List ADT

Professor Frank Kreimendahl

School of Computing and Data Science

Wentworth Institute of Technology

Sept 9, 2021

List Interface

List Interface

- A **list** data structure holds information and allows specific interactions
- lists are indexed – each element has a specific integer location
- arrays also have this property: lists and arrays *map* integers to elements
- lists support useful operations:
  - Add an element at the end
  - Add and remove an element at a specified position
  - Retrieve an element from a specific position
  - Replace an element at a specific position
  - Find a specific value
- These operations comprise the desired interface for a `List`

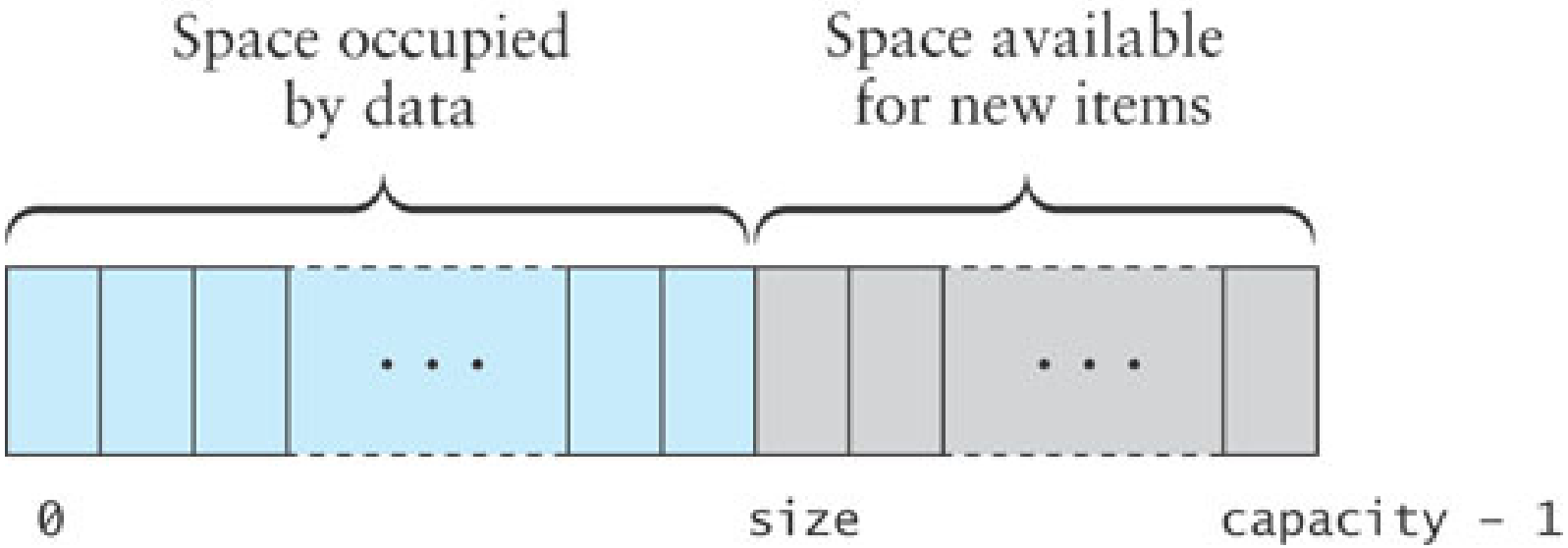
Array Implementation

Properties of Arrays

- arrays allow retrieving and replacing elements
- arrays support scanning to find a value
- arrays do not allow resizing
- arrays do not support adding or removing elements without shifting other elements
- arrays on their own do not have enough features to function as a list

Using Arrays For Lists

- `KWArrayList` class: a simple implementation of a List data structure
  - Named after our textbook authors, Koffman and Wolfgang
- Required information to maintain a list:
  - data field `capacity` to keep track of maximum possible size
  - data field `size` to keep track of current number of entries
  - data field `theData`, which is the array that holds entries



KWArrayList Class and Fields

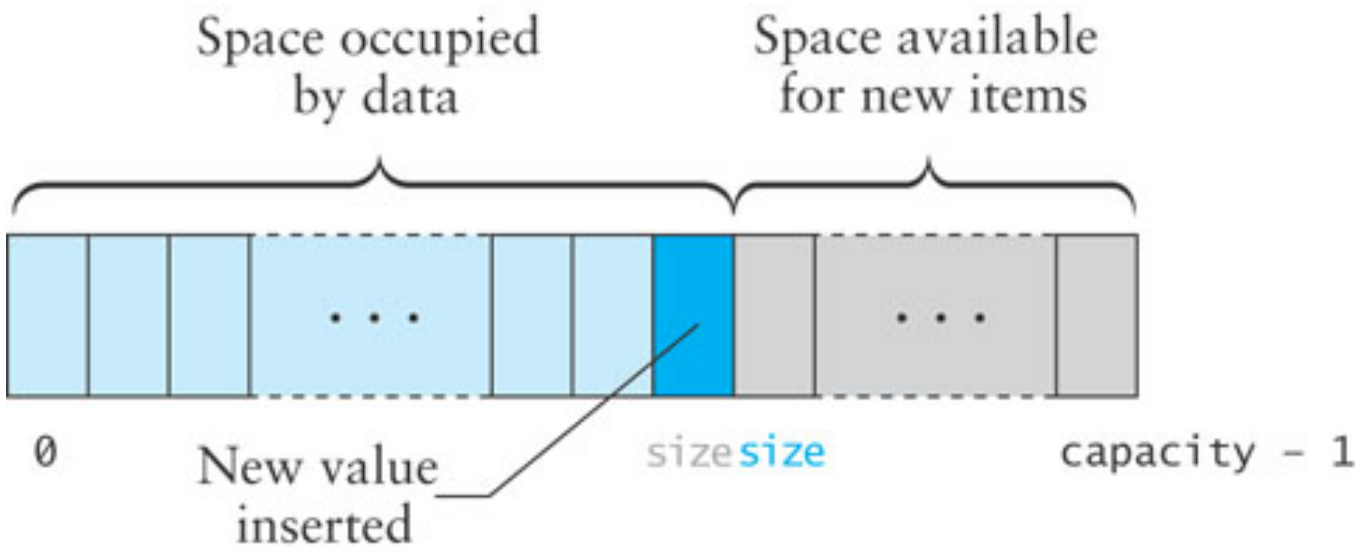
```
import java.util.*;

/** This class implements some of the methods of the
Java ArrayList class */
public class KWArrayList<E> {
  /** The default initial capacity */
  private static final int INITIAL_CAPACITY = 10;

  /** The underlying data array */
  private E[] theData;

  /** The current size */
  private int size = 0;

  /** The current capacity */
  private int capacity = 0;
}
```

KWArrayList Constructor

When someone creates a `KWArrayList` with the `new` keyword, we need to initialize some space and values. Every data structure requires some sort of setup like this.

```
public KWArrayList () {
    capacity = INITIAL_CAPACITY;
    theData = (E[]) new Object[capacity];
}
```

Add Operation: `.add(E anEntry)`

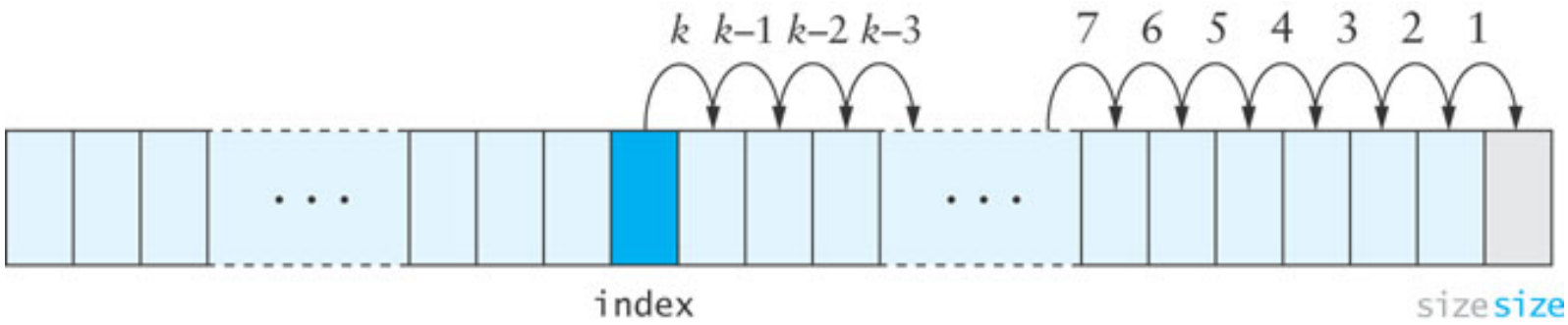


Appending to the end of the list

Consider what fields need to be updated to add the new entry:

- insert new entry at `size`
- increment `size`
- return `true` to indicate success

Add Operation: `.add(int index, E anEntry)`



Inserting at a specific position

Values to the right of the specified index must be shifted to make space, in the order shown.
What could go wrong with these operations?

