OS PROJECT

By Abhinav (2203140) & Samuel (2203135)

Deadlock Detection and Visualization Using Resource Allocation Graphs with Semaphore-based Resource Management

Project Goals:

- Visualize Deadlocks: Use a Resource Allocation Graph to model and understand deadlocks.
- Implement Detection: Create an algorithm to detect cycles in the graph, indicating deadlocks.
- Demonstrate Semaphore Impact: Show how improper semaphore usage can cause deadlocks.

Key Features : -

1. Dynamic Resource Allocation Simulation:

- Use semaphores to model mutual exclusion, synchronization, and deadlock.
- Processes request and release resources at runtime.

2. Graph Representation:

- Dynamic Resource Allocation Graph (RAG) to model the system's state.
- Nodes: Model processes and resources.
- Edges: Modeling allocation (resource → process) or requests (process → resource).

3. Deadlock Detection Algorithm:

- Implement detection algorithm, for example using cycle detection in the RAG.
- Highlight deadlock conditions whenever cycles are detected.

4. Visualization:

- Animate the changes as processes request and release resources.
- Highlight deadlocks and propose resolutions such as process termination or preemption.

5. User Interaction:

- This will provide an interface to the user for adding or removing processes or resources.
- For manually triggering the resource requests/releases.
- For viewing the step-by-step detection of deadlocks.

6. Deadlock Handling:

- Propose strategies for handling detected deadlocks, for example:
- Resource preemption
- Process termination
- Wait for Graph Simplification.

Project Overview:

This project demonstrates how deadlocks occur in a system with semaphore-based resource management and showcases deadlock detection using a Resource Allocation Graph (RAG).
The project will include:

- A simulation of processes requesting and releasing resources.
- Representation of these interactions using a RAG.
- Implementation of a deadlock detection algorithm to identify cycles in the graph.
- Demonstration of how semaphore misuse can lead to deadlocks.

Implementation Plan:

Programming Language:

- C++ (or any preferred language like Python or Java).

Key Features:

- **Graph Representation:** Use an adjacency list to represent the RAG.
- **Semaphore Operations:** Simulate semaphore wait() and signal() operations to allocate and release resources.
- **Deadlock Detection:** Use DFS to detect cycles in the RAG.
- **Visualization:** Print the RAG or generate a graphical representation (e.g., using Graphviz).

Code Walkthrough:

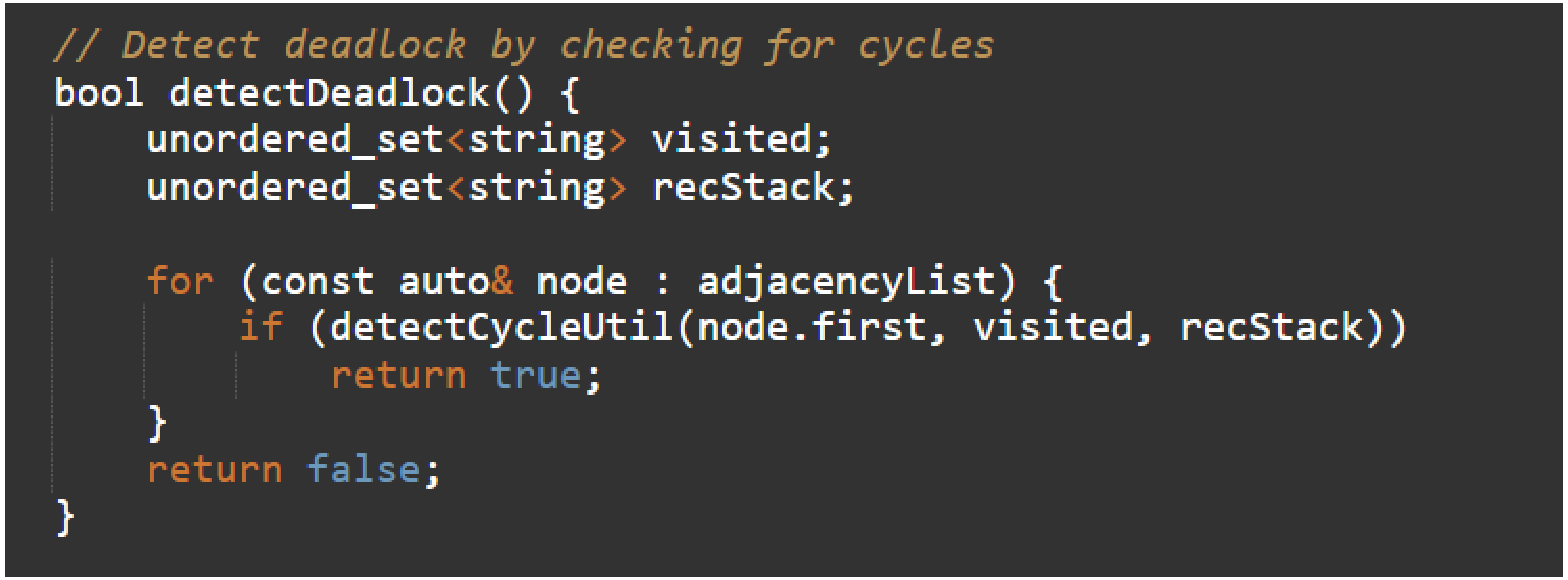
Core Functions:

