Queues

Day8 : 17/03/25

Queue : Simple Circular Array Implementation

```
struct CircularQueue {
    int *arr;
    int front;
    int rear;
    int capacity;
};

CircularQueue* createQueue(int capacity) {
    CircularQueue* q = new CircularQueue();
    q->capacity = capacity;
    q->arr = new int[capacity];
    q->front = -1;
    q->rear = -1;
    return q;
}

int size(CircularQueue* q) {
    if (q->front == -1) return 0;
    if (q->rear >= q->front) return q->rear - q->front + 1;
    return q->capacity - (q->front - q->rear - 1);
}

int isEmpty(CircularQueue* q) {
    return q->front == -1;
}

int isFull(CircularQueue* q) {
    return (q->rear + 1) % q->capacity == q->front;
}

int frontElement(CircularQueue* q) {
    if (isEmpty(q)) {
        cout << "Queue is empty." << endl;
        return -1;
    }
    return q->arr[q->front];
}

int rearElement(CircularQueue* q) {
    if (isEmpty(q)) {
        cout << "Queue is empty." << endl;
        return -1;
```

```
    }
    return q->arr[q->rear];
}

void enQueue(CircularQueue* q, int item) {
    if (isFull(q)) {
        cout << "Queue is full. Cannot enqueue." << endl;
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;
    }
    q->rear = (q->rear + 1) % q->capacity;
    q->arr[q->rear] = item;
}

int deQueue(CircularQueue* q) {
    if (isEmpty(q)) {
        cout << "Queue is empty. Cannot dequeue." << endl;
        return -1;
    }
    int item = q->arr[q->front];
    if (q->front == q->rear) {
        q->front = -1;
        q->rear = -1;
    } else {
        q->front = (q->front + 1) % q->capacity;
    }
    return item;
}

void deleteQueue(CircularQueue* q) {
    delete[] q->arr;
    delete q;
}
```

Queue : Dynamic Circular Array Implementation

```
#include <iostream>
using namespace std;

struct CircularQueue {
    int *arr;
    int front;
    int rear;
    int capacity;
```

```
};

CircularQueue* createQueue(int capacity) {
    CircularQueue* q = new CircularQueue();
    q->capacity = capacity;
    q->arr = new int[capacity];
    q->front = -1;
    q->rear = -1;
    return q;
}

int isEmpty(CircularQueue* q) {
    return q->front == -1;
}

int isFull(CircularQueue* q) {
    return (q->rear + 1) % q->capacity == q->front;
}

int size(CircularQueue* q) {
    if (isEmpty(q)) return 0;
    if (q->rear >= q->front) return q->rear - q->front + 1;
    return q->capacity - (q->front - q->rear - 1);
}

void resizeQueue(CircularQueue* q) {
    int newCapacity = q->capacity * 2;
    int *newArr = new int[newCapacity];
    int currentSize = size(q);

    for (int i = 0; i < currentSize; i++) {
        newArr[i] = q->arr[(q->front + i) % q->capacity];
    }

    delete[] q->arr;
    q->arr = newArr;
    q->capacity = newCapacity;
    q->front = 0;
    q->rear = currentSize - 1;
}

void enQueue(CircularQueue* q, int item) {
    if (isFull(q)) {
        resizeQueue(q);
    }
```

```
    if (isEmpty(q)) {
        q->front = 0;
    }
    q->rear = (q->rear + 1) % q->capacity;
    q->arr[q->rear] = item;
}

int deQueue(CircularQueue* q) {
    if (isEmpty(q)) {
        cout << "Queue is empty. Cannot dequeue." << endl;
        return -1;
    }
    int item = q->arr[q->front];
    if (q->front == q->rear) {
        q->front = -1;
        q->rear = -1;
    } else {
        q->front = (q->front + 1) % q->capacity;
    }
    return item;
}

int frontElement(CircularQueue* q) {
    if (isEmpty(q)) {
        cout << "Queue is empty." << endl;
        return -1;
    }
    return q->arr[q->front];
}

int rearElement(CircularQueue* q) {
    if (isEmpty(q)) {
        cout << "Queue is empty." << endl;
        return -1;
    }
    return q->arr[q->rear];
}

void deleteQueue(CircularQueue* q) {
    delete[] q->arr;
    delete q;
}
```

