Linked lists

Linked lists

Inserting and removing elements in the *middle* of a dynamic array takes O(n) time

- (though inserting at the end takes O(1) time)
- (and you can also delete from the middle in O(1) time if you don't care about preserving the order)

A *linked list* supports inserting and deleting elements from any position in constant time

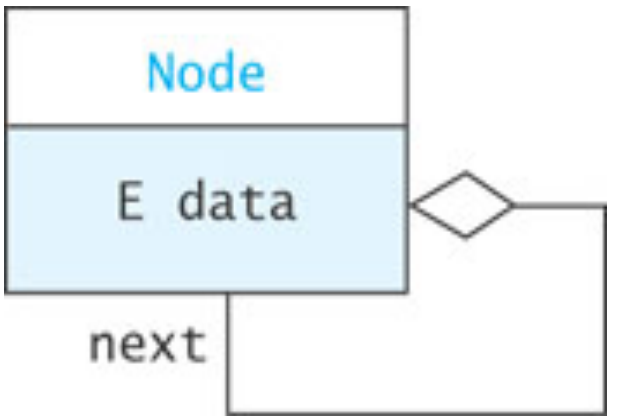
- But it takes O(n) time to access a specific position in the list

Singly-linked lists

A singly-linked list is made up of *nodes*, where each node contains:

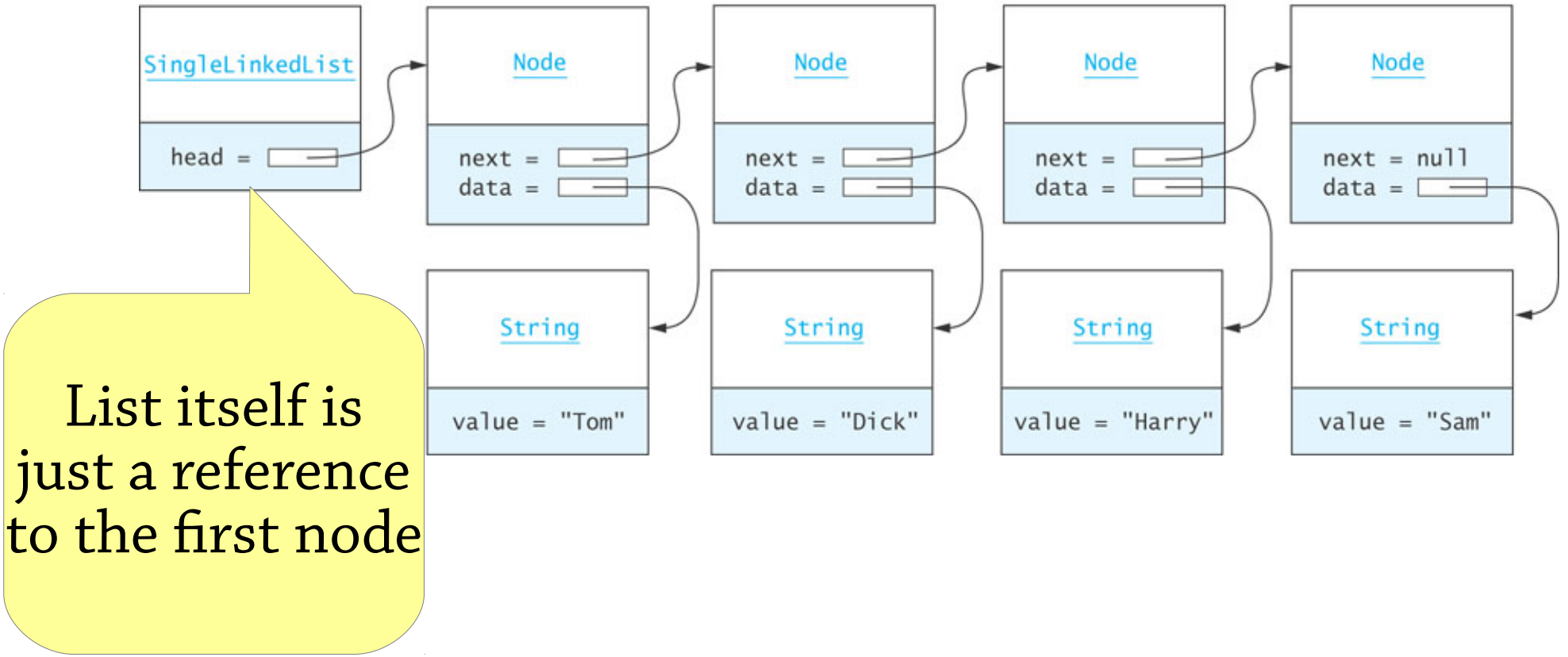
- some data (the node's value)
- a link (reference) to the next node in the list

```
class Node<E> {
  E data;
  Node<E> next;
}
```



Singly-linked lists

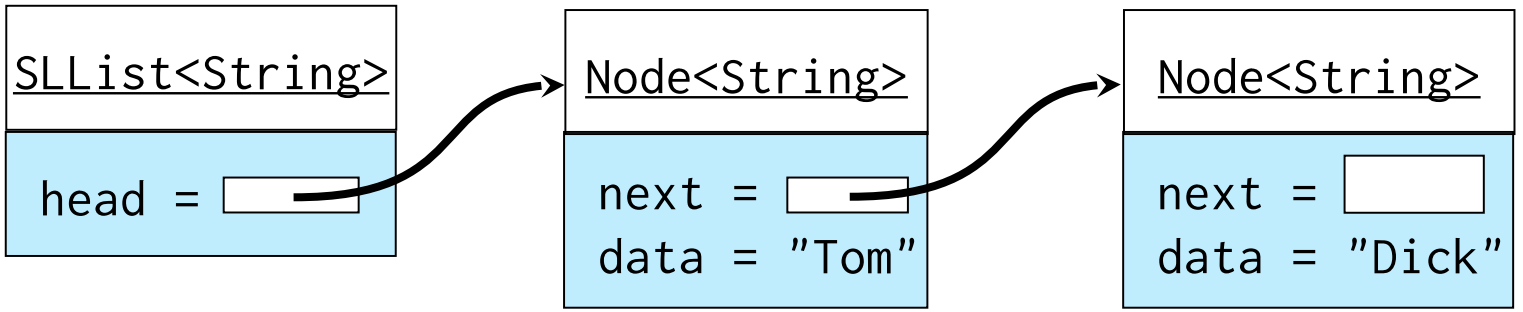
Linked-list representation of the list ["Tom", "Dick", "Harry", "Sam"]:



Operations on linked lists

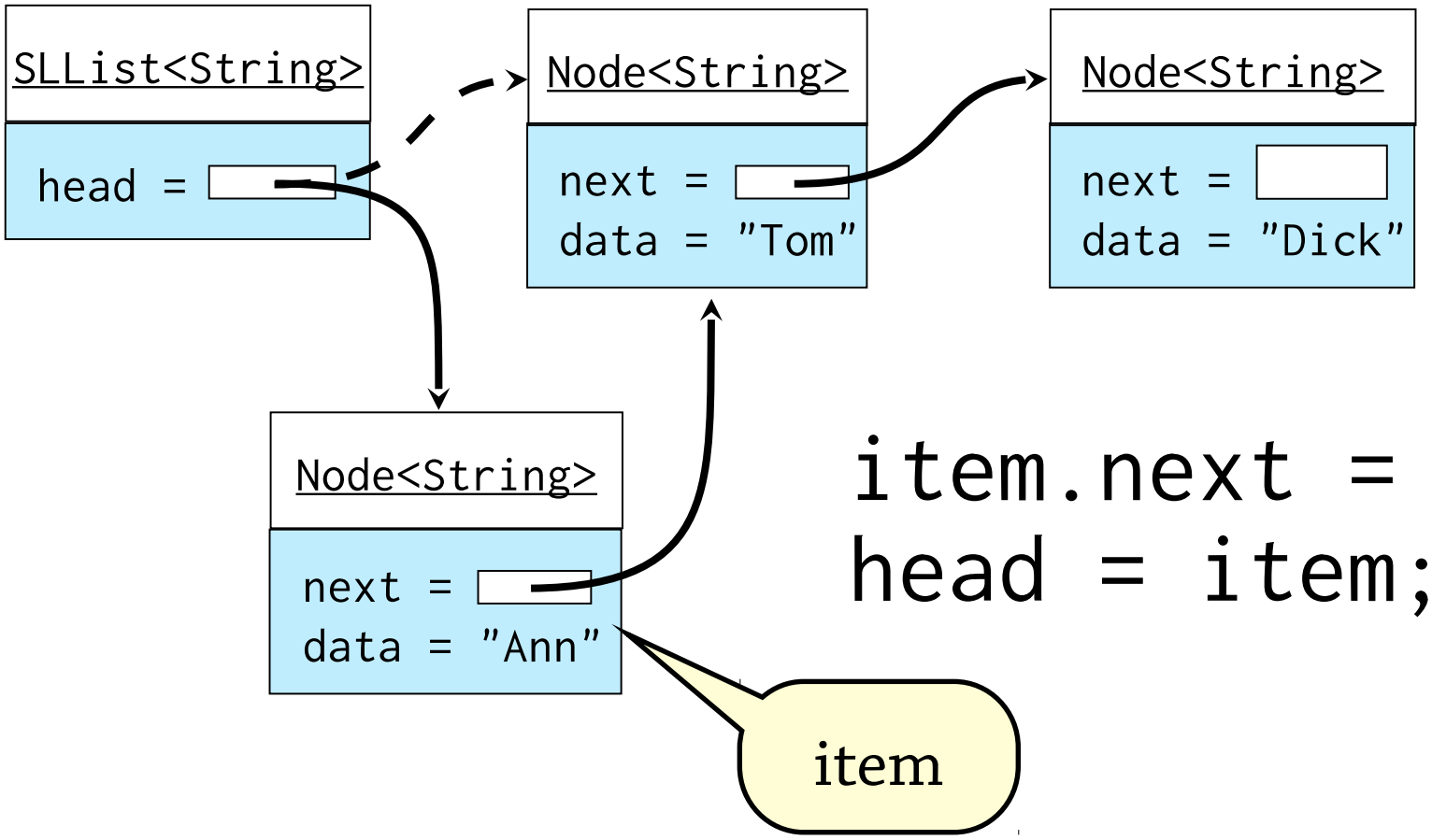
```
// Insert item at front of list
void addFirst(E item)
// Insert item after another item
void addAfter(Node<E> node, E item)
// Remove first item
void removeFirst()
// Remove item after another item
void removeAfter(Node<E> node)
```

Example list



Example of addFirst(E item)

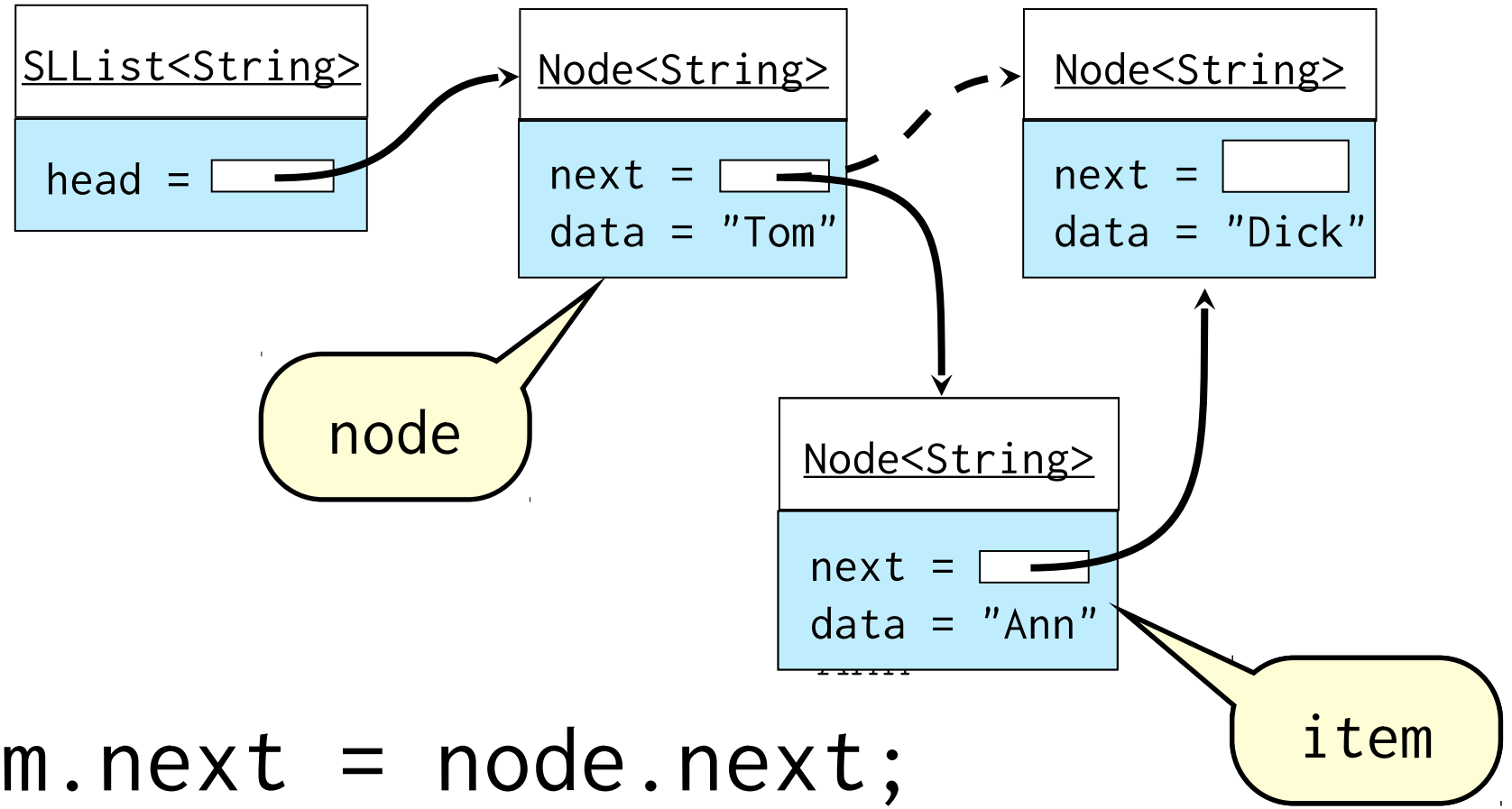
Calling `addFirst("Ann")`:




```
item.next = head;
head = item;
```

