Linked lists *(6.5, 16)*

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

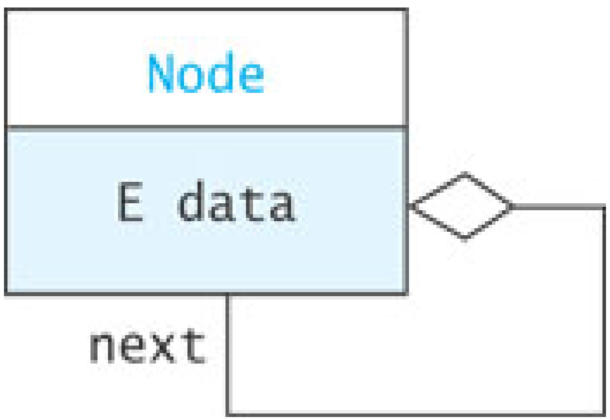
- But it takes O(n) time to access a specific position in the list

Singly-linked lists

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

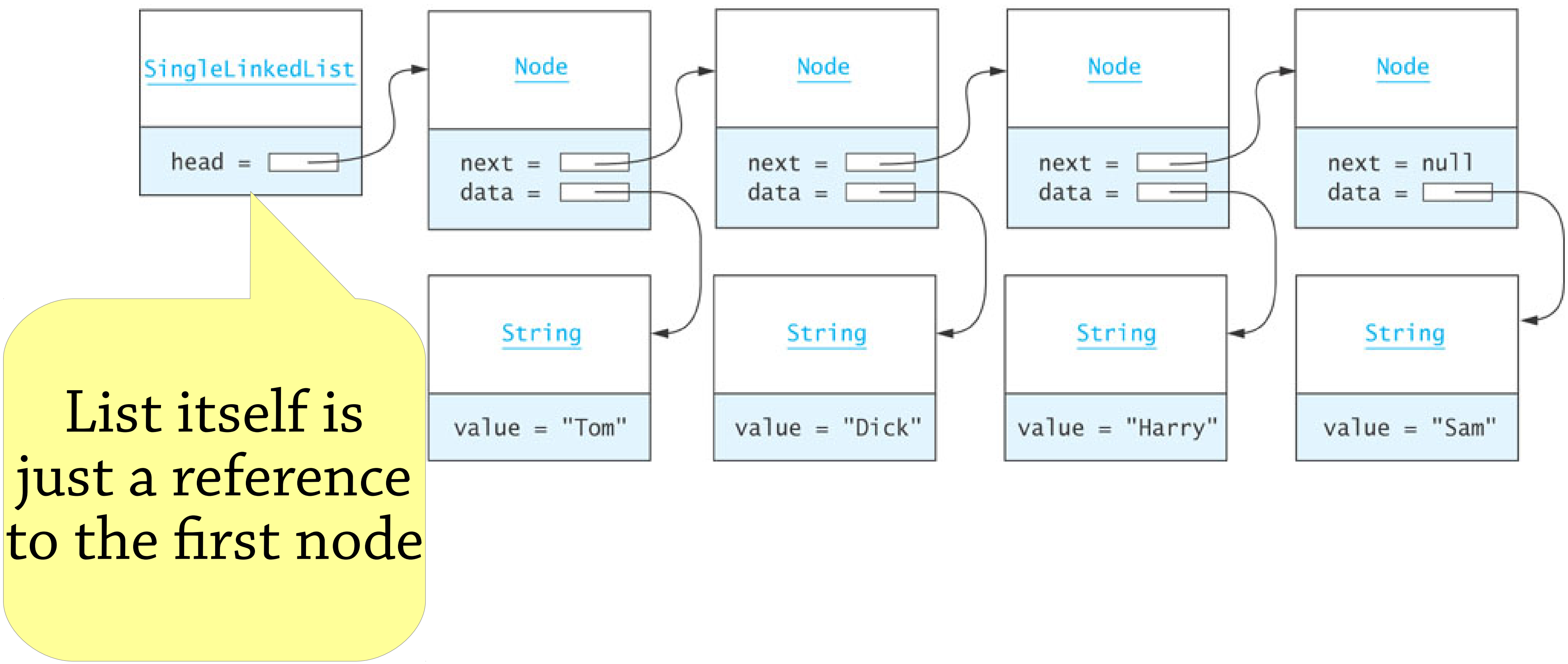
- some data (the node's value)
- a link (reference) to the next node in the list

