CSE 143 Java

List Implementation Using Arrays

Reading: Ch. 22

Implementing a List in Java

- Two implementation approaches are most commonly used for simple lists:
  - Arrays
  - Linked list
- Java Interface List
  - concrete classes ArrayList, LinkedList
  - same methods, different internals
  - List in turn extends (implements) Collection
- Our current activities:
  - Lectures on list implementations, in gruesome detail
    SimpleArrayList is a class we develop as an example
  - Projects in which lists are used

List Interface (review)

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


Just an Illusion?

- Key concept: *external view* (the *abstraction* visible to clients) vs. *internal view* (the *implementation*)
- SimpleArrayList may present an illusion to its clients
  - Appears to be a simple, unbounded list of elements
  - Actually may be a complicated internal structure
- The programmer as illusionist...

- This is what abstraction is all about

Java Arrays (Review)

- **Key difference from other languages: declaring an array doesn't create it – it must be allocated with new**

```
int[ ] numbers;
numbers = new int[42];          // creates numbers[0]..numbers[41]
```

or

```
int[ ] numbers = new int[42];
```

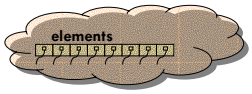
- **Size is fixed when array is allocated**
- **Element access: *arrayname*[*position*]**
- **Every array object can report how many elements it contains**

```
int capacity = numbers.length
```


Using an Array to Implement a List

- **Idea: store the list elements in an array instance variable**

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



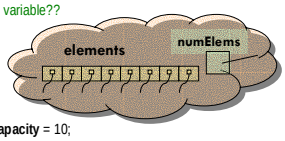
- **Issues:**
  - **How big to make the array?**
  - **Why make the array of type Object[ ]? Pros, cons?**
  - **Algorithms for adding and deleting elements (add and remove methods)**
  - **Later: performance analysis of the algorithms**

Space Management: Size vs. Capacity

- **Idea: allocate extra space in the array,**
  - **possibly more than is actually needed at a given time**
- **Definitions**
  - ***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? We'll get to that...**

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;
    ...
}
```

Review: what is the "static final"?

Constructors

- **We'll provide two constructors:**

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

/** Construct new list with default capacity */
public SimpleArrayList( ) {
    this(defaultCapacity);
}
```

- **Review: this(...)**
  - means what?
  - can be used where?

size, *isEmpty*: Code

- **size:**

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

- **isEmpty:**

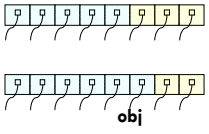
```
/** Return whether the list is empty (has no elements) */
public boolean isEmpty( ) {
    return size( ) == 0;          // OR return numElems == 0;
}
```

- **Each choice has pros and cons: what are they?**

Method *add*: simple version

- **Assuming there is unused capacity ...**

```
/** Add object obj to the end of this list.
    @return true iff the object was added successfully.
      This implementation always returns true. */
public boolean add(Object obj) {
```




```
    return true;
}
```


Method *add*: simple version

- **Assuming there is unused capacity ...**

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

- **addAll(array or list) left as an exercise – try it at home!**

clear

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

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


Method *get*

```
/** Return object at position pos of this list
The list is unchanged
*/
public Object get(int pos) {
        return elements[pos];
}
```

- **Anything wrong with this?**
  Hint: what are the preconditions?

A Better *get* Implementation

- **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 >= numElems) {
        throw new IndexOutOfBoundsException( );
    }
    return elements[pos];
}
```

- **Question: is a "throws" clause required?**
- **Exercise: write out the preconditions more fully**
- **Exercise: specify and implement the *set* method**

Method *indexOf*

- **Sequential search for first "equal" object**

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

- **Exercise: write postconditions**

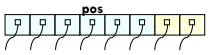
Method *contains*

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

- **Also possible to write the search loop here. Tradeoffs?**
- **Exercise: define "this list contains object obj" more rigorously**

remove(pos): Specification

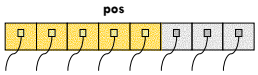
```
/** Remove the object at position pos from this list.  Return the removed element.
    0 <= pos < size( ), or IndexOutOfBoundsException is thrown */
public Object remove(int pos) {
...
    return removedElem;
}
```



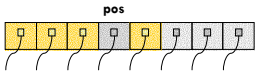
- **Postconditions: quite a bit more complicated this time...**
  - **Try writing them out!**
- **Key observation for implementation:**
  - **we need to compact the array after removing something in the middle; slide all later elements left one position**

Array Before and After *remove*

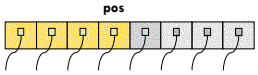
- Before



- After – Wrong!



- After – Right!



remove(pos)

