Module 9: List Structures

Ron K. Cytron *

Department of Computer Science and Engineering *
Washington University in Saint Louis

Thanks to Alan Waldman
for comments that improved these slides

Prepared for 2u

Semester Online

Monolog 0

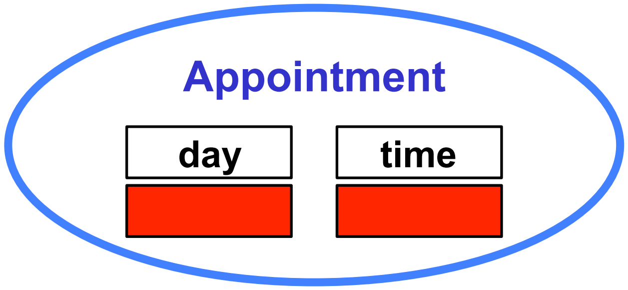
9.0 Introduction

- **Recall methods**
  - **One method can call another**
  - **A method can even call itself – this is recursion**
- **Recall objects**
  - **One object can reference (point to) other objects**
  - **An object can even reference itself, a form of recursion**
  - **More commonly, an object references another object of the same time**
    - These are also called recursive data structures
  - **We study such objects here**
- **Lists, Sets, Maps**
  - **Are built out of such *recursive* objects**

End of Monologue

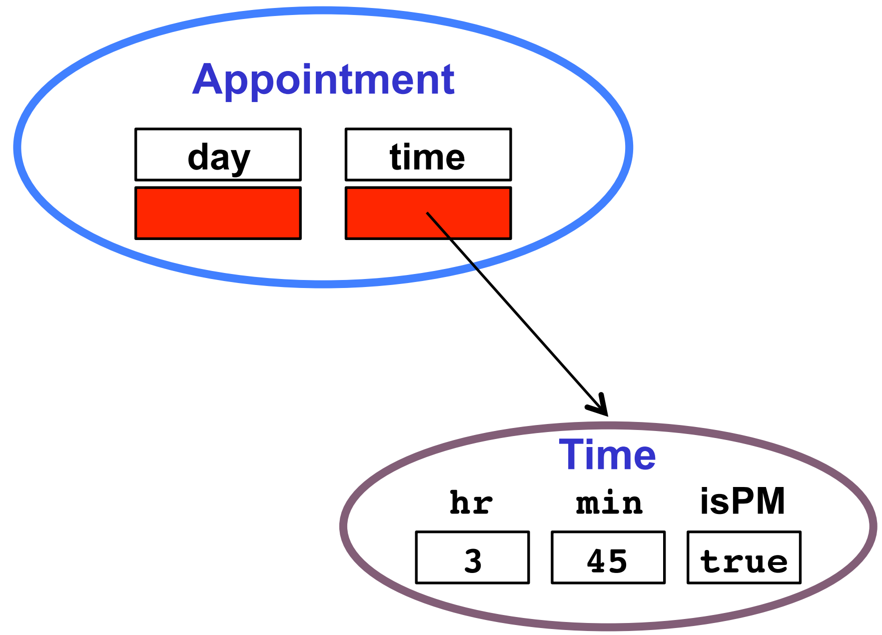
9.1 List Nodes

- **Objects can reference other objects**



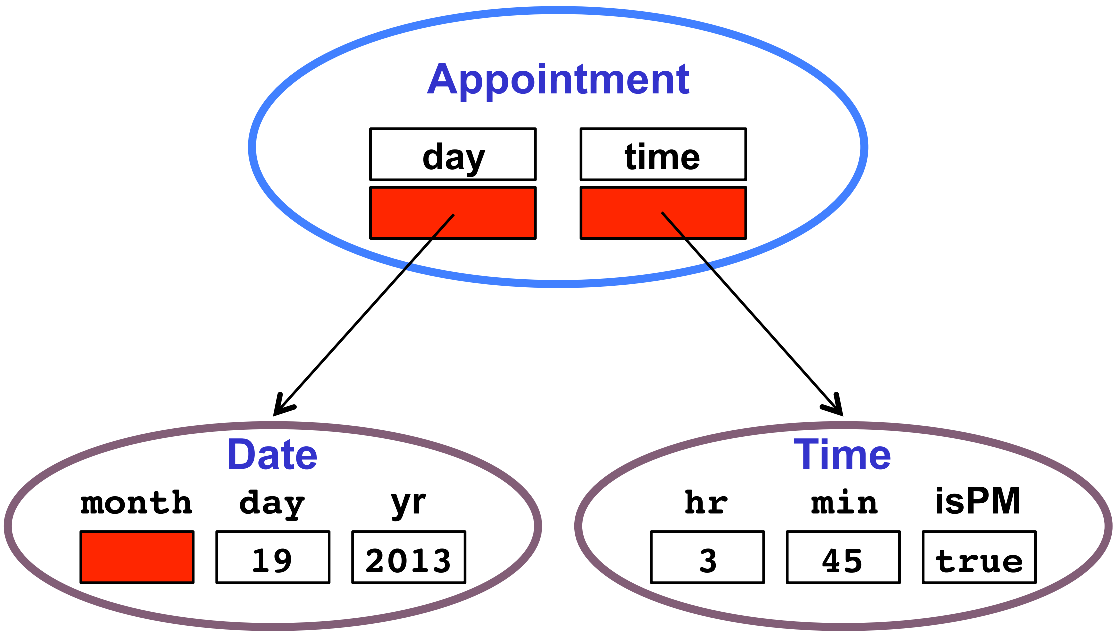
9.1 List Nodes

- **Objects can reference other objects**
  - **Which may contain primitive values**



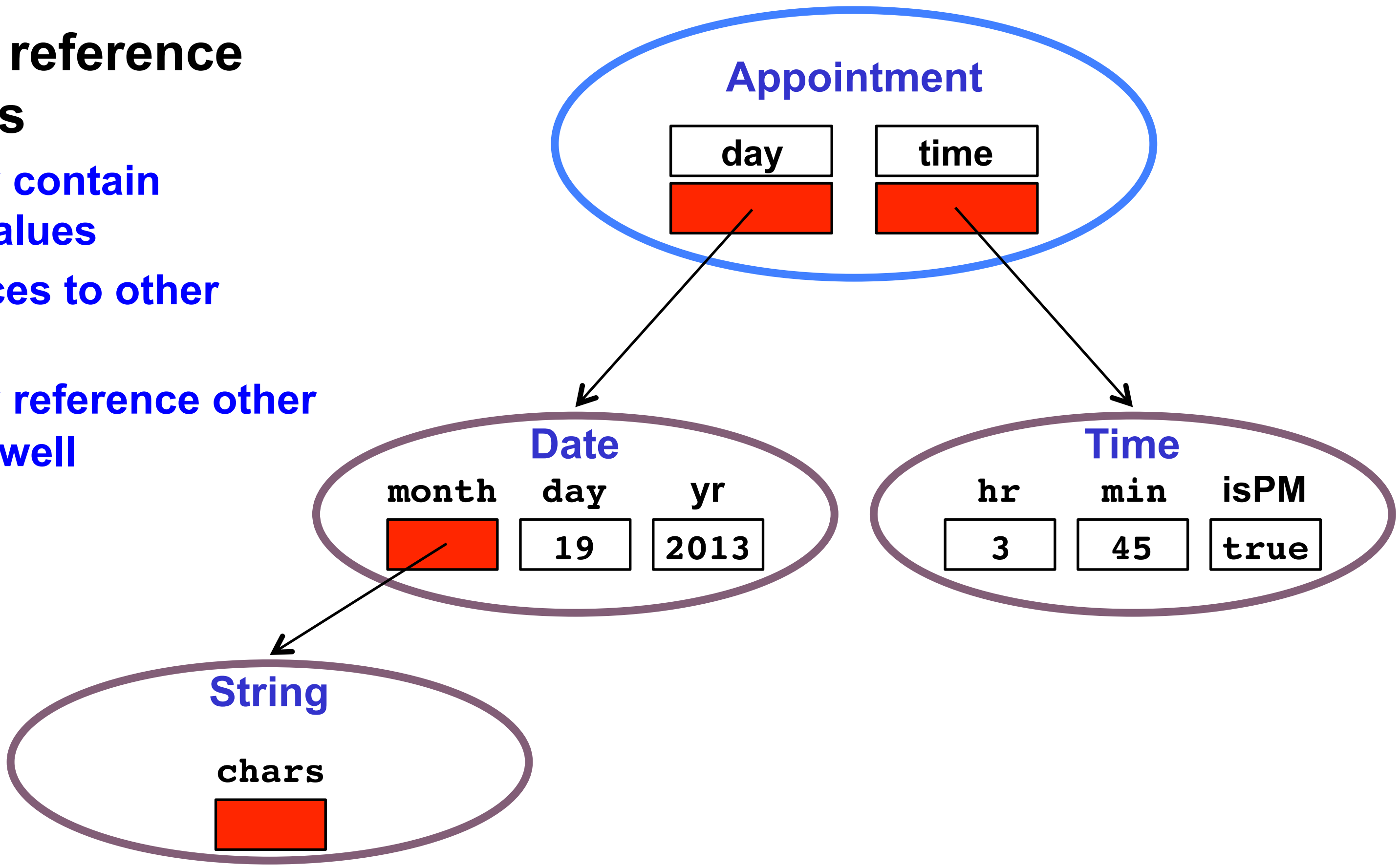
9.1 List Nodes

- **Objects can reference other objects**
  - **Which may contain primitive values**
  - **Or references to other objects**



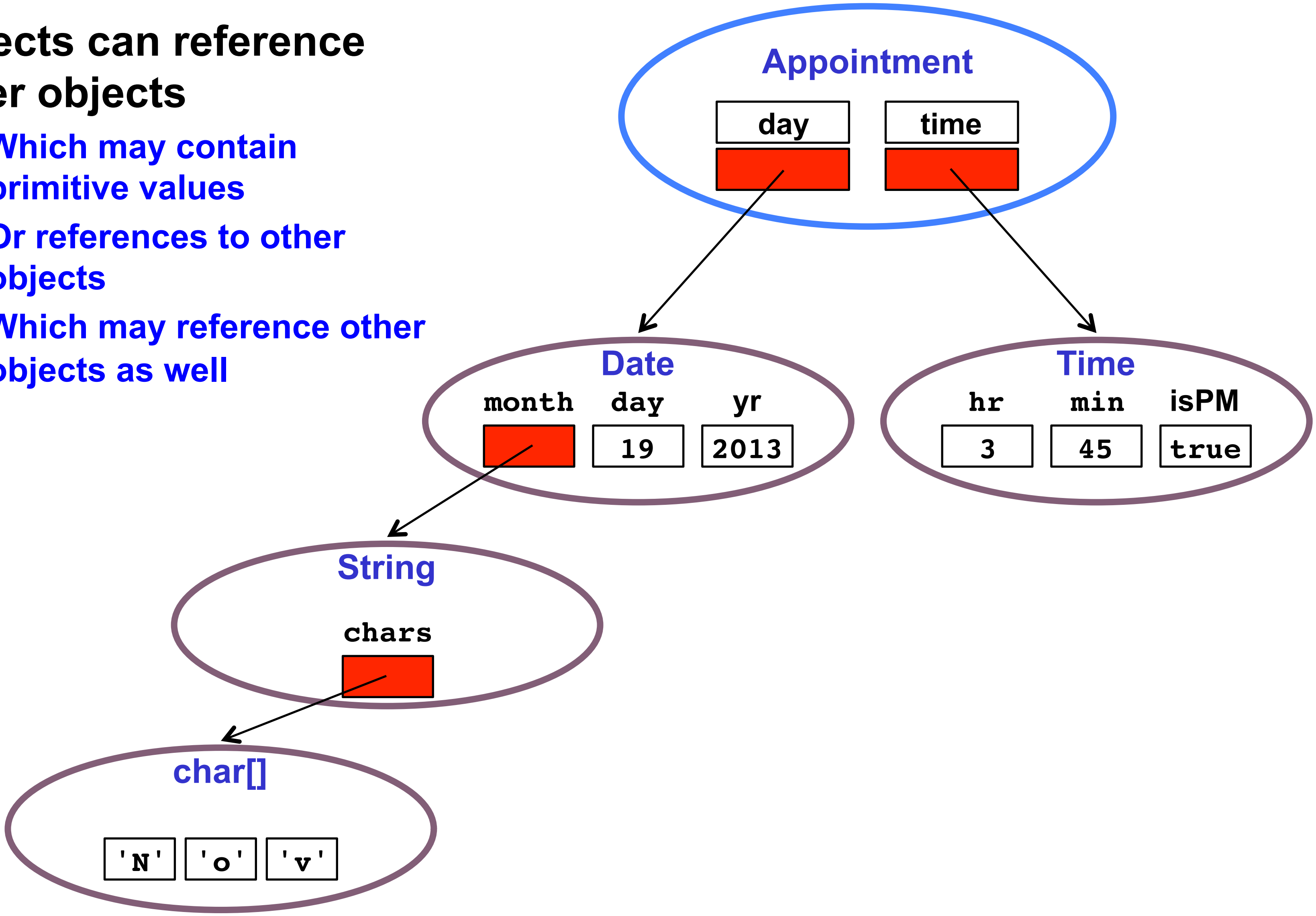
9.1 List Nodes

- **Objects can reference other objects**
  - **Which may contain primitive values**
  - **Or references to other objects**
  - **Which may reference other objects as well**



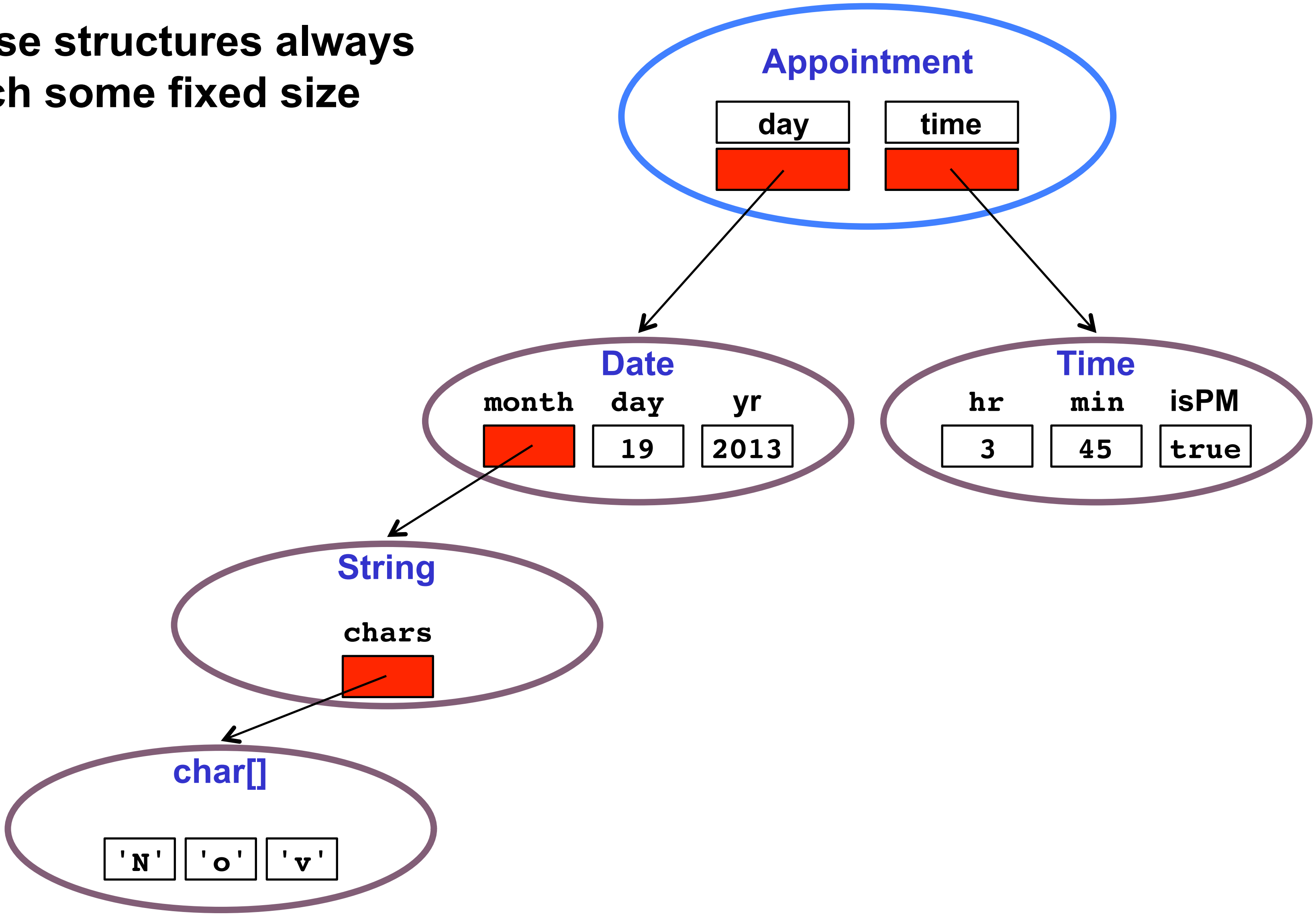
9.1 List Nodes

- Objects can reference other objects
  - Which may contain primitive values
  - Or references to other objects
  - Which may reference other objects as well



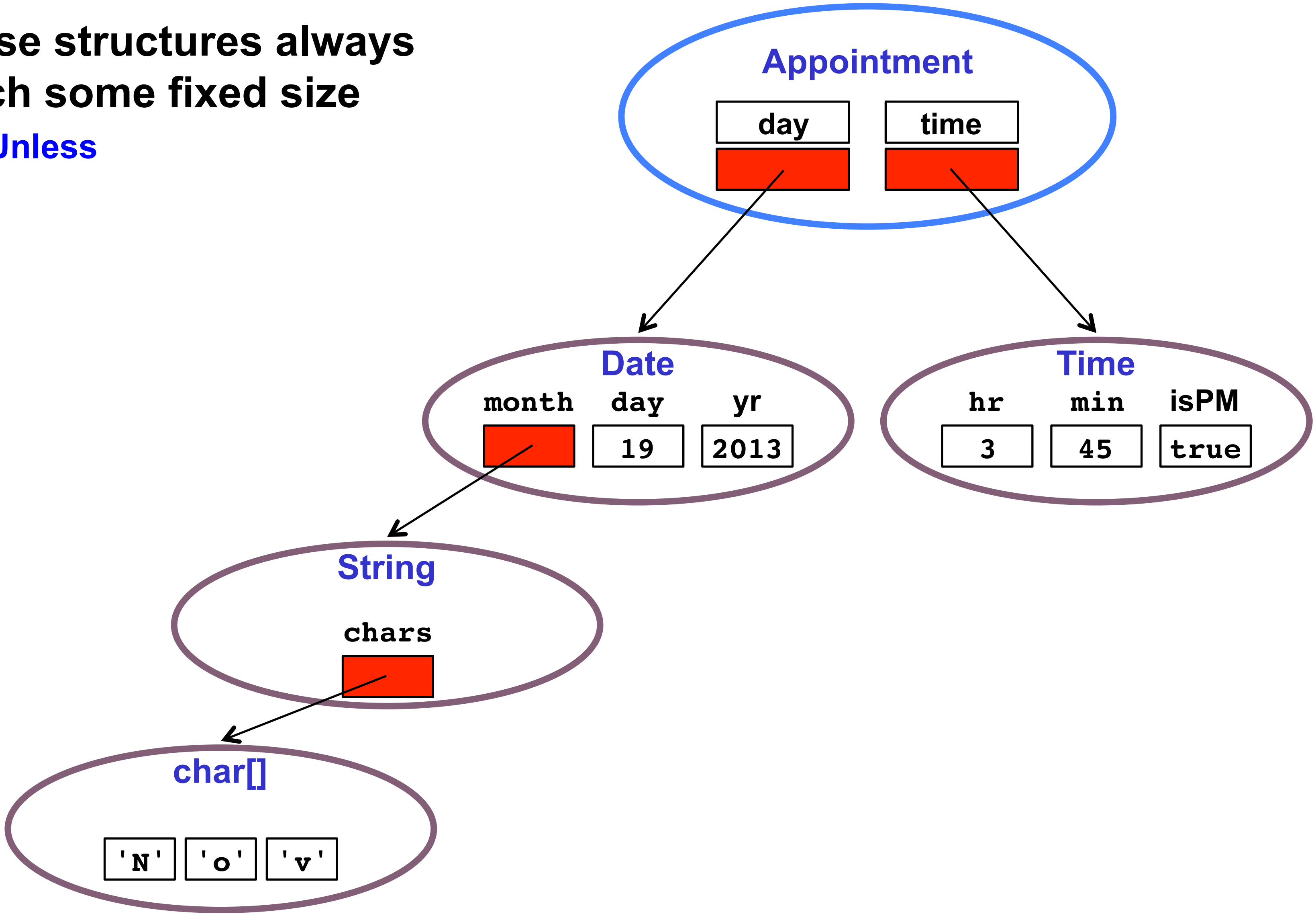
9.1 List Nodes

- These structures always reach some fixed size



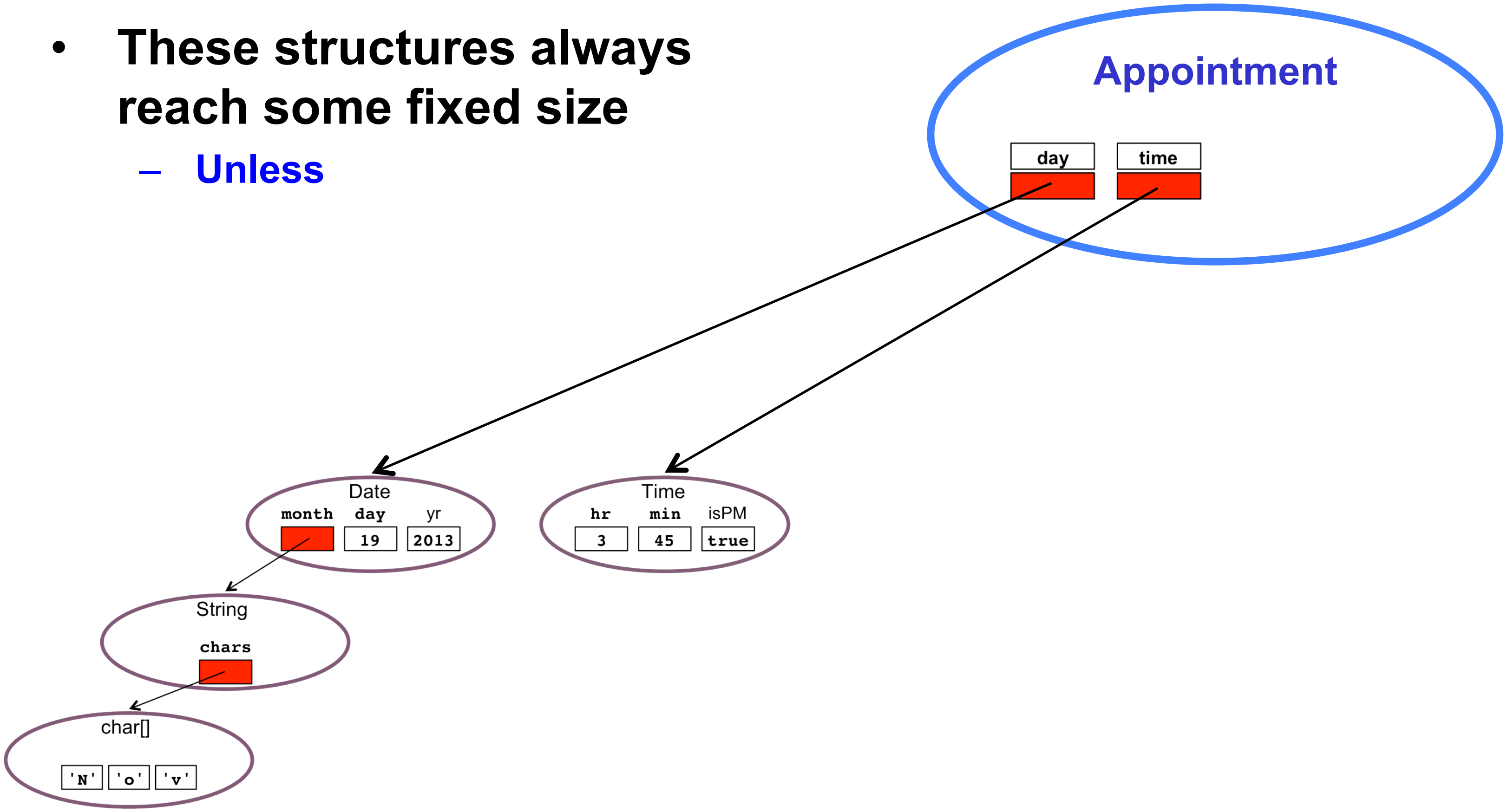
9.1 List Nodes

- These structures always reach some fixed size
  - Unless



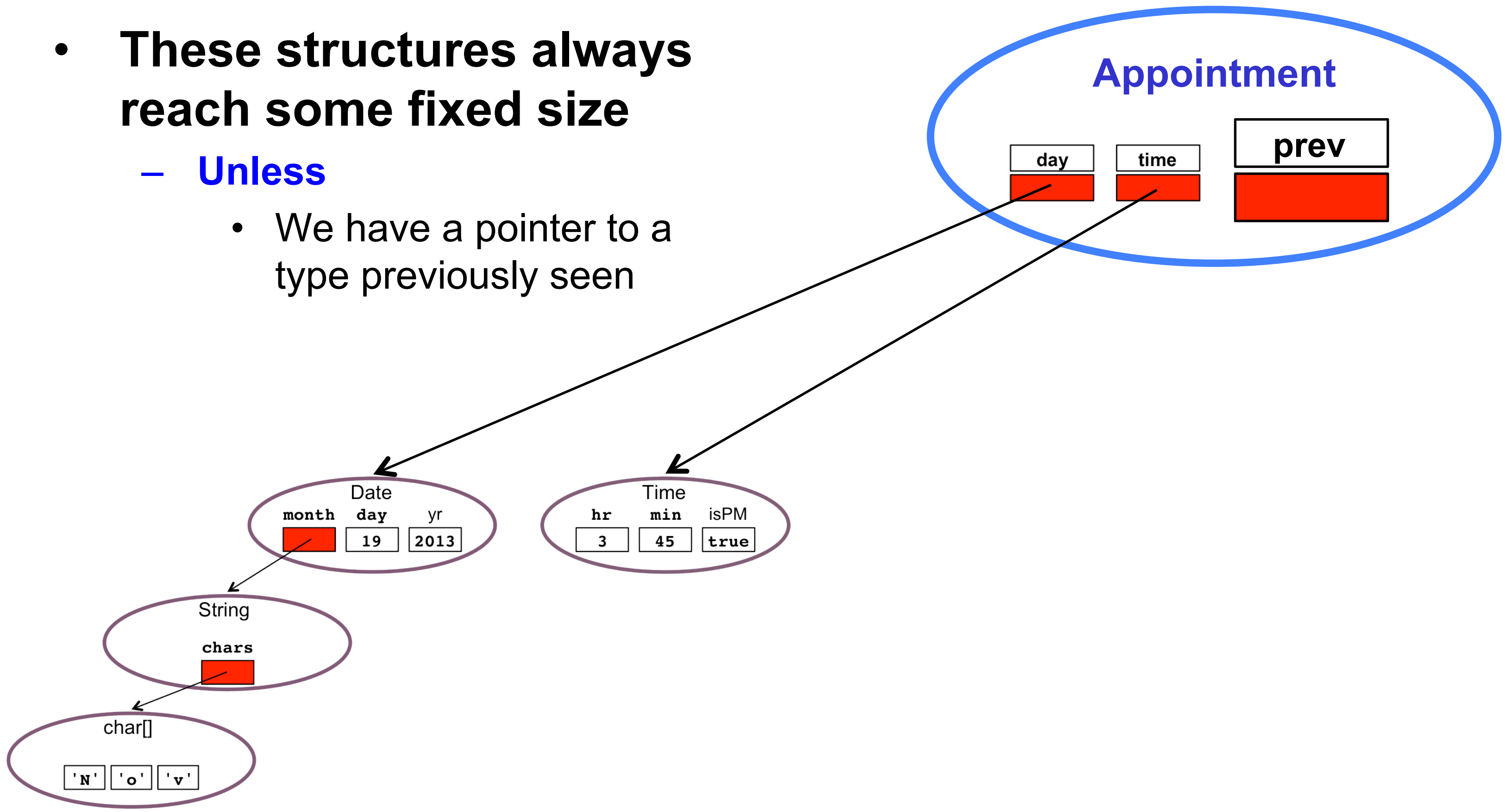
9.1 List Nodes

- These structures always reach some fixed size
  - Unless



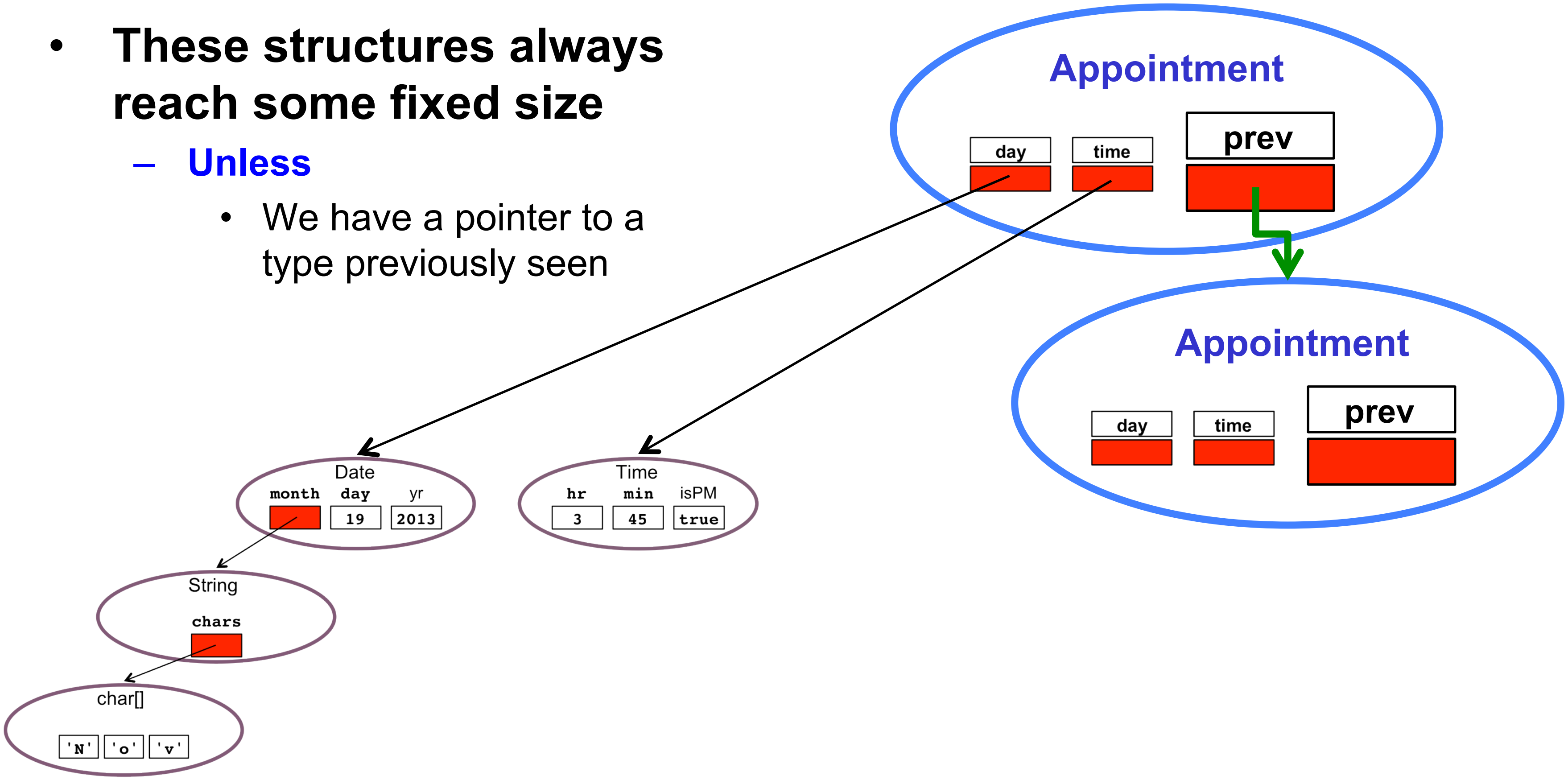
9.1 List Nodes

- **These structures always reach some fixed size**
  - **Unless**
    - We have a pointer to a type previously seen



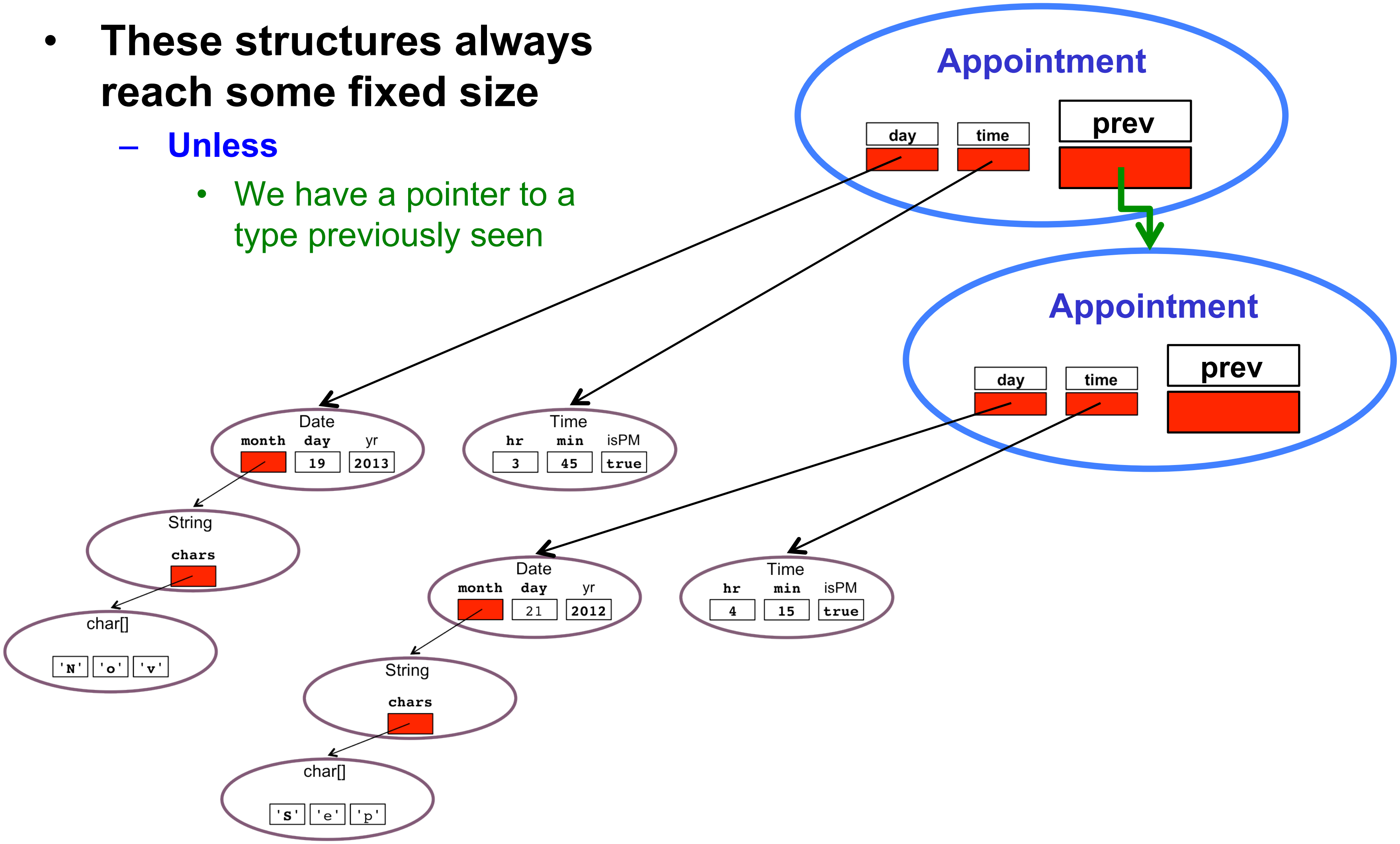
9.1 List Nodes

- **These structures always reach some fixed size**
  - **Unless**
    - We have a pointer to a type previously seen



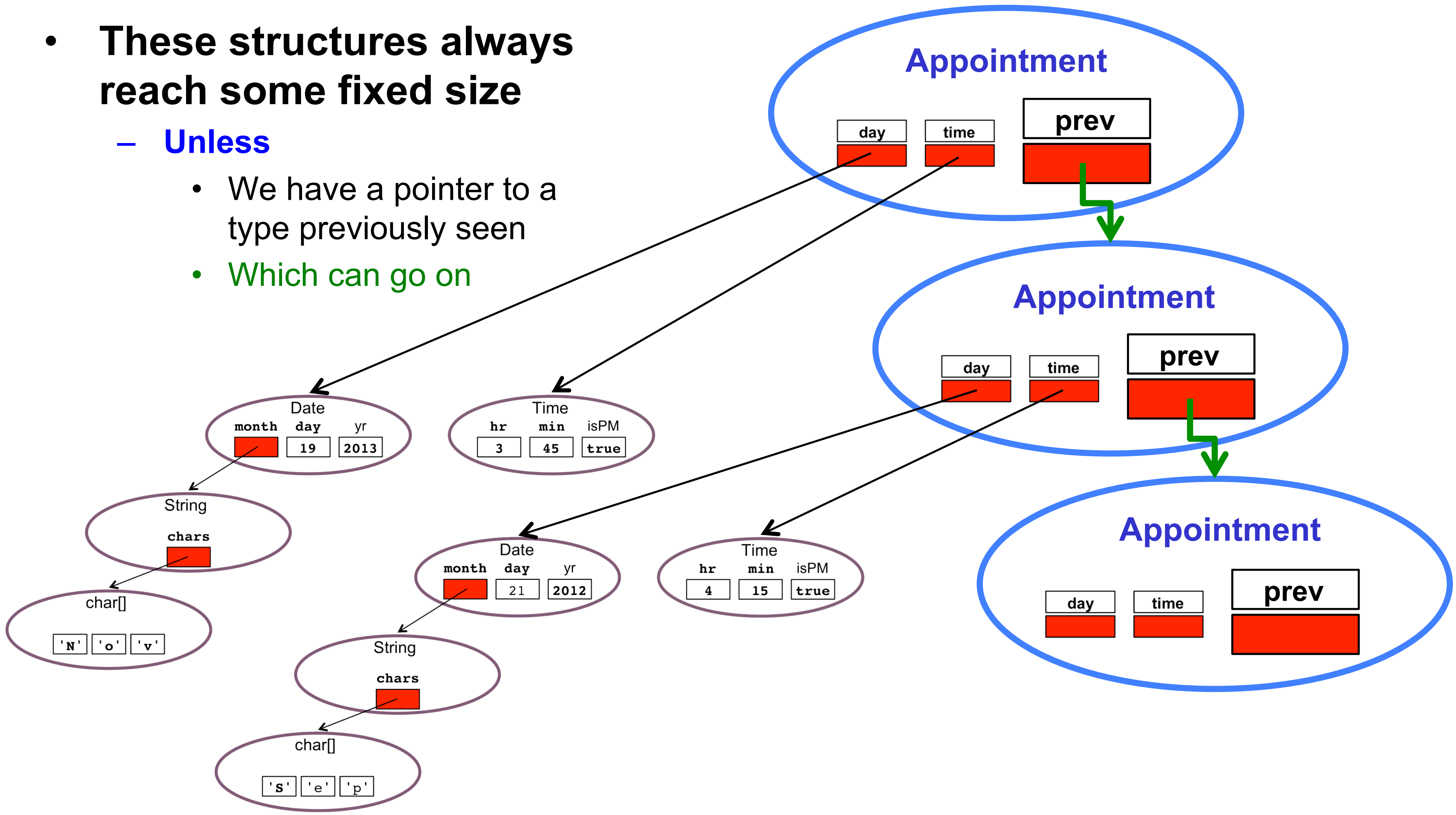
9.1 List Nodes

- **These structures always reach some fixed size**
  - **Unless**
    - We have a pointer to a type previously seen



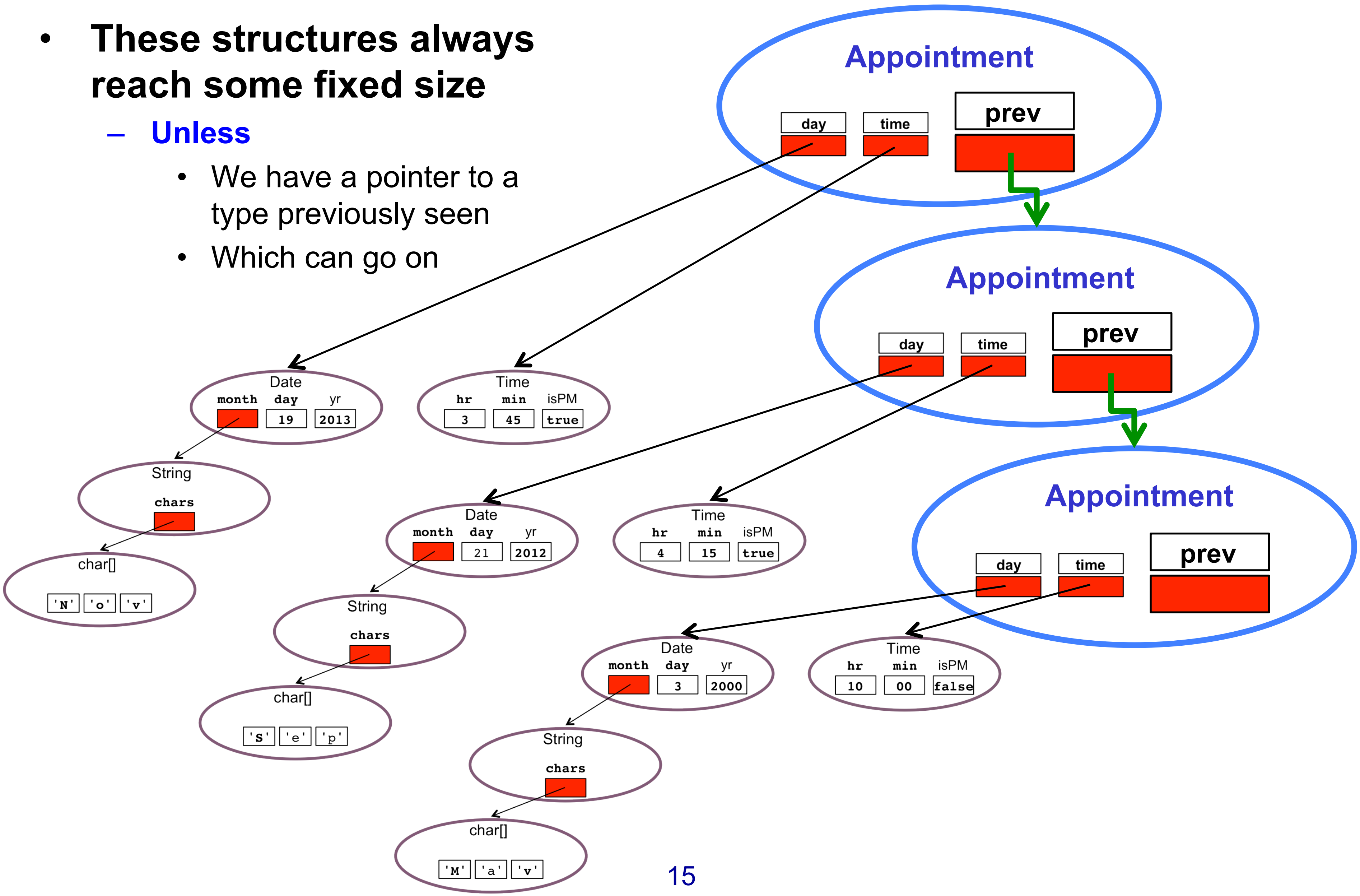
9.1 List Nodes

- **These structures always reach some fixed size**
  - **Unless**
    - We have a pointer to a type previously seen
    - Which can go on



