Data Structures and Algorithms

BSc. In Computing

Semester 5 Lecture 2

Lecturer: Dr. Simon McLoughlin

Linked Lists

Preliminaries

- Object references
- Resizable arrays
- Reference-based linked lists

Programming with linked lists

- Displaying the contents of a linked list
- Deleting a specified node from a linked list
- Inserting a node into a specified position of a linked list
- A reference based implementation of the ADT list
- Comparing array based and reference based implementation
- Passing a linked list to a method

Variations of the linked list

- Tail references
- Circular Linked Lists
- Dummy Head Nodes
- Doubly linked lists

This lecture reviews Java references and introduces you to the data structure“linked list”.

We will study algorithms for fundamental linked list operations such as insertion and deletion.

We describe several variations of the basic linked list.

A linked list can be used when implementing many of the ADTs that we will discuss this semester.

The ADT list as described in the previous lecture has operations for insert, delete, and retrieve items, given their positions in the list.

A close examination of the array based implementation of the ADT list shows that an array is not always the best data structure to use to maintain a collection of data.

An array has a fixed size, but the ADT list can have an arbitrary length.

Therefore, we really cannot use an array to implement a list because the number of items on the list may exceed the size of the array.

Also…
While the most intuitive means of imposing an order on data is to sequence it physically, this approach has its disadvantages.

In a physical ordering, the successor of an item **x** is the next data item in sequence after **x**, that is, the item to the right.

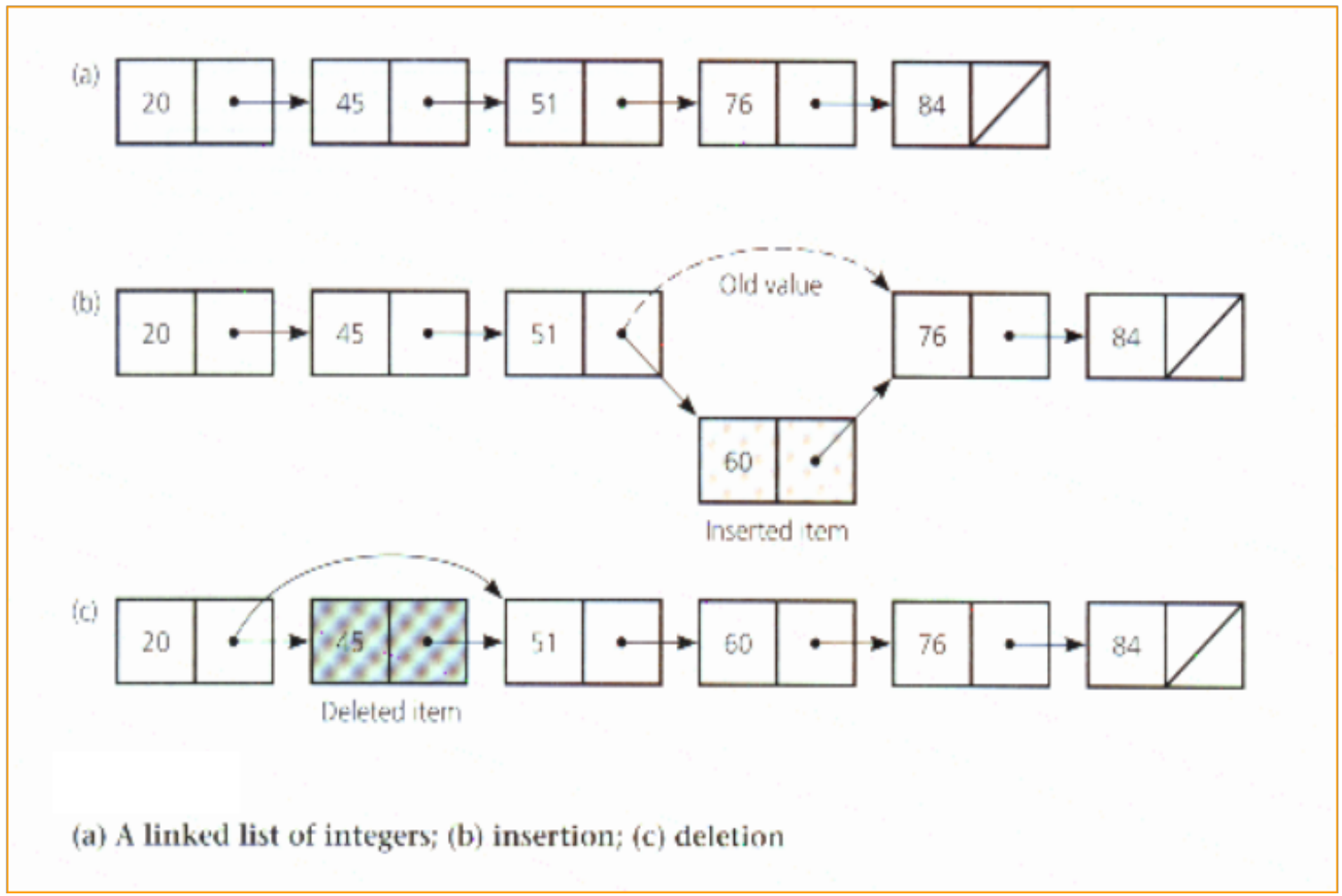
That means that when we insert or delete an item we must physically reorder the items by shifting to the right, or to the left.

Shifting data can be a VERY time consuming process, especially if we need to insert and delete lots of data.

Essentially, an array-based implementation of a list is not much use in a real world situation!

What alternatives are available?

To get a conceptual idea of a list implementation that does not involve shifting, consider the following diagram.



(a) A linked list of integers; (b) insertion; (c) deletion

In the diagrams, each item in the list is actually linked to the next item

- Therefore, if you know where an item is, you can determine its successor, which can be anywhere physically.

This flexibility not only allows you to insert and delete data items without shifting data, but it also allows you to increase the size of the list easily.

A linked list can grow as needed

To insert a new item, simply find its place in the list and set two links

To delete an item, find the item and change a link to bypass the item

Because the items are linked to one another, this data structure is called a **linked list**

Object References

Before we examine linked lists and their use in the implementation of an ADT, we need to examine how Java references can be used to implement a link list.

- Java allows an object to reference another, and we can use this ability to build a linked list.

- When you declare a variable that refers to an object of a given class, you are creating a reference to the object.

Remember that an object reference does not come into existence until you apply the new operator.

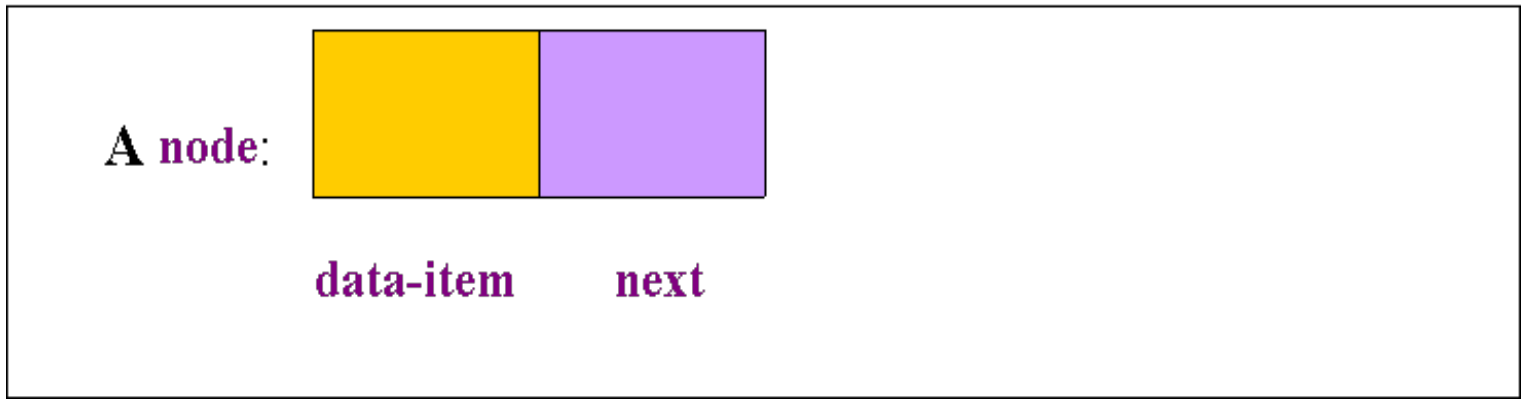
- A reference variable, or a re ference, contains the location or address in memory of an object.

By using a reference to a particular object, you can locate the object and access the objects' (public) data

Reference-based Linked Lists

A linked list contains components that are linked to one another.

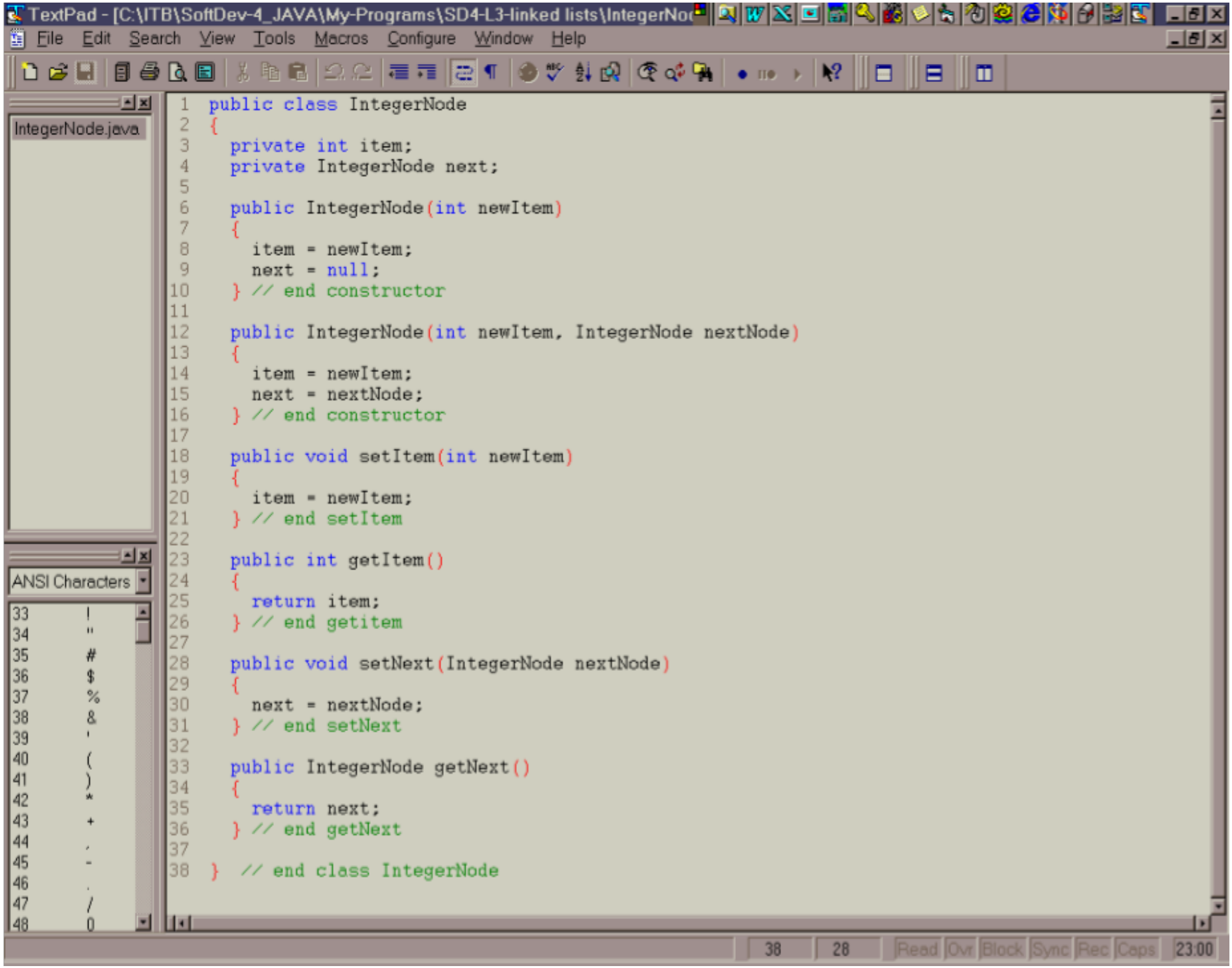
Each component, called a node, contains both data and a "link" to the next item.



