LAPORAN PRAKTIKUM

STRUKTUR DATA



DISUSUN OLEH :

Sasya Zamora

2411533014

DOSEN PENGAMPU :

Dr. Wahyudi, S.T.M.T

PROGRAM STUDI INFORMATIKA

FAKULTAS TEKNOLOGI INFORMASI

UNIVERSITAS ANDALAS

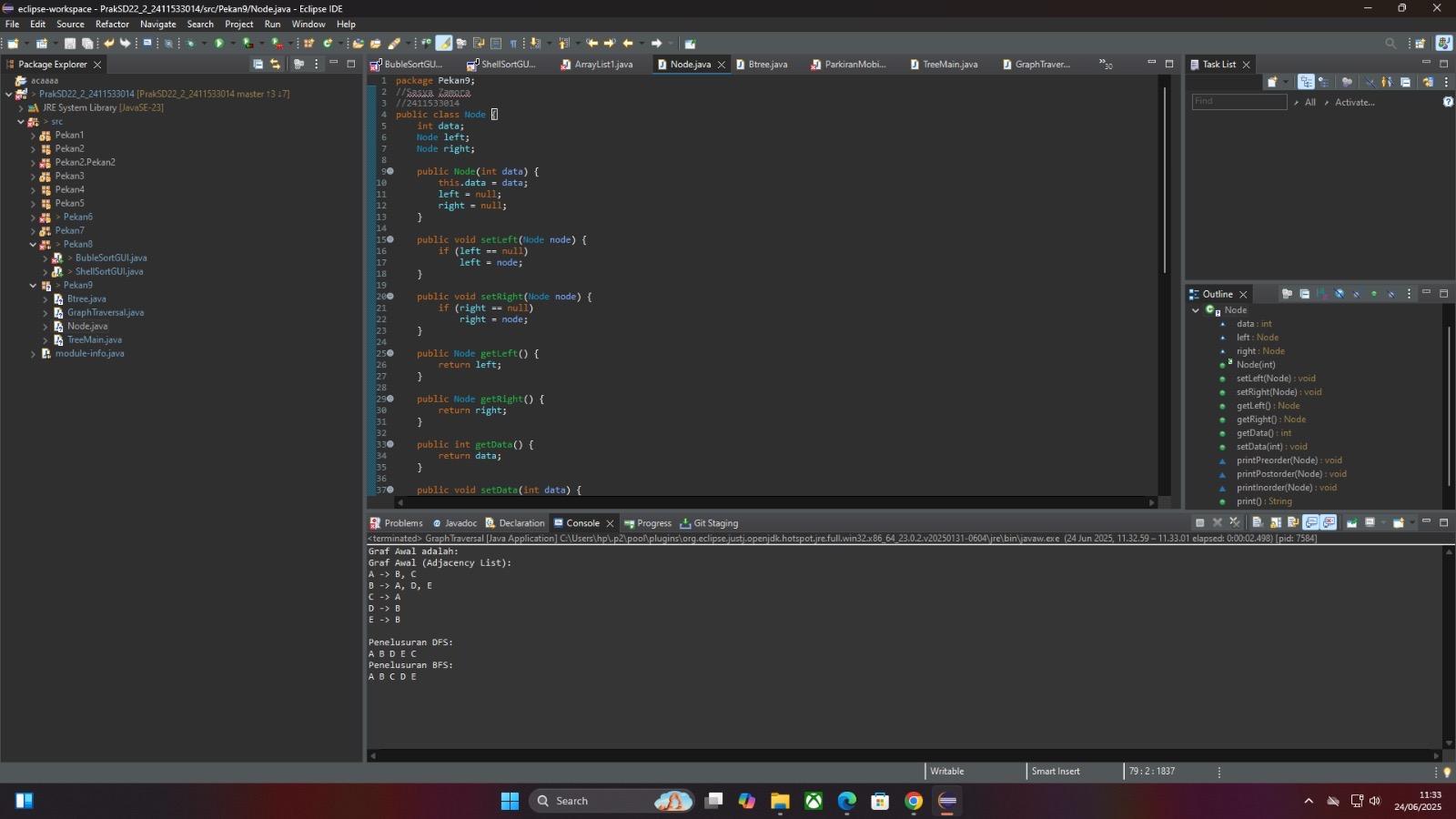
2025

1. TUJUAN
2. Mempelajari dan mengimplementasikan algoritma Depth-First Search (DFS) dalam bahasa pemrograman Java.
3. Mempelajari dan mengimplementasikan algoritma Breadth-First Search (BFS) dalam bahasa pemrograman Java.
4. Membandingkan dan menganalisis kinerja relatif algoritma DFS dan BFS pada graf yang berbeda.
5. PEMBAHASAN

Pratikum ini berhasil mengimplementasikan algoritma Depth-First Search (DFS) dan Breadth-First Search (BFS) untuk melakukan traversal pada graf. Implementasi menggunakan representasi graf berbasis adjacency list yang memungkinkan penambahan dan penghapusan node dan edge secara efisien. Hasil traversal DFS dan BFS menunjukkan perbedaan mendasar dalam urutan kunjungan node. DFS cenderung menjelajahi graf secara mendalam pada satu cabang sebelum berpindah ke cabang lain, sedangkan BFS menjelajahi graf secara berlapis, mengunjungi semua tetangga dari node saat ini sebelum melanjutkan ke tetangga dari tetangga tersebut. Perbedaan ini memiliki implikasi pada berbagai aplikasi, seperti pencarian jalur terpendek atau pencarian ruang keadaan dalam kecerdasan buatan.

