Lists

Weiss sec. 6.5 (sort of)
ch. 17 (sort of)

Arrays

- Random access

  a[38] gets 39th element

- But fixed size, specified at construction

  e.g. *pickLineFromFile*( ) in Assignment01

- One approach:

  java.util.ArrayList

  – Keep an array of objects as a private field

  – Similar to array of objects, but using get(…) and set(…)

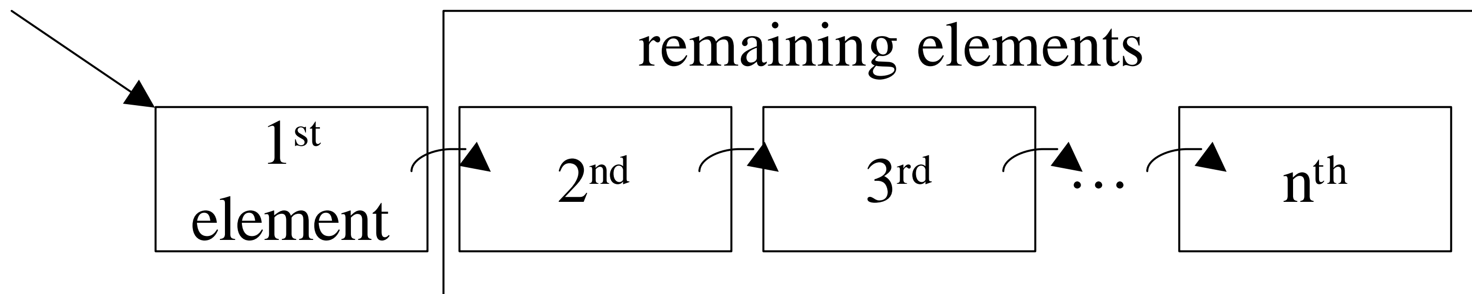
  – Resizes on demand

    - Maintain *size* and *capacity* >= *size*
    - If ever *capacity* < *size*, then double capacity and copy elements

