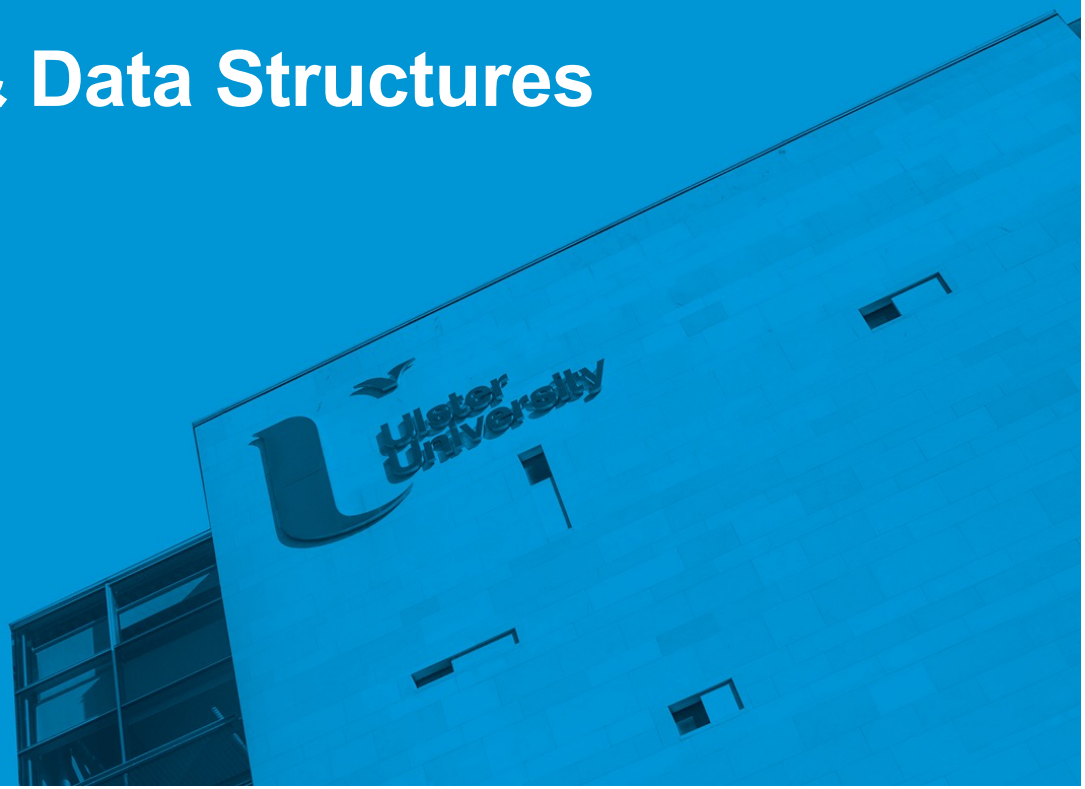
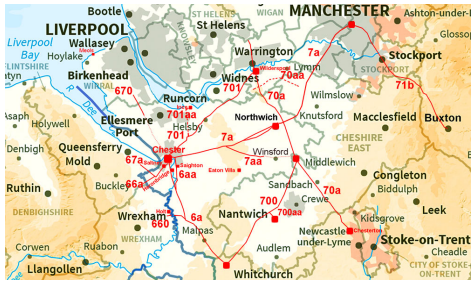
Ulster University

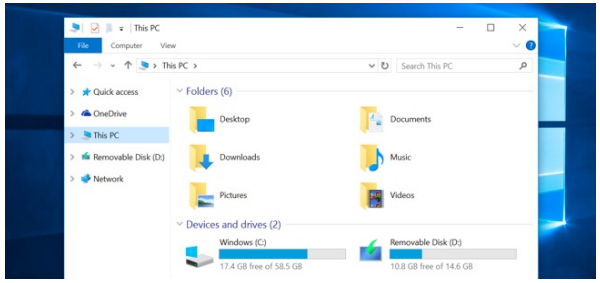
COM498 Algorithms & Data Structures

3.4 Queues



Recall Examples of Data Organisation





- Queue – First in, first out – another common data organisation technique in everyday life