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



Singly-linked lists

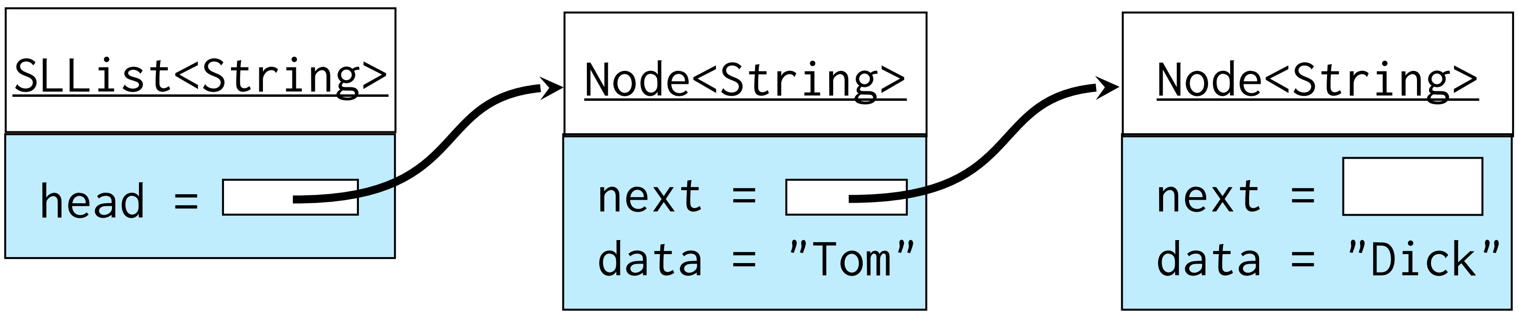
Linked-list representation of the list
["Tom", "Dick", "Harry", "Sam"]:



Operations on linked lists

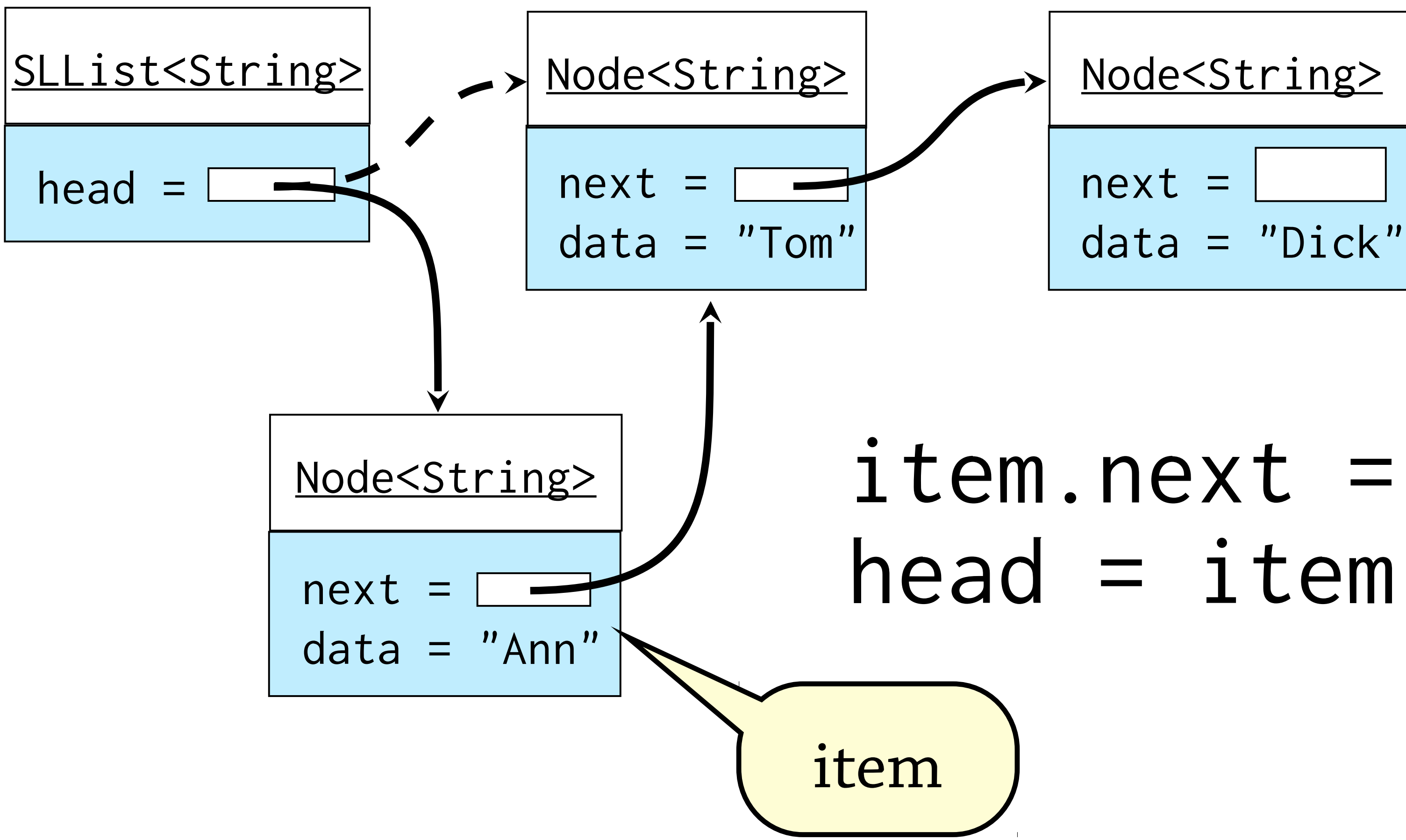
```
// Insert item at front of list
void addFirst(E item)
// Insert item after another item
void addAfter(Node<E> node, E item)
// Remove first item
void removeFirst()
// Remove item after another item
void removeAfter(Node<E> node)
```

Example list



Example of addFirst(E item)

