Lecture 2: Queues

EECG142- Data Structures

FIFO

Textbook:
Data Structures via C++: Objects by Evolution
by A. Michael Berman

First year - EECE Department

Spring 2025

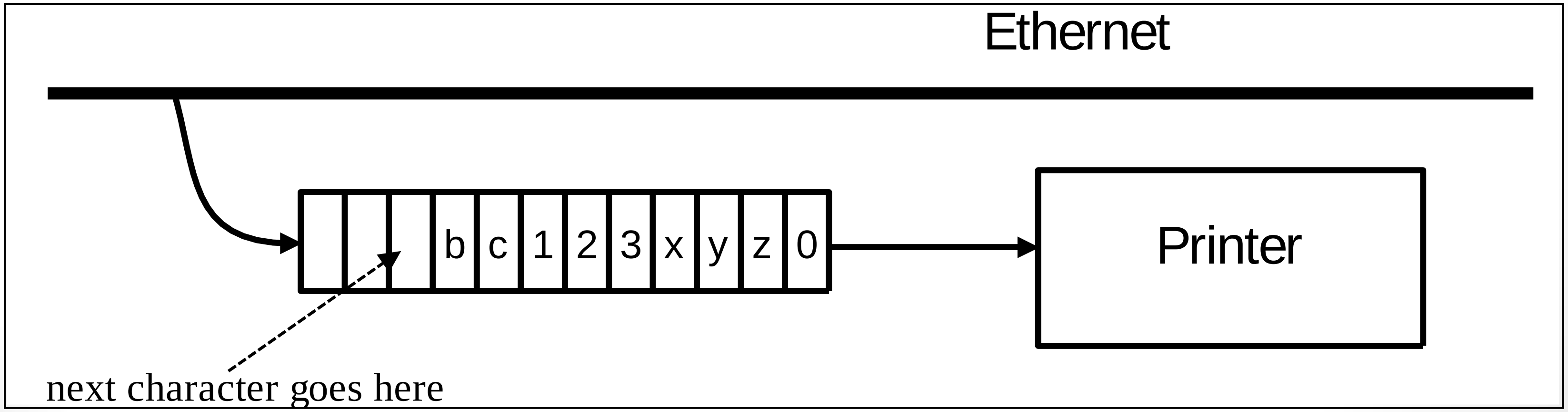
This lecture covers

1. Queue definition and examples
2. Queue implementation using linear arrays
3. Queue implementation using circular arrays
4. Queue implementation using linked-lists

Queue Example

Examples of queues include:

- people waiting in line for a movie or a shop.
- jobs waiting to be executed by a processors
- documents needed to be printed by a printer
- data in a buffer waiting to be transmitted

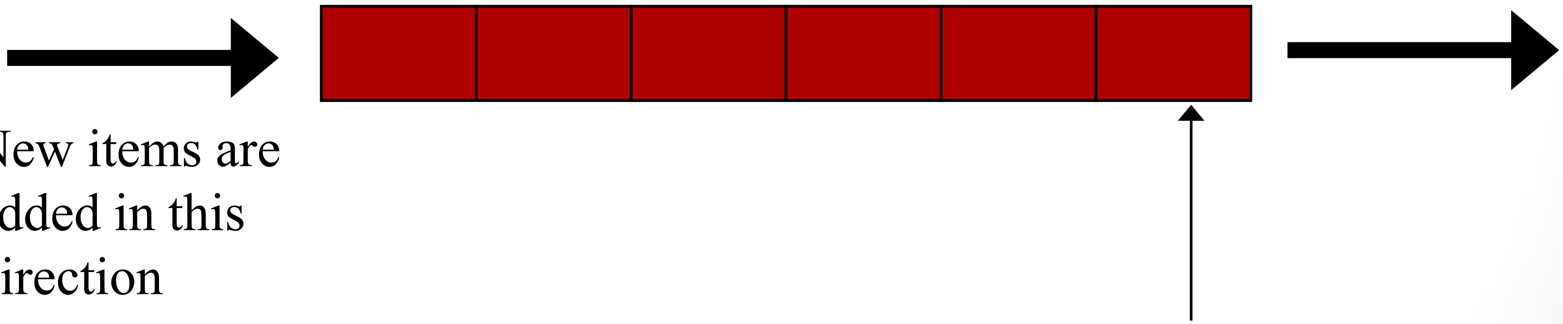


Queues

Unlike stacks, queues have a First in, first out (FIFO) property. Items are added to the rear and removed from the front

Items enter at rear and leave at front

Items are taken out from here

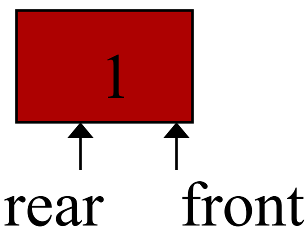


New items are added in this direction

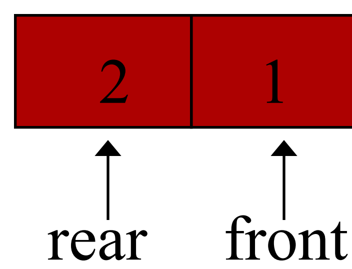
Front item has waited Longer than all other queue entries

Simple Queue Operations

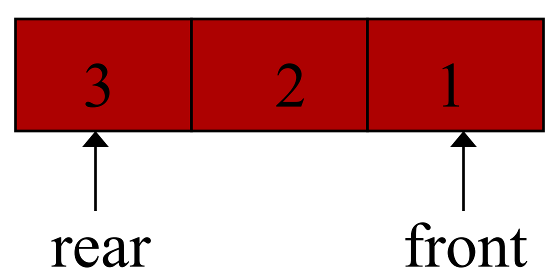
q.enqueue(1)



q.enqueue(2)



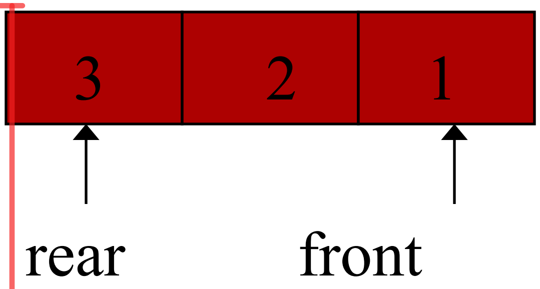
q.enqueue(3)



Simple Queue Operations

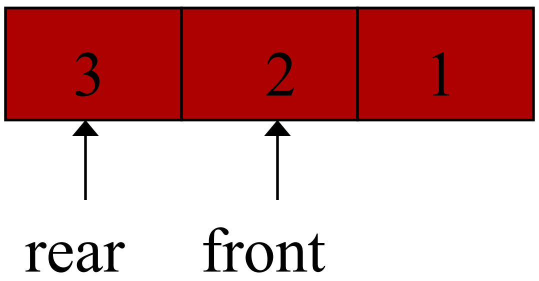
cout << q.front();

