COSC 2006: Data Structures I

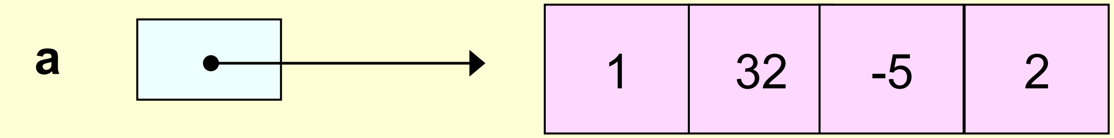
Linked Structures

IntNode class

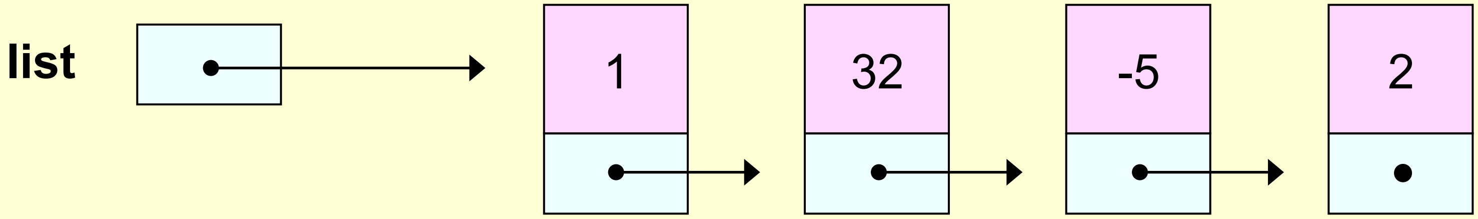
Linked Bag class

Linear Data Structures (1)

An array of integers
Contiguous memory locations

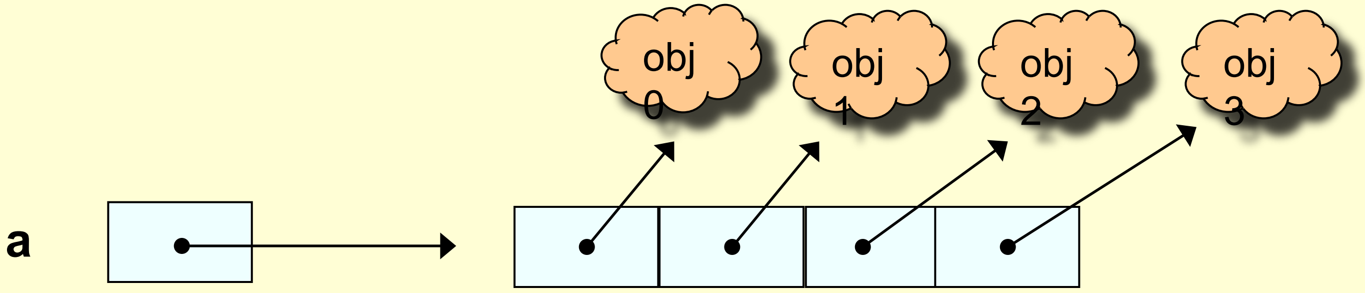


A linked list of integers defined by linked nodes
Nodes can be anywhere in memory

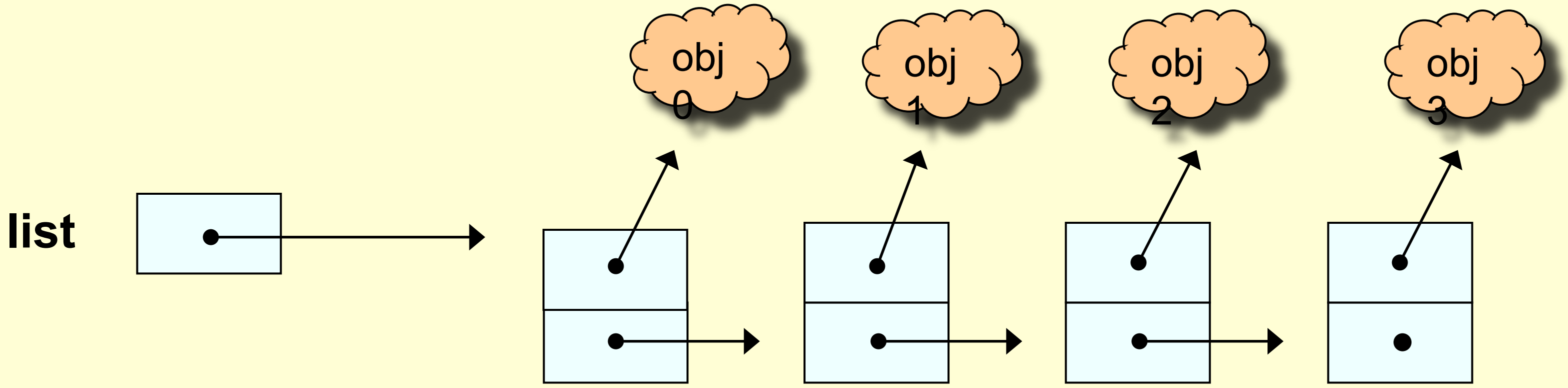


Linear Data Structures (2)

An array of objects

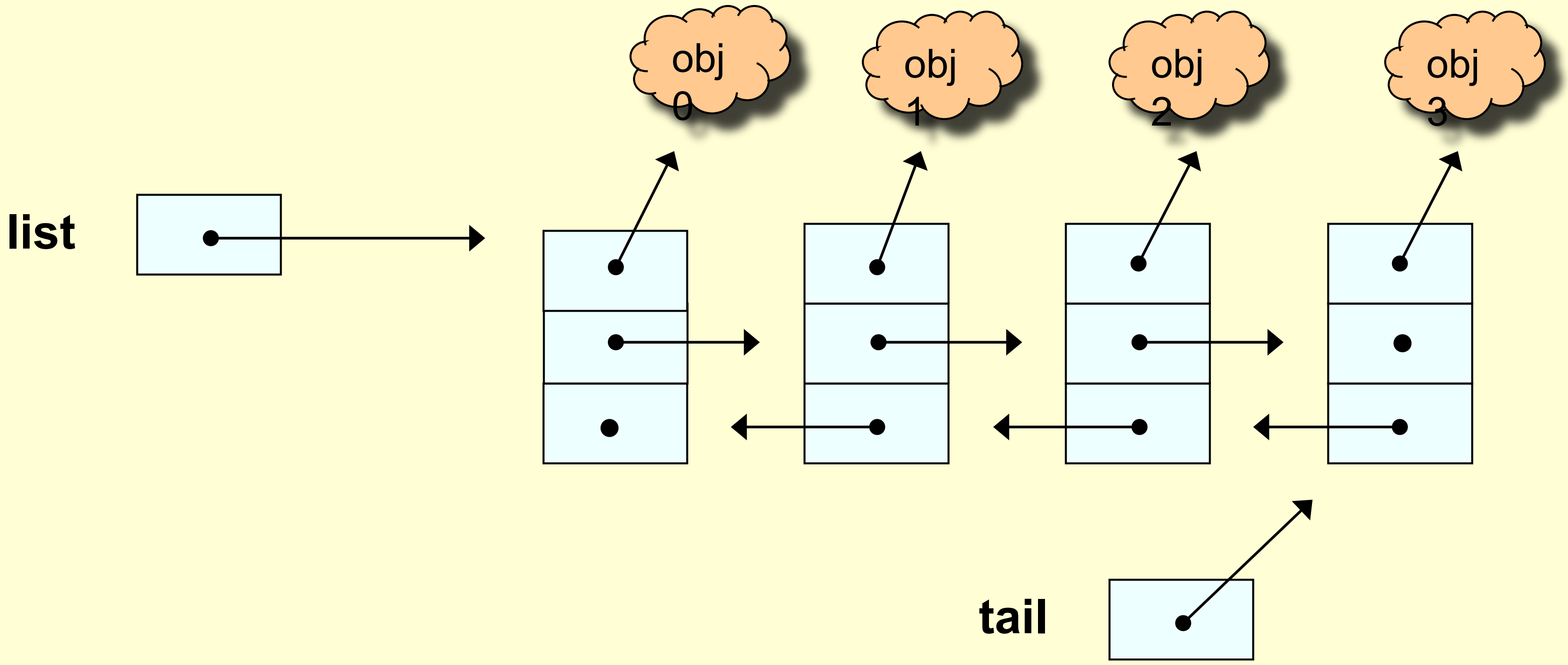


A singly linked list of objects



Linear Data Structures (3)

A doubly linked list of objects



Nodes and links (1)

- A node is an object that contains a data part and a link part
- The link part is a reference to another node which in turn contains a data part and and a link part and so on
- The end of the list is indicated by a null reference in the link part of the node
- We can follow the links to access the data

Nodes and links (2)

- A node is a self-referential structure
- We will first consider lists of integers and other primitive types
- First node of a list is often called the head
- Last node of a list is often called the tail

The IntNode class (1)

Main uses this class to define nodes, instance methods that operate on nodes and static methods that operate on entire lists of nodes.

```
public class IntNode
{
   private int data; // data part of node
   private IntNode link; // link to next node

   // instance methods that operate on nodes

   // static methods that operate on lists
}
```

The IntNode class (2)

Class design for the constructor and instance methods

```
public class IntNode
{  public IntNode(int data, IntNode link) {...}

   public int getData() {...}
   public IntNode getLink() {...}

   public void setData(int data) {...}
   public void setLink(IntNode link) {...}

   public void addNodeAfter(int element) {...}
   public void removeNodeAfter() {...}
   public String toString() {...}

   // static methods on next slide
```

we added this method

The IntNode class (3)

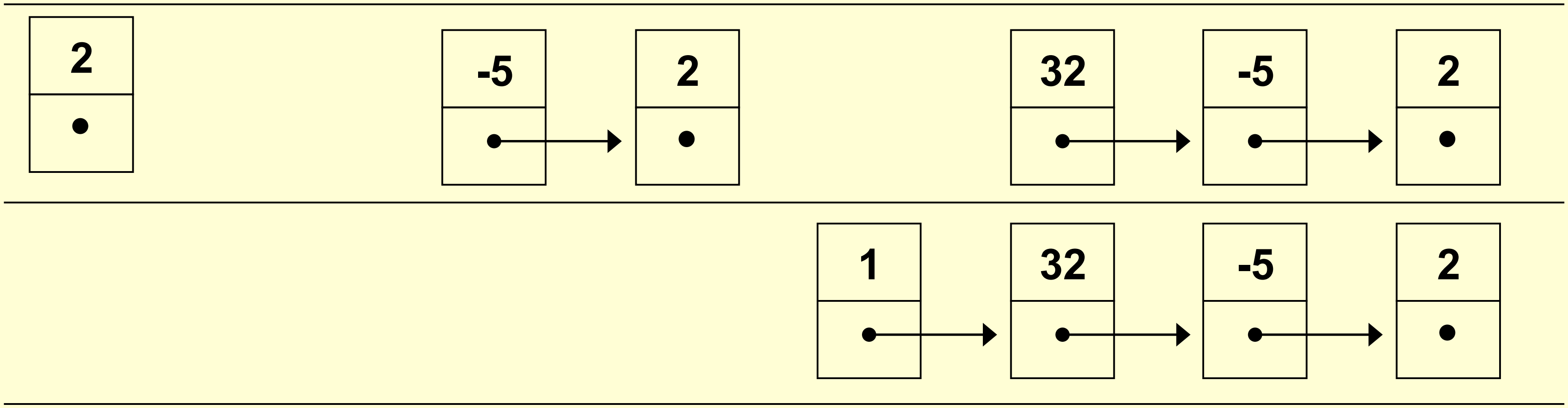
Class design for the static methods operating on lists of nodes

```
   public static IntNode listCopy(
      IntNode source) {...}
   public static IntNode[] listCopyWithTail(
      IntNode source) {...}
   public static int listLength(
      IntNode head) {...}
   public static IntNode[] listPart(
      IntNode start, IntNode end) {...}
   public static IntNode listPosition(
      IntNode head, int position) {...}
   public static IntNode listSearch(
      IntNode head, int target) {...}
}
```

Constructing a specific list (1)

The list <1, 32, -5, 2> can be constructed one node at a time in reverse order using the folllowing statements which always insert at the head of the list

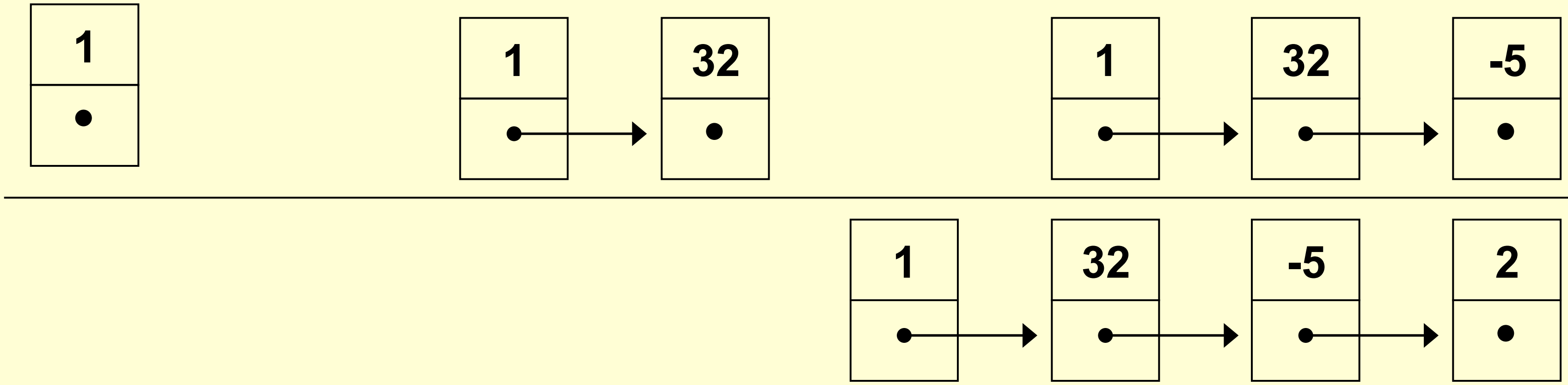
```
IntNode head = new IntNode(2,null);
head = new IntNode(-5,head);
head = new IntNode(32,head);
head = new IntNode(1,head);
```



Constructing a specific list (2)

Here is another way to construct the list <1, 32, -5, 2> in left to right order by inserting at the tail. Here we assume that the private data fields ARE NOT accessible.

