1.00 Lecture 36

Data Structures:
Linked lists

Reading for next time: Big Java: 16.5-16.6

Lists as an Abstract Data Type

A *list* is a collection of elements that has an order.

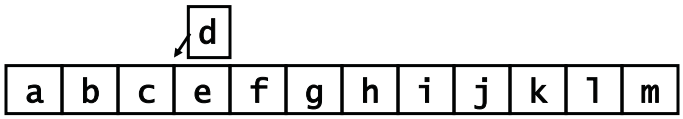
- **It can have arbitrary length.**
- **You should be able to efficiently insert or delete an element anywhere.**
- **You should be able to go through the list in order an element at a time.**

A `List` Interface

```
import java.util.*;

public interface List {
  public boolean isEmpty();
  public void addFirst( Object o );
  public void addLast( Object o );
  public void add(int n, Object o);    // Only in download
  public boolean contains(Object o);
  public Object removeLast()
    throws NoSuchElementException;
  public Object removeFirst()
    throws NoSuchElementException;
  public boolean remove(Object o);      // Only in download
  public void clear();
  public int size();
  public void print();
  public ListIterator listIterator();   // Only in download
}  // Java's List interface is more extensive than ours
```

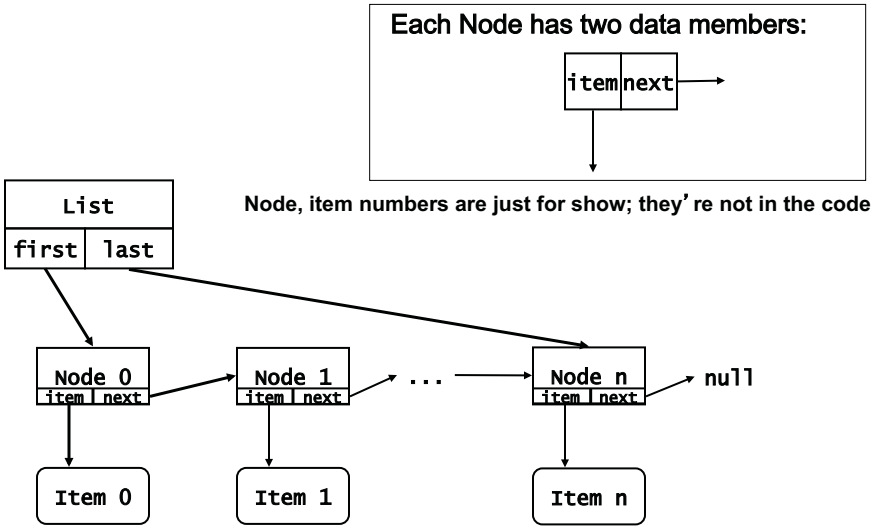
Arrays Don't Work

- If we used an array:
  - Inserting an element at any place except the end of the list is very expensive because all the elements from the point of insertion until the end must be moved back to make room for the new entry.
  - There is a similar problem with deletion.



- For this reason, lists use a *linked* implementation.

Singly Linked List Diagram



Singly Linked Lists, 2

- The `List` points to the first `Node`, and to the last `Node` to make it easier to append items.
  - *"points to"* means has a reference to.
- A `Node` doesn't contain the `item`.
  - It has a reference to the `item`, which can be any `Object`.
  - By pointing to, rather than containing the `item`, we can have one `Node` (and `List`) implementation that works for all lists, regardless of what object type they hold.
- The last `Node`'s `next` data member is `null`, indicating the end of the list.

