Recursive Objects

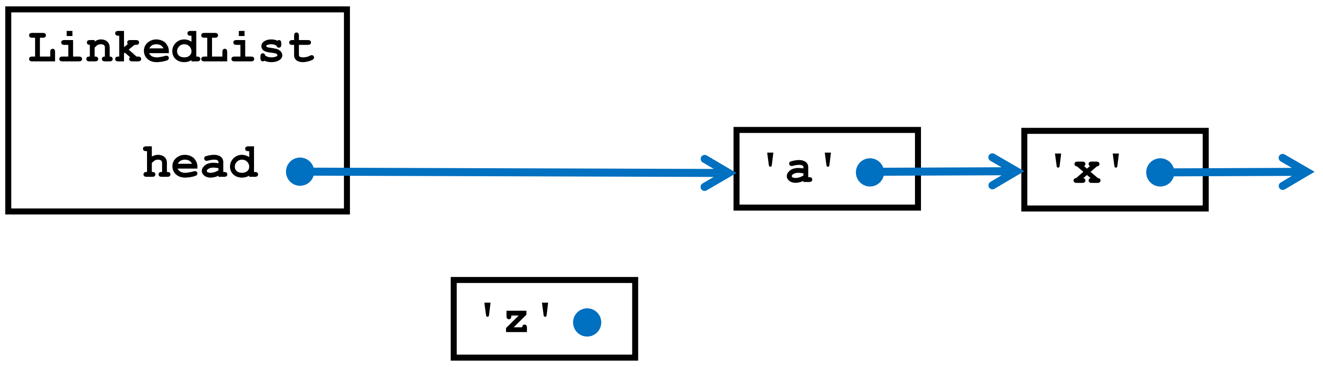
Singly Linked List (Part 2)

Operations at the head of the list

- operations at the head of the list require special handling because there is no node before the head node

Adding to the front of the list

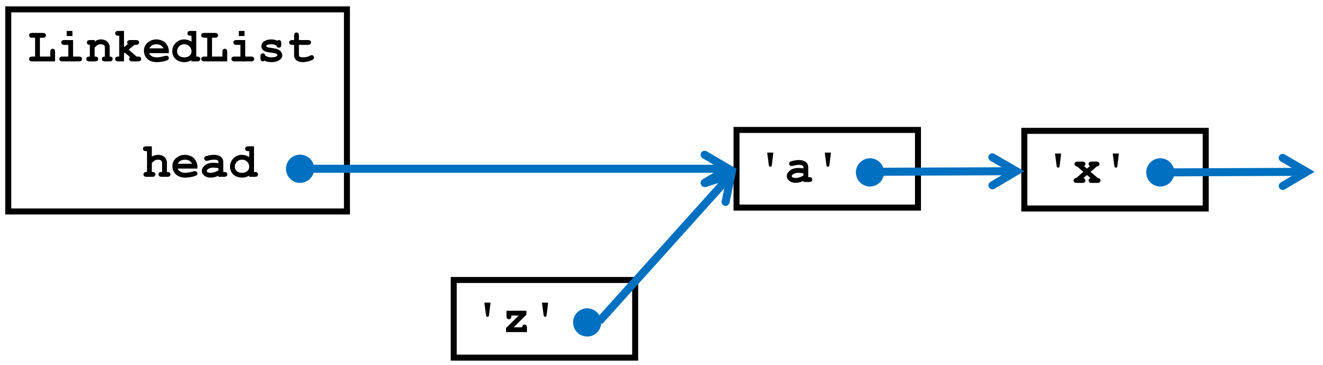
- adding to the front of the list



- `t.addFirst('z')` or `t.add(0, 'z')`

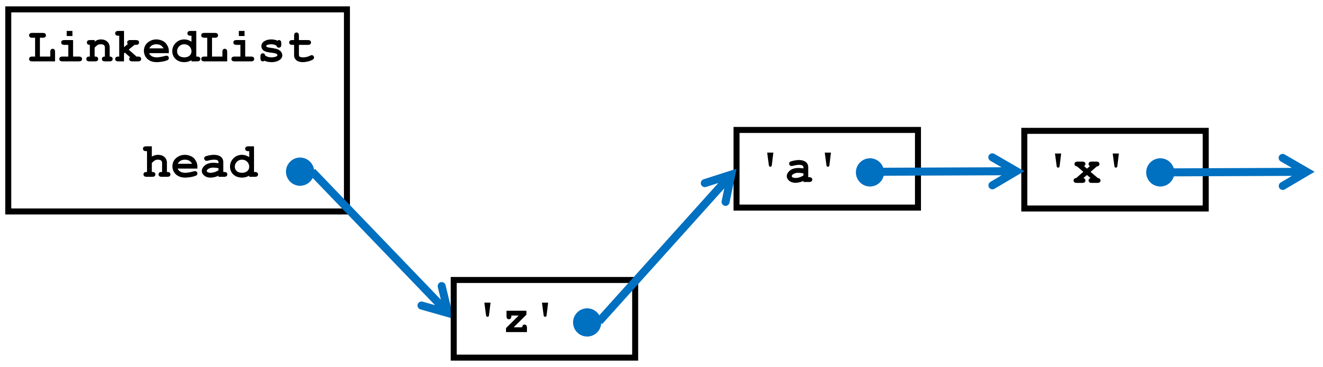
Adding to the front of the list

- must connect to the rest of the list



Adding to the front of the list

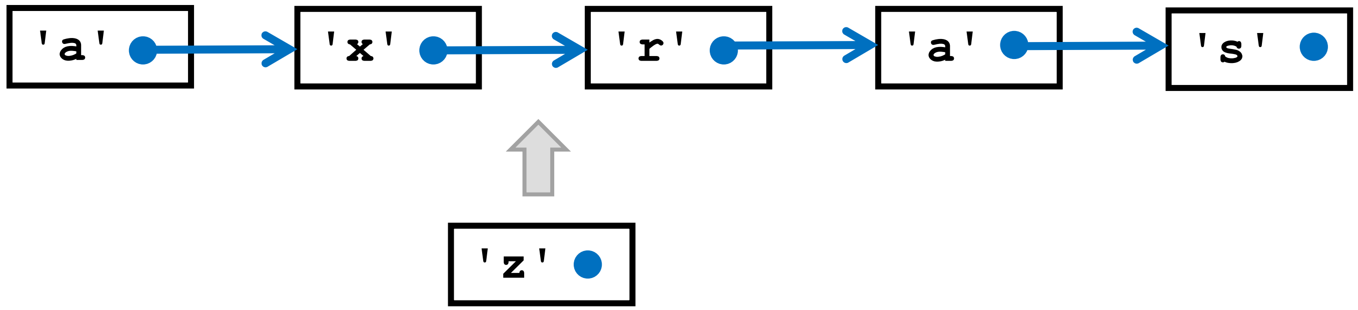
- then re-assign head of linked list



```
/**
 * Inserts the specified element at the beginning of this list.
 *
 * @param c the character to add to the beginning of this list.
 */
public void addFirst(char c) {
  Node newNode = new Node(c);
  newNode.next = this.head;
  this.head = newNode;
  this.size++;
}
```

