Java **LinkedList** class; Iterators

- Introduction to linked lists
  - comparison with arrays
- Useful LinkedList methods
- Traversing a LinkedList: iterators
- ListIterator methods
- Using an iterator to…
  - examine elements
  - modify elements
  - insert elements
  - remove elements
Announcements

• Lab 7 has been published; includes advanced preparation. (uses LinkedList class)
• PA2 due tomorrow.
• Office hours changes this week: on piazza
• Lecture this Thur (10/5) was pretaped. Can watch on your computer.
• Policy on regrades: you have one week from when the work is returned to initiate a regrade.
• Check your scores were entered correctly on d2l.
Review of big-O

• Do Sequence class from last lecture.
Review

- Want to store a collection of things (elements).
- All elements are the same type
- Want random access to elements
- Can use an array (or ArrayList):

```
0  1  2  3  4  5
10 20 30 40      
```

...
Introduction

• Alternate: linked list
  – Only use as much space as you need at a time.
  – Can insert and delete from middle without shifting values left or right by one.
  – However no random access based on location. E.g., get element at position $k$ is not constant time:
    – has to traverse to element $k$
Linked list implementations

• Will discuss code for writing our own linked lists later this semester (using C++)
• Java (and C++) has a LinkedList class:
  `LinkedList<ElementType>`
• has some of the same methods as `ArrayList`
• but, WARNING, some of them run slower. E.g.,
  ```
  list.get(i)
  list.set(i, newVal)
  ```
Using ArrayList methods with LinkedLists

```java
void printList(LinkedList<Integer> list) {
    for (int i = 0; i < list.size(); i++) {
        System.out.println(list.get(i));
    }
}
```

• What is the big-O time to run this code?
Using ArrayList methods with LinkedLists

for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}

• A bad way to traverse a linked list.

• Generally avoid using the methods that take an index: e.g., add(i, object), remove(i), set(i, object)
Putting elements in a LinkedList

- Create an empty list:
  ```java
  LinkedList<Integer> list = new LinkedList<Integer>();
  ```
- Put some stuff in the list:
  ```java
  list.add(10);
  list.add(20);
  list.add(30);
  list.add(40);
  ```
- Adding to the end (or beginning) is efficient: O(1)
- Internally uses a "tail" pointer (or equivalent)
Other LinkedList methods

• Operations that access the beginning or end are efficient:

// suppose list contains :
    [Anne, Sally, George, Carol]

list.addFirst("Gaga");

list.getFirst() // returns Gaga

list.getLast() // returns Carol

list.removeFirst(); // removes Gaga

list.removeLast(); // removes Carol
So, how do we traverse a LinkedList?

• Recall: for loop with get(i) is a bad idea.
• Have to use a ListIterator object
• Associate it with a particular list
• Abstracts the idea of some position in the list
• We can also use it to add or remove from the middle.
ListIterator

- Iterator interface is similar to Scanner:
  
  ```java
  next()
  hasNext()
  ```

- Guard calls to `next()` with a call to `hasNext()` so you don't go past the end of the list

- To get an iterator positioned at the start of `list`:
  
  ```java
  ListIterator<String> iter = list.listIterator();
  ```
ListIterator

• Iterator points between two elements.
• 5 possible positions for iterator on the following list:

   [Anne, Sally, George, Carol]
Traversing with a `ListIterator`

```java
// print out all the elements of the list:
ListIterator<String> iter = list.listIterator();
while (iter.hasNext()) {
    String word = iter.next();
    System.out.println(word);
}
```

Suppose `list` contains:

```java
[Anne, Sally, George, Carol]
```

`next()` returns the element after `iter` position and advances `iter` beyond that element.
next() changes state of iterator

- Want to print out all values $\geq 60$
- Suppose list contains:
  \[\{33, 94, 56, 59\}\]
- What is the output of the following code:

```java
ListIterator<Integer> iter = list.listIterator();
while (iter.hasNext()) {
    if (iter.next() $\geq$ 60) {
        System.out.println(iter.next());
    }
}
```
Let’s write a non-buggy version…

ListIterator<Integer> iter = list.listIterator();
modifying elements using iterator

Suppose list contains:

\[33, 94, 86, 59\]

- Adds 10 points to everyone's score?

```java
ListIterator<Integer> iter = list.listIterator();
while (iter.hasNext()) {
    int current = iter.next();
    current += 10;
}
```

- How to modify the values in the list?
modifying elements using iterator (cont.)

- How to modify the values actually in the list?
  
  ```java
  iter.set(newValue)
  ```
  replaces the element last returned by `next()`

- Suppose list contains:
  
  `[33, 94, 86, 59]`

- Add 10 points to everyone's score:
  
  ```java
  ListIterator<Integer> iter = list.listIterator();
  while (iter.hasNext()) {
      int current = iter.next();
      iter.set(current+10);
  }
  ```
Lists containing mutable objects

- We've modified the object reference (only way to change an immutable object), using `set`
- Could modify contents of a mutable object instead by using a mutator.
- Translate all Points in a list (mutable objects):
  ```java
  ListIterator<Point> iter = list.listIterator();
  while (iter.hasNext()) {
      Point current = iter.next();
      current.translate(10, 20);
  }
  ```
ArrayLists containing mutable objects

- (Review) Similarly with ArrayList:
- Translate all Points in an ArrayList:

```java
ArrayList<Point> pointList = ...;
for (int i = 0; i < pointList.size(); i++) {
    Point current = pointList.get(i);
    current.translate(10, 20);
}
```
Inserting/removing from the middle of the list

- Review: more efficient than with array, don't have to shift a bunch of elements.
- Still would have to traverse to get to the correct place to insert/remove.
- Use the iterator `add` / `remove` methods
ListIterator **add** method

• Recall **iter** is positioned between two values.

  \[\text{[Anne, Carol, George, Sally]}\]

• **iter.add(newValue)**

  inserts **newValue** at that position

• after operation, iterator is positioned after **newValue**

• Suppose **newValue = "Tom"**

  \[\text{[Anne, Carol, Tom, George, Sally]}\]
Example of using add

Duplicate all the values in a list:

list before = [Anne, Carol, George]
list after =
    [Anne, Anne, Carol, Carol, George, George]

public static void dupe(LinkedList<String> list) {
ListIterator `remove` method

- Recall `iter` is positioned between two values.
  
  `[Anne, Carol, George, Sally]`

- `iter.remove()` removes the element that was returned by the last call to `next()`

- after operation, iterator is positioned where the old value used to be

  `[Anne, George, Sally]`
Example of using `remove`

Remove all values below a threshold (e.g., 60)

list before = [93, 86, 57, 59, 100]
list after = [93, 86, 100]

```java
void removeLT(LinkedList<Integer> list,
              int threshold) {
```
More on LinkedLists

• There are more LinkedList and ListIterator methods that may be useful for lab 8.
  – E.g., you can also iterate backwards over a list.

• Remember: avoid using the LinkedList methods that take an index as a param in a loop.
  – Note: if index is 0 or size()−1 it’s ok, because optimizes those cases with head and tail pointer (O(1))

• Use online documentation for more information.