Implementing
Linked Lists (pt. 2)

Based on slides by Prof. Burton Ma

Iterable Interface

```
public interface Iterable<T>
```

Implementing this interface allows an object to be the target of the "foreach" statement.

`Iterator<T>`	`iterator()` Returns an iterator over a set of elements of type **`T`**.

Iterator

- To implement **`Iterable`** we need to provide an iterator object that can iterate over the elements in the list

`boolean`	**`hasNext()`** Returns true if the iteration has more elements.
`E`	**`next()`** Returns the next element in the iteration.
`void`	**`remove()`** Removes from the underlying collection the last element returned by this iterator (optional operation).

Implementing Iterable

- Having our linked list implement **`Iterable`** would be very convenient for clients

```
// for some LinkedList t

for (Character c : t) {
  // do something with c
}
```

Iterable Interface

```
public interface Iterable<T>
```

Implementing this interface allows an object to be the target of the "foreach" statement.

`Iterator<T>`	`iterator()` Returns an iterator over a set of elements of type **`T`**.

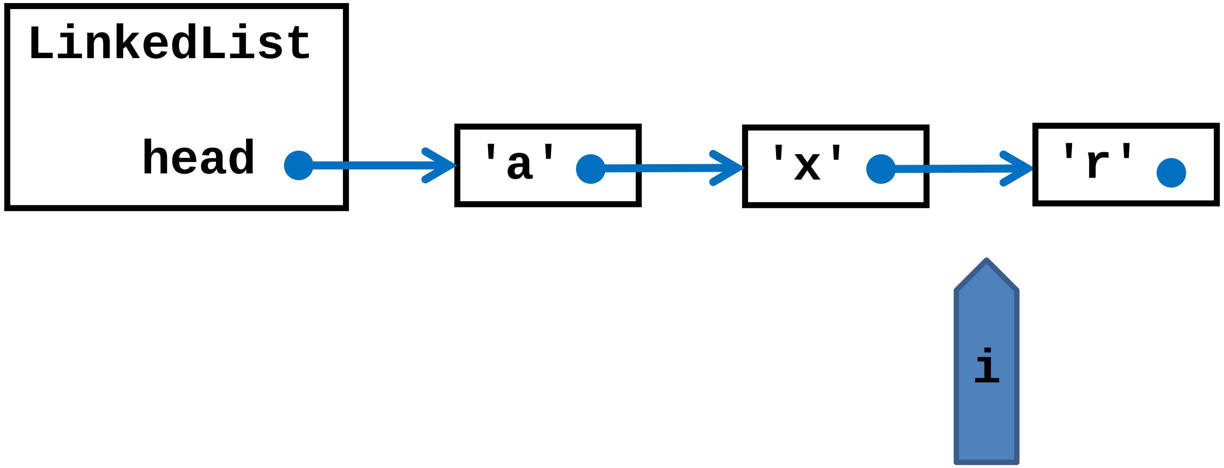
Iterator

- To implement **`Iterable`** we need to provide an iterator object that can iterate over the elements in the list

`boolean`	**`hasNext()`** Returns true if the iteration has more elements.
E	**`next()`** Returns the next element in the iteration.
`void`	**`remove()`** Removes from the underlying collection the last element returned by this iterator (optional operation).

LinkedList Iterator

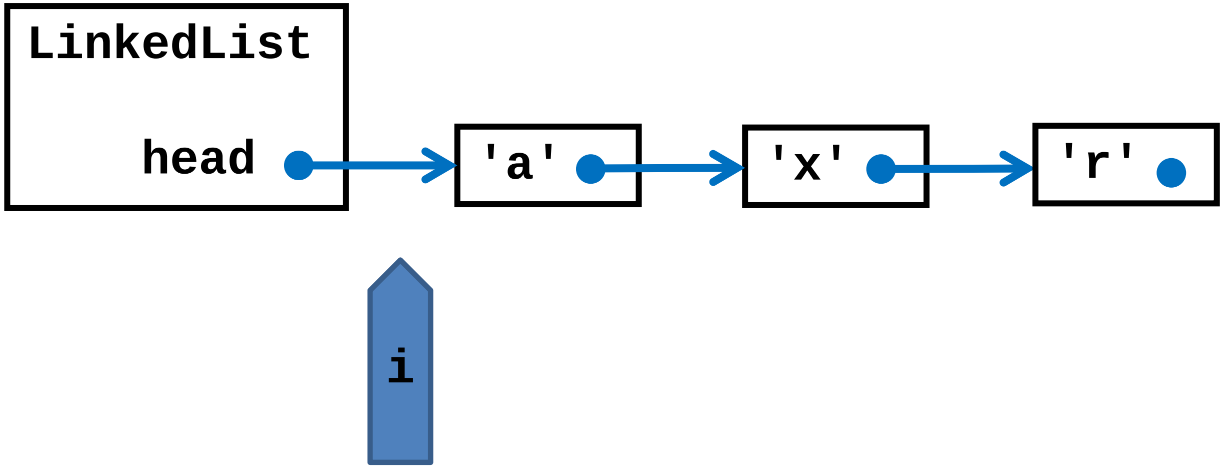
- Think of the iterator as lying between elements in the list (like a cursor)



iterator between 'x' and 'r'

