

CSC 427: Data Structures and Algorithm Analysis

Fall 2007

Java Collections & List implementations

- Collection classes:
 - List (ArrayList, LinkedList), Set (TreeSet, HashSet), Map (TreeMap, HashMap)
- ArrayList implementation
- LinkedList implementation
- iterators

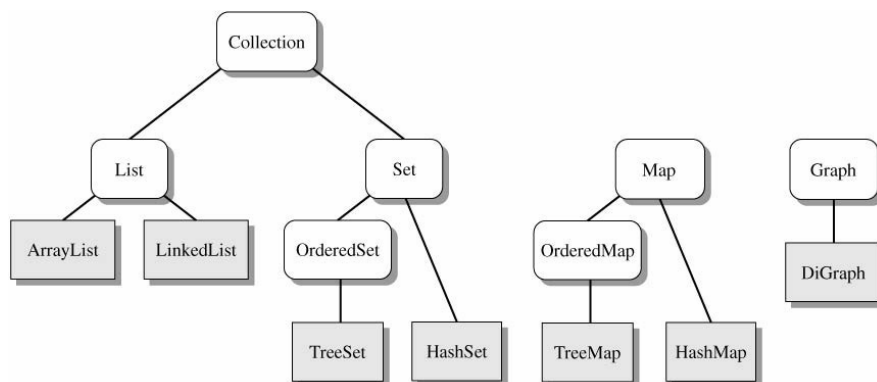
1

Java Collection classes

a collection is an object (i.e., data structure) that holds other objects

the Java Collection Framework is a group of generic collections

- defined using interfaces abstract classes, and inheritance



2

Sets & Maps

java.util.Set interface: an unordered collection of items, with no duplicates

```
public interface Set<E> extends Collection<E> {
    boolean add(E o); // adds o to this Set
    boolean remove(Object o); // removes o from this Set
    boolean contains(Object o); // returns true if o in this Set
    boolean isEmpty(); // returns true if empty Set
    int size(); // returns number of elements
    void clear(); // removes all elements
    Iterator<E> iterator(); // returns iterator
    ...
}
```

implemented by
TreeSet & HashSet
MORE LATER

java.util.Map interface: a collection of key → value mappings

```
public interface Map<K, V> {
    boolean put(K key, V value); // adds key→value to Map
    V remove(Object key); // removes key→? entry from Map
    V get(Object key); // returns true if o in this Set
    boolean containsKey(Object key); // returns true if key is stored
    boolean containsValue(Object value); // returns true if value is stored
    boolean isEmpty(); // returns true if empty Set
    int size(); // returns number of elements
    void clear(); // removes all elements
    Set<K> keySet(); // returns set of all keys
    ...
}
```

implemented by
TreeMap & HashMap
MORE LATER

3

Dictionary revisited

note: our Dictionary class could have been implemented using a Set

- the TreeSet implementation provides $O(\log N)$ add, remove, and contains
- the HashSet implementation provides *on average* $O(1)$ add, remove, and contains

implementation details later

```
import java.util.Set;
import java.util.HashSet;
import java.util.Scanner;
import java.io.File;

public class Dictionary {
    private Set<String> words;

    public Dictionary() {
        this.words = new HashSet<String>();
    }

    public Dictionary(String filename) {
        this();
        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String nextWord = infile.next();
                this.add(nextWord);
            }
        } catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }

    public void add(String newWord) {
        this.words.add(newWord.toLowerCase());
    }

    public void remove(String oldWord) {
        this.words.remove(oldWord.toLowerCase());
    }

    public boolean contains(String testWord) {
        return this.words.contains(testWord.toLowerCase());
    }
}
```

4

Word frequencies

a variant of Dictionary is WordFreq

- stores words & their frequencies (number of times they occur)
- can represent the word → counter pairs in a Map

implementation details later

```
import java.util.Map;
import java.util.TreeMap;
import java.util.Scanner;
import java.io.File;

public class WordFreq {
    private Map<String, Integer> words;

    public WordFreq() {
        words = new TreeMap<String, Integer>();
    }

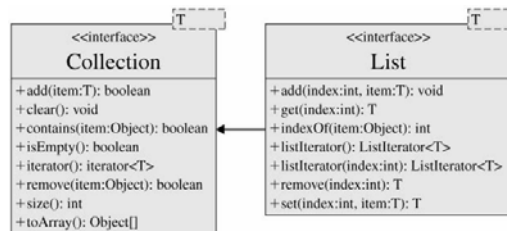
    public WordFreq(String filename) {
        this();
        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String nextWord = infile.next();
                this.add(nextWord);
            }
        } catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }

    public void add(String newWord) {
        String cleanWord = newWord.toLowerCase();
        if (words.containsKey(cleanWord)) {
            words.put(cleanWord, words.get(cleanWord)+1);
        } else {
            words.put(cleanWord, 1);
        }
    }

    public void showAll() {
        for (String str : words.keySet()) {
            System.out.println(str + ": " + words.get(str));
        }
    }
}
```