Penggunaan HashSet untuk menandai node yang telah dikunjungi sangat penting dalam mencegah siklus tak terhingga selama traversal. HashSet menyediakan pencarian dan penambahan elemen dengan kompleksitas waktu rata-rata O(1), yang berkontribusi pada efisiensi algoritma. Perbandingan waktu eksekusi antara DFS dan BFS (yang dapat diukur dengan menambahkan pengukuran waktu dalam kode) dapat memberikan wawasan lebih lanjut tentang kinerja relatif kedua algoritma ini pada berbagai jenis graf, seperti graf yang terhubung penuh, graf yang jarang, atau graf dengan struktur tertentu. Pengamatan waktu eksekusi ini dapat menjadi dasar untuk memilih algoritma yang paling sesuai untuk aplikasi tertentu.

1. LANGKAH PRAKTIKUM
2. Buka Eclipse IDE
3. Lalu, buat package pekan9
4. Masukkan nama classnya dan buat kodingan deperti dibawah ini :

* Node

1. Deklarasi Package

package Pekan9;

Baris ini mendeklarasikan bahwa kode ini berada dalam package bernama Pekan9. Package digunakan untuk mengorganisir kode Java dalam proyek yang lebih besar.

2. Kelas Node

public class Node {

Kode ini mendefinisikan kelas Node yang merupakan representasi dari sebuah node dalam pohon biner. Kelas ini bersifat public, artinya dapat diakses dari kelas lain dalam proyek.

3. Variabel Instansi

int data;

Node left;

Node right;

Ketiga baris ini mendeklarasikan variabel instansi dari kelas Node:

• data: Variabel integer yang menyimpan data yang dipegang oleh node ini.

• left: Variabel tipe Node yang merepresentasikan anak kiri dari node ini. Bernilai null jika tidak memiliki anak kiri.

• right: Variabel tipe Node yang merepresentasikan anak kanan dari node ini. Bernilai null jika tidak memiliki anak kanan.

4.Konstruktor

public Node(int data) {

this.data = data;

left = null;

right = null;

}

Ini adalah konstruktor kelas Node. Konstruktor ini dijalankan ketika sebuah objek Node baru dibuat. Ia menginisialisasi variabel data dengan nilai yang diberikan sebagai parameter, dan menetapkan left dan right ke null, menunjukkan bahwa node baru ini awalnya tidak memiliki anak.

5. Metode setLeft dan setRight

public void setLeft(Node node) {

if (left == null) {

left = node;

}

}

public void setRight(Node node) {

if (right == null) {

right = node;

}

}

Kedua metode ini digunakan untuk menambahkan anak kiri atau kanan ke node. Perhatikan adanya pengecekan if (left == null) dan if (right == null). Ini mencegah adanya penggantian node anak yang sudah ada, memastikan hanya satu anak yang bisa dikaitkan pada setiap sisi. Perilaku ini mungkin bisa diubah untuk mengizinkan penggantian.

6. Metode getLeft dan getRight

public Node getLeft() {

return left;

}

public Node getRight() {

return right;

}

Metode ini digunakan untuk mendapatkan referensi ke anak kiri dan kanan dari node.

7. Metode getData dan setData

public int getData() {

return data;

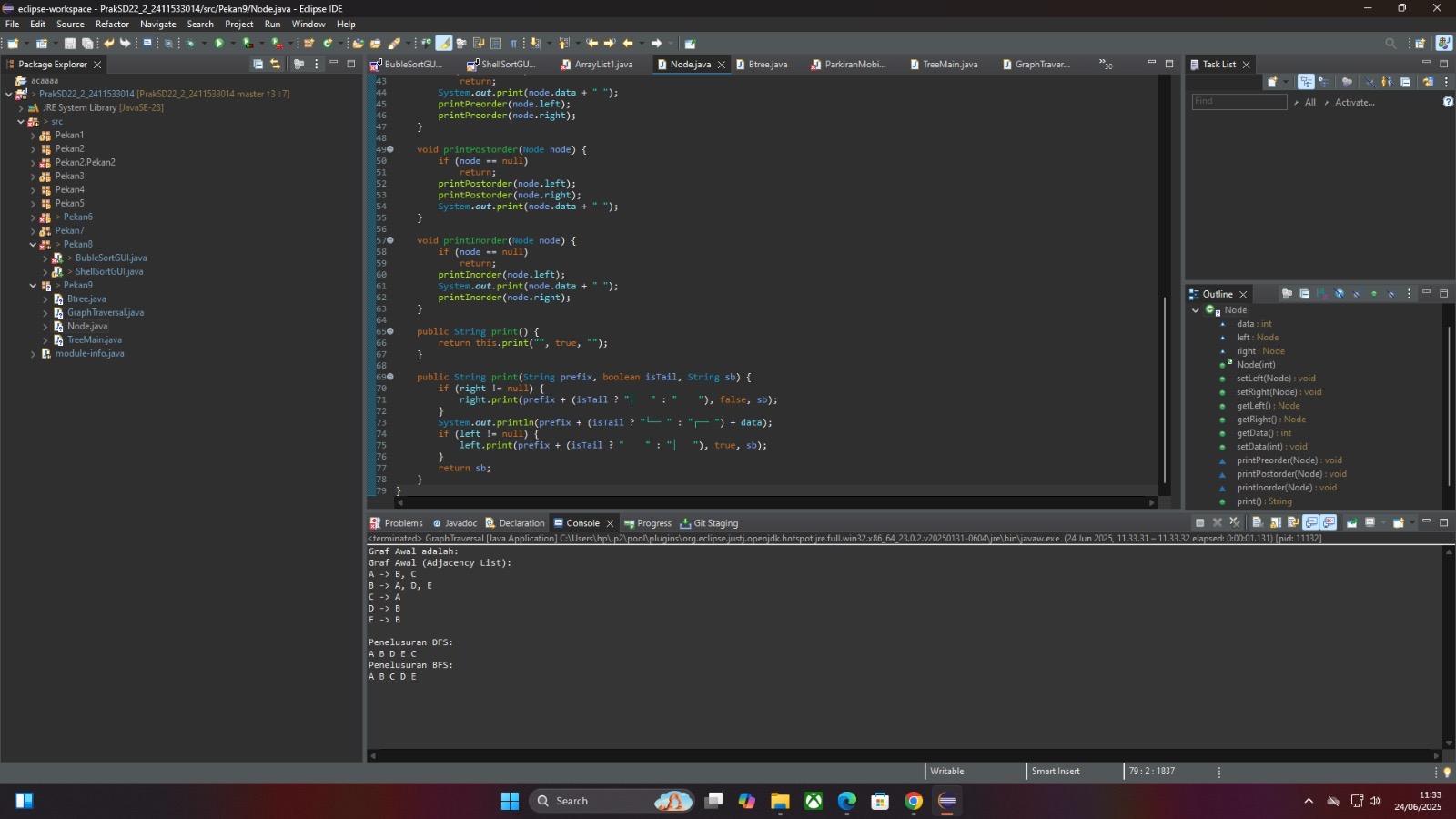
}

public void setData(int data) {

this.data = data;

