DATA STRUCTURE AND ALGORITHM

COURSE CODE:- PCCST303

NAME OF FACULTY:- AYSHA
AP/CSE

SEMESTER S3

DATA STRUCTURES AND ALGORITHMS

(Common to CS/CA/CM/CD/CR/AI/AM/AD/CB/CN/CC/CU/CI/CG)

Course Code	**PCCST303**	**CIE Marks**	40
Teaching Hours/Week (L: T:P: R)	3:1:0:0	**ESE Marks**	60
Credits	4	**Exam Hours**	2 Hrs. 30 Min.
Prerequisites (if any)	UCEST105	**Course Type**	Theory

Course Objectives:

1. To provide the learner a comprehensive understanding of data structures and algorithms.
2. To prepare them for advanced studies or professional work in computer science and related fields.

SYLLABUS

Module No.	Syllabus Description	Contact Hours
1	**Basic Concepts of Data Structures** Definitions; Data Abstraction; Performance Analysis - Time & Space Complexity, Asymptotic Notations; Polynomial representation using Arrays, Sparse matrix (*Tuple representation*); Stacks and Queues - Stacks, Multi-Stacks, Queues, Circular Queues, Double Ended Queues; Evaluation of Expressions- Infix to Postfix, Evaluating Postfix Expressions.	**11**
2	**Linked List and Memory Management** Singly Linked List - Operations on Linked List, Stacks and Queues using Linked List, Polynomial representation using Linked List; Doubly Linked List; Circular Linked List; Memory allocation - First-fit, Best-fit, and Worst-fit allocation schemes; Garbage collection and compaction.	**11**
3	**Trees and Graphs** Trees :- Representation Of Trees; Binary Trees - Types and Properties, Binary Tree Representation, Tree Operations, Tree Traversals; Expression Trees; Binary Search Trees - Binary Search Tree Operations; Binary Heaps - Binary Heap Operations, Priority Queue. Graphs :- Definitions; Representation of Graphs; Depth First Search and	**11**

	Breadth First Search; Applications of Graphs - Single Source All Destination.	
4	**Sorting and Searching** Sorting Techniques :- Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Radix Sort. Searching Techniques :- Linear Search, Binary Search, Hashing - Hashing functions : Mid square, Division, Folding, Digit Analysis; Collision Resolution : Linear probing, Quadratic Probing, Double hashing, Open hashing.	**11**

Course Assessment Method
(CIE: 40 marks, ESE: 60 marks)

Continuous Internal Evaluation Marks (CIE):

Attendance	Assignment/ Microproject	Internal Examination-1 (Written)	Internal Examination- 2 (Written)	Total
5	**15**	**10**	**10**	**40**

End Semester Examination Marks (ESE)

In Part A, all questions need to be answered and in Part B, each student can choose any one full question out of two questions

Part A	Part B	Total
• 2 Questions from each module. • Total of 8 Questions, each carrying 3 marks **(8x3 =24 marks)**	• Each question carries 9 marks. • Two questions will be given from each module, out of which 1 question should be answered. • Each question can have a maximum of 3 sub divisions. **(4x9 = 36 marks)**	**60**

Course Outcomes (COs)

At the end of the course students should be able to:

Course Outcome		Bloom's Knowledge Level (KL)
CO1	Identify appropriate data structures for solving real world problems.	**K3**
CO2	Describe and implement linear data structures such as arrays, linked lists, stacks, and queues.	**K3**
CO3	Describe and Implement non linear data structures such as trees and graphs.	**K3**
CO4	Select appropriate searching and sorting algorithms to be used in specific circumstances.	**K3**

Note*: K1- Remember, K2- Understand, K3- Apply, K4- Analyse, K5- Evaluate, K6- Create*

CO-PO Mapping Table (Mapping of Course Outcomes to Program Outcomes)

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	3	3									3
CO2	3	3	3									3
CO3	3	3	3									3
CO4	3	3	3									3

Note: *1: Slight (Low), 2: Moderate (Medium), 3: Substantial (High), -: No Correlation*

Text Books				
Sl. No	**Title of the Book**	**Name of the Author/s**	**Name of the Publisher**	**Edition and Year**
1	Fundamentals of Data Structures in C	Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed,	Universities press,	2/e, 2007
2	Introduction to Algorithms	Thomas H Cormen, Charles Leisesrson, Ronald L Rivest, Clifford Stein	PHI	3/e, 2009

Reference Books				
Sl. No	**Title of the Book**	**Name of the Author/s**	**Name of the Publisher**	**Edition and Year**
1	Classic Data Structures	Samanta D.	Prentice Hall India.	2/e, 2018
2	Data Structures and Algorithms	Aho A. V., J. E. Hopcroft and J. D. Ullman	Pearson Publication.	1/e, 2003
3	Introduction to Data Structures with Applications	Tremblay J. P. and P. G. Sorenson	Tata McGraw Hill.	2/e, 2017
4	Theory and Problems of Data Structures	Lipschuts S.	Schaum's Series	2/e, 2014

Video Links (NPTEL, SWAYAM…)	
Module No.	**Link ID**
1	https://nptel.ac.in/courses/106102064
2	https://ocw.mit.edu/courses/6-851-advanced-data-structures-spring-2012/

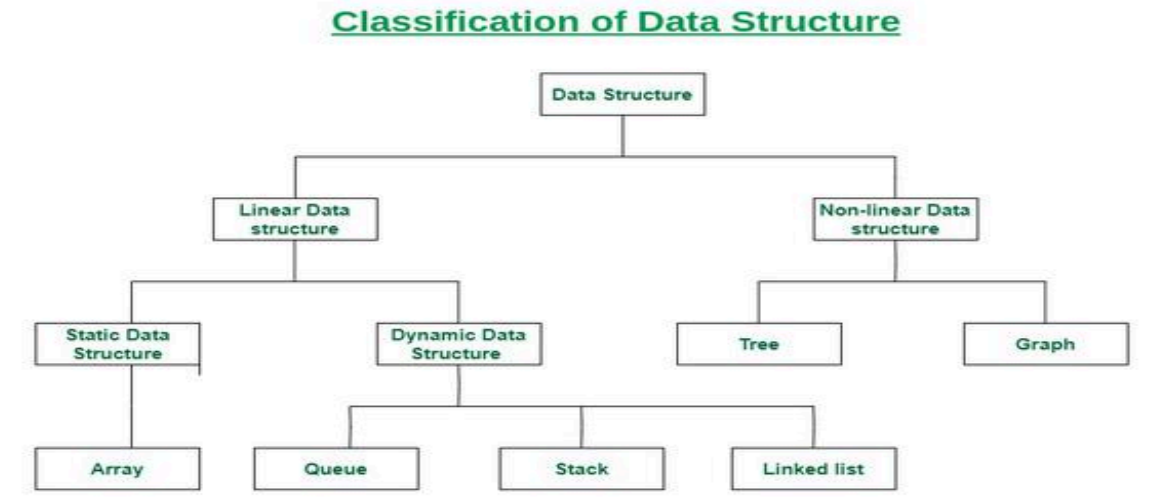
MODULE 1

Basic Concepts of Data Structures

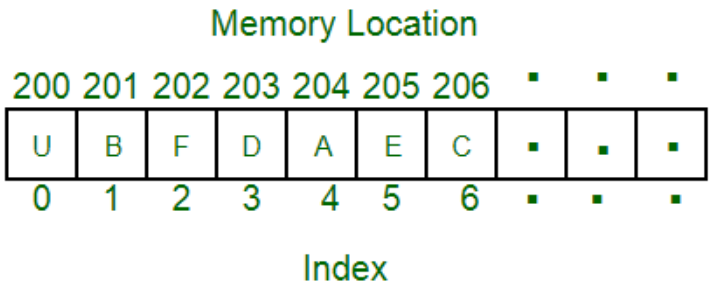
Definitions; Data Abstraction; **Performance Analysis** - Time & Space Complexity, Asymptotic Notations; Polynomial representation using Arrays, Sparse matrix (Tuple representation); **Stacks and Queues** - Stacks, Multi-Stacks, Queues, Circular Queues, Double Ended Queues; **Evaluation of Expressions**- Infix to Postfix, Evaluating Postfix Expressions.

DATA STRUCTURE

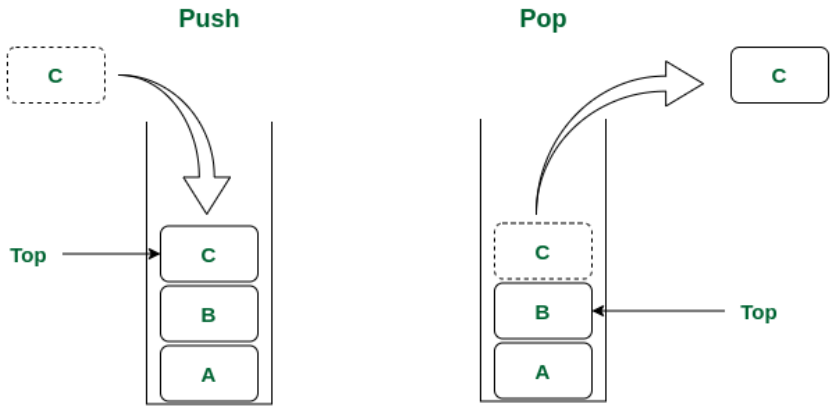
- Data structures are a way of organizing and storing data in a computer so that it can be used efficiently.
- It is a fundamental concept of computer science
- It represents the knowledge of data to be organized in memory.
- A data structure is a particular way of organizing data in a computer. Not only organizing the data. It also used for organizing, processing, retrieving and
- storing data.
- A data structure is a specialized format for organizing, processing, retrieving and
- storing data.
- Data: Data is a raw and unorganized fact that is required to be processed to make it
- meaningful
- Structure: a way of organizing.



- **<u>Linear Data Structure:</u>** A data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure. Examples are array, stack, queue, etc.
    - **<u>Static data structure:-</u>** In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.
    - Eg:- **ARRAY** -An array is a collection of items of the same data type stored at contiguous memory locations.

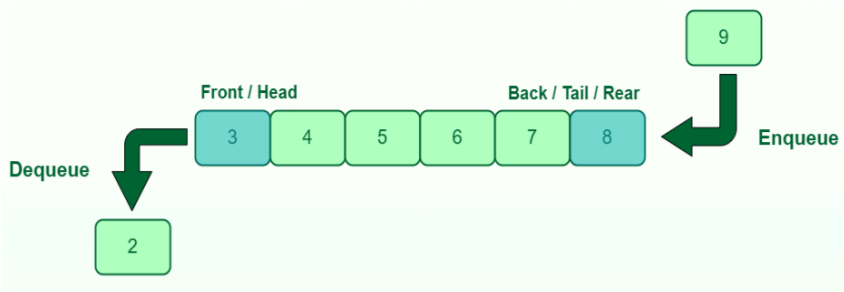


    - **<u>Dynamic data structure:-</u>** In Dynamic data structure the size of the structure is not fixed and can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time.
    - Eg:- 1) **STACK** - A **stack** is a linear data structure that follows the **Last-In-First-Out (LIFO)** principle, meaning that the last element added to the stack is the first one to be removed.



Stack Data Structure

- 2) **QUEUE** -A **queue** is a linear data structure that follows the **First-In-First-Out (FIFO)** principle. In a queue, the first element added is the first one to be removed.

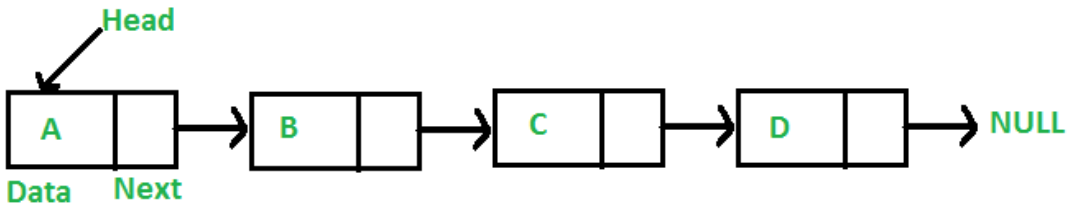


Queue Data Structure

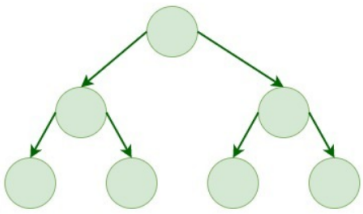
- 3) **LINKED LIST**-A L**inked List** is a linear data structure which looks like a chain of nodes, where each node contains a **data** field and a **reference(link)** to the next node in the list. Unlike Arrays, Linked List elements are not stored at a contiguous location.

Common Features of Linked List:

- **Node:** Each element in a linked list is represented by a node, which contains two components:
    - **Data:** The actual data or value associated with the element.
    - **Next Pointer(or Link):** A reference or pointer to the next node in the linked list.
- **Head:** The first node in a linked list is called the "head." It serves as the starting point for traversing the list.
- **Tail:** The last node in a linked list is called the "tail."



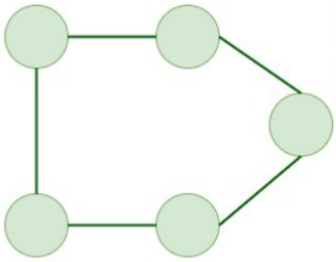
- **Non-linear Data Structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. Examples are trees and graphs.
    - TREES:- A tree is a hierarchical data structure where each element (called a node) is connected to one or more nodes in a parent-child relationship. Trees are often used to represent hierarchical structures, such as file systems or organizational charts.



Tree

    - GRAPH:- A graph is a collection of nodes (vertices) and edges, where edges connect the nodes. Graphs are more general and flexible than trees and can

represent complex relationships, such as social networks, transportation systems, or web pages.



Graph

Linear Data Structure	Non-Linear Data Structure
Elements are stored successively.	Elements are stored in a hierarchical or interconnected way.
Examples of Linear Data Structure such as Arrays, Linked Lists, Stacks, Queues.	Examples of Non-Linear Data Structures such as Trees and Graphs.
The traversal is sequential, one element after another.	The traversal can be complex, and multiple paths can be followed.
Contiguous memory allocation.	Memory is allocated dynamically, not in sequence.
Each element has a single successor.	Each element can have multiple successors.
Implementing simple data operations.	Representing complex relationships like social networks, and file systems.

Applications of Data Structures:

Data structures are used in a wide range of computer programs and applications, including:

- **Databases:** Data structures are used to organize and store data in a database, allowing for efficient retrieval and manipulation.

- **Operating systems:** Data structures are used in the design and implementation of operating systems to manage system resources, such as memory and files.
- **Computer graphics:** Data structures are used to represent geometric shapes and other graphical elements in computer graphics applications.
- **Artificial intelligence:** Data structures are used to represent knowledge and information in artificial intelligence systems.

Advantages of Data Structures:

The use of data structures provides several advantages, including:

- **Efficiency:** Data structures allow for efficient storage and retrieval of data, which is important in applications where performance is critical.
- **Flexibility:** Data structures provide a flexible way to organize and store data, allowing for easy modification and manipulation.
- **Reusability:** Data structures can be used in multiple programs and applications, reducing the need for redundant code.
- **Maintainability:** Well-designed data structures can make programs easier to understand, modify, and maintain over time.

ALGORITHMS

- Definition: An ***algorithm*** is a finite set of instructions to accomplish a particular task.
- An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- An algorithm is thus a sequence of computational steps that transform the input into the output
- An algorithm is independent of the programming language.
- In addition, all algorithms must satisfy the following criteria:

  (1) ***Input***. There are zero or more quantities that are externally supplied.

(2) ***Output***. At least one quantity is produced.

(3) ***Definiteness***. Each instruction is clear and unambiguous.

(4) ***Finiteness***. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

(5) ***Effectiveness***. Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible.

PERFORMANCE ANALYSIS

An algorithm is said to be efficient and fast, if it takes **less time to execute** & **consume less memory space**

Performance is analyzed based on 2 criteria

1. Space Complexity
2. Time Complexity

❖ Space Complexity

- Analysis of **space complexity of an algorithm** or program is the **amount of memory it needs to run to completion**.
- The space needed by a program consists of the following components.
    - **Fixed space requirements:** Independent on the number and size of the programs input and output. It include
        - Instruction Space (Space needed to store the code)
        - Space for simple variable
        - Space for constants
    - **Variable space requirements:** This component consists of
        - Space needed by structured variable whose size depends on the particular instance *I* of the problem being solved

- Space required when a function uses recursion

- Total Space Complexity **S(P)** of a program is

$$\mathbf{S(P) = C + S_p(I)}$$

Here $\mathbf{S_p(I)}$ is Variable space requirements of program P working on an instance *I*.

C is a constant representing the fixed space requirements

- Example :

```
int sum(int A[], int n)
{
    int sum=0, i;
    for(i=0;i<n;i++)
    {
        Sum=sum+A[i];
        return sum;
    }
}
```

ANSWER:- From the given code, it is the variable space requirement. Because, there is no loop condition and recursion. So the variables are n,sum,i,&A[ ]. All 4 variables stored in the 4 bytes of integer datatype.

n= 4 bytes

sum= 4 bytes

i= 4 bytes

A[ ]= 4 bytes + n(times) bytes

The total space complexity of a given code is

=n+sum+i+A[ ]

= 4+4+4+4n = 4n+12 bytes

```
void main()
    {
        int x,y,z,sum;
        printf("Enter 3 numbers");
        scanf("%d%d%d",&x,&y,&z);
        sum = x+y+z;
        printf("The sum = %d",sum);
    }
```

ANSWER:- From the given code, it is the fixed space requirement. Because, there is no loop condition and recursion. So the variables are x,y,z,&sum. All 4 variables stored in the 4 bytes of integer datatype.

x= 4 bytes

y= 4 bytes

z= 4 bytes

sum= 4 bytes

The total space complexity of a given code is

=x+y+z+sum

= 4+4+4+4 = 16 bytes

❖ Time Complexity

- The **time complexity of an algorithm** or a program is the **amount of time it needs to run to completion**.
- **$T(P)=C+T_P$**

  Here **C** is compile time

  T_p is Runtime

- For calculating the time complexity, we use a method called **Frequency Count** ie, counting the number of steps
  - Comments – 0 step
  - Assignment statement – 1 Step
  - Conditional statement – 1 Step
  - Loop condition for 'n' numbers – n+1 Step
  - Body of the loop – n step
  - Return statement – 1 Step
- Examples:

  How to calculate time complexity of a program?

		Frequency count
1. Int sum(int a[ ], int n)	—>	0
{		
s=0;	—>	1
for(i=0; i < n; i++)	—>	1+(n+1)+n
s=s+a[i];	—>	n
Return s;	—>	1
}		
		3n+4

- When we analyze an algorithm it depends on the input data, there are three cases :
  a. **Best case:** The best case is the minimum number of steps that can be executed for the given parameters.
  b. **Average case:** The average case is the average number of steps executed on instances with the given parameters.
  c. **Worst case**: In the worst case, is the maximum number of steps that can be executed for the given parameters

ASYMPTOTIC NOTATION

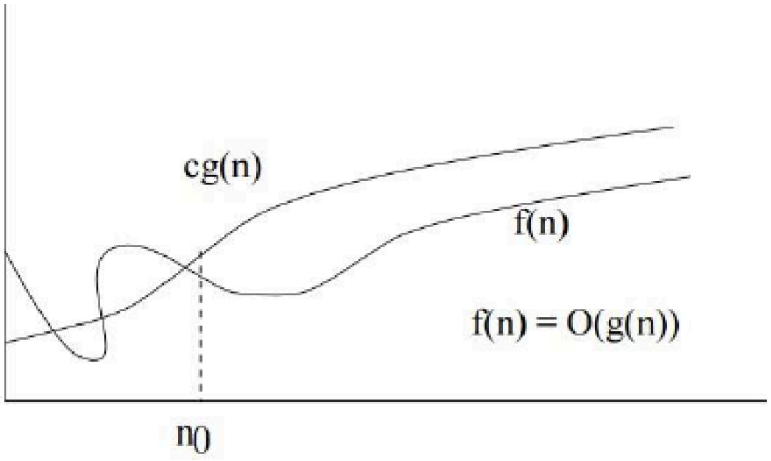
- Complexity of an algorithm is usually a function of n.
- Behavior of this function is usually expressed in terms of one or more standard functions.
- Expressing the complexity function with reference to other known functions is called **asymptotic complexity**.
- Three basic notations are used to express the asymptotic complexity

1. **Big – Oh notation O :** Upper bound of the algorithm
2. **Big – Omega notation Ω :** Lower bound of the algorithm
3. **Big – Theta notation Θ :** Average bound of the algorithm

1. Big – Oh notation O

- Formal method of expressing the upper bound of an algorithm's running time.
- i.e. it is a measure of the longest amount of time it could possibly take for an algorithm to complete.
- It is used to represent the **worst case** complexity.
- **f(n) = O(g(n))** if and only if there are two positive constants **c** and **n0** such that

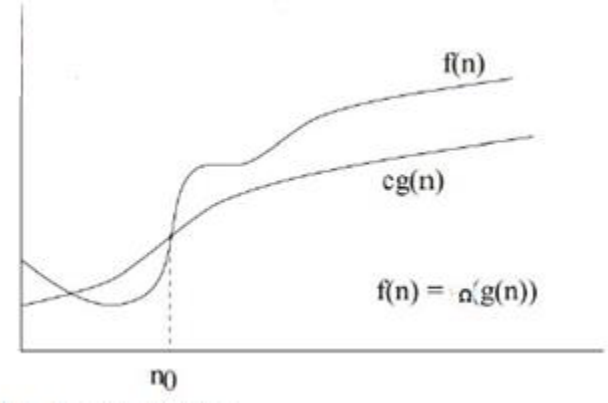
  f(n) ≤ c g(n) for all **n ≥ n0** .
- Then we say that **"f(n) is big-O of g(n)"**.



2. **Big – Omega notation Ω**

- **f(n) = Ω(g(n))** if and only if there are two positive constants **c** and **n0** such that

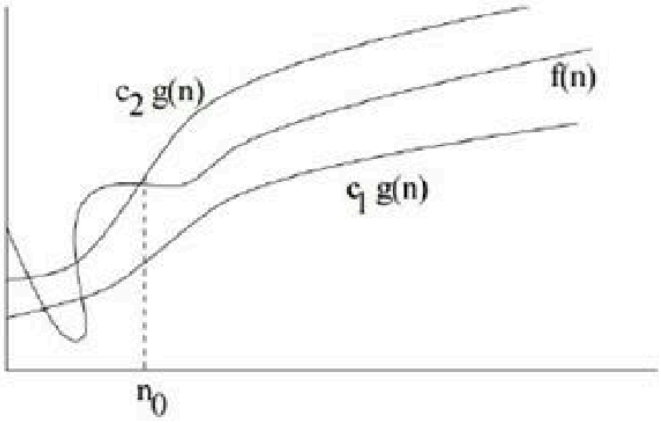
  f(n) ≥ c g(n) for all **n ≥ n0.**
- Then we say that **"f(n) is omega of g(n)"**.



3. **Big – Theta notation Θ**

- **f(n) = Θ(g(n))** if and only if there are three positive constants c1, c2 and n0 such that

  c1 g(n) ≤ f(n) ≤ c2 g(n) for all **n ≥ n0** .
- Then we say that **"f(n) is theta of g(n)"**.



POLYNOMIAL REPRESENTATION USING ARRAYS

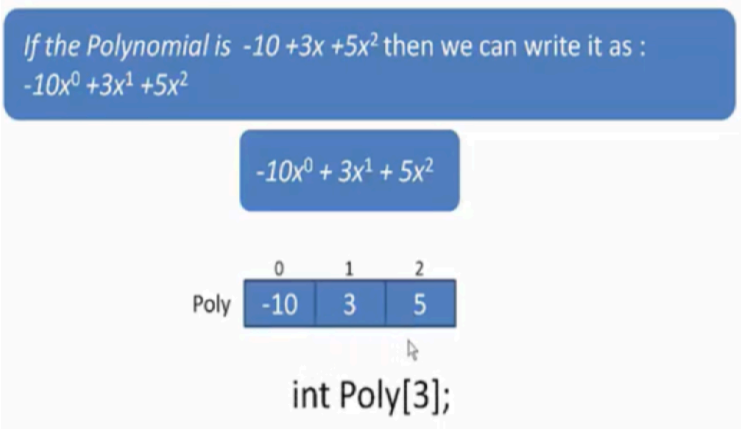
- A polynomial is a sum of terms where each term has the form $\mathbf{ax^e}$, Where x is the variable, a is the coefficient and e is the exponent.

A general polynomial $A(x)$ can be written as

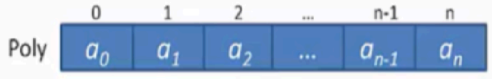
$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

where $a_n \neq 0$ and we say that the degree of A is n.

Polynomial representation using Arrays



A polynomial of a single variable $A(x)$ can be written as $a_0 + a_1X + a_2X^2 + \ldots + a_nX^n$ where $a_n \neq 0$ and degree of $A(X)$ is n.



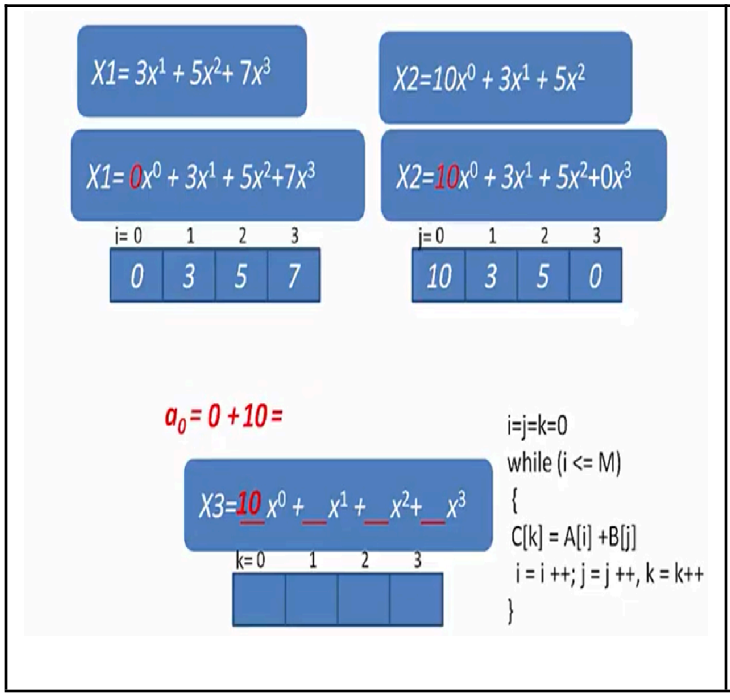
For a polynomial of degree n, n+1 terms are required

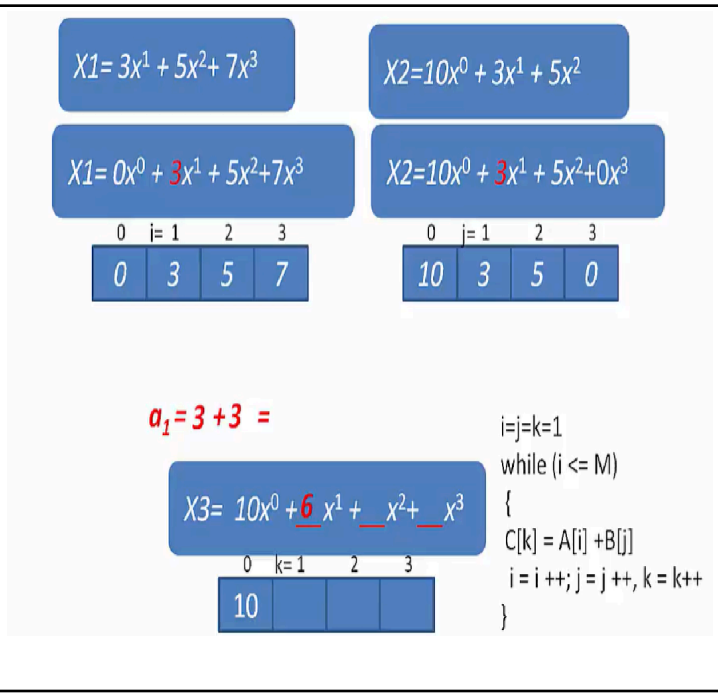
$X1 = 3x^1 + 5x^2 + 7x^3$ — Degree of X1 is M=3

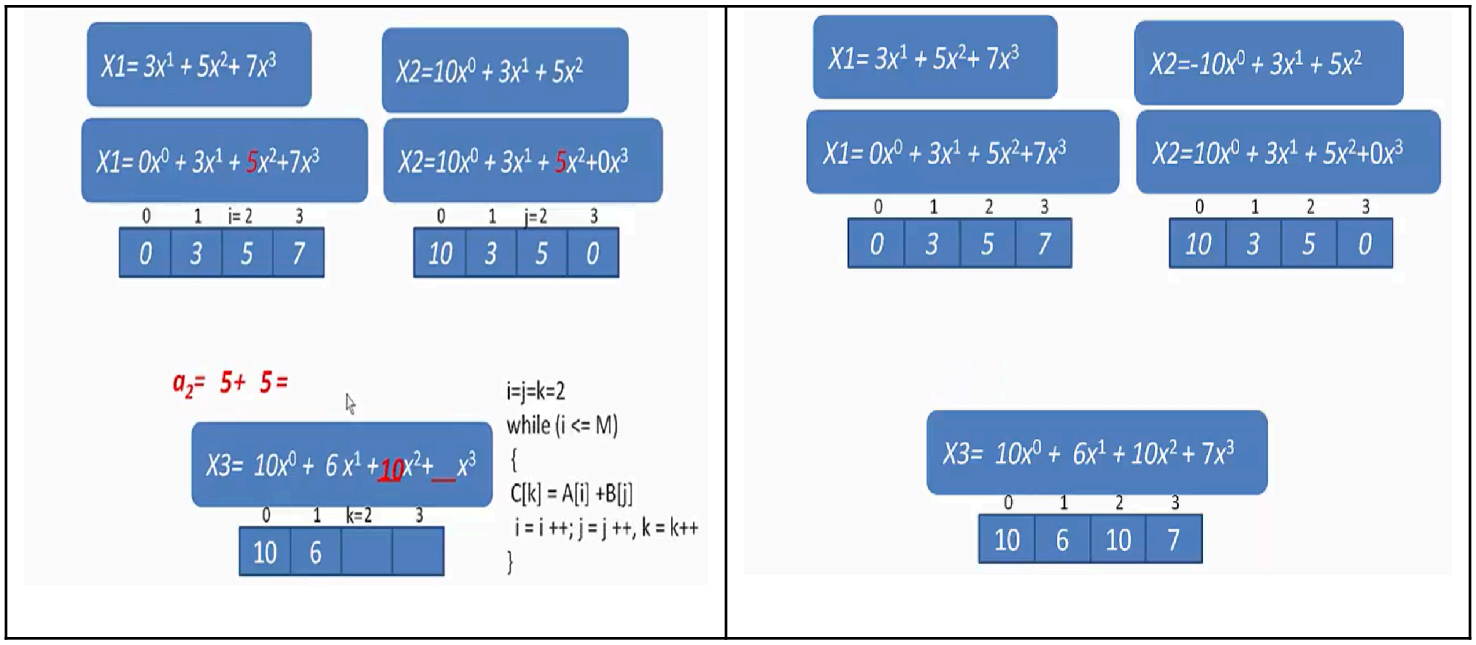
$X2 = 10x^0 + 3x^1 + 5x^2$ — Degree of X2 is N=2

- Identify the value of Highest degree polynomial.
- Write polynomial X3 with degree Max(degree of X1 and degree of X2).

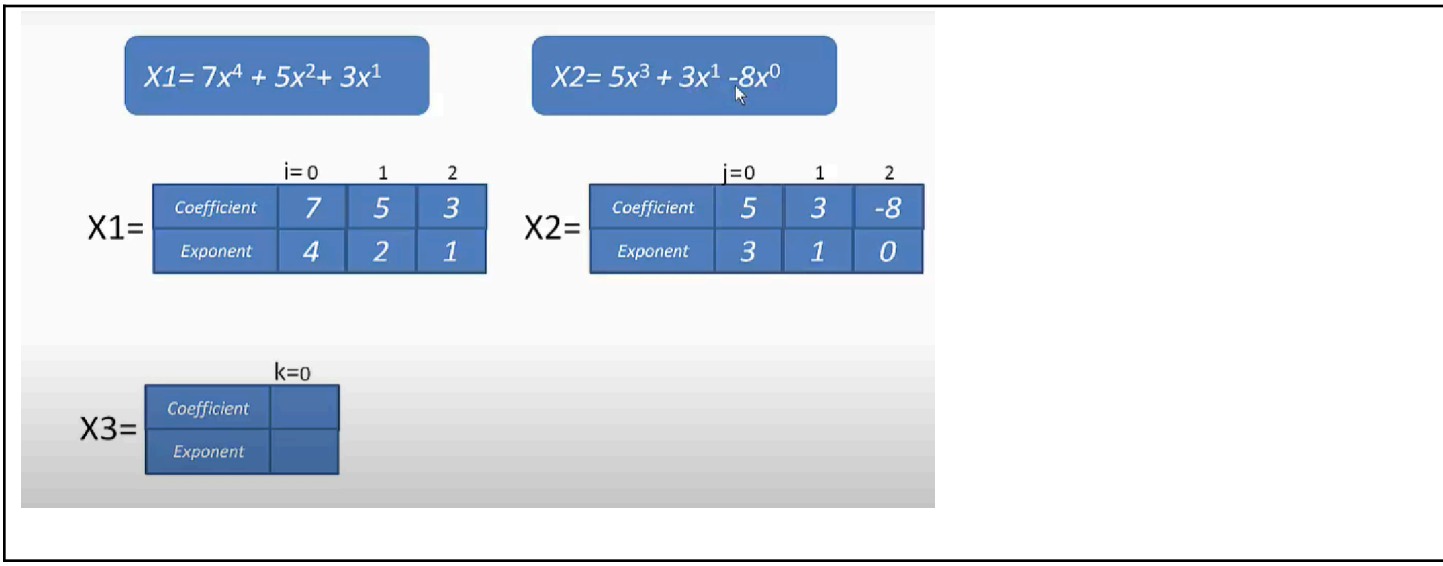
Polynomial Addition Example







Steps of Polynomial Addition



$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	i=0	1	2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	j=0	1	2
Coefficient	5	3	-8
Exponent	3	1	0

4 > 3

X3=

	k=0
Coefficient	
Exponent	

CASE-1

If the exponent of the term pointed by j in X2 is less than the exponent of the current term pointed by i of X1 , then copy the current term of X1 pointed by i in the location pointed by k in polynomial X3. Advance the pointer i and k to the next term.

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	i=0	1	2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	j=0	1	2
Coefficient	5	3	-8
Exponent	3	1	0

X3=

	k=0
Coefficient	7
Exponent	4

```
if(X1[i].expo > X2[j].expo)
{
 X3[k]. coeff = X1[i]. coeff;
 X3[k].expo = X1[i].expo;
 i = i + 1
 k = k + 1
}
```

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	0	i=1	2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	j=0	1	2
Coefficient	5	3	-8
Exponent	3	1	0

X3=

	0	k=1	2
Coefficient	7	5	
Exponent	4	3	

CASE-2

If the exponent of the term pointed by j in X2 is greater than the exponent of the current term pointed by i of X1, then copy the current term of X2 pointed by j in the location pointed by k in polynomial X3. Advance the pointer j and k to the next term.

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	0	i=1	2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	j=0	1	2
Coefficient	5	3	-8
Exponent	3	1	0

X3=

	0	k=1	2
Coefficient	7	5	
Exponent	4	3	

```
if(X1[i].expo < X2[j].expo)
{
 X3[k]. coeff = X2[j]. coeff;
 X3[k].expo = X2[j].expo;
 j = j + 1
 k = k + 1
}
```

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	0	1	i=2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	0	j=1	2
Coefficient	5	3	-8
Exponent	3	1	0

X3=

	0	1	2	k=3
Coefficient	7	5	5	
Exponent	4	3	2	

```
if(X1[i].expo > X2[j].expo)
{
X3[k]. coeff = X1[i]. coeff
X3[k].expo = X1[i].expo;
i = i + 1
k = k + 1
```

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	0	1	i=2
Coefficient	7	5	3
Exponent	4	2	1

X2=

	0	j=1	2
Coefficient	5	3	-8
Exponent	3	1	0

1 = 1

X3=

	0	1	2	k=3	4
Coefficient	7	5	5		
Exponent	4	3	2		

CASE-3

If the exponents of the two terms of polynomials X1 and X2 are equal, then the coefficients are added, and the new term is stored in the resultant polynomial X3 and advance I, j and k to track to the next term.

$X1 = 7x^4 + 5x^2 + 3x^1$

$X2 = 5x^3 + 3x^1 - 8x^0$

X1=

	0	1	i=2
Coefficient	7	5	3
Exponent	4	2	1

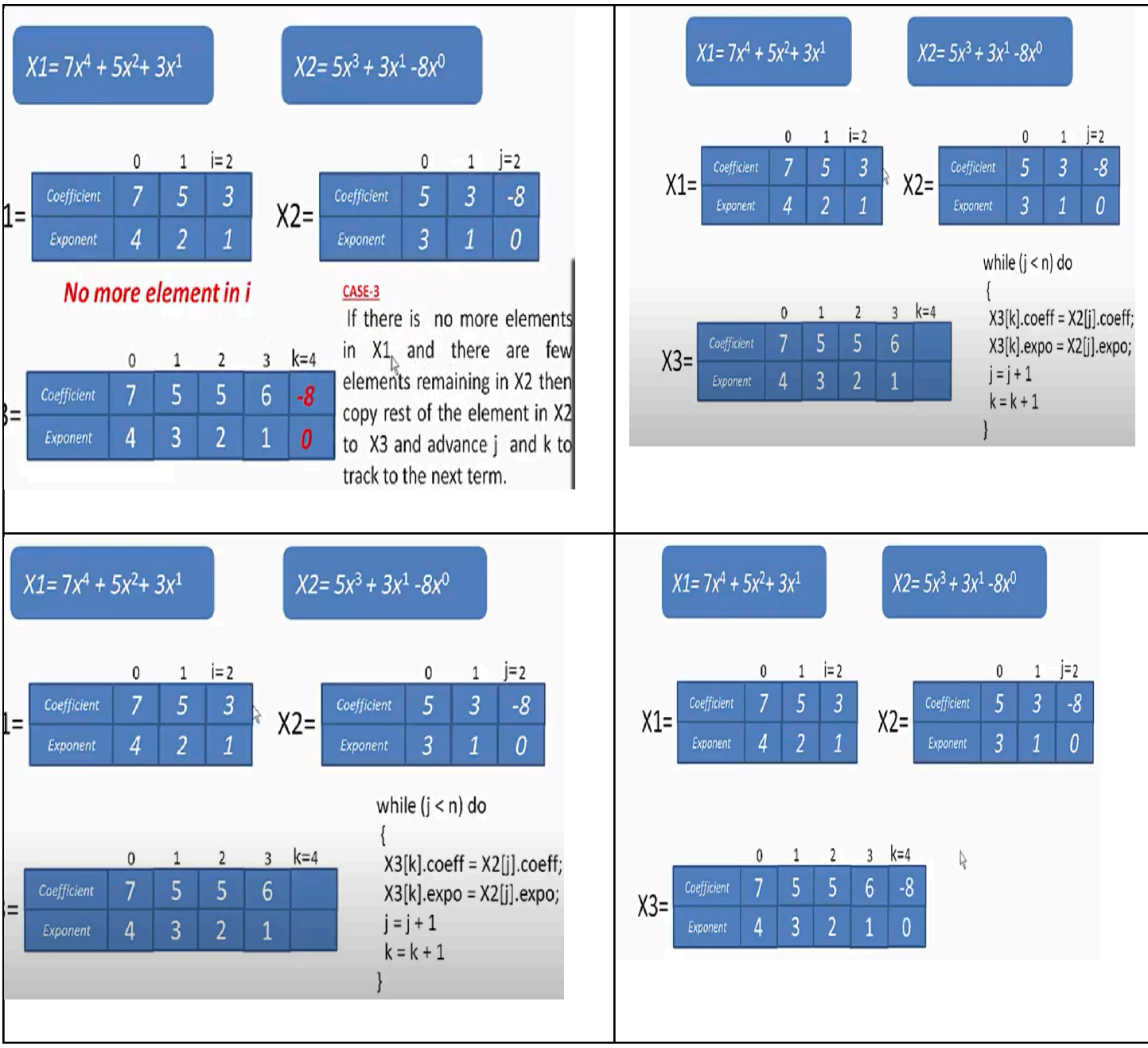
X2=

	0	1	j=2
Coefficient	5	3	-8
Exponent	3	1	0

X3=

	0	1	2	3	k=4
Coefficient	7	5	5	6	
Exponent	4	3	2	1	

```
if(X1[i].expo == X2[j].expo)
{
X3[k].coeff = X1[i].coeff +
                        X2[j].coeff
X3[k].expo = X1[i].expo;
i = i + 1
j=j+1
k = k + 1
```



1. SPARSE MATRIX

- A matrix is a two-dimensional data object made of 'm' rows and 'n' columns, therefore having total m x n values. If most of the elements of the matrix have 0 values, then it is called a **sparse matrix**.
- **A sparse matrix is a matrix which contains very few non-zero elements.**

Example:

```
0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0
```

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

Triplets

(0,2,3)

(0,4,4)

(1,2,5)

(1,3,7)

(3,1,2)

(3,2,6)

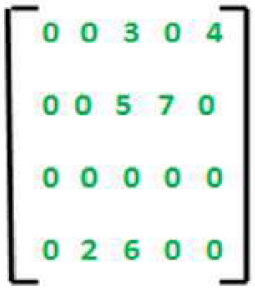
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays:

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

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



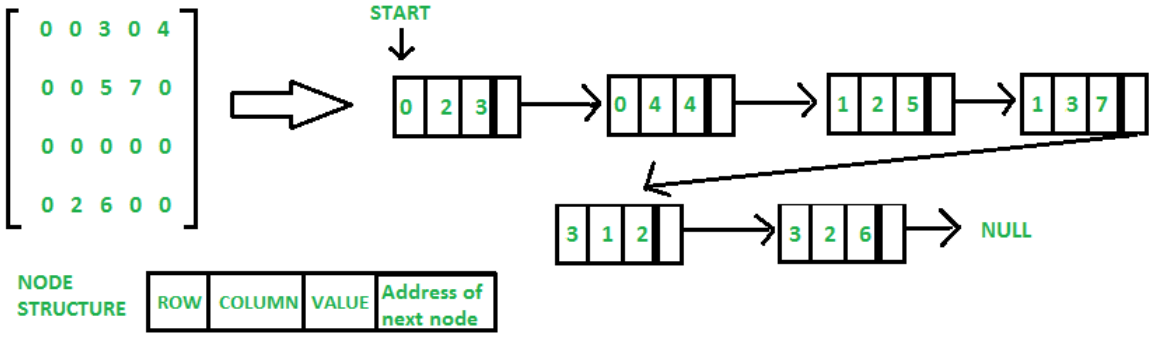
Row	Column	Value
0	2	3
0	4	4
1	2	5
1	3	7
3	1	2
3	2	6

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Method 2: Using Linked Lists

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

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



Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.

- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements.

