CS310 - Advanced Data Structures and Algorithms

Fall 2014 – Class 3

Feb. 4, 2014

Announcements

- HW2 is online, due next Tuesday, Feb. 11.
- HW1 due today in class.
- Apply for an account ASAP if you haven't yet, you will need it for HW2 and PA1 next week.

Collections in Java

- A collection is a container of objects.
- A collection may be ordered or unordered.
- It may or may not allow duplicates.
- The Java Collection Interface lays down the foundation – see pg. 239.
- Chapter 6 covers the Java Collections:
- List, Stacks, Queues, Sets, and Maps.

Collections in Java

```
 // Collection interface; the root of all 1.5 collections.
public interface Collection<AnyType> extends Iterable<AnyType>,
java.io.Serializable
{     int size(); // How many items are in this collection.
    boolean isEmpty(); // Is this collection empty?
    boolean contains( Object x ); // is X in this collection?
    boolean add( AnyType x ); // Adds x to collection.
    boolean remove( Object x ); // Removes x from collection.
    void clear(); // Change collection size to zero.
    // Obtains an Iterator object to traverse collection.
    Iterator<AnyType> iterator( );
    // Obtains a primitive array view of the collection.
    Object [] toArray();
    // Obtains a primitive array view of the collection.
    <OtherType> OtherType[] toArray(OtherType [] arr );
}
```

The Collections Interface

- Note how all the elements for a Collection are Objects, since a type parameter can only take on Object types.
- The only non-Object types are int, double, char, etc., the primitive types.
- However, each of these has a corresponding “wrapper” Object type: Integer, Double, etc., and autoboxing makes it easy to use these collections.

Iterators Traverse Collections

```
 // Iterator interface
public interface Iterator<AnyType> extends
java.util.Iterator<AnyType>
{
    // Are there any items not iterated over
    boolean hasNext();
    // Obtains the next (as yet unseen)
    // item in the collection
    AnyType next();
    // Remove the last item returned by next.
    // Can only be called once after next
    void remove();
}
```

A Simple Client Program Example

```
public class ReadStringsWithArrayList {
// Read an unlimited number of String;
    public static ArrayList getStrings( )
    {
        BufferedReader in = new BufferedReader( new
        InputStreamReader( System.in ) );
        ArrayList<String> array = new ArrayList<String>( );
        // create a collection
        String oneLine;
        System.out.println("Enter any number of strings,
            one per line;");
        System.out.println( "Terminate with empty line:  " );
        try {
            while( ( oneLine = in.readLine( ) ) != null &&
            !oneLine.equals( "" ) )
                array.add( oneLine ); // add to collection
        } catch( IOException e ) {
            System.out.println( "Unexpected IO Exception has shortened
                amount read‘‘);
        }
        System.out.println( "Done reading" );
        return array;
    }
}
```

A Simple Client Program Example

```
public class ReadStringsWithArrayList
{
  public static void main( String [ ] args )
  {
    ArrayList array = getStrings( );
    for( int i = 0; i < array.size( ); i++ )
      // Loop through a collection
      System.out.println( array.get( i ) );
  }
}
```

Client Program Example

- We can replace the BufferedReader by Scanner as in:
  ```
  Scanner in = new Scanner(System.in);
  ```
- Then loop over in.hasNextLine(), oneLine = in.nextLine().
- Improve the code by changing ArrayList to ArrayList<String>.
- ArrayList is the "raw" type, and is a superclass of ArrayList<T> for any T, so it's good Java here, but unnecessarily loosely typed.
- Possible to use Iterator for the printing:
  ```
  Iterator<String> itr = array.iterator();
  // any Collection has an iterator
  while (itr.hasNext())
  System.out.println(itr.next());
  ```

Encapsulation of Collection Objects

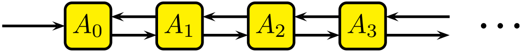
- In Java, the collection object is fully encapsulated.
- We can't see the collection itself.
- We are allowed to get references to the objects in the collection.
- Each of them should be individually encapsulated.
- The coverage on "basic iterators" in Weiss, Ch 6, pp 232-236, doesn't get to full encapsulation.
- To see how to do this, see Ch. 15.The secret is in using an inner class for the iterator.

Iterators in Java

- Add an object to the collection by handing over the reference to the element object.
- The caller can still have a ref to it, so the object has 2 refs to it, the caller's and the collection's.
- This is a case of "aliasing".
- Can cause problems if the external ref is used to change the object in the collection.

