

Data Structures and Algorithms

18K1CS01

- Mrs.N.Subha
- Mrs.G.Muthamiz selvi
- Mrs.N.Anuradha
- Mrs.K.Hemalatha

Data Structures and Algorithms(18K1CS01)

Objective: To provide basic understanding on common Data Structures and Algorithms.

Unit -1: Introduction: History of Algorithms – Definition, Structure and Properties of Algorithms – Development of an Algorithm – Data Structures and Algorithms – Data Structure - Definition and Classification. Analysis of Algorithms: Efficiency of Algorithms – Asymptotic Notations - Time Complexity of an Algorithm Using O Notation – Average, Best and Worst Case Complexities. Arrays: Introduction – Array Operations – Number of Elements in an Array – Representation of Arrays in Memory - Applications.

Unit –II: Stacks: Introduction – Stack Operations – Applications. Queues: Introduction – Operations on Queues – Circular Queues – Other Types of Queues – Applications. Linked Lists: Introduction – Singly Linked List – Circularly Linked Lists – Doubly Linked Lists – Applications.

Unit –III: Trees and Binary Trees: Introduction – Trees: Definition and Basic Terminologies – Representation of Trees – Binary Trees: Basic Terminologies and Types – Representation of Binary Trees – Binary Tree Traversals – Threaded Binary Trees – Application. Graphs: Introduction – Definitions and Basic Terminologies – Representations of Graphs – Graph Traversals – Applications.

Unit- IV: Binary Search Trees and AVL Trees: Binary Search Trees: Definition and Operations – AVL Trees: Definition and Operations. Hash Tables: Hash Table Structure – Hash Functions – Linear Open Addressing- Chaining.

Unit –V: Searching: Linear Search – Binary Search – Fibonacci Search. Internal Sorting: Bubble Sort – Insertion Sort – Selection Sort – Merge Sort – Quick Sort – Heap Sort.

Text:

“Data Structures And Algorithms Concepts, Techniques and Applications” – G.A.V Pai - Published by Tata McGraw Hill Education Pvt. Ltd., - Sixth reprint 2011.

Chapters:1,2.1,2.3,2.4,2.6,3,4,5,6.1–6.4,6.6,8,9,10.2,10.3,13.2,13.5,15.2,15.5,15.6,16.2-16.5,16.7,16.8

Reference:

- 1."Data Structures" – LIPSCHUTA -Tata Mcgraw Hill,Schaum's Outline Series.-2006.
- 2."Fundamentals of Data Structure" – Ellis Horowitz, Sartaj Sahni and Sanguthevar- 2006.
- 3."Data Structure and Algorithm"- Alfred V.Aho,John E.Hopcroft,Jeffrey D.Ullman-pearson education – 2012.

M. S. S. S.
Head, Dept of Computer Science,
Kunthavai Natchiar Govt Arts College (W).
THANJAVUR- 7.

Introduction

- Ubiquitous Machine
- Computer
- Omnipresent
- Omnipotent
- Omniscient

Basic Applications of Computer

- ❑ Home. **Computers** are used at homes for several purposes like online bill payment, watching movies or shows at home, home tutoring, social media access, playing games, internet access, etc. ...
- ❑ Medical Field. ...
- ❑ Entertainment. ...
- ❑ Industry. ...
- ❑ Education. ...
- ❑ Government. ...
- ❑ Banking. ...
- ❑ Business.

Algorithm

- Defined as process, or procedure or method or recipe.
- It is a specific set of rules to obtain a definite output from specific inputs.

History of Algorithms

- ❓ The word algorithm originates from the arabic word *algorism*.
- ❓ Arabic mathematician name Abu Jafer Mohammed Ibn Musa Al Khwarizmi.
- ❓ First algorithm designer.
- ❓ Designed for adding numbers.

Definition, Structure and properties of Algorithms

Definition:

Defined as a finite sequence of instructions.

Structure :

- i) Input step
- ii) Assignment step
- iii) Decision step
- iv) Repetitive step
- v) Output step

Properties:

1. Finiteness
2. Definiteness
3. Generality
4. Effectiveness
5. Input-Output

Development of an Algorithm

1. Problem Statement
2. Model Formulation
3. Algorithm Design
4. Algorithm Correctness
5. Implementation
6. Algorithm analysis
7. Program Testing
8. Documentation

Data Structure-Definition and Classification

Data Type – The type of values that variable in a programming language hold.

Primitive Data Types - Integer, real, character, boolean

Data Object – A list of elements

Abstract Data Types – Data objects which comprise the data structure, and their fundamental operation.

Defined as a set of data D over a domain L and supporting a list of operations O .

Analysis of Algorithms

- Efficiency of Algorithms
- Asymptotic Notations
- Time complexity of an algorithm using O notation
- Average, Best and Worst case Complexities

Efficiency of Algorithms

The performance of algorithms can be measured on the scales of Time and Space.

Time: Looking for fastest algorithm for the problem or performs its task in the minimum possible time.

The performance measure is called **Time Complexity-Running time of the algorithm or program.**

Space: Looking for an algorithm that consumes or needs limited memory space for its execution.

The performance measure is called **Space Complexity-Space needed for the algorithm or program.**

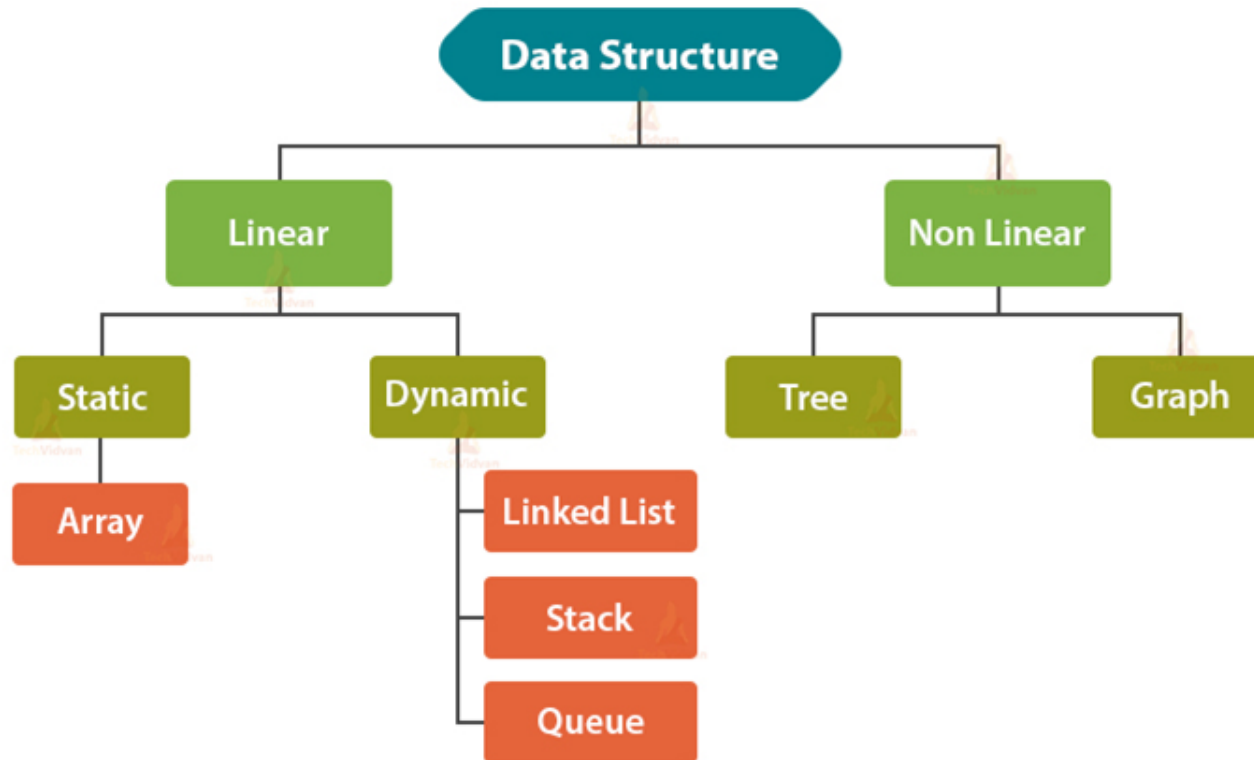
Time complexity of an algorithm using O notation

- “Big Oh” Notation.
- $O(1)$ – Constant time
- $O(n)$ – Linear time
- $O(n^2)$ – Quadratic time
- $O(n^3)$ - Cubic time
- $O(\log n)$ – Logarithmic time
- $O(2^n), O(3^n), O(k^n)$ – Exponential time
- $O(1) \leq O(\log n) \leq O(n) \leq O(n \log n) \leq O(n^2) \leq O(n^3) \leq O(2^n)$

Average, Best and Worst Case Complexities

- The **worst-case complexity** of the algorithm is the function defined by the **maximum number of steps** taken on any instance of size n .
- The **best-case complexity** of the algorithm is the function defined by the **minimum number of steps** taken on any instance of size n .
- Finally, the **average-case complexity** of the algorithm is the function defined by the **average number of steps** taken on any instance of size n .

Classification of Data Structure



INTRODUCTION

- An array is an ADT whose objects are *sequence of elements of the same type*.
- The *two operations* performed on it are **store** and **retrieve**.
- Thus if a is an array the operations can be represented as
STORE (a, i, e) and RETRIEVE (a, i)
where i is termed as the **index** and e is the **element** that is to be stored in the array.
- These functions are equivalent to
 $a[i] := e$
where i - subscript
 a - array variable name
 e - value in programming language.

CONTD...

- Arrays could be of
 - **One-dimensional**
 - **Two dimensional**
 - **Three-dimensional/Multidimensional.**

- One-dimensional arrays are mathematically likened to vectors.
- Two-dimensional arrays are likened to matrices.
- Two-dimensional arrays have rows and columns associated with them.

Examples of arrays

$$A[1:5] \begin{array}{|c|c|c|c|c|} \hline 6 & -4 & 3 & 2 & 11 \\ \hline \end{array}$$

(a) One-dimension

$$B[1:3, 1:2] \begin{array}{cc} & \begin{array}{cc} 1 & 2 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} -6 & 4 \\ 3 & 2 \\ 7 & -5 \end{bmatrix} \end{array}$$

(b) Two-dimension

Contd...

- $A[1:5]$ refers to a one-dimensional array where 1, 5 are referred to as the lower and upper indexes or the lower and upper bounds of the index range.
- Similarly, $B[1:3, 1:2]$ refers to a two-dimensional array with 1, 3 and 1, 2 being the lower and upper indexes of the rows and columns respectively.
- Also, each element of the array viz., $A[i]$ or $B[i, j]$ resides in a memory location also called a cell. Here cell refers to a unit of memory and is machine dependent.

Array Operations

- An array when viewed as a data structure supports only two operations viz.,
- (i) **storage of values** - writing into an array
(STORE (a, i, e)) and,
- (ii) **retrieval of values** - reading from an array
(RETRIEVE (a, i))

Array operations: Store and Retrieve

- For example, if A is an array of 5 elements then.

OBJECT	REPRESENTATION IN MEMORY	OPERATIONS	RESULT OF THE OPERATIONS																								
$A[1:5]$	<table border="1"><tr><td>A</td><td>6</td><td>-4</td><td>3</td><td>2</td><td>11</td></tr><tr><td></td><td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td><td>[5]</td></tr></table>	A	6	-4	3	2	11		[1]	[2]	[3]	[4]	[5]	STORE (A, 3, 17) RETRIEVE (A, 2)	<table border="1"><tr><td>A</td><td>6</td><td>-4</td><td>17</td><td>2</td><td>11</td></tr><tr><td></td><td>[1]</td><td>[2]</td><td>[3]</td><td>[4]</td><td>[5]</td></tr></table> -4	A	6	-4	17	2	11		[1]	[2]	[3]	[4]	[5]
A	6	-4	3	2	11																						
	[1]	[2]	[3]	[4]	[5]																						
A	6	-4	17	2	11																						
	[1]	[2]	[3]	[4]	[5]																						

Number of Elements in an Array

- The computation of size of the array by way of number of elements.
- When arrays are declared in a program, it is essential that the number of memory locations needed by the array are booked before hand.

One-dimensional array

- Let $A[1:u]$ be a one-dimensional array.
- The size of the array, as is evident is u and the elements are $A[1], A[2], \dots, A[u-1], A[u]$.
- In the case of the array $A[l : u]$
- where l is the lower bound and u is the upper bound of the index range.
- The number of elements is given by $(u - l + 1)$.

Example

The number of elements in

(i) $A[1:26] = 26$

(ii) $A[5:53] = 49$ (i.e. $53 - 5 + 1$)

(iii) $A[-1:26] = 28$

Multi-dimensional array

A multi-dimensional array $A[1 : u_1, 1 : u_2, \dots, 1 : u_n]$ has a size of $u_1 \cdot u_2 \dots u_n$ elements, (i.e.) $\prod_{i=1}^n u_i$.

Figure 3.4 illustrates a three-dimensional array and its size. Generalizing, the array $A[l_1 : u_1, l_2$

$: u_2, l_3 : u_3 \dots l_n : u_n]$ has a size of $\prod_{i=1}^n (u_i - l_i + 1)$ elements.

Size of a three dimensional array

Array:

Elements

Number of elements

$A[1:2 \ 1:2 \ 1:3]$

$A[1, 1, 1] \ A[1, 1, 2] \ A[1, 1, 3]$

$A[1, 2, 1] \ A[1, 2, 2] \ A[1, 2, 3]$

$A[2, 1, 1] \ A[2, 1, 2] \ A[2, 1, 3]$

$A[2, 2, 1] \ A[2, 2, 2] \ A[2, 2, 3]$

$$2 \times 2 \times 3 = 12$$

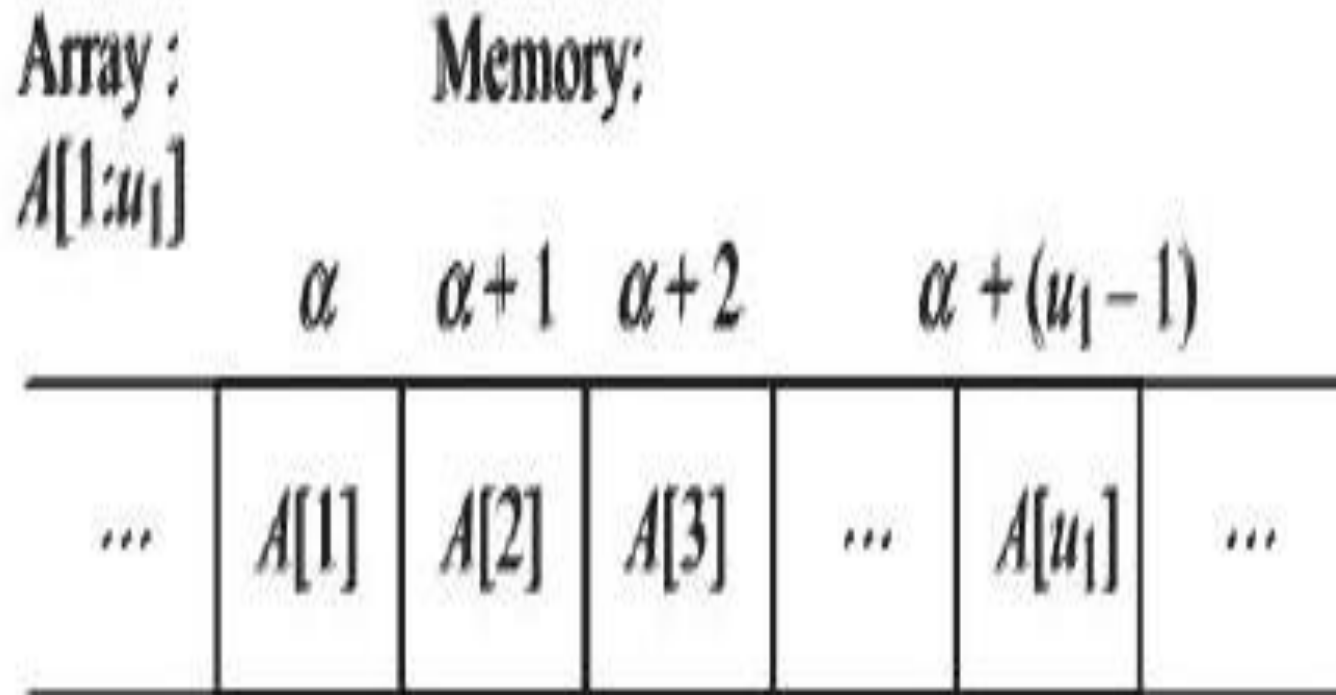
Representation of Arrays in Memory

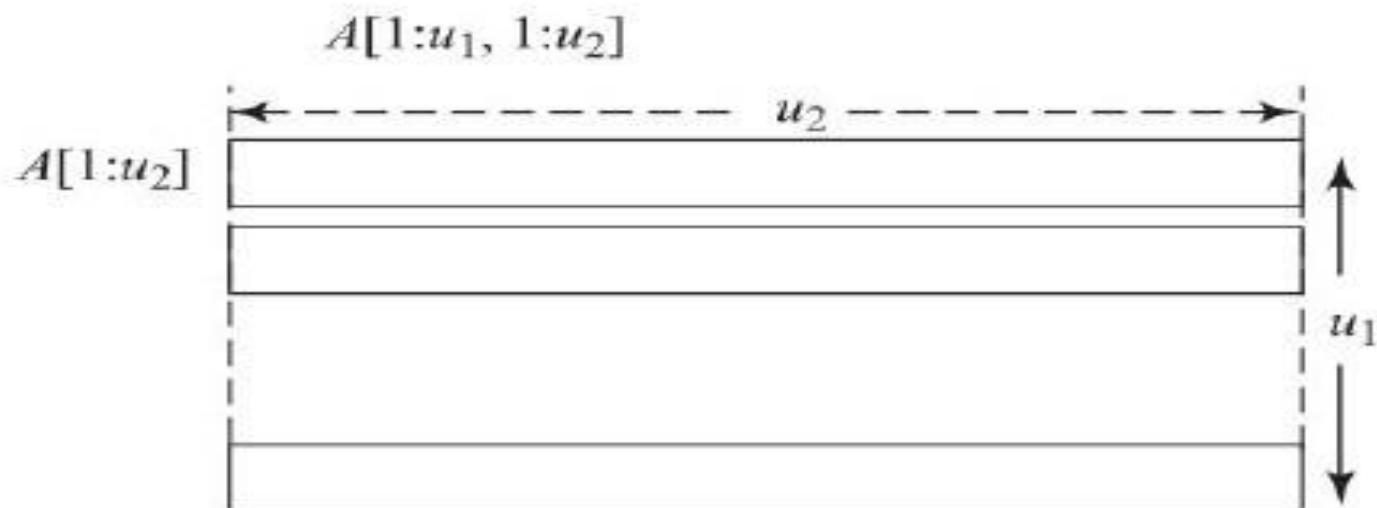
- The name of the array is associated with the address of the starting memory location so as to facilitate efficient storage and retrieval.
- The computer memory is considered one-dimensional (linear) it has to accommodate arrays which are multi-dimensional.

One-dimensional array

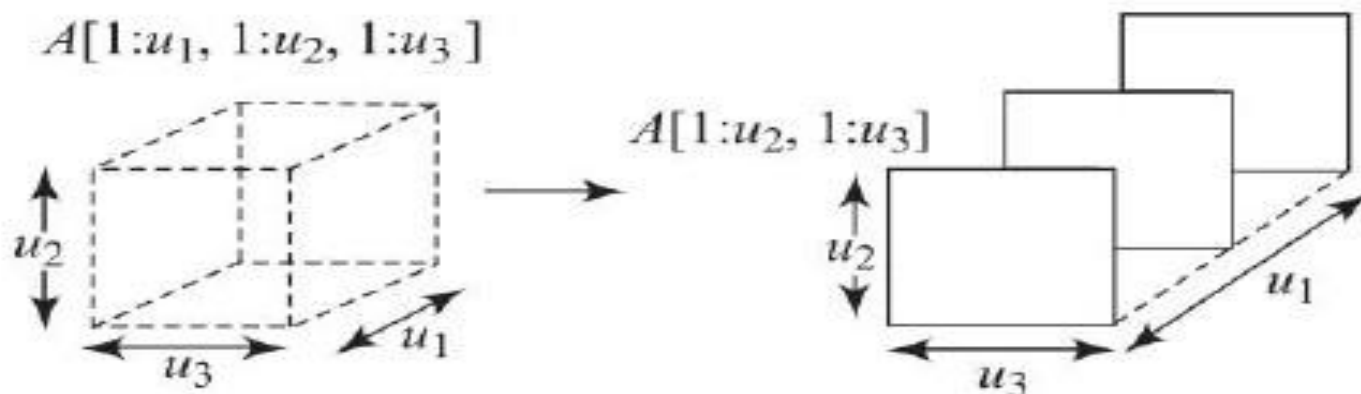
- Consider the array $A(1 : u_1)$ and let a be the address of the starting memory location referred to as the base address of the array.
- $A[1]$ occupies the memory location whose address is a , $A[2]$ occupies $a + 1$ and so on. In general, the address of $A[i]$ is given by $a + (i - 1)$.
- In general, for a one-dimensional array $A(l_1 : u_1)$ the address of $A[i]$ is given by $a + (i - l_1)$, where a is the base address.

Representation of one-dimensional arrays in memory





(a) Two-dimensional array viewed in terms of one-dimensional arrays

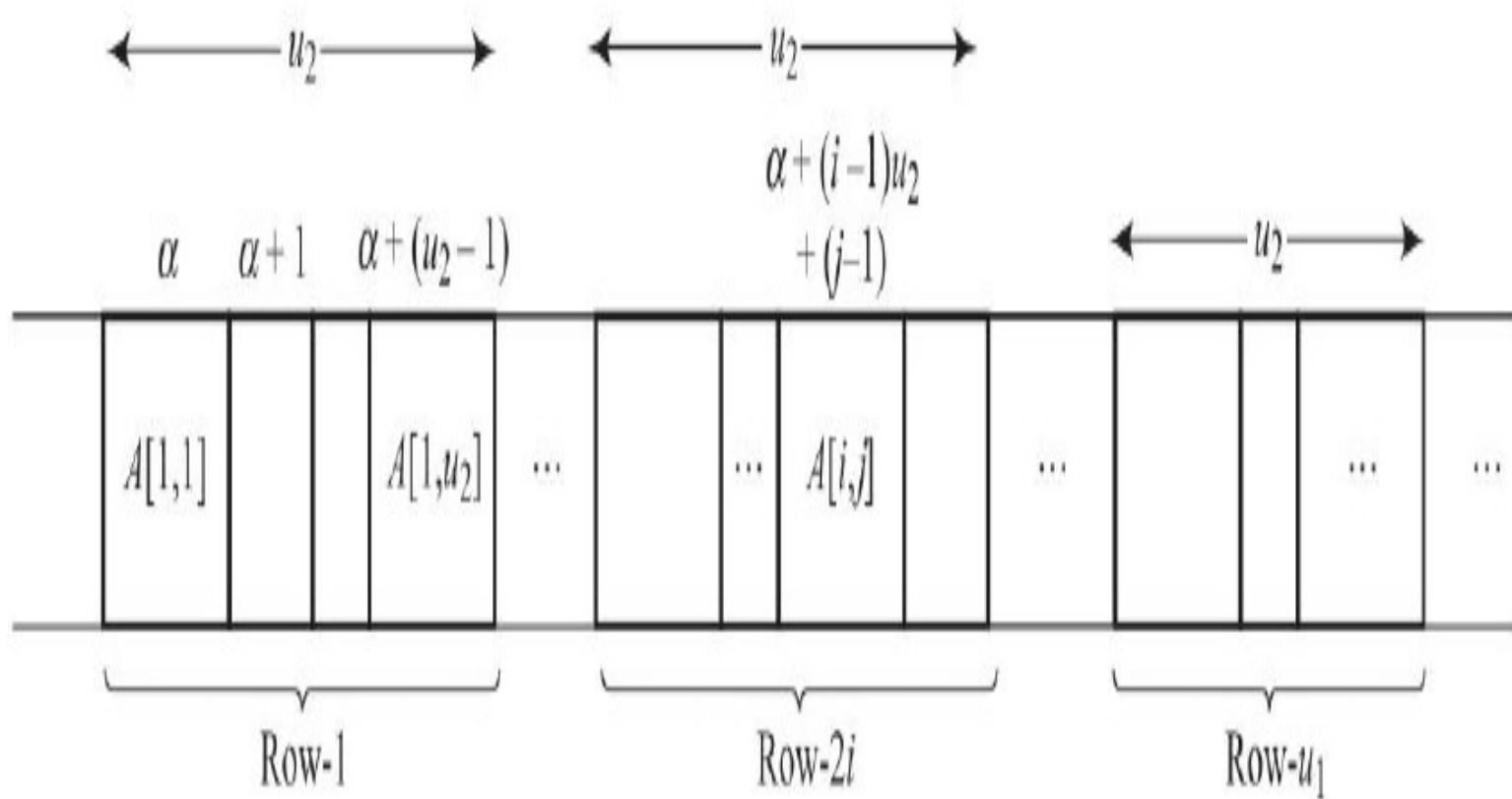


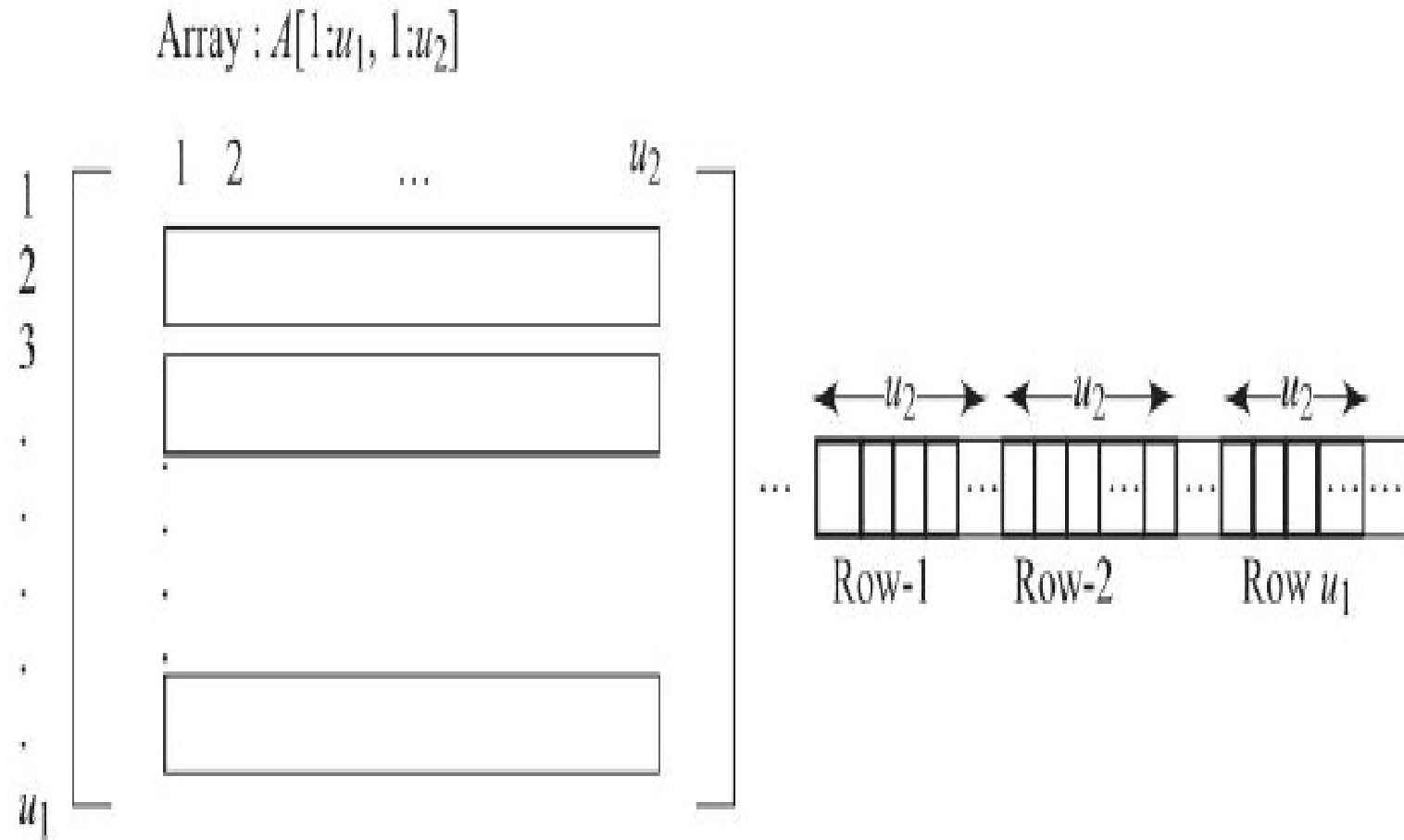
(b) Three-dimensional array viewed in terms of two-dimensional arrays

Two-dimensional array

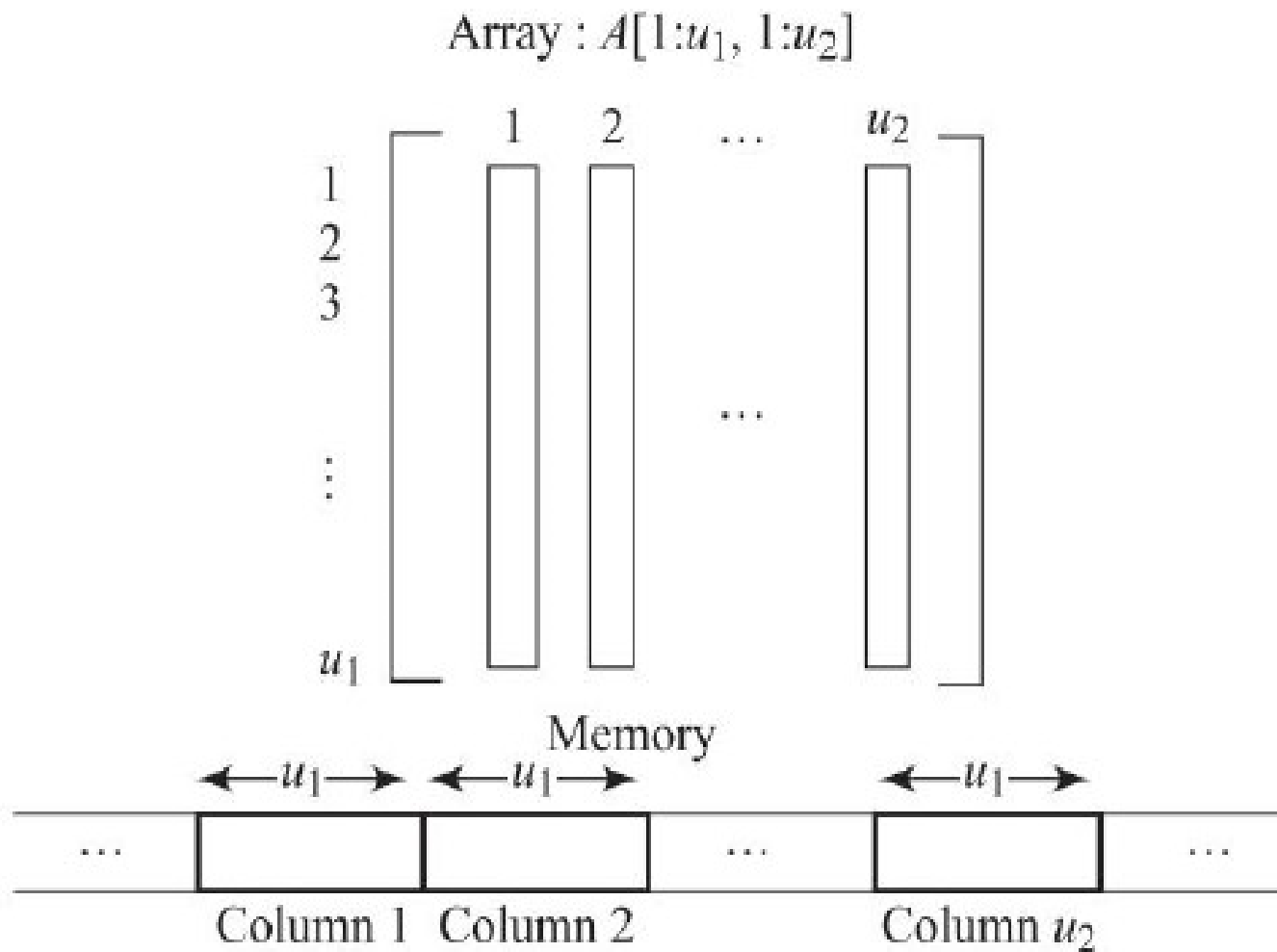
- The array $A[1 : u_1, 1 : u_2]$ which is to be stored in the memory.
- It is helpful to imagine this array as u_1 number of one-dimensional arrays of length u_2 .
- Thus if $A[1, 1]$ is stored in address a , the base address, then $A[i, 1]$ has address $a + (i - 1) u_2$, and $A[i, j]$ has address $a + (i - 1) u_2 + (j - 1)$.
- compute the address of $A[i, j]$ as $a + (i - 1) u_2 + (j - 1)$.
- The addresses of array elements are expressed in terms of the **cells**.
- In general, for a two-dimensional array $A[l_1 : u_1, l_2 : u_2]$ the address of $A[i, j]$ is given by
- $a + (i - l_1)(u_2 - l_2 + 1) + (j - l_2)$

Representation of a two-dimensional array in memory





(a) Row major order



(b) Column major order

Three-dimensional array

- the three-dimensional array $A[1 : u_1, 1 : u_2, 1 : u_3]$. As discussed before, we shall imagine
- it to be u_1 number of two-dimensional arrays of dimension $u_2 \cdot u_3$.
- the address of $A[i, 1, 1]$ would be $a + (i - 1) u_2 \cdot u_3$.
- Similarly the address of $A[i, j, 1]$ requires accessing the first room on the j th floor of the i th
- building which works out to $a + (i - 1) u_2 u_3$
- $+ (j - 1) u_3$. Proceeding on similar lines, the address
- of $A[i, j, k]$ is given by $a + (i - 1) u_2 u_3 + (j - 1) u_3 + (k - 1)$.

