Queues

- ADT for storing First In First Out (FIFO) data
  - Two ended sequence of elements (Stack is one ended)
  - Enqueue - elements are added at back of queue
  - Dequeue - elements are removed from front of queue
- Many applications
  - Waiting lines for service, check out
  - Printer queues
  - Filling orders
- Systems software
  - Process queues in time sharing systems
  - Input queues - keyboard, communications ports
  - Buffered read/write requests
- Model of "first come, first served"

An Interface for a Queue

```java
interface QueueI<E> {
    // Add an element at the back of the queue
    public void enqueue(E o);
    // Remove front element in queue and return it
    public E dequeue() throws EmptyQueueException;
    // Return front element on queue, don't remove it
    public E peek() throws EmptyQueueException;
    // Return true if queue is empty
    public boolean isEmpty();
    // Remove all elements from the queue
    public void clear();
}
```
Implementing a Queue with an Array

- Use an array of elements for storage
  - Keep track of number of elements currently in queue (size)
  - Keep index of earliest element placed in queue (front)
    - As queue is emptied, front index increases
  - Keep index of last element placed in queue (back)
    - As queue is filled, back index increases

- Considerations
  - As queue fills, we approach end of array
  - But there may be empty space at beginning if elements were removed
  - For efficient use of array space, treat as circular
    - Reset indices to zero if they would increase beyond array
  - Still need to deal with problem if we need more space than there are elements in the array

An Array-Based Implementation

Figure 7.7
- a) A naive array-based implementation of a queue; b) rightward drift can cause the queue to appear full
An Array-Based Implementation

- A circular array eliminates the problem of rightward drift

Queue with Circular Array

- Invariants are values of front, back, size

```java
class Queue<E> implements QueueI<E> {
    private Object[] data; // Elements stored in array
    private int front; // Index of first element (if there is one)
    private int back; // Index of last element
    private int size; // Number of elements in queue
    public Queue() {
        front = back = size = 0;
        data = new Object[4];
    }
    public boolean isEmpty() { return size == 0; }
    public void clear() {
        front = back = size = 0; data = new Object[4];
    }
}
```
Queue with Circular Array (cont.)

- Increment front as element is removed, and reset to zero if it would be beyond last slot in array

```java
public String dequeue() throws EmptyQueueException {
    if (isEmpty()) throw new EmptyQueueException();
    E element = (E)data[front++];
    --size;
    if (front == data.length) front = 0; // Wrap around
    return element;
}
```

```java
public E peek() throws EmptyQueueException {
    if (isEmpty()) throw new EmptyQueueException();
    return (E)data[front];
}
```

Queue with Circular Array (cont.)

- If queue storage grows, then copy elements to new storage, but shift front-back indexing to start of array

```java
public void enqueue(E o) {
    if (size == data.length) {
        Object[] nData = new Object[2*data.length];
        // Copy elements, shifting front to location zero
        for (int i = 0; i < size; ++i)
            nData[i] = data[(front+i)%(data.length)];
        back = (back + (data.length)) - front;
        front = 0;
        data = nData;
    }
    data[back++] = o;
    if (back == data.length) back = 0; // Wrap around
    ++size;
}
```
Array Implementation Performance

- O(1) time for dequeue, peek, isEmpty
  - Small fixed overhead to implement circularity
- Average case O(1) time for enqueue
  - Worst case is for growth, then O(n) time where n is number of elements currently in queue
  - Results in uneven behavior
  - Note space is not recovered, even if it is never necessary again
  - So space requirement depends on largest number of elements enqueue'd, but not dequeue'd

Linked Implementation of Queue

- With each element in the queue, keep reference to next element
  - Use a small inner container class of parameterized element type
    
    ```java
    private class QueueItem {
      E element;
      QueueItem next;
      QueueItem(E e) { element = e; next = null; }
    }
    ```

- The queue keeps references to the first and last elements
  - Added elements are linked to the end
  - Removed elements are unlinked from the front
Queue Implemented with Links

- Invariants are values of front, back

```java
class Queue<E> implements QueueI<E> {
    private class QueueItem {
        // ... 
    }
    private QueueItem front; // First element (if there is one)
    private QueueItem back;  // Last element

    public Queue() { front = back = null; }
    public boolean isEmpty() { return front == null; }
    public void clear() { front = back = null; }

    public void enqueue(E e) {
        if (isEmpty()) front = back = new QueueItem(e);
        else {
            back.next = new QueueItem(e);
            back = back.next;
        }
    }

    public E dequeue() throws EmptyQueueException {
        if (isEmpty()) throw new EmptyQueueException();
        E element = front.element;
        if (front == back) clear();
        else front = front.next;
        return element;
    }

    public E peek() throws EmptyQueueException {
        if (isEmpty()) throw new EmptyQueueException();
        return front.element;
    }
}
```
Linked Implementation Performance