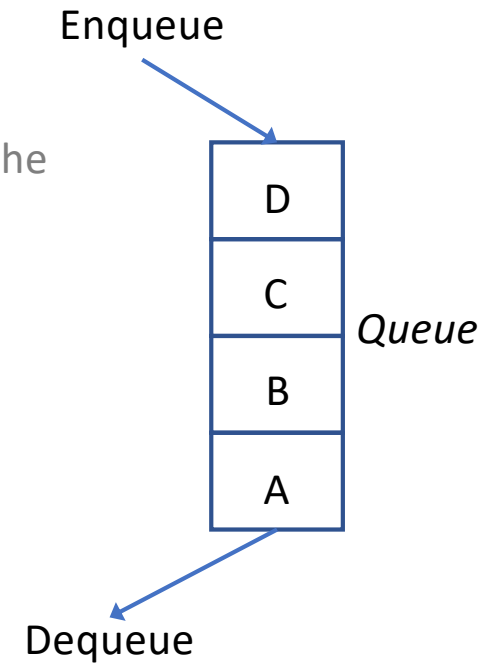
Queues

- A queue is another name for a waiting line
- Used within operating systems and to simulate real-world events
- Come into play whenever processes or events must wait
- Entries organized first-in, first-out (chronologically)

- Sometimes more flexibility is needed:
- A double-ended queue permits operations on both oldest and newest entries (known as a deque)
- When the importance of an object depends on criteria other than its arrival time, you can assign it a priority (known as a priority queue)

Queue Operations

- Like a stack the queue ADT organizes entries in the order in which they were added
- The behaviour of a queue is also known as: **FIFO** (First In, First Out)



- In the queue all additions are to the **back** of the queue (the entry at the back is the latest item added to the queue)
- The item that was added earliest is at the **front** of the queue
- The operation that adds an entry to the queue is traditionally called **enqueue**
- The operation that removes an entry from the queue is traditionally called **dequeue**

Queue Operations

- Like the stack, the queue also restricts access to its entries (can only look at or the earliest entry)
- In addition to enqueue and dequeue, the operation to retrieve the top entry without removing it is called **getFront**

- Typically you cannot search a queue for a specific entry
- The only way to look at an entry not at the front of the queue is to repeatedly remove items from the stack until the desired item reaches the front

Queue Operations

Java Interface for the Queue ADT

```
public interface QueueInterface<T> {
    public void enqueue(T newEntry);
    public T dequeue();
    public T getFront();
    public boolean isEmpty();
    public void clear();
}
```

- `enqueue(T newEntry)` → Add new entry to back of the queue
- `dequeue()` → Remove entry from front of the queue
- `getFront()` → Return entry from front of the queue
- `isEmpty()` → Check for no entries in queue
- `clear()` → Remove all entries from the queue

Scenario

- In your LinkedList project, create the file *QueueInterface.java* and implement the interface class `QueueInterface`

Queue Operations

- Assume the class `Queue` implements `QueueInterface`

```
Queue<String> myQueue = new Queue<String>();

myQueue.enqueue("Jim");
myQueue.enqueue("Jess");
myQueue.enqueue("Jill");
myQueue.enqueue("Jane");
myQueue.enqueue("Joe");
```

