EECE Department

ELC 303-B

Queues

Dr. Ahmed Khattab

Objectives

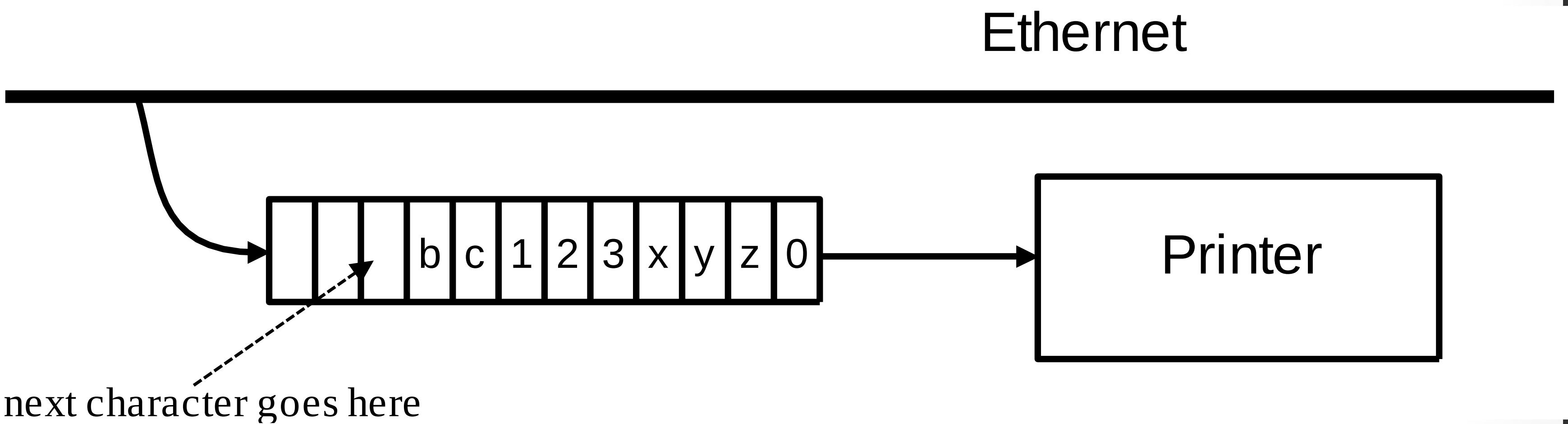
1. Understanding and applying the Queue ADT.
2. Implementing a Queue using an array and links.
3. Friendship and Inheritance in C++.
4. Implementing a Queue via list Inheritance.
5. Other Queue types: circular, priority, …

Queue Example

- Real world examples
  - Cars in line at a tool booth
  - People waiting in line for a movie

- Computer world examples
  - Jobs waiting to be executed
  - Jobs waiting to be printed
  - Input buffers
    - Characters waiting to be processed

Printer Input Buffer Example

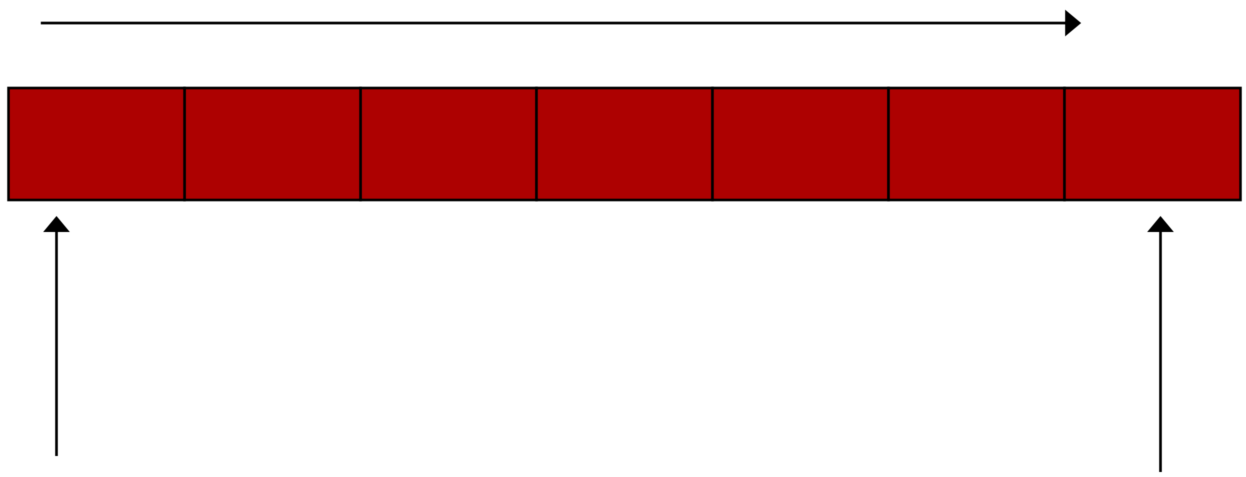


Common Queue Properties

- Queues have the property that, the earlier an item enters a queue, the earlier it will leave the queue:
  - First in, first out (FIFO)

Example

Items enter at rear and leave at front



Rear item is most recent
addition to queue

Front item has waited
Longer than all other
queue entries

Queues and lists

- A queue is a restricted form of a list.
- Additions to the queue must occur a the rear.
- Deletions from the queue must occur at the front

Queue ADT

Characteristics

A Queue Q stores items of some type (queueElementType), with First-In, First-Out (FIFO) access.

Operations

queueElementType A.dequeue()

Precondition: !isEmpty()

Postcondition: $Q_{post} = Q_{pre}$ with front removed

Returns: The least-recently enqueued item (the front).

Queue ADT operations (continued)

void Q.enqueue(queueElementType x)

Precondition: None

Postcondition: $Q_{post} = Q_{pre}$ with x added to the rear.

queueElementType Q.front()

Precondition: !isEmpty()

Postcondition: None

Returns: The least-recently enqueued item (the front).

bool Q.isEmpty()

Precondition: None

Postcondition: None

Returns: true if and only if Q is empty, i.e., contains no data items.

Code Example

```
int main()
{
  char c;
  Queue < char > q;
  Stack < char > s;
```

Code Example (continued)

```
// read characters until '.' found, adding each to Q and S.
  while (1) {
    cin  >>  c;
    if (c == '.') break;     // when '.' entered, leave the loop
    q.enqueue(c);
    s.push(c);
  }
  while (!q.isEmpty()) {
    cout << "Q: " << q.dequeue() << '\t';
    cout << "S: " << s.pop() << '\n';
  }
  return 0;
}
```

Sample Program Output

- Q: a   S: c
- Q: b   S: b
- Q: c   S: a

- Queues preserve order while stacks reverse it.

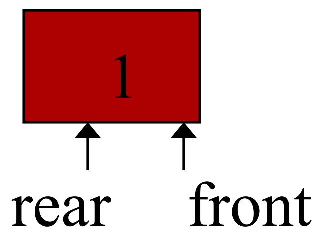
Exercise 9-1

What is the output resulting from the following sequence, where q is an initially empty queue of int:

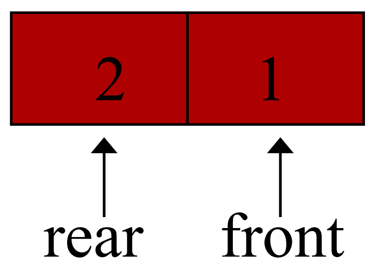
```
q.enqueue(1);
q.enqueue(2);   // different than text
q.enqueue(3);
cout << q.front();
cout << q.dequeue();
cout << q.front();
if (q.isEmpty())
        cout << “empty\n”;
else
        cout << “not empty\n”;
```

Program Analysis

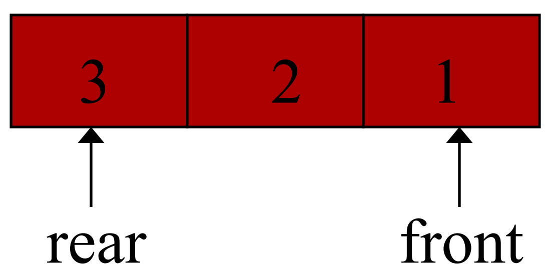
q.enqueue(1)



q.enqueue(2)



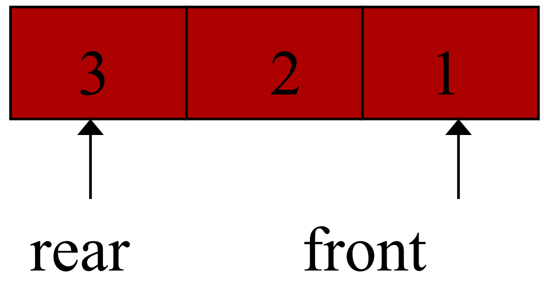
q.enqueue(3)