KWArrayList.add(int, E)

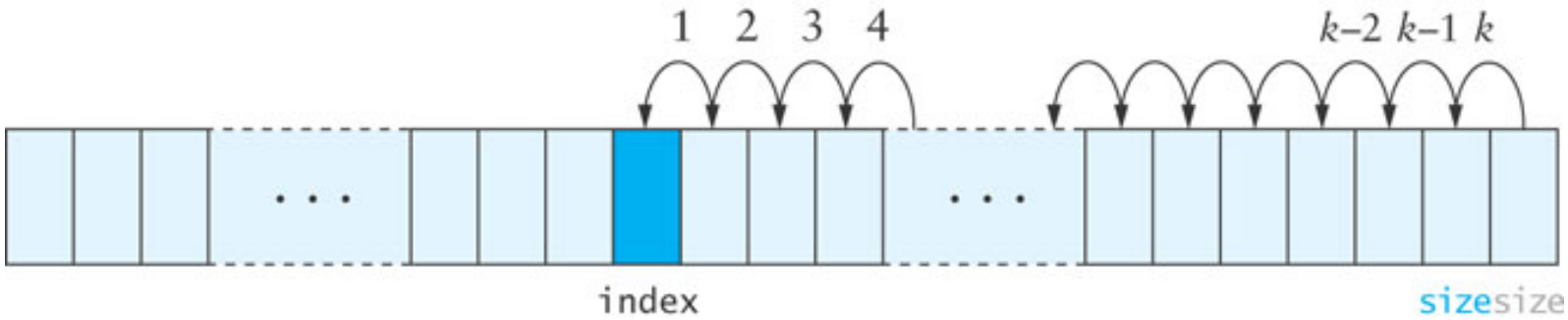
```
public void add (int index, E anEntry) {
  // check bounds
  if (index < 0 || index > size) {
    throw new ArrayIndexOutOfBoundsException(index);
  }
  // Make sure there is room
  if (size >= capacity) {
    reallocate();
  }
  // shift data
  for (int i = size; i > index; i--) {
    theData[i] = theData[i-1];
  }
  // insert item
  theData[index] = anEntry;
  size++;
}
```

.get and .set

```
// retrieve an entry from a specific location
public E get (int index) {
  if (index < 0 || index >= size) {
    throw new ArrayIndexOutOfBoundsException(index);
  }
  return theData[index];
}
```

```
// replace a value at a location and return old value
public E set (int index, E newValue) {
  if (index < 0 || index >= size) {
    throw new ArrayIndexOutOfBoundsException(index);
  }
  E oldValue = theData[index];
  theData[index] = newValue;
  return oldValue;
}
```

Remove Operation: `.remove(int index)`



Removing from a specific position

The gap from a removed entry must be filled by shifting other entries

.remove(int index)

```
// remove a value from a location, fill the gap,
// and return removed value
public E remove (int index) {
  if (index < 0 || index >= size) {
    throw new ArrayIndexOutOfBoundsException(index);
  }

  E returnValue = theData[index];

  for (int i = index + 1; i < size; i++) {
    theData[i-1] = theData[i];
  }

  size--;
  return returnValue;
}
```

`private` Reallocate Operation

We can't resize an array, but we can create a larger array and copy current values to the new one

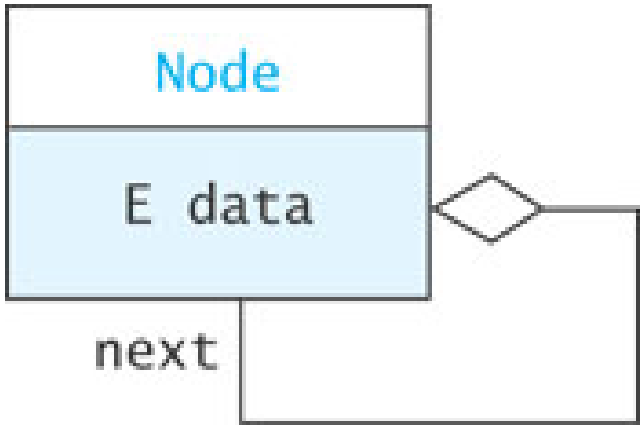
This method is marked as `private` and is only called from the `.add` method when the current array is full

```
private void reallocate () {
  capacity *= 2;
  theData = Arrays.copyOf(theData, capacity);
}
```

Nodes

Node Description

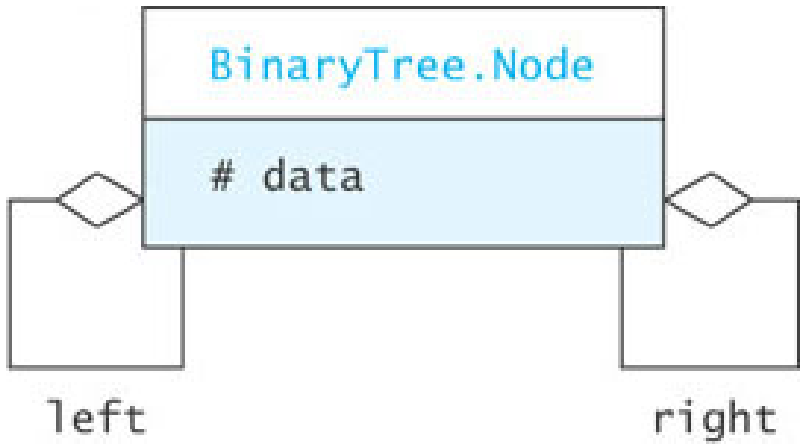
- Nodes are useful building blocks for many data structure implementations
- A Node class contains:
  - One data entry
  - At least one link to a node
- A link is a reference to a node



List Node with `data` and `next` Node link

Nodes for Other Structures

In future data structures, we will see Nodes hold different information to support the necessary structure, such as:



Binary Tree Node with `data` and two child Nodes

Single-Linked List

Single-Linked Lists

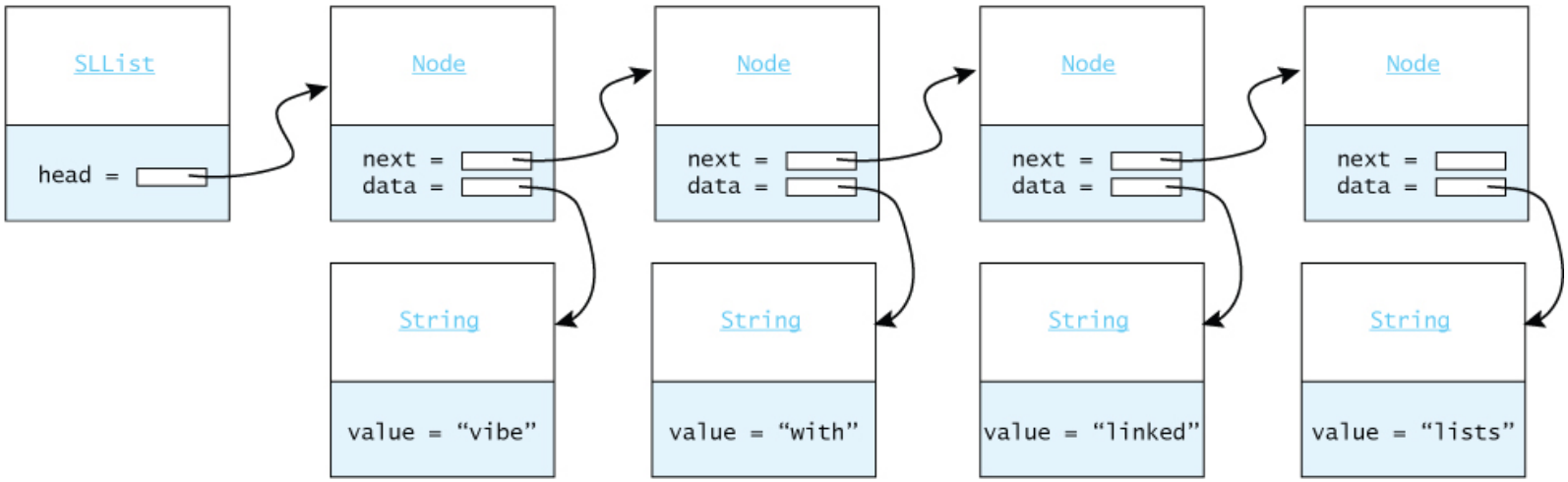
- A chain of Nodes can be used to implement all of the operations of a list interface
- This structure has different strengths and weaknesses than storing entries in an array
  - Arrays can retrieve values with less work
  - Linked Lists can add/remove values from arbitrary locations with less work

List Node Inner Class

```
private static class Node<E> {
  private E data;
  private Node<E> next;

  // Creates a new node with a null next field
  private Node(E dataItem) {
    data = dataItem;
    next = null;
  }

 // Creates a new node that references another node
  private Node(E dataItem, Node<E> nodeRef) {
    data = dataItem;
    next = nodeRef;
  }
}
```

List of Nodes

- `SLList` only links to one Node in the list – called the `head`
- Each Node connects to the next Node
- The end of the list is marked with a Node that has a null `next` value



A `SLList` of `String`

SLList

The beginning of a SLList class:

```
public class SLList<E> {
  private Node<E> head = null;
  private int size = 0;

  // list methods inserted here
}
```

Considering Operations with Nodes

Now that we have decided on our basic building block, a `Node`, we must consider the operations to build a functional list:

- To modify a list, we will have to reassign at least one `next` value in the list
- To query a list, we will have to traverse the list – go through each Node in succession
- When designing a data structure, we must consider all desired operations
- We must also consider edge cases – often a special case if a data structure is empty or full

