Heap and Tree Data Structures in C++

1. Tree Data Structure

A tree is a hierarchical data structure with nodes connected by edges. The topmost node is called the root.

Basic Tree Implementation

cpp

```cpp
#include <iostream>

#include <vector>

using namespace std;


// Basic TreeNode structure

template <typename T>

class TreeNode {

public:

    T data;

    vector<TreeNode*> children;


    TreeNode(T val) : data(val) {}


    void addChild(TreeNode* child) {

        children.push_back(child);

    }

};


// Binary Tree Node

template <typename T>

class BinaryTreeNode {
```

```
public:
    T data;
    BinaryTreeNode* left;
    BinaryTreeNode* right;

    BinaryTreeNode(T val) : data(val), left(nullptr), right(nullptr) {}
};

// Binary Tree Class
template <typename T>
class BinaryTree {
private:
    BinaryTreeNode<T>* root;

    BinaryTreeNode<T>* insertHelper(BinaryTreeNode<T>* node, T value) {
        if (!node) return new BinaryTreeNode<T>(value);

        if (value < node->data) {
            node->left = insertHelper(node->left, value);
        } else {
            node->right = insertHelper(node->right, value);
        }
        return node;
    }

    void inorderHelper(BinaryTreeNode<T>* node) {
        if (!node) return;
```

```
        inorderHelper(node->left);
        cout << node->data << " ";
        inorderHelper(node->right);
    }

    void preorderHelper(BinaryTreeNode<T>* node) {
        if (!node) return;
        cout << node->data << " ";
        preorderHelper(node->left);
        preorderHelper(node->right);
    }

    void postorderHelper(BinaryTreeNode<T>* node) {
        if (!node) return;
        postorderHelper(node->left);
        postorderHelper(node->right);
        cout << node->data << " ";
    }

public:
    BinaryTree() : root(nullptr) {}

    void insert(T value) {
        root = insertHelper(root, value);
    }

    void inorder() {
```

```
        cout << "Inorder: ";
        inorderHelper(root);
        cout << endl;
    }

    void preorder() {
        cout << "Preorder: ";
        preorderHelper(root);
        cout << endl;
    }

    void postorder() {
        cout << "Postorder: ";
        postorderHelper(root);
        cout << endl;
    }
};

// Example usage
void treeExample() {
    cout << "=== Binary Tree Example ===" << endl;

    BinaryTree<int> tree;
    tree.insert(5);
    tree.insert(3);
    tree.insert(7);
    tree.insert(2);
```

```
    tree.insert(4);
    tree.insert(6);
    tree.insert(8);

    tree.inorder();    // 2 3 4 5 6 7 8
    tree.preorder();   // 5 3 2 4 7 6 8
    tree.postorder();  // 2 4 3 6 8 7 5
}
```

2. Heap Data Structure

A heap is a special tree-based data structure that satisfies the heap property:

- **Max Heap**: Parent node >= children nodes
- **Min Heap**: Parent node <= children nodes

Heap Implementation

cpp

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template <typename T>
class MaxHeap {
private:
    vector<T> heap;

    void heapifyUp(int index) {
        while (index > 0) {
            int parent = (index - 1) / 2;
```

```
        if (heap[index] > heap[parent]) {
            swap(heap[index], heap[parent]);
            index = parent;
        } else {
            break;
        }
    }
}

void heapifyDown(int index) {
    int size = heap.size();
    while (true) {
        int left = 2 * index + 1;
        int right = 2 * index + 2;
        int largest = index;

        if (left < size && heap[left] > heap[largest]) {
            largest = left;
        }
        if (right < size && heap[right] > heap[largest]) {
            largest = right;
        }

        if (largest != index) {
            swap(heap[index], heap[largest]);
            index = largest;
        } else {
```

```
                break;
            }
        }
    }

public:
    MaxHeap() {}

    void insert(T value) {
        heap.push_back(value);
        heapifyUp(heap.size() - 1);
    }

    T extractMax() {
        if (heap.empty()) {
            throw out_of_range("Heap is empty");
        }

