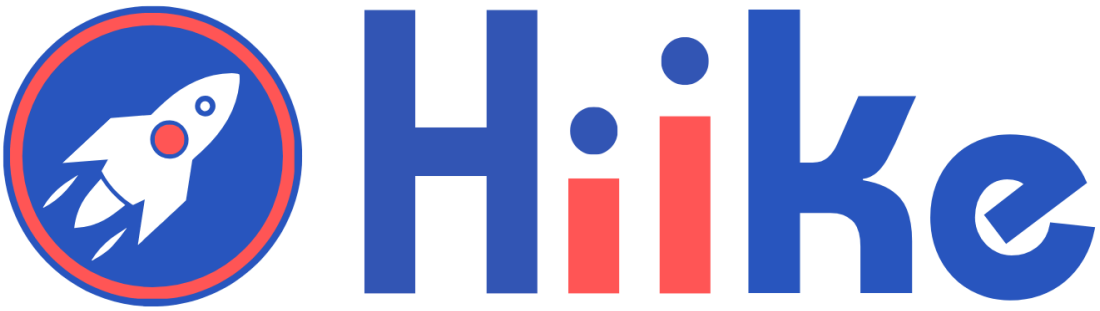
Queues

Detailed Notes



A queue is a fundamental data structure that operates based on the First In, First Out (FIFO) principle. This section provides a detailed overview of queues, their operations, and their application through example problems from LeetCode.

Characteristics of a Queue

FIFO Structure: The first element added to the queue will be the first one to be removed.

Operations:

- Enqueue: Add an element to the end of the queue.
- Dequeue: Remove the element from the front of the queue.
- Front/Peek: View the front element without removing it.
- IsEmpty: Check whether the queue is empty.
- Size: Return the number of elements in the queue.

Real-World Analogies

- A line of customers at a bank or ticket counter.
- Print queue in a computer system.

Implementation in Various Programming Languages

C++ (STL):

Use `#include<queue>`

- `std::queue<int> q;` // declaring a queue
- `q.push(10);` // adding an item (enqueue)
- `q.pop();` // removing the front item (dequeue)
- `q.front();` // accessing the front item
- `q.empty();` // checking if the queue is empty

Java:

Use `import java.util.LinkedList;` and `import java.util.Queue;`

- `Queue<Integer> queue = new LinkedList<>();` // declaring a queue
- `queue.add(10);` // adding an item
- `queue.remove();` // removing the front item
- `queue.peek();` // accessing the front item
- `queue.isEmpty();` // checking if the queue is empty

Python:

Use `from collections import deque`

- `queue = deque()` // declaring a queue
- `queue.append(10)` // adding an item (enqueue)
- `queue.popleft()` // removing the front item (dequeue)
- `queue[0]` // accessing the front item if queue is not empty
- `len(queue) == 0` // checking if the queue is empty

LeetCode Problems with Pseudocode

Example 2: Number of Recent Calls

Problem Statement: Write a class RecentCounter to count recent requests. It has only one method: ping(int t), where t represents some time in milliseconds. Return the number of pings that have been made from 3000 milliseconds ago until now. Any ping with time in [t - 3000, t] should count, including the current ping.

Pseudocode:

```
class RecentCounter:
    Initialize a queue

    function ping(t):
        Enqueue t into the queue
        while the front of the queue is less than t - 3000:
            Dequeue from the queue
        return the size of the queue
```

Why this works with a queue:

This problem uses a queue to keep track of the times of all recent calls within a sliding window of 3000 milliseconds before the current time t. As each new call time t is recorded with ping(t), older times more than 3000 ms ago are removed from the queue, and the size of the queue gives the count of recent calls. This showcases how queues can manage data efficiently in a sliding window or time frame context.

Example: Dota2 Senate

Problem Statement: In the world of Dota2, there are two parties: the Radiant and the Dire. The Dota2 senate consists of senators from these two parties. Given a string representing the sequence of senators, where 'R' represents a Radiant senator and 'D' represents a Dire senator, these senators sit in a circular senate. Each senator can ban one of the opposite party's senators from the senate during their turn, and the senators take turns in their original order. The simulation continues until one party cannot ban any more senators, meaning one party can eventually make all the decisions.

Your task is to predict which party will win the final control of the senate.

Pseudocode:

```
function predictPartyVictory(senate):
    Initialize two queues, radiant and dire
    for each index and senator in senate:
        if senator is 'R':
            radiant.enqueue(index)
        else:
            dire.enqueue(index)

    while radiant and dire queues are not empty:
        radiantIndex = radiant.dequeue()
        direIndex = dire.dequeue()
        if radiantIndex < direIndex:
            radiant.enqueue(radiantIndex + length of senate)  # Radiant
gets another turn in the next round
        else:
            dire.enqueue(direIndex + length of senate)  # Dire gets
another turn in the next round

    if radiant is empty:
        return "Dire"
    else:
        return "Radiant"
```

Why this works with a queue:

In the Dota2 Senate problem, the key challenge is to simulate a circular queue where senators from two parties (Radiant and Dire) take turns banning each other in a specific order. The solution uses two queues to manage the sequence of actions for senators from both parties. Each queue stores the indices of the senators in the order they get to act, representing Radiant and Dire senators, respectively. As we iterate through the queues, we compare the indices of the senators at the front of each queue to determine which senator acts first. The senator with the lower index bans the opponent senator first, ensuring that actions are taken in the correct cyclic order. After acting, the senator's index is incremented by the total number of senators and re-enqueued, simulating the circular nature of the seating and the game's rules. This process continues until one of the queues is empty, meaning all senators from one party have been banned, thereby determining the winning party. This approach efficiently models the cyclical and sequential dynamics of the senate's decision-making process, where each senator's ability to act or be banned directly impacts the outcome.