Adding to the middle of the list

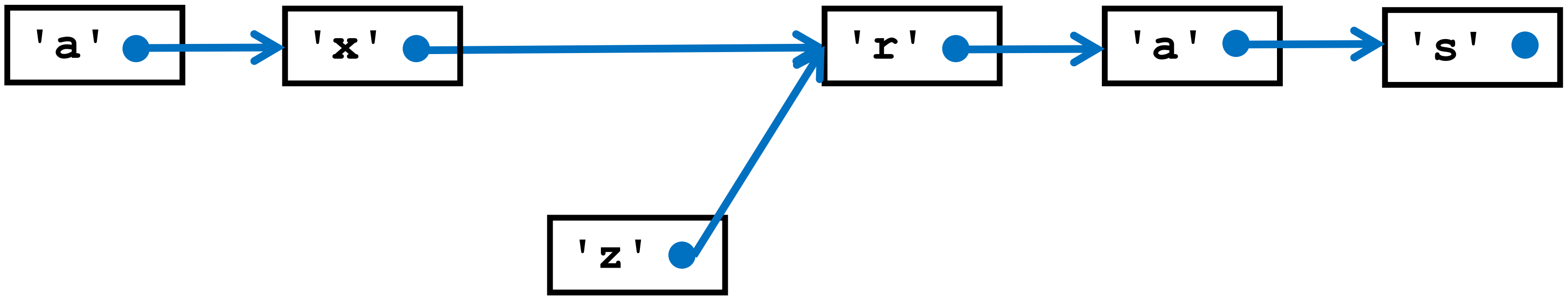
- adding to the middle of the list



- `t.add(2, 'z')`

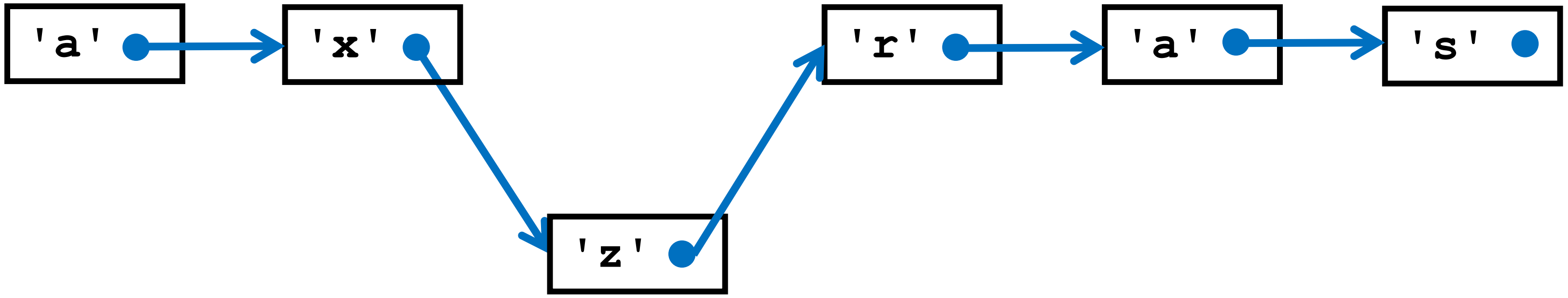
Adding to the middle of the list

- must connect to the rest of the list



Adding to the middle of the list

- then re-assign the link from the previous node



- notice that we to know the node previous to the inserted node

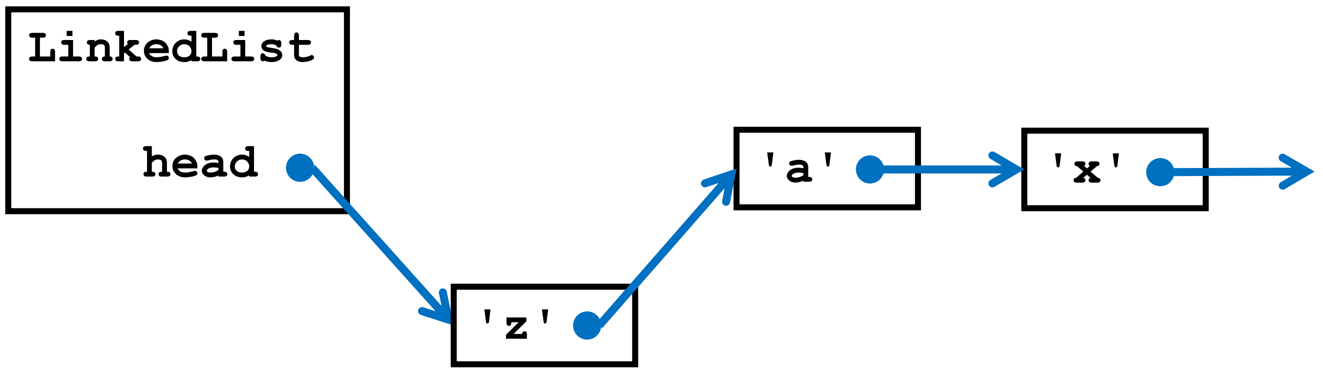
```
/**
 * Insert an element at the specified index in the list.
 *
 * @param index the index to insert at
 * @param c the character to insert
 */
public void add(int index, char c) {
  if (index < 0 || index > this.size) {
    throw new IndexOutOfBoundsException("Index: " + index + ", Size: "
          + this.size);
  }
  if (index == 0) {
    this.addFirst(c);
  }
  else {
    LinkedList.add(index - 1, c, this.head);
    this.size++;
  }
}
```

recursive method

```
/**
 * Insert an element at the specified index after the
 * specified node.
 *
 * @param index the index after prev to insert at
 * @param c the character to insert
 * @param prev the node to insert after
 */
private static void add(int index, char c, Node prev) {
  if (index == 0) {
    Node newNode = new Node(c);
    newNode.next = prev.next;
    prev.next = newNode;
    return;
  }
  LinkedList.add(index - 1, c, prev.next);
}
```

Removing from the front of the list

- removing from the front of the list

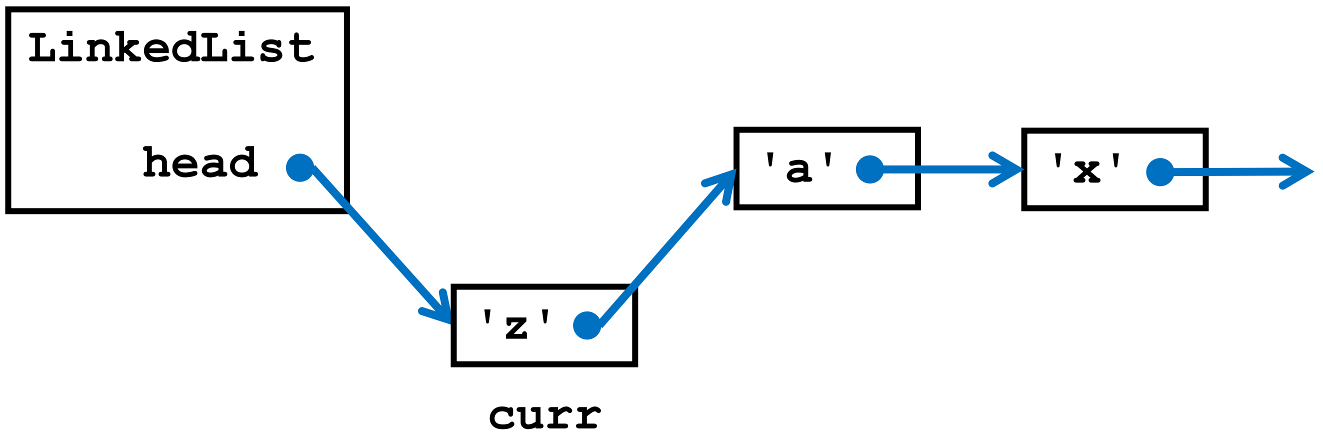


- **`t.removeFirst()`** or **`t.remove(0)`**

- also returns the element removed

Removing from the front of the list

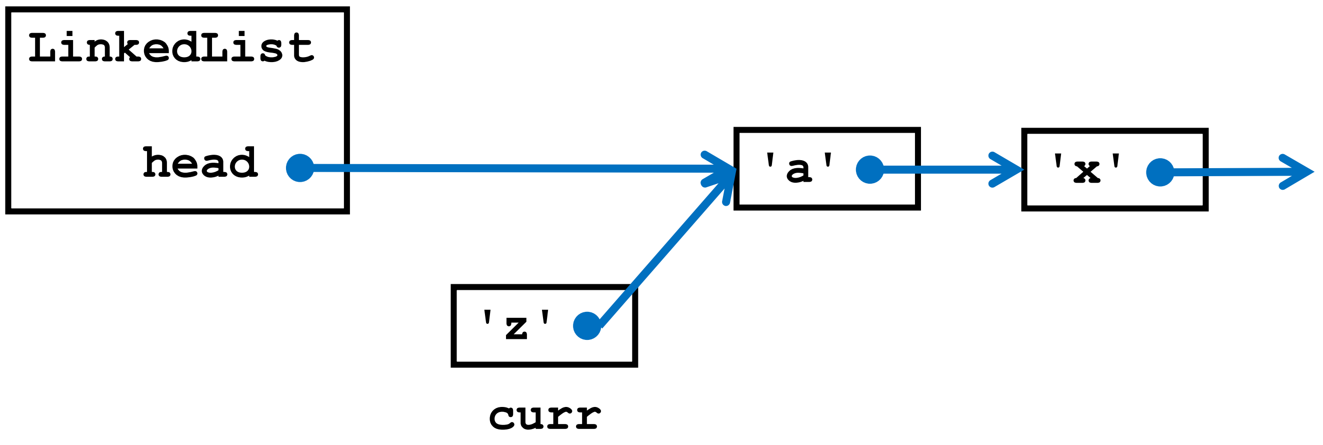
- create a reference to the node we want to remove



```
Node curr = this.head;
```