1. STACK

- It is a linear data structure in which elements are placed one above another.
- A stack is an ordered collection of homogeneous data elements where the insertion and deletion operations take place only at one end called **Top** of the stack.
- **LIFO -** In stack elements are arranged in **Last-In-First-Out** manner (**LIFO**). So it is also called LIFO lists.
- Anything added to the stack goes on the "**top**" of the stack.
- Anything removed from the stack is taken from the "**top**" of the stack.
- Things are removed in the reverse order from that in which they were inserted

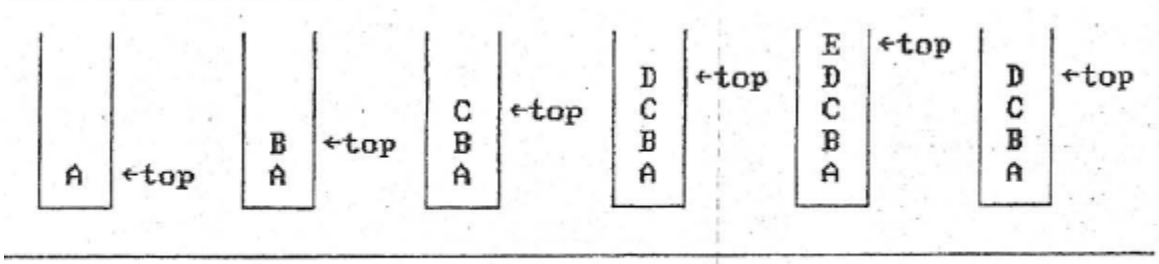
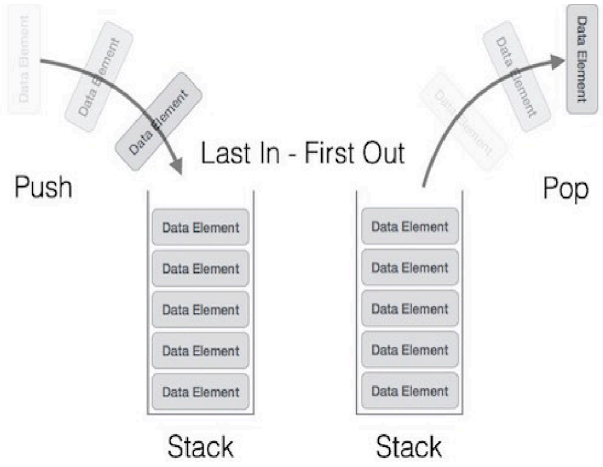


Figure 3.1: Inserting and deleting elements in a stack

Operations of Stack

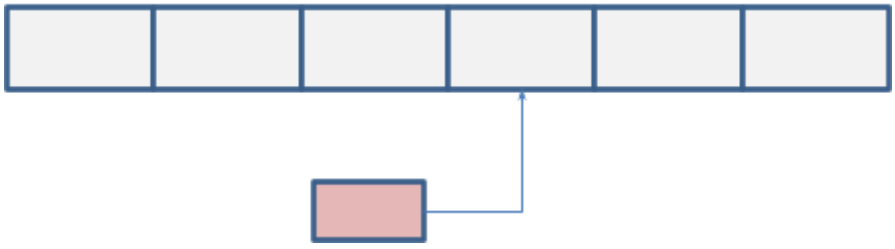
- Two basic operations of stack:
  - **PUSH :** Insert an element at the top of stack
  - **POP:** Delete an element from the top of stack



- An element in the stack is termed as **ITEM.**
- Initially top is set to -1, to indicate an **empty stack**. **(Top = -1)**
- The maximum no. of elements that a stack can accommodate is termed **MAX_SIZE**.
- If stack is full **Top = MAX_SIZE - 1**

Array representation of stack

- Stack can be represented using a linear array.
- There is a pointer called TOP to indicate the top of the stack



top

- **<u>Overflow:</u>** If we try to insert a new element in the stack top (push) which is already **full**, then the situation is called stack overflow.
- **<u>Underflow:</u>** If we try to delete an element (pop) from an **empty** stack, the situation is called stack underflow.

Basic Operations

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.
- **peek()** − get the top data element of the stack, without removing it.

```
int peek() {
        return stack[top];
 }
```

- **isFull()** − check if stack is full.

```
bool isfull() {
        if (top == MAX_SIZE)
                return true;
         else
         return false;
}
```

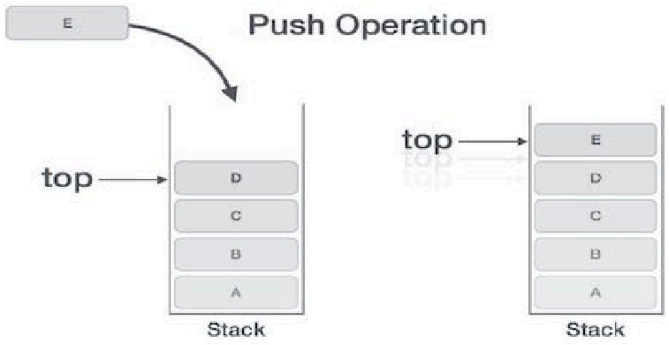
- **isEmpty()** − check if stack is empty.

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

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.



Algorithm: PUSH()

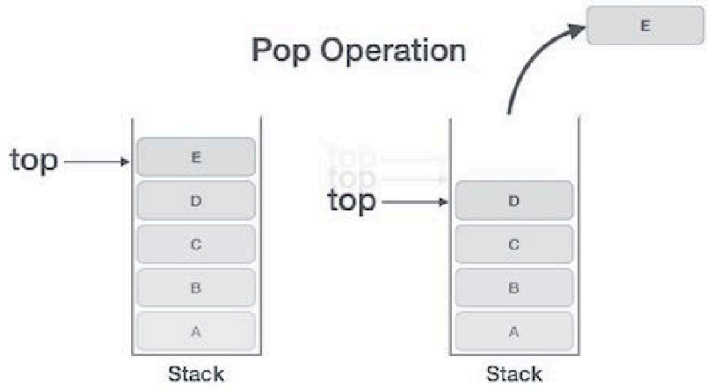
- Let A be an array with Maximum size as MAX_SIZE. Initially, top= -1

```
Start
if top < MAX_SIZE – 1
    set top=top+1
    Set A[top]=item
else
     print “OVERFLOW”
exit
```

POP Operation

A Pop operation may involve the following steps –

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



Algorithm: POP()

```
Start
if  top= -1 then
    print "UNDERFLOW"
else
   set  item=A[top]
   Set  top=top-1
exit
```

Applications of stack

- Reversing an array
  - A B C D
  - Pushing to stack A B C D
  - Popping from stack D C B A
- Undo operations

- Infix to prefix, infix to postfix conversion
- Tree Traversal
- Evaluation of postfix expressions

2. QUEUES

- A queue is an ordered collection of homogeneous data elements. In which insertion is done at one end called **REAR** and deletion is done at another end called **FRONT**.
- **FIFO -** In queue elements are arranged in **First-In-First-Out** manner (**FIFO**).
- First inserted element is removed first

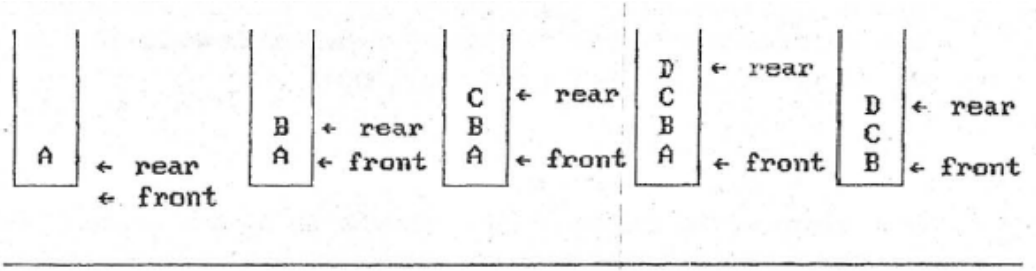
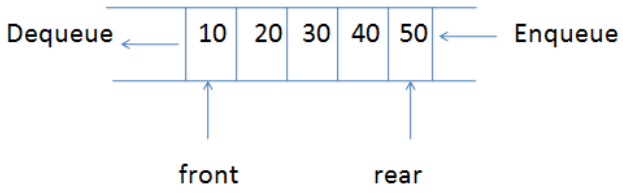


Figure 3.4: Inserting and deleting elements in a queue

- Two basic operations of queue:
  1. **Enqueue ->** Insert an element at the **rear** end of queue.
  2. **Dequeue->** Delete an element from the **front** end of queue.



- Initial case **rear = -1** and **front = 0**, **MAX SIZE** is the size of the queue.
- If **rear = front** then queue contains only a **single element**
- If **rear < front** then queue is **empty**
- **Queue full :** rear = n-1 and front =0

- Whenever an element is deleted from the queue, the value of FRONT is increased by 1.
- i.e. FRONT=FRONT+1
- Similarly, whenever an element is added to the queue, the REAR is incremented by 1 as,
- REAR=REAR+1

Array Representation of Queue

A one-dimensional array, say $Q[1 \ldots N]$, can be used to represent a queue. Figure 5.3 shows an instance of such a queue. With this representation, two pointers, namely FRONT and REAR, are used to indicate the two ends of the queue. For the insertion of the next element, the pointer REAR will be the consultant and for deletion the pointer FRONT will be the consultant.

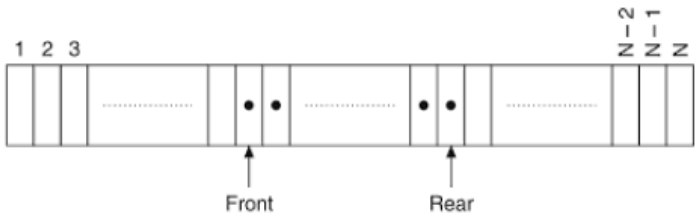


Figure 5.3 Array representation of a queue.

Basic Operations

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.
- **peek()** − Gets the element at the front of the queue without removing it.

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

- **isfull()** − Checks if the queue is full

```
bool isfull()
{
        If (rear == MAXSIZE - 1)
```

```
            return true;
        else
            return false;
    }
```

- **isempty()** − Checks if the queue is empty.

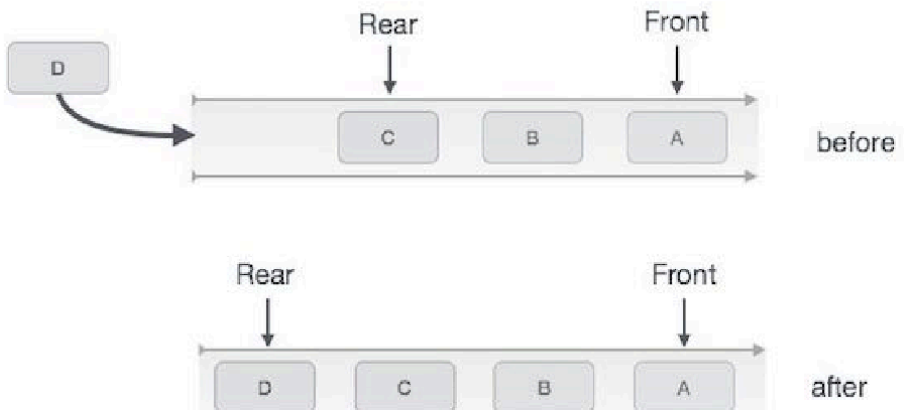
```
    bool isempty()
    {
        if(front < 0 || front > rear)
            return true;
        else
            return false;
    }
```

Enqueue Operation

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

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

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



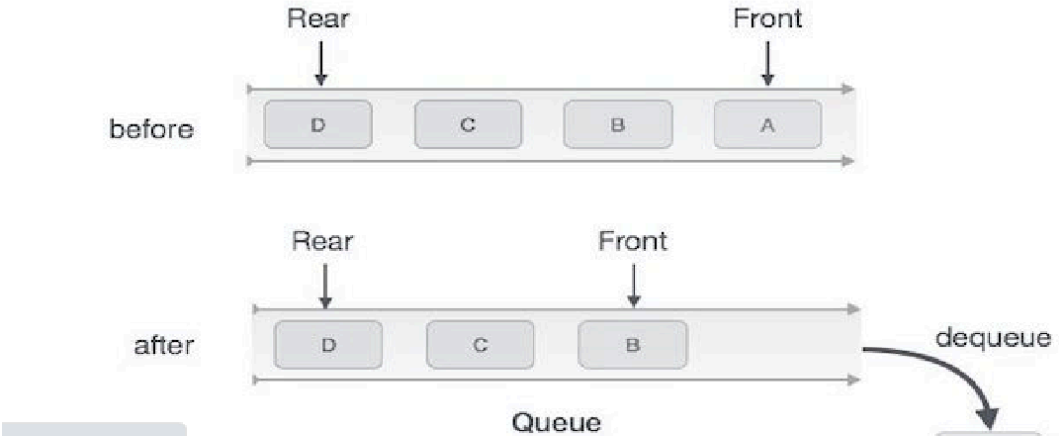
Algorithm : Enqueue

```
Start
if rear = MAX_SIZE – 1 then
        print "OVERFLOW"
else
      set rear = rear + 1
      Set A[rear]=item
exit
```

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.



Algorithm : Dequeue

```
Start
if rear < front then
      print "UNDER FLOW"
else
     set  item = A[front}
     set front = front + 1
exit
```

Type of Queues

- Circular Queue
- Priority Queue
- Doubly ended Queue

3. CIRCULAR QUEUE

- To utilize space properly, a circular queue is derived.
- In this queue the elements are inserted in a circular manner.
- So that no space is wasted at all.
- Circular queue empty:

  FRONT= -1

  REAR= -1
- Circular queue full:

  (rear + 1) % max_size = Front
- It is a modification of simple queue in which the rear pointer is set to the initial location, whenever it reaches the location max_size – 1.

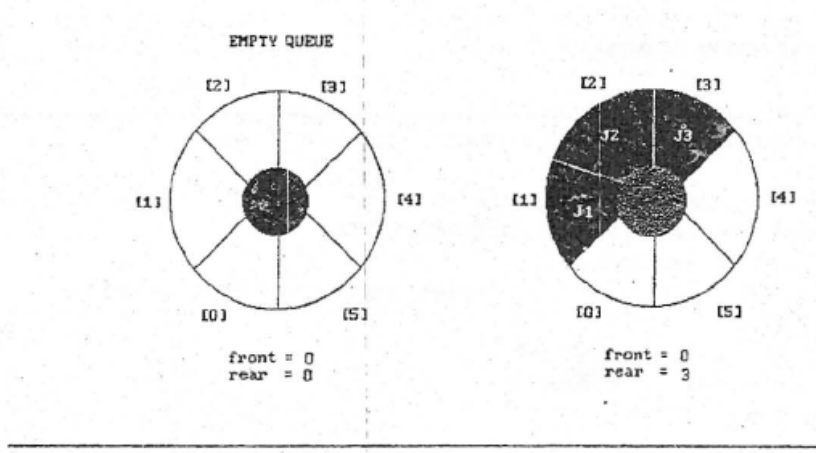


Figure 3.6: Empty and nonempty circular queues

Insertion Algorithm (ENQUEUE)

```
if (front == -1 & rear == -1)
        set front =0 and rear = 0
        Set a[rear]=item
else if (front = (rear+1) % max_size) then
         Print over flow
else
        set rear = (rear + 1)% max_size
         Set a[rear] = item
Exit
```

Deletion Algorithm (DEQUEUE)

```
if front = -1 and rear = -1 then
        print underflow and exit
else if front = rear
        set item= a[front]
        set front = -1 and rear = -1
else
        set item= a[front]
        set front = (front + 1) % max_size
Exit
```

4. PRIORITY QUEUE

- Regular queue follows a First In First Out (FIFO) order to insert and remove an item. Whatever goes in first, comes out first.
- In a priority queue, an item with the highest priority comes out first.
- Therefore, the FIFO pattern is no longer valid.
- Every item in the priority queue is associated with a priority.
- It does not matter in which order we insert the items in the queue
- The item with higher priority must be removed before the item with the lower priority.
- If two elements have the same priority, they are served according to their order in the queue.

Operations on a priority queue

1. **EnQueue**: EnQueue operation inserts an item into the queue. The item can be inserted at the end of the queue or at the front of the queue or at the middle. The item must have a priority.
2. **DeQueue**: DeQueue operation removes the item with the highest priority from the queue.

3. **Peek**: Peek operation reads the item with the highest priority.

1. **Enqueue Operation**
    1. IF((Front == 0)&&(Rear == N-1))
    2. PRINT “Overflow Condition”
    3. Else IF(Front == -1& rear == -1)
    4. Front = Rear =0
    5. Queue[Rear] = Data
    6. Priority[Rear] = Priority
    7. ELSE IF(Rear ==N-1)
    8. FOR (i=Front;i<=Rear;i++)
    9. FOR(i=Front;i<=Rear;i++)
    10. Q[i-Front] =Q[i]
    11. Pr[i-Front] = Pr[i]
    12. Rear = Rear-Front
    13. Front = 0
    14. FOR(i = r;i>f;i–)
    15. IF(p>Pr[i])
    16. Q[i+1] = Q[i] Pr[i+1] = Pr[i]
    17. ELSE
    18. Q[i+1] = data Pr[i+1] = p
    19. Rear++.
2. **Dequeue operation**
    1. IF(Front == -1)
    2. PRINT “Queue Under flow condition”
    3. ELSE
    4. PRINT”Q[f],Pr[f]”
    5. IF(Front==Rear)

6. Front = Rear = -1
7. ELSE
8. FRONT++

Applications of Priority Queue

1. CPU Scheduling
2. Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3. All queue applications where priority is involved.
4. For load balancing and interrupt handling in an operating system

5. DOUBLY ENDED QUEUE

It is a list of elements in which insertion and deletion are perform at both ends

- It has 4 operations
  1. Insertion at rear end
  2. Insertion at front end
  3. Deletion at rear end
  4. Deletion at front end

1. **Algorithm : Insertion at rear end**

```
Start
if rear = MAX_SIZE – 1 then
        print "OVERFLOW"
Else
     set rear = rear + 1
      Set A[rear]=item
exit
```

2. **Insertion at front end**

```
Start
if front = 0 then
    print "OVERFLOW" and exit
Else
    set front = front - 1
    Set A[front]=item
exit
```

3. **Deletion at front end**

```
Start
if front = 0 and rear = -1 then
     print "UNDER FLOW" and exit
set item = A[front]
if front = rear then
    set front = 0 and rear = -1
Else set front = front + 1
exit
```

4. **Deletion at rear end**

```
Start
if front = 0 and rear = -1 then
    print "UNDER FLOW" and exit
set item = A[rear]
if front = rear then
    set front = 0 and rear = -1
Else set rear = rear - 1
exit
```

CONVERSION & EVALUATION OF EXPRESSIONS

- **Infix Expression:** The operator occurs between the operands

  <operand> <operator> <operand>

  Eg: a+b

- **Prefix Expression (Polish notation):** The operators occurs before the operand

  <operator> <operand> <operand>

  Eg : +ab

- **Postfix Expression (Reverse Polish notation):** The operators occurs after the operand

  <operand> <operand> <operator>

  Eg : ab+

Token	Operator	Precedence[1]	Associativity
() [] –> .	function call array element struct or union member	17	left-to-right
–– ++	increment, decrement[2]	16	left-to-right
–– ++ ! ~ – + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ –	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += –= /= *= %= <<= >>= &= ^= |=	assignment	2	right-to-left
,	comma	1	left-to-right

Infix to prefix

? a + b * c

Ans: a + b * c [Two operands + & * consider the highest one]
(* is highest)

a + * b c

+ a * b c

Infix to postfix

? a + b * c

Ans:- a + b * c

a + b c *

a b c * +

? A + (B * C - D / E) - F

Ans:- A + (B * C - D / E) - F

[Consider the expression inside bracket]

A + (* B C - / D E) - F

A + - * B C / D E - F

+ A - * B C / D E - F

- + A - * B C / D E F

? A + (B * C - D / E) - F

Ans:- A + (B * C - D / E) - F

A + (B C * - D E /) - F

A + (B C * D E / -) - F

A B C * D E / - + - F

A B C * D E / - + F -

Convert the following expression into postfix.

1. (A + (B * C - (D / E ↑ F) * G) * H)

First consider the inner bracket, that contain / & ↑ operands. ↑ has the highest degree.

(A + (B * C - (D / E ↑ F) * G) * H)

(A + (B * C - (D / EF ↑) * G) * H)

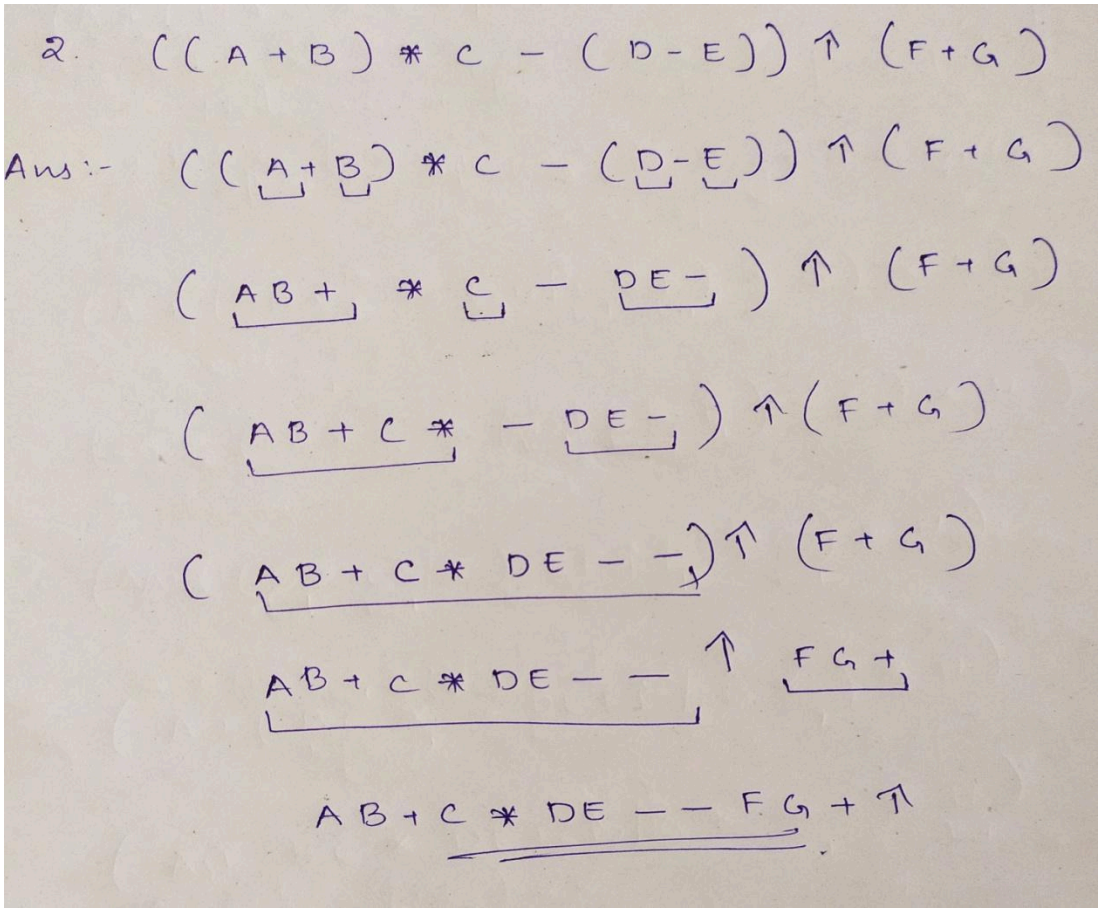
(A + (B * C - DEF ↑ / * G) * H)

(A + (BC * - DEF ↑ / G *) * H)

(A + BC * DEF ↑ / G * - * H)

(A + BC * DEF ↑ / G * - H *)

ABC * DEF ↑ / G * - H * +



A. Postfix Expression Evaluation

Given P is the postfix expression, the following algorithm uses a stack to hold operands. It finds the value of the arithmetic expression P, Written in postfix notation.

Algorithm:

Step 1: Add **") "** at the end of P

Step 2: Scan P from left – right & repeat the steps 3 & 4

Step 3: If an operand occurs, PUSH it to stack.

Step 4: If an operator ✕ occurs, then

A: Remove the top elements of the stack.

When A is the top element and B is the next top element

B: Evaluate B ✕ A

C: Place the result of step B back to stack

Step 5: Set the value equals to TOP element of the stack.

1. **Evaluate the expression 5 * (6 + 2) – 12 / 4**

   Ans : Convert to postfix notation

   5 * 6 2 + - 12 / 4

   5 6 2 + * - 12 4 /

   = 5 6 2 + * 12 4 / -

   Add **") "** at the end of P

   P = 5 6 2 + * 12 4 / -)

Scanned Symbol	**Stack**
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
-	37

2. **Evaluate the expression (6 + 2) / (4 – 2 * 1)**

   Ans: Convert to postfix notation

   6 2 + / (4 – 2 1 *)

   6 2 + / 4 2 1 * -

   6 2 + 4 2 1 * - /

   P = 6 2 + 4 2 1 * - / **)**

Scanned Symbol	**Stack**
6	6
2	6 2
+	8
4	8 4
2	8 4 2
1	8 4 2 1
*	8 4 2
-	8 2
/	4

B. Infix to Postfix conversion

Here the operators used are **^ , * , / , + , -.** The following algorithm converts an Infix expression Q to postfix expression P. This algorithm also uses a stack which holds the left parenthesis and operators. We begin by pushing a Left parenthesis to stack and adding a right parenthesis at the end of Q.

<u>Algorithm</u>

Step 1: PUSH left parenthesis **"("** into stack and add right parenthesis **") "** at the end of Q.

Step 2: Scan the expression Q from Left – Right and repeat the step 3 to 6 for each element of Q until this stack is empty.

Step 3: If an operand occurs add it to P.

Step 4: If a Left parenthesis occurs then PUSH it to stack

Step 5: If an operator $\times$ occurs then

A: Repeatedly POP the stack and add to P, each operator which has same or higher precedence than $\times$

B: add $\times$ to stack

Step 6: If a Right parenthesis occurs then

A: Repeatedly POP from stack and add to P each operator until a left parenthesis occurs.

B: Remove the left parenthesis

Step 7: Exit

1. **Q = A + (B * C - (D / E ^ F) * G) * H**

Ans : Add right parenthesis at the end of the expression

Q = A + (B * C - (D / E ^ F) * G) * H **)**

Symbol Scanned	**Stack**	**p**
	(	
A	(	A
+	(+	A

(	(+ (	A
B	(+ (	AB
*	(+ (*	AB
C	(+ (*	ABC
-	(+ (-	ABC*
(	(+ (- (	ABC*
D	(+ (- (	ABC*D
/	(+ (- (/	ABC*D
E	(+ (- (/	ABC*DE
^	(+ (- (/ ^	ABC*DE
F	(+ (- (/ ^	ABC*DEF
)	(+ (-	ABC*DEF ^ /
*	(+ (- *	ABC*DEF ^ /
G	(+ (- *	ABC*DEF ^ /G
)	(+	ABC*DEF ^ /G * -
*	(+ *	ABC*DEF ^ /G * -
H	(+ *	ABC*DEF ^ /G * - H
)		ABC*DEF ^ /G * - H * +

2. **Q = ((A + B) * C – (D – E)) ^ (F + G)**

Ans:

Q = ((A + B) * C – (D – E)) ^ (F + G))

Symbol Scanned	**Stack**	**p**
0	(	
(	((	
(	(((	
A	(((	A
+	(((+	A
B	(((+	AB
)	((	AB+
*	((*	AB+
C	((*	AB+C
-	((-	AB+C*
(	((- (	AB+C*
D	((- (	AB+C*D
-	((- (-	AB+C*D
E	((- (-	AB+C*DE
)	((-	AB+C*DE-

)	(	AB+C*DE--
^	(^	AB+C*DE--
(	(^ (	AB+C*DE--
F	(^ (	AB+C*DE--F
+	(^ (+	AB+C*DE--F
G	(^ (+	AB+C*DE--FG
)	(^	AB+C*DE--FG+
)		AB+C*DE—FG+^

3. **Q = (A + B) * C / D + E ^ F / G**

   Ans :

   Q = (A + B) * C / D + E ^ F / G)

Symbol Scanned	**Stack**	**p**
	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	AB+

*	(*	AB+
C	(*	AB+C
/	(/	AB+C*
D	(/	AB+C*D
+	(+	AB+C*D/
E	(+	AB+C*D/E
^	(+ ^	AB+C*D/E
F	(+ ^	AB+C*D/EF
/	(+ /	AB+C*D/EF^
G	(+ /	AB+C*D/EF^G
)		AB+C*D/EF^G/+

MODULE 2

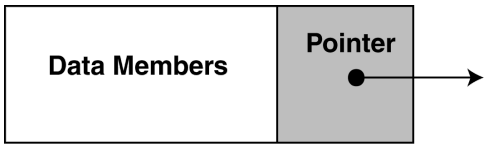
Linked List and Memory Management

Singly Linked List - Operations on Linked List, Stacks and Queues using Linked List, Polynomial representation using Linked List; Doubly Linked List; Circular Linked List; **Memory allocation** - First-fit, Best-fit, and Worst-fit allocation schemes; Garbage collection and compaction.

---

1. LINKED LIST

- A linked list is an ordered **collection of finite, homogeneous data elements called nodes** where the linear order is maintained by means of links or pointers.
- A linked list is a **dynamic data structure** where the amount of memory required can be varied during its use.
- In the linked list, the adjacency between the elements is maintained by means of **links or pointers.**
- A link or pointer actually is the **address** (memory location) of the subsequent element.
- An element in a linked list is a specially termed **node**, which can be viewed as shown in the figure.
- A node consists of two fields : **DATA** (to store the actual information) and **LINK** (to point to the next node)



- A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list.

- ***Head*:** pointer to the first node
- The last node points to NULL

- Depending on the requirements the pointers are maintained, and accordingly the linked list can be classified into **three major groups**:

1. Single linked list
2. Circular linked list
3. Double linked list.

1. SINGLE LINKED LIST

- In any single linked list, every **"Node"** contains two fields, **data** and **link**.
- The **data** field is used to store actual value of that node and **link** field is used to store the address of the next node in the sequence.
- Each node contains only one link which points to the subsequent node in the list.
- The header node points to the 1st node in the list
- The link field of the last node contain NULL(ϕ) value.
- Here one can move from left to right only. So it is also called one-way list

Representation of a linked list in memory

Two ways:

1. Static representation using array
2. Dynamic representation using free pool storage

1. Static representation

Two arrays are maintained:

- One for data and other for links.

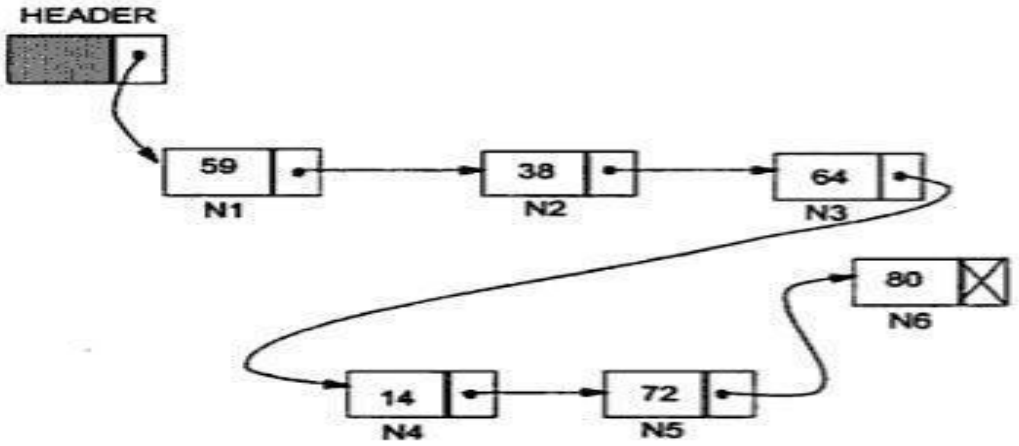
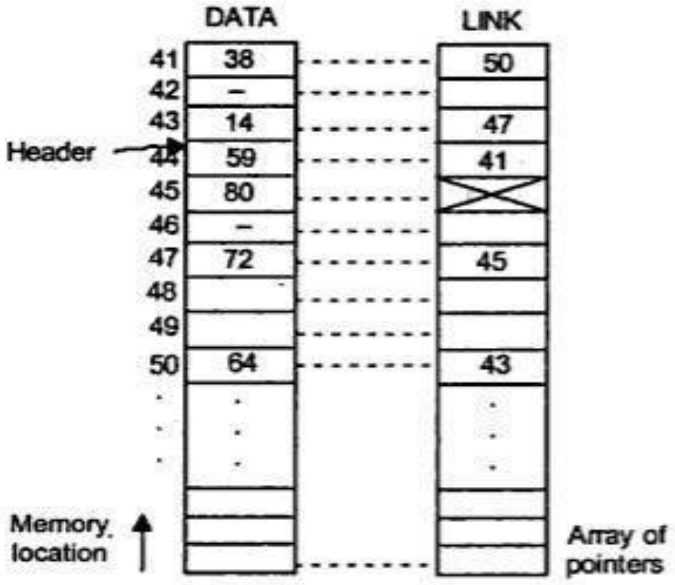


Fig. 3.2 A single linked list with 6 nodes.



2. Dynamic representation

- The efficient way of representing a linked list is using the **free pool of storage**.
- There is a
  - ***memory bank*** : Collection of free memory spaces &
  - ***memory manager*****:** a program
- Whenever a node is required, the request is placed to the memory manager.
- It will search the memory bank for the block. If found, it will be granted.
- ***Garbage collector*****:** Another program that returns the unused node to the memory bank.

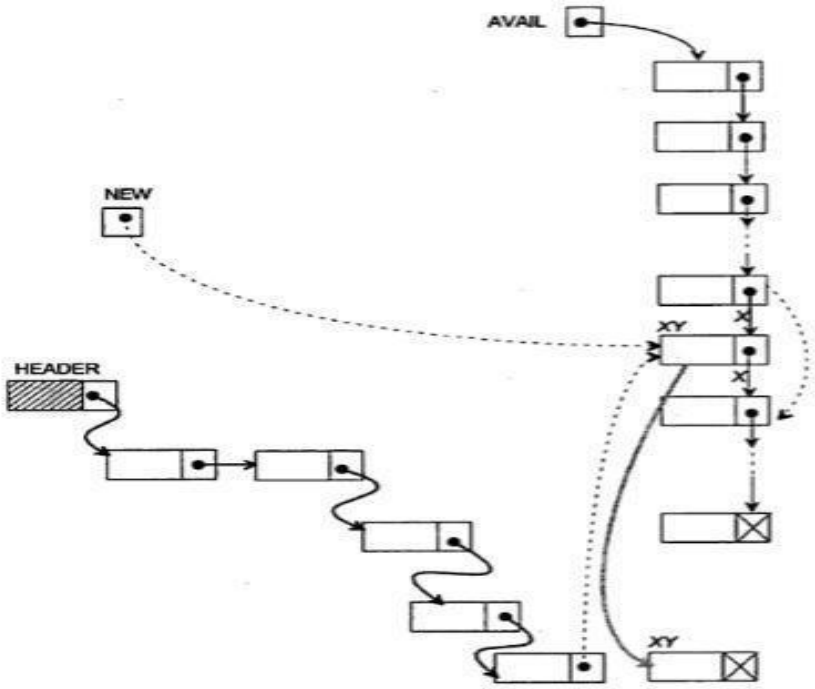
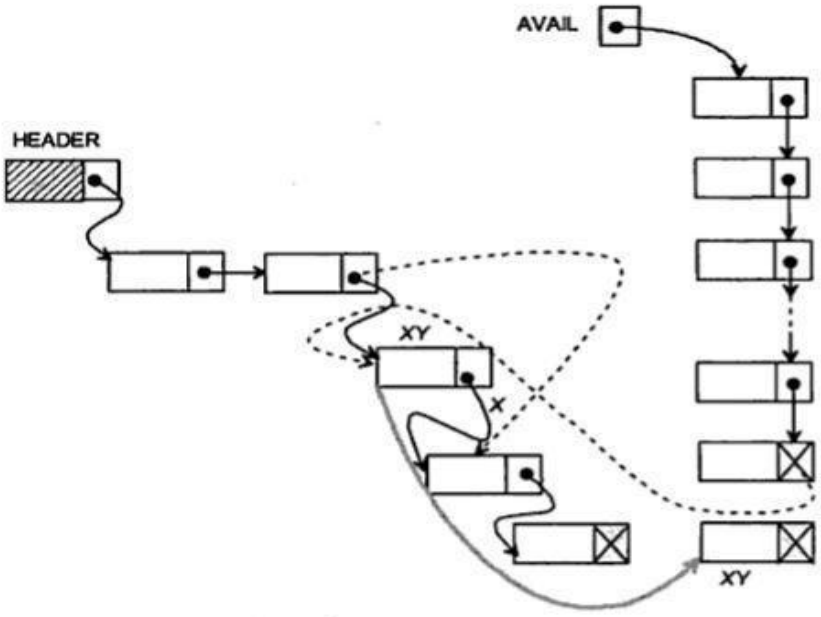


Fig. 3.4(a) Allocation of a node from memory bank to a linked list.

- Returning a node to memory bank



Operations on a Single Linked List

- Traversing the list
- Inserting a node into the list
- Deleting a node from the list

- Merging the list with another to make a larger list

Node creation

```
struct node
{
        int data;
        struct node *link;
};
```

- Function used for memory allocation is **"malloc"**

  New_node = (struct node*)malloc (sizeof (struct node))

1. Traversing a single linked list

- Here we visit every node in the list starting from the first node to the last one.

  Traverse()

  1. Set ptr=head; //initialize the pointer ptr
  2. While (ptr!=null) do
  3. print ptr->data
  4. ptr= ptr->next; //ptr now points to the next node

2. Inserting a node into the list

A. Inserting at the front (as a first element)

B. Inserting at the end(as a last element)

C. Inserting at any other position

A. Inserting at the front

-allocate the space for the new node and store data into the data part of the node

ptr=(struct node*) malloc(sizeof(struct node));

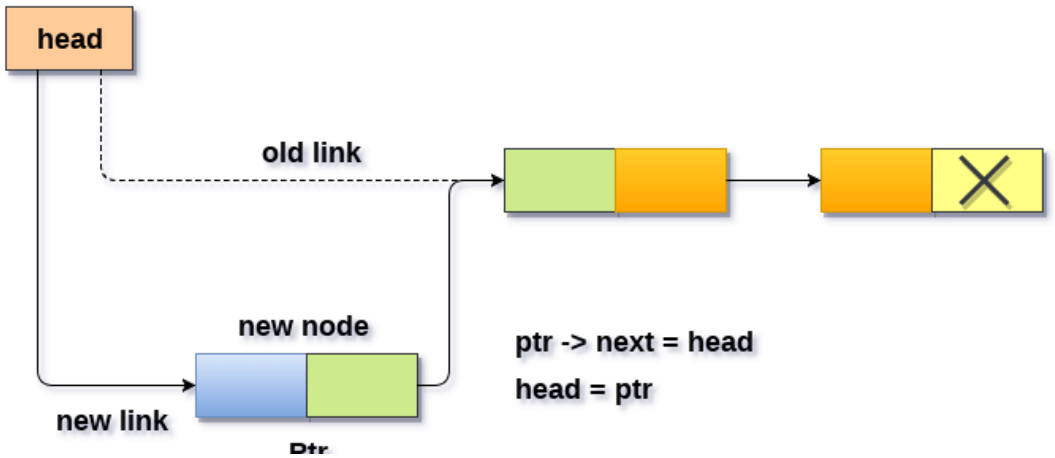
ptr->data= item

-make the link part of the new node pointing to the existing first node of the list

ptr-> next=head

-at the last, we need to make the new node as the first node of the list

head =ptr



Algorithm: Insert a new node ptr with data 'item'

1. Create a pointer ptr of type struct node
2. Create a new node ptr using malloc function
   ptr = (struct node*) malloc(sizeof(struct node));
3. if (ptr==NULL)
4. print "memory underflow, no insertion"
5. else
6. ptr->data= item
7. Set ptr-> next=head
8. head=ptr

B. Inserting at the end in order to insert at the last

- Here first we need to traverse the list to get the last node.

1. Create a pointer temp & ptr of type struct node
2. Create a new node temp using malloc function
   temp = (struct node*) malloc(sizeof(struct node));
3. Set ptr=head; //initialize the pointer ptr
4. While (ptr->link!=null) do
5. ptr= ptr->link; //ptr now points to the next node
6. ptr->link= temp
7. temp->data=item

C. Insertion- At any position in the list

1. Create a pointer temp & ptr of type struct node
2. Create a new node temp using malloc function
   temp = (struct node*) malloc(sizeof(struct node));
3. Read the value **key** of node after which a new node is to be placed
4. Set ptr=head
5. Repeat while (ptr-> data!=key) and (ptr->link!=NULL)
6. ptr=ptr-> link
7. If (ptr->link==NULL)
8. print “search fails”;
9. else
10. temp->link= ptr-> link
11. ptr->link= temp

3. Deleting a node from the list

- In a linked list, an element can be deleted:
  A. From the 1st location

B. From the last location

C. From any position in the list

free(ptr) : It will free the location pointed by ptr

A. Deletion- From the beginning

1. Create a pointer ptr of type struct node
2. If (head==NULL) then exit
3. Else set ptr = head
4. set head=ptr-> next
5. free(ptr)

B. Deletion- From the end

1. Create a pointer ptr & temp of type struct node.
2. If (head -> link ==NULL) do step 3,4,5 else goto 6
3. ptr=head
4. head=NULL
5. free(ptr)
6. ptr=head
7. temp = head -> link
8. while(tem -> link !=NULL) do 9,10 else goto 11
9. ptr=temp
10. temp= tem -> link
11. ptr-> link =NULL
12. free (temp)

C. Deletion- From any position

1. Read the value **key** that is to be deleted

2. Create pointer ptr & temp of type struct node
3. Set ptr=head
4. if head=NULL then print underflow and exit
5. temp=ptr
6. while(ptr!=null) do step 7,8,9
7. If(ptr->data=key) then
    a) temp->link=ptr->link
    b) free(ptr) & exit
8. temp=ptr
9. Ptr = ptr->link

4. Merging

- Two linked lists L1 and L2.
- Merge L2 after L1
    1. Set ptr= head1
    2. While(ptr->link!= NULL) do step 3 else goto step 4
    3. ptr=ptr->link
    4. ptr->link=head2
    5. Return(head2)
    6. Head=head1
    7. Stop

2. DOUBLE LINKED LIST

- Single linked list= one-way list
    - List can be traversed in one direction only
- Double linked list= Two-way list
    - List can be traversed in two directions

- two- way list is a linear collection of data elements called nodes where each node N is divided in to three parts
  - **Data field** contains the data of N
  - **LLINK field** contains the pointer to the preceding Node in the list
  - **RLINK field** contains the pointer to the next node in the list

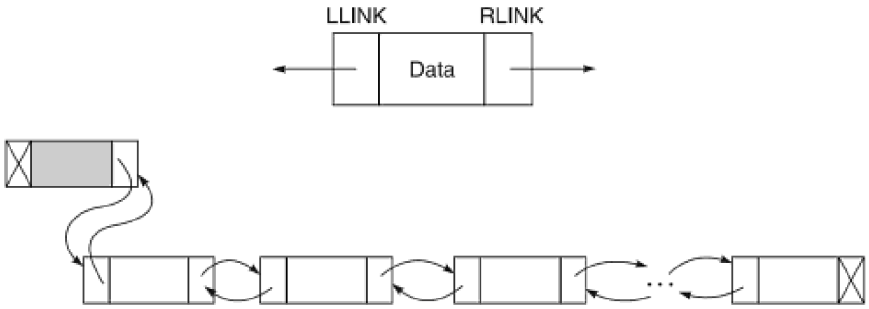


Figure 3.10 Structure of a node and a double linked list.

Operations on a Double Linked List

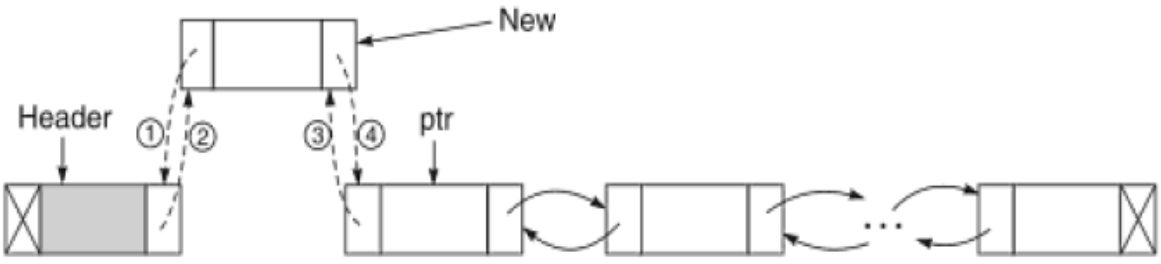
All the operations as mentioned for a single linked list can be implemented more efficiently using a double linked list.

