- **HEAP**

```cpp
void convertToMaxHeapAndPrint(int arr[], int n) {
    // Convert the min heap to max heap starting from last non-leaf node
    for (int i = n / 2 - 1; i >= 0; i--) {
        int largest = i;
        int current = i;

        while (true) {
            int left = 2 * current + 1; // Left child index
            int right = 2 * current + 2; // Right child index

            // Find the largest element among current, left child, and right child
            if (left < n && arr[left] > arr[largest]) {
                largest = left;
            }
            if (right < n && arr[right] > arr[largest]) {
                largest = right;
            }

            // If current node is the largest, then break
            if (largest == current) break;

            // Swap and continue heapifying downwards
            swap(arr[current], arr[largest]);
            current = largest;
        }
    }

    // Print the result as a max heap array
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

➢ Dijkstra Shortest Path

```
// Function to print shortest paths from source
void Graph::shortestPath(int src) {
    // Create a simple array-based priority queue
    bool visited[V];
    memset(visited, false, sizeof(visited));

    // Array to store distances and initialize all distances as INF
    int dist[V];
    for (int i = 0; i < V; ++i)
        dist[i] = INF;

    // Insert source into "queue" and initialize its distance as 0
    dist[src] = 0;

    // Iterate through all vertices
    for (int count = 0; count < V; ++count) {
        // Find the minimum distance vertex not yet processed
        int u = -1;
        for (int i = 0; i < V; ++i) {
            if (!visited[i] && (u == -1 || dist[i] < dist[u]))
                u = i;
        }

        // Mark the vertex as processed
        visited[u] = true;

        // Update distances of adjacent vertices of the current vertex
        for (auto& neighbor : adj[u]) {
            int v = neighbor.first;
            int weight = neighbor.second;

            // If a shorter path to v is found
            if (dist[v] > dist[u] + weight) {
                dist[v] = dist[u] + weight;
            }
        }
    }
```

➢ MERGE SORT

```cpp
// Function to merge two sorted linked lists into a third list
void mergeSortedLists(LinkedList& list1, LinkedList& list2, LinkedList& list3) {
    Node* ptr1 = list1.getHead(); // Pointer to traverse list1
    Node* ptr2 = list2.getHead(); // Pointer to traverse list2

    // Traverse both lists and insert nodes in sorted order
    while (ptr1 != nullptr && ptr2 != nullptr) {
        if (ptr1->getData() < ptr2->getData()) {
            list3.append(ptr1->getData()); // Add the smaller value to list3
            ptr1 = ptr1->getNext();       // Move to the next node in list1
        } else {
            list3.append(ptr2->getData()); // Add the smaller value to list3
            ptr2 = ptr2->getNext();       // Move to the next node in list2
        }
    }

    // If there are remaining nodes in list1, add them to list3
    while (ptr1 != nullptr) {
        list3.append(ptr1->getData());
        ptr1 = ptr1->getNext();
    }

    // If there are remaining nodes in list2, add them to list3
    while (ptr2 != nullptr) {
        list3.append(ptr2->getData());
        ptr2 = ptr2->getNext();
    }
}
```

To check if a graph contains cycles and calculates the total weight of the edges involved, we need to traverse the graph.

For this, we can use DFS (Depth First Search) or Union-Find (Disjoint Set Union) methods depending on whether the graph is directed or undirected. **Here is the code for:**

1. Detecting cycles in a weighted graph using DFS.

2. Returning the sum of edge weights if a cycle is detected.

```
// Recursive DFS function to detect cycles and calculate weight
bool Graph::hasCycle(int node, int parent, bool visited[], int& cycleWeight) {
    visited[node] = true;

    for (auto& neighbor : adj[node]) {
        int v = neighbor.first;
        int weight = neighbor.second;

        // If neighbor is not visited, recurse
        if (!visited[v]) {
            cycleWeight += weight; // Add weight of the edge
            if (hasCycle(v, node, visited, cycleWeight)) {
                return true;
            }
        }
        // If visited and not the parent, it's a cycle
        else if (v != parent) {
            cycleWeight += weight; // Add weight of the edge in the cycle
            return true;
        }
    }
    return false;
}
```

- **AVL tree insertion**

```
// Helper function to get the height of a node
int height(Node* n) {
    if (n == nullptr) return 0;
    return n->getHeight();
}

// Helper function to calculate the balance factor
int getBalanceFactor(Node* n) {
    if(n == nullptr) return 0;
    return height(n->getLeft()) - height(n->getRight());
}
```

```cpp
// Right rotation
Node* rightRotate(Node* y) {
    Node* x = y->getLeft();
    Node* T2 = x->getRight();

    // Perform rotation
    x->setRight(y);
    y->setLeft(T2);

    // Update heights
    y->setHeight(1 + max(height(y->getLeft()), height(y->getRight())));
    x->setHeight(1 + max(height(x->getLeft()), height(x->getRight())));

    return x;  // New root
}

// Left rotation
Node* leftRotate(Node* x) {
    Node* y = x->getRight();
    Node* T2 = y->getLeft();

    // Perform rotation
    y->setLeft(x);
    x->setRight(T2);

    // Update heights
    x->setHeight(1 + max(height(x->getLeft()), height(x->getRight())));
    y->setHeight(1 + max(height(y->getLeft()), height(y->getRight())));

    return y;  // New root
}
```

```cpp
// AVL tree insertion
Node* insert(Node* node, int key) {
    // Perform normal BST insertion
    if (node == nullptr)
        return new Node(key);

    if (key < node->getKey()) {
        node->setLeft(insert(node->getLeft(), key));
    } else if (key > node->getKey()) {
        node->setRight(insert(node->getRight(), key));
    } else {
        return node; // Duplicate keys not allowed
    }

    // Update height of the current node
    node->setHeight(1 + max(height(node->getLeft()), height(node->getRight())));

    // Get balance factor to check if node became unbalanced
    int balance = getBalanceFactor(node);

    // Balance the tree
    // Left Left Case
    if (balance > 1 && key < node->getLeft()->getKey())
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->getRight()->getKey())
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->getLeft()->getKey()) {
        node->setLeft(leftRotate(node->getLeft()));
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->getRight()->getKey()) {
        node->setRight(rightRotate(node->getRight()));
        return leftRotate(node);
    }

    return node;  // Return the unchanged node pointer
}
```