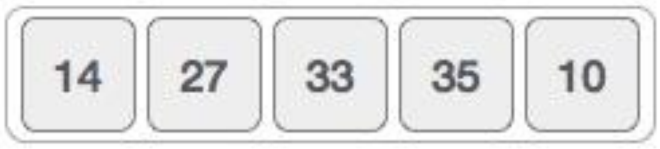
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



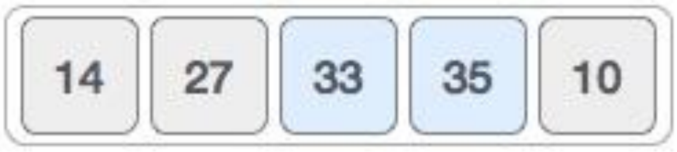
We find that 27 is smaller than 33 and these two values must be swapped.



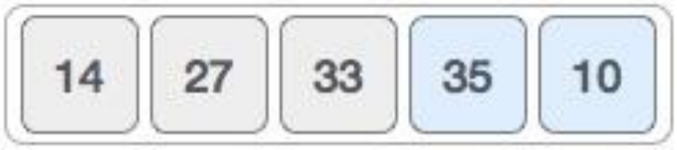
The new array should look like this −



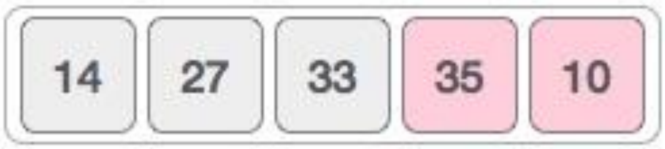
Next we compare 33 and 35. We find that both are in already sorted positions.



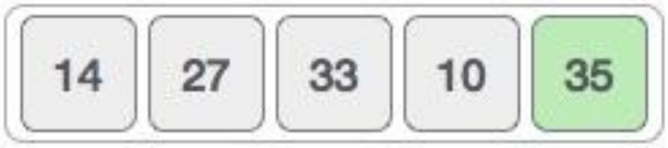
Then we move to the next two values, 35 and 10.



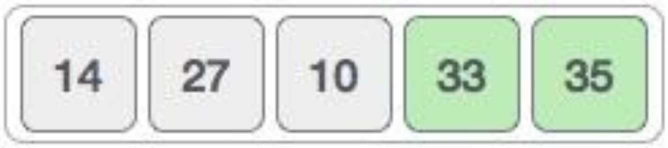
We know then that 10 is smaller 35. Hence they are not sorted.



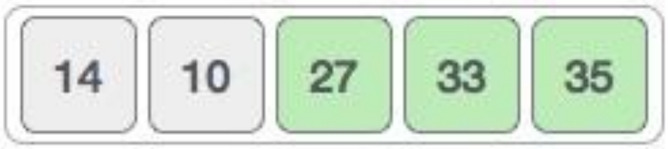
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



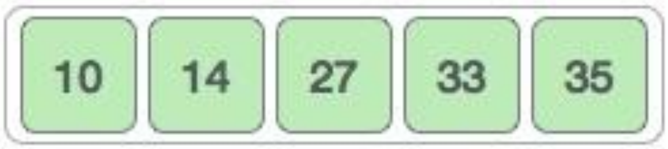
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
beginBubbleSort(list)

for all elements of list
if list[i]> list[i+1]
        swap(list[i], list[i+1])
endif
endfor

return list

endBubbleSort
```

Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of BubbleSort algorithm can be written as follows −

```
procedure bubbleSort( list : array of items )

   loop = list.count;

for i =0 to loop-1do:
      swapped =false

for j =0 to loop-1do:

/* compare the adjacent elements */
if list[j]> list[j+1]then
/* swap them */
           swap( list[j], list[j+1])
           swapped =true
endif

endfor
```

```
/*if no number was swapped that means
     array is sorted now, break the loop.*/

if(not swapped)then
break
endif

endfor

end procedure return list
```

Implementation

One more issue we did not address in our original algorithm and its improvised pseudocode, is that, after every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

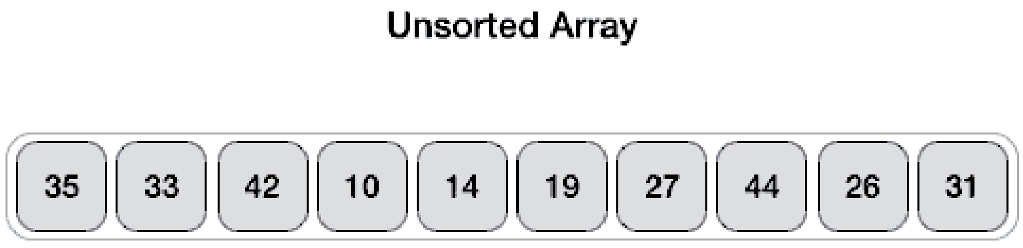
Quick Sort

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(nLogn) and image.png(n^2), respectively.

Partition in Quick Sort

Following animated representation explains how to find the pivot value in an array.



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as −

```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right -1

whileTruedo
while A[++leftPointer]< pivot do
//do-nothing
endwhile

while rightPointer >0&& A[--rightPointer]> pivot do
//do-nothing
endwhile

if leftPointer >= rightPointer
break
else
         swap leftPointer,rightPointer
endif

endwhile

   swap leftPointer,right
return leftPointer

endfunction
```

Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively

Quick Sort Pseudocode

To get more into it, let see the pseudocode for quick sort algorithm −

```
procedure quickSort(left, right)

if right-left <=0
return
else
    pivot = A[right]
    partition = partitionFunc(left, right, pivot)
    quickSort(left,partition-1)
    quickSort(partition+1,right)
endif

end procedure
```

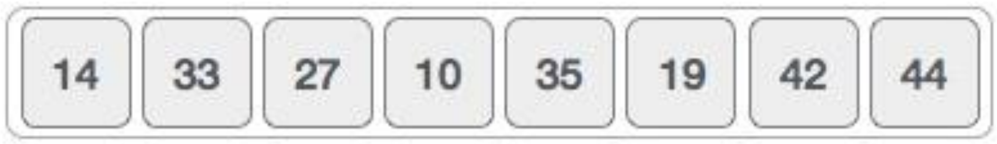
Two Way Merge Sort

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.

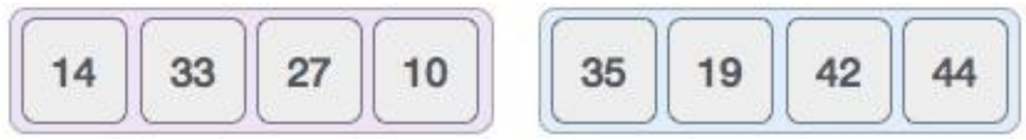
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

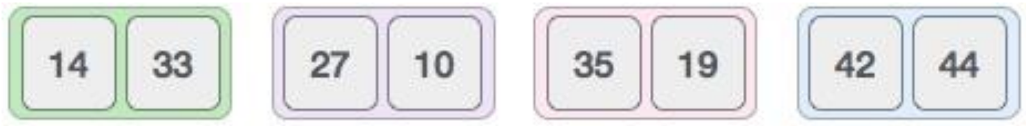
To understand merge sort, we take an unsorted array as the following −



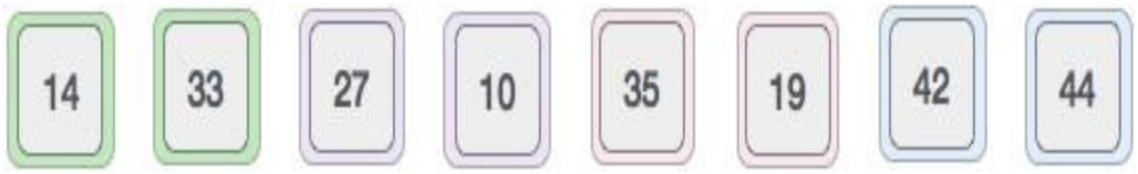
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



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

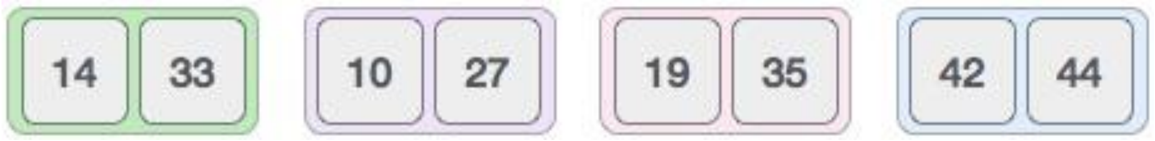


We further divide these arrays and we achieve atomic value which can no more be divided.

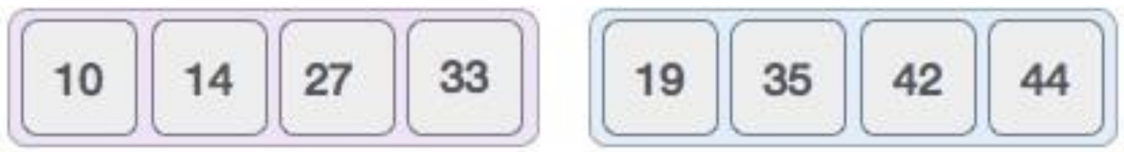


Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

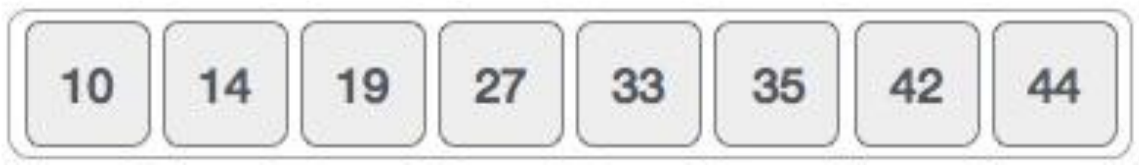
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed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

Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

```
procedure mergesort(var a as array )
if( n ==1)return a

var l1 as array = a[0]... a[n/2]
var l2 as array = a[n/2+1]... a[n]

  l1 = mergesort( l1 )
  l2 = mergesort( l2 )

return merge( l1, l2 )
end procedure

procedure merge(var a as array,var b as array )

var c as array
while( a and b have elements )
if( a[0]> b[0])
add b[0] to the endof c
remove b[0]from b
else
add a[0] to the endof c
remove a[0]from a
endif
endwhile

while( a has elements )
add a[0] to the endof c
remove a[0]from a
endwhile

while( b has elements )
add b[0] to the endof c
```

```
remove b[0]from b
endwhile

return c

end procedure
```

Heap Sort

In this tutorial, you will learn how heap sort algorithm works. Also, you will find working examples of heap sort in C, C++, Java and Python.

Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees.

The initial set of numbers that we want to sort is stored in an array e.g. `[10, 3, 76, 34, 23, 32]` and after sorting, we get a sorted array `[3,10,23,32,34,76]`
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

A complete binary tree has an interesting property that we can use to find the children and parents of any node.

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`.

Let's test it out,

```
Left child of 1 (index 0)

= element in (2*0+1) index

= element in 1 index

= 12



Right child of 1

= element in (2*0+2) index

= element in 2 index

= 9


Similarly,

Left child of 12 (index 1)

= element in (2*1+1) index

= element in 3 index

= 5


Right child of 12

= element in (2*1+2) index

= element in 4 index

= 6
```

Let us also confirm that the rules hold for finding parent of any node

```
Parent of 9 (position 2)

= (2-1)/2

= ½

= 0.5

~ 0 index

= 1



Parent of 12 (position 1)

= (1-1)/2

= 0 index

= 1
```

Sorting on Different Keys

When you sort with multiple keys, the system does not use an interim file with a prepended key. Instead it writes a temporary batch file for input into the external sort. The SortCommand specified here calls a GNU Sort:

< SortBatchOptions >

BuildSortKey = No

SortCommand = sort -o **TargetFile** *{**[[ ]] -k **FieldOffset**, **FieldLength**** }* **SourceFile**

The data between the "*{" and "}*" (in bold) is replicated for each sort field specified in the sort batches entry. The data between the "[[" and "]]" is used as a field separators.

SortCommand = sort -o **TargetFile** *{[[ ]] -k **FieldOffset**, **FieldLength** }* **SourceFile**

FieldOffset and **FieldLength** are replacement strings you can use inside a repeating section. See Replacement Strings for a complete list of available replacement strings.

Given the sample INI values defined above and the sample RCB DFD file definition, the generated sort command would appear as follows:

```
sort -o .\data\AGENT.wrk -k 1,22 -k 23,4 -k 27,45 .\data\AGENT.tmp
```

Sorting with an OptTech Sort

OTSort by OptTech is a third-party sort utility. Here is an example of how you could set the SortCommand options to execute OTSort with the SortBatches rule:

```
< SortBatchOptions >

BuildSortKey = No

SortCommand = OTSW32D **Sou
```

Practical consideration for Internal Sorting

Sorting is a technique through which we arrange the data in such a manner so that the searching of the data becomes easy. A lot of sorting techniques has been implemented till now to cope up the faster execution of the result and to manage the data comfortably . Sorting and Searching are fundamental operations in computer science. Sorting refers to the operation of arranging data in some given order. Searching refers to the operation of searching the particular record from the existing information. Normally, the information retrieval involves searching, sorting and merging. In this chapter we will discuss the searching and sorting techniques in detail.Sorting is very important in every computer application. Sorting refers to arranging of data elements in some given order. Many Sorting algorithms are available to sort the given set of elements.

Let get to know about two sorting techniques and analyze their performance. The two techniques are:

Internal Sorting

External Sorting

Internal Sorting takes place in the main memory of a computer. The internal sorting methods are applied to small collection of data. It means that, the entire collection of data to be sorted in small enough that the sorting can take place within main memory. We will study the following methods of internal sorting

1. Insertion sort

2. Selection sort

3. Merge Sort

4. Radix Sort

5. Quick Sort

6. Heap Sort

7. Bubble Sort

Also a lot of algorithms are involved in sorting . Hence we should understand first that what is an algorithm .

Informally, an algorithm is any well-defined computational procedure that takes some value,

or set of values, as input and produces some value, or set of values, as output. An algorithm is

thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a tool for solving a well-specified computational problem.

The statement of the problem specifies in general terms the desired input/output relationship.

The algorithm describes a specific computational procedure for achieving that input/output

relationship.

For example, one might need to sort a sequence of numbers into non decreasing order. This

problem arises frequently in practice and provides fertile ground for introducing many

standard design techniques and analysis tools. Here is how we formally define the sorting problem.

Insertion Sort

This is a naturally occurring sorting method exemplified by a card player arranging the cards dealt to him. He picks up the cards as they are dealt and inserts them into the required position. Thus at every step, we insert an item into its proper place in an already ordered list.

We will illustrate insertion sort with an example (refer to Figure 10.1) before presenting the formal algorithm.

Sort the following list using the insertion sort method:

Thus to find the correct position search the list till an item just greater than the target is found. Shift all the items from this point one down the list. Insert the target in the vacated slot. Repeat this process for all the elements in the list. This results in sorted list.

Bubble Sort

In this sorting algorithm, multiple swappings take place in one pass. Smaller elements move or ‘bubble’ up to the top of the list, hence the name given to the algorithm.

In this method, adjacent members of the list to be sorted are compared.If the item on top is greater than the item immediately below it, then they are swapped. This process is carried on till the list is sorted.

The detailed algorithm follows:

```
Algorithm: BUBBLE SORT 6

1. Begin

2. Read the n elements

3. for i=1 to n

for j=n downto i+1

if a[j] <= a[j-1]

swap(a[j],a[j-1])

4. End // of Bubble Sort
```

Total number of comparisons in Bubble sort :

= (N-1) +(N-2) . . . + 2 + 1

= (N-1)*N / 2 =O(N2)

This inefficiency is due to the fact that an item moves only to the next position in each pass.

Quick Sort

This is the most widely used internal sorting algorithm. In its basic form, it was invented by C.A.R. Hoare in 1960. Its popularity lies in the ease of implementation, moderate use of resources and acceptable behaviour for a variety of sorting cases. The basis of quick sort is the divide and conquer strategy i.e. Divide the problem [list to be sorted] into sub-problems [sub-lists], until solved sub problems [sorted sub-lists] are found. This is implemented as follows:

Choose one item A[I] from the list A[ ].

Rearrange the list so that this item is in the proper position, i.e., all preceding items have a lesser value and all succeeding items have a greater value than this item.

1. Place A[0], A[1] .. A[I-1] in sublist 1

2. A[I]

3. Place A[I + 1], A[I + 2] … A[N] in sublist 2

Repeat steps 1 & 2 for sublist1 & sublist2 till A[ ] is a sorted list.

As can be seen, this algorithm has a recursive structure.

The divide' procedure is of utmost importance in this algorithm. This is usually implemented as follows:

1. Choose A[I] as the dividing element.

2. From the left end of the list (A[O] onwards) scan till an item A[R] is found whose value is greater than A[I].

3. From the right end of list [A[N] backwards] scan till an item A[L] is found whose value is less than A[1].

4. Swap A[R] & A[L].

5. Continue steps 2, 3 & 4 till the scan pointers cross. Stop at this stage.

6. At this point, sublist1 & sublist are ready.

7. Now do the same for each of sublist1 & sublist2.

C:UserssaurabhDesktopbubble-sort-3.pngC:UserssaurabhDesktopbubble_sort.jpg

EXTERNAL SORT GENERAL

Binary Search Trees:

Binary Search Tree (BST)

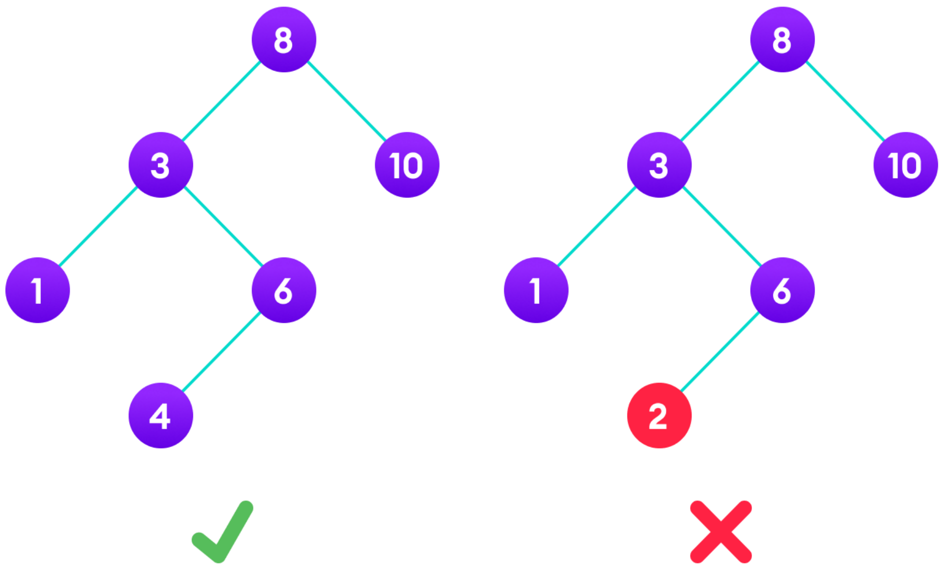
In this tutorial, you will learn how Binary Search Tree works. Also, you will find working examples of Binary Search Tree in C, C++, Java and Python.

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular binary tree is

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties



A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree

The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

There are two basic operations that you can perform on a binary search tree:

Search Operation

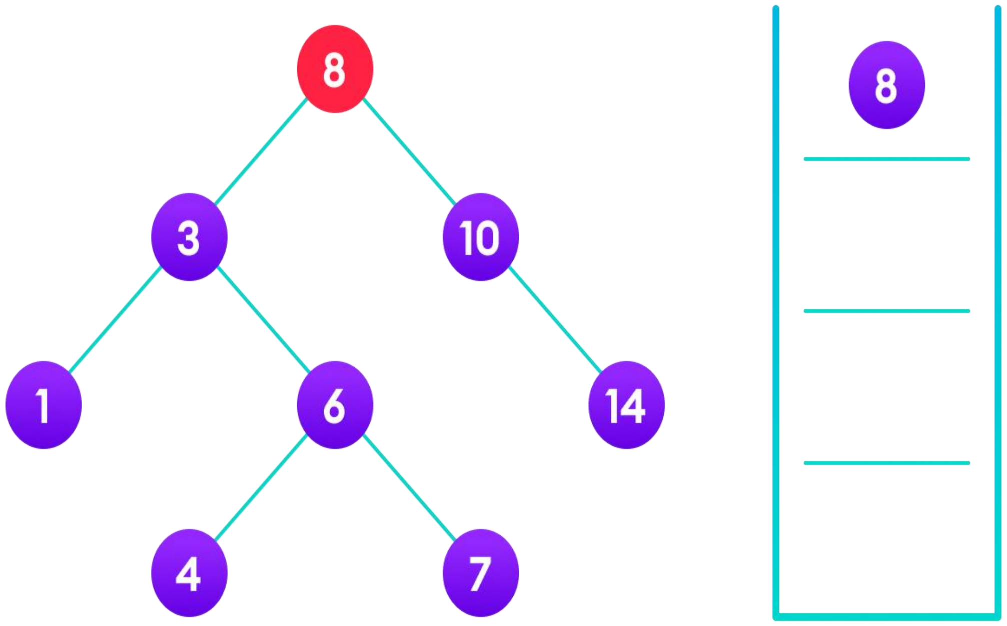
The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

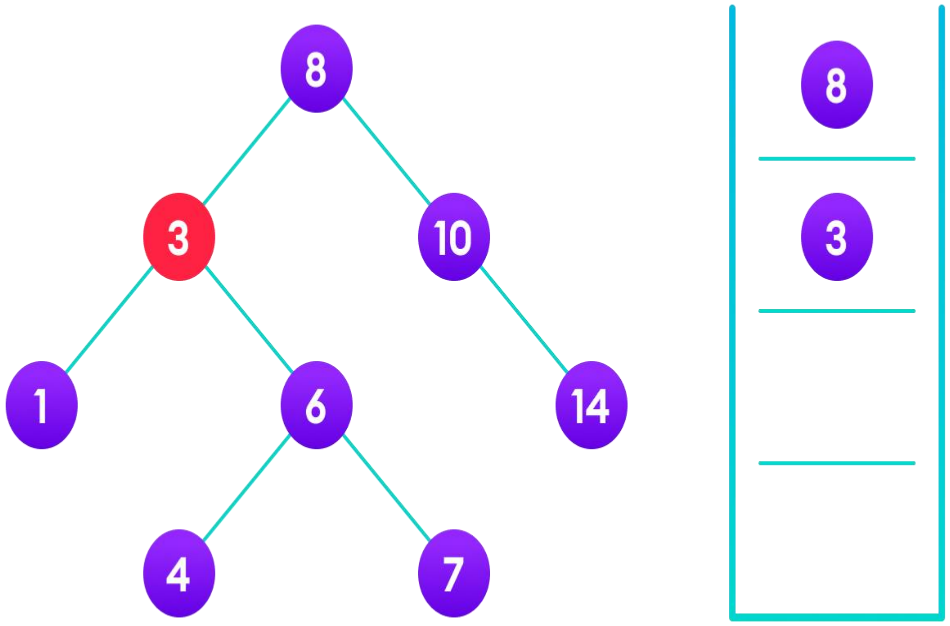
Algorithm:

```
If root == NULL
return NULL;
If number == root->data
return root->data;
If number < root->data
return search(root->left)
If number > root->data
return search(root->right)
```

Let us try to visualize this with a diagram.

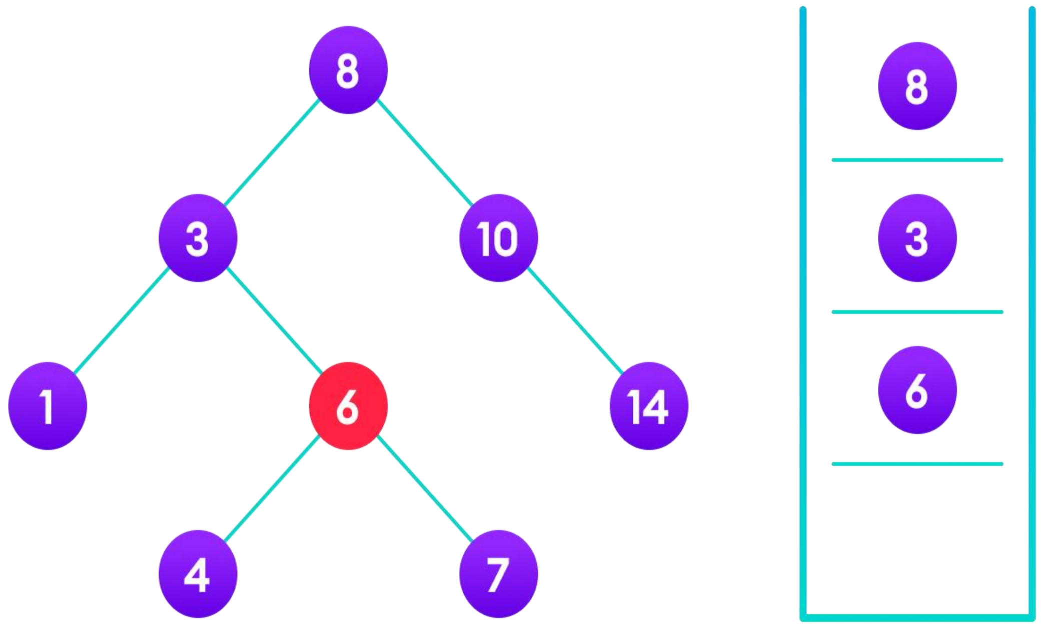


4 is not found so, traverse through the left subtree of 8

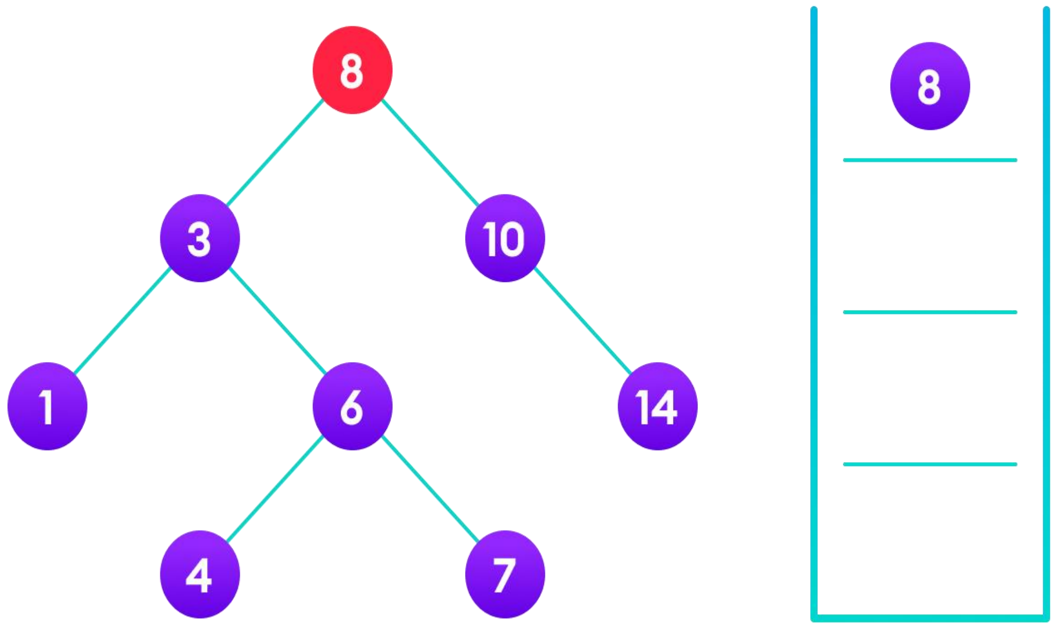


4 is

not found so, traverse through the right subtree of 3



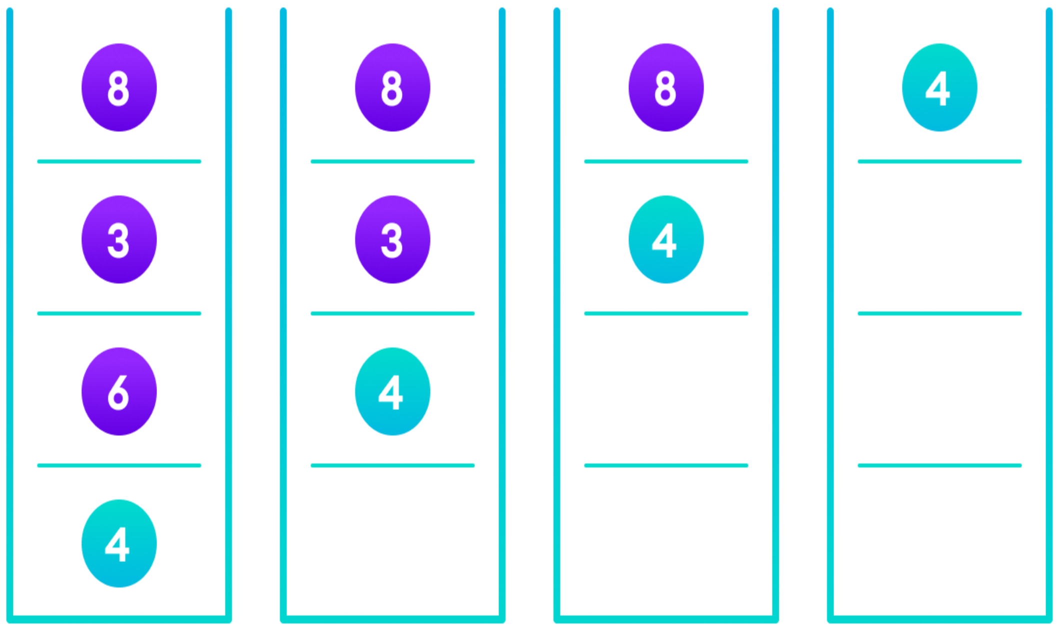
4 is not found so, traverse through the left subtree of 6



4 is found

If the value is found, we return the value so that it gets propagated in each recursion step as shown in the image below.

If you might have noticed, we have called return search(struct node*) four times. When we return either the new node or NULL, the value gets returned again and again until search(root) returns the final result.



If the value is found in any of the subtrees, it is propagated up so that in the end it is returned, otherwise null is returned

If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

Insert Operation

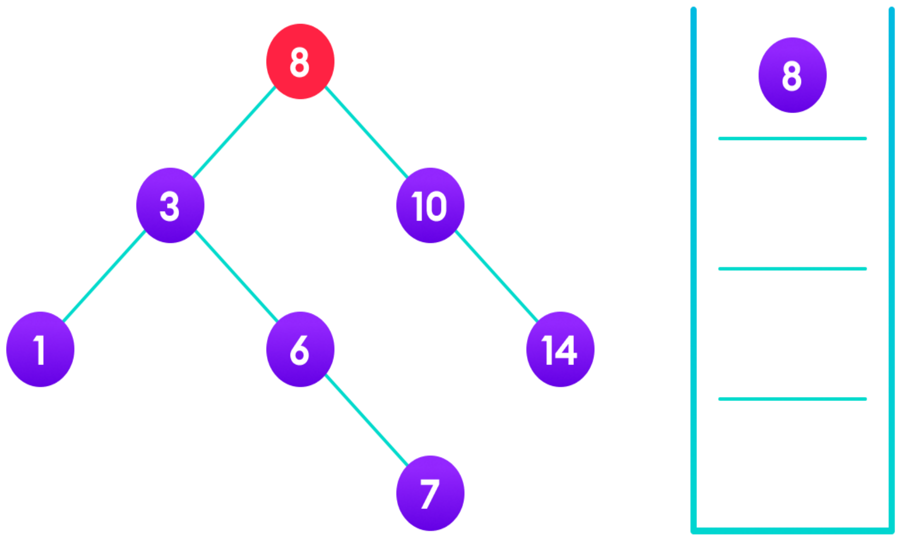
Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.

We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

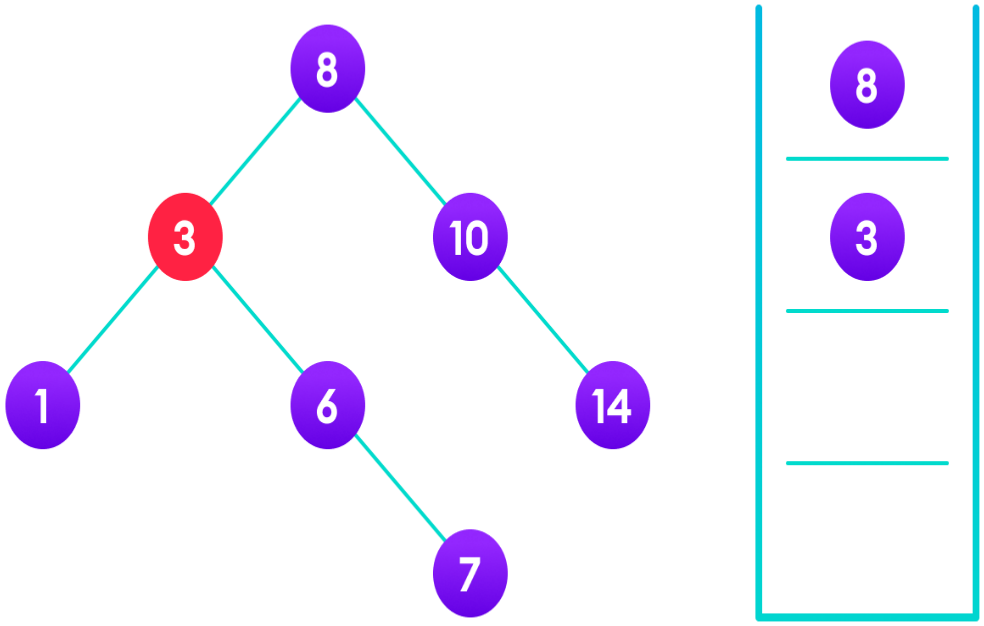
Algorithm:

```
If node == NULL
return createNode(data)
if (data< node->data)
    node->left  = insert(node->left, data);
elseif (data> node->data)
    node->right = insert(node->right, data);
return node;
```

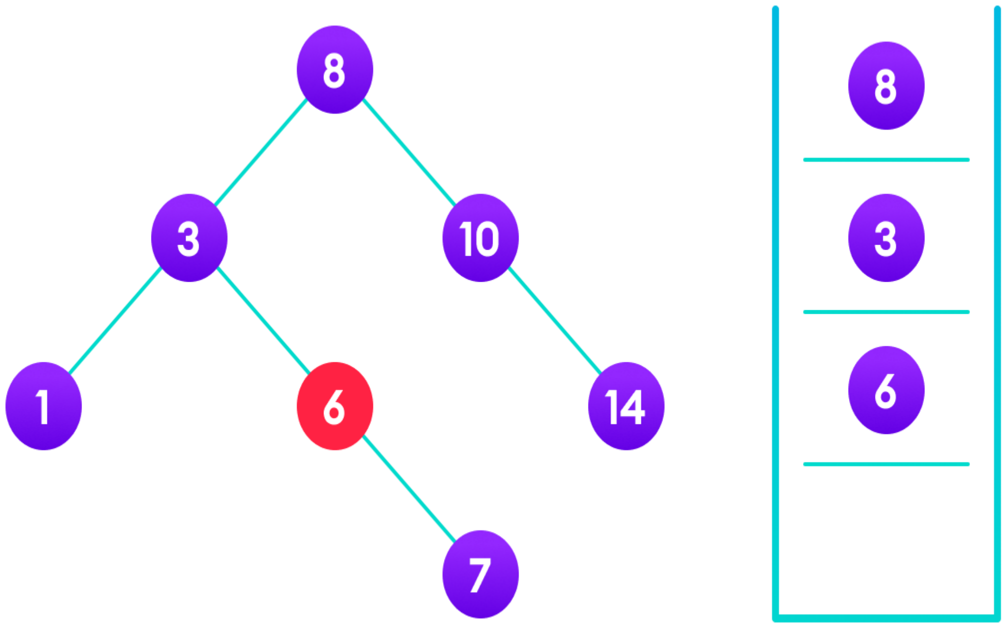
The algorithm isn't as simple as it looks. Let's try to visualize how we add a number to an existing BST.



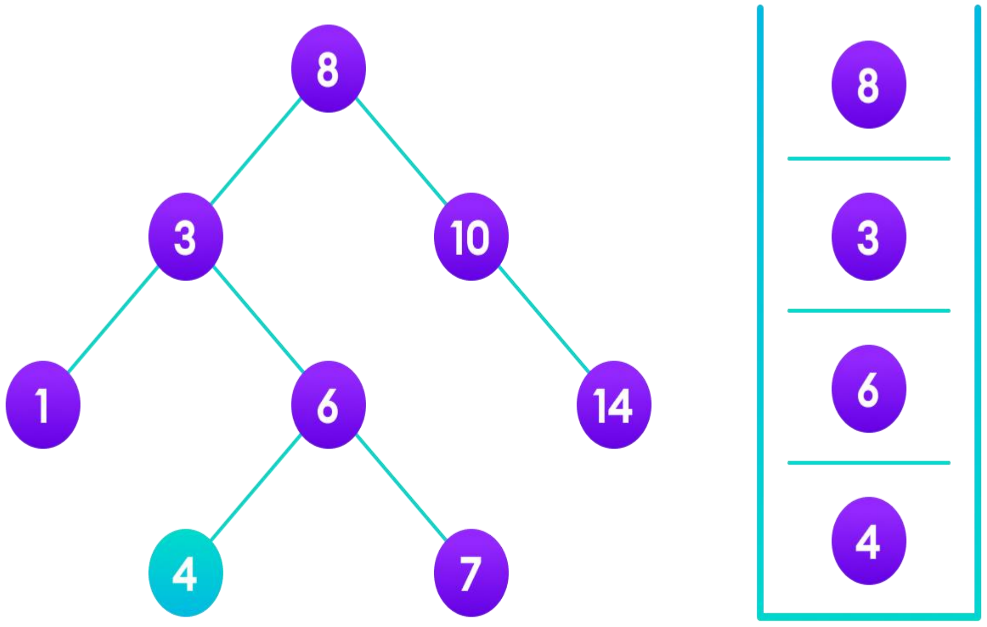
4<8 so, transverse through the left child of 8



4>3 so, transverse through the right child of 8



4<6 so, transverse through the left child of 6



Insert 4 as a left child of 6

We have attached the node but we still have to exit from the function without doing any damage to the rest of the tree. This is where the `return node;` at the end comes in handy. In the case of `NULL`, the newly created node is returned and attached to the parent node, otherwise the same node is returned without any change as we go up until we return to the root.

This makes sure that as we move back up the tree, the other node connections aren't changed.

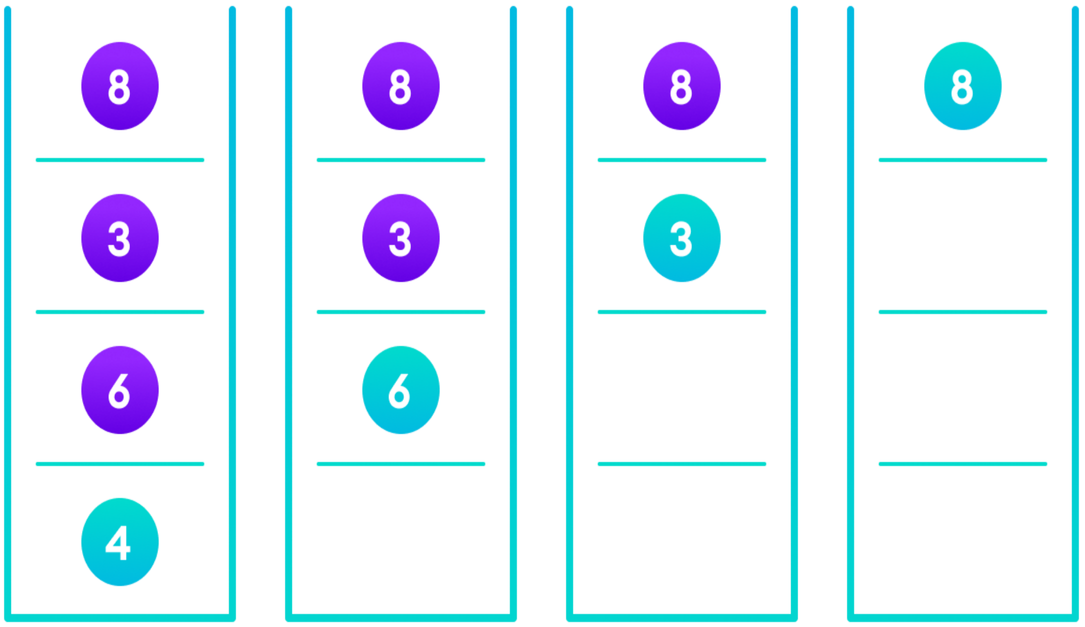


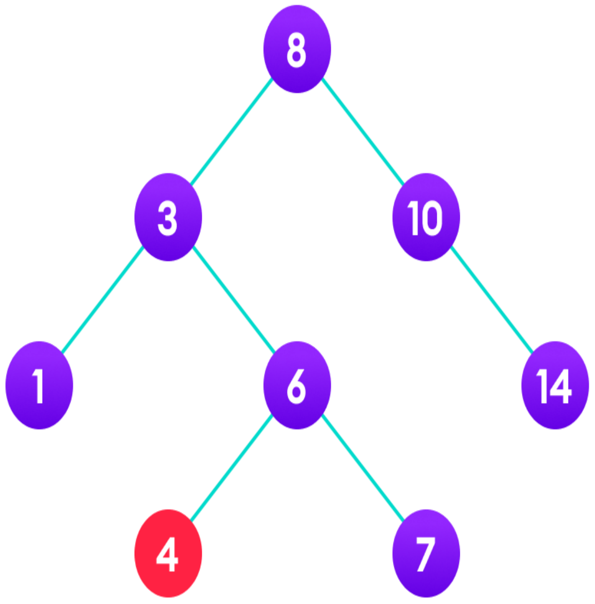
Image showing the importance of returning the root element at the end so that the elements don't lose their position during the upward recursion step.

Deletion Operation

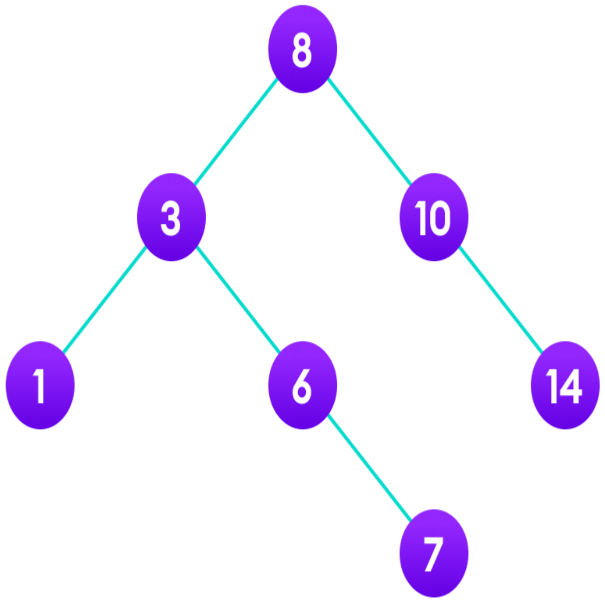
There are three cases for deleting a node from a binary search tree.

Case I

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.



4 is to be deleted

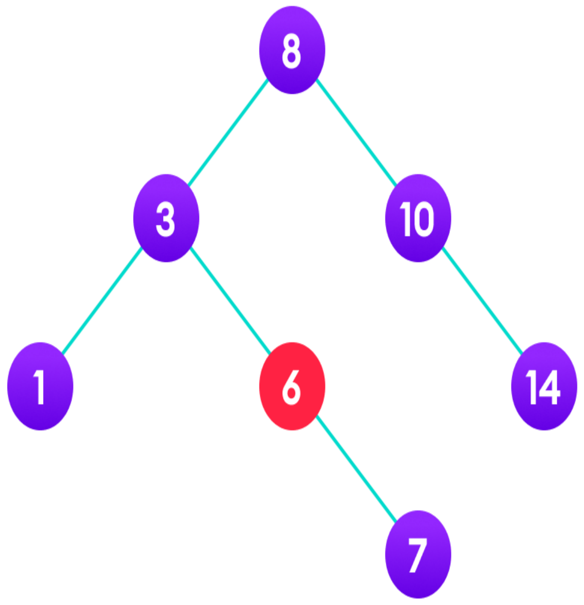


Delete the node

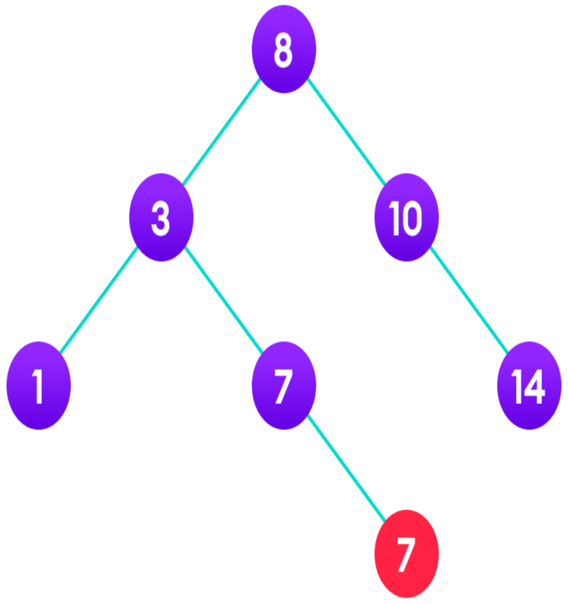
Case II

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

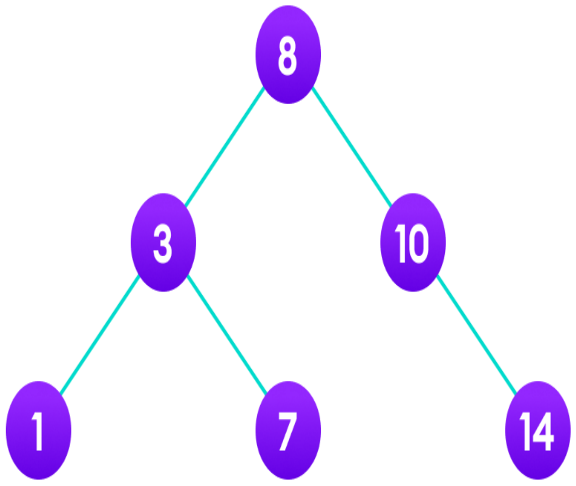
1. Replace that node with its child node.
2. Remove the child node from its original position.



6 is to be deleted



copy the value of its child to the node and delete the child



Final tree

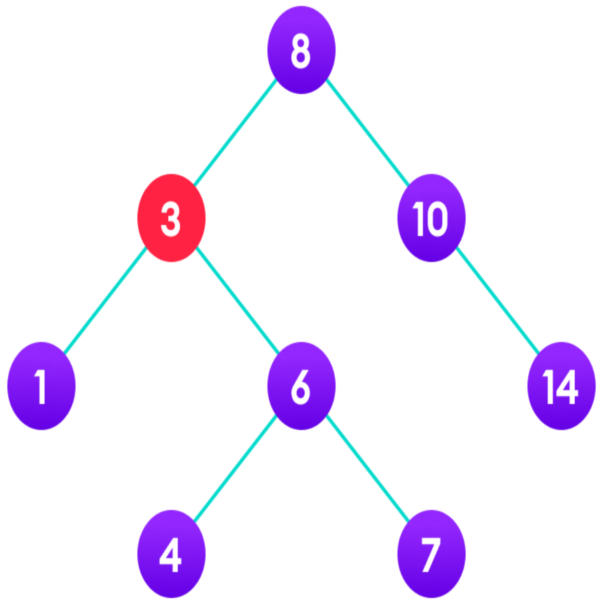
Case III

In the third case, the node to be deleted has two children. In such a case follow the steps below:

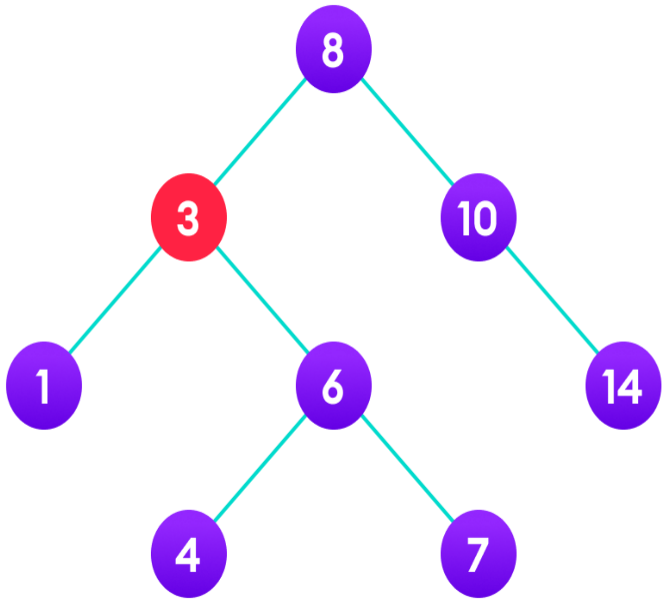
1. Get the inorder successor of that node.

2. Replace the node with the inorder successor.

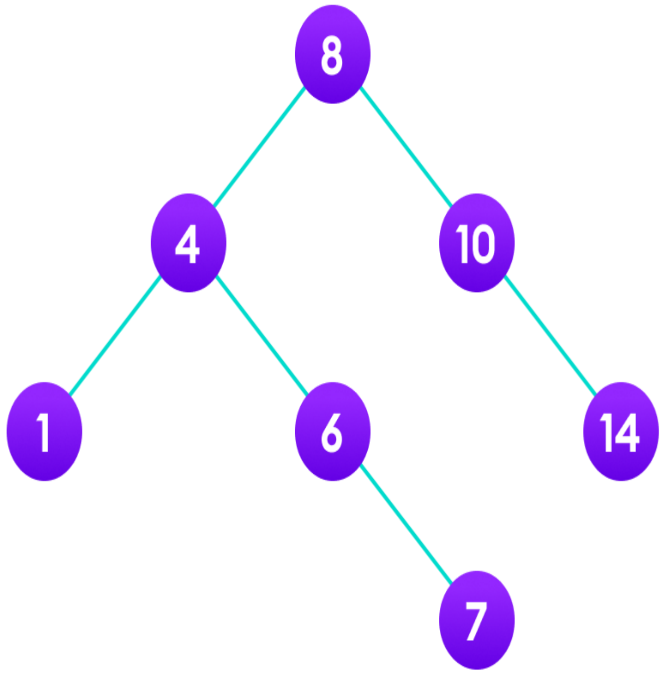
3. Remove the inorder successor from its original position.



3 is to be deleted



Copy the value of the inorder successor (4) to the node



Delete the inorder successor

Python, Java and C/C++ Examples

Python

Java

C

C++

```
# Binary Search Tree operations in Python


# Create a node
classNode:
def__init__(self, key):
        self.key = key
        self.left = None
        self.right = None


# Inorder traversal
definorder(root):
if root isnotNone:
# Traverse left
        inorder(root.left)

# Traverse root
print(str(root.key) + "->", end=' ')

# Traverse right
        inorder(root.right)


# Insert a node
definsert(node, key):

# Return a new node if the tree is empty
if node isNone:
return Node(key)
```

```
# Traverse to the right place and insert the node
if key < node.key:
        node.left = insert(node.left, key)
else:
        node.right = insert(node.right, key)

return node


# Find the inorder successor
defminValueNode(node):
    current = node

# Find the leftmost leaf
while(current.left isnotNone):
        current = current.left

return current


# Deleting a node
defdeleteNode(root, key):

# Return if the tree is empty
if root isNone:
return root

# Find the node to be deleted
if key < root.key:
        root.left = deleteNode(root.left, key)
elif(key > root.key):
        root.right = deleteNode(root.right, key)
else:
# If the node is with only one child or no child
if root.left isNone:
            temp = root.right
            root = None
return temp

elif root.right isNone:
            temp = root.left
            root = None
```

```
return temp

# If the node has two children,
# place the inorder successor in position of the node to be deleted
        temp = minValueNode(root.right)

        root.key = temp.key

# Delete the inorder successor
        root.right = deleteNode(root.right, temp.key)

return root


root = None
root = insert(root, 8)
root = insert(root, 3)
root = insert(root, 1)
root = insert(root, 6)
root = insert(root, 7)
root = insert(root, 10)
root = insert(root, 14)
root = insert(root, 4)

print("Inorder traversal: ", end=' ')
inorder(root)

print("\nDelete 10")
root = deleteNode(root, 10)
print("Inorder traversal: ", end=' ')
inorder(root)
```

Insertion and Deletion in BST

We have already discussed the Tree data structure and a Binary Search Tree. So here we are going to discuss insert, search & delete operations in BST:

1)Insertion

2)Search

3)Deletion

Insertion

The basic Binary Search tree property says that, if an element is greater then the root node, it will be present in the right subtree and if it is smaller then the root node then it will be present in the left subtree. So, for insertion, we will use this basic property of BST

Algorithm:

1. Go the root node of Tree, compare the value to be inserted with the current node value.
2. If the value to be inserted is smaller then or equal to the root node value, go to the left subtree, else go to the right subtree.
3. Compare the value again with the root node value of the subtree, and follow the step2 again.
4. On reaching the end node, insert the value left(if the value is smaller then current node value), else right.