Removing from the front of the list

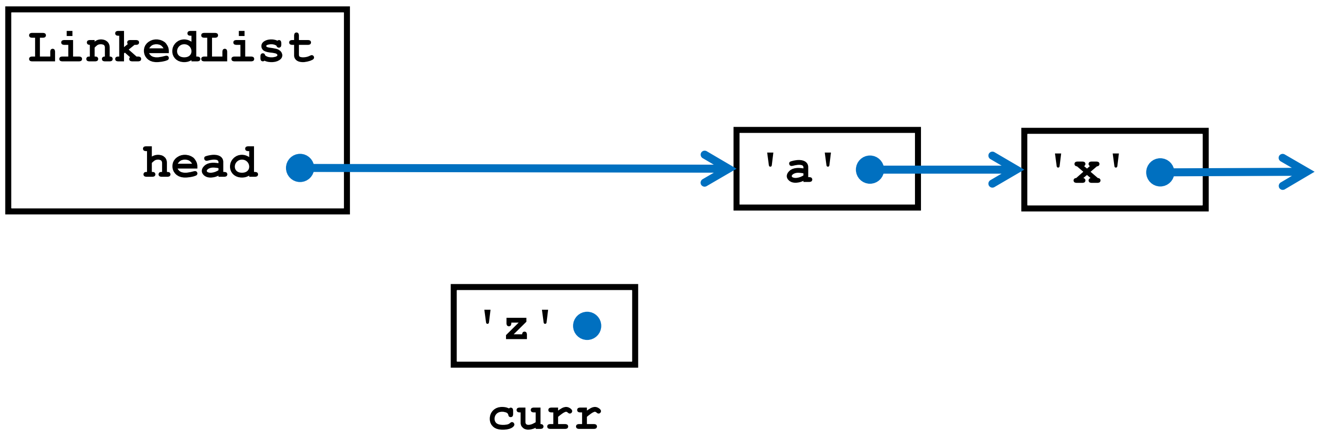
- re-assign the head node



```
this.head = curr.next;
```

Removing from the front of the list

- then remove the link from the old head node

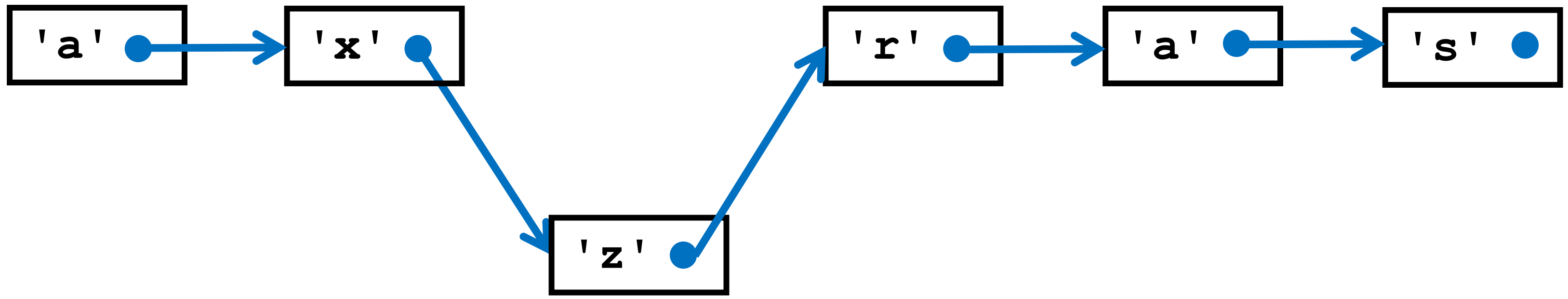


```
curr.next = null;
```

```
/**
 * Removes and returns the first element from this list.
 *
 * @return the first element from this list
 */
public char removeFirst() {
  if (this.size == 0) {
    throw new NoSuchElementException();
  }
  Node curr = this.head;
  this.head = curr.next;
  curr.next = null;
  this.size--;
  return curr.data;
}
```

Removing from the middle of the list

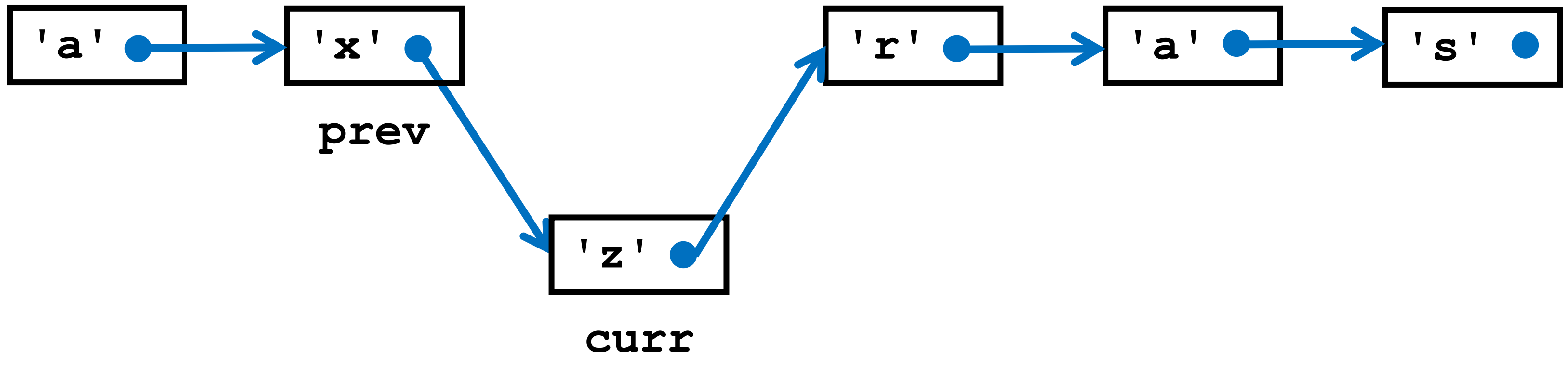
- removing from the middle of the list



- `t.remove(2)`

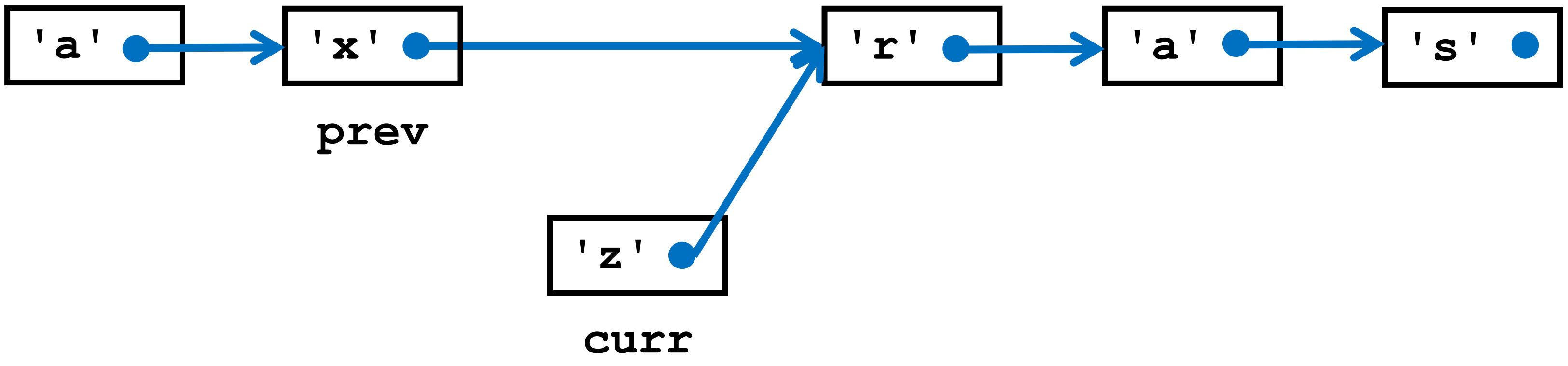
Removing from the middle of the list

- assume that we have references to the node we want to remove and its previous node



Removing from the middle of the list

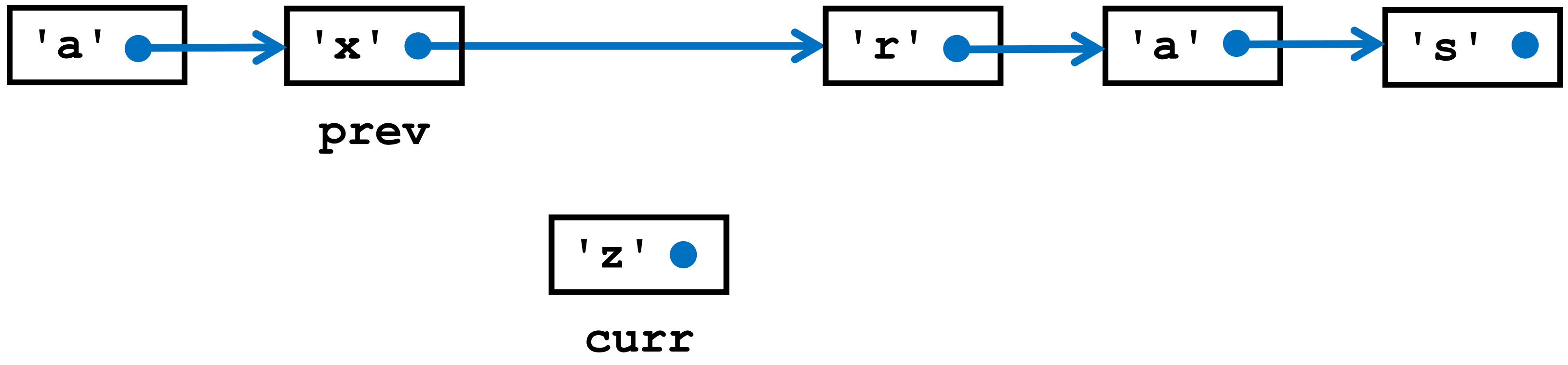
- re-assign the link from the previous node



```
prev.next = curr.next;
```

