

Implementing Linked Lists

Based on slides by Prof. Burton Ma

Recursive Objects

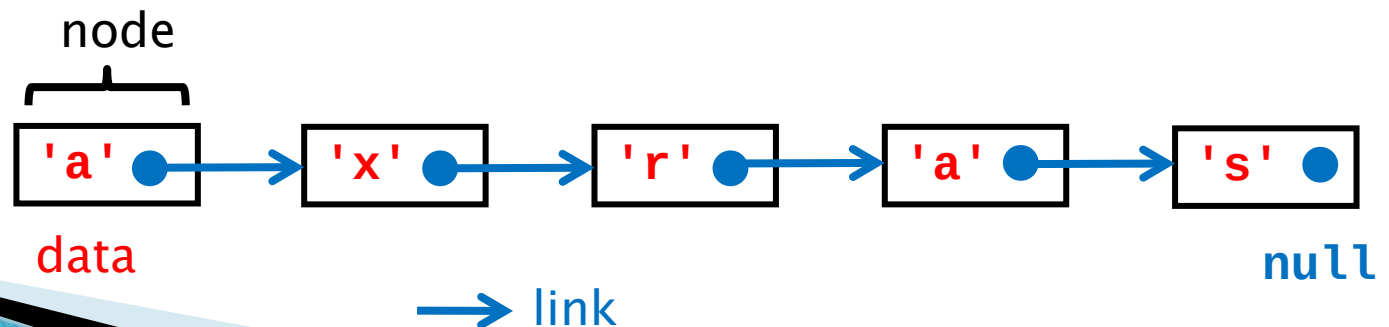
- ▶ An object that holds a reference to its own type is a recursive object
 - Linked lists and trees are classic examples in computer science of objects that can be implemented recursively

Data Structures

- ▶ Data structures (and algorithms) are one of the foundational elements of computer science
- ▶ A data structure is a way to organize and store data so that it can be used efficiently
 - List – sequence of elements
 - Set – a group of unique elements
 - Map – access elements using a key
 - many more...

Linked Lists

- ▶ A data structure made up of a sequence of nodes
- ▶ Each node has
 - Some data
 - A field that contains a reference (a *link*) to the next node in the sequence
- ▶ Suppose we have a linked list that holds chars; a picture of our linked list would be:



UML Class Diagram

LinkedList
- size : int - head : Node
...

Node
- data : char - next : Node
...

Node

- ▶ Nodes are implementation details that the client does not need to know about
 - Can be private inner classes

```
public class LinkedList
{

    private static class Node
    {
        private char data;
        private Node next;

        public Node(char c)
        {
            this.data = c;
            this.next = null;
        }
    }

    // ...
}
```

LinkedList constructor

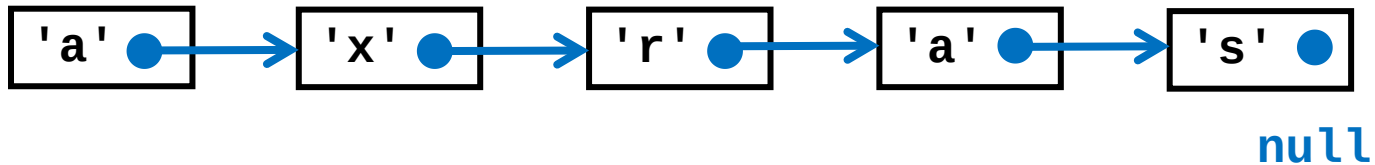
```
/**  
 * Create a linked list of size 0.  
 */  
public LinkedList()  
{  
    this.size = 0;  
    this.head = null;  
}
```


Node constructor

```
/**  
 * Create a node with the given character.  
 */  
public Node(char c)  
{  
    this.data = c;  
    this.next = null;  
}
```

Creating a Linked List

- ▶ To create the following linked list:



```
LinkedList t = new LinkedList();  
t.add('a');  
t.add('x');  
t.add('r');  
t.add('a');  
t.add('s');
```

Add to end of list

- ▶ Methods of recursive objects can often be implemented with a recursive algorithm
 - Notice the word "can"; the recursive implementation is not necessarily the most efficient implementation
- ▶ Adding to the end of the list can be done recursively
 - Base case: at the end of the list
 - I.e., **next** is **null**
 - Create new node and append it to this link
 - Recursive case: current link is not the last link
 - Add to the end of **next**

```
/**
 * Adds the given character to the end of the list.
 *
 * @param c The character to add
 */
public void add(char c)
{
    if (this.size == 0)
    {
        this.head = new Node(c);
    }
    else
    {
        LinkedList.add(c, this.head);
    }
    this.size++;
}
```

```
/**  
 * Adds the given character to the end of the list.  
 *  
 * @param c The character to add  
 * @param node The node at the head of the current sublist  
 */  
private static void add(char c, Node node)  
{  
    if (node.next == null)  
    {  
        node.next = new Node(c);  
    }  
    else  
    {  
        LinkedList.add(c, node.next);  
    }  
}
```

Getting an Element in the List

- ▶ A client may wish to retrieve the i th element from a list
 - The ability to access arbitrary elements of a sequence in the same amount of time is called *random access*
 - Arrays support random access; linked lists do not
- ▶ To access the i th element in a linked list we need to sequentially follow the first $(i - 1)$ links



`t.get(3)` **link 0** **link 1** **link 2**

- Takes $O(n)$ time versus $O(1)$ for arrays

Getting an Element in the List

- ▶ Validation?
- ▶ Setting the i th element can be done recursively
 - Base case: **index == 0**
 - Return the value held by the current link
 - Recursive case: current link is not the last link
 - Get the element at **index - 1** starting from **next**

```

/**
 * Returns the item at the specified position
 * in the list.
 *
 * @param index Index of the element to return
 * @return the element at the specified position
 * @throws IndexOutOfBoundsException if the index
 *         is out of the range
 *         {@code (index < 0 || index >= list size)}
 */
public char get(int index)
{
    if (index < 0 || index >= this.size)
    {
        throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + this.size);
    }
    return LinkedList.get(index, this.head);
}

```



```

/**
 * Returns the item at the specified position
 * in the list.
 *
 * @param index Index of the element to return
 * @param node The node at the head of the current sublist
 * @return the element at the specified position
 */
private static char get(int index, Node node)
{
    if (index == 0)
    {
        return node.data;
    }
    return LinkedList.get(index - 1, node.next);
}

```

Setting an Element in the List

- ▶ Setting the i th element is almost exactly the same as getting the i th element

```

/**
 * Sets the element at the specified position
 * in the list.
 *
 * @param index index of the element to set
 * @param c new value of element
 * @throws IndexOutOfBoundsException if the index
 *         is out of the range
 *         {@code (index < 0 || index >= list size)}
 */
public void set(int index, char c)
{
    if (index < 0 || index >= this.size)
    {
        throw new IndexOutOfBoundsException("Index: " + index +
                                           ", Size: " + this.size);
    }
    LinkedList.set(index, c, this.head);
}

```

```

/**
 * Sets the element at the specified position
 * in the list.
 *
 * @param index index of the element to set
 * @param c new value of the element
 * @param node The node at the head of the current sublist
 */
private static void set(int index, char c, Node node)
{
    if (index == 0)
    {
        node.data = c;
        return;
    }
    LinkedList.set(index - 1, c, node.next);
}

```

toString

- ▶ Finding the string representation of a list can be done recursively



- The string is
`"[a, x, r, a, s]"`
- The string is
`"[" + "a, " + toString(the list['x', 'r', 'a', 's'])"`

toString

- Base case: `next` is `null`
 - Return the value of the link as a string + "]"
- Recursive case: current link is not the last link
 - Return the value of the link as a string + ", " + the rest of the list as a string

```
public String toString()
{
    if (this.size == 0)
    {
        return "[]";
    }
    return "[" + LinkedList.toString(this.head);
}
```

```
private static String toString(Node n)
{
    if (n.next == null)
    {
        return n.data + "[]";
    }
    String s = n.data + ", ";
    return s + LinkedList.toString(n.next);
}
```

Finding an element in the list

- ▶ Often useful to ask if a list contains a particular element
- ▶ Worst case: must visit every element of the list



- E.g., `t.contains('z')`

Finding an element in the list

- ▶ Contains can be solved recursively
 - Base case: found the character we are looking for
 - I.e., node **data** is equal to the character we are searching for
 - return true
 - Base case: at the end of the list
 - I.e., node **next** is **null**
 - return false
 - Recursive case: have not found the character we are searching for and not at the end of the list
 - Search the sublist starting at **node.next**
 - return result

```
/**
 * Returns <code>true</code> if this list contains the specified element.
 *
 * @param c element to search for
 * @return <code>true</code> if this list contains the
 * specified element
 */
public boolean contains(char c)
{
    if (this.size == 0)
    {
        return false;
    }
    return LinkedList.contains(c, this.head);
}
```

```

/**
 * Returns <code>true</code> if this list contains the specified element.
 *
 * @param c element to search for
 * @param node the node at the head of the current sublist
 * @return <code>true</code> if this list contains the
 * specified element
 */
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);
}

```

Finding an element in the list

- ▶ Closely related to contains is finding the index of an element in the list
- ▶ Worst case: must visit every element of the list



