Assignment - 4

1. Problem Statement

1.1. Raymond’s Algorithm -- Diffusion-Computation approach

2. Source Code

2.1. >> Main.java

```java
/* About Project
   This is Raymond’s Algorithm Diffusion-Computation approach using threads
*/
/* About Main
       The Main class provided constructs a tree structure from a file based on hierarchical
indentation,
       using a stack to manage node relationships and a hashmap to store nodes by their IDs.
       It includes functionality to prompt the user for a file using AWT's FileDialog, and
       prints the tree recursively starting from the root node found in the hashmap.
       The main method orchestrates these operations, also implementing threading for each
node to simulate concurrent processes,
       although the use of reverse order in the TreeMap for threading purposes is unusual
unless specifically required by the test scenario.
       Overall, while the class showcases tree manipulation and concurrent execution, it
appears geared towards testing or demonstration
       rather than implementing Raymond's inverted tree-based mutual exclusion algorithm
directly.
*/

/* Author
  Rahul Biswas, Arijit Mondal
*/
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Stack;
import java.util.TreeMap;
import java.util.Comparator;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.awt.*;

public class Main {

   // Method to build a tree structure from a file
   public static HashMap<Integer, Node> treeBuilder(String filePath) throws IOException {
      BufferedReader reader = new BufferedReader(new FileReader(filePath));
```

```
        String line;
        int id = 1;
        Stack<Node> precedenceStack = new Stack<>();
        HashMap<Integer, Node> treeNodes = new HashMap<>();

        // Read lines from the file
        while ((line = reader.readLine()) != null) {
            // Calculate the level of indentation (assuming 4 spaces per level)
            int level = countLeadingSpaces(line) / 4;
            String nodeName = line.trim();
            Node newNode = new Node(id, nodeName, level);

            if (level == 0) {
                // If it's a root node, push to stack
                precedenceStack.push(newNode);
                System.out.println("New Node (leaf) pushed: name: " + newNode.getName() + "
id: " + newNode.getId());
            } else {
                // Connect child nodes to their parents
                while (!precedenceStack.isEmpty()) {
                    Node child = precedenceStack.peek();
                    if (child.getLevel() < level) {
                        child.addParent(newNode); // Set current node as child's parent
                        treeNodes.put(child.getId(), child); // Add child to treeNodes map
                        precedenceStack.pop(); // Remove processed node from stack
                        System.out.println("Child Popped: name: " + child.getName() + " id: " +
child.getId()
                                + " parentid: " + child.getParentID());
                    } else {
                        break;
                    }
                }
                // Push current node to stack
                precedenceStack.push(newNode);
                System.out.println("New Node (internal) pushed: name: " + newNode.getName() +
" id: " + newNode.getId());
            }
            id++;
        }

        int rootCounter = 0;
        // Pop remaining nodes from stack to finalize the tree
        while (!precedenceStack.isEmpty()) {
            Node node = precedenceStack.pop();
            node.setStatus(Status.PHOLD); // Set status of node
            node.addParent(null); // Set parent of root node to null
            treeNodes.put(node.getId(), node); // Add root node to treeNodes map
            System.out.println("Root Popped: name: " + node.getName() + " id: " + node.getId()
+ " parent Id: " + node.getParentID());
            System.out.println();
            System.out.println();
            System.out.println("System Log:");
            System.out.println("Information Message: Node " + node.getName() + " with ID " +
node.getId() + " have been set to PHOLD initially.");
            rootCounter++;
        }

        reader.close();
        // Check if exactly one root node exists in the tree
        if (rootCounter != 1) {
            System.out.println("System Message: Multiple roots found.");
            return null;
        }
        return treeNodes; // Return the constructed treeNodes map
    }

    // Method to count leading spaces in a string
    public static int countLeadingSpaces(String line) {
        int count = 0;
        while (count < line.length() && line.charAt(count) == ' ') {
            count++;
        }
        return count;
    }

    // Method to open file dialog and return selected file path
    private static String openFileDialog() {
        FileDialog fd = new FileDialog((Frame) null, "Open", FileDialog.LOAD);
        fd.setVisible(true);
        String filename = fd.getFile();
        if (filename != null) {
            return fd.getDirectory() + filename; // Return full file path
        }
        return null;
    }

    // Method to print the tree structure starting from the root node
    public static void printTree(HashMap<Integer, Node> nodeMap) {
        Node root = null;
        // Find the root node (node with no parent)
        for (Node node : nodeMap.values()) {
            if (node.getParentID() == 0) {
                root = node;
                break;
            }
        }

        // Print the tree recursively starting from the root
        if (root != null) {
            printNode(root, 0, nodeMap); // Start printing from the root node
        } else {
            System.out.println("No root node found.");
        }
    }

    // Recursive method to print nodes and their children
    private static void printNode(Node node, int level, Map<Integer, Node> nodeMap) {
        for (int i = 0; i < level; i++) {
            System.out.print("  "); // Indentation based on level
        }
        System.out.println(node.getName()); // Print node's name

        // Recursively print children nodes
        for (Node child : nodeMap.values()) {
            if (child.getParentID() == node.getId()) {
                printNode(child, level + 1, nodeMap); // Print child nodes recursively
            }
        }
    }

    // Main method
    public static void main(String[] args) {
        try {
            String filePath = openFileDialog(); // Open file dialog to select input file
            if(filePath == null){
                System.out.println("\nNo Input File is selected!!");
                System.exit(0);
            }
            HashMap<Integer, Node> treeNodes = treeBuilder(filePath); // Build tree from
selected file

            List<Thread> threads = new LinkedList<>();
            TreeMap<Integer, Node> sortedMap = new TreeMap<>(Comparator.reverseOrder());
            sortedMap.putAll(treeNodes); // Sort nodes in reverse order by ID

            if (sortedMap != null) {
                // Start threads for each node in reverse order
                for (Node node : sortedMap.values()) {
                    Thread thread = new Thread(node);
                    threads.add(thread);
                    thread.start(); // Start thread for each node
                }

                // Wait for all threads to complete
                for (Thread thread : threads) {
                    try {
                        thread.join(); // Wait for thread to finish
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

                // After threads finish, print the final inverted tree structure
                System.out.println("\nAll system processes have finished.");
                System.out.println("\nFinal Inverted tree is:");
                printTree(treeNodes);

                System.exit(0); // Exit the program
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2.2. **>> Node.java**

