Data Structures and Algorithm Analysis

9

Dr. Syed Asim Jalal
Department of Computer Science
University of Peshawar

Queue Implementation through Linked List

Queue implementation using Linked List

- In Linked List implementation data will be stored in Linked List Nodes.
- We will have two pointers Front and Rear.

- The question is Should we use Singly Linked List or Doubly Linked List?
- Should Front be pointing to Head of the Linked List and Rear point to the End of the Linked List or vice verse?

- These decisions need to be made for efficient operations of Queue – i.e. Enqueue and Dequeue operations.

- We will use singly linked and not doubly linked list as as we don't need traversal in both directions.
- For singly linked list the Insert operation works in constant time for both ends (front and end).
- For single linked list the Remove operation works in constant time only at the Front, and not in the End of the linked list.
  - Because in remove operation in singly list, we need to traverse all the way to the second last node to remove node from the end of the list.
- It, therefore, makes sense to make head of the linked list the Front of the queue for dequeue operations.
- Enqueue will be performed at the End of the linked list. So the End of the linked list become Rear of the Queue.,

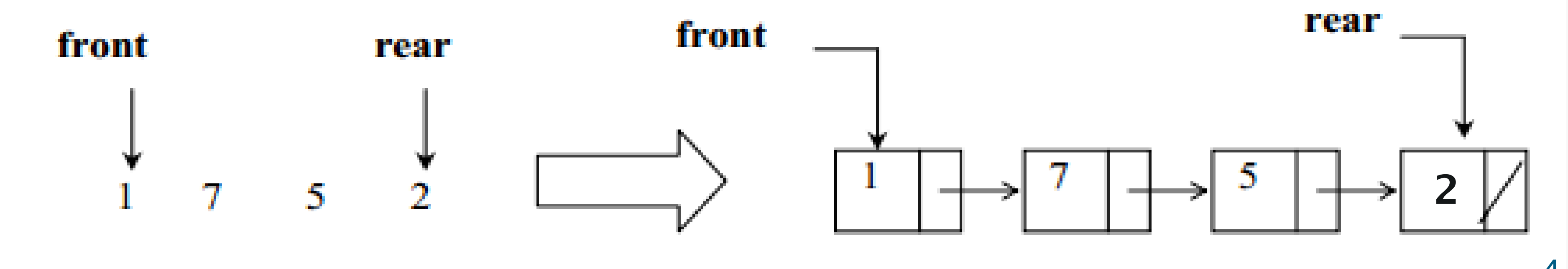


Fig 3. Queue implementation using linked list

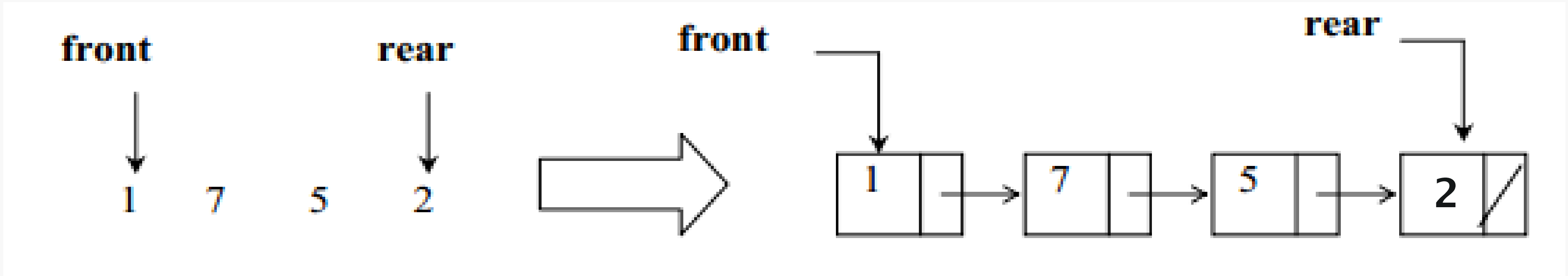


Fig 3. Queue implementation using linked list

- This figure shows queue implemented using linked list with front and rear pointers.
- The front element is removed with dequeue() operation.

