Lists

Overview

- Arrays
  - Random access: ☺
  - Fixed size: cannot grow on demand after creation: ☹

- Characteristics of some applications:
  - do not need random access
  - require a data structure that can grow and shrink dynamically to accommodate different amounts of data

⇒ Lists satisfy this requirement.

- Let us study
  - list creation
  - accessing elements in a list
  - inserting elements into a list
  - deleting elements from a list

Lists

- List is a sequence of cells in which each cell contains
  - a data item of type Object
  - a reference to the next cell in the sequence
    - null if this is the last cell in the sequence
    - empty list: null

- List is a sequential-access data structure
  - to access data in position n of sequence, we must access cells 0..n-1

- We will define a class called ListCell from which we will build lists.

Class ListCell

```java
class ListCell {
    protected Object datum;
    protected ListCell next;

    public ListCell(Object o, ListCell n) {
        datum = o;
        next = n;
    }

    public Object getDatum() {//sometimes called car
        return datum;
    }

    public ListCell getNext() {//sometimes called cdr
        return next;
    }

    public void setDatum(Object o) {//sometimes called rplaca
        datum = o;
    }

    public void setNext(ListCell l) {//sometimes called rplacd
        next = l;
    }
}
```

By convention, we will not show the instance methods when drawing cells.
Building a list

ListCell l = new ListCell(new Integer(24), null);

To keep things simple, we will not show Integer objects explicitly in our pictures, but only show the value contained in them.

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

One way:
ListCell p = new ListCell(t, new ListCell(s, new ListCell(e, null)));

Another way:
ListCell p = new ListCell(e, null);

To keep things simple, we will not show Integer objects explicitly in our pictures, but only show the value contained in them.

Accessing list elements

• Lists are sequential-access data structures.
  – to access the contents of cell n in sequence, you must access cells 0..n-1
• 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 to fields in cells can be done the same way
  – p.setDatum(new Integer(53)); //update data field of first cell
  – p.getNext().setDatum(new Integer(53)); //update field of second cell
  – p.getNext().setNext(null); //chop off third cell

Access example: linear search

// scan list looking for object o and return true if found
public static boolean search(Object o, ListCell l) {
  for (ListCell current = l; current != null; current = current.getNext())
    if (current.getDatum().equals(o)) return true;
  // drop out of loop if object not found
  return false;
}

search("dolly", p); // returns true
search("molly", p); // returns false
search("dolly", null); // returns false

// here is another version. Why does this work? Draw stack picture to understand.
public static boolean search(Object o, ListCell l) {
  for (; l != null; l = l.getNext())
    if (l.getDatum().equals(o)) return true;
  // drop out of loop if object not found
  return false;
}
Recursion on lists

- Recursion can be done on lists in a manner similar to recursion on integers.
- Almost always
  - base case: empty list
  - recursive case: assuming you can solve problem on (smaller) list obtained by eliminating first cell, write down solution for list
- Many list problems can be solved very simply by using this idea.
  - Some problems though are easier to solve iteratively.

Recursion example: linear search

- Base case: empty list
  - return false
- Recursive case: non-empty list
  - if data in first cell equals object o, return true
  - else return result of doing linear search on rest of list

```
public static boolean recSearch(Object o, ListCell l) {
    if (l == null) return false;
    else return l.getDatum().equals(o) || recSearch(o, l.getNext());
}
```

Execution of recursive program

Iteration is sometimes better

- Given a list, create a new list with elements in reverse order from input list.

```
//intuition: think of reversing a pile of coins
public static ListCell reverse(ListCell l) {
    ListCell rev = null;  
    for (; l != null; l = l.getNext())
        rev = new ListCell(l.getDatum(), rev);
    return rev;
}
```

- It is not obvious how to write this simply in a recursive divide-and-conquer style.
List with header

- Some authors prefer to have a List class that is distinct from ListCell class.
- List object is like a head element that always exists even if list itself is empty.

class List {
    protected ListCell head;
    public List (ListCell l) {
        head = l;
    }
    public ListCell getHead()
        ...........
        public void setHead (ListCell l)
        ...........
}

Variations of 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

Example of use of List class

- Let us write code to
  - insert object into unsorted list
  - delete the first occurrence of an object in an unsorted list.
- It is just as easy to write code without the header element.
- Methods for insertion/deletion will be instance methods in the List class.
- signatures:
  public void insert(Object o);
  public void delete(Object o);

Insertion into list

- Let us write two insert methods
  - insert at head of list
class List {
    protected ListCell head;
    public List (ListCell l) {
        head = l;
    }
    public ListCell getHead()
        ...........
        public void setHead (ListCell l)
        ...........
}
Example of use of insert methods

List p = new List(null); //create List object with empty list
p.insertHead(new Integer(-7)); //list now contains -7
p.insertHead(new Integer(24)); //list contains 24 and -7
p.insertTail(new Integer(87));

Remove first item from list

//extract first element of list
public Object deleteFirst(){
    //if list is not empty
    if (head != null) {
        Object t = head.getDatum();
        head = head.getNext();
        return t;
    }
    //otherwise, attempt to get from empty list
    else return "error";
}