1



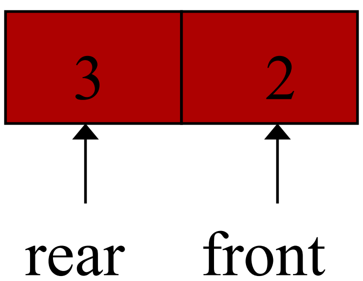
cout << q.dequeue();

1



cout << q.front();

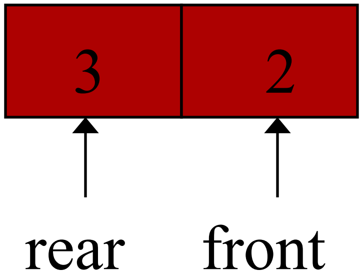
2



Simple Queue Operations

```
if (q.isEmpty())
  cout << “empty” << endl;
else
  cout << “not empty” << endl;
```

not empty



Array-based Implementation

- You need to keep track of the queue’s front and rear.
- The rear index of the queue will normally be higher than its front index with the exceptions
  - Empty queue (rear < front)
  - One-item queue (rear = front)
- A full queue has its rear index = array size-1

front -> entrance
rear -> exit

front -> first element
rear -> last element

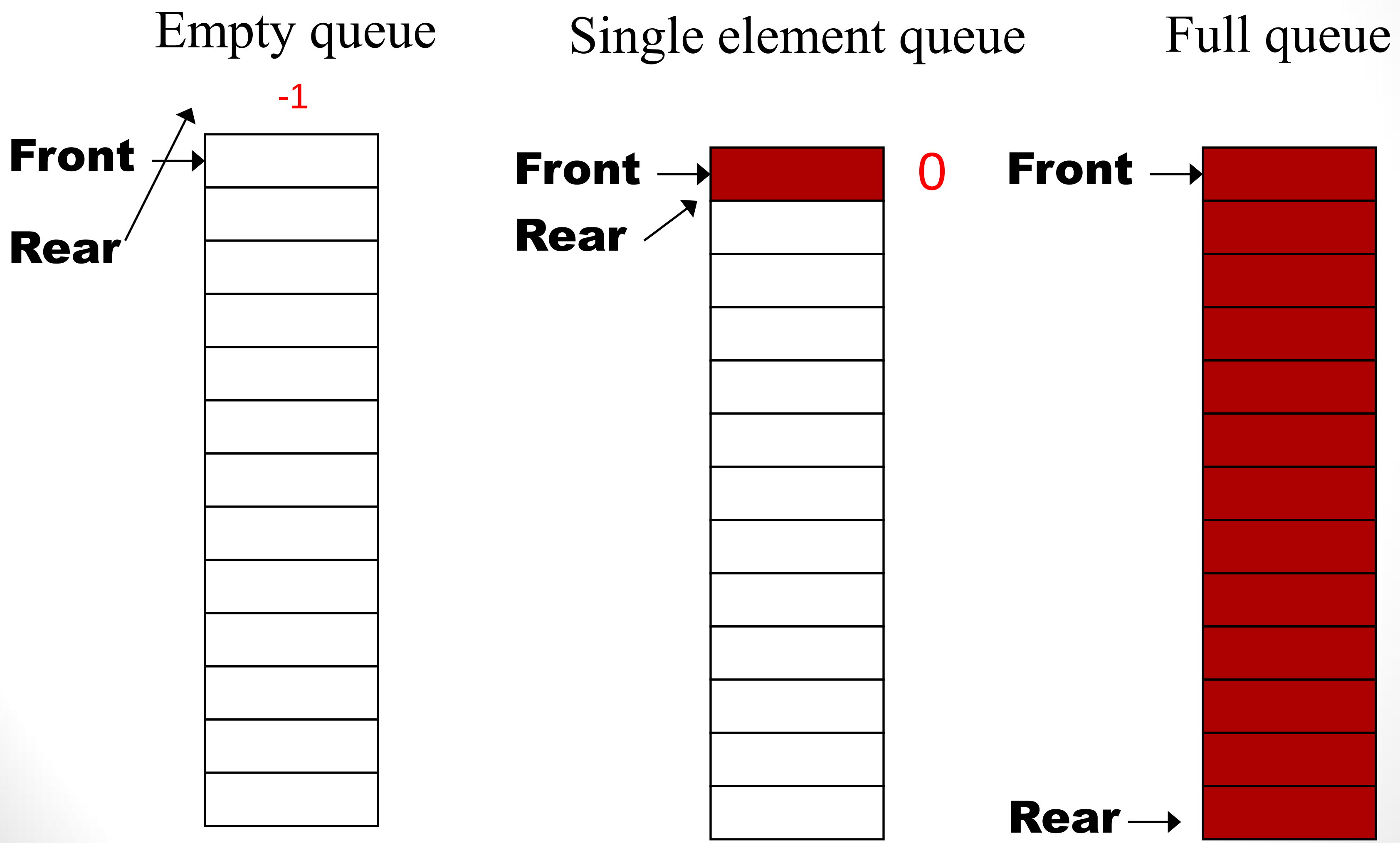
3 cases

1) rear < front empty

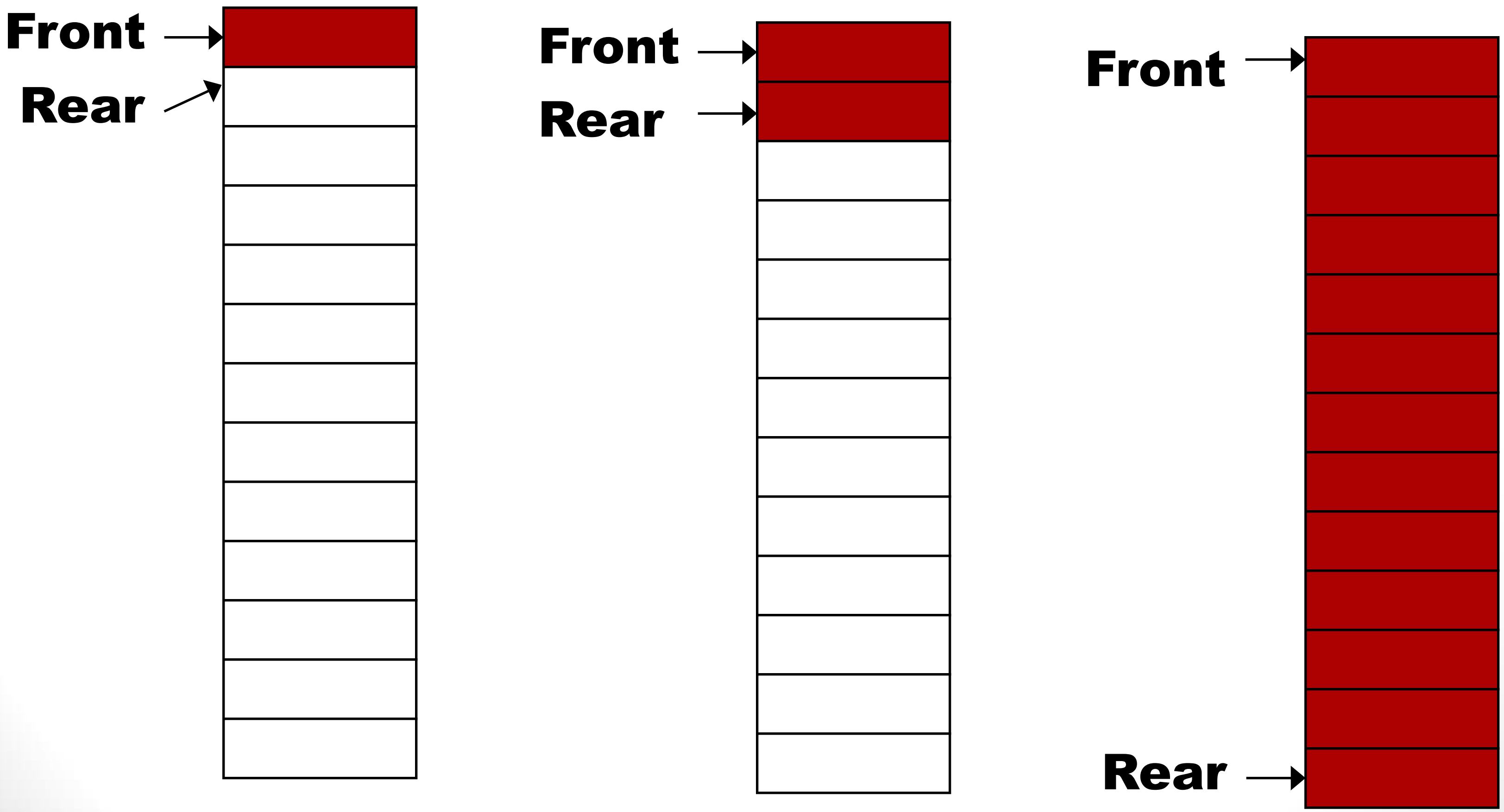
2) rear > front not empty

2.1) rear = front one element (always true)
2.2) rear = max-1 full

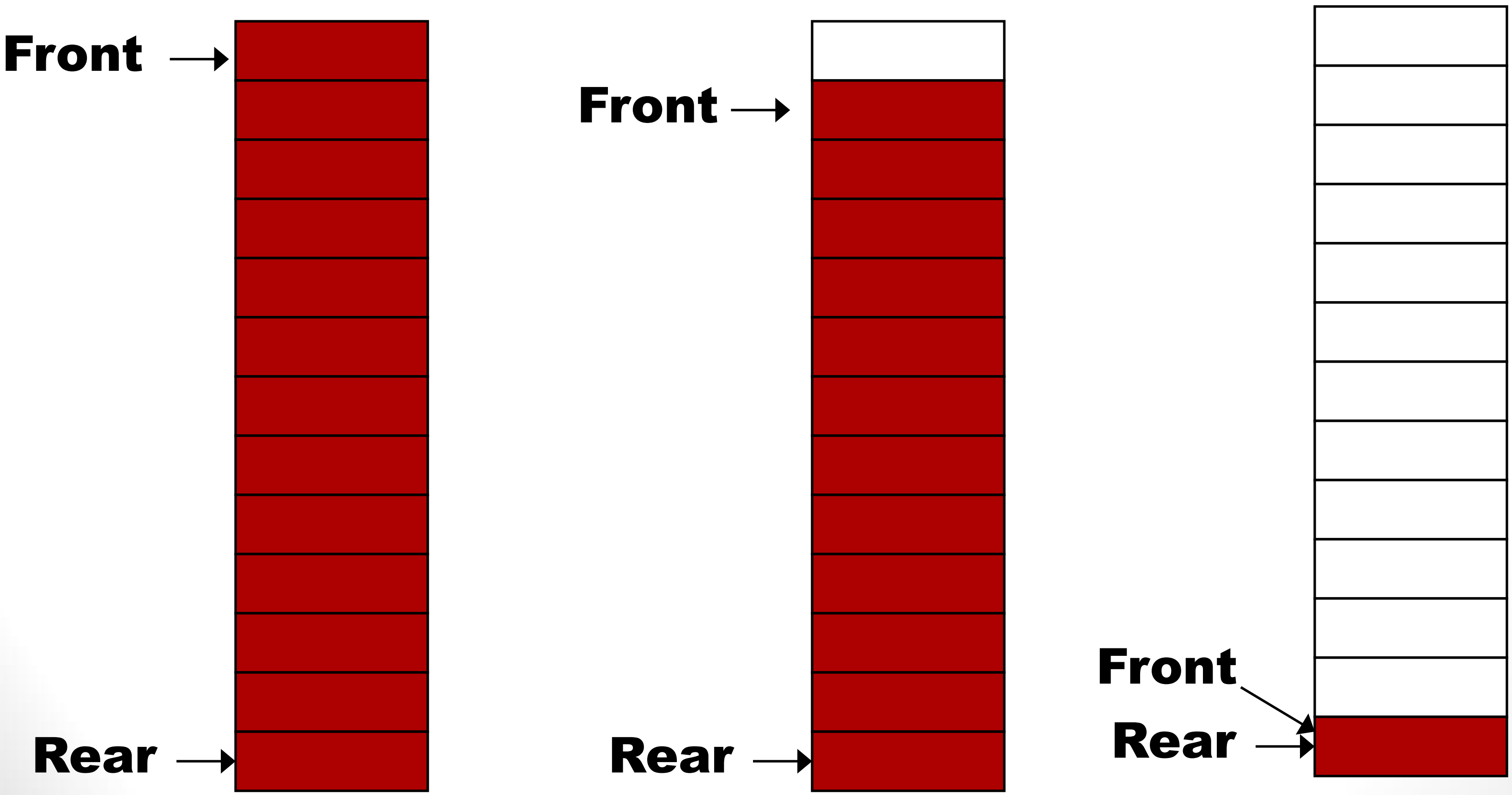
Special Cases of a Queue



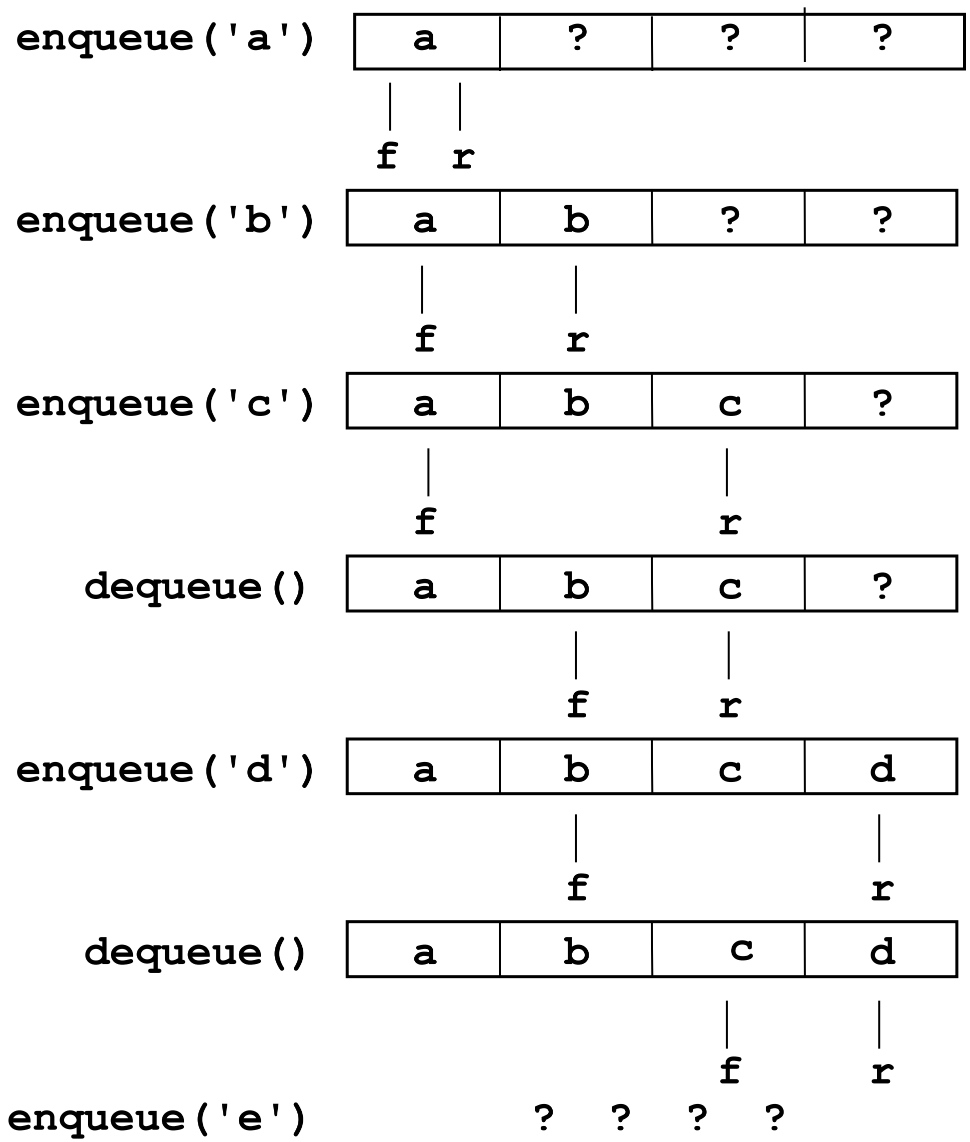
Enqueueing Items



Dequeuing Items

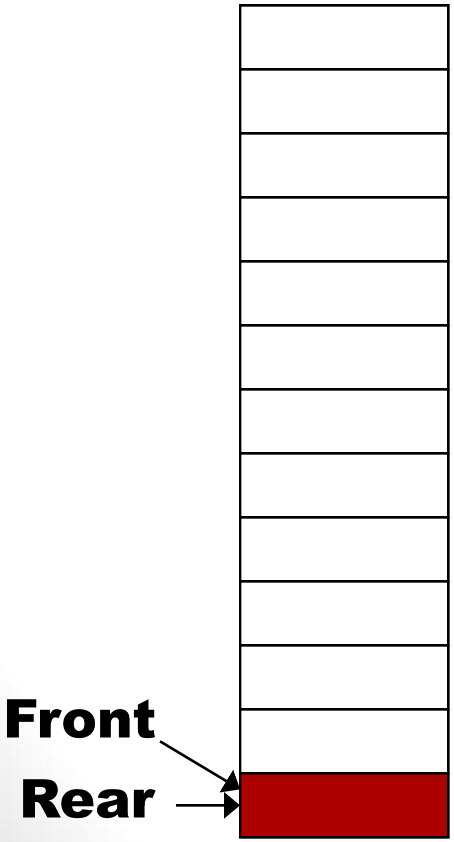


Implementation Problems



