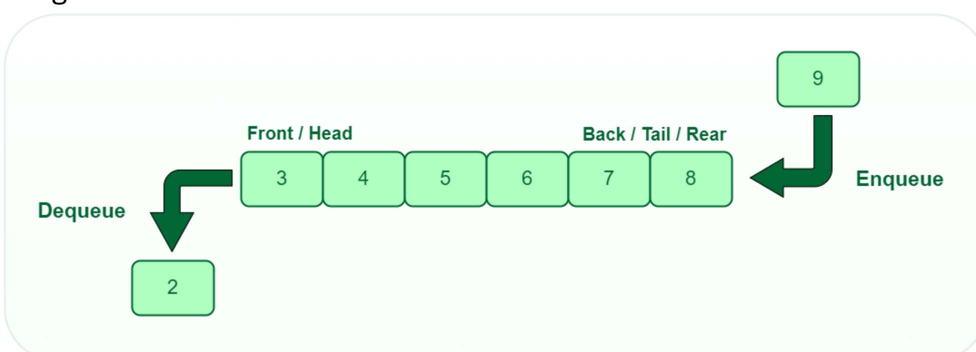
Queue – Use-Case Implementation

Members:

Student's Name	Student's ID
Thang Saoly	IDTB110257
Phon Sovan Ey	IDTB110218
Noy Sokbolen	IDTB110181
Rith Vichet	IDTB110140

>> Our Solution:

1. Scenario: **Bank teller simulation**
2. Implementation path: **Linked List Queue**
3. Report checklist
    a. Design
        i. The variant is a **linked list**.
        ii. Why it fits? Because Bank teller simulation scenario is **not fixed in size** – the number of people may vary each time.
        iii. Diagram:



    b. Methods: Defined API

```cpp
// Add a new customer to the rear of the queue
    void enqueue(string name, int numberOrder){
        Customer* newCustomer = new Customer;  // Allocate memory for new customer
        newCustomer->name = name;
        newCustomer->numberOrder = numberOrder;
        newCustomer->next = nullptr;

        // If the queue is empty, front and rear both point to the new customer
        if(rear == nullptr){
            front = rear = newCustomer;
            cout << name << " (" << numberOrder << ") enqueued to queue" << endl;
            return;
        }

        // Otherwise, link the new customer at the end and update rear
        rear->next = newCustomer;
        rear = newCustomer;

        cout << name << " (" << numberOrder << ") enqueued to queue" << endl;
    }
```

        i.

```cpp
// Remove and return the customer at the front of the queue
    Customer dequeue(){
        if(front == nullptr){  // Queue is empty
            cout << "Queue Underflow" << endl;
            return Customer(); // Return a default customer
        }

        Customer* temp = front;    // Store the node to be deleted
        front = front->next;       // Move front to the next element

        // If queue becomes empty after dequeue, update rear as well
        if(front == nullptr){
            rear = nullptr;
        }

        Customer data = *temp;     // Copy customer data
        delete temp;               // Free memory of dequeued node
        return data;               // Return the dequeued customer
    }

    // Return (peek) the customer at the front without removing
    Customer peek(){
```

ii.

```cpp
// Return (peek) the customer at the front without removing
    Customer peek(){
        if(front == nullptr){      // Queue is empty
            cout << "Queue is empty" << endl;
            return Customer();     // Return default customer
        }
        return *front;             // Return copy of front customer
    }
```

iii.

```cpp
// Check if the queue is empty
    bool isEmpty(){
        return front == nullptr;    // True if no elements
    }
```

iv.

v.

```
// Check if the queue is full
    bool isFull(){
        return false;            // Linked list queue can grow dynamically
    }
```

c. Edge case handled: Edge cases handle empty dequeue/peek; full on array; last-item removal in list is already handled in the method and for memory safety we defined a destructor so to free the memory.

```
// Destructor: free all nodes to avoid memory leaks
    ~Queue()
    {
        while (front != nullptr)
        {
            Customer *tmp = front;
            front = front->next;
            delete tmp;
        }
        rear = nullptr;
    }
```

d. Complexity
    i. Time complexity (per operation)
        1. Constructor: O(1).
        2. enqueue(name, numberOrder): O(1) pointer work (allocate one node). Cost to copy the name string is O(L) where L = length of name, so overall O(1 + L) ≈ O(L) dominated by string copy.
        3. dequeue(): O(1) pointer updates and delete. Copying the dequeued Customer into return value costs O(L) for the string, so O(1 + L) ≈ O(L).
        4. peek(): O(1) pointer access, plus O(L) to copy the Customer for return.
        5. isEmpty(), isFull(): O(1).
        6. Destructor (~Queue): O(n) to visit/delete every node; total cost includes destroying each Customer and its string contents (sum of string lengths).
    ii. Space complexity

        1. Per element: O(1) extra memory for the node (Customer struct: int + pointer + std::string storage). Total heap usage is O(n) for n enqueued elements (plus string storage).
        2. Additional queue object overhead: O(1) (two pointers front/rear).
        3. No internal fixed-size array, so no wasted reserved capacity in the queue itself.
    iii. Why this is acceptable
        1. Enqueue/dequeue are O(1) in pointer operations, which is the expected optimal behaviour for a queue - good for streaming or many small operations.
        2. Dynamic linked nodes mean the queue isn't artificially limited by capacity; isFull() correctly returns false.
        3. The O(L) string-copy cost is unavoidable if stored strings by value; acceptable for typical use.
        4. Destructor is O(n) and necessary to prevent leaks; doing it at object destruction is standard and acceptable.

e. Evidence of correctness

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   COMMENTS

PS D:\Code\Year2\C++\Week4\queue> cd "d:\Code\Year2\C++\Week4\queue\" ; if ($?) { g++ main.cpp -o main } ; if ($?) { .\main }
Bank teller simulation
Saoly (1) enqueued to queue
Ey (2) enqueued to queue
Bolen (3) enqueued to queue
Chet (4) enqueued to queue

Serving customers:
Next in line: Saoly (1)
Serving: Saoly (1)

Next in line: Ey (2)
Serving: Ey (2)

Next in line: Bolen (3)
Serving: Bolen (3)

Next in line: Chet (4)
Serving: Chet (4)

All customers served.
PS D:\Code\Year2\C++\Week4\queue>
```