Wolaita Sodo University

School of Informatics

Department of Computer Science

Data Structure and Algorithm Training Module

Compiled by Arba Asha (MSc.)

Reveiewed by :

Melaku Bayih(MSc.)

Dawit Uta(MTech.)

Mesay Wana(MSc.)

April, 2023

Wolaita Sodo, Ethiopia

Table of Contents

Chapter One

Introduction to Data Structures and Algorithms

Course Description

This course focuses on the study of data structures, algorithms and program efficiency. Topics Include: analysis of time and space requirements of algorithms; program efficiency improving techniques, abstract data types such as linked lists, stacks, queues, trees (traversal, implementations); simple searching algorithms (linear search, binary search, ...), simple sorting algorithms (bubble sort, insertion sort, selection sort, ...), advanced sorting algorithms (merge sort, quick sort, heap sort ...)

Course Objectives

- To introduce the most common data structures like stack, queue, linked list
- To give alternate methods of data organization and representation
- To enable students use the concepts related to Data Structures and Algorithms to solve real world problems
- To practice Recursion, Sorting, and searching on the different data structure
- To implement the data structures with a chosen programming language

1. Introduction

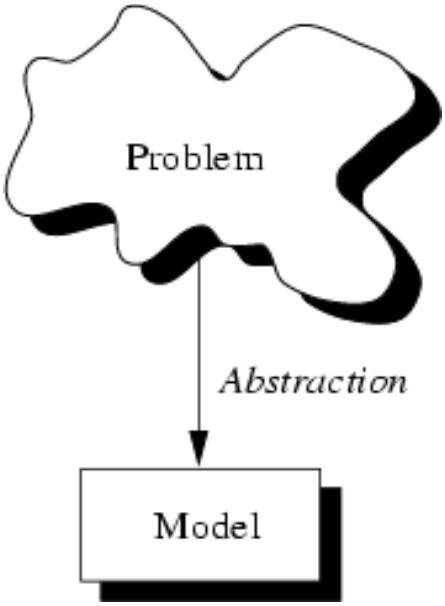
A **program** is written in order to solve a problem. A solution to a problem actually consists of two things:

- A way to organize the data
- Sequence of steps to solve the problem

The way data are organized in a computer's memory is said to be **Data Structure** and the sequence of computational steps to solve a problem is said to be an **algorithm**. Therefore, a program is nothing but data structures plus an algorithm.

1.1. Introduction to Data Structures

Given a problem, the first step to solve the problem is obtaining one's own abstract view, or *model*, of the problem. This process of modeling is called ***abstraction***.



The model defines an abstract view to the problem. This implies that the model focuses only on problem related stuff and that a programmer tries to define the *properties* of the problem.

These properties include

- The *data* which are affected and
- The *operations* that are involved in the problem.

With abstraction you create a well-defined entity that can be properly handled. These entities define the *data structure* of the program.

An entity with the properties just described is called an *abstract data type* (ADT).

1.1.1. Abstract Data Types

An ADT consists of an abstract data structure and operations. Put in other terms, an ADT is an abstraction of a data structure.

The ADT specifies:

1. What can be stored in the Abstract Data Type
2. What operations can be done on/by the Abstract Data Type.

For example, if we are going to model employees of an organization:

- This ADT stores employees with their relevant attributes and discarding irrelevant attributes.
- This ADT supports hiring, firing, retiring, … operations.

A data structure is a language construct that the programmer has defined in order to implement an abstract data type.

There are lots of formalized and standard Abstract data types such as:

- ✓ Stacks,
- ✓ Queues,
- ✓ Trees, etc.

Do all characteristics need to be modeled?

Not at all

- It depends on the scope of the model
- It depends on the reason for developing the model

1.1.2. Abstraction

Abstraction is a process of classifying characteristics as relevant and irrelevant for the particular purpose at hand and ignoring the irrelevant ones.

Applying abstraction correctly is the essence of successful programming.

How do data structures model the world or some part of the world?

- The value held by a data structure represents some specific characteristic of the world
- The characteristic being modeled restricts the possible values held by a data structure
- The characteristic being modeled restricts the possible operations to be performed on the data structure.

Data Structure Operations

- Searching
  Finding the location of the data element (key) in the structure
- Insertion
  - Adding a new data element to the structure
- Deletion
  - Removing a data element from the structure
- Sorting
  - Arrange the data elements in a logical order (ascending/descending)
- Merging
  - Combining data elements from two or more data structures into one.

Classification of Data Structure

We may classify these data structures as linear and non-linear data structures. However, this is not the only way to classify data structures. In linear data structure the data items are arranged in a linear sequence like in an array. In a non-linear, the data items are not in sequence. An example of a non-linear data structure is a tree.

Data structures may also be classified as homogenous and non- homogenous data structures. An Array is a homogenous structure in which all elements are of same type. In non-homogenous structures the elements may or may not be of the same type. Records are common example of non-homogenous data structures.

Another way of classifying data structures is as static or dynamic data structures. Static structures are ones whose sizes and structures associated memory location are fixed at compile time. Dynamic structures are ones, which expand or shrink as required during the program execution and their associated memory locations change. Records are a common example of non-homogenous data structures.

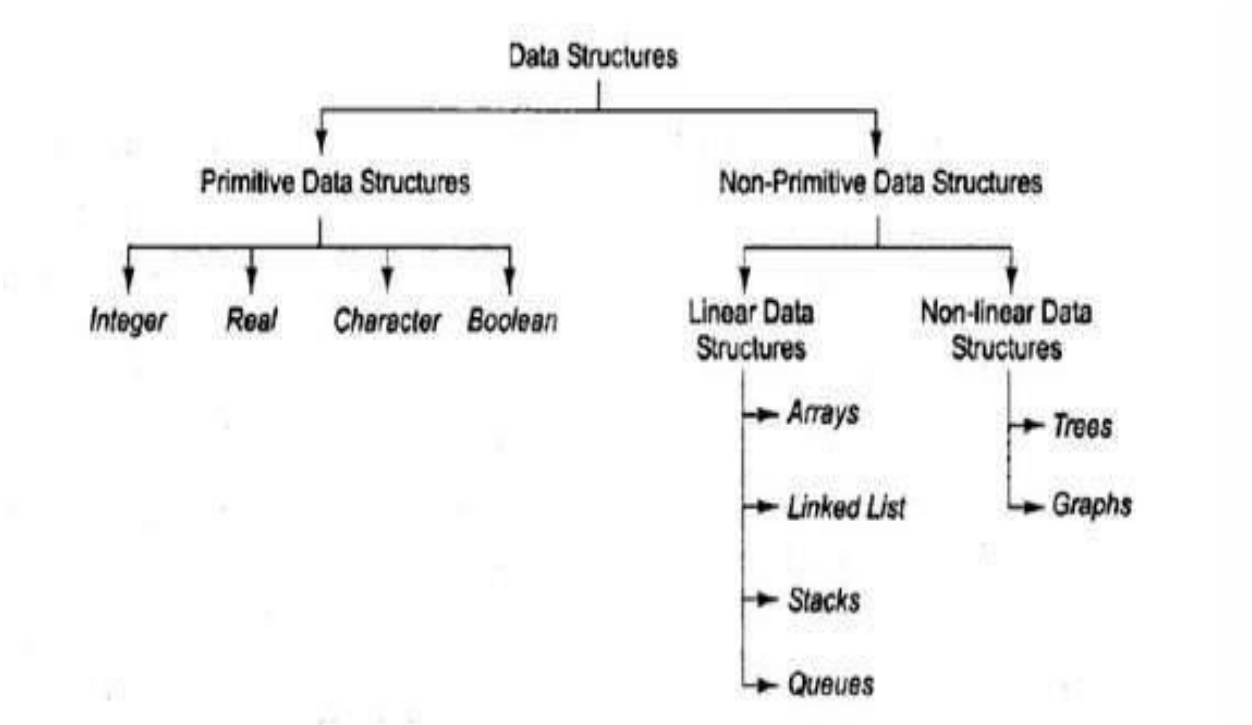


Fig: classification of data structure.

1.2. Algorithm and algorithm analysis

Algorithm

An algorithm is a well-defined computational procedure that takes some value or a set of values as input and produces some value or a set of values as output. Data structures model the static part of the world. They are unchanging while the world is changing. In order to model the dynamic part of the world we need to work with algorithms. Algorithms are the dynamic part of a program's world model.

An algorithm transforms data structures from one state to another state in two ways:

- An algorithm may change the value held by a data structure
- An algorithm may change the data structure itself

The *quality* of a data structure is related to its ability to successfully model the characteristics of the world. Similarly, the *quality* of an algorithm is related to its ability to successfully simulate the changes in the world.

However, independent of any particular world model, the quality of data structure and algorithms is determined by their ability to work together well. Generally speaking, correct data structures lead to simple and efficient algorithms and correct algorithms lead to accurate and efficient data structures.

1. 2.1. Properties of an algorithm

- **Finiteness**: Algorithm must complete after a finite number of steps.
- **Definiteness**: Each step must be clearly defined, having one and only one interpretation. At each point in computation, one should be able to tell exactly what happens next.
- **Sequence**: Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted.
- **Feasibility**: It must be possible to perform each instruction.
- **Correctness**: It must compute correct answer for all possible legal inputs.
- **Language Independence**: It must not depend on any one programming language.
- **Completeness**: It must solve the problem completely.
- **Effectiveness**: It must be possible to perform each step exactly and in a finite amount of time.

- **Efficiency**: It must solve with the least amount of computational resources such as time and space.
- **Generality**: Algorithm should be valid on all possible inputs.
- **Input**/**Output**: There must be a specified number of input values, and one or more result values.
- ***It must terminate*** (it may not go in to an infinite loop).

1.2.2. Algorithm Analysis Concepts

Algorithm analysis refers to the process of determining the amount of computing time and storage space required by different algorithms. In other words, it's a process of predicting the resource requirement of algorithms in a given environment.

In order to solve a problem, there are many possible algorithms. One has to be able to choose the best algorithm for the problem at hand using some scientific method. To classify some data structures and algorithms as good, we need precise ways of analyzing them in terms of *resource requirement*. The main resources are:

- Running Time (most important)
- Memory Usage
- Communication Bandwidth

Running time is usually treated as the most important since computational time is the most precious resource in most problem domains.

There are two approaches to measure the efficiency of algorithms:

- Empirical: Programming competing algorithms and trying them on different instances.
- Theoretical: Determining the quantity of resources required mathematically (Execution time, memory space, etc.) needed by each algorithm.

However, it is difficult to use actual clock-time as a consistent measure of an algorithm's efficiency, because clock-time can vary based on many things. For example,

- Specific processor speed
- Current processor load
- Specific data for a particular run of the program
    - Input Size
    - Input Properties
- Operating Environment

Accordingly, we can analyze an algorithm according to the number of operations required, rather than according to an absolute amount of time involved. This can show how an algorithm's efficiency changes according to the size of the input.

1.2.3 Complexity Analysis

Complexity Analysis is the systematic study of the cost of computation, measured either in time units or in operations performed, or in the amount of storage space required.

The goal is to have a meaningful measure that permits comparison of algorithms independent of operating platform.

There are two things to consider:

- **Time Complexity**: Determine the approximate number of operations required to solve a problem of size n.
- **Space Complexity:** Determine the approximate memory required to solve a problem of size n.

Complexity analysis involves two distinct phases:

- **Algorithm Analysis**: Analysis of the algorithm or data structure to produce a function T (n) that describes the algorithm in terms of the operations performed in order to measure the complexity of the algorithm.
- **Order of Magnitude Analysis**: Analysis of the function T (n) to determine the general complexity category to which it belongs.

There is no generally accepted set of rules for algorithm analysis. However, an exact count of operations is commonly used.

i. Analysis Rules:

1. We assume an arbitrary time unit.

2. Execution of one of the following operations takes time 1:

    - Assignment Operation
    - Single Input/output Operation
    - Single Boolean Operations
    - Single Arithmetic Operations

- Function Return

3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
4. Loops: Running time for a loop is equal to the running time for the statements inside the loop * number of iterations.
   The total running time of a statement inside a group of nested loops is the running time of the statements multiplied by the product of the sizes of all the loops.

   For nested loops, analyze inside out.

   - Always assume that the loop executes the maximum number of iterations possible.
5. Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

Examples:

1. int count(){=1

   int k=0; =1

   cout<< "Enter an integer";=1

   cin>>n;=1

   1 1+n n

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

   k=k+1;2n

   return 0;}1

Time Units to Compute

---

1 for the assignment statement: int k=0

1 for the output statement.

1 for the input statement.

In the for loop:

1 assignment, *n+1* tests, and *n* increments.

n loops of 2 units for an assignment, and an addition.

1 for the return statement.

T (n)= *1+1+1+(1+n+1+n)+2n+1 = 4n+6 = O(n)*

2. int total(int n)

```
{
int sum=0;
for (int i=1;i<=n;i++)
    sum=sum+1;
 return sum;
 }
```

Time Units to Compute

---

1 for the assignment statement: int sum=0

In the for loop:

1 assignment, *n+1* tests, and *n* increments.

n loops of 2 units for an assignment, and an addition.

1 for the return statement.

T (n)= *1+ (1+n+1+n)+2n+1 = 4n+4 = O(n)*

3. void func()

```
{
int x=0;
int i=0;
int j=1;
cout<< “Enter an Integer value”;
cin>>n;
while (i<n){
    x++;
     i++;
 }
while (j<n)
 {
    j++;
 }
}
```

Time Units to Compute

---

1 for the first assignment statement: x=0;

1 for the second assignment statement: i=0;

1 for the third assignment statement: j=1;

1 for the output statement.

1 for the input statement.

In the first while loop:

n+1 tests

n loops of 2 units for the two increment (addition) operations

In the second while loop:

n tests

n-1 increments

T (n)= *1+1+1+1+1+n+1+2n+n+n-1 = 5n+5 = O(n)*

4. int sum (int n)

```
{
 int partial_sum = 0;
 for (int i = 1; i <= n; i++)
        partial_sum = partial_sum +(i * i * i);
 return partial_sum;
}
```

Time Units to Compute

---

1 for the assignment.

1 assignment, *n+1* tests, and *n* increments.

n loops of 4 units for an assignment, an addition, and two multiplications.

1 for the return statement.

T (n)= *1+(1+n+1+n)+4n+1 = 6n+4 = O(n)*

1.2.4 . Formal Approach to Analysis

In the above examples we have seen that analysis is a bit complex. However, it can be simplified by using some formal approach in which case we can ignore initializations, loop control, and book keeping.

for Loops: Formally

- In general, a for loop translates to a summation. The index and bounds of the summation are the same as the index and bounds of the for loop.

```
for (int i = 1; i <= N; i++) {
    sum = sum+i;
}
```

$$\sum_{i=1}^{N} 1 = N$$

- Suppose we count the number of additions that are done. There is 1 addition per iteration of the loop, hence *N* additions in total.

Nested Loops: Formally

- Nested for loops translate into multiple summations, one for each for loop.

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
        sum = sum+i+j;
    }
}
```

$$\sum_{i=1}^{N} \sum_{j=1}^{M} 2 = \sum_{i=1}^{N} 2M = 2MN$$

- Again, count the number of additions. The outer summation is for the outer for loop.

Consecutive Statements: Formally

- Add the running times of the separate blocks of your code

```
for (int i = 1; i <= N; i++) {
    sum = sum+i;
}
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        sum = sum+i+j;
    }
}
```

$$\left[\sum_{i=1}^{N} 1\right] + \left[\sum_{i=1}^{N}\sum_{j=1}^{N} 2\right] = N + 2N^2$$

Conditionals: Formally

- If (test) s1 else s2: Compute the maximum of the running time for s1 and s2.

```
if (test == 1) {
    for (int i = 1; i <= N; i++) {
        sum = sum+i;
}}
else for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            sum = sum+i+j;
}}
```

$$\max\left(\sum_{i=1}^{N} 1, \sum_{i=1}^{N}\sum_{j=1}^{N} 2\right) = \max\left(N, 2N^2\right) = 2N^2$$

1.3 Measures of Times

In order to determine the running time of an algorithm it is possible to define three functions $T_{best}(n)$, $T_{avg}(n)$ and $T_{worst}(n)$ as the best, the average and the worst case running time of the algorithm respectively.

Average Case (T_{avg}): The amount of time the algorithm takes on an "average" set of inputs.
Worst Case (T_{worst}): The amount of time the algorithm takes on the worst possible set of inputs.
Best Case (T_{best}): The amount of time the algorithm takes on the smallest possible set of inputs.

We are interested in the worst-case time, since it provides a bound for all input – this is called the "Big-Oh" estimate.

Asymptotic Notations

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

1.3.1. The Big-Oh Notation

Big-Oh notation is a way of comparing algorithms and is used for computing the complexity of algorithms; i.e., the amount of time that it takes for computer program to run . It's only concerned with what happens for very a large value of n. Therefore only the largest term in the expression (function) is needed. For example, if the number of operations in an algorithm is $n^2 – n$, n is insignificant compared to n^2 for large values of n. Hence the n term is ignored. Of course, for small values of n, it may be important. However, Big-Oh is mainly concerned with large values of n.

Formal Definition: f (n)= O (g (n)) if there exist c, k ∈ $\mathscr{R}^+$ such that for all n≥ k, f (n) ≤ c.g (n).

Examples: The following points are facts that you can use for Big-Oh problems:

- 1<=n for all n>=1
- n<=n^2 for all n>=1
- 2^n <=n! for all n>=4
- $\log_2$n<=n for all n>=2
- n<=n$\log_2$n for all n>=2

1. f(n)=10n+5 and g(n)=n. Show that f(n) is O(g(n)).

   To show that f(n) is O(g(n)) we must show that constants c and k such that

   f(n) <=c.g(n) for all n>=k

   Or 10n+5<=c.n for all n>=k

   Try c=15. Then we need to show that 10n+5<=15n

   Solving for n we get: 5<5n or 1<=n.

   So f(n) =10n+5 <=15.g(n) for all n>=1.

   (c=15,k=1).

2. f(n) = $3n^2$ +4n+1. Show that f(n)=O(n^2).

   4n <=$4n^2$ for all n>=1 and 1<=n^2 for all n>=1

$3n^2 + 4n + 1 <= 3n^2 + 4n^2 + n^2$ for all $n >= 1$

$<= 8n^2$ for all $n >= 1$

So we have shown that $f(n) <= 8n^2$ for all $n >= 1$

Therefore, f (n) is $O(n^2)$ $(c=8, k=1)$

Remark:

Demonstrating that a function f(n) is big-O of a function g(n) requires that we find specific constants c and k for which the inequality holds (and show that the inequality does in fact hold).

Big-O expresses an *upper bound* on the growth rate of a function, for sufficiently large values of n. An *upper bound* is the best algorithmic solution that has been found for a problem.

In simple words, f (n) =O(g(n)) means that the growth rate of f(n) is less than or equal to g(n).

Big-O Theorems

For all the following theorems, assume that f(n) is a function of n and that k is an arbitrary constant.

Theorem 1: k is O(1)
Theorem 2: A polynomial is O(the term containing the highest power of n).

Polynomial's growth rate is determined by the leading term

- If *f(n)* is a polynomial of degree *d*, then *f(n)* is $O(n^d)$

In general, f(n) is big-O of the dominant term of f(n).

Theorem 3: k*f(n) is O(f(n))

Constant factors may be ignored

E.g. $f(n) = 7n^4 + 3n^2 + 5n + 1000$ is $O(n^4)$

Theorem 4(Transitivity): If f(n) is O(g(n))and g(n) is O(h(n)), then f(n) is O(h(n)).

Theorem 5: For any base b, $\log_b(n)$ is O(logn).

All logarithms grow at the same rate

$\log_b n$ is $O(\log_d n)$ $\forall b, d > 1$

1.3.2 Omega **Notation**

Just as O-notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.
Formal Definition: A function f(n) is Ω(g (n)) if there exist constants c and k ∈ ℛ+ such that f(n) >=c. g(n) for all n>=k.

f(n)= *Ω(g (n))* means that f(n) is greater than or equal to some constant multiple of g(n) for all values of n greater than or equal to some k.

Example*:* If $f(n) = n^{2,}$ then f(n)= *Ω(n)*

In simple terms, f(n)= *Ω(g (n))* means that the growth rate of f(n) is greater that or equal to g(n).

1.3.3. Theta Notation

A function f (n) belongs to the set of Θ (g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1.g(n) and c2.g(n), for sufficiently large values of n.

Formal Definition: A function f (n) is Θ (g(n)) if it is both *O(g(n))* and *Ω (g(n)).* In other words, there exist constants c1, c2, and k >0 such that c1.g (n)<=f(n)<=c2. g(n) for all n >= k

If f(n)= Θ (g(n)), then g(n) is an asymptotically tight bound for f(n).

In simple terms, f(n)= Θ (g(n)) means that f(n) and g(n) have the same rate of growth.

Example: 3n +2 =O(n) as 3n<=3n +2<=4n for all n>=2,

All these are technically correct, but the last expression is the best and tight one. Since $2n^2$ and n^2 have the same growth rate, it can be written as f(n)= $\Theta(n^2)$.

Relational Properties of the Asymptotic Notations

Transitivity

- if f(n)=Θ(g(n)) and g(n)= Θ(h(n)) then f(n)=Θ(h(n)),
- if f(n)=O(g(n)) and g(n)= O(h(n)) then f(n)=O(h(n)),
- if f(n)=Ω(g(n)) and g(n)= Ω(h(n)) then f(n)=Ω (h(n)),

Symmetry

- $f(n)=\Theta(g(n))$ if and only if $g(n)=\Theta(f(n))$.

Transpose symmetry

- $f(n)=O(g(n))$ if and only if $g(n)=\Omega(g(n)$,

Reflexivity

- $f(n)=\Theta(f(n))$,
- $f(n)=O(f(n))$,
- $f(n)=\Omega(f(n))$.

CHAPTER TWO

Simple Sorting and Searching Algorithms

2.1. Sorting Algorithms

Sorting is one of the most important operations performed by computers. Sorting is a process of reordering a list of items in either increasing or decreasing order.

The following are simple sorting algorithms used to sort small-sized lists:

1. Insertion Sort
2. Selection Sort
3. Bubble Sort

1. Insertion Sort

The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use

an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

It's the most instinctive type of sorting algorithm. The approach is the same approach that you use for sorting a set of cards in your hand. While playing cards, you pick up a card, start at the beginning of your hand and find the place to insert the new card, insert it and move all the others up one place.

Insertion sort is a faster and more improved sorting algorithm than selection sort. In selection sort the algorithm iterates through all of the data through every pass whether it is already sorted or not. However, insertion sort works differently, instead of iterating through all of the data after every pass the algorithm only traverses the data it needs to until the segment that is being sorted is sorted. Again there are two loops that are required by insertion sort and therefore two main variables, which in this case are named 'i' and 'j'. Variables 'i' and 'j' begin on the same index after every pass of the first loop, the second loop only executes if variable 'j' is greater then index 0 AND arr[j] < arr[j - 1]. In other words, if 'j' hasn't reached the end of the data AND the value of the index where 'j' is at is smaller than the value of the index to the left of 'j', finally 'j' is decremented. As long as these two conditions are met in the second loop it will keep executing, this is what sets insertion sort apart from selection sort. Only the data that needs to be sorted is sorted. As always a visual representation helps.

Basic Idea:

Find the location for an element and move all others up, and insert the element.

The process involved in insertion sort is as follows:

1. The left most value can be said to be sorted relative to itself. Thus, we don’t need to do anything.
2. Check to see if the second value is smaller than the first one. If it is, swap these two values. The first two values are now relatively sorted.

3. Next, we need to insert the third value in to the relatively sorted portion so that after insertion, the portion will still be relatively sorted.
4. Remove the third value first. Slide the second value to make room for insertion. Insert the value in the appropriate position.
5. Now the first three are relatively sorted.
6. Do the same for the remaining items in the list.

Implementation

```
void insertion_sort(int list[]){

int temp;

for(int i=1;i<n;i++){

        temp=list[i];

      for(int j=i; j>0 && temp<list[j-1];j--)

          { // work backwards through the array finding where temp should go

            list[j]=list[j-1];

            list[j-1]=temp;

          }//end of inner loop

        }//end of outer loop

}//end of insertion_sort
```

Analysis of insertion sort

How many comparisons?

$1+2+3+\ldots+(n-1) = O(n^2)$

How many swaps?

$1+2+3+\ldots+(n-1) = O(n^2)$

How much space?

In-place algorithm

2. Selection Sort

Selection sort is one of the basic algorithms for sorting data, its simplicity proves useful for sorting small amounts of data. Selection sort works by first starting at the beginning array (index 0) and traverses the entire array comparing each value with the current index, if it is smaller than the current index than that index is saved. Once the loop has traversed all the data and if a smaller value than the current index was found a swap is made between the current index in the index where the smaller value was found. The current index is then incremented, now to index 1, the algorithm repeats. Of course, a visual representation is usually more useful.

Basic Idea:

- Loop through the array from i=0 to n-1.
- Select the smallest element in the array from i to n
- Swap this value with value at position i.

Implementation:

```
void selection_sort(int list[])

{

int i,j, smallest;

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

    smallest=i;

  for(j=i+1;j<n;j++){
```

```
      if(list[j]<list[smallest])
           smallest=j;
        }//end of inner loop
          temp=list[smallest];
          list[smallest]=list[i];
          list[i]=temp;
        } //end of outer loop
}//end of selection_sort
```

Analysis

How many comparisons?

$(n-1)+(n-2)+\ldots+1= O(n^2)$

How many swaps?

$n=O(n)$

How much space?

In-place algorithm

Time Complexity of selection sort

The time complexity of selection sort is $O(n^2)$, for best, average, and worst case scenarios. Because of this selection sort is a very inefficient sorting algorithm for large amounts of data, it's sometimes preferred for very small amounts of data such as the example above. The complexity is $O(n^2)$ for all cases because of the way selection sort is designed to traverse the data. The outer loops first iteration has n comparisons (where n is the number of elements in the data) the second

iteration would have n-1 comparisons followed by n-2, n-3, n-4...thus resulting in $O(n^2)$ time complexity.

3. Bubble Sort

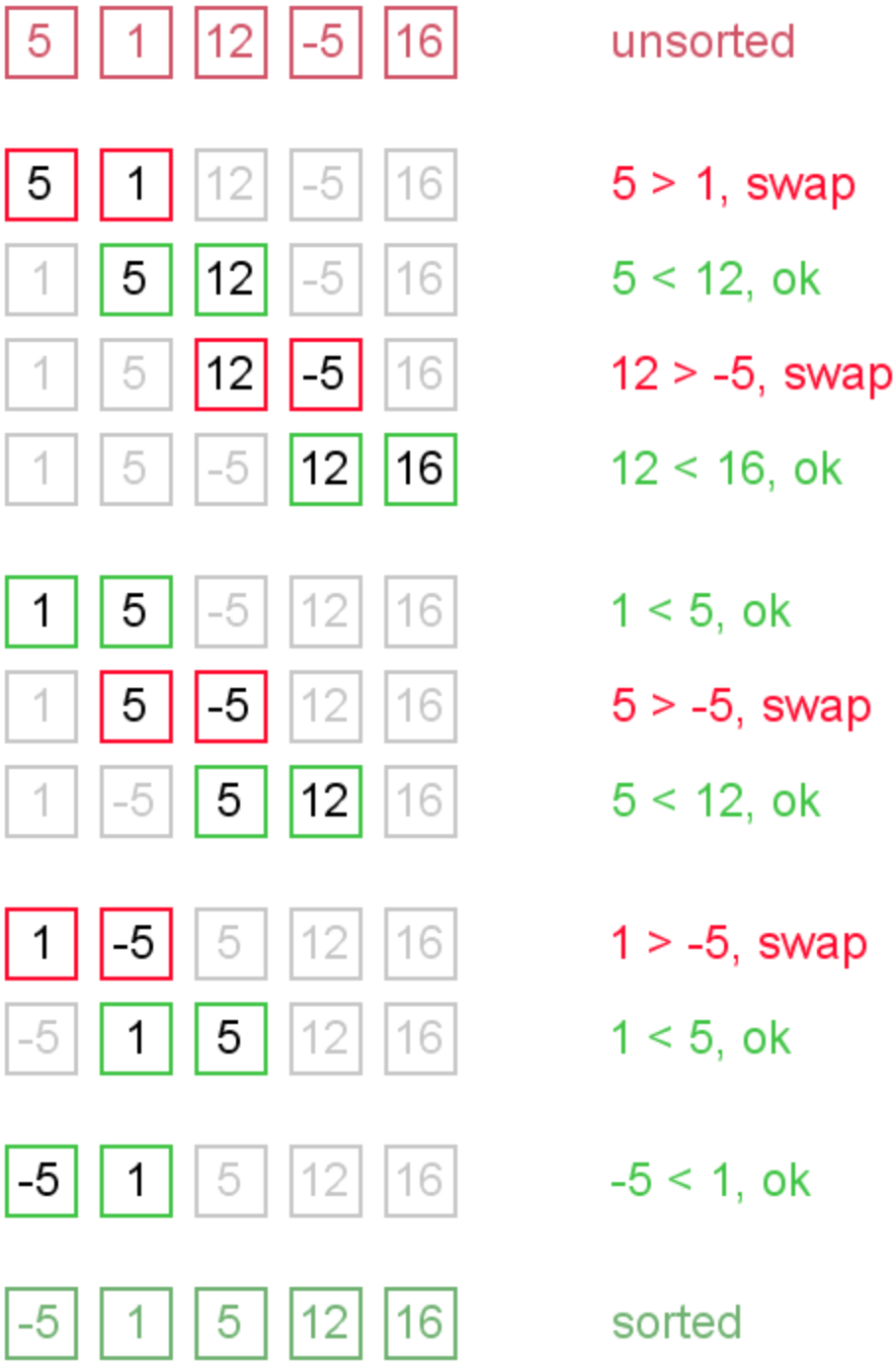
Bubble sort is the simplest algorithm to implement and the slowest algorithm on very large inputs. Bubble sort is a simple and well-known sorting algorithm. It is used in practice once in a blue moon and its main application is to make an introduction to the sorting algorithms. Bubble sort belongs to $O(n^2)$ sorting algorithms, which makes it quite inefficient for sorting large data volumes. Bubble sort is **stable** and **adaptive**.

Algorithm

1. Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.
2. If at least one swap has been done, repeat step 1.

You can imagine that on every step big bubbles float to the surface and stay there. At the step, when no bubble moves, sorting stops. Let us see an example of sorting an array to make the idea of bubble sort clearer.

Example. Sort {5, 1, 12, -5, 16} using bubble sort.



Implementation:

void bubble_sort(list[])