Program output

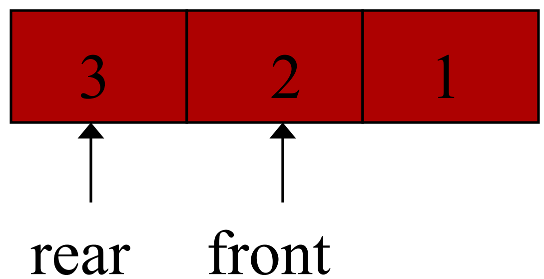
cout << q.front();

1



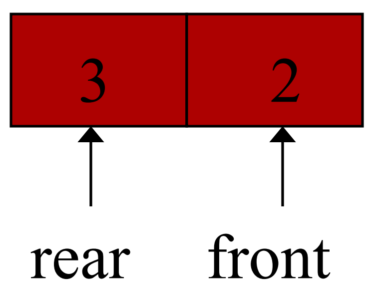
cout << q.dequeue();

1



cout << q.front();

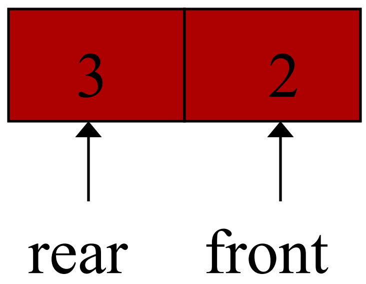
2



Program Continued

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

not empty



Array Implementation

- Must keep track of front and rear
  - more complicated than a stack

Rear and front

- With data in a queue, implemented as an array, rear will normally be greater than front.

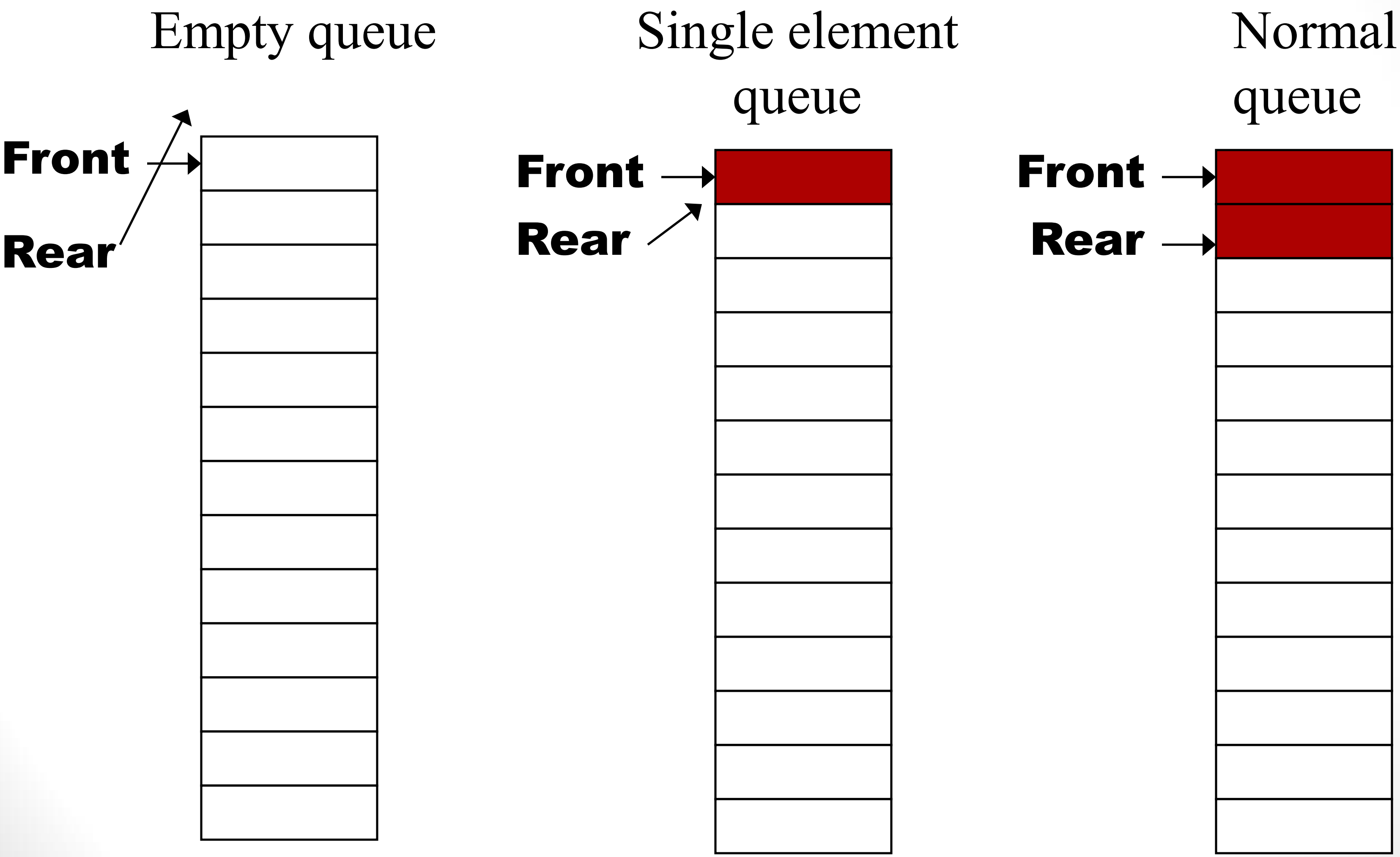
- There are 3 special situations which must be addressed

  1. Rear < front (empty queue)

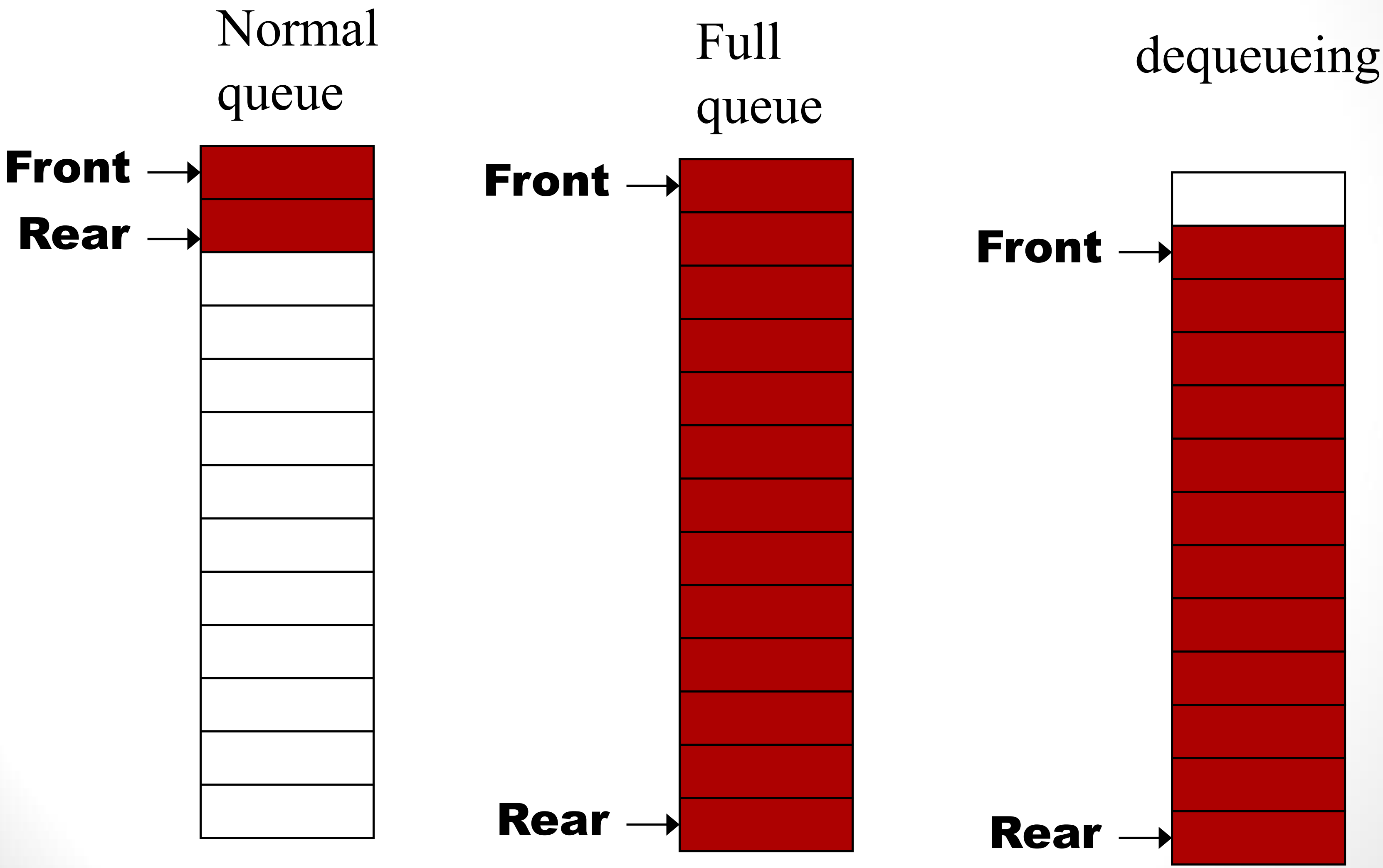
  2. Rear = front (one-entry queue)

