Perfect 👌 — this is a **basic Bubble Sort** question — super common in lab exams.  
Let’s go step-by-step as if you’re new to DSA.

**🧱 Q: Warehouse Package Sorting Using Bubble Sort**

**Question:**

A warehouse system stores package IDs in the order they arrive.

To prepare for dispatch, the IDs must be sorted in ascending order.

Write a program using Bubble Sort to arrange the following IDs:

[5, 4, 3, 2, 1]

**🔹 Beginner Explanation**

* Bubble Sort works by **repeatedly swapping adjacent elements** if they’re in the wrong order.
* In each pass, the **largest element “bubbles up”** to the end.
* It’s one of the simplest sorting algorithms — good for small lists.

**🔹 Step-by-Step Process**

**Given array:** [5, 4, 3, 2, 1]

We’ll compare adjacent elements in each pass.

**Pass 1:**

Compare and swap wherever needed:

[5, 4, 3, 2, 1]

→ (5>4) swap → [4,5,3,2,1]

→ (5>3) swap → [4,3,5,2,1]

→ (5>2) swap → [4,3,2,5,1]

→ (5>1) swap → [4,3,2,1,5]

Largest element 5 is now at the end ✅

**Pass 2:**

Now last element (5) is fixed. Compare first 4 elements:

[4,3,2,1,5]

→ (4>3) swap → [3,4,2,1,5]

→ (4>2) swap → [3,2,4,1,5]

→ (4>1) swap → [3,2,1,4,5]

4 is now second largest, placed correctly ✅

**Pass 3:**

[3,2,1,4,5]

→ (3>2) swap → [2,3,1,4,5]

→ (3>1) swap → [2,1,3,4,5]

3 goes to correct place ✅

**Pass 4:**

[2,1,3,4,5]

→ (2>1) swap → [1,2,3,4,5]

Sorted ✅

✅ **Final Sorted Array:**

[1, 2, 3, 4, 5]

**🔹 C++ Program (Bubble Sort)**

#include <iostream>

using namespace std;

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 adjacent elements

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

int main() {

int arr[] = {5, 4, 3, 2, 1};

int n = 5;

bubbleSort(arr, n);

cout << "Sorted package IDs: ";

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

return 0;

}

**🔹 Output:**

Sorted package IDs: 1 2 3 4 5

**🔹 Complexity:**

* **Time Complexity:**
  + Best case (already sorted): O(n)
  + Worst case: O(n²)
* **Space Complexity:** O(1) (in-place sorting)

Perfect 👍 — now we’ll solve the same warehouse package sorting problem,  
but this time using **Insertion Sort**.

Let’s go step-by-step so you understand **what’s happening** and have a ready C++ program for your exam.

**🧱 Q: Warehouse Package Sorting Using Insertion Sort**

**Question:**

A warehouse system stores package IDs in the order they arrive.

To prepare for dispatch, the IDs must be sorted in ascending order.

Write a program using Insertion Sort to arrange the following IDs:

[5, 4, 3, 2, 1]

**🔹 Beginner Explanation**

* **Insertion Sort** works the same way you sort playing cards in your hand.
  + Pick one card (element) at a time.
  + Insert it in the correct position among the already sorted cards.
* It’s efficient for **small or partially sorted data**.
* Works **in-place** (no extra memory needed).

**🔹 Step-by-Step Process**

**Given array:** [5, 4, 3, 2, 1]

We sort it from **left to right** by inserting each new element into the already sorted left portion.