LinkedList Iterator

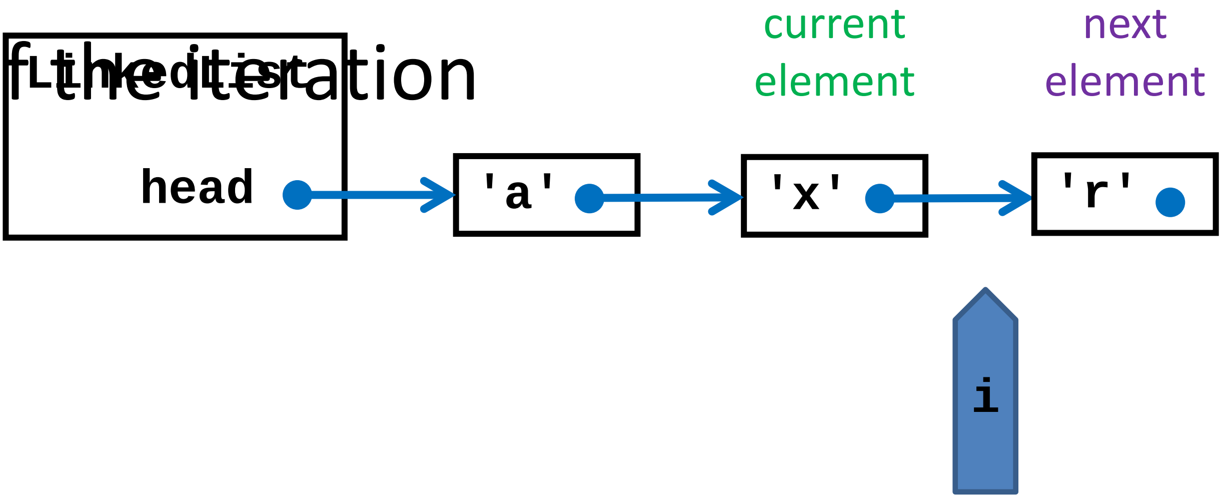
- Think of the iterator as lying between elements in the list (like a cursor)



iterator at the start of the iteration
(between nothing and 'a')

LinkedList Iterator

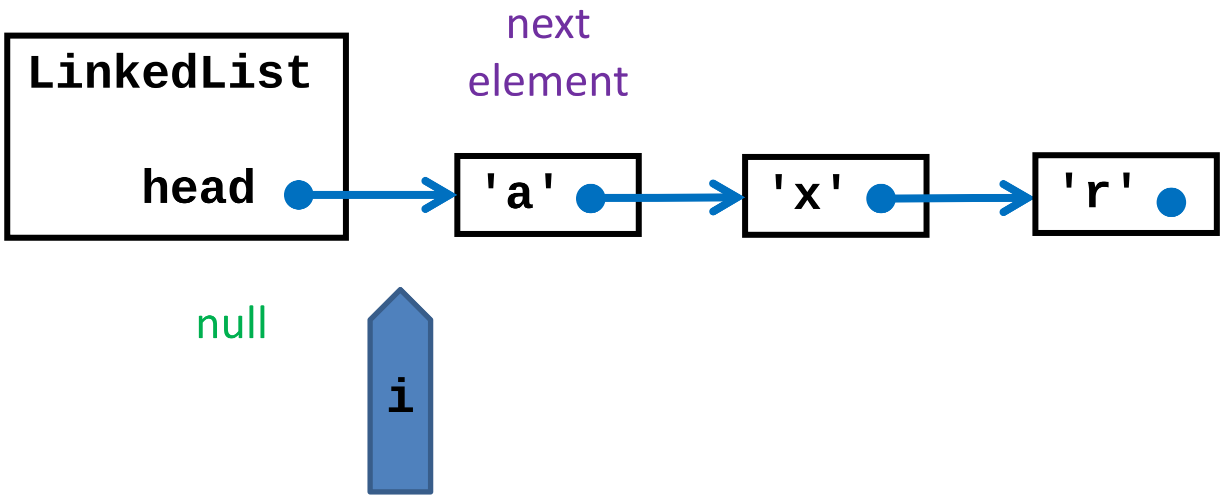
- Because the iterator is between elements, there is a current element and next element of the iteration



iterator between 'x' and 'r'

LinkedList Iterator

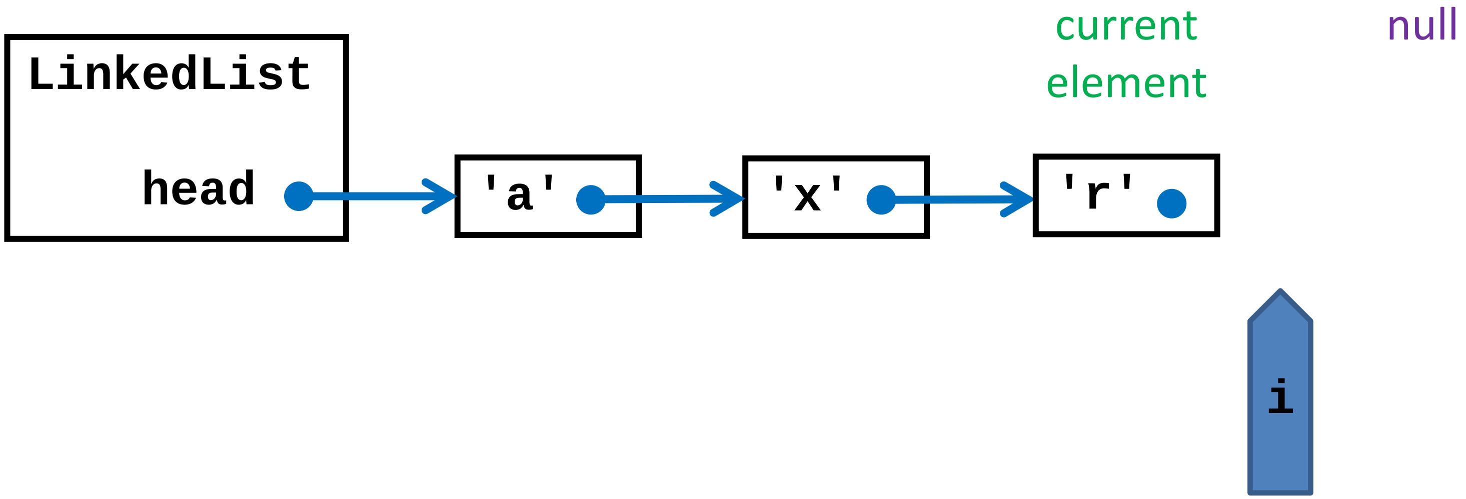
- The current element is **`null`** at the start of the iteration



iterator at the start of the iteration
(between nothing and 'a')