  3. Rear = array size (full queue)

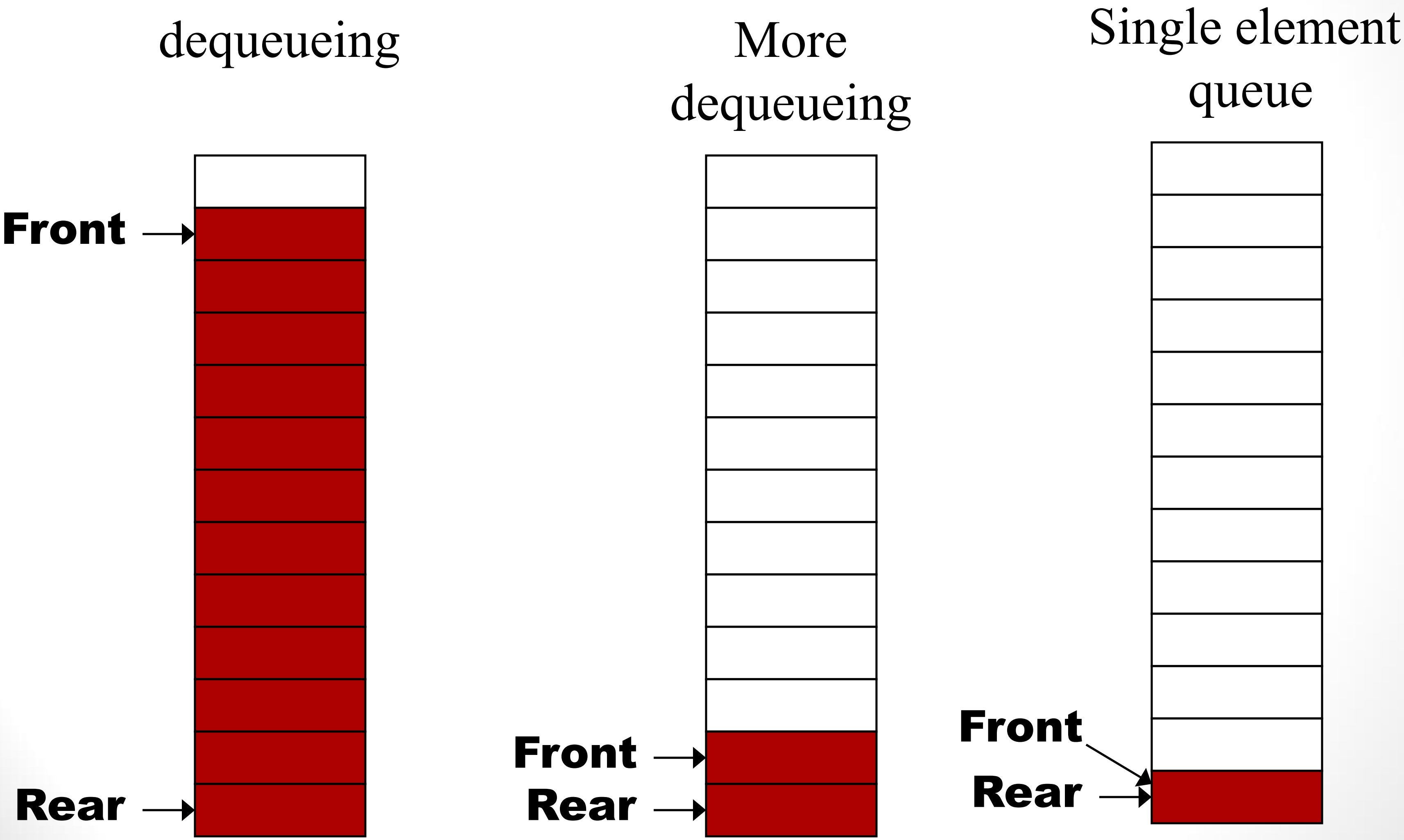
Example: enqueueing



Various Queue States

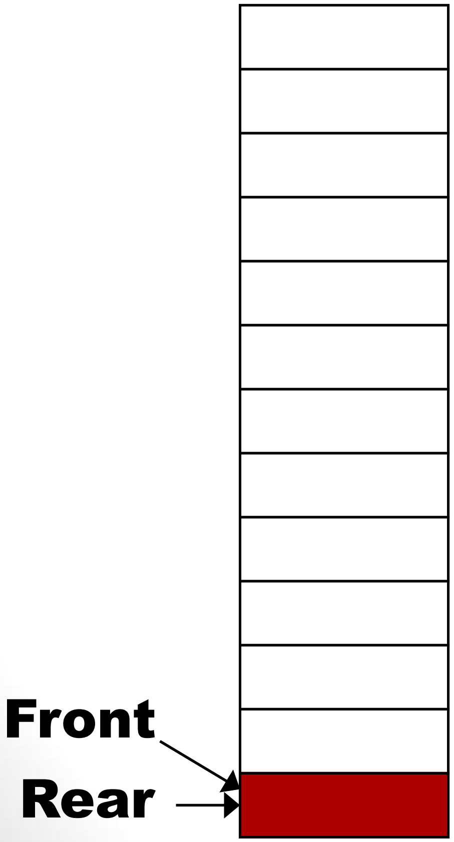


dequeueing


dequeueing
Front
Rear
More dequeueing
Front
Rear
Single element queue
Front
Rear

A problem

Single element queue



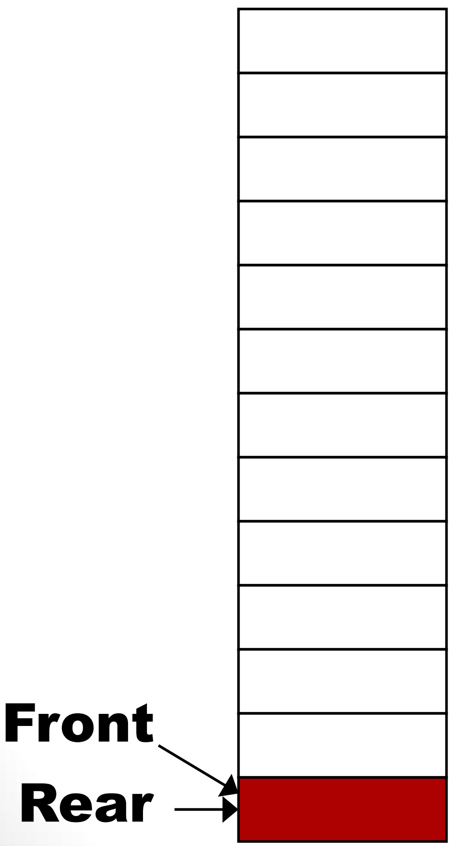
We cannot increase rear beyond the limits of the array.
Therefore, this queue must be regarded as full, even though there are plenty of available memory cells.

One way to solve this might be to
Wait until the queue empties out and
Then reset it and start again.

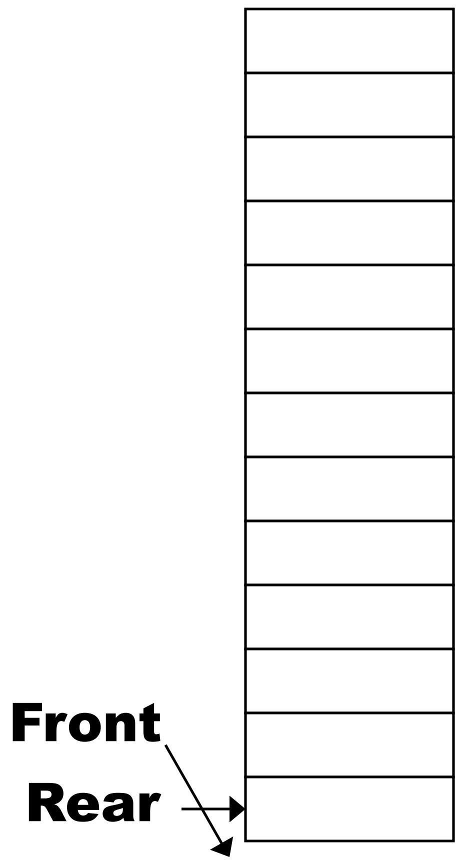
In the computer processing job example however, this would be unacceptable since many jobs could arrive while the queue is emptying out.

