DSA Question Bank

Bubble Sort:

```
#include <iostream>
using namespace std;
int main() {
    int arr[5] = {5, 4, 3, 2, 1}; // given package IDs
    int n = 5;
    for(int i = 0; i < n - 1; i++) {       // number of passes
        for(int j = 0; j < n - i - 1; j++) { // compare adjacent elements
            if(arr[j] > arr[j + 1]) {
                // swap them if they are in the wrong order
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }    cout << "Sorted Package IDs: ";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Insertion Sort

```
#include <iostream>
using namespace std;
int main() {
   int arr[5] = {5, 4, 3, 2, 1}; // given package IDs
   int n = 5;
      for (int i = 1; i < n; i++) {
      int key = arr[i];      // take the current element
      int j = i - 1;
      // Move elements that are greater than key to one position ahead
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j--;
      }
      arr[j + 1] = key;      // insert the key at the correct position
   }
   // Print the sorted array
   cout << "Sorted Package IDs: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
   return 0;
}
```

Selection Sort

```
#include <iostream>
```

```
using namespace std;
int main() {
   int arr[5] = {5, 4, 3, 2, 1}; // given package IDs
   int n = 5;
    for (int i = 0; i < n - 1; i++) {
      int minIndex = i; // assume the first element is the smallest
       for (int j = i + 1; j < n; j++) {
          if (arr[j] < arr[minIndex]) {
              minIndex = j; // update index of smallest element
          }
      }
      int temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
   }
   cout << "Sorted Package IDs: ";
   for (int i = 0; i < n; i++) {
  cout << arr[i] << " ";
   }

   return 0;
}
```

Linked List

```
#include <iostream>
```

```
using namespace std;
struct Node {
   int data;       // stores patient ID
   Node* next;     // pointer to next node
};
int main() {
   // Create three nodes
   Node* first = new Node();
   Node* second = new Node();
   Node* third = new Node();
   first->data = 111;
   second->data = 123;
   third->data = 124;
   first->next = second;
   second->next = third;
   third->next = NULL; // last node points to NULL
    cout << "Patient IDs in linked list: ";
   Node* temp = first;
   while (temp != NULL) {
      cout << temp->data << " -> ";
      temp = temp->next;
   }
   cout << "NULL" << endl;
   return 0;
}
```

Binary Tree Searching

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

```
struct Node {
    int data;       // stores student roll number
    Node* left;     // pointer to left child
    Node* right;    // pointer to right child
};
Node* createNode(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
 void inorder(Node* root) {
    if (root == NULL)
        return;
    inorder(root->left);         // visit left child
    cout << root->data << " ";   // print root data
    inorder(root->right);        // visit right child
}
int main() {
        Node* root = createNode(50);
    root->left = createNode(30);
    root->right = createNode(70);
    root->left->left = createNode(20);
    root->left->right = createNode(40);
    root->right->left = createNode(60);
     cout << "Inorder Traversal of Binary Tree: ";
    inorder(root);
    cout << endl;
```

```
    return 0;
}
```

Counting Sort

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

int main() {
    int wallets[10] = {2, 2, 2, 3, 3, 0, 0, 1, 4}; // 9 wallets given, let's add one more for total
10
    wallets[9] = 1; // adding 10th wallet

    int max_money = 6;
    int count[7] = {0}; // counts from 0 to 6

    // Count the money in each wallet
    for(int i = 0; i < 10; i++) {
        count[wallets[i]]++;
    }

    // Print the sorted wallets
    cout << "Wallets in ascending order: ";
    for(int i = 0; i <= max_money; i++) {
        for(int j = 0; j < count[i]; j++) {
            cout << i << " ";
        }
    }
```

```
    return 0;
}
```

Quick Sort

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

// Quick Sort function
void quickSort(int arr[], int start, int end) {
    if(start >= end) return;

    int pivot = arr[end]; // last element as pivot
    int i = start;

    for(int j = start; j < end; j++) {
        if(arr[j] < pivot) {
            swap(arr[i], arr[j]);
            i++;
        }
    }
    swap(arr[i], arr[end]);

    quickSort(arr, start, i - 1);
    quickSort(arr, i + 1, end);
}

int main() {
    int scores[12] = {45, 12, 78, 34, 23, 89, 67, 11, 90, 54, 32, 76};
```

```
    quickSort(scores, 0, 11);

    cout << "Scores in ascending order: ";
    for(int i = 0; i < 12; i++) {
        cout << scores[i] << " ";
    }

    return 0;
}
```

Traverse Linked List

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

// Node structure
struct Player {
    string name;
    Player* next;
};

int main() {
    int n;
    cout << "Enter number of players: ";
    cin >> n;
    cin.ignore();
```

```
    Player* head = nullptr;
    Player* last = nullptr;

    // Input players and create linked list
    for(int i = 0; i < n; i++) {
        Player* p = new Player;
        cout << "Enter name of player " << i+1 << ": ";
        getline(cin, p->name);
        p->next = nullptr;

        if(head == nullptr) head = p;  // first player
        else last->next = p;           // link previous to current

        last = p; // move last pointer
    }

    // Chief guest meets players
    cout << "\nChief Guest meets players:\n";
    Player* curr = head;
    while(curr) {
        cout << "Meeting " << curr->name << endl;
        curr = curr->next;
    }

    return 0;
}
```