```
/** 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 >= numElems) {
        throw new IndexOutOfBoundsException( );
    }
    Object removedElem = elements[pos];
    for (int k = pos+1; k < numElems; k++) {
        elements[k-1] = elements[k];   // slide k'th element left by one index
    }
    numElems[numElems-1] = null;   // erase extra ref. to last element, for GC
    numElems--;
    return removedElem;
}
```

remove(Object)

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

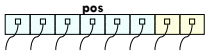
- **Pre- and postconditions are not quite the same as remove(pos)**

add Object at position

```
/** Add object obj at position pos in this list.  List changes, so return true
    0 <= pos < size( ), or IndexOutOfBoundsException is thrown */
public boolean add(int pos, Object obj) {

    ...
```

- **Key implementation idea:**
  - **we need to make space in the middle; slide all later elements right one position**
- **Pre- and postconditions?**



add(pos, obj): Code

```
/** Add object obj at position pos in this list.  List changes, so return true
    0 <= pos < size( ), or IndexOutOfBoundsException is thrown */
public boolean add(int pos, Object obj) {
    if (pos < 0 || pos >= numElems) {
        throw new IndexOutOfBoundsException( );
    }
    if (numElems >= elements.length) {
        // out of room – for now, ...
        throw new RuntimeException("list capacity exceeded");
    }
    ... continued on next slide ...
```


add(pos, obj) (continued)

```
    ...
    // preconditions have been met
    // first create a space
    for (int k = numElems - 1; k >= pos; k --) {   // must count down!
        elements[k+1] = elements[k];          // slide k'th element right by one index
    }
    numElems ++;

    // now store object in the space opened up
    elements[pos] = obj;              // erase extra ref. to last element, for GC
    return true;
}
```

add Revisited – Dynamic Allocation

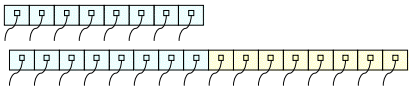
- Our original version of add checked for the case when adding an object to a list with no spare capacity
  - But did not handle it gracefully: threw an exception
- Better handling: "grow" the array
- Problem: Java arrays are fixed size – can't grow or shrink
- Solution: Make a new array of needed size & copy contents of old array to new, then add
- This is called *dynamic allocation*

Dynamic Allocation Algorithm

Algorithm

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

   i.e., make the array name refer to the new array



- Issue: How big should the new array be?

Method *add* with Dynamic Allocation

- This implementation has the dynamic allocation hidden away...

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

/** 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 (numElems + extraCapacity > elements.length) {
        // we need to grow the array – allocate new array and copy elements
        int newCapacity = elements.length * 2 + extraCapacity;
        Object[ ] newElements = new Object[newCapacity];
        for (int k = 0; k < numElems; k++) {
            newElements[k] = elements[k];
        }
        elements = newElements;
    }
}
```

- Note: this is *ensure* extra capacity, not add extra capacity.
- Pre- and Post- conditions?

Method *iterator*

- Review: Collection method *iterator*( ) returns a suitable Iterator for objects of that class
  - Key Iterator methods: boolean hasNext( ), Object next( )
  - Method remove( ) is optional for Iterator in general, but expected to be implemented for lists. [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!
- Except for remove( ), iterator operations should never modify the underlying list

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) {
       list = list;
       nextItemPos = 0;
    }

    ...
```


Class SimpleListIterator (2)

```
/** return true if more objects remain in this iteration */
public boolean hasNext( ) {
    return nextItemPos < list.size( );
}
/** return next item in this iteration and advance.
Note: changes the state of the Iterator but not of the List
   @throws NoSuchElementException if iteration has no more elements */
public Object next( ) {
    if ( ! hasNext( ) ) {
       throw new NoSuchElementException( );
    }
    Object result = list.get(nextItemPos);
    nextItemPos ++;
    return result;
}
```

Design Questions

- Why create a separate Iterator object?
- Couldn't the list itself have..
  - ...operations for iteration?
    hasNext( )
    next( )
    reset( ) //start iterating again from the beginning
  - ...private instance variable for nextPos?
- Would it have been better to implement the iterator as a nested class inside the simple list class?
  - Yes, probably

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
- Caution: Frequent sliding and growing is likely to be expensive....