2/21/2012

CS/ENGRD 2110
Object-Oriented Programming and Data Structures
Spring 2012
Thorsten Joachims

Lecture 8: Lists

List Overview

- **Purpose**
  - Maintain an ordered set of elements (with possible duplication)

- **Common operations**
  - Create a list
  - Access elements of a list sequentially
  - Insert elements into a list
  - Delete elements from a list

- **Arrays**
  - Random access :)
  - Fixed size: cannot grow or shrink after creation :( 

- **Linked Lists**
  - No random access :
  - Can grow and shrink dynamically :) 

A Simple List Interface

```java
public interface List<T> {
    public void insert(T element); // add to front
    public void delete(T element);
    public boolean contains(T element);
    public int size();
    public String toString();
}
```

Often also:
- Insert at last position, insert at position i
- Get first element, get last element
- Reverse
- Etc.

Generic Types

- "List<E>" is read as "List of E".
- To use the interface `List<E>`, supply an actual type argument, e.g., `List<Integer>`:
  - `List<Integer> list = new LinkedList<Integer>();`
- All occurrences of the formal type parameter `E` in this case are replaced by the actual type argument `Integer` in this case

List Data Structures

- **Array**
  - Must specify array size at creation
  - Insert, delete require moving elements
  - Must copy array to a larger array when it gets full

- **Linked list**
  - uses a sequence of linked cells
  - we will define a class `ListCell` from which we build lists

List Terminology

- **Head** = first element of the list
- **Tail** = rest of the list

```
33 <-> 10 <-> 7 <-> 1 <-> 8 <-> 21
```

head
tail
Class ListCell

class ListCell<T> {
    private T datum;
    private ListCell<T> next;

    public ListCell(T datum, ListCell<T> next) {
        this.datum = datum;
        this.next = next;
    }

    public T getDatum() { return datum; }
    public ListCell<T> getNext() { return next; }
    public void setDatum(T obj) { datum = obj; }
    public void setNext(ListCell<T> c) { next = c; }
}

Building a Linked List

ListCell<Integer> c = new ListCell<Integer>(
    new Integer(24), null);

Integer t = new Integer(24);
Integer s = new Integer(-7);
Integer e = new Integer(87);

ListCell<Integer> p =
    new ListCell<Integer>(t, 
        new ListCell<Integer>(s, 
            new ListCell<Integer>(e, null)));

Building a Linked List (cont’d)

Another way:

Integer t = new Integer(24);
Integer s = new Integer(-7);
Integer e = new Integer(87);
// Can also use "autoboxing"

ListCell<Integer> p
    = new ListCell<Integer>(e, null);
p = new ListCell<Integer>(s, p);
p = new ListCell<Integer>(t, p);

Accessing List Elements

• Linked Lists are sequential-access data structures.
  – To access contents of cell n in sequence, you must access cells 0 ... n-1
• Access
  – Accessing data in first cell: p.getDatum()
  – Accessing data in second cell: p.getNext().getDatum()
  – Accessing next field in second cell: p.getNext().getNext()
• Writing
  – Update data in first cell: p.setDatum(new Integer(53));
  – Update data in second cell: p.setNext().setDatum(new Integer(53));
  – Chop off third cell: p.getNext().setNext(null);

Access Example: Linear Search

// Here is another version. Why does this work?
public static boolean contains(Object x, ListCell c) {
    for (; c != null; c = c.getNext()) {
        if (c.getDatum().equals(x)) return true;
    }
    return false;
}

// Scan list looking for x, return true if found
public static boolean contains(Object x, ListCell c) {
    for (ListCell lc = c; lc != null; lc = lc.getNext()) {
        if (lc.getDatum().equals(x)) return true;
    }
    return false;
}

Recursion on Lists

• Recursion can be done on lists
  – Similar to recursion on integers
• Almost always
  – Base case: empty list
  – Recursive case: Assume you can solve problem on the tail, use that in the solution for the whole list
• Many list operations can be implemented very simply by using this idea
  – Although some are easier to implement using iteration
Recursive Search

- **Base cases**
  - return false
  - if data in first cell equals object x, return true

- **Recursive case**
  - return the result of doing linear search on the tail

```java
public static boolean search(Object x, ListCell c) {
    if (c == null) return false;
    if (c.getDatum().equals(x)) return true;
    return search(x, c.getNext());
}
```

Reversing a List

- **Given a list, create a new list with elements in reverse order**
- **Intuition:** think of reversing a pile of coins

```java
public static ListCell reverse(ListCell c) {
    ListCell rev = null;
    for (; c != null; c = c.getNext()) {
        rev = new ListCell(c.getDatum(), rev);
    }
    return rev;
}
```

List with Header

- **Sometimes it is preferable to have a List class distinct from the ListCell class**
- **The List object is like a head element that always exists even if list itself is empty**
  ```java
class List {
    protected ListCell head;
    public List(ListCell c) {
        head = c;
    }
    public ListCell getHead() {
        return head;
    }
    public void setHead(ListCell c) {
    }
}
```

Recursive Reverse

```java
public static ListCell reverse(ListCell c, ListCell r) {
    if (c == null) return r;
    return reverse(c.getNext(), new ListCell(c.getDatum(), r));
}
```

Variations on List with Header

- **Header can also keep other info**
  - Reference to last cell of list
  - Number of elements in list
  - Search/insertion/deletion as instance methods
  - ...
Special Cases to Worry About

- Empty list
  - add
  - find
  - delete
- Front of list
  - insert
- End of list
  - find
  - delete
- Lists with just one element

Example: Delete from a List

- Delete first occurrence of x from a list
- Intuitive idea of recursive code:
  - If list is empty, return null
  - If datum at head is x, return tail
  - Otherwise, return list consisting of
    - head of list, and
    - list that results from deleting x from the tail

// recursive delete
public static ListCell delete(Object x, ListCell c)
{
  if (c == null) return null;
  if (c.getDatum().equals(x)) return c.getNext();
  c.setNext(delete(x, c.getNext()));
  return c;
}

Iterative Code for Delete

public void delete (Object x) {
  if (head == null) return;
  if (head.getDatum().equals(x)) { //x in first cell?
    head = head.getNext();
    return;
  }
  ListCell current = head;
  ListCell scout = head.getNext();
  while ((scout != null) && !scout.getDatum().equals(x)) {
    current = scout;
    scout = scout.getNext();
  }
  if (scout != null) current.setNext(scout.getNext());
  return;
}

Doubly-Linked Lists

- In some applications, it is convenient to have a `ListCell` that has references to both its predecessor and its successor in the list.

Doubly-Linked vs Singly-Linked

- Advantages of doubly-linked over singly-linked lists
  - some things are easier — e.g., reversing a doubly-linked list can be done simply by swapping the previous and next fields of each cell
  - don't need the scout in iterative delete
- Disadvantages
  - doubly-linked lists require twice as much space
  - insert and delete take more time
Java ArrayList

- “Extensible array”
- Starts with an initial capacity = size of underlying array
- If you try to insert an element beyond the end of the array, it will allocate a new (larger) array, copy everything over invisibly
  - Appears infinitely extensible

- Advantages:
  - random access in constant time
  - dynamically extensible

- Disadvantages:
  - Allocation, copying overhead