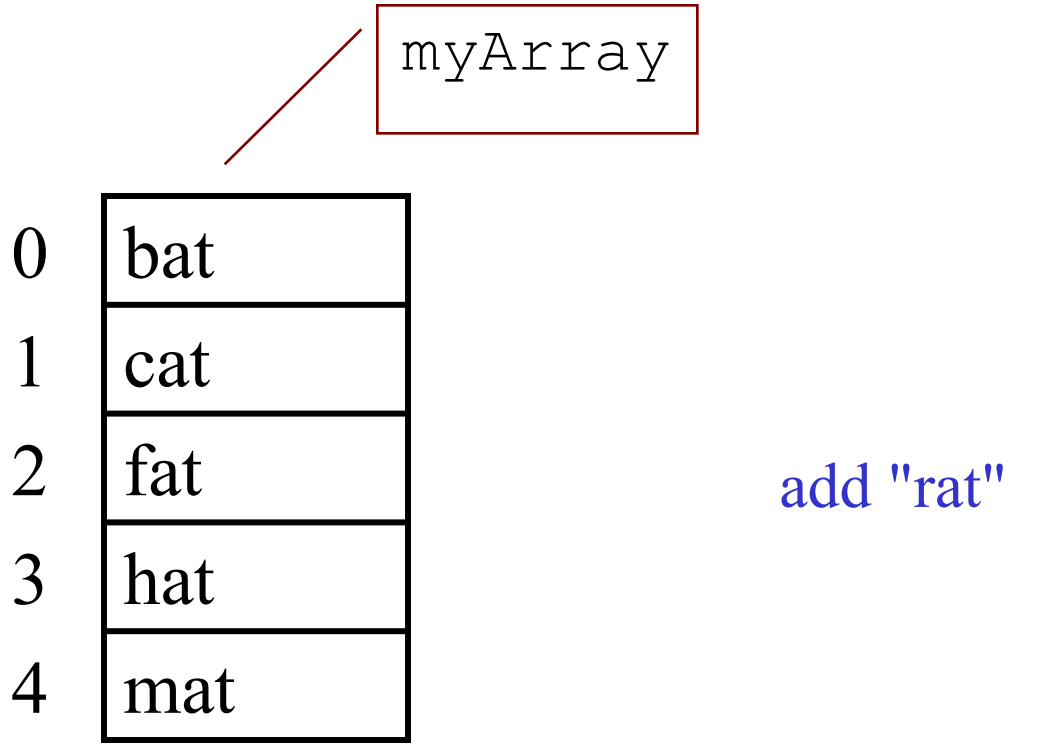
Dynamic vs. Static Structures

The Advantages of Linked Lists

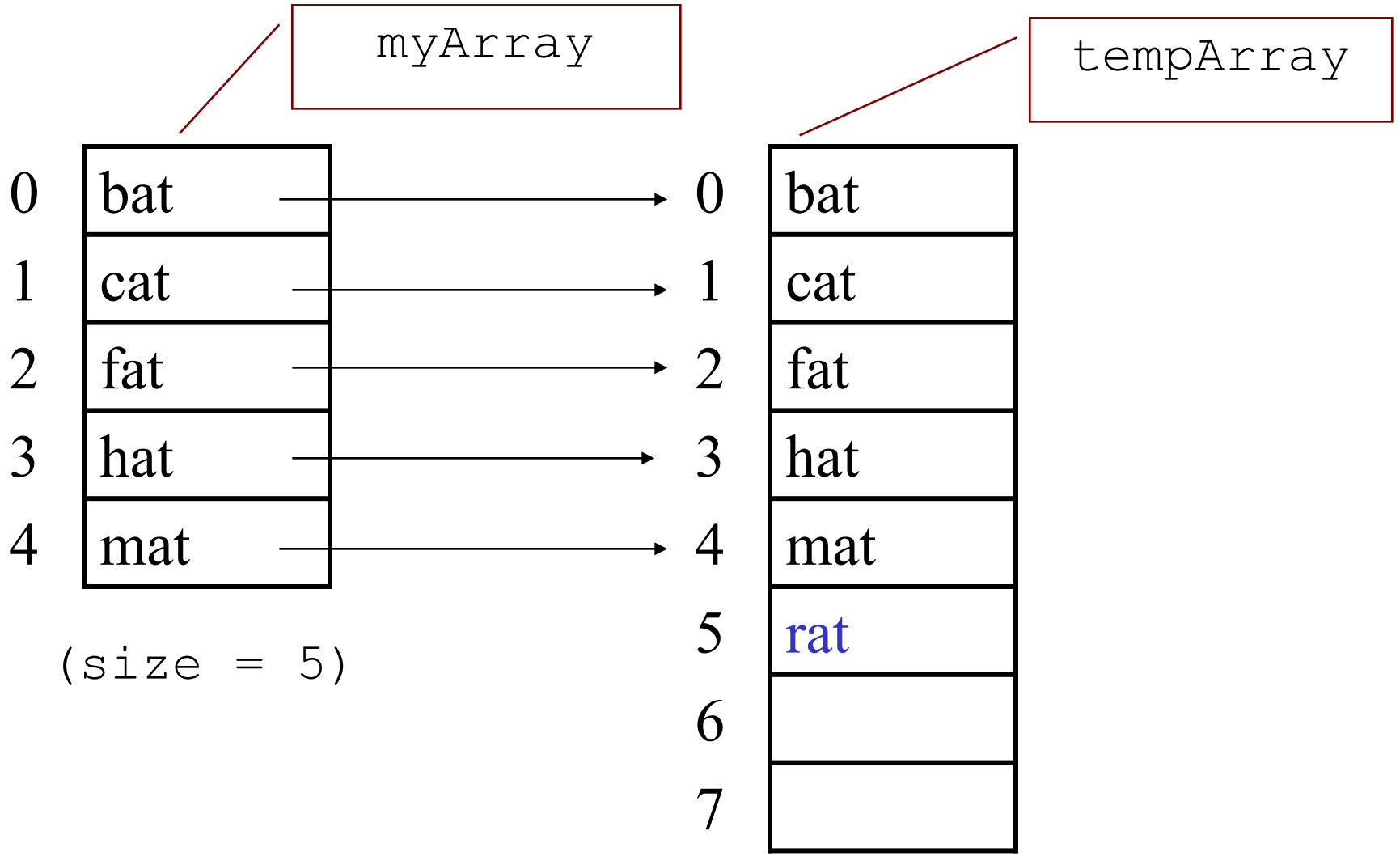
Static Storage Structures

- Once declared, they are fixed in size
- `String [] SSNs = new String[10]`
- The size may be a "variable"
- `String [] SSNs = new String[n]`
- Or a constant
- `String [] SSNs = new String[MAX_SSNS]`
- But once declared, the size is fixed