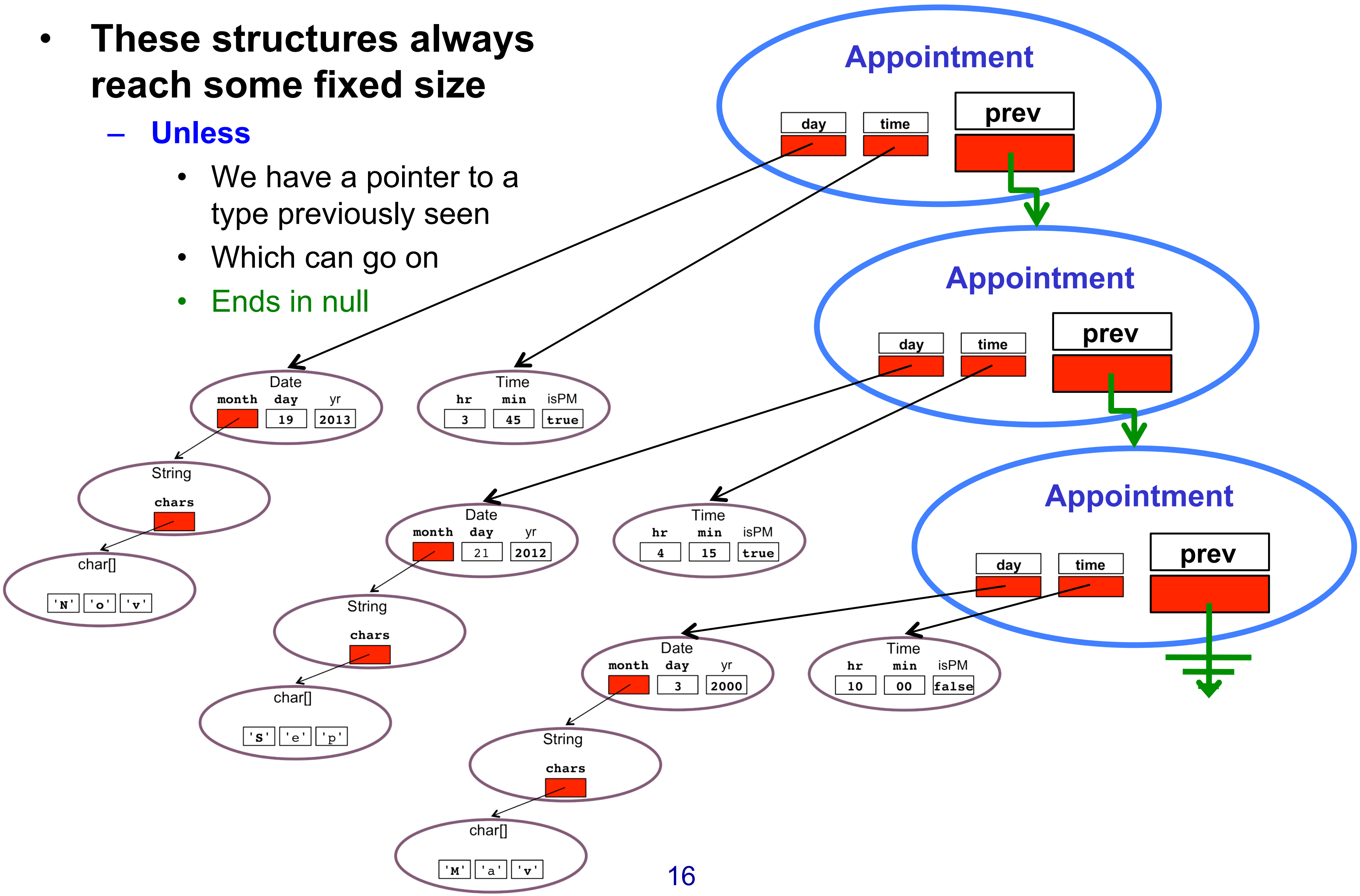
9.1 List Nodes

- **These structures always reach some fixed size**
  - **Unless**
    - We have a pointer to a type previously seen
    - Which can go on



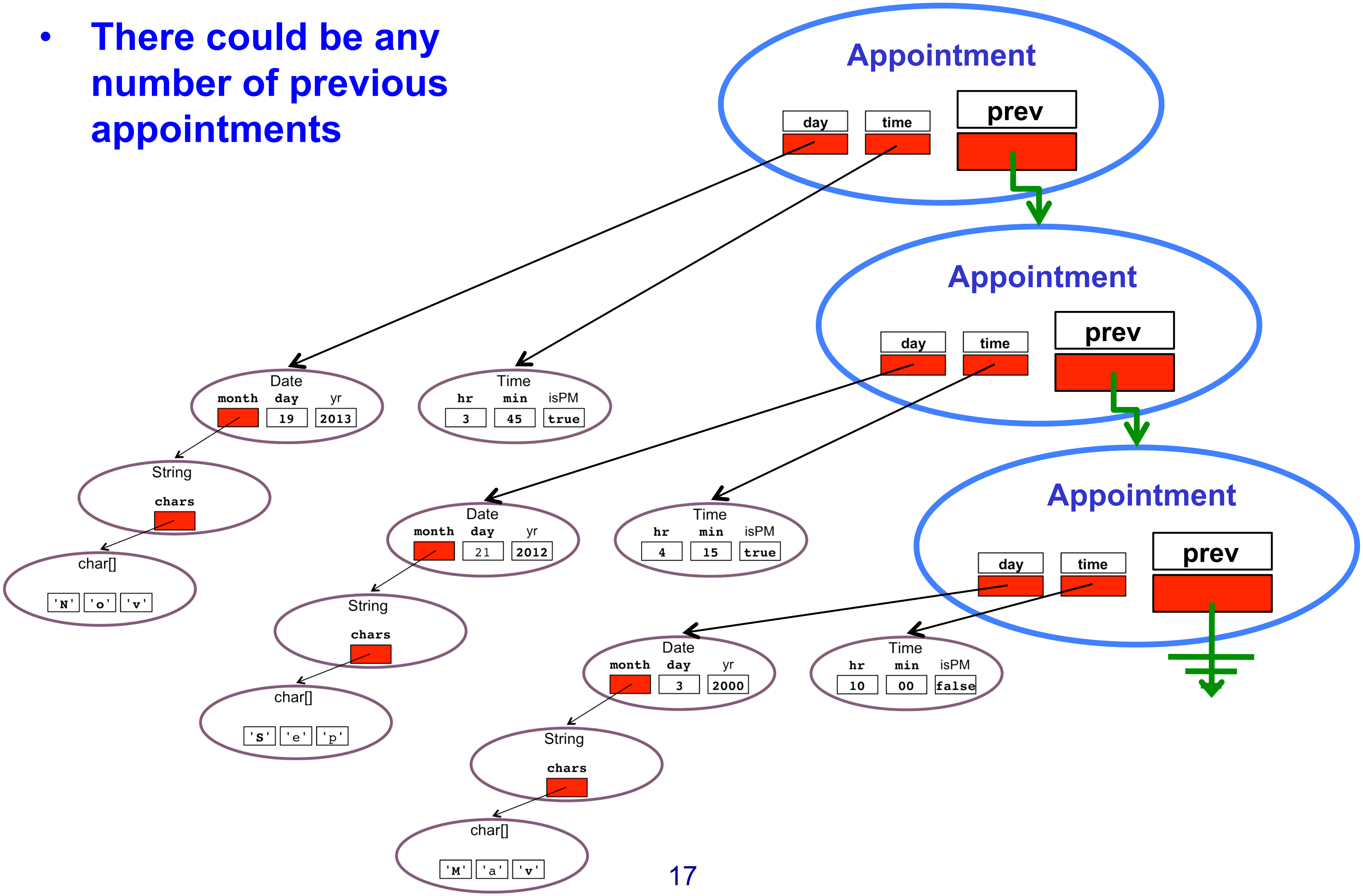
9.1 List Nodes

- **These structures always reach some fixed size**
  - **Unless**
    - We have a pointer to a type previously seen
    - Which can go on
    - Ends in null



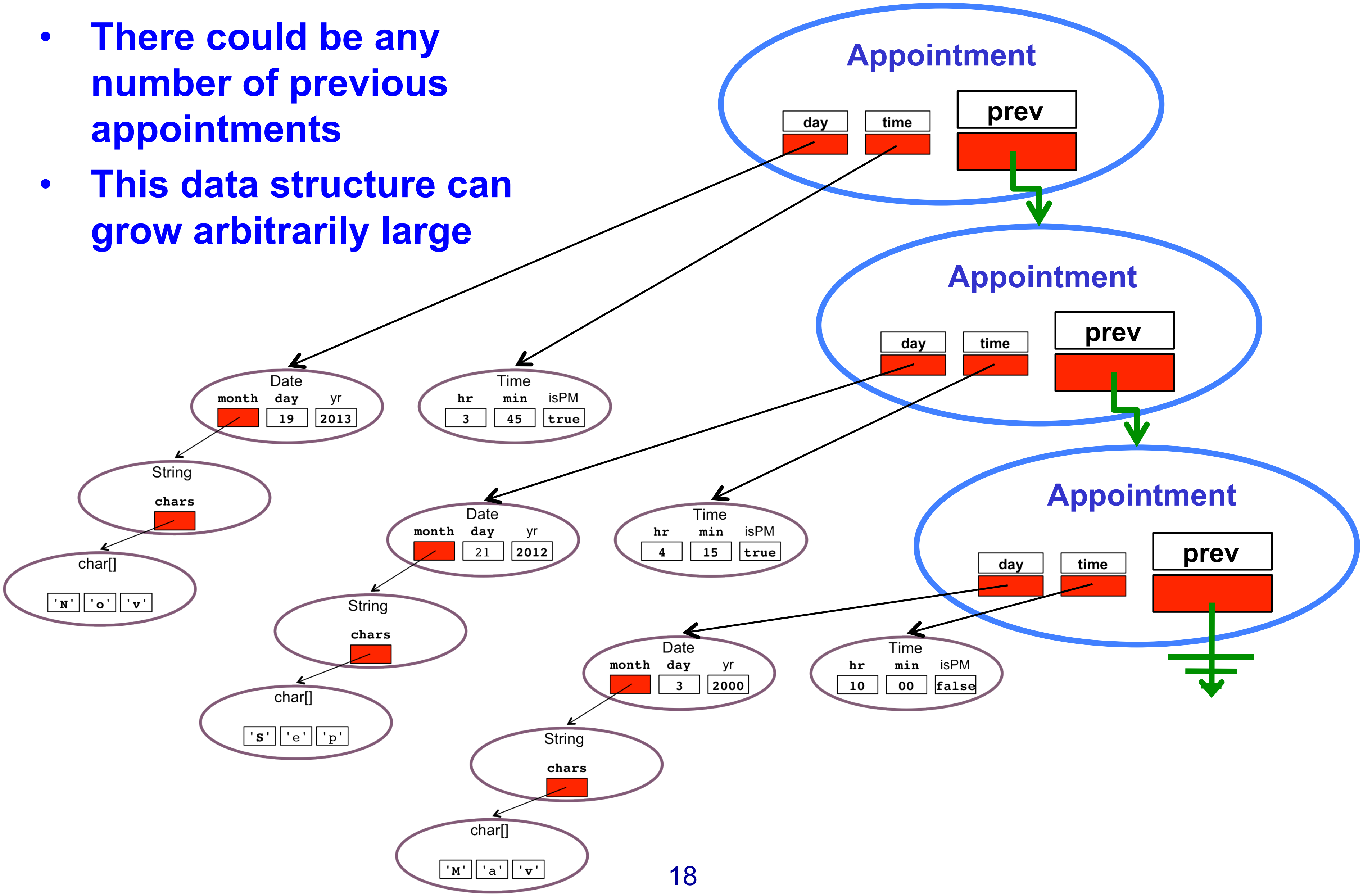
9.1 List Nodes

- There could be any number of previous appointments



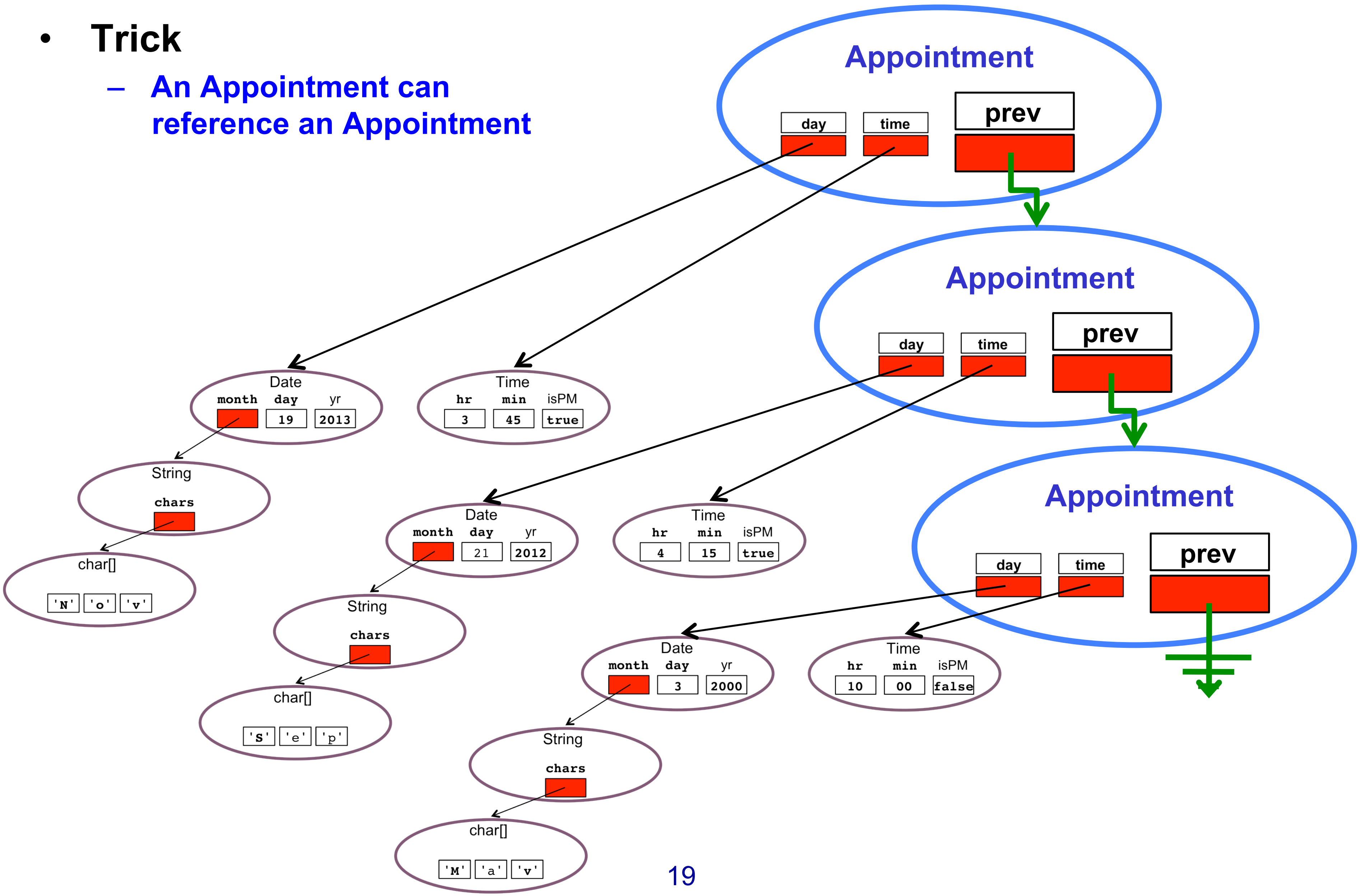
9.1 List Nodes

- There could be any number of previous appointments
- This data structure can grow arbitrarily large



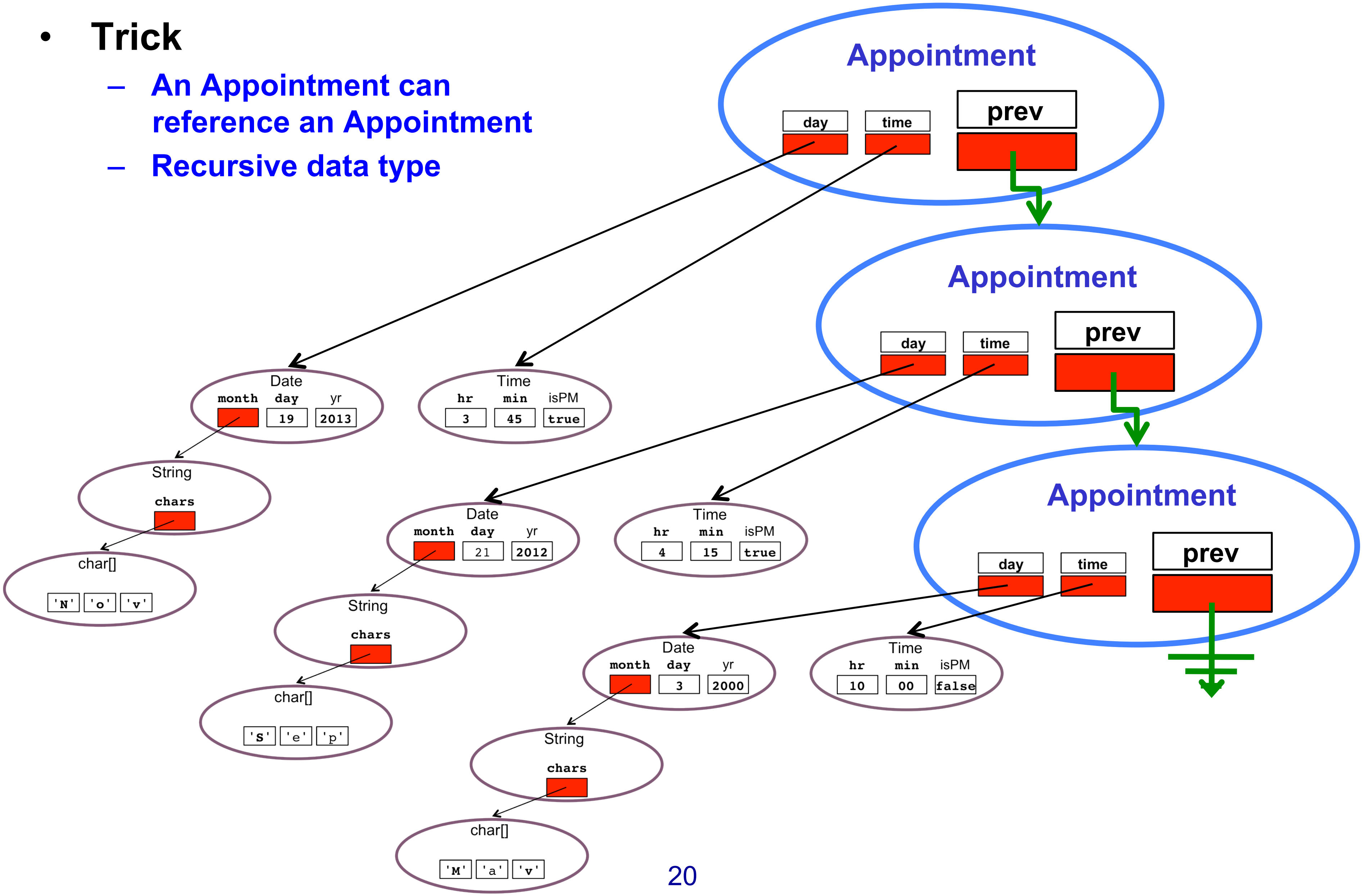
9.1 List Nodes

- **Trick**
  - **An Appointment can reference an Appointment**



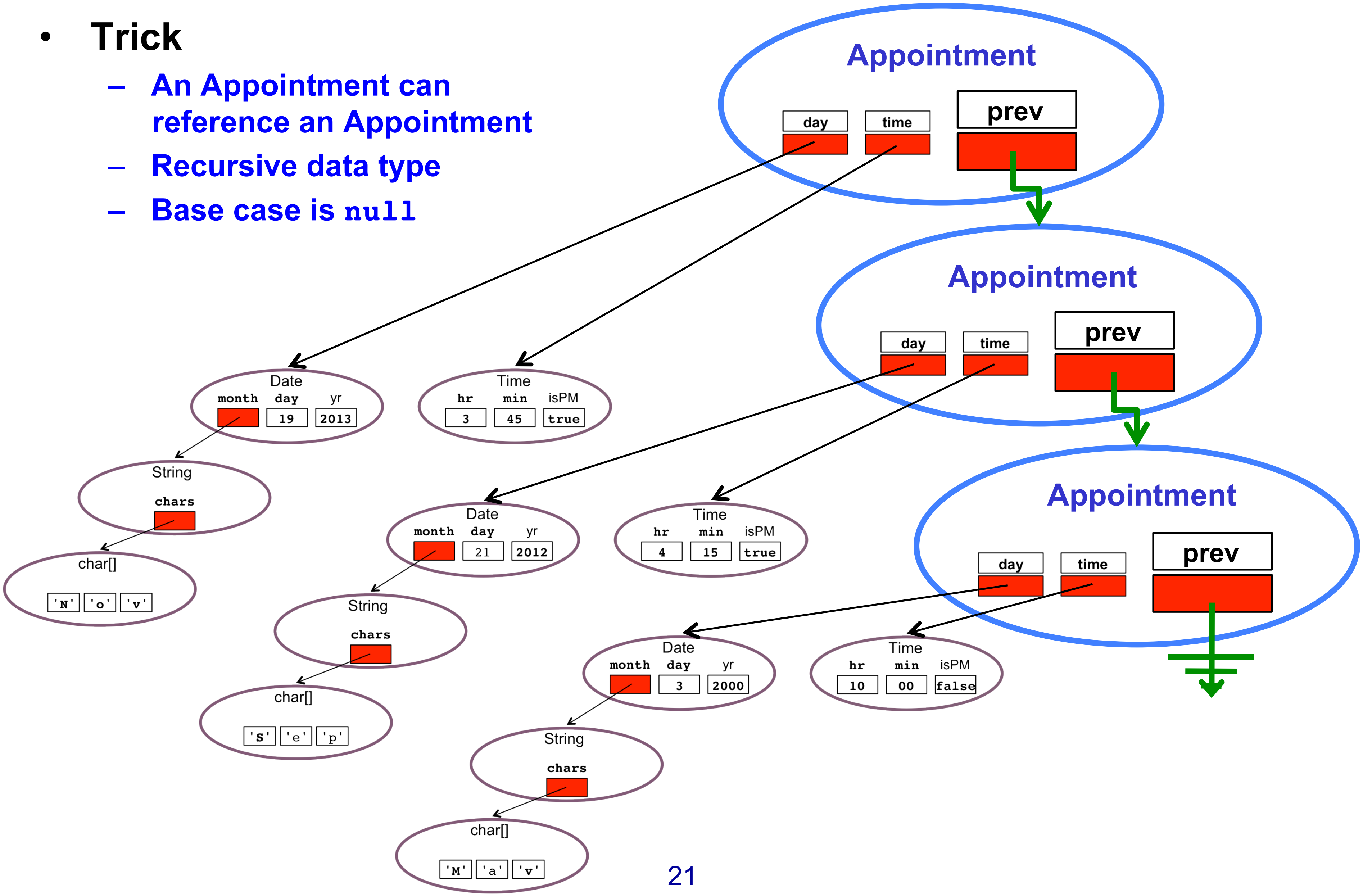
9.1 List Nodes

- **Trick**
  - **An Appointment can reference an Appointment**
  - **Recursive data type**



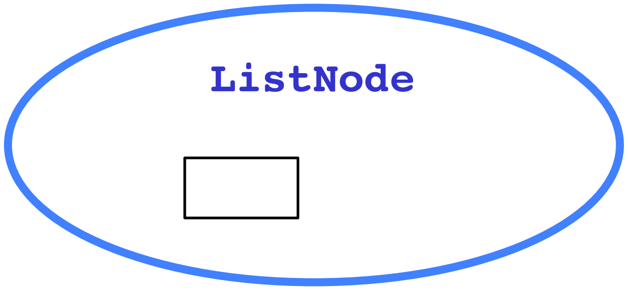
9.1 List Nodes

- **Trick**
  - **An Appointment can reference an Appointment**
  - **Recursive data type**
  - **Base case is `null`**



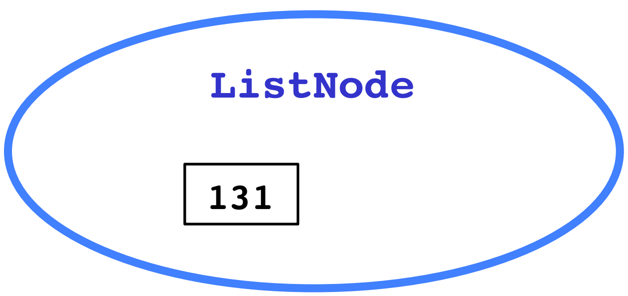
9.1 List Nodes

- **Generic form of this idea**
  - **`ListNode`**
    - has-a
      - payload
        - » Value of interest in the list



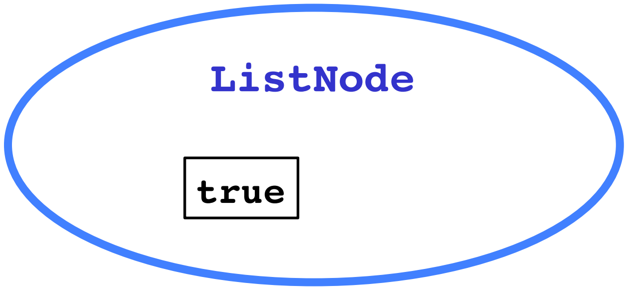
9.1 List Nodes

- **Generic form of this idea**
  - **`ListNode`**
    - has-a
      - payload
        - » Could be an `int`



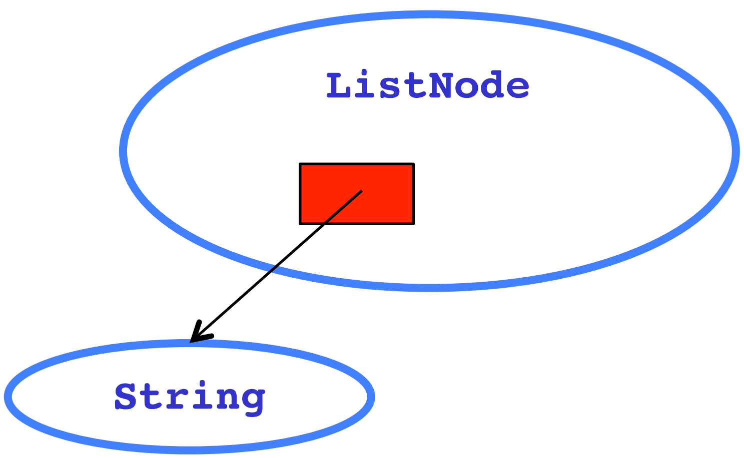
9.1 List Nodes

- **Generic form of this idea**
  - **`ListNode`**
    - has-a
      - payload
        - » Could be a `boolean`



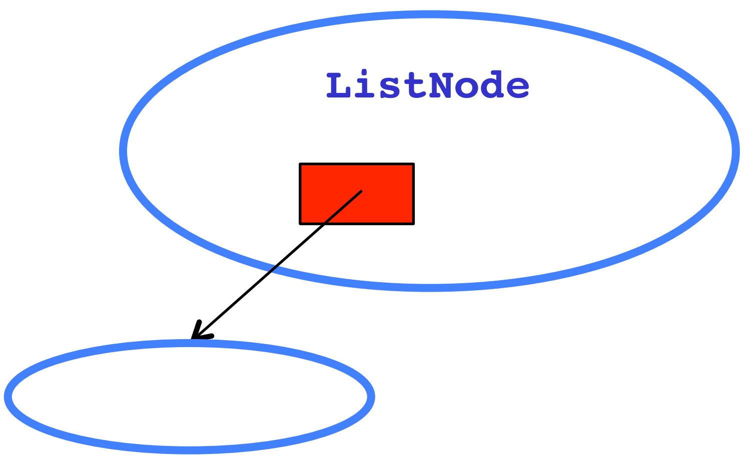
9.1 List Nodes

- **Generic form of this idea**
  - **`ListNode`**
    - has-a
      - payload
        - » Could be a `String`



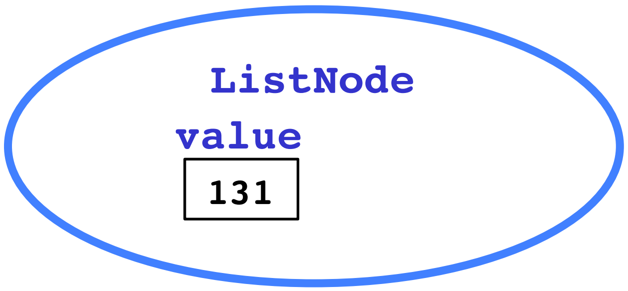
9.1 List Nodes

- **Generic form of this idea**
  - **`ListNode`**
    - has-a
      - payload
        - » Could be or any other kind of reference



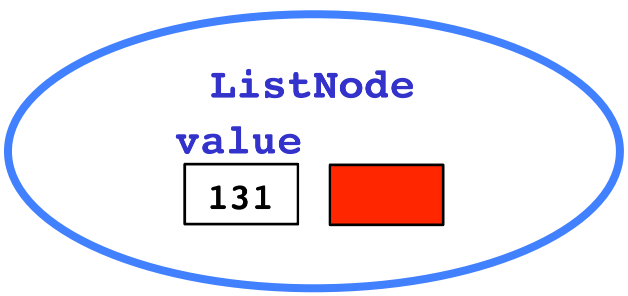
9.1 List Nodes

- **Generic form of this idea**
  - **`ListNode`**
    - has-a
      - payload
        - » Let's assume it's an `int`, named `value`



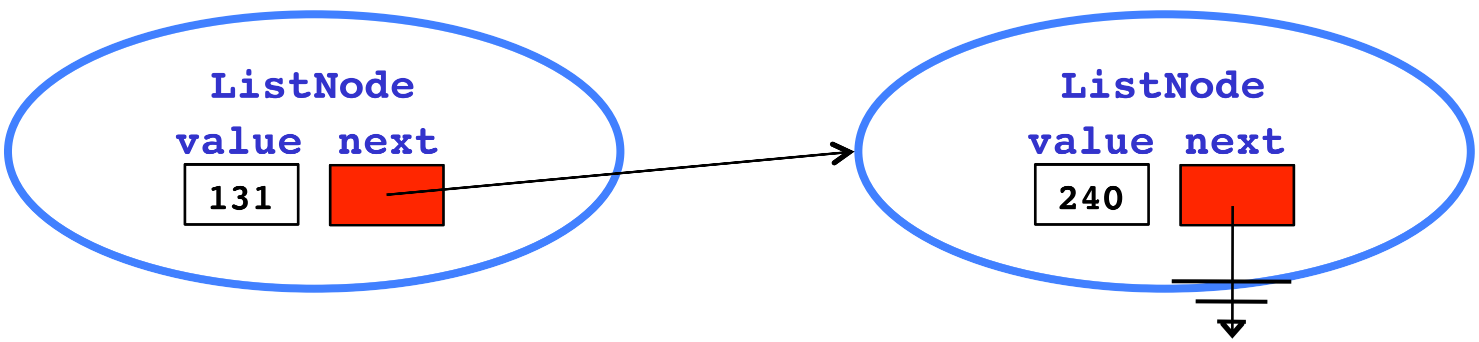
9.1 List Nodes

- **Generic form of this idea**
  - **`ListNode`**
    - has-a
      - payload
        - » Let's assume it's an `int`, named `value`
      - reference to the next node in the list



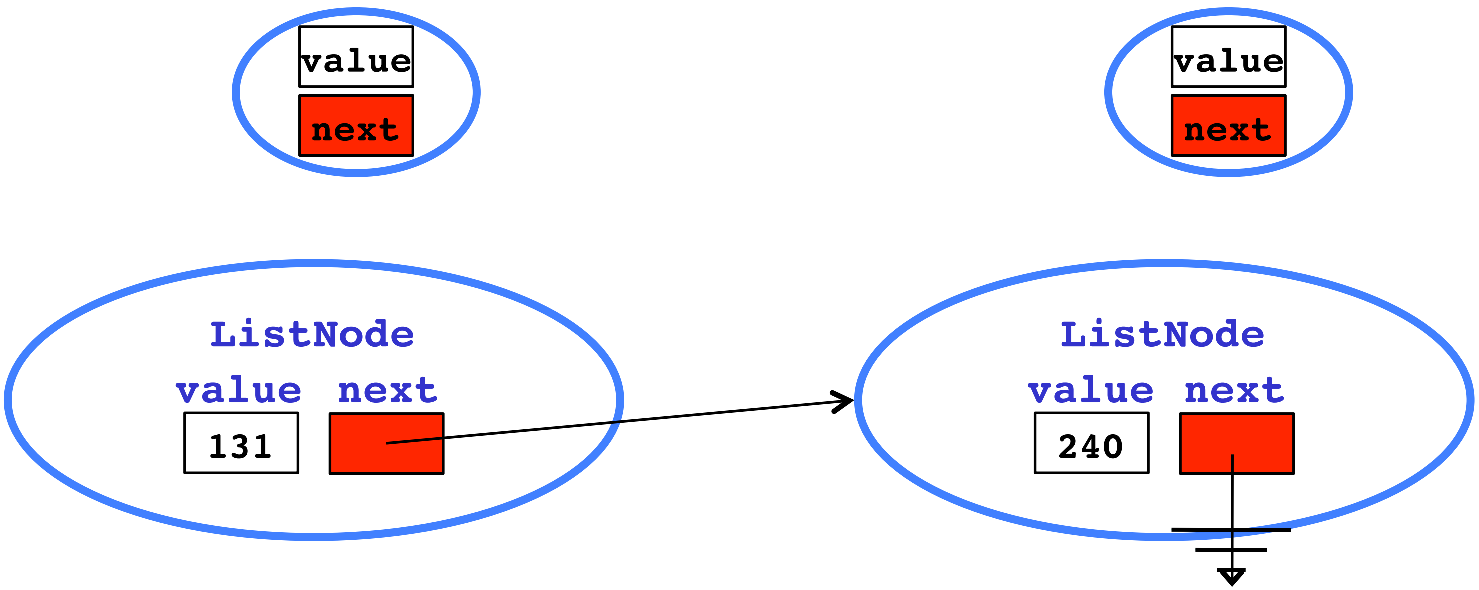
9.1 List Nodes

- **Generic form of this idea**
  - **`ListNode`**
    - has-a
      - payload
        - » Let's assume it's an `int`, named `value`
      - reference to the next node in the list
        - » Let's assume it's named `next`



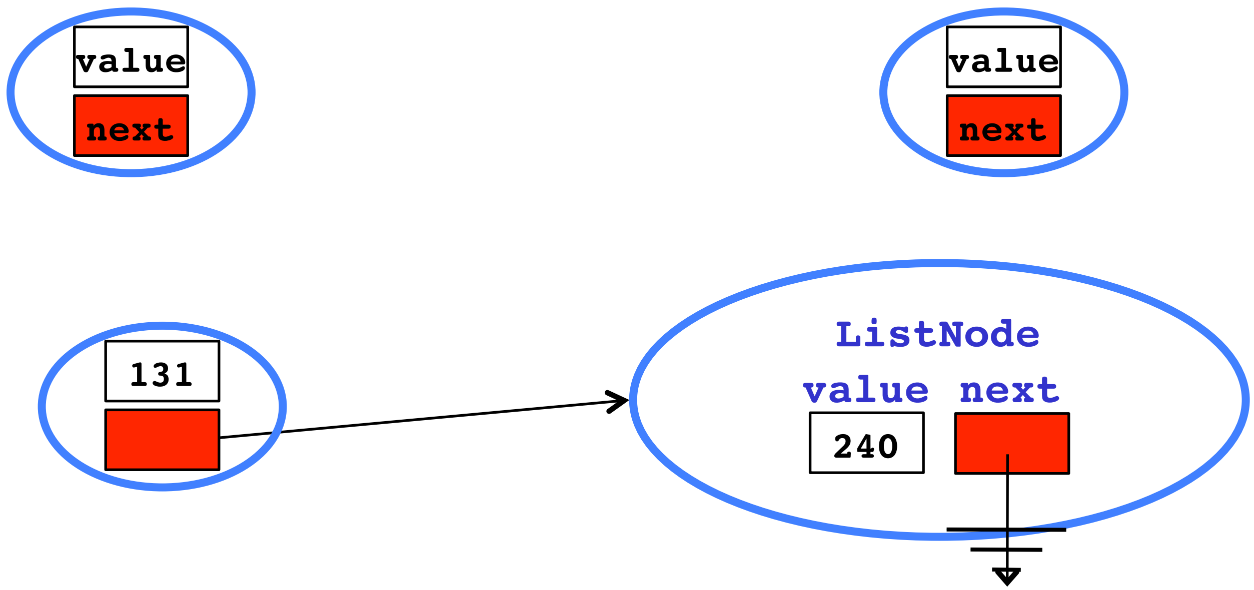
9.1 List Nodes

- **Generic form of this idea**
  - **ListNode**
    - to save space we will stack the contents vertically:



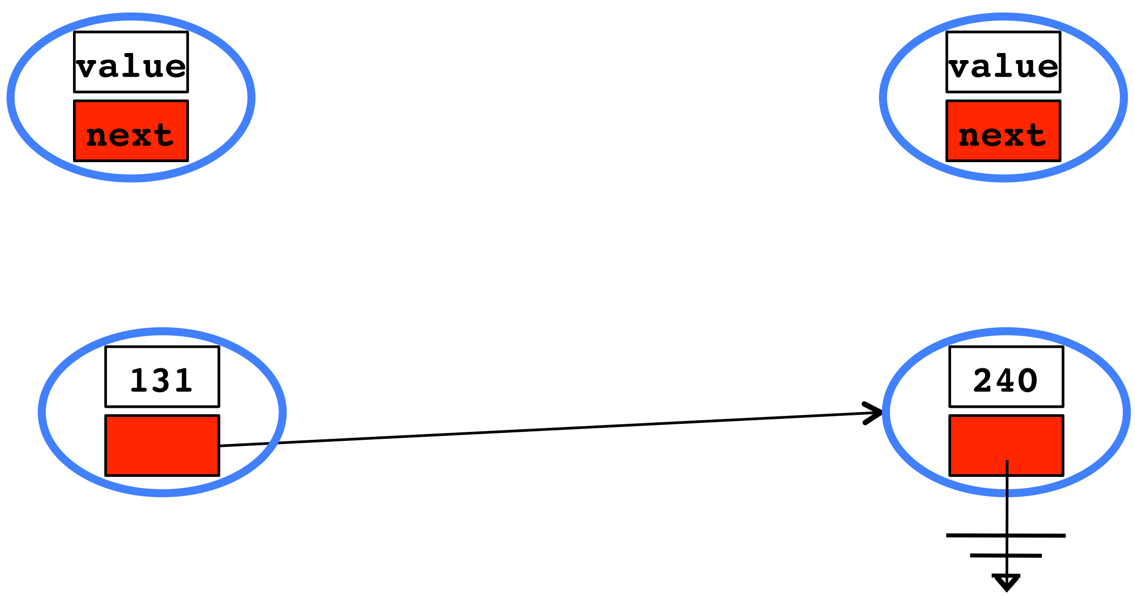
9.1 List Nodes

- **Generic form of this idea**
  - **`ListNode`**
    - to save space we will stack the contents vertically:



9.1 List Nodes

- **Generic form of this idea**
  - **`ListNode`**
    - to save space we will stack the contents vertically:



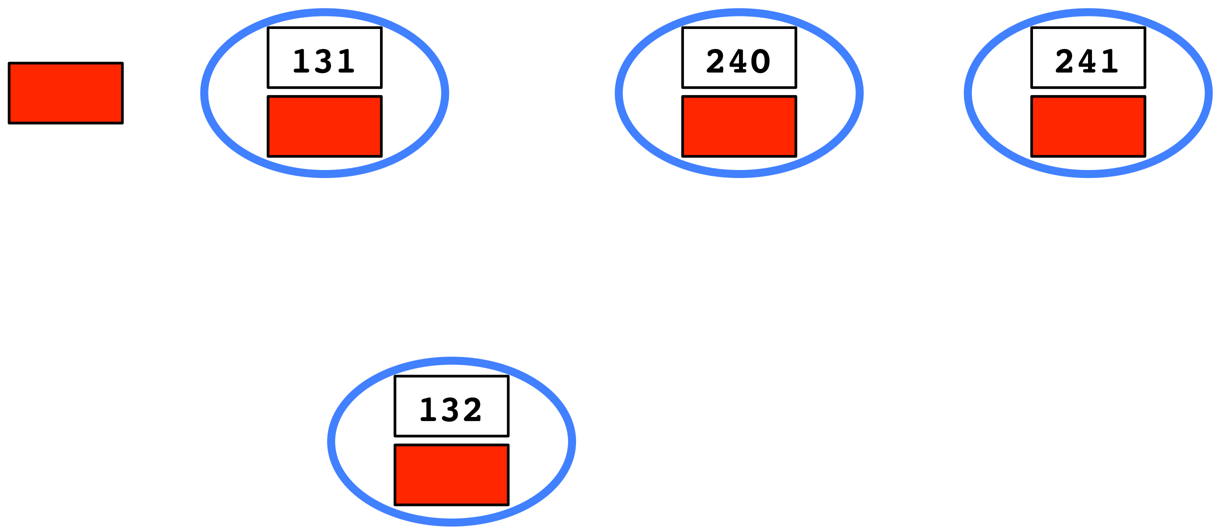
End of Oyster

9.2 Roundtable

- **What is the difference between**
  - **A person (has-a finger)**
  - **A finger pointing to a person**
  - **A `ListNode` object**
  - **A `ListNode` reference**
- **The only way to make an object**
  - **`new ListNode(….)`**
  - **creates what we show as a circle on these slides**
  - **has-a**
    - `int` value
    - `ListNode` next
- **The only way to declare a reference**
  - **`ListNode ref;`**

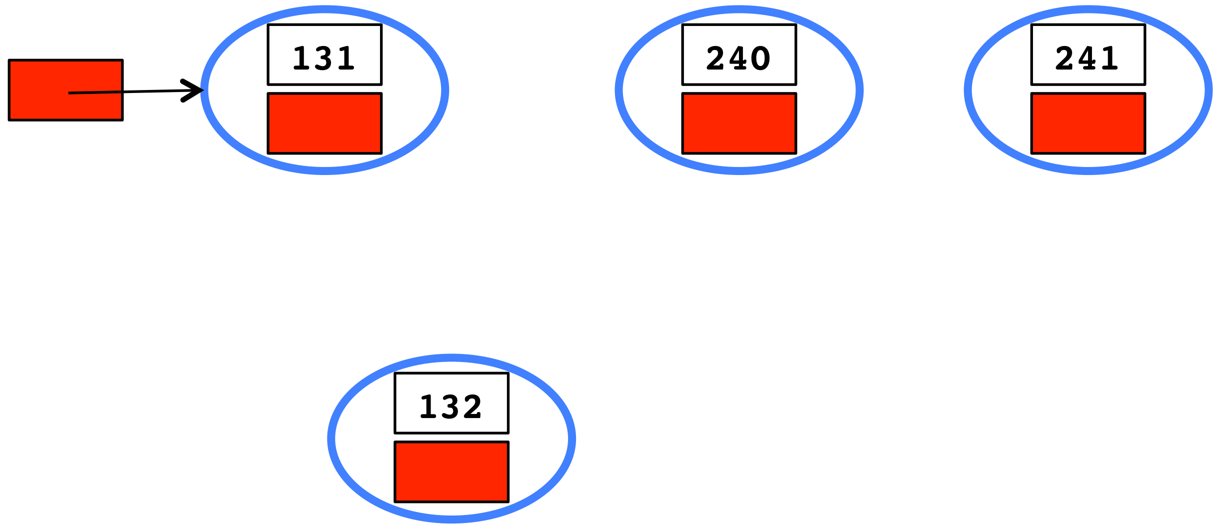
9.2 Roundtable

- How many objects do you see below?
- How many references?



