Corse Title: DATA STRUCTURE

Partho Sarathi Sarker
Asst. Professor: Dept. Of CSE

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

Data Structure (UGV) Format	
Course Code: CSE-0613-1205	Credits: 03
Exam Hours: 03	CIE Marks: 90
Course for 2nd Semester, Bachelor of Science in Computer Science Engineering (CSE)	SEE Marks: 60

Course Learning Outcome (CLOs): After Completing this course successfully, the student will be able to…

CLO1	Describe the fundamental concepts of data structures and their applications in various domains.
CLO2	Understand the basic and advanced data structures, and be able to analyse their time and space complexity.
CLO3	Create and implement basic and advanced data structures using programming languages like C++, Java, or Python.
CLO4	Apply appropriate data structures to solve real-world problems, such as file structures, symbol tables, and network data structures.
CLO5	Identify and select appropriate data structures and algorithms to solve a given problem based on its requirements and constraints.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

Summary of Course Content

Sl. No.	COURSE CONTENT	HRs	CLOs
1	Introduction to Data Structures: Overview of data structures, Types of data structures: linear and nonlinear, Arrays and linked lists, Stacks and queues, Trees and graphs	3	CLO1 CLO2
2	Overview and Types of data structures: Arrays and linked lists, Stacks and queues, Implementation of basic data structures, Operations on basic data structures, Time and space complexity analysis	4	CLO3
3	Arrays and linked lists: Sorting algorithms: bubble sort, insertion searching algorithms: linear search, binary search. Struct Variable, Link list Creation, Insertion, Deletion, searching, traversing	9	CLO4
4	Stacks and queues: push,pop , enque, dequue function definition, overflow, underflow condition, design stack and queue with array and analyse various condition	7	CLO4 CLO5
5	Nonlinear Data Structures: Trees, Type of Trees, Tree Representation, Graphs, Type of Graphs, Graph representation, BFS, DFS, Pre-order, in-order, post-order search.	7	CLO5

Recommended Books:

1. "Data Structures and Algorithm Analysis in C++" by Mark A. Weiss
2. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
3. "Data Structures Using C" by Reema Thareja

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

ASSESSMENT PATTERN

CIE- Continuous Internal Evaluation (90 Marks)

Bloom's Category Marks (out of 90)	Tests (45)	Assignments (15)	Quizzes (15)	Attendance (15)
Remember	5	03		
Understand	5	04	05	
Apply	15	05	05	
Analyze	10			
Evaluate	5	03	05	
Create	5			

SEE- Semester End Examination (60 Marks)

Bloom's Category	Test
Remember	7
Understand	7
Apply	20
Analyze	15
Evaluate	6
Create	5

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

week no	Topics	Teaching Learning Strategy(s)	Assessment Strategy(s)	Alignment to CLO
1	Introduction to Data Structures: Describe concepts, importance, and types of data structures	Lecture, multimedia, group discussion	Feedback, Q&A, assessment of LOs	CLO1
2	Operation on Data Structure, Time-Space Complexity, Algorithm, Array definition	Lecture, multimedia, practical examples	Feedback, Q&A, quizzes	CLO2
3	Arrays:Initialization, access, Types of Array, Array Addressing: Row major, Column Major	Lecture, multimedia, practical examples	Feedback, Q&A, quizzes	CLO2
4	Sorting and Searching Algorithms: Linear Search, Binary Search	Lecture, multimedia, practical examples	Feedback, Q&A, quizzes	CLO2
5	Sorting and Searching Algorithms: Selection Sort, Bubble Sort	Lecture, multimedia, hands-on practice	Midterm Quiz #1, assessment of LOs	CLO1
6	Sorting and Searching Algorithms: Insertion Sort, Quick Sort	Lecture, multimedia, hands-on practice		CLO2
7	Stacks and Queues, Array Implementation of Stack Push, Pop Operation of a Stack.	Lecture, multimedia, hands-on practice	Feedback, Q&A, lab assignments	CLO2, CLO3
8	Stacks and Queues Enqueue and Dequeue operation	Lecture, multimedia, hands-on practice	Feedback, Q&A, lab assignments	CLO2, CLO3

9	Link List Definition, Difference between array and Link List, Link list creation	Lecture, multimedia, hands-on practice	Feedback, Q&A, lab assignments	CLO2, CLO3
10	Linked Lists Operation: Create, traverse, search, insert, and delete operations in linked lists	Lecture, multimedia, hands-on practice	Feedback, Q&A, lab assignments	CLO2, CLO3
11	Linked Lists Operation: Create, traverse, search, insert, and delete operations in linked lists	Lecture, multimedia, hands-on practice	Feedback, Q&A, lab assignments	CLO2, CLO3
12	Trees: type, properties Understand and implement tree structures and their traversal	Lecture, multimedia, hands-on practice	Feedback, Q&A, lab assignments	CLO2, CLO3
13	Pre-order, in-order, post-order Representation Implement tree traversal techniques	Lecture, multimedia, hands-on practice	Midterm Quiz #2, lab assignments	CLO2-CLO5
14	Graphs: Representation and type	Lecture, multimedia, hands-on practice	Feedback, Q&A, lab assignments	CLO2-CLO5
15	Breadth First Search Algorithm(BFS), Application	Lecture, multimedia, hands-on practice	Final Exam, lab assignments	CLO2-CLO5
16	Depth First Search Algorithm(DFS), Application	Lecture, multimedia, hands-on practice	Final Exam, lab assignments	CLO2-CLO5
17	Review Class			

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 1

Introduction to Data Structures: Overview of data structures, Types of data structures: linear and nonlinear, Arrays and linked lists, Stacks and queues, Trees and graphs

DATA STRUCTUE

Let us start with questions:

1. What is Data?

2. What is Structure?

DATA STRUCTUE

Data means raw facts or information that can be processed to get results or products.

Some elementary items constitute a unit and that unit may be considered as a structure.

- A structure may be treated as a frame or pro-forma where we organize some elementary items in different ways.

DATA STRUCTURE

- **Data structure is a structure where we organize elementary data items in different ways and there exits structural relationship among the items.**
  - That means, a data structure is a means of structural relationships of elementary data items for storing and retrieving data in computer's memory.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

Data Structure [Cont..]

Usually elementary data items are the *elements* of a data structure.

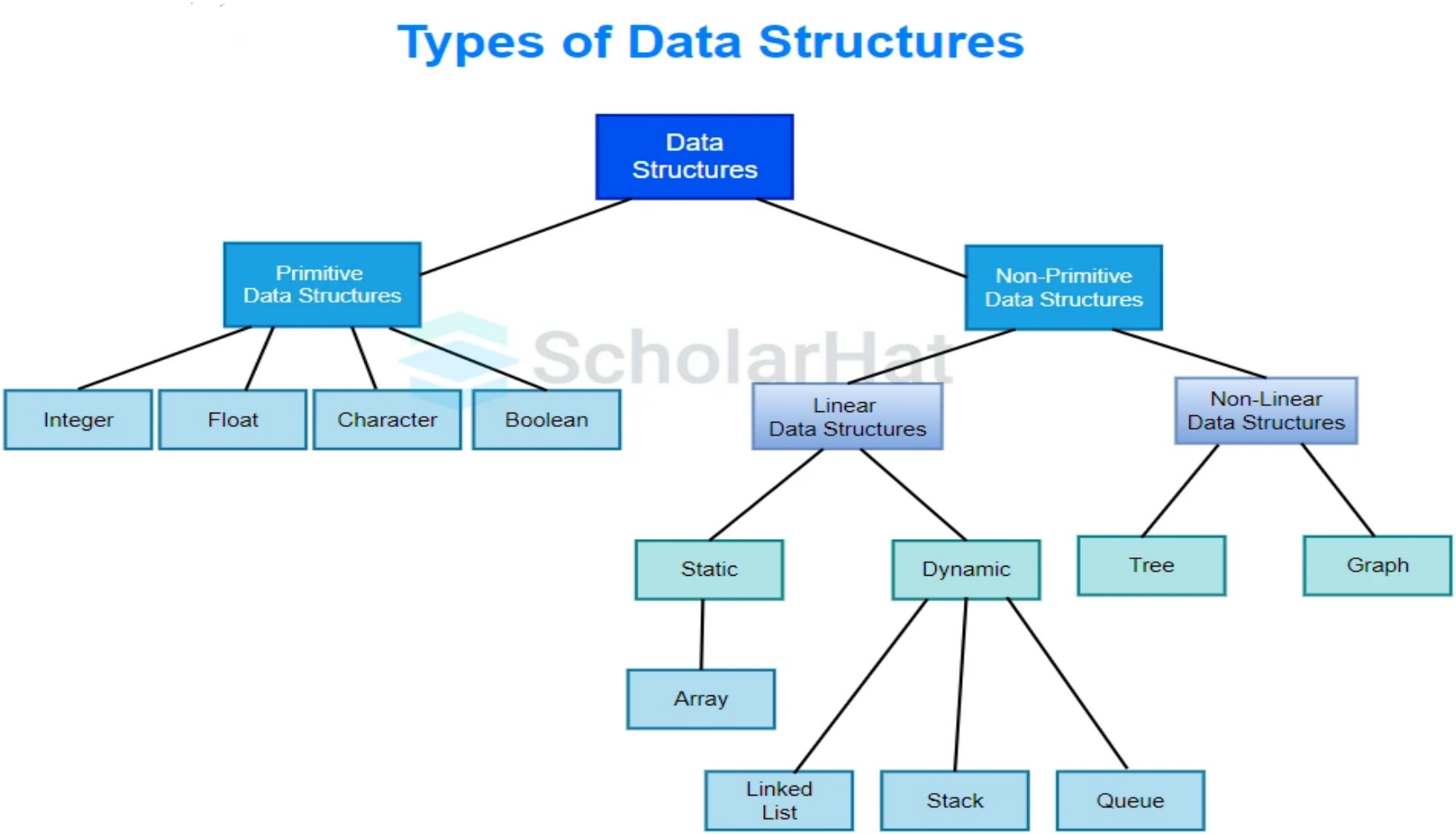
However, a data structure may be an *element* of another data structure. That means a data structure may contain another data structure.

DATA STRUCTURE [CONTD.]

- Example of Data Structures:
  - Array, Linked List, Stack, Queue, Tree, Graph, Hash Table etc.
- Types of elementary data item:
  - Character, Integer, Floating point numbers etc.
- Expressions of elementary data in C/C++
- Elementary data item - Expression in C/C++
  - Character - **char**
  - **Integer - int**
  - **Floating point number - float**

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

TYPE OF DATA STRUCTURE



A SIMPLE QUESTION ?

What are the major (basic) operations that can be performed on data structure?

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 2
OPERATION ON DATA STRUCTURE, TIME-SPACE COMPLEXITY, ALGORITHM, ARRAY DEFINITION

OPERATIONS ON DATA STRUCTURE

Basic:

- insertion (addition)
- deletion (access)
- searching (locate)

Additional (special):

- sorting
- merging etc

ALGORITHM

- Set of instructions that can be followed to perform a task. In other words sequence of steps that can be followed to solve a problem.
- To write an algorithm we do not strictly follow grammar of any particular programming language.
- However its language may be near to a programming language.

ALGORITHM

- Each and every algorithm can be divided into *three sections*.
  - First section is ***input*** section, where we show which data elements are to be given.
  - The second section is very important one, which is ***operational*** or ***processing*** **section**. Here we have to do all necessary operations, such as computation, taking decision, calling other procedure (algorithm) etc.
  - The third section is ***output***, where we display the result found from the second section.

PROGRAM

- Sequence of instructions of any programming language that can be followed to perform a particular task.
- Like an algorithm generally a program has three sections such as input, processing and output.

PROGRAM

- In a program usually we use a large amount of data. Most of the cases these data are not elementary items, where exists structural relationship between elementary data items.
  - That means the programs uses data structure(s).

- For a particular problem (usually for complex problem), at first we may write an algorithm then the algorithm may be converted into a program.

COMPLEXITY OF ALGORITHM

- Two types of complexities:
  - Time complexity
  - Space complexity.

TIME COMPLEXITY

- This complexity is related to execution time of the algorithm.

- It depends on the number of element (item) comparisons and number of element movement (movement of data from one place to another).

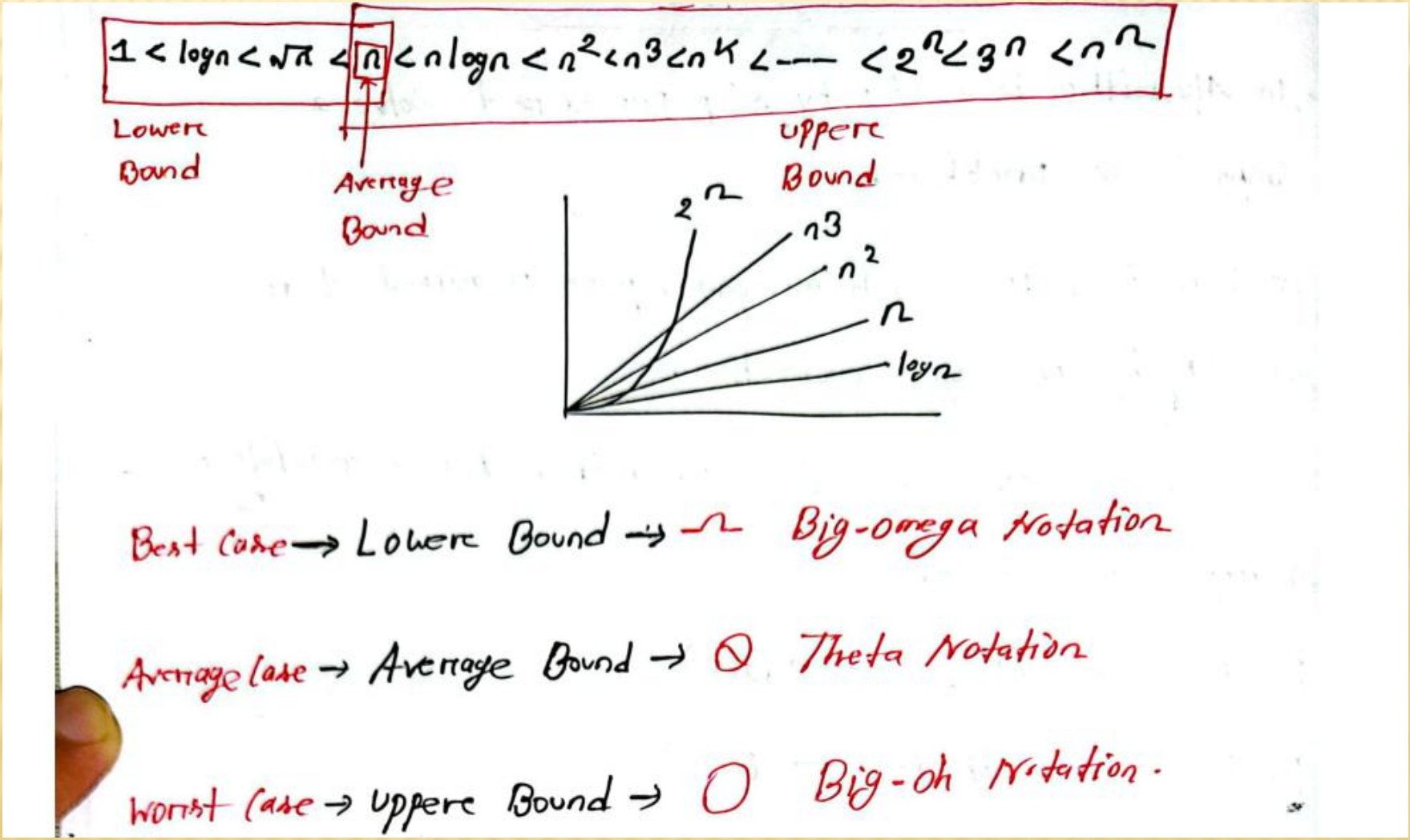
SPACE COMPLEXITY

- This complexity is related to space (memory) needs in the main memory for the data set used to implement the algorithm.
- That means if there *n* data items used in an algorithm, the space complexity of the algorithm will be proportional to *n*.

Symbolic Notation for Time Complexity

- The complexity of an algorithm (either time complexity or space complexity) is represented using asymptotic notations.
- One of the asymptotic notations is O (big-oh) notation.
- Big-oh (O) notation is also called upper bound of the complexity.

Symbolic Notation



Sample Questions of this chapter

1. Define data and data structure with example.
2. What are the elementary data items ? Give example.
3. What is data structure? What are the major operations that can be performed on data structure?
4. What is the difference between an algorithm and a program?

5. What do you mean by time and space complexities?
6. A data structure may be an element of another data structure. Explain this statement with example.
7. There are three sections in an algorithm, name these sections. Which one is most important ? Explain.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 3
ARRAYS:INITIALIZATION, ACCESS, TYPES OF ARRAY, ARRAY ADDRESSING: ROW MAJOR, COLUMN MAJOR

DEFINITION OF AN ARRAY

- An array is a finite set of same type of data items.
  In other words, it is a collection of homogeneous data items (elements).
- The elements of an array are stored in successive memory locations.
- Any element of an array is referred by array name and index number (subscript).
- There may have many dimensional arrays. But usually two types of array are widely used; such as
  - one dimensional (linear) array and
  - two dimensional array.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

TYPES OF ARRAY

One Dimensional Array

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

ONE DIMENSIONAL ARRAY

- An array that can be represented by only one dimension such as **row or column** and that holds finite number of same type of data items is called one dimensional (linear) array.

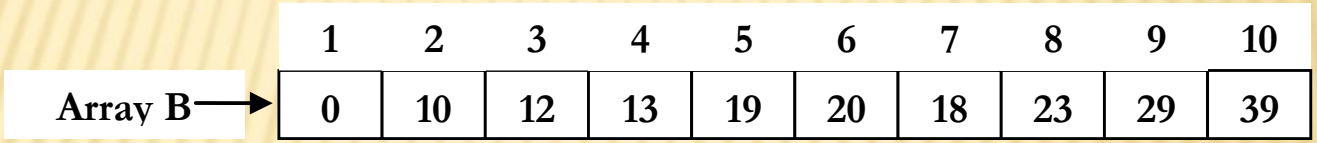


Figure 2.1: Graphical representation of one dimensional array.

Here 1, 2, 3, … … …, 10 are **index number**, and 0, 10, 12, … … …, 39 are **data items or elements** of the array and B is the array name.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

ONE DIMENSIONAL ARRAY [CONTD.]

- Symbolically an element of the array is expressed as B_i or B[*i*], which denotes i^{th} element of the array, B.
- Thus B[4], B[9] denotes respectively the 4^{th} element and the 9^{th} element of the array, B.
- The name of the array usually is a name constituted by one or more characters.
- Thus array name may be *A*, *S*, *Stock*, *Array1* etc.
- The element of an array may be number (integer or floating point number) or character

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

ONE DIMENSIONAL ARRAY [CONTD.]

- Expression of one dimensional array in C/C++:
  - For integer array:
    - int a[10];
  - For character array:
    - char b[30];
  - For floating point array:
    - float B[10];

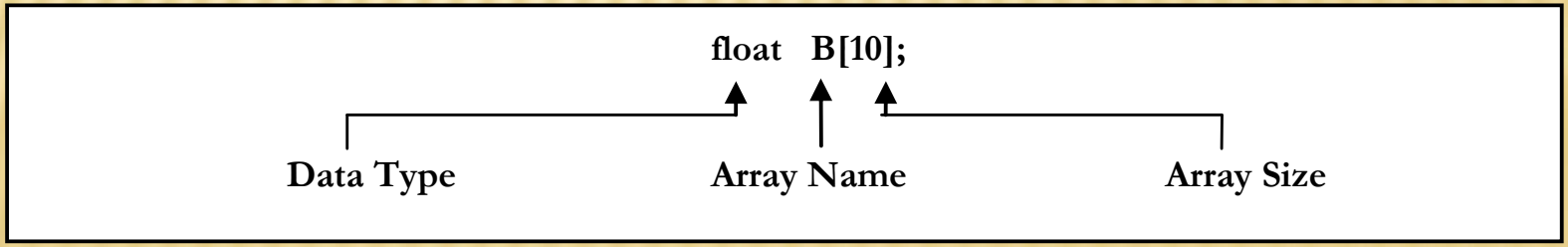


Figure 2.2: Declaration of Array in C/C++

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

ONE DIMENSIONAL ARRAY [CONTD.]

Store an element into an array

- *B*[4] = 19;

  it means 19 will be stored in the cell number 4 or 5 of the array of *B*.

  If there is any (previous) value that will be overwritten.

Read (retrieve) a value (element) from an array

- *x* = *B*[6];

it means the value of *x* will be 20, if the cell number 6 or 7 of the array, *B* contains 20.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

STORING DATA TO THE ARRAY

- Code in C/C++ for storing data in an array
  - int *x*[10];
  - for (*i* = 0; *i* < 10; ++*i*)
  - scanf ("%d", &*x*[*i*]);

Since the size of the array is 10, so we should enter data to the array 10 times.

That is why we use a loop of 10 times using a variable i.

C/C++ language starts indexing from 0 (zero), so the loop also starts from 0 (zero).

ONE DIMENSIONAL ARRAY [CONTD.]

- Code in C/C++ for accessing data from an array and the data will be displayed on the monitor's screen:

```
int x[10];
for (i =0; i < 10; ++i)
printf ("%d", x[i]);
```

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

POINTER AND DYNAMIC ARRAY

We can declare a pointer as follows:

int *a;

Here a is pointer variable that point integer type data.

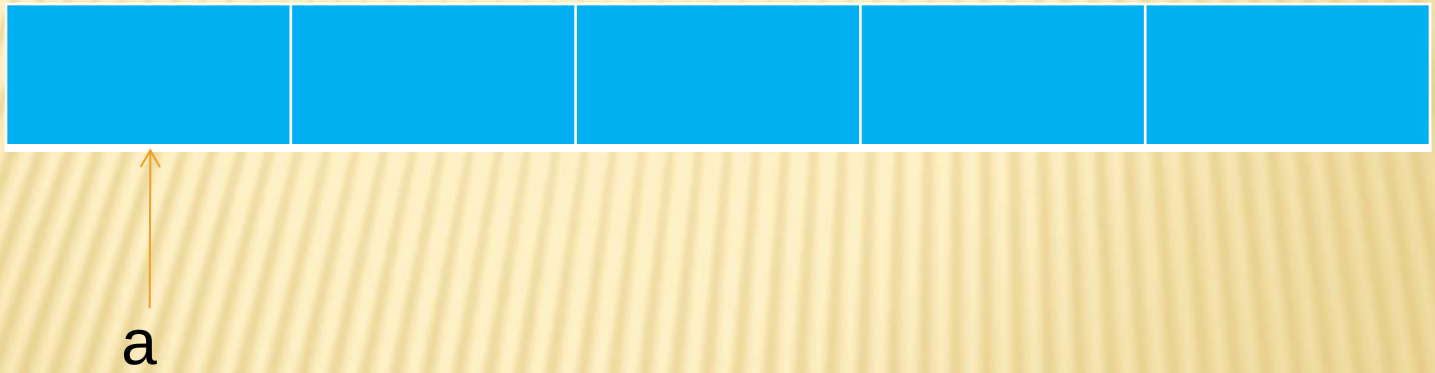
Similarly we can declare pointer for other types of data.

Using pointer and keyword, now we can declare dynamic array.

DYNAMIC ARRAY

Dynamic array declaration:

```
int *a;
a= new int [5];
```



DYNAMIC ARRAY

Variable can be used for size of the array.
Such as:

```
int s, *a;
cin>>s;
a= new int [s];
```

Size can be increased also:

```
s= s+1;
```

DYNAMIC ARRAY

Entering data to an array:

```
int s, i,*a;
cin>>s;
a= new int [s];
for( i =0;i<s;++i)
cin>>a[i];
```

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

TO FIND OUT LARGEST ELEMENT

- **Problem 2.1:**
  - Given a list of elements, write an algorithm to store the list of elements (numbers) in an array and find out the largest element of the list.
- **Algorithm 2.1: Algorithm to search the largest element of a list**

1. Input: $x[1 \ldots n]$;
2. for ($i = 1$; $i \leq n$; ++i)
   store data to $x[i]$;
3. *large* = $x[1]$;
4. for ($i = 2$; $i \leq n$; ++i)
   if ($x[i] > large$), *large* = $x[i]$;
5. Output: the largest number is, ***large***

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

TO FIND OUT LARGEST ELEMENT

Code for algorithm 2.1 in C/C++

```
int  x[10],i, large;
cout<< "Enter data to the array:";
 for (i = 0; i < 10; ++i )
   {
    cin>>x[i];
    }
 large = x[0];
 for (i = 1; i < 10; ++i )
        if (x[i] > large), large = x[i];
cout<<"the largest is:"<< large;
```

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

FIND OUT THE SUMMATIONS OF EVEN AND ODD NUMBERS

- **Algorithm 2.3: Algorithm to find the summation of even and odd numbers**

1. Input: $A[1...n]$, $sum\_odd = 0$, $sum\_even = 0$;
   //An array and variables to store the summation
2. for ($i = 1$; $i \leq n$; $++i$)
   {
   if ($A[i]$ % 2 = = 0),
   $sum\_even = sum\_even + A[i]$;
   else
   $sum\_odd = sum\_odd + A[i]$;
   }
3. Output: Summation of odd numbers, sum_odd and summation of even numbers, sum_even)

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

SUMMATION OF NUMBERS IN ODD AND EVEN INDICES SEPARATELY

- **Algorithm 2.4: Algorithm to find the summation of even and odd indexed numbers**

1. Input: $A[1...n]$, $sum\_odd = 0$, $sum\_even = 0$;
   //An array and variables (to store the summation)
2. for ($i = 1$; $i \leq n$; ++i)
   {
   if (i % 2 = = 0), $sum\_even = sum\_even + A[i]$;
   else $sum\_odd = sum\_odd + A[i]$;
   }
3. Output: Summation of numbers in odd indices, *sum_odd* and summation of numbers in even indices, *sum_even*.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

TYPES OF ARRAY

Two Dimensional Array

DEFINITION OF TWO DIMENSIONAL ARRAY

- Two dimensional array is an array that has two dimensions, such as row and column.
- Total number of elements in a two dimensional array can be calculated by multiplication of the number of rows and the number of columns.
- If there are *m* rows and *n* columns, then the total number of elements is $m \times n$, and $m \times n$ is called the size of the array.
- Of course, the data elements of the array will be same type.
- In mathematics, the two dimensional array is called a matrix and in business it is called table.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

SYMBOLIC REPRESENTATION OF TWO DIMENSIONAL ARRAY

A two dimensional array can be represented using symbols as follows:

$$A_{ij} \text{ or } A[i,j] \quad for\, 1 \le i \le m \, and \, 1 \le j \le n$$

Where m is the number of rows and
n is the number of columns.

A[1 . . . . m, 1 . . . . n]

m rows n columns

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

TWO DIMENSIONAL ARRAY

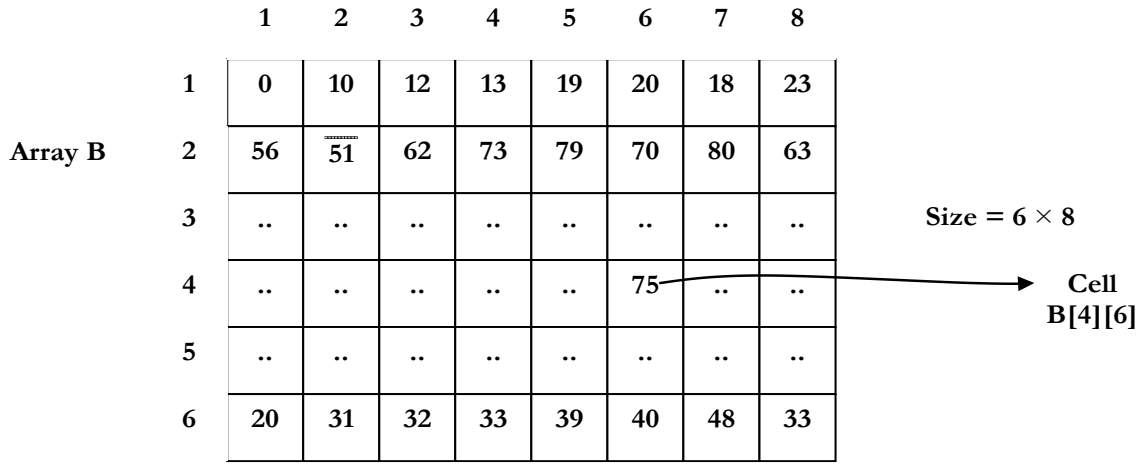


Figure 2.4: Graphical representation of two dimensional array

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

EXPRESSION OF TWO DIMENSIONAL ARRAY

Two dimensional array can be expressed in C/C++ as follows:

```
int A[5][4];
```

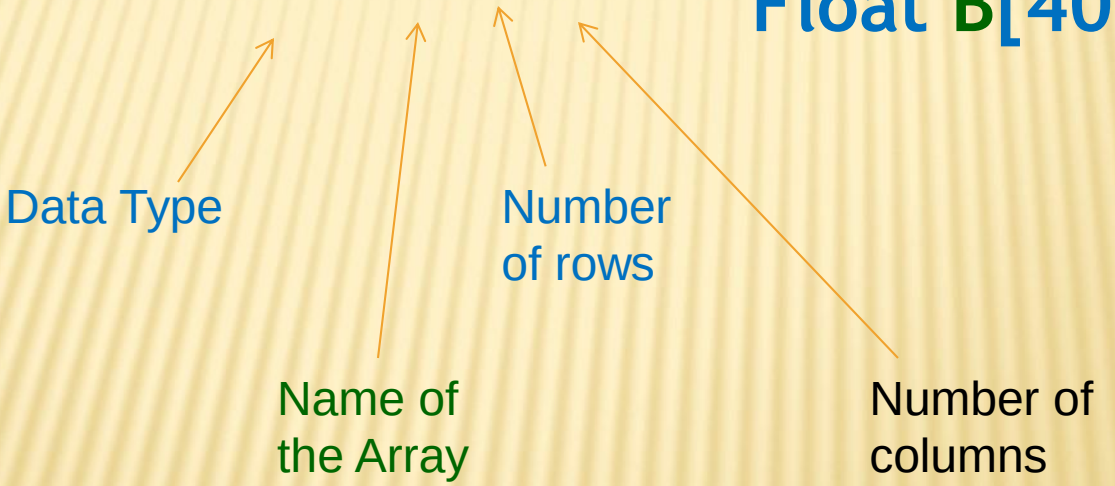
Here int is the type of the array,
A is the name of the array,
5 is the number of rows and
4 is the number of columns.

