Linked Lists

Chapters 5

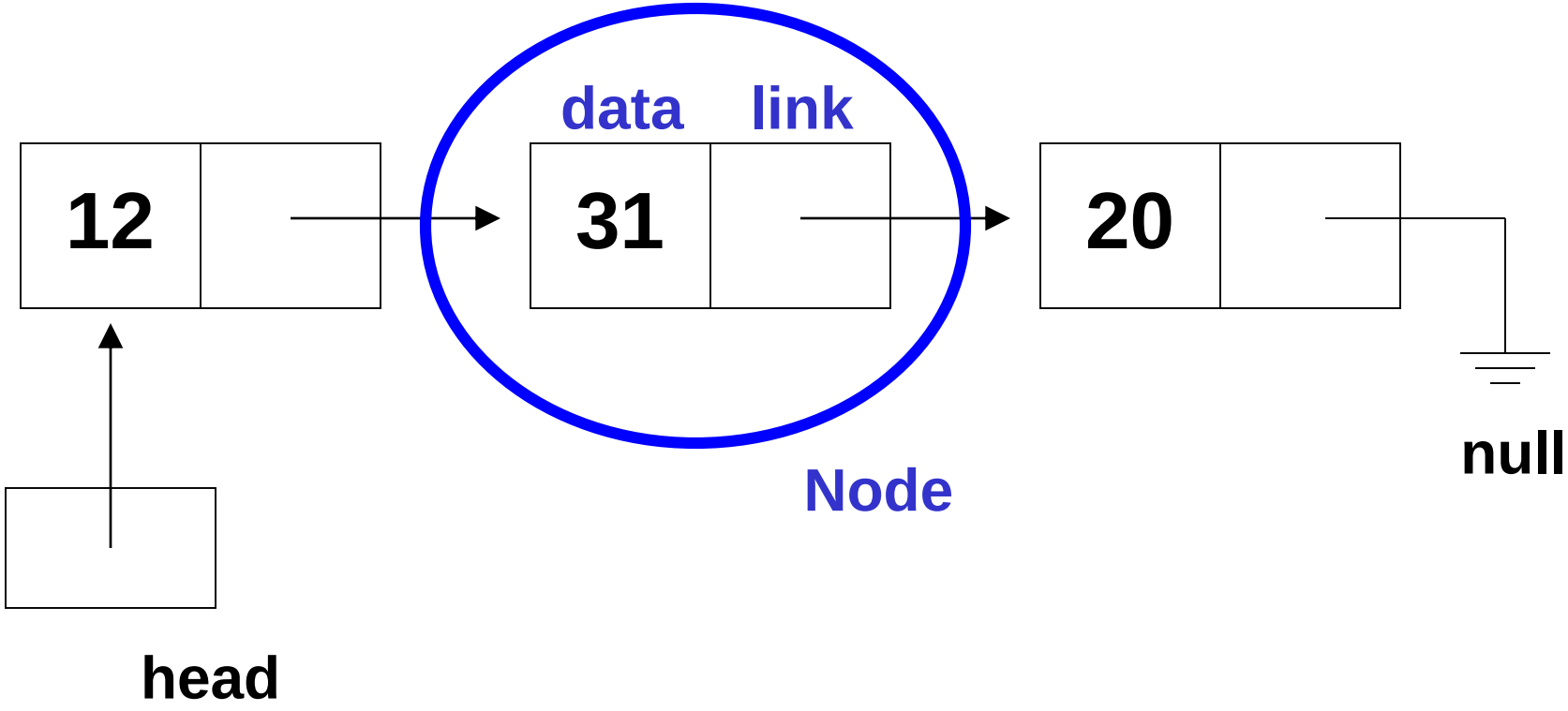
Fundamentals

- A singly-linked list is a sequence of data elements (of the same type) arranged one after another conceptually.
- Each element is stored in a node.
- Each node also contains a link that connects this node to the next node in the list.

Fundamentals (cont'd)

- A special link called the head references the first node in a list.
- Some lists may also have a special link called the tail that references the last node in a list.
- A cursor is a link that points to one of the nodes of the list.
- A list may be empty. (i.e. head = tail = null).

Conceptual Picture



Actual Picture

WORD ADDRESS	CONTENTS
1000	31 (2nd number)
1002	1008
1004	12 (1st number)
1006	1000
1008	20 (3rd number)
1010	0
1012	1004

head

Defining a Node

```
public class IntNode
{
    private int data;
    private IntNode link;

    // IntNode methods
}
```

Constructor

```
public IntNode(int initialData)
{
    data = initialData;
    link = null;
}
```

Accessor Methods

```
public int getData()
{
    return data;
}
public IntNode getLink()
{
    return link;
}
```

Mutator Methods

```
public void setData(int newData)
{
     data = newData;
}
public void setLink(IntNode newLink)
{
     link = newLink;
}
```

Defining a Linked List

```
public class IntList
{
    private IntNode head;
    private IntNode tail;
    private IntNode cursor;

    // IntList methods
}
```