5

ArrayList implementation



recall: ArrayList implements the List interface

- which is itself an extension of the Collection interface
- underlying list structure is an array
 - get(index), add(item), set(index, item) → O(1)
 - add(index, item), indexOf(item), contains(item), remove(index), remove(item) → O(N)

6

ArrayList class structure

the ArrayList class has as fields

- the underlying array
- number of items stored

the default initial capacity is defined by a constant

- capacity != size

```
public class SimpleArrayList<E> implements Iterable<E>{
    private static final int INIT_SIZE = 10;
    private E[] items;
    private int numStored;

    public SimpleArrayList() {
        this.clear();
    }

    public void clear() {
        this.numStored = 0;
        this.ensureCapacity(INIT_SIZE);
    }

    public void ensureCapacity(int newCapacity) {
        if (newCapacity > this.size()) {
            E[] old = this.items;
            this.items = (E[]) new Object[newCapacity];
            for (int i = 0; i < this.size(); i++) {
                this.items[i] = old[i];
            }
        }
    }
}
```

interestingly: you can't create a generic array

```
this.items = new E[capacity]; // ILLEGAL
```

can work around this by creating an array of Objects, then casting to the generic array type

7

ArrayList: add

the add method

- throws an exception if the index is out of bounds
- calls ensureCapacity to resize the array if full
- shifts elements to the right of the desired index
- finally, inserts the new value and increments the count

the add-at-end method calls this one

```
public void add(int index, E newItem) {
    this.rangeCheck(index, "ArrayList add()", this.size());
    if (this.items.length == this.size()) {
        this.ensureCapacity(2*this.size() + 1);
    }

    for (int i = this.size(); i > index; i--) {
        this.items[i] = this.items[i-1];
    }
    this.items[index] = newItem;
    this.numStored++;
}

private void rangeCheck(int index, String msg, int upper) {
    if (index < 0 || index > upper)
        throw new IndexOutOfBoundsException("\n" + msg +
            ": index " + index + " out of bounds. " +
            "Should be in the range 0 to " + upper);
}

public boolean add(E newItem) {
    this.add(this.size(), newItem);
    return true;
}
```

8

ArrayList: size, get, set, indexOf, contains

size method

- returns the item count

get method

- checks the index bounds, then simply accesses the array

set method

- checks the index bounds, then assigns the value

indexOf method

- performs a sequential search

contains method

- uses indexOf

```
public int size() {
    return this.numStored;
}

public E get(int index) {
    this.rangeCheck(index, "ArrayList get()", this.size()-1);
    return items[index];
}

public E set(int index, E newItem) {
    this.rangeCheck(index, "ArrayList set()", this.size()-1);
    E oldItem = this.items[index];
    this.items[index] = newItem;
    return oldItem;
}

public int indexOf(E oldItem) {
    for (int i = 0; i < this.size(); i++) {
        if (oldItem.equals(this.items[i])) {
            return i;
        }
    }
    return -1;
}

public boolean contains(E oldItem) {
    return (this.indexOf(oldItem) >= 0);
}
```

9

ArrayList: remove

the remove method

- checks the index bounds
- then shifts items to the left and decrements the count
- note: could shrink size if becomes $\frac{1}{2}$ empty

the other remove

- calls indexOf to find the item, then calls remove(index)

```
public void remove(int index) {
    this.rangeCheck(index, "ArrayList remove()", this.size()-1);

    for (int i = index; i < this.size()-1; i++) {
        this.items[i] = this.items[i+1];
    }
    this.numStored--;
}

public boolean remove(E oldItem) {
    int index = this.indexOf(oldItem);
    if (index >= 0) {
        this.remove(index);
        return true;
    }
    return false;
}
```

could we do this more efficiently?
do we care?