The type of the array is integer means the data of the array are integers.

EXPRESSION [CONT…]

Another Example of expression:

Float B[40][4];



Partho Sarathi SArker
Asst Professor: Dept. Of CSE

TO STORE AND RETRIEVE VALUES IN AND FROM ARRAY

- Data can be stored in a two dimensional array using loop or directly as shown below:
- i) storing data taken from keyboard

```
 int B[7][3];
for (int i = 0; i < 7; ++ i)
  {
        for (int j = 0; j < 3; ++ j)
  scanf ("%d", &B[i][j]);
       }
```

II) DIRECT INSERTION OF DATA IN TWO DIMENSIONAL ARRAY

```
int B[7][3] =   {
  { 1,   2,   3},
  { 9, 10, 11},
    … …. ….,
     … …. ….,
     … …. ….,
     … …. ….,
   {22, 25, 40}
                    };
```

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

TWO DIMENSIONAL ARRAY REPRESENTATION IN MEMORY

- The elements of a two dimensional array are stored in computer's memory *row by row* or *column by column*.
- If the array is stored as *row by row*, it is called *row-major order*.
- If the array is stored as *column by column*, it is called *column-major order*.
- Suppose there is a two-dimensional array of size 5 × 6. That means, there are 5 rows and 6 columns in the array.

ARRAY REPRESENTATION IN MEMORY

- In *row-major order*, elements of a two dimensional array are ordered as –
- $A_{11}, A_{12}, A_{13}, A_{14}, A_{15}, A_{16}, A_{21}, A_{22}, A_{23}, A_{24}, A_{25}, A_{26}, A_{31}, \ldots\ldots\ldots\ldots, A_{46}, A_{51}, A_{52}, \ldots\ldots\ldots, A_{56}.$

- and in *column-major order*, elements are ordered as –
- $A_{11}, A_{21}, A_{31}, A_{41}, A_{51}, A_{12}, A_{22}, A_{32}, A_{42}, A_{52}, A_{13}, \ldots\ldots\ldots\ldots, A_{55}, A_{16}, A_{26}, \ldots\ldots\ldots, A_{56}.$

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

LOCATION OF AN ELEMENT(ARRAY ADDRESSING)

- Location of an element of a two-dimensional array
  - *Row-major Order*:
    - If Loc (A[*i*, *j*]) denotes the location in the memory of the element A[*i*][*j*] or A*ij*, then in row-major order –
      - Loc (A[*i*, *j*]) = Base (A) + (*n* (*i* - 1) + (*j* - 1)) * w;
    - Here Base (A) is starting or base address of the array A, *n* is the number of columns and w is the width of each cell, i.e, number bytes per cell.

LOCATION OF AN ELEMENT

- ***Column-major Order*:**
  - In column-major order,
    - Loc (A[*i*, *j*]) = Base (A) + (*m* (*j* - 1) + (*i* - 1)) * w;
  - Here Base (A) is starting or base address of the array A, *m* is the number of rows and w is the cell width.

LOCATION OF AN ELEMENT

- **Example:**
  - Base address, Base (A) = 100, Size of the array = 5 × 6. If the type of array is integer then find Loc (A[4, 3]).
- **Solution:**

(2 bytes for each integer cell in C/C++)

If the array is stored in row-major order:

Loc (A[4, 3]) = Base (A) + (n (i - 1) + (j - 1))* 2
= 100 + (6 × 3 + 2)* 2
= 100 + 40
= 140

If the array is stored in memory in column-major order:

Loc (A[4, 3]) = Base (A) + m (j - 1) + (i - 1)* 2
= 100 + (5 × 2 + 3)* 2
= 100 + 26
=126

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

TWO DIMENSIONAL ARRAY

Operations on two dimensional array

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

THE SUMMATION OF THE DIAGONAL ELEMENTS

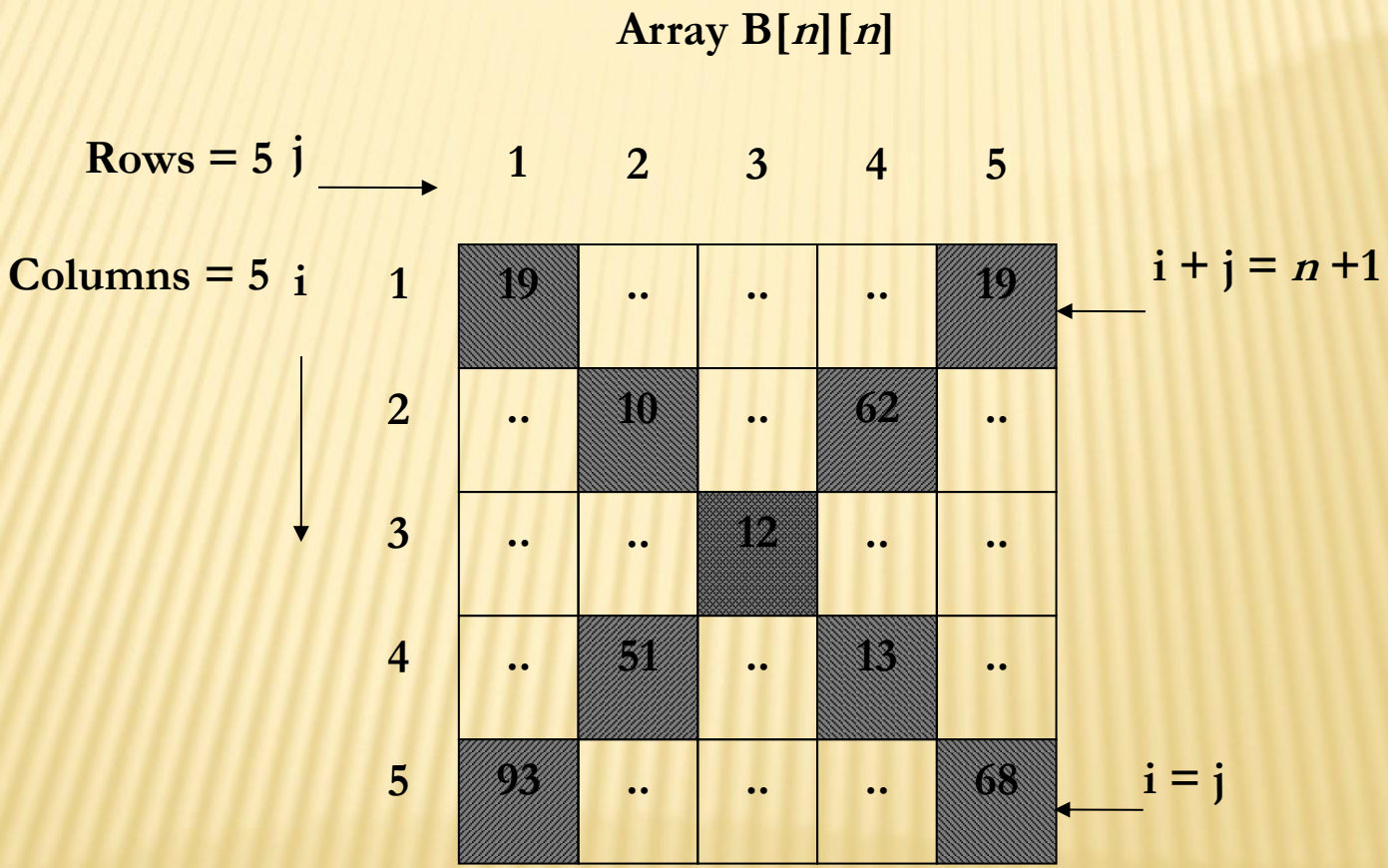


Figure 2.9: Diagonal elements of a two-dimensional array.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

ALGORITHM TO FIND OUT SUMMATION OF DIAGONAL ELEMENTS

- 1. Input: B[*1 . . . n*, 1...*n*], *sum* = 0;
  //a two dimensional array

- 2. Find each diagonal element and add them
  for (*i* = 1; *i* ≤ *n*; ++*i*)
  {for (*j* = i; *j* ≤ n; ++*j*)
  if (*i* = *j* || *i* + *j* = *n* + 1), *sum* = *sum* + B[i, j]
  }

- 3. Output: Print *sum* as the result of summation of diagonal elements.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

PROGRAMMING CONSIDERATION

For top-right to left-bottom diagonal in C/C++:

$$i + j = n - 1$$

1 sholud be deducted from row index and another 1 should be deducted from column index.

So, $n+1-2 = n-1$.

Therefore:

$$i + j = n - 1;$$

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

SAMPLE QUESTIONS OF THIS CHAPTER

1. Define linear array with example.
2. Declare a linear array of size 5, store data and show (print) them using code.
3. What is dynamic array? Give example.
4. Algorithms related to linear array.
5. What is two-dimensional array? Give example.
6. Write code to store (enter) data using a two-dimensional array of size 3x4 and print them.

SAMPLE QUESTIONS …..

7. How two-dimensional array can be represented into computer's memory? Explain with example.
8. Algorithms related to two-dimensional array.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

Thank You.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 4
SORTING AND SEARCHING ALGORITHMS: LINEAR SEARCH, BINARY SEARCH

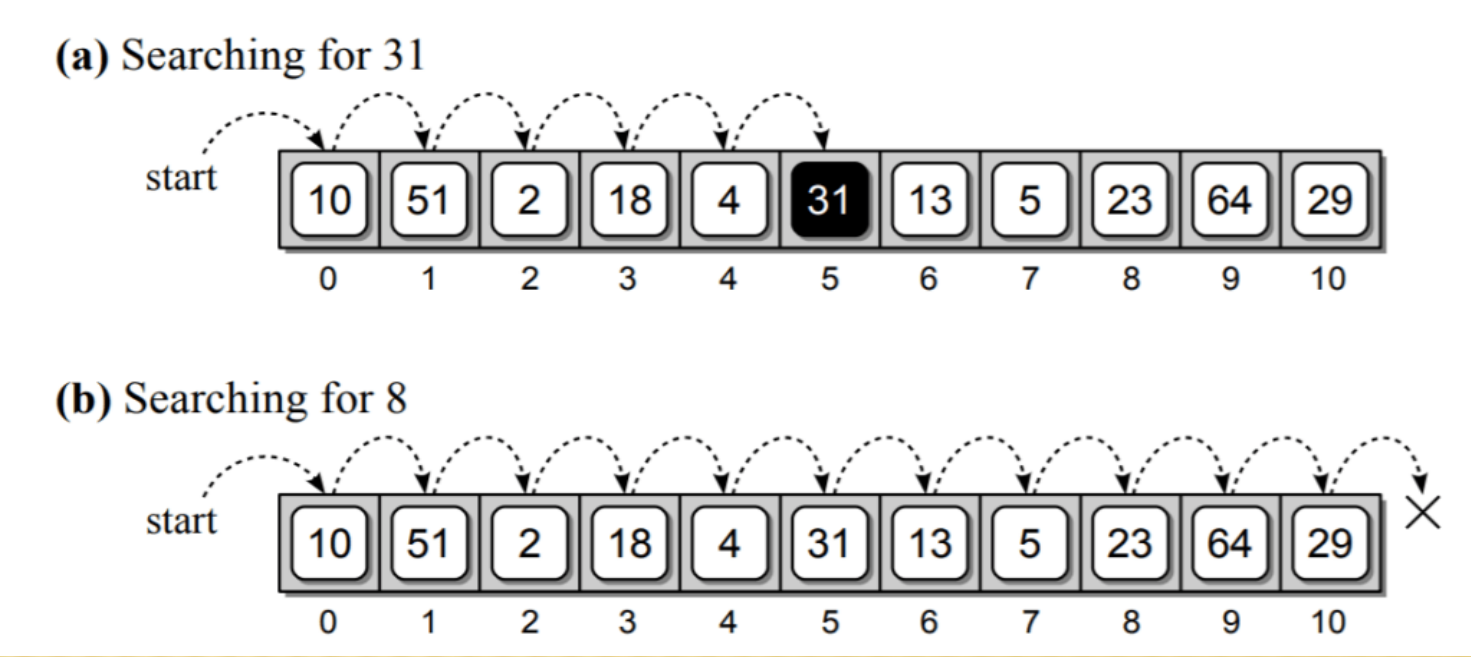
LINEAR SEARCH

- Linear search is also called as **sequential search algorithm.** It is the simplest searching algorithm.

- In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

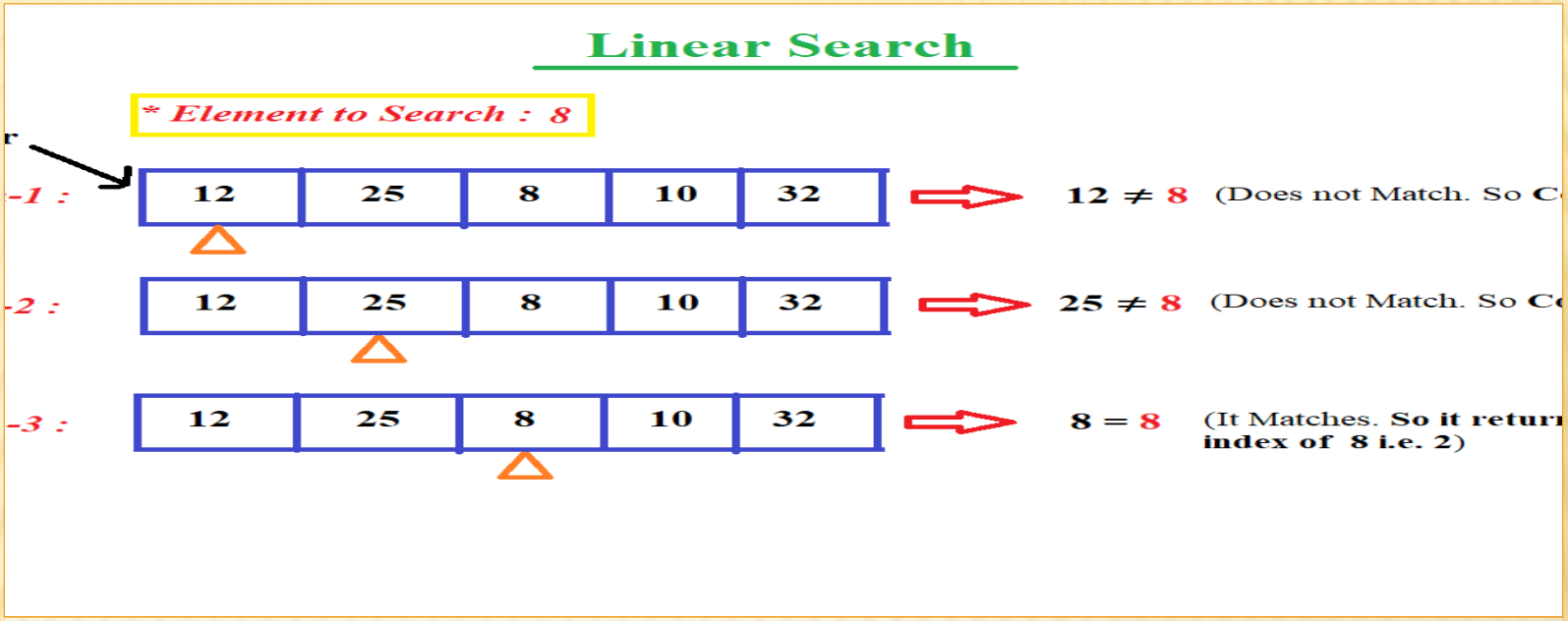
LINEAR SEARCH

- It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is **O(n).**



LINEAR SEARCH PROCEDURE

Partho Sarathi SArker
Asst Professor: Dept. Of CSE



The steps used in the implementation of Linear Search are listed as follows -

- First, we have to traverse the array elements using a **for** loop.
- In each iteration of **for loop,** compare the search element with the current array element, and -
  - If the element matches, then return the index of the corresponding array element.
  - If the element does not match, then move to the next element.
- If there is no match or the search element is not present in

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

LINEAR SEARCH ALGORITHM

Linear_Search(a, n, val) // 'a' is the given array, 'n' is the size of given array, 'val' is the value to search

Step 1: set pos = -1

Step 2: set i = 1

Step 3: repeat step 4 while i <= n

Step 4: if a[i] == val

set pos = i

go to step 6

[end of if]

set i = i + 1

[end of loop]

Step 5: if pos = -1

print "value is not present in the array "

[end of if]

Step 6: exit

BINARY SEARCH

- **Binary Search Algorithm is a searching algorithm used in a sorted array by repeatedly dividing the search interval in half.**
- **The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).**

BINARY SEARCH PROCEDURE

- Below is the step-by-step algorithm for Binary Search:

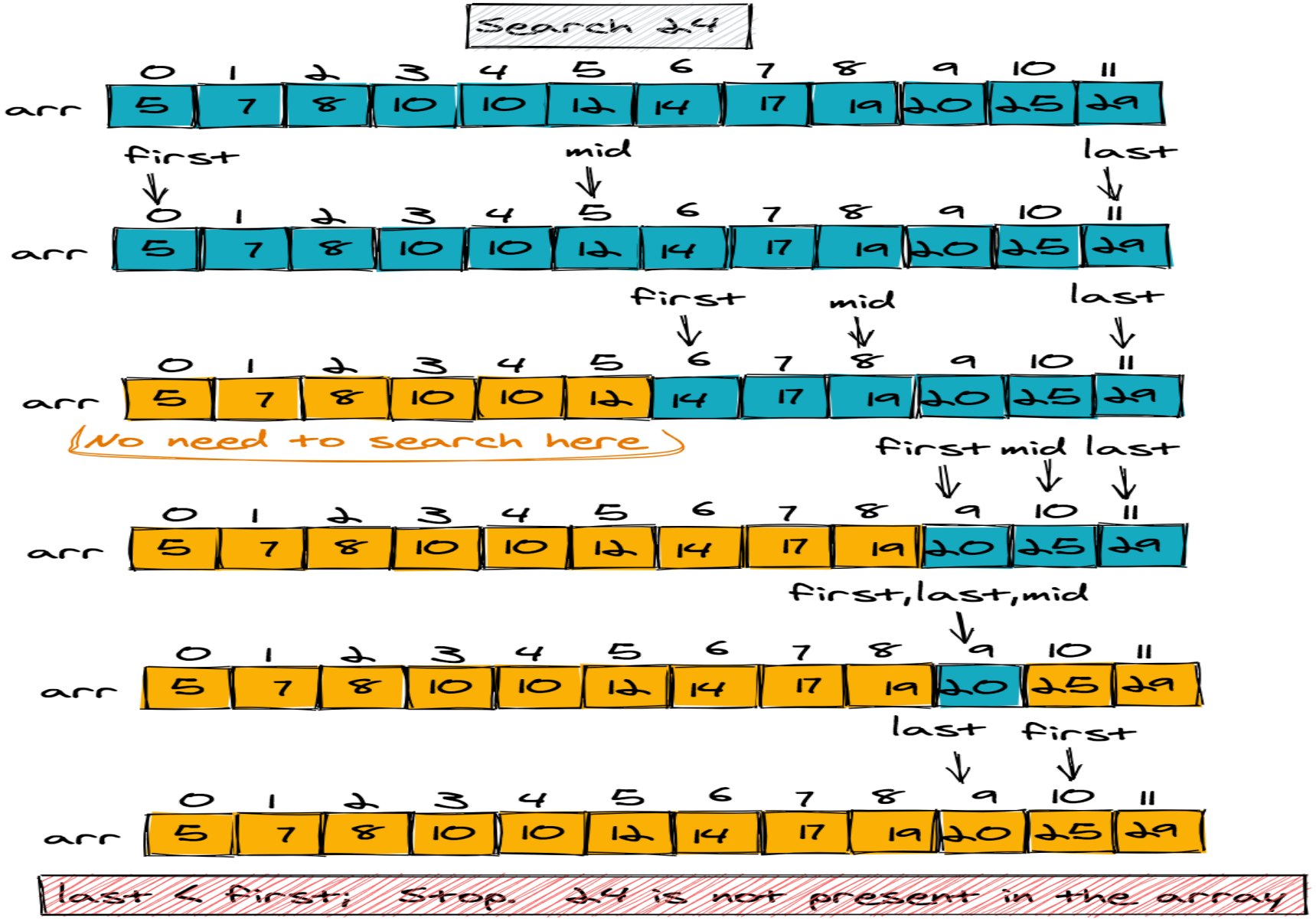
- Divide the search space into two halves by finding the middle index “mid”.
- Compare the middle element of the search space with the **key**.

BINARY SEARCH PROCEDURE

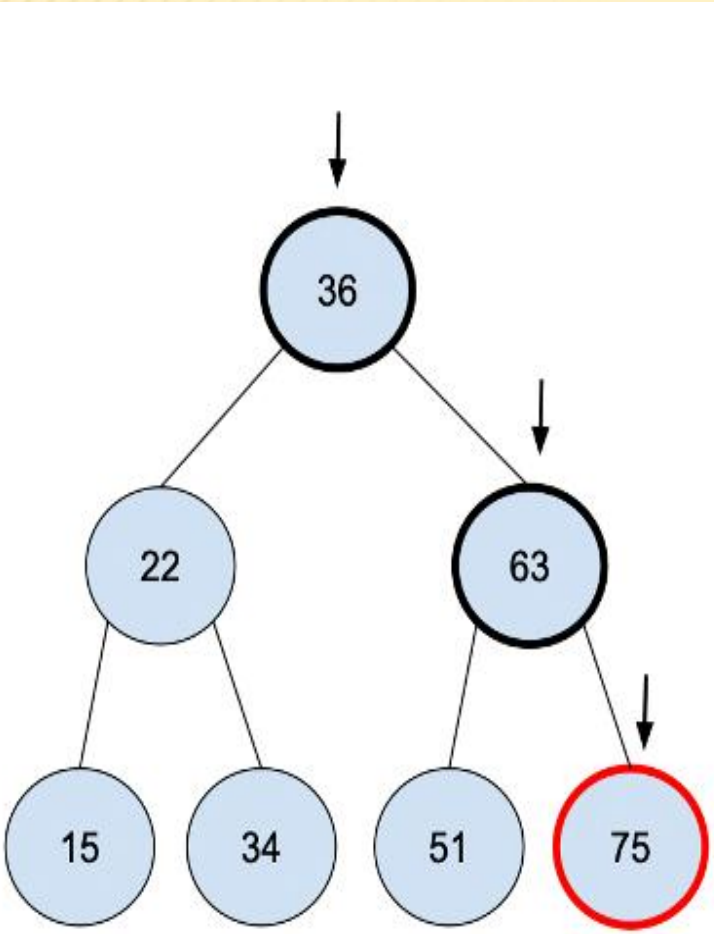
- If the **key** is found at middle element, the process is terminated.
- If the **key** is not found at middle element, choose which half will be used as the next search space.
  - If the **key** is smaller than the middle element, then the **left** side is used for next search.
  - If the **key** is larger than the middle element, then the **right** side is used for next search.
- This process is continued until the **key** is found or the total search space is exhausted.

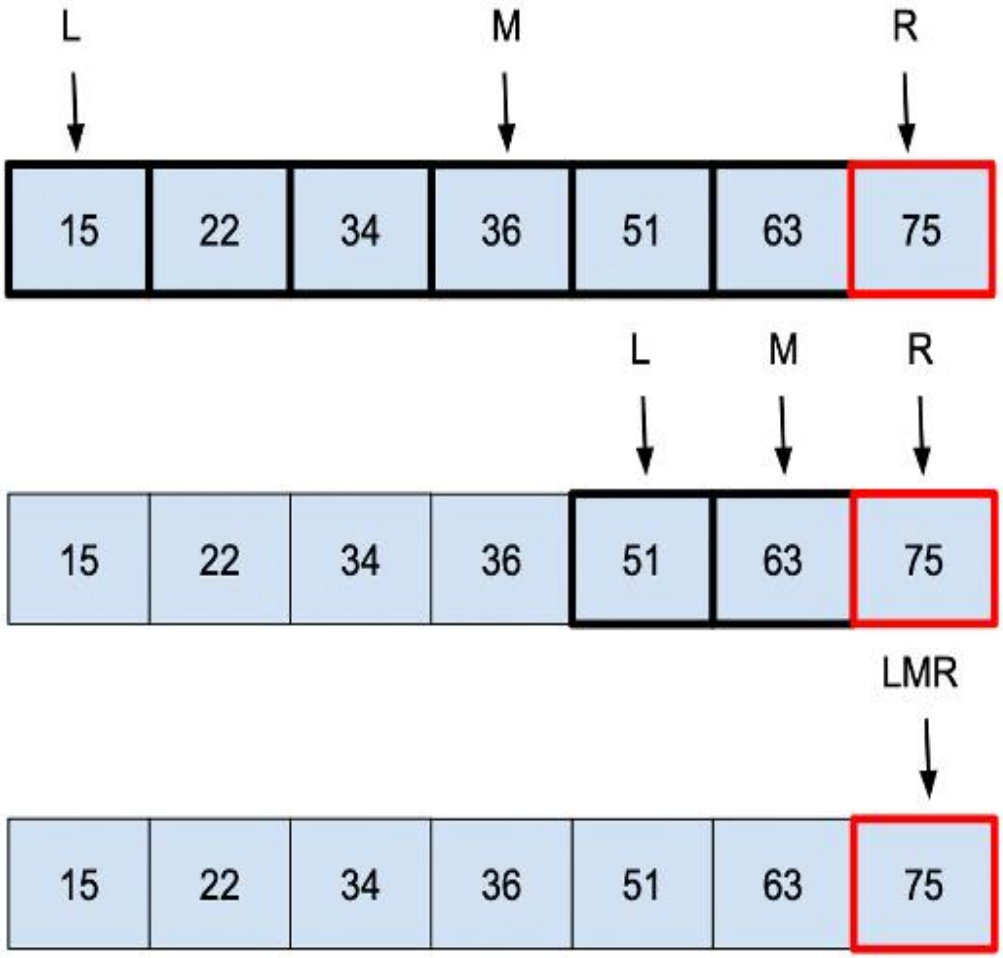
BINARY SEARCH GRAPHICAL REPRESENTATION

Partho Sarathi SArker
Asst Professor: Dept. Of CSE



BINARY SEARCH GRAPHICAL REPRESENTATION





THANK YOU

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 5
SELECTION SORT, BUBBLE SORT

SELECTION SORT

- **Selection Sort** is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the **smallest (or largest)** element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

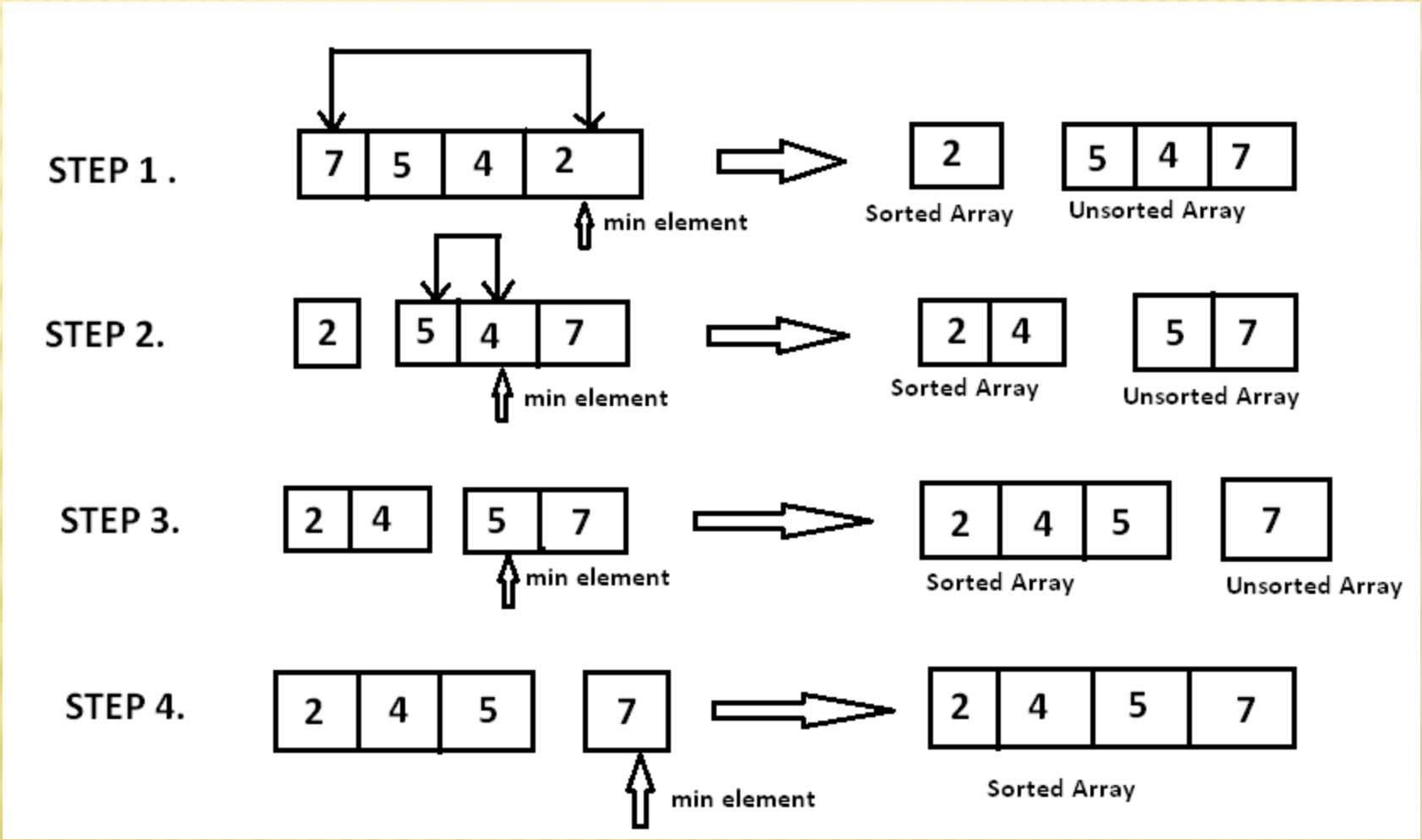
SELECTION SORT PROCEDURE

1. First we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then we find the smallest among remaining elements (or second smallest) and swap it with the second element.
3. We keep doing this until we get all elements moved to correct position.

SELECTION SORT ALGORITHM

- Set MIN to location 0.
- Search the minimum element in the list.
  - Swap with value at location MIN.
  - Increment MIN to point to next element.
- Repeat until the list is sorted.

