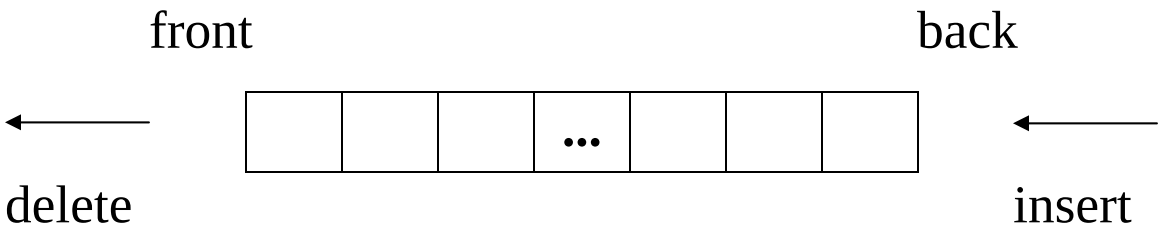
Lecture 7: ADT Queue

Read: Chpt. 7, Carrano.

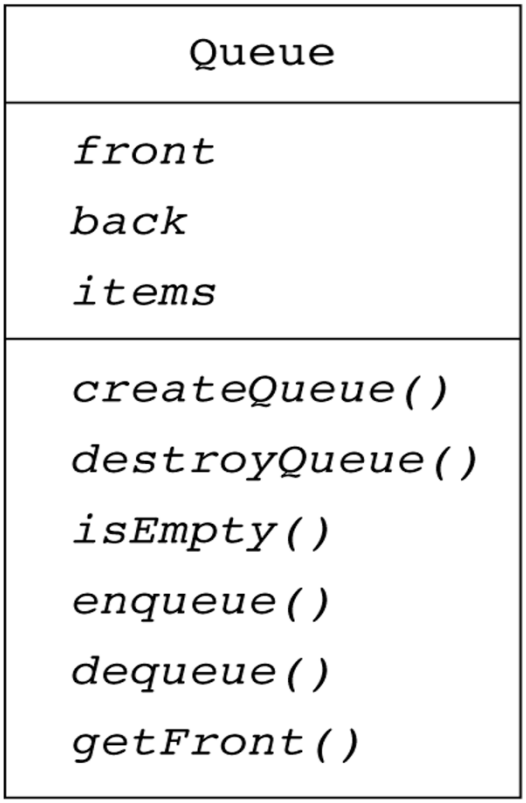
Queue: First-In-First-Out (FIFO) linear ADT; insertions can only take place at one end (back) and removals at the other end (front) of the ADT.



Applications:

- Resource allocation: Access to resources by processes inside a computer.
- Simulation of real-time systems.
- Facilitate certain types of traversals of more complex data structures (e.g., breadth-first traversal of trees)

UML for class Queue:



+createQueue( )

+destroyQueue( )

+isEmpty( ): boolean {query}

+enqueue(in newItem: QueueItemType) throw QueueException

+dequeue(out queueFront: QueueItemType) throw QueueException

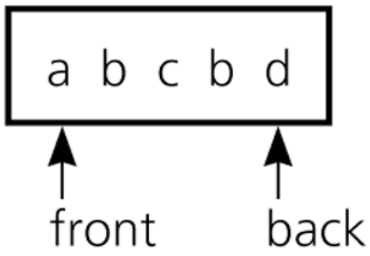
+getFront(out queueFront: QueueItemType) throw QueueException

Application: Palindrome recognizer.