```
/* About node.java
      The Node class models individual nodes within a distributed system, where each
node communicates with its parent node
      to coordinate access to critical sections using token-based synchronization. Key
attributes include a unique identifier
      (id), name (nodeName), hierarchical level (level), parent reference (parent),
      current status (Status.NONE, Status.REQUESTING, Status.PHOLD, Status.ABORT),
and various counters and queues for managing requests
      and abort conditions. The class implements the Runnable interface to support
concurrent execution, encapsulating logic for requesting and sending tokens,
      updating statuses based on probabilistic events, and managing time spent in critical
states. Nodes interact exclusively
      with their parent node to regulate access to shared resources, transitioning states
dynamically based on internal conditions and external interactions.
      This setup enables nodes to operate autonomously yet collaboratively within the
distributed system, demonstrating fundamental
      concepts of mutual exclusion and distributed synchronization.
*/

import java.util.Queue;
import java.util.Random;
import java.util.concurrent.ConcurrentLinkedQueue;

public class Node implements Runnable {
    private int id;
    private String nodeName;
    private int level;
    private Node parent;
    private Status status = Status.NONE; // Current status of the node
    private Random random = new Random(); // Random number generator for
probabilistic behavior
    private int InCriticalStateClock = -1; // Clock for managing time in critical state
    private int abortCounter = 0; // Counter for abort conditions
    private int rootAbortCounter = 0; // Counter for root abort conditions
    private Queue<Node> requestQueue = new ConcurrentLinkedQueue<>(); // Queue
for holding requests
    private Boolean isRequested = false; // Flag indicating if a request has been made

    // Constructor to initialize the node with an ID, name, and level
    public Node(int id, String nodeName, int level) {
        this.id = id;
        this.nodeName = nodeName;
        this.level = level;
    }

    // Getter for the node's level
    public int getLevel() {
        return level;
    }

    // Getter for the node's ID
    public int getId() {
        return id;
    }

    // Method to set the parent node
    public void addParent(Node parent) {
        this.parent = parent;
    }

    // Getter for the node's name
    public String getName() {
        return nodeName;
    }

    // Getter for the ID of the parent node
    public Integer getParentID() {
        return (parent != null ? parent.getId() : 0);
    }

    // Setter for the node's status
    public void setStatus(Status st) {
        this.status = st;
    }

    // Run method required by the Runnable interface
    @Override
    public synchronized void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                // Update the node's status and handle related logic
                setStatus();
                checkClock();

                // Node is requesting token from its parent
                if (status == Status.REQUESTING && !isRequested) {
                    requestQueue.add(this);
                    isRequested = true;
                    System.out.println("Request Message :: Node " + this.getName() + " [ " +
status + " ] with ID "
                            + this.getId() + " is requesting token from " + this.parent.getName());
                    parent.requestToken(this); // Request token from parent
                }
            // Node is in a hold state and has requests in the queue
            if (status == Status.PHOLD && !requestQueue.isEmpty() &&
InCriticalStateClock <= 0) {
               this.isRequested = false;
               Node requestedNode = requestQueue.poll(); // Retrieve next request
               if (requestedNode.getId() == this.id) {
                  // Node enters critical state
                  System.out.println("Information Message :: Node " +
requestedNode.getName() + " with ID "
                        + requestedNode.getId() + " is in critical state");
                  InCriticalStateClock = random.nextInt(21) + 10; // Set clock for critical
state
                  this.addParent(null); // Node has no parent in critical state
               } else {
                  // Node sends token to requested node
                  System.out.println("Send Message :: Node " + this.getName() + " [ " + status
+ " ] with ID "
                        + this.getId() + " is sending token to " + requestedNode.getName());
                  requestedNode.sendToken(); // Send token to requested node
                  System.out.println("Information Message :: Node " +
requestedNode.getName() + " with ID "
                        + requestedNode.getId() + " is now parent of " + getName());
                  this.addParent(requestedNode); // Set requested node as parent
                  if (!requestQueue.isEmpty()) {
                     // Node has more requests, continue requesting
                     this.setStatus(Status.REQUESTING);
                     isRequested = true;
                     System.out.println("Request Message :: Node " + this.getName() + " [ " +
status
                           + " ] with ID " + this.getId() + " is requesting token back from "
                           + this.parent.getName() + " to send " +
requestQueue.peek().getName());
                     parent.requestToken(this); // Request token from parent
                  } else {
                     this.setStatus(Status.NONE); // No more requests, set status to NONE
                  }
               }
            }

            Thread.sleep(600); // Adjust delay time as needed
            // Node is terminated due to abort status
            if (status == Status.ABORT) {
               System.out.println("Termination Message :: Node " + nodeName + " with ID: "
+ id
                     + " has been terminated due to an empty request queue.");
               Thread.currentThread().interrupt(); // Interrupt current thread
            }
         }
      } catch (Exception e) {
         e.printStackTrace(); // Print stack trace for any exceptions
      }
   }

   // Method for requesting token from parent node
```

```
    public void requestToken(Node child) throws InterruptedException {
        Thread.sleep(500); // Adjust delay time as needed
        requestQueue.add(child); // Add child node to request queue
        if (status == Status.NONE && !isRequested) {
            isRequested = true;
            System.out.println("Request Message :: Node " + this.getName() + " [ " + status + "
] with ID "
                    + this.getId() + " is requesting token from " + this.parent.getName());
            parent.requestToken(this); // Request token from parent
        }
    }

    // Method for sending token to child node
    public void sendToken() throws InterruptedException {
        Thread.sleep(500); // Adjust delay time as needed
        this.setStatus(Status.PHOLD); // Set node's status to PHOLD (holding token)
    }

    // Method to update the node's status based on specific conditions
    private void setStatus() {
        // Update status if current status is NONE
        if (this.status == Status.NONE) {
            int randomValue = random.nextInt(1000); // Generate random value for
probabilistic behavior
            if (randomValue < 20) { // 20 out of 1000 probability (2%) for requesting status
                this.setStatus(Status.REQUESTING);
            } else {
                if (requestQueue.isEmpty())
                    abortCounter++; // Increment abort counter if request queue is empty
                else
                    abortCounter = 0; // Reset abort counter if requests are pending
            }
            // Set node's status to ABORT if abort conditions are met
            if (abortCounter == 60) {
                this.setStatus(Status.ABORT);
            }
        }

        // Handle root node abort conditions when in PHOLD status and clock is 0
        if (status == Status.PHOLD && InCriticalStateClock == 0) {
            if (requestQueue.isEmpty())
                rootAbortCounter++; // Increment root abort counter if request queue is
empty
            else
                rootAbortCounter = 0; // Reset root abort counter if requests are pending
            // Set node's status to ABORT if root abort conditions are met
            if (rootAbortCounter == 20) {
                this.setStatus(Status.ABORT);
            }
        }
        // Check if parent node is in abort status and update node's status accordingly
        if (parent != null) {
            if (this.status != Status.PHOLD && parent.status == Status.ABORT) {
                this.status = Status.ABORT;
            }
        }
    }

    // Method to decrement the critical state clock
    private void checkClock() {
        if (InCriticalStateClock != 0) {
            InCriticalStateClock--; // Decrement clock if node is in critical state
        }
    }
}
```

2.3. >> Status.java