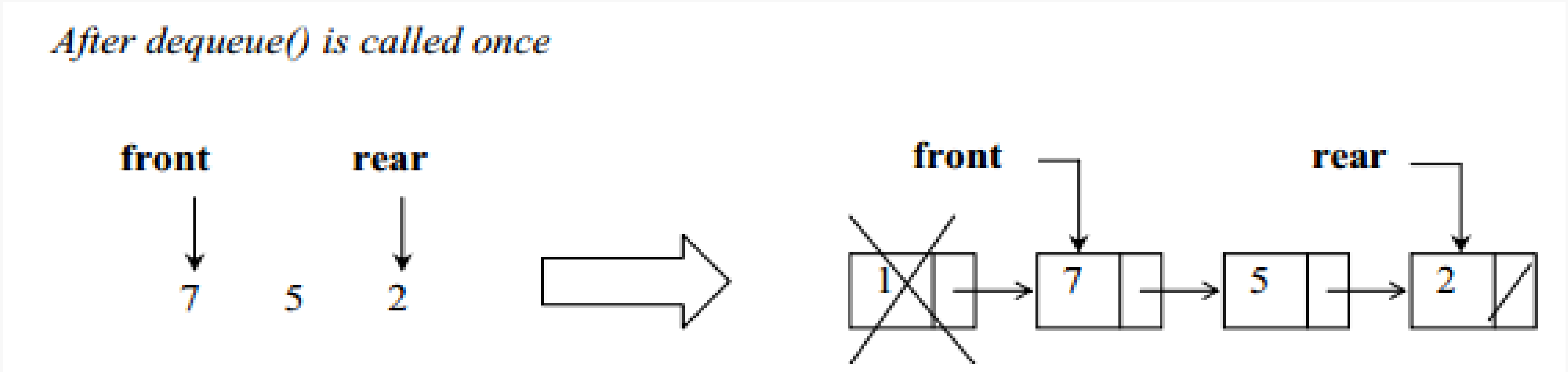
