UNIT 1

INTRODUCTION TO DATA STRUCTURE

Introduction: Dynamic aspects of operations on data, Characteristics of data, Creation, Manipulation and Operations on data, Data Structure and its types, Abstract Data Types (ADTs), Analysis of algorithms and its Types, Asymptotic Notation. Arrays: Allocation, Operations and Storage with one-dimensional arrays and multidimensional arrays. Stacks: Operations on Stacks, Applications of Stacks, Queues: Operations and its types. Linked lists: Operations and Types of linked list (singly, doubly and circularly linked list), Implementation of Stack and Queue using linked list, Applications of Linked List. Case Study: Tower of Hanoi, Sparse matrix.

Introduction

As a race, humans have always been faced with numerous challenges. Computers are arguably the most powerful problem-solving tool we have invented yet. There is only one problem, computers live in a virtual world outside our own. They don't understand the challenges and problems we go through.

So we have to describe to them, step by step, instructions for solving any instance of a particular problem. This is often referred to as an algorithm. Now, programming allows us to implement these algorithms on a computer and solve our problems!

Data and Information:

Data can be defined as a representation of facts, concepts, or instructions in a formalized manner, which should be suitable for communication, interpretation, or processing by human or electronic machine.

Data is represented with the help of characters such as alphabets (A-Z, a-z), digits (0-9) or special characters (+,-,/,*,<,>,= etc.)

In computing, data is defined as any form of information that has been gathered and organized in a meaningful format wherein they could be processed further. In other words, Data are known facts that can be recorded and have implicit meaning.

Data could be in the form of audio files, text documents, software programs, images etc. It is stored on the computer hard disk in binary digital format meaning it can be stored and processed digitally as well as could be transferred from one system to another.

Information is organized or classified data, which has some meaningful values for the receiver. Information is the processed data on which decisions and actions are based.

For the decision to be meaningful, the processed data must qualify for the following characteristics –

- **Timely** – Information should be available when required.
- **Accuracy** – Information should be accurate.

- **Completeness –** Information should be complete.

Characteristics of Data:

- Data should be precise which means it should contain accurate information. Precision saves time of the user as well as their money.
- Data should be relevant and according to the requirements of the user. Hence the legitimacy of the data should be checked before considering it for usage.
- Data should be consistent and reliable. False data is worse than incomplete data or no data at all.
- Relevance of data is necessary in order for it to be of good quality and useful. Although in today's world of dynamic data any relevant information is not complete at all times however at the time of its usage, the data has to be comprehensive and complete in its current form.
- A high quality data is unique to the requirement of the user. Moreover it is easily accessible and could be processed further with ease.

Variables

Variables are names acts as place holders for representing data.

For Example: x+2y-2 = 1, here the names x and y are place holders(variables)

Data Types

To solve problems with computers, we first have to express the problem as something the computer can understand. While trying to describe problems, we often realize they involve some kind of data we have to manipulate. It can be numbers, letters or something more complicated. Now, all programming languages come with some predefined ways of representing data while writing our programs.

The most basic form of data representation are data types.

Formally, we can define a data type as a set of data values having predefined characteristics.

A datatype in a programming language is a set of data with values having predefined characteristics

Take for example the integer data type. It allows us to represent whole numbers and thus provides us a way to solve problems that deals with counting. Other example of data types includes Boolean, Float, Double, String and Character.

There are two types of data types:

- System defined data types (also called primitive data types)
- User defined data types

System defined data types:

Data types which are defined by system are called *primitive data types*. The primitive data types which are provided by many programming languages are : int, float, char, double, bool, etc.

User defined data types:

Most programming language allows the users to define their own data types called as user defined data types.

For Example: Structures in C/C++ and classes in java

Data Structures

A data structure is a way of storing data in a computer so that it can be used efficiently and it will allow the most efficient algorithm to be used. The choice of the data structure begins from the choice of an abstract data type (ADT).

A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible.

Data structure introduction refers to a scheme for organizing data, or in other words it is an arrangement of data in computer's memory in such a way that it could make the data quickly available to the processor for required calculations.

A data structure should be seen as a logical concept that must address two fundamental concerns.

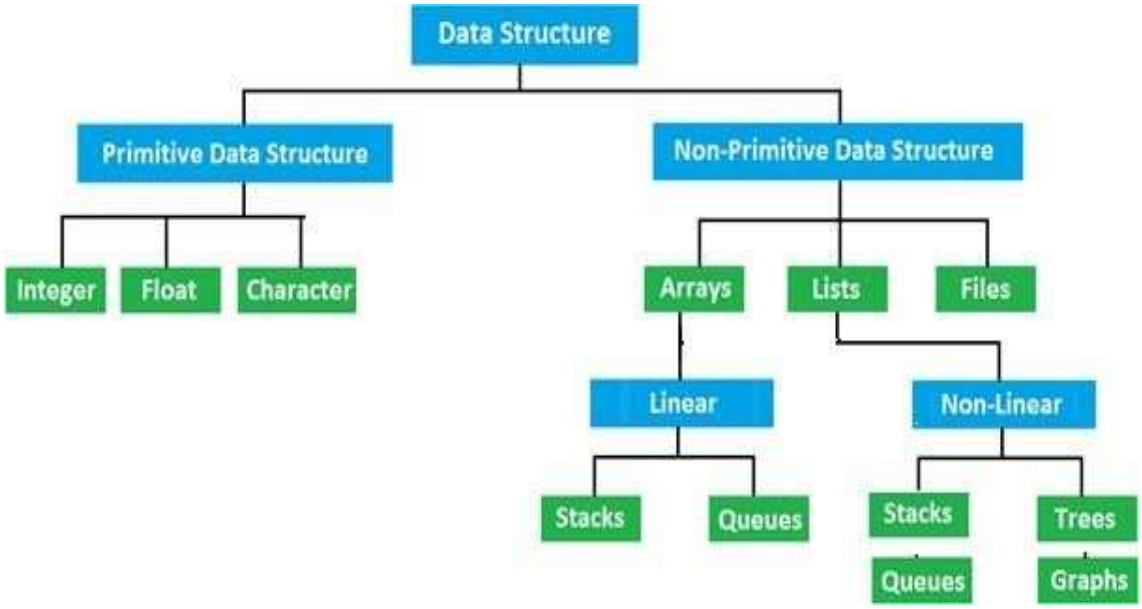
1. First, how the data will be stored, and

2. Second, what operations will be performed on it.

As data structure is a scheme for data organization so the functional definition of a data structure should be independent of its implementation. The functional definition of a data structure is known as ADT (Abstract Data Type) which is independent of implementation.

The way in which the data is organized affects the performance of a program for different tasks. Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data. Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs.

Classification of Data Structures:



Data structures can be classified as

- Simple data structure
- Compound data structure
- Linear data structure
- Non Linear data structure

Simple Data Structure:

Simple data structure can be constructed with the help of primitive data structure. A primitive data structure used to represent the standard data types of any one of the computer languages. Variables, arrays, pointers, structures, unions, etc. are examples of primitive data structures.

Compound Data structure:

Compound data structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user.

It can be classified as

- Linear data structure
- Non-linear data structure

Linear Data Structure:

Linear data structures can be constructed as a continuous arrangement of data elements in the memory.

It can be constructed by using array data type. In the linear Data Structures the relationship of adjacency is maintained between the data elements.

Operations applied on linear data structure:

The following list of operations applied on linear data structures

1. Add an element
2. Delete an element
3. Traverse
4. Sort the list of elements
5. Search for a data element

For example Stack, Queue, Tables, List, and Linked Lists.

Non-linear Data Structure:

Non-linear data structure can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear Data structure the relationship of adjacency is not maintained between the data items.

Operations applied on non-linear data structures:

The following list of operations applied on non-linear data structures.

1. Add elements
2. Delete elements
3. Display the elements
4. Sort the list of elements
5. Search for a data element

For example: Tree, Decision tree, Graph and Forest

What is an Algorithm?

An algorithm is the step-by-step instructions to solve a given problem

Structure and Properties of Algorithm:

An algorithm has the following structure

1. Input Step
2. Assignment Step
3. Decision Step
4. Repetitive Step
5. Output Step

An algorithm is endowed with the following properties:

1. Finiteness: An algorithm must terminate after a finite number of steps.

2. Definiteness: The steps of the algorithm must be precisely defined or unambiguously specified.

3. Generality: An algorithm must be generic enough to solve all problems of a particular class.

4. Effectiveness: the operations of the algorithm must be basic enough to be put down on pencil and paper. They should not be too complex to warrant writing another algorithm for the operation.

5. Input-Output: The algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

Why Analysis of Algorithms?

Algorithm analysis helps us determining which of them is efficient in terms of time and space consumed.

The goal of analysis of algorithms is to compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory, developer's effort etc.)

Practical Algorithm Design Issues:

1. To save time (Time Complexity): A program that runs faster is a better program.

2. To save space (Space Complexity): A program that saves space over a competing program is considerable desirable.

Efficiency of Algorithms:

The performances of algorithms can be measured on the scales of time and space. The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical and the other is experimental.

In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity: The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program. In other words, it is the amount of computer time it needs to run to completion.

Space Complexity: The space complexity of an algorithm or program is a function of the space needed by the algorithm or program to run to completion.

The time complexity of an algorithm can be computed either by an empirical or theoretical approach. The empirical or posteriori testing approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem. The time taken by the execution of the programs for various instances of the problem are noted and compared. The algorithm whose implementation yields the least time is considered as the best among the candidate algorithmic solutions.

Time and space complexity depends on lots of things like hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing the algorithm. We will only consider the execution time of an algorithm.

Order of growth is how the time of execution depends on the length of the input. In the above example, we can clearly see that the time of execution is linearly depends on the length of the array. Order of growth will help us to compute the running time with ease. We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input. We use different notation to describe limiting behavior of a function.

O-notation:
To denote asymptotic upper bound, we use O-notation.

For a given function g(n), we denote by O(g(n)) (pronounced "big-oh of g of n") the set of functions:

O(g(n))= { f(n) : there exist positive constants c and n0 such that $0 \le f(n) \le c*g(n)$ for all $n \ge n0$ }

Ω-notation:
To denote asymptotic lower bound, we use Ω-notation.

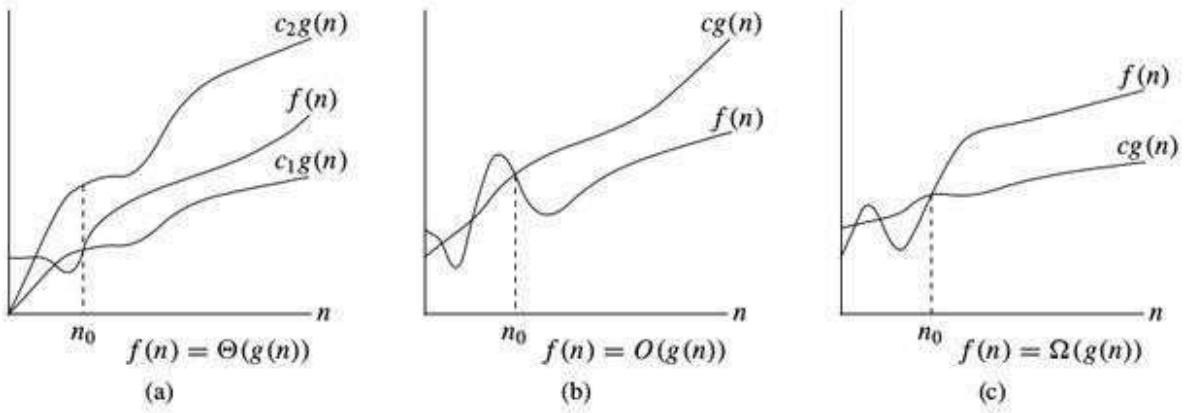
For a given function g(n), we denote by Ω(g(n)) (pronounced "big-omega of g of n") the set of functions:

Ω(g(n))= { f(n) : there exist positive constants c and n0 such that $0 \le c*g(n) \le f(n)$ for all $n \ge n0$ }

Θ-notation:
To denote asymptotic tight bound, we use Θ-notation.

For a given function g(n), we denote by Θ(g(n)) (pronounced "big-theta of g of n") the set of functions:

Θ(g(n))= { f(n) : there exist positive constants c1,c2 and n0 such that $0 \le c1*g(n) \le f(n) \le c2*g(n)$ for all $n > n0$ }



Time complexity notations

While analysing an algorithm, we mostly consider O-notation because it will give us an upper limit of the execution time i.e. the execution time in the worst case.
To compute O-notation we will ignore the lower order terms, since the lower order terms are relatively insignificant for large input.

Let f(N)=2∗N^2+3∗N+5
O(f(N))=O(2∗N^2+3∗N+5)=O(N^2)

Lets consider some example:

1.

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = 0; j < i; j++)
      count++;
```

Lets see how many times **count++** will run.

When i=0, it will run 0 times.
When i=1, it will run 1 times.
When i=2, it will run 2 times and so on.
Total number of times **count++** will run is 0+1+2+...+(N−1)=N∗(N−1)/2. So the time complexity will be O(N^2).

2.

```
int count = 0;
for (int i = N; i > 0; i /= 2)
   for (int j = 0; j < i; j++)
      count++;
```

This is a tricky case. In the first look, it seems like the complexity is O(N∗logN). N for the j's loop and logN for i's loop. But its wrong. Lets see why.

Think about how many times **count++** will run.

When i=N, it will run N times.
When i=N/2, it will run N/2 times.
When i=N/4, it will run N/4 times and so on.
Total number of times **count++** will run is N+N/2+N/4+...+1=2∗N. So the time complexity will be O(N)

Abstract Data Type:

What is Abstract Data Type?

To simplify the process of solving the problems, combine the data structures along with their operations and are called Abstract Data Types (ADTs).

An ADT consists of two parts:

- Declaration of data
- Declaration of operations

Why ADT?

Modularity

- ✓ divide program into small functions
- ✓ easy to debug and maintain
- ✓ easy to modify
- ✓ group work

Reuse

- ✓ do some operations only once

Easy to change the implementation

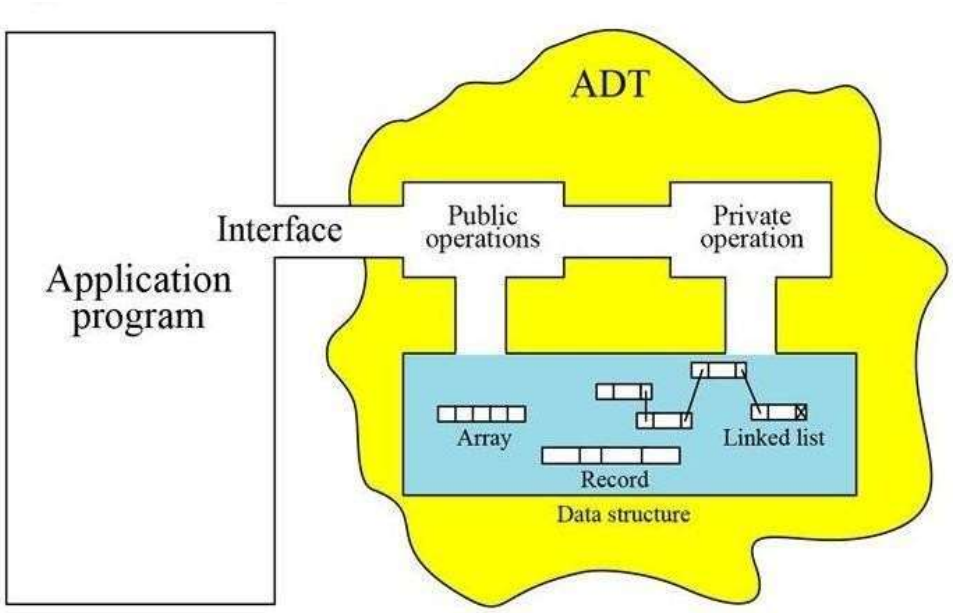
- ✓ transparent to the program

Definition:

- Declaration of data
- Declaration of operation
- Encapsulation of data and operations

Basically, It is a data declaration packaged together with the **operations** that are meaningful for the data type.

Model of an Abstract Data Type:



To implement an ADT, you need to choose:

A data representation

- ✓ must be able to represent all necessary values of the ADT
- ✓ should be private

An algorithm for each of the necessary operation:

- ✓ must be consistent with the chosen representation
- ✓ all auxiliary (helper) operations that are not in the contract should be private

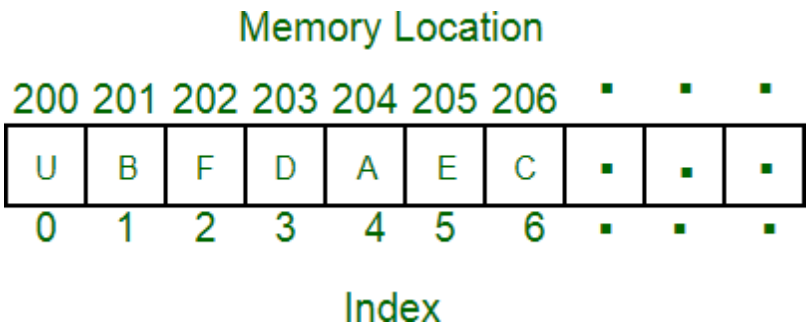
- ❑ Remember: Once other people are using it, It's easy to add functionality

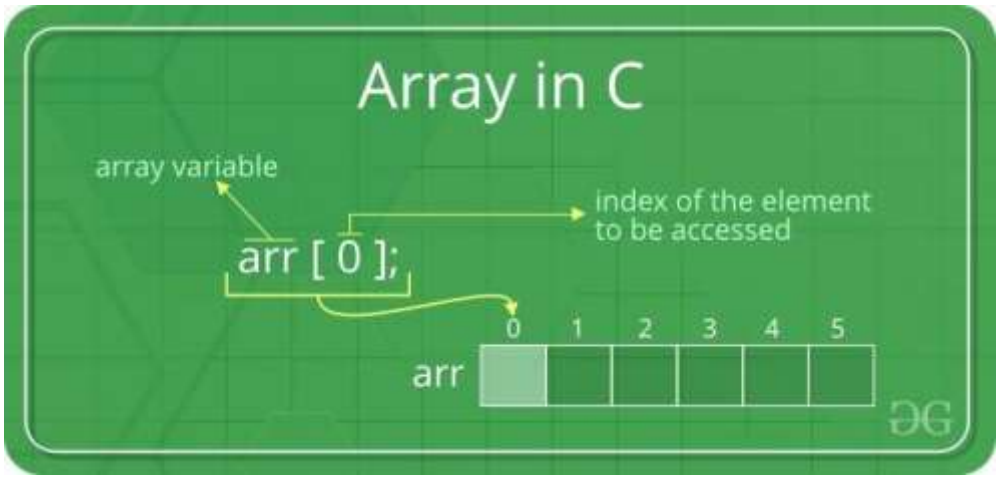
List ADT:

Array Implementation:

An array is collection of items stored at contiguous memory locations. The idea is to store multiple items of same type together.

This makes it easier to calculate the position of each element by simply adding an offset to a base value.





Need of Array

When there is a demand to store multiple values under a common name which can be accessed with the help of index value.

Types of Array

1. One Dimension Array, example: int a[5];
2. Multi Dimension Array, example: int a[2][3];

Array Declaration, Initialization and Accessing Values

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

// An Integer array in C/C++/Java
int arr2[] = {10, 20, 30, 40, 50};

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

Advantages of using arrays:

- Arrays allow **random access** of elements. This makes accessing elements by position faster.

- Arrays have **better cache locality** that can make a pretty big difference in performance.

API's of Array

Create API

```
Int* create()
{
        int i,n; printf("Enter array size:\n");
        scanf("%d",&n);
         for(i=0;i<n;i++)
        {
                printf("Enter the values:");
                scanf("%d", &x[i]);
        }
printf("Values are saved\n\n\n");
return x;
}
```

Insert Front API

```
Int* insert_front(int n, int* x,int y)
{
        int i;
        scanf("%d",&y);
        for(i=n;i>0;i--)
        {
                x[i]=x[i-1];
        }
        x[0]=y;
        return x;
}
```

Insert Rear API

```
Int* insert_rear(int n, int* x,int y )
{
        int i;
        scanf("%d",&y);
        for(i=n-1;i>=n-1;i--)
        {
                x[i+1]=x[i];
```

```
        }
        n++;
        x[n-1]=y;
        return x;
}
```

Insert Particular Position API

```
Int* insert_position(int n , int* x,int p,int y )
{
        int i;
        scanf("%d%d",&p,&y);
        for(i=n;i>p;i--)
        {
        x[i]=x[i-1];
        }
        x[p]=y;
printf("%d",y);
printf("x[%d]=%d",i,x[i]);
return x;
}
```

Delete Front API

```
Int* delete_front(int n, int* x)
{
        int i;
        for(i=0;i<n-1;i++)
        {
        x[i]=x[i+1];
        }
        return x;
}
```

Delete Rear API

```
Int* delete_rear(int n , int* x )
{
        int i;
        for(i=n;i>=n-1;i--)
        x[i]= NULL;
```

```
        return x;
}
```

Delete Element API

```
Int* delete_element(int n , int* x, int y )
{
        int i,s,p;
        scanf("%d",&y);
        for(i=0;i<n;i++)
        if (x[i]==y)
        s=1; p=i; break;
        if(s==1)
        for(i=p;i<n-1;i++)
        x[i]=x[i+1];
        return x;
}
```

Search API

```
void search(int n, int* x,int y)
{
        int i;
        scanf("%d",&y);
        for(i=0;i<n;i++)
        {
                if(y==x[i])
                {
                printf("Value is present in:%d\n\n\n",i);
                }
        }
}
```

Update API

```
void update(int n, int* x, int p, int y)
{
        int i;
        scanf("%d%d",&p,&y);
        for(i=0;i<n;i++)
        {
```

```
        x[p]=y;
        }
}
```

Traverse Forward API

```
void traverse_forward(int n, int* x)
{
        int i;
        for(i=0;i<n;i++)
        {
        printf("Values are :%d\n",x[i]);
        }
}
```

Traverse Backward API

```
void traverse_backward(int n, int* x)
{
        int i;
        for(i=n-1;i>=0;i--)
        {
        printf("Values are :%d\n",x[i]);
        }
}
```

Example Program with Time Complexity

3 Sum Problem

3-SUM. Given N distinct integers, how many triples sum to exactly zero?

30 -40 -20 -10 40 0 10 5

	a[i]	a[j]	a[k]	sum
1	30	-40	10	0
2	30	-20	-10	0
3	-40	40	0	0
4	-10	0	10	0

Solution:

```
#include<stdio.h>
int main( )
{
int a[4]={-10,0,-5,15};
int t,count=0,x,i,j,k;
for(i=0;i<4;i++)
printf("%d ",a[i]);
for(i=0;i<4;i++)
{
        for(j=i+1;j<4;j++)
        {
                for(k=j+1;k<4;k++)
                {
                        if((a[i]+a[j]+a[k])==0)
                        count++;
                }
        }
}
printf("\n");
printf("Count= %d",count);
}
```

Output:

-10 0 -5 15

Count= 1

Time Complexity: $O(n^3)$

Objective Questions

Taken from: https://www.sanfoundry.com/data-structure-questions-answers-array-array-operations/

1. Which of these best describes an array?
a) A data structure that shows a hierarchical behavior
b) Container of objects of similar types
c) Container of objects of mixed types
d) All of the mentioned
Answer: b
Explanation: Array contains elements only of the same type.

2. How do you initialize an array in C?
a) int arr[3] = (1,2,3);
b) int arr(3) = {1,2,3};
c) int arr[3] = {1,2,3};
d) int arr(3) = (1,2,3);
Answer: c
Explanation: This is the syntax to initialize an array in C.

3. How do you instantiate an array in Java?
a) int arr[] = new int(3);
b) int arr[];
c) int arr[] = new int[3];
d) int arr() = new int(3);
Answer: c
Explanation: Note that option b is declaration whereas option c is to instantiate an array.

4. Which of the following is a correct way to declare a multidimensional array in Java?
a) int[][] arr;
b) int arr[][];
c) int []arr[];
d) All of the mentioned
Answer: d
Explanation: All the options are syntactically correct.

5. What is the output of the following piece of code?

```
public class array
{
        public static void main(String args[])
        {
                int []arr = {1,2,3,4,5};
                System.out.println(arr[2]);
                System.out.println(arr[4]);
        }
}
```

a) 3 and 5
b) 5 and 3
c) 2 and 4
d) 4 and 2
Answer: a
Explanation: Array indexing starts from 0.

6. What is the output of the following piece of code?

```
public class array
{
        public static void main(String args[])
        {
```

```
                int []arr = {1,2,3,4,5};
                System.out.println(arr[5]);
        }
}
```

a) 4
b) 5
c) ArrayIndexOutOfBoundsException
d) InavlidInputException
Answer: c

Explanation: Trying to access an element beyond the limits of an array gives ArrayIndexOutOfBoundsException.

7. When does the ArrayIndexOutOfBoundsException occur?
a) Compile-time
b) Run-time
c) Not an error
d) None of the mentioned
Answer: b

Explanation: ArrayIndexOutOfBoundsException is a run-time exception and the compilation is error-free.

8. Which of the following concepts make extensive use of arrays?
a) Binary trees
b) Scheduling of processes
c) Caching
d) Spatial locality
Answer: d

Explanation: Whenever a particular memory location is referred, it is likely that the locations nearby are also referred, arrays are stored as contigous blocks in memory, so if you want to access array elements, spatial locality makes it to access quickly.

9. What are the advantages of arrays?
a) Easier to store elements of same data type
b) Used to implement other data structures like stack and queue
c) Convenient way to represent matrices as a 2D array
d) All of the mentioned
Answer: d

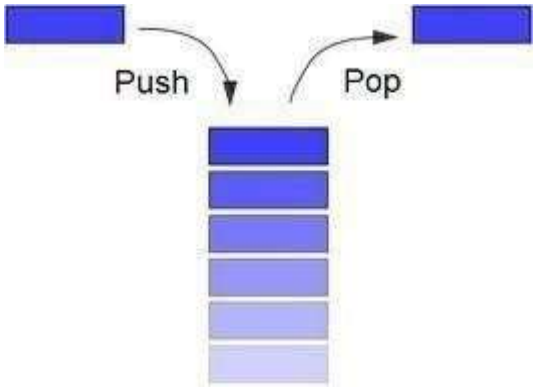
Explanation: Arrays are simple to implement when it comes to matrices of fixed size and type, or to implement other data structures.

10. What are the disadvantages of arrays?
a) We must know before hand how many elements will be there in the array
b) There are chances of wastage of memory space if elements inserted in an array are lesser than than the allocated size
c) Insertion and deletion becomes tedious
d) All of the mentioned
Answer: d

STACK

WHAT IS STACK?

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

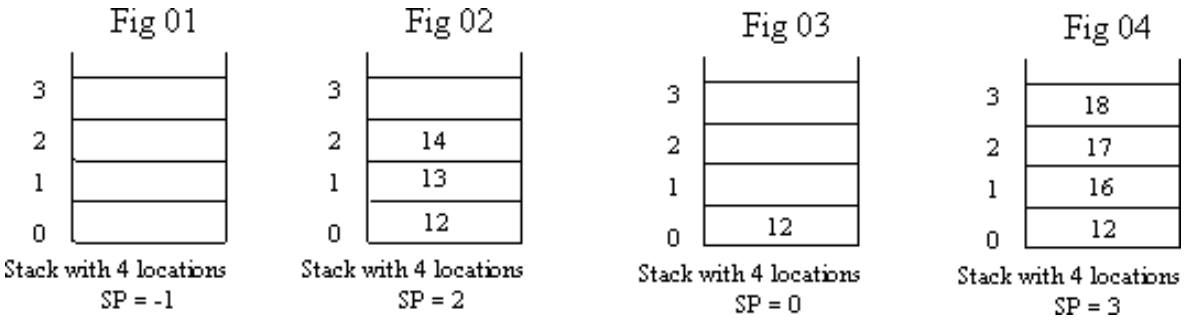


CONCEPT OF STACK:

Stack (ADT) Data Structure:

Stack is an Abstract data structure (ADT) works on the principle Last In First Out (LIFO). The last element add to the stack is the first element to be delete. Insertion and deletion can be takes place at oneend called TOP. It looks like one side closed tube.

- The add operation of the stack is called push operation
- The delete operation is called as pop operation.
- Push operation on a full stack causes stack overflow.
- Pop operation on an empty stack causes stack underflow.
- SP is a pointer, which is used to access the top element of the stack.
- If you push elements that are added at the top of the stack;
- In the same way when we pop the elements, the element at the top of the stack is deleted.



WHY STACK?

A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enoughspace to accept an entity to be pushed, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into underflow state, which means no items are present in stack to be removed.

OPERATIONS OF STACK:

There are two operations applied on stack they are

1. push
2. pop.

While performing push & pop operations the following test must be conducted on the stack.

1) Stack is empty or not

2) Stack is full or not

Push:

Push operation is used to add new elements in to the stack. At the time of addition first check the stack isfull or not. If the stack is full it generates an error message "stack overflow".

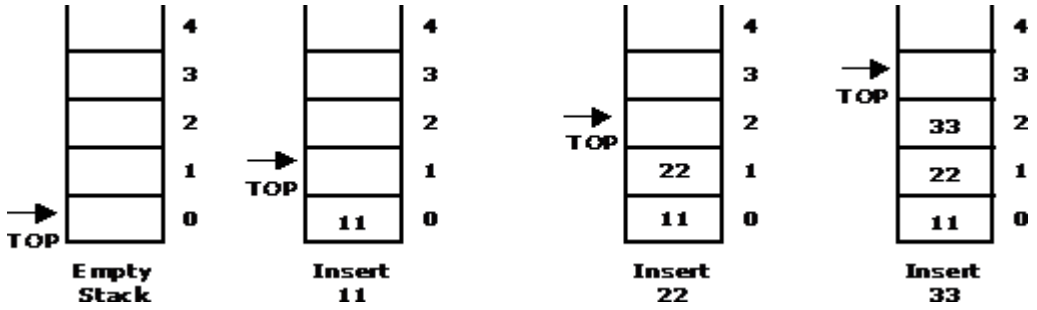
Pop:

Pop operation is used to delete elements from the stack. At the time of deletion first check the stack isempty or not. If the stack is empty it generates an error message "stack underflow".

Representation of a Stack using Arrays:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a stack overflow condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a stack underflow condition.

When a element is added to a stack, the operation is performed by push().



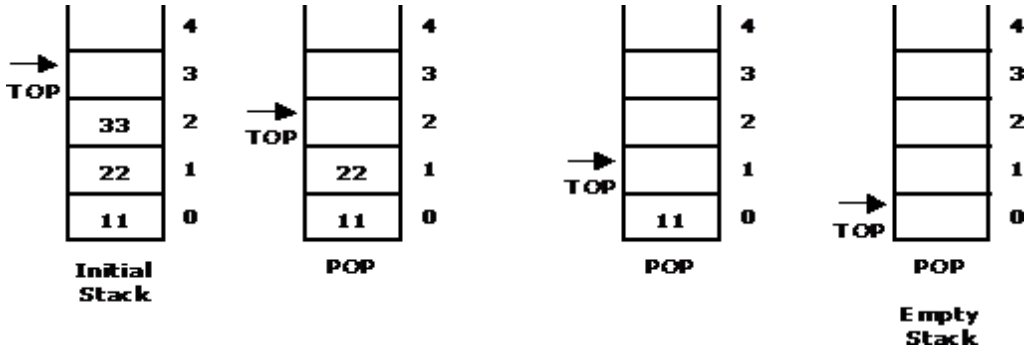
When an element is taken off from the stack, the operation is performed by pop().

STACK: Stack is a linear data structure which works under the principle of last in first out. Basic operations: push, pop, display.

1. **PUSH**: if (top==MAX), display **Stack overflow** else reading the data and making stack [top]
   =data and incrementing the top value by doing top++.

2. **POP:** if (top==0), display **Stack underflow** else printing the element at the top of the stack anddecrementing the top value by doing the top.

3. **DISPLAY**: IF (TOP==0), display **Stack is empty** else printing the elements in the stack fromstack [0] to stack [top].



Before implementing actual operations, first follow the below steps to create an empty stack.

- **Step 1 -** Include all the **header files** which are used in the program and define aconstant **'SIZE'** with specific value.
- **Step 2 -** Declare all the **functions** used in stack implementation.
- **Step 3 -** Create a one dimensional array with fixed size (**int stack[SIZE]**)
- **Step 4 -** Define a integer variable **'top'** and initialize with **'-1'**. (**int top = -1**)

- **Step 5 -** In main method, display menu with list of operations and make suitable functioncalls to perform operation selected by the user on the stack.

Push(value) - Inserting value into the stack:

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

- **Step 1 -** Check whether **stack** is **FULL**. (**top == SIZE-1**)
- **Step 2 -** If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** andterminate the function.
- **Step 3 -** If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] tovalue (**stack[top] = value**).

Push Routine:

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

Pop() - Delete a value from the Stack:

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use thefollowing steps to pop an element from the stack...

- **Step 1 -** Check whether **stack** is **EMPTY**. (**top == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** andterminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**
).

Pop Routine:

```
 void pop()
```

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

Display() - Displays the elements of a Stack:

We can use the following steps to display the elements of a stack...

- **Step 1 -** Check whether **stack** is **EMPTY**. (**top == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top.Display **stack[i]** value and decrement **i** value by one (**i--**).
- **Step 4 -** Repeat above step until **i** value becomes '0'.

Time Complexity:

operation	ArrayStack	
push	Append at index n O(1)*	Prepend at index 0 O(n)
peek	Get index n-1 O(1)	Get index 0 O(1)
pop	Delete index n-1 O(1)*	Delete index 0 O(n)

n: number of items, *: average

Stack Applications:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on returnthey are popped off.
6. Depth first search uses a stack data structure to find an element from agraph

Expression Parsing:

What is an Expression?

In any programming language, if we want to perform any calculation or to frame a condition etc., weuse a set of symbols to perform the task. These set of symbols makes an expression.

An expression is a collection of operators and operands that represents a specific value.

Operands are the values on which the operators can perform the task. Here operand can be a directvalue or variable or address of memory location.

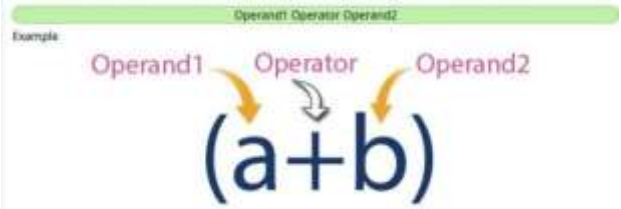
Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

- Infix Expression
- Postfix Expression
- Prefix Expression

Infix Expression

In infix expression, operator is used in between the operands.The general structure of an Infix expression is as follows..



Postfix Expression:

In postfix expression, operator is used after operands. We can say that "**Operator follows theOperands**".

The general structure of Postfix expression is as follows...

Prefix Expression:

In prefix expression, operator is used before operands. We can say that "**Operands follows theOperator**".

The general structure of Prefix expression is as follows...

Every expression can be represented using all the above three different types of expressions. And we can convert an expression from one form to another form like **Infix to Postfix**, **Infixto Prefix**, **Prefix to Postfix** and vice versa.

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

In-fix to Postfix Transformation:

Procedure:

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. a) If the scanned symbol is left parenthesis, push it onto the stack.

   b) If the scanned symbol is an operand, then place directly in the postfix expression (output).

   c) If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

   d) If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise,push the scanned operator onto the stack.

Example

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where **^** is an exponential operator.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '-'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Evaluating Arithmetic Expressions:

Procedure:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed ontothe stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack.

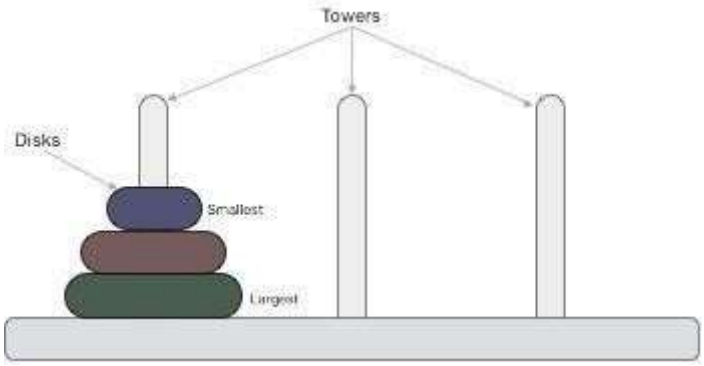
Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

Symbol	Operand 1	Operand 2	Value	Stack	Remarks
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols areplaced on the stack.
+	2	3	5	6, 5, 5	Next a „+" is read, so 3 and 2 are popped from the stack andtheir sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed

*	5	8	40	6, 5, 40	Now a „*" is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a „+" is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, „+" pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	**288**	Finally, a „*" is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

TOWER OF HANOI

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



These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are –

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

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks –

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

So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are –

Step 1 – Move n-1 disks from **source** to **aux**
Step 2 – Move n^{th} disk from **source** to **dest**
Step 3 – Move n-1 disks from **aux** to **dest**

A recursive algorithm for Tower of Hanoi can be driven as follows –

```
START
Procedure Hanoi(disk, source, dest, aux)

  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest) // Step 1
    move disk from source to dest // Step 2
    Hanoi(disk - 1, aux, dest, source) // Step 3
  END IF

END Procedure
STOP
```

QUEUE

What is Queue?

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

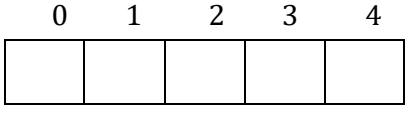
A real-world example of queue can be a single-lane one-way road, where the vehicle enters first,exits first. More real-world examples can be seen as queues at the ticket windowsand bus-stops.

Why Queue?

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

Queue representation using array:

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



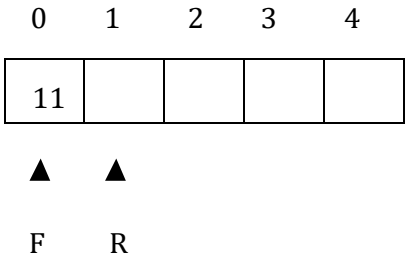
Q u e u e E mp t y

F RO NT = RE A R = 0

▲▲

F R

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

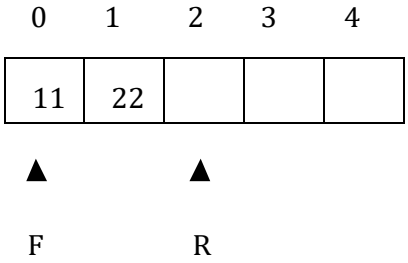


▲ ▲

F R

RE A R = RE A R + 1
= 1F RO NT = 0

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

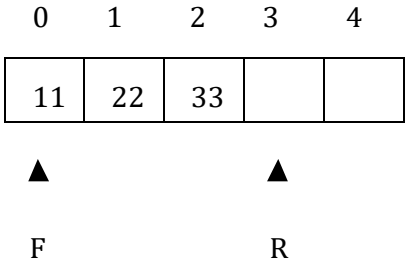


▲ ▲

F R

RE A R = RE A R + 1
= 2F RO NT = 0

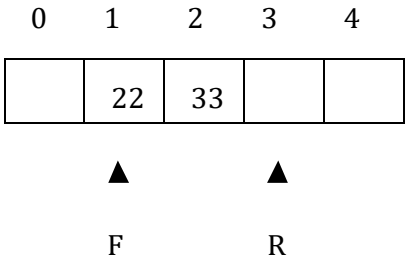
Again insert another element 33 to the queue. The status of the queue is:



RE A R = RE A R + 1
= 3

F RO NT = 0

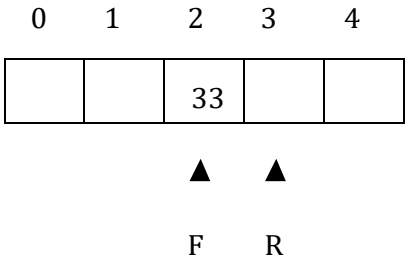
Now, delete an element. The element deleted is the element at the front of the queue. So the status of thequeue is:



RE A R = 3

F RO NT = F R O NT + 1 = 1

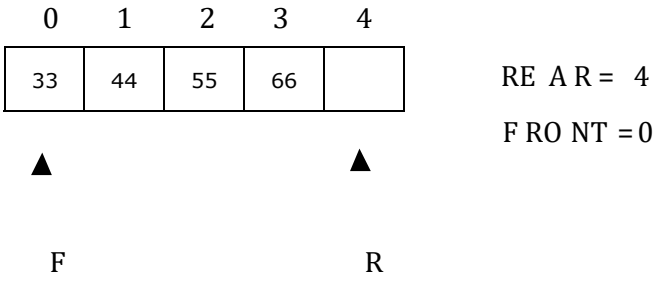
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22is deleted. The queue status is as follows:



RE A R = 3

F RO NT = F R O NT + 1 = 2

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



Basic Operations:

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek

   return queue[front]
```

end procedure

Implementation of peek() function in C programming language –

Example

```
int peek() {
   return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

Algorithm

```
 begin procedure isfull
   if rear equals to MAXSIZE
   return true
   else
   return false
   endif
end procedure
```

Example

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function –

Algorithm

```
begin procedure isempty
   if front is less than MIN OR front is greater than rear
      return true
   else
      return false
   endif
end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.Here's the C programming code –

Example

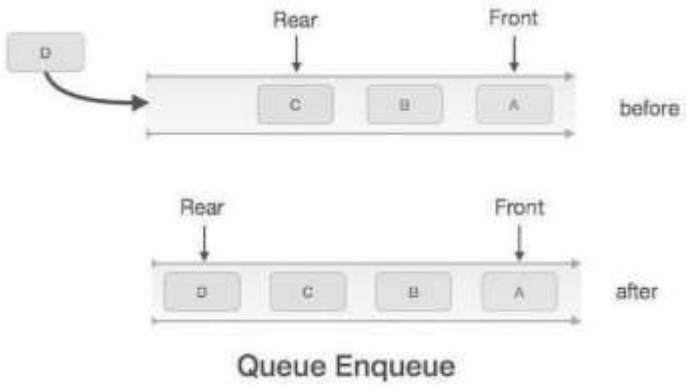
```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
     return false;}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparativelydifficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

- **Step 1** – Check if the queue is full.
- **Step 2** – If the queue is full, produce overflow error and exit.
- **Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)
   if queue is full
      return overflow
   endif
   rear ← rear + 1
   queue[rear] ← data
   return true
end procedure
```

Example

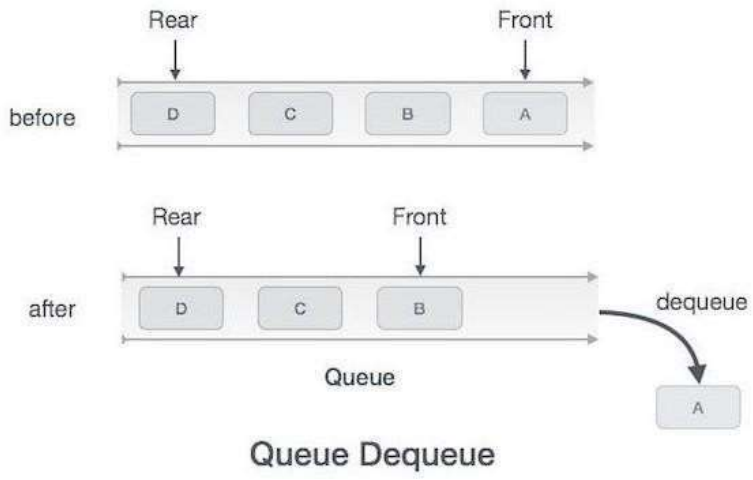
```
int enqueue(int data)if(isfull())
        return 0;
rear = rear + 1;
queue[rear] = data;
      return 1;
```

end procedure

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Queue Dequeue

Algorithm for dequeue operation

procedure dequeue

if queue is empty

return underflow

```
    end if
    data = queue[front]
    front ← front + 1
    return true
  end procedure
```

Example

```
  int dequeue() {
    if(isempty())
      return 0;
    int data = queue[front];
    front = front + 1;
    return data;
  }
```

Applications of Queue:

- Printer Spooling
- CPU Scheduling
- Mail Service
- Keyboard Buffering

LINKED LISTS

What is Linked lists?

Linked lists and arrays are similar since they both store collections of data. Array is the most commondata structure used to store collections of elements. Arrays are convenient to declare and provide the easy syntax to access any element by its index number. Once the array is set up, access to any element is convenient and fast.

Why Linked Lists?

1. Linked lists provide flexibility in inserting a dataitem at a specified position and deletion of the data item from the given position.
2. Many complex applications can be easily carried out with linked lists.

Linked List Concepts:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list. The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

1.Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of aprogram.
2.Linked lists have efficient memory utilization. Here, memory is not preallocated.
3. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
4. Insertion and Deletions are easier and efficient.

Disadvantages of linked lists:

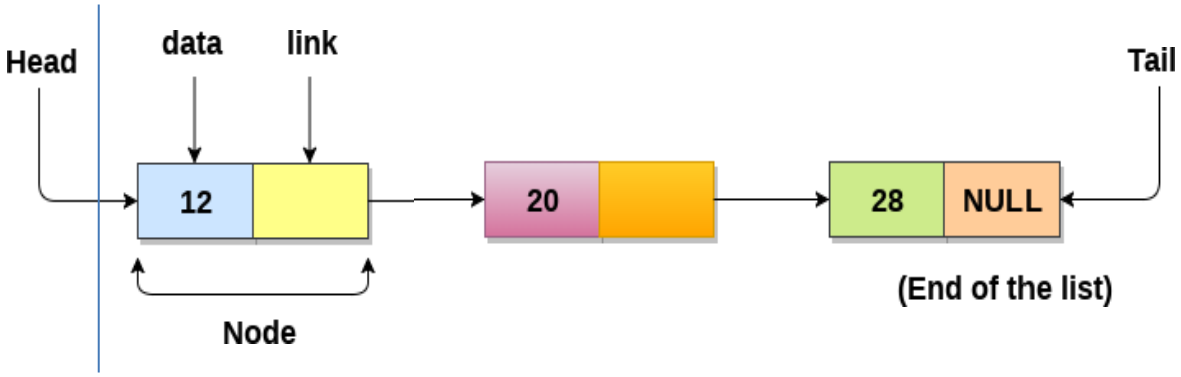
1. It consumes more space because every node requires a additional pointer to store address of thenext node.
2. Searching a particular element in list is difficult and also time consuming.

Single Linked List:

Single linked list is a sequence of elements in which every element has link to its next elementin the sequence.In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field. The data field is used to stor actual value of the node and next field is used to st re the address of next node in the sequence.

The graphical representation of a node in a single linked list is as follows...

Each node is allocated in the heap using malloc(), so the node memory continues to existuntil it is explicitly de-allocated using free().



- The beginning of the linked list is stored in a "**head**" pointer which points to the first node.

- The first node contains a pointer to the second node.
- The second node contains a pointer to the third node, ... and so on.
- The last node in the list has its next field set to NULL to mark the end of the list.
- Code can access any node in the list by starting at the **head** and following the next pointers.

The **head** pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

Operations on Single Linked List

Create function to create the linked list.void

```
create()

{

    struct node *temp,*ptr;

    printf("nEnter the data value for the node:t");

    scanf("%d",&temp->info);

    temp->next=NULL;

    if(start==NULL)

    {
        start=temp;
    }
    else
    {
        ptr=start;
        while(ptr->next!=NULL)
        {
            ptr=ptr->next;

            ptr->next=temp;

        }
     }
```

The following operations are performed on a Single Linked List

- Insertion
- Deletion
- Display

Before we implement actual operations, first we need to set up an empty list. First, perform the

following steps before implementing actual operations.

Step 1 - Include all the **header files** which are used in the program.
Step 2 - Declare all the **user defined functions**.
Step 3 - Define a **Node** structure with two members **data** and **next**
Step 4 - Define a Node pointer **'head'** and set it to **NULL**.
Step 5 - Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

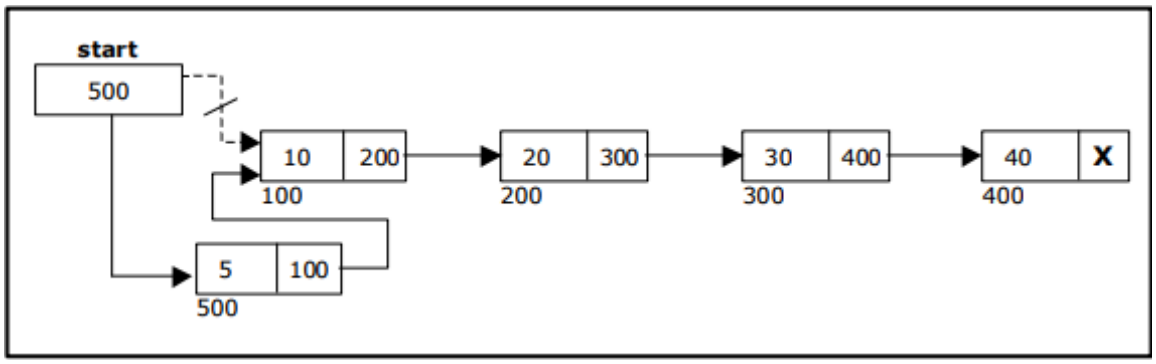
In a single linked list, the insertion operation can be performed in three ways. They are as follows.