LinkedList Iterator

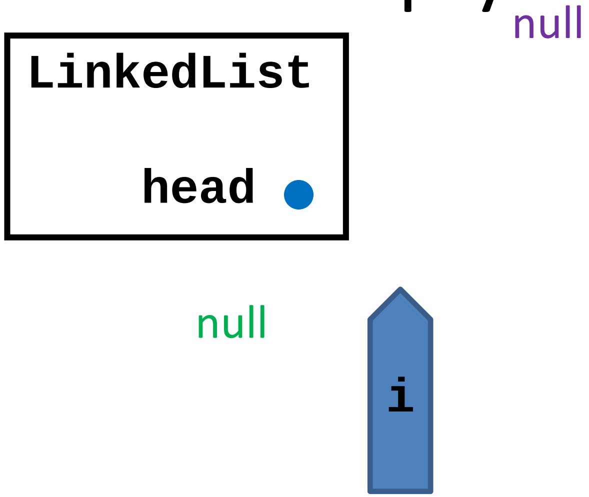
- The next element is **`null`** at the end of the iteration



iterator between 'x' and nothing

LinkedList Iterator

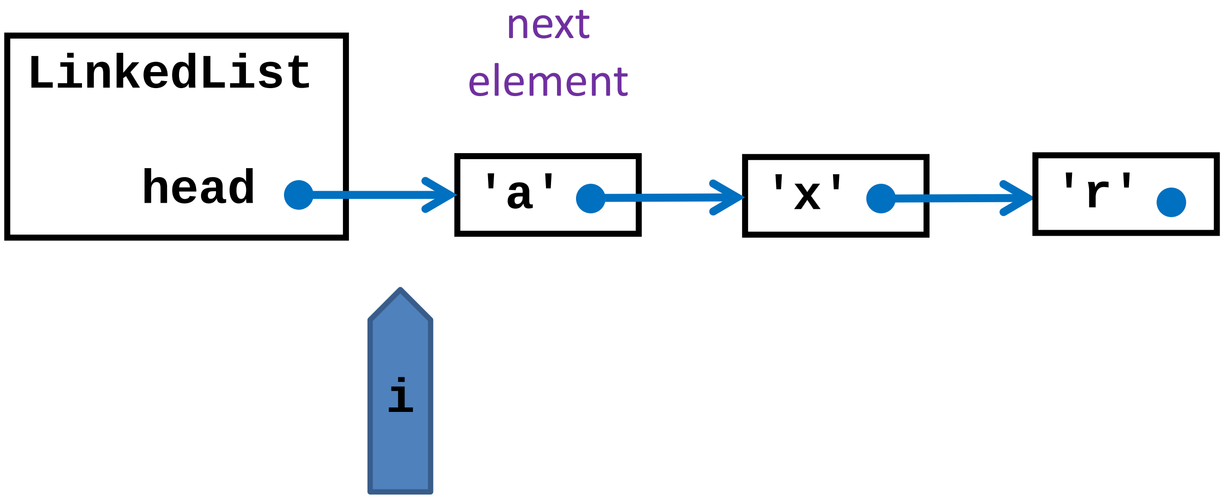
- Both the current and next elements are **`null`** if the list is empty



iterator at the start of the iteration

LinkedList Iterator: hasNext

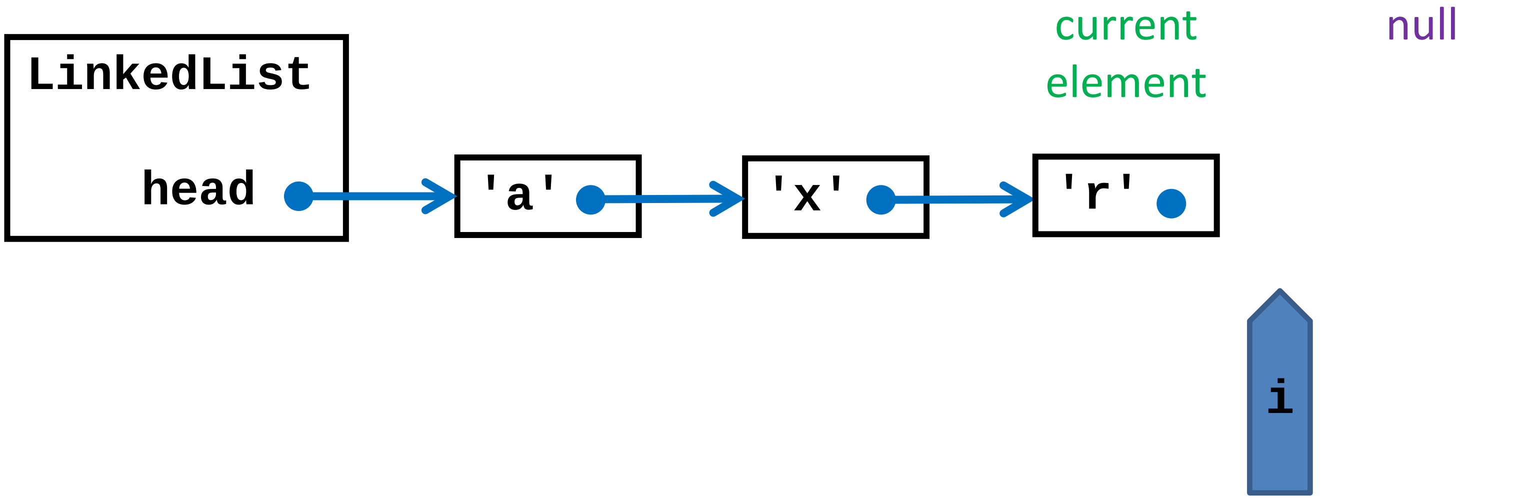
- **`hasNext()`** returns true if there is at least one more element in the iteration