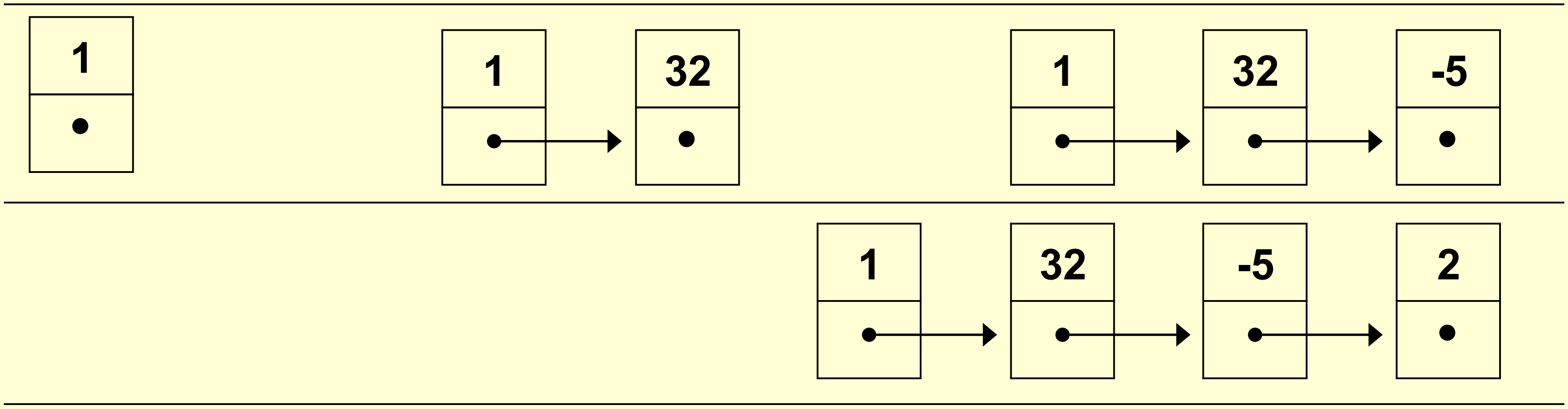
```
IntNode head = new IntNode(1, null);
head.setLink(new IntNode(32, null));
head.getLink().setLink(new IntNode(-5, null));
head.getLink().getLink().setLink(new
                                IntNode(2,null));
```



Constructing a specific list (3)

LIke previous slide but assuming that the private data fields are directly accessible.

```
IntNode head = new IntNode(1, null);
head.link = new IntNode(32, null);
head.link.link = new IntNode(-5, null);
head.link.link.link = new IntNode(2,null);
```



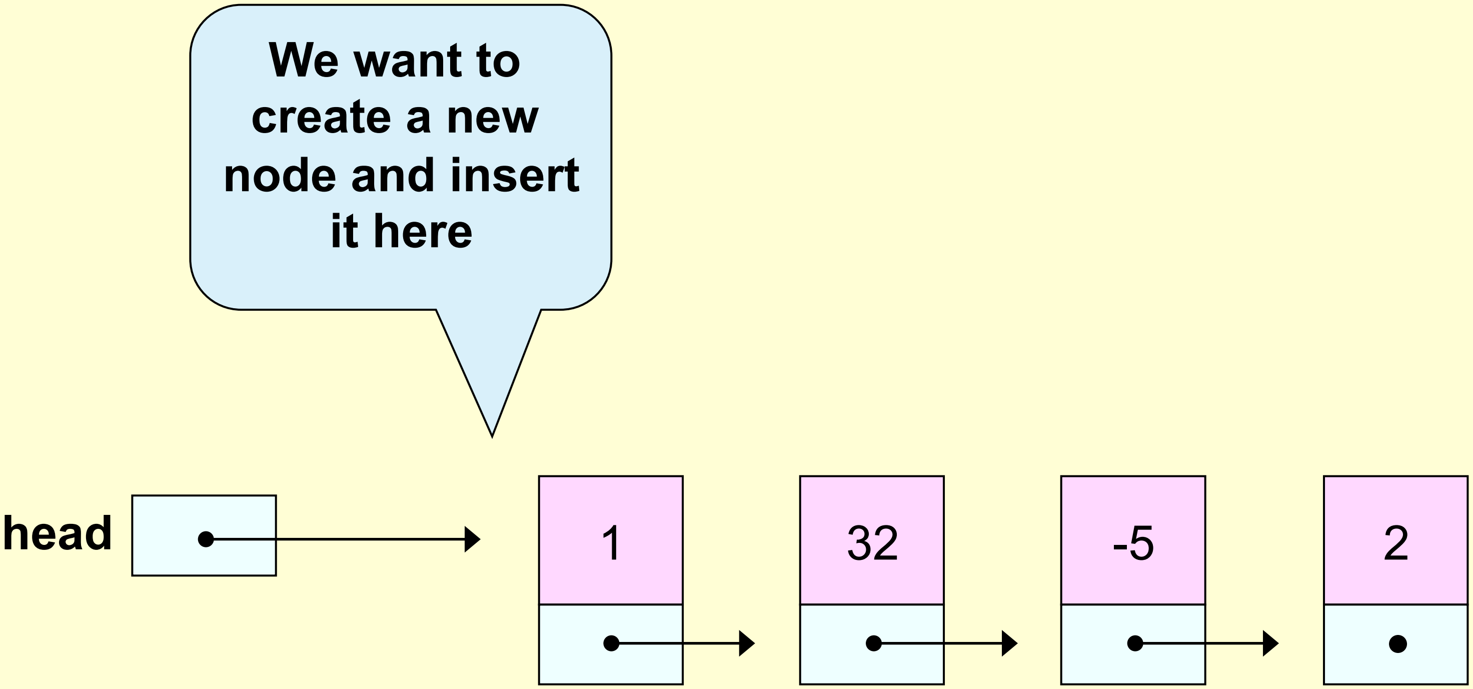
Constructing a specific list (4)

The list <1, 32, -5, 2> can be constructed in order using the single statement.

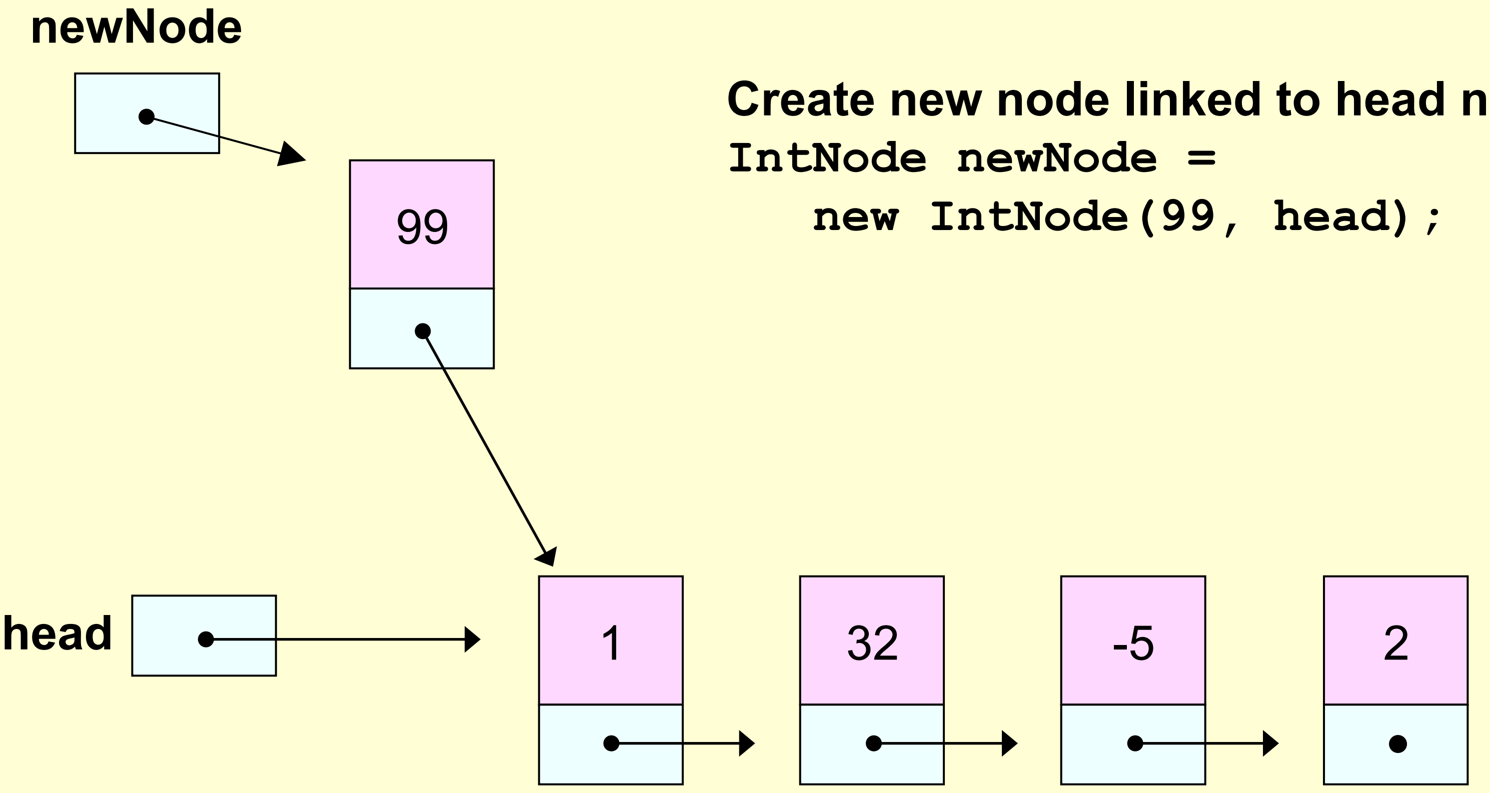
```
IntNode head =
   new IntNode(1,
   new IntNode(32,
   new IntNode(-5,
   new IntNode(2,null))));
```

This technique is useful for creating simple lists to be used in testing the IntNode class.

Adding node at head of list (1-4)



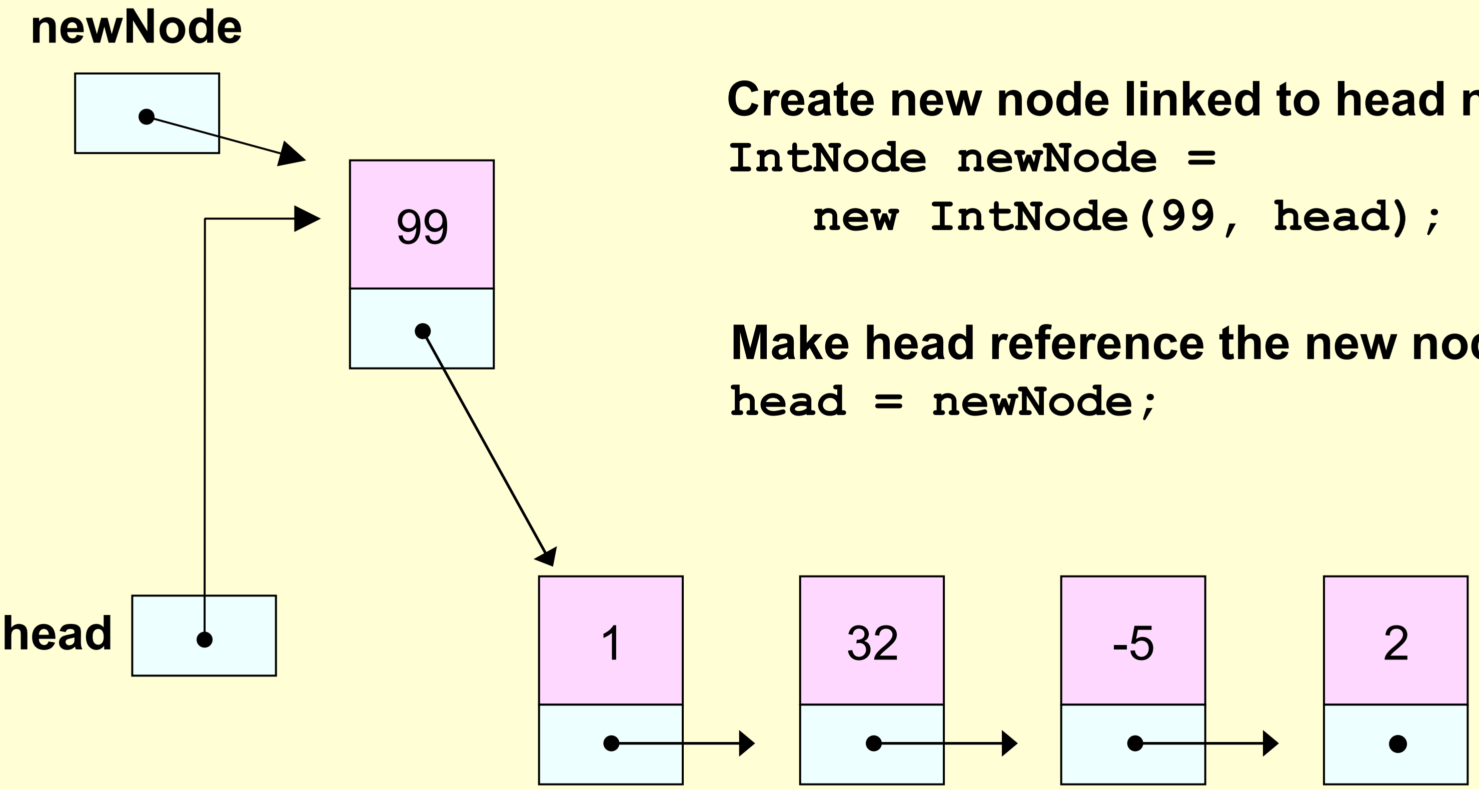
Adding node at head of list (2-4)



