Algorithms

Karl Solomon

January 2, 2025

Contents

Part I
Data Structures

1 Queue/FIFO

2 Stack/LIFO

3 Heap/Priority Queue

4 Tree/Graph

4.1 Binary Tree

- Check if two BST are same

```
bool isSame(Node* root1, Node* root2) {
    if (!root1 && !root2) return true;
    if (!root1 || !root2) return false;
    return root1->data == root2->data && isSame(root1->left, root2
       ->left) && isSame(root1->right, root2->right);
}
```

- Insert Element

```
void insertNode(Node* root, int data) {
    if (root == nullptr) {
        root = new Node(data);
    } else {
        if (data < root->data) {
            insertNode(root->left, data);
        } else {
            insertNode(root->right, data);
        }
    }
}
```

- Delete Element

```
Node* deleteBSTNode(Node* root, int data) {
    if (root == nullptr) return root;
    if (root->data == data) {
        // Only 1 node (root)
        if (root->left == nullptr && root->right == nullptr) {
            delete root;
            root = nullptr;
        }
        // Only 1 child (right)
        else if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            root = temp;
        }
        // Only 1 child (left)
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            root = temp;
        }
        // Both children exist. Take either min from right sub-
           tree (implemented here) or max from left sub-tree and
           set
```

```cpp
        // to deleted node's position
        else {
            Node* temp = root->right;
            while (temp->left != nullptr) {
                temp = temp->left;
            }
            root->data = temp->data;
            root->right = deleteBSTNode(root->right, temp->data);
        }
    }
    return root;
}
```

- Lowest Common Ancestor

```cpp
Node* lowestCommonAncestor(Node* root, int data1, int data2) {
    // Base case
    if (root == nullptr) {
        return nullptr;
    }

    // If both data1 and data2 are smaller than root, then LCA
       lies in left subtree
    if (data1 < root->data && data2 < root->data) {
        return lowestCommonAncestor(root->left, data1, data2);
    }

    // If both data1 and data2 are greater than root, then LCA
       lies in right subtree
    if (data1 > root->data && data2 > root->data) {
        return lowestCommonAncestor(root->right, data1, data2);
    }

    // If one data is on the left and the other is on the right,
       then root is the LCA
    return root;
}
```

- Determine if tree is BST

```cpp
bool isBST(Node* root, int min, int max) {
    if (root == nullptr) return true;
    if (root->data < min || root->data > max) return false;
    return isBST(root->left, min, root->data - 1) && isBST(root->
       right, root->data + 1, max);
}

bool isBST(Node* root) {
    bool res = true;
    if (root == nullptr) {
        return res;
    }
    if (root->left == nullptr && root->right == nullptr) {
        return res;
    }
    return isBST(root, INT_MIN, INT_MAX);
}
```

- Traversal
  - Level-Order

```cpp
/*
 * - We use a 'std::queue<Node*>' to keep track of nodes to visit.
 * - We start by pushing the root node into the queue.
 * - While the queue is not empty, we:
 *   - Dequeue the front node, process it by adding its data to the result queue.
 *   - Enqueue its left and right children if they exist.
 * - This process continues until all nodes have been visited, resulting in a level-order traversal of the BST.
 */

std::queue<int> traverseBSTLevelOrder(Node* root) {
    std::queue<int> result;
    if (root == nullptr) {
        return result;
    }

    std::queue<Node*> nodeQueue;
    nodeQueue.push(root);

    while (!nodeQueue.empty()) {
        Node* current = nodeQueue.front();
        nodeQueue.pop();
        result.push(current->data);

        if (current->left != nullptr) {
            nodeQueue.push(current->left);
        }
        if (current->right != nullptr) {
            nodeQueue.push(current->right);
        }
    }

    return result;
}
```

– In-order: Left subtree, then node, then right subtree

```cpp
std::vector<int> bstTraverseInOrder(Node* root) {
    std::vector<int> res;
    if (root == nullptr) return res;
    res = bstTraverseInOrder(root->left);
    res.push_back(root->data);
    std::vector<int> right = bstTraverseInOrder(root->right);
    res.insert(res.end(), right.begin(), right.end());
}
```

– PreOrder: Node, then left subtree, then right subtree

– PostOrder: Left subtree, then right subtree, then Node

4.2 Graphs

Storing Graphs:

- Adjacency Matrix: 2D array NxN. [n,m] = nonZeroNum means there is an edge from node n to node m. All nodes should not include edges to self. Edges can be weighted (to suggest things like cost/distance). In directional-graphs you can have [n,m] = nonZero, but [m,n] = 0. Cons: O(N2̂) memory. This is especially bad for sparse graphs.

- Adjacency List: Store edges as a list of neighbor nodes. Each node contains a list of neighbors. It is possible to represent weights here with a list of std::pair<Node*, int>.