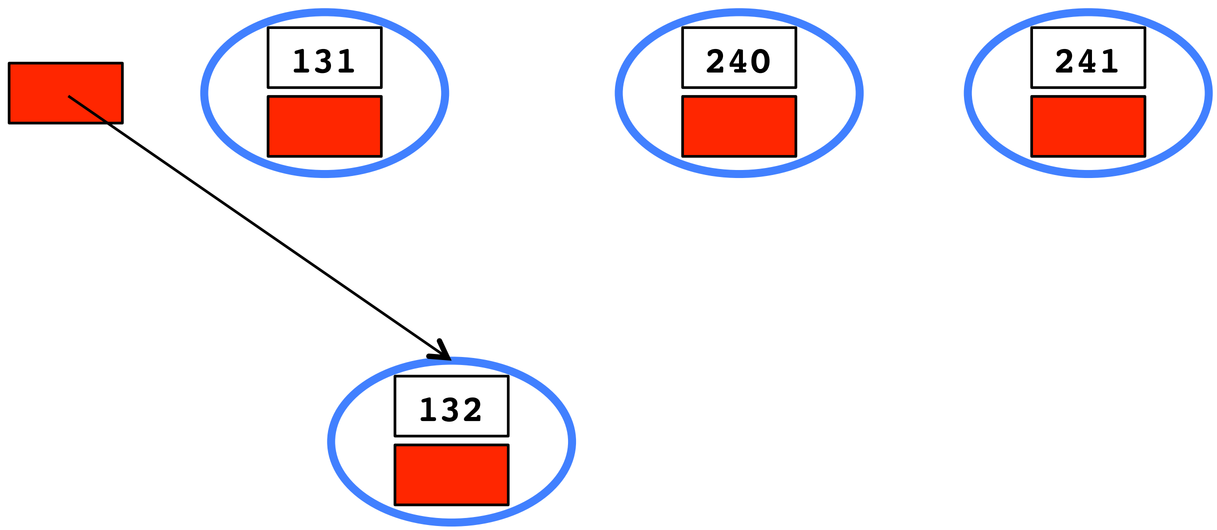
9.2 Roundtable

- **A reference**
  - **is a red box**
  - **either points at an object (a circle)**



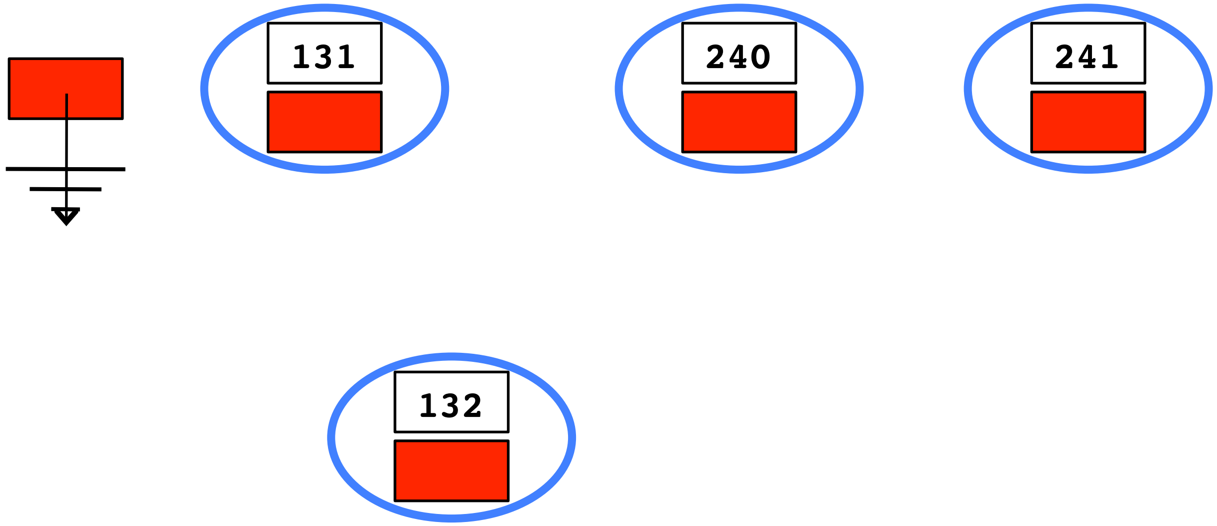
9.2 Roundtable

- **A reference**
  - **is a red box**
  - **either points at an object (a circle)**



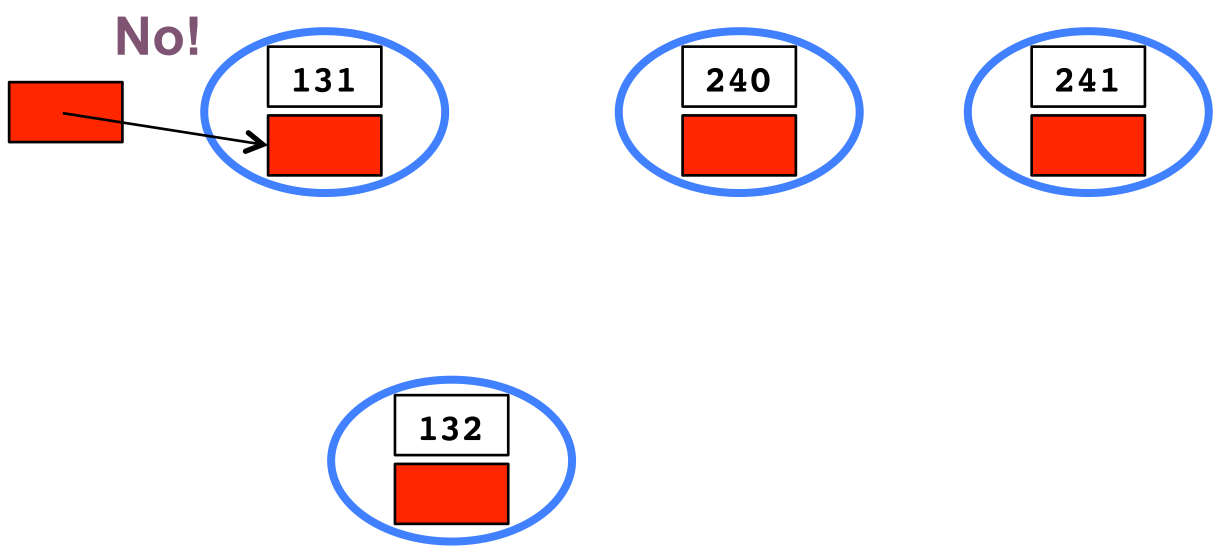
9.2 Roundtable

- **A reference**
  - **is a red box**
  - **either points at an object (a circle)**
  - **or is null**



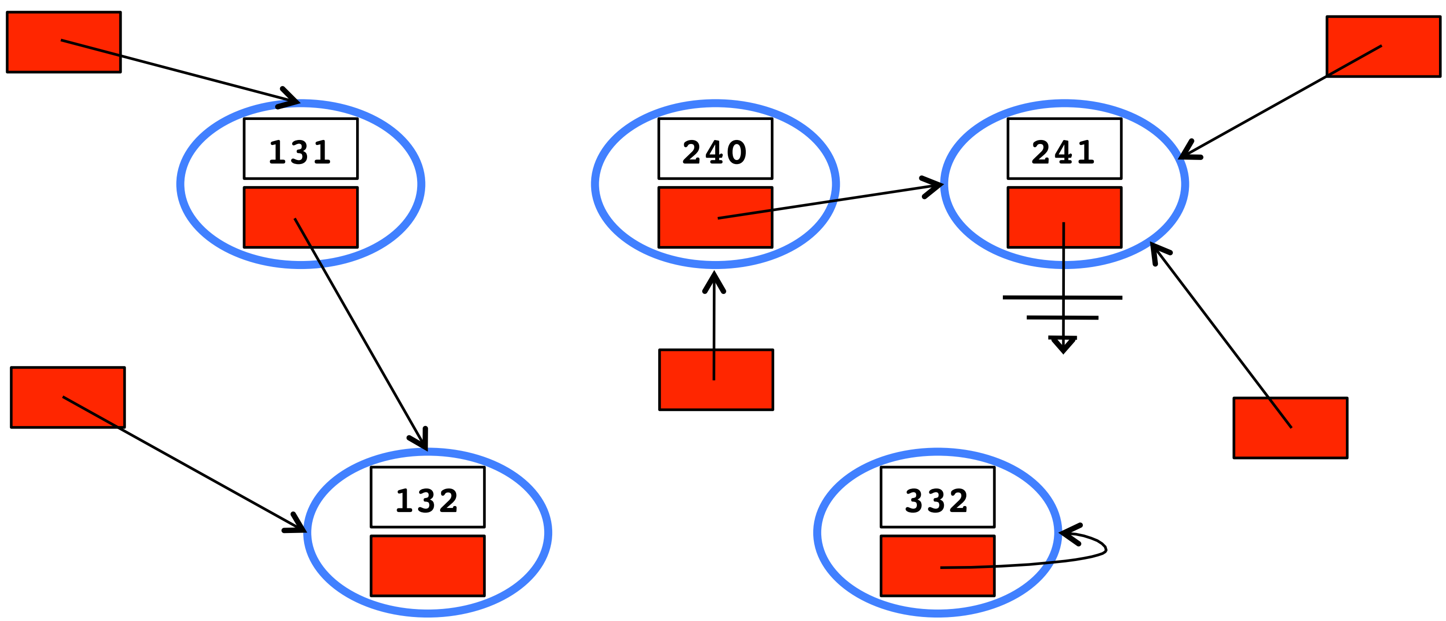
9.2 Roundtable

- **A reference**
  - **is a red box**
  - **it *never* points to another red box !**



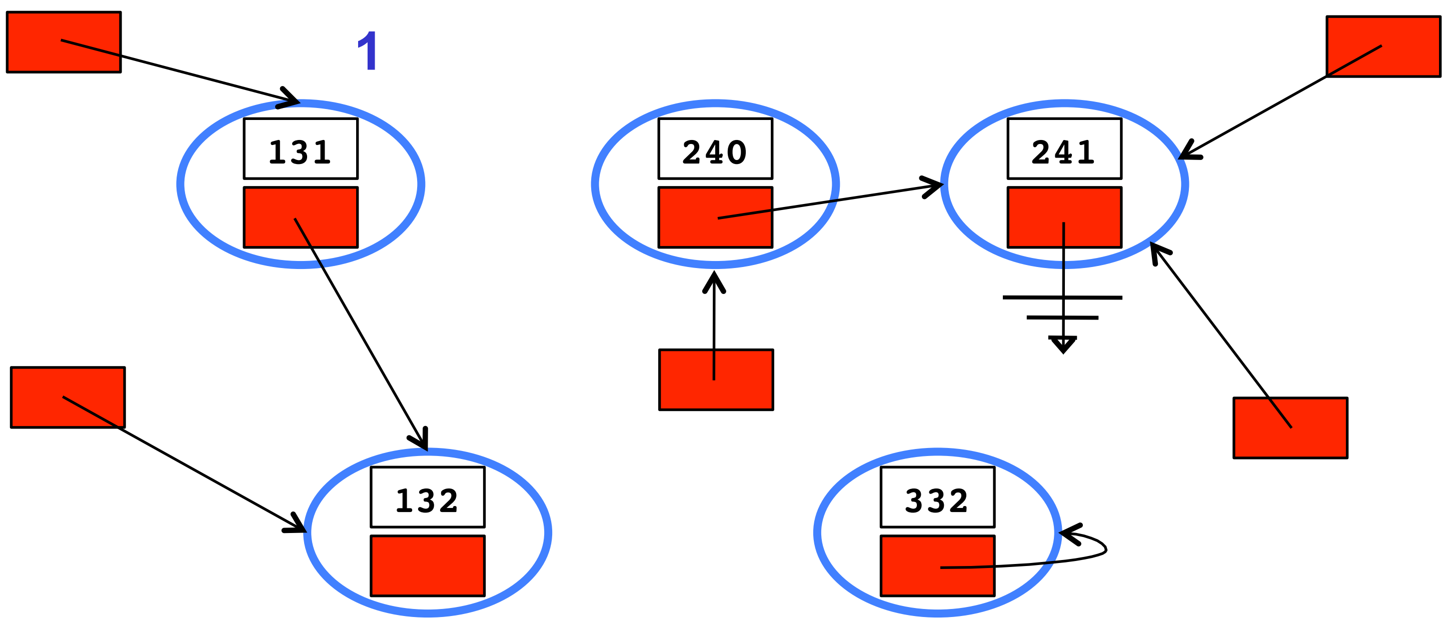
9.2 Roundtable

- **How many objects do you see below?**
- **How many references?**



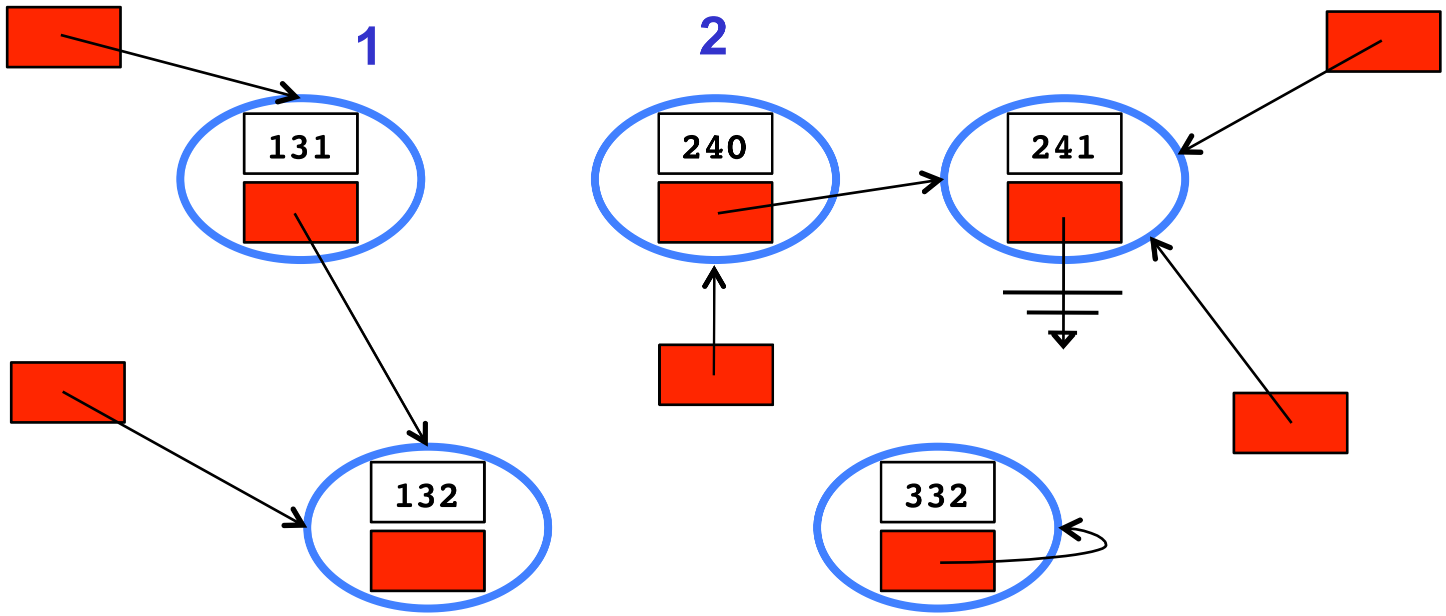
9.2 Roundtable

- **How many objects do you see below?**



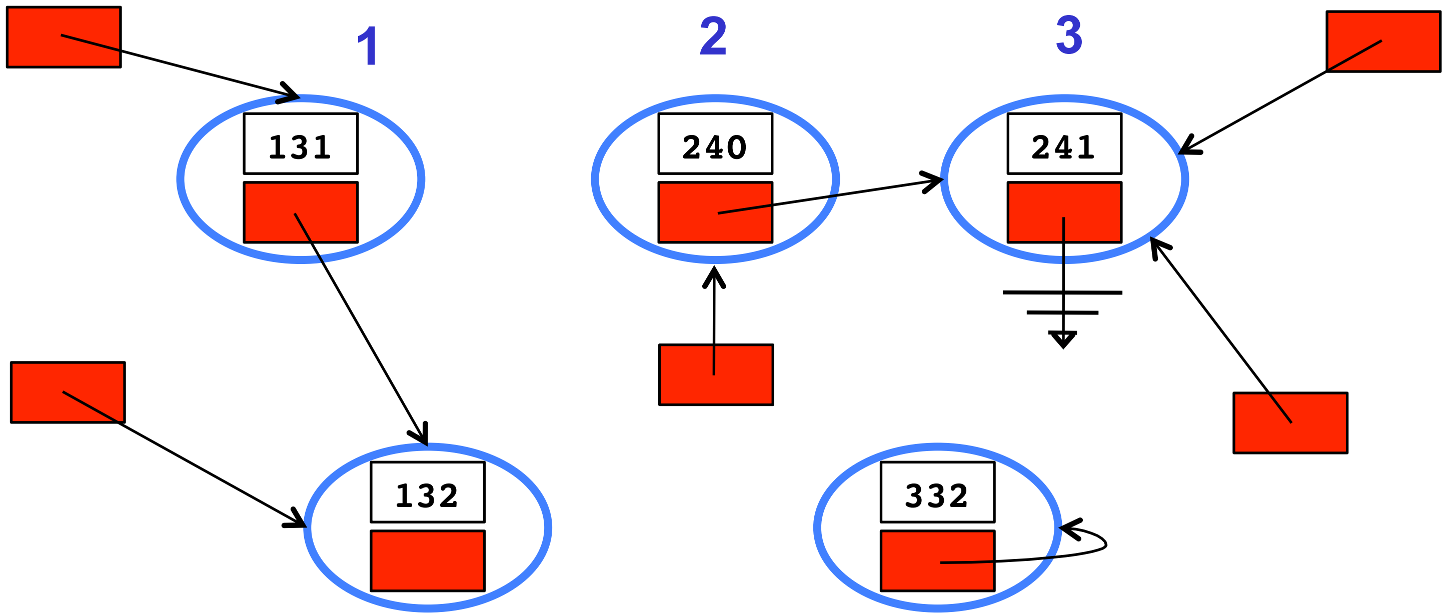
9.2 Roundtable

- **How many objects do you see below?**



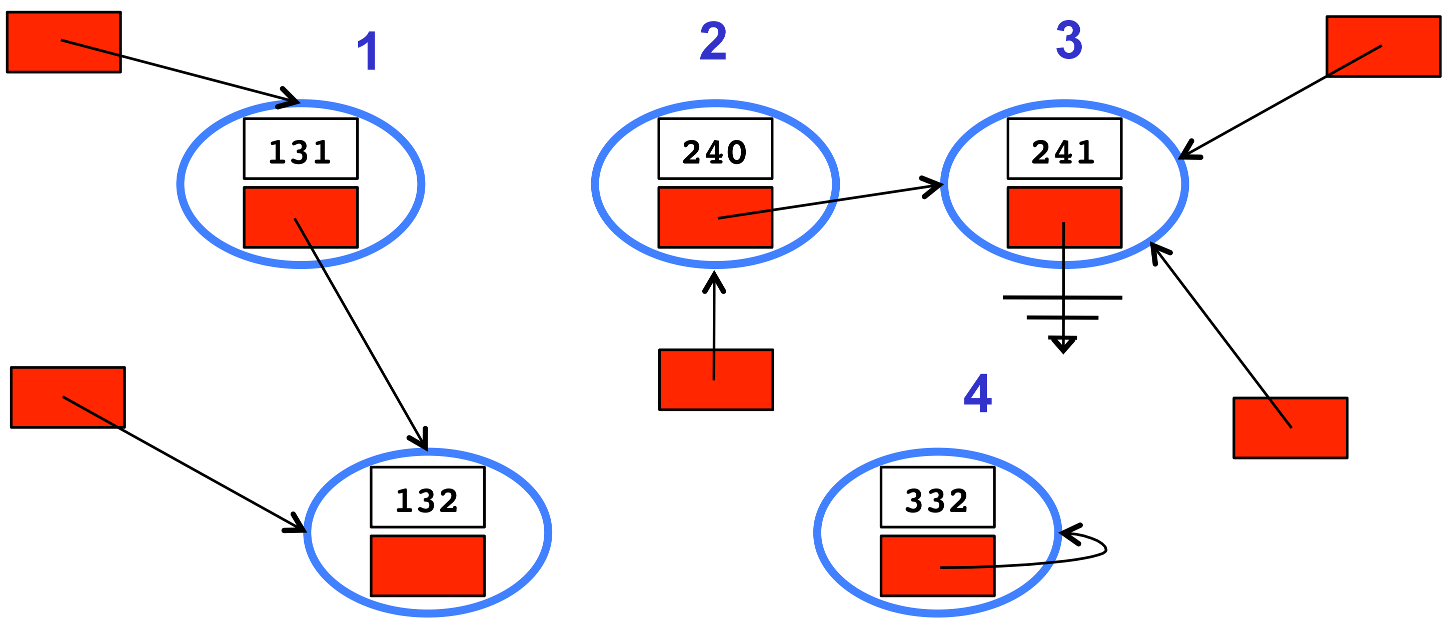
9.2 Roundtable

- **How many objects do you see below?**



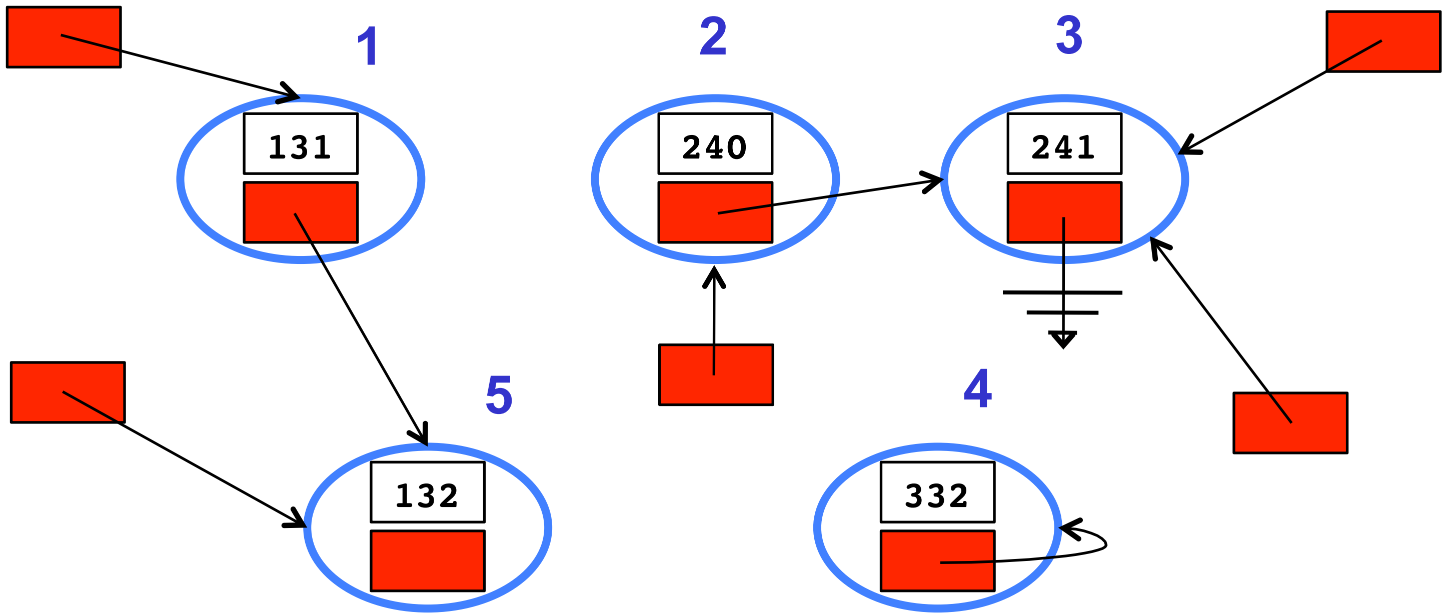
9.2 Roundtable

- **How many objects do you see below?**



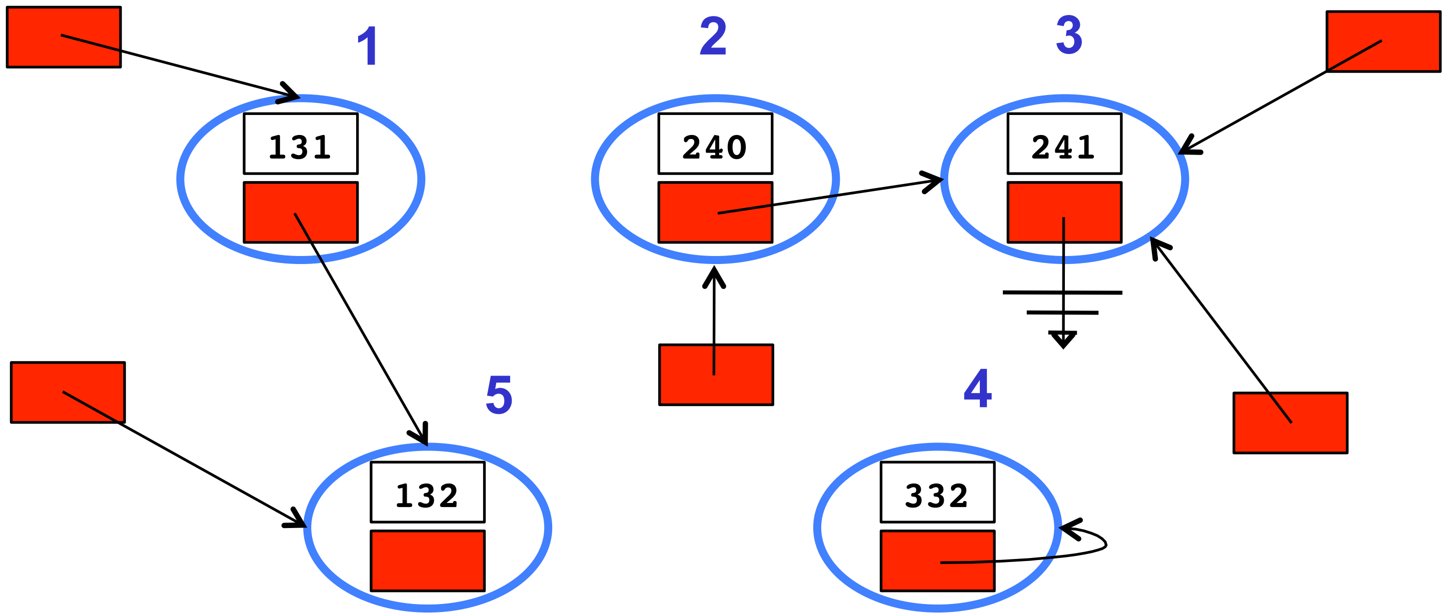
9.2 Roundtable

- How many objects do you see below?



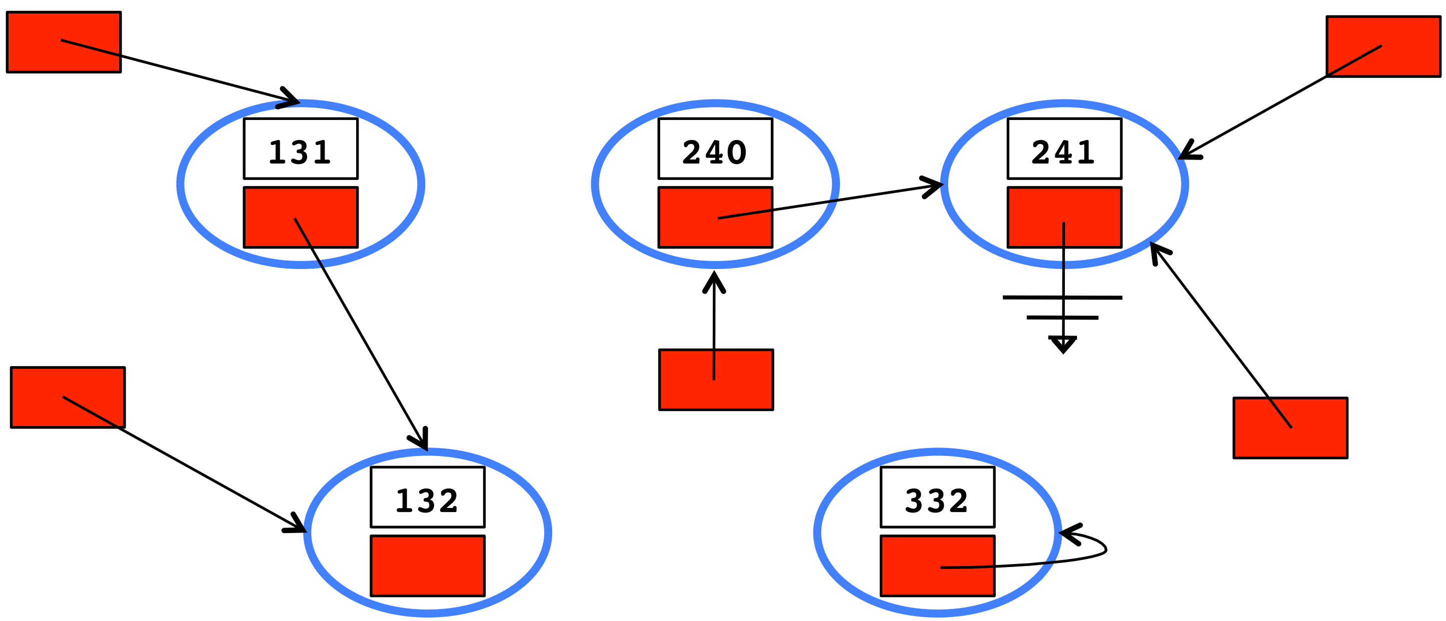
9.2 Roundtable

- **How many objects do you see below?**
  - **5 objects, each is a `ListNode`**



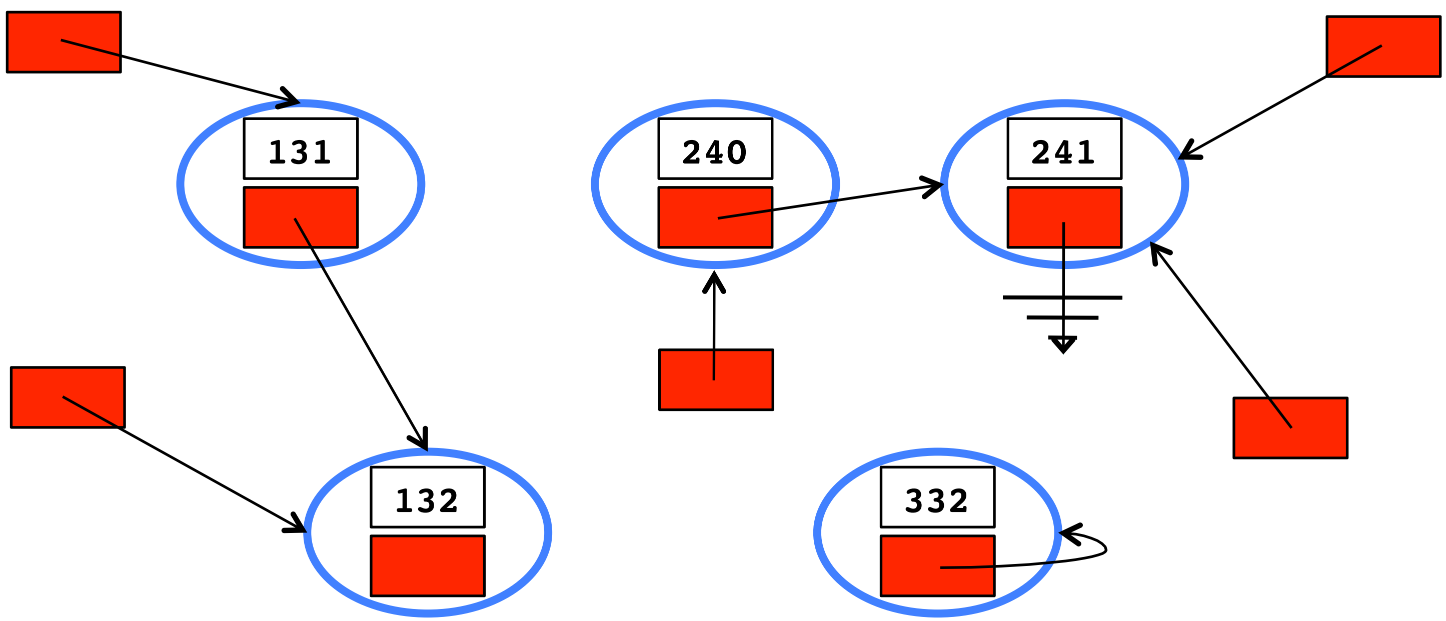
9.2 Roundtable

- How many `ListNode` references do you see?



9.2 Roundtable

- **How many `ListNode` references do you see?**
  - **10 red boxes – each is a `ListNode` reference**



End of Roundtable

Oyster 3

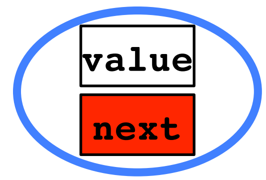
9.3 Some `ListNode` gymnastics

- We will implement `List<T>` using `ListNode`
  - But first, some exercises using `ListNode`
- We will use either
  - iteration
  - recursion
- Our methods will be either
  - static, so they accept an `ListNode` parameter
  - nonstatic, so they operate on `this ListNode`

9.3 Some `ListNode` gymnastics

- **Declaration of `ListNode`**

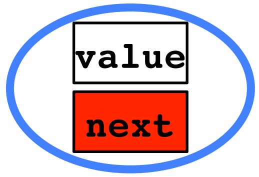
```
public class ListNode<T> {



}
```



9.3 Some `ListNode` gymnastics

- **Declaration of `ListNode`**

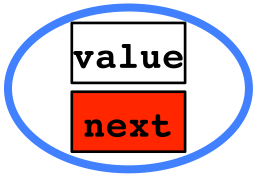
```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```



9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`

```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```

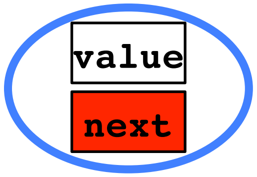


- These instance variables are missing the `private` declaration

9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`

```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```

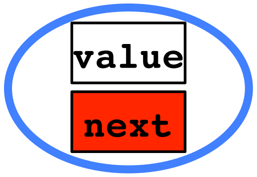


- These instance variables are missing the `private` declaration
  - The fields can be set or examined by any class in the same package as `ListNode`

9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`

```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```

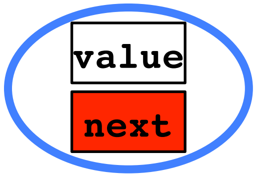


- These instance variables are missing the `private` declaration
  - The fields can be set or examined by any class in the same package as `ListNode`
  - Avoids the need for getters and setters

9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`

```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```

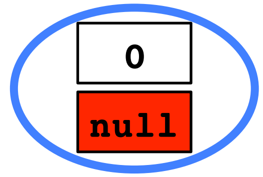


- These instance variables are missing the `private` declaration
  - The fields can be set or examined by any class in the same package as `ListNode`
  - Avoids the need for getters and setters
  - Makes the resulting code less cluttered

9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`

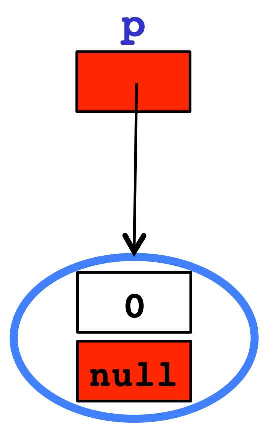
```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```



- Given
  - `new ListNode<Integer>();`

9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`

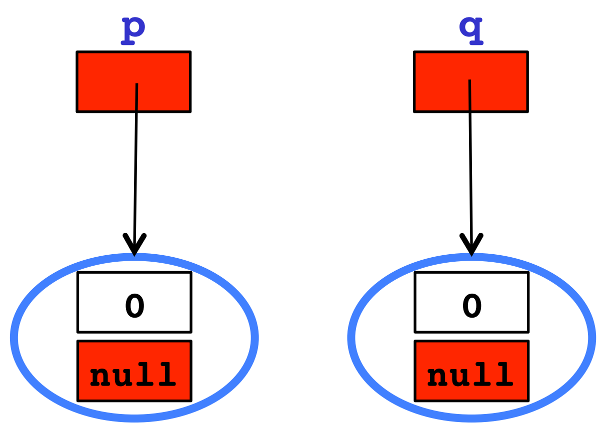



```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```

- Given
  - `ListNode<Integer> p = new ListNode<Integer>();`

9.3 Some ListNode gymnastics

- Declaration of ListNode

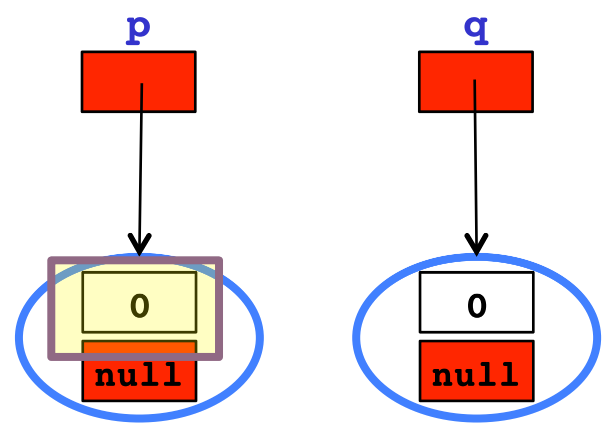



```
public class ListNode<T> {
     T value;
     ListNode<T> next;
}
```

- Given
  - `ListNode<Integer> p = new ListNode<Integer>();`
  - `ListNode<Integer> q = new ListNode<Integer>();`

9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`

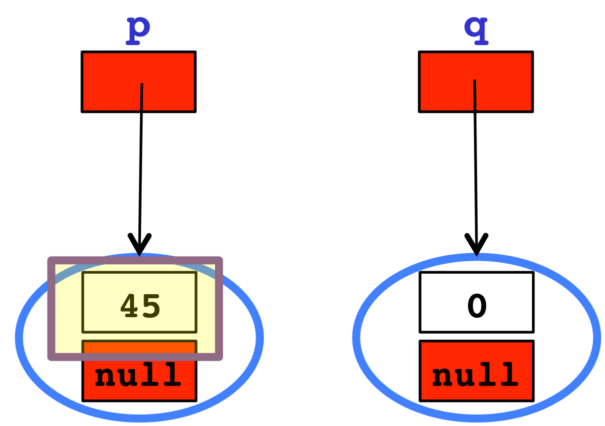



```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```

- Given
  - `ListNode<Integer> p = new ListNode<Integer>();`
  - `ListNode<Integer> q = new ListNode<Integer>();`
- We can say
  - `p.value =`

9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`

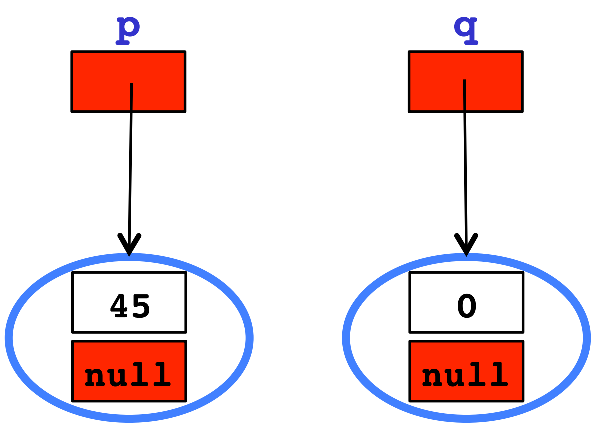



```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```

- Given
  - `ListNode<Integer> p = new ListNode<Integer>();`
  - `ListNode<Integer> q = new ListNode<Integer>();`
- We can say
  - `p.value = 45;`

9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`

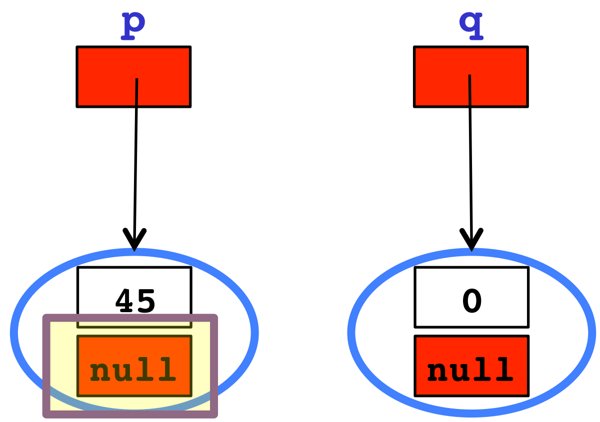



```
public class ListNode<T> {
     T value;
     ListNode<T> next;
}
```

- Given
  - `ListNode<Integer> p = new ListNode<Integer>();`
  - `ListNode<Integer> q = new ListNode<Integer>();`
- We can say
  - `p.value = 45;  // p cannot be null`

9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`

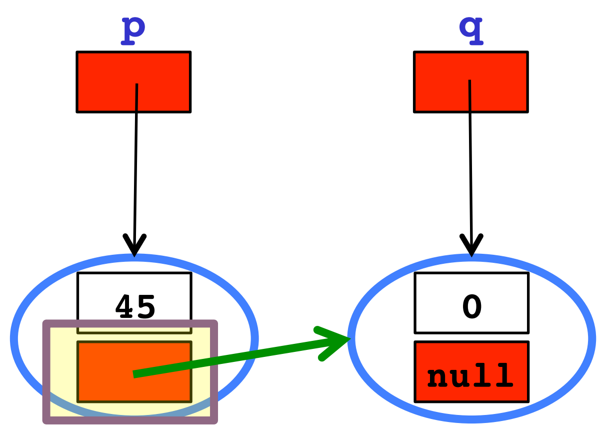



```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```

- Given
  - `ListNode<Integer> p = new ListNode<Integer>();`
  - `ListNode<Integer> q = new ListNode<Integer>();`
- We can say
  - `p.value = 45;  // p cannot be null`
  - `p.next  = q;`

9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`

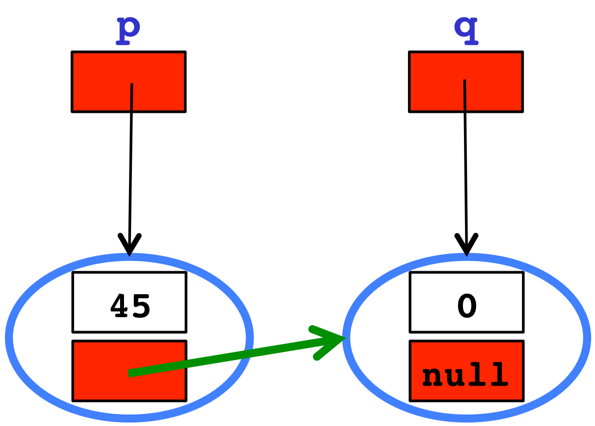



```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```

- Given
  - `ListNode<Integer> p = new ListNode<Integer>();`
  - `ListNode<Integer> q = new ListNode<Integer>();`
- We can say
  - `p.value = 45;  // p cannot be null`
  - `p.next  = q;`

9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`

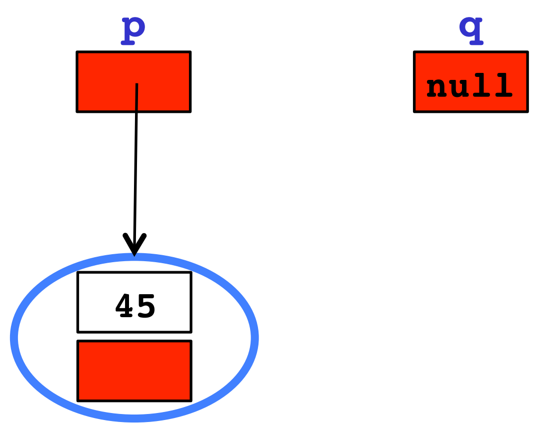



```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```

- Given
  - `ListNode<Integer> p = new ListNode<Integer>();`
  - `ListNode<Integer> q = new ListNode<Integer>();`
- We can say
  - `p.value = 45;  // p cannot be null`
  - `p.next  = q;`

9.3 Some ListNode gymnastics

- Declaration of ListNode

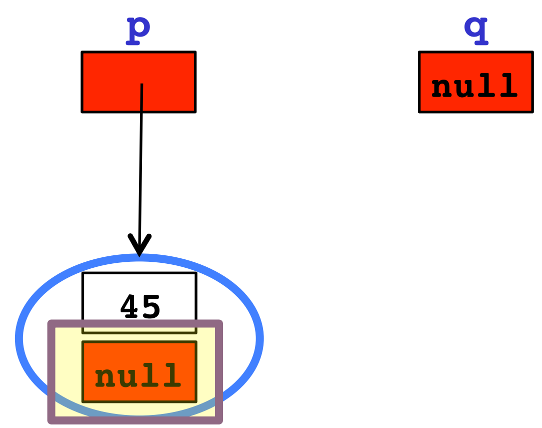



```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```

- Given
  - ```
    ListNode<Integer> p = new ListNode<Integer>();
    ```
  - ```
    ListNode<Integer> q = new ListNode<Integer>();
    ```
- We can say
  - ```
    p.value = 45;  // p cannot be null
    ```
  - ```
    p.next  = q;   // q could be null here
    ```

9.3 Some ListNode gymnastics

- Declaration of ListNode

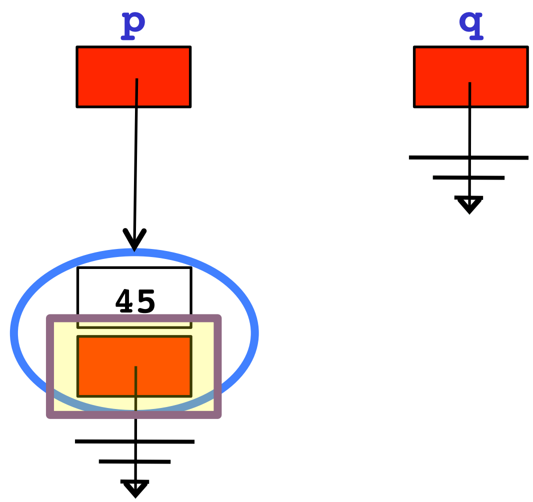



```
public class ListNode<T> {
     T value;
     ListNode<T> next;
}
```

- Given
  - ListNode<Integer> p = new ListNode<Integer>();
  - ListNode<Integer> q = new ListNode<Integer>();
- We can say
  - p.value = 45;   // p cannot be null
  - p.next = q;   // q could be null here

9.3 Some `ListNode` gymnastics

- Declaration of `ListNode`



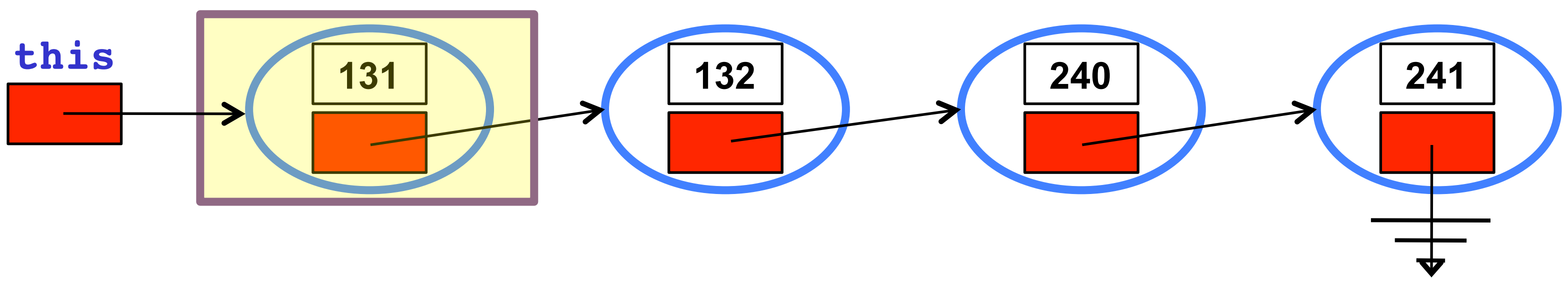

```
public class ListNode<T> {
    T value;
    ListNode<T> next;
}
```

- Given
  - `ListNode<Integer> p = new ListNode<Integer>();`
  - `ListNode<Integer> q = new ListNode<Integer>();`
- We can say
  - `p.value = 45;   // p cannot be null`
  - `p.next  = q;    // q could be null here`

9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

```
System.out.println(
   this
);
```

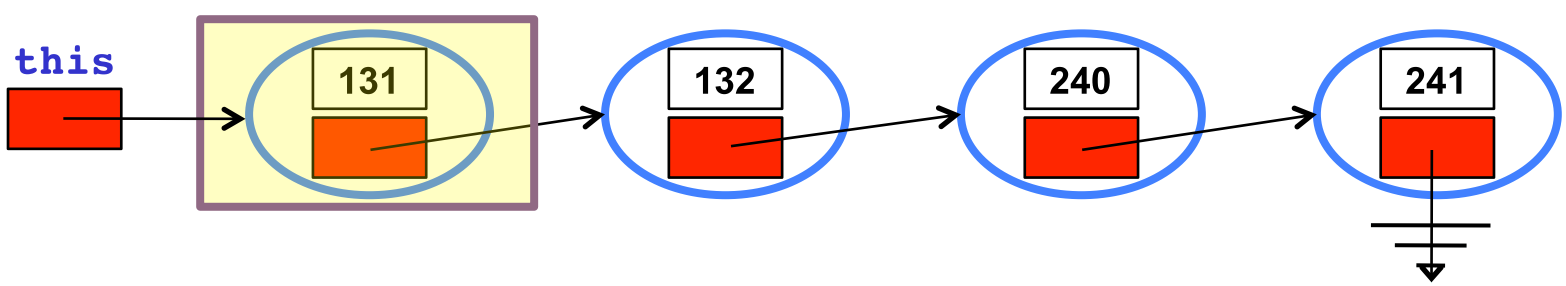


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