Create new node linked to head node

```
IntNode newNode =
   new IntNode(99, head);
```

Adding node at head of list (3-4)



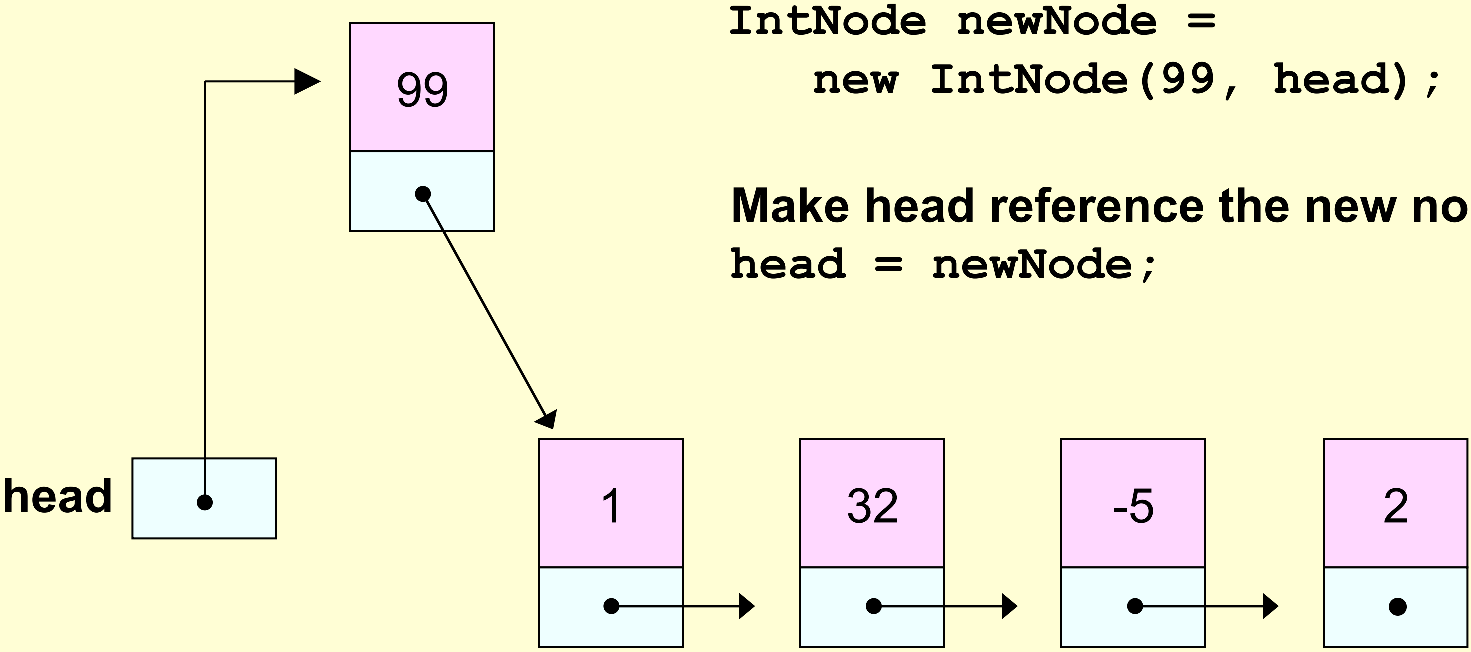
Create new node linked to head node

```
IntNode newNode =
   new IntNode(99, head);
```

Make head reference the new node

```
head = newNode;
```

Adding node at head of list (4-4)



Create new node linked to head node

```
IntNode newNode =
   new IntNode(99, head);
```

Make head reference the new node

```
head = newNode;
```

This can be done in one statement:

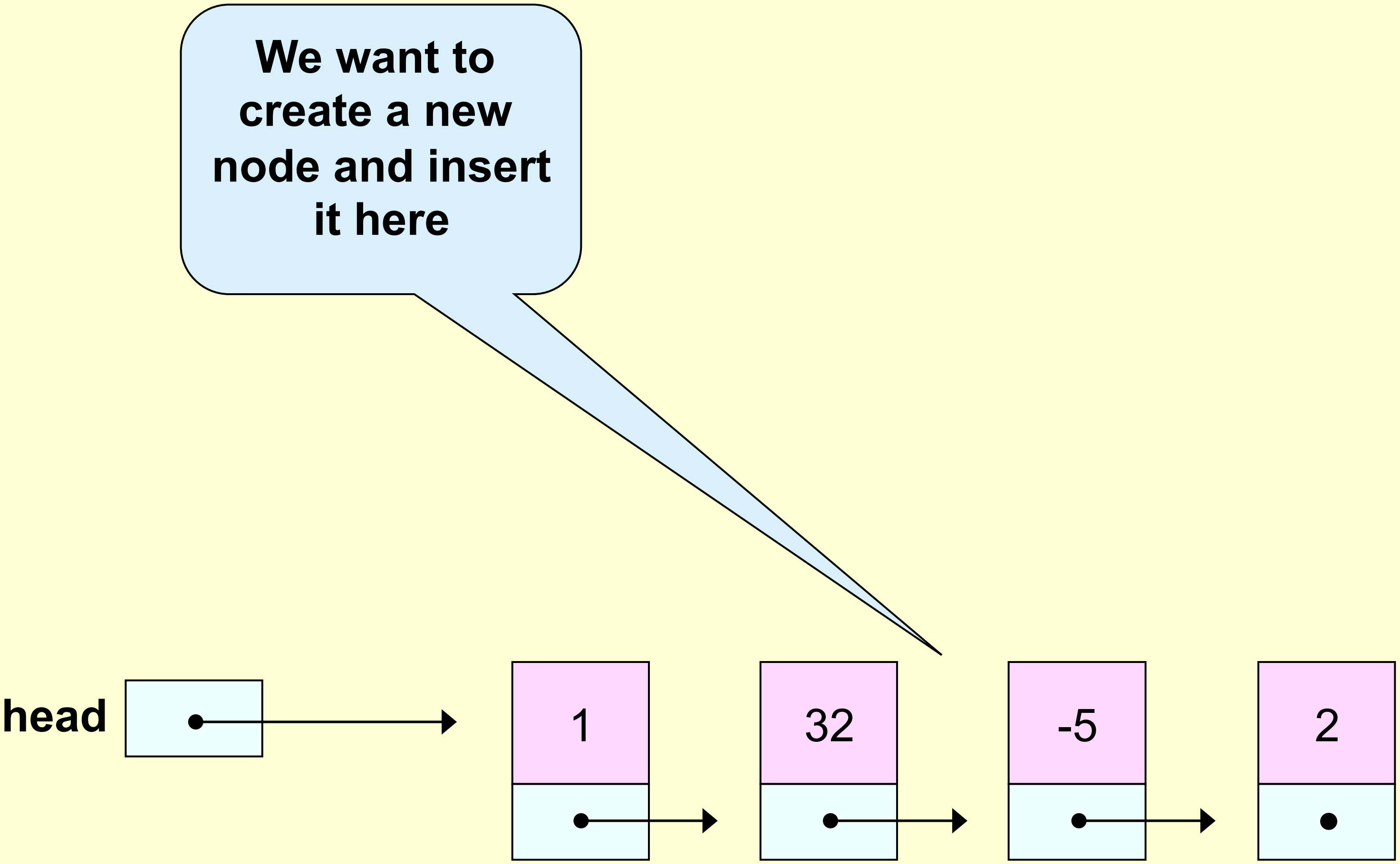
```
head = new IntNode(99,head);
```

Adding node at head of list (2)

- To add a new node at the head of a list use
  - `head = new IntNode(newData, head);`
- If head is null this also works to give a one-element list
  - `head = new IntNode(newData, head);`
- Construct a one-element list using

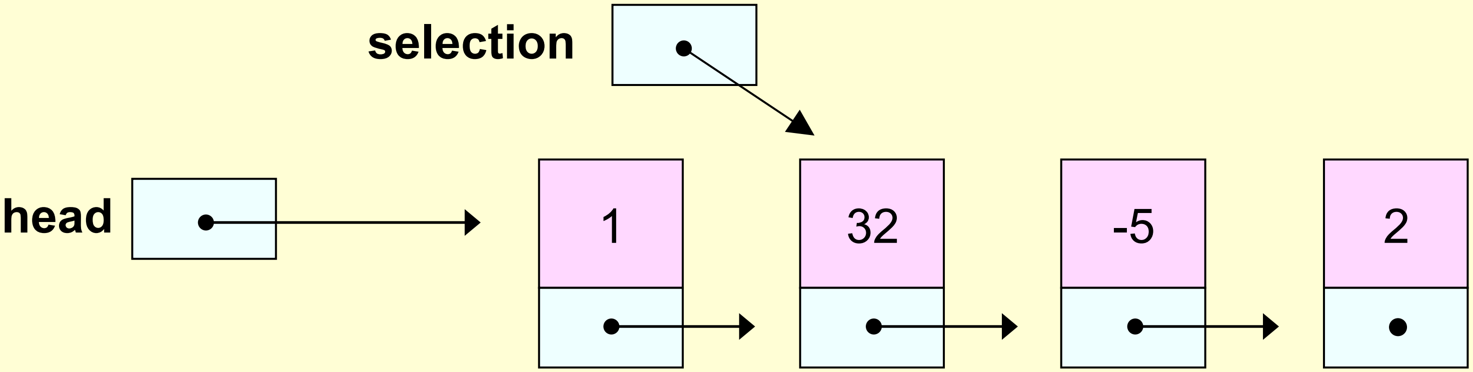
```
IntNode head =
      new IntNode(newData, null);
```

Adding node after a node (1-6)



We want to create a new node and insert it here

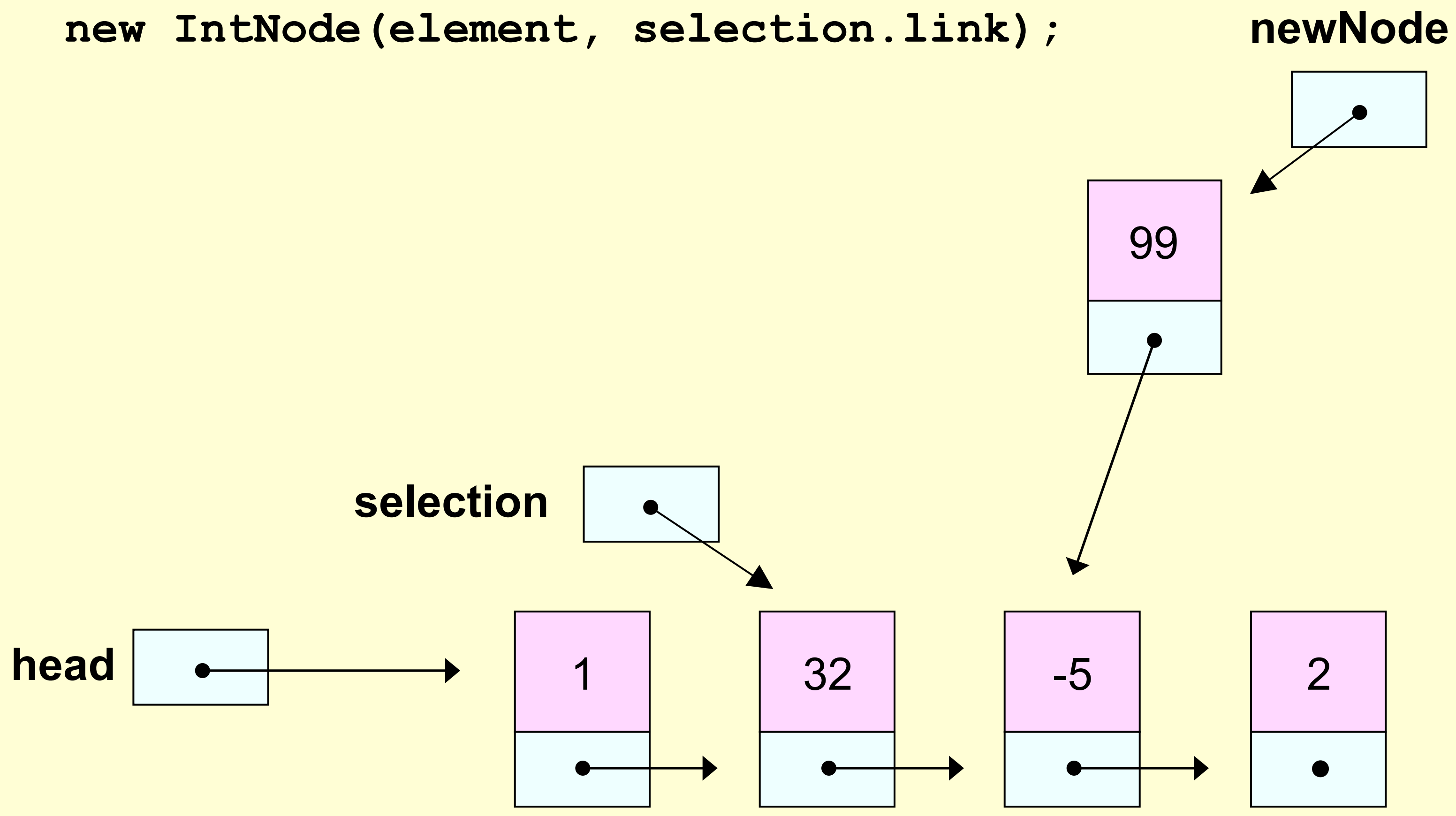
Adding node after a node (2-6)



Adding node after a node (3-6)

Create new node and set its link

```
IntNode newNode =
   new IntNode(element, selection.link);
```