Removing from the middle of the list

- then remove the link from the current node



```
curr.next = null;
```

```
/**
 * Removes the element at the specified position in this list
 *
 * @param index the index of the element to be removed
 * @return the element previously at the specified position
 */
public char remove(int index) {
  if (index < 0 || index >= this.size) {
    throw new IndexOutOfBoundsException("Index: " + index +
                                        ", Size: " + this.size);
  }
  if (index == 0) {
    return this.removeFirst();
  }
  else {
    char result = LinkedList.remove(index - 1, this.head, this.head.next);
    this.size--;
    return result;
  }
}
```

recursive method

```
/**
 * Removes the element at the specified position relative to the
 * current node.
 *
 * @param index
 *          the index relative to the current node of the
 *          element to be removed
 * @param prev
 *          the node previous to the current node
 * @param curr
 *          the current node
 * @return the element previously at the specified position
 */
private static char remove(int index, Node prev, Node curr) {
  if (index == 0) {
    prev.next = curr.next;
    curr.next = null;
    return curr.data;
  }
  return LinkedList.remove(index - 1, curr, curr.next);
}
```

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

`public interface Iterator<E>`

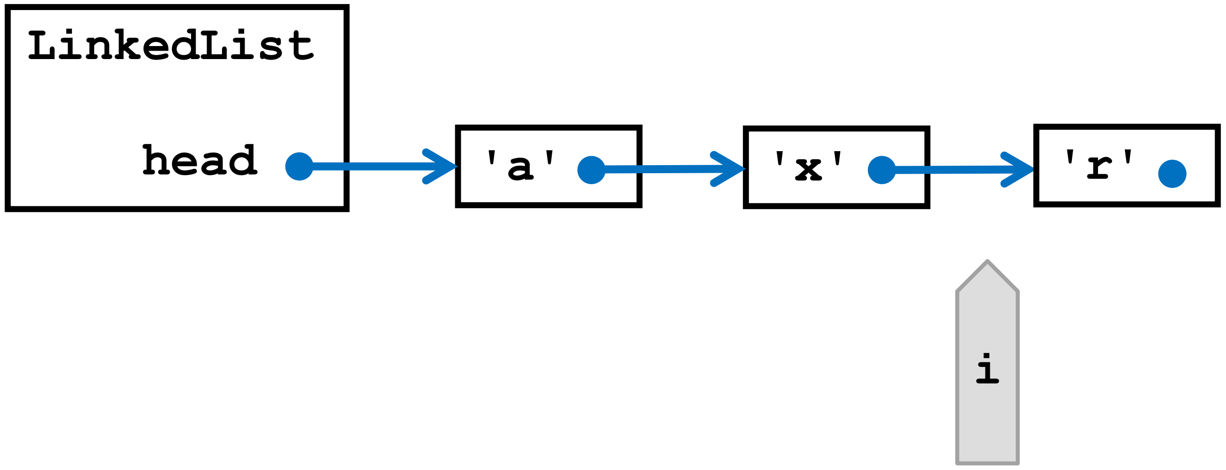
An iterator over a collection.

`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).

Recursive Objects (Part 3)

LinkedList Iterator

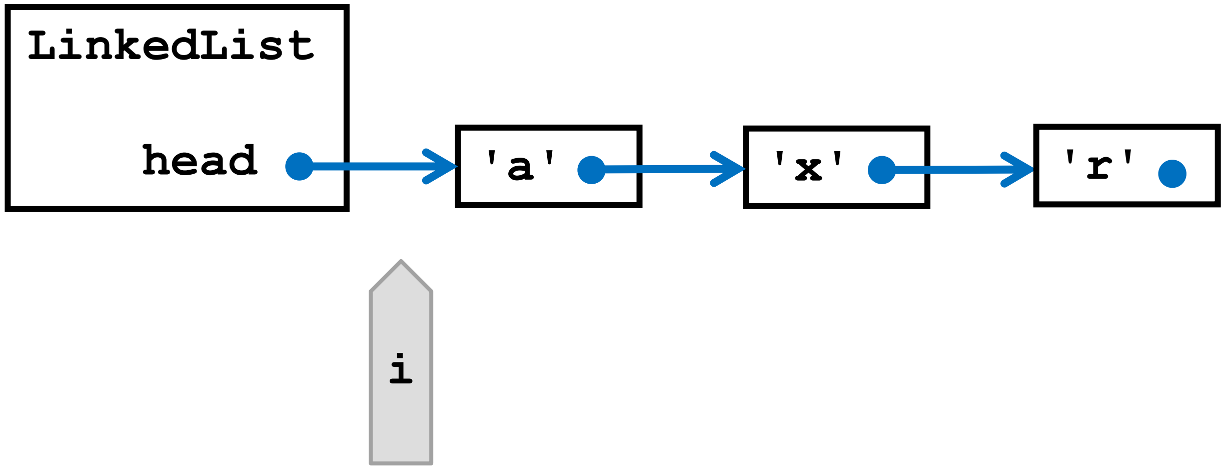
- think of the iterator as lying between elements in the list (like a cursor)



iterator between 'x' and 'r'

LinkedList Iterator

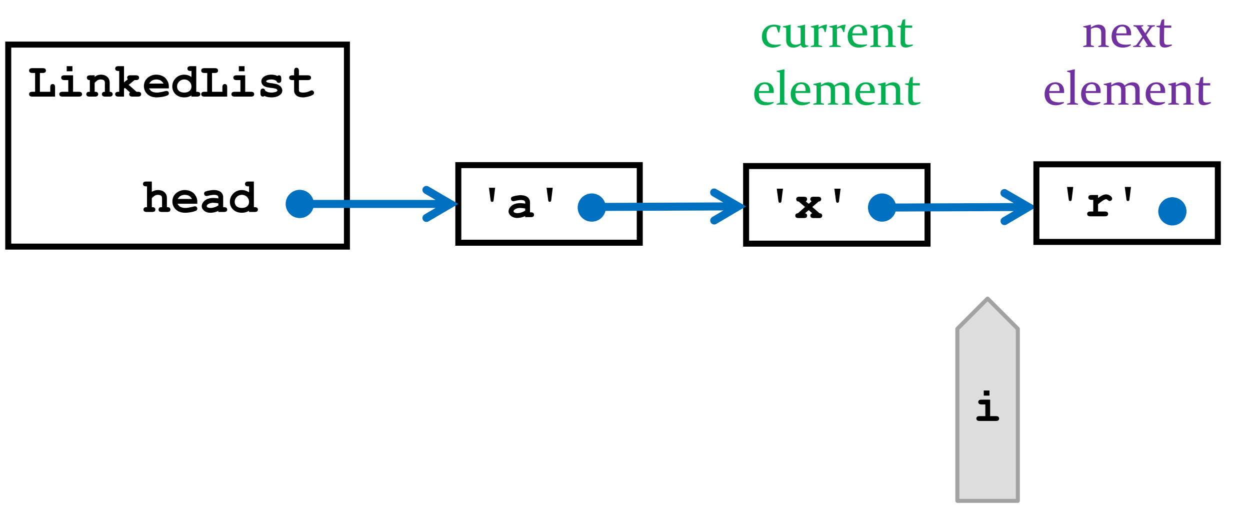
- think of the iterator as lying between elements in the list (like a cursor)