Inserting a node into a Double Linked List (DLL)

Let us consider the algorithms of following cases of insertion in a DLL

i) Inserting a node in the front,
ii) Inserting a node at the end, and
iii) Inserting a node at any position in a double linked list.

i) Inserting a node in the front

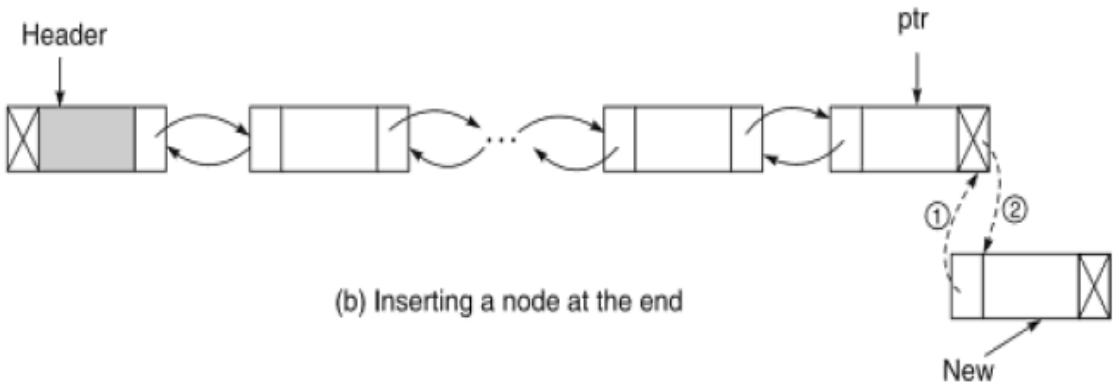


(a) Inserting a node in the front

Steps:

1. ptr = HEADER→RLINK // Points to the first node
2. new = **GetNode**(NODE) // Avail a new node from the memory bank
3. **If** (new ≠ NULL) then // If new node is available
4. new→LLINK = HEADER // Newly inserted node points the header as 1 in Figure 3.11(a)
5. HEADER→RLINK = new // Header now points to then new node as 2 in Figure 3.11(a)
6. new→RLINK = ptr // See the change in pointer shown as 3 in Figure 3.11(a)
7. ptr→LLINK = new // See the change in pointer shown as 4 in Figure 3.11(a)
8. new→DATA = *X* // Copy the data into the newly inserted node
9. **Else**
10. **Print** "Unable to allocate memory: Insertion is not possible"
11. **EndIf**
12. **Stop**

ii) Inserting a node at the end

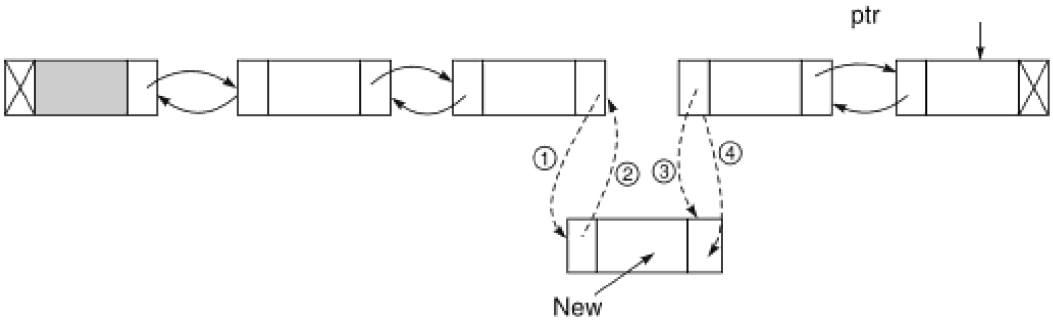


(b) Inserting a node at the end

Steps:

1. ptr = HEADER
2. **While** (ptr→RLINK ≠ NULL) **do** // Move to the last node
3. ptr = ptr→RLINK
4. **EndWhile**
5. new = **GetNode**(NODE) // Avail a new node
6. **If** (new ≠ NULL) **then** // If the node is available
7. new→LLINK = ptr // Change the pointer shown as 1 in Figure 3.11(b)
8. ptr→RLINK = new // Change the pointer shown as 2 in Figure 3.11(b)
9. new→RLINK = NULL // Make the new node as the last node
10. new→.DATA = *X* // Copy the data into the new node
11. **Else**
12. **Print** "Unable to allocate memory: Insertion is not possible"
13. **EndIf**
14. **Stop**

iii) Insertion- after an element key



(c) Inserting a node at any intermediate position

Figure 3.11 Inserting a node at various positions in a double linked list.

Steps:

1. ptr = HEADER
2. **While** (ptr→DATA ≠ KEY) and (ptr→RLINK ≠ NULL) // Move to the key node if the // current node is not the KEY node or if the list reaches the end
3. ptr = ptr→RLINK
4. **EndWhile**
5. new = GetNode(NODE) // Get a new node from the pool of free storage
6. **If** (new = NULL) **then** // When the memory is not available
7. **Print** (Memory is not available)
8. **Exit** // Quit the program
9. **EndIf**
10. **If** (ptr→RLINK = NULL) **then** // If the KEY is not found in the list
11. new→LLINK = ptr
12. ptr→RLINK = new // Insert at the end
13. new→RLINK = NULL
14. new→DATA = *X* // Copy the information to the newly inserted node
15. **Else** // The KEY is available
16. ptr1 = ptr→RLINK // Next node after the key node
17. new→LLINK = ptr // Change the pointer shown as 2 in Figure 3.11(c)
18. new→RLINK = ptr1 // Change the pointer shown as 4 in Figure 3.11(c)
19. ptr→RLINK = new // Change the pointer shown as 1 in Figure 3.11(c)
20. ptr1→LLINK = new // Change the pointer shown as 3 in Figure 3.11(c)
21. ptr = new // This becomes the current node
22. new→DATA = *X* // Copy the content to the newly inserted node
23. **EndIf**
24. **Stop**

Deleting a node from a Double Linked List (DLL)

Just like Insertion, Deleting a node from a Double Linked List consists of following cases:

i) Deleting a Node from the front of a DLL,

ii) Deleting a Node at the end of a DLL, and

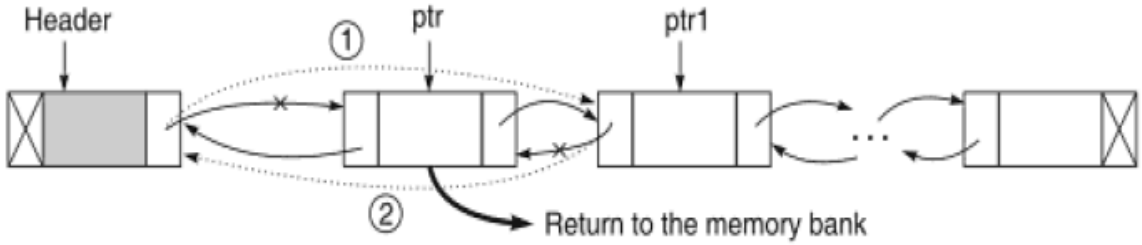
iii) Deleting a Node from any intermediate position.

i) Deletion- from 1st location

Steps:

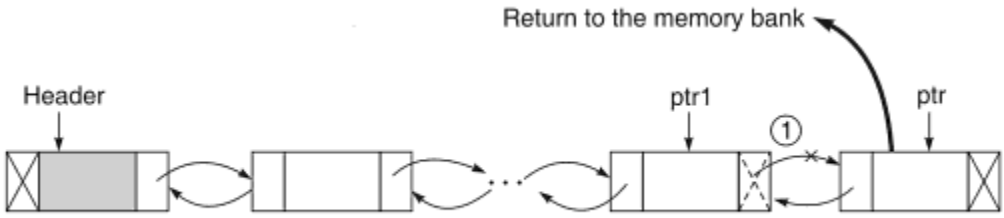
```
1.  ptr = HEADER→RLINK                                  // Pointer to the first node
2.  If (ptr = NULL) then                                // If the list is empty
3.      Print "List is empty: No deletion is made"
4.      Exit
5.  Else
6.      ptr1 = ptr→RLINK                                // Pointer to the second node
7.      HEADER→RLINK = ptr1          // Change the pointer shown as 1 in Figure 3.12(a)
8.      If (ptr1 ≠ NULL)             // If the list contains a node after the first node of deletion
9.          ptr1→LLINK = HEADER      // Change the pointer shown as 2 in Figure 3.12(a)
10.     EndIf
11.     ReturnNode (ptr)             // Return the deleted node to the memory bank
12. EndIf
13. Stop
```

Note that the algorithm *DeleteFront_DL* works even if the list is empty.



(a) Deleting the node placed at the front

ii) Deletion- from last location

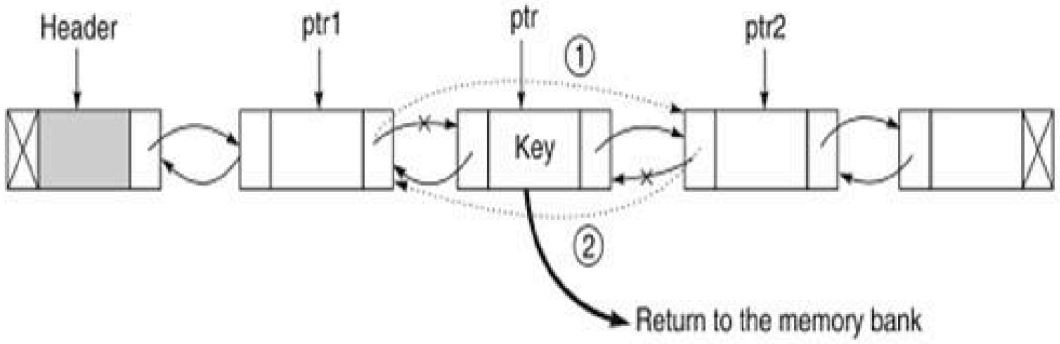


(b) Deleting the node placed at the end

Algorithm DeleteEnd_DL
Input: A double linked list with data.
Output: A reduced double linked list.
Data structure: Double linked list structure whose pointer to the header node is the *HEADER*.
Steps:

```
1.  ptr = HEADER
2.  While (ptr→RLINK ≠ NULL) do                    // Move to the last node
3.      ptr = ptr→RLINK
4.  EndWhile
5.  If (ptr = HEADER) then                         // If the list is empty
6.      Print "List is empty: No deletion is made"
7.      Exit                                       // Quit the program
8.  Else
9.      ptr1 = ptr→LLINK                           // Pointer to the last but one node
10.     ptr1→RLINK = NULL          // Change the pointer shown as 1 in Figure 3.12(b)
11.     ReturnNode (ptr)           // Return the deleted node to the memory bank
12. EndIf
13. Stop
```

iii) Deletion- from intermediate location



(c) Deleting a node from any intermediate position

Steps:

```
1.  ptr = HEADER→RLINK                                   // Move to the first node
2.  If (ptr = NULL) then
3.      Print "List is empty: No deletion is made"
4.      Exit
5.  EndIf                                                // Quit the program
6.  While (ptr→DATA ≠ KEY) and (ptr→RLINK ≠ NULL) do     // Move to the desired node
7.      ptr = ptr→RLINK
8.  EndWhile
9.  If (ptr→DATA = KEY) then                             // If the node is found
10.     ptr1 = ptr→LLINK                                 // Track the predecessor node
11.     ptr2 = ptr→RLINK                                 // Track the successor node
12.     ptr1→RLINK = ptr2          // Change the pointer shown as 1 in Figure 3.12(c)
13.     If (ptr2 ≠ NULL) then                            // If the deleted node is the last node
14.         ptr2→LLINK = ptr1      // Change the pointer shown as 2 in Figure 3.12(c)
15.     EndIf
16.     ReturnNode(ptr)                  // Return the free node to the memory bank
17. Else
18.     Print "The node does not exist in the given list"
19. EndIf
20. Stop
```

3. CIRCULAR LINKED LIST

- In a single linked list, the link field of the last node is null.
- If we utilize this link field to store the pointer of the header node, a number of advantages can be gained.
- A linked list, whose last node points back to the first node, instead of containing the null pointer is called a **circular list**

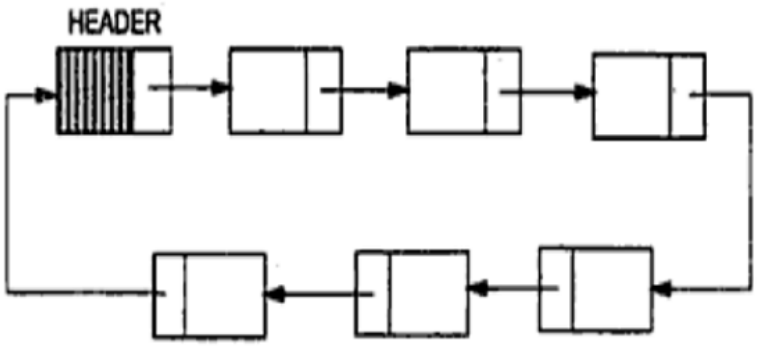


Fig. 3.8 A circular linked list.

- **Advantages:**

  1. **Accessibility of a member node** – here every member node is accessible from any node by merely chaining through the list

  eg: Finding of earlier occurrence or post occurrence of a data will be easy

  2. **Null link problem-** Null value in next field may create problem during the execution of the program if proper care is not taken

  3. Some **easy-to-implement operations** - Operations like merging, splitting, deletion, dispose of an entire list etc can be done easily with circular list

- **Disadvantages:**

  - If not cared, system may get trap into in infinite loop
    - It occurs when we are unable to detect the end of the list while moving from one node to the next
    - Solution: Special node can be maintained with data part as NULL and this node does not contain any valid information. So its just a wastage of memory space

Insertion in circular linklist

- We want to insert data 'X' after a given position, 'pos'
- Here we are using a pointer called last, which points to the last node

1. Create a pointer temp and q of type struct node
2. Set q=last->link and i=1
3. While(i<pos) do step 4
4. q=q->link & increment I
5. Create a new node temp usin malloc function

   temp = (struct node*) malloc(sizeof(struct node));
6. temp->link=q->link
7. temp->data = X
8. q->link = temp

Deletion in circular Linked List

1. if last = NULL print under flow and exit

// Linkedlist containing only one node

2. If last -> link = last & last -> data = key then do the steps 3,4,5
3. temp= last
4. Last = NULL
5. free(temp)
6. q = last ->link

//Deleting first node

7. if q->data =key do 8,9,10
8. temp = q
9. Last->link= q->link
10. free(temp)

// deleting Middle node

11. Repeat steps 12 to 16 while q->link!=last
12. if q->link->data =key do step 13,14,15
13. temp = q->link
14. q->link= tem->link
15. Free(temp)

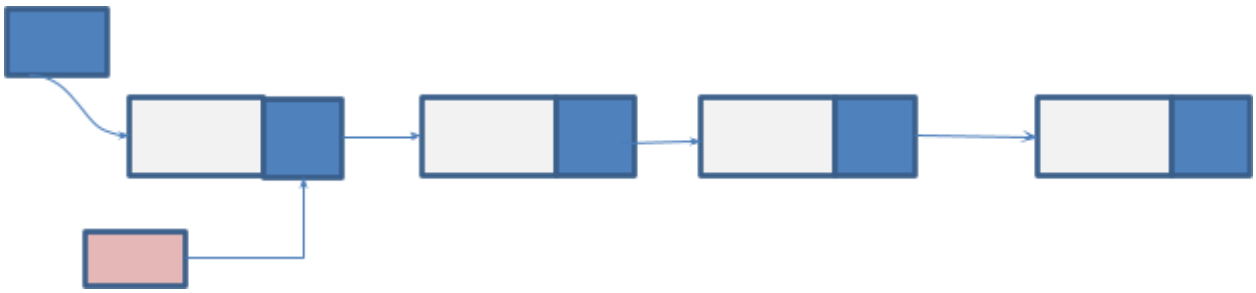
//Deleting last node

16. If q->link ->data = key
17. temp = q->link
18. q->link=last->link
19. Free(temp)
20. Last=q

4. STACKS USING LINKED LIST

- Stack can also be represented using a singly linked list.
- Linked lists have many advantages compared to arrays.
- In the linked list, the **DATA** field contains the elements of stack and **LINK** field points to the next element in the stack.

- Here **Push** operation is accomplished by **inserting a new node in the front** or start of the list.
- **Pop** is done by **removing the element from the front** of the list



Insertion- At the beginning

Algorithm: PUSH()

1. Create a new node temp //struct node *temp = (struct node*) malloc(sizeof(struct node));
2. If (temp==NULL)
3. print "memory underflow, no insertion"
4. else
5. temp->data= item
6. temp-> link=head
7. head=temp

Deletion- From the beginning

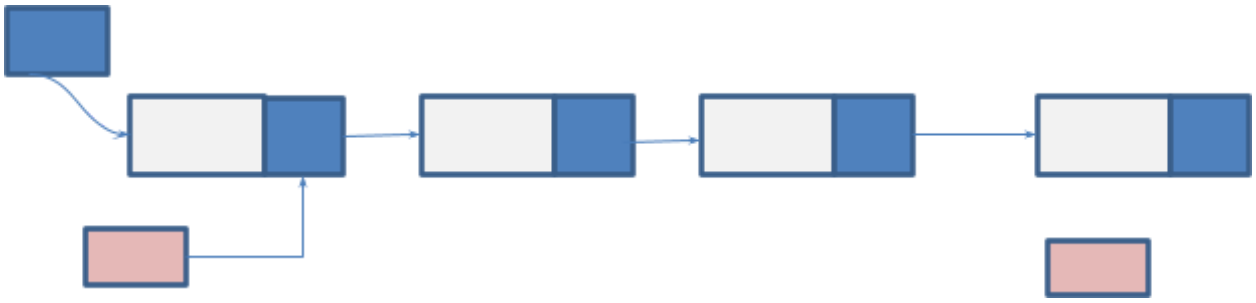
Algorithm: POP()

1. Start
2. If(head==null)
3. print "underflow"
4. Else
5. print the deleted element 'head-> data'
6. head= head->link

5. QUEUES USING LINKED LIST

- Queue can also be represented using a singly linked list.

- Linked lists have many advantages compared to arrays.
- In linked list, the Data field contains the elements of queue and Next pointer points to the next element in the queue.
- Here **enqueue** operation is accomplished by **inserting a new node in the tail** or end of the list.
- **Dequeue** is done by **removing the element from the beginning** of the list



Insertion- At the end

Algorithm: Enqueue()

1. Set ptr=head; //initialize the pointer ptr
2. While (ptr->link!=null) do
3. ptr= ptr->link; //ptr now points to the next node
4. ptr->link= temp
5. temp->data=item

Deletion – At the front

Algorithm: DEQUEUE()

1. Start
2. If(head==null)
3. print “underflow”
4. Else
5. print the deleted element ‘head-> data’
6. head= head-> link

POLYNOMIAL REPRESENTATION USING LINKED LIST

Polynomial having a single variable

Let us consider the general form of a polynomial having a single variable:

$$P(x) = a_n x^{e_n} + a_{n-1} x^{e_{n-1}} + \cdots + a_1 x^{e_1}$$

where $a_i x^{e_i}$ is a term in the polynomial so that a_i is a non-zero coefficient and e_i is the exponent. We will assume an ordering of the terms in the polynomial such that $e_n > e_{n-1} > \ldots > e_2 > e_1 \geq 0$. The structure of a node in order to represent a term can be decided as shown below:

Considering the single linked list representation, a node should have three fields: COEFF (to store the coefficient a_i), EXP (to store the exponent e_i) and a LINK (to store the pointer to the next node representing the next term). It is evident that the number of nodes required to represent a polynomial is the same as the number of terms in the polynomial. An additional node may be considered for a header. As an example, let us consider that the single linked list representation of the polynomial $P(x) = 3x^8 - 7x^6 + 14x^3 + 10x - 5$ would be stored as shown in Figure 3.18.

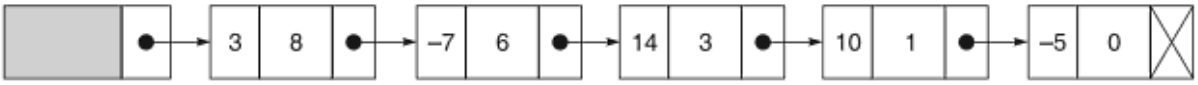


Figure 3.18 Linked list representation of a polynomial (single variable).

Note that the terms whose coefficients are zero are not stored here. Next let us consider two basic operations, namely the addition and multiplication of two polynomials using this representation.

Polynomial addition

In order to add two polynomials, say P and Q, to get a resultant polynomial R, we have to compare their terms starting at their first nodes and moving towards the end one by one. Two pointers Pptr and Qptr are used to move along the terms of P and Q. There may arise three cases during the comparison between the terms of two polynomials.

(i) **Case 1:** The exponents of two terms are equal. In this case the coefficients in the two nodes are added and a new term is created with the values

$$\text{Rptr}\rightarrow\text{COEFF} = \text{Pptr}\rightarrow\text{COEFF} + \text{Qptr}\rightarrow\text{COEFF}$$

and

$$\text{Rptr}\rightarrow\text{EXP} = \text{Pptr}\rightarrow\text{EXP}$$

(ii) **Case 2:** Pptr→EXP > Qptr→EXP, i.e. the exponent of the current term in P is greater than the exponent of the current term in Q. Then, a duplicate of the current term in P is created and inserted in the polynomial R.

(iii) **Case 3:** Pptr→EXP < Qptr→.EXP, i.e. the case when the exponent of the current term in P is less than the exponent of the current term in Q. In this case, a duplicate of the current term of Q is created and inserted in the polynomial R. The algorithm *PolynomialAdd_LL* is described as below:

Algorithm PolynomialAdd_LL

Input: Two polynomials P and Q whose header pointers are *PHEADER* and *QHEADER*.
Output: A polynomial R is the sum of P and Q having the header *RHEADER*.
Data structure: Single linked list structure for representing a term in a single variable polynomial.

Steps:

```
1.  Pptr = PHEADER→LINK, Qptr = QHEADER→LINK
    //Get a header node for the resultant polynomial//
2.  RHEADER = GetNode(NODE)
3.  RHEADER→LINK = NULL, RHEADER→EXP = NULL, RHEADER→COEFF = NULL
4.  Rptr = RHEADER                          // Current pointer to the resultant polynomial R
5.  While (Pptr ≠ NULL) and (Qptr ≠ NULL) do
6.      CASE: Pptr→EXP = Qptr→EXP                                   // Case 1
7.          new = GetNode (NODE)
8.          Rptr→LINK = new, Rptr = new
9.          Rptr→COEFF = Pptr→COEFF + Qptr→COEFF
10.         Rptr→EXP = Pptr→EXP
11.         Rptr→LINK = NULL
12.         Pptr = Pptr→LINK, Qptr = Qptr→LINK
13.     CASE: Pptr→EXP > Qptr→EXP                                   // Case 2
14.         new = GetNode (NODE)
15.         Rptr→LINK = new, Rptr = new

16.         Rptr→COEFF = Pptr→COEFF
17.         Rptr→EXP = Pptr→EXP
18.         Rptr→LINK = NULL
19.         Pptr = Pptr→LINK
20.     CASE: Pptr→EXP < Qptr→EXP                                   // Case 3
21.         new = GetNode (NODE)
22.         Rptr→LINK = new, Rptr = new
23.         Rptr→COEFF = Qptr→COEFF
24.         Rptr→EXP = Qptr→EXP
25.         Rptr→LINK = NULL
26.         Qptr = Qptr→LINK
27. EndWhile
```

```
28.     If (Pptr ≠ NULL) and (Qptr = NULL) then        // To add the trailing part of P, if any
29.     While (Pptr ≠ NULL) do
30.         new = GetNode(NODE
31.         Rptr→LINK = new, Rptr = new
32.         Rptr→COEFF = Pptr→COEFF
33.         Rptr→EXP = Pptr→Exp
34.         Rptr→LINK = NULL
35.         Pptr = Pptr→LINK
36.     EndWhile
37. EndIf
38. If (Pptr = NULL) and (Qptr ≠ NULL) then        //To add the trailing part of Q, if any
39.     While (Qptr ≠ NULL) do
40.         new = GetNode(NODE)
41.         Rptr→LINK = new, Rptr = new
42.         Rptr→COEFF = Qptr→COEFF
43.         Rptr→EXP = Qptr→EXP
44.         Rptr→LINK = NULL
45.         Qptr = Qptr→LINK
46.     EndWhile
47. EndIf
48. Return(RHEADER)
49. Stop
```

Polynomial multiplication

Suppose, we have to multiply two polynomials P and Q so that the result will be stored in R, another polynomial. The method is quite straightforward: let Pptr denote the current term in P and Qptr be that of in Q. For each term of P we have to visit all the terms in Q; the exponent values in two terms are added (R→EXP = P→EXP + Q→EXP), the coefficient values are multiplied (R→COEFF = P→COEFF × Q→COEFF), and these values are included into R in

such a way that if there is no term in R whose exponent value is the same as the exponent value obtained by adding the exponents from P and Q, then create a new node and insert it to *R* with the values so obtained (that is, R→COEFF, and R→EXP); on the other hand, if a node is found in *R* having same exponent value R→EXP, then update the coefficient value of it by adding the resultant coefficient (R→COEFF) into it. The algorithm *PolynomialMultiply_LL* is described as below:

Algorithm PolynomialMultiply_LL
Input: Two polynomials P and Q having their headers as *PHEADER*, *QHEADER*.
Output: A polynomial R storing the result of multiplication of P and Q.
Data structure: Single linked list structure for representing a term in a single variable polynomial.

Steps:

1. Pptr = PHEADER, Qptr = QHEADER
   / Get a node for the header of R */*
2. RHEADER = **GetNode**(NODE)
3. RHEADER→LINK = NULL, RHEADER→COEFF = NULL, RHEADER→EXP = NULL
4. **If** (Pptr→LINK = NULL) or (Qptr→LINK = NULL) **then**
5. Exit // No valid operation possible
6. **EndIf**
7. Pptr = Pptr→.LINK
8. **While** (Pptr ≠ NULL) **do** // For each term of P
9. **While** (Qptr ≠ NULL) **do**
10. C = Pptr→COEFF × Qptr→COEFF
11. X = Ppt→EXP + Qptr→EXP

    / Search for the equal exponent value in R */*
12. Rptr = RHEADER
13. **While** (Rptr ≠ NULL) and (Rptr→EXP > X) **do**
14. Rptr1 = Rptr
15. Rptr = Rptr→LINK
16. **If** (Rptr→EXP = X) **then**
17. Rpt→COEFF = Rptr→COEFF + C
18. **Else** // Add a new node at the correct position in R
19. new = **GetNode**(NODE)
20. new→EXP = X, new→COEFF = C
21. **If** (Rptr→LINK = NULL) **then**
22. Rptr→LINK = new // Append the node at the end
23. new→LINK = NULL
24. **Else**
25. Rptr1→LINK = new // Insert the node in ordered position
26. new→LINK = Rptr
27. **EndIf**
28. **EndIf**
29. **EndWhile**
30. **EndWhile**
31. **EndWhile**
32. **Return** (RHEADER)
33. **Stop**

Memory Management

The basic task of any program is to manipulate data. These data should be stored in memory during their manipulation. There are two memory management schemes for the storage allocations of data:

1. Static storage management
2. Dynamic storage management

Static Storage Management

In the case of the *static storage management* scheme, the net amount of memory required for various data for a program is allocated before the start of the execution of the program. Once memory is allocated, it can neither be extended nor be returned to the memory bank for the use of other programs at the same time.

Dynamic Storage Management

On the other hand, the *dynamic storage management* scheme allows the user to allocate and deallocate as per the requirement during the execution of programs. This dynamic memory management scheme is suitable in multiprogramming as well as in single-user environment where generally more than one program reside in the memory and their memory requirement can be known only during their execution. An operating system (OS) generally provides the service of dynamic memory management. The data structure for implementing such a scheme is a linked list.

There are various principles on which the dynamic memory management scheme is based. These principles are listed below.

1. *Allocation schemes:* Here, we discuss how a request for a memory block will be serviced. There are two strategies:
   (a) Fixed block allocation
   (b) Variable block allocation. There are four strategies under this:
      (i) First-fit and its variant
      (ii) Next-fit
      (iii) Best-fit
      (iv) Worst-fit
2. *Deallocation schemes:* Here, we discuss how to return a memory block to the memory bank whenever it is no longer required. Two strategies are known for the deallocation schemes:
   (i) Random deallocation
   (ii) Ordered deallocation

We will discuss two more systems for the implementation of allocation and deallocation schemes:

1. Boundary tag system
2. Buddy system

There is, again, one more principle called *garbage collection* to maintain a memory bank so that it can be utilized efficiently.

Now we discuss below the dynamic memory management schemes and possible use of a linked list therein.

Memory Representation

A memory bank or a pool of free storage is often a collection of non-contiguous blocks of memory. Their linearity can be maintained by means of pointers between one block to another, or in other words a memory bank is a linked list where links maintain the adjacency of blocks. Regarding the size of the blocks, there are two practices: fixed block storage and variable block storage.

Fixed Block Storage

This is the simplest storage maintenance method. Here each block is of the same size. The size is determined by the system manager (user). Here, the memory manager (a program of OS) maintains a pointer AVAIL which points a list of non-contiguous memory blocks. The below figure shows a memory bank with fixed size blocks.

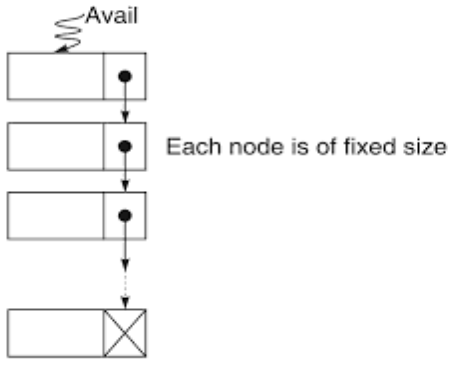


Figure Pool of free storages with fixed size blocks.

A user program communicates with the memory manager by means of two functions *GetNode*( ) and *ReturnNode*( ), which are discussed below.

Procedure GetNode

Input: This procedure avails a block from memory bank for a data whose type is represented by *NODE*.

Output: Returns a pointer to the memory block if available else a message.

Steps:

```
1. If (AVAIL = NULL) then                          // Memory is exhausted
       Print "Memory is insufficient"
2. Else
3.     ptr = AVAIL
4.     AVAIL = AVAIL→LINK
5.     Return(ptr)          // Return pointer of the available block to the caller
6. EndIf
7. Stop
```

The procedure *GetNode* is to get a memory block to store data of type *NODE* (by passing this as an argument we need to mention the size of the memory block required). This procedure when invoked by a program, returns a pointer to the first block in the pool of free storage. The AVAIL then points to the next block. The link modification is shown (by the dotted line) in Figure 3.20. If AVAIL = NULL, it indicates that no more memory is available for allocation.

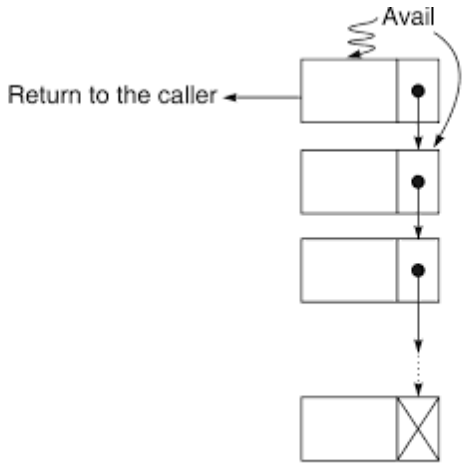


Figure 3.20 Getting a block from the memory bank.

Similarly, whenever a memory block is no more required, it can be returned to the memory bank through a procedure *ReturnNode* which is stated below:

Procedure ReturnNode
Input: This procedure returns a block of memory referenced by the pointer *PTR*.
Output: The memory block is returned to the memory bank.
Remark: Naïve approach, that is, get it as you find it.

Steps:

```
1. ptr1 = AVAIL
2. While (ptr1→LINK ≠ NULL) do          // Move to the end of the list
3.     ptr1 = ptr1→LINK
4. EndWhile
5. ptr1→LINK = PTR
6. PTR→LINK = NULL
7. Stop
```

The procedure *ReturnNode* appends a returned block (bearing pointer *PTR*) at the end of the pool of free storage pointed by AVAIL. Change in pointers can be seen in Figure 3.21 as a dotted line.

So far as the implementation of fixed block allocation is concerned, this is the most simple strategy. But the main drawback of this strategy is the wastage of space. For example, suppose each memory block is of size 1K (1024 bytes); now for a request of a memory block, say, of size 1.1K we have to avail 2 blocks (that is 2K memory space), thus wasting 0.9K memory space. Making the size of the block too small reduces the wastage of space; however, it also reduces the overall performance of the scheme.

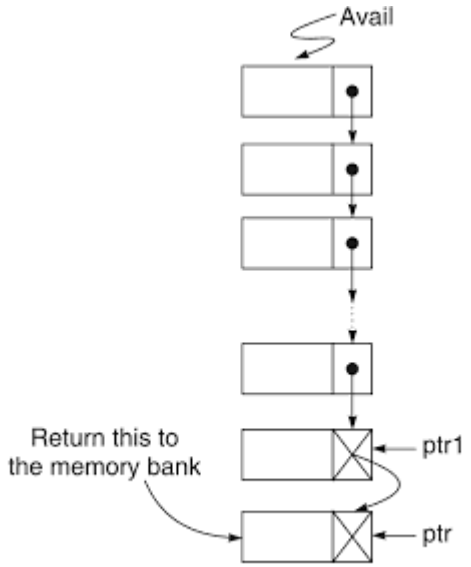


Figure 3.21 Returning a block to the memory bank.

Variable Block Storage

To overcome the disadvantages of fixed block storage, we can maintain blocks of variable sizes instead of those of fixed sizes. Procedures for *GetNode* and *ReturnNode* with this storage management are stated below.

Procedure GetNode
Input: This procedure gets a block of memory from the memory bank.
Output: Returns a pointer to the memory block if available else a message.
Remark: With variable sized memory blocks policy.

Steps:

```
1.  If (AVAIL = NULL) then
2.      Print "Memory bank is insufficient"
3.      Exit                                              // Quit the program
4.  EndIf
5.  ptr = AVAIL
6.  While (ptr→LINK ≠ NULL) and (ptr→SIZE < SizeOf(NODE)) do
                                                          // Move to the right block
7.      ptr1 = ptr
8.      ptr = ptr→LINK
9.  EndWhile
10. If (ptr→.LINK = NULL) and (ptr→SIZE < SizeOf(NODE)) then
11.     Print "Memory request is too large: Unable to serve"
12. Else
13.     ptr1→LINK = ptr→LINK
14.     Return(ptr)
15. EndIf
16. Stop
```

This procedure assumes that blocks of memory are stored in ascending order of their sizes. The node structure maintains a field to store the size of a block, namely SIZE. *SizeOf*( ) is a procedure that will return the size of a node (see Figure 3.22). Note that the above procedure will return a block of exactly the same size or more than the size that a user program requests. Next, let us describe the procedure *ReturnNode* to dispose a block into the pool of free storage in the ascending order of block sizes.

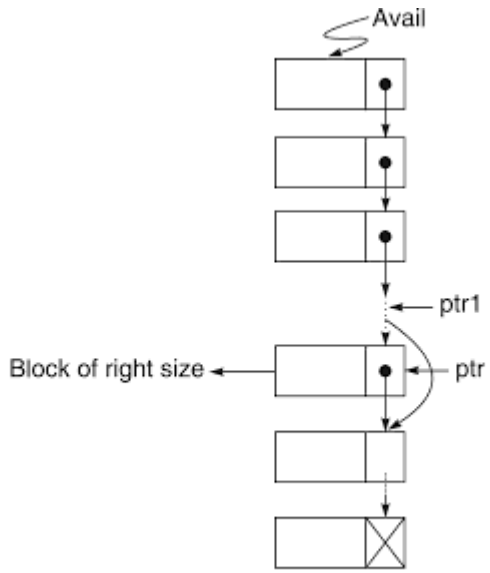


Figure 3.22 Availing a node from a pool of free storages with variable sized blocks.

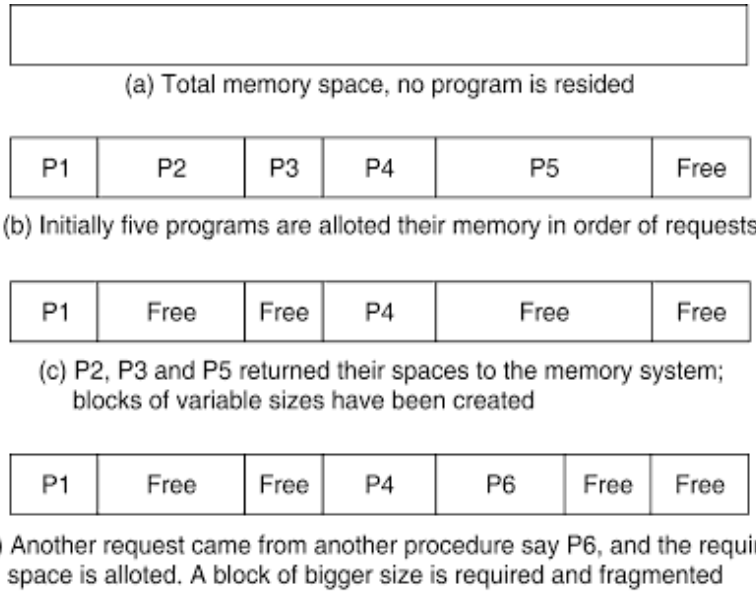
Procedure ReturnNode

Input: This procedure returns a block of memory referenced by the pointer *PTR*.
Output: The memory block is returned to the memory bank.
Remark: With variable sized memory blocks policy.

Steps:

```
1.  ptr1 = AVAIL
2.  While (ptr1→SIZE < PTR→SIZE) and (ptr1→LINK ≠ NULL)) do
                                                    // Move to the right position
3.      ptr2 = ptr1
4.      ptr1 = ptr1→LINK
5.  EndWhile
6.  If (PTR→SIZE < ptr1→SIZE) then
7.      ptr2→LINK = PTR
8.      PTR→LINK = ptr1
9.  Else
10.     ptr1→LINK = PTR
11.     PTR→LINK = NULL
12. EndIf
13. Stop
```

So far as memory space utilization is concerned, the variable sized block storage policy is preferred to that of the fixed sized block storages. During the discussion on procedure *GetNode* and *ReturnNode*, we have assumed that memory blocks of various sizes are available and they are linked with each other. But the actual case is different. Note that initially when there is no program in the memory, the entire memory is a block. The size of the blocks is then automatically generated, through the use of memory system, with several requests of various sizes and their returns. This is explained in Figure 3.23. We assume that the memory system starts with five programs: P1, P2, P3, P4 and P5.



(a) Total memory space, no program is resided

(b) Initially five programs are alloted their memory in order of requests

(c) P2, P3 and P5 returned their spaces to the memory system; blocks of variable sizes have been created

(d) Another request came from another procedure say P6, and the required space is alloted. A block of bigger size is required and fragmented

Figure 3.23 Partition of the memory into smaller blocks during dynamic storage allocation.

From the foregoing discussions, it can be concluded that the dynamic memory management system should provide the following services:

(a) Searching the memory for a block of requested size and servicing the request (allocation)

(b) Handling a free block when it is returned to the memory manager.

(c) Coalescing the smaller free blocks into larger block(s) (garbage collection and compaction).

To serve these facilities, we have two memory management systems: boundary tag system and buddy system.

Storage Allocation Strategies

In order to service a request for a memory block of given size, any one of the following well- known strategies can be used.

(a) **First-Fit** allocation

(b) **Best-Fit** allocation

(c) **Worst-Fit** Allocation

(d) **Next-Fit** Allocation

Let us discuss all these allocation strategies assuming that the memory system has to serve a request for a block of size.

First-fit storage allocation

This is the simplest of all the storage allocation strategies. Here the list of available storages is searched and as soon as a free storage block of size $\geq N$ is found the pointer of that block is sent to the calling program after retaining the residue space. Thus, for example, for a block of size 2K, if the first-fit strategy finds a block of 7K, then after retaining a storage block of size 5K, 2K memory will be sent to the caller.

Best-fit storage allocation

This strategy will not allocate a block of size $> N$, as it is found in the first-fit method; instead it will continue searching to find a suitable block so that the block size is closer to the block size of request. For example, for a request of 2K, if the list contains the blocks of sizes, 1K, 3K, 7K, 2.5K, 5K, then it will find the block of size 2.5K as the suitable block for allocation. From this block after retaining 0.5K, pointer for 2K block will be returned.

Worst-fit storage allocation

The best-fit strategy finds a block which is small and nearest to the block size as requested, whereas the worst-fit strategy is a reverse of the best-fit strategy. It allocates the largest block available in the available storage list. The idea behind the worst-fit is to reduce the rate of production of small blocks which are quite common when the best-fit strategy is used for memory allocation.

GARBAGE COLLECTION AND COMPACTION

Garbage collection and memory compaction are essential techniques in automatic memory management, ensuring efficient memory utilization and preventing issues like memory leaks and fragmentation.Garbage collection and compaction are crucial data structure concepts in memory management, particularly in languages that automate memory allocation like Java and Python. Garbage collection identifies and removes unused memory, while compaction rearranges remaining objects to create a contiguous block of free memory.

Garbage Collection:

- Garbage collection is the process of automatically reclaiming memory that is no longer being used by the program.
- It prevents memory leaks, where unused memory is not released, leading to performance issues and crashes.
- Garbage collectors identify and mark objects that are no longer reachable from the program's root objects (objects that are directly accessible to the program). Once marked, these objects can be reclaimed by the garbage collector.
- **Examples:**
  Common garbage collection algorithms include mark-and-sweep, copying collection, and generational garbage collection.
- **Mark-and-Sweep**: This algorithm operates in two phases: the "mark" phase identifies all reachable objects, and the "sweep" phase deallocates memory occupied by unreachable objects. While effective, it can lead to memory fragmentation since it doesn't relocate objects in memory.
- **Copying Collection**: This technique divides the heap into two halves. Live objects are copied from one half to the other, leaving behind a contiguous block of free memory, which helps in reducing fragmentation.
- **Generational GC**: Based on the observation that most objects die young, this approach segregates objects by their lifespan into generations. Young generations are collected more frequently, improving efficiency.

Compaction:

- Compaction is the process of rearranging the live objects in memory so they occupy a contiguous block of memory.
- To reduce external fragmentation, which occurs when free memory is scattered throughout the heap, making it difficult to allocate large blocks of memory.
- During garbage collection, live objects are moved to one end of the heap, and unused memory is freed at the other end.
- Compaction is often used in conjunction with garbage collection algorithms like mark-and-compact.
- **Mark–Compact Algorithm**: Combines marking of live objects with a compaction phase that moves these objects to one end of the memory, updating references accordingly. This reduces fragmentation and improves allocation efficiency.

MODULE 3

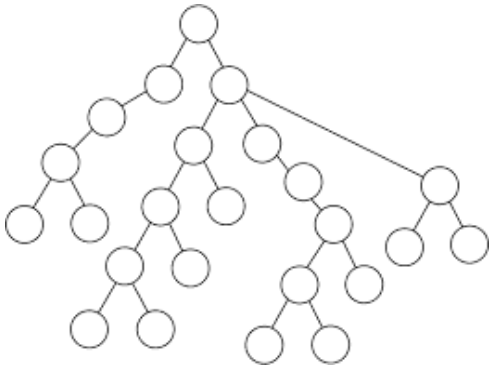
Trees and Graphs

Trees :- Representation Of Trees; Binary Trees - Types and Properties, Binary Tree Representation, Tree Operations, Tree Traversals; Expression Trees; Binary Search Trees - Binary Search Tree Operations; Binary Heaps - Binary Heap Operations, Priority Queue. **Graphs** :- Definitions; Representation of Graphs; Depth First Search and Breadth First Search; Applications of Graphs - Single Source All Destination.