**Pass 1 (i = 1):**

* Key = 4
* Compare with 5 → since 5 > 4, shift 5 one position right.  
  Array → [5, 5, 3, 2, 1]  
  Insert 4 in the first position → [4, 5, 3, 2, 1]

**Pass 2 (i = 2):**

* Key = 3
* Compare with 5 (5 > 3) → shift right
* Compare with 4 (4 > 3) → shift right  
  Insert 3 at start → [3, 4, 5, 2, 1]

**Pass 3 (i = 3):**

* Key = 2
* Compare with 5 → shift right
* Compare with 4 → shift right
* Compare with 3 → shift right  
  Insert 2 at index 0 → [2, 3, 4, 5, 1]

**Pass 4 (i = 4):**

* Key = 1
* Compare with 5 → shift right
* Compare with 4 → shift right
* Compare with 3 → shift right
* Compare with 2 → shift right  
  Insert 1 at index 0 → [1, 2, 3, 4, 5]

✅ **Final Sorted Array:**

[1, 2, 3, 4, 5]

**🔹 C++ Program (Insertion Sort)**

#include <iostream>

using namespace std;

void insertionSort(int arr[], int n) {

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

// Move elements greater than key to one position ahead

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

int main() {

int arr[] = {5, 4, 3, 2, 1};

int n = 5;

insertionSort(arr, n);

cout << "Sorted package IDs: ";

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

return 0;

}

**🔹 Output:**

Sorted package IDs: 1 2 3 4 5

**🔹 Complexity:**

* **Time Complexity:**
  + Best case (already sorted): O(n)
  + Worst case (reverse order): O(n²)
* **Space Complexity:** O(1)

**📦 Q: Warehouse Package Sorting Using Selection Sort**

**Question:**

A warehouse system stores package IDs in the order they arrive.

To prepare for dispatch, the IDs must be sorted in ascending order.

Write a program using Selection Sort to arrange the following IDs:

[5, 4, 3, 2, 1]

**🔹 Beginner Explanation**

* **Selection Sort** divides the list into two parts:
  + **Sorted part** (on the left)
  + **Unsorted part** (on the right)
* It repeatedly **selects the smallest element** from the unsorted part  
  and places it at the beginning of the unsorted region.
* Works by **finding minimum → swapping → moving boundary forward.**

**🔹 Step-by-Step Dry Run**

**Initial Array:** [5, 4, 3, 2, 1]

**Pass 1:**

Find smallest element from index 0–4  
→ minimum = 1 (index 4)  
Swap with first element (index 0):  
[1, 4, 3, 2, 5]

**Pass 2:**

Find smallest element from index 1–4  
→ minimum = 2 (index 3)  
Swap with index 1:  
[1, 2, 3, 4, 5]

**Pass 3:**

Find smallest element from index 2–4  
→ minimum = 3 (already at position 2)  
No swap needed:  
[1, 2, 3, 4, 5]

**Pass 4:**

Find smallest element from index 3–4  
→ minimum = 4 (already in place)  
No swap needed.  
Array remains [1, 2, 3, 4, 5]

✅ **Final Sorted Array:**

[1, 2, 3, 4, 5]

**🔹 C++ Program (Selection Sort)**

#include <iostream>

using namespace std;

void selectionSort(int arr[], int n) {

for (int i = 0; i < n - 1; i++) {

int minIndex = i; // assume current is smallest

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[minIndex])

minIndex = j;

}

// Swap smallest element with current position

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

int main() {

int arr[] = {5, 4, 3, 2, 1};

int n = 5;

selectionSort(arr, n);

cout << "Sorted package IDs: ";

for (int i = 0; i < n; i++)

cout << arr[i] << " ";

return 0;

}

**🔹 Output:**

Sorted package IDs: 1 2 3 4 5

**🔹 Complexity:**

| **Case** | **Time Complexity** | **Space** |
| --- | --- | --- |
| Best | O(n²) | O(1) |
| Average | O(n²) | O(1) |
| Worst | O(n²) | O(1) |

* Because we **always** look through the entire list to find the minimum each time.

**🔹 Summary:**

| **Concept** | **Explanation** |
| --- | --- |
| Algorithm | **Selection Sort** |
| Steps | Find minimum, swap with first unsorted element |
| Sorted Output | [1, 2, 3, 4, 5] |
| Complexity | O(n²) |
| Code | ✅ Provided |

**🏥 Q: Hospital Management System – Linked List of Patient IDs**

**Question:**

A hospital management system stores patient IDs in a linked list to maintain their

admission order. You are given the following sequence of patient IDs:

111 → 123 → 124 → NULL

Write a program to create and display this linked list.