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list.

Step 1 - Create a **newNode** with given value.
Step 2 - Check whether list is **Empty** (**head** == **NULL**)
Step 3 - If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.
Step 4 - If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

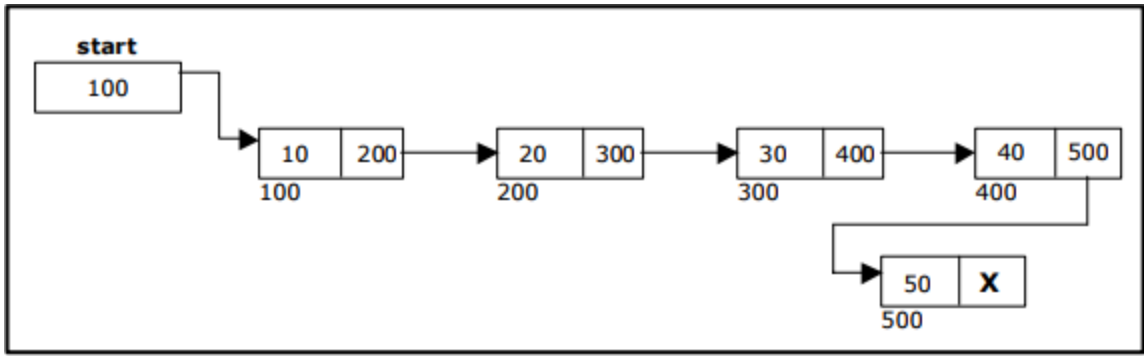


Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list.

Step 1 - Create a **newNode** with given value and **newNode → next** as **NULL**.
Step 2 - Check whether list is **Empty** (**head** == **NULL**).
Step 3 - If it is **Empty** then, set **head** = **newNode**.
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list(until **temp → next** is equal to **NULL**).

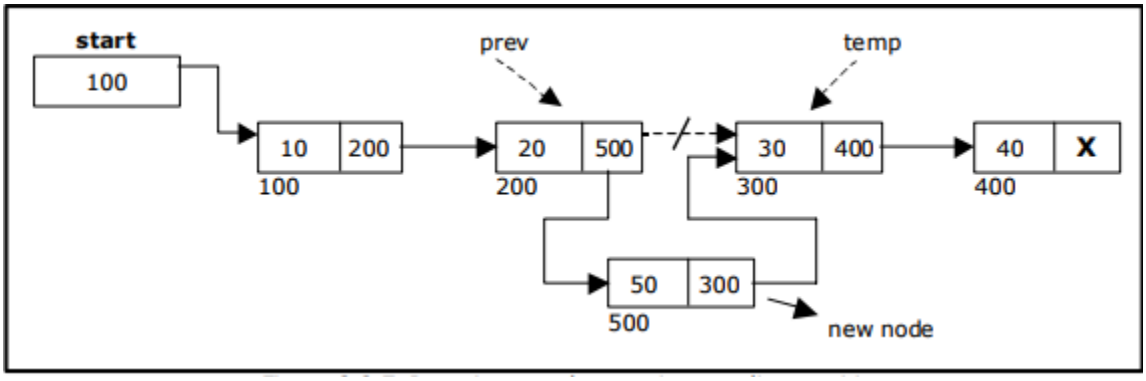
Step 6 - Set temp → next = newNode.



Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

Step 1 - Create a **newNode** with given value.
Step 2 - Check whether list is **Empty** (**head** == **NULL**)
Step 3 - If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
Step 6 - Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
Step 7 - Finally, Set '**newNode → next** = **temp → next**' and '**temp → next** = **newNode**'



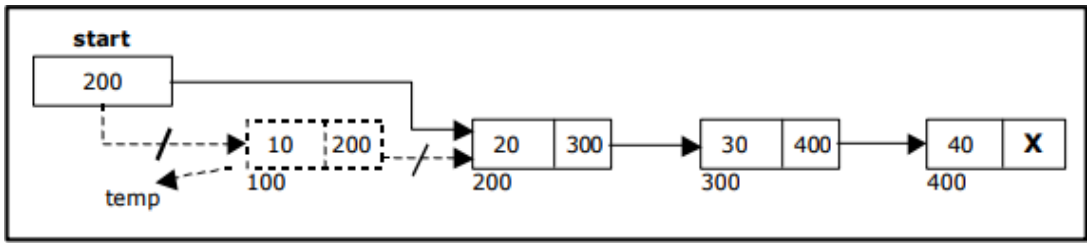
Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

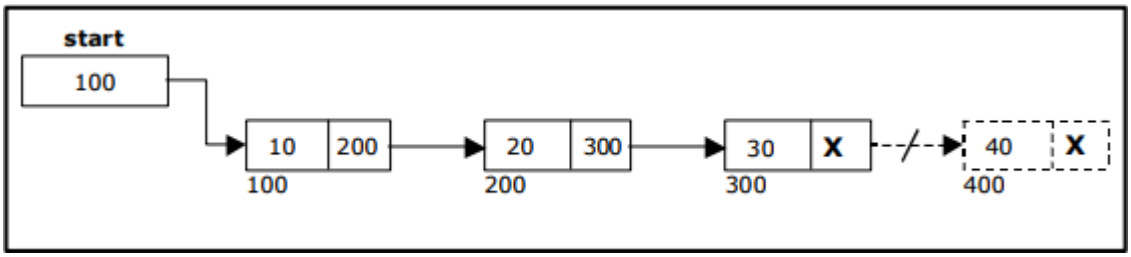
We can use the following steps to delete a node from beginning of the single linked list...
Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** andterminate the function.
Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
Step 4 - Check whether list is having only one node (**temp → next** == **NULL**)
Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
Step 6 - If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.



Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list.

Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** andterminate the function.
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialize **'temp1'** with **head**.
Step 4 - Check whether list has only one Node (**temp1 → next** == **NULL**)
Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate thefunction. (Setting **Empty** list condition)
Step 6 - If it is **FALSE**. Then, set **'temp2 = temp1** ' and move **temp1** to its next node. Repeatthe same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)
Step 7 - Finally, Set **temp2 → next** = **NULL** and delete **temp1**.



Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list.
Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** andterminate the

function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize'**temp1**' with **head**.

Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the lastnode. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

Step 5 - If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.

Step 6 - If it is reached to the exact node which we want to delete, then check whether list ishaving only one node or not

Step 7 - If list has only one node and that is the node to be deleted, thenset **head** = **NULL** and delete **temp1** (**free(temp1)**).

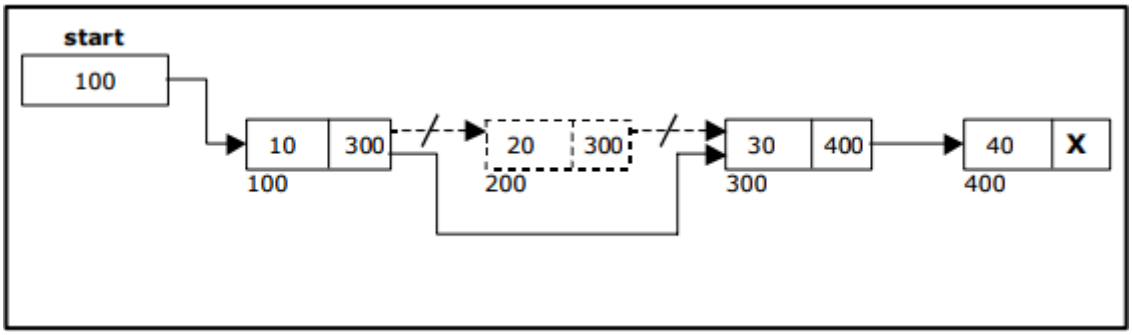
Step 8 - If list contains multiple nodes, then check whether **temp1** is the first node in the list(**temp1 == head**).

Step 9 - If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.

Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1 →** next == NULL).

Step 11 - If **temp1** is last node then set **temp2 →next** = **NULL** anddelete **temp1** (**free(temp1)**).

Step 12 - If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).



Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list.

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the lastnode

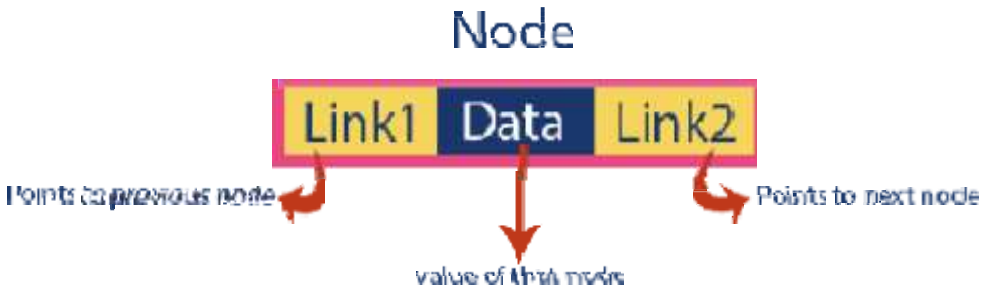
Step 5 - Finally display **temp → data** with arrow pointing to **NULL** (**temp → data --->NULL**).

Double Linked List:

In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we cannot traverse back. We can solve this kind of problem by using a double linked list.

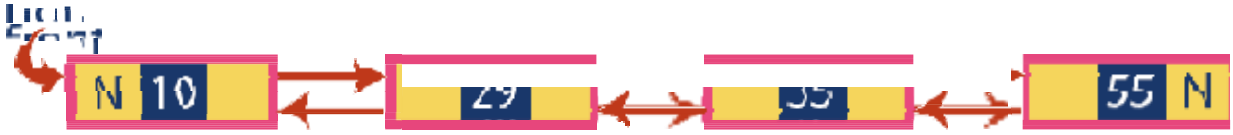
Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence**.**

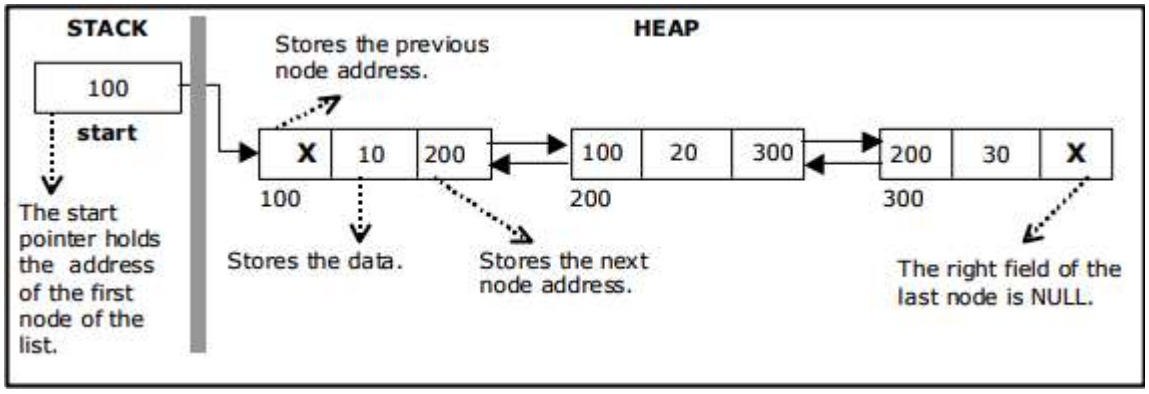
In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure.



Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Example





Operations on Double Linked List

In a double linked list,

1. Insertion
2. Deletion
3. Display

Insertion

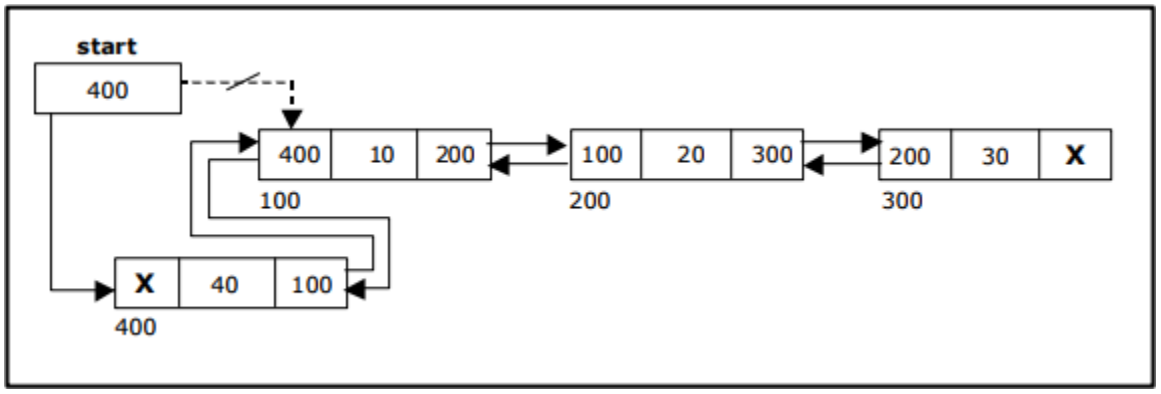
In a double linked list, the insertion operation can be performed in three ways as follows.

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list.

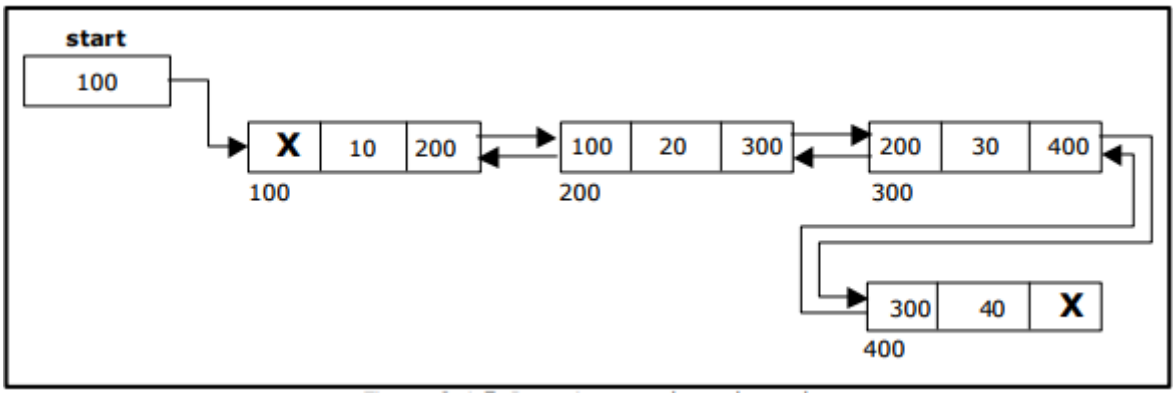
Step 1 - Create a **newNode** with given value and **newNode → previous** as **NULL**.
Step 2 - Check whether list is **Empty** (**head** == **NULL**)
Step 3 - If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
Step 4 - If it is **not Empty** then, assign **head** to **newNode → next** and **newNode** to **head**.



Inserting At End of the list

We can use the following steps to insert a new node at end of the double linked list...

Step 1 - Create a **newNode** with given value and **newNode → next** as **NULL**.
Step 2 - Check whether list is **Empty** (**head** == **NULL**)
Step 3 - If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **head**.
Step 4 - If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list(until **temp → next** is equal to **NULL**).
Step 6 - Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

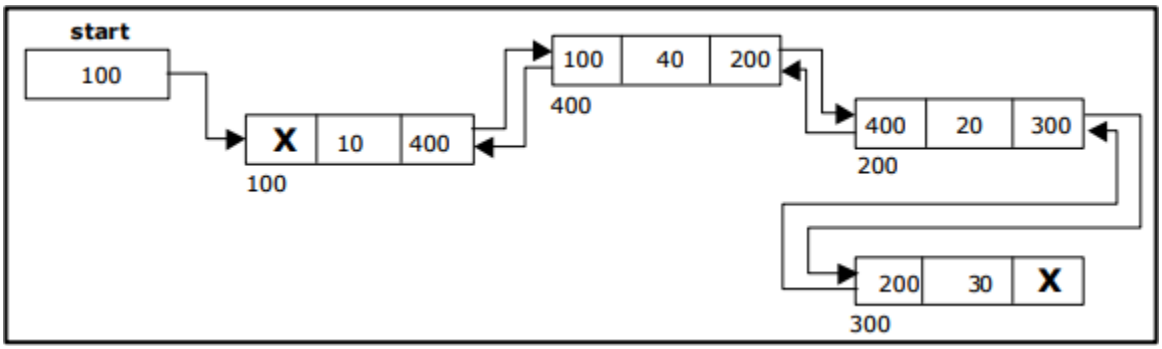


Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the double linked list...

Step 1 - Create a **newNode** with given value.
Step 2 - Check whether list is **Empty** (**head** == **NULL**)
Step 3 - If it is **Empty** then, assign **NULL** to both **newNode → previous** & **newNode → next** and set **newNode** to **head**.

Step 4 - If it is **not Empty** then, define two node pointers **temp1** & **temp2** and initialize **temp1** with **head**.
Step 5 - Keep moving the **temp1** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
Step 6 - Every time check whether **temp1** is reached to the last node. If it is reached to the last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp1** to next node.
Step 7 - Assign **temp1 → next** to **temp2**, **newNode** to **temp1 → next**, **temp1** to **newNode → previous**, temp2 to newNode → next and newNode to temp2 → previous.



Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

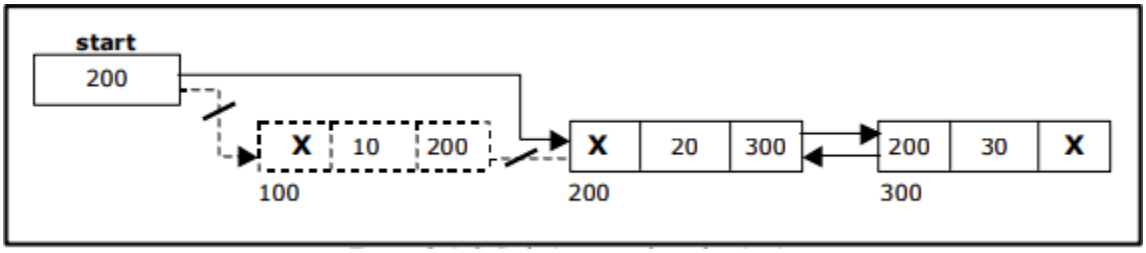
1. Deleting from Beginning of the list

2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the double linked list...

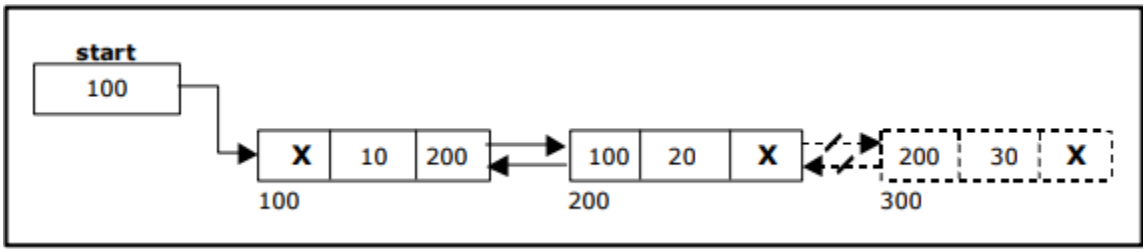
Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** andterminate the function.
Step 3 - If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
Step 4 - Check whether list is having only one node (**temp → previous** is equal to **temp →** next)
Step 5 - If it is **TRUE**, then set **head** to **NULL** and delete **temp** (Setting **Empty** listconditions)
Step 6 - If it is **FALSE**, then assign **temp → next** to **head**, **NULL** to **head → previous** anddelete **temp**.



Deleting from End of the list

We can use the following steps to delete a node from end of the double linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** andterminate the function.
Step 3 - If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
Step 4 - Check whether list has only one Node (**temp → previous** and **temp → next** bothare **NULL**)
Step 5 - If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from thefunction. (Setting **Empty** list condition)
Step 6 - If it is **FALSE**, then keep moving **temp** until it reaches to the last node in the list.(until **temp → next** is equal to **NULL**)
Step 7 - Assign **NULL** to **temp → previous → next** and delete **temp**.



Deleting a Specific Node from the list

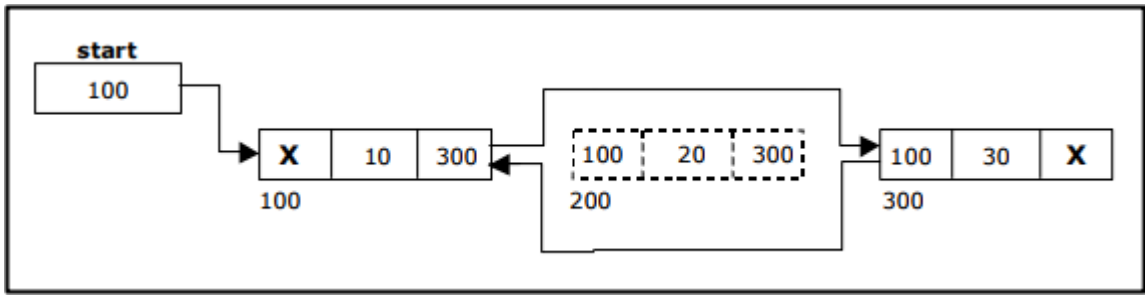
We can use the following steps to delete a specific node from the double linked list...

Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** andterminate the function.

Step 3 - If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
Step 4 - Keep moving the **temp** until it reaches to the exact node to be deleted or to the lastnode.
Step 5 - If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the fuction.
Step 6 - If it is reached to the exact node which we want to delete, then check whether list ishaving only one node or not
Step 7 - If list has only one node and that is the node which is to be deleted thenset **head** to **NULL** and delete **temp** (**free(temp)**).
Step 8 - If list contains multiple nodes, then check whether **temp** is the first node in the list(**temp == head**).
Step 9 - If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **NULL** (**head → previous = NULL**) and delete **temp**.

Step 10 - If **temp** is not the first node, then check whether it is the last node in the list (**temp** → next == NULL).
Step 11 - If **temp** is the last node then set **temp** of **previous** of **next** to **NULL** (**temp → previous → next = NULL**) and delete **temp** (**free(temp)**).

Step 12 - If **temp** is not the first node and not the last node, thenset **temp** of **previous**of **next** to **temp** of **next** (**temp → previous → next = temp →next**), **temp** of **next** of **previous** to **temp**of **previous** (**temp → next → previous = temp → previous**) and delete **temp** (**free(temp)**).



Displaying a Double Linked List

We can use the following steps to display the elements of a double linked list.
Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
Step 3 - If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
Step 4 - Display **'NULL <--- '**.
Step 5 - Keep displaying **temp → data** with an arrow (**<===>**) until **temp** reaches to the lastnode
Step 6 - Finally, display **temp → data** with arrow pointing to **NULL** (**temp → data --->NULL**).

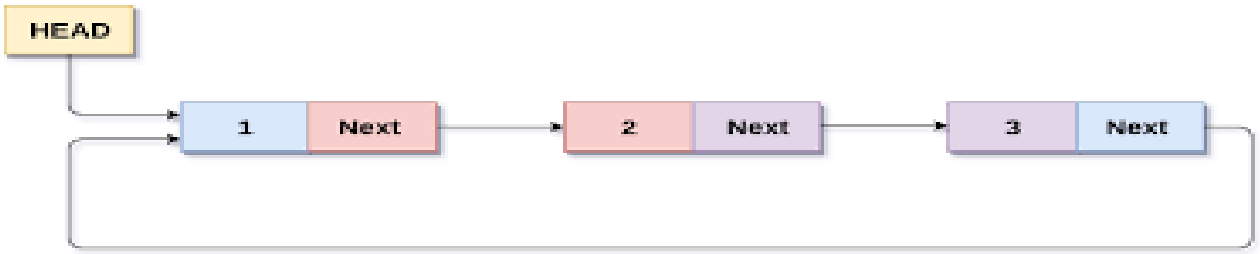
Circular Linked List

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

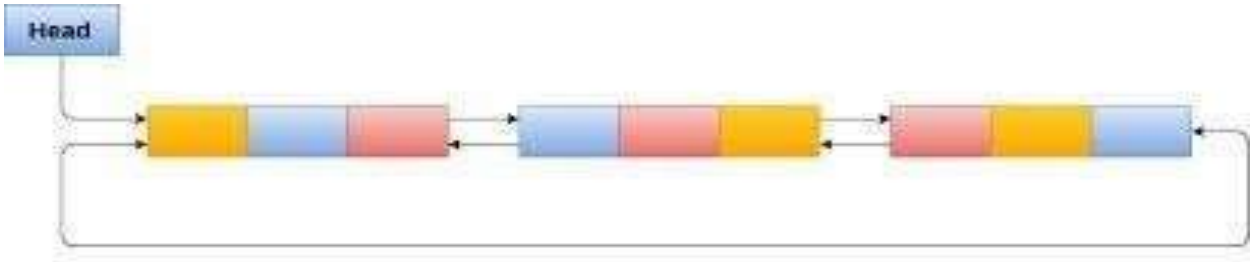
A Circular Linked list is a sequence of elements in which every element has a link to its next

element in the sequence and the last element has a link to the first element. That means

circular linked list is similar to the single linked list except that the last node points to the first nodein the list.



Circular Singly Linked List

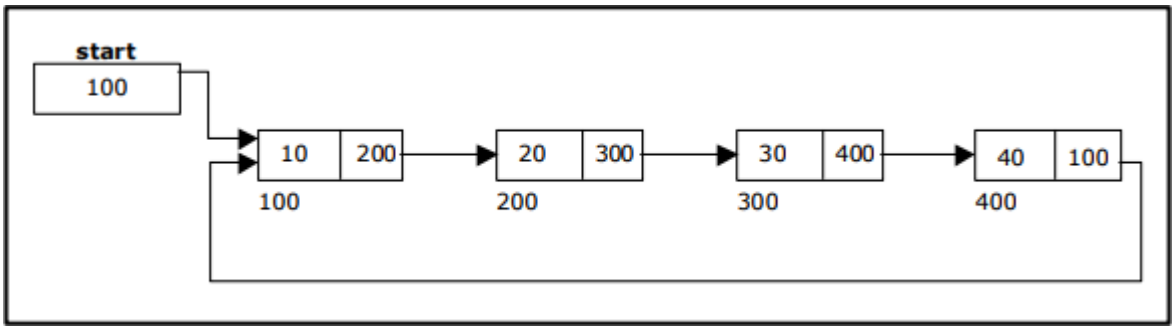


Circular Doubly Linked List

Circular Singly Linked List:

It is just a singly linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called start pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

Creating a circular single Linked List with „n" number of nodes:



The basic operations in a circular single linked list are:

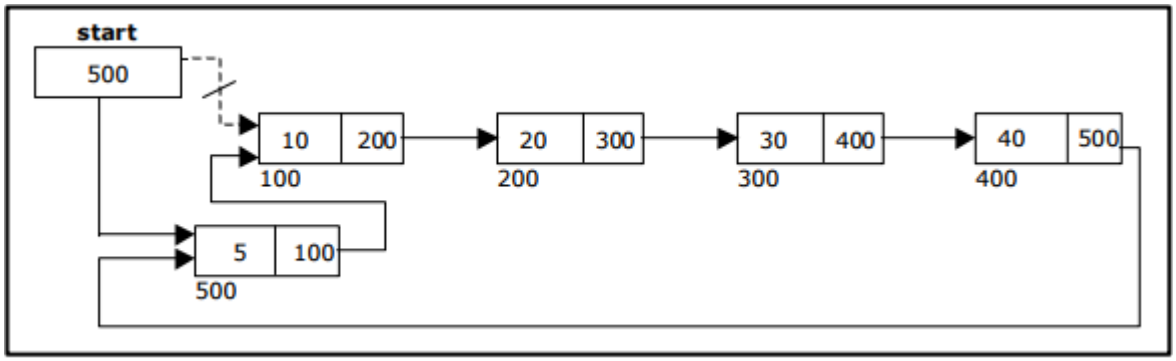
- Creation.
- Insertion.

- Deletion.
- Traversing.

Inserting a node at the beginning:

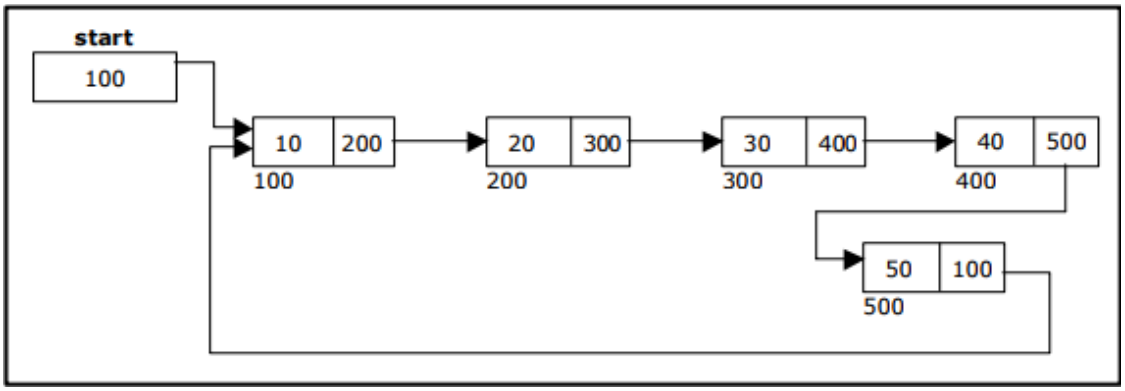
Step 1 - Create a **newNode** with given value.
Step 2 - Check whether list is **Empty** (**head** == **NULL**)
Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head** .
Step 4 - If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
Step 5 - Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next** == **head**').

Step 6 - Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.



Inserting a node at the end:

Step 1 - Create a **newNode** with given value.
Step 2 - Check whether list is **Empty** (**head** == **NULL**).
Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list(until **temp → next** == **head**).
Step 6 - Set **temp → next** = **newNode** and **newNode → next** = **head**.



Inserting At Specific location in the list (After a Node)

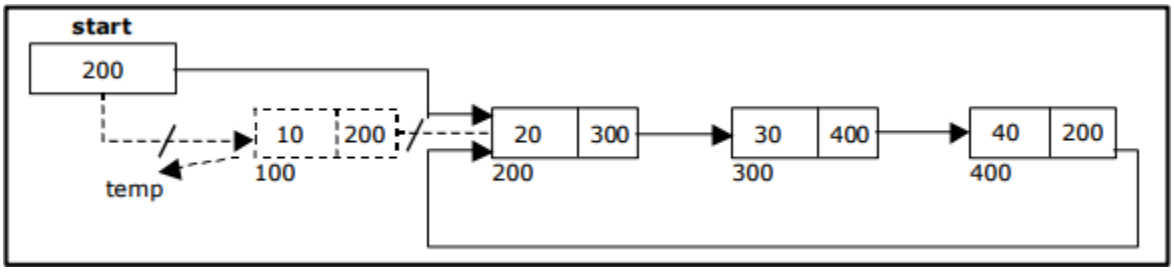
We can use the following steps to insert a new node after a node in the circular linked list...

Step 1 - Create a **newNode** with given value.
Step 2 - Check whether list is **Empty** (**head** == **NULL**)
Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
Step 6 - Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
Step 7 - If **temp** is reached to the exact node after which we want to insert the newNode thencheck whether it is last node (temp → next == head).
Step 8 - If **temp** is last node then set **temp → next** = **newNode** and **newNode →next** = **head**.
Step 9 - If **temp** is not last node then set **newNode → next** = **temp → next** and **temp →** next = newNode.

Deleting a node at the beginning:

Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** andterminate the function.
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initializeboth **'temp1'** and **'temp2'** with **head**.
Step 4 - Check whether list is having only one node (**temp1 → next** == **head**)
Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
Step 6 - If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next** == **head**)

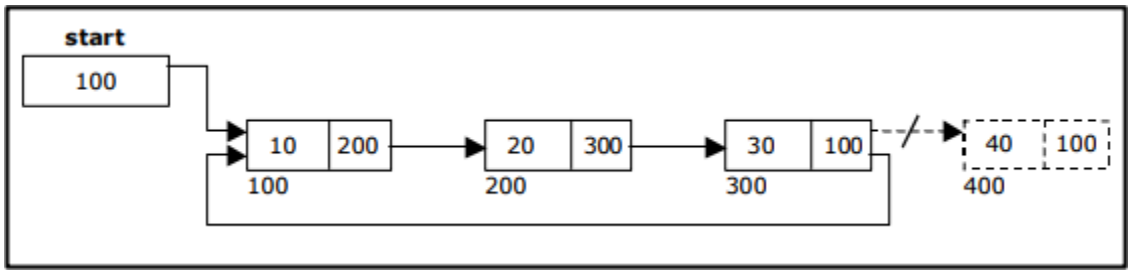
Step 7 - Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.



Deleting a node at the end:

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** andterminate the function.
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialize **'temp1'** with **head**.
Step 4 - Check whether list has only one Node (**temp1 → next == head**)
Step 5 - If it is **TRUE**. Then, set **head = NULL** and delete **temp1**. And terminate from thefunction. (Setting **Empty** list condition)
Step 6 - If it is **FALSE**. Then, set **'temp2 = temp1 '** and move **temp1** to its next node. Repeatthe same until **temp1** reaches to the last node in the list. (until **temp1 → next == head**)
Step 7 - Set **temp2 → next = head** and delete **temp1**.

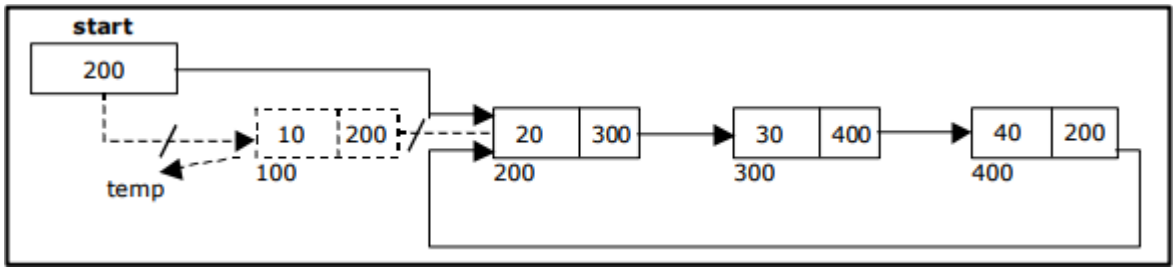


Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

Step 1 - Check whether list is **Empty** (**head == NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** andterminate the function.
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialize**'temp1'** with **head**.
Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the lastnode. And every time set **'temp2 = temp1'** before moving the **'temp1'** to its next node.
Step 5 - If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
Step 6 - If it is reached to the exact node which we want to delete, then check whether list ishaving only one node (**temp1 → next == head**)
Step 7 - If list has only one node and that is the node to be deleted thenset **head = NULL** and delete **temp1** (**free(temp1)**).
Step 8 - If list contains multiple nodes then check whether **temp1** is the first node in the list(**temp1 == head**).
Step 9 - If **temp1** is the first node then set **temp2 = head** and keep moving **temp2** to its nextnode until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → next = head** and delete **temp1**.

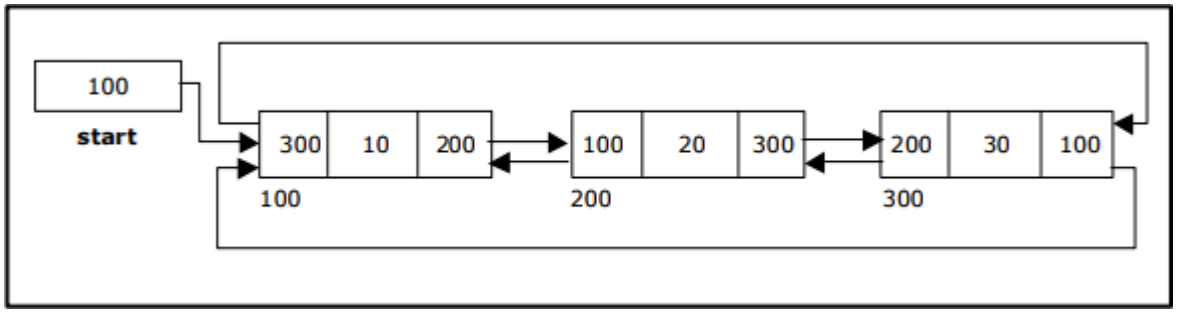
Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
Step 11 - If **temp1** is last node then set **temp2 → next = head** and delete **temp1** (**free(temp1)**).
Step 12 - If **temp1** is not first node and not last node then set **temp2 → next = temp1 → next** and delete **temp1** (**free(temp1)**).



Circular Double Linked List:

A circular double linked list has both successor pointer and predecessor pointer in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operationsperformed on double linked list. In circular double linked list the right link of the right most node points back to the start node and left link of the first node points to the last node.

A circular double linked list is shown in figure

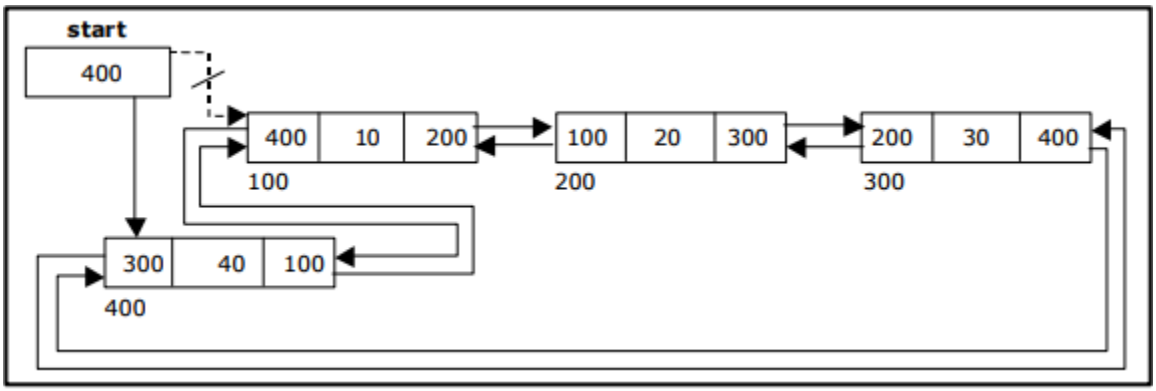


Creating a Circular Double Linked List with „n" number of nodes

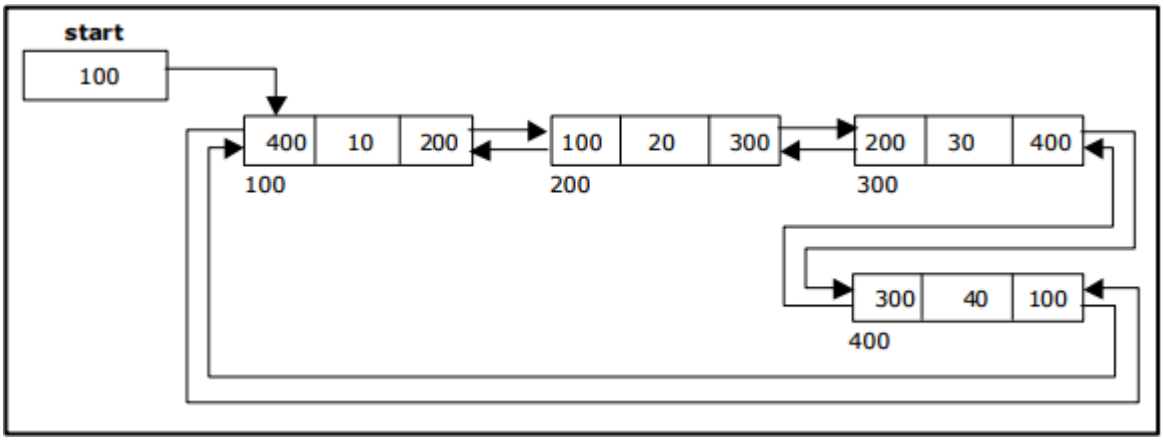
The basic operations in a circular double linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

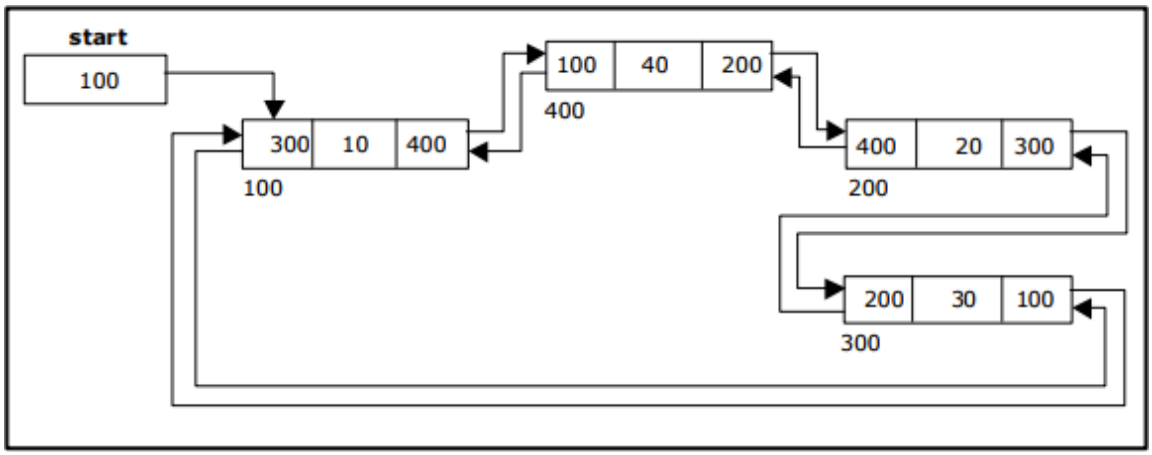
Inserting a node at the beginning:



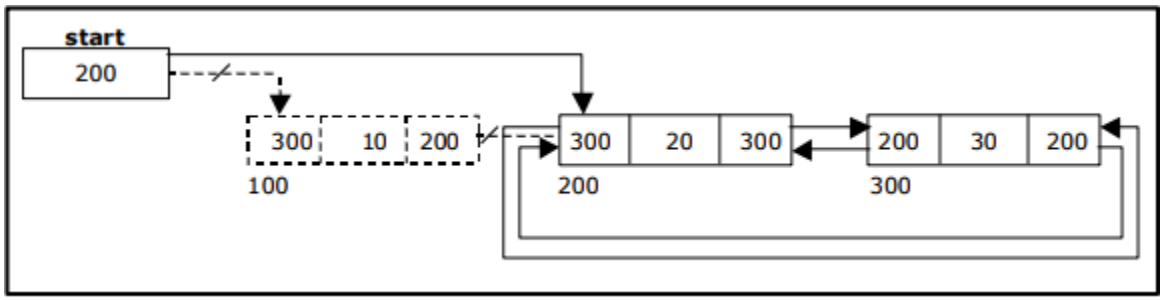
Inserting a node at the end:



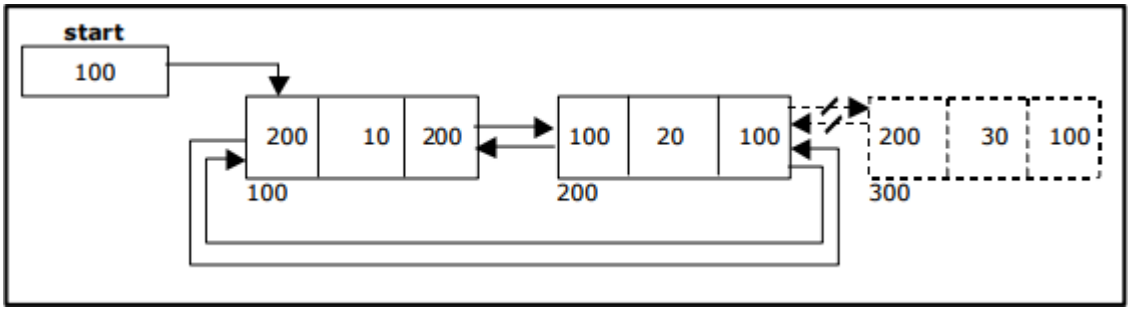
Inserting a node at an intermediate position:



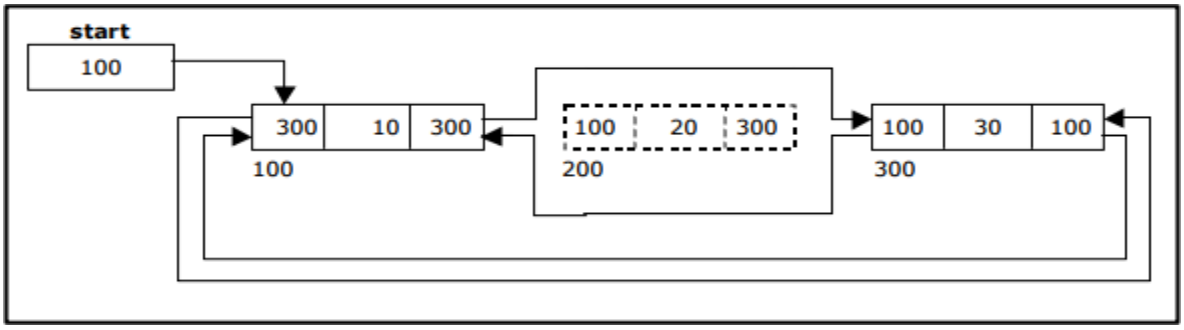
Deleting a node at the beginning:



Deleting a node at the end:



Deleting a node at Intermediate position:



Merits and demerits of Linked List

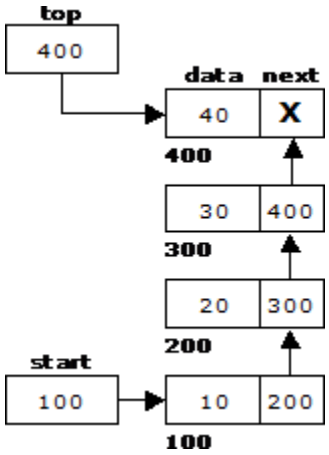
1. The major disadvantage of doubly linked lists (over singly linked lists) is that they require more space (every node has two pointer fields instead of one). Also, the code to manipulate doubly linked lists needs to maintain the prev fields as well as the next fields; the more fields that have to be maintained, the more chance there is for errors.

2. The major advantage of doubly linked lists is that they make some operations (like the removal of a givennode, or a right-to-left traversal of the list) more efficient.

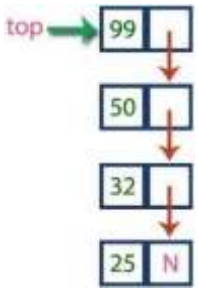
3. The major advantage of circular lists (over non-circular lists) is that they eliminate some extra-case code for some operations (like deleting last node). Also, some applications lead naturally to circular list representations.

Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one endcalled top. We can perform similar operations at one end of list using top pointer.



In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.



In the above example, the last inserted node is 99 and the first inserted node is 25. The order ofelements inserted is 25, 32,50 and 99.

Stack Operations using Linked List

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.

- **Step 3 -** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack.

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set **newNode → next** = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set **newNode → next** = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
- **Step 4 -** Then set '**top** = **top → next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- **Step 1 -** Check whether stack is **Empty** (**top** == **NULL**).
- **Step 2 -** If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the sameuntil **temp** reaches to the first node in the stack. (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

Routine for stack creation

```
STACK create_stack( void )

{
```

```
STACK S;
S = (STACK) malloc( sizeof( struct node ) );if(
S == NULL )
fatal_error("Out of space!!!");
return S;
}
```

Routine for Make Null

```
void make_null( STACK S )
{
if( S != NULL )
S->next = NULL;
else
error("Must use create_stack first");
}
```

Routine for Push

```
void push( element_type x, STACK S )
{
node_ptr tmp_cell;
tmp_cell = (node_ptr) malloc( sizeof ( struct node ) );if(
tmp_cell == NULL )
fatal_error("Out of space!!!");
else
{
tmp_cell->element = x;
tmp_cell->next = S->next;
S->next = tmp_cell;
```

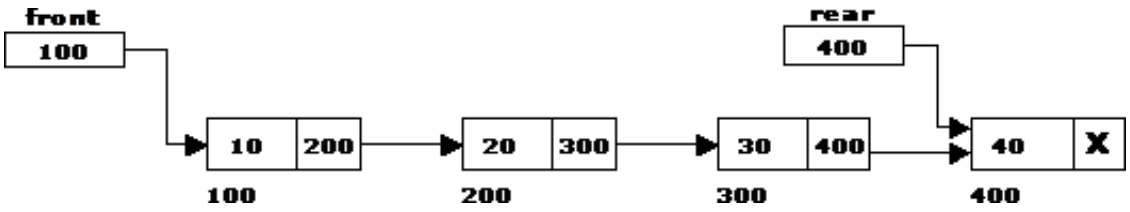
```
}

}

element_type top( STACK S )

{

if( is_empty( S ) )

error("Empty stack");

else

return S->next->element;

}
```

Routine for Pop

```
void pop( STACK S )

{

node_ptr first_cell; if(

is_empty( S ) )

error("Empty stack");

else

{

first_cell = S->next;

S->next = S->next->next;

free( first_cell );

}

}
```

Linked List Implementation of Queue:

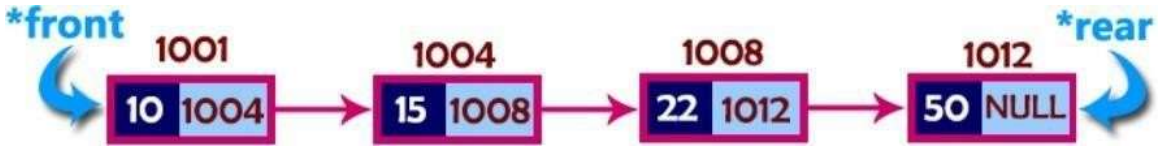
We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers front and rear for our linked queue implementation.



