CS 124 W05 Tutorial 3: References and Linked Lists
Solutions

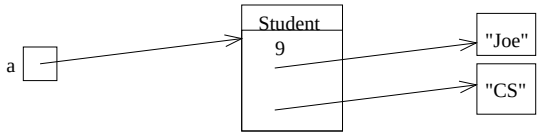
January 20, 2005

1 References

For the following problems, draw what is located in memory after the code is run.

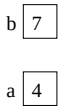
Problem 1

```
public void foo(Student b)
{ b = new Student(23, "Bob", "AM");
}

Student a = new Student(9, "Joe", "CS");
foo(a);
```



Problem 2

```
int b = 4;
int a = b;
b = 7;
```



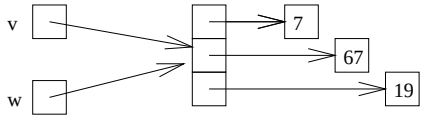
Problem 3

```
public void bar(int d)
{ d++;
}
```

```
int c = 42;
bar(c);
```

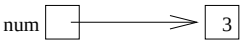
Problem 4

```
Vector v = new Vector();
v.addElement(new Integer(7));
v.addElement(new Integer(67));
Vector w;
w = v;
w.addElement(new Integer(19));
```



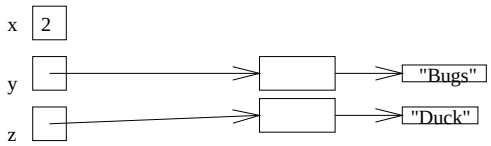
Problem 5

```
public void foo(Integer num)
{ num = new Integer(15);
}

Integer num = new Integer(3);
foo(num);

**NOTE** - My apologies for not explaining this answer properly.
The answer IS in fact 3. The confusion was around the actual
variable name 'num'. When the method foo is called, it creates
a new stack, and in it a variable called num.

THIS IS NOT THE SAME VARIABLE AS THE OTHER 'num'.

If you like, you can think of it as num_foo. When you run foo(),
num_foo is created and a COPY of the reference to Integer(3)
is assigned to num_foo. When you assign a new Integer object to
num_foo, you have done nothing to the original num object.

I hope this clarifies any remaining confusion. If you have any questions,
please let me know and I'll do everything I can to help you out.
```



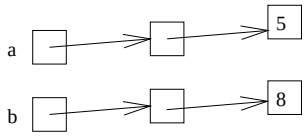
Problem 6

```
public void alter(int a, Person b, Person c)
{ a = 4;
  b = new Person("Bunny");
  c.setName("Duck");
}

int x = 2;
Person y = new Person("Bugs");
Person z = new Person("Daffy");
alter(x,y,z);
```



Problem 7

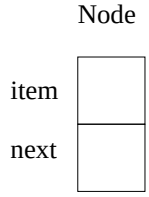
```
public void swap(Node c, Node d)
{ Node temp = c;
  c = d;
  d = temp;
  d.setItem(new Integer(5));
}

Node a = new Node();
Node b = new Node();
a.setItem(new Integer(2));
b.setItem(new Integer(3));
swap(a, b);
b.setItem(new Integer(8));
```



2 Linked Lists

2.1 Node and DLNode

Note that the Node class looks like the following;




```
public class Node
{
  private Object item;
  private Node next;

  public Node(Object item) { ... }
  public Node(Object item, Node next) { ... }

  public void setNext(Node next) { ... }
  public void setItem(Object item) { ... }
  public Node getNext() { ... }
  public Object getItem() { ... }
}
```

The Node class can be used to implement singly-linked and circularly-linked lists. However, to implement doubly-linked lists we will require the DLNode:

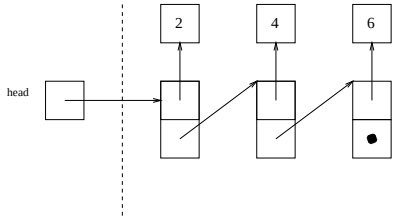
```
public class DLNode
{
  private Object item;
  private DLNode next;
  private DLNode prev;

  public DLNode(Object item) { ... }
  public DLNode(Object item, DLNode next, DLNode prev) { ... }

  public void setNext(Node next) { ... }
  public void setPrev(DLNode prev) { ... }
  public void setItem(Object item) { ... }
  public DLNode getNext() { ... }
  public DLNode getPrev() { ... }
  public Object getItem() { ... }
}
```

2.2 Using the Node class

For the following figures, write the code that takes one figure to the next.

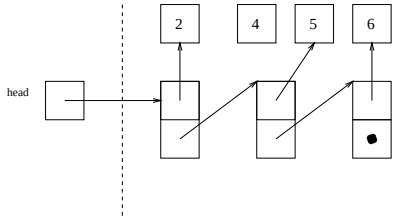
```
Node head = new Node(new Integer(2));
head.setNext(new Node(new Integer(4)));
head.getNext().setNext(new Node(new Integer(6)));
```

1.



