Data Structure and Algorithm

CSE 2101

Md Rifat Hossain

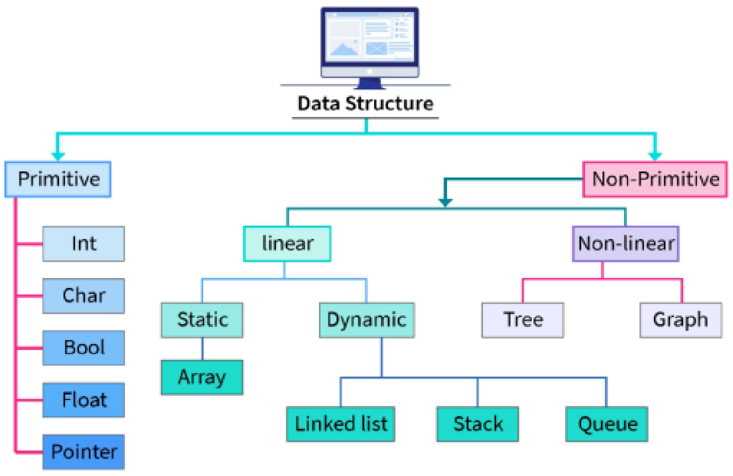
ID:230140

Dept of CSE, PUST

❖ Define Data Structures and Algorithms. What are the purpose of using data structures in problem solving? Describe with Proper Example. [CSE-12,13,14]

Ans:

- **Data Structure**: A data structure is a way of organizing and storing data so that it can be accessed and modified efficiently. Examples include arrays, linked lists, stacks, queues, trees, and graphs.
- **Algorithm:** An algorithm is a step-by-step procedure or set of rules designed to perform a specific task or solve a problem. Examples include sorting algorithms (like quicksort, mergesort) and searching algorithms (like binary search).



•

Purpose of Data Structures in Problem Solving:

Data structures play a crucial role in problem-solving because they help in:

- **Efficient Data Management**: They allow for the efficient storage, retrieval, and manipulation of data. For example, hash tables provide fast lookup times.
- **Optimized Performance**: Choosing the right data structure can significantly improve the performance of an algorithm.
- **Memory Efficiency**: Some data structures, like linked lists, optimize memory usage by dynamically allocating memory instead of using fixed-size storage.
- **Reducing Time Complexity**: Efficient data structures minimize the time required for operations like searching, sorting, and inserting elements.
- **Scalability**: Well-structured data enables programs to handle large amounts of data efficiently, making them scalable.

Example: Shortest Path Problem: Find the shortest path between two cities on a map.

1. **Data Structure**:
   - Use a **graph** to represent the cities (nodes) and roads (edges).
   - Each edge can have a weight representing the distance between two cities.

2. **Algorithm**:
    - Use **Kruskal's algorithm** to compute the shortest path between two cities.

- The **graph** data structure organizes the data (cities and roads) in a way that makes it easy to apply Kruskal's algorithm.
- Without the graph, the algorithm would lack the necessary structure to efficiently process the data.

❖ What is linear and Non-linear Data Structures? Can Non-Linear data structures can be transformed into the structured ones?[CSE-13]

Ans:

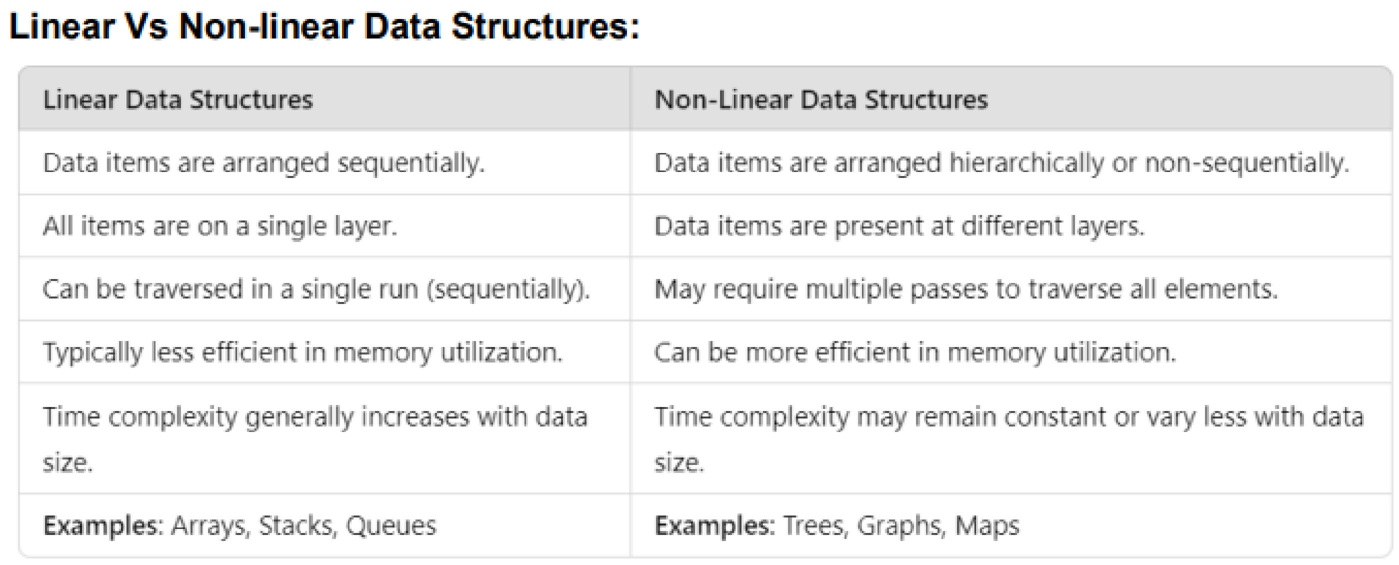
❖ A **linear data structure** is a data structure where elements are arranged sequentially or linearly. Each element has a unique predecessor and successor, except for the first and last elements.

❖ A **non-linear data structure** is a data structure where elements are not stored sequentially. Instead, they are arranged in a hierarchical or interconnected manner.

❖ Yes, **non-linear data structures** can be transformed into structured (linear) ones using appropriate techniques. The choice of transformation depends on the problem's requirements. Some examples include:

- **Trees → Arrays (via traversal)**
- **Graphs → Adjacency Matrices or Lists**
- **Hash Tables → Sorted Lists**

Difference between linear and Non-Linear Data Structures:



Linear Vs Non-linear Data Structures:

Linear Data Structures	Non-Linear Data Structures
Data items are arranged sequentially.	Data items are arranged hierarchically or non-sequentially.
All items are on a single layer.	Data items are present at different layers.
Can be traversed in a single run (sequentially).	May require multiple passes to traverse all elements.
Typically less efficient in memory utilization.	Can be more efficient in memory utilization.
Time complexity generally increases with data size.	Time complexity may remain constant or vary less with data size.
Examples: Arrays, Stacks, Queues	**Examples:** Trees, Graphs, Maps

Control Structure: [CSE-12]

A **control structure** is a block of programming that determines the flow of execution of a program based on certain conditions or loops. It allows a program to make decisions, repeat tasks, and execute specific blocks of code conditionally. Control structures are fundamental to writing dynamic and flexible programs.

Types of Control Structures:

1.Sequential Control Structure:

- The default mode of execution where statements are executed one after another in the order they appear.

2.Selection (Decision) Control Structure:

Allows the program to choose between different paths of execution based on a condition

- **if statement**: Executes a block of code if a condition is true.
- **if-else statement**: Executes one block of code if a condition is true, and another block if it is false.
- **switch statement**: Executes one of many blocks of code based on the value of a variable or expression.

3.Repetition (Loop) Control Structure:

- Allows the program to repeat a block of code multiple times based on a condition.

- **for loop:** Repeats a block of code a specific number of times.
- **while loop:** Repeats a block of code as long as a condition is true.
- **do-while loop:** Repeats a block of code at least once, then continues as long as a condition is true.