        T maxValue = heap[0];
        heap[0] = heap.back();
        heap.pop_back();

        if (!heap.empty()) {
            heapifyDown(0);
        }

        return maxValue;
```

```
    }

    T getMax() {
        if (heap.empty()) {
            throw out_of_range("Heap is empty");
        }
        return heap[0];
    }

    bool isEmpty() {
        return heap.empty();
    }

    int size() {
        return heap.size();
    }

    void printHeap() {
        cout << "Heap: ";
        for (T val : heap) {
            cout << val << " ";
        }
        cout << endl;
    }
};

// Min Heap Implementation
```

```
template <typename T>
class MinHeap {
private:
    vector<T> heap;

    void heapifyUp(int index) {
        while (index > 0) {
            int parent = (index - 1) / 2;
            if (heap[index] < heap[parent]) {
                swap(heap[index], heap[parent]);
                index = parent;
            } else {
                break;
            }
        }
    }

    void heapifyDown(int index) {
        int size = heap.size();
        while (true) {
            int left = 2 * index + 1;
            int right = 2 * index + 2;
            int smallest = index;

            if (left < size && heap[left] < heap[smallest]) {
                smallest = left;
            }
```

```
            if (right < size && heap[right] < heap[smallest]) {
                smallest = right;
            }

            if (smallest != index) {
                swap(heap[index], heap[smallest]);
                index = smallest;
            } else {
                break;
            }
        }
    }

public:
    MinHeap() {}

    void insert(T value) {
        heap.push_back(value);
        heapifyUp(heap.size() - 1);
    }

    T extractMin() {
        if (heap.empty()) {
            throw out_of_range("Heap is empty");
        }

        T minValue = heap[0];
```

```
        heap[0] = heap.back();
        heap.pop_back();

        if (!heap.empty()) {
            heapifyDown(0);
        }

        return minValue;
    }

    T getMin() {
        if (heap.empty()) {
            throw out_of_range("Heap is empty");
        }
        return heap[0];
    }

    bool isEmpty() {
        return heap.empty();
    }

    int size() {
        return heap.size();
    }
};

// Example usage
```

```
void heapExample() {
    cout << "\n=== Max Heap Example ===" << endl;

    MaxHeap<int> maxHeap;
    maxHeap.insert(10);
    maxHeap.insert(20);
    maxHeap.insert(15);
    maxHeap.insert(30);
    maxHeap.insert(5);

    maxHeap.printHeap();  // 30 20 15 10 5

    cout << "Max element: " << maxHeap.extractMax() << endl;  // 30
    cout << "Next max: " << maxHeap.getMax() << endl;  // 20

    cout << "\n=== Min Heap Example ===" << endl;

    MinHeap<int> minHeap;
    minHeap.insert(10);
    minHeap.insert(20);
    minHeap.insert(15);
    minHeap.insert(30);
    minHeap.insert(5);

    cout << "Min element: " << minHeap.extractMin() << endl;  // 5
    cout << "Next min: " << minHeap.getMin() << endl;  // 10
}
```

3. Using STL Containers

C++ Standard Library provides implementations for both:

cpp

```cpp
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>
using namespace std;

void stlExamples() {
    cout << "\n=== STL Examples ===" << endl;

    // Priority Queue (Max Heap by default)
    priority_queue<int> maxHeapPQ;
    maxHeapPQ.push(10);
    maxHeapPQ.push(20);
    maxHeapPQ.push(15);
    maxHeapPQ.push(30);
    maxHeapPQ.push(5);

    cout << "Max Heap using priority_queue: ";
    while (!maxHeapPQ.empty()) {
        cout << maxHeapPQ.top() << " ";
        maxHeapPQ.pop();
    }
    cout << endl;
```

```cpp
// Min Heap using priority_queue with greater comparator
priority_queue<int, vector<int>, greater<int>> minHeapPQ;
minHeapPQ.push(10);
minHeapPQ.push(20);
minHeapPQ.push(15);
minHeapPQ.push(30);
minHeapPQ.push(5);

cout << "Min Heap using priority_queue: ";
while (!minHeapPQ.empty()) {
    cout << minHeapPQ.top() << " ";
    minHeapPQ.pop();
}
cout << endl;