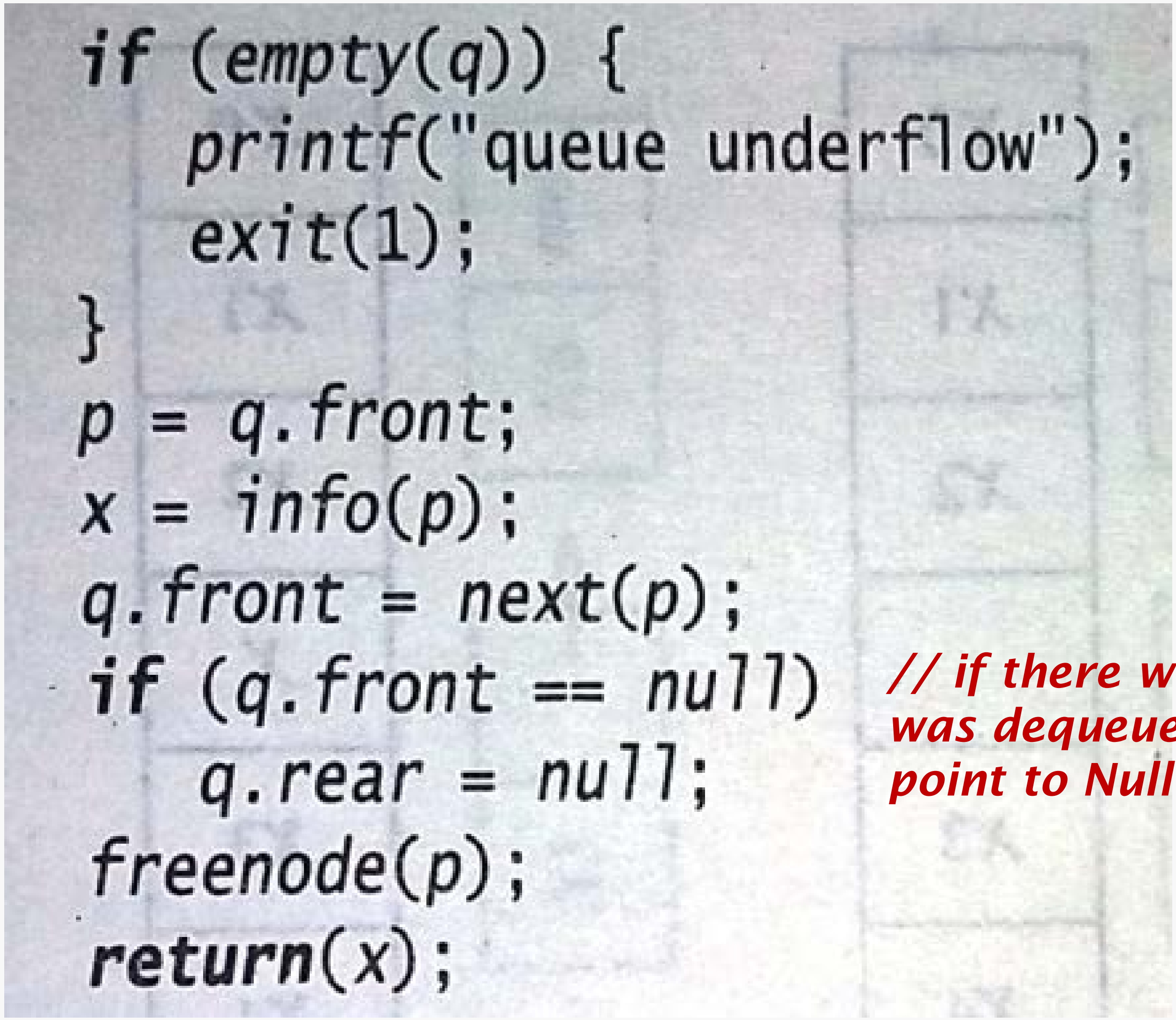