Algorithm:

An **algorithm** is a **step-by-step procedure or a set of well-defined instructions** designed to perform a specific task or solve a particular problem. It takes some input, processes it, and produces an output.

How to Write an Algorithm:

Step 1: Identify the Problem

Step 2: Determine Inputs and Outputs

Step 3: Define the Steps Clearly

Break the solution into a sequence of logical steps.

Ensure each step is precise and unambiguous.

Step 4: Check and Optimize

Verify the correctness of the steps.

Optimize for efficiency (time and space complexity).

❖ **Example of an Algorithm:**

Algorithm to Find the Sum of Two Numbers

Step 1: Start

Step 2: Input two numbers (A, B)

Step 3: Compute sum = A + B

Step 4: Display the sum

Step 5: Stop

Complexity of Algorithms: [CSE-13]

The **complexity of an algorithm** refers to the amount of resources (time and space) required to execute it as a function of the input size (n). It helps us analyze and compare the efficiency of algorithms. There are two main types of complexity:

1. **Time Complexity**:
   - Measures the amount of time an algorithm takes to complete as a function of the input size.
   - Example: $O(n)$, $O(n^2)$, $O(\log n)$.
2. **Space Complexity**:
   - Measures the amount of memory an algorithm uses as a function of the input size.
   - Example: $O(1)$, $O(n)$, $O(n^2)$.

Big O Notation:

Big O notation is used to describe the upper bound of an algorithm's complexity. It provides a way to compare the efficiency of algorithms in terms of how they scale with input size.

Comparing $O(n)$ and $O(n^2)$ Time Complexity

$O(n)$ Time Complexity:

- The algorithm's runtime grows **linearly** with the input size.
- Example: Iterating through an array of size n.

$O(n^2)$ Time Complexity:

- The algorithm's runtime grows **quadratically** with the input size.
- Example: Nested loops where each loop iterates n times.

Which Algorithm Performs Better?

- **$O(n)$** performs better than **$O(n^2)$** for large input sizes because it scales more efficiently.
- As n increases, the runtime of an $O(n^2)$ algorithm grows much faster than that of an $O(n)$ algorithm.

Example: Linear Search vs. Bubble Sort

1. Linear Search (O(n))

- **Algorithm**: Searches for an element in an array by checking each element one by one.
- **Time Complexity**: O(n).
- **Explanation**:
  - In the worst case, it checks all n elements.
  - Runtime increases linearly with input size.

Code:

```
int linearSearch(int arr[], int n, int target) {
  for (int i = 0; i < n; i++) {
    if (arr[i] == target) {
      return i; // Return index if found
    }
  }
  return -1; // Return -1 if not found
}
```

2. Bubble Sort ($O(n^2)$)

- **Algorithm**: Sorts an array by repeatedly swapping adjacent elements if they are in the wrong order.
- **Time Complexity**: $O(n^2)$.
- **Explanation**:
  - It uses nested loops, where the outer loop runs n times and the inner loop runs n times in the worst case.
  - Runtime increases quadratically with input size.

```
void bubbleSort(int arr[], int n) {
  for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        swap(arr[j], arr[j + 1]); // Swap if out of order }
    }
  }
}
```

Conclusion:

- **O(n)** algorithms are more efficient than **$O(n^2)$** algorithms for large input sizes.
- For example, Linear Search (O(n)) is faster than Bubble Sort ($O(n^2)$) as n grows.
- When designing algorithms, aim for lower time complexity (e.g., O(n), O(log n)) to ensure better performance, especially for large datasets.

Array:

An **array** is a collection of elements of the same data type stored in contiguous memory locations. It allows random access using an index. Arrays can be **one-dimensional**, **two-dimensional**, or **multi-dimensional** based on how data is organized.

Linear Array:[CSE-14]

A **linear array** is a **one-dimensional** array where elements are arranged sequentially in a straight line. It follows a fixed size and provides direct access to elements using indexing.

Basic Opeartions :

The data in the data structures are processed by certain operations.

Traversing: Visiting each element in a data structure to access or display its contents.

Searching: Finding a specific element within a data structure.

Insertion: Adding a new element into a data structure.