Applications

- Two concepts that are useful to computer science and also serve as applications of arrays.
 1. Sparse matrices
 2. Ordered lists.

Sparse matrix

- A matrix is a mathematical object which finds its applications in various scientific problems.
- A matrix is an arrangement of $m \cdot n$ elements arranged as m rows and n columns.
- The Sparse matrix is a matrix with zeros as the dominating elements.

Matrix and a sparse matrix

$$\begin{bmatrix} 2 & 4 & 6 & 8 \\ 1 & 2 & 0 & 2 \\ 0 & 1 & 1 & 6 \\ 2 & 0 & 1 & 4 \end{bmatrix}$$

(a) Matrix

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

(b) Sparse Matrix

Ordered lists

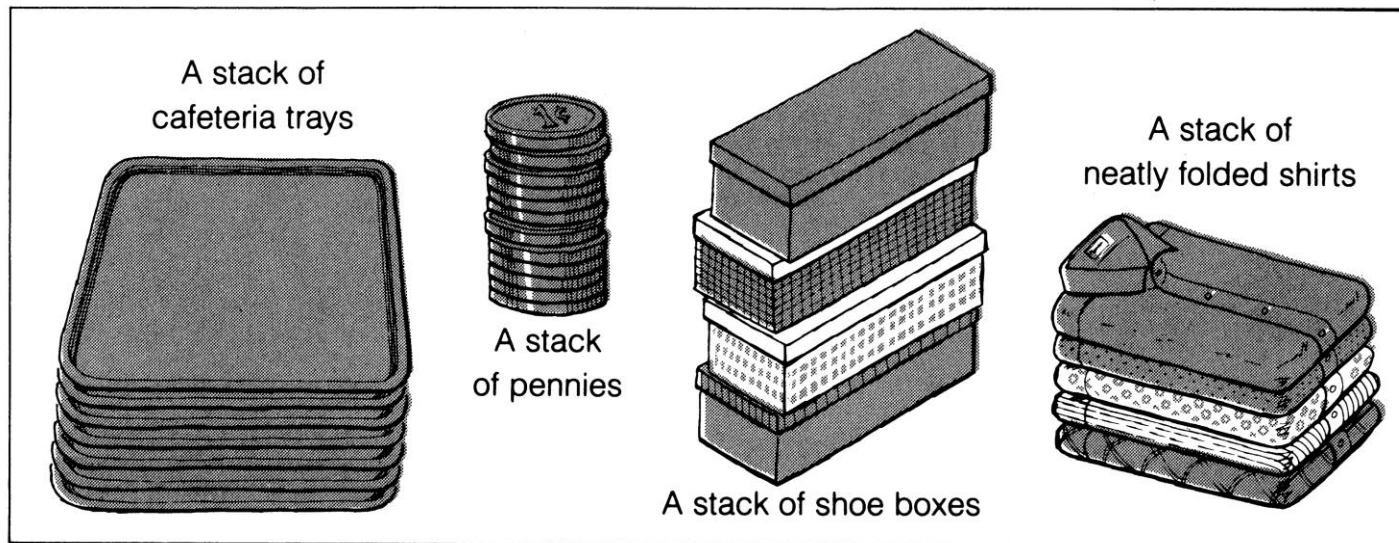
- One of the simplest and useful data objects in computer science is an ***ordered list or linear list***.
- An ordered list can be either empty or non empty. In the latter case, the elements of the list are known as ***atoms***, chosen from a set D.
- The ordered lists provide a variety of operations such as ***retrieval, insertion, deletion, update*** etc.
- The most common way to represent an ordered list is by using a one-dimensional array.
- Such a representation is termed ***sequential mapping***.
- **Example**
 - (i) (sun, mon, tue, wed, thu, fri, sat)
 - (ii) (a1, a2, a3, a4, ..., a_n)
 - (iii) (Unix, CP/M, Windows, Linux)

operations performed on ordered lists, with examples.

Operation	Original ordered list	Resultant ordered list after the operation
Insertion (Insert a_6)	(a_1, a_2, a_7, a_9)	$(a_1, a_2, a_6, a_7, a_9)$
Deletion (Delete a_9)	(a_1, a_2, a_7, a_9)	(a_1, a_2, a_7)
Update (update a_2 to a_5)	(a_1, a_2, a_7, a_9)	(a_1, a_5, a_7, a_9)

What is a stack?

- It is an ordered group of homogeneous items or elements.
- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).
- The last element to be added is the first to be removed (**LIFO**: Last In, First Out).



Stack Specification

- Definitions: (provided by the user)
 - *MAX_ITEMS*: Max number of items that might be on the stack
 - *ItemType*: Data type of the items on the stack
- Operations
 - MakeEmpty
 - Boolean IsEmpty
 - Boolean IsFull
 - Push (ItemType newItem)
 - Pop (ItemType& item)

Push (ItemType newItem)

- *Function*: Adds newItem to the top of the stack.
- *Preconditions*: Stack has been initialized and is not full.
- *Postconditions*: newItem is at the top of the stack.

Pop (ItemType& item)

- *Function*: Removes topItem from stack and returns it in item.
- *Preconditions*: Stack has been initialized and is not empty.
- *Postconditions*: Top element has been removed from stack and item is a copy of the removed element.

Stack overflow

- The condition resulting from trying to push an element onto a full stack.

```
if(!stack.IsFull())  
    stack.Push(item);
```

Stack underflow

- The condition resulting from trying to pop an empty stack.

```
if(!stack.IsEmpty())  
    stack.Pop(item);
```

Implementing stacks using dynamic array allocation

```
template<class ItemType>
class StackType {
public:
    StackType(int);
    ~StackType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Push(ItemType);
    void Pop(ItemType&);
```

Implementing stacks using dynamic array allocation (cont.)

```
template<class ItemType>
StackType<ItemType>::StackType(int max)
{
    maxStack = max;
    top = -1;
    items = new ItemType[max];
}
```

```
template<class ItemType>
StackType<ItemType>::~~StackType()
{
    delete [ ] items;
}
```

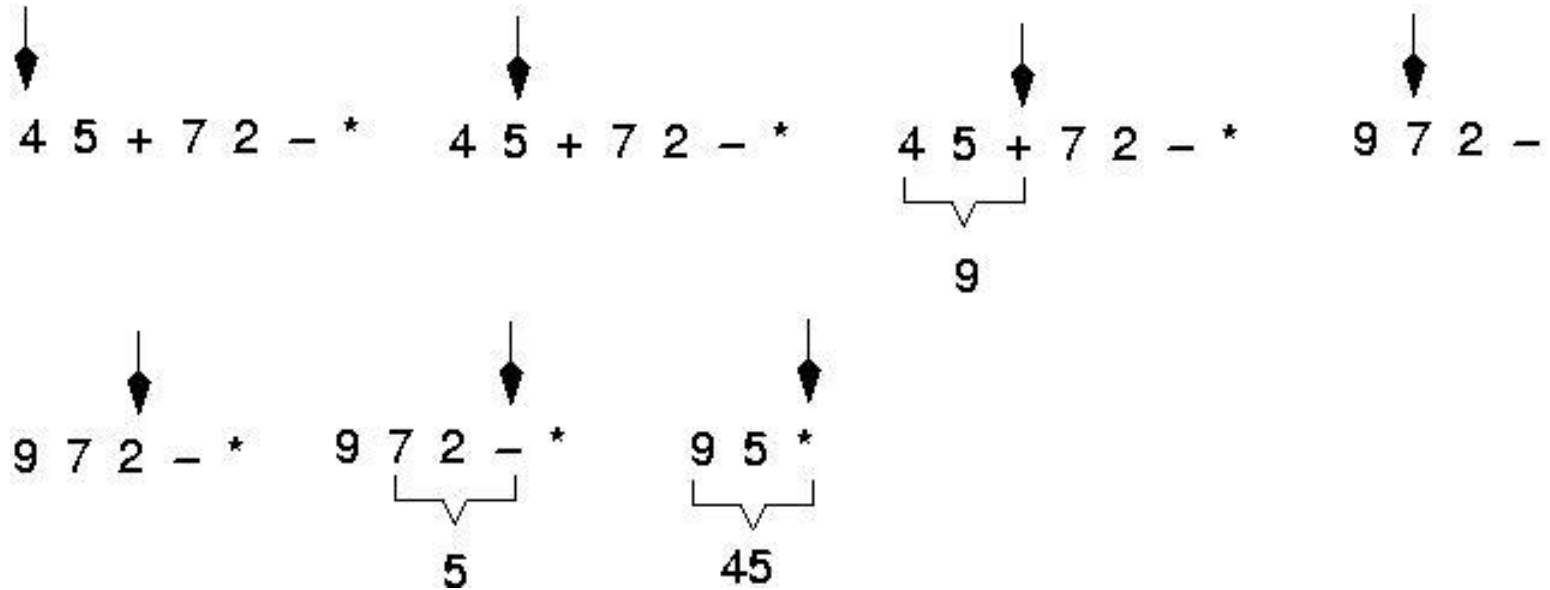
Example: postfix expressions

- Postfix notation is another way of writing arithmetic expressions.
- In postfix notation, the operator is written after the two operands.

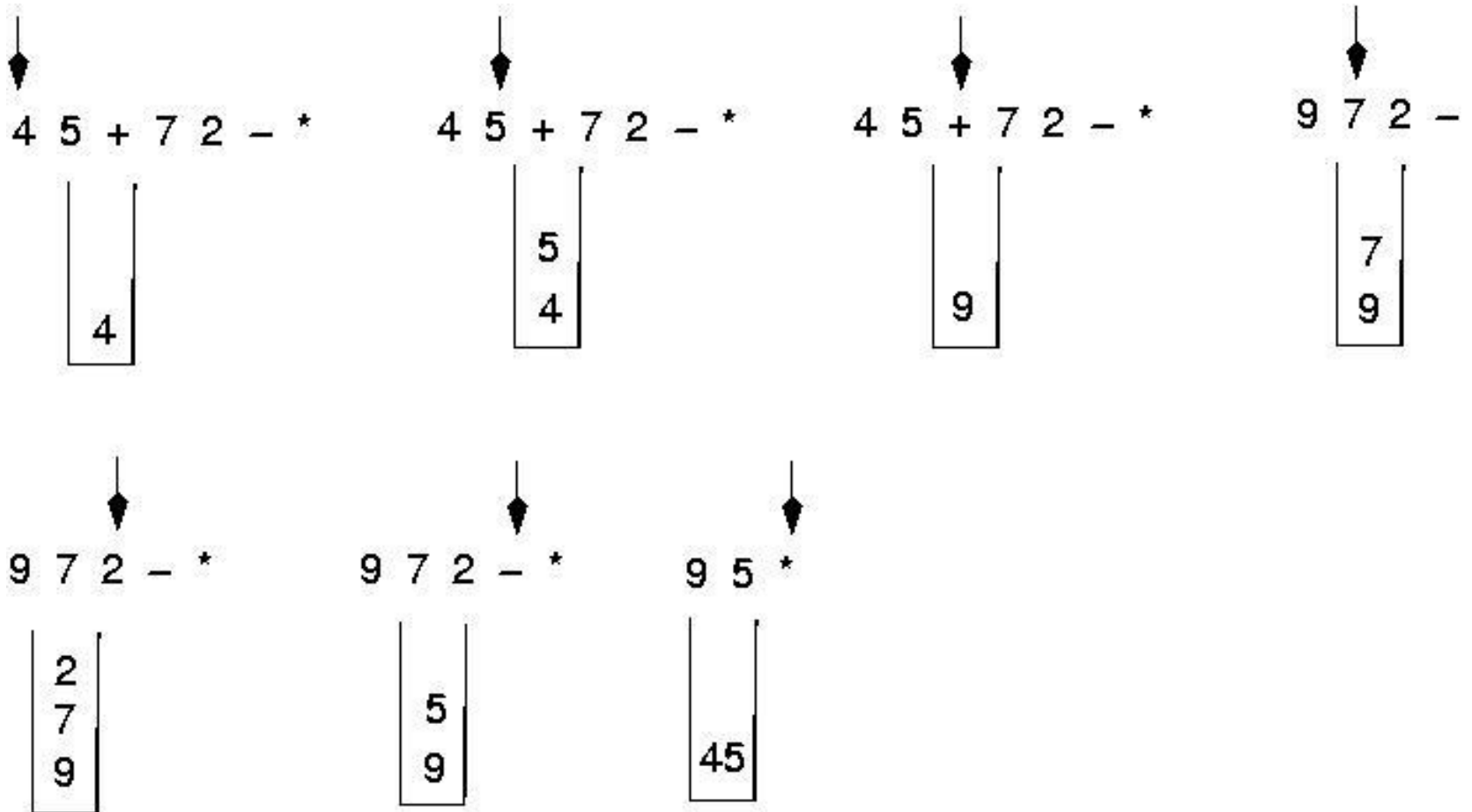
infix: 2+5 postfix: 2 5 +

- Expressions are evaluated from left to right.
- Precedence rules and parentheses are never needed!!

Example: postfix expressions (cont.)



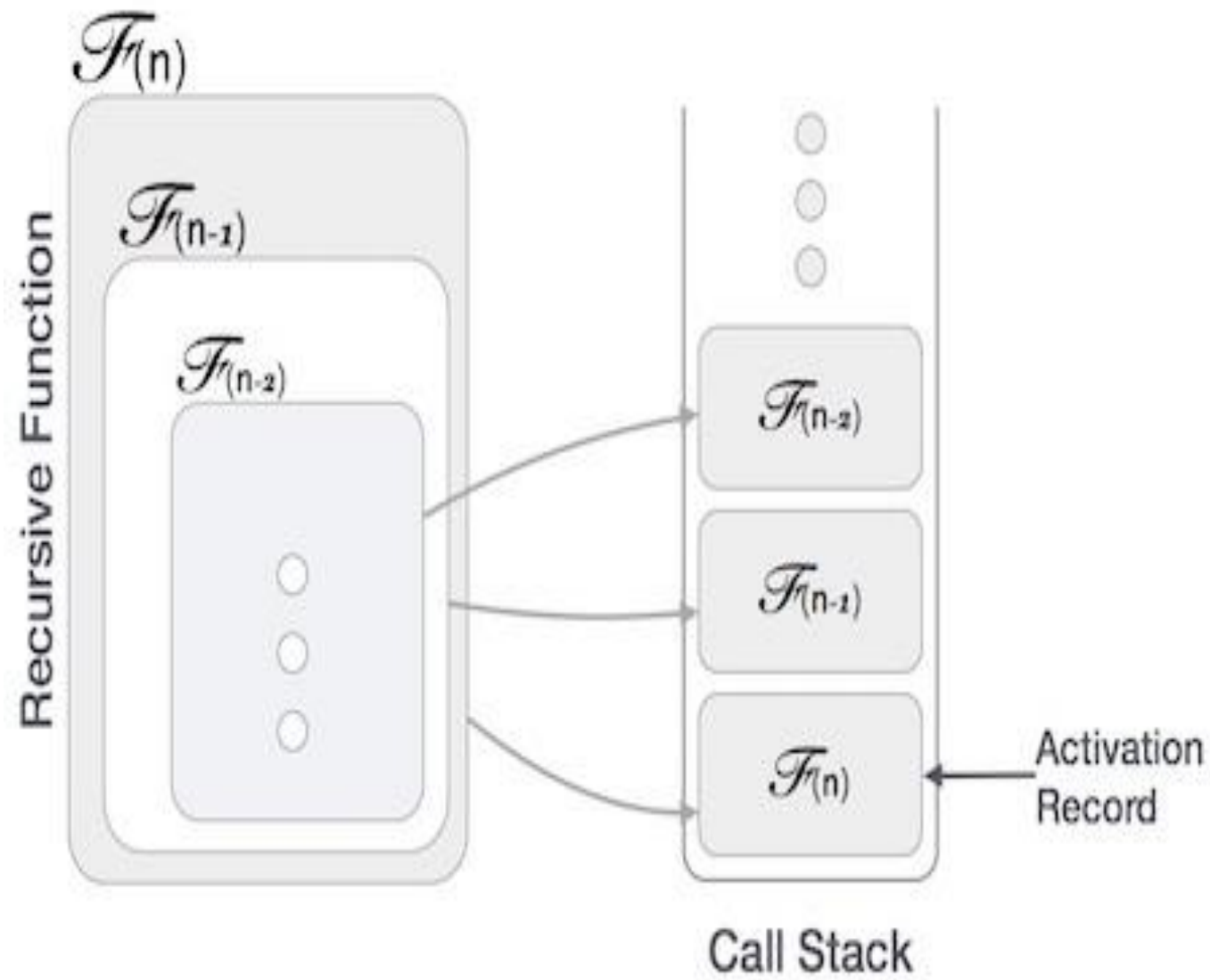
Postfix expressions: Algorithm using stacks (cont.)