Constructor

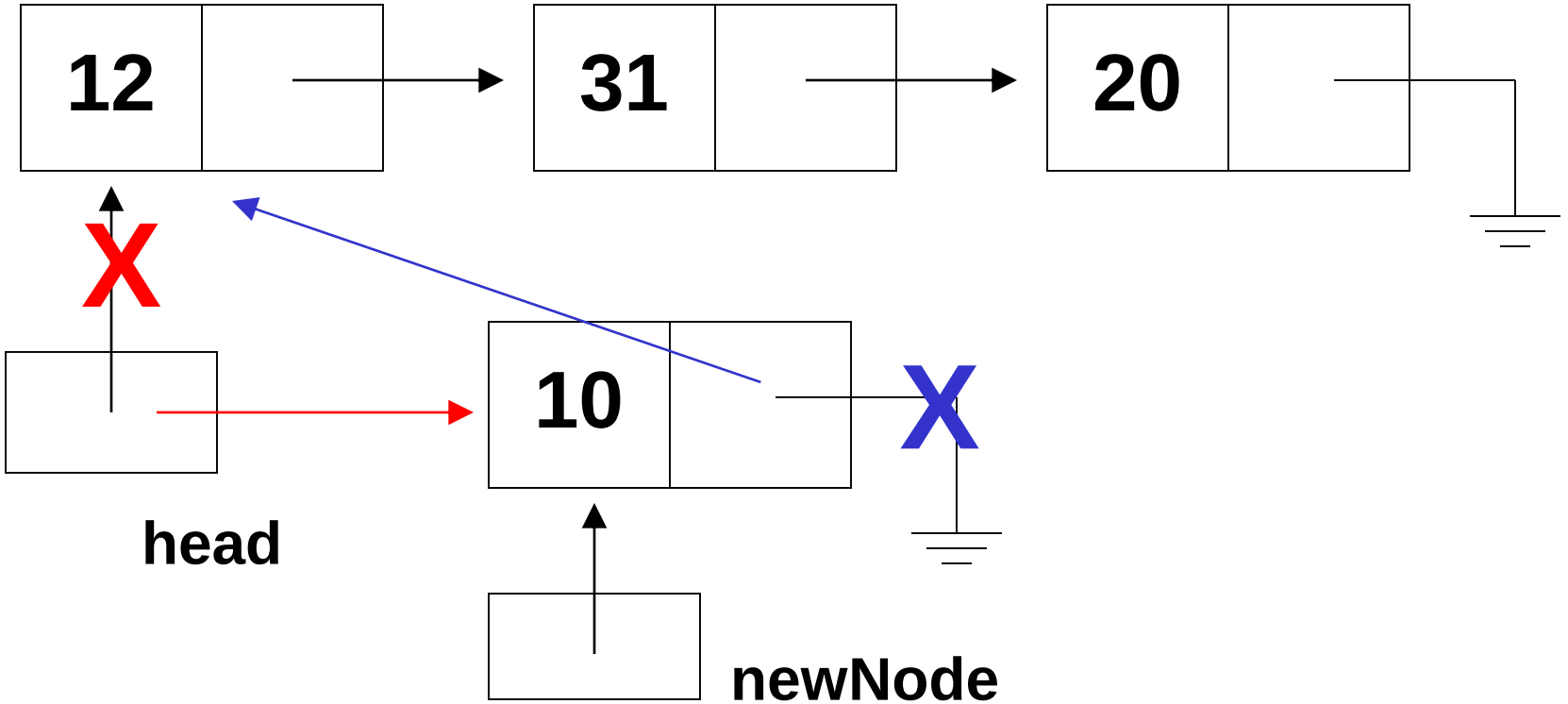
```
public IntList()
{
    head = null;
    tail = null;
    cursor = null;
}
```

Add to head of list

newNode = new IntNode(element);

newNode.setLink(head);

head = newNode;