`i.hasNext()` is **`true`**

LinkedList Iterator: hasNext

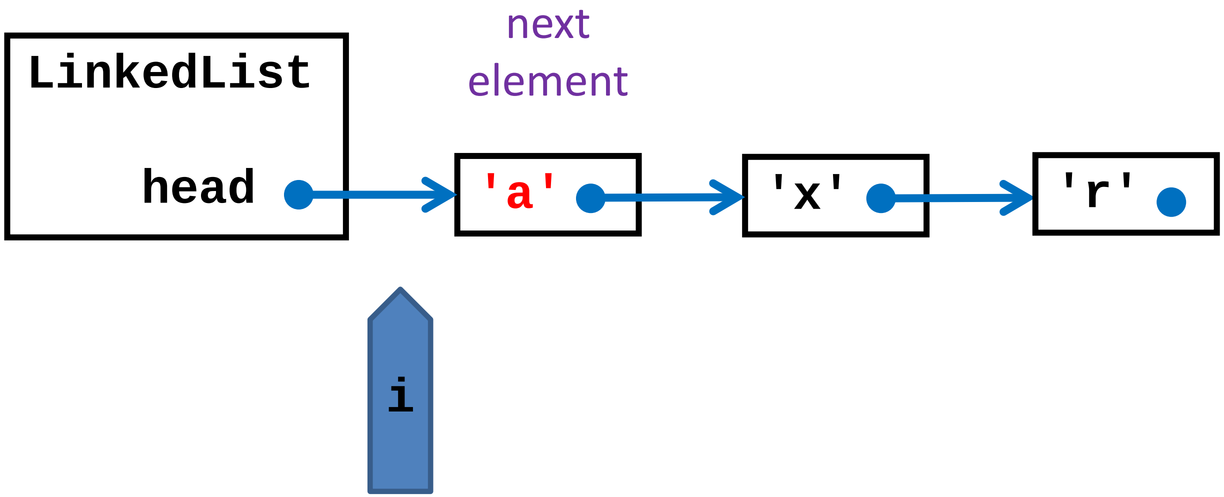
- **`hasNext()`** returns false at the end of the iteration



`i.hasNext()` is **`false`**

LinkedList Iterator: next

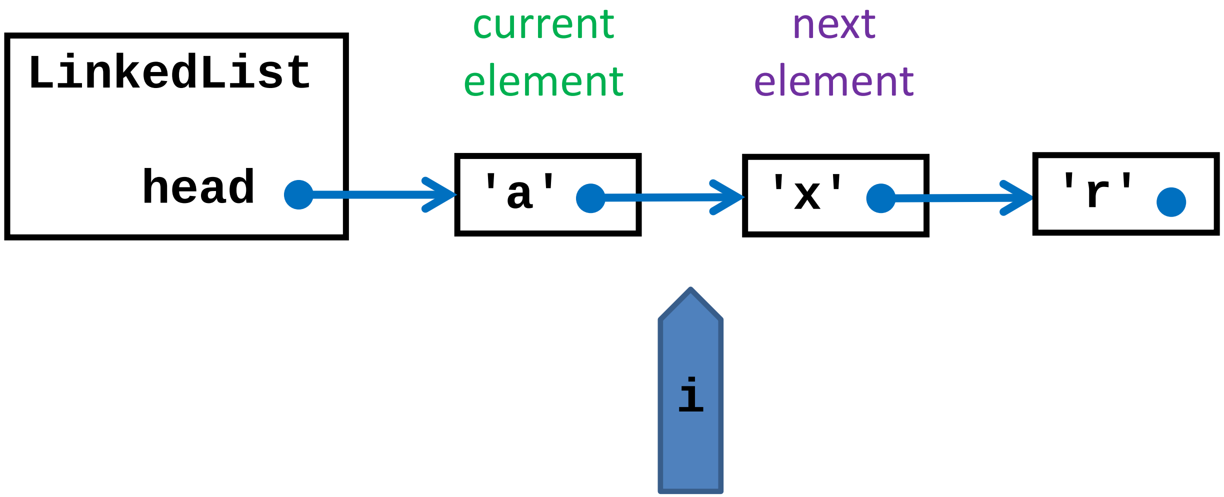
- Invoking **`next()`** returns the next element…