Recursive

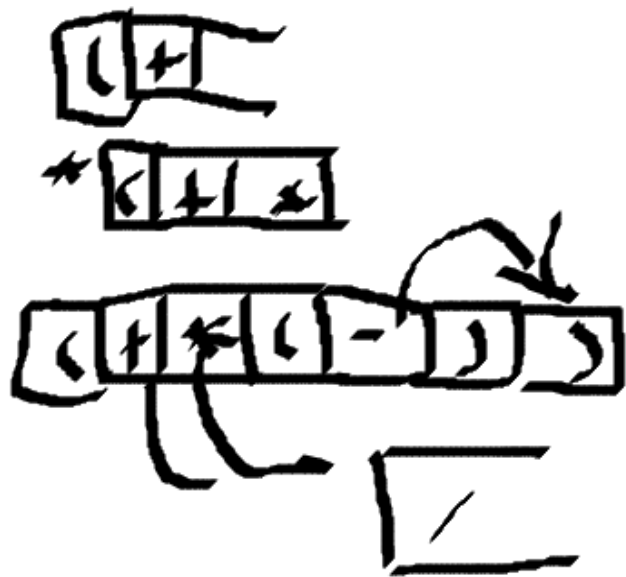
Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

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

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C *

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



A
A
AB
AB
AB
ABC
ABC
ABCD
ABCD-
ABCD-*+
ABCD-*+E
ABCD-*+E/

INFIX TO PREFIX CONVERSION

STEP1: REVERSE THE INFIX STRING .MUST EXCHANGE LEFT AND RIGHT PARENTHESSES.


STEP 2: OBTAIN THE POSTFIX EXPRESSION OF THE INFIX EXPRESSION
OBTAIN IN STEP 1

STEP3: REVERSE THE POSTFIX EXPRESSION TO GET THE PREFIX
EXPRESSION

EXAMPLE: $(A+B)*(C-D)$

STEP1: $(D-C)*(B+A)$

STEP2: DC- BA+*

STEP3: $*+AB-CD$  PREFIX NOTATION

QUEUES-CHAPTER-5

OUT COMES:

- WHAT IS QUEUE?
- OPERATIONS ON QUEUES
- UNDERSTANDING HOW TO IMPLEMENTING THE QUEUE
- TYPES OF QUEUE
- APPLICATIONS OF QUEUE



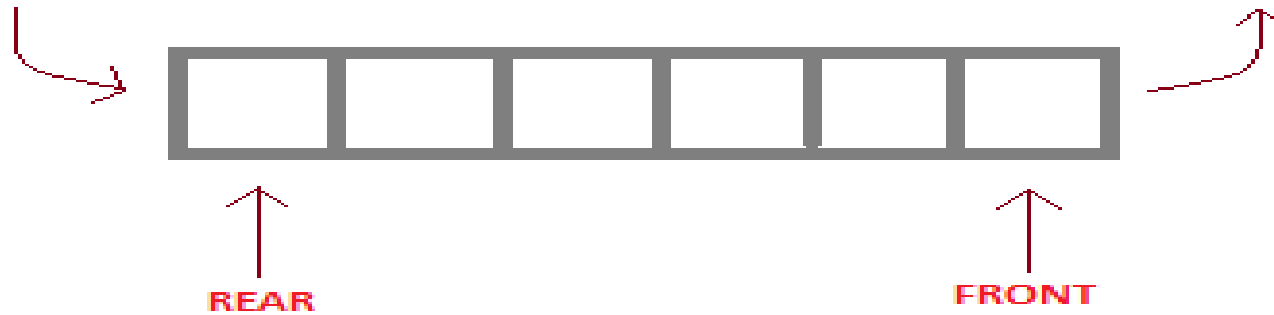
QUEUES

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



enqueue() operation

dequeue() operation



enqueue() is the operation for adding an element into Queue.

dequeue() is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Front=-1 Queue is empty

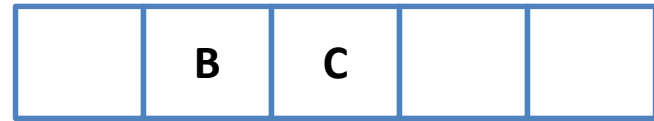
1



0 1 2 3 4
Rear=-1

Front=1 Delete

5



0 1 2 3 4
Rear=2

Front=0 Insert A

2



0 1 2 3 4
Rear=0

Front=2 Delete

6



0 1 2 3 4
Rear=2

Front=0 Insert B

3



0 1 2 3 4
Rear=1

Front=3 Delete

7



0 1 2 3 4
Rear=2
Queue is empty

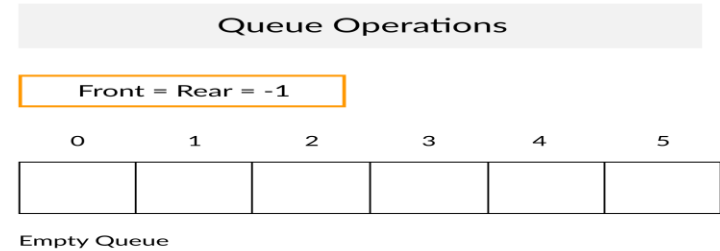
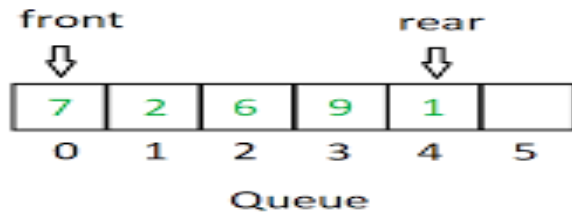
Front=0 Insert C

4



0 1 2 3 4
Rear=2

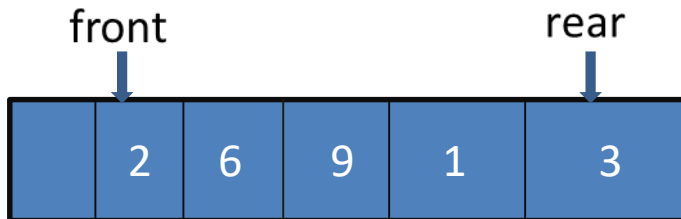
**Entry point is called Rear &
Exit point is called Front**



Enqueuing 3



Dequeuing



OPERATIONS :

1. ENQUEUE

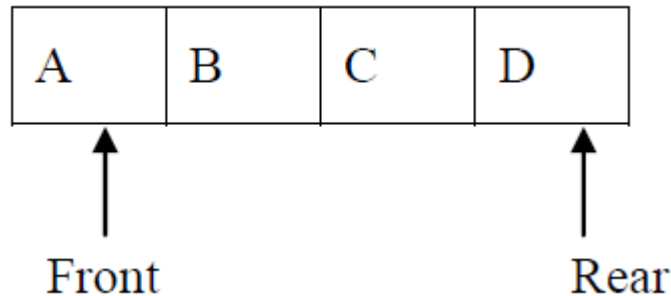
Adds an element at the beginning of the queue. If the queue is full, then it is an overflow.

2. DEQUEUE

Deletes an element at the end of the queue. If the queue is empty, then it is an underflow.

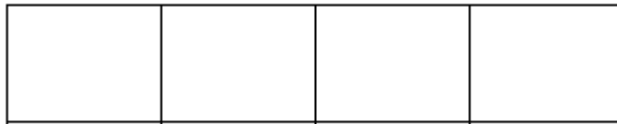
'Queue Full(Overflow)' Condition

- **Queue Full(Overflow):**
 - Inserting an element in a queue which is already full is known as Queue Full condition ($\text{Rear} = \text{Max}-1$).
 - When the queue is fully occupied and enqueue() operation is called queue overflow occurs.
- **Example: Queue Full:**
 - Before inserting an element in queue 1st check whether space is available for new element in queue. This can be done by checking position of rear end. Array begins with 0th index position & ends with max-1 position. If numbers of elements in queue are equal to size of queue i.e. if rear end position is equal to max-1 then queue is said to be full. **Size of queue = 4**



'Queue Empty(Underflow)' Condition

- **Queue Empty:**
 - Deleting an element from queue which is already empty is known as Queue Empty condition ($\text{Front} = \text{Rear} = -1$)
 - When the queue is fully empty and `dequeue()` operation is called queue underflow occurs.
 -
- **Queue Empty:**
 - Before deleting any element from queue check whether there is an element in the queue. If no element is present inside a queue & front & rear is set to -1 then queue is said to be empty.
 - **Size of queue = 4**
 - $\text{Front} = \text{Rear} = -1$



Disadvantages of linear queue

- On deletion of an element from existing queue, front pointer is shifted to next position.
- This results into virtual deletion of an element.
- By doing so memory space which was occupied by deleted element is wasted and hence inefficient memory utilization is occur.

Types of Queues in Data Structure

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Dequeue (Double Ended Queue)

Simple Queue

The simple queue is a normal queue where insertion takes place at the FRONT of the queue and deletion takes place at the END of the queue.

enqueue()

1. Check if the queue is full.
2. If the queue is full, then print "Queue overflow".
3. Else increment REAR by 1.
4. Assign $QUEUE [REAR] = ELEMENT$

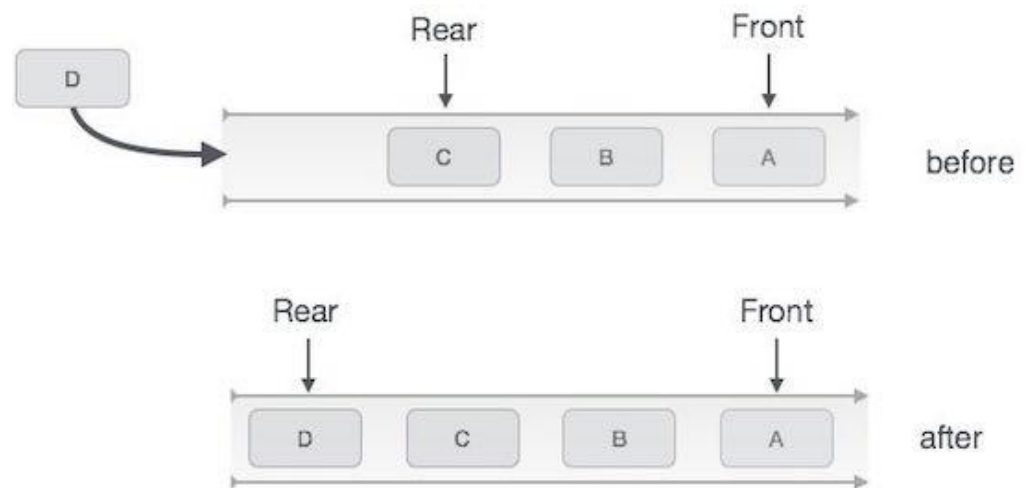
Algorithm for enqueue operation

procedure enqueue(data)

if queue is full
return overflow
endif

rear \leftarrow rear + 1
queue[rear] \leftarrow data
return true

end procedure



Queue Enqueue

dequeue()

1. Check if the queue is empty.
2. If the queue is empty, the print "Queue underflow".
3. Copy the element at the front of the queue to some temporary variable, $TEMP = QUEUE[FRONT]$.
4. Increment FRONT by 1.
5. Print temp and delete it.