(a) – (e)

```
front = myQueue.getFront();
front = myQueue.dequeue();
```

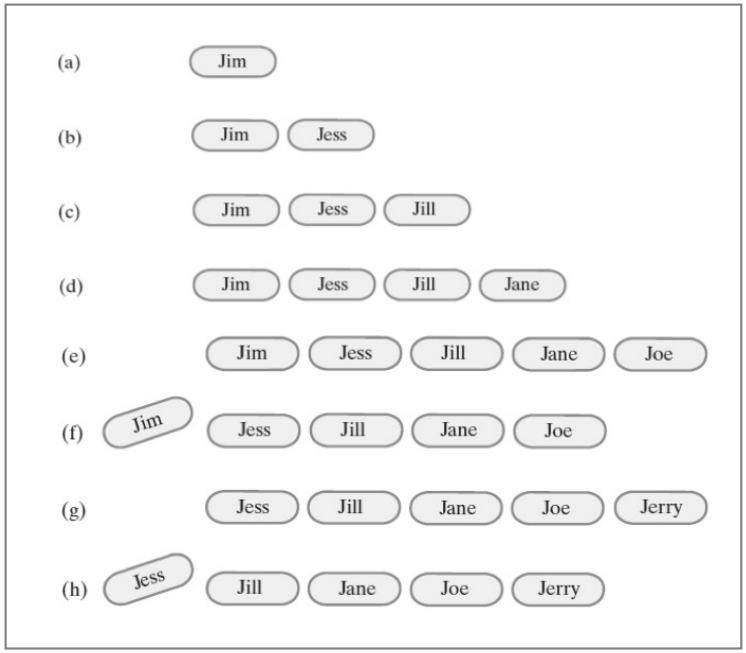
(f)

```
myQueue.enqueue("Jerry");
```

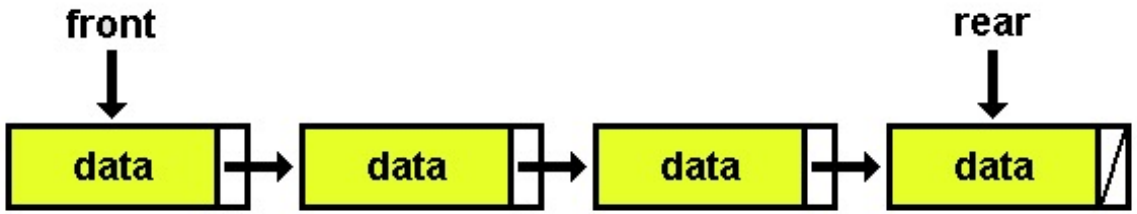
(g)

```
front = myQueue.getFront();
front = myQueue.dequeue();
```

(h)



Linked Implementation of Queue



- Need to maintain two pointers, `front` and rear `to` point to either end of the list
- An is added to the queue (`enqueue`) by inserting it at the `rear`. The `next` field of the element pointed at by rear will now point to the new element and `rear` is updated to point to the new element
- An entry is removed (`dequeue`) by removing the element pointed at `by first`, which is updated to the `next` field of the current (old) first element

Linked Chain Implementation

Ulster University

```
public class Queue<T> implements QueueInterface<T> {
   private MyNode<T> front;
   private MyNode<T> rear;

   public Queue() {
      front = null;
      rear = null;
   }
   // methods enqueue(), dequeue(), getFront(), isEmpty(),
   // clear() as defined in the interface
}
```

Scenario

- Create the file *Queue.java* and provide the code for the class `Queue` as an implementation of `QueueInterface` that uses a singly-linked list as the data organisation technique
  - Define the `front` instance variable that points to the first element in the chain
  - Define the `rear` instance variable that points to the last element in the chain
  - Define the constructor that creates a new queue object
  - Provide the implementation of all methods specified in the `QueueInterface` class
- Check your implementation by providing a `main()` method that:
  - Creates a queue of `String` objects and enqueues 3 values into the queue
  - Attempts to return and then dequeue 4 values from the queue
  - Adds another 3 values to the queue and checks for an empty queue
  - Clears the queue and repeats the check for an empty queue

Java Class Library: The Interface `Queue`

- Found in package **`java.util`**

java.util

Interface Queue<E>

Type Parameters:

`E` - the type of elements held in this collection

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Subinterfaces:

BlockingDeque<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>

Method Summary

Methods

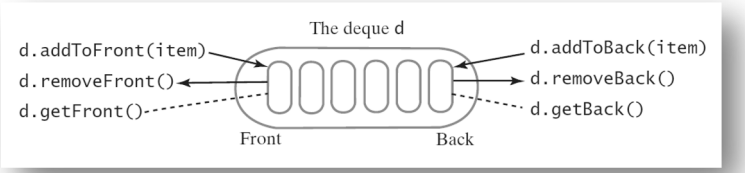
Modifier and Type	Method and Description
boolean	**`add`**`(E e)` Inserts the specified element into this queue if it is possible to do so imn throwing an `IllegalStateException` if no space is currently availab
`E`	**`element`**`()` Retrieves, but does not remove, the head of this queue.
boolean	**`offer`**`(E e)` Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
`E`	**`peek`**`()` Retrieves, but does not remove, the head of this queue, or returns `null` if this queue is empty.
`E`	**`poll`**`()` Retrieves and removes the head of this queue, or returns `null` if this queue is empty.
`E`	**`remove`**`()` Retrieves and removes the head of this queue.