Implementation Problems

This single element queue is still considered full



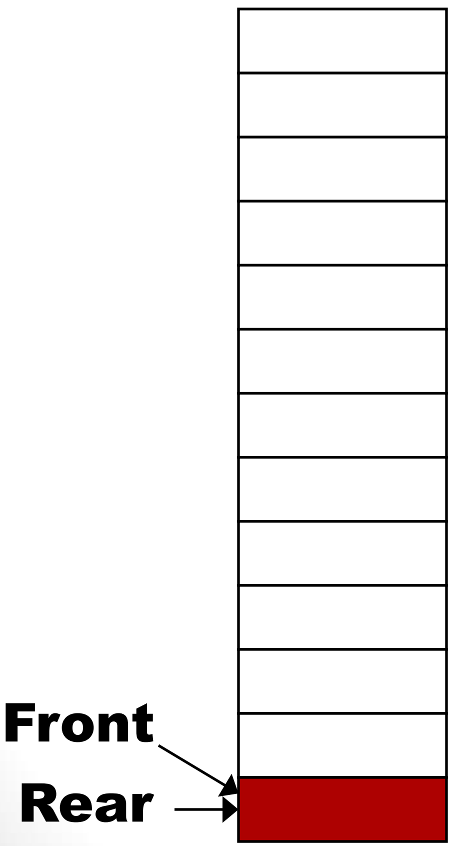
- Once the rear reaches the last index of the array, the queue is considered full and cannot accept more items even if there are empty cells after dequeuing

- We need to wait till the queue becomes empty and then reset the rear index.

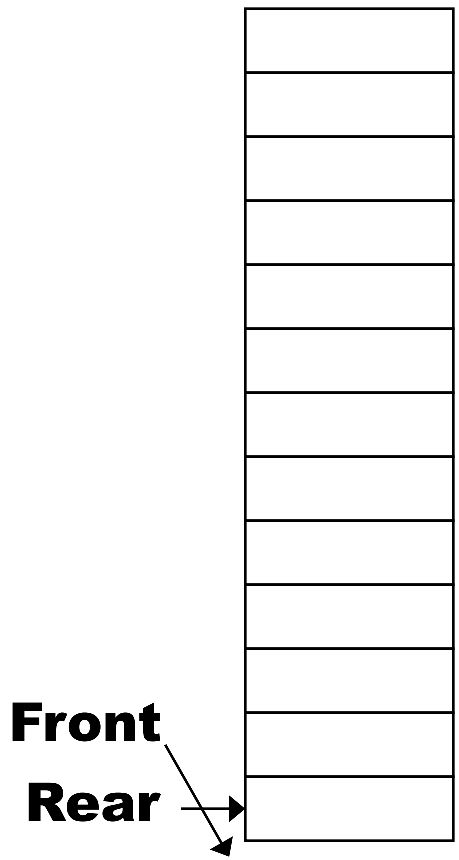
- Such way is inefficient and unacceptable in many applications

Implementation Problems

Single element queue



Is the queue empty or full?



