1.00 Lecture 27

Data Structures:
Linked lists

Reading for next time: Big Java: 20.5

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.

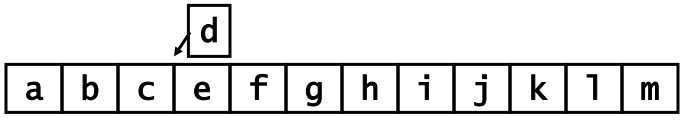
A `List` Interface

```
import java.util.*;

public interface List {
  public boolean isEmpty();
  public void addFirst( Object o );
  public void addLast( Object o );
  public 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
}
// This interface doesn't have an add() method to
// place an element in an arbitrary position. It's
// straightforward but tedious—we don't cover it
```

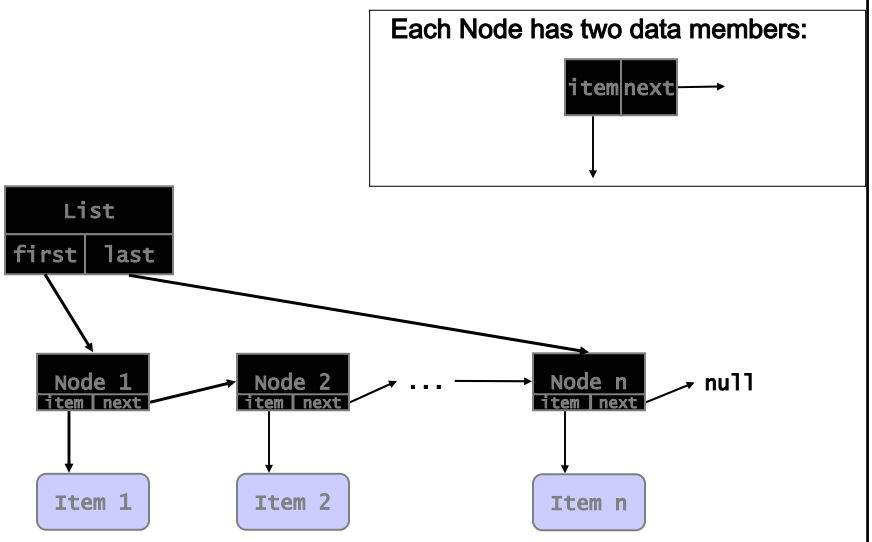
Arrays Don't Work

- 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 Inner 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 inner classes; you don't have to
```

Generic vs. Typed Lists

- The `List` interface we have specified is general like the Java `ArrayList` class. It stores and retrieves `Objects`.
- If you are writing your own list class, and you know you will only be handling `Students`, say, you can replace the `Object` fields with `Student` fields. For example,

```
private static class Node {
    Student s;
    Node next;
    Node( Student s, Node n ) {
       this.s = s;
       next = n;
    }
}

public void addFirst( Student s );  // Etc.
```

The SLinkedList Data Members

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

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

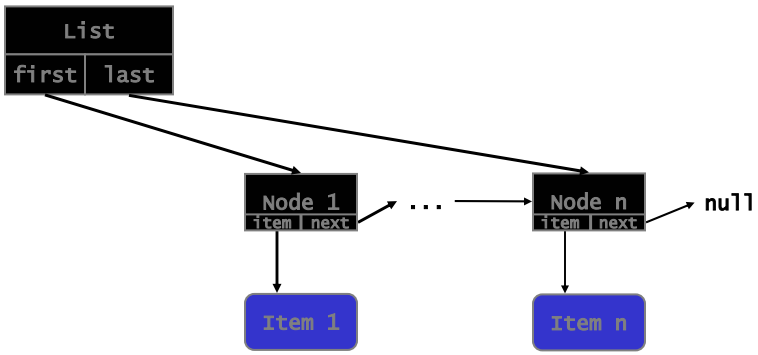
Beware the Special Case

- 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.

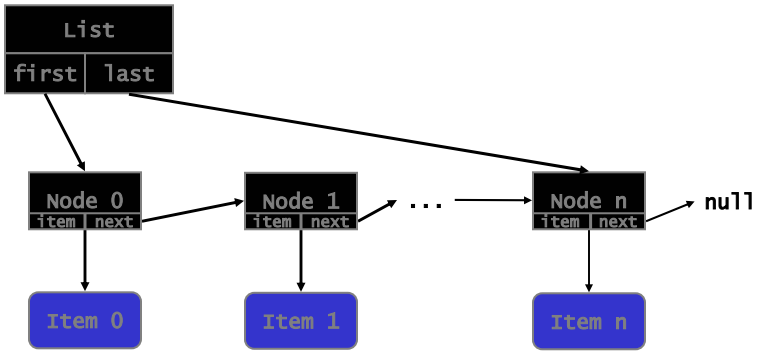
addFirst(Object o)