```
/* About status class:
  The Status enum class defines a set of constants representing different states within a
program or algorithm.
   PHOLD signifies a node or process in a holding state, often seen in mutual exclusion
algorithms where nodes await
   resource availability or critical section access. REQUESTING indicates active resource
request, crucial for managing
   concurrent access in distributed systems to prevent conflicts. NONE represents a default
or undefined state when no specific action is ongoing.
   ABORT flags abnormal termination conditions, if no node is making a request, or if it is
making a request
   to no one for an extended period of time, it may be terminated.,
   enforcing rules governing resource access and ensuring orderly process interactions in
distributed environments.
   */

public enum Status {
    PHOLD,      // Node is holding a resource
    REQUESTING, // Node is requesting access to a critical section
    NONE,       // Default or no specific state
    ABORT       // Operation or process is aborted
}
```

3. <u>Pre-requisites</u>

3.1.<u>Java Development Kit (JDK):</u> Ensure JDK is installed and properly configured on your system. This code appears to be written in Java, so JDK is essential for compiling and running Java programs.

3.2. **<u>Input File Format:</u>** This modified indentation presentation of a tree structure provides a clear and intuitive way to represent hierarchical relationships in a textual format. It's versatile enough for various applications where tree-like structures need to be managed, parsed, or displayed. Understanding and correctly interpreting this format is essential for building robust applications that work with hierarchical data effectively. The structure uses indentation to denote parent-child relationships, where each line represents a node in the tree. The indentation level indicates the depth of the node in the tree hierarchy:

- Nodes are aligned to the left margin and represent the leaf nodes of the tree.
- Subsequent Depths: Nodes are indented to the right by a consistent number (4) of spaces relative to their child nodes.

  Examples of this provided in Sample Input file folder.

3.3. Code run: Use your IDE or command-line tools (like javac for compilation and java for execution) to compile and run the Main class. Ensure all dependencies are resolved and paths (for input files) are correctly specified.

Steps to run without any IDE:

1. Open terminal in folder. Then run these commands sequentially,

The program uses System.out.println() extensively for logging and displaying messages related to node processing, thread actions, and system status. Ensure the console output is clear and visible during program execution.

3.4. Abstract Window Toolkit: The method uses FileDialog to prompt the user to select an input file. Ensure that the AWT components (FileDialog and Frame) work correctly on your system.

3.5. Node Behavior: Each Node in the program simulates a process in a distributed system. Understand how each Node transitions through different states (Status) based on random events (method) and interactions with other nodes (and methods).

3.6. System Resources: Ensure your system has sufficient resources (CPU, memory) to handle potentially intensive thread operations, especially if dealing with a large number of nodes or complex tree structures.

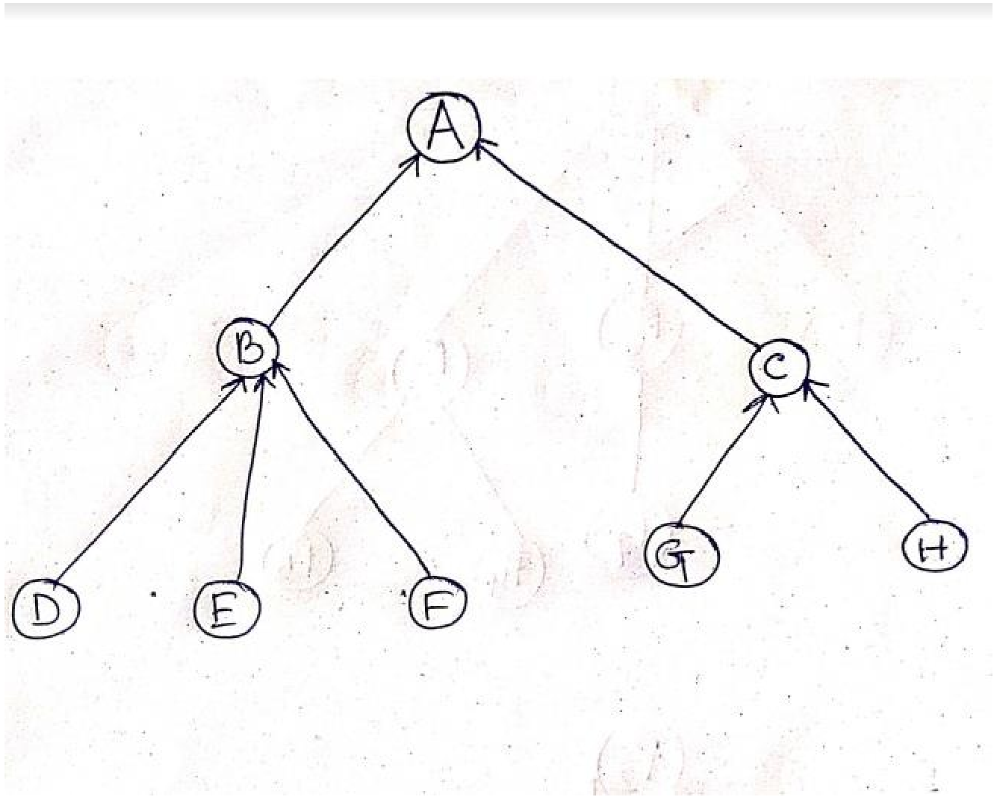
3.7. Understanding of Java I/O and Multithreading: Familiarity with Java I/O operations ( ) and multithreading ( ) is crucial, as the code involves file reading, concurrent processing using threads, and synchronization.

3.8. Java Enum Usage: Understanding how Java enums (enum) work and how they are utilized for defining states ( ) in the Node class.

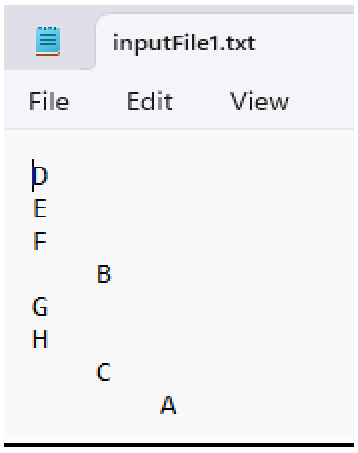
4. Results

4.1. Result 1

4.1.1. Input Tree Diagram

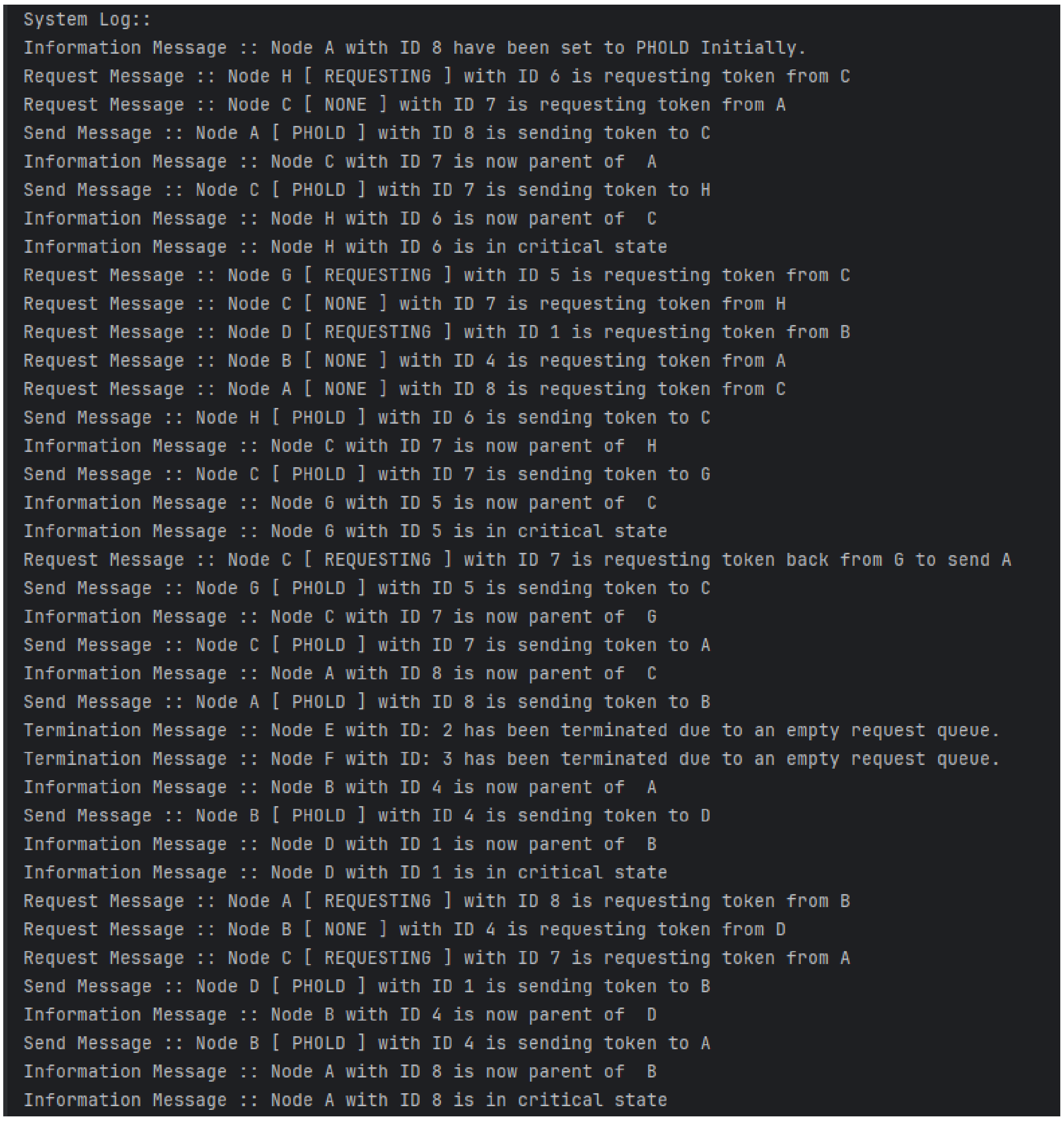


4.1.2. Input File



