Linked Lists

Chris Kiekintveld
CS 2401 (Fall 2010)
Elementary Data Structures and Algorithms
Motivation

- Suppose I have an array: 1,4,10,19,6
- I want to insert a 7 between the 4 and the 10
- What do I need to do?
Linked Lists
aka “Reference-Based Lists”

♦ Linked list
  ♦ List of items, called nodes
  ♦ The order of the nodes is determined by the address, called the link, stored in each node

♦ Every node (except the last node) contains the address of the next node

♦ Components of a node
  ♦ Data: stores the relevant information
  ♦ Link: stores the address of the next node
Linked Lists (continued)

- Head or first
  - Holds the address of the first node in the list
- The info part of the node can be either a value of a primitive type or a reference to an object
Linked Lists (continued)

- **Class Node**
  - Represents nodes on a list
  - It has two instance variables
    - `info` (of type `int`, but it can be any other type)
    - `link` (of type `Node`)

```java
public class Node {
    public int info;
    public Node link;
}
```
Linked List: Some Properties

Figure 16-4 Linked list with four nodes

Table 16-1 Values of head and some of the nodes of the linked list in Figure 16-4

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>head</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>head.info</td>
<td>17</td>
<td>Because head is 2000 and the info of the node at location 2000 is 17</td>
</tr>
<tr>
<td>head.link</td>
<td>2800</td>
<td></td>
</tr>
<tr>
<td>head.link.info</td>
<td>92</td>
<td>Because head.link is 2800 and the info of the node at location 2800 is 92</td>
</tr>
</tbody>
</table>
Linked List: Some Properties

♦ Now consider the statement current = head;

Figure 16-5 Linked list after current = head; executes

Table 16-2 Values of current and some of the nodes of the linked list in Figure 16-5

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>current</td>
<td>2000</td>
</tr>
<tr>
<td>current.info</td>
<td>17</td>
</tr>
<tr>
<td>current.link</td>
<td>2800</td>
</tr>
<tr>
<td>current.link.info</td>
<td>92</td>
</tr>
</tbody>
</table>
Linked List: Some Properties

Now consider the statement

```java
current = current.link;
```

Figure 16-6 List after the statement `current = current.link;`

Table 16-3 Values of `current` and some of the nodes of the linked list in Figure 16-6

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>current</code></td>
<td>2800</td>
</tr>
<tr>
<td><code>current.info</code></td>
<td>92</td>
</tr>
<tr>
<td><code>current.link</code></td>
<td>1500</td>
</tr>
<tr>
<td><code>current.link.info</code></td>
<td>63</td>
</tr>
</tbody>
</table>
# Linked List: Some Properties

![Linked List Diagram](image)

## Table 16-4  Values of various reference variables and nodes of the linked list in Figure 16-6

<table>
<thead>
<tr>
<th>Reference Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>head.link.link</td>
<td>1500</td>
</tr>
<tr>
<td>head.link.link.info</td>
<td>63</td>
</tr>
<tr>
<td>head.link.link.link</td>
<td>3600</td>
</tr>
<tr>
<td>head.link.link.link.info</td>
<td>45</td>
</tr>
<tr>
<td>current.link.link</td>
<td>3600</td>
</tr>
<tr>
<td>current.link.link.info</td>
<td>45</td>
</tr>
<tr>
<td>current.link.link.link</td>
<td>null</td>
</tr>
<tr>
<td>current.link.link.link.link.info</td>
<td>Does not exist</td>
</tr>
</tbody>
</table>
Traversing a Linked List

‣ Basic operations of a linked list that require the link to be traversed
  ‣ Search the list for an item
  ‣ Insert an item in the list
  ‣ Delete an item from the list

‣ You cannot use head to traverse the list
  ‣ You would lose the nodes of the list
  ‣ Use another reference variable of the same type as head: current
Traversing a Linked List

The following code traverses the list

current = head;
while (current != null) {
    //Process current
    current = current.link;
}

Exercise

- Write code to print out the data stored in each node in a linked list

```java
    current = head;
    while (current != null)
    {
        System.out.println(current.info + " ");
        current = current.link;
    }
```
Exercise

- Write code to set the data in the $5^{th}$ node to be 10

```java
current = head;
cnt = 0;
while (cnt < 4 && current != null) {
    current = current.link;
}
if (current != null) {
    current.info = 10;
}
```
Insertion

♦ Consider the following linked list

![Linked list before item insertion](image16-7.png)

**Figure 16-7** Linked list before item insertion

♦ You want to create a new node with \texttt{info 50} and insert it after \texttt{p}
Insertion

- The following statements create and store 50 in the `info` field of a new node

```java
Node newNode = new Node();  // create newNode
newNode.info = 50;          // store 50 in the new node
```

Figure 16-8 Create newNode and store 50 in it
Insertion

- The following statements insert the node in the linked list at the required place
  
  ```java
  newNode.link = p.link;
  p.link = newNode;
  ```

- The sequence of statements to insert the node is very important
Insertion (continued)

**Figure 16-9** List after the statement

```
newNode.link = p.link; executes
```

**Figure 16-10** List after the statement

```
p.link = newNode; executes
```
Deletion

♦ Consider the following linked list

![Diagram of a linked list with nodes 45, 65, 34, 76, and a pointer to 34]

**Figure 16-15** Node to be deleted is with info 34

♦ You want to delete node with info 34
Deletion (continued)

- The following statement removes the nodes from the list

\[ p\.link = p\.link\.link \]

*Figure 16-16* List after the statement

\[ p\.link = p\.link\.link; \text{ executes} \]
Deletion (continued)