Fig 4. Removal of one element from queue using dequeue()

The figure of the queue showing one element removal is also depicted in the next figure. Note that the Front pointer has moved to the next element with value 7 in the list after removing the front element.

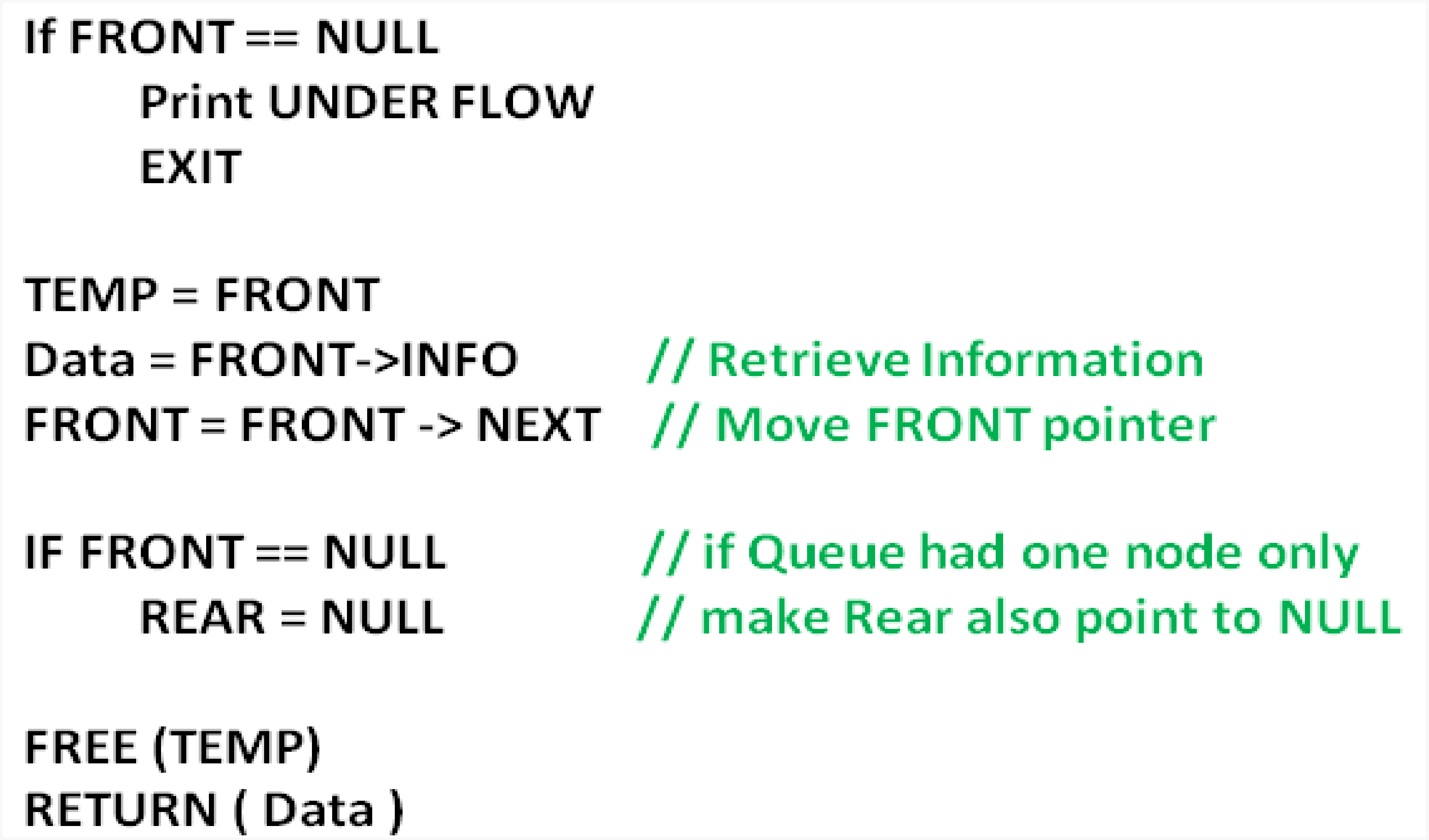
Delete / dequeue Algorithm



```
if (empty(q)) {
    printf("queue underflow");
    exit(1);
}
p = q.front;
x = info(p);
q.front = next(p);
if (q.front == null)
    q.rear = null;
freenode(p);
return(x);
```

// if there was only one Node, and it was dequeued then update the Rear point to Null as well.

Dequeue Algorithm Version 2



```
If FRONT == NULL
        Print UNDER FLOW
        EXIT

TEMP = FRONT
Data = FRONT->INFO          // Retrieve Information
FRONT = FRONT -> NEXT       // Move FRONT pointer

IF FRONT == NULL            // if Queue had one node only
        REAR = NULL         // make Rear also point to NULL

FREE (TEMP)
RETURN ( Data )
```