```
System.out.println(
   this
);
```

The results of this depend on the implementation of `toString()` within `ListNode`



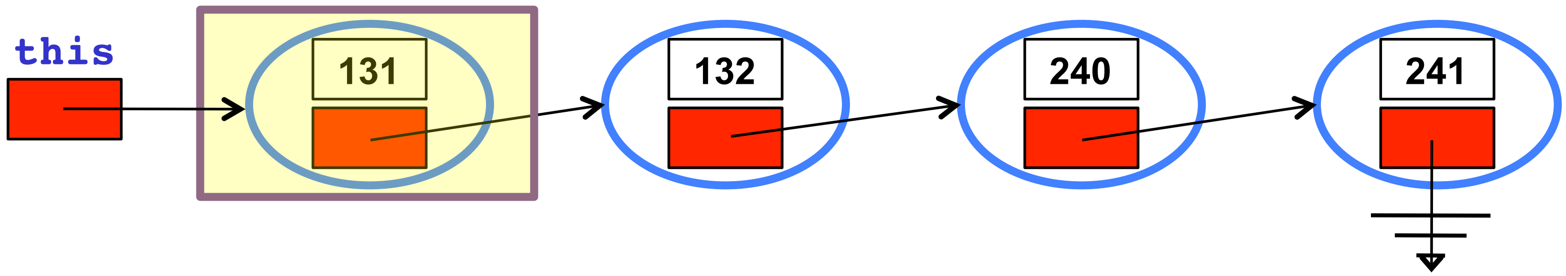
9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

```
System.out.println(
   this
);
```

The results of this depend on the implementation of `toString()` within `ListNode`

`"A ListNode"`



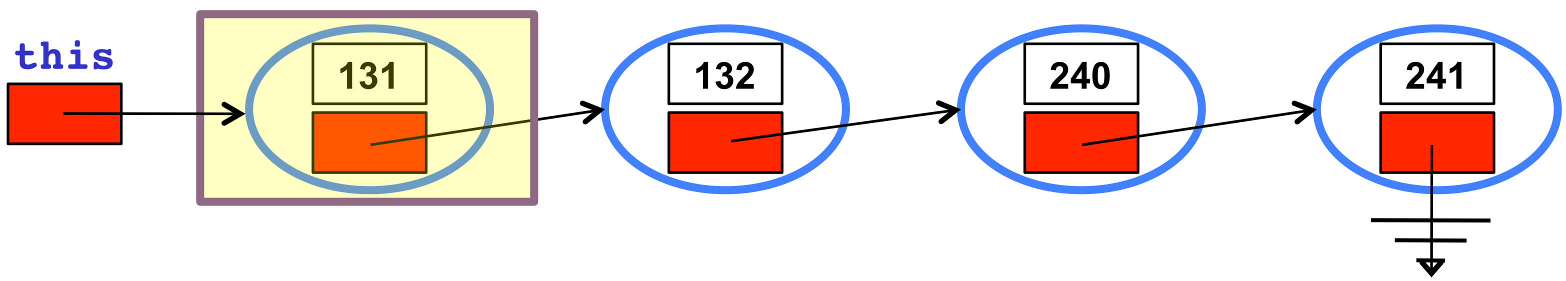
9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

```
System.out.println(
   this
);
```

The results of this depend on the implementation of `toString()` within `ListNode`

```
"A ListNode with 131"
```

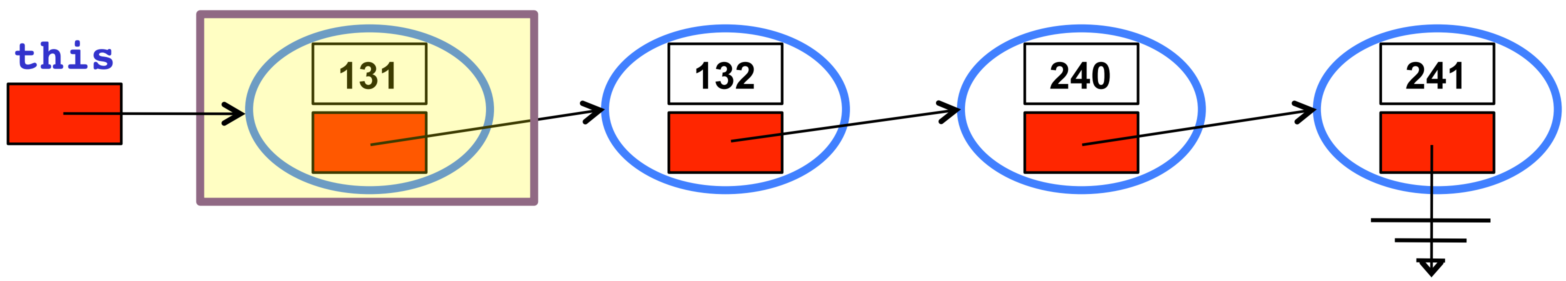


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

```
System.out.println(
   this
);
```

The results of this depend on the implementation of `toString()` within `ListNode`

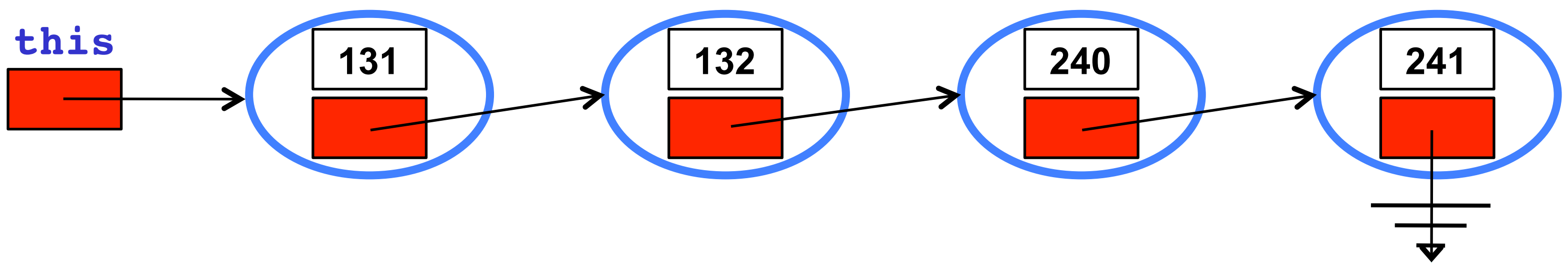


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

```
System.out.println(
   this.value
);
```

131

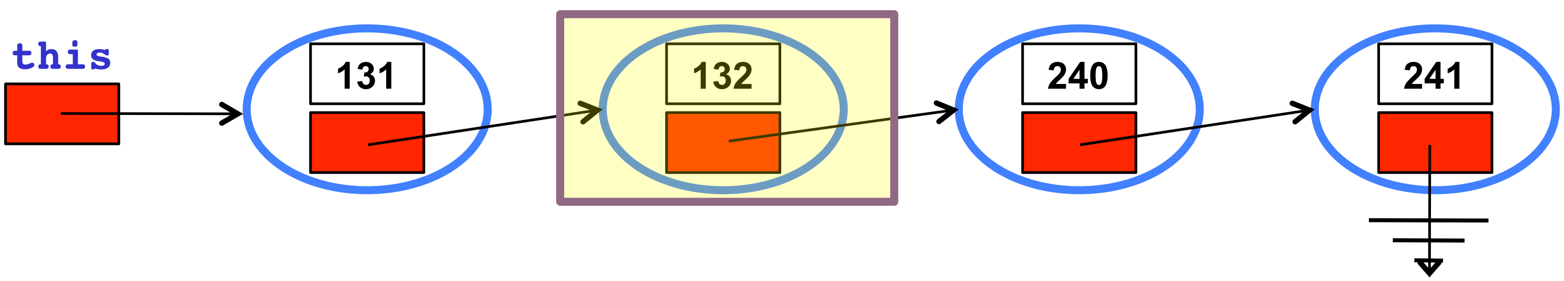


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

```
System.out.println(
   this.next
);
```

"A ListNode with 132"

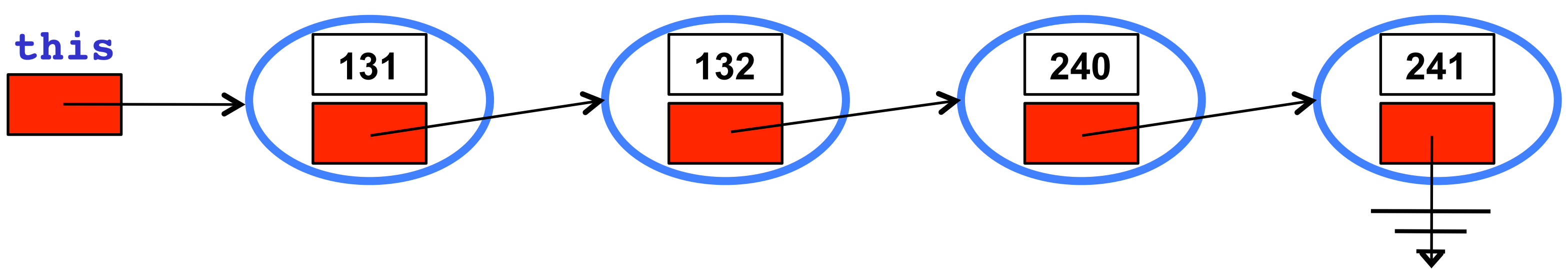


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;


}
```

```
System.out.println(
   this.next.value
);
```

132

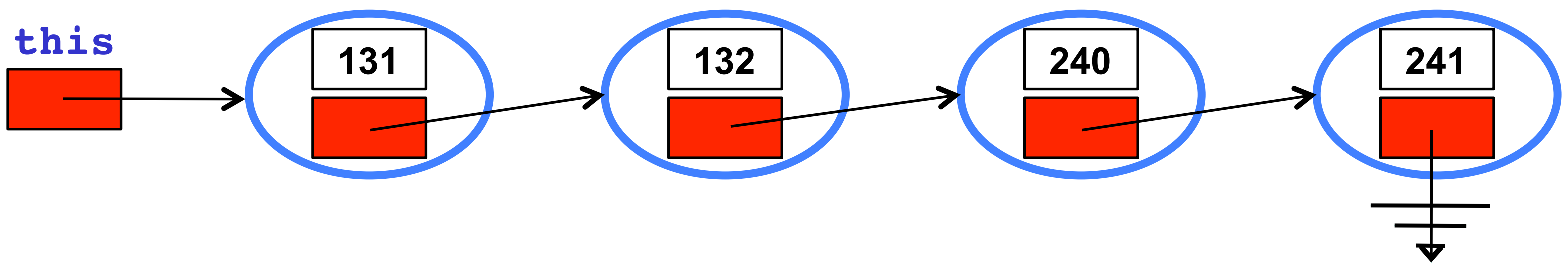


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;


}
```

```
System.out.println(
    this.next.next.next.value
);
```

241

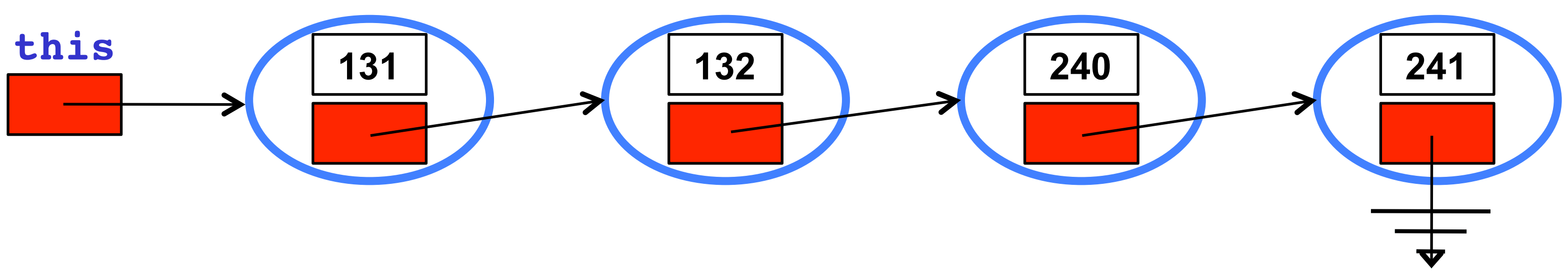


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;




}
```

```
System.out.println(
    this.next.next.next.next
);
```

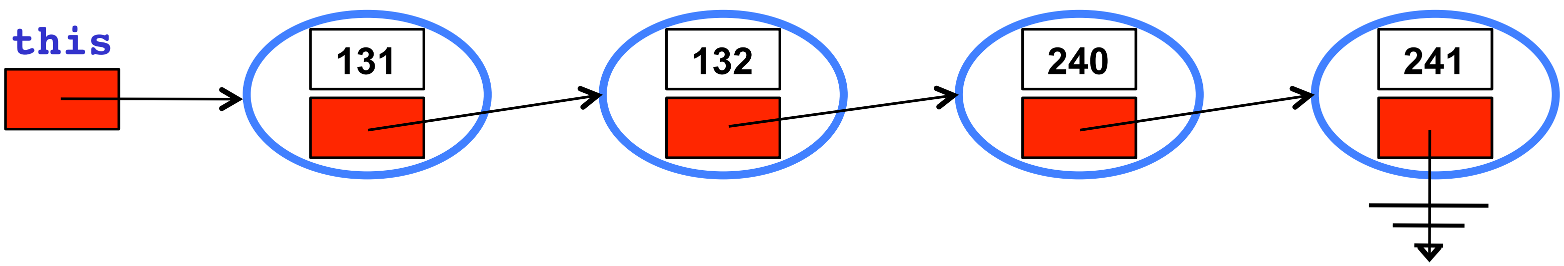
`null`



9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

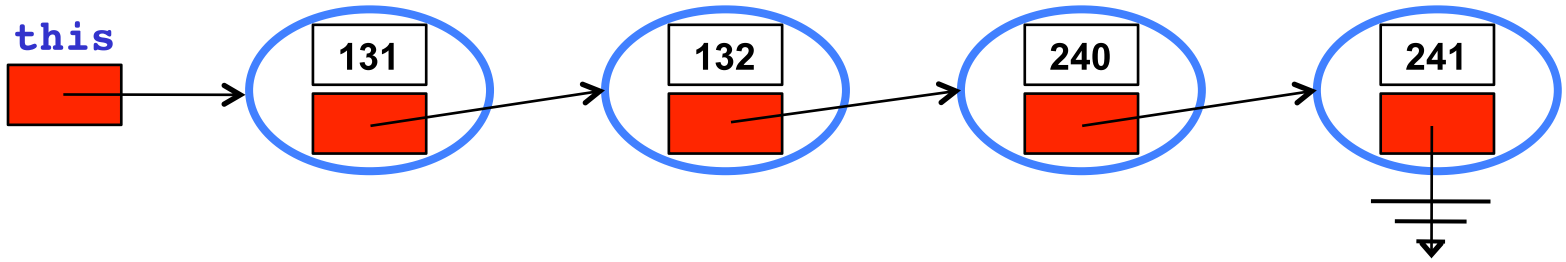
Now let's try changing the
values and structure of the list



9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

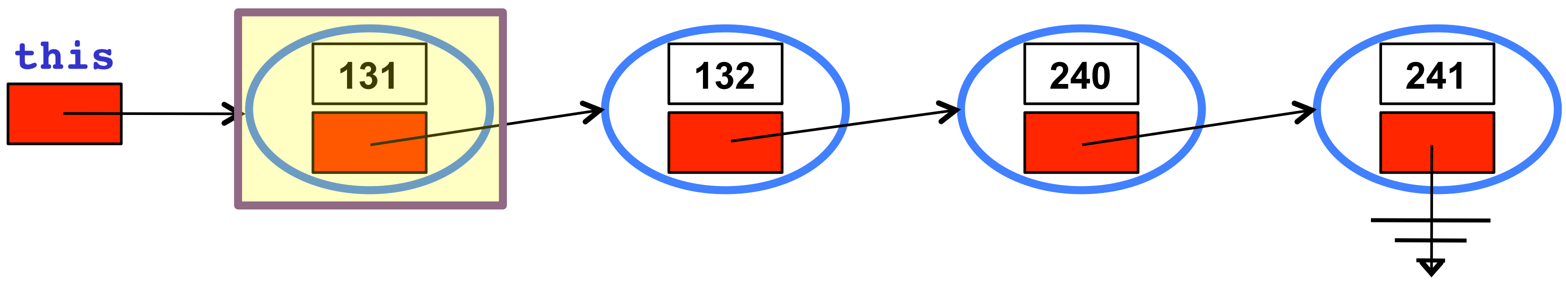
```
this.value = 100;
```



9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

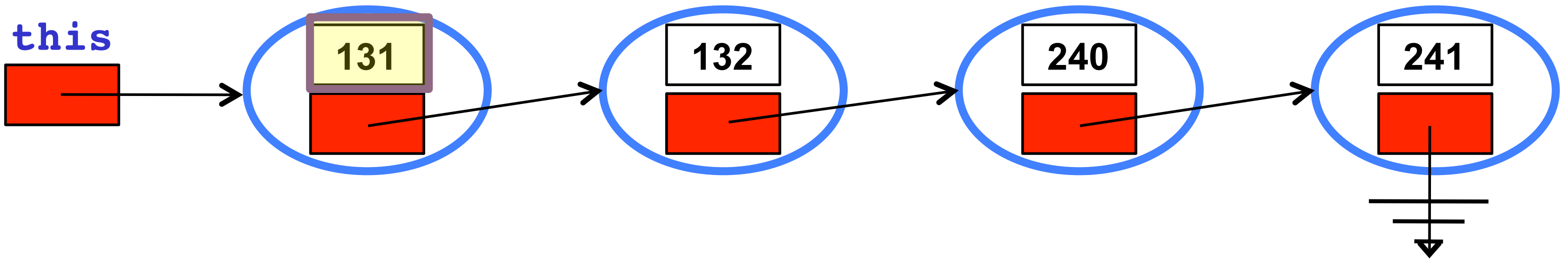
```
this.value = 100;
```



9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

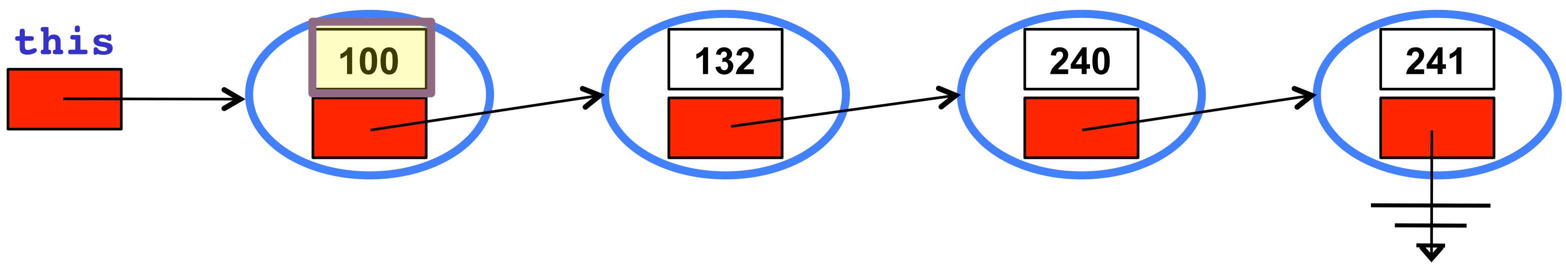
```
this.value = 100;
```



9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

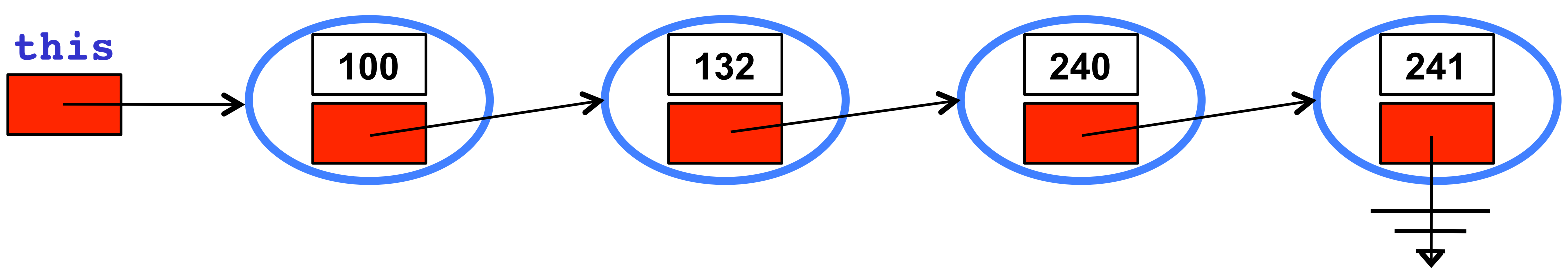
```
this.value = 100;
```



9.3 Some ListNode gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

}
```

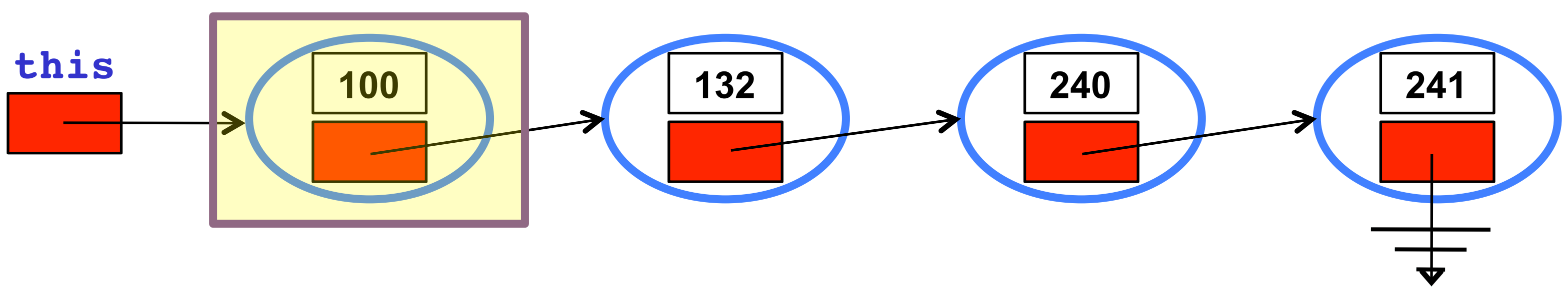
```
this.next.next.next.value = 101;
```



9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

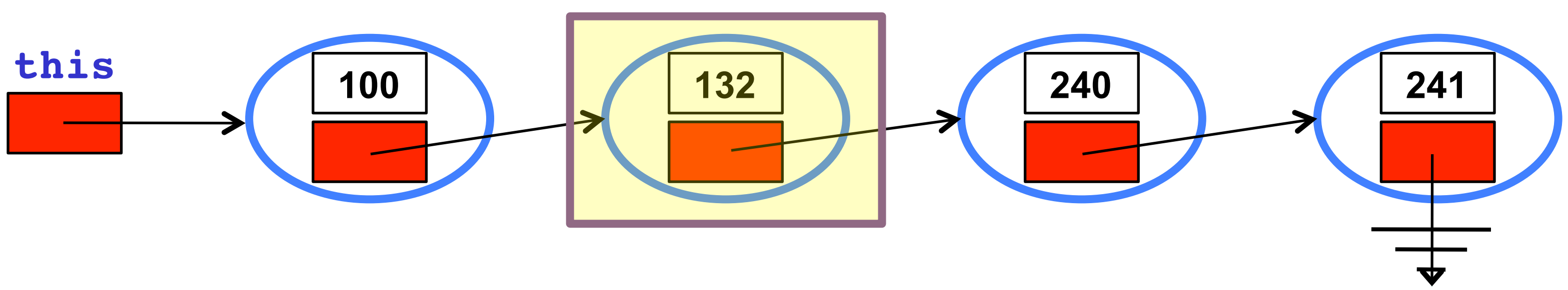
```
this.next.next.next.value = 101;
```



9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

`this.next.next.next.value = 101;`

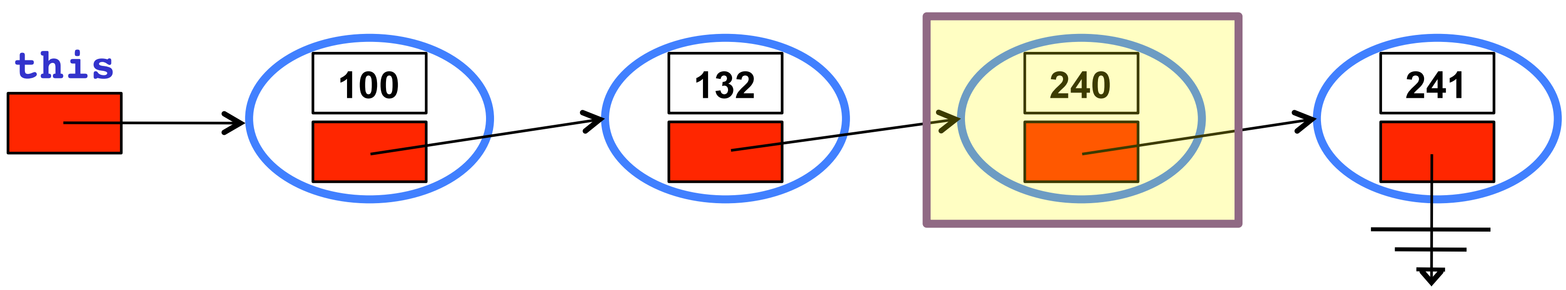


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;
```

```
this.next.next.next.value = 101;
```

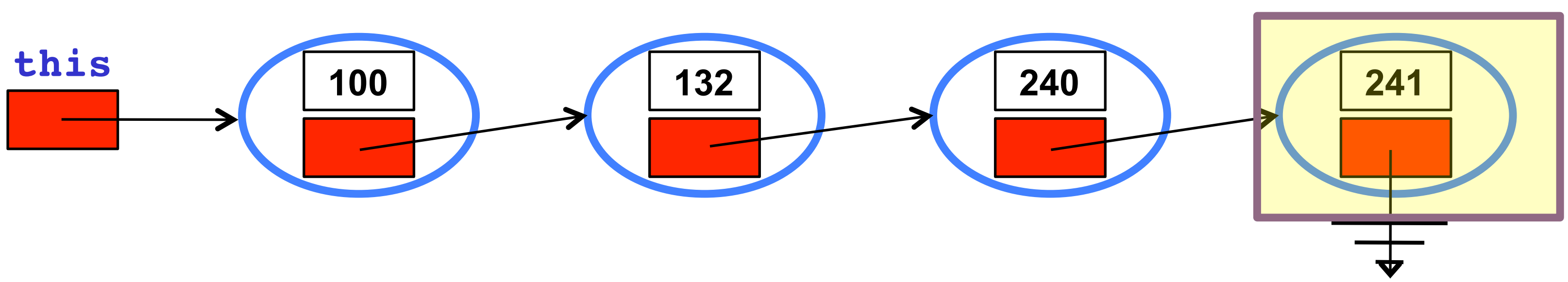
```
}
```



9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;
}
```

```
this.next.next.next.value = 101;
```

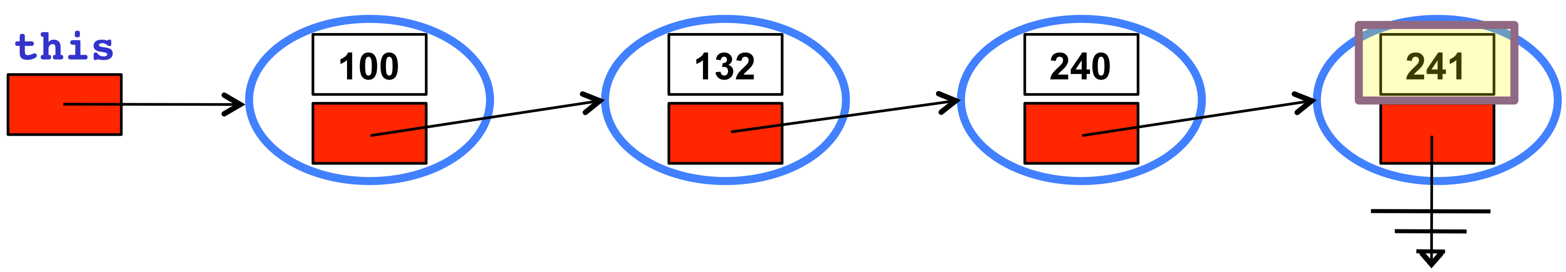


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;
```

`this.next.next.next.value = 101;`

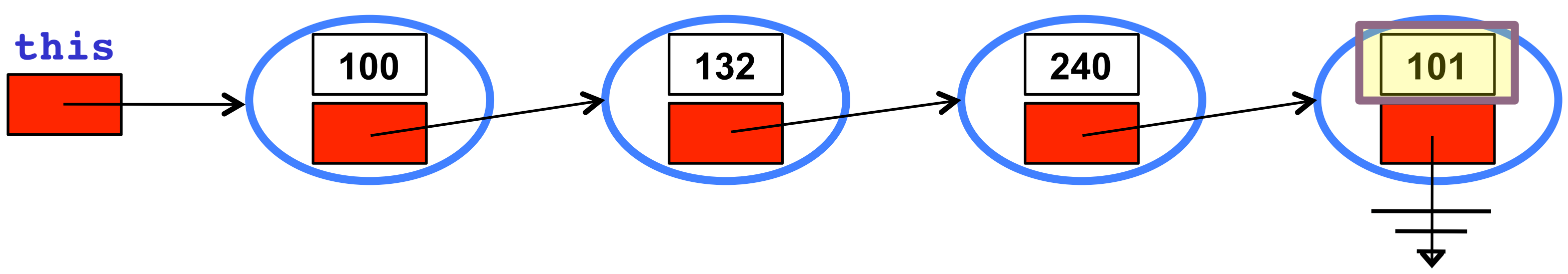
```
}
```



9.3 Some `ListNode` gymnastics

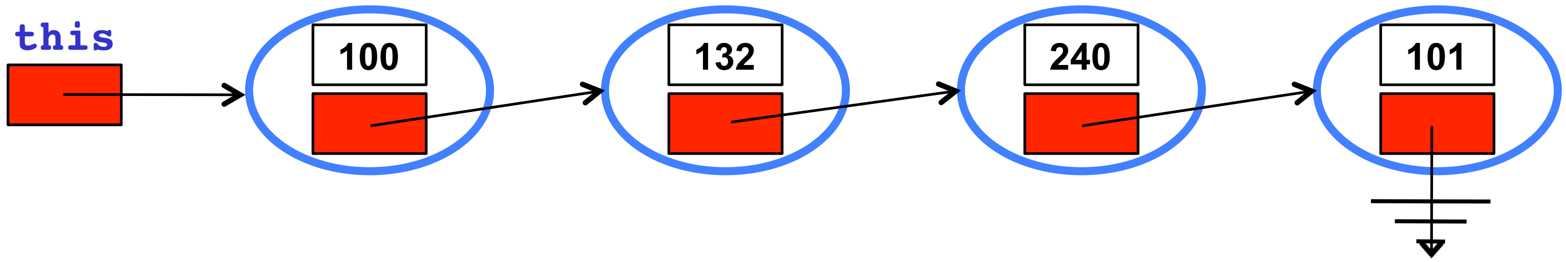
```
public class ListNode<T> {
      T value;
      ListNode<T> next;

}
```

```
this.next.next.next.value = 101;
```



9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;




}
```

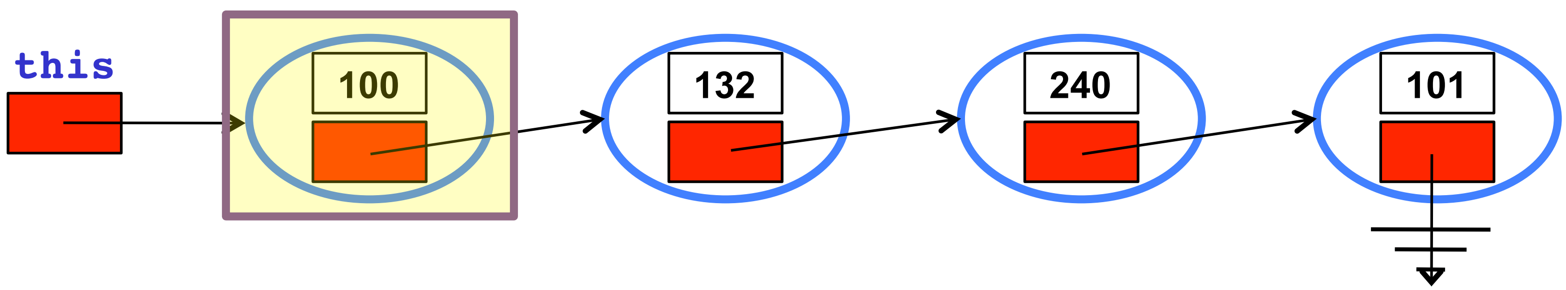


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;
```

```
this.next =
```

```
}
```

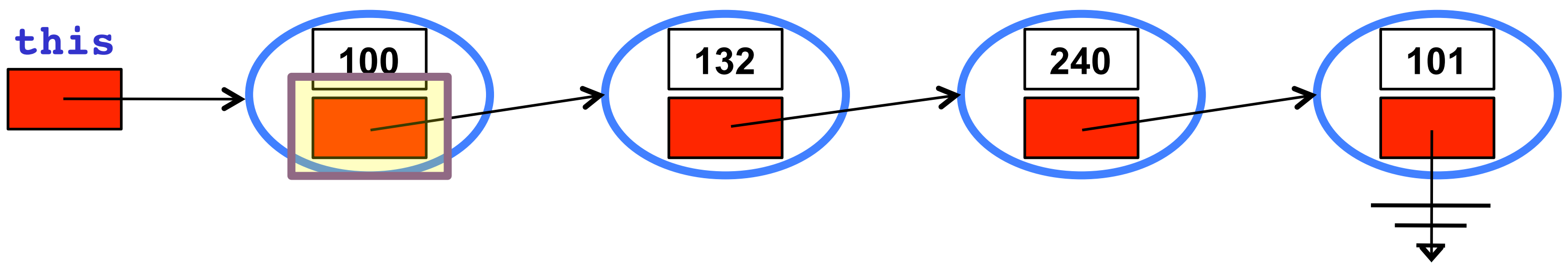


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;
```

this.next =

```
}
```

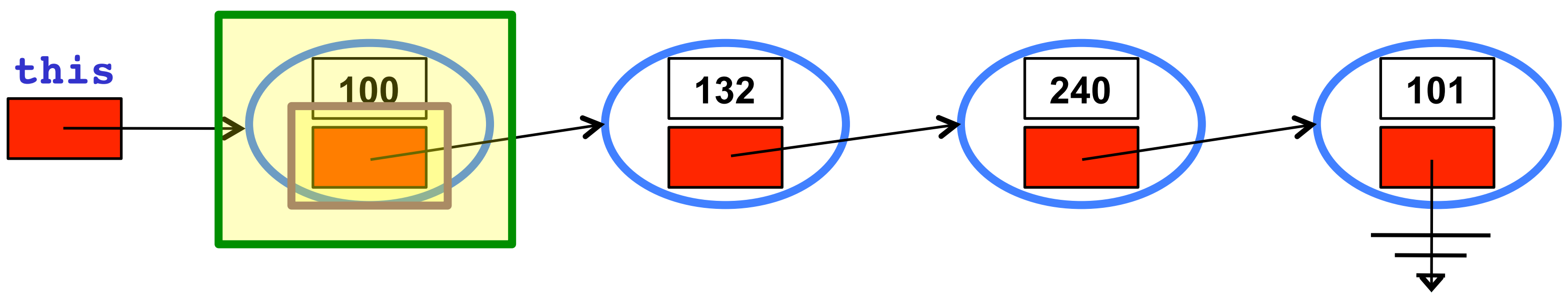


9.3 Some ListNode gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;
```

```
this.next = this.next.next;
```

```
}
```

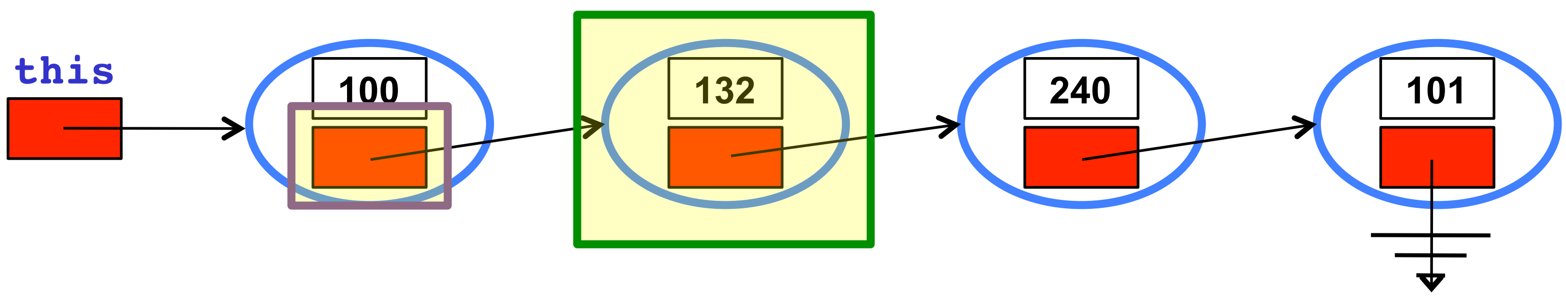


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;
```

```
this.next = this.next.next;
```

```
}
```

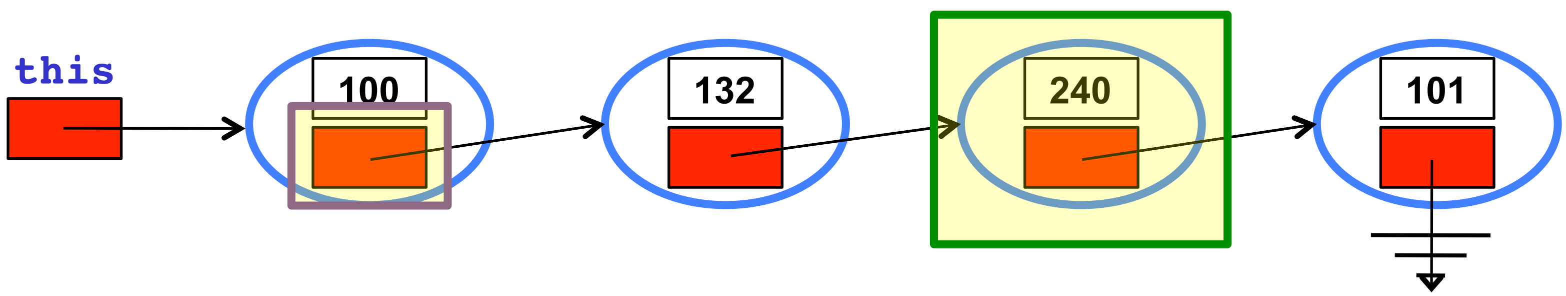


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;
```

```
this.next = this.next.next;
```

```
}
```

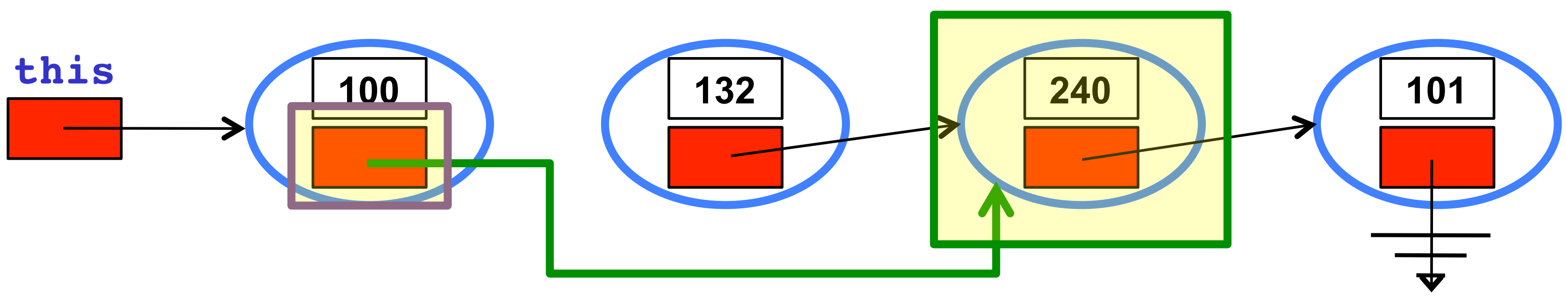


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;
```

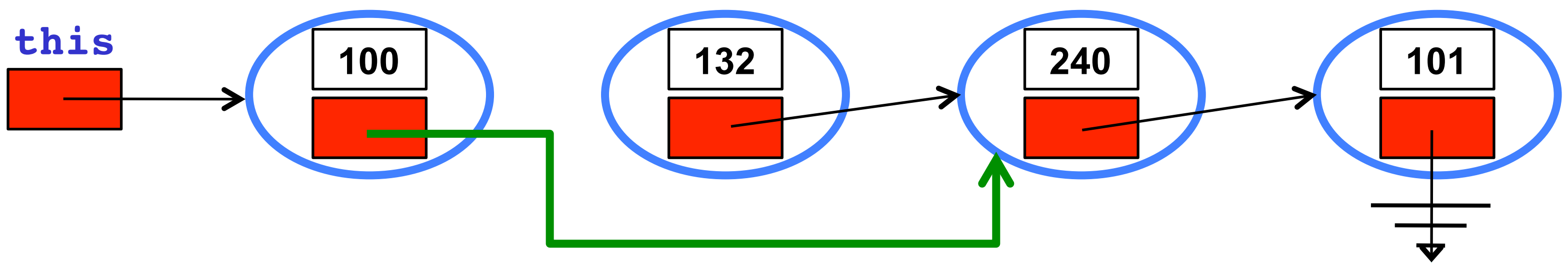
```
this.next = this.next.next;
```

```
}
```



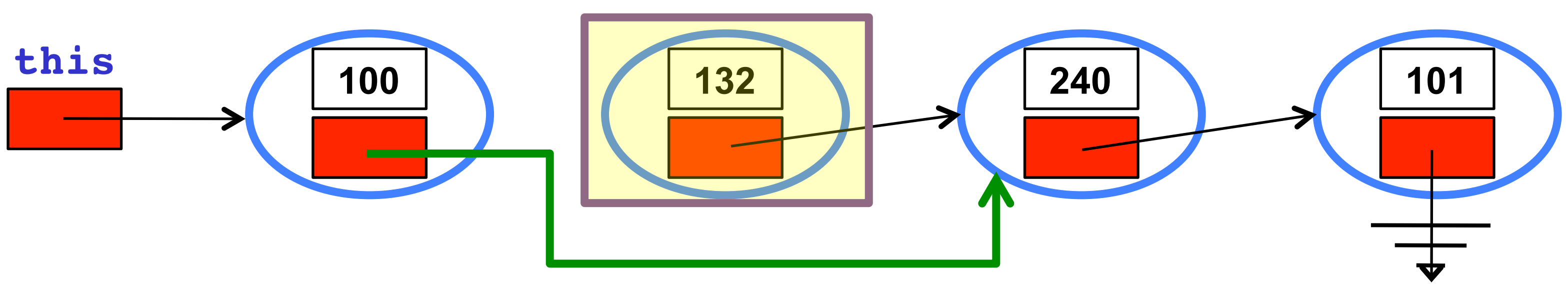
9.3 Some ListNode gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;




}
```



9.3 Some `ListNode` gymnastics

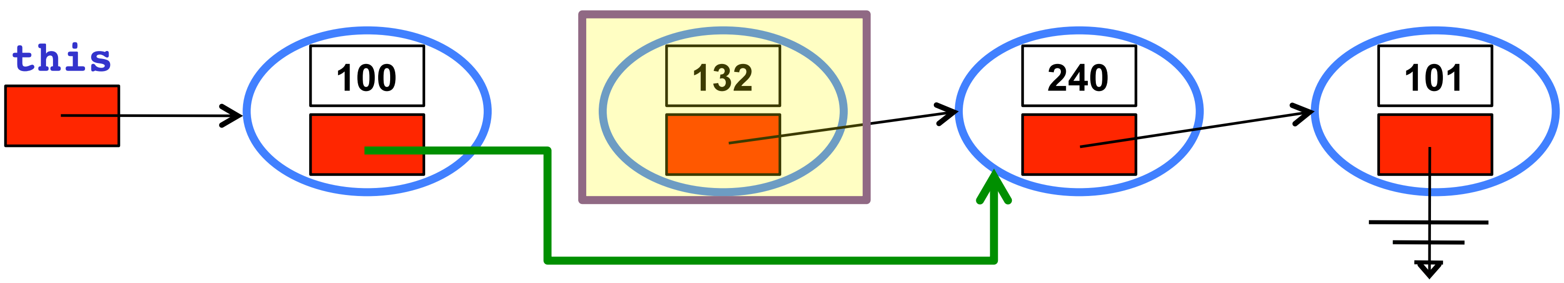
```
public class ListNode<T> {
      T value;
      ListNode<T> next;




}
```