Once the array is filled, there is no more room

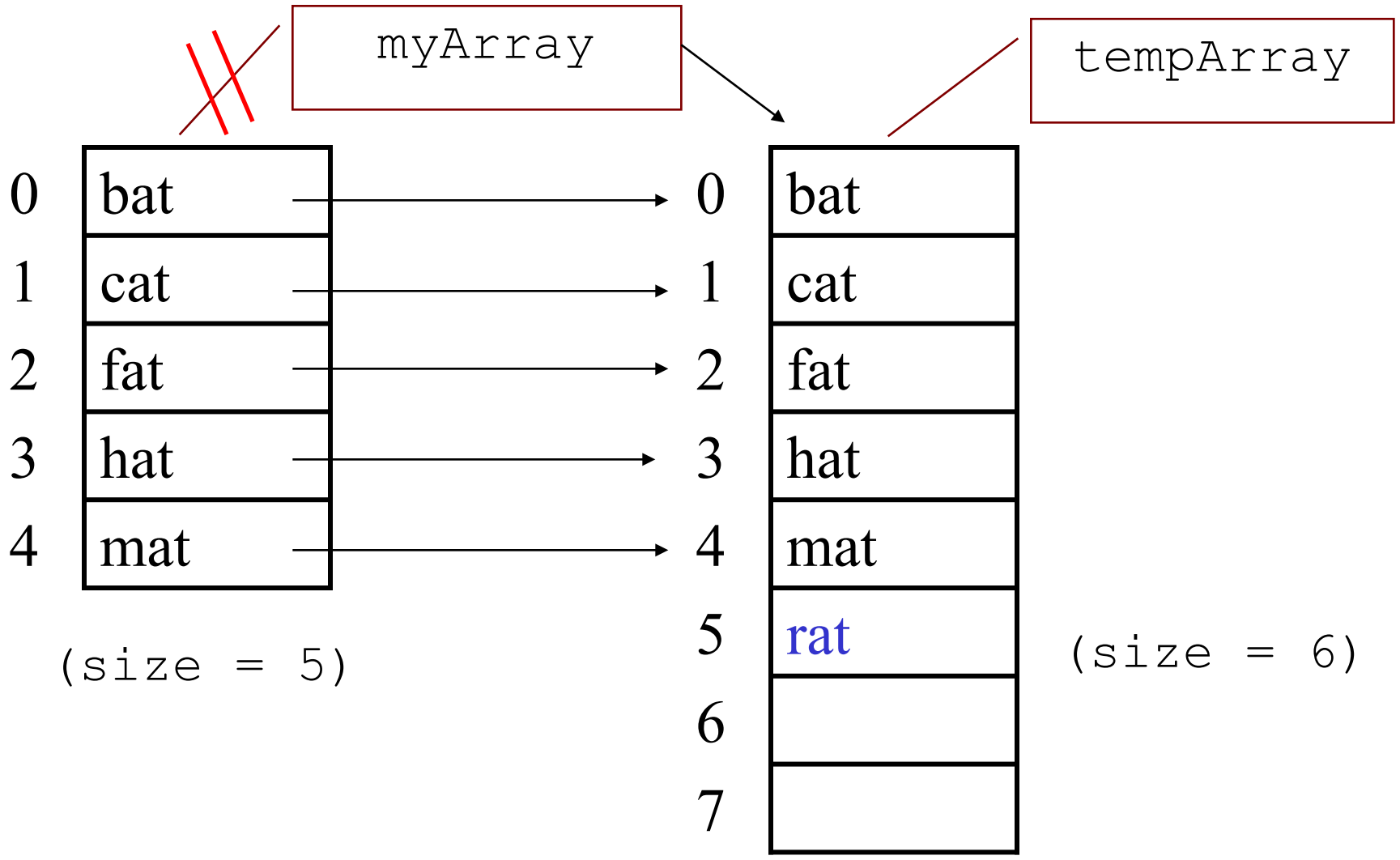


One solution is to create an array that is bigger, copy all the elements, and add the new item



```
tempArray = new String[8];

for (int i=0; i< size; i++)

   tempArray[i] = myArray[i];

tempArray[size++] = "rat";

myArray = tempArray
```

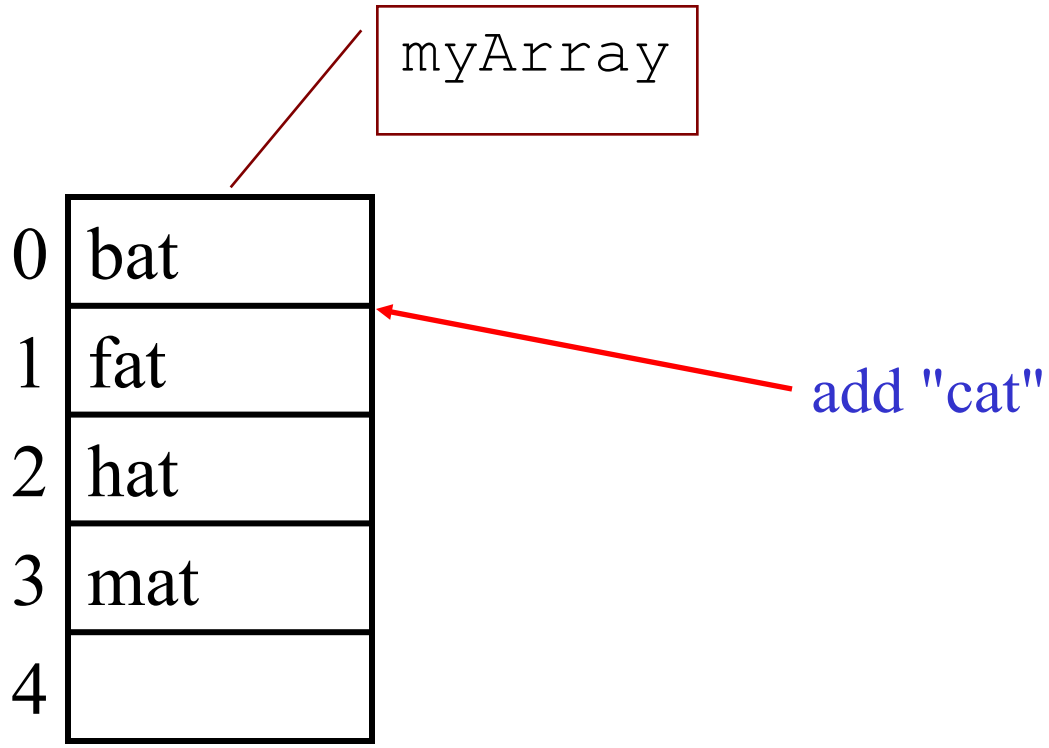
myArray = tempArray



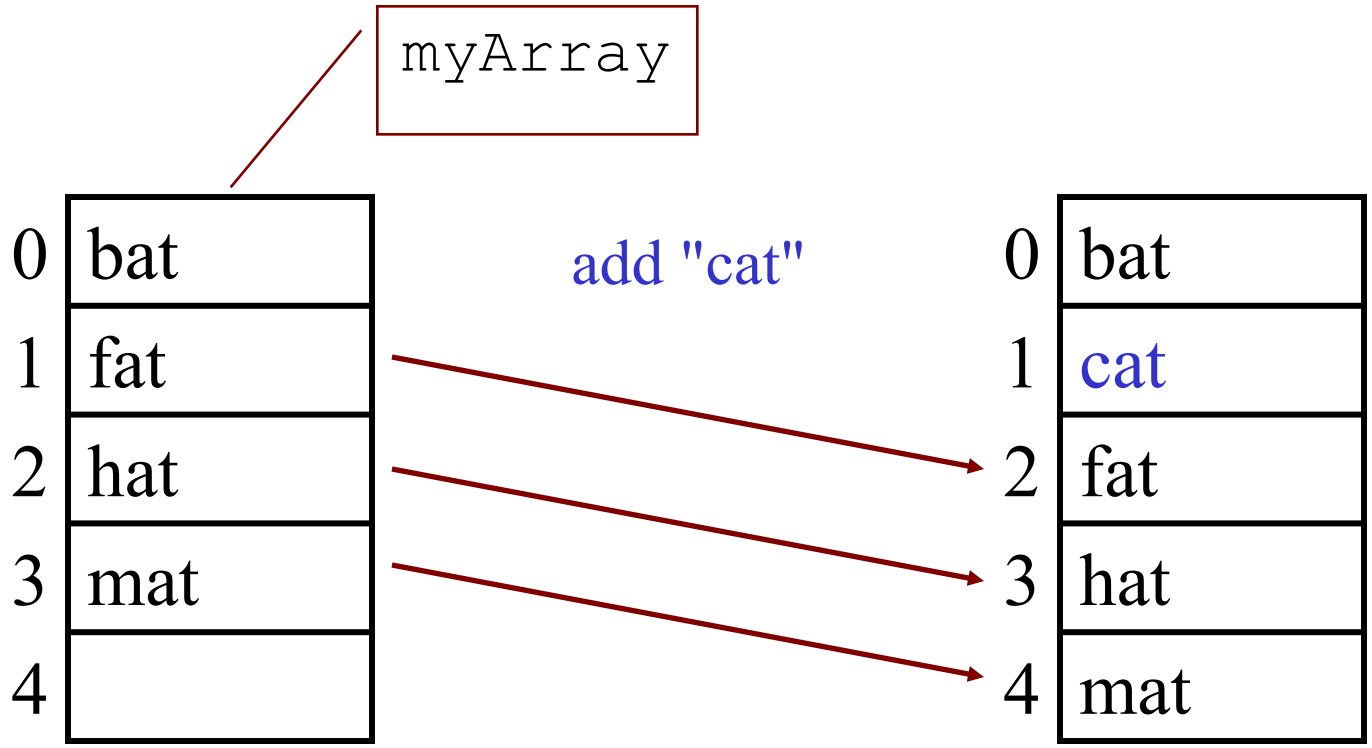
```
tempArray = new String[8];
for (int i=0; i< size; i++)
    tempArray[i] = myArray[i];
tempArray[size++] = "rat";
myArray = tempArray
```

- This loop costs a lot of operations (copies) therefore it costs a lot of time.
- If the array has 100,000 items in it, it will take 100,000 assignments to copy the array!