Another difficultly

Single element queue



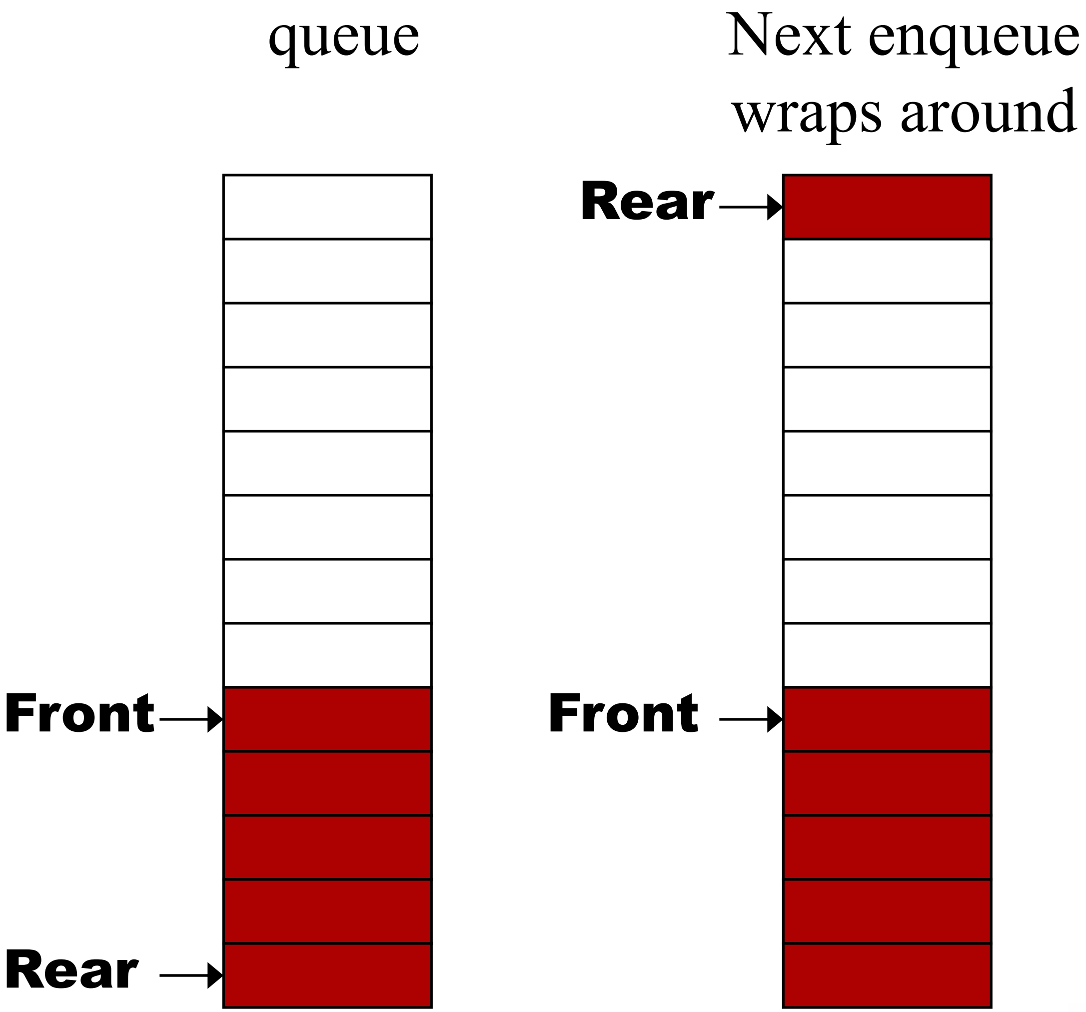
Is the queue empty or full?



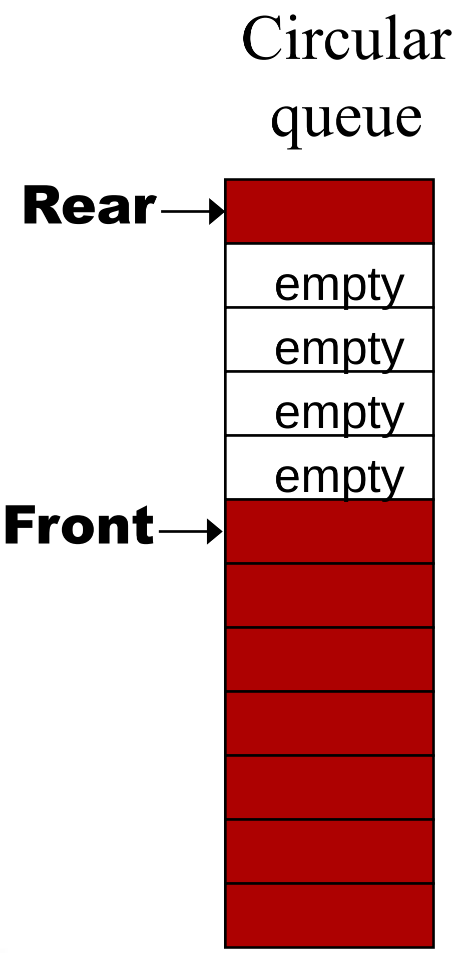
Our definition of empty is rear < front, so it is empty.

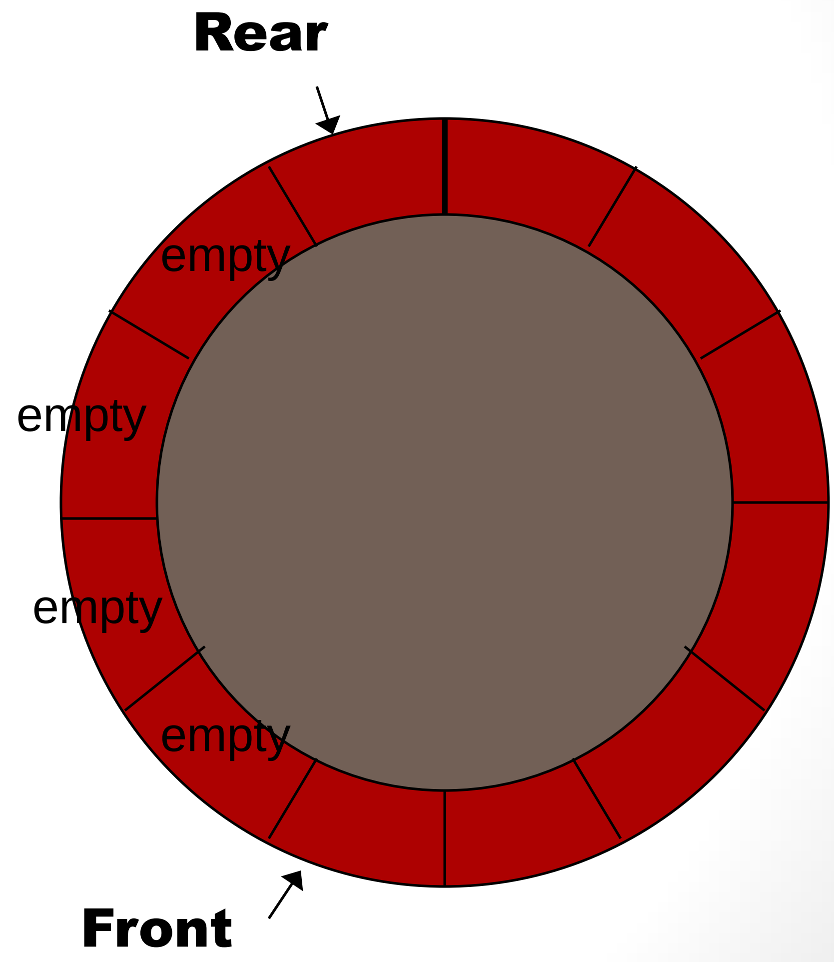
But, rear has reached its limit. It cannot go beyond the array, Therefore, the queue is full!