Our definition of empty is rear < front, so it is empty.

But, rear has reached its limit. It cannot go beyond the array, Therefore, the queue is full!

Solution: Wrapping Around

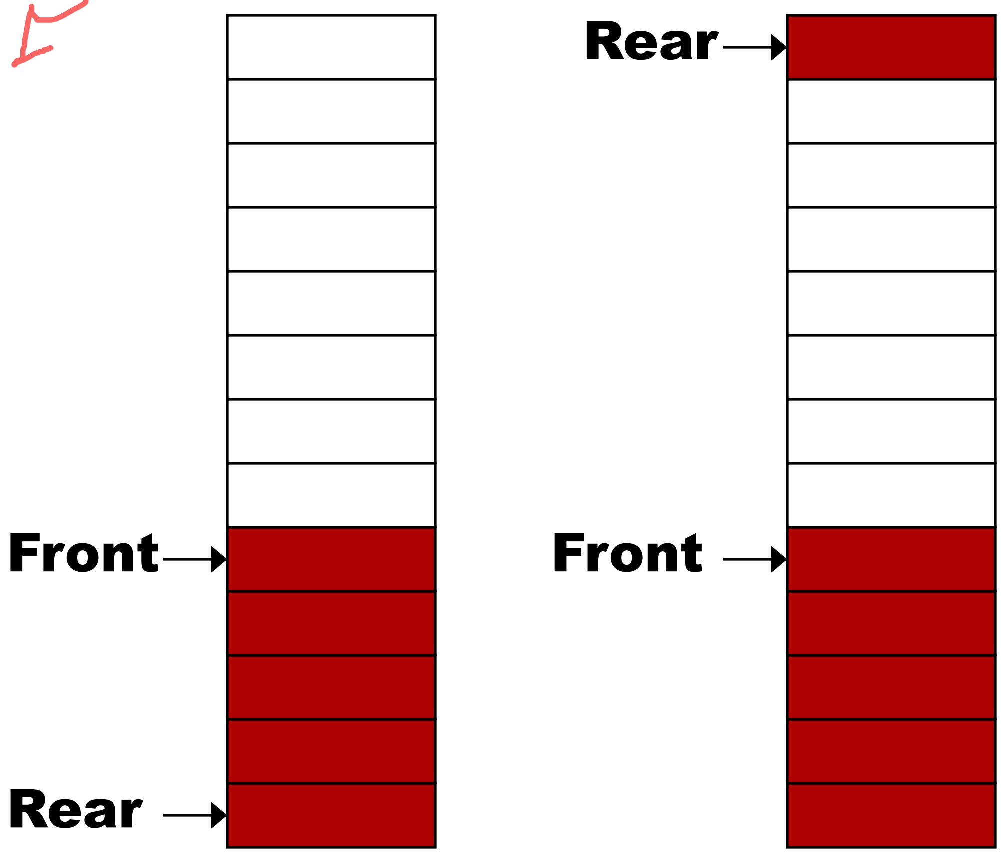
13 12

If rear + 1 > maxQueue -1, then rear = 0

13 13

rear = (rear+1) % maxQueue