Collection Types: The List Interface

- A List is an ordered sequence of elements: $a_0, a_1, a_2, ..., a_{n-1}$.



- List interface on pg. 248, extending the Collection interface on pg. 239.

Collection Types: The List Interface

```
// List interface.
public interface List<AnyType> extends Collection<AnyType>
{
  // Returns the item at position idx.
  AnyType get( int idx );

  // Changes the item at position idx.
  AnyType set( int idx, AnyType newVal );

  // Obtains a ListIterator object used to traverse the collection
bi-directionally.
  ListIterator<AnyType> listIterator( int pos );
}
```

Collection Types: The List Interface

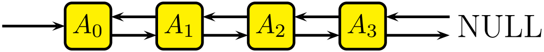
```
// ListIterator interface for List interface.
public interface ListIterator<AnyType> extends
Iterator<AnyType>
{
  // Tests if there are more items in the collection
when iterating in reverse.
  boolean hasPrevious( );
  //Obtains the previous item in the collection when
traversing in reverse.
  AnyType previous( );
  // Remove the last item returned by next or
previous.
  //Can only be called once after next or previous.
  void remove( );
}
```

Collection Types: The List Interface

- The two most important classes that implement the List interface are **LinkedList** and **ArrayList**.
- They have different performance for large lists.
- Both have extra methods over and above the List interface.

Mental Model of a List

- See the picture on pg. 251 – a 4 member list



- We can get(0), ..., get(3) and access any particular object ref.
- We can set(0, b) and replace the object at 0 with b.
- What happens if we set(4, b)?
- To grow the list we need to use add(Object x), but where does it go??
- This is fast because the LinkedList tracks the end-of-list.

Collections in Java

- A ListIterator starting from 0, has a next method that returns element 0 on first call, element 1 on second, etc.
- Should test with hasNext before doing a next.
- If hasNext returns false, the iterator is at end of list (EOL).
- It starts at beginning of list, so there are 5 different iterator states for 4 elements:

```
< A0 > < A1 > < A2 > < A3 >
↑ original iterator, before element 0
       ↑ after first next, returning A0
         (just before element 1)
               ↑ after 2nd next, returning A1
                      ↑ after 3rd next, returning A2
                              ↑ after last element
```

Mental Model of a ListIterator

- The iterator is positioned just before the element to be returned by next.
- A ListIterator can go both ways.
- At each point in time, an iterator is positioned just after the element that would be returned by previous(), and just before the element that would be returned by next().
- When we talk about numerical position in a list, its normally about the position of an element, not directly the iterator.
- We can talk about the iterator as being between certain numbered elements.

Mental Model of a ListIterator

- With the listIterator(int pos) method, the pos determines an element position and the method returns an iterator positioned prior to that element, or at EOL if pos == size of list.
- listIterator(0) gives an iterator positioned before the very first element, etc.
- A special case is that extra position after all the elements: this is attained by using N as an arg, the number of elements in the list.
- **Remember: an iterator has N+1 possible positions for N elements.**

Question

- What happens if next returns A1, then another next returns A2, and then a previous is done – is A1 returned?
- No! We've just gone past A2 one way, and now we go back across it again, so A2 gets returned again.

Question

- What happens if next returns A1, then another next returns A2, and then a previous is done – is A1 returned?
- No! We've just gone past A2 one way, and now we go back across it again, so A2 gets returned again.

List Client Code (pg. 250)

```
 class TestArrayList
{
    public static void main( String [ ] args )
    {
        ArrayList<Integer> lst = new ArrayList<Integer>( );
        lst.add( 2 );
        lst.add( 4 );
        ListIterator<Integer> itr1 = lst.listIterator( 0 );
        System.out.print( "Forward:  " );
        while( itr1.hasNext( ) )
            System.out.print( itr1.next( ) + " " );
        System.out.println( );
        System.out.print( "Backward:  " );
        while( itr1.hasPrevious( ) )
            System.out.print( itr1.previous( ) + " " );
        System.out.println( );
        System.out.print( "Backward:  " );
        ListIterator<Integer> itr2 =
        lst.listIterator( lst.size( ) ););
        while( itr2.hasPrevious( ) )
            System.out.print( itr2.previous( ) + " " );
        System.out.println( );
        System.out.print( "Forward:  ");
        for( Integer x :  lst )
            System.out.print( x + " " );
        System.out.println( );
    }

}
```

List Client Code (pg. 250)

- Here the ListIterator<Integer> goes all the way down the list to EOL, then back along the list, so the turn-around occurs at EOL.
- Another ListIterator<Integer> starts from "lst.size()", which would be 4 for our list.
- This is an artificial element number denoting the EOL position of the iterator.
- Again, there are n+1 different iterator states for n elements, and these are numbered from 0 to n.