- O(1) time for all methods
- Storage overhead is one reference per element in queue
- Storage use is exactly proportional to number of elements currently in queue

Reusing LinkedList for Queue

```java
class Queue<E> implements QueueI<E> {
    private LinkedList<E> elements = new LinkedList<E>();
    public boolean isEmpty() { return elements.isEmpty(); }  
    public void clear() { elements.clear(); }
    public void enqueue(E e) { elements.addLast(e); }
    public E dequeue() throws EmptyQueueException {
        if (isEmpty()) throw new EmptyQueueException();
        return elements.removeFirst();
    }
    public E peek() throws EmptyQueueException {
        if (isEmpty()) throw new EmptyQueueException();
        return elements.getFirst();
    }
}
```
Reusing LinkedList for Queue

- This Queue implementation is just a thin wrapper around LinkedList
- Requires LinkedList to have methods to
  - add element to end of list
  - retrieve first element in list
  - retrieve and remove first element
  - check if list is empty
  - reset list to empty
- Java LinkedList provides methods for all of these
  - Java LinkedList implements the Java Queue interface
  - Java also has various implementations of Queue interface
- Performance of Queue depends on performance of LinkedList

Priority Queues

- As we have seen, a Queue is based on First In First Out as the criteria of enqueue and dequeue actions
- Many applications use additional criteria
  - A priority is assigned to each element
  - Primary criteria for dequeue is priority
  - Secondary criteria is order inserted into queue
  - So, elements with high priority jump to head of line
- Examples
  - Checking in customers with imminent departing flights before other customers
  - Treating severest medical cases in emergency rooms
  - Handling preferred customers
  - Servicing critical system processes
Priority Queue Implementations

- What is the nature of priorities of elements?
  - Finite or infinite number of priority values?
- Suppose there is a fixed set of priorities
  - E.g., low-medium-high
  - An implementation could use a list of queues, each corresponding to a priority value, all initially empty
  - Enqueue adds element to appropriate priority queue
  - Dequeue removes element from highest priority queue

Enqueue method of Priority Queue
- Could be the same as for Queue, but elements are comparable according to their priority property
- Could add priority value parameter to enqueue()

Priority Queue with Fixed Priorities

```java
class FixedPriorityQueue implements QueueI {
    private Queue[] pqueues; // Queue for each priority
    public FixedPriorityQueue() {
        pqueues = new Queue[MAXP];
        for (int i = 0; i < MAXP; ++i) pqueues[i] = new Queue();
    }
    public boolean isEmpty() {
        for (int i = 0; i < MAXP; ++i)
            if (!pqueues[i].isEmpty()) return false;
        return true;
    }
}
```
Priority Queue with Fixed Priorities

- All operations are $O(1)$ since MAXP is fixed
- Assume each element knows its priority and lowest priority value is the "best"

```java
// Enqueue puts element in appropriate priority
public void enqueue(Object o) {
    pqueues[o.priority()].enqueue(o);
}

public Object dequeue() throws EmptyQueueException {
    for (int i = 0; i < MAXP; ++i)
        if (!pqueues[i].isEmpty()) return pqueues[i].dequeue();
    throw new EmptyQueueException();
}
```

"All operations are $O(1)$ since MAXP is fixed"
Assume each element knows its priority and lowest priority value is the "best"

Simulations and Queues

- Software simulations of real world systems
  - Queues of customers being served by agents
  - Network servers of communication packets
- Mathematical models predict frequencies and types of events
- Software simulation permits experimentation
  - Effect of adding agents, processors, different service algorithms
### Event Queues

- Queue of events to be processed
  - Time placed in queue
  - Time required to handle event
- Simulate clock, handle event at its time
  - Handling event may advance clock
  - Can measure wait time
  - Can determine if queue gets backed up
  - Analyze effect of adding more agents, processors

### Discrete Event Simulation

- Basic algorithm
  - Event has arrival time
  - Event has execution actions
    - Time required for execution
    - Side effects – may add more events to queue

```java
while ( !eventQueue.empty() &&
       eventQueue.peek().arrivalTime() <= clock) {
    Event currentEvent = eventQueue.dequeue();
    clock += currentEvent.execute();
}
```
Priority Queue Implementations

