Data Structures & Algorithms

(DSA-self paced)

from

GeeksForGeeks

A Project Report

Submitted in partial fulfillment of the requirements for the award of degree

Bachelor of Technology in

Electronics and Communication Engineering (Hons.)

Submitted to:

Lovely Professional University, Phagwara,

Punjab

Submitted by:

Mohit Rawat

Reg. No. **11904463**

GeeksforGeeks

CERTIFICATE

OF COURSE COMPLETION

THIS IS TO CERTIFY THAT

Mohit Rawat

has successfully completed the course on DSA Self paced of duration 10 weeks.

Sandeep Jain

Mr. Sandeep Jain
Founder & CEO, GeeksforGeeks

https://media.geeksforgeeks.org/courses/certificates/18e495a48422888f4011594a6387f72e.pdf

www.geeksforgeeks.org

TABLE OF CONTENT

Introduction:-

Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms:

1) Θ Notation: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants.

2) Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.

3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation can be useful when we have lower bound on time complexity of an algorithm. The Omega notation is the least used notation among all three.

We can have three cases to analyze an algorithm:

1. Worst Case
2. Average Case
3. Best Case

Worst Case Analysis (Usually Done) In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes the maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be O(N), where N is the number of elements in the array.

Average Case Analysis (Sometimes done) In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (N+1).

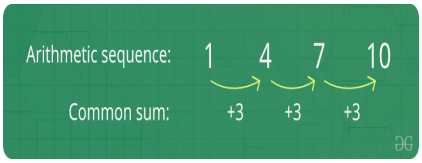
Best Case Analysis (Bogus) : In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on N). So time complexity in the best case would be O(1)

Mathematics.

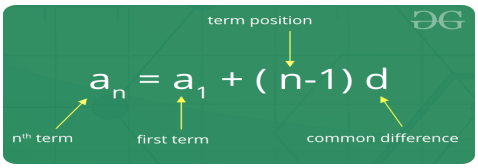
Arithmetic and Geometric Progressions

Arithmetic Progression

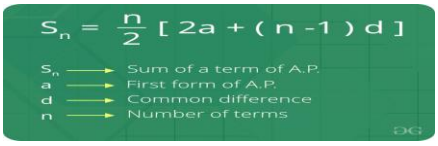
A sequence of numbers is said to be in an **Arithmetic progression** if the difference between any two consecutive terms is always the **same**. In simple terms, it means that the next number in the series is calculated by adding a fixed number to the previous number in the series. For example, 2, 4, 6, 8, 10 is an AP because the difference between any two consecutive terms in the series (common difference) is same (4 - 2 = 6 - 4 = 8 - 6 = 10 - 8 = 2).



Formula of n^{th} term of an A.P :
If 'a' is the initial term and 'd' is the common difference.Thus, the explicit formula is:



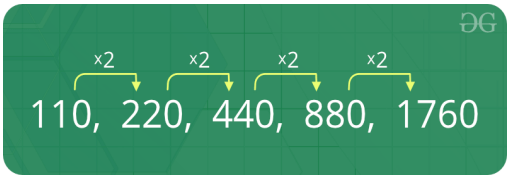
Formula of sum of first n term of A.P:



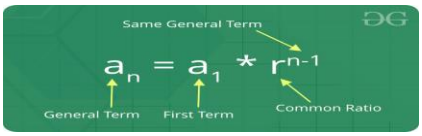
Geometric Progression

A sequence of numbers is said to be in a **Geometric progression** if the ratio of any two consecutive terms is always the same. In simple terms, it means that the next number in the series is calculated by multiplying a fixed number to the previous number in the series. For example, 2, 4, 8, 16 is a GP

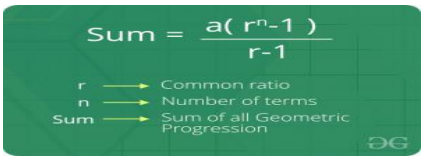
because ratio of any two consecutive terms in the series (common ratio) is the same (4 / 2 = 8 / 4 = 16 / 8 = 2).



Formula of nth term of a Geometric Progression : If 'a' is the first term and 'r' is the common ratio. Thus, the explicit formula is:

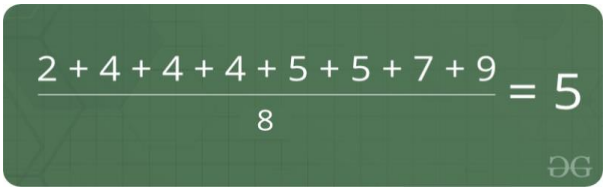


Formula of sum of nth term of Geometric Progression:



Mean and Median

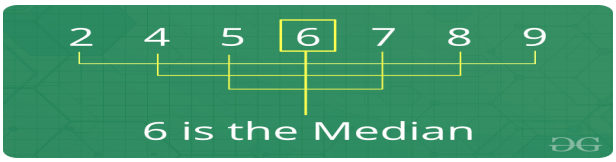
Mean:-
Mean is defined as the average of a given set of data. Let us consider the sequence of numbers **2, 4, 4, 4, 5, 5, 7, 9**, the mean (average) of this given sequence is 5.



Median:-

Median is the middle value of a set of data. To determine the median value in a sequence of numbers, the numbers must first be arranged in an ascending order.

- If the count of numbers in the sequence is ODD, the median value is the number that is in the middle, with the same amount of numbers below and above.
- If the count of numbers in the sequence is EVEN, the median is the average of the two middle values.



Prime Numbers :-

A **prime number** is a whole number greater than 1, which is only divisible by 1 and itself. First few prime numbers are : 2, 3, 5, 7, 11, 13, 17, 19, 23, ...

LCM and HCF

Factors and Multiples: All numbers that divide a number completely, i.e., without leaving any remainder, are called factors of that number. For example, 24 is completely divisible by 1, 2, 3, 4, 6, 8, 12, 24. Each of these numbers is called a factor of 24 and 24 is called a multiple of each of these numbers.

LCM : LCM stands for *Least Common Multiple*. The lowest number that is exactly divisible by each of the given numbers is called the least common multiple of those numbers. For example, consider the numbers 3, 31, and 62 (2 x 31). The LCM of these numbers would be 2 x 3 x 31 = 186.

HCF : The term HCF stands for *Highest Common Factor*. The largest number that divides two or more numbers is the highest common factor (HCF) for those numbers. For example, consider the numbers 30 (2 x 3 x 5), 36 (2 x 2 x 3 x 3), 42 (2 x 3 x 7), 45 (3 x 3 x 5). 3 is the largest number that divides each of these numbers, and hence, is the HCF for these numbers.

Factorials :-

Factorial: In mathematics, the factorial of a number say **N** is denoted by **N!**. The factorial of a number is calculated by multiplying all the integers between 1 and N(both inclusive).

For Example, **4!** = **4 * 3 * 2 * 1** = **24**.

Permutation:-

Permutation is the different arrangements of a given number of elements taken one by one, or some, or all at a time. For example, if we have two elements A and B, then there are two possible arrangements, AB and BA.

Number of permutations when 'r' elements are arranged out of a total of 'n' elements is ${}^{n}\mathbf{P_r = n! / (n}$

Combination

Combination is the different selections of a given number of elements taken one by one, or some, or all at a time. For example, if we have two elements A and B, then there is only one way to select two items, we select both of them.

Modular Arithmetic

Let us take a look at some of the **basic rules and properties** that can be applied in Modular Arithmetic (Addition, Subtraction, Multiplication etc.). Consider numbers **a** and **b** operated under modulo **M**.

1. (a + b) mod M = ((a mod M) + (b mod M)) mod M.
2. (a - b) mod M = ((a mod M) - (b mod M)) mod M.
3. (a * b) mod M = ((a mod M) * (b mod M)) mod M.

Bit Magic

Bitwise Algorithms Basics

The Bitwise Algorithms are used to perform operations at bit-level or to manipulate bits in different ways. The bitwise operations are found to be much faster and are some times used to improve the efficiency of a program.

For example: To check if a number is even or odd. This can be easily done by using Bitwise-AND(&) operator. If the last bit of the operator is set than it is ODD otherwise it is EVEN. Therefore, if num & 1 not equals to zero than num is ODD otherwise it is EVEN.

Bitwise Operators

The operators that works at Bit level are called bitwise operators. In general there are six types of Bitwise Operators as described below:

- **& (bitwise AND)** Takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1. Suppose **A = 5** and **B = 3**, therefore **A & B = 1**.
- **| (bitwise OR)** Takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1. Suppose **A = 5** and **B = 3**, therefore **A | B = 7**.
- **^ (bitwise XOR)** Takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different. Suppose **A = 5** and **B = 3**, therefore **A ^ B = 6**.
- **<< (left shift)** Takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
- **>> (right shift)** Takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
- **~ (bitwise NOT)** Takes one number and inverts all bits of it. Suppose **A = 5**, therefore **~A = 2**.

Important Facts about Bitwise Operators:

- The left shift and right shift operators cannot be used with negative numbers.
- The bitwise XOR operator is the most useful operator from technical interview perspective. We will see some very useful applications of the XOR operator later in the course.
- The bitwise operators should not be used in place of logical operators.

Recursion

Recursion Basics

What is Recursion?

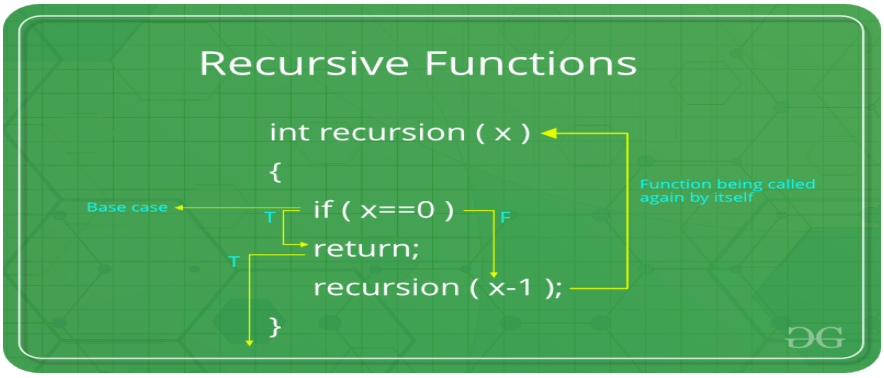
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

What is the base condition in recursion?

In a recursive program, the solution to the base case is provided and the solution of bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
   if (n < = 1) // base case
      return 1;
   else
      return n*fact(n-1);
}
```

In the above example, the base case for n < = 1 is defined and a larger value of a number can be solved by converting to a smaller one till the base case is reached.



How a particular problem is solved using recursion?

The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

Why Stack Overflow error occurs in recursion? If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
   // wrong base case (it may cause
   // stack overflow).
   if (n == 100)
      return 1;

   else
      return n*fact(n-1);
}
```

If fact(10) is called, it will call fact(9), fact(8), fact(7), and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

How memory is allocated to different function calls in recursion?