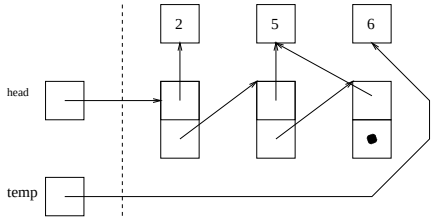

```
head.getNext().setItem(new Integer(5));
```

2.



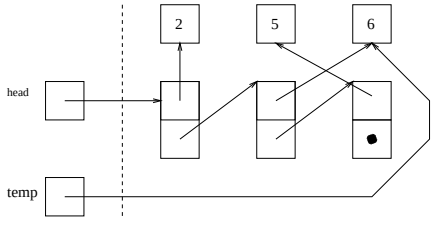

```
Object temp = head.getNext().getNext().getItem();
head.getNext().getNext().setItem(head.getNext().getItem());
```

3.



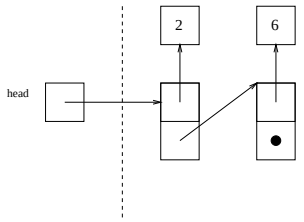

```
head.getNext().setNext(temp);
```

4.



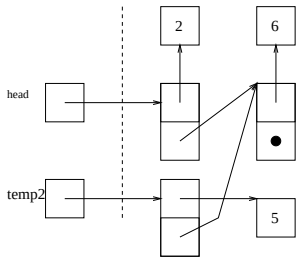
```
head.getNext().setNext(null);
temp = null; (and null temp box omitted)
```

5.



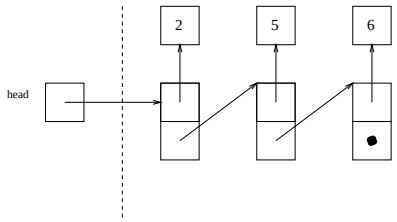

```
Node temp2 = new Node(new Integer(5), head.getNext());
```

6.



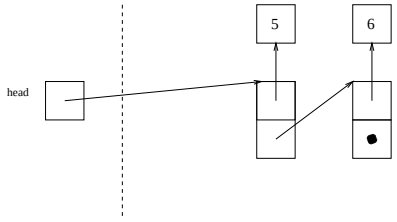

```
head.setNext(temp2);
temp2 = null; (and null temp2 box omitted)
```

7.



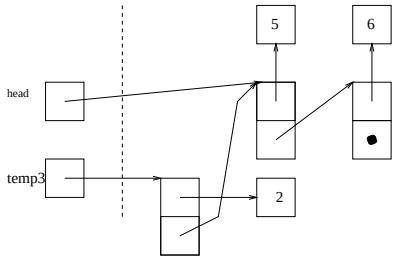

```
head = head.getNext();
```

8.



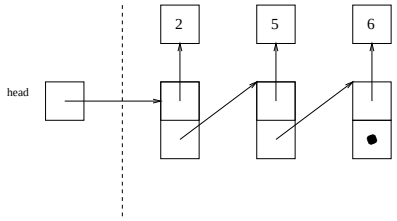
```
Node temp3 = new Node(new Integer(2), head);
```

9.



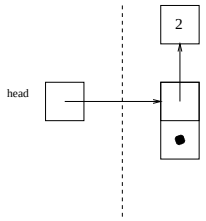

```
head = temp3;
temp3 = null; (and null temp3 omitted)
```

10.



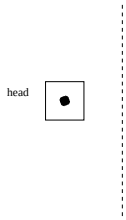

```
head.setNext(null);
```

11.




```
head = null;
```

12.



3 Two implementations of ListInterface

3.1 Circularly Linked List

```
public class CircularlyLinkedList implements ListInterface
{
   private Node tail;
   private int numItems;

   public CircularlyLinkedList()
   {
      numItems = 0;
      tail = null;
   }

   public boolean isEmpty()
   // post: returns true iff this list contains no elements
   {
      return numItems == 0;
   }

   public int size()
   // post: returns the number of elements in this list
   {
      return numItems;
   }

   public void add(int index, Object item) throws
      ListIndexOutOfBoundsException, ListException
   // post: if 1 <= index <= size() + 1 and the list is not full,
   //       then item is inserted at index and other items at
   //       location >index are renumbered accordingly
   //
   //       throws ListIndexOutOfBoundsException if
   //       index < 1 or index > size() + 1
   //       throws ListException if the list is full
   {
      // Check for a valid index
      if(index < 1 || index > size() + 1)
         throw new ListIndexOutOfBoundsException();

      // Note that ListException will never be thrown as this
      // reference-based list cannot be full

      Node curr = new Node(item);
```

```
   // Adding element into an empty list
   if(numItems == 0)
   {
      tail = curr;
      curr.setNext(curr);
   }
   // Adding to front of non-empty list
   else if(index == 1)
   {
      curr.setNext(tail.getNext());
      tail.setNext(curr);
   }
   // Adding to end of non-empty list
   else if(index == numItems + 1)
   {
      curr.setNext(tail.getNext());
      tail.setNext(curr);
      tail = curr;
   }
   // Adding to middle of non-empty list
   else
   {
      Node prev = this.find(index-1);
      curr.setNext(prev.getNext());
      prev.setNext(curr);
   }
   numItems++;
}

public void remove(int index)
   throws ListIndexOutOfBoundsException
// post: if 1 <= index <= size() then the item at
//       position index is removed and all others
//       are renumbered accordingly
//       throws ListIndexOutOfBoundsException
//       if index < 1 or index > size()
{
   // Check for a valid index
   if(index < 1 || index > size())
      throw new ListIndexOutOfBoundsException();

   // Removing the only element in the list
   if(numItems == 0)
   {
      tail = null;
   }
```

```
   // Removing the first element in the list
   else if(index == 1)
   {
      tail.setNext(tail.getNext().getNext());
   }
   // Removing a middle to last element in the list
   else
   {
      Node prev = find(index - 1);
      prev.setNext(prev.getNext().getNext());

      // If removing the last element, update tail
      if(index == numItems - 1)
         tail = prev;
   }
   numItems--;
}

public Object get(int index) throws
   ListIndexOutOfBoundsException
// post: if 1 <= index <= size() then the item at position
//       index is returned
//       throws ListIndexOutOfBoundsException if index < 1
//       or index > size()
{
   // Check for a valid index
   if(index < 1 || index > size())
      throw new ListIndexOutOfBoundsException();

   return find(index).getItem();
}

public void removeAll()
// post: isEmpty()
{
   tail = null;
   numItems = 0;
}

private Node find(int index)
// pre:  1 <= index <= size()
// post: returns the Node at index
{
   // Start with the first node
   Node curr = tail.getNext();
```

```
      // Find the node at index
      for(int i = 1; i < index; i++)
         curr = curr.getNext();

      return curr;
   }
}
```

