CMPSCI 187: Programming With Data Structures

Lecture 11: Linked Data Structures
3 October 2011

Linked Data Structures

- The Basic Idea
- Advantages and Disadvantages
- The Class `LinearNode` in L&C
- Inserting and Deleting in a Linear Linked List
- Searching a Linear Linked List

The Basic Idea of Linked Data Structures

- A Java object can hold any kind of data, including another object of the same class as itself.
- Of course this means that some data field of the object holds a **pointer** to another object of the same class.
- Suppose we extend the class `Dog` to a class `SledDog` which includes a field `SledDog next`, so that if `king` is a `SledDog`, then `king.next` is the dog immediately following King in his sled team.
- If King is the lead dog, the other dogs in the team are `king.next`, `king.next.next`, `king.next.next.next`, and so on. Every dog in the team is reachable from King by following pointers.

More Basics of Linked Structures

- If we are primarily studying the linkages between data items, we call the individual items **nodes**, the usual term in mathematics (along with **vertices**) for the “dots” in a diagram with dots, lines, and arrows -- a **graph**.

- The sled dog team is a particular kind of structure, where every dog has at most one “next” dog, and in addition every dog has at most one predecessor in the team. These two conditions force the structure to be a straight line or a circular loop.

- Stacks, and many other interesting data structures, can be modeled as kinds of **linear lists**. Imagine that we have a pointer to the top node, and it has a pointer to the next node, and that has a pointer to the next, and so on. As we will see, we can push and pop without touching pointers beyond the top’s.

- Later we’ll see more complex linked structures, such as **binary trees**.

Advantages and Disadvantages

- Linked structures take good advantage of the heap memory model of Java. Whenever you want another data item, you grab memory space for it out of the heap and save a pointer to that space so that you can find it as you need it. We'll see that linked structures can expand to arbitrary size without needing the resizing we saw in array structures.

- We generally give up random access in linked structures -- to reach an arbitrary element of a linear list we need to follow all the pointers from the head of the list to the element.

- If an array structure is small enough to be put in faster memory, we might be able to operate on it more quickly -- the compiler can compute addresses by pointer arithmetic, which can be faster than pointer jumping.

The Class `LinearNode` in L&C

- This is a generic class where each node has a content element of type T.
- We can assemble a structure out of linear nodes with just these methods.
- That structure might be a stack, but this class is flexible enough for other structures as well.

```
public class LinearNode<T> {
   private LinearNode<T> next;
   private T element;
   public LinearNode( ) {next = null; element = null;}
   public LinearNode(T elem) {next = null; element = elem;}
   public LinearNode<T> getNext( ) {return next;}
   public void setNext(LinearNode<T> node) {next = node;}
   public T getElement( ) {return element;}
   public void setElement(T elem) {element = elem;}}
```

Inserting and Deleting in a Linear Linked List

- Let’s implement a class `DogTeam` with `LinearNode<SledDog>` objects.
- A DogTeam will have a node with a lead dog, and each node except the last will have a pointer to the next node. The last node’s next pointer will be null.
- To add a new dog at the lead, we change the lead node field, but we must save the pointer to the old lead dog to be the next node for the new lead node.
- Adding a dog in the middle is similar, as long as we make sure that the new dog comes after the one before it and has its next pointer to the next dog.
- Removing a dog from the team requires us to change its predecessor’s next pointer: `balto.next = balto.next.next` cuts out the dog that used to be after Balto. (The correct code would use `getElement` and `getNext`.)

Some Code for the `DogTeam` Class

- Basic ideas: Save anything that might be important before you overwrite it, and check when you are done that all relevant nodes have the right pointers.
- This is O(1) time *if* you already have the place to put the new element and a pointer to it.
- Of course this would make more sense to do generically, but it's good to see.

```
public void insertAfter (LinearNode<SledDog> nodeAfter,
                         SledDog newDog) {
   LinearNode<SledDog> temp = nodeAfter.getNext( );
   LinearNode<SledDog> newNode =
          new LinearNode<SledDog>(newDog);
   nodeAfter.setNext(newNode);
   newNode.setNext(temp);}
```

Searching a Linear Linked List

- Let's write a boolean method `isInTeam` for `DogTeam`.
- We check each dog in the team by following pointers until either (a) we find the target dog, or (b) we run out of dogs and conclude that the target is not there.
- The worst-case running time of this is O(n), where n is the size of the team.
- Note once again how we use && carefully to avoid a NullPointerException.

```
public boolean isInTeam (SledDog sd) {
   LinearNode<SledDog> thisNode = leadNode;
   while (thisNode != null &&
          !thisNode.getElement( ).equals(sd))
      thisNode = thisNode.getNext( );
   return (thisnode != null);}
```