Data Structures

Week 6: Queues and its applications

Irfan Mehmood

Sejong University

Yulgokgwan 205B

irfan@sejong.ac.kr

Queues

- A stack is LIFO (Last-In First Out) structure.
- In contrast, a ***queue*** is a FIFO (First-In First-Out) structure.
- A queue is a linear structure for which items can be only inserted at one end and removed at another end.

Queue Operations

Enqueue(X) – place X at the *rear* (last/back) of the queue.

Dequeue() -- remove the *front* element and return it.

Front() -- return front element without removing it.

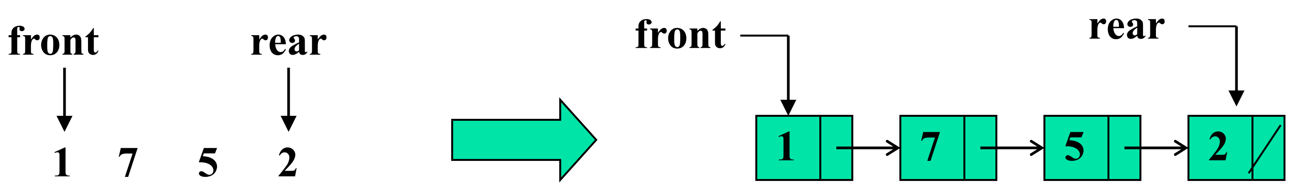
IsEmpty() -- return TRUE if queue is empty, FALSE otherwise

Implementing Queue

- Using linked List: *Recall*
- Insert works in constant time for either end of a linked list.
- Remove works in constant time only.
- Seems best that head of the linked list be the front of the queue so that all removes will be from the front.
- Inserts will be at the end of the list.

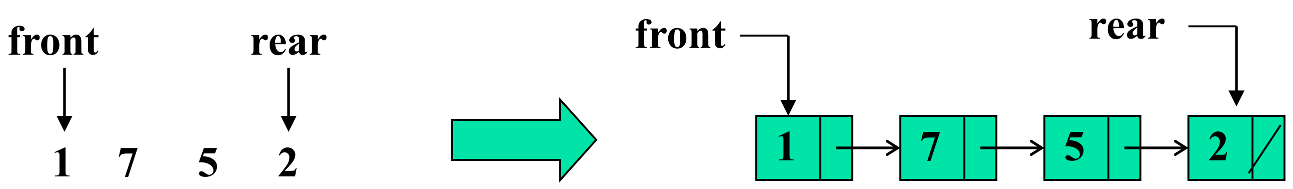
Implementing Queue

- Using linked List:

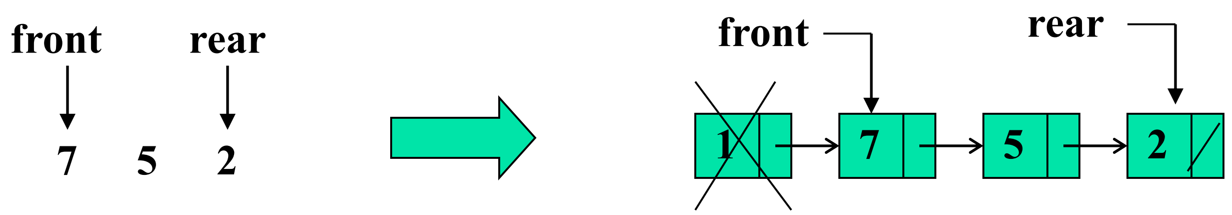


Implementing Queue

- Using linked List:



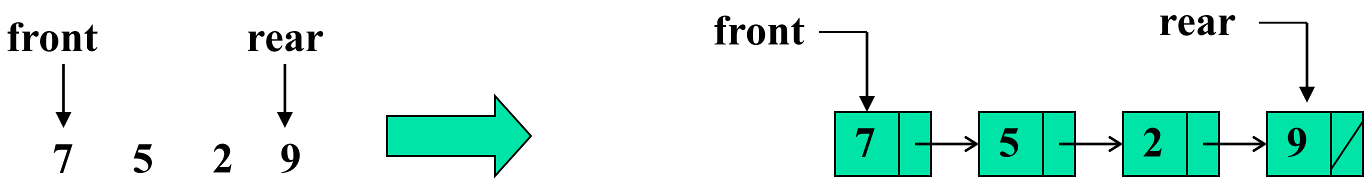
dequeue()



Implementing Queue

- Using linked List:

enqueue(9)



Implementing Queue

```
int dequeue()
{
    int x = front->get();
    Node* p = front;
    front = front->getNext();
    delete p;
    return x;
}
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
    rear->setNext(newNode);
    rear = newNode;
}
```

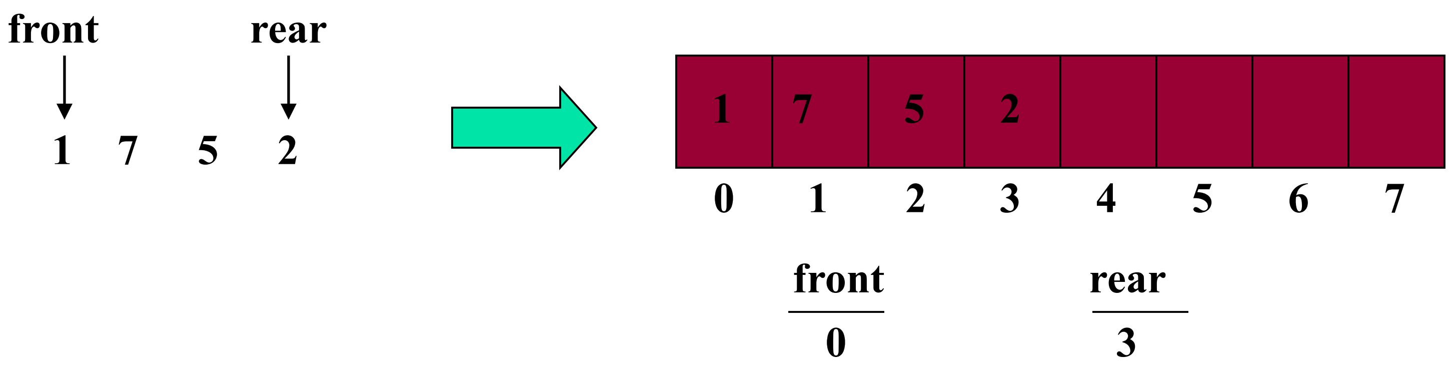
Implementing Queue

```
int front()
{
    return front->get();
}


int isEmpty()
{
     return ( front == NULL );
}
```

Queue using Array

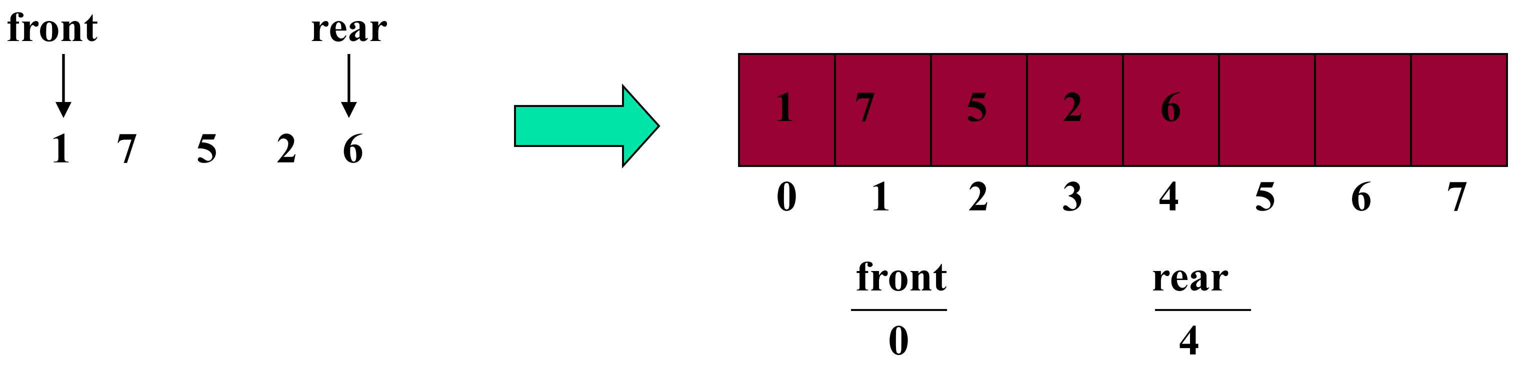
- If we use an array to hold queue elements, both insertions and removal at the front (start) of the array are expensive.
- This is because we may have to shift up to “n” elements.
- For the stack, we needed only one end; for queue we need both.
- To get around this, we will not shift upon removal of an element.