`i.next() == 'a'` is **`true`**

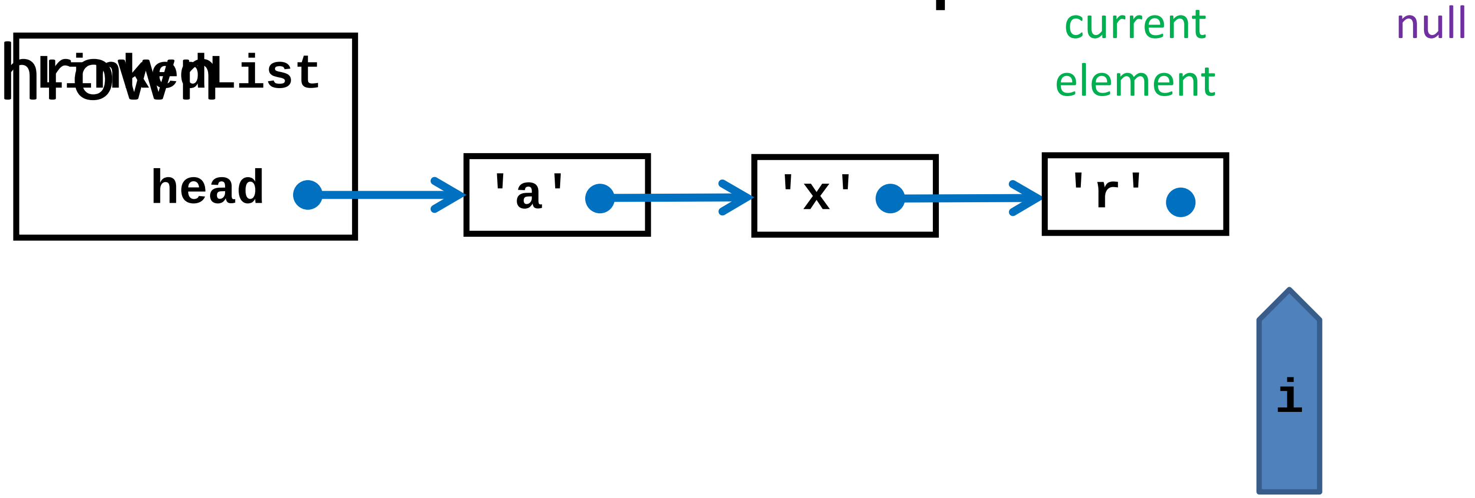
LinkedList Iterator: next

- …and causes the iterator to move to its next position in the iteration



LinkedList Iterator: next

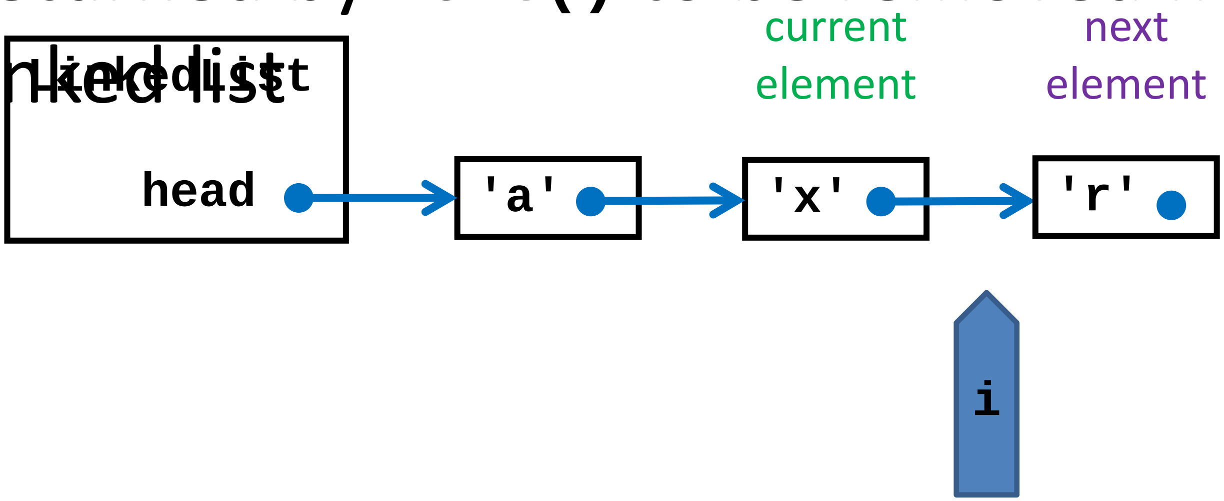
- Invoking **`next()`** at the end of the iteration causes a **`NoSuchElementException`** to be thrown



`i.next()` causes a **`NoSuchElementException`**

LinkedList Iterator: remove

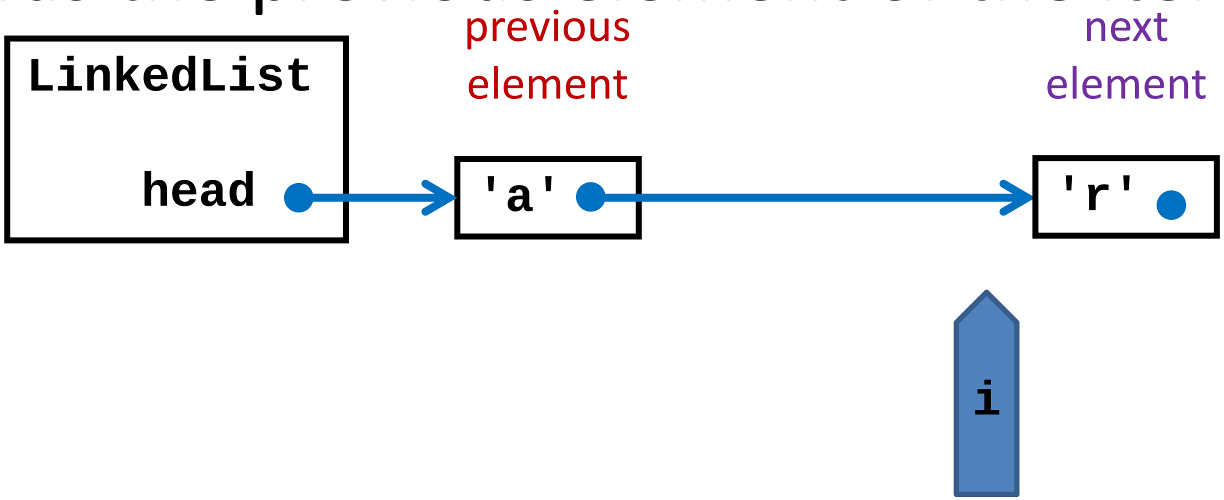
- **`remove()`** causes the element most recently returned by **`next()`** to be removed from the linked list