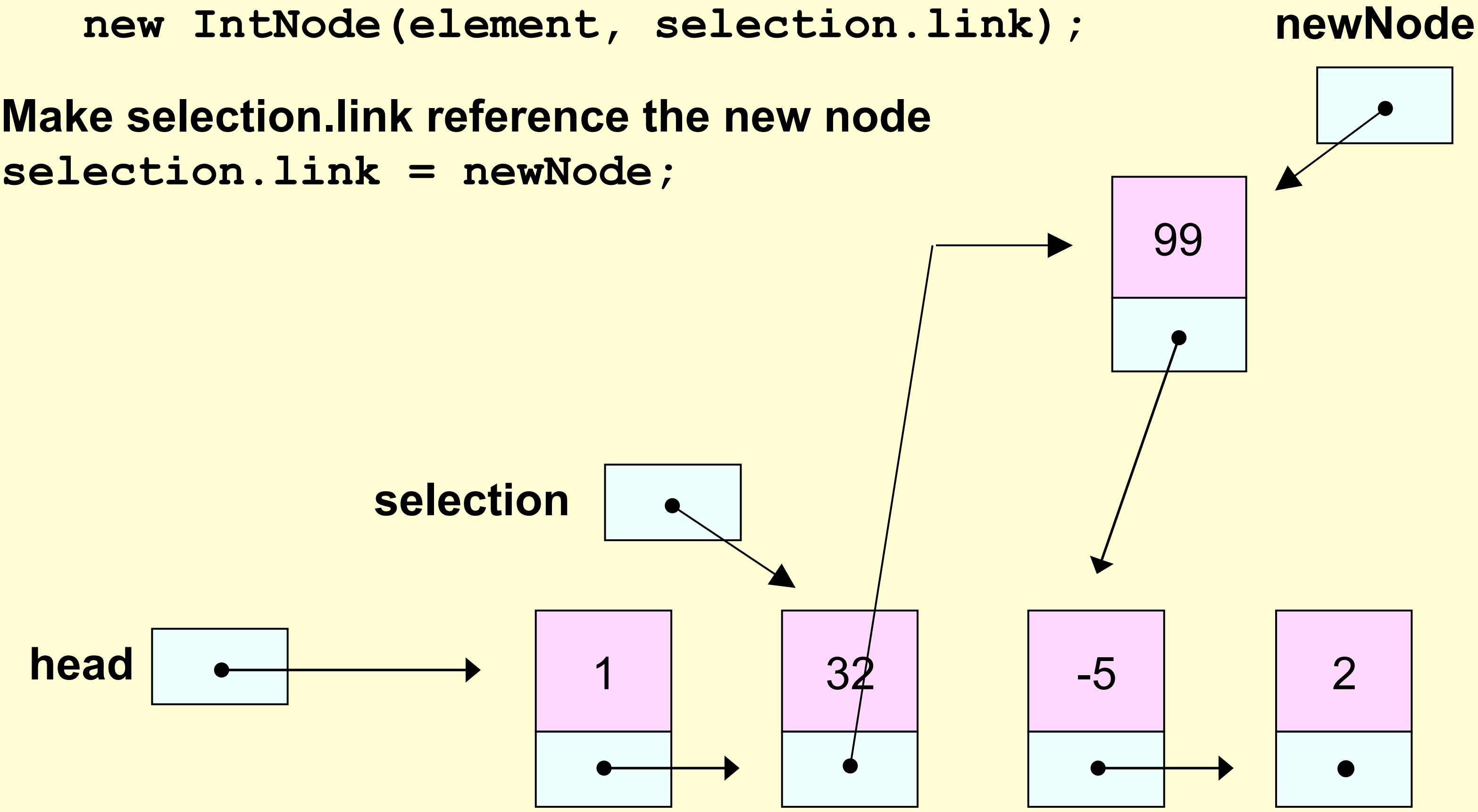
Adding node after a node (4-6)

Create new node and set its link

```
IntNode newNode =
   new IntNode(element, selection.link);
```

Make selection.link reference the new node

```
selection.link = newNode;
```



Adding node after a node (5-6)

Create new node and set its link

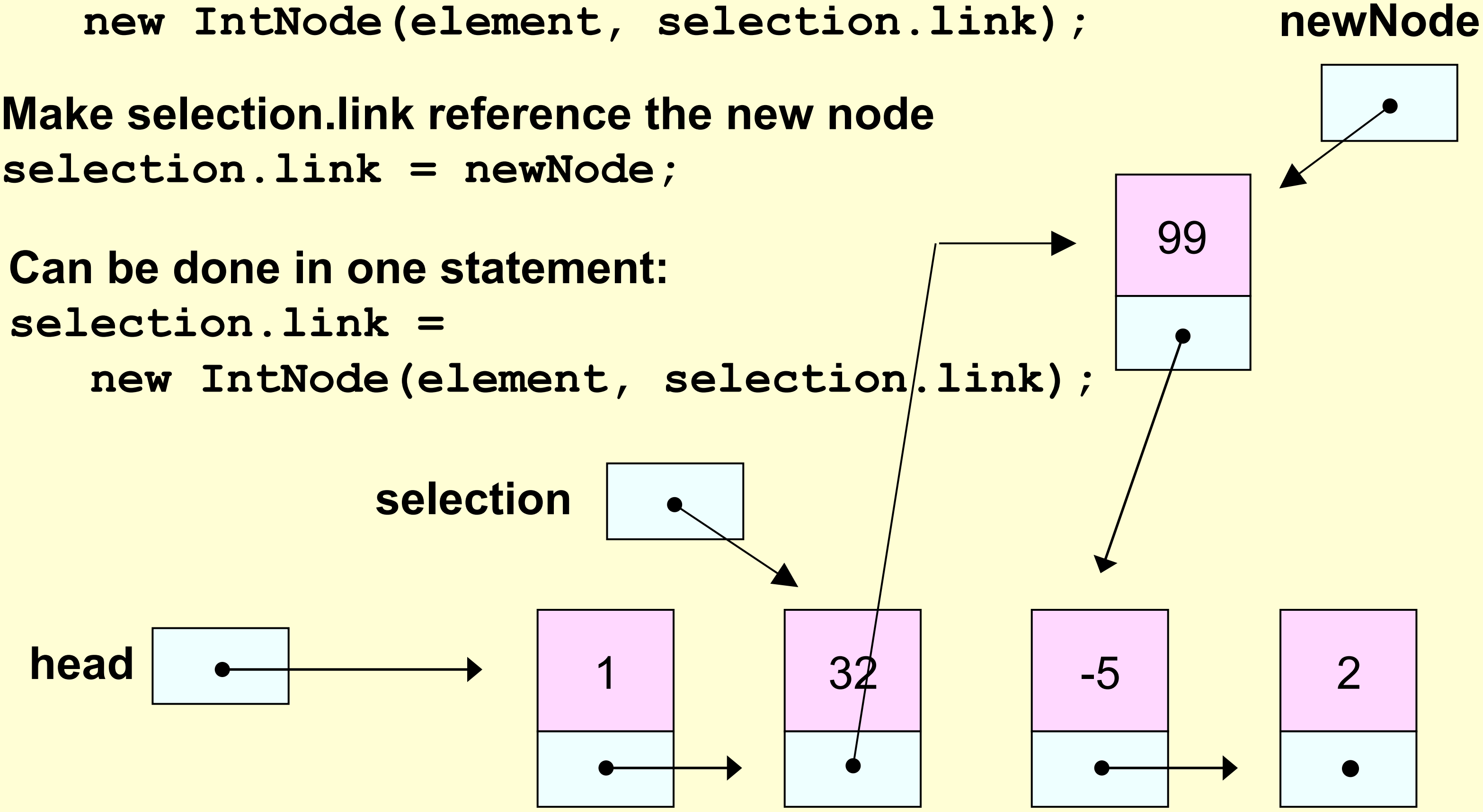
```
IntNode newNode =
   new IntNode(element, selection.link);
```

Make selection.link reference the new node

```
selection.link = newNode;
```

Can be done in one statement:

```
selection.link =
   new IntNode(element, selection.link);
```



Adding node after a node (6-6)

Create new node and set its link

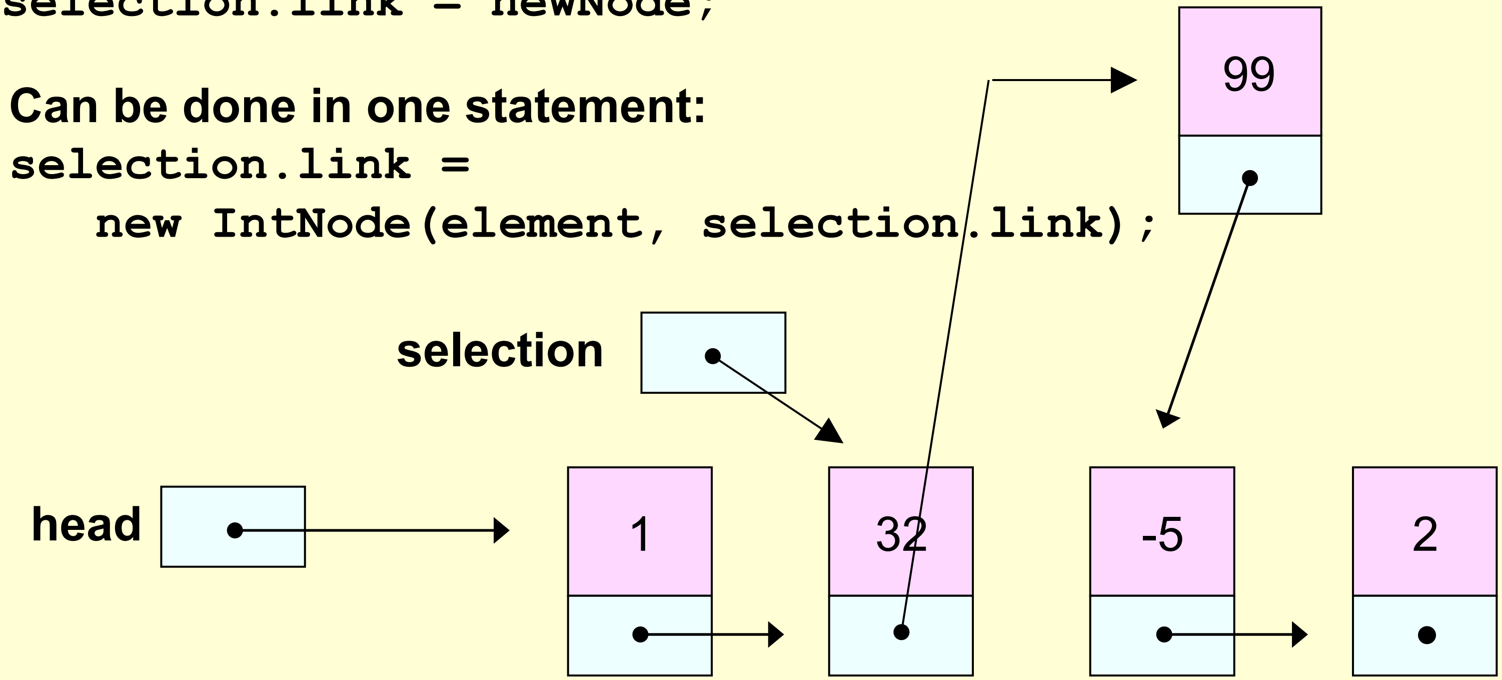
```
IntNode newNode =
   new IntNode(element, selection.link);
```

Make selection.link reference the new node

```
selection.link = newNode;
```

Can be done in one statement:

```
selection.link =
   new IntNode(element, selection.link);
```



Adding node after tail

Assume that tail is a reference to the tail of the list

Create new node and set its link

```
IntNode newNode = new IntNode(element, null);
```

same as general result with selection.link replaced by null

Make tail.link reference the new node

```
tail.link = newNode;
```

Simplify and update tail:

```
tail.link = new IntNode(element, null);
tail = tail.link;
```

addNodeAfter method

- If selection is a reference to the node we want to add after then

  -
    ```
    selection.link =
        new IntNode(element, selection.link);
    ```
- Letting selection be "this" we get

```
public void addNodeAfter(int element)
{
   link = new IntNode(element, link);
}
```

Constructing lists (1)

We can now construct a list by first constructing a one-eleement list and then using addNodeAfter
Following statements construct the list <1, 32, -5, 2>

```
IntNode head = new IntNode(1, null); // <1>
head.addNodeAfter(2); // <1,2>
head.addNodeAfter(-5); // <1,-5,2>
head.addNodeAfter(32); // <1,32,-5,2>
```

Note that this is a strange way to construct a list since we are always adding after the head: we construct one-element list, then add remaining nodes in reverse order.

Constructing lists (2)

Same list <1, 32, -5, 2> can be constructed using the following statements.