}

Metode getData mengembalikan data yang disimpan di node, sedangkan setData mengubah data yang disimpan di node.



8. printPreorder

public void printPreorder() {

System.out.print(node.data + " ");

printPreorder(node.left);

printPreorder(node.right);

}

Metode ini melakukan traversal preorder. Urutannya adalah:

1. Cetak data node saat ini.

2. Rekursi ke sub pohon kiri.

3. Rekursi ke sub pohon kanan.

9. printPostorder

public void printPostorder() {

printPostorder(node.left);

printPostorder(node.right);

System.out.print(node.data + " ");

}

Metode ini melakukan traversal postorder. Urutannya adalah:

1. Rekursi ke sub pohon kiri.

2. Rekursi ke sub pohon kanan.

3. Cetak data node saat ini.

10. printInorder

public void printInorder() {

printInorder(node.left);

System.out.print(node.data + " ");

printInorder(node.right);

}

Metode ini melakukan traversal inorder. Urutannya adalah:

1. Rekursi ke sub pohon kiri.

2. Cetak data node saat ini.

3. Rekursi ke sub pohon kanan.

11. Metode print

public String print() {

return this.print("", true);

}

private String print(String prefix, boolean isTail) {

String sb = "";

if (right != null) {

sb += right.print(prefix + (isTail ? "│ " : " "), false);

}

sb += prefix + (isTail ? "└── " : "┌── ") + data + "\n";