```
{
 int i,j,temp;
 for(i=0;i<n; i++){
     for(j=n-1;j>i; j--){
          if(list[j]<list[j-1]){
                    temp=list[j];
                    list[j]=list[j-1];
                    list[j-1]=temp;
                    }//swap adjacent elements
               }//end of inner loop
          }//end of outer loop
}//end of bubble_sort
```

Analysis of Bubble Sort

How many comparisons?

$(n-1)+(n-2)+\ldots+1= O(n^2)$

How many swaps?

$(n-1)+(n-2)+\ldots+1= O(n^2)$

Space?

In-place algorithm.

Complexity analysis of bubble sort

Average and worst case complexity of bubble sort is $O(n^2)$. Also, it makes $O(n^2)$ swaps in the worst case. Bubble sort is adaptive. It means that for almost sorted array it gives O(n) estimation. Avoid implementations, which don't check if the array is already sorted on every step (any swaps made). This check is necessary, in order to preserve adaptive property.

General Comments

Each of these algorithms requires *n*-1 passes: each pass places one item in its correct place. The i^{th} pass makes either *i* or *n* - *i* comparisons and moves. So:

$$\begin{aligned} T(n) &= 1 + 2 + 3 + \ldots + (n-1) \\ &= \sum_{i=1}^{n-1} i \\ &= \tfrac{n}{2}(n-1) \end{aligned}$$

or $O(n^2)$. Thus these algorithms are only suitable for small problems where their simple code makes them faster than the more complex code of the O(*n* log*n*) algorithm. As a rule of thumb, expect to find an O(*n* log*n*) algorithm faster for *n*>10 - *but the exact value depends very much on individual machines!*.

Empirically it's known that Insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort. In short, there really isn't any reason to use the selection sort - use the insertion sort instead.

If you really want to use the selection sort for some reason, try to avoid sorting lists of more than a 1000 items with it or repetitively sorting lists of more than a couple hundred items.

Searching

Searching is a process of looking for a specific element in a list of items or determining that the item is not in the list.

Searching is the algorithmic process of finding a particular item in a collection of items. A search typically answers either True or False as to whether the item is present. On occasion it may be modified to return where the item is found. For our purposes here, we will simply concern ourselves with the question of membership.

There are two simple searching algorithms:

1. Sequential Search, and
2. Binary Search

1. Linear Search (Sequential Search)

Pseudocode

Loop through the array starting at the first element until the value of target matches one of the array elements.

If a match is not found, return –1.

Time is proportional to the size of input (*n)* and we call this time complexity *O(n)*.

Example Implementation:

```
int Linear_Search(int list[], int key)

{

int index=0;
```

```
int found=0;
do{
if(key==list[index])
    found=1;
else
   index++;
}while(found==0&&index<n);
if(found==0)
    index=-1;
return index;
}
```

2. Binary Search

This searching algorithms works only on an ordered list.

The basic idea is:

- Locate midpoint of array to search
- Determine if target is in lower half or upper half of an array.
    - If in lower half, make this half the array to search
    - If in the upper half, make this half the array to search
- Loop back to step 1 until the size of the array to search is one, and this element does not match, in which case return –1.

The computational time for this algorithm is proportional to $\log_2 n$. Therefore the time complexity is O*(*log *n)*

Example Implementation:

```
int Binary_Search(int list[],int k)
{
int left=0;
int right=n-1;
int found=0;
do{
mid=(left+right)/2;
if(key==list[mid])
    found=1;
else{
     if(key<list[mid])
          right=mid-1;
     else
          left=mid+1;
      }
}while(found==0&&left<right);
if(found==0)
  index=-1;
else
  index=mid;
```

```
return index;

}
```

Chapter 3: Linked Lists (4hr)

3.1. Review on Pointer and Dynamic Memory allocation

3.2. Singly Linked List and Its Implementation

3.3. Doubly Linked List and Its Implementation

3.4. Circular Linked Lists and Its Implementation

Chapter Three

Linked lists and Structures

3.1. Structures

Structures are aggregate data types built using elements of primitive data types.

Structures are defined using the struct keyword:

```
E.g.  struct Time{
          int hour;
          int minute;
          int second;
      };
```

The struct keyword creates a new user defined data type that is used to declare variables of an aggregate data type.

Structure variables are declared like variables of other types.

Syntax: struct <structure tag> <variable name>;

```
E.g.  struct Time timeObject,
          struct Time *timeptr;
```

3.1.1. Accessing Members of Structure Variables

The Dot operator (.): to access data members of structure variables.

The Arrow operator (->): to access data members of pointer variables pointing to the structure.

E.g. Print member hour of timeObject and timeptr.

```
cout<< timeObject.hour; or

cout<<timeptr->hour;
```

TIP: timeptr->hour is the same as (*timeptr).hour.

The parentheses is required since (*) has lower precedence than (.).

3.1.2. Self-Referential Structures

Structures can hold pointers to instances of themselves.

```
struct list{
        char name[10];
        int count;
        struct list *next;
};
```

However, structures cannot contain instances of themselves.

3.2. Singly Linked Lists

Linked lists are the most basic self-referential structures. Linked lists allow you to have a chain of structs with related data.

Array vs. Linked lists

Arrays are *simple* and *fast but we m*ust specify their size at construction time. This has its own drawbacks. If you construct an array with space for *n*, tomorrow you may need *n*+1.Here comes a need for a more flexible system.

Advantages of Linked Lists

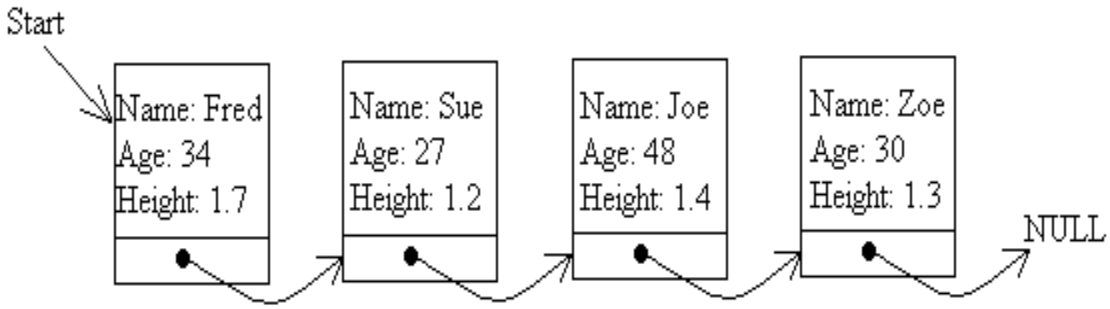
Flexible space use by dynamically allocating space for each element as needed. This implies that one need not know the size of the list in advance. Memory is efficiently utilized.

A linked list is made up of a chain of nodes. Each node contains:

- the data item, and
- a pointer to the next node

3.2.1. Creating Linked Lists in C++

A linked list is a data structure that is built from structures and pointers. It forms a chain of "nodes" with pointers representing the links of the chain and holding the entire thing together. A linked list can be represented by a diagram like this one:



This linked list has four nodes in it, each with a link to the next node in the series. The last node has a link to the special value NULL, which any pointer (whatever its type) can point to, to show that it is the last link in the chain. There is also another special pointer, called Start (also called head), which points to the first link in the chain so that we can keep track of it.

3.2.2. Defining the data structure for a linked list

The key part of a linked list is a structure, which holds the data for each node (the name, address, age or whatever for the items in the list), and, most importantly, a pointer to the next node. Here we have given the structure of a typical node:

```
struct node
 {  char name[20];    // Name of up to 20 letters
    int age
    float height;     // In metres
    node *nxt;// Pointer to next node
 };
struct node *start_ptr = NULL;
```

The important part of the structure is the line before the closing curly brackets. This gives a pointer to the next node in the list. This is the only case in C++ where you are allowed to refer to a data type (in this case **node**) before you have even finished defining it!

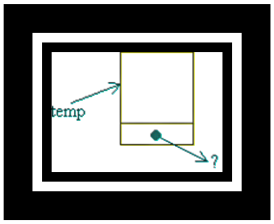
We have also declared a pointer called **start_ptr** that will permanently point to the start of the list. To start with, there are no nodes in the list, which is why **start_ptr** is set to NULL.

3.2.3. Adding a node to the list

The first problem that we face is how to add a node to the list. For simplicity's sake, we will assume that it has to be added to the end of the list, although it could be added anywhere in the list (a problem we will deal with later on).

Firstly, we declare the space for a pointer item and assign a temporary pointer to it. This is done using the **new** statement as follows:

temp = new node;



We can refer to the new node as ***temp**, i.e. **"the node that temp points to"**. When the fields of this structure are referred to, brackets can be put round the ***temp** part, as otherwise the compiler will think we are trying to refer to the fields of the pointer. Alternatively, we can use the arrow pointer notation.

That's what we shall do here.

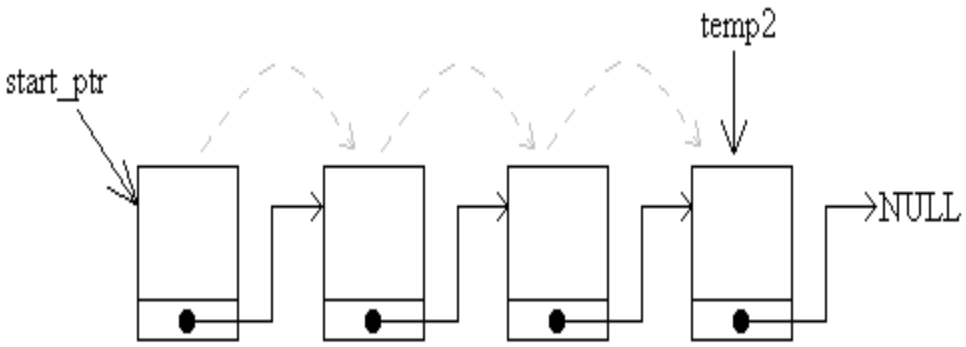
Having declared the node, we ask the user to fill in the details of the person, i.e. the name, age, address or whatever:

```
cout << "Please enter the name of the person: ";
cin >> temp->name;
cout << "Please enter the age of the person : ";
cin >> temp->age;
cout << "Please enter the height of the person : ";
cin >> temp->height;
temp->nxt = NULL;
```

The last line sets the pointer from this node to the next to NULL, indicating that this node, when it is inserted in the list, will be the last node. Having set up the information, we have to decide what to do with the pointers. Of course, if the list is empty to start with, there's no problem - just set the Start pointer to point to this node (i.e. set it to the same value as temp):

```
if (start_ptr == NULL)
   start_ptr = temp;
```

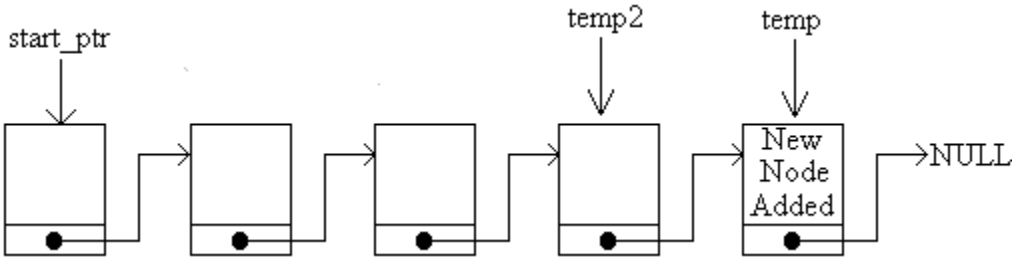
It is harder if there are already nodes in the list. In this case, the secret is to declare a second pointer, **temp2**, to step through the list until it finds the last node.




```
temp2 = start_ptr;
  // We know this is not NULL - list not empty!
while (temp2->nxt != NULL)
 {  temp2 = temp2->nxt;   // Move to next link in chain
 }
```

The loop will terminate when **temp2** points to the last node in the chain, and it knows when this happened because the **nxt** pointer in that node will point to NULL. When it has found it, it sets the pointer from that last node to point to the node we have just declared:

```
temp2->nxt = temp;
```



The link **temp2->nxt** in this diagram is the link joining the last two nodes. The full code for adding a node at the end of the list is shown below, in its own little function:

```
void add_node_at_end ()
 {  node *temp, *temp2;   // Temporary pointers

    // Reserve space for new node and fill it with data
    temp = new node;
```

```
    cout << "Please enter the name of the person: ";
    cin >> temp->name;
    cout << "Please enter the age of the person : ";
    cin >> temp->age;
    cout << "Please enter the height of the person : ";
    cin >> temp->height;
    temp->nxt = NULL;

    // Set up link to this node
    if (start_ptr == NULL)
       start_ptr = temp;
    else
     { temp2 = start_ptr;
       // We know this is not NULL - list not empty!
       while (temp2->nxt != NULL)
         {  temp2 = temp2->nxt;
            // Move to next link in chain
         }
       temp2->nxt = temp;
     }
}
```

3.2.4. Displaying the list of nodes

Having added one or more nodes, we need to display the list of nodes on the screen. This is comparatively easy to do. Here is the method:

1. Set a temporary pointer to point to the same thing as the start pointer.
2. If the pointer points to NULL, display the message "End of list" and stop.
3. Otherwise, display the details of the node pointed to by the start pointer.
4. Make the temporary pointer point to the same thing as the **nxt** pointer of the node it is currently indicating.
5. Jump back to step 2.

The temporary pointer moves along the list, displaying the details of the nodes it comes across. At each stage, it can get hold of the next node in the list by using the **nxt** pointer of the node it is currently pointing to. Here is the C++ code that does the job:

```
//displaying the list of nodes
temp = start_ptr;
do
 {  if (temp == NULL)
     cout << "End of list" << endl;
   else
    {  // Display details for what temp points to
      cout << "Name : " << temp->name << endl;
      cout << "Age : " << temp->age << endl;
      cout << "Height : " << temp->height << endl;

      // Move to next node (if present)
      temp = temp->nxt;
    }
 } while (temp != NULL);
```

Check through this code, matching it to the method listed above. It helps if you draw a diagram on paper of a linked list and work through the code using the diagram.

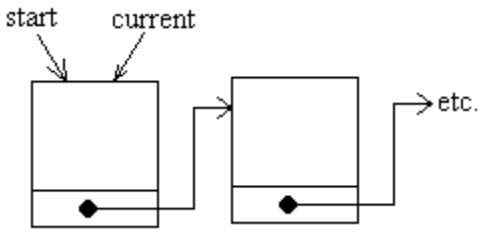
3.2.5. Navigating through the list

One thing you may need to do is to navigate through the list, with a pointer that moves backwards and forwards through the list, like an index pointer in an array. This is certainly necessary when you want to insert or delete a node from somewhere inside the list, as you will need to specify the position.

We will call the mobile pointer **current**. First of all, it is declared, and set to the same value as the **start_ptr** pointer:

```
node *current;
current = start_ptr;
```

Notice that you don't need to set current equal to the *address* of the start pointer, as they are both pointers. The statement above makes them both point to the same thing:



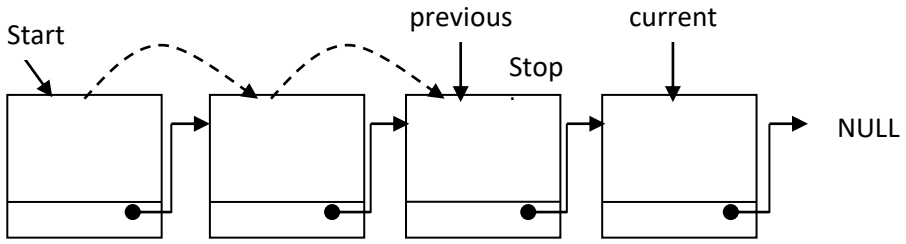
It's easy to get the current pointer to point to the next node in the list (i.e. move from left to right along the list). If you want to move current along one node, use the nxt field of the node that it is pointing to at the moment:

```
current = current->nxt;
```

In fact, we had better check that it isn't pointing to the last item in the list. If it is, then there is no next node to move to:

```
if (current->nxt == NULL)
    cout << "You are at the end of the list." << endl;
else
    current = current->nxt;
```

Moving the current pointer back one step is a little harder. This is because we have no way of moving back a step automatically from the current node. The only way to find the node before the current one is to start at the beginning, work our way through and stop when we find the node before the one we are considering at the moment. We can tell when this happens, as the **nxt** pointer from that node will point to exactly the same place in memory as the current pointer (i.e. the current node).



First of all, we had better check to see if the current node is also first the one. If it is, then there is no "previous" node to point to. If not, check through all the nodes in turn until we detect that we are just behind the current one (Like a pantomime - "behind you!")

```
if (current == start_ptr)
    cout << "You are at the start of the list" << endl;
else
  { node *previous;     // Declare the pointer
    previous = start_ptr;

    while (previous->nxt != current)
      {    previous = previous->nxt;
      }
    current = previous;
  }
```

The else clause translates as follows: Declare a temporary pointer (for use in this else clause only). Set it equal to the start pointer. All the time that it is not pointing to the node before the current node, move it along the line. Once the previous node has been found, the current pointer is set to that node - i.e. it moves back along the list.

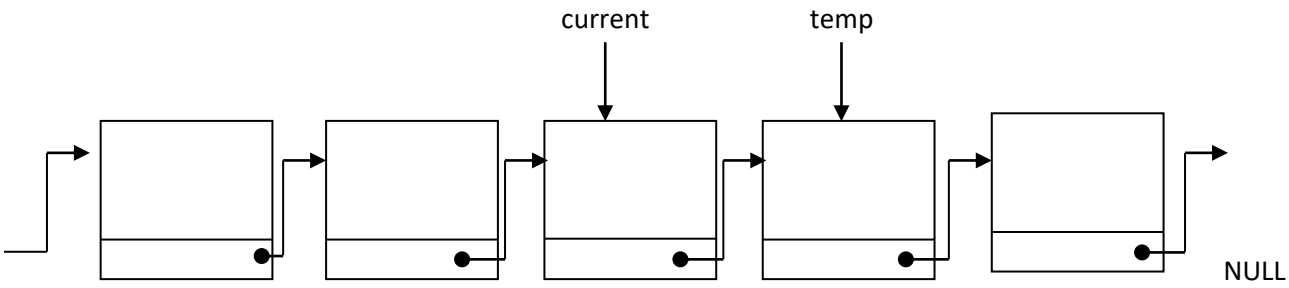
Now that you have the facility to move back and forth, you need to do something with it. Firstly, let's see if we can alter the details for that particular node in the list:

```
cout << "Please enter the new name of the person: ";
cin >> current->name;
cout << "Please enter the new age of the person : ";
cin >> current->age;
cout << "Please enter the new height of the person : ";
cin >> current->height;
```

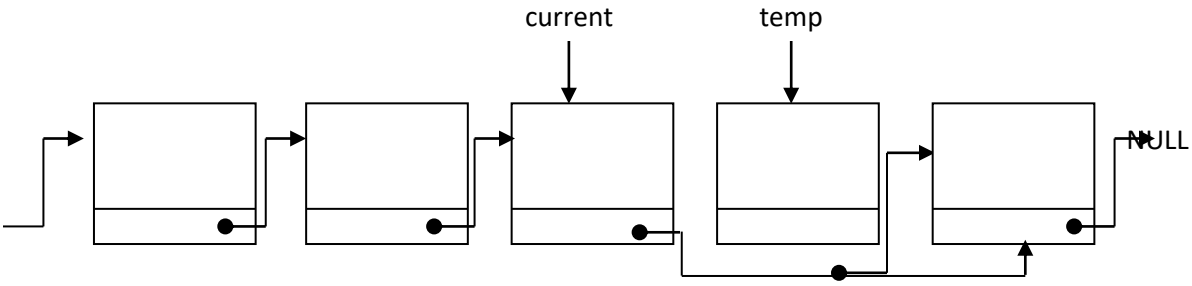
The next easiest thing to do is to delete a node from the list directly after the current position. We have to use a temporary pointer to point to the node to be deleted. Once this node has been

"anchored", the pointers to the remaining nodes can be readjusted before the node on death row is deleted. Here is the sequence of actions:

1. Firstly, the temporary pointer is assigned to the node after the current one. This is the node to be deleted:



2. Now the pointer from the current node is made to leap-frog the next node and point to the one after that:



3. The last step is to delete the node pointed to by **temp**.

Here is the code for deleting the node. It includes a test at the start to test whether the current node is the last one in the list:

```
if (current->nxt == NULL)
    cout << "There is no node after current" << endl;
else
  { node *temp;
    temp = current->nxt;
    current->nxt = temp->nxt;     // Could be NULL
    delete temp;
```

```
}
```

Here is the code to ***add*** a node after the current one. This is done similarly, but we haven't illustrated it with diagrams:

```
if (current->nxt == NULL)
    add_node_at_end();
else
  { node *temp;
    new temp;
    get_details(temp);
    // Make the new node point to the same thing as
    // the current node
    temp->nxt = current->nxt;
    // Make the current node point to the new link
    // in the chain
    current->nxt = temp;
  }
```

We have assumed that the function **add_node_at_end()** is the routine for adding the node to the end of the list that we created near the top of this section. This routine is called if the current pointer is the last one in the list so the new one would be added on to the end.

Similarly, the routine **get_temp(temp)** is a routine that reads in the details for the new node similar to the one defined just above.

... and so ...

3.2.6. Deleting a node from the list

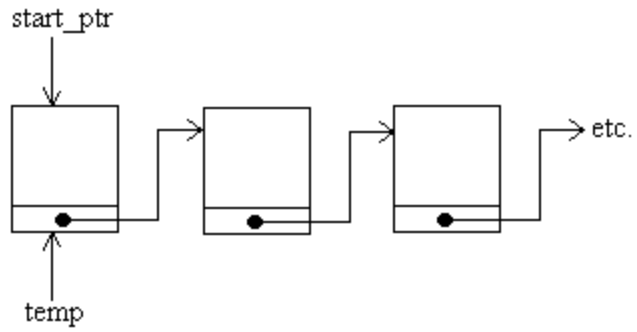
When it comes to deleting nodes, we have three choices: Delete a node from the start of the list, delete one from the end of the list, or delete one from somewhere in the middle. For simplicity, we shall just deal with deleting one from the start or from the end.

When a node is deleted, the space that it took up should be reclaimed. Otherwise the computer will eventually run out of memory space. This is done with the **delete** instruction:

delete temp; // Release the memory pointed to by temp

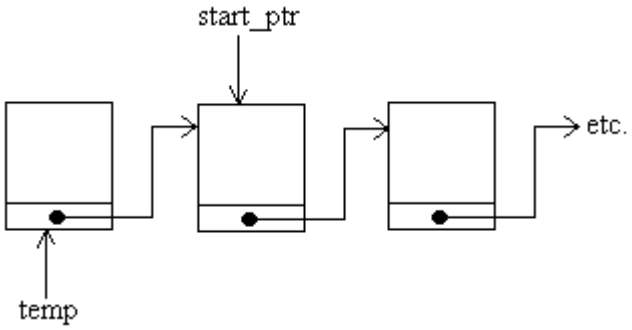
However, we can't just delete the nodes willy-nilly as it would break the chain. We need to reassign the pointers and then delete the node at the last moment. Here is how we go about deleting the first node in the linked list:

temp = start_ptr; // Make the temporary pointer
// identical to the start pointer

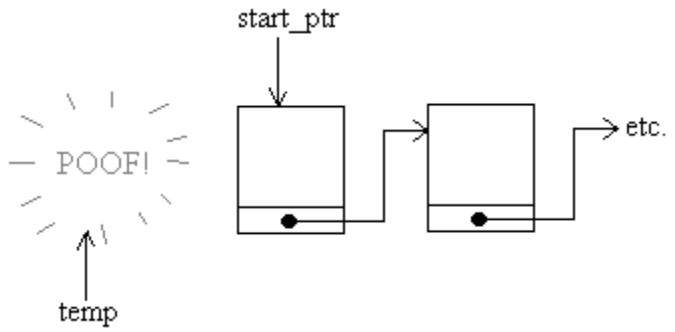


Now that the first node has been safely tagged (so that we can refer to it even when the start pointer has been reassigned), we can move the start pointer to the next node in the chain:

start_ptr = start_ptr->nxt; // Second node in chain.



delete temp; // Wipe out original start node



Here is the function that deletes a node from the start:

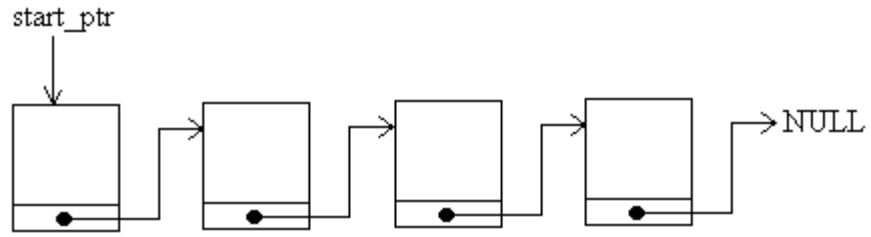
```
void delete_start_node()
  { node *temp;
    temp = start_ptr;
    start_ptr = start_ptr->nxt;
    delete temp;
  }
```

Deleting a node from the end of the list is harder, as the temporary pointer must find where the end of the list is by hopping along from the start. This is done using code that is almost identical to that used to insert a node at the end of the list. It is necessary to maintain two temporary pointers, **temp1** and **temp2**. The pointer **temp1** will point to the last node in the list and **temp2** will point to the previous node. We have to keep track of both as it is necessary to delete the last node and immediately afterwards, to set the **nxt** pointer of the previous node to NULL (it is now the new last node).

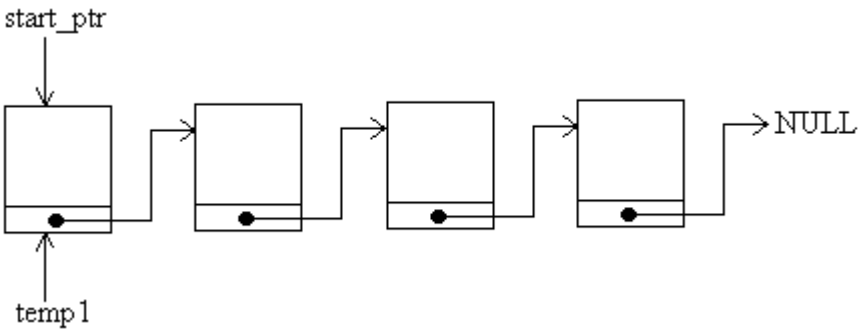
1. Look at the start pointer. If it is NULL, then the list is empty, so print out a "No nodes to delete" message.
2. Make **temp1** point to whatever the start pointer is pointing to.
3. If the **nxt** pointer of what temp1 indicates is NULL, then we've found the last node of the list, so jump to step 7.
4. Make another pointer, **temp2**, point to the current node in the list.
5. Make **temp1** point to the next item in the list.
6. Go to step 3.

7. If you get this far, then the temporary pointer, **temp1**, should point to the last item in the list and the other temporary pointer, **temp2**, should point to the last-but-one item.
8. Delete the node pointed to by **temp1**.
9. Mark the **nxt** pointer of the node pointed to by **temp2** as NULL - it is the new last node.

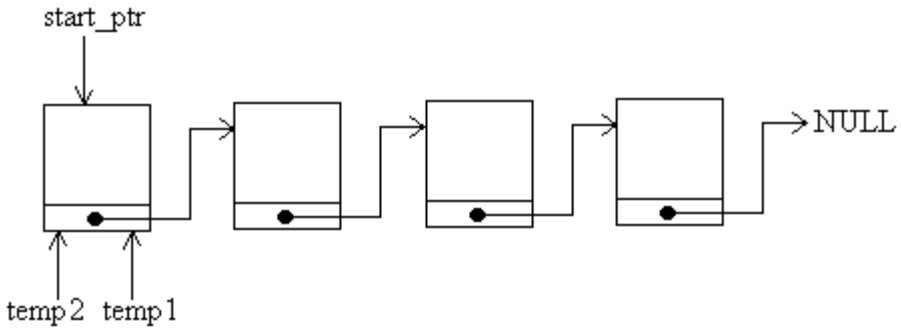
Let's try it with a rough drawing. This is always a good idea when you are trying to understand an abstract data type. Suppose we want to delete the last node from this list:



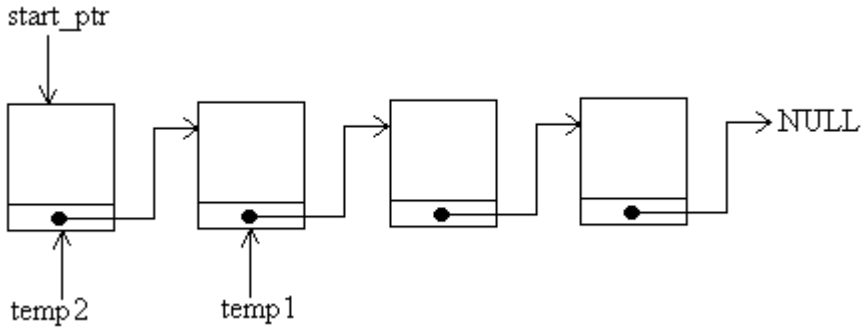
Firstly, the start pointer doesn't point to NULL, so we don't have to display a "Empty list, wise guy!" message. Let's get straight on with step2 - set the pointer **temp1** to the same as the start pointer:



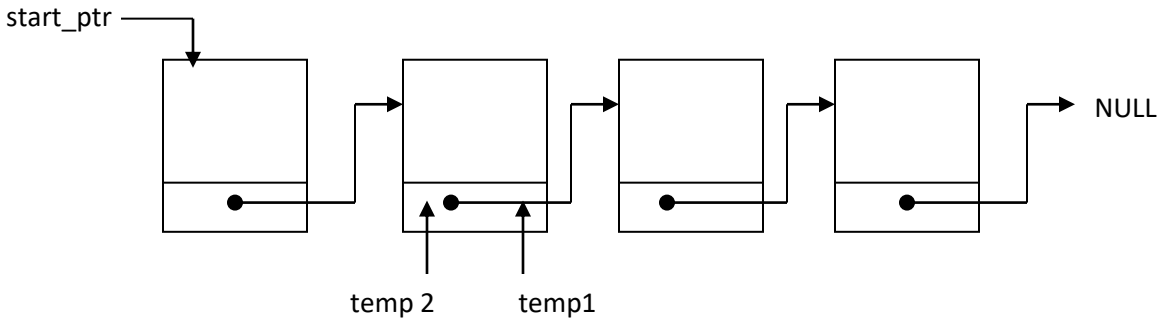
The **nxt** pointer from this node isn't NULL, so we haven't found the end node. Instead, we set the pointer **temp2** to the same node as **temp1**



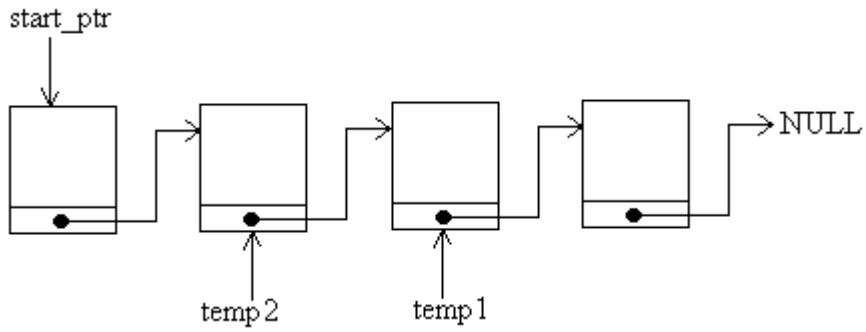
and then move temp1 to the next node in the list:



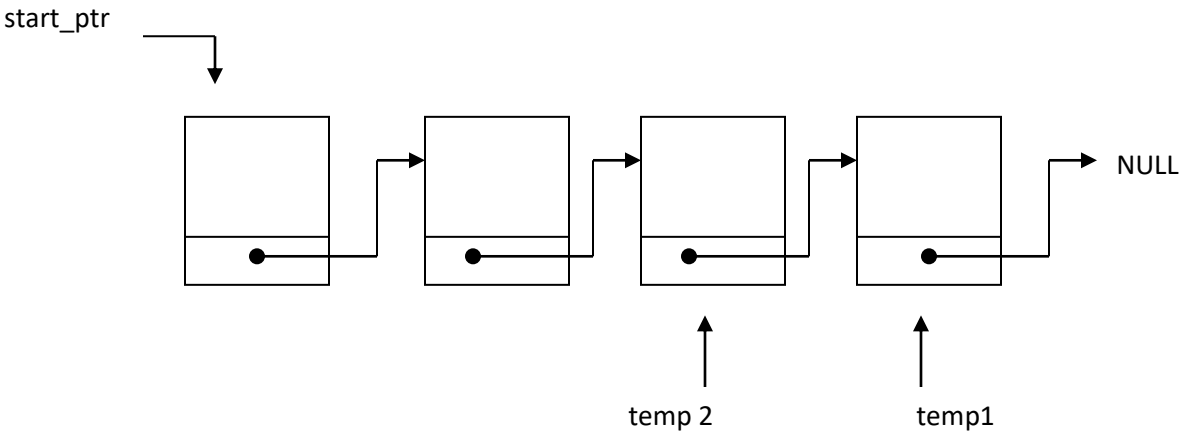
Going back to step 3, we see that temp1 still doesn't point to the last node in the list, so we make temp2 point to what temp1 points to



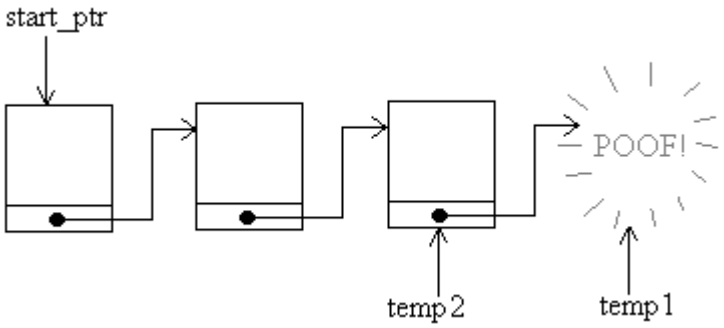
and **temp1** is made to point to the next node along:



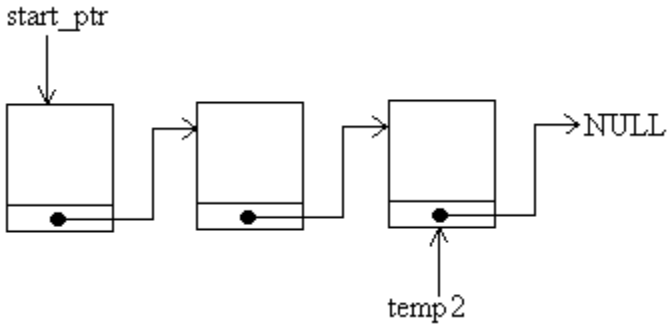
Eventually, this goes on until **temp1** really is pointing to the last node in the list, with **temp2** pointing to the penultimate node:



Now we have reached step 8. The next thing to do is to delete the node pointed to by **temp1**



and set the **nxt** pointer of what **temp2** indicates to NULL:



We suppose you want some code for all that! All right then ....