- E.g., `t.indexOf('s')`

Finding an element in the list

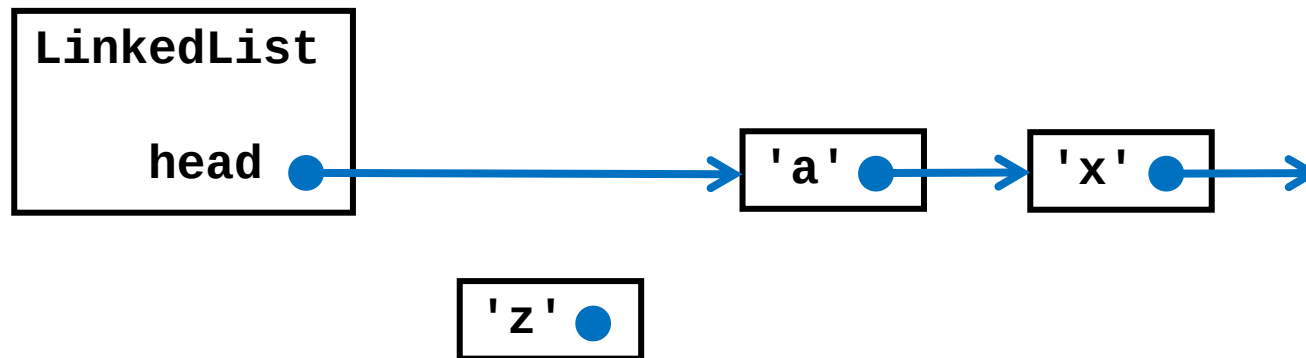
- ▶ **indexOf** can be solved recursively
 - Base case: found the character we are looking for
 - I.e., node **data** is equal to the character we are searching for
 - return 0
 - Base case: at the end of the list
 - I.e., node **next** is **null**
 - return -1
 - Recursive case: have not found the character we are searching for and not at the end of the list
 - Search the sublist starting at **node.next**
 - return 1 + result

```
/**
 * Returns the index of the first occurrence of the
 * specified element in this list, or -1 if this list
 * does not contain the element.
 *
 * @param c
 *         element to search for
 * @return the index of the first occurrence of the
 *         specified element in this list, or -1 if this
 *         list does not contain the element
 */
public int indexOf(char c)
{
    if (this.size == 0)
    {
        return -1;
    }
    return LinkedList.indexOf(c, this.head);
}
```

```
private static int indexOf(char c, Node n)
{
    if (n.data == c)
    {
        return 0;
    }
    if (n.next == null)
    {
        return -1;
    }
    int i = LinkedList.indexOf(c, n.next);
    if (i == -1)
    {
        return -1;
    }
    return 1 + i;
}
```

Adding to the front of the list

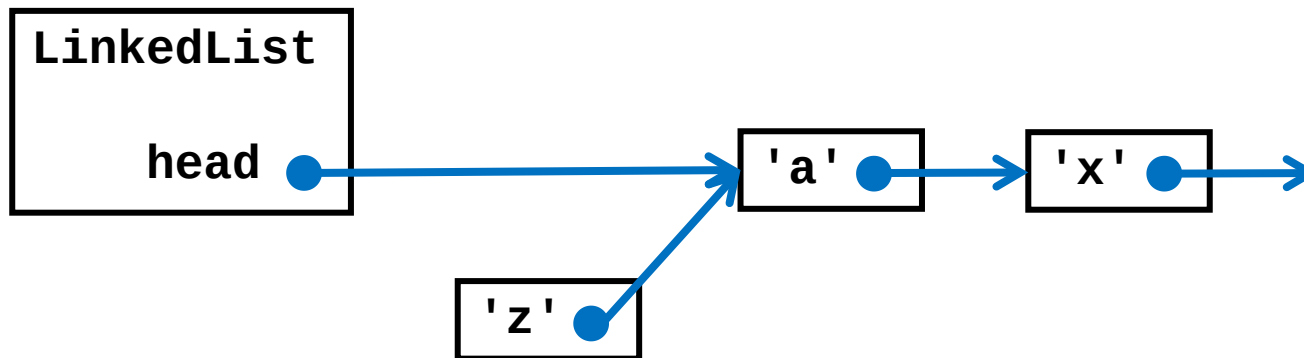
- ▶ Adding to the front of the list



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

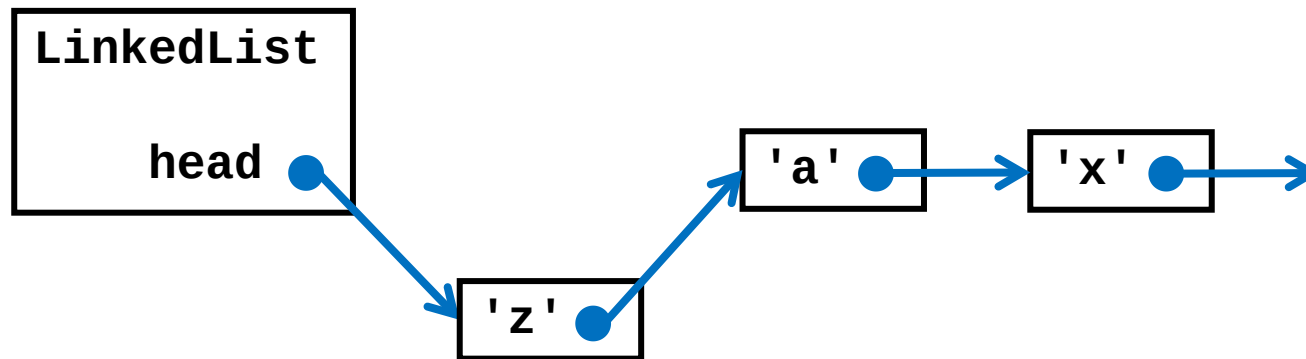
Adding to the front of the list

- ▶ Must connect to the rest of the list



Adding to the front of the list

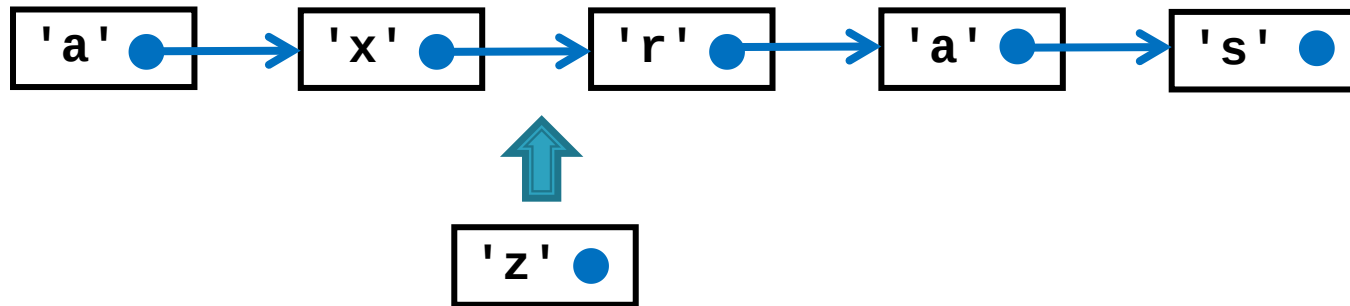
- ▶ Then re-assign head of linked list



```
/**  
 * 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

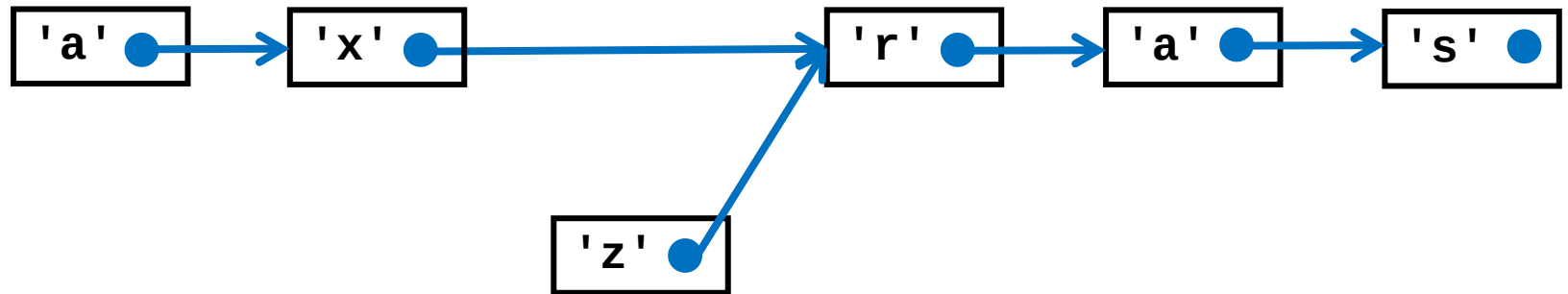
- ▶ Adding to the middle of the list



- ▶ `t.add(2, 'z')`

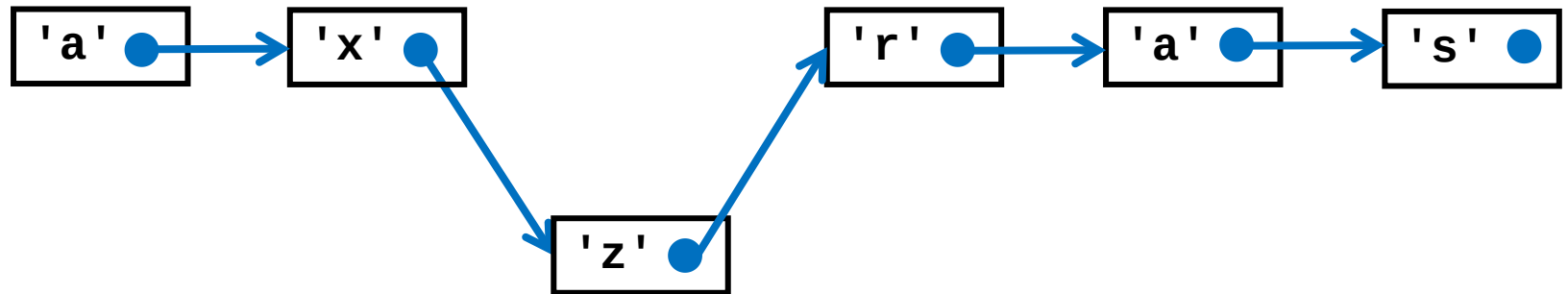
Adding to the middle of the list

- ▶ Must connect to the rest of the list



Adding to the middle of the list

- ▶ Then re-assign the link from the previous node



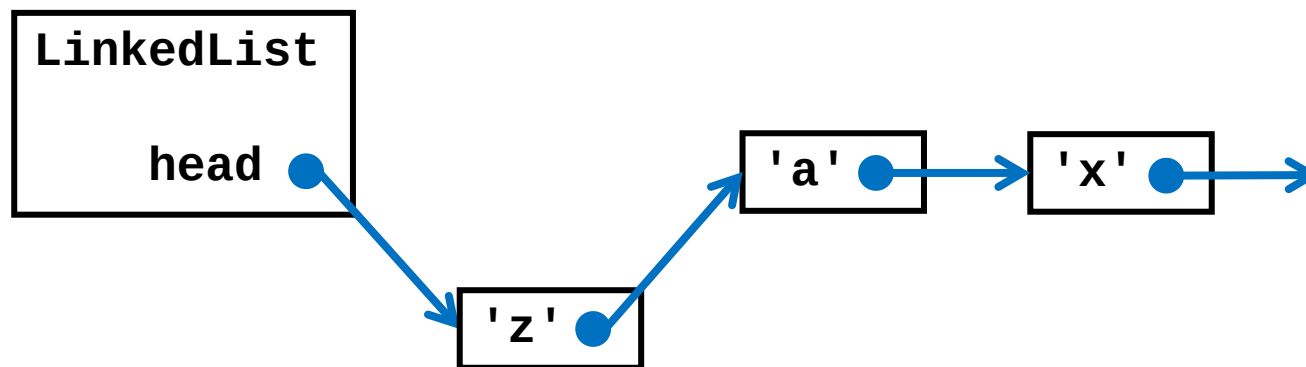
- ▶ Notice that we to know the node previous to the inserted node