Queue using Array



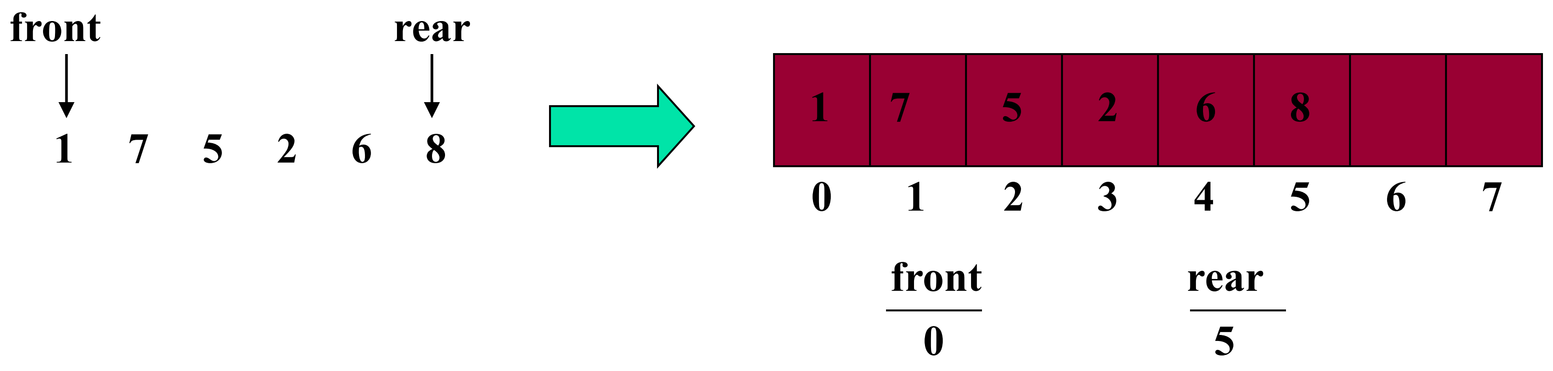
Queue using Array

enqueue(6)



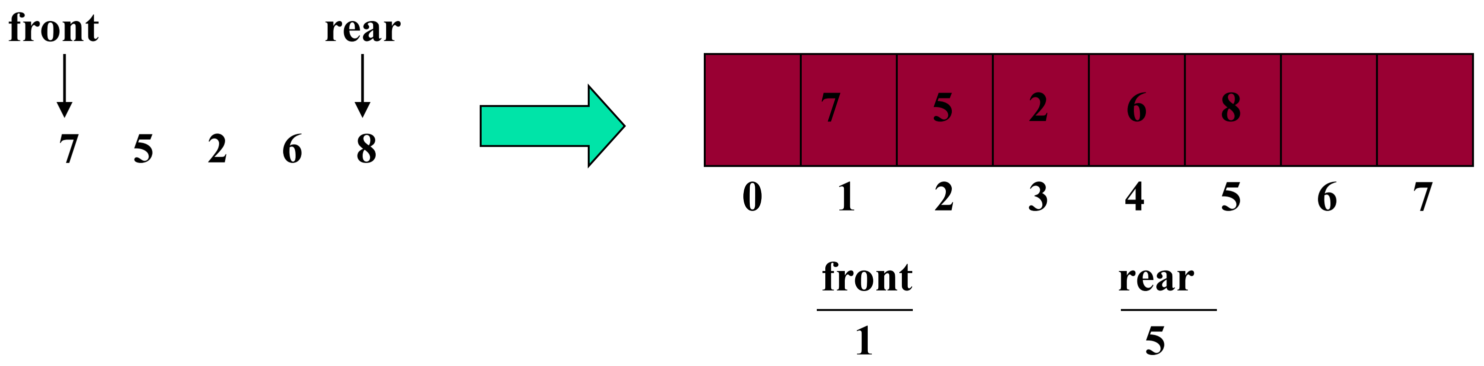
Queue using Array

enqueue(8)



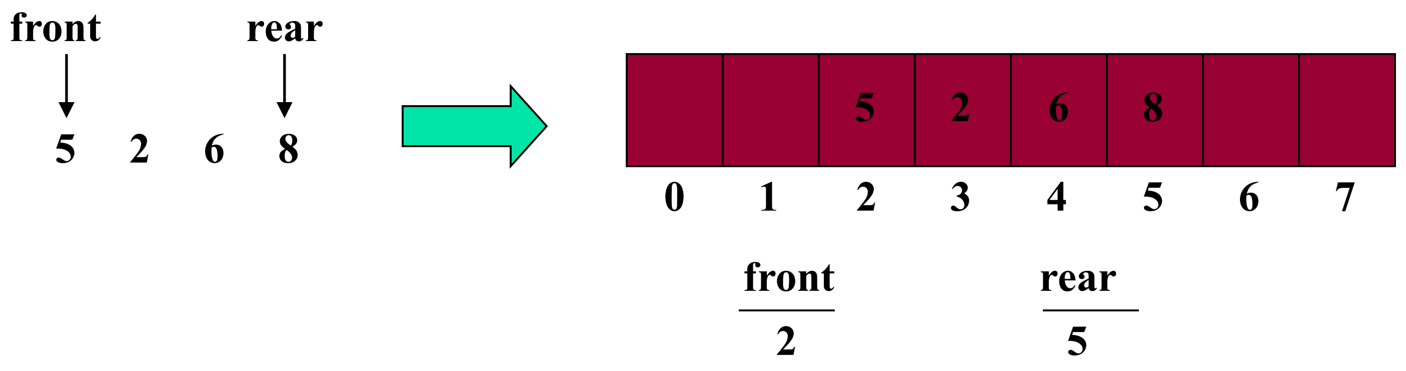
Queue using Array

dequeue()



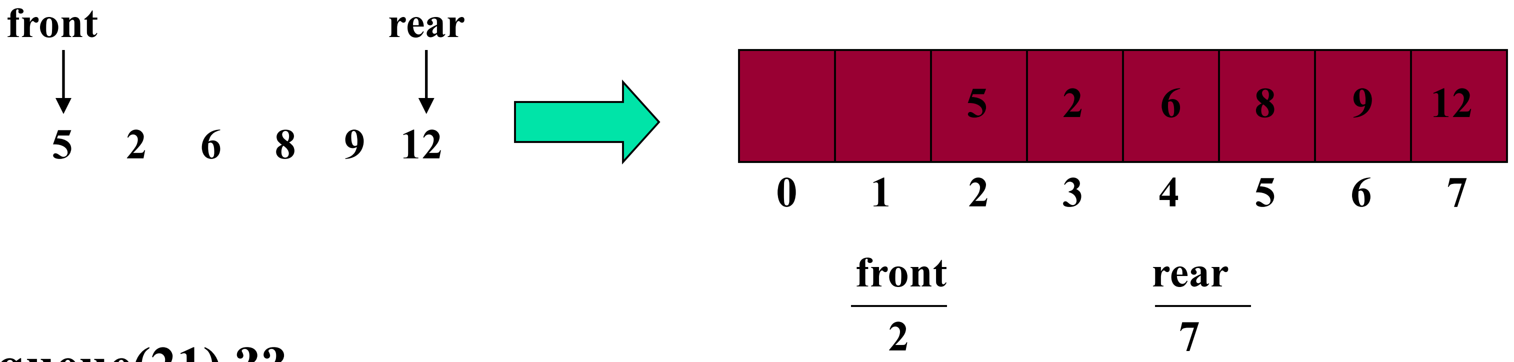
Queue using Array

dequeue()



Queue using Array

enqueue(9)
enqueue(12)



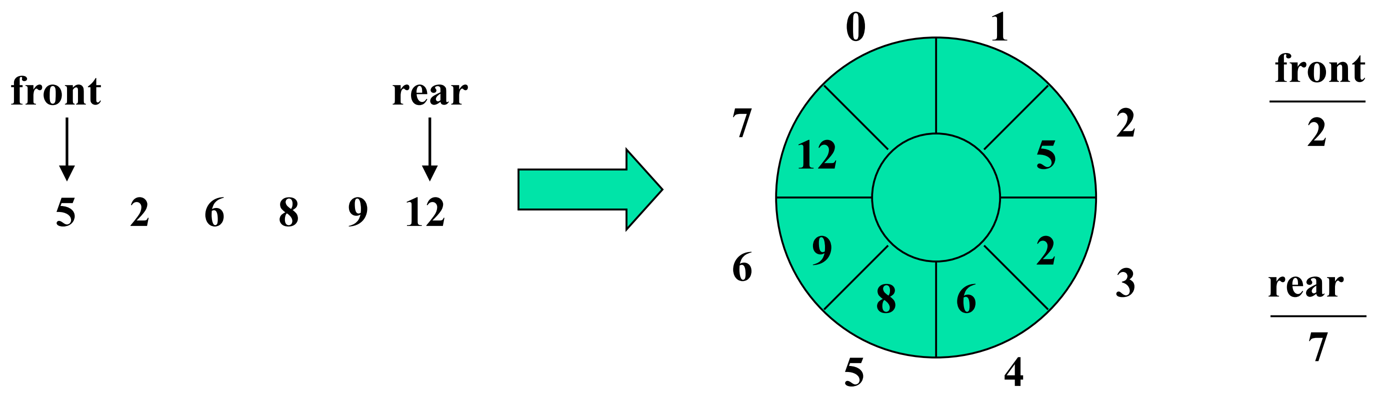
enqueue(21) ??