`i.remove()` causes 'x' to be removed

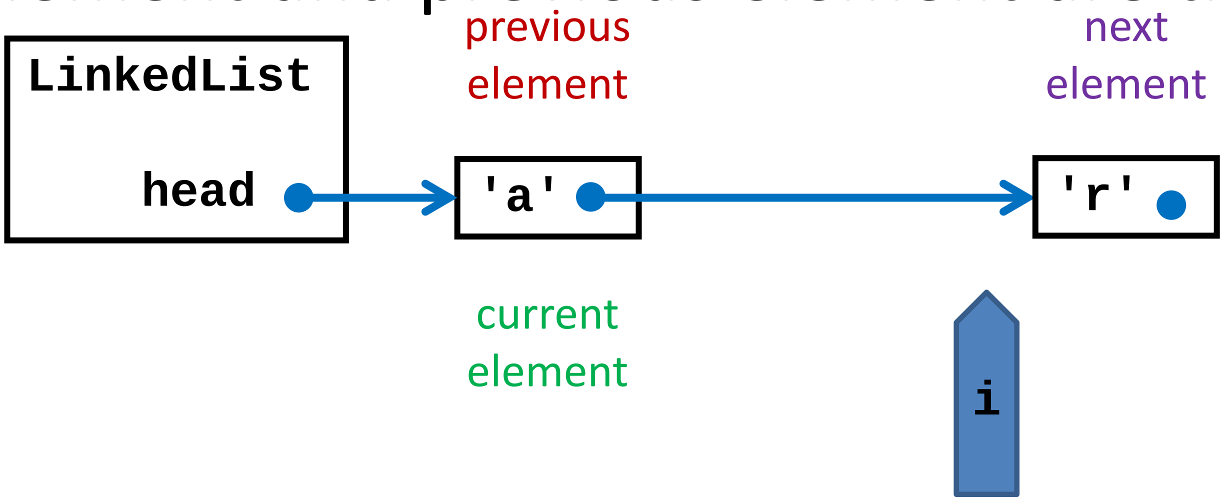
LinkedList Iterator: remove

- Notice that the iterator needs to know what was the previous element of the iteration



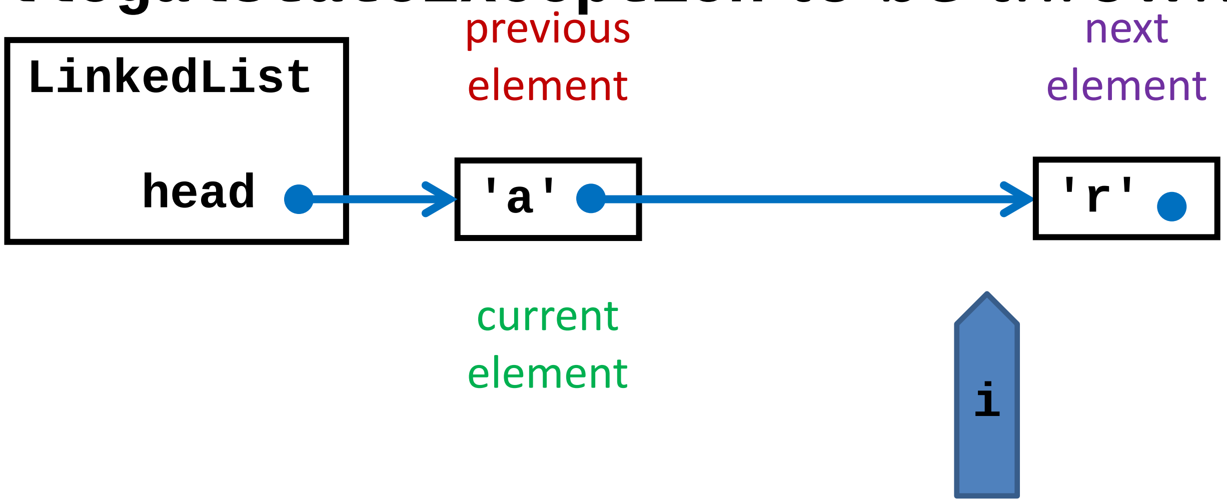
LinkedList Iterator: remove

- After removing the element the current element and previous element are the same