Such links are Java references variables.

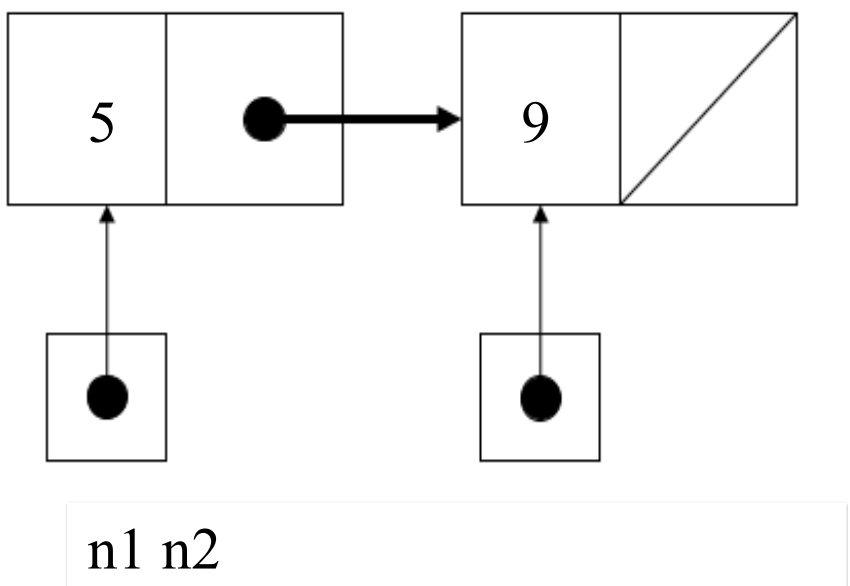
Each node in a list can be implemented as an object.

For example:
If you want to create a linked list of integers, you could use the following class definition.


TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L3-linked lists\IntegerNo
File Edit Search View Tools Macros Configure Window Help
IntegerNode.java
ANSI Characters
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
1 public class IntegerNode
2 {
3 private int item;
4 private IntegerNode next;
5
6 public IntegerNode(int newItem)
7 {
8 item = newItem;
9 next = null;
10 } // end constructor
11
12 public IntegerNode(int newItem, IntegerNode nextNode)
13 {
14 item = newItem;
15 next = nextNode;
16 } // end constructor
17
18 public void setItem(int newItem)
19 {
20 item = newItem;
21 } // end setItem
22
23 public int getItem()
24 {
25 return item;
26 } // end getitem
27
28 public void setNext(IntegerNode nextNode)
29 {
30 next = nextNode;
31 } // end setNext
32
33 public IntegerNode getNext()
34 {
35 return next;
36 } // end getNext
37
38 } // end class IntegerNode
38
28
Read
Ovr
Block
Sync
Rec
Caps
23:00

We can use the above class as follows:

```
IntegerNode n1  = new IntegerNode();
IntegerNode n2 = new IntegerNode();

n1.setItem(5);          // set data item in first node
n2.setItem(9);          // set data item in second node

n1.setNext(n2);         // link the nodes
```



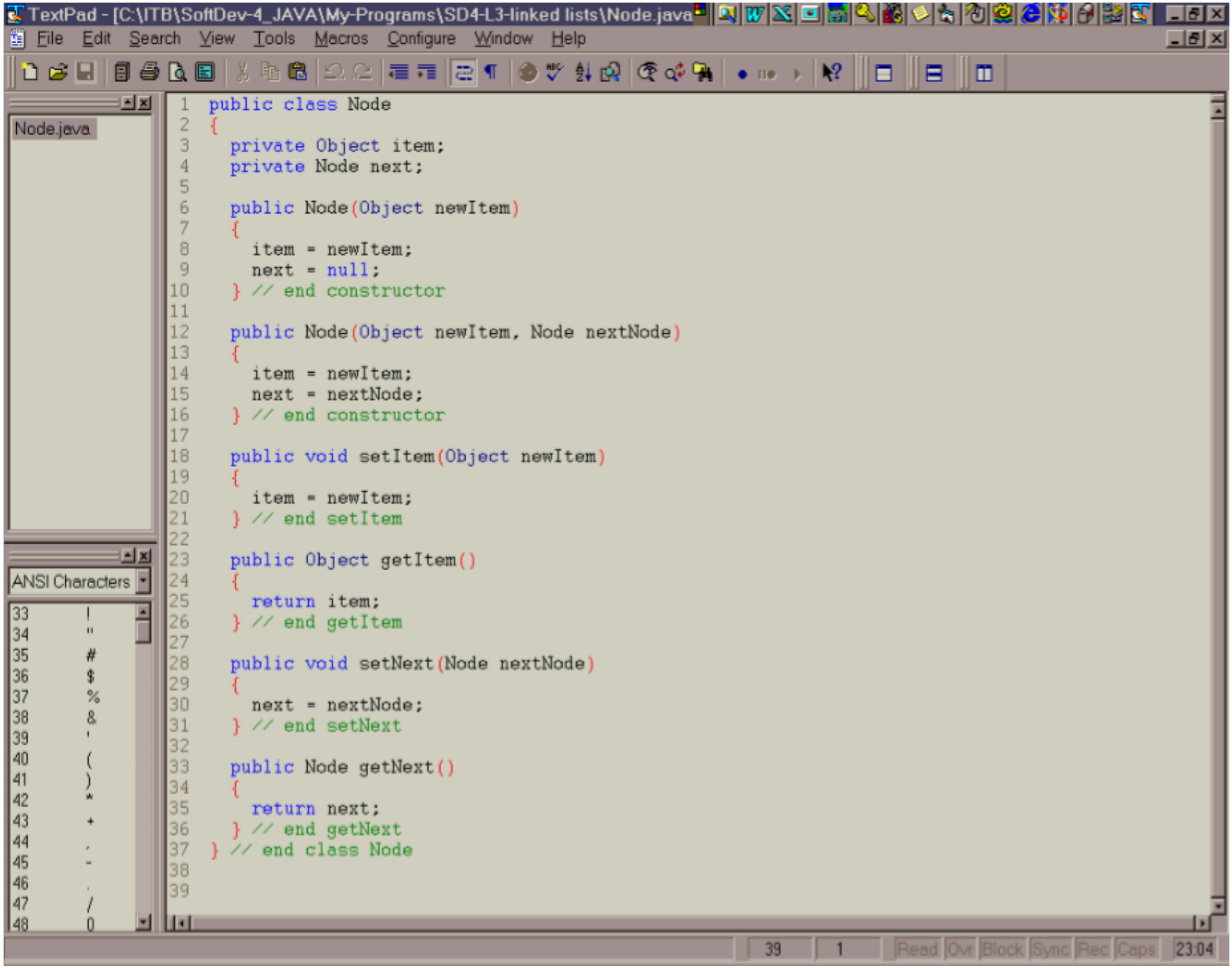
The result of linking two instances of**IntegerNode**

This definition of a node restricts the data to a single integer field.

- Since we would like to have this class be as reusable as possible, it would be better to change the data field to be of **type Object**.

In Java, every class is ultimately derived from t he class Object through inheritance.

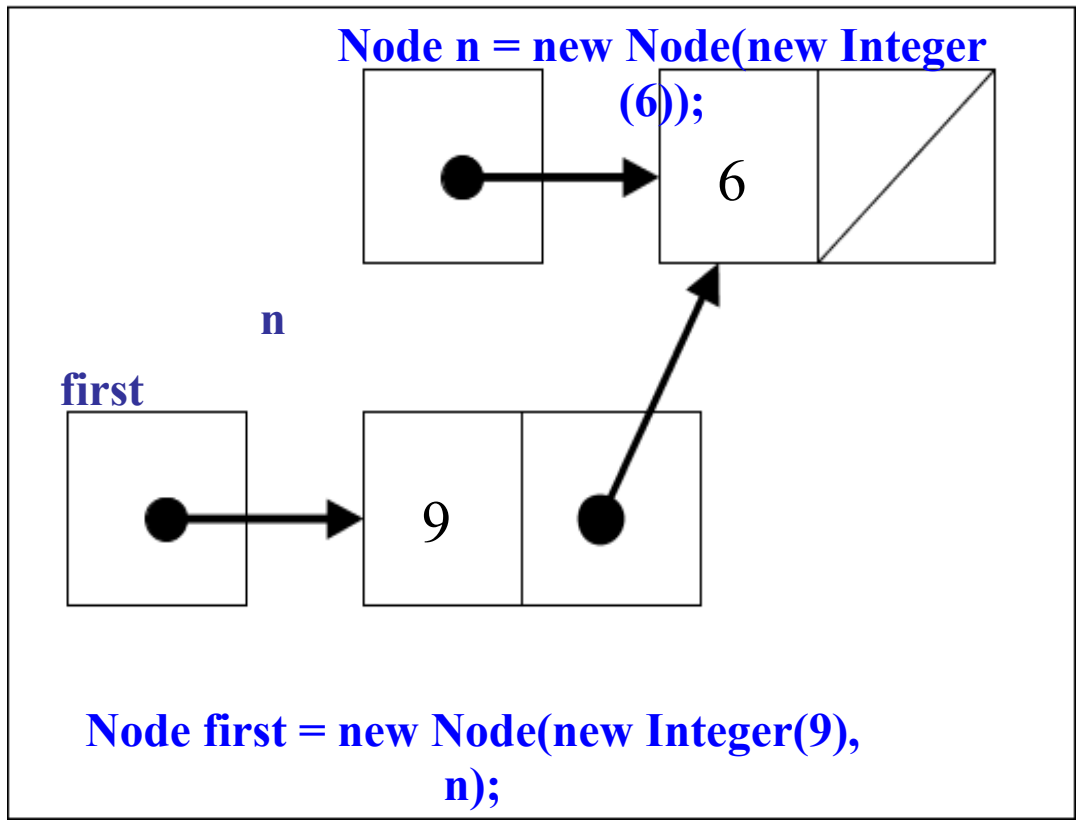
- This means that any class created in Java could use this node definition for storing objects.


TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L3-linked lists\Node.java]
File Edit Search View Tools Macros Configure Window Help
Node.java
ANSI Characters
public class Node
{
private Object item;
private Node next;
public Node(Object newItem)
{
item = newItem;
next = null;
} // end constructor
public Node(Object newItem, Node nextNode)
{
item = newItem;
next = nextNode;
} // end constructor
public void setItem(Object newItem)
{
item = newItem;
} // end setItem
public Object getItem()
{
return item;
} // end getItem
public void setNext(Node nextNode)
{
next = nextNode;
} // end setNext
public Node getNext()
{
return next;
} // end getNext
} // end class Node
39
1
Read Ovr Block Sync Rec Caps
23:04

We can use this class as follows

```
Node n = new Node(newInteger(6));
Node first = new Node(newInteger(9), n);
```

What does this look like?



Using the Node constructor to initialise a data field and a link value

The constructors are used to initialise the data field and the link field.

Although the data portion of each node in a link list can reference any object, the figure illustrates data items that are concerned with Integer objects.

What is the value of the data field ***next*** in the last node in the list?

- By setting this field to **null**, we can easily detect when we have reached the end of the linked list.

Where can we find the start of the list?

- As we have described the linked list so far, nothing references the beginning of the list.

If you cannot get to the beginning of the list, you will not be able to progress in walking the list.

We therefore need an additional reference variable to help us locate the first node in the list.

This is called the **linked list head**.

- The reference variable head does **not** exist within one of the nodes.

head is a simple reference variable that is **external** to the linked list

head enables you to access the start of he linked list.