The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queueusing linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **userdefined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu of list of operations and makesuitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue.

- **Step 1 -** Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
- **Step 2 -** Check whether queue is **Empty** (**rear** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **rear → next** =

newNode and **rear** = **newNode**.**deQueue() - Deleting an**

Element from Queue

We can use the following steps to delete a node from the queue.

- **Step 1 -** Check whether **queue** is **Empty** (**front == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** andterminate from the function
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- **Step 4 -** Then set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

Step 1 - Check whether queue is **Empty** (**front** == **NULL**).

Step 2 - If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.

Step 4 - Display '**temp → data** --->' and move it to the next node. Repeat the same until'**temp**' reaches to '**rear**' (**temp → next** != **NULL**).

Step 5 - Finally! Display '**temp → data** ---> **NULL**'.

Applications of Linked List

- Implementation of stacks and queues.
- Implementation of graphs : Adjacency list representation of graphs is most popular which is uses
  linked list to store adjacent vertices.
- Dynamic memory allocation : We use linked list of free blocks.
- Maintaining directory of names.
- Performing arithmetic operations on long integers.
- Manipulation of polynomials by storing constants in the node of linked list.
- Representing sparse matrix

UNIT - II
TREES

Introduction to Trees –Definitions and concepts, Representation of Binary Tree, Types of Binary tree, properties, structure and applications of binary tree, Binary Tree Traversal with Recursive and Non-Recursive- Expression Tree, Binary Search Trees– Definition, PropertiesandOperationsofbinarysearch tree.Priority Queue: Operations, Implementations- Heaps: properties and types of Heaps, Binary Heap: Representation, Declaration and Creation of Heap, Heapifying, Inserting and Deleting an element, Destroying Heap, Heap Sort. Case Study: Priority Queue in bandwidth management

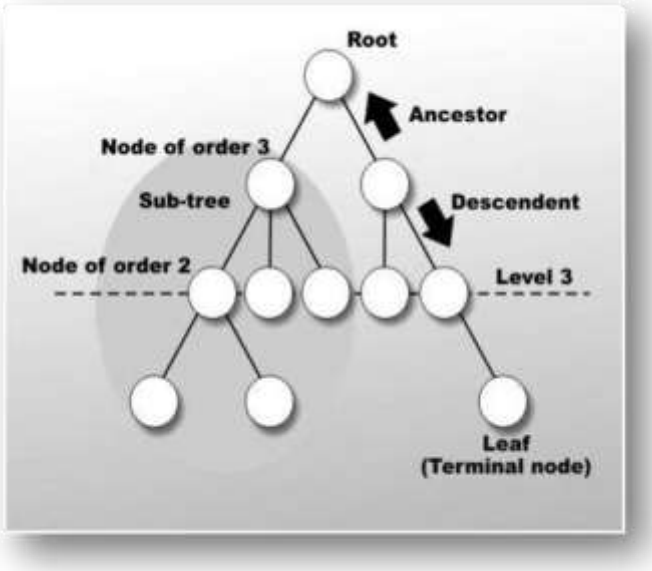
Trees Basic Concepts:

A **tree** is a non-empty set one element of which is designated the root of the tree while the remaining elements are partitioned into non-empty sets each of which is a sub-tree of the root.

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies thefollowing

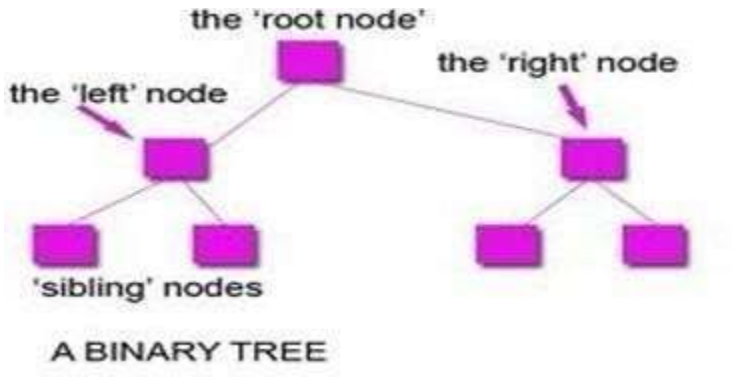
- If T is not empty, T has a special tree called the root that has noparent.
- Each node v of T different than the root has a unique parent node w; each node with parent w isa childofw.

Tree nodes havemany useful properties. The **depth** of a node is the length of the path (or the number of edges) from the root to that node. The **height** of a node is the longest path from that node to its leaves. The height of a tree is the height of the root. A **leaf node** has no children -- its only path is up to its parent.



Binary Tree:

In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of anodecalledthe **root** together withtwo binarytreescalledthe **left subtree**and the **right subtree**.



Tree Terminology:

Leaf node

A node with no children is called a leaf (or external node). A node which is not a leaf is called an internal node.

Path: A sequence of nodes $n_1, n_2, \ldots, n_k$, such that n_i is the parent of n_{i+1} for i = 1, 2,. . ., k - 1. The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself.

Siblings: The children of the same parent are called siblings.

Ancestor and Descendent If there is a path from node A to node B, then A is called an ancestor of B and *B* is called a descendent of A.

Subtree: Any node of a tree, with all of its descendants is a subtree.

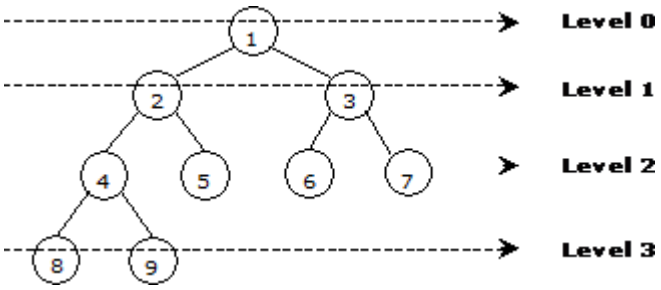
Level: The level of the node refers to its distance from the root. The root of the tree has level O, and the level of any other node in the tree is one more than the level of its parent.

The maximum number of nodes at any level is 2^n.

Height:The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest pathfromthe nodeto aleaf.Thetermdepthisalsousedto denote heightof thetree.

Depth: Thedepthofanodeisthenumber ofnodesalongthepathfromtheroottothatnode.

Assigninglevelnumbers and Numberingofnodes fora binarytree: Thenodesofabinarytree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent.



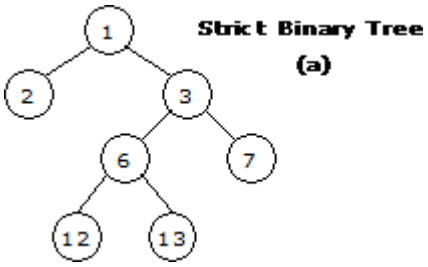
Properties of Binary Trees:

Some of the important properties of a binary tree are as follows:

1. If *h* = height of a binary tree,then
    a. Maximum number of leaves=2^h
    b. Maximumnumberof nodes=2^{h+1}-1
2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l +1.
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l node at levell.
4. The total number of edges in a full binary tree with n node is n -1.

Strictly Binarytree:

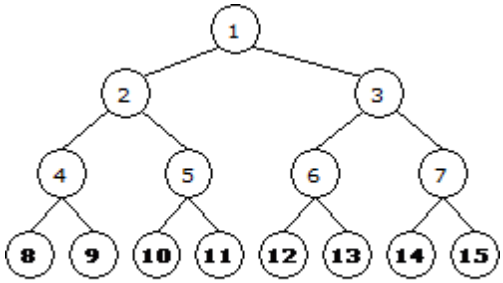
If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a strictly binary tree. Thus the tree of figure 7.2.3(a) is strictly binary. A strictly binary tree with n leaves always contains 2n - 1nodes.



Full Binary Tree:

A full binary tree of height h has all its leaves at level h. Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

Afullbinarytreewithheight*h* has2^{h+1}-1 nodes. Afullbinarytree ofheighth is*a strictlybinary tree*all of whose leaves are at levelh.



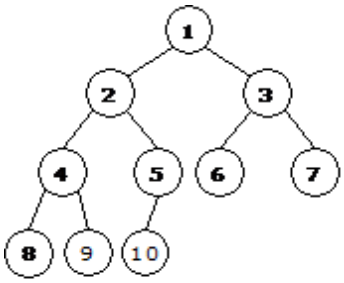
Full binary tree (d)

For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

Complete Binary Tree:

A binary tree with *n* nodes is said to be **complete** if it contains all the first *n* nodes of the above numbering scheme.

A complete binary tree of height h looks like a full binary tree down to level h-1, and the level h is filled from left to right.



Complete binary tree (c)

Perfect Binary Tree:

A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.
Following are examples of Perfect Binary Trees.

18

/ \

15 30

/ \ /\

40 50 10040

18

/ \

15 30

A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has $2^h - 1$ node.

Example of Perfect binary tree is ancestors in family. Keep a person at root, parents as children, parents of parents as their children.

Balanced Binary Tree:

A binary tree is balanced if height of the tree is O(Log n) where n is number of nodes. For Example, AVL tree maintain O(Log n) height by making sure that the difference between heights of left and rightsubtrees is 1. Red-Blacktrees maintain O(Log n) height bymakingsurethatthenumber of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide O(log n) time for search, insert anddelete.

Representation of Binary Trees:

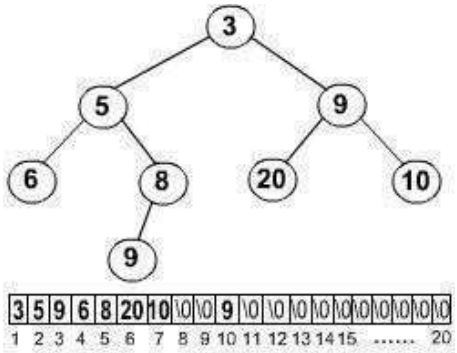
1. Array Representation of BinaryTree
2. Pointer-based.

Array Representation of Binary Tree:

A single array can be used to represent a binary tree.

For these nodes are numbered / indexed according to a scheme giving 0 to root. Then all the nodes are numbered from left to right level by level from top to bottom. Empty nodes are also numbered. Then each node having an index i is put into the array as its i^{th} element.

In the figure shown below the nodes of binary tree are numbered according to the given scheme.



Thefigureshowshowabinarytreeisrepresentedasanarray. Theroot 3 is the 0th element whileits leftchild 5isthe 1 stelement of thearray. Node 6 does not haveanychildsoitschildreni.e. 7 th and 8 th element of the array are shown as a Null value.

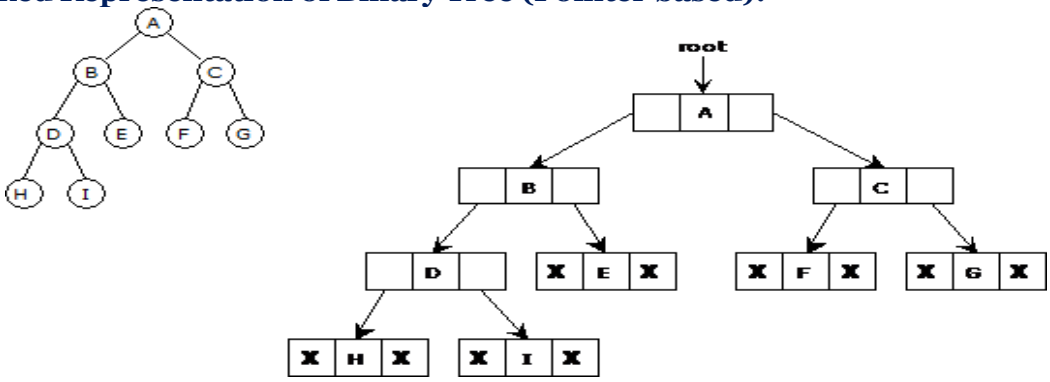
It is found that if n is the number or index of a node, then its left child occurs at (2n + 1)th position and right child at (2n + 2) th position of the array. If any node does not have any of its child, then null value is stored at the corresponding index of the array.

The following program implements the above binary tree in an array form. And then traverses the tree in inordertraversal.

```
# Python implementation to construct a Binary Tree
from # parentarray

# A node
structure class
Node:
    # A utility function to create a new
    node definit(self, key):
        self.key = key
        self.left =
        Noneself.right
        =None

""" Creates a node with key as 'i'. If i is
    root,then it changes root. If parent of i is not
    created,then it createsparentfirst
"""
defcreateNode(parent, i, created, root):

    # If this node is already
    created if created[i] is not
    None:
        return

    # Create a new node and set
    created[i] created[i] = Node(i)

    # If 'i' is root, change root pointer and return
```

```
    if parent[i] == -1:
        root[0] = created[i] # root[0] denotes root of the
        tree return

    # If parent is not created, then create parent
    first if created[parent[i]] is None:
        createNode(parent, parent[i], created, root )

    # Find parent
    pointer p =
    created[parent[i]]

    # If this is first child of
    parent if p.left is None:
        p.left =
    created[i] # If
    second child else:
        p.right = created[i]


# Creates tree from parent[0..n-1] and returns root
of the # created tree
defcreateTree(parent
    ): n = len(parent)

    # Create and array created[] to keep track
    # of created nodes, initialize all entries as
    None created = [None for i in range(n+1)]

    root = [None]
    for i in
    range(n):
        createNode(parent,  i,  created,

    root) return root[0]

 #Inorder traversal of
tree  definorder(root):
  if root is not None:
        inorder(root.left
        )  print  root.key,
        inorder(root.righ
        t)

# Driver Method
parent = [-1, 0, 0, 1, 1, 3, 5]
root = createTree(parent)
print "Inorder Traversal of constructed
tree" inorder(root)
```

Linked Representation of Binary Tree (Pointer based):



Binary trees can be represented by links where each node contains the address of the left child and the right child. If any node has its left or right child empty then it will have in its respective link field, a null value. A leaf node has null value in both of its links.

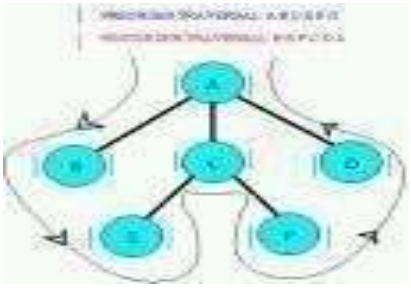
```
# Python program to create a Complete Binary Tree
from # its linked list representation

# Linked List
node class
ListNode:

    # Constructor to create a new
    node definit (self,data):
      self.data = data
      self.next =None

# Binary Tree Node
structure
classBinaryTreeNode:

  # Constructor to create a new
  node definit (self,data):
    self.data = data
    self.left =
    Noneself.right
    =None

# Class to convert the linked list to Binary
Tree class Conversion:

  # Constructor for storing head of linked
  list # and root for the Binary Tree
  definit (self, data=
    None): self.head =
    None self.root = None
```

```
def push(self, new_data):

    # Creating a new linked list node and storing data
    new_node = ListNode(new_data)

    # Make next of new node as
    head new_node.next = self.head

    # Move the head to point to new
    node self.head = new_node

def   convertList2Binary(self):

    # Queue to store the parent
    nodes q = []

    # Base Case
    ifself.head is None:
        self.root = None
        return

    # 1.) The first node is always the root
    node, # and add it to the queue
    self.root =
    BinaryTreeNode(self.head.data)
    q.append(self.root)

    # Advance the pointer to the next
    node self.head = self.head.next

    # Until th end of linked list is reached,
    do: while(self.head):

    # 2.a) Take the parent node from the q
    and # and remove it from q
    parent = q.pop(0) # Front of queue

   # 2.c) Take next two nodes from the linked
   list. # We will add them as children of the
   current
   # parent node in step 2.b.
   # Pushthem intothe queuesothattheywill
   be # parent to the future node
       leftChild= None
       rightChild =
       None

       leftChild =
       BinaryTreeNode(self.head.data)
```

```
            q.append(leftChild)
            self.head =
            self.head.nextif(self.hea
            d):
               rightChild =
               BinaryTreeNode(self.head.data)
               q.append(rightChild)
               self.head = self.head.next

            #2.b) Assign the left and right children of parent
            parent.left =
            leftChildparent.right =
            rightChild

    definorderTraversal(self,
        root): if(root):
            self.inorderTraversal(root.left
            ) print root.data,
            self.inorderTraversal(root.rig
            ht)

# Driver Program to test above

function#  Objectofconversion  class
conv =
Conversion()
conv.push(36)
conv.push(30)
conv.push(25)
conv.push(15)
conv.push(12)
conv.push(10)

conv.convertList2Binary()

print "Inorder Traversal of the contructed Binary Tree
is:" conv.inorderTraversal(conv.root)
```

Binary Tree Traversals:

Traversal of a binary tree means to visit each node in the tree exactly once. The tree traversal is used in all tit.



In a linear list nodes are visited from first to last, but a tree being a non linear one we need definite rules. Th ways to traverse a tree. All of them differ only in the order in which they visit the nodes.

The three main methods of traversing a tree are:

- Inorder Traversal
- PreorderTraversal
- PostorderTraversal

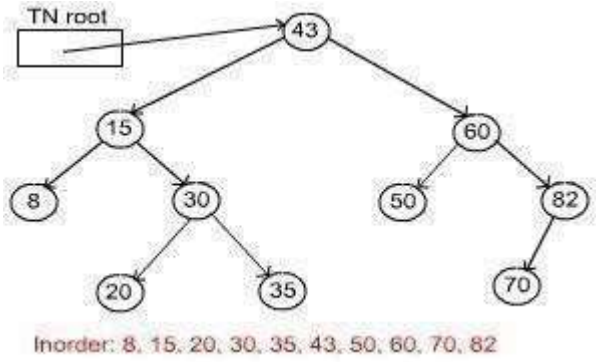
In all of them we do not require to do anything to traverse an empty tree. All the traversal methods are base functions since a binary tree is itself recursive as every child of a node in a binary tree is itself a binary tree.

Inorder Traversal:

To traverse a non empty tree in inorder the following steps are followed recursively.

- Visit theRoot
- Traverse the leftsubtree
- Traverse the rightsubtree

The inorder traversal of the tree shown below is as follows.



Preorder Traversal:

Algorithm Pre-order(tree)

1. Visit theroot.
2. Traverse the left sub-tree, i.e., callPre-order(left-sub-tree)
3. Traverse the right sub-tree, i.e., callPre-order(right-sub-tree)

Post-order Traversal:

Algorithm Post-order(tree)

1. Traverse the left sub-tree, i.e., callPost-order(left-sub-tree)
2. Traverse the right sub-tree, i.e., callPost-order(right-sub-tree)
3. Visit theroot.

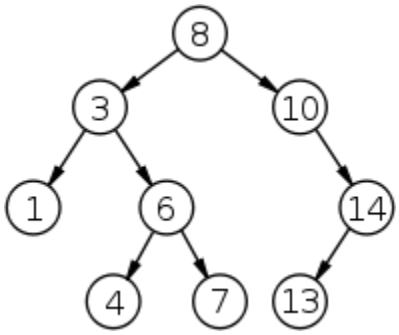
```
# Python program for tree traversals

# A class that represents an individual node
in a # Binary Tree
class Node:
    definit (self,key):
        self.left = None
        self.right  =
        None self.val
        = key


# A function to do inorder tree
traversal defprintInorder(root):

    if root:

        # First recur on left
        child
        printInorder(root.left)

        # then print the data of
        node print(root.val),

        # now recur on right
        child
        printInorder(root.right)



# A function to do postorder tree
traversal defprintPostorder(root):

    if root:

        # First recur on left
```

```
        child
        printPostorder(root.lef
        t)

        # the recur on right
        child
        printPostorder(root.rig
        ht)

        # now print the data of
        node print(root.val),


# A function to do postorder tree
traversal defprintPreorder(root):

    if root:

        # First print the data of
        node print(root.val),

        # Then recur on left
        child
        printPreorder(root.left)

        # Finally recur on right
        child
        printPreorder(root.right)

root = Node(1)
root.left     = Node(2)
root.right    =
Node(3) root.left.left
=Node(4)
root.left.right = Node(5)
print "Preorder traversal of binary tree
is" printPreorder(root)

print "\nInorder traversal of binary
tree is" printInorder(root)

print "\nPostorder traversal of binary
tree is" printPostorder(root)

Time Complexity: O(n^2).
```

Binary Search Tree:

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

- The left sub-tree of a node contains only nodes with keys less than the node‘skey.
- The right sub-tree of a node contains only nodes with keys greater than the node‘skey.
- The left and right sub-tree each must also be a binary search tree. There must be no duplicatenodes**.**



The above properties of Binary Search Tree providean ordering among keys sothat the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

Searching a key

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

```
# A utility function to search a given key in
BST defsearch(root,key):
  # Base Cases: root is null or key is present at root
  if root is None or root.val ==
     key: return root
  # Key is greater than root's
  key if root.val< key:
     return  search(root.right,key)#
    Key is smaller than root's
 key return search(root.left,key)
```

Priority Queues

Priority Queue is an extension of queue with following properties.
1) Every item has a priority associated withit.
2) An element with high priority is dequeued before an element with lowpriority.
3) If two elements have the same priority, they are served according to their order in thequeue.

A typical priority queue supports following operations.
insert(item, priority): Inserts an item with given priority. **getHighestPriority():** Returns the highest priority item. **deleteHighestPriority():** Removes the

highest priority item.

Implementation priority queue

Using Array: A simple implementation is to use array of following structure.

insert() operation can be implemented by adding an item at end of array in O(1) time.

getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.

deleteHighestPriority() operation can be implemented byfirst linearly searching an item, then removing the item by moving all subsequent items one position back.

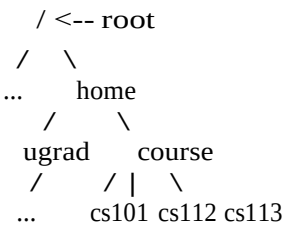
We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage withlinked list is **deleteHighestPriority()** can be more efficient as we don't have to move items.

Applications of Priority Queue:
1) CPUScheduling
2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree,etc
3) All queue applications where priority isinvolved.

Application of Trees:

1. One reason to use trees might be because you want to store information that naturally formsa hierarchy. For example, the file system on acomputer:

file system
———



2. If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of $O(\log_n)$ forsearch.

3) We can insert/delete keys in moderate time (quicker than Arrays and slower thanUnordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of $O(\log_n)$ forinsertion/deletion.

4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have

anupper limit on number of nodes as nodes are linked usingpointers.

The following are the common uses of tree.
1. Manipulate hierarchicaldata.
2. Make information easy to search (see treetraversal).
3. Manipulate sorted listsofdata.
4. As a workflow for compositing digital images for visualeffects.
5. Routeralgorithms

Basic Graph Concepts:

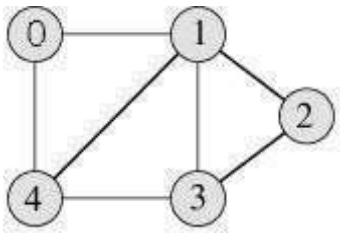
Graph is a data structure that consists of following two components:
1. A finite set of vertices also called asnodes.
2. A finite set of ordered pair of the form (u, v) called asedge.

The pair is ordered because (u, v) is not same as (v, u) in case of directed graph (di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graph and its representations:

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used insocial networks likelinkedIn, facebook. For example, infacebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, genderand locale.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.
1. AdjacencyMatrix
2. AdjacencyList

Thereareotherrepresentationsalsolike, Incidence Matrix and Incidence List. Thechoiceofthegraph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D

array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

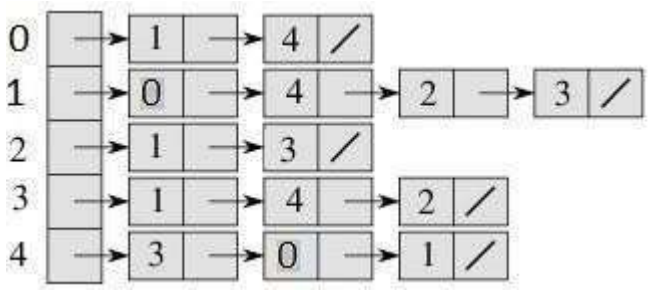
Adjacency Matrix Representation of the above graph

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

Cons: Consumes more space O(V^2). Even if the graph is sparse (contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Adjacency List:

An array of linkedlists isused. Size of the arrayis equal to numberof vertices. Let the array be array[]. An entry array[i] represents the linked list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.

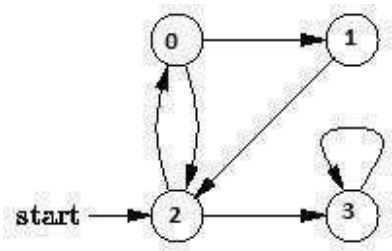


Adjacency List Representation of the above Graph

Breadth First Traversal for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visitedarray.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don‘t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Breadth First Traversal of the following graph is 2, 0, 3,1.



Algorithm: Breadth-First Search

Traversal BFS(V, E, *s*)

```
for each u in V − {s}
    do color[u] ← WHITE
       d[u] ← infinity
       π[u] ← NIL
       color[s] ← GRAY
       d[s] ← 0
       π[s] ← NIL
       Q ← {}
       ENQUEUE(Q, s)
    while Q is non-empty
     do u ←DEQUEUE(Q)
          for each v adjacent to u
              do if color[v] ← WHITE
                 then color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
          DEQUEUE(Q)
       color[u] ← BLACK
```

Applications of Breadth First Traversal

1) ShortestPathandMinimumSpanningTreeforunweightedgraph Inunweightedgraph,theshortestpathisthe path with least number of edges. With Breadth First, we always reach a vertex from given source using minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2) Peer to Peer Networks. In Peer to Peer Networks like BitTorrent, Breadth First Search is used to find all neighbornodes.

3) Crawlers in Search Engines: Crawlers build index using Bread First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of built tree can belimited.

4) Social Networking Websites: In social networks, we can find people within a given distance _k' from a person using Breadth First Search till _k'levels.

5) GPS Navigation systems: Breadth First Search is used to find all neighboringlocations.

6) Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

7) In Garbage Collection: Breadth First Search is used in copying garbage collection using Cheney's algorithm.

8) Cycle detection in undirected graph: Inundirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. In directed graph, only depth first search can beused.

9) Ford-Fulkerson algorithm In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to$O(VE^2)$.

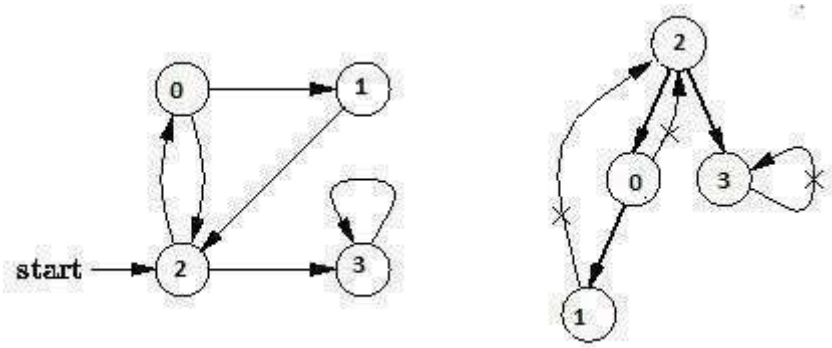
10) To test if a graph is Bipartite We can either use Breadth First or Depth FirstTraversal.

11) Path Finding We can either use Breadth First or Depth First Traversal to find if there is a path between twovertices.

12) Finding all nodes within one connected component: We can either use Breadth First or Depth First Traversal to find all nodes reachable from agivennode.

Depth First Traversal for a Graph

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For example, in the following graph, we start traversal from vertex
2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. Depth First Traversal of the following graph is 2, 0, 1, 3



Algorithm Depth-First Search

The DFS forms a depth-first forest comprised of more than one depth-first trees. Each tree is made of edges (u, v) such that u is gray and v is white when edge (u, v) is explored. The following pseudocode for DFS uses a global timestamp time.

DFS (V, E)

```
for each vertex u in V[G]
    do color[u] ← WHITE π[u] ← NIL
            time ← 0
        for each vertex u in V[G]
          do if color[u] ← WHITE
              then DFS-Visit(u)
```

DFS-Visit(u)

```
color[u] ← GRAY time ← time + 1
d[u] ← time
for each vertex v adjacent to u
          do if color[v] ← WHITE
              then π[v] ← u
           DFS-Visit(v) color[u] ←
BLACK time ← time +1
f[u] ← time
```

Applications of Depth First Search

Depth-first search (DFS) is an algorithm (or technique) for traversing agraph. Following

are the problems that use DFS as a building block.

HEAP:

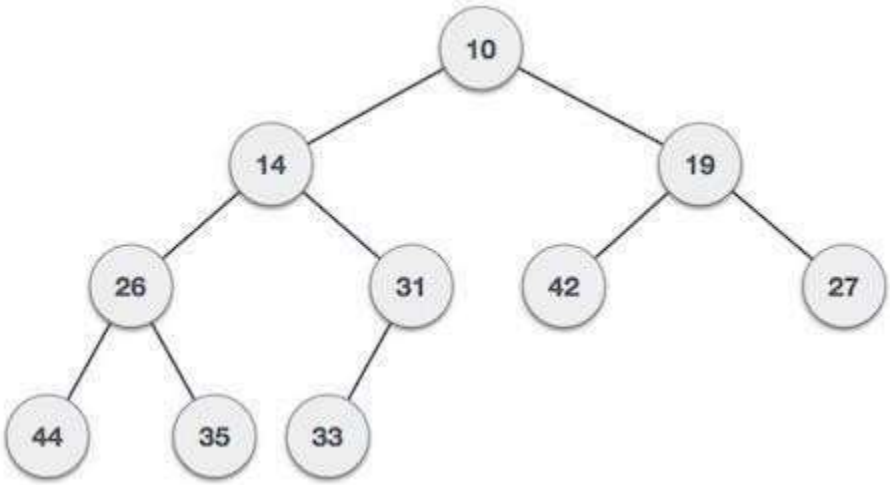
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If **α** has child node **β** then −

$$key(\alpha) \geq key(\beta)$$

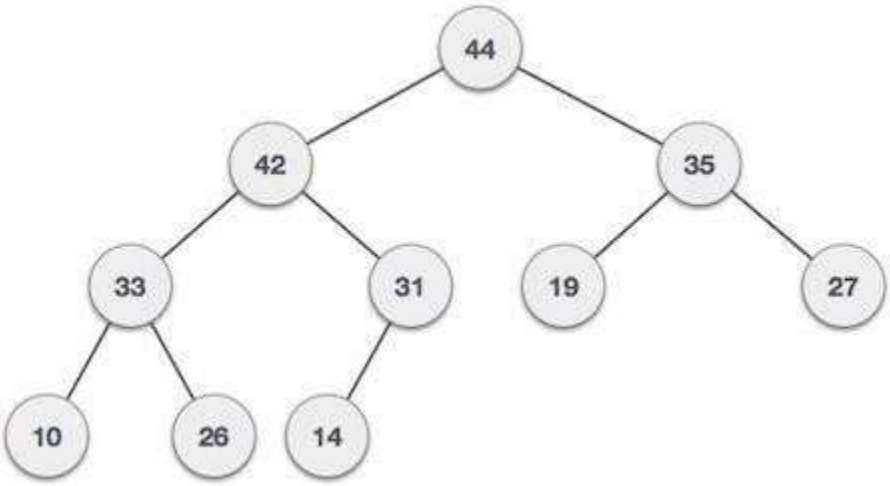
As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types −

For Input → 35 33 42 10 14 19 27 44 26 31

Min-Heap − Where the value of the root node is less than or equal to either of its children.



Max-Heap − Where the value of the root node is greater than or equal to either of its children.



Both trees are constructed using the same input and order of arrival.

Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

Step 1 − Create a new node at the end of heap.
Step 2 − Assign new value to the node.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.

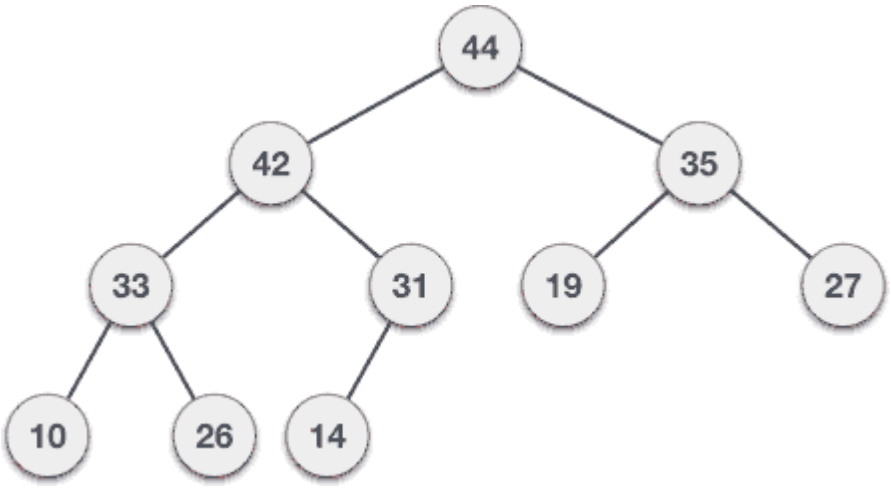
Note − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

Step 1 − Remove root node.
Step 2 − Move the last element of last level to root.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.



The **binary heap** data structure is an array that can be viewed as a complete binary tree. Each node of the binary tree corresponds to an element of the array. The array is filled on all levels except possibly lowest (the lowest level is filled in left to right order and need not be complete).

Binary Min-Heaps

A *heap* is a tree-based structure, but it doesn't use the binary-search differentiation between the left and right sub-trees to create a linear ordering. Instead, a binary heap only specifies the relationship between a parent and its children. For a *min-heap*, a node must be less than all of its children and it, in turn, must be greater than its parent (if any). Thus, a *binary min-heap* is a binary tree that satisfies the *min-heap* property.

Important: there is no other relationship between the children of a node other than theyareall greater than their common parent. The failure to comprehend this has previously been the greatest source oferrors.

Thus, it is reasonable to compare binary search trees and binary heaps as is shown in Table 1.

Table 1. Properties of binary search trees versus binary heaps.

Binary Search Tree	Binary Heap
Given a node, all objects in the left sub-tree are less than thenode, all objects in the right sub-tree are greater than the node, and both sub-trees are also binarysearch trees.	Given a node, all strict descendants are greater than the node, and both sub-trees are also binary heaps.

Normally, for a priority queue, you would store objects and their associated priorities. For simplicity, our heaps will store only the priority—the associated object with that priority could be integrated into the heaps that we will be implementing.

Figure 1 gives an example of a heap. You will notice that the four largest values are in the left sub-tree which also contains the next larger value from 3.

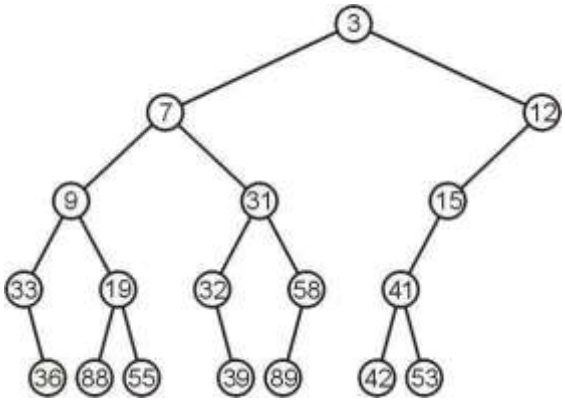


Figure 1. A binary heap.Operations The

operations we will look at include *top*, *pop*, and *push*.

Top

Finding the smallest entry in Figure 1 is an □(1) operation—just return the root node.

Pop

Removing the top element could be implemented as easily as removing the top (in this case, 3) and then recursively promoting the smaller of the children. Thus, removing 3 from Figure 1 results in the heap shown in Figure 2. Because we are always promoting the smaller of the two children, we are guaranteed that the min-heap structure is maintained.

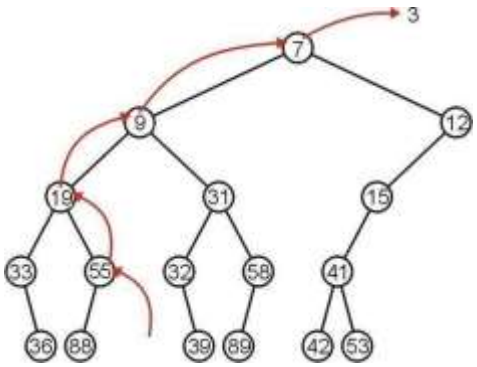


Figure 2. The binary min-heap resulting from a pop operation being performed on the heap in Figure 1

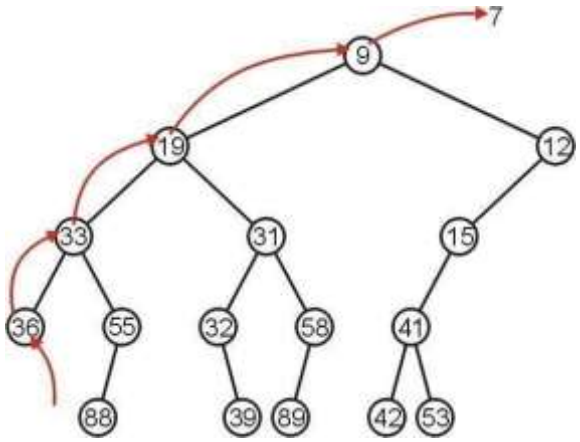


Figure 3. The binary min-heap resulting from a pop operation being performed on the heap in Figure 2.

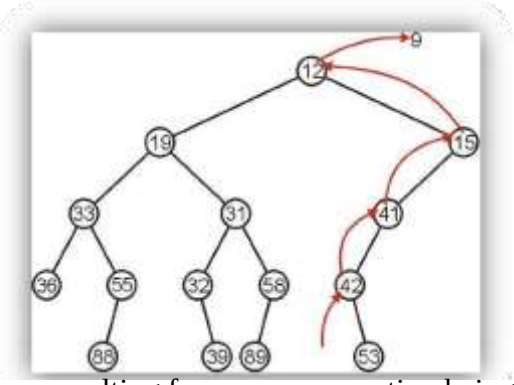


Figure 4. The binary min-heap resulting from a pop operation being performed on the heap in Figure 3.

Push

When we insert a new object into a min-heap, we have one of two options:

1. Insert the object at the root, or
2. Insert the object at an empty node.

In the first case, we would recursively apply a rule such as replace the node with minimum of the current node and the value being inserted and insert the larger of the two into one of the two sub-heaps.

In the second case, you would recursively apply a rule such as swap the inserted node with its parent until the parent is less than the inserted value or we are at the root.

We will look at a few examples of the second option. Inserting 17 into the tree in Figure 4 could occur at any location, so we will choose one arbitrary empty node. We compare it with its parent,32, and swap these two, for 17 < 32. We then examine the new parent of 17 and notice 31 > 17, so we continue swapping. We continue until finally 12 < 17. This process is called percolating up (similar to the actions of a coffee percolator). Figure 5 shows this percolation.

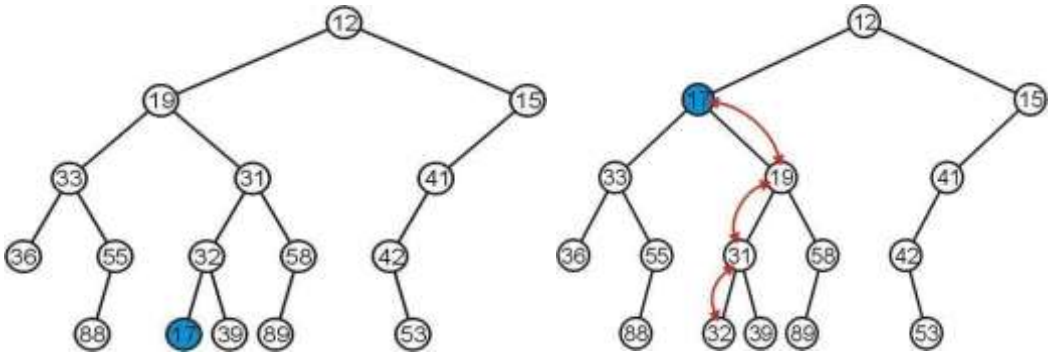


Figure 5. Inserting 17 at an arbitrary location and percolating it up.

One issue with such an insertion is: how do we know where the empty nodes are located? We will solve this in the following implementation.

Complete-tree Implementation

There are numerous data structures that implement min-heaps, including using a complete binary

tree, leftist heaps, skew heaps, binomial heaps, and Fibonacci heaps. While some of therun-time characteristics of using complete binary trees are sub-optimal as compared to others, it is most efficient with respect to memory usage.

If you recall the definition of a *complete binary tree* in Topic 4.6, you will recall the entries of a complete tree are filled in breadth-first traversal order. Figure 6 shows a min-heap that contains the same nodes as Figure 5 but in the structure of a complete tree.

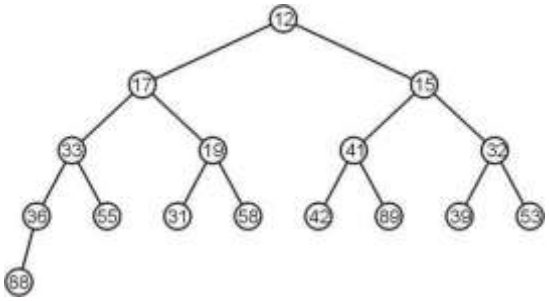


Figure 6. The min-heap of Figure 5 rearranged to satisfy the shape of a complete tree.

At this point, determining where one can insert a new node is trivial: there is only one place thata new can be inserted: to the right of 88. Suppose we are inserting 25. We would insert 25 at this location and then percolate it up to the appropriate location within the heap.

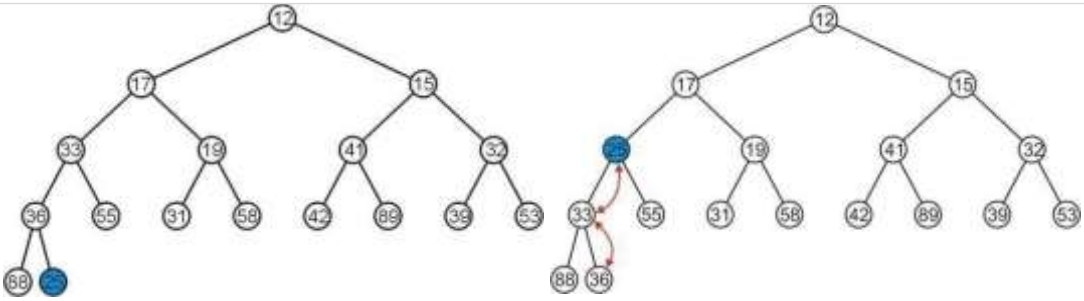


Figure 7. The insertion of and percolating up of 25 into the min-heap in Figure 6.

Thus, the use of a complete tree makes the implementation of Option 2 for performing pushes into a binary min-heap described in §7.2.1.3 straight-forward. Unfortunately, if we use the same rule for performing a pop, as described in §7.2.1.2, we run into a problem. Suppose we pop the top off of the tree in Figure 7. In order to maintain the min-tree structure, we would promote 15, 32, 39, and this would leave a gap in the tree as shown in Figure 8.

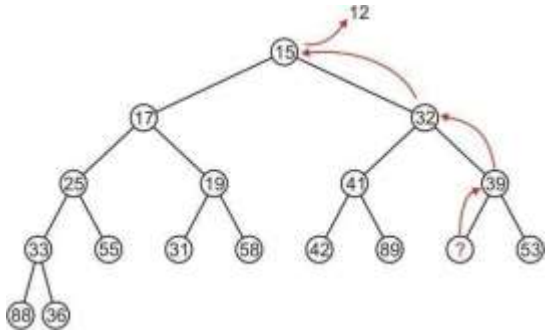


Figure 8. Popping the top off of the heap in Figure 7.

With the empty node at that location, the tree is no longer complete. Instead, consider the following strategy for popping the top:
Replace the root node with the last entry in the complete tree. Then proceed to percolate that value down into the heap until the resulting tree is again a min heap.
For example, in Figure 9, we replace 12 with the last entry, 36, and then we proceed to swap 36with the minimum of the two children until 36 is greater than both its children.

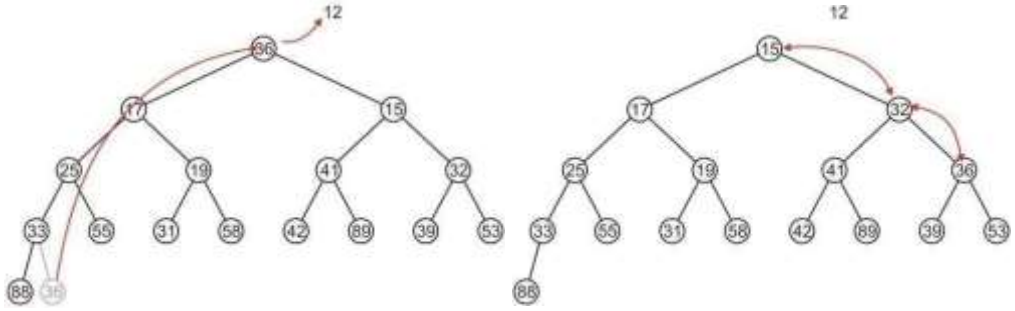


Figure 9. Popping the top of Figure 7 by replacing 12 with the last entry and percolating it down until it is greater than both itschildren.

This maintains the complete tree structure. As another example, suppose we pop the top again. Replace 15 with the last entry, 88, and percolate 88 down until it is less than both its children.In this case, we continue percolating until 88 ends up in a leaf node, as shown in Figure 10.

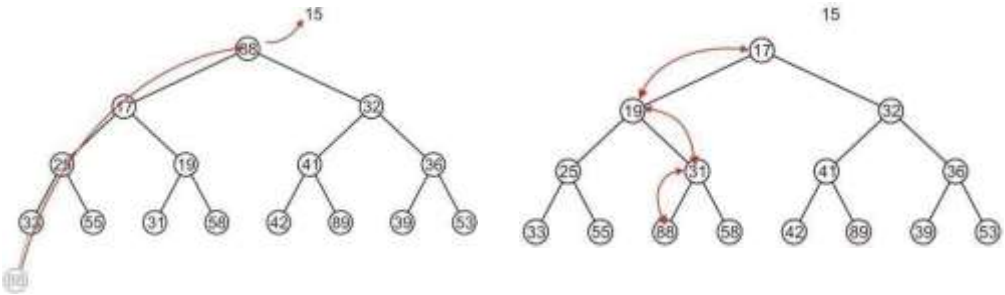


Figure 10. Popping the top from the final min-heap in Figure 9.

Two further pops are shown in Figures 11 and 12.

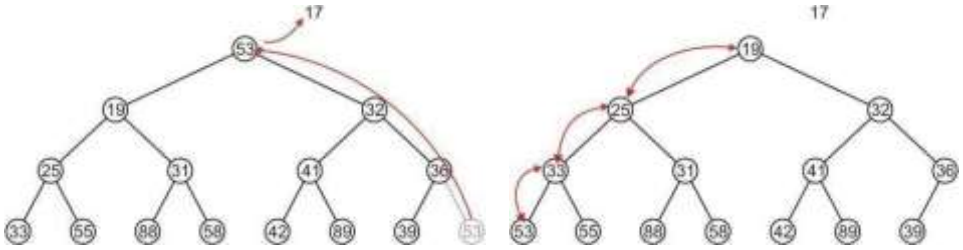


Figure 11. Popping the top from the final tree in Figure 10.

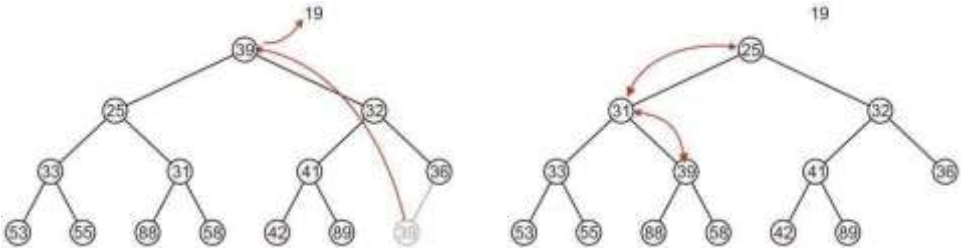


Figure 12. Popping the top from the final tree in Figure 11.

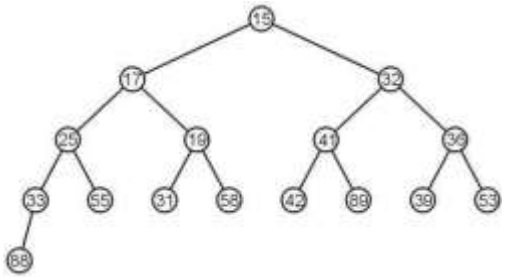
Thus, we can maintain the min-heap in the shape of a complete tree with both push and pop operations. We will now proceed to consider how we can implement a binary min-heap using an array.

Array Implementation of a Binary Min-Heap

we discussed how we could store a complete tree in an array by using the entries 1 through n. Inthis case, the children of the object at index k are located in indices $2k$ and $2k + 1$ while the parent is located at index $k/2$.