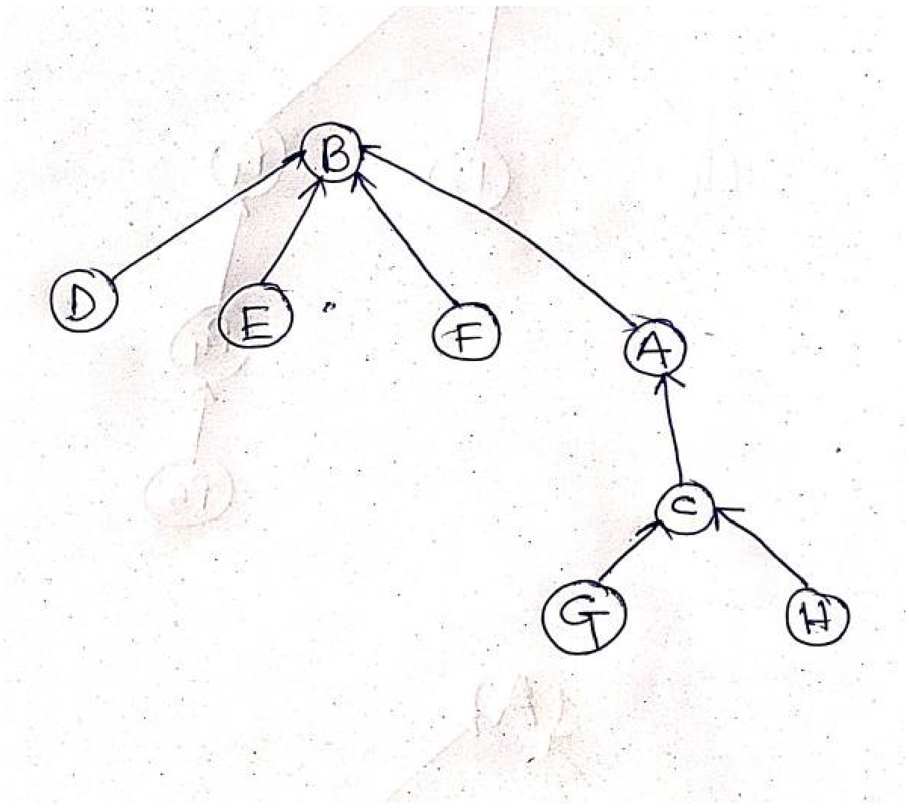
```
D
E
F
      B
G
H
      C
            A
```

4.1.3. Output File



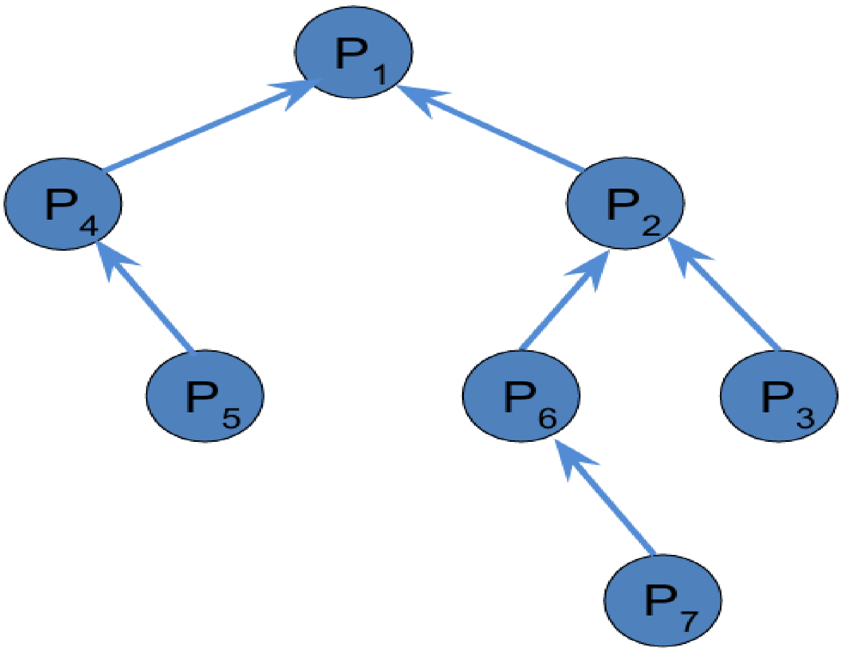
```
System Log::
Information Message :: Node A with ID 8 have been set to PHOLD Initially.
Request Message :: Node H [ REQUESTING ] with ID 6 is requesting token from C
Request Message :: Node C [ NONE ] with ID 7 is requesting token from A
Send Message :: Node A [ PHOLD ] with ID 8 is sending token to C
Information Message :: Node C with ID 7 is now parent of  A
Send Message :: Node C [ PHOLD ] with ID 7 is sending token to H
Information Message :: Node H with ID 6 is now parent of  C
Information Message :: Node H with ID 6 is in critical state
Request Message :: Node G [ REQUESTING ] with ID 5 is requesting token from C
Request Message :: Node C [ NONE ] with ID 7 is requesting token from H
Request Message :: Node D [ REQUESTING ] with ID 1 is requesting token from B
Request Message :: Node B [ NONE ] with ID 4 is requesting token from A
Request Message :: Node A [ NONE ] with ID 8 is requesting token from C
Send Message :: Node H [ PHOLD ] with ID 6 is sending token to C
Information Message :: Node C with ID 7 is now parent of  H
Send Message :: Node C [ PHOLD ] with ID 7 is sending token to G
Information Message :: Node G with ID 5 is now parent of  C
Information Message :: Node G with ID 5 is in critical state
Request Message :: Node C [ REQUESTING ] with ID 7 is requesting token back from G to send A
Send Message :: Node G [ PHOLD ] with ID 5 is sending token to C
Information Message :: Node C with ID 7 is now parent of  G
Send Message :: Node C [ PHOLD ] with ID 7 is sending token to A
Information Message :: Node A with ID 8 is now parent of  C
Send Message :: Node A [ PHOLD ] with ID 8 is sending token to B
Termination Message :: Node E with ID: 2 has been terminated due to an empty request queue.
Termination Message :: Node F with ID: 3 has been terminated due to an empty request queue.