Queue after enqueue(9) call

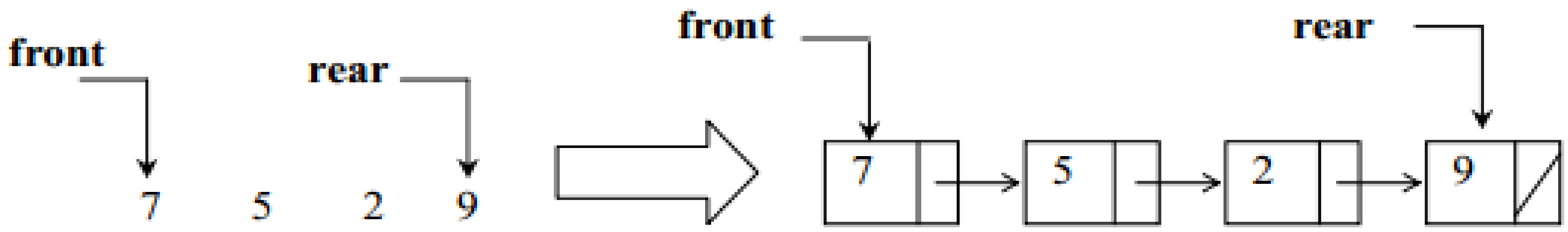
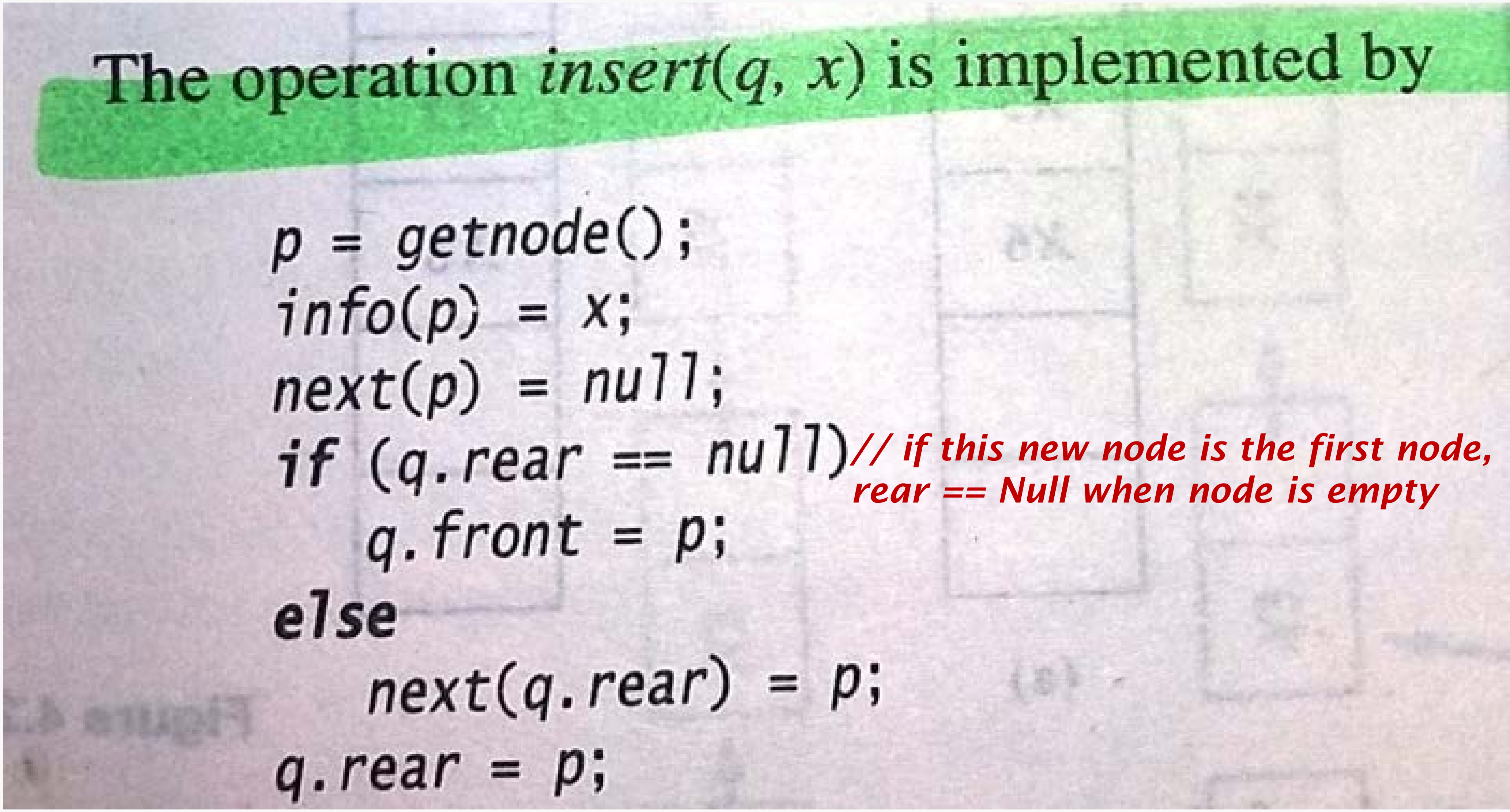