// Make heap from vector
vector<int> vec = {10, 20, 15, 30, 5};
make_heap(vec.begin(), vec.end());  // Creates max heap

cout << "Max element in heap: " << vec.front() << endl;  // 30

// Add new element
vec.push_back(40);
push_heap(vec.begin(), vec.end());
cout << "New max element: " << vec.front() << endl;  // 40

// Remove max element
```

```
    pop_heap(vec.begin(), vec.end());
    vec.pop_back();
    cout << "Max after pop: " << vec.front() << endl;  // 30
}
```

4. Practical Applications

```cpp
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

class HeapApplications {
public:
    // Heap Sort using Max Heap
    static void heapSort(vector<int>& arr) {
        MaxHeap<int> heap;

        // Build heap
        for (int num : arr) {
            heap.insert(num);
        }

        // Extract elements in sorted order
        for (int i = arr.size() - 1; i >= 0; i--) {
            arr[i] = heap.extractMax();
        }
```

```
}

// Find K largest elements
static vector<int> findKLargest(const vector<int>& nums, int k) {
    MinHeap<int> minHeap;

    for (int num : nums) {
        minHeap.insert(num);
        if (minHeap.size() > k) {
            minHeap.extractMin();
        }
    }

    vector<int> result;
    while (!minHeap.isEmpty()) {
        result.push_back(minHeap.extractMin());
    }
    return result;
}

// Priority Queue example
struct Task {
    int priority;
    string name;

    bool operator<(const Task& other) const {
```

```
            return priority < other.priority; // Higher priority first
        }
    };

    static void taskScheduler() {
        priority_queue<Task> taskQueue;

        taskQueue.push({1, "Low priority task"});
        taskQueue.push({3, "Medium priority task"});
        taskQueue.push({5, "High priority task"});
        taskQueue.push({2, "Another low priority task"});

        cout << "\n=== Task Execution Order ===" << endl;
        while (!taskQueue.empty()) {
            Task task = taskQueue.top();
            taskQueue.pop();
            cout << "Executing: " << task.name
                 << " (Priority: " << task.priority << ")" << endl;
        }
    }
};

void applicationExamples() {
    cout << "\n=== Application Examples ===" << endl;

    // Heap Sort
```

```cpp
    vector<int> arr = {12, 11, 13, 5, 6, 7};
    cout << "Original array: ";
    for (int num : arr) cout << num << " ";
    cout << endl;

    HeapApplications::heapSort(arr);

    cout << "Sorted array: ";
    for (int num : arr) cout << num << " ";
    cout << endl;

    // K largest elements
    vector<int> nums = {3, 2, 1, 5, 6, 4};
    int k = 3;
    vector<int> kLargest = HeapApplications::findKLargest(nums, k);

    cout << k << " largest elements: ";
    for (int num : kLargest) cout << num << " ";
    cout << endl;

    // Task scheduler
    HeapApplications::taskScheduler();
}
```

5. Comparison and Key Differences

Aspect	Tree	Heap
Structure	General hierarchy	Complete binary tree
Ordering	Varies (BST, AVL, etc.)	Heap property (min/max)
Operations	Insert, Delete, Search, Traversal	Insert, Extract min/max
Complexity	Varies by type	O(log n) for insert/extract
Use Case	Searching, sorting, hierarchy	Priority queue, heap sort

6. Complete Example Program

cpp

```
int main() {
    cout << "===== Tree and Heap Data Structures in C++ =====\n" << endl;

    // Run examples
    treeExample();
    heapExample();
    stlExamples();
    applicationExamples();

    return 0;
}
```

Key Points to Remember:

1. **Trees** are hierarchical structures with parent-child relationships
2. **Heaps** are complete binary trees satisfying heap property
3. **Max Heap**: Root is maximum element; **Min Heap**: Root is minimum element
4. **Time Complexities**:

- Heap insert/extract: O(log n)
- Heap build: O(n)
- Heap sort: O(n log n)

5. **STL provides**:
   - priority_queue for heap
   - make_heap, push_heap, pop_heap algorithms

Common Use Cases:

- **Heap**: Priority queues, scheduling, heap sort, graph algorithms
- **Tree**: File systems, databases, expression parsing, hierarchical data