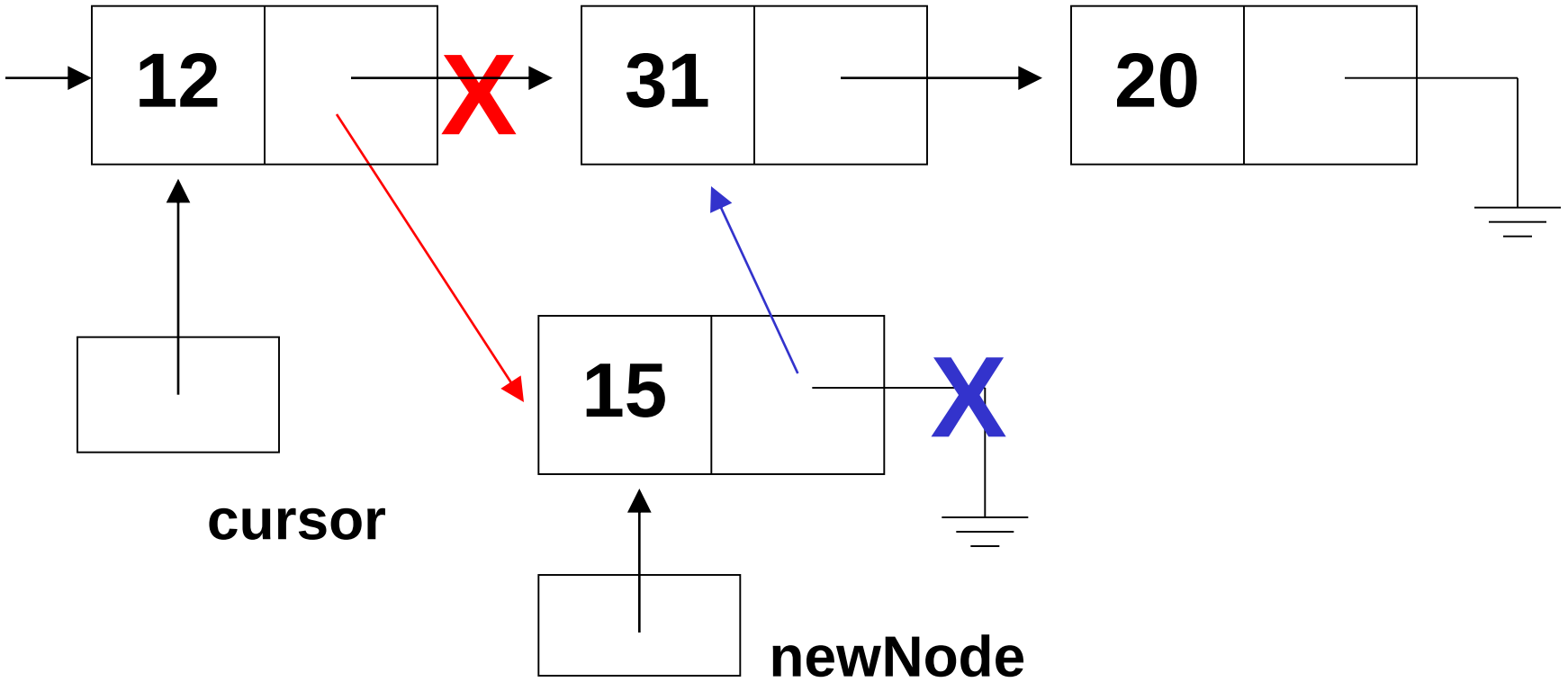


Add a new head of list

```
public void addNewHead(int element) {
      IntNode newNode =
           new IntNode(element);
      newNode.setLink(head);
      head = newNode;
      if (tail == null) tail = head;
      cursor = head;
}
```

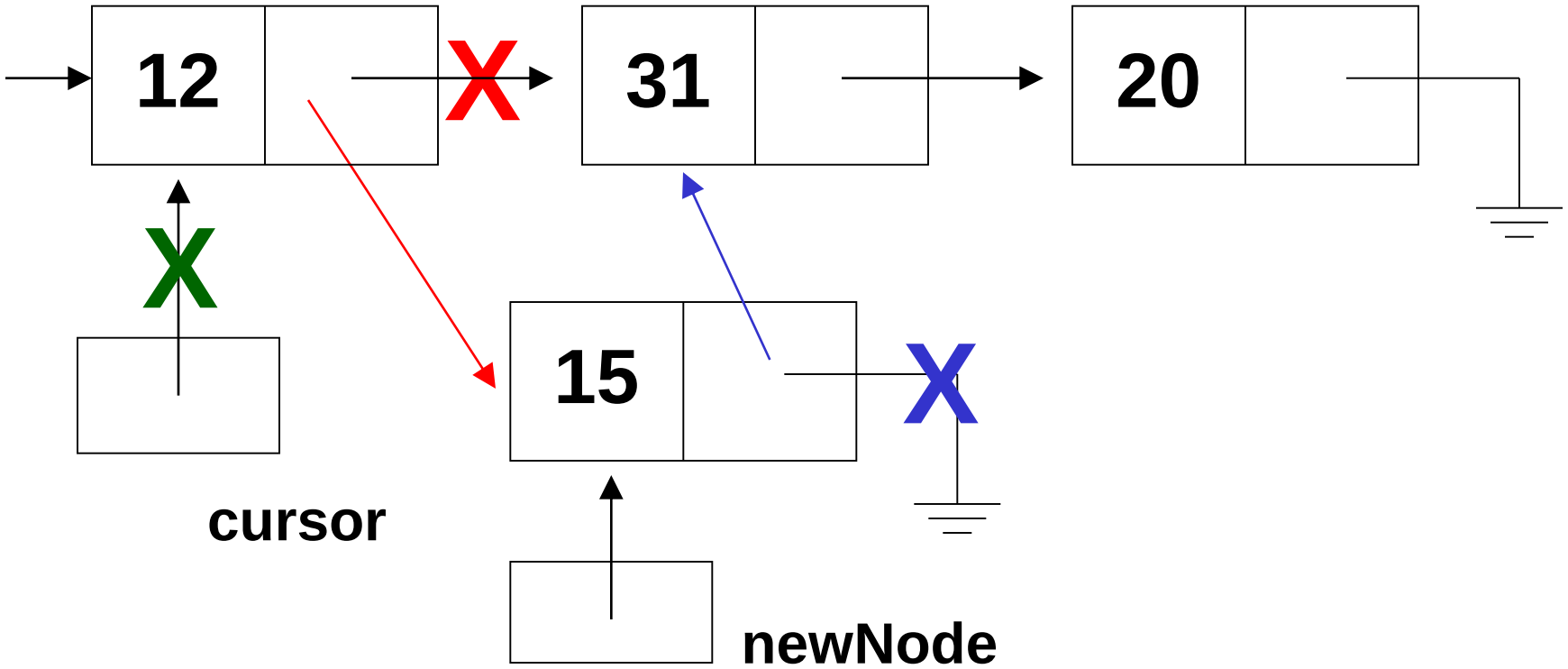
Add after cursor (general)

```
newNode.setLink(cursor.getLink());
cursor.setLink(newNode);
```



Add after cursor (general)

```
newNode.setLink(cursor.getLink());
cursor.setLink(newNode);
cursor = newNode;
```



Add integer after cursor

```
public void addIntAfter(int element)
{
     IntNode newNode =
          new IntNode(element);
     if (cursor == null)
     {
          head = newNode;
          tail = newNode;
          cursor = newNode;
     }
```