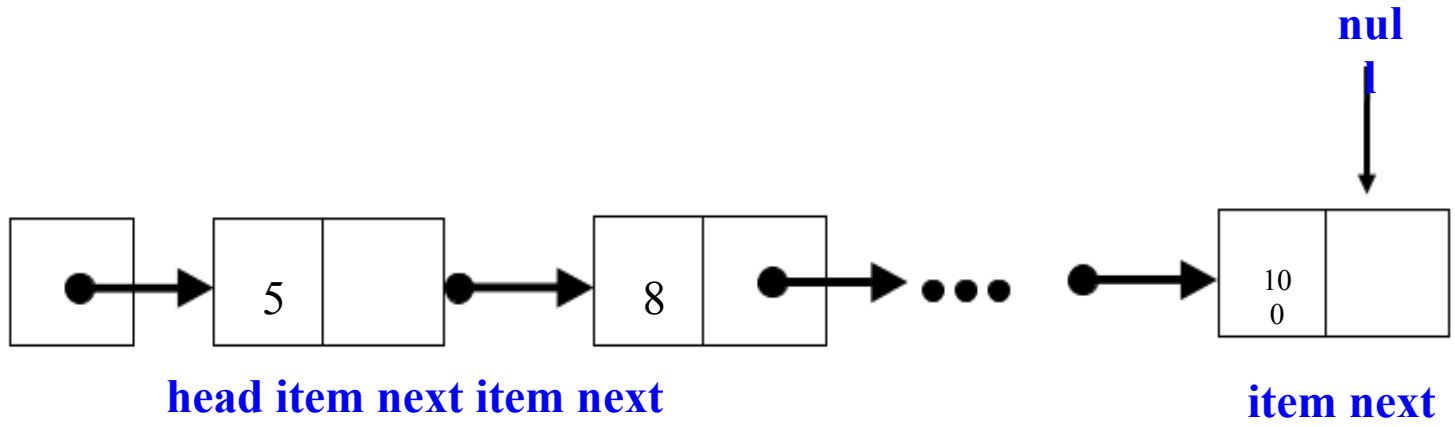
Note that **head** always exists, even if there are no nodes in the linked list.

The statement :

Node head = null;

Creates that variable **head** whose value is initially null.

This indicates that **head** does not reference anything and that the list is empty.



A head reference to a linked list

Head = new Node(new Integer (5));
head = null;

Programming with linked lists

We can develop algorithms to

- **display** the data portions of a linked list
- **insert** items into a linked list
- **deleting** items from a linked list

The linked list operations are the basis for many of the other data structures and will therefore be of great use to us.

Displaying the contents of a Linked List

Let us assume that we now have a linked list and want to display the data in the list.

A high-level pseudocode solution is:

Let a variable **curr** reference the first node in the linked list.

```
while ( the curr reference is not null)
{
      display the data portion of the current node
      set the current reference to the next field of thecurrent node
}//end while
```

The solution requires that we keep track of the current position within the linked list.

The reference variable **curr** references the current node

Initially **curr** must reference the first node so we make it equal to head

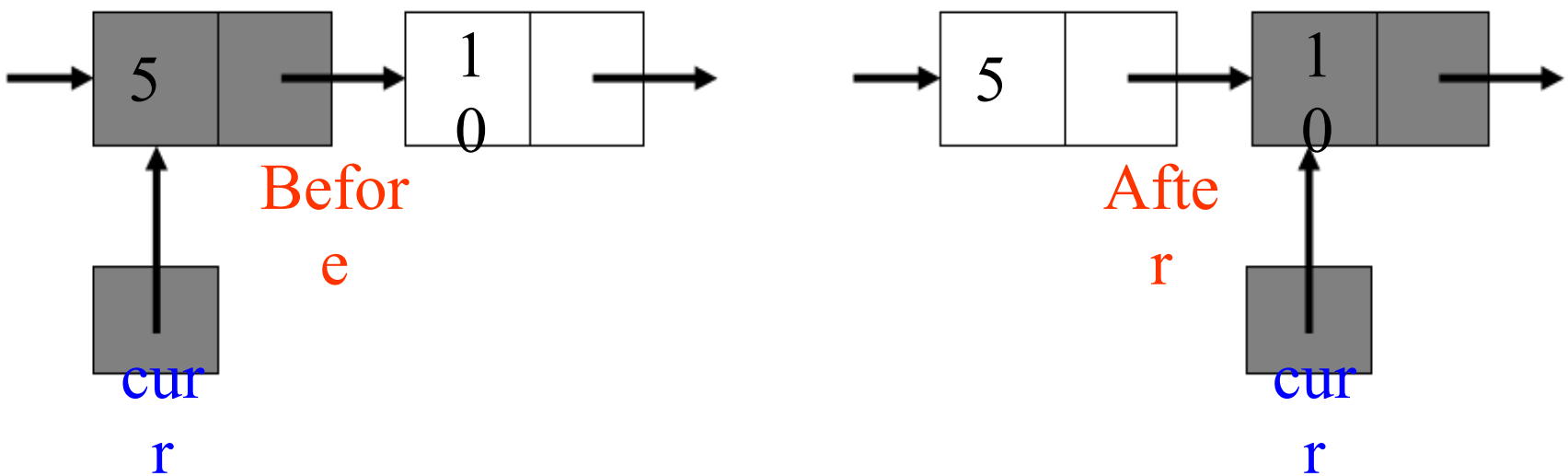
```
Node curr = head;
```

To display the data portion of the current node, we use the Java statement:

```
System.out.println(curr.getItem() );
```

To advance the current position to the next node use the Java statement:

```
curr = curr.getNext();
```



We can use a loop such as:

```
// display the data in a linked list that head references
// Loop invariant: curr references the next node to be displayed
for (Node curr = head; curr != null; curr = curr.getNext() )
{
     System.out.println(curr.getItem() );
} //end for
```

The variable **curr** references each node in a nonempty linked list during the course of the for loops execution.

The data item is displayed

After the last node is displayed, **curr** becomes **null** and the loop terminates.

Note:

- Do not compare **curr.getNext()** with **null** instead of **curr**. When **curr** references the last node of the nonempty linked list, **curr.getNext ()** is **null** and the loop would terminate before displaying the data in the last node.

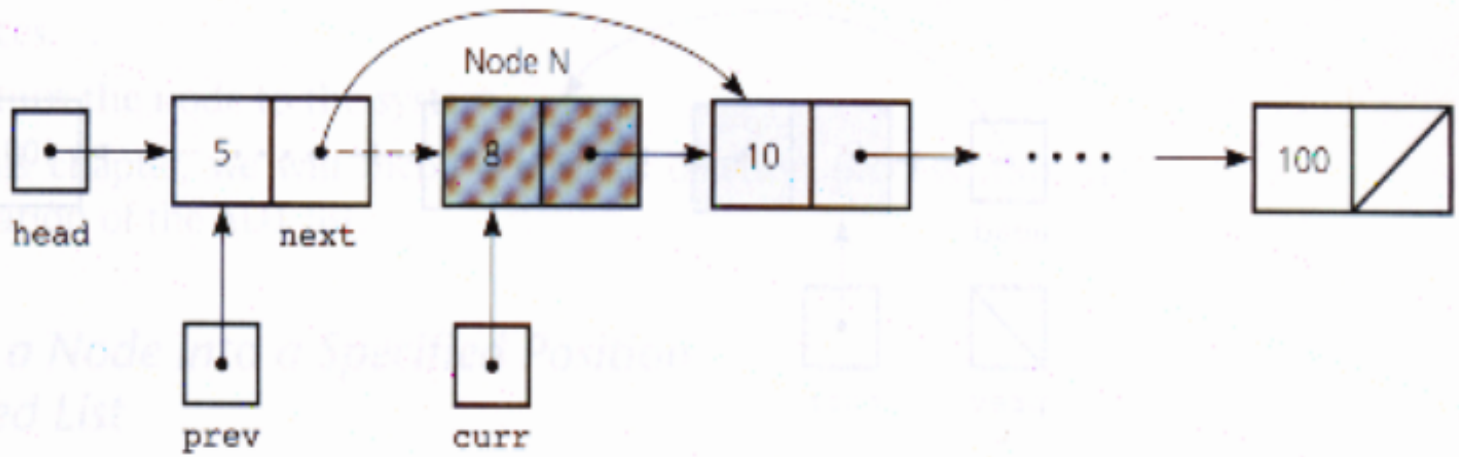
- Also, when the list is empty, that is, when **head** and **curr** are **null** then **curr.getNext()** will throw a **NullPointerException**

List Traversal

Displaying a linked list is an example of a common operation, that of **List Traversal**.

- A traversal sequentially visits each node in the list until it reaches the end of the list.
- Our example visits each node in the list to display its dat a item until it reaches the end of the list.
- Displaying a linked list does not alter the list in any way.

Deleting a Specified Node from a Linked List



Deleting a node from a linked list

Notice in the diagram that, in addition to **head**, two reference variables are used, **curr** and **prev**.

The task is to delete the node that**curr** references.

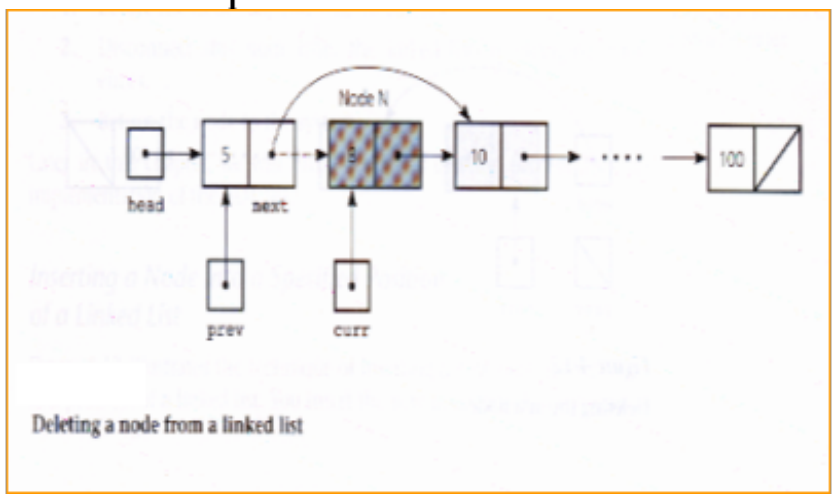
We need **prev** to help with this operation.

We can delete a **node N**, which **curr** references by altering the value of the reference **next** in the node that precedes **node N**.

We need to set this field to reference the node that follows N thereby bypassing N on the chain.

The assignment statement that follows is all that is required:

prev.setNext(curr.getNext());



Deleting a node from a linked list

We need something different if the node in question is the first node.

In this situation, **prev** cannot reference the node that preceded the first node as, by definition, there is none.

Deletion of the first node is a special case.

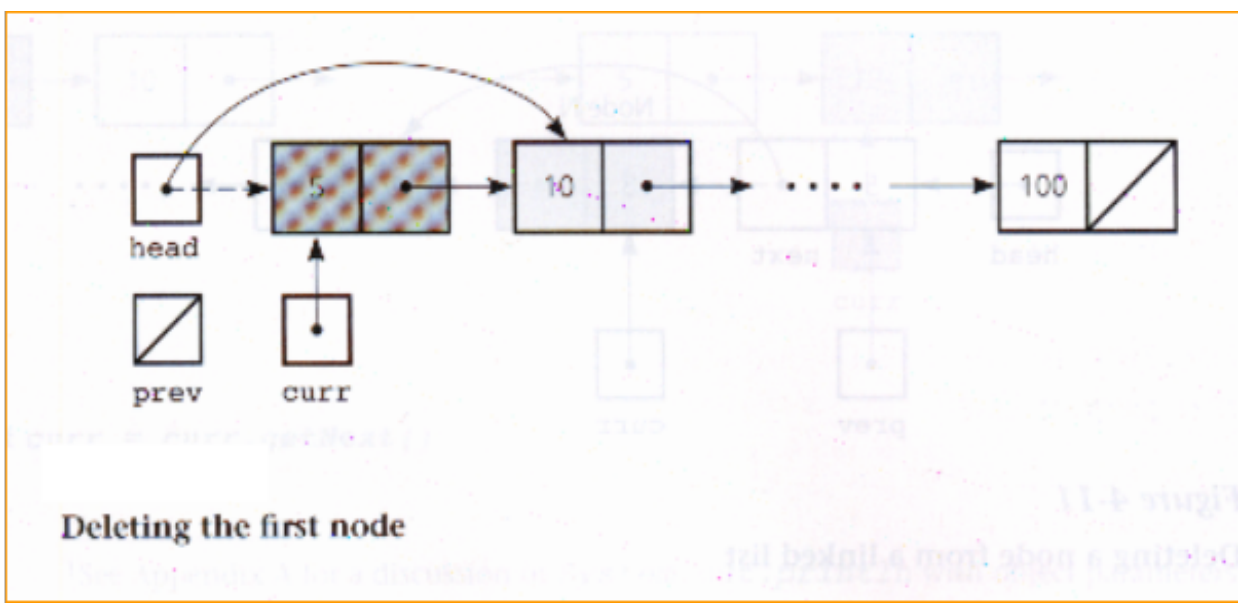
In this case, **curr** references the **first** node and **prev** is null.

