Linked Implementation of List ADT

- We will now cover link-based implementations of the List ADT. We will discuss two of them: **`LinkedUnsortedList`** and **`LinkedSortedList`**.
- In addition to what we've learned about a Linked List so far, we now need to be able to add or remove elements from anywhere on the linked list, including in the middle of a linked list; this is more complicated.
- We will also discuss indexed methods, such as return the i-th element. The cost of this is `O(N)` which is less efficient than an array-based implementation.

Linked Implementation of List ADT

- Similar to the array-based implementation, the link-based implementation implements the `ListInterface`, both the sorted version and unsorted.
- Recall the assumptions for lists we gave in the last lecture:
  - duplicate elements are allowed.
  - no `null` elements.
  - methods return values to indicate success / failure (as opposed to throw exceptions).
  - sorted lists are in non-decreasing order

LinkedUnsortedList

```
public class LinkedUnsortedList<T>
             implements ListInterface<T>
{
    protected int numElements;
    protected LLNode<T> list; // head of list

    public LinkedUnsortedList( ) {
        numElements= 0;
        list = null;
    }
```

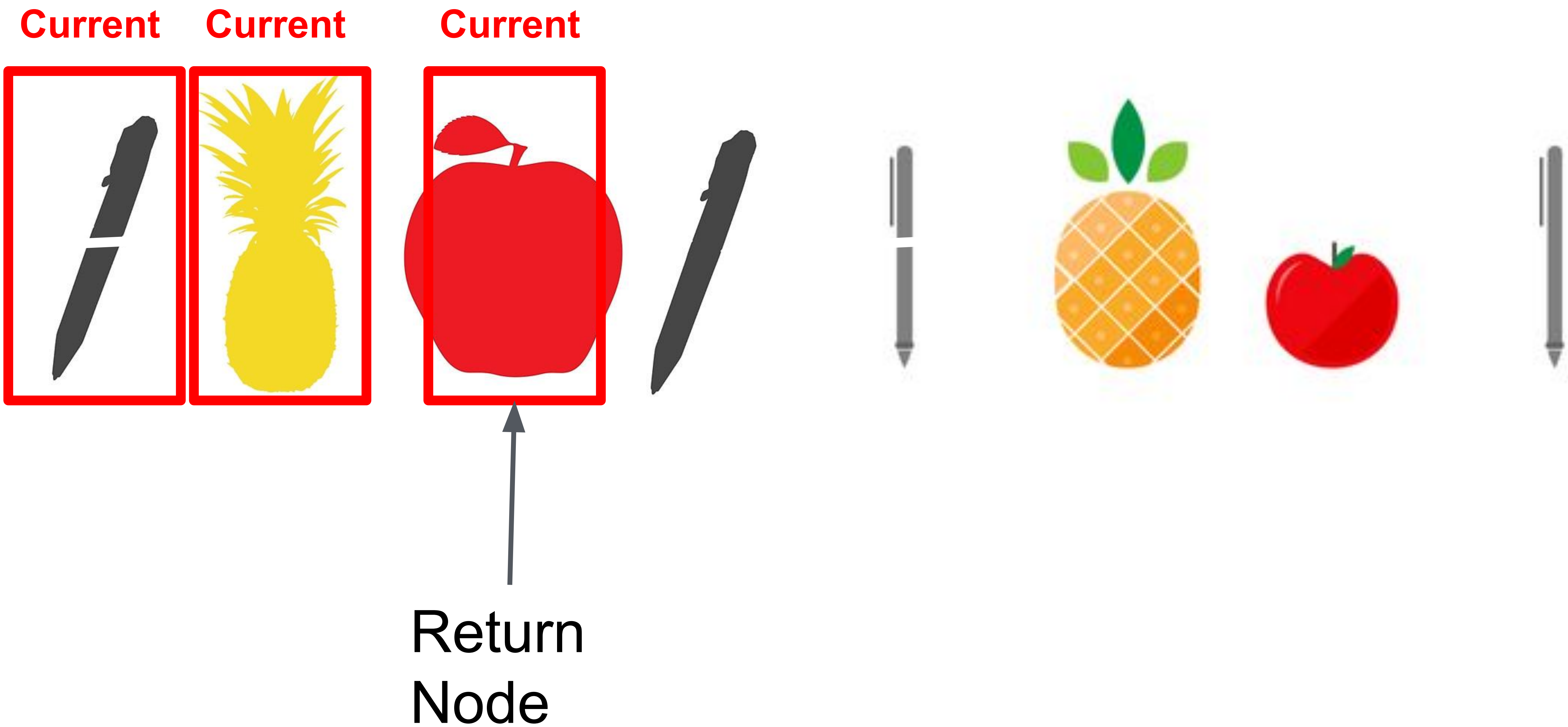
The `find` Method

- The `find` method traverses the linked structure to search for the element being requested. So how do we indicate the result of the search?

- If an element is found, we need a pointer to its node. So the `find` method should return a `LLNode<T>` object.

- If the element is not found, we return a `null`. Therefore a non-`null` return value means element is found, and a `null` return value means the element is not found.

The `find` Method



Version 1 of `find`

```
protected LLNode<T> find (T target) {
  LLNode<T> curr = list; // start from head
  while (curr != null) {
    if (curr.getInfo().equals(target))
      break;
    curr = curr.getLink();
  }
  return curr; // if not found, curr would be
               // null at this point
}
```

Version 2 of `find`

We can make it more concise:

```java
protected LLNode<T> find (T target) {
  LLNode<T> curr = list;
  while (curr != null &&
         !curr.getInfo().equals(target)) {
    curr = curr.getLink();
  }
  return curr;
}
```

Clicker Question #1

```
while (!curr.getInfo().equals(target) &&
        curr!=null) {
    curr = curr.getLink();
}
```

In the above code, we have switched the order of the two conditions around the && statement. What would happen? (**Hint**: think about how `cond1&&cond2` is evaluated)

