TREE AND GRAPH DATA STRUCTURE REPORT

From 21/05/2024 to 24/05/2024

By

GROUP 5 COE-1

Submitted in fulfillment of the requirement for

DIPLOMA

IN

COMPUTER ENGINEERING

DAR-ES-SALAAM INSTITUTE OF TECHNOLOGY

P.O.BOX 11007 DAR ES SALAAM

TANZANIA

ACADEMIC YEAR 2022/2023

GROUP PARTICIPANTS	REG. NO
HAMDAN MWIRU	220222484939
EDMOND JAMES	220222437119
NSAJIGWA MWANGAMA	220222424489

ENG. Yustin M

…….

DATE OF SUBMITTION 24/05/2024

Book Report

Tree and Graph Data Structure Description

Abstract

Data structures are a fundamental component of computer science, allowing for the efficient storage and manipulation of data. Two of the most critical and widely-used types of data structures are trees and graphs. This report delves into the various types and characteristics of these data structures, their applications, and their respective algorithms.

Background Information on Data Structures

Introduction to Data Structures

Data structures are a crucial aspect of computer science, providing the means to manage and organize data efficiently. They are the foundation upon which algorithms

operate, determining how data is stored, accessed, and manipulated. A good understanding of data structures allows developers to write efficient and optimized code, which is essential for solving complex problems in various domains such as databases, networking, operating systems, and artificial intelligence.

Types of Data Structures

Data structures can be broadly categorized into two types:

1. **Primitive Data Structures**

- **Integers**: Whole numbers without a fractional part.
- **Floats**: Numbers with a fractional part.
- **Characters**: Single letters or symbols.
- **Booleans**: True or false values.

2. **Non-Primitive Data Structures**

- **Linear Data Structures**: Data elements are arranged in a sequential manner.
- **Arrays**: Fixed-size structures that hold elements of the same data type.
- **Linked Lists**: Consist of nodes where each node contains data and a reference to the next node.
- **Stacks**: Follow Last In, First Out (LIFO) principle.
- **Queues**: Follow First In, First Out (FIFO) principle.
- **Non-Linear Data Structures**: Data elements are not arranged in a sequential manner.
- **Trees**: Hierarchical structures with a root node and subnodes forming a parent-child relationship.
- **Graphs**: Consist of vertices (nodes) and edges (connections between nodes).

Importance of Data Structures

Data structures are essential for several reasons:

1. **Efficiency**: Proper data structures enable efficient data manipulation, improving the performance of algorithms.

2. **Organization**: They provide a systematic way of organizing data to facilitate easier access and modification.
3. **Scalability**: Well-chosen data structures can handle growing amounts of data without significant performance degradation.
4. **Reusability**: Data structures are often implemented as reusable components, making it easier to solve new problems using established methods.

Tree Data Structures

A tree is a hierarchical data structure that simulates a tree-like model of a hierarchy with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

Types of Trees

1. **Binary Tree**

- Each node has at most two children referred to as the left child and the right child.
- **Applications**: Expression parsing, Huffman coding trees.

2. **Binary Search Tree (BST)**

- A binary tree with the property that for each node, all elements in the left subtree are less, and all elements in the right subtree are greater.
- **Applications**: Searching and sorting.

3. **AVL Tree**

- A self-balancing binary search tree where the difference between heights of left and right subtrees cannot be more than one for all nodes.
- **Applications**: Databases and memory management.

4. **Red-Black Tree**

- A self-balancing binary search tree where each node has an extra bit for denoting the color of the node, either red or black.
- **Applications**: Balancing data in search trees.

5. **Segment Tree**

- A binary tree used for storing intervals or segments. It allows querying which of the stored segments contain a given point.
- **Applications**: Range queries like sum, minimum, maximum, etc.

6. **Fenwick Tree (Binary Indexed Tree)**

- A data structure that provides efficient methods for calculation and manipulation of the prefix sums of a table of values.
- **Applications**: Cumulative frequency tables.