```
IntNode head = new IntNode(1, null);
head.addNodeAfter(32);
head.getLink().addNodeAfter(-5);
head.getLink().getLink().addNodeAfter(2);
```

*Note that since we are outside the **IntNode** class we cannot use expressions like **head.link** since link is a private data field. Instead we must use **head.getLink()***

Testing addNodeAfter (1)

Testing addNodeAfter to add a node with data 15 after the head
For the list <10,20,30,40> the result is <10,15,20,30,40>

```
IntNode head =
   new IntNode(10, new IntNode(20,
   new IntNode(30, new IntNode(40, null))));

head.addNodeAfter(15);

System.out.println("After add = " + head);
```

Testing addNodeAfter (2)

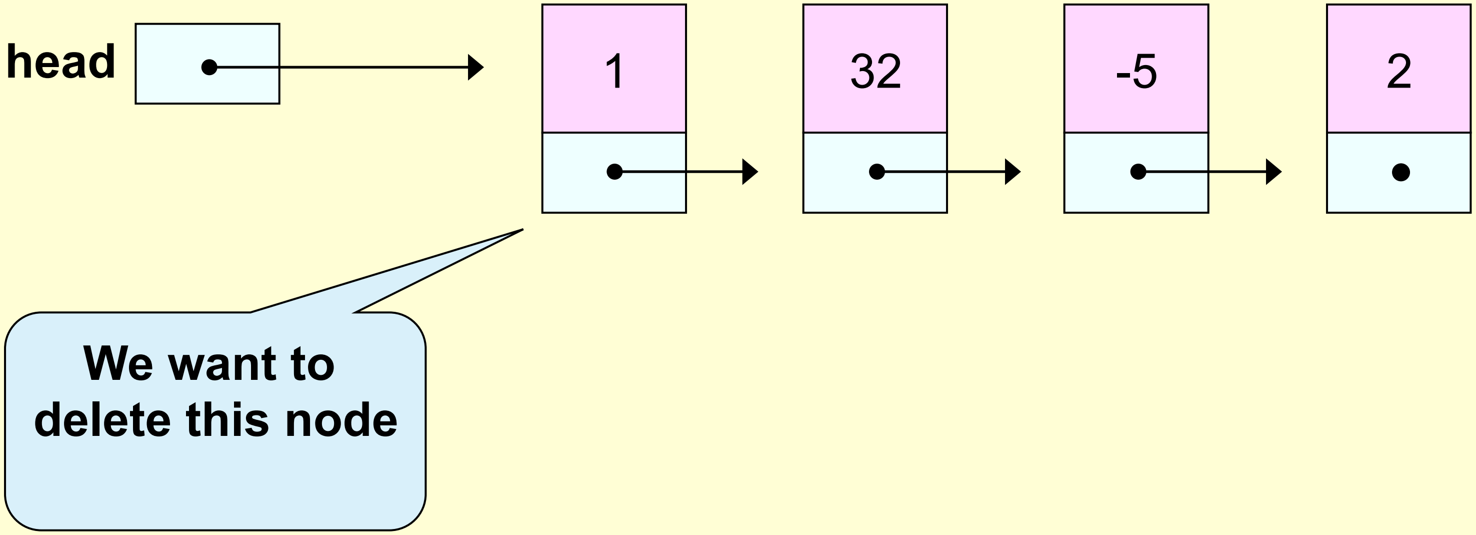
Testing addNodeAfter to add a node with data 50 after the last node of a list.
For the list <10,20,30,40> the result is <10,20,30,40,50>

```
IntNode head =
   new IntNode(10, new IntNode(20,
   new IntNode(30, new IntNode(40, null))));

IntNode tail =
    head.getLink().getLink(),getLink();
tail.addNodeAfter(50);

 System.out.println("After add = " + head);
```

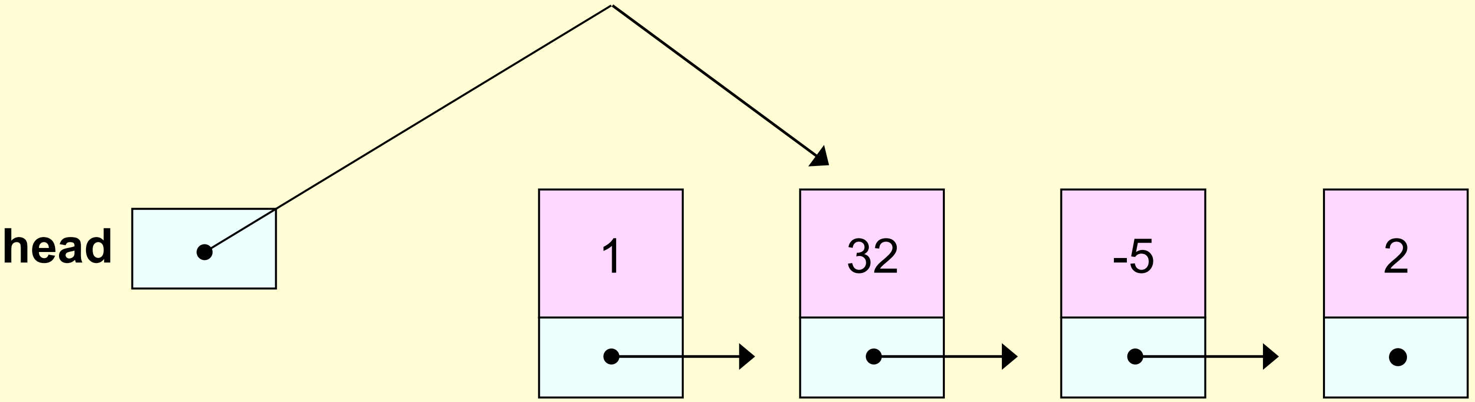
Removing node from head (1-4)



We want to delete this node

Removing node from head (2-4)

```
head = head.getLink();
```

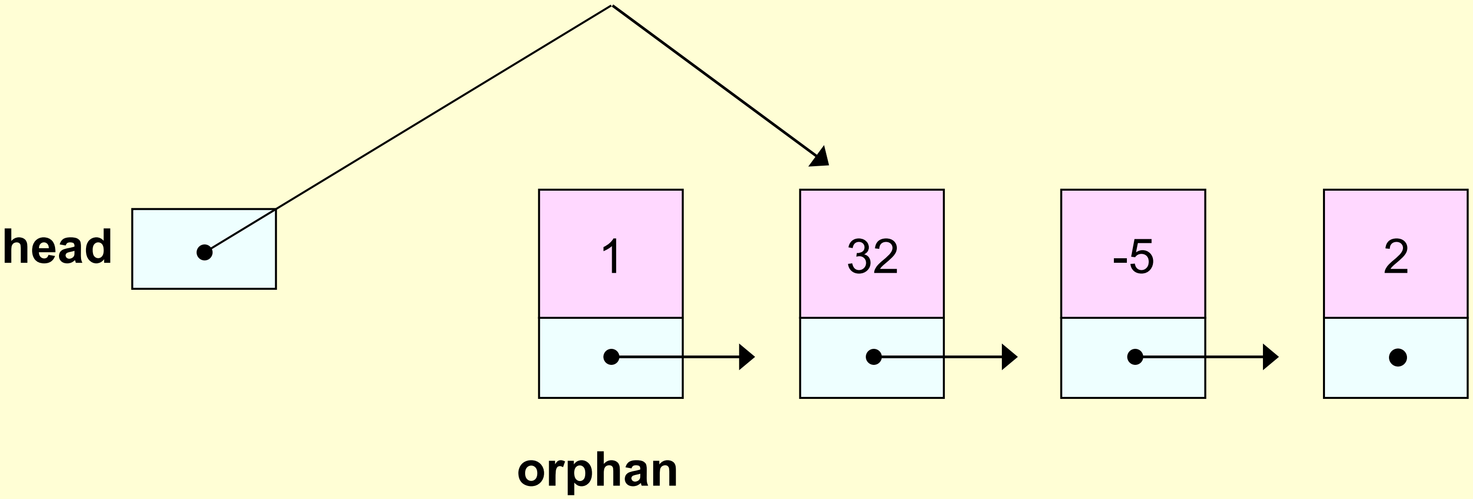


If inside the `IntNode` class we can use

```
head = head.link;
```

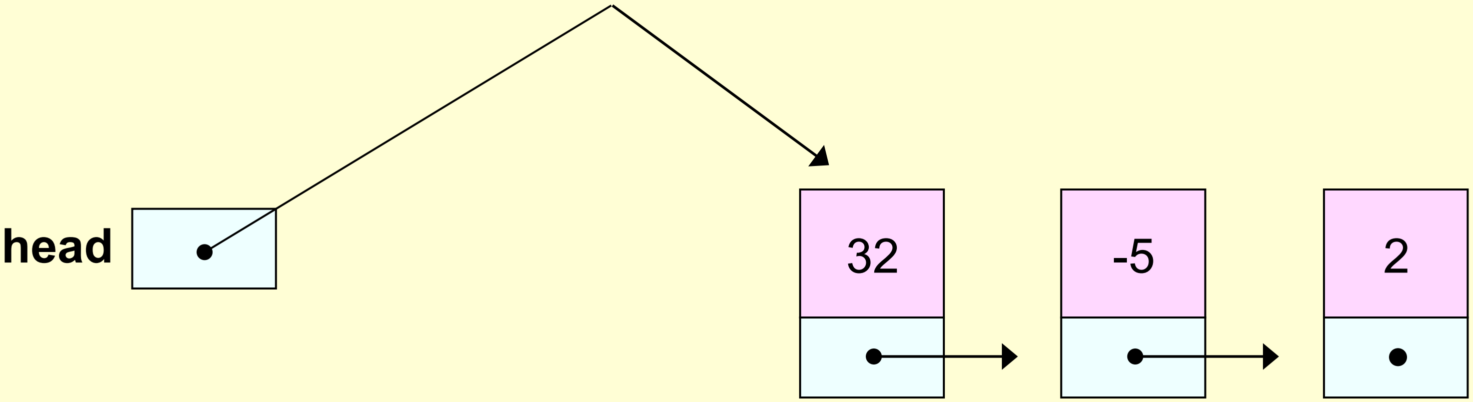
Removing node from head (3-4)

```
head = head.getLink();
```



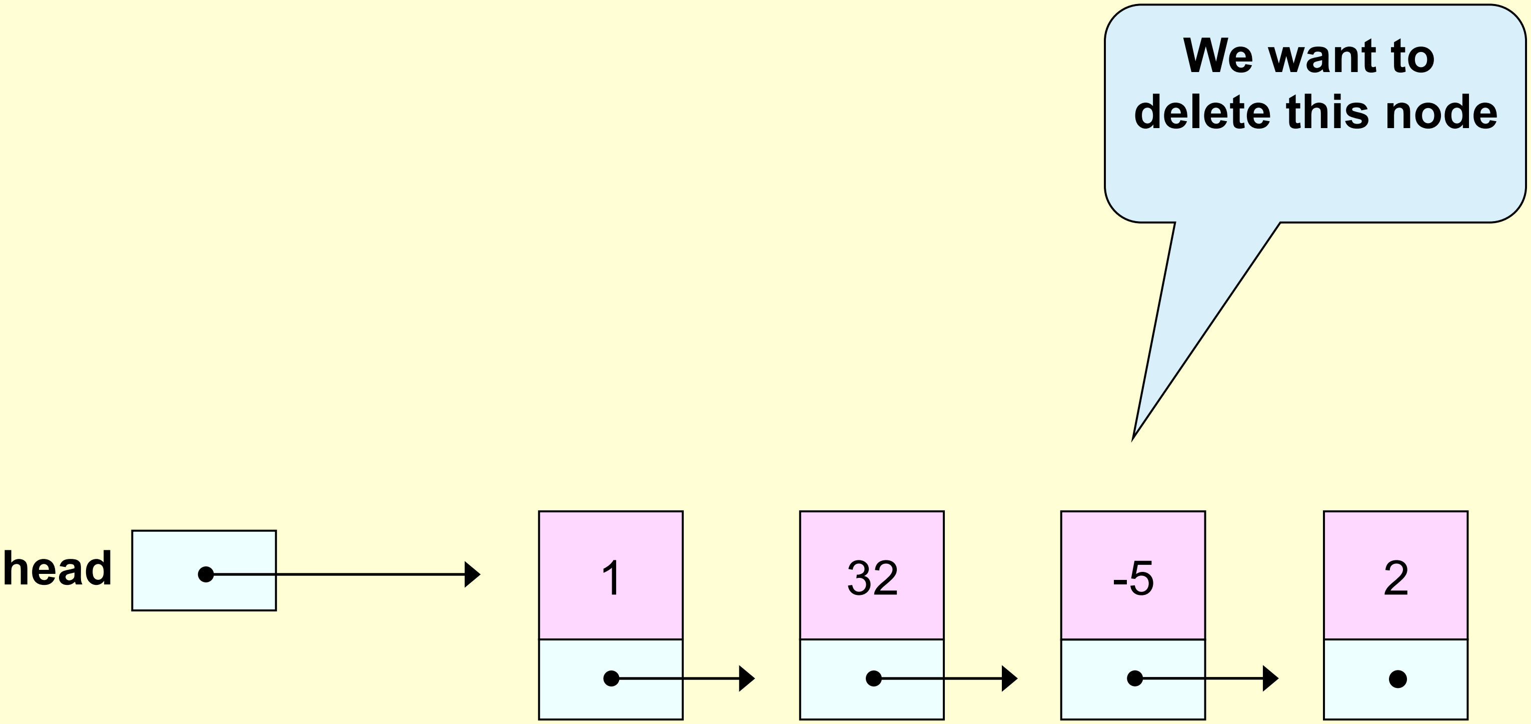
Removing node from head (4-4)

