The List ADT
Objectives

- Define a list abstract data type
- Examine different classes of lists
- Examine various list implementations
- Compare list implementations
Lists

• A list is a linear collection, like a stack and queue, but more flexible: adding and removing elements from a list does not have to happen at one end or the other

• We will examine three types of list collections:
  • ordered lists
  • unordered lists
  • indexed lists
Ordered Lists

- **Ordered list**: Its elements are ordered by some inherent characteristic of the elements
- **Examples**:
  - Names in alphabetical order
  - Numeric scores in ascending order
- So, the elements themselves determine where they are stored in the list
Conceptual View of an Ordered List

New values must be inserted so that the ordering of the list is maintained.
Unordered Lists

- **Unordered list**: the order of the elements in the list is *not* based on a characteristic of the elements, but is determined by the *user* of the list

- A new element can be put
  - on the front of the list,
  - or on the rear of the list,
  - or after a particular element already in the list

- **Examples**: shopping list, to-do list, …
Conceptual View of an Unordered List

New values can be inserted anywhere in the list
Indexed Lists

- **Indexed list**: elements are referenced by their *numeric position* in the list, called its *index*
- It is the *position* in the list that is important, and the user can determine the order in which the items go in the list
- Every time the list changes, the *position* (index) of an element may change
- **Example**: current first-place holder in the bobsled race
Conceptual View of an Indexed List

New values can be inserted at any position in the list
List Operations

- Operations common to *all* list types include:
  - *Adding/removing* elements
  - *Checking the status* of the list (*isEmpty*, *size*)
  - *Iterating through* the elements in the list (more on this later!)
- The key differences between the list types involve the way elements are *added*
# The Common Operations on a List

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeFirst</td>
<td>Removes the first element from the list</td>
</tr>
<tr>
<td>removeLast</td>
<td>Removes the last element from the list</td>
</tr>
<tr>
<td>remove</td>
<td>Removes a particular element from the list</td>
</tr>
<tr>
<td>first</td>
<td>Gets the element at the front of the list</td>
</tr>
<tr>
<td>last</td>
<td>Gets the element at the rear of the list</td>
</tr>
<tr>
<td>contains</td>
<td>Determines if a particular element is in the list</td>
</tr>
<tr>
<td>isEmpty</td>
<td>Determines whether the list is empty</td>
</tr>
<tr>
<td>size</td>
<td>Determines the number of elements in the list</td>
</tr>
<tr>
<td>toString</td>
<td>Returns a string representation of the list</td>
</tr>
</tbody>
</table>
Operation Particular to an Ordered List

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Adds an element to the list (in the correct place)</td>
</tr>
</tbody>
</table>
# Operations Particular to an Unordered List

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addToFront</td>
<td>Adds an element to the front of the list</td>
</tr>
<tr>
<td>addToRear</td>
<td>Adds an element to the rear of the list</td>
</tr>
<tr>
<td>addAfter</td>
<td>Adds an element after a particular element already in the list</td>
</tr>
</tbody>
</table>
# Operations Particular to an Indexed List

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Adds an element at a particular index in the list</td>
</tr>
<tr>
<td>set</td>
<td>Sets the element at a particular index in the list overwriting any element that was there</td>
</tr>
<tr>
<td>get</td>
<td>Returns a reference to the element at the specified index</td>
</tr>
<tr>
<td>indexOf</td>
<td>Returns the index of the specified element</td>
</tr>
<tr>
<td>remove</td>
<td>Removes and returns the element at a particular index</td>
</tr>
</tbody>
</table>
List Operations