Solution: Wrapping Around



The Circular Queue

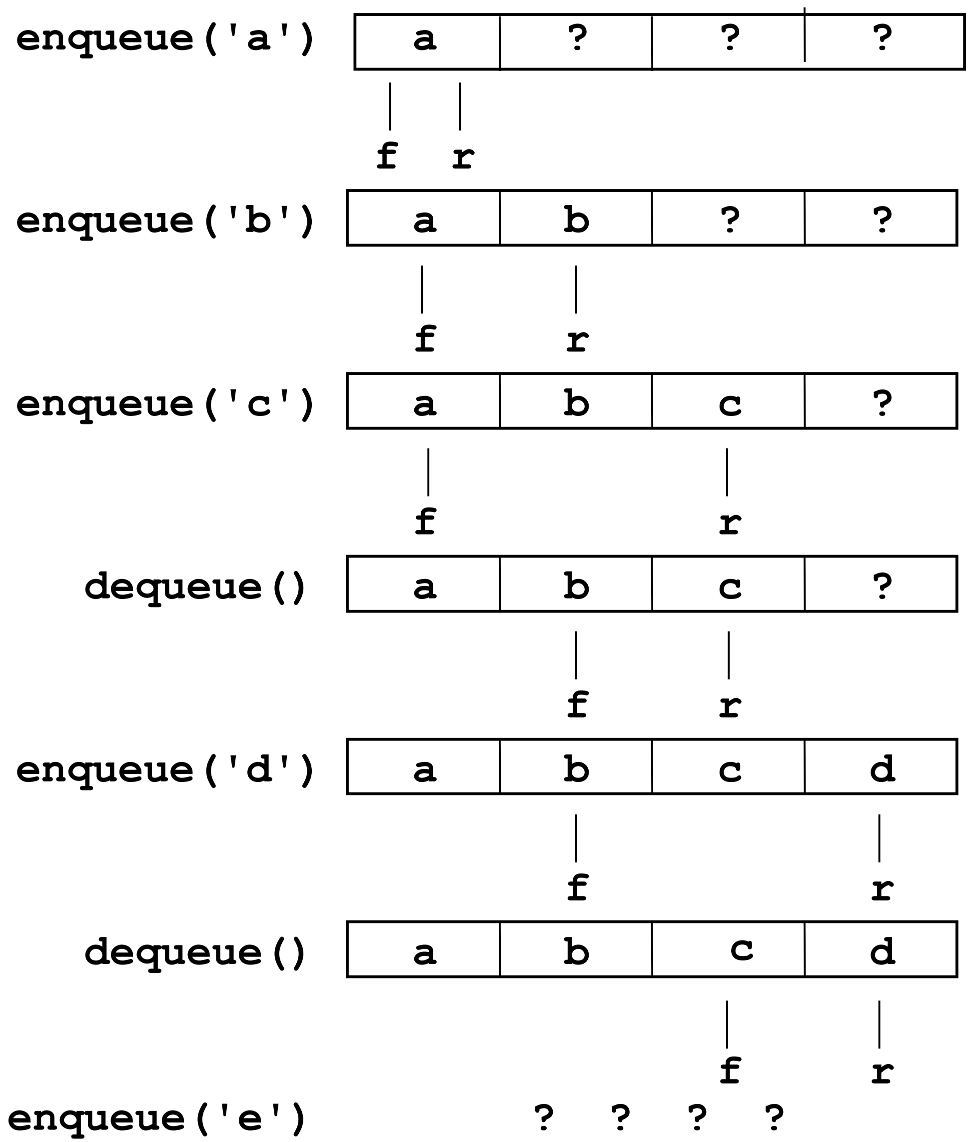




Author's Example

- Example:
  - Queue is an array of 4 elements
  - We wish to enqueue and dequeue multiple items

enqueue and dequeue operations



A Paradox

- With an array, it is easy to run out of space in the queue to enqueue items and yet still have unused memory cells that can store data!

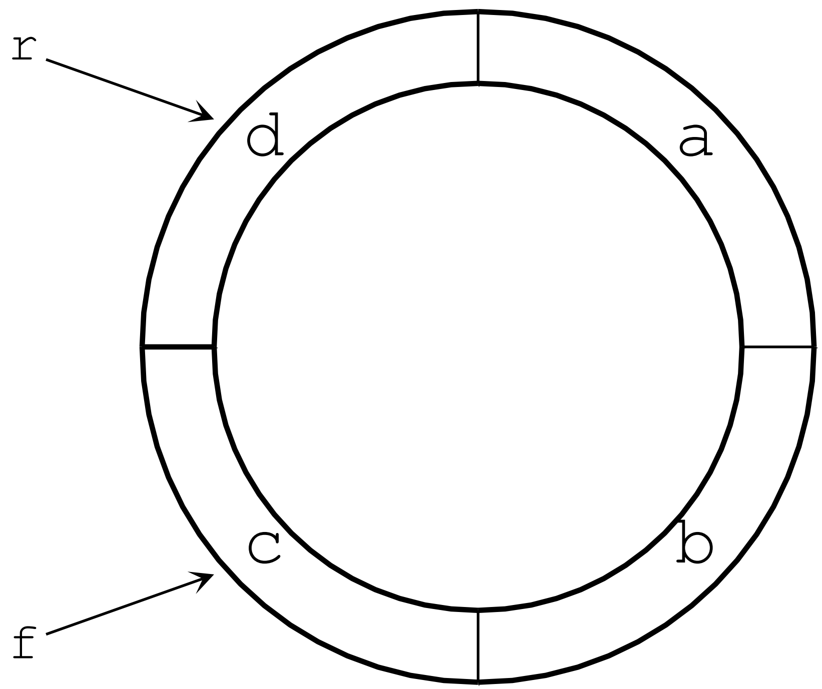
- To get around this, we have to make it possible to ‘wrap around’

- Both ‘rear’ and ‘front’ must be able to wrap around.

How to wrap around?

- If rear + 1 > maxQueue -1 then set rear to 0
- OR
- rear = (rear+1) % maxQueue

A Circular Queue



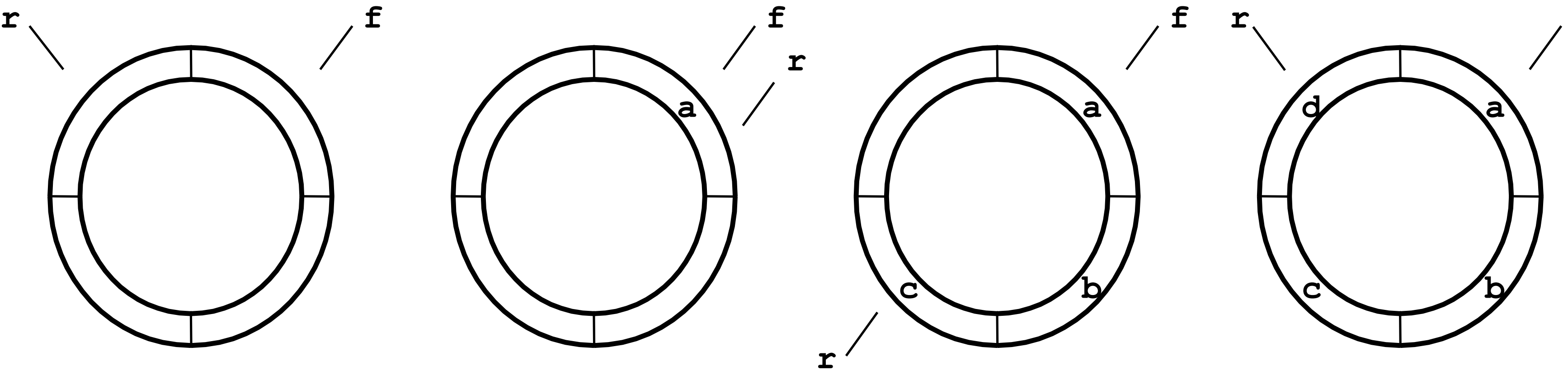
Main Issues (full and empty)

- Wrapping around allows us to avoid an erroneous ‘full’
- But, it means that we cannot use ‘rear < front’ as a test of whether the queue is empty because rear will become < front as soon as it wraps around

Rear and front

- If there is one element in the queue then rear should be the same as front, therefore, rear must start < front.
- You could check for an empty queue with something like: nextPos(rear) == front
- But, when the queue is full it is also true that nextPos(rear) == front!

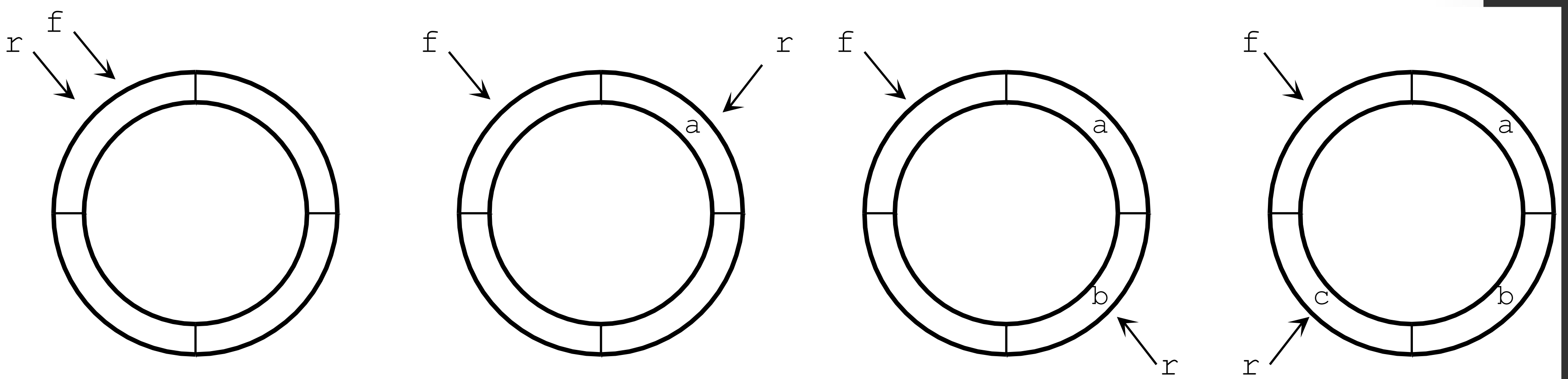
Queue implementation in which full and empty queues cannot be distinguished



