CSE211 DATA STRUCTURES

LAB 3 FALL 2024

QUEUE OPERATIONS

Prerequisites

Open the terminal and execute the following commands after downloading the `tarball` file:

```
cd /mnt/c/Users/user/Downloads && tar -xvf lab3_1.tar.gz --one-top-level=lab3_1
cd /mnt/c/Users/user/Downloads/lab3_1 && make all
code .
```

Introduction

In this lab, you will implement advanced operations on a Queue data structure using C++. The Queue is implemented as a template class that can store elements of any type T. Your task is to implement the following challenging operations:

1. `orangesRotting`: Simulate the spread of rotting through a grid of oranges
2. `taskScheduler`: Schedule tasks with cooldown periods
3. `jumpGame`: Determine if end position is reachable

Project Structure

```
.
├── bin/
│   └── queue
├── include/
│   ├── Queue.hpp
│   └── Color.hpp
├── obj/
│   ├── Queue.o
│   ├── Color.o
│   └── main.o
├── src/
│   ├── Queue.cpp
│   ├── Color.cpp
│   └── main.cpp
├── instructions.md
└── Makefile
```

Implementation Details

1. orangesRotting

- **Purpose**: Simulate rotting oranges spreading in a grid
- **Parameters**: 2D grid where 0=empty, 1=fresh orange, 2=rotten orange
- **Return**: Minimum time until all oranges rot, or -1 if impossible
- **Example**:

```
Input:  [[2,1,1],
        [1,1,0],
        [0,1,1]]
Output: 4  // Takes 4 minutes to rot all oranges

Step-by-step process:
Initial state (minute 0):
2 1 1    // 2=rotten, 1=fresh, 0=empty
1 1 0
0 1 1

After minute 1:
2 2 1    // Top-middle orange rots
2 1 0    // Left-middle orange rots
0 1 1

After minute 2:
2 2 2    // Top-right orange rots
2 2 0    // Middle-middle orange rots
0 1 1

After minute 3:
2 2 2
2 2 0
0 2 1    // Bottom-middle orange rots

After minute 4:
2 2 2
2 2 0
0 2 2    // Bottom-right orange rots

Final state reached after 4 minutes
All reachable fresh oranges have rotted
```

- **Explanation**:
  1. Rotten orange at (0,0) starts spreading
  2. Each minute, rot spreads to adjacent fresh oranges
  3. Adjacent means up, down, left, or right (no diagonals)
  4. Process continues until no more fresh oranges can rot
  5. Returns -1 if any fresh orange remains unreachable

2. taskScheduler

- **Purpose**: Schedule tasks with required cooldown periods
- **Parameters**: vector of tasks and cooldown period n
- **Return**: Minimum time to complete all tasks
- **Example**:

```
Input:  tasks = ['A','A','A','B','B','B'], n = 2
Output: 8
Schedule: A B _ A B _ A B  // '_' represents idle time

Input:  tasks = ['A','A','A','B','B','B','C','C','C'], n = 3
Output: 11
Schedule: A B C _ A B C _ A B C  // '_' represents idle time

Input:  tasks = ['A','A','A'], n = 2
Output: 7
Schedule: A _ _ A _ _ A  // Must wait 2 units between same tasks
```

3. jumpGame

- **Purpose**: Determine if last position is reachable
- **Parameters**: vector of integers representing maximum jump length
- **Return**: true if end is reachable, false otherwise
- **Example**:

```
Input:  [2,3,1,1,4]
Output: true  // Can jump: 0->1->4

Step-by-step process:
Position: 0 1 2 3 4
Array:    2 3 1 1 4

Step 1: Start at index 0 (value=2)
- Can jump 1 or 2 steps forward
- Possible destinations: index 1 or 2
[2 3 1 1 4]
 ^
 Current position
 Can jump to positions 1,2

Step 2: Jump to index 1 (value=3)
- Can jump 1,2,or 3 steps forward
- Possible destinations: index 2,3,4
[2 3 1 1 4]
   ^
   Current position
   Can jump to positions 2,3,4
   Found path to end! (1->4)

Success path found: 0->1->4
Return: true
```

```
-----------------------------------------------

Input:  [3,2,1,0,4]
Output: false  // Can't reach end

Step-by-step process:
Position: 0 1 2 3 4
Array:    3 2 1 0 4

Step 1: Start at index 0 (value=3)
- Can jump 1,2,or 3 steps forward
- Possible destinations: index 1,2,3
[3 2 1 0 4]
 ^
 Current position
 Can jump to positions 1,2,3

Step 2: Try all possible jumps
From index 1 (value=2):
- Can reach indices 2,3
From index 2 (value=1):
- Can reach index 3
From index 3 (value=0):
- Can't jump forward
[3 2 1 0 4]
       ^
       Stuck here (value=0)
       Can't reach index 4

No path found to reach the end
Return: false
```

- **Explanation**:
  1. Start from first position (index 0)
  2. At each position, can jump from 1 up to the value at current position
  3. Use BFS to explore all possible jump combinations
  4. Return true if any path reaches last index
  5. Return false if no path can reach the end
  6. Key insight: Need to try all possible jump lengths, not just maximum

Testing

1. Build and run:

```
make clean  # Clean previous builds
make all    # Compile all files
make run    # Execute the program
```

Restrictions

❌ Do not modify:

- Queue.hpp interface
- main.cpp test cases
- Project structure
- Build system

❌ Do not use:

- External libraries
- Global variables
- Additional data structures (except where specified)

Academic Integrity

- Individual work only
- No code sharing
- No plagiarism
- Violations result in zero grade

Submission

1. Test thoroughly
2. Clean build files: `make clean`
3. Send only the `Queue.cpp` file to the course portal

Good luck with your implementation!