```
/**
 * Insert an element at the specified index 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);
}
```

```
/**
 * 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++;
    }
}
```


Removing from the front of the list

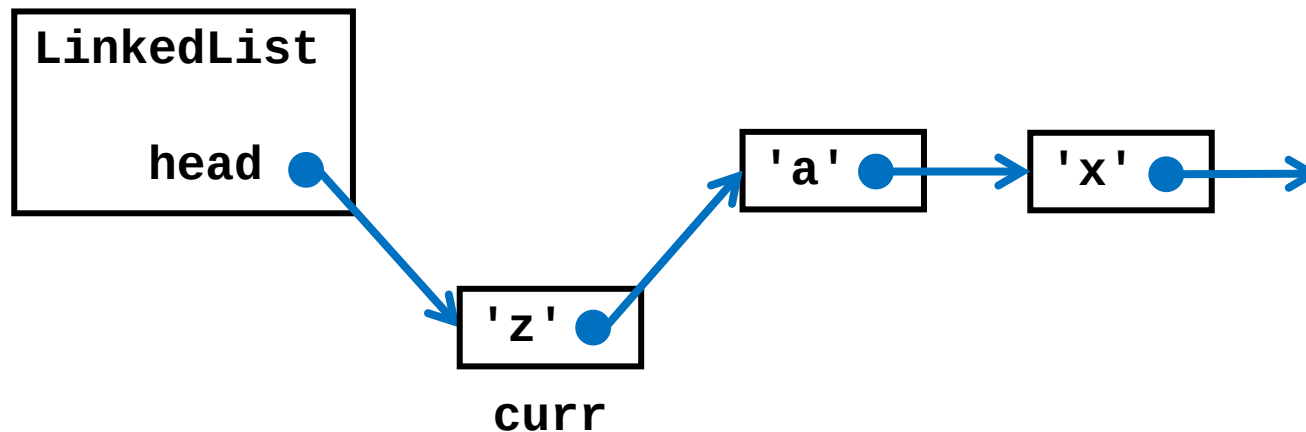
- ▶ Removing from the front of the list



- ▶ `t.removeFirst()` or `t.remove(0)`
- ▶ Also returns the element removed

Removing from the front of the list

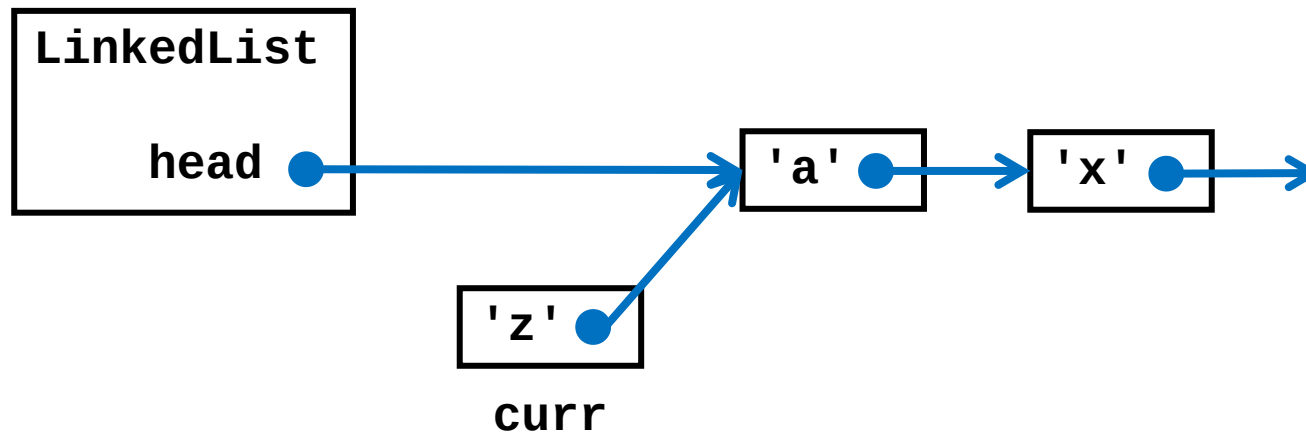
- ▶ Create a reference to the node we want to remove



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

Removing from the front of the list

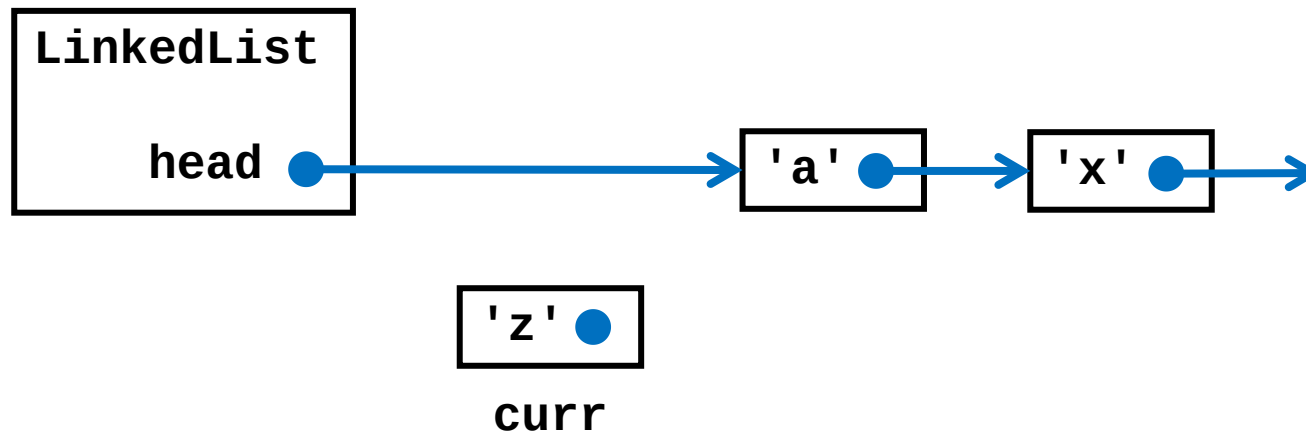
- Re-assign the head node



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

Removing from the front of the list

- ▶ Then remove the link from the old head node

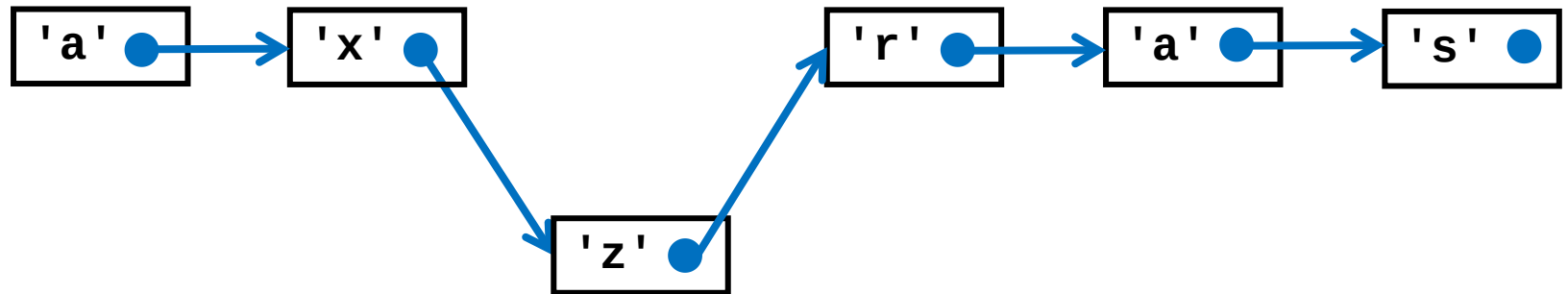


```
curr.next = null;
```

```
/**
 * 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

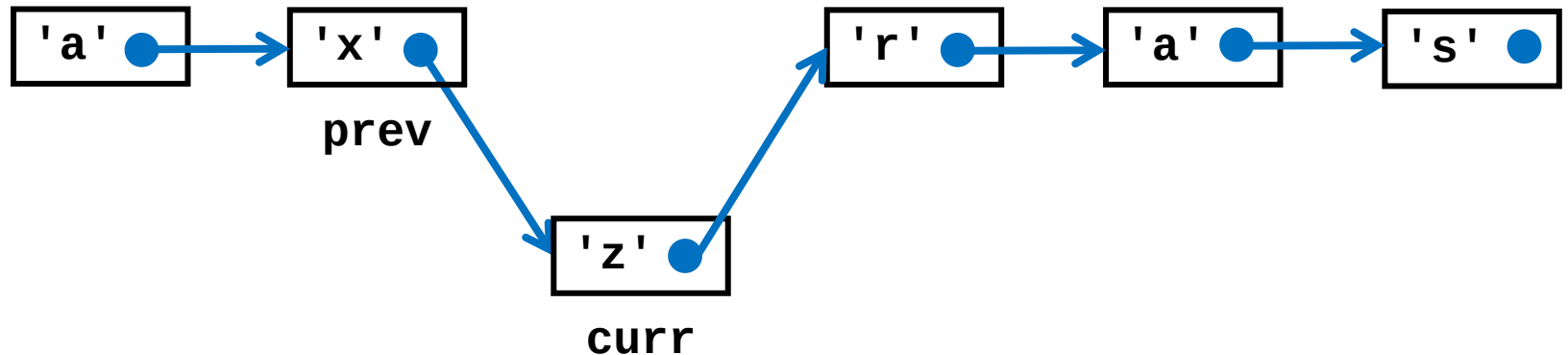
- ▶ Removing from the middle of the list