```
void delete_end_node()
  { node *temp1, *temp2;
    if (start_ptr == NULL)
        cout << "The list is empty!" << endl;
    else
      { temp1 = start_ptr;
        while (temp1->nxt != NULL)
          { temp2 = temp1;
            temp1 = temp1->nxt;
          }
        delete temp1;
        temp2->nxt = NULL;
      }
  }
```

The code seems a lot shorter than the explanation!

Now, the sharp-witted amongst you will have spotted a problem. If the list only contains one node, the code above will malfunction. This is because the function goes as far as the **temp1** = **start_ptr** statement, but never gets as far as setting up **temp2**. The code above has to be adapted so that if the first node is also the last (has a NULL **nxt** pointer), then it is deleted and the **start_ptr** pointer is assigned to NULL. In this case, there is no need for the pointer **temp2**:

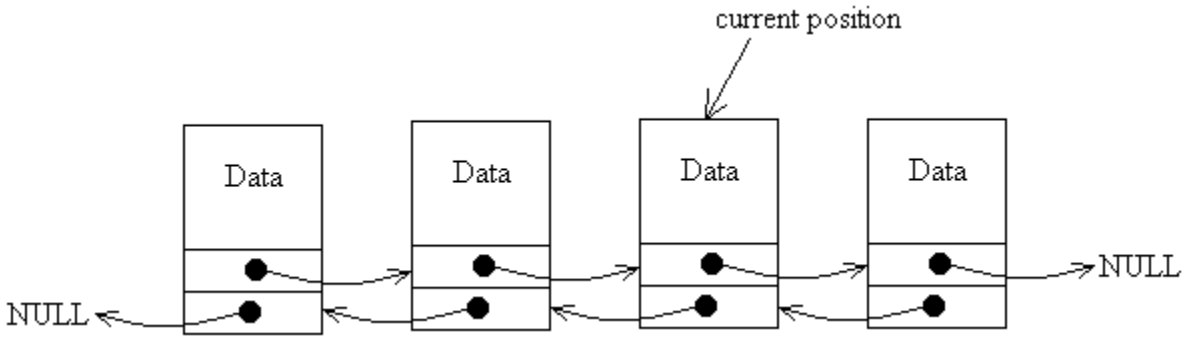
```
void delete_end_node()
```

```
{ node *temp1, *temp2;
 if (start_ptr == NULL)
     cout << "The list is empty!" << endl;
 else
   { temp1 = start_ptr;
    if (temp1->nxt == NULL)     // This part is new!
      { delete temp1;
       start_ptr = NULL;
      }
    else
   { while (temp1->nxt != NULL)
         { temp2 = temp1;
           temp1 = temp1->nxt;
         }
        delete temp1;
        temp2->nxt = NULL;
      }
   }

}
```

3.3. Doubly Linked Lists

That sounds even harder than a linked list! Well, if you've mastered how to do singly linked lists, then it shouldn't be much of a leap to doubly linked lists

A doubly linked list is one where there are links from each node in both directions:



You will notice that each node in the list has two pointers, one to the next node and one to the previous one - again, the ends of the list are defined by NULL pointers. Also there is no pointer to the start of the list. Instead, there is simply a pointer to some position in the list that can be moved left or right.

The reason we needed a start pointer in the ordinary linked list is because, having moved on from one node to another, we can't easily move back, so without the start pointer, we would lose track of all the nodes in the list that we have already passed. With the doubly linked list, we can move the current pointer backwards and forwards at will.

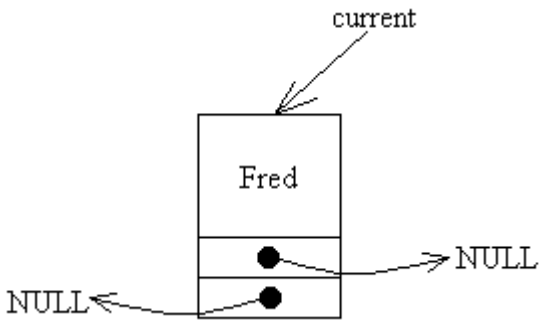
3.3.1. Creating Doubly Linked Lists

The nodes for a doubly linked list would be defined as follows:

```
struct node{
  char name[20];
  node *nxt;    // Pointer to next node
  node *prv;    // Pointer to previous node
 };
node *current;
current = new node;
```

```
current->name = "Fred";
current->nxt = NULL;
current->prv = NULL;
```

We have also included some code to declare the first node and set its pointers to NULL. It gives the following situation:



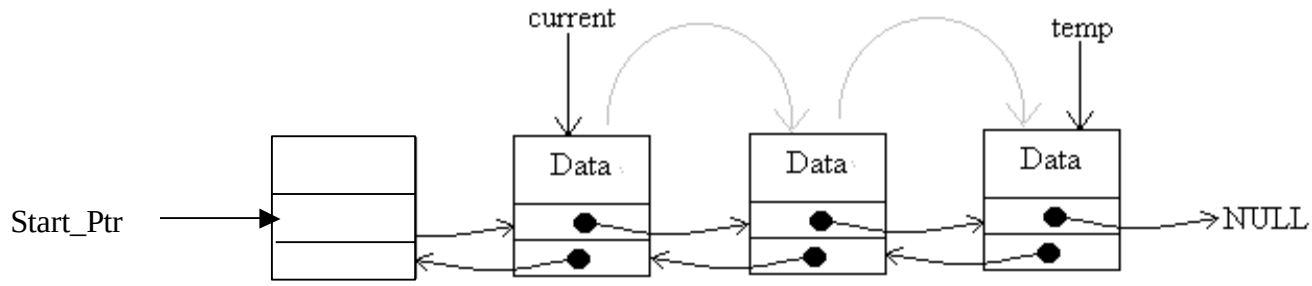
We still need to consider the directions 'forward' and 'backward', so in this case, we will need to define functions to add a node to the start of the list (left-most position) and the end of the list (right-most position).

3.3.2. Adding a Node to a Doubly Linked List

```
void add_node_at_start (string new_name)
 { // Declare a temporary pointer and move it to the start
   node *temp = current;
   while (temp->prv != NULL)
    temp = temp->prv;
   // Declare a new node and link it in
   node *temp2;
   temp2 = new node;
   temp2->name = new_name;  // Store the new name in the node
   temp2->prv = NULL;       // This is the new start of the list
   temp2->nxt = temp;       // Links to current list
   temp->prv = temp2;
 }
```

```
void add_node_at_end ()
 { // Declare a temporary pointer and move it to the end
   node *temp = current;
   while (temp->nxt != NULL)
    temp = temp->nxt;
   // Declare a new node and link it in
   node *temp2;
   temp2 = new node;
   temp2->name = new_name;  // Store the new name in the node
   temp2->nxt = NULL;       // This is the new start of the list
   temp2->prv = temp;       // Links to current list
   temp->nxt = temp2;
 }
```

Here, the new name is passed to the appropriate function as a parameter. We'll go through the function for adding a node to the right-most end of the list. The method is similar for adding a node at the other end. Firstly, a temporary pointer is set up and is made to march along the list until it points to last node in the list.



After that, a new node is declared, and the name is copied into it. The nxt pointer of this new node is set to NULL to indicate that this node will be the new end of the list.

The prv pointer of the new node is linked into the last node of the existing list.

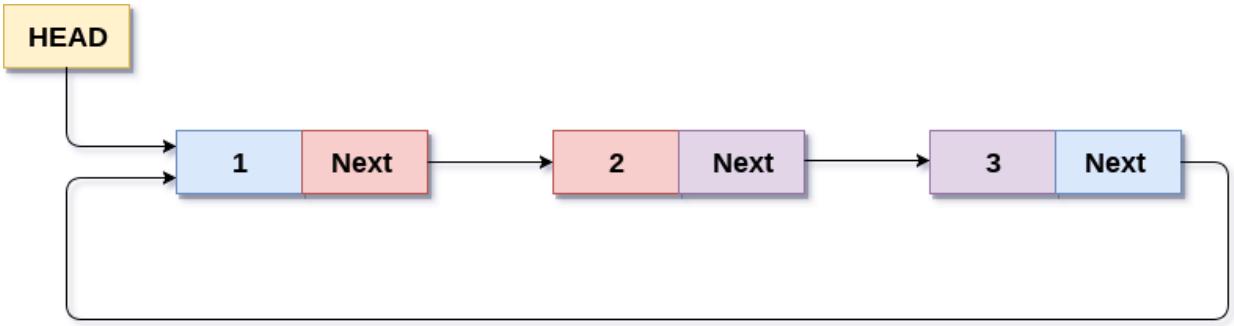
The nxt pointer of the current end of the list is set to the new node.

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



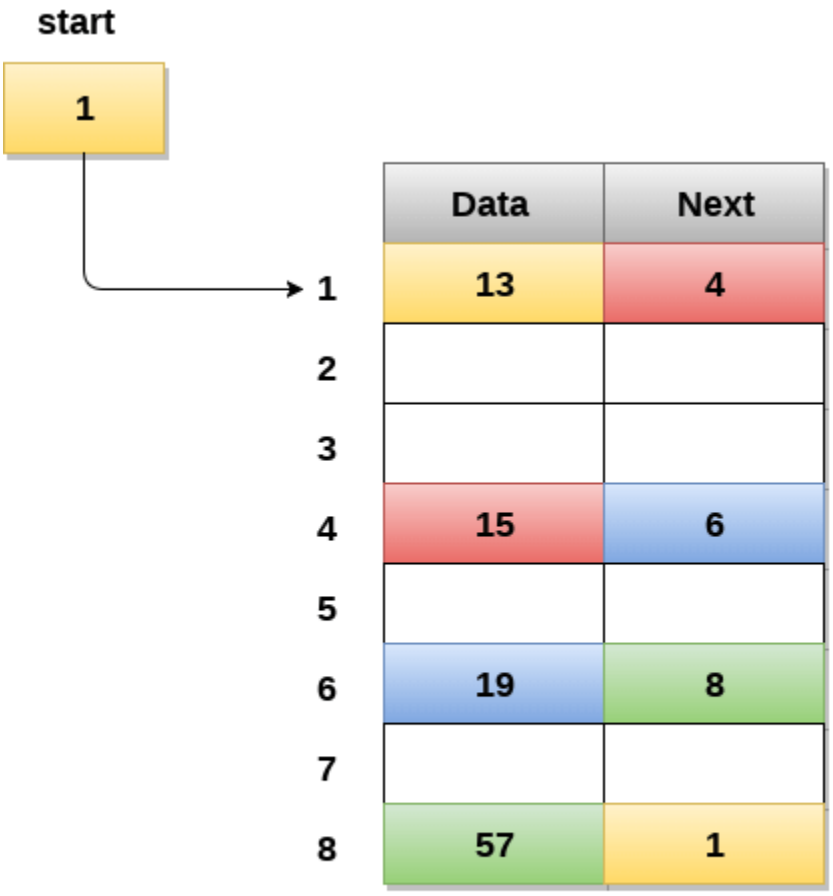
Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



Memory Representation of a circular linked list

We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

Operations on Circular Singly linked list:

Insertion

SN	Operation	Description
1	Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2	Insertion at the end	Adding a node into circular singly linked list at the end.

Deletion & Traversing

SN	Operation	Description
1	Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2	Deletion at the end	Removing the node from circular singly linked list at the end.
3	Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4	Traversing	Visiting each element of the list at least once in order to perform some specific operation.

Chapter 4: Stacks

What is a Stack?

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

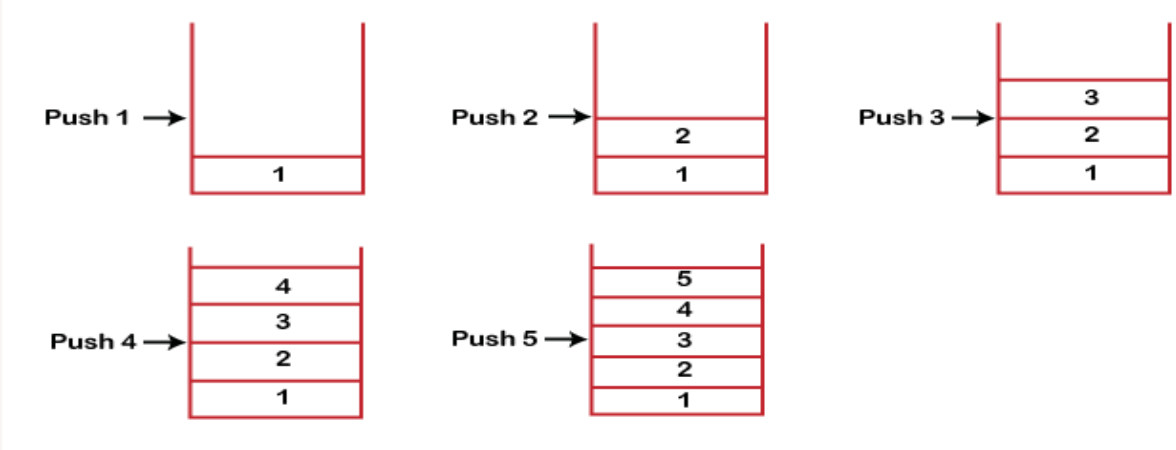
Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

The following are some common operations implemented on the stack:

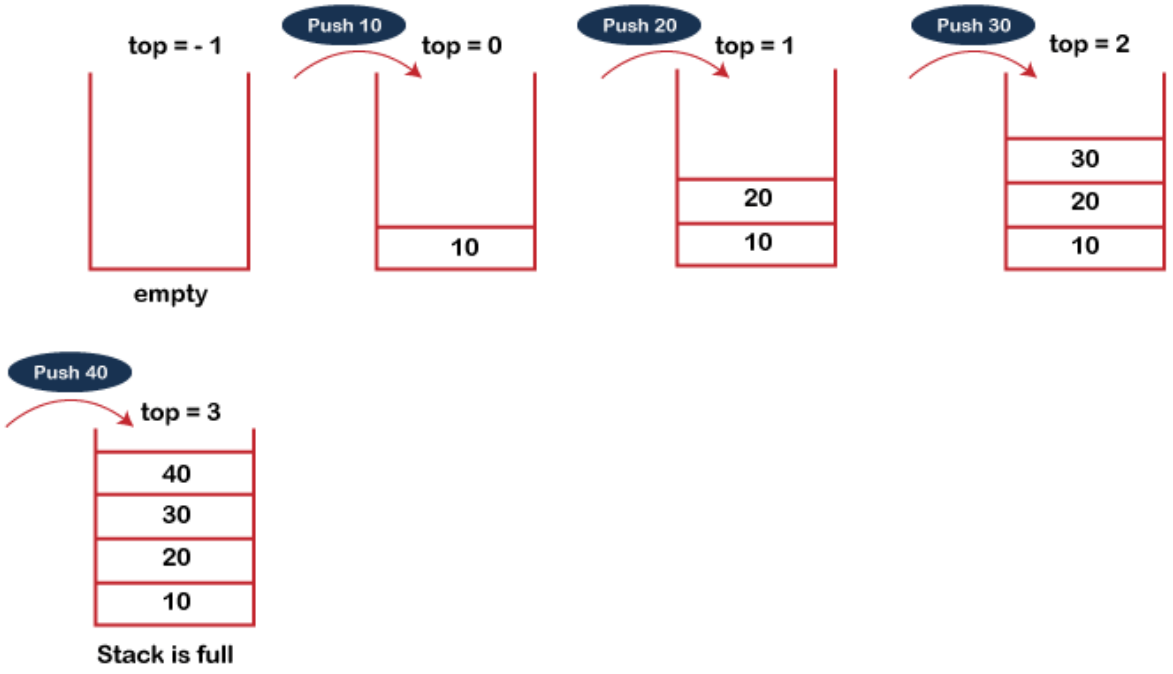
- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH operation

The steps involved in the PUSH operation is given below:

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.

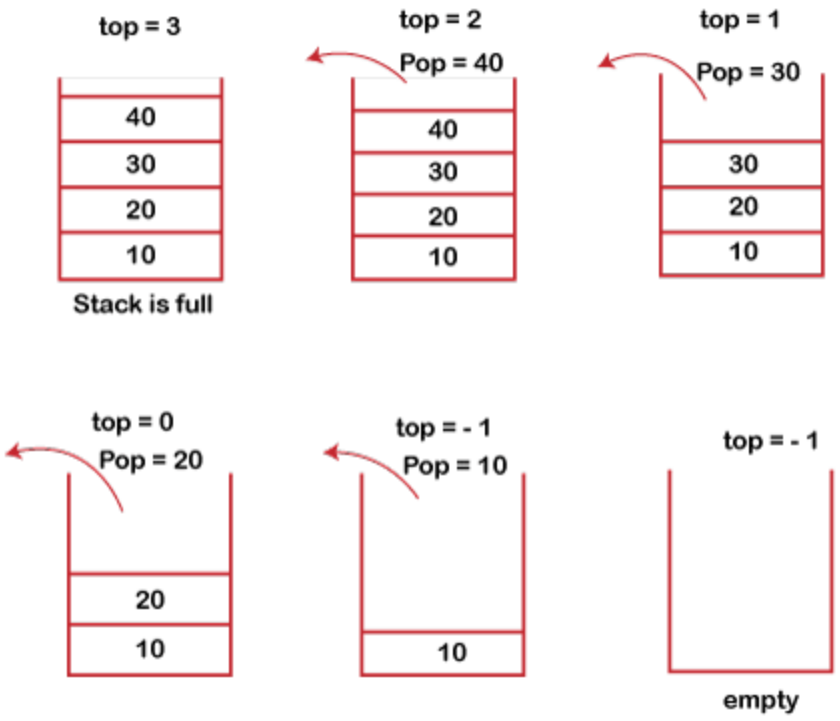
- The elements will be inserted until we reach the ***max*** size of the stack.



POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



pplications of Stack

The following are the applications of the stack:

Balancing of symbols: Stack is used for balancing a symbol. For example, we have the following program:

```
int main()
{
  cout<<"Hello";
  cout<<"javaTpoint";
}
```

As we know, each program has *an opening* and *closing* braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

- **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "**javaTpoint**" string, so we can achieve this with the help of a stack. First, we push all the characters of the string in a stack until we reach the null character. After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
- **UNDO/REDO:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
  If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
- **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
- **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
- **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
- Infix to prefix
- Infix to postfix
- Prefix to infix
- Prefix to postfix
  Postfix to infix

- **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refere to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:

```
begin
if top = n then stack full
   top = top + 1
  stack (top) : = item;
end
```

Time Complexity : o(1)

implementation of push algorithm in C language

```
    void push (int val,int n) //n is size of the stack
{
   if (top == n )
 printf("\n Overflow");
 else
  {
 top = top +1;
 stack[top] = val;      }
}
```

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

```
begin
   if top = 0 then stack empty;
   item := stack(top);
   top = top - 1;
end;
```

Time Complexity : o(1)

Implementation of POP algorithm using C language

```
int pop ()
```

```
{
   if(top == -1)
   {
      printf("Underflow");
      return 0;
   }
   else
   {
      return stack[top - - ];
   }
}
```

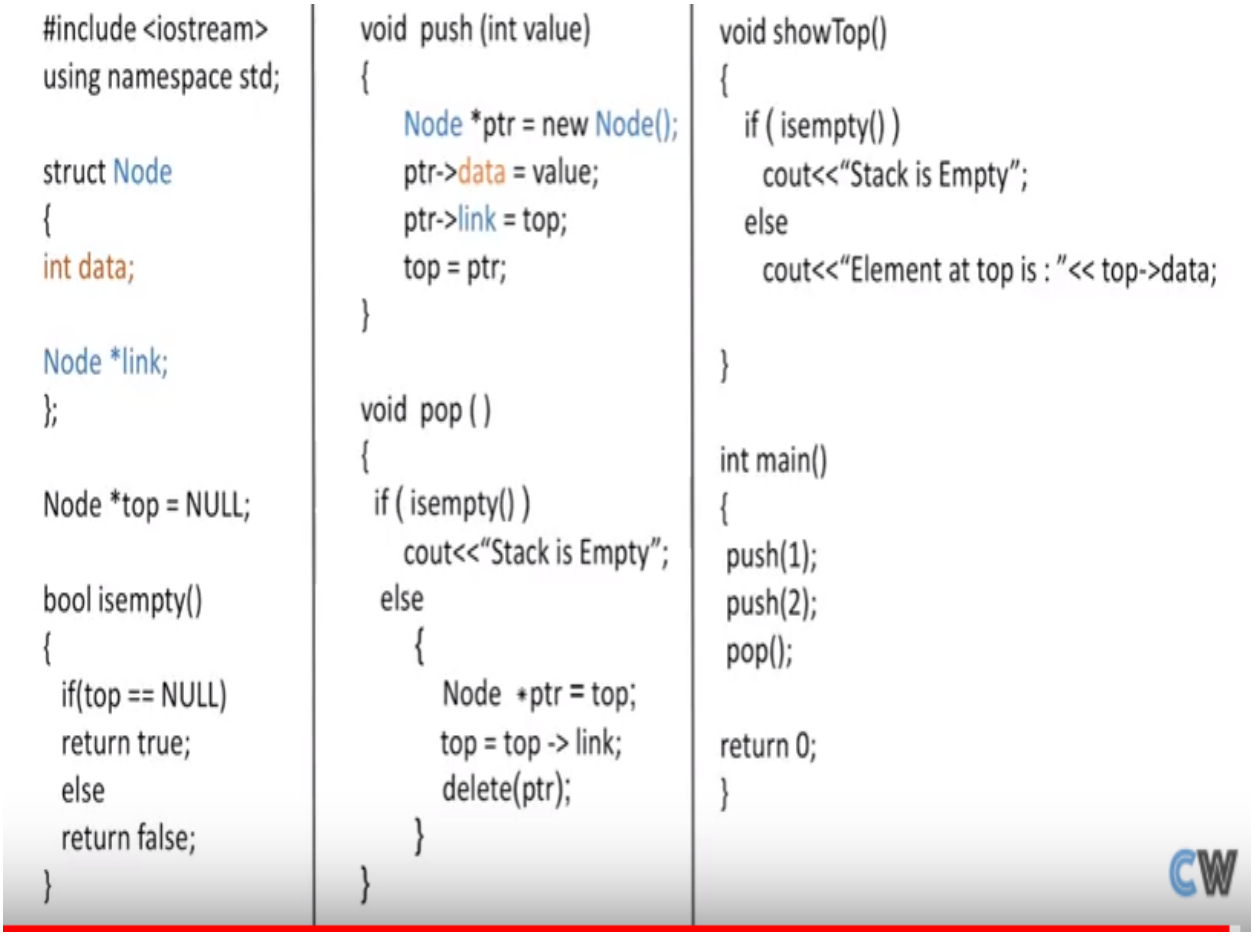
There are two ways we can implement a stack:

- Using an array
- Using a linked list

