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 Queue

Bus Stop

front

rear
Bus Stop Queue
The Abstract Class queue

template <class T>

class queue
{

    public:

    virtual ~queue() {}
    virtual bool empty() const = 0;
    virtual int size() const = 0;
    virtual T& front() = 0;
    virtual T& back() = 0;
    virtual void pop() = 0;
    virtual void push(const T& theElement) = 0;

};
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.
Lee’s Wire Router

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

Label all reachable unlabeled squares 3 units from start.
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.
**Lee’s Wire Router**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

End pin reached. Traceback.
Derive From arrayList

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
</table>

0  1  2  3  4  5  6

- when front is left end of list and rear is right end
  - empty() => queue::empty()
    - O(1) time
  - size() => queue::size(0)
    - O(1) time
  - front() => get(0)
    - O(1) time
  - back() => get(size() - 1)
    - O(1) time
Derive From arrayList

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

- pop() => erase(0)
  - $O(\text{size})$ time
- push(theElement) => insert(size(), theElement)
  - $O(1)$ time
when front is right end of list and rear is left end

- empty() => queue::empty()
  - O(1) time
- size() => queue::size(0)
  - O(1) time
- front() => get(size() - 1)
  - O(1) time
- back() => get(0)
  - O(1) time
Derive From array

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6

• `pop() => erase(size() - 1)`
  – O(1) time
• `push(theElement) => insert(0, theElement)`
  – O(size) time
Derive From arrayList

- 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
  - empty() => extendedChain::empty()
    - O(1) time
  - size() => extendedChain::size()
    - O(1) time
• front() => get(0)
   - O(1) time

• back() => getLast() … new method
Derive From ExtendedChain

- push(theElement) => push_back(theElement)
  - O(1) time
- pop() => erase(0)
  - O(1) time
when front is right end of list and rear is left end

- empty() => extendedChain::empty()
  - O(1) time

- size() => extendedChain::size()
  - O(1) time
• front() => getLast()
  - \(O(1)\) time

• back() => get(0)
  - \(O(1)\) time
Derive From extendedChain

- push(theElement) => insert(0, theElement)
  - O(1) time

- pop() => erase(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.

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 `theFront` and `theBack`.
  - `theFront` is one position counterclockwise from first element
  - `theBack` gives position of last element
  - use `front` and `rear` in figures
Push An Element

• Move *rear* one clockwise.
Push An Element

- Move rear one clockwise.
- Then put into queue[rear].
Pop An Element

• Move front one clockwise.
Pop An Element

- Move **front** one clockwise.
- Then extract from queue[front].
Moving rear Clockwise

• rear++;
  
  if (rear == arrayLength) rear = 0;

• rear = (rear + 1) % arrayLength;
Empty That Queue
Empty That Queue

C

rear

front

B

[0] [1] [2] [3] [4] [5]
Empty That Queue
Empty That Queue

- When a series of removes causes the queue to become empty, \( \text{front} = \text{rear} \).
- When a queue is constructed, it is empty.
- So initialize \( \text{front} = \text{rear} = 0 \).
A Full Tank Please

A
B
C

[0] [1] [2] [3] [4] [5]

rear

front
A Full Tank Please
A Full Tank Please
A Full Tank Please

When a series of adds causes the queue to become full, \texttt{front} = \texttt{rear}.

So we cannot distinguish between a full queue and an empty queue!
Ouch!!!!!

- Remedies.
  - Don’t let the queue get full.
    - When the addition of an element will cause the queue to be full, increase array size.
    - This is what the text does.
  - Define a boolean variable `lastOperationIsPush`.
    - Following each `push` set this variable to `true`.
    - Following each `pop` set to `false`.
    - Queue is empty iff `(front == rear) && !lastOperationIsPush`
    - Queue is full iff `(front == rear) && lastOperationIsPush`
Ouch!!!!

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