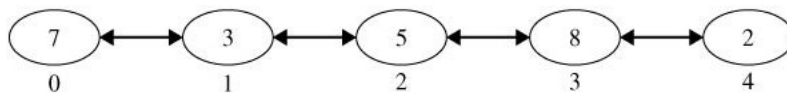
10

ArrayLists vs. LinkedLists

to insert or remove an element at an interior location in an ArrayList requires shifting data $\rightarrow O(N)$

LinkedList is an alternative structure

- stores elements in a sequence but allows for more efficient interior insertion/deletion
- elements contain links that reference previous and successor elements in the list



- if given a reference to an interior element, can reroute the links to insert/delete an element in $O(1)$

11

Singly-linked lists

in CSC222, you worked with singly-linked lists

- the list was made of Nodes, each of which stored data and a link to the next node in the list
- can provide a constructor and methods for accessing and setting these two fields
- a reference to the front of the list must be maintained

```
public class Node<E> {
    private E data;
    private Node<E> next;

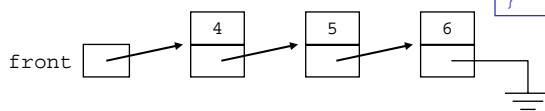
    public Node(E data, Node<E> next) {
        this.data = data;
        this.next = next;
    }

    public E getData() {
        return this.data;
    }

    public Node<E> getNext() {
        return this.next;
    }

    public void setData(E newData) {
        this.data = newData;
    }

    public void setNext(Node<E> newNext) {
        this.next = newNext;
    }
}
```



12

Exercises

to create an empty linked list:

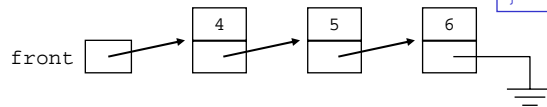
```
front = null;
```

to add to the front:

```
front = new Node(3, front);
```

remove from the front:

```
front = front.getNext();
```



```
public class Node<E> {  
    private E data;  
    private Node<E> next;  
  
    public Node(E data, Node<E> next) {  
        this.data = data;  
        this.next = next;  
    }  
  
    public E getData() {  
        return this.data;  
    }  
  
    public Node<E> getNext() {  
        return this.next;  
    }  
  
    public void setData(E newData) {  
        this.data = newData;  
    }  
  
    public void setNext(Node<E> newNext) {  
        this.next = newNext;  
    }  
}
```

13

Exercises

get value stored in first node:

get value in kth node:

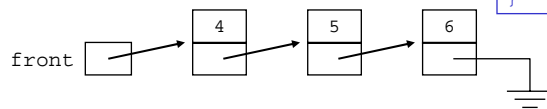
indexOf:

add at end:

add at index:

remove:

remove at index:



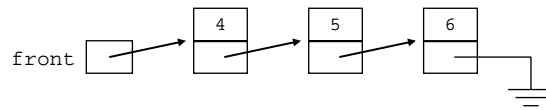
```
public class Node<E> {  
    private E data;  
    private Node<E> next;  
  
    public Node(E data, Node<E> next) {  
        this.data = data;  
        this.next = next;  
    }  
  
    public E getData() {  
        return this.data;  
    }  
  
    public Node<E> getNext() {  
        return this.next;  
    }  
  
    public void setData(E newData) {  
        this.data = newData;  
    }  
  
    public void setNext(Node<E> newNext) {  
        this.next = newNext;  
    }  
}
```

14

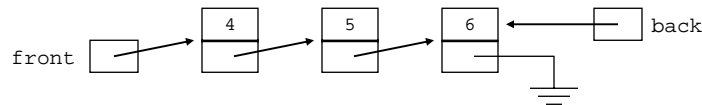
LinkedList implementation

we could implement the LinkedList class using a singly-linked list

- however, the one-way links are limiting
- to insert/delete from an interior location, really need a reference to the previous location
e.g., `remove(item)` must traverse and keep reference to previous node, so that when the correct node is found, can route links from previous node



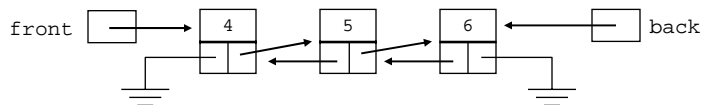
- also, accessing the end requires traversing the entire list
can handle this one special case by keeping a reference to the end as well



15

Doubly-linked lists

a better, although more complicated solution, is to have bidirectional links



- to move forward or backward in a doubly-linked list, use previous & next links
- can start at either end when searching or accessing
- insert and delete operations need to have only the reference to the node in question
- big-Oh?