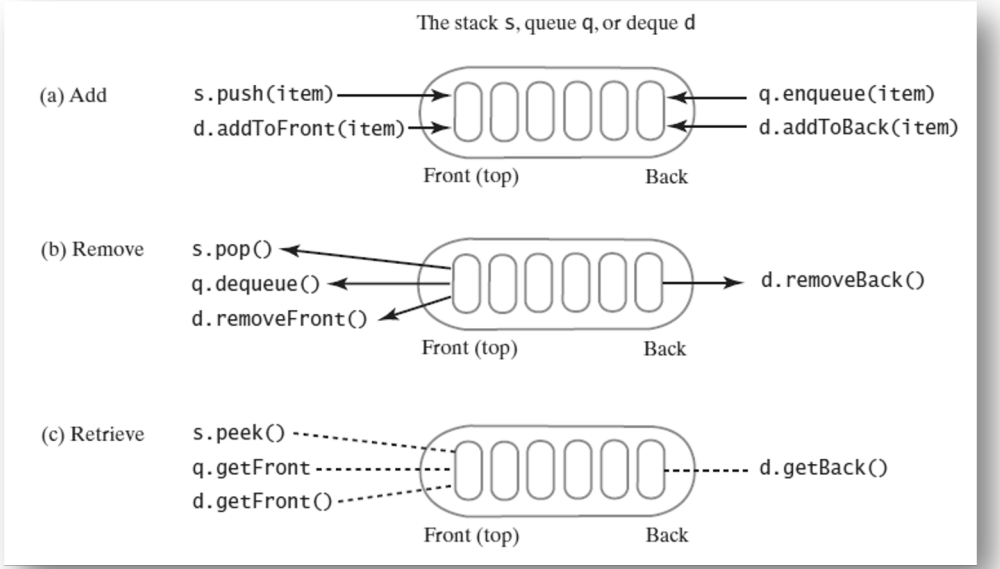
Methods inherited from interface java.util.Collection

`addAll, clear, contains, containsAll, equals, hashCode, isEmpty, iterator, remove, removeAll, retainAll, size, toArray, toArray`

The Deque ADT

- A double-ended queue (allows add, remove, retrieve at front and back of queue)
- Known as a **deque** (pronounced *"deck"*)
- Has both queue-like operations and stack-like operations:





(a) Add, (b) Remove, (c) Retrieve operations for a stack *s*, queue *q* and deque *d*

Java Interface for the Deque ADT

Ulster University

```
public interface DequeInterface<T> {
    public void addToFront(T newEntry);
    public void addToBack(T newEntry);
    public T removeFront();
    public T removeBack();
    public T getFront();
    public T getBack();
    public boolean isEmpty();
    public void clear();
}
```

- Add new entry to front of deque
- Add new entry to back of deque
- Remove entry from front of deque
- Remove entry from back of deque
- Return entry from front of deque
- Return entry from back of deque
- Check for deque empty
- Remove all entries from deque

Java Class Library: The Interface `Deque`

- Found in package **`java.util`**

java.util

Interface Deque<E>

Type Parameters:

`E` - the type of elements held in this collection

All Superinterfaces:

Collection<E>, Iterable<E>, Queue<E>

All Known Subinterfaces:

BlockingDeque<E>

All Known Implementing Classes:

ArrayDeque, ConcurrentLinkedDeque, LinkedBlockingDeque, LinkedList

Method Summary

Methods

Modifier and Type	Method and Description
boolean	**`add(E e)`** Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) if it is pos violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is c
void	**`addFirst(E e)`** Inserts the specified element at the front of this deque if it is possible to do so immediately without violating capacity restr
void	**`addLast(E e)`** Inserts the specified element at the end of this deque if it is possible to do so immediately without violating capacity restri
boolean	**`contains(Object o)`** Returns `true` if this deque contains the specified element.
`Iterator<E>`	**`descendingIterator()`** Returns an iterator over the elements in this deque in reverse sequential order.
`E`	**`element()`** Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque).
`E`	**`getFirst()`** Retrieves, but does not remove, the first element of this deque.
`E`	**`getLast()`** Retrieves, but does not remove, the last element of this deque.
`Iterator<E>`	**`iterator()`** Returns an iterator over the elements in this deque in proper sequence.
boolean	**`offer(E e)`** Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and `false` if no space is currently available.
boolean	**`offerFirst(E e)`** Inserts the specified element at the front of this deque unless it would violate capacity restrictions.
boolean	**`offerLast(E e)`** Inserts the specified element at the end of this deque unless it would violate capacity restrictions.

Sources: https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

The Priority Queue ADT

- Consider how a hospital buildings assign a priority to each patient that *overrides* the time at which the patient arrived
- The priority queue ADT organises objects according to their priorities
- Definition of "*priority*" depends on the nature of the items in the queue (a priority of 1 can be the highest priority or it can be the lowest priority!)