When any function is called from main(), the memory is allocated to it on stack. A recursive function calls itself, the memory for the called function is allocated on top of memory allocated to the calling function and a different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

Let us take the example of how recursion works by taking a simple function:

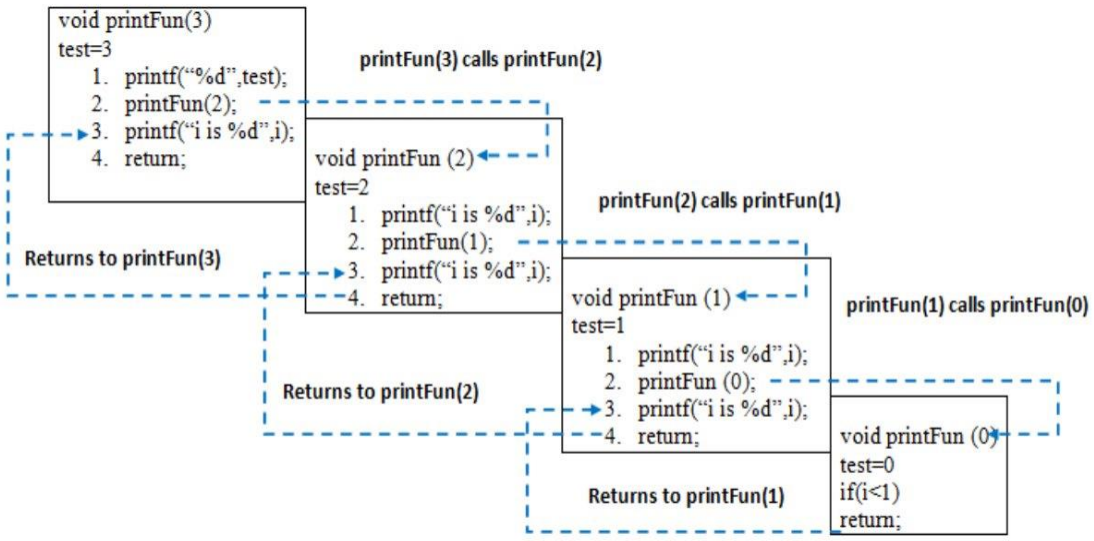
```
void printFun(int test)
{
   if (test < 1)
      return;
   else
   {
      print test;
      printFun(test-1);   // statement 2
```

```
    print test;
    return;
  }
}

// Calling function printFun()
int test = 3;
printFun(test);
```

Output :

```
3 2 1 1 2 3
```

When **printFun(3)** is called from main(), memory is allocated to **printFun(3)** and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints '3'. In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable test is initialized to 2 and statements 1 to 4 are pushed in the stack.
Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**. **printFun(0)** goes to if statement and it returns to **printFun(1)**. Remaining statements of **printFun(1)** are executed and it returns to **printFun(2)** and so on. In the output, values from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in below diagram.



Disadvantage of Recursion: Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa.

Recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached. A recursive program also has greater time requirements because of function calls and return overhead.

Advantages of Recursion: Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of the stack data structure.

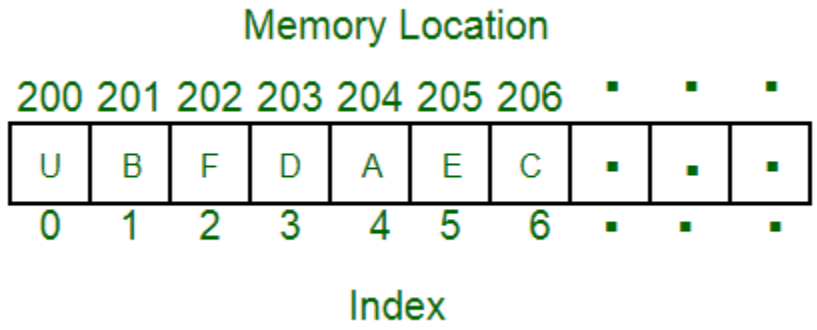
Array.

Introduction to Arrays

An array is a collection of items of the same data type stored at contiguous memory locations. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

For simplicity, we can think of an array as a fleet of stairs where on each step a value is placed (let's say one of your friends). Here, you can identify the location of any of your friends by simply knowing the count of the step that they are on.

Remember: "Location of the next index depends on the data type that we use".



The above image can be looked at as a top-level view of a staircase where you are at the base of the staircase. Each element can be uniquely identified by their index in the array (in a similar way where you could identify your friends by the step on which they were on in the above example).

Defining an Array: Array definition is similar to defining any other variable. There are two things that are needed to be kept in mind, **the data type of the array elements** and the **size** of the array. The size of the array is fixed and the memory for an array needs to be allocated before use, the size of an array cannot be increased or decreased dynamically.

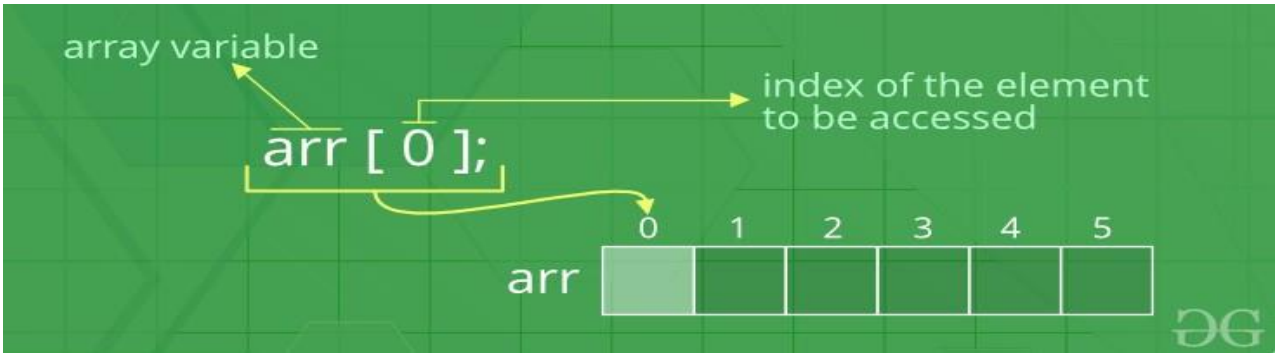
Generally, arrays are declared as:

dataType arrayName[arraySize];

An array is distinguished from a normal variable
by brackets [and].

Accessing array elements: Arrays allows to access elements randomly. Elements in an array can be accessed using indexes. Suppose an array named **arr** stores N elements. Indexes in an array are in the range of **0 to N-1**, where the first element is present at 0-th index and consecutive elements are placed at consecutive indexes. Element present at i^{th} index in the array **arr[]** can be accessed as arr[i].

The below image shows an array **arr[]** of size 5:



Advantages of using arrays:

- Arrays allow random access of elements. This makes accessing elements by their position faster.
- Arrays have better cache locality that can make a pretty big difference in performance.

Searching in an Array

Searching for an element in an array means to check if a given element is present in the array or not. This can be done by accessing elements of the array one by one starting from the first element and checking whether any of the elements matches with the given element.

We can use loops to perform the above operation of array traversal and access the elements, using indexes.

Suppose the array is named **arr[**] with size **N** and the element to be searched is referred to as **key**. Below is the algorithm to perform the search operation in the given array.

```
for(i = 0; i < N; i++)
{
   if(arr[i] == key)
   {
```

```
    print "Element Found";
  }
  else
  {
    print "Element not Found";
  }
}
```

Time Complexity of this search operation will be O(N) in the worst case as we are checking every element of the array from 1st to last, so the number of operations is N.

Insertion and Deletion in Arrays

Insertion in Arrays:-

Given an array of a given size. The task is to insert a new element in this array. There are two possible ways of inserting elements in an array:

1. Insert elements at the end of the array.
2. Insert element at any given index in the array.

Special Case: A special case is needed to be considered is that whether the array is already full or not. If the array is full, then the new element can not be inserted.

Consider the given array is **arr[]** and the initial size of the array is N, that is the array can contain a maximum of N elements and the length of the array is **len**. That is, there are *len* number of elements already present in this array.

- **Insert an element K at end in arr[]**: The first step is to check if there is any space left in the array for new element. To do this check,

```
if(len < N)
   // space left
else
   // array is full
```

If there is space left for the new element, insert it directly at the end at position **len + 1** and index **len**:

```
arr[len] = k;
```

Time Complexity of this insert operation is constant, i.e. O(1) as we are directly inserting the element in a single operation.

Deletion in Arrays:-

To delete a given element from an array, we will have to first search the element in the array. If the element is present in the array then delete operation is performed for the element otherwise the user is notified that the array does not contains the given element.

Consider the given array is **arr[]** and the initial size of the array is N, that is the array can contain a maximum of N elements and the length of the array is **len**. That is, there are *len* number of elements already present in this array.

Deleting an element K from the array arr[]: Search the element K in the array arr[] to find the index at which it is present.

```
for(i = 0; i < N; i++)
{
   if(arr[i] == K)
      idx = i; return;
   else
      Element not Found;
}
```

Now, to delete the element present at index **idx**, left shift all of the elements present after *idx* by one place and finally reduce the length of the array by 1.

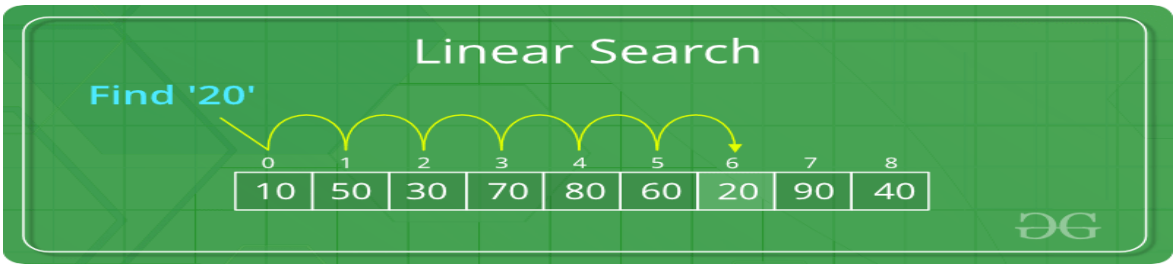
```
for(i = idx+1; i < len; i++)
{
   arr[i-1] = arr[i];
}

len = len-1;
```

Time Complexity in worst case of this insertion operation can be linear i.e. O(N) as we might have to shift all of the elements by one place to the left.

Searching.

Linear Search :-

Linear Search means to sequentially traverse a given list or array and check if an element is present in the respective array or list. The idea is to start traversing the array and compare elements of the array one by one starting from the first element with the given element until a match is found or end of the array is reached.

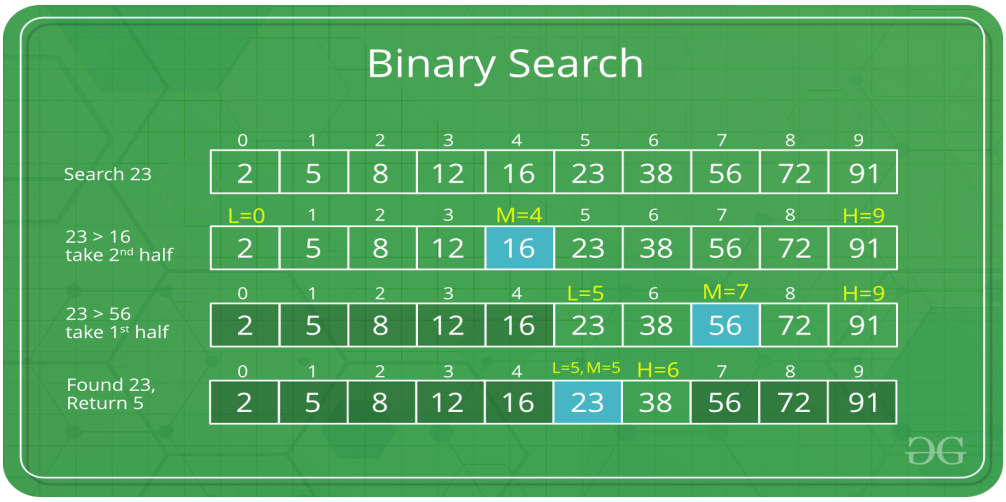


Time Complexity: O(N). Since we are traversing the complete array, so in worst case when the element X does not exists in the array, number of comparisons will be N. Therefore, *worst case time complexity of the linear search algorithm is O(N)*.

Binary Search

Binary Search is a searching algorithm for searching an element in a sorted list or array. Binary Search is efficient than Linear Search algorithm and performs the search operation in logarithmic time complexity for sorted arrays or lists.

Binary Search performs the search operation by repeatedly dividing the search interval in half. The idea is to begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.



Time Complexity: O(Log N), where N is the number of elements in the array.

Sorting.

Introduction to Sorting :-

Sorting any sequence means to arrange the elements of that sequence according to some specific criterion.

For Example, the array arr[] = {5, 4, 2, 1, 3} after *sorting in increasing order* will be: arr[] = {1, 2, 3, 4, 5}. The same array after *sorting in descending order* will be: arr[] = {5, 4, 3, 2, 1}.

In-Place Sorting: An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list.

