|  |
| --- |
| LINKED LISTS |
|  | ============ |
|  | Last week, we discussed the idea of a Scheme-like list in Java which we called |
|  | a linked list. A linked list is made up of \_nodes\_. Each node has two |
|  | components: an value, and a reference to the tail node in the list. These |
|  | components are analogous to "car" and "cdr" in Scheme. However, our node is an |
|  | explicitly defined object. |
|  |  |
|  | public class IntList { // IntList is a recursive type |
|  | public int value; |
|  | public IntList tail; // Here we're using IntList before |
|  | } // we've finished declaring it. |
|  |  |
|  | Let's make some IntLists. |
|  |  |
|  | IntList l1 = new IntList(), l2 = new IntList(), l3 = new IntList(); |
|  | l1.value = 7; |
|  | l2.value = 0; |
|  | l3.value = 6; |
|  |  |
|  | ------------- ------------- ------------- |
|  | | ----- | | ----- | | ----- | |
|  | |value| 7 | | |value| 0 | | |value| 6 | | |
|  | l1-->| ----- | l2-->| ----- | l3-->| ----- | |
|  | | ----- | | ----- | | ----- | |
|  | | tail| ? | | | tail| ? | | | tail| ? | | |
|  | | ----- | | ----- | | ----- | |
|  | ------------- ------------- ------------- |
|  |  |
|  | Now let's link them together. |
|  |  |
|  | l1.tail = l2; |
|  | l2.tail = l3; |
|  |  |
|  | What about the last node? We need a reference that doesn't reference anything. |
|  | In Java, this is called "null". |
|  |  |
|  | l3.tail = null; |
|  |  |
|  | ------------- ------------- ------------- |
|  | | ----- | | ----- | | ----- | |
|  | |value| 7 | | |value| 0 | | |value| 6 | | |
|  | l1-->| ----- | l2-->| ----- | l3-->| ----- | |
|  | | ----- | | ----- | | ----- | |
|  | | tail| .-+-+-------->| tail| .-+-+-------->| tail| X | | |
|  | | ----- | | ----- | | ----- | |
|  | ------------- ------------- ------------- |
|  |  |
|  | To simplify programming, let's add some constructors to the IntList class. |
|  |  |
|  | public IntList(int i, IntList n) { |
|  | value = i; |
|  | tail = n; |
|  | } |
|  |  |
|  | public IntList(int i) { |
|  | this(i, null); |
|  | } |
|  |  |
|  | These constructors allow us to emulate Scheme's "cons" operation. |
|  |  |
|  | IntList l1 = new IntList(7, new IntList(0, new IntList(6))); |
|  |  |
|  | Linked list insertion |
|  | ---------------------------- |
|  | Unlike arrays, linked lists can keep growing until memory runs out. |
|  |  |
|  | The following method inserts a new value into the list immediately after "this". |
|  |  |
|  | public void insertAfter(int value) { |
|  | tail = new IntList(value, tail); |
|  | } |
|  |  |
|  | l2.insertAfter(3); |
|  |  |
|  | ------------- ------------- ------------- ------------- |
|  | | ----- | | ----- | | ----- | | ----- | |
|  | |value| 7 | | |value| 0 | | |value| 3 | | |value| 6 | | |
|  | l1-->| ----- | l2-->| ----- | | ----- | l3-->| ----- | |
|  | | ----- | | ----- | | ----- | | ----- | |
|  | | tail| .-+-+------>| tail| .-+-+-->| tail| .-+-+------>| tail| X | | |
|  | | ----- | | ----- | | ----- | | ----- | |
|  | ------------- ------------- ------------- ------------- |
|  |  |
|  |  |
|  | A List Class (new stuff) |
|  | ------------ |
|  | There are two problems with the IntList idea. |
|  |  |
|  | (1) Suppose x and y are pointers to the same shopping list. Suppose we insert |
|  | a new value at the beginning of a String version of IntList thusly: |
|  |  |
|  | y = x; |
|  | x = new StringList("soap", x); |
|  |  |
|  | y doesn't point to the new value; y still points to the second value in x's |
|  | list. If y goes shopping for x, he'll forget to buy soap. Gross. |
|  |  |
|  | (2) How do you represent an empty list? The obvious way is "x = null". |
|  | However, Java won't let you call an IntList method--or any method--on |
|  | a null object. If you write "x.insertAfter(value)" when x is null, you'll |