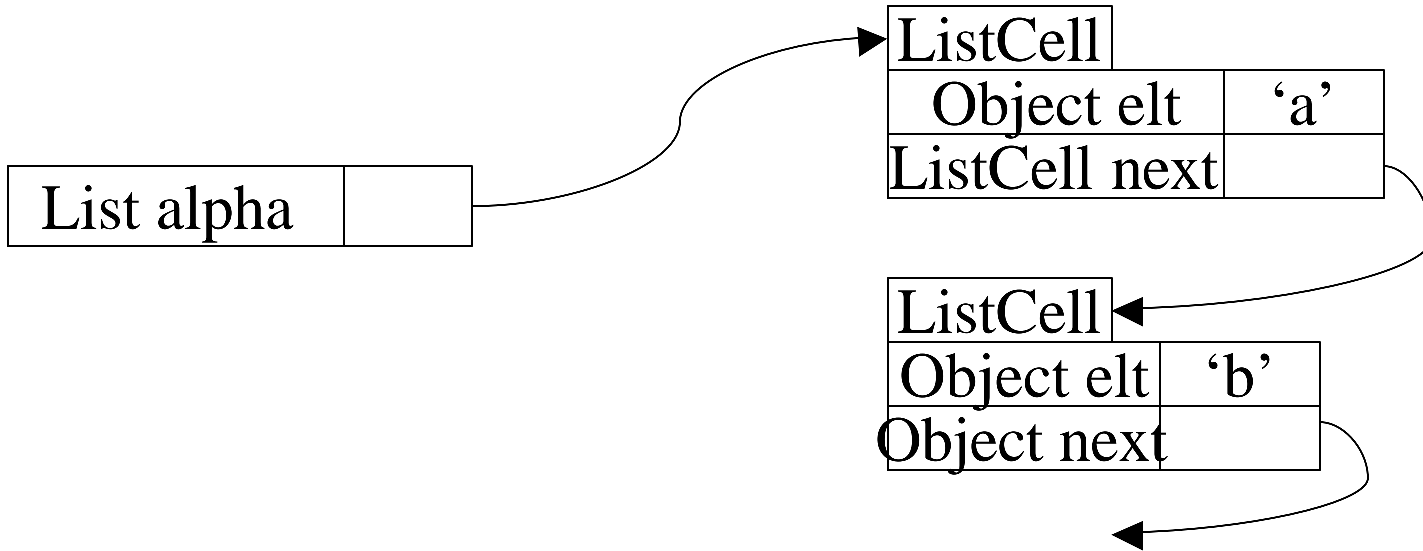
  – Efficient, usually

Lists

- A Different Approach
- Many applications don't need fast random access:
  - grow and shrink on demand
  - fast access only to certain elements (e.g., the first one)
- *Abstract Data Type* List:
  - ordered sequence of "cells", each containing an Object
  - first cell is accessible
  - given any cell, the next cell is accessible (if there is one)
  - Yes, this is smells inductive/recursive

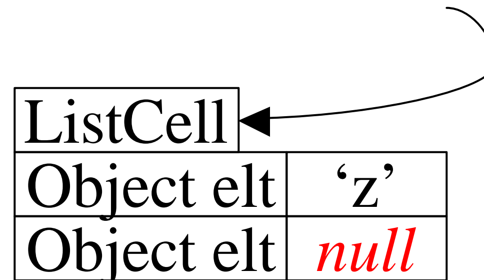


Implementing Using References



Note: member methods not shown

- getElt( ), setElt( )
- getNext( ), setNext( )
- etc.



Code

```
public class ListCell {
    private Object elt;
    private ListCell next;

    public ListCell(Object first, ListCell rest) {
        elt = first;
        next = rest;
    }

    public Object getElt( ) { return elt; }                   // sometimes: car
    public ListCell getNext() { return next; }                // sometimes: cdr

    public void setElt(Object first) { elt = first; }         // sometimes: rplaca
    public void setNext(ListCell rest) { next = rest; } // sometimes: rplacd
}
```

Building a List

```
public static void main(String args[ ]) {
  Integer a = new Integer(25);
  Integer b = new Integer(-9);
  Integer c = new Integer(3);

  ListCell p = new ListCell(a, new ListCell(b, new ListCell(c, null)));
}
```

Resulting data structure?

Building a List: #2

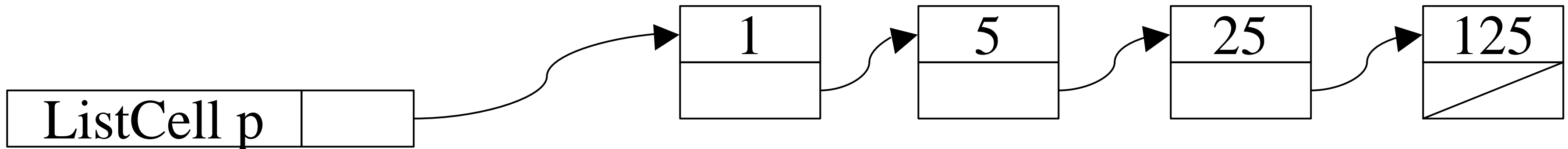
```
public static void main(String args[ ]) {
  Integer a = new Integer(25);
  Integer b = new Integer(-9);
  Integer c = new Integer(3);

  ListCell p;
  p = new ListCell(a, null);
  p = new ListCell(b, p);
  p = new ListCell(c, p);
}
```

Resulting data structure?

Accessing List Elements

```
p; // 1st element
p.getNext(); // 2nd element
p.getNext().getNext(); // 3rd element
p.getNext().getNext().getNext(); // 4th
```

```
p.setElt(8); // set 1st
p.getNext().getElt(); // get 2nd
p.getNext().getNext().setNext(null); // chop
p.getNext().getNext().getNext().getElt();
    // crash: throws NullPointerException
```



- Accessing n^{th} cell requires following n pointers

Linear Search

- Want to see if list contains a particular item
  - compare using *equals*( )

```
static boolean search(ListCell p, Object o) {
  for (ListCell curr = p; curr != null; curr = curr.getNext()) {
    if (curr.getElt().equals(o)) return true;
  }
  return false; // if loop drops out, object not found
}
```

- Q: Why static?

