READ ME file – Ex4

Daniel Kuris

danielkuris6@gmail.com

Tree Implementation in C++

Overview

This project implements a generic k-ary tree in C++, where each node can have up to k children. The tree supports various traversal methods, including pre-order, post-order, in-order, breadth-first search (BFS), depth-first search (DFS), and heap traversal. The code is templated to support any data type for the tree nodes.

Class Hierarchy

Tree Class

The Tree class is the main class that represents the k-ary tree. It contains a nested Node struct representing each node in the tree, along with the following members:

- root: A shared pointer to the root node of the tree.
- k: An integer representing the maximum number of children each node can have.

The Tree class provides the following public methods:

- add_root(const T& value): Adds a root node with the given value.
- add_sub_node(const T& parent_value, const T& child_value): Adds a child node with the given value to the node with the specified parent value.

The Tree class also defines the following nested iterator classes:

- PreOrderIterator
- PostOrderIterator
- InOrderIterator
- BFSIterator
- DFSIterator
- HeapIterator

Each iterator class implements the necessary operators for traversal.

Node Struct

The Node struct represents each node in the tree and contains the following members:

- value: The value stored in the node.
- children: A vector of shared pointers to the node's children.

Libraries Used

The following standard libraries are used in this implementation:

- <vector>: Used for storing children of each node.
- <memory>: Used for managing dynamic memory with smart pointers.
- <queue>: Used for implementing BFS traversal.
- <stack>: Used for implementing DFS, pre-order, post-order, and in-order traversals.
- <algorithm>: Used for heap operations.
- <iterator>: Used for defining iterator tags and supporting iterator operations.
- <functional>: Used for custom comparison operations in heap.
- <stdexcept>: Used for handling exceptions.

Iterator Implementations

PreOrderIterator

The PreOrderIterator class traverses the tree in pre-order (root, left, right) using a stack.

- **Data Structure Used**: std::stack<std::shared_ptr<Node>>
- **Algorithm**: Start with the root node. Push the current node's children onto the stack in reverse order, then pop the stack to get the next node.

PostOrderIterator

The PostOrderIterator class traverses the tree in post-order (left, right, root) using a stack.

- **Data Structure Used**: std::stack<std::shared_ptr<Node>>
- **Algorithm**: Use a stack to store nodes. Push children onto the stack if they have not been visited, then pop and visit nodes in post-order sequence.

InOrderIterator

The InOrderIterator class traverses the tree in in-order (left, root, right) using a stack.

- **Data Structure Used**: std::stack<std::shared_ptr<Node>>
- **Algorithm**: Use a stack to store nodes. Traverse the leftmost path and push nodes onto the stack. Visit nodes, then traverse the right path.

BFSIterator

The BFSIterator class traverses the tree in breadth-first order using a queue.

- **Data Structure Used**: std::queue<std::shared_ptr<Node>>
- **Algorithm**: Use a queue to store nodes at each level. Dequeue nodes, visit them, then enqueue their children.

DFSIterator

The DFSIterator class traverses the tree in depth-first order using a stack.

- **Data Structure Used**: std::stack<std::shared_ptr<Node>>
- **Algorithm**: Use a stack to store nodes. Push the current node's children onto the stack, then pop the stack to get the next node.

HeapIterator

The HeapIterator class traverses the tree as a heap (a specialized tree-based data structure) using a vector.

- **Data Structure Used**: std::vector<T>
- **Algorithm**: Convert the tree to a heap using std::make_heap and traverse the elements in the heap.

Handling Different Values of k

Binary Tree (k = 2)

When k = 2, the tree behaves as a binary tree, where each node can have at most two children. The left child is stored at index 0 of the children vector, and the right child is stored at index 1. This structure is inherently supported by the Tree class, and all iterator implementations work seamlessly with this configuration.

k-ary Tree (k > 2)

When k > 2, the tree becomes a k-ary tree, where each node can have up to k children. The children are stored in the children vector from index 0 to k-1. The Tree class and iterator implementations are designed to handle this configuration by iterating over all possible child indices. The traversal algorithms and heap operations remain consistent regardless of the value of k.