Information Message :: Node B with ID 4 is now parent of  A
Send Message :: Node B [ PHOLD ] with ID 4 is sending token to D
Information Message :: Node D with ID 1 is now parent of  B
Information Message :: Node D with ID 1 is in critical state
Request Message :: Node A [ REQUESTING ] with ID 8 is requesting token from B
Request Message :: Node B [ NONE ] with ID 4 is requesting token from D
Request Message :: Node C [ REQUESTING ] with ID 7 is requesting token from A
Send Message :: Node D [ PHOLD ] with ID 1 is sending token to B
Information Message :: Node B with ID 4 is now parent of  D
Send Message :: Node B [ PHOLD ] with ID 4 is sending token to A
Information Message :: Node A with ID 8 is now parent of  B
Information Message :: Node A with ID 8 is in critical state
```

4.1.4. Output Tree Diagram

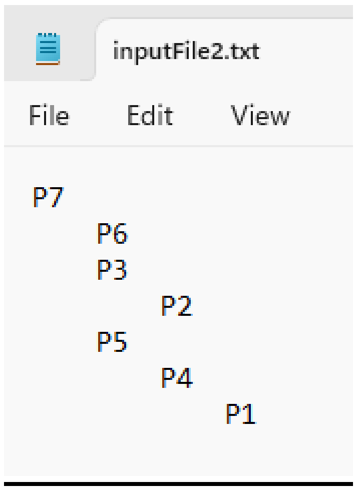


4.2. Result 2

4.2.1. Input Tree Diagram

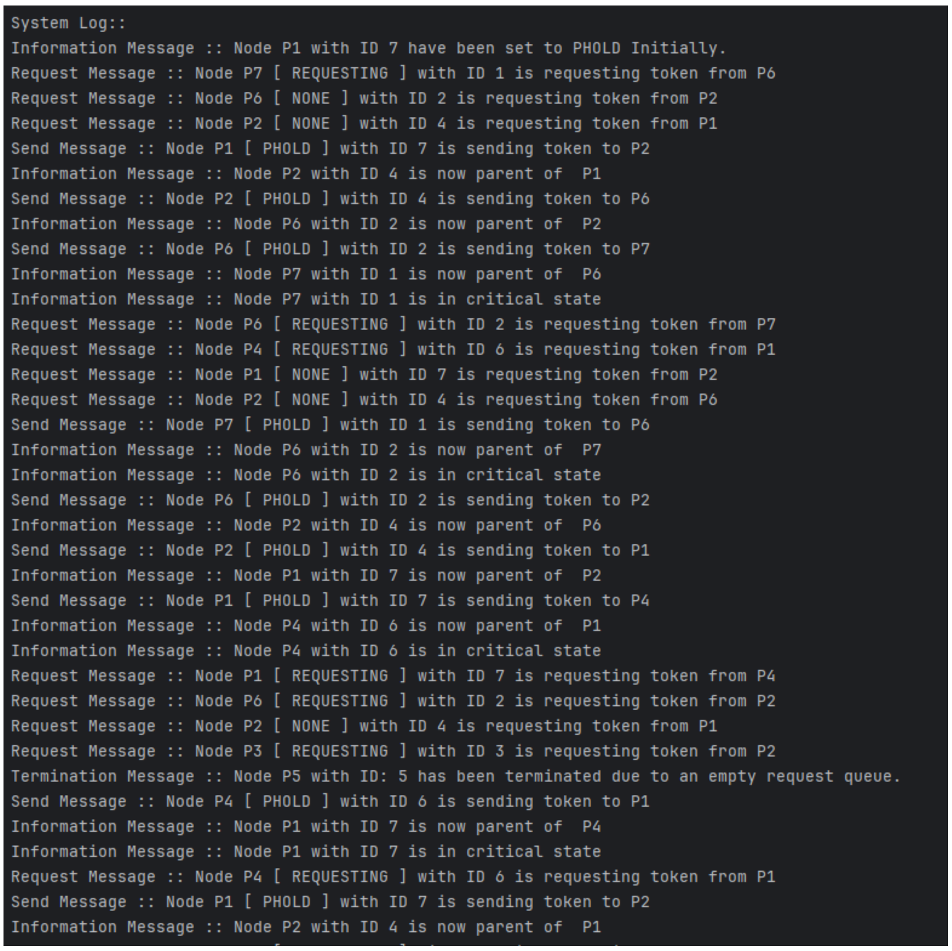


4.2.2. Input File



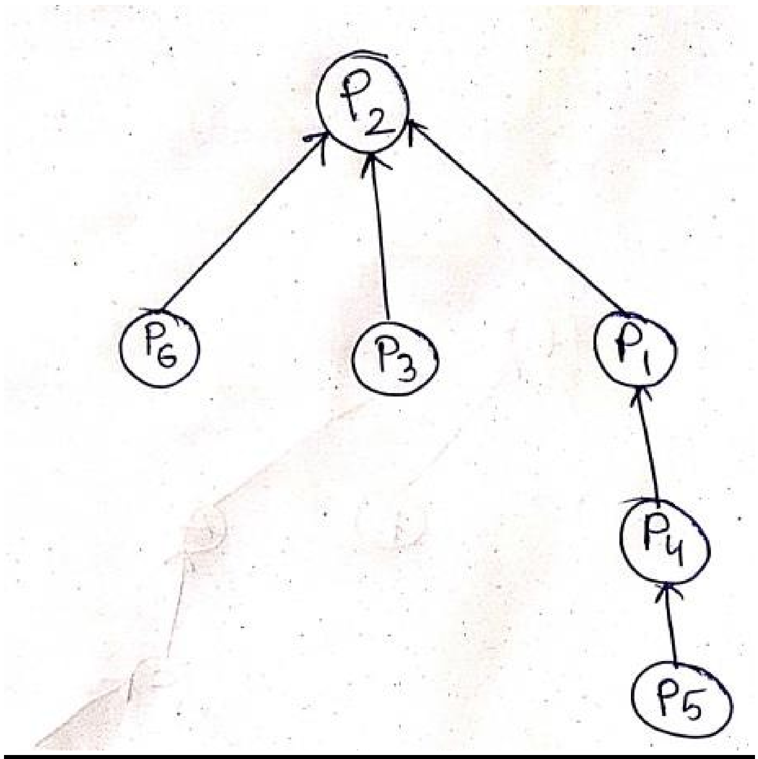
```
P7
	P6
	P3
		P2
	P5
		P4
			P1
```

4.2.3. Output File



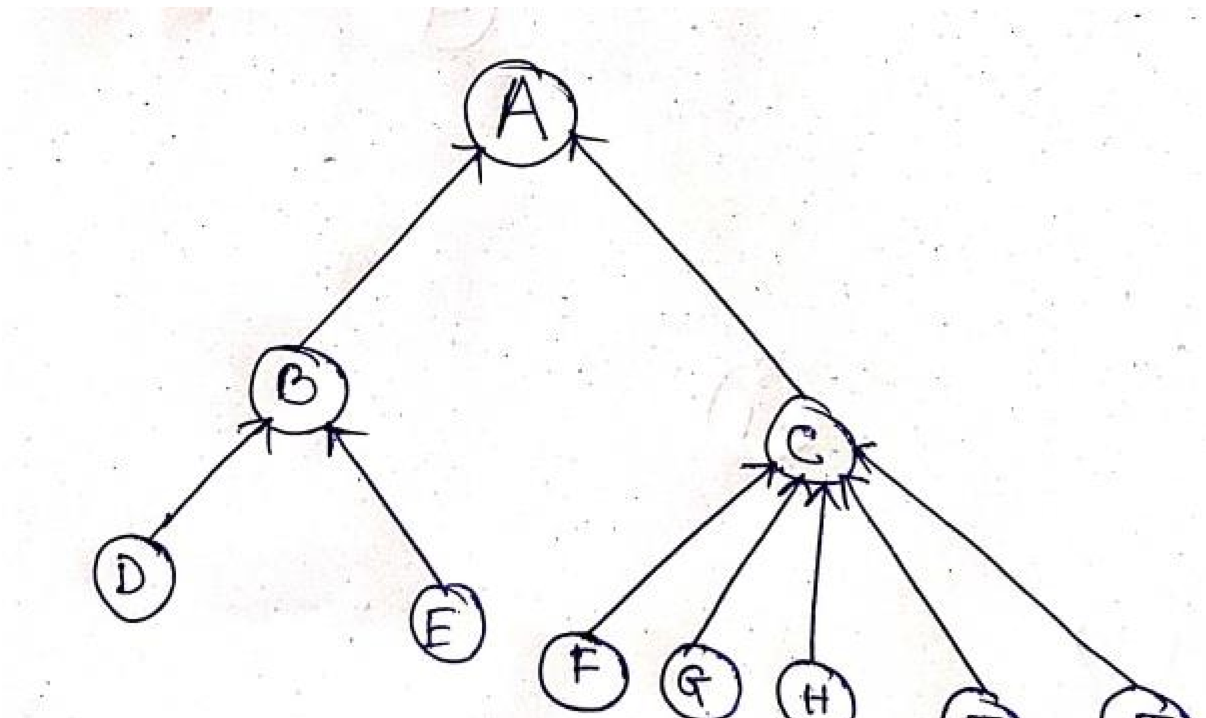
```
System Log::
Information Message :: Node P1 with ID 7 have been set to PHOLD Initially.
Request Message :: Node P7 [ REQUESTING ] with ID 1 is requesting token from P6
Request Message :: Node P6 [ NONE ] with ID 2 is requesting token from P2
Request Message :: Node P2 [ NONE ] with ID 4 is requesting token from P1
Send Message :: Node P1 [ PHOLD ] with ID 7 is sending token to P2
Information Message :: Node P2 with ID 4 is now parent of  P1
Send Message :: Node P2 [ PHOLD ] with ID 4 is sending token to P6
Information Message :: Node P6 with ID 2 is now parent of  P2
Send Message :: Node P6 [ PHOLD ] with ID 2 is sending token to P7
Information Message :: Node P7 with ID 1 is now parent of  P6
Information Message :: Node P7 with ID 1 is in critical state
Request Message :: Node P6 [ REQUESTING ] with ID 2 is requesting token from P7