Example of addAfter

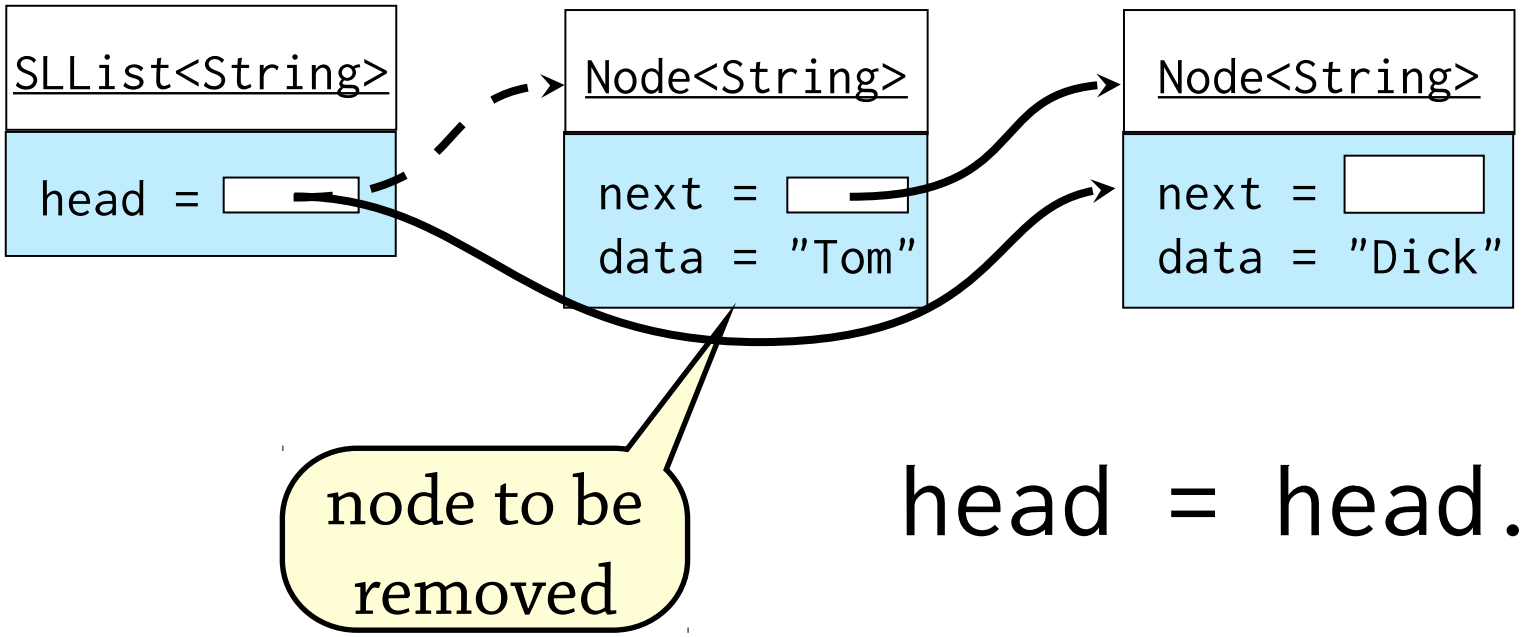
Calling addAfter(tom, “Ann”):




```
item.next = node.next;
node.next = item;
```

Example of removeFirst

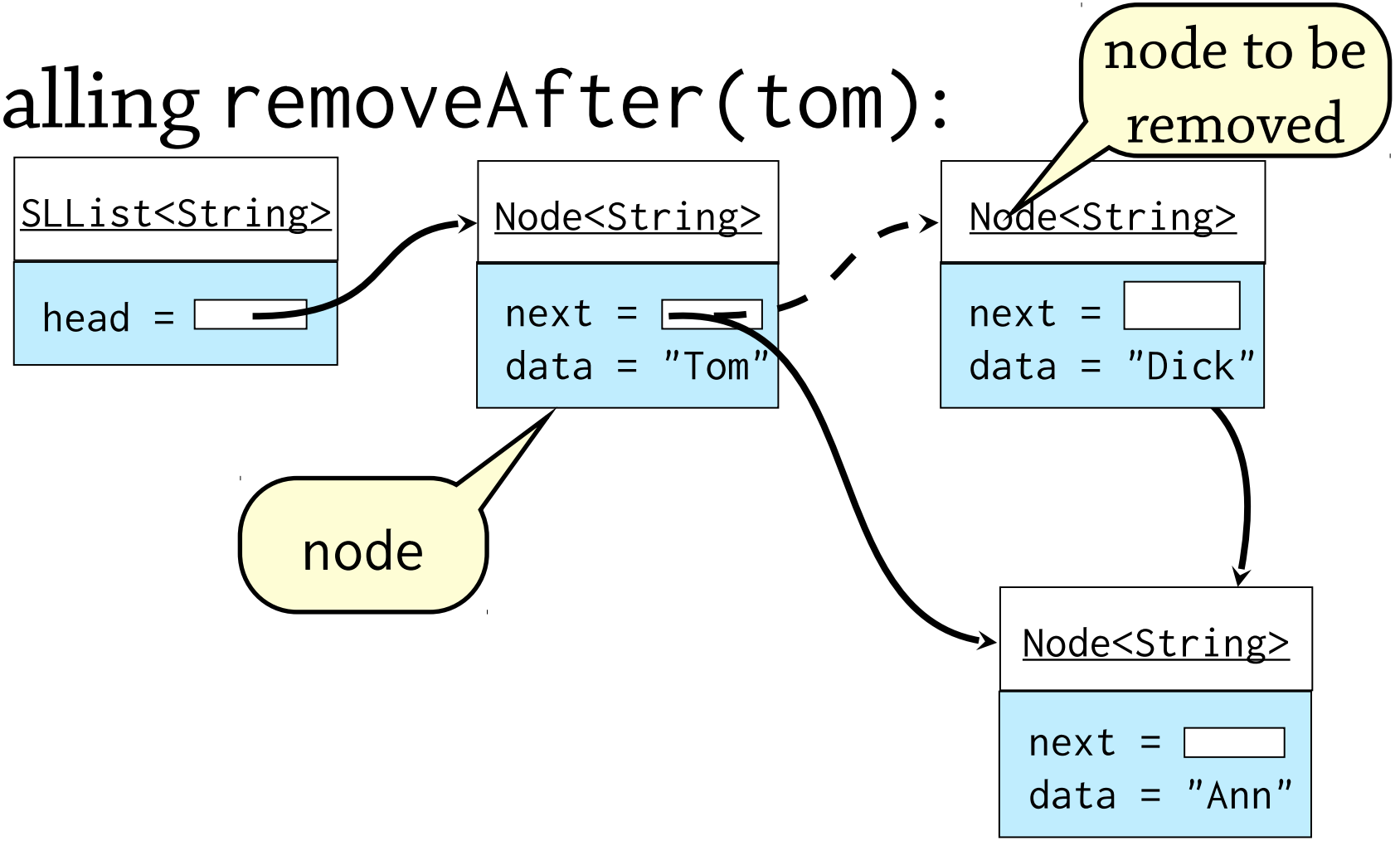
Calling `removeFirst()`:




```
head = head.next;
```

