STUDENT OUTLINE

Lesson 33 – Java Lists and Iterators

INTRODUCTION: This lesson presents the Java library `List` interface and the `LinkedList` class that implements the `List` interface. The lesson also introduces the use of iterators to traverse linked lists.

The key topics for this lesson are:

A. The `List` Interface
B. `LinkedList` Library Implementation of the `List` Interface
C. Traversing a List using `Iterator` or `ListIterator` Objects

VOCABULARY: TRAVERSE ITERATOR

DISCUSSION:

A. The `List` Interface

1. The List interface from the `java.util` package gives a formal description of a list. The Java library also provides two classes, `ArrayList` and `LinkedList` that implement the `List` interface.

2. Like `ArrayList` and other collection classes, `LinkedList` can store objects that have the `Object` data type. Since any class extends `Object`, you can put any kind of object into a list. Since the methods that retrieve values from the list return an object, you have to cast the object back into its original type when it is retrieved from the list. For example:

```
List classList = new LinkedList();
classList.add(?APCS?);
...
String favoriteClass = (String)classList.get(1);
```

3. These classes assume that the values in the list have the type `Object`. To put primitive data types such as **`int`**s or **`double`**s into the list, you need to first convert them to into objects using the appropriate wrapper class, `Integer` or `Double`. For example:

```
List numList = new LinkedList();
numList.add(new Integer(23));
numList.add(new Double(3.14159));
...
int i = ((Integer)numList.get(0)).intValue();
double d = ((Double)numList.get(1)).doubleValue();
```

4. It is assumed that the elements in the list are indexed starting from 0. "Logical" indices are used even if a list is not implemented as an array. Methods that use indices generate a run-time error if an index is out of bounds.

5. A few commonly used `List` methods are summarized below.

```
boolean add(Object element)
// Appends the given element to the end of this list

void add(int index, Object element)
// Inserts the specified element at the specified position

int size()
// Returns the number of elements in this list

Object get(int index)
// Returns the element at the specified position in this list

Object set(int index, Object element)
// Replaces the element at the specified position in this list
//   with the specified element

Object remove(int index)
// Removes the first occurrence in this list of the specified
//   element
```

B. <u>`LinkedList` Library Implementation of the `List` Interface</u>

1. The `java.util` package of the standard class library has a `LinkedList` class. The `LinkedList` class implements the `List` interface. As the class name indicates, the underlying implementation of the `LinkedList` class is a linked list.

2. In addition to the methods specified by the `List` interface, the `LinkedList` class provides a few specialized methods that allow easy access to both ends of the list as follows:

```
void addFirst(Object element)
// Inserts the given element at the beginning of this list.

void addLast(Object element)
// Appends the given element to the end of this list.

Object getFirst()
// Returns the first element in this list

Object getLast()
// Returns the last element in this list

Object removeFirst()
// Removes and returns the first element from this list

Object removeLast()
// Removes and returns the last element from this list
```

C. Traversing a List using `Iterator` or `ListIterator` Objects

1. A traversal of a list is an operation that visits all the elements of the list in sequence and performs some operation. For example, the following loop can be used to traverse a linked list:

   ```
   ListNode node = first;    // start from the first node
   while (node != null)
   {
     SomeClass value = (SomeClass)node.getValue();
     // process value
     ...
   }
   ```

   When the linked list is implemented as an encapsulated class, `first` is no longer directly accessible. To provide access to the elements of a list and maintain the protection afforded by encapsulation, the Java library supplies an `Iterator` type.

2. An iterator is an object associated with the list. When an iterator is created, it points to a specific element in the list, usually the first. We call the iterator's methods to check whether there are more elements to be visited and to obtain the next element.

3. In Java, the iteration concept is expressed in the library interface `java.util.Iterator`. The `Iterator` interface is used by classes that represent a collection of objects such as a list, providing a way to move through the collection one object at a time. The `Iterator` interface is not used to represent the list itself, it merely represents a way to move through the elements of the list.

4. An `Iterator` object provides three basic methods:

   ```
   Object next()
   // Returns the next element in the iteration

   Object hasNext()
   // Returns true if the iteration has more elements

   void remove()
   // Removes the last element returned by next from the list
   ```

5. A list traversal would be implemented with an iterator as follows:

   ```
   LinkedList list = new LinkedList()
   // Add values to the list
   ...
   ```

```
Iterator iter = list.iterator();
while (iter.hasNext())
{
  Object obj = iter.next()
  System.out.println(obj);
}
```

Note that the list itself provides an iterator when its `iterator()` method is called.

6. A limitation of the `Iterator` interface is that an iterator always iterates from the beginning of the list and only in one direction.