Subtle Version

- Don't need local variable

```
static boolean search(ListCell p, Object o) {
  for ( ; p != null; p = p.getNext()) {
    if (p.getElt().equals(o)) return true;
  }
  return false; // if loop drops out, object not found
}
```

- Q: What happens to the original list *p*?

Recursive Version

- Recursive data structures call for recursive methods
  - sometimes
- We can do recursion/induction on lists, just as for natural numbers. Compare:
  - for (int i = 1; i <= n; i++) …
  - for (ListCell curr = p; cur!=null; curr = curr.getNext( )) …

- base case: solve for an empty list (or 1-element list)
- recursive case: solve for larger list by first removing first element and recursing, then obtaining solution for whole list

Recursive Code

```
static boolean search(ListCell p, Object o) {
    if (p == null) return false;
    else return p.getElt().equals(o) | | search(p.getNext(), o);
}
```

- Note: clever use of ||
  - recursive call to search(…) only happens if first clause evaluates to false

Recursion Not Solution to Everything

- Reversing a list:

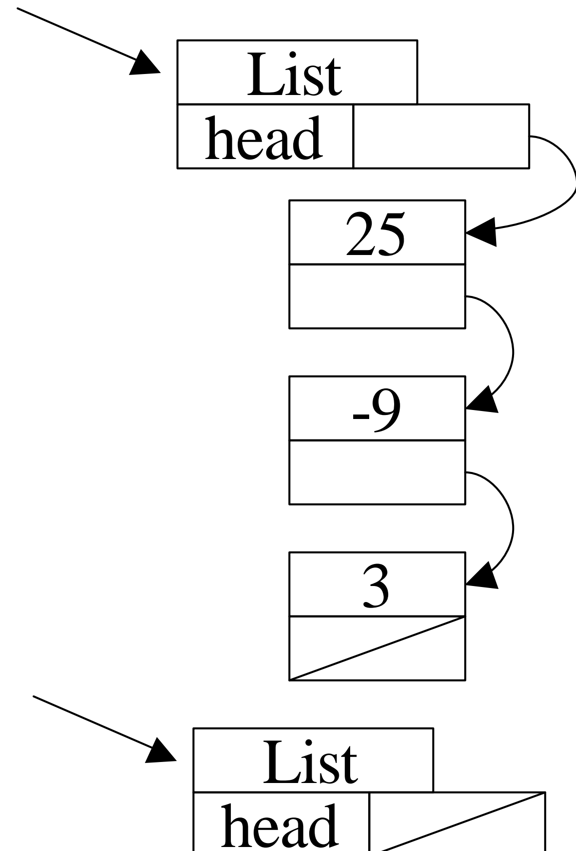
```
static ListCell reverse(ListCell p) {
  ListCell r = null;
  for ( ; p != null; p = p.getNext())
    r = new ListCell(p.getElt(), r);
  return r;
}
```

- Think: reversing a stack of papers

Variations on Lists: Headers

- Adding a header
  - one instance of List, containing many ListCells
  - always exists, even when list is empty