TREES

- Arrays, linked lists, stacks and queues were examples of **linear data structures** in which elements are arranged in a **linear fashion** (ie, one dimensional representation).
- Tree is another very useful data structure in which elements are appearing in a **non-linear fashion**, which requires a two dimensional representation.

Example Figure:



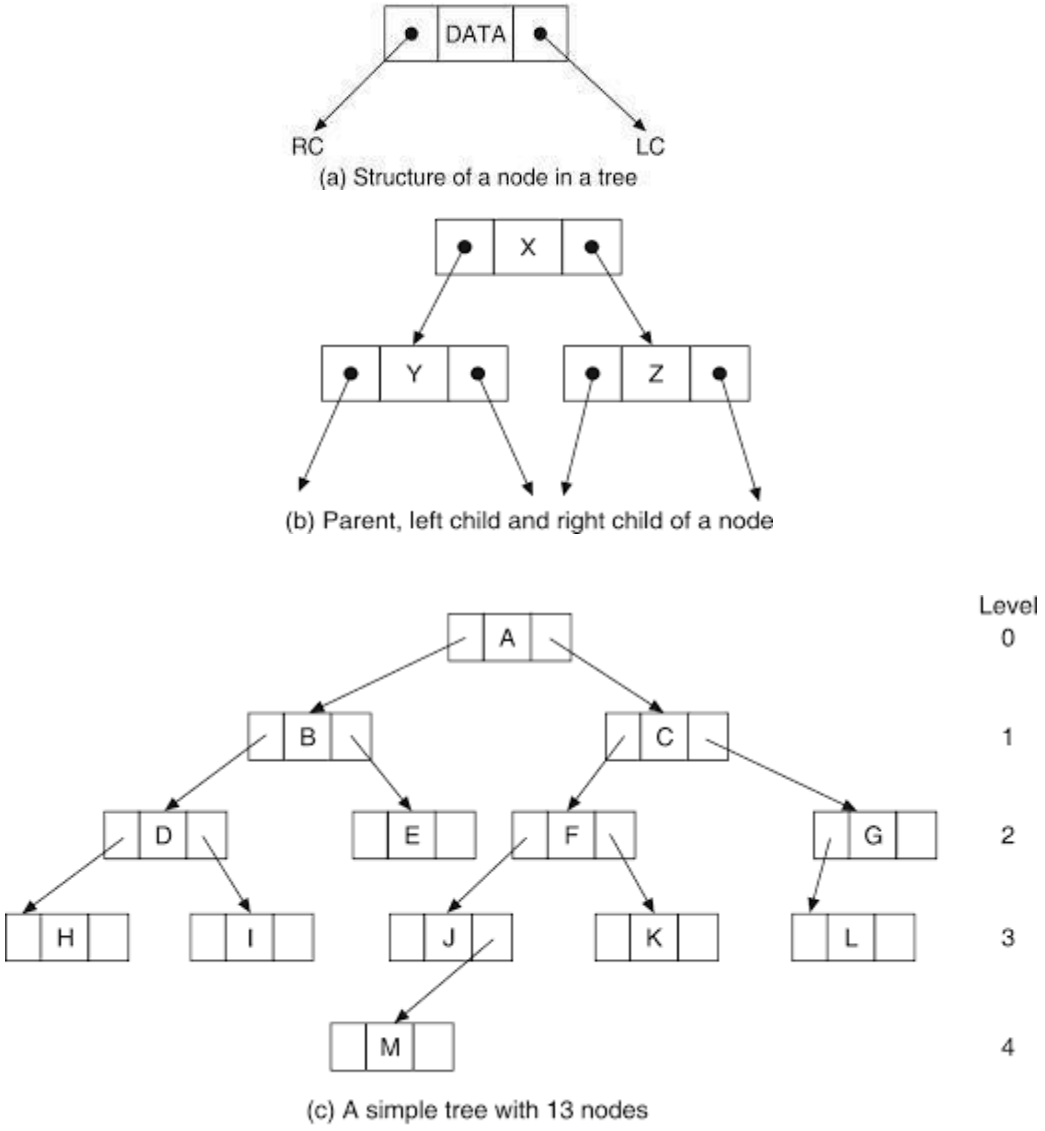
Basic Terminologies

Node: This is the main component of any tree structure. The concept of the node is the same as that used in a linked list. A node of a tree stores the actual data and links to the other node.

Figure 7.4(a) represents the structure of a node.

Parent: The parent of a node is the immediate predecessor of a node. Here, X is the parent of Y and Z. See Figure 7.4(b).

Child: If the immediate predecessor of a node is the parent of the node then all immediate successors of a node are known as child. For example, in Figure 7.4(b), Y and Z are the two child of X. The child which is on the left side is called the left child and that on the right side is called the right child.



(a) Structure of a node in a tree

(b) Parent, left child and right child of a node

(c) A simple tree with 13 nodes

Figure 7.4 A tree and its various components.

Link: This is a pointer to a node in a tree. For example, as shown in Figure 7.4(a). LC and RC are two links of a node. Note that there may be more than two links of a node.

Root: This is a specially designated node which has no parent. In Figure 7.4(c). A is the root node.

Leaf: The node which is at the end and does not have any child is called leaf node. In Figure 7.4(c), H, I. K. L. and M are the leaf nodes. A leaf node is also alternatively termed a terminal node.

Level: Level is the rank in the hierarchy. The root node has level 0. If a node is at level l, then its child is at level l + 1 and the parent is at level l - 1. This is true for all nodes except the root node, being at level zero. In Figure 7.4(c), the levels of various nodes are depicted.

Height: The maximum number of nodes that is possible in a path starting from the root node to a leaf node is called the height of a tree. For example, in Figure 7.4(c), the longest path is

A-C-F-J-M and hence the height of this tree is 5. It can be easily seen that h=l_{max} + 1.

where h is the height and l_{max} is the maximum level of the tree.

Degree: The maximum number of children that is possible for a node is known as the degree of a node. For example, the degree of each node of the tree as shown in Figure 7.4(c) is 2.

Sibling: The nodes which have the same parent are called siblings. For example, in Figure 7.4(c). J and K are siblings.

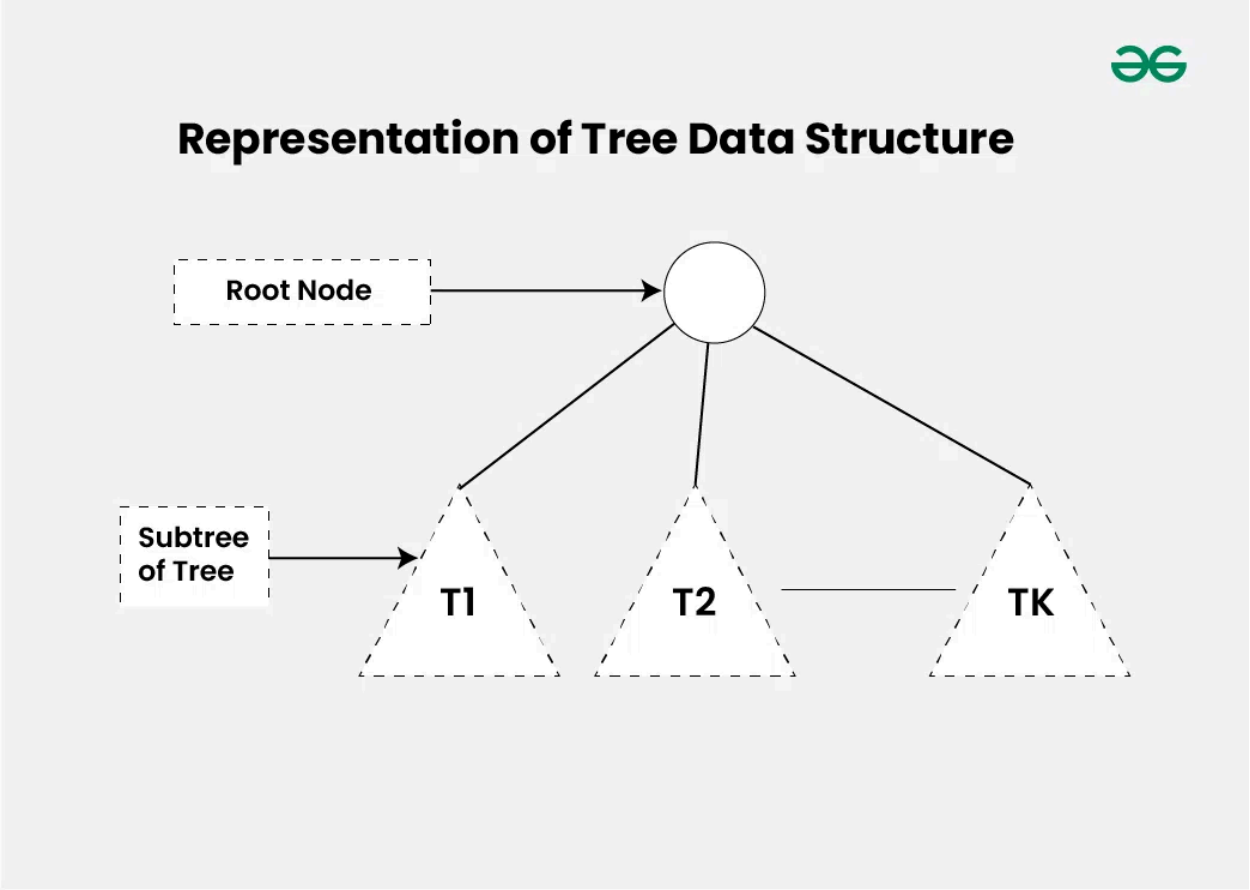
Different texts use different terms for the above defined terms, such as depth for height. branch or edge for link, arity for degree, external node for leaf node and internal node for a node other than a leaf node.

REPRESENTATION OF TREES

Let us define a tree. A tree is a finite set of one or more nodes such that.

1) There is a specially designated node called the root.

2) The remaining nodes are partitioned into n (n → 0) disjoint sets T_1, T_2 ,.....,T_n. where each T_i, (i = 1, 2,....., n) is a tree; T_1,T_2,.....,T_n are called subtrees of the root.



To illustrate the above definition, let us consider the sample tree shown in Figure 7.6.

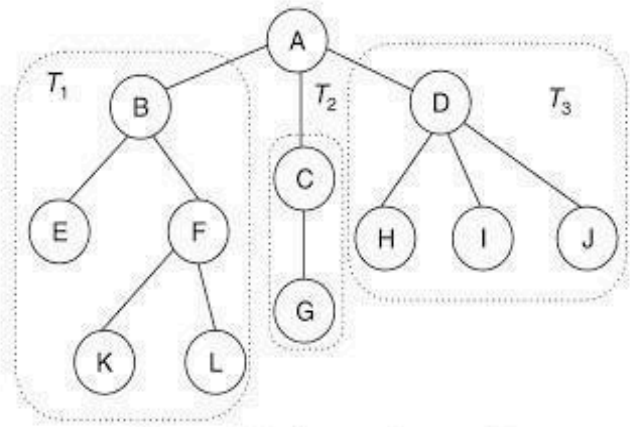


Figure 7.6 A sample tree *T*.

TYPES OF TREES

Tree data structure can be classified into three types based upon the number of children each node of the tree can have. The types are:

- Binary Tree
- Binary Search Tree (BST)
- AVL Tree
- B-Tree

1. BINARY TREES

- Any node N in a binary tree has either 0,1 or 2 successors.
- A tree can never be empty, but a binary tree may be empty.
- A tree can have any no. of children, but in a binary tree, a node can have at most two children.

Figure 7.7 depicts a sample binary tree.

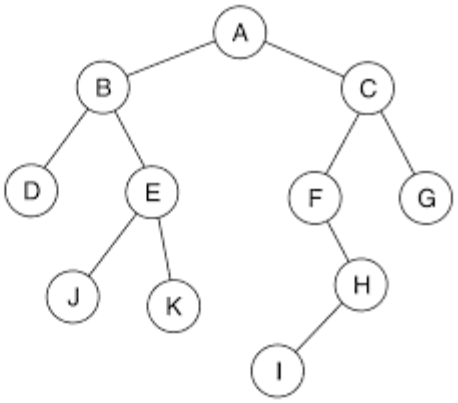


Figure 7.7 A sample binary tree with 11 nodes.

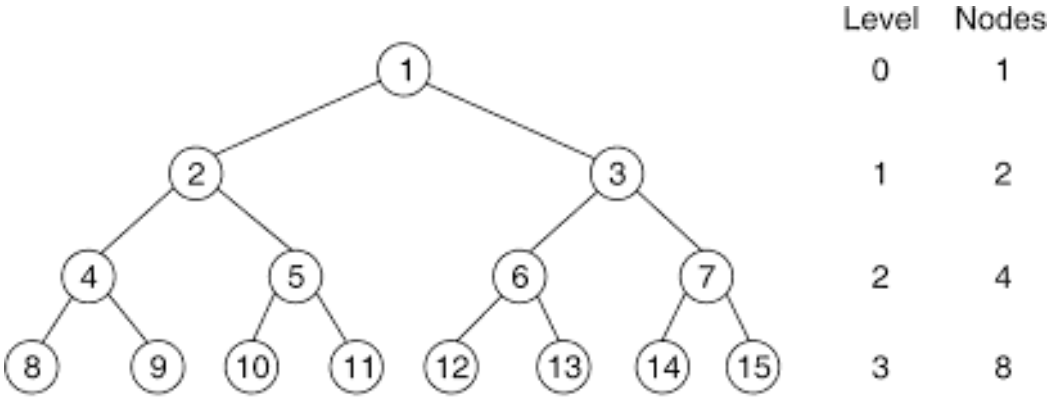
One can easily notice the main difference between the definitions of a tree and a binary tree. A tree can never be empty but a binary tree may be empty. Another difference is that in the case of a binary tree a node may have at most two children (that is, a tree having degree = 2), whereas in the case of a tree, a node may have any number of children.

Two special situations of a binary tree are possible:

- full binary tree
- complete binary tree.
- Skew binary tree

➢ **Full binary tree**

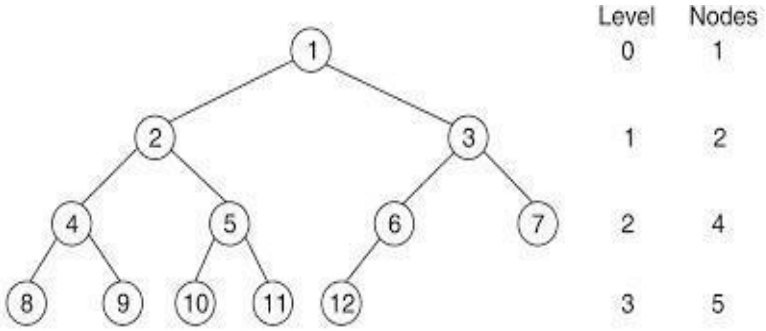
A binary tree is a full binary tree if it contains the maximum possible number of nodes at all levels. Figure 7.8(a) shows such a tree with height 4.



(a) A full binary tree of height 4

➢ **Complete binary tree**

A binary tree is said to be a complete binary tree if all its levels, except possibly the last level, have the maximum number of possible nodes, and all the nodes in the last level appear as far left as possible. Figure 7,8(b) depicts a complete binary tree.



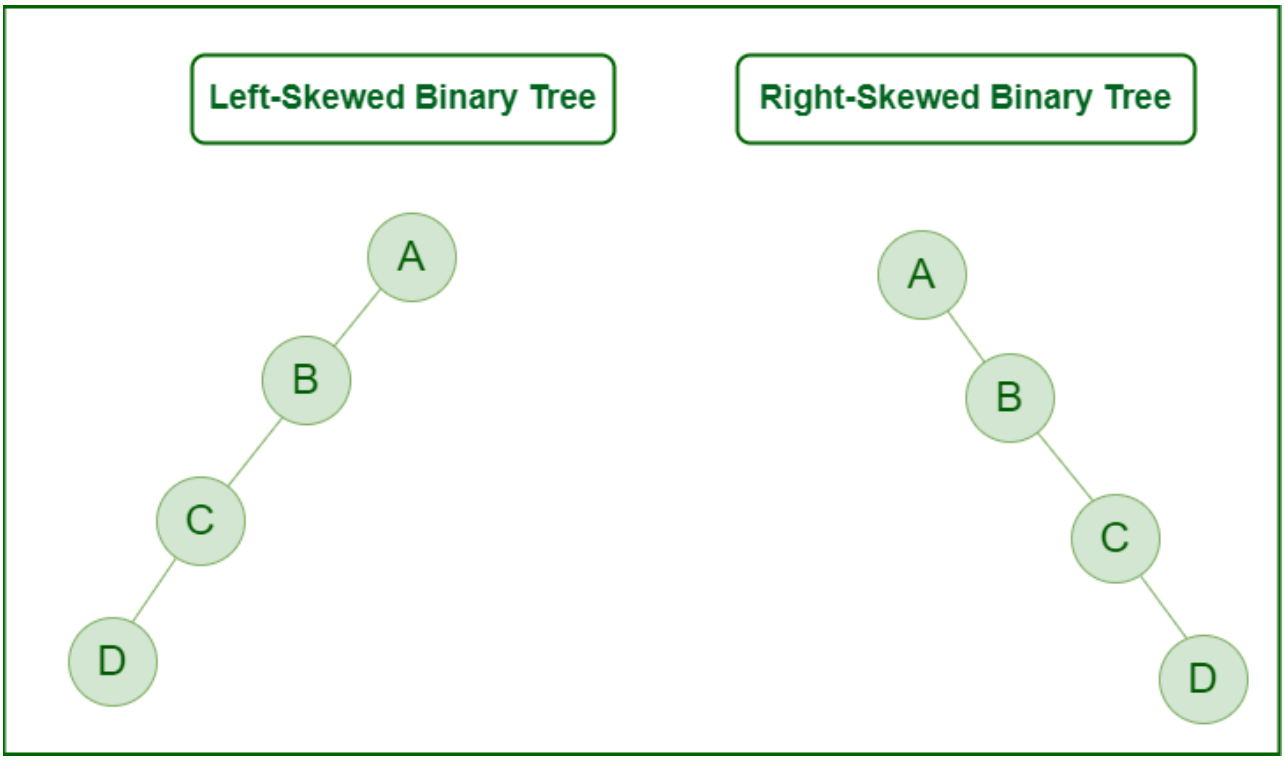
(b) A complete binary tree of height 14

Figure 7.8 Two special cases of binary trees.

Observe that the binary tree represented in Figure 7.7 is neither a full binary tree nor a complete binary tree.

➢ **Skew binary tree**

- Consider a binary tree with n nodes.
- A binary tree is said to be a skewed binary tree, if at all its levels, all nodes contain only one child .
- If the maximum height possible is h_{max}=n, then it is called skew binary tree.



Properties of Binary Tree

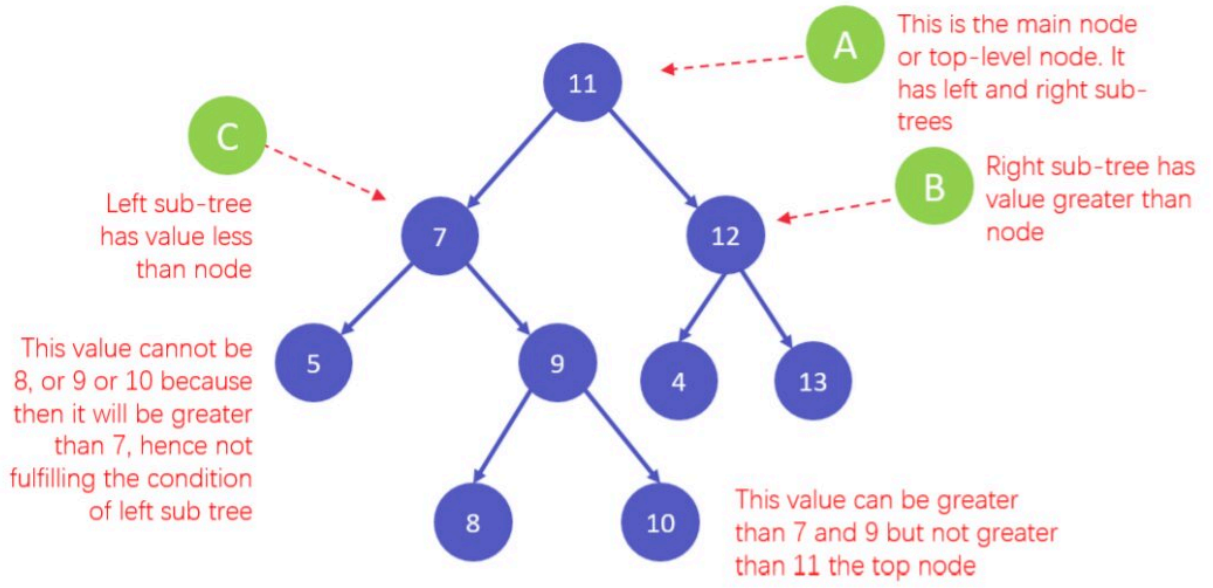
Some of the important properties of the binary tree are given here.

- At any level in a binary tree, there can only be a maximum of two nodes.
- The smallest binary tree with a height of H will have H + 1 nodes.
- The largest binary tree with a height of H will have 2H + 1 – 1 nodes.
- The number of leaf nodes in a binary tree is equal to the number of nodes with two children, plus one.
- The maximum number of nodes at level i in a binary tree is 2i.
- Searching in a binary tree takes a time complexity of O(log2N) time (where N is the number of nodes).

2. **Binary Search Tree**

A binary search tree (BST) is also called an ordered or sorted binary tree in which the value at the left sub-tree is lesser than that of the root and the right subtree has a value greater than that of the root.

Every binary search tree is a binary tree. However, not every binary tree is a binary search tree. What's the difference between a binary tree and a binary search tree? The most important difference between the two is that in a BST, the left child node's value must be less than the parent's, while the right child node's value must be higher.



Properties of a Binary Search tree

- Each node has a maximum of up to two children.
- The value of all the nodes in the left sub-tree is less than the value of the root.
- The value of all the nodes in the right subtree is greater than or equal to the value of the root.
- This rule is recursively valid for all the left and right subtrees of the root.

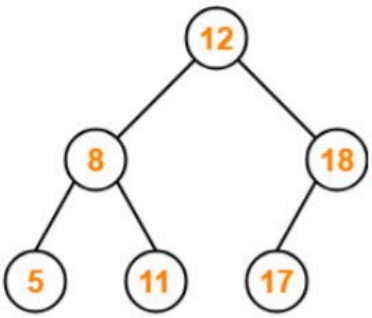
Applications of a Binary Search Tree

- Databases: Many databases use BSTs to store and search for data. For example, MySQL and SQLite use BSTs to store indexes of the data they contain, making queries much faster.

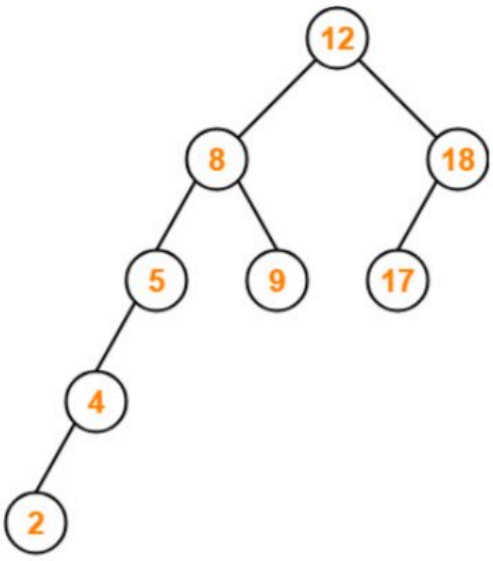
- File Systems: File systems use BSTs to store and organize files on a hard drive. For example, the Windows NTFS file system uses BSTs to store the Master File Table (MFT), which contains information about all the files on the hard drive.
- Computer Networks: BSTs can be used to store routing tables in computer networks. This allows routers to quickly find the best route for a packet to take from one network to another.
- Machine Learning: BSTs can be used in decision tree-based machine learning algorithms, where the decision tree is a binary search tree that makes decisions based on the input features.
- Game Trees: BSTs can be used to represent game trees in game theory, allowing for efficient searching of possible game states and moves.

3. AVL TREE

AVL trees are a special kind of self-balancing binary search tree where the height of every node's left and right subtree differs by at most one.



AVL Tree Example



Not an AVL Tree

Properties of an AVL tree

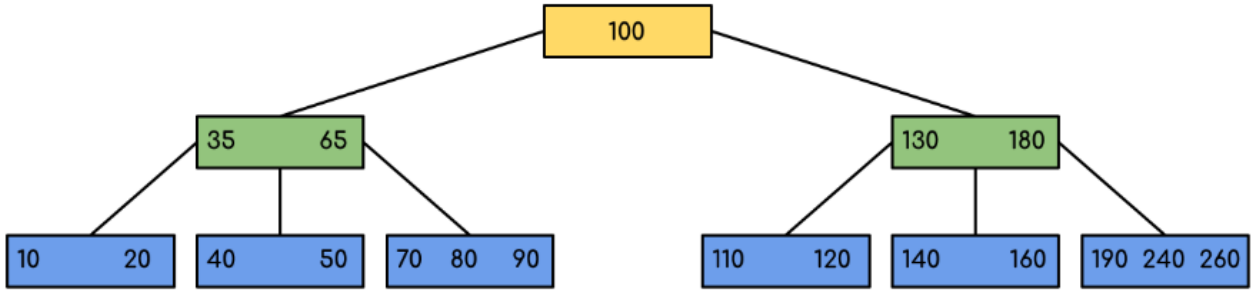
- The heights of the two child subtrees of any node differ by at most one.
- Balance Factor = (Height of Left Subtree – Height of Right Subtree).
- -1 Balance factor represents that the right subtree is one level higher than the left.
- 0 Balance factor represents that the height of the left subtree is equal to that of the right subtree.
- 1 Balance factor means that the left subtree is one level higher than the right subtree.
- The maximum possible number of nodes in the AVL tree of height H is $2H+1 – 1$
- The minimum number of nodes in the AVL Tree of height H is given by a recursive relation: $N(H) = N(H-1) + N(H-2) + 1$
- Minimum possible height of AVL Tree using N nodes = $\lfloor \log 2N \rfloor$ i.e floor value of log 2N
- The maximum height of the AVL Tree using N nodes is calculated using recursive relation: $N(H) = N(H-1) + N(H-2) + 1$

Applications of AVL trees

- In-memory sorts of sets and dictionaries
- Database applications that require frequent lookups for data

4. **B-TREE**

B tree is a self-balancing search tree wherein each node can contain more than one key and more than two children. It is a special type of m-way tree and a generalized binary search tree. B-tree can store many keys in a single node and can have multiple child nodes. This reduces the height and enables faster disk access.



Properties of a B-Tree

- Every node contains at most m children.
- Every node contains at least m/2 children (except the root node and the leaf node).
- The root nodes should have a minimum of 2 nodes.
- All leaf nodes should be at the same level.

Application of B-trees

- Databases and file systems
- Multilevel indexing
- For quick access to the actual data stored on the disks
- To store blocks of data

REPRESENTATION OF BINARY TREE

Implicit & Explicit representation

- **Implicit representation**
  - **Sequential / Linear** representation, using arrays.
- **Explicit representation**
  - **Linked list representation,** using pointers**.**

1) Sequential representation

- This representation is static.
- A block of memory for an array is allocated, before storing the actual tree.
- Once the memory is allocated, the size of the tree will be fixed.
- Nodes are stored level by level, starting from the zeroth level.
- The root node is stored in the starting memory location, as the first element of the array.
- Consider a linear array TREE

Characteristics

- Space Efficiency: For complete or nearly complete binary trees, array representation is space-efficient. However, for sparse trees, it can waste a lot of space due to the empty slots in the array.
- Performance: Accessing nodes, as well as their children and parents, is fast because it involves simple arithmetic operations on indices.
- Limitations: The size of the tree is fixed and needs to be known in advance, or resizing the array might be required, which is a costly operation.

Rules for storing elements in TREE are:

1. The root R of T is stored in location 0.
2. For any node with index i, 1<i<=n:

PARENT(i)= i/2

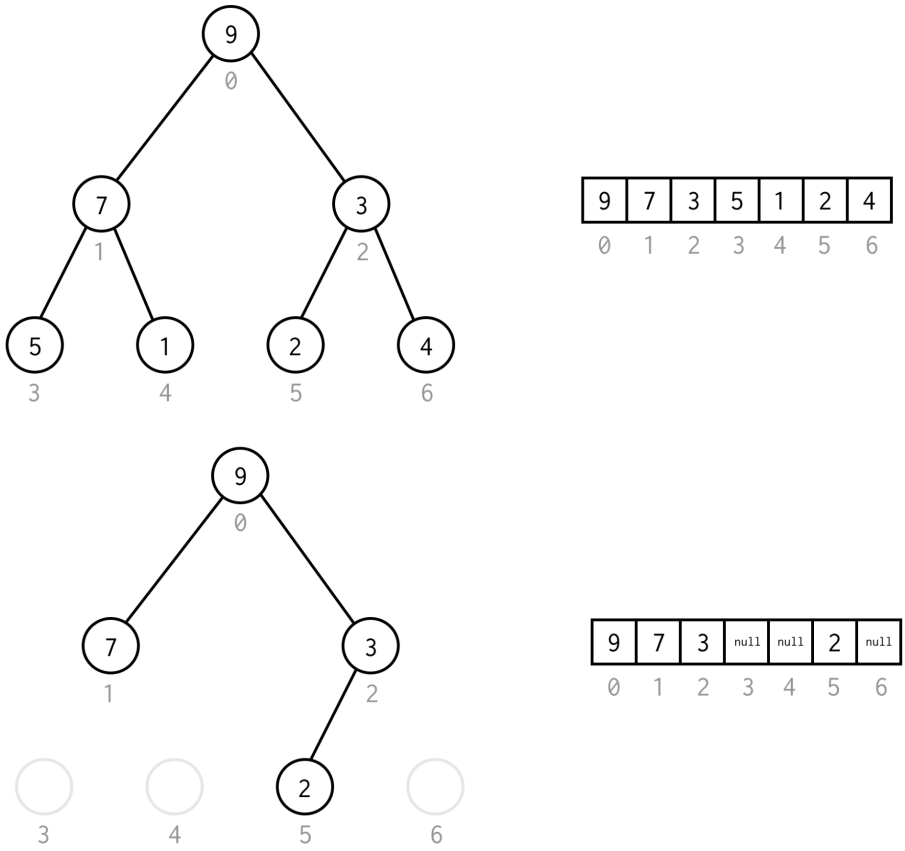
For the node when i=1,there is no parent.

LCHILD(i)=2*i

If 2* i >n, then i has no left child

RCHILD(i)=2*i+1

If 2*i+1>n, then i has no right child



Sequential representation- Advantages:

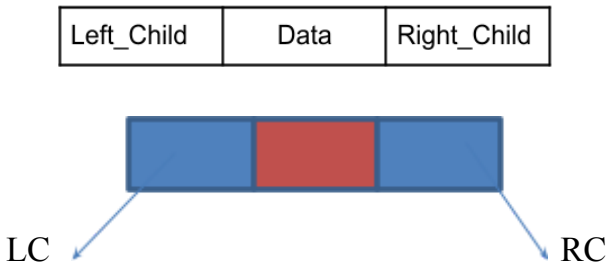
1. Any node can be accessed from any other node by calculating the index.
2. Here, data is stored simply without any pointers to their successor or predecessor.
3. Programming languages, where dynamic memory allocation is not possible(like BASIC, FORTRAN), array representation is only possible.

Sequential representation- Disadvantages:

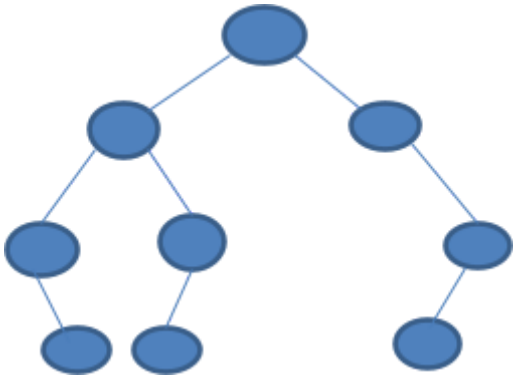
1. Other than full binary trees, the majority of the array entries may be empty.
2. It allows only static representation. It is not possible to enhance the tree structure, if the array structure is limited.
3. Inserting a new node and deletion of an existing node is difficult, because it requires considerable data movement

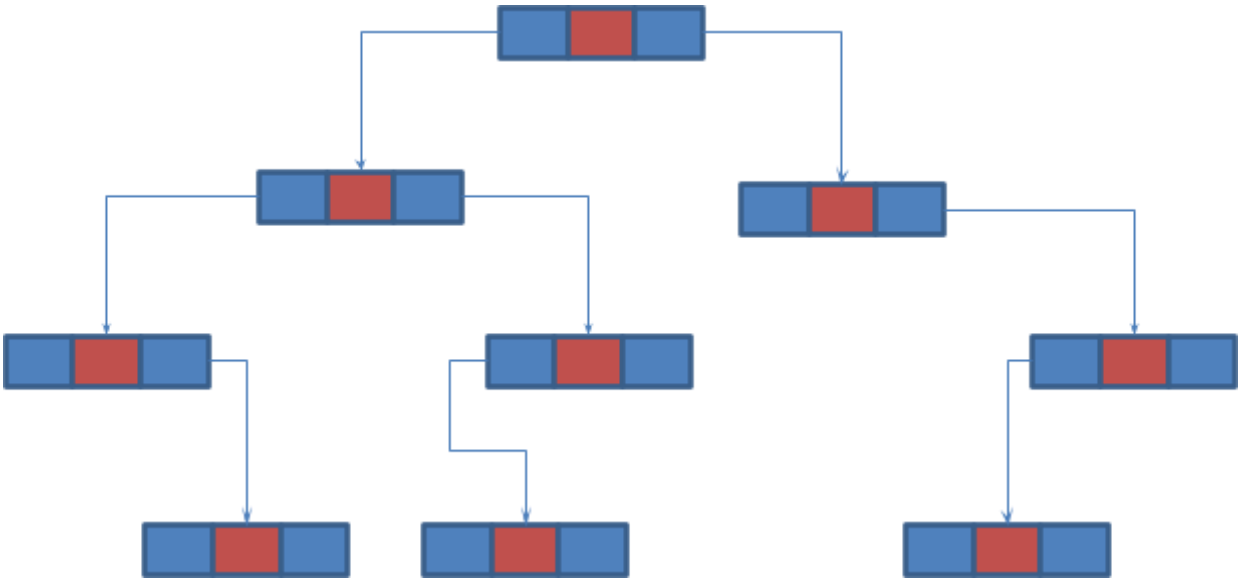
2) Linked list representation

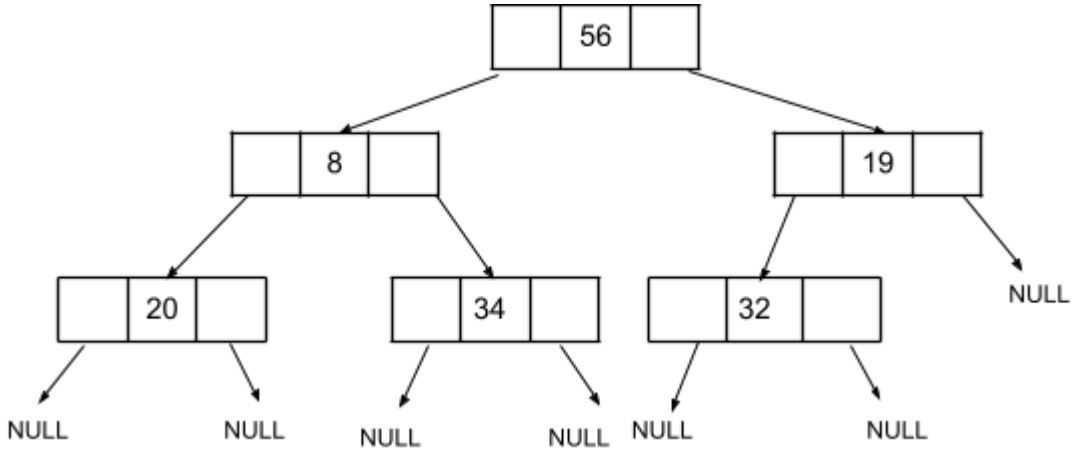
- It consist of three parallel arrays DATA, LC and RC



- Each node N of T will correspond to a location K such that:
  - DATA[K] contains the data at the node N
  - LC[K] contains the location of the left child of node N
  - RC[K] contains the location of the right child of node N







Characteristics

- **Space Efficiency:** This method is more space-efficient for sparse trees, as it allocates memory only for existing nodes.
- **Dynamic Size:** The tree can grow or shrink dynamically, allowing for more flexibility in operations like addition or deletion of nodes.
- **Performance:** Accessing children or parent nodes requires following pointers, which might be slightly slower than direct index-based access in arrays.
- **Overhead:** Each node requires extra memory for the pointers, in addition to the data it stores.

```
struct node {
        int data;
          struct node * left;
            struct node *right;
            };
```

Linked list representation - Advantages:

- It can easily grow or shrink as needed, so it uses only the memory it needs.
- Adding or removing nodes is straightforward and requires only pointer adjustments.
- Only uses memory for the nodes that exist, making it efficient for sparse trees.

Linked list representation -Disadvantages:

- Needs extra memory for pointers.
- Finding a node can take longer because you have to start from the root and follow pointers.

OPERATIONS OF TREE

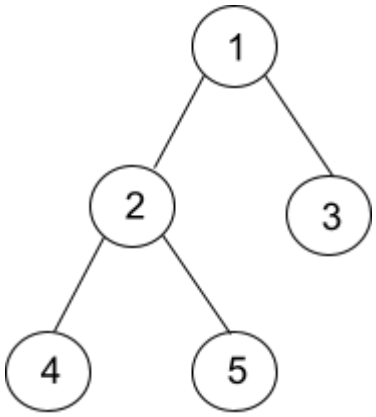
Tree operations include insertion, deletion, searching, and traversal. These operations are fundamental for managing and manipulating data within tree structures:

1. Insertion

Insertion involves adding a new node to the tree. The location of the new node depends on the type of tree.

Example:

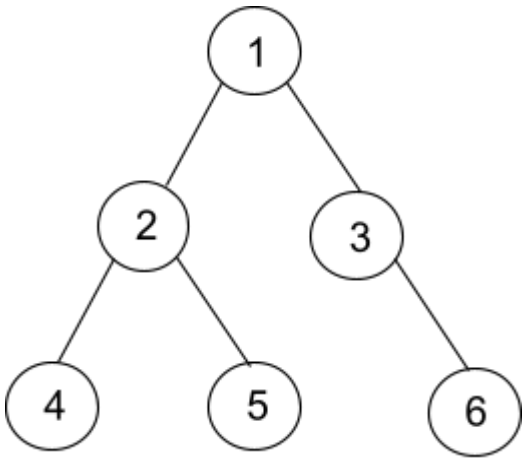
Consider inserting the value 6 into a binary tree:



Steps to Insert 6:

1. Start at the root node.
2. Find the first available position in level order.
3. Insert 6 as the right child of 3.

Result:

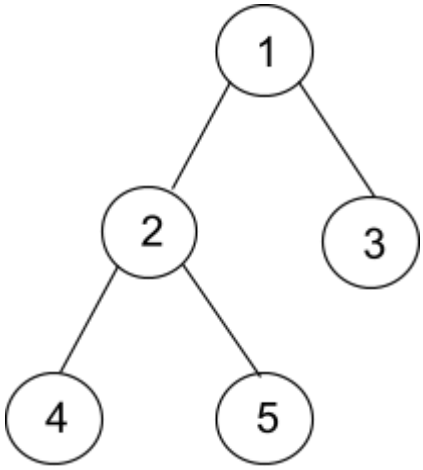


2. Deletion

Deletion involves removing a node from the tree. In a **binary tree**, the node to be deleted is replaced by the deepest and rightmost node to maintain the tree's structure.

Example:

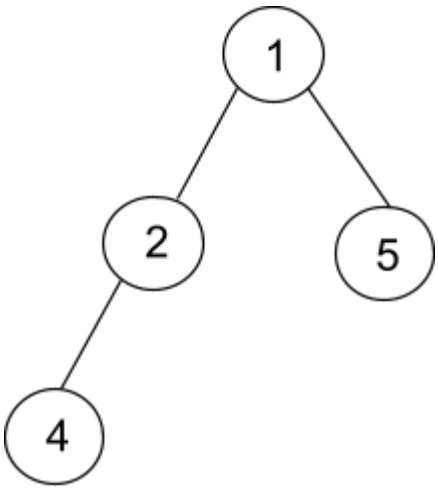
Consider deleting the value 3 from the following binary tree:



Steps to Delete 3:

1. Find the node to be deleted (3).
2. Find the deepest and rightmost node (5).
3. Replace 3 with 5.
4. Remove the deepest and rightmost node.

Result:

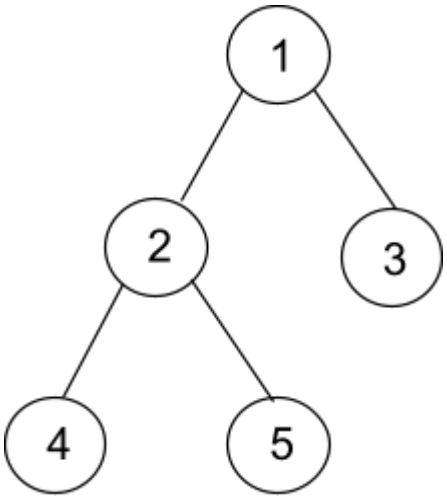


3. Search

Searching involves finding a node with a given value in the tree. The search operation can

be implemented using any traversal method.

Example:
Consider searching for the value 4 in the following binary tree:



Steps to Search for 4:

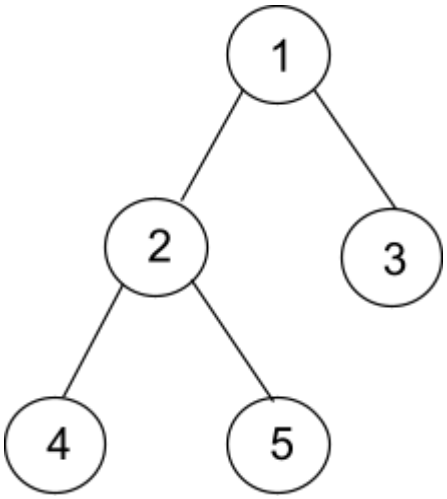
1. Start at the root node (1).
2. Check the left subtree.
3. Move to node 2.
4. Check the left subtree of node 2.
5. Find the node 4.

Result: Node 4 is found.

4. Traversal

Traversal involves visiting all the nodes in the tree in a specific order. The main traversal methods are in-order, pre-order, post-order, and level-order.

Example:
Consider the following binary tree:



In-order Traversal:

1. Traverse left subtree of 1: 4, 2, 5
2. Visit root node: 1
3. Traverse right subtree of 1: 3

Result: 4, 2, 5, 1, 3

Pre-order Traversal:

1. Visit root node: 1
2. Traverse left subtree of 1: 2, 4, 5
3. Traverse right subtree of 1: 3

Result: 1, 2, 4, 5, 3

Post-order Traversal:

1. Traverse left subtree of 1: 4, 5, 2
2. Traverse right subtree of 1: 3
3. Visit root node: 1

Result: 4, 5, 2, 3, 1

Level-order Traversal:

Visit nodes level by level from left to right.