When you delete the first node of the list, the value of head must be changed to reflect the fact that, after the deletion, the list has a new first node.

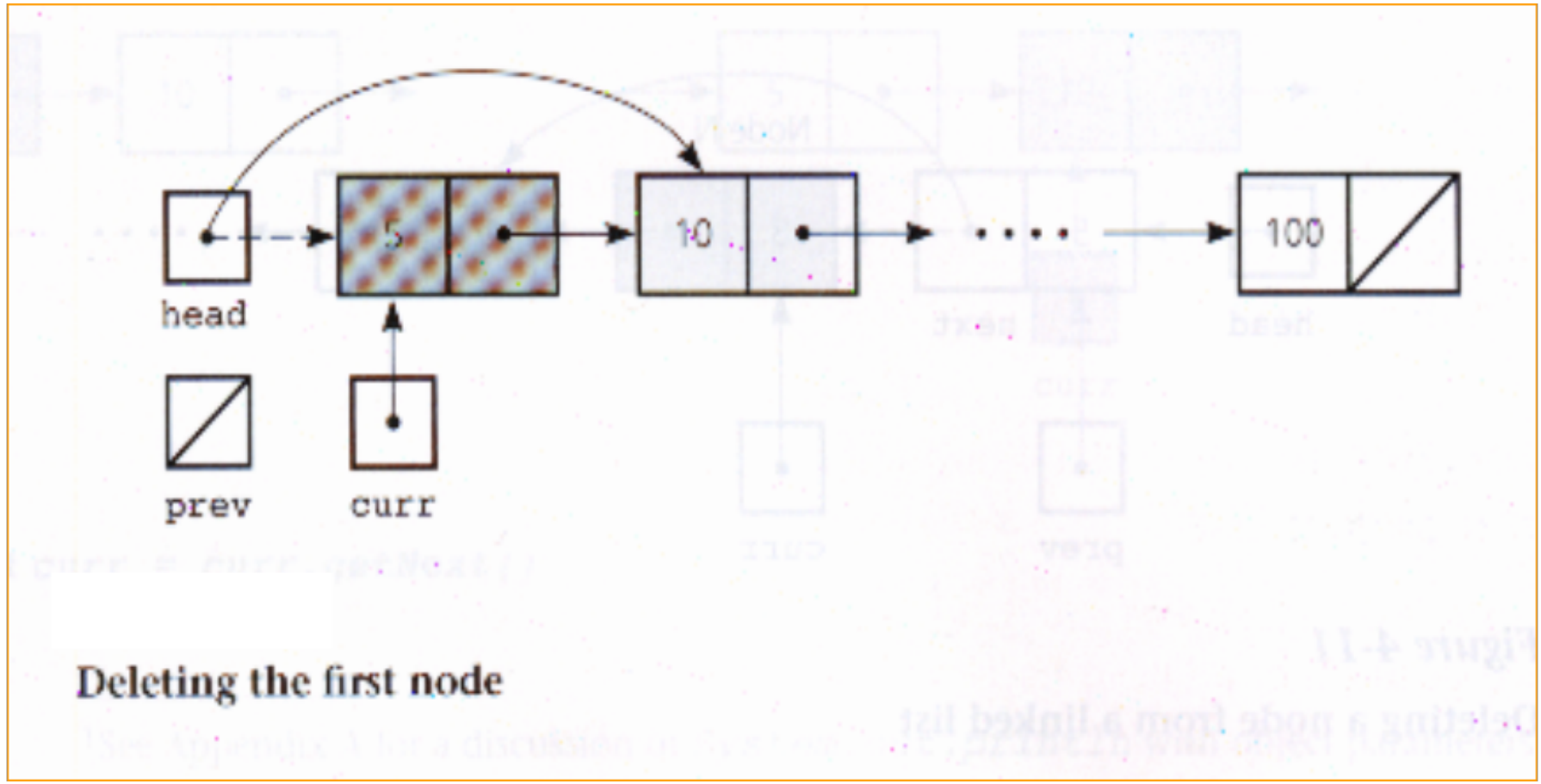
That is, the node that was second prior to the deletion is now first.

We achieve this by:

head = head.getNext();



Deleting the first node



Deleting the first node

The **deletion process** has three high level steps

1. **Locate** the node that you want to delete

2. **Disconnect** this node from the linked list by changing references

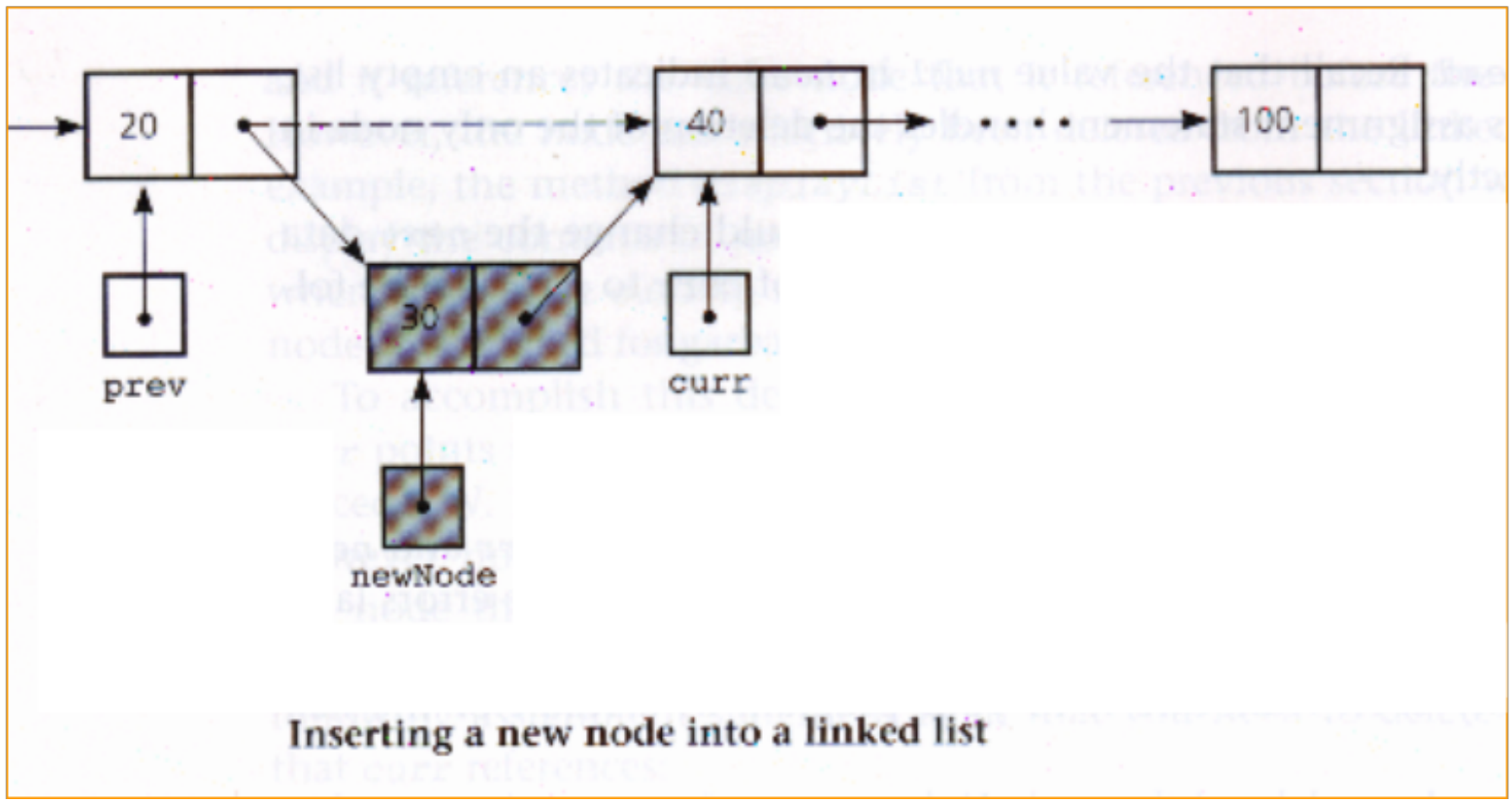
3. **Return** the node to the system

Inserting a Node into a Specified Position of a Linked List

We insert the node, which the reference variable **newNode** references, between the two nodes that **prev** and **curr** reference

We can achieve the insertion by using:

```
newNode.setNext(curr);
prev.setNext(newNode);
```



Inserting a new node into a linked list

Where does newNode come from, … and go?

Establish **prev** and **curr** by walking the linked list until you find the correct position in the list for the new item.

Use the new operator to create a new node:

newNode = new Node(data item);

We can now insert the **newNode** into the list as previously described.

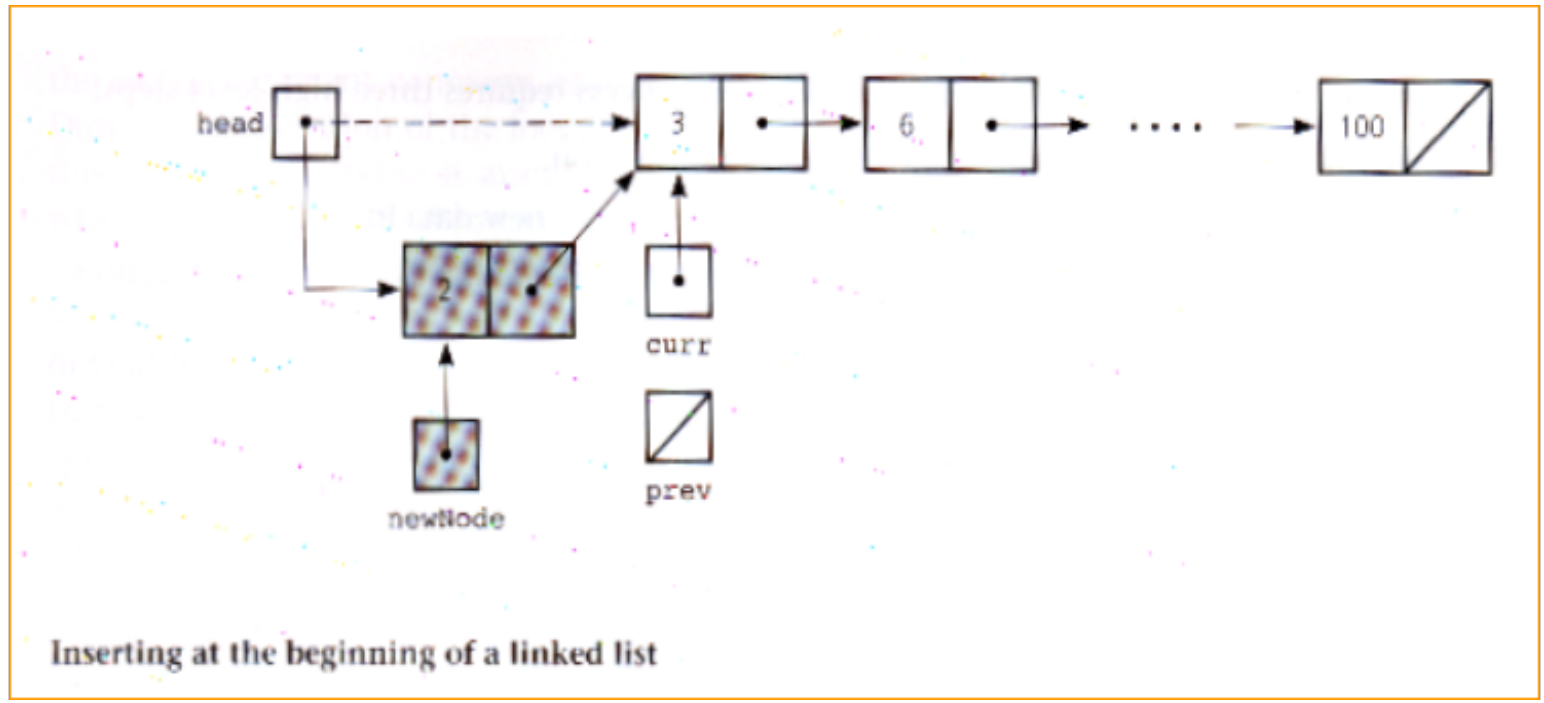
Special Case

Insertion into a linked list has a special case, that is, at the beginning of the list.