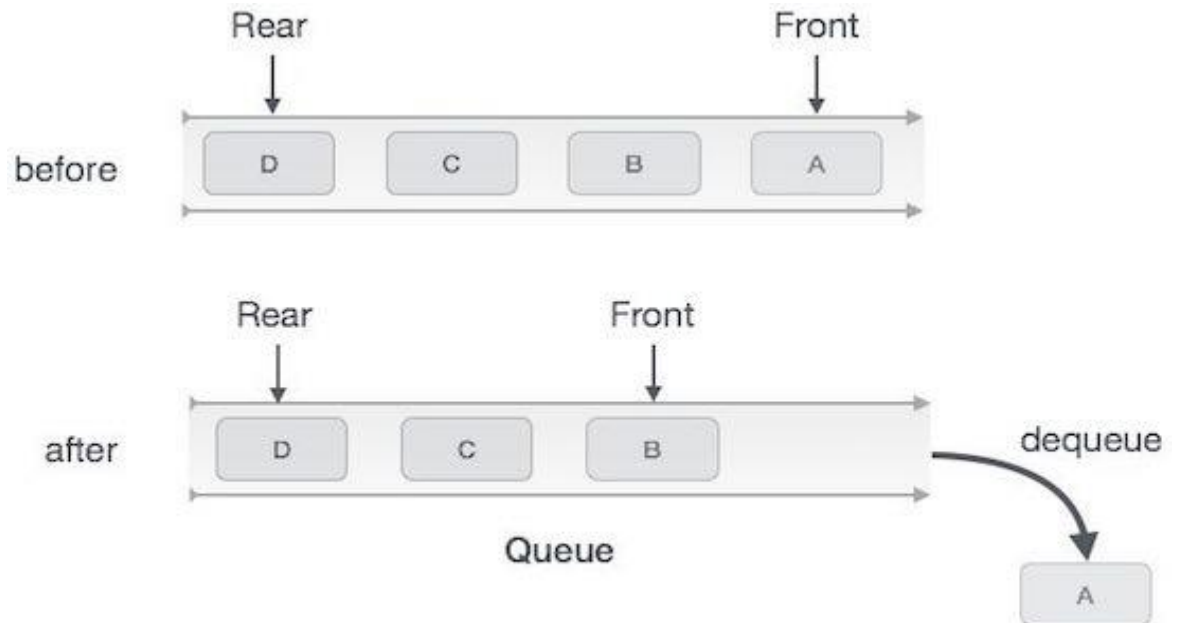
Algorithm for dequeue operation

procedure dequeue

if queue is empty
return underflow
end if

data = queue[front]
front ← front + 1
return true

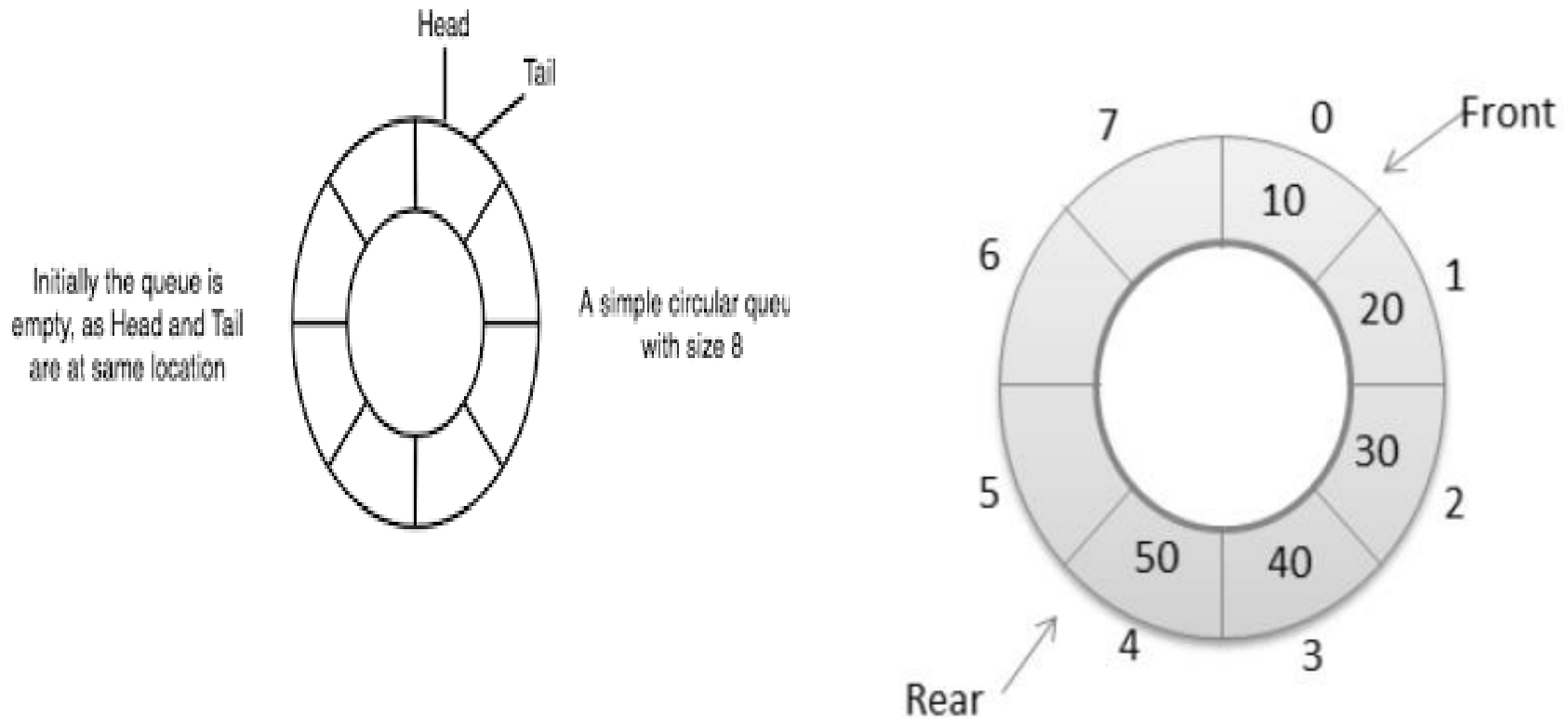
end procedure



Queue Dequeue

Circular Queue

In a circular queue, the last node is connected to the first node. Circular queue is also called as Ring Buffer. Insertion in a circular queue happens at the FRONT and deletion at the E



CIRCULAR QUEUE

- A queue, in which the last node is connected back to the first node to form a cycle, is called as circular queue.
- Circular queue are the queues implemented in circular form rather than in a straight line.
- Circular queues overcome the problem of unutilized space in linear queue implemented as an array.
- The main disadvantage of linear queue using array is that when elements are deleted from the queue, new elements cannot be added in their place in the queue, i.e. the position cannot be reused.

Priority Queue

In a priority queue, the nodes will have some predefined priority. Insertion in a priority queue is performed in the order of arrival of the nodes. The node having the least priority will be the first to be removed from the priority queue.



Structure of priority queue Node

Unit - III

Trees and Binary Trees

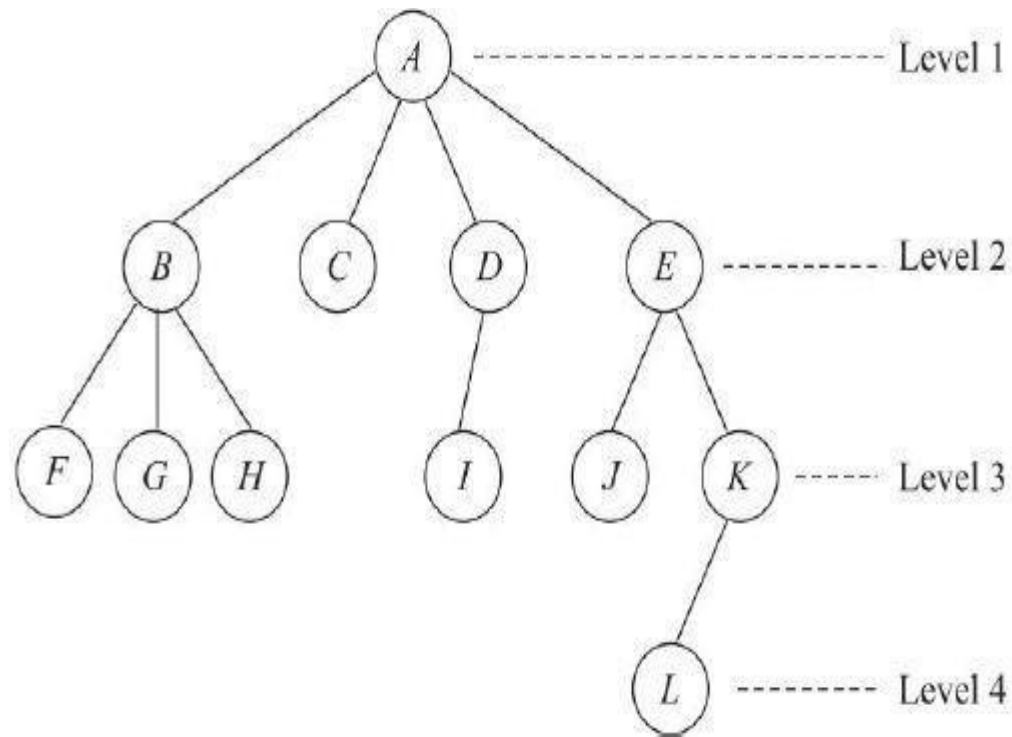
- The data structures of trees and graphs are termed non-linear data structures.
- Trees and their variants, binary trees and graphs, have emerged as truly powerful data structures registering immense contribution to the development of efficient algorithms or efficient solutions to various problems in science and engineering.

Trees: Definition and Basic Terminologies

Definition of trees

- A tree is defined as a finite set of one or more nodes such that
 - (i) there is a specially designated node called the root and
 - (ii) the rest of the nodes could be partitioned into t disjoint sets ($t \geq 0$) each set representing a tree T_i , $i = 1, 2, \dots, t$ known as subtree of the tree.
- A node is an item of information.
- The links between the nodes termed as branches, represent an association between the items of information.

- The definition of the tree emphasizes on the aspect of (i) connectedness and (ii) absence of closed loops or what are termed cycles.
- Beginning from the root node, the structure of the tree permits connectivity of the root to every other node in the tree.
- In general, any node is reachable from any where in the tree.
- Branches providing the links between the nodes.
- The structure ensures that no set of nodes link together to form a closed loop or a cycle.



Basic terminologies of trees

- There are several basic terminologies associated with the tree.
- The specially designated node called root.
- The number of subtrees of a node is known as the degree of the node.
- Nodes that have zero degree are called leaf nodes or terminal nodes.
- The rest of them are called as non terminal nodes.
- These nodes which hang from branches emanating from a node are known as children and the node from which the branches emanate is known as the parent node.

- Children of the same parent node are referred to as siblings.
- The ancestors of a given node are those nodes that occur on the path from the root to the given node.
- The degree of a tree is the maximum degree of the node in the tree.
- The level of a node is defined by letting the root to occupy level 1 .
- Thus if a parent node occupies level i , its children should occupy level $i+1$.
- This renders the tree to have a hierarchical structure with root occupying the top most level of 1.
- The height or depth of a tree is defined to be the maximum level of any node in the tree.
- depth of a node to be the length of the longest path

- A forest is a set of zero or more disjoint trees.
- The removal of the root node from a tree results in a forest (of its subtrees!).
- The degree of node E is 2 and L is 0. F, G, H, C, I, J and L are leaf or terminal nodes and all the remaining nodes are non leaf or non terminal nodes.
- Nodes F, G and H are children of B and B is a parent node.
- Nodes J, K and nodes F, G, H are sibling nodes with E and B as their respective parents.
- For the node L, nodes A, E and K are ancestors.
- The degree of the tree is 4 which is the maximum degree reported by node A. While node A which is the root node occupies level 1, its children B,C,D and E occupy level 2 and so on.
- The height of the tree is its maximum level which is 4.
- Removal of A yields a forest of four disjoint (sub)

Representation of Trees

- Though trees are better understood in their pictorial forms, a common representation of a tree to suit its storage in the memory of a computer, is a list.
- The tree could be represented in its list form as (A (B(F,G,H), C, D(I), E(J,K(L))))).
- The root node comes first followed by the list of subtrees of the node.
- This is repeated for each subtree in the tree.
- This list form of a tree, paves way for a naïve representation of the tree as a linked list.

Binary Trees: Basic Terminologies and Types

Basic terminologies

- A binary tree has the characteristic of all nodes having at most two branches, that is, all nodes have a degree of at most 2.
- A binary tree can therefore be empty or consist of a root node and two disjointed binary trees termed left subtree and right subtree.

Types of binary trees

- A binary tree of height h which has all its permissible maximum number of nodes viz., $2^{h+1} - 1$ intact is known as a full binary tree of height h .
- Note the specific method of numbering the nodes.
- A binary tree with n' nodes and height h is complete if its nodes correspond to the nodes which are numbered 1 to n ($n' \leq n$) in a full binary tree of height h .
- In other words, a complete binary tree is one in which its nodes follow a sequential numbering that increments from a left-to-right and top-to-bottom fashion.
- A full binary tree is therefore a special case of a complete binary tree.

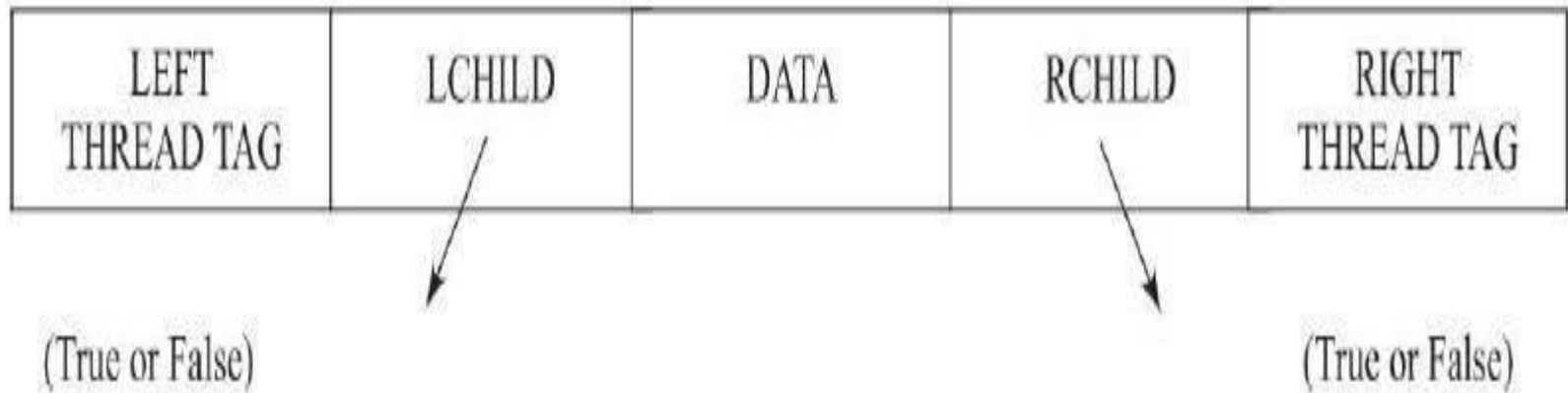
Binary Tree Traversals

- An important operation that is performed on a binary tree is its traversal.
- A traversal of a binary tree is where its nodes are visited in a particular but repetitive order, rendering a linear order of the nodes or information represented by them.
- A traversal is governed by three actions, viz. Move left (L), Move Right (R) and Process Node (P).
- In all, it yields six different combinations of LPR, LRP, PLR, PRL and RLP. Of these, three have
- emerged significant in computer science. They are,
- LPR 'Inorder traversal
- LRP 'Postorder traversal
- PLR 'Preorder traversal.

Inorder Traversal , Preorder, Postorder Traversal

- The traversal keeps moving left in the binary tree until one can move no further, processes the node and moves to the right to continue its traversal again.
- In the absence of any node to the right, it retracts backwards by a node and continues the traversal.
- The traversal proceeds by keeping to the left until it is no further possible, turns right to begin again or if there is no node to the right, processes the node and retraces its direction by one node to continue its traversal.
- The traversal processes every node as it moves left until it can move no further. Now it turns right to begin again or if there is no node in the right, retracts until it can move right to continue its traversal.

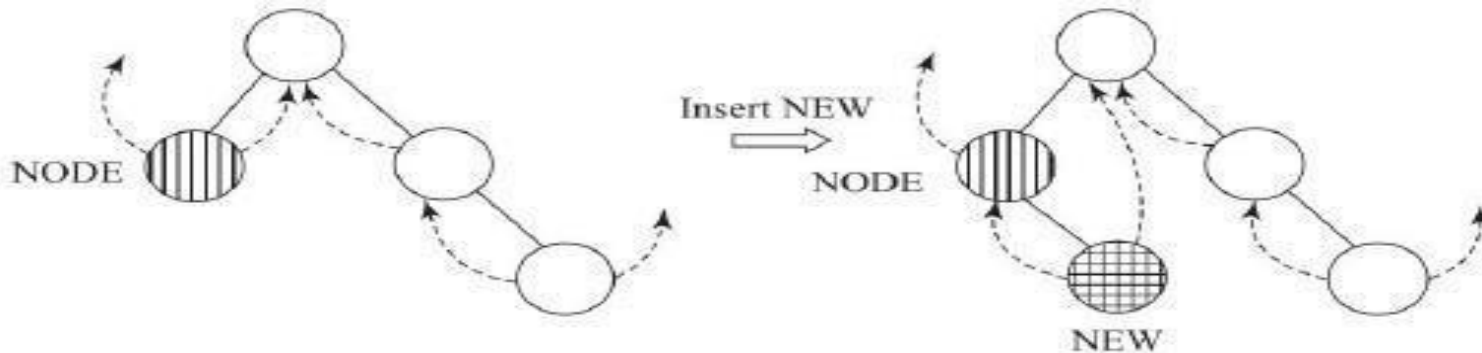
Linked representation of a threaded binary tree



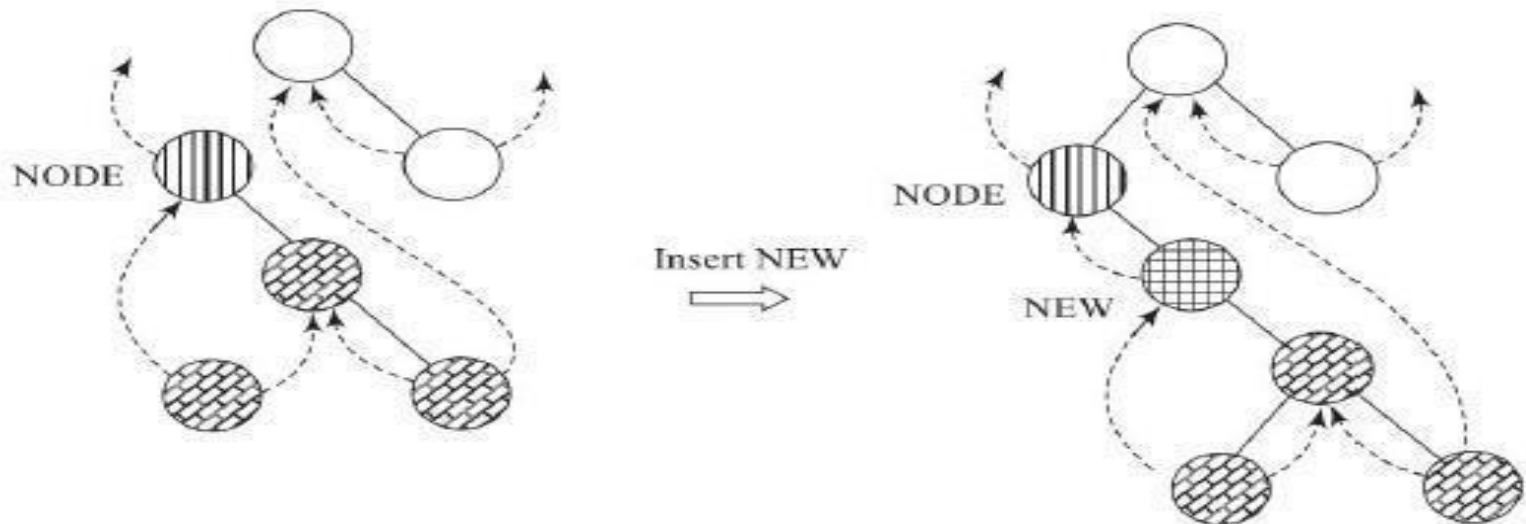
- If the LEFT THREAD TAG or RIGHT THREAD TAG is marked true then LCHILD and RCHILD fields represent threads otherwise they represent links.
- The linked representation of a threaded binary tree includes a head node. The dangling threads point to the head node.
- The head node by convention has its LCHILD pointing to the root node and therefore has its LEFT THREAD TAG set to false.
- The RIGHT THREAD TAG field is also set to false but the RCHILD link points to the head node itself.

