Queues

- Queue ADT
- Queue Implementation
- Priority Queues
Queue

- A restricted access container that stores a linear collection.
  - Very common for solving problems in computer science that require data to be processed in the order in which it was received.
  - Provides a **first-in first-out** (FIFO) protocol.

- New items are added at the **back** while existing items are removed from the **front** of the queue.
The Queue ADT

- A **queue** stores a linear collection of items with access limited to a first-in first-out order.
  - New items are added to the back.
  - Existing items are removed from the front.

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue()</td>
</tr>
<tr>
<td>isEmpty()</td>
</tr>
<tr>
<td>length()</td>
</tr>
<tr>
<td>enqueue(item)</td>
</tr>
<tr>
<td>dequeue()</td>
</tr>
</tbody>
</table>
Queue Example

- The following code creates the queue from earlier.

```python
Q = Queue()
Q.enqueue(28)
Q.enqueue(19)
Q.enqueue(45)
Q.enqueue(13)
Q.enqueue(7)
```
Queue Example

- We can remove items from the queue and add more items.

```python
x = Q.dequeue()
Q.enqueue(21)
Q.enqueue(74)
```
Queue Implementation

- Several common ways to implement a stack:
  - Python list
    - easiest to implement
  - Circular array
    - fast operations with a fixed size queue.
  - Linked list
    - reduces memory wastes by eliminating the extra capacity created with a vector.
Queue: Python List

- How is the data organized within the Python list?
  - Add new items to the end of the list.
  - Remove items from the front of the list.
# Implementation of the Queue ADT using a Python list.

class Queue:
    def __init__(self):
        self._qList = list()

    def isEmpty(self):
        return len(self) == 0

    def __len__(self):
        return len(self._qList)

    def enqueue(self, item):
        self._qList.append(item)

    def dequeue(self):
        assert not self.isEmpty(), "Cannot dequeue from an empty queue."
        return self._qList.pop(0)
Queue Analysis: Python List

<table>
<thead>
<tr>
<th>Queue Operation</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>q = Queue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>len(q)</td>
<td>O(1)</td>
</tr>
<tr>
<td>q.isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>q.enqueue(x)</td>
<td>O(n)</td>
</tr>
<tr>
<td>x = q.dequeue()</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Queue: Circular Array

- **circular array** – an array viewed as a circle instead of a line.

- Items can be added/removed without having to shift the remaining items in the process.
- Introduces the concept of a maximum-capacity queue that can become full.
Queue: Circular Array

- How should the data be organized within the array?
  - **count field** – number of items in the queue.
  - **front and back markers** – indicate the array elements containing the queue items.
Queue: Circular Array

- To enqueue an item:
  - new item is inserted at the position following back
  - back is advanced by one position
  - count is incremented by one.
- Suppose we enqueue 32:
Queue: Circular Array

- To dequeue an item:
  - the value in the `front` position is saved
  - `front` is advanced by one position.
  - `count` is decremented by one.
- Suppose we dequeue an item:
Queue: Circular Array

- Suppose we enqueue items 8 and 23:
Queue: Circular Array

- What happens if we enqueue 39?
  - Since we are using a circular array, the same steps are followed.
  - But since back is at the end of the array, it wraps around to the front.
Queue: Circular Array

```python
class Queue:
    def __init__(self, maxSize):
        self._count = 0
        self._front = 0
        self._back = maxSize - 1
        self._qArray = Array(maxSize)

    def isEmpty(self):
        return self._count == 0

    # A new operation specifically for the circular array.
    def isFull(self):
        return self._count == len(self._qArray)

    def __len__(self):
        return self._count

    # ...
Queue: Circular Array

```python
class Queue:
    # ...  
    def enqueue(self, item):
        assert not self.isFull(), "Cannot enqueue to a full queue."
        maxSize = len(self._qArray)
        self._back = (self._back + 1) % maxSize
        self._qArray[self._back] = item
        self._count += 1

