CMPSCI 187: Programming With Data Structures

Lecture #25: More Kinds of Lists
David Mix Barrington
5 November 2012

More Kinds of Lists

- Circular Linked Lists
- Doubly Linked Lists
- Operations for Doubly Linked Lists
- Linked Lists With Headers and Trailers
- A Linked List as an Array of Nodes
- A Specialized List for Bytes
- Case Study: Large Integers

Circular Linked Lists

- As we saw with queues, it's easy to adapt a linked linear structure into a linked **circular** structure. We merely have the last linear node in our list link to the first node in the list, instead of to `null`.

- We also have the external pointer link to the last node in the list rather than the first. (The first node is then `list.getLink( )` instead of just `list`.) This makes it easy to add nodes at either the beginning or the end of the list. Removing is also straightforward with a special case for removing the only node.

- To iterate through the list, we begin with by setting `currentPos` to `list.getLink( )`and then iterate with `currentPos = currentPos.getLink( )` until we reach the last node, which we can detect with `currentPos == list`.

- A circular list is much like having both head and tail pointers in a linear list, but makes a few things easier.

Doubly Linked Lists

- We've seen a few places where the asymmetry between the forward and backward directions in a singly list poses a problem. We can't easily remove a node from the tail of the list, for example. If we ever wanted to traverse a singly linked list backward, we'd be in big trouble.

- A natural way to remove the asymmetry is to give each node two pointers, one forward and one backward. It solves the problems above, at the cost of (1) doubling the amount of memory devoted to links and (2) roughly doubling the number of pointers that have to be changed in adding or removing a node.

```
public class DLLNode<T> extends LLNode<T> {
   private DLLNode<T> back;
   public DLLNode (T info) {super(info); back = null;}
   public void setBack (DLLNode<T> back) {this.back = back;}
   public DLLNode<T> getBack( ) {return back;}}
```

Operations on Doubly Linked Lists

- These methods assume that the instance variable `location` has already been set to the node before which we want to add, or to the node that we want to remove. This code won't add after the last position, and we would have to deal with several special cases in the `remove` method.

```
public void add (T element) {
   DLLNode<T> newNode = new DLLNode<T> (element);
   if (isEmpty( )) {list = newNode; return;}
   newNode.setBack(location.getBack( ));
   newNode.setLink(location);
   location.getBack( ).setLink(newNode);
   location.setBack(newNode);}

public T remove ( ) {
   // special cases for first or last node
   location.getBack( ).setLink(location.getLink( ));
   location.getLink( ).setBack(location.getBack( ));}
```

Linked Lists With Headers and Trailers

- We've already seen the notion of a **sentinel** in a linked list -- a node with null contents at the last position, so that if `currentPos` is any node *with content*, it is still safe to call methods of the node `currentPos.getLink( ).` Our test for the end of the list changes from `(currentPos.getLink( ) == null)` to `(currentPos.getLink( ).getInfo( ) == null).`

- We can use a similar idea to avoid the problems we've just seen with the beginning and end of a doubly-linked list. We put dummy nodes at both the head and tail of the list, so that an "empty list" has two nodes.

- In a sorted list, we can give the header node a value that will be "less than" any legitimate value, and the trailer node a value that will be "greater than" any legitimate value. This minimum and maximum values would depend on the class, and the new list class would expect them to be declared as constants.

A Linked List as an Array of Nodes

- We've mentioned one drawback of linked structures -- when we allocate new nodes by asking the operating system to give us memory for them one by one, we may wind up with nodes that are in a slower area of memory, or nodes that are not near each other in memory, so that they won't all be kept in a faster **cache** at one time.

- A trick to avoid this, very common in systems software, is to hold the linked list entirely within a single **array of nodes**. Each node has an `info` component like that of an `LLNode`, but its "link" is an `int` giving an index into the array. The array's memory will be allocated all at once and thus be contiguous and more likely to be cached together.

- We can set the initial state of the array to have a "free list" of nodes linked together. To put a new node onto our list, we take the first node of the free list and update the free list accordingly. We can keep as many lists in the array as will fit in the available memory.

A Specialized List for Bytes

- Once we understand the basic principles of linked lists, we can design a specialized list class for any particular purpose not well served by our existing list types. In Section 7.5, DJW design a list type to support the large integer case study of Section 7.6 -- they need a list of bytes (8-bit integers) such that they can iterate forward or backward, and add elements at either the front or rear of the list.

- It's quite straightforward to implement this interface with a doubly linked list. Each node is not a `DLLNode<Byte>` but an `SListNode`, from a class defined within the `SpecializedList` class.

```
public interface SpecializedListInterface {
   void resetForward( );
   byte getNextElement( );
   void resetBackward( );
   byte getPriorElement( );
   int size( ); // number of bytes in the list
   void addFront (byte element);
   void addEnd (byte element);}
```

Case Study: Large Integers

- Integers are usually represented in Java by `int` or `long` primitive values, but even long values are limited to about plus or minus 10^{19}. In some circumstances, however, we may want to work with much larger numbers. For example, the **RSA public-key cryptosystem** can be broken by **factoring** a publicly available integer, so it can only be secure if that integer is so large that known factoring methods don't have time to work. This makes factoring algorithms that work on 100-digit or 200-digit integers somewhat important.

- We won't look at many details of DJW's `LargeInt` class in this course, but the basic idea is that any number can be represented as a sequence of digits, and a single digit can be stored in a `byte`. (We could even store pairs of digits in one byte per pair, but DJW want simpler code.) Then we can add or multiply these sequences by carrying out operations on the individual digits, just as we would on paper. We also need methods to convert `int` or `long` values into `LargeInt` values, and a `toString` method as well.