Growing threaded binary

tree



(a) Insertion of node NEW to the right of NODE: Right subtree of NODE is empty



(b) Insertion of node NEW to the right of NODE: Right subtree of NODE is non-empty

That expressions are represented in three forms viz., infix, postfix and prefix.

An **infix expression** which is the commonly used representation of an expression follows the scheme

<operand> <operator> <operand>.

Examples are $A + B$, $A * B$.

Postfix expressions follow the scheme **< operand > < operand > < operator >.**

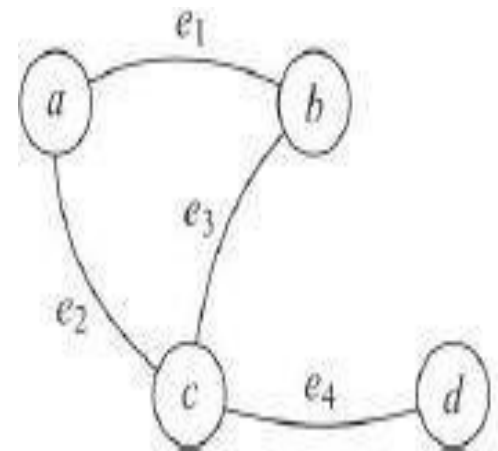
Examples are $AB+$, AB^* .

Prefix expressions follow the scheme **< operator > < operand > < operand >.**

Graph

A graph $G = (V, E)$ consists of a finite non empty set of vertices V also called points or nodes and a finite set E of unordered pairs of distinct vertices called edges or arcs or links.

Here $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, c), (b, c), (c, d)\}$.



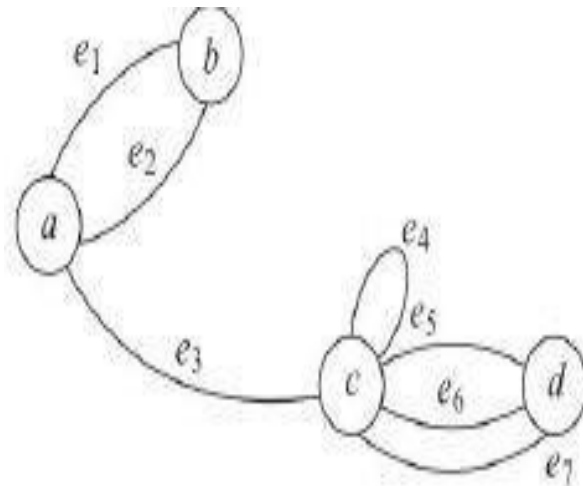
V : Vertices : $\{a, b, c, d\}$

E : Edges : $\{e_1, e_2, e_3, e_4\}$

A graph $G = (V, E)$ where $E = \emptyset$, is called as a null or empty graph. A graph with one vertex and no edges is called a trivial graph.

Multigraph

A multigraph $G = (V, E)$ also consists of a set of vertices and edges except that E may contain multiple edges (i.e.) edges connecting the same pair of vertices, or may contain loops or self edges.

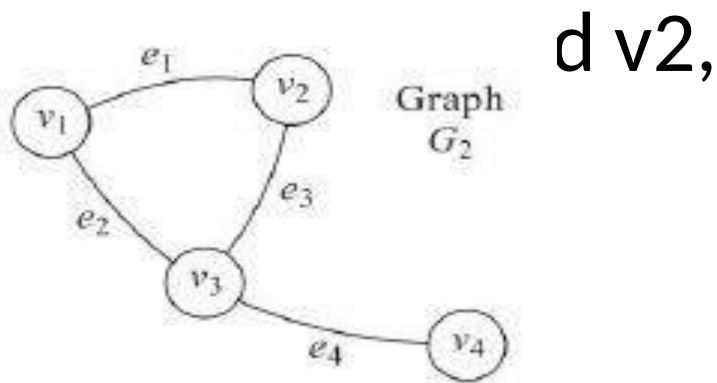


Directed and undirected graphs

A graph makes reference to unordered pairs of vertices as edges is known as an undirected graph.

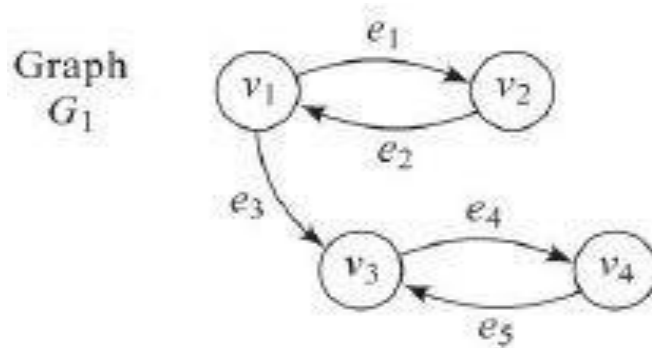
The edge e_{ij} of such an undirected graph is represented as (v_i, v_j) where v_i, v_j are distinct vertices. Thus an undirected edge (v_i, v_j) is equivalent to (v_j, v_i) .

e_1 is an undirected edge (i.e.) $e_1 = (v_1, v_2)$.



Directed and undirected graphs

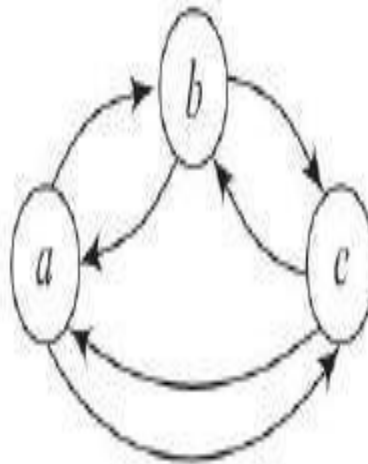
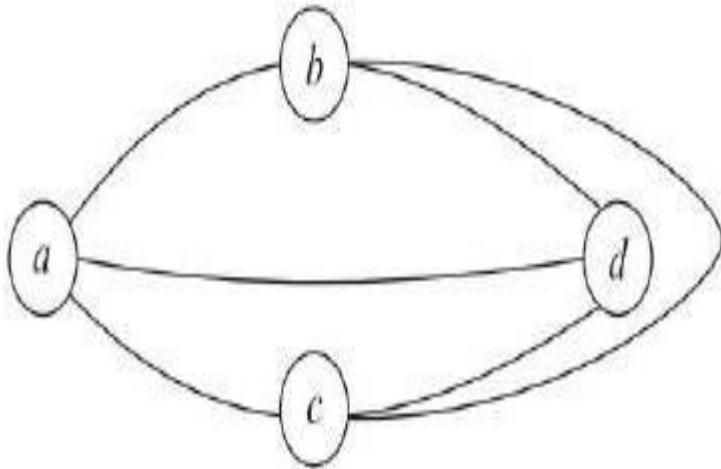
Directed graphs or digraphs make reference to edges which are directed (i.e.) edges which are ordered pairs of vertices.



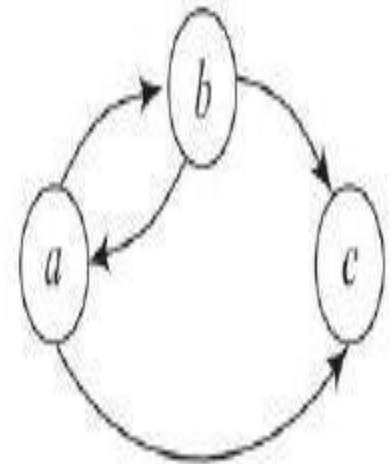
Complete graphs

An n vertex undirected graph with exactly $n \cdot (n-1) / 2$ edges is said to be complete.

Directed graph = $n \cdot (n-1)$



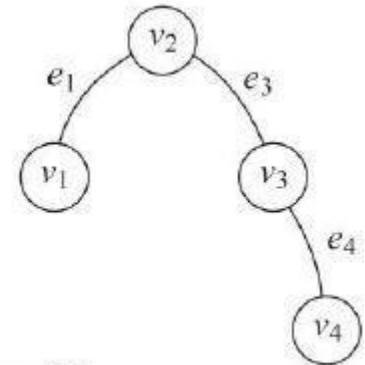
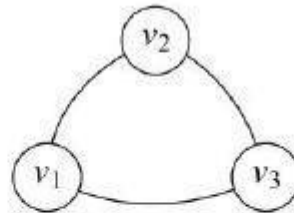
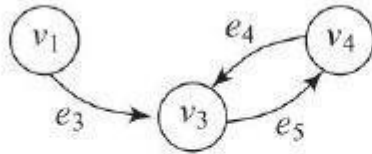
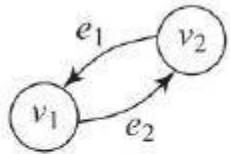
(a) Complete



(b) Not complete

Subgraph

A subgraph $G' = (V', E')$ of a graph $G = (V, E)$ is such that $V' \subseteq V$ and $E' \subseteq E$.



(a) Subgraphs of G_1

(b) Subgraphs of G_2

Path

A path from a vertex v_i to vertex v_j in an undirected graph G is a sequence of vertices.

A simple path is a path in which all the vertices except possibly the first and last vertices are distinct.

The length of a path is the number of edges on it.

A cycle is a simple path in which the first and last vertices are the same. A cycle is also known

Connected graphs

Two vertices v_i , v_j in a graph G are said to be connected only if there is a path in G between v_i and v_j .

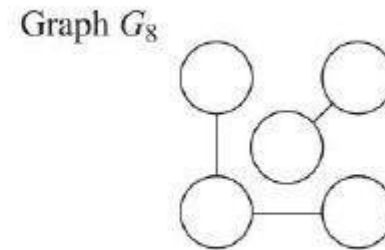
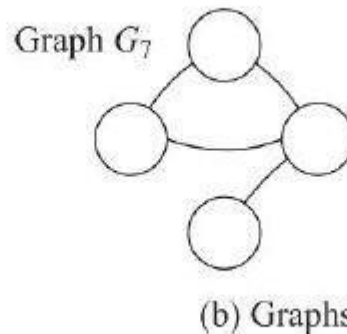
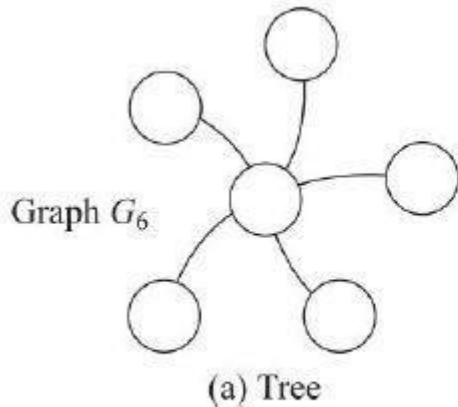
In an undirected graph if v_i and v_j are connected then it automatically holds that v_j and v_i are also connected.

An undirected graph is said to be a connected graph if every pair of distinct vertices v_i , v_j are connected.

Trees

A tree is defined to be a connected acyclic graph. The following properties are satisfied by a tree:

- (i) There exists a path between any two vertices of the tree, and
- (ii) No cycles must be present in the tree. In other words



Graphs which are trees and not trees

Degree

The degree of a vertex in an undirected graph is the number of edges incident to that vertex.

A vertex with degree one is called as a pendant vertex or end vertex.

A vertex with degree zero and hence has no incident edges is called an isolated vertex.

Hamiltonian circuit

A Hamiltonian circuit in a connected graph is defined as a closed walk that traverses every vertex of G exactly once, except of course the starting vertex at which the walk terminates.

A circuit in a connected graph G is said to be Hamiltonian if it includes every vertex of G .

Representations of Graphs

The representation of graphs in a computer can be categorized as (i) sequential representation and (ii) linked representation.

Sequential representation of graphs:

The sequential or the matrix representation of graphs have the following methods:

- (i) Adjacency matrix representation
- (ii) Incidence matrix representation
- (iii) Circuit matrix representation
- (iv) Cut set matrix representation
- (v) Path matrix representation

Graph Traversals

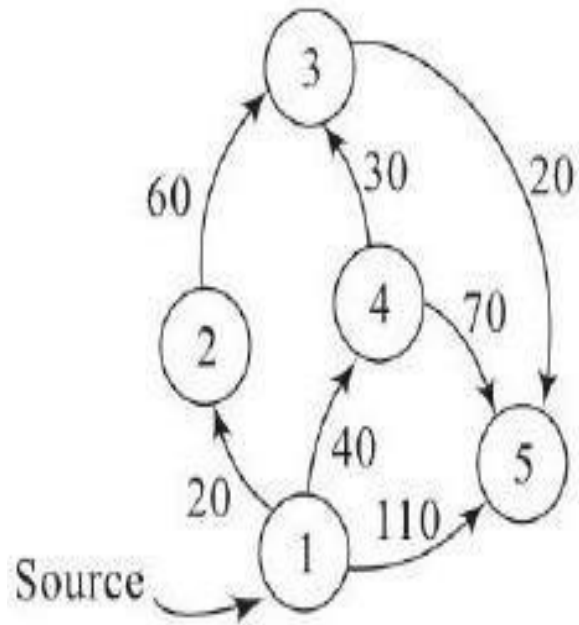
Graphs support the following traversals:

- **Breadth first Traversal**-The Breadth First Search (**BFS**) traversal is an algorithm, which is used to visit all of the nodes of a given graph. In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one.
- **Depth first Traversal**-Depth-first search (**DFS**) is an algorithm for traversing or searching tree or graph **data structures**. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Single-source, shortest-path problem

Given a network of cities and the distances between them, the objective of the single-source, shortest-path problem is to find the shortest path from a city (termed source) to all other cities connected to it.

The network of cities with their distances is represented as a weighted digraph.



(a) Weighted digraph

Cost:

	1	2	3	4	5
1	0	20	∞	40	110
2	∞	0	60	∞	0
3	∞	∞	0	∞	20
4	∞	∞	30	0	70
5	∞	∞	∞	∞	0

(b) Cost matrix $C_{5 \times 5}$

Thus the shortest path between vertex 1 and vertex 5 is 1 ' 4 ' 3 ' 5 and the distance is given

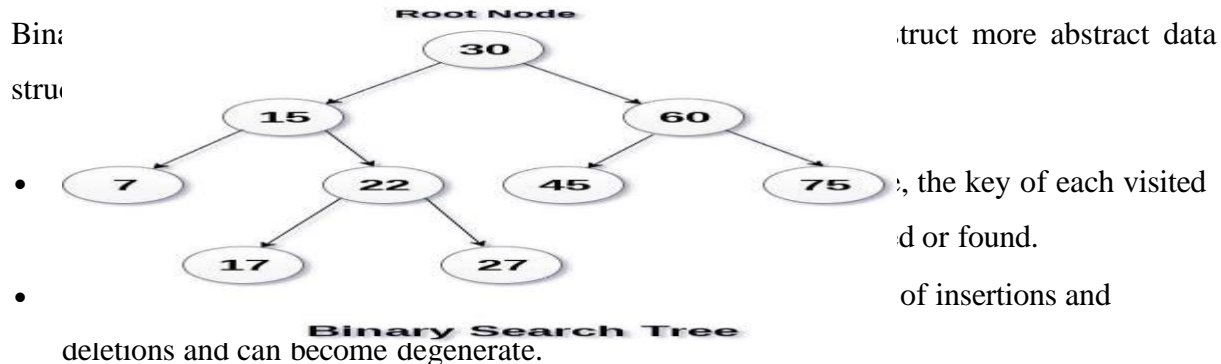
by $DISTANCE[5]$ is 90.