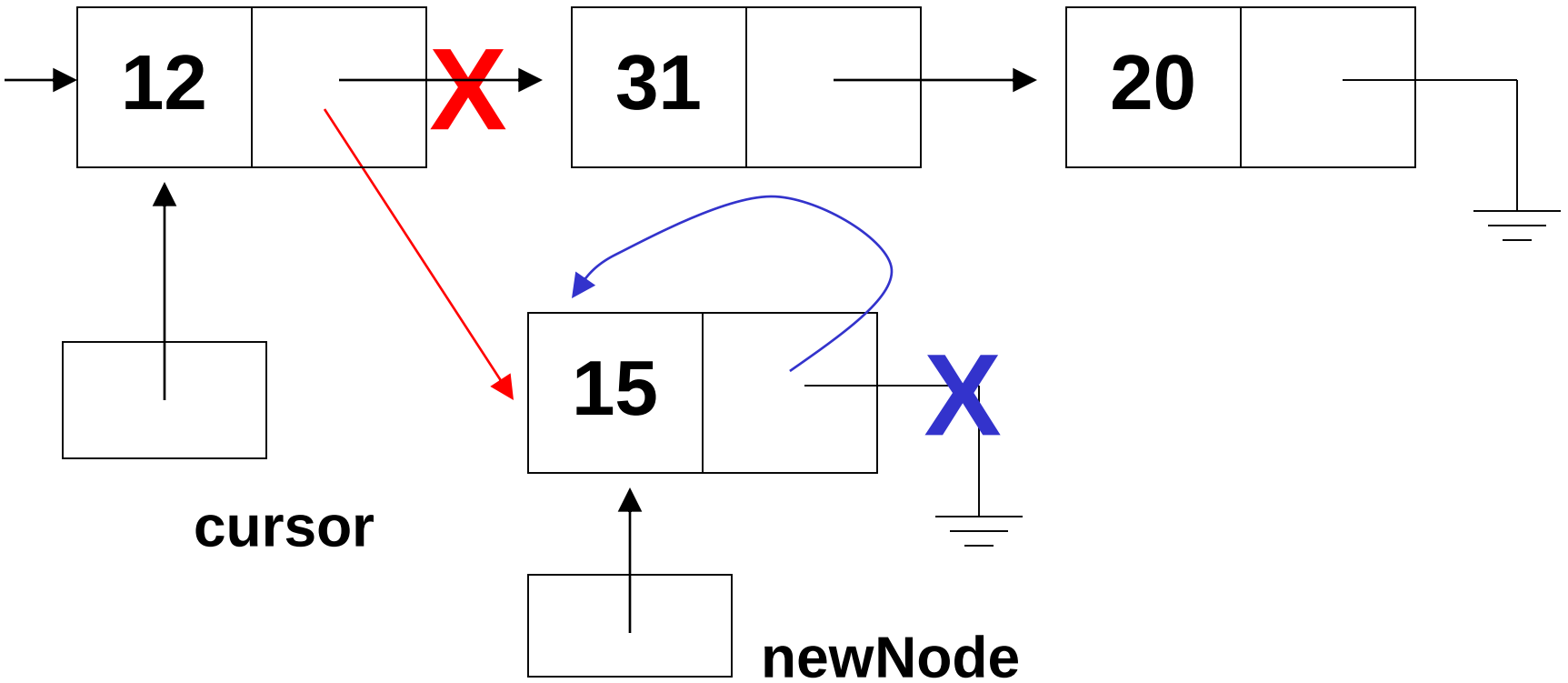
Add integer after cursor (cont'd)

```
  else {
      newNode.setLink
           (cursor.getLink());
      cursor.setLink(newNode);
      cursor = newNode;//advance cursor
      if (cursor.getLink() == null)
           tail = cursor;
    }
}
```

Watch out!

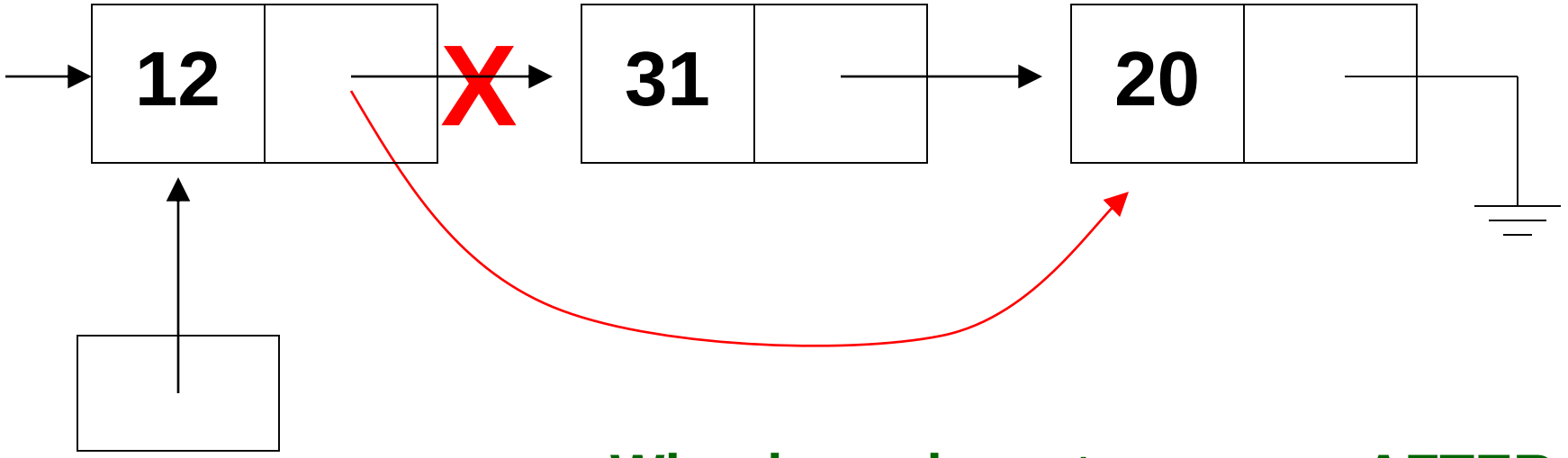
cursor.setLink(newNode);

newNode.setLink(cursor.getLink());



Remove after cursor (general)

cursor.setLink(cursor.getLink().getLink());



Why do we have to remove AFTER the cursor? Can't we just remove the cursor node ?