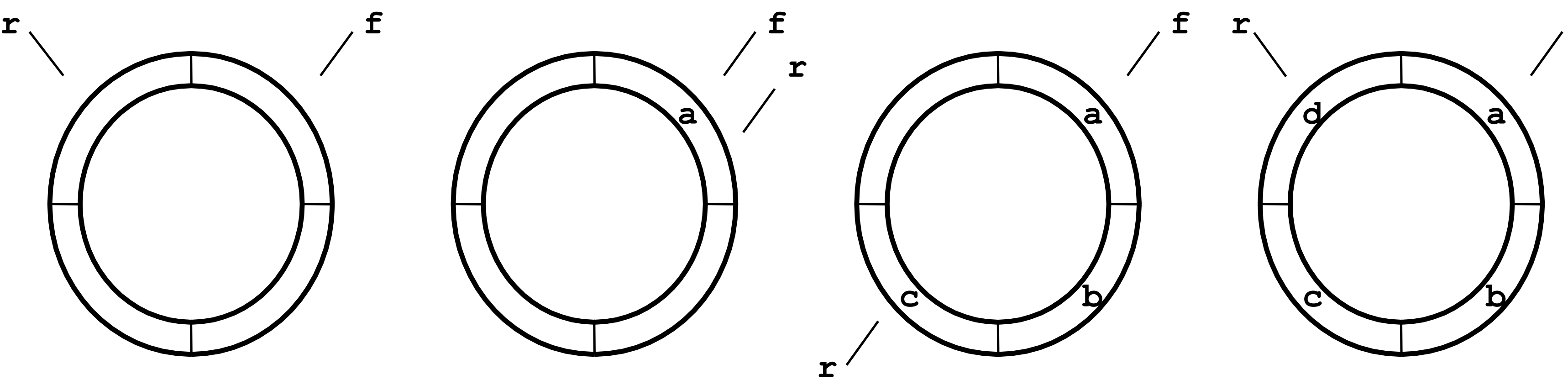
Solution

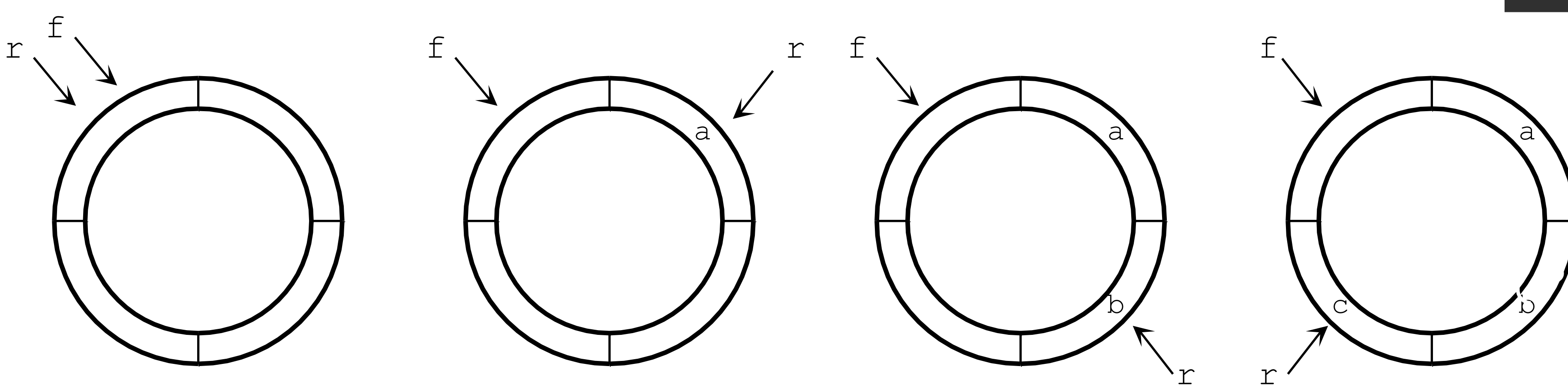
- To solve this dilemma we let the definition of empty be rear==front
- Where front points to an empty cell
- Then the test for empty is rear==front
- And the test for full is nextPos(rear) == front

Corrected queue implementation demonstrated

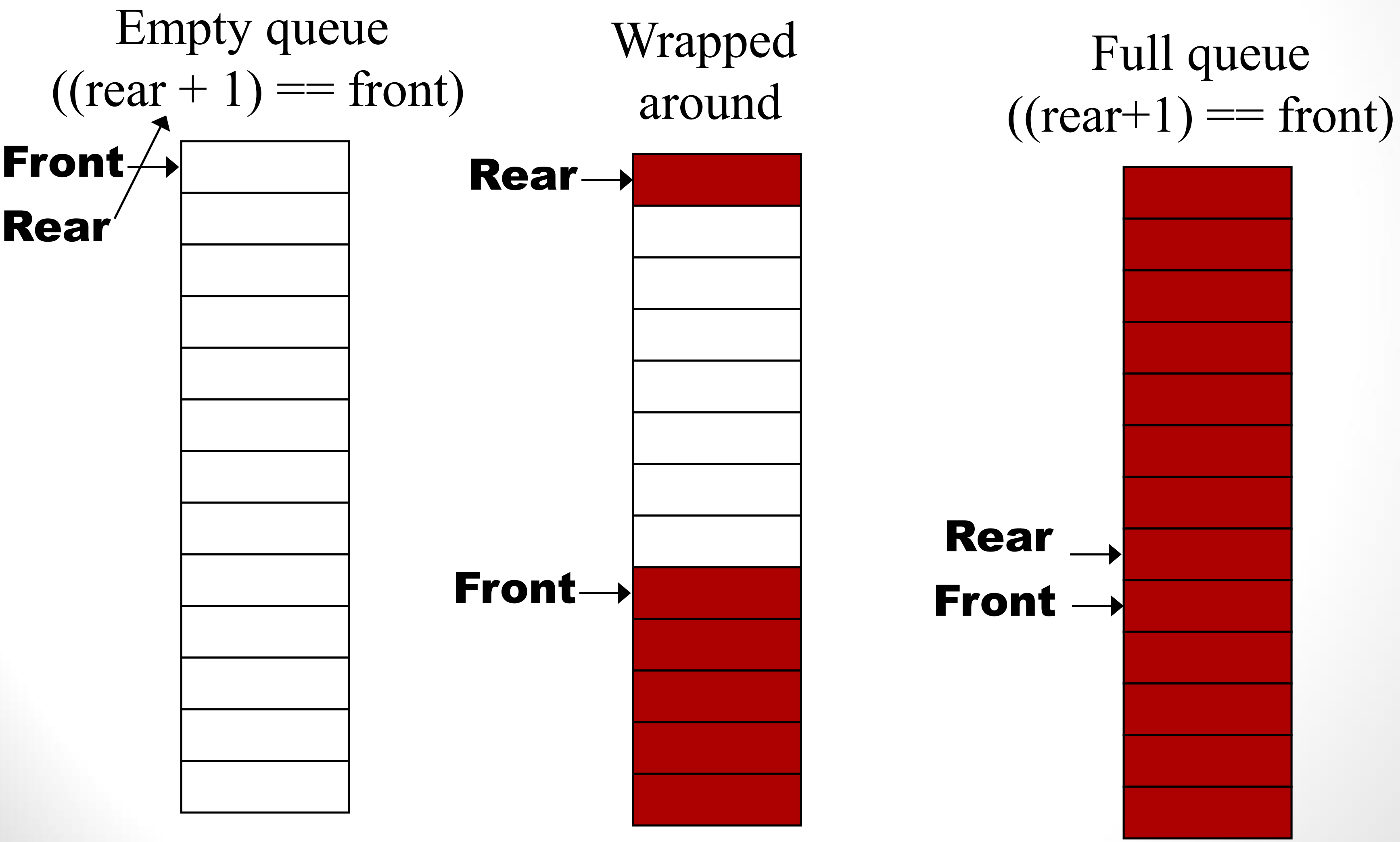


Alternate queue implementations (poor vs. good design)