Another problem: adding items to an ordered (sorted) array



Another problem: adding items to an ordered (sorted) array



```
// add "cat" after position n

for (int i=size; i > n; i--)

   myArray[i+1] = myArray[i];

tempArray[n+1] = "cat";
```

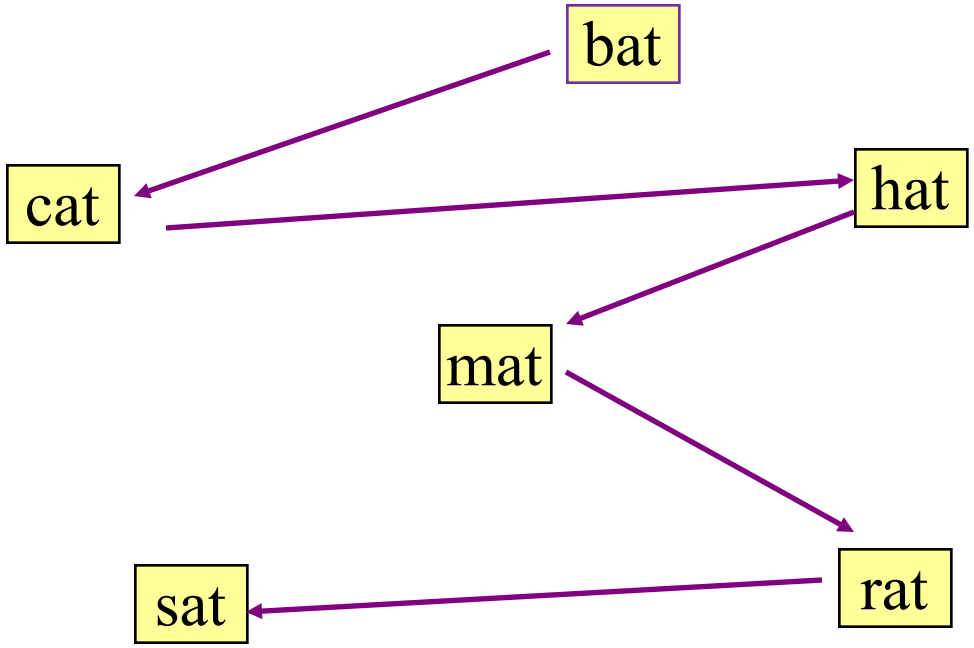
```
// add "cat" after position n
for (int i=size; i > n; i--)
   myArray[i+1] = myArray[i];
tempArray[n+1] = "cat";
```

Again, there is a loop, which will cost time.

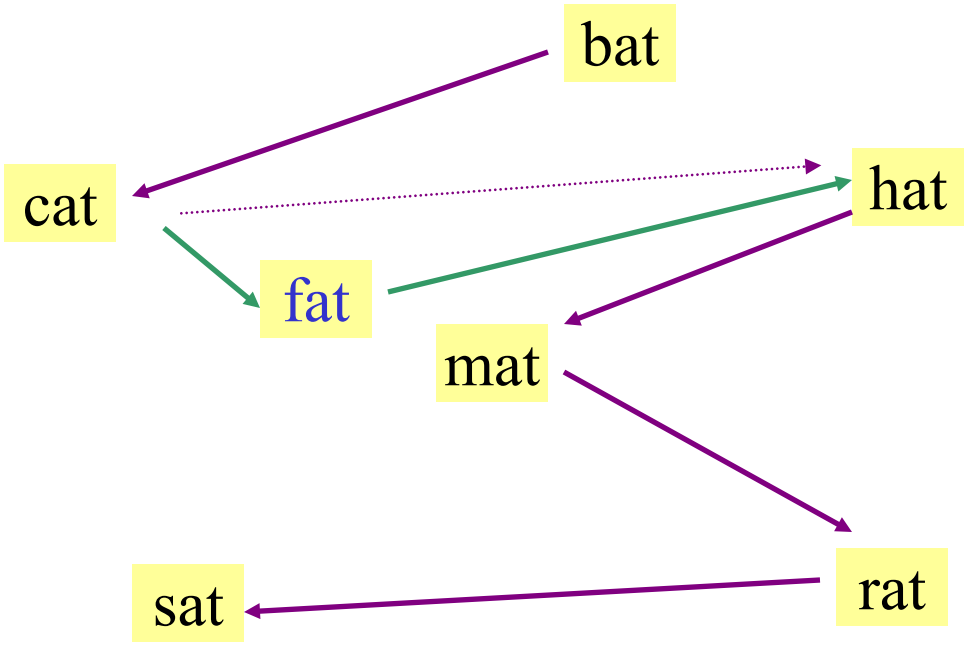
A solution is *dynamic* structures

- Dynamic structures use only as much memory as they need.
- Dynamic structures rely on themselves to determine the order of the data items they contain.
- Dynamic structures are not stored in contiguous memory

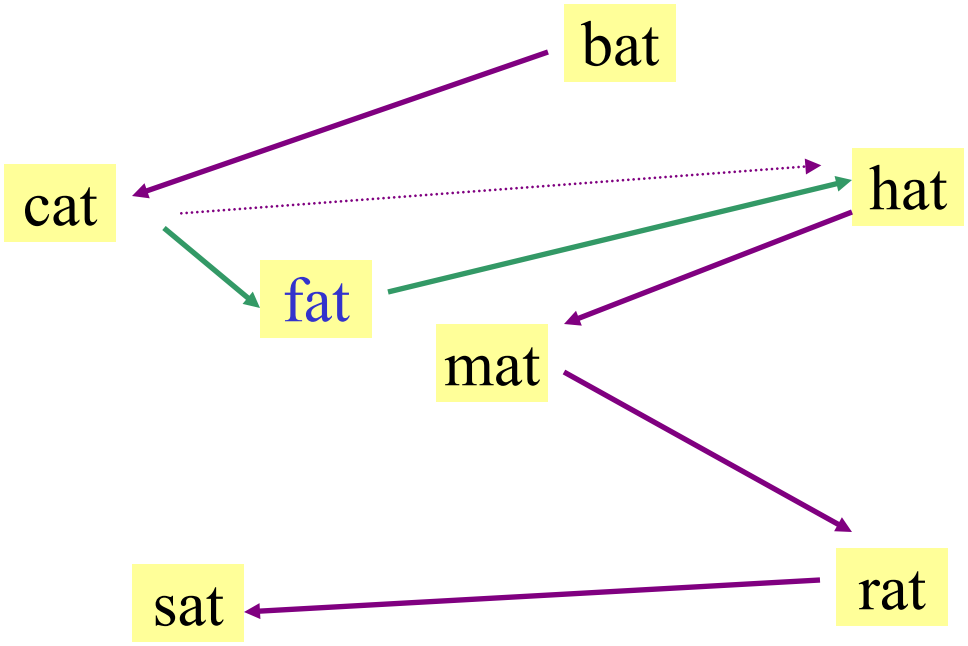
An *abstract* dynamic structure



add "fat" after "cat"



