Q.1-implimentatiion of stack using array .

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

class Stack {
   int* arr;
   int top, capacity;

public:
   Stack(int size) : capacity(size), top(-1) { arr
= new int[size]; }
   ~Stack() { delete[] arr; }

   void push(int x) {
      if (top == capacity - 1) cout << "Stack
Overflow\n";
      else arr[++top] = x;
   }
   int pop() {
      return (top == -1) ? (cout << "Stack
Underflow\n", -1) : arr[top--];
   }
   int peek() {
      return (top == -1) ? (cout << "Stack is
Empty\n", -1) : arr[top];
   }
   bool isEmpty() { return top == -1; }
   void display() {
      if (isEmpty()) cout << "Stack is Empty\n";
      else for (int i = 0; i <= top; i++) cout <<
arr[i] << " ";
      cout << "\n";
   }
};

int main() {
   Stack s(5);
   s.push(10); s.push(20); s.push(30);
   s.display(); // Output: 10 20 30
   cout << "Top: " << s.peek() << "\n"; //
Output: 30
   s.pop(); s.display(); // Output: 10 20
   return 0;
}
```

Q.2-implimentatiion of Queue using array

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

class Queue {
   int* arr, front, rear, capacity;

public:
   Queue(int size) : capacity(size), front(0),
rear(-1) { arr = new int[size]; }
   ~Queue() { delete[] arr; }

   void enqueue(int x) {
      if (rear == capacity - 1) cout << "Queue
Overflow\n";
      else arr[++rear] = x;
   }
   int dequeue() {
      return (front > rear) ? (cout << "Queue
Underflow\n", -1) : arr[front++];
   }
   int peek() {
      return (front > rear) ? (cout << "Queue is
Empty\n", -1) : arr[front];
   }
   bool isEmpty() { return front > rear; }
   void display() {
      if (isEmpty()) cout << "Queue is
Empty\n";
      else for (int i = front; i <= rear; i++) cout
<< arr[i] << " ";
      cout << "\n";
   }
};

int main() {
   Queue q(5);
   q.enqueue(10); q.enqueue(20);
q.enqueue(30);
   q.display(); // Output: 10 20 30
   cout << "Front: " << q.peek() << "\n"; //
Output: 10
   q.dequeue(); q.display(); // Output: 20 30
   return 0;
}
```

Q.3-implimentatiion of Linked List & Doubly List .

----Linked List----

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

struct Node {
   int data;
   Node* next;
   Node(int x) : data(x), next(nullptr) {}
};

class LinkedList {
   Node* head;

public:
   LinkedList() : head(nullptr) {}

   void insert(int x) {
      Node* newNode = new Node(x);
      newNode->next = head;
      head = newNode;
   }

   void display() {
      Node* temp = head;
      while (temp) {
         cout << temp->data << " ";
         temp = temp->next;
      }
      cout << "\n";
   }
};

int main() {
   LinkedList list;
   list.insert(10);
   list.insert(20);
   list.insert(30);
   list.display(); // Output: 30 20 10
   return 0;
}
```

```
#include <iostream>

using namespace std;

struct Node {

   int data;

   Node* prev;

   Node* next;

   Node(int x) : data(x), prev(nullptr), next(nullptr) {}

};

class DoublyLinkedList {

   Node* head;

public:

   DoublyLinkedList() : head(nullptr) {}

   void insert(int x) {

      Node* newNode = new Node(x);

      newNode->next = head;

      if (head) head->prev = newNode;

      head = newNode;

   }

   void display() {

      Node* temp = head;

      while (temp) {

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

         temp = temp->next;

      }

      cout << "\n";

   }

};

int main() {

   DoublyLinkedList list;

   list.insert(10);

   list.insert(20);

   list.insert(30);

   list.display(); // Output: 30 20 10

   return 0; }
```

Q.4-implimentatiion of Linear & Binary Search.

----LINEAR----

```
#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; // Key found
   }
   return -1; // Key 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);
   cout << (result != -1 ? "Found at index " +
to_string(result) : "Not Found") << "\n";
   return 0;
}
```

----BINARY----

```
#include <iostream>

using namespace std;

int binarySearch(int arr[], int n, int key) {
   int low = 0, high = n - 1;
   while (low <= high) {
      int mid = low + (high - low) / 2;
      if (arr[mid] == key) return mid;   // Key
found
      else if (arr[mid] < key) low = mid + 1; //
Search right half
      else high = mid - 1;            // Search left
half
   }
   return -1; // Key not found
}

int main() {
   int arr[] = {10, 20, 30, 40, 50};
   int n = sizeof(arr) / sizeof(arr[0]);
   int key = 40;
   int result = binarySearch(arr, n, key);
   cout << (result != -1 ? "Found at index " +
to_string(result) : "Not Found") << "\n";
   return 0;
}
```

Q.5-implimentatiion of DFS & BFS.

----DFS----