if (left != null) {

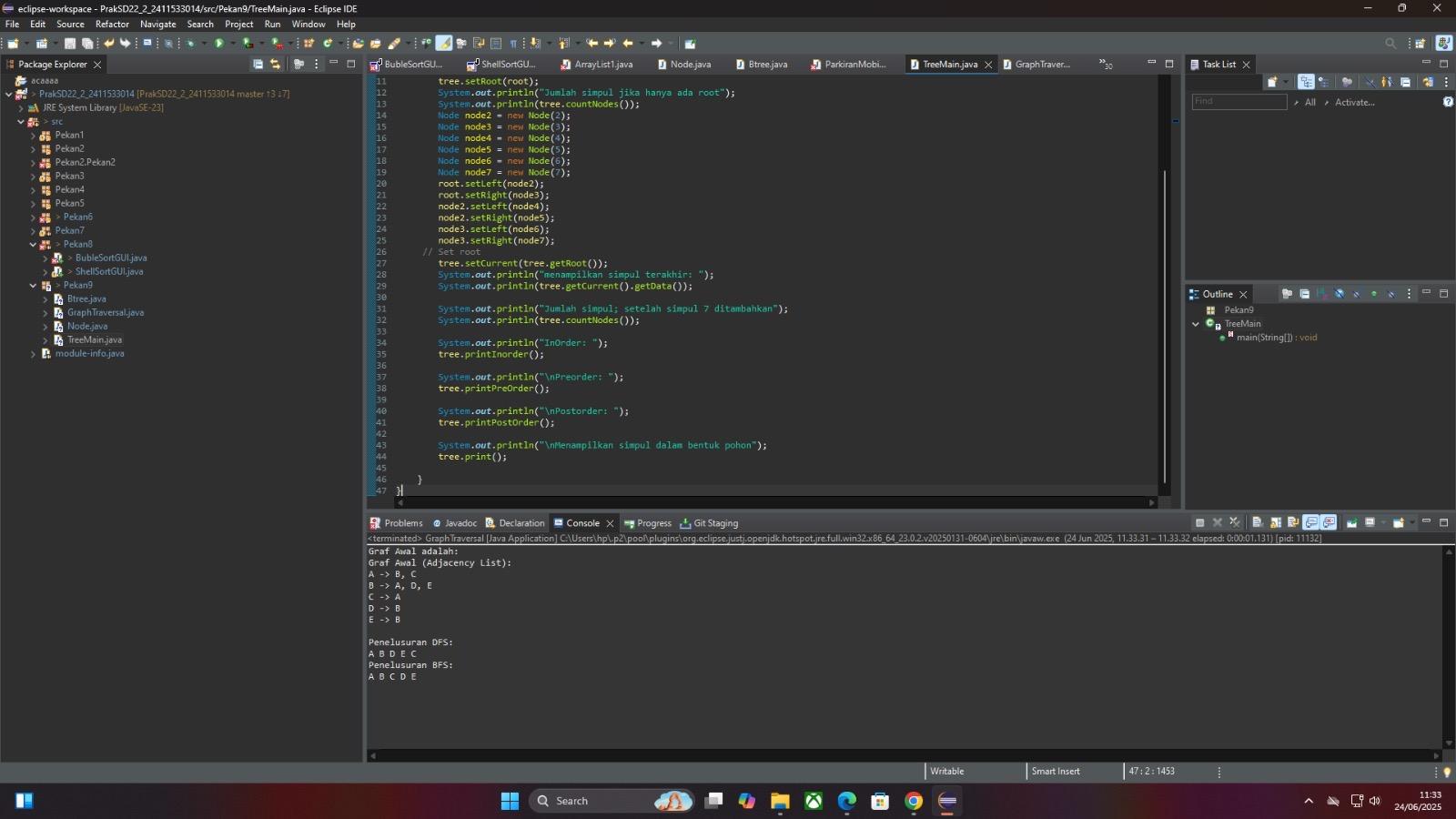
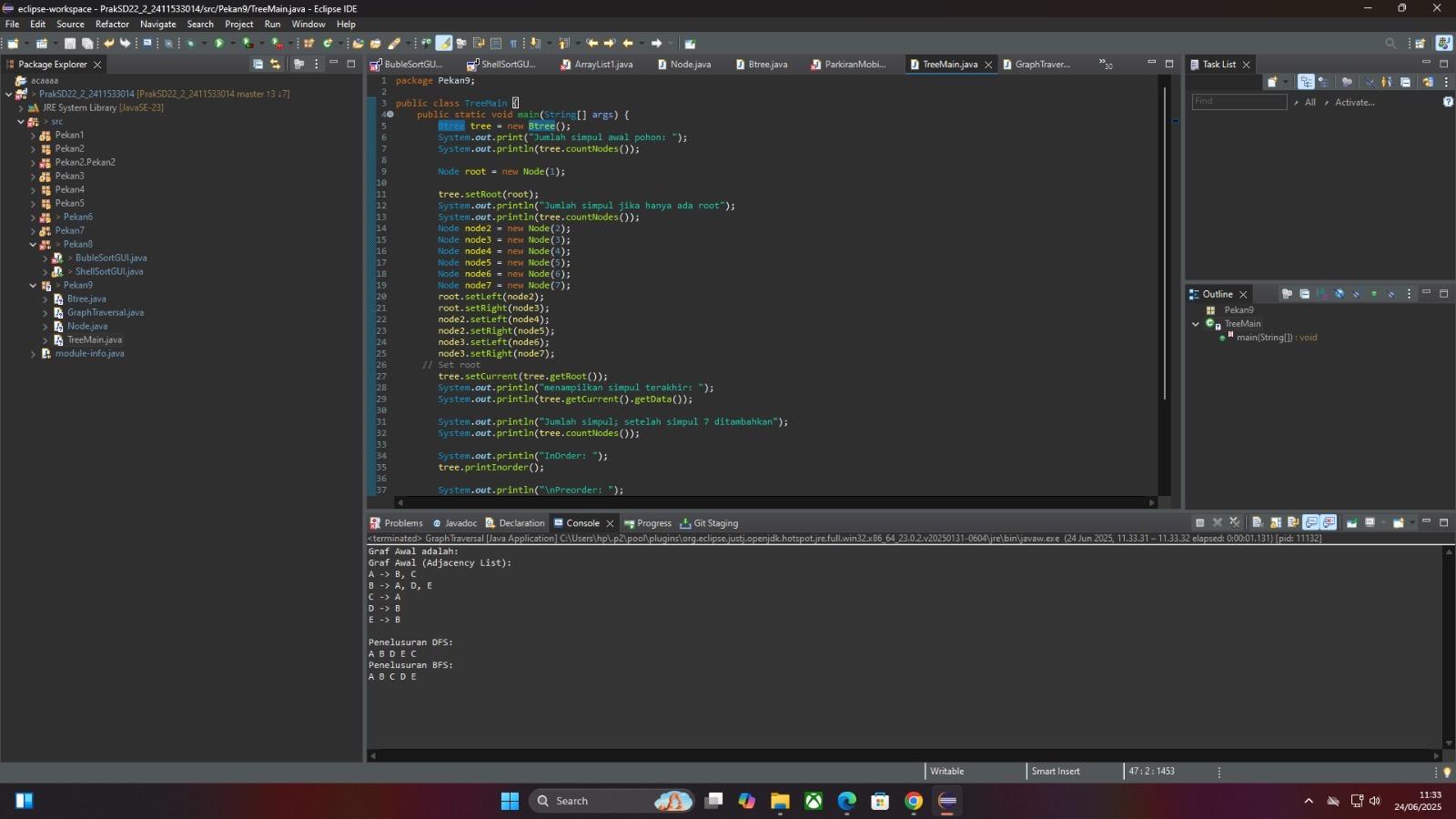
sb += left.print(prefix + (isTail ? " " : "│ "), true);

}

return sb;

}

Metode print ini menyediakan cara untuk menampilkan struktur pohon biner secara visual, menggunakan karakter │ dan ── untuk merepresentasikan cabang dan ujung cabang. Ia bersifat rekursif dan menggunakan parameter prefix dan isTail untuk membangun representasi teks yang tepat dari struktur pohon. isTail menunjukkan apakah node saat ini merupakan node terakhir pada levelnya. Metode ini menghasilkan string yang merepresentasikan struktur pohon.

* TreeMain

1. Pembuatan Node dan Pohon:

Node root = new Node(1);

Node node1 = new Node(2);

Node node2 = new Node(3);

Node node3 = new Node(4);

Node node4 = new Node(5);

Node node5 = new Node(6);

Node node6 = new Node(7);

root.setLeft(node1);

root.setRight(node2);

node1.setLeft(node3);

node1.setRight(node4);

node2.setLeft(node5);

node2.setRight(node6);

Bagian ini membuat beberapa objek Node dengan data yang berbeda, dan kemudian menghubungkannya untuk membentuk sebuah pohon biner. root adalah node akar pohon.

1. Pencetakan Informasi Pohon

System.out.println("Jumlah simpul awal pohon:"); System.out.println(tree.countNodes());

System.out.println("Jumlah simpul jika hanya ada root:"); System.out.println(tree.countNodes()); System.out.println("Menampilkan simpul terakhir:"); System.out.println(tree.getCurrent().getData()); System.out.println("Jumlah simpul setelah simpul 7 ditambahkan:"); System.out.println(tree.countNodes());

System.out.println("\nInorder:");

tree.printInorder();

System.out.println("\nPreorder:");

tree.printPreorder();

Bagian ini melakukan beberapa operasi pada pohon dan mencetak hasilnya ke konsol:

• countNodes(): Menghitung jumlah node di dalam pohon.