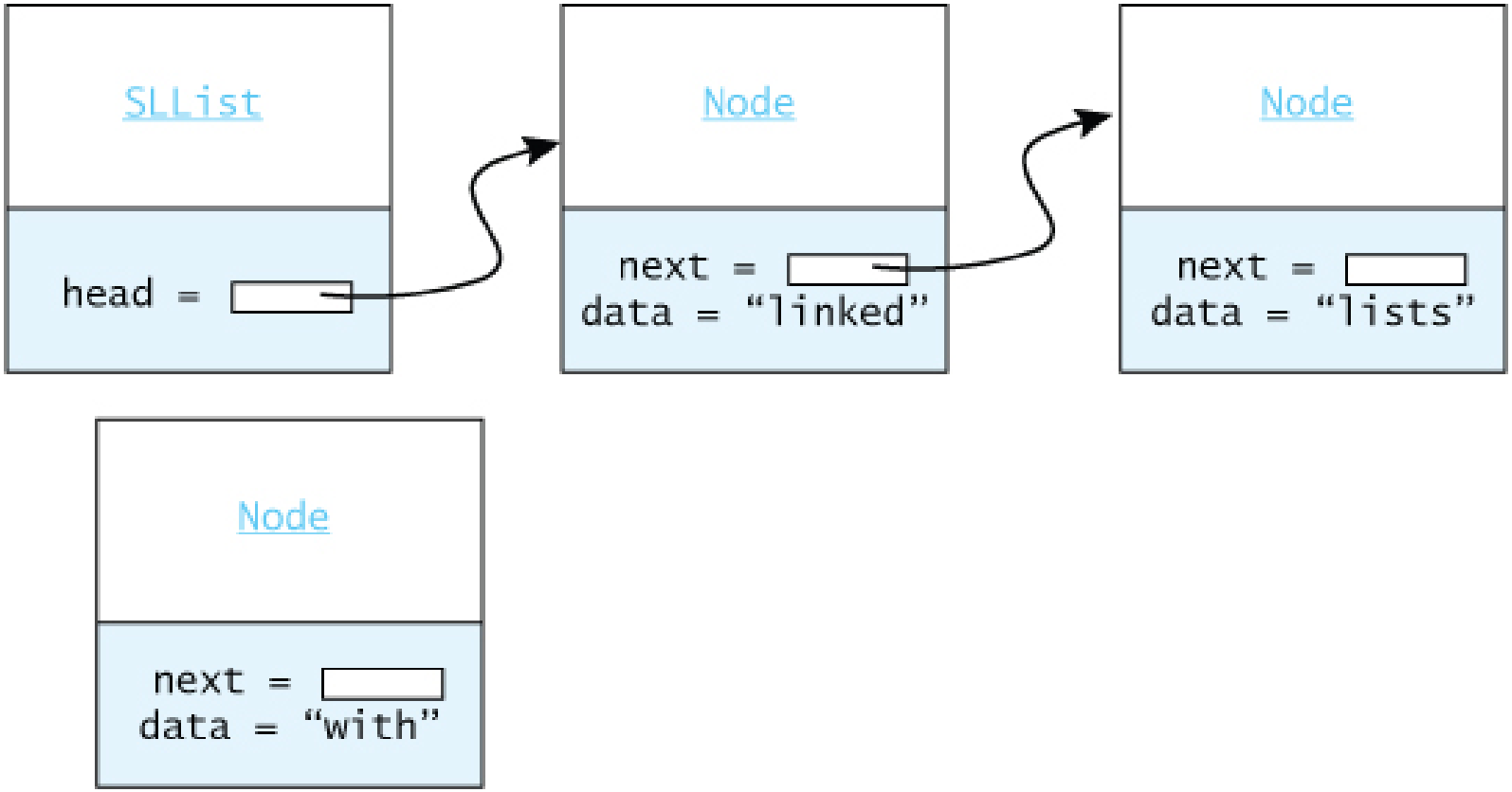
SLList Private Methods

In order to implement the basic List operations, we will build private methods to help implement the public ones:

- `addFirst(E item)`
- `addAfter(Node node, E item)`
- `removeFirst()`
- `removeAfter(Node node)`
- `getNode(int)`

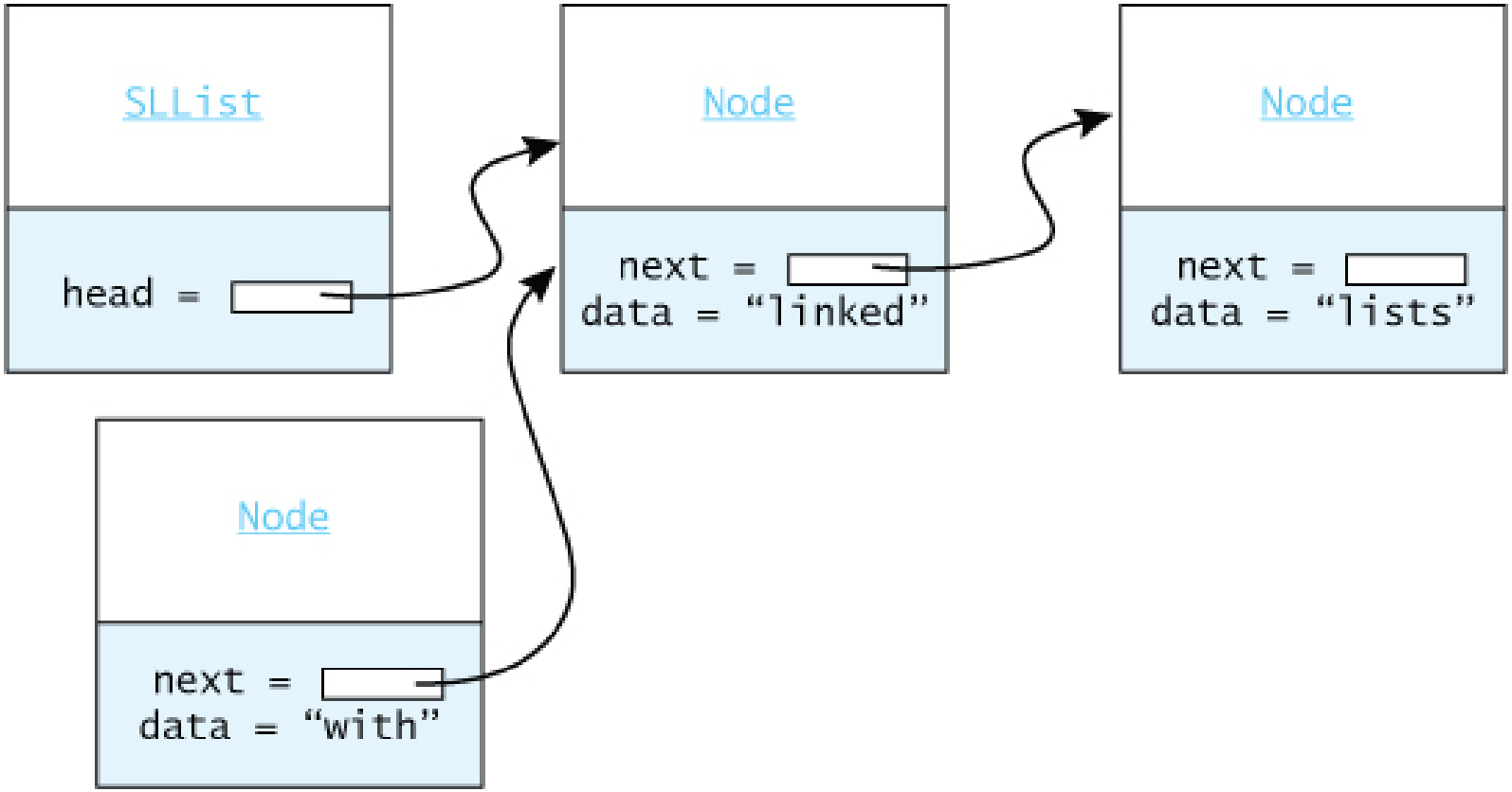
We will use these private methods to build `get`, `set`, `add`, `remove`, `indexOf`, and `size`

First step of `.addFirst("with")`



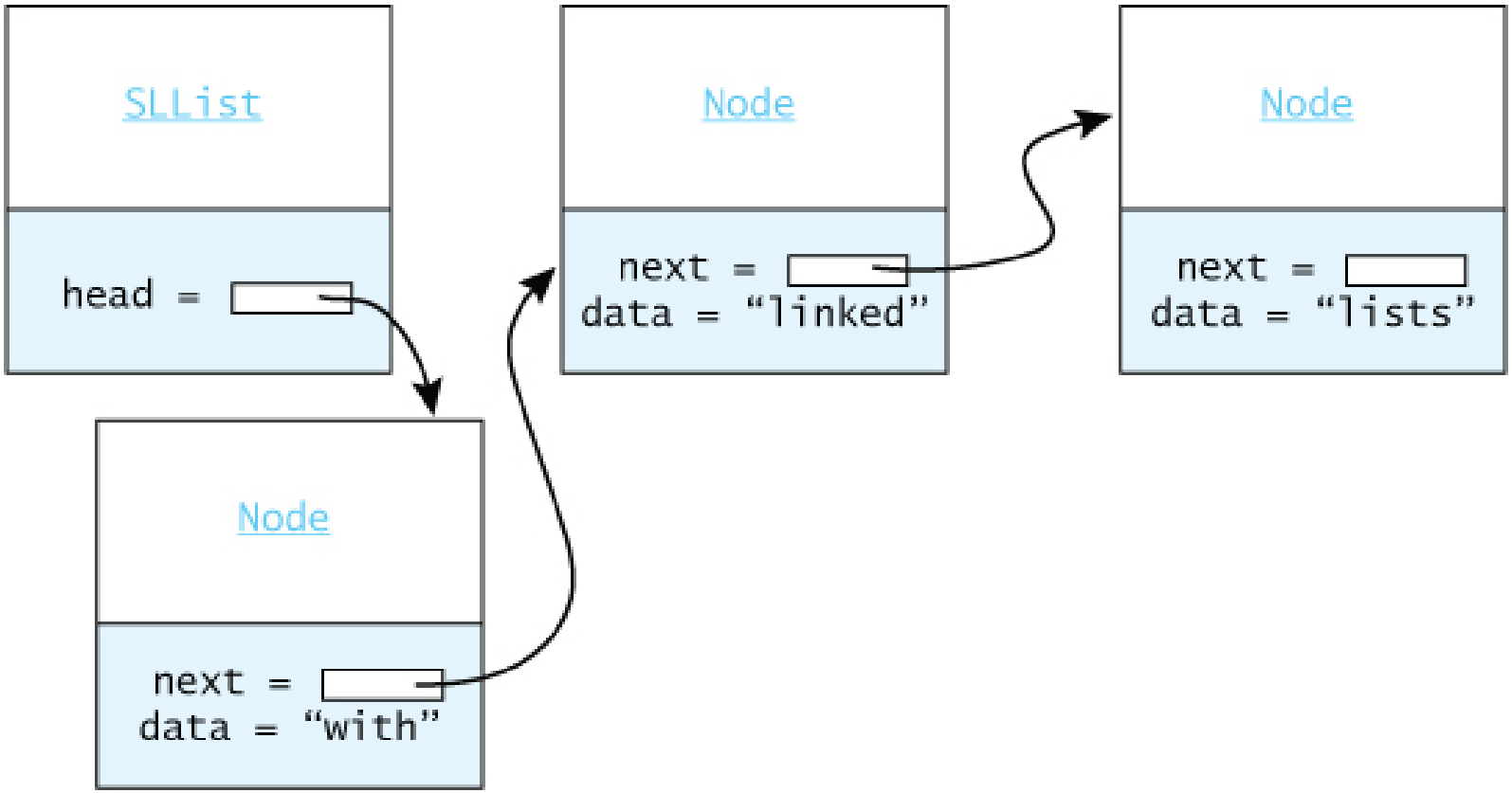
Create a new `Node` to hold the data we want to store