`t.remove(2)`

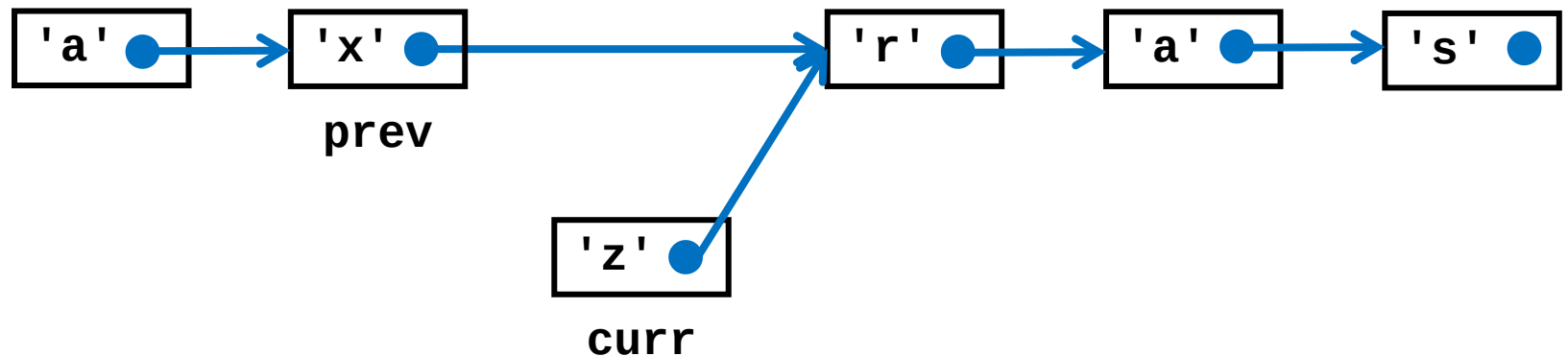
Removing from the middle of the list

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



Removing from the middle of the list

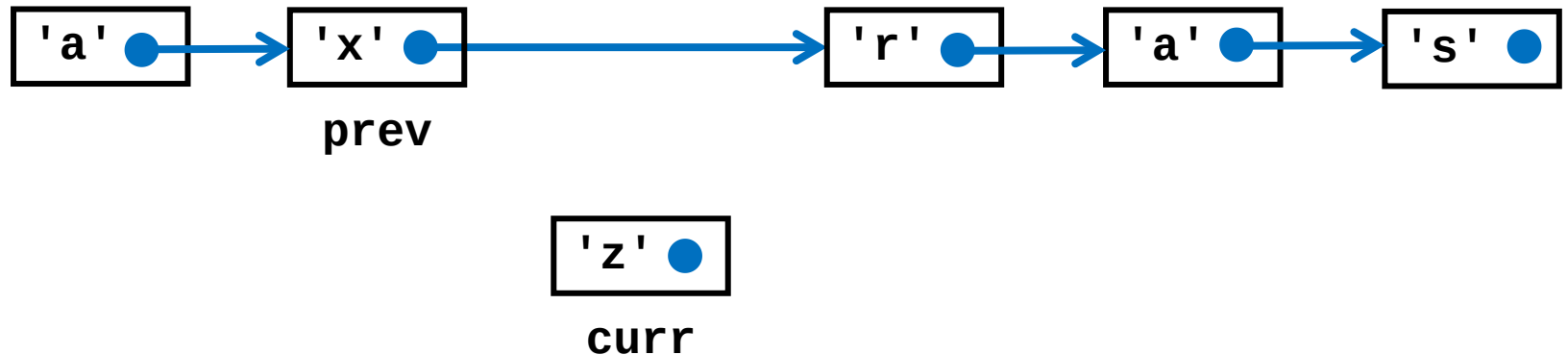
- Re-assign the link from the previous node



```
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 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);
}

```

```

/**
 * 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;
    }
}

```

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>

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

public interface Iterator<E>

boolean

hasNext()

Returns true if the iteration has more elements.

E

next()

Returns the next element in the iteration.

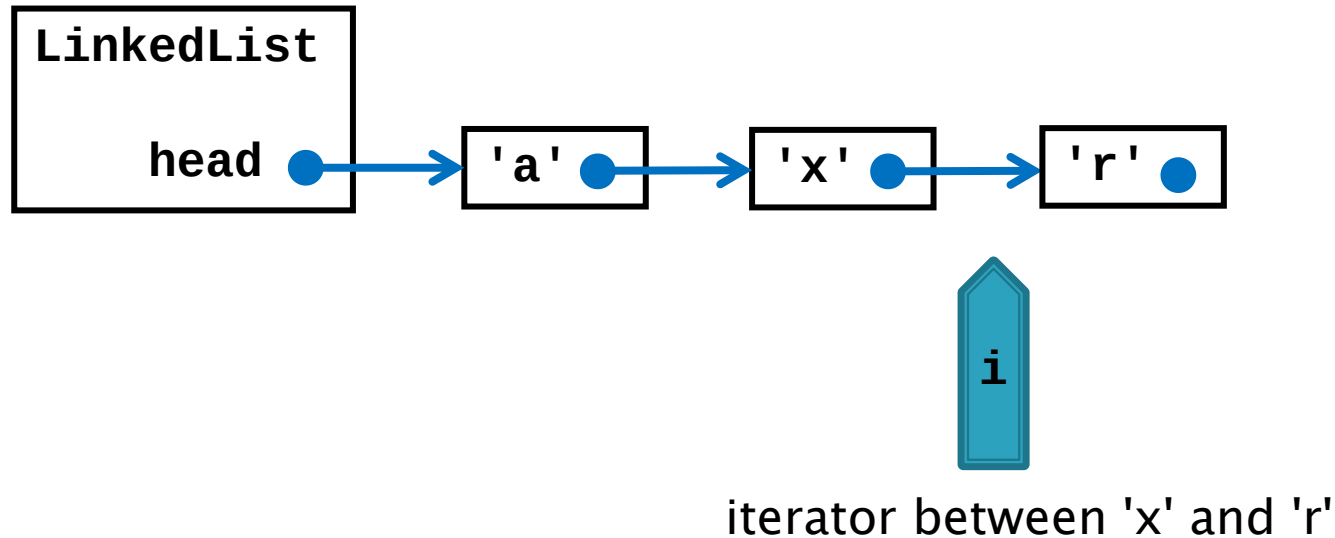
void

remove()

Removes from the underlying collection the last element returned by this iterator (optional operation).

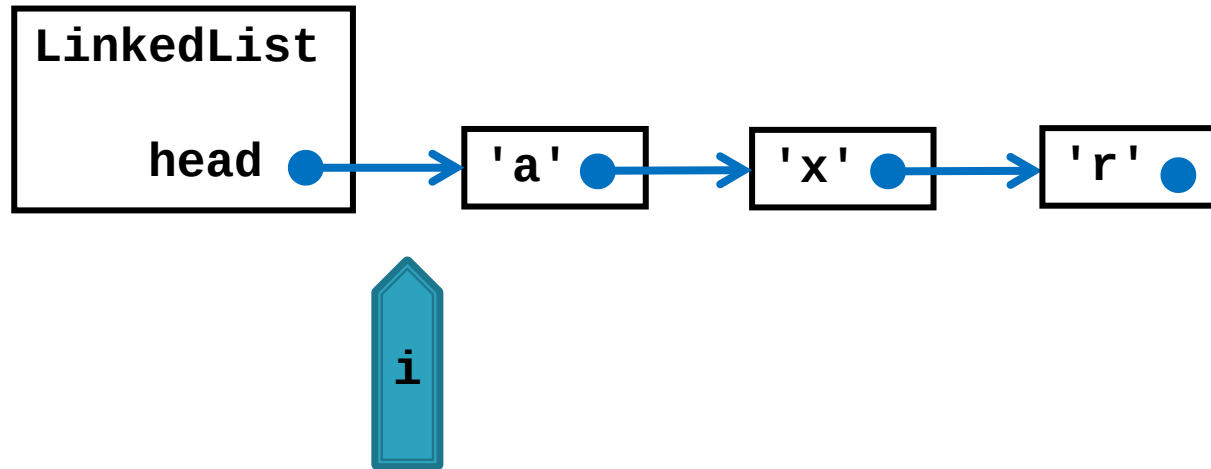
LinkedList Iterator

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



LinkedList Iterator

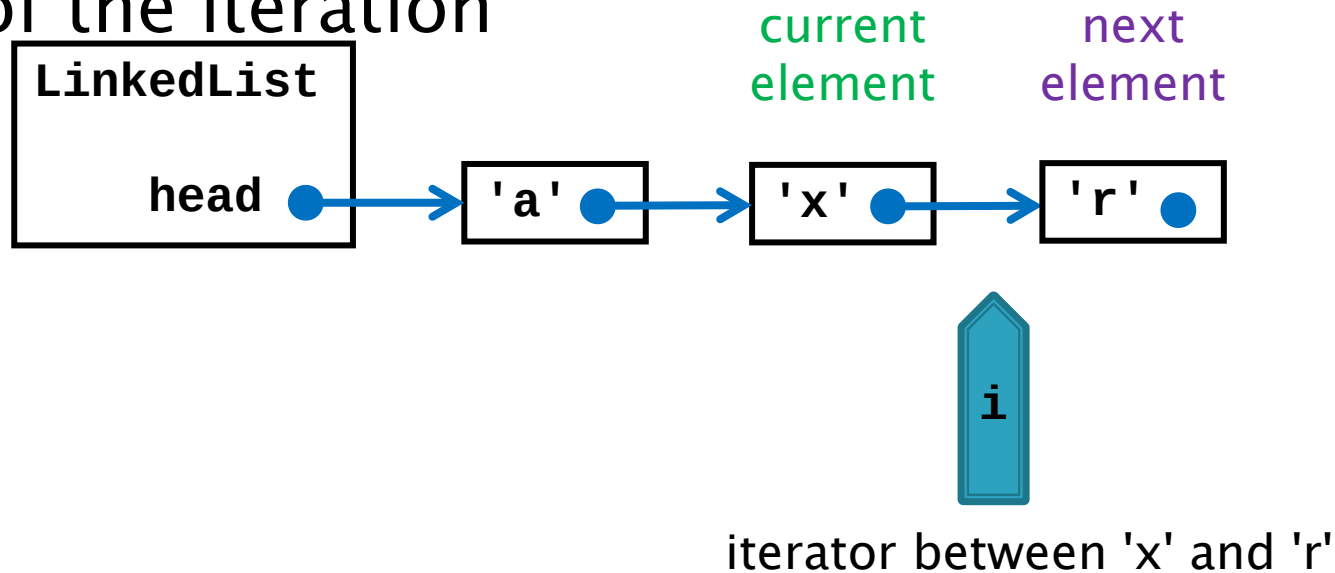
- ▶ Think of the iterator as lying between elements in the list (like a cursor)