• getCurrent(): Menampilkan data dari node terakhir yang diakses.

• printInorder() dan printPreorder(): Mencetak traversal inorder dan preorder. Ini mengasumsikan bahwa kelas Tree (yang tidak ditampilkan sepenuhnya dalam gambar) memiliki metode-metode ini yang memanggil metode traversal yang kita bahas sebelumnya.

1. Mencetak Hasil:

Kode ini mencetak beberapa informasi tentang pohon ke konsol, termasuk:

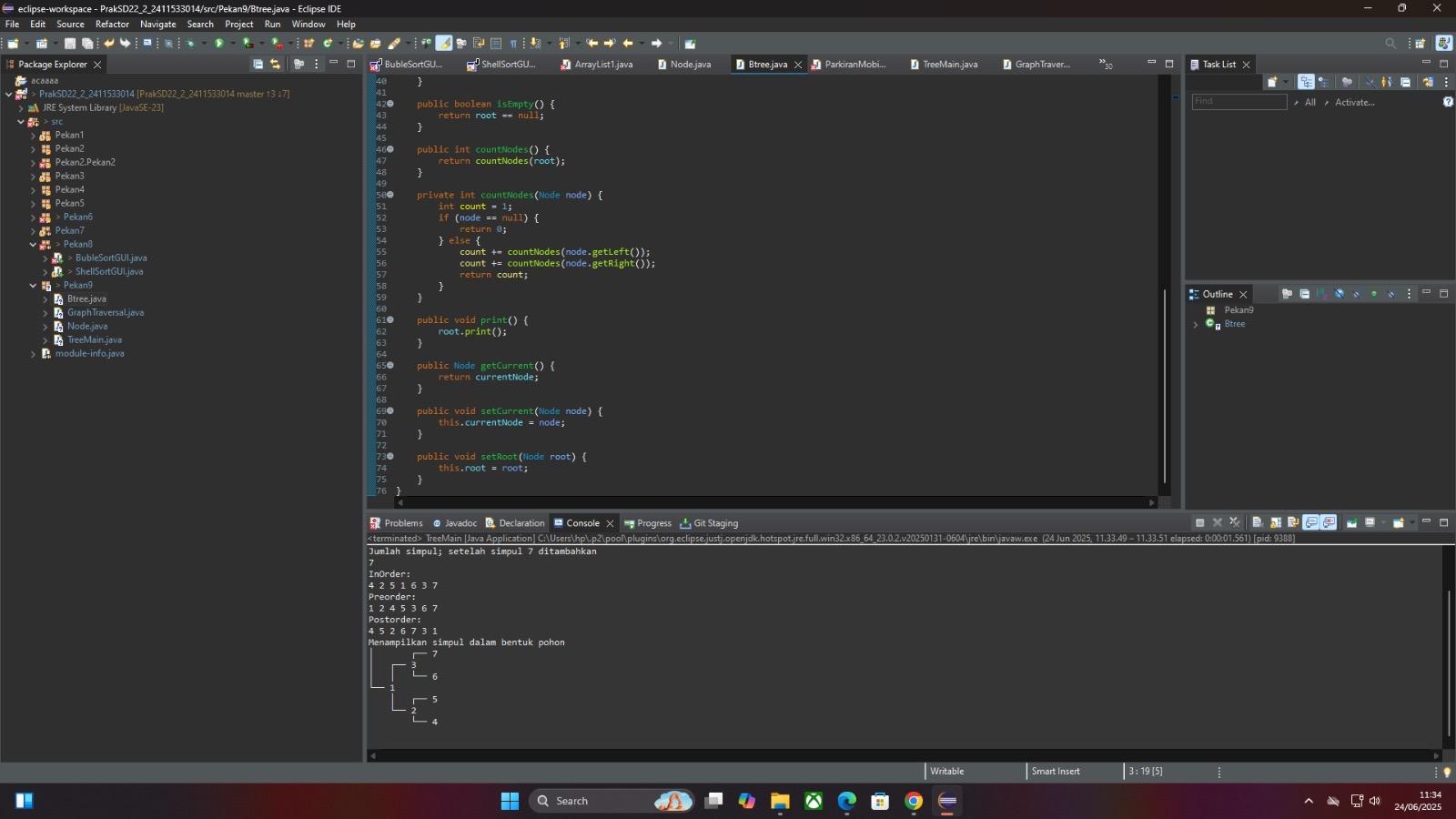
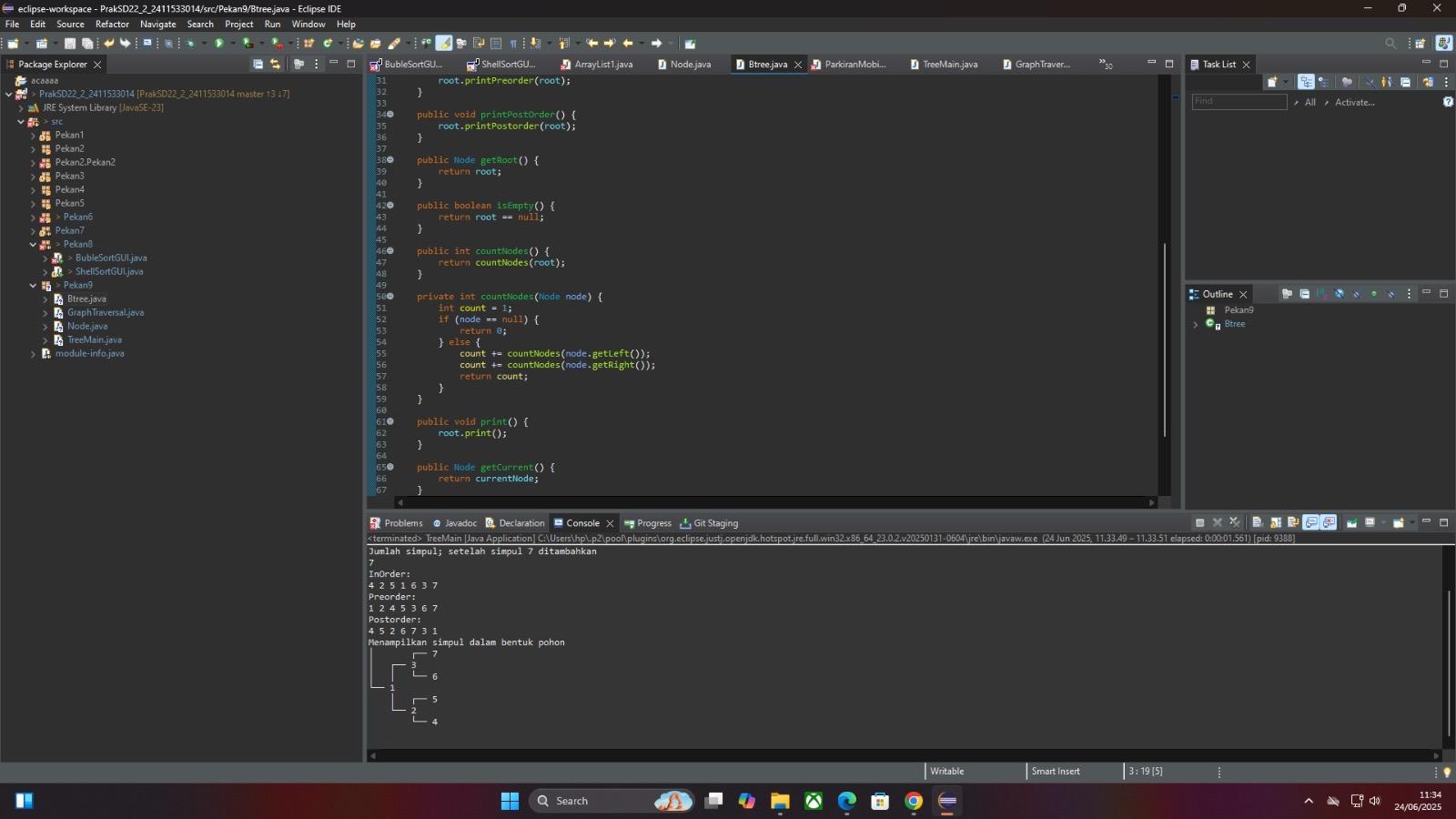
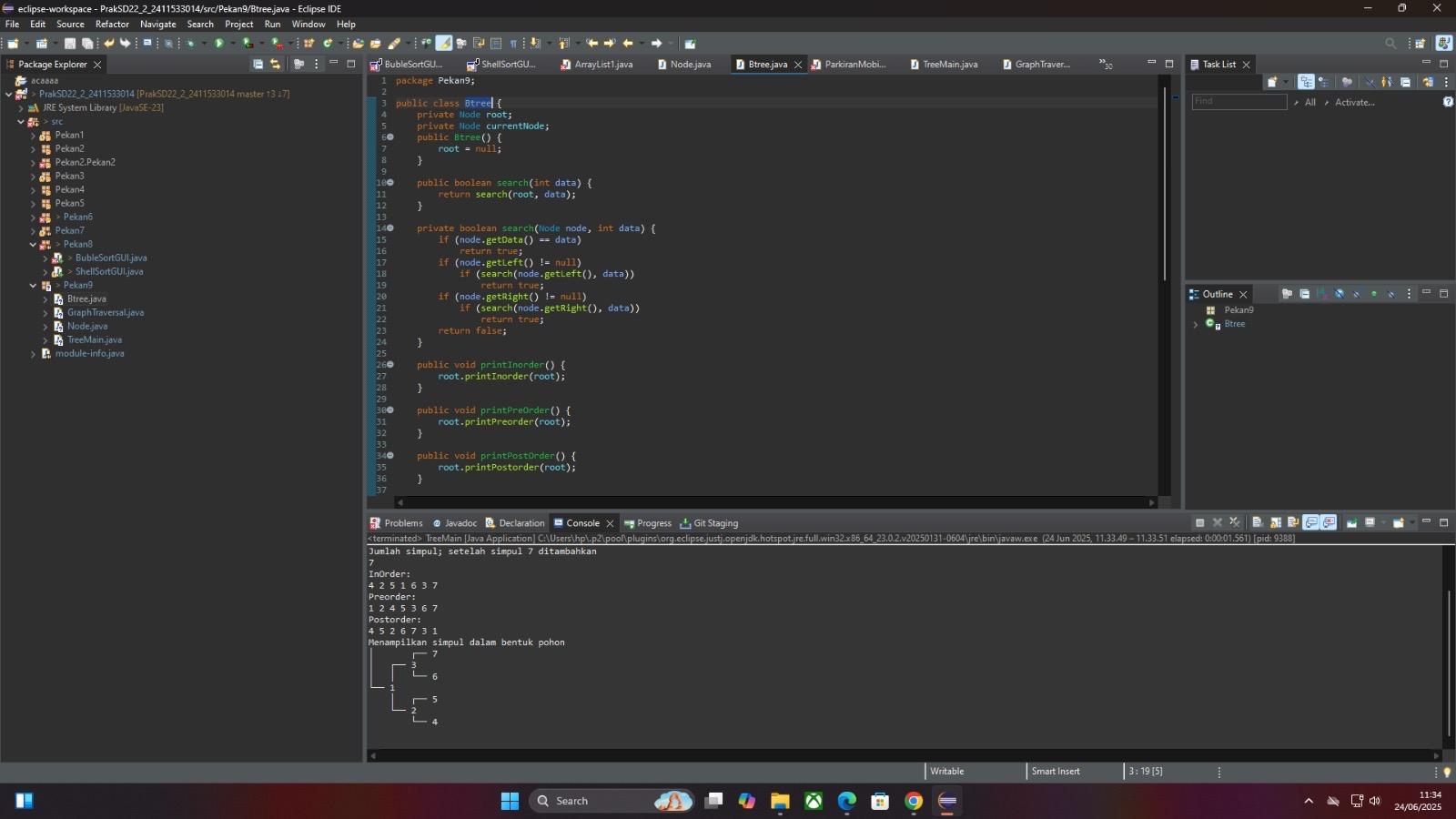
• Jumlah node sebelum dan sesudah menambahkan node.