Second step of `.addFirst("with")`



Copy the current `head` reference to `next` of new Node

Third step of `.addFirst("with")`



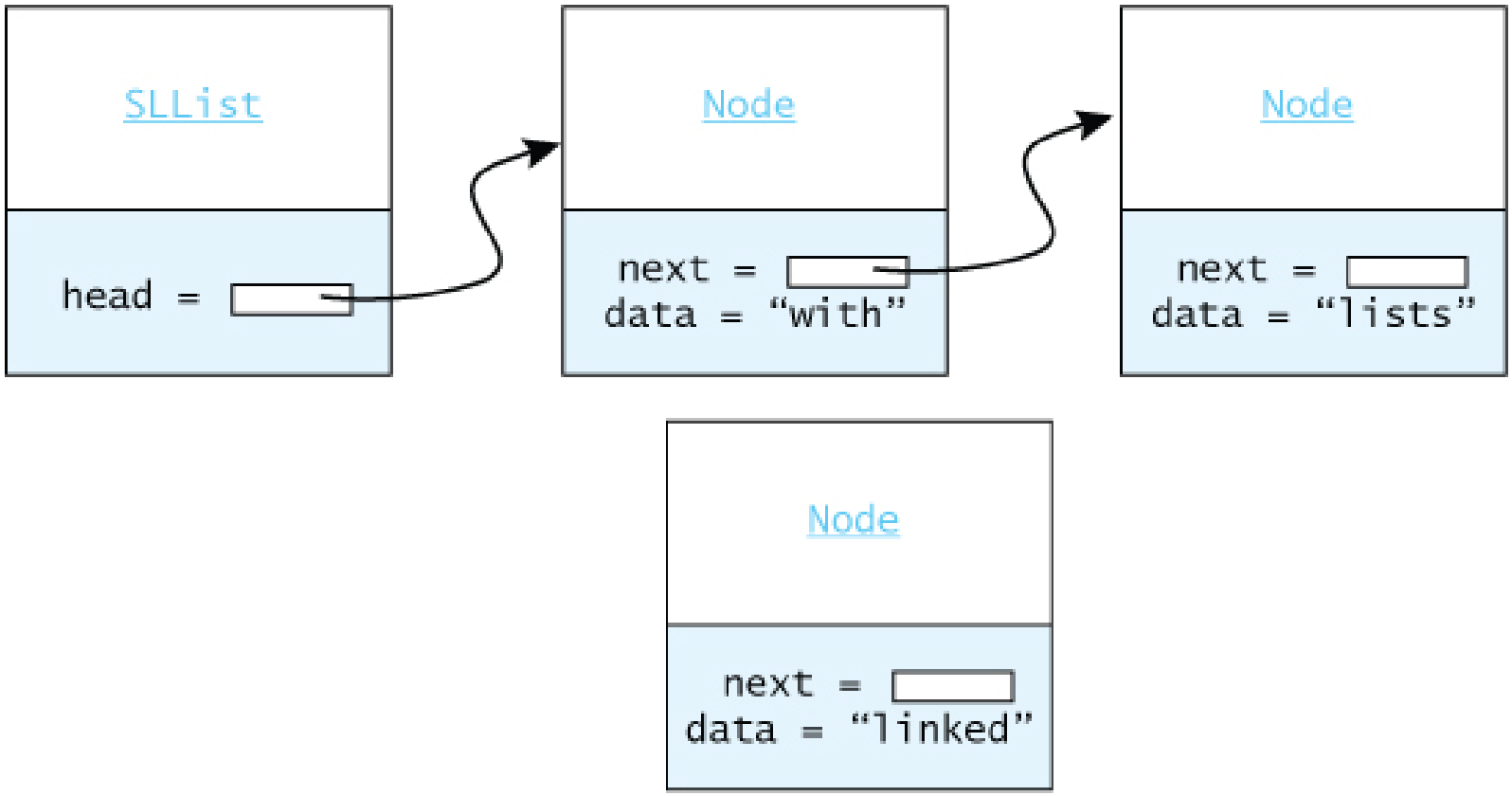
Change `head` to reference new Node

addFirst() Implementation

```
private void addFirst (E item) {
  Node<E> temp = new Node<E>(item, head);
  head = temp;
  size++;
}
```

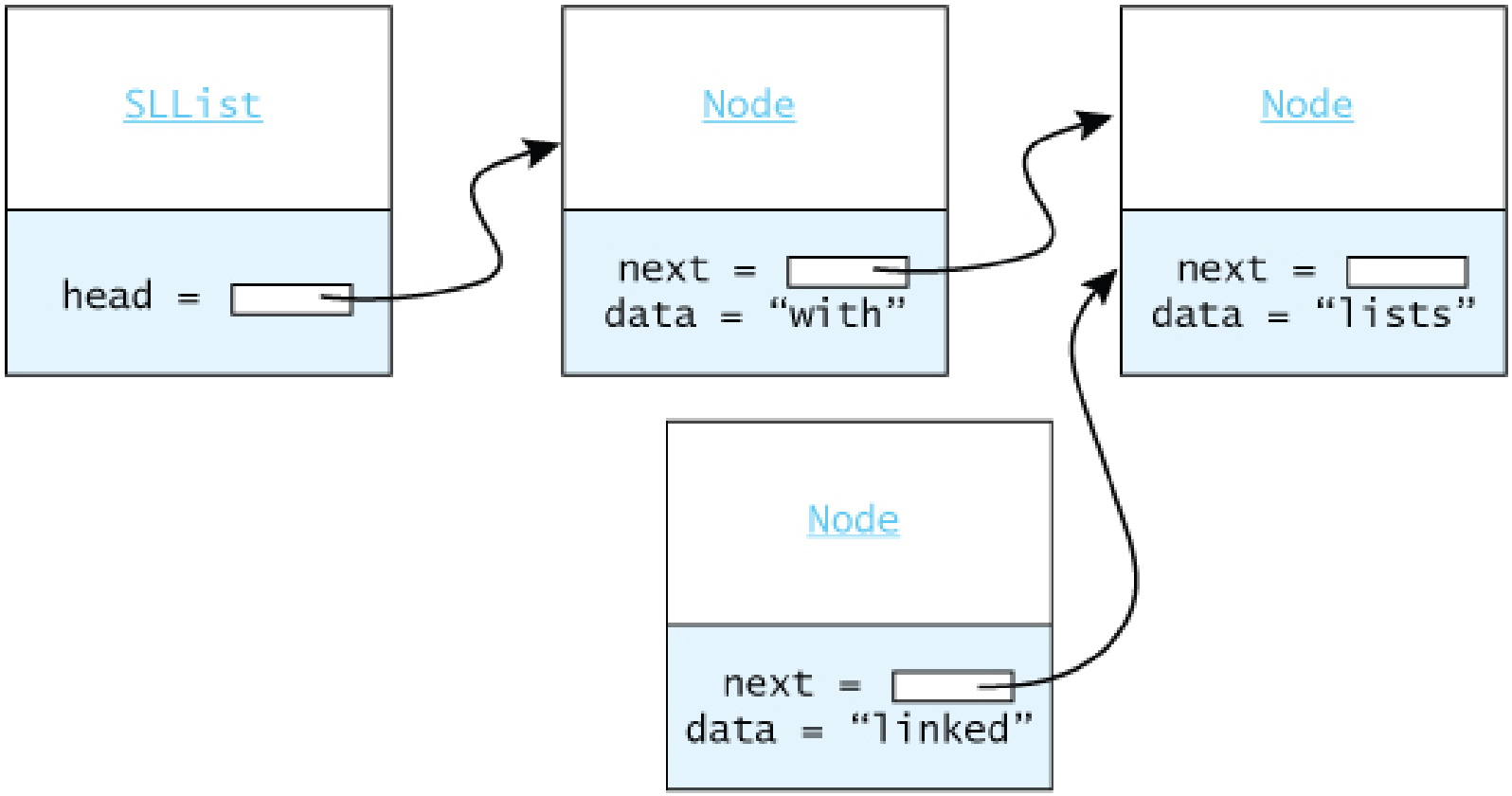
Two cases to consider: list is empty at start, or list is not empty

First step of `.addAfter(withN, "linked")`



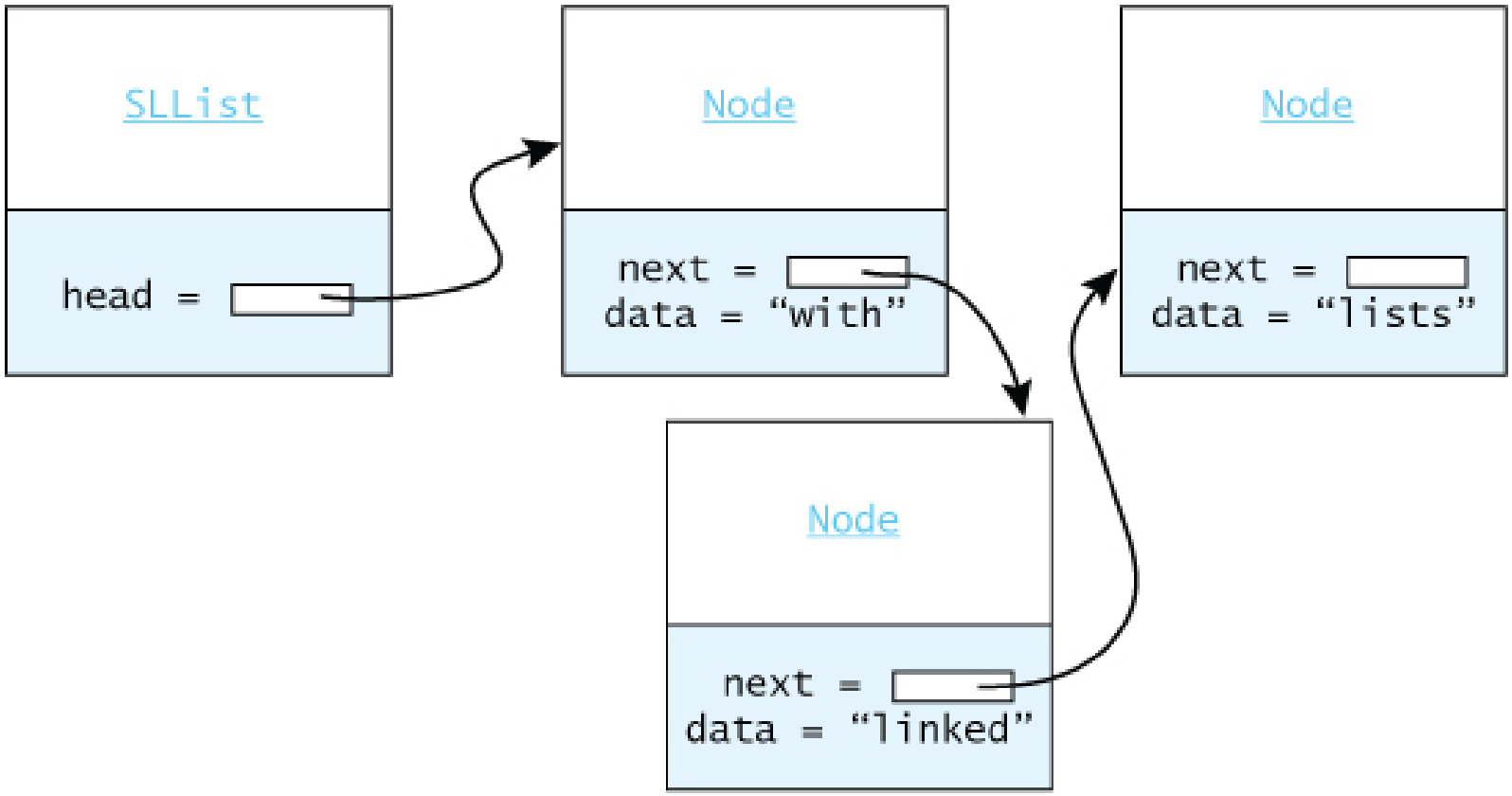
Create a new `Node` to hold the data we want to store