The Node Nested Class

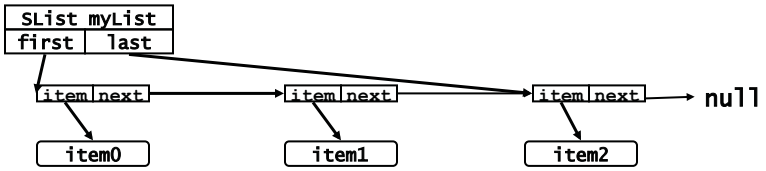
```
public class SLinkedList implements List {
    private static class Node {   // Pkg access in download
       Object item;               // to support visual demo
       Node next;

       Node( Object o, Node n ) {
          item = o; next = n;
        }
   }
   . . .
// This example uses nested class Node (static keyword)

// Also, we could use generics (e.g. <T>) but we use just
// Objects for simplicity. Generic version in download.

// Our SLinkedList is simpler than Java LinkedList class
// but uses very similar implementations
```

The `SLinkedList` Data Members

```
private Node first = null;
private Node last = null;
private int length = 0;
```

- Only `first` is necessary.
- `last` and `length` could be found by traversing the list
  - Having these members and keeping them up to date makes the methods `size()` and `addXXX()` faster.
  - Implementations vary on this point.

Exercise 1: Programming Links



- Linked lists use references to the components of the list.
- In `SList`, you refer to first item on the list (`item0`) as `first.item`.
- How would you refer to the second item on the list if it was present?
- How could you tell if there was a 2nd item? Write an `if` statement.
- How would you refer to the last item on the list?

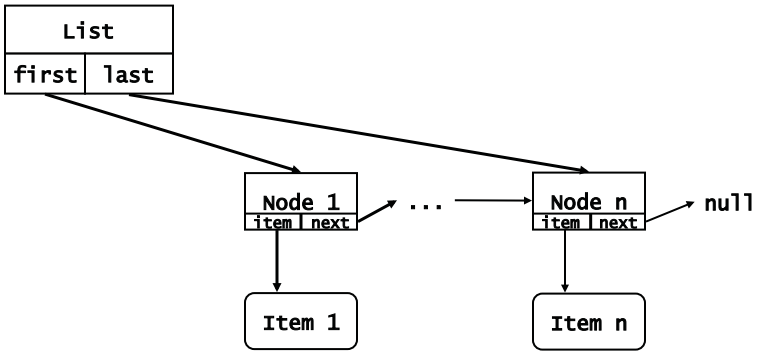
Beware the Special Case

- The tricky part about implementing a linked list is not implementing the normal case for each of the methods, for instance, removing an object from the middle of the list.
- What's tricky is making sure that your methods will work in the exceptional and boundary cases.
- For each method, you should think through whether the implementation will work on
  - an empty list,
  - a list with only one or two elements,
  - on the first element of a list,
  - on the last element of a list.

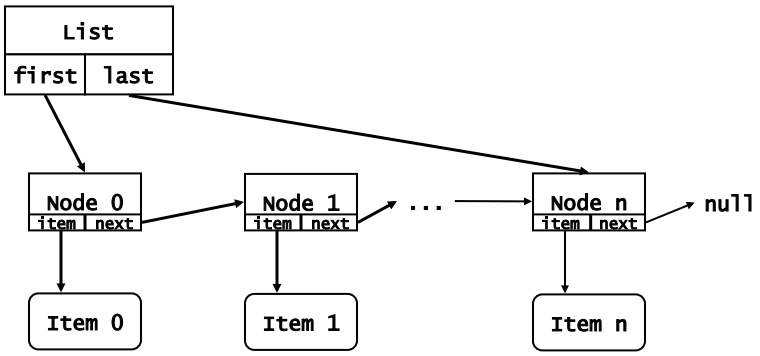
addFirst(Object o)

```
public void addFirst(Object o)
{
    if ( first == null ) {       // If the list is empty
        first = new Node( o , null);
        last = first;
    }
    else {
        first = new Node( o, first );
    }
    length++;
}
```