(a) nothing would change

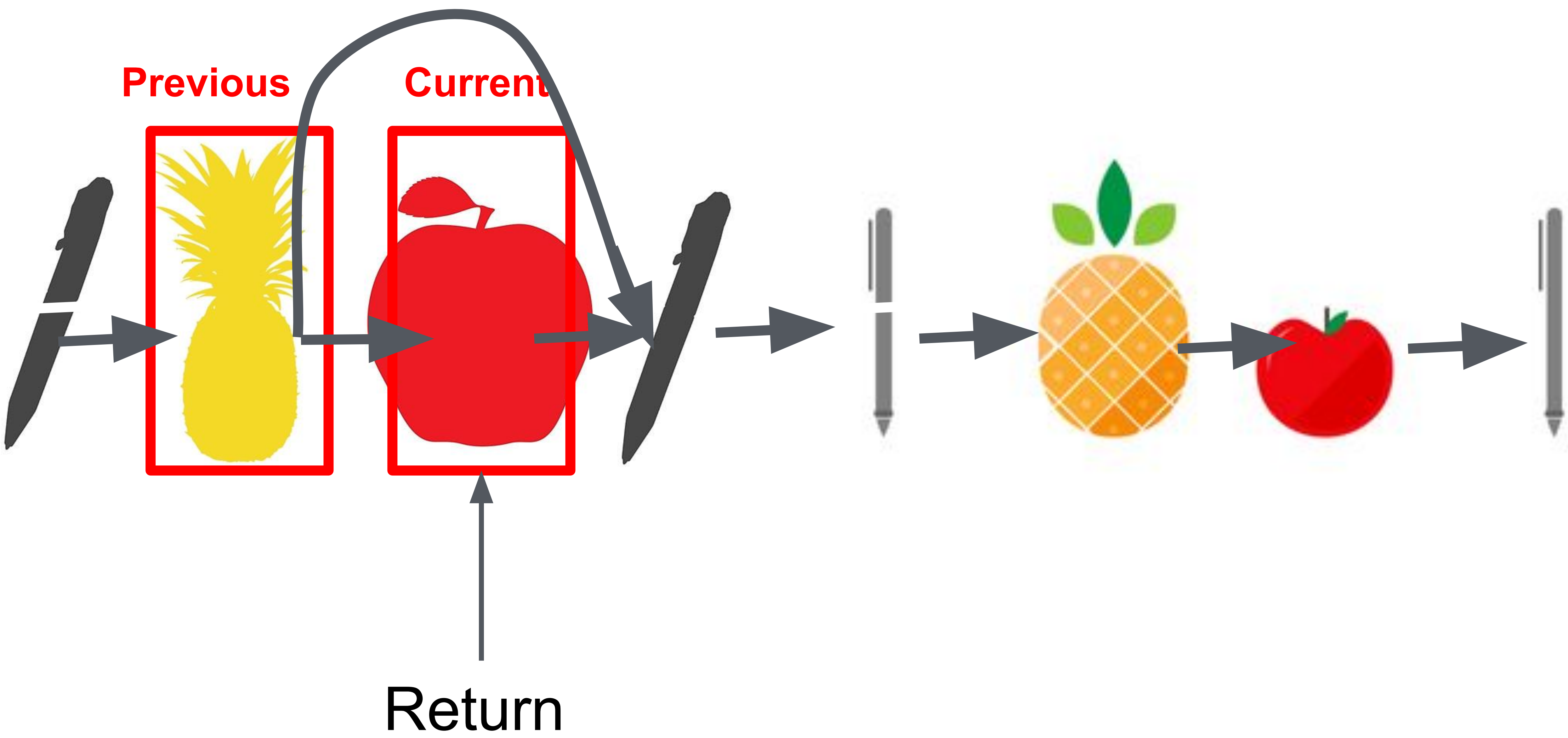
(b) `curr` will always be `null` after the loop ends.

