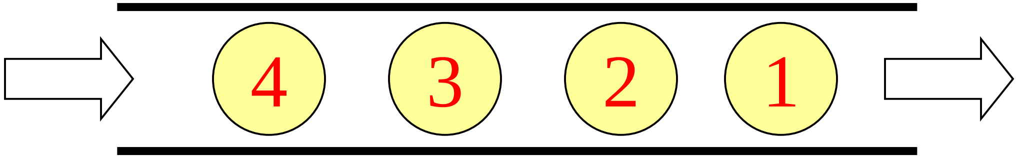
Queues

What is a queue?

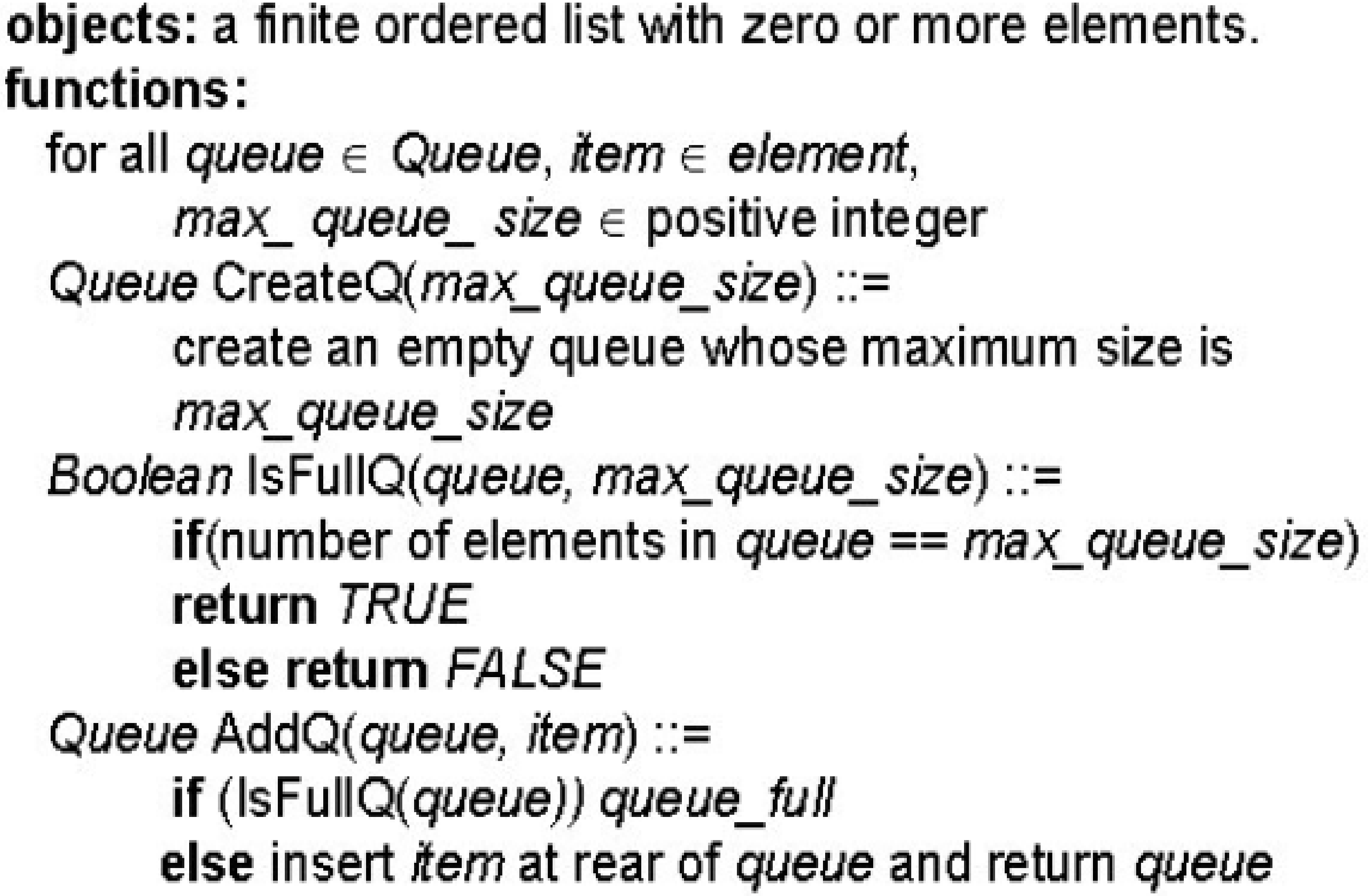
- It is an ordered group of homogeneous items of elements.
- Queues have two ends:
  - Elements are added at one end.
  - Elements are removed from the other end.
- The element added first is also removed first (**FIFO**: First In, First Out).