Deletion: Removing an existing element from a data structure.

Sorting: Arranging elements in a specific order (e.g., ascending or descending).

Merging: Combining elements from two or more data structures into one.

Searching: [CSE-13,14]

Linear Search:

 Definition: A simple searching algorithm that checks each element in the list **one by one** until the desired element is found or the list ends.

 Time Complexity: O(n) (Worst case: checks all elements)

C++ Code:

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

int linearSearch(int arr[], int n, int key) {
  for (int i = 0; i < n; i++) {
    if (arr[i] == key) {
      return i;  // Return the index where key is found
    }
  }
  return -1;  // Element not found
}
```

```
int main() {
  int arr[] = {10, 20, 30, 40, 50};
  int n = sizeof(arr) / sizeof(arr[0]);
  int key = 30;

  int result = linearSearch(arr, n, key);
  if (result != -1)
    cout << "Element found at index " << result << endl;
  else
    cout << "Element not found" << endl;
 return 0;
}
```

Binary Search: [CSE-13,14]

- **Definition:** A searching algorithm that works on a **sorted array** by repeatedly dividing the search range in half.
- **Time Complexity: O(log n)** (Much faster than Linear Search)

Q: explain binary search with the data [2 5 6 9 3 1 8] and write down its pseudocode.[CSE-14]

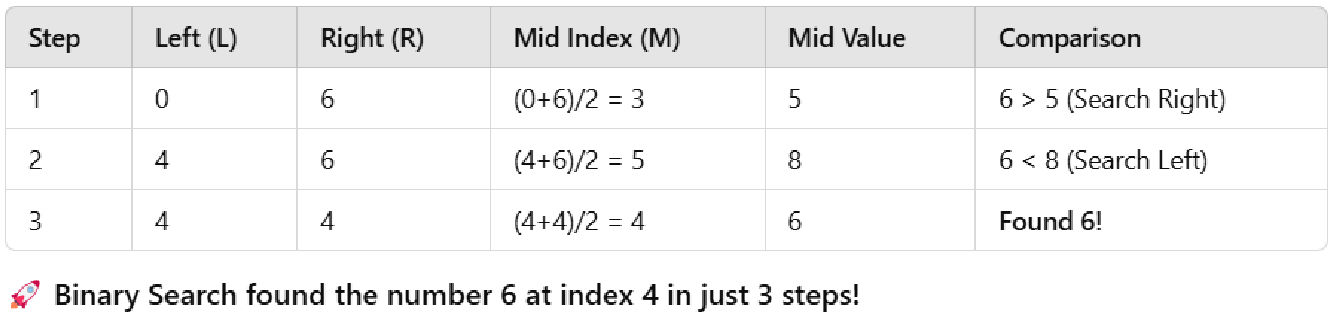
Ans:

Step 1: Sort the Array

Given data: **[2, 5, 6, 9, 3, 1, 8]**
Sorted array: **[1, 2, 3, 5, 6, 8, 9]**

Step 2: Perform Binary Search

Let's search for **key = 6** in the sorted array [1, 2, 3, 5, 6, 8, 9].



Step	Left (L)	Right (R)	Mid Index (M)	Mid Value	Comparison
1	0	6	(0+6)/2 = 3	5	6 > 5 (Search Right)
2	4	6	(4+6)/2 = 5	8	6 < 8 (Search Left)
3	4	4	(4+4)/2 = 4	6	**Found 6!**

🚀 Binary Search found the number 6 at index 4 in just 3 steps!

Pseudocode for Binary Search:

```
BinarySearch(arr, key, left, right):
1. Repeat while left ≤ right:
   2. mid = (left + right) / 2
   3. If arr[mid] == key → return mid  (Element found)
   4. If arr[mid] < key → left = mid + 1  (Search right)
   5. Else → right = mid - 1  (Search left)
   6. Return -1 (Element not found).
```

C++ Code:

```
#include <iostream>
#include <algorithm> // For sorting
using namespace std;

int binarySearch(int arr[], int left, int right, int key) {
  while (left <= right) {
    int mid = left + (right - left) / 2;

    if (arr[mid] == key) return mid;  // Element found
    if (arr[mid] < key) left = mid + 1;  // Search right half
    else right = mid - 1;  // Search left half }
return -1;  // Element not found}
```

