STUDENT OUTLINE

Lesson 30 – Linked Lists

INTRODUCTION: In this lesson you begin to study a new data structure, the linked list, which is used to implement a list of elements arranged in some kind of order. The linked list structure uses memory that shrinks and grows as needed but not in the same way as arrays. The discussion of linked lists includes the specification and implementation of a node class, which incorporates the fundamental notion of a single element of a linked list.

The key topics for this lesson are:

A. Declarations for a Linked List
B. Methods for Manipulating Nodes
C. Implementing Linked Lists
D. Traversing a Linked List
E. Pitfalls of Linked Data Structures

VOCABULARY:

NODE	LINKED LIST
TRAVERSE	NODE
NULL REFERENCE	

DISCUSSION:

A. Declarations for a Linked List

1. A linked list is a sequence of elements arranged one after another, with each element connected to the next element by a "link." The link to the next element is combined with each element in a component called a *node*. A node is represented pictorially as a box with an element written inside of the box and a link drawn as an arrow and used to connect one node to another.

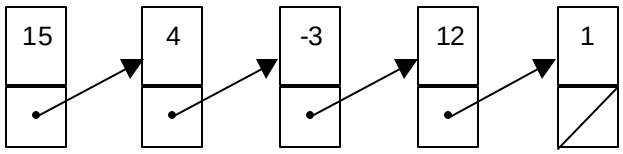


Figure 30-1

In addition to connecting two nodes, the links also place the nodes in a particular order. In Figure 30-1 above, the five nodes form a chain with the first node linked to the second; the second node linked to the third node; and so on until the last node is reached. The last node is a special case since it is not linked to another node and is indicated with a diagonal line.

2. Each node contains two pieces of information: an element and a reference to another node. This can be implemented as a Java class for a node using an instance variable to hold the element, and a second instance variable that is a reference to another node as follows:

```
public class ListNode
{
  private Object value;   // the element stored in this node
  private ListNode next;  // reference to next node in List
  ...
}
```

3. The declaration seems circular and in some ways it is, but the compiler will allow such definitions. A `ListNode` will have two data members, an `Object` and a reference to another `ListNode`. The instance variable `next` will point to the next `ListNode` in a linked list.

4. The `ListNode` class is constructed so that the elements of a list are objects (i.e., have the `Object` data type). Since any class extends `Object`, you can put any kind of object into a list, including arrays and strings.

5. Whenever a program builds and manipulates a linked list, the nodes are accessed through one or more references to nodes. Typically, a program includes a reference to the first node (`first`) and a reference to the last node (`last`).

```
ListNode first;
ListNode last;
```

A program can proceed to create a linked list as shown below. The `first` and `last` reference variables provide access to the first and last nodes of the list

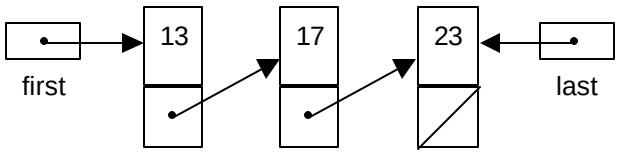


Figure 30-2

6. Figure 30-2 illustrates a linked list with a reference to the first node that terminates at the final node (indicated by a diagonal line in the reference field of the last node). Instead of a reference to another node, the final node contains a *null reference*, which is a special Java constant. The null reference can be used for any reference variable that has nothing to refer to. There are several common situations where the null reference is used:

a. When a reference variable is first declared and there is not yet an object for it to refer to, it can be given an initial value of the null reference.

b. The null reference occurs in the link part of the final node of a linked list.
c. When a linked list does not yet have any nodes, the null reference is used for the `first` and `last` reference variables to indicate an *empty* list.

In a program, the null reference is written as the keyword **`null`**.

B. Methods for Manipulating Nodes

See, *ListNode.java* [*]

1. Methods for the `ListNode` class will consist of those for creating, accessing, and modifying nodes

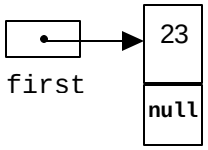
2. The constructor for the `ListNode` class is responsible for creating a node and initializing the two instance variables of a new node. The node's constructor has two arguments, which are the initial values for the node's data and link variables. The constructor's implementation copies its two parameters to the instance variables, `value` and `next`:

```
public ListNode(Object initValue, ListNode initNext)
// post: constructs a new element with object initValue,
//       followed by next element
{
  value = initValue;
  next = initNext;
}
```

3. As an example, the constructor can be used by a program to create the first node of a linked list.

```
ListNode first;
first = new ListNode(new Integer(23), null);
```

After execution of the two statements, `first` refers to the header node of a small linked list that contains just one node with the `Integer` 23.