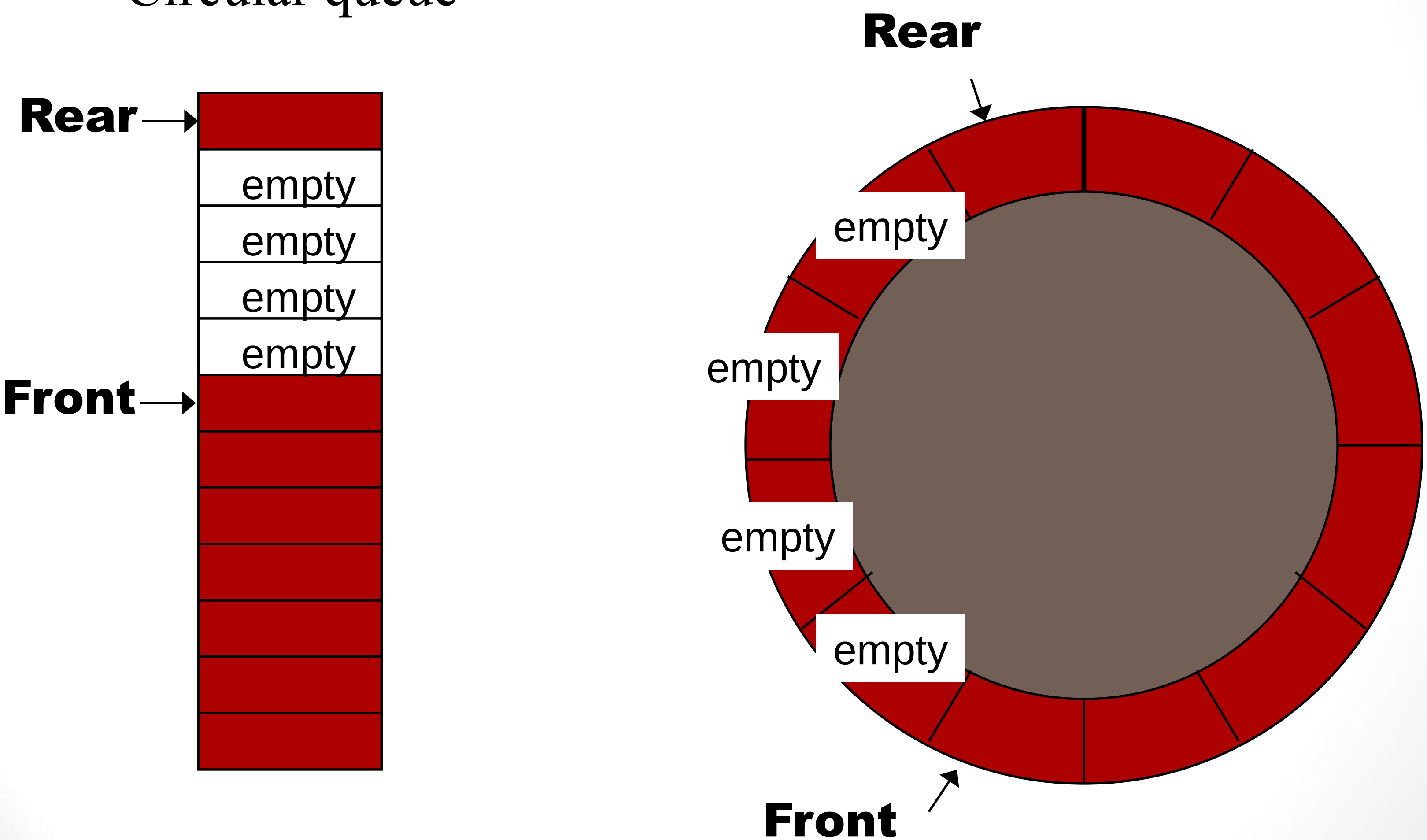
index



Next enqueue wraps around

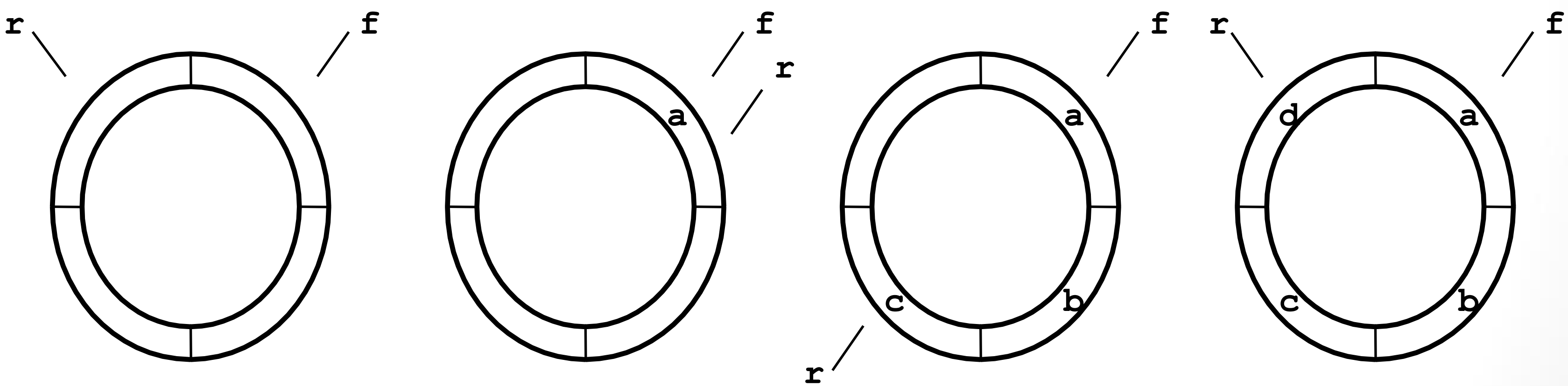
The Circular Queue

Circular queue

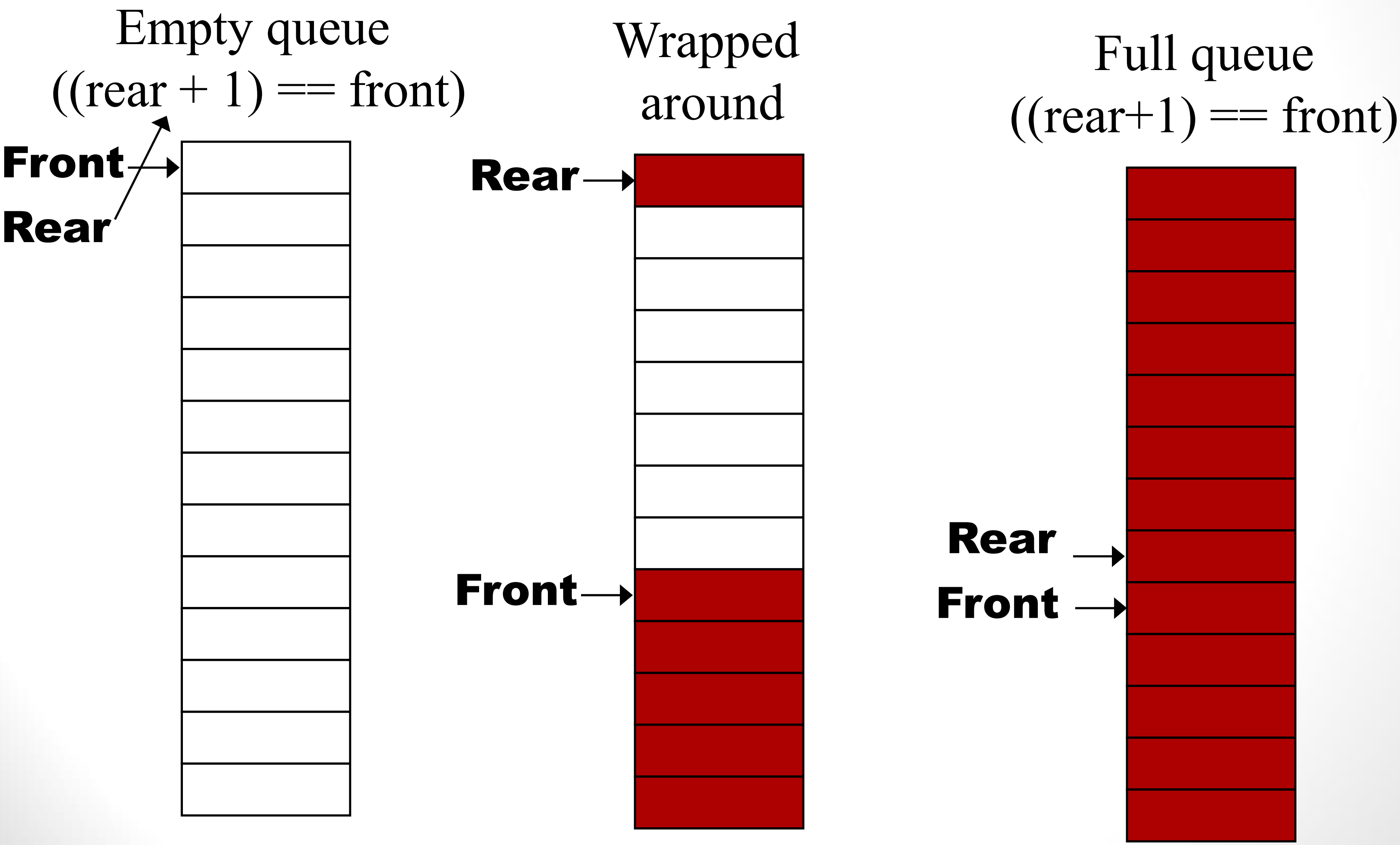


Is this queue empty or full?

We cannot use ‘rear < front’ as a test of empty circular queue because of the wrapping around.