UNIT - IV

BINARY SEARCH TREE:

A binary search tree is a [rooted binary tree](#), whose internal nodes each store a key (and optionally, an associated value), and each has two distinguished sub-trees, commonly denoted *left* and *right*. The tree additionally satisfies the [binary search](#) property: the key in each node is greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree. The leaves (final nodes) of the tree contain no key and have no structure to distinguish them from one another.

Often, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records. The major advantage of binary search trees over other data structures is that the related [sorting algorithms](#) and [search algorithms](#) such as [in-order traversal](#) can be very efficient.



Operations in Binary search Tree:

Binary search trees support three main operations: insertion of elements, deletion of elements, and lookup (checking whether a key is present).

Searching

Searching in a binary search tree for a specific key can be programmed recursively or iteratively. We begin by examining the [root node](#). If the tree is *null*, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left

```
Algorithm: Procedure to retrieve ITEM from a binary search tree T
Procedure FIND – BST (T, ITEM, LOC)
/* Loc is the address of the node containing Item which is to be retrieved from the binary
search tree T. In case of unsuccessful search the procedure prints the message “ITEM not
found” and returns loc as NIL*/
If T= NIL then { Print {“ binary search tree T is empty”}
                Exit;}
Else
Loc = T;
While { LOC ≠ NIL} do
  Case:
    :ITEM =DATA (LOC) : return(LOC);
    :ITEM =DATA (LOC) : LOC=LCHILD(LOC);
Insertion
    : ITEM =DATA (LOC) : LOC=RCHILD(LOC);
  End case
End while
```

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before. Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'newNode') as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key

```
void insert(Node*& root, int key, int value) {
```

```
  if (!root)
```

```
    root = new Node(key, value);
```

```
  else if (key == root->key)
```

```
    root->value = value;
```

```
  else if (key < root->key)
```

```
    insert(root->left, key, value);
```

Deletion

```
  else // key > root->key
```

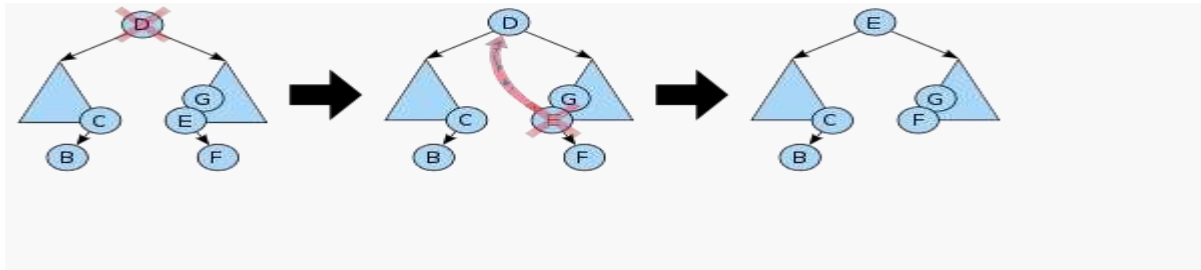
```
    insert(root->right, key, value);
```

When removing a node from a binary *search* tree it is mandatory to maintain the in-order

sequence of the nodes. There are many possibilities to do this. However, the following

method which has been proposed by T. Hibbard in 1962^[4] guarantees that the heights of the subject subtrees are changed by at most one. There are three possible cases to consider:

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted *D*. Do not delete *D*. Instead, choose either its [in-order](#) predecessor node or its in-order successor node as replacement node *E* (s. figure). Copy the user values of *E* to *D*.^[note 2] If *E* does not have a child simply remove *E* from its previous parent *G*. If *E* has a child, say *F*, it is a right child. Replace *E*



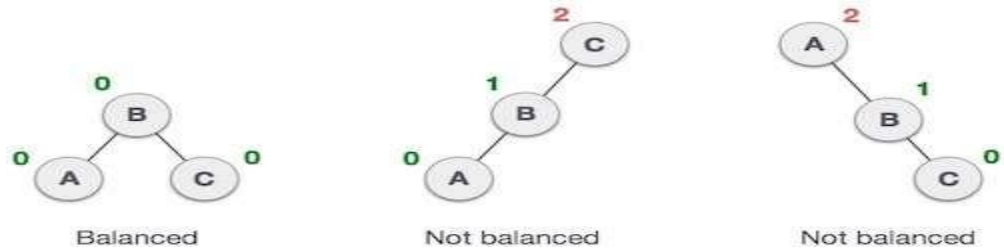
Deleting a node with two children from a binary search tree. First the leftmost node in the right subtree, which in this case is the node E, is identified. Its value is copied into the node D being deleted. The in-order successor can then be easily deleted because it has at most one child. Nodes with two children are harder to delete. A node's in-order successor is its right subtree's left-most child, and a node's in-order predecessor is the left subtree's right-most child. In either case, this node will have only one or no child at all. Delete it according to one of the two simpler cases above.

Consistently using the in-order successor or the in-order predecessor for every instance of the two-child case can lead to an unbalanced tree, so some implementations select one or the other at different times.

Runtime analysis: Although this operation does not always traverse the tree down to a leaf, this is always a possibility; thus in the worst case it requires time proportional to the height of the tree. It does not require more even when the node has two children, since it still follows a single path and does not visit any node twice.

AVL TREES:

AVL trees were proposed by **Adelson, Velski & Landis** in 1962. **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-



In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

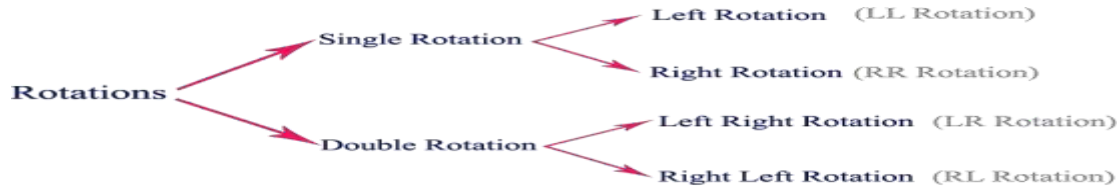
Balance Factor = height(left-subtree) – height(right-subtree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques. $|h(T_L) - h(T_H)| \leq 1$ where $h(T_L)$ and $h(T_H)$ are the heights of the left subtree and right subtree of T respectively.

Insertion into an AVL search tree:

The insertion of an element u into an AVL search tree, proceeds exactly as one would to insert u in a binary search tree. However, if after insertion the balance factors of any of the nodes turns out to be anything other than 0 or +1 or -1, then the tree is said to be unbalanced. To balance itself, an AVL tree may perform the following four kinds of rotations

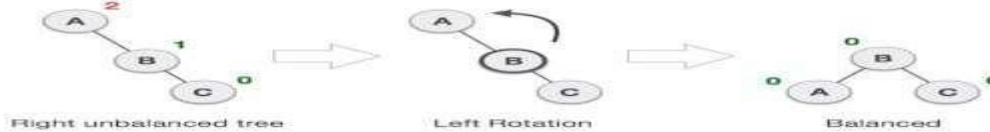
- Left rotation
- Right rotation
- Left-Right rotation



The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

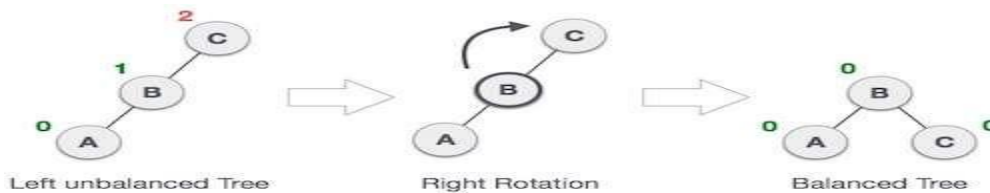
Left Rotation

If a tree becomes unbalanced when a node is inserted into the right subtree of the right



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of **B**.

Right Rotation

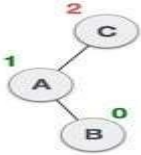
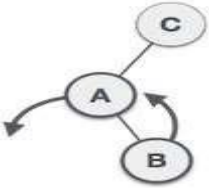


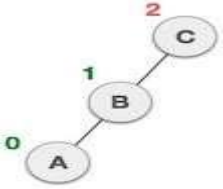
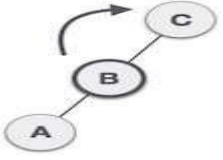
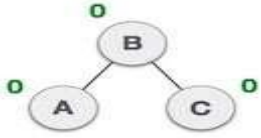
e left subtree.

As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

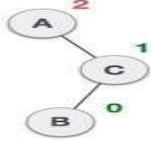
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

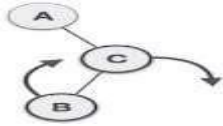
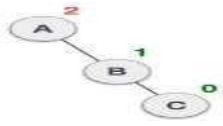
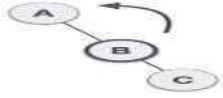
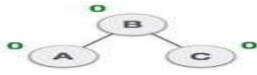
State	Action
	<p>A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>
	<p>We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.</p>

	<p>Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
	<p>We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.</p>
	<p>The tree is now balanced.</p>

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right

State	Action
	<p>A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.</p>

	<p>First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.</p>
	<p>Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.</p>
	<p>A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.</p>
	<p>The tree is now balanced.</p>

Operations on an AVL Tree

The following operations are performed on AVL tree...

1. Search
2. Insertion
3. Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with $O(\log n)$ time complexity. The search operation in the AVL tree is similar to the search operation in a Binary search tree. We use the following steps to search an element in AVL tree...

- **Step 3** - If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4** - If both are not matched, then check whether search element is smaller or larger than that node value.
- **Step 5** - If search element is smaller, then continue the search process in left subtree.
- **Step 6** - If search element is larger, then continue the search process in right subtree.
- **Step 7** - Repeat the same until we find the exact element or until the search element is compared with the leaf node.
- **Step 8** - If we reach to the node having the value equal to the search value, then display "Element is found" and terminate the function.
- **Step 9** - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

Insertion Operation in AVL Tree

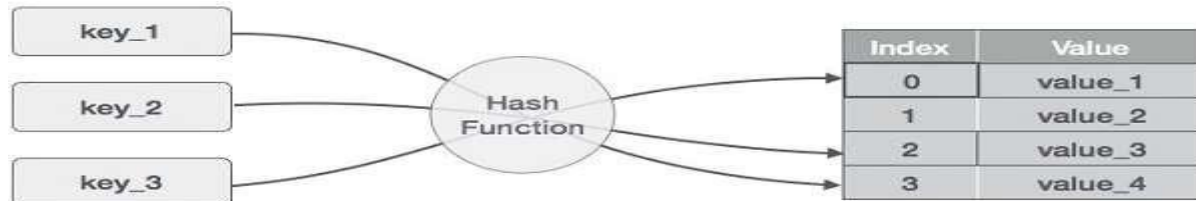
In an AVL tree, the insertion operation is performed with **$O(\log n)$** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

- **Step 1** - Insert the new element into the tree using Binary Search Tree insertion logic.
- **Step 2** - After insertion, check the **Balance Factor** of every node.
- **Step 3** - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- **Step 4** - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and

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

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

Hashing



format.

- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)

Sr.No.	Key	Hash	Array Index
• (13,78)	1	1 % 20 = 1	1
• (37,98)			

2	2	$2 \% 20 = 2$	2
3	42	$42 \% 20 = 2$	2
4	4	$4 \% 20 = 4$	4
5	12	$12 \% 20 = 12$	12
6	14	$14 \% 20 = 14$	14
7	17	$17 \% 20 = 17$	17
8	13	$13 \% 20 = 13$	13
9	37	$37 \% 20 = 17$	17

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	$1 \% 20 = 1$	1	1
2	2	$2 \% 20 = 2$	2	2

3	42	$42 \% 20 = 2$	2	3
4	4	$4 \% 20 = 4$	4	4
5	12	$12 \% 20 = 12$	12	12
6	14	$14 \% 20 = 14$	14	14
7	17	$17 \% 20 = 17$	17	17
8	13	$13 \% 20 = 13$	13	13
9	37	$37 \% 20 = 17$	17	18

What is Collision?

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

How to handle Collisions?

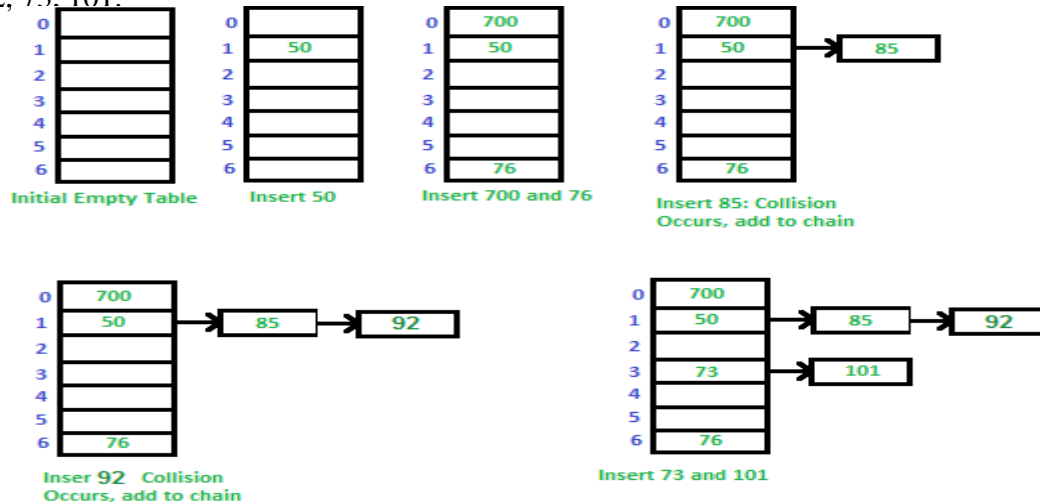
There are mainly two methods to handle collision:

1) Separate Chaining

2) Open Addressing
Separate Chaining:

In this article, only separate chaining is discussed. We will be discussing Open addressing in the next post.
 The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

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



Advantages:

- 1) Simple to implement.
- 2) Hash table never fills up, we can always add more elements to the chain.
- 3) Less sensitive to the hash function or load factors.
- 4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

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

UNIT-V

SEARCHING

Searching:

Searching are designed to check for an element or retrieve an element from any data structure where its stored. There are two types.

1. Sequential.

2. Interval.

1. Sequential:

In this the list or array is traversed sequentially and every element is checked.

Ex. Linear search.

2. Interval:

These type of searching algorithms are much more efficient than linear search as they repeatedly target the center of the search structure and divide the search space in half

Ex. Binary search

Linear Search

Linear search is the simplest searching algorithm that searches for an element in a list in sequential order. We start at one end and check every element until the desired element is not found.

Let's consider the following array to understand the working of the algorithm.

How Linear Search Works?

- A linear search scans one item at a time
- The worst case complexity is $O(n)$
- Time taken to search elements keep increasing as the number of elements are increased.

11	21	30	54	92	90	87	17
----	----	----	----	----	----	----	----

Now, suppose we want to search 92 in the above-mentioned array, the linear search algorithm shall follow the steps mentioned below.

Step 1: The algorithm begins from the left-hand side, and the element to be searched is matched

11	21	30	54	92	90	87	17
92							

Step 2: Now the algorithm moves to the next element and compares the two elements to check if matching happens.

11	21	30	54	92	90	87	17
	92						

Step 3: Similarly, the searching happens until no match happens.

11	21	30	54	92	90	87	17
		92					

11	21	30	54	92	90	87	17
			92				

Step 4: Finally, when the match happens, the algorithm returns the position of the element.

11	21	30	54	92	90	87	17
				92			

Ordered Linear search:

- Where the elements of a list have been already sorted, our search algorithm can be improved. Assuming the elements have been sorted in ascending order, the search operation can take advantage of the ordered nature of the list to make search more efficient.
- Let $L = \{k_1, k_2, k_3, k_4, \dots, k_n\}, k_1 < k_2 < k_3 < \dots < k_n$ be the list of ordered elements. To search for a key k in the list L , we undertake a linear search comparing k with each of the k_i so long as $k > k_i$ comparing k with the data elements of the list L progresses.

Algorithm:

Procedure LINEAR SEARCH ORDERED(L, n, k)

$I=0$;

While($(i < n)$ and $(k > L[i])$)do

$I=i+1$;

Endwhile

If($k=L[i]$) then

{ Ordered List of elements:

Print("key found");

Return(i);

}

$L[0,5]=\{16,18,56,78,90,100\}$

Let us search for the key $k=78$

Step1:	18	56	78	90	100
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

$k=78$

Step2:	18	56	78	90	100
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

$k=78$

Step3:	18	56	78	90	100
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

$k=78$

Step 4:	18	56	78	90	100
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

$k=78$

Finally key found ,Algorithm return the position of the element.