SELECTION SORT EXAMPLE



COMPLEXITY OF SELECTION SORT

- **Time Complexity: $O(n^2)$** ,as there are two nested loops:

- One loop to select an element of Array one by one = O(n)
- Another loop to compare that element with every other Array element = O(n)
- Therefore overall complexity = O(n) * O(n) = O(n*n) = $O(n^2)$

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.

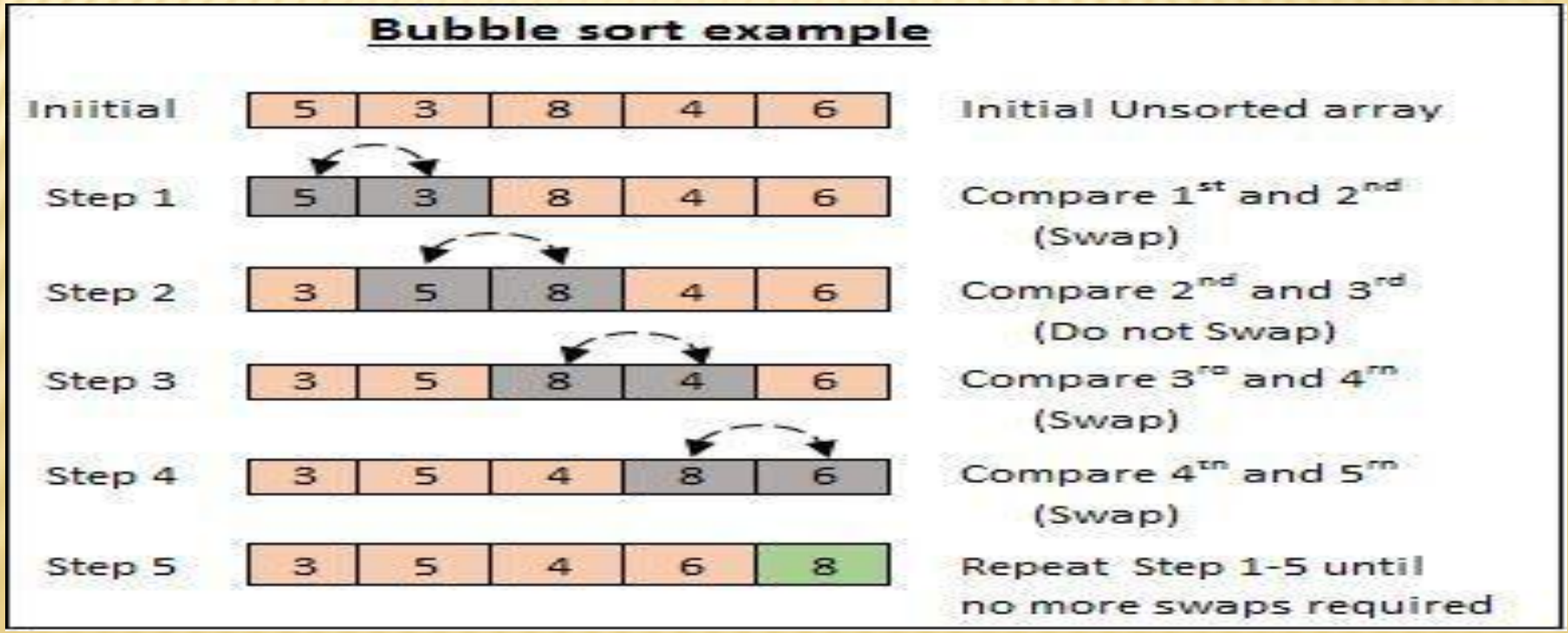
BUBBLE SORT

- We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after second pass, the second largest element goes to second last position and so on.
- In every pass, we process only those elements that have already not moved to correct position. After k passes, the largest k elements must have been moved to the last k positions.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

BUBBLE SORT

- In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.



BUBBLE SORT ALGORITHM

- Algorithm:
- Sequential-Bubble-Sort (A)
- fori ← 1 to length [A] do
  - for j ← length [A] down-to i +1 do
    - if A[A] < A[j-1] then
      - Exchange A[j] ↔ A[j-1]

SAMPLE QUESTION

- Analyze advantages and disadvantages of selection sort and bubble sort
- Analyze time complexity of bubble sort
- Compare selection sort and bubble sort

THANK YOU

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 6
INSERTION SORT, QUICK SORT

INSERTION SORT

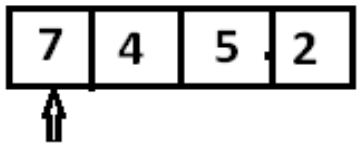
- **Insertion sort** is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

INSERTION SORT PROCEDURE

- We start with second element of the array as first element in the array is assumed to be sorted.
- Compare second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the first two elements and put at its correct position
- Repeat until the entire array is sorted.

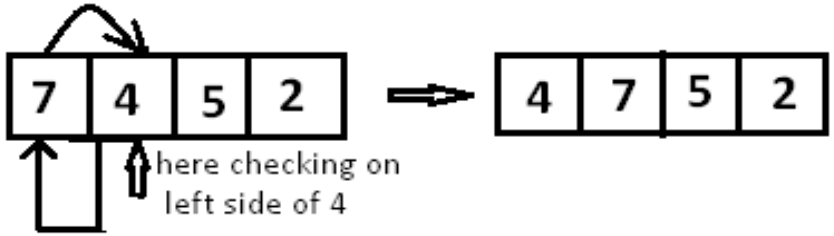
INSERTION SORT PROCEDURE

STEP 1.



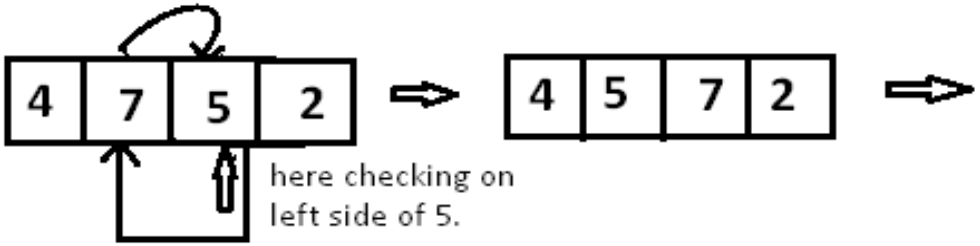
No element on left side of 7,so no change in its position .

STEP 2.



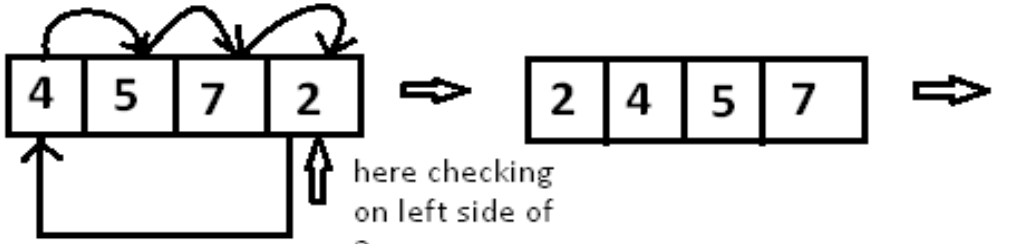
As 7>4 ,therfore 7 will be moved forward and 4 will be moved to 7's position.

STEP 3.



As 7>5,7 will be moved forward,but 4 <5,so no change in position of 4. And 5 will be moved to position of 7.

STEP 4.



As all the element 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

INSERTION SORT ALGORITHM

- **Step 1** – If it is the first element, it is already sorted. return 1;
- **Step 2** – Pick next element
- **Step 3** – Compare with all elements in the sorted sub-list
- **Step 4** – Shift all the elements in the sorted sub-list that is greater than the value to be sorted
- **Step 5** – Insert the value
- **Step 6** – Repeat until list is sorted

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

INSERTION SORT ALGORITHM

- Algorithm: Insertion-Sort(A)
- for j = 2 to A.length
  - key = A[j]
  - i = j – 1
  - while i > 0 and A[i] > key
    - A[i + 1] = A[i]
    - i = i -1
  - A[i + 1] = key

INSERTION SORT ALGORITHM

- Run time of this algorithm is very much dependent on the given input.
- If the given numbers are sorted, this algorithm runs in **O(n)** time. If the given numbers are in reverse order, the algorithm runs in **$O(n^2)$** time.

QUICK SORT

- **QuickSort** is a sorting algorithm based on the Divide and Conquer 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.

HOW DOES QUICKSORT ALGORITHM WORK?

- QuickSort works on the principle of **divide and conquer**, breaking down the problem into smaller sub-problems.
- There are mainly three steps in the algorithm:

1. **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).

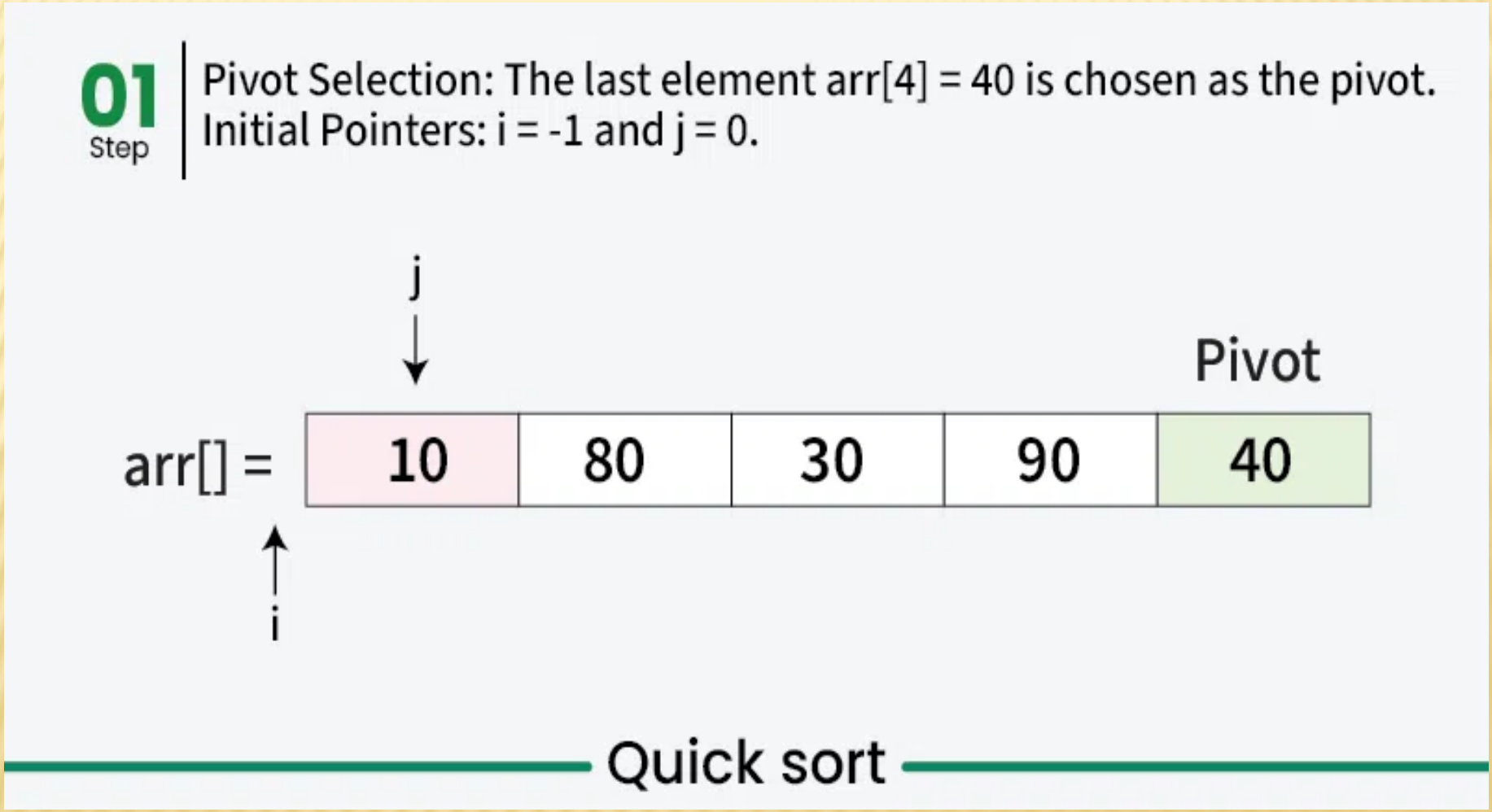
2. **Partition the Array:** Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.

HOW DOES QUICKSORT ALGORITHM WORK?

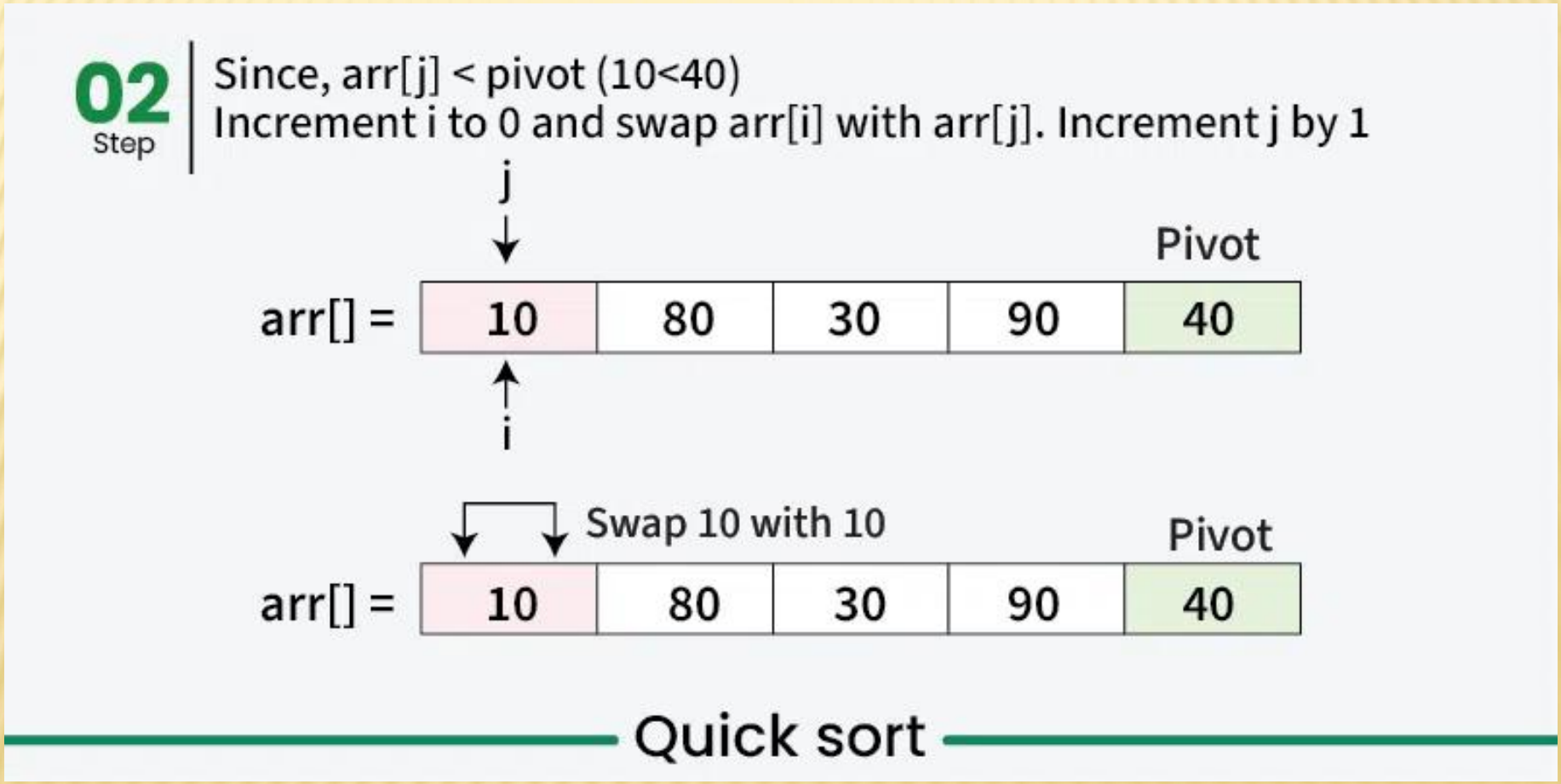
3. **Recursively Call:** Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).

4. **Base Case:** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

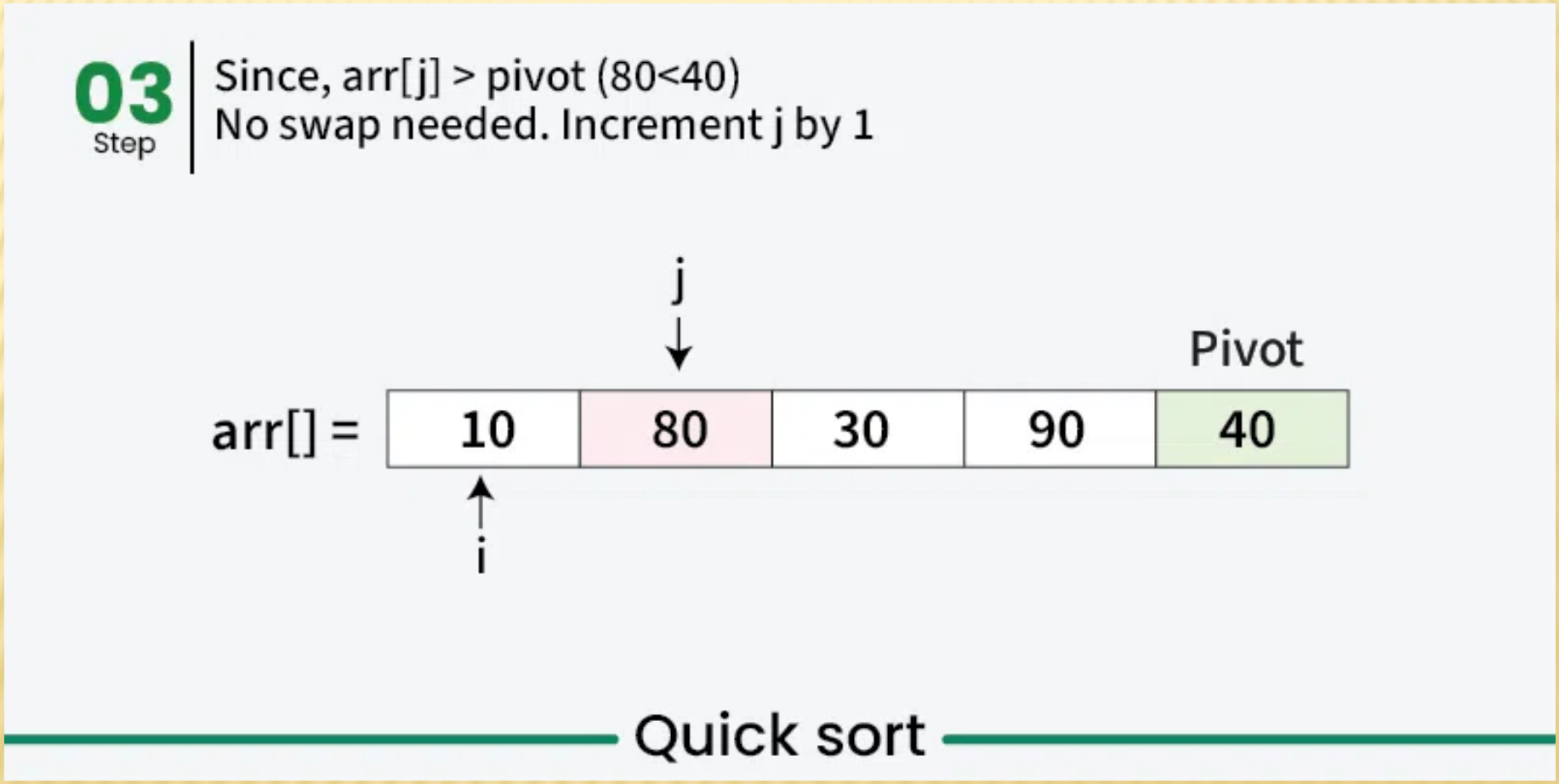
GRAPHICAL ILLUSTRATION OF QUICK SORT



GRAPHICAL ILLUSTRATION OF QUICK SORT



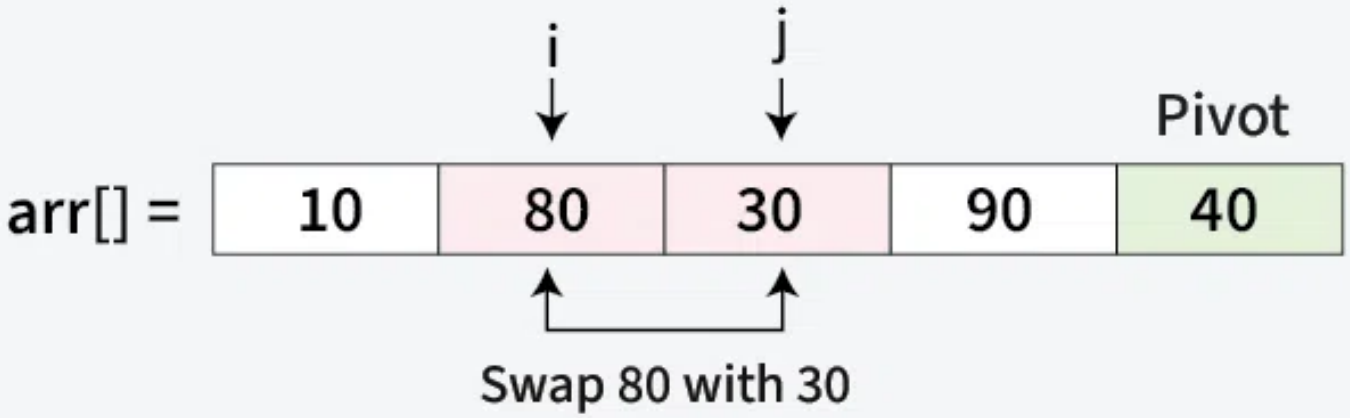
GRAPHICAL ILLUSTRATION OF QUICK SORT



GRAPHICAL ILLUSTRATION OF QUICK SORT

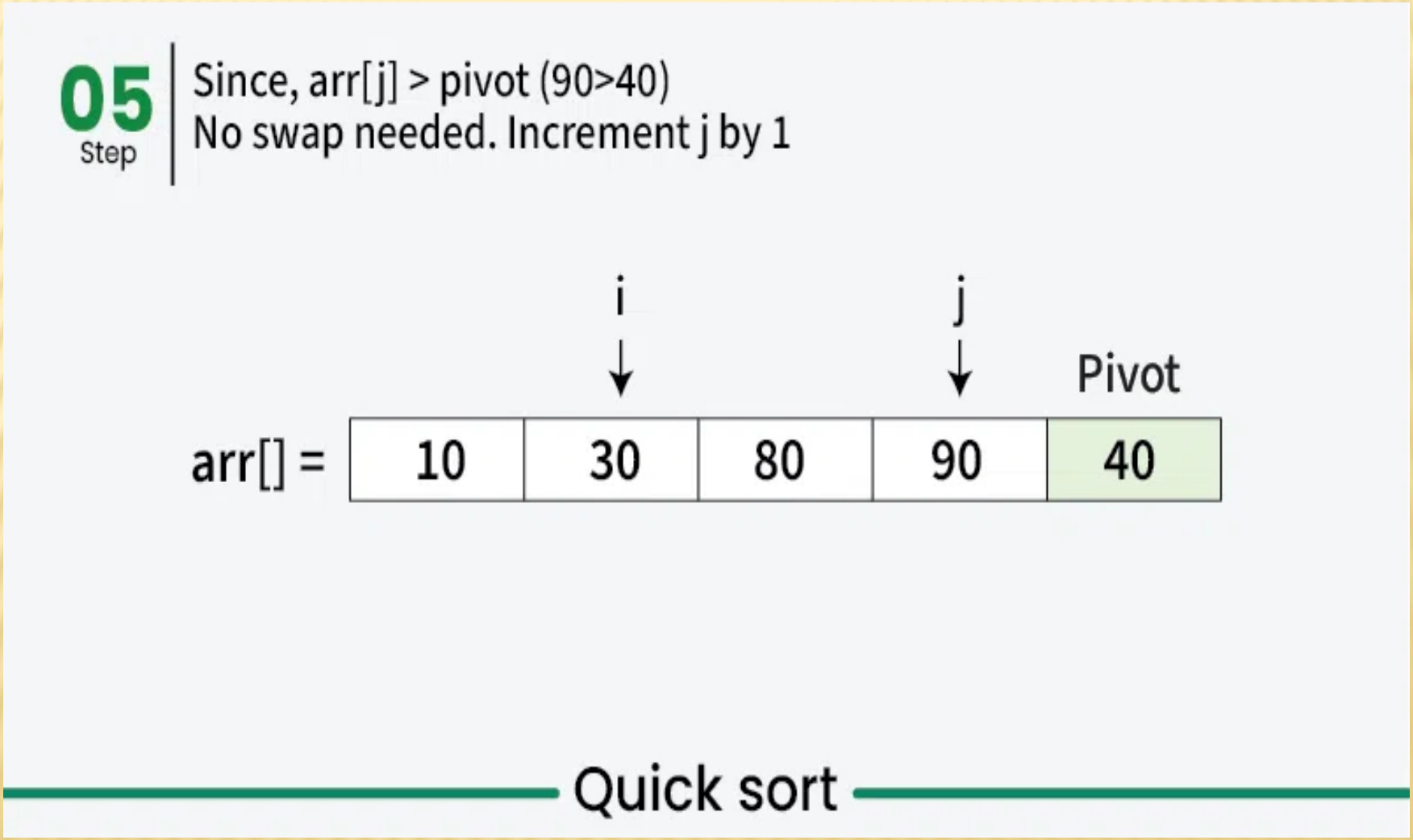
04 Step

Since, arr[j] < pivot (30<40)
Increment i by 1 and swap arr[i] with arr[j]. Increment j by 1



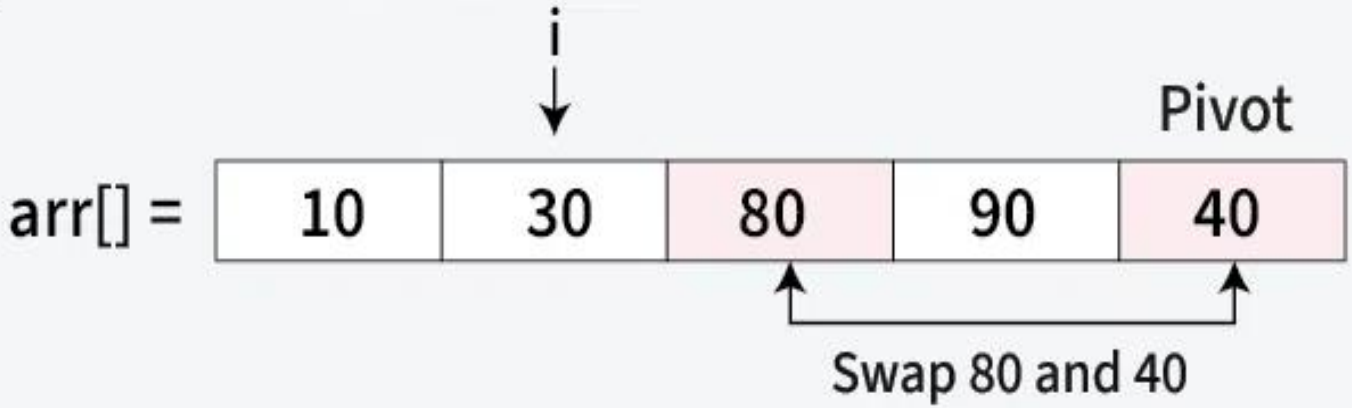
Quick sort

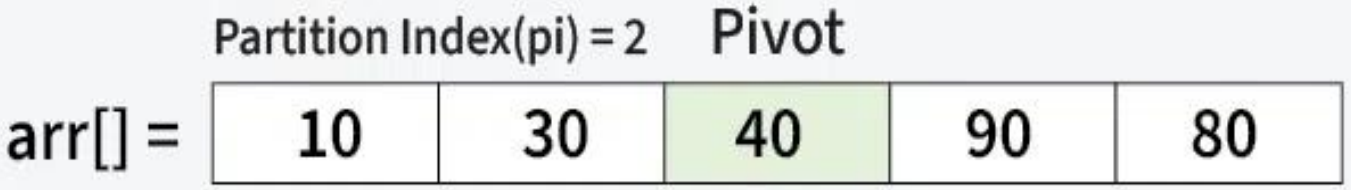
GRAPHICAL ILLUSTRATION OF QUICK SORT



GRAPHICAL ILLUSTRATION OF QUICK SORT

06 Step | Since traversal of j has ended. Now move pivot to its correct position, Swap arr[i + 1] = arr[2] with arr[4] = 40.





Quick sort

SAMPLE QUESTION

- Advantage and Disadvantage of Insertion Sort and Quick Sort
- Analyze Time Complexity of Quick Sort
- Compare all above Four Sort Technique.

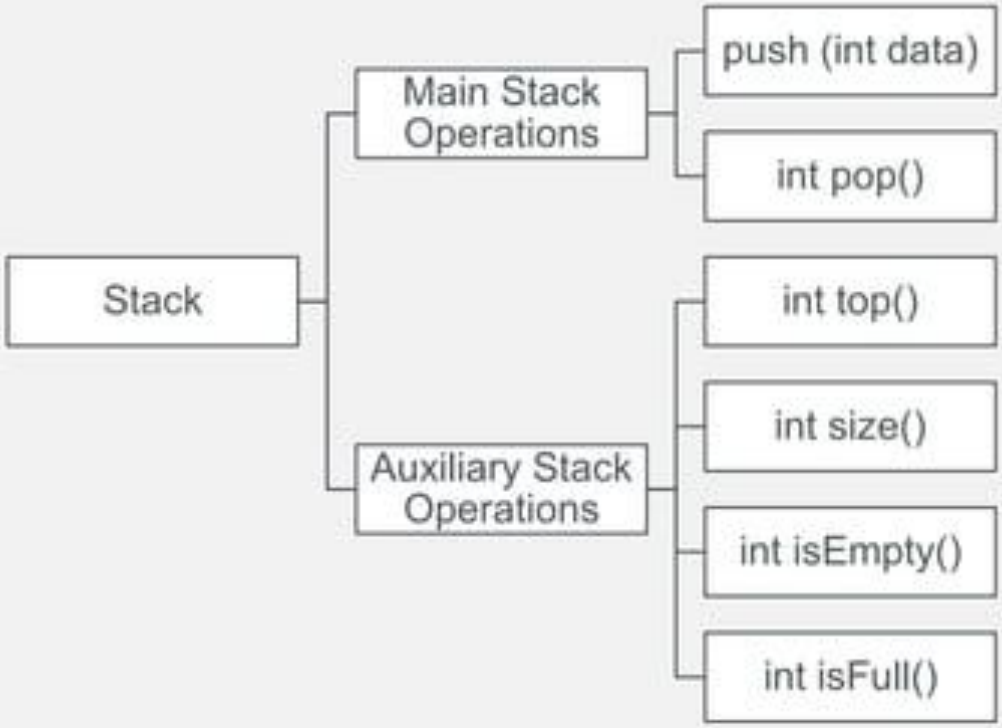
Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 6
STACKS AND QUEUES
PUSH, POP OPERATION OF A STACK.