In this tutorial, we will see three of such in-place sorting algorithms, namely:

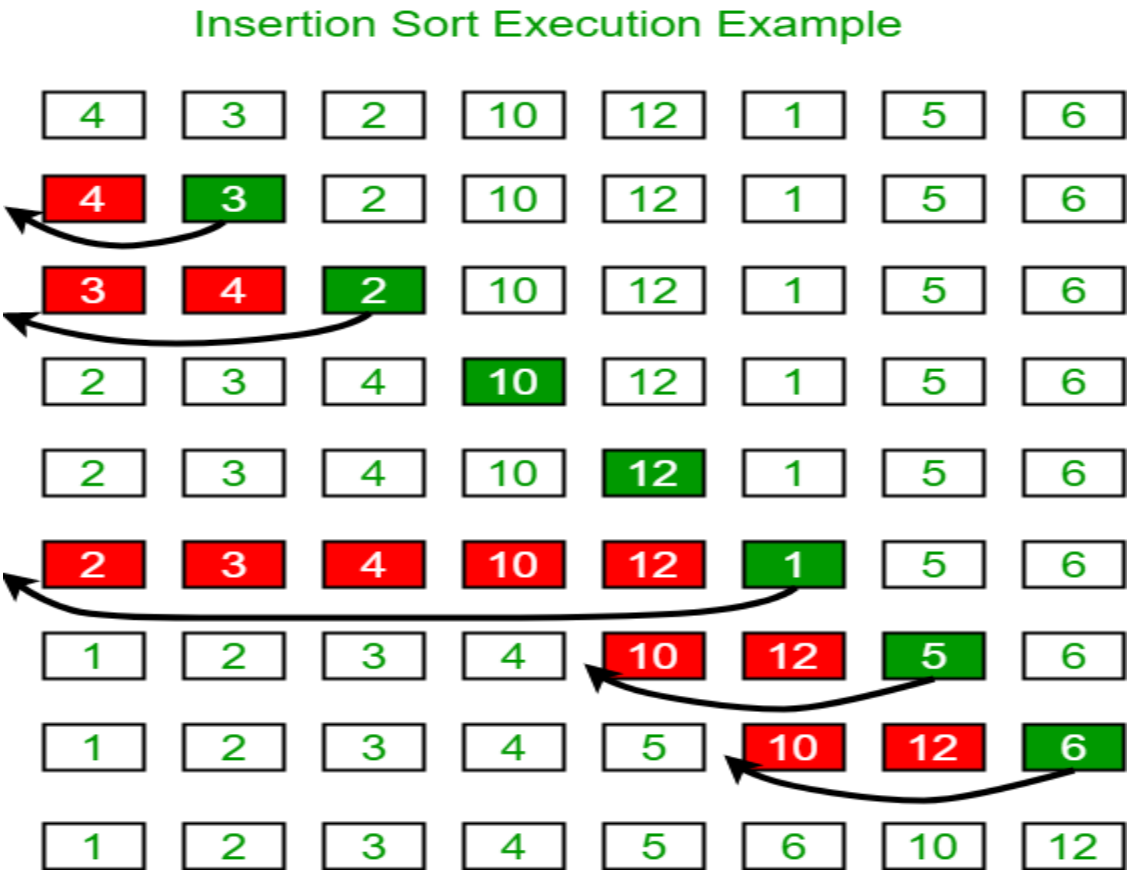
- Insertion Sort
- Selection Sort
- Bubble Sort

Insertion Sort:-

Insertion Sort is an In-Place sorting algorithm. This algorithm works in a similar way of sorting a deck of playing cards.

The idea is to start iterating from the second element of array till last element and for every element insert at its correct position in the subarray before it.

In the below image you can see, how the array [4, 3, 2, 10, 12, 1, 5, 6] is being sorted in increasing order following the insertion sort algorithm.



Algorithm:

Step 1: If the current element is 1st element of array,
it is already sorted.
Step 2: Pick next element
Step 3: Compare the current element will all elements
in the sorted sub-array before it.
Step 4: Shift all of the elements in the sub-array before
the current element which are greater than the current

element by one place and insert the current element
at the new empty space.
Step 5: Repeat step 2-3 until the entire array is sorted.

Time Complexity: O(N^2), where N is the size of the array.

Bubble Sort:-

Bubble Sort is also an in-place sorting algorithm. This is the simplest sorting algorithm and it works on the principle that:

In one iteration if we swap all adjacent elements of an array such that after swap the first element is less than the second element then at the end of the iteration, the first element of the array will be the minimum element.

Bubble-Sort algorithm simply repeats the above steps N-1 times, where N is the size of the array.

Example: Consider the array, arr[] = {5, 1, 4, 2, 8}.

- **First Pass:** (**5 1** 4 2 8) --> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
  (1 **5 4** 2 8) --> (1 **4 5** 2 8), Swap since 5 > 4
  (1 4 **5 2** 8) --> (1 4 **2 5** 8), Swap since 5 > 2
  (1 4 2 **5 8**) --> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
- **Second Pass:** (**1 4** 2 5 8) --> (**1 4** 2 5 8)
  (1 **4 2** 5 8) --> (1 **2 4** 5 8), Swap since 4 > 2
  (1 2 **4 5** 8) --> (1 2 **4 5** 8)
  (1 2 4 **5 8**) --> (1 2 4 **5 8**)
  Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
- **Third Pass:** (**1 2** 4 5 8) --> (**1 2** 4 5 8)
  (1 **2 4** 5 8) --> (1 **2 4** 5 8)
  (1 2 **4 5** 8) --> (1 2 **4 5** 8)
  (1 2 4 **5 8**) --> (1 2 4 **5 8**)

  Time Complexity: O(N^2)

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Function Implementation:

```
void selectionSort(int arr[], int n)
{
  int i, j, min_idx;
  // One by one move boundary of unsorted subarray
  for (i = 0; i < n-1; i++)
  {
    // Find the minimum element in unsorted array
    min_idx = i;
    for (j = i+1; j < n; j++)
     if (arr[j] < arr[min_idx])
       min_idx = j;
    // Swap the found minimum element with the first element
    swap(&arr[min_idx], &arr[i]);
  }
}
```

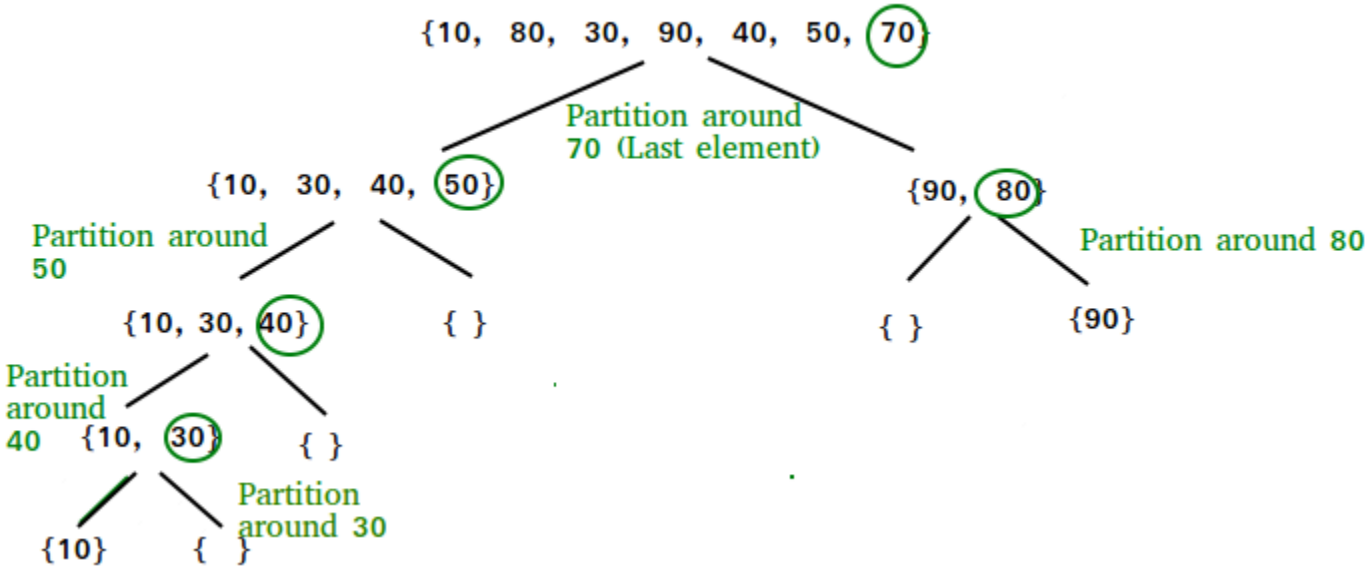
Time Complexity: $O(N^2)$

Quick Sort :-

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



Partition Algorithm:
There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

Implementation:

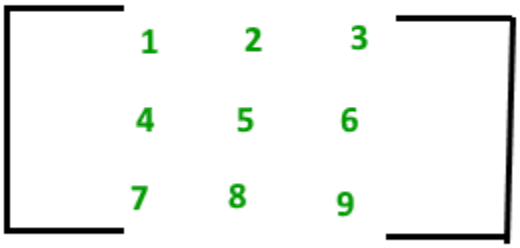
```
int partition (int arr[], int low, int high)
{
   int pivot = arr[high];    // pivot
   int i = (low - 1);  // Index of smaller element
   for (int j = low; j <= high- 1; j++)
   {
      // If current element is smaller than or
      // equal to pivot
      if (arr[j] <= pivot)
      {
         i++;    // increment index of smaller element
         swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
```

Matrix.

Introduction to Matrix

A **matrix** represents a collection of numbers arranged in order of rows and columns. It is necessary to enclose the elements of a matrix in parentheses or brackets.

A matrix with 9 elements is shown below:



The above Matrix M has 3 rows and 3 columns. Each element of matrix [M] can be referred to by its row and column number. For example, $a_{23} = 6$

Order of a Matrix : The order of a matrix is defined in terms of its number of rows and columns.

Order of a matrix = No. of rows × No. of columns

Therefore, Matrix [M] is a matrix of order 3 × 3.

Matrix Implementation

Matrix in programming languages can be implemented using **2-D arrays**. 2-D arrays or Two-Dimensional arrays in simple words can be defined as an *array of arrays*.

Elements in a 2-D array are stored in a tabular form in row major order. A two – dimensional array can be seen as a table with 'x' rows and 'y' columns where the row number ranges from 0 to (x-1) and column number ranges from 0 to (y-1). A two – dimensional array with 3 rows and 3 columns is shown below:

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Declaring 2-D Arrays:

- Syntax for declaring 2-D Array in C++:

data_type array_name[size1][size2]

Where,
data_type: Type of data to be stored in the array.
Here data_type is valid C/C++ data type
array_name: Name of the array
size1: Number of rows
size2: Number of columns

Example:
int arr[2][5];

*The above example, creates a 2-D array named **arr** with 2 rows and 5 columns in **C/C++**.*

- Declaring 2-D array in Java:

data_type[][] array_name = new data_type[size1][size2]

Where,
data_type: Type of data to be stored in the array.
Here data_type is valid Java data type
array_name: Name of the array
size1: Number of rows
size2: Number of columns

Example:
int arr[2][5];

*The above example, creates a 2-D array named **arr** with 2 rows and 5 columns in **Java**.*

Size of 2-D arrays: The total number of elements that can be stored in a 2-D array can be easily calculated by multiplying the size of both dimensions. For Example, the above declared array *arr* can store a maximum of 2*5 = 10 elements.

Accessing 2-D array elements

Elements in two-dimensional arrays are commonly referred by x[i][j] where '*i*' is the row number and '*j*' is the column number.

Syntax:

```
arr[row_index][column_index]
```

For example:

```
arr[0][0] = 1;
```

The above example represents the element present in first row and first column.

Printing all elements of a 2-D array: To print all the elements of a Two-Dimensional array we can use nested for loops. We will require two for loops. One to traverse the rows and another to traverse columns.

Consider a 2-D array named **arr[][]** has **N** rows and **M** columns. Below code snippet traverses all of the elements of the 2-D array in row-major order and prints them:

```
// Traversing number of Rows
for (int i = 0; i < N; i++)
{
   // Traversing number of Columns
   for (int j = 0; j < M; j++)
   {
      // Access each element and print it
      print arr[i][j];
   }
}
```

Hashing.