- We use Java interfaces to formally define the lists ADTs
- Note that interfaces can be defined via *inheritance* (derived from other interfaces)
  - Define the common list operations in one interface
    - See *ListADT.java*
    - Derive the thee others from it
      - see *OrderedListADT.java*
      - see *UnorderedListADT.java*
      - see *IndexedListADT.java*
ListADT Interface

```java
import java.util.Iterator;
public interface ListADT<T> {

    // Removes and returns the first element from this list
    public T removeFirst();

    // Removes and returns the last element from this list
    public T removeLast();

    // Removes and returns the specified element from this list
    public T remove(T element);

    // Returns a reference to the first element on this list
    public T first();

    // Returns a reference to the last element on this list
    public T last();

    // cont’d..
```
// ..cont’d
// Returns true if this list contains the specified target element
public boolean contains (T target);
// Returns true if this list contains no elements
public boolean isEmpty(
);
// Returns the number of elements in this list
public int size(
);
// Returns a string representation of this list
public String toString(
);}
public interface OrderedListADT<T> extends ListADT<T>
{
    // Adds the specified element to this list at the proper location
    public void add (T element);
}
public interface UnorderedListADT<T> extends ListADT<T> {
    // Adds the specified element to the front of this list
    public void addToFront (T element);

    // Adds the specified element to the rear of this list
    public void addToRear (T element);

    // Adds the specified element after the specified target
    public void addAfter (T element, T target);
}
public interface IndexedListADT<T> extends ListADT<T> {
    // Inserts the specified element at the specified index
    public void add(int index, T element);
    // Sets the element at the specified index
    public void set(int index, T element);
    // Returns a reference to the element at the specified index
    public T get(int index);
    // Returns the index of the specified element
    public int indexOf(T element);
    // Removes and returns the element at the specified index
    public T remove(int index);
}
Discussion

• Note that the `remove` method in the `IndexedList ADT` is overloaded
  • Why? Because there is a `remove` method in the parent `List ADT`
    • This is *not* overriding, because the parameters are different
List Implementation using Arrays

- Container is an array
- Fix one end of the list at index 0 and shift \textit{as needed} when an element is added or removed
- Is a shift needed when an element is added
  - at the front?
  - somewhere in the middle?
  - at the end?
- Is a shift needed when an element is removed
  - from the front?
  - from somewhere in the middle?
  - from the end?
An Array Implementation of a List

An array-based list `ls` with 4 elements

```
list
4
rear
```

```
0  1  2  3  4  5
?  ?  ?  •••  ?
```

```
0  1  2  3
?  ?  ?
```
public T remove (T element) throws ElementNotFoundException
{
    T result;
    int index = find (element);  // uses helper method find
    if (index == NOT_FOUND)
        throw new ElementNotFoundException("list");
    result = list[index];
    rear--;
    // shift the appropriate elements
    for (int scan=index; scan < rear; scan++)
        list[scan] = list[scan+1];
    list[rear] = null;
    return result;
}
// Returns the array index of the specified element,  
// or the constant NOT_FOUND if it is not found.
private int find (T target) 
{  
  int scan = 0, result = NOT_FOUND;
  boolean found = false;
  if (! isEmpty( ))
    while (! found && scan < rear)
      if (target.equals(list[scan]))
        found = true;
      else
        scan++;
  if (found)
    result = scan;
  return result;
}
// Returns true if this list contains the specified element.

public boolean contains (T target)
{
    return (find(target) != NOT_FOUND);
    // uses helper method find
}
The **Comparable** Interface

- For an ordered list, the *actual* class for the generic type $T$ *must* have a way of comparing elements so that they can be ordered
  - So, it must implement the **Comparable** interface, *i.e.* it must define a method called `compareTo`
- But, the *compiler* does not know whether or not the class that we use to fill in the generic type $T$ will have a `compareTo` method
The Comparable Interface

- So, to make the compiler happy:
  - Declare a variable that is of type `Comparable<T>`
  - Convert the variable of type `T` to the variable of type `Comparable<T>`

```java
Comparable<T> temp = (Comparable<T>)element;
```

- Note that an object of a class that implements `Comparable` can be referenced by a variable of type `Comparable<T>`
// Adds the specified Comparable element to the list, keeping the elements in sorted order.