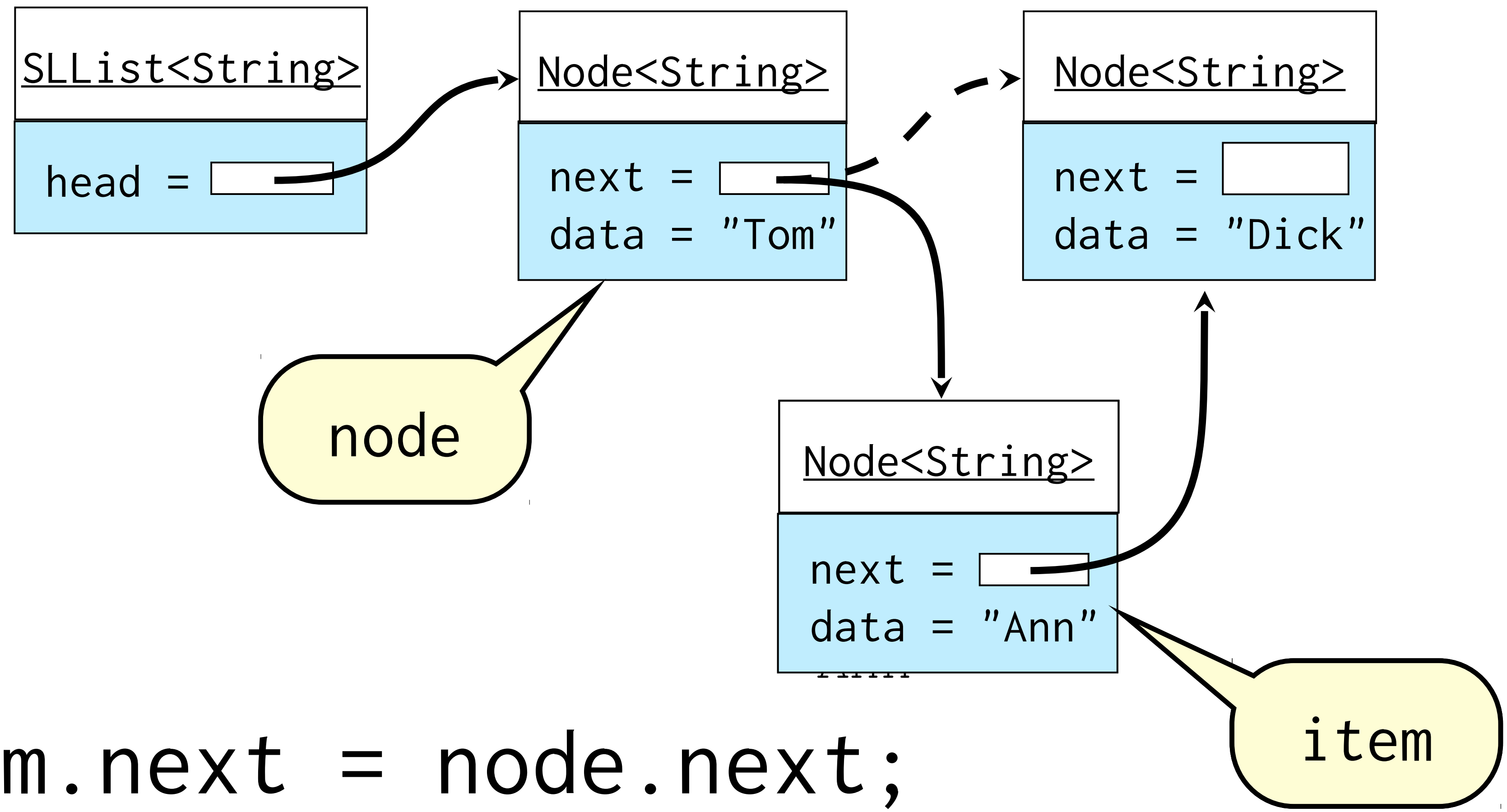
Calling addFirst("Ann"):