Remove after cursor

```
public void removeIntAfter()
{
     if (cursor != tail) {
          cursor.setLink(
               cursor.getLink()
             .getLink());
          if (cursor.getLink() == null)
               tail = cursor;//last node
     }
}
```

Remove head of list

```
public void removeHead()
{
     if (head != null)
          head = head.getLink();
     if (head == null)
          tail = null;
     cursor = head;
}
```

Working with cursor

```
public boolean advanceCursor() {
     if (cursor != tail) {
          cursor = cursor.getLink();
          return true;
     }
     else
          return false;
}
```

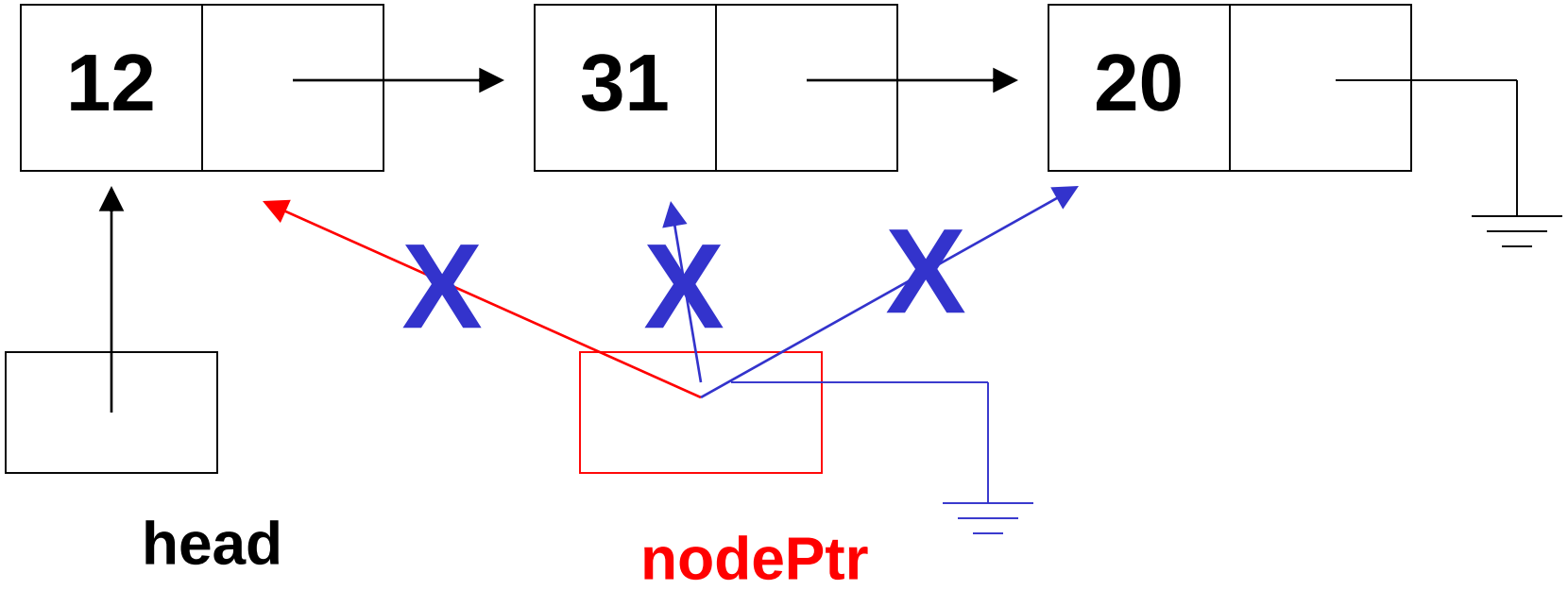
Working with cursor (cont'd)

```
public void resetCursor() {
     cursor = head;
}
public boolean isEmpty() {
     return (cursor == null);
}
```

“Traverse” a list

nodePtr = head;

nodePtr = nodePtr.getLink();



Length of List

```
public int listLength()
{
     IntNode nodePtr = head;
     int answer = 0;
     while (nodePtr != null) {
          answer++;
          nodePtr = nodePtr.getLink();
     }
     return answer;
}
```

Point of Caution

- **Why didn't we define nodePtr this way in listLength?**

```
IntNode nodePtr = new IntNode(0);
nodePtr = head;
```

- **Use "new" only when you actually need a new node!**
- **If you are defining a variable to reference a node that already exists, don't use "new"!**

Search the list

```
public boolean listSearch(int target) {
     IntNode nodePtr = head;
     while (nodePtr != null) {
       if (target == nodePtr.getData()) {
          cursor = nodePtr;
          return true;
       }
       nodePtr = nodePtr.getLink();
     }
     return false;
}
```

Set cursor to specific position

```
public boolean listPosition(int position)
  {
     IntNode nodePtr = head;
     int i = 1;
     if (position <= 0) throw ...etc...
     while (i<position && nodePtr != null){
          nodePtr = nodePtr.getLink();
          i++;
     }
     if (nodePtr != null) cursor = nodePtr;
     return (nodePtr != null);
  }
```

Copy a list

```
public static IntList listCopy
  (IntList source) {
     IntList newList = new IntList();
     IntNode nodePtr = source.head;
     while (nodePtr != null) {
       newList.addIntAfter
          (nodePtr.getData());
       nodePtr = nodePtr.getLink();
      }
     return newList;
}
```

Additional IntList Methods

```
public int getNodeData() throws
  EmptyListException {
  if (cursor == null)
     throw new EmptyListException(…);
  return (cursor.getData());
}
public void setNodeData(int element)
  throws EmptyListException {
  if (cursor == null)
     throw new EmptyListException(...);
  cursor.setData(element);
}
```

The Bag ADT using Lists

```
public class IntLinkedBag
  implements Cloneable
{
    private IntList data;
    private int manyItems;
```

Implementation (cont'd)

```
public IntLinkedBag() {
  manyItems = 0;
  data = new IntList();
}
```

No need to worry about the bag's capacity since a linked list has no maximum capacity! Only one constructor is needed.