- Previous statement removed the node
  - However, the memory may still be occupied by this node
- System’s automatic garbage collector reclaims memory occupied by unreferenced nodes
  - Use `System.gc();` to run the garbage collector
Building a Linked List

- You can build a list in two ways: forward or backward

- Forward manner
  - A new node is always inserted at the end of the linked list

- Backward manner
  - A new node is always inserted at the beginning of the linked list
Building a Linked List Forward

- You need three reference variables
  - One to point to the front of the list
    - Cannot be moved
  - One to point to the last node of the list
  - One to create the new node
- Next two slides show the code for creating a linked list forward
Building a Linked List Forward (continued)

Node buildListForward()
{
    Node first, newNode, last;
    int num;
    System.out.println("Enter integers ending with -999:");
    num = console.nextInt();
    first = null;
    while (num != -999)
    {
        newNode = new Node();
        newNode.info = num;
        newNode.link = null;
Building a Linked List Forward (continued)

```java
if (first == null)
    {
        first = newNode;
        last = newNode;
    }
else
    {
        last.link = newNode;
        last = newNode;
    }
num = console.nextInt();
}//end while
return first;
}//end buildListForward
```
Building a Linked List Backward

- You only need two reference variables
  - One to point to the front of the list
    - Changes each time a new node is inserted
  - One to create the new node
- Next slide shows the code for creating a linked list backward
Node buildListBackward() {
    Node first, newNode;
    int num;
    System.out.println("Enter integers ending with -999:");
    num = console.nextInt();
    first = null;
    while (num != -999) {
        newNode = new Node();     //create a node
        newNode.info = num;       //store the data in newNode
        newNode.link = first;     //put newNode at the beginning of the list
        first = newNode;          //update the head of the list,
        num = console.nextInt();  //get the next number
    }
    return first;
}//end buildListBackward
Exercise

- Write code to take in an array of ints and returns the head reference to a linked list of the ints

```java
public Node createLinkedList(int[] a) {

}
```
Write code to take in an array of ints and returns the head reference to a linked list of the ints

```java
public Node createLinkedList(int[] a) {
    Node head = new Node();
    head.info = a[length-1];
    head.link = null;

    for (int i = a.length-2; i >=0; i--) {
        Node n = new Node();
        n.info = a[i];
        n.link = head.link;
        head = n;
    }
    return head;
}
```
Linked Lists

Chris Kiekintveld
CS 2401 (Fall 2010)
Elementary Data Structures and Algorithms
Abstract Data Types (ADT)

- A set of data
- A set of operations that can be done on the data

- In Java, implemented using interfaces
  - Methods are abstract
  - Cannot create an instance of an ADT
  - Need a concrete implementation
  - Can be multiple implementations!
Example: Lists

- A list is an ordered sequence of elements of a single type of data
- There are multiple ways to implement lists
  - Array-Based lists (Vectors)
  - Reference-Based lists (Linked Lists)
Using an ADT

public int sum(List<Integer> myList)
{
    int tempSum = myList.get(0);
    for (int i=1; i <= myList.size(); i++) {
        tempSum+= myList.get(i);
    }
    return tempSum;
}

List<Integer> myList = new Vector<Integer>();
l.add(3);
l.add(5);
l.add(5);

System.out.println(sum(myList));
To use a Linked List Instead...

```java
public int sum(List<Integer> myList){
    int tempSum = myList.get(0);
    for (int i=1; i <= myList.size(); i++) {
        tempSum+= myList.get(i);
    }
    return tempSum;
}

List<Integer> myList = new LinkedList<Integer>();
l.add(3);
l.add(5);
System.out.println(sum(myList));
```
Linked List Iterators

- An iterator is an object that produces each element of a collection one element at a time
- An iterator has at least two methods: `hasNext` and `next`
  - `hasNext`: Determines whether there is a next element in the collection
  - `next`: Gives access to the next element in the list
Iterators in Java: Typical Example

```java
List<Integer> tmp = new Vector<Integer>();
tmp.add(7);
tmp.add(4);

Iterator itr = tmp.iterator();
while(itr.hasNext()) {
    int num = itr.next();
    System.out.println(num);
}
```
Doubly Linked Lists

- Linked list in which every node has a next pointer and a back pointer
- A doubly linked list can be traversed in either direction

Figure 16-48 Doubly linked list
Doubly Linked List Nodes

♦ **Class** DoublyLinkedListNode

```java
public class DoublyLinkedListNode<T> implements Cloneable {
    T info;
    DoublyLinkedListNode<T> next;
    DoublyLinkedListNode<T> back;

    //place constructors and methods here
}
```
Doubly Linked lists make certain things easier...

```java
public void reversePrint()
{
    DoublyLinkedListNode<T> current; //reference variable to
    //traverse the list

    current = last; //set current to point to the last node
    while (current != null)
    {
        System.out.print(current.info + " ");
        current = current.back;
    }
}
```
Circular Linked Lists

- A linked list in which the last node points to the first node
- It is convenient to make `first` point to the last node

Figure 16-56 Circular linked list with more than one node
Exercise

♦ Write a method swap(List myList, i, j) that interchanges the elements in positions i and j

public void swap(List myList, int i, int j) {
    int itemp = myList.get(i);
    int jtemp = myList.get(j);

    myList.remove(i);
    myList.add(i, jtemp);

    myList.remove(j);
    myList.add(j, itemp);
}

What happens if we first remove both elements, and then add both?
Exercise

- Write a **recursive** method to determine whether a linked list is sorted in descending order or not (return a boolean)

```java
boolean isSorted(Node A){
    if (A == null | a.link == null) {
        return true;
    }
    if (a.info > a.link.info) {
        return false;
    }
    return isSorted(A.link);
}
```