To achieve this,

- Make the next field of the newNode reference what head references
- Make head reference the newNode

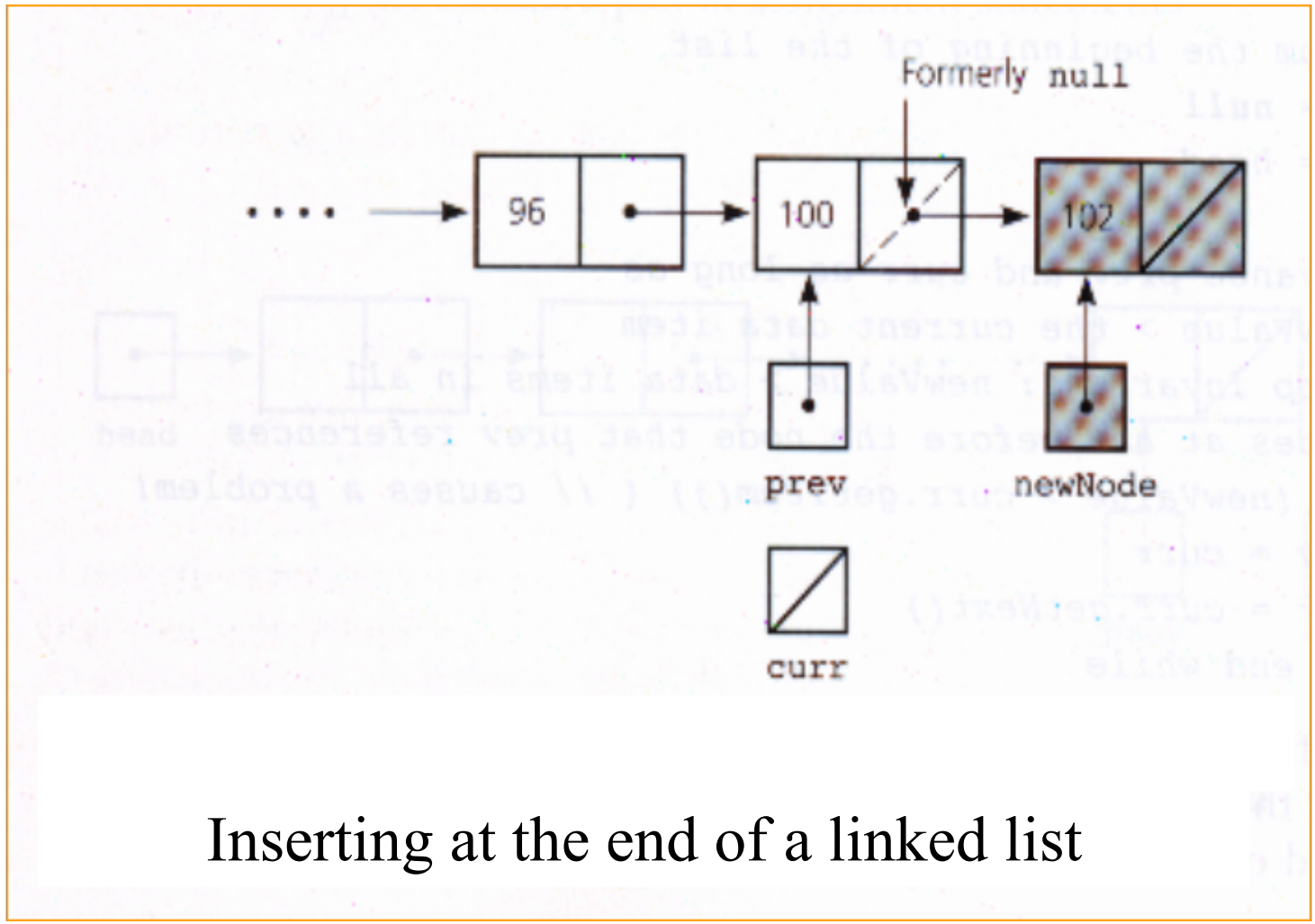
newNode.setNext(head);
head = newNode;



Inserting at the beginning of a linked list

Insertion at the end of a Linked List

```
newNode.setNext(curr);
prev.setNext(newNode);
```



Inserting at the end of a linked list

The insertion process requires three high-level steps

1. **Determine** the point of insertion

2. **Create** a new node and store the new data in it.

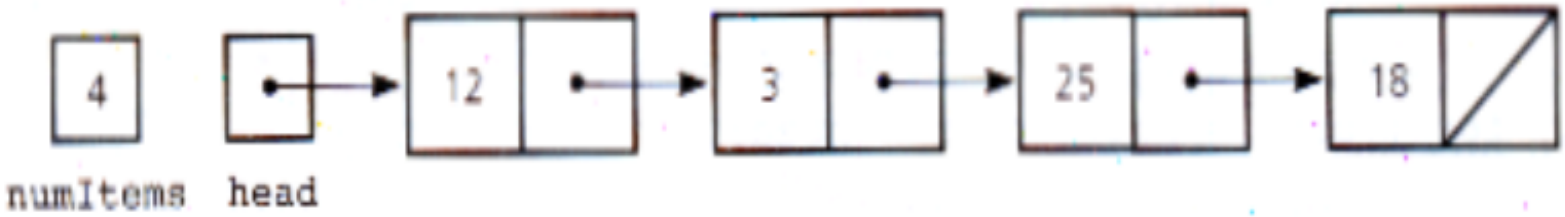
3. **Connect** the new node to the linked list by changing references.

Determining *prev* and *curr*

Pseudocode

```
// determine the point of insertion into the linked list
// initialise prev and curr to start the traversal
// from the beginning of the list

prev = null
curr = head

// Advance prev and curr as long as newValue > the current data item.
// Do not go beyond the end of the list
// Loop invariant: newValue > data items in all nodes
// and before the node that prev references

while (curr != null && newValue > curr.getItem() )
{
      prev = curr
      curr = curr.getNext();
} //end while
```

A reference based implementation of the ADT List



A reference-based implementation of the ADT List

TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L3-linked lists\ListInterfac

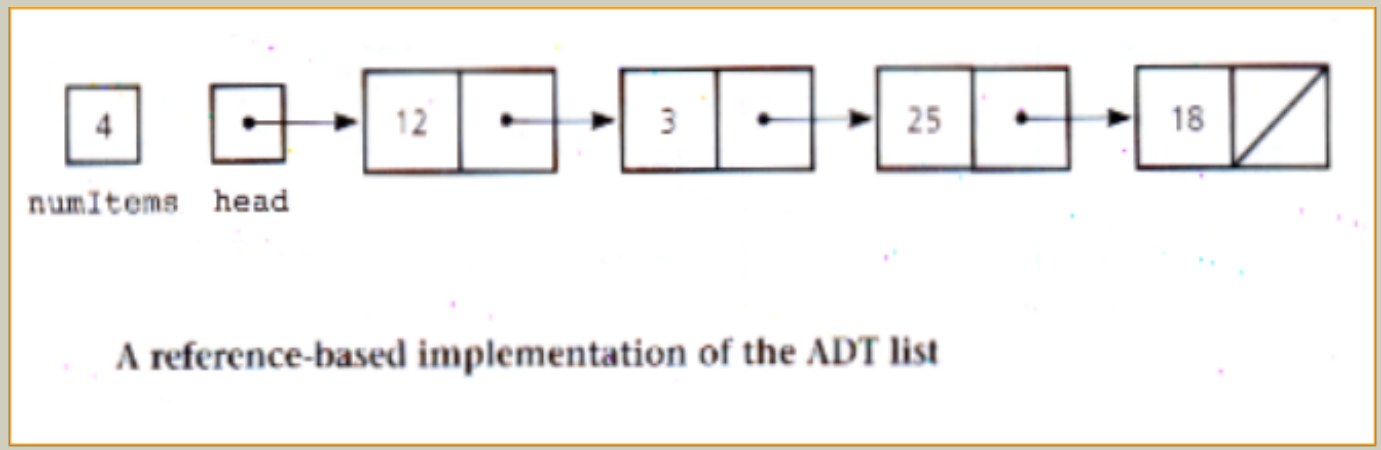
File Edit Search View Tools Macros Configure Window Help

ListInterface.java

```
1  // ********************************************************
2  // Interface for the ADT list
3  // ********************************************************
4
5  public interface ListInterface
6  {
7    // list operations:
8    public boolean isEmpty();
9
10   public int size();
11
12   public void add(int index, Object item) throws ListIndexOutOfBoundsException;
13
14   public void remove(int index) throws ListIndexOutOfBoundsException;
15
16   public Object get(int index)  throws ListIndexOutOfBoundsException;
17
18   public void removeAll();
19
20 } // end ListInterface
```