**🔹 Beginner Explanation**

* A **Linked List** is a collection of **nodes**.
* Each **node** contains:
  + Data (here, the patient ID)
  + A pointer to the **next node**
* The **last node** points to NULL, meaning the list ends.

**Visual Representation:**

+------+ +------+ +------+

| 111 | → | 123 | → | 124 | → NULL

+------+ +------+ +------+

**🔹 Step-by-Step Logic**

1. **Define a node structure** having:
   * An integer data
   * A pointer next to another node
2. **Create 3 nodes** manually:
   * Node1 → data = 111
   * Node2 → data = 123
   * Node3 → data = 124
3. **Link them:**
   * Node1.next = Node2
   * Node2.next = Node3
   * Node3.next = NULL
4. **Display** the list by traversing from head until NULL.

**🔹 C++ Code (Simple & Beginner Friendly)**

#include <iostream>

using namespace std;

// Define a Node structure

struct Node {

int data;

Node\* next;

};

int main() {

// Step 1: Create nodes

Node\* head = new Node();

Node\* second = new Node();

Node\* third = new Node();

// Step 2: Assign data

head->data = 111;

second->data = 123;

third->data = 124;

// Step 3: Link nodes

head->next = second;

second->next = third;

third->next = NULL;

// Step 4: Display linked list

Node\* temp = head;

cout << "Patient ID List: ";

while (temp != NULL) {

cout << temp->data << " -> ";

temp = temp->next;

}

cout << "NULL" << endl;

return 0;

}

**🔹 Output:**

Patient ID List: 111 -> 123 -> 124 -> NULL

**🔹 Key Notes (for Viva)**

| **Concept** | **Explanation** |
| --- | --- |
| What is a Linked List? | A dynamic data structure where elements (nodes) are linked using pointers. |
| Why use it? | Easy insertion/deletion without shifting elements like in arrays. |
| Drawback? | Sequential access — you can’t directly jump to an element like in an array. |
| End of list? | Marked by a NULL pointer. |

**🎓 Q: University Exam System – Binary Tree of Student Roll Numbers**

**Question:**

A university’s examination system stores student roll numbers in a binary tree

for efficient searching. Given the structure of the tree, implement and display

the binary tree.

50

/ \

30 70

/ \ /

20 40 60

**🔹 Beginner Explanation**

* A **Binary Tree** is a data structure where **each node has at most two children**:
  + **Left child**
  + **Right child**
* Each node contains:
  + Data (here: student roll number)
  + Pointers to left and right child nodes.

The given structure represents the tree:

(50)

/ \

(30) (70)

/ \ /

(20)(40)(60)

**🔹 Objective**

* Create this exact binary tree manually.
* Then display the roll numbers (using traversal — e.g., **inorder**, **preorder**, or **postorder**).

**🔹 C++ Program**

We’ll use **struct Node** to define the tree and create each node manually.

#include <iostream>

using namespace std;

// Define a node structure

struct Node {

int data;

Node\* left;

Node\* right;

};

// Function to create a new node

Node\* newNode(int value) {

Node\* node = new Node();

node->data = value;

node->left = node->right = NULL;

return node;

}

// Traversal Functions

void inorder(Node\* root) {

if (root != NULL) {

inorder(root->left);

cout << root->data << " ";

inorder(root->right);

}

}

void preorder(Node\* root) {

if (root != NULL) {

cout << root->data << " ";

preorder(root->left);

preorder(root->right);

}

}

void postorder(Node\* root) {

if (root != NULL) {

postorder(root->left);

postorder(root->right);

cout << root->data << " ";

}

}

int main() {

// Create nodes manually according to the structure

Node\* root = newNode(50);

root->left = newNode(30);

root->right = newNode(70);

root->left->left = newNode(20);

root->left->right = newNode(40);

root->right->left = newNode(60);

cout << "Inorder Traversal: ";

inorder(root);

cout << "\nPreorder Traversal: ";

preorder(root);

cout << "\nPostorder Traversal: ";

postorder(root);

cout << endl;

return 0;