- Ordered by arrival time
  - Enqueue - insert at back of queue \( O(1) \)
  - Dequeue – search for lowest priority item \( O(n) \)
- Queue kept in priority order
  - Enqueue - insert in correct priority position \( O(n) \)
  - Dequeue – remove from front \( O(1) \)
- Either case requires \( O(n) \)
  - Would like to enqueue and dequeue in \( O(\log n) \)

Heaps

- ADT to facilitate ordered insertion, removal
  - Similar to binary search tree
  - Not sorted – weaker ordering
  - Always complete (balanced)
- Heap kept as array with invariants:
  - Size is number of elements currently in heap
  - Elements in locations 0...size-1
  - \( \text{heap}[0] \) is highest priority
  - \( \text{heap}[2*i+1] \) and \( \text{heap}[2*i+2] \) are the children of \( \text{heap}[i] \) if they exist, and each has lower priority than \( \text{heap}[i] \)
Heap Example

- size = 10, lesser value is higher priority

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>10</td>
<td>13</td>
<td>21</td>
<td>16</td>
<td>18</td>
<td>25</td>
<td>30</td>
<td>19</td>
<td>20</td>
<td>22</td>
<td>-</td>
<td>-</td>
<td>...</td>
</tr>
</tbody>
</table>

- This has the heap property

```
10
  
13
  
21
  16 18
  25 30
  19 20 22
  - - - - -
```

Heap Enqueue

- Insert at end, then sift up:
  - If parent is worse priority, swap with parent
  - Parent of heap[i] is heap[(i-1)/2]

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>10</td>
<td>13</td>
<td>21</td>
<td>16</td>
<td>18</td>
<td>25</td>
<td>30</td>
<td>19</td>
<td>20</td>
<td>22</td>
<td>18</td>
<td>22</td>
<td>19</td>
</tr>
</tbody>
</table>

- This still has the heap property (why?)
HeapPriorityQueue Implementation

- This implementation requires objects to implement the Java Comparable interface

```java
class PriorityQueue<E extends Comparable<E>> implements QueueI<E> {
    private int size;   // Number of elements in the heap
    private E[] heap;   // Heap is stored as an array
    private static final int HEAPALLOC=10; //increment for heap change

    // Constructor allocates initial array for heap
    public PriorityQueue() {
        size = 0;
        heap = allocHeap(HEAPALLOC);
    }

    public boolean isEmpty() { return size == 0; }
    public void clear() { size = 0; }
}
```

enqueue Implementation

- Put at end of array, sift up

```java
public void enqueue(E o) {
    // Grow the heap if needed
    if (heap.length == size) growHeap();
    // Put the new element at the end
    heap[size] = o;
    // Sift up to get current in right place
    for (int current = size++, parent = (current-1)/2; current > 0;
        current = parent, parent = (current-1)/2) {
        // If current has better priority than parent, swap
        if (heap[current].compareTo(heap[parent]) < 0)
            swap(parent, current);
        else   // Done - current is in the right place
            break;
    }
}
```
Heap Dequeue

- Remove element at head, replace with one from end, then sift it down to correct position:
  - If a child has better priority, swap with best child
  - Children of heap[i] are heap[2*i+1] and heap[2*i+2]

| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>12</td>
<td>18</td>
<td>21</td>
<td>16</td>
<td>18</td>
<td>25</td>
<td>30</td>
<td>19</td>
<td>20</td>
<td>22</td>
<td>18</td>
<td>-</td>
<td>...</td>
</tr>
</tbody>
</table>

- This still has the heap property (why?)

dqueue Implementation

```java
public E dequeue() throws EmptyQueueException {
    if (size == 0) throw new EmptyQueueException();
    E first = heap[0]; // Will return first, replace with last
    if (--size > 0) {
        heap[0] = heap[size]; // Sift down to get current in right place
        for (int current = 0, child = 2*current+1; current < size;
            current = child, child = 2*current+1) {
            // Find best priority child
            if ((child+1)<size && heap[child+1].compareTo(heap[child])<0)
                child = child + 1;
            // Swap if child is better priority
            if (child<size && heap[child].compareTo(heap[current])<=0)
                swap(child, current);
            else break; // Otherwise done - current is in right place
        }
    }
    return first;
}
```
Binary Heap Complexity

- Enqueue
  - Worst case \(O(\log n)\)
- Dequeue
  - Worst case \(O(\log n)\)
- Peek
  - \(O(1)\)