```
int main() {
 int arr[] = {2, 5, 6, 9, 3, 1, 8};
 int n = sizeof(arr) / sizeof(arr[0]);
 int key = 6;

 sort(arr, arr + n);  // Sort the array first
 int result = binarySearch(arr, 0, n - 1, key);

 if (result != -1)
   cout << "Element found at index: " << result << endl;
 else
   cout << "Element not found" << endl;}
```

Sorting: [CSE-12,13,14]

A Sorting Algorithm is used to rearrange a given array or list of elements according to a comparison operator on the elements.

Types of Sorting Techniques in Data Structure :

Several sorting techniques in data structure can be used to sort data elements in an array or list. The most common types of sorting in data structure are:-

Bubble Sort: Repeatedly swaps adjacent elements if they are in the wrong order.

Selection Sort: Selects the smallest (or largest) element from the unsorted portion and places it in the correct position.

Insertion Sort: Builds the sorted array one element at a time by inserting each new element into its proper position.

Merge Sort: Divides the array into halves, recursively sorts each half, and then merges the sorted halves.

Quick Sort: Chooses a pivot element, partitions the array around the pivot, and recursively sorts the partitions.

1. Bubble Sort:

Initial Array: [5, 1, 4, 2, 8]

Steps:

1. Pass 1:

○ [1, 5, 4, 2, 8] (5 and 1 swapped)

○ [1, 4, 5, 2, 8] (5 and 4 swapped)

○ [1, 4, 2, 5, 8] (5 and 2 swapped)

○ [1, 4, 2, 5, 8] (8 is in the correct position)

2. Pass 2:

○ [1, 4, 2, 5, 8] (no swap needed between 1 and 4)

○ [1, 2, 4, 5, 8] (4 and 2 swapped)

3. Pass 3:

○ [1, 2, 4, 5, 8] (no swap needed)

4. Pass 4:

○ [1, 2, 4, 5, 8] (no swap needed)

Sorted Array: [1, 2, 4, 5, 8]

C++ Code:

```
 #include <iostream>
 using namespace std;
// Function to perform Bubble Sort
 void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
       // Last i elements are already sorted
       for (int j = 0; j < n - i - 1; j++) {
          if (arr[j] > arr[j + 1]) {
             swap(arr[j], arr[j + 1]); }
       }
    }
 }
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
       cout << arr[i] << " "; }
    cout << endl;}
```

```
// Main function
 int main() {
int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Original array: ";
    printArray(arr, n);

    bubbleSort(arr, n);

    cout << "Sorted array: ";
    printArray(arr, n);
  return 0;
}
```

2. Selection Sort:

Initial Array: [5, 1, 4, 2, 8]

Steps:

1. Pass 1:

○ Find minimum in [5, 1, 4, 2, 8]: Minimum is 1

○ Swap 5 with 1: [1, 5, 4, 2, 8]

2. Pass 2:

○ Find minimum in [5, 4, 2, 8]: Minimum is 2

○ Swap 5 with 2: [1, 2, 4, 5, 8]

3. Pass 3:

○ Find minimum in [4, 5, 8]: Minimum is 4

○ No swap needed: [1, 2, 4, 5, 8]

4. Pass 4:

○ Find minimum in [5, 8]: Minimum is 5

○ No swap needed: [1, 2, 4, 5, 8]

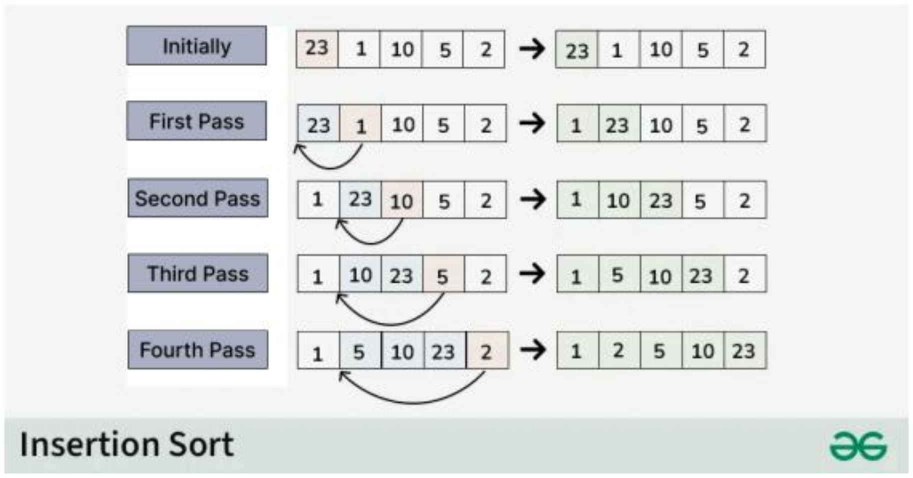
Sorted Array: [1, 2, 4, 5, 8]

C++ Code:

```
//C++ Code:
#include <iostream>
using namespace std;
void selectionSort(int arr[], int n) {
for (int i = 0; i < n - 1; i++) {
// Find the minimum element in the unsorted part
int minIndex = i;
for (int j = i + 1; j < n; j++) {
if (arr[j] < arr[minIndex]) {
minIndex = j;}
}
// Swap the found minimum element with the first
unsorted element
swap(arr[i], arr[minIndex]);}
}
```

```
int main() {
   int arr[] = {5, 1, 4, 2, 8};
   int n = sizeof(arr) / sizeof(arr[0]);
 cout << "Initial Array: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";}
   cout << endl;
  selectionSort(arr, n);
 cout << "Sorted Array: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }cout << endl;
return 0; }
```

3. Insertion Sort:



C++ Code:

```
#include <iostream>
using namespace std;
void insertionSort(int arr[], int n) {
   for (int i = 1; i < n; i++) {
      int key = arr[i]; // Current element to be inserted
      int j = i - 1;
// Move elements of arr[0..i-1] that are greater than key
      // to one position ahead of their current position
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j--;   }
      arr[j + 1] = key; // Insert the key in the correct position}
}
```

```
int main() {
   int arr[] = {5, 1, 4, 2, 8};
   int n = sizeof(arr) / sizeof(arr[0]);

   cout << "Initial Array: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " "; }
   cout << endl;
insertionSort(arr, n);
  cout << "Sorted Array: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " "; }
   cout << endl;
return 0;}
```

Note: Array insert ,delete,searching try yourself.

Stack: **A stack is a linear data structure that follows the Last In, First Out (LIFO) principle, meaning the last element added is the first one removed.**

Operations:

1. Push: Add an element to the top of the stack.

○ Example: Push 10 onto [1, 2, 3], resulting in [1, 2, 3, 10].

2. Pop: Remove and return the top element from the stack.

○ Example: Pop from [1, 2, 3, 10], resulting in the stack [1, 2, 3] and the popped element 10.

3. Peek: View the top element without removing it.

○ Example: Peek on [1, 2, 3, 10], resulting in 10

4.isEmpty: Checks if the stack is empty.

Applications:

1. Function Call Management: Keeps track of function calls and returns.

2. Expression Evaluation: Used in parsing and evaluating expressions.

3. Backtracking Algorithms: Helps in scenarios like maze solving.

C++ Code:

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
  stack<int> s;
 // Push elements onto the stack
  s.push(10);
  s.push(20);
  s.push(30);
// Display the top element
  cout << "Top element: " << s.top() << endl;
// Pop the top element
  s.pop();
  cout << "Top element after pop: " << s.top() << endl;
// Push another element
  s.push(40);
  cout << "Top element after push: " << s.top() << endl;
if (s.empty()) {
    cout << "Stack is empty." << endl;
  } else {
    cout << "Stack is not empty." << endl;
  }
// Display all elements in the stack
  cout << "Elements in the stack: ";
  while (!s.empty()) {
    cout << s.top() << " ";
    s.pop(); }
  cout << endl;
 // Check if the stack is empty after popping all elements
  if (s.empty()) {
    cout << "Stack is empty." << endl;
  } else {
    cout << "Stack is not empty." << endl;}
return 0;}
```

Queue:

A queue is a data structure that follows the First In First Out (FIFO) principle. Elements are added at the back (rear) and removed from the front (front) of the queue.