}

**🔹 Output:**

Inorder Traversal: 20 30 40 50 60 70

Preorder Traversal: 50 30 20 40 70 60

Postorder Traversal: 20 40 30 60 70 50

**🔹 Understanding Traversals**

| **Traversal Type** | **Visit Order** | **Output (for this tree)** |
| --- | --- | --- |
| **Inorder** | Left → Root → Right | 20 30 40 50 60 70 |
| **Preorder** | Root → Left → Right | 50 30 20 40 70 60 |
| **Postorder** | Left → Right → Root | 20 40 30 60 70 50 |

**🔹 Viva Quick Answers**

| **Question** | **Answer** |
| --- | --- |
| What is a binary tree? | A data structure where each node has at most two children. |
| Why use it for searching? | Searching in binary trees (especially BSTs) is faster than linear search. |
| What is the root node here? | 50 |
| What are the leaf nodes? | 20, 40, 60 |
| What traversal gives sorted order in a BST? | Inorder Traversal ✅ |

**📚 Q: Online Library – Hashing Book IDs**

**Question:**

An online library wants to store book IDs efficiently using hashing.

The hash function used is:

h(key) = key % table\_size

If the book IDs are [1, 2, 3, 4] and the hash table size is 3,

insert the keys into the hash table and show the final table representation.

**🔹 Beginner Explanation**

* A **hash table** stores data using a **hash function** that maps each key to an index.
* The formula here is simple:
* h(key) = key % table\_size
* This means:
  + Divide the key by the table size
  + The **remainder** is the index where the key will be stored.

**🔹 Step-by-Step Insertion**

**Given:**

* Keys = [1, 2, 3, 4]
* Table size = 3  
  → So valid indexes are **0, 1, 2**

**Insert Key 1**

h(1) = 1 % 3 = **1**  
→ Place key 1 at index 1

**Table:**

| **Index** | **Value** |
| --- | --- |
| 0 | — |
| 1 | 1 |
| 2 | — |

**Insert Key 2**

h(2) = 2 % 3 = **2**  
→ Place key 2 at index 2

**Table:**

| **Index** | **Value** |
| --- | --- |
| 0 | — |
| 1 | 1 |
| 2 | 2 |

**Insert Key 3**

h(3) = 3 % 3 = **0**  
→ Place key 3 at index 0

**Table:**

| **Index** | **Value** |
| --- | --- |
| 0 | 3 |
| 1 | 1 |
| 2 | 2 |

**Insert Key 4**

h(4) = 4 % 3 = **1**  
→ Index 1 is already occupied by key 1.  
→ This is a **collision**.

Now, there are multiple ways to handle collisions:

1. **Linear Probing**
2. **Chaining**
3. **Quadratic Probing**

Let’s assume the simplest → **Linear Probing** (most common for exams).

**Linear Probing Rule:**

If the slot is occupied,  
→ check (index + 1) % table\_size, and so on until an empty slot is found.

So:

* h(4) = 1 → occupied
* Try index = (1 + 1) % 3 = 2 → occupied
* Try index = (2 + 1) % 3 = 0 → occupied  
  → Table full ❌ (no empty slot available)

✅ **Final Table (after all insert attempts):**

| **Index** | **Value** |
| --- | --- |
| 0 | 3 |
| 1 | 1 |
| 2 | 2 |

* Keys inserted successfully: 1, 2, 3
* Key 4 could **not** be inserted (hash table full).

**🔹 C++ Code (Simple Hash Table Insertion)**

#include <iostream>

using namespace std;

#define SIZE 3

int hashFunc(int key) {

return key % SIZE;

}

void insert(int table[], int key) {

int index = hashFunc(key);

int start = index;

while (table[index] != -1) { // linear probing

index = (index + 1) % SIZE;

if (index == start) {

cout << "Table full! Cannot insert " << key << endl;

return;

}

}

table[index] = key;