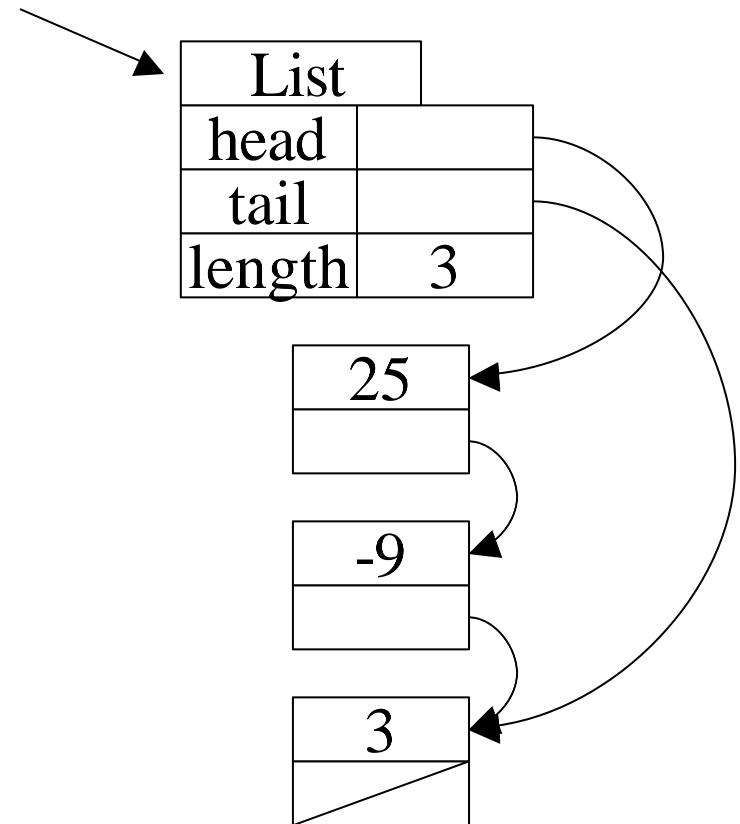
```
public class List {
  private ListCell head;
  public List(ListCell h) { head = h; }
  … getHead … setHead …
}
```

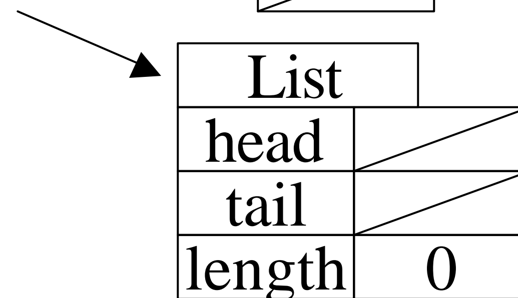


- convenient place for list instance methods:
  - search, insert, delete, etc.

Variations on Lists : Header With Tail

- Store other info in header as well
  - pointer to tail of list
  - length of list

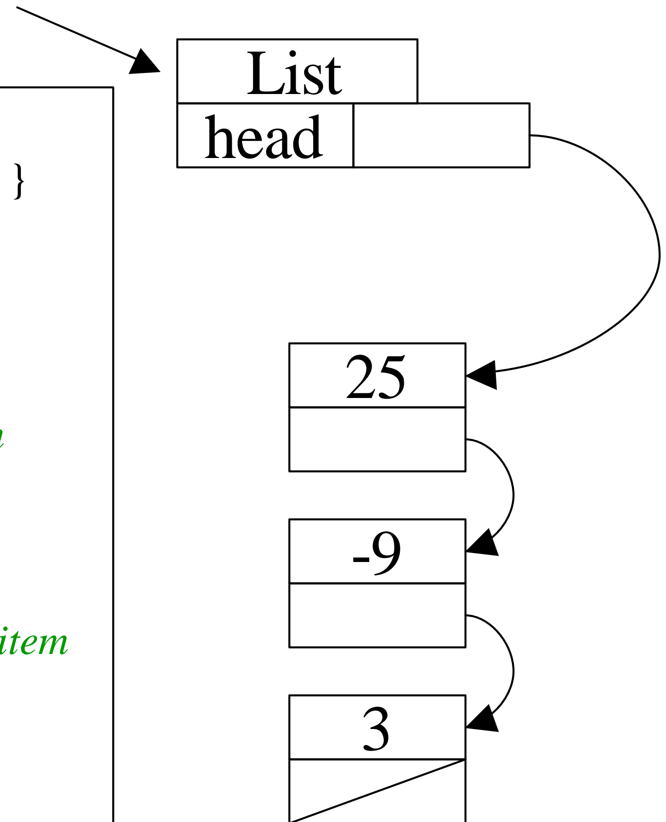




Insertion

- Using header with no tail pointer: others similar
- insertHead(Object o)