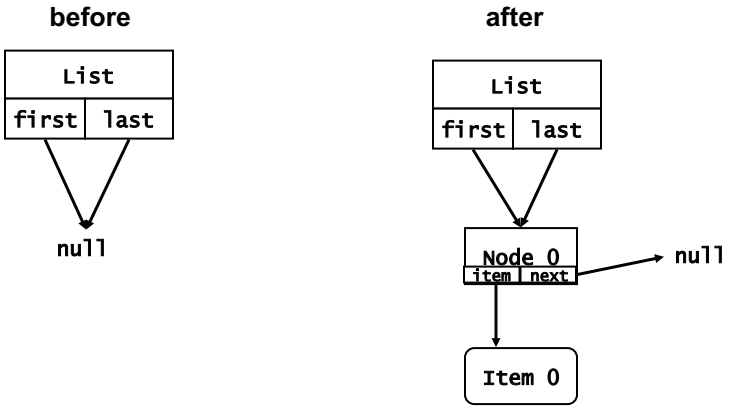
addFirst(), before



addFirst(), after



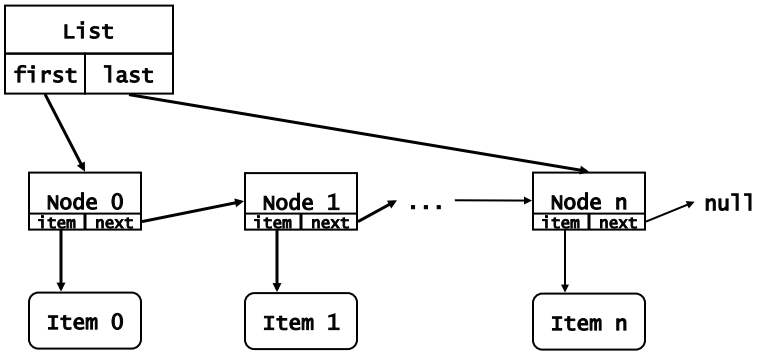
addFirst(), special case



Exercise 2

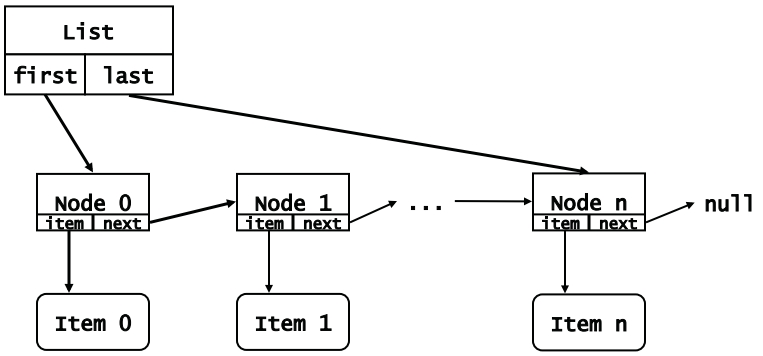
- **Download List and SLinkedList**
- **Write addLast() in SLinkedList:**
  - **Draw a picture of the list before and after**
  - **Handle the special case of a currently empty list**
  - **Remember to increment the list length**