Fig 5. Insertion of one element using enqueue(9)

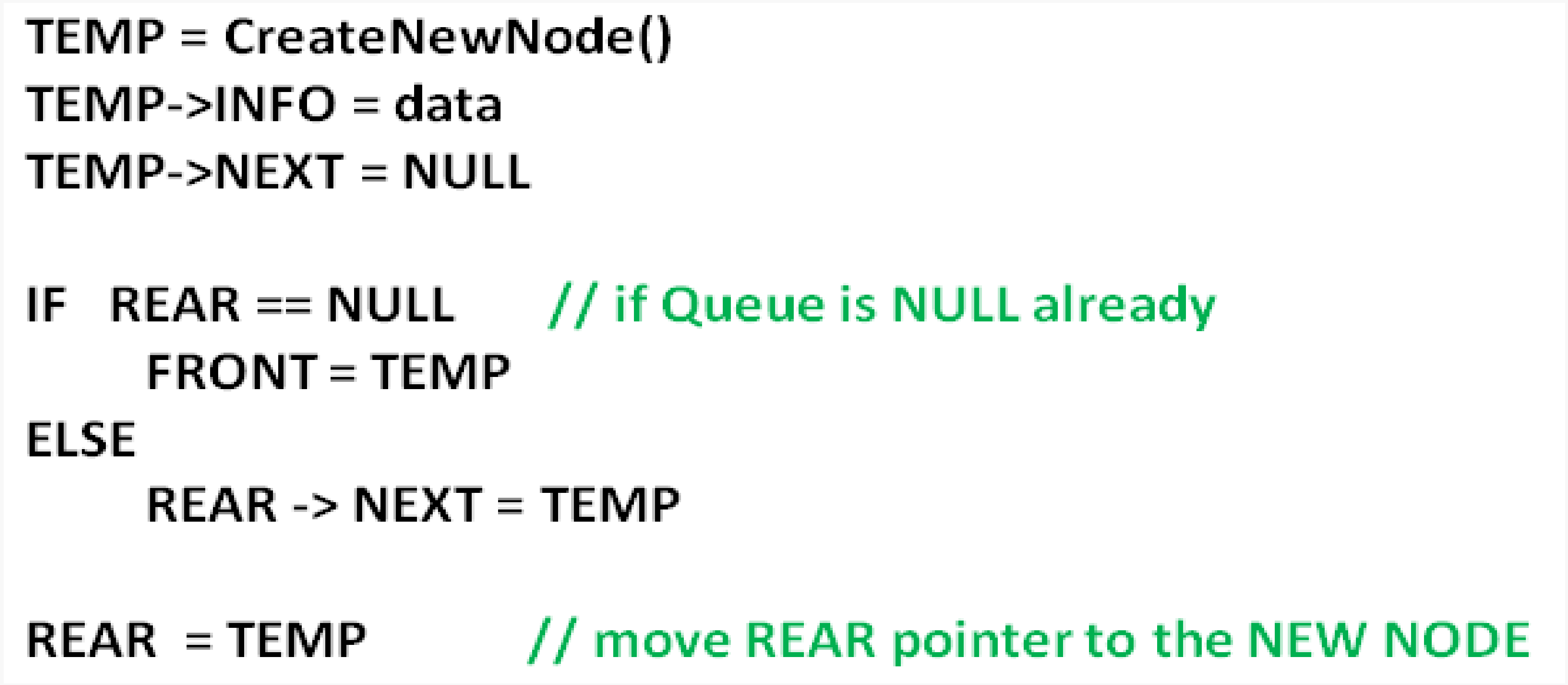
Insert / enqueue Algorithm



The operation *insert*(*q*, *x*) is implemented by

```
p = getnode();
info(p) = x;
next(p) = null;
if (q.rear == null)// if this new node is the first node,
                     rear == Null when node is empty
    q.front = p;
else
    next(q.rear) = p;
q.rear = p;
```