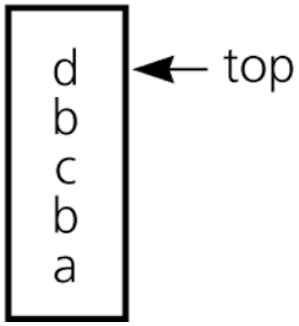
Example:

String: abcbd

Queue:



Stack:



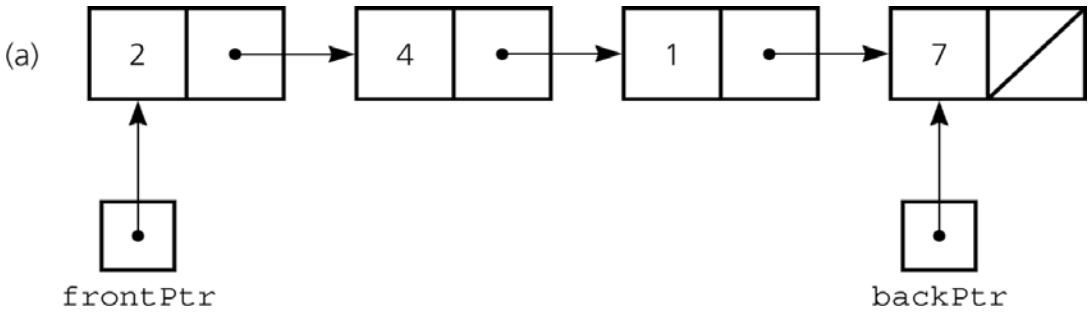
Algorithm:

```
isPal(in str: string) boolean
{  aQueue.createQueue( )
   aStack.createStack( )
   while ( str != empty) do
   {  nextChar = getNextChar(str)
      aQueue.enqueue(nextChar)
      aStack.push(nextChar) }    // endwhile
   equal = true;
   while (!aQueue.isEmpty( ) and equal)
   {  aQueue.getFront(queueFront)
      aStack.getTop(stackTop)
      if (queueFront = stackTop)
      {  aQueue.dequeue( )
         aStack.pop( ) }
      else equal = false }         // endwhile
   return equal
}  // endisPal
```

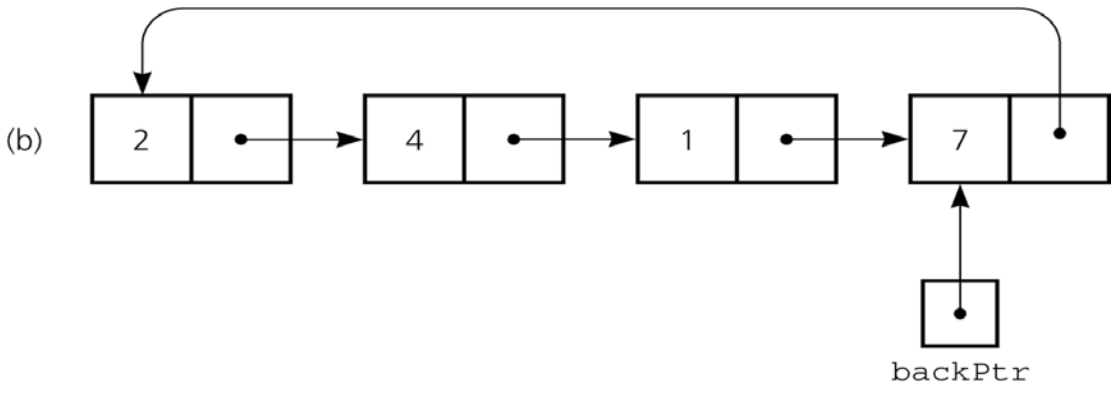
Queue Implementations:

1. Pointer-based linked implementation:

   (a) Using singly linked list with both front & back pointers:



   (b) Using circular linked list with a single back pointer:



   Observe that

   frontPtr = backPtr→next

```
// Header file QueueP.h for the ADT queue.
// Pointer-based implementation using singly linked list

#include "QueueException.h"
typedef desired-type-of-queue-item QueueItemType;

class Queue
{
public:
// constructors and destructor:
   Queue();                          // default constructor
   Queue(const Queue& Q);            // copy constructor
   ~Queue();                         // destructor
```

```
// Queue operations:
   bool isEmpty() const;
   void enqueue(QueueItemType newItem) throw
                                          (QueueException);
   void dequeue() throw (QueueException);
   void dequeue(QueueItemType& queueFront) throw
                                          (QueueException);
  void getFront(QueueItemType& queueFront) const throw
                                          (QueueException);

  private:
  struct QueueNode
  {  QueueItemType item;
     QueueNode *next;
  }; // end struct
  QueueNode *frontPtr;
  QueueNode *backPtr;
}; // end class
// End of header file.
```

```
// Implementation file QueueP.cpp for the ADT queue.
// Pointer-based implementation.
#include "QueueP.h" // header file
#include <cstddef>
#include <cassert>

Queue::Queue() : backPtr(NULL), frontPtr(NULL)
{
} // end default constructor

Queue::Queue(const Queue& Q)
{ // Implementation left as an exercise (Exercise 6)
      …
} // end copy constructor
```

```
Queue::~Queue()
{
   while (!isEmpty())
      dequeue();
   assert ((backPtr == NULL) && (frontPtr == NULL));
} // end destructor

bool Queue::isEmpty() const
{
   return bool(backPtr == NULL);
} // end isEmpty

void Queue::enqueue(QueueItemType newItem)
{
   QueueNode *newPtr = new QueueNode;
   if (newPtr == NULL)            // check allocation
      throw QueueException(
         "QueueException: enqueue cannot allocate memory");
```

```
   else
   {  // allocation successful
      newPtr->item = newItem;
      newPtr->next = NULL;
     // insert the new node
     if (isEmpty())
        // insertion into empty queue
          frontPtr = newPtr;
     else
        // insertion into nonempty queue
          backPtr->next = newPtr;

      backPtr = newPtr; // new node is at back
   } // end if
} // end enqueue
```

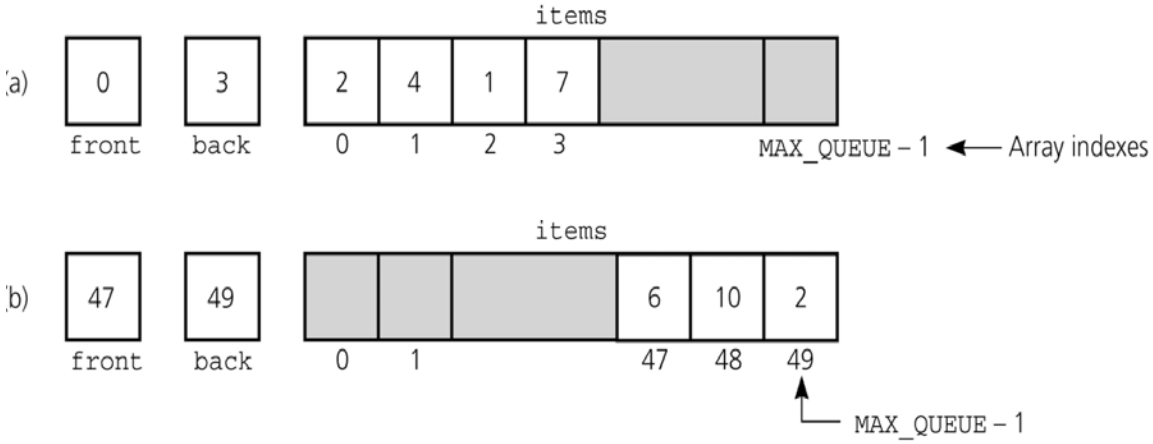
```
void Queue::dequeue()
{  if (isEmpty())
      throw QueueException(
         "QueueException: empty queue, cannot dequeue");
   else
   {  // queue is not empty; remove front
      QueueNode *tempPtr = frontPtr;
      if (frontPtr == backPtr)
      {  // yes, one node in queue
         frontPtr = NULL;
         backPtr = NULL;
      }
      else
         frontPtr = frontPtr->next;
      tempPtr->next = NULL; // defensive strategy
      delete tempPtr;
   } // end if
} // end dequeue
```

```
void Queue::dequeue(QueueItemType& queueFront)
{
  if (isEmpty())
  throw QueueException(
    "QueueException: empty queue, cannot dequeue");
  else
  {  // queue is not empty; retrieve front
    queueFront = frontPtr->item;
    dequeue(); // delete front
  } // end if
} // end dequeue
```

```
void Queue::getFront(QueueItemType& queueFront) const
{
  if (isEmpty())
    throw QueueException(
      "QueueException: empty queue, cannot getFront");
  else
    // queue is not empty; retrieve front
    queueFront = frontPtr->item;
} // end getFront
// End of implementation file.
```