WHAT IS STACK

- A stack is an ordered list in which insertion and deletion are done at one end, called *top*.
- The last element inserted is the first one to be deleted.
- Hence it is called Last In First Out(LIFO) or First In Last Out(FILO).

STACK ADT


Stack
Main Stack Operations
Auxiliary Stack Operations
push (int data)
int pop()
int top()
int size()
int isEmpty()
int isFull()

STACK OPERATIONS

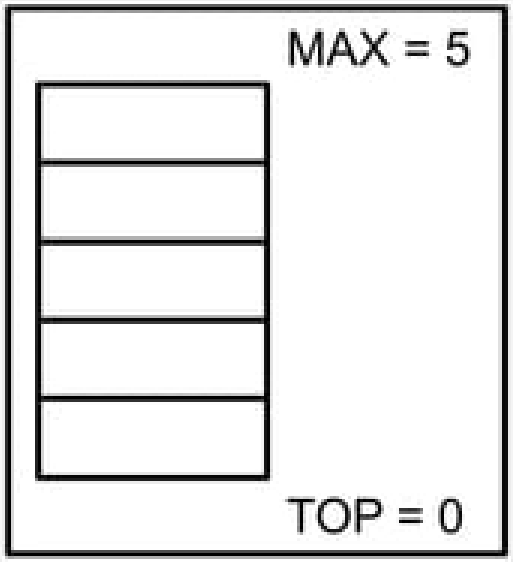
Operation	Description
push(int data)	inserts *data* into stack
int pop()	removes and returns last inserted element
int top()	returns last inserted element
int size()	returns number of elements in stack
int isEmpty()	indicate if any element is stored or not
int isFull()	indicates if stack is full or not

STACK APPLICATIONS

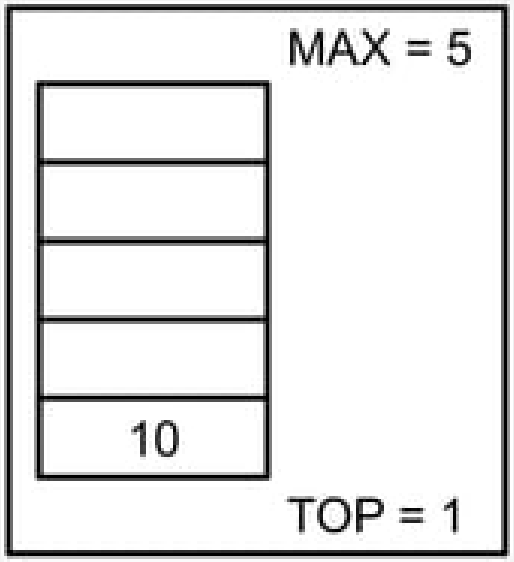
- Balancing of Symbols
- Infix to Postfix conversion
- Evaluation of postfix expression
- Implementing function calls
- History of a web browser
- Undo Sequence
- Tree Traversal

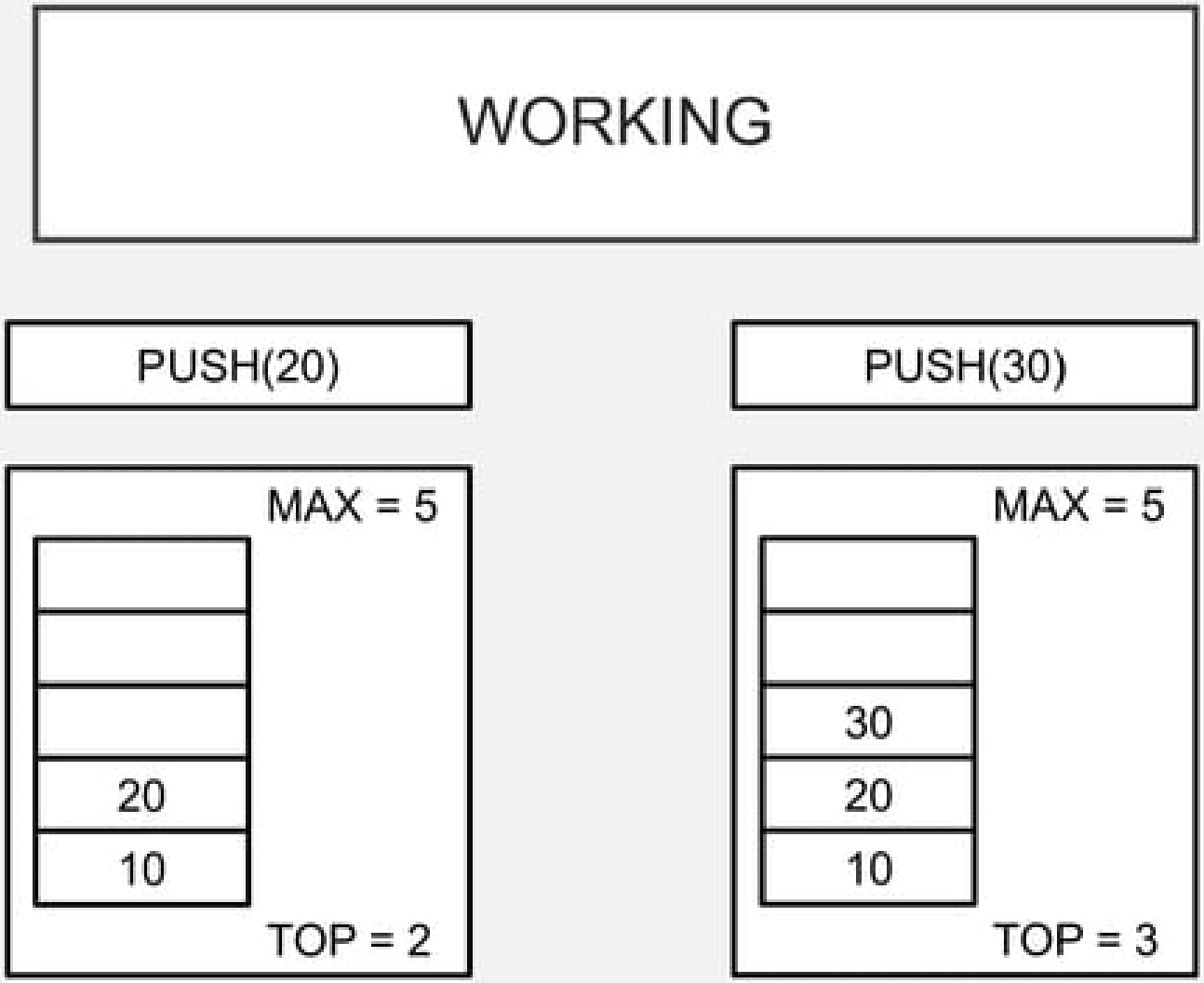
WORKING

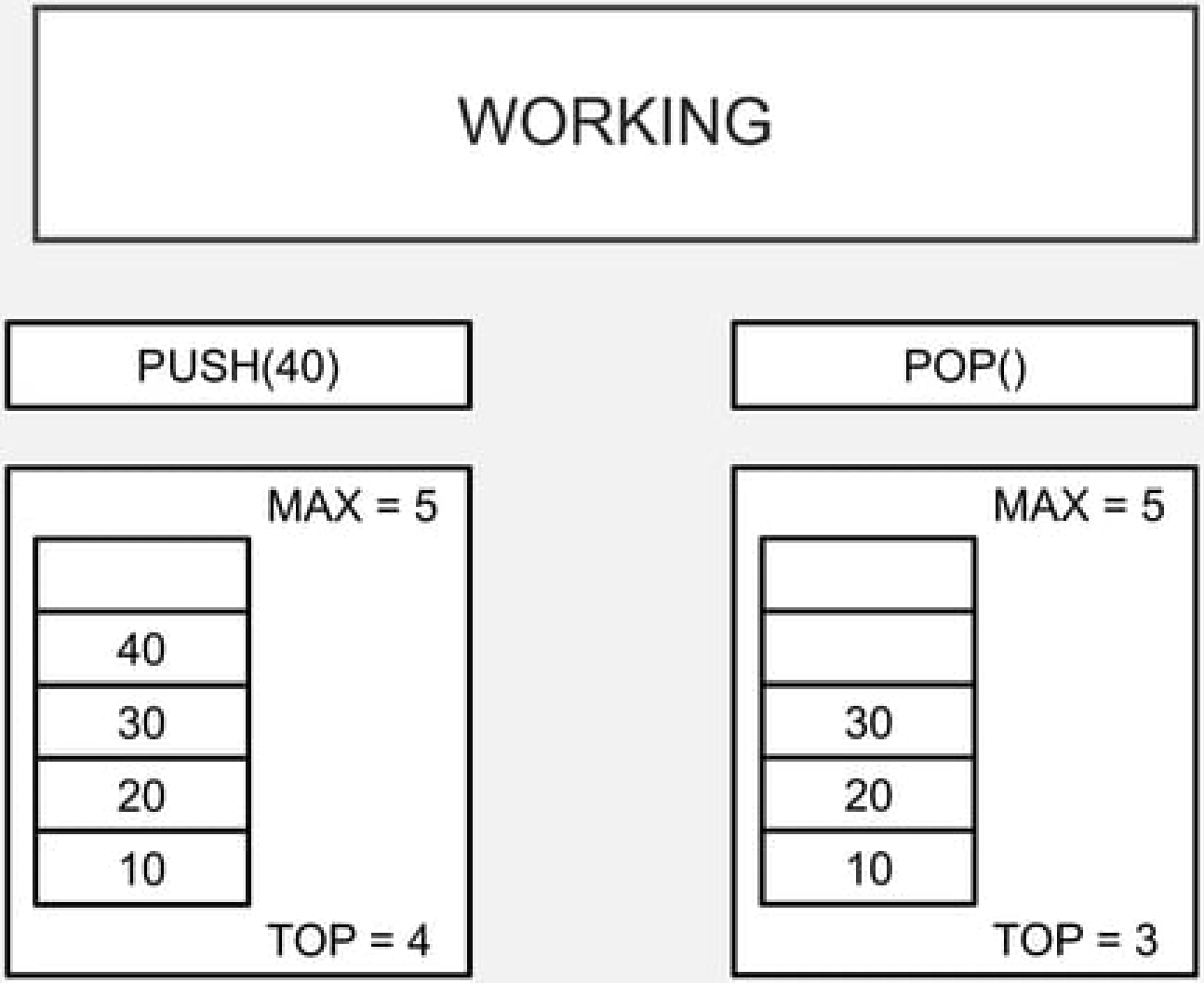
CREATE STACK
SIZE 5

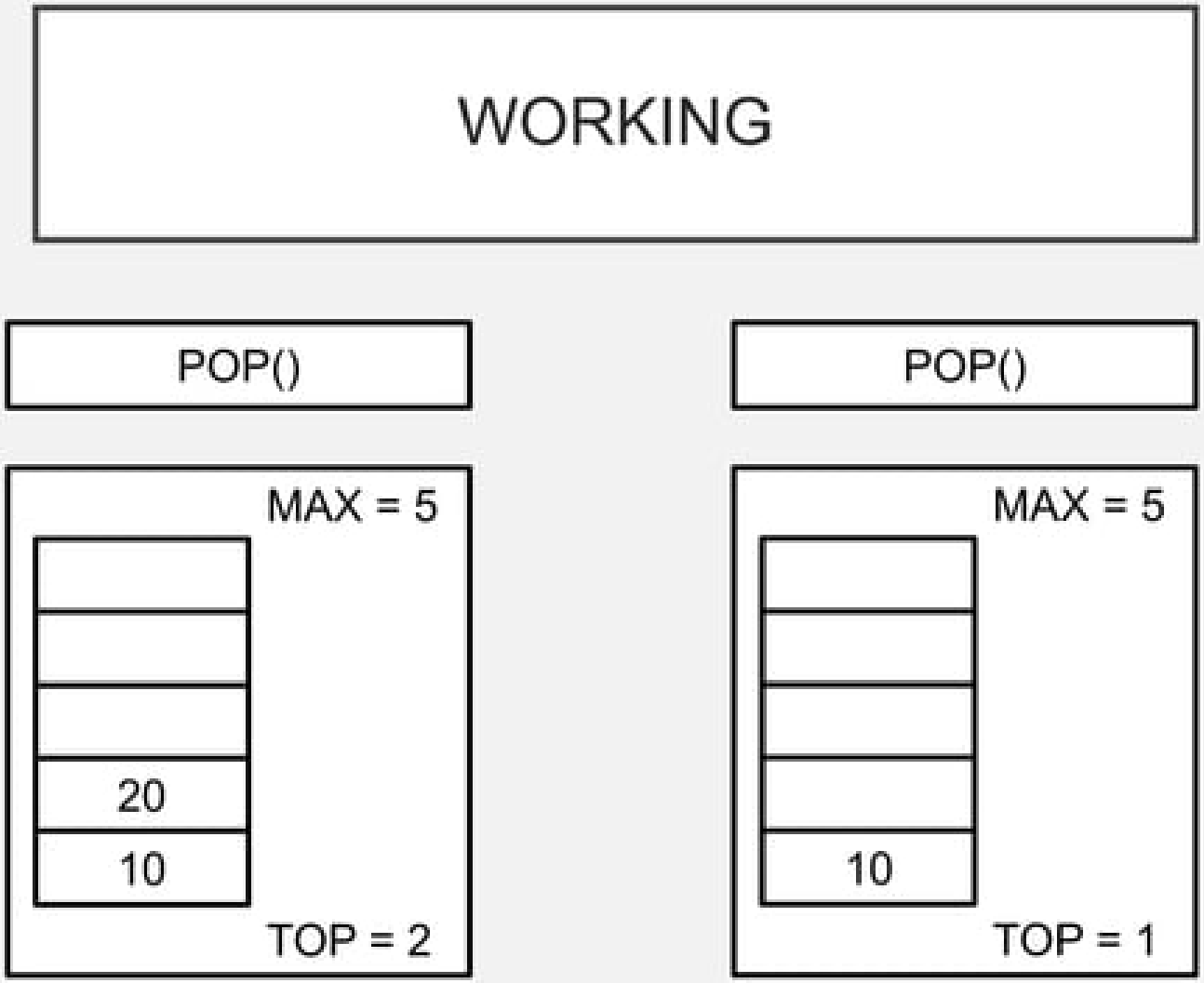


PUSH(10)

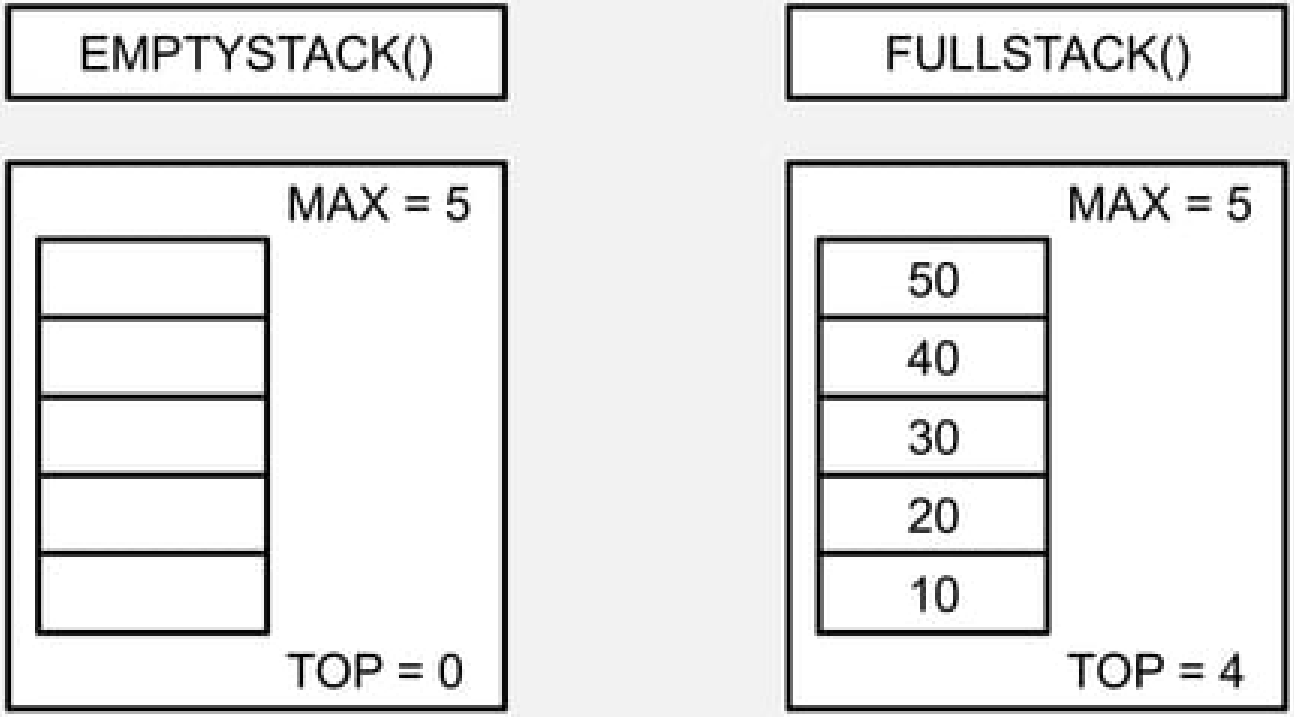



WORKING
PUSH(20)
MAX = 5
20
10
TOP = 2
PUSH(30)
MAX = 5
30
20
10
TOP = 3


WORKING
PUSH(40)
POP()
MAX = 5
40
30
20
10
TOP = 4
MAX = 5
30
20
10
TOP = 3


WORKING
POP()
MAX = 5
20
10
TOP = 2
POP()
MAX = 5
10
TOP = 1

WORKING



IMPLEMENTATION BY ARRAY

- Advantages
  - Best Performance
- Disadvantages
  - Fixed Size
- Basic Implementation
  - Initially Empty Array
  - Field to record where next data is placed
  - if array is full, push(item) else return false
  - if array is empty, pop() return item on top else NULL

IMPLEMENTATION BY ARRAY

- CREATING STRUCTURE
  - struct ArrayStruct{
    - int top; //Keep track of top Element
    - int capacity; //capacity of the stack
    - int *array; //pointer to address of first index
  - }
- Space Complexity (for *n* operations)– O*(n)*

IMPLEMENTATION BY ARRAY

- isEmpty(){
  - return (S->top == -1);
- }

- Time Complexity
  - O(1)

- isFull(){
  - return(S->top==S->capacity-1);
- }

- Time Complexity
  - O(1)

IMPLEMENTATION BY ARRAY

- void push(struct Array *S,int data){
  - if(isFull(S))
    - cout<<“FULL”;
  - else
    - S->array[++S->top]=data;
- }

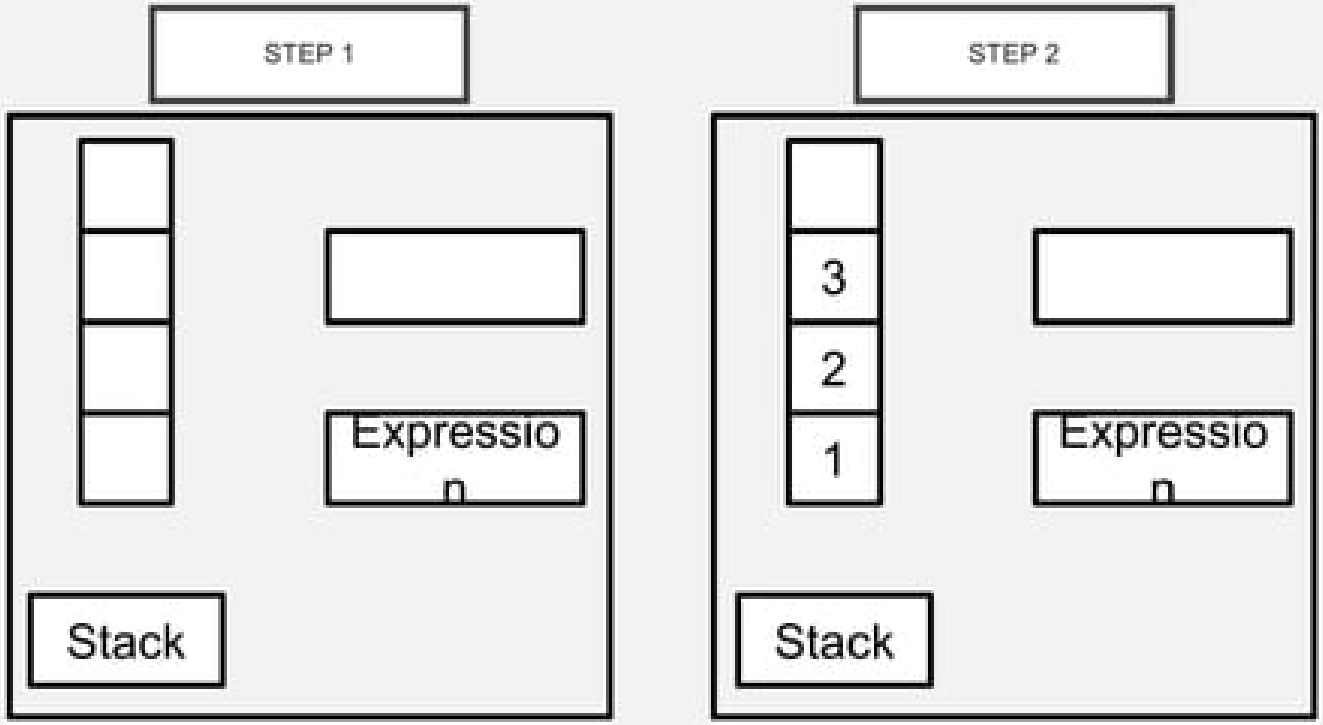
- Time Complexity
  - O(1)

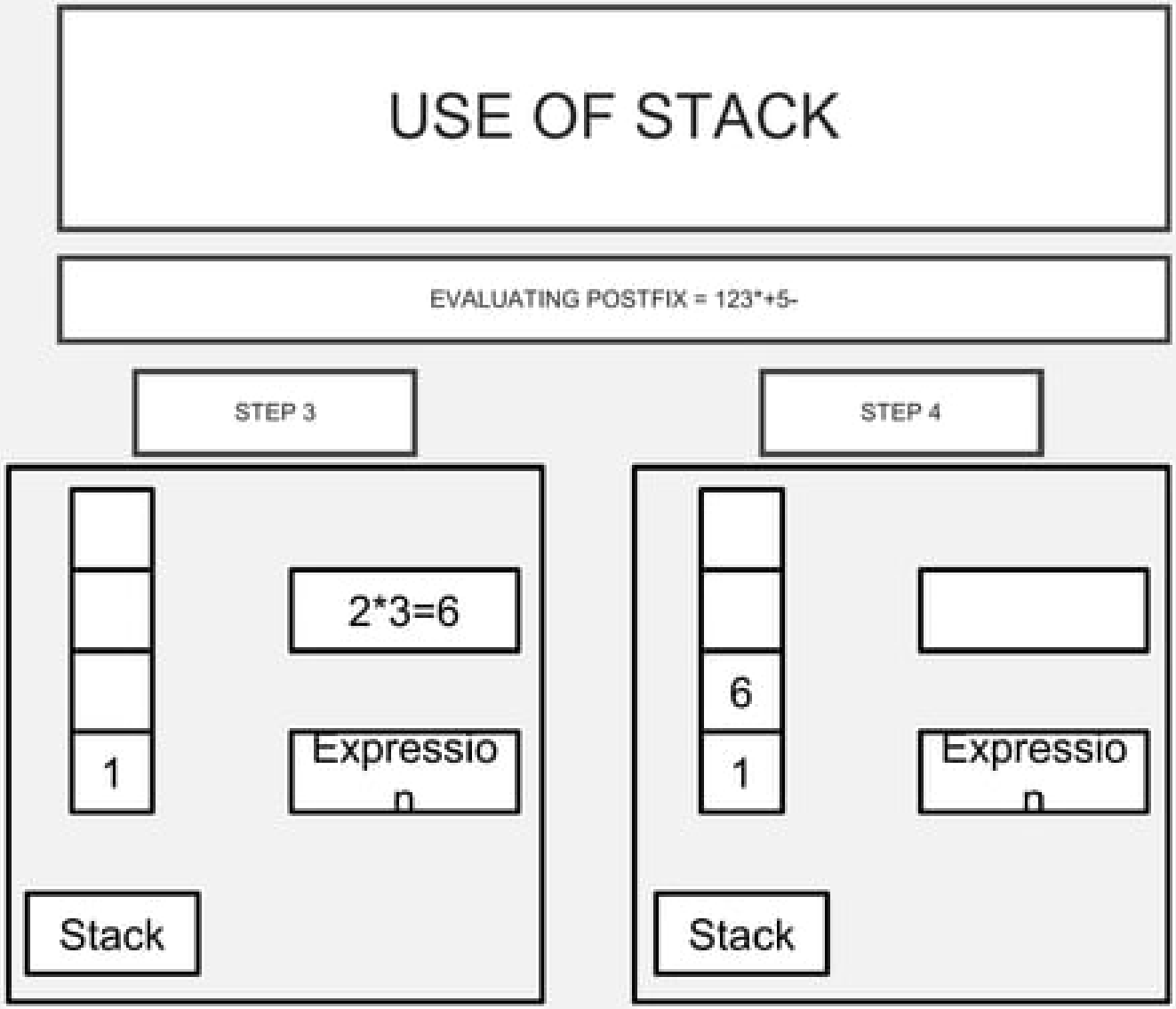
- int pop(struct Array *S){
  - if(isEmpty(S)){
    - cout<<“EMPTY”;
    - return;}
  - else
    - return (S->array[S->top]);
- }

- Time Complexity
  - O(1)

USE OF STACK

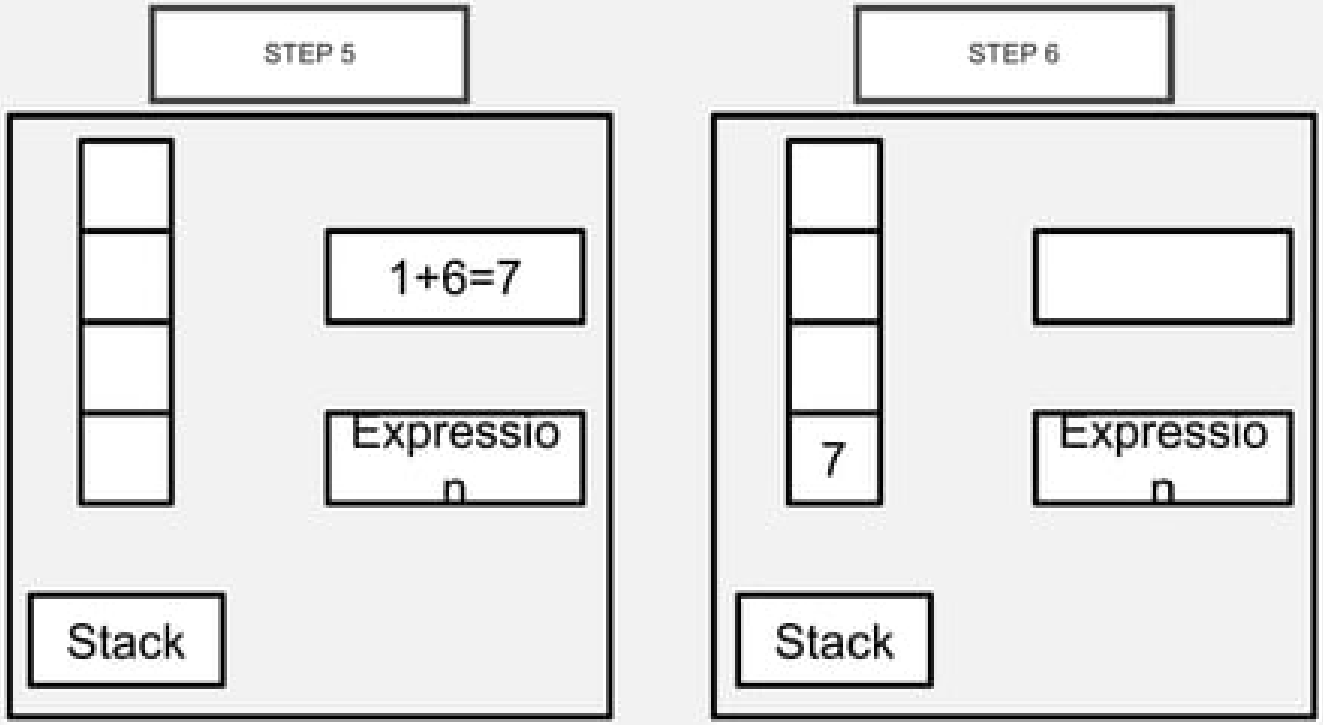
EVALUATING POSTFIX = 123*+5-




USE OF STACK
EVALUATING POSTFIX = 123*+5-
STEP 3
2*3=6
1
Expressio
n
Stack
STEP 4
6
1
Expressio
n
Stack