Second step of `.addAfter(withN, "linked")`



Copy the current `withN.next` reference to `next` of new Node

Third step of `.addAfter(withN, "linked")`



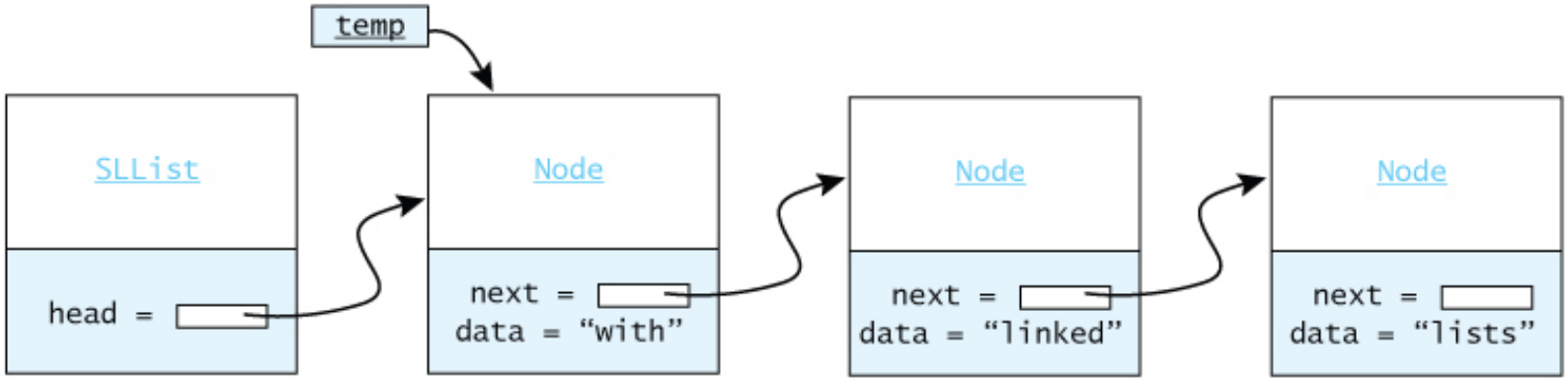
Change `withN.next` to reference new Node

`addAfter()` Implementation

```
private void addAfter (Node<E> node, E item) {
  Node<E> temp = new Node<E>(item, node.next);
  node.next = temp;
  size++;
}
```

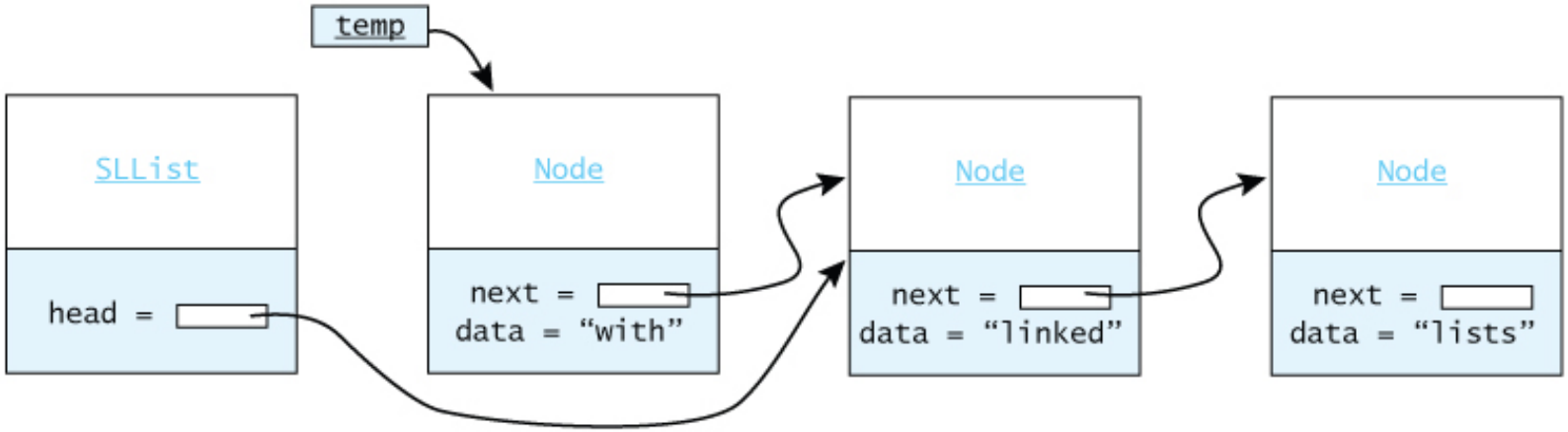
Two cases to consider: list is empty after `node`, or list is not empty

First step of `.removeFirst()`



Create a `temp` reference to the `Node` that will be removed

Second step of `.removeFirst()`



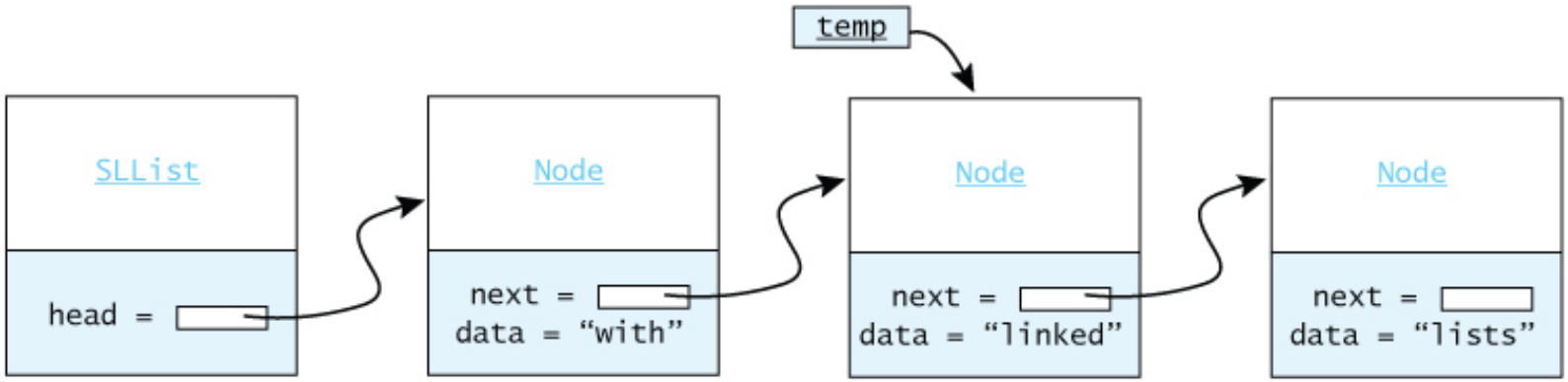
Change `head` to skip first Node

removeFirst() Implementation

```
private E removeFirst () {
  Node<E> temp = head;
  if (head != null) {
    head = head.next;
  }
  if (temp != null) {
    size--;
    return temp.data;
  } else {
    return null;
  }
}
```

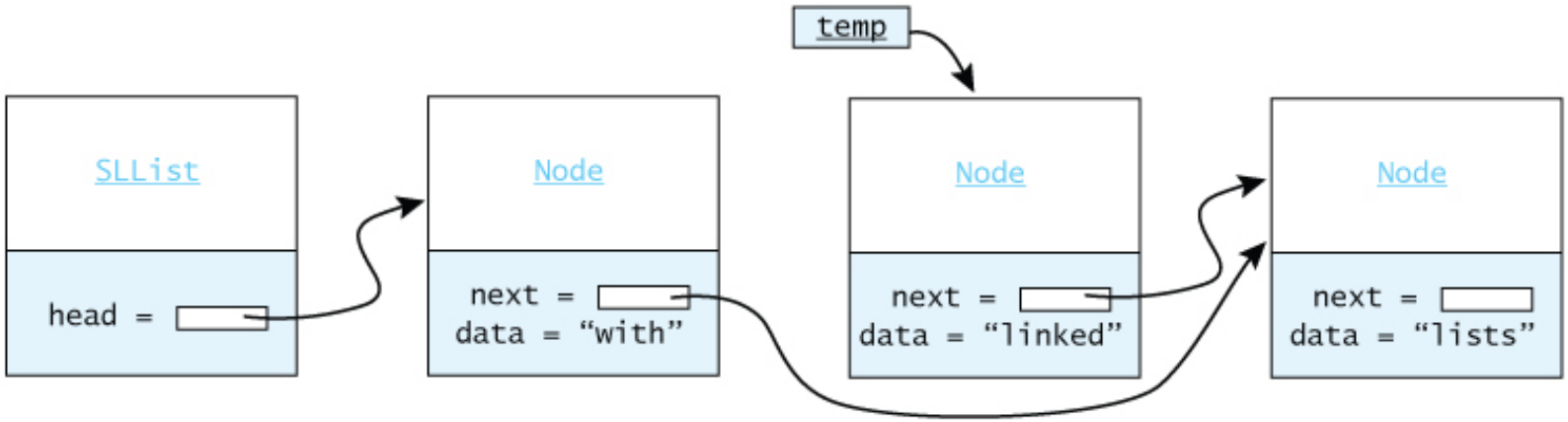
Two cases to consider: list is empty at start, or list is not empty