2. Array-based implementation:

Consider the following simple array data structure:

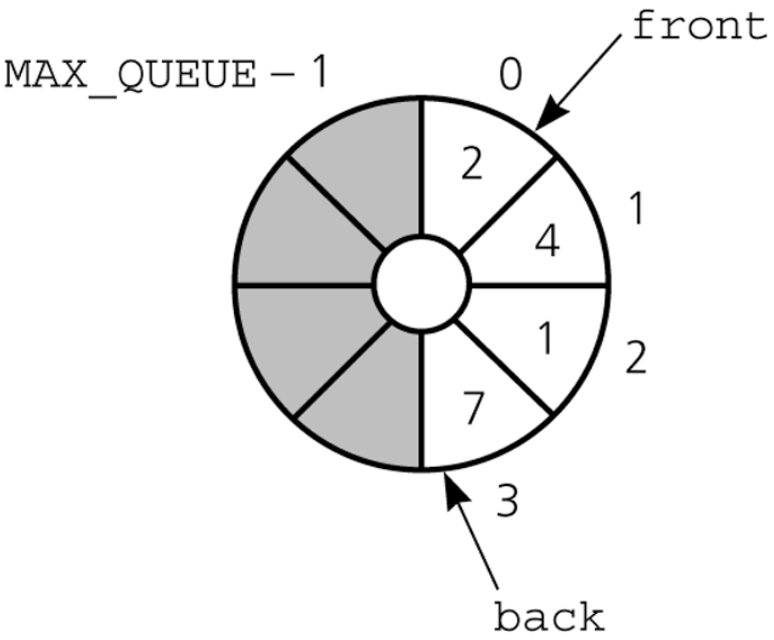


Problem: Can't insert even Queue is not full!

Solution: Array needs to be shifted or ***wrapped around***.

Circular Array Implementation of Queue:

Consider an array with size MAX_QUEUE being wrap around at its two ends.



Initially,

front = 0,

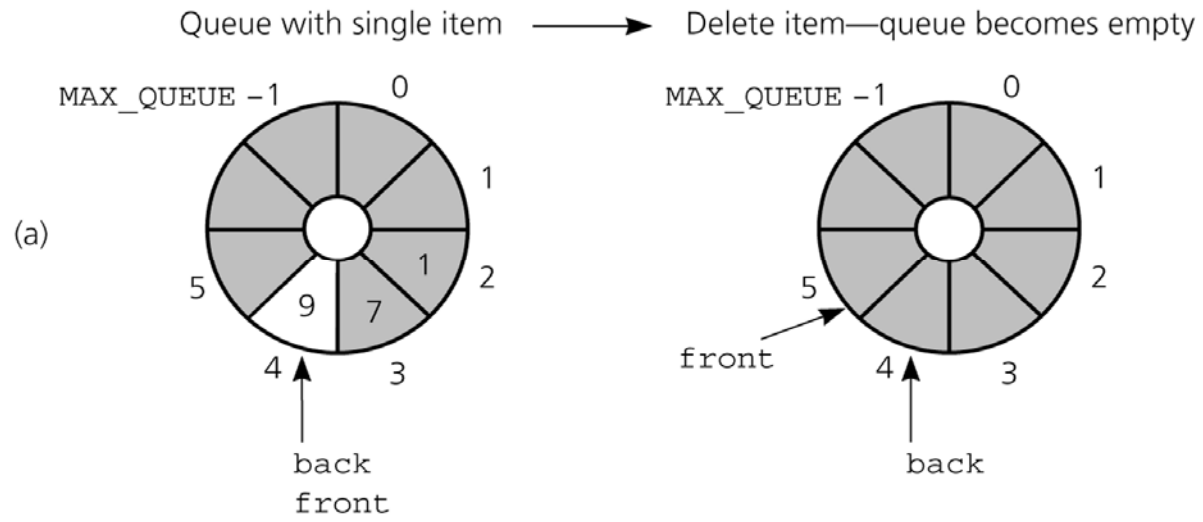
back = MAX_QUEUE – 1.

