Announcements
• If you’re curious
  • The best bonus-point solution to the stock problem from A1 is posted on the website
  • There is also a separate discussion of the meaning of “choose at random”
• A2 is online (since Friday)  • Due Sept 20

List 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 these requirements
• Let us study
  • List creation
  • Accessing elements in a list
  • Inserting elements into a list
  • Deleting elements from a list

List Operations
• An ADT (Abstract Data Type):
  • Specifies public functionality
  • Hides implementation detail from users
  • Allows us to improve/replace implementation
  • Forces us to think about fundamental operations (i.e., the interface) separately from the implementation
• List Operations:
  • Create
  • Insert object
  • Delete object
  • Find object
  • Size?, Full?, Empty?, Replace Object, …
• A Java interface corresponds nicely to an ADT

A Simple List Interface
public interface List {
  public void insert (Object element);
  public void delete (Object element);
  public boolean contains (Object element);
  public int size ()
}

• Methods are specified, but no implementation

List Data Structures
• Can use an array
  • Need to specify array size
  • Inserts & Deletes require moving elements
  • Must copy array (to a larger array) when it gets full
• Can use a sequence of linked cells
  • We’ll focus on this kind of implementation
  • We define a class ListCell from which we build lists
Class ListCell

class ListCell {
    private Object datum;
    private ListCell next;

    public ListCell(Object d, ListCell n) {
        datum = d;
        next = n;
    }
    public Object getDatum() { return datum; }
    public ListCell getNext() { return next; }
    public void setDatum(Object o) { datum = o; }
    public void setNext(ListCell c) { next = c; }
}

Building a List

ListCell c = 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 to build a list with multiple cells:
ListCell p = new ListCell(t, new ListCell(s, new ListCell(e,null)));

Building a List (Cont’d)

Another way:
Integer t = new Integer(24); // Can also use “autoboxing”
Integer s = new Integer(-7);
Integer e = new Integer(87);
ListCell p = new ListCell(e,null);
p = new ListCell(s,p);
p = new ListCell(t,p);
Note: assignment of form p = new ListCell(s,p); does not create a circular list

Accessing List Elements

- Lists are sequential-access data structures.
- To access 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
  - Update data in first cell: p.setDatum(new Integer(53));
  - Update data in second cell: p.getNext().setDatum(new Integer(53));
  - Chop off third cell: p getNext().setNext(null);

Access Example: Linear Search

// Scan list looking for object o and return true if found:
public static boolean search (Object x, ListCell c) {
    for (ListCell current = c; current != null; current = current.getNext())
        if (current.getDatum().equals(x)) return true;
    return false;
}

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

// Here is another version. Why does this work? Draw stack picture to understand:
public static boolean search (Object x, ListCell c) {
    for (; c != null; c = c.getNext())
        if (c.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 (smaller) list obtained by eliminating first cell...
- Many list operations can be implemented very simply by using this idea
  - Some operations though are easier to implement using iteration
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

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

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 c) {
      ListCell rev = null;
      for (; c != null; c = c.getNext())
          rev = new ListCell(c.getDatum(), rev);
      return rev;
  }
- It is not obvious how to write this simply using a recursive style

List with Header

- Some authors prefer to have a List class that is distinct from ListCell class.
- The List object is like a head element that always exists even if list itself is empty.
  class List {
      protected ListCell head;
      public List (ListCell c) {
          head = c;
      }
      public ListCell getHead()
      public void setHead(ListCell c)
  }

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 object x from list c
- Recursive delete
- Iterative delete
- Intuitive idea of recursive code:
  - If list c is empty, return null
  - If first element of c is x, return rest of list c
  - Otherwise, return list consisting of
    - First element of c, and
    - List that results from deleting x from rest of list c

```java
public static ListCell deleteRecursive (Object x, ListCell c) {
    if (c == null) return null;
    if (c.getDatum().equals(x)) return c.getNext();
    c.setNext(deleteRecursive(x, c.getNext()));
    return c;
}
```

Iterative delete

- Two steps:
  - Locate cell that is the predecessor of cell to be deleted (i.e., the cell containing x)
  - Keep two cursors, scout and current
  - Scout is always one cell ahead of current
  - Stop when scout finds cell containing x, or falls off end of list
- If scout finds cell, update next field of current cell to splice out object x from list
- Note: Need special case for x in first cell

```java
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.

```java
class DLLCell {
    private Object datum;
    private DLLCell next;
    private DLLCell previous;
    ......
}
```

Doubly-Linked vs. Singly-Linked

- In some cases it is easier to work with doubly-linked lists than with (singly-linked) 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

Tree Overview

- Tree recursive data structure (similar to list)
  - Each cell may have two or more successors (children)
  - Each cell has at most one predecessor (parent)
  - Distinguished cell called root has no parent
  - All cells are reachable from root
- Binary tree: tree in which each cell can have at most two children: a left child and a right child