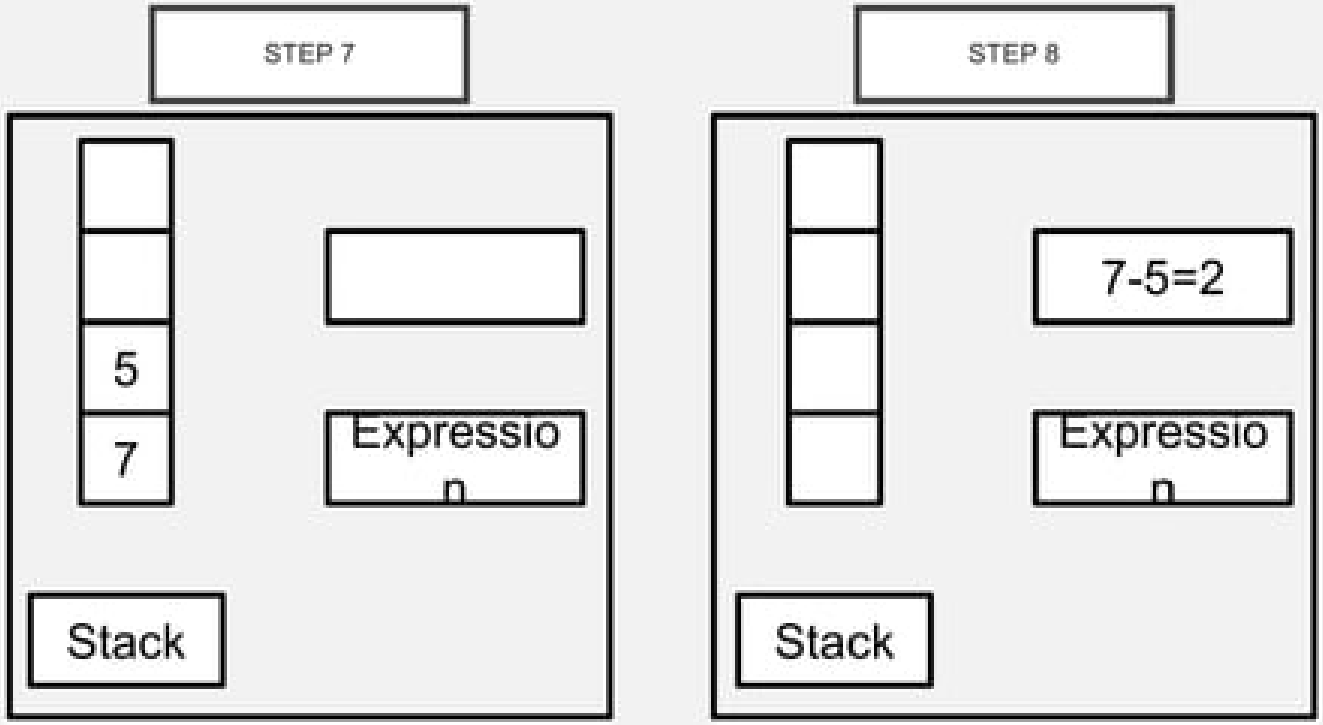
USE OF STACK

EVALUATING POSTFIX = 123*+5-



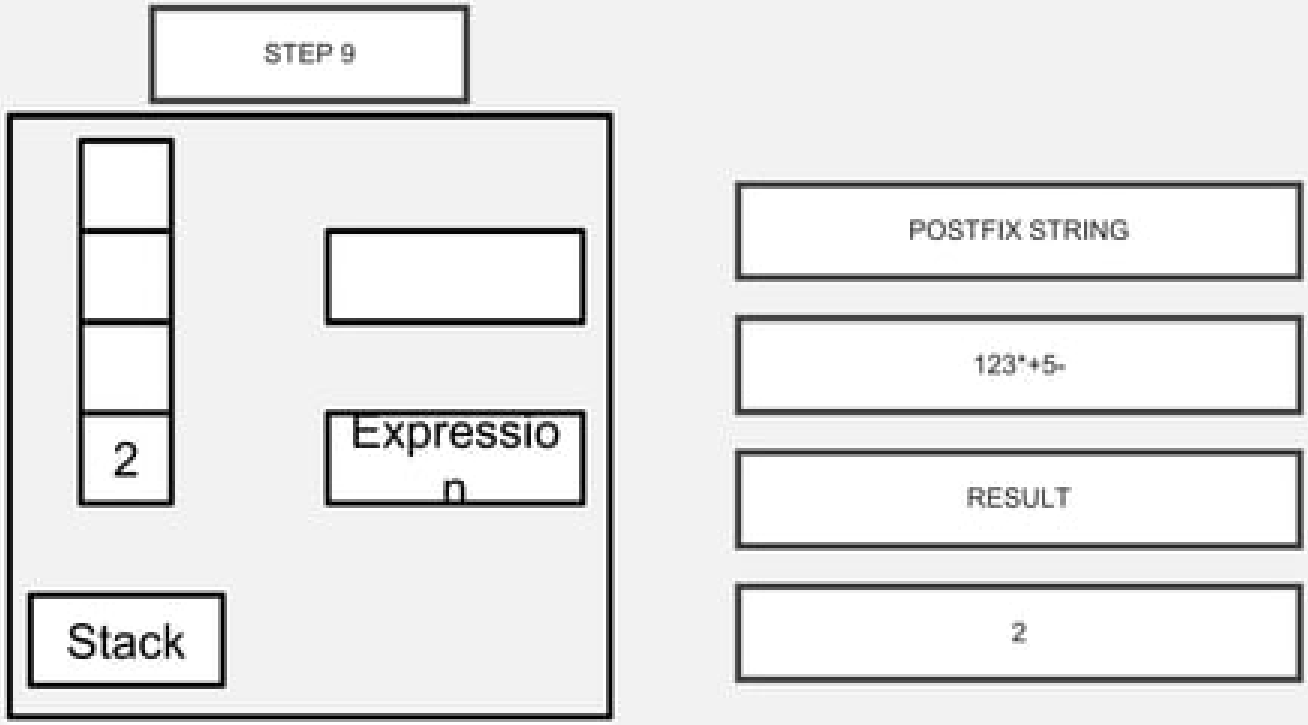
USE OF STACK

EVALUATING POSTFIX = 123*+5-



USE OF STACK

EVALUATING POSTFIX = 123*+5-



REFERENCES

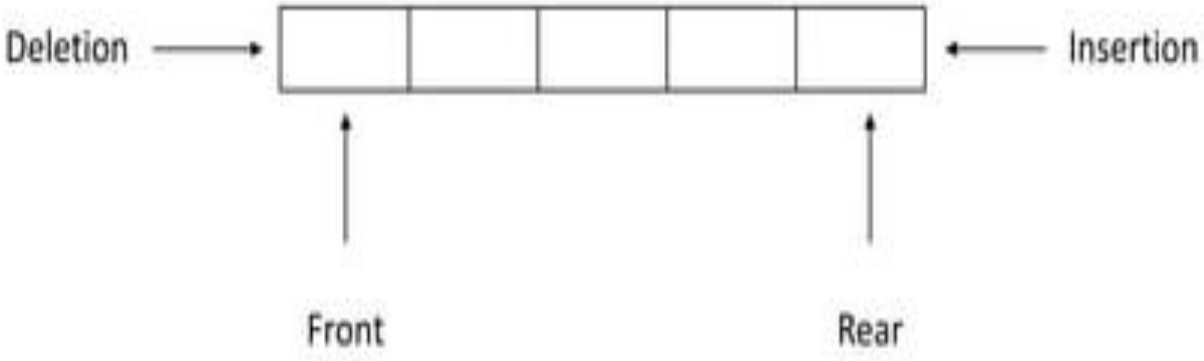
- Data Structures and Algorithms, Narsimha Karumanchi
- https://www.geeksforgeeks.org/stack-data-structure/
- https://www.geeksforgeeks.org/tag/data-structures-stack/

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 8
STACKS AND QUEUES
ENQUEUE AND DEQUEUE OPERATION

Definition of Queue

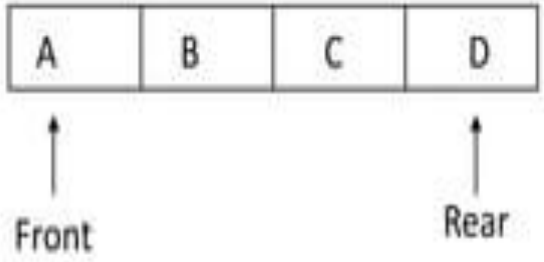
An ordered collection of items from which items may be deleted from one end called the front and into which items may be inserted from other end called rear is known as **Queue**.
It is a linear data structure.
It is called **First In First Out (FIFO)** list. Since in queue, first element will be the first element out.



(a) Insertion and deletion of elements in Queue

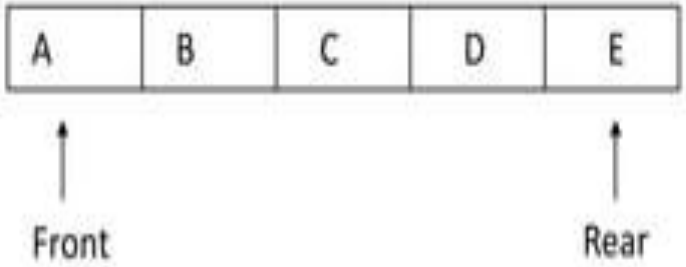
Queue (Example)

Queue with four elements A, B, C and D.
A is at front and D is at rear.



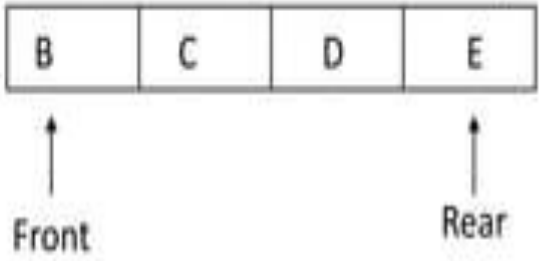
(b) Queue

Element E will be added at the rear end.



(c) Front and near pointer after insertion

Element can be deleted only from the front.
Thus, A will be the first to be removed from the queue.



Difference between Stack & Queue.

SR	Stack	Queue
1	Stack is a LIFO (Last in first out) data structure.	Queue is a FIFO (first in first out) data structure.
2	In a stack, all insertion and deletions are permitted only at one end of stack called top of stack.	In a queue items are inserted at one end called the rear and deleted from the other end called the front.
3	Stack is widely used in recursion.	Queue is more useful in many of the real life applications.
4	**Example**: A stack of plates, a stack of coin etc.	**Example**: A queue of people waiting to purchase tickets, in a computer system process waiting for line printer etc.

Operations on a Queue.

1. **Create**: Creates empty queue.
2. **Insert**: Add new element at rear.
3. **Delete**: Delete element at front.
4. **Isempty**: Checks queue is empty or not.
5. **Isfull**: Checks queue is full or not.

Algorithm to insert and delete element in the Queue.

1. Create an empty queue by giving name and
2. Initially assign Rear = -1, front = -1.
3. If choice == Insert then
   if Rear == max-1 then
   print "queue is full"
   else
   Rear = Rear +1
   q [Rear] = element
4. If choice == Delete then
   If front == -1 then
   print "queue is empty"
   else
   front = front +1
5. Stop

```
void enqueue(int x)
{
 if(rear == n-1)
 {
  printf("overflow");
 }

elseif(front == -1 && rear == -1)
 {
  front = rear = 0;
  queue[rear] = x;
 }
else
 {
  rear++;
  queue[rear] = x;
 }
}
```

```
void dequeue()
{
if(front == -1 && rear == -1)
 {
  printf("Queue is empty");
 }

elseif( front == rear)
 {
  printf("%d", queue[front]);
  front = rear = -1;
 }
else
 {
  printf("%d", queue[front]);
  front++;
 }
```

```
void display()
{
 int i;
 if(front == -1 &
 {
  printf("Qu
 }
else
{
 for(i = front;
  {
   printf("%d
  }
 }
}
```

n = 4

```
void dequeue()
{
if(front==-1 && rear==-1)
 {
  printf("Queue is empty");
 }

else if(front==rear)
 {
  printf("%d", queue[front]);
  front=rear=-1;
 }
else
  {
```

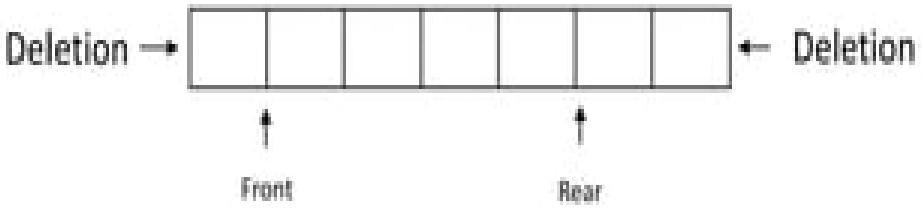
```
void display()
{
  int i;
 if(front==-1 && rear==-1)
  {
    printf("Queue is empty");
  }
 else
 {
   for(i=front; i<=rear; i++)
     {
       printf("%d", queue[i]);
     }
```

Queue as an abstract data type

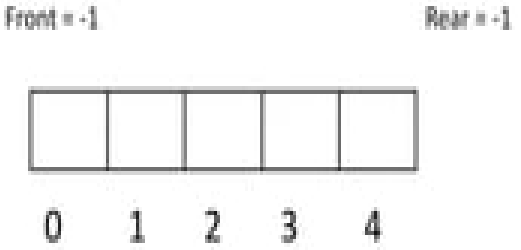
(1) Initialize the queue to be empty.
(2) Check if the queue is empty or not.
(3) Check if the queue is full or not.
(4) Insert a new element in the queue if it is not full.
(5) Retrieve the value of first element of the queue, if the queue is not empty.
(6) Delete the first element from the queue if it is not empty.

Queue abstract data type = Queue elements + Queue operations.

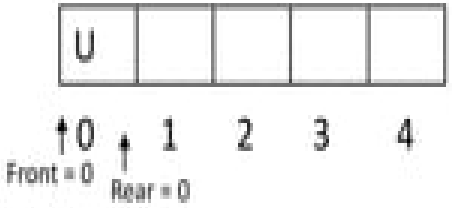
Representation of an Queue as an Array



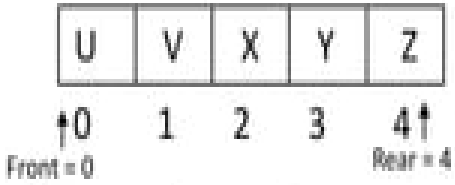
(a) Queue with array



(b) Empty Queue



(c) Queue after adding an element U



(d) Queue after adding 5 element

Types of Queue

1. Linear Queue.
2. Circular Queue.
3. Double ended Queue.
4. Priority Queue.

Application of Queue

- Queue are used in computer for scheduling of resources to application . These resources are CPU, Printer etc.
- In batch Programming, multiple jobs are combined into a batch and the first program is executed first, the second is executed next and so on in a sequential manners.
- A queue is used in break first search traversal of a graph & level wise traversal of tree.
- Computer simulation, Airport simulation.

THANK YOU

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

Week 9

Linked Lists Operation:

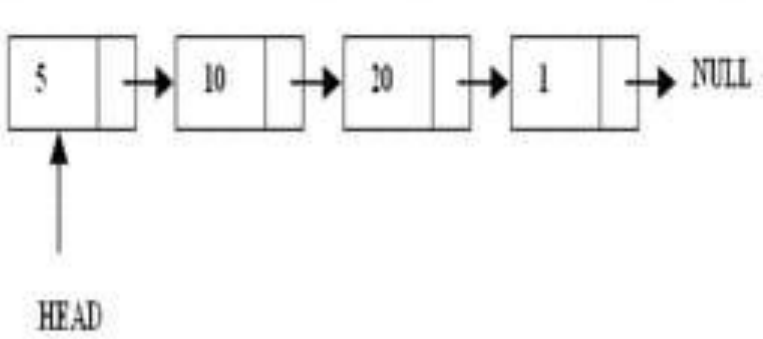
Create, traverse, search, insert, and delete operations in linked lists

TYPES

- Linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.
- Linked list comprise of group or list of nodes in which each node have link to next node to form a chain
- Types of linked list
  - Singly linked list
  - Doubly linked list
  - Circular linked list

What are Linked Lists

- A linked list is a linear data structure.
- Nodes make up linked lists.
- Nodes are structures made up of data and a pointer to another node.
- Usually the pointer is called next.



Partho Sarathi SArker
Asst Professor: Dept. Of CSE

Arrays Vs Linked Lists

Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

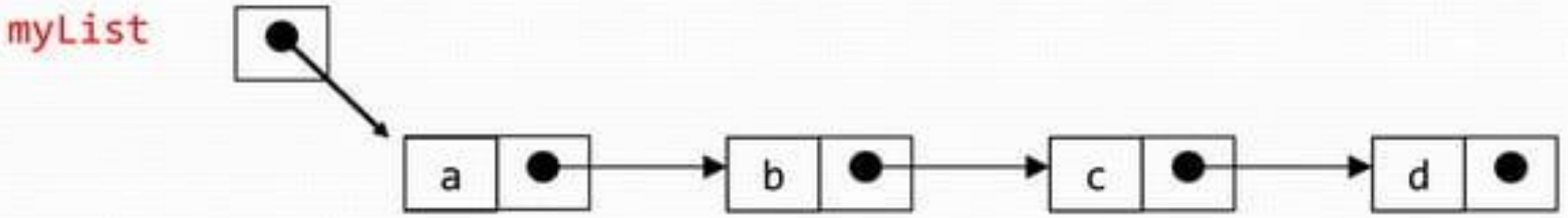
Singly Linked List

- Each node has only one link part
- Each link part contains the address of the next node in the list
- Link part of the last node contains NULL value which signifies the end of the node

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

Schematic representation

- Here is a singly-linked list (SLL):



- Each node contains a value(data) and a pointer to the next node in the list

- **myList** is the header pointer which points at the first node in the list

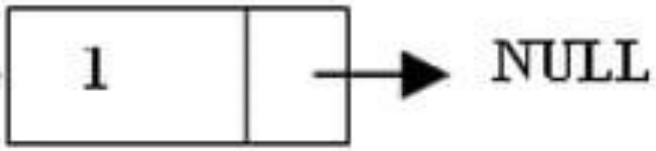
Basic Operations on a list

- Creating a List
- Inserting an element in a list
- Deleting an element from a list
- Searching a list
- Reversing a list

Creating a node

```
struct node{
        int data;              // A simple node of a linked list
        node*next;
    }*start;                   //start  points at the first node
start=NULL ;                      initialised to NULL at beginning
```

```
node* create( int  num)  //say num=1 is passed from main
{
   node*ptr;
   ptr= new  node;  //memory allocated dynamically
        if(ptr==NULL)
          'OVERFLOW' // no memory available
            exit(1);
        else
        {
           ptr->data=num;
           ptr->next=NULL;
           return ptr;
        }
}
```



To be called from main() as:-

```
void main()
{
    node* ptr;
    int data;
    cin>>data;
    ptr=create(data);
}
```

Inserting the node in a SLL

There are 3 cases here:-

- Insertion at the beginning
- Insertion at the end
- Insertion after a particular node

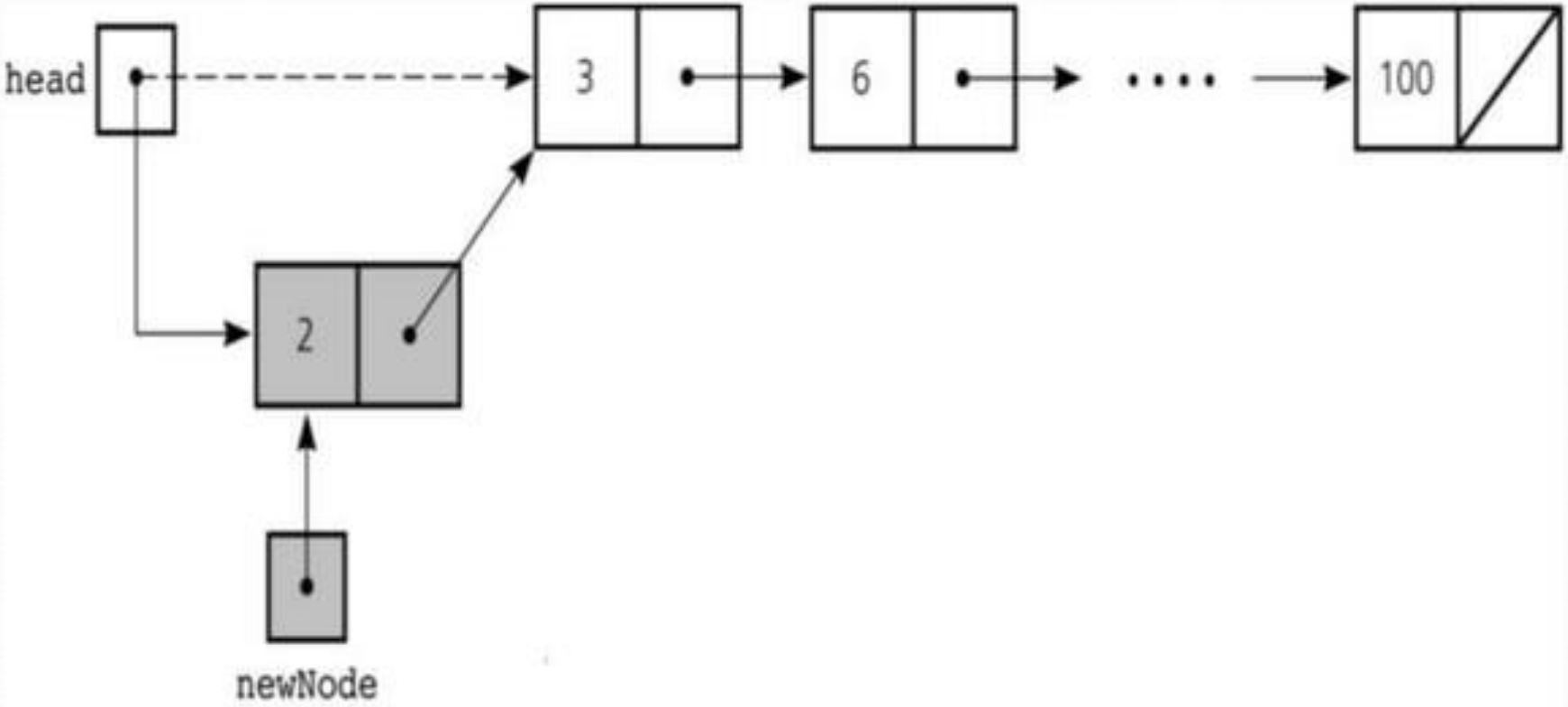
Insertion at the beginning

There are two steps to be followed:-

a) Make the next pointer of the node point towards the first node of the list

b) Make the start pointer point towards this new node

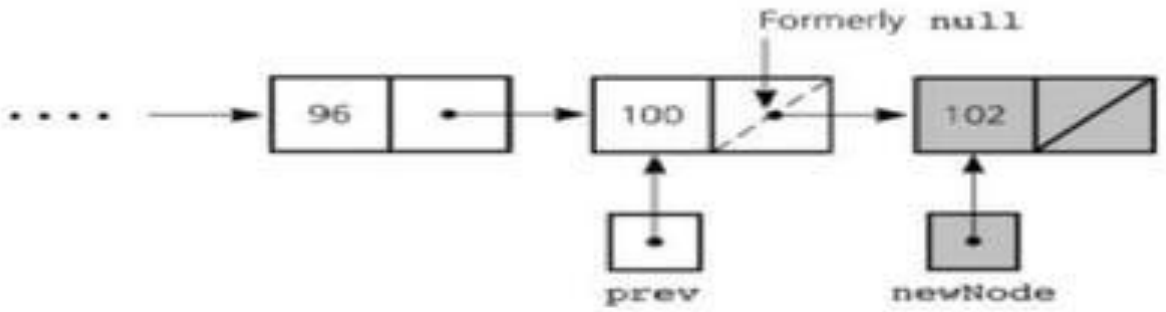
- If the list is empty simply make the start pointer point towards the new node;


head
3
6
100
2
newNode

```
void insert_beg(node* p)
{
node* temp;
        if(start==NULL)    //if the list is empty
        {
          start=p;
          cout<<"\nNode inserted successfully at the
                    beginning";
        }
        else {
              temp=start;
              start=p;
              p->next=temp;  //making new node point  at
            }                          the first node of the list
}
```

Inserting at the end

Here we simply need to make the next pointer of the last node point to the new node

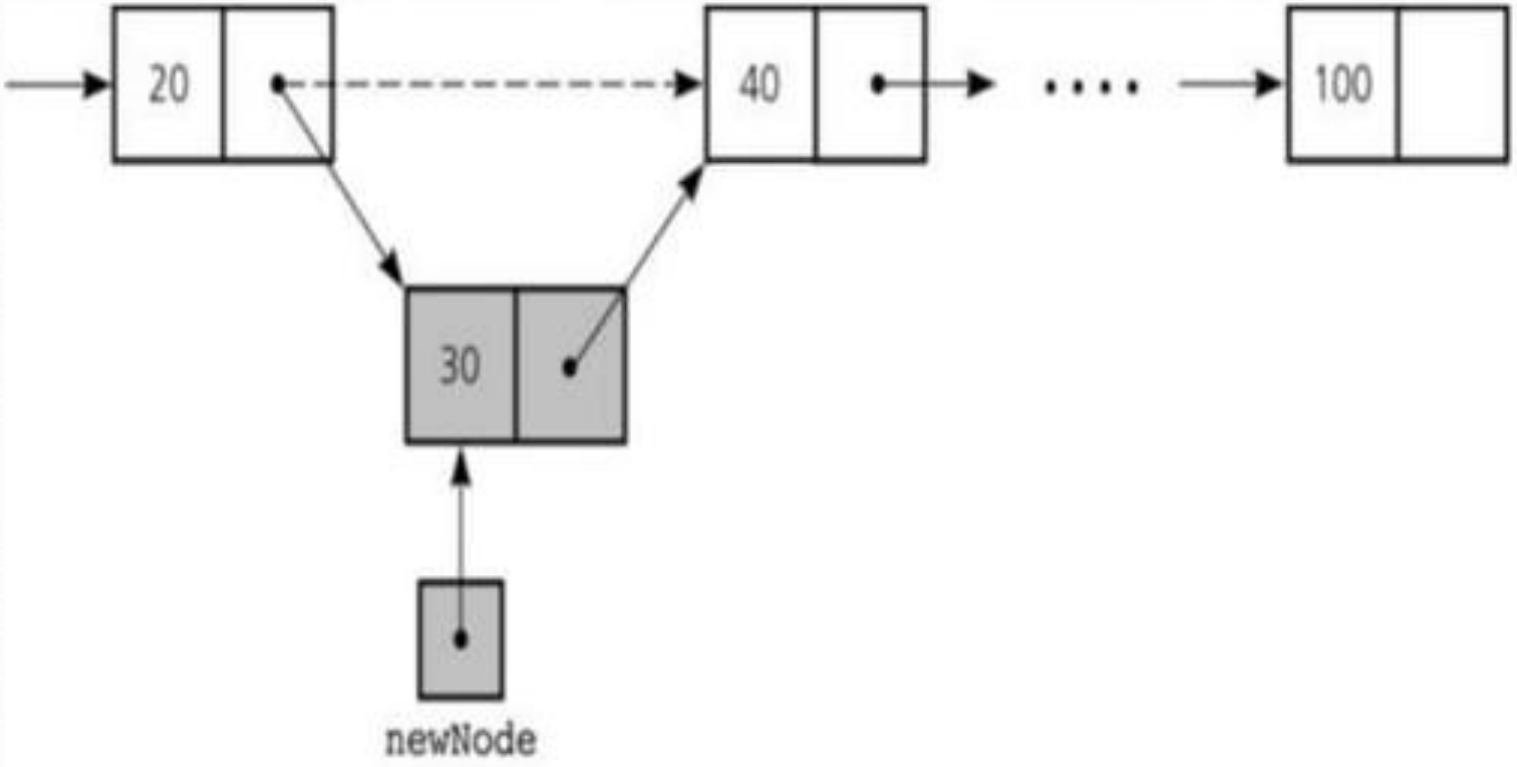


```
void insert_end(node* p)
{
node *q=start;
          if(start==NULL)
          {
            start=p;
            cout<<"\nNode inserted successfully at the end...!!!\n";
          }
        else{
                    while(q->link!=NULL)
                      q=q->link;
              q->next=p;
             }
}
```

Inserting after an element

Here we again need to do 2 steps :-

- Make the next pointer of the node to be inserted point to the next node of the node after which you want to insert the node
- Make the next pointer of the node after which the node is to be inserted, point to the node to be inserted


20
40
100
30
newNode

```
void insert_after(int c,node* p)
{
node* q;
q=start;
      for(int i=1;i<c;i++)
      {
        q=q->link;
              if(q==NULL)
             cout<<"Less than "<<c<<" nodes in the list...!!!";
      }
 p->link=q->link;
 q->link=p;
cout<<"\nNode inserted successfully";
}
```

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 10
LINKED LISTS OPERATION: CREATE, TRAVERSE, SEARCH, INSERT, AND DELETE OPERATIONS IN LINKED LISTS

Deleting a node in SLL

Here also we have three cases:-

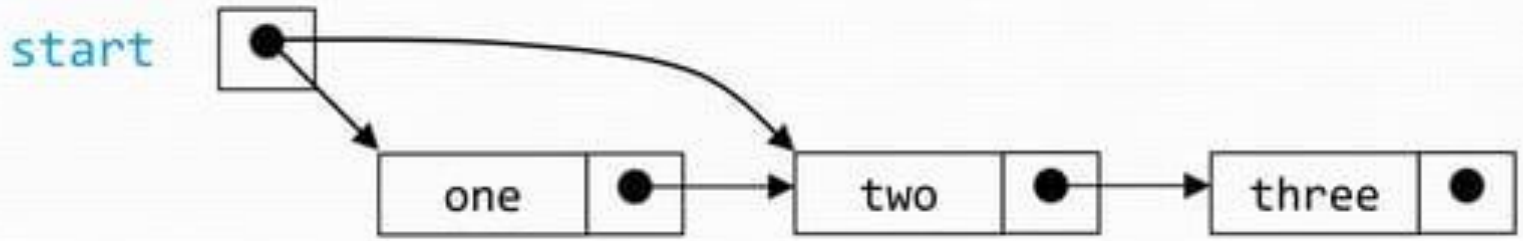
- Deleting the first node
- Deleting the last node
- Deleting the intermediate node

Partho Sarathi Sarker
Asst Professor: Dept. Of CSE

Deleting the first node

Here we apply 2 steps:-

- Making the start pointer point towards the 2nd node
- Deleting the first node using delete keyword