Queue Specification

- Definitions: (provided by the user)
  - *MAX_ITEMS*: Max number of items that might be on the queue
  - *ItemType*: Data type of the items on the queue

Queue as abstract data type



```
objects: a finite ordered list with zero or more elements.
functions:
  for all queue ∈ Queue, item ∈ element,
          max_ queue_ size ∈ positive integer
  Queue CreateQ(max_queue_size) ::=
          create an empty queue whose maximum size is
          max_queue_size
  Boolean IsFullQ(queue, max_queue_size) ::=
          if(number of elements in queue == max_queue_size)
          return TRUE
          else retum FALSE
  Queue AddQ(queue, item) ::=
          if (IsFullQ(queue)) queue_full
         else insert item at rear of queue and return queue
```

Enqueue (ItemType newItem)

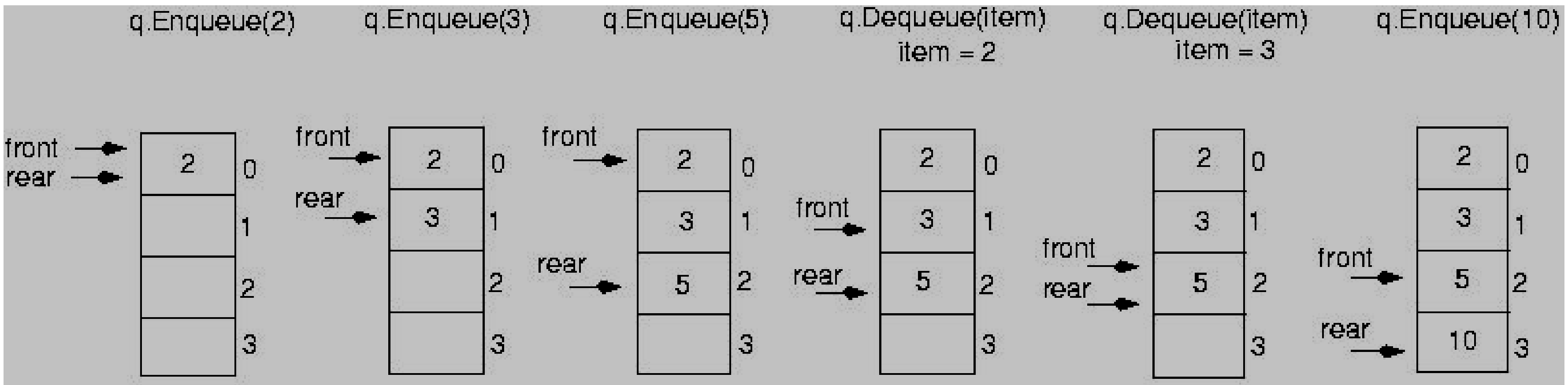
- *Function*: Adds newItem to the rear of the queue.
- *Preconditions*: Queue has been initialized and is not full.
- *Postconditions*: newItem is at rear of queue.

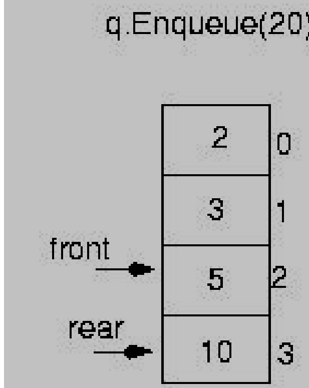
Dequeue (ItemType& item)

- *Function*: Removes front item from queue and returns it in item.
- *Preconditions*: Queue has been initialized and is not empty.
- *Postconditions*: Front element has been removed from queue and item is a copy of removed element.

Implementation issues

- Implement the queue as a *circular structure*.
- How do we know if a queue is full or empty?
- Initialization of *front* and *rear*.
- Testing for a *full* or *empty* queue.