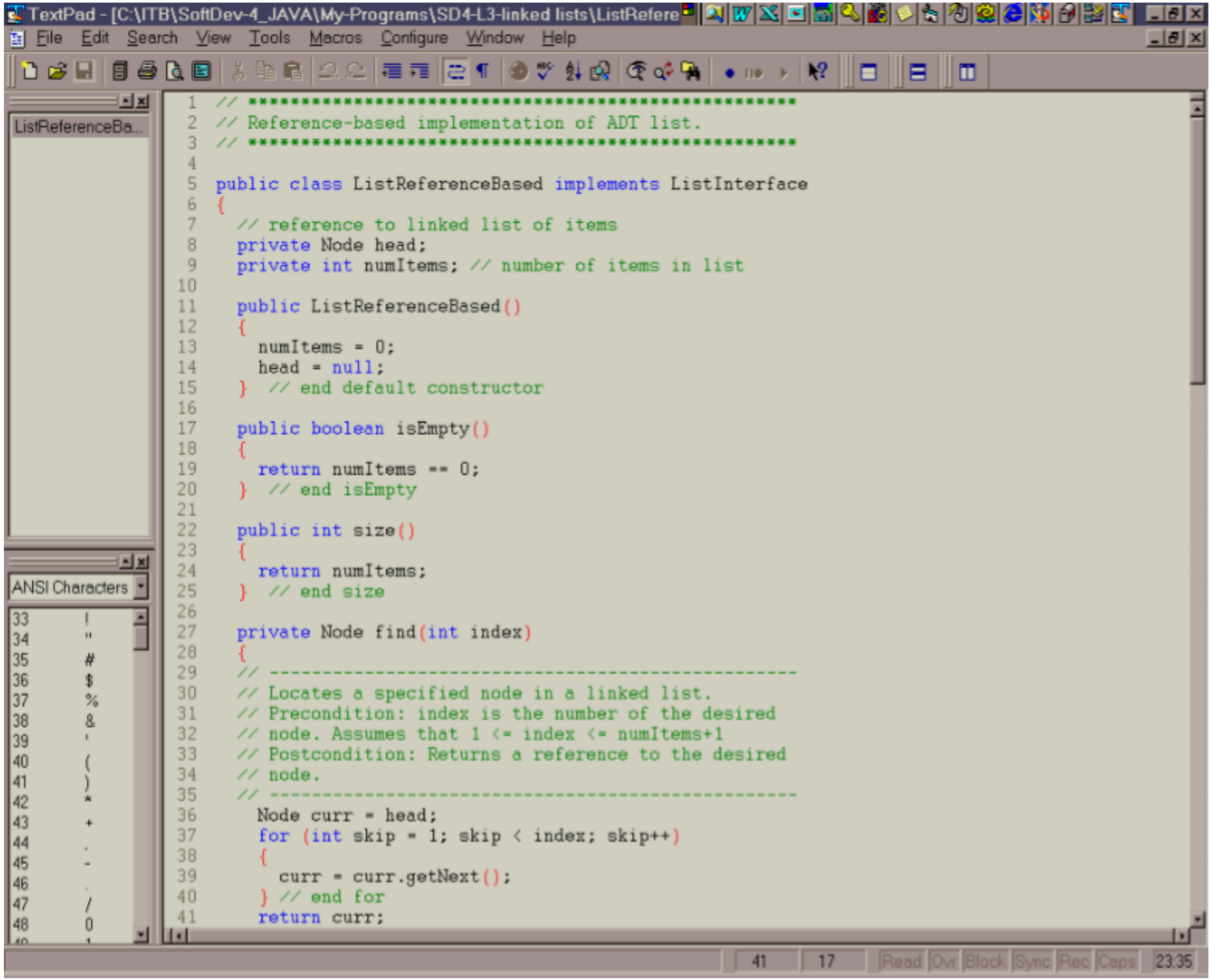
ANSI Characters

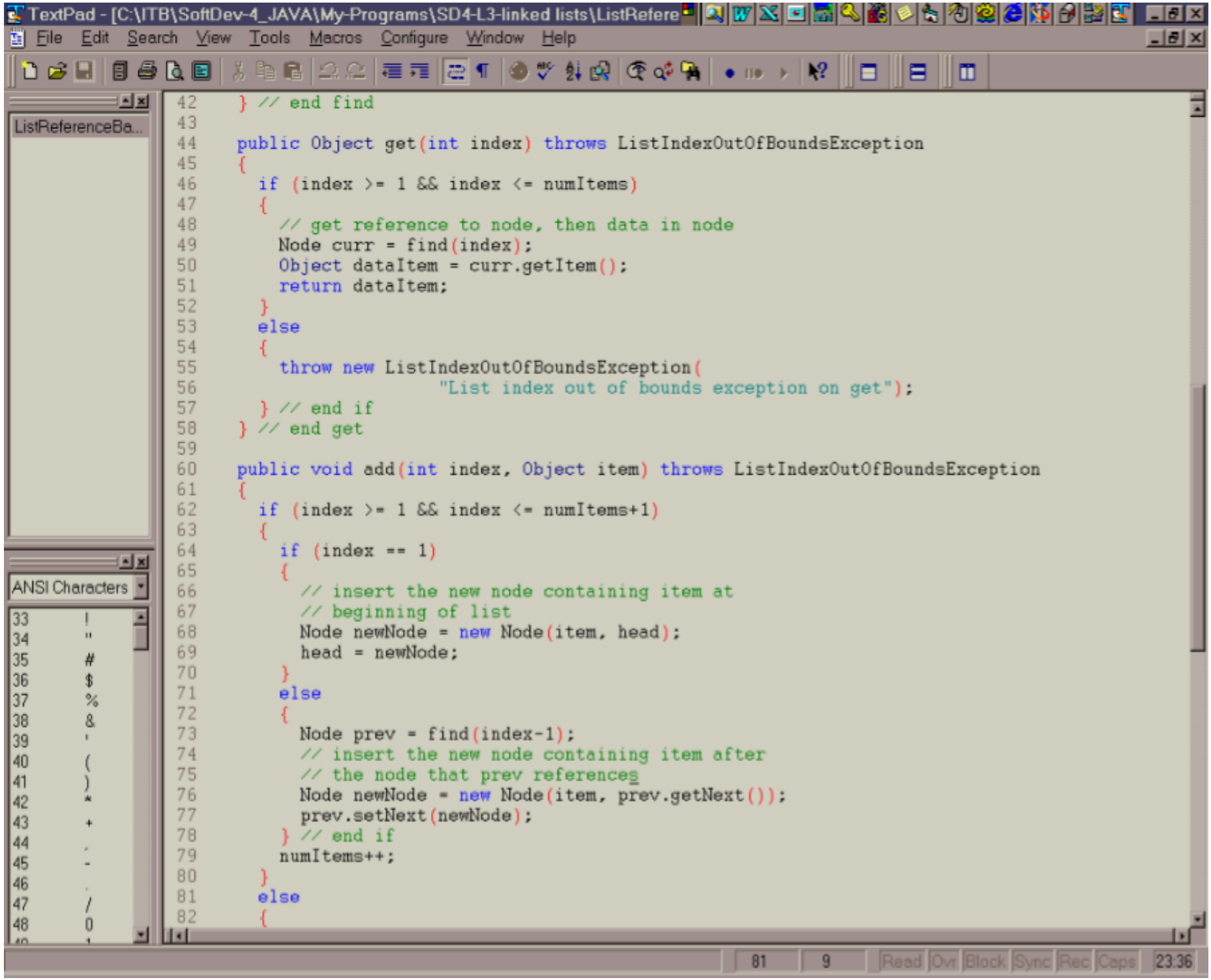
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0

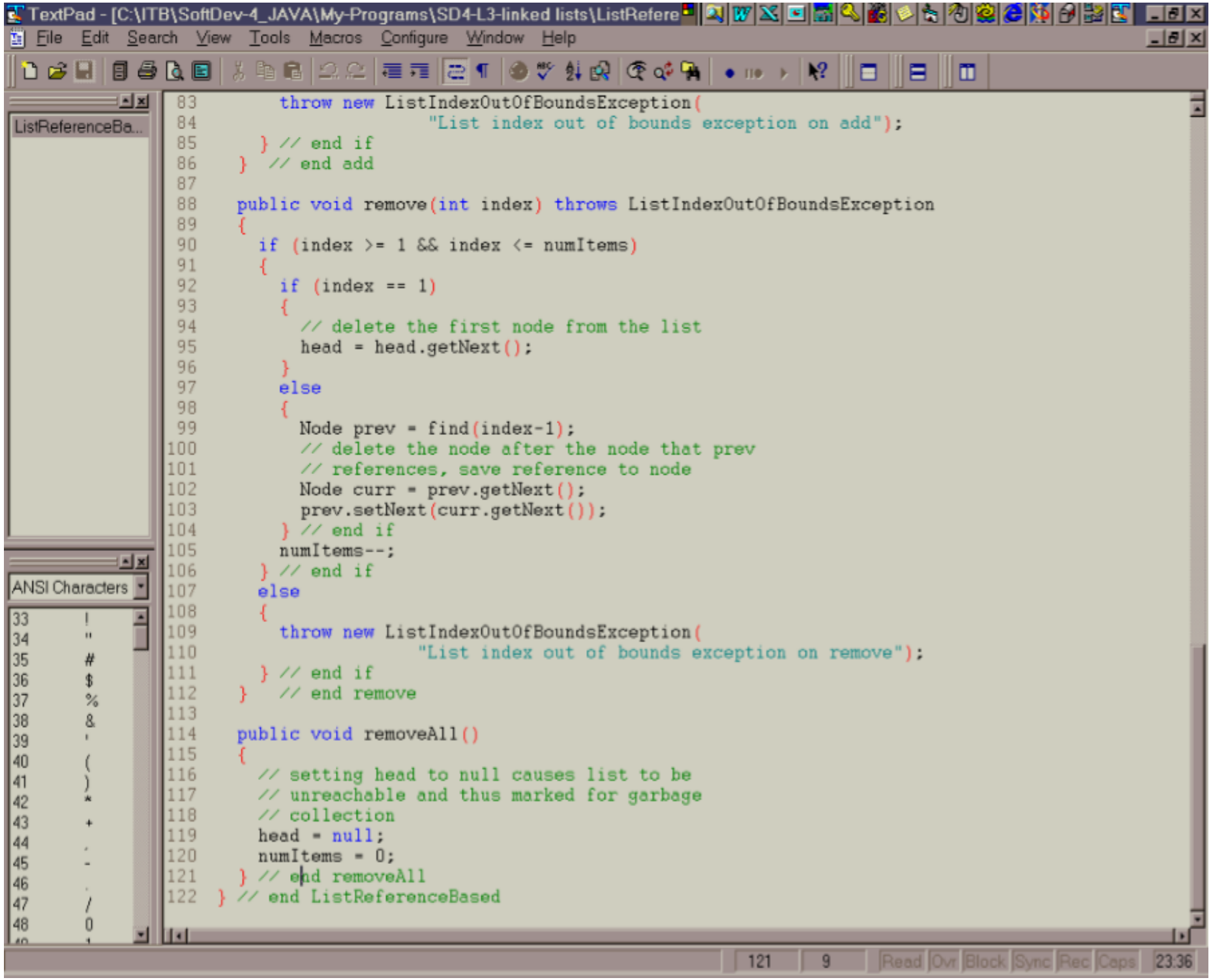


A reference-based implementation of the ADT list

20 23 Read Ovr Block Sync Rec Caps 23:34


TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L3-linked lists\ListRefere
File Edit Search View Tools Macros Configure Window Help
ListReferenceBa...
ANSI Characters
1 // **
2 // Reference-based implementation of ADT list.
3 // **
4
5 public class ListReferenceBased implements ListInterface
6 {
7 // reference to linked list of items
8 private Node head;
9 private int numItems; // number of items in list
10
11 public ListReferenceBased()
12 {
13 numItems = 0;
14 head = null;
15 } // end default constructor
16
17 public boolean isEmpty()
18 {
19 return numItems == 0;
20 } // end isEmpty
21
22 public int size()
23 {
24 return numItems;
25 } // end size
26
27 private Node find(int index)
28 {
29 // --
30 // Locates a specified node in a linked list.
31 // Precondition: index is the number of the desired
32 // node. Assumes that 1 <= index <= numItems+1
33 // Postcondition: Returns a reference to the desired
34 // node.
35 // --
36 Node curr = head;
37 for (int skip = 1; skip < index; skip++)
38 {
39 curr = curr.getNext();
40 } // end for
41 return curr;
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
41
17
Read Ovr Block Sync Rec Caps
23:35


TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L3-linked lists\ListRefere
File Edit Search View Tools Macros Configure Window Help
ListReferenceBa...
ANSI Characters
42 } // end find
43
44 public Object get(int index) throws ListIndexOutOfBoundsException
45 {
46 if (index >= 1 && index <= numItems)
47 {
48 // get reference to node, then data in node
49 Node curr = find(index);
50 Object dataItem = curr.getItem();
51 return dataItem;
52 }
53 else
54 {
55 throw new ListIndexOutOfBoundsException(
56 "List index out of bounds exception on get");
57 } // end if
58 } // end get
59
60 public void add(int index, Object item) throws ListIndexOutOfBoundsException
61 {
62 if (index >= 1 && index <= numItems+1)
63 {
64 if (index == 1)
65 {
66 // insert the new node containing item at
67 // beginning of list
68 Node newNode = new Node(item, head);
69 head = newNode;
70 }
71 else
72 {
73 Node prev = find(index-1);
74 // insert the new node containing item after
75 // the node that prev references
76 Node newNode = new Node(item, prev.getNext());
77 prev.setNext(newNode);
78 } // end if
79 numItems++;
80 }
81 else
82 {
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
81 9
Read Ovr Block Sync Rec Caps
23:36


TextPad - [C:\ITB\SoftDev-4_JAVA\My-Programs\SD4-L3-linked lists\ListRefere
File Edit Search View Tools Macros Configure Window Help
ListReferenceBa...
ANSI Characters
throw new ListIndexOutOfBoundsException(
"List index out of bounds exception on add");
} // end if
} // end add
public void remove(int index) throws ListIndexOutOfBoundsException
{
if (index >= 1 && index <= numItems)
{
if (index == 1)
{
// delete the first node from the list
head = head.getNext();
}
else
{
Node prev = find(index-1);
// delete the node after the node that prev
// references, save reference to node
Node curr = prev.getNext();
prev.setNext(curr.getNext());
} // end if
numItems--;
} // end if
else
{
throw new ListIndexOutOfBoundsException(
"List index out of bounds exception on remove");
} // end if
} // end remove
public void removeAll()
{
// setting head to null causes list to be
// unreachable and thus marked for garbage
// collection
head = null;
numItems = 0;
} // end removeAll
} // end ListReferenceBased
121 9 Read Ovr Block Sync Rec Caps 23:36

Comparing Array-based and Reference based Implementations

When you select an implementation for an ADT, determine the advantages, disadvantages and implications.