<code>add(item)</code>	<code>add(index, item)</code>
<code>get(index)</code>	<code>set(index, item)</code>
<code>indexOf(item)</code>	<code>contains(item)</code>
<code>remove(index)</code>	<code>remove(item)</code>

16

Exercises

to create an empty list:

```
front = null;
back = null;
```

to add to the front:

```
front = new DNode(3, null, front);
if (front.getNext() == null) {
    back = front;
}
else {
    front.getNext().setPrevious(front);
}
```

remove from the front:

```
front = front.getNext();
if (front == null) {
    back = null;
}
else {
    front.setPrevious(null);
}
```

```
public class DNode<E> {
    private E data;
    private DNode<E> previous;
    private DNode<E> next;

    public DNode(E d, DNode<E> p, DNode<E> n) {
        this.data = d;
        this.previous = p;
        this.next = n;
    }

    public E getData() {
        return this.data;
    }

    public DNode<E> getPrevious() {
        return this.previous;
    }

    public DNode<E> getNext() {
        return this.next;
    }

    public void setData(E newData) {
        this.data = newData;
    }

    public void setPrevious(DNode<E> newPrevious) {
        this.previous = newPrevious;
    }

    public void setNext(DNode<E> newNext) {
        this.next = newNext;
    }
}
```

17

Exercises

get value stored in first node:

get value in kth node:

indexOf:

add at end:

add at index:

remove:

remove at index:

```
public class DNode<E> {
    private E data;
    private DNode<E> previous;
    private DNode<E> next;

    public DNode(E d, DNode<E> p, DNode<E> n) {
        this.data = d;
        this.previous = p;
        this.next = n;
    }

    public E getData() {
        return this.data;
    }

    public DNode<E> getPrevious() {
        return this.previous;
    }

    public DNode<E> getNext() {
        return this.next;
    }

    public void setData(E newData) {
        this.data = newData;
    }

    public void setPrevious(DNode<E> newPrevious) {
        this.previous = newPrevious;
    }

    public void setNext(DNode<E> newNext) {
        this.next = newNext;
    }
}
```

18

Dummy nodes

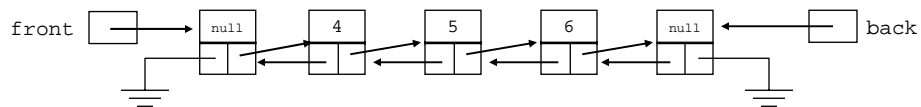
every time you add/remove, you need to worry about updating front & back

- if add at front/end of list, must also update end/front if previously empty
- if remove from front/end of list, must update end/front if now empty

the ends lead to many special cases in the code

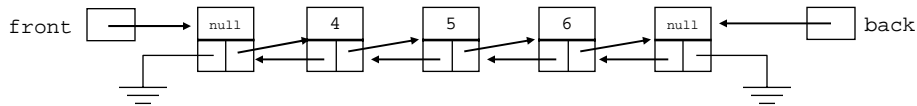
SOLUTION: add dummy nodes to both ends of the list

- the dummy nodes store no actual values
- instead, they hold the places so that the front & back never change
- removes special case handling



19

Exercises



to create an empty list (with dummy nodes):

```
front = new DNode(null, null, null);  
back = new DNode(null, front, null);  
front.setNext(back);
```

remove from the front:

```
front.setNext(front.getNext().getNext());  
front.getNext().setPrevious(front);
```

add at the front:

```
front.setNext(new DNode(3, front, front.getNext()));  
front.getNext().getNext().setPrevious(front.getNext());
```

get value stored in first node:

get value in kth node:

indexOf:

add at end:

add at index:

remove:

remove at index:

20

LinkedList class structure

the LinkedList class has an inner class

- defines the DNode class

fields store

- reference to front and back dummy nodes
- node count

the constructor

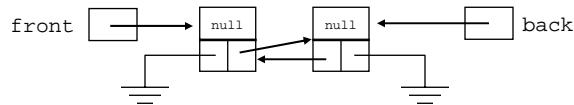
- creates the front & back dummy nodes
- links them together
- initializes the count

```
public class SimpleLinkedList<E> implements Iterable<E>{
    private class DNode<E> {
        . . .
    }

    private DNode<E> front;
    private DNode<E> back;
    private int numStored;

    public SimpleLinkedList() {
        this.clear();
    }

    public void clear() {
        this.front = new DNode(null, null, null);
        this.back = new DNode(null, front, null);
        this.front.setNext(this.back);
        this.numStored = 0;
    }
}
```



