CS 61B Data Structures and Programming Methodology

June 30 2008

David Sun
Announcements

• Midterm I is next Wed during class. If you have timing conflicts, email cs61b@imail today.
Today

• Lists
• Invariants
From last time...

• Arrays:
  – Good: random access to items.
  – Bad: hard to expand, insert items, or delete items.

• Linked lists:
  – Good: easy to expand, insert, delete.
  – Bad: must access in sequence, pointer manipulation can be tricky.

• Both used to represent a sequence of things, but with different syntax.
• Java library has types to represent collections of objects, including
  – Lists (sequences) of objects: ArrayList, LinkedList.
  – Sets of objects: TreeSet, HashSet.
  – Maps (dictionaries): TreeMap, HashMap.

• Names of these classes reflect implementations, but they “publicize” very similar interfaces to the outside.

• Thus, easy to change from using ArrayList to LinkedList.
List classes `ArrayList` and `LinkedList` both share many public methods, including:

- `size()`, `isEmpty()`: Number of items, test for 0 items.
- `get(k)`: Get item #k
- `remove(k)`: Remove item #k.
- `clear()`: Make the list empty.
- `set(k, x)`: Set item #k to x.
- `add(x)`, `add(k, x)`: Add item to end, or a position k.
- `contains(x)`: True *iff* there is an item that equals x (according to `.equals` method).
- `indexOf(x)`: Gives the position of the first item that `.equals` x, or -1 if there is none.

- Both expand as needed (automatically).
- A few methods specialized to one or the other (e.g. `LinkedList.removeFirst()`).
Example

```java
static void readAndReverse (Scanner input, PrintStream output) {
    ArrayList<String> L = new ArrayList<String>();
    while (input.hasNext ())
        L.add (input.next ()�;
    for (k = L.size ()-1; k >= 0; k -= 1)
        output.printf ("%s ", L.get (k));
}
```

• What if we used a LinkedList<String>? What problem might there be with that?
Iterators

• Problem:
  – Indexing as for arrays (via .get) not always best (fastest) way to get items.
  – But would like to use same interface (same methods, same text) for ArrayList and LinkedList.

• Solution
  – the library has class called Iterator, which acts like a “moving finger” through a collection of objects.

```java
static void prListNode (ArrayList<String> L) {
    System.out.printf ("{%n");
    for (Iterator<String> place = L.iterator ();
         place.hasNext (); )
        System.out.printf (" %s%n", place.next ());
}
```
List Manipulation

• Often you need to manipulate all or a subset of the objects in a list in the same way.
  – Incrementing all the integers in the list by the same number
  – Delete any object in the list that satisfies a condition.

• There are two types:
  – Destructive: the modifications are made on the original list.
  – Non-Destructive: the original list is kept intact, a new list with the modifications applied to it are returned.

• Two implementation “flavors”:
  – Recursive
  – Iterative
Destructive Incrementing

public class List {
    public void dincrList(int n) {
        dincrList(head, n);
    }
    /*recursive version*/
    static void dincrList (ListNode P, int n) {
        if (P == null)
            return;
        else {
            P.item += n;
            dincrList (P.next, n);
        }
    }
    /*iterative version*/
    static void dincrList (ListNode L, int n) {
        for (ListNode p = L; p != null; p = p.next)
            p.item += n;
    }
}
Recursive Non-Destructive Deletion

• If L is the list [2, 1, 2, 9, 2], we want removeAll(L,2) to be the new list [1, 9].

/** The list resulting from removing all instances of X from L
 * non-destructively. */

static ListNode removeAll (ListNode L, int x) {
    if (L == null)
        //return null with all x’s removed
        return null;
    else if (L.item == x)
        //L with all x’s removed L!=null
        return removeAll (L.next, x);
    else
        //L with all x’s removed, L!=null && L.item !=x
        return new ListNode (L.item, removeAll (L.next, x));
}
Iterative Non-destructive Deletion

static ListNode removeAll (ListNode L, int x) {
    ListNode result, last;
    result = last = null;
    for ( ; L != null; L = L.next) {
        /* L != null and result is all the elements in the
        list not equal to x from the beginning to L (not
        including L) */
        if (x == L.item)
            continue;
        else if (last == null)
            result = last = new ListNode (L.item, null);
        else
            last = last.next = new ListNode (L.item, null);
    }
    return result;
}
Recursive Destructive Deletion

/** The list resulting from removing all instances of X from L.
 * The original list may be destroyed. */
static ListNode dremoveAll (ListNode L, int x) {
    if (L == null)
        return null;
    else if (L.item == x)
        return dremoveAll (L.next, x);
    else {
        L.next = dremoveAll (L.next, x);
        return L;
    }
}
Iterative Destruction Deletion

/** The list resulting from removing all instances of X from L.
 * Original contents of L may be destroyed. */
static ListNode dremoveAll (ListNode, int x) {
    ListNode result, last;
    result = last = null;
    while (L != null) {
        ListNode next = L.next;
        if (x != L.item) {
            if (last == null)
                result = last = L;
            else
                last = last.next = L;
        }
        L.next = null;
        L = next;
    }
    return result;
}
Invariants

• Try to give a description of how things look on any arbitrary iteration of the loop.
  – This description is known as a loop invariant, because it is true from one iteration to the next.
• The loop body then must
  – Start from any situation consistent with the invariant;
  – Make progress in such a way as to make the invariant true again.

```java
while (condition) {
    // Invariant true here
    loop body
    // Invariant again true here
}
// Invariant true and condition false.
```

• So if (invariant and not condition ) is enough to insure we’ve got the answer, we’re done!
Next Class

• Inheritance

• Due:
  – Wed: Lab assignment “day2”