```
#include <iostream>
using namespace std;
#define SIZE 5
int A[SIZE];
int top=-1;

bool isempty()
{
  if(top==-1)
    return true;
  else
    return false;
}

void  push(int  value)
{
   if(top==SIZE-1)
   {
   cout<<"Stack  is  full!\n";
   }
   else
   {
     top++;
     A[top]=value;
   }
}
```

```
void pop()
{
  if(isempty())
    cout<<"Stack is empty!\n";
  else   top--;
}


void  displayStack()
{
  if(isempty())
   {
    cout<<"Stack  is  empty!\n";
   }
  else {
   for(int  i=0  ;  i<=top;  i++)
        cout<<A[i]<<"  ";
  }
}
```

```
void show_top()
{
   if(isempty())
     cout<<"Stack is empty!\n";
   else
     cout<<"Element at top is: "<<A[top]<<"\n";
}

int main()
{
push(2);
push(3);

displayStack();

pop();

Show_top ();

return 0;
}
```



```
#include <iostream>
using namespace std;

struct Node
{
int data;

Node *link;
};

Node *top = NULL;

bool isempty()
{
  if(top == NULL)
  return true;
  else
  return false;
}
```

```
void  push (int value)
{
    Node *ptr = new Node();
    ptr->data = value;
    ptr->link = top;
    top = ptr;
}

void  pop ( )
{
 if ( isempty() )
    cout<<"Stack is Empty";
 else
    {
       Node  *ptr = top;
       top = top -> link;
       delete(ptr);
    }
}
```

```
void showTop()
{
  if ( isempty() )
    cout<<"Stack is Empty";
  else
    cout<<"Element at top is : "<< top->data;

}

int main()
{
 push(1);
 push(2);
 pop();

return 0;
}
```

Chapter Five

Queue

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to be as First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.

Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

Complexity

Data Structure	Time Complexity								Space Compleity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Types of Queue

In this article, we will discuss the types of queue. But before moving towards the types, we should first discuss the brief introduction of the queue.

What is a Queue?

Queue is the data structure that is similar to the queue in the real world. A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Queue can also be defined as the list or collection in which the insertion is done from one end known as the **rear end** or the **tail** of the queue, whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the last person enters in the queue gets the ticket at last. Similar approach is followed in the queue in data structure.

Chapter Six

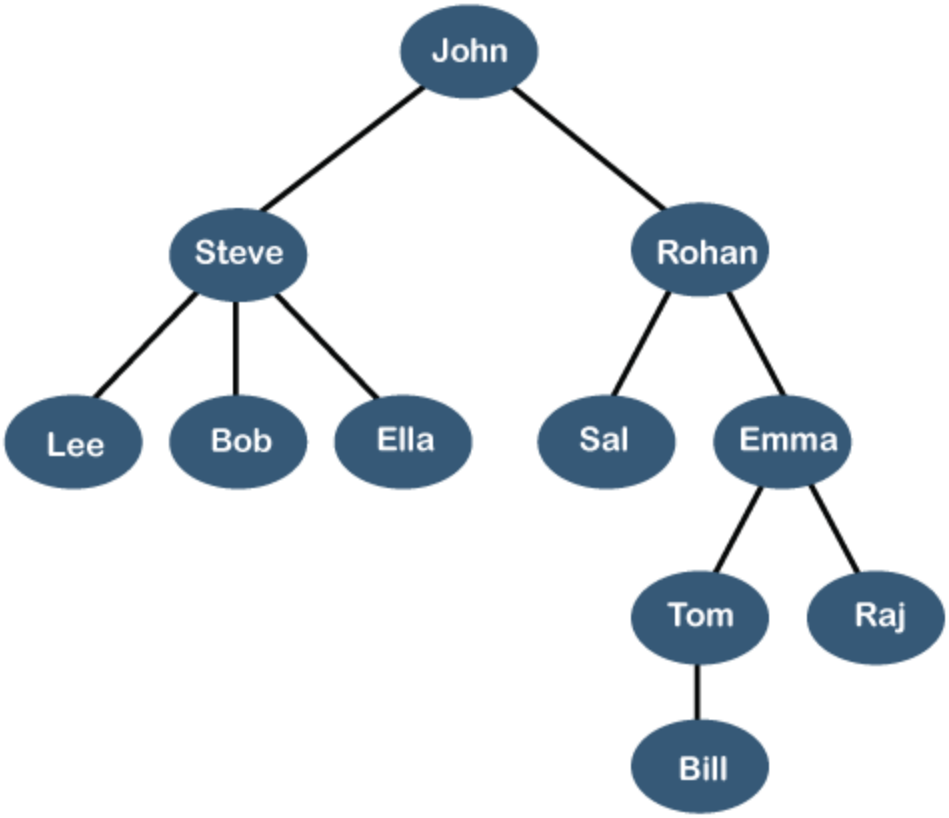
Tree Data Structure

We read the linear data structures like an array, linked list, stack and queue in which all the elements are arranged in a sequential manner. The different data structures are used for different kinds of data.

Some factors are considered for choosing the data structure:

- **What type of data needs to be stored**?: It might be a possibility that a certain data structure can be the best fit for some kind of data.
- **Cost of operations:** If we want to minimize the cost for the operations for the most frequently performed operations. For example, we have a simple list on which we have to perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the ***binary search***. The binary search works very fast for the simple list as it divides the search space into half.
- **Memory usage:** Sometimes, we want a data structure that utilizes less memory.

A tree is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:



The above tree shows the **organization hierarchy** of some company. In the above structure, ***john*** is the **CEO** of the company, and John has two direct reports named as ***Steve*** and ***Rohan***. Steve has three direct reports named ***Lee, Bob, Ella*** where ***Steve*** is a manager. Bob has two direct reports named ***Sal*** and ***Emma***. **Emma** has two direct reports named ***Tom*** and ***Raj***. Tom has one direct report named ***Bill***. This particular logical structure is known as a ***Tree***. Its structure is similar to the real tree, so it is named a ***Tree***. In this structure, the ***root*** is at the top, and its branches are moving in a downward direction. Therefore, we can say that the Tree data structure is an efficient way of storing the data in a hierarchical way.

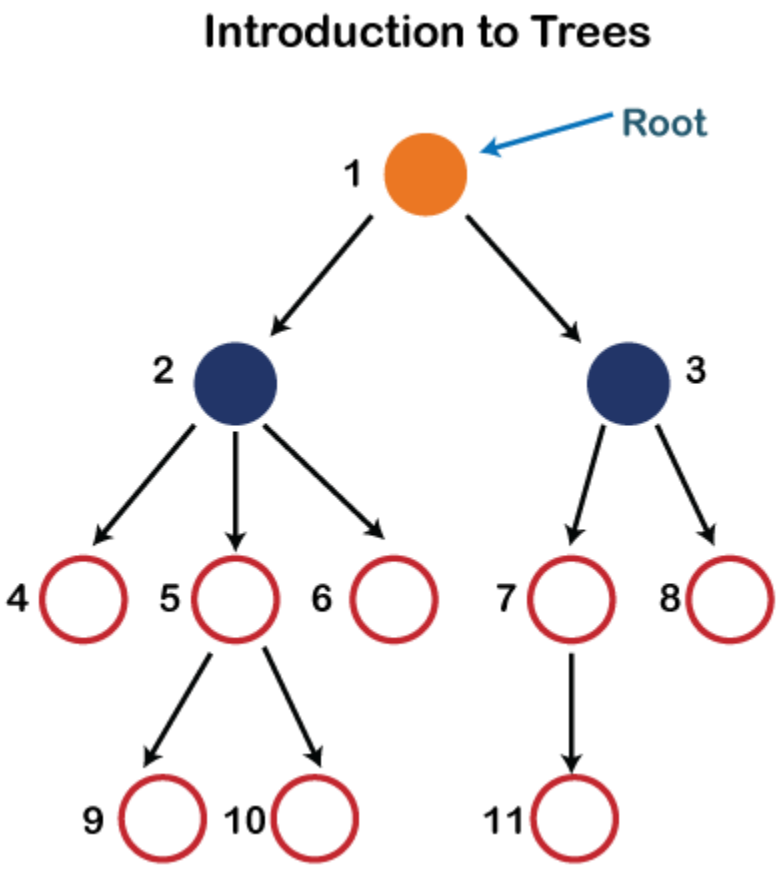
Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.

- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.

Some basic terms used in Tree data structure.

Let's consider the tree structure, which is shown below:



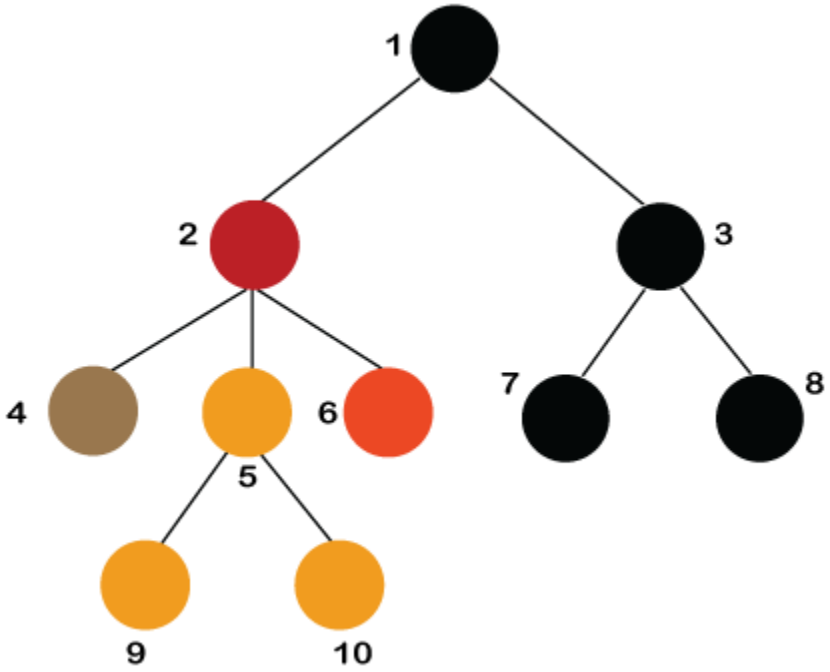
In the above structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- **Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various

applications.

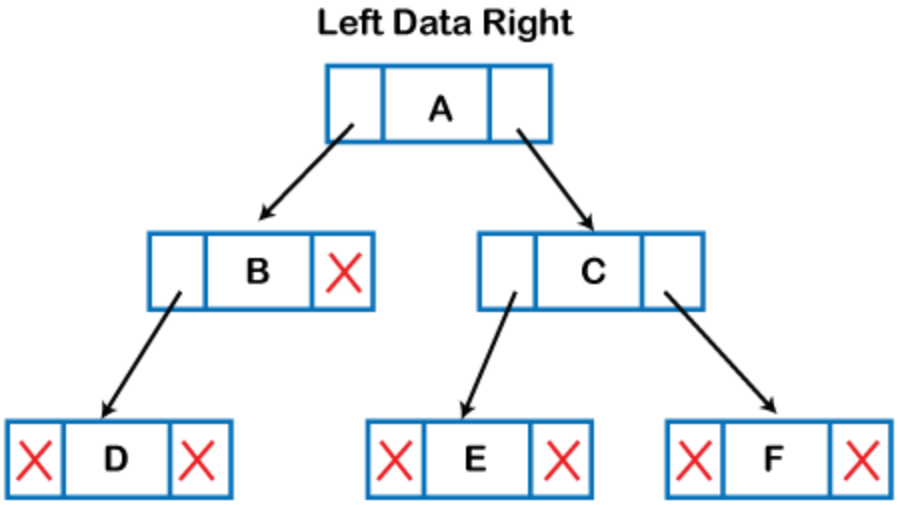


- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

Based on the properties of the Tree data structure, trees are classified into various categories.

Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:

```
1. struct node
2. {
3.   int data;
4. struct node *left;
5. struct node *right;
6. }
```

The above structure can only be defined for the binary trees because the binary tree can have utmost two children, and generic trees can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Applications of trees

The following are the applications of trees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file

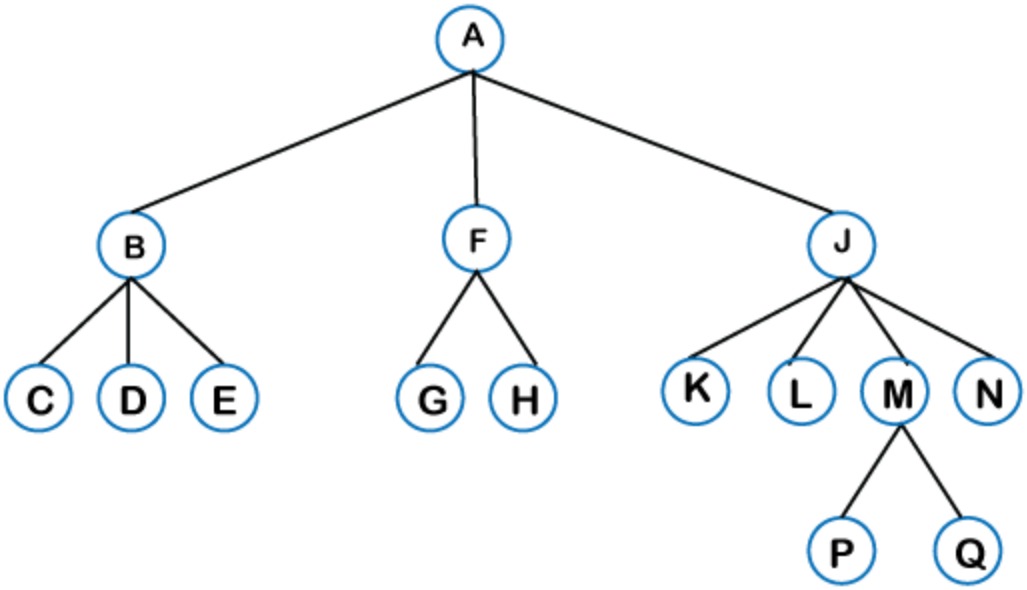
and folder are in the form of the naturally hierarchical data and stored in the form of trees.

- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

Types of Tree data structure

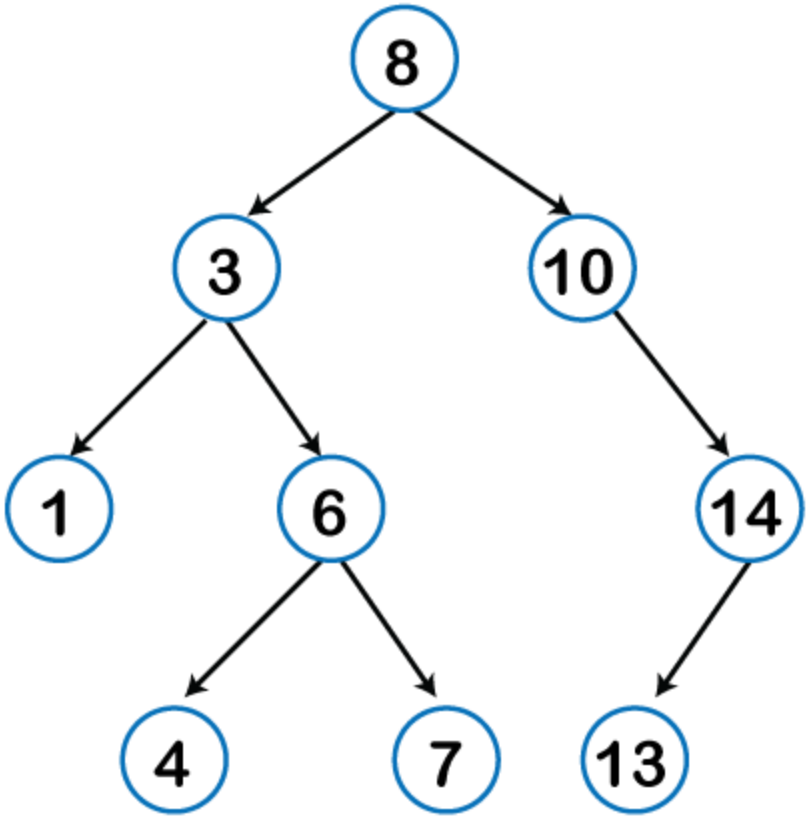
The following are the types of a tree data structure:

- **General tree:** The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as ***subtrees***.



There can be ***n*** number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered. Every non-empty tree has a downward edge, and these edges are connected to the nodes known as ***child nodes***. The root node is labeled with level 0. The nodes that have the same parent are known as ***siblings***.

- **Binary tree:** Here, binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have utmost two child nodes. Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



To know more about the binary tree, click on the link given below: https://www.javatpoint.com/binary-tree

- **Binary Search tree:** Binary search tree is a non-linear data structure in which one node is connected to ***n*** number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer). Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.

A node can be created with the help of a user-defined data type known as ***struct,*** as shown below:

```
1. struct node
2. {
3.     int data;
4.     struct node *left;
5. struct node *right;
6. }
```

The above is the node structure with three fields: data field, the second field is the left pointer of the node type, and the third field is the right pointer of the node type.

To know more about the binary search tree, click on the link given below:

https://www.javatpoint.com/binary-search-tree

- **AVL tree**

It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the ***binary tree*** as well as of the ***binary search tree***. It is a self-balancing binary search tree that was invented by ***Adelson Velsky Lindas***. Here, self-balancing means that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the ***balancing factor***.

We can consider a tree as an AVL tree if the tree obeys the binary search tree as well as a balancing factor. The balancing factor can be defined as the ***difference between the height of the left subtree and the height of the right subtree***. The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1, or 1.

To know more about the AVL tree, click on the link given below:

https://www.javatpoint.com/avl-tree

- **Red-Black Tree**

The red-Black tree is the binary search tree. The prerequisite of the Red-Black tree is that we should know about the binary search tree. In a binary search tree, the value of the left-subtree should be less than the value of that node, and the value of the right-subtree should be greater than the value of that node. As we know that the time complexity of binary search in the average case is $\log_2 n$, the best case is O(1), and the worst case is O(n).

When any operation is performed on the tree, we want our tree to be balanced so that all the operations like searching, insertion, deletion, etc., take less time, and all these operations will have the time complexity of $\boldsymbol{\log_2 n}$.

The red-black tree is a self-balancing binary search tree. AVL tree is also a height balancing binary search tree then **why do we require a Red-Black tree**. In the AVL tree, we do not know how many rotations would be required to balance the tree, but in the Red-black tree, a maximum of 2 rotations are required to balance the tree. It contains one extra bit that represents either the red or black color of a node to ensure the balancing of the tree.

- **Splay tree**

The splay tree data structure is also binary search tree in which recently accessed element is placed at the root position of tree by performing some rotation operations. Here, ***splaying*** means the recently accessed node. It is a ***self-balancing*** binary search tree having no explicit balance condition like **AVL** tree.

It might be a possibility that height of the splay tree is not balanced, i.e., height of both left and right subtrees may differ, but the operations in splay tree takes order of **logN** time where **n** is the number of nodes.

Splay tree is a balanced tree but it cannot be considered as a height balanced tree because after each operation, rotation is performed which leads to a balanced tree.

- **Treap**

Treap data structure came from the Tree and Heap data structure. So, it comprises the properties of both Tree and Heap data structures. In Binary search tree, each node on the left subtree must be equal or less than the value of the root node and each node on the right subtree must be equal or greater than the value of the root node. In heap data structure, both right and left subtrees contain larger keys than the root; therefore, we can say that the root node contains the lowest value.

In treap data structure, each node has both ***key*** and ***priority*** where key is derived from the Binary search tree and priority is derived from the heap data structure.

The **Treap** data structure follows two properties which are given below:

- Right child of a node>=current node and left child of a node <=current node (binary tree)
- Children of any subtree must be greater than the node (heap)

- **B-tree**

B-tree is a balanced **m-way** tree where **m** defines the order of the tree. Till now, we read that the node contains only one key but b-tree can have more than one key, and more than 2 children. It always maintains the sorted data. In binary tree, it is possible that leaf nodes can be at different levels, but in b-tree, all the leaf nodes must be at the same level.

If order is m then node has the following properties:

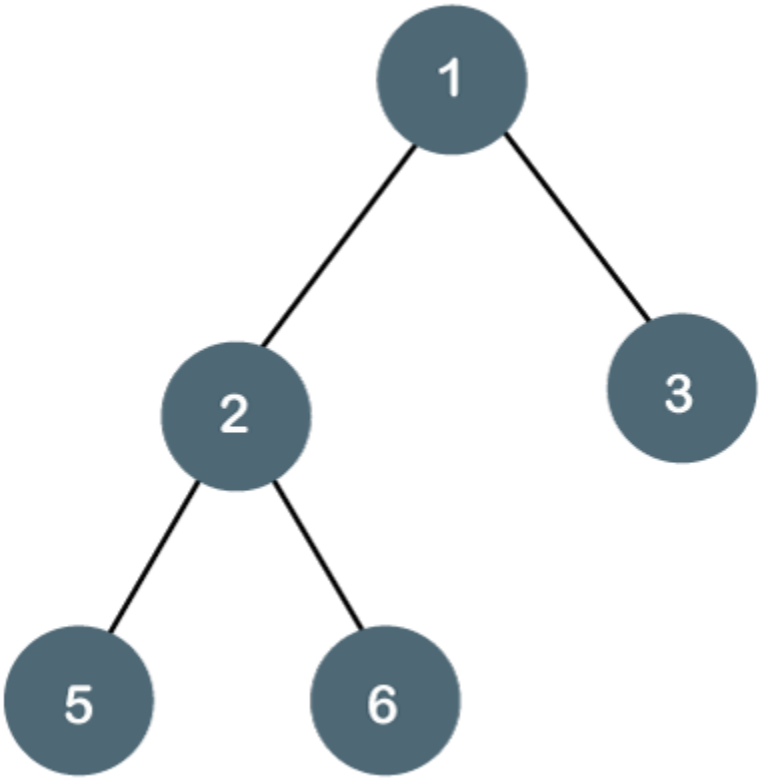
- Each node in a b-tree can have maximum **m** children
- For minimum children, a leaf node has 0 children, root node has minimum 2 children and internal node has minimum ceiling of m/2 children. For example, the value of m is 5 which means that a node can have 5 children and internal nodes can contain maximum 3 children.
- Each node has maximum (m-1) keys.

The root node must contain minimum 1 key and all other nodes must contain atleast **ceiling of m/2 minus 1** keys.

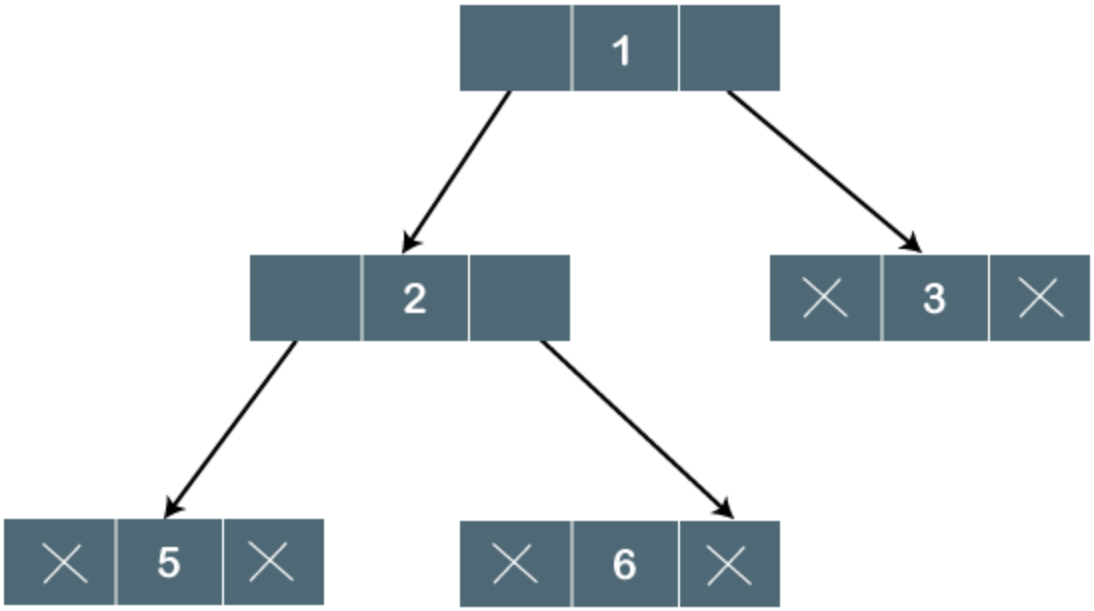
Binary Tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Let's understand the binary tree through an example.



The above tree is a binary tree because each node contains the utmost two children. The logical representation of the above tree is given below:



In the above tree, node 1 contains two pointers, i.e., left and a right pointer pointing to the left and right node respectively. The node 2 contains both the nodes (left and right node); therefore, it has two pointers (left and right). The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contain **NULL** pointer on both left and right parts.

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \ldots 2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

If there are 'n' number of nodes in the binary tree.

The minimum height can be computed as:

As we know that,

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

Taking log on both the sides,

$\log_2(n+1) = \log2(2^{h+1})$

$\log_2(n+1) = h+1$

$h = \log_2(n+1) - 1$

The maximum height can be computed as:

As we know that,

$n = h+1$

$h = n-1$

Types of Binary Tree

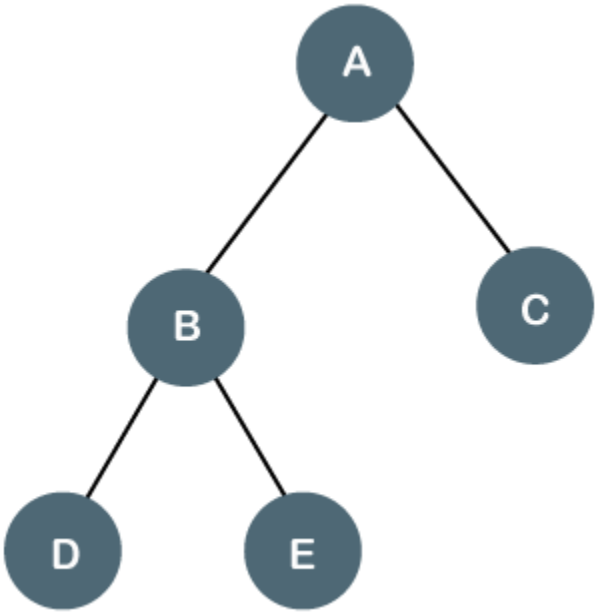
There are four types of Binary tree:

- **Full/ proper/ strict Binary tree**
- **Complete Binary tree**
- **Perfect Binary tree**
- **Degenerate Binary tree**
- **Balanced Binary tree**

1. Full/ proper/ strict Binary tree

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

Let's look at the simple example of the Full Binary tree.



In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., 2^{h+1} -1.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of the full binary tree can be computed as:

n= 2*h - 1

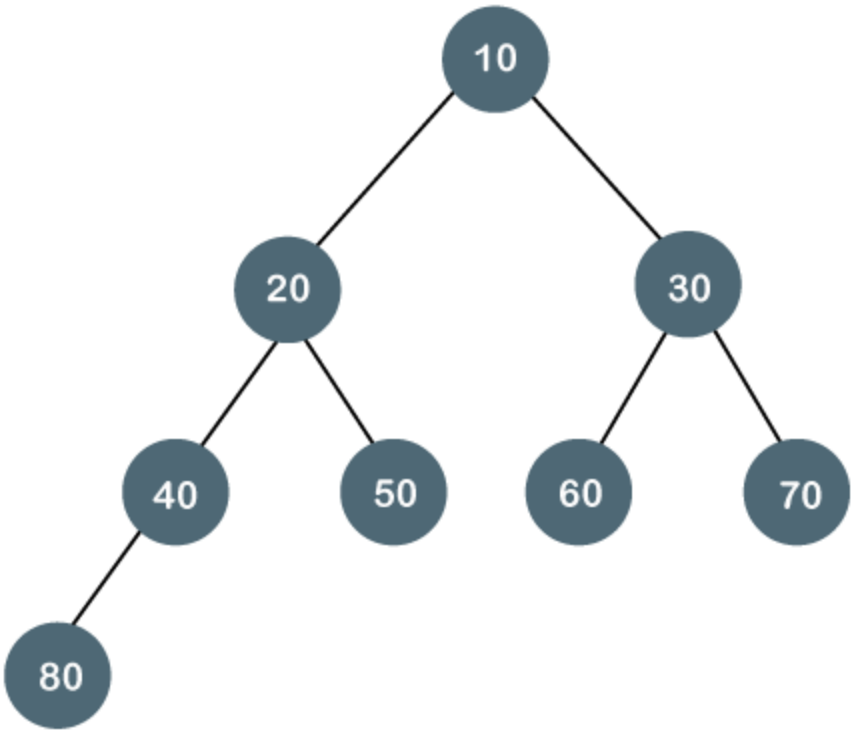
n+1 = 2*h

h = n+1/2

Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

Let's create a complete binary tree.



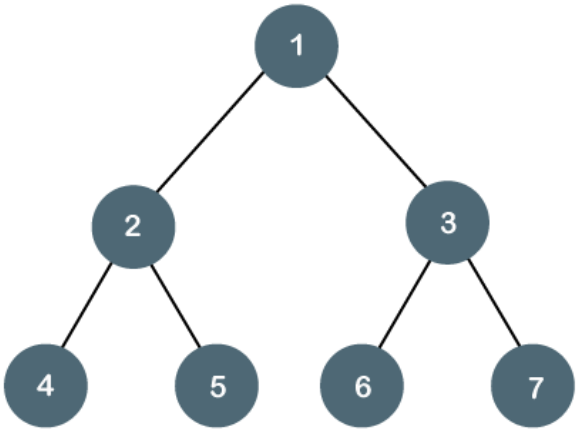
The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

Properties of Complete Binary Tree

- The maximum number of nodes in complete binary tree is $2^{h+1} - 1$.
- The minimum number of nodes in complete binary tree is 2^{h}.
- The minimum height of a complete binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of a complete binary tree is

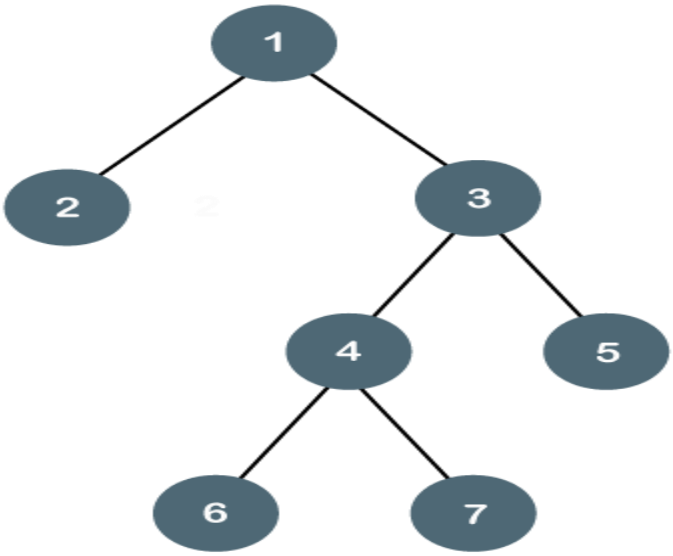
Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



Let's look at a simple example of a perfect binary tree.

The below tree is not a perfect binary tree because all the leaf nodes are not at the same level.

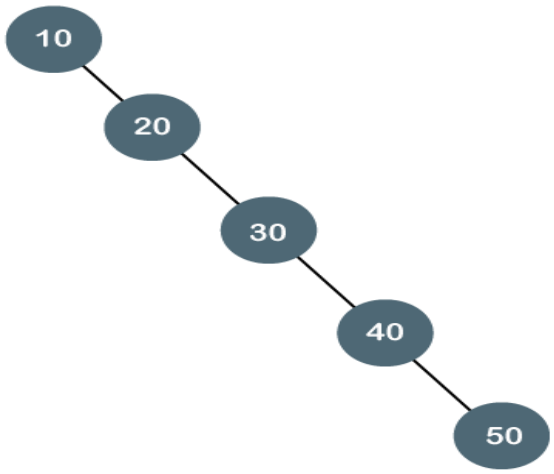


Note: All the perfect binary trees are the complete binary trees as well as the full binary tree, but vice versa is not true, i.e., all complete binary trees and full binary trees are the perfect binary trees.

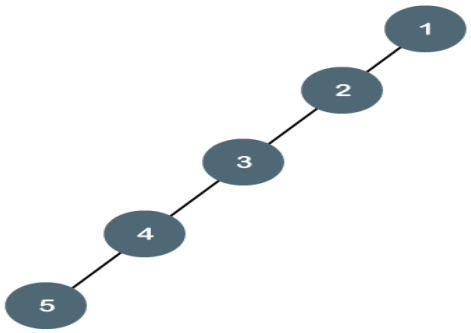
Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one children.

Let's understand the Degenerate binary tree through examples.



The above tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.

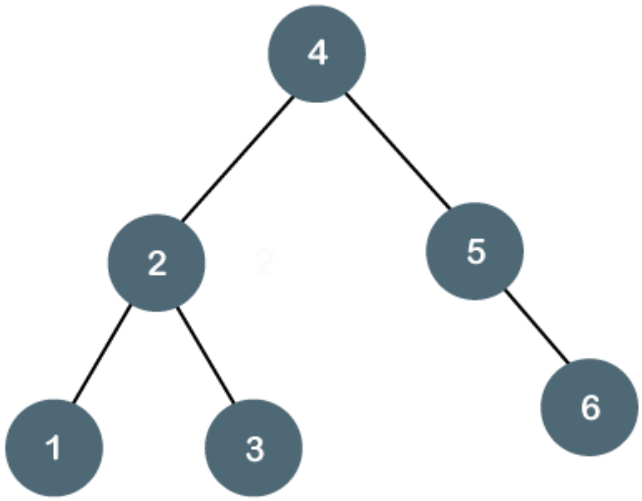


The above tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.

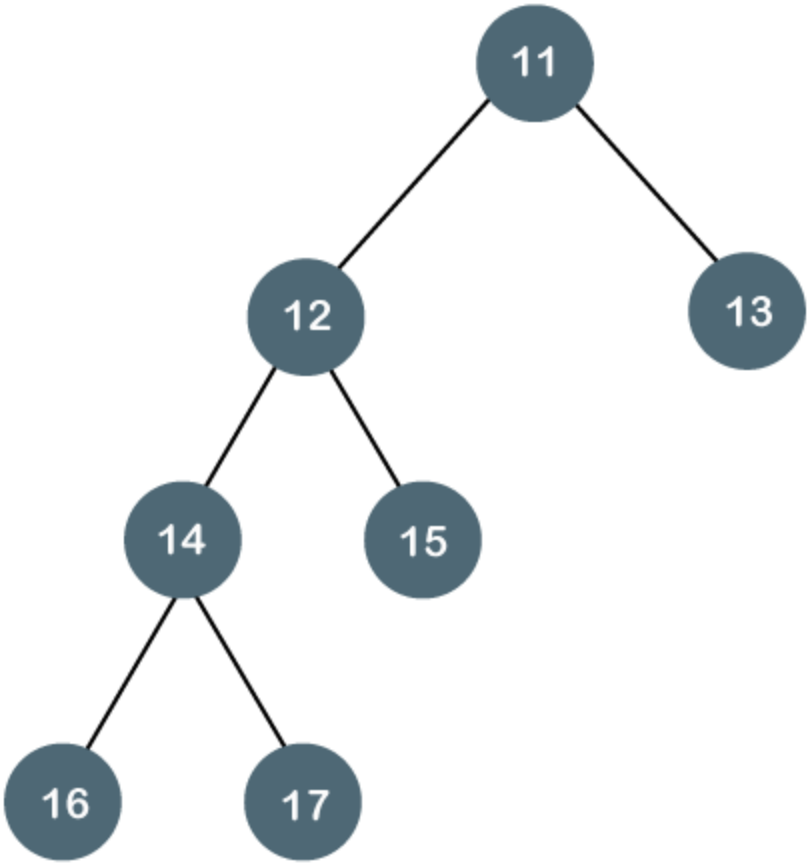
Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by atmost 1. For example, *AVL* and *Red-Black trees* are balanced binary tree.

Let's understand the balanced binary tree through examples.



The above tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.



The above tree is not a balanced binary tree because the difference between the left subtree and the right subtree is greater than 1.

Binary Tree Implementation

A Binary tree is implemented with the help of pointers. The first node in the tree is represented by the root pointer. Each node in the tree consists of three parts, i.e., data, left pointer and right pointer. To create a binary tree, we first need to create the node.

Tree traversal (Inorder, Preorder an Postorder)

In this article, we will discuss the tree traversal in the data structure. The term 'tree traversal' means traversing or visiting each node of a tree. There is a single way to traverse the linear data structure such as linked list, queue, and stack. Whereas, there are multiple ways to traverse a tree that are listed as follows -

- Preorder traversal
- Inorder traversal
- Postorder traversal

So, in this article, we will discuss the above-listed techniques of traversing a tree. Now, let's start discussing the ways of tree traversal.

Preorder traversal

This technique follows the 'root left right' policy. It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.

So, in a preorder traversal, each node is visited before both of its subtrees.

The applications of preorder traversal include -

- It is used to create a copy of the tree.
- It can also be used to get the prefix expression of an expression tree.

Algorithm

1. Until all nodes of the tree are not visited
2.
3. Step 1 - Visit the root node
4. Step 2 - Traverse the left subtree recursively.
5. Step 3 - Traverse the right subtree recursively.

Example

Now, let's see the example of the preorder traversal technique.

Now, start applying the preorder traversal on the above tree. First, we traverse the root node **A;** after that, move to its left subtree **B**, which will also be traversed in preorder.

So, for left subtree B, first, the root node **B** is traversed itself; after that, its left subtree **D** is traversed. Since node **D** does not have any children, move to right subtree **E**. As node E also does not have any children, the traversal of the left subtree of root node A is completed.

Now, move towards the right subtree of root node A that is C. So, for right subtree C, first the root node **C** has traversed itself; after that, its left subtree **F** is traversed. Since node **F** does not have any children, move to the right subtree **G**. As node G also does not have any children, traversal of the right subtree of root node A is completed.

Therefore, all the nodes of the tree are traversed. So, the output of the preorder traversal of the above tree is -

A → B → D → E → C → F → G

To know more about the preorder traversal in the data structure, you can follow the link Preorder traversal.

Postorder traversal

This technique follows the 'left-right root' policy. It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.

So, in a postorder traversal, each node is visited after both of its subtrees.

The applications of postorder traversal include -

- It is used to delete the tree.
- It can also be used to get the postfix expression of an expression tree.

Algorithm

1. Until all nodes of the tree are not visited
2.
3. Step 1 - Traverse the left subtree recursively.
4. Step 2 - Traverse the right subtree recursively.
5. Step 3 - Visit the root node.

Example

Now, let's see the example of the postorder traversal technique.

Now, start applying the postorder traversal on the above tree. First, we traverse the left subtree B that will be traversed in postorder. After that, we will traverse the right subtree **C** in postorder. And finally, the root node of the above tree, i.e., **A**, is traversed.

So, for left subtree B, first, its left subtree **D** is traversed. Since node **D** does not have any children, traverse the right subtree **E**. As node E also does not have any children, move to the root node **B.** After traversing node **B,** the traversal of the left subtree of root node A is completed.

Now, move towards the right subtree of root node A that is C. So, for right subtree C, first its left subtree **F** is traversed. Since node **F** does not have any children, traverse the right subtree **G**. As node G also does not have any children, therefore, finally, the root node of the right subtree, i.e., **C,** is traversed. The traversal of the right subtree of root node A is completed.

At last, traverse the root node of a given tree, i.e., **A**. After traversing the root node, the postorder traversal of the given tree is completed.

Therefore, all the nodes of the tree are traversed. So, the output of the postorder traversal of the above tree is -

D → E → B → F → G → C → A

To know more about the postorder traversal in the data structure, you can follow the link Postorder traversal.

Inorder traversal

This technique follows the 'left root right' policy. It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal.

So, in the inorder traversal, each node is visited in between of its subtrees.

The applications of Inorder traversal includes -

- It is used to get the BST nodes in increasing order.
- It can also be used to get the prefix expression of an expression tree.

Algorithm

1. Until all nodes of the tree are not visited
2.
3. Step 1 - Traverse the left subtree recursively.
4. Step 2 - Visit the root node.
5. Step 3 - Traverse the right subtree recursively.

Example

Now, let's see the example of the Inorder traversal technique.

Now, start applying the inorder traversal on the above tree. First, we traverse the left subtree **B** that will be traversed in inorder. After that, we will traverse the root node **A**. And finally, the right subtree **C** is traversed in inorder.

So, for left subtree **B**, first, its left subtree **D** is traversed. Since node **D** does not have any children, so after traversing it, node **B** will be traversed, and at last, right subtree of node B, that is **E**, is traversed. Node E also does not have any children; therefore, the traversal of the left subtree of root node A is completed.

After that, traverse the root node of a given tree, i.e., **A**.

At last, move towards the right subtree of root node A that is C. So, for right subtree C; first, its left subtree **F** is traversed. Since node **F** does not have any children, node **C** will be traversed, and at last, a right subtree of node C, that is, **G**, is traversed. Node G also does not have any children; therefore, the traversal of the right subtree of root node A is completed.

As all the nodes of the tree are traversed, the inorder traversal of the given tree is completed. The output of the inorder traversal of the above tree is -

D → B → E → A → F → C → G

To know more about the inorder traversal in data structure, you can follow the link Inorder Traversal.

Complexity of Tree traversal techniques

The time complexity of tree traversal techniques discussed above is **O(n)**, where '**n**' is the size of binary tree.

Whereas the space complexity of tree traversal techniques discussed above is **O(1)** if we do not consider the stack size for function calls. Otherwise, the space complexity of these techniques is **O(h)**, where '**h**' is the tree's height.

Implementation of Tree traversal

Now, let's see the implementation of the above-discussed techniques using different programming languages.

Program: Write a program to implement tree traversal techniques in C.

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. struct node {
5.     int element;
6.     struct node* left;
7.     struct node* right;
8. };
9.
10. /*To create a new node*/
11. struct node* createNode(int val)
12. {
13.     struct node* Node = (struct node*)malloc(sizeof(struct node));
14.     Node->element = val;
15.     Node->left = NULL;
16.     Node->right = NULL;
17.
18.     return (Node);
19. }
20.
21. /*function to traverse the nodes of binary tree in preorder*/
22. void traversePreorder(struct node* root)
23. {
24.     if (root == NULL)
25.         return;
26.     printf(" %d ", root->element);
27.     traversePreorder(root->left);
```

```
28.     traversePreorder(root->right);
29. }
30.
31.
32. /*function to traverse the nodes of binary tree in Inorder*/
33. void traverseInorder(struct node* root)
34. {
35.     if (root == NULL)
36.         return;
37.     traverseInorder(root->left);
38.     printf(" %d ", root->element);
39.     traverseInorder(root->right);
40. }
41.
42. /*function to traverse the nodes of binary tree in postorder*/
43. void traversePostorder(struct node* root)
44. {
45.     if (root == NULL)
46.         return;
47.     traversePostorder(root->left);
48.     traversePostorder(root->right);
49.     printf(" %d ", root->element);
50. }
51.
52.
53. int main()
54. {
55.     struct node* root = createNode(36);
56.     root->left = createNode(26);
57.     root->right = createNode(46);
58.     root->left->left = createNode(21);
```

```
    root->left->right = createNode(31);
    root->left->left->left = createNode(11);
    root->left->left->right = createNode(24);
    root->right->left = createNode(41);
    root->right->right = createNode(56);
    root->right->right->left = createNode(51);
    root->right->right->right = createNode(66);

    printf("\n The Preorder traversal of given binary tree is -\n");
    traversePreorder(root);

    printf("\n The Inorder traversal of given binary tree is -\n");
    traverseInorder(root);

    printf("\n The Postorder traversal of given binary tree is -\n");
    traversePostorder(root);

    return 0;
}
```

Output

Program: Write a program to implement tree traversal techniques in C#.

```
using System;

class Node {
    public int value;
    public Node left, right;

```

```
7.      public Node(int element)
8.      {
9.          value = element;
10.         left = right = null;
11.     }
12. }
13.
14. class BinaryTree {
15.     Node root;
16.
17.     BinaryTree() { root = null; }
18.     void traversePreorder(Node node)
19.     {
20.         if (node == null)
21.             return;
22.         Console.Write(node.value + " ");
23.         traversePreorder(node.left);
24.         traversePreorder(node.right);
25.     }
26.
27.     void traverseInorder(Node node)
28.     {
29.         if (node == null)
30.             return;
31.         traverseInorder(node.left);
32.         Console.Write(node.value + " ");
33.         traverseInorder(node.right);
34.     }
35.
36.     void traversePostorder(Node node)
37.     {
```

```
38.         if (node == null)
39.             return;
40.         traversePostorder(node.left);
41.         traversePostorder(node.right);
42.         Console.Write(node.value + " ");
43.     }
44.
45.
46.     void traversePreorder() { traversePreorder(root); }
47.     void traverseInorder() { traverseInorder(root); }
48.     void traversePostorder() { traversePostorder(root); }
49.
50.     static void Main()
51.     {
52.         BinaryTree bt = new BinaryTree();
53.         bt.root = new Node(37);
54.         bt.root.left = new Node(27);
55.         bt.root.right = new Node(47);
56.         bt.root.left.left = new Node(22);
57.         bt.root.left.right = new Node(32);
58.         bt.root.left.left.left = new Node(12);
59.         bt.root.left.left.right = new Node(25);
60.         bt.root.right.left = new Node(42);
61.         bt.root.right.right = new Node(57);
62.         bt.root.right.right.left = new Node(52);
63.         bt.root.right.right.right = new Node(67);
64.         Console.WriteLine("The Preorder traversal of given binary tree is - ");
65.         bt.traversePreorder();
66.         Console.WriteLine();
67.         Console.WriteLine("The Inorder traversal of given binary tree is - ");
68.         bt.traverseInorder();
```

```
69.         Console.WriteLine();
70.         Console.WriteLine("The Postorder traversal of given binary tree is - ");
71.         bt.traversePostorder();
72.     }
73. }
```

Output

Program: Write a program to implement tree traversal techniques in C++.

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  struct node {
6.      int element;
7.      struct node* left;
8.      struct node* right;
9.  };
10.
11. /*To create a new node*/
12. struct node* createNode(int val)
13. {
14.     struct node* Node = (struct node*)malloc(sizeof(struct node));
15.     Node->element = val;
16.     Node->left = NULL;
17.     Node->right = NULL;
18.
19.     return (Node);
20. }
```

```
21.
22. /*function to traverse the nodes of binary tree in preorder*/
23. void traversePreorder(struct node* root)
24. {
25.     if (root == NULL)
26.         return;
27.     cout<<" "<<root->element<<" ";
28.     traversePreorder(root->left);
29.     traversePreorder(root->right);
30. }
31.
32. /*function to traverse the nodes of binary tree in Inorder*/
33. void traverseInorder(struct node* root)
34. {
35.     if (root == NULL)
36.         return;
37.     traverseInorder(root->left);
38.     cout<<" "<<root->element<<" ";
39.     traverseInorder(root->right);
40. }
41.
42. /*function to traverse the nodes of binary tree in postorder*/
43. void traversePostorder(struct node* root)
44. {
45.     if (root == NULL)
46.         return;
47.     traversePostorder(root->left);
48.     traversePostorder(root->right);
49.     cout<<" "<<root->element<<" ";
50. }
51.
```

```
52. int main()
53. {
54.     struct node* root = createNode(38);
55.     root->left = createNode(28);
56.     root->right = createNode(48);
57.     root->left->left = createNode(23);
58.     root->left->right = createNode(33);
59.     root->left->left->left = createNode(13);
60.     root->left->left->right = createNode(26);
61.     root->right->left = createNode(43);
62.     root->right->right = createNode(58);
63.     root->right->right->left = createNode(53);
64.     root->right->right->right = createNode(68);
65.     cout<<"\n The Preorder traversal of given binary tree is -\n";
66.     traversePreorder(root);
67.
68.     cout<<"\n The Inorder traversal of given binary tree is -\n";
69.     traverseInorder(root);
70.
71.     cout<<"\n The Postorder traversal of given binary tree is -\n";
72.     traversePostorder(root);
73.     return 0;
74. }
```

Output

Program: Write a program to implement tree traversal techniques in Java.

```
1. class Node {
2.     public int value;
```

```
3.      public Node left, right;
4.
5.      public Node(int element)
6.      {
7.          value = element;
8.          left = right = null;
9.      }
10. }
11.
12. class Tree {
13.     Node root;  /* root of the tree */
14.
15.     Tree() { root = null; }
16.   /*function to print the nodes of given binary in Preorder*/
17.     void traversePreorder(Node node)
18.     {
19.         if (node == null)
20.             return;
21.         System.out.print(node.value + " ");
22.         traversePreorder(node.left);
23.         traversePreorder(node.right);
24.     }
25.   /*function to print the nodes of given binary in Inorder*/
26.     void traverseInorder(Node node)
27.     {
28.         if (node == null)
29.             return;
30.         traverseInorder(node.left);
31.         System.out.print(node.value + " ");
32.         traverseInorder(node.right);
33.     }
```

```
34.  /*function to print the nodes of given binary in Postorder*/
35.  void traversePostorder(Node node)
36.  {
37.     if (node == null)
38.        return;
39.     traversePostorder(node.left);
40.     traversePostorder(node.right);
41.     System.out.print(node.value + " ");
42.  }
43.
44.
45.  void traversePreorder() { traversePreorder(root); }
46.  void traverseInorder() { traverseInorder(root); }
47.  void traversePostorder() { traversePostorder(root); }
48.
49.  public static void main(String args[])
50.  {
51.     Tree pt = new Tree();
52.     pt.root = new Node(36);
53.     pt.root.left = new Node(26);
54.     pt.root.right = new Node(46);
55.     pt.root.left.left = new Node(21);
56.     pt.root.left.right = new Node(31);
57.     pt.root.left.left.left = new Node(11);
58.     pt.root.left.left.right = new Node(24);
59.     pt.root.right.left = new Node(41);
60.     pt.root.right.right = new Node(56);
61.     pt.root.right.right.left = new Node(51);
62.     pt.root.right.right.right = new Node(66);
63.
64.     System.out.println();
```

```
65.         System.out.println("The Preorder traversal of given binary tree is - ");
66.         pt.traversePreorder();
67.         System.out.println("\n");
68.         System.out.println("The Inorder traversal of given binary tree is - ");
69.         pt.traverseInorder();
70.         System.out.println("\n");
71.         System.out.println("The Postorder traversal of given binary tree is - ");
72.         pt.traversePostorder();
73.         System.out.println();
74.     }
75. }
```

Output

After the execution of the above code, the output will be -

Conclusion

In this article, we have discussed the different types of tree traversal techniques: preorder traversal, inorder traversal, and postorder traversal. We have seen these techniques along with algorithm, example, complexity, and implementation in C, C++, C#, and java.

Chapter 7

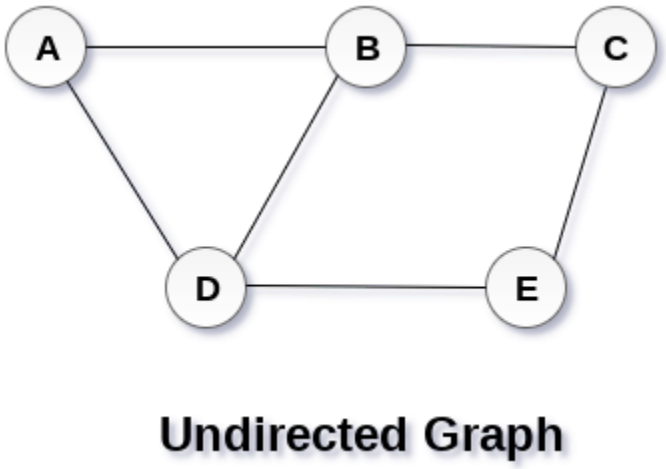
Graph

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Definition

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



Undirected Graph

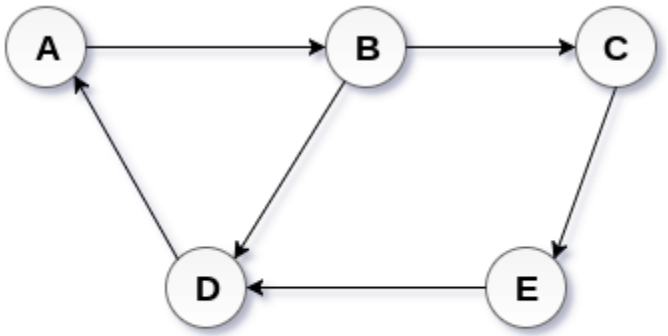
Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges

are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.

A directed graph is shown in the following figure.



Directed Graph

Graph Terminology

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.

Simple Path

If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.

Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Chapter 8: Advanced Sorting and Searching algorithms

8.1. Advanced Sorting

8.1.1. Shell sort

Shell Sort Algorithm

In this article, we will discuss the shell sort algorithm. Shell sort is the generalization of insertion sort, which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

It is a sorting algorithm that is an extended version of insertion sort. Shell sort has improved the average time complexity of insertion sort. As similar to insertion sort, it is a comparison-based and in-place sorting algorithm. Shell sort is efficient for medium-sized data sets.

In insertion sort, at a time, elements can be moved ahead by one position only. To move an element to a far-away position, many movements are required that increase the algorithm's execution time. But shell sort overcomes this drawback of insertion sort. It allows the movement and swapping of far-away elements as well.

This algorithm first sorts the elements that are far away from each other, then it subsequently reduces the gap between them. This gap is called as **interval.** This interval can be calculated by using the **Knuth's** formula given below -

1. hh = h * 3 + 1
2. where, 'h' is the interval having initial value 1.

Now, let's see the algorithm of shell sort.

Algorithm

The simple steps of achieving the shell sort are listed as follows -

1. ShellSort(a, n) // 'a' is the given array, 'n' is the size of array
2. for (interval = n/2; interval > 0; interval /= 2)
3. for (i = interval; i < **n**; i += 1)
4. temp = a[i];
5. for (j = i; j >= interval && a[j - interval] > temp; j -= interval)
6. a[j] = a[j - interval];
7. a[j] = temp;
8. End ShellSort

Working of Shell sort Algorithm

Now, let's see the working of the shell sort Algorithm.

To understand the working of the shell sort algorithm, let's take an unsorted array. It will be easier to understand the shell sort via an example.

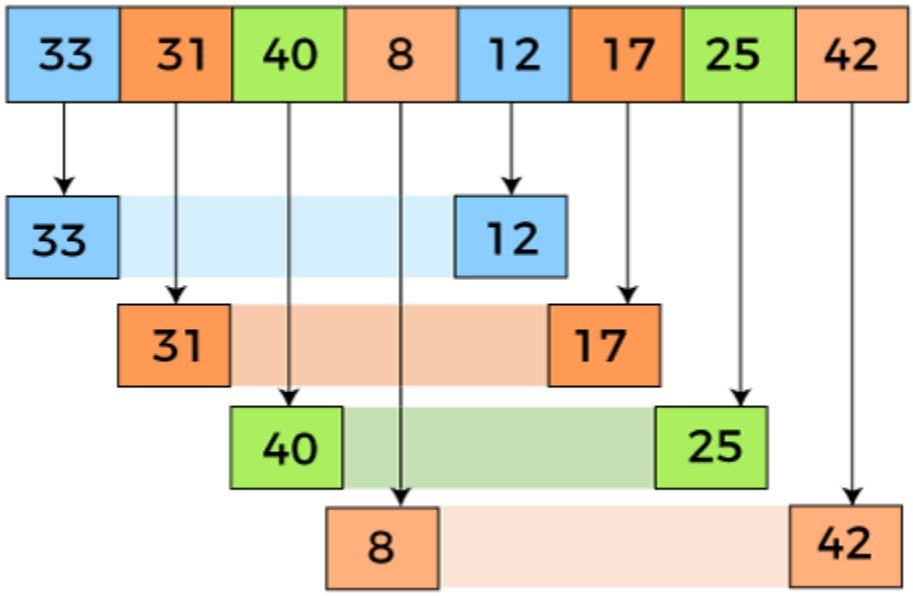
Let the elements of array are -

We will use the original sequence of shell sort, i.e., N/2, N/4,....,1 as the intervals.

In the first loop, n is equal to 8 (size of the array), so the elements are lying at the interval of 4 (n/2 = 4). Elements will be compared and swapped if they are not in order.

Here, in the first loop, the element at the 0^{th} position will be compared with the element at 4^{th} position. If the 0^{th} element is greater, it will be swapped with the element at 4^{th} position. Otherwise, it remains the same. This process will continue for the remaining elements.

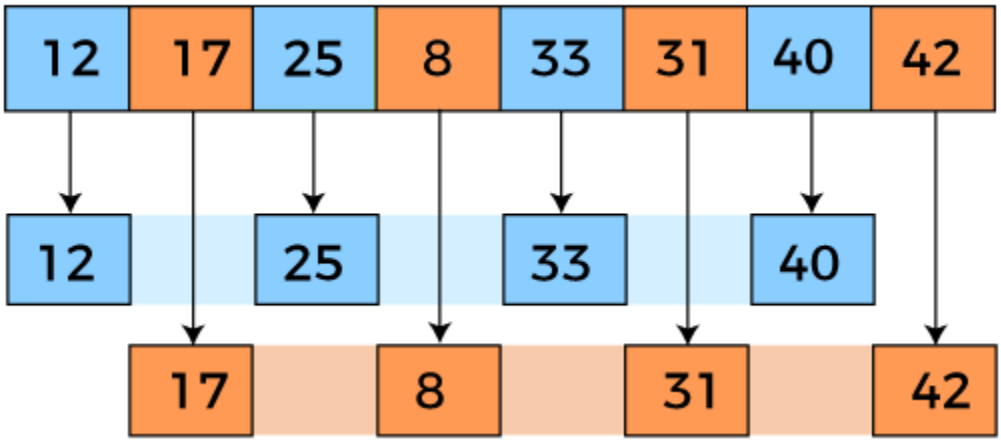
At the interval of 4, the sublists are {33, 12}, {31, 17}, {40, 25}, {8, 42}.



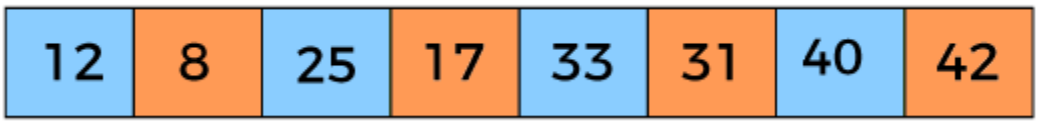
Now, we have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows -

In the second loop, elements are lying at the interval of 2 (n/4 = 2), where n = 8.

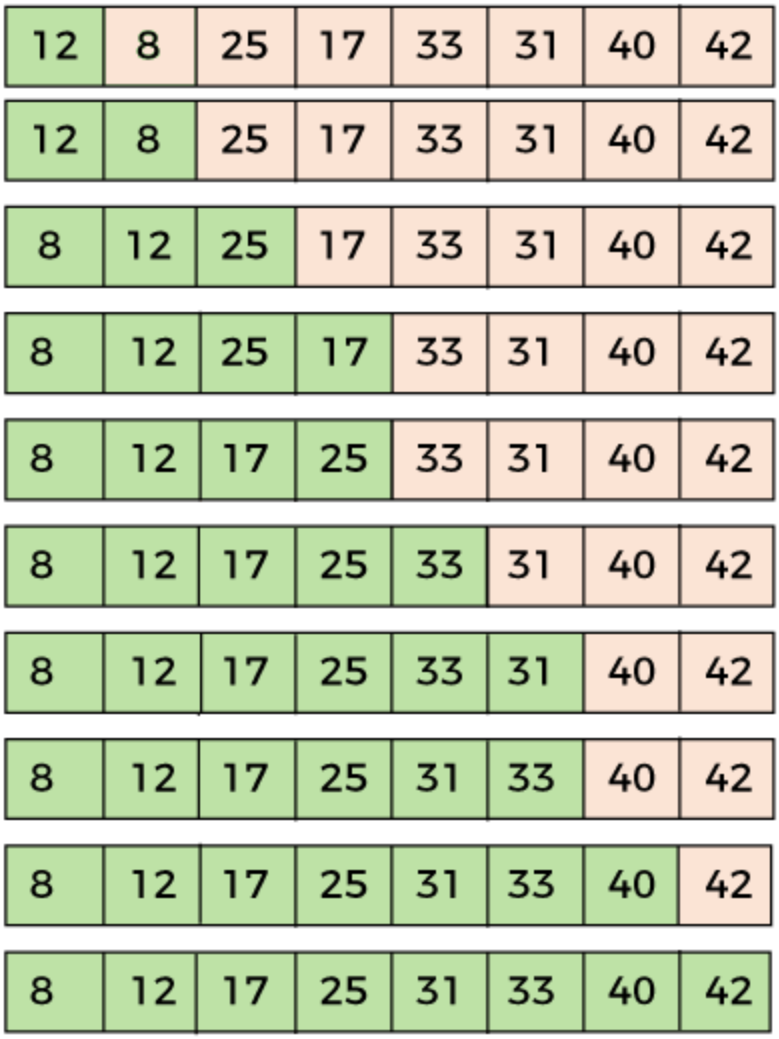
Now, we are taking the interval of **2** to sort the rest of the array. With an interval of 2, two sublists will be generated - {12, 25, 33, 40}, and {17, 8, 31, 42}.



Now, we again have to compare the values in every sub-list. After comparing, we have to swap them if required in the original array. After comparing and swapping, the updated array will look as follows -



In the third loop, elements are lying at the interval of 1 (n/8 = 1), where n = 8. At last, we use the interval of value 1 to sort the rest of the array elements. In this step, shell sort uses insertion sort to sort the array elements.



Now, the array is sorted in ascending order.

Shell sort complexity

Now, let's see the time complexity of Shell sort in the best case, average case, and worst case. We will also see the space complexity of the Shell sort.

1. Time Complexity

Case	Time Complexity

Best Case	O(n*logn)
Average Case	$O(n*log(n)^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e., the array is already sorted. The best-case time complexity of Shell sort is **O(n*logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Shell sort is **O(n*logn).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Shell sort is **$O(n^2)$.**

2. Space Complexity

Space Complexity	O(1)
Stable	NO

- The space complexity of Shell sort is O(1).

Implementation of Shell sort

Now, let's see the programs of Shell sort in different programming languages.

Program: Write a program to implement Shell sort in C language.

