CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #10

John Ridgway

March 3, 2015

Review of the Grid Class

- DJW present a class called `Grid` in Section 4.4. A Grid object has a two-dimensional array of booleans, called `grid`.
- We, though not DJW, call a pair (i, j) with `grid[i][j]` true a land square, and one with `grid[i][j]` false a water square.
- They have a constructor to set the grid values randomly, with the average percentage of land squares as a parameter.

Continents (Blobs)

- A continent (or blob for DJW) is a set of squares such that from one of them, you can get to any of the others by an NESW path on land squares, but you cannot get to any land square off the continent.
- DJW present a method `markBlob` that marks all the squares on a continent, and a method `countBlobs` to determine how many continents are in a given grid.

Recursive Marking

- The `markBlob` method operates recursively – it can be summarized as mark the target square, then call `markBlob` on any unseen land neighbors it has.
- This *implicitly* manages the search with a stack, specifically the **method stack** in which the interpreter stores records of the recursive calls.
- The search is *depth-first* because, for example, it explores all neighbors of the first neighbor before any of the second neighbor.

Recursion on Linked Structures

- We can define a linked list recursively once we have the definition of the `LLNode<T>` class.
- A linked list of `T` objects is either empty (when its head pointer is `null`) or consists of an `LLNode` whose `info` field contains a `T` object and whose `next` field is the head of a linked list.
- Every valid linked list is formed by these two rules.

Recursion on Linked Structures

- If we can define what we want to do recursively, we can easily write recursive code.
- To add an element to the tail of the list, there are two cases. If the list is now empty, we make a new node and set `head` to it.
- If there is a first node in the list, we can remove it (keeping a pointer to it), add our new element to the tail of the remaining list, and put the first node back.

Code for addToTail

```
public void addToTail(T elem) {
// add node containing elem to tail of list
   if (head == null) {
      LLNode<T> newNode = new LLNode<T>(elem);
      head = newNode;}
   else { LLNode<T> temp = head;
      head = head.getLink();
      addToTail(elem);
      temp.setNext(head);
      head = temp; }}
```

Our base case is an empty list, we make progress by the list getting smaller with each call, and if the recursive call works, we are sure of our removal and replacement of the head.

Clicker Question #1

`LLNode<T> curr = head; // code from below goes here` Which two lines of code leave `curr` pointing to the last node in the list, *assuming there is at least one node in the list*?

A.
```
while (curr != null)
    curr = curr.getNext();
```

B.
```
while (curr != null)
    curr.setNext(curr.getNext());
```

C.
```
while (curr != null)
    curr.setNext(curr.getNext());
```

D.
```
while (curr.getNext() != null)
    curr.getNext();
```

Recursion at Machine Level

- Weve mentioned before how method calls are handled at the machine level.
- When method A calls method B, the execution of A is suspended while B runs, and the machine stores an *activation record* that will allow it to restore As context when B finishes.
- If B then calls C, the activation record for B goes on top of As on the *call stack*.

Recursion at Machine Level (continued)

- Its no different when a method calls itself, directly or indirectly.
- We get multiple activation records on the call stack for the same method. Note that each of these is independent, with, for example, separate copies of each local variable.
- Running our `factorial(n)`, for example, eventually produces a stack of `n` records.

Reversing a Linked List

Consider the problem of replacing a linked list with another, containing the same elements in the opposite order. This can be done recursively:

```
public void reverse() {
   if (head == null) return;
   if (head.getNext() == null) return;
   LLNode<T> temp = head;
   head = head.getNext();
   temp.setNext(null);
   reverse();
   addToTail(temp.getContents()); }
```

Reversing a Linked List (continued)

- The base case is a list of size 0 or 1.
- We make progress because each recursive call is to a list that is smaller by one.
- If the recursive call works, we remove the head element, reverse the rest of the list, and put the old head back at the tail.

Clicker Question #2

What happens if we switch the fifth and sixth lines of this method, to get the code below?

```
public void reverse() {
   if (head == null) return;
   if (head.getNext() == null) return;
   LLNode<T> temp = head;
   temp.setNext(null);
   head = head.getNext();
   reverse();
   addToTail(temp.getContents()); }
```

A. nothing changes

B. lists longer than one element are truncated

C. it won't compile

D. the list stays in its original order

Printing a List Backwards

- DJW give a different example of a recursive procedure on a linked list, which they put in their `LinkedStack` class. It prints out the contents of each node to `System.out`, but in reverse order, with the tail element first and the head element last.
- The idea is simple given the recursive definition. If the list is empty we have nothing to do. If it is not, we first print the list starting with `head.link`, then print out the contents of the head node.

Printing a List Backwards (continued)

As is common in such cases, they use a helper method. The general method takes any node pointer as argument, then the specific method calls the general one on the head pointer of the list. Note this takes $O(n)$ time on a list of length n.

```
private void revPrint(LLNode<T> listRef) {
  if (listRef != null) { revPrint(listRef.getLink());
    System.out.println(" " + listRef.getInfo()); }}

public void revPrint(){ revPrint(top); }
```