Implementation (cont'd)

```
public int getCapacity() {
  return Integer.MAX_VALUE;
}

public int size() {
  return manyItems;
}
```

Implementation (cont'd)

```
public void ensureCapacity
  (int minimumCapacity) {
  // no work is needed
}
```

Implementation (cont'd)

```
public void add(int element) {
  data.addNewHead(element);
  manyItems++;
}
```

Order of Complexity?

Implementation (cont'd)

```
public int countOccurrences(int target) {
  int answer = 0;
  int index;
  data.resetCursor();
  for (index=0; index<manyItems; index++)
  {
     if (target == data.getNodeData())
         answer++;
     data.advanceCursor();
   }
  return answer;
}
```

Order of Complexity?

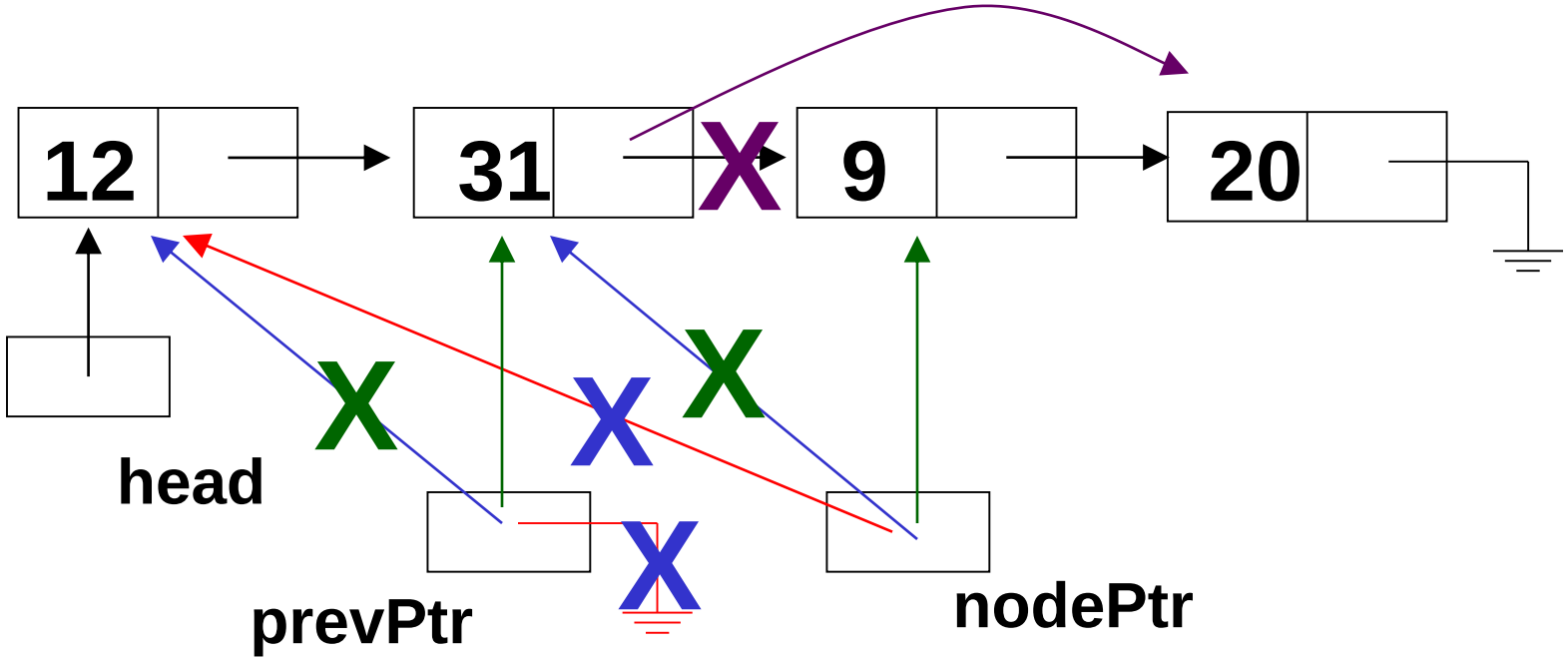
Removing an element

- **Other methods can be implemented in a similar manner.**
- **Watch out!**
  `remove` isn't easy with singly linked lists!
- **How can we remove an element with the structure we have?**
- **Use listSearch to move cursor to location of target**
- **BUT we can't remove that node!**

Trailing pointers

- **Use a trailing pointer to keep track of the previous node to the current node we're examining.**
- **Once we find the node we want to remove, the trailing pointer will point to the previous node which we can connect to the next node after the current node.**

Example (remove 9)



Step 1:
nodePtr = head
prevPtr = null

Step 2 :
prevPtr = nodePtr
nodePtr = nodePtr.getLink()

Step 2 (again)