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

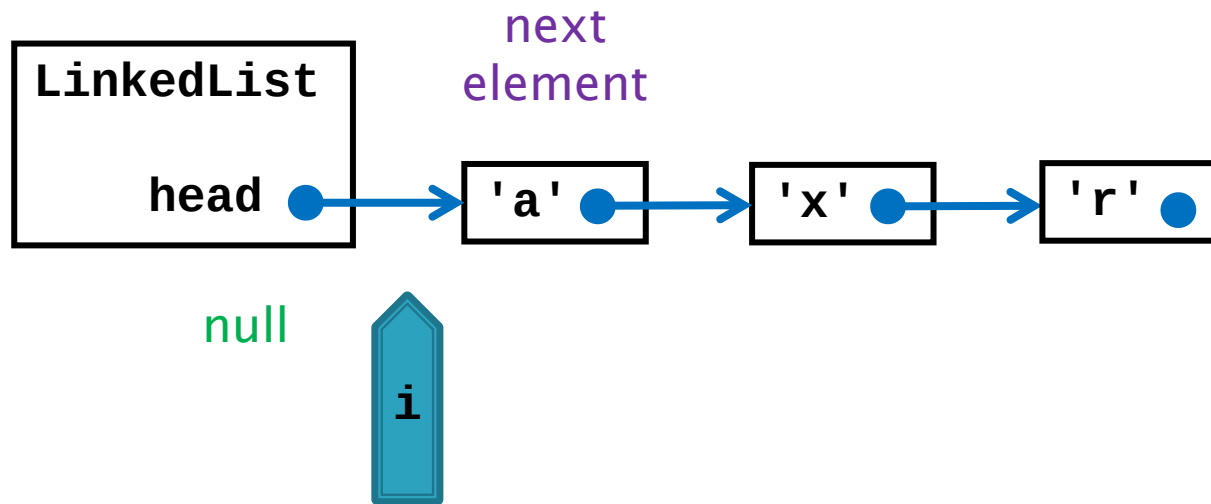
LinkedList Iterator

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



LinkedList Iterator

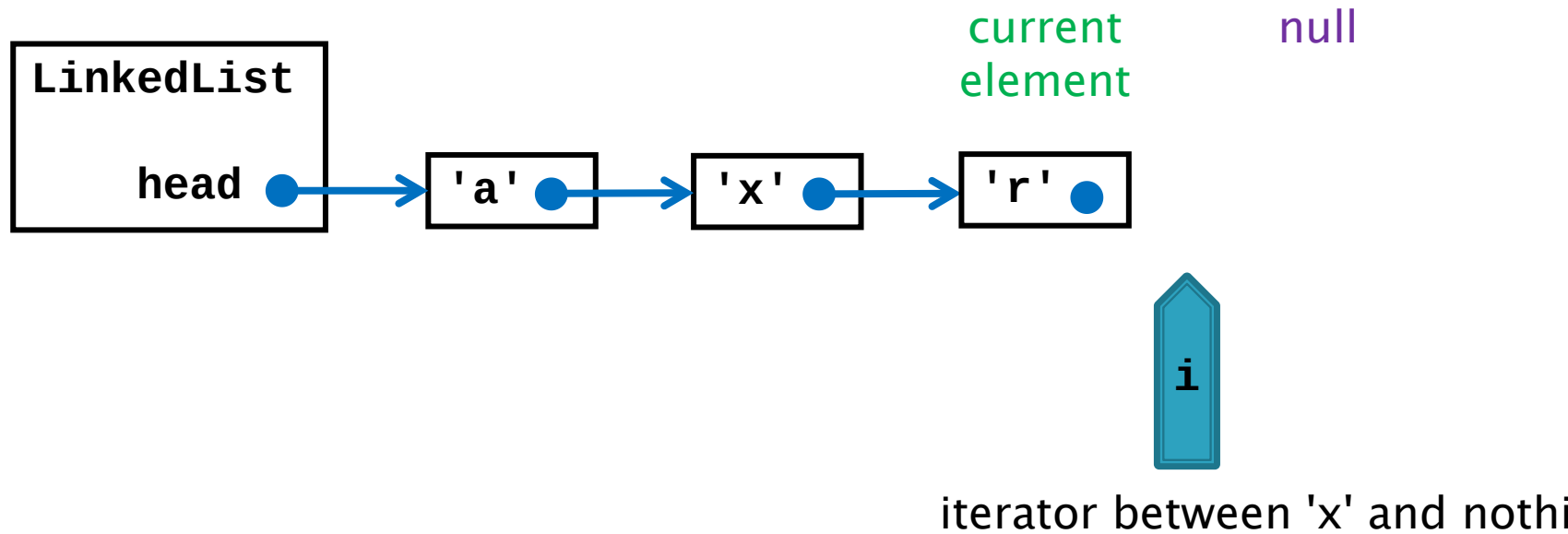
- ▶ The current element is `null` at the start of the iteration