```
head = head.getLink();
```

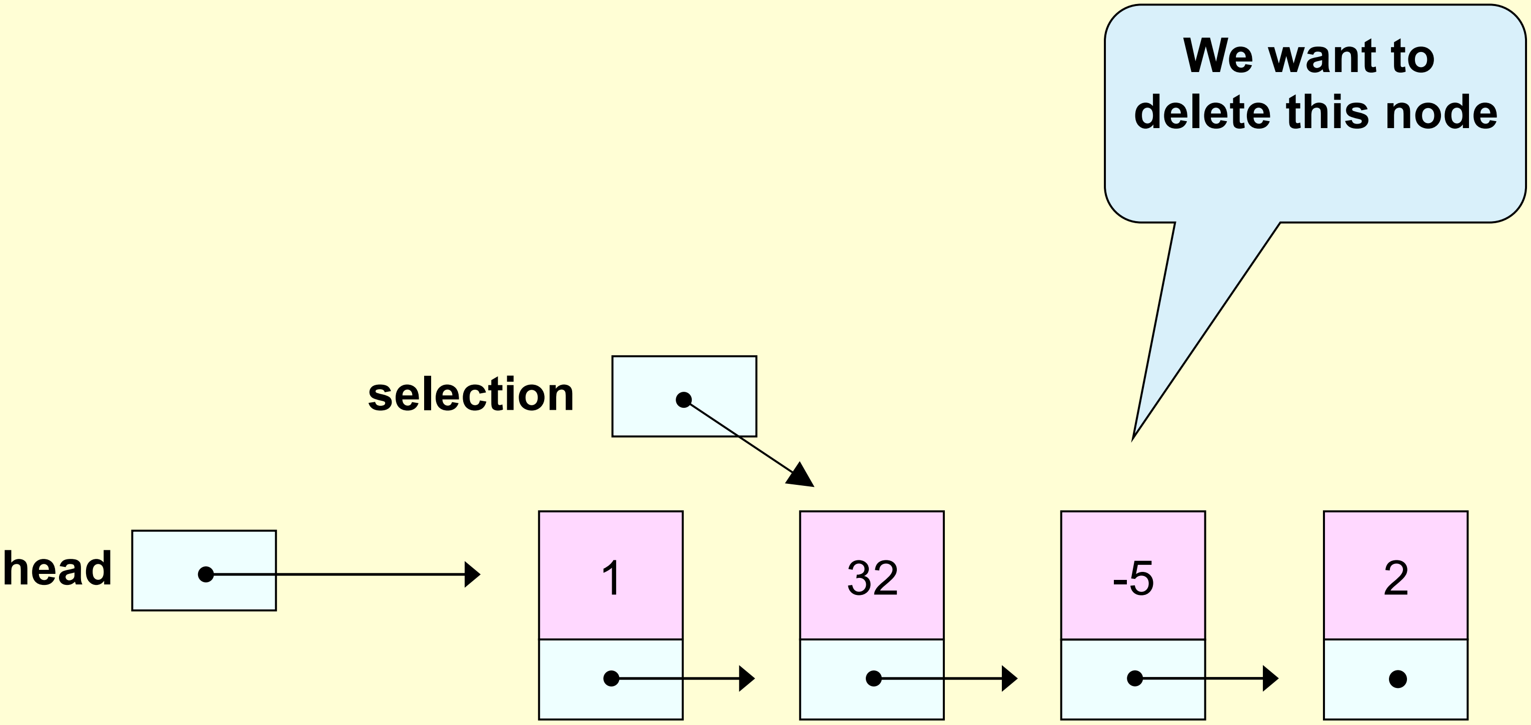


Works even for a one-element list since `head.getLink()` has the value `null` in this case.

Removing node after a node (1-4)

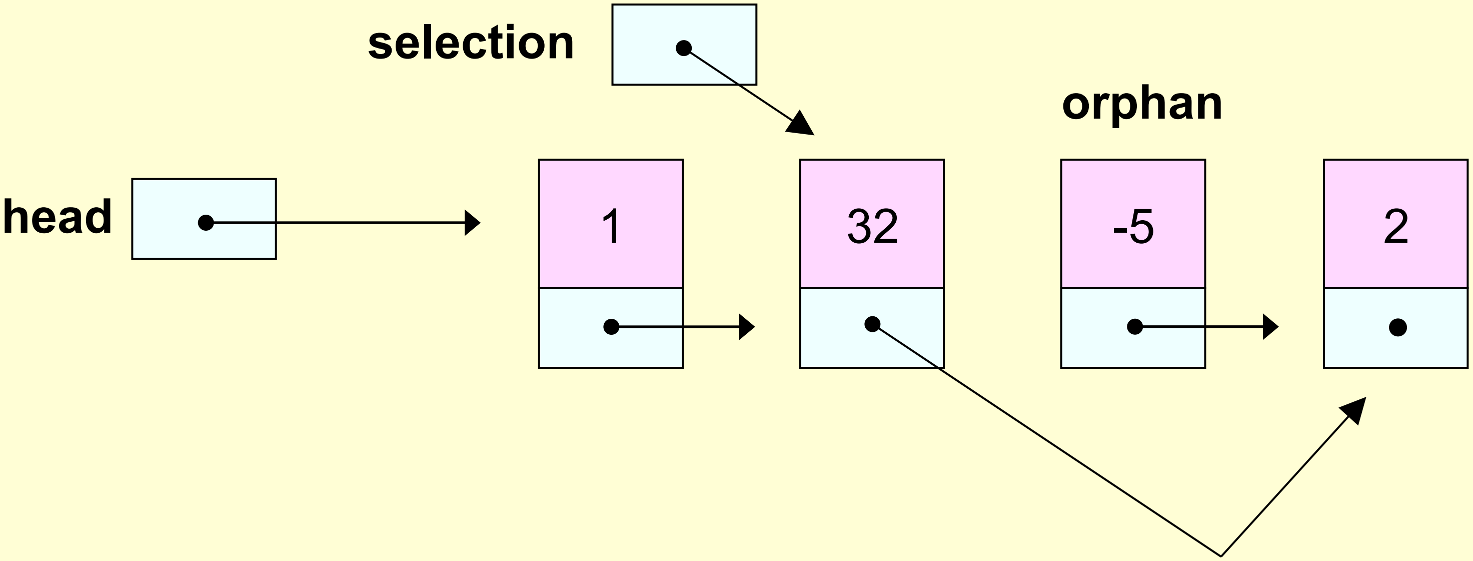


Removing node after a node (2-4)



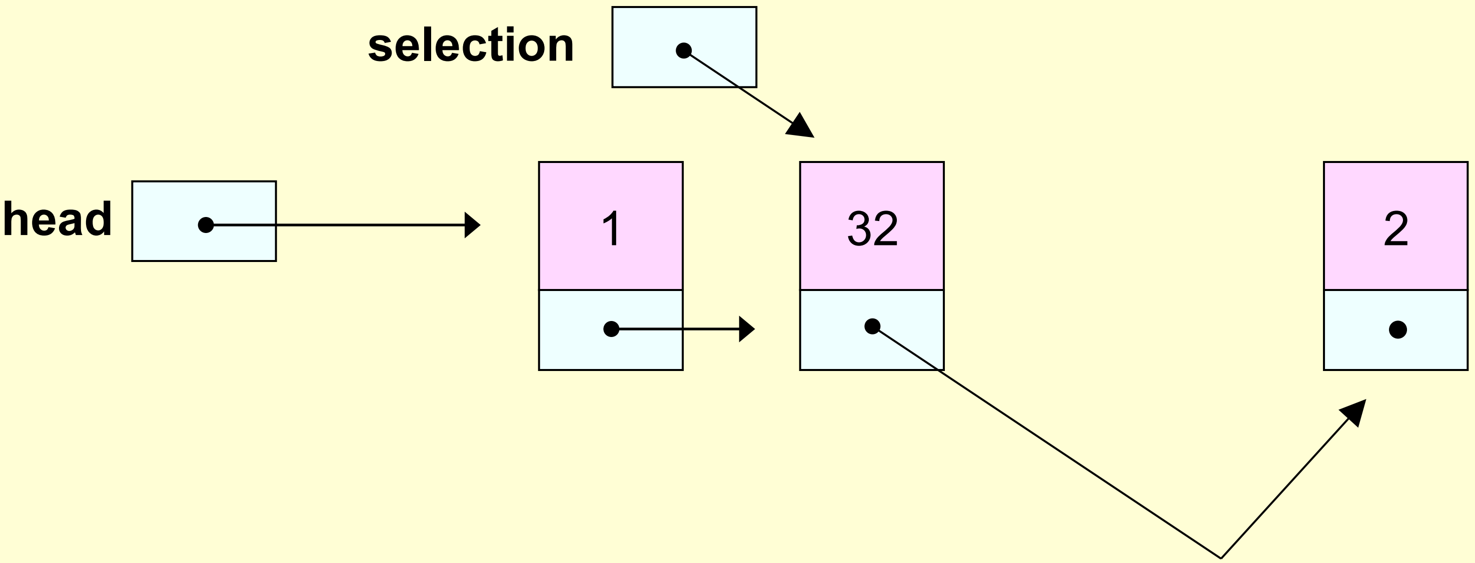
Removing node after a node (3-4)

```
selection.link = selection.link.link;
```



Removing node after a node (4-4)

selection.link = selection.link.link;



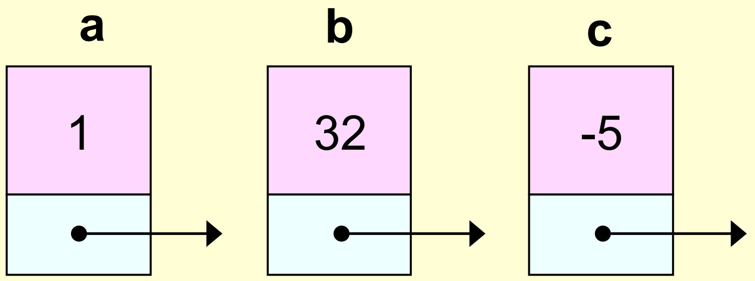
removeNodeAfter method

- If selection is a reference to the node we want to remove after then
    - `selection.link = selection.link.link;`
- Letting selection be "this" we get

```
public void removeNodeAfter(int element)
{
    link = link.link;
}
```

Another way: link=link.link;

- Consider 3 nodes labelled a, b, c.



- We can delete (remove) node b using

```
a.link = c;           // skip over b
a.link = b.link;      // since c is b.link
a.link = a.link.link; // since b is a.link
link = link.link;     // letting a be "this"
```

Testing removeNodeAfter (1)

Testing removeNodeAfter to remove the node after the head
For the list <10,20,30,40> the result is <10,30,40>

```
IntNode head =
   new IntNode(10, new IntNode(20,
   new IntNode(30, new IntNode(40, null))));

head.removeNodeAfter();

System.out.println("After remove = " + head);
```

Important Note: To remove the head node it is always necessary to use head = head.getLink();

Testing removeNodeAfter (2)

Testing removeNodeAfter to remove the last node of a list
For the list <10,20,30,40> the result is <10,30,40>

```
IntNode head =
   new IntNode(10, new IntNode(20,
   new IntNode(30, new IntNode(40, null))));

 IntNode beforeTail = head.getLink().getLink();
 beforeTail.removeNodeAfter();

 System.out.println("After remove = " + head);
```

Note: If removeNodeAfter is applied to the last node of a list an exception is thrown.

Length of a list (1)

- The static listLength method needs to count the number of nodes in a list.
- To do this we need to traverse the list:
  - begin with a reference to the head and advance this reference until it reaches the null reference in the last node
  - see toString for another traversal example
- Each time we advance the reference we add 1 to a counter.

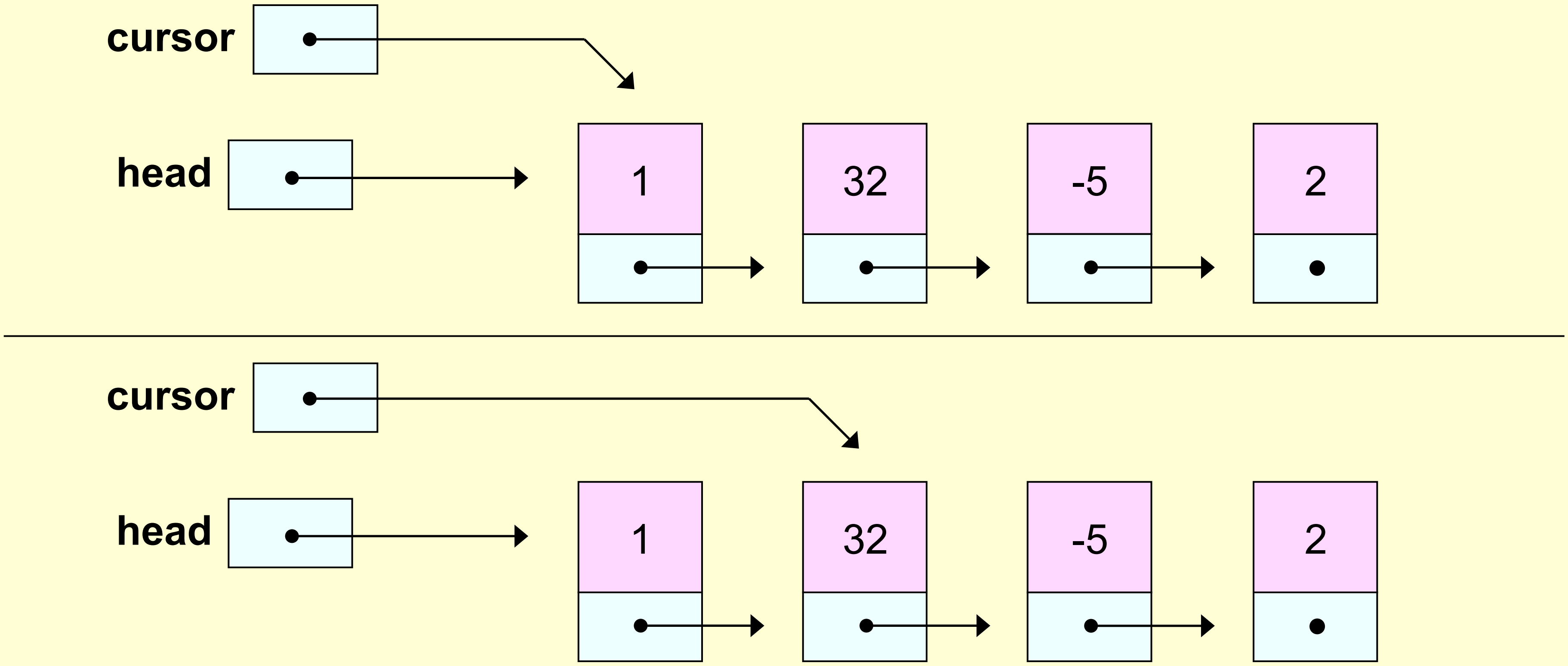
Length of a list (2)

- Pseudo-code for list traversal

```
cursor ← first node of list
WHILE cursor is not null DO
   count ← count + 1
   advance cursor to next node
END WHILE
```

- A for loop can also be used

Length of a list (3)



Cursor is advanced using `cursor = cursor.link;`

Length of a list (4)

Using a while loop

```
IntNode cursor = head;
int count = 0;
while (cursor != null)
{  count++;
   cursor = cursor.link;
}
```

Using a for loop

```
IntNode cursor;
int count = 0;
for (cursor = head; cursor != null;
                          cursor = cursor.link)
   count++;
```

static listLength method

**listLength** returns number of nodes in a list of type **intNode**. 0 is returned if the list is empty. A for loop is used here but a while loop could also be used

```
public static int listLength(IntNode head)
{
   int count = 0;
   for (IntNode cursor = head;
        cursor != null; cursor = cursor.link)
   {
      count++;
   }
   return count;
}
```

General traversal model (1)

For loop traversal model for a linked list

```
for (IntNode cursor = head;
   cursor != null; cursor = cursor.link)
{
   // process the data in the node
   // referenced by cursor

}
```

General traversal model (2)

while loop traversal model for a linked list

```
IntNode cursor = head;

while (cursor != null)
{
   // Process the data in the node
   // referenced by cursor

   cursor = cursor.link; // advance to next node
}
```

Example: sum of integers

A traversal can be used to sum the integers in the nodes of an IntNode list

```
IntNode cursor = head;
int sum = 0;

while (cursor != null)
{
   sum += cursor.getData();

   cursor = cursor.link; // advance to next node
}
```

Example: toString method

- We have added **`toString`** since it is useful for displaying a list when testing.