Result: 1, 2, 3, 4, 5

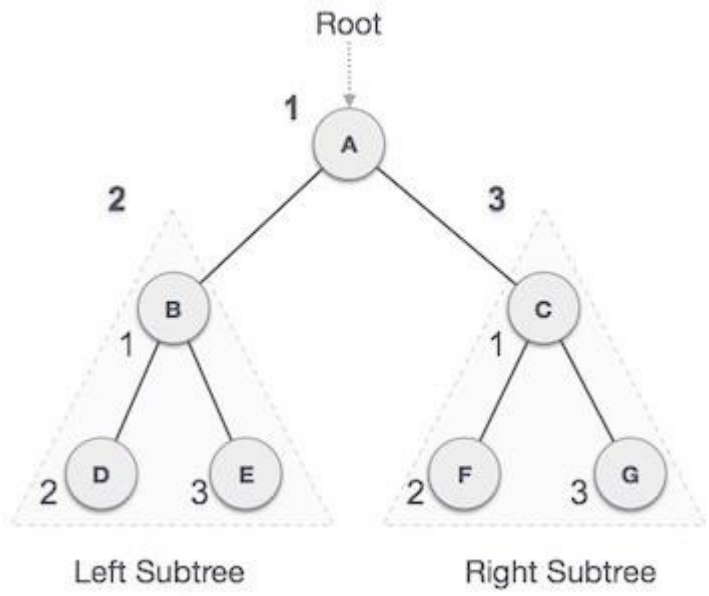
BINARY TREE TRAVERSALS

- Traversal is a process to visit all the nodes of a tree and may print their values too.
- Because all nodes are connected via edges (links) we always start from the root node.
- That is, we cannot randomly access a node in a tree.
- There are three ways which we use to traverse a tree –
    1. **Preorder traversal (R, T_l, T_r)**
    2. **Inorder traversal (T_l, R, T_r)**
    3. **Postorder traversal (T_l, T_r, R)**

Preorder Traversal

In this traversal, the root is visited first, then the left sub-tree in preorder fashion, and then the right sub-tree in preorder fashion. Such a traversal can be defined as follows:

- Visit the root node R.
- Traverse the left sub-tree of R in preorder.
- Traverse the right sub-tree of R in preorder.



RESULT:- **A → B → D → E → C → F → G**

Algorithm Preorder

Input: ROOT is the pointer to the root node of the binary tree.
Output: Visiting all the nodes in preorder fashion.
Data structure: Linked structure of binary tree.

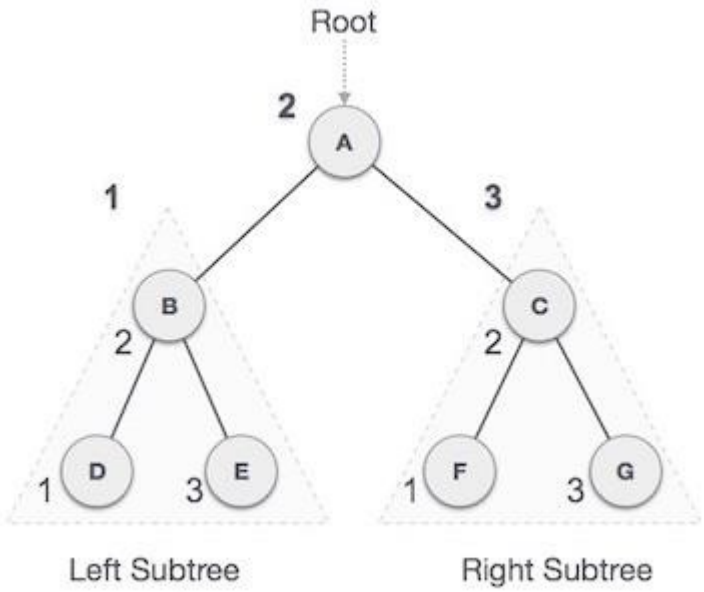
Steps:

```
1. ptr = ROOT                                  // Start from the ROOT
2. If (ptr ≠ NULL) then                        // If it is not an empty node
3.     Visit(ptr)                              // Visit the node
4.     Preorder(ptr→LC)                        // Traverse the left sub-tree of the node in preorder
5.     Preorder(ptr→RC)                        // Traverse the right sub-tree of the node in preorder
6. EndIf
7. Stop
```

Inorder Traversal

With this traversal, before visiting the root node, the left sub-tree of the root node is visited, then the root node and after the visit of the root node the right sub-tree of the root node is visited. Visiting both the sub-trees is in the same fashion as the tree itself. Such a traversal can be stated as follows:

- Traverse the left sub-tree of the root node R in inorder.
- Visit the root node R.
- Traverse the right sub-tree of the root node R in inorder.



RESULT:- **D B E A F C G**

Algorithm Inorder

Input: ROOT is the pointer to the root node of the binary tree.
Output: Visiting all the nodes in inorder fashion.
Data structure: Linked structure of binary tree.

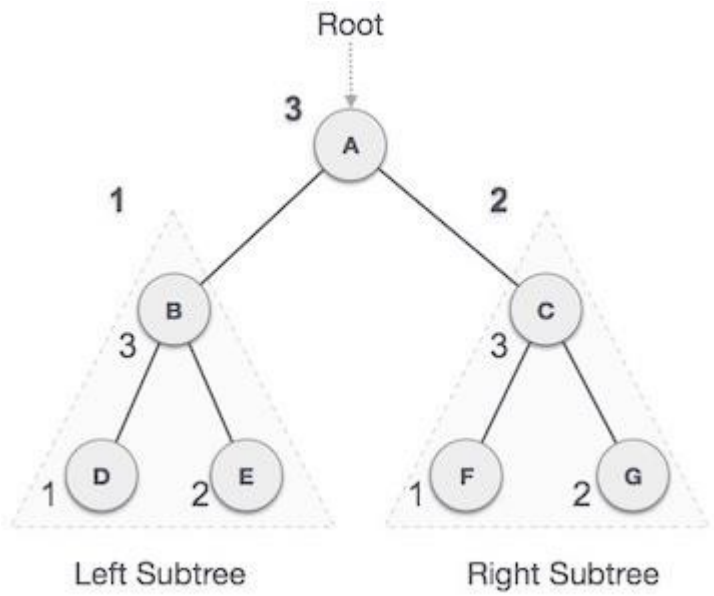
Steps:

```
1. ptr = ROOT                                   // Start from ROOT
2. If (ptr ≠ NULL) then                         // If it is not an empty node
3.     Inorder(ptr→LC)                          // Traverse the left sub-tree of the node in inorder
4.     Visit(ptr)                               // Visit the node
5.     Inorder (ptr→RC)                         // Traverse the right sub-tree of the node in inorder
6. EndIf
7. Stop
```

Postorder Traversal

Here, the root node is visited in the end, that is, first visit the left sub-tree, then the right sub tree, and lastly the root. A definition for this type of tree traversal is stated below:

- Traverse the left sub-tree of the root R in postorder
- Traverse the right sub-tree of the root R in postorder
- Visit the root node R.



RESULT:- **D → E → B → F → G → C → A**

Algorithm Postorder

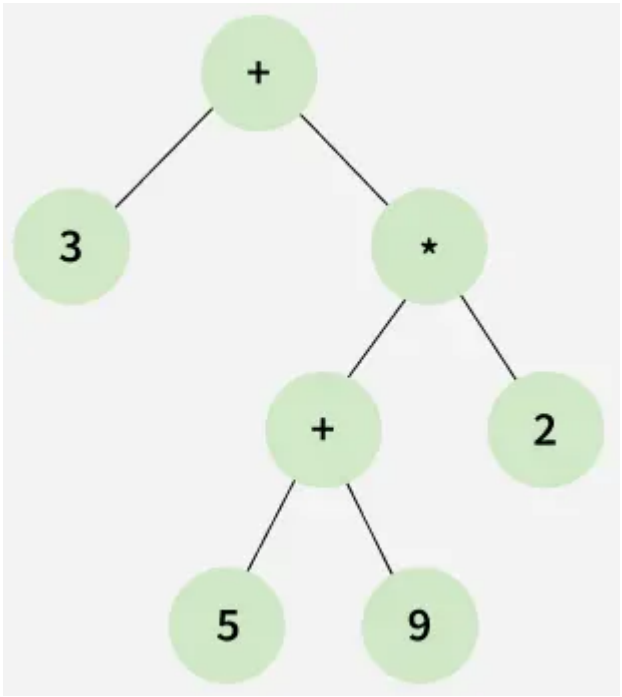
Input: ROOT is the pointer to the root node of the binary tree.
Output: Visiting all the nodes in preorder fashion.
Data structure: Linked structure of binary tree.

Steps:

```
1. ptr = ROOT                                   // Start from the root
2. If (ptr ≠ NULL) then                         // If it is not an empty node
3.     Postorder(ptr→LC)                        // Traverse the left sub-tree of the node in inorder
4.     Postorder(ptr→RC)                        // Traverse the right sub-tree of the node in inorder
5.     Visit(ptr)                               // Visit the node
6. EndIf
7. Stop
```

Expression Tree

An expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for 3 + ((5 + 9) * 2) would be:



7.5.2 Binary Search Tree

A binary tree T is termed *binary search* tree (or *binary sorted* tree) if each node N of T satisfies the following property:

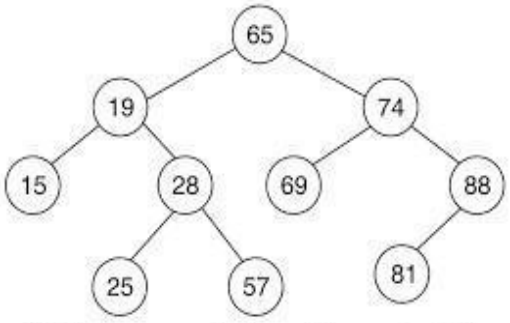
The value at N is greater than every value in the left sub-tree of N and is less than every value in the right sub-tree of N.

Figure 7.31 shows two binary search trees for two different types of data.

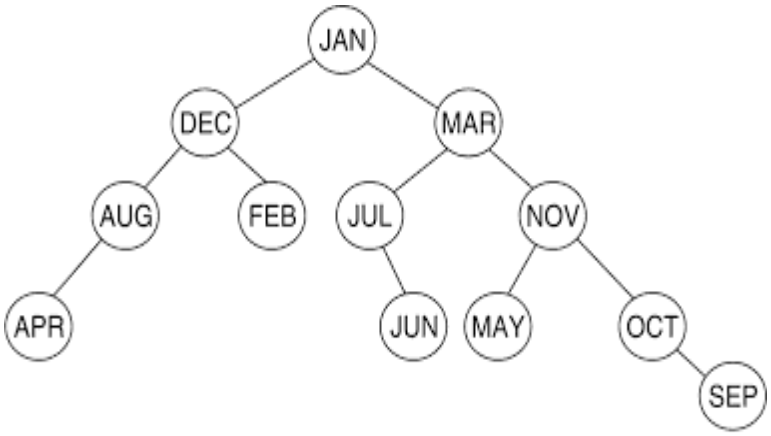
Observe that in Figure 7.31(b), the lexicographical ordering is taken among the data whereas in Figure 7.31(a), numerical ordering is taken.

Now, let us discuss the possible operations on any binary search tree. Operations which are generally encountered to manipulate this data structure are:

- Searching data
- Inserting data
- Deleting data
- Traversing the tree.



(a) A binary search tree with numeric data



(b) A binary search tree with alphabetic data

Searching a binary search tree

Searching data in a binary search tree is much faster than searching data in arrays or linked lists. This is why in the applications where frequent searching operations need to be performed, this data structure is used to store data. In this section, we will discuss how this operation can be defined.

Suppose in a binary search tree *T*, ITEM the item of search. We will assume that the tree is represented using a linked structure.

We start from the root node *R*. Then, if ITEM is less than the value in the root node *R*, we proceed to its left child; if ITEM is greater than the value in the node *R*, we proceed to its right child. The process will be continued till the ITEM is not found or we reach a dead end, that is, the leaf node. Figure 7.32 shows the track (in shaded line) for searching of 54 in a binary search tree.

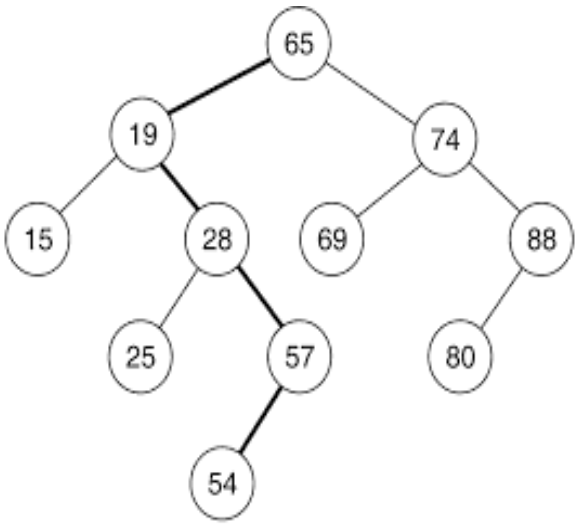


Figure 7.32 Searching 54 in a binary search tree.

Algorithm Search_BST

Input: ITEM is the data that has to be searched.
Output: If found then pointer to the node containing data ITEM else a message.
Data structure: Linked structure of the binary tree. Pointer to the root node is ROOT.

Steps:

```
1.  ptr = ROOT, flag = FALSE                                  // Start from the root
2.  While (ptr ≠ NULL) and (flag = FALSE) do
3.      Case: ITEM < ptr→DATA                                 // Go to the left sub-tree
4.          ptr = ptr→LCHILD
5.      Case: ptr→DATA = ITEM                                 // Search is successful
6.          flag = TRUE
7.      Case: ITEM > ptr→DATA                                 // Go to the right sub-tree
8.          ptr = ptr→RCHILD
9.      EndCase
10. EndWhile
11. If (flag = TRUE) then                                     // Search is successful
12.     Print "ITEM has found at the node", ptr
13. Else
14.     Print "ITEM does not exist: Search is unsuccessful"
15. EndIf
16. Stop
```

Inserting a node into a binary search tree

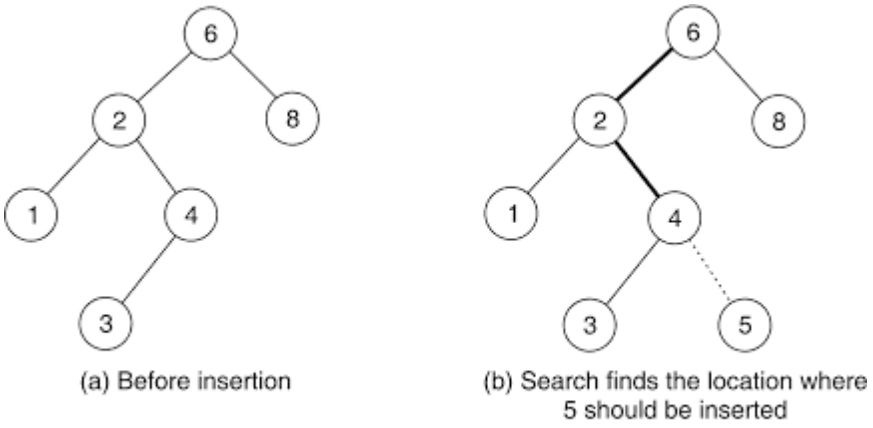
The insertion operation on a binary search tree is conceptually very simple. It is, in fact, one step more than the searching operation. To insert a node with data, say ITEM, into a tree, the tree is required to be searched starting from the root node. If ITEM is found, do nothing, otherwise ITEM is to be inserted at the dead end where the search halts. Figure 7.33 shows the insertion of 5 into a binary tree. Here, search proceeds starting from the root node as 6–2–4 then halts when it finds that the right child is null (dead end). This simply means that if 5 occurs, then it should have occurred on the right part of the node 4. So, 5 should be inserted as the right child of 4.

Algorithm Insert_BST

Input: ITEM is the data component of a node that has to be inserted.
Output: If there is no node having data ITEM, it is inserted into the tree else a message.
Data structure: Linked structure of binary tree. Pointer to the root node is ROOT.

1. Start
2. Create a node temp and insert ITEM in it.
3. If(ROOT==null)
4. Set ROOT=temp
5. Else
6. Set ptr= ROOT

```
7.  while(ptr≠ null)
8.          Set parent= ptr
9.          if( ITEM< ptr->data)
10.             ptr=ptr->LCHILD
11.             if( ptr==null)
12.                 parent->LCHILD=temp
13.         else
14.                 ptr= ptr->RCHILD
15.                   if (ptr==null)
16.                   parent->RCHILD= temp
```



(a) Before insertion

(b) Search finds the location where 5 should be inserted

Figure 7.33 Inserting 5 into a binary search tree.

Deletion in a BST

- There are the following possible cases when we delete a node:

  1. The node to be deleted has **no children**. In this case, all we need to do is delete the node.

  2. The node to be deleted has **only one child** (left or right subtree). We delete the node and attach the subtree to the deleted node's parent.

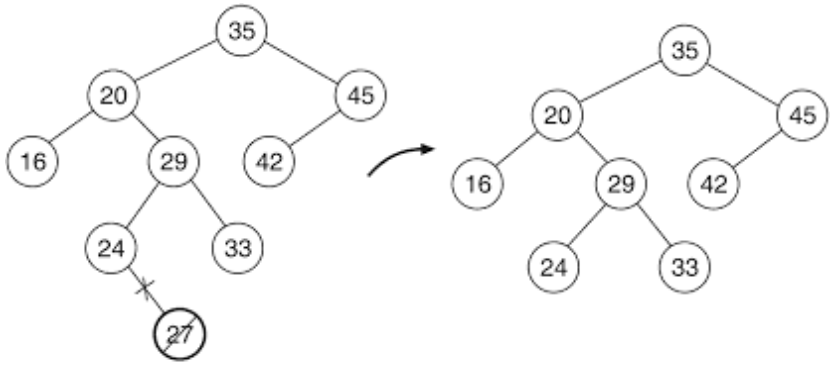
  3. The node to be deleted has **two children**. It is possible to delete a node from the middle of a tree, but the result tends to create very unbalanced trees.

Deletion from the middle of a tree

- We can find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data.
- We can find the smallest node on the deleted node's right subtree and move its data to replace the deleted node's data.
- Either of these moves preserves the integrity of the binary search tree.

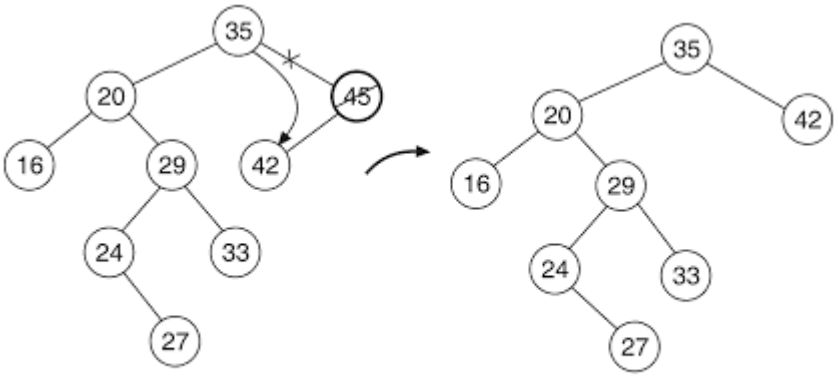
Deletion in a BST: Example

Case 1: The node to be deleted has no children.



1. Deletion of the node 27

Case 2: The node to be deleted has exactly one child.

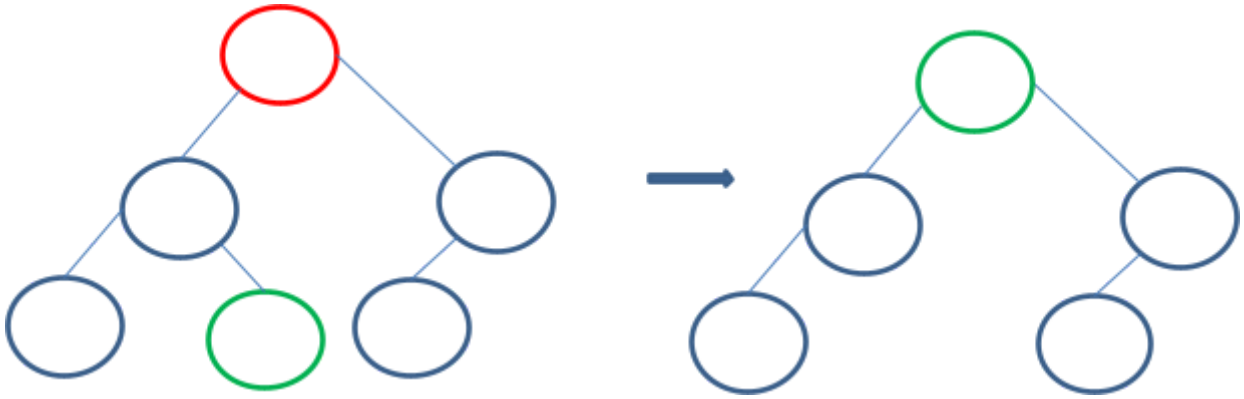


2. Deletion of the node 45

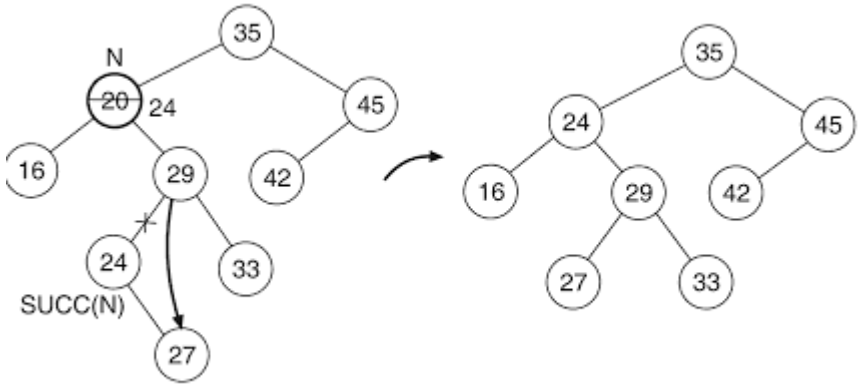
<u>Case 3</u>: The node to be deleted has two children.

Two methods:

1) We can find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data.



2) We can find the smallest node on the deleted node's right subtree and move its data to replace the deleted node's data.



(c) Deletion of the node 20

Deletion Algorithm

Delete(item, ptr)

1. if ptr != null then do step 2 – 7
2. if item < ptr -> data then
   Delete(item,ptr->lchild)
3. else if item > ptr -> data
   Delete(item, ptr -> rchild)
4. else if (ptr -> lchild = null) and (ptr -> rchild = null)
   ptr = null // Deleting leaf node
5. else if (ptr -> lchild = null) then ptr = ptr -> rchild // Single child
6. else if (ptr -> rchild = null) then ptr = ptr -> lchild // Single child
7. else set ptr -> data = deletemin(ptr -> rchild) // Deleting if more children are present

Function deletemin(ptr)

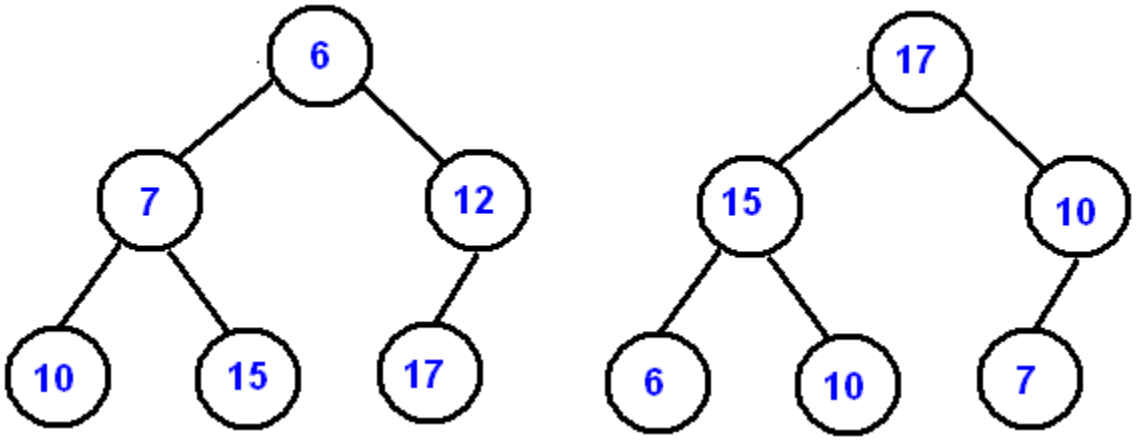
1. if ptr -> lchild = null then return ptr ->item
2. else return deletemin(ptr -> lchild)

BINARY HEAP

A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:

- the *min-heap property*: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
- the *max-heap property*: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

Throughout this chapter the word "heap" will always refer to a min-heap.



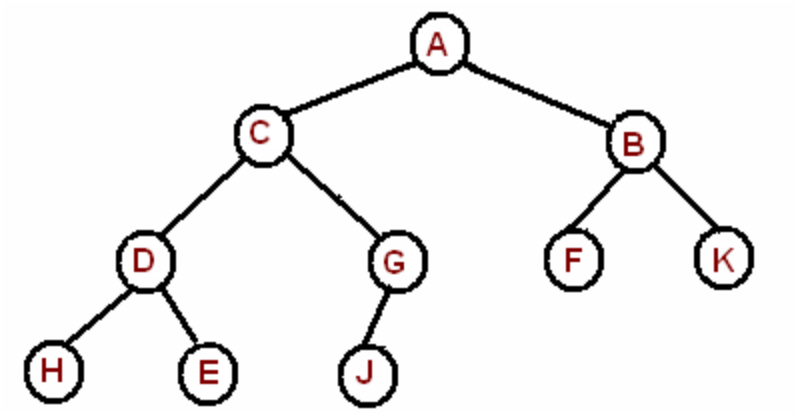
In a heap the highest (or lowest) priority element is always stored at the root, hence the name "heap". A heap is not a sorted structure and can be regarded as partially ordered. As you see from the picture, there is no particular relationship among nodes on any given level, even among the siblings.

Since a heap is a complete binary tree, it has the smallest possible height - a heap with N nodes always has O(log N) height.

A heap is a useful data structure when you need to remove the object with the highest (or lowest) priority. A common use of a heap is to implement a priority queue.

Array Implementation

A complete binary tree can be uniquely represented by storing its level order traversal in an array.



0	1	2	3	4	5	6	7	8	9	10
	A	C	B	D	G	F	K	H	E	J

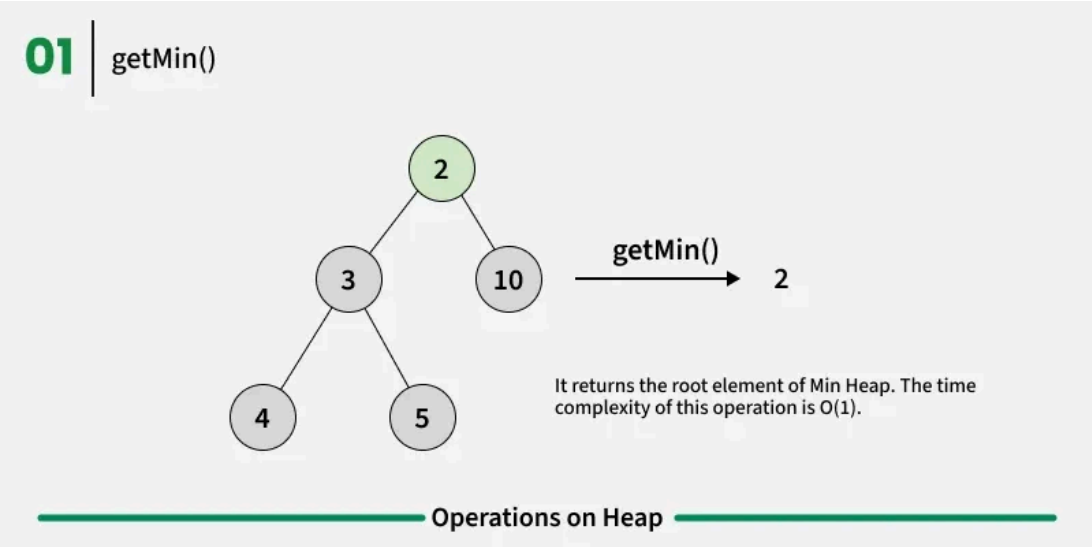
The root is the second item in the array. We skip the index zero cell of the array for the convenience of implementation. Consider k-th element of the array, the

its left child is located at 2*k index
its right child is located at 2*k+1. index
its parent is located at k/2 index

Operations of Binary Heap

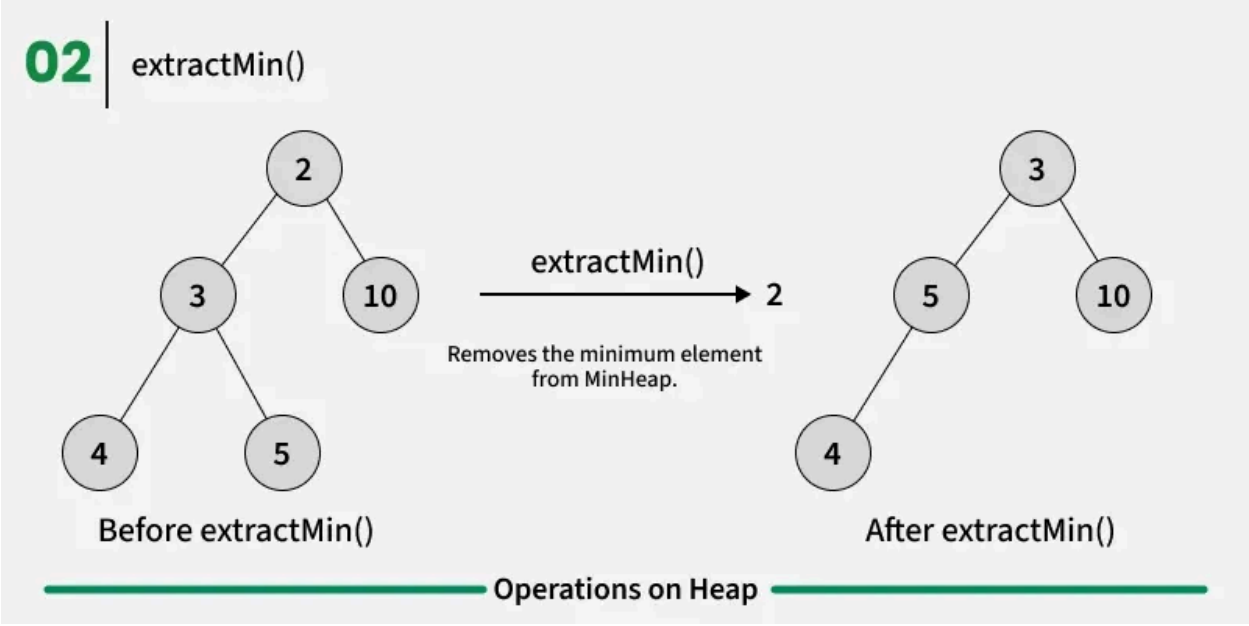
1. **getmin():-**

   It returns the minimum value in the Binary Heap. The root Node is the Minimum value in Min Heap.



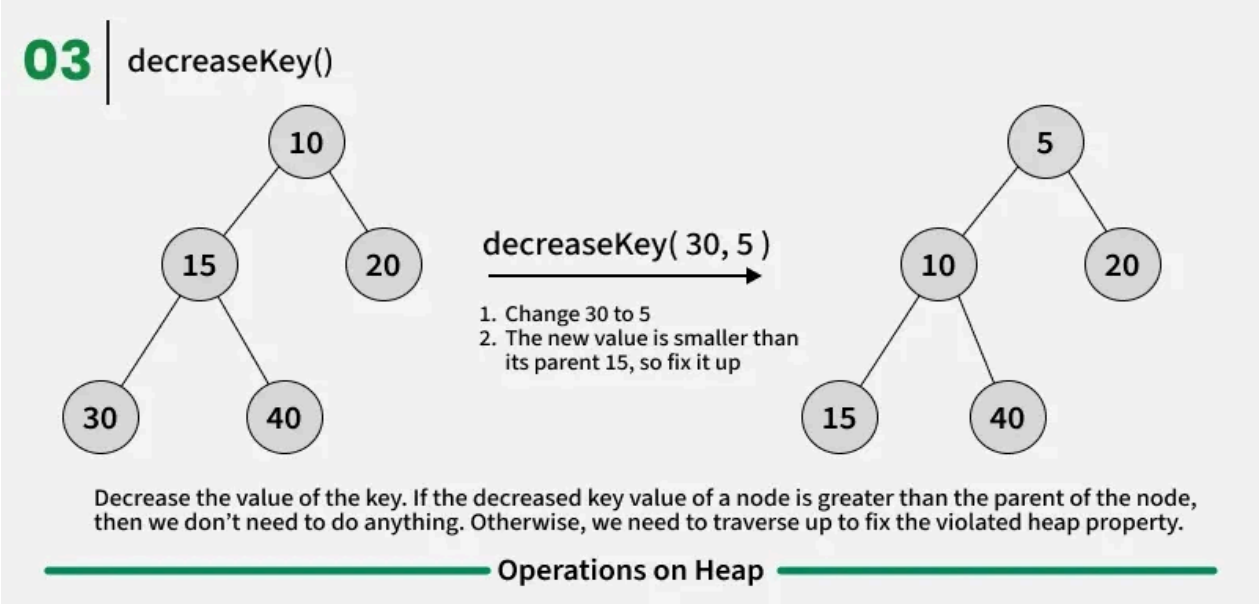
2. **exractmin()**

   The extractMin() operation in a binary heap (specifically a min-heap) removes and returns the minimum element from the heap while maintaining the heap property. This is done by first replacing the root (minimum) node with the last node in the heap, reducing the heap's size, and then performing a heapify operation on the root to restore the heap property



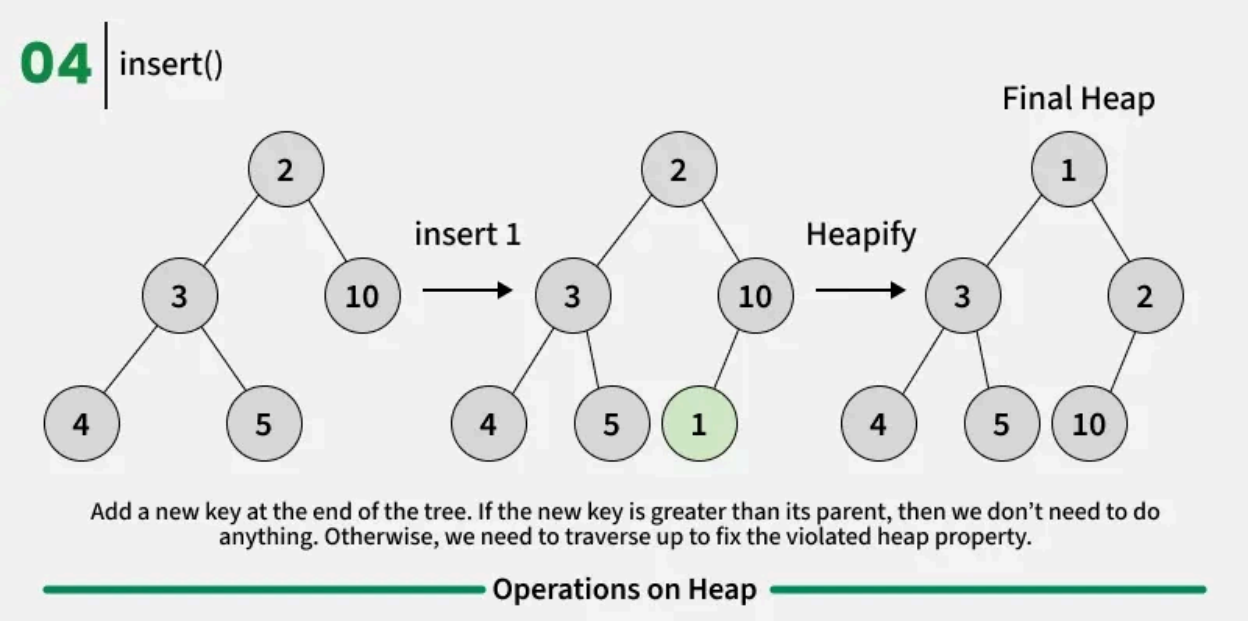
3. **decreasekey()**

The decrease key operation replaces the value of a node with a given value with a lower value, and the increase key operation does the same but with a higher value. This involves finding the node with the given value, changing the value, and then down-heapifying or up-heapifying to restore the heap property.



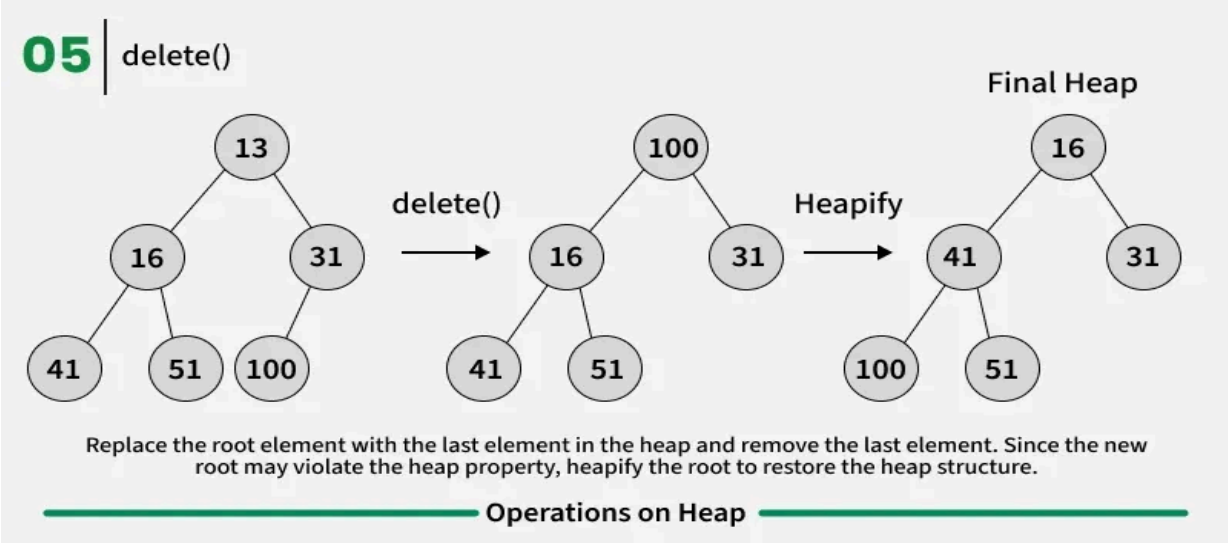
4. **Insert**

The new element is initially appended to the end of the heap (as the last element of the array). The heap property is repaired by comparing the added element with its parent and moving the added element up a level (swapping positions with the parent). This process is called "percolation up". The comparison is repeated until the parent is larger than or equal to the percolating element.



5. **DeleteMin**

The minimum element can be found at the root, which is the first element of the array. We remove the root and replace it with the last element of the heap and then restore the heap property by *percolating down*. Similar to insertion, the worst-case runtime is O{log n).

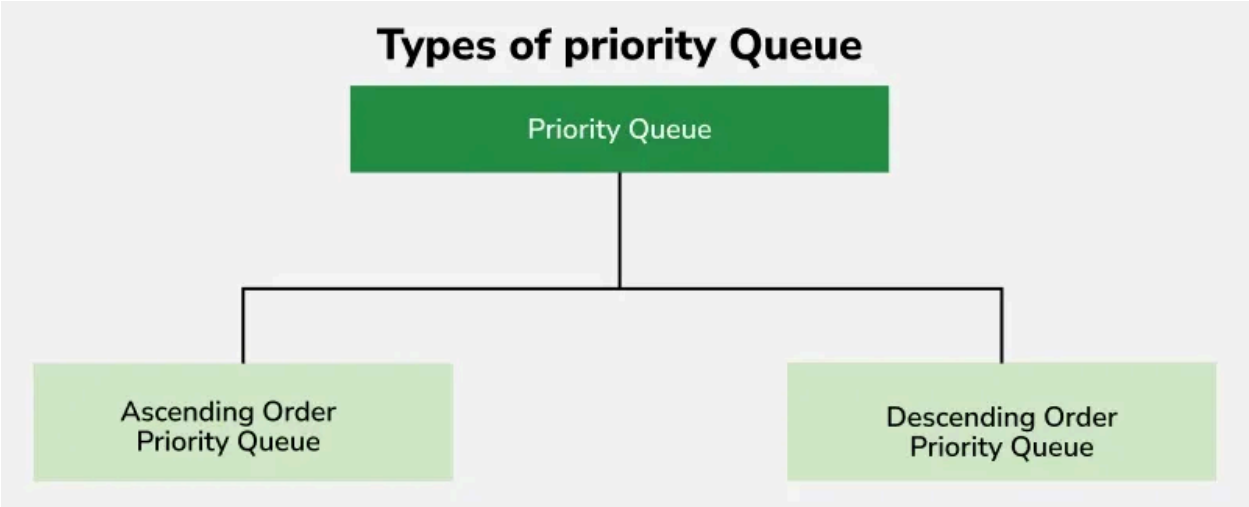


Priority Queue:-

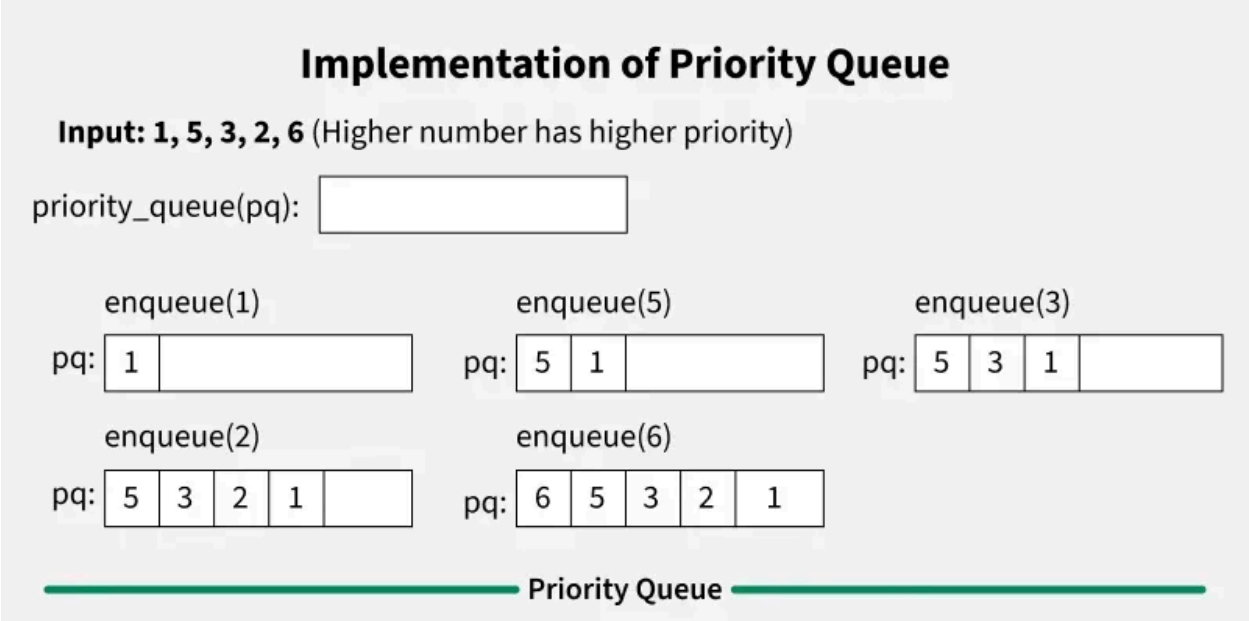
A priority queue is a type of queue that arranges elements based on their priority values.