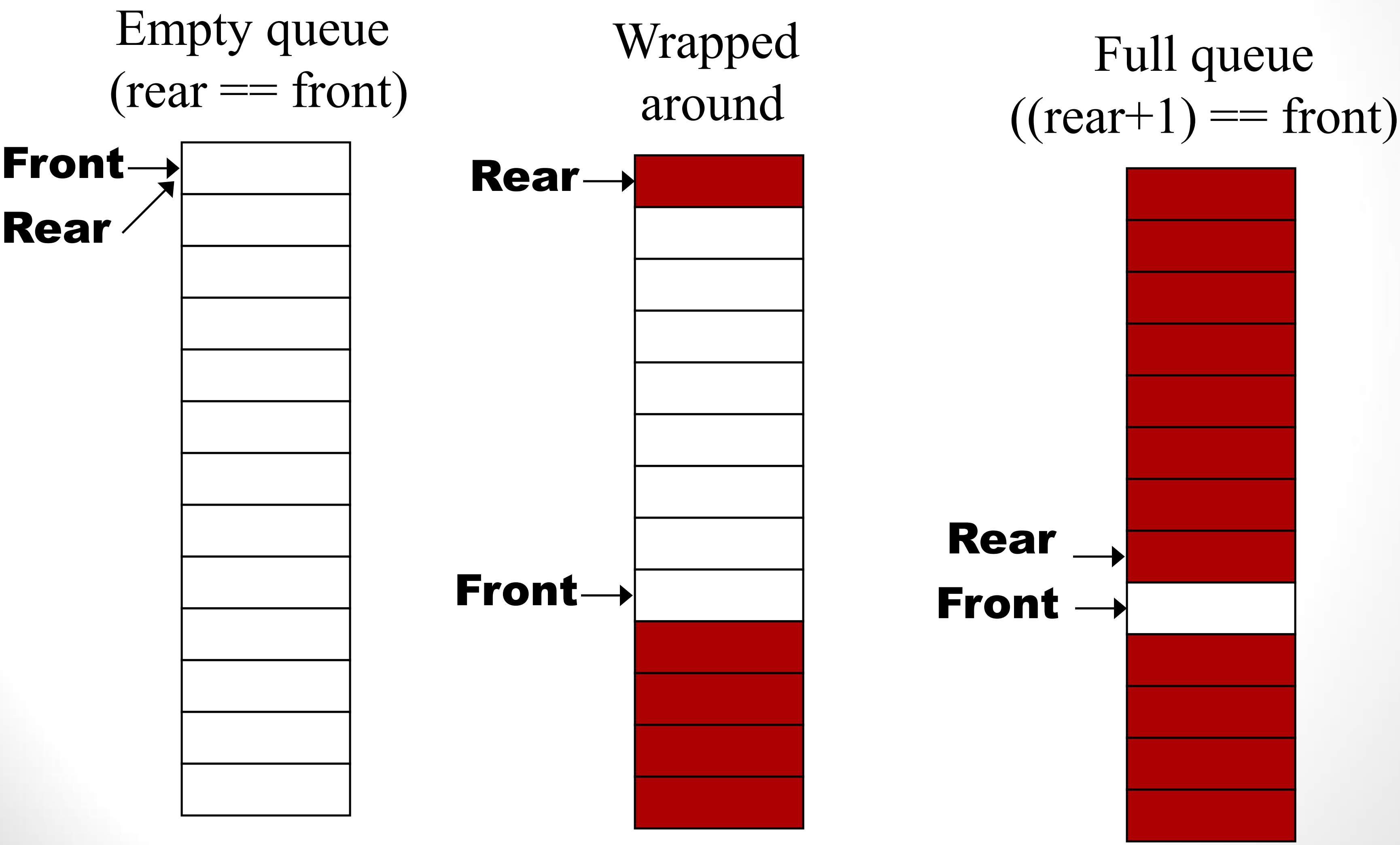




Explanation using Conventional Arrays



Solution



Empty queue
(rear == front)
Front
Rear
Wrapped
around
Rear
Front
Full queue
((rear+1) == front)
Rear
Front

Queue template

```
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];
};
```

NextPos( ), does the wrap around

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

Constructor

```
template < class queueElementType >
Queue < queueElementType >::Queue()
{ // start both front and rear at 0
   f = 0;
   r = 0;
}
```

enqueue( )

```
template < class queueElementType >
void
Queue < queueElementType > :: enqueue(queueElementType e)
{  // add e to the rear of the queue,
   // advancing r to next position
   assert(nextPos(r) != f);
   r = nextPos(r);
   elements[r] = e;
}
```

dequeue( )

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

front( )

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

isEmpty( )

```
template < class queueElementType >
bool
Queue < queueElementType >::isEmpty()
{
  // return true if the queue is empty, that is,
  // if front is the same as rear
  return bool(f == r);
}
```

Dynamic Queues

- The advantages of linked list implementation are
  - The size is limited only by the pool of available nodes (the heap)
  - There is no need to wrap around anything.

- Advantage of using the heap
  - No need for program to check for a full queue (this is handled by the 'new' function.

Header for Queue as Dynamic 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;
  nodePtr r;
};
```

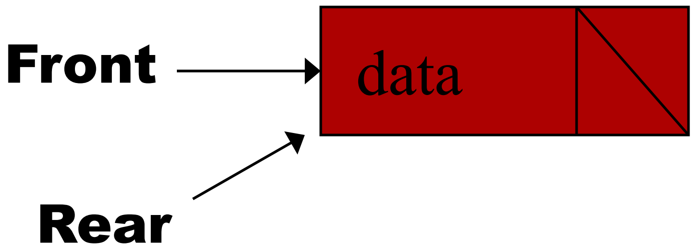
Implementation file, constructor