Queue using Array

- We have inserts and removal running in constant time but we created a new problem.
- Cannot insert new elements even though there are two places available at the start of the array.
- Solution: allow the queue to "**wrap around**".

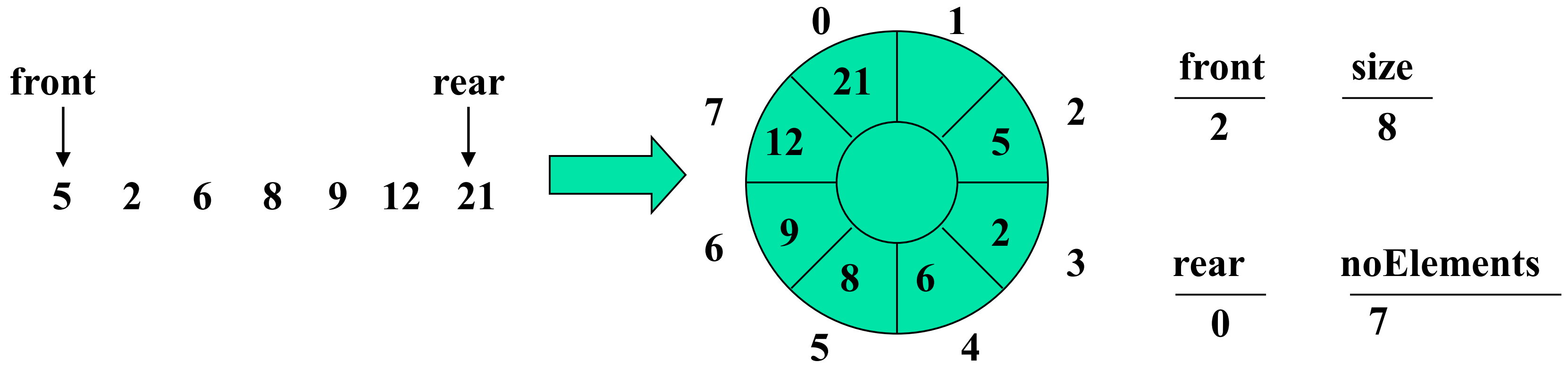
Queue using Array

- Basic idea is to picture the array as a *circular array*.



Queue using Array

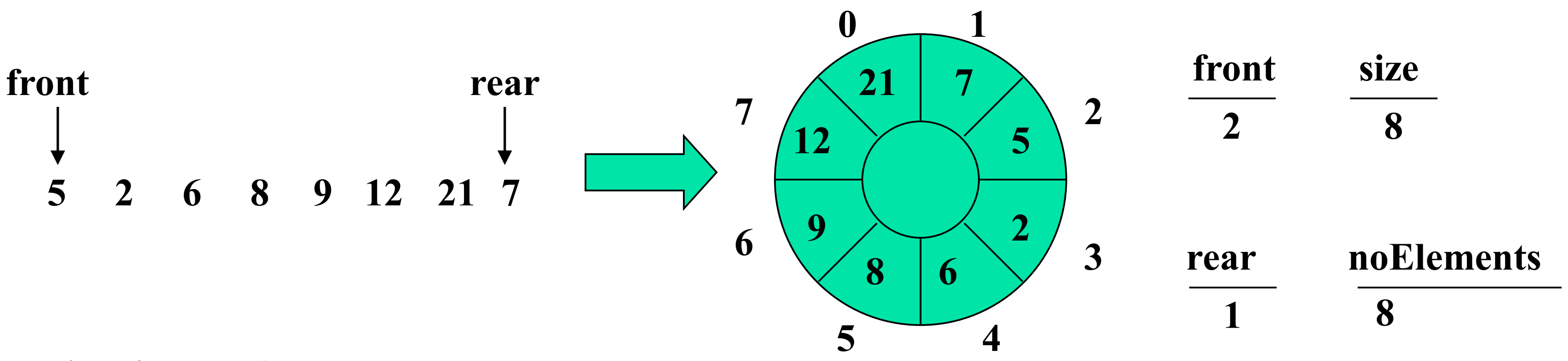
enqueue(21)



```
void enqueue(int x)
{
    rear = (rear+1)%size;
    array[rear] = x;
       noElements = noElements+1;
}
```

Queue using Array

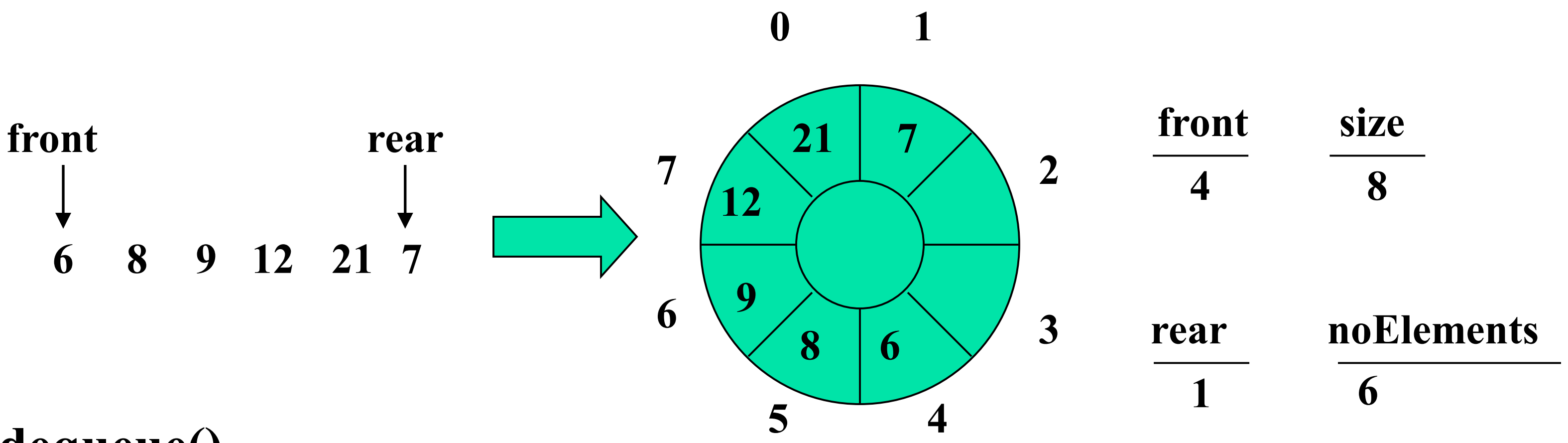
enqueue(7)



```
int isFull()
{
   return noElements == size;
}

int isEmpty()
{
      return noElements == 0;
}
```

Queue using Array

dequeue()



```
int dequeue()
{
   int x = array[front];
      front = (front+1)%size;
      noElements = noElements-1;
      return x;
}
```

Another implementation of Queue

Introduction to the Queue ADT

- Queue: a FIFO (first in, first out) data structure.
- Examples:
  - people in line at the theatre box office
  - print jobs sent to a printer
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list

Queue Locations and Operations

- rear: position where elements are added
- front: position from which elements are removed
- enqueue: add an element to the rear of the queue
- dequeue: remove an element from the front of a queue

Contents of IntQueue.h

Contents of `IntQueue.h`

```
// Specification file for the IntQueue class
#ifndef INTQUEUE_H
#define INTQUEUE_H

class IntQueue
{
private:
   int *queueArray;  // Points to the queue array
   int queueSize;    // The queue size
   int front;        // Subscript of the queue front
   int rear;         // Subscript of the queue rear
   int numItems;     // Number of items in the queue
```