Queue : Linked List Implementation

```
struct Node {
```

```
    int data;
    Node* next;
}

struct LinkedListQueue {
    Node* front;
    Node* rear;
}

LinkedListQueue* createQueue() {
    LinkedListQueue* q = new LinkedListQueue();
    q->front = nullptr;
    q->rear = nullptr;
    return q;
}

bool isEmpty(LinkedListQueue* q) {
    return q->front == nullptr;
}

void enQueue(LinkedListQueue* q, int item) {
    Node* newNode = new Node();
    newNode->data = item;
    newNode->next = nullptr;
    if(isEmpty(q)) {
        q->front = newNode;
        q->rear = newNode;
    }
}

int deQueue(LinkedListQueue* q) {
    if(isEmpty(q)) {
        cout << "Queue is empty. Cannot dequeue.\n";
        return -1;
    }
    Node* temp = q->front;
    int item = temp->data;
    q->front = q->front->next;
    if(q->front == nullptr) {
        q->rear = nullptr;
    }
    delete temp;
    return item;
}
```

```
int frontElement(LinkedListQueue* q) {
    if(isEMpty(q)) {
        cout << "Empty Queue.\n";
        return -1;
    }
    return q->front->data;
}

int rearElement(LinkedListQueue* q) {
    if(isEmpty(q)) {
        cout << "Empty Queue.\n";
        return -1;
    }
    return q->rear->data;
}

void deleteQueue(LinkedListQueue* q) {
    while(!isEmpty(q)) {
        deQueue(q);
    }
    delete q;
}
```

Reversing a queue

```
void reverseQueue(CircularQueue* q) {
    if(isEmpty(q)) return;
    int item = deQueue(q);
    reverseQueue(q);
    enQueue(q, item);
}
```

Implement a queue using two stacks

```
// implementation of stack using arrays code first

struct QueueUsingStacks {
    Stack* s1;
    Stack* s2;
};

QueueUsingStacks* createQueue(int capacity) {
    QueueUsingStacks* queue = new QueueUsingStacks();
    queue->s1 = createStack(capacity);
    queue->s1 = createStack(capacity);
    return queue;
}
```

```
void enQueue(QueueUsingStacks* queue, int item) {
    push(queue->s1, item);
}

int deQueue(QueueUsingStacks* queue) {
    if (isStackEmpty(queue->s1) && isStackEmpty(queue->s2)) {
        cout << "Queue is empty. Cannot dequeue." << endl;
        return -1;
    }
    if (isStackEmpty(queue->s2)) {
        while (!isStackEmpty(queue->s1)) {
            push(queue->s2, pop(queue->s1));
        }
    }
    return pop(queue->s2);
}
```