```
insert_BST(value, tree){

   if(tree == NULL)

      return new Node(value);



   if(tree->value >= value)

      tree->left = insert_BST(value, tree->left)

   else

      tree->right = insert_BST(value, tree->right)
   return tree;

}
```

Search

We will follow here logic similar to insertion i.e if the node if the root node value is equal to the value to be searched, then we will return *search is successful*, else if the root node value is less then the value to be searched, then we will move to the left subtree to find the element, else we will move to the right subtree to search the element.

Algorithm

```
search_BST(value, tree){

    if (tree == NULL)

        return "tree is empty";



    if(tree->value == value)

        return "search is successful";

    else if (tree->value > value)

        search_BST(value, tree->left);

    else

        search_BST(value, tree->right);

}
```

Deletion

Deletion is a little tricky is Binary Search Tree. One important thing to consider is that after a deletion operation any of the BST property should not be violated.

Algorithm

In deletion operation any of the 3 cases will arise:

1.

```
1.  Node to be deleted is a leaf node

2.        10                              10

3.      /    \       delete(9)            /    \

4.     5      17       -------->          5      17

5.   /  \    /  \                       /      /  \

     3   9   16   18                    3      16   18
```

If the node to be deleted is a leaf node, then we will delete it directly from the tree

```
6.  Node to be deleted have only one child

7.        10                             10

8.      /    \        delete(5)           /    \

9.  5        17       -------->8       17

10.     \     /  \                      /  \    /   \

11.8    16   18                     7    9   16   18

12.    /  \
          7    9
```

If the node to be deleted has only one child node, then we will delete the node and attach its child(left or right) to its parent node. *Here we deleted node 5 and attached its child 8 to its parent node 10.*

13. **Node to be deleted have both the child**

```
14.         10                             10
15.       /    \          delete(5)          /     \
16. 5        17        -------->4        17
17.   /   \     /  \                        /   \     /   \
18.  2     8   16   18                        2    8   16   18
19. / \   / \                            /    /  \
20. 1  4  7   9                            1    7   9
21.
22. OR
23.
24.
25.         10                             10
26.       /    \          delete(5)          /     \
27. 5        17        ---------->7        17
28.   /   \     /  \                        /   \     /   \
29.  2     8   16   18                        2     8   16   18
```

30. / \   / \   / \   \

31. 1 4 **7**   9   1   4   9

If the node to be deleted has both the child, then delete the node and replace it with either the rightmost child of its left successor or the leftmost child of its right successor. *Here in the 1st tree, we replaced 5 with the rightmost child 4 of its left successor 2 and in the second tree, we replaced 5 with the leftmost child 7 of its left successor 8.*

CPP Implementation of above Algorithm

```
#include <bits/stdc++.h>

usingnamespacestd;


classNode{

  public:

    intdata;

    Node *left;

    Node *right;


  public:

    Node(intx){

      data = x;

      left = NULL;

      right = NULL;

    }


};
```

```
Node *insert_BST(Node *tree, intval)
{
  if(tree == NULL){
    cout<<val <<" is inserted into BST successfully"<<endl;
    return(newNode(val));
  }
  if(tree->data >= val)
    tree->left = insert_BST(tree->left, val);
  else
    tree->right = insert_BST(tree->right, val);

  returntree;
}

intsearch_BST(Node *tree, intval)
{
  if(tree == NULL){
    cout<<val<<" is not present in the tree\n";
    return0;
  }

  string ret_val;
  if(tree->data == val)
    cout<<val<<" is present in the tree\n";
  elseif(tree->data > val)
    search_BST(tree->left, val);
  else
     search_BST(tree->right, val);

  return0;

}
Node* delete_BST(Node *tree, intval)
{
  if(tree == NULL){
    cout<<"value is not present in BST"<<endl;
    returntree;
  }

  if(tree->data > val)
    tree->left = delete_BST(tree->left, val);
  elseif(tree->data < val)
    tree->right = delete_BST(tree->right, val);

  else{
    //if left child of the node is empty
    if(tree->left == NULL){
      Node *temp = tree->right;
      free(tree);
      tree= temp;
    }
    //if right child of the node is empty
    elseif(tree->right == NULL){
      Node *temp = tree->left;
      tree = temp;
      free(temp);
    }
    //if both left and right child exists for the node
    else{
      Node *temp = tree->left;
      while(temp->right->right!=NULL){   // traverse until you don't reach, the second last right node of th
        temp = temp->right;
      }
      tree->data = temp->right->data;       // update the value to be deleted with the value of the rightmos
      Node *temp2 = temp->right->left;      // pointer to the left child of the last right node
      free(temp->right);                    // delete the last node
      temp->right = temp2;                  // assign the left child of last right node to second last right node
    }

  }
  returntree;
}

voidinorder(Node *tree)
```

```
{
    if(tree != NULL)
    {
        inorder(tree->left);
        cout<<tree->data<<" ";
        inorder(tree->right);
    }
}

intmain(){

  Node *tree;

  tree = newNode(10);

  tree->left = newNode(5);
  tree->left->left = newNode(2);
  tree->left->right = newNode(8);
  tree->left->left->left = newNode(1);
  tree->left->left->right = newNode(4);
  tree->left->right->left = newNode(7);

  tree->right = newNode(17);
  tree->right->left = newNode(16);
  tree->right->right = newNode(18);
  insert_BST(tree, 9);

  cout<<"inorder traversal of the current tree is: ";

  inorder(tree);

  cout<<endl;


  search_BST(tree,9);


  tree = delete_BST(tree,5);

  cout<<"deleted 5 successfully \n";

  search_BST(tree,5);

  cout<<"inorder traversal of the current tree is: ";

  inorder(tree);

  cout<<endl;


  return0;

}
```

Time Complexity

The time complexity of *Insert operation* is *O(N)*, *Search Operation* is *O(N)* & *Delete Operation* is *O(N),* where N is the number of nodes in BST. *Note, this worst-case time complexity and will occur when the tree is left or right-skewed.*

This is how we will perform insert, search & delete operation in Binary Search Tree (BST).

Complexity of Search Algorithm

We have learned that in order to write a computer program which performs some task we must construct a suitable algorithm. However, whatever algorithm we construct is unlikely to be unique – there are likely to be many possible algorithms which can perform the same task. Are some of these algorithms in some sense better than others? Algorithm analysis is the study of this question.

In this chapter we will analyse four algorithms; two for each of the following common tasks:

- sorting: ordering a list of values
- searching: finding the position of a value within a list

Algorithm analysis should begin with a clear statement of the task to be performed. This allows us both to check that the algorithm is correct and to ensure that the algorithms we are comparing perform the same task.

Although there are many ways that algorithms can be compared, we will focus on two that are of primary importance to many data processing algorithms:

- time complexity: how the number of steps required depends on the size of the input
- space complexity: how the amount of extra memory or storage required depends on the size of the input

Note

Common sorting and searching algorithms are widely implemented and already available for most programming languages. You will seldom have to implement them yourself outside of the exercises in these notes. It is nevertheless important for you to understand these basic algorithms, because you are likely to use them within your own programs – their space and time complexity will thus affect that of your own algorithms. Should you need to select a specific sorting or searching algorithm to fit a particular task, you will require a good understanding of the available options.

Sorting algorithms

The sorting of a list of values is a common computational task which has been studied extensively. The classic description of the task is as follows:

Given a list of values and a function that compares two values, order the values in the list from smallest to largest.

The values might be integers, or strings or even other kinds of objects. We will examine two algorithms:

- Selection sort, which relies on repeated selection of the next smallest item
- Merge sort, which relies on repeated merging of sections of the list that are already sorted

Other well-known algorithms for sorting lists are insertion sort, bubble sort, heap sort, quicksort and shell sort.

There are also various algorithms which perform the sorting task for restricted kinds of values, for example:

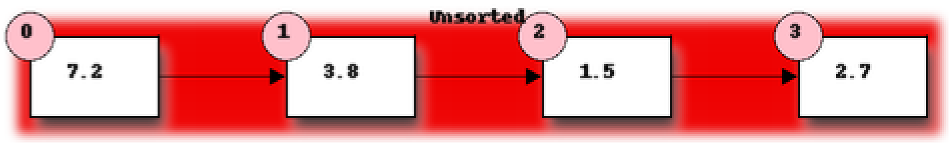
- Counting sort, which relies on the values belonging to a small set of items
- Bucket sort, which relies on the ability to map each value to one of a small set of items
- Radix sort, which relies on the values being sequences of digits

If we restrict the task, we can enlarge the set of algorithms that can perform it. Among these new algorithms may be ones that have desirable properties. For example, Radix sort uses fewer steps than any generic sorting algorithm.

Selection sort

To order a given list using selection sort, we repeatedly select the smallest remaining element and move it to the end of a growing sorted list.

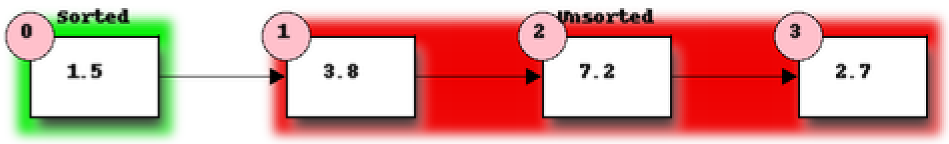
To illustrate selection sort, let us examine how it operates on a small list of four elements:



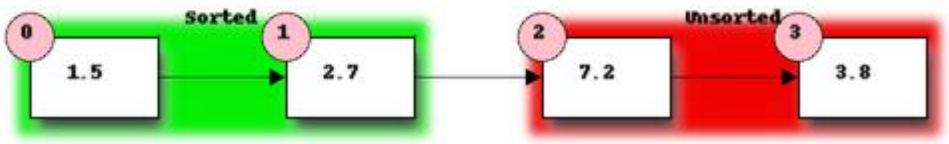
Initially the entire list is unsorted. We will use the front of the list to hold the sorted items – to avoid using extra storage space – but at the start this sorted list is empty.

First we must find the smallest element in the unsorted portion of the list. We take the first element of the unsorted list as a candidate and compare it to each of the following elements in turn, replacing our candidate with any element found to be smaller. This requires 3 comparisons and we find that element 1.5 at position 2 is smallest.

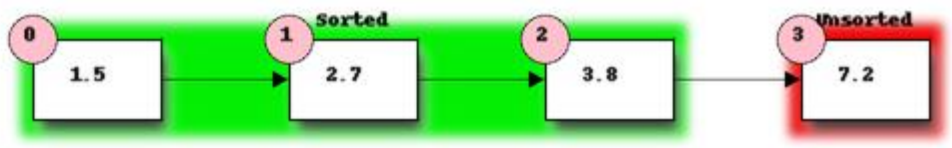
Now we will swap the first element of our unordered list with the smallest element. This becomes the start of our ordered list:



We now repeat our previous steps, determining that 2.7 is the smallest remaining element and swapping it with 3.8 – the first element of the current unordered section – to get:



Finally, we determine that 3.8 is the smallest of the remaining unordered elements and swap it with 7.2:



The table below shows the number of operations of each type used in sorting our example list:

Sorted List Length	**Comparisons**	**Swaps**	**Assign smallest candidate**
0 -> 1	3	1	3
1 -> 2	2	1	2
2 -> 3	1	1	2
Total	**6**	**3**	**7**

Note that the number of comparisons and the number of swaps are independent of the contents of the list (this is true for selection sort but not necessarily for other sorting algorithms) while the number of times we have to assign a new value to the smallest candidate depends on the contents of the list.

More generally, the algorithm for selection sort is as follows:

1. Divide the list to be sorted into a sorted portion at the front (initially empty) and an unsorted portion at the end (initially the whole list).
2. Find the smallest element in the unsorted list:

1. Select the first element of the unsorted list as the initial candidate.
2. Compare the candidate to each element of the unsorted list in turn, replacing the candidate with the current element if the current element is smaller.
3. Once the end of the unsorted list is reached, the candidate is the smallest element.

3. Swap the smallest element found in the previous step with the first element in the unsorted list, thus extending the sorted list by one element.
4. Repeat the steps 2 and 3 above until only one element remains in the unsorted list.

Note

The Selection sort algorithm as described here has two properties which are often desirable in sorting algorithms.

The first is that the algorithm is in-place. This means that it uses essentially no extra storage beyond that required for the input (the unsorted list in this case). A little extra storage may be used (for example, a temporary variable to hold the candidate for the smallest element). The important property is that the extra storage required should not increase as the size of the input increases.

The second is that the sorting algorithm is stable. This means that two elements which are equal retain their initial relative ordering. This becomes important if there is additional information attached to the values being sorted (for example, if we are sorting a list of people using a comparison function that compares their dates of birth). Stable sorting algorithms ensure that sorting an already sorted list leaves the order of the list unchanged, even in the presence of elements that are treated as equal by the comparison.

Exercise 1

Complete the following code which will perform a selection sort in Python. ”...” denotes missing code that should be filled in:

```
defselection_sort(items):
"""Sorts a list of items into ascending order using the
    selection sort algoright.
    """
forstepinrange(len(items)):
# Find the location of the smallest element in
# items[step:].
location_of_smallest=step
forlocationinrange(step,len(items)):
# TODO: determine location of smallest
...
```