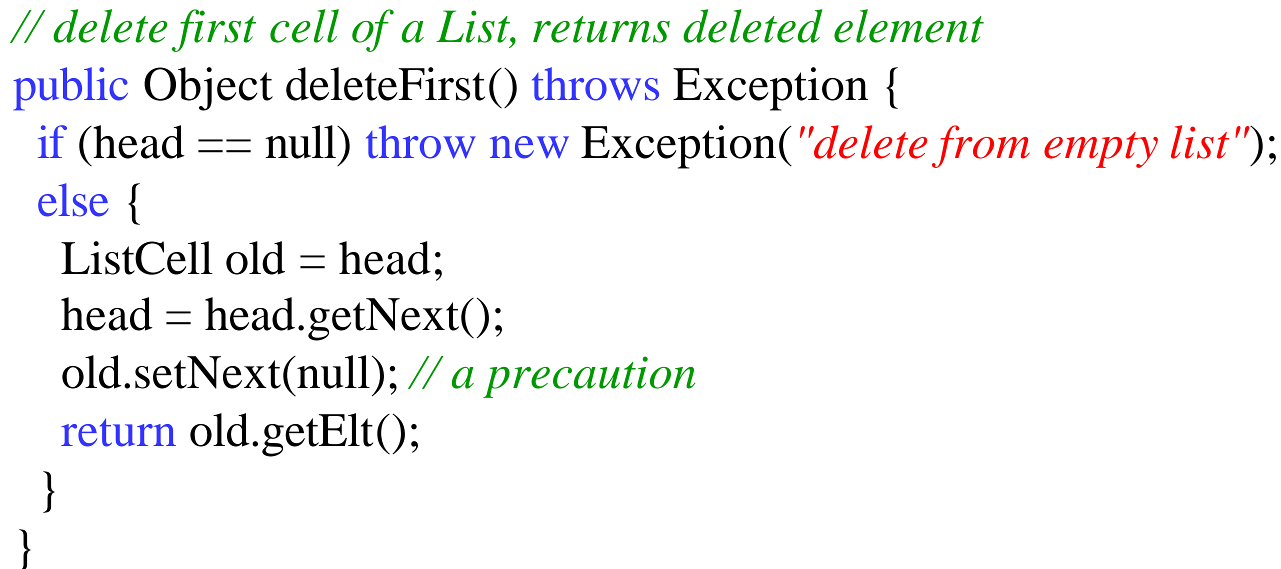
```
public class List {
 public void insertHead(Object o) {  head = new ListCell(o, head);  }

  public void insertTail(Object o) {
    ListCell newcell = new ListCell(o, null); // will be at the tail
    if (head == null) {
      // special case: list was empty; new cell is the first and last item
      head = newcell;
    } else {
      // find tail cell, then add new cell to tail
      // note: this loop has no body; it stops with tail pointing to last item
      ListCell tail;
      for (tail = head; tail.getNext() != null; tail = tail.getNext());
      tail.setNext(newcell);
    }
  }
}
```



ex: new List(null).insertHead("Hello");

Deleting Items: First Cell

- Cut cell out of list, return the element deleted
- One special case, one general case