Since primitive data types (**`int`**, **`double`**, **`boolean`**, etc.), are not objects, it is necessary to convert them to objects using the appropriate wrapper class (`Integer` for **`int`**, `Double` for **`double`**, etc.) to construct a list.

[*] Adapted from the College Board's *AP Computer Science AB: Implementation Classes and Interfaces*.

4. Getting and setting the data and link of the node is accomplished with an accessor method and a modification method for each of its instance variables as follows:

```
public Object getValue()
// post: returns value associated with this element
{
  return value;
}

public ListNode getNext()
// post: returns reference to next value in list
{
  return next;
}

public void setValue(Object theNewValue)
{
  value = theNewValue;
}

public void setNext(ListNode theNewNext)
// post: sets reference to new next value
{
  next = theNewNext;
}
```

5. The following short program using `ListNode` will illustrate the syntax of accessing the data members of a `ListNode`.

```
public static void main(String[] args)
{
  ListNode list;

  list = new ListNode(new Integer(13), null);

  System.out.println("The node contains: " +
                     (Integer)list.getValue());

  list.setValue(new Integer(17));
  System.out.println("The node contains: " +
                     (Integer)list.getValue());
}
```

Run Output:

```
The node contains: 13
The node contains: 17
```

C. Implementing Linked Lists

See *ListDemo.java* and *SinglyLinkedList.java*

1. In this section, we will look at the implementation of a linked list of `ListNode` objects. This class encapsulates the list operations that maintain the links as the list is modified. To keep the class simple, we will implement only a singly linked list, and the list class will supply direct access only to the first list element.

2. The `SinglyLinkedList` class holds a reference `first` to the first `ListNode` (or **`null`**, if the list is completely empty). Access to the first node is provided by the `getFirst` method. If the list is empty, a `NoSuchElementException` is thrown (see Lesson 17, *Exceptions*).

```
class SinglyLinkedList
{
  private ListNode first;

  public SinglyLinkedList()
  {
    first = null;
  }

  public Object getFirst()
  {
    if (first == null)
    {
      throw new NoSuchElementException();
    }
    else
      return first.getValue();
  }
}
```

3. Additional nodes are added to the head of the list with the `addFirst` method. When a new link is added to the list, it becomes the head of the list, and the link that the old list had becomes the next link:

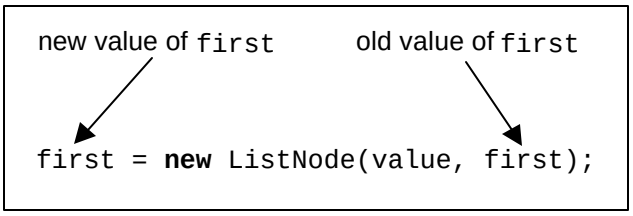
```
class SinglyLinkedList
{
  ...
  public void addFirst(Object value)
  {
    first = new ListNode(value, first);
  }
  ...
}
```

4. The statement `ListNode(value, first)` invokes the constructor. The line of code

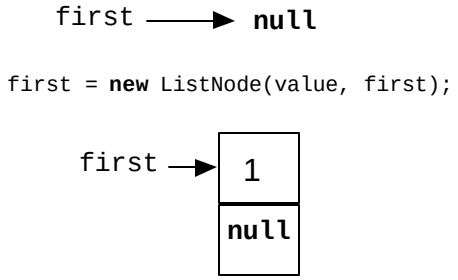
```
first = new ListNode(value, first);
```

is broken down as follows:

a. The **`new`** command allocates enough memory to store a `ListNode`.
b. The new `ListNode` will be assigned the values of `value` and `first`.
c. The address of this newly constructed `ListNode` is assigned to `first`.
d. It is important to understand the old and new values of `first`:



5. When `value = 1`, `first = null`. A new node is constructed with the values 1 and **`null`**. `first` points to this new node. In a sense, the constructor provides a new node between the variable `first` and the node that `first` formerly referenced.

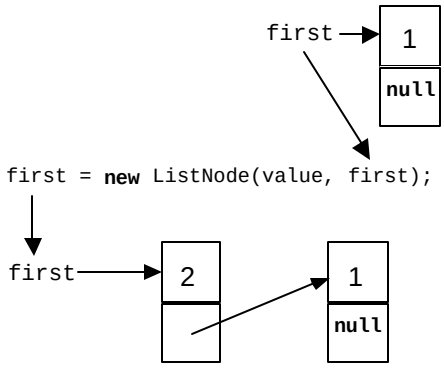


before the call of the constructor

call the constructor, `first` is passed as a **`null`** value

`first` is changed, references the newly constructed node

6. When value = 2, `first` is already pointing to the first node of the linked list. When the constructor is called, the new node is constructed and placed between `first` and the node `first` used to point to.



The value of `first` passed to the `ListNode` constructor is used to initialize the `next` field of the new node.