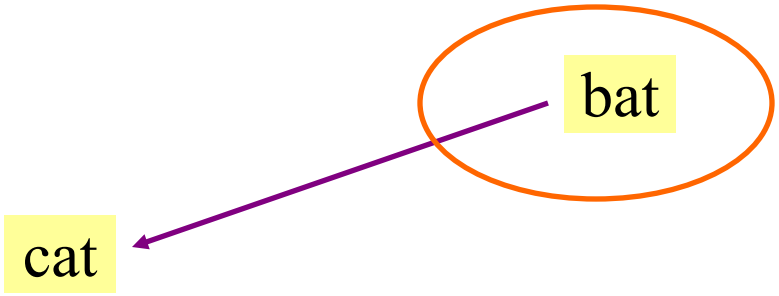
add "fat" after "cat"




```
make "fat" point to what "cat" was
     pointing to;
make "cat" point to "fat"
```

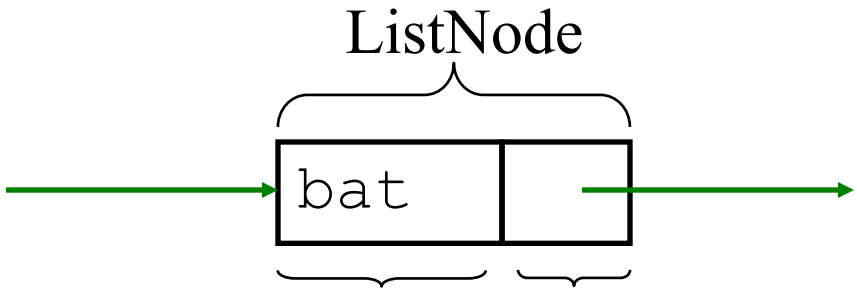
No loop!

Create a real class for a "node" in the list



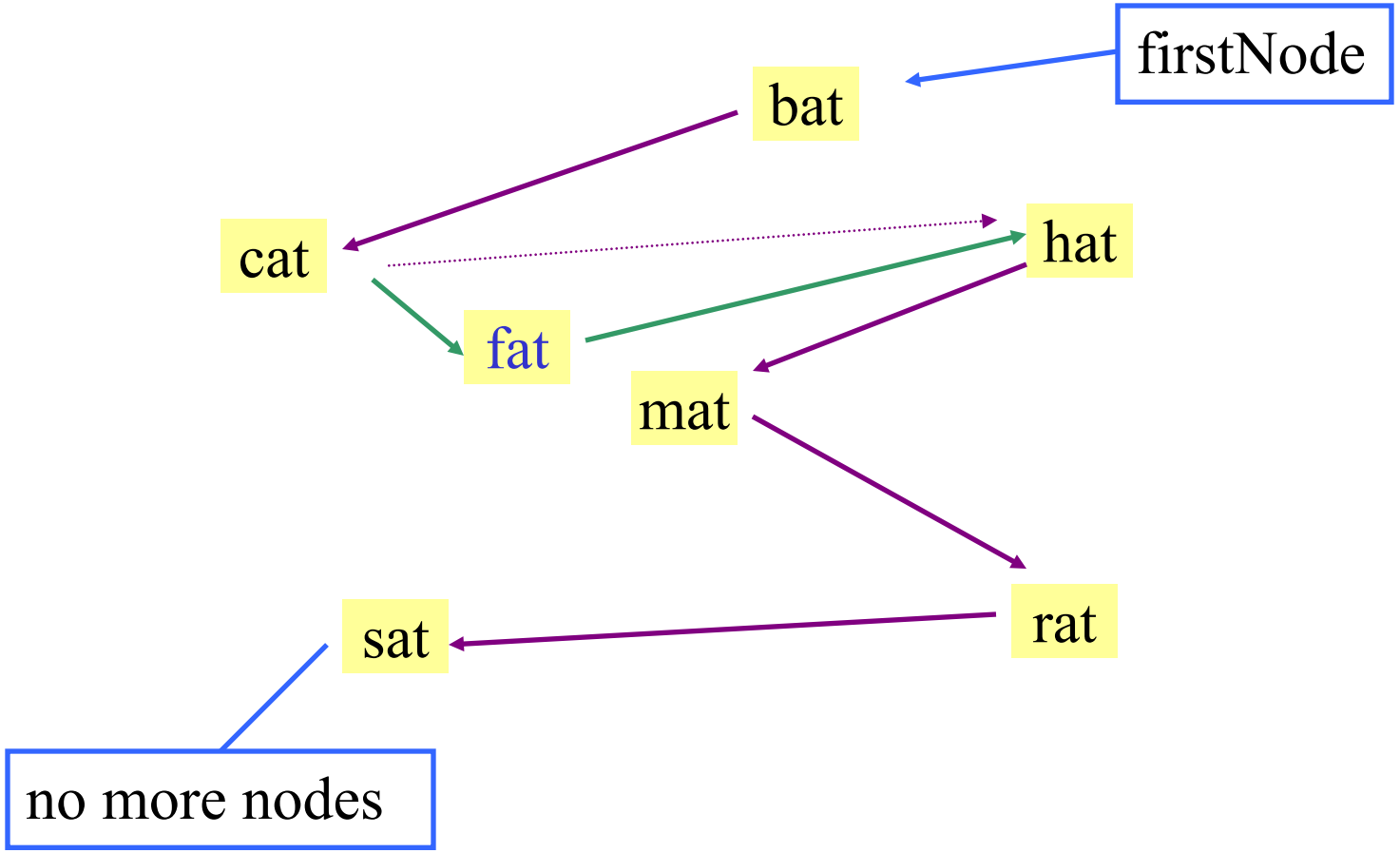
- The node containing "bat" has two components:
- The actual *data* of the node ("bat")
- A *reference* linking it to the *next* node in the list


```
class ListNode  {
   String data;
   ListNode next;
   public ListNode(String d, ListNode ln)  {
      data = d;
      next = ln;
   }  // constructor
}  // class ListNode
```

