CSE 143 Java

Lists via Arrays

Reading: Ch. 22

List Interface (review)

```
int size( )
boolean isEmpty( )
boolean add(Object o)
boolean addAll(Collection other)
void clear( )
Object get(int pos)
boolean set(int pos, Object o)
int indexOf(Object o)
boolean contains(Object o)
Object remove(int pos)
boolean remove(Object o)
boolean add(int pos, Object o)
Iterator iterator( )
```

- How to implement these operations?

Implementing a List

- Key concept: *external view* (the **abstraction** visible to clients) vs. *internal view* (the **implementation**)
- Two implementation approaches are most commonly used for simple lists:
  - List via Arrays
  - Linked list
- In Java, interface List
  - concrete classes ArrayList, LinkedList
  - same methods, different internals
- Our current activities:
  - Lectures on list implementations, in gruesome detail
  - Projects in which lists are used

First Strategy: Use an Array

- Idea: store the list elements in an array

```
// Simple version of ArrayList for CSE143 lecture example
public class SimpleArrayList implements List {
    /** variable to hold all elements of the list*/
    private Object[ ] elements;
    ...
}
```

- Issues:
  - How big to make the array?
  - Algorithms for adding and deleting elements (add and remove methods)
  - Coming soon: performance analysis of the algorithms

Space Management: Size vs. Capacity

- Idea: allocate extra space in the array,
  - possibly more than is actually needed at a given time
  - *size*: the number of elements in the list, from the client's view
  - *capacity*: the length of the array (the maximum size)
  - invariant: 0 <= *size* <= *capacity*
- When list object created, create an array of some initial maximum capacity
  - What happens if we try to add more elements than the initial capacity? see later…

List Representation

```
public class SimpleArrayList implements List {
    // instance variables
    private Object[ ] elements;    // elements stored in elements[0..numElems-1]
    private int numElems;          // size: # of elements currently in the list
    // capacity ?? Why no capacity variable??

    // default capacity
    private static final int defaultCapacity = 10;

    …
}
```

Why make the array of type Object[ ]? Pros, cons?

Review: what is the "static final"?

Constructors

- We'll provide two constructors: one where user specifies initial capacity, the other that uses a default initial capacity

```
/** Construct new list with specified capacity */
public SimpleArrayList(int capacity) {
    this.elements = new Object[capacity];
    this.numElems = 0;
}
/** Construct new list with default capacity */
public SimpleArrayList( ) {
    this(defaultCapacity);
}
```

  - Footnote: this(…) can appear as the first statement of a constructor to call another constructor for the same class

size, isEmpty

- size:

```
/** Return size of this list */
public int size( ) {
    return this.numElems;
}
```

- isEmpty:

```
/** Return whether the list is empty (has no elements) */
public boolean isEmpty( ) {
    return this.size( ) == 0;
}
```

  - could put isEmpty in an abstract superclass, since it doesn't depend on the implementation at all

Method add (simple version)

- Assuming there is unused capacity ...

```
/** Add object o to the end of this list
    @return true, since list is always changed by an add */
public boolean add(Object o) {
    if (this.numElems < this.elements.length) {
        this.elements[this.numElems] = o;
        this.numElems ++;
    } else {
        // yuck; what can we do here?  here's a temporary measure....
        throw new RuntimeException("list capacity exceeded");
    }
    return true;
}
```

- addAll(…) left as an exercise (can be put in an abstract superclass)

clear

- Logically, all we need to do is set numElems to 0
  - Why?
- But it's probably a good idea to null out all of the object references in the list. Why?

```
/** Empty this list */
public void clear( ) {
    for (int k = 0; k < this.numElems; k++) {
        this.elements[k] = null;
    }
    this.numElems = 0;
}
```


get, set

- We want to catch out-of-bounds arguments, including ones that reference unused parts of array elements

```
/** Return object at position pos of this list
    0 <= pos < size( ), or IndexOutOfBoundsException is thrown */
public Object get(int pos) {
    if (pos < 0  ||  pos >= this.numElems) {
        throw new IndexOutOfBoundsException( );
    }
    return this.elements[pos];
}
```

- Question: is a "throws" clause required?
- set method is similar, and left as an exercise

indexOf

- Sequential search for first "equal" object

```
/** return first location of object o in this list if found, otherwise return –1 */
public int indexOf(Object o) {
    for (int k = 0; k < this.size( ); k++) {
        Object elem = this.get(k);
        if (elem.equals(o)) {
            // found item; return its position
            return k;
        }
    }
    // item not found
    return -1;
}
```

