(More) Recursion and Lists

Lecture 4
CS211 – Summer 2008

Recursion examples: Problem 1

- Recursive function to reverse a string

Recursion examples: Problem 1

- Another solution

```
public String revStr_helper(String str, String sofar) {
  if(str.equals("")) return sofar;
  else
    return revStr_helper(str.substring(1, str.length()),
                         str.charAt(0) + sofar);
}

public String reverseString(String str) {
  return revStr_helper(str, "");
}
```


Tail recursion

- In a *tail recursive* method, the recursive call is the last command executed by the method
  - Tail recursive example:

```
return revStr_helper(str.substring(1, str.length()),
                     str.charAt(0) + sofar);
```

  - Non-tail recursive:

```
return revString(str.substring(1, str.length())) +
       str.charAt(0);
```

- Tail recursive methods can be executed more efficiently
  - No need to maintain a stack!

Recursion examples: Problem 2

- Recursive method to compute GCD of two integers

Recursion examples: Problem 3

- Method that outputs binary representation of integer

Recursion examples: Problem 4

- Merge two sorted strings of characters

Possible Quiz #1 Solution

```
// pre: n >= 0
static void printBinary(int n) {
  // base case: n can be represented in 1 bit
  if (n < 2)
    System.out.print(n);
  else {
    // general case: print everything except last bit
    // and then print last bit
    printBinary(n/2);
    System.out.print(n%2);
  }
}
```

Possible Quiz #1 Solution - Variation

```
// pre: n >= 0
static void printBinary(int n) {
  // base case: n can be represented in 1 bit
  if (n < 2)
    System.out.print(n);
  else {
    printBinary(n/2);
    printBinary(n%2);
  }
}
```


Possible Quiz #1 Solution – Simpler?

```
// pre: n >= 0
static void printBinary(int n) {
  if (n >= 2)
    printBinary(n / 2);

  System.out.print(n % 2);
}
```


List Overview

- Arrays
  - Random access (good)
  - Fixed size: cannot grow on demand after creation (bad)
- But 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
- Common operations
  - 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
  - Replace object
  - Size? Empty?
  - Usually sequential access (not random access)
- A Java *interface* corresponds nicely to an ADT

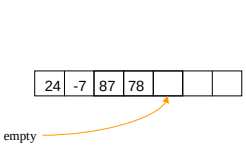
A Simple List Interface

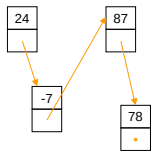
```
public interface List {
    public void insert(Object element);
    public void delete(Object element);
    public boolean contains(Object element);
    public int size();
}
```

- Methods are specified, but *no* implementation

List Data Structures

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

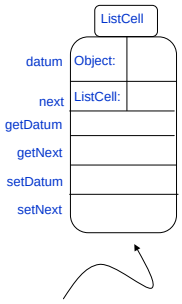




Class ListCell

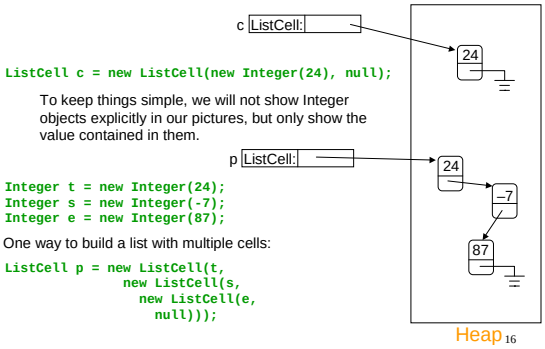
```
class ListCell {
   private Object datum;
   private ListCell next;

   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;}
}
```



By convention, we will not show the instance methods when drawing cells

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);
```

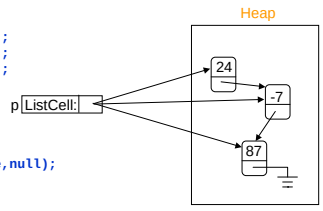
One way to build a list with multiple cells:

```
ListCell p = new ListCell(t,
               new ListCell(s,
                 new ListCell(e,
                   null)));
```


Building a List (cont'd)

Another way:

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

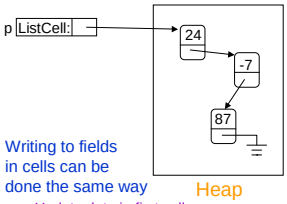



```
ListCell p = new ListCell(e,null);
p = new ListCell(s,p);
p = new ListCell(t,p);
```

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

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


A Recursive Version

```
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());
}
```

```
public static boolean search(Object x, ListCell c) {
   return c != null &&
      (c.getDatum().equals(x) || search(x, c.getNext()));
}
```


Recursion on Lists

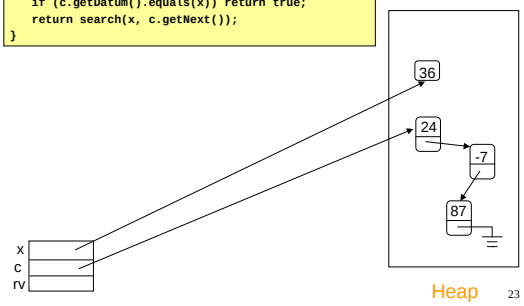
- Recursion can be done on lists
  - Similar to recursion on integers
- Almost always
  - Base case: empty list
  - Recursive case: Assume you can solve problem on (smaller) list obtained by eliminating first cell…
- Many list operations can be implemented very simply by using this idea
  - Although some operations are easier to implement using iteration

Recursive Search

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

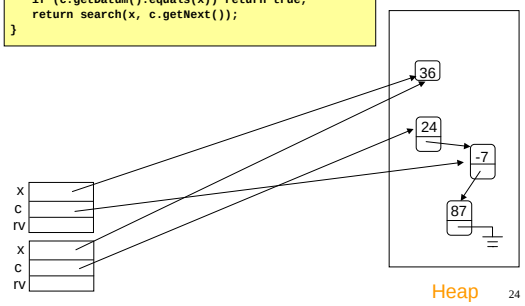
Execution of Recursive Program

```
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());
}
```



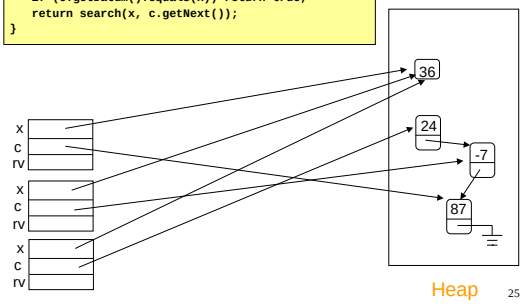
Execution of Recursive Program

```
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());
}
```



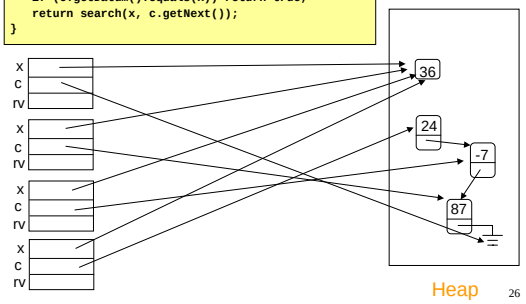
Execution of Recursive Program

```
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());
}
```



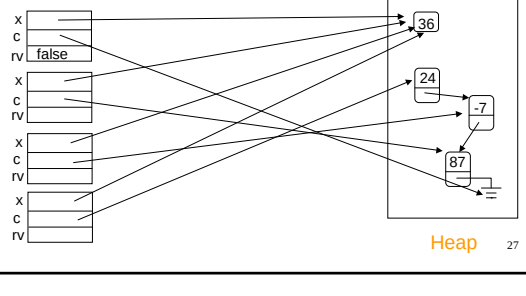
Execution of Recursive Program