First step of `.removeAfter(withN)`



Create a `temp` reference to the `Node` that will be removed

Second step of `.removeAfter(withN)`



Change `withN.next` to skip next Node

`removeAfter()` Implementation

```
private E removeAfter (Node<E> node) {
  Node<E> temp = node.next;
  if (temp != null) {
    node.next = temp.next;
    size--;
    return temp.data;
  } else {
    return null;
  }
}
```

Two cases to consider: list is empty after `node`, or list is not empty

`getNode()` Implementation

One last utility to map an index to a Node in the list:

```
private Node<E> getNode(int index) {
  Node<E> node = head;
  for (int i=0; i<index && node != null; i++) {
    node = node.next;
  }
  return node;
}
```

With this, we can build public methods in a straightforward manner.

get() Implementation

```
public E get (int index) {
  if (index < 0 || index >= size) {
    String s = Integer.toString(index);
    throw new IndexOutOfBoundsException(s);
  }

  Node<E> node = getNode(index);
  return node.data;
}
```

`set()` Implementation


```
public E set (int index, E anEntry) {
  if (index < 0 || index >= size) {
    String s = Integer.toString(index);
    throw new IndexOutOfBoundsException(s);
  }

  Node<E> node = getNode(index);
  E result = node.data;
  node.data = anEntry;
  return result;
}
```

`add(int, E)` Implementation


```
public void add (int index, E item) {
  if (index < 0 || index > size) {
    String s = Integer.toString(index);
    throw new IndexOutOfBoundsException(s);
  }

  if (index == 0) {
    addFirst(item);
  } else {
    Node<E> node = getNode(index-1);
    addAfter(node, item);
  }
}
```

`add(E)` Implementation


```
// adds an item to the end of the list
public boolean add (E item) {
  add(size, item);
  return true;
}
```

Double-Linked Lists

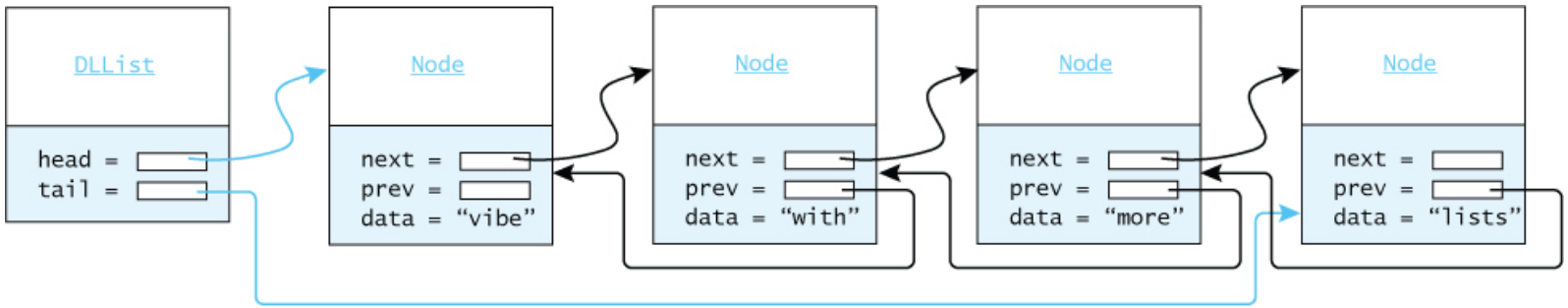
Limitations of Single-Linked Lists

- To insert a Node, we need to find the Node prior to it
- To remove a Node, we need to find the Node prior to it
- We can only traverse the List forward

We can fix some of these issues by modifying Nodes to keep track of both next and previous Nodes

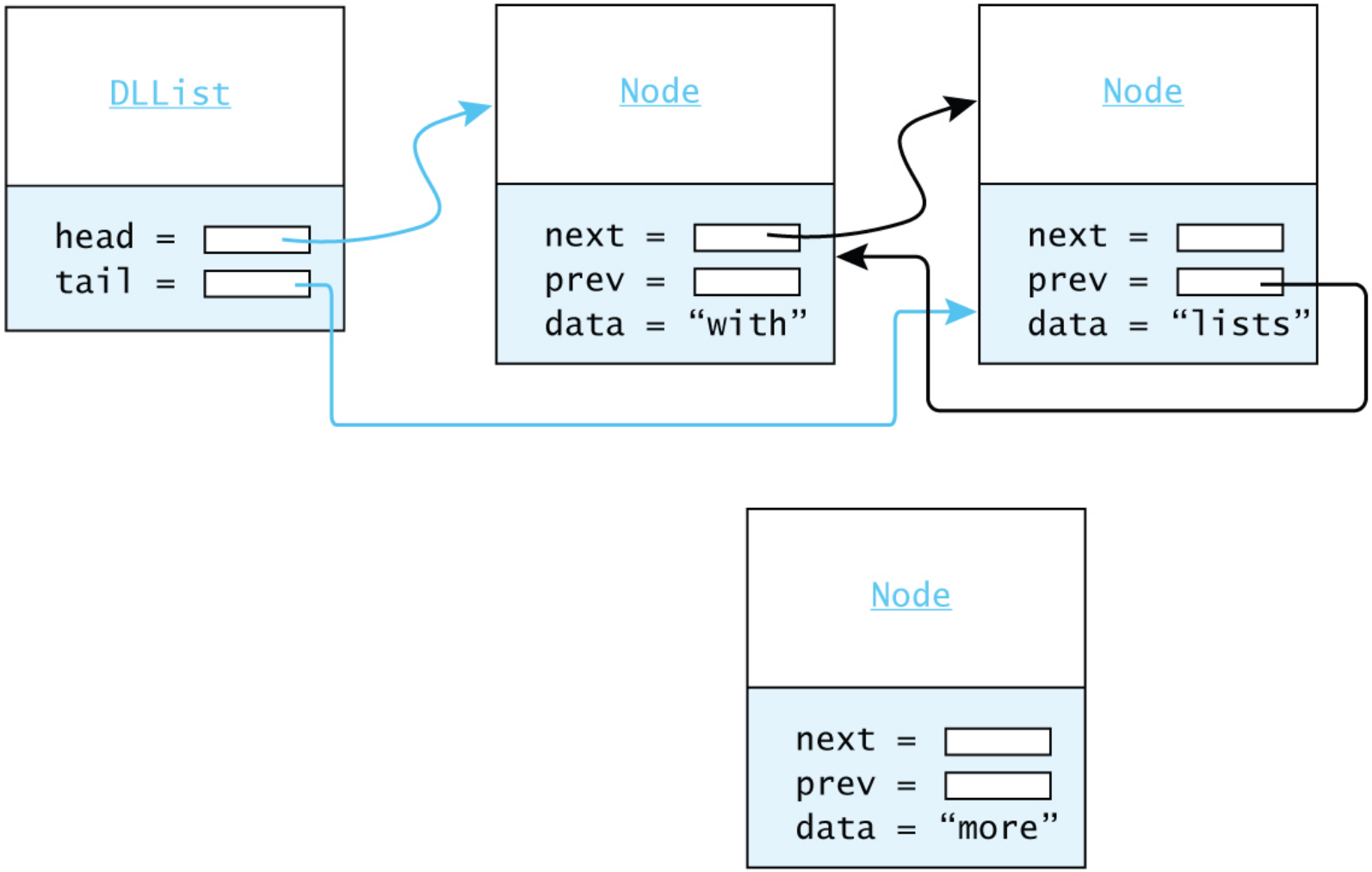
Double-Linked List of Nodes

- `DLList` links to two Nodes in the list – called the `head` and `tail`
- Each Node connects to the next Node *and* the previous Node
- The end of the list is marked with a Node that has a null `next` value
- The beginning of the list is marked with a Node that has a null `prev` value
- All Nodes have two references, so updating data means more updates in the structure