Example of removeAfter

Calling `removeAfter(tom)`:




```
node.next = node.next.next;
```

A problem

It's bad API design to need both `addFirst` and `addAfter` (likewise `removeFirst` and `removeAfter`):

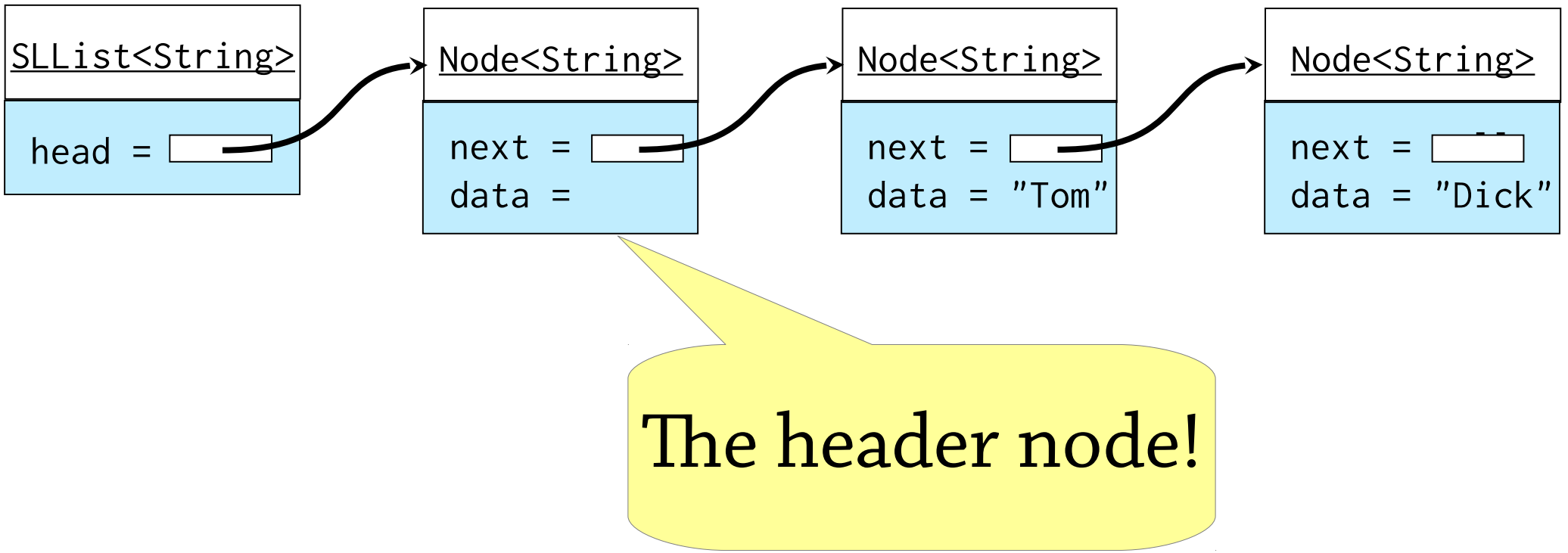
- Twice as much code to write – twice as many places to introduce bugs!
- Users of the list library will need special cases in their code for dealing with the first node

Idea: add a *header node*, a fake node that sits at the front of the list but doesn't contain any data

Instead of `addFirst(x)`, we can do `addAfter(headerNode, x)`

List with header node (16.1.1)

If we want to add “Ann” before “Tom”, we can do `addAfter(head, “Ann”)`



Doubly-linked lists

In a singly-linked list you can only go *forwards* through the list:

- If you're at a node, and want to find the previous node, too bad! Only way is to search forward from the beginning of the list