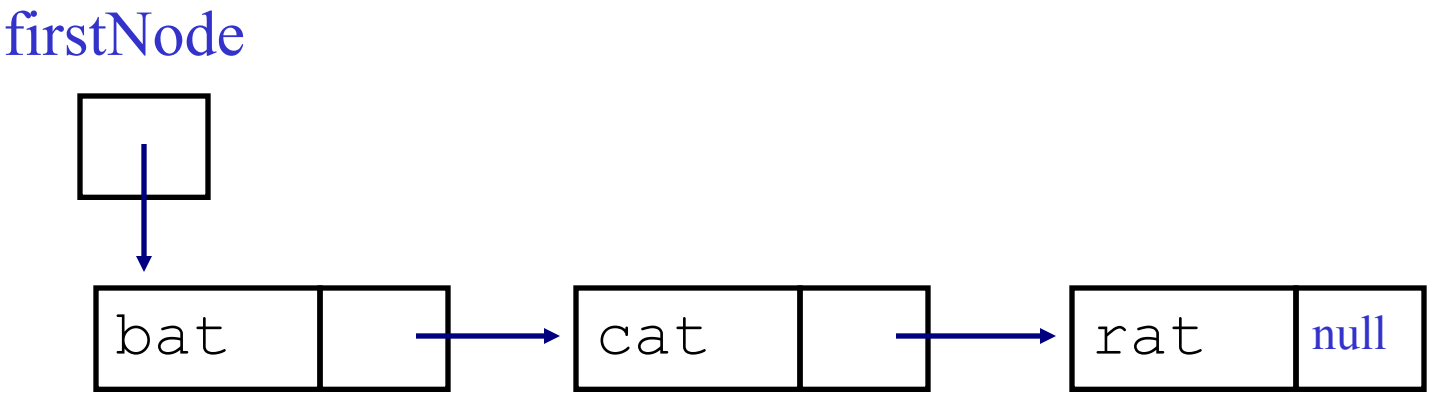


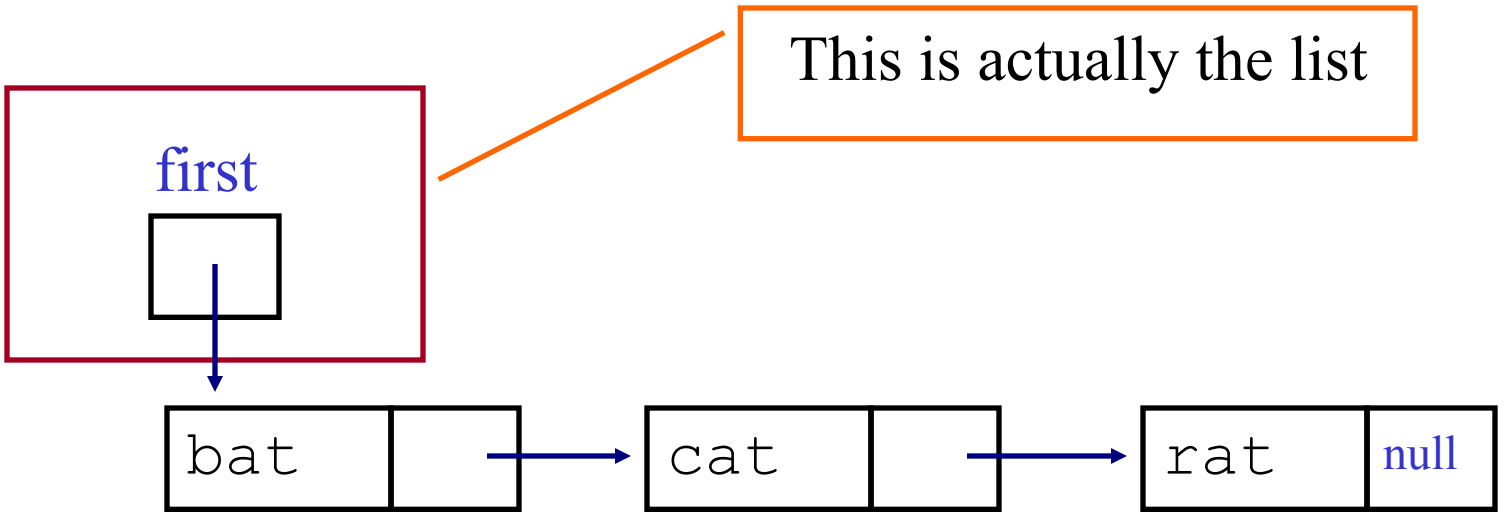

```
class ListNode  {
    String data;
    ListNode next;
    public ListNode(String d, ListNode ln)  {
        data = d;
        next = ln;
    }  // constructor
}  // class ListNode
```

To complete the list structure we need two more things...



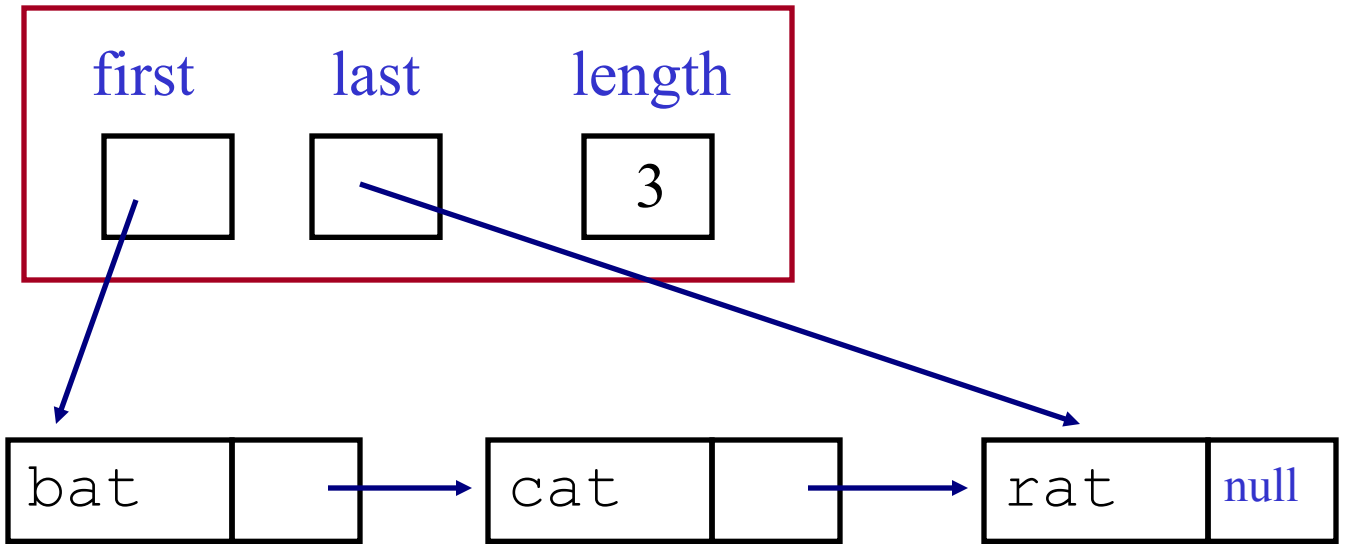
Now we have a complete list:



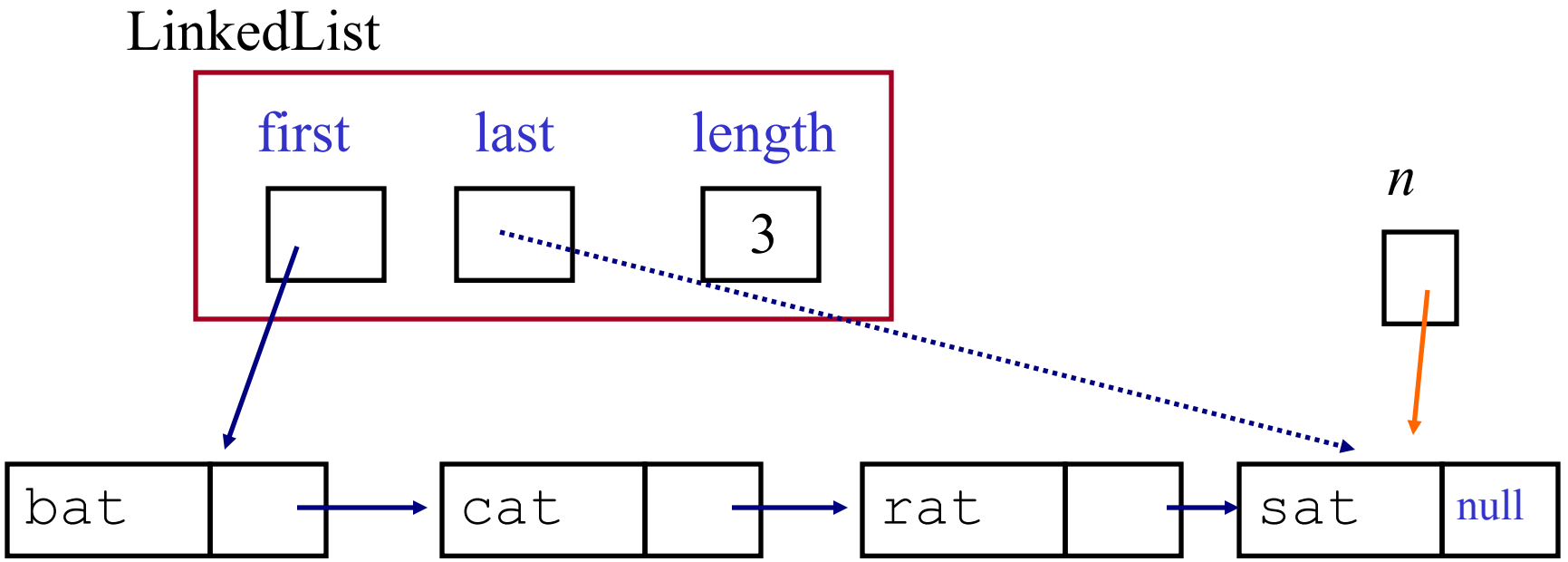



```
public class LinkedList  {

   private ListNode first;
```