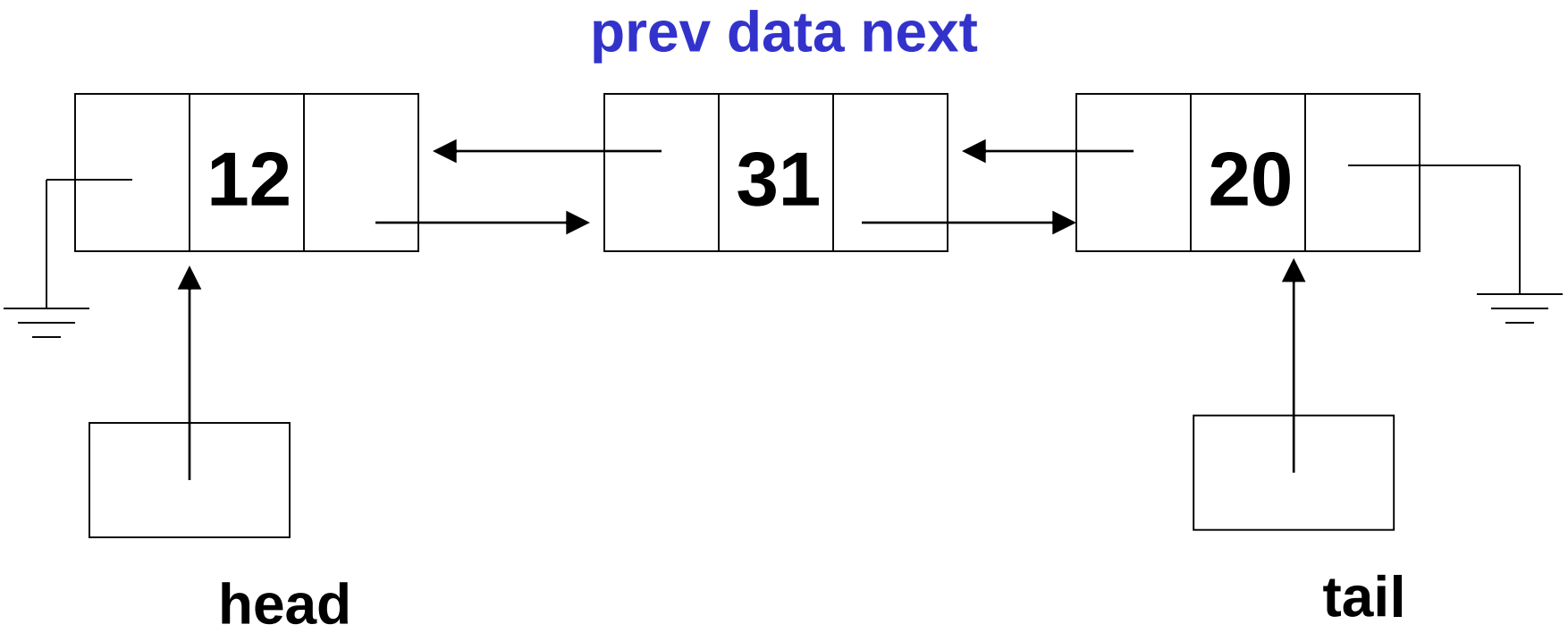
Step 3: prevPtr.setLink(nodePtr.getLink())

Implementation

```
public boolean remove(int target) {
  IntNode nodePtr = head;
  IntNode prevPtr = null;
  while (nodePtr != null &&
     nodePtr.getData() != target) {
          prevPtr = nodePtr;
          nodePtr = nodePtr.getLink();
  }
  if (nodePtr != null)
     prevPtr.setLink(nodePtr.getLink());
  return (nodePtr != null);
} Order of Complexity?
```