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

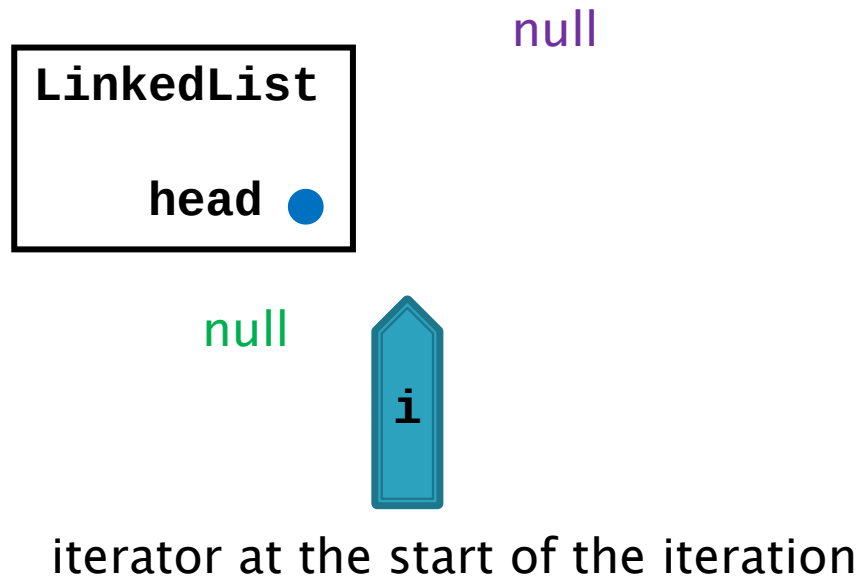
LinkedList Iterator

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



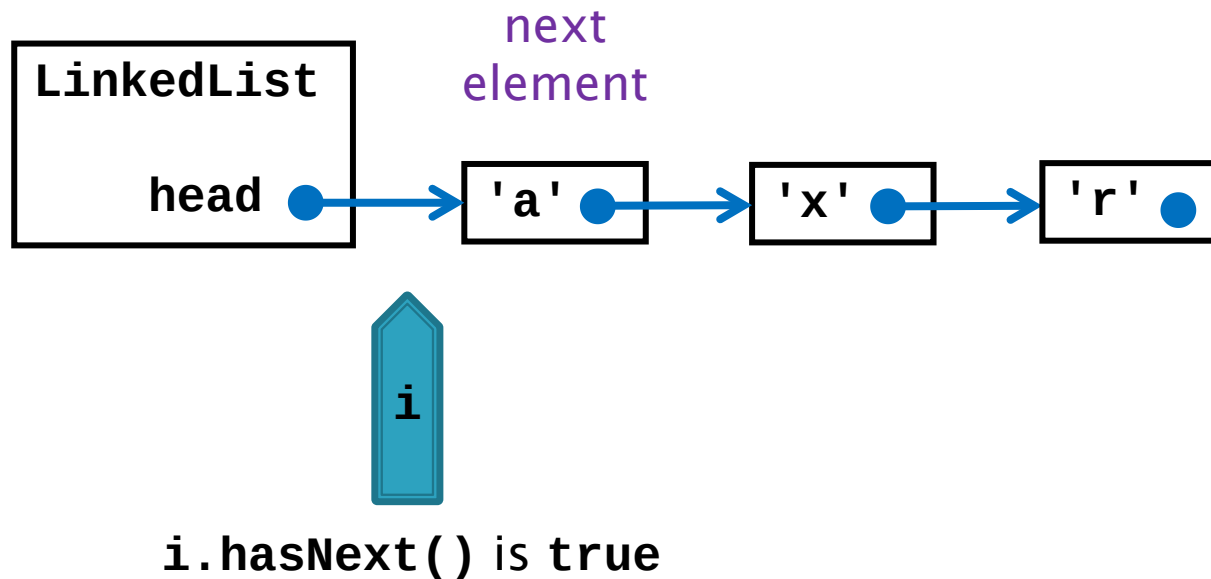
LinkedList Iterator

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



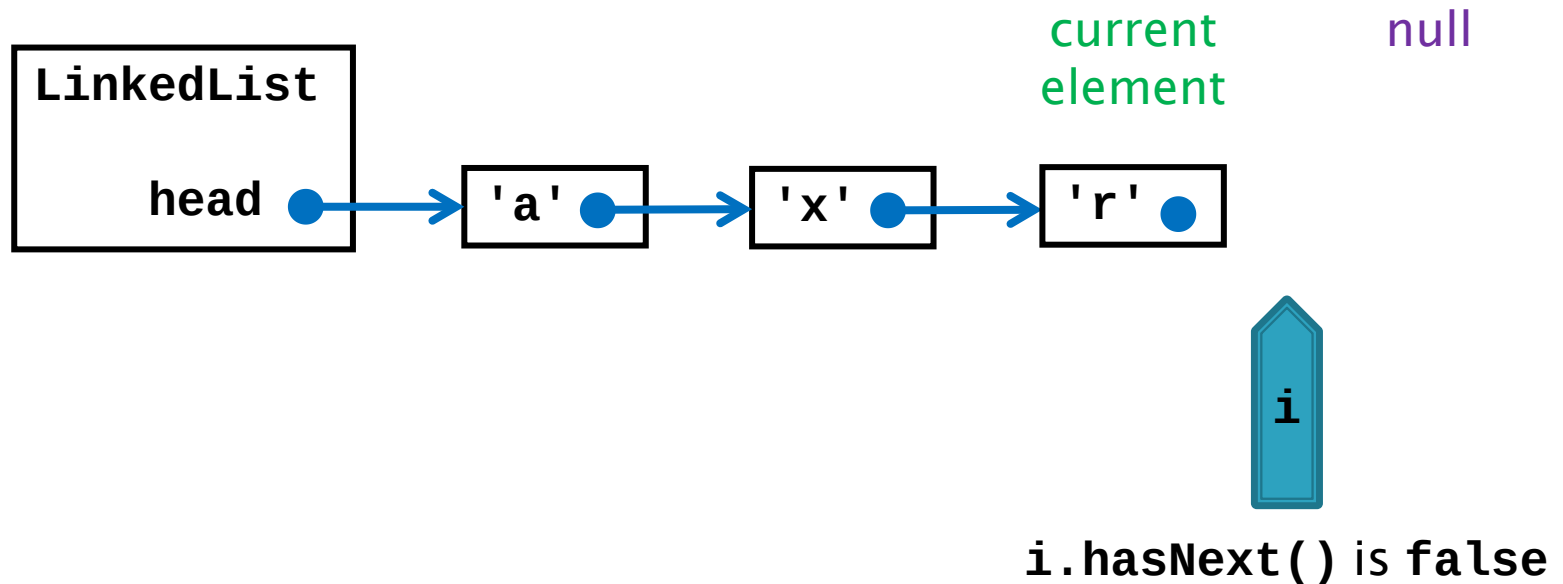
LinkedList Iterator: hasNext

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



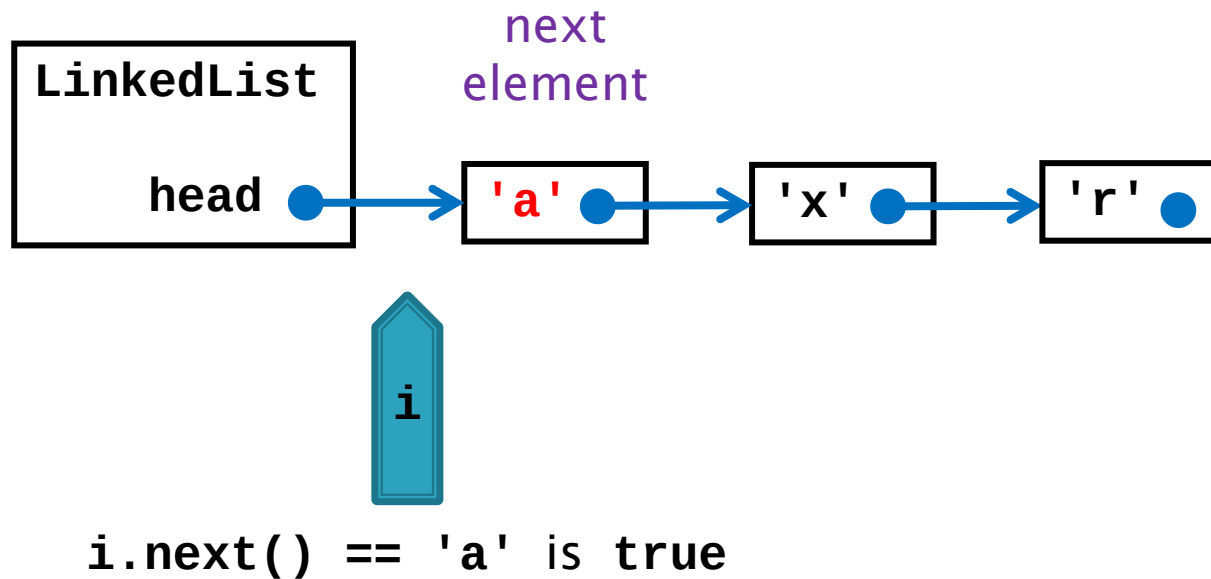
LinkedList Iterator: hasNext

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



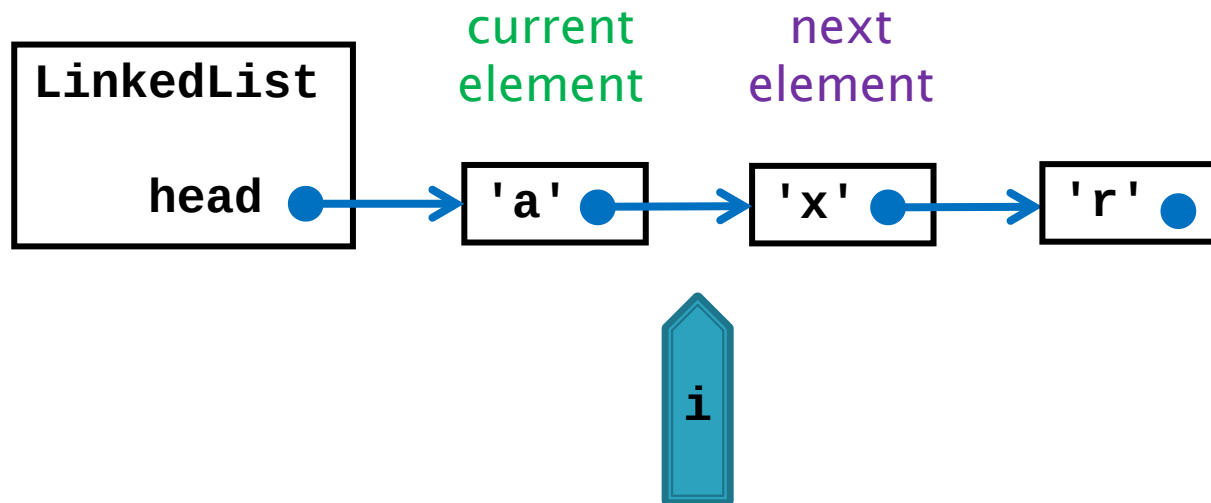
LinkedList Iterator: next

- ▶ Invoking `next()` returns the next element...