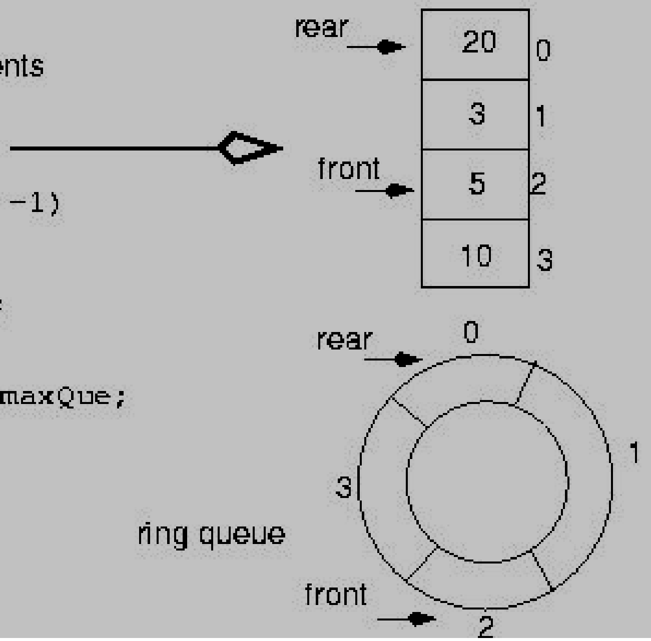
Let the queue elements

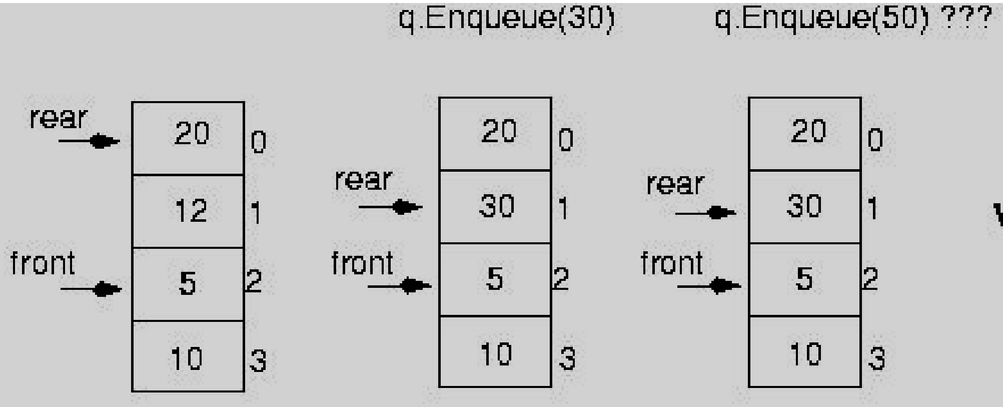
"wrap around"

```
if(rear == maxQue -1)
  rear = 0;
else
  rear = rear + 1;
```

or

```
rear = (rear + 1) % maxQue;
```

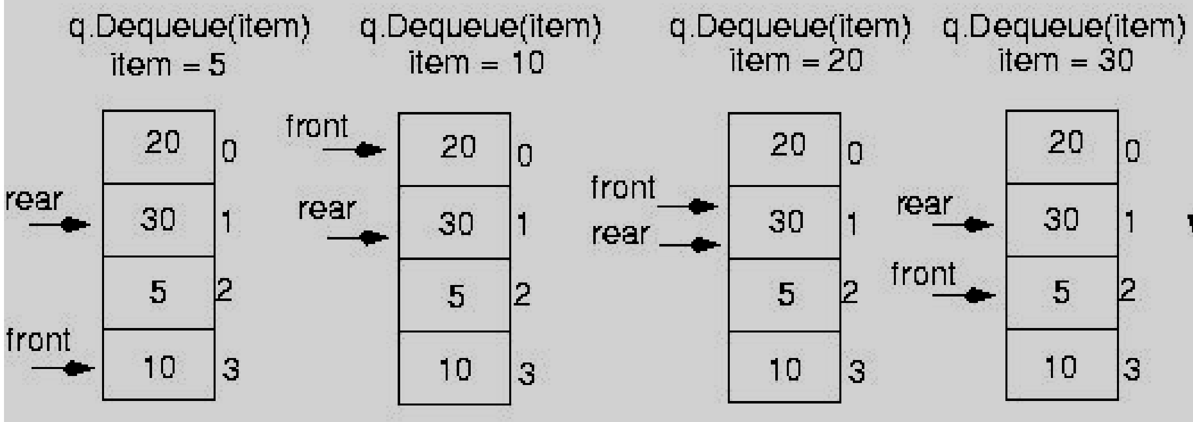




The queue is full !!