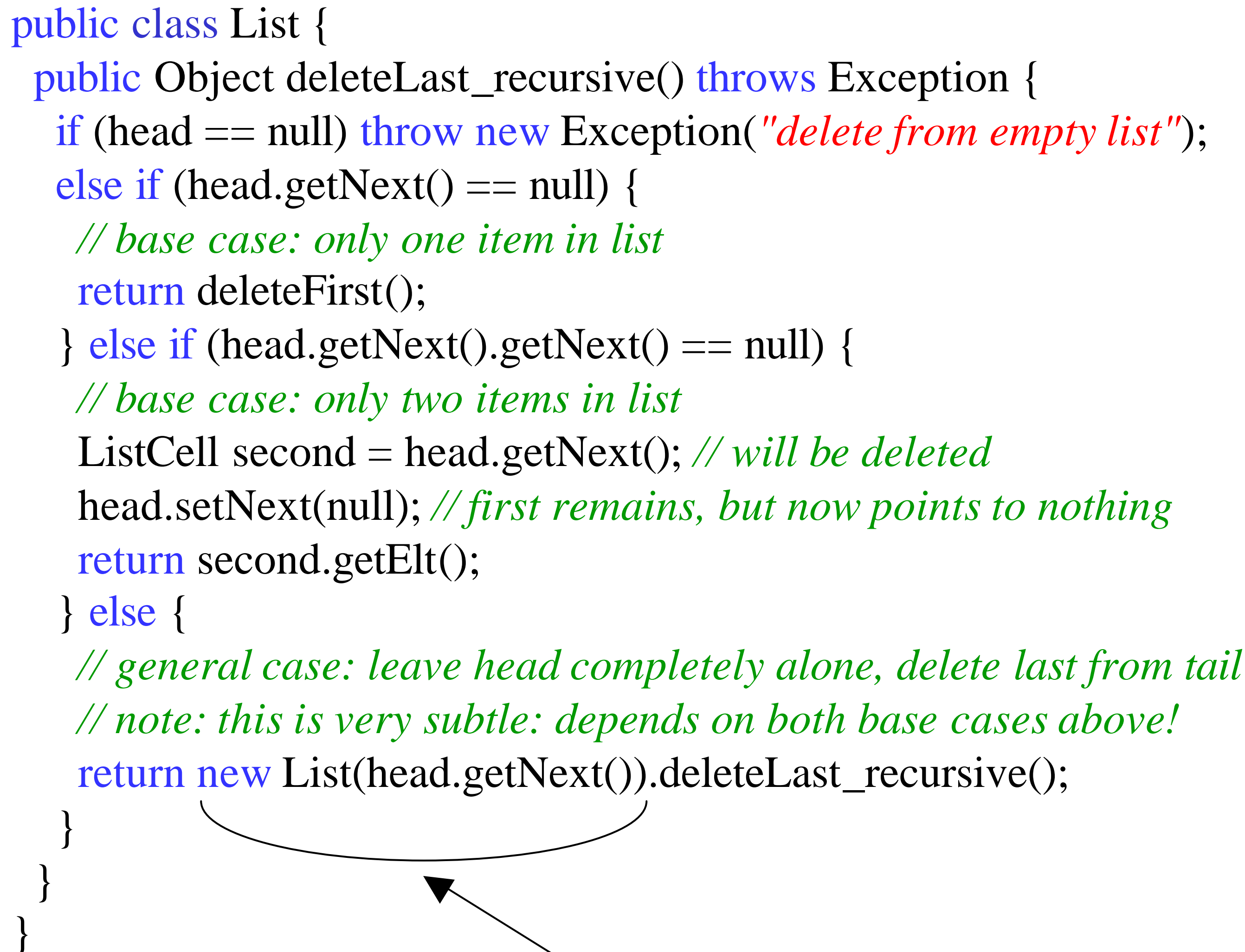



```
// delete first cell of a List, returns deleted element
public Object deleteFirst() throws Exception {
  if (head == null) throw new Exception("delete from empty list");
  else {
    ListCell old = head;
    head = head.getNext();
    old.setNext(null); // a precaution
    return old.getElt();
  }
}
```

Deleting Items: Last Cell

- Two cells may need to be affected:
  - last cell is removed, and returned (as before)
  - next-to-last cell now points to *null*
- Recursive approach
  - base case: empty list → error
  - base case: one element list → same as deleteFirst( )
  - base case: two elements → head points to null
head.next is removed, returned
  - recursive case: many elements → leave head alone
remove last from remainder