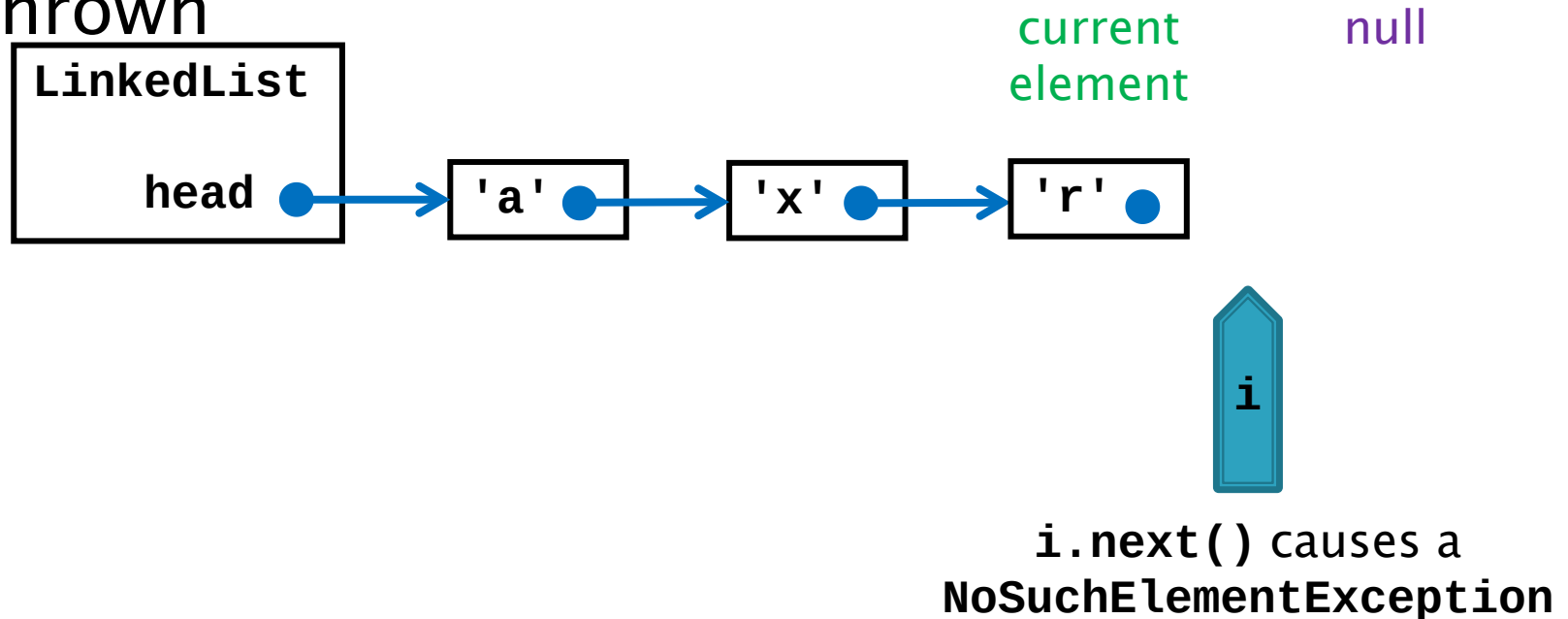
LinkedList Iterator: next

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



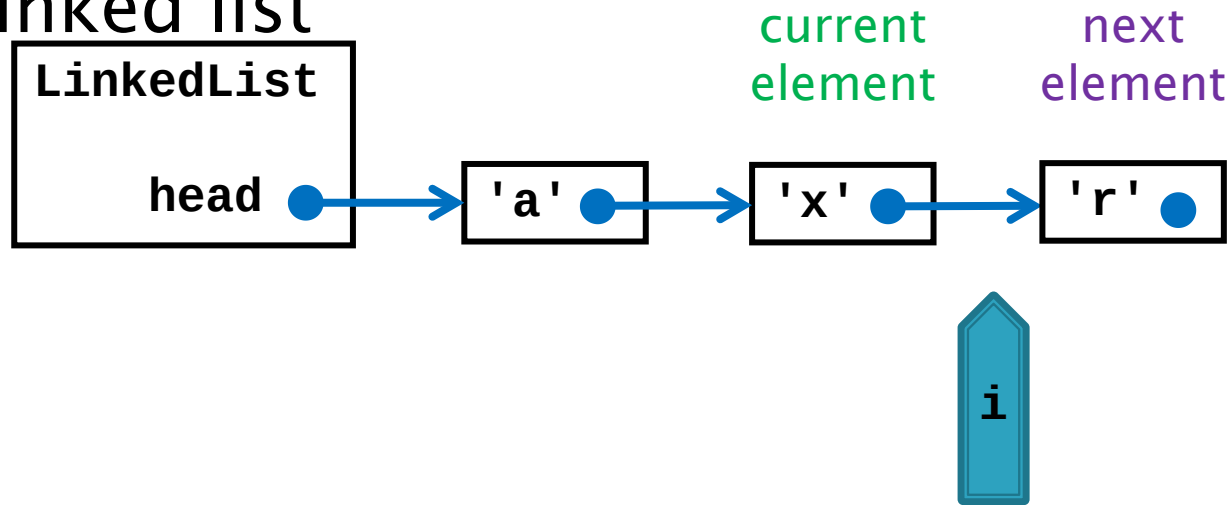
LinkedList Iterator: next

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



LinkedList Iterator: remove

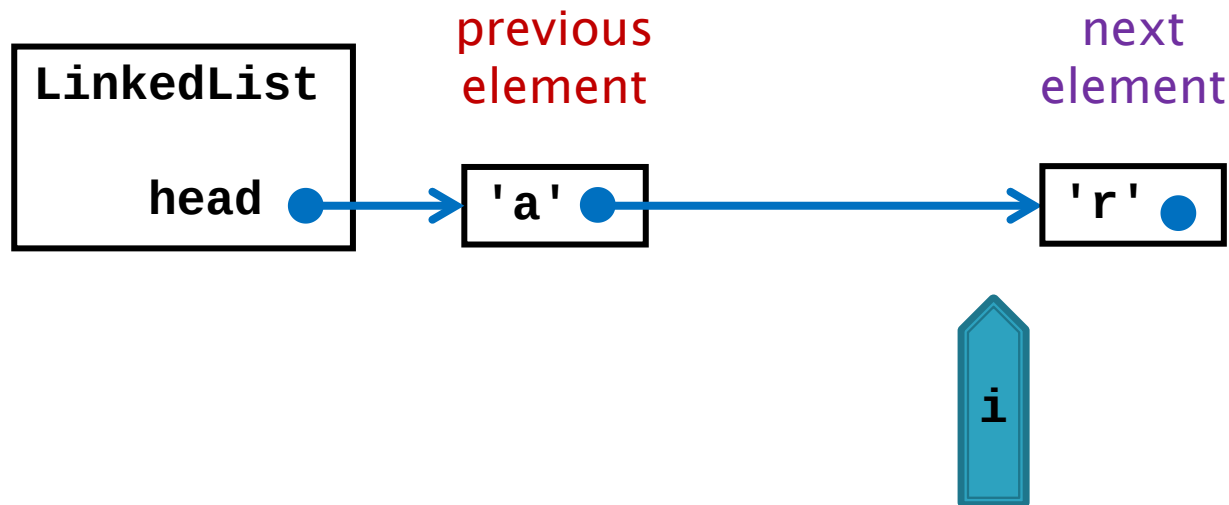
- ▶ `remove()` causes the element most recently returned by `next()` to be removed from the linked list