```
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++;
}
```

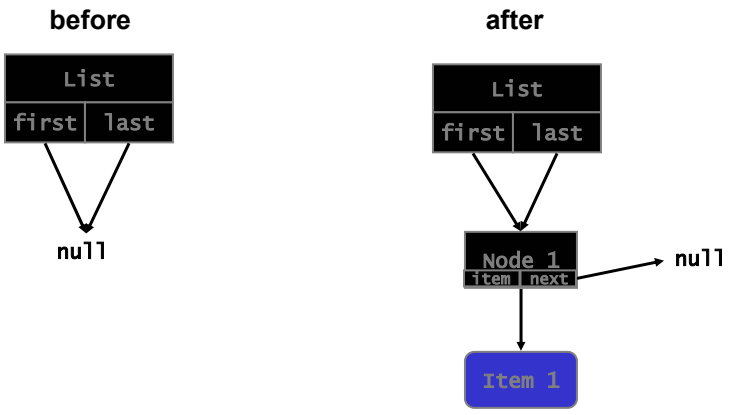
addFirst(), before



addFirst(), after



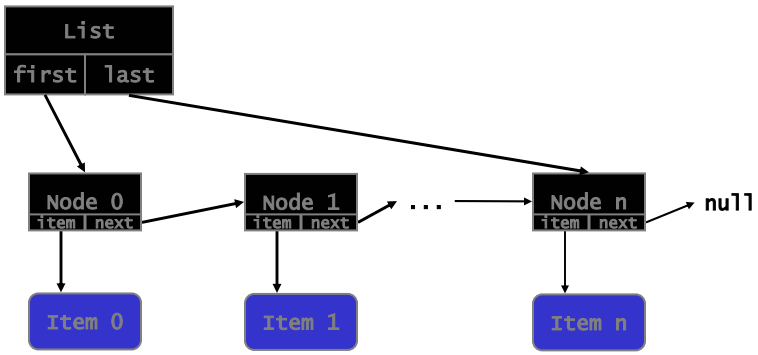
addFirst(), special case



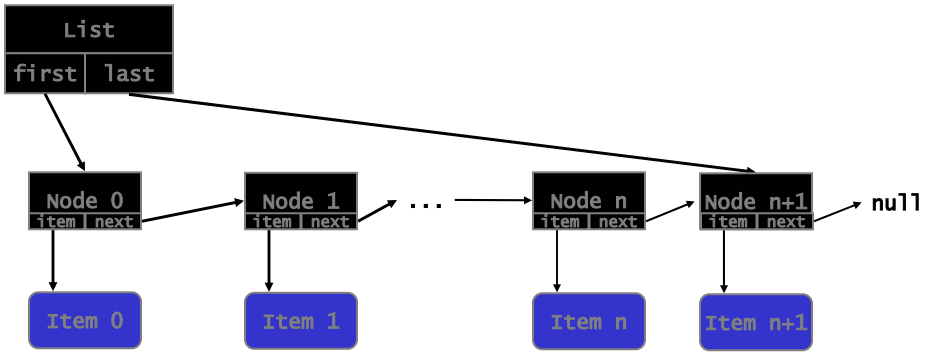
Exercise

- **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**

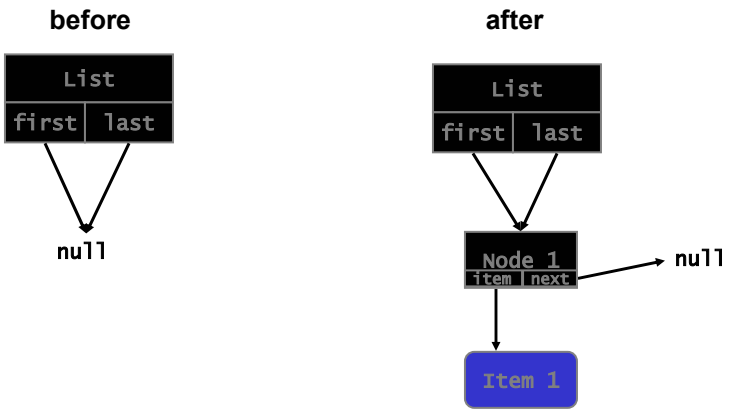
Solution: addLast(), before



Solution: addLast(), after

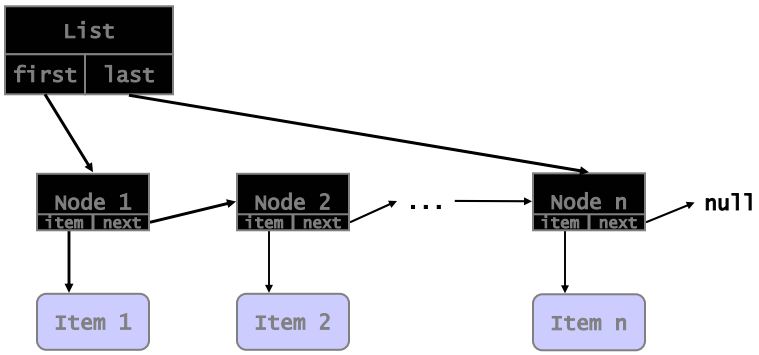


Solution: addLast(), special case