    def dequeue(self):
        assert not self.isEmpty(), "Cannot dequeue from an empty queue."
        item = self._qArray[self._front]
        maxSize = len(self._qArray)
        self._front = (self._front + 1) % maxSize
        self._count -= 1
        return item
```
Queue Analysis: Circular Array

<table>
<thead>
<tr>
<th>Queue Operation</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>q = Queue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>len(q)</td>
<td>O(1)</td>
</tr>
<tr>
<td>q.isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>q.enqueue(x)</td>
<td>O(1)</td>
</tr>
<tr>
<td>x = q.dequeue()</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Queue: Linked List

- How should the data be organized?
  - Use both head and tail references.
  - Let the head of the list represent the front of the queue and the tail the back.
Queue: Linked List

# Implementation of the Queue ADT using a linked list.

class Queue :
    def __init__( self ) :
        self._qhead = None
        self._qtail = None
        self._count = 0

    def isEmpty( self ) :
        return self._qhead is None

    def __len__( self ) :
        return self._count

# ...

# Private storage class for creating the linked list nodes.
class _QueueNode:
    def __init__( self, item ) :
        self.item = item
        self.next = None
Queue: Linked List

class Queue:
    # ...
    def enqueue(self, item):
        node = _QueueNode(item)
        if self.isEmpty() :
            self._qhead = node
        else:
            self._qtail.next = node
        self._qtail = node
        self._count += 1

    def dequeue(self):
        assert not self.isEmpty(), "Cannot dequeue from an empty queue."
        node = self._qhead
        if self._qhead is self._qtail:
            self._qtail = None
        self._qhead = self._qhead.next
        self._count -= 1
        return node.item
Queue Analysis: Linked List

<table>
<thead>
<tr>
<th>Queue Operation</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>q = Queue()</td>
<td>O(1)</td>
</tr>
<tr>
<td>len(q)</td>
<td>O(1)</td>
</tr>
<tr>
<td>q.isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>q.enqueue(x)</td>
<td>O(1)</td>
</tr>
<tr>
<td>x = q.dequeue()</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Priority Queues

- Some applications require the use of a queue in which items are assigned a priority.
  - higher priority items are dequeued first.
  - items with equal priority still follow FIFO.
- Two types:
  - **bounded** – limited range of priorities.
  - **unbounded** – unlimited range.
The Priority Queue ADT

- A *priority queue* is a queue in which each item is assigned a priority and items with a higher priority are removed before those with lower priority.
  - Integer values are used for the priorities.
  - **Smaller integers have a higher priority.**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PriorityQueue()</td>
<td>Constructor</td>
</tr>
<tr>
<td>BpriorityQueue( numLevels )</td>
<td>Constructor</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>Checks if the queue is empty</td>
</tr>
<tr>
<td>length()</td>
<td>Returns the length of the queue</td>
</tr>
<tr>
<td>enqueue( item, priority )</td>
<td>Adds an item with a specified priority</td>
</tr>
<tr>
<td>dequeue()</td>
<td>Removes and returns the item with the highest priority</td>
</tr>
</tbody>
</table>
Priority Queue Example

Consider the following code segment:

```python
Q = BpriorityQueue( 6 )
Q.enqueue( "purple", 5 )
Q.enqueue( "black", 1 )
Q.enqueue( "orange", 3 )
Q.enqueue( "white", 0 )
Q.enqueue( "green", 1 )
Q.enqueue( "yellow", 5 )
```
Priority Queue Implementation

- How should the ADT be implemented. We must consider:
  - A priority must be associated with each item in the queue.
  - The next item dequeued is the item with the highest priority.
  - If multiple items have the same priority, those must be dequeued in a FIFO order.
Unbounded Priority Queue

- We explore two approaches:
  - Python list
  - Linked list
Unbounded Priority Q: Python List

- We can use a Python list to implement the unbounded Priority Queue ADT.
  - How do we associate a priority with each item?

  ```python
  class PriorityQEntry:
  def __init__(self, item, priority):
      self.item = item
      self.priority = priority
  ```

- How should the entries be organized?
  - append new items to the end, or
  - keep the items in sorted order based on priority.
Unbounded Priority Q: Python List

- Sample instance for the earlier priority queue.
# Implementation of the unbounded Priority Queue ADT using a Python
# list with new items appended to the end.

class PriorityQueue:
    def __init__(self):
        self._qList = list()

    def isEmpty(self):
        return len(self) == 0

    def __len__(self):
        return len(self._qList)

# ...
class PriorityQueue:
    # ...
    def enqueue(self, item, priority):
        entry = _PriorityQEntry(item, priority)
        self._qList.append(entry)

    def dequeue(self):
        assert not self.isEmpty(), "Cannot dequeue from an empty queue."

        # Find the entry with the highest priority.
        highest = self._qList[0].priority
        high_index = 0
        for i in range(self.len()):
            if self._qList[i].priority < highest:
                highest = self._qList[i].priority
                high_index = i

        # Remove the entry with the highest priority and return the item.
        entry = self._qList.pop(high_index)
        return entry.item
Unbounded Priority Q: Linked List

- We can use a singly linked list:
  - Head and tail references.
  - Append new entries to the end.
Priority Queue Analysis

- The worst case analysis for the two implementations.

<table>
<thead>
<tr>
<th>Queue Operation</th>
<th>Python List</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>q = PriorityQueue()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>len(q)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>q.isEmpty()</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>q.enqueue(x)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>x = q.dequeue()</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Bounded Priority Queue

- We can use the Python list or linked list to implement the bounded Priority Queue.
  - Both require linear time to dequeue an item.
  - We can improve this time using an array of queues.
Bounded Priority Q Implementation

```python
from array import Array
from llistqueue import Queue

class BPriorityQueue:
    def __init__(self, numLevels):
        self._qSize = 0
        self._qLevels = Array(numLevels)
        for i in range(numLevels):
            self._qLevels[i] = Queue()

    def isEmpty(self):
        return len(self) == 0

    def __len__(self):
        return self._qSize

# ...
```
class BPriorityQueue:
    # ...  
    def enqueue(self, item, priority):
        assert priority >= 0 and priority < len(self._qLevels), "Invalid priority level."
        self._qLevels[priority].enqueue(item)
        self._qSize += 1

def dequeue(self):
    # Make sure the queue is not empty.
    assert not self.isEmpty(), "Cannot dequeue from an empty queue."

    # Find the first non-empty queue.
    i = 0
    p = len(self._qLevels)
    while i < p and self._qLevels[i].isEmpty():
        i += 1
    self._qSize -= 1
    # We know the queue is not empty, so dequeue from the ith queue.
    return self._qLevels[i].dequeue()