Example 1

For example, the tree resulting in Figure 9 would be stored in the array shown in Figure 13.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	15	17	32	25	19	41	36	33	55	31	58	42	89	39	53	88

Figure 13. The complete tree stored as an array using array indices 1 through 16.

Thus, for example, we see that:

- The children of the root, 15 at the root, are located at indices 2 and 3 (17 and 32),
- The children of 17 (index 2) are located at indices 4 and 5 (25 and 19), and
- The children of 19 (index 5) are located at indices 10 and 11 (31 and 58).

If we wanted to pop the minimum element, we would remove 15 and replace it with 88:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	88	17	32	25	19	41	36	33	55	31	58	42	89	39	53	~~88~~

We would then compare 88 with its children in entries 2 and 3. Of these, 17 < 32 < 88 so we

swap 88 and 17.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	17	**88**	32	25	19	41	36	33	55	31	58	42	89	39	53	

Again, we compare 88 and its two children at entries 4 and 5 and determine 19 < 25 < 88, and therefore we swap 88 and 19.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	17	19	32	25	**88**	41	36	33	55	31	58	42	89	39	53	

Again, we compare 88 and its two children at entries 10 and 11 and determine 31 < 58 < 88,and therefore we swap 31 and 88.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	17	19	32	25	31	41	36	33	55	88	58	42	89	39	53	

At this point, 20 and 21 > 15, and thus, 88 has no children to compare to, and thus we are left with a min heap in the shape of a complete tree stored as an array. This results in the complete tree shown in Figure9.

Example 2

Consider the binary min-heap shown in Figure 14.

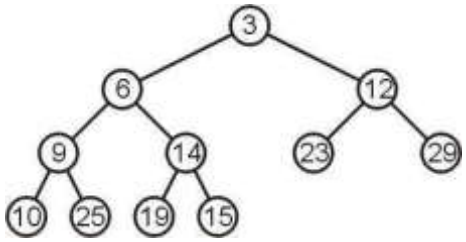


Figure 14. A binary min-heap in the shape of a complete tree.

The array representation of this complete would be

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	3	6	12	9	14	23	29	10	25	19	15					

7.2.3.2.1 Two Insertions (Pushes) into Figure 14

If we were to insert 26, we would first place 26 into the next available index, 12, and compare26 with its parent, 23 at location 12/2 = 6.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	3	6	12	9	14	23	29	10	25	19	15	**26**				

Because 26 > 23, we leave 26 in index 12, as shown in Figure 15.

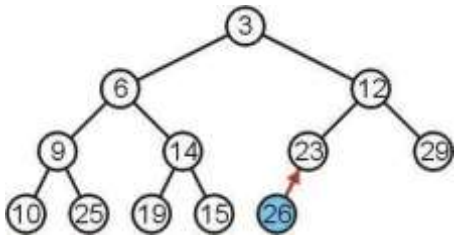


Figure 15. Inserting 26 into the min-heap in Figure 14.

Suppose now we insert 8 in the min heap in Figure 14. It would first be placed into index 13, it would then be compared to the entry 23 at index 13/2 = 6 and because 8 < 23, we would swap them. We would then compare 8 to the entry 12 at 6/2 = 3 and because 8 < 12, we would swap them. Finally, we note that 8 is greater than its parent, the root, so we stop.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	3	6	**8**	9	14	**12**	29	10	25	19	15	26	**23**			

The resulting tree is shown in Figure 16.

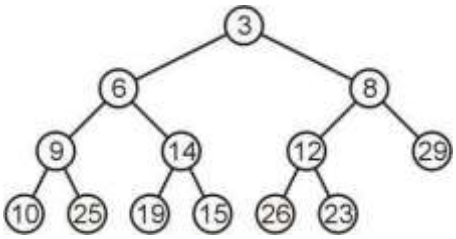


Figure 16. The tree in Figure 15 after pushing thevalue 8.

Three Removals (Pops) From Figure 16

To pop the top off of Figure 16, we remove 3 and copy 23 to entry 1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	23	6	8	9	14	12	29	10	25	19	15	26	~~23~~			

We now percolate 23 down by comparing it with its children: 6 < 8 < 23, so we swap 23 and 6:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	6	**23**	8	9	14	12	29	10	25	19	15	26				

We compare 23 with its two children at 4 and 5 and swap 23 and 9:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	6	**9**	8	**23**	14	12	29	10	25	19	15	26				

We compare 23 with its two children at 8 and 9 and swap 23 and 10:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	6	9	8	**10**	14	12	29	**23**	25	19	15	26				

A this point, 2·8 = 16 and 2·8 + 1 = 17 are both greater than 12 (the last entry), so we are finished.

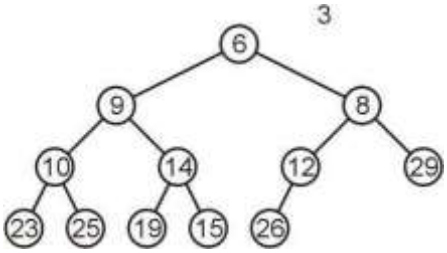


Figure 17. The result of popping the top of Figure 16.

If we pop the minimum of Figure 17, we remove 6 from the tree and replace it with 26:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	26	9	8	10	14	12	29	23	25	19	15	~~26~~				

Next, we compare it with its children and swap 26 and 8:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	8	9	**26**	10	14	12	29	23	25	19	15					

We compare 26 with its children 12 and 29 and swap 26 with 12:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

	8	9	**12**	10	14	**26**	29	23	25	19	15					

A this point, 2·6 = 12 and 2·6 + 1 = 13 are greater than 11 (the last entry), so we are finished.The result is in Figure 18.

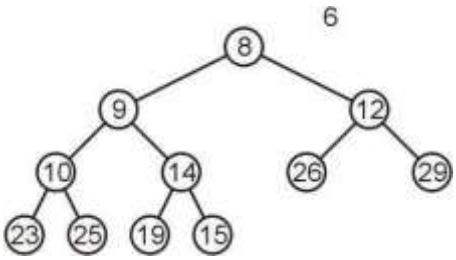


Figure 18. The result of popping the top of Figure 17.

~~Performing one final pop, we remove 8 and replace it with 15:~~

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

	15	9	12	10	14	26	29	23	25	19	~~15~~					

We compare it with its children and swap 9 and 15:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

	9	**15**	12	10	14	26	29	23	25	19						

We compare 15 with its children 10 and 14 and swap 15 and 10:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	9	**10**	12	**15**	14	26	29	23	25	19						

We compare 15 with its children 23 and 25 (at indices 8 and 9) and realize 15 < 23 <25, so we are finished. Figure 19 shows the resulting min heap.

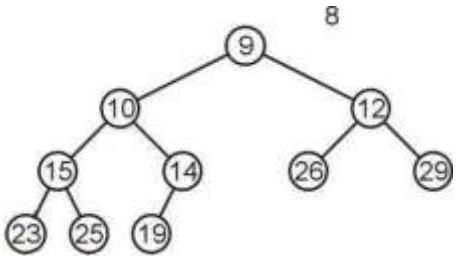


Figure 19. The result of popping the top from Figure 18.

Run-time Analysis

Accessing the minimum is accessing the 2nd array entry: □(1).

When we pop the top object, we replace it with an object at the lowest depth—it is very likely that this object will be percolated down to the bottom again, and thus, we may assume the run time will be O(ln(*n*)).

How about push? Intuitively, it may appear that the average run time is O(ln(*n*)) and if the object being inserted is less than the top, then the run time will be □(ln(*n*)). However, if we assume that a new object could occur at any location, what is the average run time assuming a random insertion?

We have already seen that at least half the entries of a complete tree are leaf nodes, as is demonstrated in Figure 20.

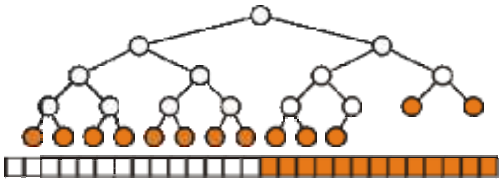
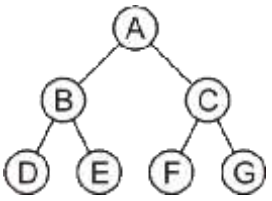


Figure 20. A complete tree noting the leaf nodes.

To simplify the analysis, let us assume that the heap is in the shape of a perfect tree and abalanced distribution of values in the heap. In this case, any object that is inserted that has a priority lower than the meanquarter will percolate once,priority will not percolate up even once. Those in the nextthose in the next eighth will percolate twice, and so on. Therefore, on average, there are 2^k nodes that will require $h - k$ percolations where k runsfrom 0 to h.

Binary Max-Heaps

A binary max-heap is one that is ordered so that a node is always greater than all of its strictdescendants. All other operations are similar to that of a binary min-heap.



UNIT 3

Special Trees & Hashing

AVLTree: Properties, Declarations, Rotations, Insertion, Deletion. B-Tree:Properties, Insertion, Deletion –Trie: Insertion, Deletion, Searching.Hashing - Separate Chaining – Open Addressing – Linear Probing – Quadratic Probing – Double Hashing –Rehashing. Case Study: DNA sequence matching

WHAT IS AVL TREES?

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

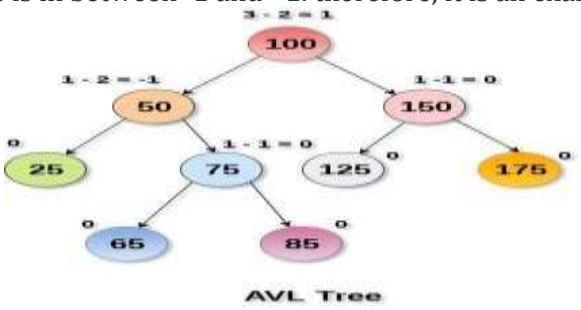
Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL



OPERATIONS ON AVL TREE

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

SN	Operation	Description
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.

2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

WHY AVL TREES?

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

AVL ROTATIONS:

We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

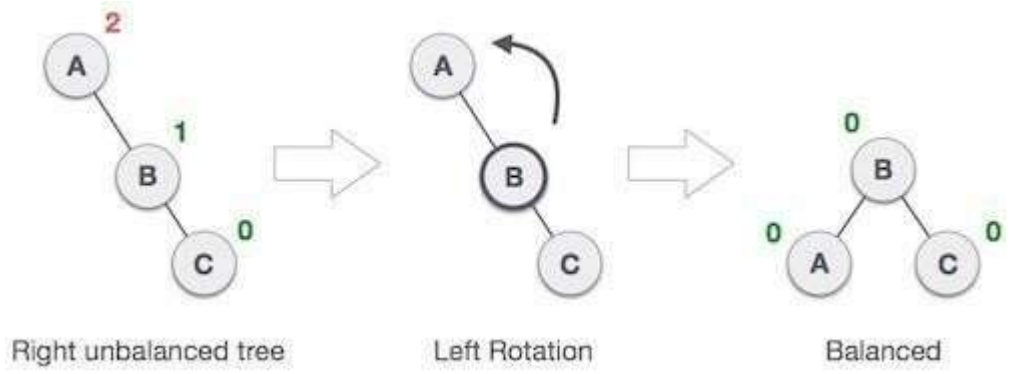
Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. RR Rotation

When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation

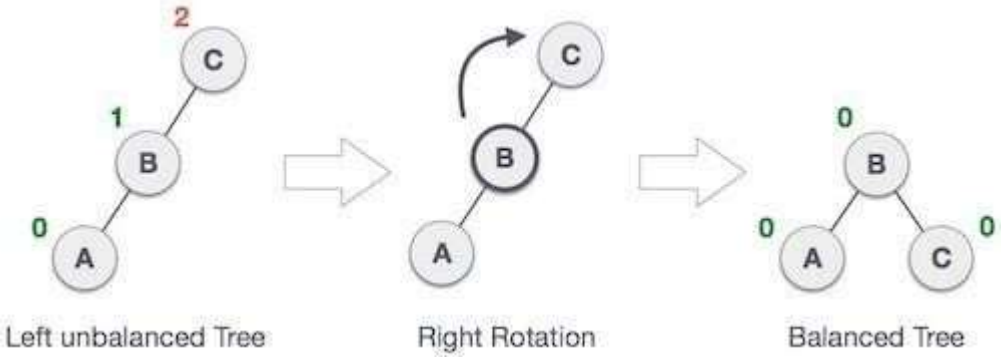
is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation

is clockwise rotation, which is applied on the edge below a node having balance factor 2.



In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

Double rotations are bit tougher than single rotation which has

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation

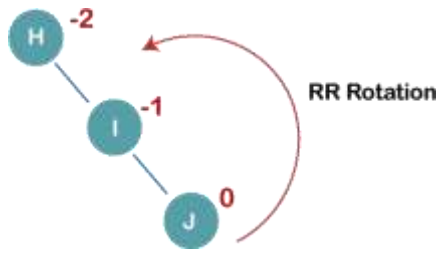
= LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

EXAMPLE 1:

Q: Construct an AVL tree having the following elements

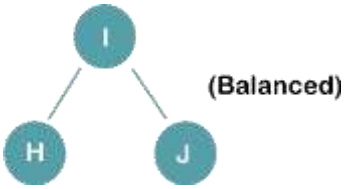
H, I, J, B, A, E, C, F, D, G, K, L

1. Insert H, I, J

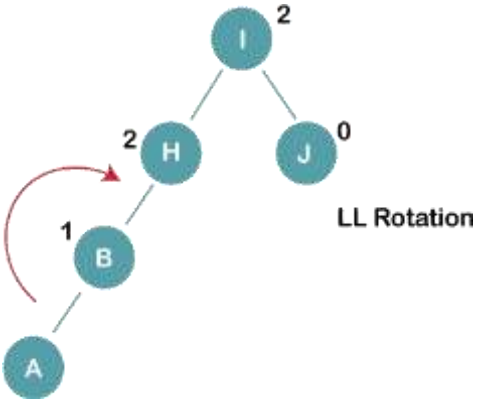


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

The resultant balance tree is:

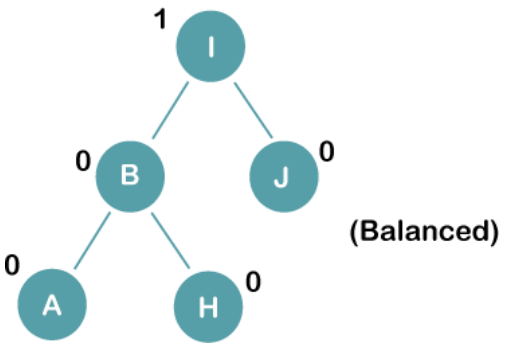


2. Insert B, A

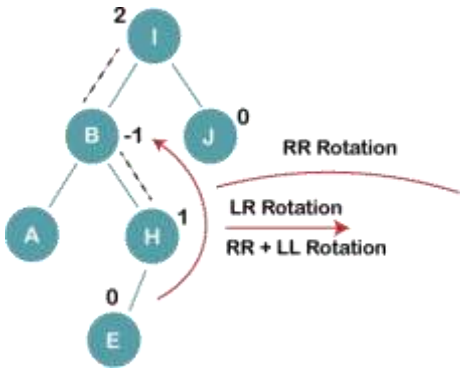


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.

The resultant balance tree is:



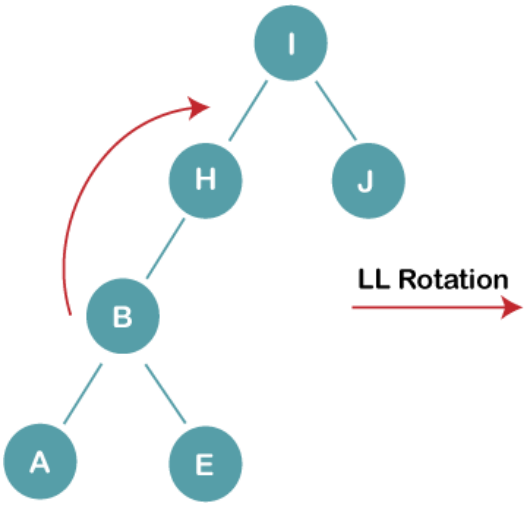
3. Insert E



On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

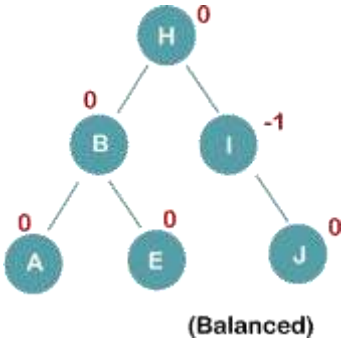
3 a) We first perform RR rotation on node B

The resultant tree after RR rotation is:

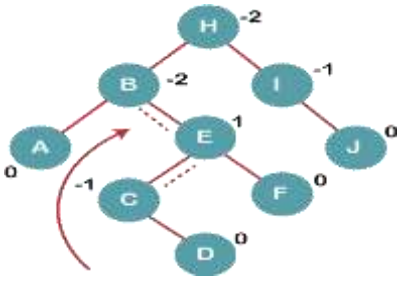


3b) We first perform LL rotation on the node I

The resultant balanced tree after LL rotation is:



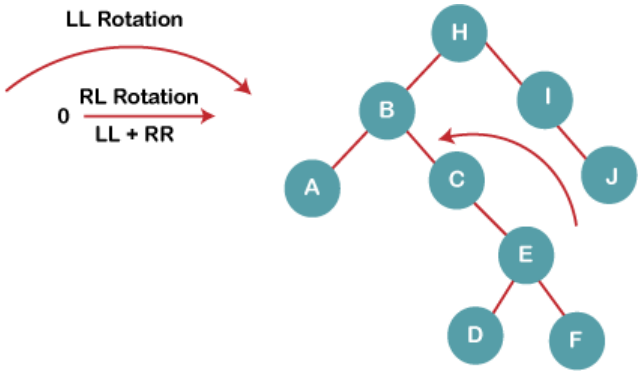
4. Insert C, F, D



On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.

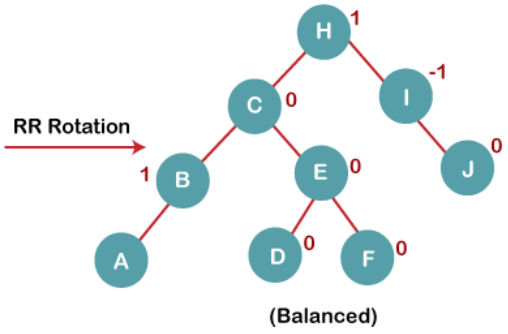
4a) We first perform LL rotation on node E

The resultant tree after LL rotation is:

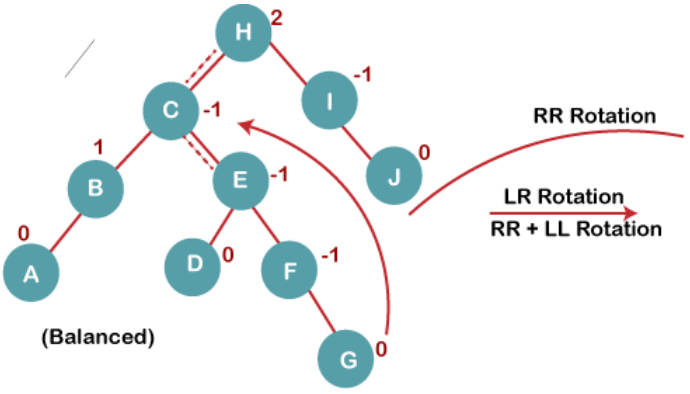


4b) We then perform RR rotation on node B

The resultant balanced tree after RR rotation is:



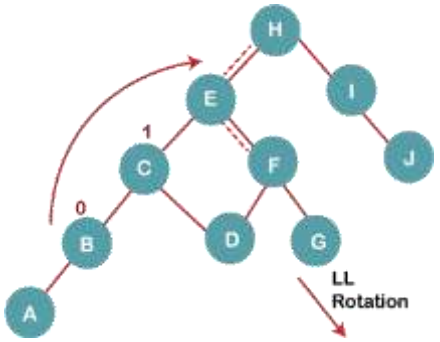
5. Insert G



On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

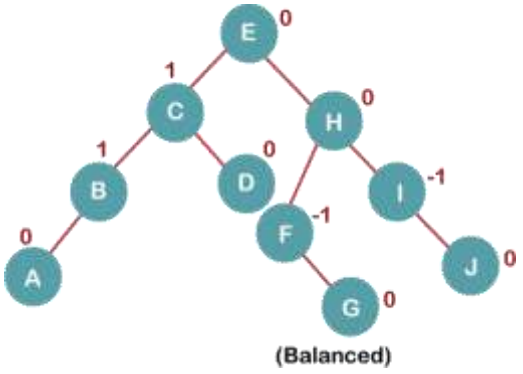
5 a) We first perform RR rotation on node C

The resultant tree after RR rotation is:

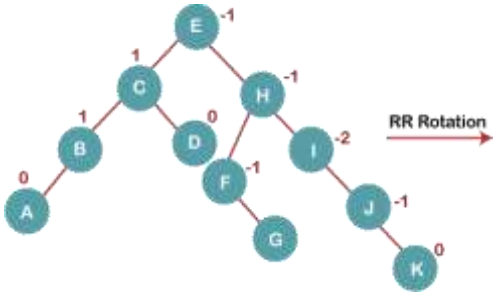


5 b) We then perform LL rotation on node H

The resultant balanced tree after LL rotation is:

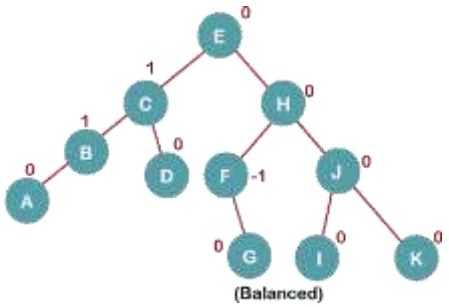


6. Insert K



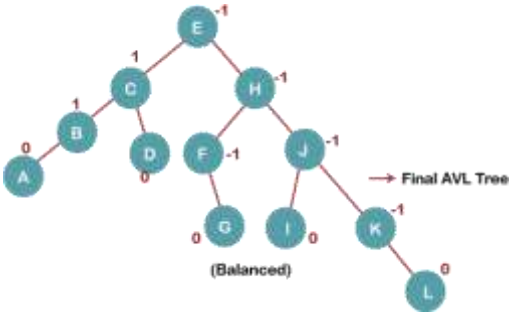
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

The resultant balanced tree after RR rotation is:



7. Insert L

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree



B Tree

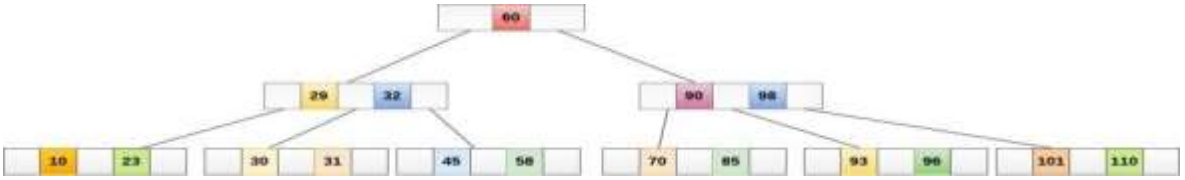
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

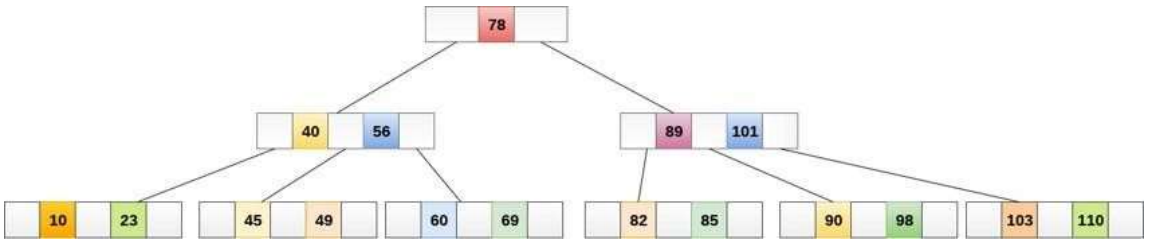
Operations

Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.



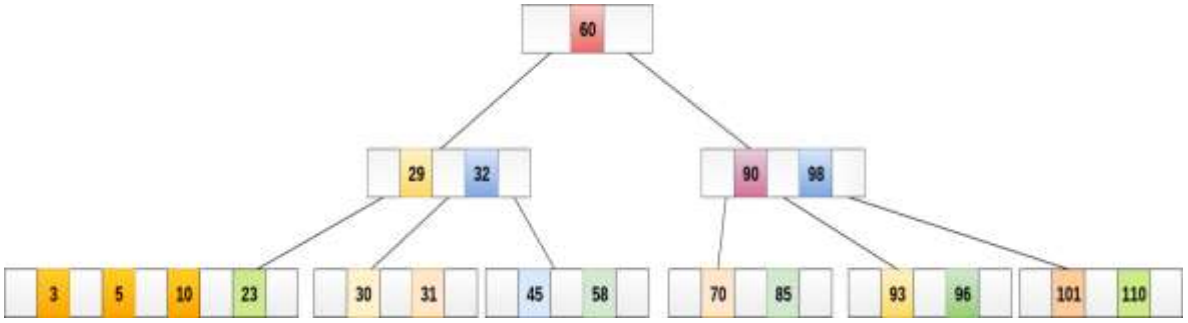
Inserting

Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

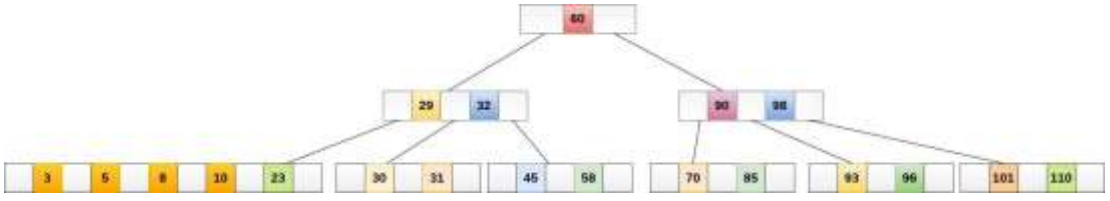
1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at the median.
    - Push the median element upto its parent node.
    - If the parent node also contain m-1 number of keys, then split it too by following the same steps.

Example:

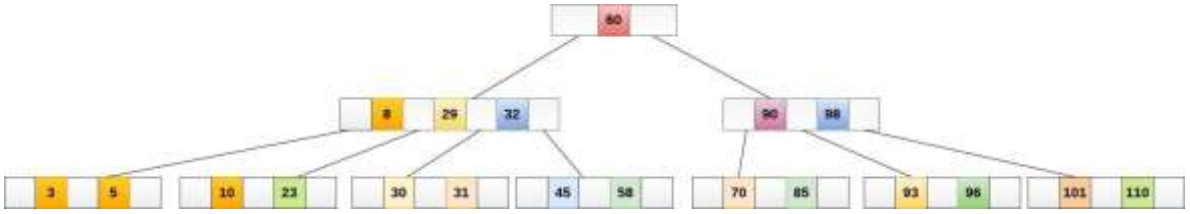
Insert the node 8 into the B Tree of order 5 shown in the following image.



8 will be inserted to the right of 5, therefore insert 8.



The node, now contain 5 keys which is greater than (5 -1 = 4) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.



Deletion

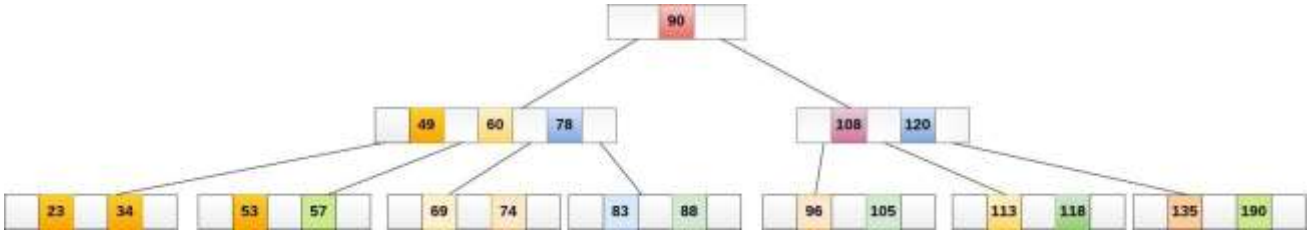
Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

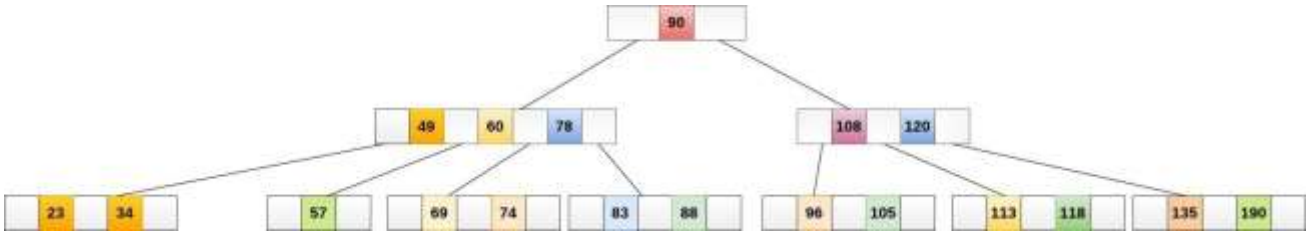
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

Example 1

Delete the node 53 from the B Tree of order 5 shown in the following figure.

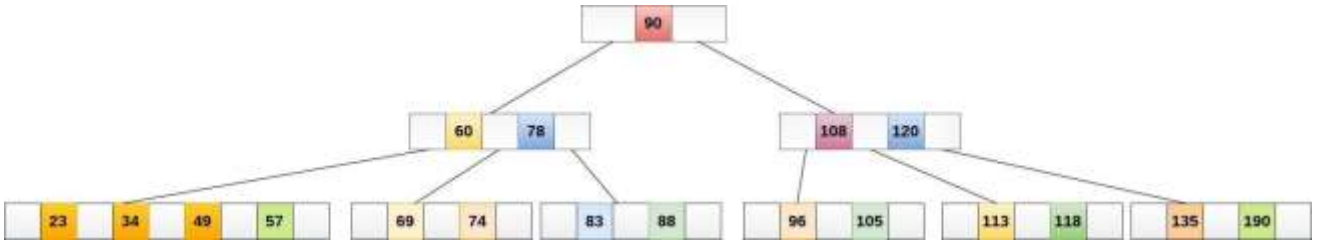


53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.



Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

Rules for B-Tree

Here, are important rules for creating B_Tree

- All leaves will be created at the same level.
- B-Tree is determined by a number of degree, which is also called “order” (specified by an external actor, like a programmer), referred to as

  m
  onwards. The value of

  m
  depends upon the block size on the disk on which data is primarily located.

- The left subtree of the node will have lesser values than the right side of the subtree. This means that the nodes are also sorted in ascending order from left to right.

- The maximum number of child nodes, a root node as well as its child nodes can contain are calculated by this formula:

  m – 1
  For example:

  m = 4
  max keys: 4 – 1 = 3

- Every node, except root, must contain minimum keys of

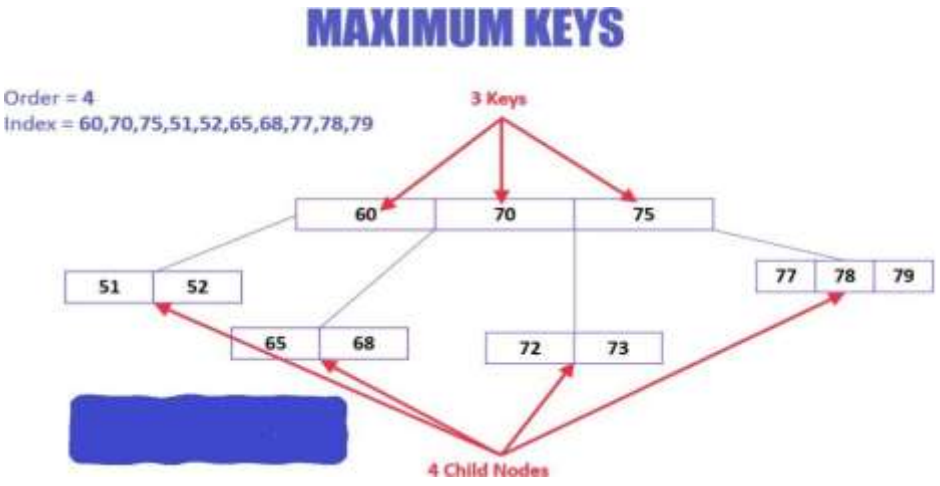
  [m/2]-1
  For example:

  m = 4
  min keys: 4/2-1 = 1.

- The maximum number of child nodes a node can have is equal to its degree, which is

  m

- The minimum children a node can have is half of the order, which is m/2 (the ceiling value is taken).
- All the keys in a node are sorted in increasing order.



Why use B-Tree

Here, are reasons of using B-Tree

- Reduces the number of reads made on the disk
- B Trees can be easily optimized to adjust its size (that is the number of child nodes) according to the disk size
- It is a specially designed technique for handling a bulky amount of data.
- It is a useful algorithm for databases and file systems.

- A good choice to opt when it comes to reading and writing large blocks of data

History of B Tree

- Data is stored on the disk in blocks, this data, when brought into main memory (or RAM) is called data structure.
- In-case of huge data, searching one record in the disk requires reading the entire disk; this increases time and main memory consumption due to high disk access frequency and data size.
- To overcome this, index tables are created that saves the record reference of the records based on the blocks they reside in. This drastically reduces the time and memory consumption.
- Since we have huge data, we can create multi-level index tables.
- Multi-level index can be designed by using B Tree for keeping the data sorted in a self-balancing fashion.

Search Operation

The search operation is the simplest operation on B Tree.

The following algorithm is applied:

- Let the key (the value) to be searched by "k".
- Start searching from the root and recursively traverse down.
- If k is lesser than the root value, search left subtree, if k is greater than the root value, search the right subtree.
- If the node has the found k, simply return the node.
- If the k is not found in the node, traverse down to the child with a greater key.
- If k is not found in the tree, we return NULL.

Insert Operation

Since B Tree is a self-balancing tree, you cannot force insert a key into just any node.

The following algorithm applies:

- Run the search operation and find the appropriate place of insertion.
- Insert the new key at the proper location, but if the node has a maximum number of keys already:
- The node, along with a newly inserted key, will split from the middle element.
- The middle element will become the parent for the other two child nodes.
- The nodes must re-arrange keys in ascending order.

TIP

The following is not true about the insertion algorithm:

Since the node is full, therefore it will split, and then a new value will be inserted

- Search the appropriate position in the node for the key
- Insert the key in the target node, and check for rules
- After insertion, does the node have more than equal to a minimum number of keys, which is 1? In this case, yes, it does. Check the next rule.

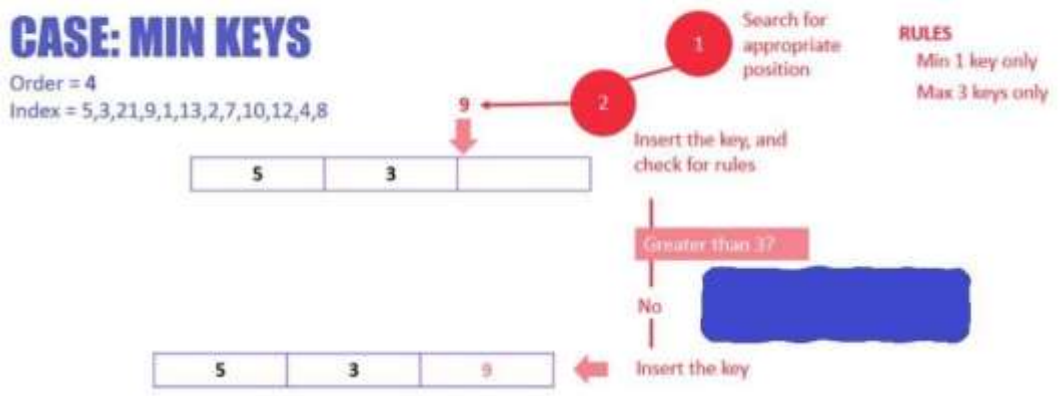
- After insertion, does the node have more than a maximum number of keys, which is 3? In this case, no, it does not. This means that the B Tree is not violating any rules, and the insertion is complete.
- he node has reached the max number of keys
- The node will split, and the middle key will become the root node of the rest two nodes.
- In case of even number of keys, the middle node will be selected by left bias or right bias.
- Let the key (the value) to be searched by "k".
- Start searching from the root and recursively traverse down.
- If k is lesser than the root value, search left subtree, if k is greater than the root value, search the right subtree.
- If the node has the found k, simply return the node.
- If the k is not found in the node, traverse down to the child with a greater key.
- If k is not found in the tree, we return NULL.

Insert Operation

Since B Tree is a self-balancing tree, you cannot force insert a key into just any node.

The following algorithm applies:

- Run the search operation and find the appropriate place of insertion.
- Insert the new key at the proper location, but if the node has a maximum number of keys already:
- The node, along with a newly inserted key, will split from the middle element.
- The middle element will become the parent for the other two child nodes.
- The nodes must re-arrange keys in ascending order.



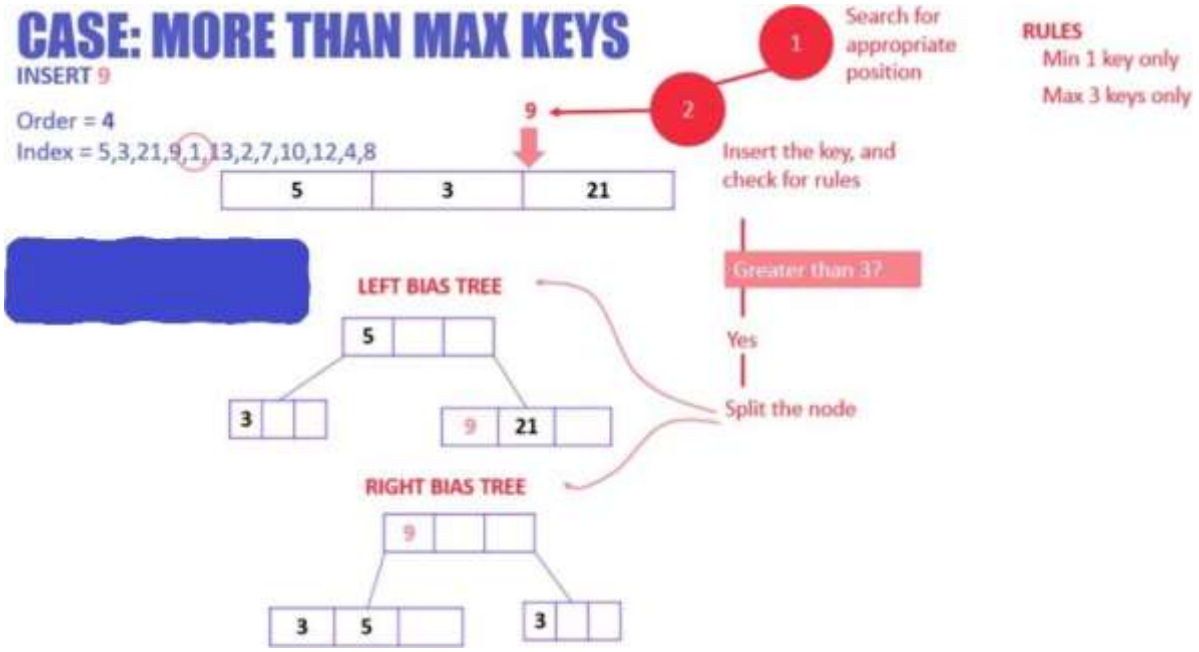
-

The following is not true about the insertion algorithm:

TIP Since the node is full, therefore it will split, and then a new value will be inserted

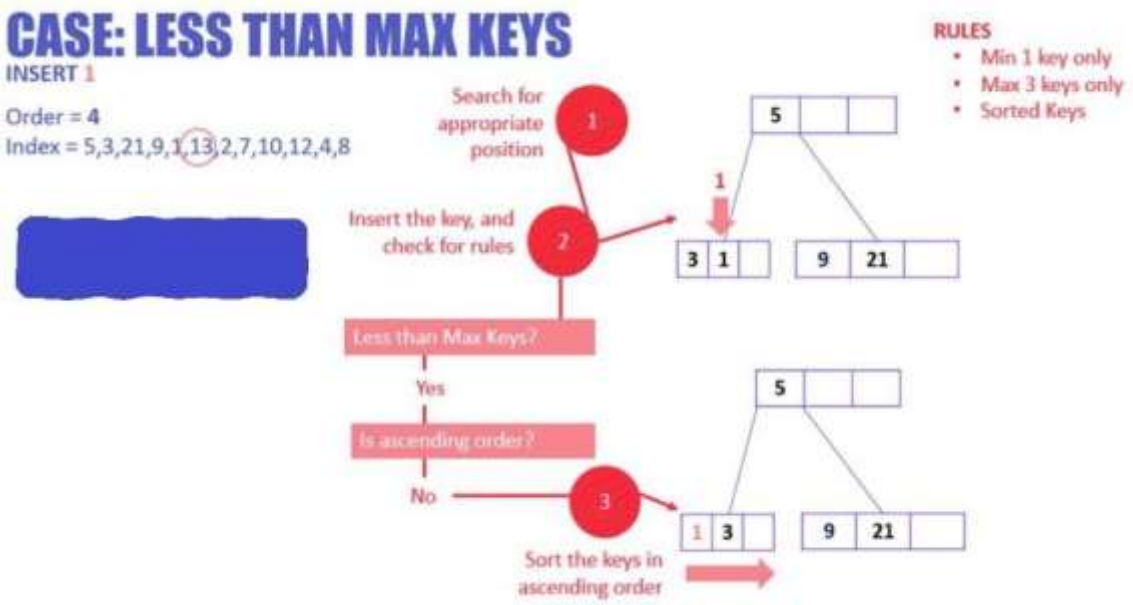
- Search the appropriate position in the node for the key
- Insert the key in the target node, and check for rules
- After insertion, does the node have more than equal to a minimum number of keys, which is 1? In this case, yes, it does. Check the next rule.

- After insertion, does the node have more than a maximum number of keys, which is 3? In this case, no, it does not. This means that the B Tree is not violating any rules, and the insertion is complete.
- The node has reached the max number of keys
- The node will split, and the middle key will become the root node of the rest two nodes.
- In case of even number of keys, the middle node will be selected by left bias or right bias.
- The node has less than max keys
- 1 is inserted next to 3, but the ascending order rule is violated
- In order to fix this, the keys are sorted



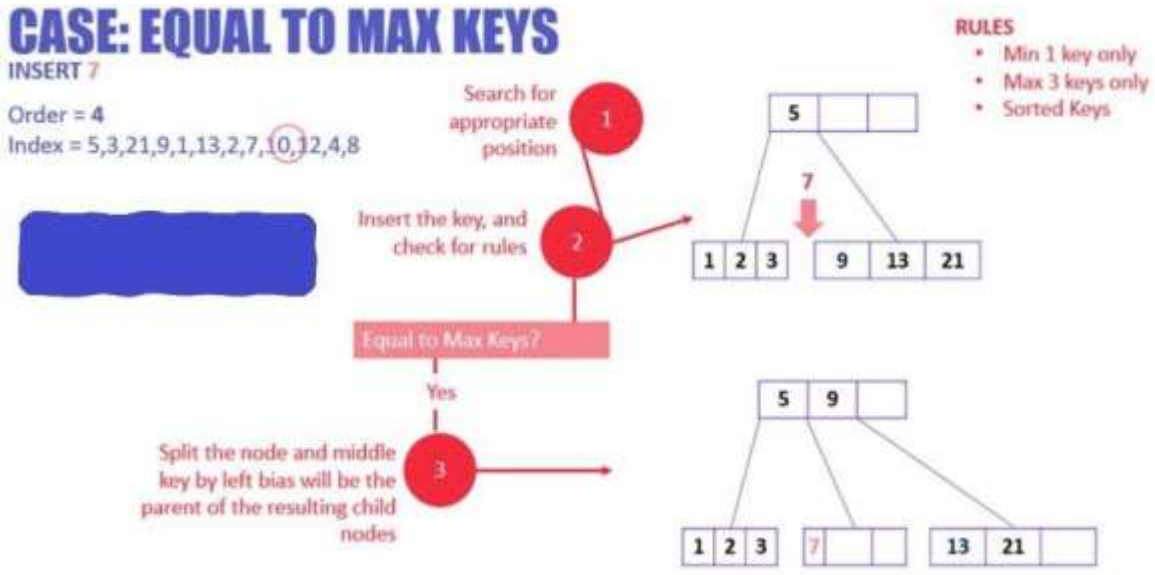
In the above example:

- The node has reached the max number of keys
- The node will split, and the middle key will become the root node of the rest two nodes.
- In case of even number of keys, the middle node will be selected by left bias or right bias.



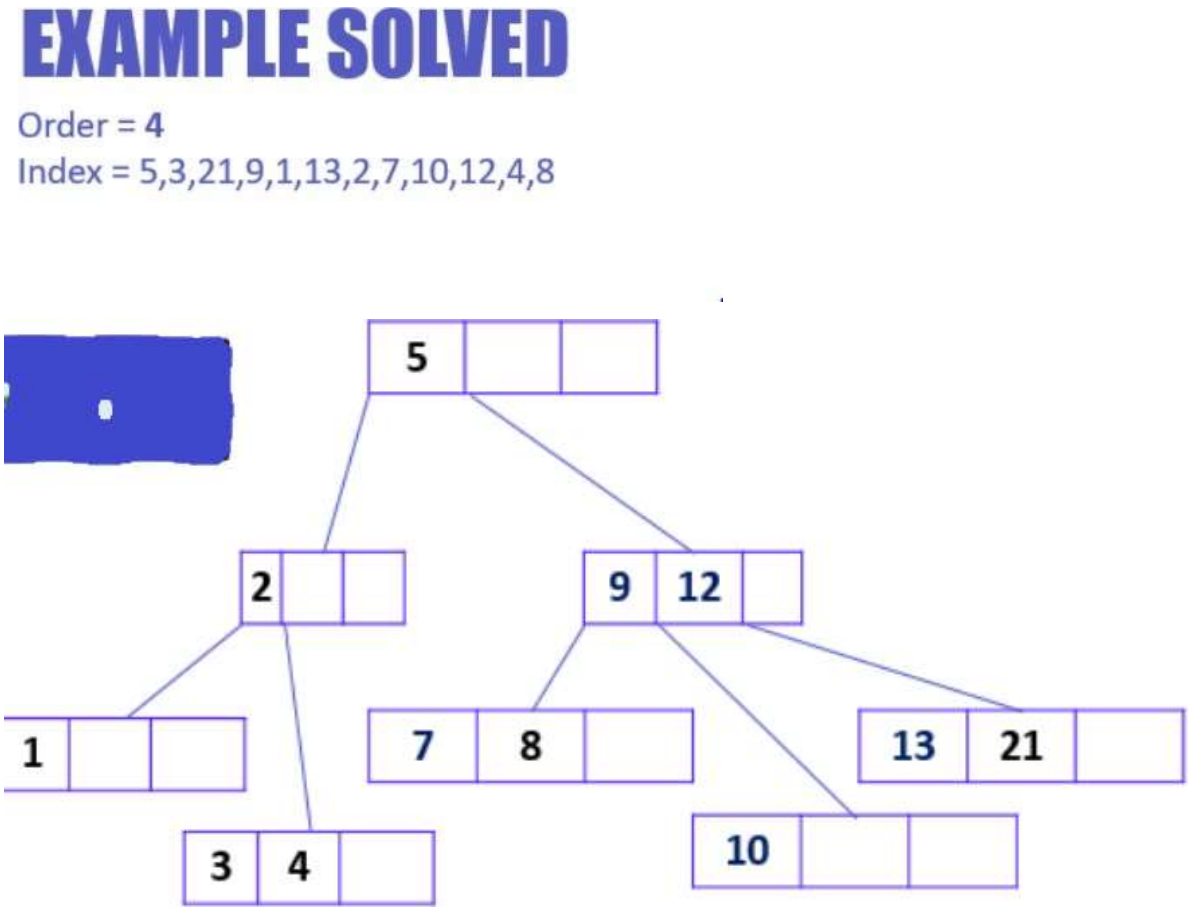
Similarly, 13 and 2 can be inserted easily in the node as they fulfill less than max keys rule for the nodes.

In the above example:



- The node has keys equal to max keys.
- The key is inserted to the target node, but it violates the rule of max keys.
- The target node is split, and the middle key by left bias is now the parent of the new child nodes.
- The new nodes are arranged in ascending order.

Similarly, based on the above rules and cases, the rest of the values can be inserted easily in the B Tree.



Delete Operation

The delete operation has more rules than insert and search operations.