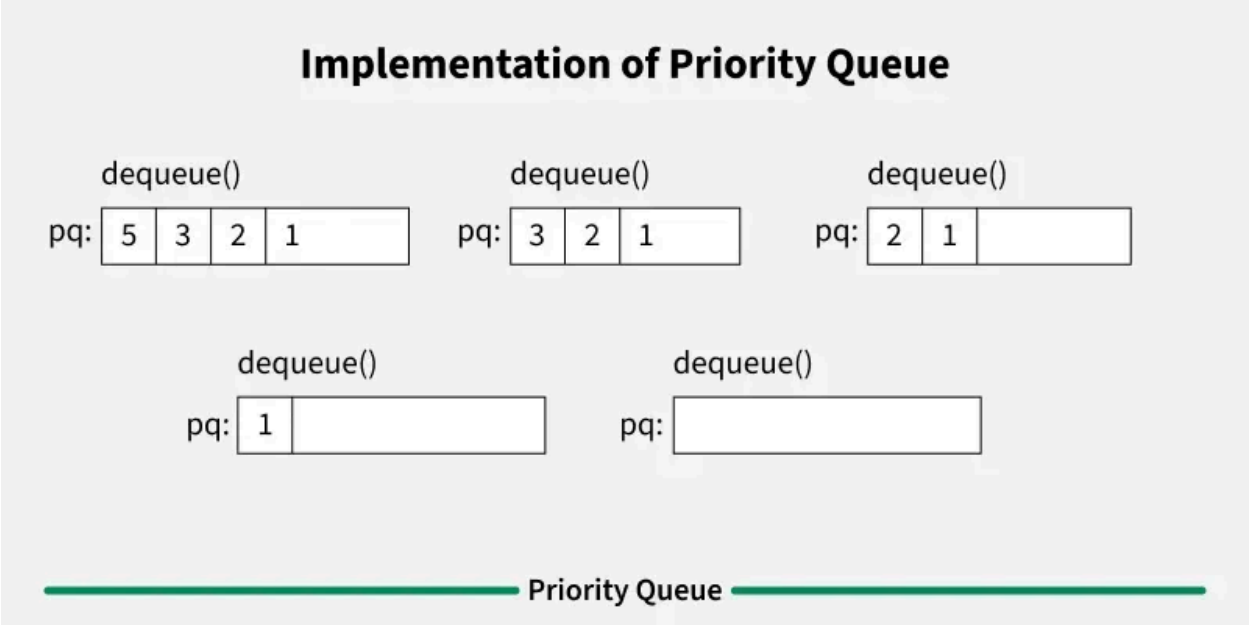
- Each element has a priority associated. When we add an item, it is inserted in a position based on its priority.
- Elements with higher priority are typically retrieved or removed before elements with lower priority.
- Binary heap is the most common method to implement a priority queue. In binary heaps, we have easy access to the min (in min heap) or max (in max heap) and binary heap being a complete binary tree are easily implemented using arrays. Since we use arrays, we have a cache friendliness advantage also.
- Priority Queue is used in algorithms such as Dijkstra's algorithm, Prim's algorithm, and Huffman Coding.

Types of Priority Queue



- Ascending Order Priority Queue : In this queue, elements with lower values have higher priority. For example, with elements 4, 6, 8, 9, and 10, 4 will be dequeued first since it has the smallest value, and the dequeue operation will return 4.
- Descending order Priority Queue : Elements with higher values have higher priority. The root of the heap is the highest element, and it is dequeued first. The queue adjusts by maintaining the heap property after each insertion or deletion.





Operations on a Priority Queue

A typical priority queue supports the following operations:

1) Insertion : If the newly inserted item is of the highest priority, then it is inserted at the top. Otherwise, it is inserted in such a way that it is accessible after all higher priority items are accessed.

2) Deletion : We typically remove the highest priority item which is typically available at the top. Once we remove this item, we need not move the next priority item at the top.

3) Peek : This operation only returns the highest priority item (which is typically available at the top) and does not make any change to the priority queue.

GRAPHS

- Graph is an important non-linear data structure.
- A tree is, in fact, a special kind of graph structure.
- In tree structure, there is a hierarchical relationship between parent and children, that is, one parent and many children.
- On the other hand, in graphs, relationships are less restricted. Here, relationships are from many parents to many children.
- The below figure represents the two non-linear data structures.

Figure:

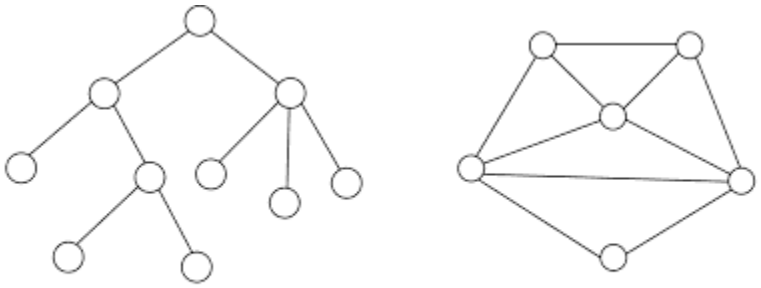


Figure Two non-linear data structures: tree and graph.

Formal definition of graph

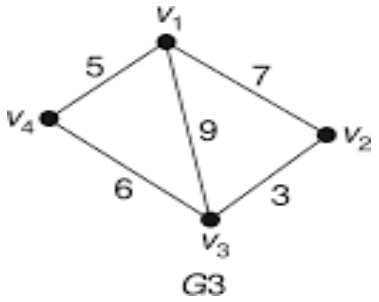
- A graph can be represented as **G=(V,E)**

Graph. A *graph* G consists of two sets:

(i) A set V, called the set of all vertices (or nodes)
(ii) A set E, called the set of all edges (or arcs). This set E is the set of all pairs of elements from V.

For example, let us consider the graph $G1$ in Figure . Here

$$V = \{v_1, v_2, v_3, v_4\}$$
$$E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_3, v_4)\}$$

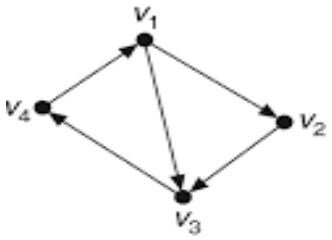


G3

Graph Terminologies

Digraph. A *digraph* is also called a *directed* graph. It is a graph G, such that, $G = <V, E>$, where V is the set of all vertices and E is the set of ordered pairs of elements from V. For example, graph $G2$ is a digraph, where

$$V = \{v_1, v_2, v_3, v_4\}$$
$$E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_3, v_4), (v_4, v_1)\}$$



G2

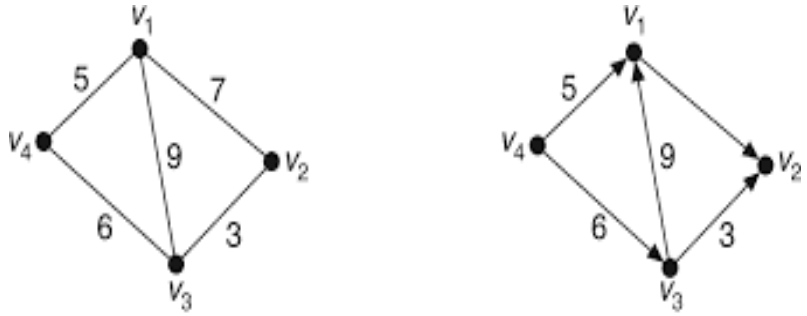
Here, if an order pair (v_i, v_j) is in E then there is an edge directed from v_i to v_j (indicated by the arrowhead).

Note that, in the case of an undirected graph (simple graph), the pair (v_i, v_j) is unordered, that is, (v_i, v_j) and (v_j, v_i) are the same edges, but in case of digraph they correspond to two different edges.

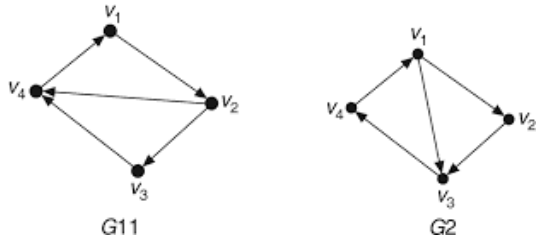
Here, if an order pair (v_i, v_j) is in E then there is an edge directed from v_i to v_j (indicated by the arrowhead).

Note that, in the case of an undirected graph (simple graph), the pair (v_i, v_j) is unordered, that is, (v_i, v_j) and (v_j, v_i) are the same edges, but in case of digraph they correspond to two different edges.

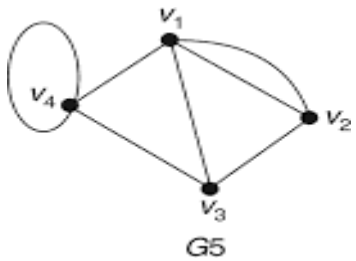
Weighted graph. A graph (or digraph) is termed *weighted* graph if all the edges in it are labelled with some weights. For example, $G3$ and $G4$ are two weighted graphs in Figure 8.3.



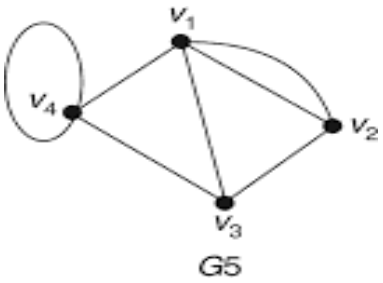
Adjacent vertices. A vertex v_i is *adjacent* to (or neighbour of) another vertex say, v_j, if there is an edge from v_i to v_j. For example, in $G11$ (Figure 8.3), v_2 is adjacent to v_3 and v_4, v_1 is not adjacent to v_4 but to v_2. Similarly the neighbours of v_3 in graph $G2$ are v_1 and v_2 (but not v_4).



Self loop. If there is an edge whose starting and end vertices are same, that is, (v_i, v_i) is an edge, then it is called a *self loop* (or simply, a loop). For example, the graph $G5$ (in Figure 8.3) has a self loop at vertex v_4.



Parallel edges. If there is more than one edge between the same pair of vertices, then they are known as the *parallel* edges. For example, there are two parallel edges between v_1 and v_2 in graph $G5$ of Figure 8.3. A graph which has either self loop or parallel edges or both, is called *multigraph*. In Figure 8.3, $G5$ is thus a multigraph.



G5

Simple graph (digraph). A graph (digraph) if it does not have any self loop or parallel edges is called a *simple graph* (digraph).

The following graphs G2,G6 and G9 are examples of simple graph since it does not contain any self-loop or parallel edges.

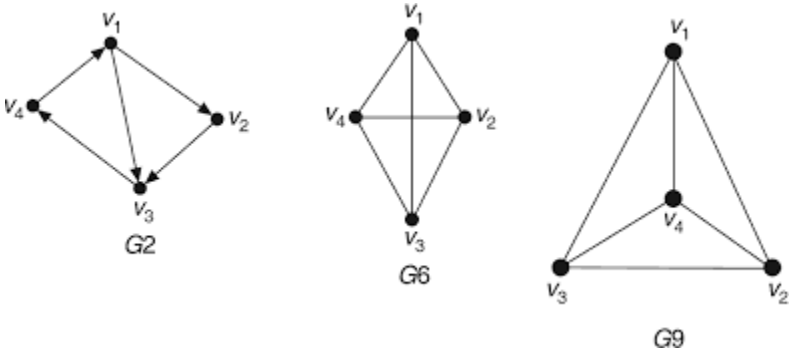


Figure: Examples of simple graphs

The below graphs G5 and G10 are not simple graphs. Here, the graph G5 contains both self loop and parallel edges, whereas graph G10 contains parallel edges.

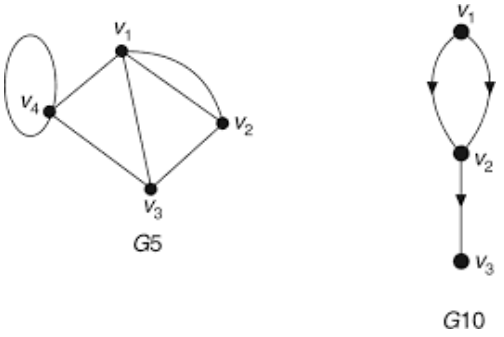
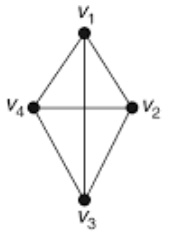
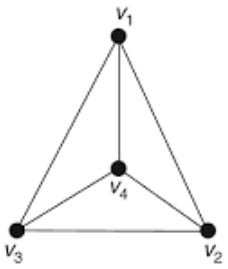


Figure: Examples of graphs which are not simple

Complete graph. A graph (digraph) G is said to be *complete* if each vertex v_i is adjacent to every other vertex v_j in G. In other words, there are edges from any vertex to all other vertices. For examples, $G6$ and $G9$ are two complete graphs.

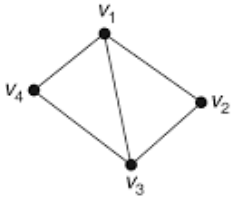


G6

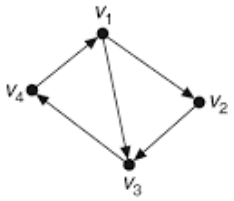


G9

Both G1 and G2 contain cycles.

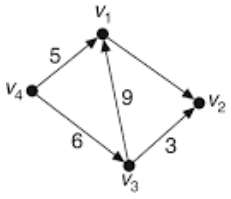


G1

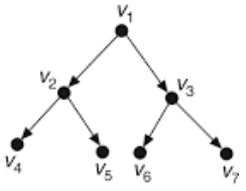


G2

G4 and G7 are two acyclic graphs.

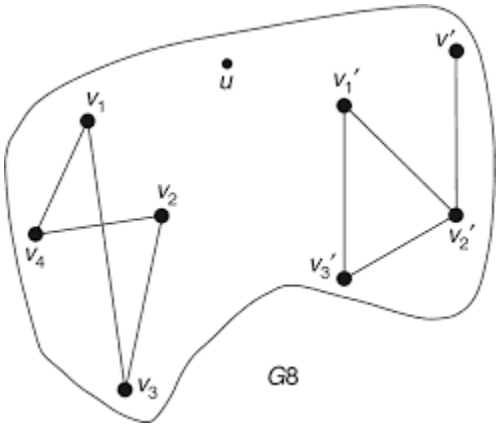


G4



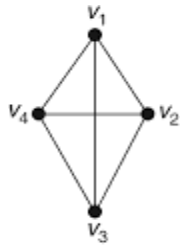
G7

Isolated vertex. A vertex is *isolated* if there is no edge connected from any other vertex to the vertex. For example, in $G8$ (Figure 8.3) the vertex u is an isolated vertex.



G8

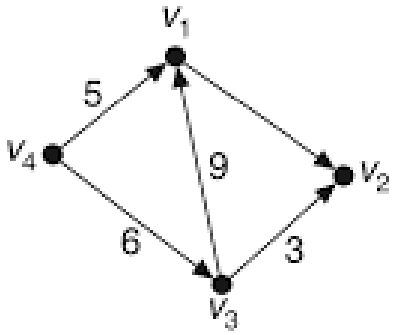
Degree of vertex. The number of edges connected with vertex v_i is called the *degree* of vertex v_i and is denoted by *degree*(v_i). For example, degree(v_i) = 3, $\forall\ v_i \in G6$ (Figure 8.3).



G6

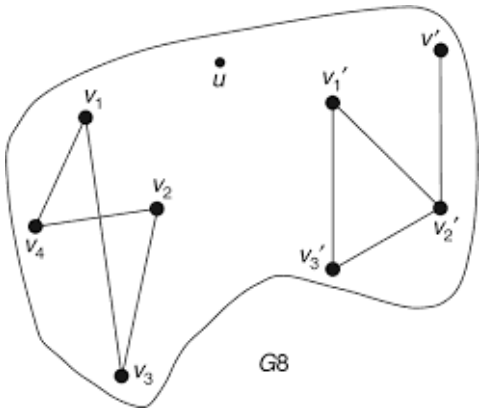
But for a digraph, there are two degrees: *indegree* and *outdegree*. Indegree of v_i denoted as indegree(v_i) = number of edges incident into v_i. Similarly, outdegree(v_i) = number of edges emanating from v_i. For example, let us consider the digraph *G*4. Here:

indegree (v_1) = 2,	outdegree (v_1) = 1
indegree (v_2) = 2	outdegree (v_2) = 0
indegree (v_3) = 1	outdegree (v_3) = 2
indegree (v_4) = 0	outdegree (v_4) = 2



G4

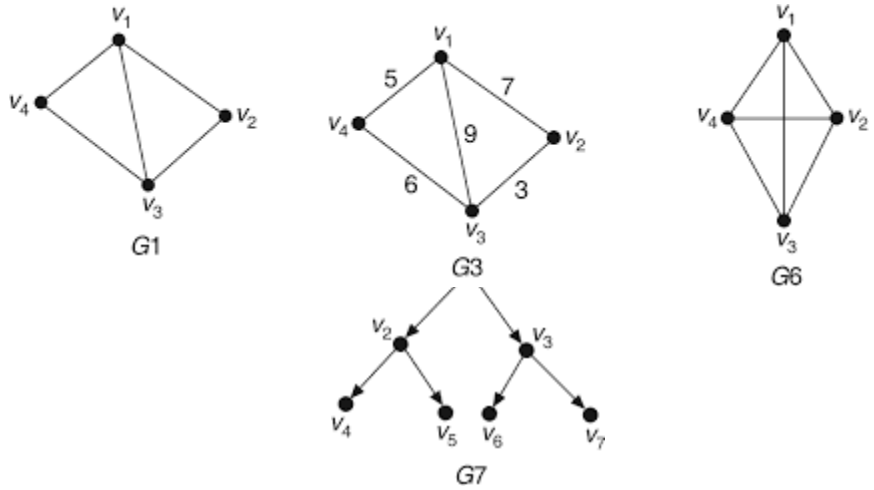
Pendant vertex. A vertex v_i is *pendant* if its indegree(v_i) = 1 and outdegree(v_i) = 0. For example, in *G*8 v' is a pendant vertex. In *G*7, there are four pendant vertices v_4, v_5, v_6 and v_7.



G8

Examples for connected graphs

G8 – Not connected



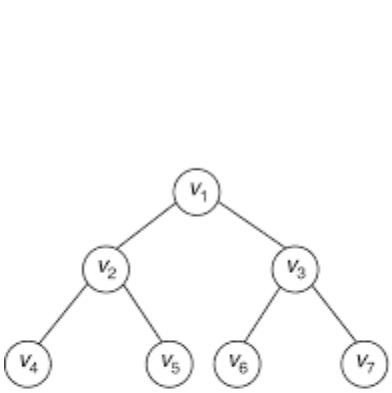
Connected graph. In a graph (not digraph) G, two vertices v_i and v_j are said to be *connected* if there is a path in G from v_i to v_j (or v_j to v_i). A graph is said to be connected if for every pair of distinct vertices v_i, v_j in G, there is a path. For example, $G1$, $G3$ and $G6$ are connected graphs but $G8$ is not (Figure 8.3).

Representation of Graphs

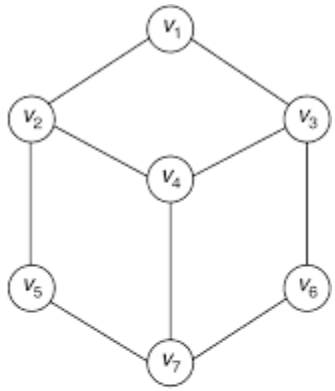
A graph can be represented in many ways. Some of the representations are:

1. Set representation
2. Linked representation
3. Sequential (matrix) representation

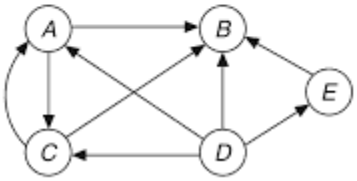
Consider the following graphs to be illustrated using the above representations.



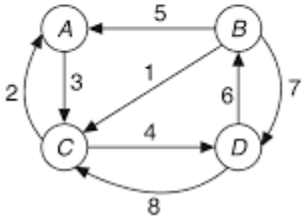
G1



G2



G3



G4

1. Set Representation

This is one of the straightforward methods of representing a graph. With this method, two sets are maintained: (i) V, the set of vertices, (ii) E, the set of edges, which is the subset of $V \times V$. But if the graph is weighted, the set E is the ordered collection of three tuples, that is, $E = W \times V \times V$, where W is the set of weights.

Let us see, how all the graphs given in Figure 8.5 can be represented with this technique.

Graph G1

representation and the most efficient one from the memory point of view, this method of representation is not useful so far as the manipulation of graph is concerned.

$E(G1) = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_2, v_5), (v_3, v_6), (v_3, v_7)\}$

Graph G2

$V(G2) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

$E(G2) = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_2, v_5), (v_3, v_4), (v_3, v_6), (v_4, v_7), (v_5, v_7), (v_6, v_7)\}$

Graph G3

$V(G3) = \{A, B, C, D, E\}$

$E(G3) = \{(A, B), (A, C), (C, B), (C, A), (D, A), (D, B), (D, C), (D, E), (E, B)\}$

Graph G4

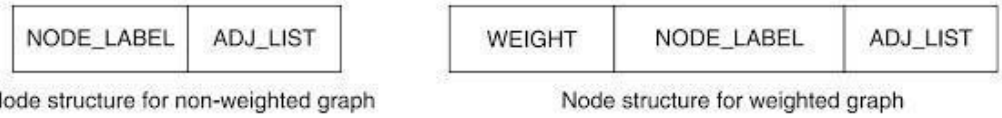
$V(G4) = \{A, B, C, D\}$

$E(G4) = \{(3, A, C), (5, B, A), (1, B, C), (7, B, D), (2, C, A), (4, C, D), (6, D, B), (8, D, C)\}$

Note that, if the graph is a multigraph and undirected, this method does not allow to store parallel edges, as in a set, two identical elements cannot exist. Although it is a straightforward

Linked Representation

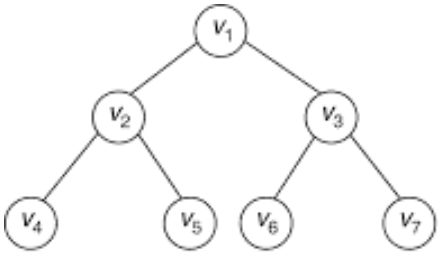
Linked representation is another space-saving way of graph representation. In this representation, two types of node structures are assumed as shown in Figure 8.6.



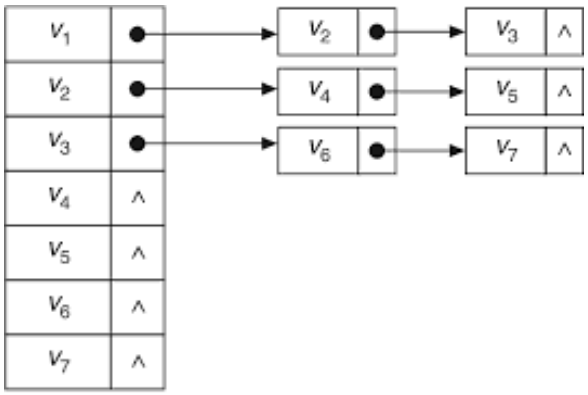
Node structure for non-weighted graph Node structure for weighted graph

Figure 8.6 Node structures in linked representation.

The linked list representation of graph G1 is as shown below.

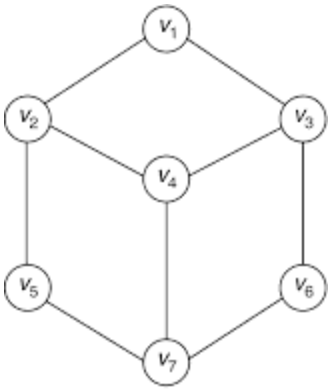


G1

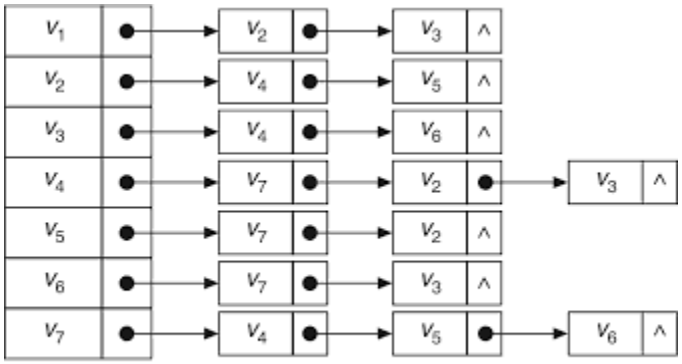


(a) Representation of graph *G1*

The linked list representation of graph G2 is as shown below.

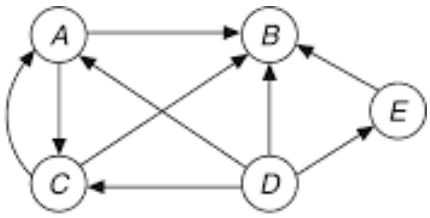


G2

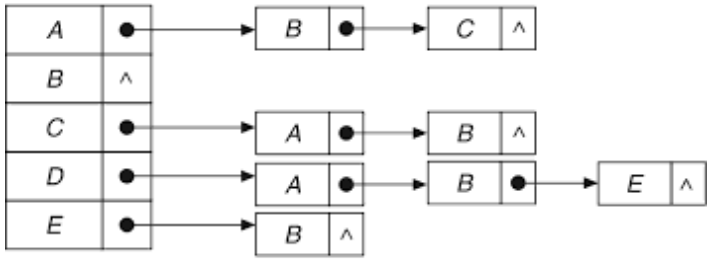


(b) Representation of graph *G2*

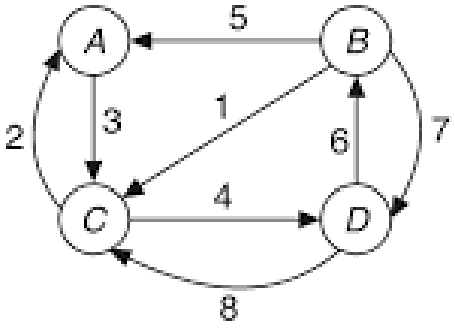
The linked representation of graph G3 is as shown below.



G3



(c) Representation of graph *G*3



*G*4

8.3.3 Matrix Representation

Matrix representation is the most useful way of representing any graph. This representation uses a square matrix of order $n \times n$, n being the number of vertices in the graph. A general representation is shown in Figure 8.8.

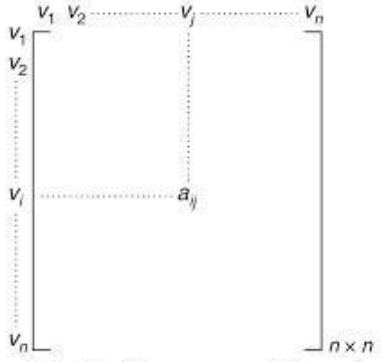
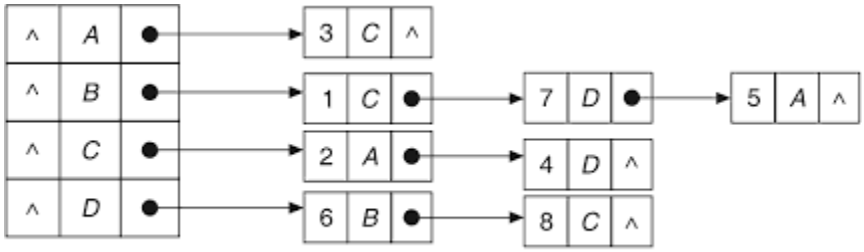


Figure 8.8 Matrix representation of graph.



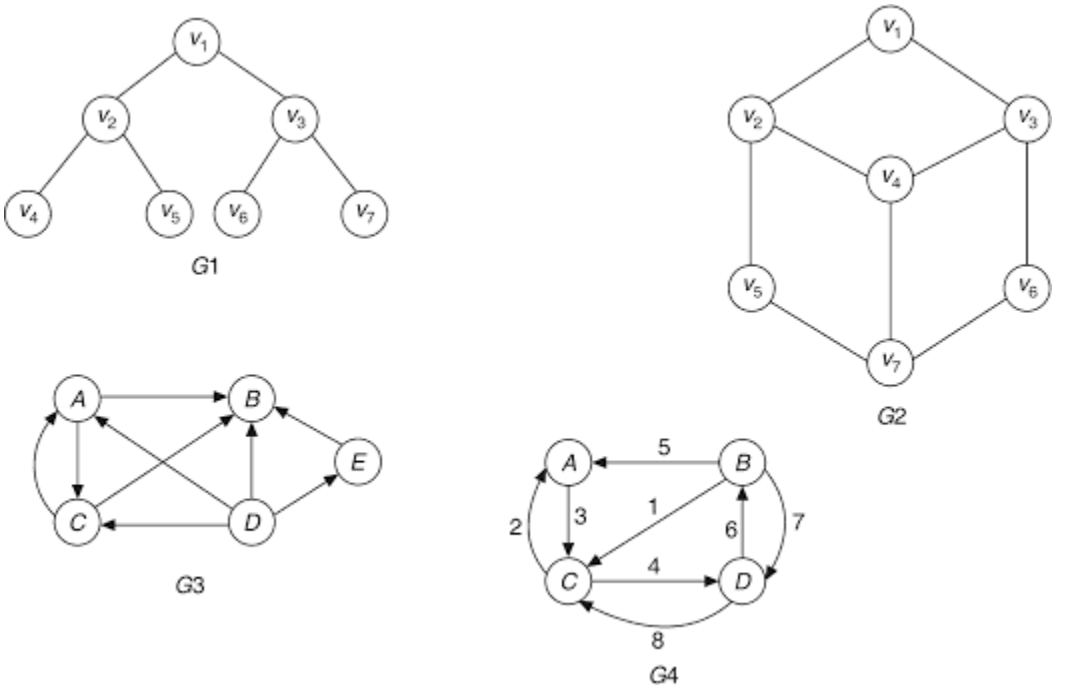
(d) Representation weighted digraph *G*4

Entries in the matrix can be decided as follows:

$$a_{ij} = 1, \text{ if there is an edge from } v_i \text{ to } v_j$$
$$= 0, \text{ otherwise}$$

This matrix is known as *adjacency* matrix because an entry stores the information whether two vertices are adjacent or not. Also, the matrix is alternatively termed *bit* matrix or *Boolean* matrix as the entries are either a 0 or a 1.

The adjacency matrix is useful to store multigraph as well as weighted graph. In case of multigraph representation, instead of entry 1, the entry will be the number of edges between two vertices. And in case of weighted graph, the entries are the weights of the edges between the vertices instead of 0 or 1. Figure 8.9 shows the adjacency matrix representation of graphs *G*1, *G*2, *G*3 and *G*4 given in Figure 8.5.



	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	0	1	1	0	0	0	0
v_2	1	0	0	1	1	0	0
v_3	1	0	0	0	0	1	1
v_4	0	1	0	0	0	0	0
v_5	0	1	0	0	0	0	0
v_6	0	0	1	0	0	0	0
v_7	0	0	1	0	0	0	0

*G*1

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	0	1	1	0	0	0	0
v_2	1	0	0	1	1	0	0
v_3	1	0	0	1	0	1	0
v_4	0	1	1	0	0	0	1
v_5	0	1	0	0	0	0	1
v_6	0	0	1	0	0	0	1
v_7	0	0	0	1	1	1	0

*G*2

	A	*B*	*C*	*D*	*E*
A	0	1	1	0	0
B	0	0	0	0	0
C	1	1	0	0	0
D	1	1	1	0	1
E	0	1	0	0	0

*G*3

	A	*B*	*C*	*D*
A	0	0	3	0
B	5	0	1	7
C	2	0	0	4
D	0	6	8	0

*G*4

Figure 8.9 Adjacency matrix representation of graphs given in Figure 8.5.

GRAPH TRAVERSAL

There are 2 types of traversals

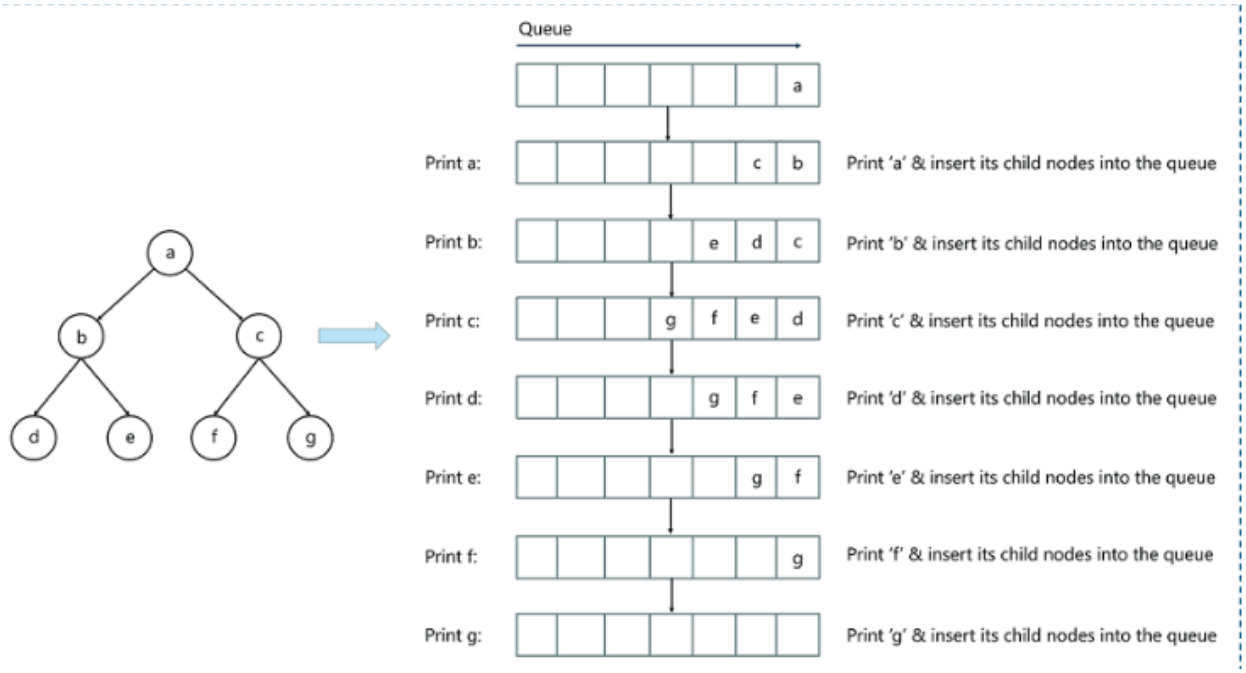
1. Breadth First search
2. Depth First Search

BREADTH-FIRST SEARCH

Here the data structure used is **QUEUE**

Algorithm

1. Initialize all the nodes as unvisited
2. Insert the first node/ starting node Vi to queue
3. Repeat 4 and 5 until queue is empty
4. Delete the node from Queue and mark it as visited
5. Insert the unvisited adjacent nodes of Vi to queue
6. Repeat until all the nodes are visited
7. Stop

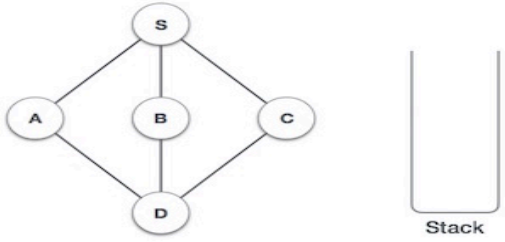
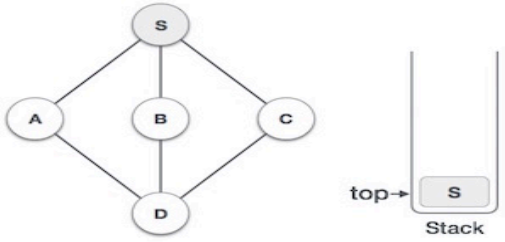


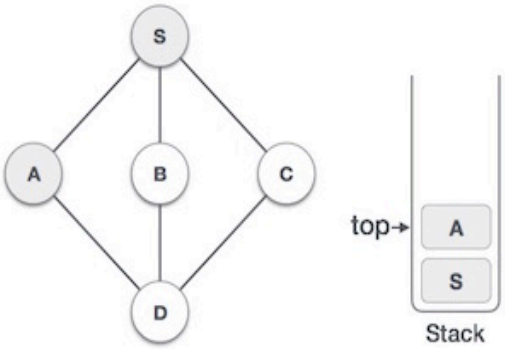
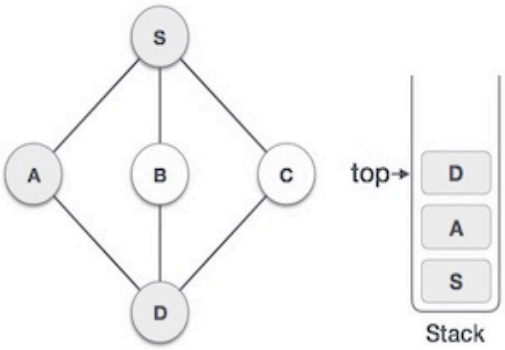
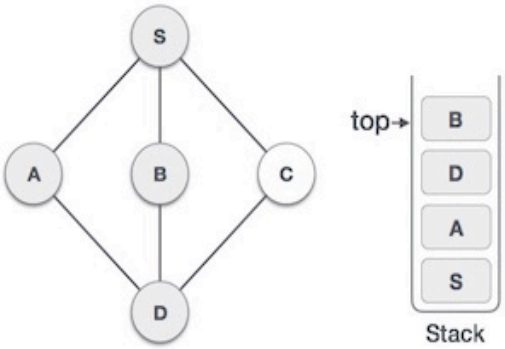
DEPTH-FIRST SEARCH

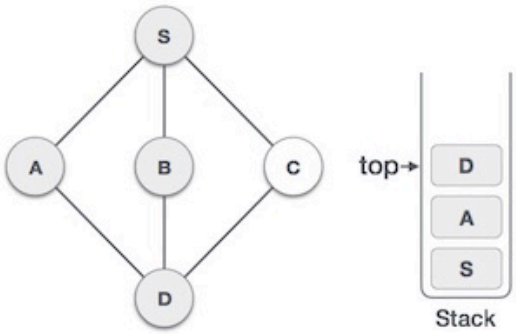
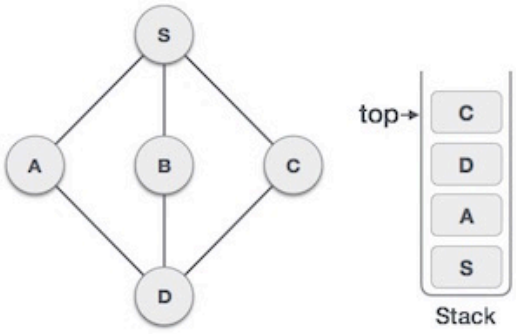
Here the data structure used is **STACK**

Algorithm

1. Initialize all the nodes as visited
2. PUSH the first node/ starting node Vi to stack
3. Repeat 4 and 5 until stack is empty
4. POP the top node of stack Vi and mark it as visited
5. PUSH the unvisited adjacent nodes of Vi to stack
6. Repeat until all the nodes are visited
7. Stop

Step	Traversal	Description
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

Applications of Graph Data Structure

Real-life application of graph data structure in various fields are:

- **Computer Science:-** Graphs are used to model many problems and solutions in computer science, such as representing networks, web pages, and social media connections. Graph algorithms are used in pathfinding, data compression, and scheduling.

- **Social Networks:-** Graphs represent and analyze social networks, such as the connections between individuals and groups.

- **Transportation:-** Graphs can be used to model transportation systems, such as roads and flights, and to find the shortest or quickest routes between locations.

- **Computer Vision:-** Graphs represent and analyze images and videos, such as tracking objects and detecting edges.

- **Natural Language Processing:-** Graphs can represent and analyze text, such as in syntactic and semantic dependency graphs.

- **Telecommunication:-** Graphs are used to model telecommunication networks, such as telephone and computer networks, and to analyze traffic and routing.

- **Circuit Design:-** Graphs are used in the design of electronic circuits, such as logic circuits and circuit diagrams.

- **Bioinformatics:-** Graphs model and analyze biological data, such as protein-protein interaction and genetic networks.

- **Operations research:-** Graphs are used to model and analyze complex systems in operations research, such as transportation systems, logistics networks, and supply chain management.

- **Artificial Intelligence:-** Graphs are used to model and analyze data in many AI applications, such as machine learning, Artificial Intelligence, and natural language processing.

Single Source All Destination

In graph theory, the "single-source all-destinations" problem involves finding the shortest path from a single starting node (the source) to every other node in the graph. This means determining the most efficient route for travel from a specific origin to all other possible destinations within the graph.

Definition:

- Source: A designated starting node in the graph.
- Destinations: All other nodes in the graph that are reachable from the source.
- Shortest Path: The path with the minimum total weight (or cost) connecting the source to each destination.

Algorithms:

Dijkstra's Algorithm:
A widely used algorithm that finds the shortest paths from a source node to all other reachable nodes in a graph where edge weights are non-negative. It works by iteratively expanding a "frontier" of nodes, starting from the source and exploring neighbors until the shortest path to each destination is found.

Bellman-Ford Algorithm:
An algorithm that can handle graphs with negative edge weights, but is generally slower than Dijkstra's for graphs without negative cycles. It also finds shortest paths from a source to all other nodes.

MODULE 4

Sorting and Searching

Sorting Techniques :- Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Radix Sort. **Searching Techniques** :- Linear Search, Binary Search, **Hashing -** Hashing functions : Mid square, Division, Folding, Digit Analysis; **Collision Resolution** : Linear probing, Quadratic Probing, Double hashing, Open hashing.

SORTING TECHNIQUES

- Sorting techniques are fundamental algorithms in computer science and data structures.
- They are used to arrange data in a specific order, typically in ascending or descending order, making it easier to search for and retrieve information efficiently.
- There are various sorting techniques, each with its own advantages and disadvantages.

What is Sorting?

Sorting is the process of arranging a collection of data elements in a specific order, typically in ascending or descending order.

Sorting Techniques

Here are some of the most commonly used sorting techniques:

1. Bubble Sort: Bubble sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

2. Selection Sort: Selection sort divides the input list into two parts: a sorted sublist and an unsorted sublist. It repeatedly selects the minimum element from the unsorted sublist and moves it to the sorted sublist.

3. Insertion Sort: Insertion sort builds the final sorted array one item at a time. It takes one element from the input data and inserts it into its correct position within the sorted array.

4. Quick Sort: Quick sort is a divide-and-conquer sorting algorithm that works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

5. Merge Sort: Merge sort is another divide-and-conquer algorithm. It divides the unsorted list into n sub-lists, each containing one element, and then repeatedly merges sub-lists to produce new sorted sub-lists until there is only one sub-list remaining.

6. Heap Sort: Heap sort is based on the data structure called a binary heap. It involves building a binary heap from the input data and repeatedly extracting the maximum element from the heap until the heap is empty.

7. Counting Sort: Counting sort is a non-comparison-based sorting algorithm. It works well for integers or objects with a small, finite range of values. It counts the number of occurrences of each element and uses this information to place elements in their correct positions.

8. Radix Sort: Radix sort is another non-comparison-based sorting algorithm that works well for integers. It sorts data by processing individual digits, starting from the least significant digit to the most significant digit.

9. Bucket Sort: Bucket sort divides the input data into a fixed number of buckets, and each bucket is sorted individually. After sorting, the buckets are concatenated to obtain the final sorted list.

10. Tim Sort: Tim sort is a hybrid sorting algorithm that combines elements of merge sort and insertion sort. It is designed for sorting real-world data and takes advantage of partially ordered data.

BUBBLE SORT

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.