Hashing - Inroduction

Hashing is a method of storing and retrieving data from a database efficiently.

Suppose that we want to design a system for storing employee records keyed using phone numbers. And we want the following queries to be performed efficiently:

1. Insert a phone number and the corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and the related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. An array of phone numbers and records.
2. A linked list of phone numbers and records.
3. A balanced binary search tree with phone numbers as keys.
4. A direct access table.

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

With a **balanced binary search tree**, we get a moderate search, insert and delete time. All of these operations can be guaranteed to be in O(Logn) time.

Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as indexes in the array. An entry in the array is NIL if the phone number is not present, else the array entry stores pointer to records corresponding to the phone number. Time complexity wise this solution is the best of all, we can do all operations in O(1) time. For example, to insert a phone number, we create a record with details of the given phone number, use the phone number as an index and store the pointer to the record created in the table.
This solution has many practical limitations. The first problem with this solution is that the extra space required is huge. For example, if the phone number is of n digits, we need $O(m * 10^n)$ space for the table where m is the size of a pointer to the record. Another problem is an integer in a programming language may not store n digits.

Due to the above limitations, the Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and performs extremely well as compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing, we get O(1) search time on average (under reasonable assumptions) and O(n) in the worst case.

Hashing is an improvement over Direct Access Table. The idea is to use a hash function that converts a given phone number or any other key to a smaller number and uses the small number as an index in a table called a hash table.

Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as an index in the hash table.
A good hash function should have following properties:

1. It should be efficiently computable.
2. It should uniformly distribute the keys (Each table position be equally likely for each key).

For example, for phone numbers, a bad hash function is to take the first three digits. A better function will consider the last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a big key, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- **Chaining:**The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple, but it requires additional memory outside the table.
- **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one

examine the table slots until the desired element is found or it is clear that the element is not present in the table.

Open Addressing

Open Addressing: Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Important Operations:

- Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- Search(k): Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.
- Delete(k): ***Delete operation is interesting***. If we simply delete a key, then the search may fail. So slots of the deleted keys are marked specially as "deleted".

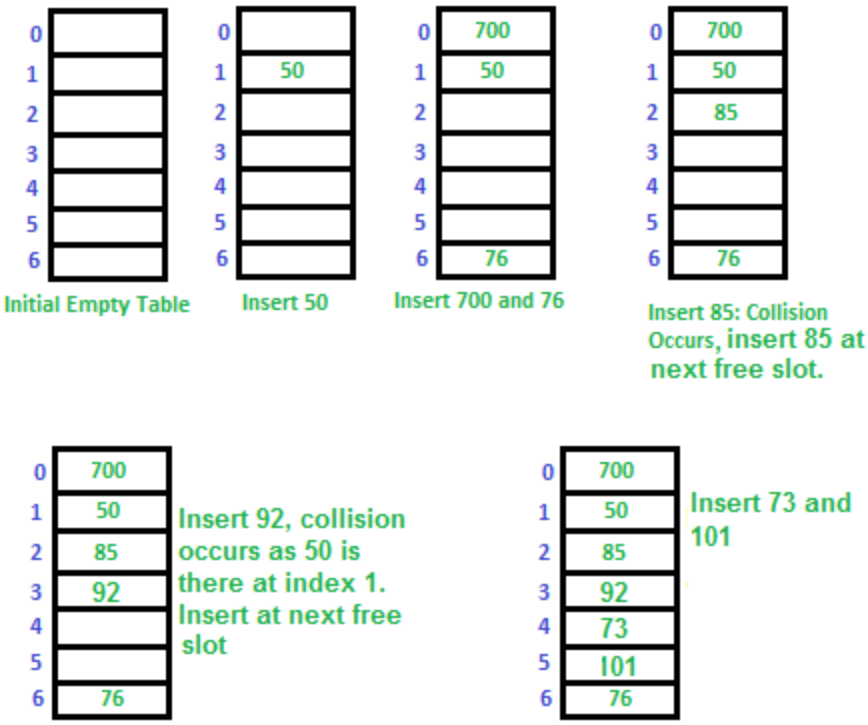
Insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Open Addressing is done in the following ways:

1. ***Linear Probing:*** In linear probing, we linearly probe for the next slot. For example, the typical gap between the two probes is 1 as taken in the below example also.
   let **hash(x)** be the slot index computed using a hash function and **S** be the table size.

```
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
.....................................................
.....................................................
```

   Let us consider a simple hash function as “key mod 7” and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

2. ***Quadratic Probing*** We look for i^2'th slot in i'th iteration.

   let hash(x) be the slot index computed using hash function.
   If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
   If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
   If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S

Hashing in C++ using STL

Hashing in C++ can be implemented using different containers present in STL as per the requirement. Usually, STL offers the below-mentioned containers for implementing hashing:

- set
- unordered_set
- map
- unordered_map

Strings

Introduction to Strings

Strings are defined as a stream of characters. Strings are used to represent text and are generally represented by enclosing text within quotes as: *"This is a sample string!"*.

Different programming languages have different ways of declaring and using Strings. We will learn to implement strings in **C/C++ and Java**.

Strings in C/C++

In C/C++, Strings are defined as an array of characters. The difference between a character array and a string is that the string is terminated with a special character '\0'.

Declaring Strings: Declaring a string is as simple as declaring a one-dimensional array. Below is the basic syntax for declaring a string.

```
char str_name[size];
```

In the above syntax, *str_name* is any name given to the string variable and size is used to define the length of the string, i.e the number of characters strings will store. Please keep in mind that there is an extra terminating character which is the Null character ('\0') used to indicate the termination of string which differs strings from normal character arrays.

Initializing a String: A string can be initialized in different ways. We will explain this with the help of an example. Below is an example to declare a string with the name as *str* and initialize it with **"GeeksforGeeks"**.

1. char str[] = "GeeksforGeeks";

2. char str[50] = "GeeksforGeeks";

3. char str[] = {'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};

4. char str[14] = {'G','e','e','k','s','f','o','r','G','e','e','k','s','\0'};

Printing a string array: Unlike arrays we do not need to print a string, character by character. The C/C++ language does not provide an inbuilt data type for strings but it has an access specifier "**%s**" which can be used to directly print and read strings.

```
// C/C++ program to illustrate string
#include<bits/stdc++.h>
int main()
{
   // declare and initialize string
   char str[] = "Geeks";
   // print string
   printf("%s",str);
   return 0;
}
```

Sample Program:

```
// C++ program to illustrate strings
#include<bits/stdc++.h>
using namespace std;
int main()
{
   // declare and initialize string
   string str = "Geeks";
   // print string
   cout<<str;
   return 0;
}
```

Run

Output:

```
Geeks
```

Linked List.

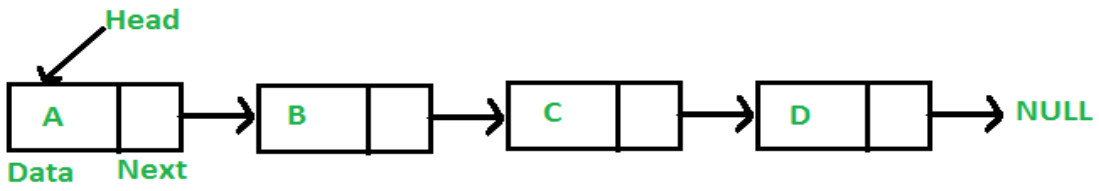
Linked List | Introduction

Linked Lists are linear or sequential data structures in which elements are stored at non-contiguous memory locations and are linked to each other using pointers.

Like arrays, linked lists are also linear data structures but in linked lists elements are not stored at contiguous memory locations. They can be stored anywhere in the memory but for sequential access, the nodes are linked to each other using pointers.

Each element in a linked list consists of two parts:

- **Data**: This part stores the data value of the node. That is the information to be stored at the current node.
- **Next Pointer**: This is the pointer variable or any other variable which stores the address of the next node in the memory.



Advantages of Linked Lists over Arrays: Arrays can be used to store linear data of similar types, but arrays have the following limitations:

1. The size of the arrays is fixed, so we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage. On the other hand, linked lists are dynamic and the size of the linked list can be incremented or decremented during runtime.
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 shift.

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

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

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000). Deletion is also expensive with arrays unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

On the other hand, nodes in linked lists can be inserted or deleted without any shift operation and is efficient than that of arrays.

Disadvantages of Linked Lists:

1. Random access is not allowed in Linked Lists. We have to access elements sequentially starting from the first node. So, we cannot do a binary search with linked lists efficiently with its default implementation. Therefore, lookup or search operation is costly in linked lists in comparison to arrays.
2. Extra memory space for a pointer is required with each element of the list.
3. Not cache-friendly. Since array elements are present at contiguous locations, there is a locality of reference which is not there in the case of linked lists.

Representation of Linked Lists

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head node of the list. If the linked list is empty, then the value of the head node is NULL.

Each node in a list consists of at least two parts:

1. data
2. Pointer (Or Reference) to the next node

In C/C++, we can represent a node using structure. Below is an example of a linked list node with integer data.

```
struct Node
{
   int data;
   struct Node* next;
};
```

Stack.

Introduction to Stacks

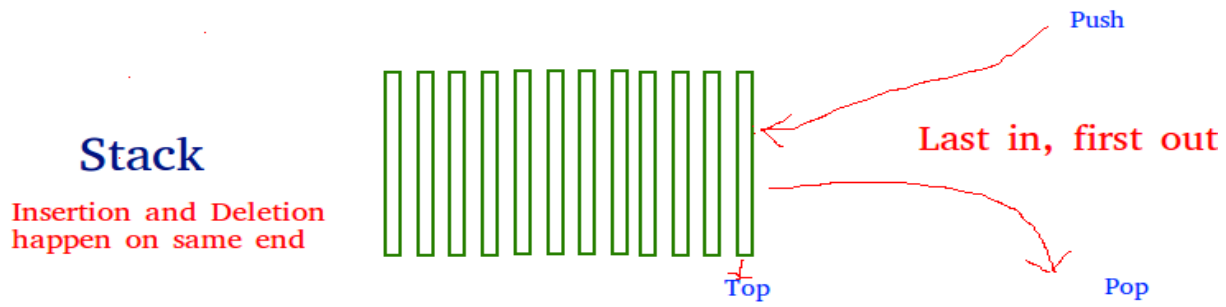
The ***Stack*** is a linear data structure, which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

- The LIFO order says that the element which is inserted at the last in the Stack will be the first one to be removed. In LIFO order, the insertion takes place at the rear end of the stack and deletion occurs at the front of the stack.
- The FILO order says that the element which is inserted at the first in the Stack will be the last one to be removed. In FILO order, the insertion takes place at the rear end of the stack and deletion occurs at the front of the stack.

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 were pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns the top element of the stack.

- **isEmpty:** Returns true if the 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 that is at the top is the first one to be removed, i.e. the plate that 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: The operations push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

Implementation: There are two ways to implement a stack.

- Using array
- Using linked list

Stack in C++ STL

The C++ STL offers a built-in class named **stack** for implementing the stack data structure easily and efficiently. This class provides almost all functions needed to perform the standard stack operations like push(), pop(), peek(), remove() etc..

Syntax:

stack< *data_type* > *stack_name*;

Here,
data_type: This defines the type of data to be
stored in the stack.
stack_name: This specifies the name of the stack.

Some Basic functions of Stack class in C++:

- **empty()** – Returns whether the stack is empty.
- **size()** – Returns the size of the stack.
- **top()** – Returns a reference to the topmost element of the stack.
- **push(g)** – Adds the element 'g' at the top of the stack.
- **pop()** – Deletes the topmost element of the stack.

All of the above functions work in O(1) time complexity.

Queue.