Java Interface for the Priority Queue

```
public interface PriorityQueueInterface<T> {
    public void enqueue(T newEntry);
    public T dequeue();
    public T getFront();
    public boolean isEmpty();
    public void clear();
}
```

Add new entry to queue in priority position

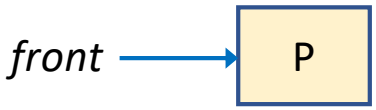
As for queue specification

Adding to Priority Queue

- Assume that the priority level can be derived from the data value
  - Note the rear pointer is no longer needed (only had a role in adding at rear)
  - E.G. alphabetical order assigns priority and elements are added by…

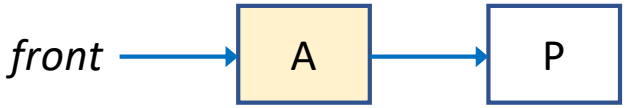
3 cases to cater for

```
pQueue.add("P");
```



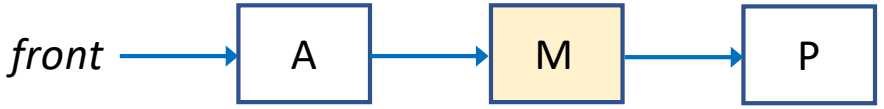
add to empty pQueue

```
pQueue.add("A");
```



add at front

```
pQueue.add("M");
```



add after existing element

```
pQueue.add("X");
```

Adding to Priority Queue

Ulster University

```
Algorithm add(newEntry)
// Add a new entry into a linked pQueue in its priority position

create new node
if queue is empty set front pointer to point to the new node          // add to empty
else if priority of node at front <  priority of new node             // add at front
        set newNode next pointer to front
        set front to newNode
     else set currentNode pointer to front                            // add after existing
          while currentNode.getNext() is not null and
                priority of currentNode.getNext() >  priority of newNode
                set currentNode to curentNode.getNext()
          set newNode next pointer to currentNode next pointer
          set currentNode next pointer to newNode
    end if
end if
```

Java Sidenote - `Comparable` Interface

- To find the place to insert an item in a priority queue, we must be able to compare one object to another.

- Use of the Generic Type **`T`** means that any object type can be used as the data payload for our nodes. Although most of our examples will compare Integers and Strings, the Java implementations given accept any `Comparable` object (i.e. objects that implement the Comparable interface)

- The `Comparable` interface contains one method `compareTo()` which is designed to return an integer that specifies the relationship between two objects:

```
obj1.compareTo(obj2)
```

Java Sidenote - `Comparable` Interface

- Allows us to implement `compareTo()` so that `obj1.compareTo(obj2)`
  - Defines an order between objects
  - Returns a negative integer (usually `-1`), zero (`0`), or a positive integer (usually `1`) if `obj1` is less than, equal to or greater than `obj2`, respectively

less than (return -1) equal to (return 0) greater than (return +1)

- Most built-in objects (i.e. `Integer`, `String`, etc.) already have an implementation for `compareTo()` so all we need do is state in our class header that we wish to use it

Java Sidenote – `Comparable` Interface

- To use `compareTo()` we need to change the header for both our Interface class and our Implementation class, so (for example…)

PQueue.java

```
public class PQueue<T extends Comparable <T>> implements
PQueueInterface<T> {

   // we can now use the compareTo() method on objects of
   // the generic class T within our code in this class
...
}
```

Java Class Library: The Class `PriorityQueue`

- Implements the interface `Queue`

java.util

Class PriorityQueue<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractQueue<E>
      java.util.PriorityQueue<E>

Type Parameters:

`E` - the type of elements held in this collection

All Implemented Interfaces:

Serializable, Iterable<E>, Collection<E>, Queue<E>

Methods

Modifier and Type	Method and Description
boolean	**add**(E e) Inserts the specified element into this priority queue.
void	**clear**() Removes all of the elements from this priority queue.
Comparator<? super E>	**comparator**() Returns the comparator used to order the elements in this queue, or null if this queue is sorted according
boolean	**contains**(Object o) Returns true if this queue contains the specified element.
Iterator<E>	**iterator**() Returns an iterator over the elements in this queue.
boolean	**offer**(E e) Inserts the specified element into this priority queue.
E	**peek**() Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
E	**poll**() Retrieves and removes the head of this queue, or returns null if this queue is empty.
boolean	**remove**(Object o) Removes a single instance of the specified element from this queue, if it is present.
int	**size**() Returns the number of elements in this collection.
Object[]	**toArray**() Returns an array containing all of the elements in this queue.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array.

Sources: https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

Challenge

- In your **LinkedList** project, create the file *PQueueInterface.java* and implement the class `PQueueInterface` as an interface for a Priority Queue class.
- Add the file *PQueue.java* and provide the class `PQueue` that implements the Priority Queue interface as a linked list structure
- Add a `main()` method to your `PQueue` class and test your structure by creating a new instance of `PQueue` where the data to be stored in each element is an integer and where larger integers have the highest priority
    - Generate and print 10 random integers in the range 1-1000 and add each as a new entry in the priority queue
    - Remove and print all elements from the queue, showing that the list of numbers are retrieved in priority order (largest values first)