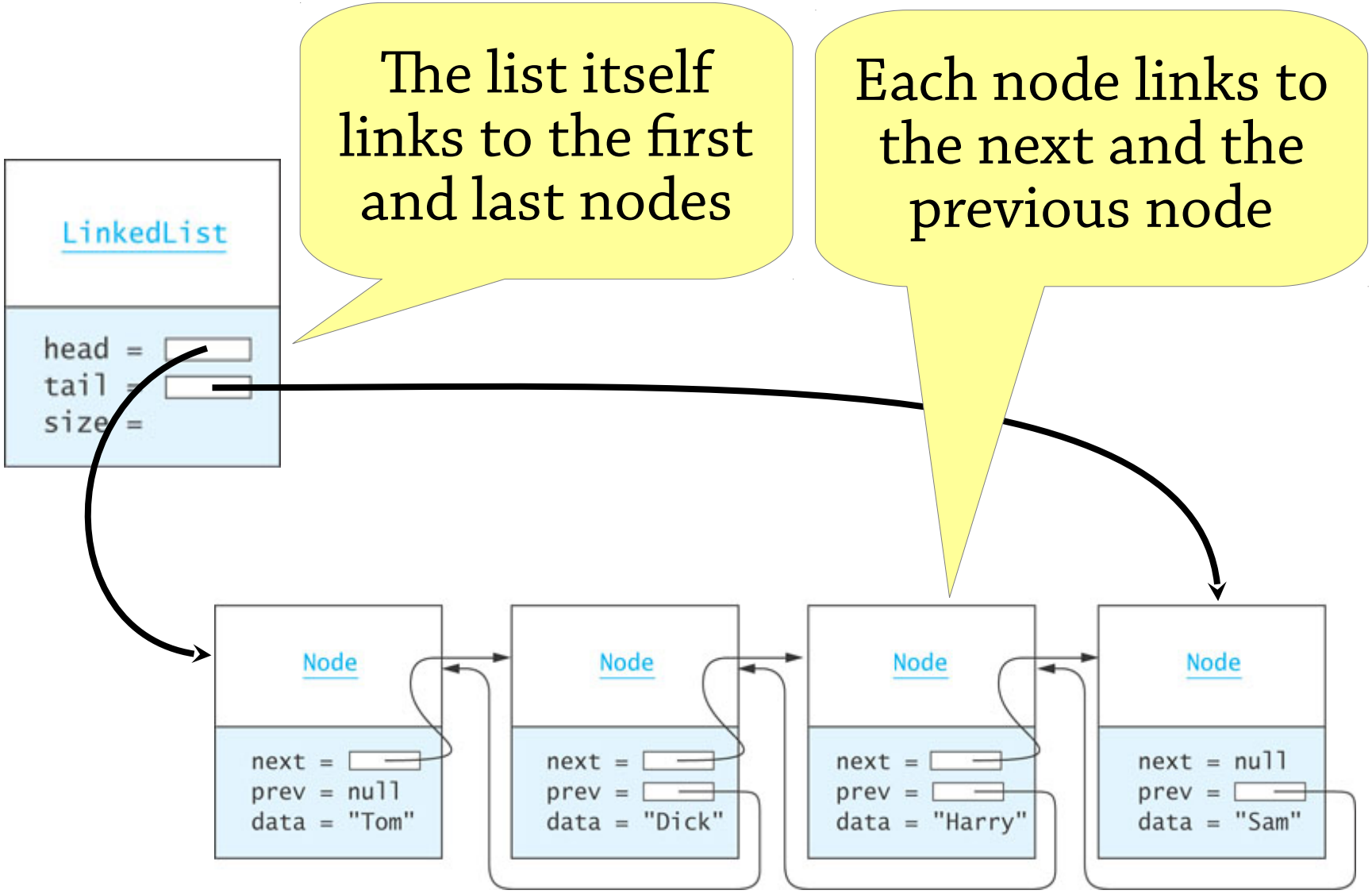
In a *doubly-linked list*, each node has a link to the next *and the previous* nodes

You can in O(1) time:

- go forwards and backwards through the list
- insert a node before or after the current one
- modify or delete the current node

The “classic” data structure for sequential access

A doubly-linked list

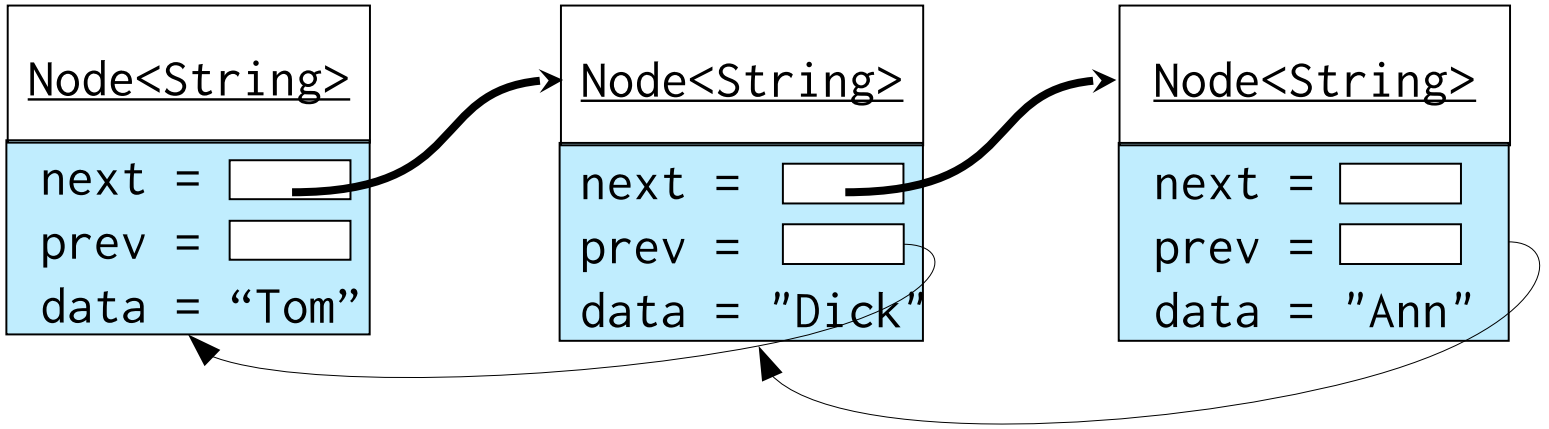


Insertion and deletion in doubly-linked lists

Similar to singly-linked lists, but you have to update the `prev` pointer too.

To delete the current node the idea is:

```
node.next.prev = node.prev;
node.prev.next = node.next;
```



Insertion and deletion in doubly-linked lists, continued

To delete the current node the idea is:

```
node.next.prev = node.prev;
node.prev.next = node.next;
```

But this CRASHES if we try to delete the first node, since then `node.prev == null`! Also, if we delete the first node, we need to update the list object's `head`.

Lots and lots of special cases for all operations:

- What if the node is the first node?
- What if the node is the last node?
- What if the list only has one element so the node is both the first *and* the last node?