```
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());
}
```



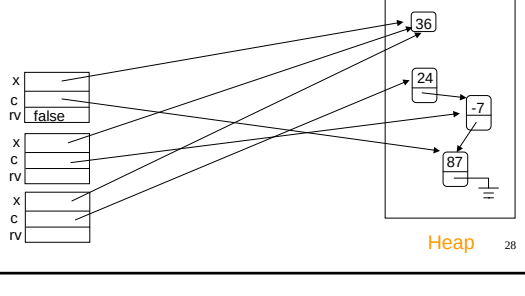
Execution of Recursive Program

```
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());
}
```



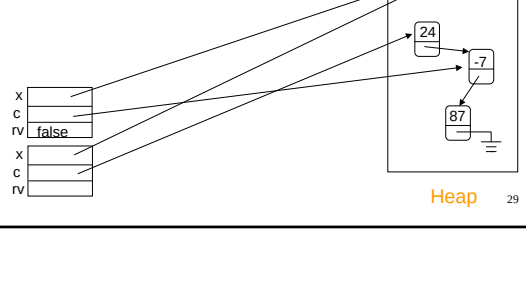
Execution of Recursive Program

```
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());
}
```



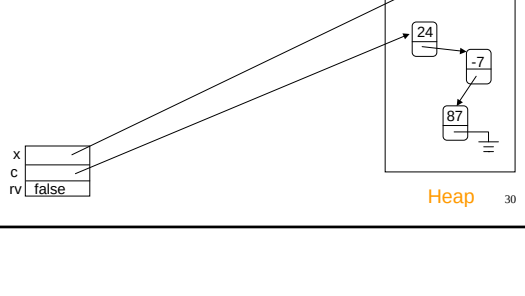
Execution of Recursive Program

```
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());
}
```



Execution of Recursive Program

```
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());
}
```



Iterative example: reversing a list

- Given a list, create a new list with elements in reverse order
- 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;
}
```


Recursive example: Delete from a List

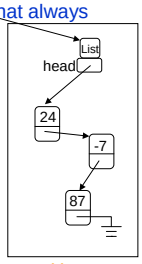
- Delete *first occurrence* of x from list c
  - Recursive delete
  - Iterative delete
- Intuitive idea of recursive code
  - If list is empty, return null
  - If first element of c is x, return rest of list c
  - Otherwise, return list consisting of
    - First element of c, and
    - List that results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

```
//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;
}
```


List with Header

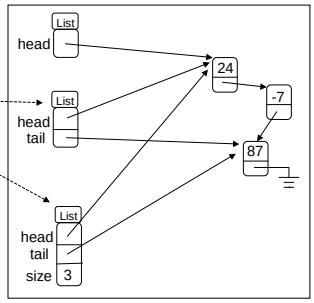
- Some authors prefer to have a List class that is distinct from ListCell class.
- The List object is like a head element that always exists even if list itself is empty.

```
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
  - …



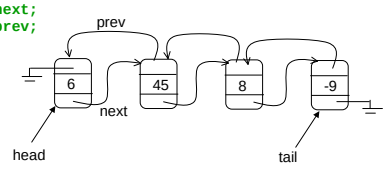
Special Cases to Worry About

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

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.

```
class DLLCell {
   private Object datum;
   private DLLCell next;
   private DLLCell prev;
   …
}
```

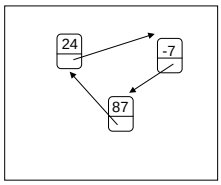


Doubly-Linked vs. Singly-Linked

- Advantages of doubly-linked lists 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
- Disadvantages
  - doubly-linked lists require more heap space than singly-linked lists
  - insert and delete take more time

Circularly Linked Lists

- In other applications, it is convenient if the list forms a cycle, with no beginning or end
  - Entire list can be traversed from any given node



Heap