9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

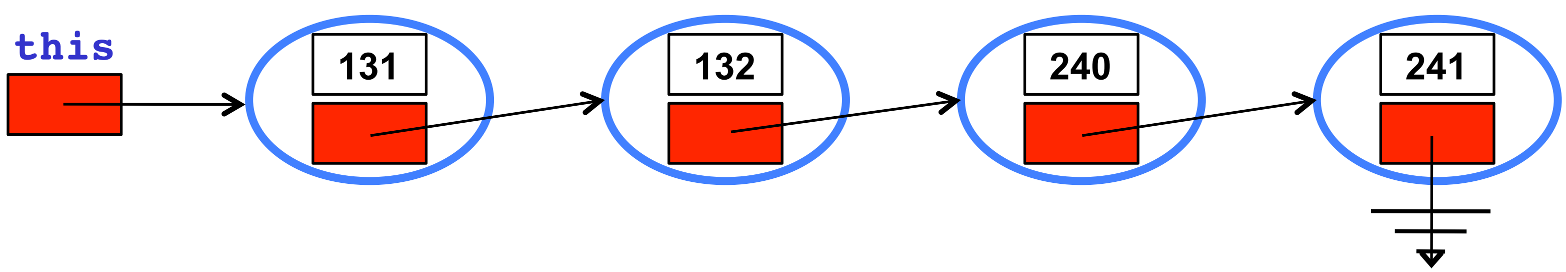
The storage associated with this unreachable node would eventually be recycled by the garbage collector



9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

}
```

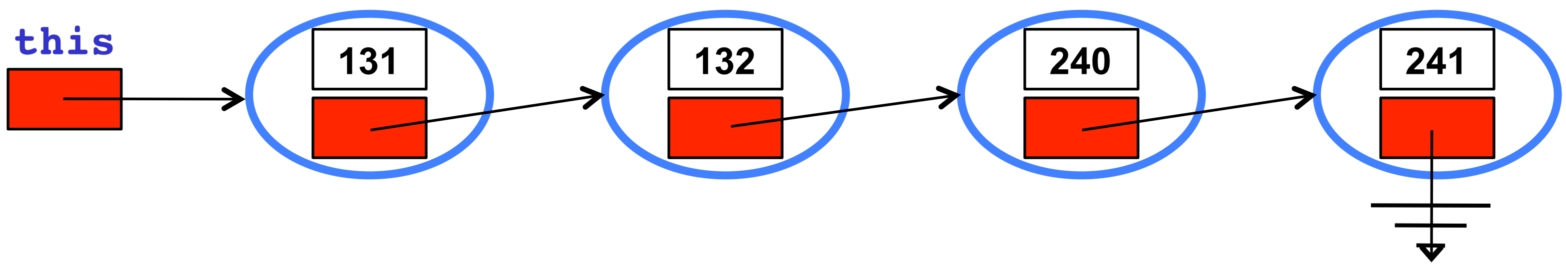
Returning to the list in its original form …



9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

```
this = null;
```

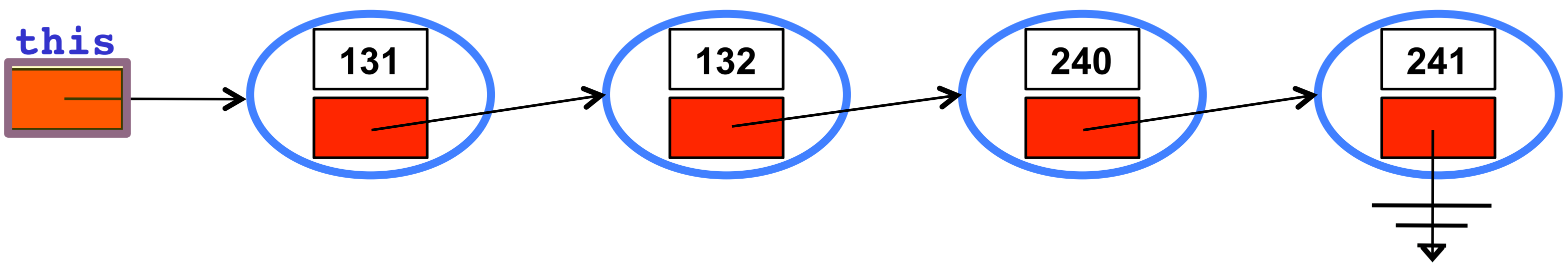


9.3 Some `ListNode` gymnastics

```
public class ListNode<T> {
      T value;
      ListNode<T> next;



}
```

~~this = null;~~

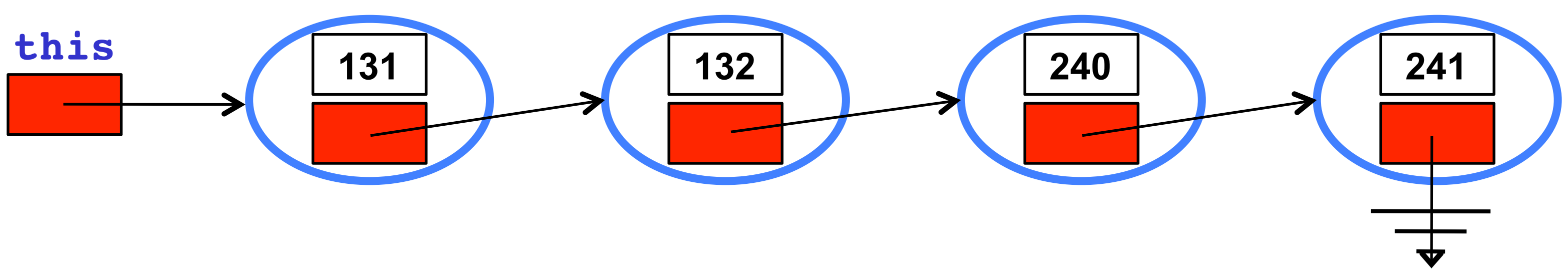
We are not allowed to change `this`. It is inherent to an object and cannot be modified.



End of Oyster

9.4 Roundtable

- **Interaction of quiz on the following diagram**
  - **What is the value of ….**
  - **Show me where …. is stored**
  - **Each time turning to online student first**



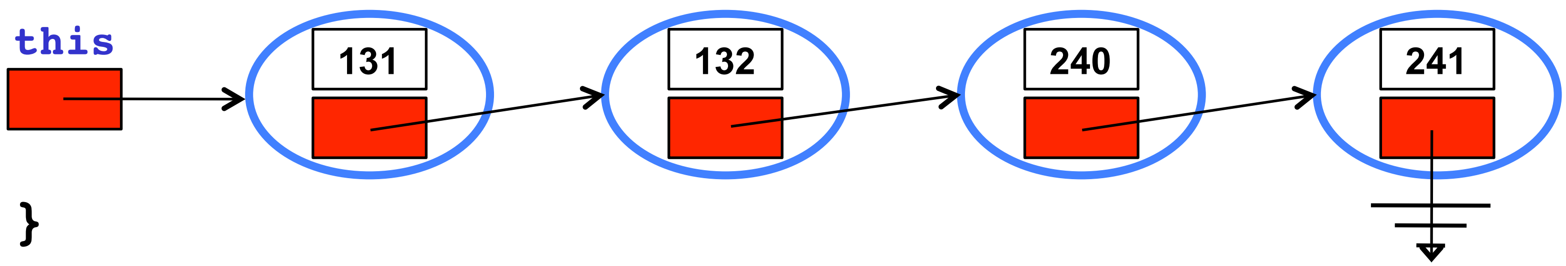
End of Roundtable

9.5 ListNode methods

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {




   }
```

Compute the length of a list beginning at `this ListNode`

- No use of `static` here
  - Method has access to `this` as the first `ListNode` in the list




```
}
```

9.5 `ListNode` methods

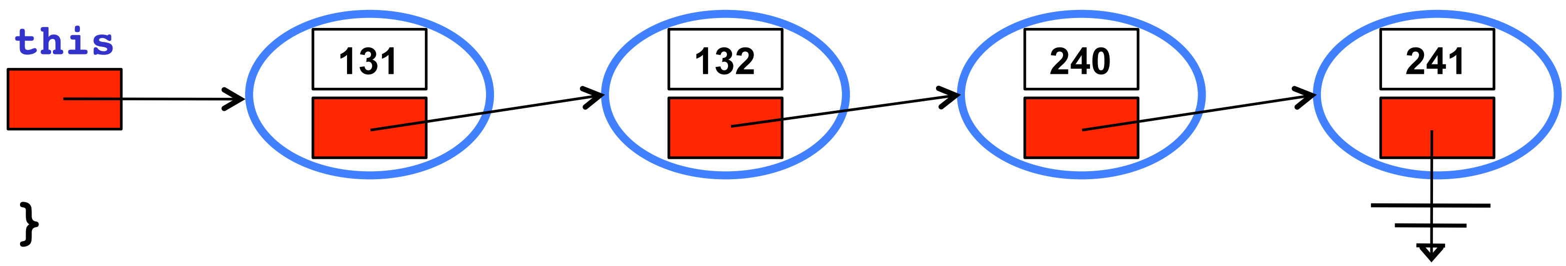
```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {
```

Compute the length of a list beginning at `this ListNode`

- No use of `static` here
  - Method has access to `this` as the first `ListNode` in the list

For this problem, we will use recursion

```
   }
```




```
}
```

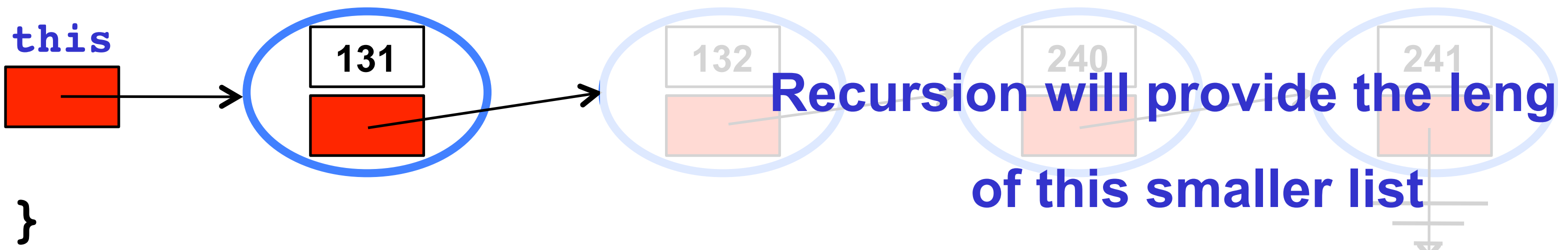
9.5 ListNode methods

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {

   }
}
```

Compute the length of a list beginning at **this ListNode**

- **No use of `static` here**
  - Method has access to `this` as the first `ListNode` in the list

For this problem, we will use recursion



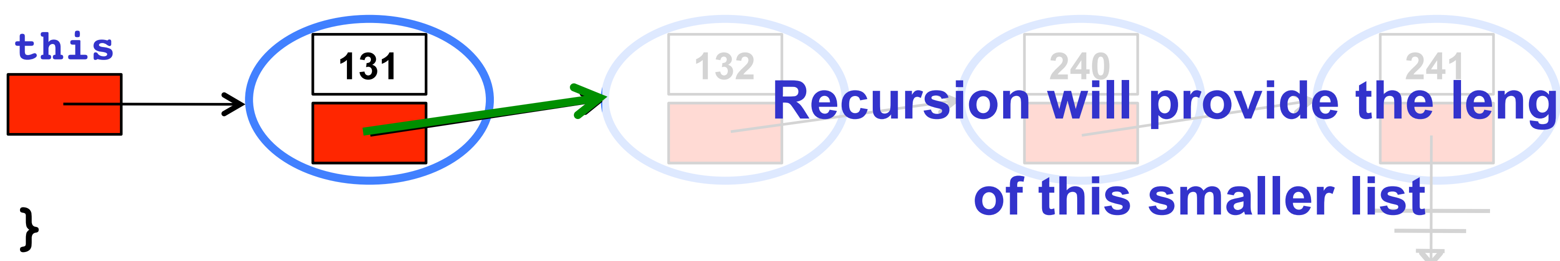
Recursion will provide the length of this smaller list

9.5 `ListNode` methods

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {




   }
```

Compute the length of a list beginning at `this` `ListNode`

- No use of `static` here
  - Method has access to `this` as the first `ListNode` in the list
  - `this.next` references the smaller list



Recursion will provide the length of this smaller list

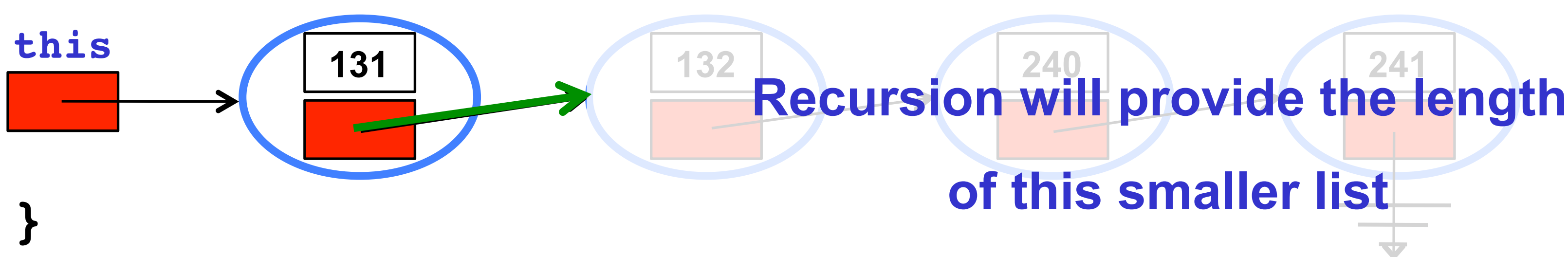
```
}
```

9.5 `ListNode` methods

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {






   }

}
```

Compute the length of a list beginning at `this ListNode`

- No use of `static` here
  - Method has access to `this` as the first `ListNode` in the list
  - `this.next` references the smaller list
  - `this.next.size()` will compute its size

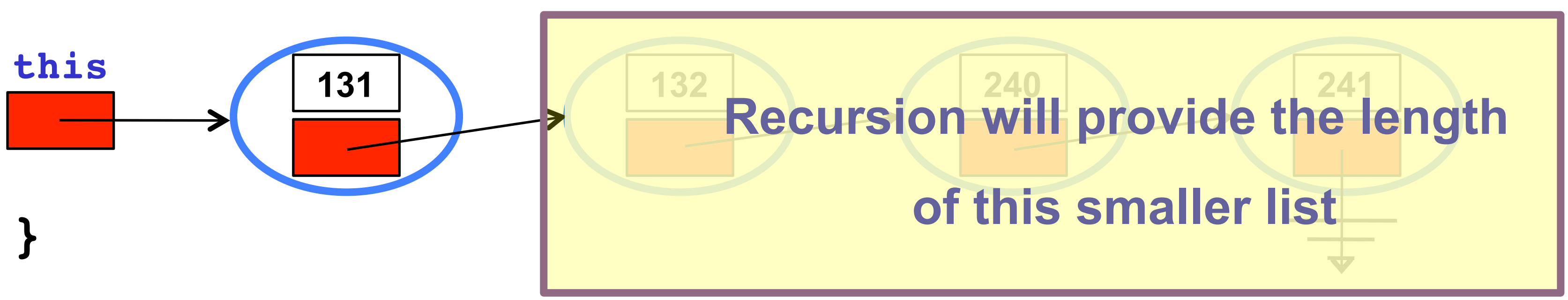


9.5 ListNode methods

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {



              return 1 + this.next.size();
   }
```

Compute the length of a list beginning at **this ListNode**

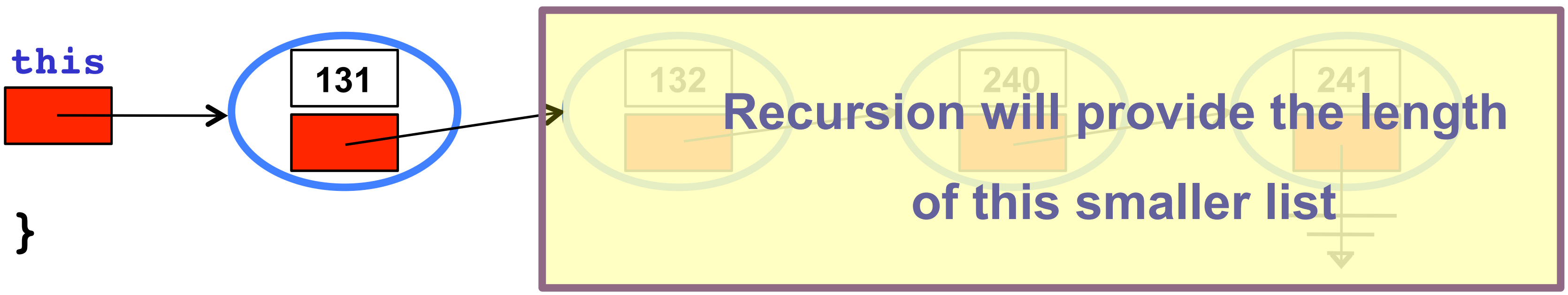
- Given the length of the smaller list
  - **this** list is just one bigger




```
}
```

9.5 `ListNode` methods

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {



                  return 1 + this.next.size();
   }
```




```
}
```

Compute the length of a list beginning at `this ListNode`

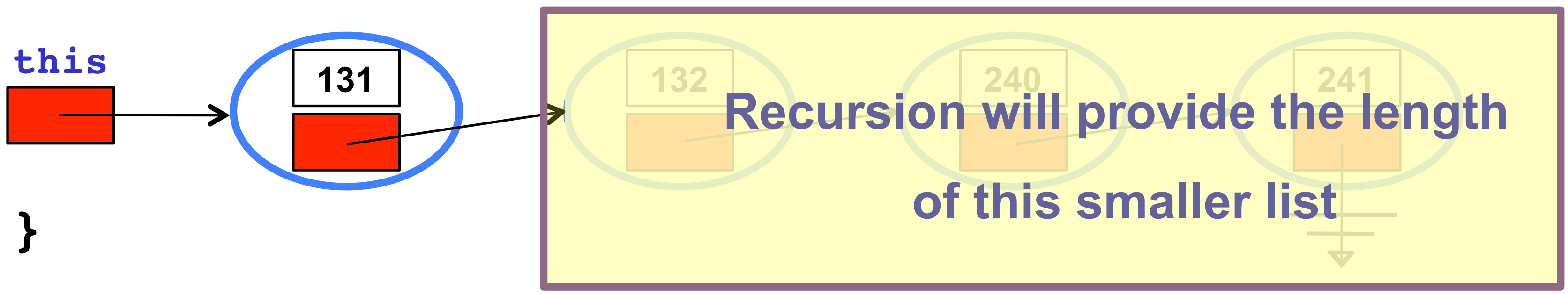
- Given the length of the smaller list
  - `this` list is just one bigger

9.5 `ListNode` methods

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {



            return 1 + this.next.size();
   }

}
```

Compute the length of a list beginning at `this ListNode`

- Given the length of the smaller list
  - `this` list is just one bigger



9.5 `ListNode` methods

Compute the length of a list beginning at `this` `ListNode`

- Base case
  - Only one node

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {
      if (this.next == null) {
            return 1;
      else
            return 1 + this.next.size();
   }
```

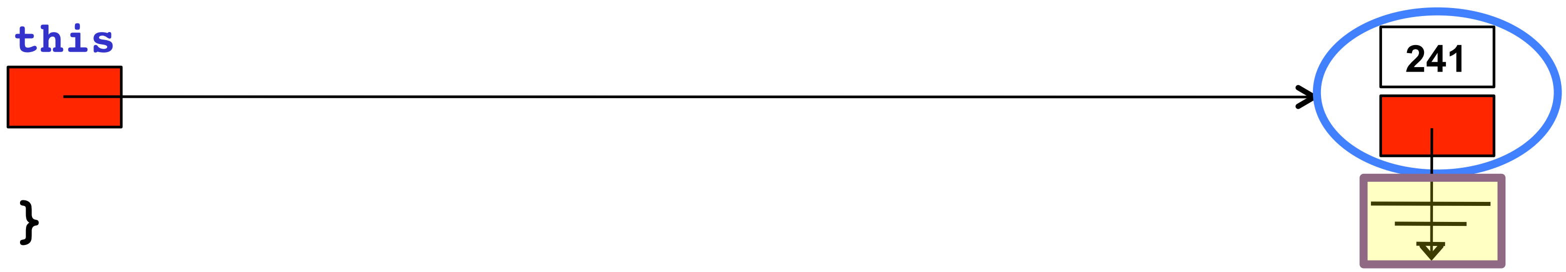

```
}
```

9.5 `ListNode` methods

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {
      if (this.next == null) {
            return 1;
      else
            return 1 + this.next.size();
   }
```

Compute the length of a list beginning at `this` `ListNode`

- Base case
  - Only one node



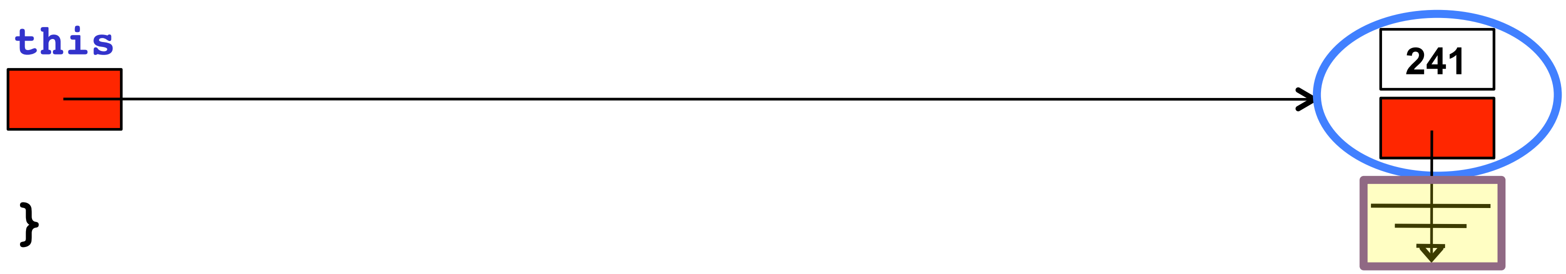

```
}
```

9.5 `ListNode` methods

Compute the length of a list beginning at `this` `ListNode`

- **Why 1 and not 0?**
  - Because `this` can never be null

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {
      if (this.next == null) {
            return 1;
      else
            return 1 + this.next.size();
   }
```



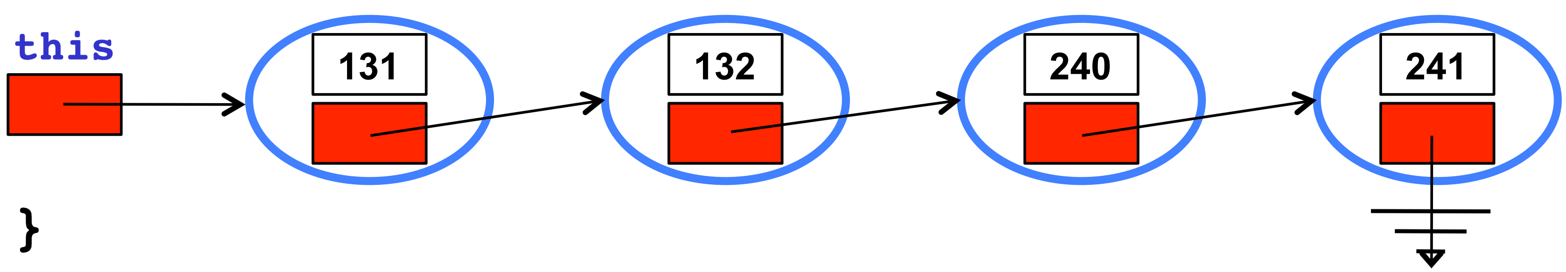

```
}
```

9.5 `ListNode` methods

Compute the length of a list beginning at `this` `ListNode`

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {
      if (this.next == null) {
            return 1;
      else
            return 1 + this.next.size();
   }
```

We next compute this.size() for the list below

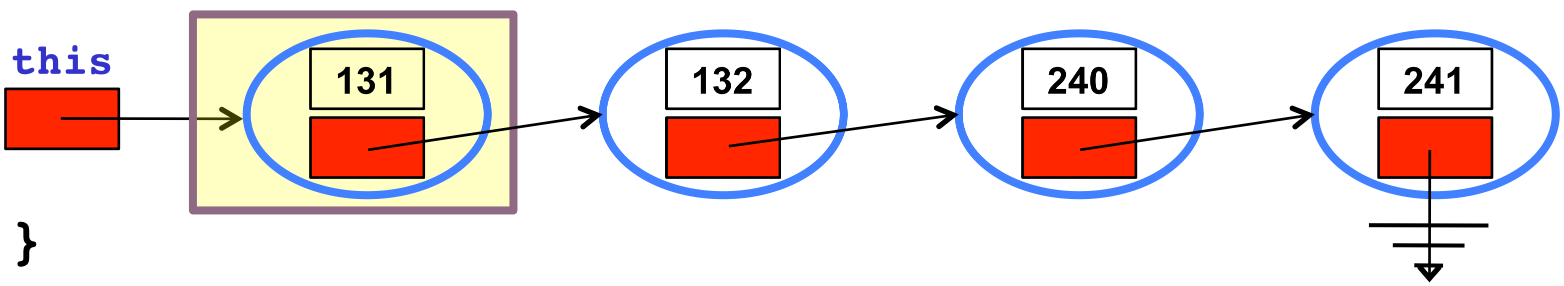



```
}
```

9.5 `ListNode` methods

Compute the length of a list beginning at `this ListNode`

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {
      if (this.next == null) {
            return 1;
      else
            return 1 + this.next.size();
   }
```



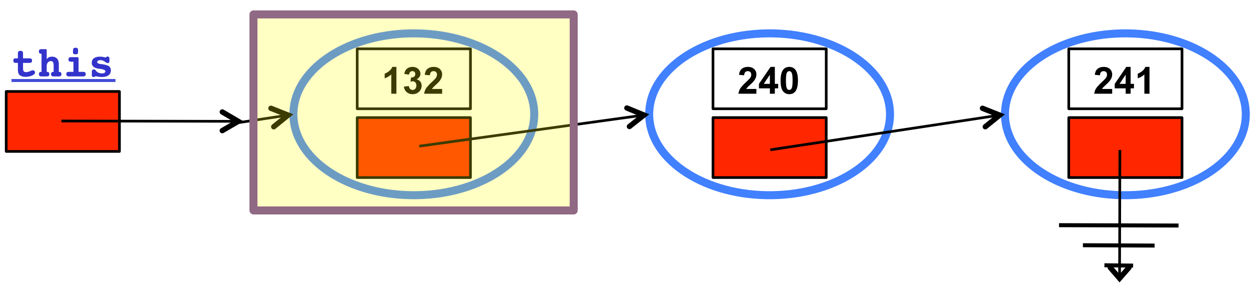

```
}
```

1 +

9.5 ListNode methods

Compute the length of a list beginning at **this ListNode**

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {
      if (this.next == null) {
            return 1;
      else
            return 1 + this.next.size();
   }
```



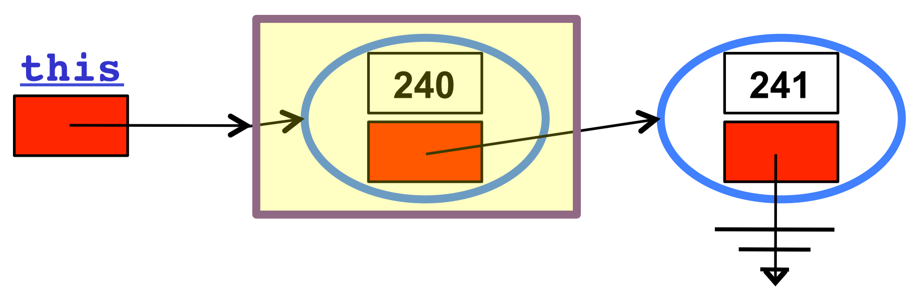

```
}
```

1 + 1 +

9.5 ListNode methods

Compute the length of a list beginning at `this` `ListNode`

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {
      if (this.next == null) {
            return 1;
      else
            return 1 + this.next.size();
   }

}
```

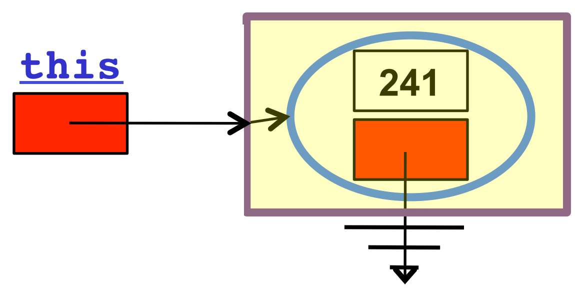


1 + 1 + 1 +

9.5 `ListNode` methods

Compute the length of a list beginning at `this ListNode`

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {
      if (this.next == null) {
            return 1;
      else
            return 1 + this.next.size();
   }

}
```

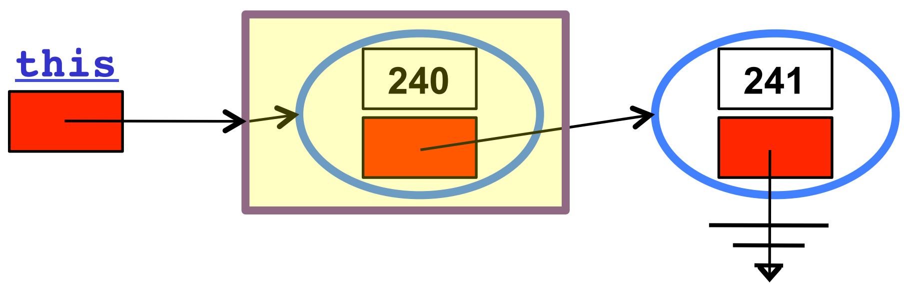


1 + 1 + 1 + 1

9.5 ListNode methods

Compute the length of a list beginning at **this ListNode**

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {
      if (this.next == null) {
            return 1;
      else
            return 1 + this.next.size();
   }

}
```

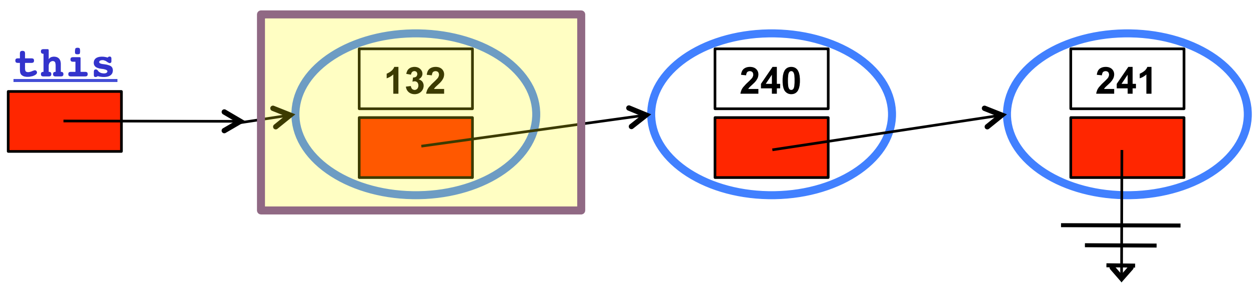


1 + 1 + 2

9.5 `ListNode` methods

Compute the length of a list beginning at `this ListNode`

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {
      if (this.next == null) {
            return 1;
      else
            return 1 + this.next.size();
   }
```



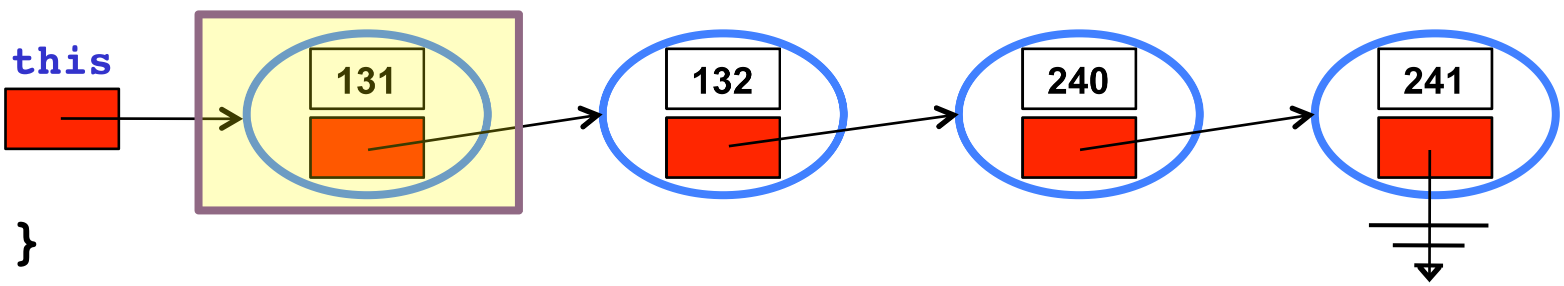

```
}
```

1 + 3

9.5 ListNode methods

```
public class ListNode<T> {
      T value;
      ListNode<T> next;

   public int size() {
      if (this.next == null) {
            return 1;
      else
            return 1 + this.next.size();
   }
```

Compute the length of a list beginning at `this` `ListNode`




```
}
```

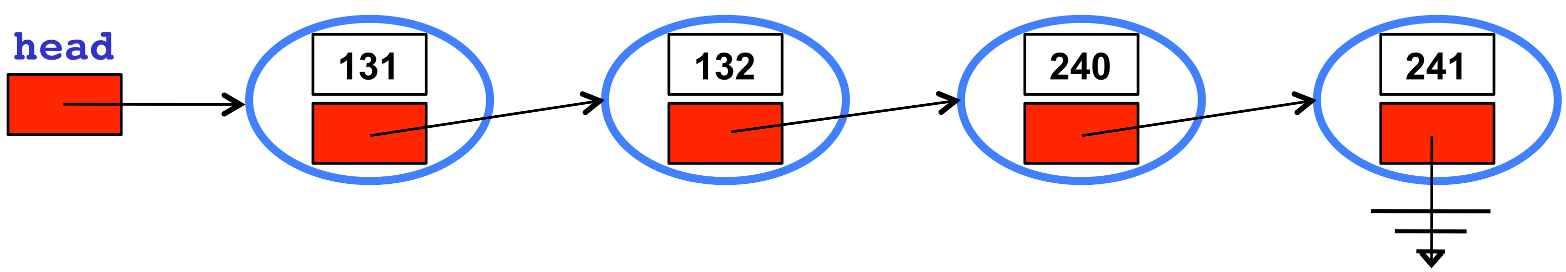
4

End of Oyster

Oyster 5b

9.5 `ListNode` methods

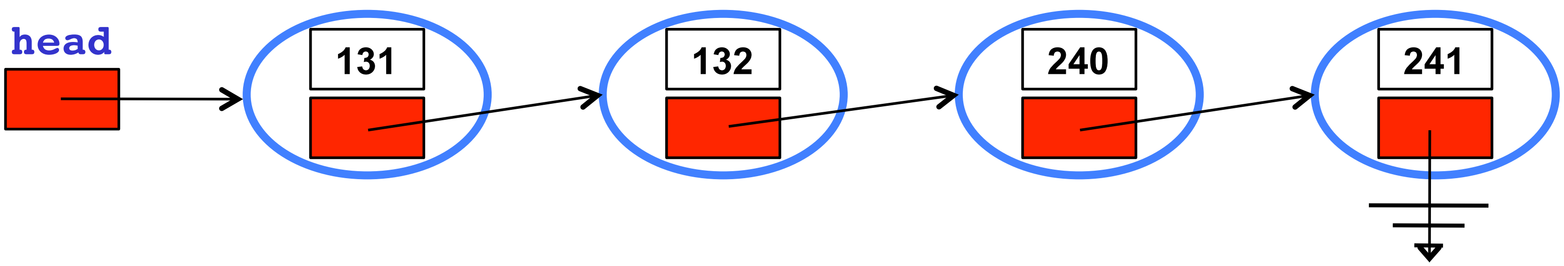
Sum the elements starting at `head`



9.5 `ListNode` methods

Sum the elements starting at **`head`**

```
public static int sum(ListNode<Integer> head) {




}
```

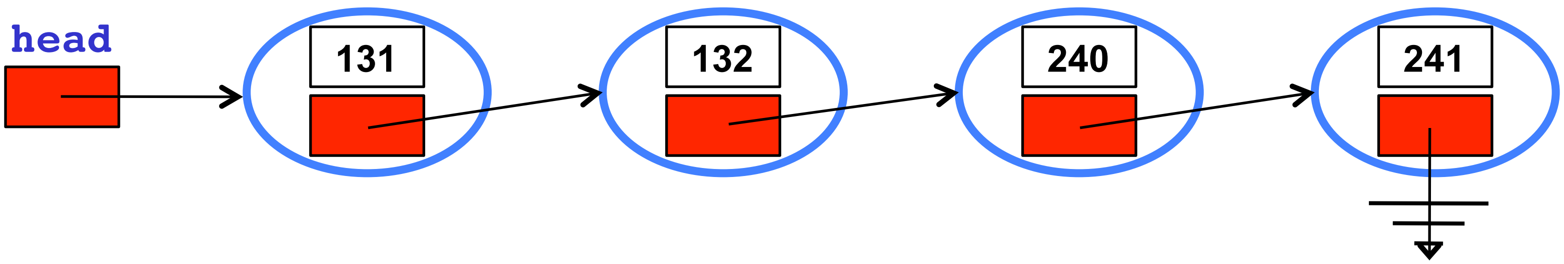


9.5 `ListNode` methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {



}
```

For this problem, we will use iteration

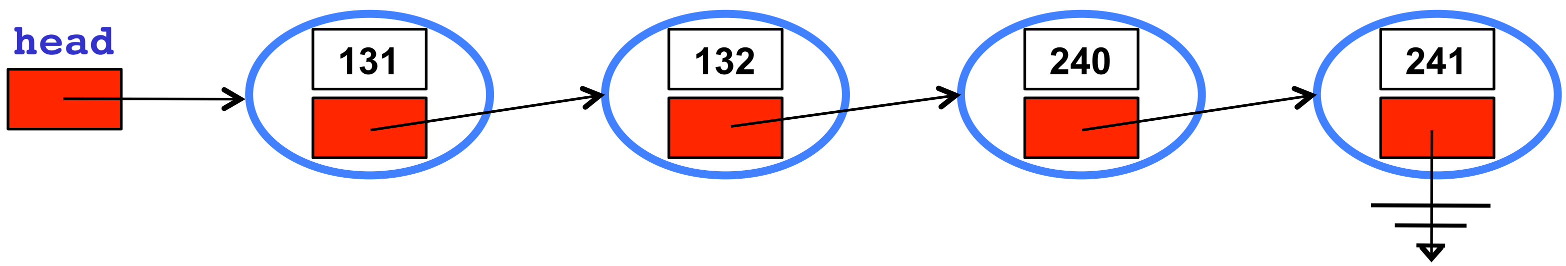


9.5 `ListNode` methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {



}
```

For this problem, we will use iteration and a `static` method

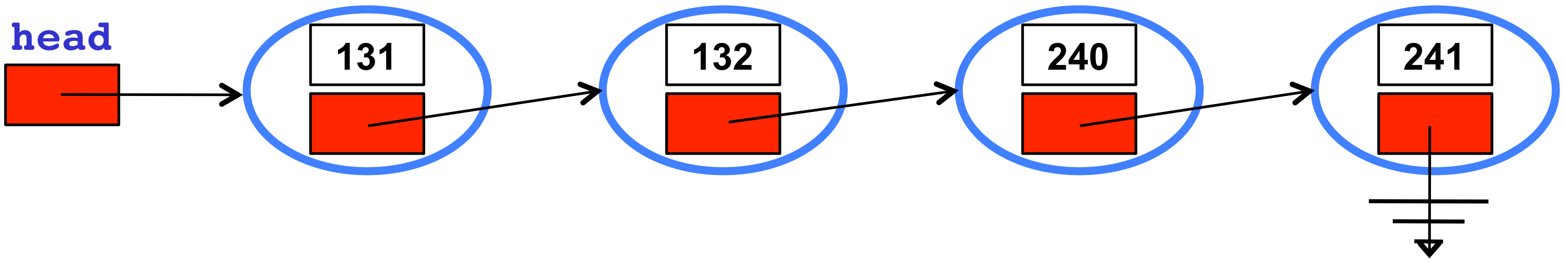


9.5 ListNode methods

Sum the elements starting at head

```
public static int sum(ListNode<Integer> head) {

}
```

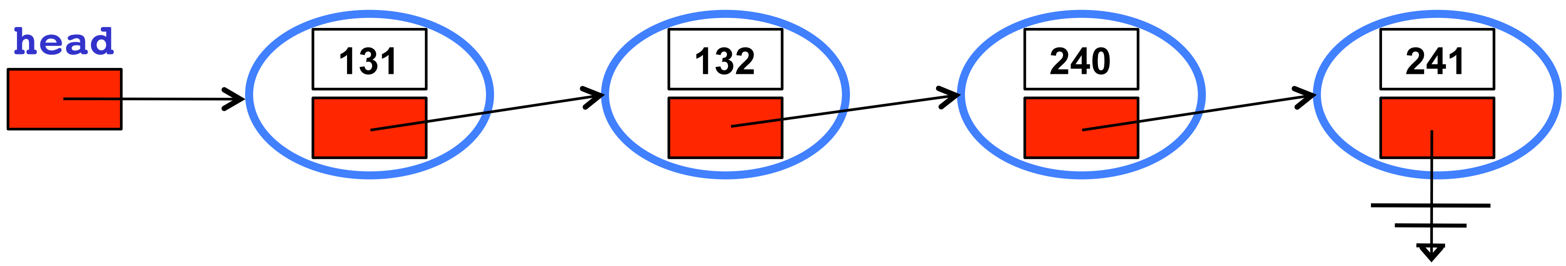
With no this, the method requires the first node as a parameter



9.5 `ListNode` methods

Sum the elements starting at `head`

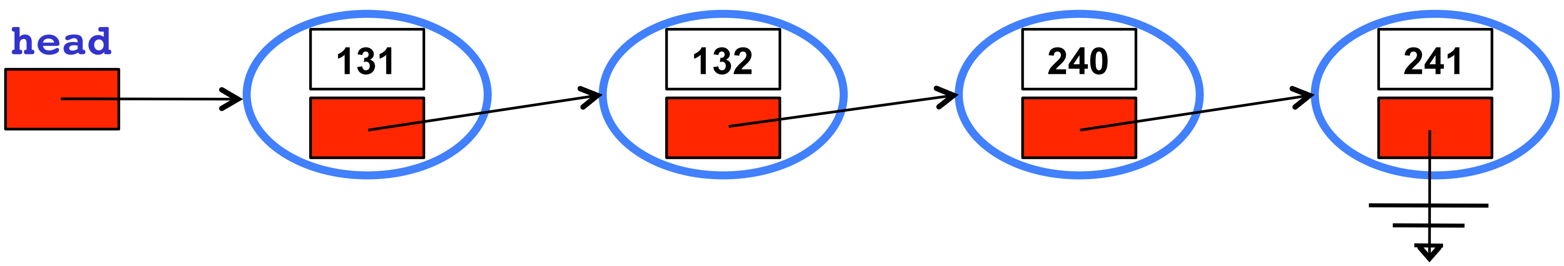
```
public static int sum(ListNode<Integer> head) {
   int ans = 0;



   return ans;
}
```



9.5 `ListNode` methods

Sum the elements starting at `head`

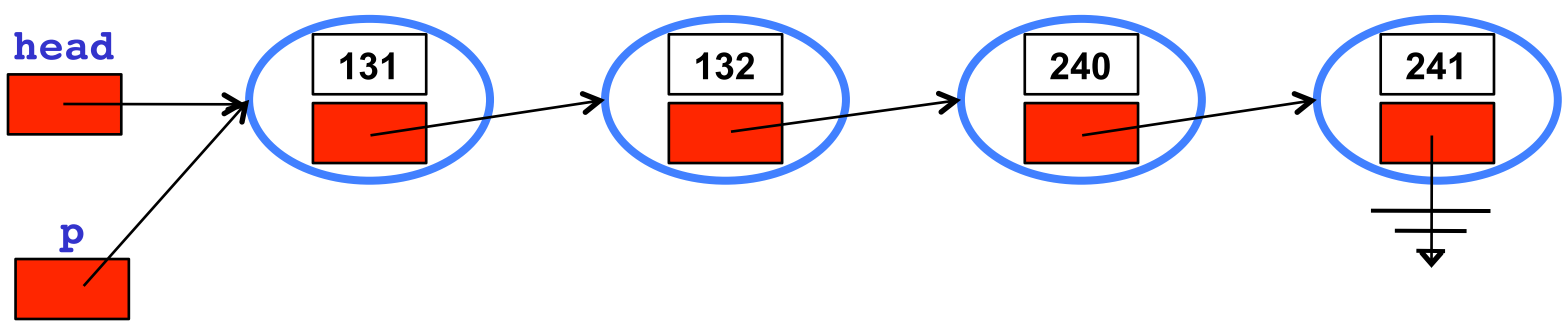
```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {

   }
   return ans;
}
```



9.5 ListNode methods

Sum the elements starting at head

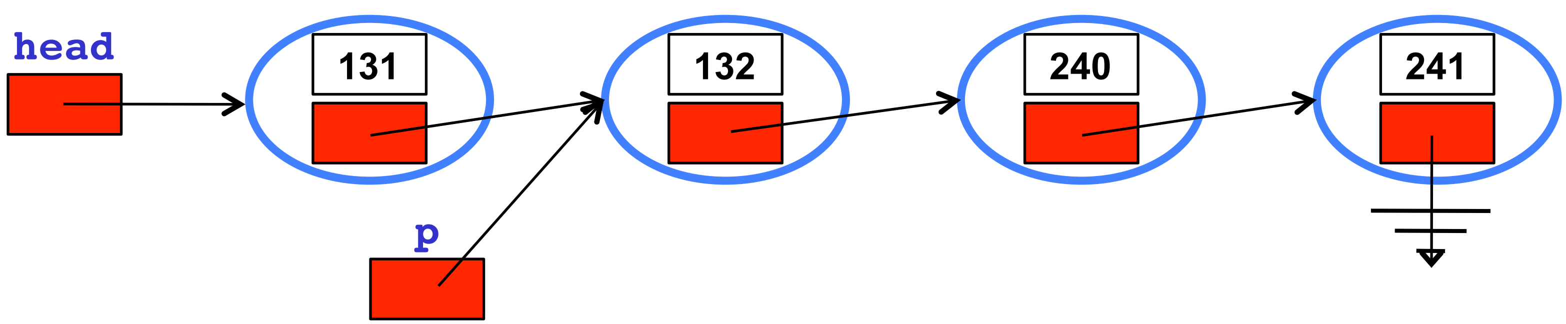
```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {

   }
   return ans;
}
```