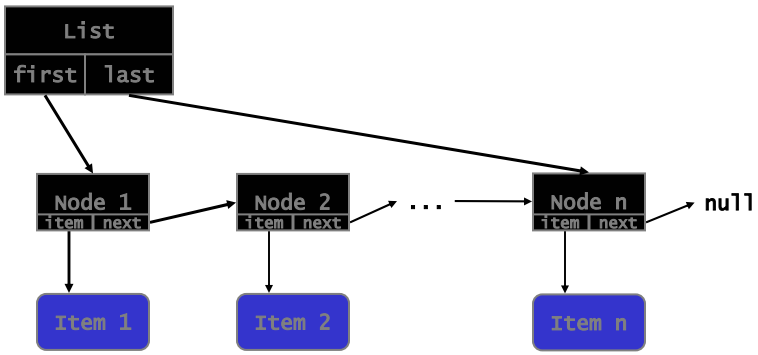


Exercise

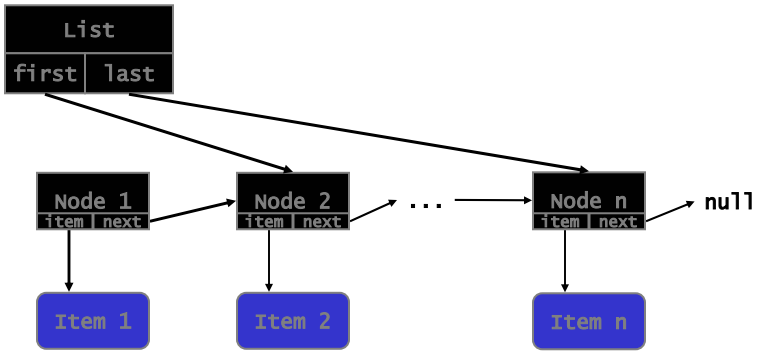
- Write the print() method
  - Check if list is empty
  - Otherwise “walk” the list and print out each item



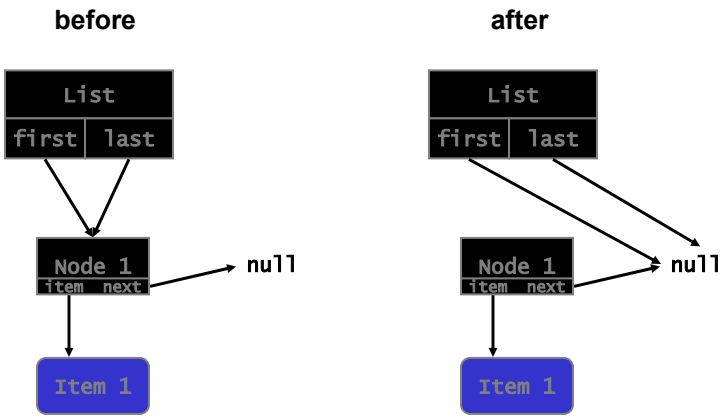
removeFirst(), before



removeFirst(), after



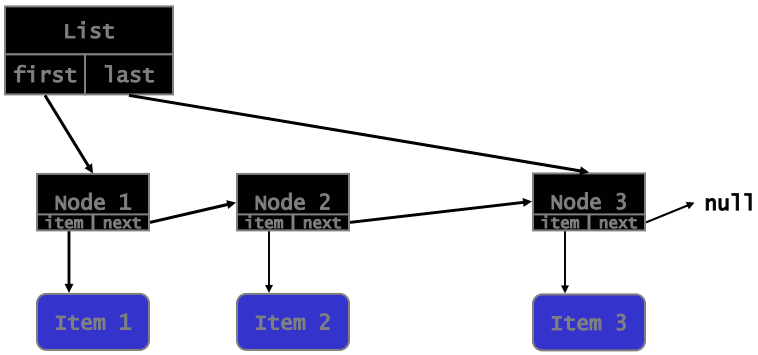
removeFirst(), special case



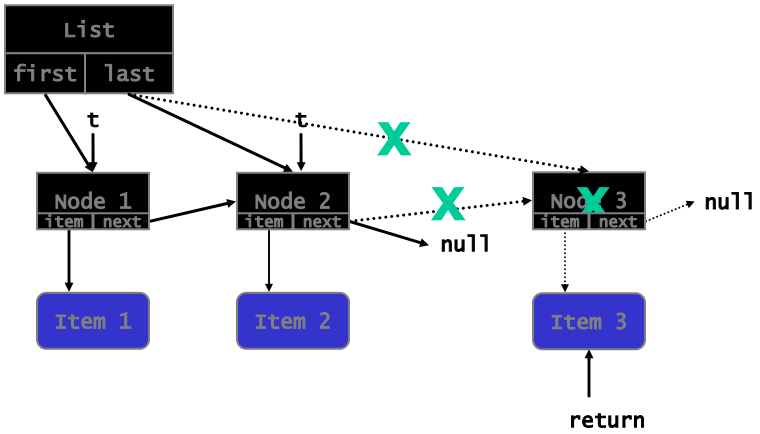
Exercise

- **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