What is the condition for a full queue ?

```
rear + 1 == front
```

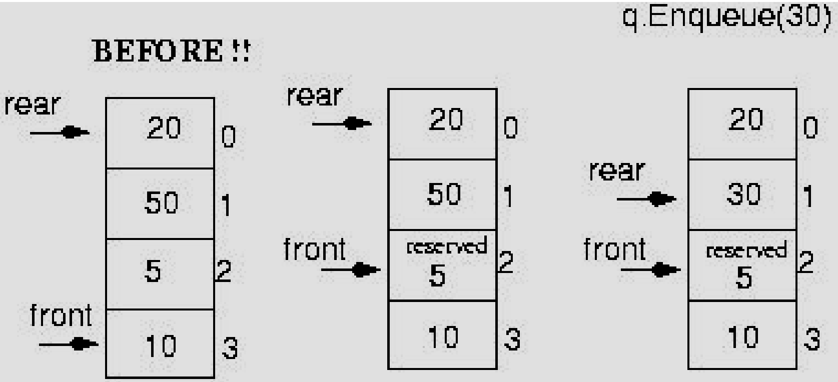


The queue is empty !!

What is the condition for an empty queue ?

```
rear + 1 == front
```

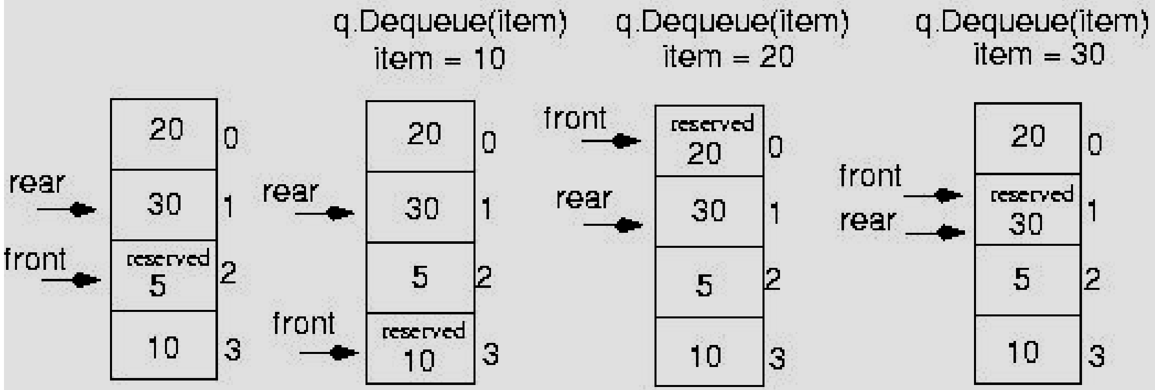
We cannot distinguish between the two cases !!!



The queue is full !!