9.5 ListNode methods

Sum the elements starting at `head`

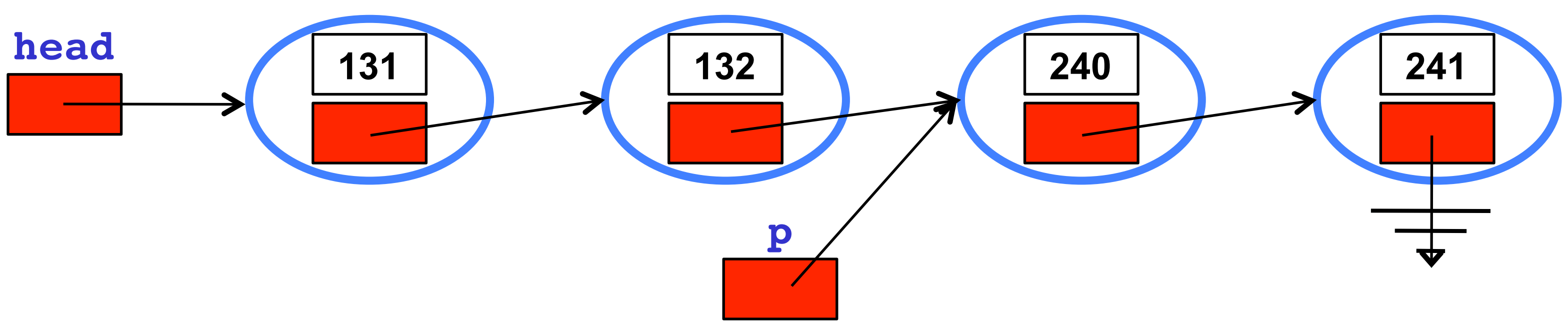
```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {

   }
   return ans;
}
```



9.5 `ListNode` methods

Sum the elements starting at `head`

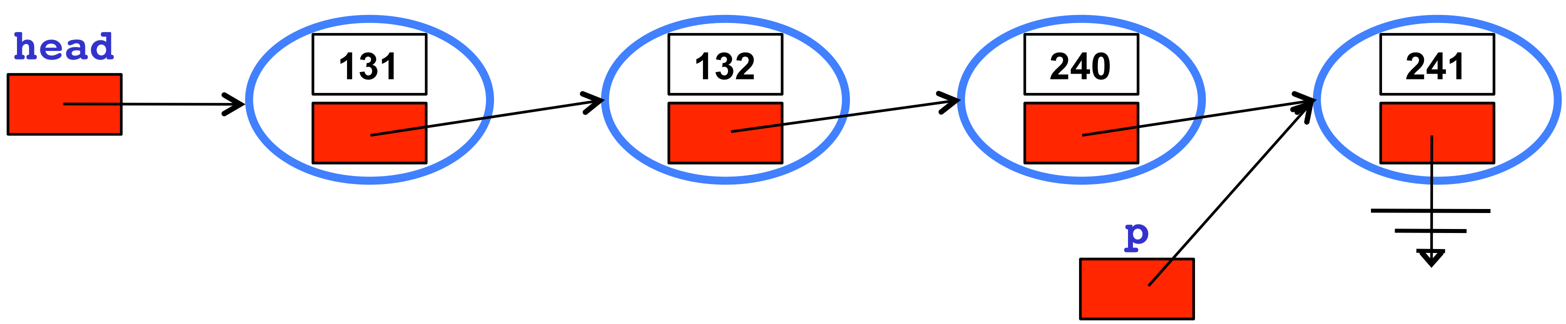
```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {

   }
   return ans;
}
```



9.5 `ListNode` methods

Sum the elements starting at `head`

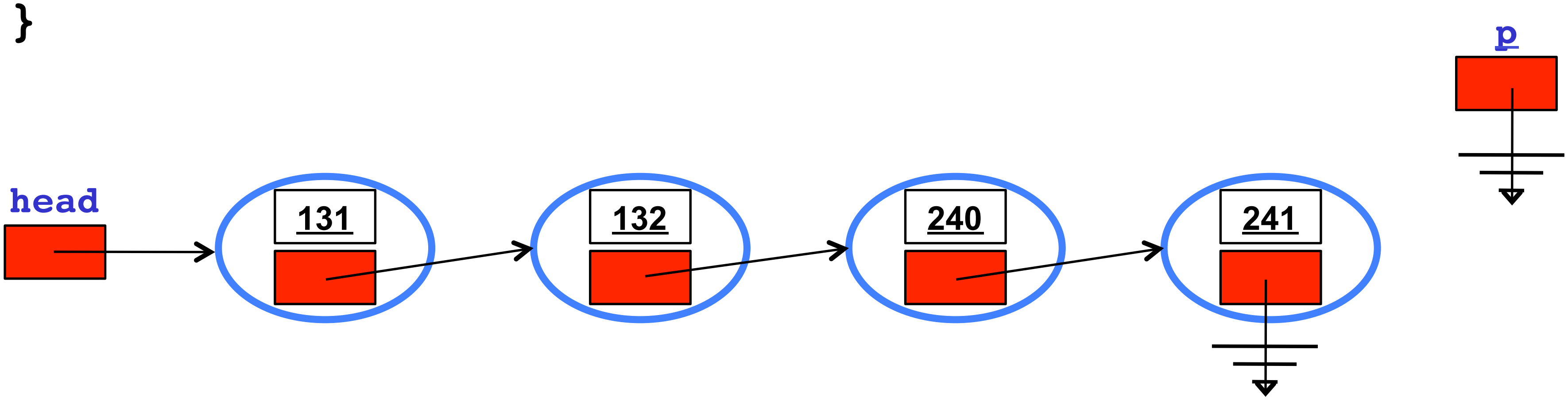
```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {

   }
   return ans;
}
```



9.5 `ListNode` methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {

   }
   return ans;
}
```

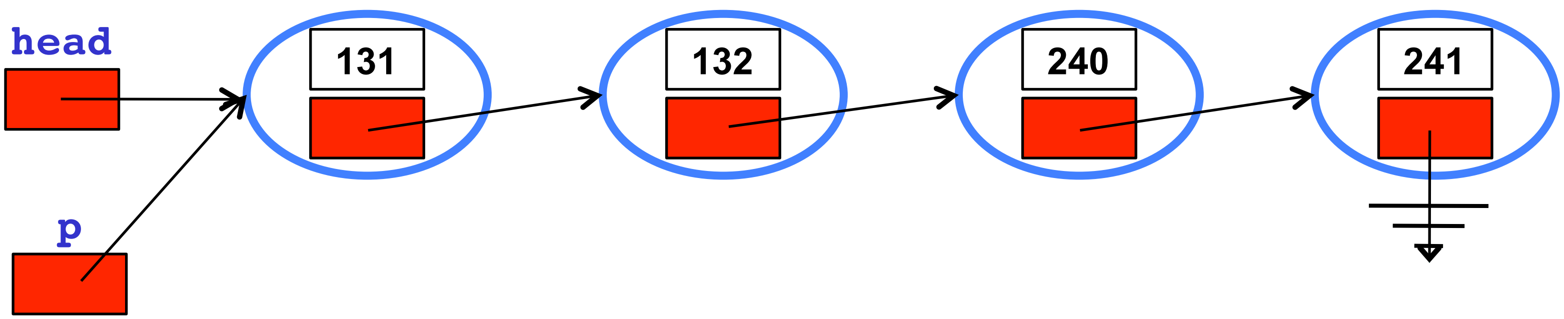


9.5 `ListNode` methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {
        ans = ans + p.value;
   }
   return ans;
}
```

ans	0

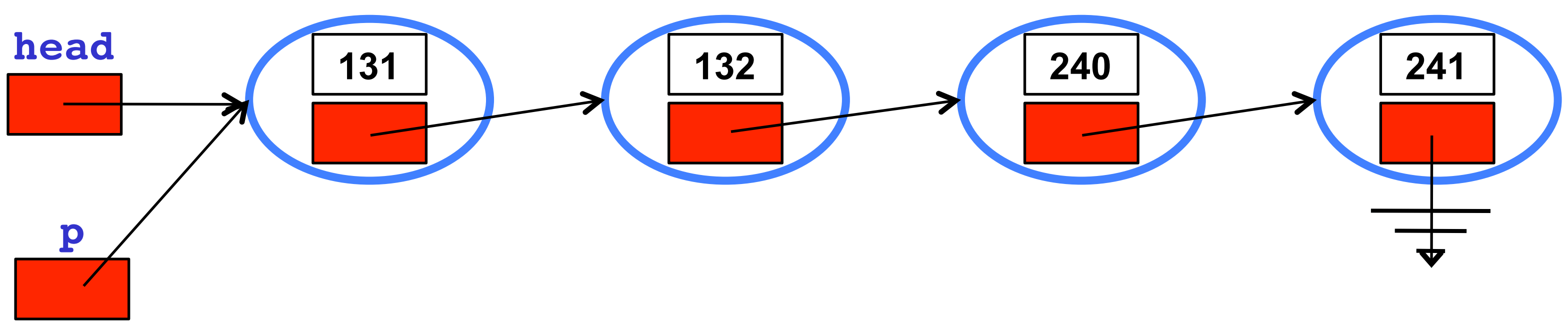


9.5 `ListNode` methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {
       ans = ans + p.value;
   }
   return ans;
}
```

ans	131

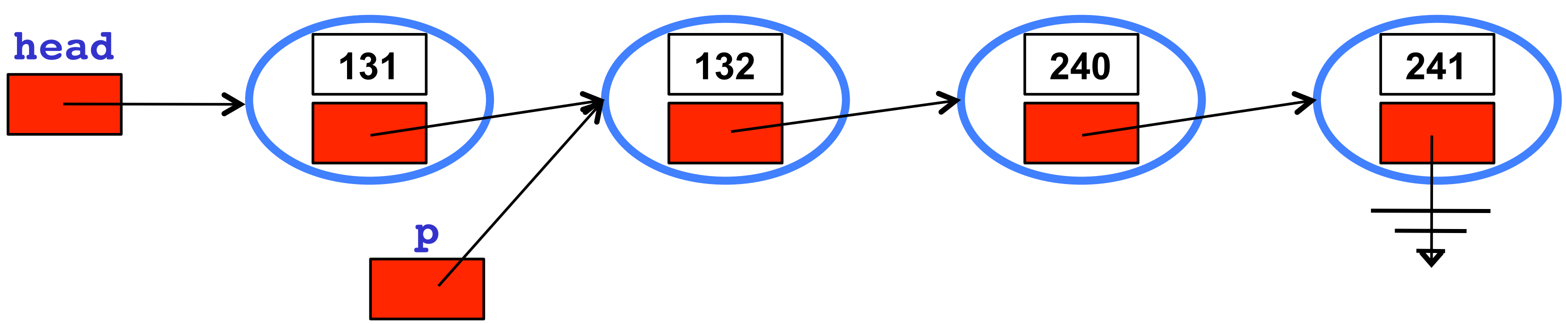


9.5 ListNode methods

Sum the elements starting at head

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {
        ans = ans + p.value;
   }
   return ans;
}
```

ans	131

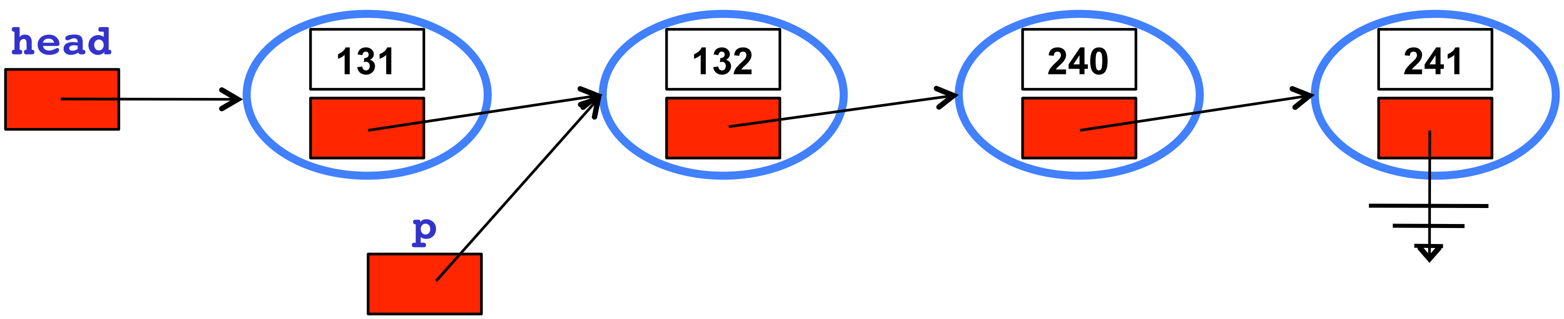


9.5 `ListNode` methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {
      ans = ans + p.value;
   }
   return ans;
}
```

ans	263

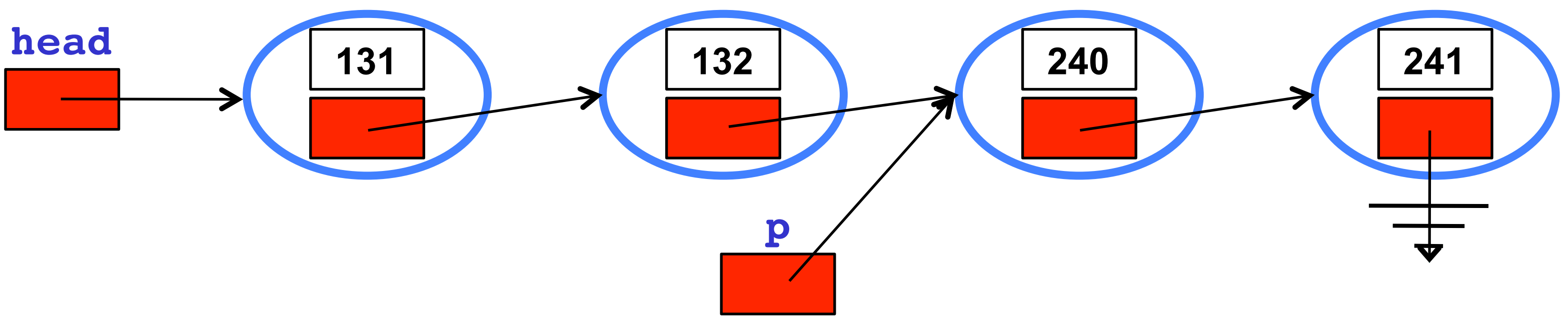


9.5 `ListNode` methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {
       ans = ans + p.value;
   }
   return ans;
}
```

ans	263

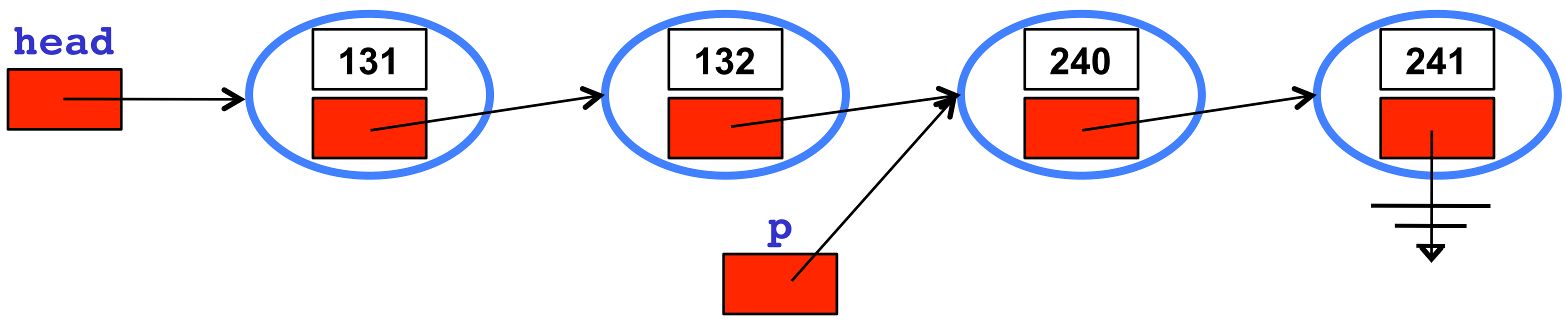


9.5 `ListNode` methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {
      ans = ans + p.value;
   }
   return ans;
}
```

ans	503

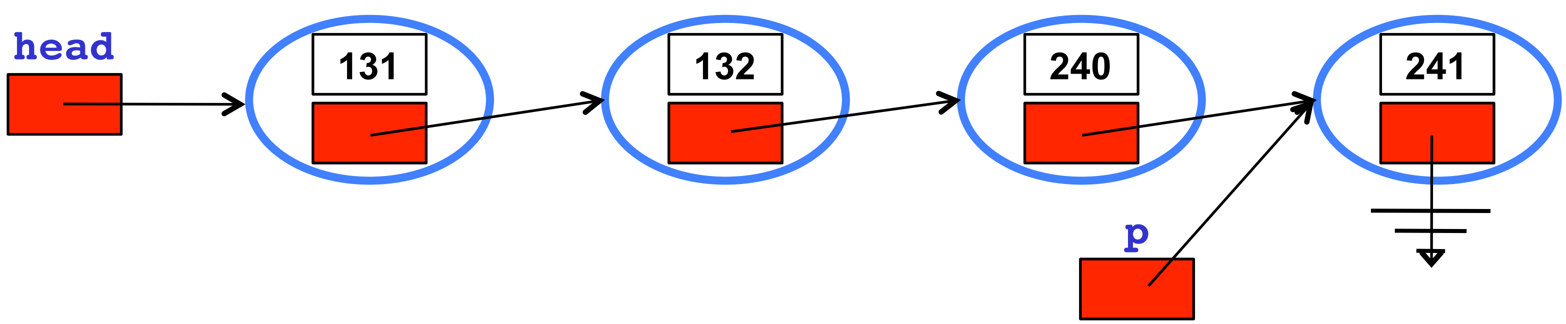


9.5 `ListNode` methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {
      ans = ans + p.value;
   }
   return ans;
}
```

ans	503

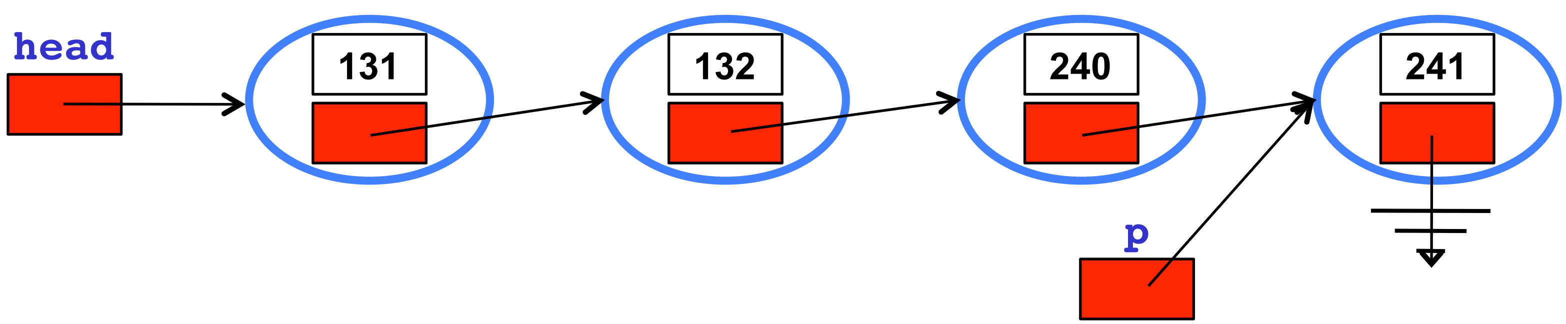


9.5 `ListNode` methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {
      ans = ans + p.value;
   }
   return ans;
}
```

ans	744

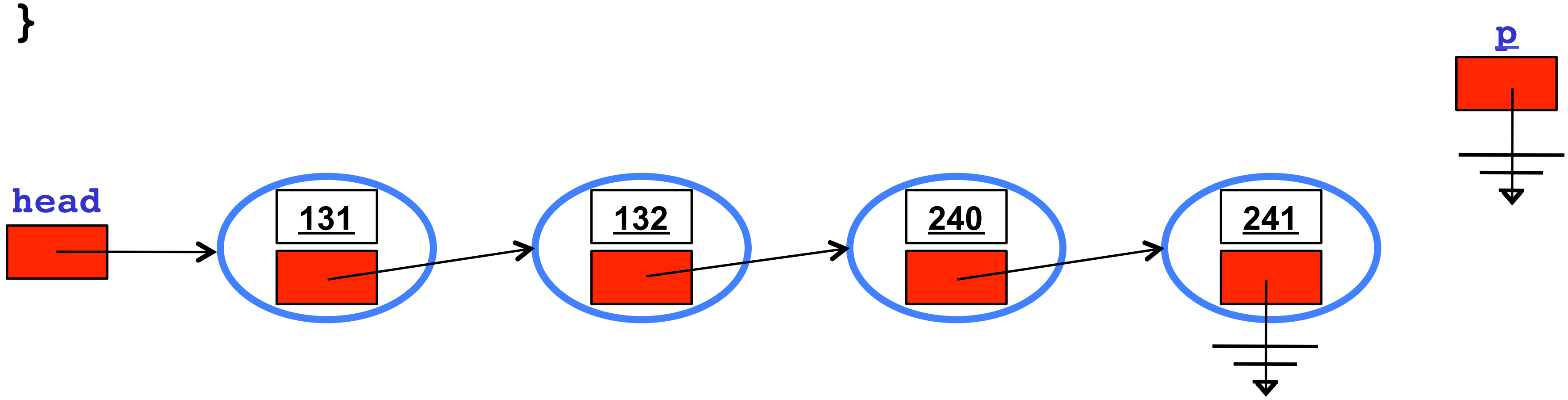


9.5 ListNode methods

Sum the elements starting at head

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {
      ans = ans + p.value;
   }
   return ans;
}
```

ans	744

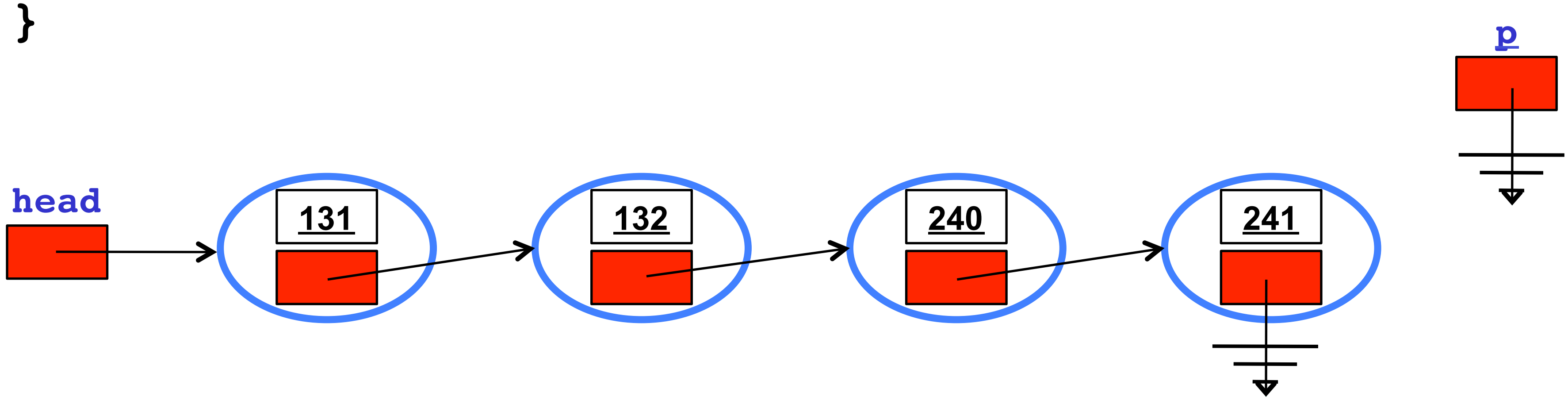


9.5 `ListNode` methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {
      ans = ans + p.value;
   }
   return ans;
}
```

ans	744

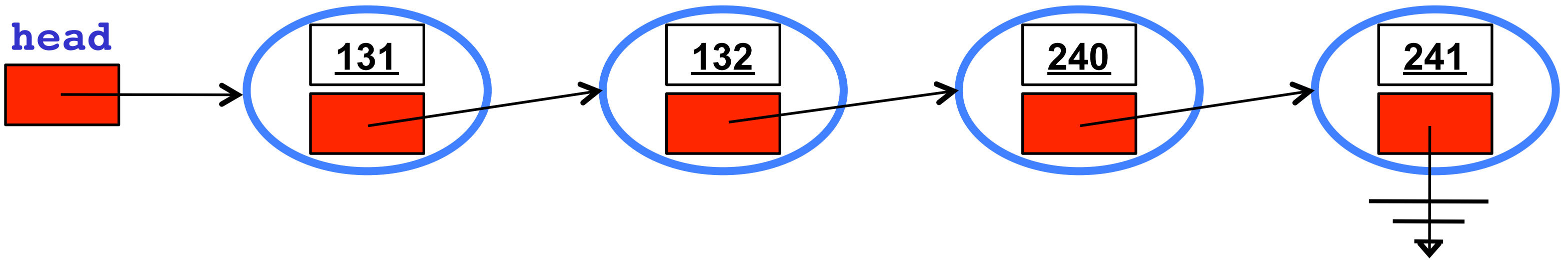


9.5 ListNode methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {
      ans = ans + p.value;
   }
   return ans;
}
```

ans	744

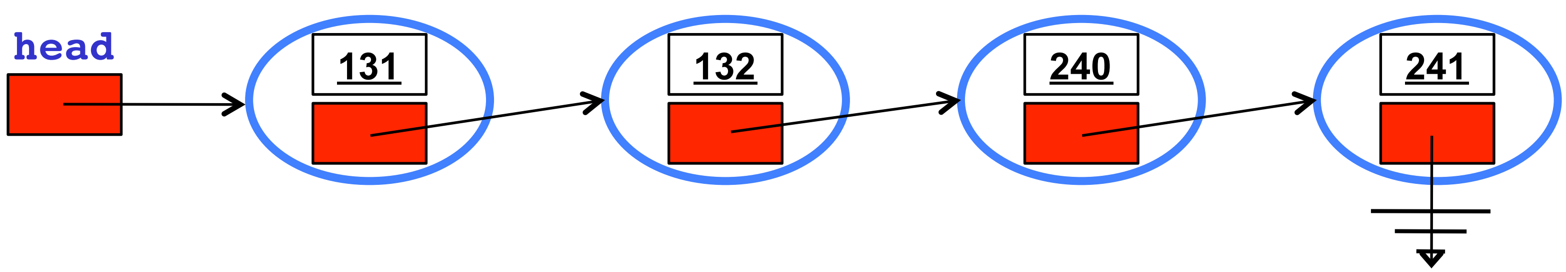


9.5 `ListNode` methods

Sum the elements starting at `head`

```
public static int sum(ListNode<Integer> head) {
   int ans = 0;
   for (ListNode<Integer> p = head; p != null; p=p.next) {
       ans = ans + p.value;
   }
   return ans;
}
```

Thus, `sum(head)` returns `744`



End of Oyster

Roundtable 6

9.6 Roundtable

- **We will complete the following method**
  - **size(), static and recursively**
    - Different base case!

End of Roundtable

BLT 7a,b,c,d

9.7 Exercises

- **Write each of the following methods**
  - **size(), static and iteratively**
  - **size(), static and recursively**
- **Write the following method**
  - **sum(), static but recursively**
    - Use a "helper" method if necessary
  - **Why can't we write a nonstatic version of the sum() method?**

End of BLT

Oyster 8a

9.8 A `LinkedList` implementation of `List`

- Consider a List ADT:
  - `T` is a type parameter

```
public interface List<T> {




}
```

9.8 A `LinkedList` implementation of `List`

- Consider a List ADT:
  - `T` is a type parameter

```
public interface List<T> {
```

We can use `T` in here to declare the type of the values contained in the list

```
}
```

9.8 A `LinkedList` implementation of `List`

- **Consider a List ADT:**

```
public interface List<T> {






}
```

9.8 A `LinkedList` implementation of `List`

- **Add the item `x` of type `T` to the beginning of the list.**

```
public interface List<T> {
  public void addFirst(T x);



}
```

9.8 A `LinkedList` implementation of `List`

- Add the item `x` of type `T` to the end of the list.

```
public interface List<T> {
  public void addFirst(T x);
  public void addLast(T x);



}
```

9.8 A `LinkedList` implementation of `List`

- Find and remove the first occurrence of `x` in the list, returning `true` if successful

```
public interface List<T> {
  public void addFirst(T x);
  public void addLast(T x);
  public boolean remove(T x);



}
```

9.8 A `LinkedList` implementation of `List`

- **Return the i^{th} element in the list**
  - **First element is at index 0**
  - **Return null if no such element exists**

```
public interface List<T> {
  public void addFirst(T x);
  public void addLast(T x);
  public boolean remove(T x);
  public T get(int i);


}
```

9.8 A `LinkedList` implementation of `List`

- **Return the size of the list**
  - **The number of elements in the list**

```
public interface List<T> {
  public void addFirst(T x);
  public void addLast(T x);
  public boolean remove(T x);
  public T get(int i);
  public int size();

}
```

9.8 A `LinkedList` implementation of `List`

- **Is the list empty?**
  - **Can use the principle of *reduction***

```
public interface List<T> {
  public void addFirst(T x);
  public void addLast(T x);
  public boolean remove(T x);
  public T get(int i);
  public int size();
  public boolean isEmpty();
}
```

9.8 A `LinkedList` implementation of `List`

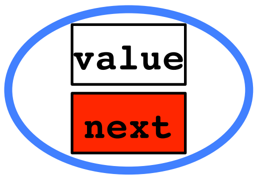
- Is the list empty?
  - Can use the principle of *reduction*
  - Same as asking if `size() == 0`

```
public interface List<T> {
  public void addFirst(T x);
  public void addLast(T x);
  public boolean remove(T x);
  public T get(int i);
  public int size();
  public boolean isEmpty();
}
```

9.8 A `LinkedList` implementation of `List`

- We next implement the List<T> interface
  - A SinglyLinkedList<T> implementation

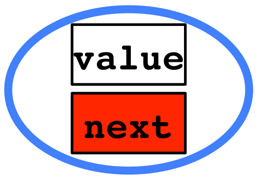
- Two kinds of objects involved
  - `ListNode<T>` has-a
    - `T value`
    - `ListNode<T> next`



9.8 A `LinkedList` implementation of `List`

- **We next implement the List<T> interface**
  - **A SinglyLinkedList<T> implementation**

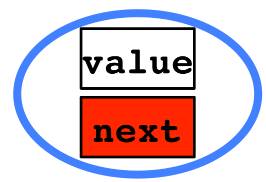
- **Two kinds of objects involved**
  - **`ListNode<T>` has-a**
    - `T value`
    - `ListNode<T> next`
    - We will use many of these per list



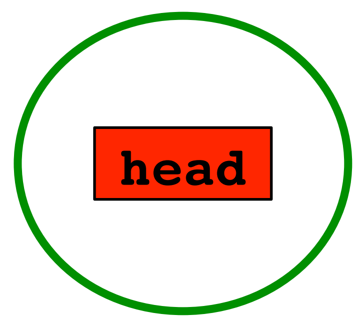
9.8 A `LinkedList` implementation of `List`

- **We next implement the List<T> interface**
  - **A SinglyLinkedList<T> implementation**

- **Two kinds of objects involved**
  - **`ListNode<T>` has-a**
    - `T value`
    - `ListNode<T> next`
    - We will use many of these per list



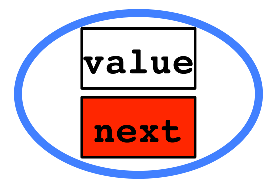
  - **`SinglyLinkedList<T>` has-a**
    - `ListNode<T> head`



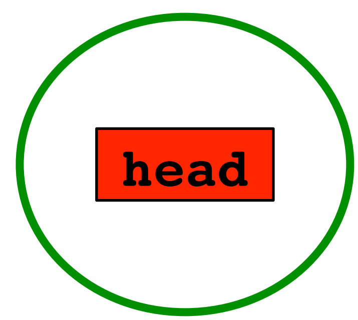
9.8 A **LinkedList** implementation of **List**

- **We next implement the List<T> interface**
  - **A SinglyLinkedList<T> implementation**

- **Two kinds of objects involved**
  - **ListNode<T> has-a**
    - T value
    - ListNode<T> next
    - We will use many of these per list

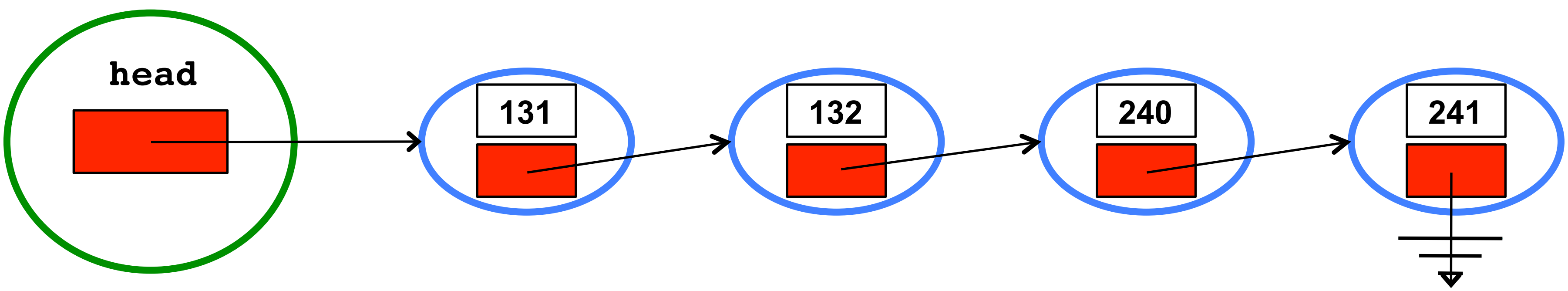


  - **SinglyLinkedList<T> has-a**
    - ListNode<T> head
    - Only one of these used per list



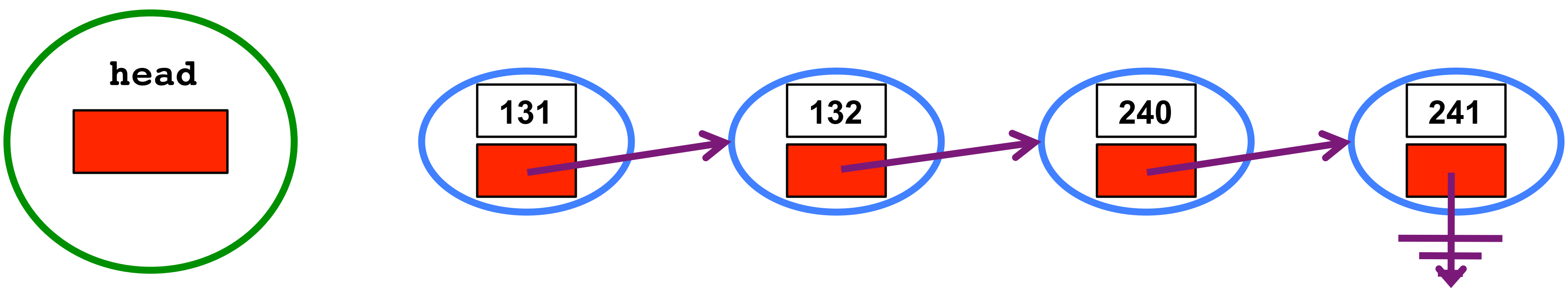
9.8 A `LinkedList` implementation of `List`

- To simply things
  - Let's assume the parametric type `T` is `Integer`
  - Our implementation will work for any type `T`



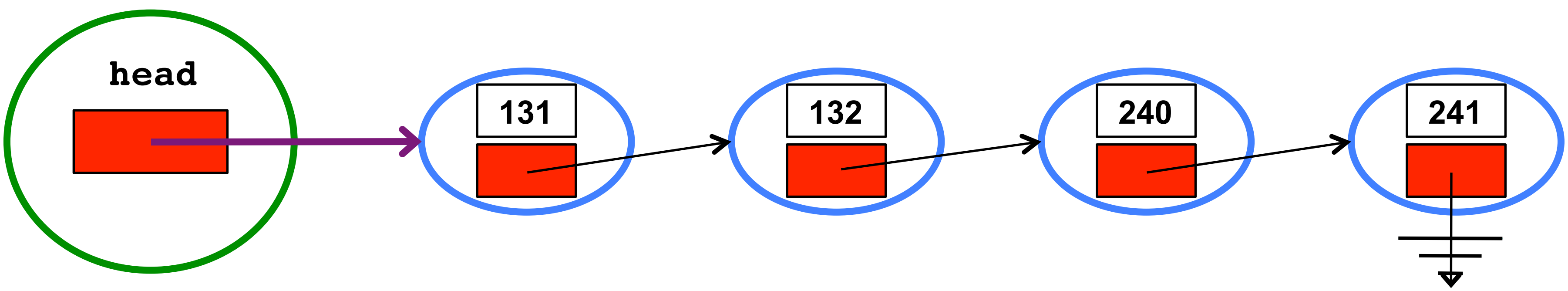
9.8 A `LinkedList` implementation of `List`

- In our singly-linked list implementation
  - Each `ListNode` points to the next element in the list



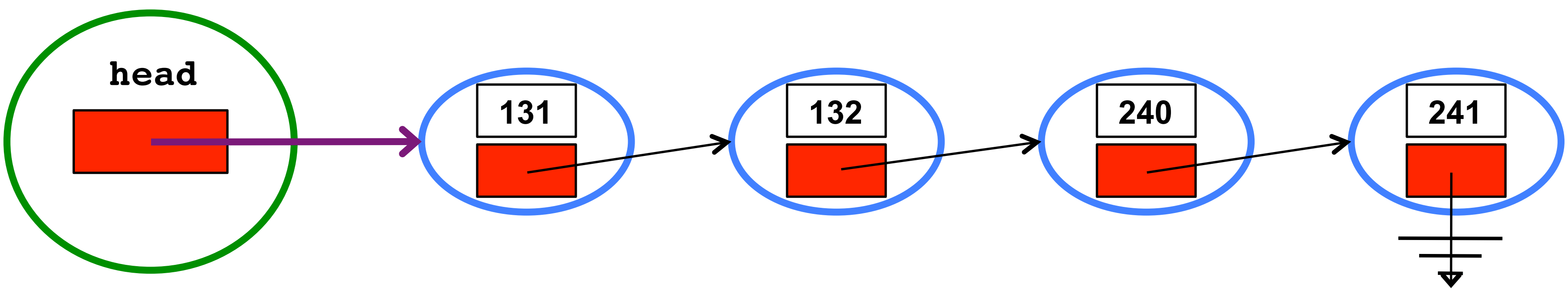
9.8 A `LinkedList` implementation of `List`

- In our singly-linked list implementation
  - Each `ListNode` points to the next element in the list
  - The head pointer in `SinglyLinkedList` points to the beginning of the list



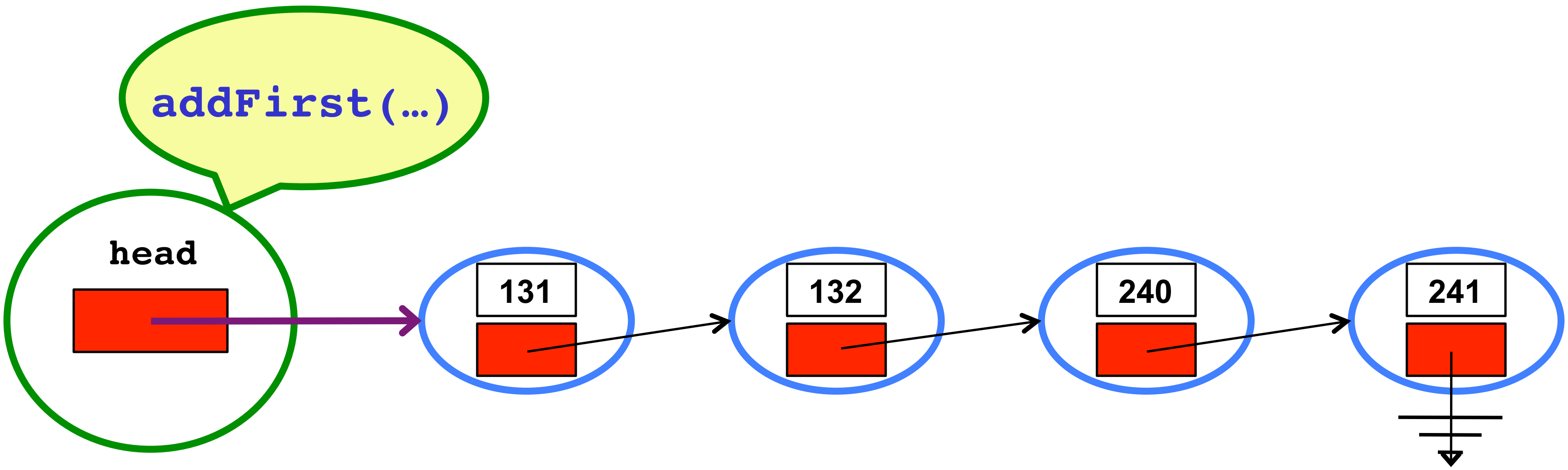
9.8 A `LinkedList` implementation of `List`

- **Note**
  - The `SinglyLinkedList` object implements the methods of interest



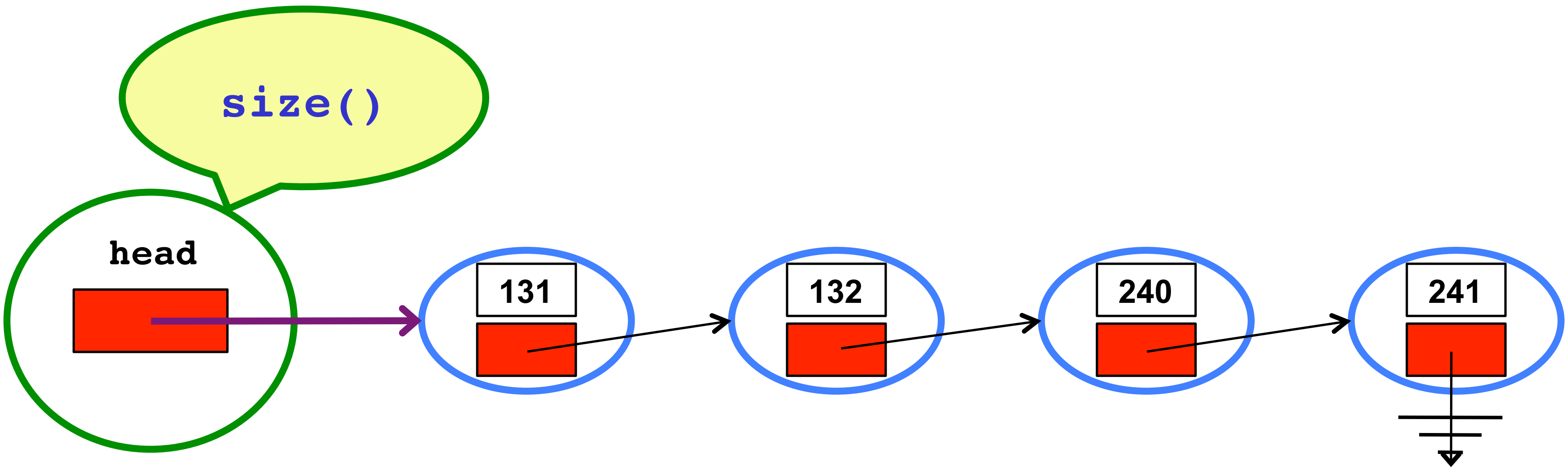
9.8 A `LinkedList` implementation of `List`

- **Note**
  - The `SinglyLinkedList` object implements the methods of interest



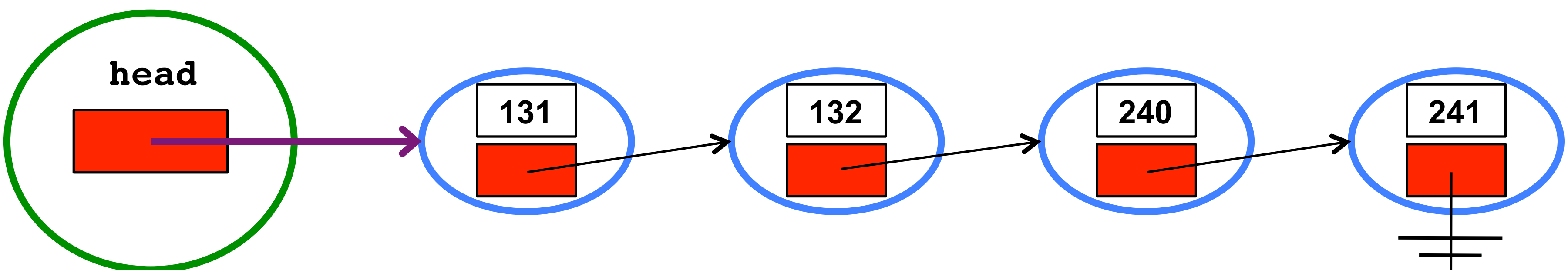
9.8 A `LinkedList` implementation of `List`