7. **Trie (Prefix Tree)**

- A type of search tree used to store a dynamic set of strings where the keys are usually strings.
- **Applications**: Autocomplete, spell checker, IP routing.

8. **Suffix Tree**

- A compressed trie of all the suffixes of a given string.
- **Applications**: String operations like substring search, longest repeated substring.

9. **B-Tree**

- A self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.
- **Applications**: Database and file systems.

10. **N-ary Tree**

- A tree in which a node can have at most N children.
- **Applications**: Directory structures, organizational structures.

Tree Algorithms

Tree Traversals

1. **In-order Traversal**
2. **Pre-order Traversal**
3. **Post-order Traversal**
4. **Level-order Traversal**

Binary Search Tree Operations

1. **Insertion**

C++ code snippet

```
void insert(int val) {
    root = insert(root, val);
}
TreeNode* insert(TreeNode* node, int val) {
    if (node == nullptr) return new TreeNode(val);
    if (val < node->val)
        node->left = insert(node->left, val);
    else if (val > node->val)
        node->right = insert(node->right, val);
    return node;
}
```

2. **Deletion**

```
void deleteNode(int val) {
    root = deleteNode(root, val);
}
```

```
TreeNode* deleteNode(TreeNode* root, int key) {
    if (root == nullptr) return root;

    if (key < root->val)
        root->left = deleteNode(root->left, key);
    else if (key > root->val)
        root->right = deleteNode(root->right, key);
    else {
        if (root->left == nullptr) {
            TreeNode* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            TreeNode* temp = root->left;
            delete root;
            return temp;
        }

        TreeNode* temp = minValueNode(root->right);
        root->val = temp->val;
        root->right = deleteNode(root->right, temp->val);
    }
    return root;
}
```

```
TreeNode* minValueNode(TreeNode* node) {
    TreeNode* current = node;
    while (current && current->left != nullptr)
        current = current->left;
    return current;
}
```

3. **Search**

C++ code snippet

```
    TreeNode* search(int val) {
        return search(root, val);
    }

    TreeNode* search(TreeNode* node, int val) {
        if (node == nullptr || node->val == val) return node
        if (val < node->val) return search(node->left, val);
        return search(node->right, val);
    }
```

Balancing Trees

1. **AVL Tree Rotations**

C++ code snippet

```
TreeNode* rightRotate(TreeNode* y) {
    TreeNode* x = y->left;
    TreeNode* T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    return x;
}
```

```
TreeNode* leftRotate(TreeNode* x) {
    TreeNode* y = x->right;
    TreeNode* T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}
```

2. **Red-Black Tree Rotations and Recoloring**

Segment Tree Operations

1. **Build**
2. **Update**
3. **Query**

Fenwick Tree Operations

1. **Update**
2. **Query**

Graph Data Structures

A graph is a collection of nodes (or vertices) and edges connecting pairs of nodes. Graphs are used to model pairwise relations between objects.

Types of Graphs

1. **Undirected Graph**

- A graph where edges have no direction.
- **Applications**: Social networks, geographic maps.

2. **Directed Graph (Digraph)**

- A graph where each edge has a direction.
- **Applications**: Web page linking, task scheduling.

3. **Weighted Graph**

- A graph where edges have weights or costs.
- **Applications**: Network routing, shortest path problems.

4. **Unweighted Graph**

- A graph where edges do not have weights.
- **Applications**: Basic connectivity and pathfinding.

5. **Cyclic Graph**

- A graph containing at least one cycle.
- **Applications**: Feedback systems, electrical circuits.

6. **Acyclic Graph**

- A graph with no cycles.
- **Applications**: Dependency resolution, course prerequisites.

7. **Directed Acyclic Graph (DAG)**

- A directed graph with no cycles.
- **Applications**: Task scheduling, version control systems.

