Queues and List-Based ADT Implementations

Wellesley College CS230
Lecture 09
Monday, February 26
Handout #18

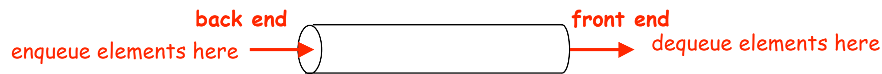
Problem Set: PS3 due Wednesday, March 7

Overview of Today's Lecture

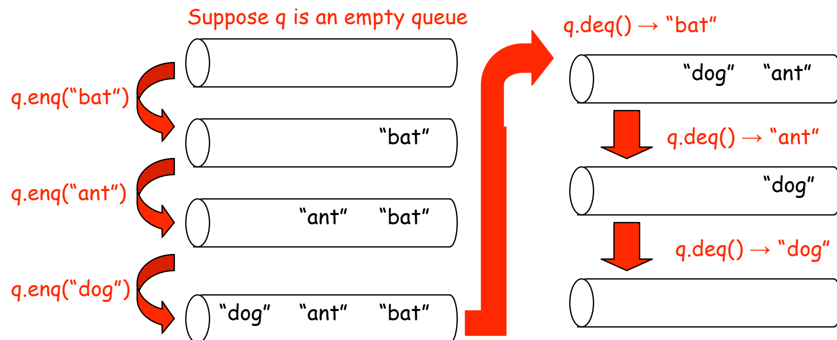
1. Queues = first-in first-out (FIFO) collections.
2. Vector implementations of stacks and queues. Efficient vector implementations of queues are complex.
3. An elegant implementation of queues based on linked lists.
4. Implementations of a mutable linked list abstraction.

A Queue is a FIFO Collection

A queue is like a tube. Different ends are used to enqueue (insert) and dequeue (remove) elements.



The enq() method enqueues an element and deq() dequeues an element. front() returns the element at the front of the queue without removing it



Queue Interface

```
import java.util.*; // imports Iterator and Iterable

/** Interface to mutable queues with elements of type T.
    A queue is a first-in-first-out collection.
    WARNING: the CS230 Queue interface is different than
    the java.util.Queue interface. */
public interface Queue<T> extends Iterable<T> {
  public void enq(T elt); // Add elt to back of queue.
  public T deq(); // Remove and return element from front of queue.
                  // Throw RuntimeExecption if queue is empty.
  public T front(); // Return front element of queue without removing it.
                    // Throw RuntimeExecption if queue is empty.
  public boolean isEmpty(); // Return true if this queue is empty; else false
  public int size(); // Return the number of elements in this queue
  public void clear(); // Remove all elements from this queue
  public Queue<T> copy(); // Return a shallow copy of this queue
  public Iterator<T> iterator(); // Return an iterator that yields
                                 // this queue's elements from front to back.
}
```

Vector Implementations of Stacks

Draw pictures from class here:

Bad Vector Implementations of Queues

Draw pictures from class here:

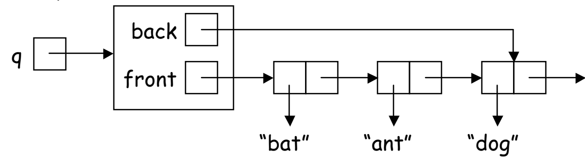
A Good Array Implementation of Queues

Draw pictures from class here:

09-7

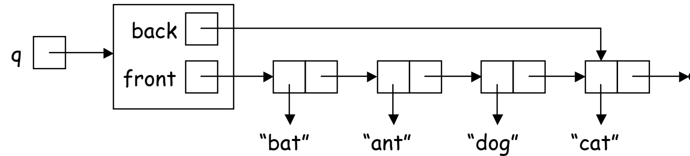
A Good Linked List Implementation of Queues