- **Note**
  - The `SinglyLinkedList` object implements the methods of interest



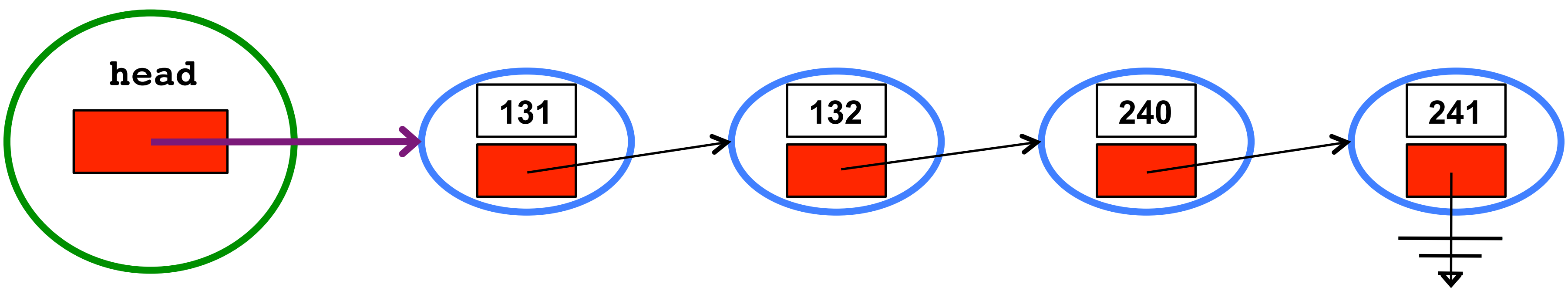
9.8 A `LinkedList` implementation of `List`

- **Note**
  - **The `SinglyLinkedList` object implements the methods of interest**
    - It does its work by following and processing the `ListNode` objects



9.8 A `LinkedList` implementation of `List`

- **Note**
  - **The `SinglyLinkedList` object implements the methods of interest**
  - **The `ListNode` objects are just containers**
    - They offer no special methods at all



9.8 A `LinkedList` implementation of `List`

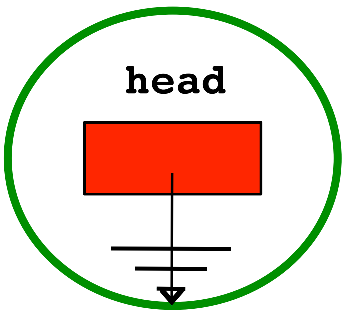
- **We will use the following program and implement methods as needed**

```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

- **Construction**

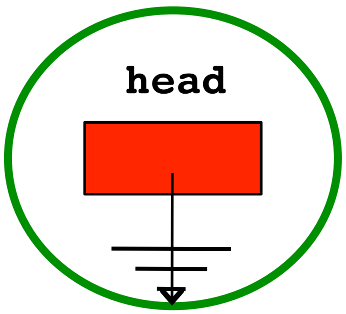
```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
```



9.8 A `LinkedList` implementation of `List`

- Construction
- The only instance variable is `head`
  - Set to `null`
  - Represents the empty list

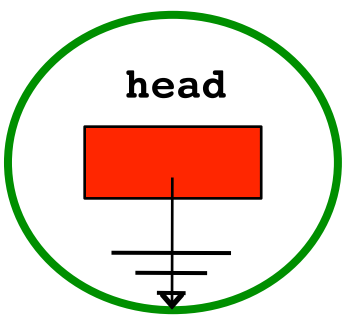
```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
```



End of Oyster

Oyster 8b 9.8 A `LinkedList` implementation of `List`

- Method `addFirst(Integer i)`
  - Special case: adding the element to an empty list




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
```

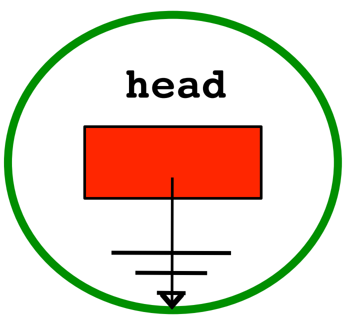
9.8 A `LinkedList` implementation of `List`

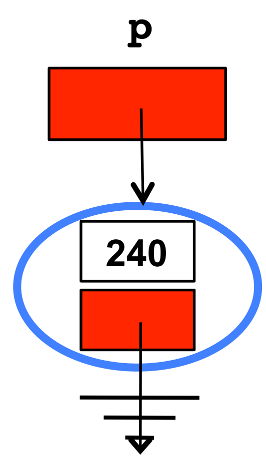
- Method `addFirst(Integer i)`
  - Special case: adding the element to an empty list
    - Create a new `ListNode` (referenced by `p`)


```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
```

9.8 A `LinkedList` implementation of `List`

- Method `addFirst(Integer i)`
  - Special case: adding the element to an empty list
    - Create a new `ListNode` (referenced by `p`)
    - Set its `value` to `i` (its `next` is `null`)

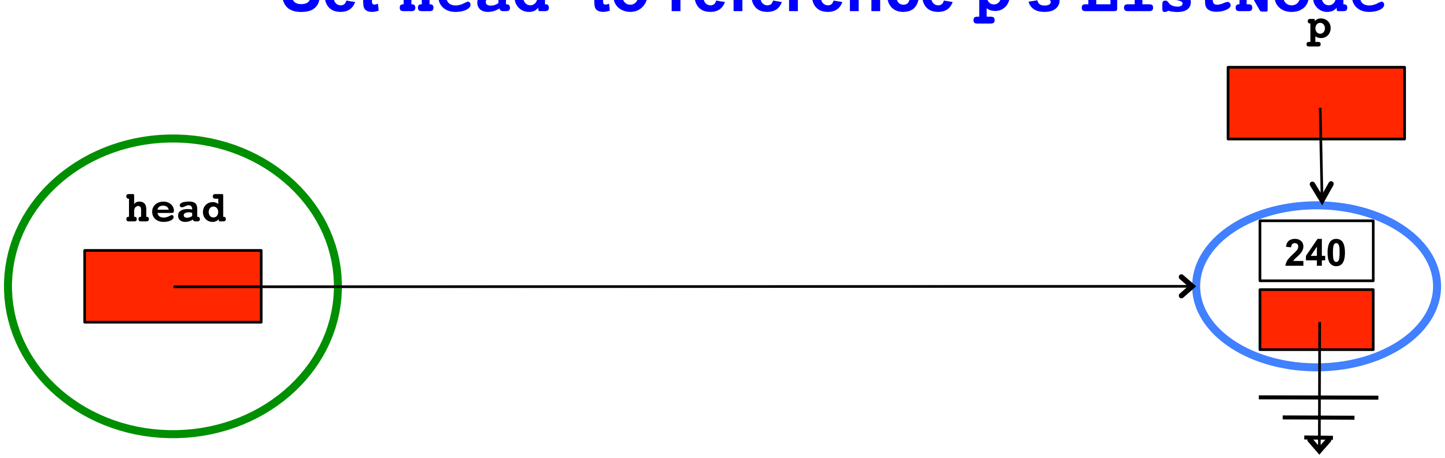





```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
```

9.8 A `LinkedList` implementation of `List`

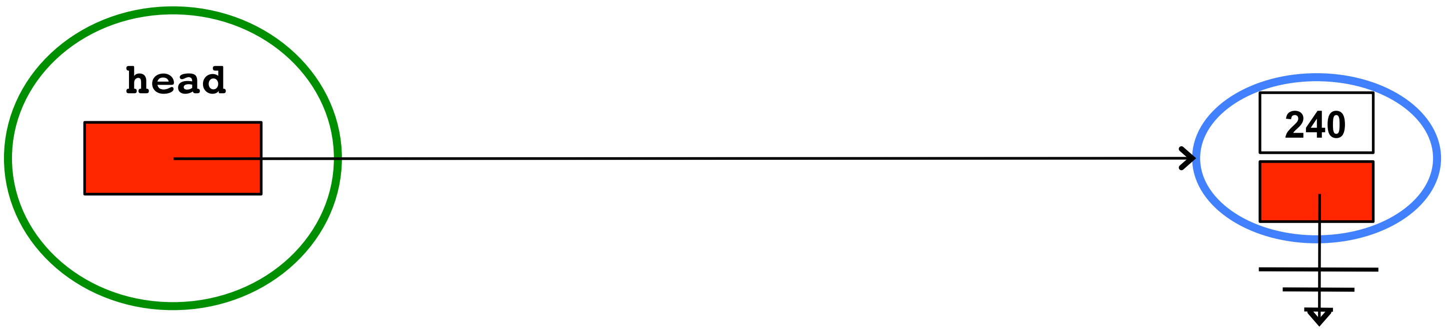
- Method `addFirst(Integer i)`
  - Special case: adding the element to an empty list
    - Create a new `ListNode` (referenced by `p`)
    - Set its `value` to `i` (its `next` is `null`)
    - Set `head` to reference `p`'s `ListNode`




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
```

9.8 A `LinkedList` implementation of `List`

- Method `addFirst(Integer i)`
  - Result shown below




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
```

9.8 A `LinkedList` implementation of `List`

- Method `addFirst(Integer i)`
  - The list is no longer empty


```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
```

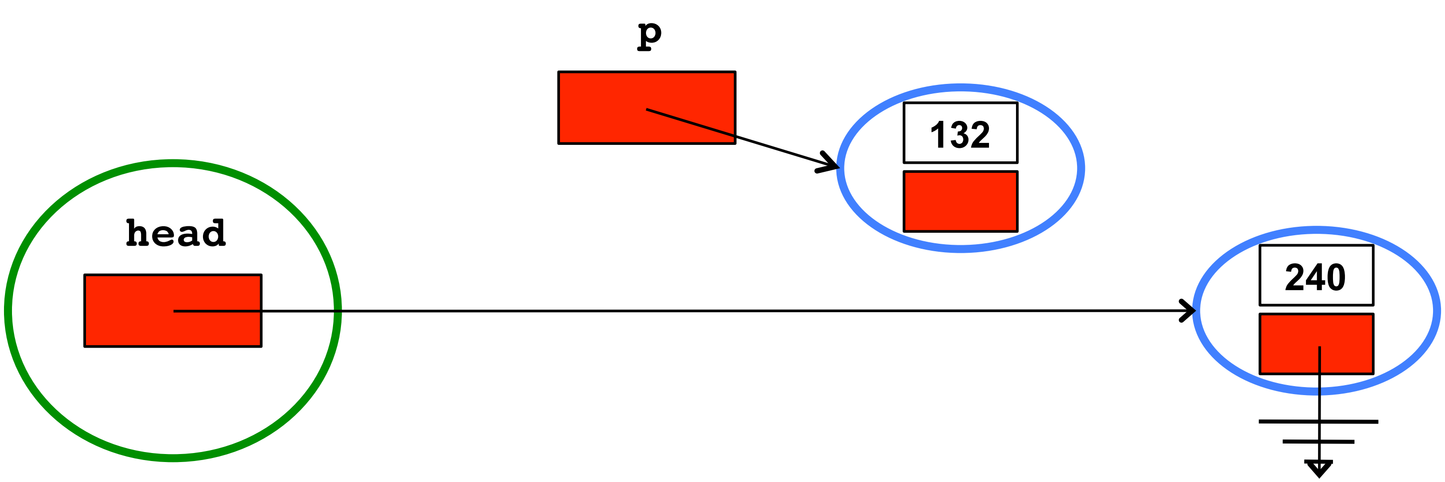
9.8 A `LinkedList` implementation of `List`

- Method `addFirst(Integer i)`
  - The list is no longer empty


```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
```

9.8 A `LinkedList` implementation of `List`

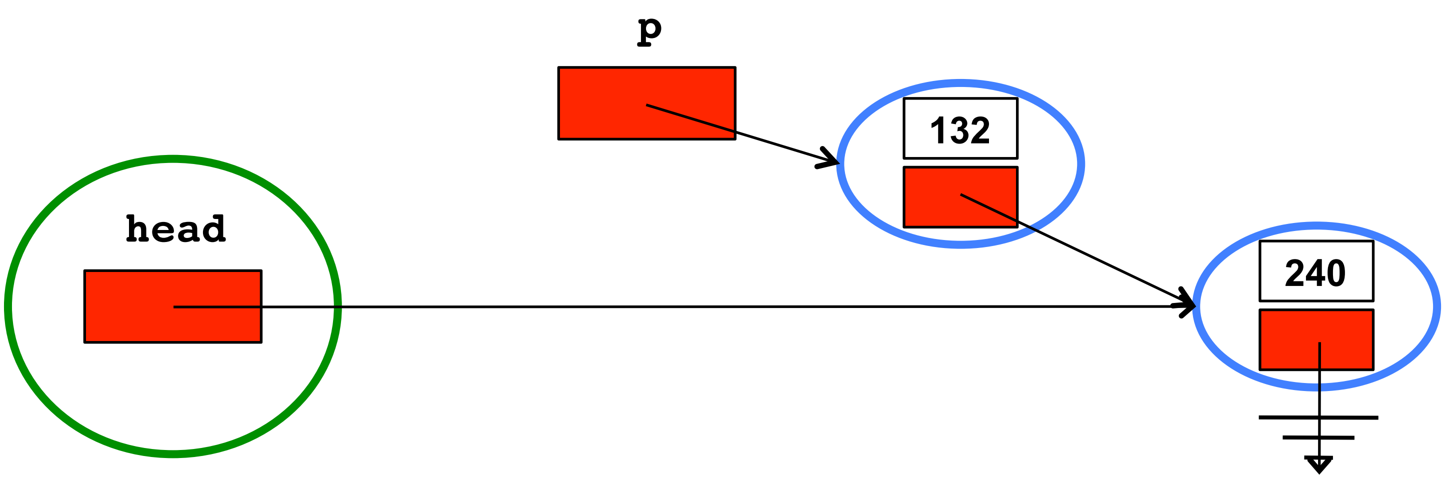
- Method `addFirst(Integer i)`
  - Adding to a non-empty list
    - Create a new `ListNode` and set its `value` to 132




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
```

9.8 A `LinkedList` implementation of `List`

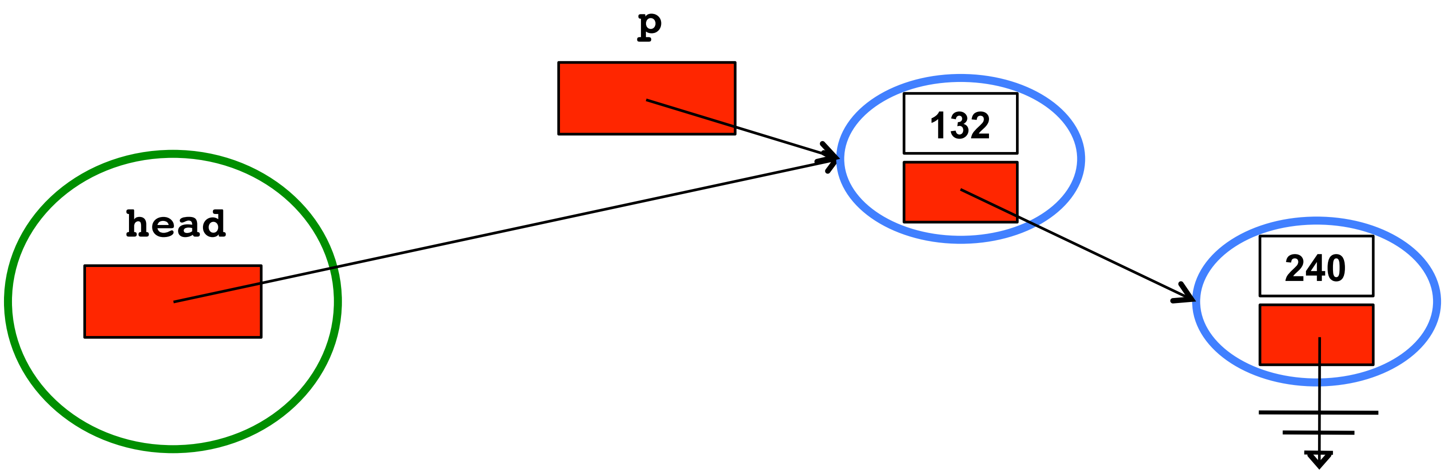
- Method `addFirst(Integer i)`
  - Adding to a non-empty list
    - Create a new `ListNode` and set its `value` to 132
    - Its `next` references `head`'s `ListNode`




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
```

9.8 A `LinkedList` implementation of `List`

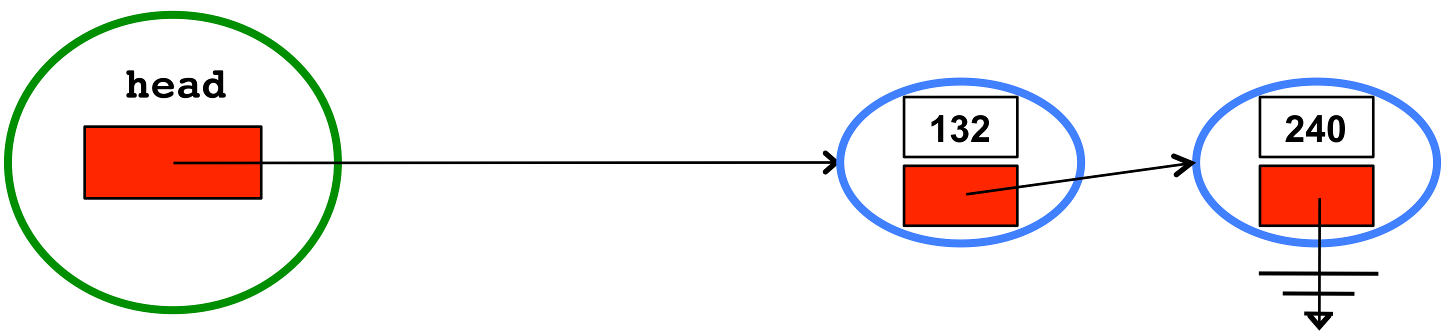
- Method `addFirst(Integer i)`
  - Adding to a non-empty list
    - Create a new `ListNode` and set its `value` to 132
    - Its `next` references `head`'s `ListNode`
    - `head` references `p's ListNode`




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
```

9.8 A `LinkedList` implementation of `List`

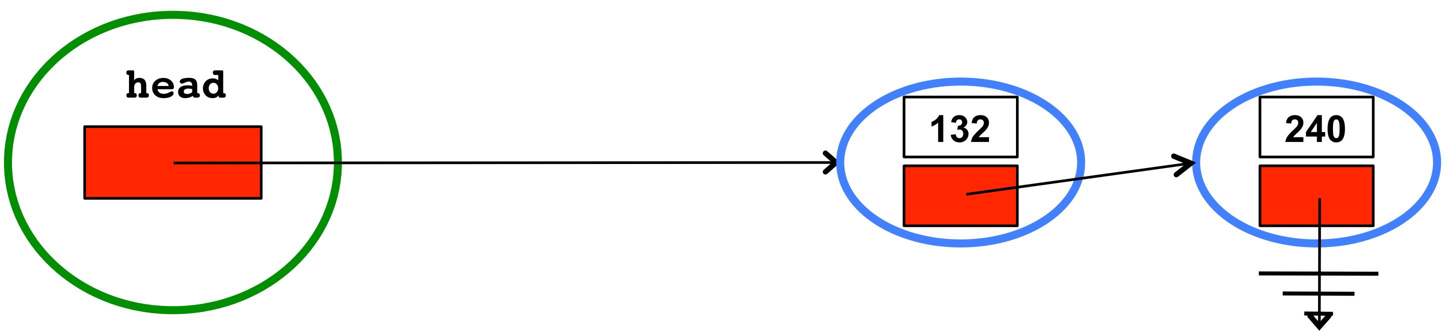
- Method `addFirst(Integer i)`
  - Result shown below




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
```

9.8 A `LinkedList` implementation of `List`

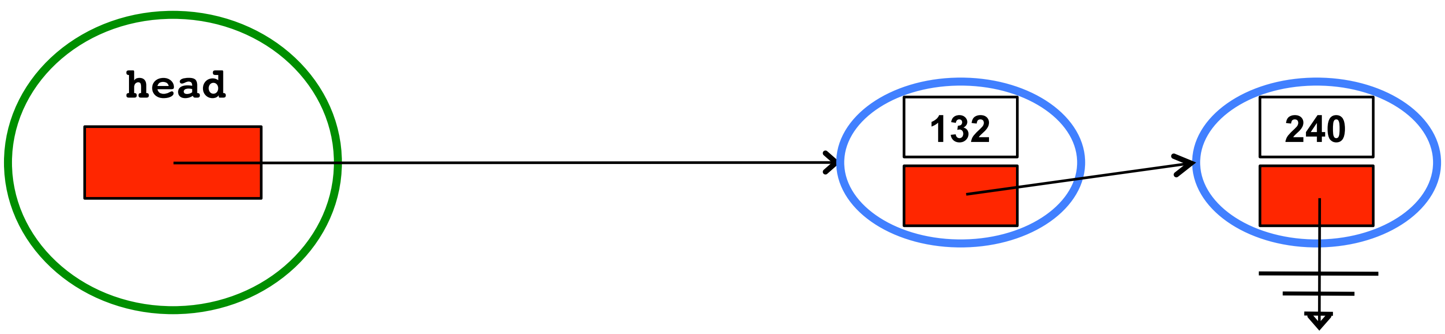
- Method `addFirst(Integer i)`
  - Result shown below




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
```

9.8 A `LinkedList` implementation of `List`

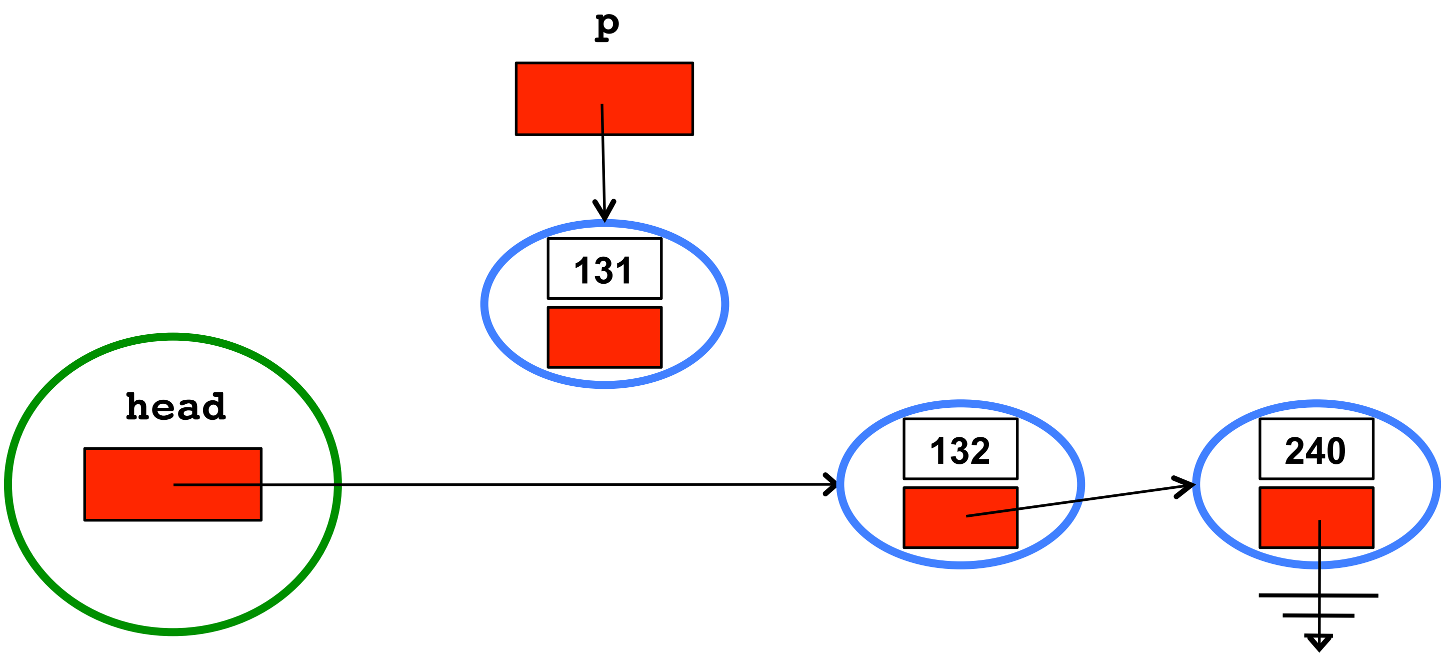
- Method `addFirst(Integer i)`
  - Same process as last time




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
```

9.8 A `LinkedList` implementation of `List`

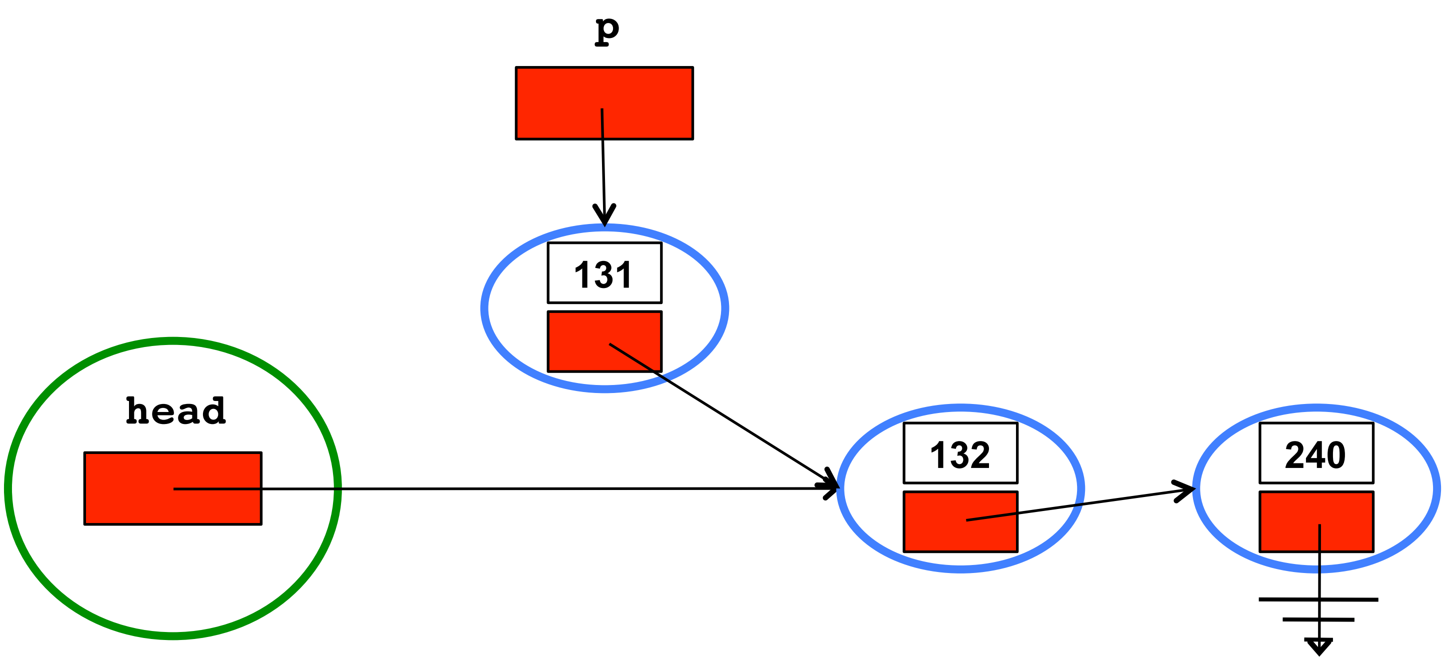
- Method `addFirst(Integer i)`
  - Same process as last time




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
```

9.8 A `LinkedList` implementation of `List`

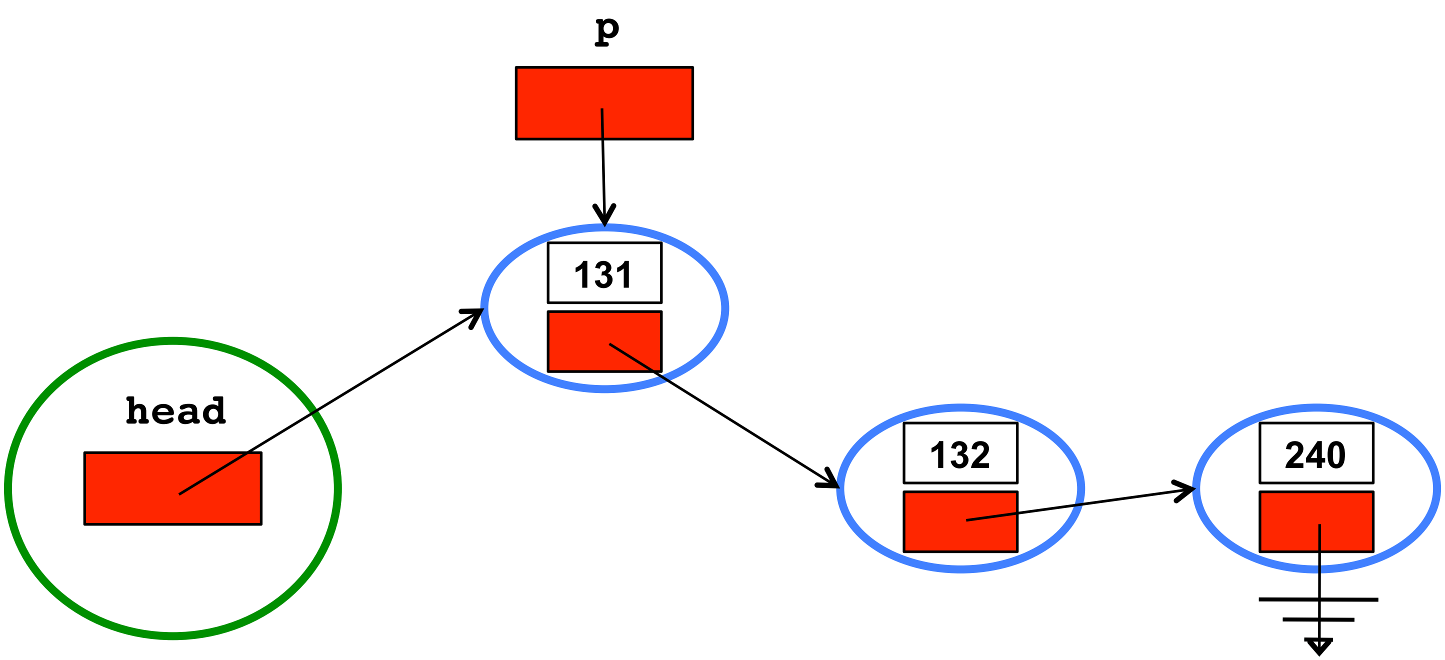
- Method `addFirst(Integer i)`
  - Same process as last time




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
```

9.8 A `LinkedList` implementation of `List`

- Method `addFirst(Integer i)`
  - Same process as last time

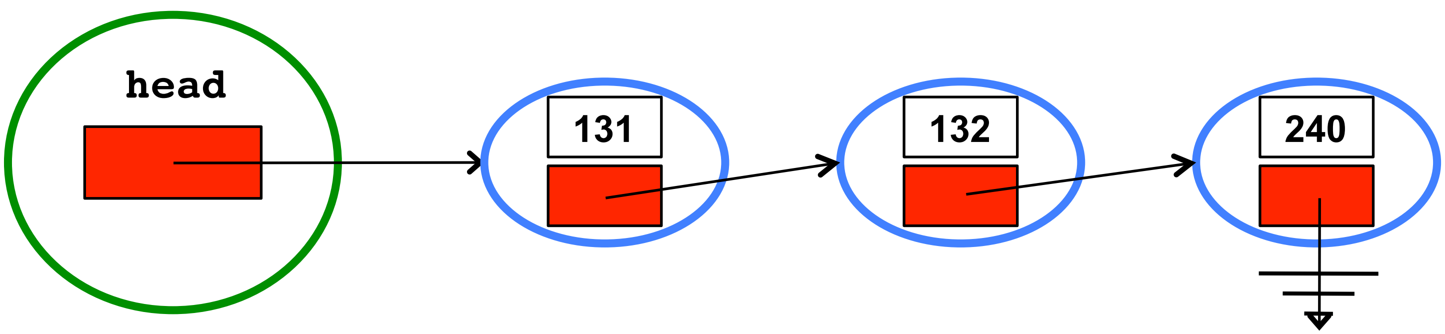



```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
```


End of Oyster

9.8 A `LinkedList` implementation of `List`

- Method `addFirst(Integer i)`
  - Result shown below

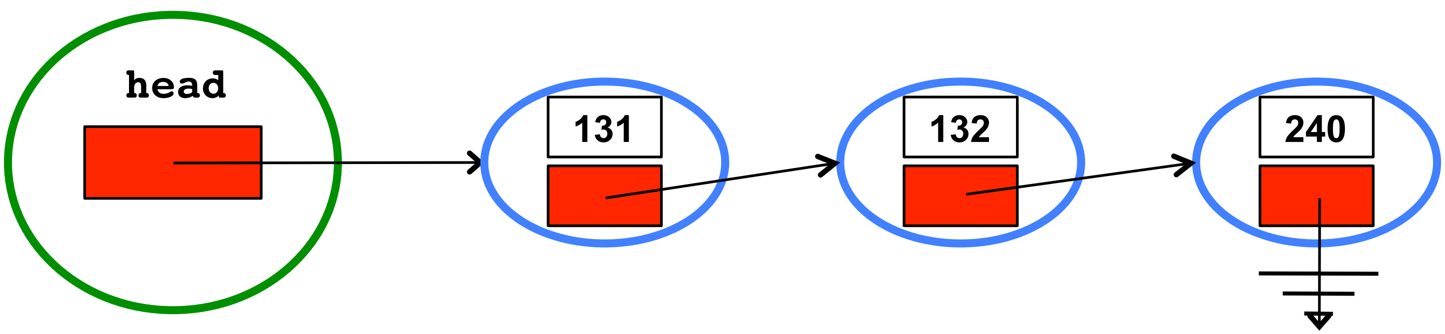



```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
```

Oyster 8c

9.8 A `LinkedList` implementation of `List`

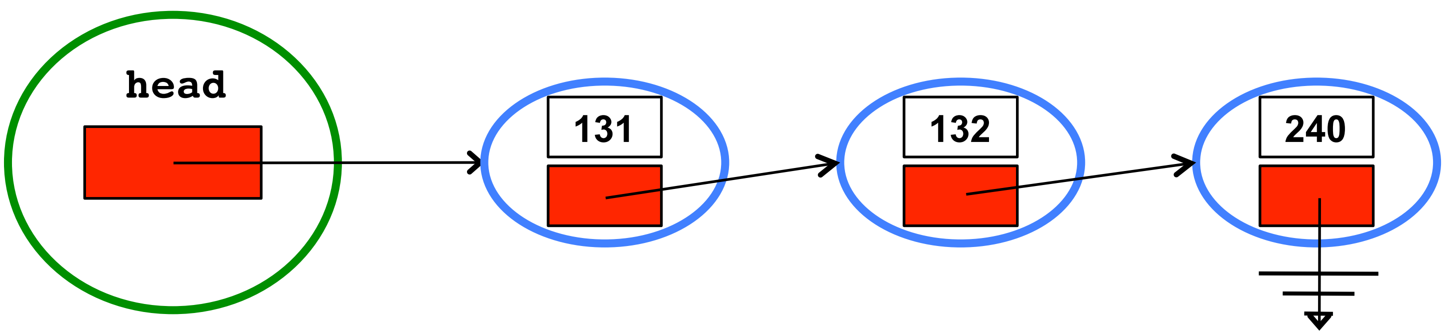
- Method `addLast(Integer i)`
  - This could happen to an empty list




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

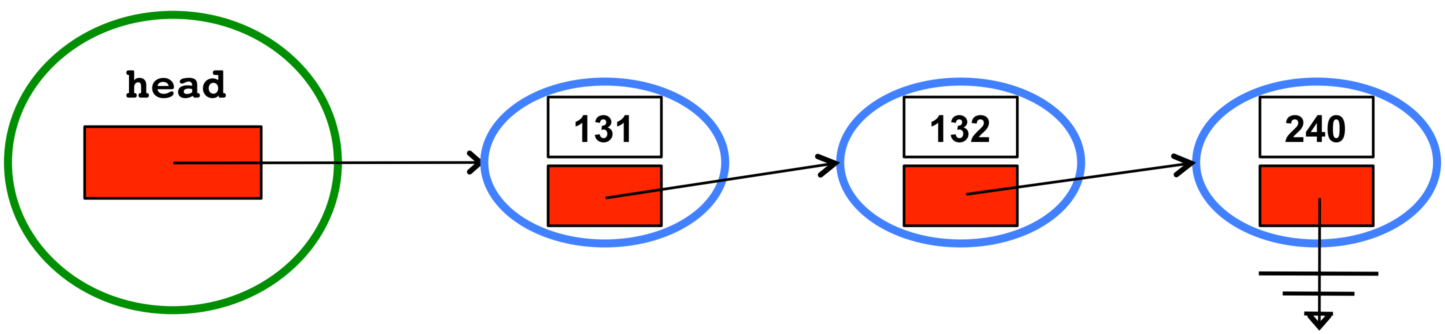
- Method `addLast(Integer i)`
  - This could happen to an empty list
  - Note: adding to the beginning or end of an empty list produces the same result
  - Use the principle of *reduction*




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

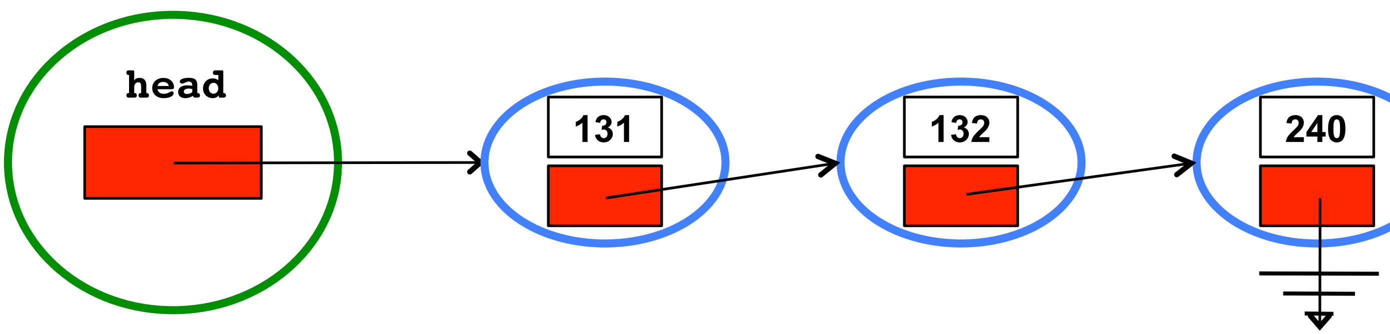
- Method `addLast(Integer i)`
  - This could happen to an empty list
  - If so, call `addFirst(i)`




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

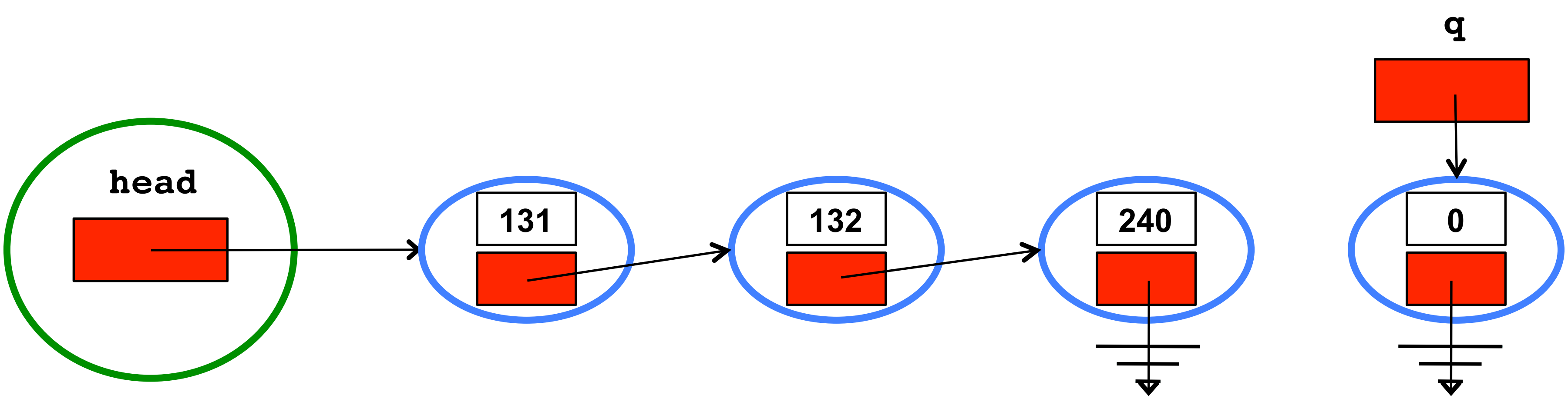
- Method `addLast(Integer i)`
  - Otherwise




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

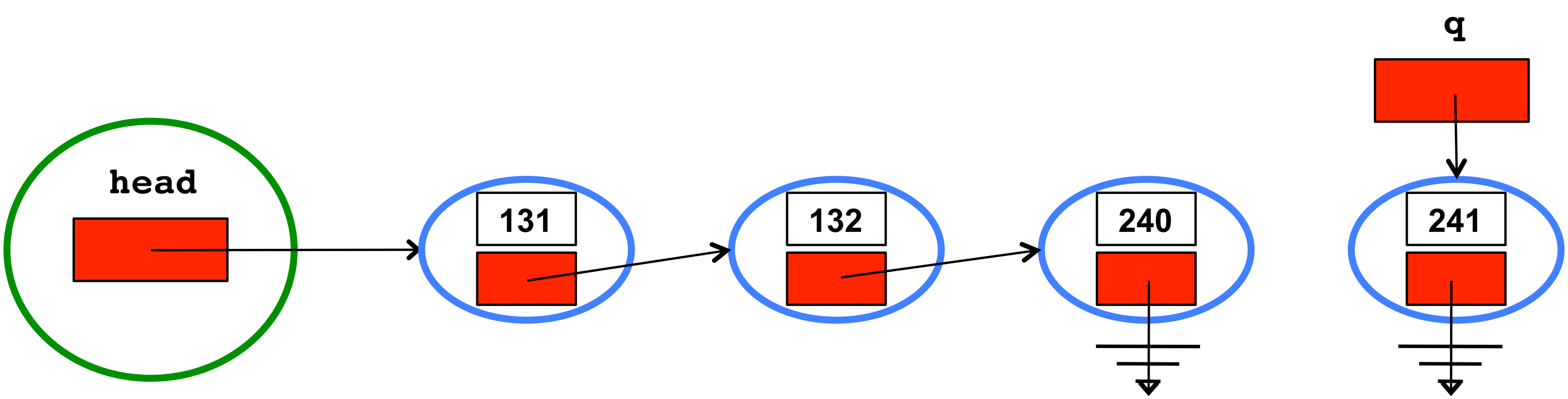
- Method `addLast(Integer i)`
  - Otherwise
    - Make the new `ListNode` (call it `q`)




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

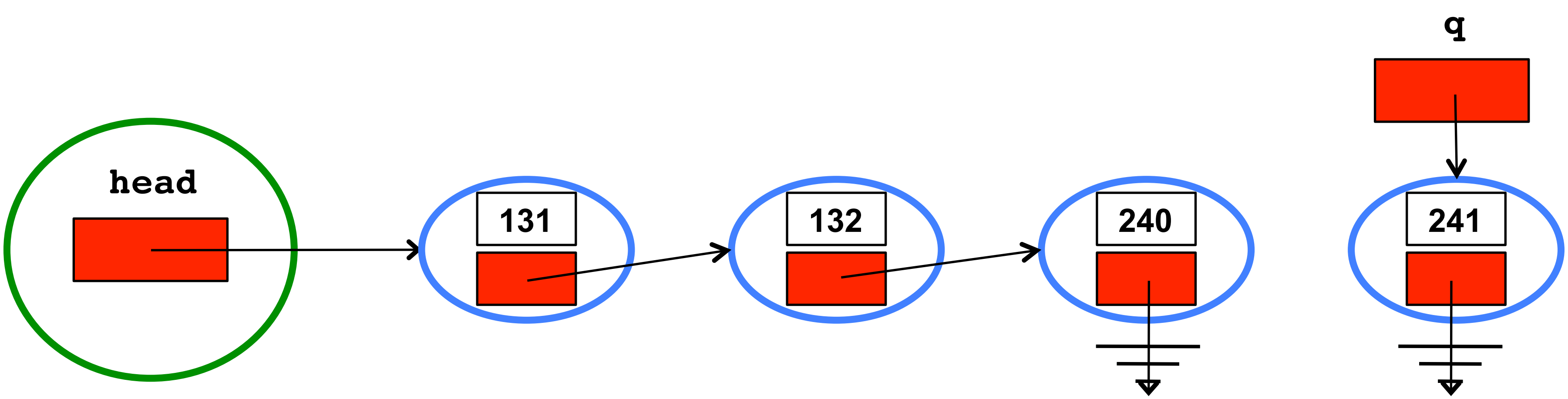
- Method `addLast(Integer i)`
  - Otherwise
    - Make the new `ListNode` and set its `value`




