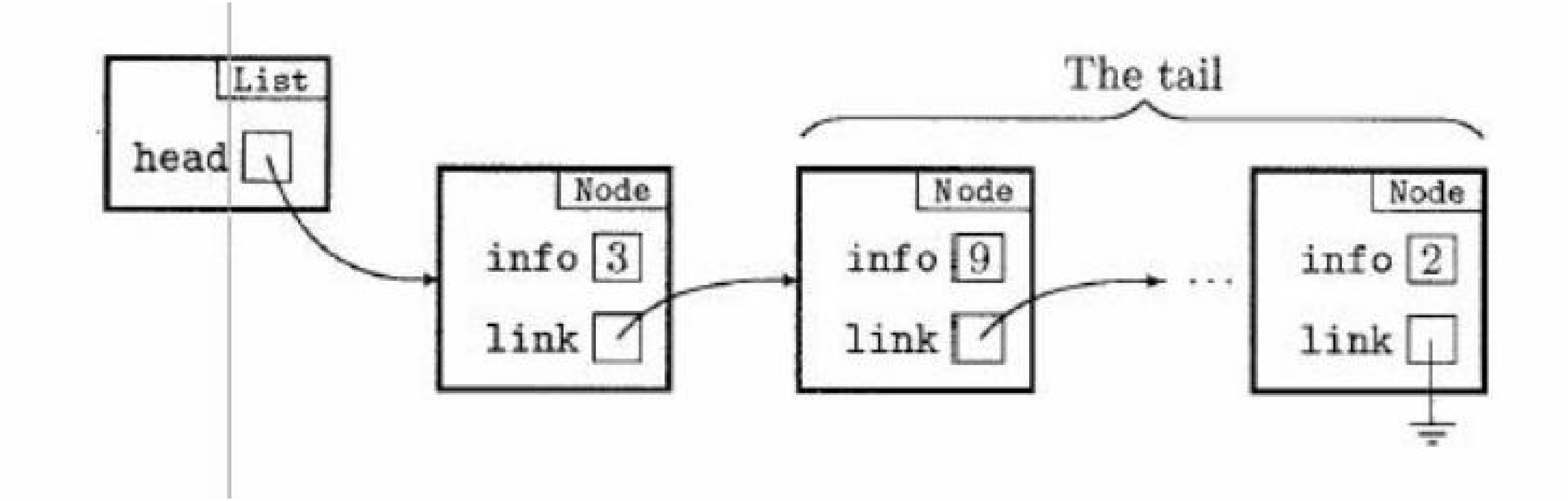
Recursive List Processing

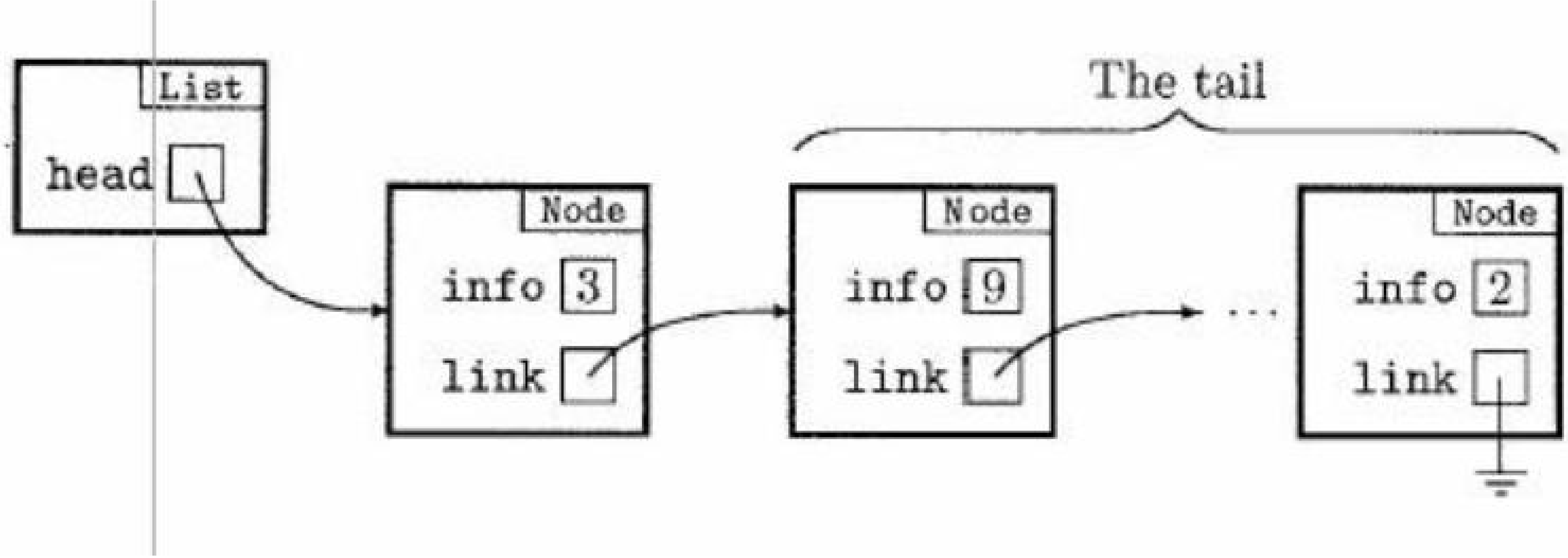
Recall:

- Every recursive process has 2 parts
  - A case in which the process is defined in terms of a simpler version of itself
  - A simple case that terminates the recursion