Operations:

1. Enqueue: Adds an element to the back of the queue.

○ **Example:** Enqueue 10 to [1, 2, 3], resulting in [1, 2, 3, 10].

2. Dequeue: Removes and returns the front element from the queue.

○ **Example:** Dequeue from [1, 2, 3, 10], resulting in the queue [2, 3, 10] and the dequeued element 1.

Applications:

Task Scheduling: Useful in operating systems for managing tasks in the correct order.

Breadth-First Search (BFS): Used in graph traversal algorithms.

Printers and Job Queues: Jobs are processed in the order they are received.

C++ Code:

```
#include <iostream>
#include <queue>
using namespace std;
int main() {
    queue<int> q;
 // Enqueue elements
    q.push(1);
    q.push(2);
    q.push(3);
// Display the front element
    cout << "Front element: " << q.front() << endl;
   q.push(10);
    cout << "Queue after enqueueing 10: ";
    queue<int> temp = q; // Copy queue for display
    while (!temp.empty()) {
        cout << temp.front() << " ";
        temp.pop();}
    cout << endl;
// Dequeue the front element
    q.pop();
    cout << "Front element after dequeue: " << q.front()
<< endl;
// Check if the queue is empty
    if (q.empty()) {
        cout << "Queue is empty." << endl;
    } else {
        cout << "Queue is not empty." << endl;}
// Display all elements in the queue
    cout << "Elements in the queue: ";
    while (!q.empty()) {
        cout << q.front() << " ";
        q.pop(); }
    cout << endl;
  // Check if the queue is empty after dequeuing all
elements
    if (q.empty()) {
        cout << "Queue is empty." << endl;
    } else {
        cout << "Queue is not empty." << endl;
}
 return 0; }
```

Lab 1: Menue based insertion, deletion of elements in any certain position of an array!

Ans:

```
#include <iostream>
using namespace std;
// Function to insert element at a specified position
void insertElement(int arr[], int &n, int element, int
position) {
   if (position < 0 || position > n) {
      cout << "Invalid position!" << endl;
      return;}
for (int i = n; i > position; i--) {
      arr[i] = arr[i - 1]; // Shift elements to the right
   }
arr[position] = element; // Insert the element
   n++; // Increase size
}
// Function to delete element at a specified position
void deleteElement(int arr[], int &n, int position) {
   if (position < 0 || position >= n) {
      cout << "Invalid position!" << endl;
      return;}
 for (int i = position; i < n - 1; i++) {
      arr[i] = arr[i + 1]; // Shift elements to the left
   }
 n--; // Decrease size}
// Function to print array
void printArray(int arr[], int n) {
   if (n == 0) {
      cout << "Array is empty." << endl;
      return;  }
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   } cout << endl;
}
```

```
int main() {
   int arr[10], n = 0;  // Array and size
   int choice, element, position;
while (true) {
      cout << "Menu:\n";
      cout << "1. Insert Element\n";
      cout << "2. Delete Element\n";
      cout << "3. Display Array\n";
      cout << "4. Exit\n";
      cout << "Enter your choice: ";
      cin >> choice;
 switch (choice) {
         case 1:
            cout << "Enter element to insert: ";
            cin >> element;
         cout << "Enter position to insert (0 to " << n << "): ";
       cin >> position;
            insertElement(arr, n, element, position); break;
         case 2:
     cout << "Enter position to delete (0 to " << n - 1 << "): ";
            cin >> position;
            deleteElement(arr, n, position);      break;
case 3:
            cout << "Array: ";
            printArray(arr, n);       break;
         case 4:
            cout << "Exiting program.\n";
            return 0;
         default:
            cout << "Invalid choice! Try again.\n"; }
   }   return 0;
}
```