Getting rid of the special cases

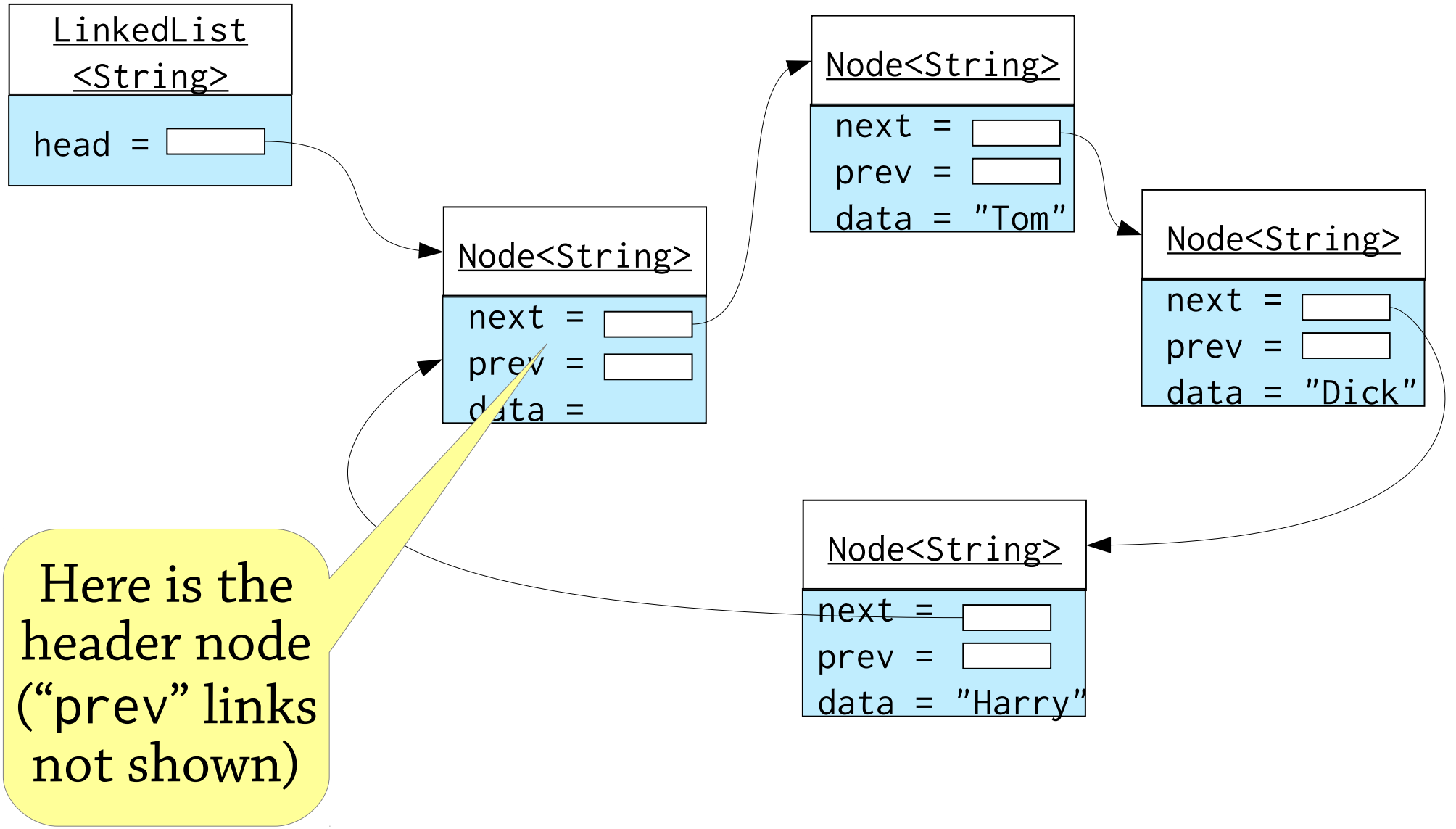
How can we get rid of these special cases?

One idea : use a header node like for singly-linked lists, but also a footer node.

- `head` and `tail` will point at the header and footer node
- No data node will have `null` as its `next` or `prev`
- All special cases gone!
- Small problem: allocates two extra nodes per list

A cute solution: *circularly-linked list with header node*

Circularly-linked list with header node



Circularly-linked list with header node

Works out quite nicely!

- `head.next` is the first element in the list
- `head.prev` is the last element
- you never need to update `head`
- no node's `next` or `prev` is ever `null`
- so no special cases!

You can even make do without the header node – then you have one special case, when you need to update `head`

Stacks and lists using linked lists

You can implement a stack using a linked list:

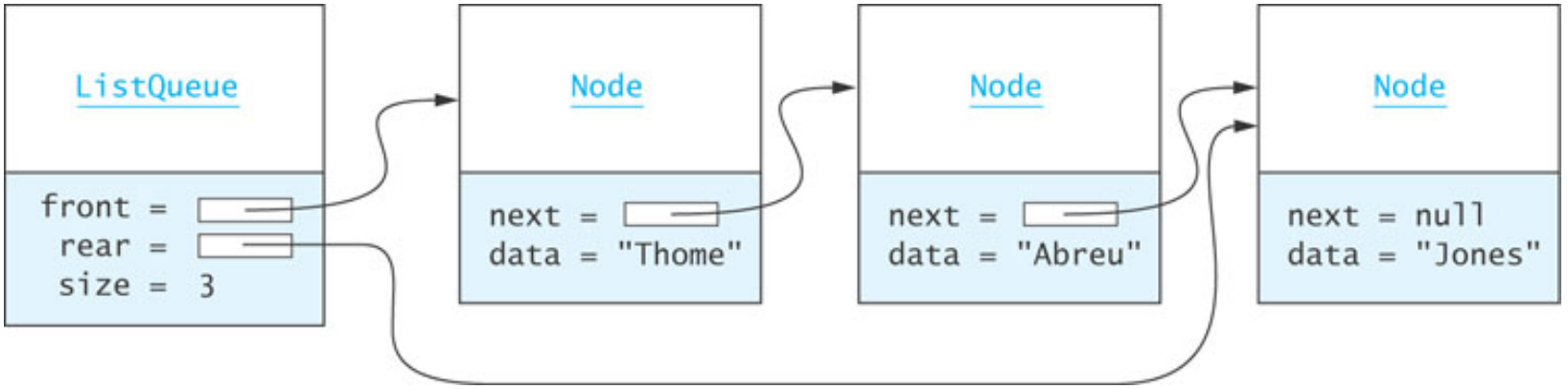
- push: add to front of list
- pop: remove from front of list

You can also implement a queue:

- enqueue: add to rear of list
- dequeue: remove from front of list

A queue as a singly-linked list

We can implement a queue as a singly-linked list with an extra `rear` pointer:



We enqueue elements by adding them to the back of the list:

- Set `rear.next` to the new node
- Update `rear` so it points to the new node

Linked lists vs dynamic arrays

Dynamic arrays:

- have O(1) random access (get and set)
- have amortised O(1) insertion at end
- have O(n) insertion and deletion in middle

Linked lists:

- have O(n) random access
- have O(1) sequential access
- have O(1) insertion in an arbitrary place (but you have to find that place first)

Complement each other!

What's the problem with this?

```
int sum(LinkedList<Integer> list) {
  int total = 0;
  for (int i = 0; i < list.size(); i++)
    total += list.get(i);
  return total;
}
```

list.get is O(n) –
so the whole thing is
$O(n^2)$!