Content of Queue:

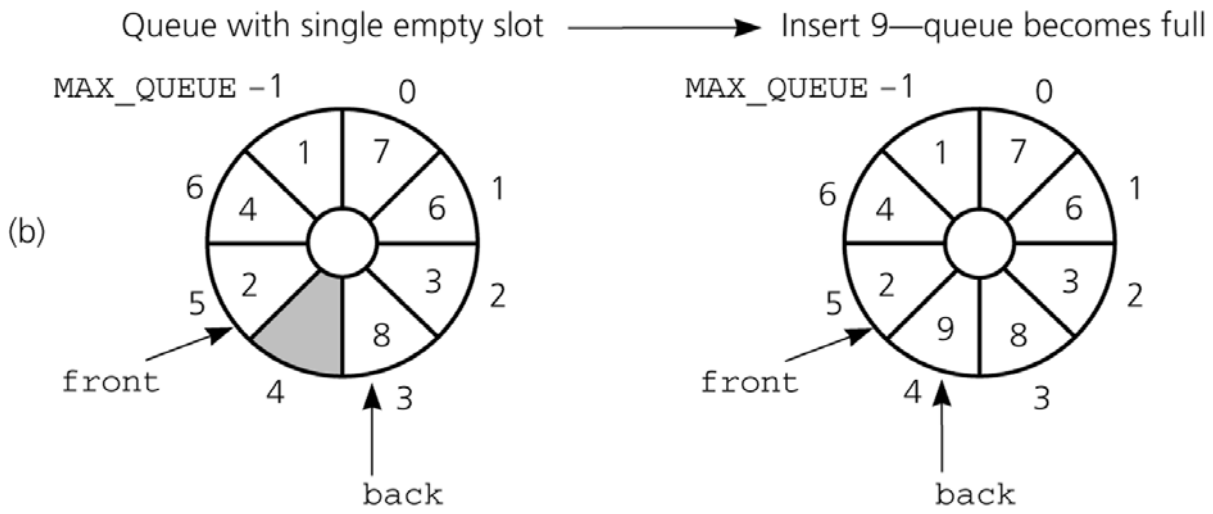
From front to back.

Q: When will the Queue be full/empty?

Empty Queue:



Full Queue:



```
// Header file QueueA.h for the ADT queue.
// Array-based implementation.

#include "QueueException.h"
const int MAX_QUEUE = maximum-size-of-queue;
typedef desired-type-of-queue-item QueueItemType;

class Queue
{
public:
// constructors and destructor:
   Queue(); // default constructor
   // copy constructor and destructor are
   // supplied by the compiler
```

```
// Queue operations:
   bool isEmpty() const;
   void enqueue(QueueItemType newItem)
         throw (QueueException);
   void dequeue() throw (QueueException);
   void dequeue(QueueItemType& queueFront)
         throw (QueueException);
   void getFront(QueueItemType& queueFront) const
         throw (QueueException);

private:
   QueueItemType items[MAX_QUEUE];
   int          front;
   int          back;
   int          count;
}; // end Queue class
// End of header file.
```

```
// Implementation file QueueA.cpp for the ADT queue.
// Circular array-based implementation.
// The array has indexes to the front and back of the
// queue. A counter tracks the number of items
// currently in the queue.

#include "QueueA.h" // header file

Queue::Queue(): front(0), back(MAX_QUEUE-1), count(0)
{
} // end default constructor

bool Queue::isEmpty() const
{
   return bool(count == 0);
} // end isEmpty
```