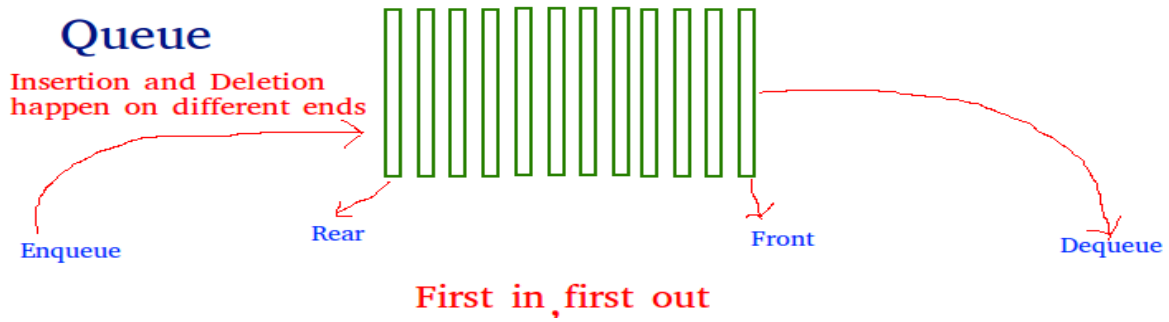
Introduction to Queues

Like *Stack* data structure, ***Queue*** is also a linear data structure that follows a particular order in which the operations are performed. The order is **F**irst **I**n **F**irst **O**ut (FIFO), which means that the element that is inserted first in the queue will be the first one to be removed from the queue. A good example of queue is any queue of consumers for a resource where the consumer who came first is served first.

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

Operations on Queue: Mainly the following four basic operations are performed on queue:

- **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.



Array implementation Of Queue: For implementing a *queue*, we need to keep track of two indices - front and rear. We enqueue an item at the rear and dequeue an item from the front. If we simply increment front and rear indices, then there may be problems, the front may reach the end of the array. The solution to this problem is to increase front and rear in a circular manner.

Consider that an Array of size **N** is taken to implement a queue. Initially, the size of the queue will be zero(0). The total capacity of the queue will be the size of the array i.e. N. Now initially, the

index *front* will be equal to 0, and *rear* will be equal to N-1. Every time an item is inserted, so the index *rear* will increment by one, hence increment it as: **rear = (rear + 1)%N** and everytime an item is removed, so the front index will shift to right by 1 place, hence increment it as: **front = (front + 1)%N**.

Example:

Array = queue[N].
front = 0, rear = N-1.
N = 5.

Operation 1:
enque(5);
front = 0,
rear = (N-1 + 1)%N = 0.
Queue contains: [5].

Operation 2:
enque(10);
front = 0,
rear = (rear + 1)%N = (0 + 1)%N = 1.
Queue contains: [5, 10].

Queue in C++ STL

The Standard template Library in C++ offers a built-in implementation of the Queue data structure for simpler and easy use. The STL implementation of queue data structure implements all basic operations on queue such as enque(), deque(), clear() etc.

Syntax:

*queue< **data_type** > **queue_name**;*

where,
data_type is the type of element to be stored
in the queue.
queue_name is the name of the queue data structure.

The functions supported by std::queue are :

- **empty()** – Returns whether the queue is empty.
- **size()** – Returns the size of the queue.
- **swap()**: Exchange the contents of two queues but the queues must be of same type, although sizes may differ.
- **emplace()**: Insert a new element into the queue container, the new element is added to the end of the queue.
- **front() and back()**: front() function returns a reference to the first element of the queue. back() function returns a reference to the last element of the queue.
- **push(g) and pop()**: The push() function adds the element 'g' at the end of the queue. The pop() function deletes the first element of the queue.

Deque.

Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

Operations on Deque:
Mainly the following four basic operations are performed on queue:

insertFront(): Adds an item at the front of Deque.
insertLast(): Adds an item at the rear of Deque.
deleteFront(): Deletes an item from front of Deque.
deleteLast(): Deletes an item from rear of Deque.
In addition to above operations, following operations are also supported
getFront(): Gets the front item from queue.
getRear(): Gets the last item from queue.
isEmpty(): Checks whether Deque is empty or not.
isFull(): Checks whether Deque is full or not.

Applications of Deque:

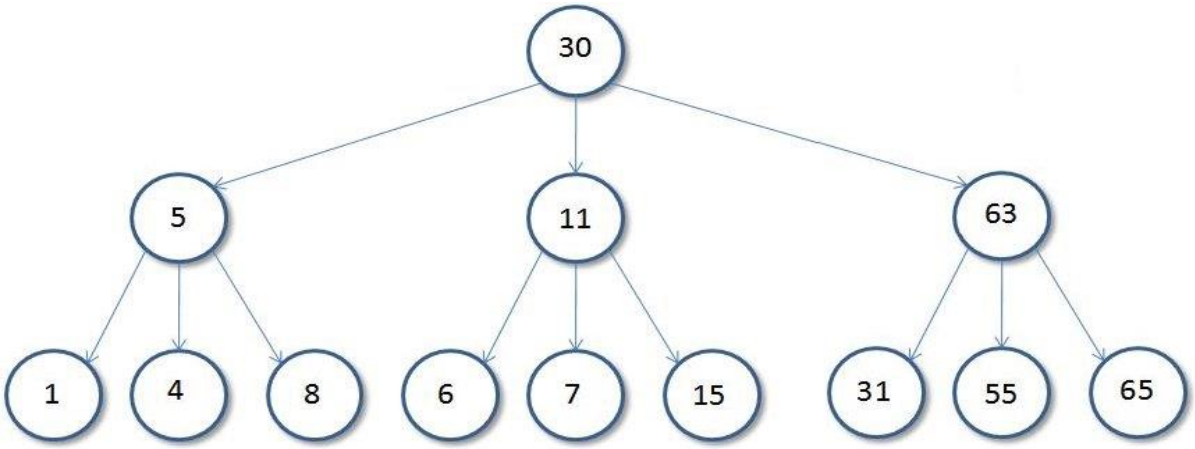
Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications.
Also, the problems where elements need to be removed and or added both ends can be efficiently solved using Deque. For example see Maximum of all subarrays of size k problem., 0-1 BFS and Find the first circular tour that visits all petrol pumps.
A Deque can be implemented either using a doubly linked list or circular array. In both implementation, we can implement all operations in O(1) time. We will soon be discussing C/C++ implementation of Deque Data structure.

Tree.

Introduction to Trees

A Tree is a non-linear data structure where each node is connected to a number of nodes with the help of pointers or references.

A Sample tree is as shown below:



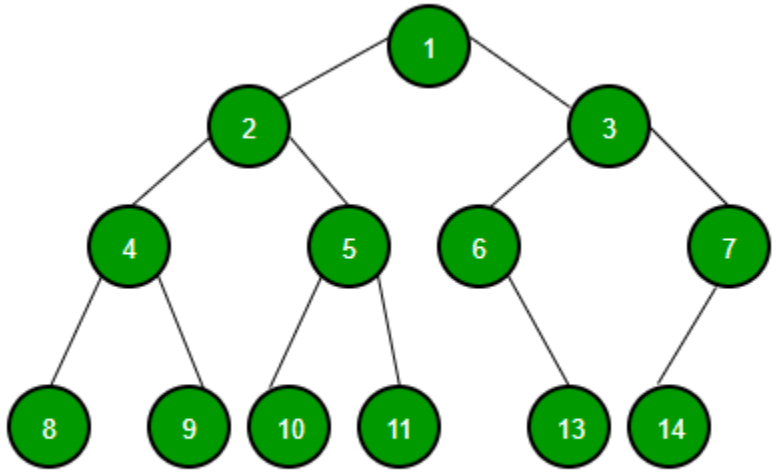
Basic Tree Terminologies:

- **Root**: The root of a tree is the first node of the tree. In the above image, the root node is the node 30.
- **Edge**: An edge is a link connecting any two nodes in the tree. For example, in the above image there is an edge between node **11** and **6**.
- **Siblings**: The children nodes of same parent are called siblings. That is, the nodes with same parent are called siblings. In the above tree, nodes *5, 11, and 63* are siblings.
- **Leaf Node**: A node is said to be the leaf node if it has no children. In the above tree, node **15** is one of the leaf nodes.
- **Height of a Tree**: Height of a tree is defined as the total number of levels in the tree or the length of the path from the root node to the node present at the last level. The above tree is of height **2**.

Binary Tree

A Tree is said to be a Binary Tree if all of its nodes have atmost 2 children. That is, all of its node can have either no child, 1 child, or 2 child nodes.

Below is a sample **Binary Tree**:



Properties of a Binary Tree:

1. ***The maximum number of nodes at level 'l' of a binary tree is*** (2^{l-1}). Level of root is 1.

   This can be proved by induction.
   For root, l = 1, number of nodes = $2^{1-1} = 1$
   Assume that the maximum number of nodes on level l is 2^{l-1}.
   Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^{l-1}$.
2. ***Maximum number of nodes in a binary tree of height 'h' is*** $(2^h - 1)$.
   Here height of a tree is the maximum number of nodes on the root to leaf path. The height of a tree with a single node is considered as 1.
   This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + .. + 2^h-1$. This is a simple geometric series with h terms and sum of this series is $2^h - 1$.
   In some books, the height of the root is considered as 0. In that convention, the above formula becomes $2^{h+1} - 1$.
3. ***In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is*** $Log_2(N+1)$. This can be directly derived from point 2 above. If we consider the

convention where the height of a leaf node is considered 0, then above formula for minimum possible height becomes $Log_2(N+1) – 1$.

4. ***A Binary Tree with L leaves has at least ($Log_2L + 1$) levels.*** A Binary tree has maximum number of leaves (and minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L.

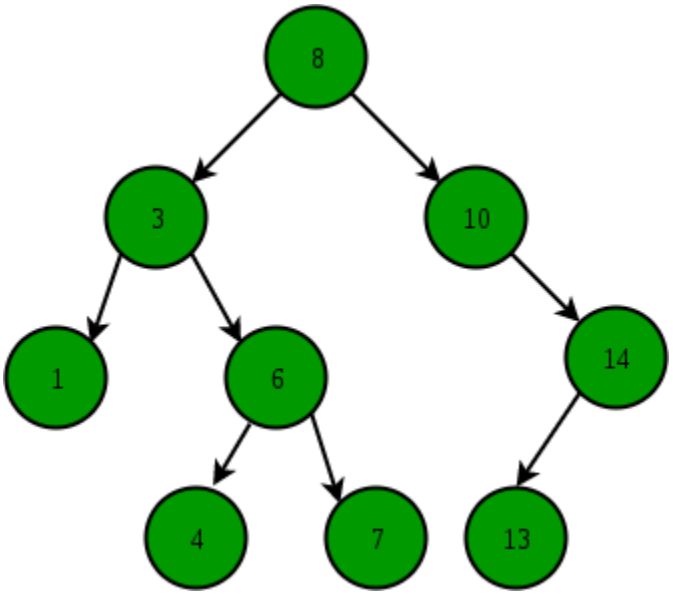
Binary Search Tree.

Introduction to Binary Search Trees

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than or equal to the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
  There must be no duplicate nodes.

Sample Binary Search Tree:



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast in comparison to normal Binary Trees. If there is no ordering, then we may have to compare every key to search a given key.

Searching a Key

Using the property of Binary Search Tree, we can search for an element in O(h) time complexity where **h** is the height of the given BST.

To search a given key in Binary Search Tree, first compare it with root, if the key is present at root, return root. If the key is greater than the root's key, we recur for the right subtree of the root node. Otherwise, we recur for the left subtree.

Implementation:

```
struct node* search(struct node* root, int key)
{
  // Base Cases: root is null or key is present at root
  if (root == NULL || root->key == key)
    return root;

  // Key is greater than root's key
  if (root->key < key)
    return search(root->right, key);
  // Key is smaller than root's key
  return search(root->left, key);
}
```

Heap.