Contents of IntQueue.h (Continued)

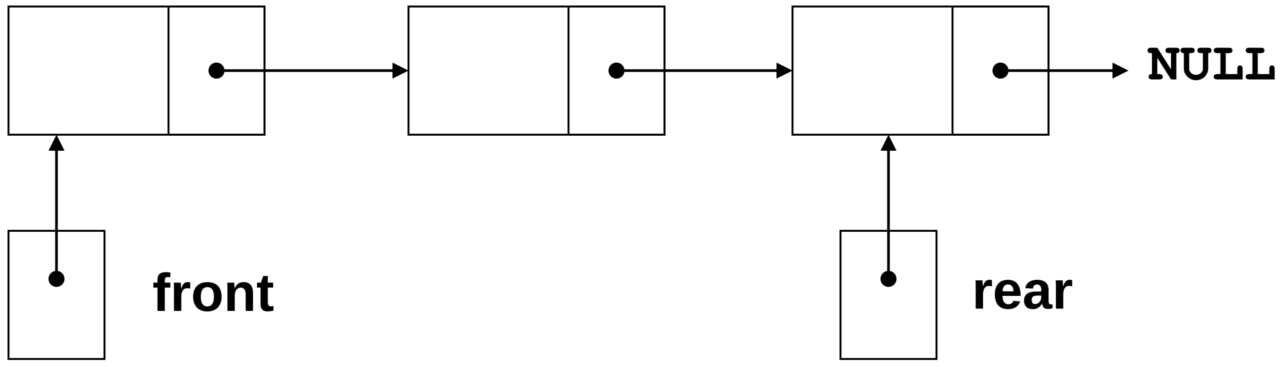
```
public:
    // Constructor
    IntQueue(int);

    // Copy constructor
    IntQueue(const IntQueue &);

    // Destructor
    ~IntQueue();

    // Queue operations
    void enqueue(int);
    void dequeue(int &);
    bool isEmpty() const;
    bool isFull() const;
    void clear();
};
#endif
```

(See IntQueue.cpp for the implementation)

Dynamic Queues

Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- Allows dynamic sizing, avoids issue of shifting elements or wrapping indices



Implementing a Queue

- Programmers can program their own routines to implement queue operations
- Can also use the implementation of queue and dequeue available in the STL

The STL `deque` and `queue` Containers

The STL `deque` and `queue` Containers

- `deque`: a double-ended queue. Has member functions to enqueue (`push_back`) and dequeue (`pop_front`)
- `queue`: container ADT that can be used to provide queue as a `vector`, list, or `deque`. Has member functions to enque (`push`) and dequeue (`pop`)

Defining a `queue`

- Defining a queue of `char`s, named cQueue, implemented using a deque:

```
deque<char> cQueue;
```

- implemented using a queue:

```
queue<char> cQueue;
```

- implemented using a `list`:

```
queue< char, list<char> > cQueue;
```

- Spaces are required between consecutive >>, << symbols

Use of Queues

- Out of the numerous uses of the queues, one of the most useful is ***simulation***.
- A simulation program attempts to model a real-world phenomenon.
- Many popular video games are simulations, e.g., SimCity, Flight Simulator etc.
- Each object and action in the simulation has a counterpart in real world.

Uses of Queues

- If the simulation is accurate, the result of the program should mirror the results of the real-world event.
- Thus it is possible to understand what occurs in the real-world without actually observing its occurrence.
- Let us look at an example. Suppose there is a bank with four **tellers**.

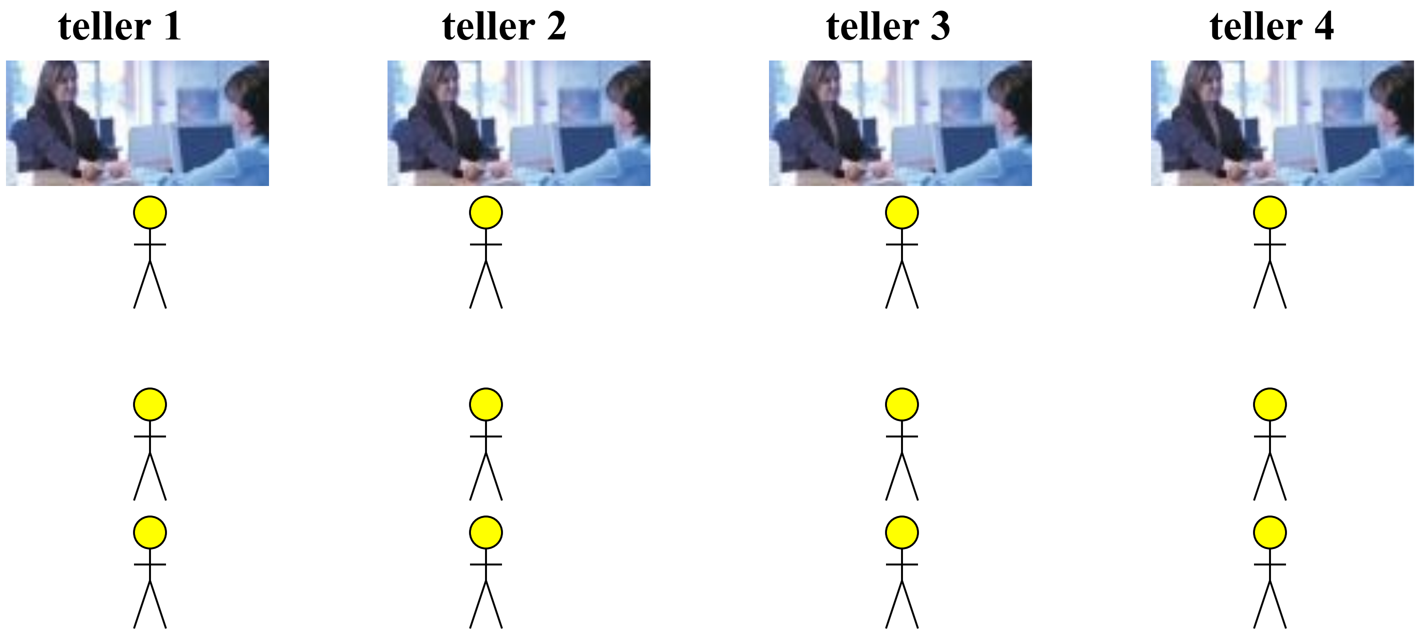
Simulation of a Bank

- A customer enters the bank at a specific time (t_1) desiring to conduct a transaction.
- Any one of the four tellers can attend to the customer.
- The transaction (withdraw, deposit) will take a certain period of time (t_2).
- If a teller is free, the teller can process the customer’s transaction immediately and the customer leaves the bank at t_1+t_2.