```
1. #include <stdio.h>
2. /* function to implement shellSort */
3. int shell(int a[], int n)
4. {
```

```
5.      /* Rearrange the array elements at n/2, n/4, ..., 1 intervals */
6.      for (int interval = n/2; interval > 0; interval /= 2)
7.      {
8.          for (int i = interval; i < n; i += 1)
9.          {
10.             /* store a[i] to the variable temp and make the ith position empty */
11.             int temp = a[i];
12.             int j;
13.             for (j = i; j >= interval && a[j - interval] > temp; j -= interval)
14.                 a[j] = a[j - interval];
15.
16.             // put temp (the original a[i]) in its correct position
17.             a[j] = temp;
18.         }
19.     }
20.     return 0;
21. }
22. void printArr(int a[], int n) /* function to print the array elements */
23. {
24.     int i;
25.     for (i = 0; i < n; i++)
26.         printf("%d ", a[i]);
27. }
28. int main()
29. {
30.     int a[] = { 33, 31, 40, 8, 12, 17, 25, 42 };
31.     int n = sizeof(a) / sizeof(a[0]);
32.     printf("Before sorting array elements are - \n");
33.     printArr(a, n);
34.     shell(a, n);
35.     printf("\nAfter applying shell sort, the array elements are - \n");
36.     printArr(a, n);
37.     return 0;
38. }
```

Output

After the execution of above code, the output will be -

```
Before sorting array elements are -
33 31 40 8 12 17 25 42
After applying shell sort, the array elements are -
8 12 17 25 31 33 40 42
```

Program: Write a program to implement Shell sort in C++.

```
1.  #include <iostream>
2.  using namespace std;
3.  /* function to implement shellSort */
4.  int shell(int a[], int n)
5.  {
6.      /* Rearrange the array elements at n/2, n/4, ..., 1 intervals */
7.      for (int interval = n/2; interval > 0; interval /= 2)
8.      {
9.          for (int i = interval; i < n; i += 1)
10.         {
11.             /* store a[i] to the variable temp and make the ith position empty */
12.             int temp = a[i];
13.             int j;
14.             for (j = i; j >= interval && a[j - interval] > temp; j -= interval)
15.                 a[j] = a[j - interval];
16.
17.             // put temp (the original a[i]) in its correct position
18.             a[j] = temp;
19.         }
20.     }
21.     return 0;
22. }
23. void printArr(int a[], int n) /* function to print the array elements */
24. {
25.     int i;
```

```
    for (i = 0; i < n; i++)
        cout<<a[i]<<" ";
}
int main()
{
    int a[] = { 32, 30, 39, 7, 11, 16, 24, 41 };
    int n = sizeof(a) / sizeof(a[0]);
    cout<<"Before sorting array elements are - \n";
    printArr(a, n);
    shell(a, n);
    cout<<"\nAfter applying shell sort, the array elements are - \n";
    printArr(a, n);
    return 0;
}
```

Output

After the execution of the above code, the output will be -

```
Before sorting array elements are -
32 30 39 7 11 16 24 41
After applying shell sort, the array elements are -
7 11 16 24 30 32 39 41
```

Program: Write a program to implement Shell sort in C#.

```
using System;
class ShellSort {
/* function to implement shellSort */
static void shell(int[] a, int n)
{
    /* Rearrange the array elements at n/2, n/4, ..., 1 intervals */
    for (int interval = n/2; interval > 0; interval /= 2)
    {
        for (int i = interval; i < n; i += 1)
        {
```

```
            /* store a[i] to the variable temp and make the ith position empty */
            int temp = a[i];
            int j;
            for (j = i; j >= interval && a[j - interval] > temp; j -= interval)
                a[j] = a[j - interval];

            /* put temp (the original a[i]) in its correct position */
            a[j] = temp;
        }
    }
}
static void printArr(int[] a, int n) /* function to print the array elements */
{
    int i;
    for (i = 0; i < n; i++)
        Console.Write(a[i] + " ");
}
static void Main()
{
    int[] a = { 31, 29, 38, 6, 10, 15, 23, 40 };
    int n = a.Length;
    Console.Write("Before sorting array elements are - \n");
    printArr(a, n);
    shell(a, n);
    Console.Write("\nAfter applying shell sort, the array elements are - \n");
    printArr(a, n);
}
}
```

Output

After the execution of the above code, the output will be -

```
Before sorting array elements are -
31 29 38 6 10 15 23 40
After applying shell sort, the array elements are -
6 10 15 23 29 31 38 40
```

Program: Write a program to implement Shell sort in Java.

```
1.  class ShellSort {
2.  /* function to implement shellSort */
3.  static void shell(int a[], int n)
4.  {
5.     /* Rearrange the array elements at n/2, n/4, ..., 1 intervals */
6.     for (int interval = n/2; interval > 0; interval /= 2)
7.     {
8.        for (int i = interval; i < n; i += 1)
9.        {
10.          /* store a[i] to the variable temp and make
11.
12. the ith position empty */
13.          int temp = a[i];
14.          int j;
15.          for (j = i; j >= interval && a[j - interval] >
16. temp; j -= interval)
17.             a[j] = a[j - interval];
18.
19.          /* put temp (the original a[i]) in its correct
20. position */
21.          a[j] = temp;
22.       }
23.    }
24. }
25. static void printArr(int a[], int n) /* function to print the array elements */
26. {
27.    int i;
28.    for (i = 0; i < n; i++)
```

```
29.         System.out.print(a[i] + " ");
30.  }
31.  public static void main(String args[])
32.  {
33.     int a[] = { 30, 28, 37, 5, 9, 14, 22, 39 };
34.     int n = a.length;
35.     System.out.print("Before sorting array elements are - \n");
36.     printArr(a, n);
37.     shell(a, n);
38.     System.out.print("\nAfter applying shell sort, the array elements are - \n");
39.     printArr(a, n);
40.  }
41.  }
```

Output

After the execution of the above code, the output will be -

```
D:\JTP>javac ShellSort.java

D:\JTP>java  ShellSort
Before sorting array elements are -
30 28 37 5 9 14 22 39
After applying shell sort, the array elements are -
5 9 14 22 28 30 37 39
D:\JTP>
```

So, that's all about the article. Hope the article will be helpful and informative to you.

Quick Sort Algorithm

In this article, we will discuss the Quicksort Algorithm. The working procedure of Quicksort is also simple. This article will be very helpful and interesting to students as they might face quicksort as a question in their examinations. So, it is important to discuss the topic.

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer

approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

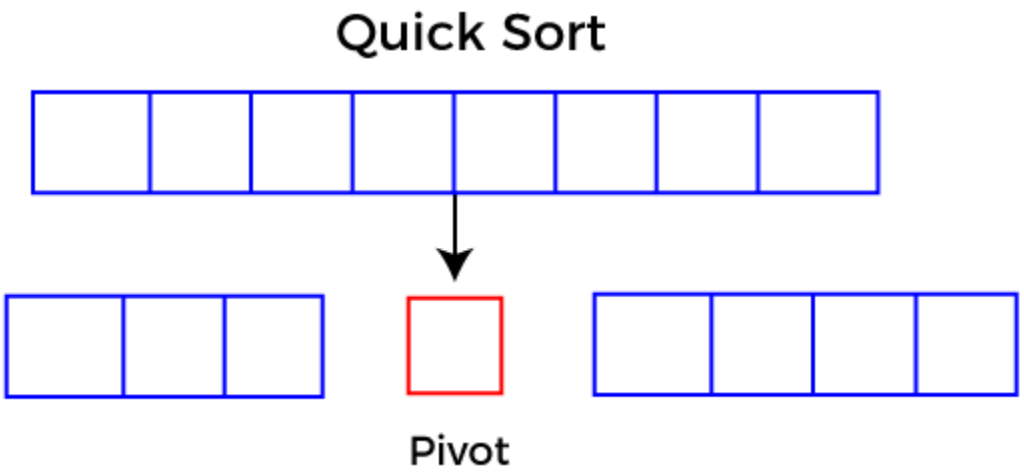
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.



Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Algorithm

Algorithm:

```
1. QUICKSORT (array A, start, end)
2. {
3.  1 if (start < end)
4.  2 {
5. 3 p = partition(A, start, end)
6. 4 QUICKSORT (A, start, p - 1)
7. 5 QUICKSORT (A, p + 1, end)
8. 6 }
9. }
```

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

```
1. PARTITION (array A, start, end)
2. {
3.  1 pivot ? A[end]
4.  2 i ? start-1
5.  3 for j ? start to end -1 {
6.  4 do if (A[j] < pivot) {
7.  5 then i ? i + 1
8.  6 swap A[i] with A[j]
9.  7  }}
```

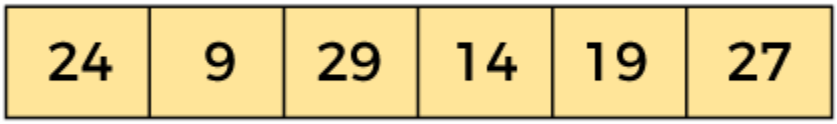
10. 8 swap A[i+1] with A[end]
11. 9 **return** i+1
12. }

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

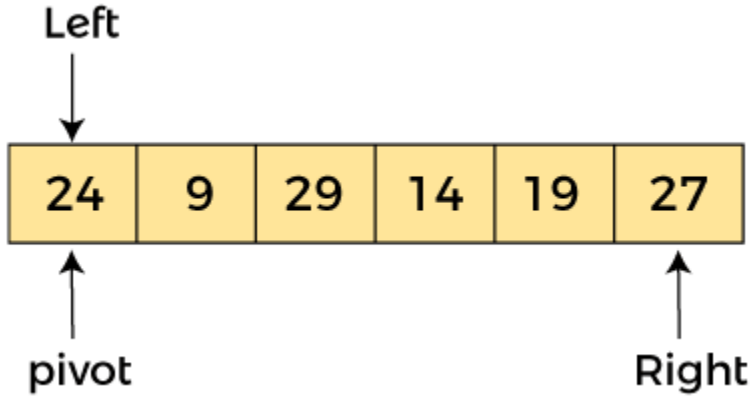
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

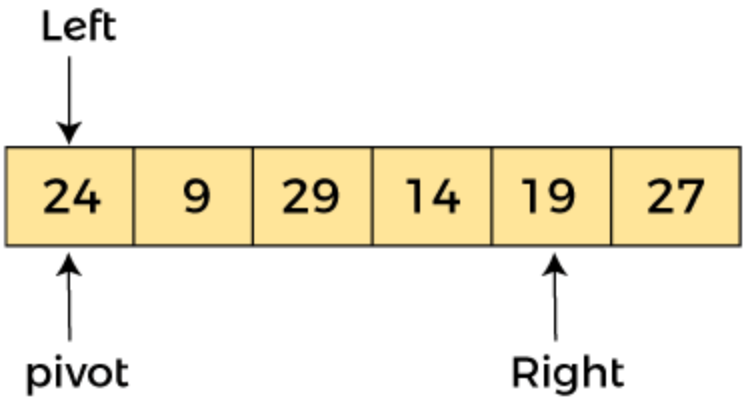


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

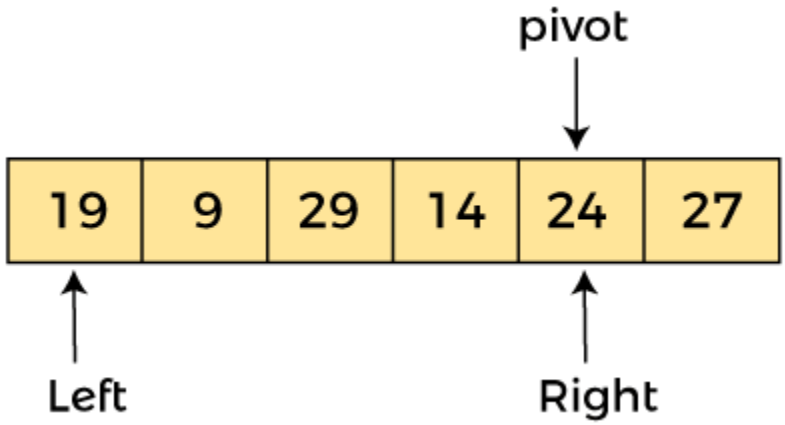


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



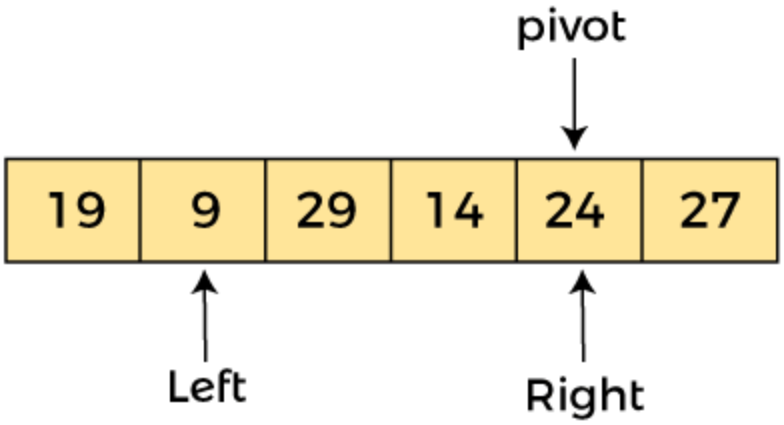
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

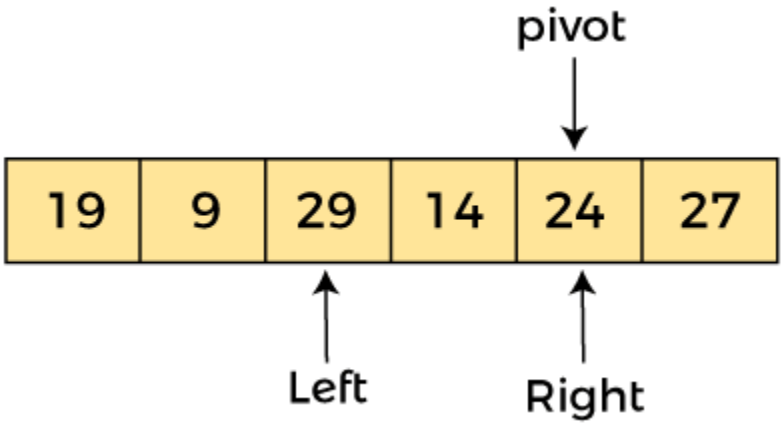


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

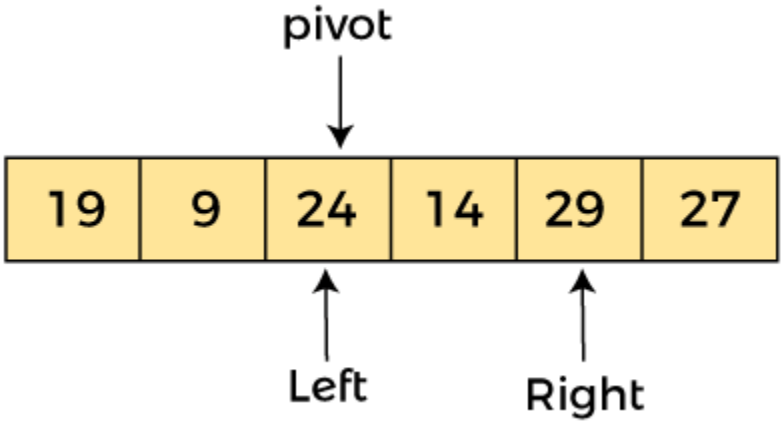
As a[pivot] > a[left], so algorithm moves one position to right as -



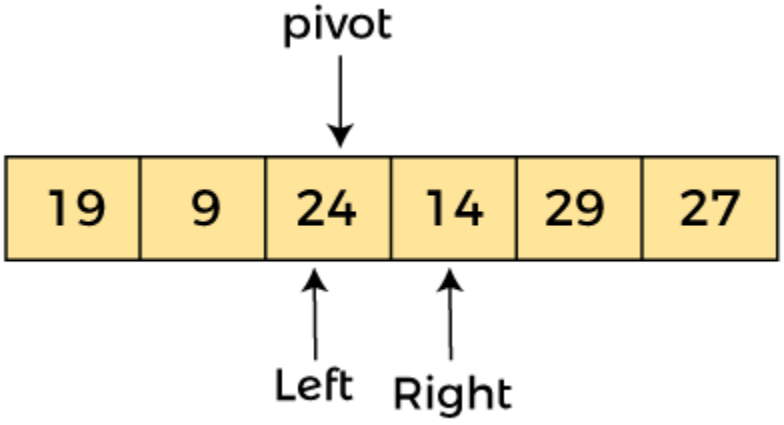
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



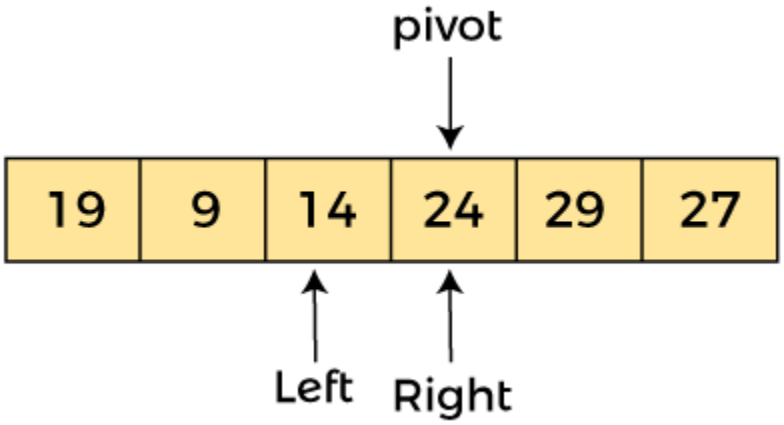
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



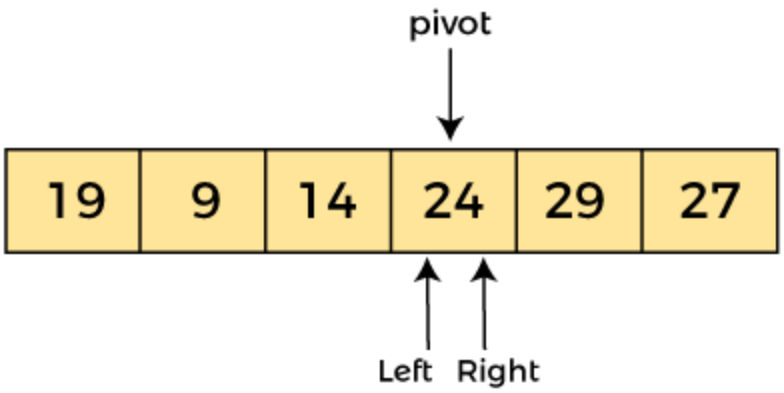
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



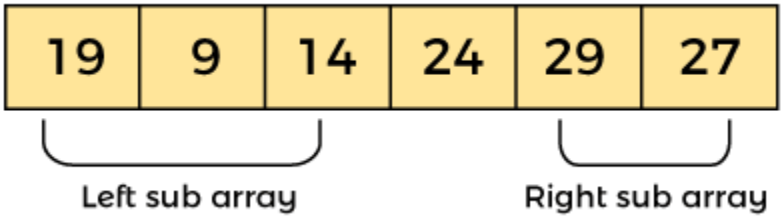
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



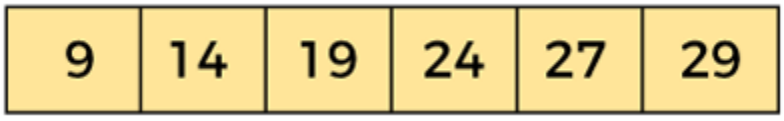
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



Quicksort complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.
- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

2. Space Complexity

Space Complexity	O(n*logn)

Stable	NO

- The space complexity of quicksort is O(n*logn).

Implementation of quicksort

Now, let's see the programs of quicksort in different programming languages.

Program: Write a program to implement quicksort in C language.

```
#include <stdio.h>
/* function that consider last element as pivot,
place the pivot at its exact position, and place
smaller elements to left of pivot and greater
elements to right of pivot.  */
int partition (int a[], int start, int end)
{
    int pivot = a[end]; // pivot element
    int i = (start - 1);

    for (int j = start; j <= end - 1; j++)
    {
        // If current element is smaller than the pivot
        if (a[j] < pivot)
        {
            i++; // increment index of smaller element
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    int t = a[i+1];
    a[i+1] = a[end];
    a[end] = t;
```

```
25.     return (i + 1);
26. }
27.
28. /* function to implement quick sort */
29. void quick(int a[], int start, int end) /* a[] = array to be sorted, start = Starting index, end = Ending index
    */
30. {
31.     if (start < end)
32.     {
33.         int p = partition(a, start, end); //p is the partitioning index
34.         quick(a, start, p - 1);
35.         quick(a, p + 1, end);
36.     }
37. }
38.
39. /* function to print an array */
40. void printArr(int a[], int n)
41. {
42.     int i;
43.     for (i = 0; i < n; i++)
44.         printf("%d ", a[i]);
45. }
46. int main()
47. {
48.     int a[] = { 24, 9, 29, 14, 19, 27 };
49.     int n = sizeof(a) / sizeof(a[0]);
50.     printf("Before sorting array elements are - \n");
51.     printArr(a, n);
52.     quick(a, 0, n - 1);
53.     printf("\nAfter sorting array elements are - \n");
54.     printArr(a, n);
55.
56.     return 0;
57. }
```

Output:

```
Before sorting array elements are -
24 9 29 14 19 27
After sorting array elements are -
9 14 19 24 27 29
```

Program: Write a program to implement quick sort in C++ language.

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  /* function that consider last element as pivot,
6.  place the pivot at its exact position, and place
7.  smaller elements to left of pivot and greater
8.  elements to right of pivot.  */
9.  int partition (int a[], int start, int end)
10. {
11.     int pivot = a[end]; // pivot element
12.     int i = (start - 1);
13.
14.     for (int j = start; j <= end - 1; j++)
15.     {
16.         // If current element is smaller than the pivot
17.         if (a[j] < pivot)
18.         {
19.             i++; // increment index of smaller element
20.             int t = a[i];
21.             a[i] = a[j];
22.             a[j] = t;
23.         }
24.     }
25.     int t = a[i+1];
26.     a[i+1] = a[end];
```

```
27.     a[end] = t;
28.     return (i + 1);
29. }
30.
31. /* function to implement quick sort */
32. void quick(int a[], int start, int end) /* a[] = array to be sorted, start = Starting index, end = Ending index
    */
33. {
34.     if (start < end)
35.     {
36.         int p = partition(a, start, end);  //p is the partitioning index
37.         quick(a, start, p - 1);
38.         quick(a, p + 1, end);
39.     }
40. }
41.
42. /* function to print an array */
43. void printArr(int a[], int n)
44. {
45.     int i;
46.     for (i = 0; i < n; i++)
47.         cout<<a[i]<< " ";
48. }
49. int main()
50. {
51.     int a[] = { 23, 8, 28, 13, 18, 26 };
52.     int n = sizeof(a) / sizeof(a[0]);
53.     cout<<"Before sorting array elements are - \n";
54.     printArr(a, n);
55.     quick(a, 0, n - 1);
56.     cout<<"\nAfter sorting array elements are - \n";
57.     printArr(a, n);
58.
59.     return 0;
```

60. }

Output:

```
Before sorting array elements are -
23 8 28 13 18 26
After sorting array elements are -
8 13 18 23 26 28
```

Program: Write a program to implement quicksort in python.

```
#function that consider last element as pivot,
#place the pivot at its exact position, and place
#smaller elements to left of pivot and greater
#elements to right of pivot.

def partition (a, start, end):
    i = (start - 1)
    pivot = a[end] # pivot element

    for j in range(start, end):
        # If current element is smaller than or equal to the pivot
        if (a[j] <= pivot):
            i = i + 1
            a[i], a[j] = a[j], a[i]

    a[i+1], a[end] = a[end], a[i+1]

    return (i + 1)

# function to implement quick sort
def quick(a, start, end): # a[] = array to be sorted, start = Starting index, end = Ending index
    if (start < end):
        p = partition(a, start, end) # p is partitioning index
        quick(a, start, p - 1)
        quick(a, p + 1, end)
```

```
26.
27.
28. def printArr(a): # function to print the array
29.     for i in range(len(a)):
30.         print (a[i], end = " ")
31.
32.
33. a = [68, 13, 1, 49, 58]
34. print("Before sorting array elements are - ")
35. printArr(a)
36. quick(a, 0, len(a)-1)
37. print("\nAfter sorting array elements are - ")
38. printArr(a)
```

Output:

```
Before sorting array elements are -
68 13 1 49 58
After sorting array elements are -
1 13 49 58 68
```

Program: Write a program to implement quicksort in Java.

```
1.  public class Quick
2.  {
3.      /* function that consider last element as pivot,
4.  place the pivot at its exact position, and place
5.  smaller elements to left of pivot and greater
6.  elements to right of pivot.  */
7.  int partition (int a[], int start, int end)
8.  {
9.      int pivot = a[end]; // pivot element
10.     int i = (start - 1);
11.
12.     for (int j = start; j <= end - 1; j++)
```

```
13.     {
14.         // If current element is smaller than the pivot
15.         if (a[j] < pivot)
16.         {
17.             i++; // increment index of smaller element
18.             int t = a[i];
19.             a[i] = a[j];
20.             a[j] = t;
21.         }
22.     }
23.     int t = a[i+1];
24.     a[i+1] = a[end];
25.     a[end] = t;
26.     return (i + 1);
27. }
28.
29. /* function to implement quick sort */
30. void quick(int a[], int start, int end) /* a[] = array to be sorted, start = Starting index, end = Ending index
    */
31. {
32.     if (start < end)
33.     {
34.         int p = partition(a, start, end);  //p is partitioning index
35.         quick(a, start, p - 1);
36.         quick(a, p + 1, end);
37.     }
38. }
39.
40. /* function to print an array */
41. void printArr(int a[], int n)
42. {
43.     int i;
44.     for (i = 0; i < n; i++)
45.         System.out.print(a[i] + " ");
```

```
46. }
47.     public static void main(String[] args) {
48.     int a[] = { 13, 18, 27, 2, 19, 25 };
49.     int n = a.length;
50.     System.out.println("\nBefore sorting array elements are - ");
51.     Quick q1 = new Quick();
52.     q1.printArr(a, n);
53.     q1.quick(a, 0, n - 1);
54.     System.out.println("\nAfter sorting array elements are - ");
55.     q1.printArr(a, n);
56.     System.out.println();
57.     }
58. }
```

Output

After the execution of above code, the output will be -

```
D:\JTP>javac Quick.java

D:\JTP>java Quick

Before sorting array elements are -
13 18 27 2 19 25
After sorting array elements are -
2 13 18 19 25 27
```

Program: Write a program to implement quick sort in php.