Request Message :: Node P4 [ REQUESTING ] with ID 6 is requesting token from P1
Request Message :: Node P1 [ NONE ] with ID 7 is requesting token from P2
Request Message :: Node P2 [ NONE ] with ID 4 is requesting token from P6
Send Message :: Node P7 [ PHOLD ] with ID 1 is sending token to P6
Information Message :: Node P6 with ID 2 is now parent of  P7
Information Message :: Node P6 with ID 2 is in critical state
Send Message :: Node P6 [ PHOLD ] with ID 2 is sending token to P2
Information Message :: Node P2 with ID 4 is now parent of  P6
Send Message :: Node P2 [ PHOLD ] with ID 4 is sending token to P1
Information Message :: Node P1 with ID 7 is now parent of  P2
Send Message :: Node P1 [ PHOLD ] with ID 7 is sending token to P4
Information Message :: Node P4 with ID 6 is now parent of  P1
Information Message :: Node P4 with ID 6 is in critical state
Request Message :: Node P1 [ REQUESTING ] with ID 7 is requesting token from P4
Request Message :: Node P6 [ REQUESTING ] with ID 2 is requesting token from P2
Request Message :: Node P2 [ NONE ] with ID 4 is requesting token from P1
Request Message :: Node P3 [ REQUESTING ] with ID 3 is requesting token from P2
Termination Message :: Node P5 with ID: 5 has been terminated due to an empty request queue.
Send Message :: Node P4 [ PHOLD ] with ID 6 is sending token to P1
Information Message :: Node P1 with ID 7 is now parent of  P4
Information Message :: Node P1 with ID 7 is in critical state
Request Message :: Node P4 [ REQUESTING ] with ID 6 is requesting token from P1
Send Message :: Node P1 [ PHOLD ] with ID 7 is sending token to P2
Information Message :: Node P2 with ID 4 is now parent of  P1
```

4.2.4. Output Tree Diagram

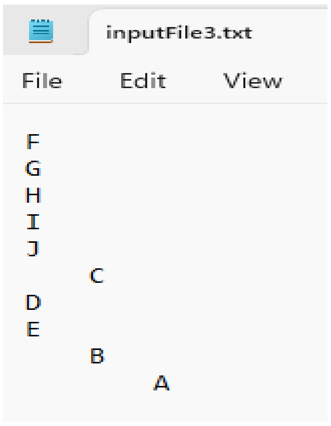


4.3. Result 3

4.3.1. Input Tree Diagram

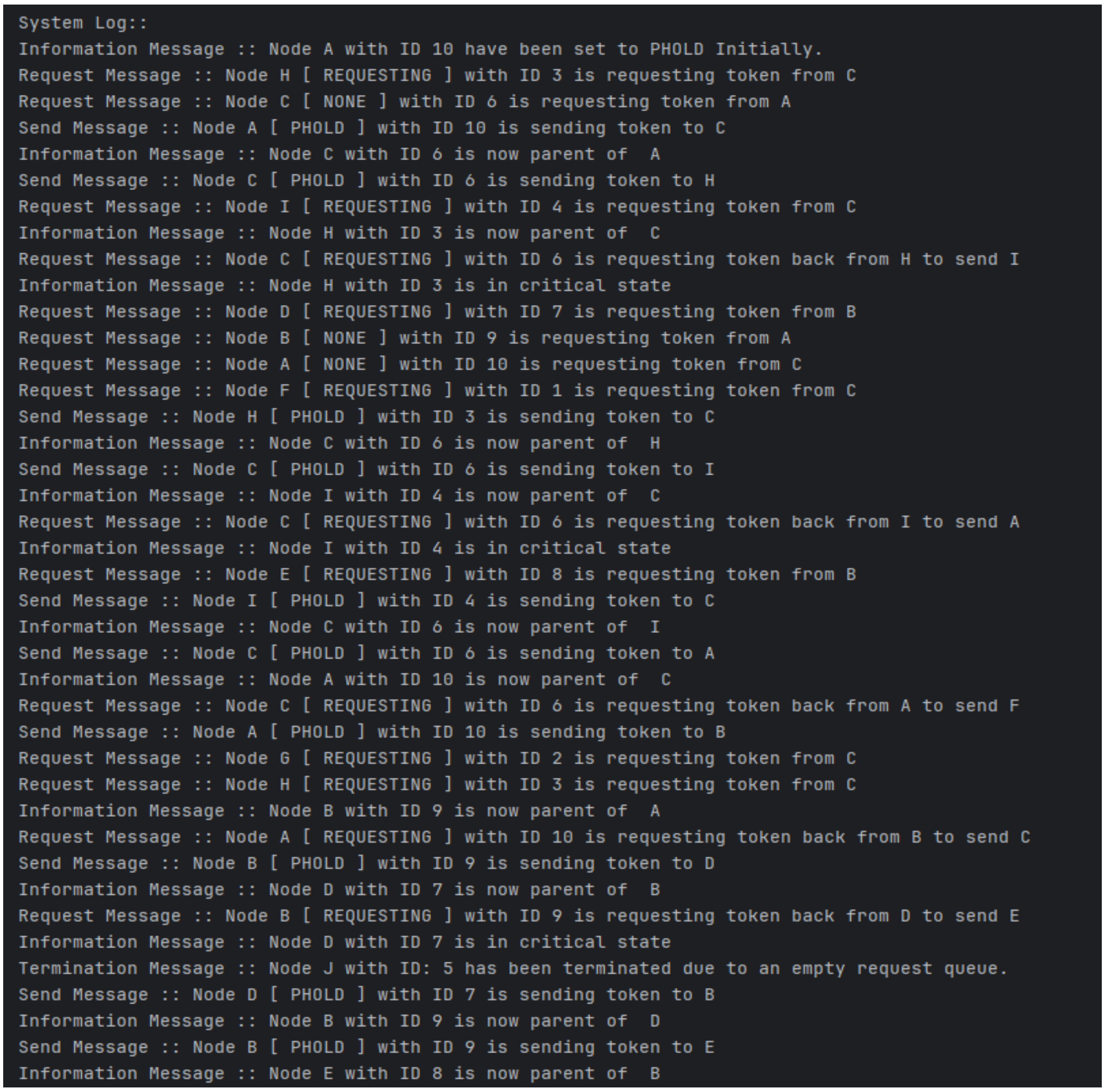


4.3.2. Input File



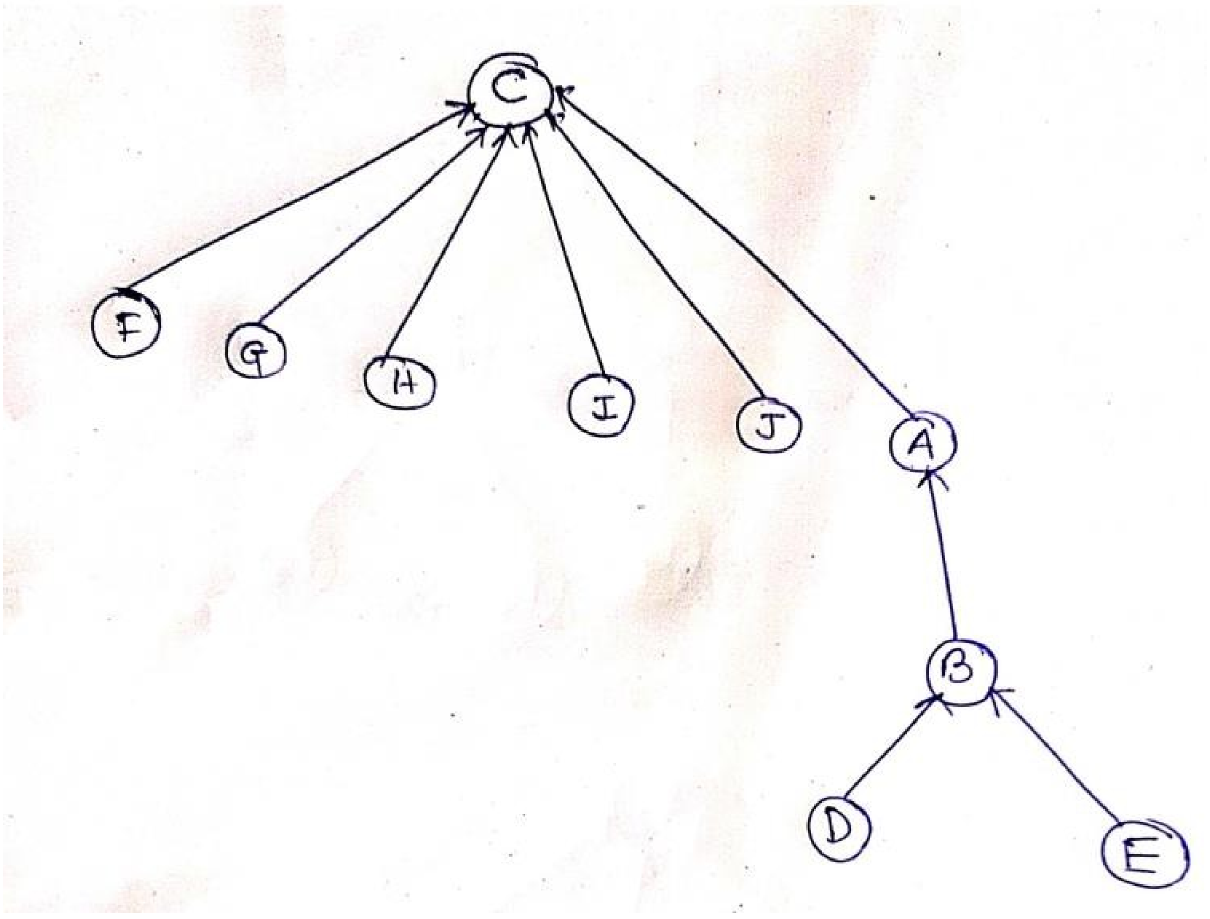
```
F
G
H
I
J
     C
D
E
     B
          A
```

4.3.3. Output File