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

LinkedList Iterator

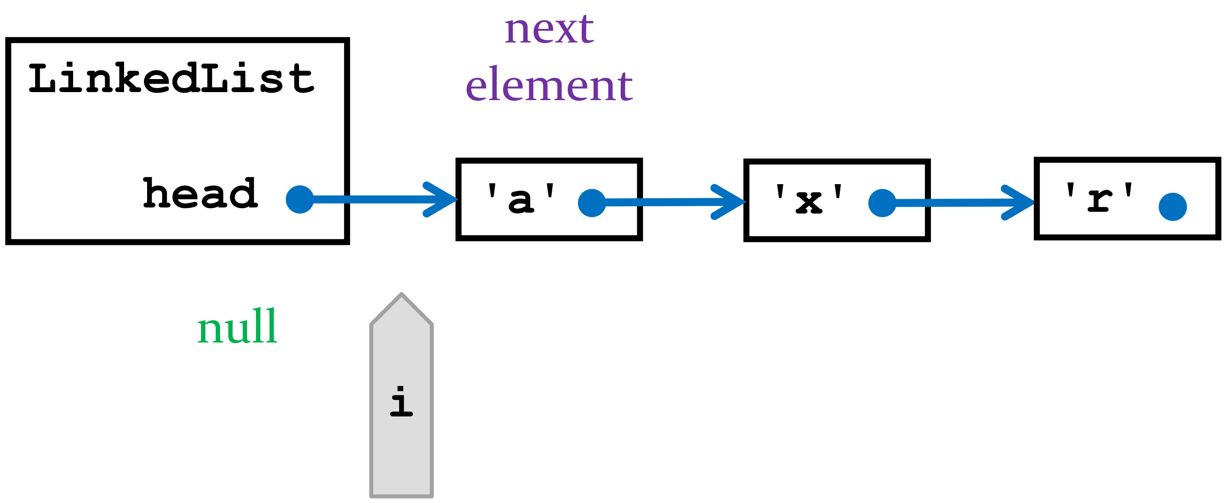
- because the iterator is between elements, there is a current element and next element of the iteration



iterator between 'x' and 'r'

LinkedList Iterator

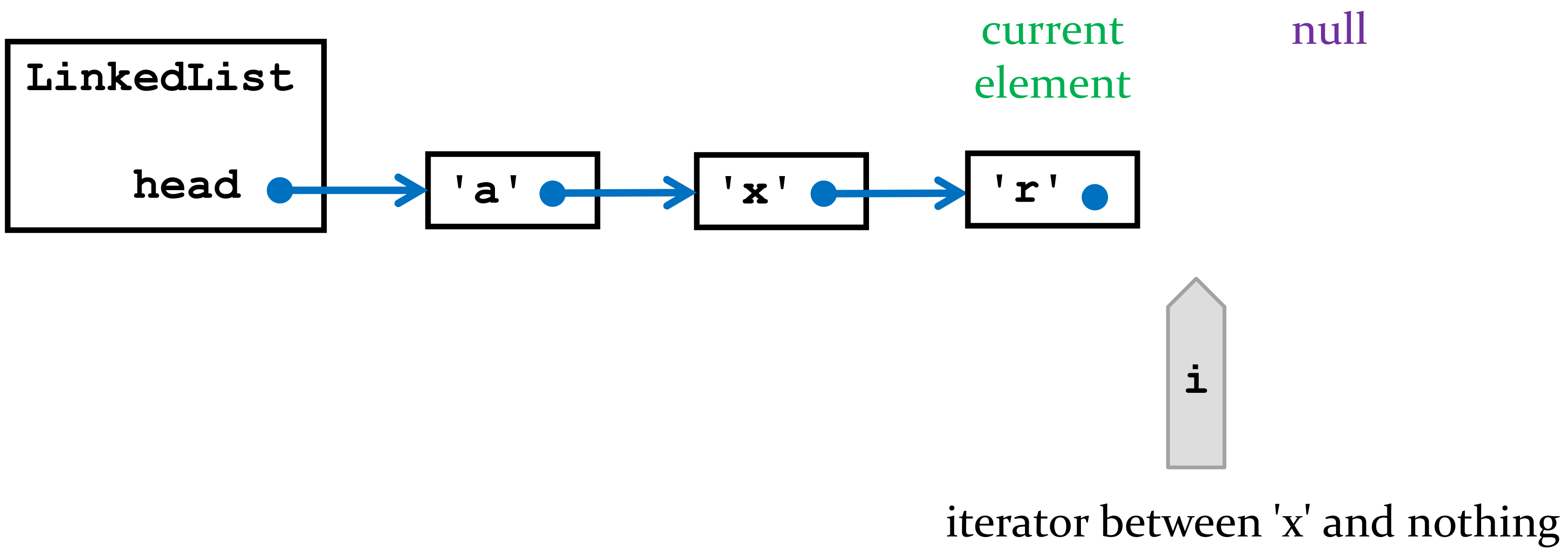
- the current element is **`null`** at the start of the iteration



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

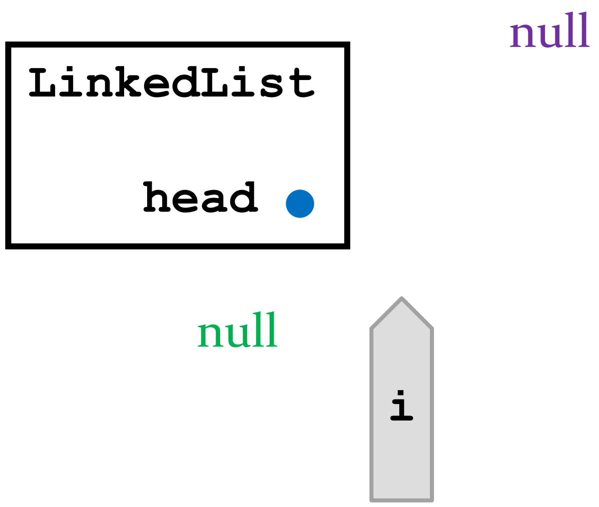
LinkedList Iterator

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



LinkedList Iterator

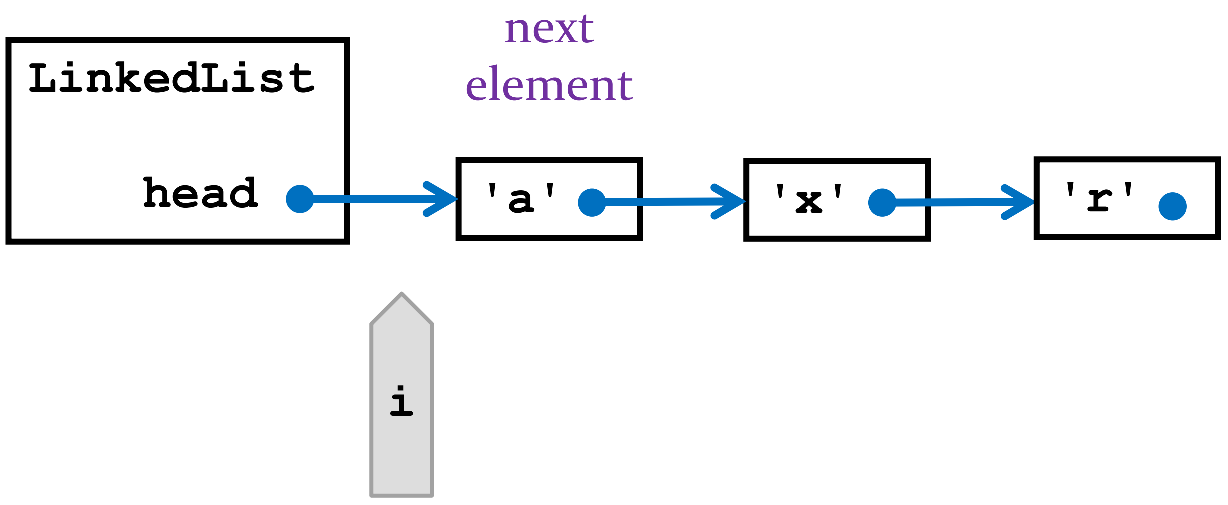
- both the current and next elements are **`null`** if the list is empty



iterator at the start of the iteration

LinkedList Iterator: hasNext

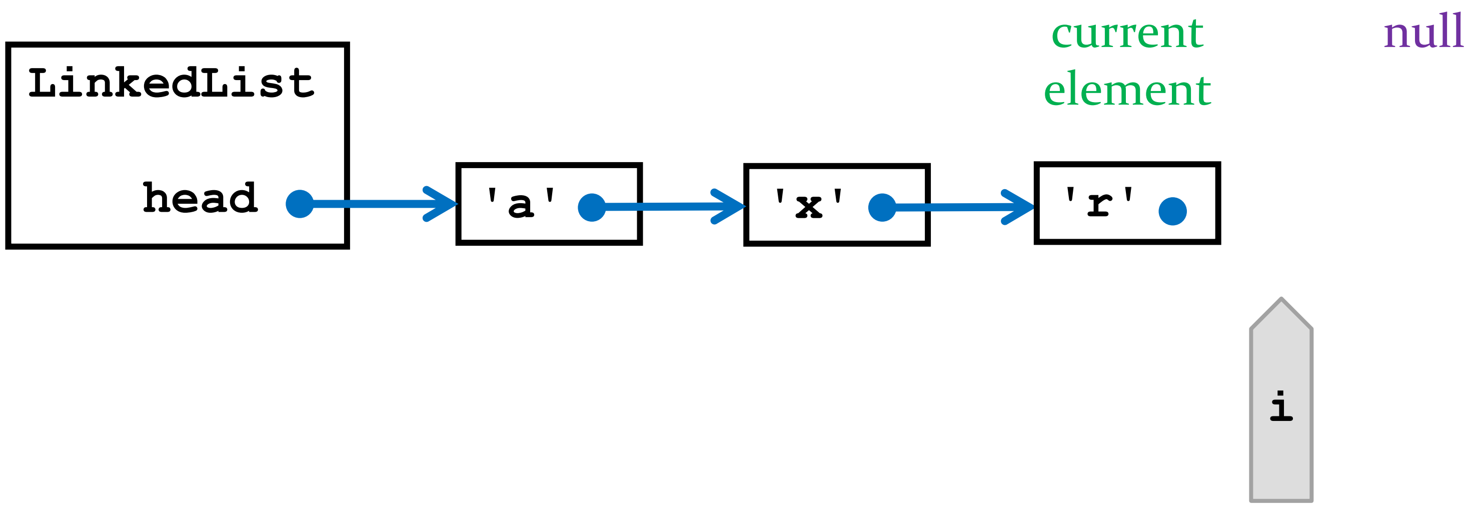
- **`hasNext()`** returns true if there is at least one more element in the iteration