D. Traversing a Linked List - Method `printList`

1. To traverse a list means to start at one end and visit all the nodes, solving a problem along the way. In the case of method `printList`, the task is to print the `value` field from each node.

```
class SinglyLinkedList
{
  ...
  public void printList()
  {
    ListNode temp = first;    // start from the first node
    while (temp != null)
    {
      System.out.print(temp.getValue() + " ");
      temp = temp.getNext();  // go to next node
    }
  }
  ...
}
```

   a. Because `temp` is an alias to `first`, we can use it to traverse the list without altering the reference to the start of the list. The `ListNode` variable, `temp`, will contain **`null`** when we are done.

   b. Until `temp` equals **`null`**, the **`while`** loop will do two steps at each node; print the data field, then advance the `temp` reference.

   c. The statement, `temp = temp.getNext()`, is a very important one as this is how we move to the next node.

2. Another common list traversal problem is counting the number of nodes in the list. The lab exercise will ask you to solve this problem.

E. Pitfalls of Linked Data Structures

1. A linked list must end with a **`null`** value. Without such a marker at the end of the list, a routine cannot "see" the end of the data structure. This assignment of a **`null`** value at the end of the list is often taken care of when a new node is packaged or through the use of a constructor.

2. When a reference variable is **`null`**, it is a programming error to invoke one of its methods or to try to access one of its instance variables. For example, a program may maintain a reference to the first node of a linked list, as follows:

```
ListNode first;
```

Initially, the list is empty and `first` is the null reference. At this point, it is a programming error to invoke one of the `ListNode`'s methods. The error would occur as a `NullPointerException`.

SUMMARY/ REVIEW: You have just been taught your first dynamic data structure, a linked list. The concept of indirection makes dealing with references a bit more difficult, but careful reading and lots of diagrams will help. Following a working program is a good start; but only by writing code will you develop proficiency with lists. The lab exercises for the next few lessons will enhance your understanding of linked lists.

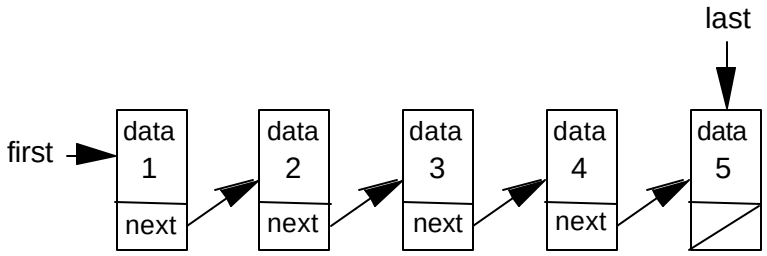
ASSIGNMENT: Lab Exercise L.A.30.1, *List1*

LAB EXERCISE

List1

Background:

1. The student outline contains a program that stored a linked list of five nodes, but in reverse order. The objective of this lab exercise is to create a linked list with the nodes assembled in order as they are generated. If a loop is used to create the values 1-5, the resulting list should look like this:



   The next value to be inserted would go at the right end of the list. To avoid traversing the entire list each time, two references will provide access at the front (first = left end) and the end (last = right end) of the list.

2. Your lab work can begin with the example programs from *ListDemo.java*, *SinglyLinkedList.java*, and *ListNode.java*.

3. An additional instance variable, `last`, should be added to the `SinglyLinkedList` class (found in *SinglyLinkedList.java*) to supply direct access to the last node in the list. The constructor should also be modified to initialize the variable.

4. Create an `addLast` method to create and add a `ListNode` (found in *ListNode.java*)to the end of the list. The method will have to deal with several cases this time, an empty list case and a general case. A two case solution is required because the links are different for each case. The following code/pseudocode is provided:

```
void addLast(Object value)
// Takes in value, creates a new node, adds the new node
// at the end of the list.
{
  if an empty list then
    set both first and last to reference the newly constructed node
  else
    construct a new node at the end of the list
}
```

5. Add a `getLast` method to return the last element of the list.

6. Add a `size` method that returns a count of the number of nodes in the list

7. Adjust the code in the other routines as needed.

Assignment:

1. Using the guidelines in the Background section above, code a linked list that stores the 20 integers from 1-20 in ascending order.

2. After the list is created, call `getLast` display the contents of the last node to the screen.

3. Call `printList` to traverse the list and print out the 20 numbers in one line on the screen.

4. Use the `size` method to display the number of nodes in the list.

Instructions:

1. Turn in your source code and a printed run output.

2. The run output will consist of the value of the first node in the list, the value of the last node in the list, 20 numbers and a count of how many nodes are in the list. An example run output is shown below:

```
First Element: 1
Last Element: 20
SinglyLinkedList: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Nodes: 20
```