What is the condition for a full queue ?

```
rear + 1 == front
```

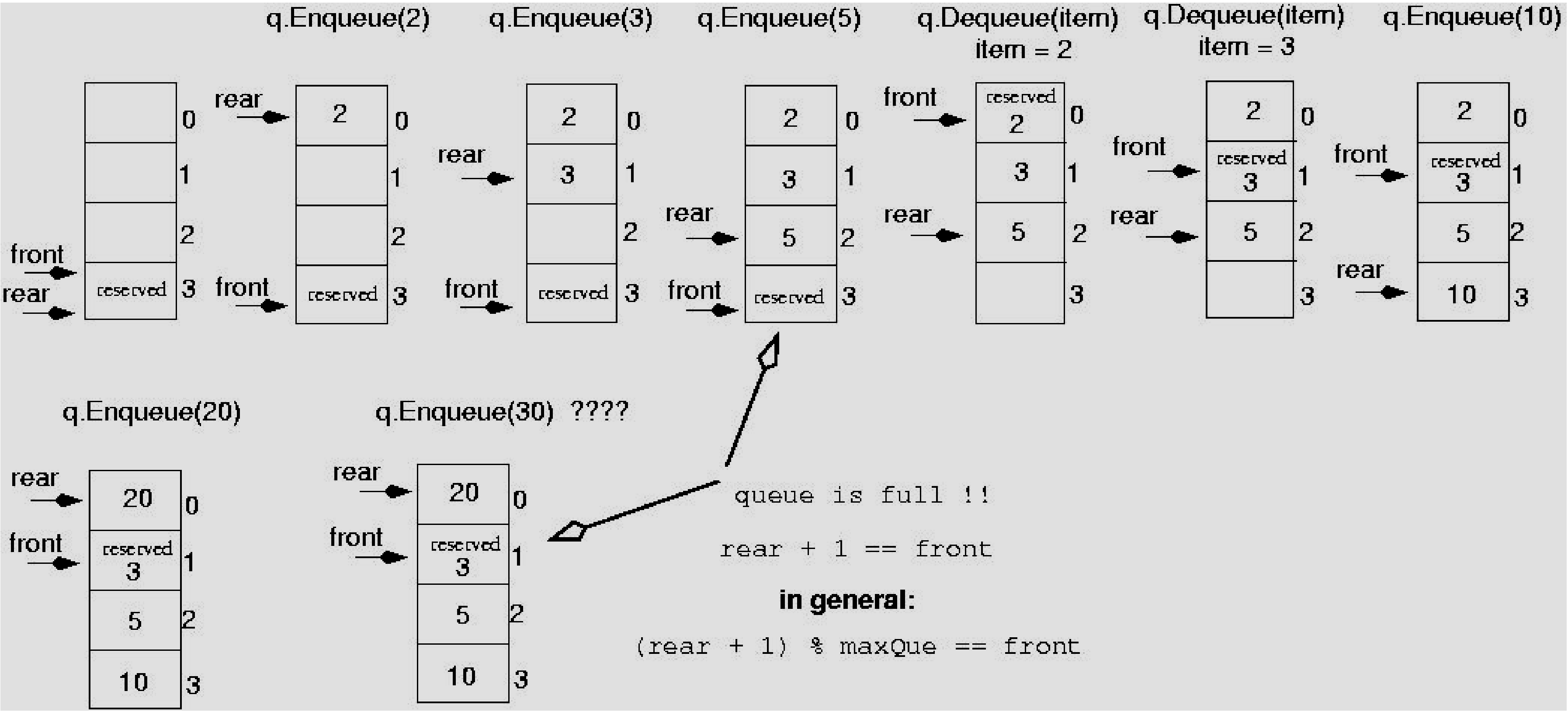


The queue is empty !!

What is the condition for an empty queue ?

```
rear == front
```

Based on this solution, one memory location is wasted !!!

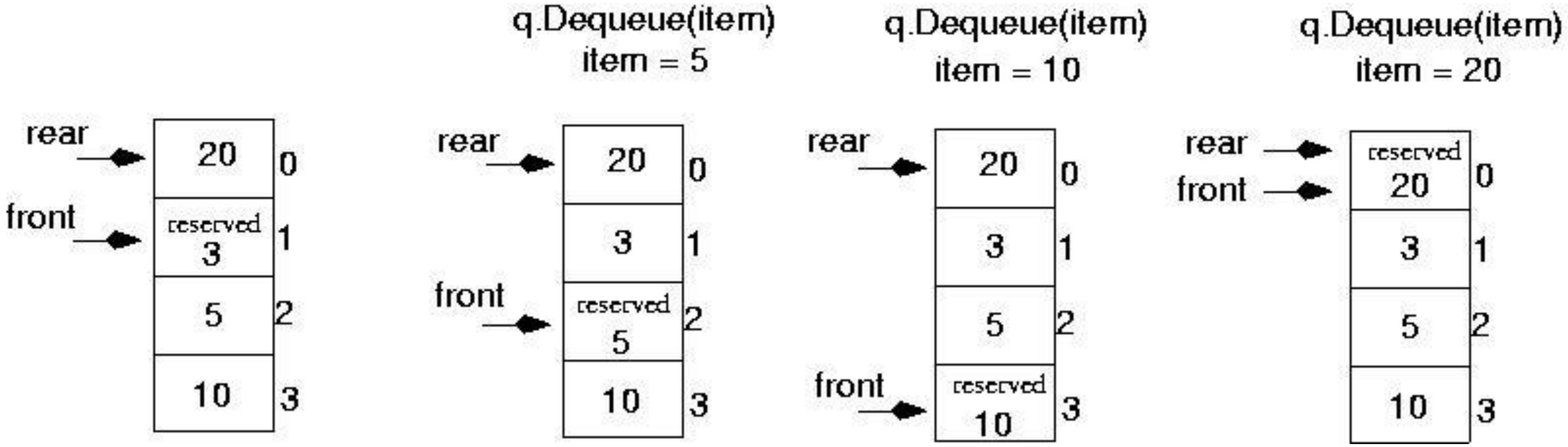


queue is full !!

