Linked Lists

Weiss, section 6.5 (pages 248–258)
List Interface

- List is an ordered collection of elements that have positions. Lists may contain duplicate elements.
- Java provides two implementations for List: ArrayList and LinkedList.

- ArrayList offers constant-time positional access using an index:
  ```java
  List<T> list = /* create and initialize a list */
  for ( int i = 0; i < list.size(); i++ ) { list.get(i); }
  ```

- Elements are added to the end of ArrayList using add method.

- Use LinkedList if you frequently add elements to the beginning of the List or delete elements while iterating through the List.
- Linked list stores elements in non-contiguous locations in memory.
- Because of this, an element is accessed using a reference to a node.
**LinkedList**

```java
class ListNode<T> {
    T data;
    ListNode<T> next;
}
```

`LinkedList` is a collection of nodes that are linked together.

```java
LinkedList<String> list = new LinkedList<String>();
list.add("Steve");
list.add("Carl");
list.addFirst("Michael");
list.addLast("Rick");
String str = list.remove(); // removes and returns the first element (Michael).
```
ListIterator

ListIterator traverses a list in both directions. hasPrevious and previous are analogues to hasNext and next.

```java
ListIterator<String> iterator = list.listIterator(list.size());
while(iterator.hasPrevious()) {
    System.out.println(iterator.previous());
}
```

• ListIterator has methods for adding and removing elements.
• add places a new element before the current position of cursor.
  iterator.next(); // returns Steve
  iterator.next(); // returns Carl
  iterator.add(“Ryan”); // adds Ryan before Carl
• remove deletes an element that is returned by next.
  iterator.next(); // returns Rick
  iterator.remove(); // removes Rick
ListIterator

• The *remove* method needs to be called only after a call to the *next* method or *previous* method.

• The zero-parameter ListIterator constructor places the cursor just before the *first* element of a list.
## Running Time Complexity

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arraylist</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>add/remove at end</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>¹add/remove at front</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>²add/remove at another place</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
<tr>
<td>³get/set</td>
<td>O(1)</td>
<td>O(N)</td>
</tr>
<tr>
<td>contains</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

¹ArrayList needs to move data one index higher or lower.
²In doubly linked list, removing an element takes O(1) time.
³ArrayList and LinkedList add elements at a specified positions.
³set replaces an element at a specified position.
public static int sum (List<Integer> lst ) {
    int total = 0;
    for ( int i = 0; i < lst.size( ); i++ )
        total += lst.get( i );
    return total;
}

The running time is $O(n)$ for ArrayList and $O(n^2)$ for LinkedList.

- The **enhanced** for loop makes running time $O(n)$ for any type of list.
  - Compiler translates the for loop into an Iterator object.

```java
public static int sum (List<Integer> lst ) {
    int total = 0;
    for ( int val : lst )
        total += val;
    return total;
}
```