The following algorithm applies:

- Run the search operation and find the target key in the nodes
- Three conditions applied based on the location of the target key, as explained in the following sections

If the target key is in the leaf node

- Target is in the leaf node, more than min keys.
  - Deleting this will not violate the property of B Tree
- Target is in leaf node, it has min key nodes
  - Deleting this will violate the property of B Tree
  - Target node can borrow key from immediate left node, or immediate right node (sibling)
  - The sibling will say **yes** if it has more than minimum number of keys

    - The key will be borrowed from the parent node, the max value will be transferred to a parent, the max value of the parent node will be transferred to the target node, and remove the target value
- Target is in the leaf node, but no siblings have more than min number of keys
    - Search for key
    - Merge with siblings and the minimum of parent nodes
    - Total keys will be now more than min
    - The target key will be replaced with the minimum of a parent node

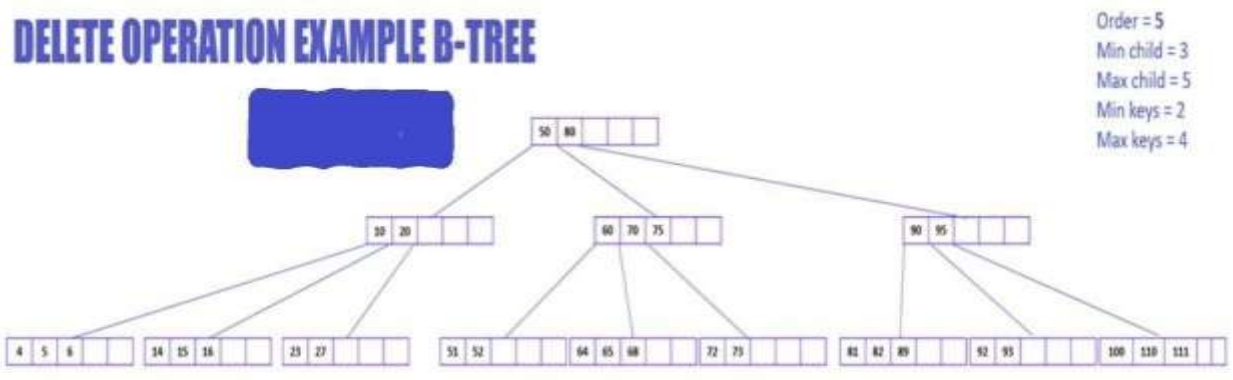
If the target key is in an internal node

- Either choose, in- order predecessor or in-order successor
- In case the of in-order predecessor, the maximum key from its left subtree will be selected
- In case of in-order successor, the minimum key from its right subtree will be selected
- If the target key's in-order predecessor has more than the min keys, only then it can replace the target key with the max of the in-order predecessor
- If the target key's in-order predecessor does not have more than min keys, look for in-order successor's minimum key.
- If the target key's in-order predecessor and successor both have less than min keys, then merge the predecessor and successor.

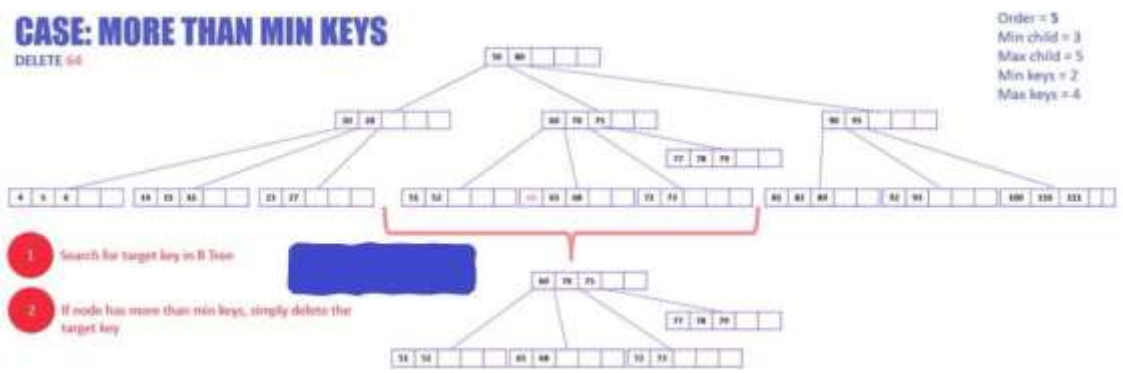
If the target key is in a root node

- Replace with the maximum element of the in-order predecessor subtree
- If, after deletion, the target has less than min keys, then the target node will borrow max value from its sibling via sibling's parent.
- The max value of the parent will be taken by a target, but with the nodes of the max value of the sibling.

Now, let's understand the delete operation with an example.

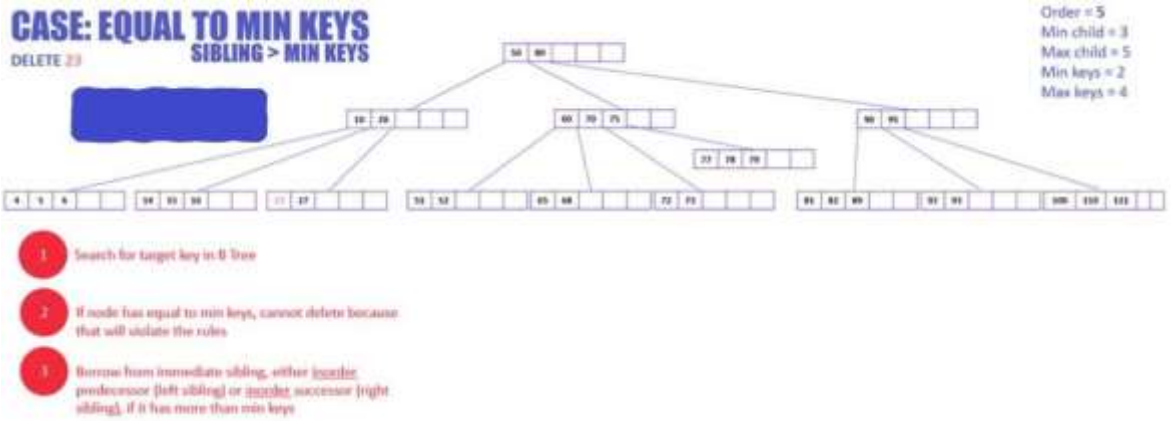


The above diagram displays different cases of delete operation in a B-Tree. This B-Tree is of order 5, which means that the minimum number of child nodes any node can have is 3, and the maximum number of child nodes any node can have is 5. Whereas the minimum and a maximum number of keys any node can have are 2 and 4, respectively.



In the above example:

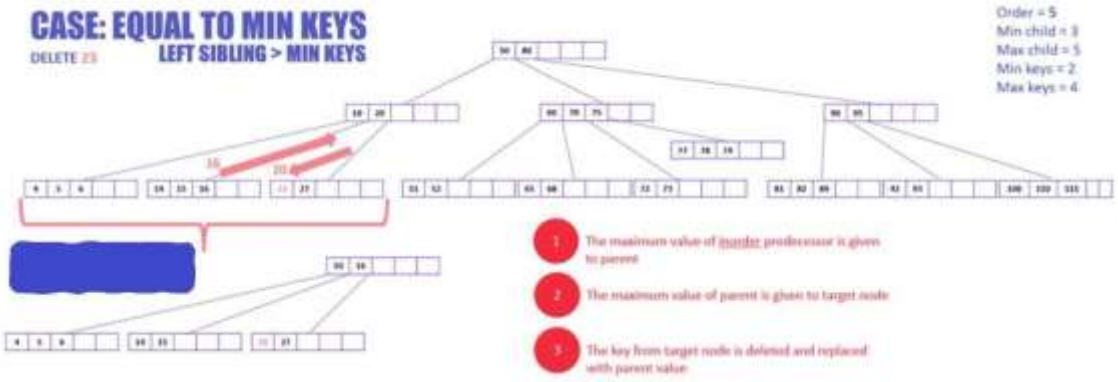
- The target node has the target key to delete
- The target node has keys more than minimum keys
- Simply delete the key



In the above example:

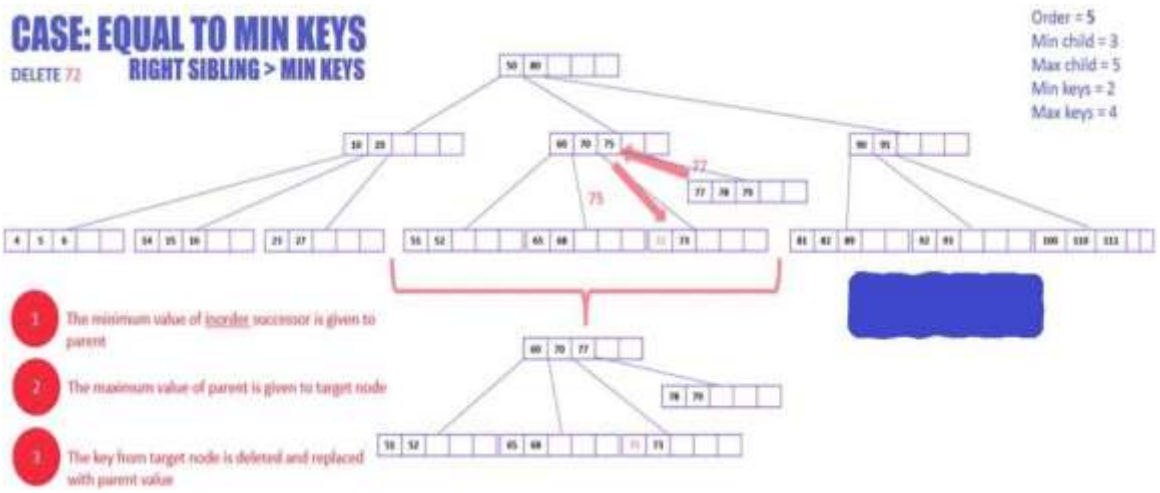
- The target node has keys equal to minimum keys, so cannot delete it directly as it will violate the conditions

Now, the following diagram explains how to delete this key:



- The target node will borrow a key from immediate sibling, in this case, in-order predecessor (left sibling) because it does not have any in-order successor (right sibling)
- The maximum value of the inorder predecessor will be transferred to the parent, and the parent will transfer the maximum value to the target node (see the diagram below)

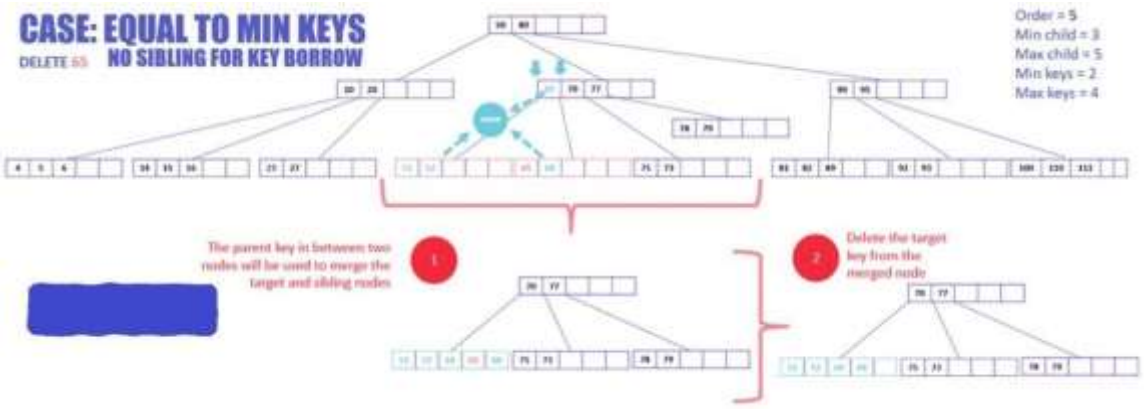
The following example illustrates how to delete a key that needs a value from its in-order successor.



- The target node will borrow a key from immediate sibling, in this case, in-order successor (right sibling) because it's in-order predecessor (left sibling) has keys equal to minimum keys.
- The minimum value of the in-order successor will be transferred to the parent, and the parent will transfer the maximum value to the target node.

In the example below, the target node does not have any sibling that can give its key to the target node. Therefore, merging is required.

See the procedure of deleting such a key:



- Merge the target node with any of its immediate siblings along with parent key

- The key from the parent node is selected that siblings in between the two merging nodes
- Delete the target key from the merged node

Delete Operation Pseudo Code

```
private int removeBiggestElement()
{
  if (root has no child)
    remove and return the last element
  else {
    answer = subset[childCount-1].removeBiggestElement()
    if (subset[childCount-1].dataCount < MINIMUM)
      fixShort (childCount-1)
    return answer
  }
}
```

Output:

The biggest element is deleted from the B-Tree.

Summary:

- B Tree is a self-balancing data structure for better search, insertion, and deletion of data from the disk.
- B Tree is regulated by the degree specified
- B Tree keys and nodes are arranged in ascending order.
- The search operation of B Tree is the simplest one, which always starts from the root and starts checking if the target key is greater or lesser than the node value.
- The insert operation of B Tree is rather detailed, which first finds an appropriate position of insertion for the target key, inserts it, evaluates the validity of B Tree against different cases, and then restructure the B Tree nodes accordingly.
- The delete operation of B Tree first searches for the target key to be deleted, deletes it, evaluates the validity based on several cases like minimum and maximum keys of the target node, siblings, and parent.

B-tree Properties

1. For each node x, the keys are stored in increasing order.
2. In each node, there is a boolean value $x.leaf$ which is true if x is a leaf.
3. If n is the order of the tree, each internal node can contain at most $n - 1$ keys along with a pointer to each child.
4. Each node except root can have at most n children and at least $n/2$ children.
5. All leaves have the same depth (i.e. height-h of the tree).
6. The root has at least 2 children and contains a minimum of 1 key.
7. If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2$, $h \geq \log_t (n+1)/2$.

Searching an element in a B-tree

Searching for an element in a B-tree is the generalized form of searching an element in a Binary Search Tree. The following steps are followed.

1. Starting from the root node, compare k with the first key of the node.
   If k = the first key of the node, return the node and the index.
2. If k.leaf = true, return *NULL* (i.e. not found).
3. If k < the first key of the root node, search the left child of this key recursively.
4. If there is more than one key in the current node and k > the first key, compare k with the next key in the node.
   If k < next key, search the left child of this key (ie. k lies in between the first and the second keys).
   Else, search the right child of the key.
5. Repeat steps 1 to 4 until the leaf is reached.

Algorithm for Searching an Element

```
6. BtreeSearch(x, k)
7.    i = 1
8.    while i ≤ n[x] and k ≥ keyi[x]       // n[x] means number of keys in x node
9.       do i = i + 1
10. if i  n[x] and k = keyi[x]
11.    then return (x, i)
12. if leaf [x]
13.    then return NIL
14. else
15.    return BtreeSearch(ci[x], k)
```

Insertion Operation

1. If the tree is empty, allocate a root node and insert the key.
2. Update the allowed number of keys in the node.
3. Search the appropriate node for insertion.
4. If the node is full, follow the steps below.
5. Insert the elements in increasing order.
6. Now, there are elements greater than its limit. So, split at the median.
7. Push the median key upwards and make the left keys as a left child and the right keys as a right child.
8. If the node is not full, follow the steps below.
9. Insert the node in increasing order.

Algorithm for Inserting an Element

```
BreeInsertion(T, k)
r  root[T]
if n[r] = 2t - 1
   s = AllocateNode()
   root[T] = s
   leaf[s] = FALSE
   n[s] <- 0
   c1[s] <- r
   BtreeSplitChild(s, 1, r)
   BtreeInsertNonFull(s, k)
```

```
else BtreeInsertNonFull(r, k)
BtreeInsertNonFull(x, k)
i = n[x]
if leaf[x]
   while i ≥ 1 and k < keyi[x]
      keyi+1 [x] = keyi[x]
      i = i - 1
   keyi+1[x] = k
   n[x] = n[x] + 1
else while i ≥ 1 and k < keyi[x]
      i = i - 1
   i = i + 1
   if n[ci[x]] == 2t - 1
      BtreeSplitChild(x, i, ci[x])
      if k &rt; keyi[x]
         i = i + 1
   BtreeInsertNonFull(ci[x], k)
BtreeSplitChild(x, i)
BtreeSplitChild(x, i, y)
z = AllocateNode()
leaf[z] = leaf[y]
n[z] = t - 1
for j = 1 to t - 1
   keyj[z] = keyj+t[y]
if not leaf [y]
   for j = 1 to t
      cj[z] = cj + t[y]
n[y] = t - 1
for j = n[x] + 1 to i + 1
   cj+1[x] = cj[x]
ci+1[x] = z
for j = n[x] to i
   keyj+1[x] = keyj[x]
keyi[x] = keyt[y]
n[x] = n[x] + 1
```

Deletion from a B-tree

In this tutorial, you will learn how to delete a key from a b-tree. Also, you will find working examples of deleting keys from a B-tree in C, C++, Java and Python.

Deleting an element on a B-tree consists of three main events: **searching the node where the key to be deleted exists**, deleting the key and balancing the tree if required.
While deleting a tree, a condition called **underflow** may occur. Underflow occurs when a node contains less than the minimum number of keys it should hold.
The terms to be understood before studying deletion operation are:

1. **Inorder Predecessor**
   The largest key on the left child of a node is called its inorder predecessor.
2. **Inorder Successor**
   The smallest key on the right child of a node is called its inorder successor.

Deletion Operation

Before going through the steps below, one must know these facts about a B tree of degree **m**.

1. A node can have a maximum of m children. (i.e. 3)
2. A node can contain a maximum of $m - 1$ keys. (i.e. 2)
3. A node should have a minimum of $\lceil m/2 \rceil$ children. (i.e. 2)
4. A node (except root node) should contain a minimum of $\lceil m/2 \rceil - 1$ keys. (i.e. 1)

There are three main cases for deletion operation in a B tree.

Case I

The key to be deleted lies in the leaf. There are two cases for it.

1. The deletion of the key does not violate the property of the minimum number of keys a node should hold.

   In the tree below, deleting 32 does not violate the above properties.
2. The deletion of the key violates the property of the minimum number of keys a node should hold. In this case, we borrow a key from its immediate neighboring sibling node in the order of left to right.

   First, visit the immediate left sibling. If the left sibling node has more than a minimum number of keys, then borrow a key from this node.

   Else, check to borrow from the immediate right sibling node.

   In the tree below, deleting 31 results in the above condition. Let us borrow a key from the left sibling node.
3. If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. **This merging is done through the parent node.**

   Deleting 30 results in the above case.

4. The internal node, which is deleted, is replaced by an inorder successor if the right child has more than the minimum number of keys.
5. If either child has exactly a minimum number of keys then, merge the left and the right children.
6. After merging if the parent node has less than the minimum number of keys then, look for the siblings as in Case I.

Case III

1.In this case, the height of the tree shrinks. If the target key lies in an internal node, and the deletion of the key leads to a fewer number of keys in the node (i.e. less than the minimum required), then look

for the inorder predecessor and the inorder successor. If both the children contain a minimum number of keys then, borrowing cannot take place. This leads to Case II(3) i.e. merging the children.

Again, look for the sibling to borrow a key. But, if the sibling also has only a minimum number of keys then, merge the node with the sibling along with the parent. Arrange the children accordingly (increasing order).

```
Delete a value from the node
int delValFromNode(int item, struct BTreeNode *myNode) {
 int pos, flag = 0;
 if (myNode) {
  if (item < myNode->item[1]) {
   pos = 0;
   flag = 0;
  } else {
   for (pos = myNode->count; (item < myNode->item[pos] && pos > 1); pos--)
    ;
   if (item == myNode->item[pos]) {
    flag = 1;
   } else {
    flag = 0;
   }
  }
  if (flag) {
   if (myNode->linker[pos - 1]) {
    copySuccessor(myNode, pos);
    flag = delValFromNode(myNode->item[pos], myNode->linker[pos]);
    if (flag == 0) {
     printf("Given data is not present in B-Tree\n");
    }
   } else {
    removeVal(myNode, pos);
   }
  } else {
   flag = delValFromNode(item, myNode->linker[pos]);
  }
  if (myNode->linker[pos]) {
   if (myNode->linker[pos]->count < MIN)
    adjustNode(myNode, pos);
  }
 }
 return flag;
}

// Delete operaiton
void delete (int item, struct BTreeNode *myNode) {
 struct BTreeNode *tmp;
 if (!delValFromNode(item, myNode)) {
  printf("Not present\n");
  return;
 } else {
  if (myNode->count == 0) {
```

```
    tmp = myNode;
    myNode = myNode->linker[0];
    free(tmp);
   }
  }
  root = myNode;
  return;
}
```

What is a Trie data structure?

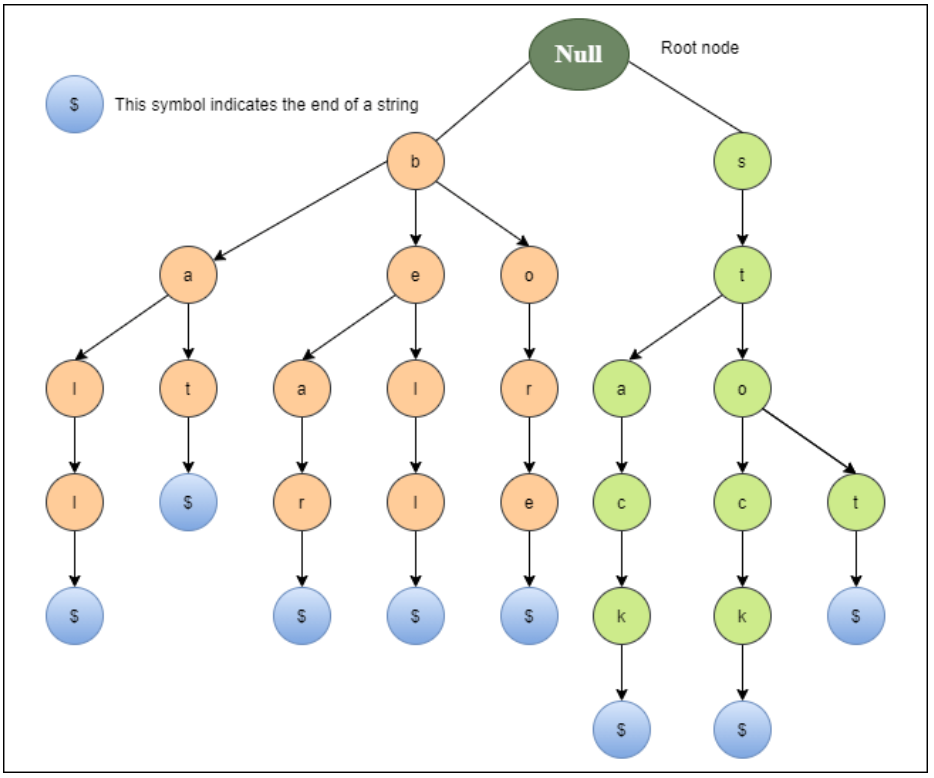
The word "**Trie**" is an excerpt from the word "**retrieval**". Trie is a sorted tree-based data-structure that stores the set of strings. It has the number of pointers equal to the number of characters of the alphabet in each node. It can search a word in the dictionary with the help of the word's prefix. For example, if we assume that all strings are formed from the letters '**a**' to '**z**' in the English alphabet, each trie node can have a maximum of **26** points.

Trie is also known as the digital tree or prefix tree. The position of a node in the Trie determines the key with which that node is connected.

Properties of the Trie for a set of the string:

1. The root node of the trie always represents the null node.
2. Each child of nodes is sorted alphabetically.
3. Each node can have a maximum of **26** children (A to Z).
4. Each node (except the root) can store one letter of the alphabet.

The diagram below depicts a trie representation for the bell, bear, bore, bat, ball, stop, stock, and stack.



Basic operations of Trie

There are three operations in the Trie:

1. Insertion of a node
2. Searching a node
3. Deletion of a node

Insert of a node in the Trie

The first operation is to insert a new node into the trie. Before we start the implementation, it is important to understand some points:

1. Every letter of the input key (word) is inserted as an individual in the Trie_node. Note that children point to the next level of Trie nodes.
2. The key character array acts as an index of children.
3. If the present node already has a reference to the present letter, set the present node to that referenced node. Otherwise, create a new node, set the letter to be equal to the present letter, and even start the present node with this new node.
4. The character length determines the depth of the trie.

Searching a node in Trie

The second operation is to search for a node in a Trie. The searching operation is similar to the insertion operation. The search operation is used to search a key in the trie.

Deletion of a node in the Trie

The Third operation is the deletion of a node in the Trie. Before we begin the implementation, it is important to understand some points:

1. If the key is not found in the trie, the delete operation will stop and exit it.
2. If the key is found in the trie, delete it from the trie.

Applications of Trie

1. Spell Checker

Spell checking is a three-step process. First, look for that word in a dictionary, generate possible suggestions, and then sort the suggestion words with the desired word at the top.

Trie is used to store the word in dictionaries. The spell checker can easily be applied in the most efficient way by searching for words on a data structure. Using trie not only makes it easy to see the word in the dictionary, but it is also simple to build an algorithm to include a collection of relevant words or suggestions.

2. Auto-complete

Auto-complete functionality is widely used on text editors, mobile applications, and the Internet. It provides a simple way to find an alternative word to complete the word for the following reasons.

- It provides an alphabetical filter of entries by the key of the node.
- We trace pointers only to get the node that represents the string entered by the user.
- As soon as you start typing, it tries to complete your input.

3. Browser history

It is also used to complete the URL in the browser. The browser keeps a history of the URLs of the websites you've visited.

Advantages of Trie

1. It can be insert faster and search the string than hash tables and binary search trees.
2. It provides an alphabetical filter of entries by the key of the node.

Disadvantages of Trie

1. It requires more memory to store the strings.
2. It is slower than the hash table.

Hashing

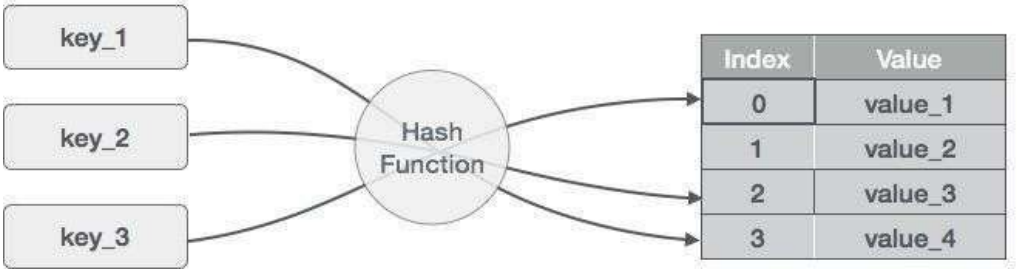
Hash Table

Hash Table is a data structure which store data in associative manner. In hash table, data is stored in array format where each data value has its own unique index value. Access of data becomes very fast if we know the index of desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of size of data. Hash Table uses array as a storage medium and uses hash technique to generate index where an element is to be inserted or to be located from.

Hashing

Hashing is a technique to convert a range o f key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and following items are to be stored. Item are in (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

S.n.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique used here, creates already used index of the array. In such case, we can search the next empty location in the array by looking into the next cell until we found an empty cell. This technique is called linear probing.

S.n.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

B sic Operations

Following are basic primary operations of a hashtable which are following.

- Search – search an element in a hashtable.
- Insert – insert an element in a hashtable.
- delete – delete an element from a hashtable.

D taItem

D fine a data item having some data, and key based on which search is to be conducted in hashtable.

```
struct DataItem { int
   data;
   int key;
};
```

HashMethod

Define a hashing method to compute the hash code of the key of the data item.

Search Operation

Whenever an element is to be searched.

```
int hashCode(int key)
{ return key % SIZE; }
```

Compute the hash code of the keypassed and locate he element u ing that hash code as index in the array. Use linear probing to get element ahead if element not found at computed hash code.

```
struct DataItem *search(int key){
   / / get the hash
   int hashIndex = hashCode(key);
   / / move in array until an empty
   while(hashArray[hashIndex]
      !=NULL){ if(hashArray
      [hashIndex]->key==key) return
         hashArray [hashIndex];
      / / go to next cell
      ++hashIndex;
      //wrap
      around the
      table
      hashIndex
      %= SIZE;
   }
```

Insert Operation

Whenever an element is to be inserted. Compute the hash code of the keypassed and locate the index using that
hashcode as index in the array. Use linear probing for empty location if an element is found at computed hash code.

Delete Operation

Whenever an element is to be deleted. Compute the hash code of the key passed and locate the index using that hash code as index in the array. Use linear probing to get element ahead if an element is not found at computed hash code. When found, store a dummy item there to keep performance of hash table intact.

```
void insert(int key,int data){
struct DataItem *item = (struct DataItem *) malloc(sizeof(struct DataItem )); item->data = data;
item->key = key;
/ / get the hash
int hashIndex = hashCode(key);
/ / move in array until an empty or deleted cell
while (hashArray [hashIndex] != NULL && hashArray [hashIndex]->key != -1){
/ / go to next cell
++hashIndex;
/ / wrap around the table hashIndex %= SIZE;
```

```
}
hashArray [hashIndex] = item;
}
struct DataItem * delete(struct DataItem * item){ int key = item->key;
/ / get the hash
int hashIndex = hashCode(key);
/ / move in array until an empty
while (hashArray [hashIndex] !=NULL ){ if (hashArray [hashIndex]->key == key){
struct DataItem * temp = hashArray [hashIndex];
/ / assign a dummy item at deleted position hashArray [hashIndex] = dummyItem ; return temp;
}
/ / go to next cell
++hashIndex;
/ / wrap around the table hashIndex %= SIZE;
}
return NULL ;
}
```

What is Collision?

Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique.

What are the chances of collisions with large table?

Collisions are very likely even if we have big table to store keys. An important observation is Birthday Paradox. Wi th only 23 persons, the probability that two people have same birthday is 50%.

How to handle Collisions?

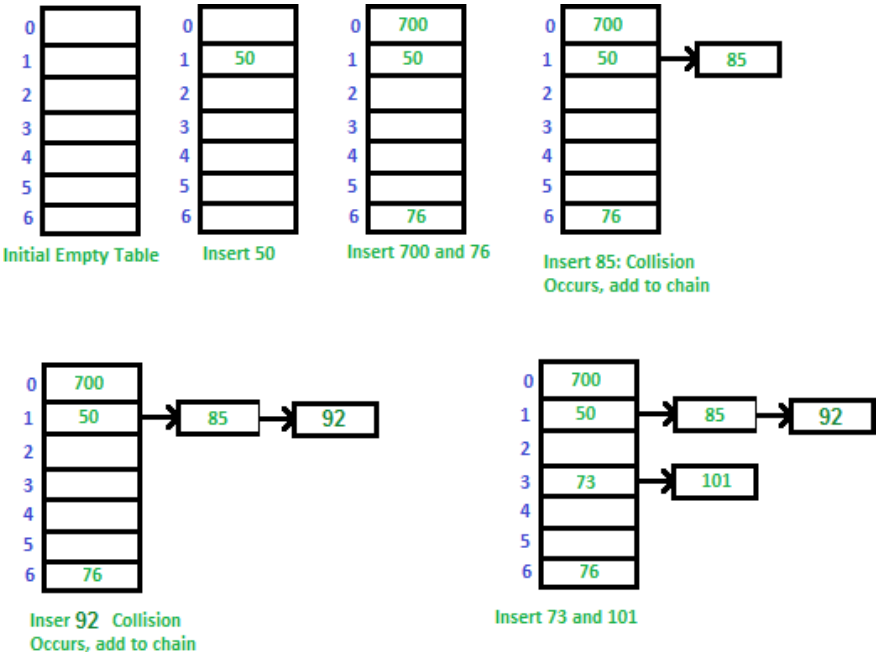
There are mainly two methods to handle collision:

1)Separate Chaining

2)Open Addressing

Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Let us consider a simple hash function as " key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Advantages:

1) Simple to implement.

2) Hash table never fills up, we c an always add more elements to chain.

3) Less sensitive to the hash function or load factors.

4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
- Wastage of Space (Some Parts of hash table are never used)
- If the chain becomes long, then search time can become O(n) in worst case.
- Uses extra space forlinks.

Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys (Note that we can increase table size by copying old data if needed).
Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.
Delete(k): Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted". Insert can insert an item in a deleted slot, but search

doesn't stop at a deleted slot.

Open Addressing is done following ways:

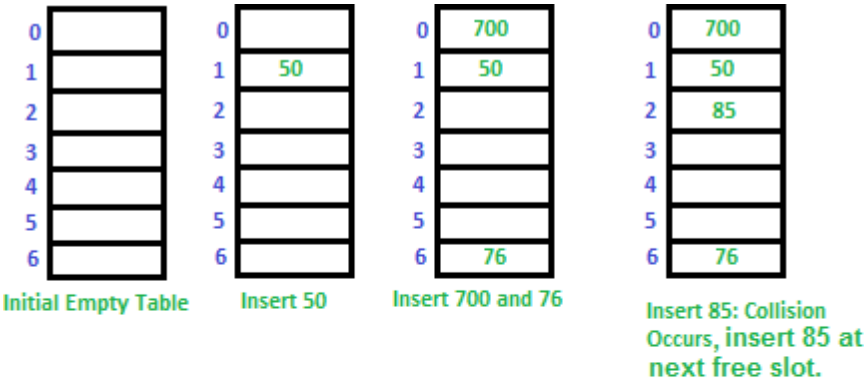
a) Linear Probing:

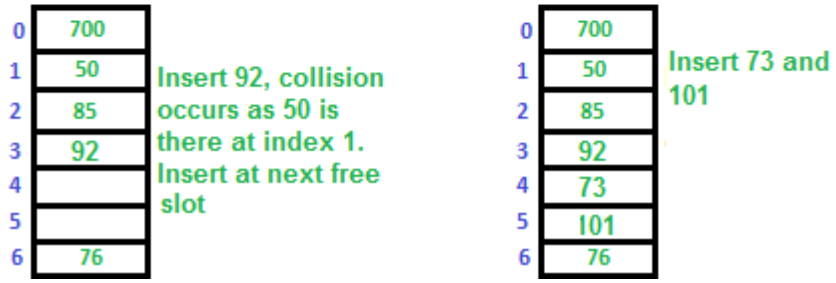
In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also. let hash(x) be the slot index computed using hash function and S be the table size

If slot hash(x) % S is full, then we try (hash(x) + 1) % S

If (hash(x) + 1)% Sis also full, then wetry(hash(x) + 2) % S If (hash(x) + 2)% Sis also full, then wetry(hash(x) + 3) % S

Letusconsiderasimplehashfunctionas"key mod 7" and sequenceofkeysas50,700,76,85, 92, 73, 101.





Clustering:

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

b) Quadratic Probing

We look for i2'th slot in i'th iteration.
lethash(x)bethe slotindexcomputed using hashfunction.

If slot hash(x)% Sis full, then we try (hash(x) + 1*1)% S

If (hash(x) + 1*1)% Sis also full, then wetry(hash(x) + 2*2)% S If (hash(x) + 2*2)% Sis also full, thenwetry(hash(x) + 3*3)% S

c) Double Hashing

We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation. let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

Comparison of above three:

- Linear probing has the best cache performance, but suffers from clustering. One more advantage of Linear probing iseasy to compute.
- Quadratic probing lies between the two in terms of cache performance and clustering.
- Double hashing has poor cache perfo rmance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

Open Addressing vs. Separate Chaining

Advantages of Chaining:

1) Chaining is Simpler toimplement.

2) In chaining, Hash table never fills up, we can always add more elements to chain. In open addressing, table may becomefull.

3) Chaining is Less sensitive to the hash function or load factors.

4) Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

5) Open addressing requires extra care for to avoid clustering and load factor.

Advantages of Open Addressing

1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cacheperformance as ev erything is stored in same table.

2) Wastage of Space (Some Parts of hash table in chaining are never used). In Open addressing, a slot can be used even if an input doesn't map to it.

3) Chaining uses extra space forlinks.

Rehashing:

- It is a closed hashingtechnique.
- If the table gets too full, then the rehashing method builds new table that is about twice as big and scan down the entire original hash table, comparing the new hash value for each element and inserting it in the newtable.
- Rehashing is very expensive since the running time is O(N), since there are N elements to rehash and the table size is roughly2N

Rehashing can be implemented in several ways like

Rehash , as soon as the table is half full

1. Rehash only when an insertion fails

Example:Suppose theelements 13, 15, 24, 6areinsertedintoanopenaddressing hash table of size 7 and if linear probing is used when collision occurs.

0	6
1	15
2	
3	24
4	
5	
6	13

If 23 is inserted, the resulting table will be over 70 percent full.

0	6
1	15
2	23
3	24
4	
5	
6	13

A new table is created. The sizeof the new table is 17, as this is the first prime number that is twice as large as the old table size.

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	1
16	

Advantages

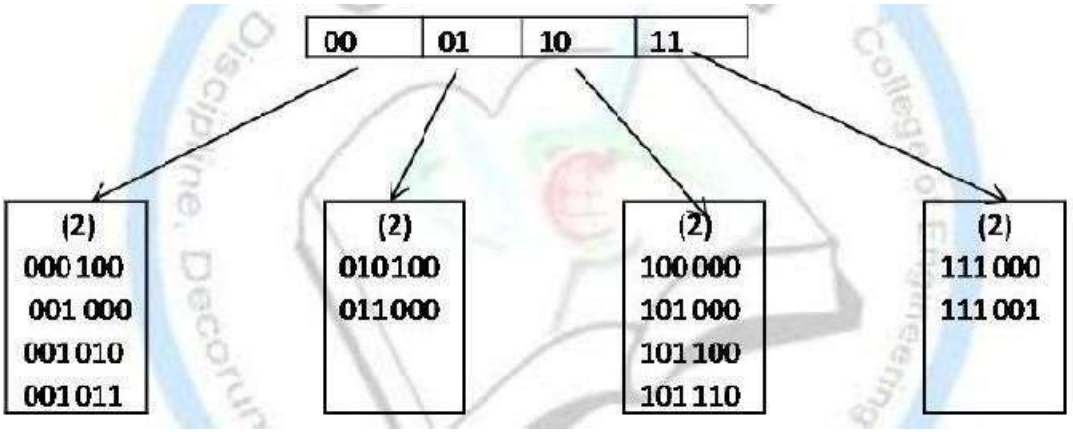
- Programmer doesn't worry about the table size
- Simple to implement

Extendible Hashing:

- When open addressing or separate hashing is used, collisions could cause several blocks to be examined during a Find operation,even for a well distributed hash table.
- Furthermore , when the table gets toofull, an extremely expensive rehashing step must be performed, which requires O(N) disk accesses.
- These problems can be avoided by using extendible hashing.
- Extendible hashing us es a tree to insert keys into the hash table.

Example

- Consider the key consists of several 6 bit integers.
- The root of the "tree" contains 4 pointers determined by the leading 2 bits.
- Ineach leaf the first 2bits are identified and indicated in parenthes is.
- D represents the number of bits used by the root(directory)
- The number of entries in the directory is 2^D

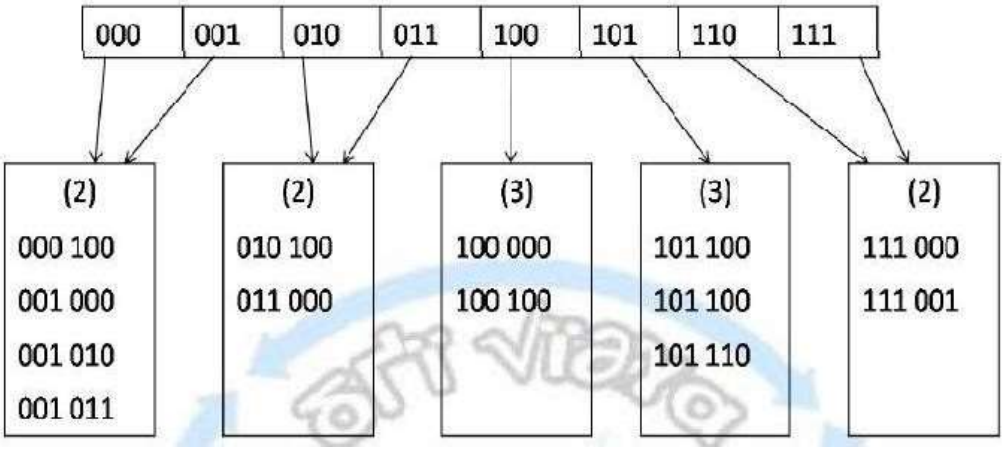


Suppose to insert the key 100100.

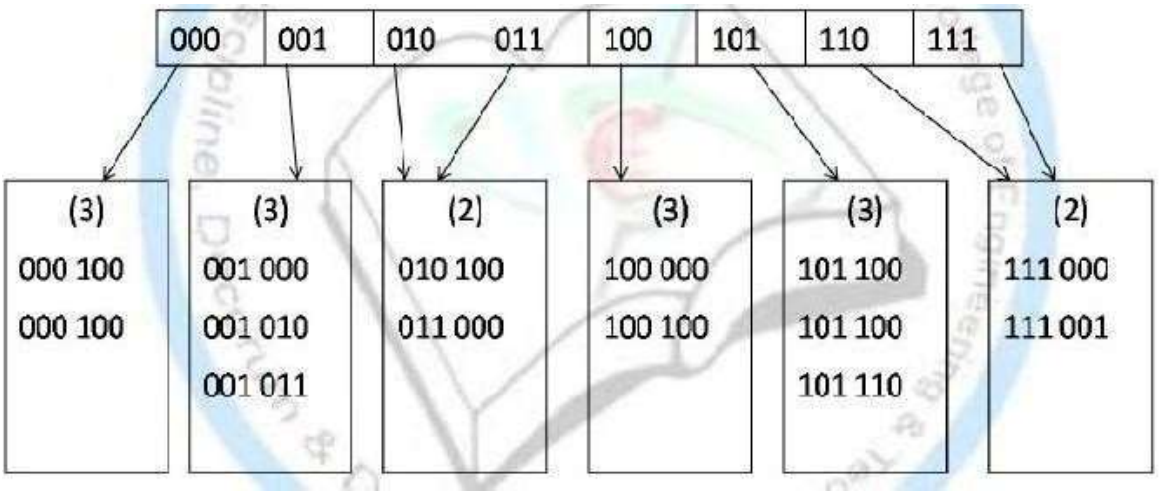
This would go to the third leaf but as the third leaf is already full.

So split this

leaf intotwo leaves, w hicharenow determined by thefirst threebits. Now the directory size is increased to 3.



Similarly if the key 0001 00 is to be inserted, then the first leaf is split into 2 leaves.



Advantages & Disadvantages:

- **Advantages**

– Provides quick access times for insert and find operations on large databases.

- **Disadvantages**

– This algorithm does not work if there are more than M duplicat

UNIT 4

Graphs

Introduction to Graphs –Operations on graphs – Representation of Graph – Topological Sort - Graph Traversal - Depth first search – Breadth First Search– Minimum Spanning Tree – Prim's Algorithm- Kruskal's Algorithm – Disjoint subsets and Union-Find algorithms– ShortestPath Algorithms – Unweighted Shortest Paths –Dijkstra's Algorithm. Applications of Graphs. Case Study: Representing Google maps, Recommendations on e-commerce websites.

Introduction to Graphs

Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

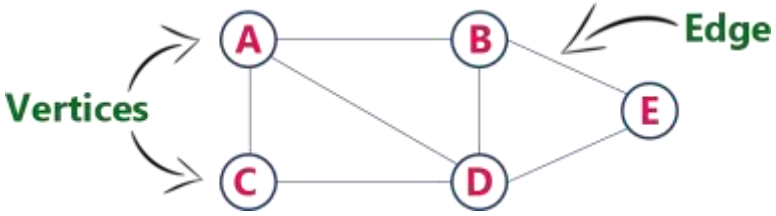
Graph is a collection of vertices and arcs in which vertices are connected with arcs

Graph is a collection of nodes and edges in which nodes are connected with edges

Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Example

The following is a graph with 5 vertices and 6 edges.
This graph G can be defined as G = (V , E)
Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Graph Terminology

We use the following terms in graph data structure...

Vertex

Individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

Edge

An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (startingVertex, endingVertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

1. Undirected Edge - An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

2. Directed Edge - A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. Weighted Edge - A weighted egde is a edge with value (cost) on it.

Undirected Graph

A graph with only undirected edges is said to be undirected graph.

Directed Graph

A graph with only directed edges is said to be directed graph.

Mixed Graph

A graph with both undirected and directed edges is said to be mixed graph.

End vertices or Endpoints

The two vertices joined by edge are called end vertices (or endpoints) of that edge.

Origin

If a edge is directed, its first endpoint is said to be the origin of it.

Destination

If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

Incident

Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

Self-loop

Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Path

A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

Representation of Graph:

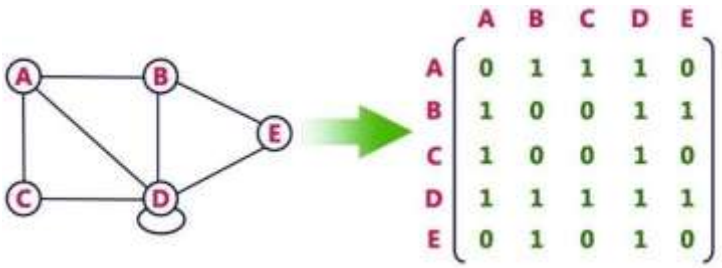
Graph data structure is represented using following representations

1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

Adjacency Matrix

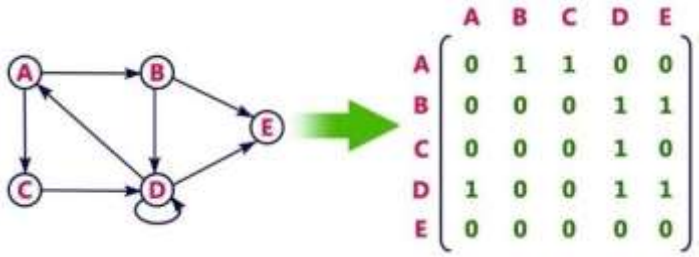
The graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

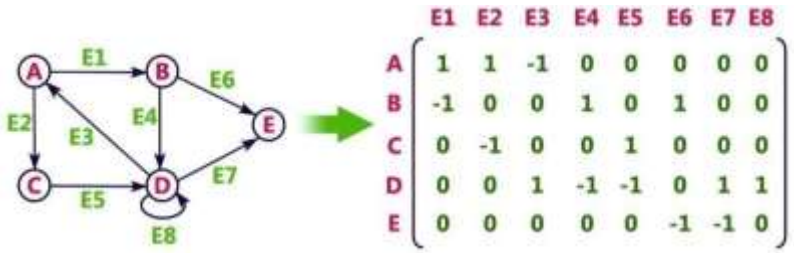
Directed graph representation.



Incidence Matrix

The graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. Inthis matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents thatthe row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

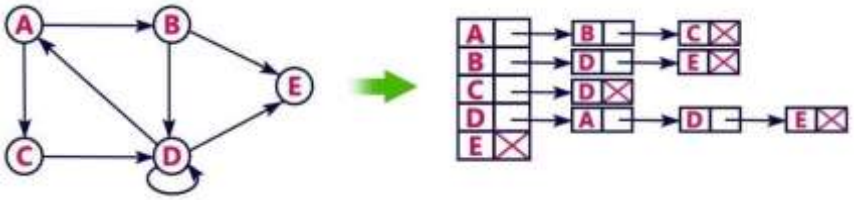
For example, consider the following directed graph representation...



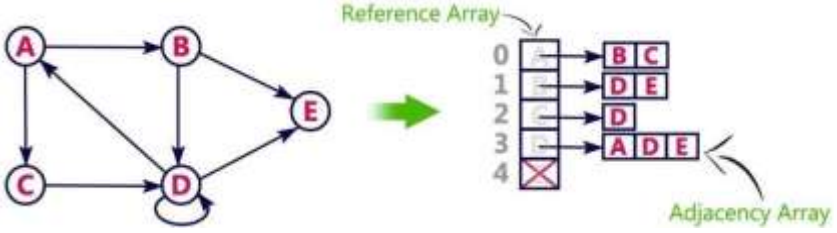
Adjacency List

Every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..



Topological Sort

What is topological sort

Topological Sort is a linear ordering of the vertices in such a way that if there is an edge in the DAG going from vertex 'u' to vertex 'v', then 'u' comes before 'v' in the ordering.

CONCEPT of Topological Sort

The topological sort algorithm takes a directed graph and returns an array of the nodes where each node appears before all the nodes it points to. The ordering of the nodes in the array is called a topological ordering. Since node 1 points to nodes 2 and 3, node 1 appears before them in the ordering.

WHY topological sort

Scheduling jobs from given dependencies among Jobs. For example, if some job requires the dependency of some other job, then we can use topological sorting. Determining the order of compilation tasks to perform in makefiles, data serializations and resolving symbol dependencies in linkers.

WHEN TO USE topological sort

It is important to note that-

- Topological Sorting is possible if and only if the graph is a **Directed Acyclic Graph**[Each node of it contains a unique value. It does not contain any cycles in it, hence called **Acyclic**.]
- There may exist multiple different topological orderings for a given directed acyclic graph.

API of topological sort

Python Flask RestFul API

Advantage:

- Requires linear time and linear space to perform.
- Effective in detecting cyclic dependencies.
- Can efficiently find feedback loops that should not exist in a combinational circuit.
- Can be used to find the shortest path between two nodes in a DAG in linear time.

Disadvantage:

- Topological sort is not possible for a graph that is not directed and acyclic.

Complexity Analysis:

- **Time Complexity:** O(V+E).

Auxiliary space: O(V).