- Claim: Can move this to an abstract superclass

contains

```
/** return true if this list contains object o, otherwise false */
public boolean contains(Object o) {
    // just use indexOf
    return this.indexOf(o)  !=  -1;
}
```


remove at position

- Key idea: we need to compact the array after removing something in the middle; slide all later elements left one position

```
/** Remove the object at position pos from this list.  Return the removed element.
    0 <= pos < size( ), or IndexOutOfBoundsException is thrown */
public Object remove(int pos) {
    if (pos < 0 || pos >= this.numElems) {
        throw new IndexOutOfBoundsException( );
    }
    Object removedElem = this.elements[pos];
    for (int k = pos+1; k < this.numElems; k++) {
        this.elements[k-1] = this.elements[k];      // slide k'th element left by one index
    }
    this.numElems --;
    this.elements[this.numElems] = null;           // erase extra ref. to last element, for GC
    return removedElem;
}
```


remove Object

```
/** Remove the first occurrence of object o from this list, if present.
    @return true if list altered, false if not */
public boolean remove(Object o) {
    int pos = indexOf(o);
    if (pos != -1) {
        remove(pos);
        return true;
    } else {
        return false;
    }
}
```


add at position

- Key idea: we need to make space in the middle; slide all later elements right one position

```
/** Add object o at position pos in this list.  List changes, so return true
    0 <= pos < size( ), or IndexOutOfBoundsException is thrown */
public boolean add(int pos, Object o) {
    if (pos < 0 || pos >= this.numElems) {
        throw new IndexOutOfBoundsException( );
    }
    if (this.numElems >= this.elements.length) {
        // yuck; what can we do here?  here's a temporary measure....
        throw new RuntimeException("list capacity exceeded");
    }
    ... continued on next slide ...
```

add at position (continued)

```
/** Add object o at position pos in this list.  List changes, so return true
    0 <= pos < size( ), or IndexOutOfBoundsException is thrown */
public boolean add(int pos, Object o) {
    ...
    // first create space
    for (int k = this.numElems - 1; k >= pos; k --) {    // must count down!
       this.elements[k+1] = this.elements[k];// slide k'th element right by one index
    }
    this.numElems ++;
    // now store object
    this.elements[pos] = o;          // erase extra ref. to last element, for GC
    return true;
}
```


add Revisited – Dynamic Allocation

- The original version of add did not handle the case when adding an object to a list with no spare capacity
- A fix: if the array is already full:
  1. allocate a new array with larger capacity,
  2. copy the elements from the old array to the new array, and
  3. replace the old array with the new one
- Question: How big should the new array be?

Method add with Dynamic Allocation

```
/** Add object o to the end of this list
    @return true, since list is always changed by an add */
public boolean add(Object o) {
     this.ensureExtraCapacity(1);
     this.elements[this.numElems] = o;
     this.numElems ++;
     return true;
}
// likewise in add(int pos, Object o)

/** Ensure that elements has at least extraCapacity free space,
    growing elements if needed */
private void ensureExtraCapacity(int extraCapacity) {
     ... magic here ...
}
```


ensureExtraCapacity

```
/** Ensure that elements has at least extraCapacity free space,
    growing elements if needed */
private void ensureExtraCapacity(int extraCapacity) {
    if (this.numElems + extraCapacity > this.elements.length) {
       // we need to grow the array
       int newCapacity = this.elements.length * 2 + extraCapacity;
       Object[] newElements = new Object[newCapacity];
       for (int k = 0; k < this.numElems; k++) {
          newElements[k] = this.elements[k];
       }
       this.elements = newElements;
    }
}
```

iterator

- A collection class should provide a method iterator( ) that returns a suitable Iterator for objects of that class
  - Core interface: boolean hasNext( ), Object next( )
  - List iterators also support remove( ) [left as an exercise]
- Idea: Iterator object holds a reference to the list it is traversing and the current position in that list.
  - Can be used for any List, not just ArrayList!

Method iterator

- In class SimpleArrayList

```
/** Return a suitable iterator for this list */
public Iterator iterator( ) {
    return new SimpleListIterator(this);
}
```


Class SimpleListIterator (1)

```
/** Iterator helper class for lists */
class SimpleListIterator implements Iterator {
    // instance variables
    private List list;              // the list we are traversing
    private int nextItemPos;        // position of next element to visit (if any left)
    // invariant: 0 <= nextItemPos <= list.size( )

    /** construct iterator object */
    public SimpleListIterator(List list) {
       this.list = list;
       this.nextItemPos = 0;
    }

    ...
```


Class SimpleListIterator (2)

```
/** return true if more objects remain in this iteration */
public boolean hasNext( ) {
    return this.nextItemPos < this.list.size( );
}
/** return next item in this iteration and advance.
    @throws NoSuchElementException if iteration has no more elements */
public Object next( ) {
    if ( ! hasNext( ) ) {
       throw new NoSuchElementException( );
    }
    Object result = this.list.get(this.nextItemPos);
    this.nextItemPos ++;
    return result;
}
```

Summary

- SimpleArrayList presents an illusion to its clients
  - Appears to be a simple, unbounded list of elements
  - Actually a more complicated array-based implementation
- Key implementation ideas:
  - capacity vs. size/numElems
  - sliding elements to implement (inserting) add and remove
  - growing to increase capacity when needed
    growing is transparent to client
- Frequent sliding and growing is likely to be expensive….