Better!

```
int sum(LinkedList<Integer> list) {
  int total = 0;
  for (int i: list)
    total += i;
  return total;
}
```

Remember –
linked lists are for
sequential access only

Linked lists – summary

Provide *sequential access* to a list

- Singly-linked – can only go forwards
- Doubly-linked – can go forwards or backwards

Many variations – header nodes, circular lists – but they all implement the same abstract data type (interface)

Can insert or delete or modify a node in O(1) time

But unlike arrays, random access is O(n)

Java: `LinkedList<E>` class

Iterators in Java

java.util.Iterator

- `Iterator<E>` is an interface which provides a uniform way to enumerate all elements in a `Collection<E>`, e.g. a `List<E>` or a `Set<E>`.
- Minimal implementation is:
  - `boolean hasNext()` – is there another element?
  - `E next()` – give me next element
- There is also an optional method for removing the last element returned by `next`.
  - `void remove()`

java.util.Iterator

- Collections can provide a default iterator by implementing the Iterable<E> interface with this method:
  - `Iterator<E> iterator()`
- Classes that implement Iterable can be looped over using the enhanced for-loop, just like arrays.
  - ```
    void printAll(Iterable<E> s) {
      for (E e : s) {
        System.out.println(e.toString());
      }
    }
    ```
- See lecture code for an example of an implementation of an iterator over a binary tree.
    ```

Tail recursion
(not on exam)

How is recursion implemented?

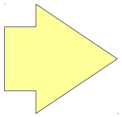
When you call function *B* from function *A*, the processor stops executing *A* and starts executing *B* (obviously)

But when *B* returns, how does it know how to go back to *A*?

Answer: the *call stack*

- Before *A* calls *B*, it will push a record of what it was doing: the next instruction to be executed, plus the values of all local variables
- When *B* returns, it will pop that record and see it should return to *A*

A recursive function



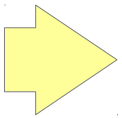
```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 3

Next line	Value of n

A recursive function



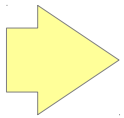
```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 3

Next line	Value of n

A recursive function



```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

Next line	Value of n

n = 3

Printed: 3

Prints 3

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

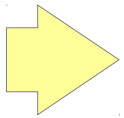
Next line	Value of n

n = 3

Printed: 3

Push state onto call stack

A recursive function



```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

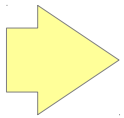
rec(3);
```

Next line	Value of n
5	3

n = 2

Printed: 3

A recursive function



```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

Next line	Value of n
5	3

n = 2

Printed: 3 2

Prints 2

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

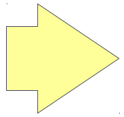
Next line	Value of n
5	3

n = 2

Printed: 3 2

Push state onto call stack

A recursive function



```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

Next line	Value of n
5	3
5	2

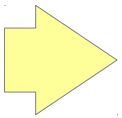
n = 1

Printed: 3 2

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```



Next line	Value of n
5	3
5	2

n = 1

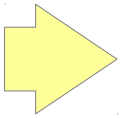
Printed: 3 2 1

Prints 1

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```



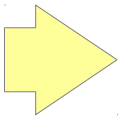
Next line	Value of n
5	3
5	2

n = 1

Printed: 3 2 1

Push state onto call stack

A recursive function



```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

Next line	Value of n
5	3
5	2
5	1

n = 0

Printed: 3 2 1

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 0

Printed: 3 2 1

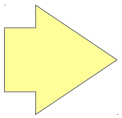
Next line	Value of n
5	3
5	2
5	1

Pop state from call stack

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```



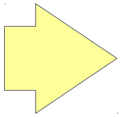
Next line	Value of n
5	3
5	2

n = 1

Printed: 3 2 1

Push state onto call stack

A recursive function



```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

Next line	Value of n
5	3
5	2
6	1

n = 0

Printed: 3 2 1

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

n = 0

Printed: 3 2 1

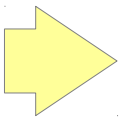
Next line	Value of n
5	3
5	2
6	1

Pop state from call stack

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```



Next line	Value of n
5	3
5	2

n = 1

Printed: 3 2 1

Pop state from call stack

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

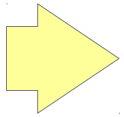
Next line	Value of n
5	3

n = 2

Printed: 3 2 1

Push state onto call stack

A recursive function



```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```

Next line	Value of n
5	3
6	2

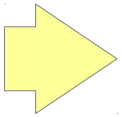
n = 1

Printed: 3 2 1

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }

rec(3);
```



Next line	Value of n
5	3
6	2

n = 1

Printed: 3 2 1 1

Prints 1

A recursive function

```
1 void rec(int n) {
2   if (n > 0) {
3     System.out.println(n);
4     rec(n-1);
5     rec(n-1);
6   }
7 }
```

How much memory does this function use?

Don't forget to include the call stack!

A recursive function

```
1 void rec(int n) {
2   if (n >
3     Syste
4     rec(n
5     rec(n
6   }
7 }
```

n levels of recursion
n items on call stack
O(n) memory use!

How much memory does this function use?

Don't forget to include the call stack!

Memory use of recursive functions

Calling a function pushes information on the call stack

Hence recursive functions use memory in the form of the call stack!

Total memory use from call stack:
O(maximum recursion depth)

Another recursive function

```
void hello() {
    System.out.println(“hello world”);
    hello();
}
```

What is this program supposed to do?

What does it actually do?

Another recursive function

```
void hello() {
    System.out.println(“hello world”);
    hello();
}
```

What is this program supposed to do?

- Print “hello world” over and over again

What does it actually do?

- `Exception in thread "main" java.lang.StackOverflowError`

The recursive call to *hello* fills the call stack!

Tail calls

```
void hello() {
   System.out.println(“hello world”);
   hello();
}
```

The recursive call is the last thing *hello* does before it returns

This is called a *tail call*, and *hello* is *tail recursive*

Idea: *don't bother pushing anything on the call stack* when making a tail call

- Since the function is going to do nothing afterwards except return again

Tail call optimisation

In languages with *tail call optimisation*:

- Tail calls don't push anything onto the call stack so don't use any stack space
- Hence tail recursion acts just like a loop
- This allows you to choose between using loops or recursion, whichever is more natural for the problem at hand

Most functional languages have TCO, since you're supposed to use tail recursion instead of looping:

- e.g. Haskell, ML, Scala, Erlang, Scheme
- but also some other civilised languages e.g. Lua

Unfortunately many languages (e.g. Java) don't :(

Is this a tail call?

```
void hello(int n) {
    if (n > 0) {
      System.out.println(“hello world”);
      hello(n-1);
    }
}
```

Is this a tail call?

```
void hello(int n) {
    if (n > 0) {
      System.out.println(“hello world”);
      hello(n-1);
    }
}
```

Yes! - nothing more happens after the recursive call to *hello*

Is this a tail call?

```
int fac(int n) {
    if (n == 0) return 1;
    else return n * fac(n-1);
}
```

Is this a tail call?

```
int fac(int n) {
   if (n == 0) return 1;
   else return n * fac(n-1);
}
```

No! - after the recursive call *fac(n-1)* returns, you have to multiply by *n*

Tail recursion using a loop

You can always write a tail-recursive function using a *while(true)*-loop instead:

```
void hello(int n) {
   while(true) {
   if (n > 0) {
     System.out.println("hello world");
     hello(n-1); n = n-1;
   } else return;
   }
}
```

Explicitly return when the recursion is finished

Instead of making a tail-recursive call, go through the loop again

Tail recursion using a loop

Tidied up a bit:

```
void hello(int n) {
    while (n > 0) {
      System.out.println(“hello world”);
      n = n-1;
    }
}
```

Searching in a binary tree

```
Node<E> search(Node<E> node, int value) {
    if (node == null) return null;
    if (value == node.value) return node;
    else if (value < node.value)
      return search(node.left);
    else
      return search(node.right);
}
```

The same, tail-recursive

```
Node<E> search(Node<E> node, int value) {
   while(true) {
   if (node == null) return null;
   if (value == node.value) return node;
   else if (value < node.value)
     node = node.left;
   else
     node = node.right;
   }
}
```

When programming in languages like Java that don't have TCO, you might need to do this transformation yourself!