```
# TODO: Exchange items[step] with items[location_of_smallest]
...
```

Exercise 2

Earlier in this section we counted the number
of comparisons, swaps and assignments used in our example.

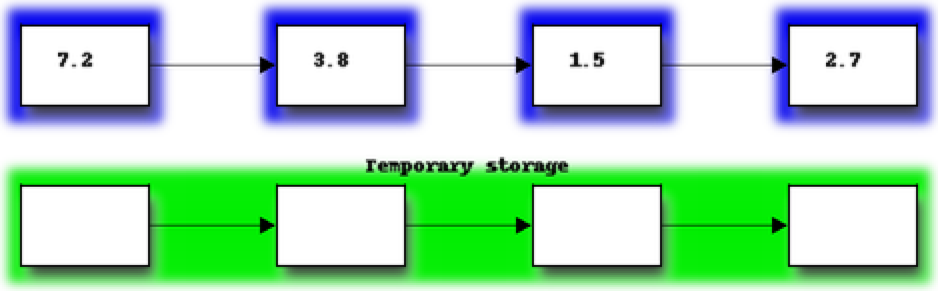
1. How many swaps are performed when we apply selection sort to a list of N items?
2. How many comparisons are performed when we apply selection sort to a list of N items?
    1. How many comparisons are performed to find the smallest element when the unsorted portion of the list has M items?
    2. Sum over all the values of M encountered when sorting the list of length N to find the total number of comparisons.
3. The number of assignments (to the candidate smallest number) performed during the search for a smallest element is at most one more than the number of comparisons. Use this to find an upper limit on the total number of assignments performed while sorting a list of length N.
4. Use the results of the previous question to find an upper bound on the total number of operations (swaps, comparisons and assignments) performed. Which term in the number of operations will dominate for large lists?

Merge sort

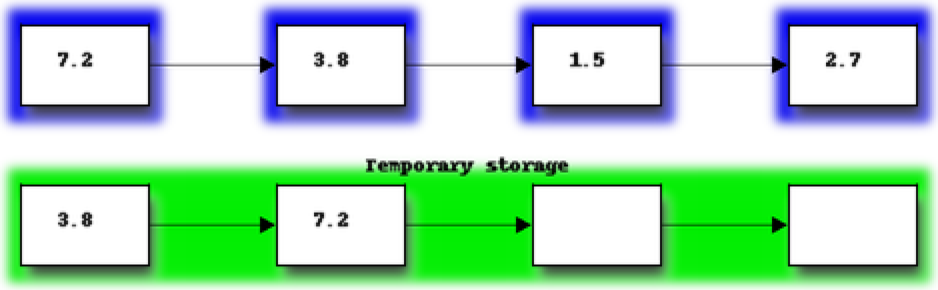
When we use merge sort to order a list, we repeatedly merge sorted sub-sections of the list – starting from sub-sections consisting of a single item each.

We will see shortly that merge sort requires significantly fewer operations than selection sort.

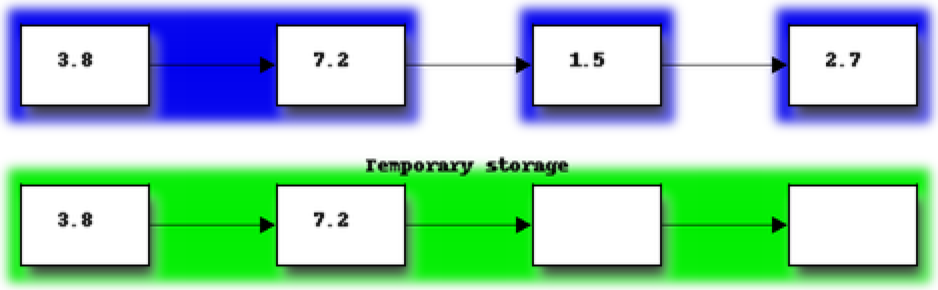
Let us start once more with our small list of four elements:



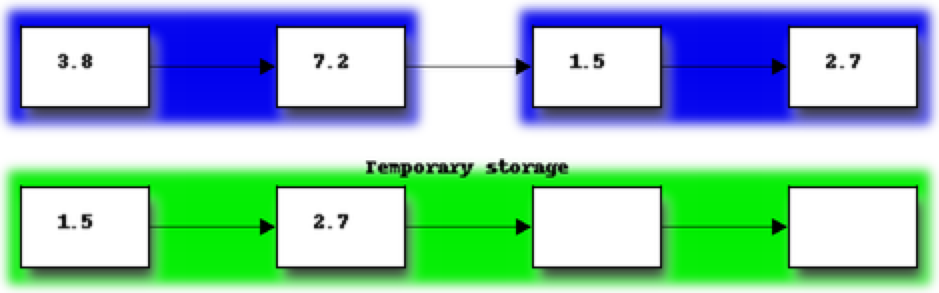
First we will merge the two sections on the left into the temporary storage. Imagine the two sections as two sorted piles of cards – we will merge the two piles by repeatedly taking the smaller of the top two cards and placing it at the end of the merged list in the temporary storage. Once one of the two piles is empty, the remaining items in the other pile can just be placed on the end of the merged list:



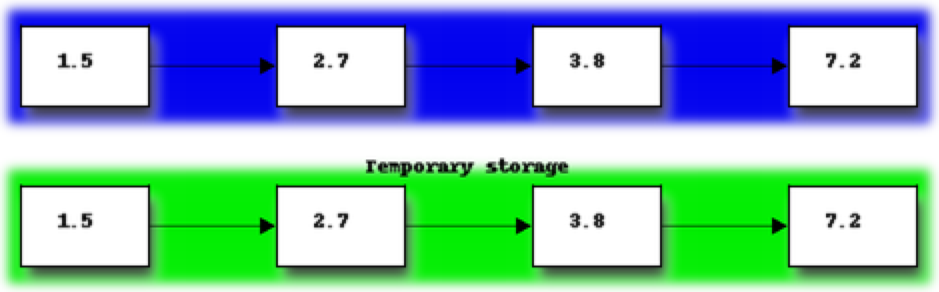
Next we copy the merged list from the temporary storage back into the portion of the list originally occupied by the merged subsections:



We repeat the procedure to merge the second pair of sorted sub-sections:



Having reached the end of the original list, we now return to the start of the list and begin to merge sorted sub-sections again. We repeat this until the entire list is a single sorted sub-section. In our example, this requires just one more merge:



Notice how the size of the sorted sections of the list doubles after every iteration of merges. After M steps the size of the sorted sections is 2^M. Once 2^M is greater than N, the entire list is sorted. Thus, for a list of size N, we need M equals $\log_2 N$ interations to sort the list.

Each iteration of merges requires a complete pass through the list and each element is copied twice – once into the temporary storage and once back into the original list. As long as there are items left in both sub-sections in each pair, each copy into the temporary list also requires a comparison to pick which item to copy. Once one of the lists runs out, no comparisons are needed. Thus each pass requires 2N copies and roughly N comparisons (and certainly no more than N).

The total number of operations required for our merge sort algorithm is the product of the number of operations in each pass and the number of passes – i.e. $2N\log_2 N$ copies and roughly $N\log_2 N$ comparisons.

The algorithm for merge sort may be written as this list of steps:

1. Create a temporary storage list which is the same size as the list to be sorted.
2. Start by treating each element of the list as a sorted one-element sub-section of the original list.
3. Move through all the sorted sub-sections, merging adjacent pairs as follows:
    1. Use two variables to point to the indices of the smallest uncopied items in the two sorted sub-sections, and a third variable to point to the index of the start of the temporary storage.
    2. Copy the smaller of the two indexed items into the indicated position in the temporary storage. Increment the index of the sub-section from which the item was copied, and the index into temporary storage.
    3. If all the items in one sub-section have been copied, copy the items remaining in the other sub-section to the back of the list in temporary storage. Otherwise return to step 3 ii.
    4. Copy the sorted list in temporary storage back over the section of the original list which was occupied by the two sub-sections that have just been merged.
4. If only a single sorted sub-section remains, the entire list is sorted and we are done. Otherwise return to the start of step 3.

Exercise 3

Write a Python function that implements merge sort. It may help to write a separate function which performs merges and call it from within your merge sort implementation.

Python's sorting algorithm

Python's default sorting algorithm, which is used by the built-in `sorted` function as well as the `sort` method of list objects, is called Timsort. It's an algorithm developed by Tim Peters in 2002 for use in Python. Timsort is a modified version of merge sort which uses insertion sort to arrange the list of items into conveniently mergeable sections.

Note

Tim Peters is also credited as the author of The Zen of Python – an attempt to summarise the early Python community's ethos in a short series of koans. You can read it by typing `import this` into the Python console.

Searching algorithms

Searching is also a common and well-studied task. This task can be described formally as follows:

Given a list of values, a function that compares two values and a desired value, find the position of the desired value in the list.

We will look at two algorithms that perform this task:

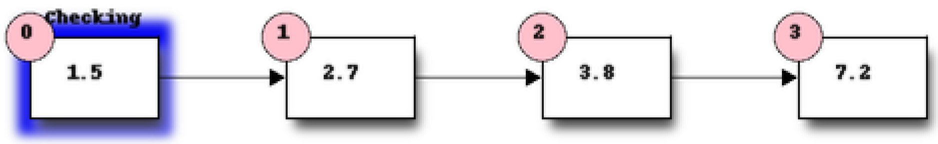
- linear search, which simply checks the values in sequence until the desired value is found
- binary search, which requires a sorted input list, and checks for the value in the middle of the list, repeatedly discarding the half of the list which contains values which are definitely either all larger or all smaller than the desired value

There are numerous other searching techniques. Often they rely on the construction of more complex data structures to facilitate repeated searching. Examples of such structures are hash tables (such as Python's dictionaries) and prefix trees. Inexact searches that find elements similar to the one being searched for are also an important topic.

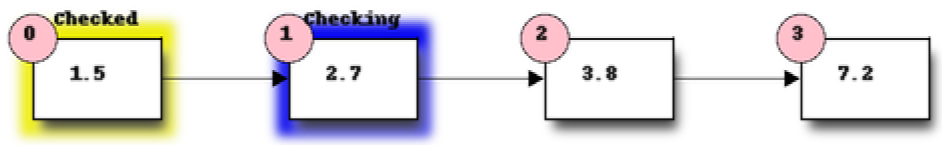
Linear search

Linear search is the most basic kind of search method. It involves checking each element of the list in turn, until the desired element is found.

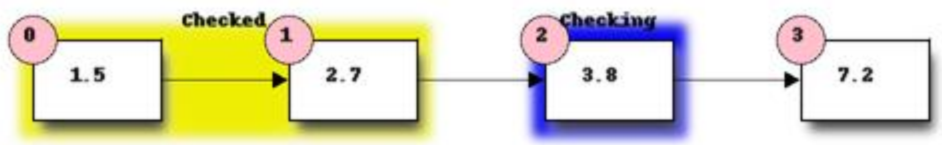
For example, suppose that we want to find the number 3.8 in the following list:



We start with the first element, and perform a comparison to see if its value is the value that we want. In this case, 1.5 is not equal to 3.8, so we move onto the next element:



We perform another comparison, and see that 2.7 is also not equal to 3.8, so we move onto the next element:



We perform another comparison and determine that we have found the correct element. Now we can end the search and return the position of the element (index 2).

We had to use a total of 3 comparisons when searching through this list of 4 elements. How many comparisons we need to perform depends on the total length of the list, but also whether the element we are looking for is near the beginning or near the end of the list. In the worst-case scenario, if our element is the last element of the list, we will have to search through the entire list to find it.

If we search the same list many times, assuming that all elements are equally likely to be searched for, we will on average have to search through half of the list each time. The cost (in comparisons) of performing linear search thus scales linearly with the length of the list.

Exercise 4

1. Write a function which implements linear search. It should take a list and an element as a parameter, and return the position of the element in the list. If the element is not

in the list, the function should raise an exception. If the element is in the list multiple times, the function should return the first position.

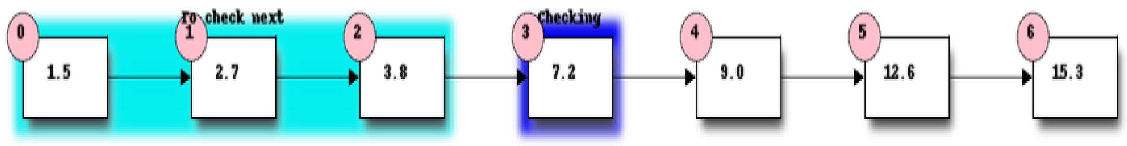
Binary search

Binary search is a more efficient search algorithm which relies on the elements in the list being sorted. We apply the same search process to progressively smaller sub-lists of the original list, starting with the whole list and approximately halving the search area every time.

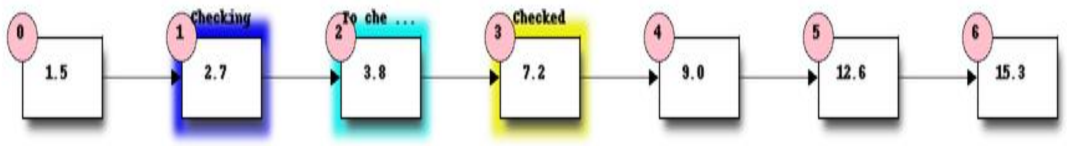
We first check the middle element in the list.

- If it is the value we want, we can stop.
- If it is higher than the value we want, we repeat the search process with the portion of the list before the middle element.
- If it is lower than the value we want, we repeat the search process with the portion of the list after the middle element.

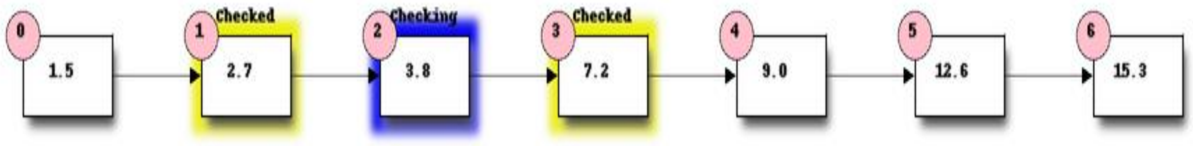
For example, suppose that we want to find the value 3.8 in the following list of 7 elements:



First we compare the element in the middle of the list to our value. 7.2 is bigger than 3.8, so we need to check the first half of the list next.



Now the first half of the list is our new list to search. We compare the element in the middle of this list to our value. 2.7 is smaller than 3.8, so we need to search the second half of this sublist next.



The second half of the last sub-list is just a single element, which is also the middle element. We compare this element to our value, and it is the element that we want.

We have performed 3 comparisons in total when searching this list of 7 items. The number of comparisons we need to perform scales with the size of the list, but much more slowly than for linear search – if we are searching a list of length N, the maximum number of comparisons that we will have to perform is $\log_2 N$.

Path Length

Whenever we describe an object's motion, the first thing we talk about is the position of the object. Where is the object? It is some distance from a zero point (the point we call the origin) in a particular direction. The change in the position is what we call displacement. Let us study more about it below.

Distance traveled by a body is the path length. For example, if a body covers half the circumference of a circle of radius r the distance traveled is $d = \pi r$. It is a scalar quantity.

Calculating Distance in One-dimensional Motion

Total distance traveled in one-dimension can be found by adding the path lengths for all parts of motion. Note that every path length is greater than 0. Athletes race in a straight track of length 200 m and return back. The total distance traveled by each athlete is $200 \times 2 = 400$ m

Browse more Topics under Motion In A Straight Line

- Average Velocity and Average Speed

- Instantaneous Velocity and Speed
- Relative Velocity
- Acceleration
- Kinematics Equations for Uniformly Accelerated Motion

Displacement Definition

Displacement of the object is equal to the length of the shortest path between the final and the initial points. Its direction is from the initial point to the final point. It is a vector quantity. For example, if a body moves along a circle of radius r and covers half the circumference, then displacement is given by $s=2r$.

In one-dimensional motion displacement of the object will be the shortest distance between final and initial point. For example, displacement of a particle in a circular motion would be zero when it reaches the starting point.

Displacement Formula

Displacement = final position – initial position

$D = X_f - X_i$

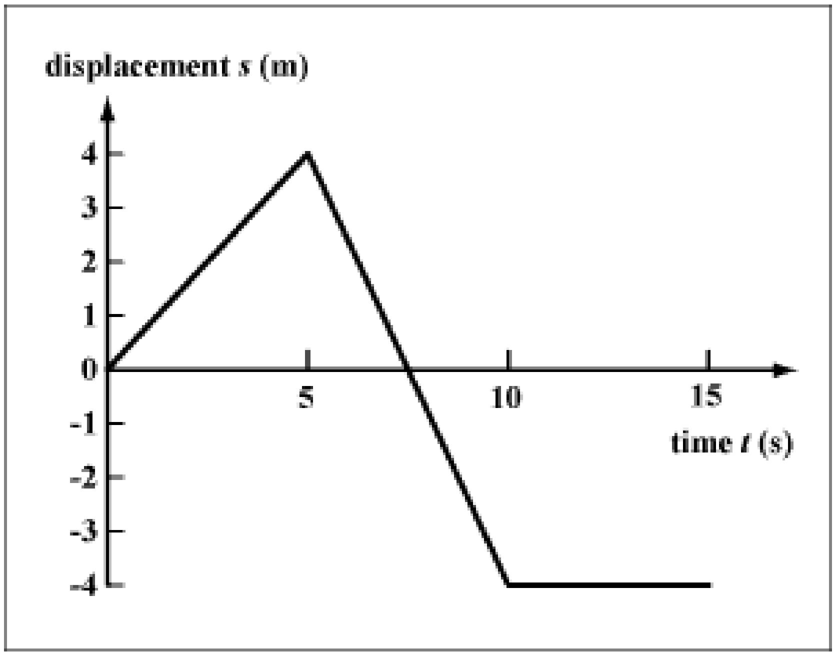
D = displacement

X_f = final position

X_i = initial position

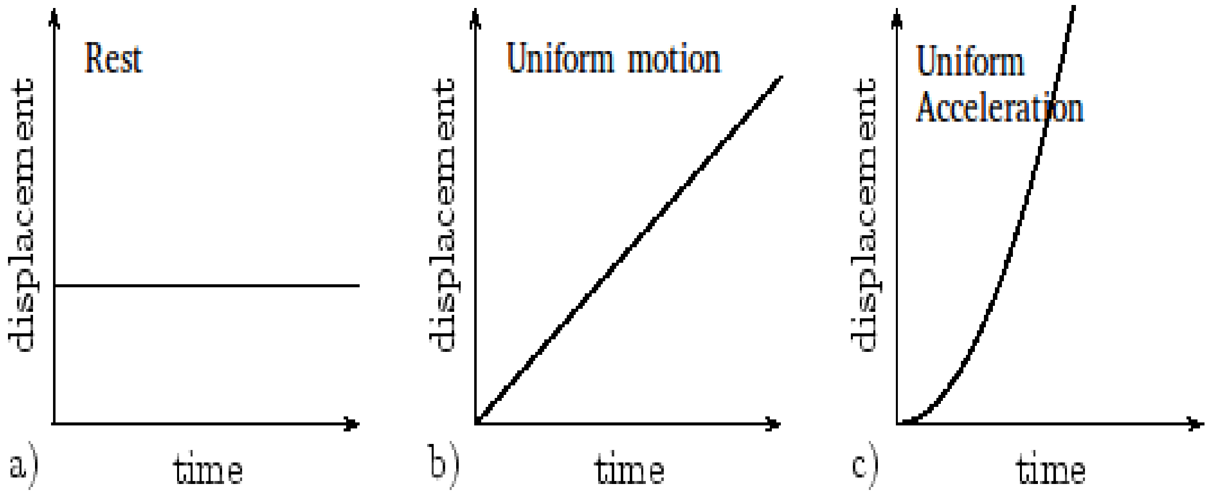
ΔX = short form for change in position

Displacement-time graph



For above graph note that displacement can be both positive and negative. Also since it is a vector, the graph is drawn for one-dimension of motion only.

Displacement-time Graph for Rest, Uniform motion and Uniform Acceleration



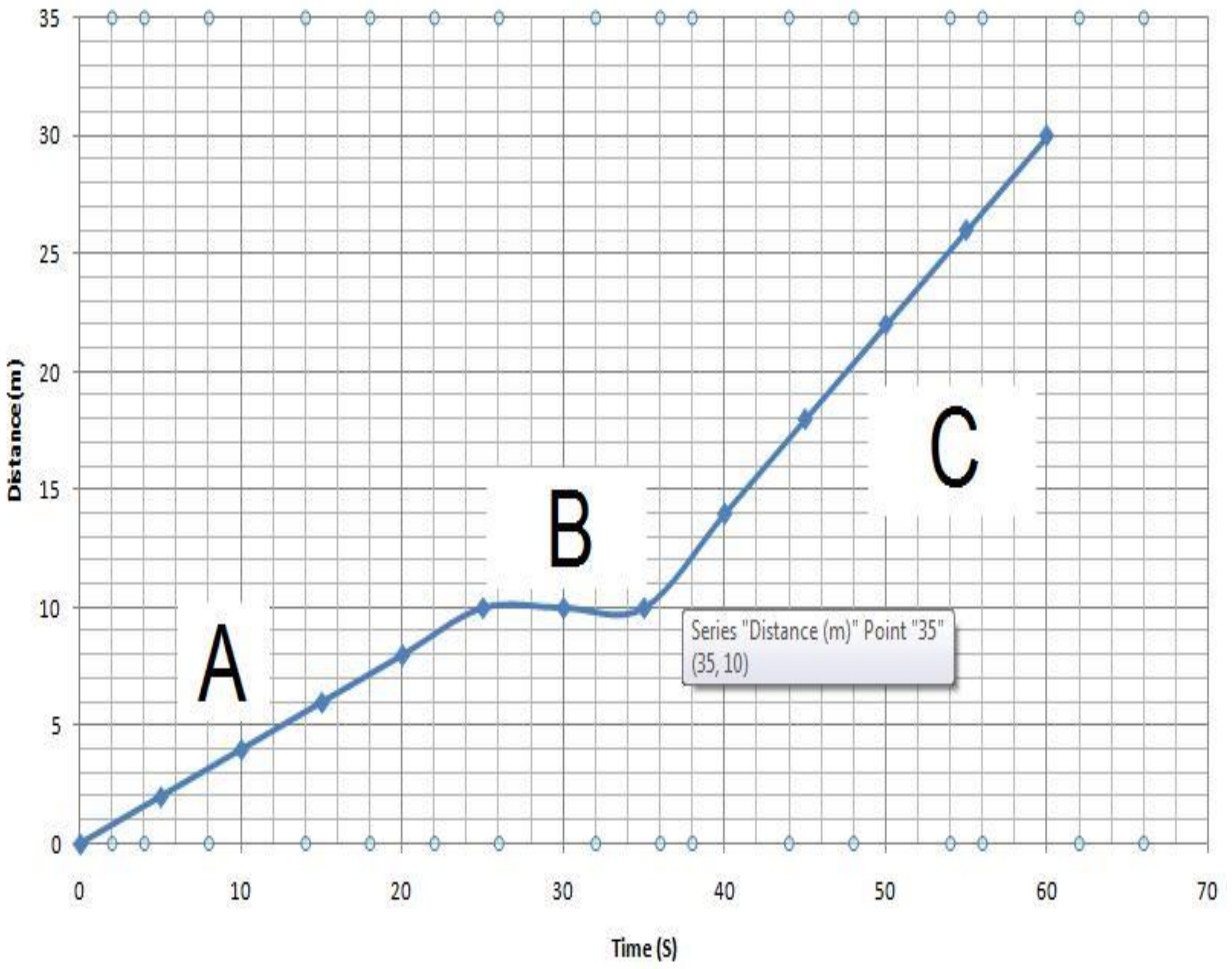
The graph for rest is a straight line with zero slopes. For uniform motion, the graph is a straight line with the non-zero slope. In case of a uniform acceleration, the graph is a parabola.

Relative Displacement

It is the displacement of a point on a structure with respect to its original location or an adjacent point on the structure that has also undergone movement, can be an effective indicator of post-event structural damage.

Mathematically it is →r=→r1−→r2r→=r→1−r→2

Distance-time graph



A distance-time graph is a graph of distance v/s time. It only lies in the first quadrant as the distance is always positive. Also, it is increasing in nature. The attached plot shows a distance-time graph.

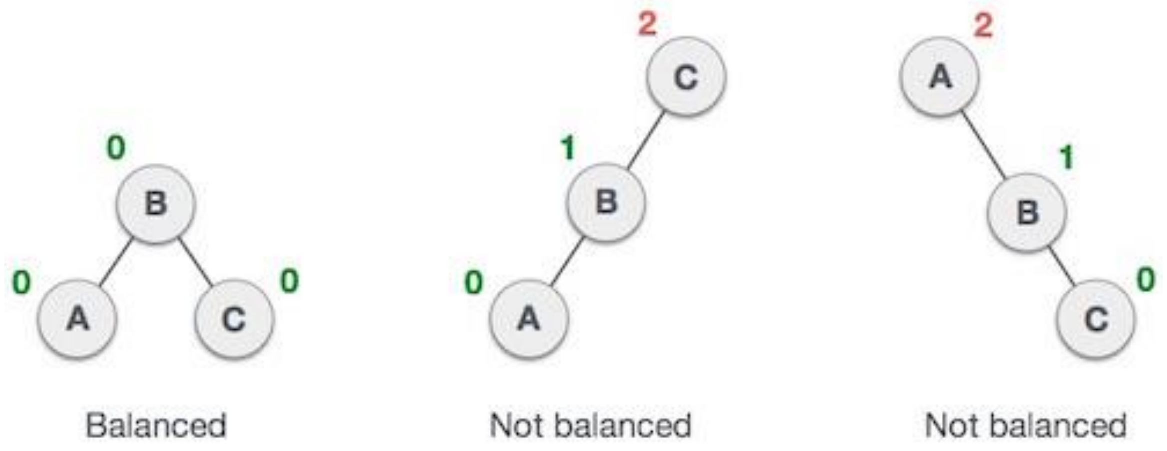
AVL Trees

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −

It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced −



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height(left-sutree) − height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

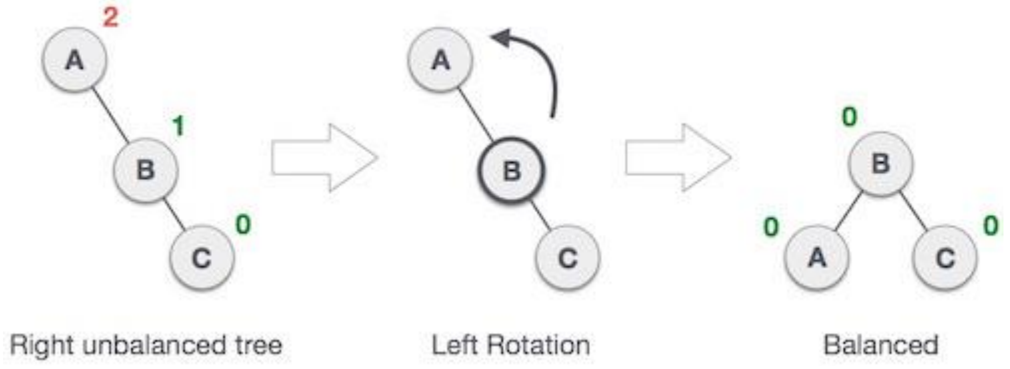
To balance itself, an AVL tree may perform the following four kinds of rotations −

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

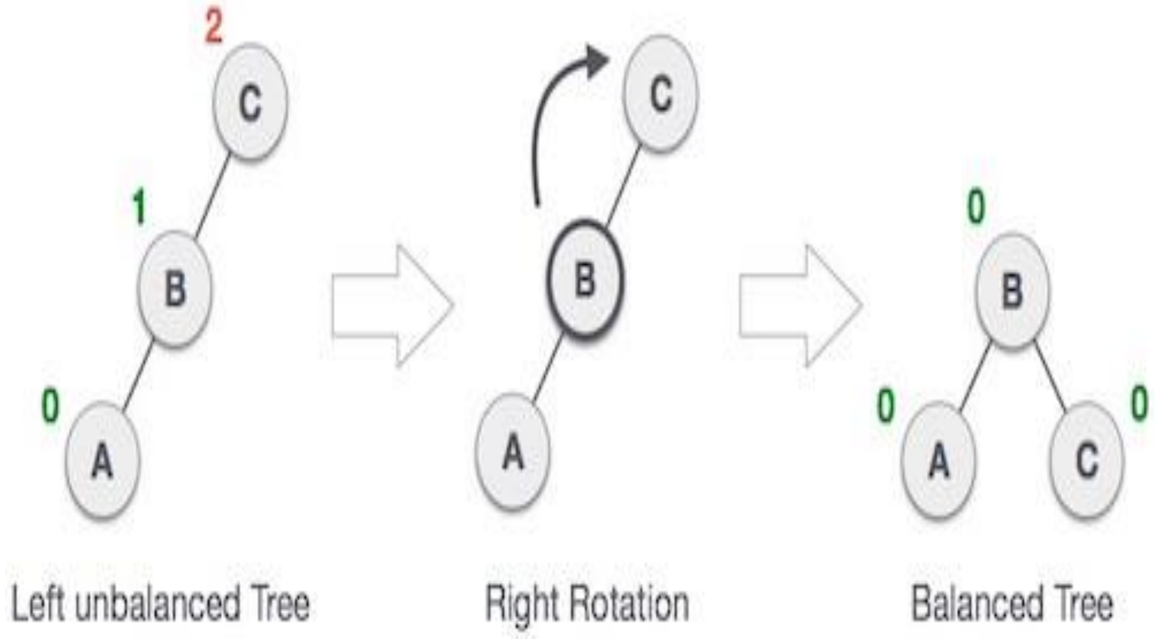
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

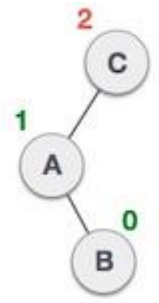
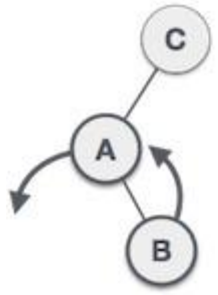
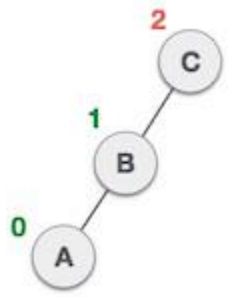
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

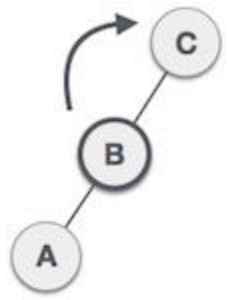
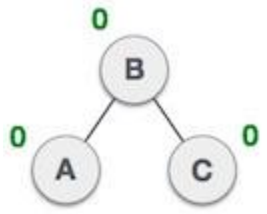


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

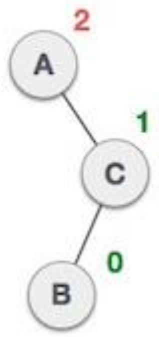
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

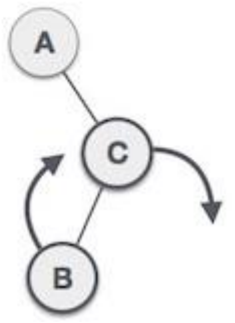
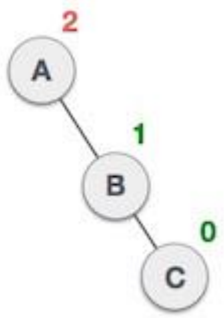
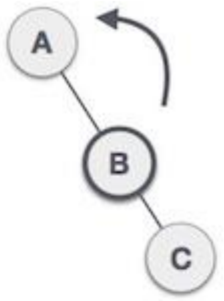
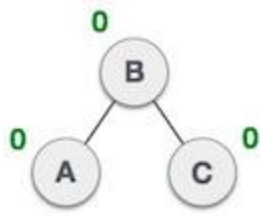
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.

	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.
	The tree is now balanced.

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.

	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

B-trees

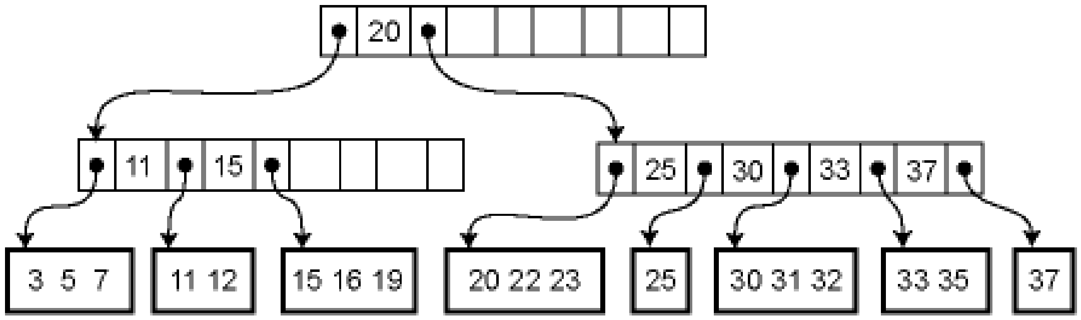
The idea we saw earlier of putting multiple set (list, hash table) elements together into large chunks that exploit locality can also be applied to trees. Binary search trees are not good for locality because a given node of the binary tree probably occupies only a fraction of any cache line. **B-trees** are a way to get better locality by putting multiple elements into each tree node.