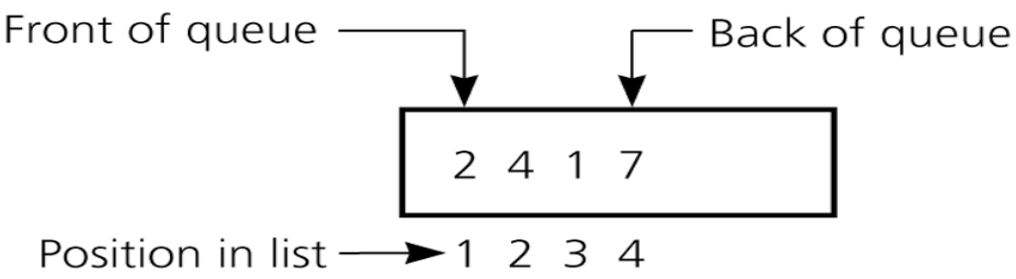
```
void Queue::enqueue(QueueItemType newItem)
{
  if (count == MAX_QUEUE)
    throw QueueException(
          "QueueException: queue full on enqueue");
  else
  {  // queue is not full; insert item
    back = (back+1) % MAX_QUEUE;
    items[back] = newItem;
    ++count;
  } // end if
} // end enqueue
```

```
void Queue::dequeue()
{
   if (isEmpty())
      throw QueueException(
            "QueueException: empty queue, cannot dequeue");
   else
   {  // queue is not empty; remove front
      front = (front+1) % MAX_QUEUE;
      --count;
   } // end if
} // end dequeue
```

```
void Queue::dequeue(QueueItemType& queueFront)
{
  if (isEmpty())
    throw QueueException(
          "QueueException: empty queue, cannot dequeue");
  else
  {  // queue is not empty; retrieve and remove front
    queueFront = items[front];
    front = (front+1) % MAX_QUEUE;
    --count;
  } // end if
} // end dequeue
```

```
void Queue::getFront(QueueItemType& queueFront) const
{
  if (isEmpty())
    throw QueueException(
          "QueueException: empty queue, cannot getFront");
  else
    //  queue is not empty; retrieve front
    queueFront = items[front];
} // end getFront
// End of implementation file.
```