3.2 Doubly Linked List

```
public class DoublyLinkedList implements ListInterface
{
   private DLNode head;
   private DLNode tail;
   private int numItems;

   public DoublyLinkedList()
   {
      numItems = 0;
      head = null;
      tail = null;
   }

   public boolean isEmpty()
   // post: returns true iff this list contains no elements
   {
      return numItems == 0;
   }

   public int size()
   // post: returns the number of elements in this list
   {
      return numItems;
   }

   public void add(int index, Object item) throws
      ListIndexOutOfBoundsException, ListException
   // post: if 1 <= index <= size() + 1 and the list is not full,
   //       then item is inserted at index and other items at
   //       location >index are renumbered accordingly
   //
   //       throws ListIndexOutOfBoundsException if
   //       index < 1 or index > size() + 1
   //       throws ListException if the list is full
   {
      // Check for a valid index
      if(index < 1 || index > size() + 1)
         throw new ListIndexOutOfBoundsException();

      // Note that ListException will never be thrown as this
      // reference-based list cannot be full

      DLNode curr = new DLNode(item);
```

```
   // Adding an element into an empty list
   if(numItems == 0)
   {
      head = curr;
      tail = curr;
   }
   // Adding to the front of a non-empty list
   else if(index == 1)
   {
      head.setPrev(curr);
      curr.setNext(head);
      head = curr;
   }
   // Adding to the end of a non-empty list
   else if(index == numItems + 1)
   {
      tail.setNext(curr)
      curr.setPrev(tail);
      tail = curr;
   }
   // Adding to the middle of a non-empty list
   else
   {
      Node prev = this.find(index - 1);
      prev.getNext().setPrev(curr);
      curr.setNext(prev.getNext());
      curr.setPrev(prev);
      prev.setNext(curr);
   }
   numItems++;
}

public void remove(int index)
   throws ListIndexOutOfBoundsException
// post: if 1 <= index <= size() then the item at
//       position index is removed and all others
//       are renumbered accordingly
//       throws ListIndexOutOfBoundsException
//       if index < 1 or index > size()
{
   // Check for a valid index
   if(index < 1 || index > size())
      throw new ListIndexOutOfBoundsException();

   // Removing the only element in the list
   if(numItems == 0)
```

```
   {
      head = null;
      tail = null;
   }
   // Removing the first element in the list
   else if(index == 1)
   {
      head = head.getNext();
      head.setPrev(null);
   }
   // Removing the last element in the list
   else if(index == numItems)
   {
      tail = tail.getPrev();
      tail.setNext(null);
   }
   // Removing a middle element in the list
   else
   {
      Node curr = find(index);
      curr.getPrev().setNext(curr.getNext());
      curr.getNext().setPrev(curr.getPrev());
   }
   numItems--;
}

public Object get(int index) throws
   ListIndexOutOfBoundsException
// post: if 1 <= index <= size() then the item at position
//       index is returned
//       throws ListIndexOutOfBoundsException if index < 1
//       or index > size()
{
   // Check for a valid index
   if(index < 1 || index > size())
      throw new ListIndexOutOfBoundsException();

   return find(index).getItem();
}

public void removeAll()
// post: isEmpty()
{
   head = null;
   tail = null;
   numItems = 0;
```

```
   }

   private Node find(int index)
   // pre:  1 <= index <= size()
   // post: returns the Node at index
   {
      // Start with the first node
      Node curr = head;

      // Find the node at index
      for(int i = 1; i < index; i++)
         curr = curr.getNext();

      return curr;
   }
}
```