B-trees were originally invented for storing data structures on disk, where locality is even more crucial than with memory. Accessing a disk location takes about 5ms = 5,000,000ns. Therefore, if you are storing a tree on disk, you want to make sure that a given disk read is as effective as possible. B-trees have a high branching factor, much larger than 2, which ensures that few disk reads are needed to navigate to the place where data is stored. B-trees may also useful for in-memory data structures because these days main memory is almost as slow relative to the processor as disk drives were to main memory when B-trees were first introduced!

A B-tree of order *m* is a search tree in which each nonleaf node has up to *m* children. The actual elements of the collection are stored in the leaves of the tree, and the nonleaf nodes contain only keys. Each leaf stores some number of elements; the maximum number may be greater or (typically) less than *m*. The data structure satisfies several invariants:

1. Every path from the root to a leaf has the same length
2. If a node has *n* children, it contains *n*−1 keys.
3. Every node (except the root) is at least half full
4. The elements stored in a given subtree all have keys that are between the keys in the parent node on either side of the subtree pointer. (This generalizes the BST invariant.)
5. The root has at least two children if it is not a leaf.

For example, the following is an order-5 B-tree (*m*=5) where the leaves have enough space to store up to 3 data records:

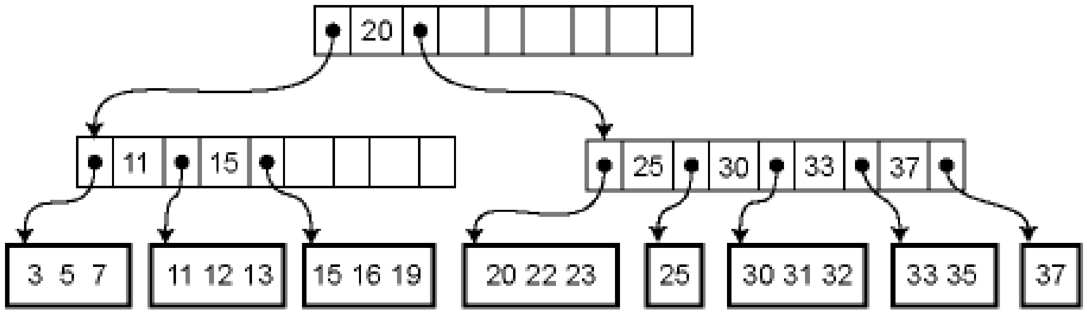


Because the height of the tree is uniformly the same and every node is at least half full, we are guaranteed that the asymptotic performance is O(lg *n*) where *n* is the size of the collection. The real win is in the constant factors, of course. We can choose *m* so that the pointers to the *m* children plus the *m*−1 elements fill out a cache line at the highest level of the memory hierarchy where we can expect to get cache hits. For example, if we are accessing a large disk database then our "cache lines" are memory blocks of the size that is read from disk.

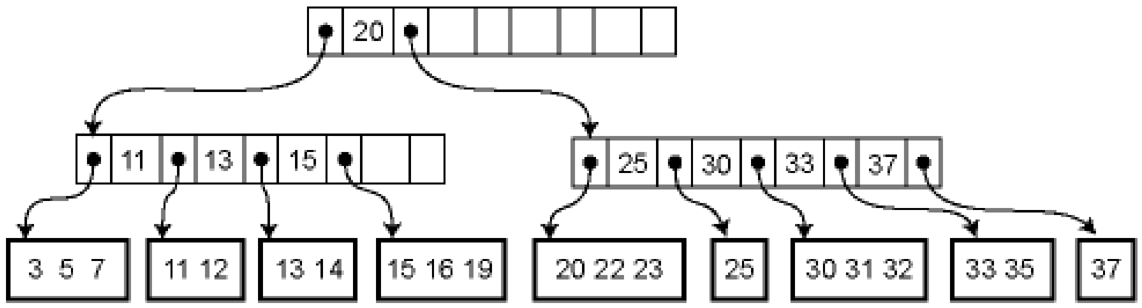
Lookup in a B-tree is straightforward. Given a node to start from, we use a simple linear or binary search to find whether the desired element is in the node, or if not, which child pointer to follow from the current node.

Insertion and deletion from a B-tree are more complicated; in fact, they are notoriously difficult to implement correctly. For insertion, we first find the appropriate leaf node into which the inserted element falls (assuming it is not already in the tree). If there is already room in the node, the new element can be inserted simply. Otherwise the current leaf is already full and must be split into two leaves, one of which acquires the new element. The parent is then updated to contain a new key and child pointer. If the parent is already full, the process ripples upwards, eventually possibly reaching the root. If the root is split into two, then a new root is created with just two children, increasing the height of the tree by one.

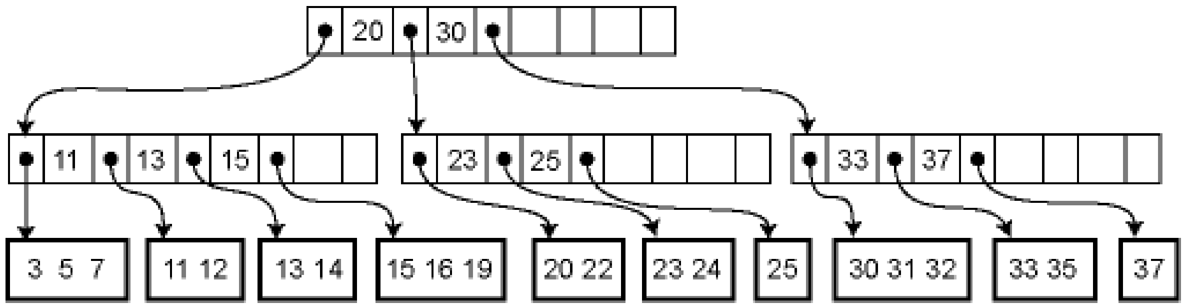
For example, here is the effect of a series of insertions. The first insertion (13) merely affects a leaf. The second insertion (14) overflows the leaf and adds a key to an internal node. The third insertion propagates all the way to the root.



insert(13)



insert(13); insert(14)



insert(13); insert(14); insert(24)

Deletion works in the opposite way: the element is removed from the leaf. If the leaf becomes empty, a key is removed from the parent node. If that breaks invariant 3, the keys of the parent node and its immediate right (or left) sibling are reapportioned among them so that invariant 3 is satisfied. If this is not possible, the parent node can be combined with that sibling, removing a key another level up in the tree and possible causing a ripple all the way to the root. If the root has just two children, and they are combined, then the root is deleted and the new combined node becomes the root of the tree, reducing the height of the tree by one.

Unit - V

Graphs:

Terminology & Representations

Overview

- Definitions: Vertices, edges, paths, etc
- Representations: Adjacency list and adjacency matrix

Definitions: Graph, Vertices, Edges

- Define a graph G = (V, E) by defining a pair of sets:
    1. V = a set of **vertices**
    2. E = a set of **edges**

- Edges:
    - Each edge is defined by a pair of vertices
    - An edge **connects** the vertices that define it
    - In some cases, the vertices can be the same

  Vertices:
    - Vertices also called **nodes**
    - Denote vertices with labels

  Representation:
    - Represent vertices with circles, perhaps containing a label
    - Represent edges with lines between circles

  Example:
    - V = {A,B,C,D}
    - E = {(A,B),(A,C),(A,D),(B,D),(C,D)}

Motivation

- Many algorithms use a graph representation to represent data or the problem to be solved

- Examples:
    - Cities with distances between
    - Roads with distances between intersection points
    - Course prerequisites
    - Network
    - Social networks
    - Program call graph and variable dependency graph

Graph Classifications

- There are seveal common kinds of graphs
    - Weighted or unweighted
    - Directed or undirected
    - Cyclic or acyclic
- Choose the kind required for problem and determined by data
- We examine each below

Kinds of Graphs: Weighted and Unweighted

- Graphs can be classified by whether or not their edges have **weights**
- **Weighted graph**: edges have a weight
    - Weight typically shows cost of traversing
    - Example: weights are distances between cities
- **Unweighted graph**: edges have no weight
    - Edges simply show connections
    - Example: course prereqs

Kinds of Graphs: Directed and Undirected

- Graphs can be classified by whether or their edges are have direction

- **Undirected Graphs**: each edge can be traversed in **either direction**
- **Directed Graphs**: each edge can be traversed **only in a specified direction**

Undirected Graphs

- **Undirected Graph**: no implied direction on edge between nodes
    - The example from above is an undirected graph
    - In diagrams, edges have no direction (ie they are not arrows)
    - Can traverse edges in either directions
- In an undirected graph, an edge is an **unordered** pair
    - Actually, an edge is a set of 2 nodes, but for simplicity we write it with parens
        - For example, we write (A, B) instead of {A, B}
        - Thus, (A,B) = (B,A), etc
        - If (A,B) ∈ E then (B,A) ∈ E
    - Formally: ∀ u,v ∈ E, (u,v)=(v,u) and u ≠ v
- A node normally does not have an edge to itself

Directed Graphs

- **Digraph**: A graph whose edges are directed (ie have a direction)
    - Edge drawn as arrow
    - Edge can only be traversed in direction of arrow
    - Example: E = {(A,B), (A,C), (A,D), (B,C), (D,C)}
    - Examples: courses and prerequisites, program call graph

- In a digraph, an edge is an **ordered** pair
  - Thus: (u,v) and (v,u) are not the same edge
  - In the example, (D,C) ∈ E, (C,D) ∉ E
  - What would edge (B,A) look like? Remember (A,B) ≠ (B,A)

- A node can have an edge to itself (eg (A,A) is valid)

Subgraph

- If graph G=(V, E)
  - Then Graph G'=(V',E') is a **subgraph** of G if V' ⊆ V and E' ⊆ E and

Example ...

Degree of a Node

- The **degree** of a node is the number of edges the node is used to define

- In the example above:
  - Degree 2: B and C
  - Degree 3: A and D
  - A and D have **odd degree**, and B and C have **even degree**
- Can also define **in-degree** and **out-degree**
  - In-degree: Number of edges pointing **to** a node
  - Out-degree: Number of edges pointing **from** a node
  - Whare are the in- and out-degree of the example?

Graphs: Terminology Involving Paths

- **Path**: sequence of vertices in which each pair of successive vertices is connected by an edge
- **Cycle**: a path that starts and ends on the same vertex
- **Simple path**: a path that does not cross itself
  - That is, no vertex is repeated (except first and last)
  - Simple paths cannot contain cycles
- **Length** of a path: Number of edges in the path
  - Sometimes the sum of the weights of the edges

Examples

- A sequence of vertices: (A, B, C, D) [Is this path, simple path, cycle?]
- (A, B, D, A, C) [path, simple path, cycle?]
- (A, B, D, A, C) [path, simple path, cycle?]
- Cycle: ?
- Simple Cycle: ?
- Lengths?

Cyclic and Acyclic Graphs

- A **Cyclic** graph contains cycles
  - Example: roads (normally)
- An **acyclic** graph contains no cycles
  - Example: Course prereqs!
- Examples - Are these cyclic or acyclic?

Connected and Unconnected Graphs and Connected Components

- An undirected graph is **connected** if every pair of vertices has a path between it
    - Otherwise it is unconnected
    - Give an example of a connected graph

- An unconnected graph can be broken in to **connected components**

- A directed graph is **strongly connected** if every pair of vertices has a path between them, in **both directions**

Trees and Minimum Spanning Trees

- Tree: undirected, connected graph with no cycles
    - Example ...
    - If G=(V, E) is a tree, how many edges in G?

- Spanning tree: a **spanning tree** of G is a connected subgraph of G that is a tree
    - Example ...

- **Minimum spanning tree** (MST): a spanning tree with minimum weight
    - Example ...

- Spanning trees and minimum spanning tree are not necessarily unique

- We will look at two famous MST algorithms: Prim's and Kruskal's

Data Structures for Representing Graphs

- Two common data structures for representing graphs:
    - Adjacency lists
    - Adjacency matrix

Adjacency List Representation

- Each node has a list of adjacent nodes

- Example (undirected graph):
    - A: B, C, D
    - B: A, D
    - C: A, D
    - D: A, B, C

- Example (directed graph):
    - A: B, C, D

    - B: D
    - C: Nil
    - D: C

- Weighted graph can store weights in list

- Space: Θ(V + E) (ie |V| + |E|)

- Time:
    - To visit each node that is adjacent to node u: Θ(degree(u))
    - To determine if node u is adjacent to node v: Θ(degree(u))

Adjacency Matrix Representation

- **Adjacency Matrix**: 2D array containing weights on edges

    - Row for each vertex

    - Column for each vertex

    - Entries contain weight of edge from row vertex to column vertex

    - Entries contain ∞ (ie Integer'last) if no edge from row vertex to column vertex

    - Entries contain 0 on diagonal (if self edges not allowed)

- Example undirected graph (assume self-edges not allowed):

$$\begin{array}{ccccc} & A & B & C & D \\ A & 0 & 1 & 1 & 1 \\ B & 1 & 0 & \infty & 1 \\ C & 1 & \infty & 0 & 1 \\ D & 1 & 1 & 1 & 0 \end{array}$$

- Example directed graph (assume self-edges allowed):

$$\begin{array}{ccccc} & A & B & C & D \\ A & \infty & 1 & 1 & 1 \\ B & \infty & \infty & \infty & 1 \\ C & \infty & \infty & \infty & \infty \\ D & \infty & \infty & 1 & \infty \end{array}$$

- Can store weights in cells
- Space: $\Theta(V^2)$
- Time:
  - To visit each node that is adjacent to node u: Θ(V)
  - To determine if node u is adjacent to node v: Θ(1)

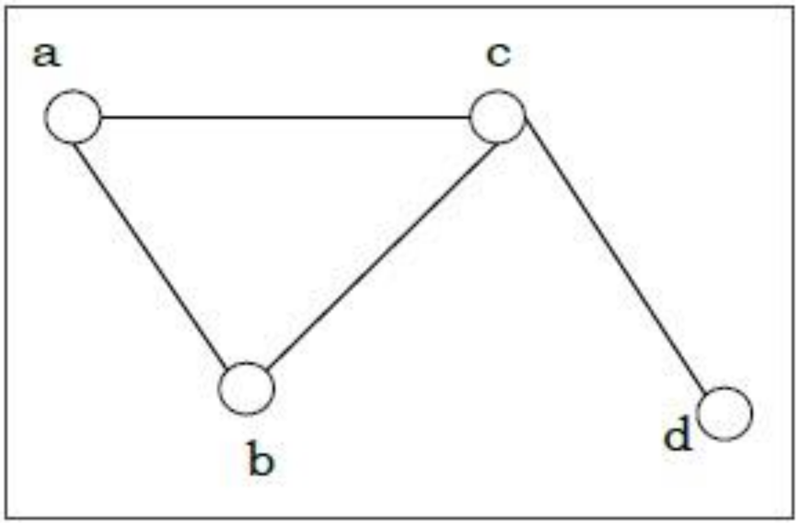
Graphs & Multi-graphs

The previous part brought forth the different tools for reasoning, proofing and problem solving. In this part, we will study the discrete structures that form the basis of formulating many a real-life problem.

The two discrete structures that we will cover are graphs and trees. A graph is a set of points, called nodes or vertices, which are interconnected by a set of lines called edges. The study of graphs, or **graph theory** is an important part of a number of disciplines in the fields of mathematics, engineering and computer science.

What is a Graph?

Definition − A graph (denoted as G=(V,E)G=(V,E)) consists of a non-empty set of vertices or nodes V and a set of edges E.
Example − Let us consider, a Graph is G=(V,E)G=(V,E) where V={a,b,c,d}V={a,b,c,d} and E={{a,b},{a,c},{b,c},{c,d}}E={{a,b},{a,c},{b,c},{c,d}}



Degree of a Vertex − The degree of a vertex V of a graph G (denoted by deg (V)) is the number of edges incident with the vertex V.

Vertex	Degree	Even / Odd
a	2	even
b	2	even
c	3	odd
d	1	odd

Even and Odd Vertex − If the degree of a vertex is even, the vertex is called an even vertex and if the degree of a vertex is odd, the vertex is called an odd vertex.

Degree of a Graph − The degree of a graph is the largest vertex degree of that graph. For the above graph the degree of the graph is 3.

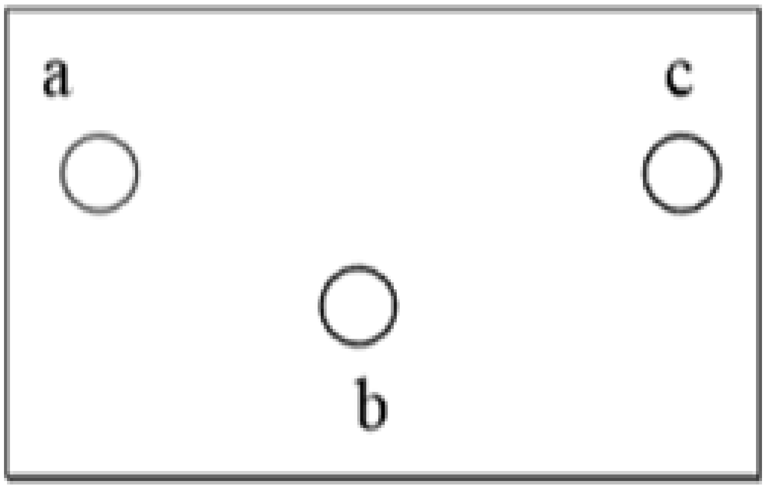
The Handshaking Lemma − In a graph, the sum of all the degrees of all the vertices is equal to twice the number of edges.

Types of Graphs

There are different types of graphs, which we will learn in the following section.

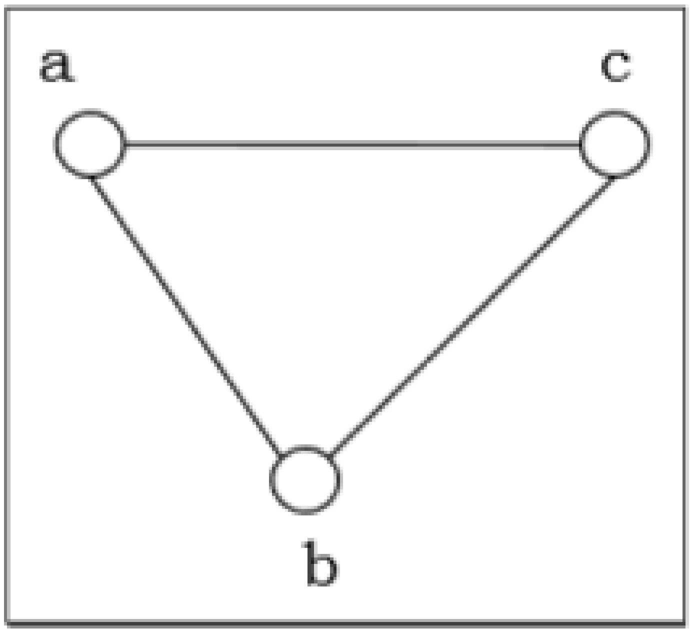
Null Graph

A null graph has no edges. The null graph of nn vertices is denoted by NnNn



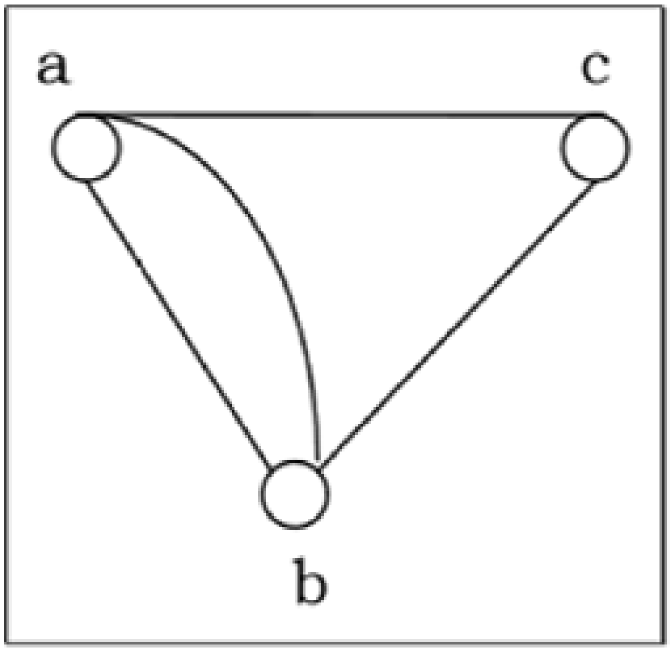
Simple Graph

A graph is called simple graph/strict graph if the graph is undirected and does not contain any loops or multiple edges.



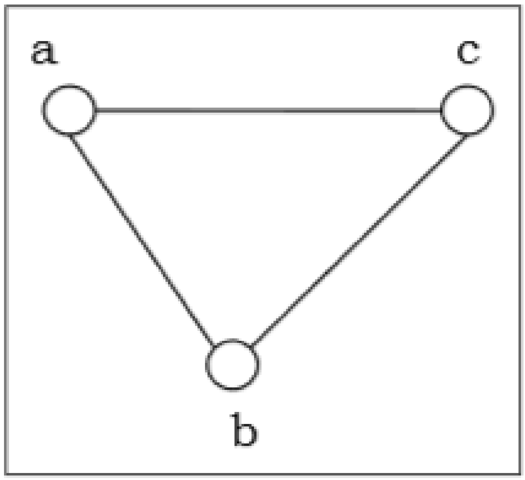
Multi-Graph

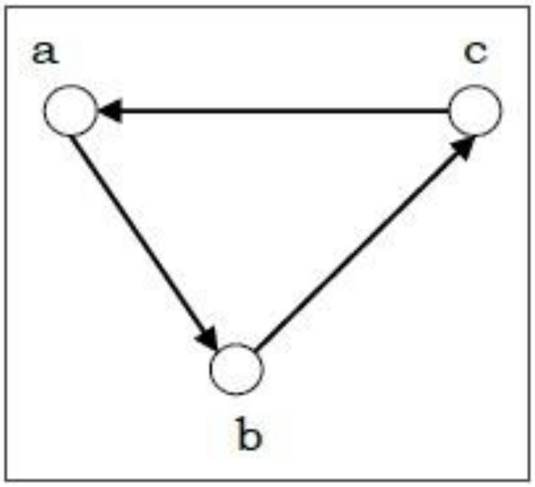
If in a graph multiple edges between the same set of vertices are allowed, it is called Multigraph. In other words, it is a graph having at least one loop or multiple edges.



Directed and Undirected Graph

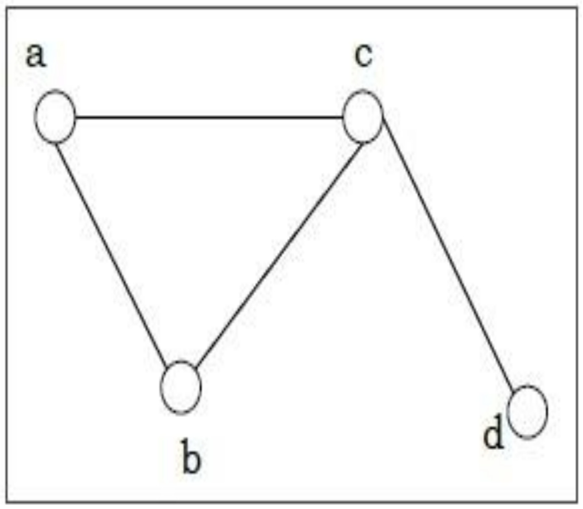
A graph G=(V,E)G=(V,E) is called a directed graph if the edge set is made of ordered vertex pair and a graph is called undirected if the edge set is made of unordered vertex pair.

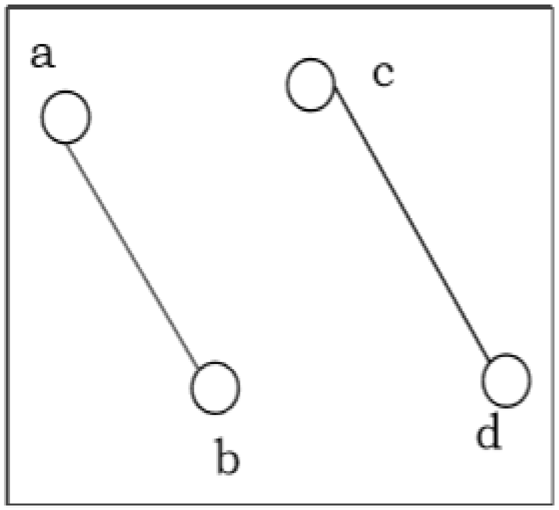




Connected and Disconnected Graph

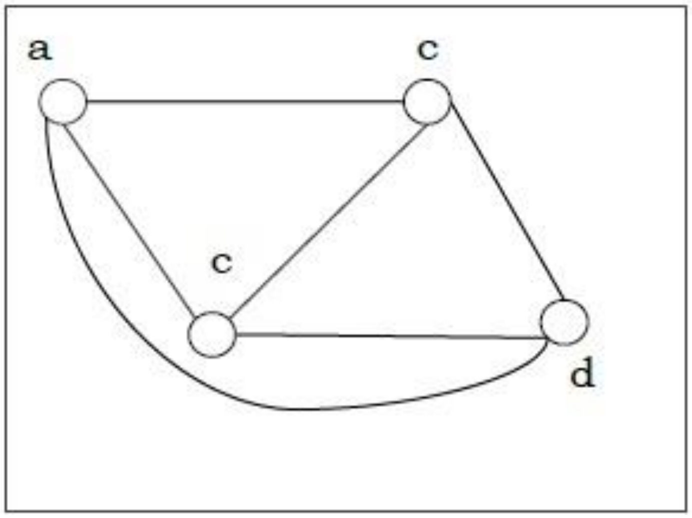
A graph is connected if any two vertices of the graph are connected by a path; while a graph is disconnected if at least two vertices of the graph are not connected by a path. If a graph G is disconnected, then every maximal connected subgraph of GG is called a connected component of the graph GG.





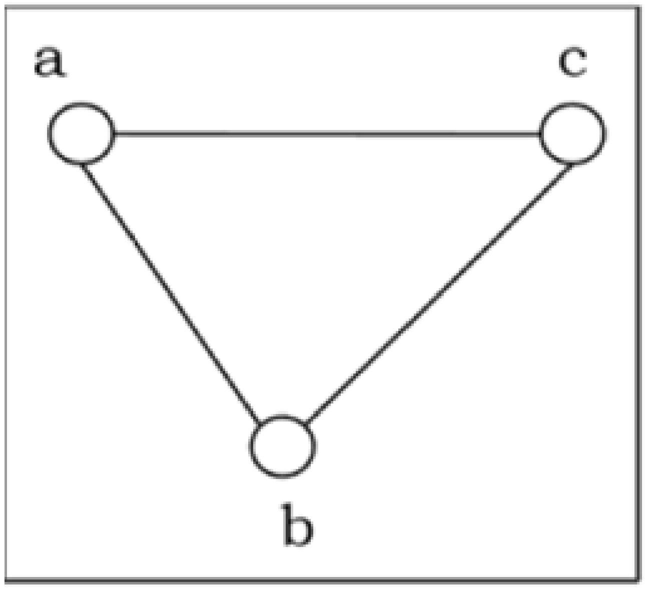
Regular Graph

A graph is regular if all the vertices of the graph have the same degree. In a regular graph G of degree rr, the degree of each vertex of GG is r.



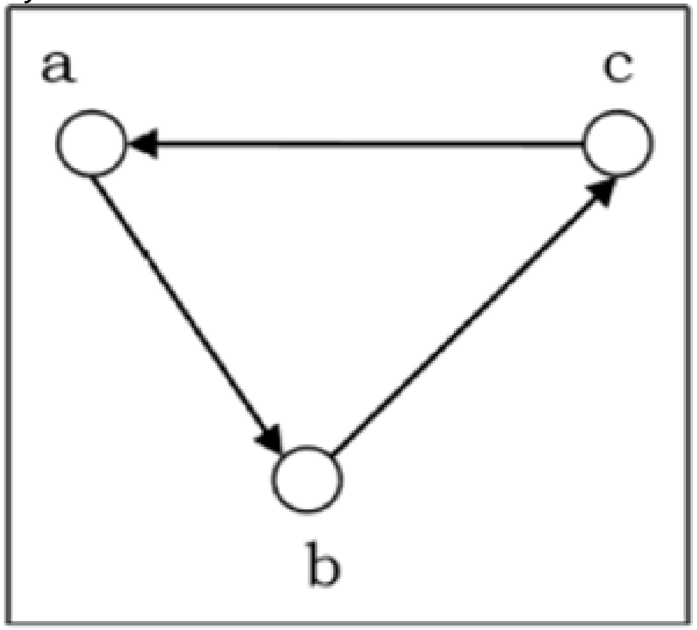
Complete Graph

A graph is called complete graph if every two vertices pair are joined by exactly one edge. The complete graph with n vertices is denoted by KnKn



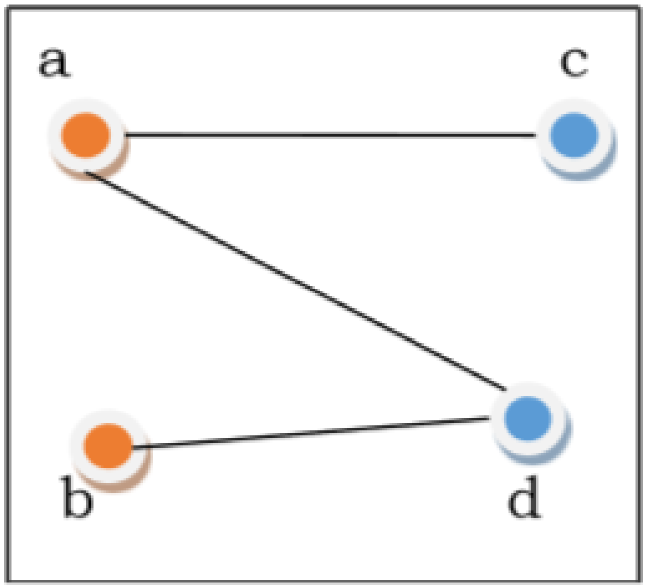
Cycle Graph

If a graph consists of a single cycle, it is called cycle graph. The cycle graph with n vertices is denoted by CnCn



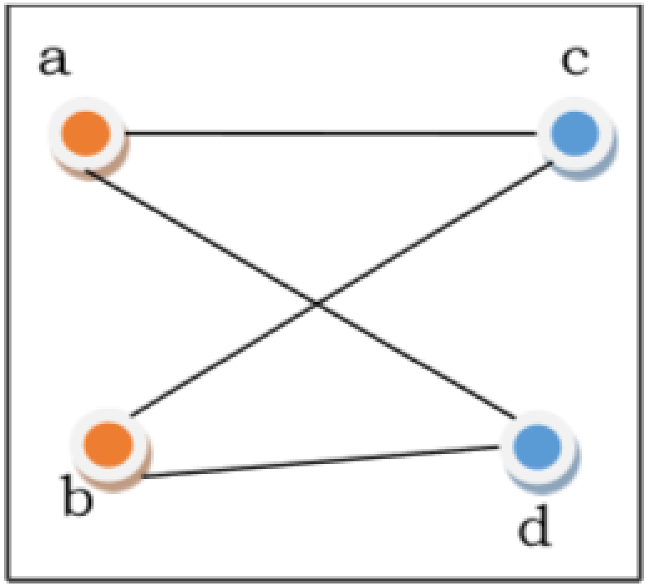
Bipartite Graph

If the vertex-set of a graph G can be split into two disjoint sets, V1V1 and V2V2, in such a way that each edge in the graph joins a vertex in V1V1 to a vertex in V2V2, and there are no edges in G that connect two vertices in V1V1 or two vertices in V2V2, then the graph GG is called a bipartite graph.



