More List Processing

Lecture 8

- In this lecture we look at various operations that can be carried out on lists, and how they're implemented.
- In particular, we'll look at,
  - appending two lists
  - reversing lists, and
  - accumulating parameters.

Appending Lists Using a Loop

- This function appends two lists together.
- First it builds up a reversed copy of `list1`.
- Then it runs along that copy adding each element to `list2`.
- Note that it uses two access functions, and doesn't manipulate pointers directly.

```
NodePtr appendLists1 (NodePtr list1,
                      NodePtr list2) {
    NodePtr tempList;
    tempList = NULL;
    while (list1 != NULL) {
        addToList( firstFromList(list1), tempList);
        list1 = restOfList(list1);
    }
    while (tempList != NULL) {
        addToList( firstFromList(tempList), list2);
        tempList = restOfList(tempList);
    }
    return list2;
}
```

```
Item firstFromList (NodePtr hdList) {
    return Item(hdList->data);
}

NodePtr restOfList (NodePtr hdList) {
    return NodePtr(hdList->next);
}
```


Appending Lists Using Recursion

- The function below uses recursion instead of a loop to append two lists together.
- It also uses the existing access procedures, and doesn't manipulate pointers directly.

```
NodePtr appendLists2 (NodePtr list1,
                      NodePtr list2) {
    NodePtr tempList;
    if (list1 == NULL)
        return list2;
    else {
        tempList = appendLists2(restOfList(list1), list2)
        addToList(firstFromList(list1), tempList);
        return tempList;
    }
}
```

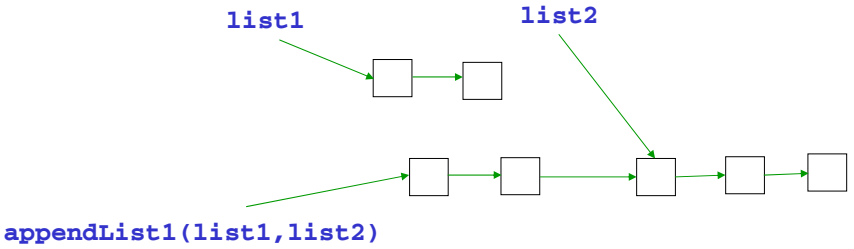

Appending Lists via Pointers

- Here's another function that appends two lists of names together using pointers instead of access routines.
- It works by
  - Finding the end of the first list, then
  - Attaching the second list onto it.

```
NodePtr appendLists3 (NodePtr list1,
                      NodePtr list2) {
    NodePtr tempList;
    if (list1 == NULL)
        return list2;
    else {
        tempList = list1;
        while (tempList->next != NULL)
            tempList = tempList->next;
        tempList->next = list2;
        return list1;
    }
}
```

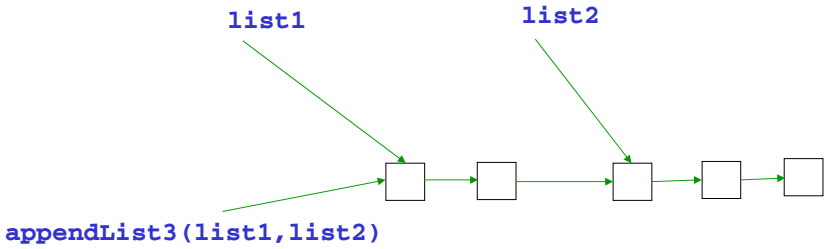
The Append Functions Compared I

- The three append functions differ not only in their style of implementation, but also in the way they use the heap.
- Both `appendList1()` creates a new copy of list1. So the appended list comprises a new copy of `list1` followed by the old copy of `list2`.
- In addition, `appendList1()` creates a duplicate temporary list `tempList`. which is merged with `list2`. At the end of the function `list1` is redundant and should be destroyed (i.e. returned to the heap) in order to save memory.



The Append Functions Compared II

- `appendList2()` function uses recursion and pushes onto the stack the entire point links of `list1`. These are then linked into `list2` one by one!
- The `appendList3()` function, on the other hand, doesn't create anything new data . But Names1 is swallowed up by the appended list.
- We can use any of these versions, so long as we're aware how they behave.



Reversing a List Using a Loop

- Here's a simple procedure that reverses a list, using a loop.

```
NodePtr reverseList1(NodePtr list) {
    NodePtr tempList = NULL;
    while (list!=NULL) {
        addToList(firstFromList(list), tempList);
        list = restOfList(list);
    }
    return tempList;
}
```

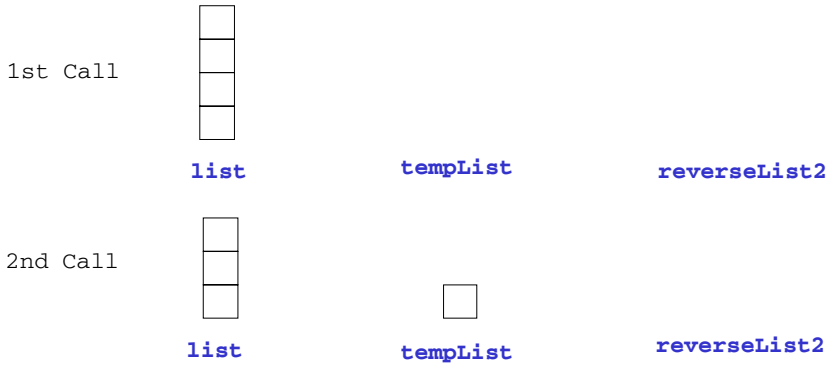

Reversing a List with Recursion

- Here is how reversing a list can be done with recursion using something called an **accumulating parameter**. The second parameter `tempList`, which is set to `NULL` by the calling routine, accumulates the final result.
- When the base case is reached, that result is returned, and it then floats back up through all the procedure exits to become the final result.

```
NodePtr reverseList2(NodePtr list,
                     NodePtr tempList) {
    if (list==NULL)
        return tempList;
    else {
        addToList(firstFromList(list), tempList);
        return reverseList2(restOfList(list), tempList);
    }
}
```

How Accumulating Parameters Work I

- Here's how **tempList** accumulates the result of the **reverseList2()** function.
- Each list is represented as a vertical pile of elements, to emphasise how the old list is gradually transfered into the accumualting parameter.
- The next slide shows how the computation winds up.



How Accumulating Parameters Work II