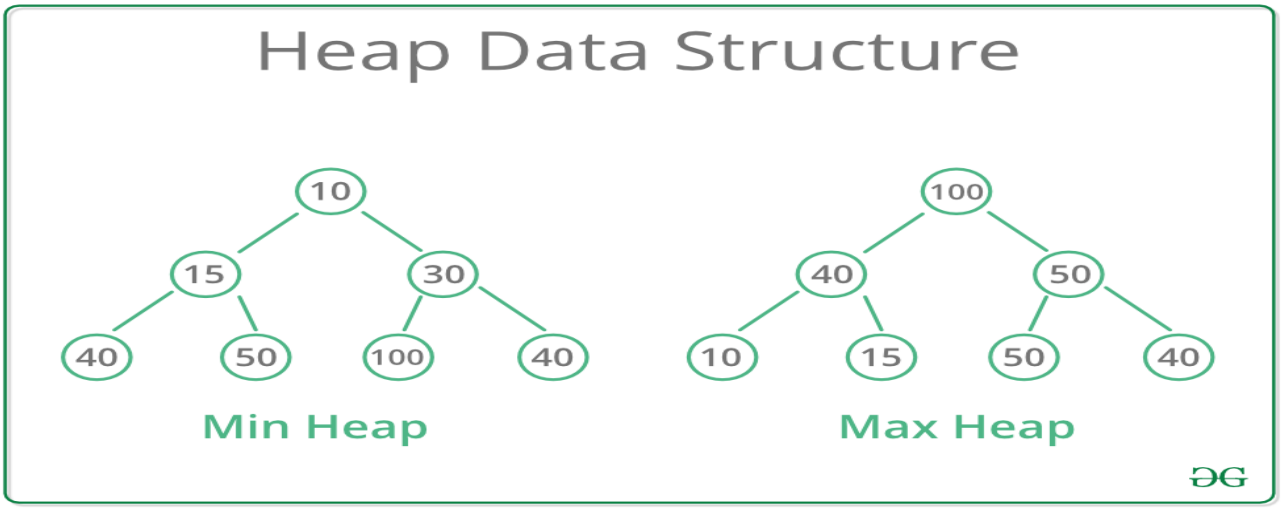
Introduction to Heaps

A Heap is a Tree-based data structure, which satisfies the below properties:

1. A Heap is a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible).

2. A Heap is either Min Heap or Max Heap. In a Min-Heap, the key at root must be minimum among all keys present in the Binary Heap. The same property must be recursively true for all nodes in the Tree. Max Heap is similar to MinHeap.

Binary Heap: A Binary Heap is a heap where each node can have at most two children. In other words, a Binary Heap is a complete Binary Tree satisfying the above-mentioned properties.



Representing Binary Heaps

Since a Binary Heap is a complete Binary Tree, it can be easily represented using Arrays.

- The root element will be at Arr[0].
- Below table shows indexes of other nodes for the i^{th} node, i.e., Arr[i]:

Arr[(i-1)/2] Returns the parent node

Arr[(2*i)+1]Returns the left child node

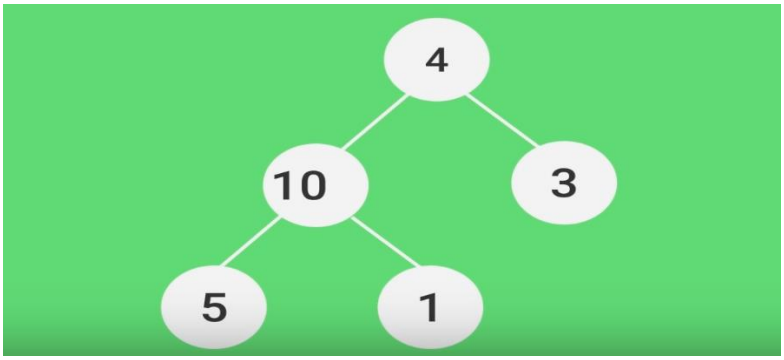
Arr[(2*i)+2]Returns the right child node

Heapifying an Element

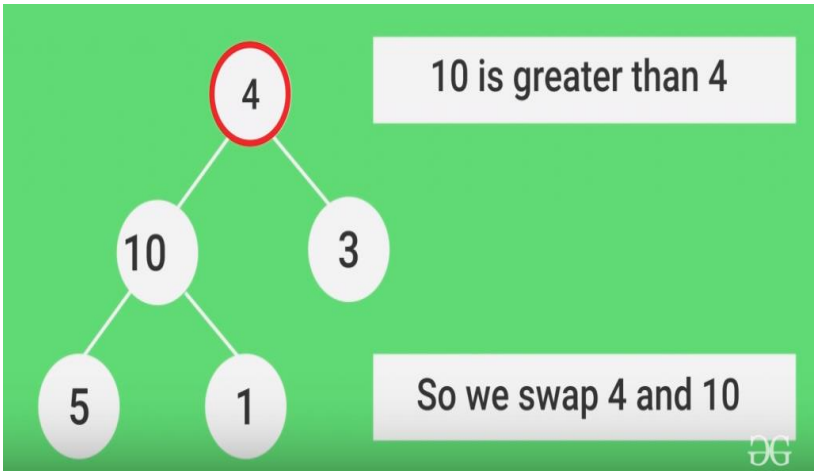
Generally, on inserting a new element onto a Heap, it does not satisfy the property of Heap as stated above on its own. The process of placing the element at the correct location so that it satisfies the Heap property is known as Heapify.

Heapifying in a Max Heap: The property of Max Heap says that every node's value must be greater than the values of its children nodes. So, to **heapify** a particular node swap the value of the node with the maximum value of its children nodes and continue the process until all of the nodes below it satisfies the Heap property.

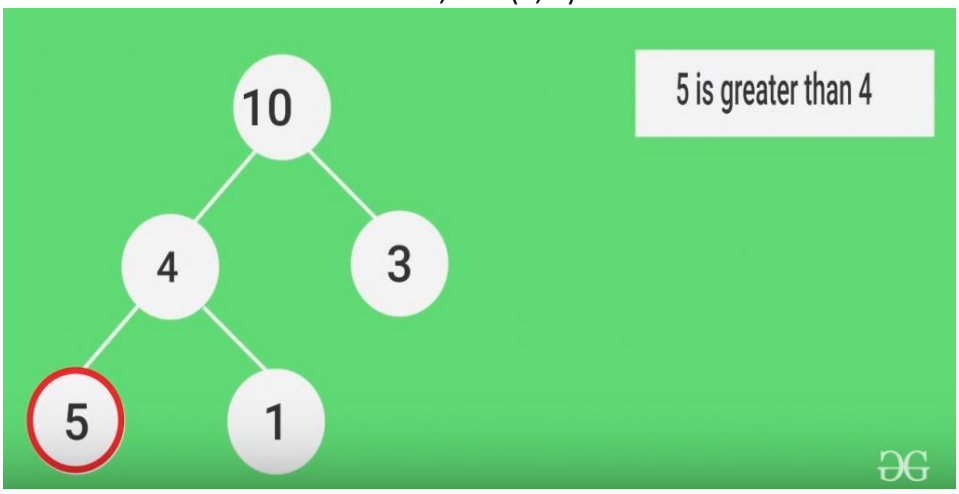
Consider the below heap as a Max-Heap:



In the above heap, node **4** does not follow the Heap property. Let's heapify the root node, ***node 4***.

- **Step 1**: Swap the node 4 with the maximum of its childrens i.e., max(10, 3).
- 

- **Step 2**: Again the node 4 does not follow the heap property. Swap the node 4 with the maximum of its new childrens i.e., max(5, 1).



- The Node 4 is now heapified successfully and placed at it's correct position.

Time Complexity: The time complexity to heapify a single node is **O(h)**, where h is equal to **log(N)** in a complete binary tree where N is the total number of nodes.

Heaps in C++

In C++ the priority_queue container can be used to implement Heaps. The priority_queue in C++ is a built-in container class which is used to implement priority queues easily. This container uses max heap internally to implement a priority queue efficiently. Therefore, it can be also used in-place of max heaps.

This container can also be modified by passing some additional parameters to be used as a Min Heap.

Syntax:

- For implementing Max Heap:

```
priority_queue< type_of_data > name_of_heap;
```

- For implementing Min Heap:

```
priority_queue< type, vector<type>, greater<type> > name_of_heap;
```

Methods of priority queue:

- **priority_queue::empty()**: The empty() function returns whether the queue is empty.
- **priority_queue::size()**: The size() function returns the size of the queue.
- **priority_queue::top()**: This function returns a reference to the top most element of the queue.
- **priority_queue::push()**: The push(g) function adds the element 'g' at the end of the queue.
- **priority_queue::pop()**: The pop() function deletes the first element of the queue.
- **priority_queue::swap()**: This function is used to swap the contents of one priority queue with another priority queue of same type and size.

Implementation:

```
// C++ program to implement Max Heap and Min Heap

// using priority_queue in C++ STL

#include <iostream>

#include <queue>

using namespace std;

int main ()

{

   // DECLARING MAX HEAP

   priority_queue <int> max_heap;


   // Add elements to the Max Heap

   // in any order

   max_heap.push(10);

   max_heap.push(30);

   max_heap.push(20);

   max_heap.push(5);

   max_heap.push(1);

   // Print element at top of the heap
```

```
  // every time and remove it until the
  // heap is not empty
  cout<<"Element at top of Max Heap at every step:\n";
  while(!max_heap.empty())
  {
     // Print Top Element
     cout<<max_heap.top()<<" ";
```

Run

Output:

```
Element at top of Max Heap at every step:
30 20 10 5 1

Element at top of Min Heap at every step:
1 5 10 20 30
```

Graph.

Introduction to Graphs

A ***Graph*** is a data structure that consists of the following two components:

1. A finite set of vertices also called nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(digraph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

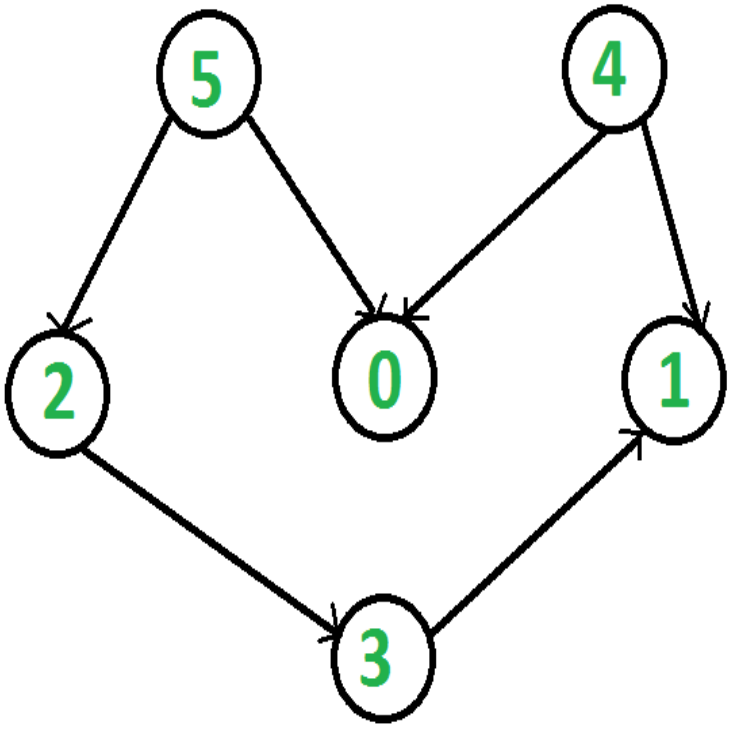
Graphs are used to represent many real-life applications:

- Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. For example Google GPS
- Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale.

Directed and Undirected Graphs

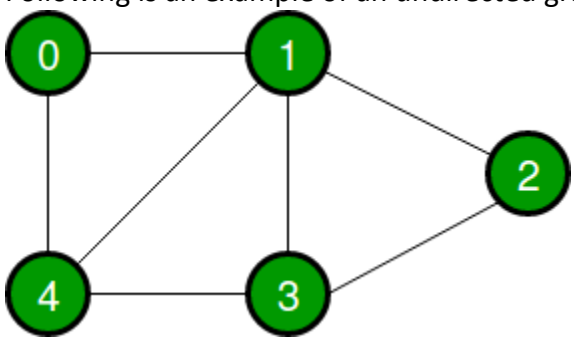
- **Directed Graphs**: The Directed graphs are such graphs in which edges are directed in a single direction.

  For Example, the below graph is a directed graph:



- **Undirected Graphs**: Undirected graphs are such graphs in which the edges are directionless or in other words bi-directional. That is, if there is an edge between vertices **u** and **v** then it means we can use the edge to go from both **u to v** and **v to u**.

