Queues

- Linear list.
- One end is called front.
- Other end is called rear.
- Additions are done at the rear only.
- Removals are made from the front only.

Bus Stop Queue
Bus Stop Queue

Bus Stop

front

rear

Bus Stop Queue

Bus Stop

front

rear
public interface Queue
{
    public boolean isEmpty();
    public Object getFrontElement();
    public Object getRearElement();
    public void put(Object theObject);
    public Object remove();
}
Revisit Of Stack Applications

• Applications in which the stack cannot be replaced with a queue.
  ▪ Parentheses matching.
  ▪ Towers of Hanoi.
  ▪ Switchbox routing.
  ▪ Method invocation and return.
  ▪ Try-catch-throw implementation.

• Application in which the stack may be replaced with a queue.
  ▪ Rat in a maze.
    ▪ Results in finding shortest path to exit.

Wire Routing
Lee’s Wire Router

Label all reachable squares 1 unit from start.

Label all reachable unlabeled squares 2 units from start.
Lee’s Wire Router

Label all reachable unlabeled squares 3 units from start.

Lee’s Wire Router

Label all reachable unlabeled squares 4 units from start.
Label all reachable unlabeled squares 5 units from start.

Label all reachable unlabeled squares 6 units from start.
Lee’s Wire Router

End pin reached. Traceback.
Derive From ArrayLinearList

when front is left end of list and rear is right end

- Queue.isEmpty() => super.isEmpty()
  - O(1) time
- getFrontElement() => get(0)
  - O(1) time
- getRearElement() => get(size() - 1)
  - O(1) time
- put(theObject) => add(size(), theObject)
  - O(1) time
- remove() => remove(0)
  - O(size) time

when rear is left end of list and front is right end

- Queue.isEmpty() => super.isEmpty()
  - O(1) time
- getFrontElement() => get(size() - 1)
  - O(1) time
- getRearElement() => get(0)
  - O(1) time
- put(theObject) => add(0, theObject)
  - O(size) time
- remove() => remove(size() - 1)
  - O(1) time
Derive From ArrayLinearList

- to perform each operation in $O(1)$ time (excluding array doubling), we need a customized array representation.

Derive From ExtendedChain

- when front is left end of list and rear is right end
  - Queue.isEmpty() => super.isEmpty()
    - $O(1)$ time
  - getFrontElement() => get(0)
    - $O(1)$ time
Derive From ExtendedChain

• getRearElement() => getLast() … new method
  – O(1) time
• put(theObject) => append(theObject)
  – O(1) time
• remove() => remove(0)
  – O(1) time

• Queue.isEmpty() => super.isEmpty()
  – O(1) time
• getFrontElement() => getLast()
  – O(1) time
Derive From ExtendedChain

- getRearElement() => get(0)
  - $O(1)$ time
- put(theObject) => add(0, theObject)
  - $O(1)$ time
- remove() => remove(size-1)
  - $O(size)$ time

Custom Linked Code

- Develop a linked class for Queue from scratch to get better performance than obtainable by deriving from ExtendedChain.
Custom Array Queue

• Use a 1D array queue.

• Circular view of array.

Custom Array Queue

• Possible configuration with 3 elements.
Custom Array Queue

• Another possible configuration with 3 elements.

Custom Array Queue

• Use integer variables front and rear.
  – front is one position counterclockwise from first element
  – rear gives position of last element
Add An Element

• Move \texttt{rear} one clockwise.

Add An Element

• Move \texttt{rear} one clockwise.
• Then put into \texttt{queue[rear]}. 
Remove An Element

• Move front one clockwise.

• Then extract from queue[front].
Moving rear Clockwise

- `rear++;`
- `if (rear == queue.length) rear = 0;`
- `rear = (rear + 1) % queue.length;`

Empty That Queue
• When a series of removes causes the queue to become empty, front = rear.
• When a queue is constructed, it is empty.
• So initialize front = rear = 0.
A Full Tank Please

A Full Tank Please
• When a series of adds causes the queue to become full, \texttt{front = rear}.
• So we cannot distinguish between a full queue and an empty queue!

\begin{itemize}
  \item \textbf{Ouch!!!!}
  \item Remedies.
    \begin{itemize}
      \item Don’t let the queue get full.
        \begin{itemize}
          \item When the addition of an element will cause the queue to be full, increase array size.
          \item This is what the text does.
        \end{itemize}
      \item Define a boolean variable \texttt{lastOperationIsPut}.
        \begin{itemize}
          \item Following each \texttt{put} set this variable to \texttt{true}.
          \item Following each \texttt{remove} set to \texttt{false}.
          \item Queue is empty iff \texttt{(front == rear) \&\& !lastOperationIsPut}
          \item Queue is full iff \texttt{(front == rear) \&\& lastOperationIsPut}
        \end{itemize}
    \end{itemize}
\end{itemize}
Ouch!!!!!

- Remedies (continued).
  - Define an integer variable `size`.
    - Following each `put` do `size++`.
    - Following each `remove` do `size--`.
    - Queue is empty iff `(size == 0)`
    - Queue is full iff `(size == queue.length)`
  - Performance is slightly better when first strategy is used.