We have seen two implementations of an ADT List, one array based, the other a reference-based linked list

Issues:

Arrays are easy to use, but they have a fixed size.

- The maximum number of items in an ADT can be difficult to predict.
- An array may waste memory and storage.
- An array based implementation may be a good choice for a small list.

Linked lists do not have a fixed size.

- In a linked list, an item/node explicitly references the next item/node
- You must traverse a linked list to access its $\mathbf{k}^{th}$ node
- The time to access the $\mathbf{k}^{th}$ node in a link list depends on **k.**
- Insertion and deletion into a linked list require list traversal
- Insertion and deletion into a linked list does not require shifting data.

Passing a Linked List to a Method

How can a method access a linked list?

Easy! **The method simply needs to have access to the linked list's head reference**

With this, the method can access the entire link list.

In the reference-based implementation of the ADT list, the reference variable **head** is a **private** data field of the class.

Variations of the Linked List

There are several variations to the linear reference -based linked list

These are:

- Tail references
- Circular Linked Lists
- Dummy Head Nodes
- Doubly Linked Lists

Tail References

In many situations, we simply want to add an item to the end of a list.

i.e., maintaining a list of requests for a book in a library would require that the new request go at the end of the waiting list.

You could use an ADT list called **waitingList** as follows:

```
waitingList.add(request, waitingList.size()+1) );
```

This **adds request** to the end of **waitingList**.

Remember that in implementing **add** to insert an item at the indicated position, we used the **method find**() to **traverse the list to that position**

The above statement performs the following four steps:

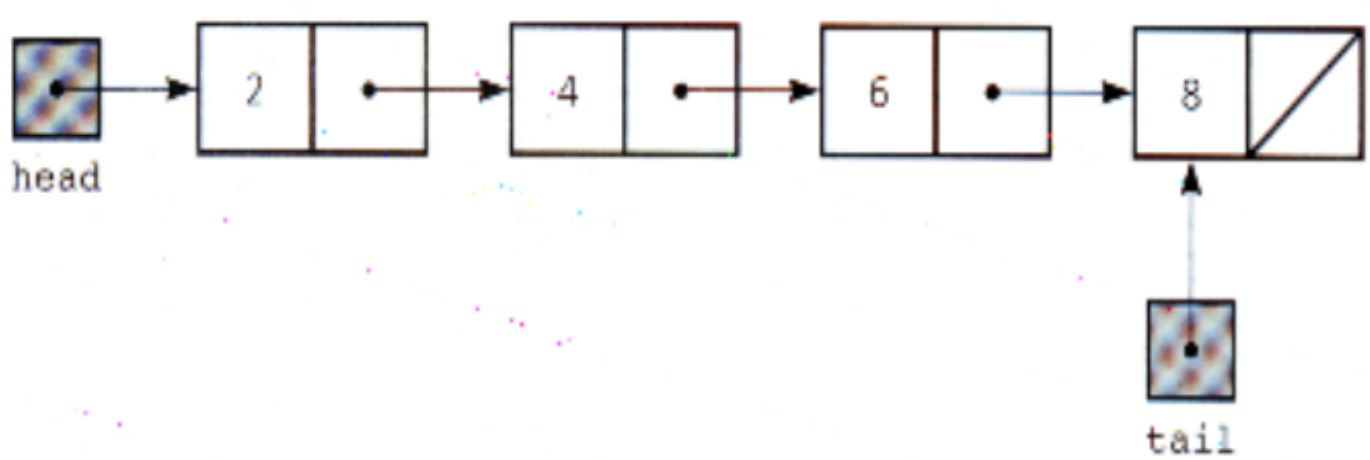
1. **Allocate** a new node for the linked list.
2. **Set** the reference in the last node in the list to reference the new node
3. **Put** the new request in the new node.
4. **Set** the reference in the new node to null.

Each time we add a new request, we must get to the**last** node in the linked list.

- One way to accomplish this is to traverse the list each time you add a new request.

- A better way is to use a **tail** reference to remember where the end of the linked list is – just like head and the start of the linked list.

Like **head**, **tail** is external to the list.



A linked list with ***head*** and ***tail*** references

We can then perform steps 1 through 4 using the single statement:

```
tail.setNext( new Node(request,null));
```

This statement sets the next reference in the last node in the list to point to a newly allocated node.

Update tail so that it references the new last node by writing:

```
tail = tail.getNext();
```

Initially, when you insert the first item into an empty linked list, tail, like head, is null.

Circular Linked Lists.

When you use a computer that is part of a network, you share the services of a server with other users.

The system must organise the users so that each can be given access to the resources in turn.

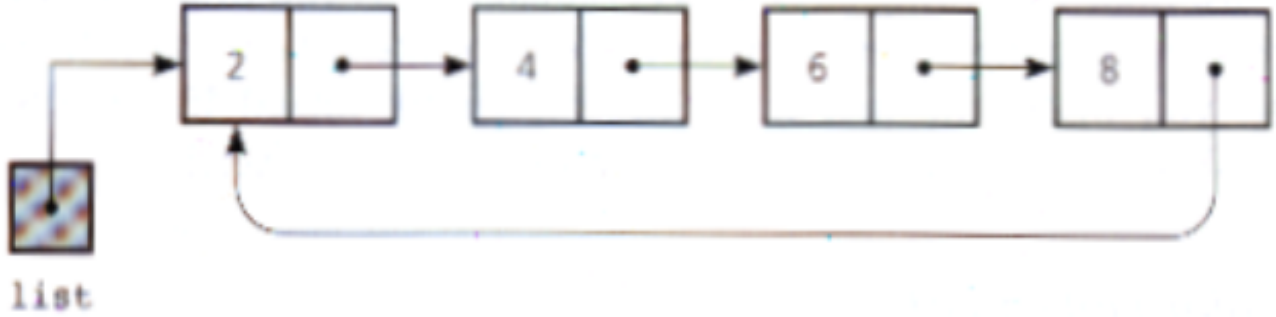
Users frequently log on and log off the system.

A linked list of user names allows the system to maintain order without shifting names when it makes insertions and deletions from the list.

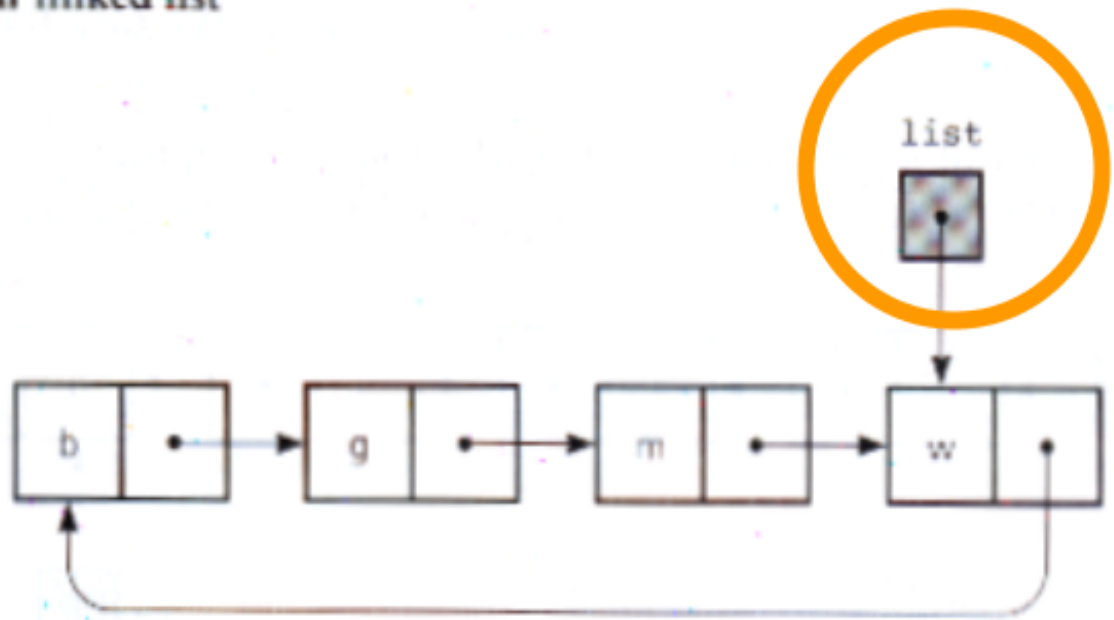
What must the system do when it reached the end of the list?
It must go to the beginning of the list and continue!

The fact that the last node of a link list does not reference any node is therefore inconvenient.

- Suppose we were to change the next portion of the list's last node to reference the first node, instead of containing null.
- The result is a circular linked list.



A circular linked list



A circular linked list with an external reference to the last node

How do you know if you have traversed the complete list?

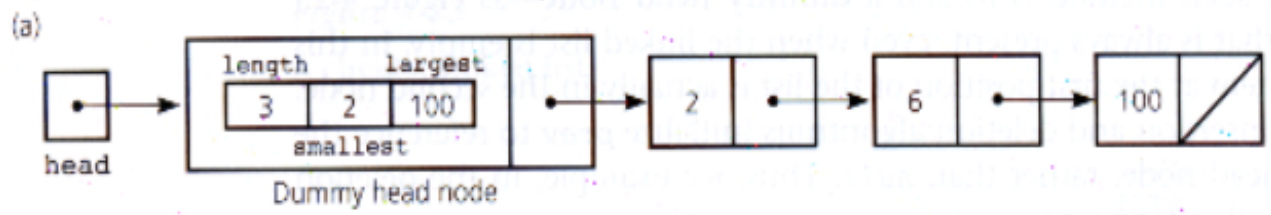
By simply comparing the current reference **curr** to the external reference **list**, you can determine when you have traversed the complete list.

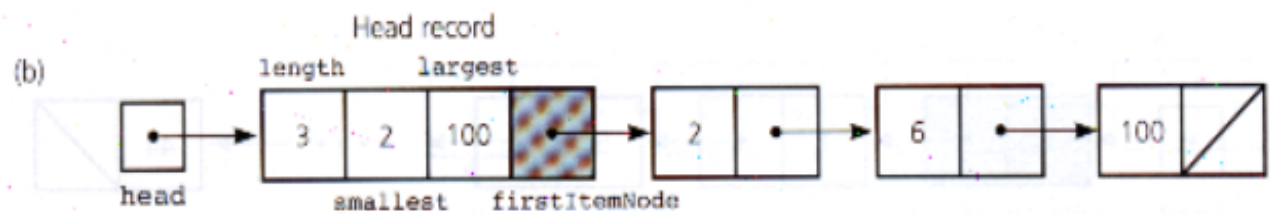
```
// display the data in a circular linked list
// list references its last node
if (list != null)
{
      //list is not empty
      Node first =  list.getNext ();       //reference first node

      Node curr = first;                   // start at first node
      // Loop invariant:  curr references next node to display
      do {
            //write data portion
            System.out.println (curr.getItem());
            Curr = curr.getNext();         //references next node
      } while (curr !=first);              //list traversed????
} //end if
```