```
public String toString()
{  StringBuffer s = new StringBuffer();
   s.append("IntNode[");
   IntNode current = this;
   while (current != null)
   {  s.append(current.data);
      if (current.link != null) s.append(",");
      current = current.link;
   }
   s.append("]");
   return s.toString();
}
```

Searching a list

- The static **`listSearch`** method searches a list for a given integer and returns a reference to the **`IntNode`** containing the data.
- If the data is not found then the method returns a null reference
- The list traversal model can be used here

static listSearch method

Search a list for a given data element called target and return a reference to the element if it is found else return null.

```
public static IntNode listSearch(IntNode head,
   int target)
{  for (IntNode cursor = head; cursor != null;
      cursor = cursor.link)
   {
      if (target == cursor.data)
      {
         return cursor;
      }
   }
   return null;
}
```

Testing listSearch

```
IntNode head =
   new IntNode(10, new IntNode(20,
   new IntNode(30, new IntNode(40, null))));
System.out.println(IntNode.listSearch(head, 10));
System.out.println(IntNode.listSearch(head, 20));
System.out.println(IntNode.listSearch(head, 30));
System.out.println(IntNode.listSearch(head, 40));
System.out.println(IntNode.listSearch(head, 50));
```

Results displayed are

```
IntNode[10, 20, 30,40]
IntNode[20, 30, 40]
IntNode[30,40]
IntNode[40]
[ ]
```

these results show the list whose head is the node containing the data found

list search by position

- Instead of searching for a given data element in a list and returning a reference to the node containing the data we can search for the node in a given position and return the reference to the node at that position.
- We assume here that positions in a list are labelled beginning at 1 instead of 0

static listPosition method

The **`listPosition`** *method returns the reference to a node in the list given its position (1,2,3,...). If there no such position then null is returned and if the specified position is not positive then an exception is thrown.*

```
public static IntNode listPosition(
   IntNode head, int position)
{
   if (position <= 0)
      throw new IllegalArgumentException("...");
   IntNode cursor = head;
   for (int i = 1;
        (i < position) && (cursor != null); i++)
      cursor = cursor.link;
   return cursor;
}
```

Testing listPosition

```
IntNode head =  new IntNode(10, new IntNode(20,
   new IntNode(30, new IntNode(40, null))));
System.out.println(IntNode.listPosition(head,1));
System.out.println(IntNode.listPosition(head,2));
System.out.println(IntNode.listPosition(head,3));
System.out.println(IntNode.listPosition(head,4));
System.out.println(IntNode.listPosition(head,5));
```

Results displayed are

```
IntNode[10, 20, 30, 40]
IntNode[20, 30, 40]
IntNode[30, 40]
IntNode[40]
[ ]
```

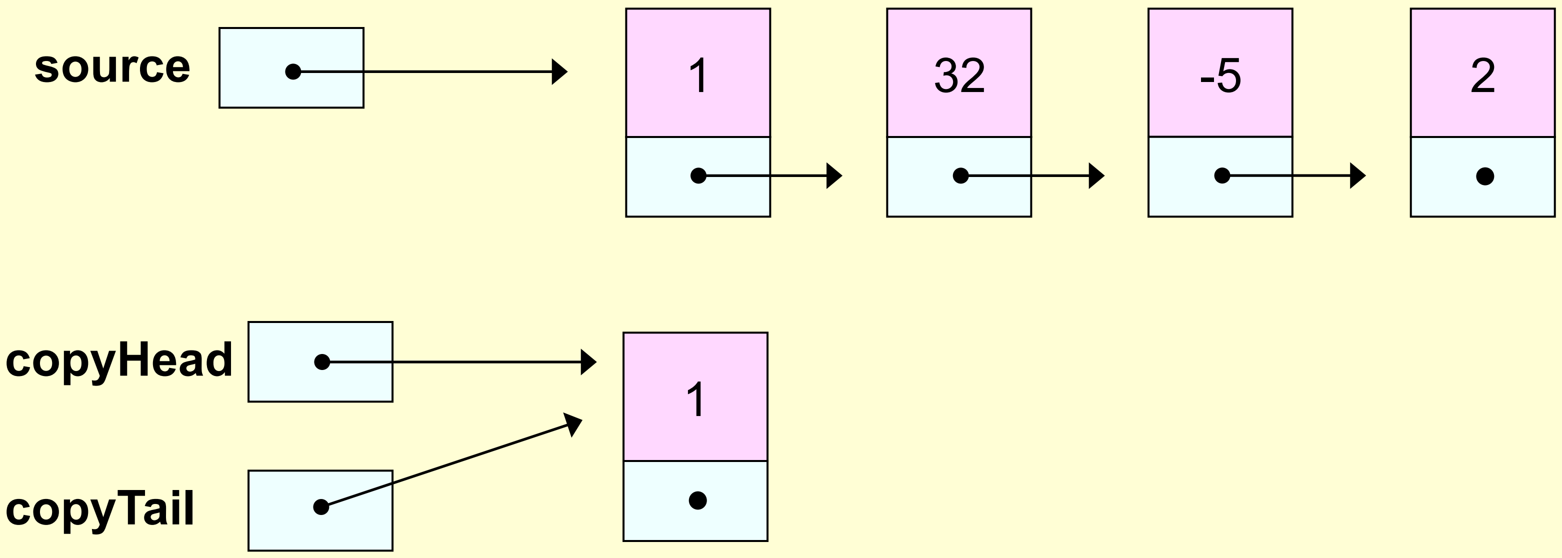
these results show the list whose head is the node containing the data found

Non-positive position throw exception, positions > 5 return null

Copying a list (1)

Make a copy of the head of the `source` list.
Set up two references `copyHead` and `copyTail` to it

```
IntNode copyHead = new IntNode(source.data, null);
IntNode copyTail = copyHead;
```



source

1

32

-5

2

copyHead

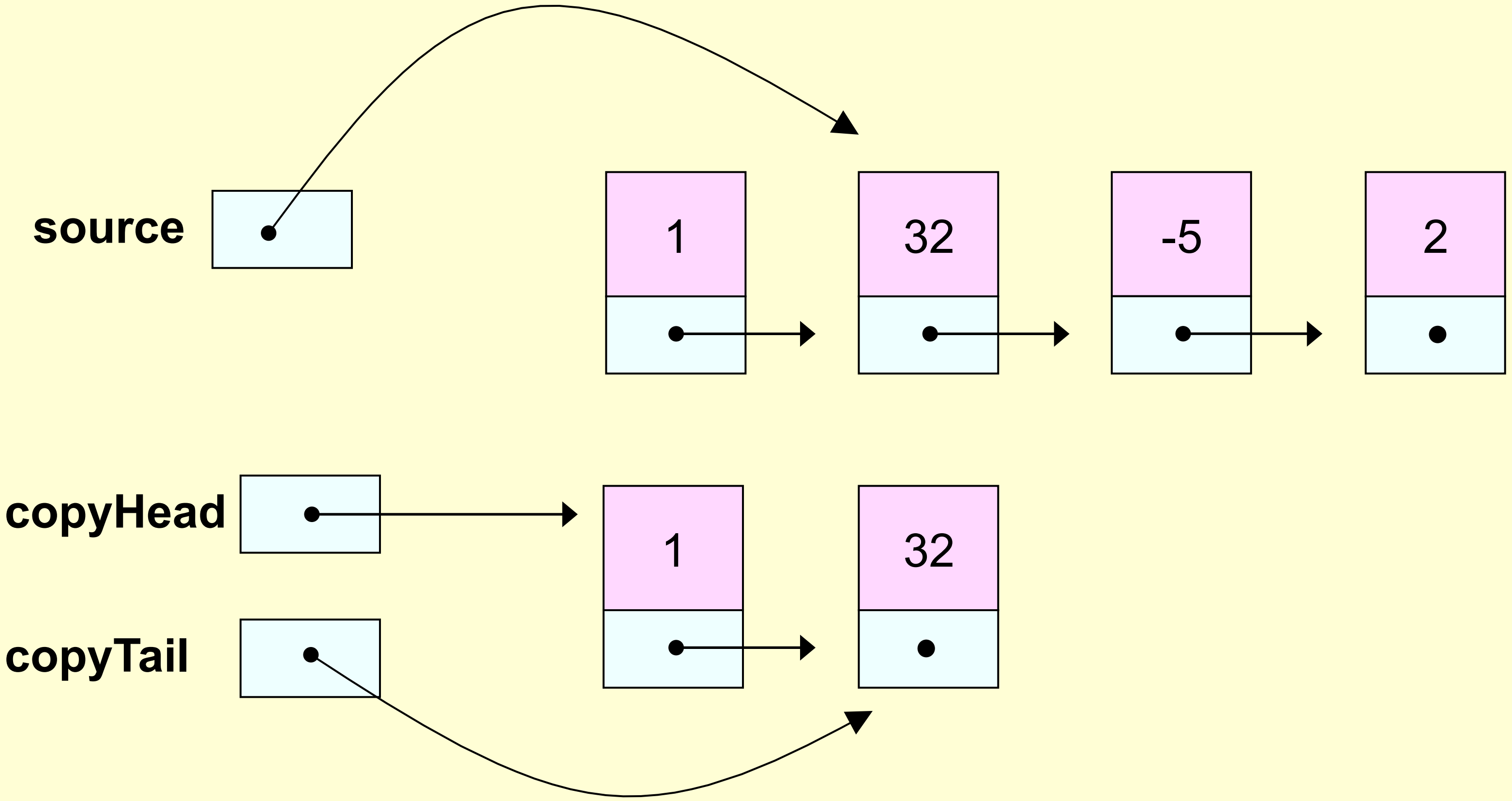
1

copyTail

Copying a list (2)

Now do the statements

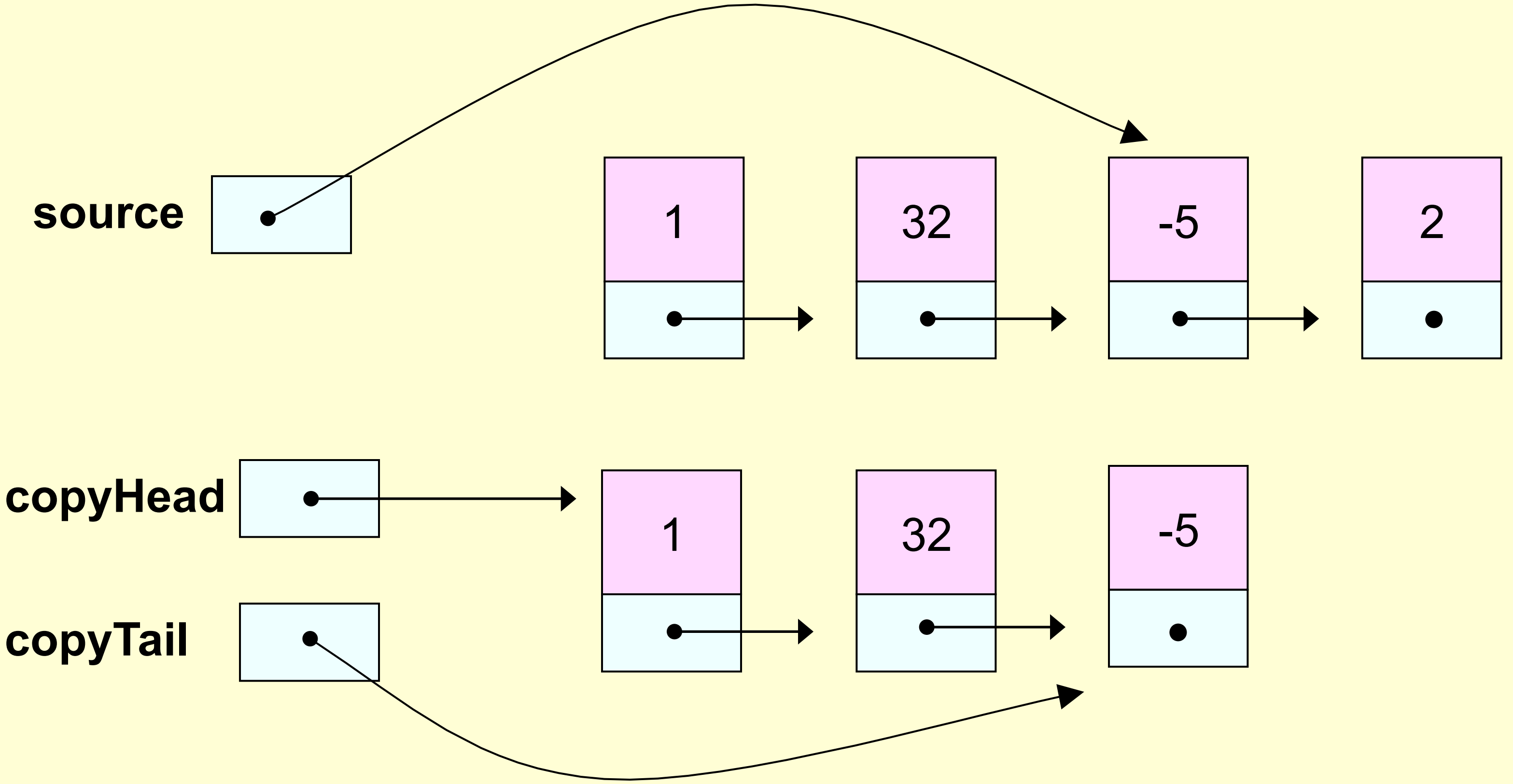
```
source = source.link;
copyTail.addNodeAfter(source.data);
copyTail = copyTail.link;
```



Copying a list (3)

Now do the same statements again