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

Example of addAfter

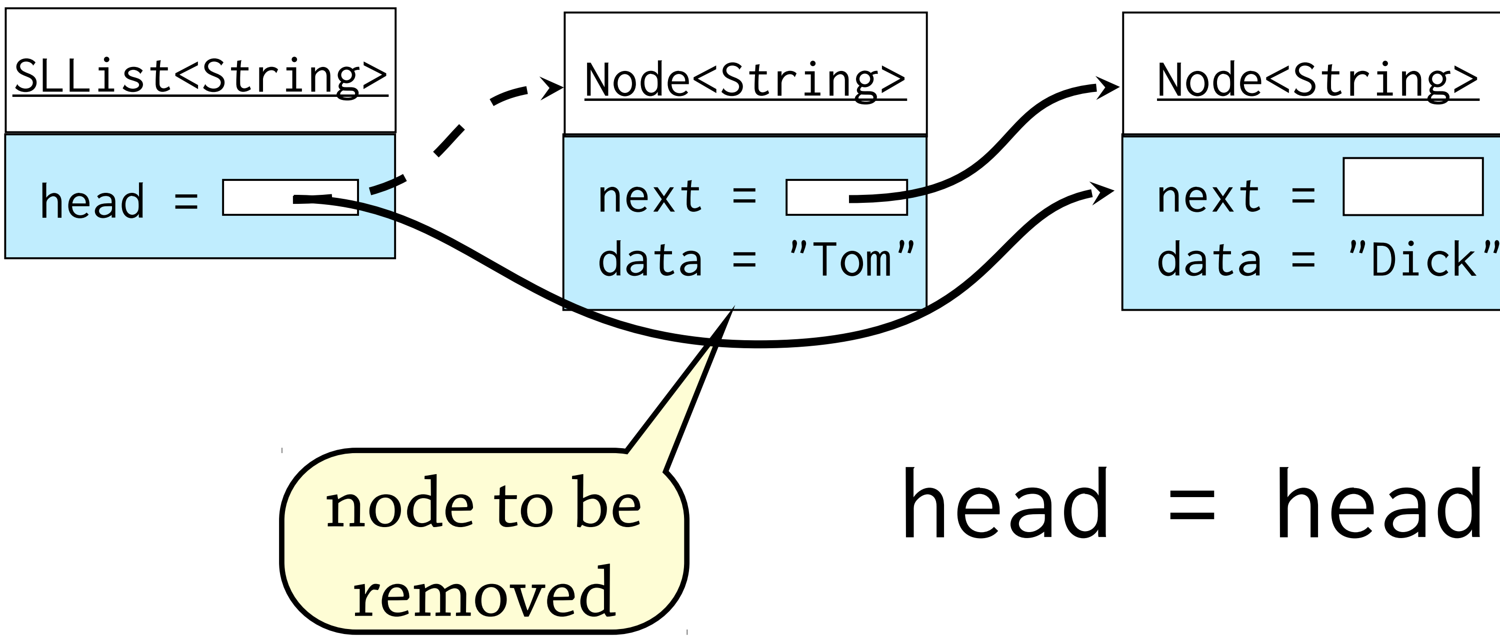
Calling `addAfter(tom, “Ann”)`:




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

Example of removeFirst

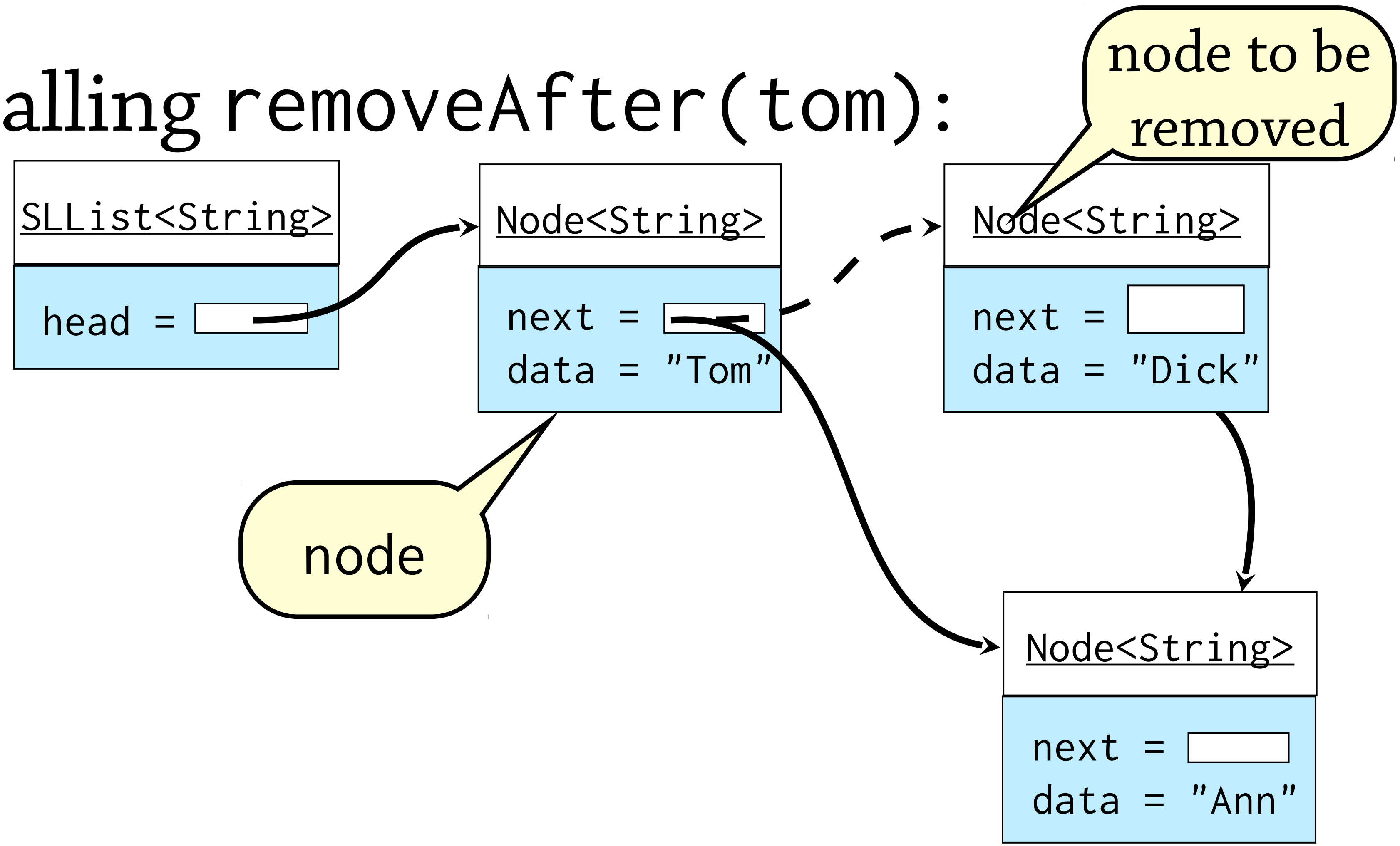
Calling `removeFirst()`:




```
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`):

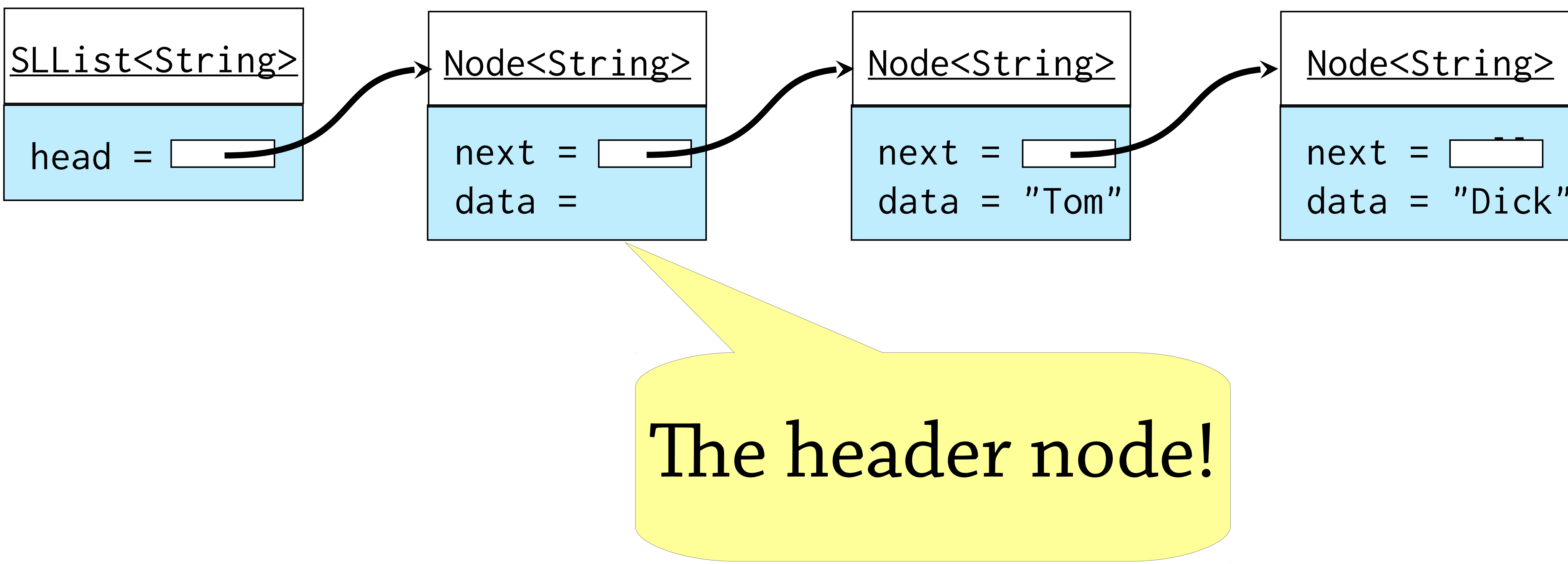