Implement a stack using two queues

```
struct Queue {
    int *arr;
    int front;
    int rear;
    int capacity;
};

Queue* createQueue(int capacity) {
    Queue* queue = new Queue();
    queue->capacity = capacity;
    queue->front = -1;
    queue->rear = -1;
    queue->arr = new int[capacity];
    return queue;
}

bool isQueueEmpty(Queue* queue) {
    return queue->front == -1;
}

bool isQueueFull(Queue* queue) {
    return (queue->rear + 1) % queue->capacity == queue->front;
}

void enQueue(Queue* queue, int item) {
    if (isQueueFull(queue)) {
```

```
        cout << "Queue is full. Cannot enqueue." << endl;
        return;
    }
    if (isQueueEmpty(queue)) {
        queue->front = 0;
    }
    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->arr[queue->rear] = item;
}

int deQueue(Queue* queue) {
    if (isQueueEmpty(queue)) {
        cout << "Queue is empty. Cannot dequeue." << endl;
        return -1;
    }
    int item = queue->arr[queue->front];
    if (queue->front == queue->rear) {
        queue->front = -1;
        queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % queue->capacity;
    }
    return item;
}

struct StackUsingQueues {
    Queue* q1;
    Queue* q2;
    int front;
    int rear;
};

StackUsingQueues* createStack(int capacity) {
    StackUsingQueues* stack = new StackUsingQueues();
    stack->q1 = createQueue(capacity);
    stack->q2 = createQueue(capacity);
    stack->front = -1;
    stack->rear = -1;
    return stack;
}

void push(StackUsingQueues* stack, int item) {
    enQueue(stack->q2, item);
    while (!isQueueEmpty(stack->q1)) {
        enQueue(stack->q2, deQueue(stack->q1));
```

```
    }
    swap(stack->q1, stack->q2);
}

int pop(StackUsingQueues* stack) {
    if (isQueueEmpty(stack->q1)) {
        cout << "Stack is empty. Cannot pop." << endl;
        return -1;
    }
    return deQueue(stack->q1);
}

int top(StackUsingQueues* stack) {
    if (isQueueEmpty(stack->q1)) {
        cout << "Stack is empty." << endl;
        return -1;
    }
    return stack->q1->arr[stack->q1->front];
}
```

Implement doubly ended queue (aka head-tail linked list)

```
struct Node {
    int data;
    Node* prev;
    Node* next;
};

struct Deque {
    Node* front;
    Node* rear;
    int size;
};

Deque* createDeque() {
    Deque* deque = new Deque();
    deque->front = nullptr;
    deque->rear = nullptr;
    deque->size = 0;
    return deque;
}

bool isEmpty(Deque* deque) {
    return deque->front == nullptr;
}
```

```
void insertFront(Deque* deque, int item) {
    Node* newNode = new Node();
    newNode->data = item;
    newNode->prev = nullptr;
    newNode->next = deque->front;

    if (isEmpty(deque)) {
        deque->rear = newNode;
    } else {
        deque->front->prev = newNode;
    }
    deque->front = newNode;
    deque->size++;
}

void insertRear(Deque* deque, int item) {
    Node* newNode = new Node();
    newNode->data = item;
    newNode->prev = deque->rear;
    newNode->next = nullptr;

    if (isEmpty(deque)) {
        deque->front = newNode;
    } else {
        deque->rear->next = newNode;
    }
    deque->rear = newNode;
    deque->size++;
}

int deleteFront(Deque* deque) {
    if (isEmpty(deque)) {
        cout << "Deque is empty. Cannot delete from front." << endl;
        return -1;
    }

    Node* temp = deque->front;
    int data = temp->data;

    deque->front = deque->front->next;
    if (deque->front == nullptr) {
        deque->rear = nullptr;
    } else {
        deque->front->prev = nullptr;
    }
```

```
    delete temp;
    deque->size--;
    return data;
}

int deleteRear(Deque* deque) {
    if (isEmpty(deque)) {
        cout << "Deque is empty. Cannot delete from rear." << endl;
        return -1;
    }

    Node* temp = deque->rear;
    int data = temp->data;

    deque->rear = deque->rear->prev;
    if (deque->rear == nullptr) {
        deque->front = nullptr;
    } else {
        deque->rear->next = nullptr;
    }
    delete temp;
    deque->size--;
    return data;
}

int getFront(Deque* deque) {
    if (isEmpty(deque)) {
        cout << "Deque is empty." << endl;
        return -1;
    }
    return deque->front->data;
}

int getRear(Deque* deque) {
    if (isEmpty(deque)) {
        cout << "Deque is empty." << endl;
        return -1;
    }
    return deque->rear->data;
}
```