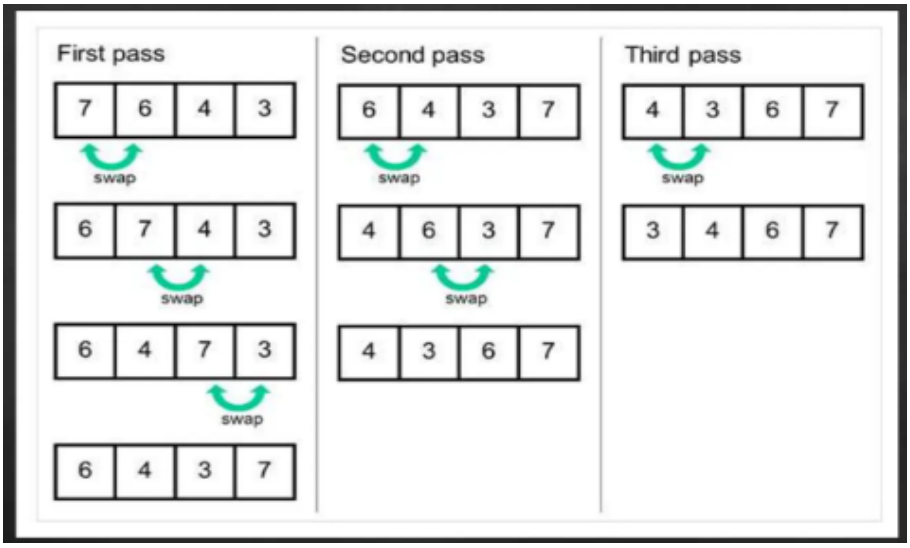
Why is it named as a bubble?

- It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.
- At each step, if two adjacent elements of a list are not in order, they will be swapped. Thus, larger elements will "bubble" to the end, (or smaller elements will be "bubbled" to the front, depending on implementation) and hence the name Bubble sort
- Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order.
- Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world.
- The bubble sort, also known as the ripple sort, is one of the least efficient sorting algorithms.

➔ Bubble short is majorly used where -
  ➔ Complexity does not matter
  ➔ Simple and shortcode is preferred

➔ Working of Bubble sort
  ➔ Compare the first two values and swap if necessary.
  ➔ Then compare the next pair of values and swap if necessary.
  ➔ This process is repeated n-1 times, where n is the number of values being sorted.

Example



Algorithm

In the algorithm given below, suppose arr is an array of n elements. The assumed swap function in the algorithm will swap the values of given array elements.

```
begin BubbleSort(arr)
    for all array elements
        if arr[i] > arr[i+1]
                swap(arr[i], arr[i+1])
        end if
    end for
    return arr
end BubbleSort
```

Efficiency of Bubble Sort

The bubble sort algorithm is a reliable sorting algorithm. This algorithm has a worst-case time complexity of O(n2). The bubble sort has a space complexity of O(1). The number of swaps in

bubble sort equals the number of inversion pairs in the given array. When the array elements are few and the array is nearly sorted, bubble sort is effective and efficient.

Advantages	**Disadvantages**
The primary advantage of the bubble sort is that it is popular and easy to implement	The main disadvantage of the bubble sort is the fact that it does not deal well with a list containing a huge number of items.
In the bubble sort,elements are swapped in place without using additional temporary storage.	The bubble sort requires n-squared processing steps for every n number of elements to be sorted.
The space requirement is at a minimum	The bubble sort is mostly suitable for academic teaching but not for real- life applications.

INSERTION SORT

What Is Insertion Sort?

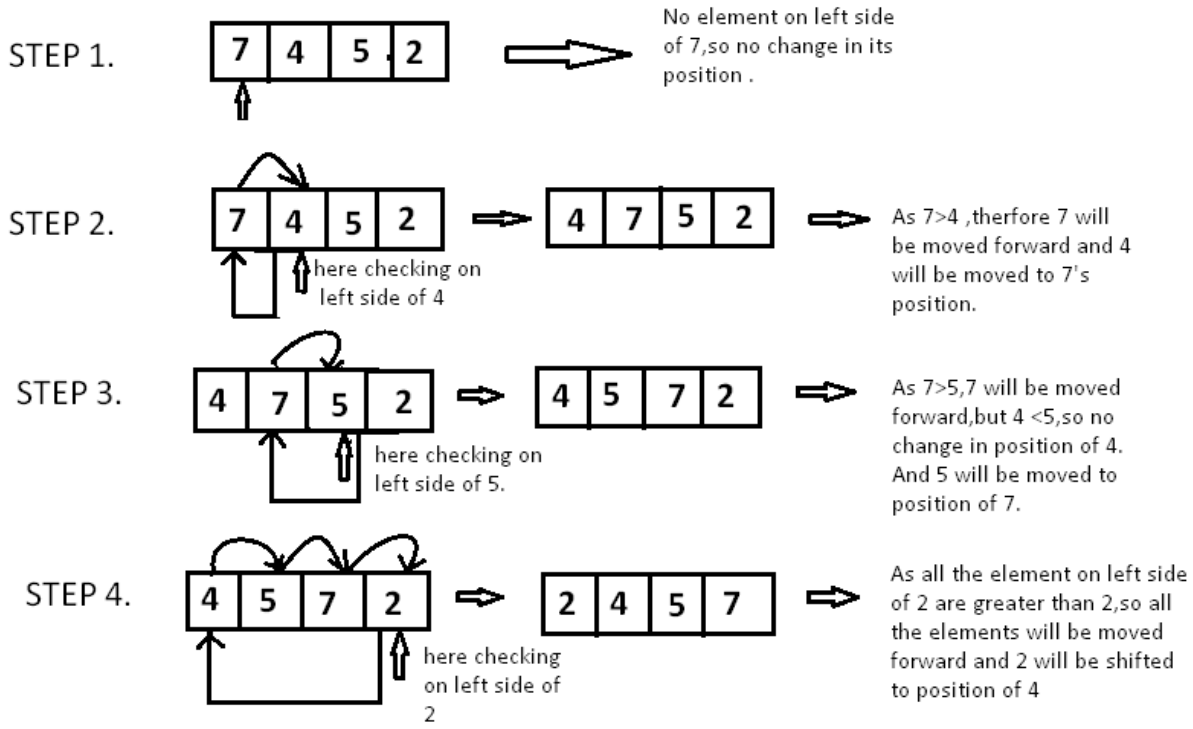
Insertion sort is a simple and intuitive sorting algorithm in which we build the sorted array using one element at a time. It is easy to implement and is quite efficient for small sets of data, especially for substantially sorted data. It is flexible — it is useful in scenarios where all the array elements are not available in the beginning.

You may have used this sorting method unknowingly at some point in your life. Insertion sort is also known as "card player" sort. If you've ever played a game of cards, chances are you have used some form of insertion sort to arrange the cards in your hand in a strategic

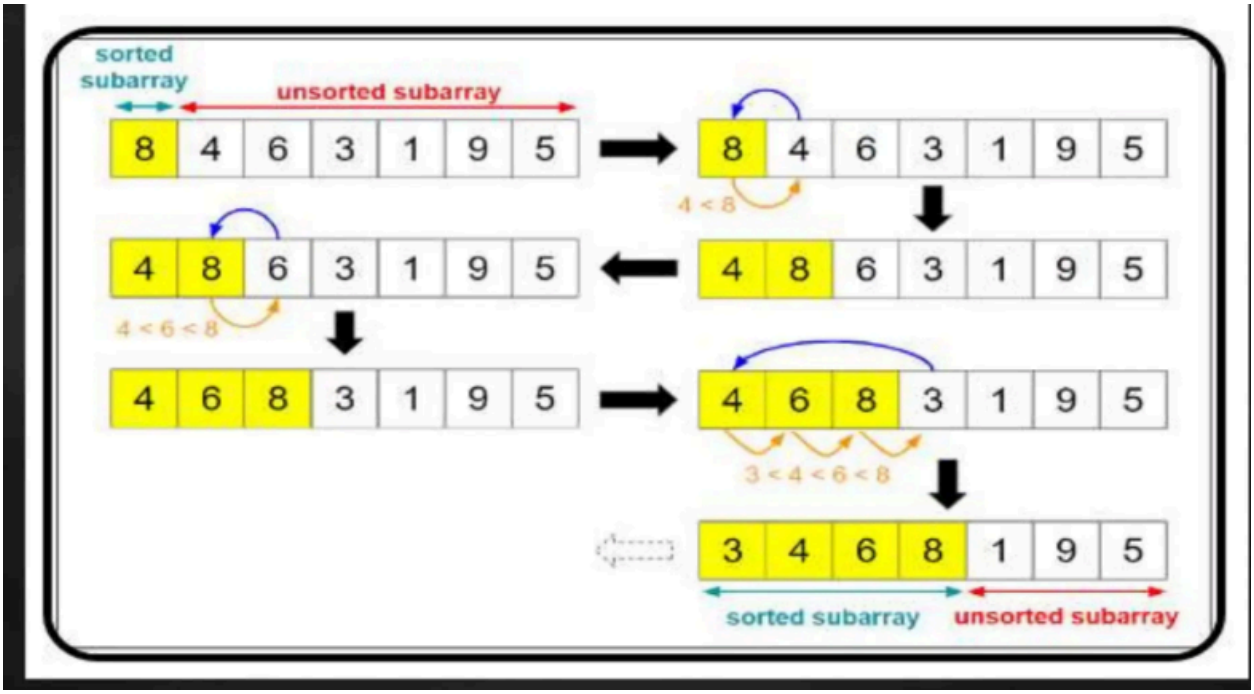
How Insertion Sort Works

1. If it is the first element, it is already sorted.
2. Pick the next element.
3. Compare with all the elements in the sorted sub-list.

4. Shift all the elements in the sorted sub-list that are greater than the value to be sorted.
5. Insert the value.
6. Repeat until the list is sorted.



Since 7 is the first element and has no other element to be compared with, it remains at its position. Now when on moving towards 4, 7 is the largest element in the sorted list and greater than 4. So, move 4 to its correct position i.e. before 7. Similarly with 5, as 7 (largest element in the sorted list) is greater than 5, we will move 5 to its correct position. Finally for 2, all the elements on the left side of 2 (sorted list) are moved one position forward as all are greater than 2 and then 2 is placed in the first position. Finally, the given array will result in a sorted array.



Insertion Sort Algorithm

```
insertionSort(array)
 mark first element as sorted
 for each unsorted element X
  'extract' the element X
  for j <- lastSortedIndex down to 0
   if current element j > X
    move sorted element to the right by 1
  break loop and insert X here
end insertionSort
```

According to the algorithm,

1. Start with the second element in the array (the first one is considered sorted).

2. Pick that element (call it X) and compare it with the elements before it (the sorted part).
3. Shift all elements that are bigger than X one position to the right.
4. Insert X in the correct position in the sorted part.
5. Repeat this for every remaining element in the array.

Insertion Sort Complexity

1. **Time Complexity**

Cases	Complexity
Best	O(n)
Average	O(n^2)
Worst	O(n^2)

2. **Space Complexity**

Insertion sort requires O(1) additional space, making it a space-efficient sorting algorithm.

Applications of Insertion Sort

- The list is small or nearly sorted.
- Simplicity and stability are important.
- Used as a subroutine in Bucket Sort
- Can be useful when array is already almost sorted (very few inversions)
- Since Insertion sort is suitable for small sized arrays, it is used in Hybrid Sorting algorithms along with other efficient algorithms like Quick Sort and Merge Sort. When the subarray size becomes small, we switch to insertion

sort in these recursive algorithms. For example IntroSort and TimSort use insertion sort.

Advantages	Disadvantages
The main advantage of the instruction sort is its simplicity	The disadvantage of the insertion sort is that it does not perform as well as other, better sorting algorithms.
It also exhibits a good performance when dealing with a small list.	With n-squared steps required for every element to be sorted, the insertion sort does not deal well with a huge list.
The insertion sort is an in-place sorting algorithm so the space requirement is minimal.	The insertion sort is particularly useful only when sorting a list of a few items.

SELECTION SORT

What Is Selection Sort?

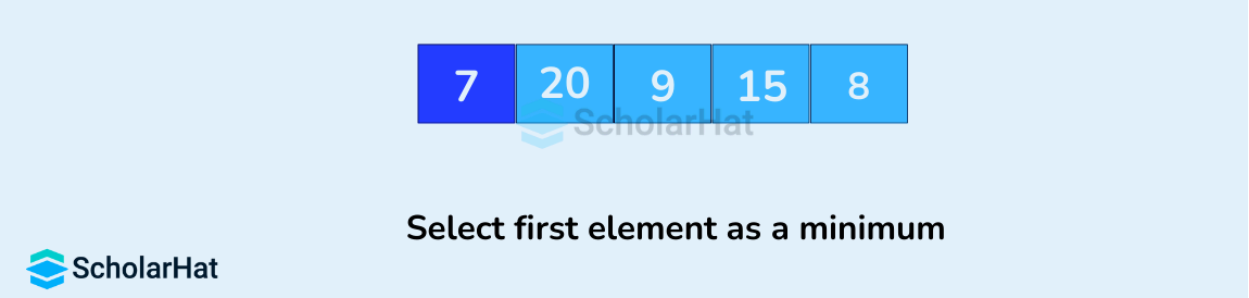
Selection sort is an in-place comparison-based sorting algorithm. This sorting algorithm is known for its simplicity and memory efficiency; it doesn't take up any extra space. The selection sort method repeatedly searches "remaining items" to find the least one and moves it to its final location.

Selection Sort Working Principle

- This algorithm divides the input array into two subparts — the sorted part and the unsorted part. Initially, the sorted part of the array is empty, and the unsorted part is the input array.
- The algorithm works on the principle of finding the lowest number from the unsorted part of the array and then swapping it with the first element of the unsorted part. This is done over and over until the entire array becomes sorted (in ascending order).

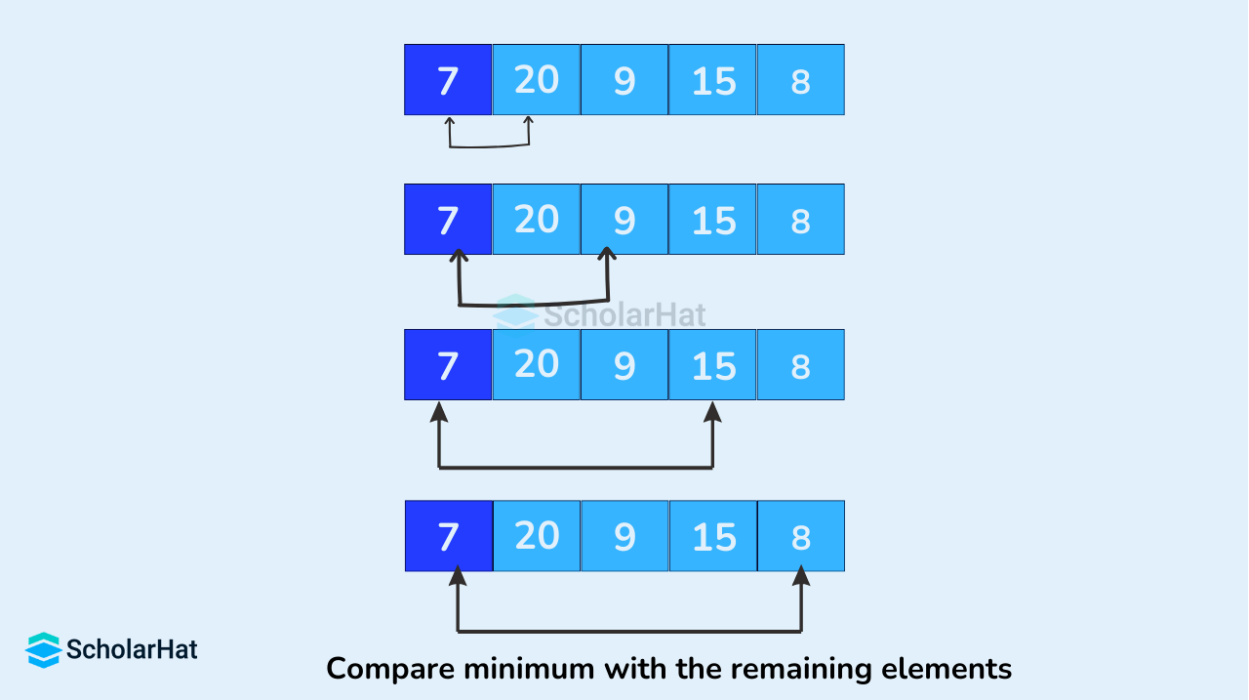
Working of Selection Sort Example

1. Set the first element as minimum. 7, 20, 9, 15, 8

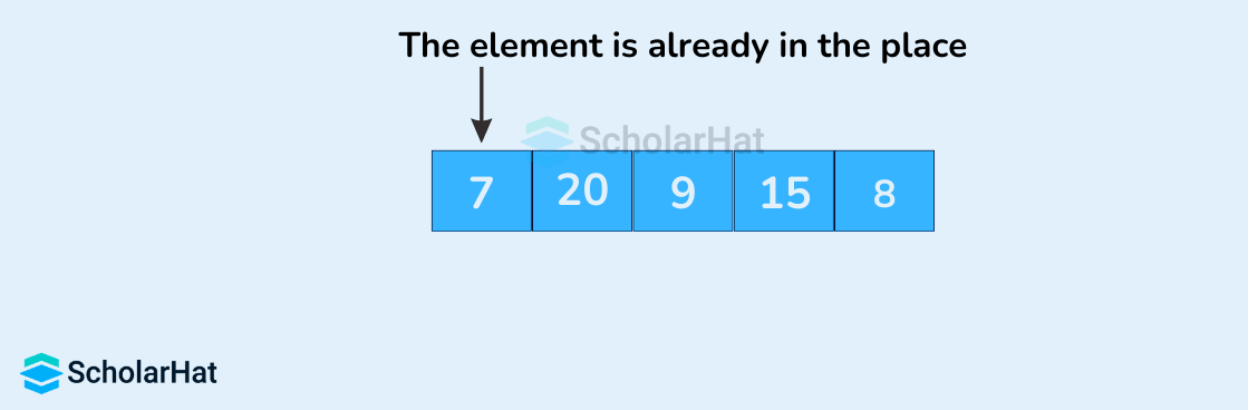


2. Compare the minimum with the second element. If the second element is smaller than the minimum, assign the second element as minimum.

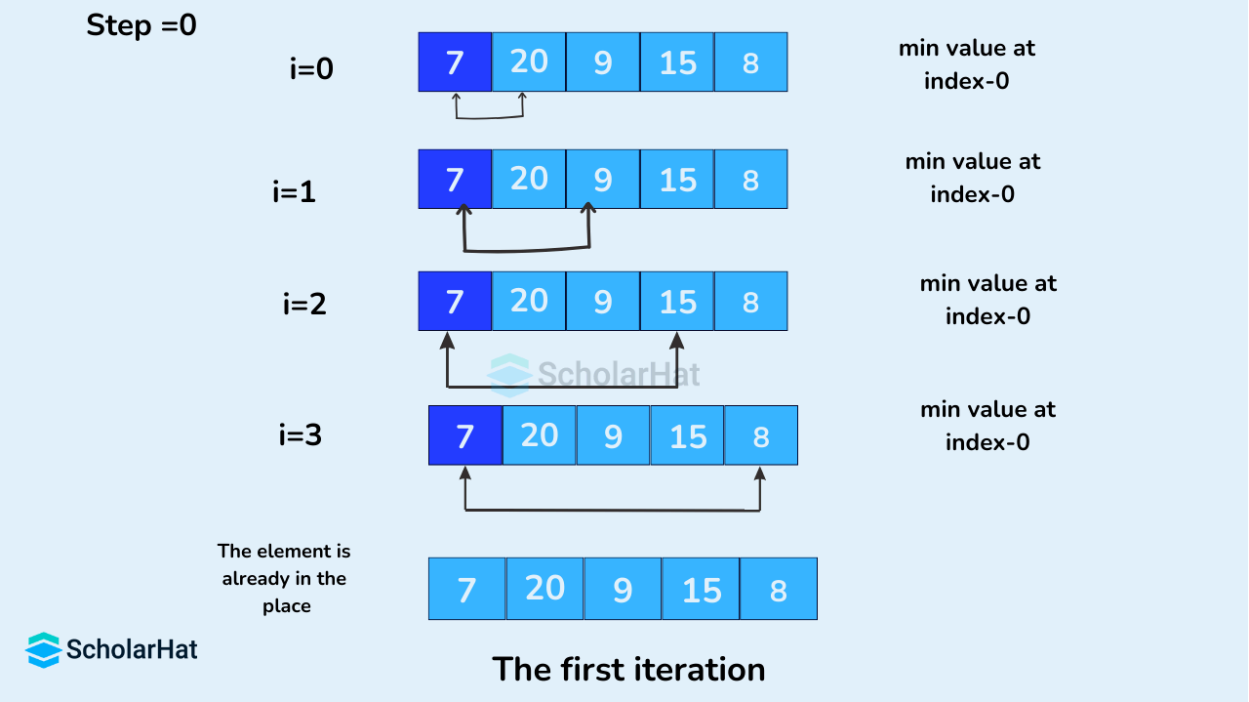
- Compare minimum with the third element. Again, if the third element is smaller, assign minimumit to the third element otherwise do nothing. The process goes on until the last element.

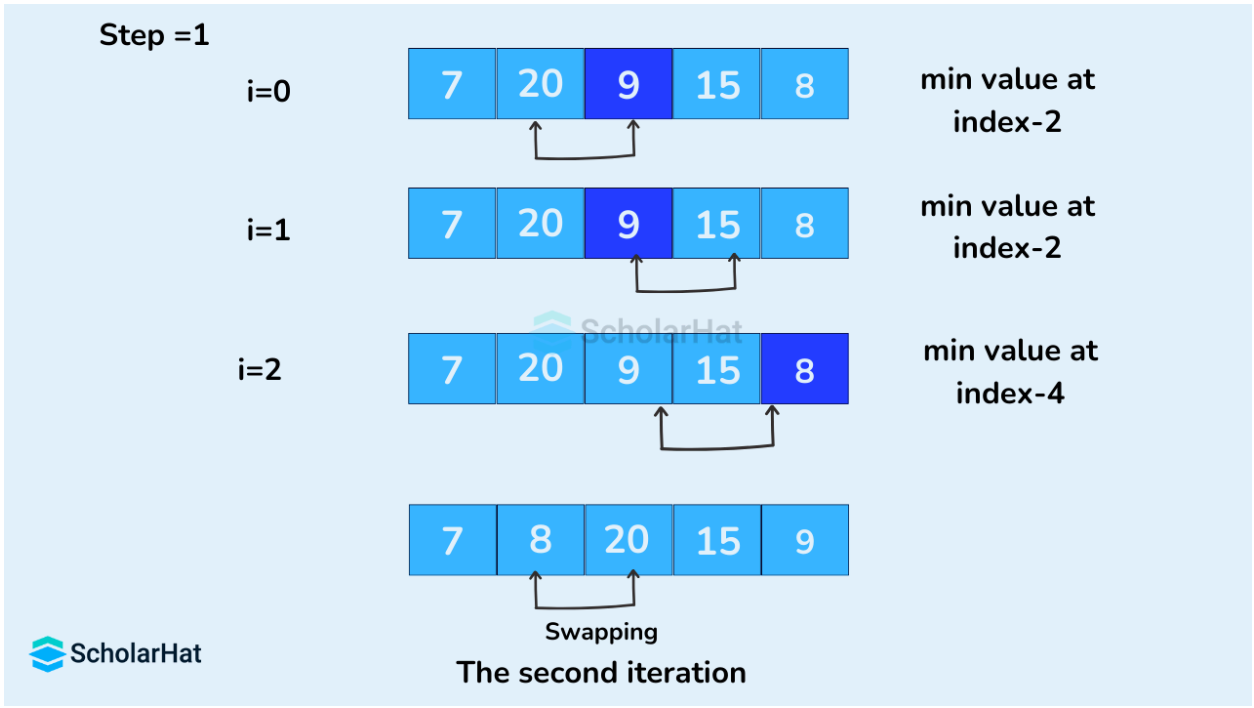


3. After each iteration, the minimum is placed in the front of the unsorted list.

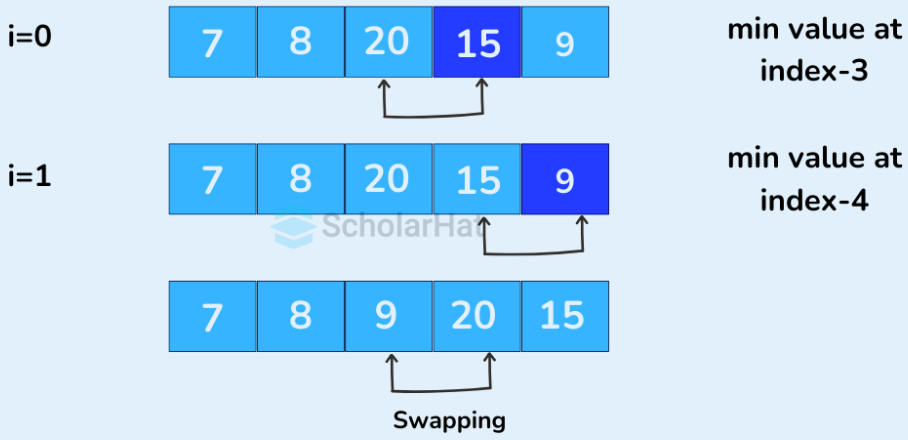


4. For each iteration, indexing starts from the first unsorted element. Steps 1 to 3 are repeated until all the elements are placed in their correct positions.



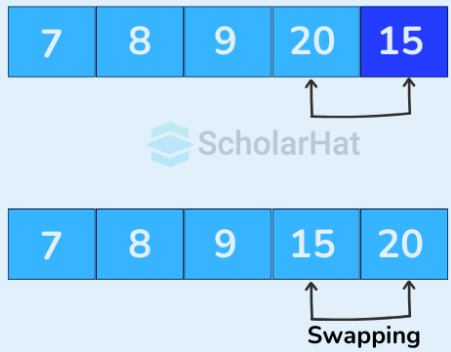

Step =1
i=0
7 20 9 15 8
min value at index-2
i=1
7 20 9 15 8
min value at index-2
i=2
7 20 9 15 8
min value at index-4
7 8 20 15 9
Swapping
ScholarHat
The second iteration

Step =2



The third iteration

Step =3



The fourth iteration

Selection Sort Algorithm

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

According to the algorithm,

6. Obtain the list or array to be sorted.
7. Set the current index as the starting index. This represents the beginning of the unsorted portion of the list.
8. Traverse the rest of the unsorted portion to find the smallest element.
9. Swap the smallest element found with the element at the current index.
10. Increment the current index by one, moving the boundary of the sorted and unsorted portions of the list.
11. Check if the current index is less than the length of the list - 1. If it is, go back to the 3rd step. If it's not, the list is now sorted in ascending order.
12. End the process.

Selection Sort Complexity

1) **Time Complexity**

The time complexity of the selection sort algorithm is O(n^2).

Let's look at the total number of comparisons made to get a better idea:

- **Worst-case complexity: O(n^2)**

  When the input array is sorted in descending order. In this case, n-1 swaps are performed

- **Best-case complexity: O(n^2)**

  When the array is already sorted (ascending order) In this case, no swap operation is performed

- **Average-case complexity: O(n^2)**

  When the array is neither sorted in ascending nor in descending order

2) **Space Complexity**

Selection sort's space complexity is O(1), as it doesn't take any extra space.

Applications of Selection Sort Algorithm

1. **Small Data Sets:** Suitable for sorting **very small lists or arrays**, where the overhead of more complex algorithms like Merge Sort or Quick Sort is unnecessary.
2. Embedded Systems: Selection sort is also used in embedded systems, where memory and processing power are limited. It is a simple algorithm that can be implemented in a small amount of code, making it a good choice for embedded systems.
3. Testing Other Sorting Algorithms: Selection sort is often used to test other sorting algorithms. Since selection sort is one of the simplest sorting algorithms, it is easy

to implement and can be used as a benchmark to compare the performance of more complex sorting algorithms.

4. Partially Sorted Data: Selection sort performs well on partially sorted data. If a large portion of the input data is already sorted, selection sort will not perform unnecessary swaps, resulting in faster sorting times
5. When Swapping is Costly: If the cost of swapping elements is high (for instance, when dealing with large records and small keys), selection sort can be efficient because it minimizes the number of swaps.

Advantages	Disadvantages
The main advantage of the selection sort is that it performs well on a small list.	The primary disadvantage of the selection sort is its poor efficiency when dealing with a huge list of items.
Because it is an in-place sorting algorithm, no additional temporary storage is required beyond what is needed to hold the original list.	The selection sort requires an n-squared number of steps for sorting n elements.
Its performance is easily influenced by the initial ordering of the items before the sorting process.	Quick sort is much more efficient than selection sort.

QUICK SORT

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Divide and conquer approach:

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

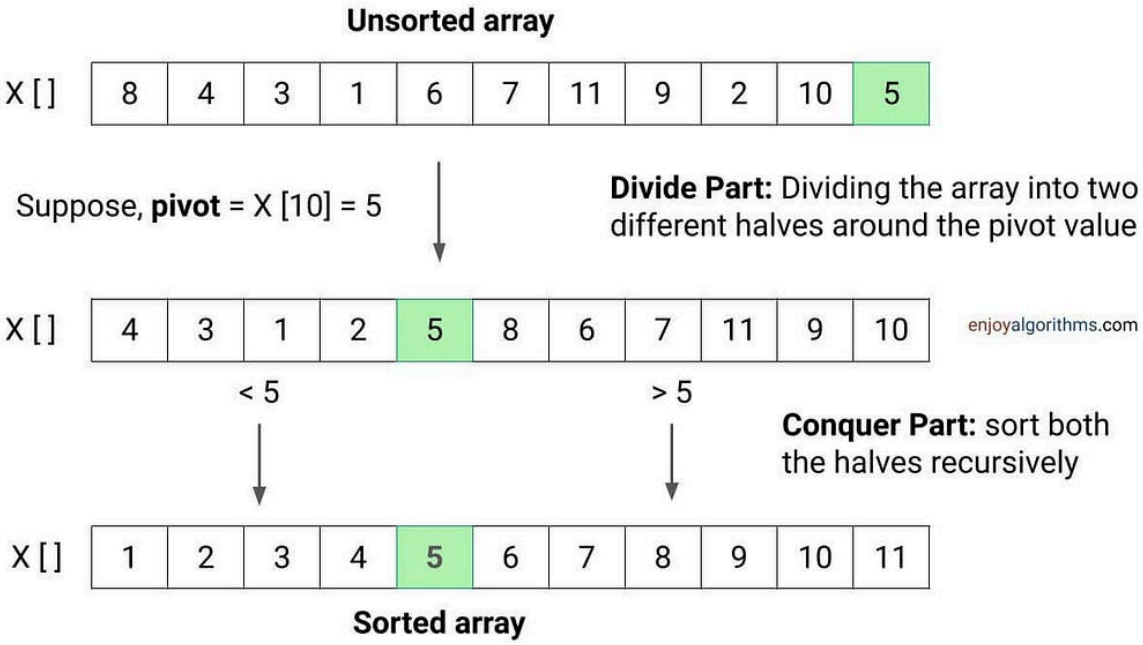
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.
Conquer: Recursively, sort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

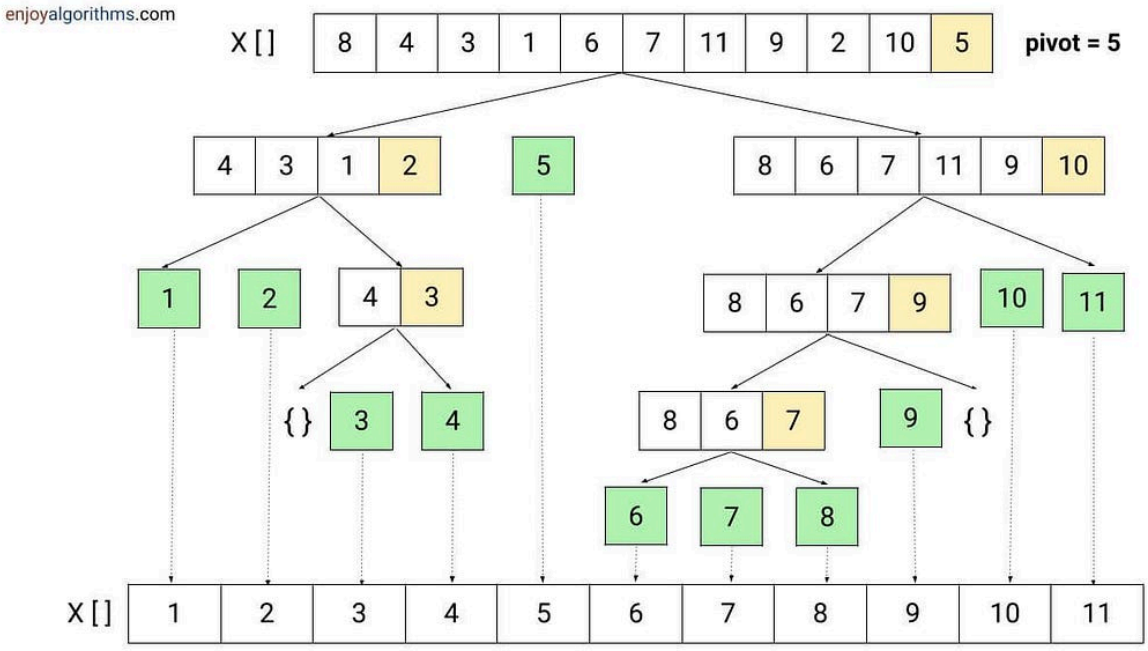
Working of Quick Sort

- The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.
- Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

Choice of Pivot:

- There are many different choices for picking pivots.
- Always pick the first element as a pivot.
- Always pick the last element as a pivot
- Pick a random element as a pivot.
- Pick the middle as the pivot.





Quick Sort Algorithm

Step 1: An array is divided into subarrays by selecting a pivot element (element selected from the array).

Step 2: While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

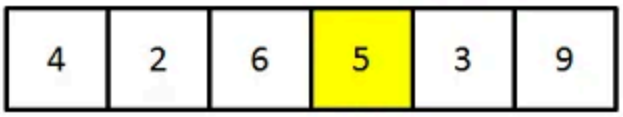
Step 3: The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.

Step 4: At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

```
function quickSort(array, low, high)
    if low < high
    pivotIndex = partition(array, low, high)
    quickSort(array, low, pivotIndex - 1)  // Before pivot
    quickSort(array, pivotIndex + 1, high) // After pivot
```

Example

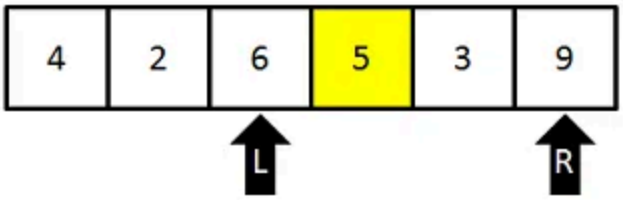
Step 1
Determine pivot



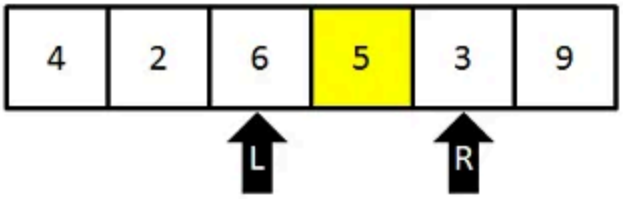
Step 2
Start pointers at left and right

Step 3
Since 4 < 5, shift left pointer

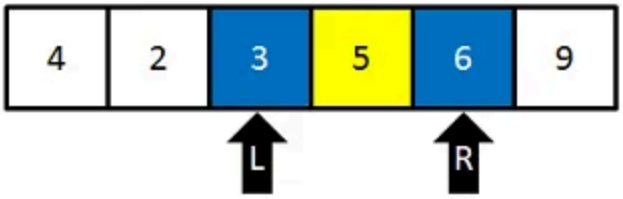
Step 4
Since 2 < 5, shift left pointer
Since 6 > 5, stop



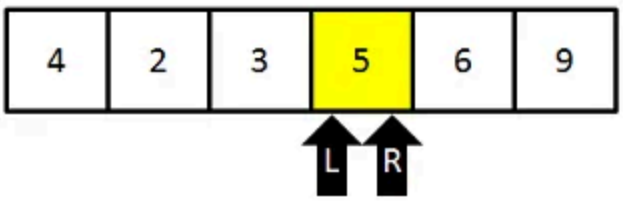
Step 5
Since 9 > 5, shift right pointer
Since 3 < 5, stop



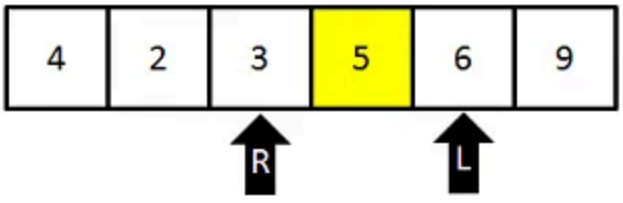
Step 6
Swap values at pointers



Step 7
Move pointers one more step



Step 8
Since 5 == 5,
move pointers one more step
Stop



Quick Sort Complexity

1. **Time complexity**

Case	Time complexity
Best	O(nlogn)
Average	O(nlogn)
Worst	O(n^2)

2. **Space complexity**

   In the worst case, the space complexity is O(n) because here, n recursive calls are made. And, the average space complexity of a quick sort algorithm is equal to O(logn).

Application of Quick Sort

1. Numerical analysis:For accuracy in calculations most of the efficiently developed algorithm uses a priority queue and quick sort is used for sorting.
2. Call Optimization: It is tail-recursive and hence all the call optimization can be done.
3. Database management: Quicksort is the fastest algorithm so it is widely used as a better way of searching.

Application of Quick Sort

1. Sorting large datasets in computer programs, because QuickSort is fast and memory-efficient.
2. Used in programming libraries like Python's sort() (internally uses Timsort, but based on QuickSort ideas) and C/C++ standard libraries.
3. Database systems use QuickSort to quickly sort query results before displaying them.
4. In file systems, for sorting names or files alphabetically or by size.

5. Used in competitive programming because of its high speed and efficiency on average.

Advantages	Disadvantages
Quick sort is very fast on average and works efficiently even for large lists of data.	Quick sort can be slow in the worst case especially if the pivot is always the smallest or largest element.
It uses very little extra memory because it sorts the list in the same place without needing another array.	It may not be stable, which means the original order of equal elements might not be preserved.
It handles big data well and usually performs better than other sorting methods like bubble sort or insertion sort.	It uses recursion, so if the list is too large then it can cause a stack overflow error.

HEAP SORT

What is a heap?

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

What is heap sort?

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

• Heapsort is the in-place sorting algorithm.

Working of Heap Sort

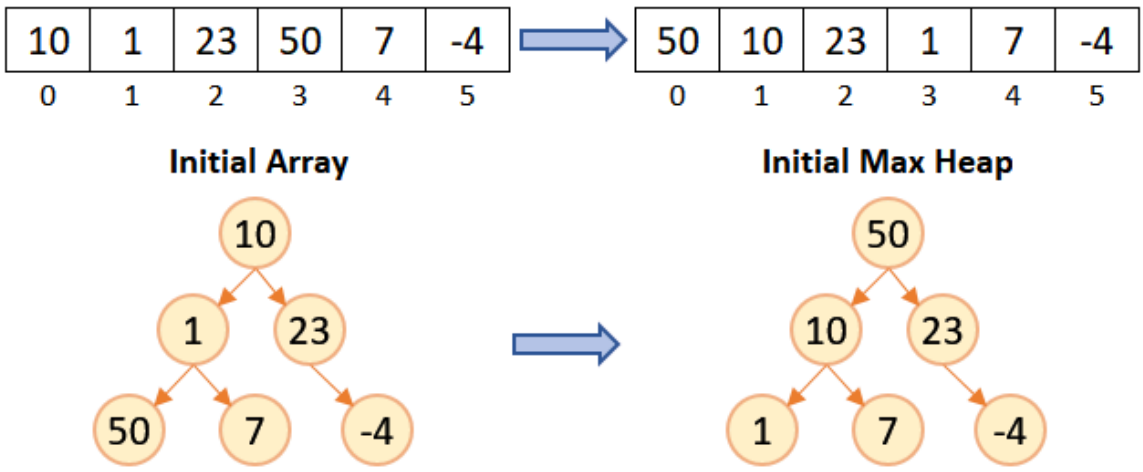
- It involves building a max heap for increasing order sorting which contains largest element of the heap at the root. For decreasing order sorting min heap is used which contains the smallest element of the heap at the root. The process step of for increasing order sorting of heap sort is summarized below:

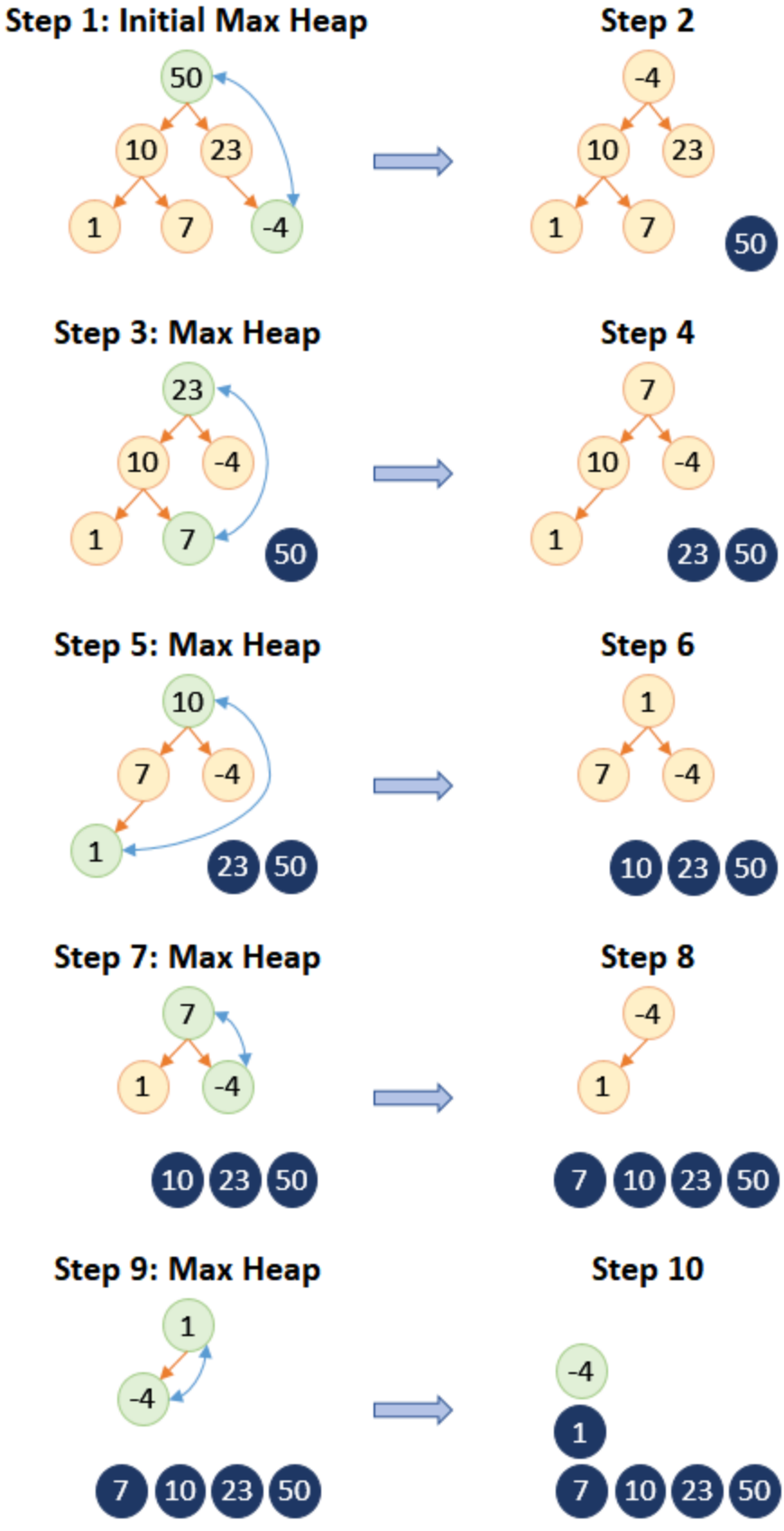
  Step 1: Build a max heap which contains largest element of the heap at the root

  Step 2: Swap the last element of the heap with the root element and remove the last element from the heap. With the remaining elements repeat step 1.