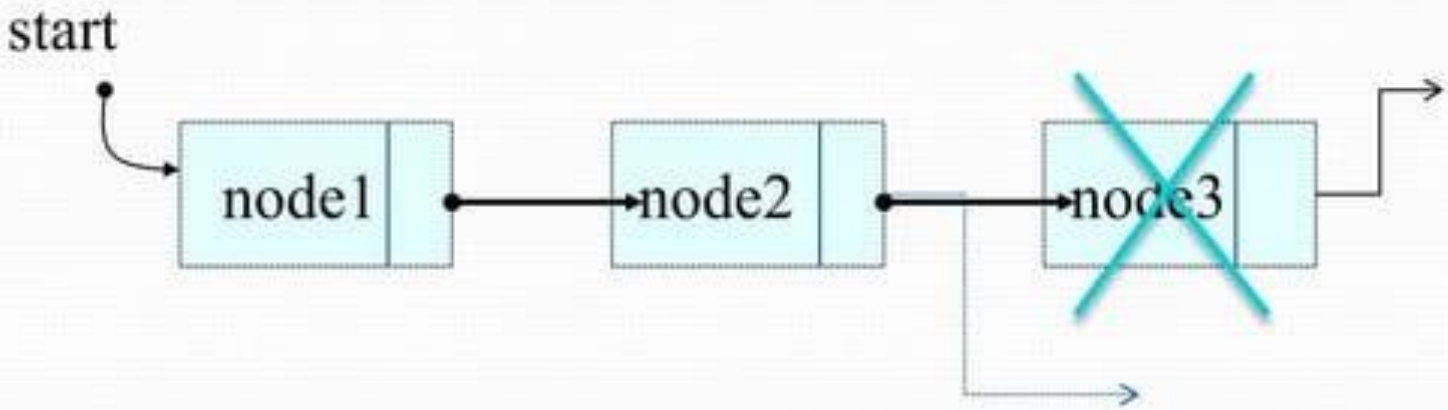
```
void del_first()
{
      if(start==NULL)
     cout<<"\nError.....List is empty\n";
     else
     {
      node* temp=start;
      start=temp->link;
      delete temp;
cout<<"\nFirst node deleted successfully....!!!";
     }
}
```

Partho Sarathi Sarker
Asst Professor: Dept. Of CSE

Deleting the last node

Here we apply 2 steps:-

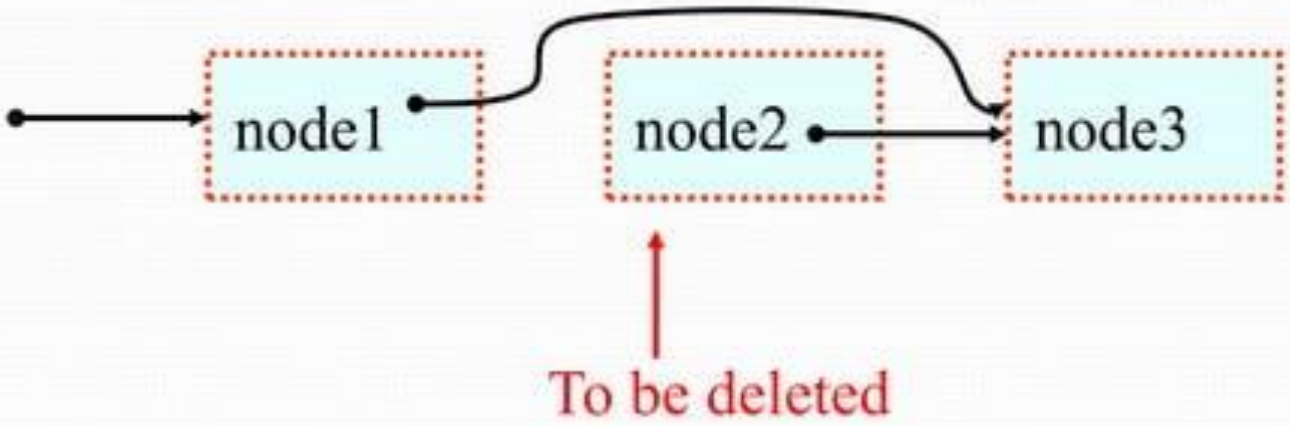
- Making the second last node's next pointer point to NULL
- Deleting the last node via delete keyword



```
void del_last()
{
    if(start==NULL)
    cout<<"\nError....List is empty";
    else
            {
               node* q=start;
                      while(q->link->link!=NULL)
                       q=q->link;
               node* temp=q->link;
               q->link=NULL;
               delete temp;
               cout<<"\nDeleted successfully...";
            }
    }
}
```

Deleting a particular node

Here we make the next pointer of the node previous to the node being deleted ,point to the successor node of the node to be deleted and then delete the node using delete keyword



```
void del(int c)
{
node* q=start;
      for(int i=2;i<c;i++)
      {
       q=q->link;
            if(q==NULL)
            cout<<"\nNode not found\n";
      }
      if(i==c)
      {
       node* p=q->link;        //node to be deleted
      q->link=p->link;      //disconnecting the node p
      delete p;
      cout<<"Deleted Successfully";
      }
}
```

Searching a SLL

- Searching involves finding the required element in the list
- We can use various techniques of searching like linear search or binary search where binary search is more efficient in case of Arrays
- But in case of linked list since random access is not available it would become complex to do binary search in it
- We can perform simple linear search traversal

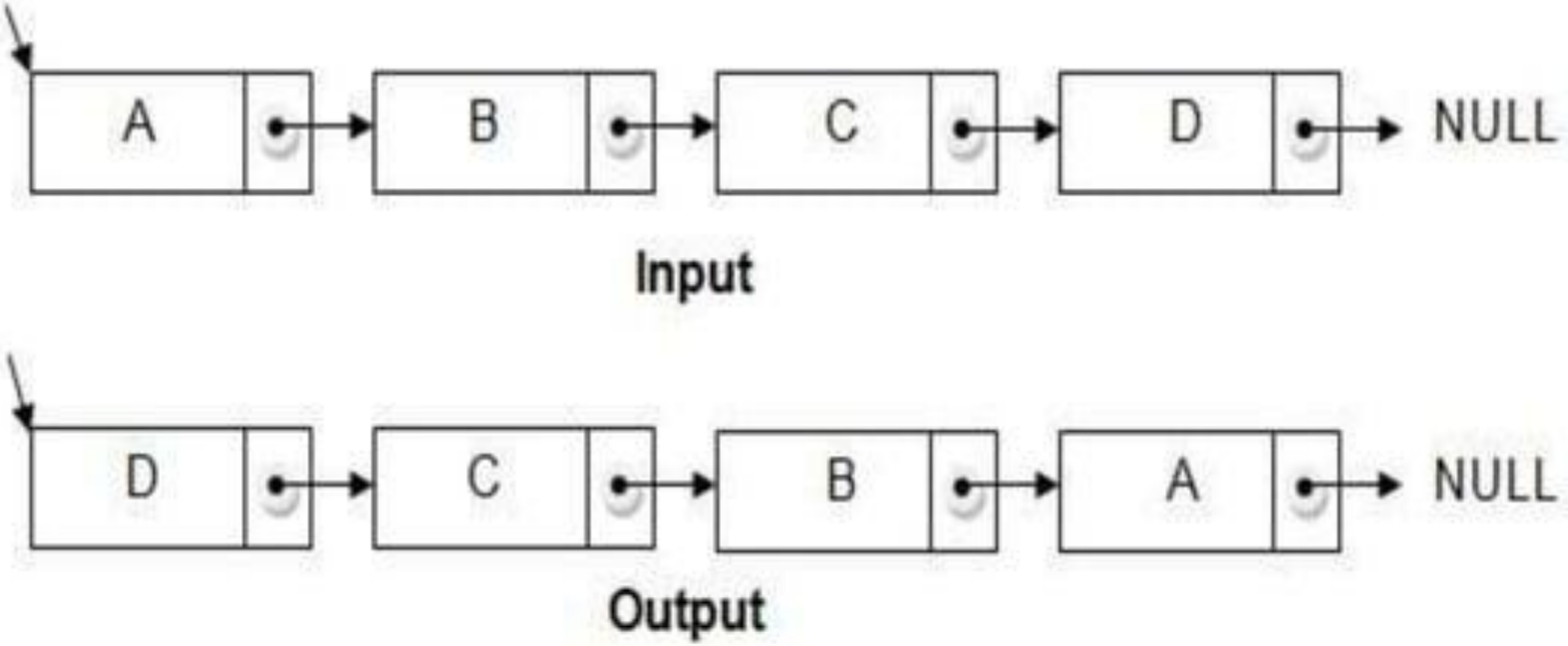
In linear search each node is traversed till the data in the node matches with the required value

```
void search(int x)
{
  node*temp=start;
    while(temp!=NULL)
      {
        if(temp->data==x)
         {
          cout<<"FOUND "<<temp->data;
          break;
         }
       temp=temp->next;
      }
}
```

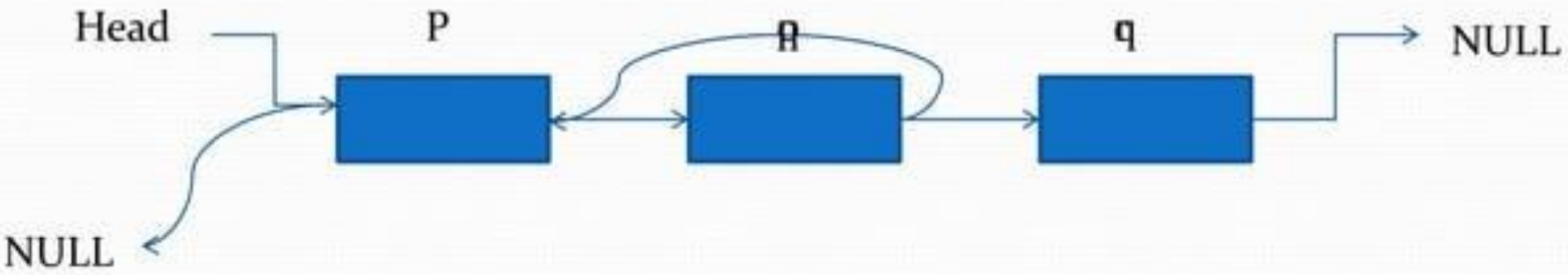
Partho Sarathi Sarker
Asst Professor: Dept. Of CSE

Reversing a linked list

- We can reverse a linked list by reversing the direction of the links between 2 nodes



- We make use of 3 structure pointers say p,q,r
- At any instant q will point to the node next to p and r will point to the node next to q



- For next iteration p=q and q=r
- At the end we will change head to the last node

Code

```
void reverse()
{
node*p,*q,*r;
   if(start==NULL)
   {
    cout<<"\nList is empty\n";
    return;
   }
p=start;
q=p->link;
p->link=NULL;
  while(q!=NULL)
  {
   r=q->link;
   q->link=p;
   p=q;
   q=r;
  }
  start=p;
  cout<<"\nReversed successfully";
}
```

Partho Sarathi Sarker
Asst. Professor, Dept. Of CSE

COMPLEXITY OF VARIOUS OPERATIONS IN ARRAYS AND SLL

Operation	ID-Array Complexity	Singly-linked list Complexity
Insert at beginning	O(n)	O(1)
Insert at end	O(1)	O(1) if the list has **tail** reference O(n) if the list has no **tail** reference
Insert at middle	O(n)	O(n)
Delete at beginning	O(n)	O(1)
Delete at end	O(1)	O(n)
Delete at middle	O(n): O(1) access followed by O(n) shift	O(n): O(n) search, followed by O(1) delete
Search	O(n) linear search O(log n) Binary search	O(n)
Indexing: What is the element at a given position k?	O(1)	O(n)

Partho Sarathi Sarker
Asst Professor: Dept. Of CSE

Doubly Linked List

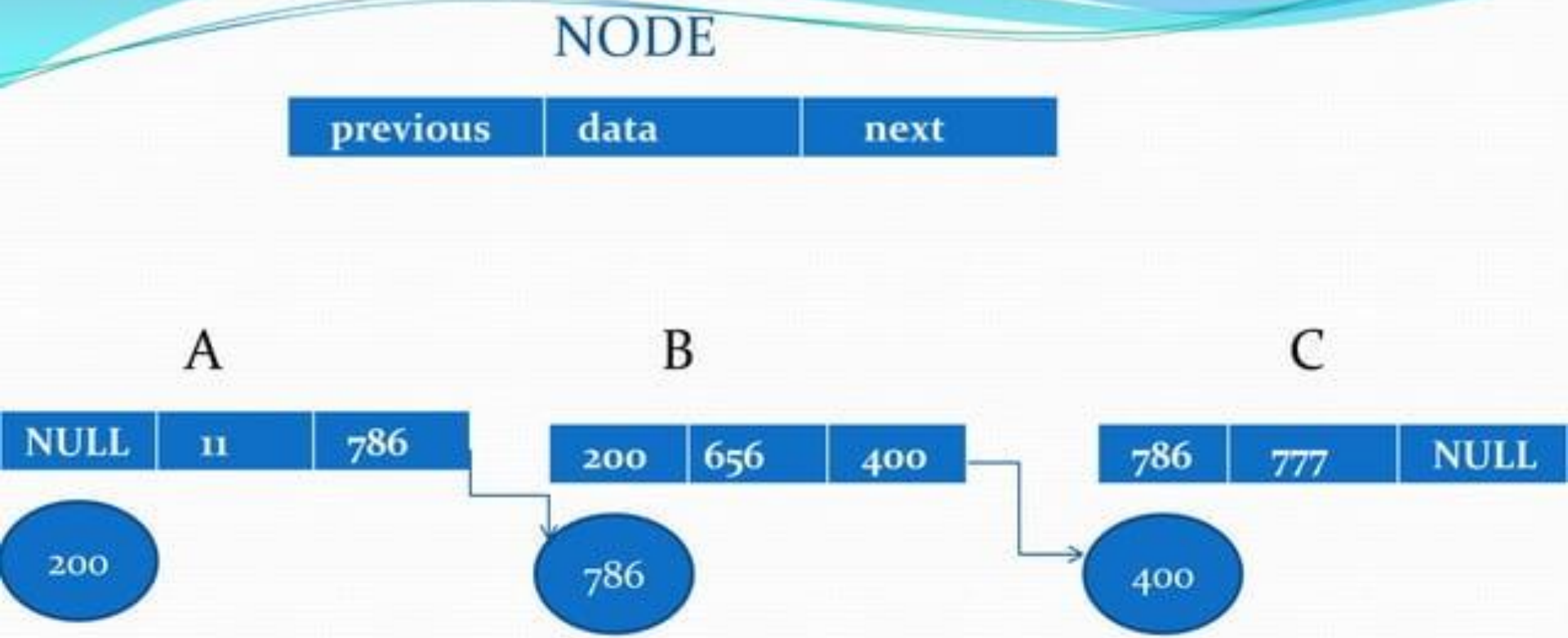
1. **Doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes.

2. Each node contains three fields ::
   -: one is data part which contain data only.
   -:two other field is links part that are point or references to the previous or to the next node in the sequence of nodes.

3. The beginning and ending nodes' **previous** and **next** links, respectively, point to some kind of terminator, typically a sentinel node or null to facilitate traversal of the list.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 11
LINKED LISTS OPERATION: CREATE, TRAVERSE, SEARCH, INSERT, AND DELETE OPERATIONS IN LINKED LISTS



A doubly linked list contain three fields: an integer value, the link to the next node, and the link to the previous node.

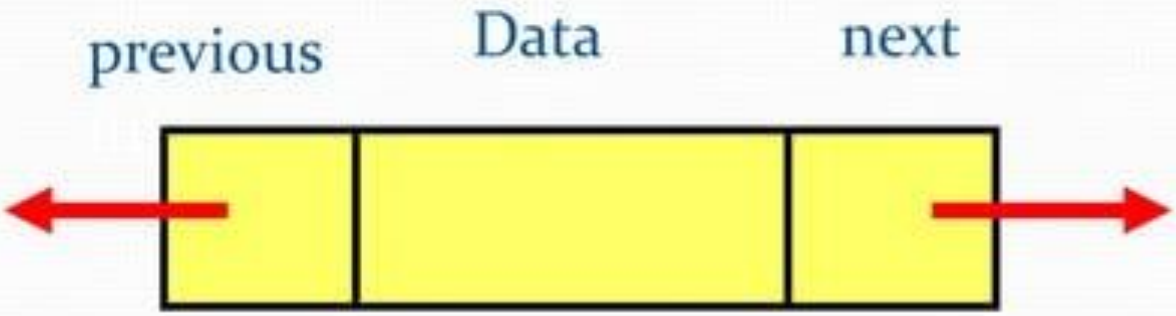
Partho Sarathi Sarker
Asst Professor: Dept. Of CSE

DLL's compared to SLL's

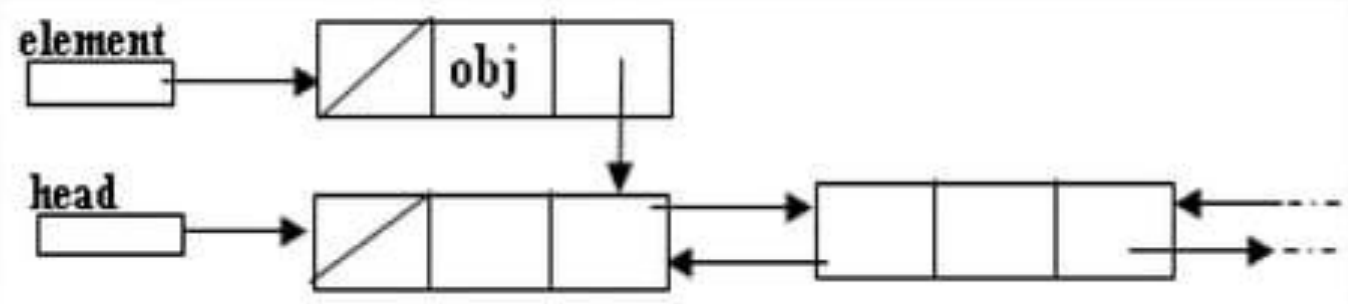
- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

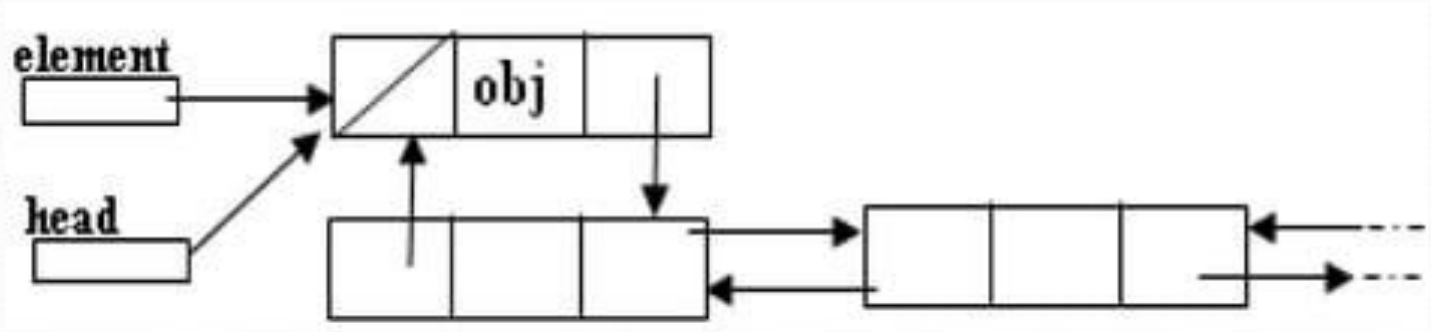
Structure of DLL

```
struct node
{
  int data;
  node*next;
  node*previous;   //holds the address of previous node
};
```



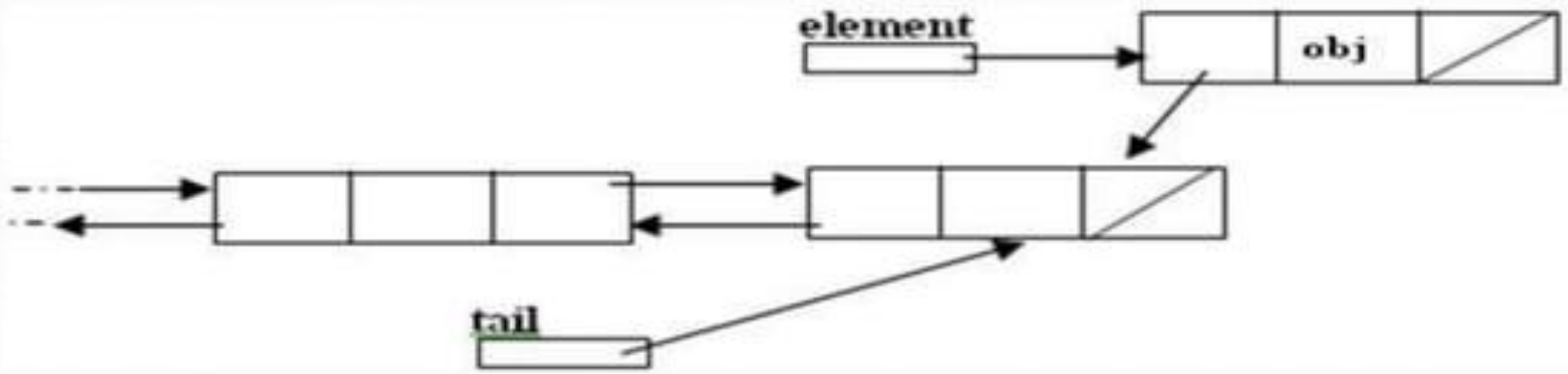
Inserting at beginning

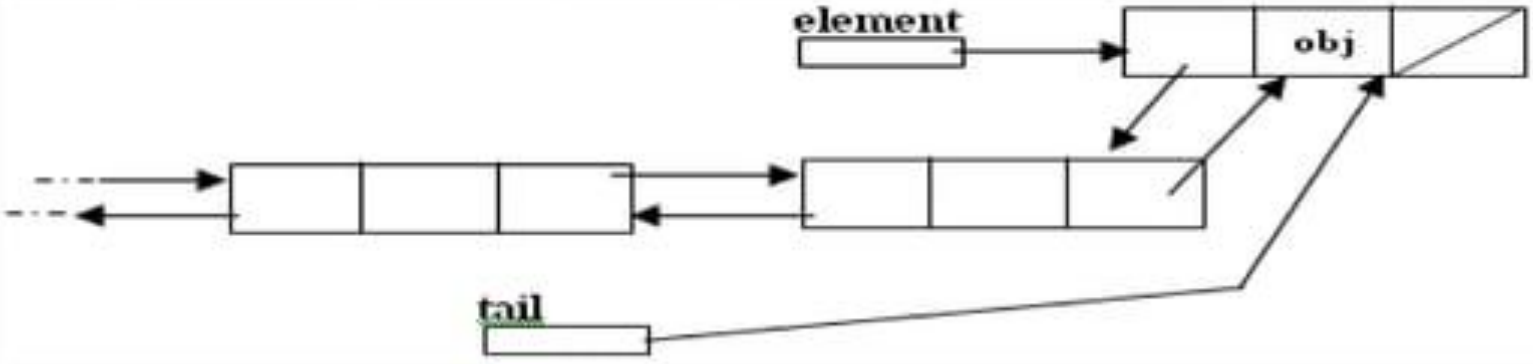




```
void insert_beg(node *p)
{
  if(start==NULL)
  {
   start=p;
   cout<<"\nNode inserted successfully at the beginning\m";
  }
  else
  {
  node* temp=start;
  start=p;
  temp->previous=p;    //making 1st node's previous point to the
   new node
  p->next=temp;        //making next of the new node point to the
   1st node
  cout<<"\nNode inserted successfully at the beginning\n";
  }
```

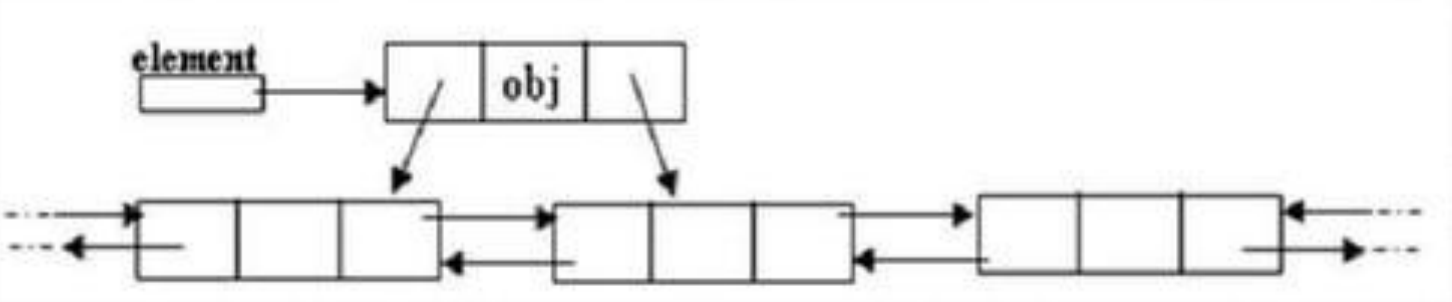
Inserting at the end



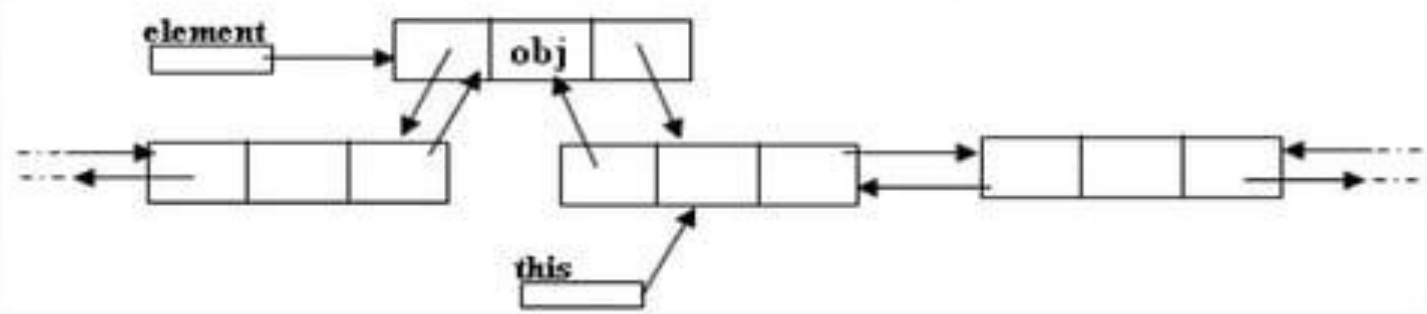


```
void insert_end(node* p)
{
    if(start==NULL)
    {
    start=p;
    cout<<"\nNode inserted successfully at the end";
    }
    else
    {
     node* temp=start;
      while(temp->next!=NULL)
      {
      temp=temp->next;
      }
     temp->next=p;
     p->previous=temp;
     cout<<"\nNode inserted successfully at the end\n";
    }
}
```

Inserting after a node



Making next and previous pointer of the node to be inserted point accordingly

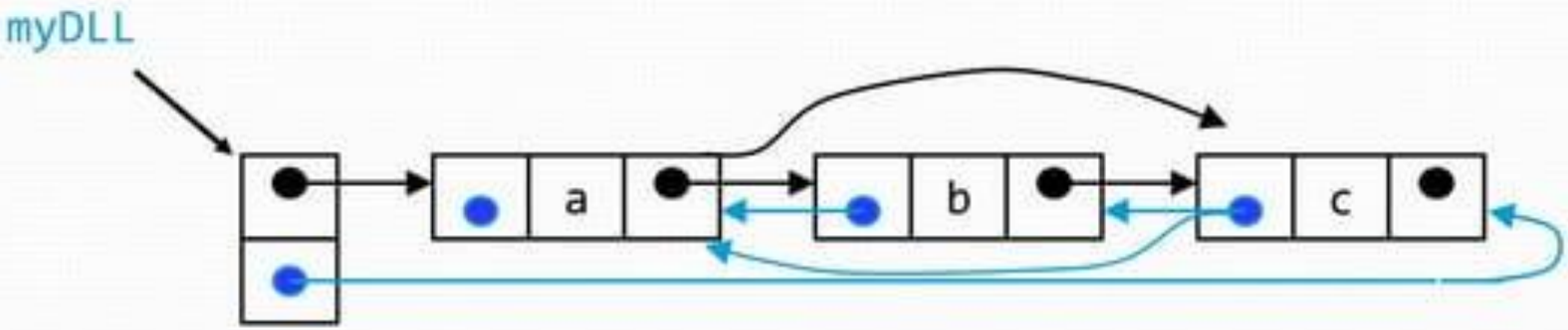


Adjusting the next and previous pointers of the nodes b/w which the new node accordingly

```
void insert_after(int c,node* p)
{
    temp=start;
    for(int i=1;i<c-1;i++)
    {
    temp=temp->next;
    }
    p->next=temp->next;
    temp->next->previous=p;
    temp->next=p;
    p->previous=temp;
    cout<<"\nInserted successfully";
}
```

Deleting a node

- Node deletion from a DLL involves changing *two* links
- In this example,we will delete node b

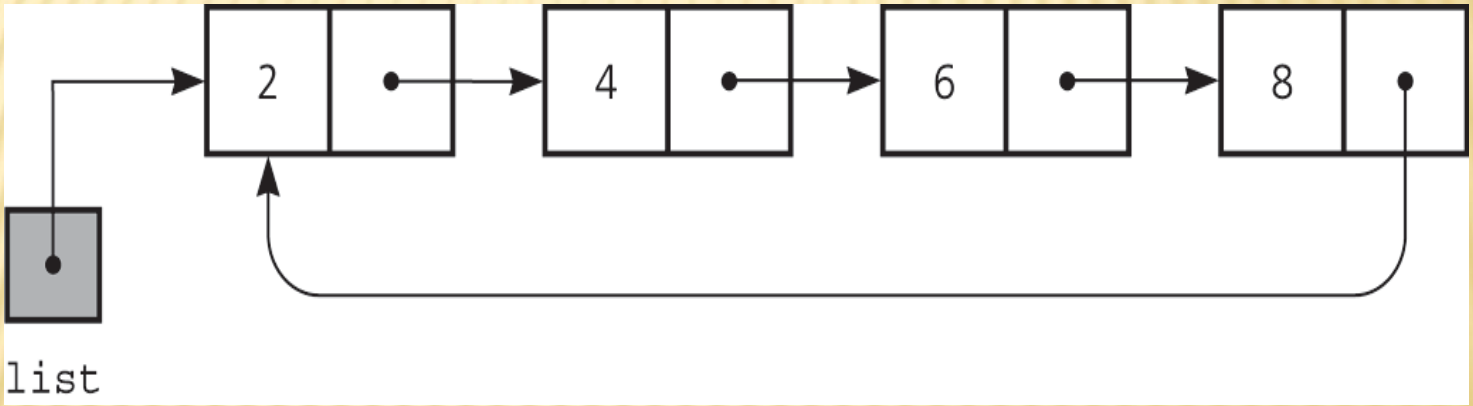


- We don't have to do anything about the links in node b
- Garbage collection will take care of deleted nodes
- Deletion of the first node or the last node is a special case

```
void del_at(int c)
{
    node*s=start;
     {
    for(int i=1;i<c-1;i++)
    {
     s=s->next;
    }
    node* p=s->next;
    s->next=p->next;
    p->next->previous=s;
    delete p;
    cout<<"\nNode number "<<c<<" deleted successfully";
     }
   }
}
```

