Outline

- **linked list removal**
- **recursive linked list code**
- **different kinds of linked lists:**
  - singly-linked lists
  - circular linked lists
  - doubly-linked lists
- **looping over collections**
- **iterators**

Removing the last node in a linked list

- removing the first node in a linked list is easy:

```
head = head.next;
```

- removing the last node requires access to the node before the last (if any). Simplified code:

```
LinkedNode<E> nextToLast = head;
while (nextToLast.next.next != null) {
  nextToLast = nextToLast.next;
}
nextToLast.next = null;
```

- This takes linear time, even if we keep a tail pointer

Recursion and Linked Lists

- **a linked node is recursively defined: each linked node may refer to another linked node**
  - there are many recursive data structures, a linked list is one of the simplest
- **the recursive case is that the linked node's `next` field refers to another linked node**
- **the base case is that the linked node's `next` field is `null`**

Recursively computing the length of a linked list

```
private int computeLength(LinkedNode<E> node) {
  if (node == null) {
    return 0;
  }
  return 1 + computeLength(node.next);
}
```

- the length of a list identified by a null reference is 0
- the length of a list is one more than the length of the list identified by node.next

Recursively adding to a linked list

```
private LinkedNode<E> addAtIndex(LinkedNode<E> node, E value, int index) {
  if (node == null) {
    assert(index == 0);
    tail = new LinkedNode<E>(value);
    return tail;
  }
  if (index == 0) {
    return new LinkedNode<E>(value, node);
  }
  node.next = addAtIndex(node.next, value, index - 1);
  return node;
}
```

- in-class exercise (in groups): call this from the method `public void add(int index, E item) {`

Recursively adding to a linked list: duplicating the list

```
private LinkedNode<E> addAtIndex(LinkedNode<E> node, E value, int index) {
  if (node == null) {
    assert(index == 0);
    tail = new LinkedNode<E>(value);
    return tail;
  }
  if (index == 0) {
    return new LinkedNode<E>(value, node);
  }
  return new LinkedNode<E>(node.item,
                         addAtIndex(node.next, value, index - 1));
}
```

Circular Linked Lists

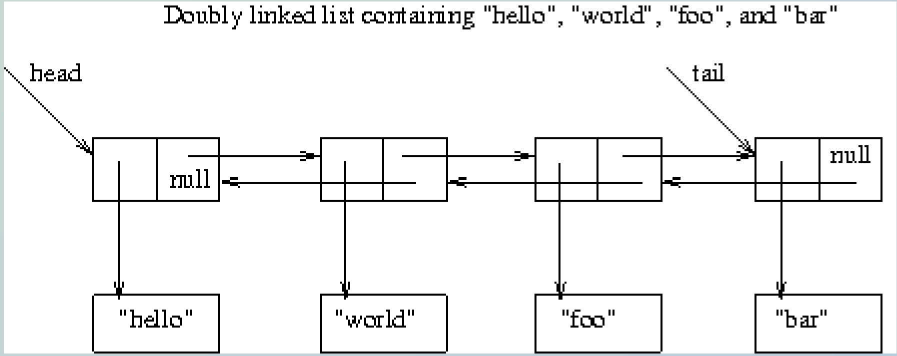
- **by convention, the `next` field of the last node in a linked list has the value `null`**
- **a different convention suggests storing the value of `head` in this field**
- **then a single class variable `tail` is sufficient:**
  - it is not necessary to have the `head` field, since `head` is the same as `tail.next`
  - computing `tail.next` is a constant-time operation
- **a complete traversal of a circular list can begin from any node, not necessarily the head or tail node**
  - but it is easy to (mistakenly) have list traversal loop forever

Circular Linked List exercise

- in small groups
- write a toString method that, given a `tail` reference, returns a string containing string representations of all the elements of the circular linked list
- in any order

Doubly-linked lists

- the linked lists so far have the limitation that it is only possible for code to follow references in one direction in the list, that is, forward
- node removal requires a reference to the node before the node to be removed
- if each node also keeps a reference to the node before it, both these problems can be solved



x

Nodes for Doubly-Linked Lists

```
private class DLinkedNode<E> {
  private E item;                   // one element
  private DLinkedNode<E> prev;      // two references, one to the node before
  private DLinkedNode<E> next;      // and one to the node after

  private DLinkedNode(E value) {
    item = value;
    next = null;
    prev = null;
  }

  private DLinkedNode(E value, DLinkedNode<E> prev, DLinkedNode<E> next) {
    item = value;
    this.next = next;
    this.prev = prev;
  }
}
```

Doubly-Linked List add

- adding after a given node (`node`) means updating the previous and next node's next and prev references:

```
DLinkedNode followingNode = node.next;
node.next = new DLinkedNode (value, node, node.next);
followingNode.prev = node.next;
```

- in-class exercise (alone or with a friend or two): draw the doubly-linked list after each of the lines of the above code
- the above code assumes that there is both a previous and next node
- if not, the code needs special cases
- a circular list, if coded correctly, needs fewer special cases