QueueTwoEndedLinkedList



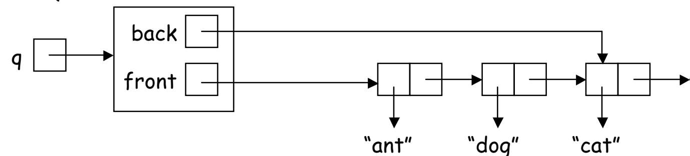
q.enq("cat")

QueueTwoEndedLinkedList



q.deq() → "bat"

QueueTwoEndedLinkedList

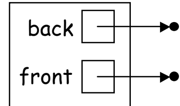


09-8

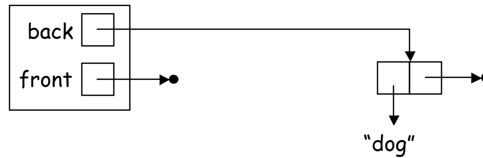
What is an Empty Queue?

We define a queue to be empty if **front** contains the empty list;
what **back** contains is irrelevant. E.g., both of the following are empty:

QueueTwoEndedLinkedList



QueueTwoEndedLinkedList



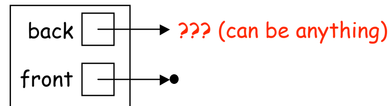
This simplifies the implementation of clear(), which can just
set **front** to the empty list.

09-9

Equeuing into an Empty Queue

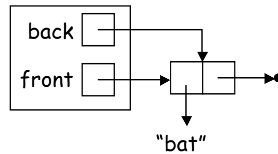
Enqueuing an element into an empty queue is a special case:

QueueTwoEndedLinkedList



q.enq("bat");

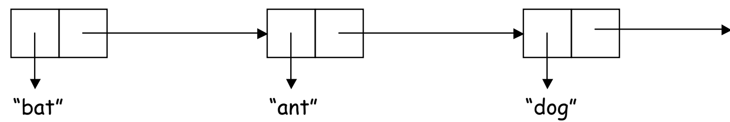
QueueTwoEndedLinkedList



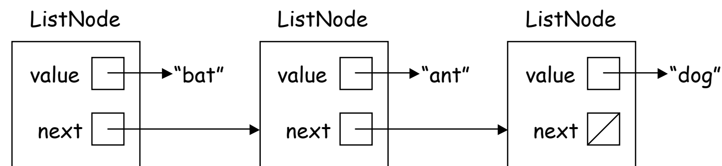
09-10

Representing Linked Lists as "Raw" Nodes

Abstract View:



Concrete View:



09-11

A ListNode<T> Class

```
class ListNode<T> {

  // Simplify things by using public instance variables.
  public T value; // Like head in CS111 list
  public ListNode<T> next; // Like tail in CS111 list

  public ListNode (T value, ListNode<T> next) {
    this.value = value;
    this.next = next;
  }

}
```

09-12

Linked-List Queue Code: Skeleton

```
public class QueueTwoEndedLinkedList<T> implements Queue<T> {

  // Instance Variables:
  private ListNode<T> front, back; // Initially contain null by default

  // Constructor Methods:
  public QueueTwoEndedLinkedList() {
    // front is null by default, and back is irrelevant at beginning, so do nothing.
  }

  // Instance Methods
  public boolean isEmpty () {
    return front == null; // By design, queue is empty if front is empty
  }

 public void clear() {
    front = null; // by design, don't need to modify back.
  }
  // Remaining instance methods go here
}
```

09-13

Linked-List Queue Code: enq()

```
public void enq (T elt) {
  if (isEmpty()) {
    // Enqueuing onto the empty list is a special case:
    front = new ListNode<T>(elt, null);
    back = front; // initializes back for empty front
  } else {
    back.next = new ListNode<T>(elt, null);
    back = back.next;
  }
}
```

09-14

Linked-List Queue Code: deq() and front()

```
public T deq () {
  if (isEmpty()) {
    throw new RuntimeException("Attempt to deq() an empty queue");
  } else {
    T result = front.value;
  }
}

public T front () {
  if (isEmpty()) {
    throw new RuntimeException("Attempt to find front() of an empty queue");
  } else {
    return front.value;
  }
}
```

