

Data Structures G5029

Lecture 4

Kingsley Sage
Room 5C16, Pevensey III
khs20@sussex.ac.uk

© University of Sussex 2006

Lecture 4

- Enumerations and iterators.
 - Not data structures, but useful Java concepts that will enable us to implement data structures efficiently.
- More about linked lists
 - Building on last lecture's material.

Enumerations

- It is frequently useful to be able to extract the elements of a vector, list, tree graph etc. one at a time and do something useful with them.
- There is a standard way of doing this in Java which makes use of the `Enumeration` interface in the `java.util` package.

```
public interface Enumeration {
    boolean hasMoreElements();
    Object nextElement();
}
```

Enumerations

- The `hasMoreElements` method returns `true` or `false` depending on whether the enumeration is exhausted.
- The `nextElement` method returns the next element in the enumeration, if there is a next element, so that successive calls to this method enumerates the elements.
- Any implementation of the `Enumeration` interface must implement these two methods.

```
// This code will enumerate the elements of a vector v ..
Enumeration e = v.elements();
while (e.hasMoreElements()) {
    System.out.println(e.nextElement());
}
```

Enumerations

- The enumeration `e` is tied to a particular vector `v` when it is initialised – there is no need for any further reference to the vector `v`.
- You cannot change direction and enumerate backwards at some point.
- Nor can you make any changes in a collection by means of the enumeration methods.

```
// This code will enumerate the elements of a vector v ..  
  
Enumeration e = v.elements();  
while (e.hasMoreElements()) {  
    System.out.println(e.nextElement());  
}
```

Iterators

- Current versions of Java provide an extended concept of enumeration through the `Iterator` interface.
- The additional functionality is that an iterator allows the caller to remove elements from the underlying collection during the iteration.

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```

- First two methods are equivalent to `hasMoreElements` and `nextElement` for `Enumeration`. `remove` method is new.

Iterators – remove method

- `remove` removes from the collection the element returned by the last call to `next`. So, for example, the following removes all elements of the vector `v`.

```
Iterator i=v.iterator();
while (i.hasNext()) {
    i.next();
    i.remove();
}
```

- `remove` may only be called once per call to `next`. An implementation must ensure that an `IllegalStateException` is thrown if either the `next` method has not yet been called, or `remove` has already been called since the last `next`.
- Behaviour of the iterator is unspecified if the collection is modified whilst the iteration is in progress in any way other than by calling `remove`.

Linked lists

- We saw last time the basic concept of the linked list and how that could be used to implement stacks (implementation for queues was left for your own study – see the on-line course notes).
- Now we go on to develop a Java class for a general purpose singly linked list. You can also have doubly linked lists, but we won't cover these here (but what extra properties might they have?).
- What methods do we need to implement ?

Linked lists

```
// For singly linked lists ...

public class LinkedList {
    public boolean isEmpty();
    public int size();
    public void addFirst(Object item);
    public Object firstItem();
    public void removeFirst();
    public boolean contains(Object item);
    public void reverse();
    public Iterator iterator();
    public boolean equals(Object other);
    public String toString();
}
```

Linked lists

- A list will be represented internally by a single `ListNode` constituting the head of the list.
- We set this value to `null` to indicate an empty list.

```
class ListNode {
    Object data;
    ListNode next;

    ListNode(Object data, ListNode next) {
        this.data = data;
        this.next = next;
    }
}

// this operator is self referential ...
```

Linked lists - constructor

- Calling the `LinkedList` constructor creates an empty list represented internally by the `null` reference. Size of list is 0.

```
class LinkedList {
    private ListNode head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }
}
```

Linked lists – insertion & deletion

- The following method inserts an item at the head of the list, whether the list was previously empty or not ...

```
public void addFirst(Object item) {
    head = new ListNode(item, head);
    ++size;
}
```

- The following method removes the item at the head of the list. The element referred to by the previous head's next field becomes the new head of the list ..

```
public void removeFirst() {
    if (head != null) {
        head = head.next; --size;
    } else {
        throw new NoSuchElementException();
    }
}
```

Linked lists – reversal

- How does this work?

```
public void reverse() {
    ListNode current = head;
    head = null;
    while (current != null) {
        ListNode save = current;
        current = current.next;
        save.next = head;
        head = save;
    }
}
```

Linked lists – enumeration

- In order to be able to enumerate a linked list, we need to provide a elements method to collect all the items into a collection:

```
public Enumeration elements(); // add to class def

public Enumeration elements() {
    return new Enumeration() { // anonymous inner class
        private ListNode current = head;
        public boolean hasMoreElements() {
            return (current != null);
        }
        public Object nextElements() {
            if (current != null) {
                Object result = current.data;
                current = current.next;
                return result;
            } else {
                throw new NoSuchElementException();
            }
        }
    };
}
```

Linked lists – testing equality

- Two different ways of looking at equality:
 - The two references are the same physical object
 - There are two separate lists that have the same values
- In the first case, all you would need to do is something like:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

- In the second case, you need to enumerate or iterate through the lists comparing each item pairing one at a time. Can short circuit fail as soon as any pairing does not match.

Linked lists – removing a mid item

- If you need to remove an item from part way through the list, you need to ensure that the list integrity is maintained.
- Necessary to implement `iterator` for linked lists (remember they implement a `remove` method). Full details are in the on-line course notes.



Next time ...

- Order and binary search trees.