(c) in some cases it throws `NullPointerException`

(d) in all cases it throws `NullPointerException`

(e) it will return next node of the target node

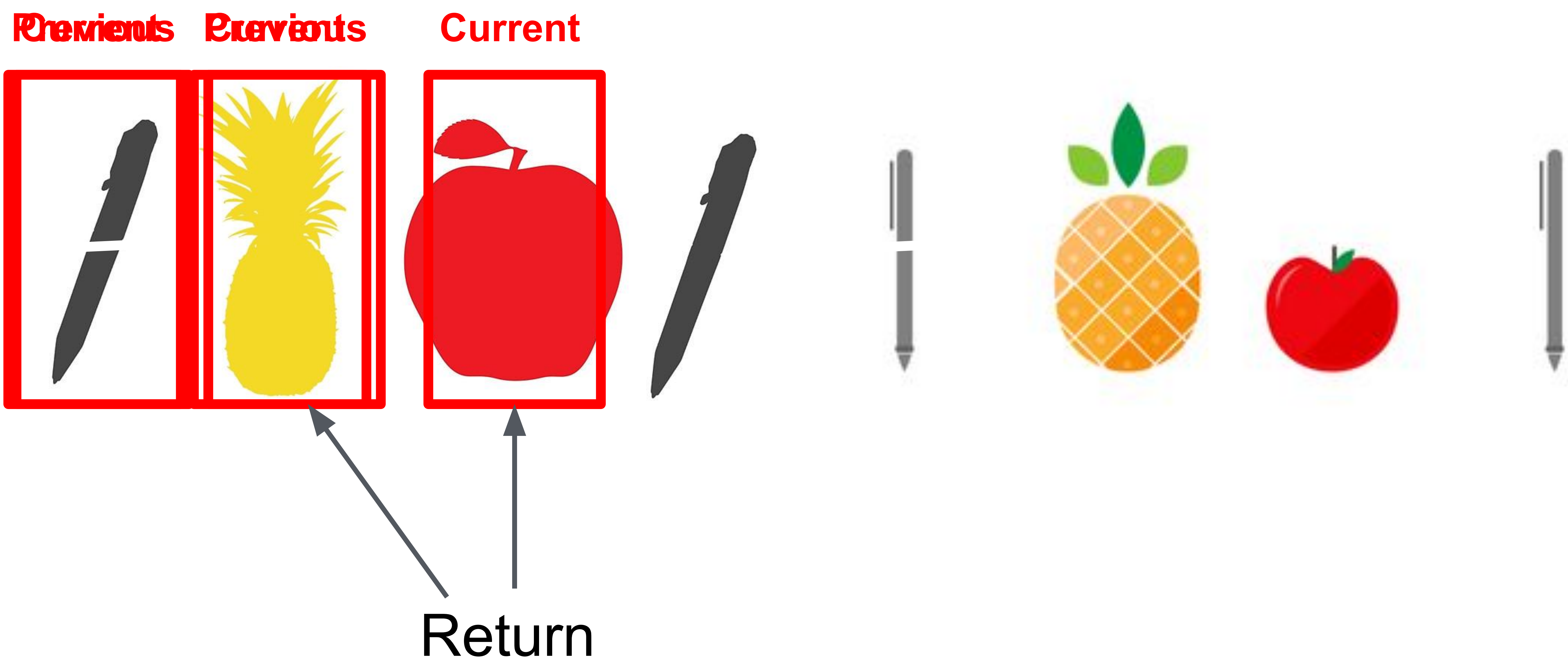
The `remove` Method



Revisiting the `find` Method

- Shortly we will see remove any node (not just first and last elements) from the list.
- To be able to remove a node somewhere in the middle of the list (say `curr`), we need to re-wire the link by setting `curr`'s predecessor to point to `curr`'s successor.
- To make use of the `find` method, it needs to return not only the found node, but also its predecessor.
- How can we have a Java method return two or more values??