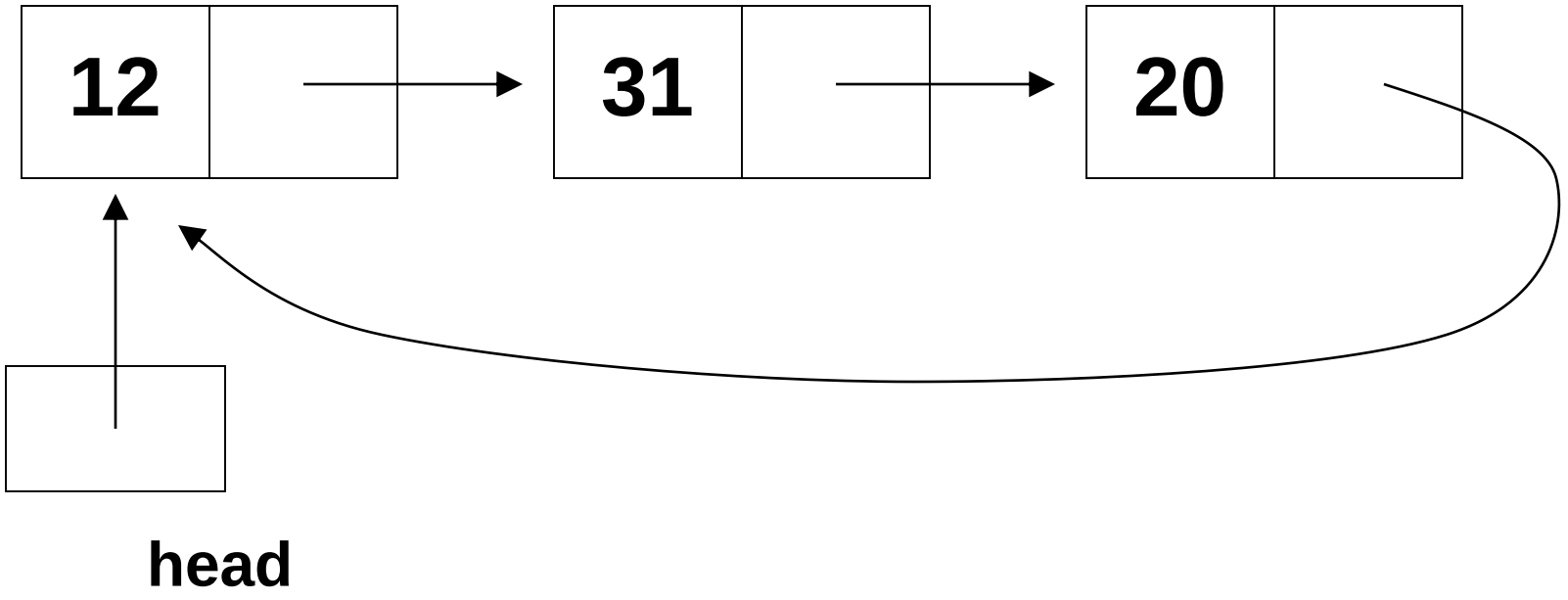
Linked List Variations

- **Doubly-linked list**



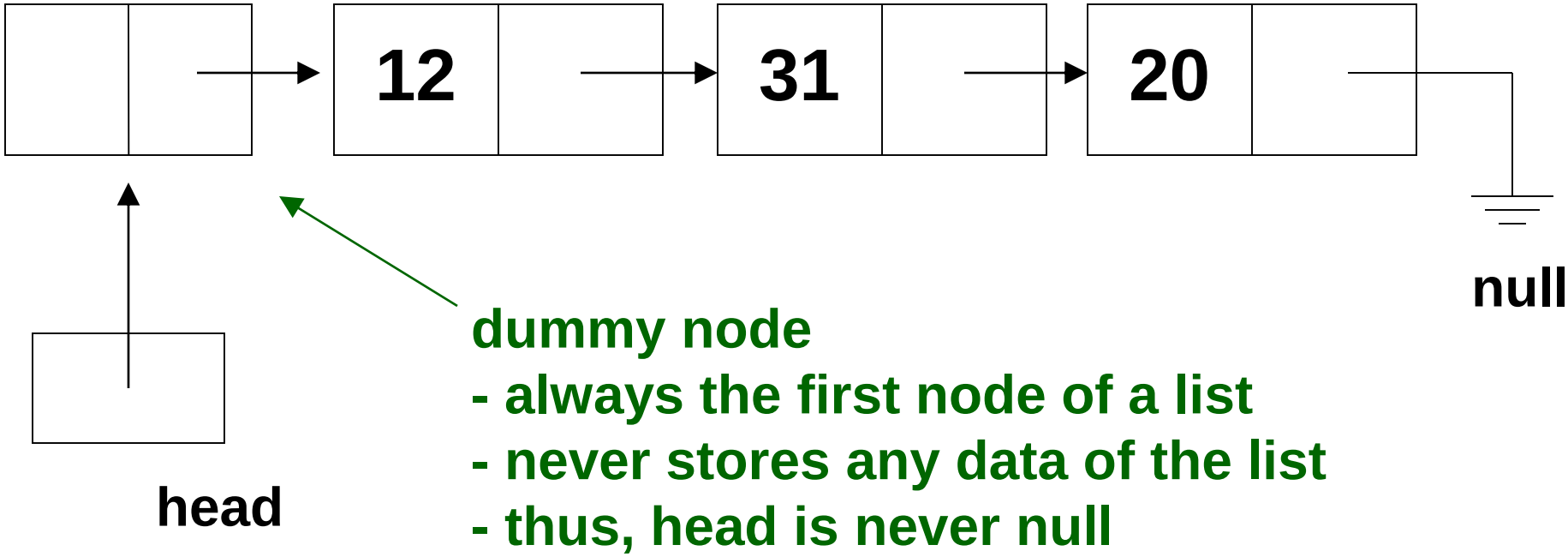
Linked List Variations

- **Circular-linked list**



Linked List Variations

- **Linked list with dummy head node**



Arrays vs. Linked Lists

Arrays

- **Better for random access to any data value**
- **Better if number of elements is known and doesn’t vary much**

Linked Lists

- **Better for additions and removals (other data elements do not need to be moved)**
- **Better if number of elements varies greatly and is not known at runtime**

The Object Data Type

- **A variable of type Object is capable of holding a reference to any kind of object.**
- **Let ObjectB be a subclass of ObjectA.**

```
ObjectA a;
ObjectB b;
```

- `a = b;`

  OK, widening conversion is automatic

- `b = a;`

  NO, narrowing conversion is not automatic

```
b = (ObjectB) a;
```

Wrapper Classes

- **Primitive data types (not objects):**
  `byte  short  int  long`
  `float  double  char  boolean`
- **Wrapper classes are object classes:**
  `Byte  Short  Integer  Long`
  `Float  Double  Character  Boolean`
- **To perform arithmetic on data in a wrapper class, you need to extract the data first**
  example:

```
Integer intObject =
        new Integer(214);
int i = intObject.intValue();
```

Generic Node

```
public class Node
{
    private Object data;
    private Node link;

    // Node methods
}
```

Constructor

```
public Node(Object initialData)
{
    data = initialData;
    link = null;
}
```

Using the constructor:

```
Integer intObject = new Integer(214);
Node newNode = new Node(intObject);
```

getData()

```
public Object getData()
{
    return data;
}
```

Using the accessor:

```
Integer I =
  (Integer)newNode.getData();
System.out.println(I.intValue());
```

setData()

```
public void setData(Object newData)
{
    data = newData;
}
```

Using the mutator:

```
Integer J = new Integer(220);
newNode.setData(J);
```

A generic linked list

```
public class List
{
    private Node head;
    private Node tail;
    private Node cursor;

    // List methods
}
```

A sample method

```
public void addNewHead(Object element)
{
     Node newNode =
          new Node(element);
     newNode.setLink(head);
     head = newNode;
     if (tail == null) tail = head;
     cursor = head;
}
```

Another method

```
public boolean listSearch(Object target) {
  Node nodePtr = head;
  while (nodePtr != null) {
     if (target.equals(nodePtr.getData()))
     {
          cursor = nodePtr;
          return true;
      }
      nodePtr = nodePtr.getLink();
  }
  return false;
}
```

CAREFUL! target could be null (code not shown here)

Using a generic data structure

- **To store data in the list, put it in a wrapper (if necessary) before you insert it into the list.**
- **If you extract data from the list, remove it from the wrapper (if necessary) before processing it.**
- **REMEMBER: If you extract data from the list, the accessor will return an `Object` which has to be typecast (narrowed).**

Iterators (optional)

`public class List implements Iterator`

If a class implements the Iterator interface, it must provide the following methods:

`public boolean hasNext()`

`public Object next()`

`public void remove()`

- **Using a loop, iterators allow you to step through a collection, like a list, just as an index allows you to step through an array.**