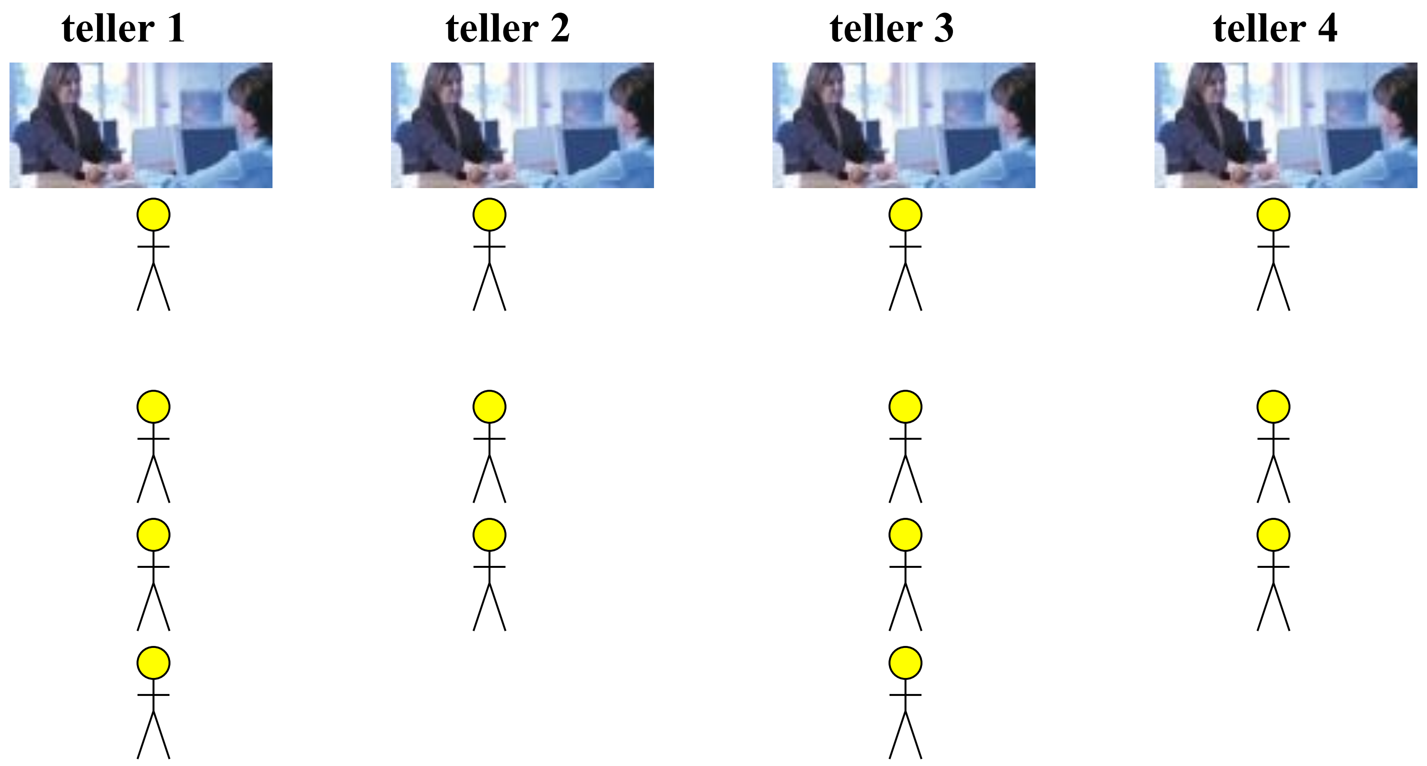
Simulation of a Bank

- It is possible that none of the four tellers is free in which case there is a line of customers are *each* teller.
- An arriving customer proceeds to the back of the shortest line and waits for his turn.
- The customer leaves the bank at t_2 time units after reaching the front of the line.
- The time spent at the bank is t_2 plus time waiting in line.