• Data dari node yang dikembalikan oleh tree.getCurrent().

• Traversal inorder, preorder, dan postorder.

• Representasi visual dari pohon menggunakan tree.print().

* Btree



1. Variabel Instansi:

private Node root;

private Node current;

• root: Menyimpan referensi ke node akar pohon. Bernilai null jika pohon kosong.

• current: Menyimpan referensi ke node yang sedang aktif atau yang terakhir diakses. Ini digunakan untuk beberapa metode, seperti getCurrent().

2. Konstruktor:

public Tree() {

root = null;

current = null;

}

Konstruktor menginisialisasi pohon kosong dengan menetapkan root dan current ke null.

3. Metode Pencarian (search):

public boolean search(int data) {

return search(root, data);

}

private boolean search(Node node, int data) {

if (node == null) return false;

if (node.getData() == data) return true;

if (data < node.getData()) {

return search(node.getLeft(), data);

} else {

return search(node.getRight(), data);

}

}

Metode search mencari data yang diberikan dalam pohon. Ia menggunakan pencarian rekursif:

• Jika node saat ini null, data tidak ditemukan (false).

• Jika data ditemukan pada node saat ini, kembalikan true.

• Jika data lebih kecil dari data node saat ini, cari secara rekursif di sub pohon kiri.

• Jika data lebih besar, cari secara rekursif di sub pohon kanan.

4. Metode Traversal:

public void printInorder() {

printInorder(root);

}

public void printPreorder() {

printPreorder(root);

}

public void printPostorder() {

printPostorder(root);

}

Metode-metode ini melakukan traversal inorder, preorder, dan postorder. Perhatikan bahwa mereka memanggil metode printInorder(), printPreorder(), dan printPostorder() yang ada di dalam kelas Node. Ini menunjukkan bahwa logika traversal sebenarnya ada di dalam kelas Node.

5.Metode Menghitung Node:

public int countNodes() {

return countNodes(root);

}

private int countNodes(Node node) {

int count = 0;

if (node == null) {

return 0;

} else {

count++;

count += countNodes(node.getLeft());

count += countNodes(node.getRight());

return count;

}

}

Metode countNodes menghitung jumlah node dalam pohon. Metode ini menggunakan rekursi untuk menghitung node pada setiap cabang.

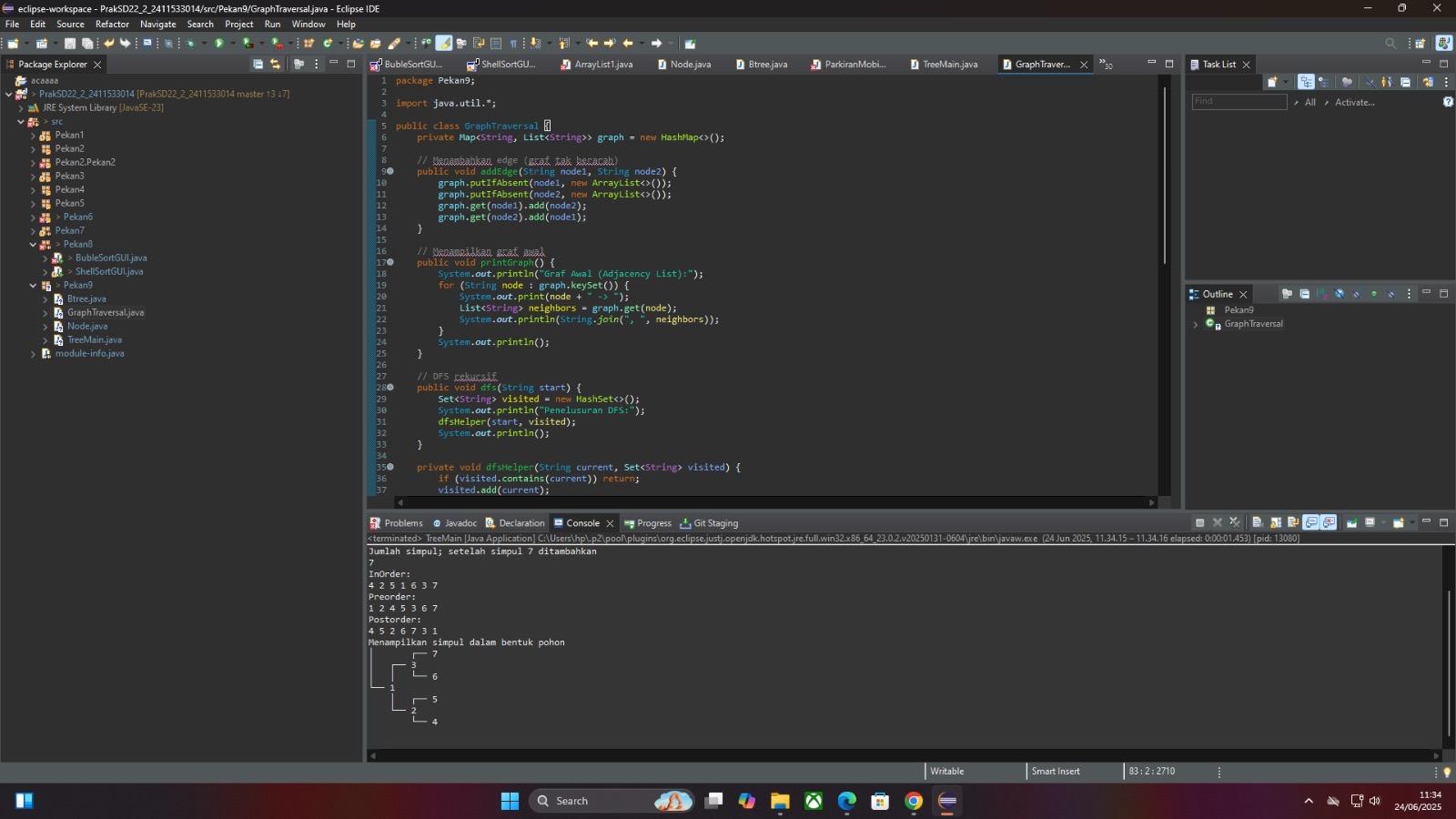
6. Metode Pencetakan Visual (print):

public void print() {

root.print();