Dummy Head Node

```
prev.setNext(curr.getNext());
```

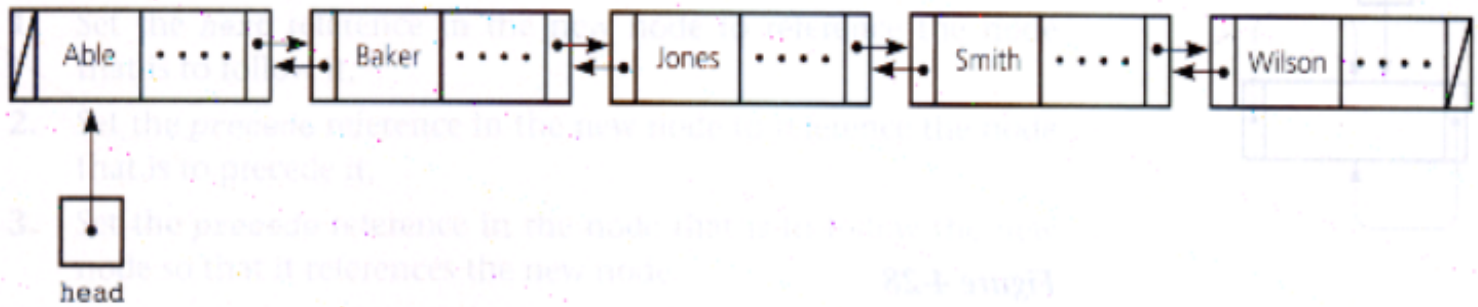
A dummy head node



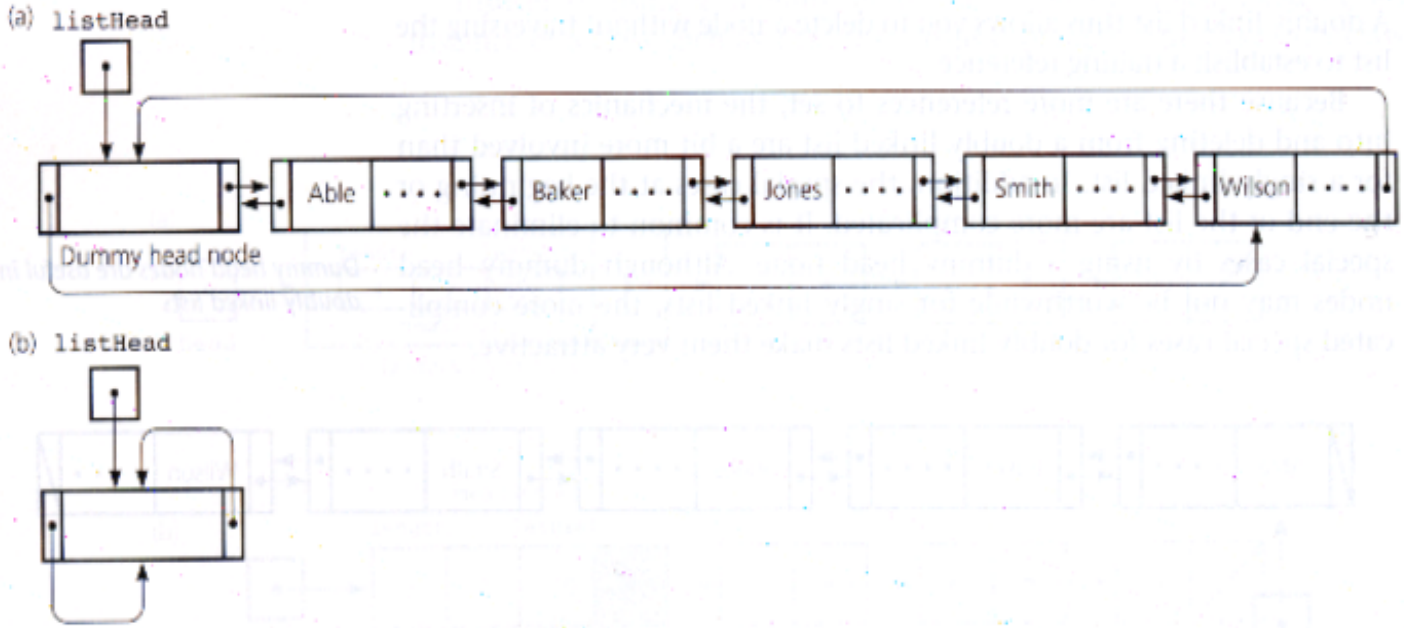


(a) A dummy head node with global information; (b) a head record with global information

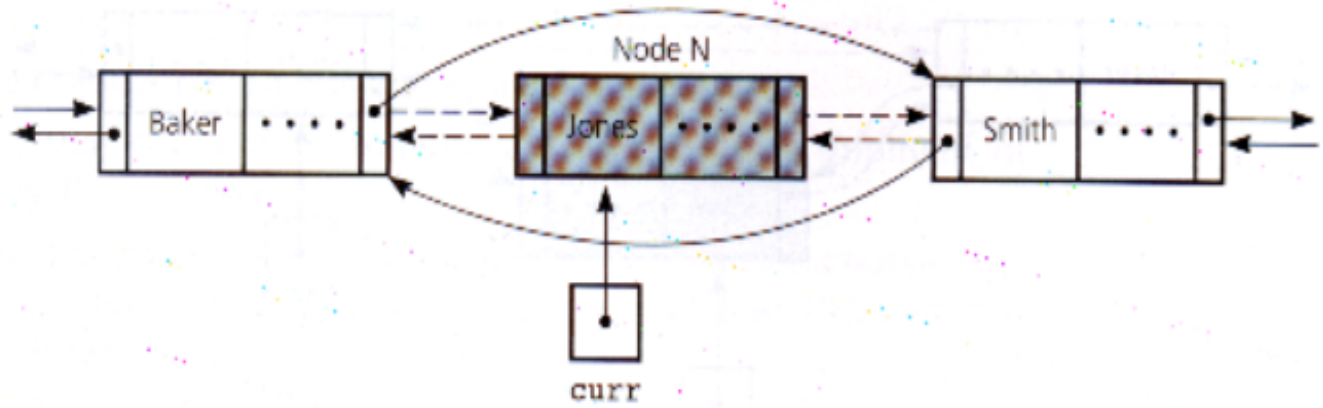
Doubly Linked Lists



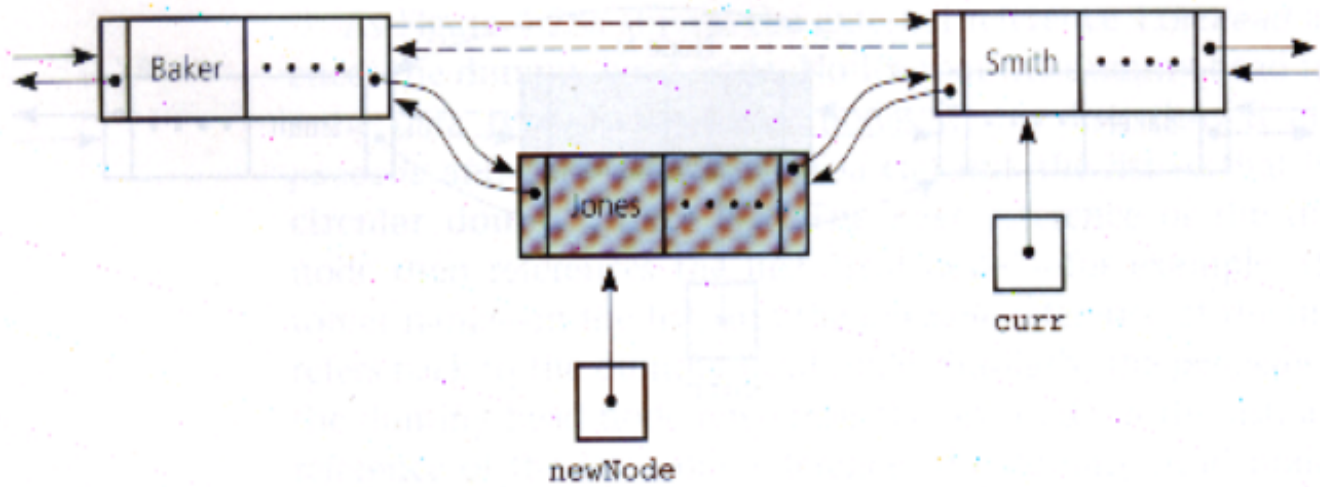
A doubly linked list



(a) A circular doubly linked list with a dummy head node; (b) an empty list with a dummy head node



Reference changes for deletion

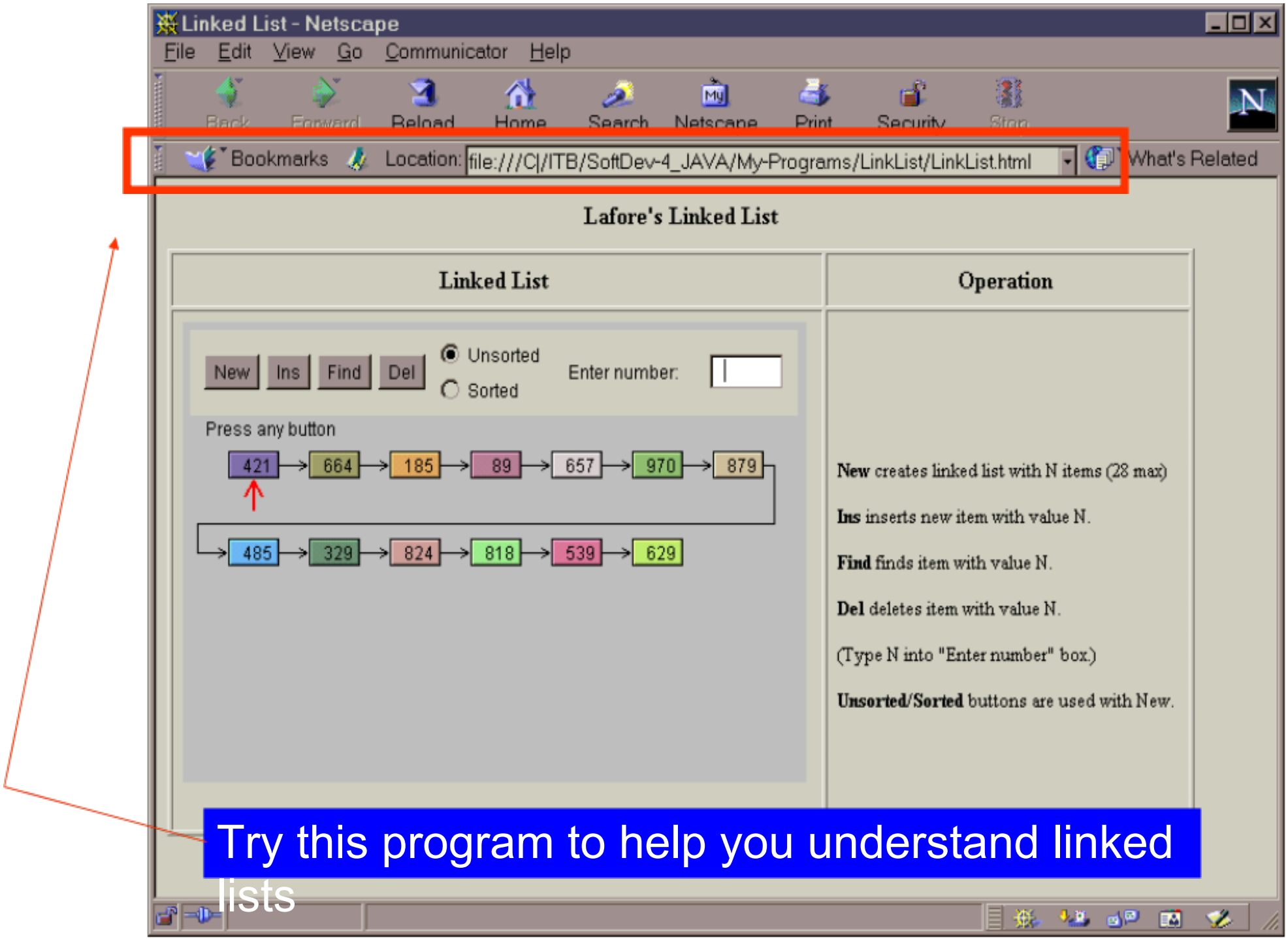


Reference changes for insertion

Programs to do this week:

Implement the ADT List using a reference-based linear linked list

- Write out the specifications for the ADT List
- Implement the ADT List as a Java class over a reference-based linear linked list such that the "walls" are reinforced (done for you, on share)
- This class must implement the ADT operations as public methods of the class
- Add a displayList method to the linked list class. It should display the list in some reasonable manner.
- Add a method called listLargest that find the largest data item in the linked list.
- Write a small driver program that demonstrates the use of the ADT class and therefore the ADT list.
- This program should print results to the screen in some simple manner


Linked List - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: file:///C|/ITB/SoftDev-4_JAVA/My-Programs/LinkList/LinkList.html What's Related
Lafore's Linked List
Linked List
Operation
New Ins Find Del
Unsorted
Sorted
Enter number:
Press any button
421 664 185 89 657 970 879
485 329 824 818 539 629
New creates linked list with N items (28 max)
Ins inserts new item with value N.
Find finds item with value N.
Del deletes item with value N.
(Type N into "Enter number" box.)
Unsorted/Sorted buttons are used with New.
Try this program to help you understand linked lists