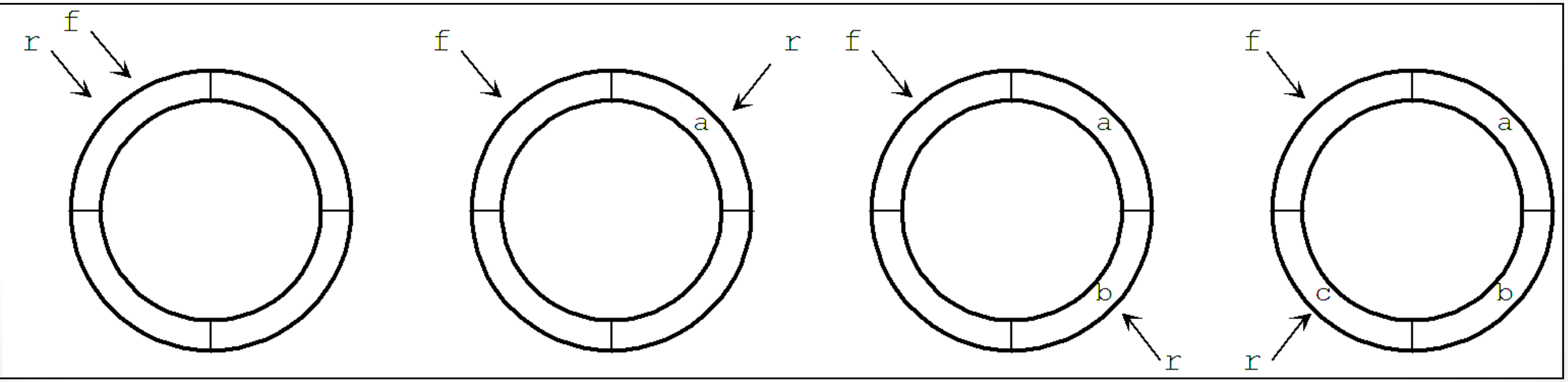


Explanation using Conventional Arrays

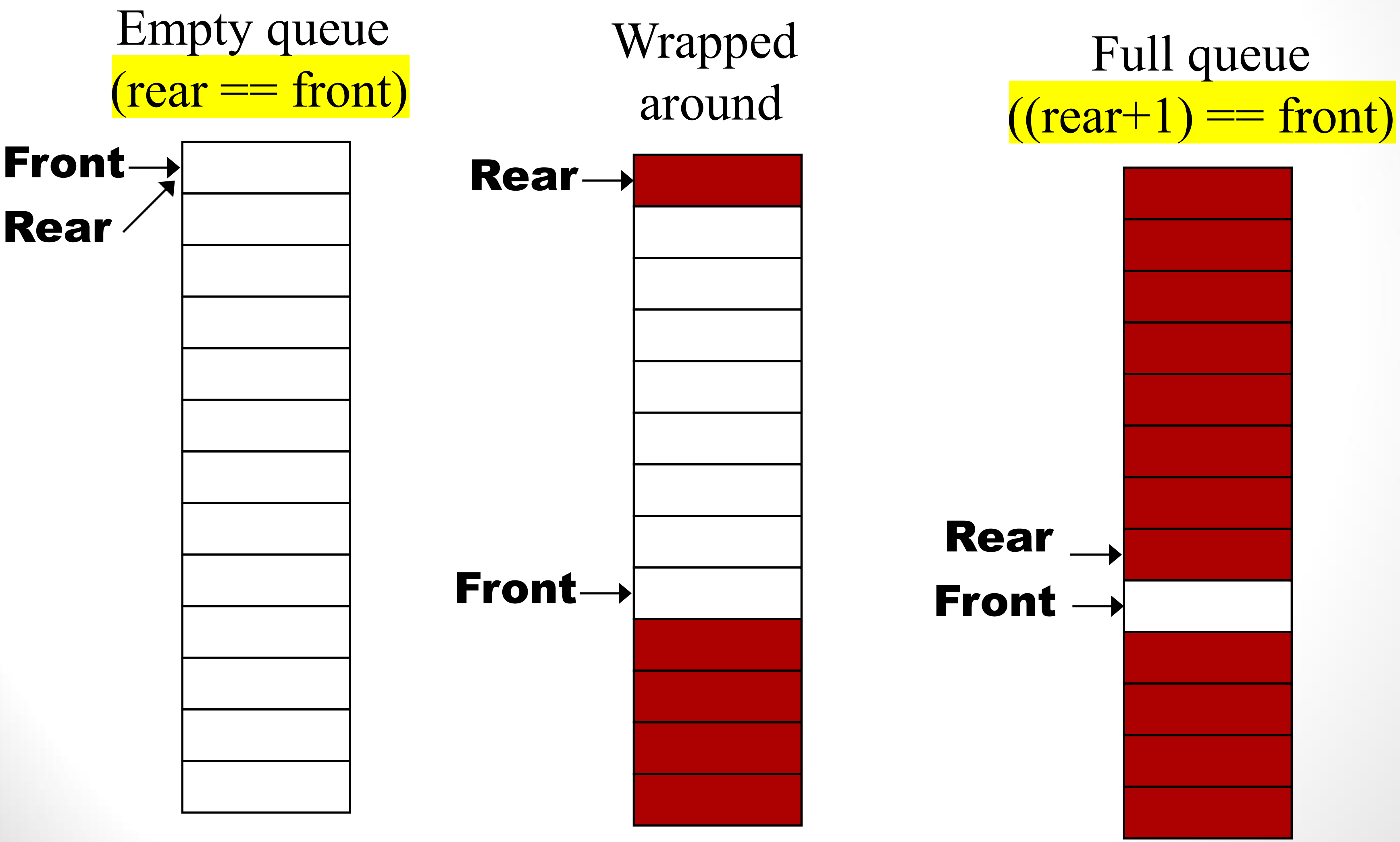


Empty or Full fix

- Let the front always point to an empty cell.
- Elements are added at nextPos(rear)
- An empty queue is defined by rear==front
- A full queue is defined by nextPos(rear) == front



Explanation using Conventional Arrays



Queue Header file

```
const int maxQueue = 200;
template < class queueElementType >
class Queue {
public:
   Queue();
   void enqueue(queueElementType e);
   queueElementType dequeue();
   queueElementType front();
   bool isEmpty();
private:
   int f; // marks the front of the queue
   int r; // marks the rear of the queue
   queueElementType elements[maxQueue];
```

empty :
Rear = front

full:
nextPos(Rear) = front

front :
before the start

Rear :
point to the end

enqueue :
1)check not full
2) rear++
3) put the elemnt

dequeue:
1)check not empty
2) front ++

any place ocupyed by front called before start and the place it left considered empty

Queue Header file (cntd.)