Iterator Remove

- An iterator sits between elements.
- When calling remove, which nearby element gets removed?
- The object removed is the last one returned by next or previous, and only one remove per movement-action is allowed.
- What happens if you next, remove, and then next again?
- You access the element just after the removed element. Because weve moved past the deleted element already, the iterator position is clear.
- If you next, remove, previous, you should get the previous-to-removed. And so on, using the model above.

Iterator Remove Example

- Remove all objects from list of EOrder equal to given object z
- Two ways:
  1. Use remove(z) removing one object each time; loop on these calls until returns false (quadratic!).
  2. Use iteration down list and call remove() of the Iterator (linear).

```
 Iterator<EOrder> itr = list.iterator();
while (itr.hasNext()) {
    EOrder o1 = itr.next();
     // note:  no cast needed:  next delivers EOrder
    System.out.println("working on " + o1);
    // check that this element needs to be removed
    if (o1.equals(z))
        itr.remove();
    // remove the element that itr just stepped over
}
```

Another Example – Remove Duplicates With Two Iterators

```
 Iterator<EOrder> itr = list.iterator();
int position = 0;
while (itr.hasNext()) {
    System.out.println("about to do next() in outer
loop...");
    EOrder o1 = itr.next();
    // throws exception here,after first remove
    System.out.println("working on " + o1);
    ListIterator listItr =
        list.listIterator(position);
    while (listItr.hasNext()) {
        EOrder o2 = listItr.next();
        if (o1.equals(o2))
            listItr.remove();
    }
    position++;
}
```

Remove Duplicates With Two Iterators

- Example: Using two iterators to remove duplicates from a LinkedList doesn't work!
- Algorithm: scan LinkedList list with itr, from list.iterator()
- For each itr-position, with element o1, initialize a ListIterator (listItr) at that position.
- Scan rest of list with listItr, removing elements that equal o1.
- The exception is ConcurrentModificationException.

Performance of LinkedList vs. ArrayList

- For ArrayList of size n
  - Get, set are very fast , O(1)
  - Append-type add is fast most of the time. If it involves array expansion, it is expensive, O(n).
  - Delete is expensive unless if it is at the end.
- For LinkedList of size n
  - Get, set depends on the index position
  - get(1) is done by two nexts down the list from the beginning of the list, and get(n-2) is done by two previous nexts from the end of the list
  - Most expensive is get(n/2)
  - Delete/add is easy once the right spot in the list is located. Remove in an iterator is O(1), but the larger task may involve O(n) nexts to get the iterator positioned.

Stacks and Queues

A Stack is a specialized List where can only insert (push), retrieve (top), and delete (pop) elements at one end.

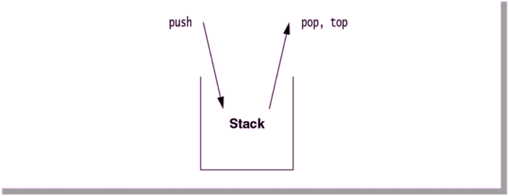


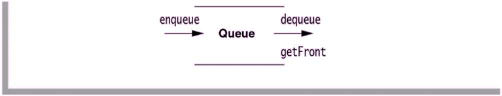
figure 6.20

The stack model: Input to a stack is by `push`, output is by `top`, and deletion is by `pop`.

A Queue is a specialized list where insert at one end, retrieve and delete at the other.

figure 6.22

The queue model: Input is by `enqueue`, output is by `getFront`, and deletion is by `dequeue`.



Linked Lists in C (optional)

- Because of C's lack of garbage collection, we hold the element objects fully inside the collection object.
- Copy-in, Copy-out behavior: Instead of handing an object reference over to the List as we do in Java, we copy in the element object, and later do a copy out in get or next. That way, the ownership of the memory of the element is well-defined:the caller still has its old element copy, and the list has its own copy. When the element is finally removed from the list, the list can de-allocate that element's memory.
- Java add object x: create a new list-node object, with a next-ref and a spot for the element reference, and copy the elements reference there.
- C add action, for "object" x: create a new list-node element, with next-pointer and an area for the whole element, and copy the element's data there.