Deleting Items: Last Cell



```java
public class List {
  public Object deleteLast_recursive() throws Exception {
    if (head == null) throw new Exception("delete from empty list");
    else if (head.getNext() == null) {
      // base case: only one item in list
      return deleteFirst();
    } else if (head.getNext().getNext() == null) {
      // base case: only two items in list
      ListCell second = head.getNext(); // will be deleted
      head.setNext(null); // first remains, but now points to nothing
      return second.getElt();
    } else {
      // general case: leave head completely alone, delete last from tail
      // note: this is very subtle: depends on both base cases above!
      return new List(head.getNext()).deleteLast_recursive();
    }
  }
}
```

Note: Why do we need “new List(…)”? Does it matter? Very subtle…

Deleting Items: Last Cell (iterative)

- Two cells may need to be affected:
  - last cell is removed, and returned (as before)
  - next-to-last cell now points to *null*
- Iterative approach:
  - Need *two* pointers scanning the list, in lock-step
  - *current* points to an element
  - *scout* points to the element after *current*
  - *current* will become the next-to-last element
  - *scout* will become the last element

Deleting Items: Last Cell (iterative)

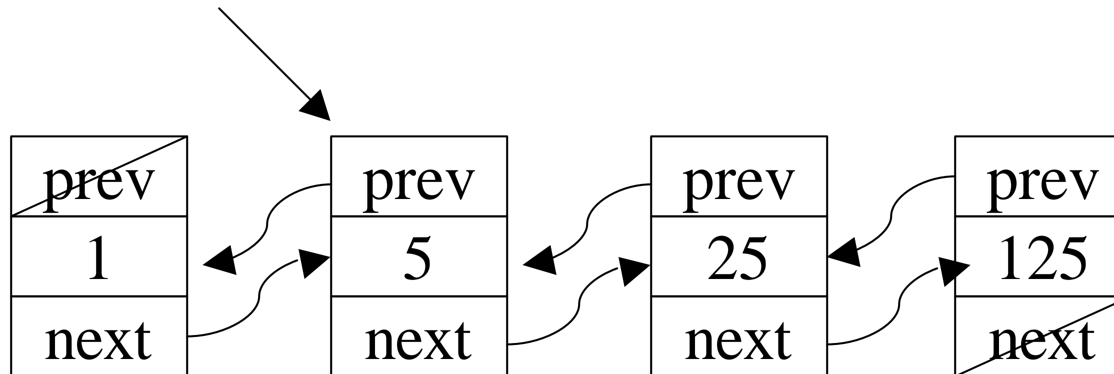
```
public Object deleteLast() throws Exception {
        if (head == null) throw new Exception("delete from empty list");
        else if (head.getNext() == null) {
          // only one item
          ListCell oldcell = head;
          head = null;
          return oldcell.getElt();
        } else {
          // general case: find last element (removed from list), and
          // also next-to-last element (update to point to nothing)
          ListCell current = head; // will be next-to-last element
          ListCell scout = current.getNext(); // always one ahead of current
          while (scout.getNext() != null) {
            current = current.getNext();
            scout = current.getNext(); // always one ahead
          }
          // loop ends when scout points at last cell
          current.setNext(null); // current becomes the new last element
          return scout.getElt();
        }
      }
```

Insert and Delete in the Middle

- Insert *c* just after cell *p*: "splicing in"
  - *c* now points to *p.next*
  - *p* now points to *c*
- Delete *c*, which is just after cell *p:* "splicing out"
  - *p* now points to *c.next*
  - *c* now points to *null*
  - will need two "cursors" as before, scanning in lock-step
- See code on web site

Lists

- So far: *singly-linked lists*
- Could also implement *doubly-linked lists*
  - Each cell maintains *next* and *prev* pointers
  - Many methods become much easier, faster, simpler
  - But… more heap space required



Moral

- Lists are not terribly complex
- But… lots of room for mistakes
  - manipulating *head, next* and *prev* references is error-prone
  - a source of many, many bugs in student programs
- Which is why…
  - we implement a good List class once
  - …and never again
- In practice (but not in school!):
  Strive to never implement something that was already implemented (by someone at least as smart as you).