Exercise 2: addLast(), before

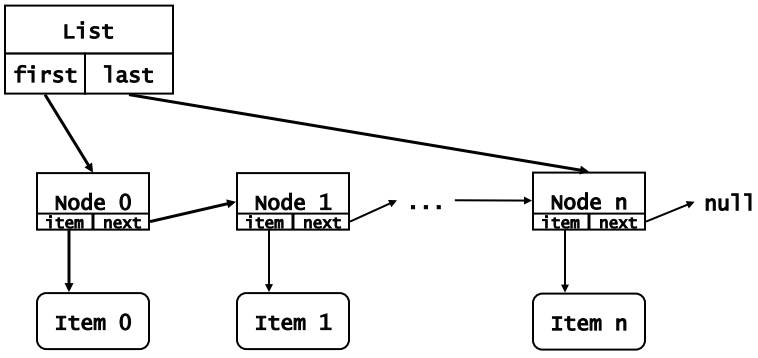


Exercise 3

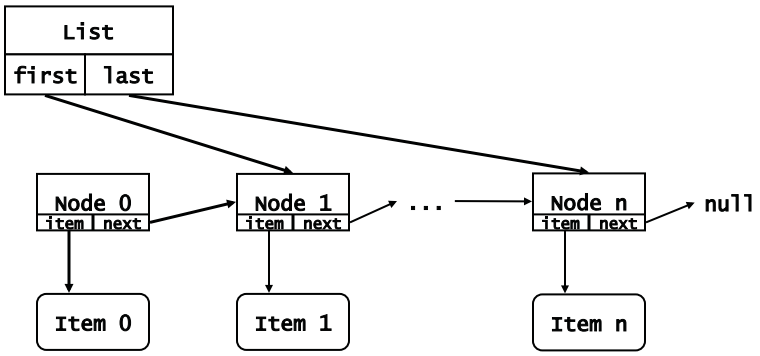
- **Write the print() method**
  - **Check if list is empty**
  - **Otherwise "walk" the list and print out each item**
    - **Use a while loop**



`removeFirst()`, before



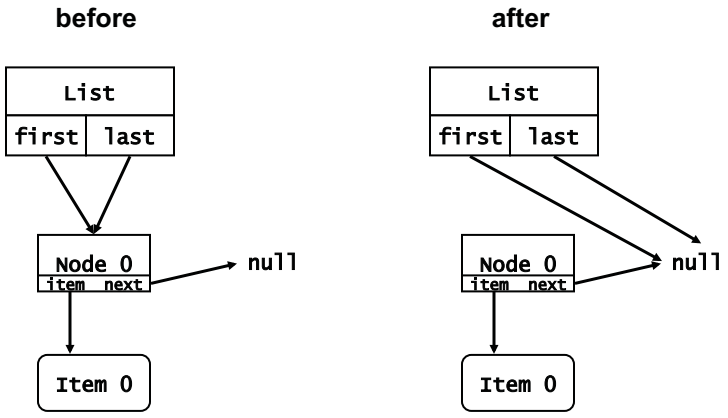
removeFirst(), after



removeFirst()

```
public Object removeFirst()
   throws NoSuchElementException
{
    if ( first == null )        // if list is empty
        throw new NoSuchElementException();
    else {
        Node t = first;
        first = first.next;
         // if list had 1 element and is now empty
        if ( first == null )
            last = null;
        length--;
        return t.item;
    }
}
```