}

int main() {

int table[SIZE];

for (int i = 0; i < SIZE; i++) table[i] = -1;

int keys[] = {1, 2, 3, 4};

for (int i = 0; i < 4; i++) insert(table, keys[i]);

cout << "\nFinal Hash Table:\n";

for (int i = 0; i < SIZE; i++)

cout << i << " -> " << (table[i] == -1 ? -1 : table[i]) << endl;

return 0;

}

**🔹 Output:**

Table full! Cannot insert 4

Final Hash Table:

0 -> 3

1 -> 1

2 -> 2

**🔹 Viva Quick Points**

| **Question** | **Answer** |
| --- | --- |
| What is the hash function used? | h(key) = key % 3 |
| What is a collision? | When two keys map to the same index. |
| How was collision handled here? | Linear probing. |
| Why couldn’t key 4 be inserted? | All 3 slots were filled. |
| What’s the advantage of hashing? | Fast lookup (O(1) average time). |

A social networking app wants to represent user connections as a graph, where each

user is a node and friendships are edges between them. Given a graph showing user

connections, create the adjacency list representation for it. Create the adjacency matrix

for the given graph.

Based on the provided image of the graph, here are the **adjacency list** and the **adjacency matrix** representations.

The graph has **6 nodes** (users) labeled 1 through 6. The edges represent the connections (friendships) between them.

**Adjacency List**

The **adjacency list** shows, for each node, a list of all nodes it is directly connected to. Since this is an **undirected graph** (friendships are typically mutual), if is connected to , then is also connected to .

* **Node 1:** Connected to nodes 2, 5, and 6.
* **Node 2:** Connected to nodes 1, 3, and 5.
* **Node 3:** Connected to nodes 2, 4, and 5.
* **Node 4:** Connected to nodes 3 and 5.
* **Node 5:** Connected to nodes 1, 2, 3, 4, and 6.
* **Node 6:** Connected to nodes 1 and 5.

**Adjacency Matrix**

The **adjacency matrix** is a square matrix where is **1** if there is an edge between node and node , and **0** otherwise. For this undirected graph, the matrix will be **symmetric** (i.e., ). The nodes are ordered 1 to 6.

| Node | 1 | 2 | 3 | 4 | 5 | 6 |
| --- | --- | --- | --- | --- | --- | --- |
| **1** | 0 | 1 | 0 | 0 | 1 | 1 |
| **2** | 1 | 0 | 1 | 0 | 1 | 0 |
| **3** | 0 | 1 | 0 | 1 | 1 | 0 |
| **4** | 0 | 0 | 1 | 0 | 1 | 0 |
| **5** | 1 | 1 | 1 | 1 | 0 | 1 |
| **6** | 1 | 0 | 0 | 0 | 1 | 0 |

Export to Sheets

A city traffic control system represents road connections between intersections as a

graph, where each intersection is a node and roads are edges. Given the graph, create

the adjacency matrix representation for it.

This request is identical to the previous one, but with a different context (traffic control/intersections instead of social networking/users). The underlying graph structure, nodes, and edges are the same as shown in the image.

Here is the **Adjacency Matrix** representation for the given graph:

The graph has **6 nodes** (intersections) labeled 1 through 6. The entry in the matrix is **1** if a road connects intersection and intersection , and **0** otherwise.

This request is identical to the previous one, but with a different context (traffic control/intersections instead of social networking/users). The underlying graph structure, nodes, and edges are the same as shown in the image.

Here is the **Adjacency Matrix** representation for the given graph:

The graph has **6 nodes** (intersections) labeled 1 through 6. The entry in the matrix is **1** if a road connects intersection and intersection , and **0** otherwise.

| Intersection | 1 | 2 | 3 | 4 | 5 | 6 |
| --- | --- | --- | --- | --- | --- | --- |
| **1** | 0 | 1 | 0 | 0 | 1 | 1 |
| **2** | 1 | 0 | 1 | 0 | 1 | 0 |
| **3** | 0 | 1 | 0 | 1 | 1 | 0 |
| **4** | 0 | 0 | 1 | 0 | 1 | 0 |
| **5** | 1 | 1 | 1 | 1 | 0 | 1 |
| **6** | 1 | 0 | 0 | 0 | 1 | 0 |

​​