}

* GraphTraversal



1. Import dan Variabel Instansi:

import java.util.\*;

public class GraphTraversal {

private HashMap<String, List<String>> graph = new HashMap<>();

• import java.util.\*;: Mengimpor berbagai kelas utilitas dari Java, termasuk HashMap dan List.

• private HashMap<String, List<String>> graph = new HashMap<>();: Mendefinisikan graf sebagai HashMap. Kunci adalah node (String), dan nilainya adalah daftar node tetangga (List). Ini merepresentasikan graf sebagai daftar adjacency.

2. Metode Menambahkan Node dan Edge:

public void addNode(String node) {

graph.putIfAbsent(node, new ArrayList<>());

}

public void addEdge(String node1, String node2) {

graph.get(node1).add(node2);

graph.get(node2).add(node1); // Untuk graf tak berarah

}

• addNode(String node): Menambahkan node ke graf. Jika node sudah ada, tidak melakukan apa-apa.

• addEdge(String node1, String node2): Menambahkan edge antara node1 dan node2. Perhatikan bahwa kode menambahkan edge dalam kedua arah (node1 ke node2 dan node2 ke node1), menunjukkan bahwa graf yang diimplementasikan adalah graf tak berarah.

3. Metode Mencetak Graf:

public void printGraph() {

System.out.println("Graf Awal (Adjacency List):");

for (String node : graph.keySet()) {

List<String> neighbors = graph.get(node);

System.out.println(node + " -> " + neighbors.toString());

}

}

Metode ini mencetak representasi graf sebagai daftar adjacency.

4. Metode Depth-First Search (DFS):

public void dfs(String start) {

Set<String> visited = new HashSet<>();

dfsHelper(start, visited);

System.out.println("Penelusuran DFS:");

System.out.println(visited);

}

private void dfsHelper(String current, Set<String> visited) {

if (visited.contains(current)) return;

visited.add(current);

for (String neighbor : graph.get(current)) {

dfsHelper(neighbor, visited);

}

}

• dfs(String start): Melakukan Depth-First Search dimulai dari node start. Ia menggunakan HashSet untuk melacak node yang sudah dikunjungi.

• dfsHelper(String current, Set<String> visited): Metode rekursif yang melakukan penelusuran DFS.

5. Metode Breadth-First Search (BFS):

public void bfs(String start) {

Set<String> visited = new HashSet<>();

Queue<String> queue = new LinkedList<>();

queue.add(start);

visited.add(start);

System.out.println("Penelusuran BFS:");

while (!queue.isEmpty()) {

String current = queue.poll();

System.out.print(current + " ");

for (String neighbor : graph.getOrDefault(current, new ArrayList<>())) {

if (!visited.contains(neighbor)) {

visited.add(neighbor);

queue.add(neighbor);

}

}

}

System.out.println();

}

Metode bfs(String start) melakukan Breadth-First Search, dimulai dari node start. Ia menggunakan Queue (dalam hal ini LinkedList) untuk menyimpan node yang akan dikunjungi, dan HashSet untuk melacak node yang sudah dikunjungi. Algoritma BFS mengunjungi semua tetangga node saat ini sebelum mengunjungi tetangga dari tetangga tersebut, berbeda dengan DFS yang menjelajahi sedalam mungkin pada satu cabang sebelum kembali.

6. Metode main:

public static void main(String[] args) {

GraphTraversal graph = new GraphTraversal();

// ... (kode untuk menambahkan node dan edge ke graf) ...

graph.printGraph();

graph.dfs("A");

graph.bfs("A");

}

Metode main membuat objek GraphTraversal, menambahkan node dan edge (kode ini tidak ditampilkan dalam gambar), kemudian mencetak graf, dan melakukan DFS serta BFS dimulai dari node "A".

1. KESIMPULAN

Pratikum ini berhasil menunjukkan implementasi dan perbandingan algoritma DFS dan BFS untuk traversal graf. Implementasi menggunakan representasi graf berbasis adjacency list, yang terbukti efisien untuk operasi penambahan dan penghapusan node dan edge. Hasil pratikum mengkonfirmasi perbedaan fundamental antara kedua algoritma tersebut dalam hal urutan kunjungan node, dengan DFS yang menjelajahi graf secara mendalam dan BFS secara berlapis. Pemahaman tentang perbedaan ini sangat penting dalam memilih algoritma yang paling sesuai untuk aplikasi spesifik, karena kinerja dan hasil yang dihasilkan dapat berbeda secara signifikan tergantung pada struktur graf dan tujuan traversal.

Keberhasilan implementasi ini memberikan pemahaman yang lebih baik tentang konsep dasar algoritma graf dan bagaimana mereka dapat diterapkan dalam berbagai konteks. Pengembangan lebih lanjut dapat melibatkan pengukuran waktu eksekusi untuk analisis kinerja yang lebih komprehensif, implementasi pada graf berarah, serta eksplorasi algoritma traversal graf lainnya. Pengalaman ini memperkuat pemahaman tentang struktur data graf dan algoritma traversal, yang merupakan konsep dasar dalam ilmu komputer.