CIRCULAR LINKED LISTS

- Last node references the first node
- Every node has a successor
- No node in a circular linked list contains *NULL*

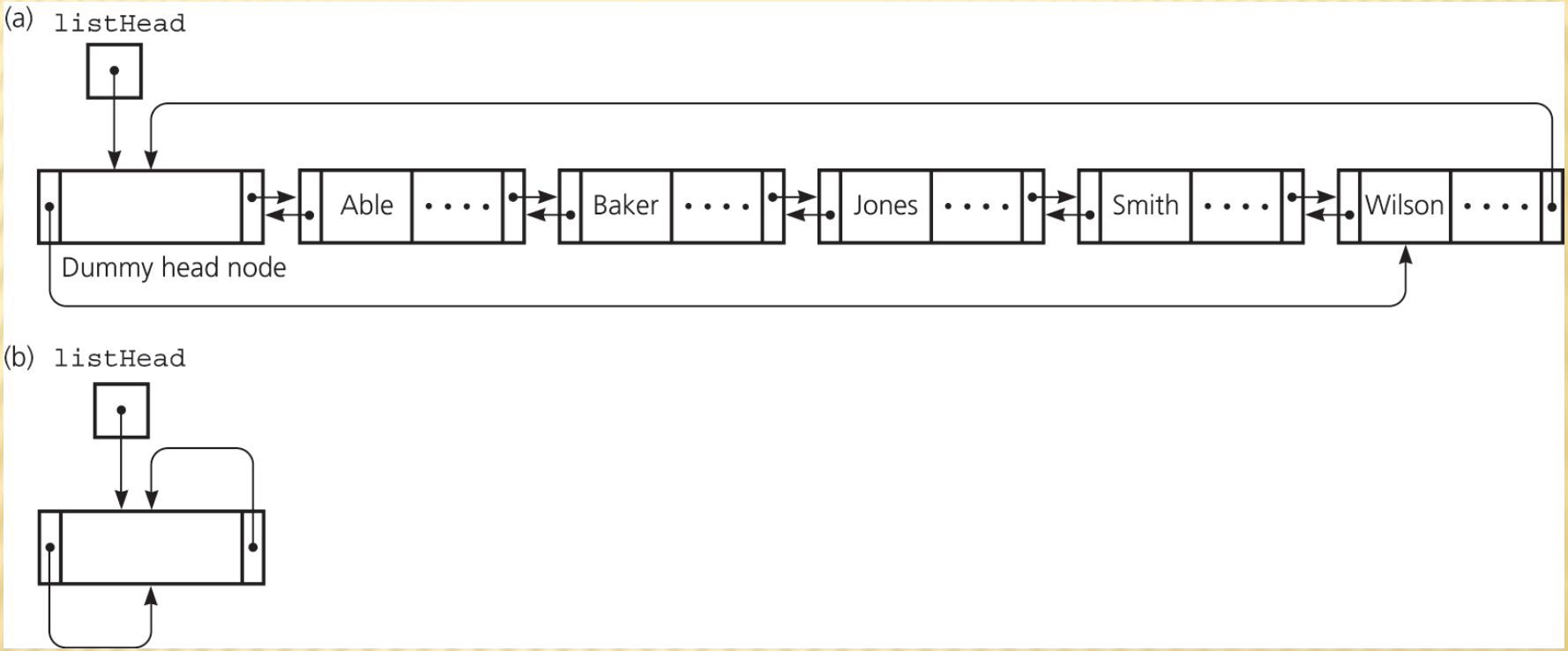


A circular linked list

CIRCULAR DOUBLY LINKED LISTS

- **Circular doubly linked list**
  - `prev` pointer of the dummy head node points to the last node
  - `next` reference of the last node points to the dummy head node
  - No special cases for insertions and deletions

CIRCULAR DOUBLY LINKED LISTS



(a) A circular doubly linked list with a dummy head node

(b) An empty list with a dummy head node

Partho Sarathi Sarker
Asst Professor: Dept. Of CSE

APPLICATIONS OF LINKED LIST

1. Applications that have an MRU list (a linked list of file names)

2. The cache in your browser that allows you to hit the BACK button (a linked list of URLs)

3. Undo functionality in Photoshop or Word (a linked list of state)

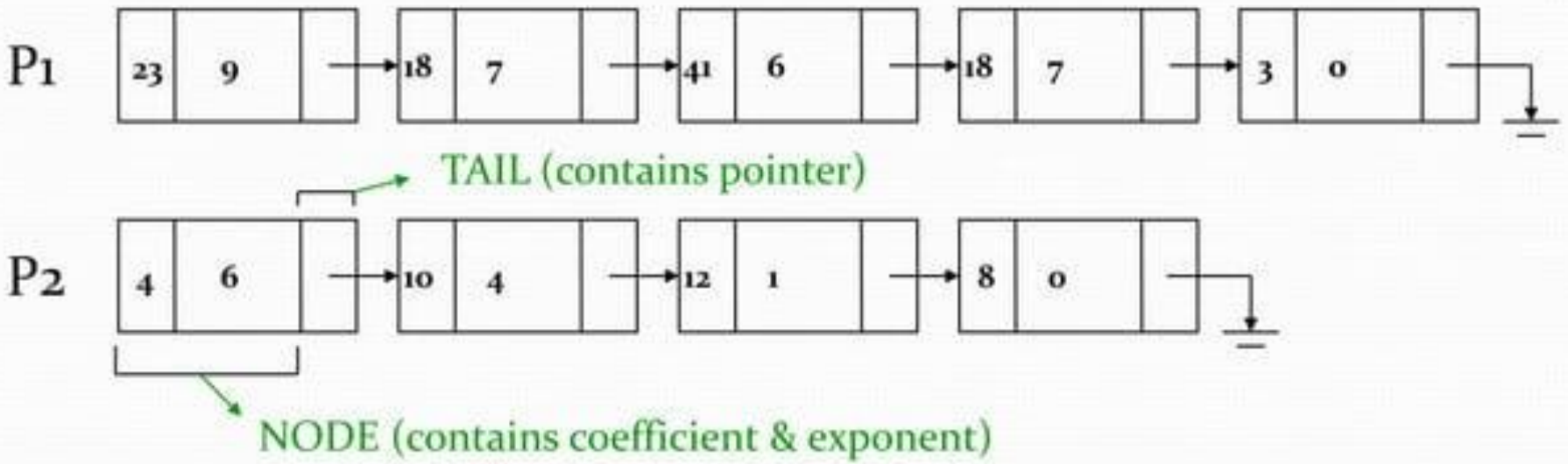
4. A stack, hash table, and binary tree can be implemented using a doubly linked list.

Partho Sarathi Sarker
Asst Professor: Dept. Of CSE

Polynomial Representation

- Linked list Implementation:

- $p1(x) = 23x^9 + 18x^7 + 41x^6 + 163x^4 + 3$
- $p2(x) = 4x^6 + 10x^4 + 12x + 8$



- Advantages of using a Linked list:
  - save space (don't have to worry about sparse polynomials) and easy to maintain
  - don't need to allocate list size and can declare nodes (terms) only as needed
- Disadvantages of using a Linked list :
  - can't go backwards through the list
  - can't jump to the beginning of the list from the end.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 12
TREES: TYPE, PROPERTIES
UNDERSTAND AND IMPLEMENT TREE STRUCTURES AND THEIR TRAVERSAL

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

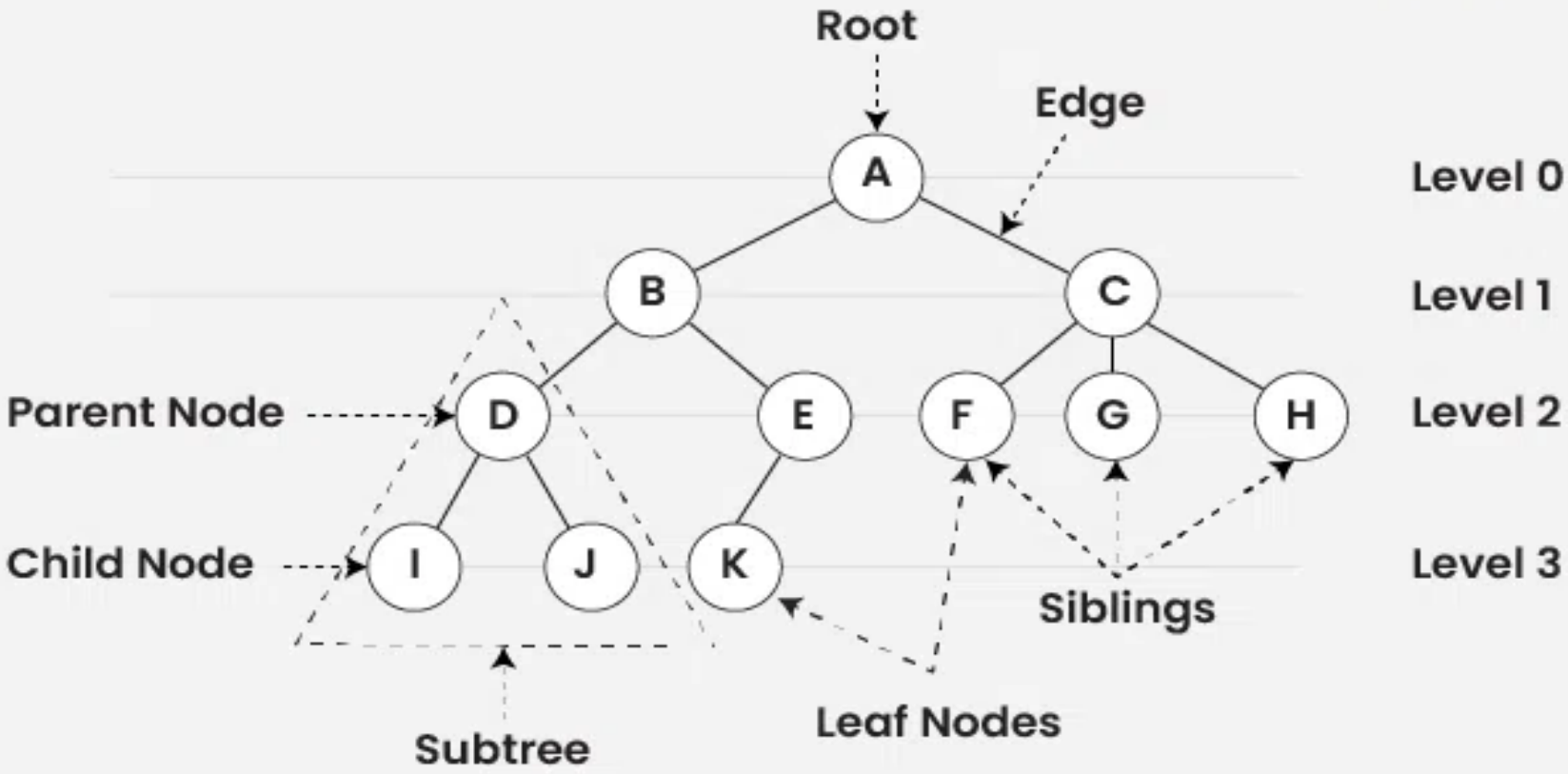
TREE DATA STRUCTURE

- **Tree data structure** is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.
- The topmost node of the tree is called the **root**, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WHY TREE IS CONSIDERED A NON-LINEAR DATA STRUCTURE?

- The data in a tree are not stored in a sequential manner i.e., they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure.


Root
Edge
A
B
C
D
E
F
G
H
I
J
K
Level 0
Level 1
Level 2
Level 3
Parent Node
Child Node
Siblings
Subtree
Leaf Nodes

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

BASIC TERMINOLOGIES IN TREE DATA STRUCTURE

- **Parent Node:** The node which is an immediate predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

BASIC TERMINOLOGIES IN TREE DATA STRUCTURE

- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{I, J, K, F, G, H}** are the leaf nodes of the tree.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

BASIC TERMINOLOGIES IN TREE DATA STRUCTURE

- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**
- **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of x.
- **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.

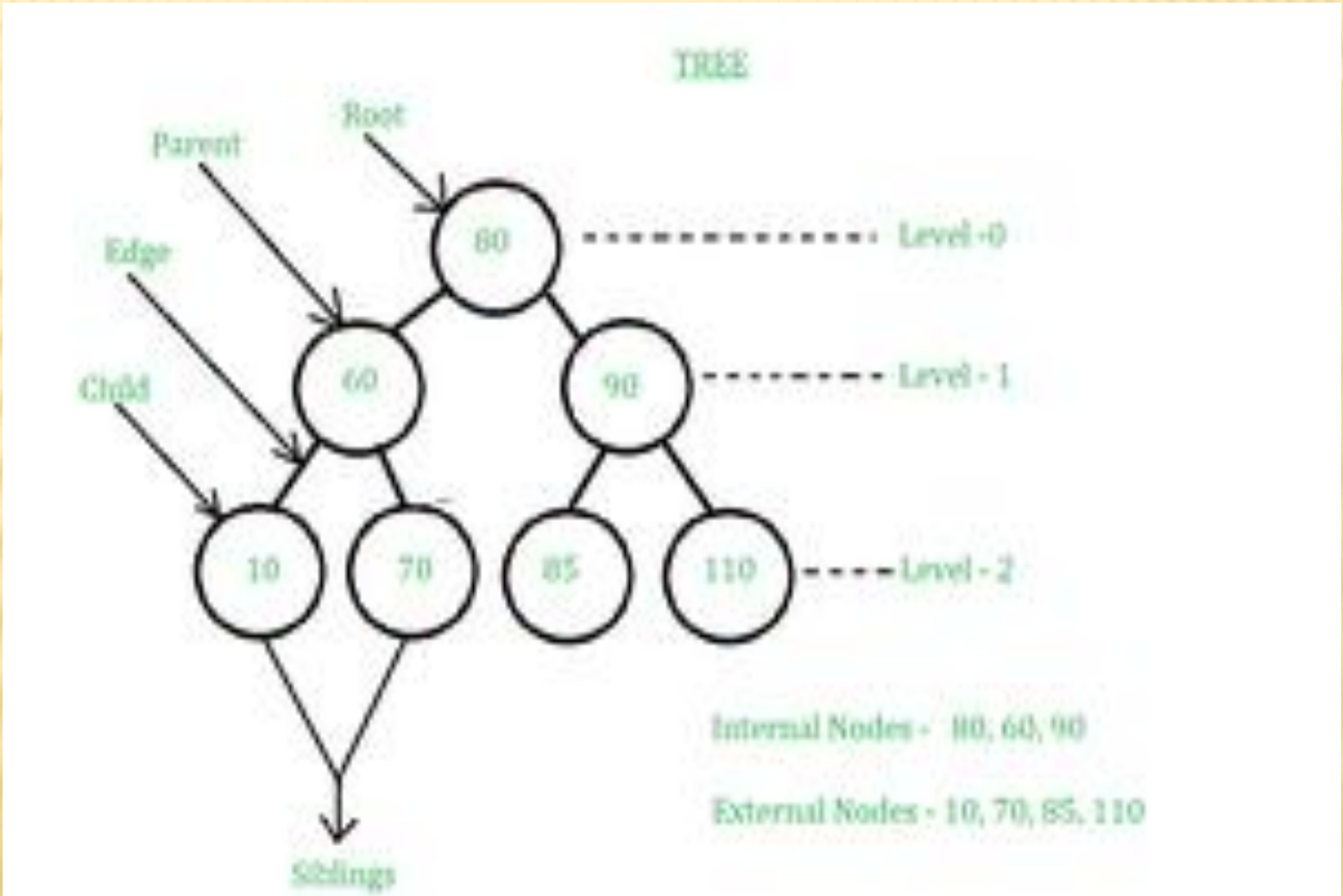
Partho Sarathi SArker
Asst Professor: Dept. Of CSE

BASIC TERMINOLOGIES IN TREE DATA STRUCTURE

- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

BASIC TERMINOLOGIES IN TREE DATA STRUCTURE



TYPES OF TREE DATA STRUCTURES:

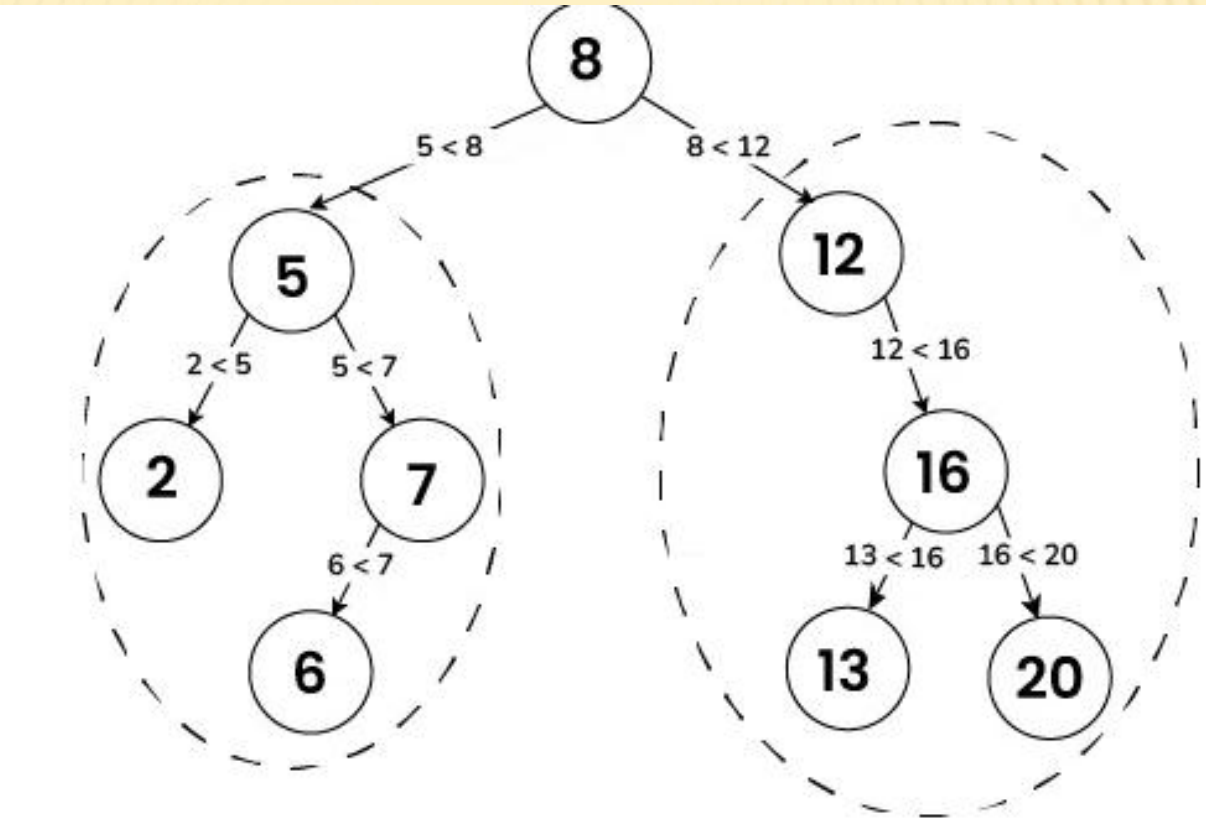
- **Binary tree**: In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees. Examples of Binary Tree are Binary Search Tree and Binary Heap.

- **Ternary Tree**: A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.
- **N-ary Tree or Generic Tree**: Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

- **Binary Search Tree** is a data structure used in computer science for organizing and storing data in a sorted manner. Binary search tree follows all properties of binary tree and for every nodes, its **left** subtree contains values less than the node and the **right** subtree contains values greater than the node. This hierarchical structure allows for efficient **Searching**, **Insertion**, and **Deletion** operations on the data stored in the tree.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

BINARY SEARCH TREE



Partho Sarathi SArker
Asst Professor: Dept. Of CSE

PROPERTIES OF TREE DATA STRUCTURE:

- **Number of edges:** An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have (N-1) edges. There is only one path from each node to any other node of the tree.
- **Depth of a node:** The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

PROPERTIES OF TREE DATA STRUCTURE:

- **Height of a node:** The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.
- **Height of the Tree:** The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

PROPERTIES OF TREE DATA STRUCTURE:

- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be **0**. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

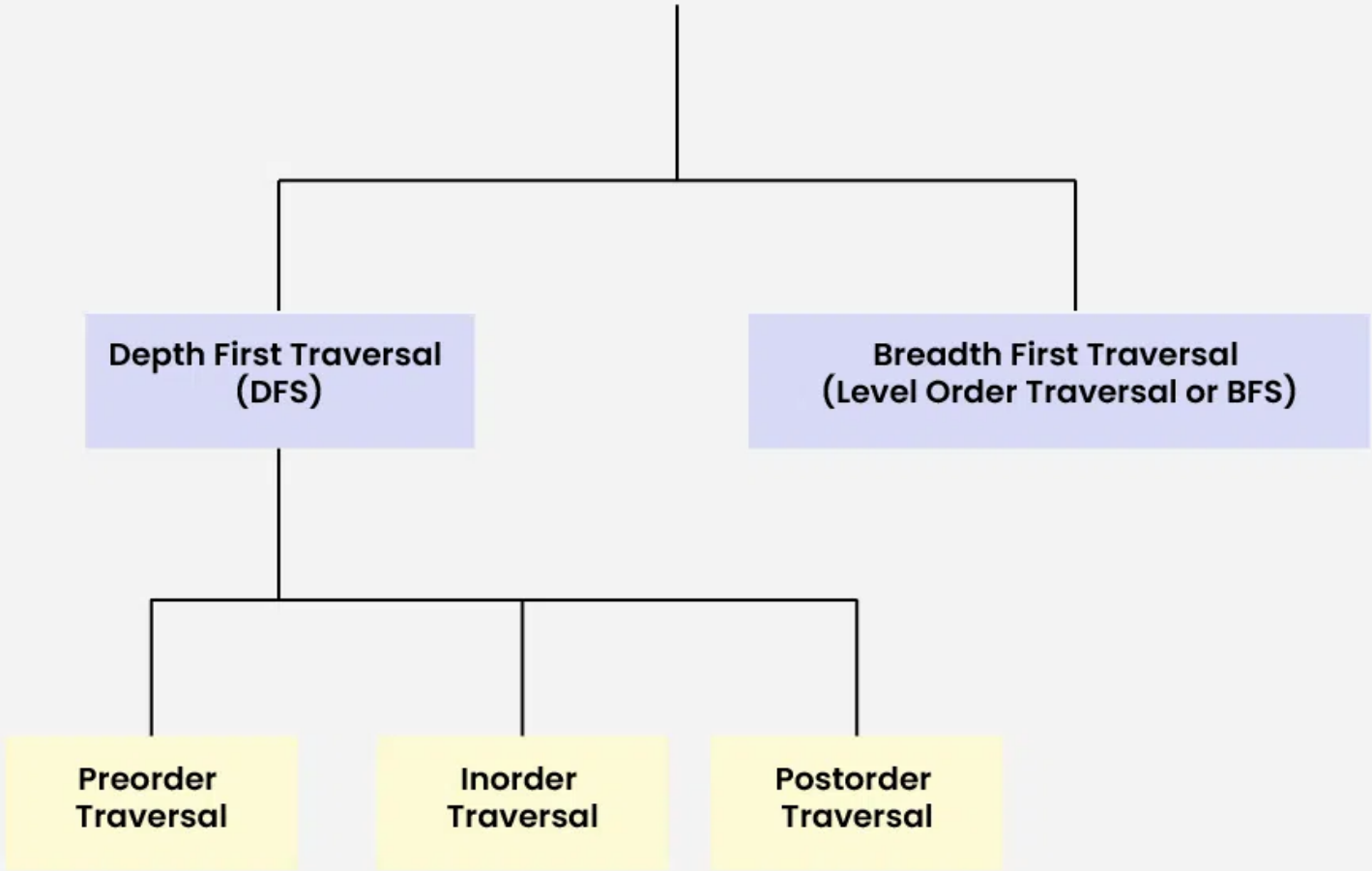
Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 13
PRE-ORDER, IN-ORDER, POST-ORDER REPRESENTATION
IMPLEMENT TREE TRAVERSAL TECHNIQUES

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

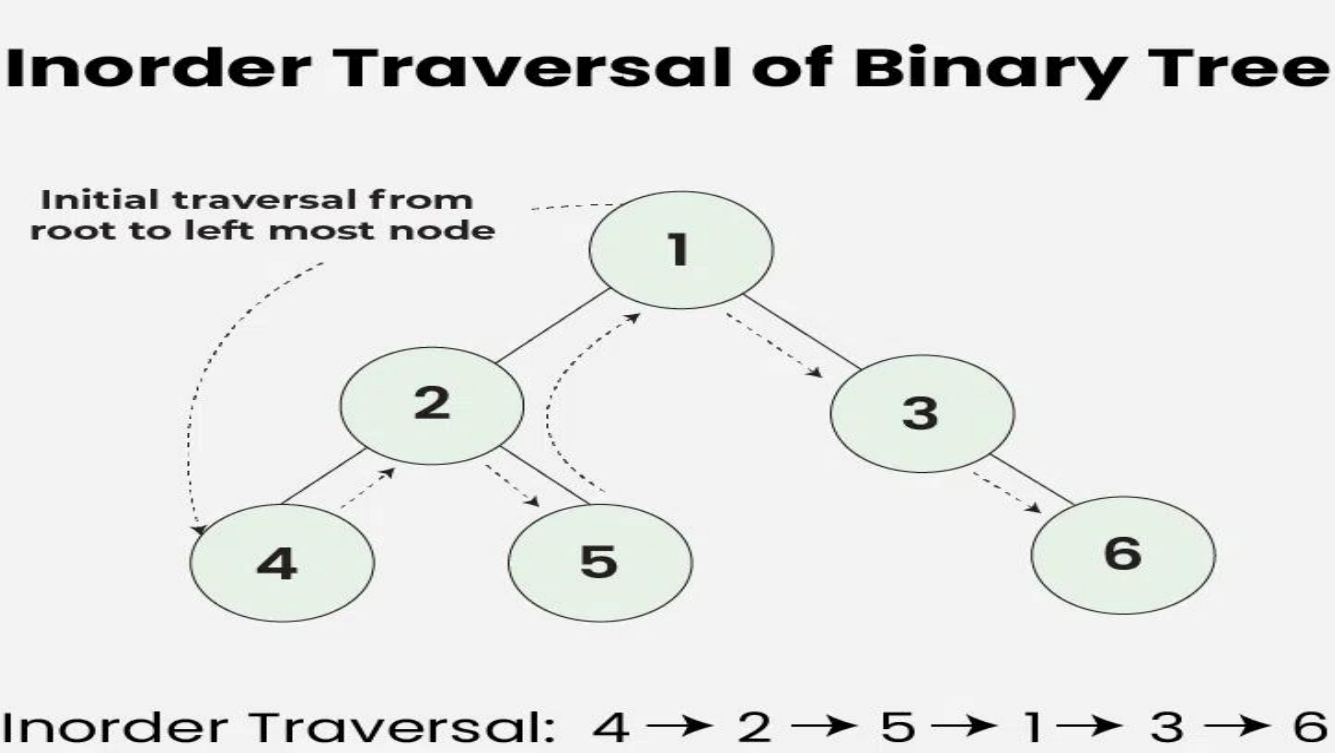
TREE TRAVERSAL TECHNIQUES

- **Tree Traversal** refers to the process of visiting or accessing each node of the tree exactly once in a certain order. Tree traversal algorithms help us to visit and process all the nodes of the tree. Since tree is not a linear data structure, there are multiple nodes which we can visit after visiting a certain node. There are multiple tree traversal techniques which decide the order in which the nodes of the tree are to be visited.