The `find` Method v2



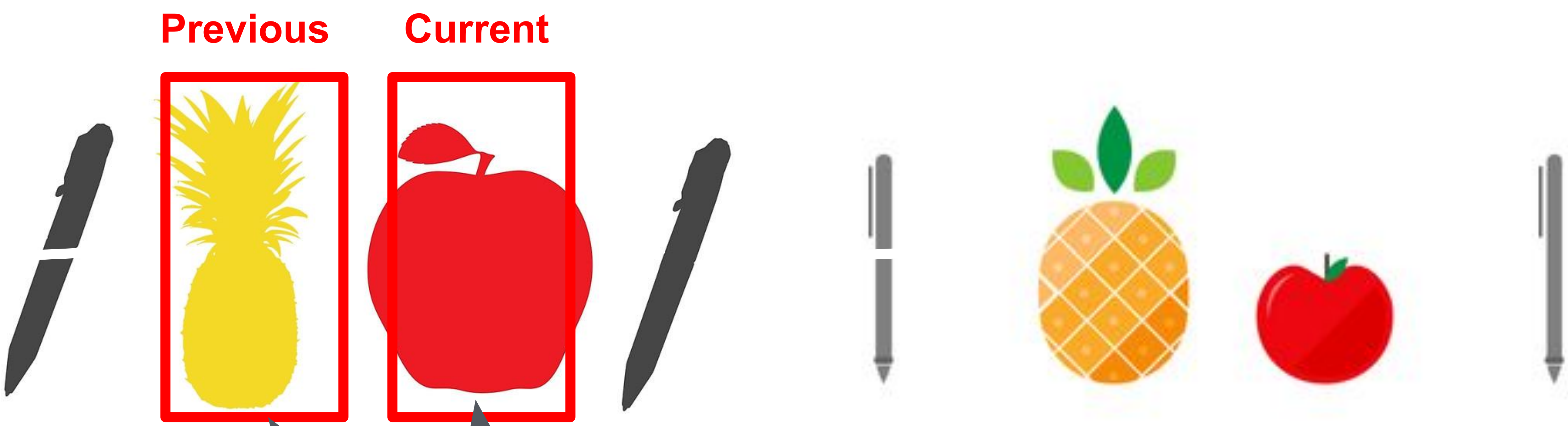
Revisiting the `find` Method

- One approach is to create a class that contains all the return values you need, and have the method return an object of that type.
  - You can also have the caller pass such an object in through argument, and