3. ADT List Implementation of Queue:



Queue OP	**List OP**
enqueue(newItem)	insert(getLength()+1,newItem)
dequeue()	remove(1)
getFront(queueFront)	retrieve(1,queueFront)

```
// Header file QueueL.h for the ADT queue.
// ADT list implementation.

#include "ListP.h"                // ADT list operations
#include "QueueException.h"
typedef ListItemType  QueueItemType;

class Queue
{
public:
// constructors and destructor:
   Queue();               // default constructor
   Queue(const Queue& Q); // copy constructor
   ~Queue();              // destructor
```

```
// Queue operations:
   bool isEmpty() const;
   void enqueue(QueueItemType newItem)
         throw (QueueException);
   void dequeue() throw (QueueException);
   void dequeue(QueueItemType& queueFront)
         throw (QueueException);
   void getFront(QueueItemType& queueFront) const
         throw (QueueException);

private:
   List aList;                        // list of queue items
}; // end queue class
// End of header file.
```

```
// Implementation file QueueL.cpp for the ADT queue.
// ADT list implementation.

#include "QueueL.h" // header file

Queue::Queue()
{
} // end default constructor

Queue::Queue(const Queue& Q): aList(Q.aList)
{
} // end copy constructor

Queue::~Queue()
{
} // end destructor
```

```
bool Queue::isEmpty() const
{
  return (aList.getLength() == 0);
} // end isEmpty

void Queue::enqueue(QueueItemType newItem)
{
  try
  {
    aList.insert(aList.getLength()+1, newItem);
  } // end try
  catch (ListException e)
  {
    throw QueueException(
      "QueueException: cannot enqueue item");
  } // end catch
} // end enqueue
```

```
void Queue::dequeue()
{  if (aList.isEmpty())
      throw QueueException(
        "QueueException: empty queue, cannot dequeue");
   else
      aList.remove(1);
} // end dequeue

void Queue::dequeue(QueueItemType& queueFront)
{  if (aList.isEmpty())
      throw QueueException(
        "QueueException: empty queue, cannot dequeue");
   else
   {
      aList.retrieve(1, queueFront);
      aList.remove(1);
   } // end if
} // end dequeue
```

```
void Queue::getFront(QueueItemType& queueFront) const
{
  if (!aList.isEmpty())
    throw QueueException(
      "QueueException: empty queue, cannot getFront");
  else
    aList.retrieve(1, queueFront);
} // end getFront
// End of implementation file.
```

Using Queues for Simulations

Q: What is Simulation?

Imitation of a real world process/system over a period of time; it employs techniques in modeling natural and man-made processes with which statistical data can be generated in summarizing/predicting the performance of the system.

Simulation is used to:

- Study the performance of the system.
- Study changes in the behavior of the system.
- Study the feasibility of building the system.
- Evaluate design alternatives.

Approach:

- Formulating the problem
- Building a model of the physical system
- Developing a simulation program from the model
- Analyzing the outputs to understand the physical system
- Verifying and validating the model
- Study changes in behavior of the physical system

Classification

- Static and Dynamic
- Deterministic and Stochastic
- Continuous and Discrete

Discrete Simulation

- Time driven
- Event driven

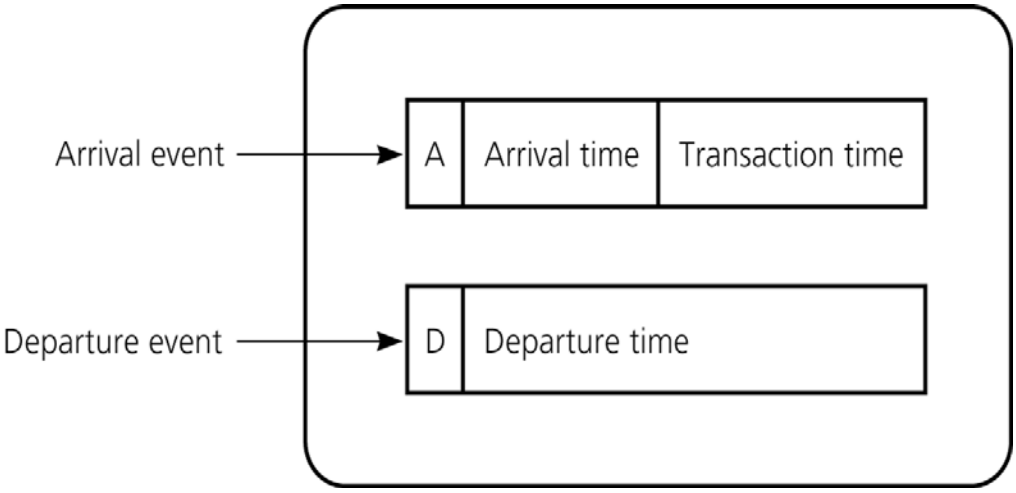
Major Components:

Clock, wait queue, and Event list

Event List:

A list contains all arrival and departure events that will occur.

Typical Instance of an Event List:



departure time

= time service begins + length of transaction.