7. A more comprehensive `ListIterator` object is returned by `List`'s `listIterator` method. `ListIterator` extends `Iterator`. A `ListIterator` can start iterations at any specified position in the list and can proceed forward or backward. For example:

```
ListIterator listIter = list.listIterator(list.size());
while (listIter.hasPrevious())
{
  SomeClass value = (SomeClass)listIter.previous();
  // process value
  ...
}
```

8. Some useful `ListIterator` methods are summarized below

```
Object next()
// Returns the next element in the iteration
Object hasNext()
// Returns true if the iteration has more elements

Object previous()
// Returns the previous element in the iteration
Object hasPrevious()
// Returns true if the previous element in the list is
//   is available, false otherwise

void add(Object obj)
// Inserts the element obj into the list immediately after
//   the last element that was returned by the next method

void set(Object obj)
// Replaces the last element returned by next or pervious
//   with the element obj

void remove()
// Removes the last element returned by next or previous
//   from the list
```

SUMMARY/ REVIEW: In Lesson 25, you developed a recursive merge sort algorithm using arrays. An unordered linked list is difficult to sort given its sequential nature, but a recursive merge sort can be developed for linked lists using the `LinkedList` class. The algorithms required (split and merge) will provide excellent practice in working with the `LinkedList`, `Iterator`, and `ListIterator` classes. After applying a recursive merge sort to a linked list, another function to reverse the list will be written.

ASSIGNMENT: Lab Exercise L.A.33.1, *MergeList*

LAB EXERCISE

MergeList

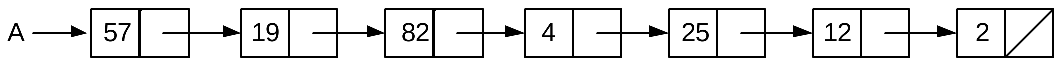
Background:

An unordered linked list is difficult to sort given its sequential nature, but a recursive merge sort can be developed for linked lists. The algorithms required (split and merge) will provide excellent practice in working with the `java.util.LinkedList` and `ListIterator` classes.

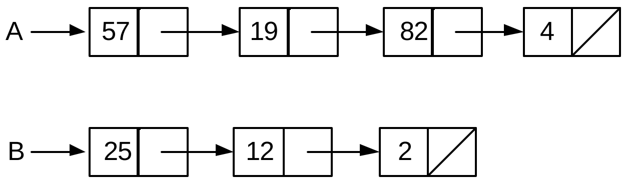
A linked list can be sorted using a recursive merge sort algorithm. Here are the steps:

1. Split the list into two smaller lists.
2. Recursively sort these two lists using merge sort.
3. Merge the two sorted lists together.

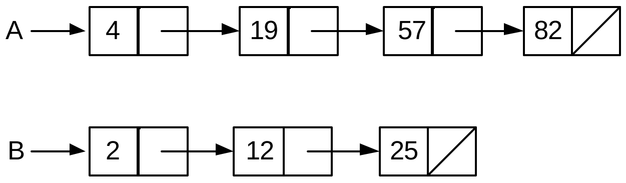
Suppose we begin with a list of seven nodes, pointed to by A.



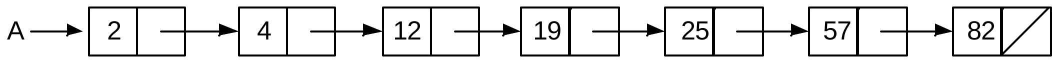
We then split this list into two smaller lists that differ in size by no more than one node. It does not matter which values end up in either list. In the example to follow, list A will take the first half of the list while list B will have the second half of the original list.



Next, we recursively sort these two lists.



Finally, we merge these two sorted lists together.



You are encouraged to look over your code for the `merge` and `mergeSort` methods from Lesson 25. Reviewing these algorithms as applied to arrays will help you to solve the same mergesort concept with linked lists.

Assignment:

1. The linked list should be of type `java.util.LinkedList`.

2. The data file to be used in this lab is (*file20.txt*).

3. Building the initial linked list from (*file20.txt*) should follow this logic. As each new piece of data (Id/Inv pair) comes off the data file, it is placed at the beginning of the list. Therefore, the first values read from the data file will end up last in the list.

4. The recursive merge sort algorithm will need the supporting algorithms of splitting and merging lists. List iterators should be used to implement these methods.

5. Your program should consist of this sequence of scripted events:

   Load the data file and build the initial list
   Print the linked list - it is unordered
   Recursively merge sort the list
   Print the linked list - it is now sorted
   Reverse the linked list
   Print the linked list - it is now in descending order

6. If your instructor chooses, you will be provided with a program shell consisting of a `MergeList` class containing a main method, and the `Item` class. All of the code development should appear in the `MergeList` class. Here are some of the specifications of the `MergeList` class.

   a. The `reverseList` method is stubbed out as a print statement.

   b. The `split` method is stubbed out as a print statement.

   c. The `merge` method is stubbed out as a print statement.

   d. Methods to read the data file and print the list are provided.

Instructions:

1. Modify and write code as necessary to satisfy the above specifications.

2. Print out the source for the `MergeList` class.

3. Turn in your source along with the run output.