Enqueue Algorithm Version 2



```
TEMP = CreateNewNode()
TEMP->INFO = data
TEMP->NEXT = NULL

IF   REAR == NULL        // if Queue is NULL already
        FRONT = TEMP
ELSE
        REAR -> NEXT = TEMP

REAR  = TEMP            // move REAR pointer to the NEW NODE
```

Priority Queue

Priority Queue

- As stated earlier, the queue is a FIFO (First in first out) structure.
- Practically, all elements of a queue does not have equal priority. Some elements may have higher priority than others. This means elements with higher priority will leave earlier than elements with lower priority.
- Priority Queue is a queue where some elements have higher priority than other.
  - In Operating system the process scheduling is implemented by Priority Queue where some processes have higher priority than others.
- FIFO is a special case of priority queue in which priority is given to the time of arrival. Element that comes first has higher priority than remaining elements.

- Priority Queue is a data structure where each element has been assigned a priority and such that the order in which elements are deleted comes from the following rules:
  - 1. An elements of higher priority is processed before any element of lower priority
  - 2. Two elements with the same priority are processed according to the order in which they were added.

- Priority Queues are two types
  - Ascending priority queues
    - Priority queues where elements having smallest priority values is processed and removed first
  - Descending Priority Queues
    - Priority queues where elements have highest priority values are processed and removed first

A stack could be considered as Descending Priority Queue where elements added latest have greatest time value and are therefore deleted first.

- Priority Queue can be implemented in two ways
  - One way linked list
  - Multiple queues

One-way List Representation

- In list implementation the priority queue is implemented in the following way.
- Each node in the list will contain three items.
  - Information fields,
  - the priority field and
  - the link field
- A node X precedes a node Y in the list, when X has higher priority than Y or both have the same priority but X was added to the list before Y
- A lower priority number mean higher priority.

- In one ways list implementation of the priority queue is that an element with highest priority is always in the beginning of the list.
- This is done through Insert algorithm

- Simple Algorithm:
  - Add element ITEM with priority number N

1. **Traverse the one-way list until finding a node X whose priority number exceeds N. Insert the ITEM in front of node X**

2. **If no such node is found, insert ITEM as the last element of the list.**

Linked Implementation: INSERT Algorithm

This algorithm takes as input data value **Data** and priority **P** as input and inserts a new node in a priority queue.

Front pointer points to the start of the queue. There is no need for the rear pointer, as nodes are not added at the rear of the queue.

PREVIOUS and CURRENT are also the QUEUE pointers.

```
READ Data and P
NewNode = New Queue Node
NewNode→ Info = DATA
NewNode→PriorityN = P
If      FRONT  ==   NULL                              // Queue is empty
                FRONT = NewNode
                FRONT→next = NULL
ELSE   IF  NewNode→PriorityN   <     FRONT→Priority       //NewNode will be at Front
               NewNode → next = FRONT
               FRONT == NewNode
ELSE

    CURRENT = FRONT                                             // Traverse for right place
    Do While  NewNode→PriorityN   !<  Current→PriorityN   AND     Current → next != NULL
                  PREVIOUS= CURRENT
                  CURRENT = CURRENT → next


    IF     NewNode →PriorityN   <  CURRENT→PriorityN
                PREVIOUS→next  =NewNode              // the right place is next to PREVIOUS
                NewNode→ next = CURRENT
    ELSE                                                 // the right place is at the end
                CURRENT→next = NewNode
                NewNode→next = NULL
```

Linked Implementation: Delete Algorithm

This algorithm deletes element from Priority Queue from the front. START points to start of queue. DATA is the value to deleted.

```
If (Front  == NULL)
        Write Underflow
        Exit
Current = Front
DATA  = CURRENT -> INFO
Front =  Front → NEXT
FREE or DELETE Current
Exit
```