```
template < class queueElementType >
Queue < queueElementType >::Queue()
{
  // set both front and rear to null pointers
  f = NULL;
  r = NULL;
}
```

enqueue( )

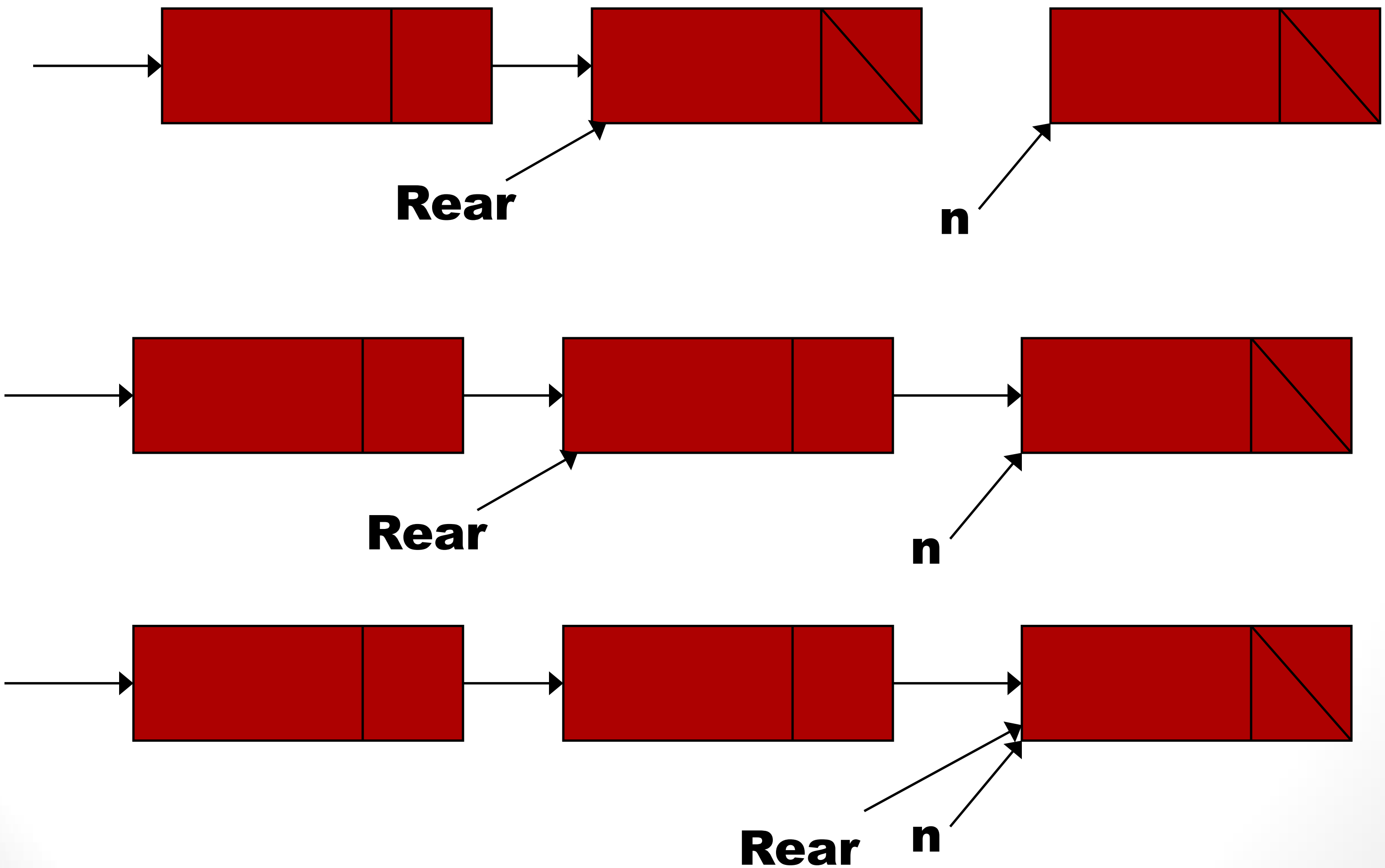
```
template < class queueElementType >
void Queue < queueElementType > ::enqueue(queueElementType e)
{// create a new node, insert it at the rear of the queue
  nodePtr n=new Node;
  assert(n);
  n->next = NULL;
  n->data = e;
  if (f != NULL) { // existing queue is not empty
    r->next = n; // add 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;
  }
}
```

A Single-Element Queue



A single element queue
Front == rear

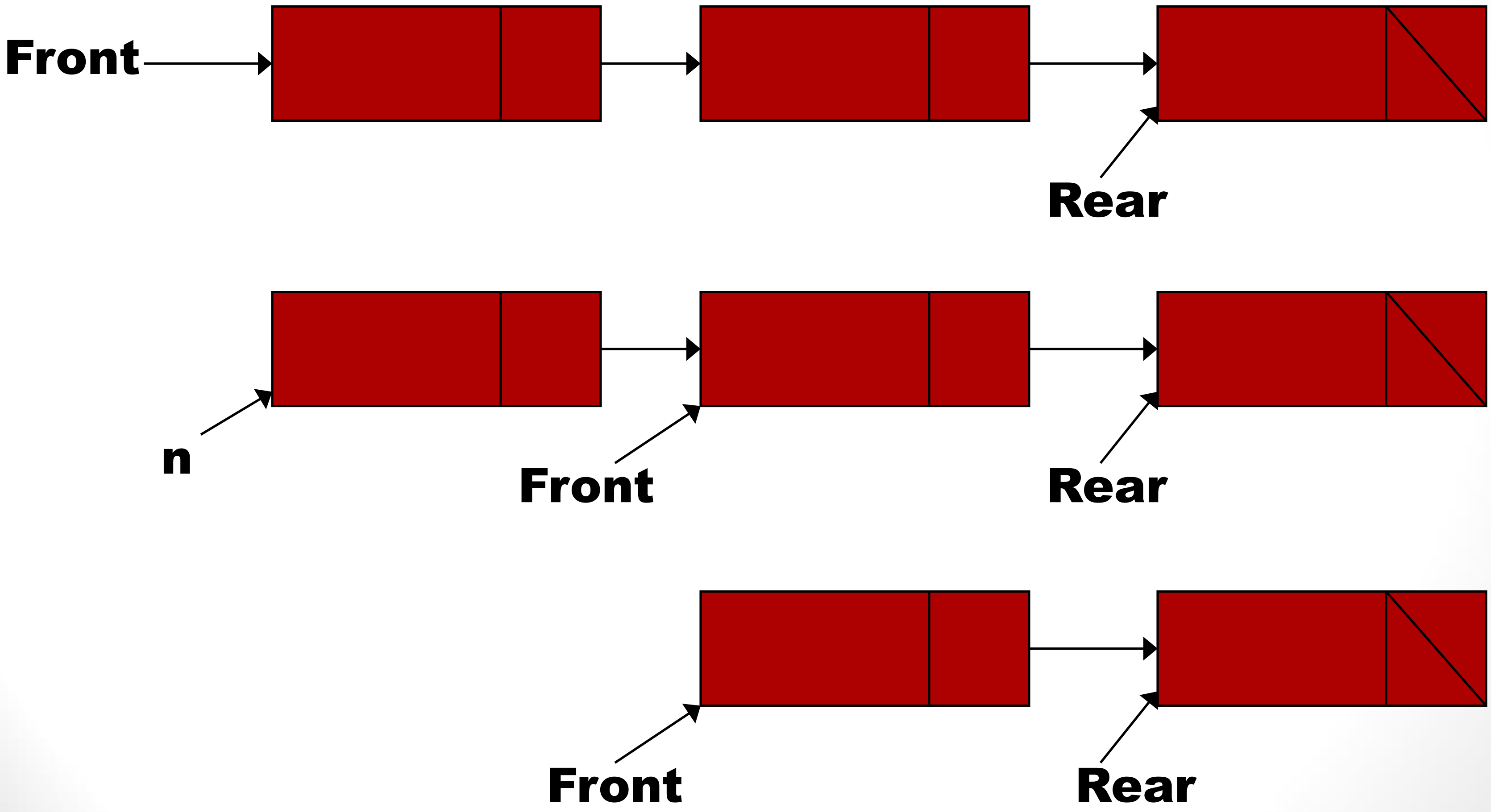
enqueueing



dequeue( )

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

dequeueing



front ( )

```
template < class queueElementType >
queueElementType
Queue < queueElementType >::front()
{
  assert(f);
  return f->data;
}
```

isEmpty( )

```
template < class queueElementType >
bool
Queue < queueElementType >::isEmpty()
{
  // true if the queue is empty -- when f is a null pointer
  return bool(f == NULL);
}
```

Friendship and Inheritance

- Friends
  - Friends are functions or classes declared with the friend keyword.
  - Friends of class A have access to the protected and private members of class A.
  - Friendships are not transitive: The friend of a friend is not considered to be a friend unless explicitly specified.

Friendship and Inheritance

- Inheritance
  - Inheritance is a mechanism of reusing and extending existing classes without modifying them.
  - Inheritance is almost like embedding an object into a class.
  - Suppose that you declare an object x of class A in the class definition of B. As a result, class B will have access to all the public data members and member functions of class A.
  - However, in class B, you have to access the data members and member functions of class A through object x.

Inheritance

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

class A {
        int  data ;
public :
        void f ( int arg ) { data = arg ; }
        int g ( ) { return data ; }
} ;

class B {
public :
        A x ;
} ;

int main ( ) {
B obj ;
obj . x . f (20) ;
cout << obj . x . g ( ) << endl ;
// cout << obj . g ( ) << endl ;
}
```

Inheritance

- Inheritance mechanism lets you use a statement like obj.g() in the above example. In order for that statement to be legal, g() must be a member function of class B

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

class A {
        int  data ;
public :
        void f ( int arg ) { data = arg ; }
        int g ( ) { return data ; }
} ;

class B : public A { } ;

int main ( ) {
        B obj ;
        obj . f (20) ;
        cout << obj . g ( ) << endl ;
}
```

Inheritance

- Syntax:

  class *DerivedClassName* : access-level *BaseClassName*

  where

  - access-level specifies the type of derivation
    - private by default, or
    - public

- Any class can serve as a base class
  - Thus a derived class can also be a base class

Notes on Inheritance

- Class A is a ***<u>base class</u>*** of class B. The names and definitions of the members of class A are included in the definition of class B.
- Class B inherits the members of class A.
- Class B is derived from class A.
- Class B contains a sub-object of type A.
- You can also add new data members and member functions to the derived class.
- You can modify the implementation of existing member functions or data by overriding base class member functions or data in the newly derived class.

Notes on Inheritance

- A derived class inherits every member of a base class <u>except</u>
  - Its constructor and its destructor.
    - However, the default constructor (i.e., its constructor with no parameters) and destructor of the base class are always called when a new object of a derived class is created or destroyed.
    - The base default constructor can be overridden.
  - Its operator=() members
  - Its friends

Queue Implementation via List Inheritance

```
#ifndef QUEUE_H
    #define QUEUE_H

    #include "List. h" // List class definition

    template < class QUEUETYPE >
    class Queue : private List< QUEUETYPE >
    {
    public :
        // enqueue calls list member function insertAtBack
        void enqueue ( const QUEUETYPE &data )
        {
            insertAtBack ( data ) ;
        } // end function enqueue
```

Queue Implementation via List Inheritance

```
        // dequeue calls List member function removeFromFront
        bool dequeue ( QUEUETYPE &data )
        {
            return removeFromFront ( data ) ;
        } // end function dequeue

        // isQueueEmpty calls List member function isEmpty
        bool isQueueEmpty ( ) const
        {
            return isEmpty ( ) ;
        } // end function isQueueEmpty

        // printQueue call s List member function print
        void printQueue ( )
        {
            print ( ) ;
        } // end function printQueue
} ; // end class Queue
#endif
```

Priority Queues

- Priority queues are a special type of queues in which queue elements are processed in order of importance/priority
- The priority queues appears in different contexts
  - packets with different priority
  - patients at emergency section
- Implementation Approaches
  - Unsorted list
    - Adv: simple insert
    - Disadv: search before dequeue
  - Linked sorted list
    - Adv: simple dequeue (always get the first element)
    - Disadv: O(N) enqueue as we need to decide where to insert the received object