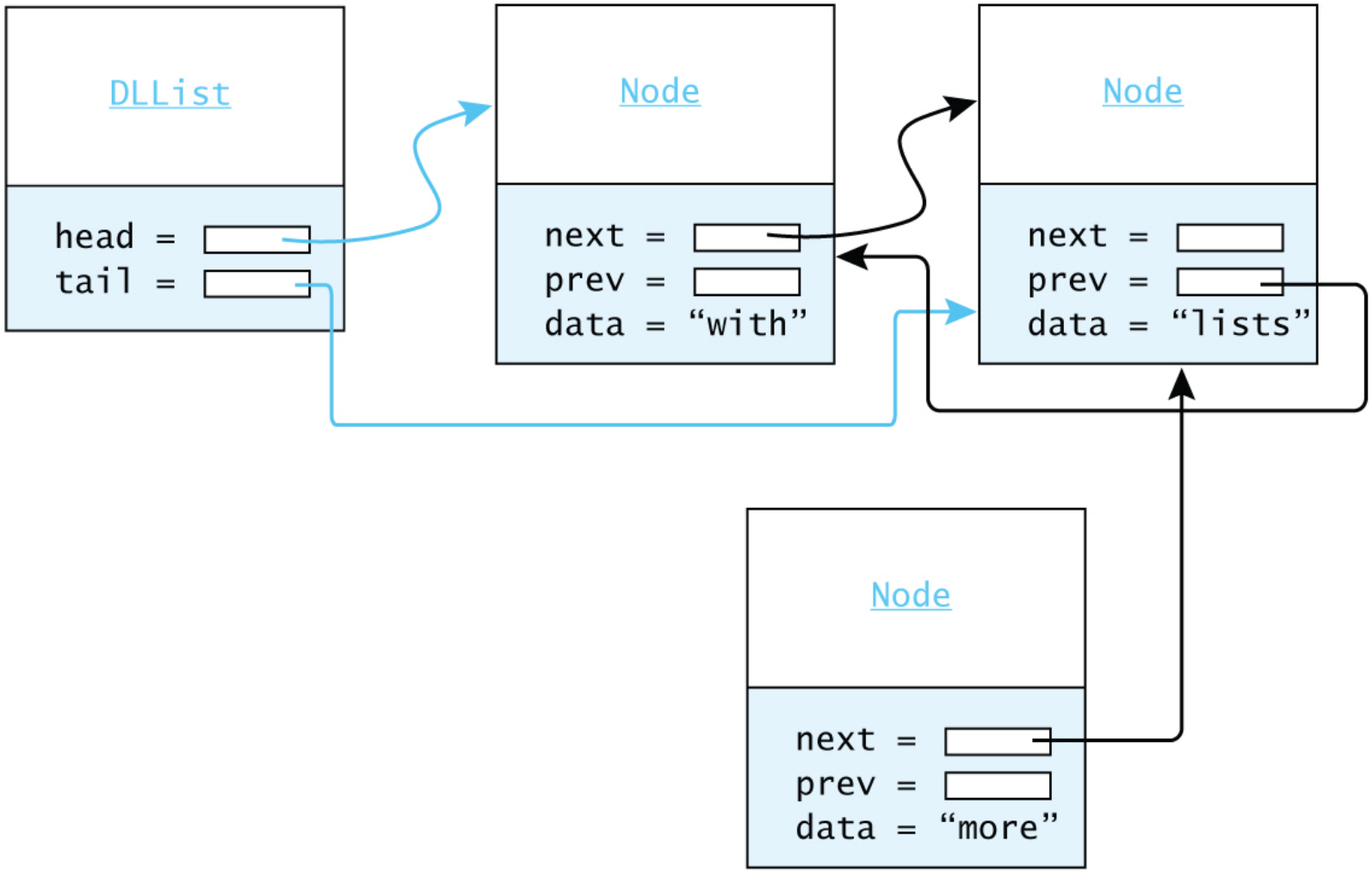
A `DLList` of `String`

Inserting a Node into a `DLList`



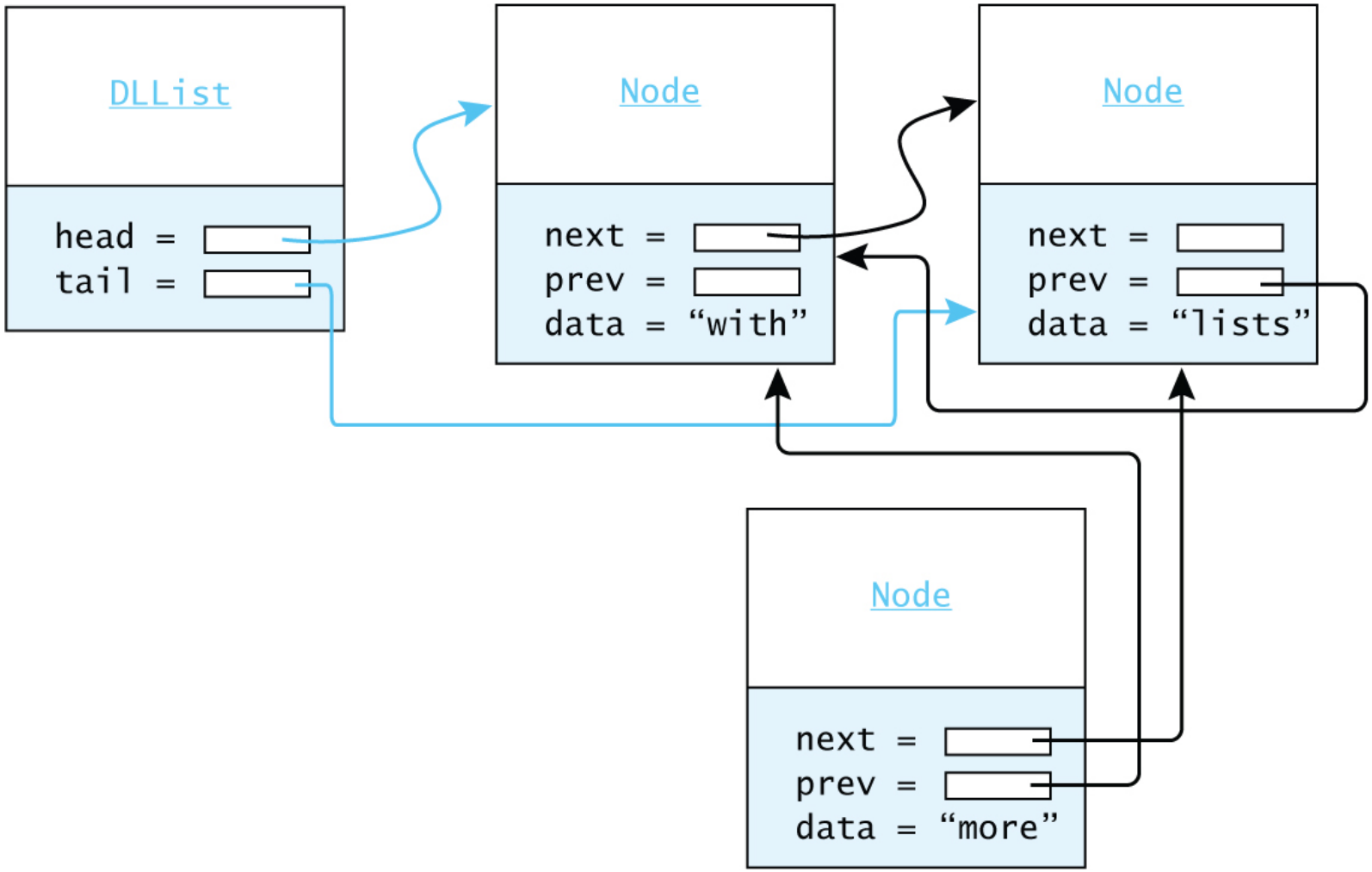
Inserting "more" after "with" Node: Create a new Node

Inserting a Node into a `DLList`



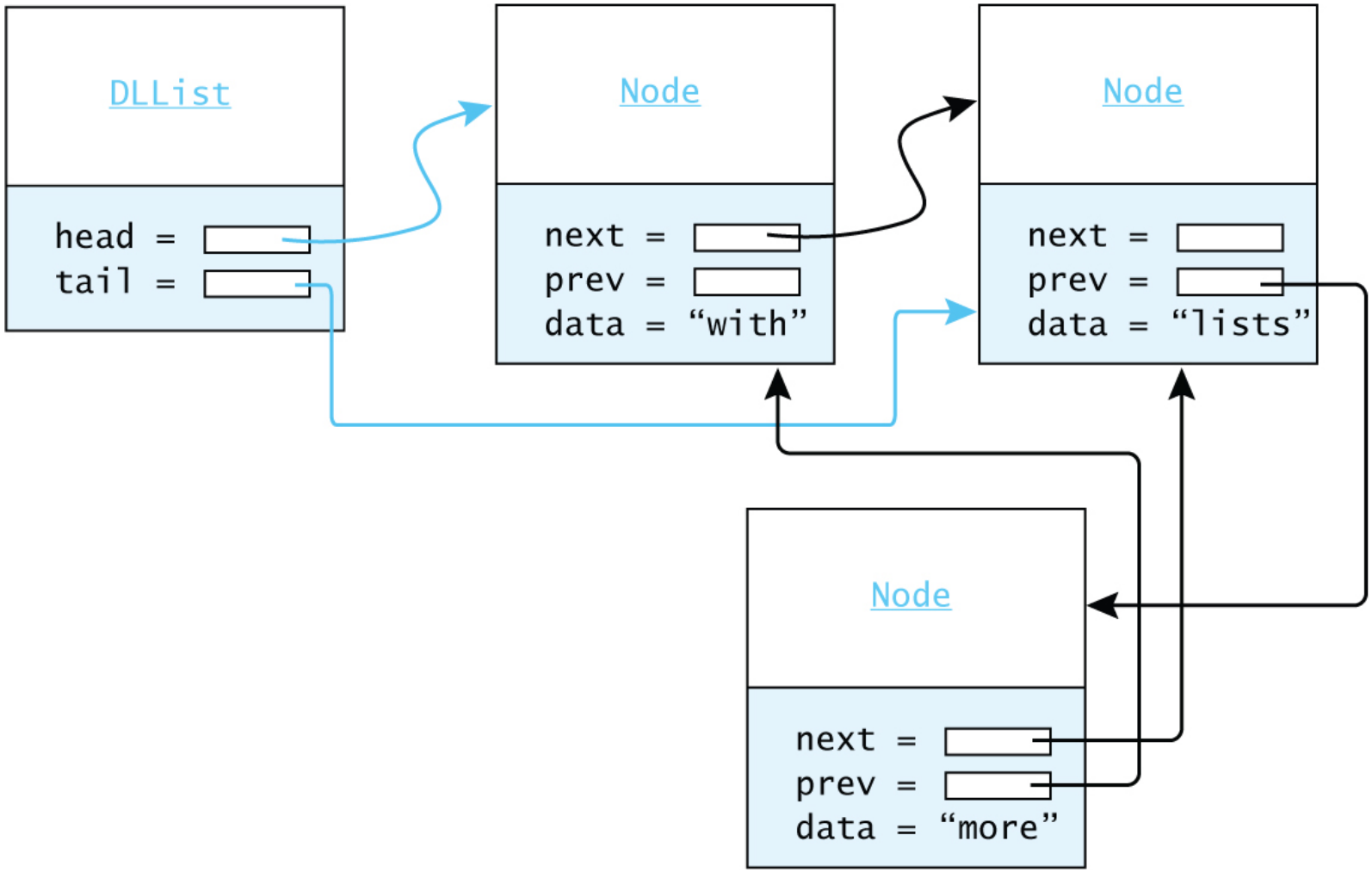
Copy `next` reference from "with" to "more"