|  | get a run-time error, even though x is declared to be a IntList. |
|  | (There are good reasons for this, which you'll learn later in the course.) |
|  |  |
|  | For these two reasons, the name 'IntList' is arguably not a very good one. It just |
|  | doesn't quite behave how our intuition would suggest. |
|  |  |
|  | For that reason, we'll do two things: |
|  |  |
|  | I. First, we'll define a new class called 'SListNode' as follows: |
|  |  |
|  | public class SListNode { |
|  | public int item; |
|  | public IntList next; |
|  | } |
|  |  |
|  | Note that an SListNode is structually EXACTLY the same as an IntList. The only differences |
|  | are the class name and the field name difference. However, we'll conceptualize |
|  | how this class is used in a completely different way. |
|  |  |
|  | II. Second, we'll create a separate List class, whose job is to maintain the head |
|  | (first node) of the list. We will put many of the methods that operate on |
|  | lists in the SList class, rather than the SlistNode class. |
|  |  |
|  | public class SList { |
|  | private SlistNode head; // First node in list. |
|  | private int size; // Number of items in list. |
|  |  |
|  | public SList() { // Here's how to represent an empty list. |
|  | head = null; |
|  | size = 0; |
|  | } |
|  |  |
|  | public void insertFront(Object item) { |
|  | head = new SlistNode(item, head); |
|  | size++; |
|  | } |
|  | } |
|  | SList object SlistNode object |
|  | ------------- ------------- String object |
|  | ----- | ----- | | ----- | ---------- |
|  | x | .-+----->| size| 1 | | | item| .-+-+---->| milk | |
|  | ----- | ----- | | ----- | ---------- |
|  | ----- | ----- | | ----- | |
|  | y | .-+----->| head| .-+-+-------------------->| next| X | | |
|  | ----- | ----- | | ----- | |
|  | ------------- ------------- |
|  |  |
|  | Now, when you call x.insertFront("fish"), every reference to that SList can see |
|  | the change. |
|  |  |
|  | SList SlistNode SlistNode |
|  | ------------- ------------- ------------- |
|  | ----- | ----- | | ----- | -------- | ----- | -------- |
|  | x | .-+-->| size| 2 | | | item| .-+-+->| fish | | item| .-+-+->| milk | |
|  | ----- | ----- | | ----- | -------- | ----- | -------- |
|  | ----- | ----- | | ----- | | ----- | |
|  | y | .-+-->| head| .-+-+-->| next| .-+-+----------->| next| X | | |
|  | ----- | ----- | | ----- | | ----- | |
|  | ------------- ------------- ------------- |
|  |  |
|  | Now y will never forget to buy fish, soap, milk, or anything else at all. |
|  |  |
|  | Another advantage of SLists |
|  | --------- |
|  |  |
|  | Another advantage of the SList class is that it can keep a record of the |
|  | SList's size (number of SlistNodes). Hence, the size can be determined more |
|  | quickly than if the SlistNodes had to be counted. |
|  |  |
|  | Today, I want to |
|  | introduce another advantage of the SList class. |
|  |  |
|  | We want the SList ADT to enforce two invariants: |
|  | (1) An SList's "size" variable is always correct. |
|  | (2) A list is never circularly linked; there is always a tail node whose |
|  | "next" reference is null. |
|  |  |
|  | Both these goals are accomplished by making sure that \_only\_ the methods of the |
|  | SList class can change the lists' internal data structures. SList ensures this |
|  | by two means: |
|  | (1) The fields of the SList class (head and size) are declared "private". |
|  | (2) No method of SList returns an SlistNode. |
|  |  |
|  | The first rule is necessary so that the evil tamperer can't change the fields |
|  | and corrupt the SList or violate invariant (1). The second rule prevents the |
|  | evil tamperer from changing list items, truncating a list, or creating a cycle |
|  | in a list, thereby violating invariant (2). |