8. **Bipartite Graph**

- A graph whose vertices can be divided into two disjoint sets such that no two graph vertices within the same set are adjacent.
- **Applications**: Job assignment, matching problems.

9. **Complete Graph**

- A graph in which there is an edge between every pair of vertices.
- **Applications**: Network topology.

10. **Sparse Graph**

- A graph with relatively few edges compared to the number of vertices.
- **Applications**: Social networks with limited connections.

11. **Dense Graph**

- A graph with a large number of edges.
- **Applications**: Highly connected networks.

Graph Algorithms

Traversal Algorithms

1. **Depth-First Search (DFS)**

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

class Graph {
    int V;
    list<int>* adj;

    void DFSUtil(int v, bool visited[]) {
        visited[v] = true;
        cout << v << " ";

        for (auto i = adj[v].begin(); i != adj[v].end(); ++i
            if (!visited[*i])
                DFSUtil(*i, visited);
    }
```

```
public:
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }

    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }

    void DFS(int v) {
        bool* visited = new bool[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;
```

```
        DFSUtil(v, visited);
    }
};

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

    cout << "Depth First Traversal starting from vertex 2:\n
    g.DFS(2);
```

```
    return 0;
}
```

2. **Breadth-First Search (BFS)**

C++ code snippet

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

class Graph {
    int V;
    list<int>* adj;

public:
    Graph(int V) {
        this->V = V;
        adj = new list<int>[V];
    }

    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }

    void BFS(int s) {
        bool* visited = new bool[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;

        queue<int> q;
        visited[s] = true;
        q.push(s);

        while (!q.empty()) {
            s = q.front();
            cout << s << " ";
```

```
            q.pop();

            for (auto i = adj[s].begin(); i != adj[s].end(); +
                if (!visited[*i]) {
                    visited[*i] = true;
                    q.push(*i);
                }
            }
        }
    }
};

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

    cout << "Breadth First Traversal starting from vertex 2:\n
    g.BFS(2);

    return 0;
}
```

Shortest Path Algorithms

1. **Dijkstra's Algorithm**

C++ code snippet

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

const int INF = numeric_limits<int>::max();

class Graph {
    int V;
    vector<pair<int, int>>* adj;

public:
    Graph(int V) {
        this->V = V;
        adj = new vector<pair<int, int>>[V];
    }

    void addEdge(int u, int v, int w) {
        adj[u].emplace_back(v, w);
        adj[v].emplace_back(u, w); // For undirected graph
    }

    void dijkstra(int src) {
        priority_queue<pair<int, int>, vector<pair<int, int>
        vector<int> dist(V, INF);

        pq.emplace(0, src);
        dist[src] = 0;

        while (!pq.empty()) {
            int u = pq.top().second;
```

```
            pq.pop();

            for (auto& neighbor : adj[u]) {
                int v
```

2. **Bellman-Ford Algorithm**
3. **Floyd-Warshall Algorithm**

Minimum Spanning Tree Algorithms

1. **Kruskal's Algorithm**
2. **Prim's Algorithm**

Network Flow Algorithms

1. **Ford-Fulkerson Algorithm**
2. **Edmonds-Karp Algorithm**

Topological Sorting

- Used in DAGs for ordering vertices.

Conclusion

Understanding the various types of trees and graphs and their corresponding algorithms is essential for solving a wide range of computational problems. Trees provide efficient structures for data management and hierarchical representation, while graphs excel in modeling complex relationships and networked data.

Trees and graphs are versatile data structures with various applications in computer science, from organizing hierarchical data to modeling networks. Understanding their properties, types, and basic operations is crucial for solving a wide range of computational problems. The provided C++ examples illustrate fundamental operations for both trees and graphs, demonstrating their practical implementation

References

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
- Sedgewick, R. (2002). *Algorithms in C++*. Addison-Wesley.

- Goodrich, M. T., & Tamassia, R. (2014). *Data Structures and Algorithms in C++*. Wiley.