have the callee modify the content of this object.

- Alternatively, if the return values are all of the same type, you can return an array that contains all return values:

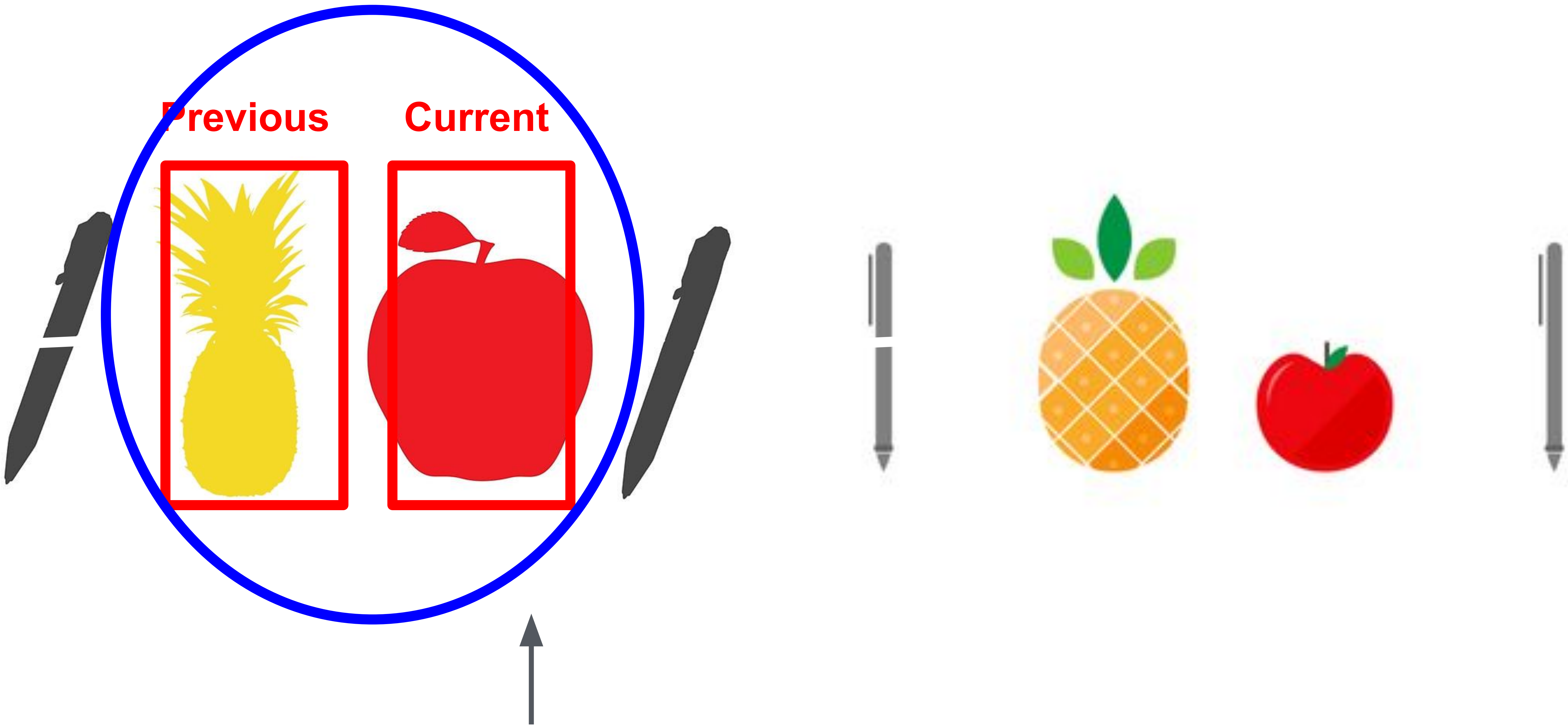
```
public int[] foo() {
   .. .. ..
   return new int[] {val1, val2, val3};
}
```