```
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
```

ListTest

```
// Download and run ListTest to use your SLinkedList

public class ListTest {
  public static void main(String[] args) {
    SLinkedList a= new SLinkedList();
    while (true) {

      String text= JOptionPane.showInputDialog(
        "Enter 1-addFirst, 2-addLast, 3-removeFirst,
        4-removeLast, 5-contains, 6-quit: ");

        int action= Integer.parseInt(text);
        if (action == 6)
          break;
        if (action== 1 || action== 2)
          text= JOptionPane.showInputDialog("Enter
                    string to store in list: ");
        // Continued
```

ListTest, p.2

```
      try {
       if (action == 5)
         text= JOptionPane.showInputDialog("Enter string
              to find ");
       if (action == 1)
         a.addFirst(text);
       else if (action == 2)
         a.addLast(text);
       else if (action == 3)
         a.removeFirst();
       else if (action== 4)
         a.removeLast();
       else
         System.out.println("List has " + text + "?: “
             + a.contains(text));
      } catch (NoSuchElementException e) {
          System.out.println("List empty (exception)"); }
      a.print();
}  }  }
```

Exercise

- **Download:**
  - **SLinkedListApp**
  - **SLinkedListView**
  - **ListUtil**
  - **Screen**
  - **ListIterator**
  - **ListIteratorView**
- **Rename your SLinkedList to SLinkedList1**
- **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**

Linked List Uses

- **Can implement stacks, queues, trees, and almost any other data structure as a linked list or array**
  - **Choose based on application needs, based on how dynamic your data is, whether you need to insert in the middle of your data…**
  - **We still have trees and graphs to cover, which handle dynamic data well**
- **Doubly-linked lists are convenient for some data structures, as shown on next page:**
  - **When you have to add and/or delete at both ends of the data structure often**

Doubly Linked List Diagram