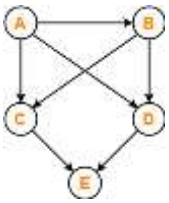
Applications of Topological Sort-

Scheduling jobs from the given dependencies among jobs

- Instruction Scheduling
- Determining the order of compilation tasks to perform in makefiles
- Data Serialization

Example 1:

Find the number of different topological orderings possible for the given graph-

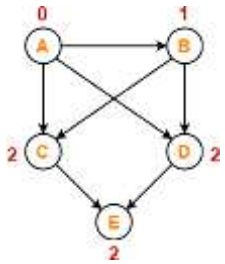


Solution-

The topological orderings of the above graph are found in the following steps-

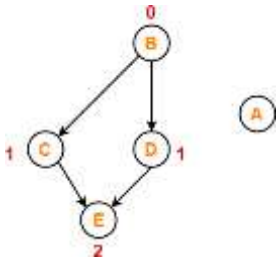
Step-01:

Write in-degree of each vertex-



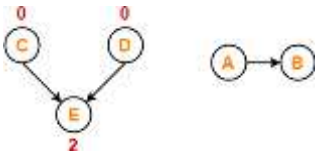
Step-02:

- Vertex-A has the least in-degree.
- So, remove vertex-A and its associated edges.
- Now, update the in-degree of other vertices.



Step-03:

- Vertex-B has the least in-degree.
- So, remove vertex-B and its associated edges.
- Now, update the in-degree of other vertices.



Step-04:

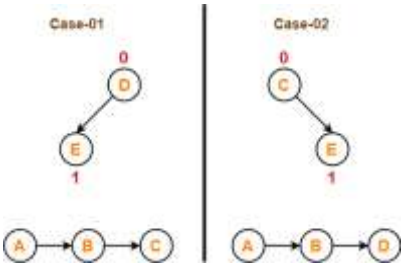
There are two vertices with the least in-degree. So, following 2 cases are possible-

In case-01,

- Remove vertex-C and its associated edges.
- Then, update the in-degree of other vertices.

In case-02,

- Remove vertex-D and its associated edges.
- Then, update the in-degree of other vertices.



Step-05:

Now, the above two cases are continued separately in the similar manner.

In case-01,

- Remove vertex-D since it has the least in-degree.
- Then, remove the remaining vertex-E.

In case-02,

- Remove vertex-C since it has the least in-degree.
- Then, remove the remaining vertex-E.

Conclusion-

For the given graph, following 2 different topological orderings are possible-

- A B C D E
- A B D C E

ALGORITHM:

Steps involved in finding the topological ordering of a DAG:
Step-1: Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited nodes as 0.
Step-2: Pick all the vertices with in-degree as 0 and add them into a queue (Enqueue operation)
Step-3: Remove a vertex from the queue (Dequeue operation) and then.

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighbouring nodes.
3. If in-degree of a neighbouring nodes is reduced to zero, then add it to the queue.

Step 4: Repeat Step 3 until the queue is empty.
Step 5: If count of visited nodes is **not** equal to the number of nodes in the graph then the topological sort is not possible for the given graph.

PSEUDOCODE:

```
topological_sort(N, adj[N][N])
    T = []
    visited = []
    in_degree = []
    for i = 0 to N
        in_degree[i] = visited[i] = 0

    for i = 0 to N
        for j = 0 to N
            if adj[i][j] is TRUE
                in_degree[j] = in_degree[j] + 1

    for i = 0 to N
        if in_degree[i] is 0
            enqueue(Queue, i)
            visited[i] = TRUE

    while Queue is not Empty
        vertex = get_front(Queue)
        dequeue(Queue)
        T.append(vertex)
        for j = 0 to N
            if adj[vertex][j] is TRUE and visited[j] is FALSE
                in_degree[j] = in_degree[j] - 1
                if in_degree[j] is 0
                    enqueue(Queue, j)
                    visited[j] = TRUE
    return T
```

Shortest-Path Algorithms

WHAT IS Shortest Path algorithm?

In data structures, Shortest path problem is a problem of finding the shortest path(s) between vertices of a given graph. Shortest path between two vertices is **a path that has the least cost as compared to all other existing paths**.

CONCEPT OF Shortest Path algorithm.

The shortest path problem is about finding a path between 2 vertices in a graph such that the total sum of the edges weights is minimum.
This problem could be solved easily using **(BFS)** if all edge weights were (1), but here weights can take any value. Three different algorithms are discussed below depending on the use-case.
WHY Shortest Path algorithm?

Our goal is **to send a message between two points in the network in the shortest time possible**. If we know the transmission-time of each computer (the weight of each edge), then we can use a standard shortest-paths algorithm.

WHEN TO USE Shortest Path algorithm?

Shortest path algorithms can be used **to solve word ladder puzzles**. Shortest path problems form the foundation of an entire class of optimization problems that can be solved by a technique called column generation. Examples include vehicle routing problem, survivable network design problem, amongst others.

Common Shortest Path Algorithms

Some common shortest path algorithms are –

- Bellman Ford's Algorithm
- Dijkstra's Algorithm
- Floyd Warshall's Algorithm

1.Dijkstra's Algorithm

*Dijkstra Algorithm is a very famous greedy algorithm.
*It is used for solving the single source shortest path problem.
*one particular source node to all other remaining nodes of the graph.

DIJKSTRA'S Unweighted shortest Path

In unweighted shortest path, all the edges are assigned to 1.

Required Information

i. **known –** Specifies whether the vertex is processed or not. It is set to 1 after it is processed, otherwise 0. Initially all vertices are unknown, so all entries marked as 0.

ii. **dv –** it specifies distance form source to vertex. Initially all vertices are unreachable except for S whose path length is zero.

iii. **pv –** It is book keeping variable, which allow us to print te actual path. i.e., the vertex which makes the changes in dv.

Procedure to find the unweighted shortest path

i. Assign the source node as S and Enqueue S.

ii. Dequeue the vertex S from queue and assign the value of that vertex to be known and then find its adjacency vertices.

iii. If the distance of the adjacent vertices is equal to infinity then change the distance of that vertex as the distance of its source vertex. Increment by 1 and enqueue the vertex.

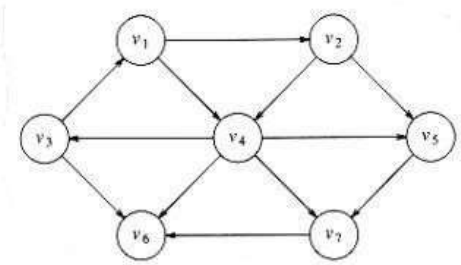
iv. Repeat step ii until the queue becomes empty.

Routine for unweighted shortest path

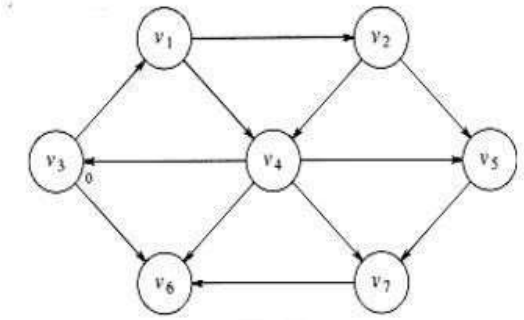
```
void unweighted( Table T )
 {
 Queue Q; Vertex v, w;
Q = CreateQueue( NumVertex );
MakeEmpty( Q );

Enqueue( S, Q ); while( !IsEmpty( Q ) )
```

```
{
v = Dequeue( Q );
T[v].known = True;
for each w adjacent to v
if( T[w].Dist = = INFINITY )
{
T[w].Dist = T[v].Dist + 1;
T[w].path = v;
Enqueue( w, Q );
}}
DisposeQueue( Q ); /* free the memory */

}
```

Example 1:

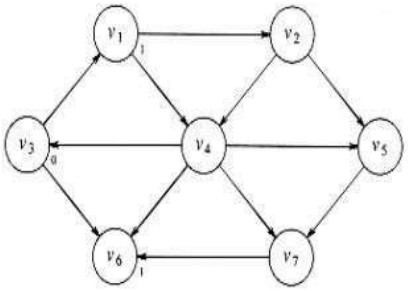


1) v3 is taken as source node and its path length is initialized to 0. v3 is inserted into Q.



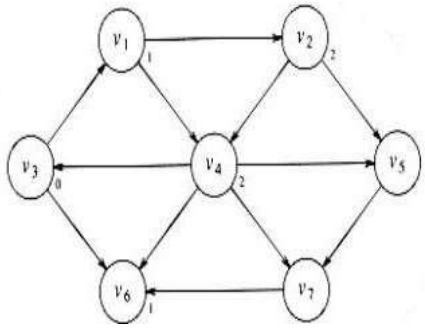
v	Known	d_v	p_v
V_1	0	∞	0
V_2	0	∞	0
V_3	0	0	0
V_4	0	∞	0
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0
Q	V_3		

2) v3 find its adjacent node whose path length is 1. v1, v6 are adjacent nodes to v3 and inserted in to queue.



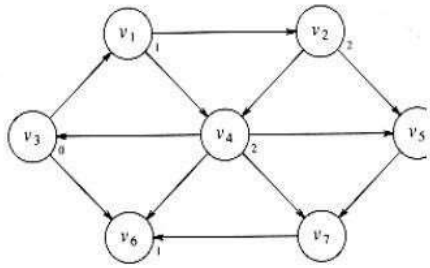
V_3 Dequeued			
v	Known	d_v	p_v
V_1	0	1	V_3
V_2	0	∞	0
V_3	1	0	0
V_4	0	∞	0
V_5	0	∞	0
V_6	0	1	V_3
V_7	0	∞	0
Q	V_1, V_6		

3) Find the adjacent node for v1. v2 and v4 are adjacent node for v1, v2 and v4 inserted into the queue.



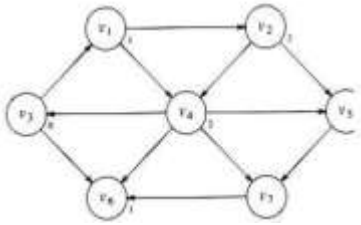
V_1 Dequeued			
v	Known	d_v	p_v
V_1	1	1	V_3
V_2	0	2	V_1
V_3	1	0	0
V_4	0	2	V_1
V_5	0	∞	0
V_6	0	1	V_3
V_7	0	∞	0
Q	V_6, V_2, V_4		

4) No adjacent vertices for v6. No change in path value for all vertices.



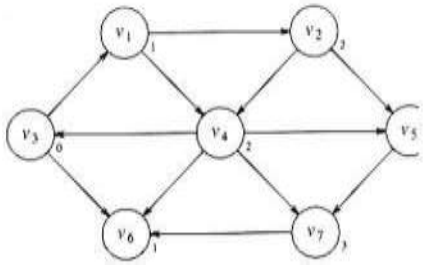
V_6 Dequeued			
v	Known	d_v	p_v
V_1	1	1	V_3
V_2	0	2	V_1
V_3	1	0	0
V_4	0	2	V_1
V_5	0	∞	0
V_6	1	1	V_3
V_7	0	∞	0
Q	V_2, V_4		

5) Find the adjacent vertices for v2. v4 and v5 are adjacent nodes to v2 and inserted into queue.



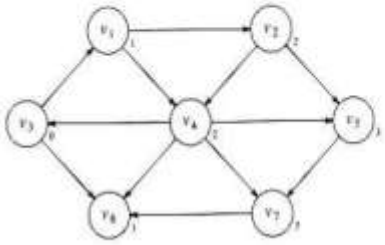
V_2 Dequeued			
v	Known	d_v	p_v
V_1	1	1	V_3
V_2	1	2	V_1
V_3	1	0	0
V_4	0	2	V_1
V_5	0	3	V_2
V_6	1	1	V_3
V_7	0	∞	0
Q	V_4, V_5		

6) Find the adjacent vertices for v4. v5, v6 and v7 are adjacent vertices. Minimum path length for v7 is 3. v7 is inserted into queue.



V_4 Dequeued			
v	Known	d_v	p_v
V_1	1	1	V_3
V_2	1	2	V_1
V_3	1	0	0
V_4	1	2	V_1
V_5	0	3	V_2
V_6	1	1	V_3
V_7	0	3	V_4
Q	V_5, V_7		

7) An adjacent vertex for v5 is v7. Already found the minimum path length from v3 to v7 is 3. So no change in dv and pv.



V_5 Dequeued			
v	Known	d_v	p_v
V_1	1	1	V_3
V_2	1	2	V_1
V_3	1	0	0
V_4	1	2	V_1
V_5	1	3	V_2
V_6	1	1	V_3
V_7	0	3	V_4
Q	V_7		

8) An adjacent vertex for v7 is v6. Already found the minimum path length from v3 to v6 is 1. So no change in dv and pv.

V_7 Dequeued			
v	Known	d_v	p_v
V_1	1	1	V_3
V_2	1	2	V_1
V_3	1	0	0
V_4	1	2	V_1
V_5	1	3	V_2
V_6	1	1	V_3
V_7	1	3	V_4
Q	Empty		

Algorithm Analysis
Running time is O(|E| + |V|)

Weighted Graph – DIJKSTRA'S

The general method to solve the single source shortest path problem is known as Dijkstra‟s algorithm. It applied to weighted graph.

Procedure
ø It uses greedy technique.

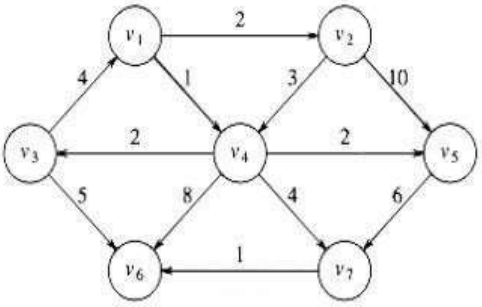
ø It proceeds in stages.

ø It selects a vertex v, which has the smallest d_v among all the unknown vertices and declares the shortest path from s to v is known.

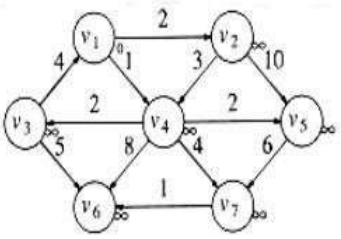
ø The remainder consists of updating the value of d_w.

ø We should $d_w = d_v + C_{v,w}$, if the new value for d_w would an improvement.

Example:



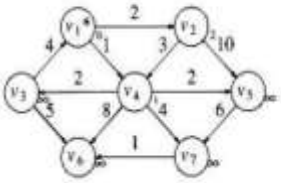
1. v_1 is taken as source.



v	Known	d_v	p_v
V_1	0	0	0
V_2	0	∞	0
V_3	0	∞	0
V_4	0	∞	0
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0

2. Now v1 is known vertex, marked as 1. Its adjacent vertices are v_2, v_4, p_v and d_v values are updated

$T[v_2]$. dist = Min ($T[v_2]$.dist, $T[v_1]$.dist + Cv_1, v_2) = Min (α , 0+2) = 2
$T[v_4]$. dist = Min ($T[v_4]$.dist, $T[v_1]$.dist + Cv_1, v_4) = Min (α , 0+1) = 1



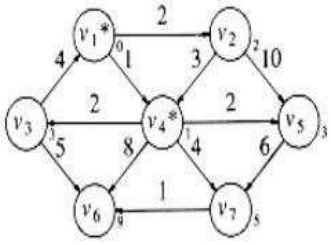
v	Known	d_v	p_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	∞	0
V_4	0	1	V_1
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0

3. Select the vertex with minimum distance away v_2 and v_4 . v_4 is marked as known vertex. Its adjacent vertices are v_3, v_5, v_6 and v_7 .
$T[v_3]$. dist = Min ($T[v_3]$.dist, $T[v_4]$.dist + Cv_4, v_3) = Min (α , 1+2) = 3

$T[v_5]$. dist = Min ($T[v_5]$.dist, $T[v_4]$.dist + Cv_4, v_5) = Min (α , 1+2) = 3

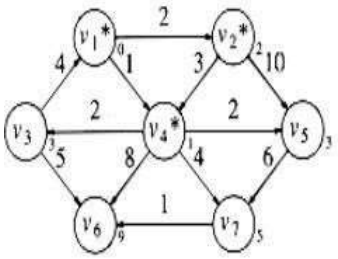
$T[v_6]$. dist = Min ($T[v_6]$.dist, $T[v_4]$.dist + Cv_4, v_6) = Min (α , 1+8) = 9

$T[v_7]$. dist = Min ($T[v_7]$.dist, $T[v_4]$.dist + Cv_4, v_7) = Min (α , 1+4) = 5



v	Known	d_v	p_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4

4. Select the vertex which is shortest distance from source v_1. v_2 is smallest one. v_2 is marked as known vertex. Its adjacent vertices are v_4 ad v_5. The distance from v_1 to v_4 and v_5 through v_2 is more comparing with previous value of d_v. No change in d_v and p_v value.

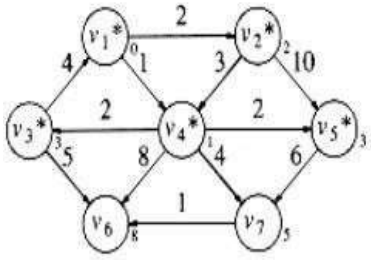


v	Known	d_v	p_v
V_1	1	0	0
V_2	1	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4

5. Select tht smallest vertex from source. v_3 and v_5 are smallest one. Adjacent vertices for v_3 is v_1 and v_6. v_1 is source there is no change in dv and pv

$T[v_6]$. dist = Min ($T[v_6]$.dist, $T[v_3]$.dist + Cv_3, v_6) = Min (9 , 3+5) = 8

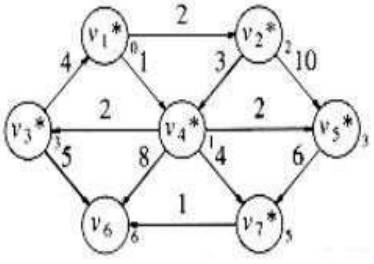
d_v and p_v values are updated. Adjacent vertices for v_5 is v_7. No change in d_v and p_v value.



v	Known	d_v	p_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	0	8	V_3
V_7	0	5	V_4

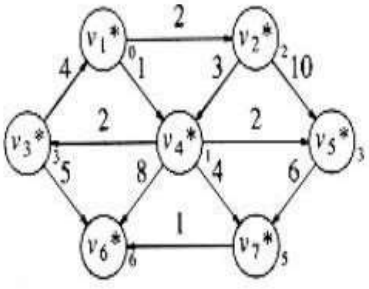
6. Next smallest vertex v_7. Its adjacent vertex is v_6. $T[v_6]$. dist = Min ($T[v_6]$.dist, $T[v_7]$.dist + Cv_7, v_6) = Min (8 , 5+1) = 6

d_v and p_v values are updated.



v	Known	d_v	p_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	0	6	V_7
V_7	1	5	V_4

7. The last vertex v_6 is declared as known. No adjacent vertices for v_6. No updation in the table.



v	Known	d_v	p_v
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	1	6	V_7
V_7	1	5	V_4

The shortest distance from source v_1 to all vertices. v_1 -> v_2 = 2
v_1 -> v_3 = 3 v_1 -> v_4 = 1 v_1 -> v_5 = 3 v_1 -> v_6 = 6 v_1 -> v_7 = 5

Pseudocode for Dijkstra's algorithm

```
void Dijkstra(Table T)
{
Vertex v, w; for( ; ;)
{
v = smallest unknown distance vertex;
if( v = = NotAVertex)
break;
T[v]. kown = True;
for each w adjacent to v if(!T[w].known)
if(T[v].Dist + Cvw < T[w]. Dist)
{/* update w*/
Decrease(T[w]. Dist to T[v].Dist + Cvw);
T[w]. path = v;
}}}
```

Algorithm Analysis

Time complexity of this algorithm

$O(|E| + |V|^2) = O(|V|^2)$

BELLMAN-FORD ALGORITHM:

BELLMAN-FORD(G,w,s)

1. INITIALIZE-SINGLE-SOURCE(G,s)

2. for i = 1 to |G.V|-1
3. for each edge (u,v) ∈ G.E

4. RELAX(u,v,w)
5. for each edge (u,v) ∈ G.E

6. if v.d > u.d + w(u,v)

7. return FALSE

8. return TRUE

INITIALIZE-SINGLE-SOURCE(G,s)

1. for each vertex v ∈ G.V

2. v.d = ∞

3. v.pi = NIL

4. s.d = 0

RELAX(u,v,w)

1. if v.d > u.d + w(u,v)

2. v.d = u.d + w(u,v)
3. v.pi = u

Basically the algorithm works as follows:

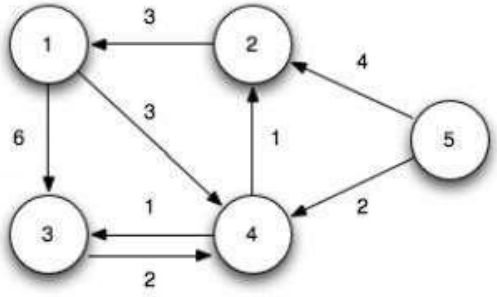
1. Initialize *d*'s, π's, and set *s.d* = 0 ⇒ O(*V*)

2. Loop |*V*|-1 times through all edges checking the relaxation condition to compute minimum distances ⇒ (|*V*|-1)

O(*E*) = O(*VE*)

4. Loop through all edges checking for negative weight cycles which occurs if any of the relaxation conditions fail ⇒ O(*E*)

Example:



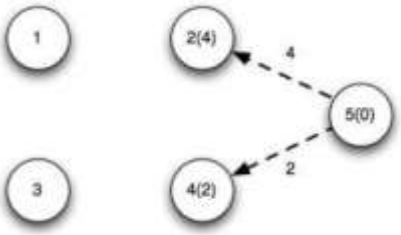
(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

1) Using vertex 5 as the source (setting its distance to 0), we initialize all the other distances to ∞.

(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	∞	∞	∞	∞	0
π	/	/	/	/	/

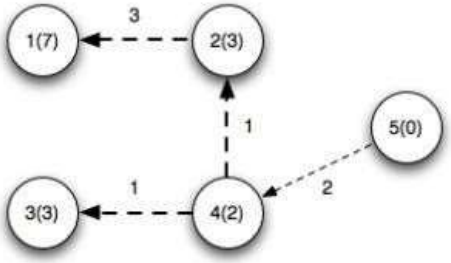
Iteration 1: Edges (u_5,u_2) and (u_5,u_4) relax updating the distances to 2 and 4



(1,3) = 6
(1,4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	∞	4	∞	2	0
π	/	5	/	5	/

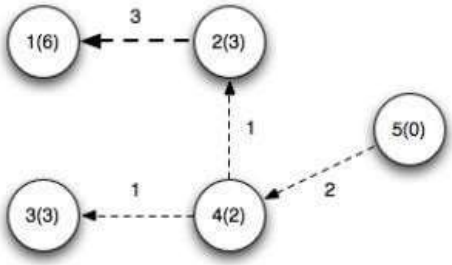
Iteration 2: Edges (u_2,u_1), (u_4,u_2) and (u_4,u_3) relax updating the distances to 1, 2, and 4 respectively. Note edge (u_4,u_2) finds a shorter path to vertex 2 by going through vertex



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	7	3	3	2	0
π	2	4	4	5	/

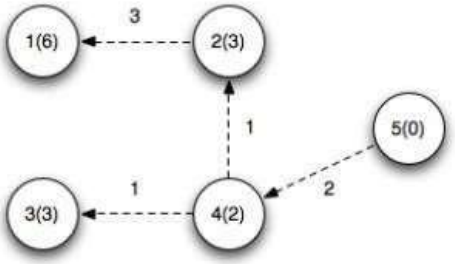
Iteration 3: Edge (u_2,u_1) relaxes (since a shorter path to vertex 2 was found in the previous iteration) updating the distance to 1



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

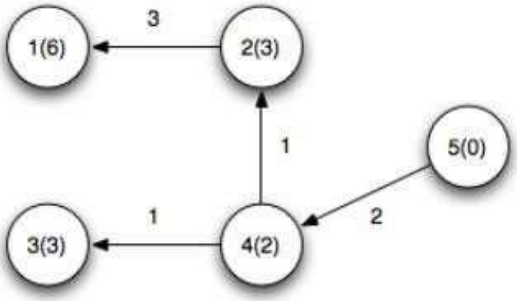
Iteration 4: No edges relax



(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

The final shortest paths from vertex 5 with corresponding distances is



	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

Negative cycle checks: We now check the relaxation condition one additional time for each edge. If any of the checks pass then there exists a negative weight cycle in the graph.

$v_3.d > u_1.d + w(1,3) \Rightarrow 4\ k\ 6 + 6 = 12$ ✓

$v_4.d > u_1.d + w(1,4) \Rightarrow 2\ k\ 6 + 3 = 9$ ✓

$v_1.d > u_2.d + w(2,1) \Rightarrow 6\ k\ 3 + 3 = 6$ ✓

$v_4.d > u_3.d + w(3,4) \Rightarrow 2\ k\ 3 + 2 = 5$ ✓
$v_2.d > u_4.d + w(4,2) \Rightarrow 3\ k\ 2 + 1 = 3$ ✓
$v_3.d > u_4.d + w(4,3) \Rightarrow 3\ k\ 2 + 1 = 3$ ✓
$v_2.d > u_5.d + w(5,2) \Rightarrow 3\ k\ 0 + 4 = 4$ ✓
$v_4.d > u_5.d + w(5,4) \Rightarrow 2\ k\ 0 + 2 = 2$ ✓

(1,3) = 6
(1.4) = 3
(2,1) = 3
(3,4) = 2
(4,2) = 1
(4,3) = 1
(5,2) = 4
(5,4) = 2

= 5 ⇒ the shortest path to vertex 1 is {5,4,2,1}.

FLOYD-WARSHALL ALGORITHM:

Step-01:

Remove all the self loops and parallel edges (keeping the lowest weight edge) from the graph.

- In the given graph, there are neither self edges nor parallel edges.

Step-02:

Write the initial distance matrix.

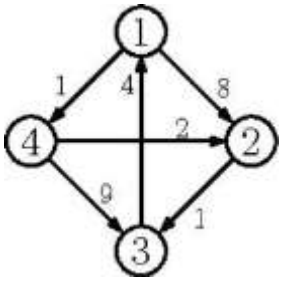
- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.

For vertices having no direct edge between them, distance value = ∞.

Pseudocode:

```
1   for(k = 0; k < n; k++) {
2     for(i = 0; i < n; i++) {
3       for(j = 0; j < n; j++) {
4         A[i, j] = min(A[i, j], A[i, k] + A[k, j])
5       }
6     }
7   }
```

Example:



$$d^{(0)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix}$$

$$d^{(1)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix}, \quad d^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix}$$

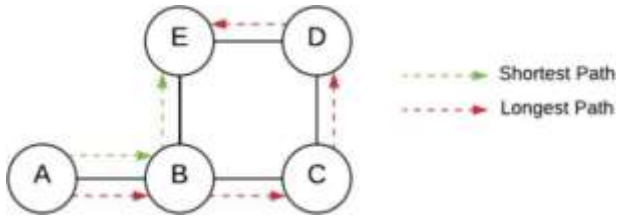
$$d^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}, \quad d^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

$$\text{final} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

Unweighted Graph

Shortest Path in Unweighted Undirected Graph using BFS

Problem: Given an unweighted undirected graph, we have to find the shortest path from the given source to the given destination using the Breadth-First Search algorithm.



The idea is to traverse the graph using Breadth-First Search Traversal until we reach the end node and print the route by tracing back the path to the start node.

How to check whether recached the end node?

Every time we visit a node, we compare it with the end node. If they match, we stop BFS.

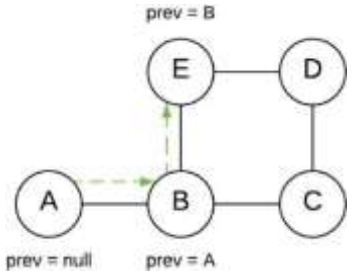
How to stop BFS when we reach the end node?

BFS uses the queue to visit the next node, it runs until the queue is empty.

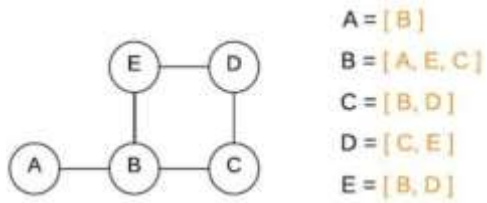
So, we can either clear the queue to stop BFS or use an explicit boolean flag such as end_reached to mark the end of BFS.

How to trace path from end to start node?

To trace the route, we use an extra node property called prev that stores the reference of the preceding node.
Every time we visit a node, we also update its prev value.



Using the prev value, we trace the route back from the end node to the starting node.
Example for the given graph, route = E <- B <- A
Shortest Path in Unweighted Graph (represented using Adjacency List) using BFS
Every vertex (or node) in the graph has an adjacency list that describes the set of its neighbors.



Advantages:-

1) It is used in Google Maps

2) It is used in finding Shortest Path.

3) It is used in geographical Maps

4) To find locations of Map which refers to vertices of graph.

5) Distance between the location refers to edges.

6) It is used in IP routing to find Open shortest Path First.

7) It is used in the telephone network.

Disadvantages:-

1) It do blind search so wastes lot of time while processing.

2) It cannot handle negative edges.

3) This leads to acyclic graphs and most often cannot obtain the right shortest path.

APPLICATIONS

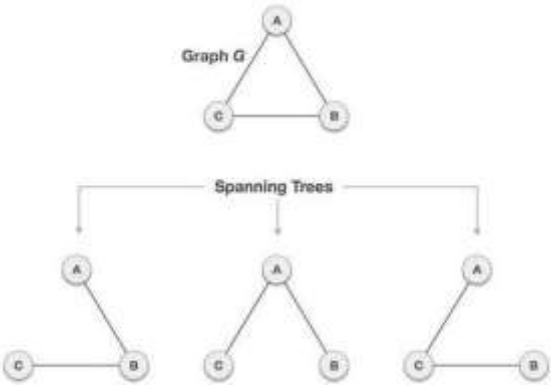
- **For map applications**, it is hugely deployed in measuring the least possible distance and check direction amidst two geographical regions like Google Maps, discovering map locations pointing to the vertices of a graph, calculating traffic and delay-timing, etc.

- **For telephone networks**, this is also extensively implemented in the conducting of data in networking and telecommunication domains for decreasing the obstacle taken place for transmission.
- Wherever addressing the need for shortest path explications either in the domain of robotics, transport, embedded systems, laboratory or production plants, etc, this algorithm is applied.

Minimum Spanning Tree

Spanning Trees:

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.



There are three spanning trees off one complete graph. A complete undirected graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes. In the above addressed example, **n is 3,** hence $\mathbf{3^{3-2} = 3}$ spanning trees are possible.

General Properties of Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

Minimum Spanning-Tree Algorithm

Two most important spanning tree algorithms here

- Kruskal's algorithm
- Prim's algorithm

Prim's Algorithm:

Prim's algorithm to find minimum cost spanning tree uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

How Prim's algorithm works

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

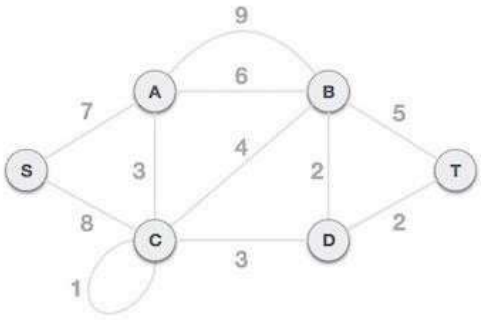
The steps for implementing Prim's algorithm are as follows:

Initialize the minimum spanning tree with a vertex chosen at random.

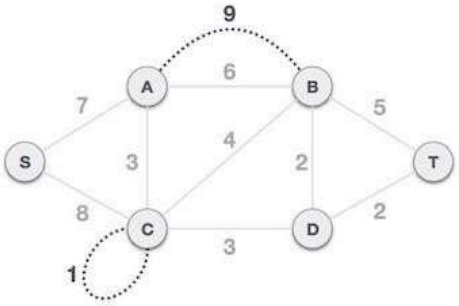
Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree

Keep repeating step 2 until we get a minimum spanning tree

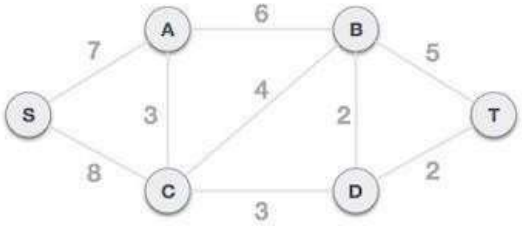
Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

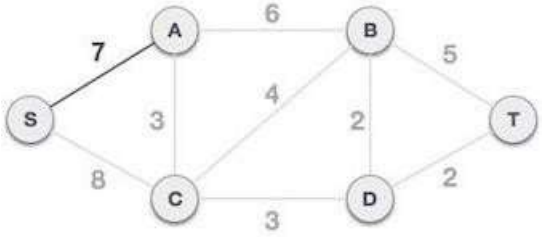


Step 2 - Choose any arbitrary node as root node

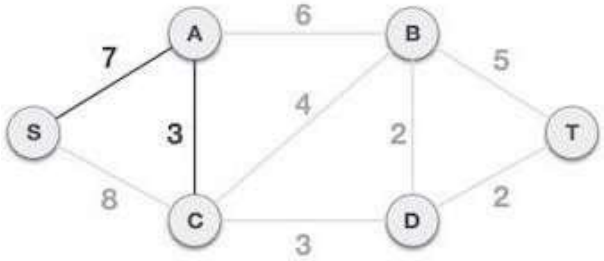
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

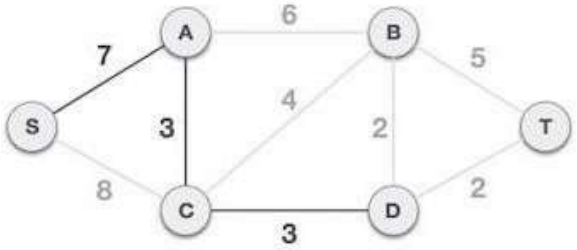
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



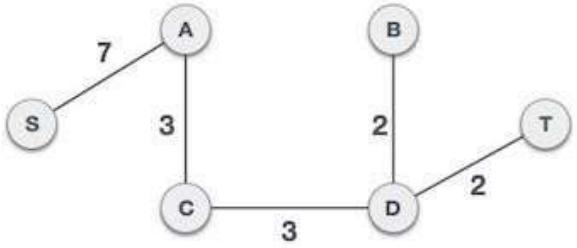
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



We may find that the output spanning tree of the same graph using two different algorithms is same.

Pseudocode

```
T = ∅;
U = { 1 };
while (U ≠ V)
   let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;
   T = T U {(u, v)}
```

U = U U {v}

Prim's Algorithm Complexity

The time complexity of Prim's algorithm is O(E log V).

Prim's Algorithm Application

- Laying cables of electrical wiring
- In network designed
- To make protocols in network cycles

Kruskal's Algorithm:

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

How Kruskal's algorithm works

It falls under a class of algorithms called greedy algorithms that find the local optimum in the hopes of finding a global optimum.

We start from the edges with the lowest weight and keep adding edges until we reach our goal.

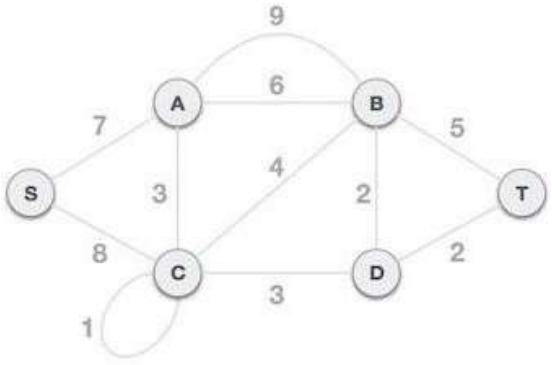
The steps for implementing Kruskal's algorithm are as follows:

Sort all the edges from low weight to high

Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.

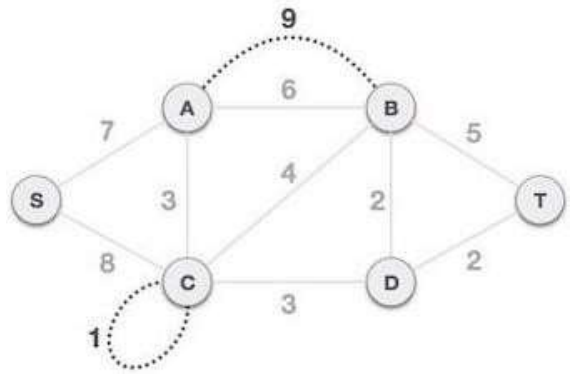
Keep adding edges until we reach all vertices.

To understand Kruskal's algorithm let us consider the following example –

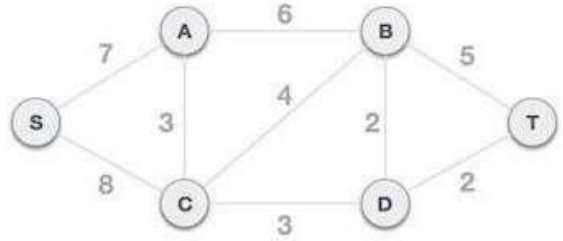


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



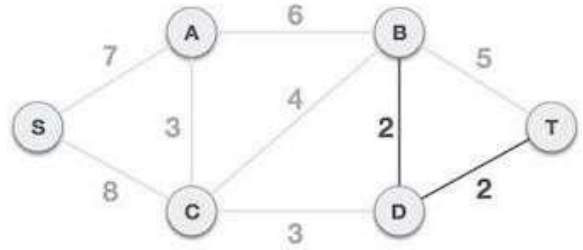
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

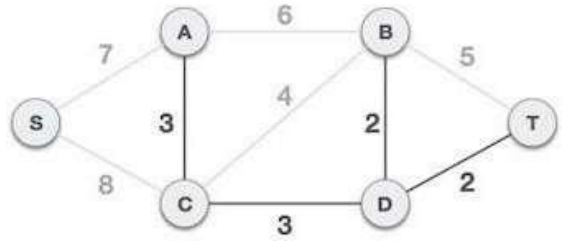
Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

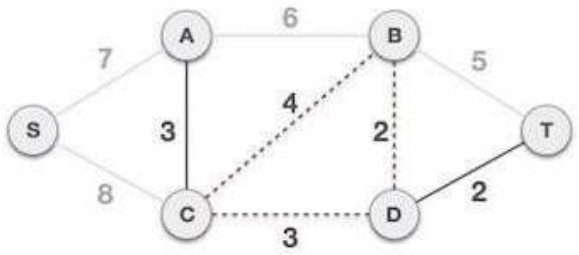


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

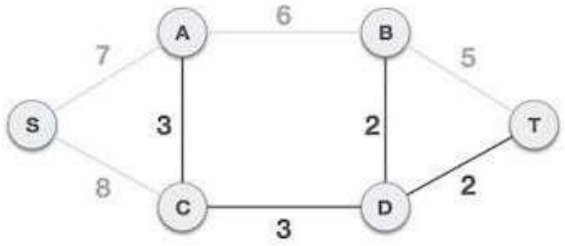
Next cost is 3, and associated edges are A,C and C,D. We add them again –



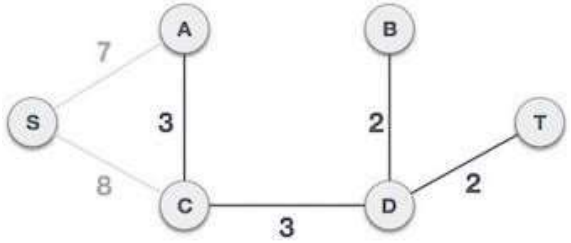
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –



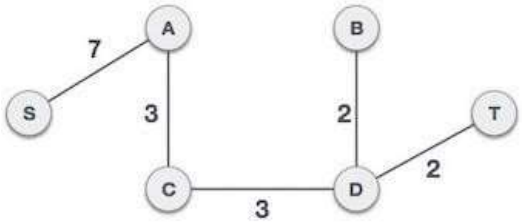
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Pseudocode

```
KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
   MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
   if FIND-SET(u) ≠ FIND-SET(v):
   A = A U {(u, v)}
   UNION(u, v)
return A
```

The time complexity Of Kruskal's Algorithm is: O(E log E)

Kruskal's Algorithm Applications

- In order to layout electrical wiring
- In computer network (LAN connection)

Advantages:

- Spanning trees are used to avoid or prevent broadcast storms in spanning tree protocol when used in networks
- This is also used in providing redundancy for preventing undesirable loops in the spanning tree or network.
- Another advantage is that it provides backup when using spanning tree protocol for the networks, which is activated as this protocol provides various paths where it can choose any open path and closes the other whenever the opened path is not functioning properly.
- A minimum spanning tree is used to find easy paths in maps and is also used for designing networks such as water supply networks or any electrical grid networks.

Disadvantages:

- There are automatic recalculations in spanning trees whenever a new spanning tree is formed, or any new network spanning tree is formed, but the drawback of this is it may cause a network outage sometimes when it modifies automatically.
- There is a protocol called spanning tree protocol; when using the spanning-tree concept in any network, this protocol may block the service if any newer protocols prevent loops while keeping the links, and such newer protocols are TRILL or NPB.

Graph Traversal

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

Depth first search :

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in thegraph to implement DFS traversal.

Algorithm:

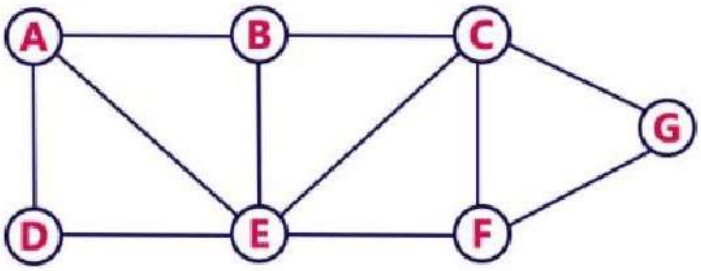
- **Step 1 -** Define a Stack of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
- **Step 3 -** Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4 -** Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5 -** When there is no new vertex to visit then use back tracking and pop one vertex from the stack.

- **Step 6 -** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7 -** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Back tracking is coming back to the vertex from which we reached the current vertex.

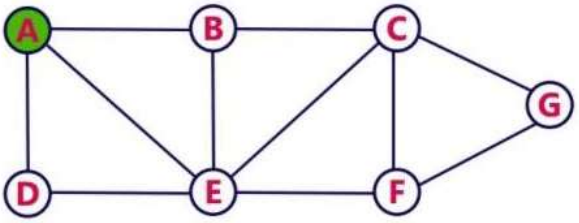
Example

Consider the following example graph to perform DFS traversal



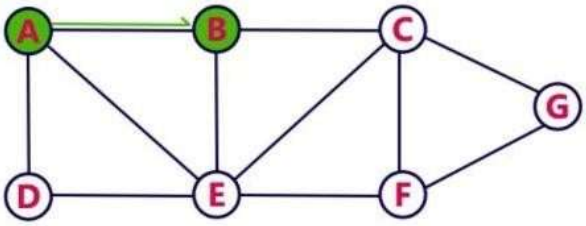
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



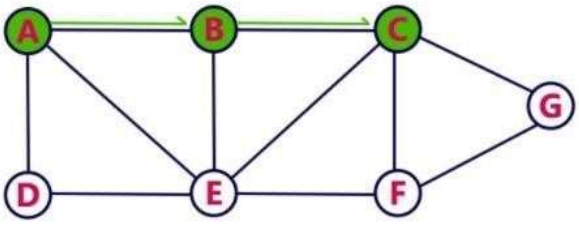
Step 2:

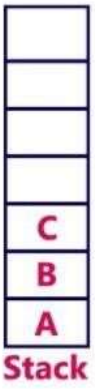
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



Step 3:

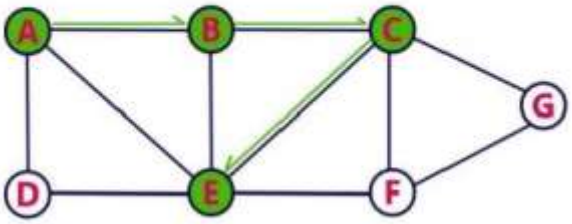
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

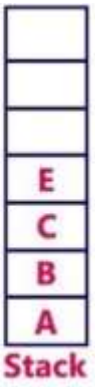




Step 4:

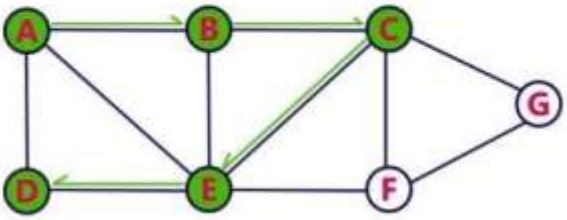
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

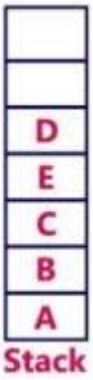




Step 5:

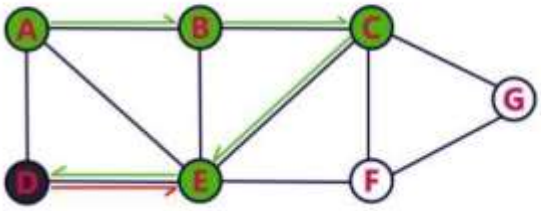
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack





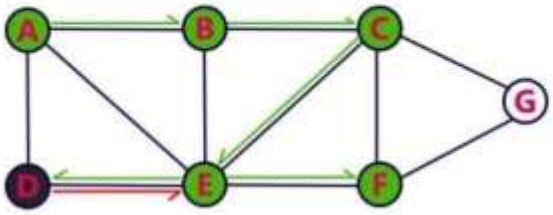
Step 6:

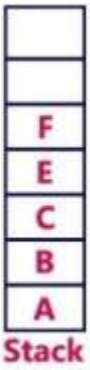
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



Step 7:

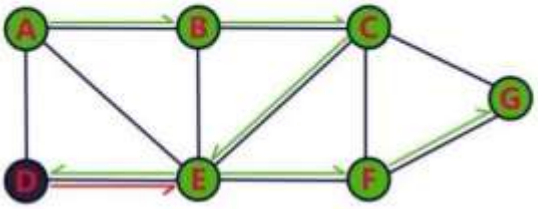
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

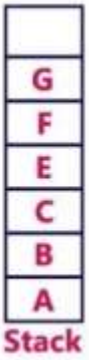




Step 8:

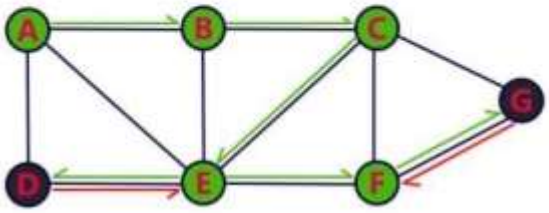
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

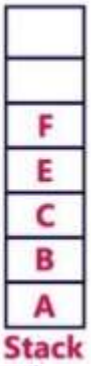




Step 9:

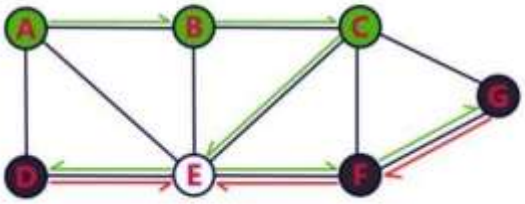
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

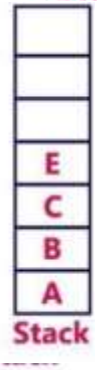




Step 10:

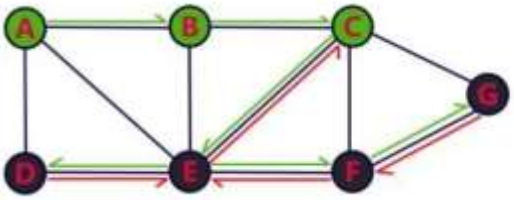
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

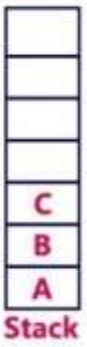




Step 11:

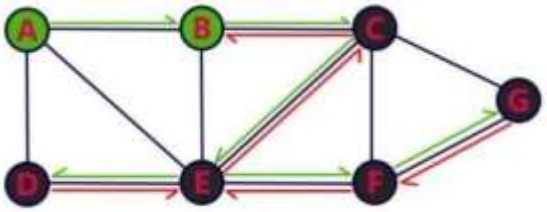
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

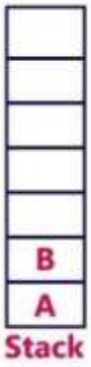




Step 12:

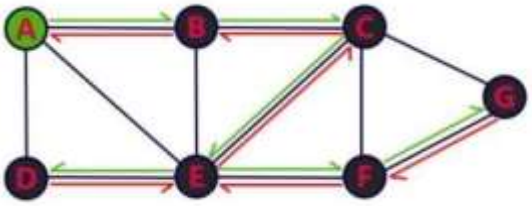
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

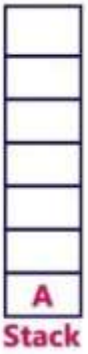




Step 13:

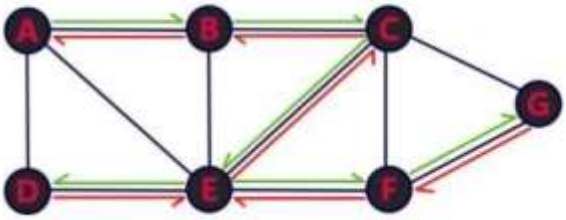
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

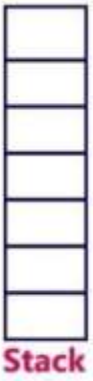




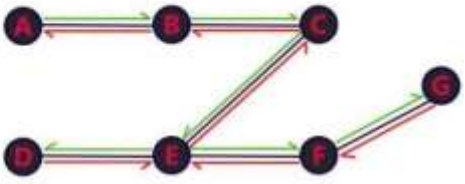
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



Pseudocode

```
DFS(G, u)
   u.visited = true
   for each v ∈ G.Adj[u]
      if v.visited == false
         DFS(G,v)

init() {
   For each u ∈ G
      u.visited = false
```

```
    For each u ∈ G
      DFS(G, u)
}
```

Complexity of Depth First Search

The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.
The space complexity of the algorithm is O(V).