Graph Algorithms:

- Bellman-Ford

```cpp
#include <iostream>
#include <limits>
#include <vector>
/*
 * - We define a 'Graph' class with a list of 'Edge' structures, each containing a source, destination, and weight.
 *   - The 'bellmanFord' function initializes distances and relaxes all edges |V| - 1 times, where |V| is the number of
 * vertices.
 *   - It checks for negative-weight cycles by attempting one more relaxation.
 *   - If a negative-weight cycle is detected, it prints a message.
 *   - Otherwise, it prints the shortest distances from the start vertex to all other vertices.
 */

struct Edge {
    int source, destination, weight;
};

class Graph {
 public:
    int vertices;
    std::vector<Edge> edges;

    Graph(int v) : vertices(v) {}

    void addEdge(int source, int destination, int weight) { edges.push_back({source, destination, weight}); }
    // The Bellman-Ford algorithm, is used to find the shortest paths from a single source vertex to all other vertices
    // in a graph. The algorithm can handle graphs with negative weight edges:
    void bellmanFord(int start) {
        std::vector<int> distance(vertices, std::numeric_limits<int>::max());
        distance[start] = 0;

        // Relax edges |V| - 1 times
        for (int i = 0; i < vertices - 1; ++i) {
            for (const auto& edge : edges) {
                if (distance[edge.source] != std::numeric_limits<int>::max() &&
                    distance[edge.source] + edge.weight < distance[edge.destination]) {
                    distance[edge.destination] = distance[edge.source] + edge.weight;
                }
            }
        }

        // Check for negative-weight cycles
        for (const auto& edge : edges) {
            if (distance[edge.source] != std::numeric_limits<int>::max() &&
```

```cpp
                distance[edge.source] + edge.weight < distance[edge.destination]) {
                std::cout << "Graph contains a negative-weight cycle" << std::endl;
                return;
            }
        }

        // Print the distances
        for (int i = 0; i < vertices; ++i) {
            std::cout << "Distance from vertex " << start << " to vertex " << i << " is " << distance[i] << std::endl;
        }
    }
};

int main() {
    Graph g(5);
    g.addEdge(0, 1, -1);
    g.addEdge(0, 2, 4);
    g.addEdge(1, 2, 3);
    g.addEdge(1, 3, 2);
    g.addEdge(1, 4, 2);
    g.addEdge(3, 2, 5);
    g.addEdge(3, 1, 1);
    g.addEdge(4, 3, -3);

    g.bellmanFord(0);

    return 0;
}
```

- Dijkstra's

```cpp
#include <iostream>
#include <limits>
#include <queue>
#include <vector>

using namespace std;

typedef pair<int, int> Edge;  // (weight, vertex)
/*
 * In this program:
 *
 * - We define a 'Graph' class with an adjacency list to store edges as pairs of (weight, vertex).
 *   - The 'dijkstra' function initializes distances and uses a priority queue to explore the shortest paths.
 *   - It updates the shortest path estimates and pushes them into the priority queue.
 *   - Finally, it prints the shortest distances from the start vertex to all other vertices.
 *
 */

class Graph {
 public:
    int vertices;
    vector<vector<Edge>> adjList;

    Graph(int v) : vertices(v), adjList(v) {}
```

```cpp

    void addEdge(int source, int destination, int weight) {
        adjList[source].push_back(make_pair(weight, destination));
        adjList[destination].push_back(make_pair(weight, source)); // For undirected graph
    }
    // Dijkstra's algorithm, is used to find the shortest paths from a single source vertex to all other vertices
    // in a graph with non-negative edge weights:
    void dijkstra(int start) {
        vector<int> distance(vertices, numeric_limits<int>::max());
        priority_queue<Edge, vector<Edge>, greater<Edge>> pq;

        distance[start] = 0;
        pq.push(make_pair(0, start));

        while (!pq.empty()) {
            int currentVertex = pq.top().second;
            pq.pop();

            for (const auto& edge : adjList[currentVertex]) {
                int weight = edge.first;
                int neighbor = edge.second;

                if (distance[currentVertex] + weight < distance[neighbor]) {
                    distance[neighbor] = distance[currentVertex] + weight;
                    pq.push(make_pair(distance[neighbor], neighbor));
                }
            }
        }

        // Print the distances
        for (int i = 0; i < vertices; ++i) {
            cout << "Distance from vertex " << start << " to vertex " << i << " is " << distance[i] << endl;
        }
    }
};

int main() {
    Graph g(5);
    g.addEdge(0, 1, 10);
    g.addEdge(0, 4, 5);
    g.addEdge(1, 2, 1);
    g.addEdge(1, 4, 2);
    g.addEdge(2, 3, 4);
    g.addEdge(3, 4, 9);
    g.addEdge(3, 0, 7);

    g.dijkstra(0);

    return 0;
}
```

- Ford-Fulkerson

- Kruskals's

- `Nearest neighbor`

- `Prim's`

- `DFS`

- `BFS`

5 Lists

5.1 Singly Linked List

5.2 Doubly Linked List

Part II
Types of Algorithms

6 Greedy

7 DP

8 DFS

9 BFS

10 Sorting

11 Network Flow

11.1 Dijkstra's