The `find` Method v2



Return array [, , , ...]

The `find` Method v2



Return Object `findVal`

```
private class findVal<T> {
  public LLNode<T> curr, prev;
  public findVal(LLNode<T> c, LLNode<T> p) {
    curr = c; prev = p;
  }
}
protected findVal<T> find (T target) {
  LLNode<T> curr = list, prev = null;
  while (curr != null) {
    if (curr.getInfo().equals(target))  break;
    else {
      prev = curr;
      curr = curr.getLink();
    }
  }
  return new findVal<T>(curr, prev);
}
```

Clicker Question #2

```
protected findVal<T> find (T target) {
  LLNode<T> curr = list, prev = null;
  while (curr != null) {
    if (curr.getInfo().equals(target))  break;
    else {
      prev = curr;
      curr = curr.getLink();
  }
  return new findVal<T>(curr, prev);
}
```

Suppose the list does <u>not</u> contain the element we are searching for. What's the value of **prev** returned?

(a) the last node (b) the first node

(c) null (d) same value as curr

The remove Method

```
public boolean remove (T element) {
  findVal<T> fval = find(element);
  if (fval.curr != null) {
    if (fval.prev == null) // node to remove is head
      list = list.getLink();
    else
      fval.prev.setLink(fval.curr.getLink());
    numElements--;
  }
  return (fval.curr != null); // true if we found it
}
```

Clicker Question #3

```
LLNode<T> next = curr.getLink(); // assume non-null
curr.setInfo(next.getInfo());
curr.setLink(next.getLink());
```

Assume `curr` points to an element somewhere in the middle of a linked list (i.e. neither the first nor the last). What's the **equivalent effect** of the above code?

(a) it removes the current element from the list.

(b) it removes the current element's successor from the list.

(c) it removes the current element's predecessor from the list.

(d) it duplicates the current element in the list.

(e) it make the current element the last node on the list.

The remove Method

```
public boolean remove (T element) {
  LLNode<T> curr = find(element);
  if (curr != null) {
    LLNode<T> next = curr.getLink();
    if (next != null) {
      curr.setInfo(next.getInfo());
      curr.setLink(next.getLink());
    } else
      curr.setInfo(null);
    numElements--;
  }
  return (curr != null); // true if we found it
}
```

Clicker Question #4

```java
public boolean remove (T element) {
  LLNode<T> curr = find(element);
  if (curr != null) {
    LLNode<T> next = curr.getLink();
    if (next != null) {
      curr.setInfo(next.getInfo());
      curr.setLink(next.getLink());
    } else
      curr.setInfo(null);
    numElements--;
  }
  return (curr != null); // true if we found it
}
```

Q: What's wrong?

(a) It works just fine
(b) NullPointerException
(c) It's wrong if curr is the head node
(d) It's wrong if curr is the tail node
(e) It's wrong for any node

The `add` Method

- For the **unsorted** list, the add method is simple: you can choose to use either front insertion, or end insertion. End insertion is recommended if you want elements to be stored in the same order as inserted.
- For the **sorted** list, the add method must ensure the sorted order of elements.
- The other methods are shared between the two classes. So we can have our `LinkedSortedList` extend `LinkedUnsortedList`, and override the `add` method.