09-15

Linked-List Queue Code: size() and copy()

```
public int size() { // Calculate the length of the linked list in front.
  int len = 0;
  for (ListNode<T> current = front; current != null; current = current.next) {
    len++;
  }
  return len;
}

public Queue<T> copy() {
  QueueTwoEndedLinkedList<T> qCopy = new QueueTwoEndedLinkedList<T>();
  if (front != null) {
    qCopy.front = new ListNode<T>(front.value, null);
    ListNode<T> prevCopy = qCopy.front;
    for (ListNode<T> current = front.next; current != null; current = current.next) {
      prevCopy.next = new ListNode<T>(current.value, null);
      prevCopy = prevCopy.next;
    }
  }
  return qCopy;
}
```

09-16

Linked-List Queue Code: toString()

```
public String toString() {
  StringBuffer buff = new StringBuffer();
  buff.append("QueueTwoEndedLinkedList[");
  if (front != null) {
    buff.append(front.value); // Treat first element specially – no comma in front
    for (ListNode<T> current = front.next; current != null; current = current.next) {
      buff.append("," + current.value);
    }
  }
  buff.append("]");
  return buff.toString();
}
```

09-17

Linked-List Queue Code: iterator()

```
public Iterator<T> iterator() {
  return new QueueIterator<T>(this);
}
```

where QueueIterator is defined as:

```
public class QueueIterator<T> implements Iterator<T> {

  private Queue<T> q; // holds a copy of the queue being iterated over

  public QueueIterator (Queue<T> q) { this.q = q.copy(); } // copy the given queue

  public boolean hasNext() { return ! q.isEmpty(); }

  public T next() { return q.deq();  } // Remove and return the next elt from copy

  public void remove() { // Don't handle this method
    throw new UnsupportedOperationException(
      "QueueIterator<T> does not support the remove() operation.");
  }
}
```

09-18

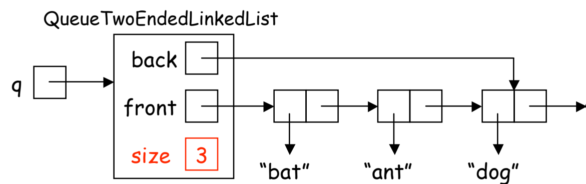
Linked-List Queue Code: main()

```
public static void main (String[] args) {
  // QueueTester.test performs simple tests on any queue implementation:
  QueueTester.test(new QueueTwoEndedLinkedList<String>());
}
```

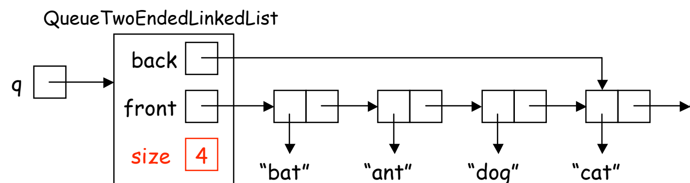
```
q = QueueTwoEndedLinkedList[]; q.size() = 0; q.front() = Attempt to find front() of an empty queue
q.enq(B); q = QueueTwoEndedLinkedList[B]; q.size() = 1; q.front() = B
q.enq(C); q = QueueTwoEndedLinkedList[B,C]; q.size() = 2; q.front() = B
q.enq(B); q = QueueTwoEndedLinkedList[B,C,B]; q.size() = 3; q.front() = B
q.enq(A); q = QueueTwoEndedLinkedList[B,C,B,A]; q.size() = 4; q.front() = B
q.deq() = B; q = QueueTwoEndedLinkedList[C,B,A]; q.size() = 3; q.front() = C
q.deq() = C; q = QueueTwoEndedLinkedList[B,A]; q.size() = 2; q.front() = B
q.deq() = B; q = QueueTwoEndedLinkedList[A]; q.size() = 1; q.front() = A
q.enq(D); q = QueueTwoEndedLinkedList[A,D]; q.size() = 2; q.front() = A
q.enq(A); q = QueueTwoEndedLinkedList[A,D,A]; q.size() = 3; q.front() = A
q.enq(C); q = QueueTwoEndedLinkedList[A,D,A,C]; q.size() = 4; q.front() = A
q2 = (Queue<String>) q.copy(); q2 = QueueTwoEndedLinkedList[A,D,A,C]; q2.size() = 4; q2.front() = A
q.clear(); q = QueueTwoEndedLinkedList[]; q.size() = 0; q.front() = Attempt to find front() ...
q2 = QueueTwoEndedLinkedList[A,D,A,C]; q2.size() = 4; q2.front() = A
Iterating the elements of q2:
0:A
1:D
2:A
3:C
```