Complete Bipartite Graph

A complete bipartite graph is a bipartite graph in which each vertex in the first set is joined to every single vertex in the second set. The complete bipartite graph is denoted by Kx,yKx,y where the graph GG contains xx vertices in the first set and yy vertices in the second set.



Representation of Graphs

There are mainly two ways to represent a graph −

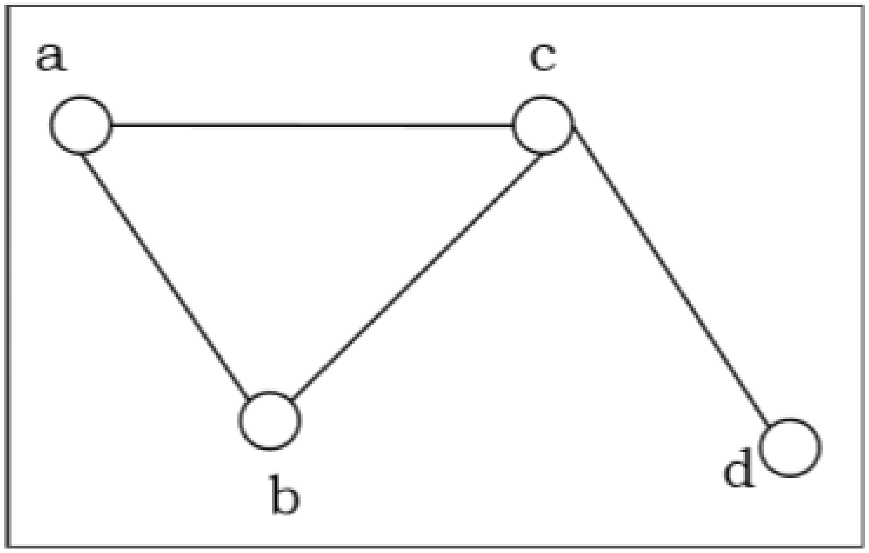
- Adjacency Matrix
- Adjacency List

Adjacency Matrix

An Adjacency Matrix A[V][V]A[V][V] is a 2D array of size V×VV×V where VV is the number of vertices in a undirected graph. If there is an edge between VxVx to VyVy then the value of A[Vx][Vy]=1A[Vx][Vy]=1 and A[Vy][Vx]=1A[Vy][Vx]=1, otherwise the value will be zero. And for a directed graph, if there is an edge between VxVx to VyVy, then the value of A[Vx][Vy]=1A[Vx][Vy]=1, otherwise the value will be zero.

Adjacency Matrix of an Undirected Graph

Let us consider the following undirected graph and construct the adjacency matrix −

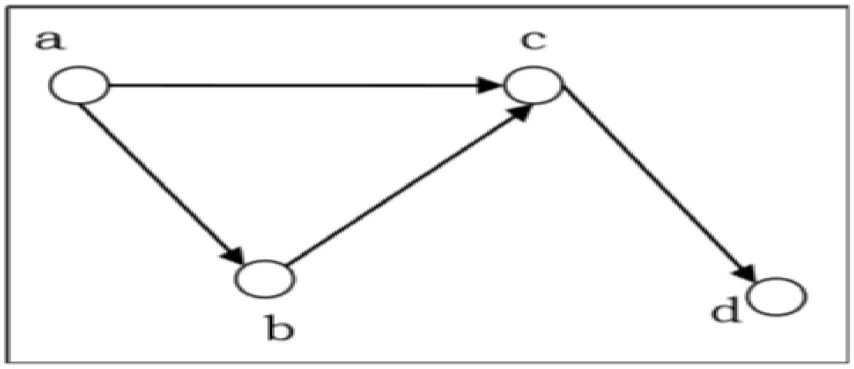


Adjacency matrix of the above undirected graph will be −

	a	**b**	**c**	**d**
a	0	1	1	0
b	1	0	1	0
c	1	1	0	1
d	0	0	1	0

Adjacency Matrix of a Directed Graph

Let us consider the following directed graph and construct its adjacency matrix −

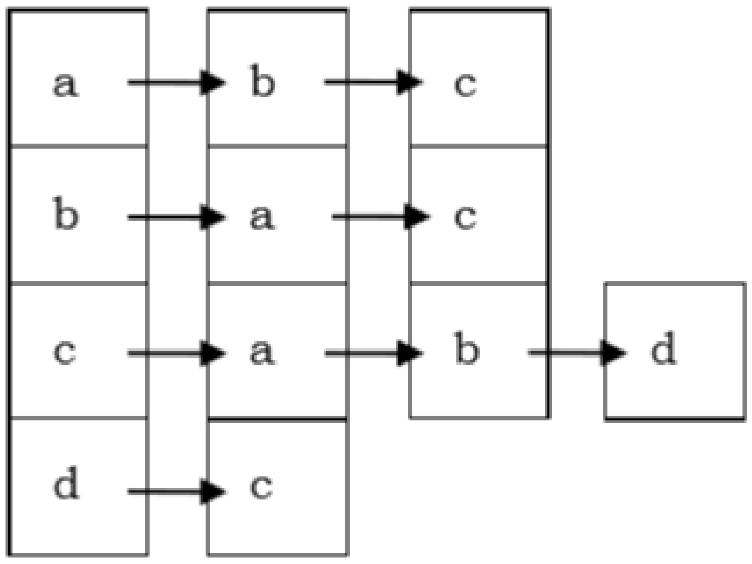


Adjacency matrix of the above directed graph will be −

	a	**b**	**c**	**d**
a	0	1	1	0
b	0	0	1	0
c	0	0	0	1
d	0	0	0	0

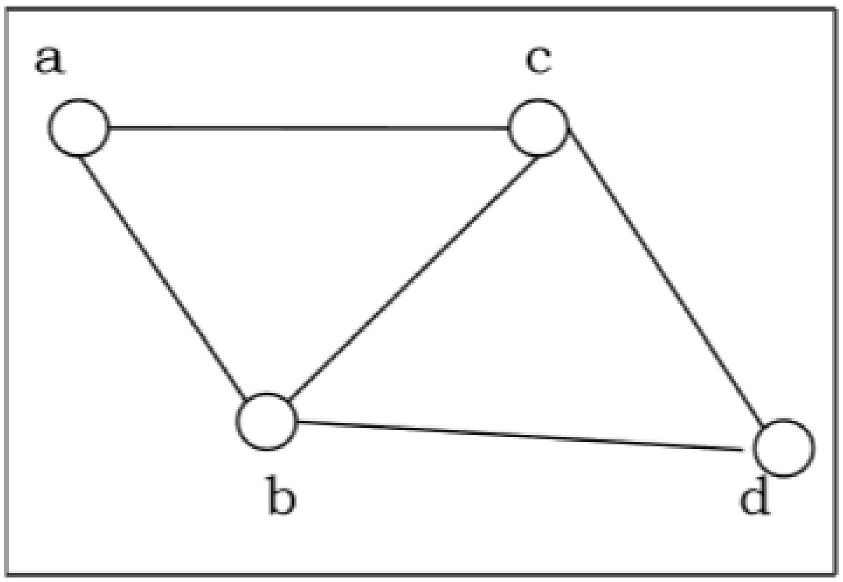
Adjacency List

In adjacency list, an array (A[V])(A[V]) of linked lists is used to represent the graph G with VV number of vertices. An entry A[Vx]A[Vx] represents the linked list of vertices adjacent to the Vx−thVx−th vertex. The adjacency list of the undirected graph is as shown in the figure below –

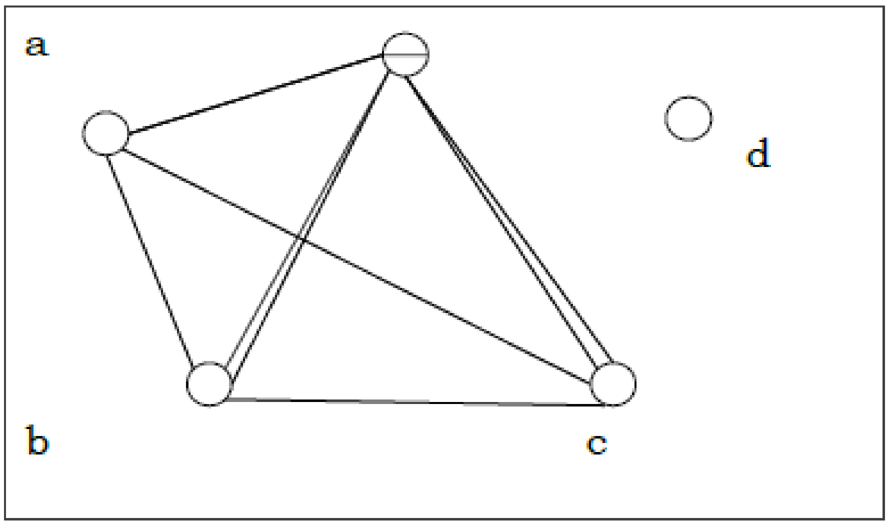


Planar vs. Non-planar graph

Planar graph − A graph GG is called a planar graph if it can be drawn in a plane without any edges crossed. If we draw graph in the plane without edge crossing, it is called embedding the graph in the plane.



Non-planar graph − A graph is non-planar if it cannot be drawn in a plane without graph edges crossing.



Isomorphism

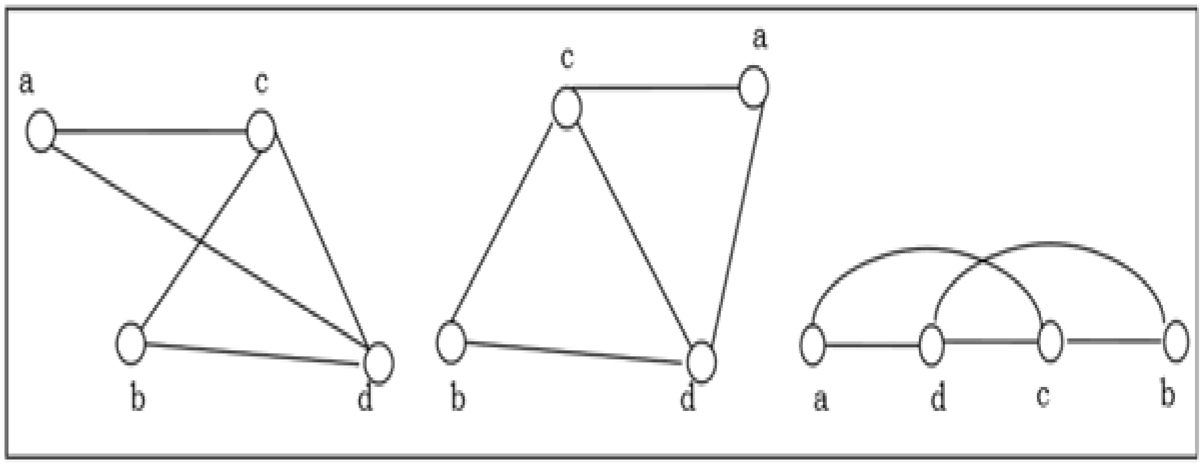
If two graphs G and H contain the same number of vertices connected in the same way, they are called isomorphic graphs (denoted by G≅HG≅H).

It is easier to check non-isomorphism than isomorphism. If any of these following conditions occurs, then two graphs are non-isomorphic −

- The number of connected components are different
- Vertex-set cardinalities are different
- Edge-set cardinalities are different
- Degree sequences are different

Example

The following graphs are isomorphic −



Homomorphism

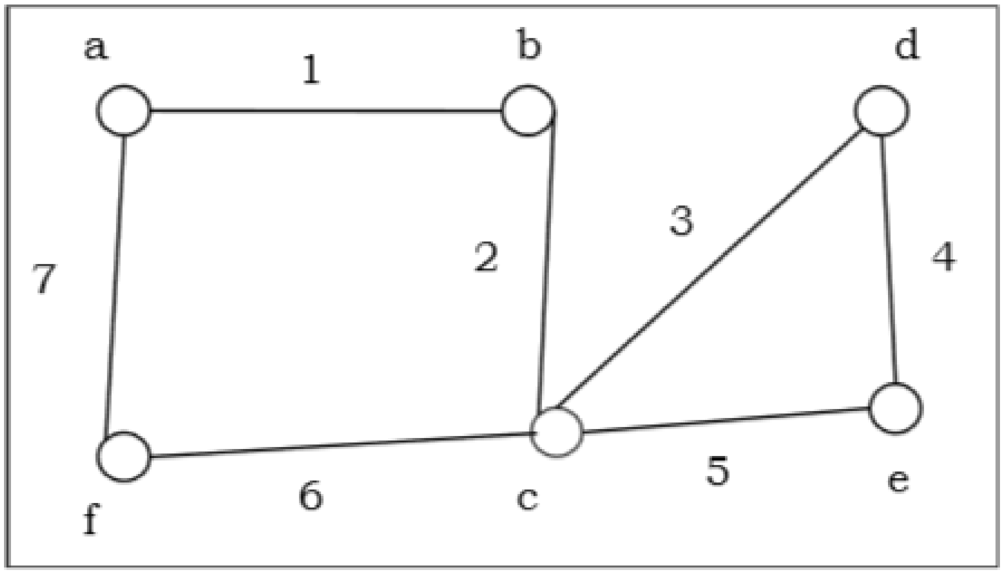
A homomorphism from a graph GG to a graph HH is a mapping (May not be a bijective mapping)h:G→Hh:G→H such that
− (x,y)∈E(G)→(h(x),h(y))∈E(H)(x,y)∈E(G)→(h(x),h(y))∈E(H). It maps adjacent vertices of graph GG to the adjacent vertices of the graph HH.

Properties of Homomorphisms

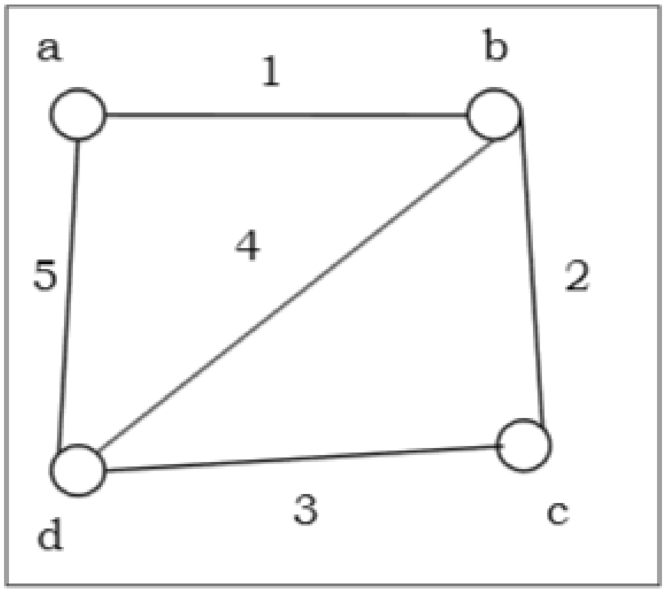
- A homomorphism is an isomorphism if it is a bijective mapping.
- Homomorphism always preserves edges and connectedness of a graph.
- The compositions of homomorphisms are also homomorphisms.
- To find out if there exists any homomorphic graph of another graph is a NPcomplete problem.

Euler Graphs

A connected graph GG is called an Euler graph, if there is a closed trail which includes every edge of the graph GG. An Euler path is a path that uses every edge of a graph exactly once. An Euler path starts and ends at different vertices.
An Euler circuit is a circuit that uses every edge of a graph exactly once. An Euler circuit always starts and ends at the same vertex. A connected graph GG is an Euler graph if and only if all vertices of GG are of even degree, and a connected graph GG is Eulerian if and only if its edge set can be decomposed into cycles.

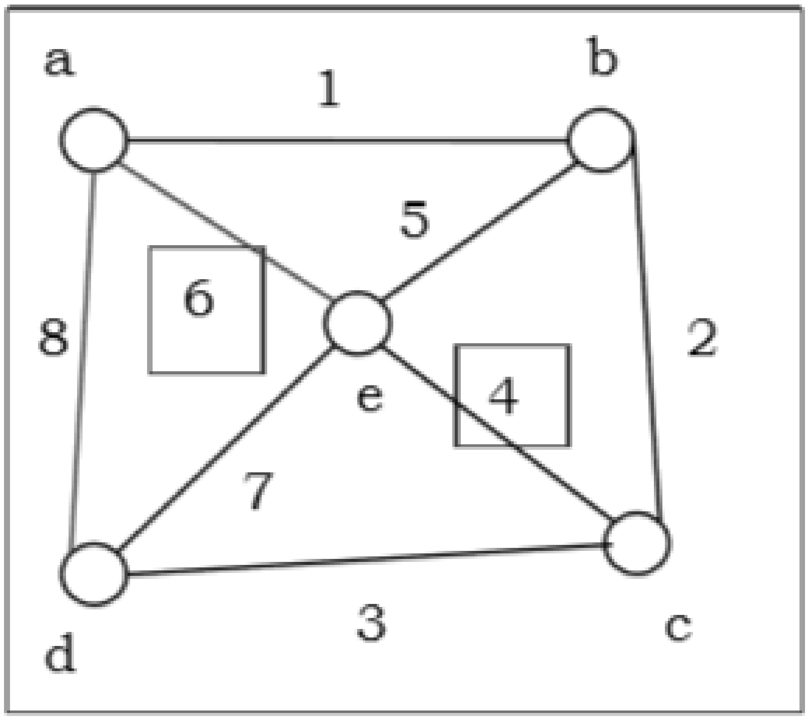


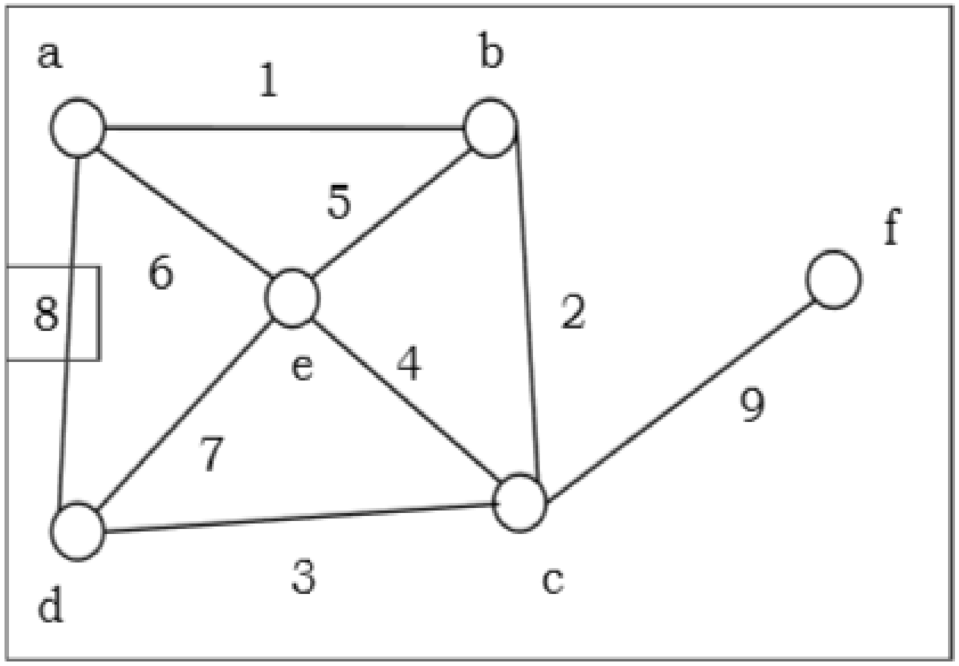
The above graph is an Euler graph as “a1b2c3d4e5c6f7g”“a1b2c3d4e5c6f7g” covers all the edges of the graph.



Hamiltonian Graphs

A connected graph GG is called Hamiltonian graph if there is a cycle which includes every vertex of GG and the cycle is called Hamiltonian cycle. Hamiltonian walk in graph GG is a walk that passes through each vertex exactly once.
If GG is a simple graph with n vertices, where n≥3n≥3 If deg(v)≥n2deg(v)≥n2 for each vertex vv, then the graph GG is Hamiltonian graph. This is called **Dirac's Theorem**.
If GG is a simple graph with nn vertices,
where n≥2n≥2 if deg(x)+deg(y)≥ndeg(x)+deg(y)≥n for each pair of non-adjacent vertices x and y, then the graph GG is Hamiltonian graph. This is called **Ore's theorem**.

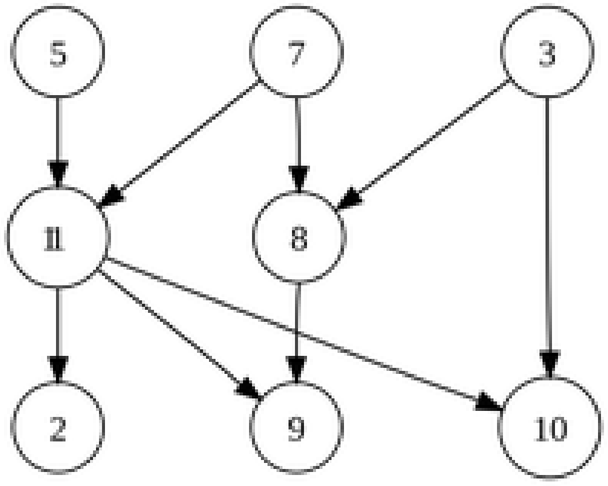




Directed Graphs

A directed graph is a graph whose edges all have an orientation and are thus represented by arrows instead of segments.

Directed graphs are useful when the relationship connecting nodes works in one direction but not necessarily the opposite direction. For example, in an overlap graph, strings ss and tt are connected with a directed edge if a suffix of ss is equal to a prefix of tt. The opposite case is not necessarily true, which is why directed edges are appropriate for the overlap graph.
The figure below illustrates a simple directed graph whose nodes are labeled with integers. This directed graph is a particular type of graph containing no cycles, fittingly called a directed acyclic graph.



Sequential Representations of Graphs

Next we consider a fundamentally different approach to implementing trees. The goal is to store a series of node values with the minimum information needed to reconstruct the tree structure. This approach, known as a ***sequential tree representation***, has the advantage of saving space because no pointers are stored. It has the disadvantage that accessing any node in the tree requires sequentially processing all nodes that appear before it in the node list. In other words, node access must start at the beginning of the node list, processing nodes sequentially in whatever order they are stored until the desired node is reached. Thus, one primary virtue of the other implementations discussed in this section is lost: efficient access (typically Θ(logn)Θ(log n) time) to arbitrary nodes in the tree. Sequential tree implementations are ideal for archiving trees on disk for later use because they save space, and the tree structure can be reconstructed as needed for later processing.

Sequential tree implementations can be used to ***serialize*** a tree structure. Serialization is the process of storing an object as a series of bytes, typically so that the data structure can be transmitted between computers. This capability is important when using data structures in a distributed processing environment.

A sequential tree implementation typically stores the node values as they would be enumerated by a preorder traversal, along with sufficient information to describe the tree's shape. If the tree has restricted form, for example if it is a full binary tree, then less information about structure typically needs to be stored. A general tree, because it has the most flexible shape, tends to require the most additional shape information. There are many possible sequential tree implementation schemes. We will begin by describing methods appropriate to binary trees, then generalize to an implementation appropriate to a general tree structure.

Because every node of a binary tree is either a leaf or has two (possibly empty) children, we can take advantage of this fact to implicitly represent the tree's structure. The most straightforward sequential tree implementation lists every node value as it would be enumerated by a preorder traversal. Unfortunately, the node values alone do not provide enough information to recover the shape of the tree. In particular, as we read the series of node values, we do not know when a leaf node has been reached. However, we can treat all non-empty nodes as internal nodes with two (possibly empty) children. Only NULL values will be interpreted as leaf nodes, and these can be listed explicitly. Such an augmented node list provides enough information to recover the tree structure.

Adjacency Matrices

In graph theory, an **adjacency matrix** is nothing but a square matrix utilised to describe a finite graph. The components of the matrix express whether the pairs of a finite set of vertices (also called nodes) are adjacent in the graph or not. In graph representation, the networks are expressed with the help of nodes and edges, where nodes are the vertices and edges are the finite set of ordered pairs.

Table of Contents:

Graphs can also be defined in the form of matrices. To perform the calculation of paths and cycles in the graphs, matrix representation is used. It is calculated using matrix operations. The two most common representation of the graphs are:

- Adjacency Matrix
- Adjacency List

We will discuss here about the matrix, its formation and its properties.

Adjacency Matrix Definition

The **adjacency matrix**, also called the **connection matrix**, is a matrix containing rows and columns which is used to represent a simple labelled graph, with 0 or 1 in the position of (V_i , V_j) according to the condition whether V_i and V_j are adjacent or not. It is a compact way to represent the finite graph containing n vertices of a m x m matrix M. Sometimes adjacency matrix is also called as **vertex matrix** and it is defined in the general form as

{10ifPi→Pjotherwise}

If the simple graph has no self-loops, Then the vertex matrix should have 0s in the diagonal. It is symmetric for the undirected graph. The connection matrix is considered as a square array where each row represents the out-nodes of a graph and each column represents the in-nodes of a graph. Entry 1 represents that there is an edge between two nodes.

The adjacency matrix for an undirected graph is symmetric. This indicates the value in the ith row and jth column is identical with the value in the jth row and ith column. Additionally, a fascinating fact includes matrix multiplication. If the adjacency matrix is multiplied by itself (matrix multiplication), if there is a nonzero value present in the ith row and jth column, there is a route from V_i to V_j of length equal to two. It does not specify the path though there is a path created. The nonzero value indicates the number of distinct paths present.

Related Links	
Orthogonal Matrix	Matrices
Graph Theory	Elementary Transformation Of Matrices

How to create an Adjacency Matrix?

If a graph G with n vertices, then the vertex matrix n x n is given by

A=⎡⎣⎢⎢⎢⎢a11a21⋮an1a12a22⋮an2⋯⋯⋱⋯a1na2n⋮ann⎤⎦⎥⎥⎥⎥

Where, the value a_{ij} equals the number of edges from the vertex i to j. For an undirected graph, the value $a_{ij} = a_{ji}$ for all i, j , so that the adjacency matrix becomes a symmetric matrix.

Mathematically, this can be explained as:

Let G be a graph with vertex set $\{v_1, v_2, v_3, \ . \ . \ . \ , v_n\}$, then the adjacency matrix of G is the n × n matrix that has a 1 in the (i, j)-position if there is an edge from v_i to v_j in G and a 0 in the (i, j)-position otherwise.

From the given directed graph, the adjacency matrix is written as

The adjacency matrix = $\begin{bmatrix}0&1&0&1&0\\1&0&0&0&0\\0&1&0&1&0\\1&1&1&0&0\\1&0&1&1&0\end{bmatrix}$

Properties

The vertex matrix is an array of numbers which is used to represent the information about the graph. Some of the properties of the graph correspond to the properties of the adjacency matrix, and vice versa. The properties are given as follows:

Matrix Powers

The most well-known approach to get information about the given graph from operations on this matrix is through its powers. The entries of the powers of the matrix give information about paths in the given graph. The theorem is given below to represent the powers of the adjacency matrix.

Theorem: Let us take, A be the connection matrix of a given graph. Then the entries i, j of A^n counts n-steps walks from vertex i to j.

Spectrum

The study of the eigenvalues of the connection matrix of a graph is clearly defined in spectral graph theory. Assume that, A be the connection matrix of a k-regular graph and v be the all-ones column vector in R^n. Then the i-th entry of Av is equal to the sum of the entries in the i^{th} row of A. This represents the number of edges proceeds from vertex i, which is exactly k. So the $A\vec{v} = \lambda\vec{v}$ and this can be expressed as:

$A=\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix}.\begin{bmatrix}k\\k\\\vdots\\k\end{bmatrix}=k\begin{bmatrix}1\\1\\\vdots\\1\end{bmatrix}$

Where $\vec{v}$ is an eigenvector of the matrix A containing the eigenvalue k

Isomorphisms

The given two graphs are said to be isomorphic if one graph can be obtained from the other by relabeling vertices of another graph. It is noted that the isomorphic graphs need not have the same adjacency matrix. Because this matrix depends on the labelling of the vertices. But the adjacency matrices of the given isomorphic graphs are closely related.

Theorem: Assume that, G and H be the graphs having n vertices with the adjacency matrices A and B. Then G and H are said to be isomorphic if and only if there is an occurrence of permutation matrix P such that B=PAP^{-1}.

Adjacency Matrix Undirected Graph

For an undirected graph, the protocol followed will depend on the lines and loops. That means each edge (i.e., line) adds 1 to the appropriate cell in the matrix, and each loop adds 2. Thus, using this practice, we can find the degree of a vertex easily just by taking the sum of the values in either its respective row or column in the adjacency matrix. This can be understood using the below example.

From this, the adjacency matrix can be shown as:

A=⎡⎣⎢⎢⎢⎢⎢⎢⎢⎢011000101011110100001010010101010010⎤⎦⎥⎥⎥⎥⎥⎥⎥⎥

Adjacency Matrix Directed Graph

As explained in the previous section, the directed graph is given as:

The adjacency matrix for this type of graph is written using the same conventions that are followed in the earlier examples.

Traversal

In this tutorial, you will learn about different tree traversal techniques. Also, you will find working examples of different tree traversal methods in C, C++, Java and Python.

Traversing a tree means visiting every node in the tree. You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.

Linear data structures like arrays, stacks, queues, and linked list have only one way to read the data. But a hierarchical data structure like a tree can be traversed in different ways.

Let's think about how we can read the elements of the tree in the image shown above.

Starting from top, Left to right