Unordered linear search:

In this search ,a key k is looked for in an ordered linear list $L=\{k_1,k_2,k_3,k_4,\dots,k_n\}$ of data elements.

Algorithm:

Procedure LINEAR SEARCH UNORDERED(L,n,k)

$i=0$;

While($(i<n)$ and $(L[i]!=k)$)do

$i=i+1$;

Endwhile

```

{
Print(“key found”);
Return(i);
}

```

Else

```
Print(“key not found”)
```

End LINEAR SEARCH UNORDERED.

UnOrdered List of elements:

23	14	98	45	67	53
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

L[0:5] = { 23, 14, 98, 45, 67, 53 }

Step 2: Search for the key k=53

23	14	98	45	67	53
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

Step 1: K=53

Step 3:

23	14	98	45	67	53
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

K=53

Step 4:

23	14	98	45	67	53
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

K=53

23	14	98	45	67	53
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

K=53

23	14	98	45	67	53
L[0]	L[1]	L[2]	L[3]	L[4]	L[5]

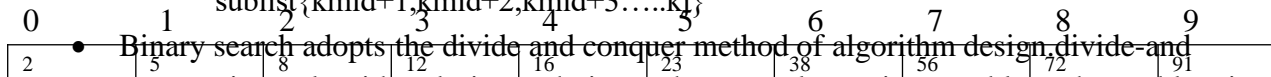
$k=53$

Finally key found ,Algorithm return the position of the element

Binary search:

- Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.
- A binary search searches for a key k in an ordered list $L=\{k_1,k_2,\dots,k_n\},k_1 < k_2 < k_3 < \dots < k_n$ of data elements, by halving the search list with each comparison until the key is either found or not found. The key k is first compared with the median element of the list viz...., k_{mid} . for a sub list $\{k_i,k_{i+1},k_{i+2},\dots,k_j\}, k_{mid}$ is obtained as the key occurring at the position mid which is computed as $mid = [(i+j)/2]$. The comparison of k with k_{mid} yields the following cases:
 - If $(k=k_{mid})$ then the binary search is done.
 - If $(k < k_{mid})$ then continue binary search in the sublist $\{k_i,k_{i+1},k_{i+2},\dots,k_{mid-1}\}$
 - If $(k > k_{mid})$ then continue binary search in the

sublist $\{k_{mid+1},k_{mid+2},k_{mid+3},\dots,k_j\}$



Binary search adopts the divide and conquer method of algorithm design, divide-and-conquer is an algorithm design technique where to solve a given problem ,the problem is

23 > 16 first recursively divided into subproblems. The subproblems that are small enough are easily solved and the solutions combined to obtain the solutions to the whole problem.

L=0	1	8	12	16	23	38	56	72	91
-----	---	---	----	----	----	----	----	----	----

Ex:

Consider an ordered list $L=\{k_1,k_2,k_3,\dots,k_{15}\}=\{2,5,8,12,16,23,38,56,72,91\}$

Search 23

23 > 56 take 1st half

		2	3	4	L=5	6	M=7	8	H=9
0	1	8	12	16	23	38	56	72	91

Found 23, return 5

		2	3	4	L=5, M=5	H=6	7	8	9
0	1	8	12	16	23	38	56	72	91

Algorithm:

Procedure binary_search(L, low, high, k)

If (low > high) then

{

Binary search = 0;

Print("key not found");

Exit();

}

Else

{

Mid = [low + high / 2]

Case:

K = L[mid];

{

Print("key found");

Binary search = 1; Mid = mid;

k>L[mid]:binary search=binary search(L,mid+1,high,k);

End case

}

End binary search

- Fibonacci Search examines closer elements in few steps. So when input array is big that cannot fit in CPU cache or in...

Fibonacci search:

- On average, fibonacci search requires 4% more comparisons than binary search
- Fibonacci search requires only addition and subtraction whereas binary search requires bit-shift, division or...

Fibonacci search can reduce the time needed to access an element in a random access principle memory that can find an element in the given **sorted array** with the help of Fibonacci series

- in $O(\log N)$ time complexity. This is based on Fibonacci series which is an infinite sequence of numbers denoting a pattern which is captured by the following equation:
 - $F(n+1)=F(n)+F(n-1)$
 - where F(i) is the ith number of the Fibonacci series where F(0) and F(1) are defined as 0 and 1 respectively.

- The first few Fibonacci numbers are:

0,1,1,2,3,5,8,13....

F(0) = 0

F(1) = 1

F(2) = F(1) + F(0) = 1 + 0 = 1

F(3) = F(2) + F(1) = 1 + 1 = 2

F(4) = F(3) + F(2) = 1 + 2 = 3 and so continues the series

120	10
-----	----

- binary search also work for the similar principle on splitting the search space to a smaller space but what makes Fibonacci search different is that it divides the array in **unequal parts** and operations involved in this search are **addition and subtraction** only which means light arithmetic operations takes place and hence **reducing the work load of the computing machine**

The element do be found be 100

According to the algorithm we will first sort the array.

10	15	30	60	90	100	120
----	----	----	----	----	-----	-----

 A=

Then check the value in Fibonacci series which is greater or equal to value of $n=7$

$7 \leq 8$

$fbM=8$ $fbM1=5$ $fbM2=3$ $offset=-1$

$i=2 // -1+3 < \min((offset+fbM2), n-1)$

60	90	100	120
----	----	-----	-----

A

$A[2]=30 < 100$

$fbM=5$ $fbM1=3$ $fbM2=2$ $offset=2$

$i=5//2+3 < 7 \min((offset+fbM2), n-1)$

Sort:

Sorting also referred to as ordering deals with arranging elements of a list or a set or records of a file in the ascending or descending order.

Bubble sort:

Bubble sort, sometimes referred to as **sinking sort**, is a simple [sorting algorithm](#) that 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. The algorithm, which is a [comparison sort](#), is named for the way smaller or larger elements "bubble" to the top of the list.

Algorithm:

Procedure for Bubble sort

For $i=1$ to $n-1$ do

For $j=1$ to $n-1$ do

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array. It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array.

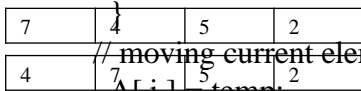
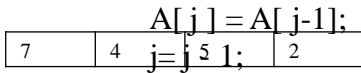
```
void insertion_sort ( int A[ ] , int n)
{
    for( int i = 0 ;i < n ; i++) {
        /*storing current element whose left side is checked for its
           correct position .*/

        int temp = A[ i ];
        int j = i;

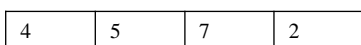
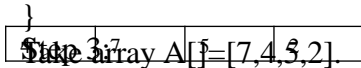
        /* check whether the adjacent element in left side is greater or
           less than the current element. */

        while( j > 0 && temp < A[ j -1]) {

            // moving the left side element to one position forward.
```



// moving current element to its correct position.
A[j] = temp; -----> As 7>4 therefore 7 will be moved forward and 4 will be moved to 7 th position.



-----> as 7>5 7 will be moved forward, but 4<5, so no change in

Step 1:

-----> no elements on left side of 7, so no change in its position

Step 2:

position of 4 and 5 will be moved to position of 7.

Step 4:	5	7	2
---------	---	---	---

2	4	5	7
---	---	---	---

-----> as all the elements on left side of 2 are greater than 2, so all the elements will be moved forward and 2 will be shifted to position of 4

Since 7 is the first element 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.

Time Complexity:

In worst case, each element is compared with all the other elements in the sorted array. For N elements, there will be N^2 comparisons. Therefore, the time complexity is $O(N^2)$

Selection sort:

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array. 1) The subarray which is already sorted. 2) Remaining subarray which is unsorted.

arr[] = 64 25 12 22 11

```
// Find the minimum element in arr[0...4]
```

```
// and place it at beginning
```

```
11 25 12 22 64
```

```
// Find the minimum element in arr[1...4]
```

```
// and place it at beginning of arr[1...4]
```

```
11 12 25 22 64
```

Merge sort:

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

- Divide the unsorted list into N sublists, each containing 1 element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.
- Repeat the process till a single sorted list of obtained.

MergeSort(arr[], l, r)

If $r > l$

1. Find the middle point to divide the array into two halves: middle $m = (l+r)/2$
2. Call mergeSort for first half:
Call mergeSort(arr, l, m)
3. Call mergeSort for second half: Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
Call merge(arr, l, m, r)

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \theta(n)$$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of Master Method and the solution of the recurrence is $\theta(n \log n)$. Time complexity of Merge Sort is $\theta(n \log n)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Applications of Merge Sort

1. [Merge Sort is useful for sorting linked lists in \$O\(n \log n\)\$ time.](#) In the case of linked lists, the case is different mainly due to the difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike an array, in the linked list, we can insert items in the middle in $O(1)$ extra space and $O(1)$ time. Therefore, the merge operation of merge sort can be implemented without extra space for linked lists.

Quick sort:

Like [Merge Sort](#), QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

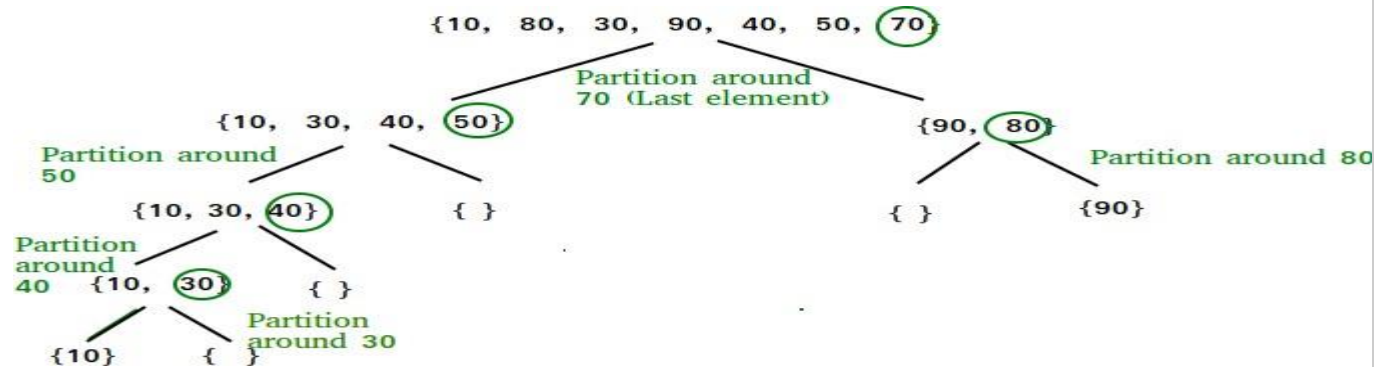
1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

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

Pseudo Code for recursive QuickSort function :

```
/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
        at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1); // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```



Partition Algorithm

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

/ low --> Starting index, high --> Ending index */*

quickSort(arr[], low, high)

{

 if (low < high)

 {

/ pi is partitioning index, arr[pi] is now
 at right place */*

 pi = partition(arr, low, high);

 quickSort(arr, low, pi - 1); // Before pi

 quickSort(arr, pi + 1, high); // After pi

 }

}

Pseudo code for partition()

```

of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1) // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}

```

Illustration of partition() :

arr[] = { 10, 80, 30, 90, 40, 50, 70 }

Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70

j = 2 : Since $\text{arr}[j] \leq \text{pivot}$, do $i++$ and $\text{swap}(\text{arr}[i], \text{arr}[j])$
i = 1
 $\text{arr}[] = \{10, 30, 80, 90, 40, 50, 70\}$ // We swap 80 and 30

j = 3 : Since $\text{arr}[j] > \text{pivot}$, do nothing
// No change in i and $\text{arr}[]$

j = 4 : Since $\text{arr}[j] \leq \text{pivot}$, do $i++$ and $\text{swap}(\text{arr}[i], \text{arr}[j])$
i = 2
 $\text{arr}[] = \{10, 30, 40, 90, 80, 50, 70\}$ // 80 and 40 Swapped
j = 5 : Since $\text{arr}[j] \leq \text{pivot}$, do $i++$ and $\text{swap}(\text{arr}[i], \text{arr}[j])$
i = 3
 $\text{arr}[] = \{10, 30, 40, 50, 80, 90, 70\}$ // 90 and 50 Swapped

We come out of loop because j is now equal to $\text{high}-1$.

Finally we place pivot at correct position by swapping $\text{arr}[i+1]$ and $\text{arr}[\text{high}]$ (or pivot)

$\text{arr}[] = \{10, 30, 40, 50, 70, 90, 80\}$ // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

Heap sort:

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining elements.

What is [Binary Heap](#)?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible

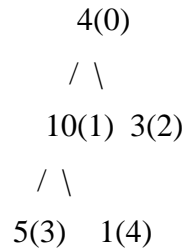
(Source [Wikipedia](#))

A [Binary Heap](#) is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min-heap. The heap can be represented by a binary tree or array.

heapification must be performed in the bottom-up order.

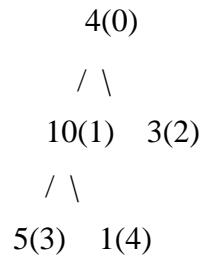
Lets understand with the help of an example:

Input data: 4, 10, 3, 5, 1

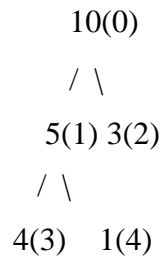


The numbers in bracket represent the indices in the array representation of data.

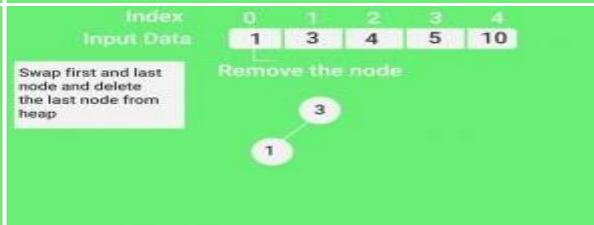
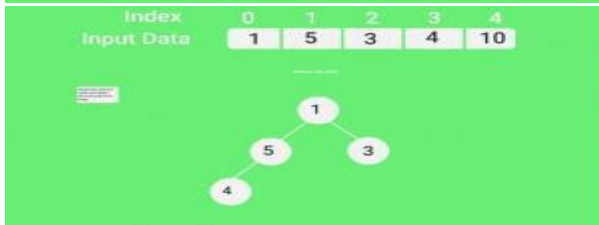
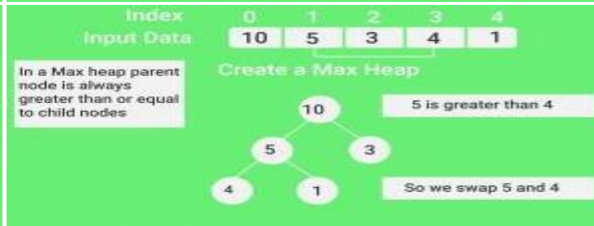
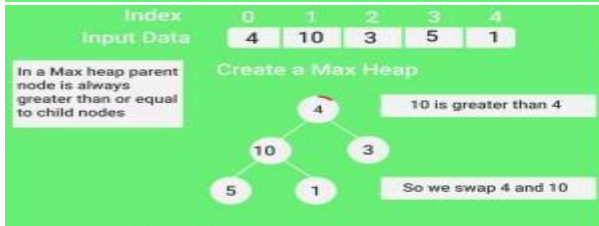
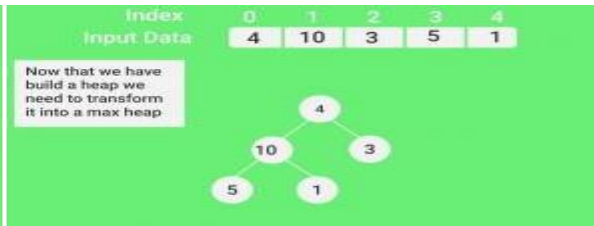
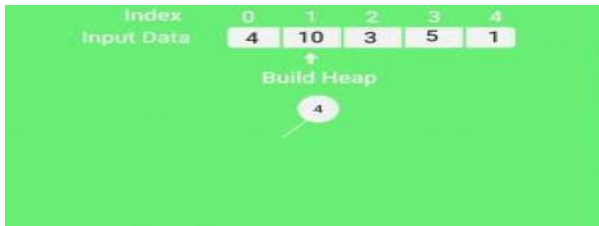
Applying heapify procedure to index 1:



Applying heapify procedure to index 0:



The heapify procedure calls itself recursively to build heap



THANK YOU