```
#include <iostream>
#include <vector>
using namespace std;
void DFS(int node, vector<vector<int>>& graph, vector<bool>& visited) {
   visited[node] = true;
   cout << node << " ";
   for (int neighbor : graph[node]) {
      if (!visited[neighbor]) DFS(neighbor, graph, visited);
   }
}
int main() {
   int n = 5; // Number of nodes
   vector<vector<int>> graph = {
      {},       // Node 0 (unused)
      {2, 3},    // Node 1
      {1, 4},    // Node 2
      {1, 5},    // Node 3
      {2},      // Node 4
      {3}       // Node 5
   };
   vector<bool> visited(n + 1, false);
   cout << "DFS: ";
   DFS(1, graph, visited); // Start DFS from node 1
   return 0;
}
```

----BFS----

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
void BFS(int start, vector<vector<int>>& graph) {
   vector<bool> visited(graph.size(), false);
   queue<int> q;
   visited[start] = true;
   q.push(start);
   while (!q.empty()) {
      int node = q.front();
      q.pop();
      cout << node << " ";
      for (int neighbor : graph[node]) {
         if (!visited[neighbor]) {
            visited[neighbor] = true;
            q.push(neighbor);
         }}}}
int main() {
   int n = 5; // Number of nodes
   vector<vector<int>> graph = {
      {},       // Node 0 (unused)
      {2, 3},    // Node 1
      {1, 4},    // Node 2
      {1, 5},    // Node 3
      {2},      // Node 4
      {3}       // Node 5
   };

   cout << "BFS: ";
   BFS(1, graph); // Start BFS from node 1
   return 0; }
```

Q.6- Create a Hash Table and Handle the collision using Linear Probing with or without Replacement.

```
#include <iostream>

#include <vector>

using namespace std;


class HashTable {

   vector<int> table;

   int size;


   int hash(int key) {

      return key % size; // Hash function

   }


public:

   HashTable(int s) : size(s), table(s, -1) {} // Initialize table with -1 (empty)


   void insert(int key) {

      int index = hash(key);

      if (table[index] == -1) {

         table[index] = key; // Place in the hashed index if empty

      } else {

         // Collision: Linear probing without replacement

         int originalIndex = hash(table[index]);

         if (originalIndex == index) {

            // Continue probing if the existing element belongs to the same slot

            index = (index + 1) % size;

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

               index = (index + 1) % size;

            }

            table[index] = key;

         } else {

            cout << "Collision at index " << index << ": Cannot replace existing key\n";

         }

      }

   }


   void display() {

      for (int i = 0; i < size; i++) {

         cout << i << " : " << (table[i] == -1 ? "Empty" : to_string(table[i])) << "\n";

      }

   }

};


int main() {

   HashTable ht(7); // Create a hash table of size 7

   ht.insert(10);

   ht.insert(20);

   ht.insert(17); // Collision: Probing without replacement

   ht.display();

   return 0;

}
```

Q.7-implimentatiion of Bubble Sort Algorithm

```
#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(arr[j], arr[j + 1]); // Swap adjacent elements
         }
      }
   }
}


int main() {
   int arr[] = {64, 34, 25, 12, 22, 11, 90};
   int n = sizeof(arr) / sizeof(arr[0]);


   bubbleSort(arr, n);


   cout << "Sorted array: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
   return 0;
}
```

Q.8-implimentatiion of Insertion Sort Algorithm

```
#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 of arr[0..i-1] that are greater than key
      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[] = {12, 11, 13, 5, 6};
   int n = sizeof(arr) / sizeof(arr[0]);
   insertionSort(arr, n);
   cout << "Sorted array: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
   return 0;
}
```

Q.9-implimentatiion of Merge Sort Algorithm

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

void merge(int arr[], int left, int mid, int right)
{
   int n1 = mid - left + 1;
   int n2 = right - mid;

   int L[n1], R[n2];
   for (int i = 0; i < n1; i++) L[i] = arr[left + i];
   for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 +
i];

   int i = 0, j = 0, k = left;
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) arr[k++] = L[i++];
      else arr[k++] = R[j++];
   }

   while (i < n1) arr[k++] = L[i++];
   while (j < n2) arr[k++] = R[j++];
}

void mergeSort(int arr[], int left, int right) {
   if (left < right) {
      int mid = left + (right - left) / 2;
      mergeSort(arr, left, mid);
      mergeSort(arr, mid + 1, right);
      merge(arr, left, mid, right);
   }
}

int main() {
   int arr[] = {12, 11, 13, 5, 6, 7};
   int n = sizeof(arr) / sizeof(arr[0]);

   mergeSort(arr, 0, n - 1);

   cout << "Sorted array: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
   return 0;
}
```

Q.10-implimentatiion of Heap Sort Algorithm

```
#include <iostream>

using namespace std;

void heapify(int arr[], int n, int i) {
   int largest = i;      // Initialize largest as root
   int left = 2 * i + 1;  // Left child
   int right = 2 * i + 2; // Right child
   if (left < n && arr[left] > arr[largest]) largest
= left;
   if (right < n && arr[right] > arr[largest])
largest = right;


   if (largest != i) { // Swap and heapify if root
is not largest
      swap(arr[i], arr[largest]);
      heapify(arr, n, largest);
   }
}
void heapSort(int arr[], int n) {
   // Build heap (rearrange array)
   for (int i = n / 2 - 1; i >= 0; i--) {
      heapify(arr, n, i);
   }
   // Extract elements from heap one by one
   for (int i = n - 1; i > 0; i--) {
      swap(arr[0], arr[i]); // Move current root
to end
      heapify(arr, i, 0);   // Call heapify on
reduced heap
   }
}

int main() {
   int arr[] = {12, 11, 13, 5, 6, 7};
   int n = sizeof(arr) / sizeof(arr[0]);


   heapSort(arr, n);


   cout << "Sorted array: ";
   for (int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
   return 0;
}
```