- Twice as much code to write – twice as many places to introduce bugs!
- Users of the list library will need special cases in their code for dealing with the first node

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

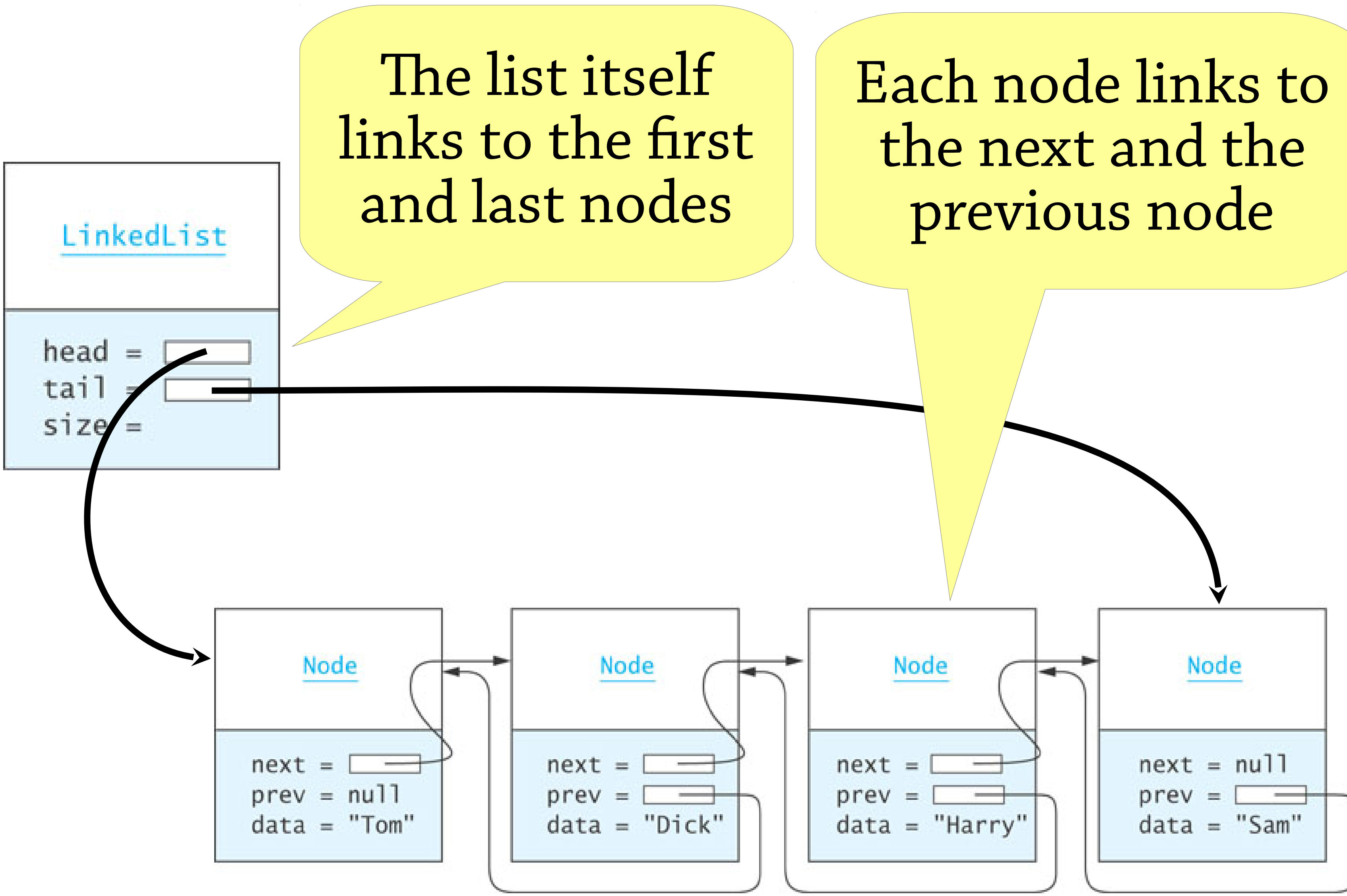