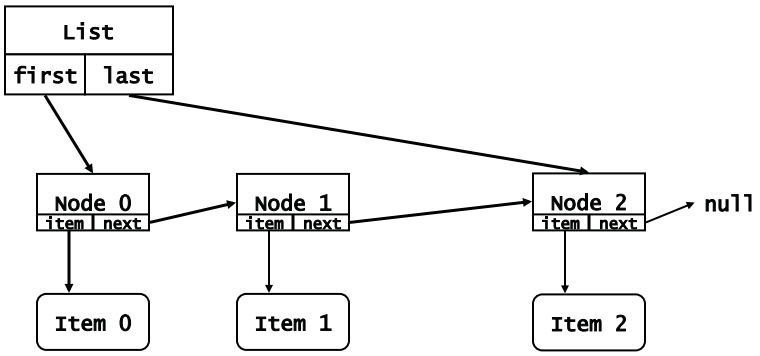
`removeFirst()`, special case



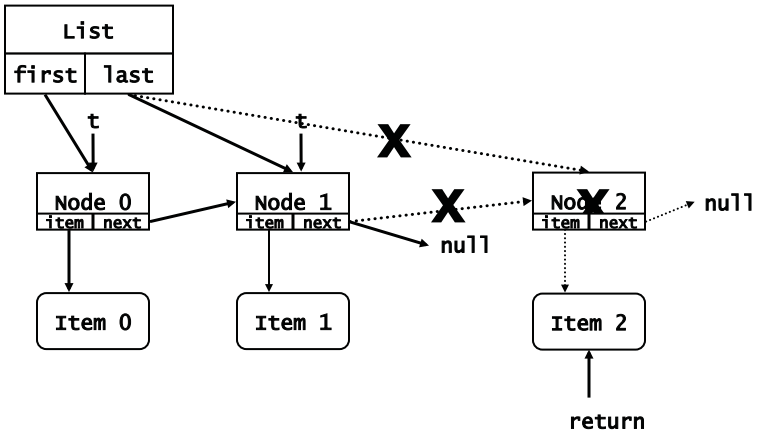
Exercise 4

- **Write removeLast()**
  - **We give you pictures of the list before and after removing the last element**
  - **We give you the special cases where the list currently has:**
    - **No elements**
    - **One element**
  - **You only need to write the standard case**
    - **Find the Node before the last Node; it will become the last Node**
    - **Set its next field to null**
    - **Return the last item (removeLast() returns an Object)**
    - **Remember to decrement list length**

removeLast(), before



removeLast(), after



Exercise 4

```
public Object removeLast()
   throws NoSuchElementException {
       if (first == null)         // Empty list
           throw new NoSuchElementException();
       else if (first == last) {  // 1 element in list
           Node t= first;
           first= last= null;
           length= 0;
           return t.item;
       }
       else {
            // Your code here (remove "return null;")
       }
   }
}
```

contains()

```
  public boolean contains(Object o) {
        boolean found= false;
        if (first == null)
            return false;
        Node t= first;
        while (t != null) {
            if (t.item.equals(o)) {
                found= true;
                break;
            }
            t= t.next;
        }
        return found;
    }
```

Other methods

```
    public int size() {
        return length;    }

    public boolean isEmpty() {
        return( first == null );    }

    public void clear() {
        first = last = null;
        length = 0;
    }

// Note that we've implemented a double ended queue:
// elements can arrive or leave at front or rear

// Download and run ListTest to use your SLinkedList
// Generic version of linked list in download:
//   List, ListIterator, ListTest, SLinkedListG
```

Exercise 5

- **Download:**
  - **SLinkedListApp, SLinkedListView, ListUtil, Screen, ListIterator, ListIteratorView**
- **Rename or copy your SLinkedList to SLinkedList1**
  - **Copy: Highlight SLinkedList in explorer, ctrl-C, ctrl-V, type new name**
  - **Rename: Highlight SLinkedList in explorer, right click, Refactor-> Rename, type new name**
- **Run SLinkedListApp and experiment**
  - **Enter one- or two-digit integers as the 'items' in the list**
  - **Remove and double aren't implemented**
  - **We don't cover ListIterator, though you can try it**
    - **Iterators are a generic interface to manage many data structures**

Java LinkedList class

- Implements Java List interface
  - More methods than our List interface in lecture:
    - add() [several], addAll(), addFirst(), addLast()
    - removeFirst(), removeLast(), etc.
    - clear(), contains(), indexOf(), size(), get(), set(), etc.
    - push(), pop(), etc. to implement stacks
    - addXXX() and removeXXX() used to implement queues and dequeues, as well as general lists
    - Choose between an ArrayDeque and LinkedList implementation for stacks, queues, dequeues
- ArrayList also implements List, which we saw much earlier this semester
  - LinkedList and ArrayList are the commonly used lists.
  - Their efficiencies are different
    - ArrayList is faster for more static data
    - LinkedList is faster more more dynamic (rapidly changing) data
  - See Javadoc for more specialized lists

Java LinkedList Class Example

```
import java.util.*;

public class ListExample {
   public static void main(String[] args) {
        LinkedList<String> sensors= new LinkedList<String>();
        sensors.addFirst("light");
        sensors.addLast("touch");
        sensors.add("slider");        // Adds at end
        for (String s: sensors)
               System.out.println(s);
        System.out.println();
        sensors.remove(0);            // Remove at index 0
        sensors.remove("slider");
        for (String s: sensors)
               System.out.println(s);
   }    // Catch exceptions to make code robust. Not in example
}
```

1.00 / 1.001 / 1.002 Introduction to Computers and Engineering Problem Solving
Spring 2012