Remove last item on list

//extract last element of list
public Object deleteLast(){
    if (head == null) return "error";
    //only one element in list?
    if (head.getNext() == null) {
        Object t = head.getDatum();
        head = null;
        return t;
    }
    //at least two elements in list
    //current and scout are cursors into list
    //both advance in lock step, scout is one cell ahead
    //stop if scout points to last cell
    ListCell current = head;
    ListCell scout = head.getNext();
    while (scout.getNext() != null){
        current = scout;
        scout = scout.getNext();
    }
    current.setNext(null); //remove last cell from list
    return scout.getDatum();
}

Delete object from list

• Delete first occurrence of an object o from a list l.
• Intuitive idea of recursive code:
  – If list l is empty, return null.
  – If first element of l is o, return rest of list l.
  – Otherwise, return list consisting of first element of l, and the list that results from deleting o from the rest of list l.
Recursive code for delete

```java
class List{
    protected ListCell head;
    // ...
    public void delete(Object o) {
        head = deleteRec(o, head);
    }
    public static ListCell deleteRec(Object o, ListCell l) {
        // If list is empty, nothing to do
        if (l == null) return null;
        // Otherwise check first element of list
        else if (l.getDatum().equals(o))
            return l.getNext();
        // Otherwise delete o from rest of list and update next field of l
        else {l.setNext(deleteRec(o, l.getNext()));
            return l;
        }
    }
}
```

Iterative delete

- Two steps:
  - locate cell that is the predecessor of cell to be deleted
  - if scout finds cell, update next field of current cell to next field of current cell to splice out object o from list

```
public void delete(Object o) {
    // Empty list?
    if (head == null) return;
    // Is first element equal to o; if so splice first cell out
    if (head.getDatum().equals(o)) {
        head = head.getNext();
        return;
    }
    // Walk down list; at end of loop, 
    // scout will point to first cell containing o, if any
    ListCell current = head;
    ListCell scout = head.getNext();
    while ((scout != null) && ! scout.getDatum().equals(o)) {
        current = scout;
        scout = scout.getNext();
    }
    if (scout != null) // Found occurrence of o
        current.setNext(scout.getNext()); // Splice out cell containing o
}
```

Insertion and deletion into sorted lists

- Assume that we have a list of Comparables sorted in increasing order.
- We want to splice a new Comparable into this list, keeping new list in sorted order as shown in figure.
- Code shows recursive code for insertion and deletion.
- We will show code that uses ListCell class directly.
Recursive insertion

Let us use notation \([f,n]\) to denote ListCell whose
• datum is \(f\)
• next is \(n\)

Pseudo-code:

\[
\text{insert (Comparable } c, \text{ ListCell } l): \\
\text{if } l \text{ is null return new ListCell}(c, \text{null}); \\
\text{else suppose } l \text{ is } [f,n] \\
\text{if } (c < f) \text{ return new ListCell}(c,l); \\
\text{else return new ListCell}(f, \text{insert}(c,n)); \\
\]

Compactly:

\[
\text{insert}(c, \text{null}) = [c, \text{null}] \\
\text{insert}(c,[f,n]) = [c,[f,n]] \quad \text{if } c < f \\
[f, \text{insert}(c,n)] \text{ if } c \geq f \\
\]

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.

```java
public static ListCell insert(Comparable c, ListCell l) {
    if ((l == null) || (c.compareTo(l.getDatum()) < 0))
        return new ListCell(c, l);
    else {
        l.setNext(insert(c, l.getNext()));
        return l;
    }
}

public static ListCell delete(Comparable c, ListCell l) {
    if ((l == null) || (c.compareTo(l.getDatum()) < 0))
        return l;
    if (c.compareTo(l.getDatum()) == 0)
        return l.getNext();//assume no duplicates
    else {l.setNext(delete(c, l.getNext()));
        return l;
    }
}

public static ListCell insertIter(Comparable c, ListCell l) {
    //locate cell that must point to new cell containing c
    //after insertion is done
    ListCell before = scan(c, l);
    if (before == null) return new ListCell(c, l);
    before.setNext(new ListCell(c, before.getNext()));
    return l;
}

protected static ListCell scan(Comparable c, ListCell l) {
    ListCell before = null; //Cursor “before” is one cell behind cursor “l”
    for (; l != null; l = l.getNext())
        if (c.compareTo(l.getDatum()) < 0)  return before;
    else before = l;
    //if we reach here, o is not in list
    return null;
}
```
• In general, it is easier to work with doubly-linked lists than with lists.
• For example, reversing a DLL can be done simply by swapping the previous and next fields of each cell.
• Trade-off: DLLs require more heap space than singly-linked lists.

Summary

• Lists are sequences of ListCell elements
  – recursive data structure
  – grow and shrink on demand
  – not random-access but sequential access data structures
• List operations:
  – create a list
  – access a list and update data
  – change structure of list by inserting/deleting cells
  – cursors
• Recursion makes perfect sense on lists. Usually
  – base case: empty list
  – recursive case: non-empty list
• Sub-species of lists
  – list with header
  – doubly-linked lists