Advantages:

DFSconsumes very less memory space.

It will reach at the goal node in a less time period than BFS if it traverses in a right path.

It may find a solution without examining much of search because we may get the desired solution in the very first go.

Disadvantages:

It is possible that may states keep reoccurring. There is no guarantee of finding the goal node. Sometimes the states may also enter into infinite loops.

Application of DFS Algorithm

- For finding the path
- To test if the graph is bipartite
- For finding the strongly connected components of a graph
- For detecting cycles in a graph

Breadth First Search:

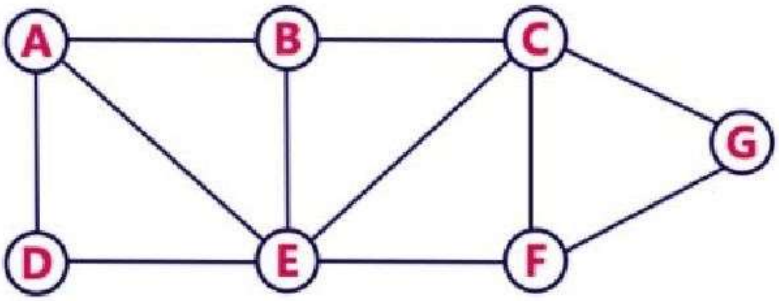
BFS (Breadth First Search)

BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

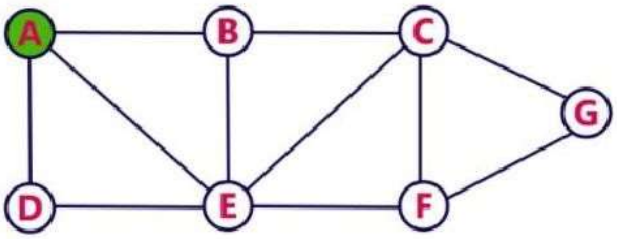
- **Step 1 -** Define a Queue of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
- **Step 3 -** Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5 -** Repeat steps 3 and 4 until queue becomes empty.
- **Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

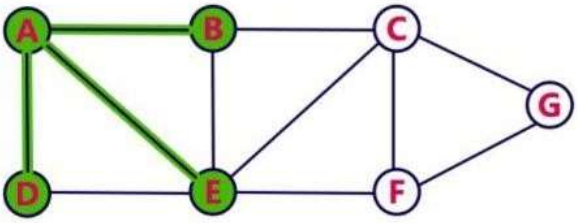


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

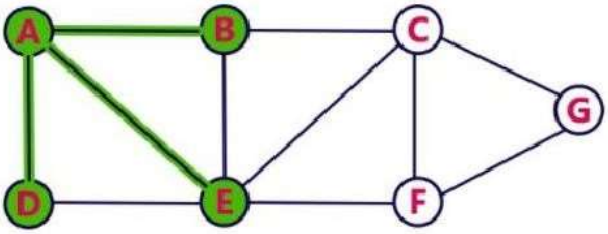


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

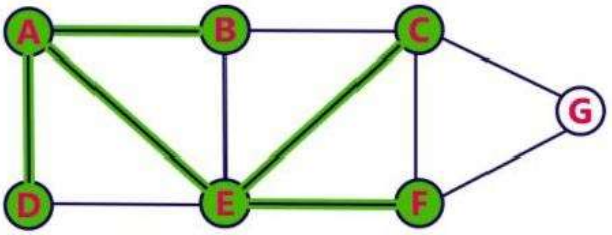


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

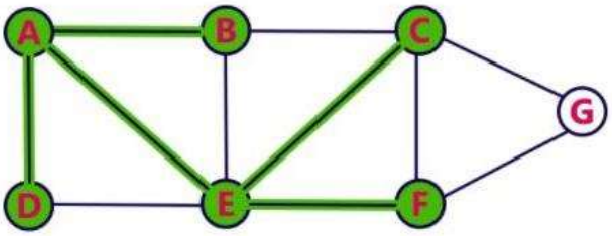


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

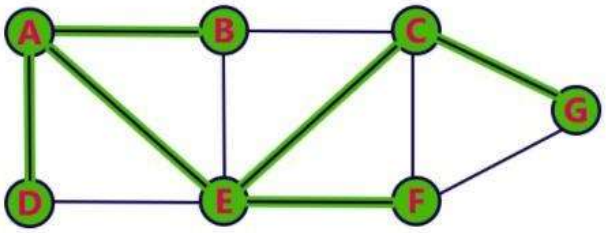


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

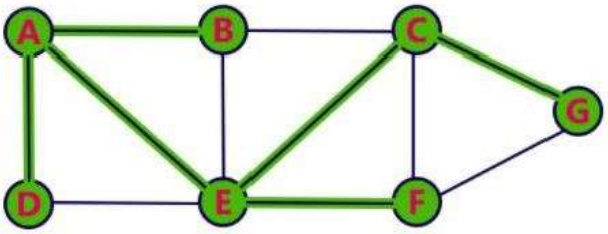


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

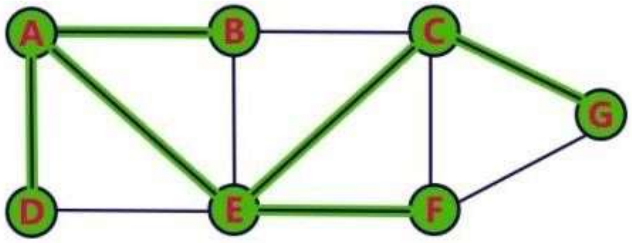


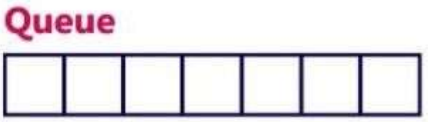
Queue

						G

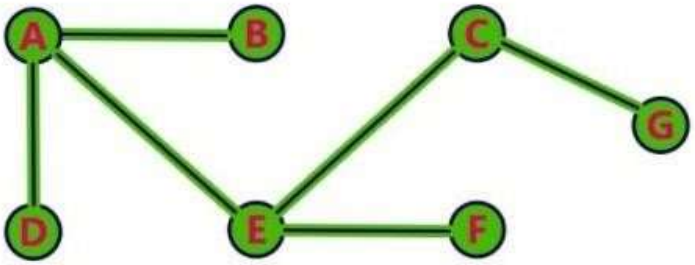
Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.





- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Pseudocode:

Input: s as the source node

```
BFS (G, s)
let Q be queue.
Q.enqueue( s )

mark s as visited
while ( Q is not empty)
v = Q.dequeue( )

for all neighbors w of v in Graph G
if w is not visited
Q.enqueue( w )
mark w as visited
```

BFS Algorithm Complexity

The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

Advantages of BFS:

1. The solution will definitely found out by BFS If there is some solution.
2. BFS will never get trapped in a blind alley, which means unwanted nodes.
3. If there is more than one solution then it will find a solution with minimal steps.

Disadvantages Of BFS:

1. Memory Constraints As it stores all the nodes of the present level to go for the next level.
2. If a solution is far away then it consumes time.

BFS Algorithm Applications

- To build index by search index
- For GPS navigation
- Path finding algorithms
- In Ford-Fulkerson algorithm to find maximum flow in a network
- Cycle detection in an undirected graph
- In minimum spanning tree

Disjoint subsets and Union-Find algorithms

What is disjoint subsets and Union Find algorithm

A *disjoint-set data structure* is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A *union-find algorithm* is an algorithm that performs two useful operations on such a data structure:

Why?

The efficiency of an algorithm sometimes depends on using an efficient data structure. A good choice of data structure can reduce the execution time of an algorithm and Union-Find is a data structure that falls in that category.

When to use?

Disjoint-set data structures model the partitioning of a set, for **example to keep track of the connected components of an undirected graph**. This model can then be used to determine whether two vertices belong to the same component, or whether adding an edge between them would result in a cycle.

Find: It determines in which subset a particular element is in and returns the representative of that particular set. An item from this set typically acts as a "representative" of the set.
Union: It merges two different subsets into a single subset, and the representative of one set becomes representative of another.
The disjoint–set also supports one other important operation called MakeSet, which creates a set containing only a given element in it.

How does Union–Find work?

We can determine whether two elements are in the same subset by comparing the result of two *Find* operations. If the two elements are in the same set, they have the same representation; otherwise, they belong to different sets. If the union is called on two elements, merge the two subsets to which the two elements belong.

How to Implement Disjoint Sets?

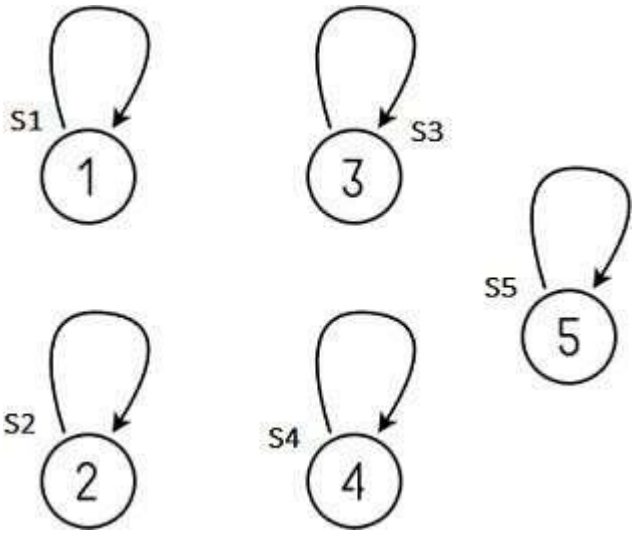
Disjoint–set forests are data structures where each set is represented by a tree data in which each node holds a reference to its parent and the representative of each set is the root of that set's tree.

- *Find* follows parent nodes until it reaches the root.
- *Union* combines two trees into one by attaching one tree's root into the root of the other.

For example, consider five disjoint sets S1 , S2 , S3 , S4 , and S5 represented by a tree, as shown below diagram. Each set initially contains only one element each, so their parent pointer points to itself or NULL.

S1 = {1}, S2 ={2}, S3 = {3}, S4 ={4} and S5 = {5}

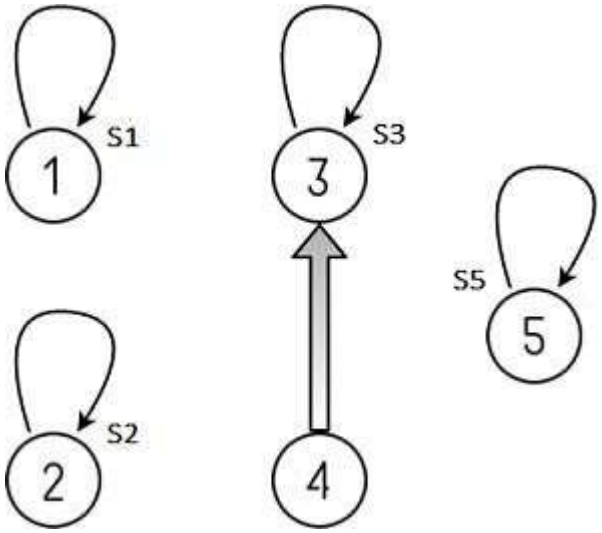
The *Find* operation on element i will return representative of S_i, where 1 <= i <= 5 , i.e., *Find* (i) = i .



If we do *Union* (S3, S4) , S3 and S4 will be merged into one disjoint set, S3 . Now,

S1 = {1}, S2 ={2}, S3 = {3, 4} and S5 = {5} .

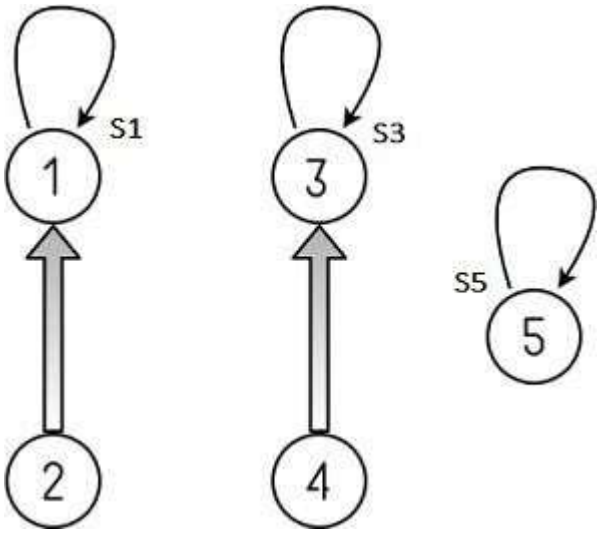
Find(4) will return representative of set S3 , i.e., *Find* (4) = 3



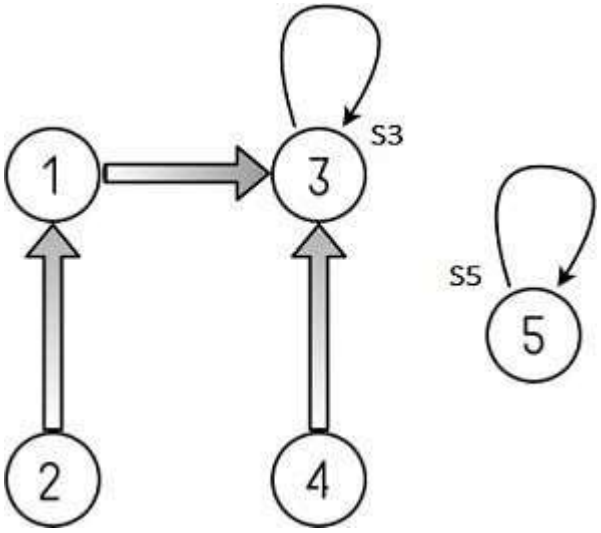
If we do *Union* (S1, S2) , S1 and S2 will be merged into one disjoint set, S1 . Now,

S1 = {1, 2}, S3 = {3, 4} and S5 = {5} .

Find(2) or *Find* (1) will return the representative of set S1 , i.e., *Find* (2) = *Find* (1) = 1



If we do *Union* (S3, S1), S3 and S1 will be merged into one disjoint set, S3. Now, S3 = {1, 2, 3, 4} and S5 = {5}.



Advantage

It provides near-constant-time operations to add new sets, to merge existing sets, and to determine whether elements are in the same set.

Disadvantage:

while connecting two elements, we do not check which subset has more element than other and sometimes it creates a big problem as in worst case we have to perform approximately linear time operations.

Applications:

Network connectivity

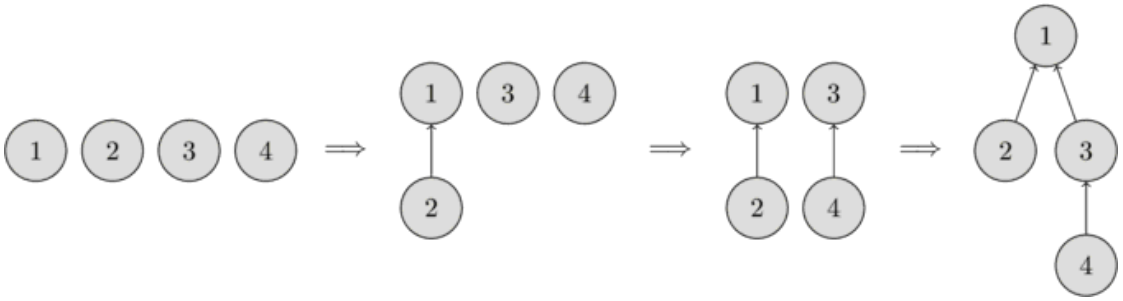
Percolation

Image Processing

Least common ancestor

Time complexity : log * N for each union find operation,where **N** is the number of elements in the set.

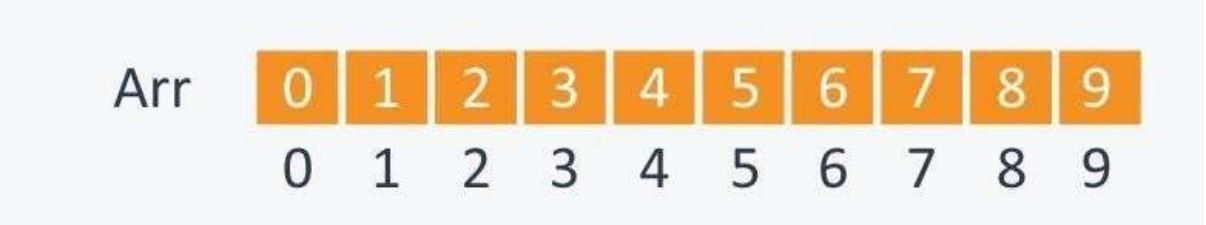
Example 1:



Example 2:

Initially there are 10 subsets and each subset has single element in it.

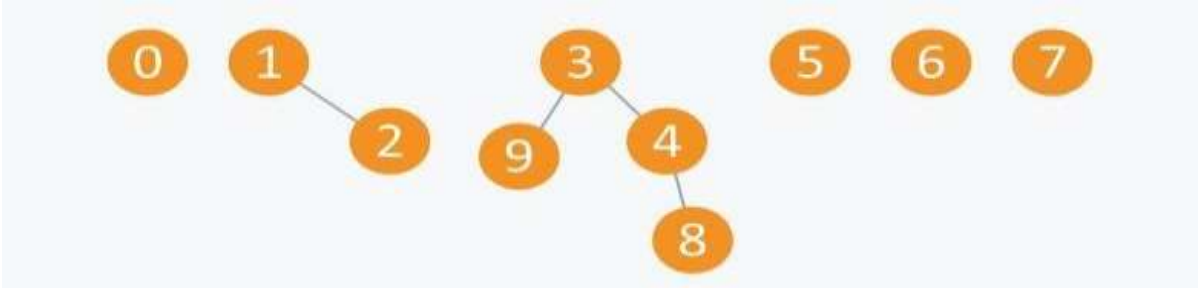
When each subset contains only single element, the array Arr is:



Let's perform some Operations: 1) Union(2, 1)

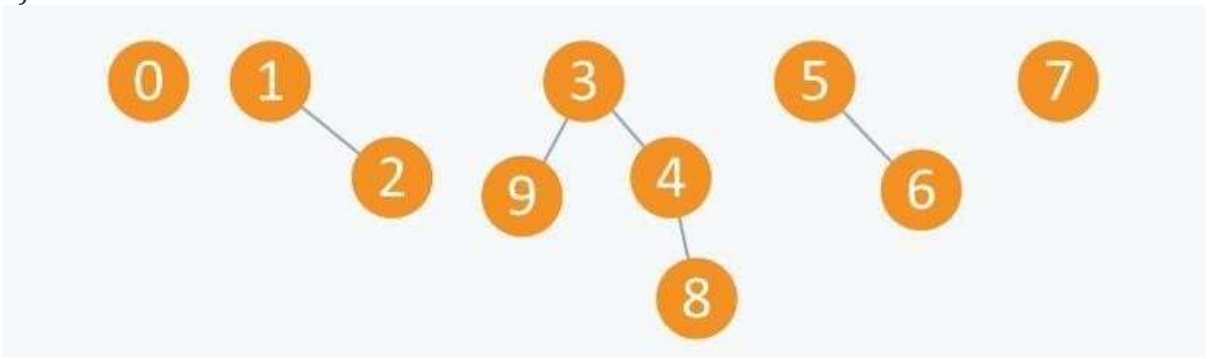
Arr will be:

2) Union(4, 3)
3) Union(8, 4)
4) Union(9, 3)



Arr will be:

5) Union(6, 5)



Arr will be:

After performing some operations of Union(A ,B), you can see that now there are 5 subsets. First has elements {3, 4, 8, 9}, second has {1, 2}, third has {5, 6}, fourth has {0} and fifth has {7}. All these subsets are said to be Connected Components.

One can also relate these elements with nodes of a graph. The elements in one subset can be considered as the nodes of the graph which are connected to each other directly or indirectly, therefore each subset can be considered as **connected component.**

From this, we can infer that Union-Find data structure is useful in Graphs for performing various operations like connecting nodes, finding connected components etc.

Let's perform some Find(A, B) operations. 1) Find (0, 7) - as 0 and 7 are disconnected ,this will gives false result.
2) Find (8, 9) -though 8 and 9 are not connected directly ,but there exist a path connecting 8 and 9, so it will give us true result.

Algorithm:

```
function MakeSet(x)
   x.parent = x

function Find(x)
   if x.parent == x
      return x
   else
      return Find(x.parent)

function Union(x, y)
   xRoot = Find(x)
   yRoot = Find(y)
   xRoot.parent = yRoot
```

Pseudocode:

```
void initialize( int Arr[ ], int N)
{
   for(int i = 0;i<N;i++)
   Arr[ i ] = i ;
}
//returns true,if A and B are connected, else it will return false.
 bool find( int Arr[ ], int A, int B)
{
if(Arr[ A ] == Arr[ B ])
return true;
else
return false;
}
//change all entries from Arr[ A ] to Arr[ B ].
void union(int Arr[ ], int N, int A, int B)
{ int TEMP = Arr[ A ];
for(int i = 0; i < N;i++)
   {
   if(Arr[ i ] == TEMP)
   Arr[ i ] = Arr[ B ];
   }
}
```

CASE STUDY

Representing Google maps

Why Use Graphs?

Graphs serve as models of a wide range of objects:

A road map

A map of airline routes

A diagram of the flow capacities in a communications or transportation network

Applications:

-Maps(Map quest, Google Maps)

-Routing Systems

Background

Google **Maps** is a web mapping service developed by Google. It offers satellite imagery, aerial photography, street maps, 360° interactive panoramic views of streets (Street View), real-time traffic conditions, and route planning for traveling by foot, car, bicycle and air (in beta), or public transportation. In 2020, Google Maps was used by over 1 billion people every month.

Google Maps first started as a C++ program designed by two Danish brothers, Lars and Jens Eilstrup Rasmussen, at the Sydney-based company Where 2 Technologies. In October 2004, the company was acquired by Google Inc. Then The Google ma wasp first launched on the Google Blog on February 8, 2005.

Google Maps is available as a mobile app for the Android and iOS mobile operating systems. The Android app was first released in September 2008.

What Is Google Map?

Google Maps is a web mapping service developed by Google, It offers satellite imagery, street maps, 360° panoramic views of streets (Street View), real-time traffic conditions (Google Traffic), and route planning for traveling by foot, car, bicycle (in beta), or public transportation.

CONCLUSION

1. It can colour the graph
2. It can add pointer to point location
3. It provide us direction with help of directed graph
4. With help of shortest path algorithm it give us appropriate location where we have to reach

UNIT 5

Sorting, Searching

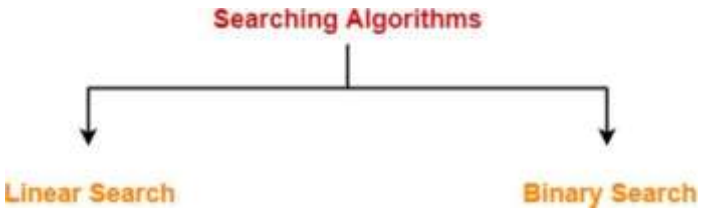
Searching Techniques: Linear Search, Binary Search. Sorting techniques: Insertion Sort, Selection sort, Merge sort, Quick Sort, Shell sort, Bubble sort, Counting sort, Bucket sort,Radix sort. Case Study: Tim Sort

Searching Techniques

WHAT IS SEARCHING?

- Searching is a process of finding a particular data item from a collection of data items based on specific criteria. Every day we perform web searches to locate data items containing in various pages.
- A search typically performed using a search key and it answers either True or False based on the item is present or not in the list.

TYPES OF SEARCHING

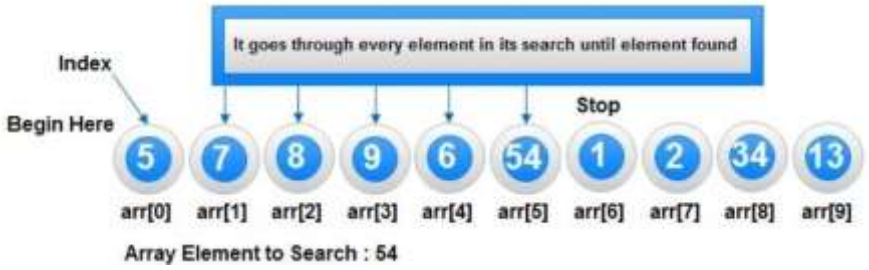


- Linear Search
- Binary Search

Linear Search

In Linear Search the list is searched sequentially and the position is returned if the key element to be searched is available in the list, otherwise -1 is returned.

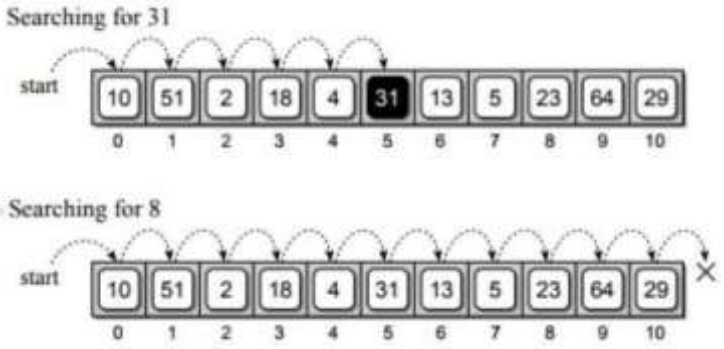
The search in Linear Search starts at the beginning of an array and move to the end, testing for a match at each item.



- Suppose there are 'n' elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on

where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison's to search an element. If search is not successful, you would need 'n' comparisons.

- The time complexity of linear search is O(n).



PSEUDOCODE:

Let array a[n] stores n elements. Determine whether element 'x' is present or not.

```
linsrch(a[n], x)
{
index = 0;
flag = 0;
while (index < n) do
{
if (x == a[index])
 {
 flag = 1;
 break;
 }
 index ++;
 }
if(flag == 1)
printf("Data found at %d position", index);
else
printf("data not found");
}
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for: 45, we'll look at 1 element before success
39, we'll look at 2 elements before success
8, we'll look at 3 elements before success
54, we'll look at 4 elements before success
77, we'll look at 5 elements before success

38 we'll look at 6 elements before success
24, we'll look at 7 elements before success
16, we'll look at 8 elements before success
4, we'll look at 9 elements before success
7, we'll look at 10 elements before success
9, we'll look at 11 elements before success
20, we'll look at 12 elements before success

For any element not in the list, we'll look at 12 elements before failure

Example 2:

Let us illustrate linear search on the following 9 elements:
Index 0 1 2 3 4 5 6 7 8
Elements -15 -6 0 7 9 23 54 82 101
Searching different elements is as follows:
1. Searching for x = 7 Search successful, data found at 3rd position.
2. Searching for x = 82 Search successful, data found at 7th position.
3. Searching for x = 42 Search un-successful, data not found.

ADVANTAGES

- Will perform fast searches of small to medium lists. With today's powerful computers, small to medium arrays can be searched relatively quickly.
- The list does not need to sorted.
- Not affected by insertions and deletions.

DISADVANTAGES

- It's time consuming for the enormous arrays.
- Inversely, slow searching of big lists. Every time a vital element matches the last element from the array or an essential element does not match any element Linear search algorithm is the worst case.

TIME COMPLEXITY:

Any algorithm is analyzed based on the unit of computation it performs. For linear search, we need to count the number of comparisons performed, but each comparison may or may not search the desired item.

Case	Best Case	Worst Case	Average Case
If item is present	1	n	n / 2
If item is not present	n	n	n

Binary Search

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.

Binary Search Algorithm can be applied only on **Sorted arrays**.

So, the elements must be arranged in-

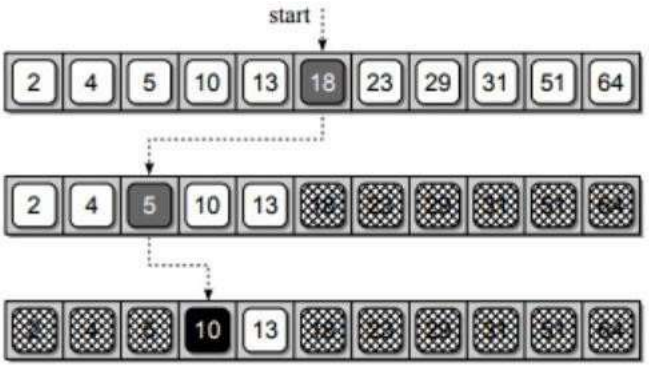
- Either ascending order if the elements are numbers.
- Or dictionary order if the elements are strings.

To apply binary search on an unsorted array,

- First, sort the array using some sorting technique.
- Then, use binary search algorithm.

Consider-

- There is a linear array 'a' of size 'n'.
- Binary search algorithm is being used to search an element 'item' in this linear array.
- If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.
- Variables beg and end keeps track of the index of the first and last element of the array or sub array in which the element is being searched at that instant.
- Variable mid keeps track of the index of the middle element of that array or sub array in which the element is being searched at that instant.



- **Searching for 10 in a sorted array using Binary Search**

Pseudocode:

```
binsrch(a[], n, x)
{
low = 1; high = n;
while (low < high) do
{
 mid = ⌊ (low + high)/2 ⌋
if (x < a[mid])
high = mid – 1;
else if (x > a[mid])
low = mid + 1;
else return mid;
}
return 0;
```

}

Example:

Let us illustrate binary search on the following 12 elements:

Index	1	2	3	4	5	6	7	8	9	10	11	12
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for x = 4: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4, found
If we are searching for x = 7: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4
low = 2, high = 2, mid = 4/2 = 2, check 7, found
If we are searching for x = 8: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8, found
If we are searching for x = 9: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9, found
If we are searching for x = 16: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9
low = 5, high = 5, mid = 10/2 = 5, check 16, found
If we are searching for x = 20: (This needs 1 comparison)
low = 1, high = 12, mid = 13/2 = 6, check 20, found
Lecture Notes 213 Dept. of Information Technology
If we are searching for x = 24: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24, found
If we are searching for x = 38: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24
low = 8, high = 8, mid = 16/2 = 8, check 38, found
If we are searching for x = 39: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39, found
If we are searching for x = 45: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 10, high = 10, mid = 20/2 = 10, check 45, found
If we are searching for x = 54: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39

low = 10, high = 12, mid = 22/2 = 11, check 54, found
If we are searching for x = 77: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 12, high = 12, mid = 24/2 = 12, check 77, found
The number of comparisons necessary by search element:
20 – requires 1 comparison;
8 and 39 – requires 2 comparisons;
4, 9, 24, 54 – requires 3 comparisons and
7, 16, 38, 45, 77 – requires 4 comparisons
Summing the comparisons, needed to find all twelve items and dividing by 12, yielding 37/12 or approximately 3.08 comparisons per successful search on the average.

Example 2:

Let us illustrate binary search on the following 9 elements:

Index	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
Elements	-15	-6	0	7	9	23	54	82	101

Solution:

The number of comparisons required for searching different elements is as follows:
1. If we are searching for x = 101: (Number of comparisons = 4)
low high mid
1 9 5
6 9 7
8 9 8
9 9 9
found

2. Searching for x = 82: (Number of comparisons = 3)
low high mid
1 9 5
6 9 7
8 9 8
found

3. Searching for x = 42: (Number of comparisons = 4)
low high mid
1 9 5
6 9 7
6 6 6
7 6 not found
4. Searching for x = -14: (Number of comparisons = 3)
low high mid
1 9 5
1 4 2
1 1 1
2 1 not found

Continuing in this manner the number of element comparisons needed to find each of

nine elements is:

Index	1	2	3	4	5	6	7	8	9
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	3	2	3	4	1	3	2	3	4

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If $x < a(1)$, $a(1) < x < a(2)$, $a(2) < x < a(3)$, $a(5) < x < a(6)$, $a(6) < x < a(7)$ or $a(7) < x < a(8)$

the algorithm requires 3 element comparisons to determine that 'x' is not present.

For all of the remaining possibilities BINSRCH requires 4 element comparisons.

Thus the average number of element comparisons for an unsuccessful search is:

$(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4$

Time Complexity of Binary Search:

In Binary Search, each comparison eliminates about half of the items from the list. Consider a list with n items, then about n/2 items will be eliminated after first comparison. After second comparison, n/4 items of the list will be eliminated. If this process is repeated for several times, then there will be just one item left in the list.

The number of comparisons required to reach to this point is $n/2^i = 1$. If we solve for i, then it gives us $i = \log n$. The maximum number is comparison is logarithmic in nature, hence the time complexity of binary search is **O(log n)**

Case	Best Case	Worst Case	Average Case
If item is present	1	O(log n)	O(log n)
If item is not present	O(log n)	O(log n)	O(log n)

ADVANTAGES:

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

DISADVANTAGES:

- It requires elements to be sorted
- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor

SORTING

Sorting in general refers to various methods of arranging or ordering things based on criteria's (numerical, chronological, alphabetical, hierarchical etc.). There are many approaches to sorting data and each has its own merits and demerits

Examples:

1) Arranging numbers in descending or ascending order.

1, 4, 5, 5, 67, 245, 456 // sorted in ascending order

2) In case of a set of characters, ordering elements in alphabetic order.

a, c, f, k, m, x, z //sorted in alphabetic order

One of the most common tasks that everyone performs in their daily life is searching and sorting.

Likewise, the computer also performs these tasks several times for different operations. A real-life example would be a dictionary, where the words are stored in alphabetical order, so searching becomes easier.

This means the smallest data element can be searched from a huge data repository if the data is stored in a sorted manner.

Types of Sorting Algorithms:

In data processing, there are various sorting methods and techniques that are not only used for sorting algorithms but are also used for analyzing the performance of other algorithms.

- Internal Sorting
- External Sorting

Internal Sorting

This method uses only the primary memory during sorting process. All data items are held in main memory and no secondary memory is required this sorting process. If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting.

There is a limitation for internal sorts; they can only process relatively small lists due to memory constraints. There are 3 types of internal sorts.

(i) SELECTION SORT :- Ex:- Selection sort algorithm, Heap Sort algorithm
(ii) INSERTION SORT :- Ex:- Insertion sort algorithm, Shell Sort algorithm
(iii) EXCHANGE SORT :- Ex:- Bubble Sort Algorithm, Quick sort algorithm

External Sorting

Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which is not fit into the main memory. So, primary memory holds the currently being sorted data only.
All external sorts are based on process of merging. Different parts of data are sorted separately and merged together. Ex:- Merge Sort

INSERTION SORT

Insertion sort is the **sorting mechanism where the sorted array is built having one item at a time**. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order.

CONCEPT

In an insertion sort, the first element in the array is considered as sorted, even if it is an unsorted array. In an insertion sort, each element in the array is checked with the previous elements, resulting in a growing sorted output list. With each iteration, the sorting algorithm removes one element at a time and finds the appropriate location within the sorted array and inserts it there. The iteration continues until the whole list is sorted.

Why

Insertion sort **is less efficient than the other sorting algorithms** like heap sort, quick sort, merge sort, etc. Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

When to use Insertion sort

Insertion sort is **used when number of elements is small**. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

Advantage

- The pure simplicity of the algorithm.
- The relative order of items with equal keys does not change.
- The ability to sort a list as it is being received.
- Efficient for small data sets, especially in practice than other quadratic algorithms — i.e. $O(n^2)$.

Disadvantage

- Insertion sort is inefficient against more extensive data sets.
- The insertion sort exhibits the worst-case time complexity of O(n2)
- It does not perform well than other, more advanced sorting algorithms.

Time Complexity

Worst Case : $O(n^2)$
Best Case : $\Omega(n)$
Average Case : $\Theta(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.
- **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 insertion sort is **$O(n^2)$**.
- **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 insertion sort is **$O(n^2)$**.

Space Complexity

Space Complexity	O(1)
Stable	YES

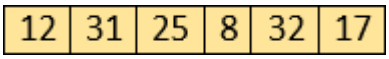
The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

Applications

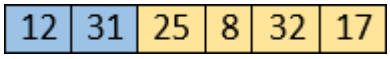
- Commercial computing.
- In discrete mathematics.
- In priority scheduling.
- In shortest-job-first scheduling.

Example

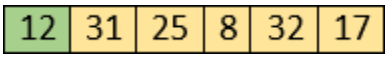
Let the elements of array are -



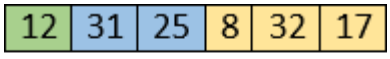
Initially, the first two elements are compared in insertion sort.

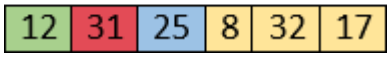


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



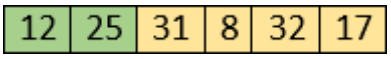
Now, move to the next two elements and compare them.



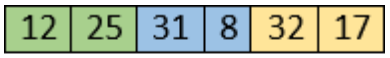


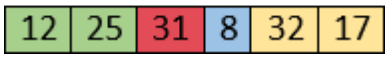
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.



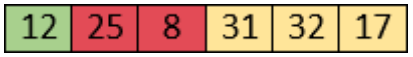
Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.



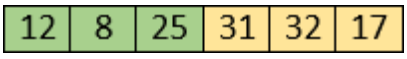


Both 31 and 8 are not sorted. So, swap them.

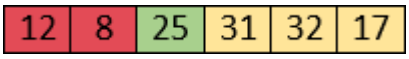
After swapping, elements 25 and 8 are unsorted.



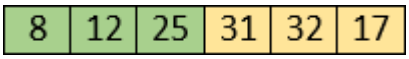
So, swap them.



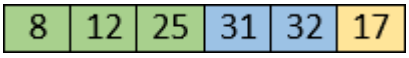
Now, elements 12 and 8 are unsorted.



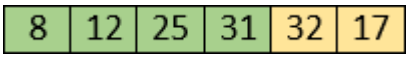
So, swap them too.



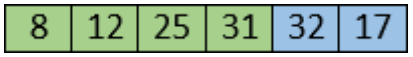
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



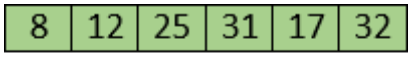
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.



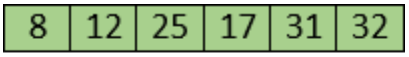
Move to the next elements that are 32 and 17.



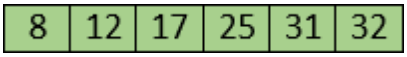
17 is smaller than 32. So, swap them.



Swapping makes 31 and 17 unsorted. So, swap them too.



Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

Algorithm

Step 1 – If it is the first element, it is already sorted. return 1;

Step 2 – Pick next element

Step 3 – Compare with all elements in the sorted sub-list

Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 – Insert the value

Step 6 – Repeat until list is sorted

Pseudocode

```
INSERTION-SORT(A)
  for i = 1 to n
          key ← A [i]
```

```
        j ← i – 1
        while j > = 0 and A[j] > key
                A[j+1] ← A[j]
                j ← j – 1
        End while
        A[j+1] ← key
End for
```

SELECTION SORT

Selection sort is **a simple sorting algorithm**. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

CONCEPT

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

Why

Selection sort can **be good at checking if everything is already sorted**. It is also good to use when memory space is limited. This is because unlike other sorting algorithms, selection sort doesn't go around swapping things until the very end, resulting in less temporary storage space used.

When to use Selection Sort algorithm

The selection sort is best used when you want to:

- You have to sort a small list of items in ascending order
- When the cost of swapping values is insignificant
- It is also used when you need to make sure that all the values in the list have been checked.

Advantage

The following are the advantages of the selection sort

- It performs very well on small lists
- It is an in-place algorithm. It does not require a lot of space for sorting. Only one extra space is required for holding the temporal variable.
- It performs well on items that have already been sorted.

Disadvantage

- It performs poorly when working on huge lists.
- The number of iterations made during the sorting is n-squared, where n is the total number of elements in the list.

- Other algorithms, such as quicksort, have better performance compared to the selection sort.

Complexity

Worst Case : $O(n^2)$
Best Case : $\Omega(n^2)$
Average Case : $\Theta(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is **$O(n^2)$**.
- **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 selection sort is **$O(n^2)$**.
- **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 selection sort is **$O(n^2)$**.

Space Complexity	O(1)
Stable	YES

Applications

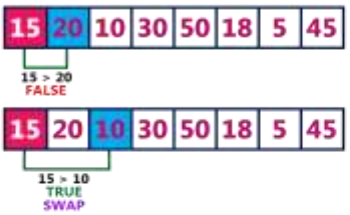
- Commercial computing.
- Search for information.
- Operations research.
- Event-driven simulation.
- Numerical computations.
- Combinatorial search

Example

Consider the following unsorted list of elements...

Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.



List after 1st iteration

Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 4th iteration

Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 5th iteration

Iteration #6

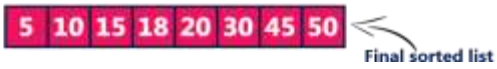
Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 6th iteration

Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration



Algorithm

The selection sort algorithm is performed using the following steps...

Step 1 - Select the first element of the list (i.e., Element at first position in the list).

Step 2: Compare the selected element with all the other elements in the list.

Step 3: In every comparision, if any element is found smaller than the selected element (for Ascending order), then both are swapped.

Step 4: Repeat the same procedure with element in the next position in the list till the entire list is sorted.

Pseudocode

```
for(i=0; i<size; i++){
    for(j=i+1; j<size; j++){
        if(list[i] > list[j])
            {
          temp=list[i];
          list[i]=list[j];
          list[j]=temp;
        }
    }
 }
```

SHELL SORT

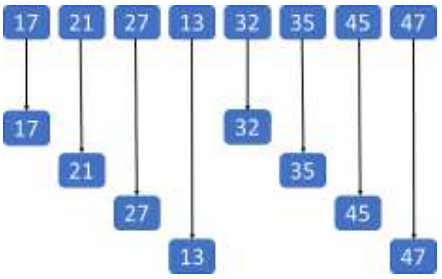
In insertion sort, we could move the elements ahead only by one position at a time. If we wish to move an element to a faraway position in insertion sort, a lot of movements were involved thus increasing the execution time. Shell sort overcomes this problem of insertion sort by allowing movement and swap of far-away elements as well.

This sorting technique works by sorting elements in pairs, far away from each other and subsequently reduces their gap. The gap is known as the **interval.** We can calculate this gap/interval with the help of Knuth's formula.

CONCEPT

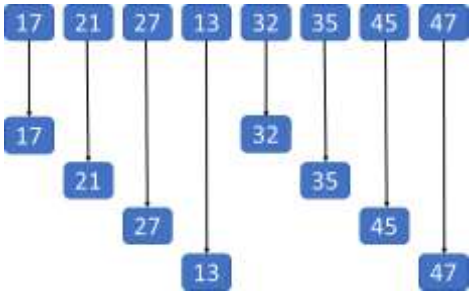
- Consider the following example to have a better understanding of how shell sort works. You must use the same array as in the previous examples. For the purpose of clarity, you must use the interval of 4 as an example.

Make a virtual sub-list of all values in the four-position interval. These are the values: (32, 17); (35, 21); (45, 27); and (13, 47).



- You compare the values in each sub-list and swap them in the original array if necessary. After this step, the new array should look like this:

- Then we pick one interval and divide it into two sub-lists: (17, 27, 32, 45) and (21, 13, 35, 47).



- You compare the values in the original array and, if necessary, swap them. This is how the array should look after this step:

- Finally, you need to use interval 1 to sort the rest of the array. The array is sorted by shell sort using insertion sort.

The following is a step-by-step guide:

Why

Shellsort is **an optimization of insertion sort that allows the exchange of items that are far apart**. The idea is to arrange the list of elements so that, starting anywhere, taking every hth element produces a sorted list. Such a list is said to be h-sorted.

When to use Shell sort

Shell sort is a special case of insertion sort. It was designed to overcome the drawbacks of insertion sort. Thus, it is more efficient than insertion sort. In shell sort, we **can swap or exchange the far away items in addition to adjacent items**.

Advantage

- Shell sort algorithm is only efficient for finite number of elements in an array.
- Shell sort algorithm is 5.32 x faster than bubble sort algorithm.

Disadvantage

- Shell sort algorithm is complex in structure and bit more difficult to understand.
- Shell sort algorithm is significantly slower than the merge sort, quick sort and heap sort algorithms.

Complexity

Best Case	O(n*logn)
Average Case	O(n*log(n)2)
Worst Case	O(n2)

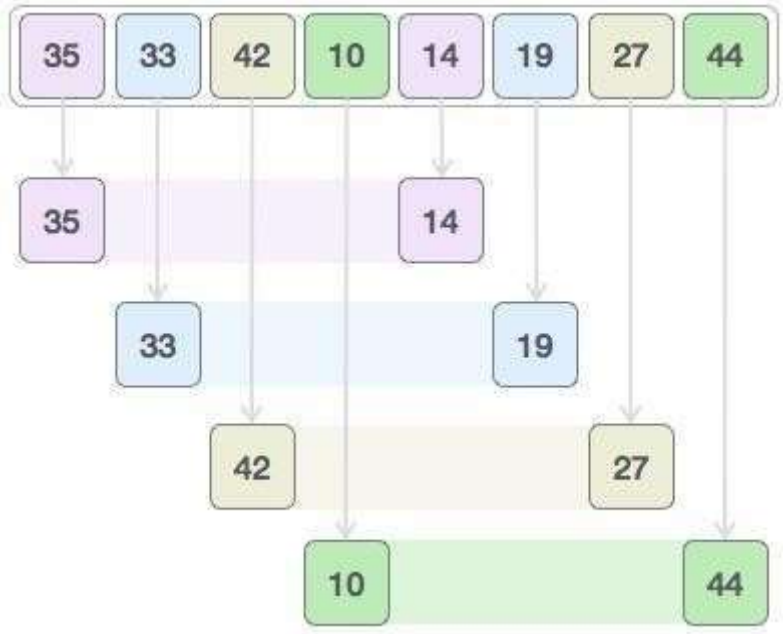
Applications

Shell Sort can be used for a variety of purposes, including:

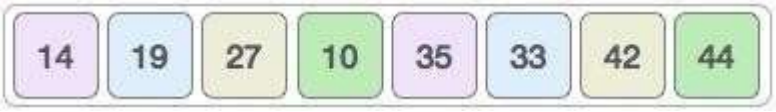
- Shell sort has a more significant cache miss percentage than quicksort and executes more operations.
- However, some versions of the qsort function in the C standard library geared at embedded devices utilize it instead of quicksort because it requires less code and does not use the call stack. The uClibc library, for example, uses Shell sort. The Linux kernel has a Shell Sort implementation for similar reasons.
- Shell sort can also be used as a sub-algorithm of the introspective sort to sort short subarrays and avoid slowdowns when the recursion depth exceeds a certain threshold. This method is used in the bzip2 compression algorithm, for example.

Example

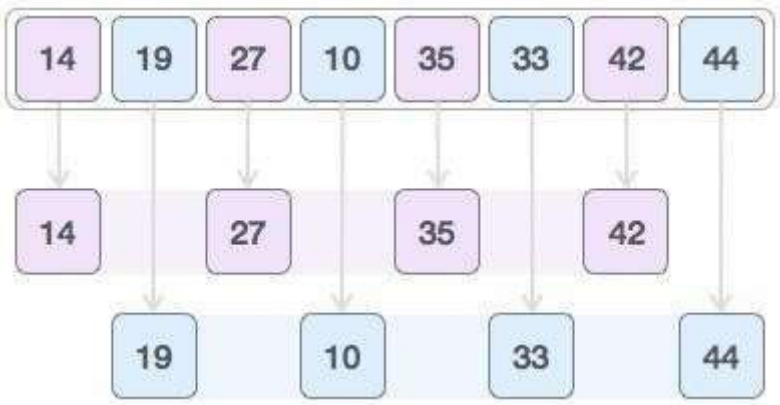
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



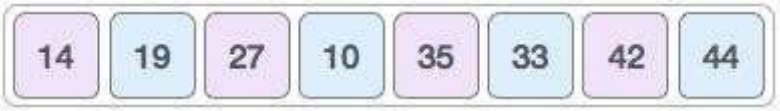
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this –



Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this –



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction –

Algorithm

Step 1: Initialize the gap size i.e. h
Step 2: Divide the array into sub-arrays each having interval of h
Step 3: Sort the sub-arrays with insertion sort
Step 4: Reduce the value of h
Step 5: Repeat the above steps until the array is sorted

Pseudocode

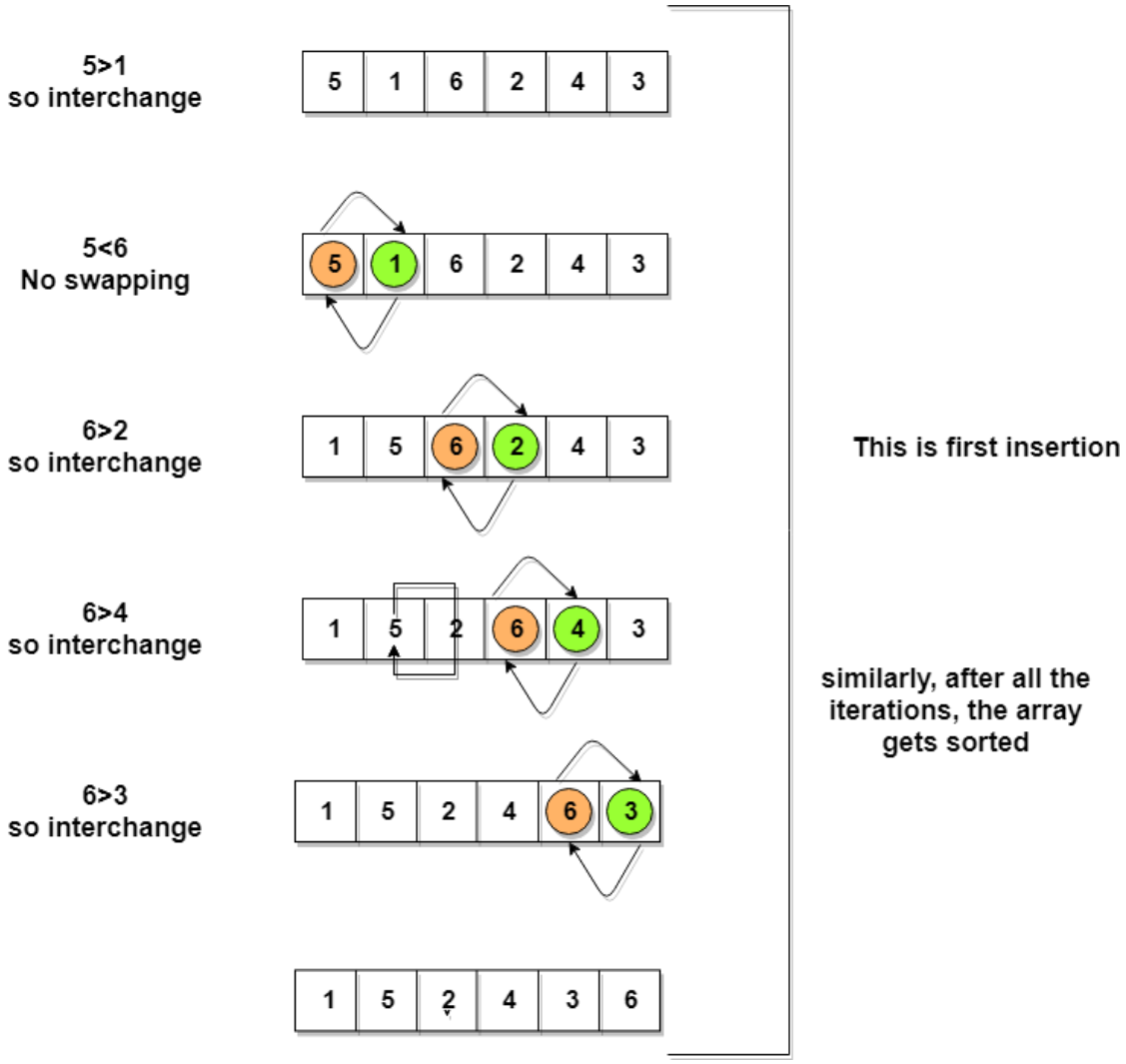
```
procedure Shell_sort(Array, n) //n = size of array
   while gap < length(Array) /3 :
      gap = interval * 3 + 1
   END while loop