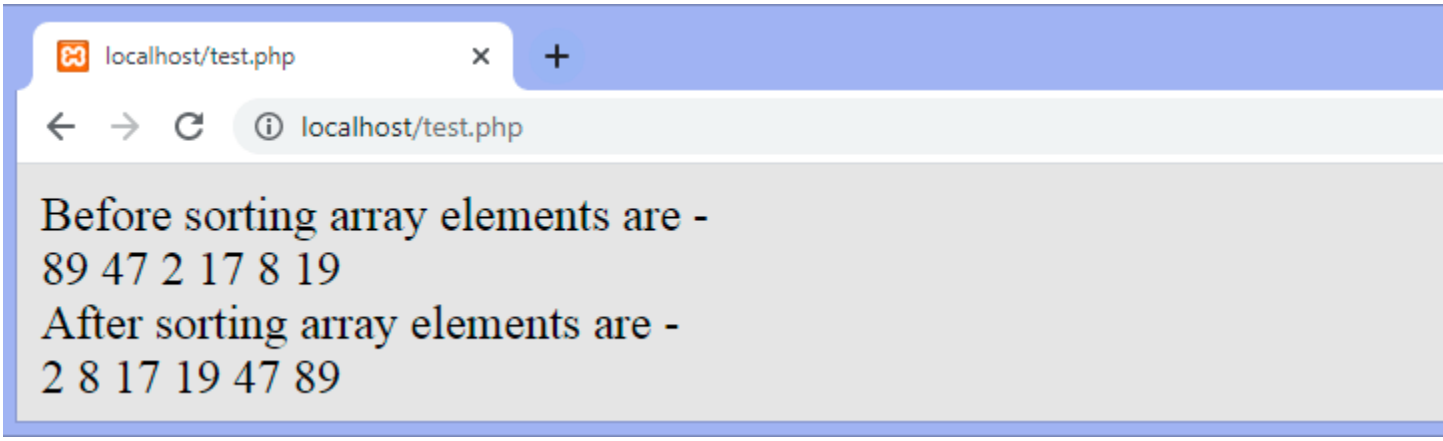
```
1. <?php
2.     /* function that consider last element as pivot,
3. place the pivot at its exact position, and place
4. smaller elements to left of pivot and greater
5. elements to right of pivot.  */
6. function partition (&$a, $start, $end)
7. {
8.     $pivot = $a[$end]; // pivot element
9.     $i = ($start - 1);
```

```
10.
11.     for ($j = $start; $j <= $end - 1; $j++)
12.     {
13.         // If current element is smaller than the pivot
14.         if ($a[$j] < $pivot)
15.         {
16.             $i++; // increment index of smaller element
17.             $t = $a[$i];
18.             $a[$i] = $a[$j];
19.             $a[$j] = $t;
20.         }
21.     }
22.     $t = $a[$i+1];
23.     $a[$i+1] = $a[$end];
24.     $a[$end] = $t;
25.     return ($i + 1);
26. }
27.
28. /* function to implement quick sort */
29. function quick(&$a, $start, $end) /* a[] = array to be sorted, start = Starting index, end = Ending index */

30. {
31.     if ($start < $end)
32.     {
33.         $p = partition($a, $start, $end); //p is partitioning index
34.         quick($a, $start, $p - 1);
35.         quick($a, $p + 1, $end);
36.     }
37. }
38.
39.     function printArray($a, $n)
40.     {
41.     for($i = 0; $i < $n; $i++)
42.     {
```

```
43.         print_r($a[$i]);
44.         echo " ";
45.     }
46.     }
47.     $a = array( 89, 47, 2, 17, 8, 19 );
48.     $n = count($a);
49.     echo "Before sorting array elements are - <br>";
50.     printArray($a, $n);
51.     quick($a, 0, $n - 1);
52.     echo "<br> After sorting array elements are - <br>";
53.     printArray($a, $n);
54.
55. ?>
```

Output

After the execution of above code, the output will be -



So, that's all about the article. Hope the article will be helpful and informative to you.

This article was not only limited to the algorithm. Along with the algorithm, we have also discussed the quick sort complexity, working, and implementation in different programming languages.

8.1.3. Heap Sort

Heap Sort Algorithm

In this article, we will discuss the Heapsort Algorithm. Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

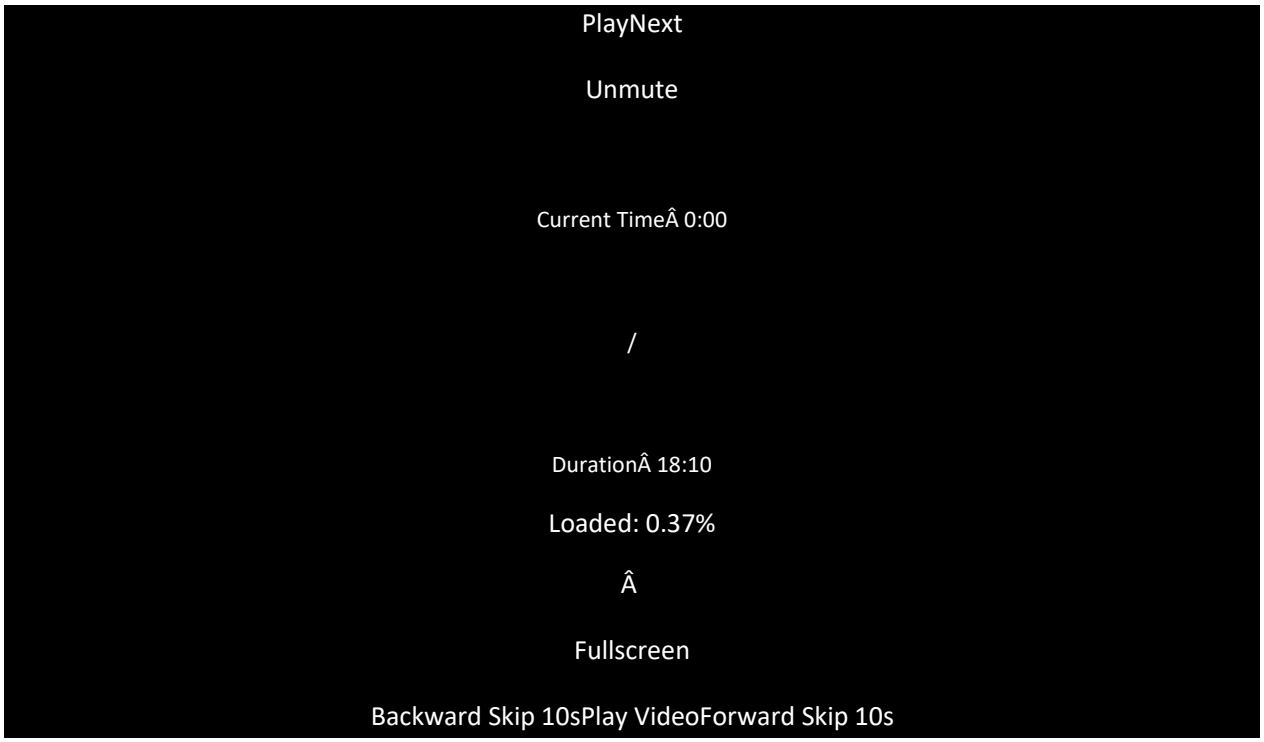
Heap sort basically recursively performs two main operations -

- Build a heap H, using the elements of array.
- Repeatedly delete the root element of the heap formed in 1st phase.

Before knowing more about the heap sort, let's first see a brief description of **Heap.**

What is a heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.



What is heap sort?

Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

Heapsort is the in-place sorting algorithm.

Now, let's see the algorithm of heap sort.

Algorithm

```
1. HeapSort(arr)
2. BuildMaxHeap(arr)
3. for i = length(arr) to 2
4.    swap arr[1] with arr[i]
5.       heap_size[arr] = heap_size[arr] ? 1
6.       MaxHeapify(arr,1)
7. End
```

BuildMaxHeap(arr)

```
1. BuildMaxHeap(arr)
2.    heap_size(arr) = length(arr)
3.    for i = length(arr)/2 to 1
4. MaxHeapify(arr,i)
5. End
```

MaxHeapify(arr,i)

```
1. MaxHeapify(arr,i)
2. L = left(i)
3. R = right(i)
4. if L ? heap_size[arr] and arr[L] > arr[i]
5. largest = L
6. else
7. largest = i
```

8. if R ? heap_size[arr] and arr[R] > arr[largest]
9. largest = R
10. if largest != i
11. swap arr[i] with arr[largest]
12. MaxHeapify(arr,largest)
13. End

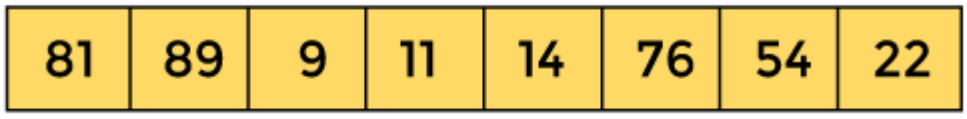
Working of Heap sort Algorithm

Now, let's see the working of the Heapsort Algorithm.

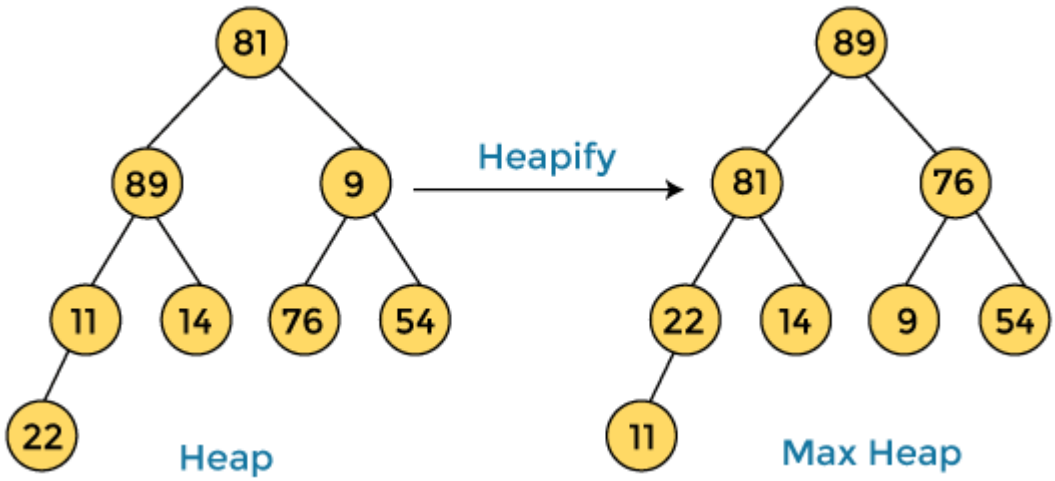
In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

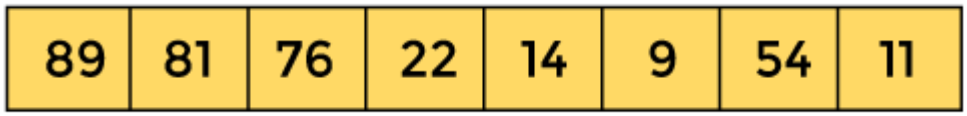
Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.



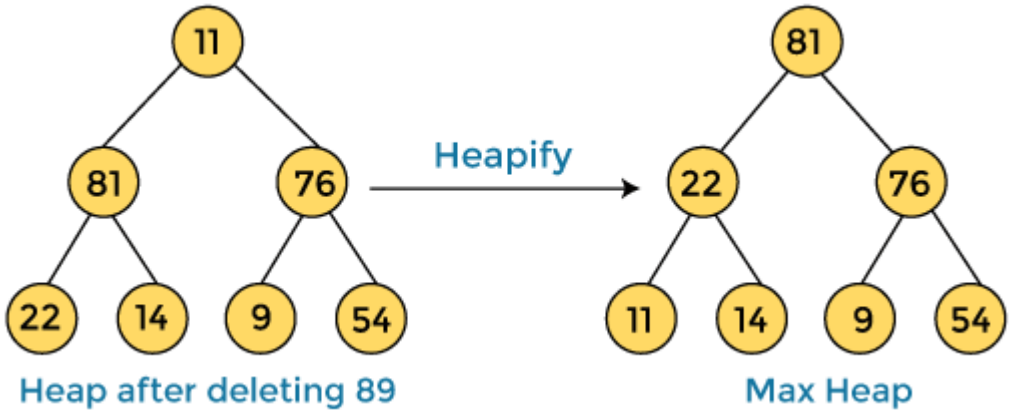
First, we have to construct a heap from the given array and convert it into max heap.



After converting the given heap into max heap, the array elements are -

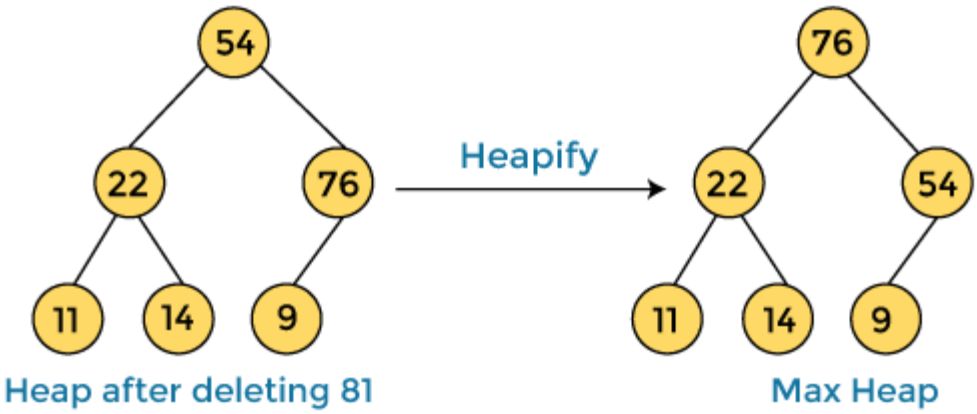


Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



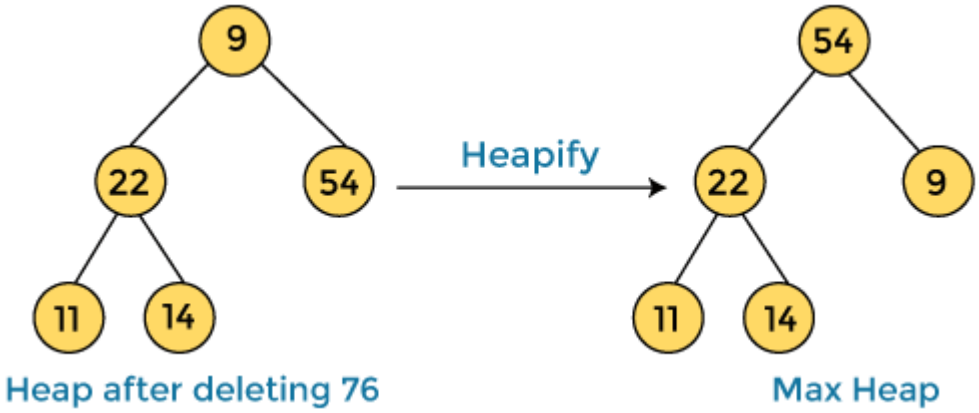
After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -

In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



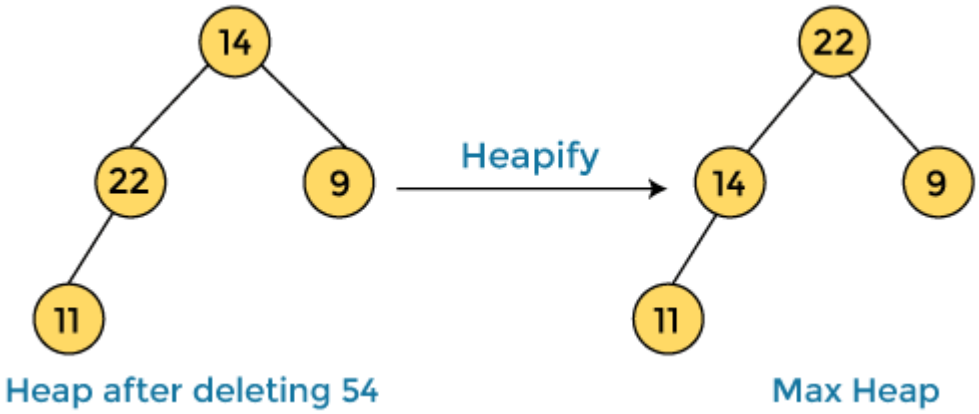
After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

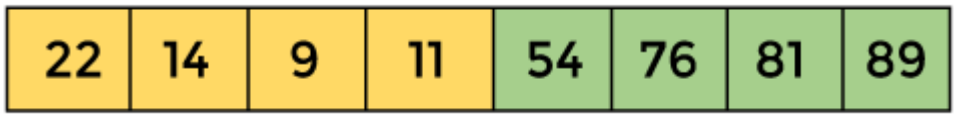


After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

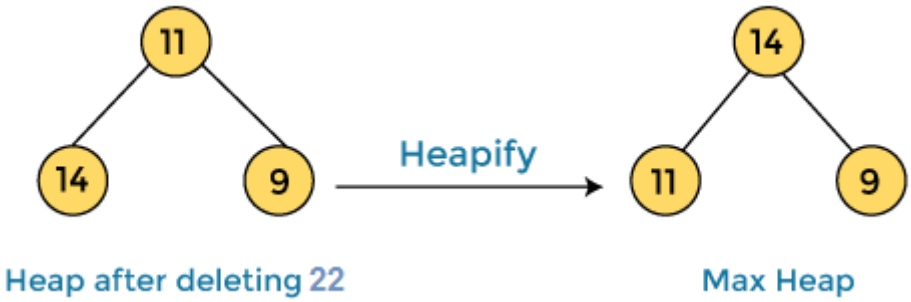
In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



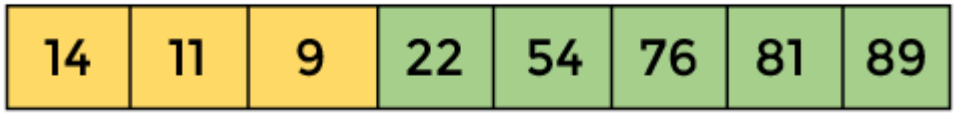
After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -



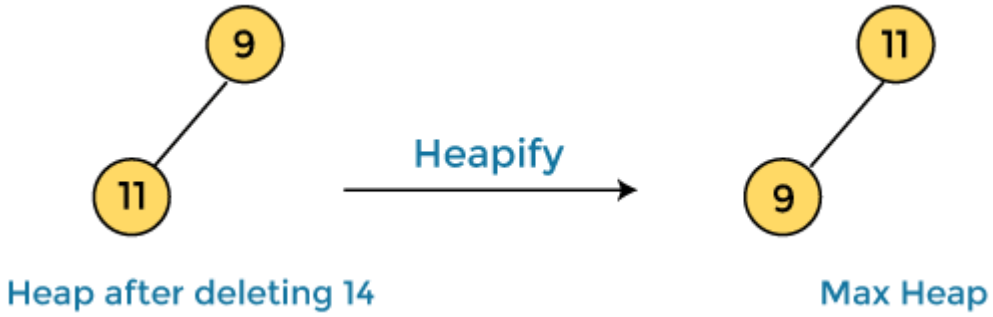
In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



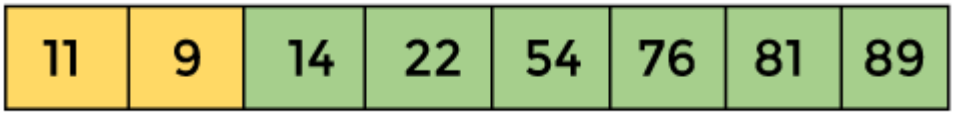
After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -



In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

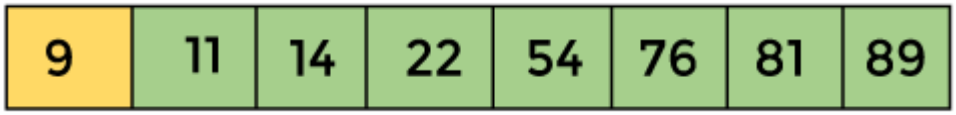


After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -



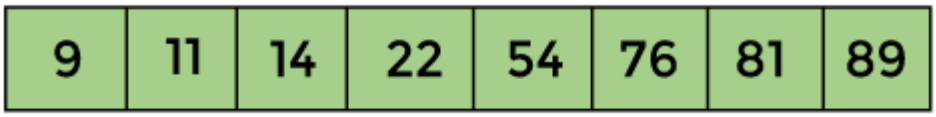
In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element **11** with **9,** the elements of array are -



Now, heap has only one element left. After deleting it, heap will be empty.

After completion of sorting, the array elements are -



Now, the array is completely sorted.

Heap sort complexity

Now, let's see the time complexity of Heap sort in the best case, average case, and worst case. We will also see the space complexity of Heapsort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n logn)
Average Case	O(n log n)
Worst Case	O(n log n)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

The time complexity of heap sort is **O(n logn)** in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is **logn.**

2. Space Complexity

Space Complexity	O(1)
Stable	N0

- The space complexity of Heap sort is O(1).

Implementation of Heapsort

Now, let's see the programs of Heap sort in different programming languages.

Program: Write a program to implement heap sort in C language.

```
#include <stdio.h>
/* function to heapify a subtree. Here 'i' is the
index of root node in array a[], and 'n' is the size of heap. */
void heapify(int a[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // left child
    int right = 2 * i + 2; // right child
    // If left child is larger than root
    if (left < n && a[left] > a[largest])
        largest = left;
    // If right child is larger than root
    if (right < n && a[right] > a[largest])
        largest = right;
    // If root is not largest
    if (largest != i) {
        // swap a[i] with a[largest]
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;

        heapify(a, n, largest);
    }
}
/*Function to implement the heap sort*/
void heapSort(int a[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(a, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        /* Move current root element to end*/
```

```
33.         // swap a[0] with a[i]
34.         int temp = a[0];
35.         a[0] = a[i];
36.         a[i] = temp;
37.
38.         heapify(a, i, 0);
39.     }
40. }
41. /* function to print the array elements */
42. void printArr(int arr[], int n)
43. {
44.     for (int i = 0; i < n; ++i)
45.     {
46.         printf("%d", arr[i]);
47.         printf(" ");
48.     }
49.
50. }
51. int main()
52. {
53.     int a[] = {48, 10, 23, 43, 28, 26, 1};
54.     int n = sizeof(a) / sizeof(a[0]);
55.     printf("Before sorting array elements are - \n");
56.     printArr(a, n);
57.     heapSort(a, n);
58.     printf("\nAfter sorting array elements are - \n");
59.     printArr(a, n);
60.     return 0;
61. }
```

Output

```
Before sorting array elements are -
48 10 23 43 28 26 1
After sorting array elements are -
1 10 23 26 28 43 48
```

Program: Write a program to implement heap sort in C++.

```
1.  #include <iostream>
2.  using namespace std;
3.  /* function to heapify a subtree. Here 'i' is the
4.  index of root node in array a[], and 'n' is the size of heap. */
5.  void heapify(int a[], int n, int i)
6.  {
7.      int largest = i; // Initialize largest as root
8.      int left = 2 * i + 1; // left child
9.      int right = 2 * i + 2; // right child
10.     // If left child is larger than root
11.     if (left < n && a[left] > a[largest])
12.         largest = left;
13.     // If right child is larger than root
14.     if (right < n && a[right] > a[largest])
15.         largest = right;
16.     // If root is not largest
17.     if (largest != i) {
18.         // swap a[i] with a[largest]
19.         int temp = a[i];
20.         a[i] = a[largest];
21.         a[largest] = temp;
22.
23.         heapify(a, n, largest);
24.     }
25. }
26. /*Function to implement the heap sort*/
27. void heapSort(int a[], int n)
28. {
29.
30.     for (int i = n / 2 - 1; i >= 0; i--)
31.         heapify(a, n, i);
32.     // One by one extract an element from heap
```

```
33.     for (int i = n - 1; i >= 0; i--) {
34.         /* Move current root element to end*/
35.         // swap a[0] with a[i]
36.         int temp = a[0];
37.         a[0] = a[i];
38.         a[i] = temp;
39.
40.         heapify(a, i, 0);
41.     }
42. }
43. /* function to print the array elements */
44. void printArr(int a[], int n)
45. {
46.     for (int i = 0; i < n; ++i)
47.     {
48.         cout<<a[i]<<" ";
49.     }
50.
51. }
52. int main()
53. {
54.     int a[] = {47, 9, 22, 42, 27, 25, 0};
55.     int n = sizeof(a) / sizeof(a[0]);
56.     cout<<"Before sorting array elements are - \n";
57.     printArr(a, n);
58.     heapSort(a, n);
59.     cout<<"\nAfter sorting array elements are - \n";
60.     printArr(a, n);
61.     return 0;
62. }
```

Output

```
Before sorting array elements are -
47 9 22 42 27 25 0
After sorting array elements are -
0 9 22 25 27 42 47
```

Program: Write a program to implement heap sort in C#.

```
1. using System;
2. class HeapSort {
3. /* function to heapify a subtree. Here 'i' is the
4. index of root node in array a[], and 'n' is the size of heap. */
5. static void heapify(int[] a, int n, int i)
6. {
7.     int largest = i; // Initialize largest as root
8.     int left = 2 * i + 1; // left child
9.     int right = 2 * i + 2; // right child
10.    // If left child is larger than root
11.    if (left < n && a[left] > a[largest])
12.        largest = left;
13.    // If right child is larger than root
14.    if (right < n && a[right] > a[largest])
15.        largest = right;
16.    // If root is not largest
17.    if (largest != i) {
18.        // swap a[i] with a[largest]
19.        int temp = a[i];
20.        a[i] = a[largest];
21.        a[largest] = temp;
22.
23.        heapify(a, n, largest);
24.    }
25. }
26. /*Function to implement the heap sort*/
27. static void heapSort(int[] a, int n)
28. {
29.    for (int i = n / 2 - 1; i >= 0; i--)
```

```
30.         heapify(a, n, i);
31.
32.     // One by one extract an element from heap
33.     for (int i = n - 1; i >= 0; i--) {
34.         /* Move current root element to end*/
35.         // swap a[0] with a[i]
36.         int temp = a[0];
37.         a[0] = a[i];
38.         a[i] = temp;
39.
40.         heapify(a, i, 0);
41.     }
42. }
43. /* function to print the array elements */
44. static void printArr(int[] a, int n)
45. {
46.     for (int i = 0; i < n; ++i)
47.         Console.Write(a[i] + " ");
48. }
49. static void Main()
50. {
51.     int[] a = {46, 8, 21, 41, 26, 24, -1};
52.     int n = a.Length;
53.     Console.Write("Before sorting array elements are - \n");
54.     printArr(a, n);
55.     heapSort(a, n);
56.     Console.Write("\nAfter sorting array elements are - \n");
57.     printArr(a, n);
58. }
59. }
```

Output

```
Before sorting array elements are -
46 8 21 41 26 24 -1
After sorting array elements are -
-1 8 21 24 26 41 46
```

Program: Write a program to implement heap sort in Java.

```
1. class HeapSort
2. {
3. /* function to heapify a subtree. Here 'i' is the
4. index of root node in array a[], and 'n' is the size of heap. */
5. static void heapify(int a[], int n, int i)
6. {
7.     int largest = i; // Initialize largest as root
8.     int left = 2 * i + 1; // left child
9.     int right = 2 * i + 2; // right child
10.    // If left child is larger than root
11.    if (left < n && a[left] > a[largest])
12.        largest = left;
13.    // If right child is larger than root
14.    if (right < n && a[right] > a[largest])
15.        largest = right;
16.    // If root is not largest
17.    if (largest != i) {
18.        // swap a[i] with a[largest]
19.        int temp = a[i];
20.        a[i] = a[largest];
21.        a[largest] = temp;
22.
23.        heapify(a, n, largest);
24.    }
25. }
26. /*Function to implement the heap sort*/
27. static void heapSort(int a[], int n)
28. {
29.    for (int i = n / 2 - 1; i >= 0; i--)
```

```
30.         heapify(a, n, i);
31.
32.     // One by one extract an element from heap
33.     for (int i = n - 1; i >= 0; i--) {
34.         /* Move current root element to end*/
35.         // swap a[0] with a[i]
36.         int temp = a[0];
37.         a[0] = a[i];
38.         a[i] = temp;
39.
40.         heapify(a, i, 0);
41.     }
42. }
43. /* function to print the array elements */
44. static void printArr(int a[], int n)
45. {
46.     for (int i = 0; i < n; ++i)
47.         System.out.print(a[i] + " ");
48. }
49. public static void main(String args[])
50. {
51.     int a[] = {45, 7, 20, 40, 25, 23, -2};
52.     int n = a.length;
53.     System.out.print("Before sorting array elements are - \n");
54.     printArr(a, n);
55.     heapSort(a, n);
56.     System.out.print("\nAfter sorting array elements are - \n");
57.     printArr(a, n);
58. }
59. }
```

Output

```
D:\JTP>javac HeapSort.java

D:\JTP>java  HeapSort
Before sorting array elements are -
45 7 20 40 25 23 -2
After sorting array elements are -
-2 7 20 23 25 40 45
D:\JTP>
```

So, that's all about the article. Hope the article will be helpful and informative to you.