Time Driven Simulation

During each step of the simulation

- The simulation clock is advanced by one tick.
- The effects of all the events from the event list that are scheduled at that time are simulated. This may lead to adding (deleting) entries to (from) the event list.

Approach:

```
set time to initial time;
establish initial state for all objects;
while not done do
   analyze current state;
   advance time;
   update state for current time
endwhile;
```

Event Driven Simulation

Move from state to state based on the occurrence of specific events.

Bank Teller Example from Carrano:

Goal: To compute average wait-time over all customers through simulation, where wait-time is the elapsed time between arrival and service of a customer.

Figure 1: A bank line at time (a) 0; (b) 12.

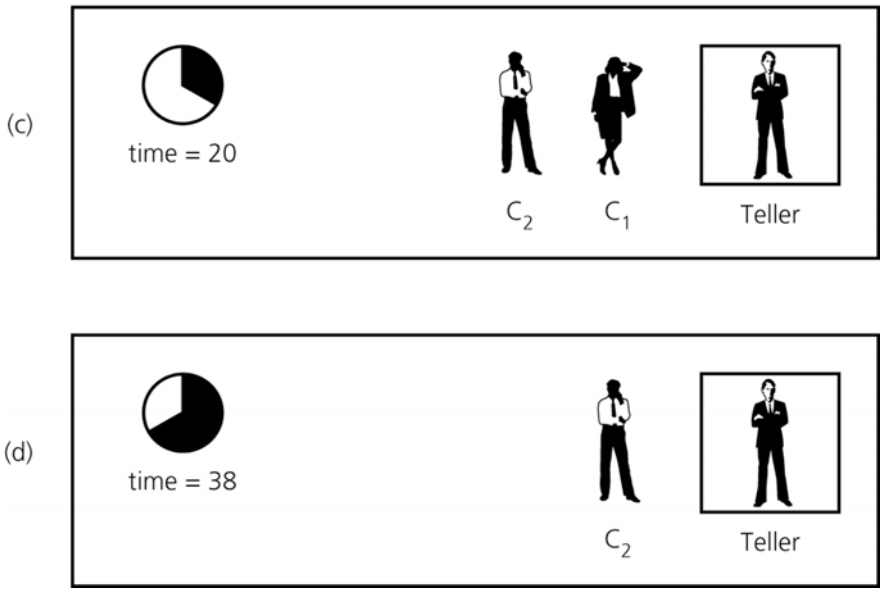


Figure 2: A bank line at time (c) 20; (d) 38.

Two types of Events:

- **Arrival event**: New customer arrives. Customer will be served if teller is idle; otherwise, must wait at end of line.
- **Departure event:** Customer has completed a transaction. Next person in line begins transaction.

Time-driven simulation:

```
set currentTime to 0;
initialize the line to "no customers";
while (currentTime <= time of final event)
   if (arrival event occurs)
      process the arrival event;
   if (departure event occurs)
      process the departure event;
   ++currentTime;
end while;
```

Problem: Only interested in those time at which an arrival/departure event occurs.

Remedy: We need to move from state to state based on the occurrence of specific events.

Event-driven simulation:

```
initialize the line to “no customers”;
while (events remain to be processed)
    currentTime = time of next event;
    if (arrival event occurs)
        process the arrival event;
    if (departure event occurs)
        process the departure event;
end while;
```

Typical Operations:

- Updating the line: Add/Remove customers.
- Updating the event list: Add/Remove events.

Processing Arrival Event:

```
// updating event list
delete arrival event for customer C from event list;
if (customer C begins transaction immediately)
    insert a departure event for C into the event list;
if (not eof)
    read a new arrival event and add it to the event list;
    // tome of event = time specified in file
```

Processing Departure Event:

```
// updating the line
delete current customer C at front of queue;
delete the departure event from the event list;
if (queue is not empty)
   front customer begins transaction;
// updating the event list
if (queue is not empty)
      insert to the event list the departure event for the
          customer at the front of queue;
```