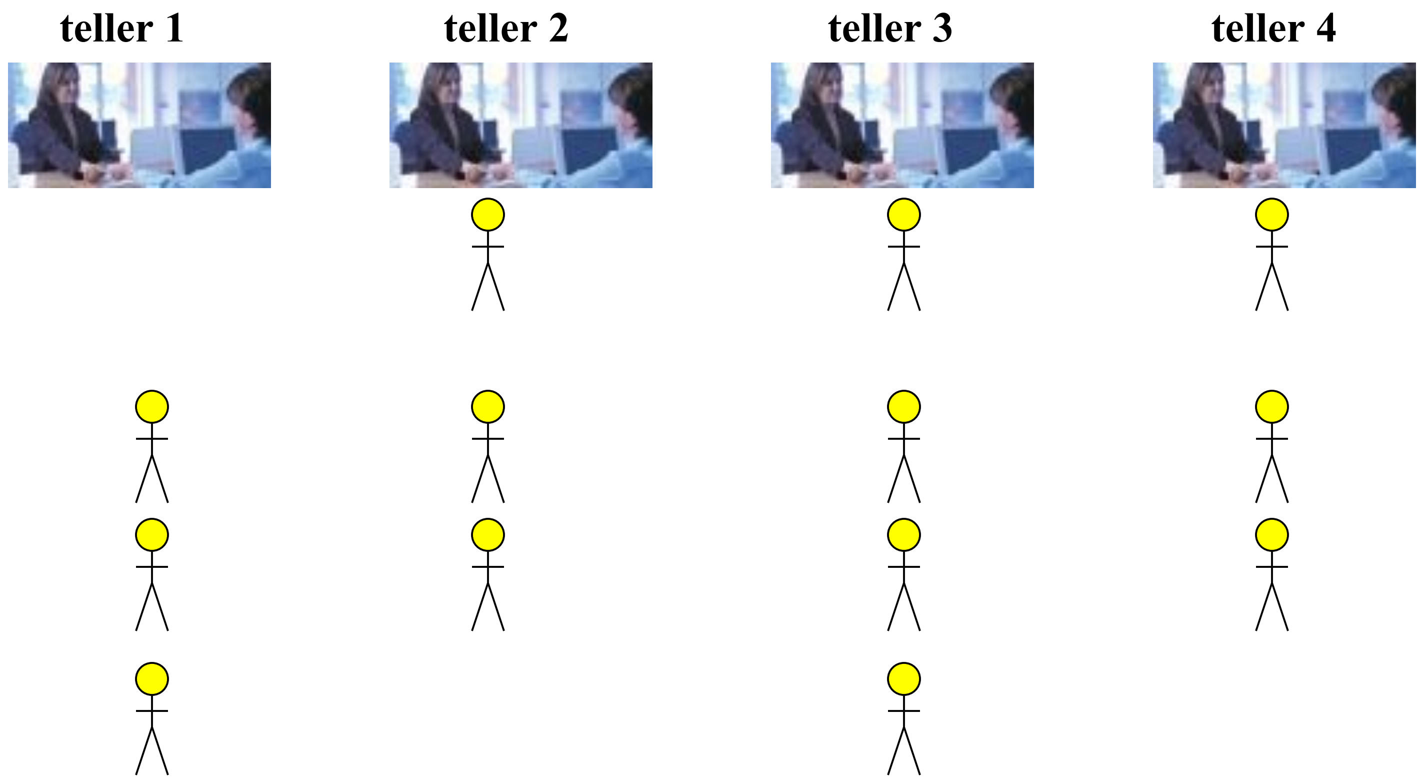
Simulation of a Bank



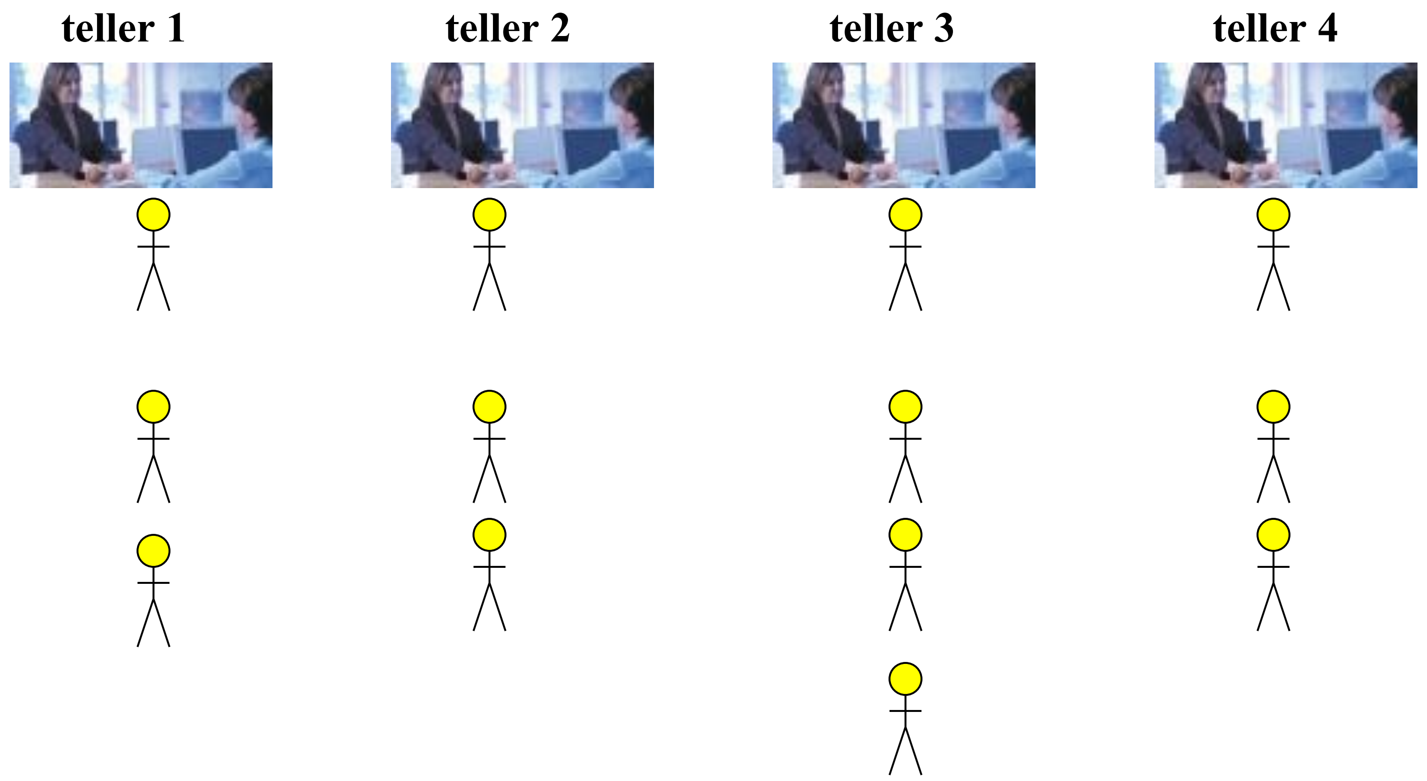
Simulation of a Bank



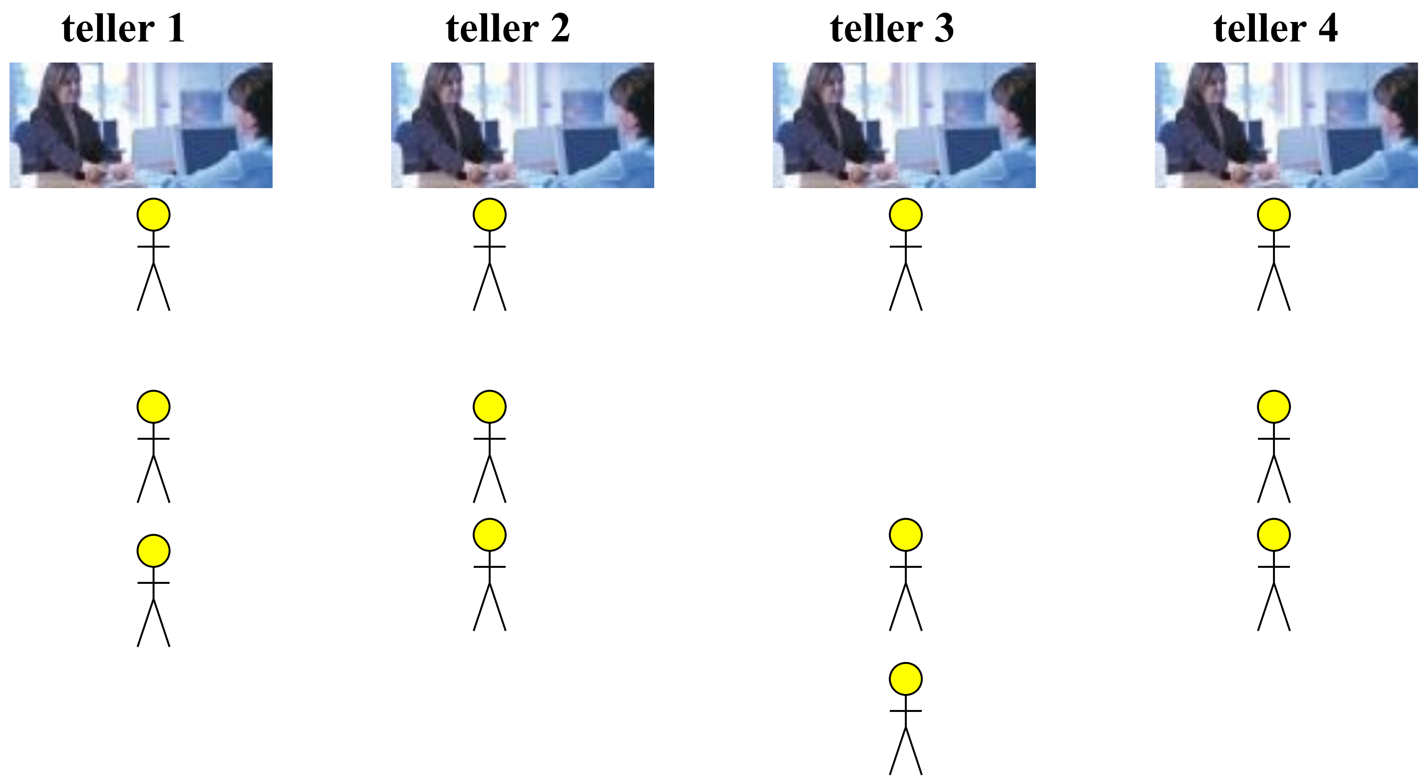
Simulation of a Bank



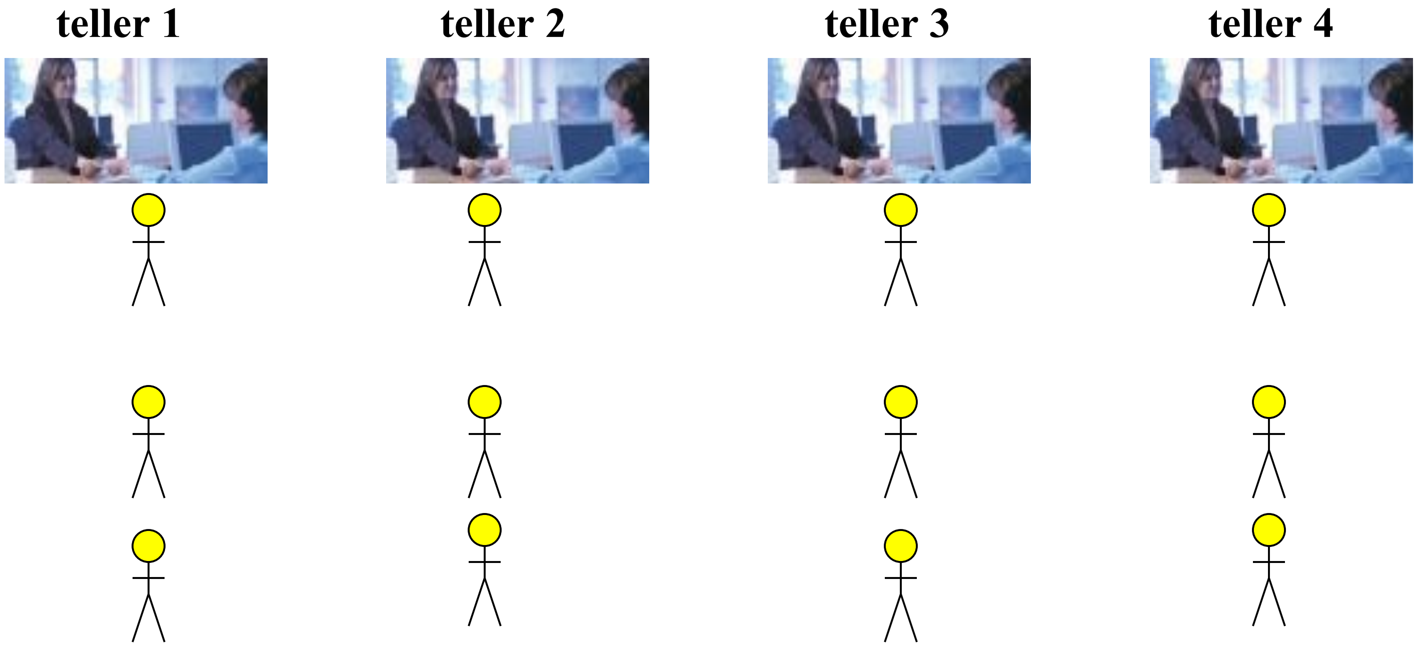
Simulation of a Bank



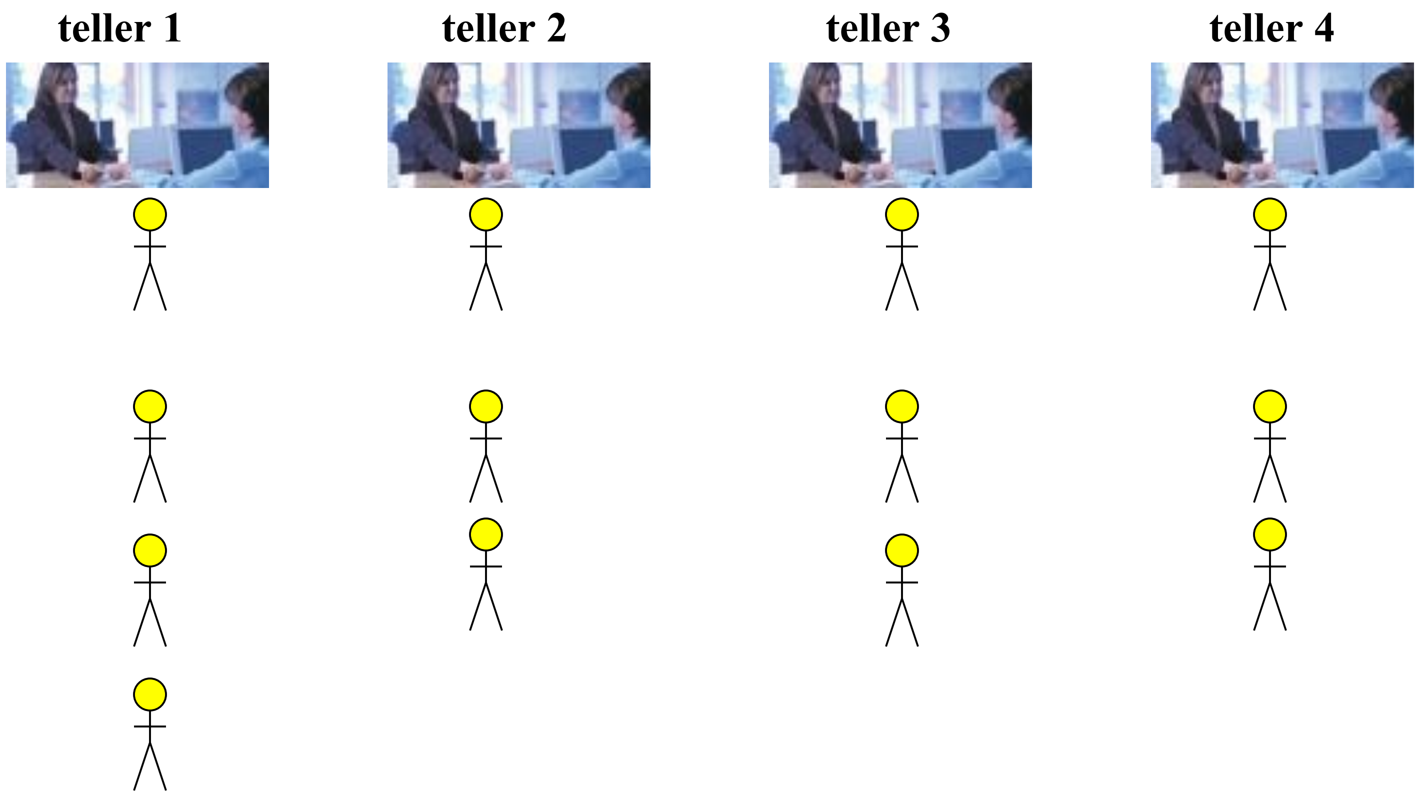
Simulation of a Bank



Simulation of a Bank



Simulation of a Bank



Simulation Models

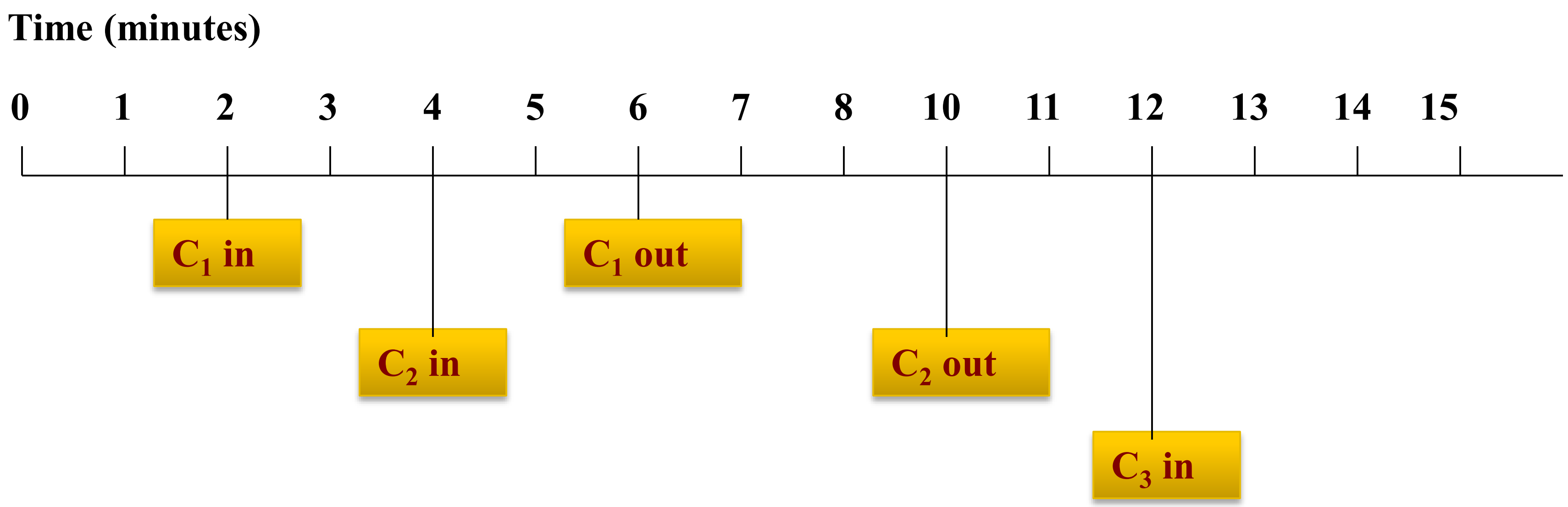
- Two common models of simulation are **time-based** simulation and **event-based** simulation.
- In time-based simulation, we maintain a timeline or a clock.
- The clock ticks. Things happen when the time reaches the moment of an event.

Timeline based Simulation

- Consider the bank example. All tellers are free.
- Customer C_1 comes in at time 2 minutes after bank opens.
- His transaction (withdraw money) will require 4 minutes.