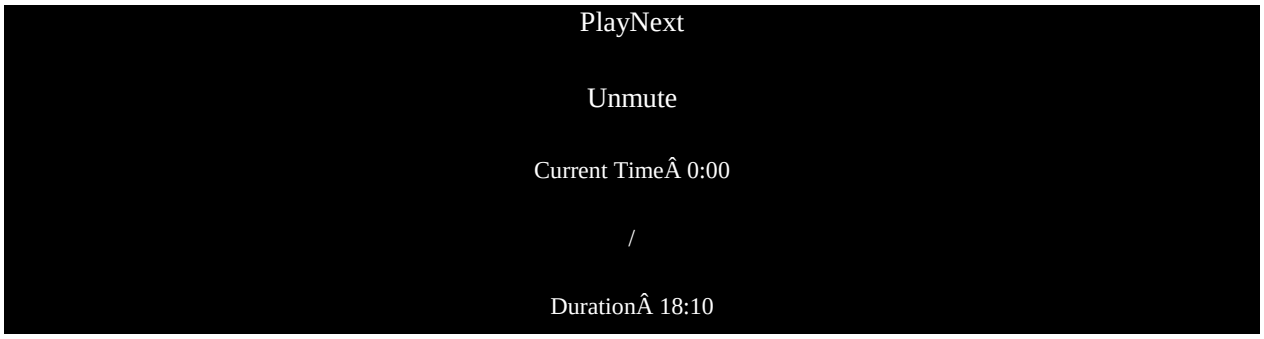
Merge Sort Algorithm

In this article, we will discuss the merge sort Algorithm. Merge sort is the sorting technique that follows the divide and conquer approach. This article will be very helpful and interesting to students as they might face merge sort as a question in their examinations. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic.

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Now, let's see the algorithm of merge sort.



Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

Algorithm

In the following algorithm, **arr** is the given array, **beg** is the starting element, and **end** is the last element of the array.

```
1. MERGE_SORT(arr, beg, end)
2.
3. if beg < end
4. set mid = (beg + end)/2
5. MERGE_SORT(arr, beg, mid)
6. MERGE_SORT(arr, mid + 1, end)
7. MERGE (arr, beg, mid, end)
8. end of if
9.
10. END MERGE_SORT
```

The important part of the merge sort is the **MERGE** function. This function performs the merging of two sorted sub-arrays that are **A[beg…mid]** and **A[mid+1…end]**, to build one sorted array **A[beg…end]**. So, the inputs of the **MERGE** function are **A[], beg, mid,** and **end**.

The implementation of the **MERGE** function is given as follows -

```
1. /* Function to merge the subarrays of a[] */
2. void merge(int a[], int beg, int mid, int end)
3. {
4.     int i, j, k;
5.     int n1 = mid - beg + 1;
6.     int n2 = end - mid;
```

```
7.
8.      int LeftArray[n1], RightArray[n2]; //temporary arrays
9.
10.     /* copy data to temp arrays */
11.     for (int i = 0; i < n1; i++)
12.     LeftArray[i] = a[beg + i];
13.     for (int j = 0; j < n2; j++)
14.     RightArray[j] = a[mid + 1 + j];
15.
16.     i = 0, /* initial index of first sub-array */
17.     j = 0; /* initial index of second sub-array */
18.     k = beg;  /* initial index of merged sub-array */
19.
20.     while (i < n1 && j < n2)
21.     {
22.        if(LeftArray[i] <= RightArray[j])
23.        {
24.           a[k] = LeftArray[i];
25.           i++;
26.        }
27.        else
28.        {
29.           a[k] = RightArray[j];
30.           j++;
31.        }
32.        k++;
33.     }
34.     while (i<n1)
35.     {
36.        a[k] = LeftArray[i];
37.        i++;
38.        k++;
39.     }
40.
```

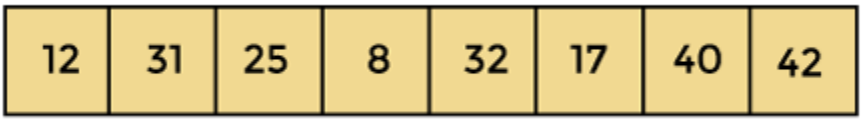
```
41.     while (j<n2)
42.     {
43.         a[k] = RightArray[j];
44.         j++;
45.         k++;
46.     }
47. }
```

Working of Merge sort Algorithm

Now, let's see the working of merge sort Algorithm.

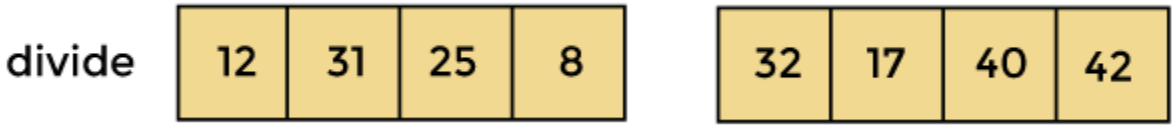
To understand the working of the merge sort algorithm, let's take an unsorted array. It will be easier to understand the merge sort via an example.

Let the elements of array are -

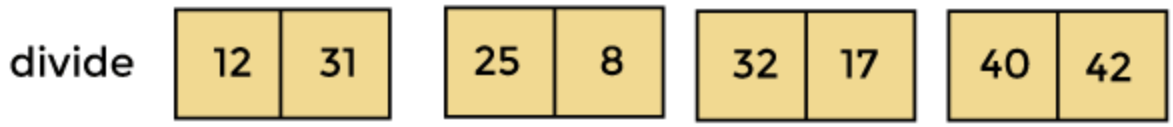


According to the merge sort, first divide the given array into two equal halves. Merge sort keeps dividing the list into equal parts until it cannot be further divided.

As there are eight elements in the given array, so it is divided into two arrays of size 4.



Now, again divide these two arrays into halves. As they are of size 4, so divide them into new arrays of size 2.

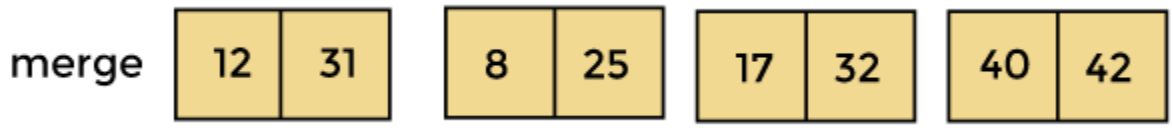


Now, again divide these arrays to get the atomic value that cannot be further divided.

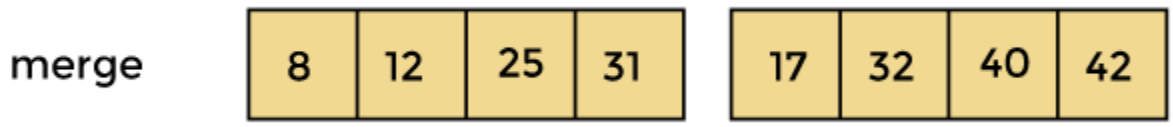
Now, combine them in the same manner they were broken.

In combining, first compare the element of each array and then combine them into another array in sorted order.

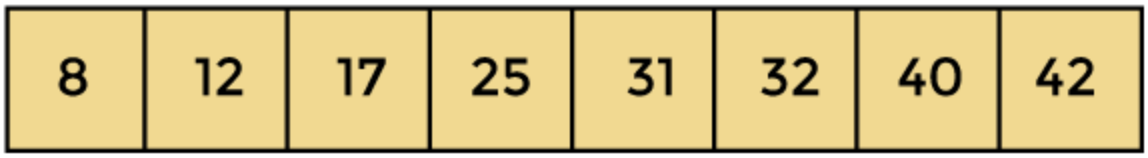
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



In the next iteration of combining, now compare the arrays with two data values and merge them into an array of found values in sorted order.



Now, there is a final merging of the arrays. After the final merging of above arrays, the array will look like -



Now, the array is completely sorted.

Merge sort complexity

Now, let's see the time complexity of merge sort in best case, average case, and in worst case. We will also see the space complexity of the merge sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n*logn)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n*logn)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n*logn)**.

2. Space Complexity

Space Complexity	O(n)
Stable	YES

- The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

Implementation of merge sort

Now, let's see the programs of merge sort in different programming languages.

Program: Write a program to implement merge sort in C language.

```
1.  #include <stdio.h>
2.
3.  /* Function to merge the subarrays of a[] */
4.  void merge(int a[], int beg, int mid, int end)
5.  {
6.      int i, j, k;
7.      int n1 = mid - beg + 1;
8.      int n2 = end - mid;
9.
10.     int LeftArray[n1], RightArray[n2]; //temporary arrays
11.
12.     /* copy data to temp arrays */
13.     for (int i = 0; i < n1; i++)
14.     LeftArray[i] = a[beg + i];
15.     for (int j = 0; j < n2; j++)
16.     RightArray[j] = a[mid + 1 + j];
17.
18.     i = 0; /* initial index of first sub-array */
19.     j = 0; /* initial index of second sub-array */
```

```
20.    k = beg;  /* initial index of merged sub-array */
21.
22.    while (i < n1 && j < n2)
23.    {
24.       if(LeftArray[i] <= RightArray[j])
25.       {
26.          a[k] = LeftArray[i];
27.          i++;
28.       }
29.       else
30.       {
31.          a[k] = RightArray[j];
32.          j++;
33.       }
34.       k++;
35.    }
36.    while (i<n1)
37.    {
38.       a[k] = LeftArray[i];
39.       i++;
40.       k++;
41.    }
42.
43.    while (j<n2)
44.    {
45.       a[k] = RightArray[j];
46.       j++;
47.       k++;
48.    }
49. }
50.
51. void mergeSort(int a[], int beg, int end)
52. {
53.    if (beg < end)
```

```
54.     {
55.         int mid = (beg + end) / 2;
56.         mergeSort(a, beg, mid);
57.         mergeSort(a, mid + 1, end);
58.         merge(a, beg, mid, end);
59.     }
60. }
61.
62. /* Function to print the array */
63. void printArray(int a[], int n)
64. {
65.     int i;
66.     for (i = 0; i < n; i++)
67.         printf("%d ", a[i]);
68.     printf("\n");
69. }
70.
71. int main()
72. {
73.     int a[] = { 12, 31, 25, 8, 32, 17, 40, 42 };
74.     int n = sizeof(a) / sizeof(a[0]);
75.     printf("Before sorting array elements are - \n");
76.     printArray(a, n);
77.     mergeSort(a, 0, n - 1);
78.     printf("After sorting array elements are - \n");
79.     printArray(a, n);
80.     return 0;
81. }
```

Output:

```
Before sorting array elements are -
12 31 25 8 32 17 40 42
After sorting array elements are -
8 12 17 25 31 32 40 42
```

Program: Write a program to implement merge sort in C++ language.

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  /* Function to merge the subarrays of a[] */
6.  void merge(int a[], int beg, int mid, int end)
7.  {
8.      int i, j, k;
9.      int n1 = mid - beg + 1;
10.     int n2 = end - mid;
11.
12.     int LeftArray[n1], RightArray[n2]; //temporary arrays
13.
14.     /* copy data to temp arrays */
15.     for (int i = 0; i < n1; i++)
16.     LeftArray[i] = a[beg + i];
17.     for (int j = 0; j < n2; j++)
18.     RightArray[j] = a[mid + 1 + j];
19.
20.     i = 0; /* initial index of first sub-array */
21.     j = 0; /* initial index of second sub-array */
22.     k = beg;  /* initial index of merged sub-array */
23.
24.     while (i < n1 && j < n2)
25.     {
26.         if(LeftArray[i] <= RightArray[j])
27.         {
28.             a[k] = LeftArray[i];
29.             i++;
30.         }
31.         else
32.         {
```

```
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}

void mergeSort(int a[], int beg, int end)
{
    if (beg < end)
    {
        int mid = (beg + end) / 2;
        mergeSort(a, beg, mid);
        mergeSort(a, mid + 1, end);
        merge(a, beg, mid, end);
    }
}

/* Function to print the array */
void printArray(int a[], int n)
{
```

```
67.     int i;
68.     for (i = 0; i < n; i++)
69.         cout<<a[i]<<" ";
70. }
71.
72. int main()
73. {
74.     int a[] = { 11, 30, 24, 7, 31, 16, 39, 41 };
75.     int n = sizeof(a) / sizeof(a[0]);
76.     cout<<"Before sorting array elements are - \n";
77.     printArray(a, n);
78.     mergeSort(a, 0, n - 1);
79.     cout<<"\nAfter sorting array elements are - \n";
80.     printArray(a, n);
81.     return 0;
82. }
```

Output:

```
Before sorting array elements are -
11 30 24 7 31 16 39 41
After sorting array elements are -
7 11 16 24 30 31 39 41
```

Program: Write a program to implement merge sort in Java.

```
1.  class Merge {
2.
3.  /* Function to merge the subarrays of a[] */
4.  void merge(int a[], int beg, int mid, int end)
5.  {
6.      int i, j, k;
7.      int n1 = mid - beg + 1;
8.      int n2 = end - mid;
9.
10.    /* temporary Arrays */
```

```
11.      int LeftArray[] = new int[n1];
12.      int RightArray[] = new int[n2];
13.
14.   /* copy data to temp arrays */
15.   for (i = 0; i < n1; i++)
16.   LeftArray[i] = a[beg + i];
17.   for (j = 0; j < n2; j++)
18.   RightArray[j] = a[mid + 1 + j];
19.
20.   i = 0; /* initial index of first sub-array */
21.   j = 0; /* initial index of second sub-array */
22.   k = beg;  /* initial index of merged sub-array */
23.
24.   while (i < n1 && j < n2)
25.   {
26.      if(LeftArray[i] <= RightArray[j])
27.      {
28.         a[k] = LeftArray[i];
29.         i++;
30.      }
31.      else
32.      {
33.         a[k] = RightArray[j];
34.         j++;
35.      }
36.      k++;
37.   }
38.   while (i<n1)
39.   {
40.      a[k] = LeftArray[i];
41.      i++;
42.      k++;
43.   }
44.
```

```
45.     while (j<n2)
46.     {
47.         a[k] = RightArray[j];
48.         j++;
49.         k++;
50.     }
51. }
52.
53. void mergeSort(int a[], int beg, int end)
54. {
55.     if (beg < end)
56.     {
57.         int mid = (beg + end) / 2;
58.         mergeSort(a, beg, mid);
59.         mergeSort(a, mid + 1, end);
60.         merge(a, beg, mid, end);
61.     }
62. }
63.
64. /* Function to print the array */
65. void printArray(int a[], int n)
66. {
67.     int i;
68.     for (i = 0; i < n; i++)
69.         System.out.print(a[i] + " ");
70. }
71.
72. public static void main(String args[])
73. {
74.     int a[] = { 11, 30, 24, 7, 31, 16, 39, 41 };
75.     int n = a.length;
76.     Merge m1 = new Merge();
77.     System.out.println("\nBefore sorting array elements are - ");
78.     m1.printArray(a, n);
```

```
79.     m1.mergeSort(a, 0, n - 1);
80.     System.out.println("\nAfter sorting array elements are - ");
81.     m1.printArray(a, n);
82.     System.out.println("");
83. }
84.
85.  }
```

Output:

```
D:\JTP>javac Merge.java

D:\JTP>java Merge

Before sorting array elements are -
11 30 24 7 31 16 39 41
After sorting array elements are -
7 11 16 24 30 31 39 41
```

Program: Write a program to implement merge sort in C#.

```
1.  using System;
2.  class Merge {
3.
4.  /* Function to merge the subarrays of a[] */
5.  static void merge(int[] a, int beg, int mid, int end)
6.  {
7.     int i, j, k;
8.     int n1 = mid - beg + 1;
9.     int n2 = end - mid;
10.
11.   //temporary arrays
12.    int[] LeftArray = new int [n1];
13.    int[] RightArray = new int [n2];
14.
15.    /* copy data to temp arrays */
16.    for (i = 0; i < n1; i++)
17.    LeftArray[i] = a[beg + i];
```

```
18.    for (j = 0; j < n2; j++)
19.    RightArray[j] = a[mid + 1 + j];
20.
21.    i = 0; /* initial index of first sub-array */
22.    j = 0; /* initial index of second sub-array */
23.    k = beg;  /* initial index of merged sub-array */
24.
25.    while (i < n1 && j < n2)
26.    {
27.       if(LeftArray[i] <= RightArray[j])
28.       {
29.          a[k] = LeftArray[i];
30.          i++;
31.       }
32.       else
33.       {
34.          a[k] = RightArray[j];
35.          j++;
36.       }
37.       k++;
38.    }
39.    while (i<n1)
40.    {
41.       a[k] = LeftArray[i];
42.       i++;
43.       k++;
44.    }
45.
46.    while (j<n2)
47.    {
48.       a[k] = RightArray[j];
49.       j++;
50.       k++;
51.    }
```

```
52. }
53.
54. static void mergeSort(int[] a, int beg, int end)
55. {
56.     if (beg < end)
57.     {
58.         int mid = (beg + end) / 2;
59.         mergeSort(a, beg, mid);
60.         mergeSort(a, mid + 1, end);
61.         merge(a, beg, mid, end);
62.     }
63. }
64.
65. /* Function to print the array */
66. static void printArray(int[] a, int n)
67. {
68.     int i;
69.     for (i = 0; i < n; i++)
70.         Console.Write(a[i] + " ");
71. }
72.
73. static void Main()
74. {
75.     int[] a = { 10, 29, 23, 6, 30, 15, 38, 40 };
76.     int n = a.Length;
77.     Console.Write("Before sorting array elements are - ");
78.     printArray(a, n);
79.     mergeSort(a, 0, n - 1);
80.     Console.Write("\nAfter sorting array elements are - ");
81.     printArray(a, n);
82. }
83.
84.  }
```

Output:

```
Before sorting array elements are - 10 29 23 6 30 15 38 40
After sorting array elements are - 6 10 15 23 29 30 38 40
```

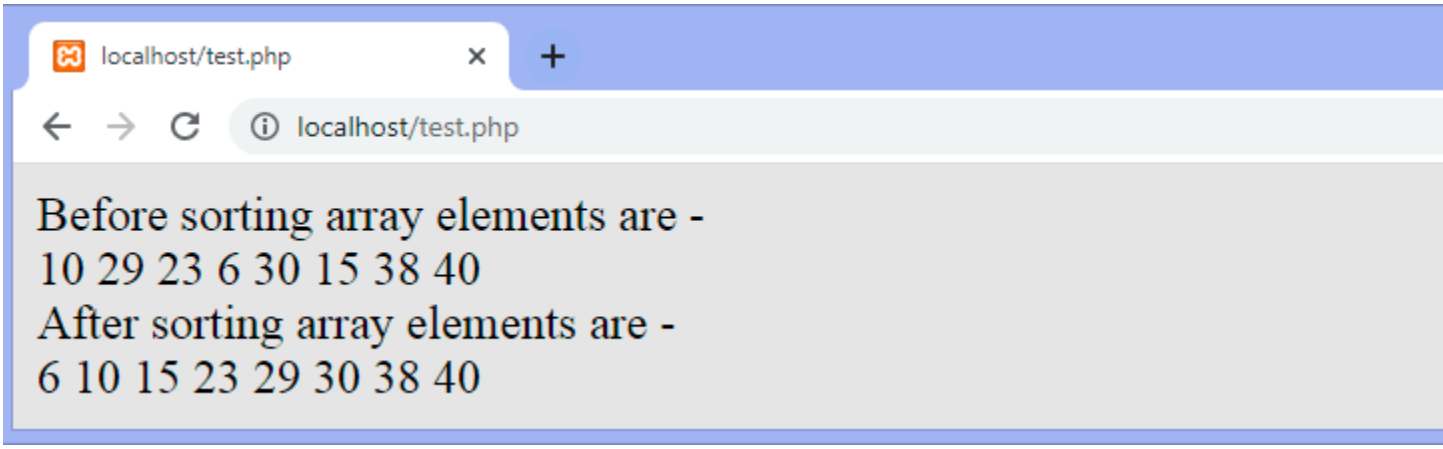
Program: Write a program to implement merge sort in PHP.

```
<?php

/* Function to merge the subarrays of a[] */
function merge(&$a, $beg, $mid, $end)
{
    $n1 = ($mid - $beg) + 1;
    $n2 = $end - $mid;

   /* temporary Arrays */
      $LeftArray = array($n1);
      $RightArray = array($n2);

   /* copy data to temp arrays */
   for ($i = 0; $i < $n1; $i++)
   $LeftArray[$i] = $a[$beg + $i];
   for ($j = 0; $j < $n2; $j++)
   $RightArray[$j] = $a[$mid + 1 + $j];

   $i = 0; /* initial index of first sub-array */
   $j = 0; /* initial index of second sub-array */
   $k = $beg;  /* initial index of merged sub-array */

   while ($i<$n1 && $j<$n2)
   {
      if($LeftArray[$i] <= $RightArray[$j])
      {
         $a[$k] = $LeftArray[$i];
```

```
28.             $i++;
29.         }
30.         else
31.         {
32.             $a[$k] = $RightArray[$j];
33.             $j++;
34.         }
35.         $k++;
36.     }
37.     while ($i<$n1)
38.     {
39.         $a[$k] = $LeftArray[$i];
40.         $i++;
41.         $k++;
42.     }
43.
44.     while ($j<$n2)
45.     {
46.         $a[$k] = $RightArray[$j];
47.         $j++;
48.         $k++;
49.     }
50. }
51.
52. function mergeSort(&$a, $beg, $end)
53. {
54.     if ($beg < $end)
55.     {
56.         $mid = (int)(($beg + $end) / 2);
57.         mergeSort($a, $beg, $mid);
58.         mergeSort($a, $mid + 1, $end);
59.         merge($a, $beg, $mid, $end);
60.     }
61. }
```

```
62.
63. /* Function to print array elements */
64.     function printArray($a, $n)
65.     {
66.     for($i = 0; $i < $n; $i++)
67.     {
68.         print_r($a[$i]);
69.         echo " ";
70.     }
71.     }
72.
73.     $a = array( 10, 29, 23, 6, 30, 15, 38, 40 );
74.     $n = count($a);
75.     echo "Before sorting array elements are - <br>";
76.     printArray($a, $n);
77.     mergeSort($a, 0, $n - 1);
78.     echo "<br> After sorting array elements are - <br>";
79.     printArray($a, $n);
80. ?>
```

Output:



So, that's all about the article. Hope the article will be helpful and informative to you.

This article was not only limited to the algorithm. We have also discussed the Merge sort complexity, working, and implementation in different programming languages.

Advanced Searching

Introduction to Hashing

Assume we want to create a system for storing employee records that include phone numbers (as keys). We also want the following queries to run quickly:

- Insert a phone number and any necessary information.
- Look up a phone number and get the information.
- Remove a phone number and any associated information.

We can consider using the following data structures to store information about various phone numbers.

- A collection of phone numbers and records.
- Phone numbers and records are linked in this list.
- Phone numbers serve as keys in a balanced binary search tree.
- Table with Direct Access.

We must search in a linear fashion for arrays and linked lists, which can be costly in practise. If we use arrays and keep the data sorted, we can use Binary Search to find a phone number in O(Logn) time, but insert and delete operations become expensive because we must keep the data sorted.

We get moderate search, insert, and delete times with a balanced binary search tree. All of these operations will be completed in O(Logn) time.

The term "access-list" refers to a set of rules for controlling network traffic and reducing network attacks. ACLs are used to filter network traffic based on a set of rules defined for incoming or outgoing traffic.

Another option is to use a direct access table, in which we create a large array and use phone numbers as indexes. If the phone number is not present, the array entry is NIL; otherwise, the array entry stores a pointer to the records corresponding to the phone number. In terms of time complexity, this solution is the best of the bunch; we can perform all operations in O(1) time. To insert a phone number, for example, we create a record with the phone number's details, use the phone number as an index, and store the pointer to the newly created record in the table.

This solution has a number of practical drawbacks. The first issue with this solution is the amount of extra space required. For example, if a phone number has n digits, we require O(m * 10n) table space, where m is the size of a pointer to record. Another issue is that an integer in a programming language cannot hold n digits.

Because of the limitations mentioned above, Direct Access Table cannot always be used. In practise, Hashing is the solution that can be used in almost all such situations and outperforms the above data structures such as Array, Linked List, and Balanced BST. We get O(1) search time on average (under reasonable assumptions) and O(n) in the worst case with hashing. Let's break down what hashing is.

What exactly do you mean by hashing?

Hashing is a popular technique for quickly storing and retrieving data. The primary reason for using hashing is that it produces optimal results by performing optimal searches.

Why should you use Hashing?

If we try to search, insert, or delete any element in a balanced binary search tree, the time complexity for the same is O. (logn). Now, there may be times when our applications need to perform the same operations in a faster, more optimised manner, and this is where hashing comes into play. All of the above operations in hashing can be completed in O(1), or constant time. It is critical to understand that hashing's worst-case time complexity remains O(n), but its average time complexity is O. (1).

Let us now look at some fundamental hashing operations.

Fundamental Operations:

- **HashTable:** Use this operation to create a new hash table.
- **Delete:** This operation is used to remove a specific key-value pair from the hash table.
- **Get:** This operation is used to find a key within the hash table and return the value associated with that key.
- **Put:** This operation is used to add a new key-value pair to the hash table.
- **DeleteHashTable:** This operation is used to remove the hash table.

Describe the hash function.

- A hash function is a fixed procedure that changes a key into a hash key.
- This function converts a key into a length-restricted value known as a hash value or hash.
- Although the hash value is typically less than the original, it nevertheless represents the original string of characters.
- The digital signature is transferred, and both the hash value and the signature are then given to the recipient. The hash value generated by the receiver using the same hash algorithm is compared to the hash value received along with the message.
- The message is sent without problems if the hash values match.

Hash Table: What is it?

- A data structure called a hash table or hash map is used to hold key-value pairs.
- It is a collection of materials that have been organised for later simple access.
- It computes an index into an array of buckets or slots from which the requested value can be located using a hash function.
- Each list in the array is referred to as a bucket.
- On the basis of the key, it contains value.
- The map interface is implemented using a hash table, which also extends the Dictionary class.
- The hash table is synchronised and only has distinct components.

Components of Hashing:

- **Hash Table:** An array that stores pointers to records that correspond to a specific phone number. If no existing phone number has a hash function value equal to the index for the entry, the entry in the hash table is NIL. In simple terms, a hash table is a generalisation of an array. A hash table provides the functionality of storing a collection of data in such a way that it is easy to find those items later if needed. This makes element searching very efficient.
- **Hash Function:** A function that reduces a large phone number to a small practical integer value. In a hash table, the mapped integer value serves as an index. So, to put it simply, a hash function is used to convert a given key into a specific slot index. Its primary function is to map every possible key to a unique slot index. The hash function is referred to as a perfect hash function if each key maps to a distinct slot index. Although it is exceedingly challenging to construct the ideal hash function, it is our responsibility as programmers to do so in a way that minimises the likelihood of collisions. This section will cover collision.

The following characteristics a decent hash function ought to have:

- Effectively calculable.
- The keys ought to be distributed equally among all table positions.
- Ought to reduce collisions.
- Low load factor should be the norm (number of items in table divided by size of the table).

A poor hash function for phone numbers, for instance, would be to use the first three digits. Consideration of the last three numbers is a better function. Please be aware that this hash function might not be the best. There could be better options.

- **Handling Collisions:** Because a hash function only returns a little number for a large key, it is possible that two keys will yield the same result. Collision occurs when a newly added key corresponds to a hash table slot that is already taken, and it needs to be

handled using a collision handling mechanism. The methods for handling collisions are as follows:

- Making each hash table cell point to a linked list of records with the same hash function value is known as "chaining." Although chaining is straightforward, more memory is needed outside of the table.
- Open Addressing: In open addressing, the hash table itself serves as the storage location for all items. Either a record or NIL is present in every table entry. When looking for an element, we look through each table slot individually until the sought-after element is discovered or it becomes obvious that the element is not in the table.

Linear Probing

In data structures, hashing produces array indexes that are already used to store a value. In this situation, hashing does a search operation and linearly probes for the subsequent empty cell.

The simplest method for handling collisions in hash tables is known as linear probing in hash algorithms. Any collision that occurred can be located using a sequential search.

Hashing twice

Two hash functions are used in the double hashing method. When the first hash function results in a collision, the second hash function is used. In order to store the value, it offers an offset index.

The double hashing method's formula is as follows:

(firstHash(key) + i * secondHash(key)) % sizeOfTable

The offset value is represented by i. The offset value is continuously increased until it encounters an empty slot.