CSC 427: Data Structures and Algorithm Analysis

Fall 2011

Java Collections & List implementations

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

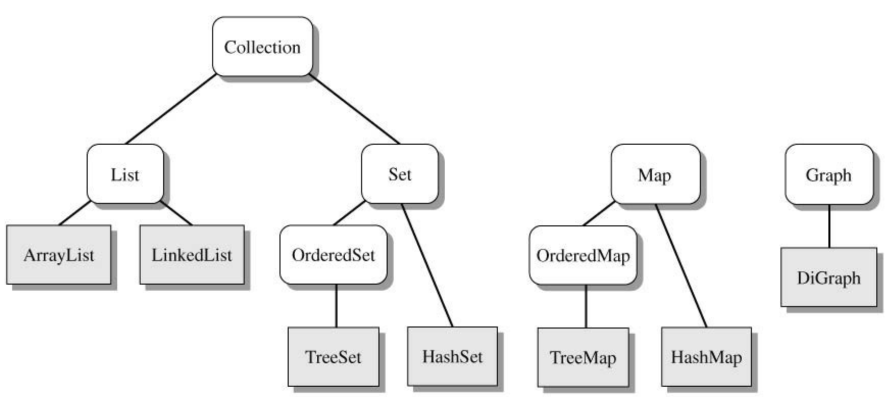
1

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

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 and TreeMap classes

TreeSet implementation

- ✓ utilizes a balanced binary search tree data structure; items must be Comparable
- ✓ provides O(log N) add, remove, and contains (guaranteed)

HashSet implementation

- ✓ HashSet utlizes a hash table data structure; all objects are hashable
- ✓ HashSet provides O(1) add, remove, and contains (on average, but can degrade)

(MORE IMPLEMENTATION DETAILS LATER)

3

Dictionary revisited

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

- Strings are Comparable, so could use either implementation
- HashSet is faster in practice
- TreeSet has the advantage that iterating over the Set elements gives them in order

```
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

Maps

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 and HashMap classes

TreeMap implementation

- ✓ utilizes a TreeSet to store key/value pairs; items must be Comparable
- ✓ provides O(log N) put, get, and containsKey (guaranteed)

HashMap implementation

- ✓ HashSet utlizes a HashSet to store key/value pairs; all objects are hashable
- ✓ HashSet provides O(1) put, get, and containsKey (on average, but can degrade)

5

Word frequencies

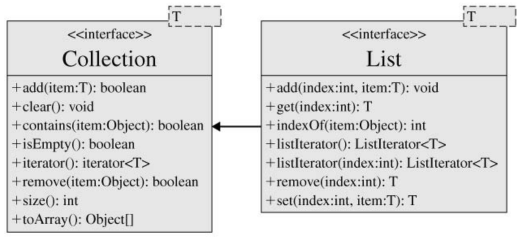
a variant of Dictionary is WordFreq

- stores words & their frequencies (number of times they occur)
- can represent the word→counter pairs in a Map
- again, could utilize either Map implementation
- since TreeMap is used, showAll displays words + counts in alphabetical order

```
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));
        }
    }
}
```

6

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)

7

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 MyArrayList<E> implements Iterable<E>{
    private static final int INIT_SIZE = 10;
    private E[] items;
    private int numStored;

    public MyArrayList() {
        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];
            }
        }
    }

    .
    .
    .
```

<u>interestingly:</u> 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

8

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;
}
```

9

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);
}
```

10

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 ½ 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?

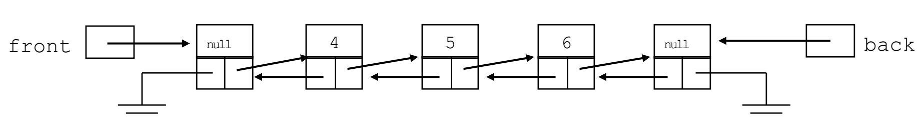
11

ArrayLists vs. LinkedLists

to insert or remove an element at an interior location in an ArrayList requires shifting data → 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



- can add/remove from either end in O(1)
- if given a reference to an interior element, can reroute the links to add/remove an element in O(1)

12

Doubly-linked Node

this class can be used to build a doubly-linked list

- note: DNode object contains two other DNode objects
- these are references to the previous and next nodes in the list

e.g., add at the front:

```
Dnode newNode =
  new DNode(3, front, front.getNext();
newNode.getPrevious().setNext(newNode,
  front.getNext());
newNode.getNext().setPrevious(front.getNext());
```

more details later

```
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;
  }
}
```


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 impementations 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 <u>efficient</u>, 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

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

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"

"foo"	"bar"	"biz"	"baz"	"boo"	"zoo"
0	1	2	3	4	5

nextIndex 0

ArrayList 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 MyArrayList<E> implements Iterable<E> {

    . . .

    public Iterator<E> iterator() {
         return new MyArrayListIterator();
    }

    private class MyArrayListIterator implements Iterator<E> {
      private int nextIndex;
      public MyArrayListIterator() {
          this.nextIndex = 0;
      }

      public boolean hasNext() {
          return this.nextIndex < MyArrayList.this.size();
      }

      public E next() {
          if (!this.hasNext()) {
              throw new java.util.NoSuchElementException();
          }
          this.nextIndex++;
          return MyArrayList.this.get(nextIndex-1);
      }

      public void remove() {
         if (this.nextIndex <= 0) {
             throw new RuntimeException("Iterator call to " +
                      "next() required before calling remove()");
         }
         MyArrayList.this.remove(this.nextIndex-1);
         this.nextIndex--;
      }
    }
}
```

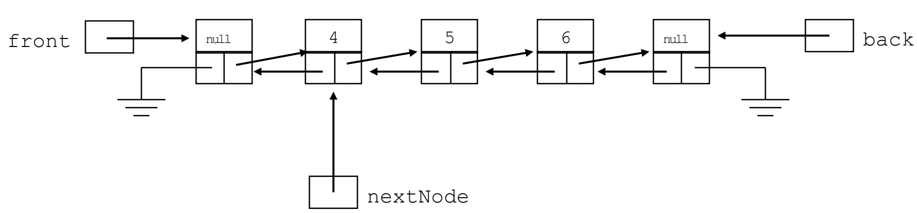

LinkedList iterator

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"



LinkedList 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 MyLinkedList<E> implement Iterable<E> {

    . . .

    public Iterator<E> iterator() {
         return new MyLinkedListIterator();
    }

    private class MyLinkedListIterator implements Iterator<E> {
      private DNode<E> nextNode;
      public MyLinkedListIterator() {
          this.nextNode = MyLinkedList.this.front.getNext();
      }

      public boolean hasNext() {
          return this.nextNode != MyLinkedList.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()");
         }
         MyLinkedList.this.remove(this.nextNode.getPrevious());
      }
  }
}
```

19