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

LinkedList Iterator: hasNext

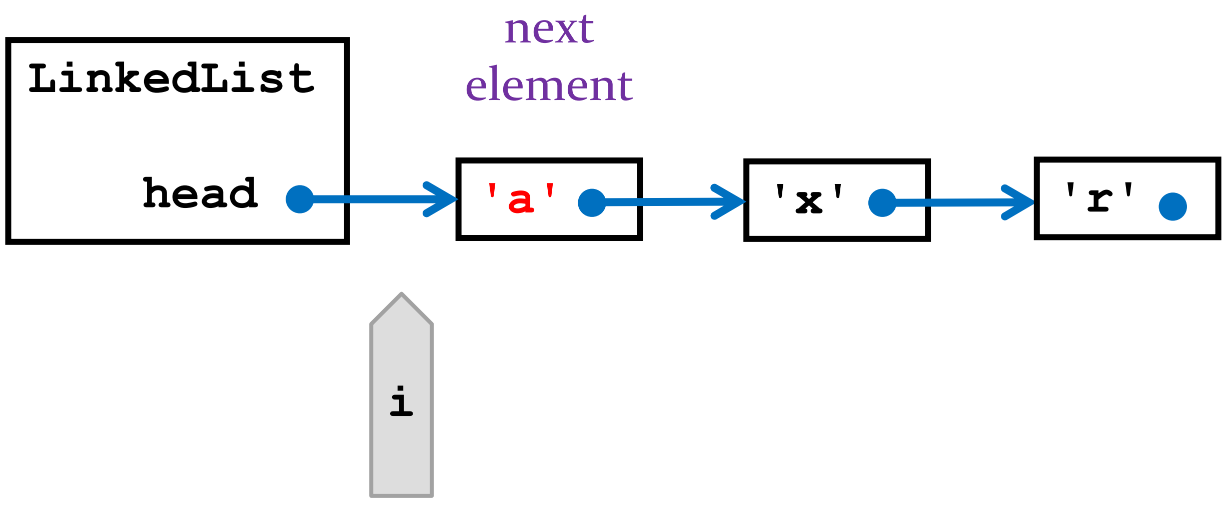
- **`hasNext()`** returns false at the end of the iteration



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

LinkedList Iterator: next

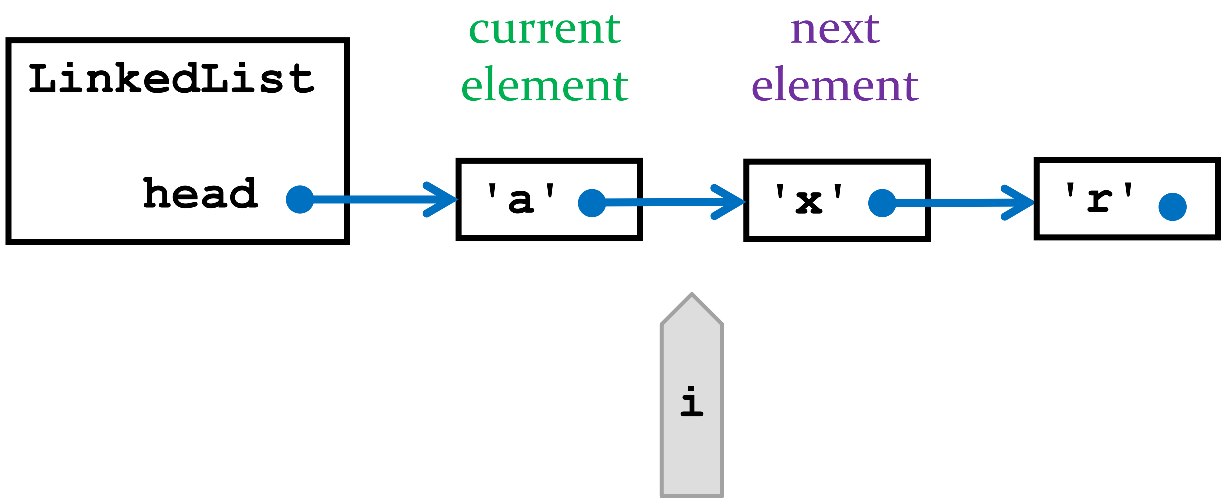
- invoking **`next()`** returns the next element...