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

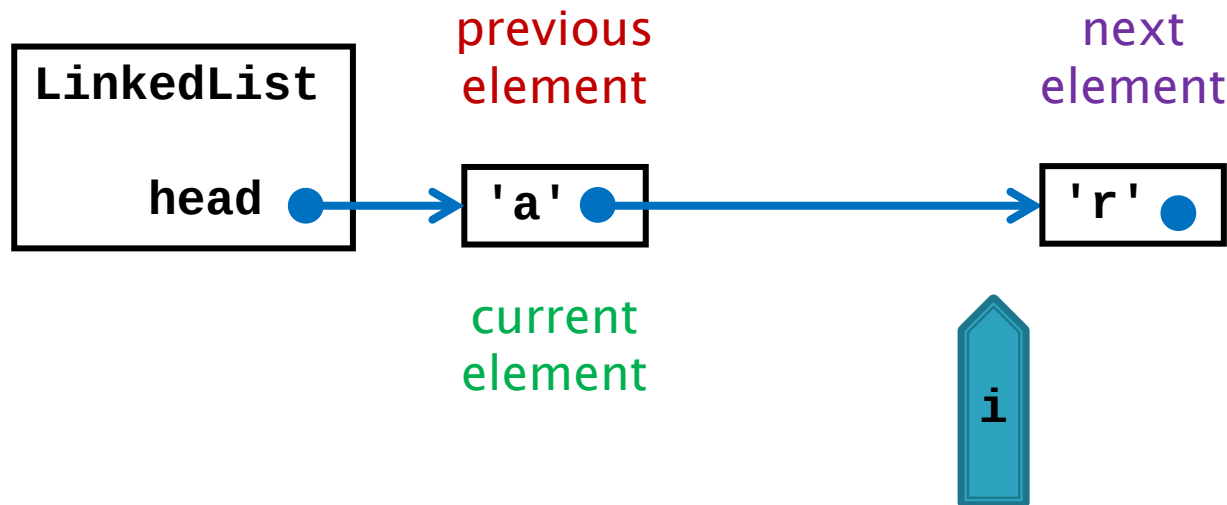
LinkedList Iterator: remove

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



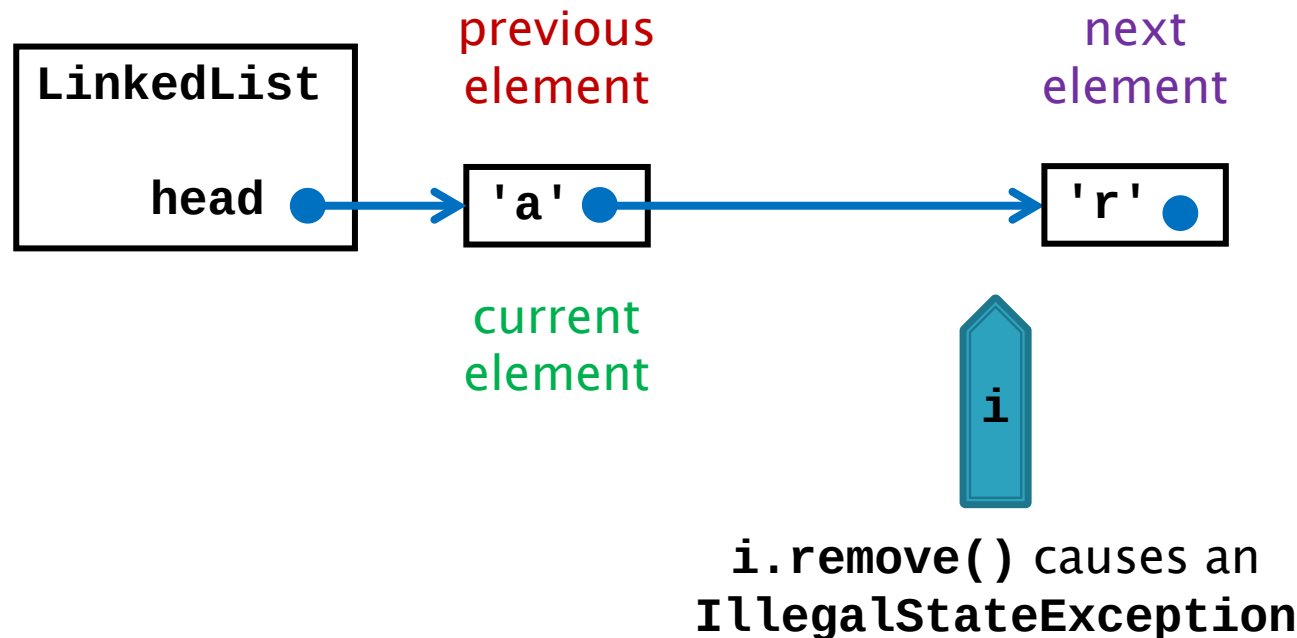
LinkedList Iterator: remove

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



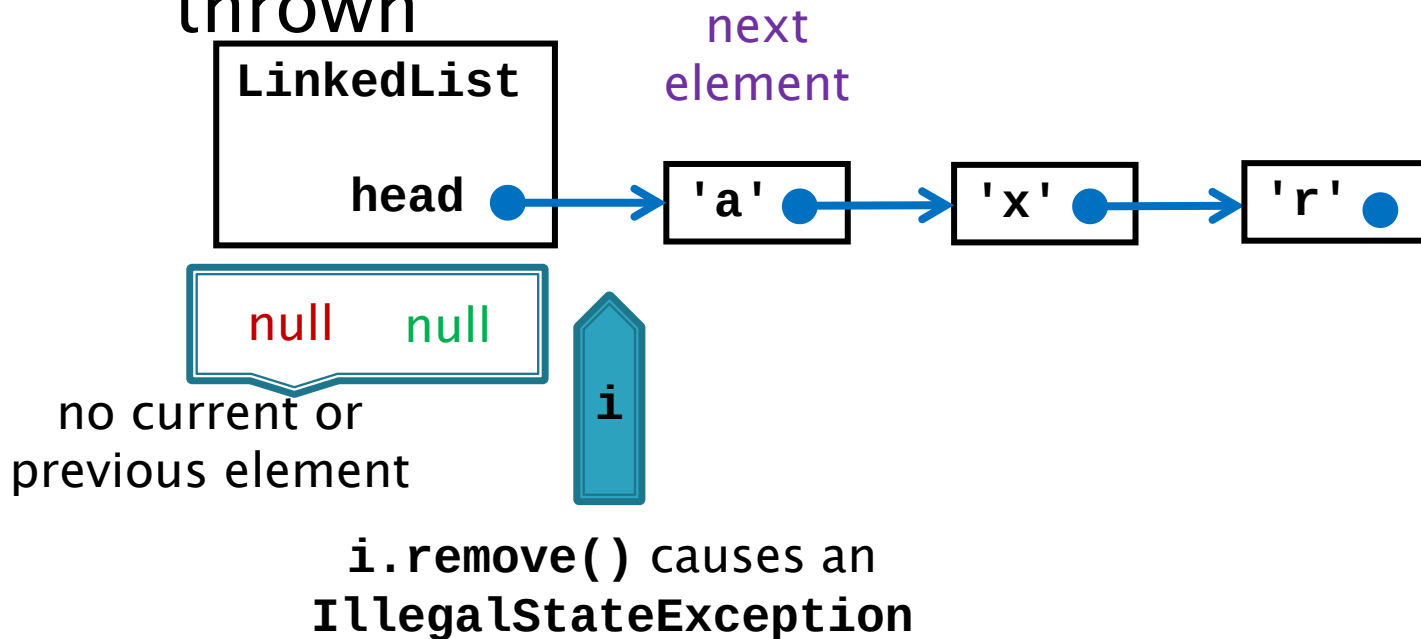
LinkedList Iterator: remove

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



LinkedList Iterator: remove

- ▶ Invoking `remove()` before calling `next()` also causes an `IllegalStateException` to be thrown



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 Constructor

```
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

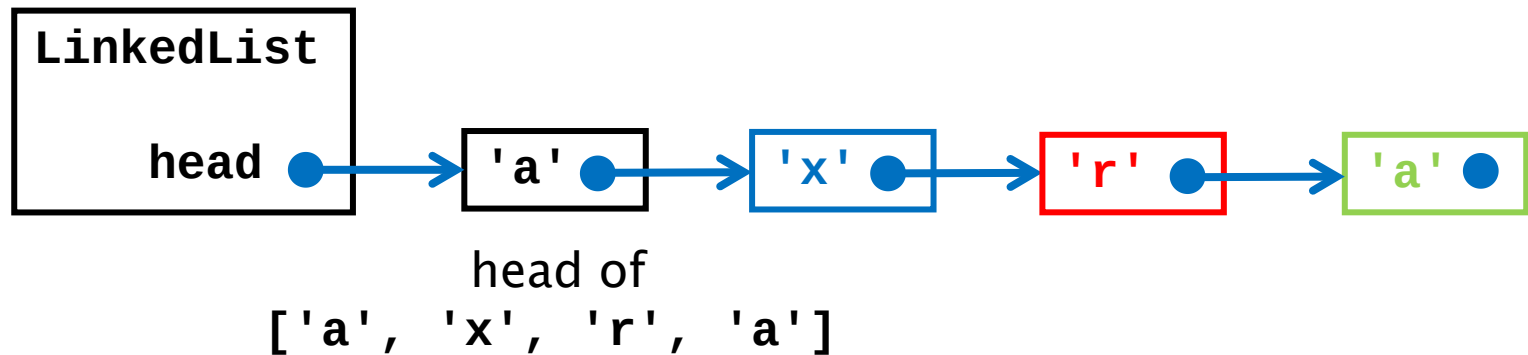
```
@Override
public Character next()
{
    if (!this.hasNext())
    {
        throw new NoSuchElementException();
    }
    this.prevNode = this.currNode;
    if (this.currNode == null)
    {
        this.currNode = head;
    }
    else
    {
        this.currNode = this.currNode.next;
    }
    return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

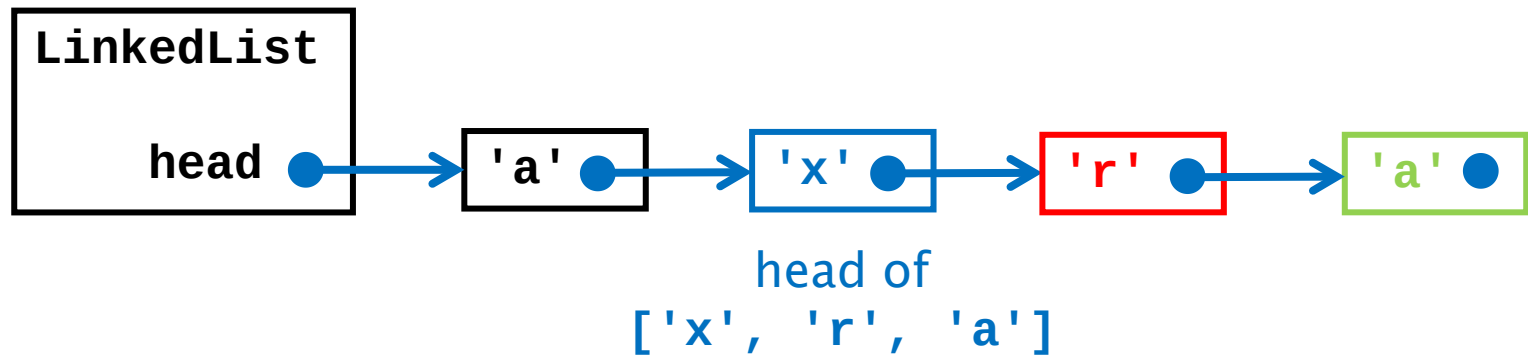
LinkedList Summary

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



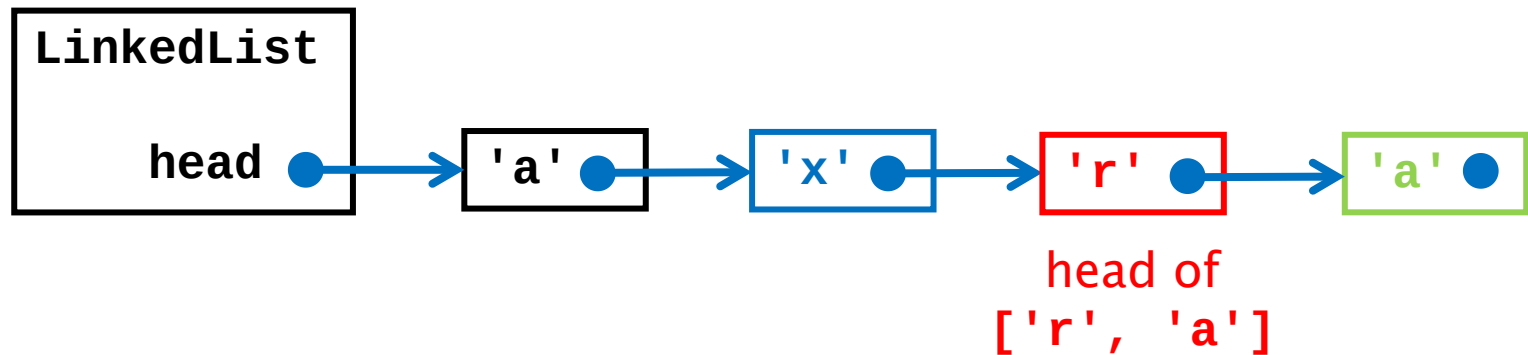
LinkedList Summary

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



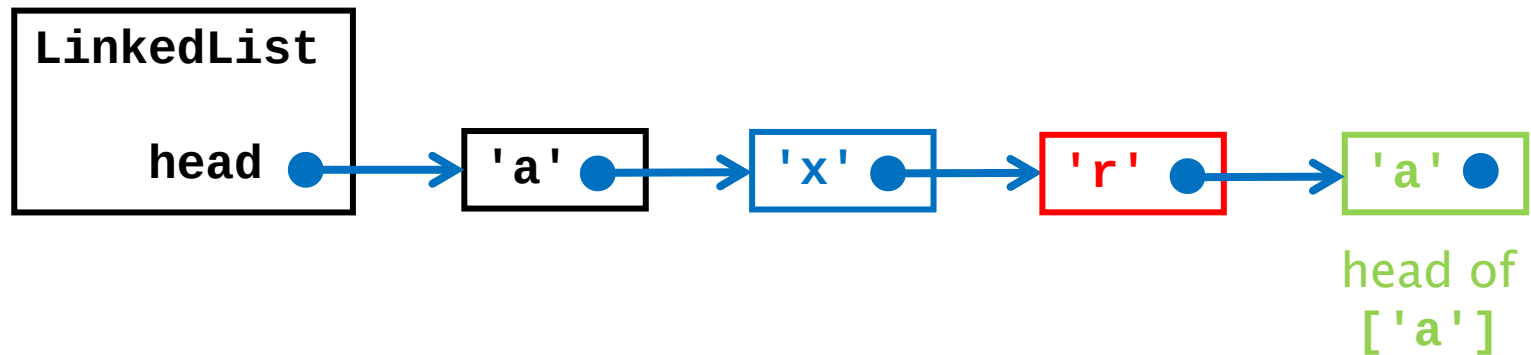
LinkedList Summary

- ▶ 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>

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).
-------------	--