LinkedList

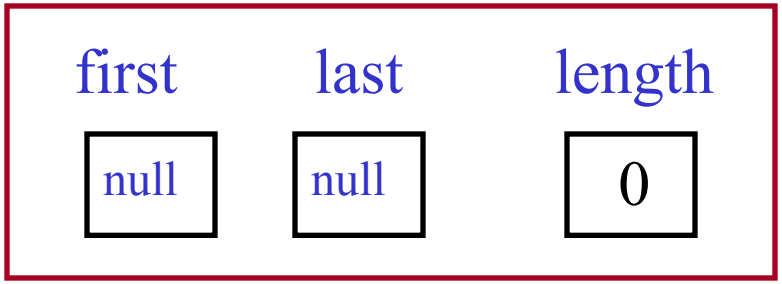



```
public class LinkedList  {

   private ListNode first;

   private ListNode last;

   private int length;
```




```
public void append (String s) {
   ListNode n = new ListNode(s);
   last.next = n;
   last = n;
   length++;
}
```

Back to the beginning... what does a *new* LinkedList look like?

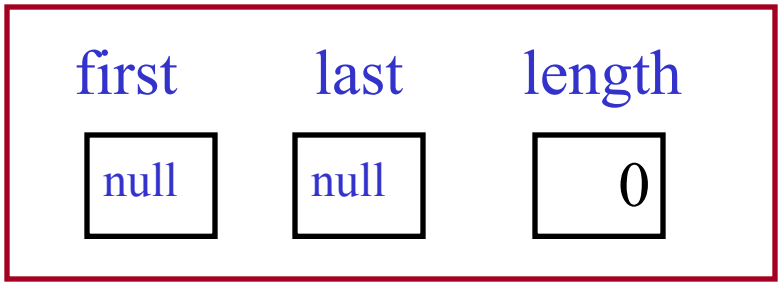



```
public class LinkedList  {

   private ListNode first = null;

   private ListNode last = null;

   private int length = 0;
```

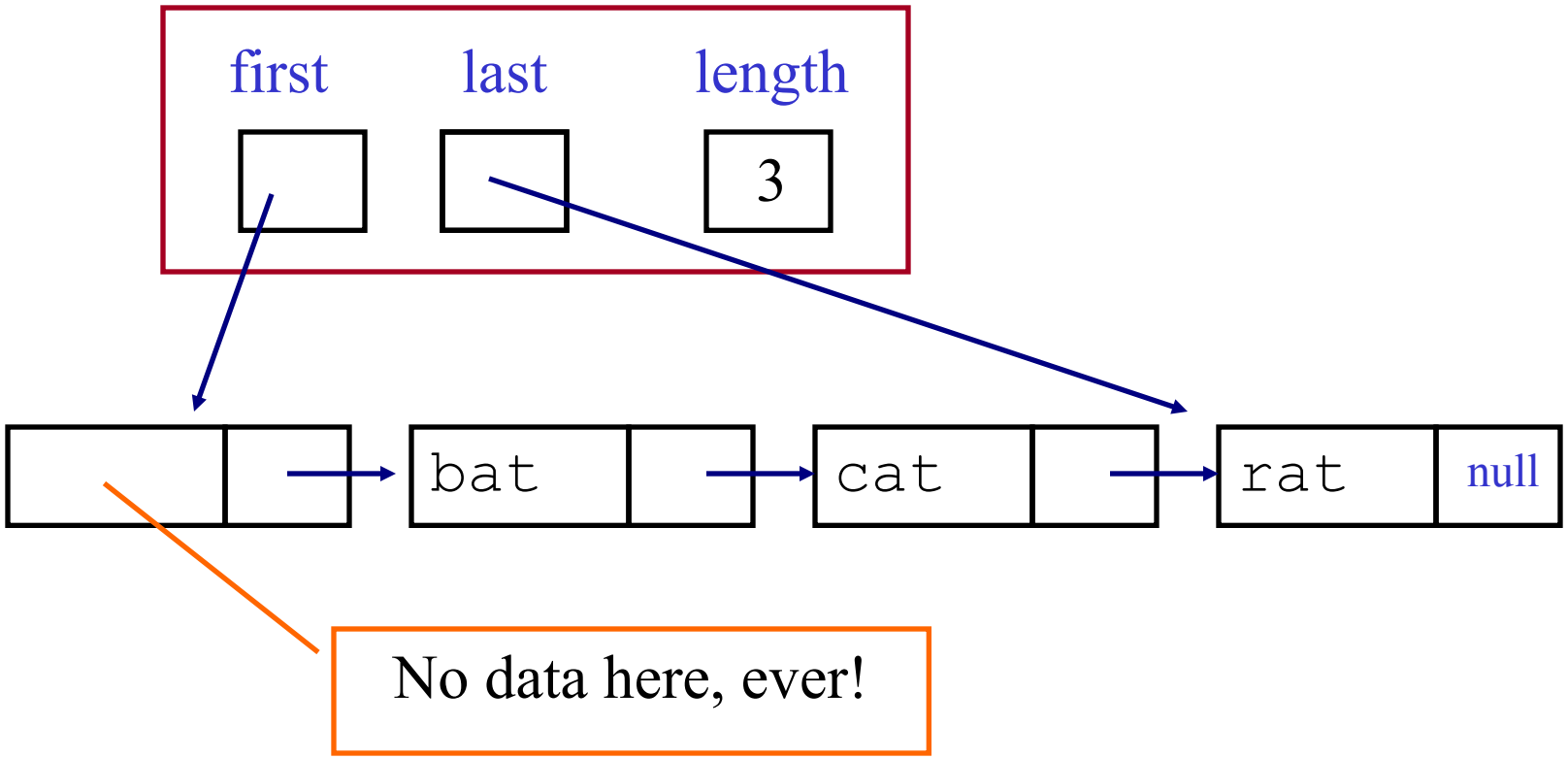
```
public static void main (String[] args)  {
    LinkedList ln = new LinkedList();
    ln.append("cat");
```

```
public void append (String s) {
  ListNode n = new ListNode(s);
  last.next = n;
  last = n;
  length++;
}
```

Error!

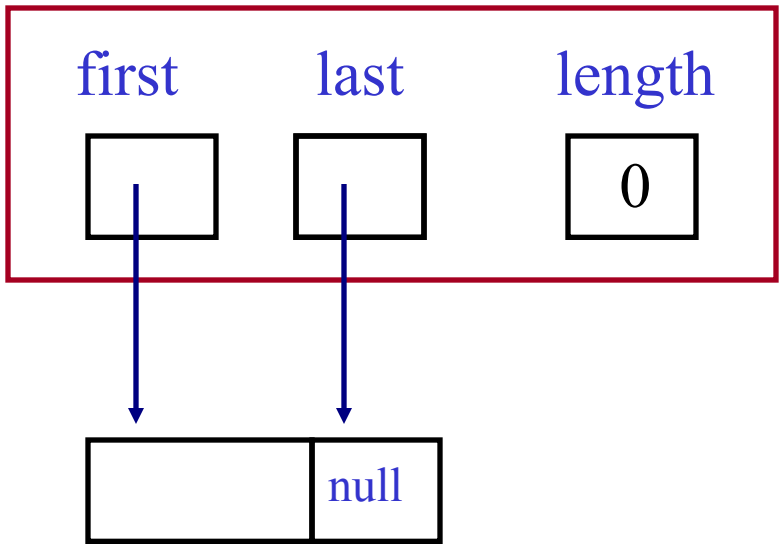


LinkedList



It is also helpful to have an empty *dummy node* at the beginning of the list

LinkedList

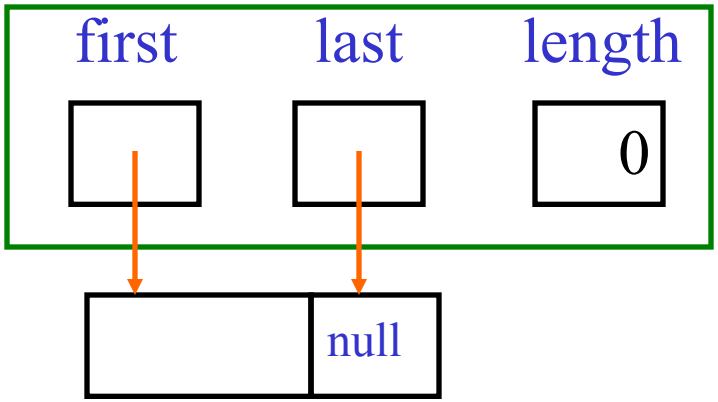



```
public class LinkedList  {
   ListNode ln = new ListNode();
   private ListNode first = ln;
   private ListNode last = ln;
   private int length = 0;
}
```