public void add (T element) {
    if (size( ) == list.length)
        expandCapacity( );
    Comparable<T> temp = (Comparable<T>)element;
    int scan = 0;
    while (scan < rear && temp.compareTo(list[scan]) > 0)
        scan++;
    for (int scan2=rear; scan2 > scan; scan2--)
        list[scan2] = list[scan2-1]
    list[scan] = element;
    rear++;
List Implementation Using Arrays, Method 2: *Circular Arrays*

- Recall circular array implementation of queues

- *Exercise*: implement list operations using a circular array implementation
List Implementation Using Links

- We can implement a *list collection* with a *linked list* as the container
  - Implementation uses techniques similar to ones we've used for stacks and queues
- We will first examine the *remove* operation for a singly-linked list implementation
- Then we’ll look at the *remove* operation for a doubly-linked list, for comparison
public T remove (T targetElement) throws ElementNotFoundException
{
    if (isEmpty( ))
        throw new ElementNotFoundException ("List");
    boolean found = false;
    LinearNode<T> previous = null
    LinearNode<T> current = front;
    // cont’d.
while (current != null && !found)
    if (targetElement.equals (current.getElement ( )))
        found = true;
    else {
        previous = current;
        current = current.getNext( );
    }
if (!found)  throw new ElementNotFoundException ("No data");

if (size( ) == 1)
    front = rear = null;
else
    if (current.equals (front))
        front = current.getNext( );
    else
        // cont’d
if (current.equals (rear)) {
    rear = previous;
    rear.setNext(null);
}
else
    previous.setNext(current.getNext());

    count--;
    return current.getElement();
}
Doubly Linked Lists

• A **doubly linked list** has **two** references in each node:
  • One to the **next** element in the list
  • One to the **previous** element

• This makes moving back and forth in a list easier, and eliminates the need for a **previous** reference in particular algorithms

• **Disadvantage?** a bit more overhead when managing the list
Implementation of a Doubly-Linked List

A doubly-linked list $dl$ with 4 elements
• See *DoubleClickNode.java*

• We can then implement the **ListADT** using a doubly linked list as the container

• Following our usual convention, this would be called *DoublyLinkedList.java*
public DoubleNode<T> find (T element) {
    DoubleNode<T> current = front;
    while (current != null && !element.equals(current.getElement()))
        current = current.getNext();
    return current;
}
public T remove (T element) throws ElementNotFoundException {
    DoubleNode<T> node = find (element);
    if (node == null) throw new ElementNotFoundException ("No element");
    if (node == front) {
        front = next;
        if (front != null) front.setPrevious(null);
    }
    else (node.getPrevious()).setNext(node.getNext());
    if (node == rear) {
        rear = node.getPrevious();
        if (rear != null) rear.setNext(null);
    }
    else (node.getNext()).setPrevious(node.getPrevious());
    count--;
    return node.getElement();
}
// Adds element to the list, keeping the list sorted.
public void add (T element) {
    Comparable<T> temp = (Comparable<T>)element;
    DoubleNode<T> newNode = new DoubleNode<T>(element);
    if (front == null) {
        front = newNode;
        rear = newNode;
    } else {
        DoubleNode<T> current = front;
        while (current != null && temp.compareTo(current.getElement()) > 0)
            current = current.getNext();
        if (current == null) {
            // Add newNode at the end of the list
            rear.setNext(newNode);
            newNode.setPrev(rear);
            rear = newNode;
        }
    }
}
else {  // newNode is not added to the end
    newNode.setNext(current);
    newNode.setPrev(current.getPrev());
    current.setPrev(newNode);
    if (newNode.getPrev() != null)
        newNode.getPrev().setNext(newNode);
    else front = newNode;
}
++count;
Analysis of List Implementations

• In both array and linked implementations, many operations are similar in efficiency.

• Most are $O(1)$, except when shifting or searching need to occur, in which case they are order $O(n)$.
  • Exercise: determine the time complexity of each operation.

• In particular situations, the frequency of the need for particular operations may guide the use of one approach over another.