- Customer C_2 arrives 4 minutes after the bank opens. Will need 6 minutes for transaction.
- Customer C3 arrives 12 minutes after the bank opens and needs 10 minutes.

Timeline based Simulation

- Events along the timeline:



Timeline based Simulation

- We could write a main clock loop as follows:

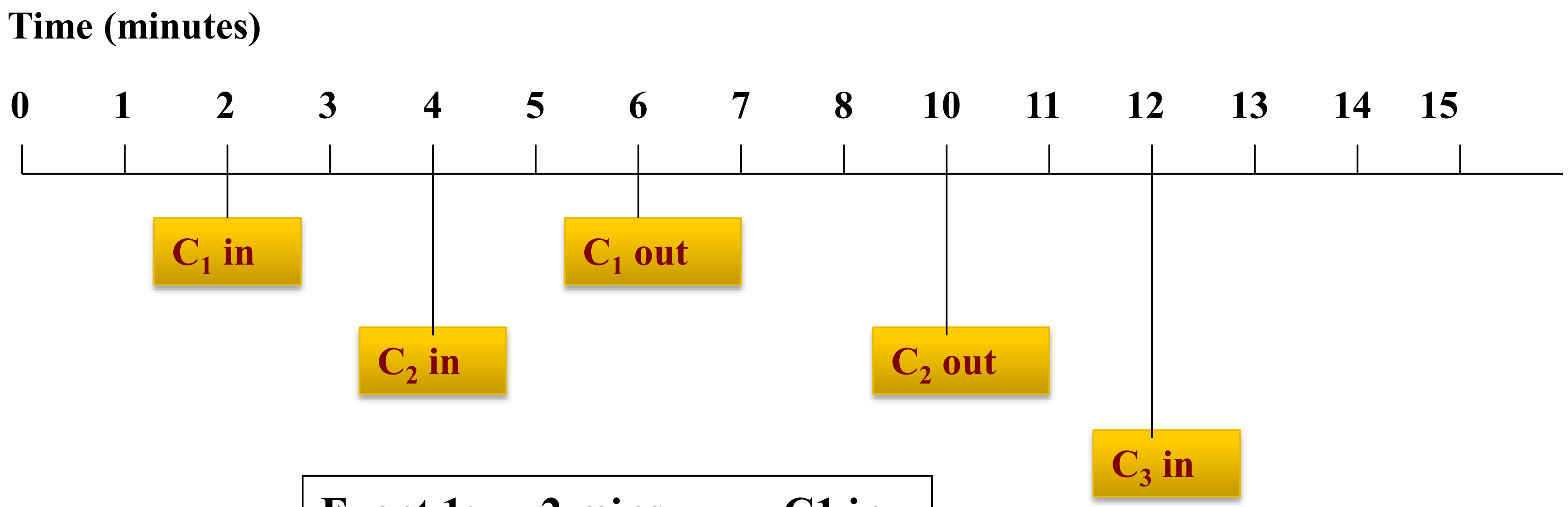
```
clock = 0;
while( clock <= 24*60 ) { // one day
    read new customer;
    **if** customer.arrivaltime == clock
        insert into shortest queue;
    check the customer at head of all four queues.
    **if** transaction is over, remove from queue.
    clock = clock + 1;  //after one minute
}
```

Event based Simulation

- Don’t wait for the clock to tic until the next event.
- Compute the time of next event and maintain a list of events in increasing order of time.
- Remove a event from the list in a loop and process it.