Doubly Linked List

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

int main() {
   string stops[] = {"A", "B", "C", "D"};
   int n = 4;

   // Onward journey
   cout << "Onward Journey: ";
   for(int i = 0; i < n; i++) {
      cout << stops[i] << " ";
   }

   // Return journey
   cout << "\nReturn Journey: ";
   for(int i = n-1; i >= 0; i--) {
      cout << stops[i] << " ";
   }

   return 0;
}
```

PRE,POST,IN trversal

```
#include using namespace std;
// Node structure for BST struct Node {
 int data;
Node* left;
```

```
Node* right;

};

// Function to create a new node

Node* createNode(int value)

{

Node* newNode = new Node;

newNode->data = value;

newNode->left = newNode->right = nullptr;

 return newNode; }

// Insert value in BST

Node* insert(Node* root, int value) {

if(root == nullptr)

 return createNode(value);

if(value < root->data)
    root->left = insert(root->left, value);
else
    root->right = insert(root->right, value);

return root;


}

// In-Order Traversal:

Left -> Root -> Right void

inOrder(Node* root) {

if(root == nullptr) return;

inOrder(root->left);

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

inOrder(root->right);
```

```
}
// Pre-Order Traversal:
Root -> Left -> Right void preOrder(Node* root) {
if(root == nullptr) return;
cout << root->data << " ";
preOrder(root->left);
preOrder(root->right);
 }
// Post-Order Traversal:
Left -> Right -> Root void postOrder(Node* root) {
 if(root == nullptr) return;
 postOrder(root->left);
 postOrder(root->right); cout << root->data << " "; }
int main() { Node* root = nullptr;
int values[] = {50, 30, 20, 40, 70, 60, 80}; int n = 7;

return 0;
}
```

Key Insertion Into Hash

```
#include <iostream>
using namespace std;
int main() {
    int table_size = 3;
    int hashTable[3] = {-1, -1, -1}; // -1 means empty
    int keys[4] = {1, 2, 3, 4};
    for (int i = 0; i < 4; i++) {
        int key = keys[i];
        int index = key % table_size; // hash function
        int start = index;
        bool inserted = false;
         do {
            if (hashTable[index] == -1) {
                hashTable[index] = key;
```

```
            inserted = true;
            break;
        }
        index = (index + 1) % table_size;
    } while (index != start);
    if (!inserted)
        cout << "Table is full! Cannot insert " << key << endl;
}
 cout << "\nFinal Hash Table:\n";
for (int i = 0; i < table_size; i++) {
    cout << "Index " << i << " → ";
    if (hashTable[i] == -1)
        cout << "Empty" << endl;
    else
        cout << hashTable[i] << endl;
}
return 0;
}
```

Matrix Creation :

```
#include <iostream>
using namespace std;
int main() {
    int n = 6;
 int graph[7][7] = {0};
 graph[1][2] = graph[2][1] = 1;
graph[1][5] = graph[5][1] = 1;
```

```
    graph[1][6] = graph[6][1] = 1;
    graph[2][3] = graph[3][2] = 1;
    graph[2][5] = graph[5][2] = 1;
    graph[3][4] = graph[4][3] = 1;
    graph[3][5] = graph[5][3] = 1;
    graph[4][5] = graph[5][4] = 1;
    graph[5][6] = graph[6][5] = 1;
     cout << "Adjacency Matrix:\n";
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
     cout << "\nAdjacency List:\n";
    for (int i = 1; i <= n; i++) {
        cout << i << " -> ";
        for (int j = 1; j <= n; j++) {
            if (graph[i][j] == 1) {
                cout << j << " ";
            }
        }
        cout << endl;
    }
    return 0;
}
```

Graph Matrix

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

int main() {
    int n = 6; // number of intersections
    int graph[7][7] = {0}; // 1-based indexing
     graph[1][2] = graph[2][1] = 1;
    graph[1][5] = graph[5][1] = 1;
    graph[1][6] = graph[6][1] = 1;
    graph[2][3] = graph[3][2] = 1;
    graph[2][5] = graph[5][2] = 1;
    graph[3][4] = graph[4][3] = 1;
    graph[3][5] = graph[5][3] = 1;
    graph[4][5] = graph[5][4] = 1;
    graph[5][6] = graph[6][5] = 1;
     cout << "Adjacency Matrix:\n";
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            cout << graph[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```