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

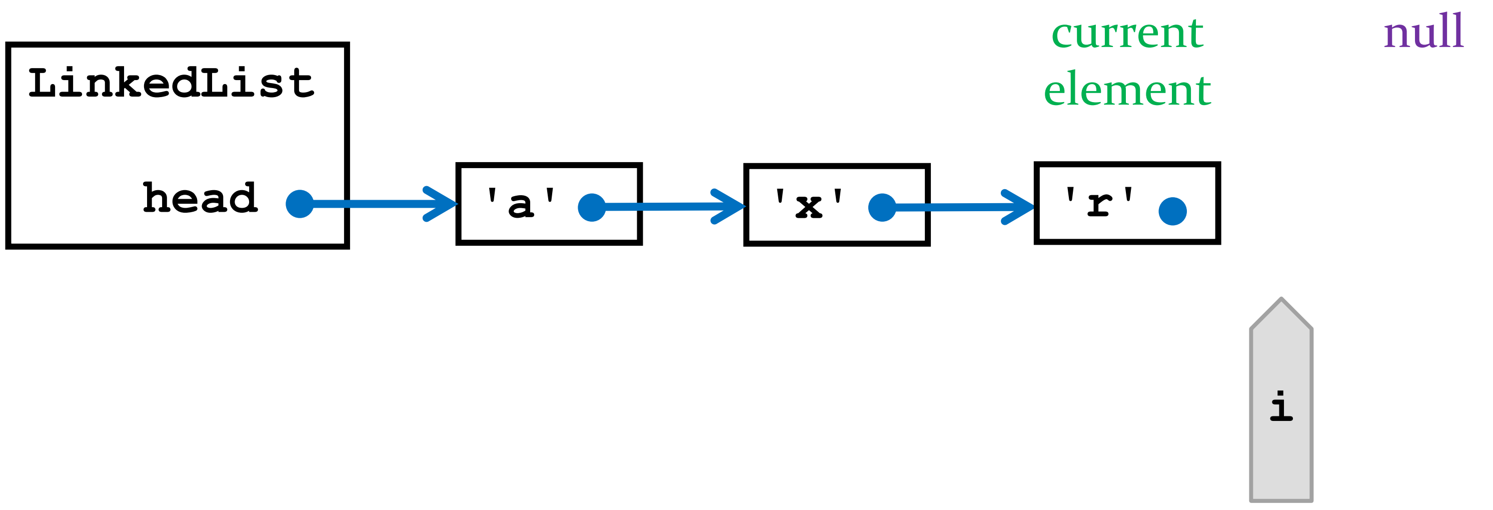
LinkedList Iterator: next

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



LinkedList Iterator: next

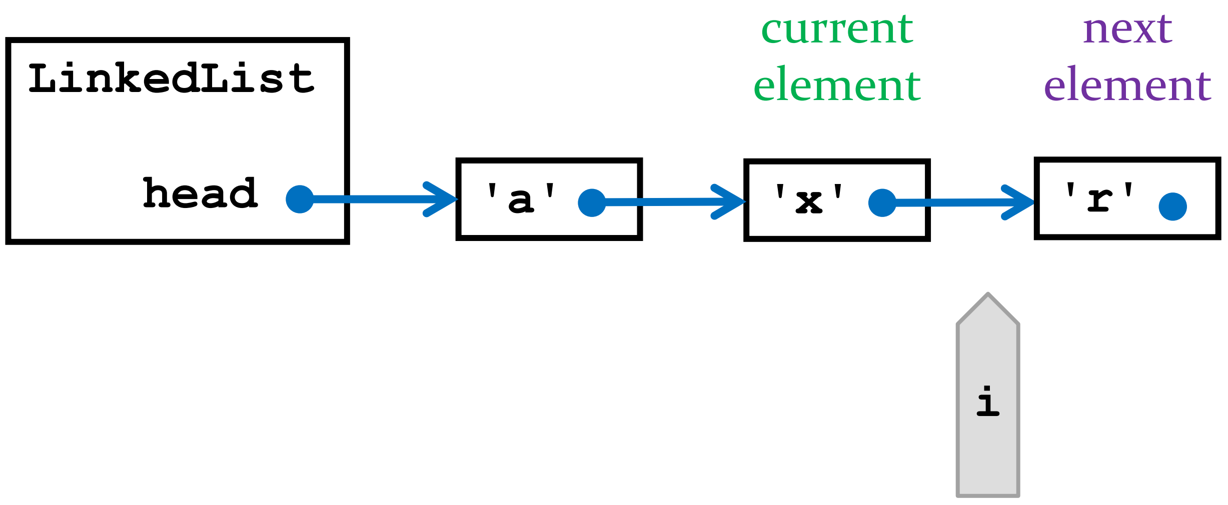
- invoking **`next()`** at the end of the iteration causes a **`NoSuchElementException`** to be thrown



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

LinkedList Iterator: remove

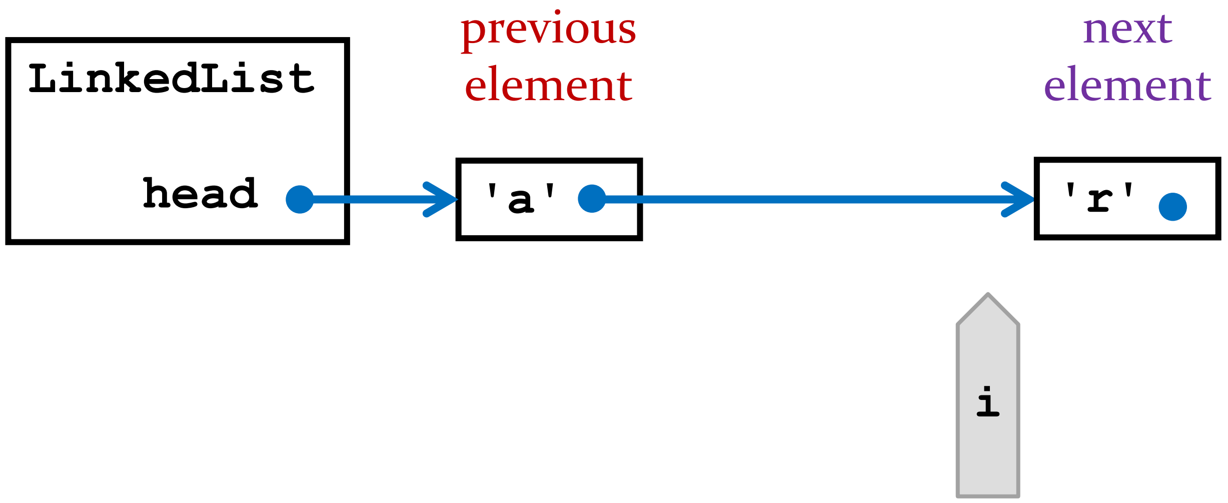
- **`remove()`** causes the element most recently returned by **`next()`** to be removed from the linked list