Inserting a Node into a `DLList`



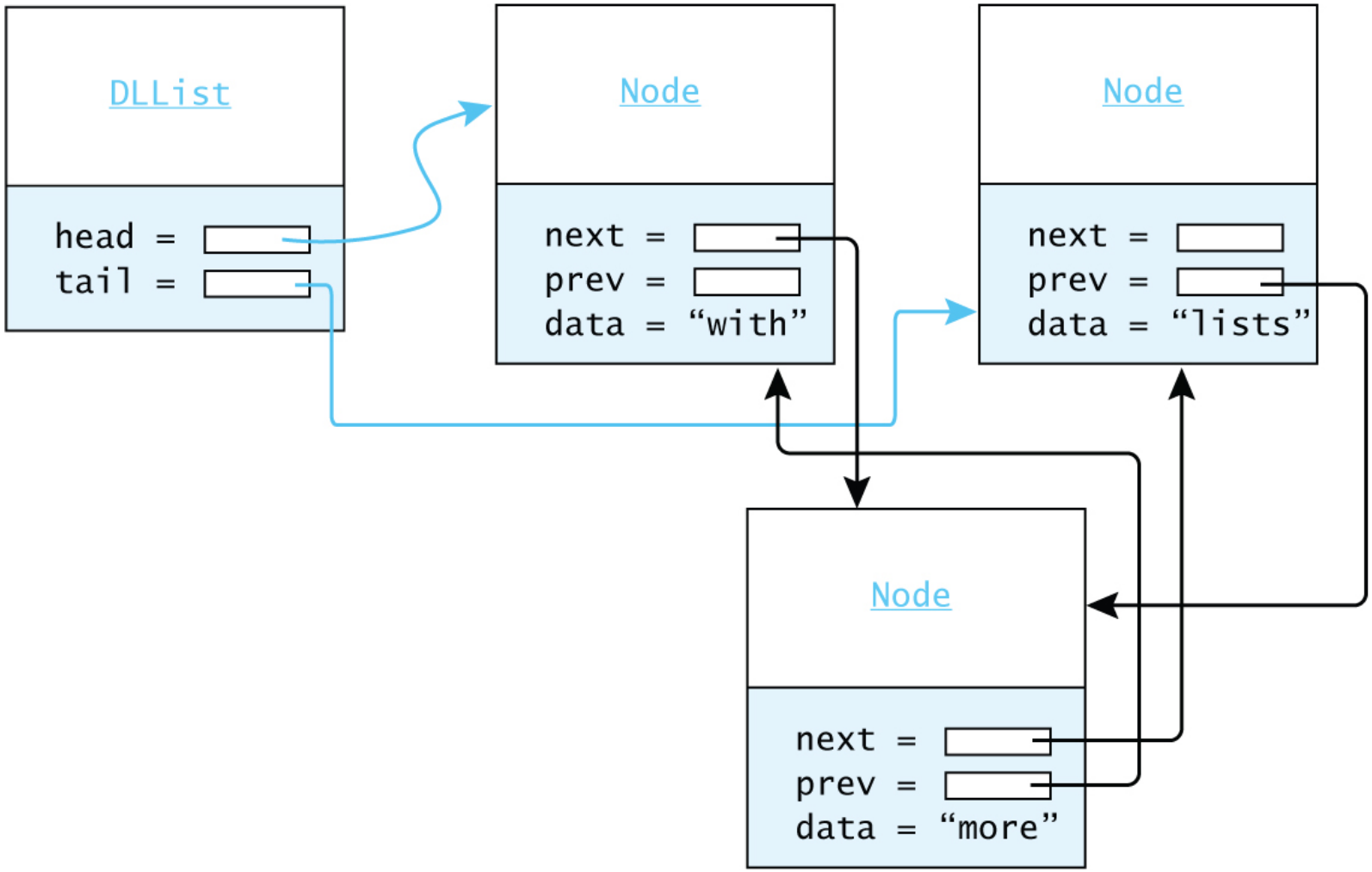
Copy `prev` reference from "lists" to "more"

Inserting a Node into a `DLList`



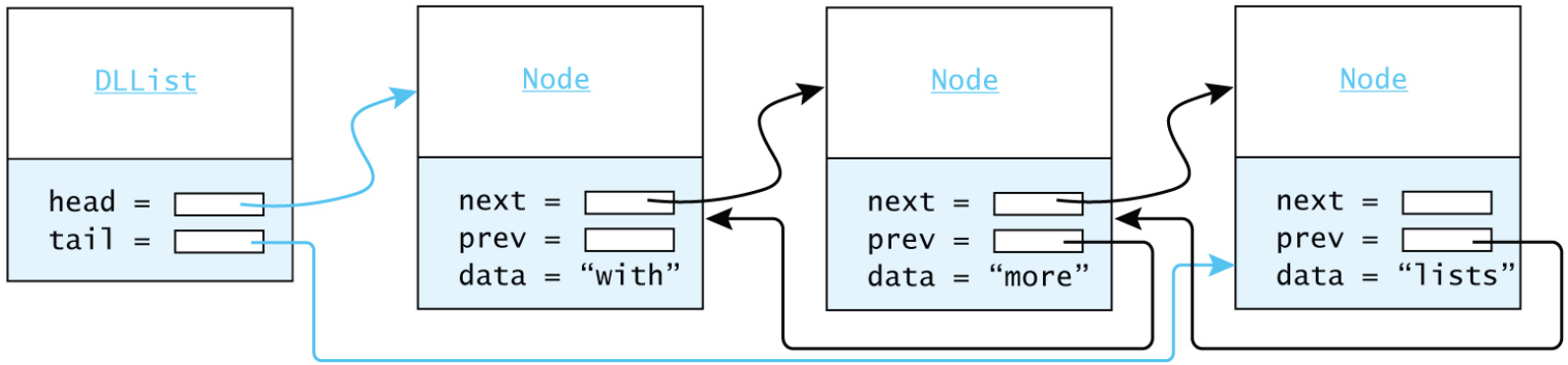
Change `prev` reference of "lists"

Inserting a Node into a `DLList`



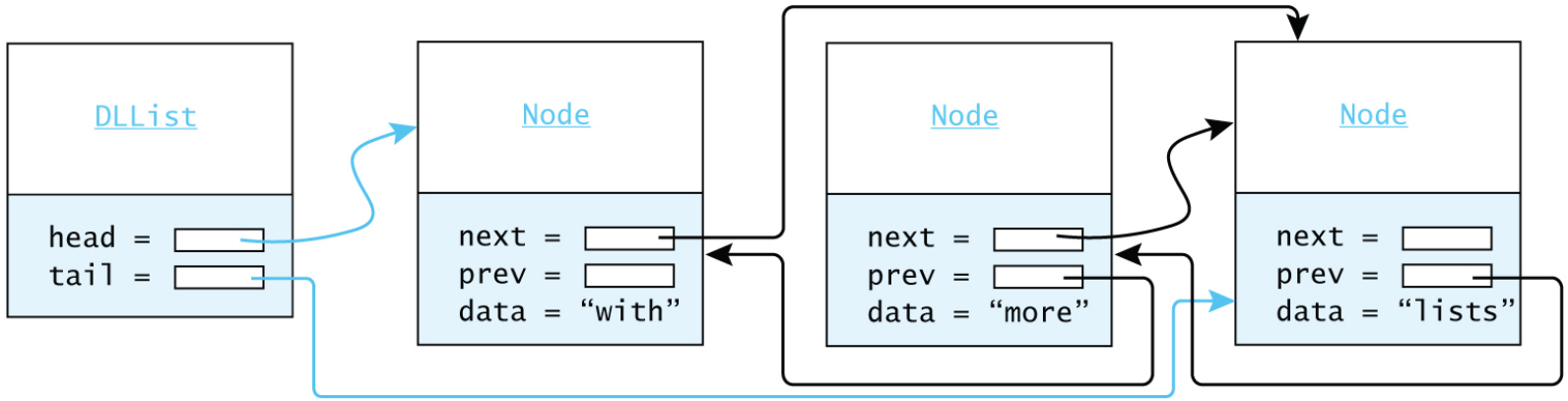
Change `next` reference of "with"

Removing a Node from a `DLList`



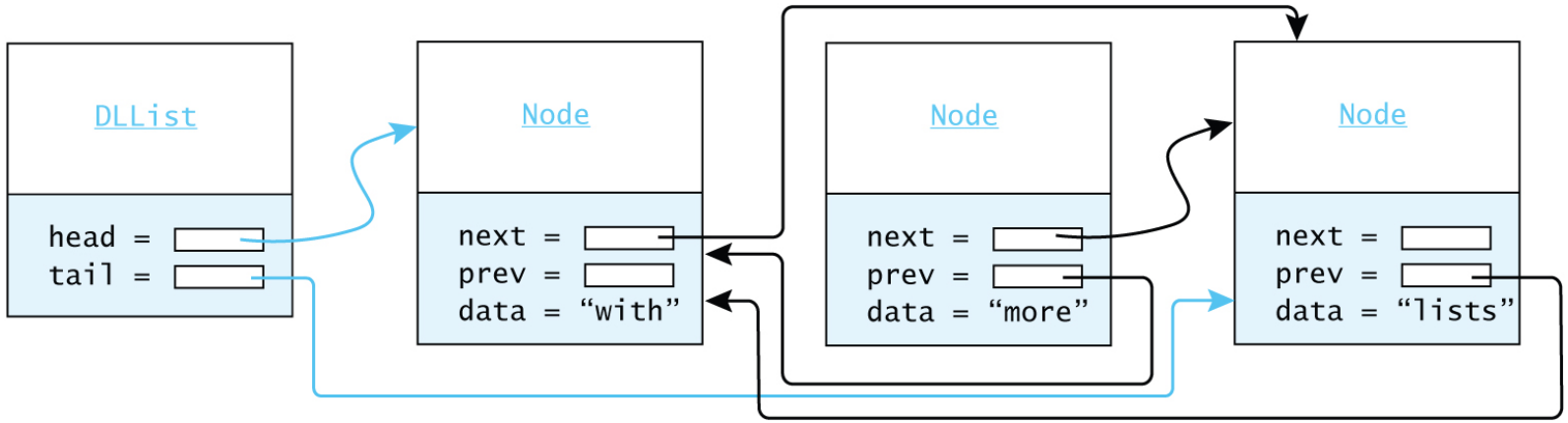
Removing "more" from list

Removing a Node from a `DLList`



Copy `next` reference from "more" to "with"

Removing a Node from a `DLList`



Copy `prev` reference from "more" to "lists"

DLList Implementation

- Modifications to the structure require more reference updates
- More edge cases – changing the front and back are both special cases
- Do query operations change? `get`? `set`? `indexOf`? `size`?
- These data structures are all missing an important tool to iterate through the values they hold – an *Iterator*