Linked Lists

2-17-2005

Opening Discussion

- What did we talk about last class? Do you have any code to show?
- Do you have any questions about the assignment?
- Can you tell me what a linked list is and what is significant about them?

Growing the Stacks and Queues

- Let’s go back to our code and look at what we have to do to make it so the arrays can grow on overflow.

The List as an ADT

- A more powerful abstract data type than the stack or queue is the list. Lists allow random access of elements so that you can add, search, and remove at will. As with the stack and the queue, the implementation can vary.
- Java has an interface for List in the java.util package that you can look at to get an idea of what a list should do.

Array Based Lists

- Just like with the stack and the queue, we can implement lists using arrays. That implementation though has some drawbacks to it.
- The main problem is that random insertions and removing require lots of copying, though we can jump to random elements quite quickly.

Linked Lists

- An alternate implementation of a list is using what are called Linked Lists. A linked list is a list where each element knows only about its neighbors.
- The simplest form of this is a singly linked list where each element knows about the next element in the list. If you keep track of the first one you can get to the entire list by following the links.

Nodes

- Each element of the list is typically called a node. The node "stores" the data that we need as well as the references/pointers to the other node(s) that it is linked to.
- The node class is not the linked list class. They are distinct types, though the linked list class will use the node class. We can make the node a private static inner class.

Heads and Tails

- One feature of a linked list is that you always have to keep track of at least one element in it. For a singly linked list it has to be the first one, the head.
- Sometimes it is also helpful to keep track of the last element of the list as well, the tail.
- Other references would be either short lived or for optimizations.

Inserting

- Linked lists excel at inserting and removing. Inserting at the beginning is very easy. Same is true for the end if we keep a tail.
- Inserting into the middle requires walking the list and keeping track of the previous element in the list. This is because you insert after elements and can’t walk the list backwards if it is singly linked.

Removing

- Removing elements from a list is a very similar operation. In this case, we walk the list to find the element, keeping track of the previous one, then set the "pointer" to go around it.

Circular Linked Lists

- It is also possible to build a list where the "tail" points back to the "head". In this case those two terms really aren't all that well defined.
- Instead we can have a pointer anywhere in the list. We still can't walk backwards, but we can walk all the way around to get to anything we want.

Doubly Linked Lists

- Another variation on lists that can be useful is the doubly linked list. In a doubly linked list, every element knows both the one before it and the one after it. With this added in, you can delete an element without walking the list, or add one without having to go looking for the previous one.
- These require a bit more work.

Sentinels

- One way to help simplify linked list code, especially for doubly linked lists, is to add a sentinel.
- This is a special node that signifies an "end" of the list. We can put it at the beginning and the end by making it a circular list.
- Doubly linked lists with sentinels are perhaps that easiest type of list because they lack special case code.

Code

- Now let’s write some code to do a singly linked list.

Minute Essay

- What questions do you have about linked lists? Can you think of an application where you would want to use a linked list instead of an array?
- The design for assignment #3 is due next Tuesday.