An Improved Queue: Caching the Size

Can change size() from a linear-time to a constant time operation by storing it in an instance variable. For example:



q.enq("cat")



Code Modifications for Queue with Cached Sizes

```
public class QueueTwoEndedLinkedListBetter<T> implements Queue<T> {

  // Instance Variables:
  private ListNode<T> front, back; // Initially contain null by default
  private int size = 0;

  public int size() { return size; } // Constant time, not linear time in # of elts

  public void clear() { front = null; size = 0; }

  public void enq (T elt) { … usual enq() code goes here …  size++; }

  public void deq (T elt) { … put size--; before return result … }

  public Queue<T> copy()  { … put qCopy.size = size;  before return qCopy … }

  // All other methods are unchanged.
}
```

MList<T> : A Mutable Linked-List Abstraction

```
// Static methods from CS111 lists:
public static <T> MList<T> empty();
public static <T> boolean isEmpty(MList<T> L) ;
public static <T> MList<T> prepend(T elt, MList<T> L) ;
public static <T> T head(MList<T> L) ;
public static <T> MList<T> tail(MList<T> L)

// New methods for changing head and tail
public static <T> void setHead (MList<T> L, T newHead);
public static <T> void setTail (MList<T> L, MList<T> newTail)

// Other static methods:
public static <T> String toString (MList<T> L);
public static <T> boolean equals (MList<T> L1, MList<T> L2);
public static <T> int length (MList<T> L)
public static <T> T nth (int n, MList<T> L);
public static <T> MList<T> lastNode (MList<T> L);
public static <T> MList<T> append(MList<T> L1, MList<T> L2)
public static <T> MList<T> appendD (MList<T> L1, MList<T> L2);
public static <T> MList<T> postpend(MList<T> L, T elt)
public static <T> MList<T> postpendD (MList<T> L, T elt);
public static <T> MList<T> reverse (MList<T> L) <T>);
public static <T> MList<T> copy (MList<T> L)
```

Two-Ended Linked-List Queue with Mlist<T>

```
public class QueueTwoEndedMList<T> implements Queue<T> {

 private static MList ML; // Local abbreviation for MList
 private MList<T> front, back;

 // Constructor Methods:
 public QueueTwoEndedMList() { front = ML.<T>empty(); } // Always invoke empty with type!

 // Instance Methods:
 public void enq (T elt) {
  if (isEmpty()) {
   front = ML.prepend(elt, ML.<T>empty()); // Enqueuing onto the empty list is a special case:
   back = front; // initializes back for empty front
  } else {
   ML.setTail(back, ML.prepend(elt, ML.<T>empty()));
   back = ML.tail(back);
  }
 }

…

}
```

Some Simplifications with Mlist<T>

```
public int size() {
 return ML.length(front); // Use MList.length() method
}

public Queue<T> copy() {
 QueueTwoEndedMList<T> qc = new QueueTwoEndedMList<T>();
 qc.front = ML.copy(front); // Use MList.copy() method
 if (! ML.isEmpty(front)) {
  qc.back = ML.lastNode(front);
 }
 return qc;
}

public Iterator<T> iterator() {
 return front.iterator(); use MList's built-in iterator:
}
```