LinkedList Iterator: remove

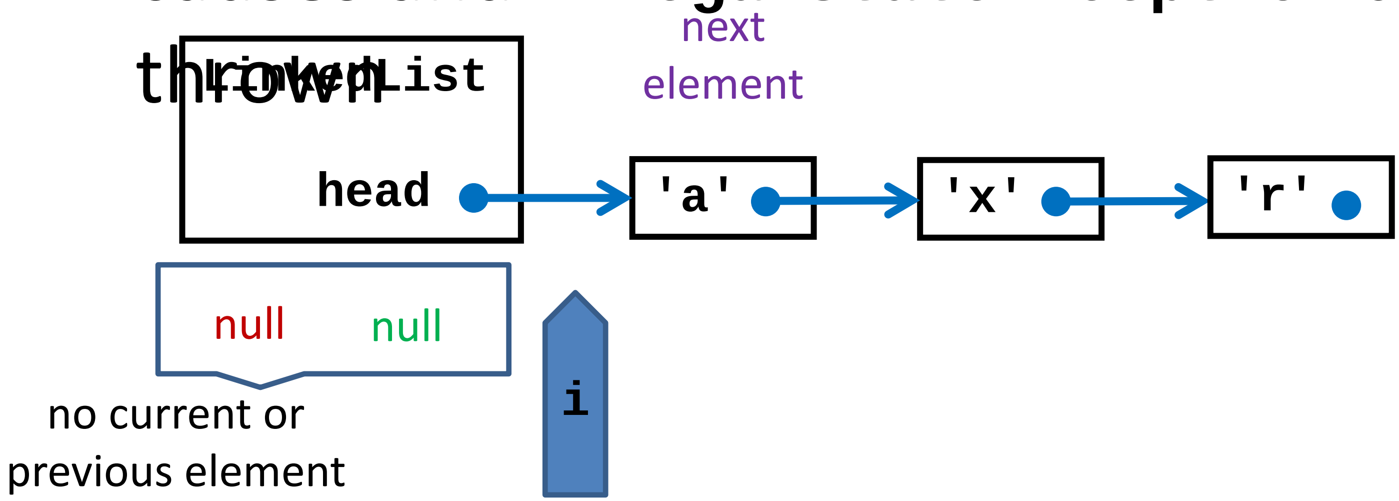
- Invoking **`remove()`** a second time causes an **`IllegalStateException`** to be thrown



`i.remove()` causes an **`IllegalStateException`**

LinkedList Iterator: remove

- Invoking **`remove()`** before calling **`next()`** also causes and **`IllegalStateException`** to be thrown



`i.remove()` causes an **`IllegalStateException`**

LinkedList Iterator: remove

- Note that using an iterator and **`remove()`** is the safest way to iterate over a collection and selectively remove elements from the collection
  - Called filtering

LinkedList Iterator: remove

```
// removes vowels from our LinkedList t

for (Iterator<Character> i = t.iterator();
     i.hasNext(); ) {
  char c = i.next();
  if (String.valueOf(c).matches("[aeiou]")) {
    System.out.println("removing " + c);
    i.remove();
  }
}
```

Implementation

- **`currNode`**
  - Reference to the node most recently returned by **`next()`**
    - This means that **`currNode`** is **`null`** at the start of the iteration
      - Requires special treatment in methods
- **`prevNode`**
  - Reference to the node previous to **`currNode`**
    - Needed for **`remove()`**

Implementation: Attributes and Ctor

```
private class LinkedListIterator implements
Iterator<Character> {

  private Node currNode;
  private Node prevNode;

  public LinkedListIterator() {
    this.currNode = null;
    this.prevNode = null;
  }
```

Implementation: hasNext

```
@Override
public boolean hasNext() {
  if (this.currNode == null) {
    return head != null;
  }
  return this.currNode.next != null;
}
```

Implementation: next

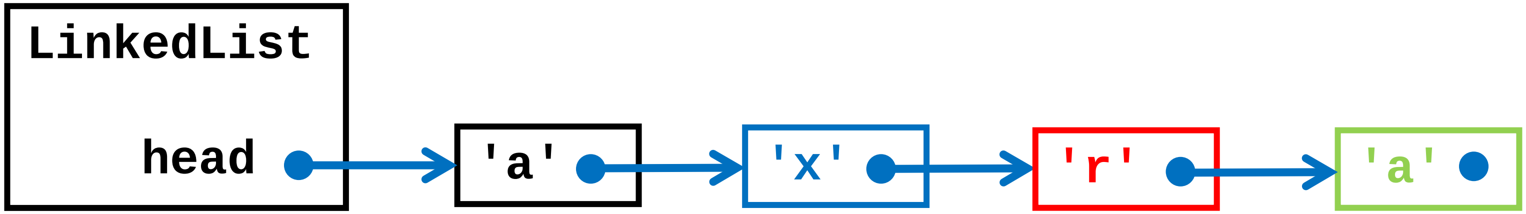
```
@Override
public Character next() {
  if (!this.hasNext()) {
    throw new NoSuchElementException();
  }
  this.prevNode = this.currNode;
  if (this.currNode == null) {
    this.currNode = head;
  }
  else {
    this.currNode = this.currNode.next;
  }
  return this.currNode.data;
}
```

Implementation: remove

```
@Override
public void remove() {
  if (this.prevNode == this.currNode) {
    throw new IllegalStateException();
  }
  if (this.currNode == head) {
    head = this.currNode.next;
  }
  else {
    this.prevNode.next = this.currNode.next;
  }
  this.currNode = this.prevNode;
  size--;
}
```