21

LinkedList: add

the add method

- similarly, throws an exception if the index is out of bounds
- calls the helper method `getNode` to find the insertion spot
- note: `getNode` traverses from the closer end
- finally, inserts a node with the new value and increments the count

add-at-end similar

```
public void add(int index, E newItem) {
    this.rangeCheck(index, "LinkedList add()", this.size());

    DNode<E> beforeNode = this.getNode(index-1);
    DNode<E> afterNode = beforeNode.getNext();

    DNode<E> newNode = new DNode<E>(newItem, beforeNode, afterNode);
    beforeNode.setNext(newNode);
    afterNode.setPrevious(newNode);

    this.numStored++;
}

private DNode<E> getNode(int index) {
    if (index < this.numStored/2) {
        DNode<E> stepper = this.front;
        for (int i = 0; i <= index; i++) {
            stepper = stepper.getNext();
        }
        return stepper;
    }
    else {
        DNode<E> stepper = this.back;
        for (int i = this.numStored-1; i >= index; i--) {
            stepper = stepper.getPrevious();
        }
        return stepper;
    }
}

public boolean add(E newItem) {
    this.add(this.size(), newItem);
    return true;
}
```

22

LinkedList: size, get, set, indexOf, contains

size method

- returns the item count

get method

- checks the index bounds, then calls getNode

set method

- checks the index bounds, then assigns

indexOf method

- performs a sequential search

contains method

- uses indexOf

```
public int size() {
    return this.numStored;
}

public E get(int index) {
    this.rangeCheck(index, "LinkedList get()", this.size()-1);
    return this.getNode(index).getData();
}

public E set(int index, E newItem) {
    this.rangeCheck(index, "LinkedList set()", this.size()-1);
    DNode<E> oldNode = this.getNode(index);
    E oldItem = oldNode.getData();
    oldNode.setData(newItem);
    return oldItem;
}

public int indexOf(E oldItem) {
    DNode<E> stepper = this.front.getNext();
    for (int i = 0; i < this.numStored; i++) {
        if (oldItem.equals(stepper.getData())) {
            return i;
        }
        stepper = stepper.getNext();
    }
    return -1;
}

public boolean contains(E oldItem) {
    return (this.indexOf(oldItem) >= 0);
}
```

23

LinkedList: remove

the remove method

- checks the index bounds
- calls getNode to get the node
- then calls private helper method to remove the node

the other remove

- calls indexOf to find the item, then calls remove(index)

```
public void remove(int index) {
    this.rangeCheck(index, "LinkedList remove()", this.size()-1);
    this.remove(this.getNode(index));
}

public boolean remove(E oldItem) {
    int index = this.indexOf(oldItem);
    if (index >= 0) {
        this.remove(index);
        return true;
    }
    return false;
}

private void remove(DNode<E> remNode) {
    remNode.getPrevious().setNext(remNode.getNext());
    remNode.getNext().setPrevious(remNode.getPrevious());
    this.numStored--;
}
```

could we do this more efficiently?
do we care?

24

Collections & iterators

many algorithms are designed around the sequential traversal of a list

- ArrayList and LinkedList implement the List interface, and so have get() and set()
- ArrayList implementations of get() and set() are O(1)
- however, LinkedList implementations are O(N)

```
for (int i = 0; i < words.size(); i++) {           // O(N) if ArrayList
    System.out.println(words.get(i));             // O(N2) if LinkedList
}
```

philosophy behind Java collections

1. a collection must define an efficient, general-purpose traversal mechanism
2. a collection should provide an *iterator*, that has methods for traversal
3. each collection class is responsible for implementing iterator methods

25

Iterator

the `java.util.Iterator` interface defines the methods for an iterator

```
interface Iterator<E> {
    boolean hasNext(); // returns true if items remaining
    E next();          // returns next item in collection
    void remove();    // removes last item accessed
}
```

any class that implements the Collection interface (e.g., List, Set, ...) is required to provide an `iterator()` method that returns an iterator to that collection

```
List<String> words;
...
Iterator<String> iter = words.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

both ArrayList and LinkedList implement their iterators efficiently, so O(N) for both

26

ArrayList iterator

an ArrayList does not really need an iterator

- get() and set() are already O(1) operations, so typical indexing loop suffices
- provided for uniformity (java.util.Collections methods require *iterable* classes)
- also required for enhanced for loop to work

to implement an iterator, need to define a new class that can

- access the underlying array (→ must be inner class to have access to private fields)
- keep track of which location in the array is "next"



27

SimpleArrayList iterator

java.lang.Iterable interface declares that the class has an iterator

inner class defines an Iterator class for this particular collection (accessing the appropriate fields & methods)

the iterator() method creates and returns an object of that class