Following is an example of an undirected graph with 5 vertices:



Representing Graphs

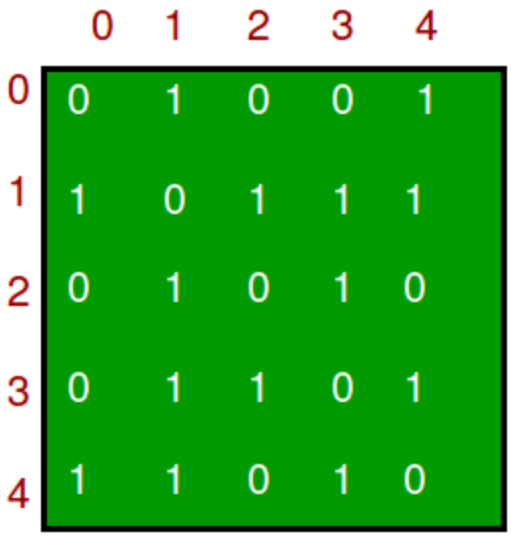
Following two are the most commonly used representations of a graph:

1. Adjacency Matrix.
2. Adjacency List.

Let us look at each one of the above two method in details:

- **Adjacency Matrix:** The Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

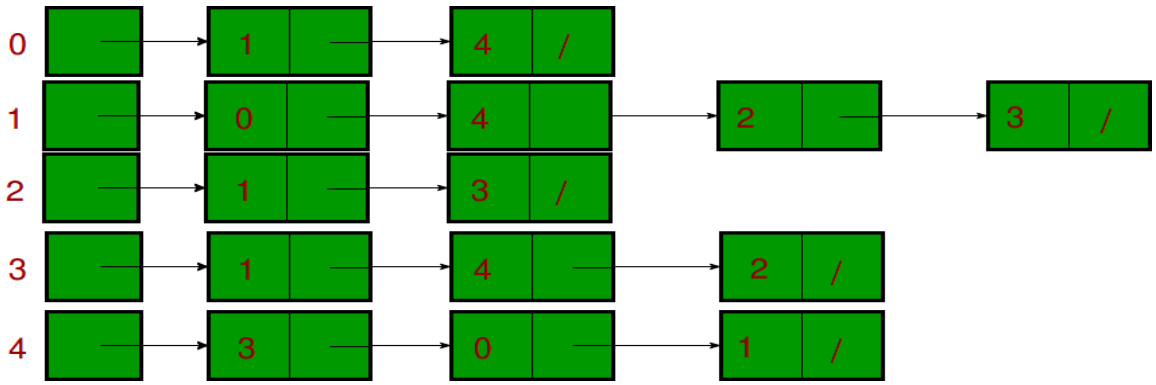
  The adjacency matrix for the above example undirected graph is:



***Pros**:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done O(1).

***Cons**:* Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.
Please see this for a sample Python implementation of adjacency matrix.

- **Adjacency List:** Graph can also be implemented using an array of lists. That is every index of the array will contain a complete list. Size of the array is equal to the number of vertices and every index **i** in the array will store the list of vertices connected to the vertex numbered *i*. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above example undirected graph.



Below is the implementation of the adjacency list representation of Graphs:

```
// A simple representation of graph using STL

#include<bits/stdc++.h>

using namespace std;

// A utility function to add an edge in an

// undirected graph.

void addEdge(vector<int> adj[], int u, int v)

{

   adj[u].push_back(v);

   adj[v].push_back(u);

}

// A utility function to print the adjacency list

// representation of graph

void printGraph(vector<int> adj[], int V)

{

   for (int v = 0; v < V; ++v)

   {
```

```
        cout << "\n Adjacency list of vertex "
             << v << "\n head ";
        for (auto x : adj[v])
          cout << "-> " << x;
        printf("\n");
    }
}
// Driver code
int main()
{
    int V = 5;
```

Output:

```
Adjacency list of vertex 0
head -> 1-> 4

Adjacency list of vertex 1
head -> 0-> 2-> 3-> 4

Adjacency list of vertex 2
head -> 1-> 3

Adjacency list of vertex 3
head -> 1-> 2-> 4

Adjacency list of vertex 4
head -> 0-> 1-> 3
```

Pros: Saves space O(|V|+|E|). In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier.

***Cons**:* Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

Breadth First Traversal of a Graph

The ***Breadth First Traversal*** or ***BFS*** traversal of a graph is similar to that of the Level Order Traversal of Trees.

The BFS traversal of Graphs also traverses the graph in levels. It starts the traversal with a given vertex, visits all of the vertices adjacent to the initially given vertex and pushes them all to a queue in order of visiting. Then it pops an element from the front of the queue, visits all of its neighbours and pushes the neighbours which are not already visited into the queue and repeats the process until the queue is empty or all of the vertices are visited.

The BFS traversal uses an auxiliary boolean array say *visited[]* which keeps track of the visited vertices. That is if **visited[i] = true** then it means that the **i-th** vertex is already visited.

Complete Algorithm:

1. Create a boolean array say ***visited[]*** of size **V+1** where *V* is the number of vertices in the graph.
2. Create a Queue, mark the source vertex visited as **visited[s] = true** and push it into the queue.
3. Until the Queue is non-empty, repeat the below steps:
    - Pop an element from the queue and print the popped element.
    - Traverse all of the vertices adjacent to the vertex poped from the queue.
    - If any of the adjacent vertex is not already visited, mark it visited and push it to the queue.

Greedy.

Introduction to Greedy Algorithms

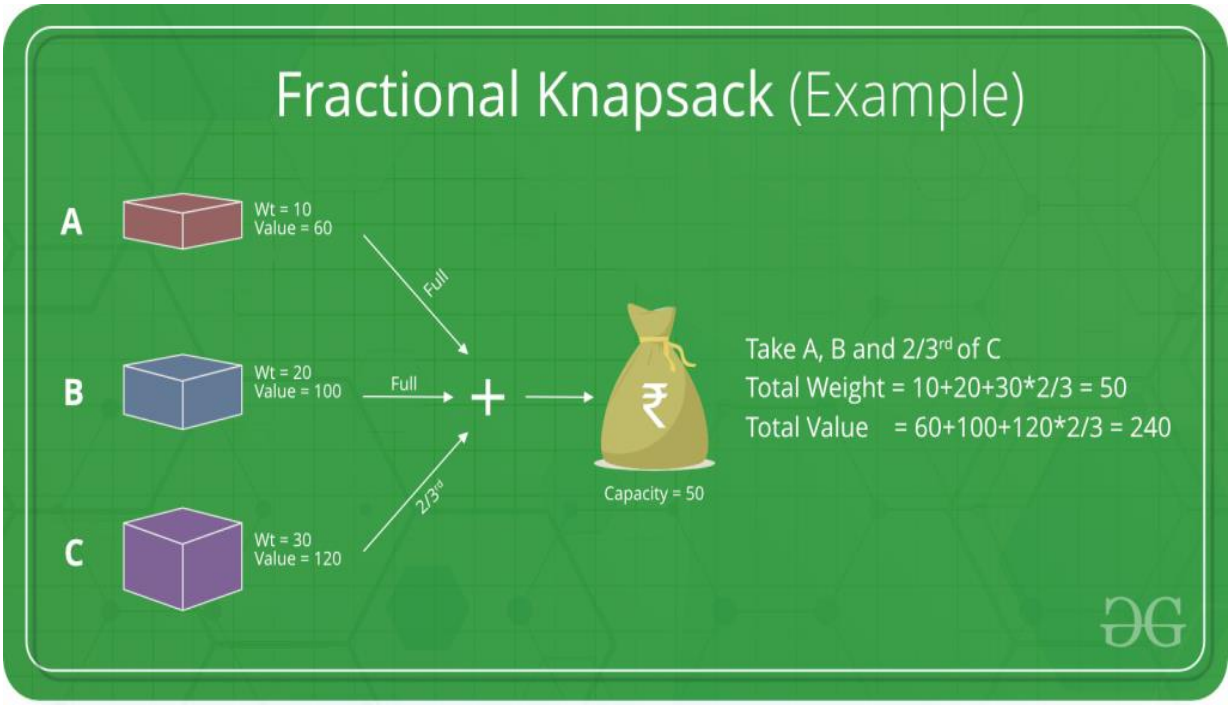
Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to the global optimal solution are best fit for Greedy.

For example, consider the **Fractional Knapsack Problem**. The problem states that:

Given a list of elements with specific values and weights associated with them, the task is to fill a Knapsack of weight W using these elements such that the value of knapsack is maximum possible.

***Note**: You are allowed to take a fraction of an element also in order to maximize the value.*

The local optimal strategy is to choose the item that has maximum value vs weight ratio. This strategy also leads to global optimal solution because we are allowed to take fractions of an item.



In general, the **Greedy Algorithm** can be applied to solve a problem if it satisfies the below property:

At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.
Let us consider one more problem known as the **Activity Selection Problem** to understand the use of Greedy Algorithms.

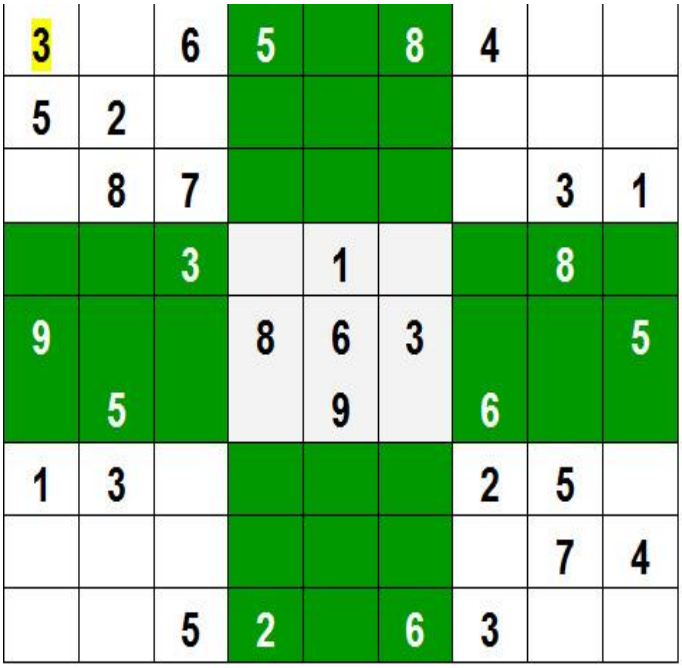
Backtracking.

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

For example, consider the SudoKo solving Problem, we try filling digits one by one.

Whenever we find that current digit cannot lead to a solution, we remove it (backtrack) and try next digit.

This is better than naive approach (generating all possible combinations of digits and then trying every combination one by one) as it drops a set of permutations whenever it backtracks.



Dynamic Programming

Introduction to Dynamic Programming

Dynamic Programming is an algorithmic approach to solve some complex problems easily and save time and number of comparisons by storing the results of past computations. The basic idea of dynamic programming is to store the results of previous calculation and reuse it in future instead of recalculating them.

We can also see Dynamic Programming as dividing a particular problem into subproblems and then storing the result of these subproblems to calculate the result of the actual problem.

Consider the problem to ***find the N-th Fibonacci number***.

We know that n-th fibonacci number fib(n) can be defined as:

```
fib(n) = fib(n-1) + fib(n-2), where n >= 2.

and,

fib(0) = 0
fib(1) = 1
```

We can see that the above function fib() to find the nth fibonacci number is divided into two subproblems fib(n-1) and fib(n-2) each one of which will be further divided into subproblems and so on.

The first few Fibonacci numbers are:

1, 1, 2, 3, 5, 8, 13, 21, 34,.....