Tree Traversal Techniques
Depth First Traversal (DFS)
Breadth First Traversal (Level Order Traversal or BFS)
Preorder Traversal
Inorder Traversal
Postorder Traversal

INORDER TRAVERSAL

- Inorder traversal visits the node in the order: **Left -> Root -> Right**



ALGORITHM FOR INORDER TRAVERSAL:

- *Inorder(tree)*

Traverse the left subtree, i.e., call Inorder(left->subtree)

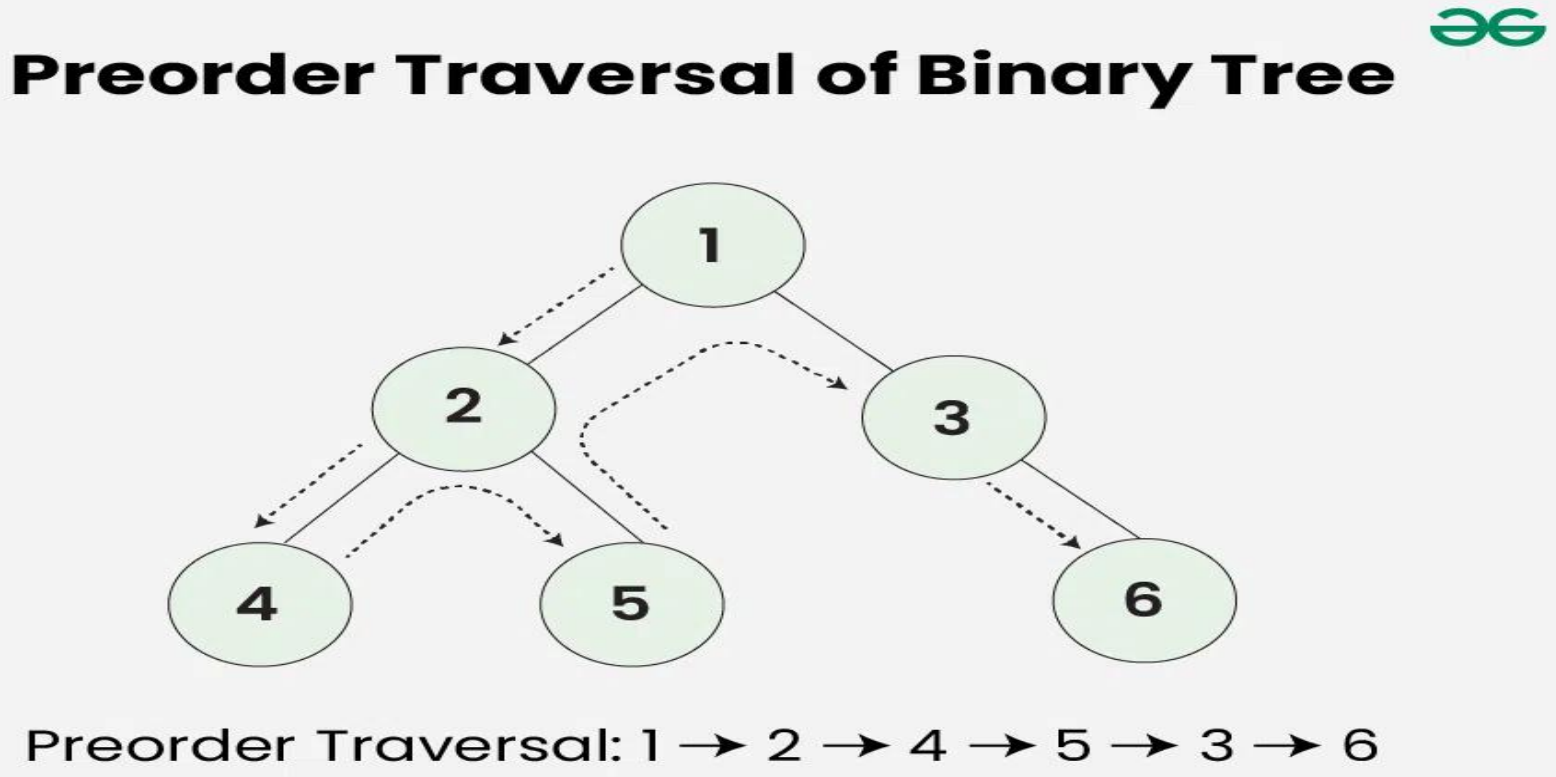
Visit the root.

Traverse the right subtree, i.e., call Inorder(right->subtree)

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

PREORDER TRAVERSAL

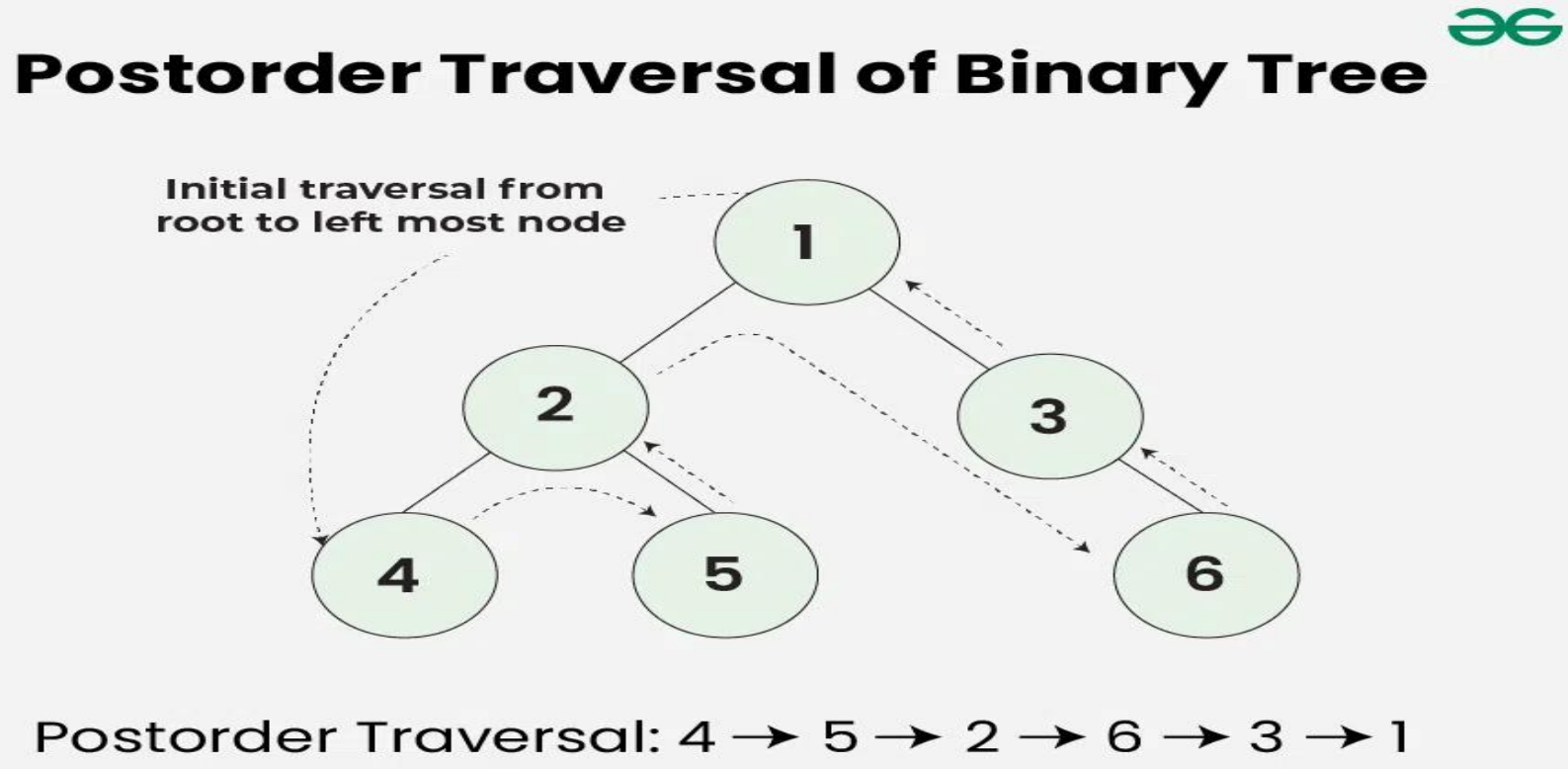
- Preorder traversal visits the node in the order: **Root -> Left -> Right**



Partho Sarathi SArker
Asst Professor: Dept. Of CSE

POSTORDER TRAVERSAL

- Postorder traversal visits the node in the order: **Left -> Right -> Root**



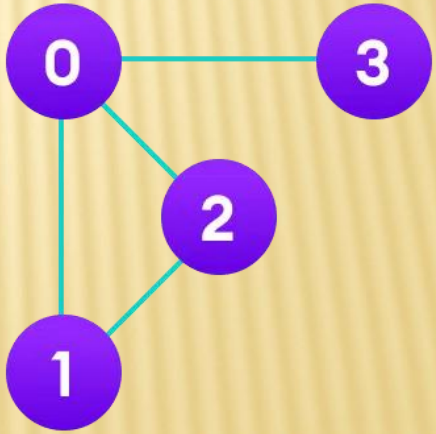
Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 14
GRAPHS: REPRESENTATION AND ALGORITHMS
UNDERSTAND AND IMPLEMENT GRAPH REPRESENTATIONS AND ALGORITHMS

- A graph data structure is a collection of nodes that have data and are connected to other nodes.
- More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)

- V = {0, 1, 2, 3}
- E = {(0,1), (0,2), (0,3), (1,2)}
- G = {V, E}



GRAPH TERMINOLOGY

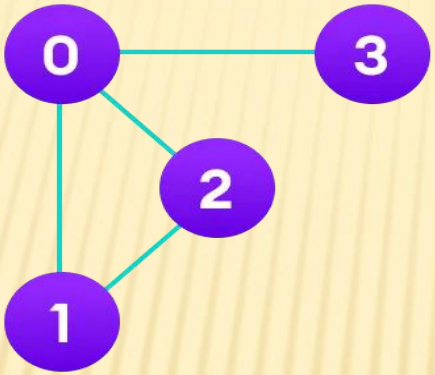
- **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

GRAPH REPRESENTATION

- Adjacency Matrix
- An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.
- If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

- The adjacency matrix for the graph we created above is…

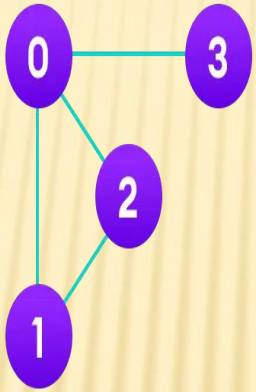


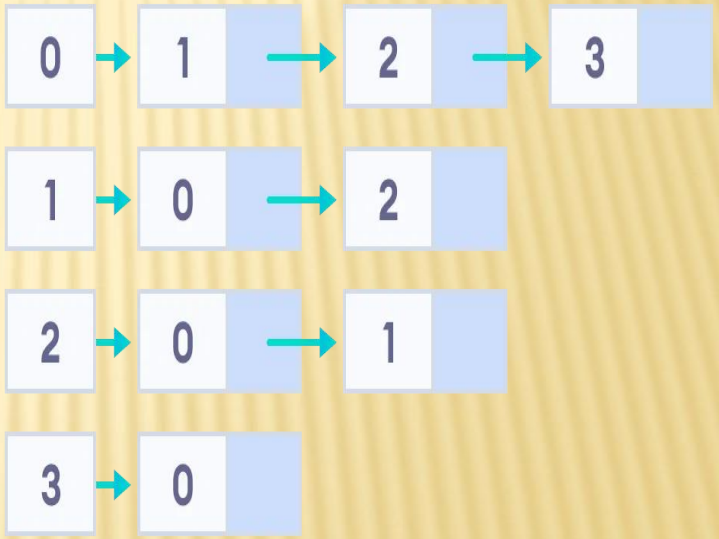
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

- Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

- **Adjacency List**
- An adjacency list represents a graph as an array of linked lists.
- The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

- The adjacency list for the graph we made in the first example is as follows:





- An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

TYPES OF GRAPH DATA STRUCTURE

- Finite Graph
- Infinite Graph
- Trivial Graph
- Simple Graph
- Multi Graph
- Null Graph
- Complete Graph
- Pseudo Graph
- Regular Graph
- Bipartite Graph
- Labelled Graph
- Digraph Graph
- Subgraph
- Connected or Disconnected Graph
- Cyclic Graph
- Vertex Labeled Graph
- Directed Acyclic Graph

- ref: https://www.educba.com/types-of-graph-in-data-structure/

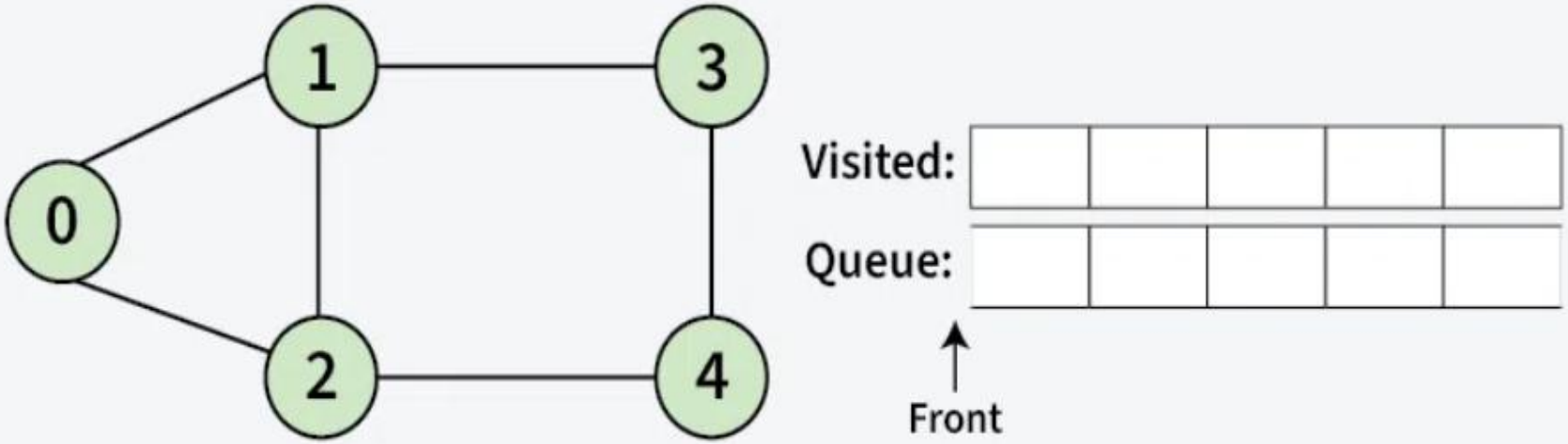
Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 15
BREADTH FIRST SEARCH ALGORITHM(BFS), APPLICATION

BREADTH FIRST SEARCH

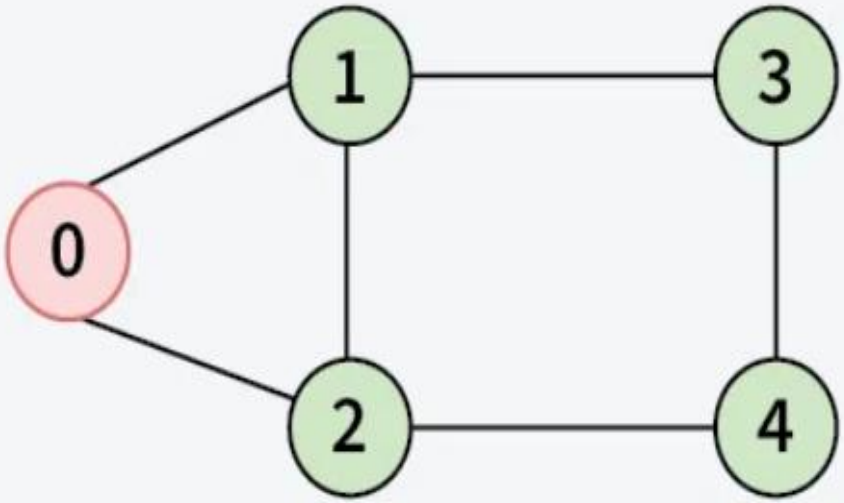
- **Breadth First Search (BFS)** is a fundamental **graph traversal algorithm.** It begins with a node, then first traverses all its adjacent. Once all adjacent are visited, then their adjacent are traversed.

01 Step | Initially queue and visited array are empty.



BFS on Graph

02 Step | Push 0 into queue and mark it visited.



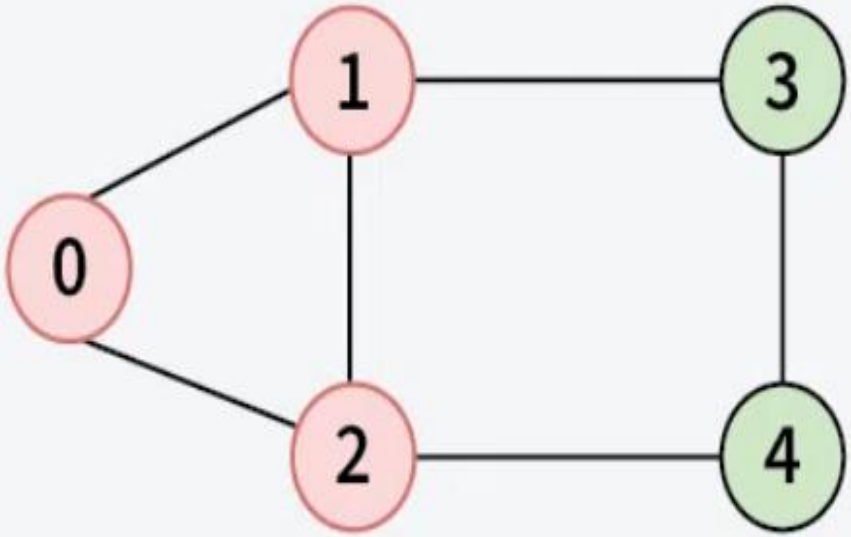
Visited: | 0 | | | | |

Queue: | 0 | | | | |

Front

BFS on Graph

03 Step | Remove 0 from the front of queue and visit the unvisited neighbours and push them into queue.



Visited:

0	1	2		

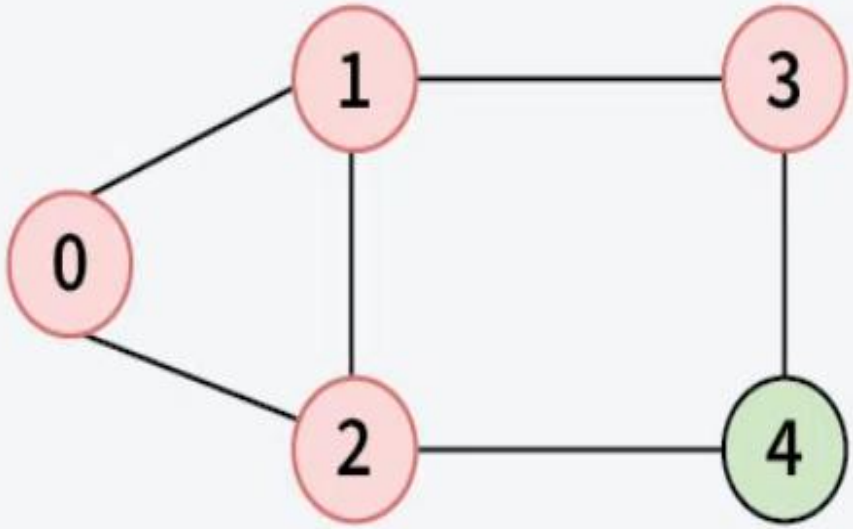
Queue:

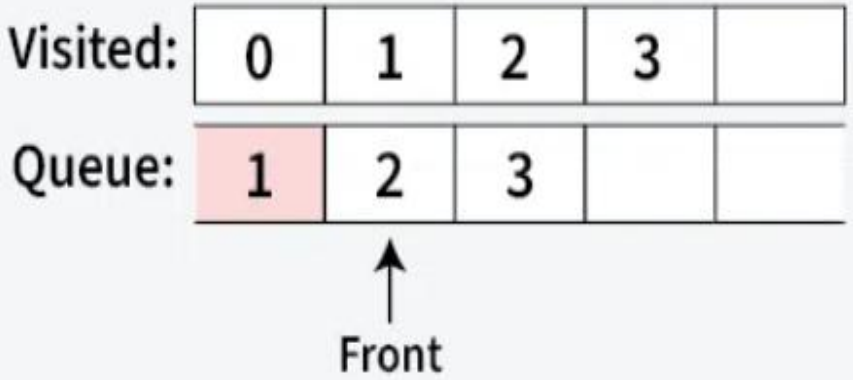
0	1	2		

Front

BFS on Graph

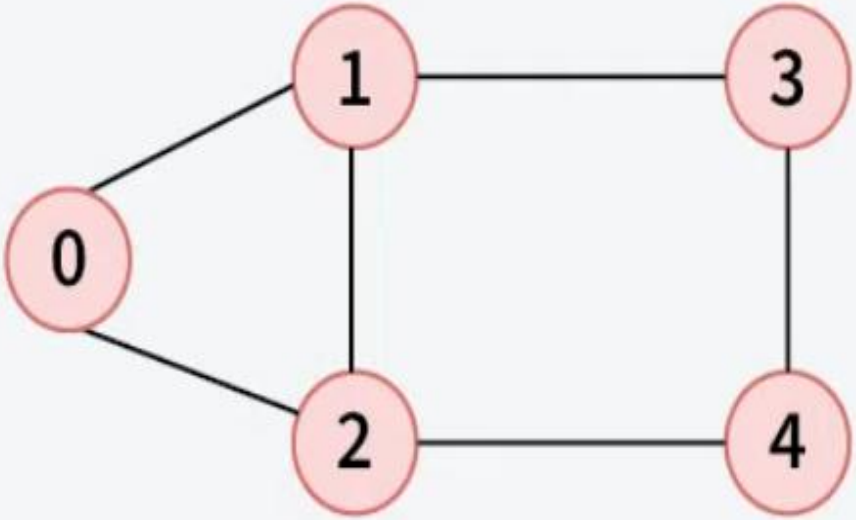
04 Step | Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.





BFS on Graph

05 Step | Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



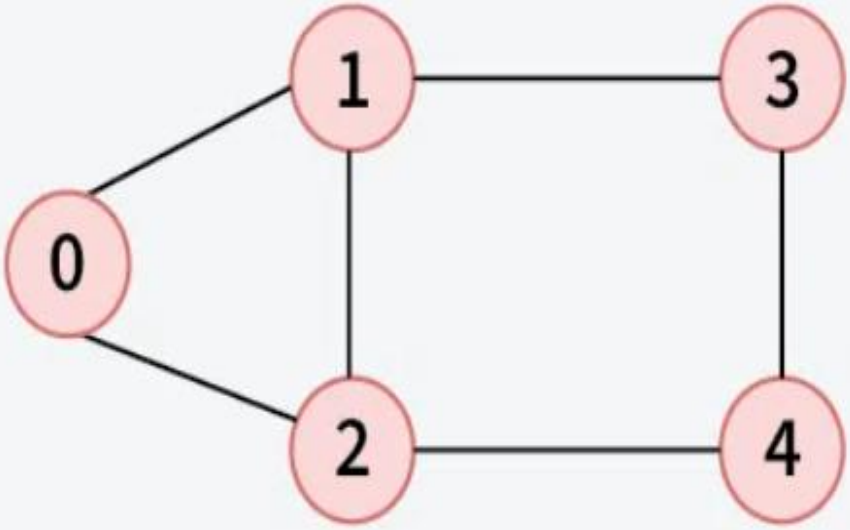
Visited:	0	1	2	3	4
Queue:	2	3	4		

Front

BFS on Graph

06 Step

Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.



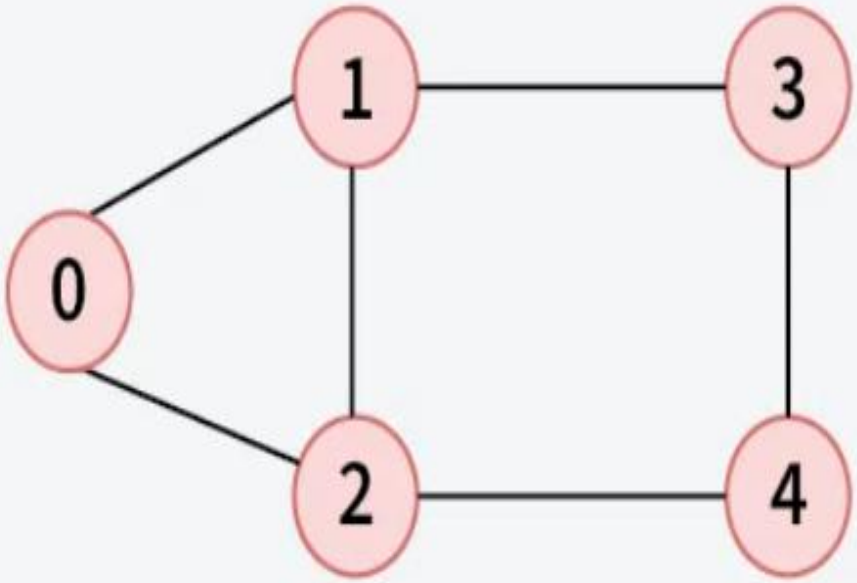
Visited:	0	1	2	3	4
Queue:	3	4			

Front

All neighbors of node 3 have been visited, proceed to the next node in the queue.

BFS on Graph

07 Step | Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.



Visited:

0	1	2	3	4

Queue:

4				

Front

All neighbors of node 4 have been visited, proceed to the next node in the queue.

BFS on Graph

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

APPLICATIONS OF BFS IN GRAPHS

- **Shortest Path Finding:** BFS can be used to find the shortest path between two nodes in an unweighted graph. By keeping track of the parent of each node during the traversal, the shortest path can be reconstructed.
- **Cycle Detection:** BFS can be used to detect cycles in a graph. If a node is visited twice during the traversal, it indicates the presence of a cycle.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

APPLICATIONS OF BFS IN GRAPHS

- **Connected Components:** BFS can be used to identify connected components in a graph. Each connected component is a set of nodes that can be reached from each other.
- **Topological Sorting:** BFS can be used to perform topological sorting on a directed acyclic graph (DAG). Topological sorting arranges the nodes in a linear order such that for any edge (u, v), u appears before v in the order.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

APPLICATIONS OF BFS IN GRAPHS

- **Level Order Traversal of Binary Trees:** BFS can be used to perform a level order traversal of a binary tree. This traversal visits all nodes at the same level before moving to the next level.
- **Network Routing:** BFS can be used to find the shortest path between two nodes in a network, making it useful for routing data packets in network protocols.

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

WEEK 16
DEPTH FIRST SEARCH ALGORITHM(DFS), APPLICATION

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

DEPTH FIRST SEARCH ALGORITHM

- A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

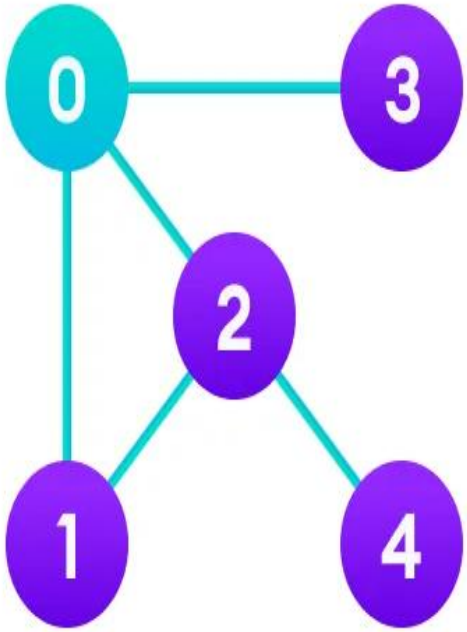
Partho Sarathi SArker
Asst Professor: Dept. Of CSE

DEPTH FIRST SEARCH ALGORITHM

- The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

DEPTH FIRST SEARCH ALGORITHM PROCEDURE

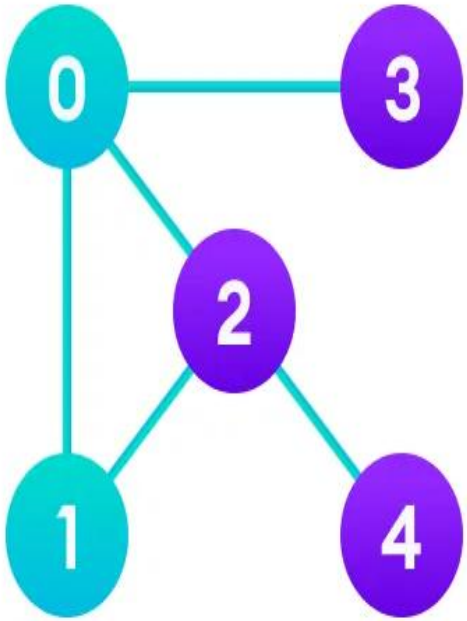


0					Visited

3	2	1			Stack

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

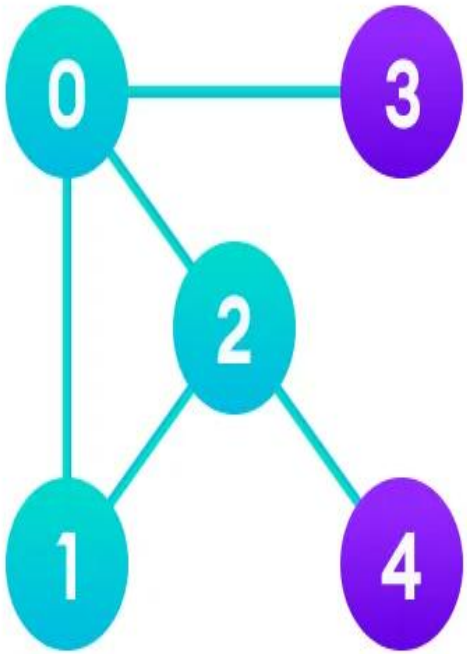
DEPTH FIRST SEARCH ALGORITHM PROCEDURE



0	1				Visited

3	2				Stack

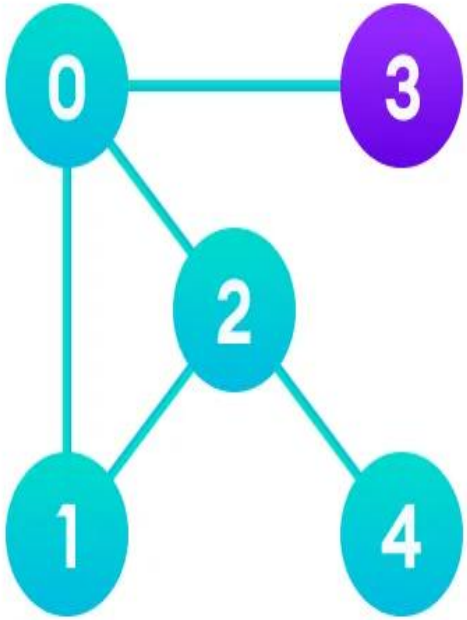
DEPTH FIRST SEARCH ALGORITHM PROCEDURE



0	1	2			Visited

3	4				Stack

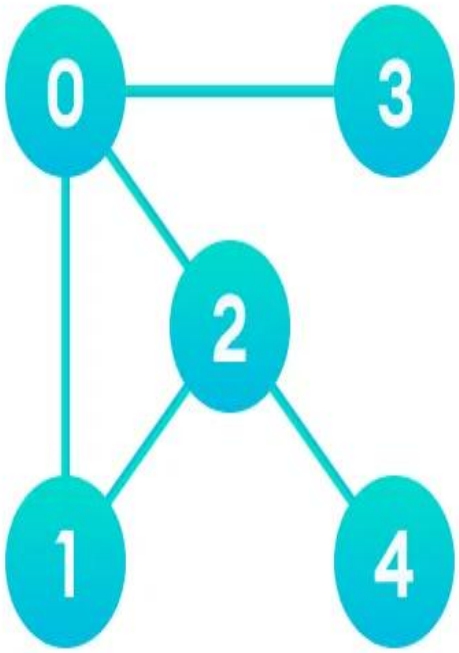
DEPTH FIRST SEARCH ALGORITHM PROCEDURE



0	1	2	4		Visited

3					Stack

DEPTH FIRST SEARCH ALGORITHM PROCEDURE



0	1	2	4	3	Visited

					Stack

Partho Sarathi SArker
Asst Professor: Dept. Of CSE

APPLICATION OF DFS ALGORITHM

1. For finding the path
2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph

THANK YOU