LinkedList Summary

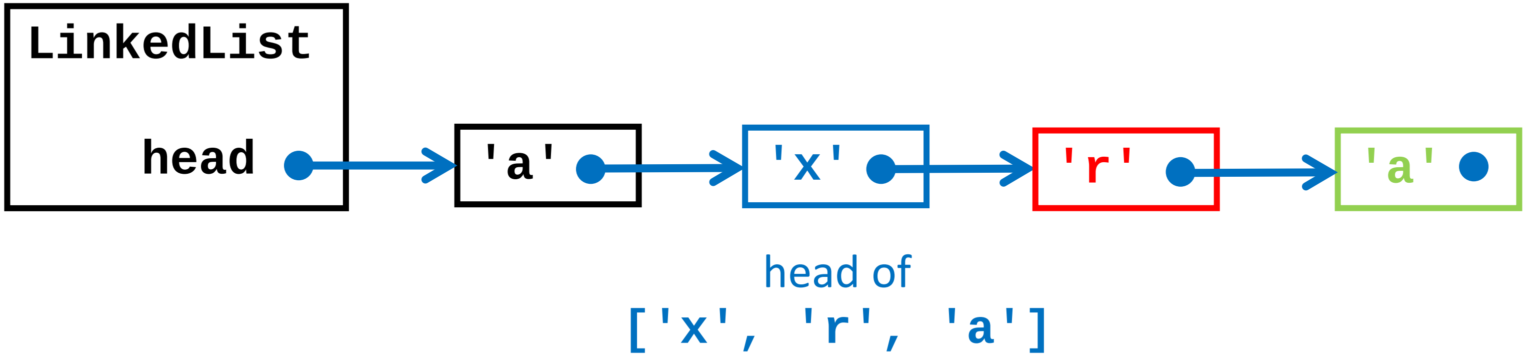
- Each node can be thought of as the head of a smaller list



head of
['a', 'x', 'r', 'a']

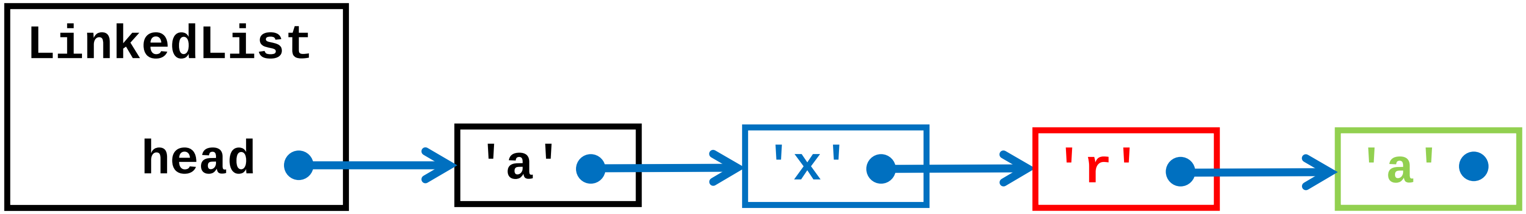
LinkedList Summary

- Each node can be thought of as the head of a smaller list



LinkedList Summary

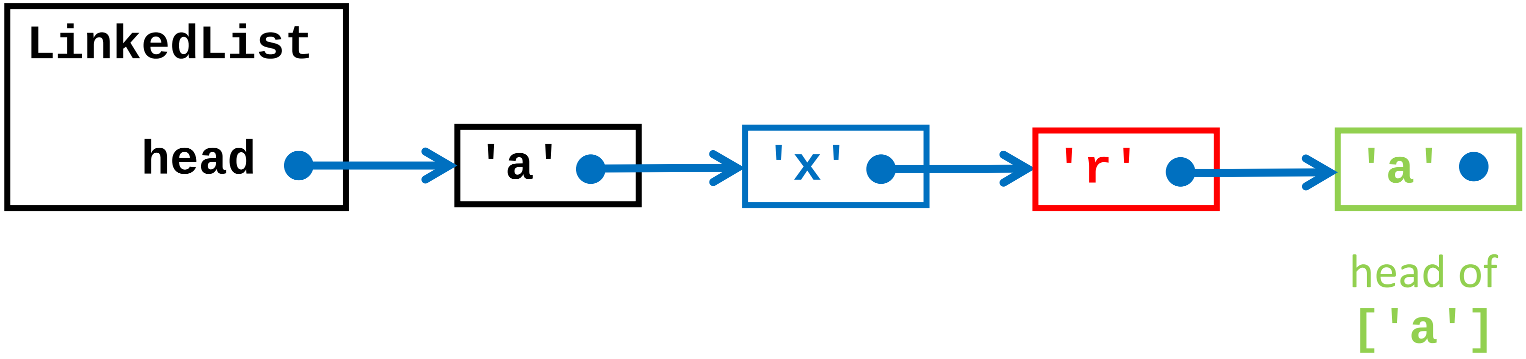
- Each node can be thought of as the head of a smaller list



head of
`['r', 'a']`

LinkedList Summary

- Each node can be thought of as the head of a smaller list



LinkedList Summary

- The recursive structure of the linked list leads to recursive algorithms that operate on the list

```
private static boolean contains(char c, Node node) {
  if (node.data == c) {
    return true;
  }
  if (node.next == null) {
    return false;
  }
  return LinkedList.contains(c, node.next);
}
```

LinkedList Summary

- Nodes are an implementation detail
  - The client only cares about the elements (characters) in the list

- **`Node`** is implemented as a private static inner class
  - private so that only **`LinkedList`** can use it
  - static because **`Node`** does not need access to any non-static attribute of **`LinkedList`**

LinkedList Summary

- By implementing the **`Iterable`** interface we give clients the ability to iterate over the elements of the list
- Clients expect to be able to do this for most collections

```
// for some LinkedList t

for (Character c : t) {
  // do something with c
}
```

LinkedList Summary

- To implement **`Iterable`** we need to provide an iterator object that can iterate over the elements in the list

`boolean`	**`hasNext()`** Returns true if the iteration has more elements.
`E`	**`next()`** Returns the next element in the iteration.
`void`	**`remove()`** Removes from the underlying collection the last element returned by this iterator (optional operation).