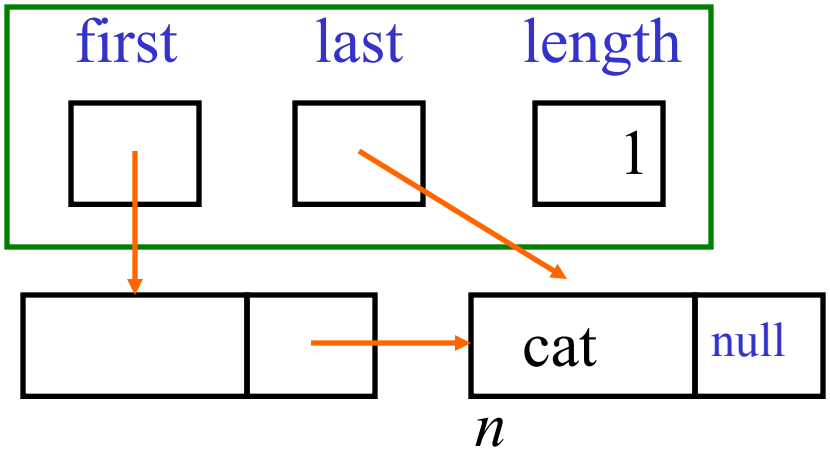
```
public static void main (String[] args)  {
    LinkedList ln = new LinkedList();
    ln.append("cat");
```

```
public void append (String s) {
  ListNode n = new ListNode(s);
  last.next = n;
  last = n;
  length++;
}
```

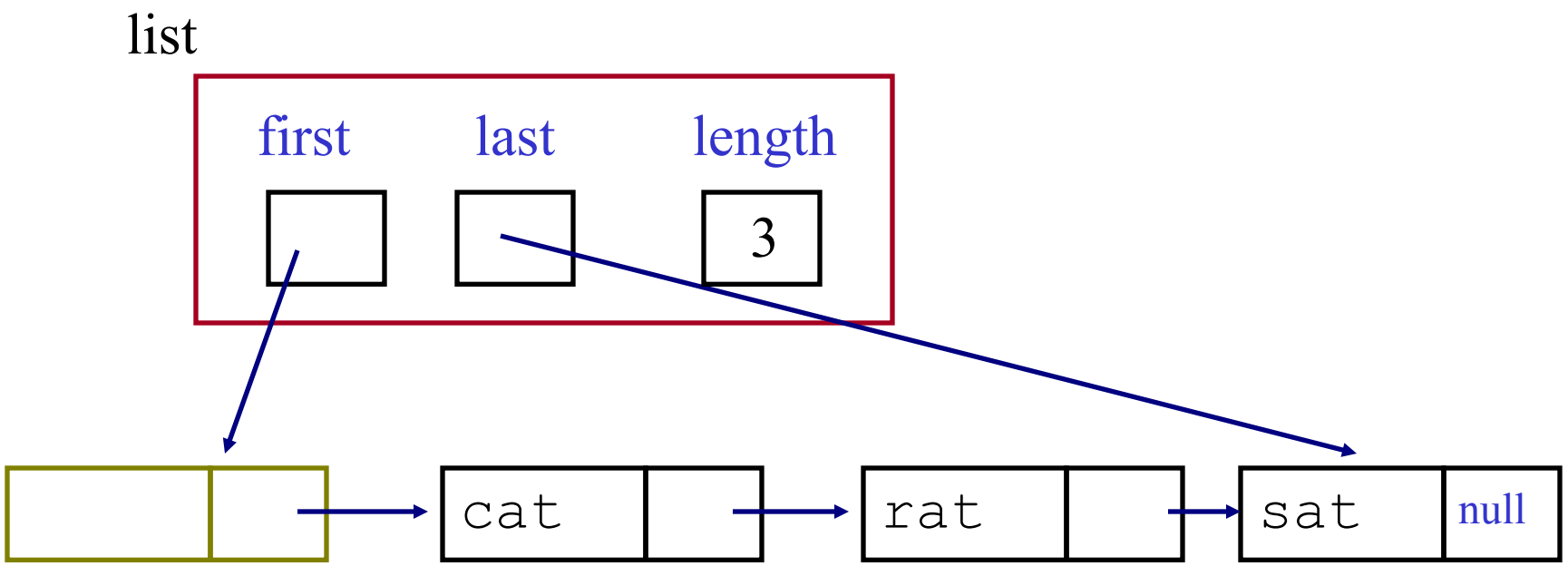
Before:



After:

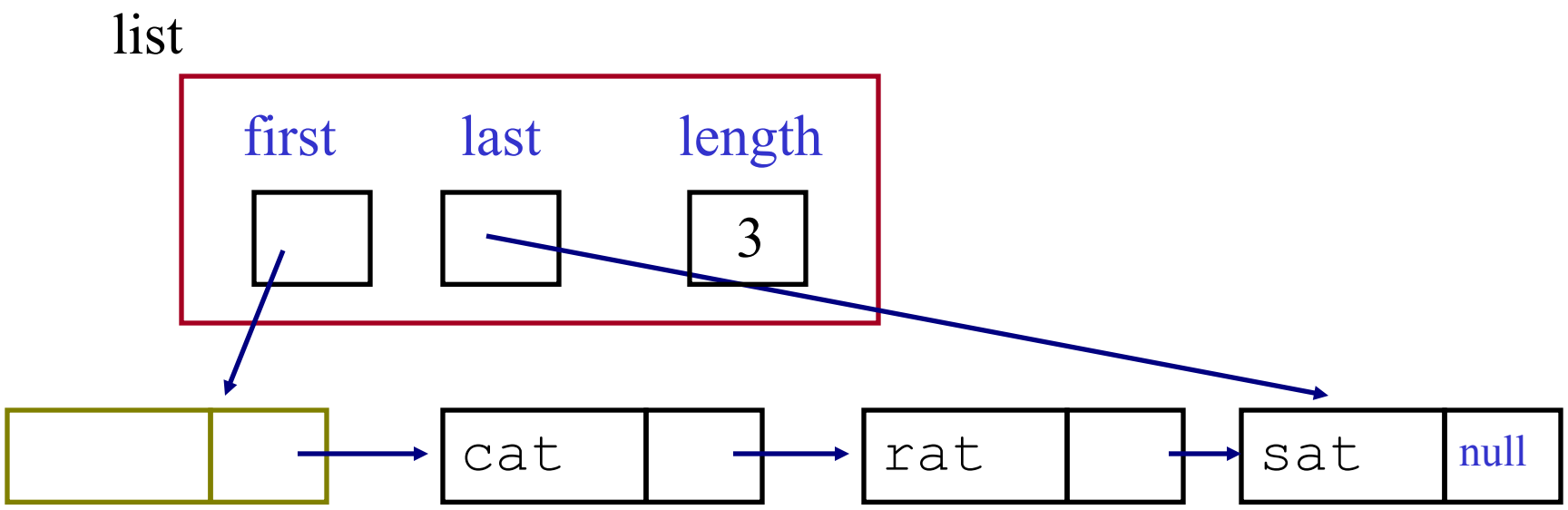


Traversing a list from beginning to end... `list.printList()`




```
public void printList () {
  ListNode p = first.next;
  while (p != null) {
    System.out.println(p.data);
    p = p.next;
  }
```

Searching a list from beginning to end... `list.find("rat");`




```
public ListNode find (String s) {
  ListNode p = first.next;
  while (p != null && !(p.data).equals(s)) {
    p = p.next;
  } // while
  return p;
}
```

Note that this method returns a *ListNodePointer*

```
public ListNode find (String s) {
  ListNode p = first.next;
  while (p != null && !(p.data).equals(s)) {
    p = p.next;
  } // while
  return p;
}
```

Why give a pointer to a node in the list to the caller?

```
// some caller's code...
ListNode n =
myList.find("rat");
n.next = null;
```

An error if *next* is not public

A disaster to the list is if this is allowed!