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

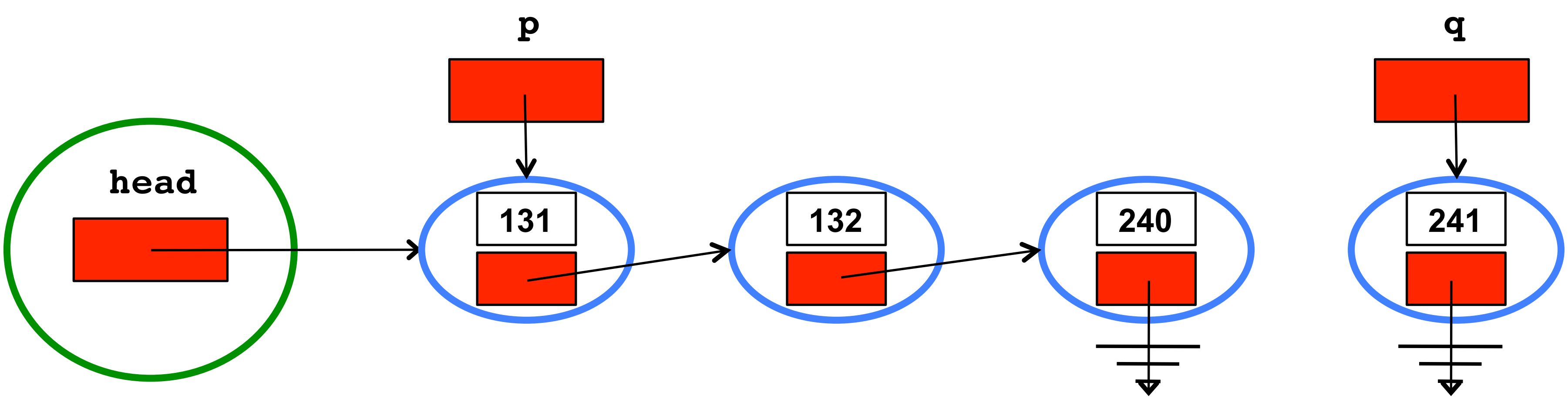
- Method `addLast(Integer i)`
  - Otherwise
    - Make the new `ListNode` and set its `value`
    - We now search for the end of the list




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

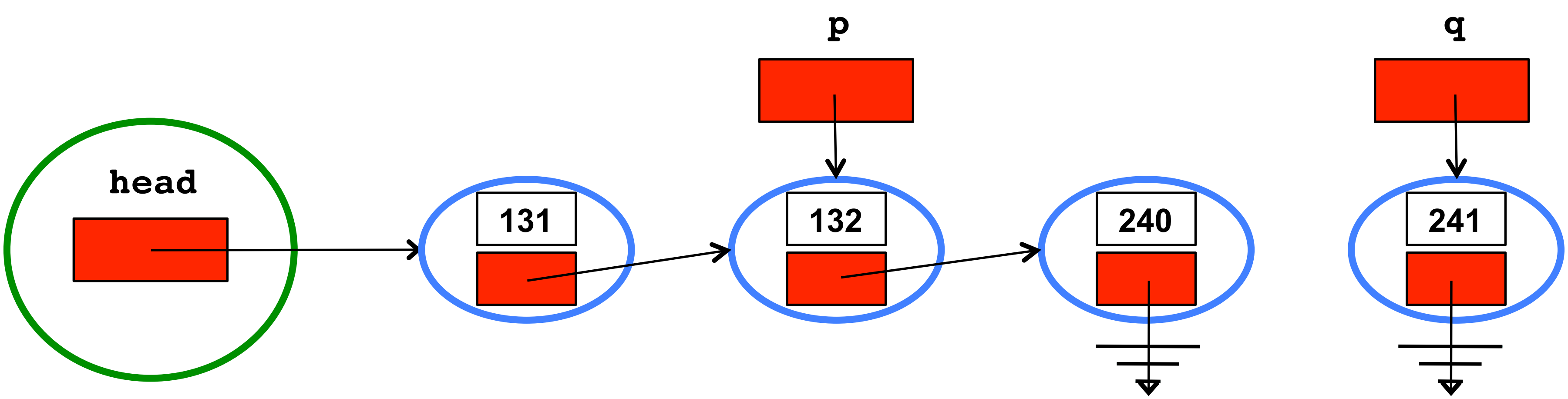
- Method `addLast(Integer i)`
  - Otherwise
    - Make the new `ListNode` and set its `value`
    - We now search for the end of the list (using `p`)




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

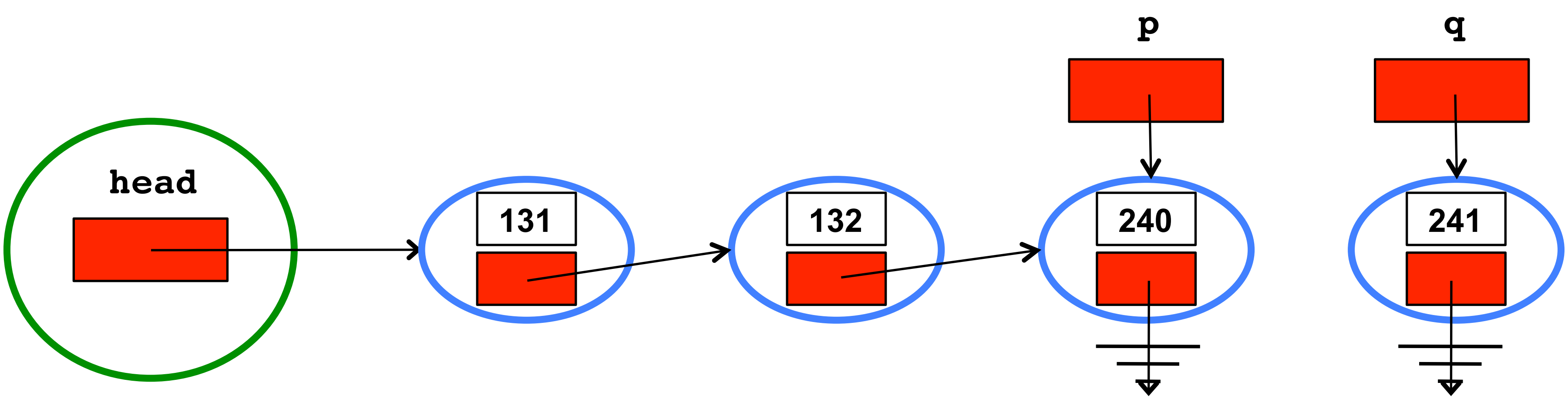
- Method `addLast(Integer i)`
  - Otherwise
    - Make the new `ListNode` and set its `value`
    - We now search for the end of the list (using `p`)




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

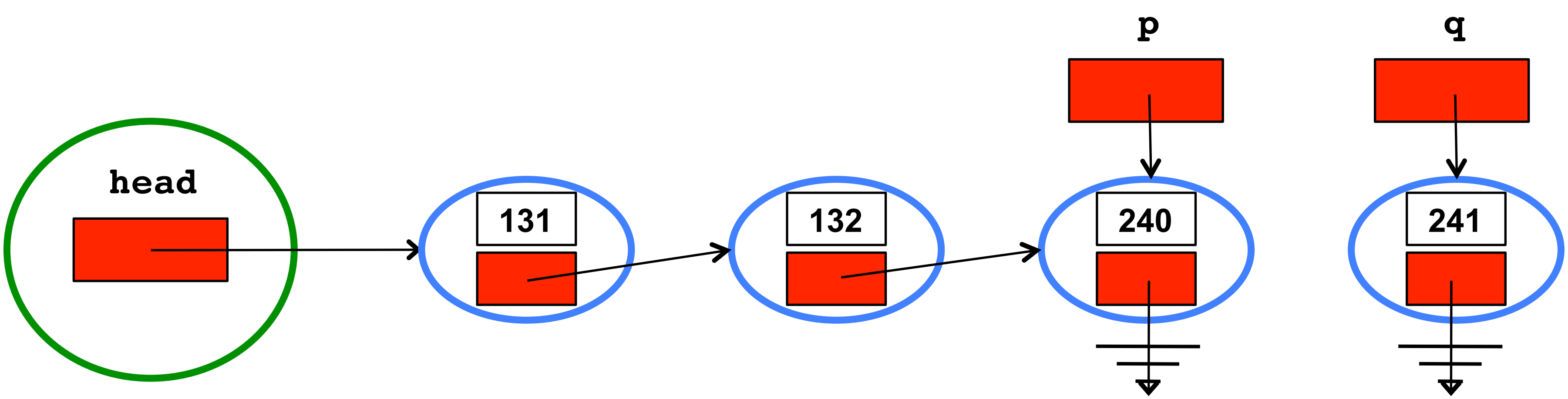
- Method `addLast(Integer i)`
  - Otherwise
    - Make the new `ListNode` and set its `value`
    - We now search for the end of the list (using `p`)




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

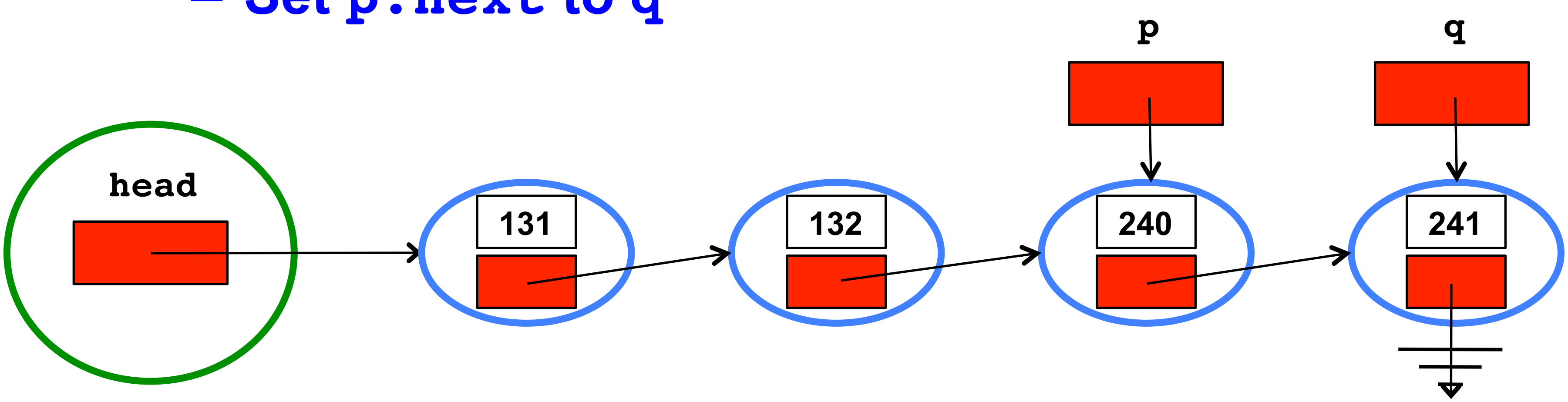
- Method `addLast(Integer i)`
  - Otherwise
    - Make the new `ListNode` and set its `value`
    - When `p` references the end




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

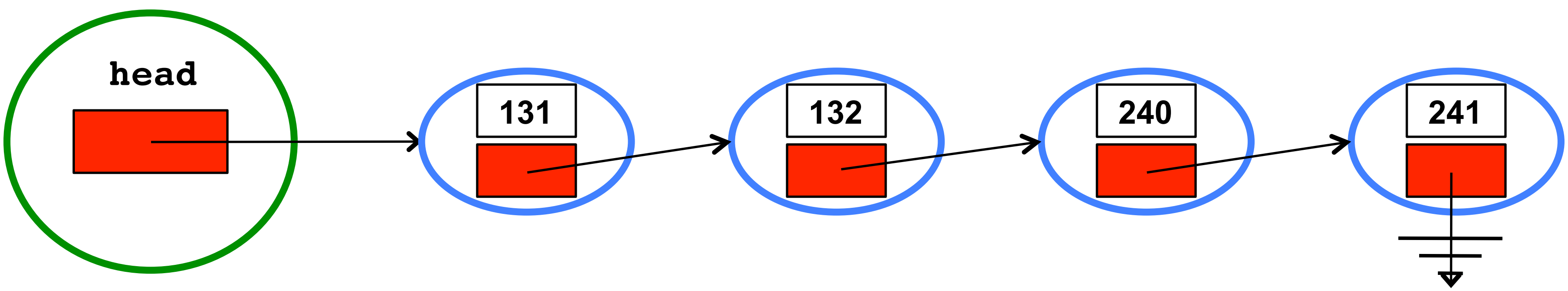
- Method `addLast(Integer i)`
  - Otherwise
    - Make the new `ListNode` and set its `value`
    - When `p` references the end
    - Set `p.next` to `q`




```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

9.8 A `LinkedList` implementation of `List`

- Method **`addLast(Integer i)`**
  - Result shown below



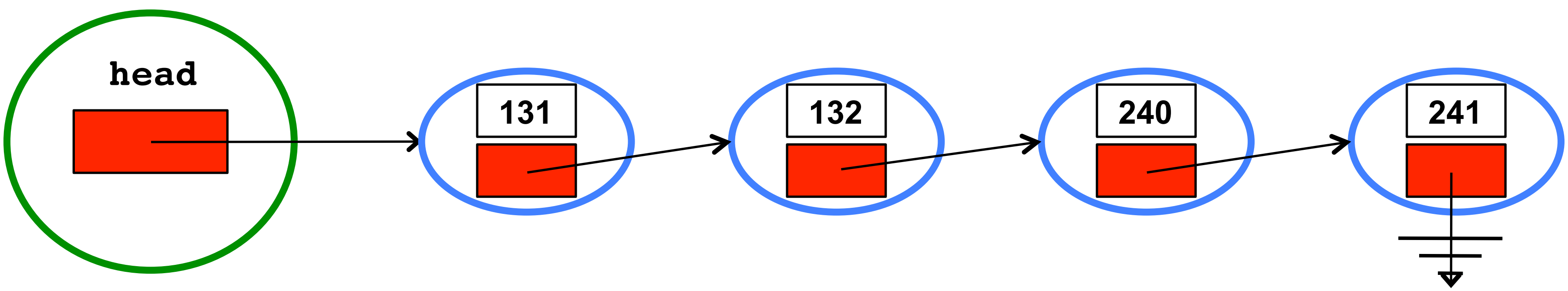

```
SinglyLinkedList<Integer> list = new SinglyLinkedList<Integer>();
list.addFirst(240);
list.addFirst(132);
list.addFirst(131);
list.addLast(241);
```

End of Oyster

Oyster 8d

9.8 A `LinkedList` implementation of `List`

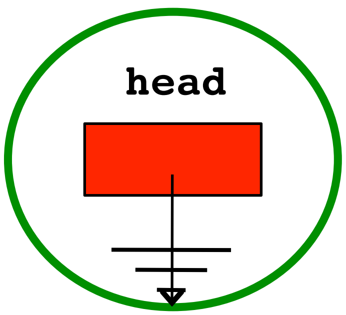
- Method `remove(Integer i)`
  - Finds and removes the first node that has `i` as its `value`




```
list.remove(132);
```

9.8 A `LinkedList` implementation of `List`

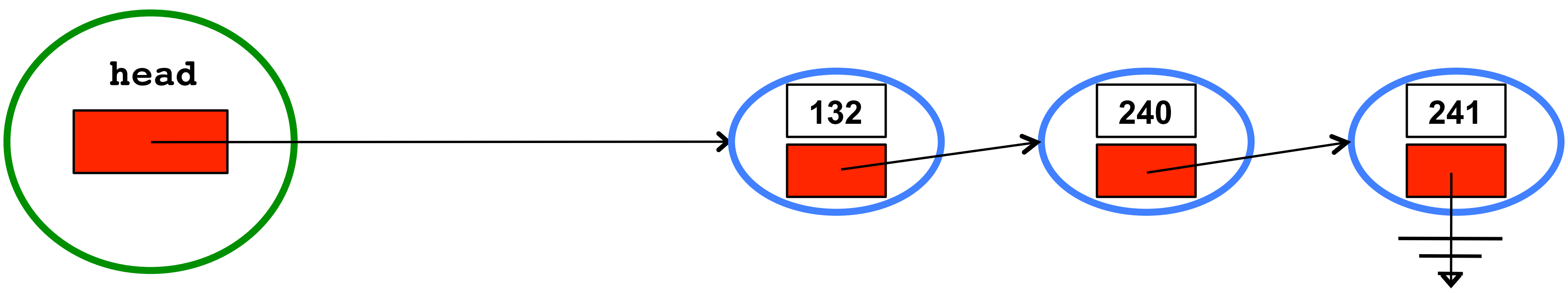
- **Method `remove(Integer i)`**
  - Special case: empty list—nothing to do




```
list.remove(132);
```

9.8 A `LinkedList` implementation of `List`

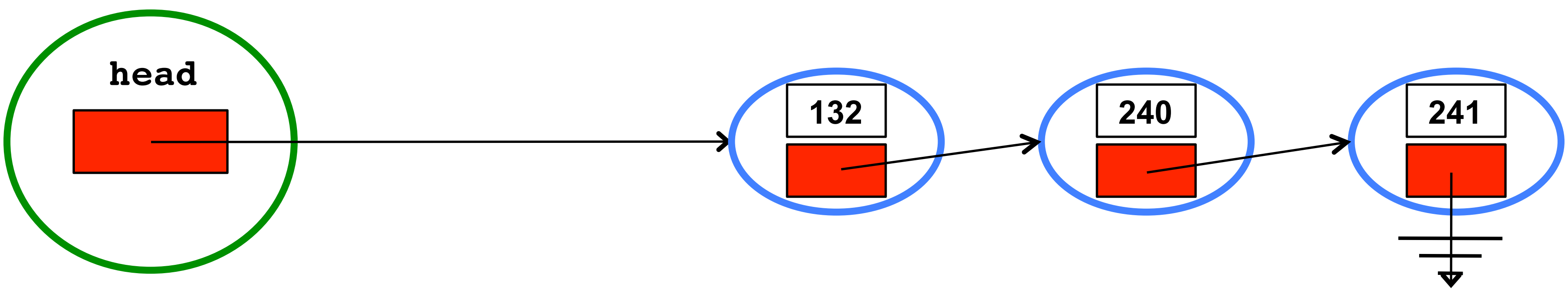
- Method `remove(Integer i)`
  - Another special case: item is found first in the list




```
list.remove(132);
```

9.8 A `LinkedList` implementation of `List`

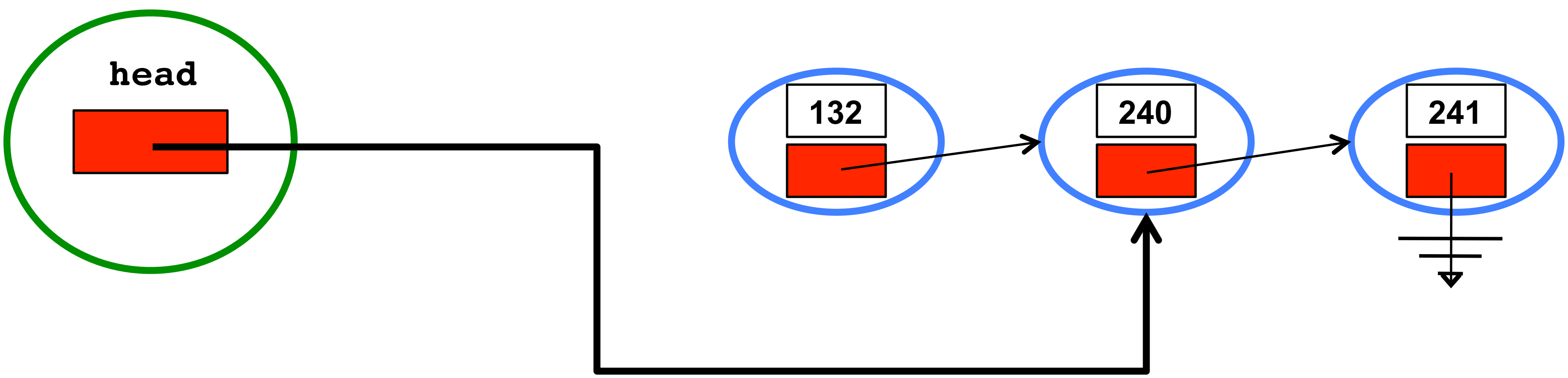
- Method `remove(Integer i)`
  - Another special case: item is found first in the list
  - Remove by setting `head = head.next`




```
list.remove(132);
```

9.8 A `LinkedList` implementation of `List`

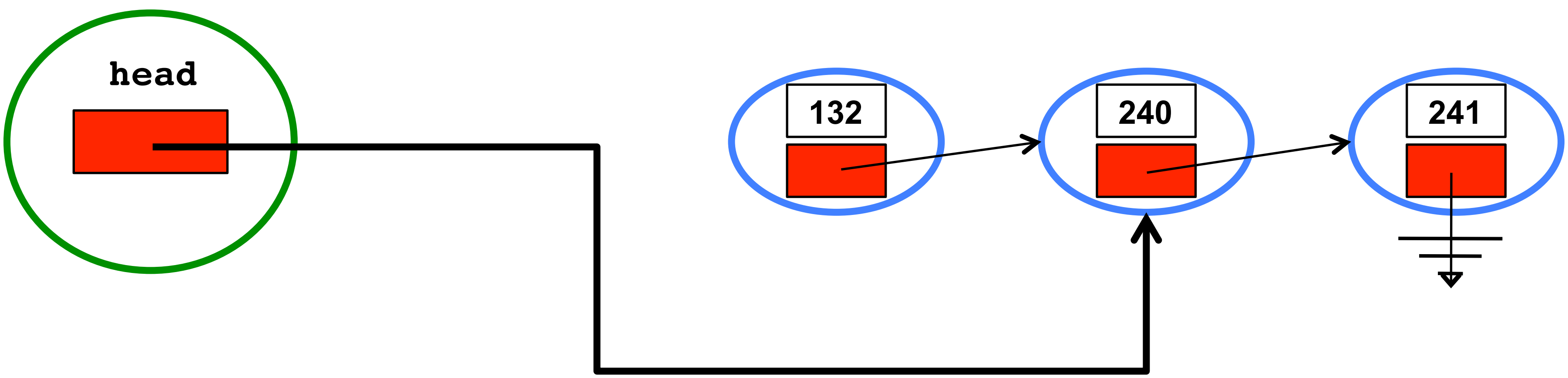
- Method `remove(Integer i)`
  - Another special case: item is found first in the list
  - Remove by setting `head = head.next`




```
list.remove(132);
```

9.8 A `LinkedList` implementation of `List`

- Method `remove(Integer i)`
  - Another special case: item is found first in the list
  - Remove by setting `head = head.next`
  - The deleted node is no longer part of this list




```
list.remove(132);
```

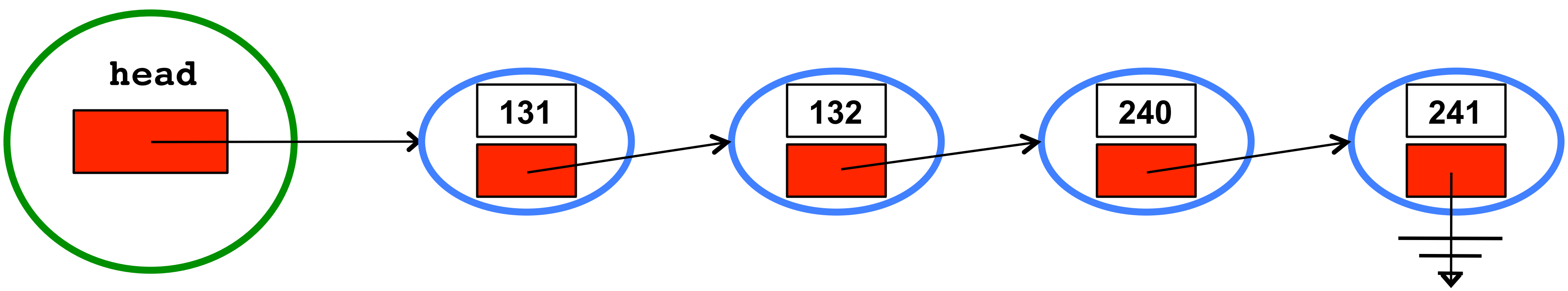
9.8 A `LinkedList` implementation of `List`

- Method `remove(Integer i)`
  - Another special case: item is found first in the list
  - Remove by setting `head = head.next`
  - The deleted node is no longer part of this list


```
list.remove(132);
```

9.8 A `LinkedList` implementation of `List`

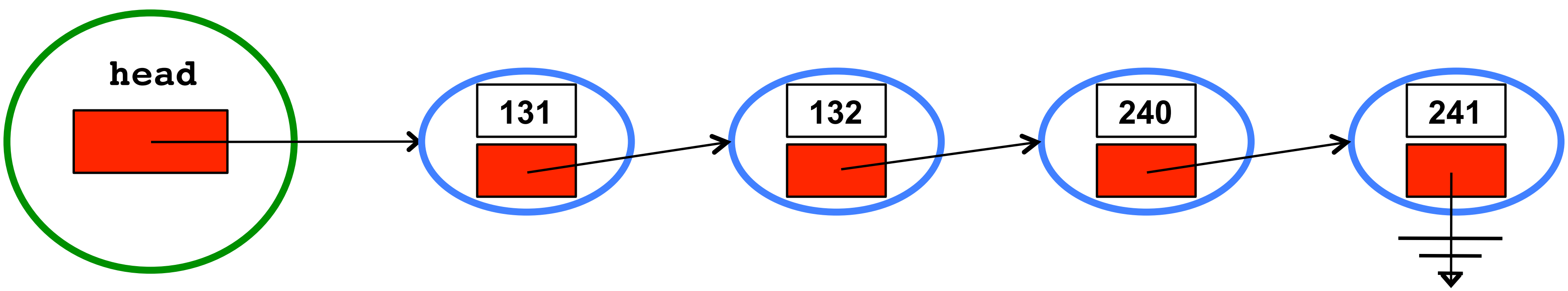
- Method `remove(Integer i)`
  - Otherwise we have to look for the item in the middle of the list




```
list.remove(132);
```

9.8 A `LinkedList` implementation of `List`

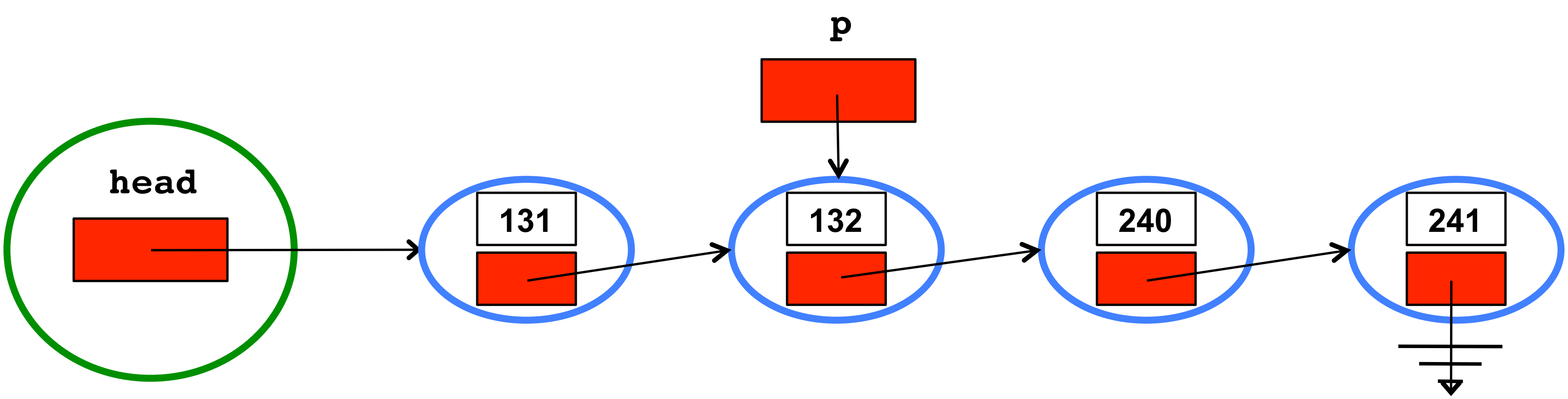
- Method `remove(Integer i)`
  - Otherwise we have to look for the item in the middle of the list
  - Suppose the item is found and referenced by p




```
list.remove(132);
```

9.8 A `LinkedList` implementation of `List`

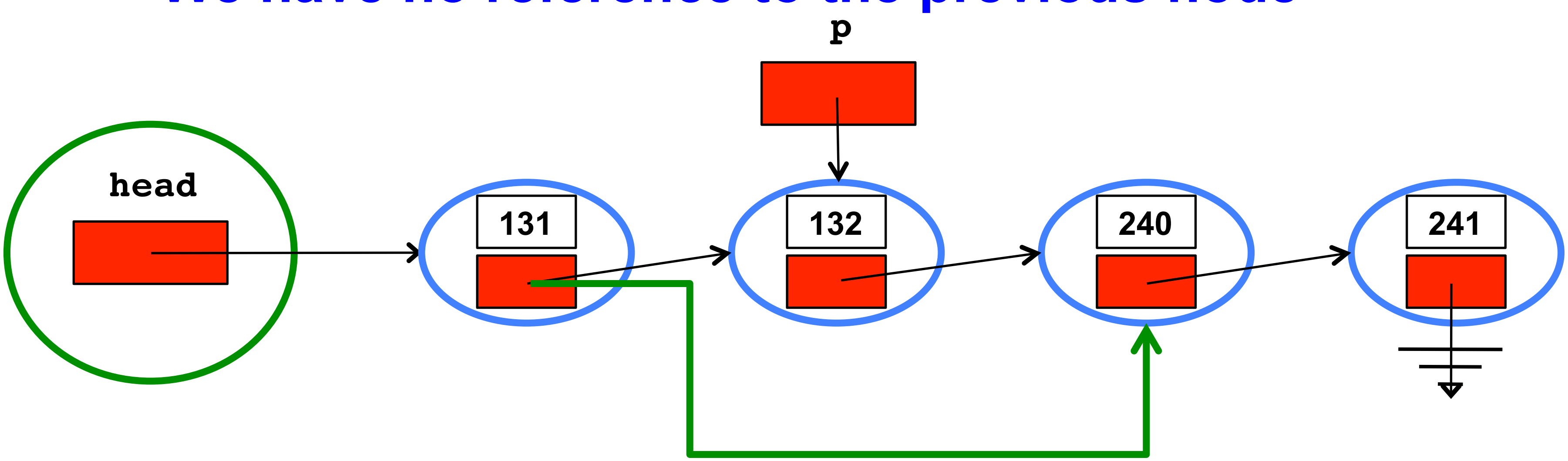
- Method `remove(Integer i)`
  - Otherwise we have to look for the item in the middle of the list
  - Suppose the item is found and referenced by p




```
list.remove(132);
```

9.8 A `LinkedList` implementation of `List`

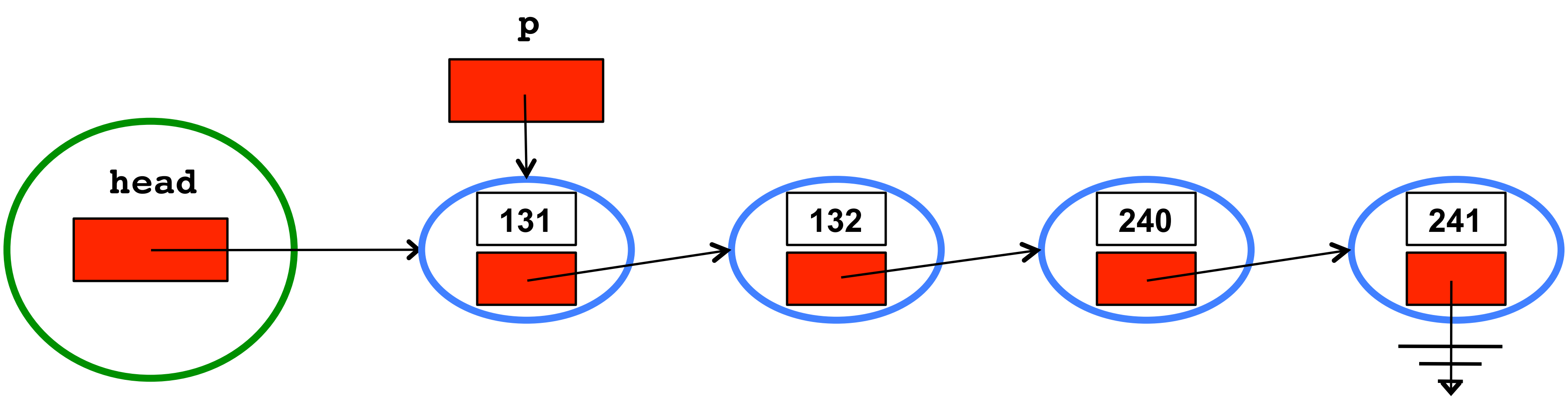
- Method `remove(Integer i)`
  - Otherwise we have to look for the item in the middle of the list
  - Suppose the item is found and referenced by p
  - We have no reference to the previous node




```
list.remove(132);
```

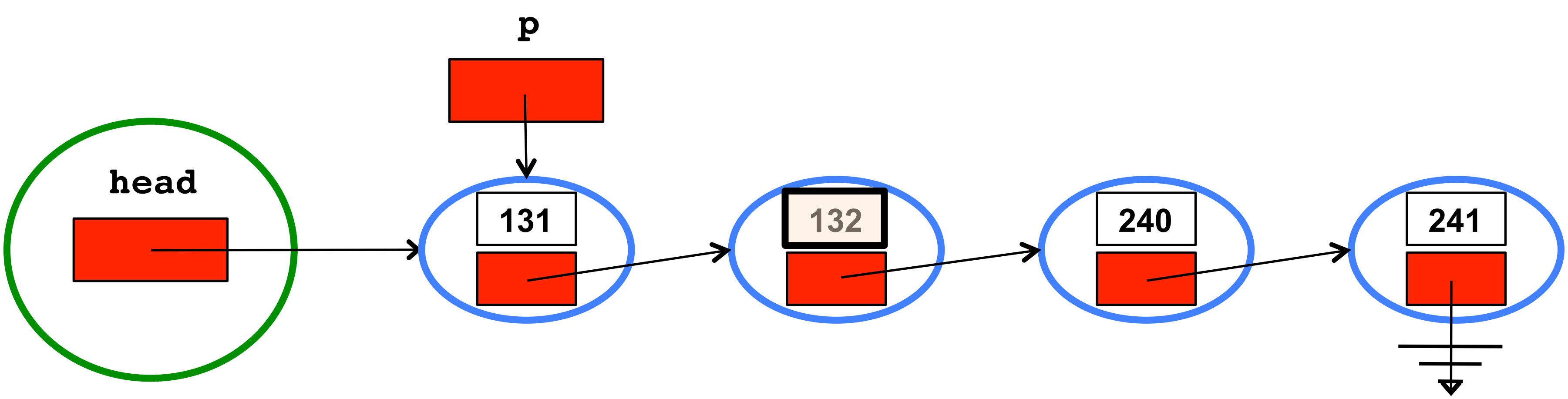
9.8 A `LinkedList` implementation of `List`

- Method `remove(Integer i)`
  - Otherwise we have to look for the item in the middle of the list
  - Trick: look at the node just to the right of `p`



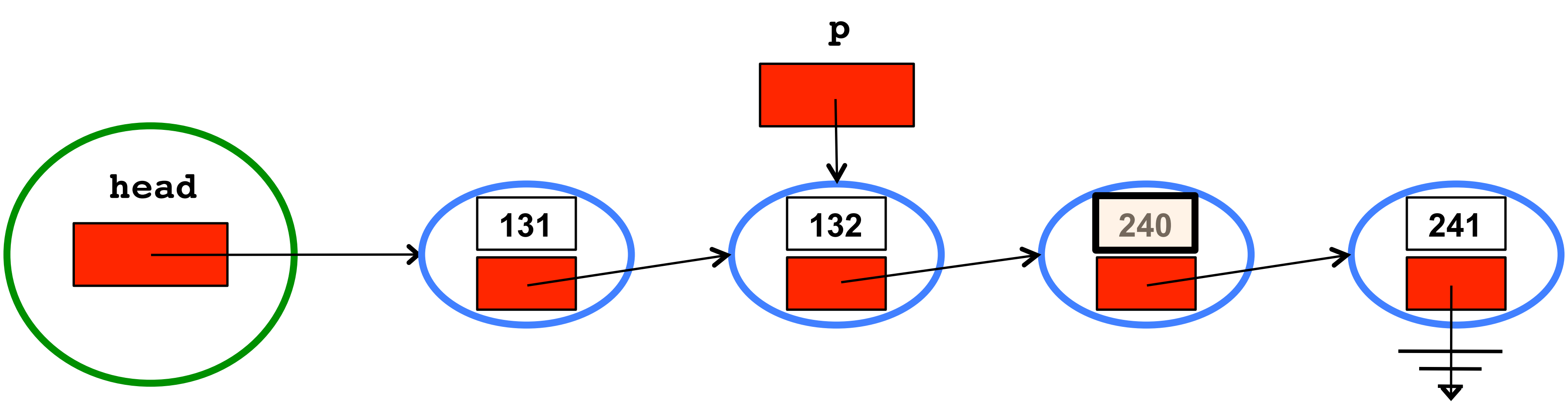
9.8 A `LinkedList` implementation of `List`

- **Method `remove(Integer i)`**
  - Otherwise we have to look for the item in the middle of the list
  - Trick: look at the node just to the right of `p`



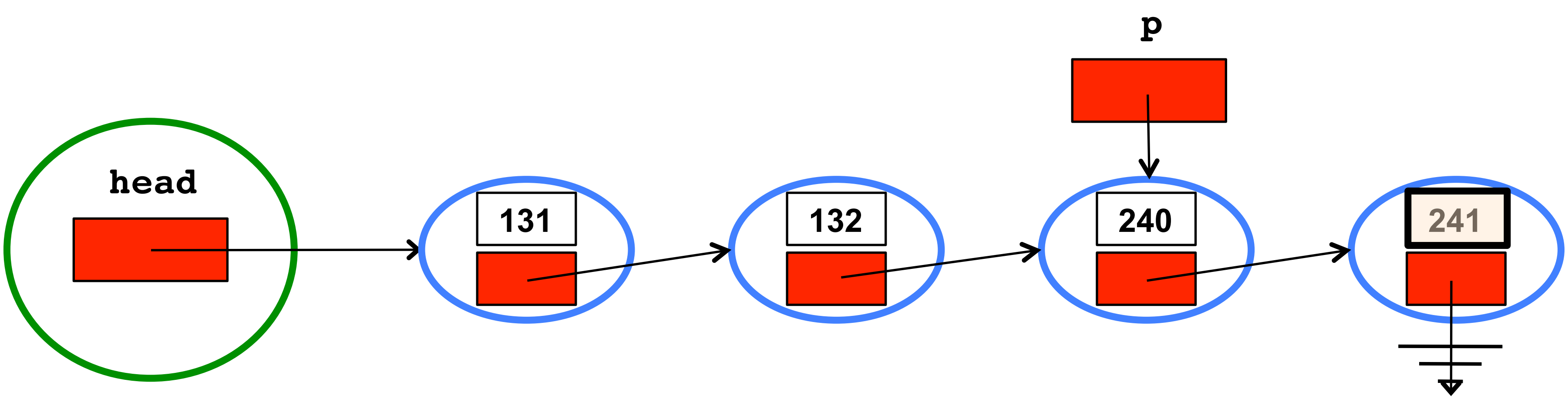
9.8 A `LinkedList` implementation of `List`

- Recall our List ADT:
- And recall our singly-linked list implementation
- The method `remove(int)`
  - Technique: look at `value` one to the right of `p`



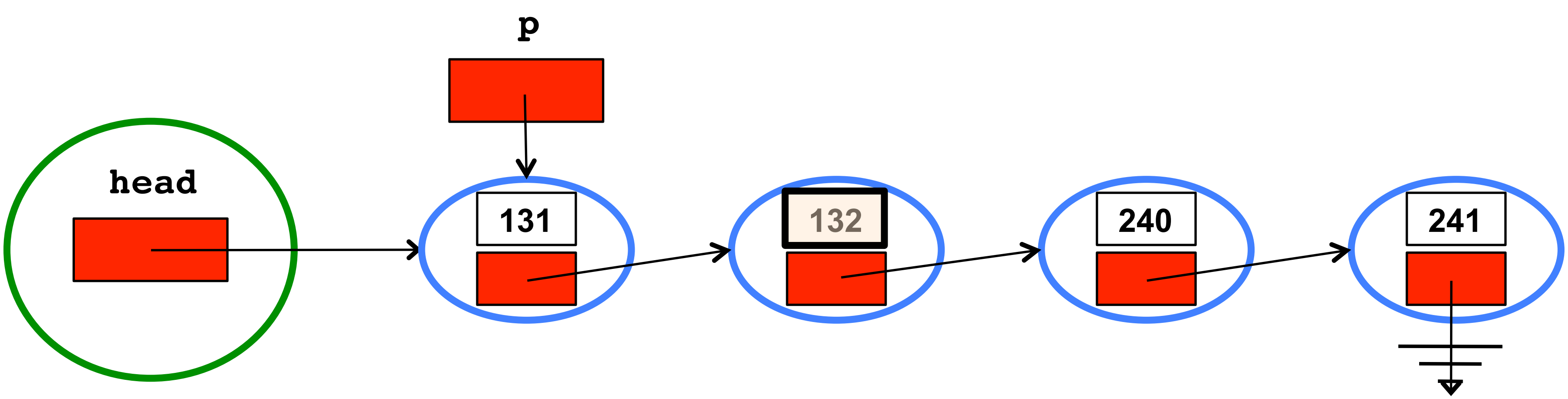
9.8 A `LinkedList` implementation of `List`

- Recall our List ADT:
- And recall our singly-linked list implementation
- The method `remove(int)`
  - Technique: look at `value` one to the right of `p`



9.8 A `LinkedList` implementation of `List`

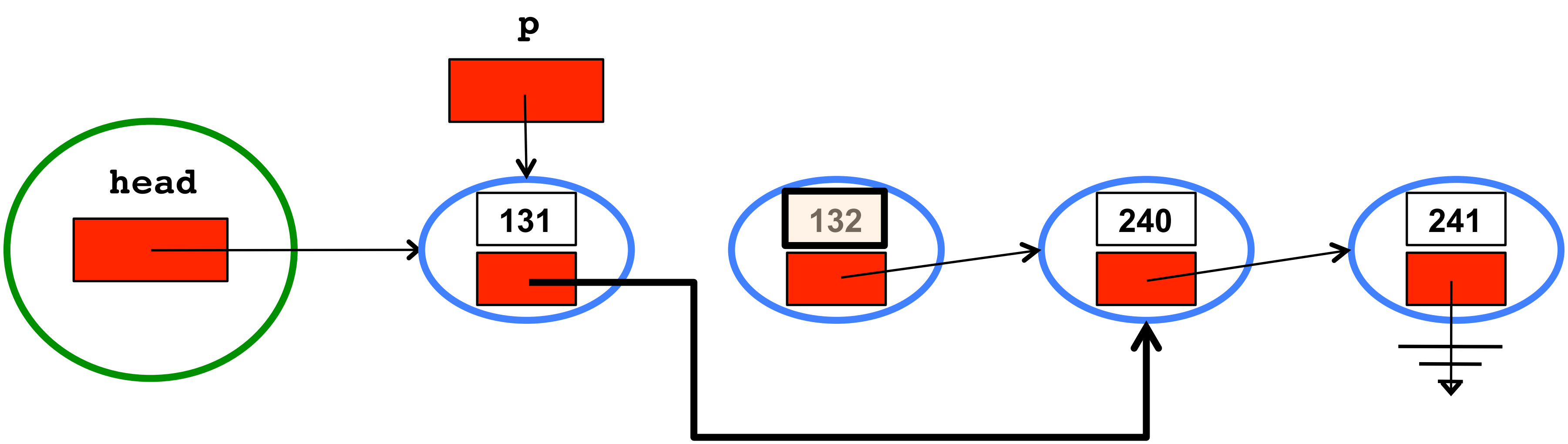
- Method `remove(Integer i)`
  - Trick: look at the node just to the right of `p`
  - When found, set `p.next = p.next.next`




```
list.remove(132);
```

9.8 A `LinkedList` implementation of `List`

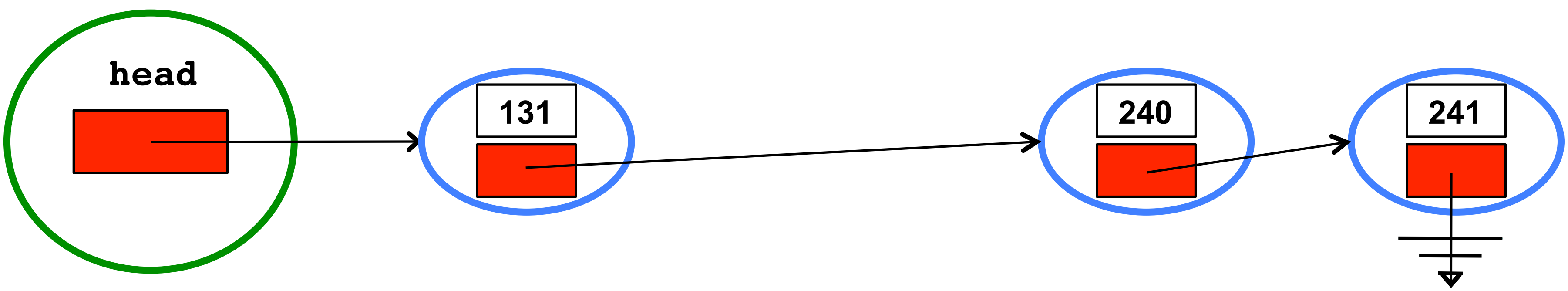
- Method `remove(Integer i)`
  - Trick: look at the node just to the right of `p`
  - When found, set `p.next = p.next.next`




```
list.remove(132);
```

9.8 A `LinkedList` implementation of `List`

- Method `remove(Integer i)`
  - Result is shown below




```
list.remove(132);
```


End of Oyster

Roundtable 9a

Roundtable

- Develop the SinglyLinkedList implementation
  - `addFirst`, `addLast`, `remove`
  - If time allows: `get`, `size`, and `isEmpty`

End of Roundtable

Roundtable 9b

Roundtable

- How is our code changed by adding a sentinel to the list?
  - A dummy node at the start of the list
  - It's there even just after the list is constructed
  - It never goes away
- Modify the methods to take account of this
  - And see how things get simpler
  - Each method – turn to student: how would you do this?
  - `addFirst`, `addLast`, `remove`
  - If time allows: `get`, `size`, and `isEmpty`

End of Roundtable