Linked Lists

- Think of a linked list as a
    - structure that is empty (the simple case) or,
    - Consists of a node followed by a link list (the tail of the original linked list)

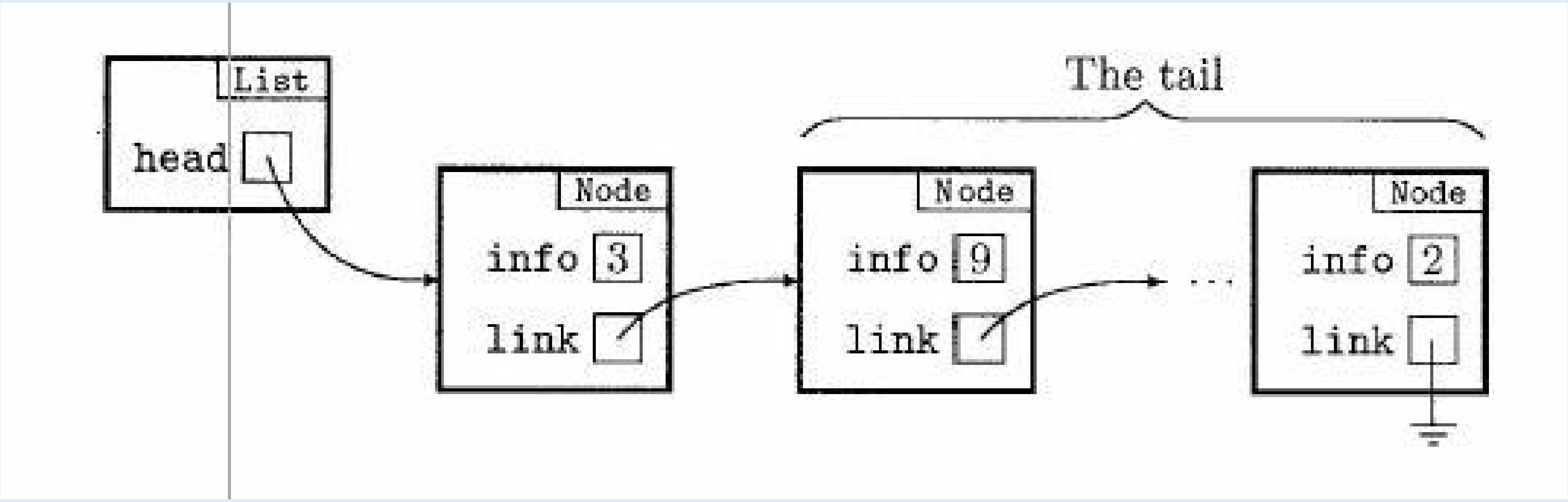




- Recursive algorithms for list processing reflect this structure by
  - processing the node referred to by the head directly and
  - processing the tail recursively with an empty list acting as a stopping condition.

Example 1 - Algorithm

```
To PRINT THE LIST
   if the list is not empty
      print the info field of the first node
      PRINT THE LIST that forms the tail
```



Example 1 - Code

```
public void printList (){
    // a first attempt
    if (head != null){
        System.out.println(head.info);
            head.link.printList();
    }
}
```

- Problem: since printList is an instance method of the class List, it must be called with an implicit List object.

- `head.link.printList();` attempts to make a call to `printList` with the object `head.link`, which is of type `Node`

Solution:

- In the List class, we use a non-recursive method that checks to see if the list is not empty
  - If true, the method then calls the recursive helper version in the Node class
  - That method is guaranteed to have a non-empty list

Example 2 – Algorithm

```
To PRINT A NON-EMPTYLIST
   print the info field of the first node
   if the tail is not empty
      PRINT THE NON-EMPTYLIST that forms the tail
```

Recall: Linked Lists

```
public class List {
  private Node head;

  class Node {
    int info;
    Node link;

    Node (int i, Node n){
      info = i;
      link = n;
    }
  }
```

Example 2 - Code

```
class List
{
   private Node head;

   public void printList (){
      if (head != null)
         head.printList();
   }

   class Node {// an inner class
      int info;
      Node link;

      Node (int i, Node n){
         info = i;
         link = n;
      }

      void printList (){
         System.out.println(info);
         if (link != null)
            link.printList();
      }

   }

}
```

Example 3 – insert a new node in a list in ascending order

```
To INSERT IN THE LIST
  if the list is empty or item < first node's info field
    insert a node containing item here
  else
    INSERT IN THE LIST referred to by the first node
```

- Once again, we will not use the algorithm in this simple form.

- Instead, we will create two versions of the insertion method –
  - one for the List class
  - one for the inner Node class.

- This non-recursive method, for the List class, will start the process of insertion of a node in an ordered list.
- The method calls the recursive version only if the new item should be inserted somewhere *after* the first node; otherwise, it does the insertion itself.

```
public void insert (int item)
{
    if (head == null || item < head.info)
        // insert item as first node of original list
        head = new Node(item,head);
    else
        // call recursive version to
        // insert item in tail of non-empty list
        head.insert(item);
}
```

- This recursive helper method, for the Node class, will insert item in its correct position in the tail of a non-empty ordered list.
- It should never be called if the new item is to be the first node in the list.

```
void insert (int item)
{
    if (link == null || item < link.info)
        // insert item right after first node
        link = new Node(item,link);
    else
        // insert after first node of non-empty tail
        link.insert(item);
}
```