Event based Simulation

- Events



Event 1:	2 mins	C1 in
Event 2:	4 mins	C2 in
Event 3:	6 mins	C1 out
Event 4:	10 mins	C2 out
Event 5:	12 mins	C3 in

Event based Simulation

- Maintain a queue of events.
- Remove the event with the earliest time from the queue and process it.
- As new events are created, insert them in the queue.
- A queue where the dequeue operation depends not on FIFO, is called a ***priority queue***.

Event based Bank Simulation

- Development of the C++ code to carry out the simulation.
- We will need the queue data structure.
- We will need the priority queue.
- Information about arriving customers will be placed in an input file.
- Each line of the file contains the items (arrival time,transaction duration)

Arriving Customers' File

- Here are a few lines from the input file.

```
00 30 10  <- customer 1
00 35 05  <- customer 2
00 40 08
00 45 02
00 50 05
00 55 12
01 00 13
01 01 09
```

- "00 30 10" means Customer 1 arrives 30 minutes after bank opens and will need 10 minutes for his transaction.
- "01 01 09" means customer arrives one hour and one minute after bank opens and transaction will take 9 minutes.

Simulation Procedure

- The first event to occur is the arrival of the first customer.
- This event placed in the priority queue.
- Initially, the four teller queues are empty.
- The simulation proceeds are follows:
- When an arrival event is removed from the priority queue, a node representing the customer is placed on the shortest teller queue.

Simulation Procedure

- If that customer is the only one on a teller queue, a event for his departure is placed on the priority queue.
- At the same time, the next input line is read and an arrival event is placed in the priority queue.
- When a departure event is removed from the event priority queue, the customer node is removed from the teller queue.

Simulation Procedure

- The total time spent by the customer is computed: it is the time spent in the queue waiting and the time taken for the transaction.
- This time is added to the total time spent by all customers.
- At the end of the simulation, this total time divided by the total customers served will be average time spent by customers.
- The next customer in the queue is now served by the teller.
- A departure event is placed on the event queue.

Code for Simulation

```
#include <iostream>
#include <string>
#include <strstream.h>

#include "Customer.cpp"
#include "Queue.h"
#include "PriorityQueue.cpp"
#include "Event.cpp"

Queue q[4];        // teller queues
PriorityQueue pq; //eventList;
int totalTime;
int count = 0;
int customerNo = 0;
```

Code for Simulation

```
main (int argc, char *argv[])
{
   Customer* c;
   Event* nextEvent;

   // open customer arrival file
  ifstream data("customer.dat", ios::in);

   // initialize with the first arriving
   // customer.
  readNewCustomer(data);
```

Code for Simulation

```
while( pq.length() > 0 )
 {
    nextEvent = pq.remove();
    c = nextEvent->getCustomer();
    if( c->getStatus() == -1 ){  // arrival event
          int arrTime = nextEvent->getEventTime();
          int duration = c->getTransactionDuration();
          int customerNo = c->getCustomerNumber();
          processArrival(data, customerNo,
                        arrTime, duration , nextEvent);
      }
      else {  // departure event
          int qindex = c->getStatus();
          int departTime = nextEvent->getEventTime();
          processDeparture(qindex, departTime,
nextEvent);
      }
  }
```

Code for Simulation

```
void readNewCustomer(ifstream& data)
{
  int hour,min,duration;
  if (data >> hour >> min >> duration) {
      customerNo++;
      Customer* c = new Customer(customerNo,
                        hour*60+min, duration);
      c->setStatus( -1 ); // new arrival
      Event* e = new Event(c, hour*60+min );
      pq.insert( e ); // insert the arrival event
  }
  else {
      data.close(); // close customer file
  }
}
```

Code for Simulation

```
int processArrival(ifstream &data, int customerNo,
                  int arrTime, int duration,
                  Event* event)
{
  int i, small, j = 0;
  // find smallest teller queue
   small = q[0].length();
   for(i=1; i < 4; i++ )
      if( q[i].length() < small ){
            small = q[i].length(); j = i;
      }

  // put arriving customer in smallest queue
  Customer* c = new Customer(customerNo, arrTime,
                        duration );
  c->setStatus(j);  // remember which queue the
  customer goes in
  q[j].enqueue(c);
```

Code for Simulation

```
    // check if this is the only customer in the.
    //  queue. If so, the customer must be marked for
    // departure by placing him on the event queue.

    if( q[j].length() == 1 ) {
        c->setDepartureTime( arrTime+duration);
        Event* e = new Event(c, arrTime+duration );
        pq.insert(e);
    }

    // get another customer from the input
    readNewCustomer(data);
}
```

Code for Simulation

```
int processDeparture(int qindex, int departTime,Event* event)
{
   Customer* cinq = q[qindex].dequeue();

   int waitTime = departTime - cinq->getArrivalTime();
   totalTime = totalTime + waitTime;
   count = count + 1;

   // if there are any more customers on the queue, mark the
   // next customer at the head of the queue for departure
   // and place him on the eventList.
   if( q[qindex].length() > 0 ) {
        cinq = q[qindex].front();
        int etime = departTime + cinq->getTransactionDuration();
        Event* e = new Event( cinq, etime);
        pq.insert( e );
   }
}
```

Code for Simulation

```
// print the final avaerage wait time.

double avgWait = (totalTime*1.0)/count;
cout << "Total time:    " << totalTime << endl;
cout << “Customer:      " << count << endl;
cout  << "Average wait: " << avgWait << endl;
```

You may be thinking that the complete picture of simulation is not visible.
How will we run this simulation? Another important tool in the simulation is animation. You have seen the animation of traffic. Cars are moving and stopping on the signals. Signals are turning into red, green and yellow. You can easily understand from the animation.

If the animation is combined with the simulation, it is easily understood. We have an animated tool here that shows the animation of the events. A programmer can see the animation of the bank simulation.

With the help of this animation, you can better understand the simulation.

In this animation, you can see the Entrance of the customers, four tellers, priority queue and the Exit. The customers enter the queue and as the tellers are free. They go to the teller straight. Customer C1<30, 10> enters the bank. The customer C1 enters after 30 mins and he needs 10 mins for the transaction. He goes to the teller 1. Then customer C2 enters the bank and goes to teller 2. When the transaction ends, the customer leaves the bank. When tellers are not free, customer will wait in the queue. In the event priority queue, we have different events. The entries in the priority queue are like *arr, 76* (arrival event at 76 min) or *q1, 80* (event in q1 at 80 min) etc. Let’s see the statistics when a customer leaves the bank. At exit, you see the customer leaving the bank as C15<68, 3><77, 3>, it means that the customer C15 enters the bank at 68 mins and requires 3 mins for his transaction. He goes to the teller 4 but the teller is not free, so the customer has to wait in the queue. He leaves the bank at 77 mins.

Code for Simulation

```
// print the final avaerage wait time.

double avgWait = (totalTime*1.0)/count;
cout << "Total time:    " << totalTime << endl;
cout << “Customer:      " << count << endl;
cout  << "Average wait: " << avgWait << endl;
```

Priority Queue

```
#include "Event.cpp"
#define PQMAX 30

class PriorityQueue {
public:
   PriorityQueue() {
        size = 0; rear = -1;
   };
   ~PriorityQueue() {};

   int full(void)
   {
        return ( size == PQMAX ) ? 1 : 0;
   };
```

Priority Queue

```
Event* remove()
{
    if( size > 0 ) {
          Event* e = nodes[0];
           for(int j=0; j < size-2; j++ )
                  nodes[j] = nodes[j+1];

           size = size-1; rear=rear-1;
           if( size == 0 ) rear = -1;

           return e;
    }
    return (Event*)NULL;
    cout << "remove - queue is empty." << endl;
};
```

Priority Queue

```
   int insert(Event* e)
   {
       if( !full() ) {
              rear = rear+1;
              nodes[rear] = e;
              size = size + 1;
              sortElements(); // in ascending order
              return 1;
       }
       cout << "insert queue is full." << endl;
       return 0;
   };

   int length() { return size; };
};
```