Check if each successive pair of integers in a given stack of integers is consecutive or not

```
bool arePairsConsecutive(Stack* stack) {
    Queue* queue = createQueue(stack->capacity);
```

```
    bool isConsecutive = true;
    while (!isStackEmpty(stack)) {
        enQueue(queue, pop(stack));
    }
    while (!isQueueEmpty(queue)) {
        push(stack, deQueue(queue));
    }
    while (!isStackEmpty(stack)) {
        int first = pop(stack);
        enQueue(queue, first);

        if (!isStackEmpty(stack)) {
            int second = pop(stack);
            enQueue(queue, second);

            if (abs(first - second) != 1) {
                isConsecutive = false;
            }
        }
    }
    while (!isQueueEmpty(queue)) {
        push(stack, deQueue(queue));
    }

    return isConsecutive;
}
```

Interleaving two halves of a given queue

```
void interleaveQueue(Queue* queue) {
    int n = queue->size;
    if (n % 2 != 0) {
        cout << "Queue size must be even to interleave." << endl;
        return;
    }

    Stack* stack = createStack(n / 2);

    for (int i = 0; i < n / 2; i++) {
        push(stack, deQueue(queue));
    }

    while (!isStackEmpty(stack)) {
        enQueue(queue, pop(stack));
    }
```

```
    for (int i = 0; i < n / 2; i++) {
        enQueue(queue, deQueue(queue));
    }

    for (int i = 0; i < n / 2; i++) {
        push(stack, deQueue(queue));
    }

    while (!isStackEmpty(stack)) {
        enQueue(queue, pop(stack));
        enQueue(queue, deQueue(queue));
    }
}
```

Given an integer k and a queue, reverse first k elements of the queue

```
void reverseFirstKElements(Queue* queue, int k) {
    if (k > queue->size) {
        cout << "not possible as k is greater than the size of the
queue." << endl;
        return;
    }

    Stack* stack = createStack(k);

    for (int i = 0; i < k; i++) {
        push(stack, deQueue(queue));
    }

    while (!isStackEmpty(stack)) {
        enQueue(queue, pop(stack));
    }

    for (int i = 0; i < queue->size - k; i++) {
        enQueue(queue, deQueue(queue));
    }
}
```

Number of Recent Calls

We have a RecentCounter class that counts the number of requests within a certain time frame. Implement the RecentCounter class:

- **`RecentCounter()` initializes the counter with zero recent requests.**

- **`int ping(int t)` adds a new request at time `t`, where `t` represents some time in milliseconds, and returns the number of requests that happened in the past 3000 milliseconds.**

```
class RecentCounter {
private:
    queue<int> q;

public:
    RecentCounter() {}

    int ping(int t) {
        q.push(t);
        while (!q.empty() && q.front() < t - 3000) {
            q.pop();
        }
        return q.size();
    }
};
```

Design a Hit Counter
Implement a data structure that records the number of hits received in the past 5 minutes (300 seconds).

Functions:

- **`HitCounter()` initializes the hit counter.**
- **`void hit(int timestamp)` records a hit at a given `timestamp`.**
- **`int getHits(int timestamp)` returns the number of hits received in the past 5 minutes from `timestamp`.**

```
class HitCounter {
private:
    vector<int> timestamps;
    vector<int> frequencies;

public:
    HitCounter() : timestamps(300, 0), frequencies(300, 0) {}

    void hit(int timestamp) {
        int index = timestamp % 300;
        if (timestamps[index] != timestamp) {
```

```
            timestamps[index] = timestamp;
            frequencies[index] = 1;
        } else {
            frequencies[index]++;
        }
    }

    int getHits(int timestamp) {
        int count = 0;
        for (int i = 0; i < 300; i++) {
            if (timestamp - timestamps[i] < 300) {
                count += frequencies[i];
            }
        }
        return count;
    }
};
```