هاااام

neve never never
do this
f++ r++

always use nextPos funtion

```
int nextPos(int p)
{
  if (p == maxQueue - 1) // at end of circle
    return 0;
  else
    return p+1;
}
};
```

int nextpos(int p){ return (p+1)%MaxSize;}

wrapping around effect

Queue Implementation

```
template < class queueElementType >
Queue < queueElementType >::Queue()
{ // start both front and rear at 0
  f = 0;
  r = 0;
}
template < class queueElementType > bool
Queue < queueElementType >::isEmpty()
{ // return true if the queue is empty
  return bool(f == r);
}
```

Queue Implementation

```
template < class queueElementType >
Void Queue < queueElementType > ::
   enqueue(queueElementType e)
{   // add e to the rear and advance r
   assert(nextPos(r) != f);
   r = nextPos(r);
   elements[r] = e;
}
```

Queue Implementation

```
template < class queueElementType >
queueElementType
Queue < queueElementType >::dequeue()
{// advance the front and return the value at the front
   assert(f != r);
   f = nextPos(f);
   return elements[f];
}
```

Queue Implementation

```
template < class queueElementType >
queueElementType
Queue < queueElementType >::front()
{
   // return value of element at the front
   assert(f != r);
   return elements[nextPos(f)];
}
```

Header for Queue as Linked-list

```
template < class queueElementType >
class Queue {
public:
   Queue();
   void enqueue(queueElementType e);
   queueElementType dequeue();
   queueElementType front();
   bool isEmpty();
```

Private section

```
private:
Struct Node;
typedef Node * nodePtr;
struct Node {
     queueElementType data;
     nodePtr next;
  };
nodePtr f;    head
  nodePtr r;  tail
};
```

linked list

Linked-lists based implementation

```
template < class queueElementType >
Queue < queueElementType >::Queue()
{// set both front and rear to null pointers
f = NULL;
  r = NULL;
}
template < class queueElementType > bool
Queue < queueElementType >::isEmpty()
{// true if the queue is empty -- when f is a null pointer
  return bool(f == NULL);
}
```

Linked-lists based implementation

```
template < class queueElementType >
queueElementType
Queue < queueElementType >::front()
{
   assert(f); not pointing to null
   return f->data;
}
```

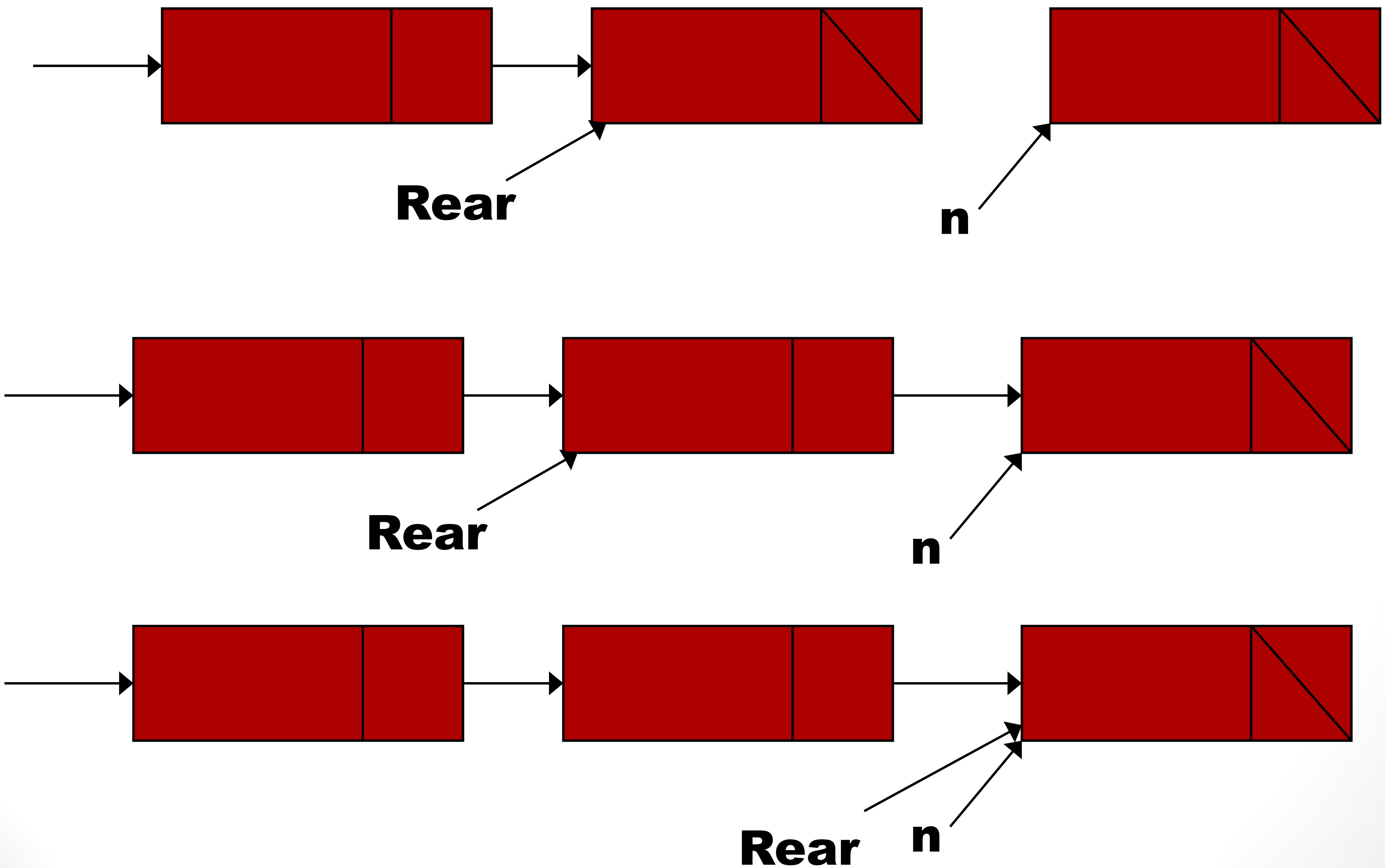
Linked-lists based implementation

```
template < class queueElementType >
void Queue < queueElementType > ::enqueue(queueElementType e)
{// create a new node, insert it at the rear of the queue
  nodePtr *n=new Node;
  assert(n);
  n->next = NULL;
  n->data = e;
  if (f != NULL) { // existing queue is not empty
    r->next = n; // add new element to end of list
    r = n;
  } else {// adding first item in the queue
    f = n; // so front, rear must be same node
    r = n;
  }
}
```

هاام

enque is the same as insert function

enqueueing



dequeue( )

```
template < class queueElementType > queueElementType
Queue < queueElementType >::dequeue()
{  assert(f); // make sure queue is not empty
queueElementType frontElement = f->data;
  nodePtr n=f;
f = f->next;
  delete n;
  if (f == NULL) // we're deleting last node
    r = NULL;
  return frontElement;
}
```

front move forward when dequeue
rear move forward when enqueue

in linked list implimintation its impossible that rear lags the front

dequeueing