Example

- To understand the heap sort, let's consider an unsorted array [10, 1, 23, 50, 7, -4]
- In the below figure, the heap structure of input array and max heap is shown. The index number of the root element of the heap is O. In max heap, the largest element of the heap always resides at the root.




Step 1: Initial Max Heap
50
10
23
1
7
-4
Step 2
-4
10
23
1
7
50
Step 3: Max Heap
23
10
-4
1
7
50
Step 4
7
10
-4
1
23
50
Step 5: Max Heap
10
7
-4
1
23
50
Step 6
1
7
-4
10
23
50
Step 7: Max Heap
7
1
-4
10
23
50
Step 8
-4
1
7
10
23
50
Step 9: Max Heap
1
-4
7
10
23
50
Step 10
-4
1
7
10
23
50

Heap Sort Complexity

1. **Time Complexity:**

   The time complexity of creating a heap is O(N) and time complexity of creating a max heap is O(logN) and overall time complexity of heap sort is O(NlogN).

2. **Space Complexity:**
   The space complexity of heap sort is O(1).

Application of Heap Sort Algorithm

1. Sorting Large Data Sets: Efficiently handles large arrays or lists due to its O(n log n) time complexity.
2. Priority Queues: Used to implement priority queues where elements are processed based on priority.
3. Real-Time Systems: Suitable for real-time systems that require guaranteed time complexity.
4. Heapsort for External Sorting: Useful in external sorting algorithms where data is too large to fit into memory.
5. Graph Algorithms: Applied in algorithms like Dijkstra's shortest path and Prim's minimum spanning tree, which utilize heap structures.

Advantages	Disadvantages
The heap sort algorithm is widely used because of its efficiency.	Heap sort requires more space for sorting.
The heap sort algorithm can be implemented as an in-place sorting algorithm.	Quick sort is much more efficient than heap in many cases
Its memory usage is minimal.	Heap sor make a tree of sorting elements

MERGE SORT

Merge Sort is a popular and efficient comparison-based sorting algorithm. It is a divide-and-conquer algorithm that divides the input list into smaller sublists, sorts those sublists, and then merges the sorted sublists to produce a fully sorted list. The key idea is to repeatedly

divide the list in half until it is broken down into smaller, sortable pieces and then combine (merge) those pieces to produce the final sorted result.

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

Algorithm

Merge Sort is a divide-and-conquer algorithm for sorting the first elements. Here are the high-level steps for the Merge Sort algorithm:

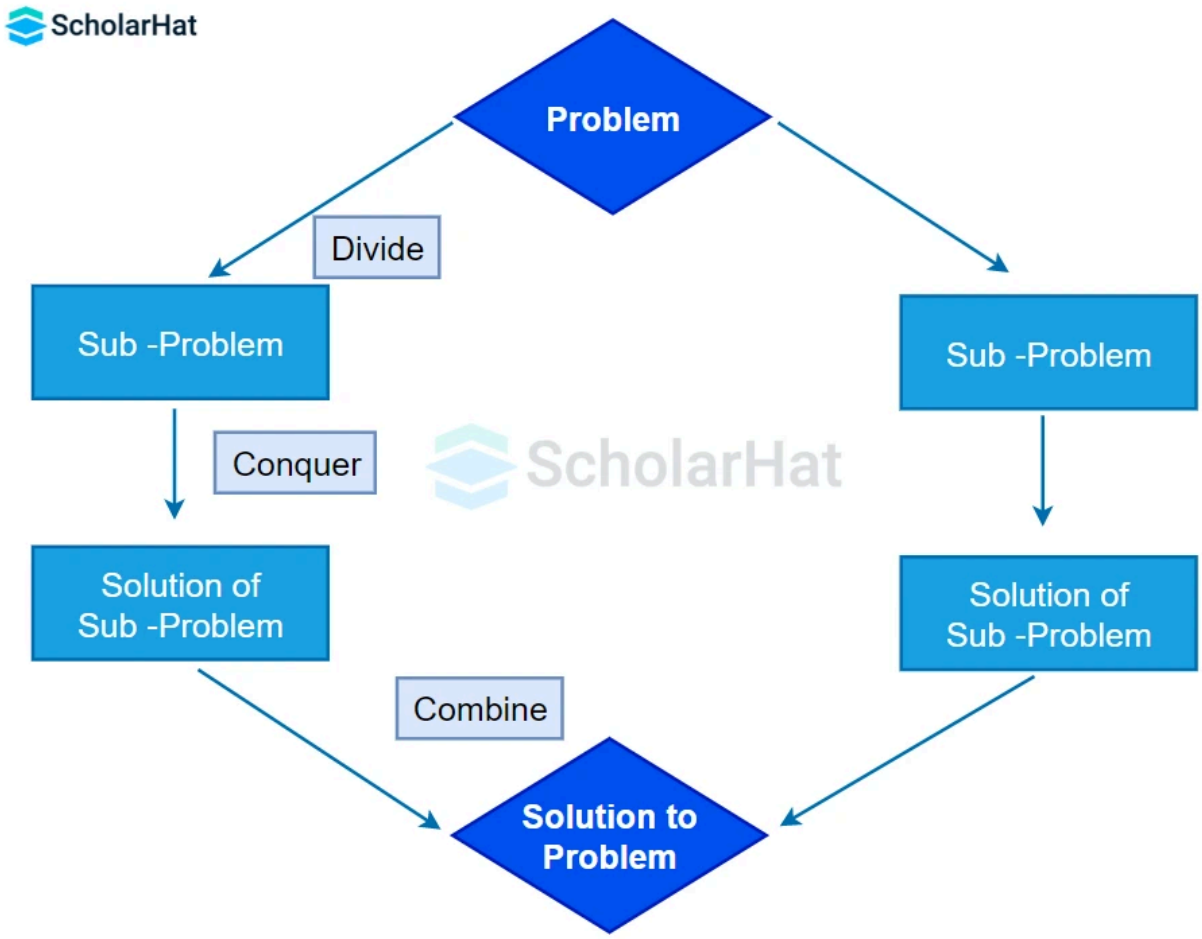
• **Divide:** Divide the unsorted list into two roughly equal halves. This can be done by finding the middle index of the list.

• **Conquer:** Recursively sort the two smaller sublists created in the previous step. This process continues until the sublists are small enough to be considered sorted by definition (i.e., they contain only one element).

• **Merge:** Merge the two sorted sublists to produce a single, sorted list. This is the most critical step in the Merge Sort algorithm.

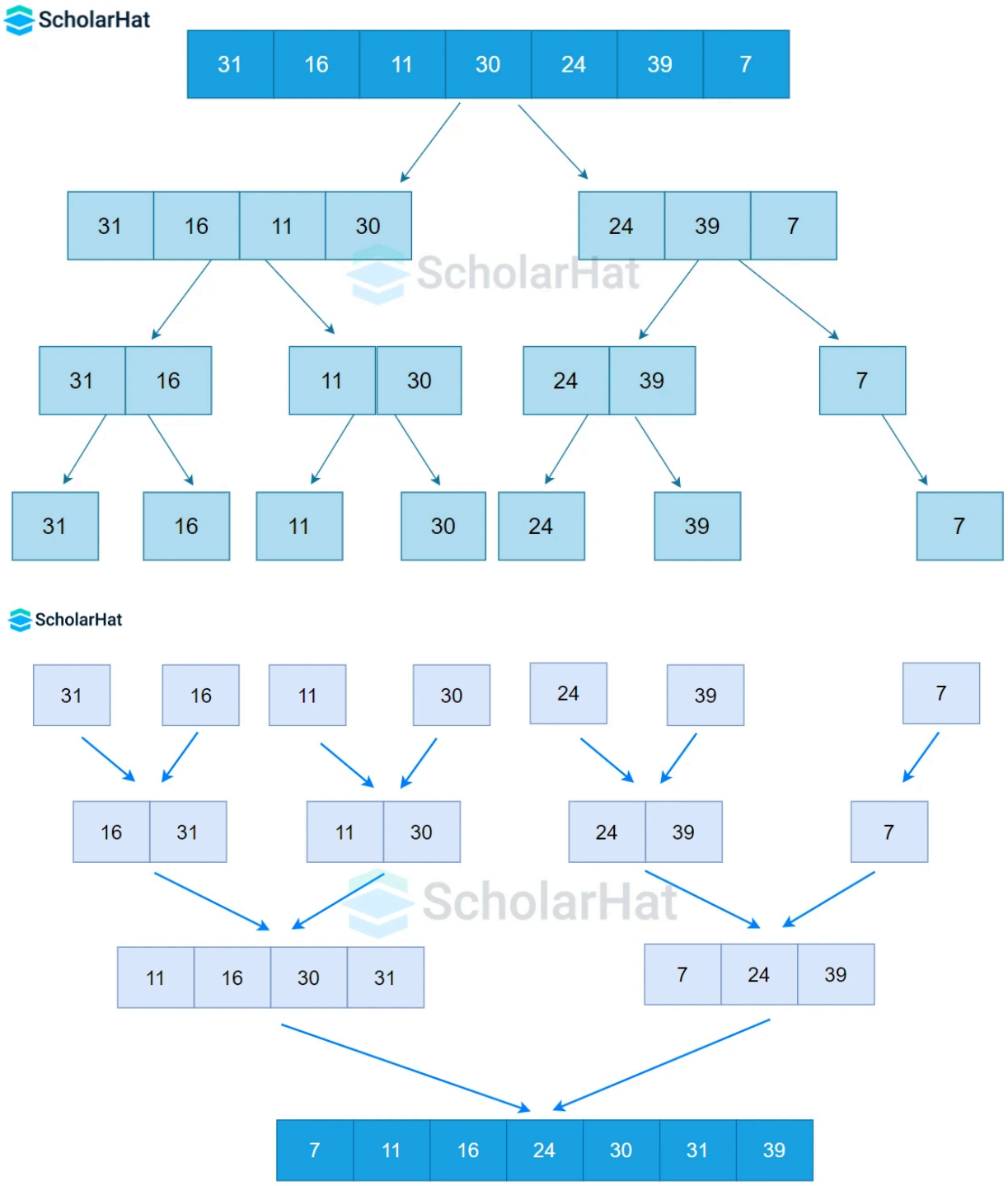
a. Create two temporary arrays (or lists) to hold the two sublists.

b. Initialize three index variables: one for each of the two sublists and one for the main merged list.




```
MergeSort(A, p, r):
   if p > r
      return
   mid = (p+r)/2
   mergeSort(A, p, mid)
   mergeSort(A, mid+1, r)
   merge(A, p, mid, r)
```

Example of Merge sort



Applications of Merge Sort:

- **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
- **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
- **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.

Advantages	Disadvantages
It can be applied to files of any size	Requires extra space N
Reading of the input during the run-creation step is sequential ⇒ not much seeking	Merge sort requires more space than other sort
If heap sort is used for the in-memory part of the merge, its operation can be overlapped with I/O	Merge sort is less efficient than other sort.

RADIX SORT

What is Radix?

- In radix sorting, the term "radix" refers to the base of the numeral system being used to represent the numbers you are sorting. It is essentially the number of unique digits or symbols that can be used to represent values in that numeral system. Commonly used radices include base-2 (binary), base-10 (decimal), and base-16 (hexadecimal).
- Example: In decimal radix sorting (base-10), there are 10 unique digits (0-9) that can be used to represent numbers. Each digit can be considered a "radix."

What is Radix Sort?

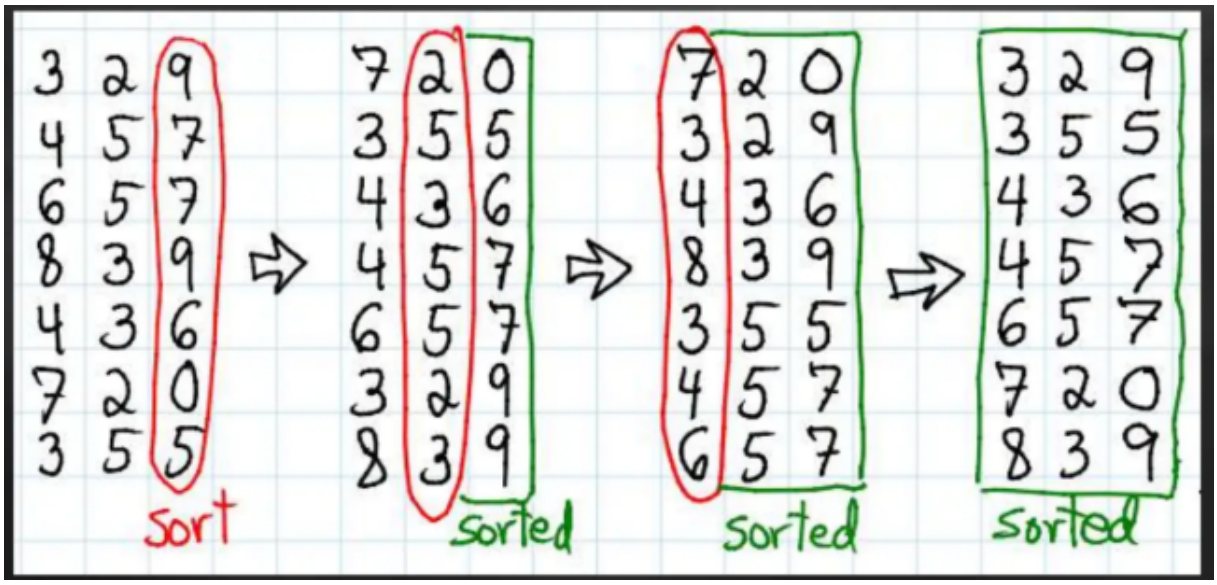
Radix sort is a non-comparative integer sorting algorithm that works by distributing elements* into a set of "buckets" or "bins" based on their individual digits or radix (e.g., decimal

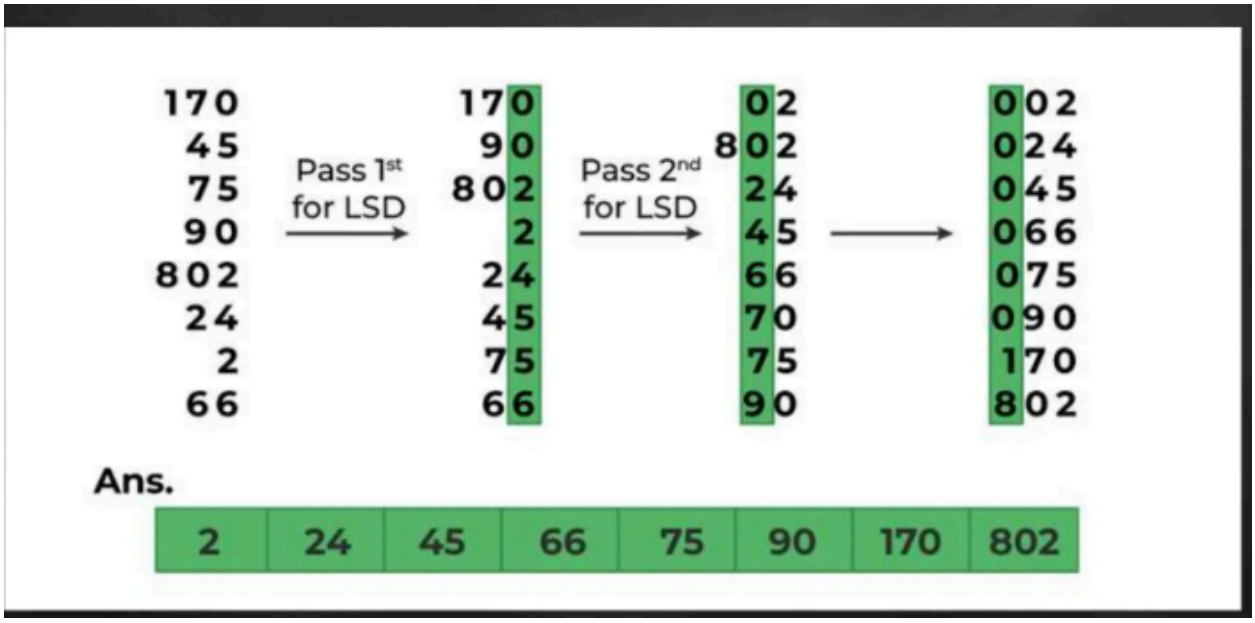
digits in the case of base-10 numbers). The algorithm processes the elements from the least significant digit (LSB) to the most significant digit (MSB) or vice versa, performing multiple passes until all digits have been considered.

Working of Radix Sort

- The Radix sort algorithm works by ordering each digit from least significant to most significant.
- In base 10, radix sort would sort by the digits in the one's place, then the ten's place, and so on.
- To sort the values in each digit place, Radix sort employs counting sort as a subroutine.
- This means that for a three-digit number in base 10, counting sort will be used to sort the 1st, 10th, and 100th places, resulting in a completely sorted list. Here's a rundown of the counting sort algorithm.

EXAMPLE





What Is a Radix Sort Algorithm?

- Radix Sort is a linear sorting algorithm.
- Radix Sort's time complexity of O(nd), where n is the size of the array and d is the number of digits in the largest number.
- It is not an in-place sorting algorithm because it requires extra space.
- Radix Sort is a stable sort because it maintains the relative order of elements with equal values.
- Radix sort algorithm may be slower than other sorting algorithms such as merge sort and Quicksort if the operations are inefficient. These operations include sub-inset lists and delete functions, and the process of isolating the desired digits.
- Because it is based on digits or letters, radix sort is less flexible than other sorts. If the type of data changes, the Radix sort must be rewritten.

Application of Radix Sort

Radix sort is a stable sorting algorithm, which means that it preserves the relative order of equal elements. It is particularly efficient when sorting a large number of integers with a limited range

of values. However, it may not be the best choice for sorting data with variable-length keys or data that doesn't have a clear radix (e.g., strings or floating-point numbers).

Advantages	Disadvantages
Fast when the keys are short, i.e. when array element range is small	The radix sort needs to be rewritten for each of the data types as it involves sorting which is based on letters or digits
Used in suffix arrays construction algorithms such as Manber's and the DC3 algorithm	As compared to other sorting algorithms used in data structures, the constant of the radix sort is greater in value.
Radix sort is a stable sort because it maintains the relative order elements with equal values.	Quick sort makes the use of in-place storing and hence it requires more space for radix sort algorithms as compared to quick sort.

SEARCHING TECHNIQUES

- Searching is the process of finding a particular item in a collection of items.
- A search typically answers whether the item is present in the collection or not.
- Searching requires a key field such as name, ID, code which is related to the target item.
- When the key field of a target item is found, a pointer to the target item is returned.
- The pointer may be an address, an index into a vector or array, or some other indication of where to find the target.
- If a matching key field isn't found, the user is informed.

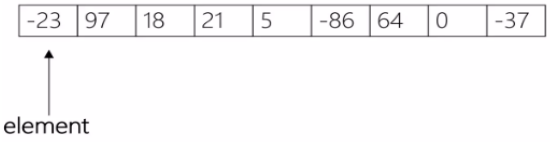
LINEAR SEARCH

- Linear Search Algorithm is the simplest search algorithm.

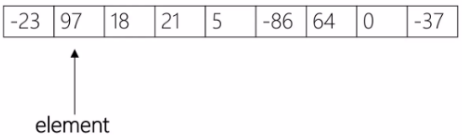
- In this search algorithm a sequential search is made over all the items one by one to search for the targeted item.
- Each item is checked in sequence until the match is found.
- If the match is found, a particular item is returned otherwise the search continues till the end.

Linear search is to check each element one by one in sequence. The following method linearSearch() searches a target in an array and returns the index of the target; if not found, it returns -1, which indicates an invalid index.

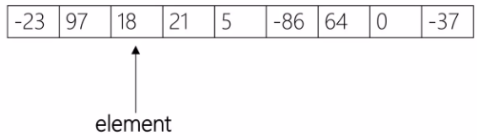
```
int linearSearch(int arr[], int target) {

for (int i = 0; i < arr.length; i++)
if (arr[i] == target) return i;
}
return -1;
```



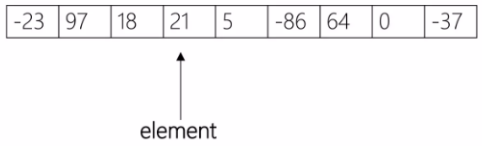
Searching for -86.



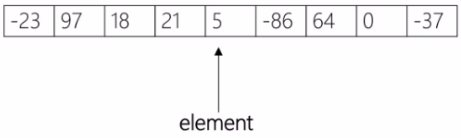
Searching for -86.



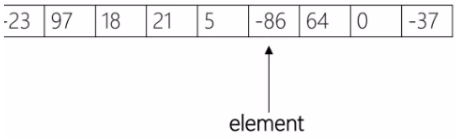
Searching for -86.



Searching for -86.



Searching for -86.



Searching for -86: found!

Linear search loops through each element in the array; each loop body takes constant time. Therefore, it runs in linear time O(n).

Case	Time Complexity
Best Case	O(1)
Average Case	O(n)
Worst Case	O(n)

Space Complexity:

Linear search uses a constant amount of auxiliary space since it only requires a single variable to iterate through the list. Therefore, the space complexity is O(1).

Applications of Linear Search Algorithm:

- **Unsorted Lists:** When we have an unsorted array or list, linear search is most commonly used to find any element in the collection.
- **Small Data Sets:** Linear Search is preferred over binary search when we have small data sets with
- **Searching Linked Lists:** In linked list implementations, linear search is commonly used to find elements within the list. Each node is checked sequentially until the desired element is found.
- **Simple Implementation:** Linear Search is much easier to understand and implement as compared to Binary Search or Ternary Search.

Advantages	Disadvantages
Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.	Linear search has a time complexity of O(N), which in turn makes it slow for large datasets.
Does not require any additional memory.	Not suitable for large arrays.
It is a well-suited algorithm for small datasets.	

BINARY SEARCH ALGORITHM

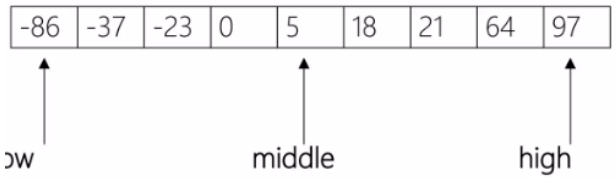
- Binary Search Algorithms are fast according to run time complexity.
- This algorithm works on the basis of divide and conquer rule.
- In this algorithm we have to sort the data collection in ascending order first then search for the targeted item by comparing the middle most item of the collection.
- If a match is found, the index of the item is returned. If the middle item is greater than the targeted item, the item is searched in the sun-array to the left of the middle item.
- Otherwise, the item is searched for in the sub-array to the right of the middle item.
- This process continues on the sub-array as well until the size of the sub array reduces to zero.

For sorted arrays, *binary search* is more efficient than linear search. The process starts from the middle of the input array:

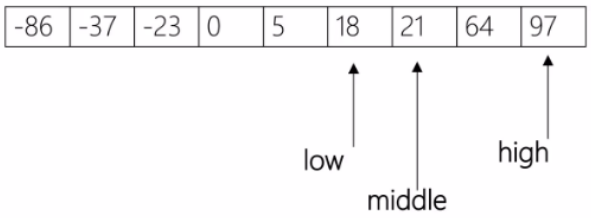
- ➔ If the target equals the element in the middle, return its index.
- ➔ If the target is larger than the element in the middle, search the right half.
- ➔ If the target is smaller, search the left half.

In the following binarySearch() method, the two index variables first and last indicate the searching boundary at each round.

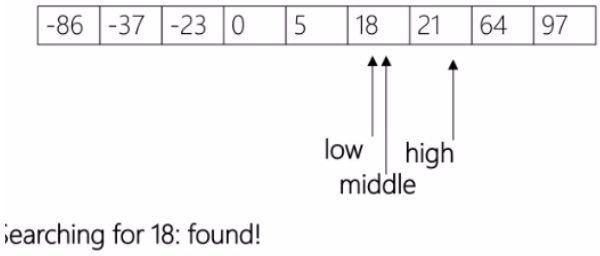
```
int binarySearch(int arr[], int target)

{
int first = 0, last = arr.length - 1;

while (first <= last)

{
int mid = (first + last) / 2; if (target == arr[mid])

return mid;

if (target > arr[mid]) first = mid + 1;
else
last = mid - 1;
}
return -1;
```



Searching for 18.



earching for 18.



Complexity Analysis of Binary Search:

➔ Time Complexity:

➔ Best Case: O(1)

➔ Average Case: O(log N)

➔ Worst Case: O(log N)

Advantages of Binary Search:

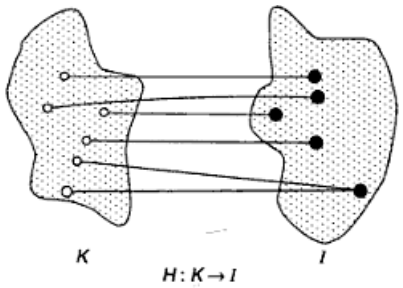
- Binary search is faster than linear search, especially for large arrays.
- More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
- Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

Drawbacks of Binary Search:

- The array should be sorted.
- Binary search requires that the data structure being searched be stored in contiguous memory locations.
- Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.

HASHING

We have seen about different search techniques (linear search, binary search) where search time is basically dependent on the no of elements and no. of comparisons performed. **Hashing** is a technique which gives constant search time. In hashing the key value is stored in the hash table depending on the hash function. The hash function maps the key into the corresponding index of the array(hash table). The main idea behind any hashing technique is to find one-to-one correspondence between a key value and an index in the hash table where the key value can be placed. Mathematically, this can be expressed as shown in figure below where K denotes a set of key values, I denotes a range of indices, and H denotes the mapping function from K to I.



All key values are mapped into some indices and more than one key value may be mapped into an index value. The function that governs this mapping is called the hash function. There are two principal criteria in deciding the hash function H:K->I as follows.

1) The function H should be very easy and quick to compute

2) It should be easy to implement

As an example let us consider a hash table of size 10 whose indices are 0,1,2,…9.Suppose a set of key values are 10,19,35,43,62,59,31,49,77,33. Let us assume the hash function as stated below

1) Add the two digits in the key

2) Take the digit at the unit place of the result as index , ignore the digits at tenth place if any

Using this hash function, the mapping from key values to indices and to hash tables are shown below.

K	l
10	1
19	0
35	8
43	7
62	8
59	4
31	4
49	3
77	4
33	6

$H: K \rightarrow l$

0	19
1	10
2	
3	49
4	59, 31, 77
5	
6	33
7	43
8	35, 62
9	

Hash table

HASH FUNCTIONS

Hash functions are a fundamental concept in computer science and play a crucial role in various applications such as data storage, retrieval, and cryptography. A hash function creates a mapping from an input key to an index in hash table.

There are various methods to define hash function

1) DIVISION METHOD

One of the fast hashing functions, and perhaps the most widely accepted, is the division method, which is defined as follows:

Choose a number h larger than the number n of keys in K. The hash function H is then defined by

H(k)=k(MOD h) if indices start from 0

Or

H(k)=k(MOD h)+1 if indices start from 1

Where k€K, a key value. The operator MOD defines the modulo arithmetic operator operation, which is equal to dividing k by h. For example if k=31 and **h=13 then,**

H(31)=31 MOD 13=5 (OR)

H(31)=31(MOD 13)+1=6

h is generally chosen to be a prime number and equal to the sizeof hash table

2) MID SQUARE METHOD

Another hash function which has been widely used in many applications is the mid square method. The hash function H is defined by H(k)=x, where x is obtained by selecting an appropriate number of bits or digits from the middle of the square of the key value k. example-

k	**:**	**1234**	**2345**	**3456**
k^2	**:**	**1522756**	**5499025**	**11943936**
H(k) :		**525**	**492**	**933**

For a three digit index requirement, after finding the square of key values,the digits at 2^{nd}, 4^{th} and 6^{th} position are chosen as their hash values.

3) FOLDING METHOD

Another fair method for a hash function is the folding method. In this method, the key k is partitioned into a number of parts k 1 , k2..kn where each part has equal no.of digits as the required address(index) width. Then these parts are added together in the hash function.

H(k)=k1+k2+…+kn. Where the last carry, if any, is ignored. There are mainly two variations of this method.

1) fold shifting method

2) fold boundary method Fold Shifting Method

In this method, after the partition even parts like k2, k4 are reversed before addition.

Fold boundary method

In this method, after the partition the boundary parts are reversed before addition

Example

-Assume size of each part is 2 then, the hash function is computed as follows

y values k :	**:2756**	**)9025**	**)43936**
opping	**: 52 27 56**	**49 90 25**	**)4 39 36**
re folding :	**-52+27+56=136**	**-49+90+25=169**	**-94+39+36=180**
d Shifting:	**-52+72+56=190**	**-49+09+25=133**	**-94+93+36=234**
d Boundary	**+52+27+65=154**	**-49+90+52=241**	**-94+39+63=207**

4) DIGIT ANALYSIS METHOD

This method is particularly useful in the case of static files where the key values of all the records are known in advance. The basic idea of this hash function is to form a hash address by extracting and/or shifting the extracted digits of the key. For any given set of keys, the position in the keys and the same rearrangement pattern must be used consistently. The decision for extraction and

rearrangement is finalized after analysis of hash functions under different criteria.

Example: given a key value 6732541, it can be transformed to the hash address 427 by extracting the digits from the even position. And then reversing this combination.ie 724 is the hash address.

Collision resolution and overflow handling techniques.

There are several methods to resolve collisions. Two important methods are listed below:

1) Closed hashing (linear probing)
2) Open hashing (chaining) Closed hashing

Suppose there is a hash table of size h and the key value is mapped to location i, with a hash function. The closed hashing can then be stated as follows.

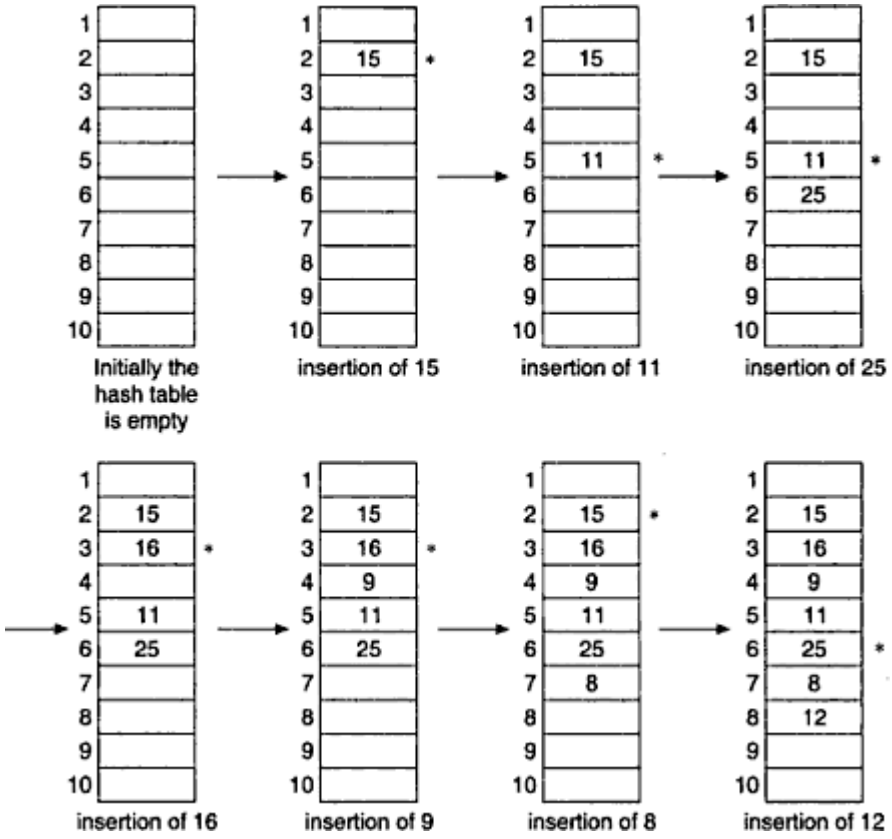
Start with the hash address where the collision has occurred,let it be i.

Then carry out a sequential search in the order:- i, i+1,i+2..h,1,2…,i-1 The search will continue until any one of the following occurs

- The key value is found
- An unoccupied location is found
- The searches reaches the location where search had started

The first case corresponds to successful search , and the other two cases correspond to unsuccessful search.Here the hash table is considered circular, so that when the last location is reached, the search proceeds to the first location of the table. This is why the technique is termed closed hashing. Since the technique searches in a straight line, it is alternatively termed as linear probing.

Example- Assume there is a hash table of size 10 and hash function uses the division method of remainder modulo 7, namely H(k)=k(MOD 7)+1.The construction of hash table for the key values 15,11,25,16,9,8,12,8 is illustrated below.



Drawback of closed hashing and its remedies

The major drawback of closed hashing is that as half of the hash table is filled, there is a tendency towards clustering.That is key values are clustered in large groups.and as a result sequential search becomes slower and slower. This kind of clustering is known as primary clustering.

The following are some solutions to avoid this situation

1) **Random probing**
2) **Double hashing**
3) **Quadratic probing**

Random Probing

Instead of using linear probing that generates sequential locations in order, a random location is generated using random probing.

An example of pseudo random number generator that generates such a random sequence is given below:

$$I=(i+m)MOD\ h+1$$

Where m and h are prime numbers. For example if m=5, and h=11 and initially=2 then random probing generates the sequence 8,3,9,4,10,5,11,6,1,7,2

Here all numbers are generated between 1 and 11 in a random order. Primary clustering problem is solved. Where as there is an issue of clustering when two keys are hashed into the same location and then they make use of the same sequence locations generated by the random probing, which is called as secondary clustering

Double Hashing

An alternative approach to solve the problem of secondary clustering is to make use of second hash function in addition to the first one. Suppose H1(k) is initially used hash function and H2(k) is the second one. These two functions are defined as

$$H1(k)=(k\ MOD\ h)+1\quad H2(k)=(k\ MOD\ (h-4))+1$$

Let h=11, and k=50 for an instance, then H1(50)=7 and H2(50)=2.

Now let k=28, then

$$H1(28)=7 \text{ and } H2(28)=5$$

Thus for the two key values hashing to the same location, rehashing generates two different locations alleviating the problem of secondary clustering.

Quadratic Probing

It is a collision resolution method that eliminates the primary clustering problem of linear probing. For linear probing, if there is a collision at location i, then the next locations i+1, i+2..etc are probed.But in quadratic probing next locations to be probed are $i+1^2$,$i+2^2$,$i+3^2$ ..etc . This method substantially reduces primary clustering, but it doesn't probe all the locations in the table.

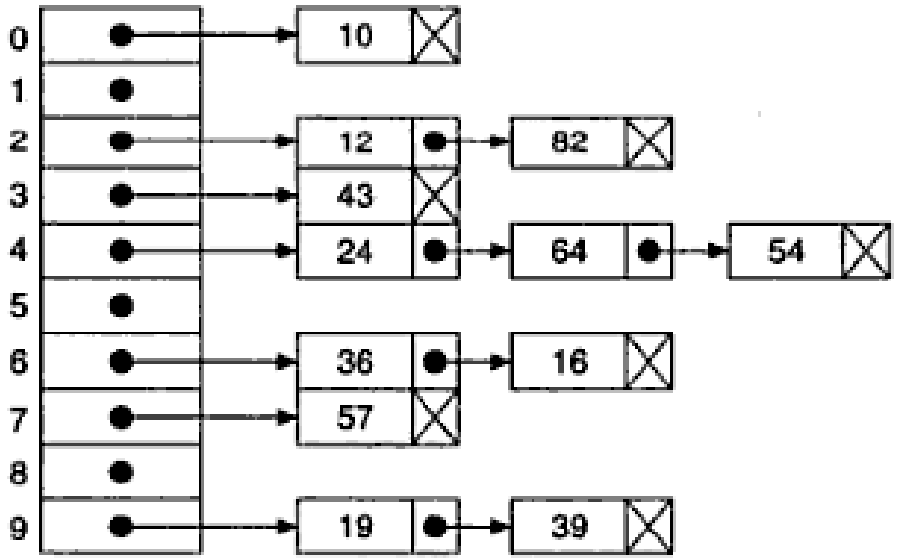
Open Hashing

Closed hashing method for collision resolution deals with arrays as hash tables and thus random positions can be quickly referred. Two main disadvantages of closed hashing are

1) It is very difficult to handle the problem of overflow in a satisfactory manner

2) The key values are haphazardly intermixed and, on the average majority of the key values are from their hash locations increasing the number of probes which degrades the overall performance

To resolve these problems another hashing method called open hashing or separate chaining is used. The chaining method uses hash table as an array of pointers. Each pointer points to a linked list. That is here the hash table is an array of list of headers. Illustrated below is an example with a hash table of size10.



For searching a key in hash table requires the following steps

1) Key is applied to hash function
2) Hash function returns the starting address of a particular linked list(where key may be present)
3) Then key is searched in that linked list

Performance Comparison Expected

Algorithm Name	Best Case	Average Case	Worst Case
Quick Sort	**O(n log n)**	**O(n log n)**	**$O(n^2)$**
Merge Sort	**O(n log n)**	**O(n log n)**	**O(n log n)**
Heap Sort	**O(n log n)**	**O(n log n)**	**O(n log n)**
Bubble Sort	**O(n)**	**$O(n^2)$**	**$O(n^2)$**
Selection Sort	**$O(n^2)$**	**$O(n^2)$**	**$O(n^2)$**
Insertion Sort	**O(n)**	**$O(n^2)$**	**$O(n^2)$**
Binary Search	**O(1)**	**O(log n)**	**O(log n)**
Linear Search	**O(1)**	**O(n)**	**O(n)**

QUESTION BANK

MODULE 1

1. Compute the time complexity of linear search algorithm using frequency count method
2. Explain space complexity
3. Explain various asymptotic notations used in analysis of algorithm
4. Calculate the frequency count of the statement x = x+1; in the following code segment for (i = 0; i< n; i++) for ('1 = 0; j< n; j++) x=x+1;
5. What do you mean by the time complexity of an algorithm? Derive the Big O notation for the function f(n) = n2+3n+2
6. Explain the best case, worst case, average case of linear search algorithm.
7. What is frequency count? Calculate the frequency count of the statement

   x : x*l; in the following code segment

   for (i:0; i< n; i+r)

   for(: l;j<n;j*=2)

   x:x* l;
8. Find the postfix expressions of the following infix expression

   a) (A+B) *K+D/(E+F*G)+H) \

   b) (A/D+B)*(K"Y)
9. Given a matrix having l0 rows and l0 columns and 12 nonzero elements. How much space can be saved by representing the matrix in sparse (tuple) form?
10. Discuss an algorithm to convert an infix expression to a postfix expression
11. Write an algorithm to find the transpose of a matrix represented in tuple form
12. Write algorithms to insert and delete elements from a circular Queue
13. Write an algorithm to add two polynomials represented using arrays
14. Write any three applications of Stack.
15. Explain PUSH and pOp operations in stack

16. What is a sparse matrix?
17. Write an algorithm to add two sparse matrices.
18. Write an algorithm to insert an element to a circular queue using array

    Convert P*{Q+RyS to postfix notation.
19. Write algorithm and step-by-step conversion using the stack.
20. Write an algorithm to search an element using binary search. Discuss its timecomplexity

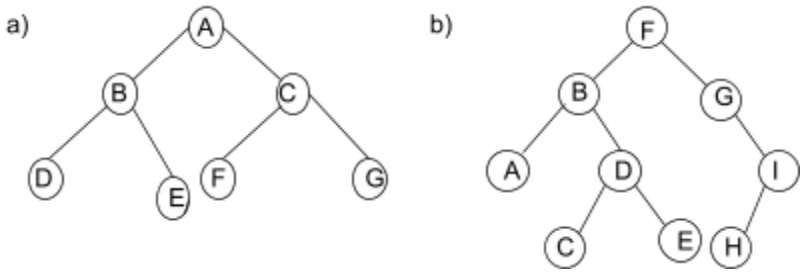
MODULE 2

1. What is dynamic memory allocation? List any two advantages of dynamic memory allocation
2. write an algorithm to count number of nodes in a single linked list
3. Write procedures to push and pop elements from a Linked List Stack
4. Memory blocks of size 202,302 and 101 are allocated for programs of size 150,100, 125, 100 and 100. Which allocation method is better in this case and why?
5. Explain memory allocation for fixed sized blocks with the help of an algorithm
6. Explain Worst-fit allocation with an example
7. Write algorithms to multiply two polynomials represented using linked list
8. Write algorithms to insert elements and delete elements from the beginning of a Circular Double Linked List
9. How can a linked list used to represent the polynomial 3xa+2*+5. Write an algorithm to add two polynomials represented using a linked list.
10. Write an algorithm to delete a given node in a singly linked list.
11. What are the advantages of linked list over arrays? Write algorithms to implement Queue using linked list
12. Given five memory partitions of 300Kb, 700Kb, 400Kb, 500Kb, g00Kb (in order), how would the first-fit,r$est-fit, and worst-fit algorithms place processes of 4l2Kb,617 Kb, I 12 Kb, and 626Kb (in order)?
13. Write polynomial addition algorithm using linked list and illustrate with an example

14. Compare singly linked list and doubly linked list
15. What do you mean by a circular linked list? Write an algorithm to perform insert and delete operations on a circular linked list'

MODULE 3

1. Write and illustrate depth first search algorithm.
2. Explain various representation of graph with example
3. Write and discuss algorithms to insert an element to Binary search tree. Show the structure of the binary search tree after adding each of the following values in that order: 2, 5, 1, 7, 10, 9, 11, 6
4. Write algorithms for Depth First Search and Breadth First Search of a Graph
5. Write an algorithm to delete a node from a Binary Search Tree
6. Explain an application of a graph with an example
7. Write the output of inorder, preorder & postorder traversals on the following tree



8. Differentiate between complete binary tree and full binary tree. Give examples for each.
9. Create a Binary Search Tree for the following values 62, 14, 96, 12, 105, 3, 75, 22, 87, 32, 20, 13, 102, 69, 125
10. What are the different ways to represent a tree in memory?
11. How is a binary tree represented using arrays? Explain with a diagram.
12. Explain the three types of tree traversals with examples:
    - Inorder
    - Preorder
    - Postorder

13. Construct an expression tree for the infix expression: (a + b) * (c - d).
14. Define a binary heap. What are its types?
15. Construct a max-heap from the given array: [4, 10, 3, 5, 1].

MODULE 4

1. What is hashing? List any two applications of hashing.
2. Explain Max Heap with an example.
3. Write an algorithm to sort a set of numbers using insertion sort
4. Explain the algorithm for Merge Sort with an example
5. Explain Quick sort algorithm with an example:
6. Illustrate the differences between selection sort and insertion sort with an example.
7. Explain with examples the different techniques for open addressing
8. How can the folding method be used for hashing?
9. What is meant by collision? Give an example.
10. Explain any four commonly used hash functions
11. Hash the following keys using open chaining method and closed linear probing method. The size of the table is 7 and the Hash function H(K) : K mod Z. Keys :{ 16, 21, 23, 50, 19, 29}
12. What do you mean by Double Hashing? Explain with an example
13. Build a max heap from the array [4, 10, 3, 5, 1] and perform one iteration of heap sort.
14. Perform radix sort on the array [170, 45, 75, 90, 802, 24, 2, 66]. Show sorting at each digit level.
15. implement linear search on an unsorted list. What is the worst-case time complexity?
16. Trace binary search on the sorted array [2, 4, 6, 10, 12, 18, 20] to find 10.
17. Why must the array be sorted before using binary search?
18. Explain how quadratic probing reduces clustering compared to linear probing.
19. Describe the Folding method and hash the key 98765432 using 4-digit folding.
20. What are the pros and cons of open hashing compared to closed hashing?