rear + 1 == front

in general:

(rear + 1) % maxQue == front


rear
front
20
reserved
3
5
10
0
1
2
3
q.Dequeue(item)
item = 5
rear
front
20
3
reserved
5
10
0
1
2
3
q.Dequeue(item)
item = 10
rear
front
20
3
5
reserved
10
0
1
2
3
q.Dequeue(item)
item = 20
rear
front
reserved
20
3
5
10
0
1
2
3

Queue Implementation

```
class QueueType {
 public:
    QueueType(int);
    QueueType();
void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Enqueue();
    void Dequeue();
```

Queue Implementation (cont.)

```
void Enqueue() {
   int val;
   if (rear == n - 1)
   cout<<"Queue Overflow"<<endl;
   else {
      if (front == - 1)
      front = 0;
      cout<<"Insert the element in queue : "<<endl;
      cin>>val;
      rear++;
      queue[rear] = val;
   }
}
```

Queue Implementation (cont.)

```
void DeQueue() {
  if (front == - 1 || front > rear) {
    cout<<"Queue Underflow ";
    return ;
  }
  else {
    cout<<"Element deleted from queue is : "<< queue[front]
<<endl;
    front++;;
  }
}
```

Queue Implementation (cont.)

```
void Display() {
  if (front == - 1)
  cout<<"Queue is empty"<<endl;
  else {
    cout<<"Queue elements are : ";
    for (int i = front; i <= rear; i++)
    cout<<queue[i]<<" ";
    cout<<endl;
  }
}
```

Circular Queue

```
class Queue
{
        // Initialize front and rear
        int rear, front;

        // Circular Queue
        int size;
        int *arr;
public:
        Queue(int s)
        {
        front = rear = -1;
        size = s;
        arr = new int[s];
        }
        void enQueue(int value);
        int deQueue();
        void displayQueue();
};
```

/* Function to create Circular queue */

```
void Queue::enQueue(int value){
        if ((front == 0 && rear == size-1) ||(rear == (front-1)%(size-1)))
        {
                printf("\nQueue is Full");
                return;
        }
        else if (front == -1) /* Insert First Element */{
                front = rear = 0;
                arr[rear] = value;
        }

        else if (rear == size-1 && front != 0){
                rear = 0;
                arr[rear] = value;
        }

        else{
                rear++;
                arr[rear] = value;
        }
}
```

```
// Function to delete element from Circular Queue
int Queue::deQueue()
{
        if (front == -1){
                printf("\nQueue is Empty");
                return INT_MIN;
        }
        int data = arr[front];
        arr[front] = -1;
        if (front == rear)
        {
                front = -1;
                rear = -1;
        }
        else if (front == size-1)
                front = 0;
        else
                front++;

        return data;
}
```