- **addNode(node):** Adds a process or resource to the RAG.
- **addEdge(from, to):** Adds an edge to the RAG when a process requests or is allocated a resource.
- **removeEdge(from, to):** Removes an edge when a process releases a resource.
- **detectDeadlock():** Checks for cycles in the RAG to detect deadlock.

CODE EXPLAINATION : -

1. Deadlock Detection (Cycle Detection :-

- This recursive function detects cycles using Depth-First Search (DFS).
- Recursively traverses the graph to find back edges (indicative of cycles).
- Uses two sets:
- visited: Keeps track of all visited nodes.
- recStack: Keeps track of the current recursive stack to detect cycles.

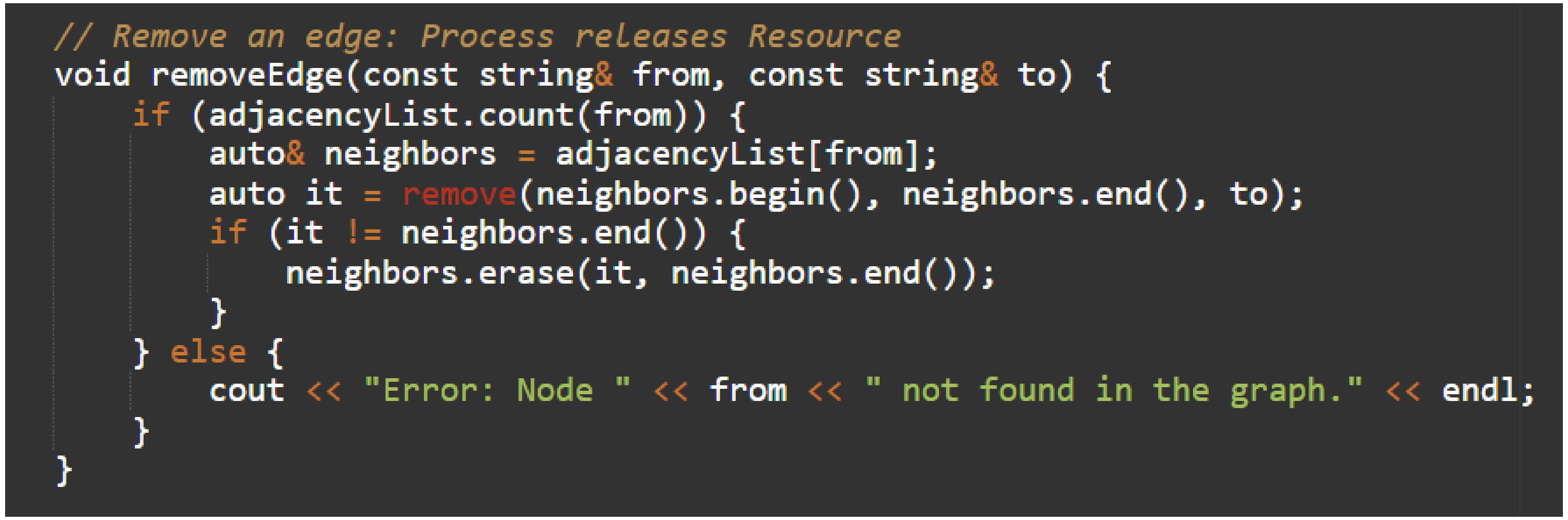


```
// Detect deadlock by checking for cycles
bool detectDeadlock() {
    unordered_set<string> visited;
    unordered_set<string> recStack;

    for (const auto& node : adjacencyList) {
        if (detectCycleUtil(node.first, visited, recStack))
            return true;
    }
    return false;
}
```