```
1 -> 12 -> 5 -> 6 -> 9
```

Starting from bottom, Left to right

```
5 -> 6 -> 12 -> 9 -> 1
```

Although this process is somewhat easy, it doesn't respect the hierarchy of the tree, only the depth of the nodes.

Instead, we use traversal methods that take into account the basic structure of a tree i.e.

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

The struct node pointed to by left and right might have other left and right children so we should think of them as sub-trees instead of sub-nodes.
According to this structure, every tree is a combination of

- A node carrying data
- Two subtrees

Remember that our goal is to visit each node, so we need to visit all the nodes in the subtree, visit the root node and visit all the nodes in the right subtree as well.

Depending on the order in which we do this, there can be three types of traversal.

Inorder traversal

1. First, visit all the nodes in the left subtree

2. Then the root node
3. Visit all the nodes in the right subtree

```
inorder(root->left)
display(root->data)
inorder(root->right)
```

Connected Component and Spanning Trees

Spanning tree can be defined as a sub-graph of connected, undirected graph G that is a tree produced by removing the desired number of edges from a graph. In other words, Spanning tree is a non-cyclic sub-graph of a connected and undirected graph G that connects all the vertices together. A graph G can have multiple spanning trees.

Minimum Spanning Tree

There can be weights assigned to every edge in a weighted graph. However, A minimum spanning tree is a spanning tree which has minimal total weight. In other words, minimum spanning tree is the one which contains the least weight among all other spanning tree of some particular graph.

Shortest path algorithms

In this section of the tutorial, we will discuss the algorithms to calculate the shortest path between two nodes in a graph.

There are two algorithms which are being used for this purpose.

- Prim's Algorithm
- Kruskal's Algorithm

Minimum Cost Spanning Trees

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of the minimum spanning trees for its connected components.

There are quite a few use cases for minimum spanning trees. One example would be a telecommunications company trying to lay cable in a new neighborhood. If it is constrained to bury the cable only along certain paths (e.g. roads), then there would be a graph containing the points (e.g. houses) connected by those paths. Some of the paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. Currency is an acceptable unit for edge weight – there is no requirement for edge lengths to obey normal rules of geometry such as the triangle inequality. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects every house; there might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost, representing the least expensive path for laying the cable.

File Structures:

Physical Storage Media File Organization

1. Several types of data storage exist in most computer systems. They vary in speed of access, cost per unit of data, and reliability.
    - **Cache:** most costly and fastest form of storage. Usually very small, and managed by the operating system.

    - **Main Memory (MM):** the storage area for data available to be operated on.

        - General-purpose machine instructions operate on main memory.
        - Contents of MM are usually lost in a power failure or ``crash''.
        - Usually too small to store the entire database.

    - **Direct-access Storage (disk):** primary medium for long-term storage.
        - Typically the entire database is stored on disk.
        - Data must be moved from disk to MM in order for the data to be operated on.
        - After operations are performed, data must be copied back to disk if any changes were made.
        - Disk storage is called **direct access** storage as it is possible to read data on the disk in any order (unlike sequential access).
        - Disk storage usually survives power failures and system crashes.

    - **Tape Storage:** used primarily for backup and archival data.
        - Cheaper, but much slower access, since tape must be read sequentially from the beginning.
        - Used as protection from disk failures!

As disk storage is so important in database implementation, we will look at disk characteristics in detail.

2. Figure 7.2 shows a simple disk.
    - The head is a device which stays close to the surface of the platter and reads or writes information encoded magnetically on the platter.
    - The platter is organized into concentric tracks of data (see Figure 7.3).
    - The **arm** can be positioned over any one of the tracks.
    - The **platter** is spun at high speed.
    - To read information, the arm is **positioned** over the correct track.
    - When the data to be accessed passes under the head,
      the **read** or **write** operation is performed.
3. Since the platter rotates at high speed, it does not take long for the contents of an entire track to pass under the head.
    - This amount of time is called **disk latency time**.
    - Relative to latency time, it takes a long time to reposition the arm.
    - The repositioning time, called **seek time**, grows as the distance the arm must move increases.
    - It is therefore useful to store related information on the same track or physically close tracks in order to minimize seek time.
4. Multiple-platter disks (see figure 7.4) are called **disk-packs**. When we use the term **disk** from now on, we will be referring to **multiple-platter** disks.
    - Multiple disk arms are moved as a unit by the **actuator**.
    - Each arm has two heads, to read disks above and below it.
    - The set of **tracks** over which the heads are located forms a **cylinder**.
    - This cylinder holds that data that is accessible within the disk latency time.
    - It is clearly sensible to store related data in the same or adjacent cylinders.
5. Data is transferred between disk and main memory in units called **blocks**.
    - A **block** is a contiguous sequence of bytes from a single track of one platter.
    - Block sizes range from 512 bytes to several thousand.
    - If several blocks from a cylinder need to be transferred, we may save time by requesting them in the order in which they pass under the heads.
    - Similarly, if blocks are from different cylinders, we may save time by requesting them in an order that minimizes actuator movement.
    - These techniques may not always be possible, or may be expensive.

Organization of records into Blocks

A database is a collection of large amount of related data. In case of RDBMS (Relational Database Management System), the data is stored in the form of relations or tables. As a normal user, we see the data stored in tables but actually this huge amount of data is stored in the form of files in physical memory.

A **File** is a collection of related records stored on the secondary storage such as magnetic disks in binary format. There are various strategies for mapping file records into blocks of disk:

1. Spanned Mapping:

In spanned mapping, the record of a file is stored inside the block even if it can only be stored partially and hence, the record is spanned over two blocks giving it the name *Spanned Mapping*.

- **Advantages:** No wastage of memory (no internal fragmentation).
- **Disadvantages:** The record which has been spanned, while accessing it we would be required to access two blocks and searching time of a block is greater than the searching time of a record inside a block as the number of blocks on the disk are too large.

2. Unspanned Mapping:

In unspanned mapping, unlike spanned strategy, the record of a file is stored inside the block only if it can be stored completely inside it.

- **Advantages:** Access time of a record is less. It is because in spanned mapping, for accessing the spanned record we were required to access two blocks and as we know accessing time of a block is much higher than that of a record. But in unspanned mapping, for a single record we need to access only a single block every time and hence, it is faster.

- **Disadvantages:** Wastage of memory is more (internal fragmentation).

Sequential Files

A sequential file is an ordinary text file. Each character in the file is assumed to be either a text character or some other ASCII control character such as newline. The character is in the character set specified when the file is opened. By default this is the platform-native character set.

Sequential files provide access at the level of lines or strings of text: that is, data that is not divided into a series of records. However, a sequential file is not well suited for binary data, because a number in a sequential file is written as a character string.

Opening sequential files

A sequential file can be opened in one of three modes: input, output, or append. After opening a file, you must close it before opening it in another mode.

The syntax is:

Open fileName [**For** {**Input** | **Output** | **Append**}] **As** fileNumber [**Len =** bufferSize] [**Charset** = MIMECharsetName]

Where Input means read-only access to the file, Output means write-only access, and Append means write-only access starting at the end of the file. Access in all three sequential modes is one line at a time. To get an unused fileNumber, use the FreeFile function.

bufferSize is the number of characters loaded into the internal buffer before being flushed to disk. This is a performance-enhancing feature: the larger the buffer, the faster the I/O. However, larger buffer sizes require more memory. The default buffer size for sequential files is 512 bytes.

MIMECharsetName designates the character set. The default is the platform-native character set, except that if a UTF-16 or UTF-8 byte order mark (BOM) is present, the BOM character set is used, and on OS/400® the CCSID is used if a BOM is not present.

When you try to open a file for sequential input, the file must already exist. If it doesn't, you get an error. When you try to open a nonexistent file in output or append mode, the file is created automatically.

Writing to sequential files

You can write the contents of variables to a sequential file that was opened in output or append mode using the Print # or Write # statement.

The parameters to Print can be strings or numeric expressions; they are converted to their string representations automatically.

This example writes the contents of Var1 and Var2 (separated by tabs, because of the commas in the statement) to the file numbered idFile.Print#idFile, Var1, Var2

```
Print #idfile, Var1, Var2
```

The Write # statement generates output compatible with the Input # statement by separating each pair of expressions with a comma, and inserting quotation marks around strings.

For example:

```
Dim supV As Variant, tailV As Variant
supV = 456
tailV = NULL
Write #idFile, "Testing", 123, supV, tailV
```

The statements generate the following line in the file numbered idFile:

```
"Testing",123,456,#NULL#
```

Indexing and Hashing

The main difference between indexing and hashing is that the **i**ndexing optimizes the performance of a database by reducing the number of disk accesses to process queries while hashing calculates the direct location of a data record on the disk without using index structure.

A database is a collection of associated data. A DBMS or Database Management System allows creating, and managing data in the databases easily. The users can write SQL queries to perform operations on the tables of a database. DBMS allows multiple users to access and use data. Furthermore, it allows performing transactions and provides data protection. Indexing and Hashing are two concepts related to DBMS.

Key Areas Covered

1. What is Indexing

– Definition, Functionality

2. What is Hashing

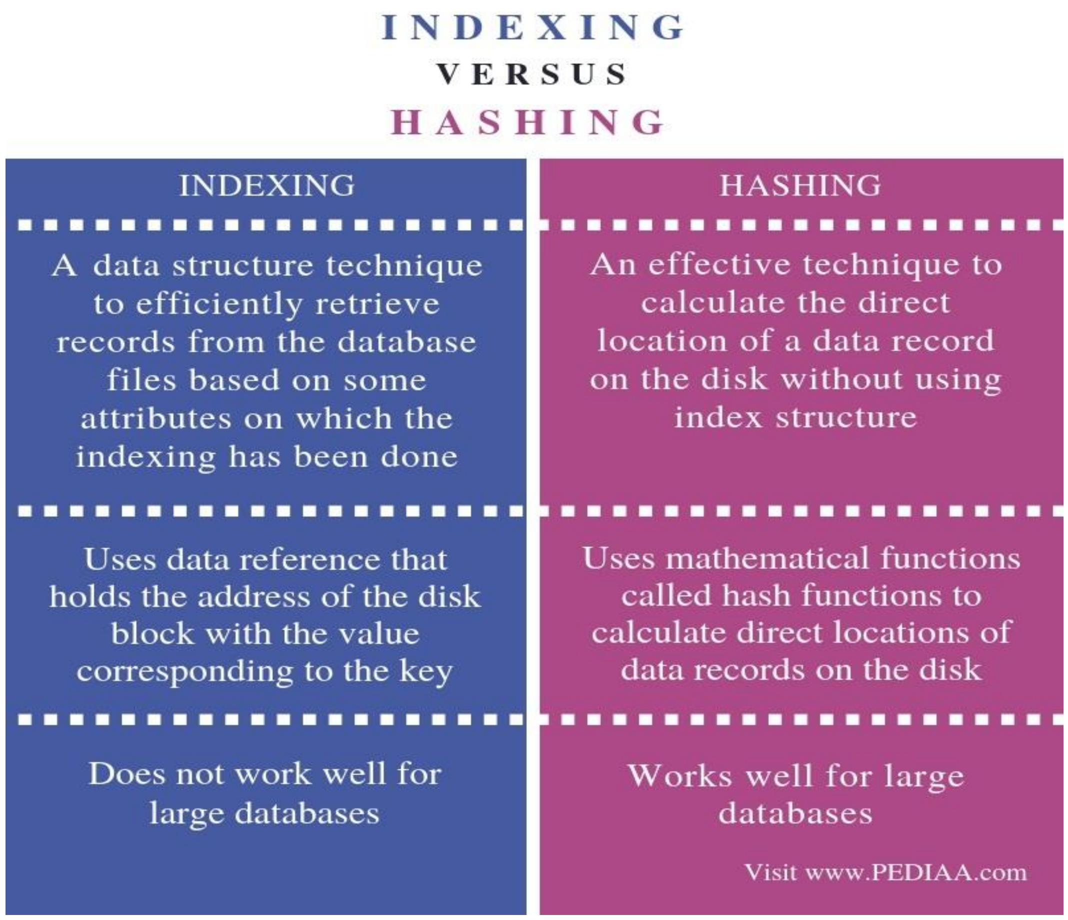
– Definition, Functionality

3. What is the Difference Between Indexing and Hashing

– Comparison of Key Differences

Key Terms

DBMS, Clustered Indexing, Hashing, Indexing, Ordered Indexing, Primary Indexing, Secondary Indexing, SQL



INDEXING VERSUS HASHING

INDEXING	HASHING
A data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done	An effective technique to calculate the direct location of a data record on the disk without using index structure
Uses data reference that holds the address of the disk block with the value corresponding to the key	Uses mathematical functions called hash functions to calculate direct locations of data records on the disk
Does not work well for large databases	Works well for large databases

Visit www.PEDIAA.com

What is Indexing

When executing SQL queries, it takes some amount of time to access data from the disk. Herein, an index is a data structure that helps to find and access data in a table of a database quickly. Indexing technique reduces the number of disks accessed to process queries.

An index consists of two sections; a search key and a data reference. The search key contains the primary key or the candidate key of the table. Data reference holds the address of the disk block that has the value corresponding to that key.
Also, there are various types of indexes. Some of them are as follows.

Ordered Indexing – Indices are sorted, making data searching faster

Primary Indexing – When the index is based on the primary key of the table, it is called a primary index. There are two types of indexes in primary key called dense and spare index. The dense index contains an index record for every search key value in the data file. In the spare index, there are index records for some data items.

Clustered indexing – Uses a combination of two or more columns to create an index. A group of records consists of records with the same characteristics. And, these groups create the indexes.

Secondary indexing – Contains another level of indexing to minimize the size of mapping.

What is Hashing

In a large database, it is not possible to search all the indexes to obtain the required data. Hashing helps to find the direct location of a specific data record on the disk without using indexing. Here, data blocks, also called data buckets, store data. A hashing function is a mathematical function. It helps to generate the addresses of those data blocks. Furthermore, the hashing function can select any column value to generate the address, but it usually uses the primary key to generate the address of the data block.

There are two types of hashing as static and dynamic hashing. In static hashing, the resultant data bucket address is always the same. However, static hashing causes bucket overflowing. Dynamic hashing is a solution to this issue. In dynamic hashing, data bucket increases or decreases depending on the number of records.

Difference Between Indexing and Hashing

Definition

Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing took place. On the other hand, hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure. Thus, this is the main difference between indexing and hashing.

Functionality

Indexing uses data reference that holds the address of the disk block with the value corresponding to the key while hashing uses mathematical functions called hash functions to calculate direct locations of data records on the disk. Hence, this is also a major difference between indexing and hashing.

Application

Another difference between indexing and hashing is that the hashing works well for large databases than indexing.

Conclusion

The main difference between indexing and hashing is that the indexing optimizes the performance of a database by reducing the number of disk accesses to process queries while hashing calculates the direct location of a data record on the disk without using index structure.

Primary indices and Secondary indices

We know that data is stored in the form of records. Every record has a key field, which helps it to be recognized uniquely.

Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to what we see in books.

Indexing is defined based on its indexing attributes. Indexing can be of the following types −

- **Primary Index −**

  Primary index is defined on an ordered data file. The data file is ordered on a **key field**. The key field is generally the primary key of the relation.

- **Secondary Index −**

  Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.

- **Clustering Index −**

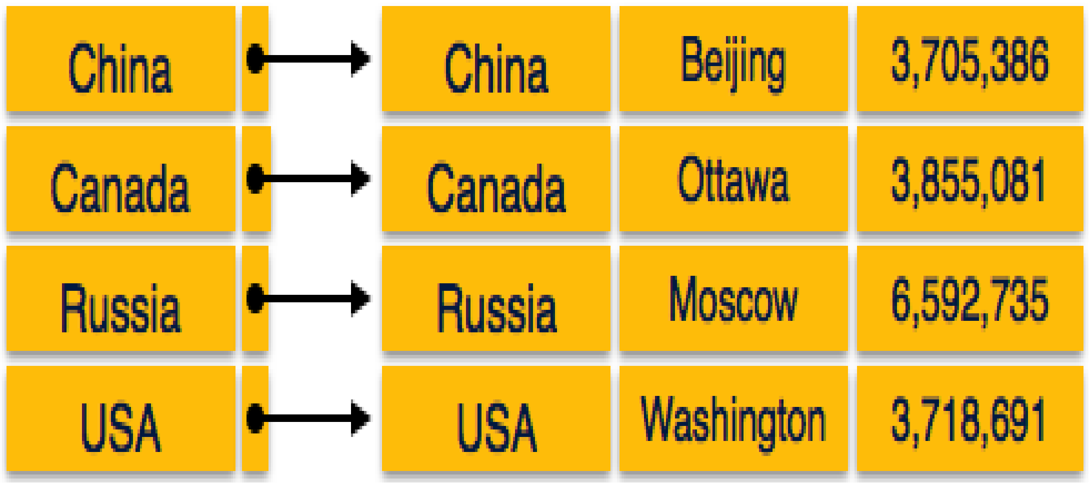
  Clustering index is defined on an ordered data file. The data file is ordered on a non-key field.

Ordered Indexing is of two types −

- Dense Index
- Sparse Index

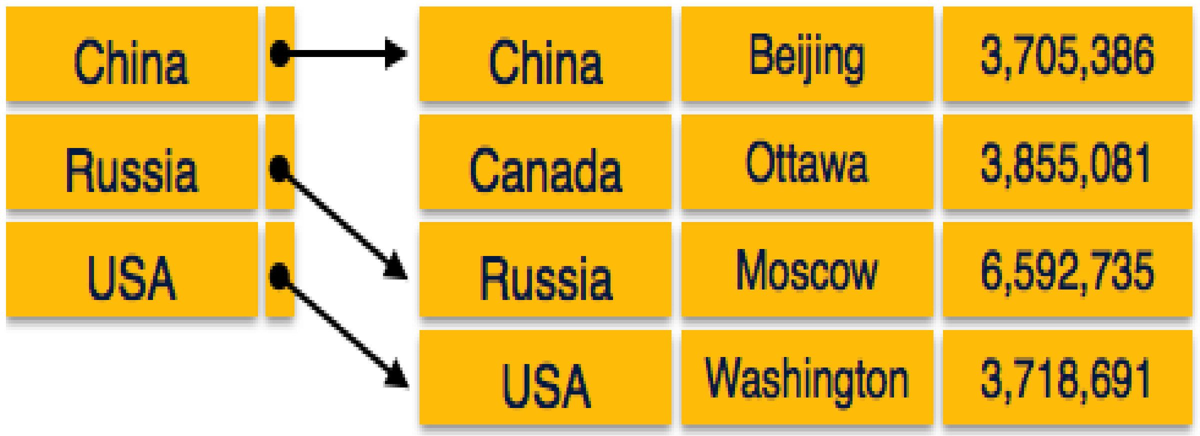
Dense Index

In dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index records contain search key value and a pointer to the actual record on the disk.



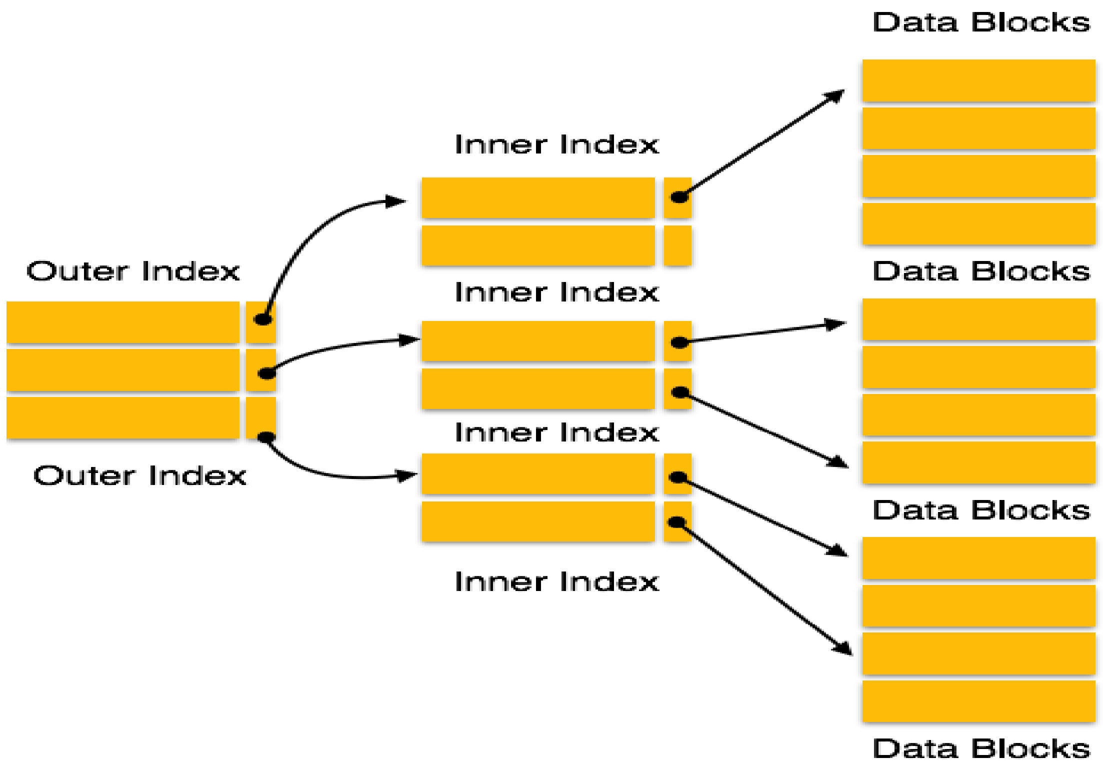
Sparse Index

In sparse index, index records are not created for every search key. An index record here contains a search key and an actual pointer to the data on the disk. To search a record, we first proceed by index record and reach at the actual location of the data. If the data we are looking for is not where we directly reach by following the index, then the system starts sequential search until the desired data is found.



Multilevel Index

Index records comprise search-key values and data pointers. Multilevel index is stored on the disk along with the actual database files. As the size of the database grows, so does the size of the indices. There is an immense need to keep the index records in the main memory so as to speed up the search operations. If single-level index is used, then a large size index cannot be kept in memory which leads to multiple disk accesses.



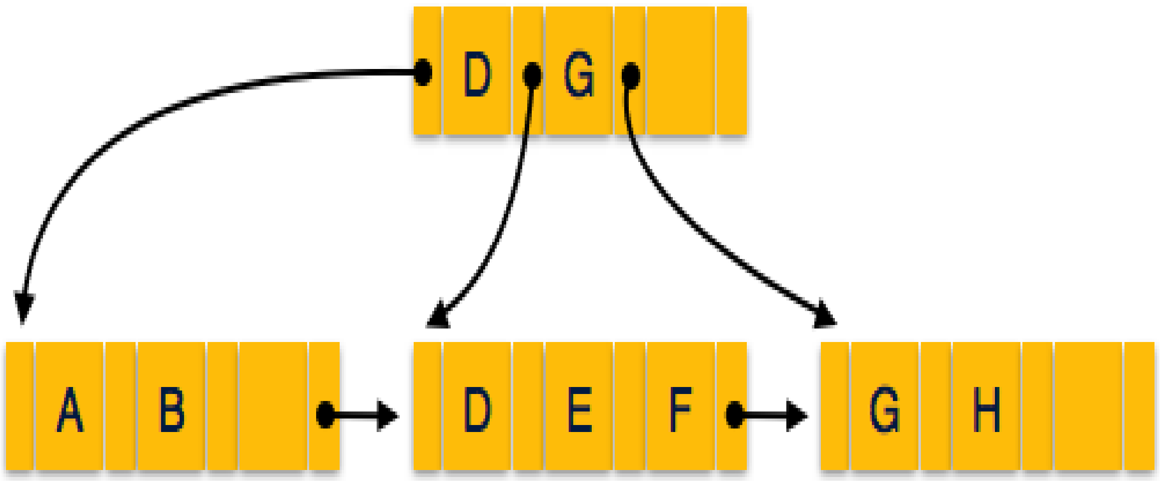
Multi-level Index helps in breaking down the index into several smaller indices in order to make the outermost level so small that it can be saved in a single disk block, which can easily be accommodated anywhere in the main memory.

B^+ Tree

A B^+ tree is a balanced binary search tree that follows a multi-level index format. The leaf nodes of a B^+ tree denote actual data pointers. B^+ tree ensures that all leaf nodes remain at the same height, thus balanced. Additionally, the leaf nodes are linked using a link list; therefore, a B^+ tree can support random access as well as sequential access.

Structure of B^+ Tree

Every leaf node is at equal distance from the root node. A B^+ tree is of the order **n** where **n** is fixed for every B^+ tree.



Internal nodes −

- Internal (non-leaf) nodes contain at least ⌈n/2⌉ pointers, except the root node.
- At most, an internal node can contain **n** pointers.

Leaf nodes −

- Leaf nodes contain at least ⌈n/2⌉ record pointers and ⌈n/2⌉ key values.
- At most, a leaf node can contain **n** record pointers and **n** key values.
- Every leaf node contains one block pointer **P** to point to next leaf node and forms a linked list.

B^+ Tree Insertion

- B^+ trees are filled from bottom and each entry is done at the leaf node.
- If a leaf node overflows −

    - Split node into two parts.
    - Partition at **i = ⌊(m+1)$_{/2}$⌋.**
    - First **i** entries are stored in one node.
    - Rest of the entries (i+1 onwards) are moved to a new node.
    - **ith** key is duplicated at the parent of the leaf.
- **If a non-leaf node overflows −**
    - Split node into two parts.
    - Partition the node at **i = ⌈(m+1)$_{/2}$⌉**.
    - Entries up to **i** are kept in one node.
    - Rest of the entries are moved to a new node.

B$^+$ Tree Deletion

- B$^+$ tree entries are deleted at the leaf nodes.
- The target entry is searched and deleted.
    - If it is an internal node, delete and replace with the entry from the left position.
- After deletion, underflow is tested,
    - If underflow occurs, distribute the entries from the nodes left to it.
- If distribution is not possible from left, then
    - Distribute from the nodes right to it.
- If distribution is not possible from left or from right, then
    - Merge the node with left and right to it.

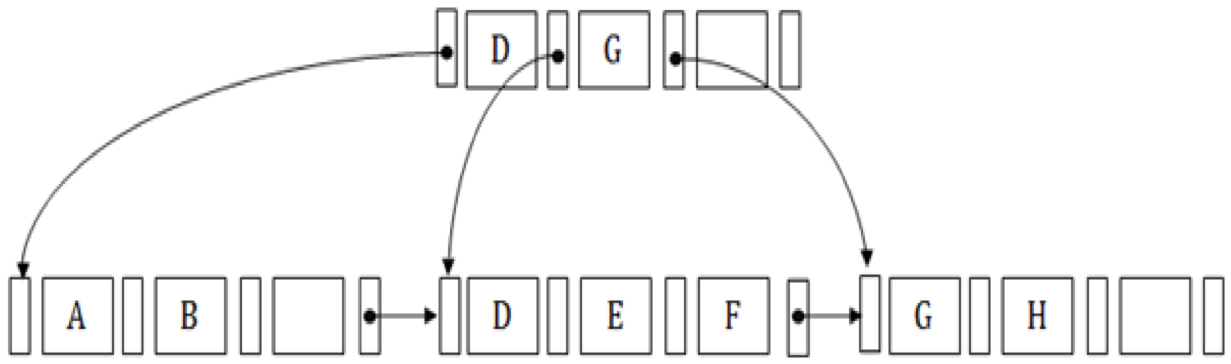
B+ Tree index Files

- The B+ tree is a balanced binary search tree. It follows a multi-level index format.
- In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
- In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.

Structure of B+ Tree

- In the B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order n where n is fixed for every B+ tree.

- It contains an internal node and leaf node.



Internal node

- An internal node of the B+ tree can contain at least n/2 record pointers except the root node.
- At most, an internal node of the tree contains n pointers.

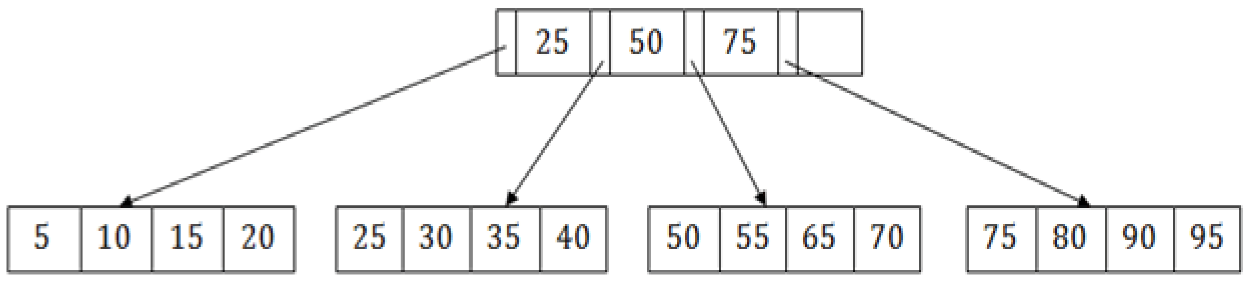
Leaf node

- The leaf node of the B+ tree can contain at least n/2 record pointers and n/2 key values.
- At most, a leaf node contains n record pointer and n key values.
- Every leaf node of the B+ tree contains one block pointer P to point to next leaf node.

Searching a record in B+ Tree

Suppose we have to search 55 in the below B+ tree structure. First, we will fetch for the intermediary node which will direct to the leaf node that can contain a record for 55.

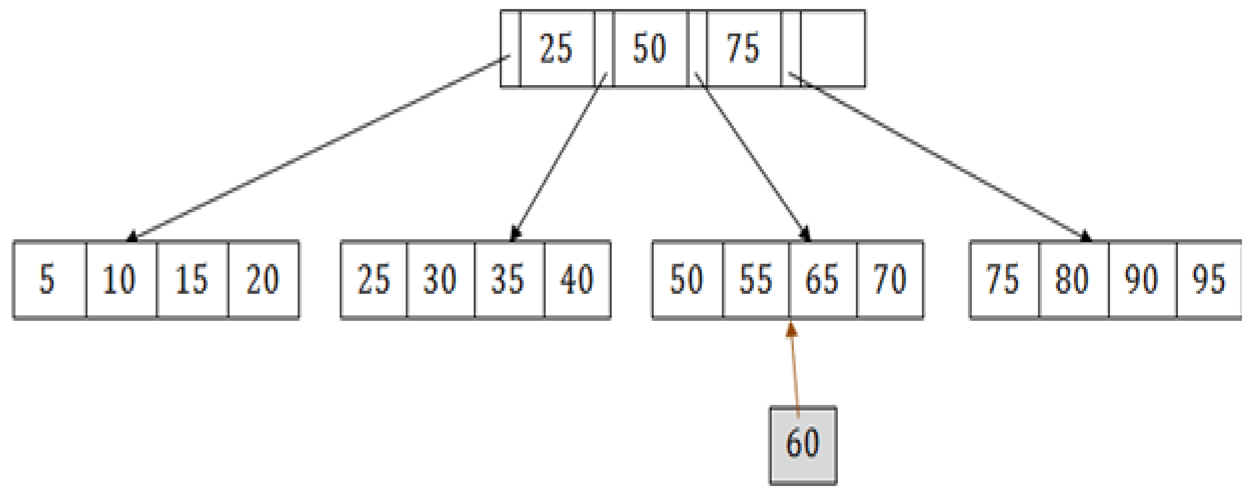
So, in the intermediary node, we will find a branch between 50 and 75 nodes. Then at the end, we will be redirected to the third leaf node. Here DBMS will perform a sequential search to find 55.



B+ Tree Insertion

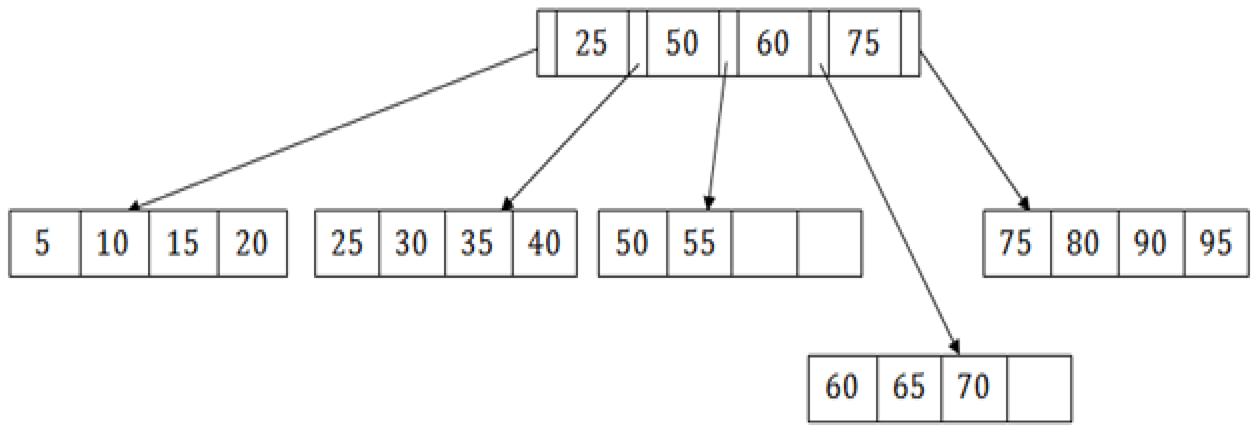
Suppose we want to insert a record 60 in the below structure. It will go to the 3rd leaf node after 55. It is a balanced tree, and a leaf node of this tree is already full, so we cannot insert 60 there.

In this case, we have to split the leaf node, so that it can be inserted into tree without affecting the fill factor, balance and order.



The 3rd leaf node has the values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node of the tree in the middle so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes.

If these two has to be leaf nodes, the intermediate node cannot branch from 50. It should have 60 added to it, and then we can have pointers to a new leaf node.

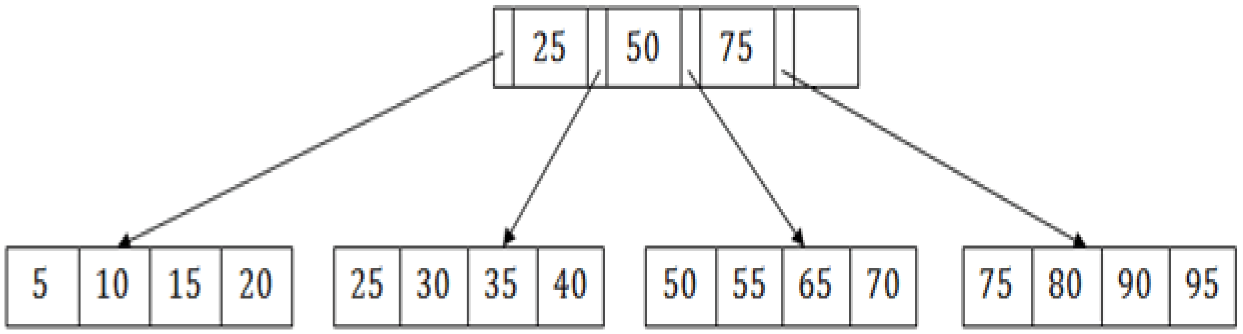


This is how we can insert an entry when there is overflow. In a normal scenario, it is very easy to find the node where it fits and then place it in that leaf node.

B+ Tree Deletion

Suppose we want to delete 60 from the above example. In this case, we have to remove 60 from the intermediate node as well as from the 4th leaf node too. If we remove it from the intermediate node, then the tree will not satisfy the rule of the B+ tree. So we need to modify it to have a balanced tree.

After deleting node 60 from above B+ tree and re-arranging the nodes, it will show as follows:



B Tree index Files

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to the main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, the B-Tree node size is kept equal to the disk block size. Since the height of the B-tree is low so total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

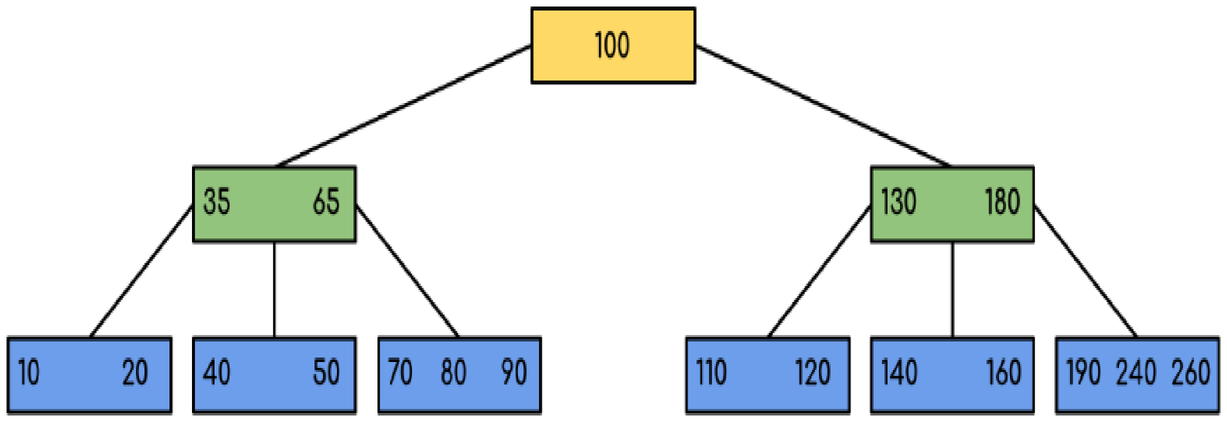
Time Complexity of B-Tree:

SR. NO.	ALGORITHM	TIME COMPLEXITY
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

“n” is the total number of elements in the B-tree.
Properties of B-Tree:

1. All leaves are at the same level.
2. A B-Tree is defined by the term *minimum degree* ‘t’. The value of t depends upon disk block size.
3. Every node except root must contain at least (ceiling)([t-1]/2) keys. The root may contain minimum 1 key.
4. All nodes (including root) may contain at most t – 1 keys.
5. Number of children of a node is equal to the number of keys in it plus 1.
6. All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
7. B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
8. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).

Following is an example of B-Tree of minimum order 5. Note that in practical B-Trees, the value of the minimum order is much more than 5.



We can see in the above diagram that all the leaf nodes are at the same level and all non-leaf have no empty sub-tree and have keys one less than the number of their children.

Indexing and Hashing Comparisons

Definition

Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing took place. On the other hand, hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure. Thus, this is the main difference between indexing and hashing.

Functionality

Indexing uses data reference that holds the address of the disk block with the value corresponding to the key while hashing uses mathematical functions called hash functions to calculate direct locations of data records on the disk. Hence, this is also a major difference between indexing and hashing.

Application

Another difference between indexing and hashing is that the hashing works well for large databases than indexing.

Conclusion

The main difference between indexing and hashing is that the indexing optimizes the performance of a database by reducing the number of disk accesses to process queries while hashing calculates the direct location of a data record on the disk without using index structure.