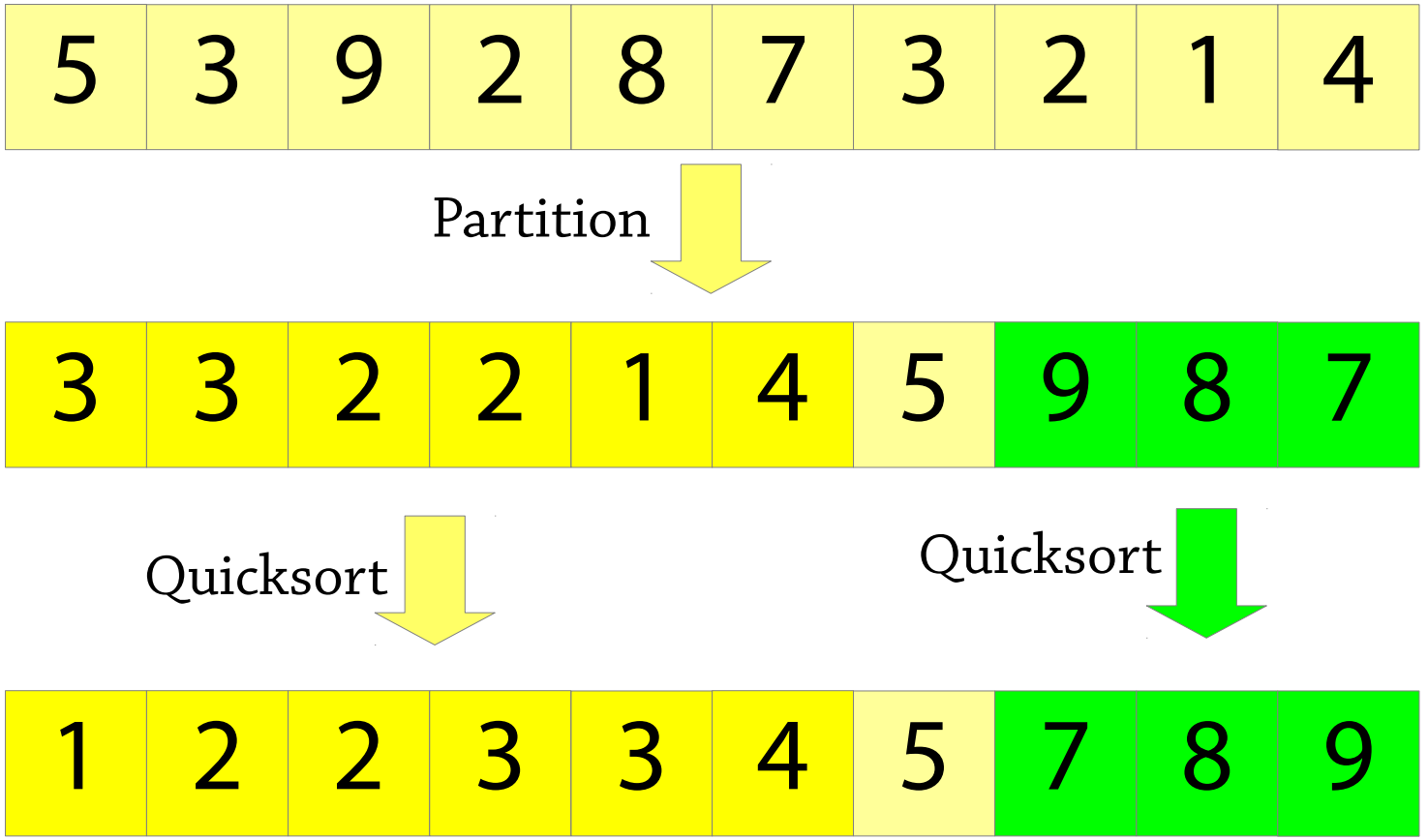
Tail calls

Remember that the total amount of extra memory used by a recursive function is **O(maximum recursion depth)**

If the language supports TCO, the amount is instead **O(maximum depth of non-tail recursive calls)** – better!

In languages without TCO, you can transform tail recursion into a loop to save stack space (memory)

A bigger example: quicksort



Quicksort

We said that quicksort was in-place, but it makes two recursive calls!

```
void sort(int[] a, int low, int high) {
   if (low >= high) return;
   int pivot = partition(a, low, high);
   sort(a, low, pivot-1);
   sort(a, pivot+1, high);
}
```

How much memory does this use in the worst case, including the call stack?

Quicksort

We said that quicksort was in-place, but it makes two recursive calls!

```
void sort(int[] a, int low, int high) {
    if (low >= hi) re
    int pivot = p
    sort(a,
    sort(a, pi
}
```

O(n), including the call stack!

How much i is use in the worst case, in s the call stack?

Quicksort

Let's make a version of quicksort that uses O(log n) stack space.

```
void sort(int[] a, int low, int high) {
   if (low >= high) return;
   int pivot = partition(a, low, high);
   sort(a, low, pivot-1);
   sort(a, pivot+1, high);
}
```

Quicksort in O(log n) space

Idea: if we are using a language with TCO, the *second* recursive call uses no stack space (it's a tail call)!

Hence, the total memory use is O(recursion depth of *first* recursive call)

So: sort the *smaller* partition with the first recursive call, and the bigger one with the second recursive call

If the array has size n, the smaller partition has size at most $n/2$, so the recursion depth is at most O(log n).

Sorting the smaller partition first

In languages with TCO (i.e. not Java), this uses O(log n) space.

```
void sort(int[] a, int low, int high) {
    if (low >= high) return;
    int pivot = partition(a, low, high);
    if (pivot – low < high – pivot) {
      sort(a, low, pivot-1);
      sort(a, pivot+1, high);
    } else {
      sort(a, pivot+1, high);
      sort(a, low, pivot-1);
    }
}
```

Sort the smaller partition first

Sorting the smaller partition first

In Java, we must transform the tail recursion into a *while(true)*-loop.

```
void sort(int[] a, int low, int high) {
   while(true) {
   if (low >= high) return;
   int pivot = partition(a, low, high);
   if (pivot – low < high – pivot) {
     sort(a, low, pivot-1);
     sort(a, pivot+1, high); low = pivot+1;
   } else {
     sort(a, pivot+1, high);
     sort(a, low, pivot-1); high = pivot-1;
   }
   }
}
```