Queues

- A **Queue** is a sequential organization of items in which first element entered is first removed. They are often referred to as FIFO, which stands for “first in first out.”

- **Examples**: standing in a line, printer queue.

- The basic operations are:
  - $insert(x)$ places $x$ at the beginning of the queue.
  - $remove()$ returns and deletes the item at the end of the queue.
### Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Queue</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateQueue</td>
<td>()</td>
<td></td>
</tr>
<tr>
<td>insert(7)</td>
<td>(7)</td>
<td></td>
</tr>
<tr>
<td>insert(8)</td>
<td>(7,8)</td>
<td></td>
</tr>
<tr>
<td>insert(5)</td>
<td>(7,8,5)</td>
<td></td>
</tr>
<tr>
<td>remove()</td>
<td>(8,5)</td>
<td>7</td>
</tr>
<tr>
<td>remove()</td>
<td>(5)</td>
<td>8</td>
</tr>
</tbody>
</table>
Queue Applications

• Operating systems:
  – Queue of jobs or processes ready to run (waiting for CPU):
  – Queues of processes waiting for I/O.
  – Files sent to printer

• Simulation of real-world queuing systems:
  – Customers in a grocery store, bank, etc.
  – Orders in a factory
  – Hospital emergency room or doctor’s office
  – Telephone calls for airline reservations, customer orders, information, etc.

• Problem applications:
  – Topological ordering: given a sequence of events, and pairs \((a, b)\) indicating that event \(a\) should occur prior to \(b\), provide a schedule.
Queues: Naive Implementation

- Using array:
  - Store items in an array. The head is the first element, and the tail is indicated by a variable `tail`.
  - \textit{insert}(x) is easy: increment \textit{tail}, and insert element.
  - \textit{remove}() is inefficient: all elements have to be shifted. Thus, \textit{remove} is $O(n)$.

```
[Diagram of a queue with elements a, b, c, d]
```

- How can we improve this?
Queues: A Better Implementation

- Keep track of both the *head* and the *tail*.
- To remove, increment *front*.

**Example:**

```
  a  b  c  d  
head = 0  tail = 3
```

Remove `a`:

```
  b  c  d  
head = 1  tail = 3
```

Update head:

```
  b  c  d  
head = 1  tail = 3
```

- There is still a problem. What is it, and how can we fix it?
Queues: Circular Array Implementation

- Previous implementation is $O(1)$ per operation, which is great.
- However, after $n$ inserts (where $n$ is the size of the array), the array is full even if the queue is logically nearly empty.
- Solution: Use wraparound to reuse the cells at the start of the array. To increment, add one, but if that goes past end, reset to zero.

- How do you detect a full or empty queue?
- We will give a simplified implementation for the queue data structure. A better implementation would detect an empty (full) queue before performing a dequeue (enqueue) operation.
Queue C++ Declaration

- Here is a C++ declaration of an integer queue using a “circular” array:

```cpp
class Queue {
private:
    int *queue;
    int cap, head, tail;
public:
    Queue(int s=100) {
        cap = s;
        queue = new int[cap];
        head = 0; tail = 0;
    }
    ~Queue() { delete queue; }
    void Enqueue(int v) {
        queue[tail] = v;
        tail = (tail+1) % cap;
    }
    int Dequeue() {
        int t = queue[head];
        head = (head + 1) % cap;
        return t;
    }
    int Empty() { return (head == tail ); }
    int Full() { return (head == (tail+1)%cap); }
};
```
public class Queue {
    private int[] queue;
    private int cap, head, tail;
    public Queue(int s) {
        cap = s;
        queue = new int[cap];
        head = 0;
        tail = 0;
    }
    public void Enqueue(int v) {
        queue[tail] = v;
        tail = (tail+1) % cap;
    }
    public int Dequeue() {
        int t = queue[head];
        head = (head + 1) % cap;
        return t;
    }
    public int Empty() {
        return (head == tail);
    }
    public int Full() {
        return (head == ((tail+1)%cap));
    }
}
Example of Queue Using Circular Array

- Here we use an array $E[0..3]$ to store the elements and the variables $h$ (head) and $t$ (tail) to keep track of the beginning and end of the queue. The value $R$ is the return value of the operation.

<table>
<thead>
<tr>
<th>Operation</th>
<th>h</th>
<th>t</th>
<th>E0</th>
<th>E1</th>
<th>E2</th>
<th>E3</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>insert(55)</td>
<td>0</td>
<td>1</td>
<td>55</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>insert(-7)</td>
<td>0</td>
<td>2</td>
<td>55</td>
<td>-7</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>insert(16)</td>
<td>0</td>
<td>3</td>
<td>55</td>
<td>-7</td>
<td>16</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>remove()</td>
<td>1</td>
<td>3</td>
<td>55</td>
<td>-7</td>
<td>16</td>
<td>?</td>
<td>55</td>
</tr>
<tr>
<td>insert(-8)</td>
<td>1</td>
<td>0</td>
<td>55</td>
<td>-7</td>
<td>16</td>
<td>-8</td>
<td>?</td>
</tr>
<tr>
<td>remove()</td>
<td>2</td>
<td>0</td>
<td>55</td>
<td>-7</td>
<td>16</td>
<td>-8</td>
<td>-7</td>
</tr>
<tr>
<td>remove()</td>
<td>3</td>
<td>0</td>
<td>55</td>
<td>-7</td>
<td>16</td>
<td>-8</td>
<td>16</td>
</tr>
<tr>
<td>insert(11)</td>
<td>3</td>
<td>1</td>
<td>11</td>
<td>-7</td>
<td>16</td>
<td>-8</td>
<td>?</td>
</tr>
</tbody>
</table>

- Note that some of the values remain physically in the array, but are logically no longer in the queue.
Queues: Linked List Implementation

- We can maintain front and tail pointers.
- remove: advance front.
- insert: add to end of list and adjust tail.

- The details are not very hard to work out, so will not be presented.