LinkedSortedList

```
public class LinkedSortedList<T extends Comparable<T>>
     extends LinkedUnsortedList<T> {
    public LinkedSortedList() {
        super();
    }
```

Note that the generic type `T` here has a conditioner. It specifies that the type `T` must implement the `Comparable<T>` interface, so `T` must have a `compareTo` method.

Without this, if you try to call the `compareTo` method on a type `T` object, the compiler will complain.

ArraySortedList

```
public class ArraySortedList<T> extends ArrayUnsortedList<T> {
  public void add (T elem) { // must preserve sorted order
    int location = 0;
    if (numElements == list.length)  enlarge();
    while (location < numElements) {
      if (((Comparable<T>)list[location]).compareTo(elem) < 0)
        location++;
      else
        break;
    }
    for (int index=numElements; index>location; index--)
      list[index] = list[index-1];
    list[location] = element;
    numElements++;
  }
```

This cast is necessary. Without it, the compiler will complain that it doesn't know T is a class that implements Comparable<T>

```java
public class LinkedSortedList<T extends Comparable<T>>
     extends LinkedUnsortedList<T> {
    public LinkedSortedList() {
        super();
    }
```

You can certainly explicitly cast a type `T` object to (`Comparable<T>`) type whenever you call the `compareTo` method (like we did in the array case). But specifying it in the header as shown above is a lot easier.

Note that we use `T` **`extends`** `Comparable<T>` rather than **`implements`**, because `T` could be anywhere below `Comparable<T>` in the inheritance hierarchy. Not just directly below it.

Method of `LinkedSortedList`

- To add to a sorted list, we skip **past all elements that are less than the new element**, until either we reach the end or we find the right element to insert the new element in front of.
- We can't make use of the `find` method because we are not searching for a specific element, instead, we are looking for the first element that's no longer smaller than the new element to be added.
- So we will basically replicate the `find` method, with the necessary changes.

```java
public void add (T element) {
    LLNode<T> curr = list, prev = null;
    while (curr != null) {
        if (curr.getInfo().compareTo(element) < 0) {
            prev = curr;
            curr = curr.getLink();
        } else break;
    }




}
```

Find the right spot

Insert new node

```java
public void add (T element) {
    LLNode<T> curr = list, prev = null;
    while (curr != null) {
        if (curr.getInfo().compareTo(element) < 0) {
            prev = curr;
            curr = curr.getLink();
        } else break;
    }
    LLNode<T> newNode = new LLNode<T>(element);
    if (prev == null) {  // adding before head
        newNode.setLink(list);
         list = newNode;
    } else {
        newNode.setLink(curr);
        prev.setLink(newNode);
    }
    numElements++;
}
```

Clicker Question #5

```
public void add (T element) {
    LLNode<T> curr = list, prev = null;
    while (curr != null) {
        if (curr.getInfo().compareTo(element) < 0) {
            prev = curr;
            curr = curr.getLink();
        } else break;
    }
```

If the list contains elements `A,B,C,E,E` to begin with, and you are adding a new element `E`, what nodes will `curr` and `prev` point to after the above loop?

(a) `curr: A, prev: null`

(b) `curr: E, prev: C`

(c) `curr: E, prev: E`

(d) `curr: null, prev: E`

(e) `curr: E, prev: null`

The Indexed Methods

- The `ListInterface` also contains several indexed methods, such as

```
boolean insert (int index, T element);
boolean set (int index, T element);
T get (int index);
int indexOf (T element);
T remove (int index);
```

- To assist the implementation of these methods, we can define a indexed version of `find` method that takes an `index` value, and returns pointers `curr` and `prev`.

```
protected findVal<T> find (int index) {
  LLNode<T> curr = list, prev = null;
  int i = 0;
  while (curr!=null && i!=index) {
    prev = curr;
    curr = curr.getLink();
    i++;
  }
  return new findVal<T>(curr, prev);
}
```

With this `find` method, we can easily implement those indexed `insert`, `remove`, `get` methods. These will be left as exercises for you to do on your own.

Questions