What can the user of this *LinkedList* class do?

```
ln = new LinkedList();
ln.append(String s);
ln.printList();
ln.find (String s);
```

Suppose the user wants to print all the words in the list that start with the letter "a"?

```
ListNode p = first.next;
  while (p != null) {
    if (p.data.charAt(0)=='a')
      System.out.println(p.data);
    p = p.next;
```

But this gives the user access to our pointers!

We need to let the user look at individual nodes in the list without granting access to its internal data.

The solution is to use another class called an *Iterator* contained in the class that has sensitive data, but which provides controlled access to the data

LinkedList

```
public LinkedListIterator reset()    {
        return new LinkedListIterator(first.next);
    }
```

User

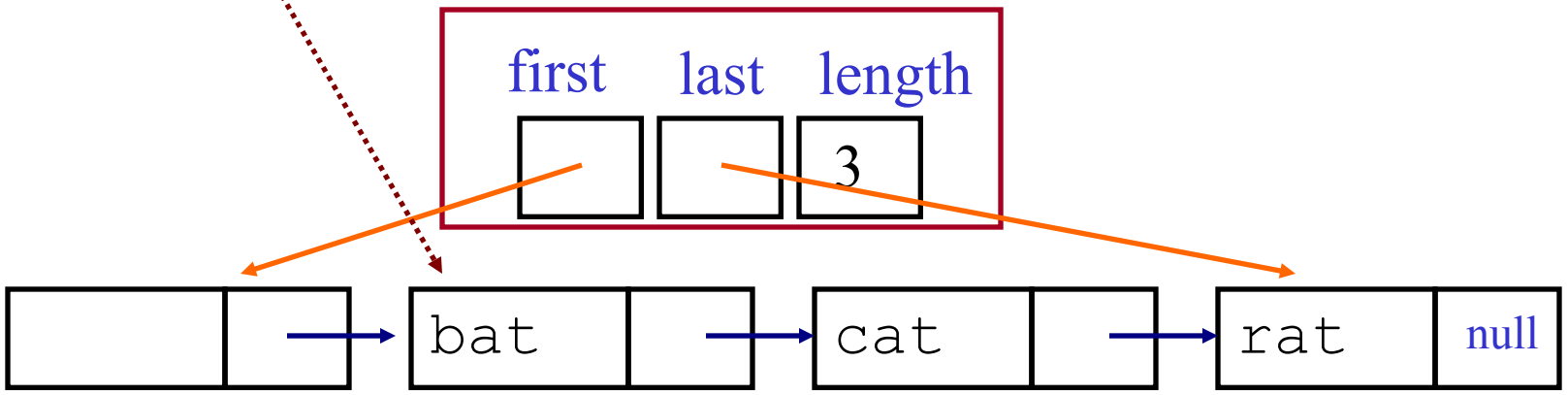
```
LinkedListIterator lli = myList.reset();
while (lli.hasNext()) {
    myString = lli.next();
    if (myString.charAt(0) == 'a' .....
}
```

LinkedList

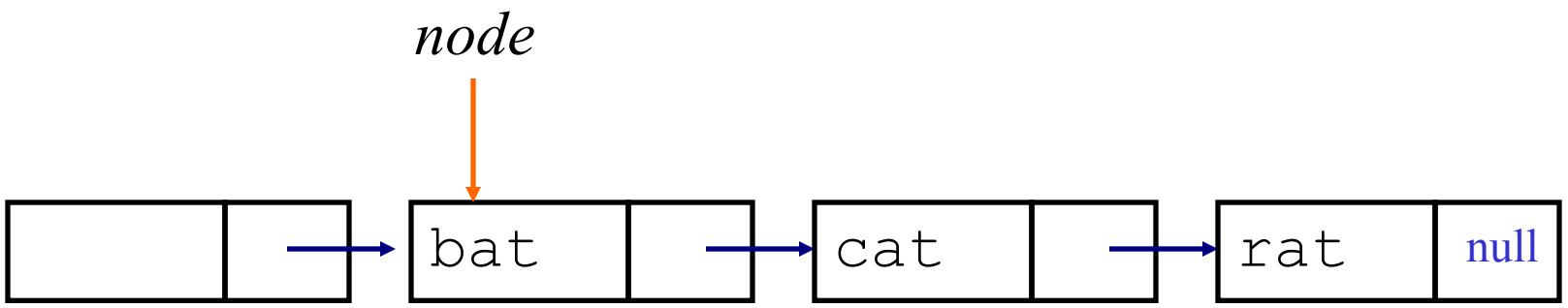
```
public LinkedListIterator reset()    {
       return new LinkedListIterator(first.next);
   }
```

LinkedListIterator

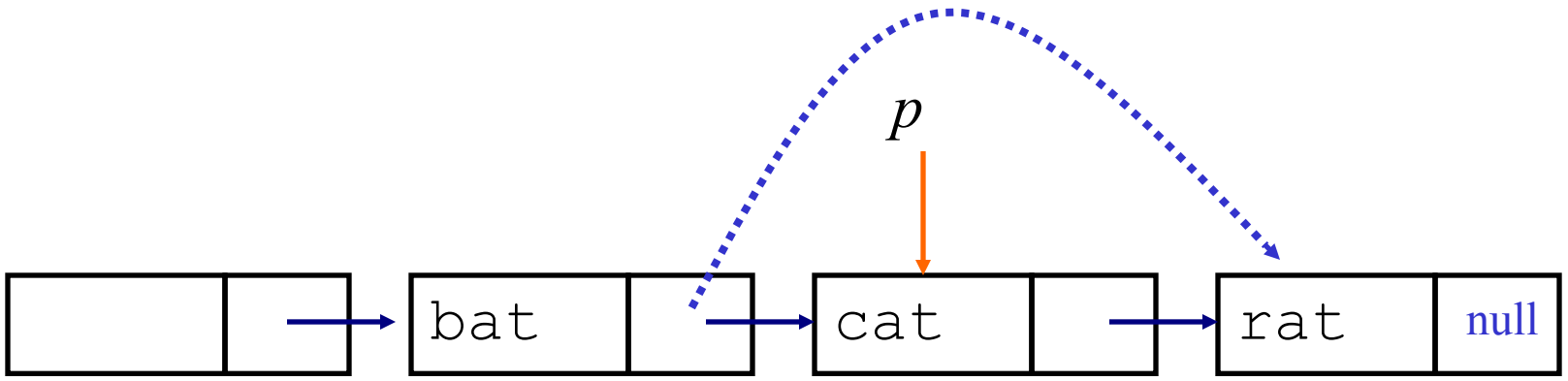
```
private ShortNode node;
public LinkedListIterator(ListNode first)   {
     node = first;
}
```



```
public boolean hasNext()  {
    return ( node != null );
}
public String next()  {
    if ( node == null )
        throw new NullPointerException("Linked list empty.");
    String currentData = node.data;
    node = node.next;
    return currentData;
}
```



```
public void remove(Listnode p)  {
    if ( p == null )
        throw new InvalidArgumentException("Null pointer.");


}
```



Back to the list..

what can it do?

Remove an item

Find element with "cat"

if found, remove it