Doubly-Linked List remove

- removing a given node (`node`) means updating the node's predecessor's next field, and the node's successor's prev field:

```
  node.prev.next = node.next;
  node.next.prev = node.prev;
```

- here are some special cases for a linked-list that is not circular:

```
if ((node == head) && (head.next == null)) {
  head = null; tail = null;
} else {
   if (node.prev != null)
      node.prev.next = node.next;
   if (node.next != null)
      node.next.prev = node.prev;
}
```

Looping over the elements of a collection

- sometimes we want do something with all of the elements of a collection
- for example, we might want to print the values
- or we might want to add all the values in a collection of numbers
- we can do a loop with `get`:

```
for (int i = 0; i < List.size(); i++) {
  E element = List.get(i);
  ... // do something with element
}
```

- if `get` takes more than constant time, this is very inefficient: the outer loop is repeated List.size times, so if `get` takes time linear in the list size, the entire loop takes time $\texttt{List.size}^2$ or $O(n^2)$.

Efficiently looping over the elements of a collection

- for a linked list, `get` takes linear time
- but accessing a list element *if we have a reference to the node containing the element* only takes constant time
- however, it is not safe to let the user program directly have access to this reference
- instead, the reference is encapsulated in an object called an <u>iterator</u>, which provides a small set of operations

using Java iterators

```
List<E> list = …
for (E element: list) {
    …
}
```

- Java internally re-writes the above loop as:

```
Iterator<E> it = list.iterator();
while (it.hasNext()) {
    E element = it.next();
    ...
}
```

Yes, but what is an iterator?

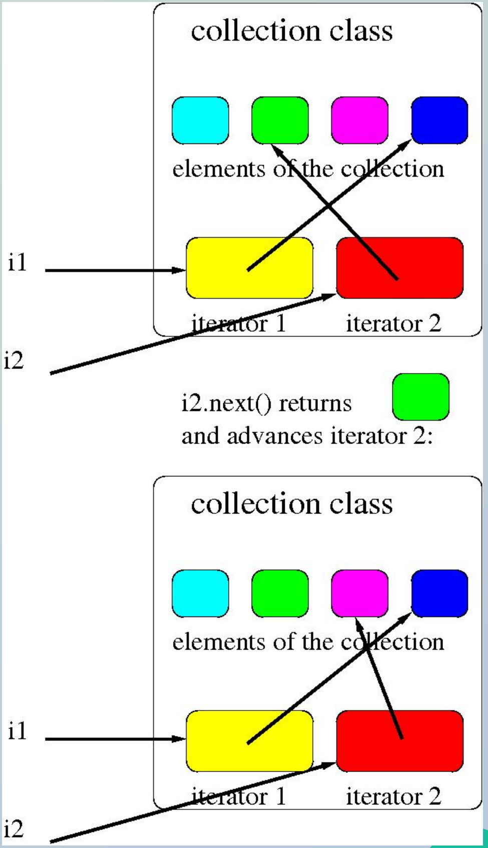
- an iterator is an object that supports the two methods `hasNext()` and `next()`
- `next()` provides access to the elements of a collection
- for example, an iterator for a linked list class would internally have a reference to the node containing the next object, here called `node`:

```
public class LinkedListIterator<E> . . . {
    LinkedNode<E> node;
```

- the iterator is a different object than the collection

example

- **i1 and i2 are iterators for the same collection**
- **advancing i2 does not affect i1**
- **next() returns the next element <u>and</u> advances the iterator**



Iterator methods

- **a Java iterator only provides two or three operations:**
- **`E next()`, which returns the next element, and also advances the reference**
  - advancing the iterator is a side effect of calling `next`
- **`boolean hasNext()`, which returns whether there is at least one more element**
- **`void remove()`, which removes the last element returned by `next()`**
  - the `remove` method is optional
- **using `remove` may invalidate any other existing (concurrent) iterators**

Automatic use of iterators

- instead of having to use the while loop to use an iterator, the `for` loop has been specialized to call the iterator

```
LinkedList<Integer> values = ...
int sum = 0;
for (Integer value: values) {
  sum = sum + value;
}
```

- Java creates and calls the iterator, but the iterator itself is not visible in the code
- the same code can loop over arrays

Java foreach

- this automated (and invisible) use of iterators with for loops is called the Java enhanced for statement or for each statement
- the foreach statement works on any expression that has a value that satisfies the Iterable interface
- The Iterable interface requires a method called `iterator`:

```
interface Iterable<E> {
  Iterator<E> iterator();  // return a new iterator
}
```

Iterator implementation

- a Java iterator may or may not be internal to the collection class
- every Java iterator must have sequential access to the elements of the collection
- every Java iterator must have at least one variable to keep track of where it is in the traversal, that is, which elements have not yet been returned
- See LinkedListIterator.java for a very simple iterator on linked lists.
- in-class exercise (everyone together): design the code for the `iterator()` method of the LinkedList class

ListIterator

- the Java `Iterator` interface is very general and reasonably powerful
- however, sometimes it is useful to be able to move backwards and forwards, and add or replace as well as remove elements
- the `ListIterator` interface adds these operations to the basic Iterator interface
- it also keeps track of the position and can return the index of the next or previous item