2. Stacks and Queues :: Queues :: Introduction

A queue is a linear (or sequential) data structure similar to a stack, but unlike a stack—where we add and remove elements from only one end—with a queue, we add elements to one end and remove elements from the other. This makes a queue an example of a **first-in-first-out** (FIFO) data structure, i.e., the first element added to the queue is the first element removed from the queue.

Since queue operations are only performed on both ends of the linear sequence, we will refer to the elements as the **front** and **rear** elements.

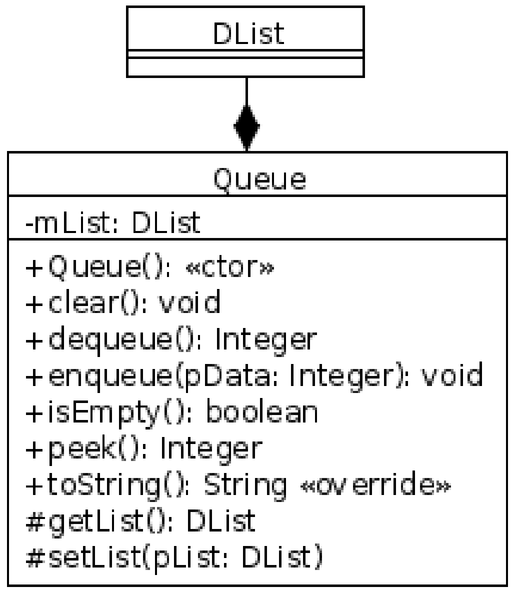
The standard queue operations are:

1. Enqueue which adds a new element to the rear of the queue.
2. Peek which gets an element from the front of the queue without removing it.
3. Dequeue which removes the element from the front of the queue.

Queues occur commonly in programming, just as they are common in real life, e.g., a line of customers at a movie theater forms a queue.

2. Stacks and Queues :: Queues :: UML Class Diagram

A queue can also be easily implemented by using a linked list to store the elements of the queue. Our *Queue* class diagram is:



A *Queue* object contains one instance variable which is a *DList* (of *Integer*). The *clear*() method is called to remove all of the elements from the queue; after calling *clear*() the queue is empty and *isEmpty*() would return true.

getList() and *setList*() are protected accessor and mutator methods for the private *mList* instance variable. They are not intended to be publicly called on a *Queue* but are made protected so they may be called by subclasses of *Queue*.

2. Stacks and Queues :: Queues :: Implementation

```
//**************************************************************************************
// CLASS: Queue (Queue.java)
//**************************************************************************************
// Implements a queue data structure using a DList to store the elements.
public class Queue {
    private DList mList;
    // Creates a new empty Queue by creating a new empty DList.
    public Queue() {
        setList(new DList());
    }
    // Removes all of the elements from this Queue. After clear() returns this Queue
    // is empty.
    public void clear() {
        getList().clear();
    }
    // Removes and returns the element that is at the front of this Queue.
    public Integer dequeue() {
        Integer front = getList().remove(0);
        return front;
    }
```

2. Stacks and Queues :: Queues :: Implementation (continued)

```
// Adds pData to the rear of this Queue.
public void enqueue(Integer pData) {
    getList().append(pData);
}

// Accessor method for mList.
protected DList getList() {
    return mList;
}

// Returns true if this Queue is empty.
public boolean isEmpty() {
    return getList().isEmpty();
}

// Returns the front element of this Queue without removing it.
public Integer peek() {
    return getList().get(0);
}

// Mutator method for mList.
protected void setList(DList pList) {
    mList = pList;
}
```

2. Stacks and Queues :: Queues :: Implementation (continued)

```
    // Overrides toString() inherited from Object. Returns a String representation of
    // the elements of this Queue by calling the DList.toString() method.
    @Override
    public String toString() {
        return getList().toString();
    }
}
```

2. Stacks and Queues :: Queues :: Time Complexity

How efficient are the peek, enqueue, and dequeue operations? From *Linked Lists* : *Section 11*:

1. **$add()$, $get()$, and $remove()$ at the head and tail of a doubly linked list is $O(1)$.**
2. $add()$, $get()$, and $remove()$ on an interior element—that is, randomly accessing elements—is $O(n)$.
3. Iterating over the elements of a linked list in sequence is an efficient operation as we can access the next element in the sequence in $O(1)$ time.

Therefore, a linked list is a good data structure for storing a linear sequence of elements where:

1. The number of elements in the list will grow and shrink, i.e., it is a dynamic data structure.
2. **Elements are primarily accessed at the beginning and ends of the list.**
3. Elements are accessed in a linear sequence.