```
System Log::
Information Message :: Node A with ID 10 have been set to PHOLD Initially.
Request Message :: Node H [ REQUESTING ] with ID 3 is requesting token from C
Request Message :: Node C [ NONE ] with ID 6 is requesting token from A
Send Message :: Node A [ PHOLD ] with ID 10 is sending token to C
Information Message :: Node C with ID 6 is now parent of  A
Send Message :: Node C [ PHOLD ] with ID 6 is sending token to H
Request Message :: Node I [ REQUESTING ] with ID 4 is requesting token from C
Information Message :: Node H with ID 3 is now parent of  C
Request Message :: Node C [ REQUESTING ] with ID 6 is requesting token back from H to send I
Information Message :: Node H with ID 3 is in critical state
Request Message :: Node D [ REQUESTING ] with ID 7 is requesting token from B
Request Message :: Node B [ NONE ] with ID 9 is requesting token from A
Request Message :: Node A [ NONE ] with ID 10 is requesting token from C
Request Message :: Node F [ REQUESTING ] with ID 1 is requesting token from C
Send Message :: Node H [ PHOLD ] with ID 3 is sending token to C
Information Message :: Node C with ID 6 is now parent of  H
Send Message :: Node C [ PHOLD ] with ID 6 is sending token to I
Information Message :: Node I with ID 4 is now parent of  C
Request Message :: Node C [ REQUESTING ] with ID 6 is requesting token back from I to send A
Information Message :: Node I with ID 4 is in critical state
Request Message :: Node E [ REQUESTING ] with ID 8 is requesting token from B
Send Message :: Node I [ PHOLD ] with ID 4 is sending token to C
Information Message :: Node C with ID 6 is now parent of  I
Send Message :: Node C [ PHOLD ] with ID 6 is sending token to A
Information Message :: Node A with ID 10 is now parent of  C
Request Message :: Node C [ REQUESTING ] with ID 6 is requesting token back from A to send F
Send Message :: Node A [ PHOLD ] with ID 10 is sending token to B
Request Message :: Node G [ REQUESTING ] with ID 2 is requesting token from C
Request Message :: Node H [ REQUESTING ] with ID 3 is requesting token from C
Information Message :: Node B with ID 9 is now parent of  A
Request Message :: Node A [ REQUESTING ] with ID 10 is requesting token back from B to send C
Send Message :: Node B [ PHOLD ] with ID 9 is sending token to D
Information Message :: Node D with ID 7 is now parent of  B
Request Message :: Node B [ REQUESTING ] with ID 9 is requesting token back from D to send E
Information Message :: Node D with ID 7 is in critical state
Termination Message :: Node J with ID: 5 has been terminated due to an empty request queue.
Send Message :: Node D [ PHOLD ] with ID 7 is sending token to B
Information Message :: Node B with ID 9 is now parent of  D
Send Message :: Node B [ PHOLD ] with ID 9 is sending token to E
Information Message :: Node E with ID 8 is now parent of  B
```

4.3.4. <u>Output Tree Diagram</u>



** These screenshots are part of the output log files; the complete logs are available in the output folder.

5. <u>Remarks</u>

5.1. <u>Structure and Functionality:</u> The code implements a hierarchical tree structure using Node objects to represent nodes with parent-child relationships. It reads input from a file, builds the tree structure using indentation levels, and manages node relationships using stacks and maps (and   ).

5.2. <u>Multithreading and Concurrency:</u> Multithreading is utilized to simulate concurrent processes (Node instances) in a distributed system. Each Node runs as a separate thread, interacting with its parent and potentially other nodes through token requests (and   methods).

5.3. <u>Error Handling and Logging:</u> The code includes basic error handling for file operations (   ) and thread interruptions (   ). Extensive logging   is used for debugging and providing information about node states, thread actions, and system messages.

5.4. <u>User Interaction and Input Handling:</u> AWT (FileDialog) is used for user interaction to select an input file, ensuring flexibility in choosing the hierarchical structure to simulate.

5.5. <u>State Management with Enums:</u> The Node class utilizes a Status enum   ) to manage node states, influencing node behavior during thread execution.

5.6. <u>Optimization and Performance Considerations:</u> The code could benefit from optimizations in terms of memory usage and thread management, especially for large tree structures or high numbers of concurrent nodes. Performance considerations include ensuring efficient traversal and synchronization among threads to avoid potential race conditions or deadlocks.

5.7. <u>User Instructions:</u> Before running the code, ensure the JDK is installed, and an appropriate IDE or text editor is set up for Java development. Prepare an input file formatted with nodes and indentation levels to simulate different hierarchical structures. Monitor the console output for detailed system messages, including node creation, thread actions, and final tree structure.

5.8. <u>Abort Functionality in Node.java:</u> In the Node.java class, the ABORT functionality manages node termination based on specific criteria, utilizing the Status for state management. Nodes transition to the ABORT state when conditions such as prolonged inactivity (tracked by) or system-wide failures (monitored via) occur. This feature is crucial for simulating realistic distributed system behaviors, where nodes may fail due to deadlock, resource exhaustion, or other critical issues. Upon entering the ABORT state, nodes cease operations, potentially terminating their threads and logging relevant termination reasons. This mechanism not only enhances the simulation's accuracy but also enables the exploration of recovery strategies and the impact of node failures on the overall distributed system dynamics.