2. Remove Edges (Release Resources):

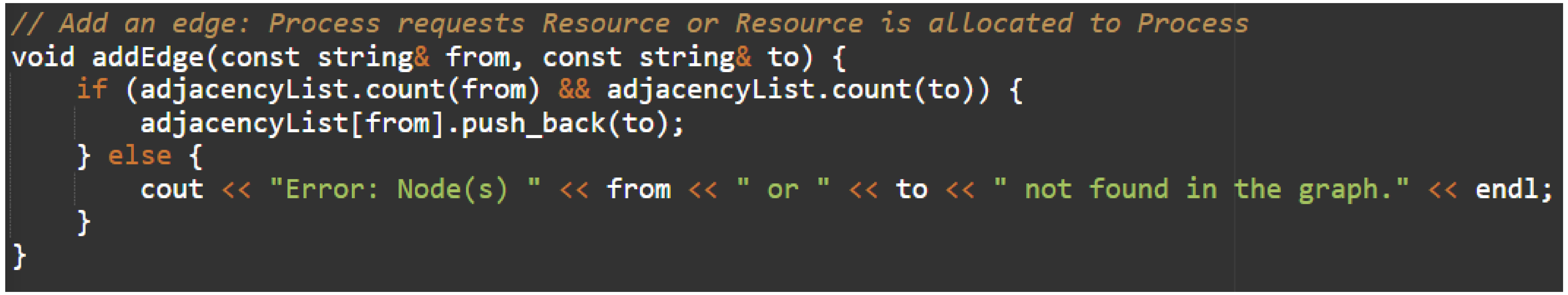
- This function removes edges when resources are released, indicating that a process has finished using a resource.
- Updates the graph by removing the relationship between a process and a resource.
- Simulates resource release.



```cpp
// Remove an edge: Process releases Resource
void removeEdge(const string& from, const string& to) {
    if (adjacencyList.count(from)) {
        auto& neighbors = adjacencyList[from];
        auto it = remove(neighbors.begin(), neighbors.end(), to);
        if (it != neighbors.end()) {
            neighbors.erase(it, neighbors.end());
        }
    } else {
        cout << "Error: Node " << from << " not found in the graph." << endl;
    }
}
```

3. Add Edges (Request or Allocation :

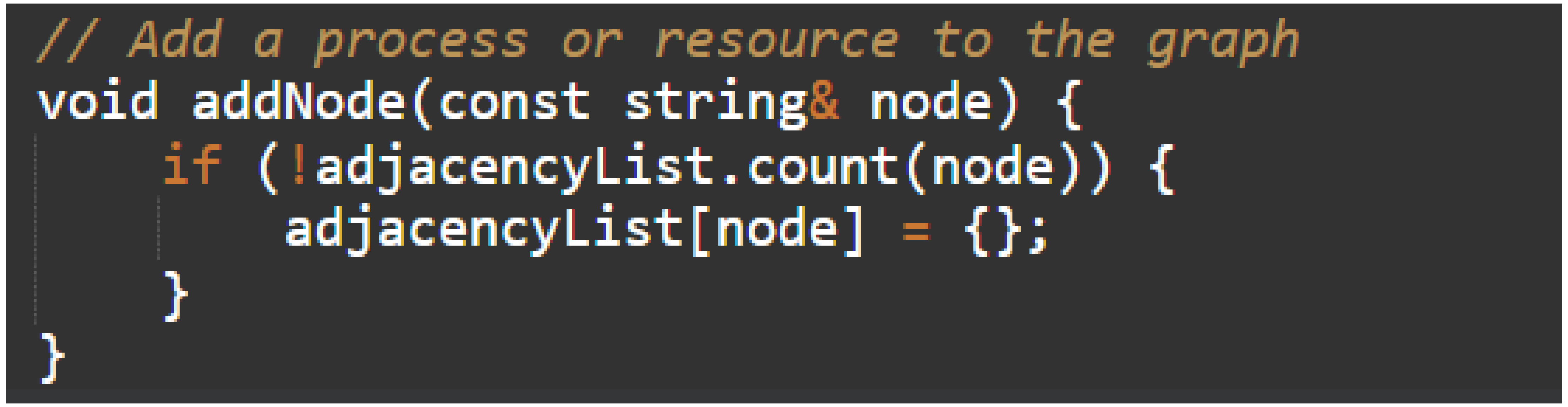
- This function connects a process and a resource. An edge from a process to a resource represents a request, and an edge from a resource to a process represents an allocation.
- Adds directed edges to the graph.
- Simulates the system's current state of resource requests or allocations.



```
// Add an edge: Process requests Resource or Resource is allocated to Process
void addEdge(const string& from, const string& to) {
    if (adjacencyList.count(from) && adjacencyList.count(to)) {
        adjacencyList[from].push_back(to);
    } else {
        cout << "Error: Node(s) " << from << " or " << to << " not found in the graph." << endl;
    }
}
```

4. Add Nodes to the Graph :

- This function ensures that processes and resources exist in the graph before adding edges.
- Adds a new process or resource to the graph.
- Prevents duplicate nodes.



```
// Add a process or resource to the graph
void addNode(const string& node) {
    if (!adjacencyList.count(node)) {
        adjacencyList[node] = {};
    }
}
```

THANK YOU