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

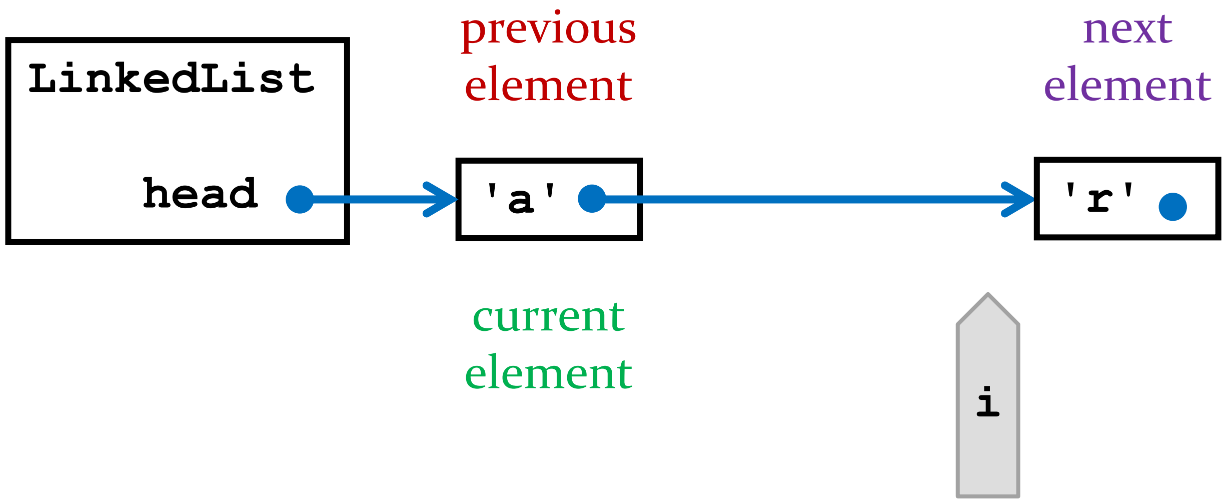
LinkedList Iterator: remove

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



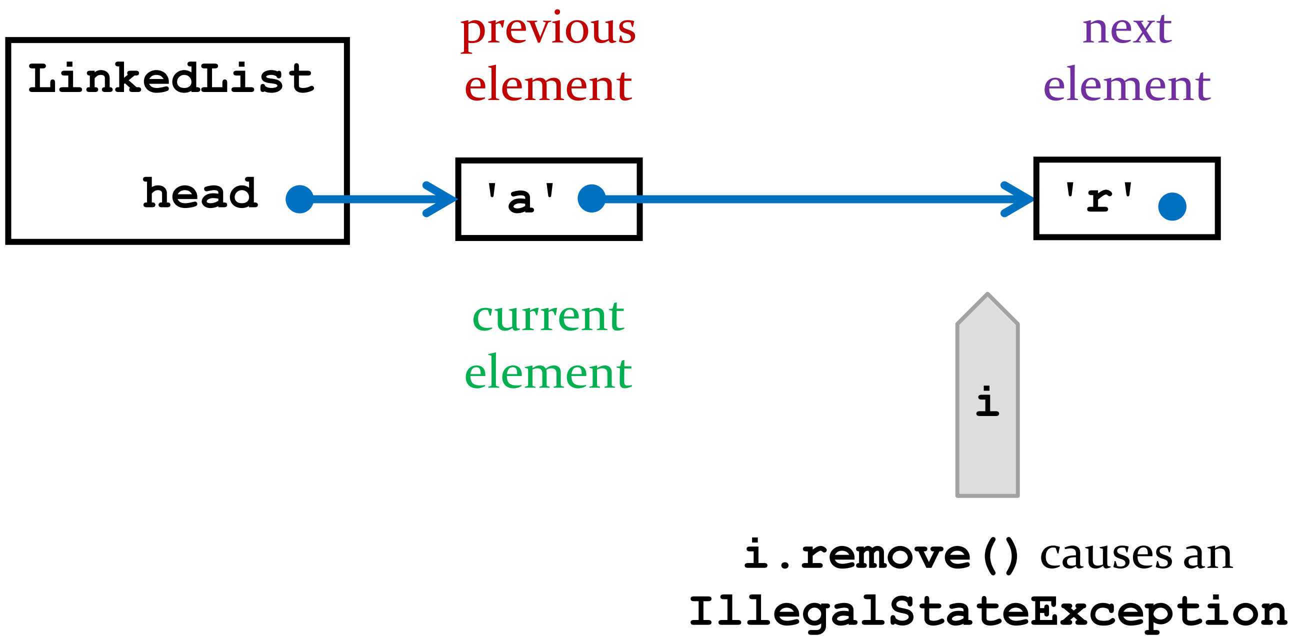
LinkedList Iterator: remove

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



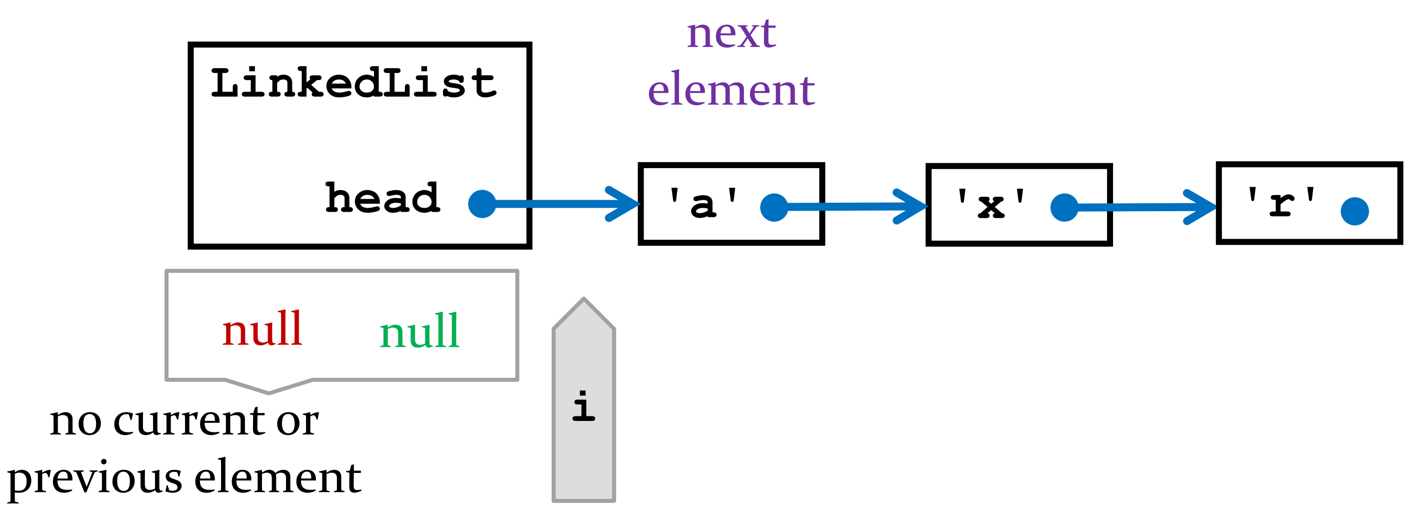
LinkedList Iterator: remove

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



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

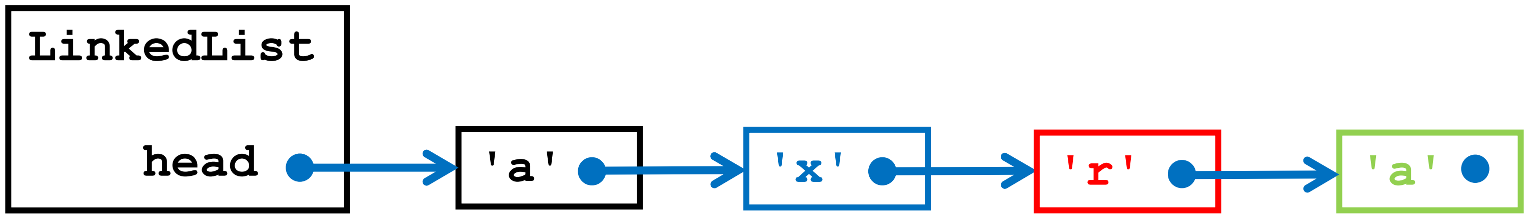
```
@Override
public Character next() {
  if (!this.hasNext()) {
    throw new NoSuchElementException();
  }
  this.prevNode = this.currNode;
  if (this.currNode == null) {
    this.currNode = head;
  }
  else {
    this.currNode = this.currNode.next;
  }
  return this.currNode.data;
}
```

Implementation: remove

```java
@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

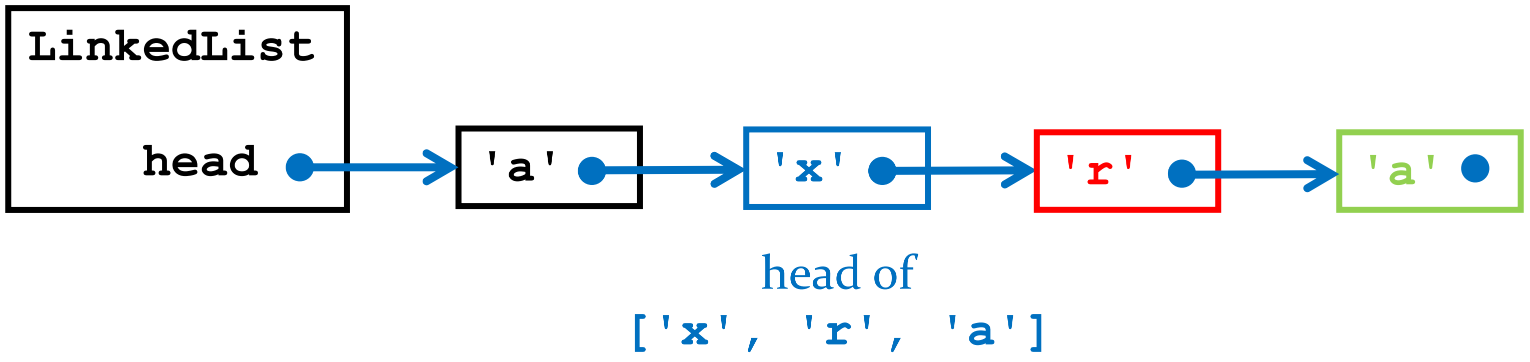
- each node can be thought of as the head of a smaller list



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

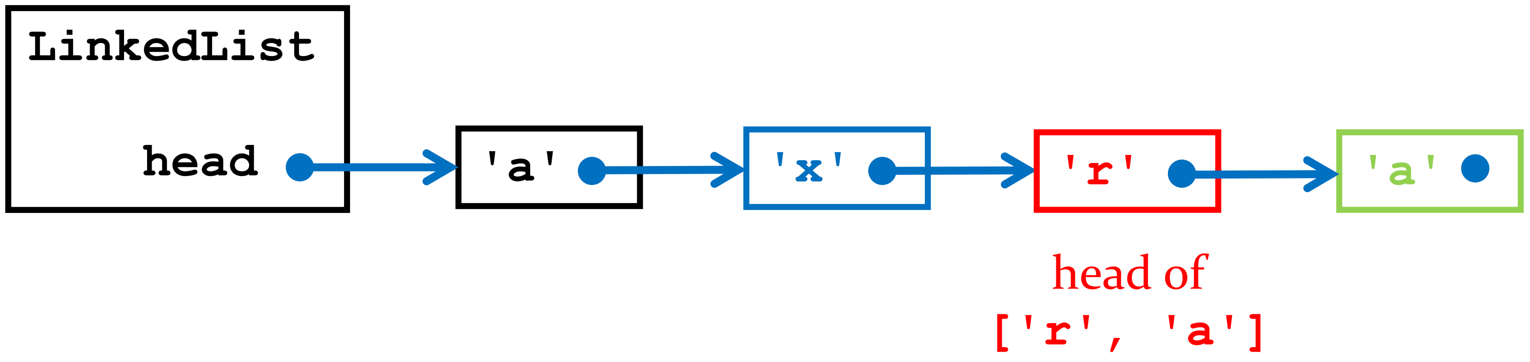
LinkedList Summary

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



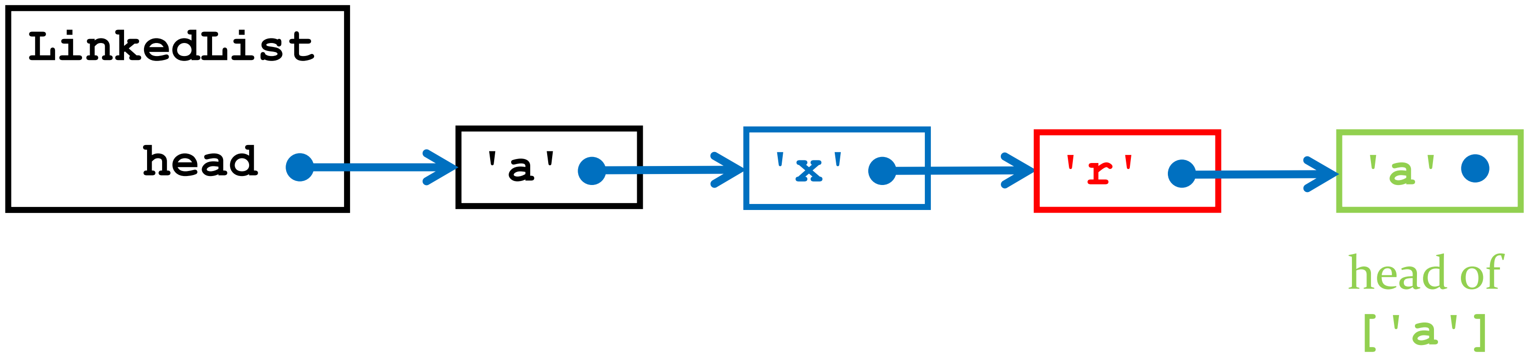
LinkedList Summary

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



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

`public interface Iterator<E>`

An iterator over a collection.

`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).