```
source = source.link;
copyTail.addNodeAfter(source.data);
copyTail = copyTail.link;
```

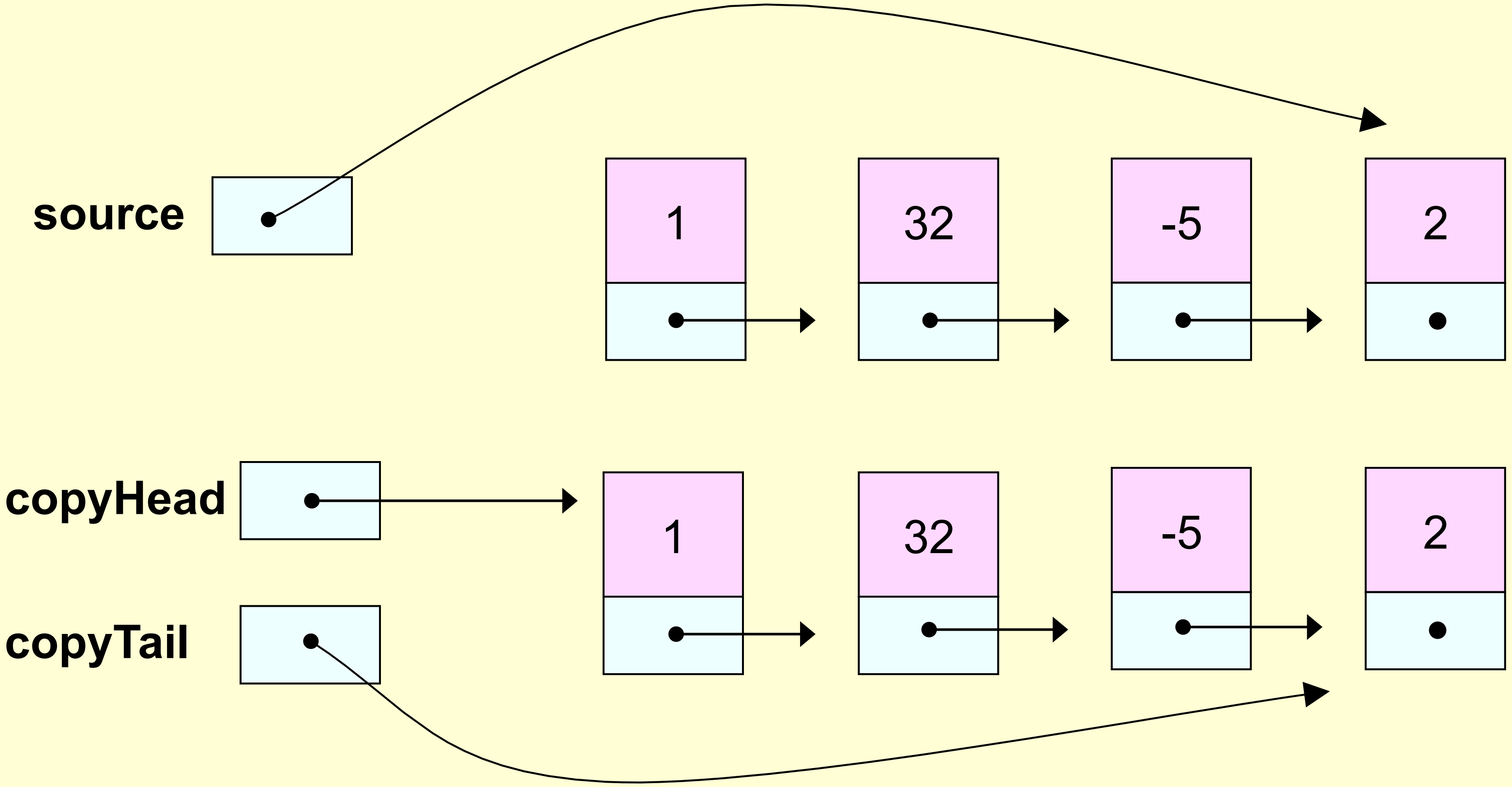


Copying a list (4)

We are done now since `source.link` is null

Now do the same statements again

```
source = source.link;
copyTail.addNodeAfter(source.data);
copyTail = copyTail.link;
```



static listCopy method

Given a list make a copy

```
public static IntNode listCopy(IntNode source)
{
   if (source == null) return null;
   IntNode copyHead =
                  new IntNode(source.data, null);
   IntNode copyTail = copyHead;

   while (source.link != null)
   {
      source = source.link; // advance
      copyTail.addNodeAfter(source.data);
      copyTail = copyTail.link; // advance
   }
   return copyHead;
}
```

another version of listCopy

This version does not use addNodeAfter

```
public static IntNode listCopy(IntNode source)
{
   if (source == null) return null;
   IntNode copyHead =
                  new IntNode(source.data, null);
   IntNode copyTail = copyHead;

   while (source.link != null)
   {  source = source.link; // advance
      copyTail.link =
         new IntNode(source.data, null);
      copyTail = copyTail.link; // advance
   }
   return copyHead;
}
```

testing listCopy method (1)

First test listCopy on the empty list

```
IntNode head = null;
IntNode copy = IntNode.listCopy(head)
System.out.println("List = " + head);
System.out.println("List copy = " + copy);
```

Result is [ ] for both lists

testing listCopy method (2)

Test listCopy on a non-empty list

```
IntNode head = new IntNode(10, new IntNode(20,
   new IntNode(30, new IntNode(40, null))));
IntNode copy = IntNode.listCopy(head);

System.out.println("list = " + head);
System.out.println("copy = " + copy);

copy.setData(99); // <99,20,30,40>
copy.addNodeAfter(99); // <99,99,20,30,40>

// original is still <10,20,30,40>

System.out.println("list = " + head);
System.out.println("copy = " + copy);
```

Check cases

- source is an empty list
  - copy will be null
- source is a one element list
  - source.link will be null so the while loop will never be executed
- source is a two or more element list
  - while loop executes until source refers to the last node which is copied before exiting the loop

copying with tail

- Sometimes it is useful to have a version of `listCopy` that returns the tail node
  - example: appending one list at end of another
- If we don't save a tail reference during the copy we will have to traverse the list again to find it in an operation like append
- Our method needs to return both the head and tail references to the copy

Returning multiple values

- Java can only return one value from a method
- We need to return two **`IntNode`** references
- This can be done by returning an array with two elements, element 0 can be the head reference and element 1 can be the tail reference.
- Could also define a pair object and return it

static listCopyWithTail method

Given a list make a copy, return references to both head and tail

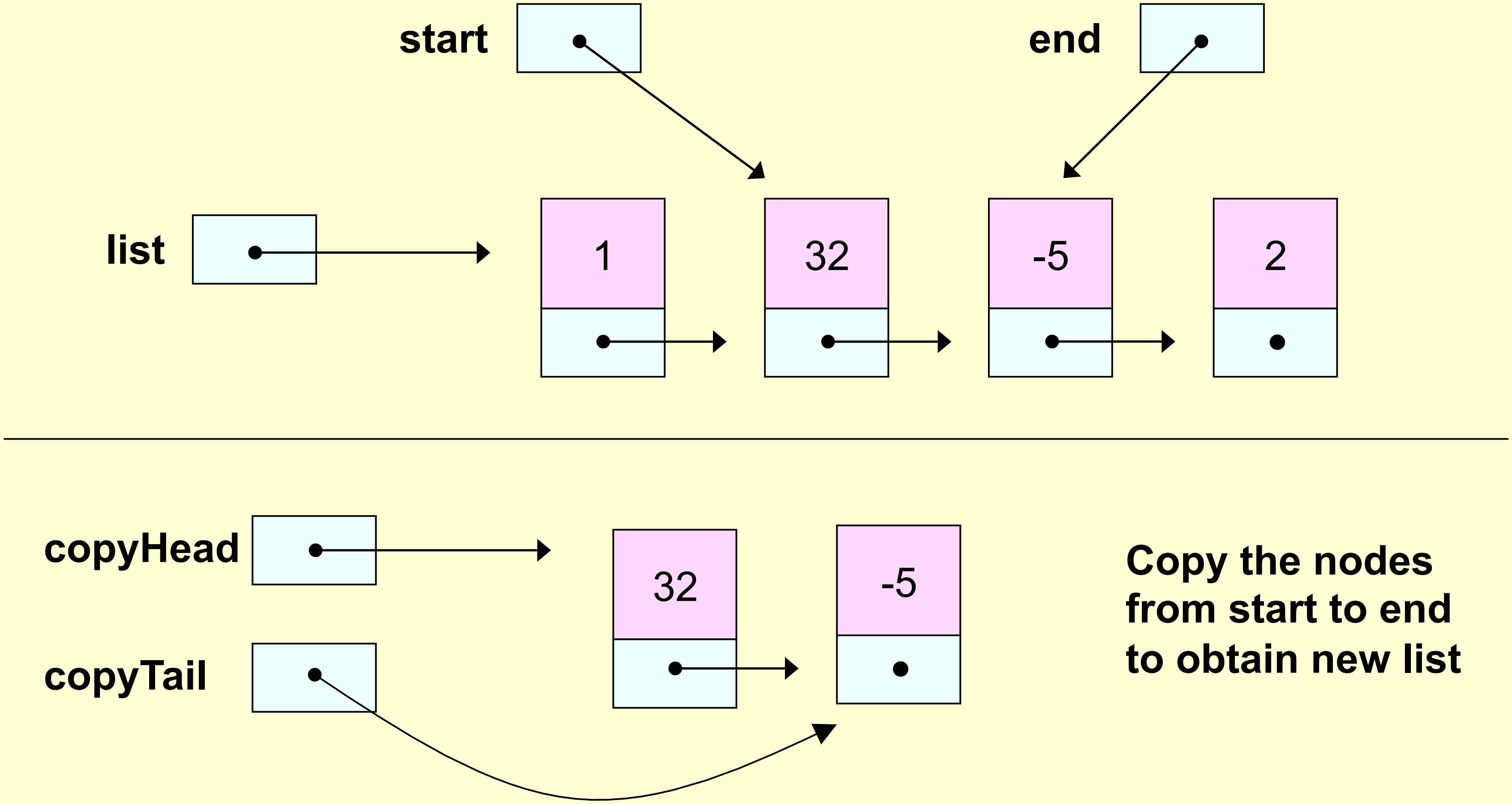
```
public static IntNode listCopyWithTail(
                                  IntNode source)
{  IntNode[] answer = new IntNode[2];
   if (source == null) return answer;
   IntNode copyHead =
                  new IntNode(source.data, null);
   IntNode copyTail = copyHead;
   while (source.link != null)
   {  source = source.link; // advance
      copyTail.addNodeAfter(source.data);
      copyTail = copyTail.link; // advance
   }
   answer[0] = copyHead; answer[1] = copyTail;
   return answer;
}
```

Copying part of a list (1)

- This is analogous to using substring in the String class to make a string that is a substring of a given String
- Here we want a method with prototype

  -
    ```
    static IntNode[] listPart(
        IntNode start, IntNode end);
    ```
- Return value [0] is a reference to head of new list and [1] is a reference to the tail

Copying part of a list (2)



Copy the nodes from start to end to obtain new list

static listPart method (1)

Given the two nodes start and end of a list make a copy of the sublist from start to end (inclusice) and return a reference to the head of the sublist and the tail of the sublist

```
public static IntNode[] listPart(IntNode start,
   IntNode end)
{
   IntNode copyHead, copyTail, cursor;
   IntNode[] answer = new IntNode[2];

   if (start == null)
      throw new IllegalArgumentException("...");
   if (end == null)
      throw new IllegalArgumentException("...");
```

static listPart method (2)

Now make first node of the new list and copy remaining nodes. Note the check for the end node in the while loop.

```
   copyHead = new IntNode(start.data, null);
   copyTail = copyHead;
   cursor = start;
   while (cursor != end)
   {  cursor = cursor.link;
      if (cursor == null) // end not found
         throw new IllegalArgumentException(".");
      copyTail.addNodeAfter(cursor.data);
      copyTail = copyTail.link;
   }
   answer[0] = copyHead; answer[1] = copyTail;
   return answer;
}
```

testing listPart method

Test listPart on list <10,20,30,40> with part given by <20,30>

```
IntNode list1 =  new IntNode(10, new IntNode(20,
   new IntNode(30, new IntNode(40, null))));

// Make start, end refer to 2nd and 3rd nodes
IntNode start = list1.getLink(); // data 20
IntNode end = start.getLink();   // data 30

IntNode[] part = IntNode.listPart(start, end);

System.out.println("head = " + part[0]);
System.out.println("tail = " + part[1]);
```

```
head = IntNode[20, 30]
tail = IntNode[30]
```

Exercise: concatenation

- Write a method with prototype

  -
    ```
    static IntNode concatenate(
          IntNode list1, IntNode list2);
    ```
- The method should use **`listCopy`** and **`listCopyWithTail`** to return a new list which is the concatenation of the two given lists **`list1`** and **`list2`**.
- **`list1`** and **`list2`** are unchanged.

Exercise: another toString

- Consider the following toString method

```
public String toString()
{
    return
    "IntNode[" + data + ", " + link + "]";
}
```

- What output does this method produce?

IntNode class summary (1)

- Defines nodes for integer data
- Unconventional approach because the instance methods addNodeAfter, removeNodeAfter are part of the node class not a separate LinkedList class.
- This means they cannot be applied to an empty list so we must always construct the head of the list first.

IntNode class summary (2)

- Static methods in the IntNode class are "helper methods"
- Their purpose is to help develop linked implementations of other ADT's
- We will show how to do this for a linked implementation of the IntBag class called IntLinkedBag