```
public class SimpleArrayList<E> implements Iterable<E> {
    . . .
    public Iterator<E> iterator() {
        return new ArrayListIterator();
    }
    private class ArrayListIterator implements Iterator<E> {
        private int nextIndex;
        public ArrayListIterator() {
            this.nextIndex = 0;
        }
        public boolean hasNext() {
            return this.nextIndex < SimpleArrayList.this.size();
        }
        public E next() {
            if (!this.hasNext()) {
                throw new java.util.NoSuchElementException();
            }
            this.nextIndex++;
            return SimpleArrayList.this.get(nextIndex-1);
        }
        public void remove() {
            if (this.nextIndex <= 0) {
                throw new RuntimeException("Iterator call to " +
                    "next() required before calling remove()");
            }
            SimpleArrayList.this.remove(this.nextIndex-1);
            this.nextIndex--;
        }
    }
}
```

28

Iterators & the enhanced for loop

given an iterator, collection traversal is easy and uniform

```
SimpleArrayList<String> words;  
.  
.  
.  
Iterator<String> iter = words.iterator();  
while (iter.hasNext()) {  
    System.out.println(iter.next());  
}
```

as long as the class implements `Iterable<E>` and provides an `iterator()` method, the enhanced for loop can also be applied

```
SimpleArrayList<String> words;  
.  
.  
.  
for (String str : words) {  
    System.out.println(str);  
}
```

29

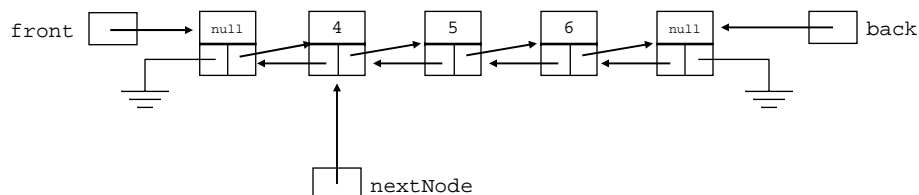
LinkedList iterator

a `LinkedList` does need an iterator to allow for efficient traversals & list processing

- `get()` and `set()` are already $O(N)$ operations, so a typical indexing loop is $O(N^2)$

again, to implement an iterator, need to define a new class that can

- access the underlying doubly-linked list
- keep track of which node in the list is "next"



30

SimpleLinkedList iterator

again, the class implements the Iterable<E> interface

inner class defines an Iterator class for this particular collection

iterator() method creates and returns an object of that type

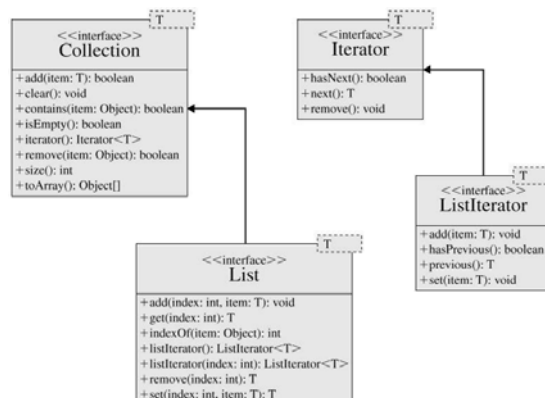
```
public class SimpleLinkedList<E> implements Iterable<E> {  
    . . .  
    public Iterator<E> iterator() {  
        return new LinkedListIterator();  
    }  
    private class LinkedListIterator implements Iterator<E> {  
        private DNode<E> nextNode;  
        public LinkedListIterator() {  
            this.nextNode = SimpleLinkedList.this.front.getNext();  
        }  
        public boolean hasNext() {  
            return this.nextNode != SimpleLinkedList.this.back;  
        }  
        public E next() {  
            if (!this.hasNext()) {  
                throw new java.util.NoSuchElementException();  
            }  
            this.nextNode = this.nextNode.getNext();  
            return this.nextNode.getPrevious().getData();  
        }  
        public void remove() {  
            if (this.nextNode == front.getNext()) {  
                throw new RuntimeException("Iterator call to " +  
                    "next() required before calling remove()");  
            }  
            SimpleLinkedList.this.remove(this.nextNode.getPrevious());  
        }  
    }  
}
```

31

Iterator vs. ListIterator

java.util.Iterator defines methods for traversing a collection

an extension, java.util.ListIterator, defines additional methods for traversing lists



32