   while gap > 0 :
      for (outer = gap; outer < length(Array); outer++):
         Insertion_value = Array[outer]
         inner = outer;
         while inner > gap-1 && Array[inner - gap] >= Insertion_value:
            Array[inner] = Array[inner - gap]
            inner = inner - gap
         END while loop
         Array[inner] = Insertion_value
      END for loop

      gap = (gap -1) /3;

   end while loop
END Shell_sort
```

Bubble sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

CONCEPT

Following are the steps involved in bubble sort(for sorting a given array in ascending order):
Starting with the first element(index = 0), compare the current element with the next element of the array.
If the current element is greater than the next element of the array, swap them.
If the current element is less than the next element, move to the next element. Repeat Step 1.
Let's consider an array with values {5, 1, 6, 2, 4, 3}
Below, we have a pictorial representation of how bubble sort will sort the given array.



So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.
Similarly after the second iteration, 5 will be at the second last index, and so on.

Why

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller elements "bubble" to the top of the list.
When to use Bubble sort
Bubble sort is mainly used in educational purposes for helping students understand the foundations of sorting. This is used to identify whether the list is already sorted.

Advantage

The built-in ability to detect whether the list is sorted efficiently is the only advantage of bubble sort over other sorting techniques.
When the list is already sorted (which is the best-case scenario), the complexity of bubble sort is only O(n).
It is faster than other in case of sorted array and consumes less time to describe whether the input array is sorted or not.

Disadvantage
The worst situation for bubble sort is when the list's smallest element is in the last position. In this situation, the smallest element will move down one place on each pass through the list, meaning that the sort will need to make the maximum number of passes through the list, namely n - 1.

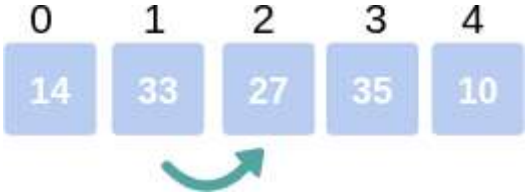
Complexity
Worst Case Time Complexity [Big-O]: O(n2)
Best Case Time Complexity [Big-omega]: O(n)
Average Time Complexity [Big-theta]: O(n2)
Space Complexity: O(1)

Applications
Bubble sort is used if
complexity does not matter
short and simple code is preferred

Example
Consider the following array: Arr=14, 33, 27, 35, 10. We need to sort this array using bubble sort algorithm.

First Pass:
We proceed with the first and second element i.e., Arr[0] and Arr[1]. Check if 14 > 33 which is false. So, no swapping happens and the array remains the same.

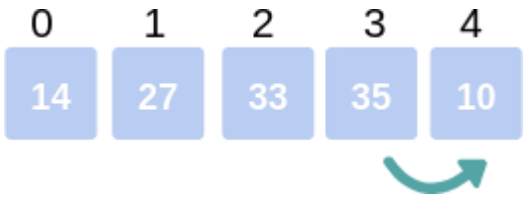
We proceed with the second and third element i.e., Arr[1] and Arr[2]. Check if 33 > 27 which is true. So, we swap Arr[1] and Arr[2].



Thus the array becomes:

We proceed with the third and fourth element i.e., Arr[2] and Arr[3]. Check if 33 > 35 which is false. So, no swapping happens and the array remains the same.

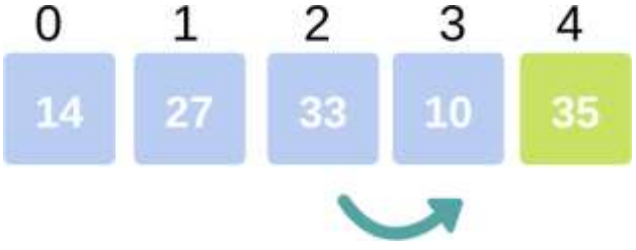
We proceed with the fourth and fifth element i.e., Arr[3] and Arr[4]. Check if 35 > 10 which is true. So, we swap Arr[3] and Arr[4].



Thus, after swapping the array becomes:

Thus, marks the end of the first pass, where the Largest element reaches its final(last) position.
Second Pass:
We proceed with the first and second element i.e., Arr[0] and Arr[1]. Check if 14 > 27 which is false. So, no swapping happens and the array remains the same.

We now proceed with the second and third element i.e., Arr[1] and Arr[2]. Check if 27 > 33 which is false. So, no swapping happens and the array remains the same.
We now proceed with the third and fourth element i.e., Arr[2] and Arr[3]. Check if 33 > 10 which is true. So, we swap Arr[2] and Arr[3].

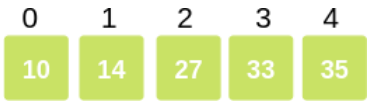


Now, the array becomes

Thus marks the end of second pass where the second largest element in the array has occupied its correct position.

Third Pass:
After the third pass, the third largest element will be at the third last position in the array.

i- th Pass:
After the ith pass, the ith largest element will be at the ith last position in the array.
n-th Pass:
After the nth pass, the nth largest element(smallest element) will be at nth last position(1st position) in the array, where 'n' is the size of the array.
After doing all the passes, we can easily see the array will be sorted.
Thus, the sorted array will look like this:



Algorithm

```
begin BubbleSort(list)

  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for

  return list

end BubbleSort
```

Pseudocode

```
procedure bubbleSort( list : array of items )

  loop = list.count;

  for i = 0 to loop-1 do:
    swapped = false

    for j = 0 to loop-1 do:

      /* compare the adjacent elements */
      if list[j] > list[j+1] then
        /* swap them */
        swap( list[j], list[j+1] )
        swapped = true
      end if

    end for

    /*if no number was swapped that means
    array is sorted now, break the loop.*/
```

```
    if(not swapped) then
      break
    end if

  end for

end procedure return list
```

QUICK SORT

Quick sort is a divide and conquer algorithm. Quick sort first divides a large list into two smaller sub- lists: the low elements and the high elements. Quick sort can then recursively sort the sub-lists.

The steps are:

1. Pick an element, called a **pivot**, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

The base case of the recursion is lists of size zero or one, which never need to be sorted.

Quick sort, or **partition-exchange sort**, is a sorting algorithm developed by Tony Hoare that, on average, makes O(n log n) comparisons to sort n items. In the worst case, it makes O(n^2) comparisons, though this behavior is rare. Quick sort is often faster in practice than other O(n log n) algorithms. It works by first of all by partitioning the array around a pivot value and then dealing with the 2 smaller partitions separately. Partitioning is the most complex part of quick sort. The simplest thing is to use the first value in the array, a[l] (or a[0] as l = 0 to begin with) as the pivot. After the partitioning, all values to the left of the pivot are <= pivot and all values to the right are > pivot. The same procedure for the two remaining sub lists is repeated and so on recursively until we have the entire listsorted.

Advantages:

- One of the fastest algorithms on average.
- Does not need additional memory (the sorting takes place in the array - this is called **in-place**
  processing).

Disadvantages: The worst-case complexity is O(N^2)

Algorithm

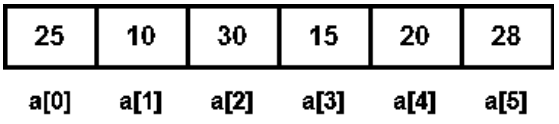
- **Step 1** – Make any element as pivot
- **Step 2** – Partition the array on the basis of pivot
- **Step 3** – Apply quick sort on left partition recursively
- **Step 4** – Apply quick sort on right partition recursively

Pseudocode

```
void quickSort(int list[10],int first,int last){
   int pivot,i,j,temp;
    if(first < last){
       pivot = first;
       i = first;
       j     =     last;
       while(i < j){
          while(list[i] <= list[pivot] && i < last)
             i++;
          while(list[j] && list[pivot])
             j--;
          if(i < j){
              temp = list[i];
              list[i] = list[j];
              list[j] = temp;
          }
       }
temp = list[pivot];
       list[pivot] = list[j];
       list[j] = temp;
       quickSort(list,first,j-1);
       quickSort(list,j+1,last);
   }
}
```

Example:

Consider the following array has to be sorted in ascending order using quick sort algorithm-



Quick Sort Example

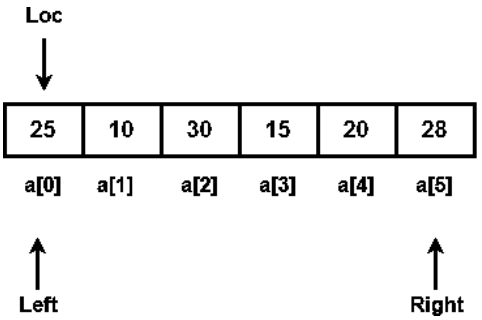
Quick Sort Algorithm works in the following steps-

Step-01:

Initially-

- **Left** and **Loc** (pivot) points to the first element of the array.
- **Right** points to the last element of the array.

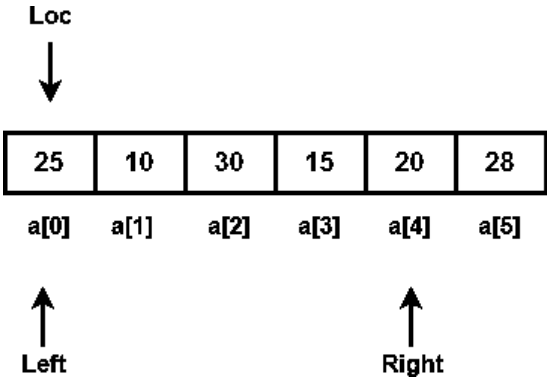
So to begin with, we set **loc** = 0, **left** = 0 and **right** = 5 as-



Step-02:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] < a[right], so algorithm moves **right** one position towards left as-

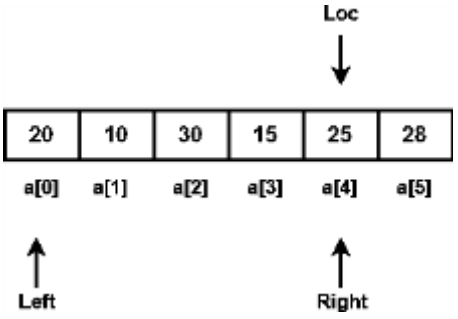


Now, **loc** = 0, **left** = 0 and **right** = 4.

Step-03:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] > a[right], so algorithm swaps a[loc] and a[right] and **loc** points at **right** as-

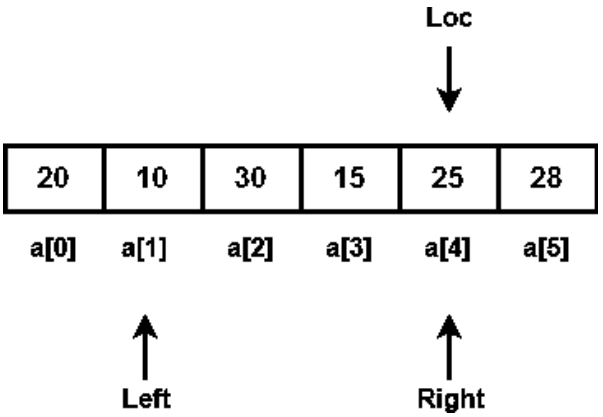


Now, **loc** = 4, **left** = 0 and **right** = 4.

Step-04:

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

As a[loc] > a[left], so algorithm moves **left** one position towards right as-

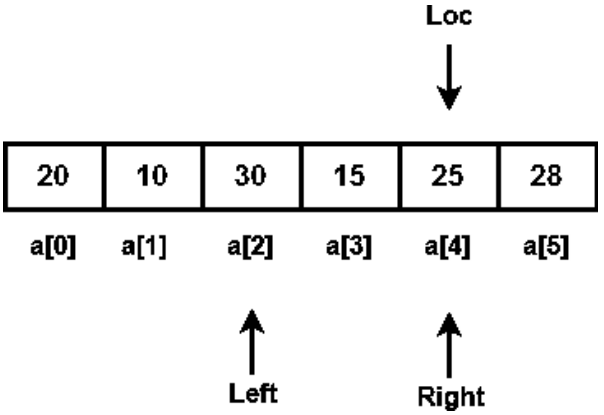


Now, **loc** = 4, **left** = 1 and **right** = 4.

Step-05:

Since **loc** points at right, so algorithm starts from **left** and move towards right.

As a[loc] > a[left], so algorithm moves **left** one position towards right as-

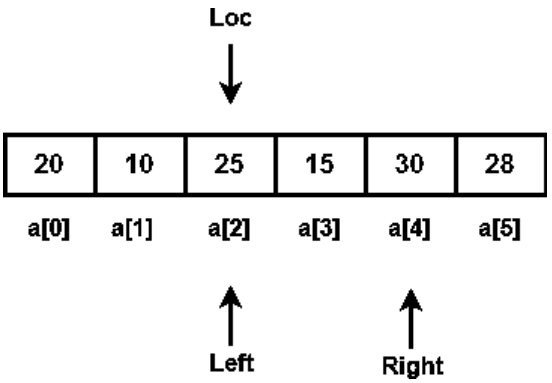


Now, **loc** = 4, **left** = 2 and **right** = 4.

Step-06:

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

As a[loc] < a[left], so we algorithm swaps a[loc] and a[left] and **loc** points at **left** as-

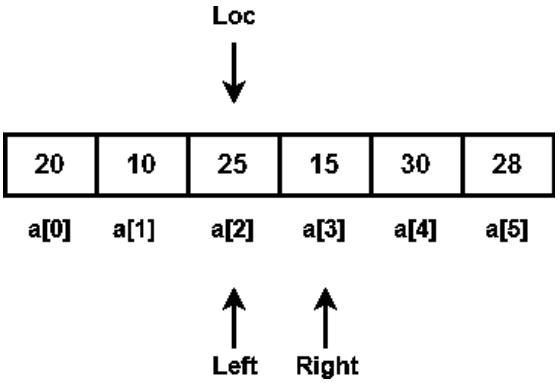


Now, **loc** = 2, **left** = 2 and **right** = 4.

Step-07:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] < a[right], so algorithm moves **right** one position towards left as-

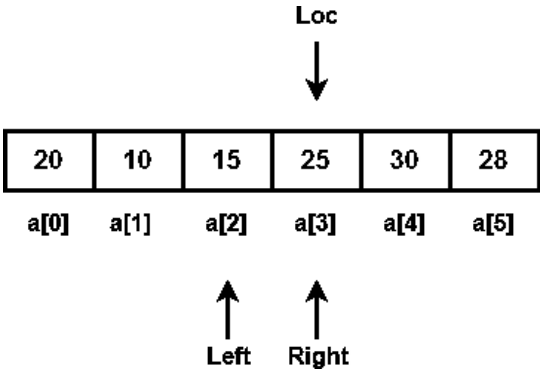


Now, **loc** = 2, **left** = 2 and **right** = 3.

Step-08:

Since **loc** points at **left**, so algorithm starts from **right** and move towards left.

As a[loc] > a[right], so algorithm swaps a[loc] and a[right] and **loc** points at **right** as-

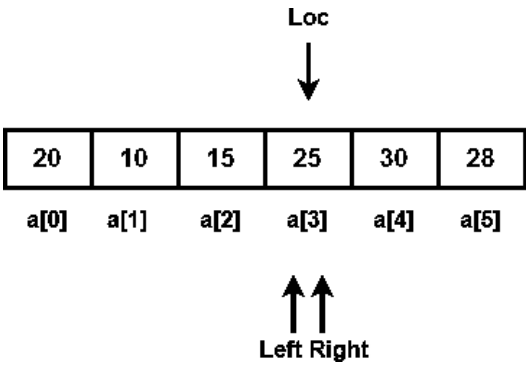


Now, **loc** = 3, **left** = 2 and **right** = 3.

Step-09:

Since **loc** points at **right**, so algorithm starts from **left** and move towards right.

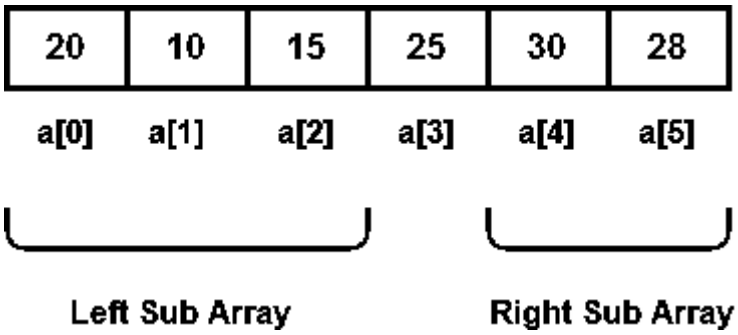
As a[loc] > a[left], so algorithm moves **left** one position towards right as-



Now, **loc** = 3, **left** = 3 and **right** = 3.

Now,

- **loc**, **left** and **right** points at the same element.
- This indicates the termination of procedure.
- The pivot element 25 is placed in its final position.
- All elements to the right side of element 25 are greater than it.
- All elements to the left side of element 25 are smaller than it.



Now, quick sort algorithm is applied on the left and right sub arrays separately in the similar manner.

Advantage

The quick sort is regarded as the best sorting algorithm. This is because of its significant advantage in terms of efficiency because it is able to deal well with a huge list of items. Because it sorts in place, no additional storage is required as well.

Disadvantage

The slight disadvantage of quick sort is that its worst-case performance is similar to average performances of the bubble, insertion or selections sorts. In general, the quick sort produces the most effective and widely used method of sorting a list of any item size.

Time Complexity Of Quick Sort
Worst-case performance O(n2)
Best-case performance
O(n log n) (simple partition) or O(n) (three-way partition and equal keys)
Average performance O(n log n)
Worst-case space complexity
O(n) auxiliary (naive)
O(log n) auxiliary

MERGE SORT

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time

complexity being O(n log n), it is one of the most respected algorithms.
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Algorithm:

Step 1 − If it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order

Pseudocode

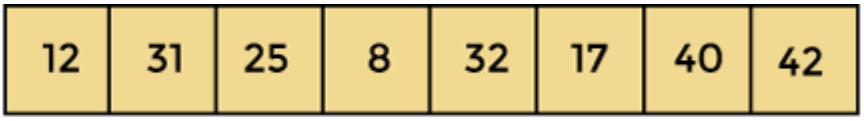
```
MERGE_SORT(arr, beg, end)
if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if
END MERGE_SORT
```

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.

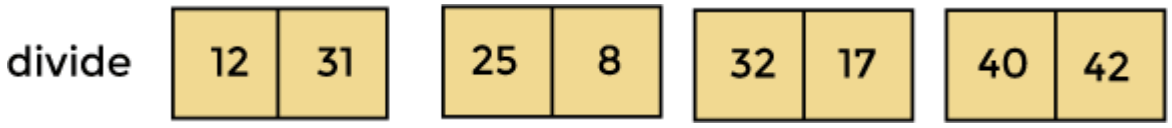
Let the elements of array are -



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.

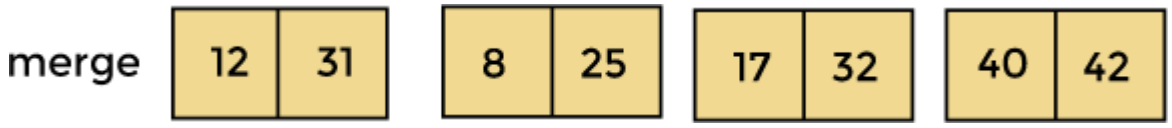


Now, again divide these arrays to get the atomic value that cannot be further divided.

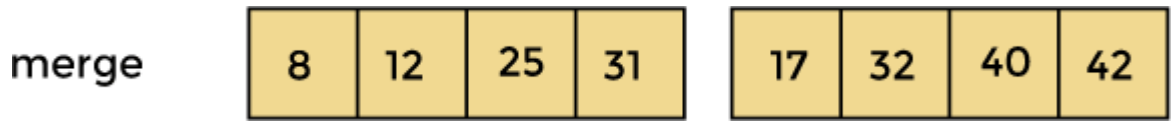
Now, combine them in the same manner they were broken.

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

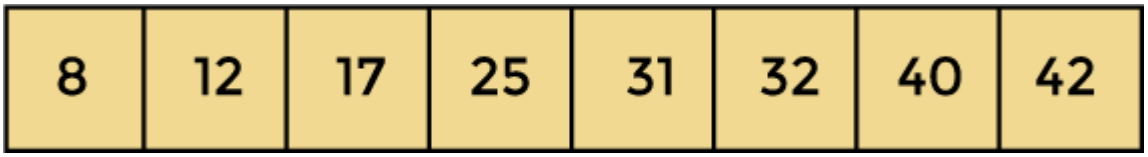
So, first compare 12 and 31, both are in sorted positions. Then compare 25 and 8, and in the list of two values, put 8 first followed by 25. Then compare 32 and 17, sort them and put 17 first followed by 32. After that, compare 40 and 42, and place them sequentially.



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.

Time Complexity

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

RADIX SORT

Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order.
In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers.
Sorting is performed from least significant digit to the most significant digit.
Radix sort algorithm requires the number of passes which are equal to the number of digits present in the largest number among the list of numbers.
For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.

Algorithm:
The Radix sort algorithm is performed using the following steps...
Step 1 - Define 10 queues each representing a bucket for each digit from 0 to 9.
Step 2 - Consider the least significant digit of each number in the list which is to be sorted.
Step 3 - Insert each number into their respective queue based on the least significant digit.
Step 4 - Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
Step 5 - Repeat from step 3 based on the next least significant digit.
Step 6 - Repeat from step 2 until all the numbers are grouped based on the most significant digit.

Pseudocode

radixSort(arr)
max = largest element in the given array
d = number of digits in the largest element (or, max)
Now, create d buckets of size 0 - 9
for i -> 0 to d
sort the array elements using counting sort (or any stable sort) according to the digits at the ith place

Example:

The steps used in the sorting of radix sort are listed as follows -

- First, we have to find the largest element (suppose **max**) from the given array. Suppose **'x'** be the number of digits in **max**. The **'x'** is calculated because we need to go through the significant places of all elements.
- After that, go through one by one each significant place. Here, we have to use any stable sorting algorithm to sort the digits of each significant place.

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

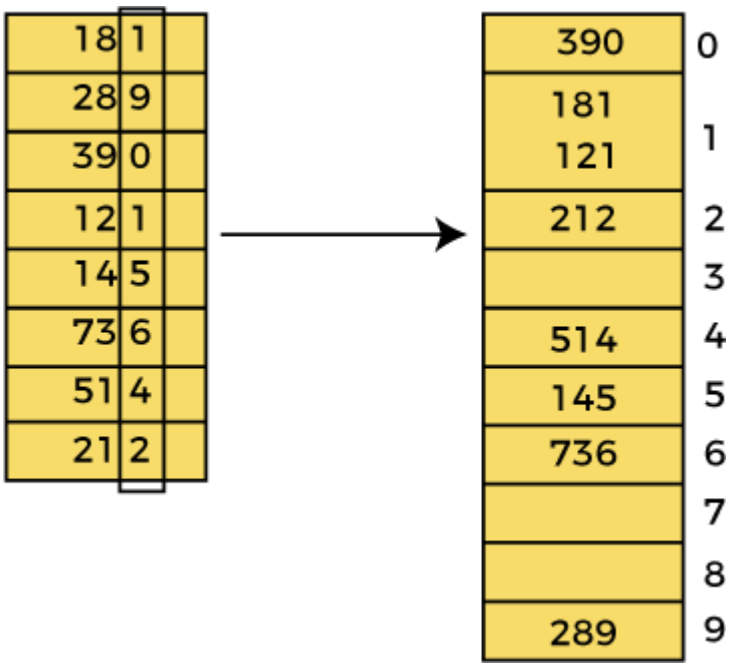
181	289	390	121	145	736	514	212

In the given array, the largest element is **736** that have **3** digits in it. So, the loop will run up to three times (i.e., to the **hundreds place**). That means three passes are required to sort the array.

Now, first sort the elements on the basis of unit place digits (i.e., **x = 0**). Here, we are using the counting sort algorithm to sort the elements.

Pass 1:

In the first pass, the list is sorted on the basis of the digits at 0's place.

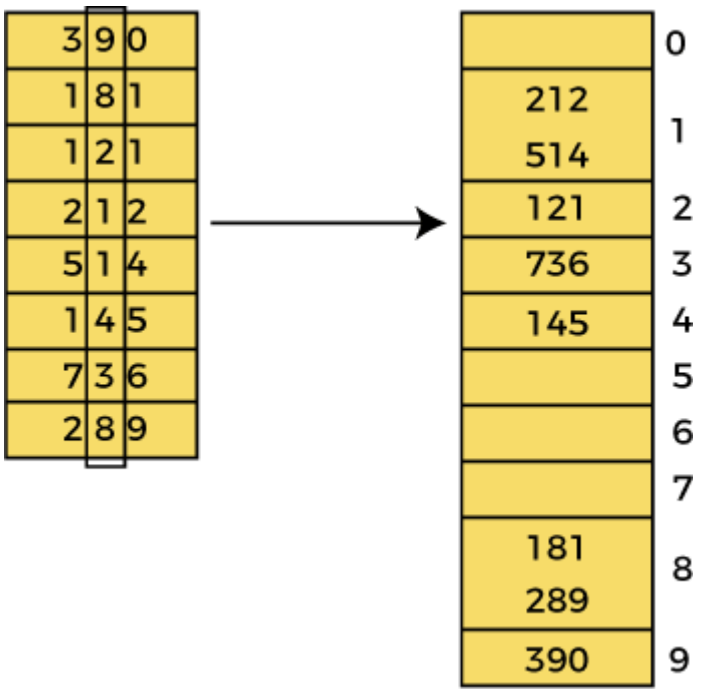


After the first pass, the array elements are -

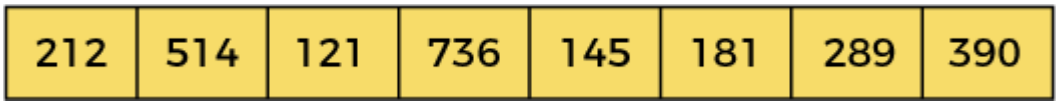
390	181	121	212	514	145	736	289

Pass 2:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10^{th} place).

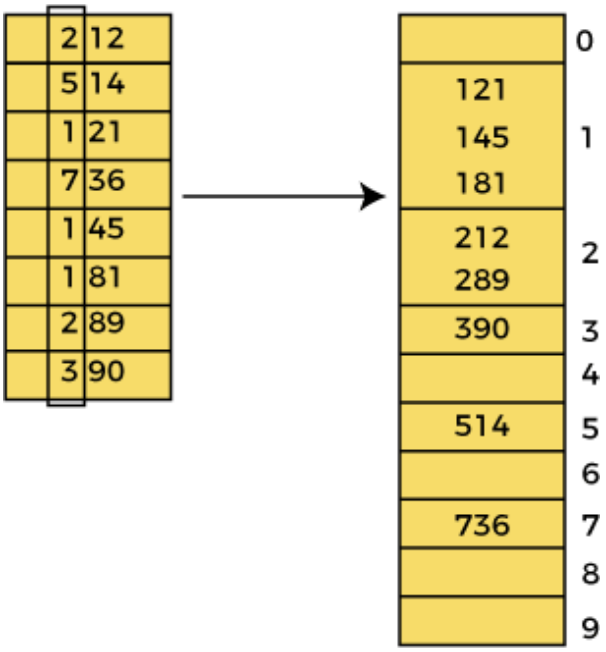


After the second pass, the array elements are -

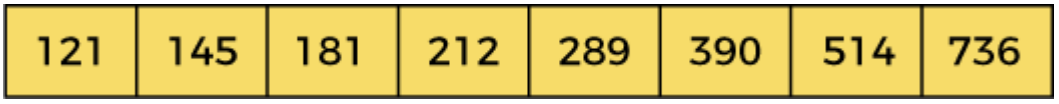


Pass 3:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 100th place).



After the third pass, the array elements are -



Now, the array is sorted in ascending order.

Time Complexity

Best Case	Ω(n+k)
Average Case	θ(nk)
Worst Case	O(nk)

BUCKET SORT

In the Bucket Sorting technique, the data items are distributed in a set of buckets. Each bucket can hold a similar type of data. After distributing, each bucket is sorted using another sorting algorithm. After that, all elements are gathered on the main list to get the sorted form.

The basic procedure of performing the bucket sort is given as follows -

- First, partition the range into a fixed number of buckets.
- Then, toss every element into its appropriate bucket.
- After that, sort each bucket individually by applying a sorting algorithm.
- And at last, concatenate all the sorted buckets.

Example:

Now, create buckets with a range from 0 to 25. The buckets range are 0-5, 5-10, 10-15, 15-20, 20-25. Elements are inserted in the buckets according to the bucket range. Suppose the value of an item is 16, so it will be inserted in the bucket with the range 15-20. Similarly, every item of the array will insert accordingly.

This phase is known to be the scattering of array elements.

Now, sort each bucket individually. The elements of each bucket can be sorted by using any of the stable sorting algorithms.

At last, gather the sorted elements from each bucket in order

Now, the array is completely sorted.

ALGORITHM:

bucketSort(a[], n)

1. Create 'n' empty buckets
2. Do **for** each array element a[i]
   2.1. Put array elements into buckets, i.e. insert a[i] into bucket[n*a[i]]
3. Sort the elements of individual buckets by using the insertion sort.
4. At last, gather or concatenate the sorted buckets.

End bucketSort

PSEUDOCODE

1. Bucket Sort(A[])
2. 1. Let B[0 ...n-1] be a **new** array
3. 2. n=length[A]
4. 3. **for** i=0 to n-1
5. 4. make B[i] an empty list
6. 5. **for** i=1 to n
7. 6. **do** insert A[i] into list B[n a[i]]
8. 7. **for** i=0 to n-1
9. 8. **do** sort list B[i] with insertion-sort
10. 9. Concatenate lists B[0], B[1],....., B[n-1] together in order
11. End

TIME COMPLEXITY

Best Case- O(n + k)

Average Case- O(n + k)

Worst Case- O(n^2)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. In Bucket sort, best case occurs when the elements are uniformly distributed in the buckets. The complexity will be better if the elements are already sorted in the buckets. If we use the insertion sort to sort the bucket elements, the overall complexity will be linear, i.e., O(n + k), where O(n) is for making the buckets, and O(k) is for sorting the bucket elements using algorithms with linear time complexity at best case.
  The best-case time complexity of bucket sort is **O(n + k)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. Bucket sort runs in the linear time, even when the elements are uniformly distributed. The average case time complexity of bucket sort is **O(n + K)**.
- **Worst Case Complexity -** In bucket sort, worst case occurs when the elements are of the close range in the array, because of that, they have to be placed in the same bucket. So, some buckets have more number of elements than others. The complexity will get worse when the elements are in the reverse order. The worst-case time complexity of bucket sort is **O(n^2)**.

SPACE COMPLEXITY

O(n*k)

ADVANTAGES

- The advantage of bucket sort is that once the elements are distributed into buckets, each bucket can be processed independently of the others.
- This means that you often need to sort much smaller arrays as a follow-up step than the original array.
- It also means that you can sort all the buckets in parallel with one another.

DISADVANTAGES

- The disadvantage is that if you get a bad distribution into the buckets, you may end up doing a tremendous amount of extra work for no benefit or a minimal benefit.

- As a result, bucket sort works best when the data are more or less uniformly distributed or where there is an intelligent way to choose the buckets given a quick set of heuristics based on the input array.

COUNTING SORT

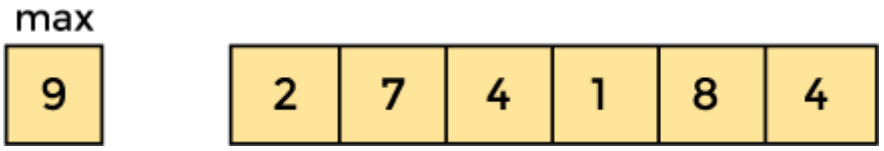
Counting sort is a stable sorting technique, which is used to sort objects according to the keys that are small numbers. It counts the number of keys whose key values are same. This sorting technique is effective when the difference between different keys are not so big, otherwise, it can increase the space complexity.

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

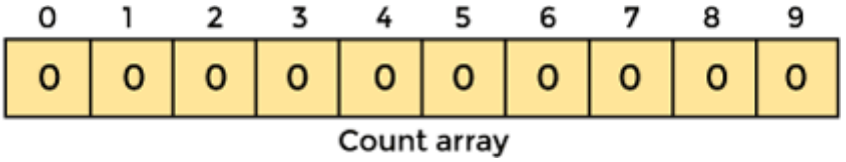
To understand the working of the counting sort algorithm, let's take an unsorted array. It will be easier to understand the counting sort via an example.

Let the elements of array are -

1. Find the maximum element from the given array. Let **max** be the maximum element.

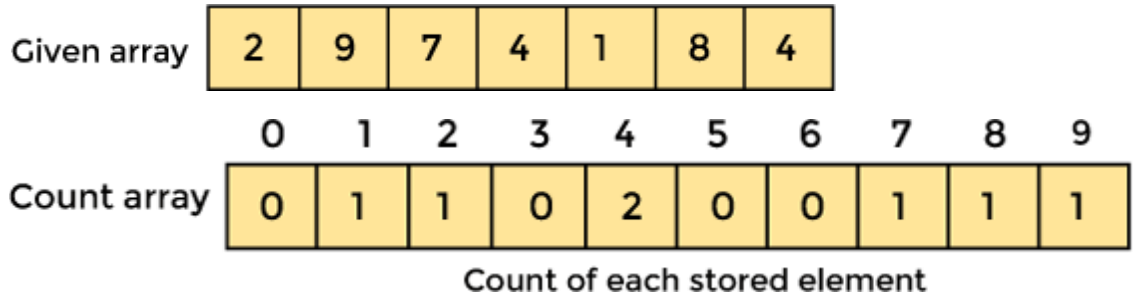


2. Now, initialize array of length **max + 1** having all 0 elements. This array will be used to store the count of the elements in the given array.

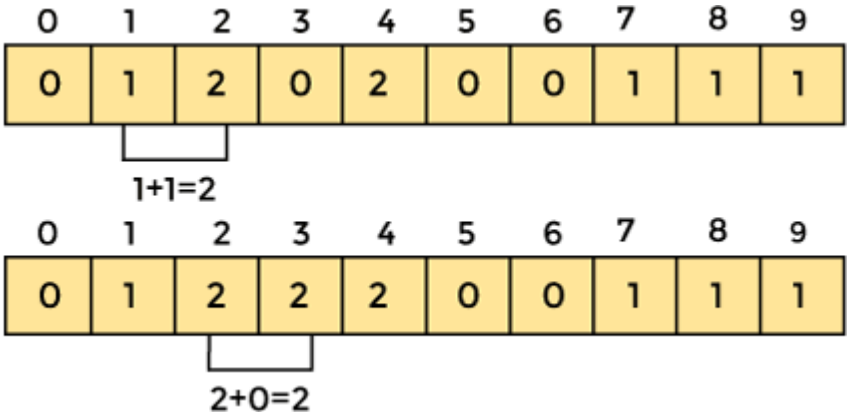


3. Now, we have to store the count of each array element at their corresponding index in the count array.

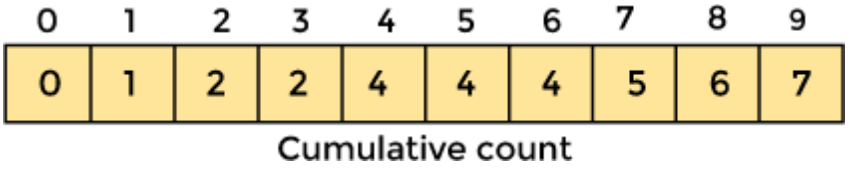
The count of an element will be stored as - Suppose array element '4' is appeared two times, so the count of element 4 is 2. Hence, 2 is stored at the 4th position of the count array. If any element is not present in the array, place 0, i.e. suppose element '3' is not present in the array, so, 0 will be stored at 3rd position.



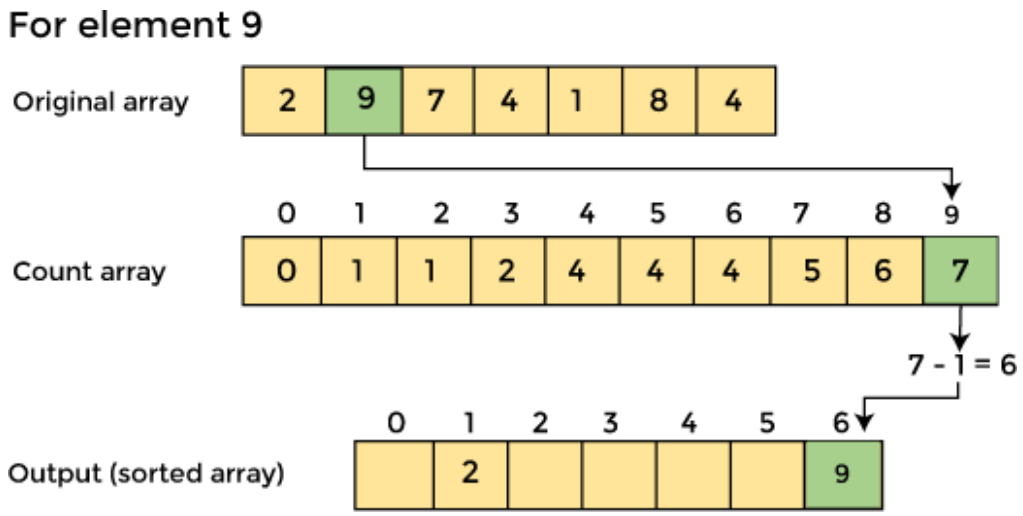
Now, store the cumulative sum of **count** array elements. It will help to place the elements at the correct index of the sorted array.



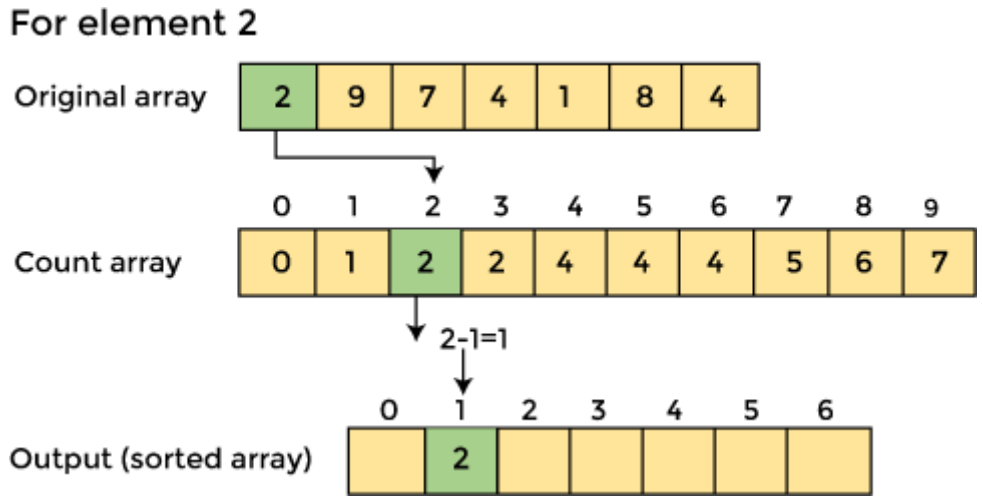
Similarly, the cumulative count of the count array is -



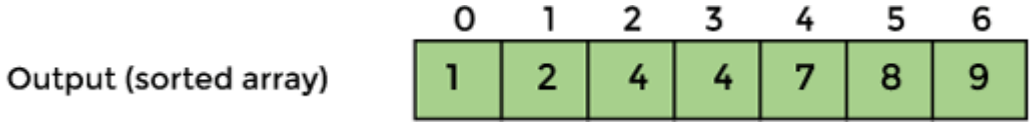
4. Now, find the index of each element of the original array



After placing element at its place, decrease its count by one. Before placing element 2, its count was 2, but after placing it at its correct position, the new count for element 2 is 1.



Similarly, after sorting, the array elements are -



Now, the array is completely sorted.

Counting Sort Algorithm

countingSort(array, size)

max <- find largest element in array

initialize count array with all zeros

for j <- 0 to size

find the total count of each unique element and

store the count at jth index in count array

for i <- 1 to max

find the cumulative sum and store it in count array itself

for j <- size down to 1

restore the elements to array

decrease count of each element restored by 1

PSEUDOCODE

```
CountingSort(A)
 //A[]-- Initial Array to Sort
 //Complexity: O(k)
 for i = 0 to k do
 c[i] = 0

 //Storing Count of each element
 //Complexity: O(n)
 for j = 0 to n do
 c[A[j]] = c[A[j]] + 1

 // Change C[i] such that it contains actual
 //position of these elements in output array
 ////Complexity: O(k)
 for i = 1 to k do
 c[i] = c[i] + c[i-1]

 //Build Output array from C[i]
 //Complexity: O(n)
 for j = n-1 downto 0 do
 B[ c[A[j]]-1 ] = A[j]
 c[A[j]] = c[A[j]] - 1
end func
```

TIME AND SPACE COMPLEXITY

In this algorithm, the initialization of the count array and the loop which performs a prefix sum on the count array takes O(k) time. And other two loops for initialization of the output array takes O(n) time. Therefore, the total time complexity for the algorithm is : O(k)+ O(n)+ O(k)+ O(n)= O(n+k).

Worst Case Time complexity: O (n+k)
Average Case Time complexity: O(n+k)
Best Case Time complexity: O(n+k)
Space Complexity: O(k)
Data Structure: Array
Sorting In Place: No
Stable: Yes

ADVANTAGES

The biggest advantage of counting sort is its complexity – O(n+k), where n is the size of the sorted array and k is the size of the helper array (range of distinct values).

DISADVANTAGES

- if non-primitive (object) elements are sorted, another helper array is needed to store the sorted elements.
- counting sort can be used only to sort discrete values (for example integers), because otherwise the array of frequencies cannot be constructed.

CASE STUDY:
TIM SORT

A very fast, O(n log n), is a hybrid stable sorting algorithm. It was implemented by Tim Peters in 2002 for use in the Python programming language and now used in java Arrays.sort() as well. Timsort first analyses the list it is trying to sort and then chooses an approach based on the analysis of the list. In the background, it's basically working by making use of two famous sorting algorithm namely merge sort and insertion sort, in a very optimistic way.

im sort uses Binary insertion sort and improved merge sort by using galloping in a combination. Binary insertion sort is the best method to sort when data is already or partially sorted or the length of run is smaller than MIN_RUN and merge sort is best when the input is large.

Tim Sort is complex, even by algorithmic standards. The operation is best broken down into parts:

Run (dividing slot)

An input array is divided into different sub-arrays, count of elements inside a sub-array is defined as a RUN, the minimum value of such runs is a MIN_RUN which is a power of 2 not more than 32 (or 64).

These sub-arrays are usually partially ordered (strictly ascending or stricly descending).When the array is decomposed into several runs, the runs of ascending order remain unchanged, and the runs of strictly descending order are reversed. Finally, several runs of ascending order are obtained.

Example:

Consider an array of goals:

1	5	9	8	6	4	5	6	7

We can see that [1, 5, 9] conforms to ascending order, [8, 6, 4] conforms to strict descending order, [4, 5, 6, 7] conforms to ascending order.

Flip the strictly descending run and then add together a new array:

1	5	9	6	8	4	5	6	7

Merge run

Let's assume that the new array is:

....	6	7	8	9	10	1	2	3	4	....

If we merge two runs using Mergesort alone
run1: [6, 7, 8, 9, 10]
run2: [1, 2, 3, 4, 5]

Mergesort compares the first element of the two runs
1 < 6, get 1
2 < 6, get 2
3 < 6, get 3
4 < 6, get 4
5 < 6, get 5

Traversing the entire run directly would eventually consume more number of merge operations if the run was longer. Because every run is in ascending order, there's no need to compare them one by one and the concept of Galloping becomes handy.

Galloping

For example, to merge the following two runs:
run1: [101, 102, 103, ... 200]
run2: [1, 2, 3, ..., 100]
Instead of comparing the elements one by one, we compare them by increasing powers of 2^n where n >= 0.

run1[0] > run2[0]
run1[0] > run2[1]
run1[0] > run2[3]
run1[0] > run2[7]
run1[0] > run2[15]
...
run1[0] > run2[2^n - 1]
run1[0] <= run2[2^(n+1) - 1]

We get a result run2 [2 ^ n - 1] < run1[0] <= run2 [2 ^ (n + 1) - 1], Since run2 is sorted, we use Binary Search to locate run1[0] in run2 very efficiently by defining left = run2[2^n - 1], right = run[2^(n+1) - 1]. Hence. the number of times we merge two runs is reduced from O(N) to O(logN).

Why not skip Galloping and just do Binary Search?

Let's give an example:
run1: [1, 3, 5, 7, 9 ... 2n+1]
run2: [0, 2, 4, 6, 8 ... 2n]

In this way, run1[0] is only larger than run2[0]. Each time we do Binary Search, we can only get a number of results, plus n times of Binary Search, so the time complexity changes from O(N) to O(NlogN).

Stack determines the order

When the original array becomes a set of various ascending runs, we need to merge two runs, but if we merge a long run with a shorter run, it will take a longer time to compare a comparitively shorter run. So Timsort maintains a stack with all the run lengths on the stack, while satisfying that the length of run on the back stack is longer than the sum of the length of run on the first two stacks, so that the length of run always decreases.

Stack : [... runA, runB, runC]
runA > runB + runC
runB > runC
This avoids run merges with too much difference in length.

For those who prefer bullets:

- Establish a minrun size that is a power of 2 (usually 32, never more than 64 or your Binary Insertion Sort will lose efficiency)
- Find a run in the first minrun of data.
- If the run is not at least minrun in length, use Insertion Sort to grab subsequent or prior items and insert them into the run until it is the correct minimum size.
- Repeat until the entire array is divided into sorted subsections.
- Use the latter half of Merge Sort to join the ordered arrays.

Complexity Analysis

By design TimSort is well suited for partially sorted data with the best case being totally sorted data. It falls into the adaptive sort family. Taking the number of runs ρ as a (natural) parameter for a refined analysis we obtained:

TimSort runs in O(N + Nlogρ) time.

- Worst case time complexity: **Θ(NlogN)**
- Average case time complexity: **Θ(NlogN)**
- Best case time complexity: **Θ(N)**
- Space complexity: **Θ(N)**