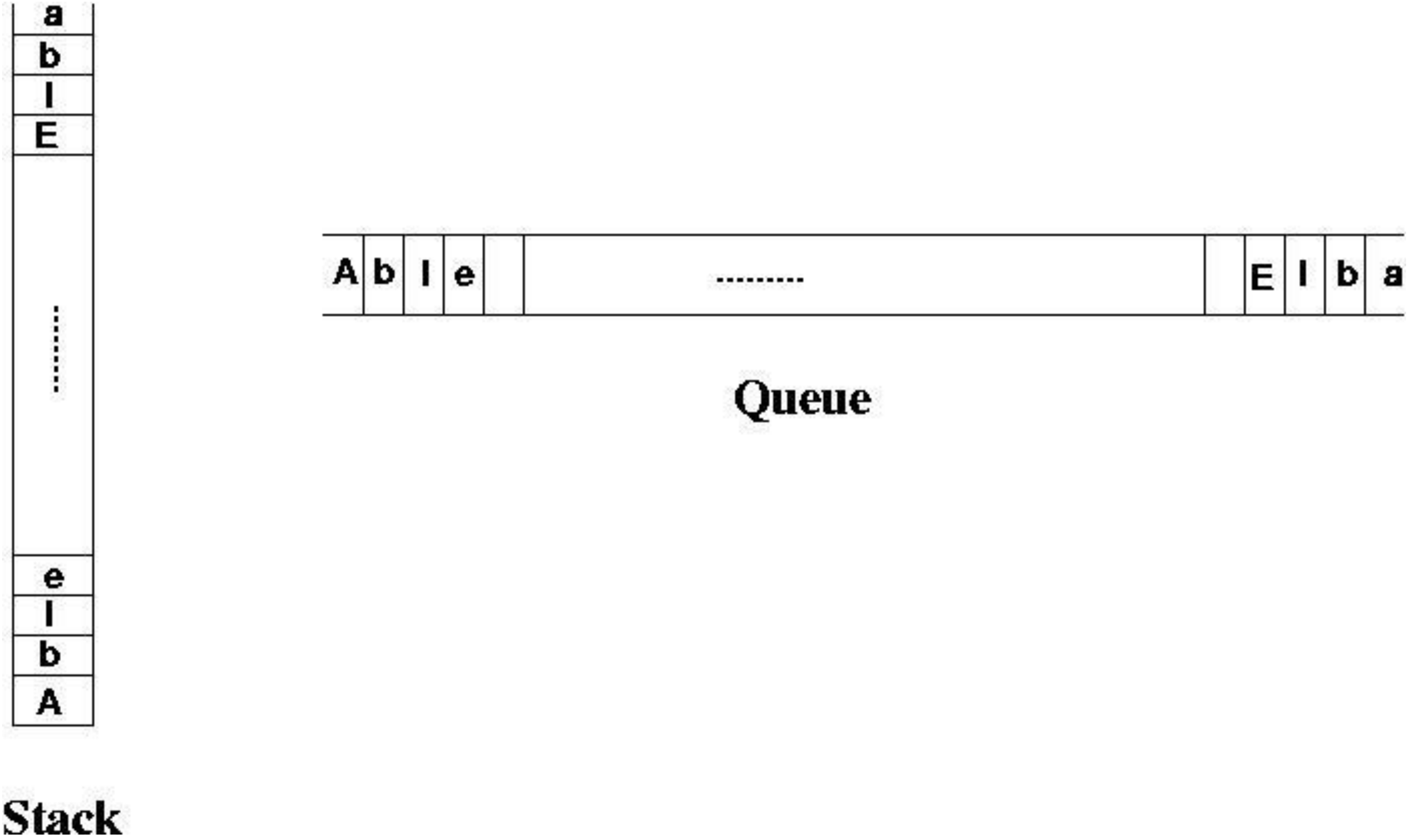
Example: recognizing palindromes

- A *palindrome* is a string that reads the same forward and backward.

Able was I ere I saw Elba

- We will read the line of text into both a stack and a queue.
- Compare the contents of the stack and the queue character-by-character   to see if they would produce the same string of characters.

Example: recognizing palindromes



Example: recognizing palindromes

```
#include <iostream.h>
#include <ctype.h>
#include "stack.h"
#include "queue.h“
int main()
{
 StackType<char> s;
 QueType<char> q;
 char ch;
 char sItem, qItem;
 int mismatches = 0;
```

Example: recognizing palindromes

```
 while( (!q.IsEmpty()) && (!s.IsEmpty()) ) {

   s.Pop(sItem);
   q.Dequeue(qItem);

   if(sItem != qItem)
     ++mismatches;
 }
 if (mismatches == 0)
   cout << "That is a palindrome" << endl;
 else
  cout << That is not a palindrome" << endl;

 return 0;
}
```

Case Study: Simulation

- Queuing System: consists of *servers* and *queues* of objects to be served.

- Simulation: a program that determines how long items must wait in line before being served.

Case Study: Simulation (cont.)

- Inputs to the simulation:

  (1) the length of the simulation

  (2) the average transaction time

  (3) the number of servers

  (4) the average time between job arrivals

Case Study: Simulation (cont.)

- Parameters the simulation must vary:

  (1) number of servers

  (2) time between arrivals of items

- Output of simulation: average wait time.