The recursive program to find N-th Fibonacci number is shown below:

```
/* simple recursive program for Fibonacci numbers */

int fib(int n)

{
```

```
  if ( n <= 1 )
    return n;
  return fib(n-1) + fib(n-2);
}
```

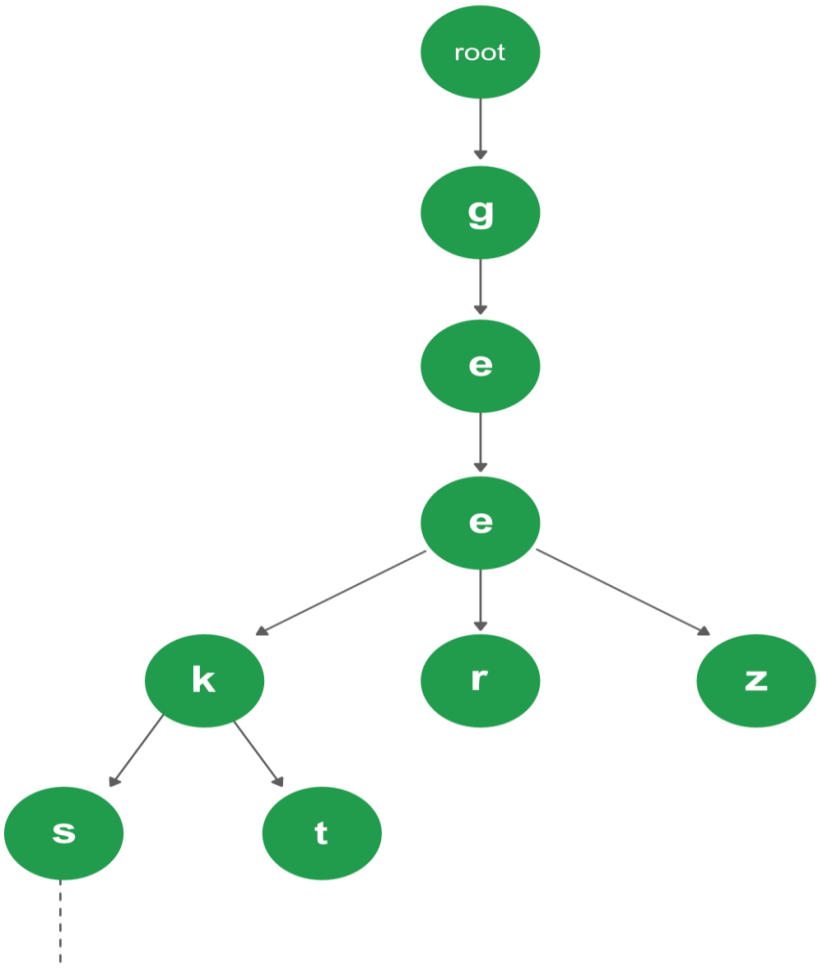
Recursion tree for execution of fib(5):

Trie

The Trie data structure is an efficient information **re-*trie*-val** data structure. The Trie data struture is used to efficiently search for a particular string key among a list of such keys. Using the trie, the lookup operation can be performed in time complexity of **O(key_length)**.

Representation:
A trie is represented as a tree where each node contains 26 pointers which is equal to the number of characters in the English alphabets. In Trie basically, the common prefix of all strings are represented as a common path in the tree.

For Example, look at the below sample Trie:



There are 4 different strings in the Trie above: ["geeks", "geekt", "geer", "geez"]. Since the prefix **"gee"** is common among all of the strings, so it is a common path in the Trie as well. Again for the strings "geeks" and "geekt", the common prefix is "geek", so after dividing paths from "gee", the node with character "k" is common to these two strings.

Insertion in a Trie

Inserting a key to **Trie** is a simple approach. Every character of the input key is inserted as an individual Trie node. Note that the children are an array of pointers (or references) to next level trie nodes. The key character acts as an index into the array of children. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth.

Searching a Key in Trie

Searching for a key is similar to the insert operation, however, we only compare the characters and move down. The search can terminate due to the end of a string or lack of key in the trie. In the former case, if the *isEndofWord* field of the last node is true, then the key exists in the trie. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.

Segment and Binary Indexed Trees.

Introduction to Segment Trees

Segment Trees are **Binary Tree** which is used to store intervals or segments. That is each node in a segment tree basically stores the segment of an array. Segment Trees are generally used in problems where we need to solve queries on a range of elements in arrays.

Let us consider the following problem to understand Segment Trees.

Problem: We have an array arr[0 . . . n-1]. We should be able to perform the below operations on the array:

1. Find the sum of elements from index l to r where 0 <= l <= r <= n-1.
2. Change value of a specified element of the array to a new value x. We need to do arr[i] = x where 0 <= i <= n-1.

General Solution: A simple solution is to run a loop from index **l** to **r** and calculate the sum of elements in the given range. To update a value, simply do arr[i] = x. The first operation takes O(N) time for every query, where N is the number of elements in the range [l,r] and the second operation takes O(1) time.

Can we optimize the time complexity of the first operation in the above solution?

Yes, we can optimize the first operation to be solved in O(1) time complexity by storing **presum**. We can keep an auxiliary array say *sum[]* in which the **i-th** element will store the sum of

first **i** elements of the original array. So, whenever we need to find the sum of a range of elements, we can simply calculate it by (**sum[r]-sum[l-1]**). But in this solution the complexity to perform the second operation of updating an element increases from **O(1) to O(N)**.

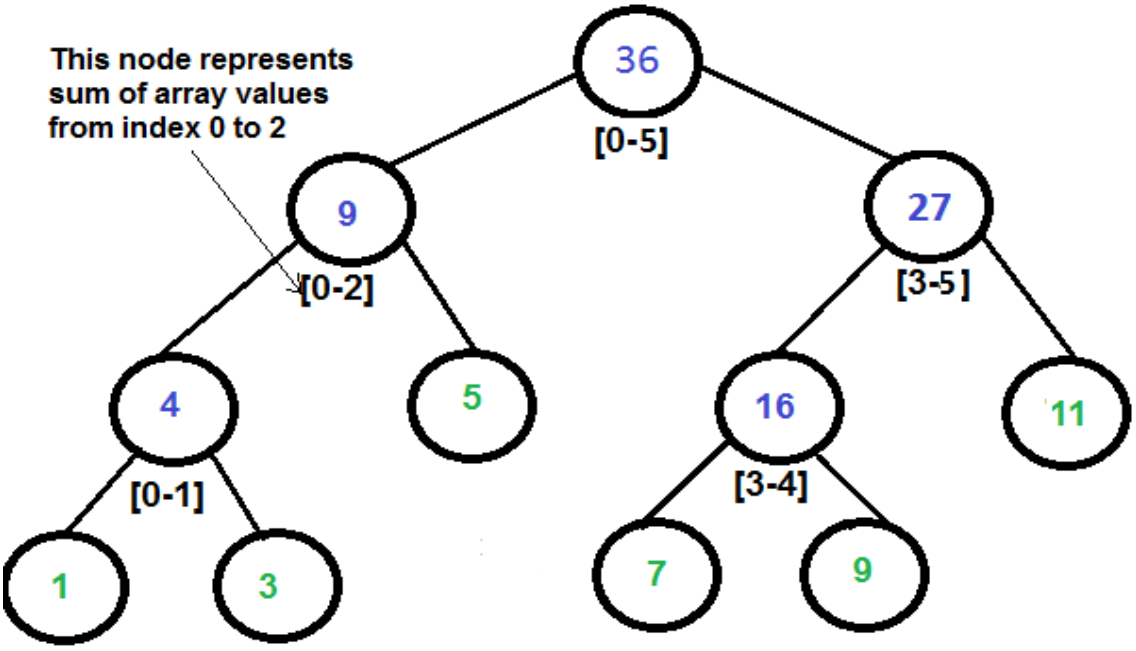
What if the number of query and updates are equal? Can we perform both the operations in O(log n) time once given the array?

We can use a segment tree to perform both of the operations in O(log N) time complexity.

Representation of Segment trees:

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is the sum of leaves under a node.

An array representation of the tree is used to represent Segment Trees. For each node at index **i**, the left child is at index **2*i + 1**, right child at **2*i + 2** and the parent is at **(i - 1)/2**.



Segment Tree for input array {1, 3, 5, 7, 9, 11}

Array representation of Segment Trees: Like Heap, segment tree is also represented as an array. The difference here is, it is not a complete binary tree. It is rather a full binary tree (every node has 0 or 2 children) and all levels are filled except possibly the last level. Unlike Heap, the last level may have gaps between nodes. Below are the values in the segment tree array for the above diagram.

Array representation of segment tree for input array {1, 3, 5, 7, 9, 11} is,

st[] = {36, 9, 27, 4, 5, 16, 11, 1, 3, DUMMY, DUMMY, 7, 9, DUMMY, DUMMY}

The dummy values are never accessed and have no use. This is some wastage of space due to simple array representation. We may optimize this wastage using some clever implementations, but the code for sum and update becomes more complex.

Construction of Segment Tree from the given array: We start with a segment arr[0 . . . n-1] and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the sum in the corresponding node.

All levels of the constructed segment tree will be completely filled except the last level. Also, the tree will be a Full Binary Tree because we always divide segments into two halves at every level. Since the constructed tree is always a full binary tree with n leaves, there will be n-1 internal nodes. So the total number of nodes will be 2*n – 1. Note that this does not include dummy nodes.

What is the total size of the array representing segment tree? If n is a power of 2, then there are no dummy nodes. So the size of the segment tree is 2n-1 (n leaf nodes and n-1) internal nodes. If n is not a power of 2, the size of the tree will be 2*x – 1 where x is the smallest of 2 greater than n. For example, when n = 10, then size of array representing segment tree is 2*16-1 = 31.

Disjoint Set.

Disjoint Set Data Structure

Introduction

A **disjoint-set** data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

For Example:

*Consider that there are 5 students in a classroom namely, **A, B, C, D, and E**.*

*They will be denoted as 5 different subsets: **{A}, {B}, {C}, {D}, {E}**.*

After some point of time, **A** became friends with **B**, and **C** became friend with **D**. So, A and B will now belong to a same set and C and D will now belong to another same set.

The disjoint data structure will now be: **{A, B}, {C, D}, {E}**.

If at any point in time, we want to check that if any two students are friends or not, then we can simply check whether they belong to the same set.

As explained above, there are generally two types of operations performed on a Disjoint-Set data structure:

- **Union(A, B)**: This operation tells to merge the sets containing elements A and B respectively by performing a Union operation on the sets.
- **Find(A)**: This operation tells to find the subset to which the element A belongs.

Implementation: The disjoint set data structure can be implemented using a **Parent array** representation. If we are dealing with n items, i'th element of the array represents the i'th item. More precisely, the i'th element of the array is the parent of the i'th item.

- **Implementing Find Operation**: It can be implemented by recursively traversing the parent array until we hit a node who is the parent of itself.

```
// Finds the representative of the set
// that i is an element of
int find(int i)
{
   // If i is the parent of itself
   if (parent[i] == i)
   {
      // Then i is the representative of
      // this set
      return i;
   }
   else
   {
      // Else if i is not the parent of
      // itself, then i is not the
      // representative of his set. So we
      // recursively call Find on its parent
      return find(parent[i]);
   }
}
```

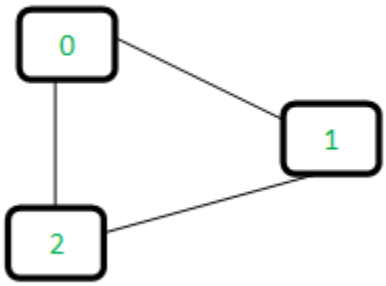
- **Implementing Union Operation**: It takes as input two elements and finds the representatives of their sets using the find operation, and then finally puts either one of the trees (representing the set) under the root node of the other tree, effectively merging the trees and the sets.

```
// Unites the set that includes i
// and the set that includes j
void union(int i, int j)
{
   // Find the representatives
   // (or the root nodes) for the set
   // that includes i
   int irep = find(i),

   // And do the same for the set
   // that includes j
   int jrep = find(j);
```

```
    // Make the parent of i's representative
    // be j's  representative effectively
    // moving all of i's set into j's set)
    Parent[irep] = jrep;
}
```

Application: There are a lot of applications of Disjoint-Set data structure. Consider the problem of **detecting a cycle in a Graph**. It can be easily solved using the Disjoint Set and Union-Find algorithm. This method assumes that the graph does not contain any self-loop.

Let us consider the following graph:



For each edge, make subsets using both the vertices of the edge. If both the vertices are in the same subset, a cycle is found.