In a *doubly-linked list*, each node has a link to the next *and the previous* nodes

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

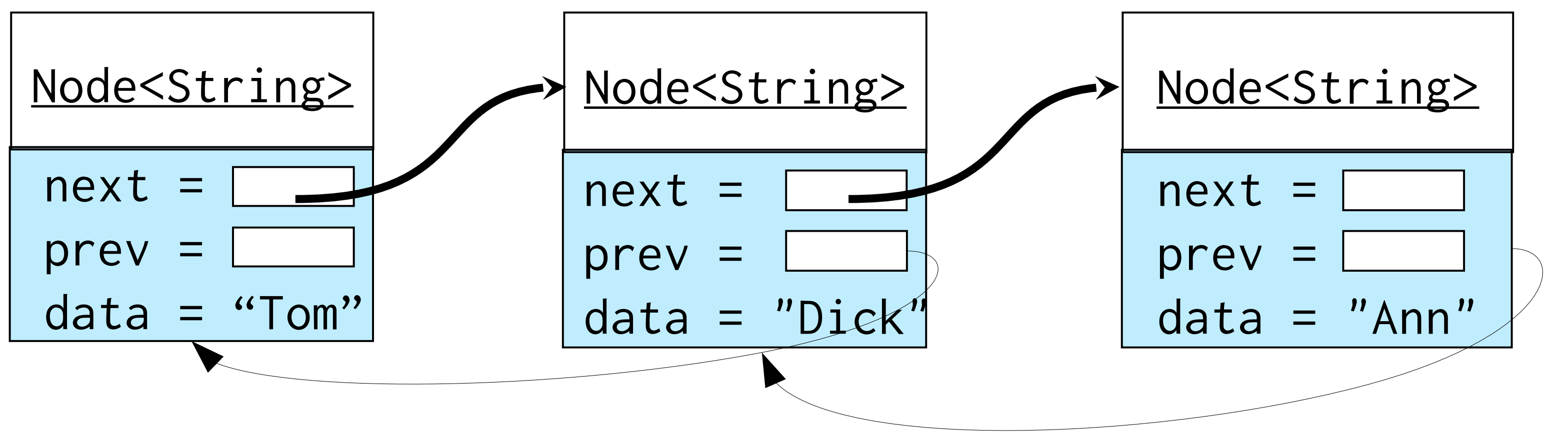


Insertion and deletion in doubly-linked lists

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

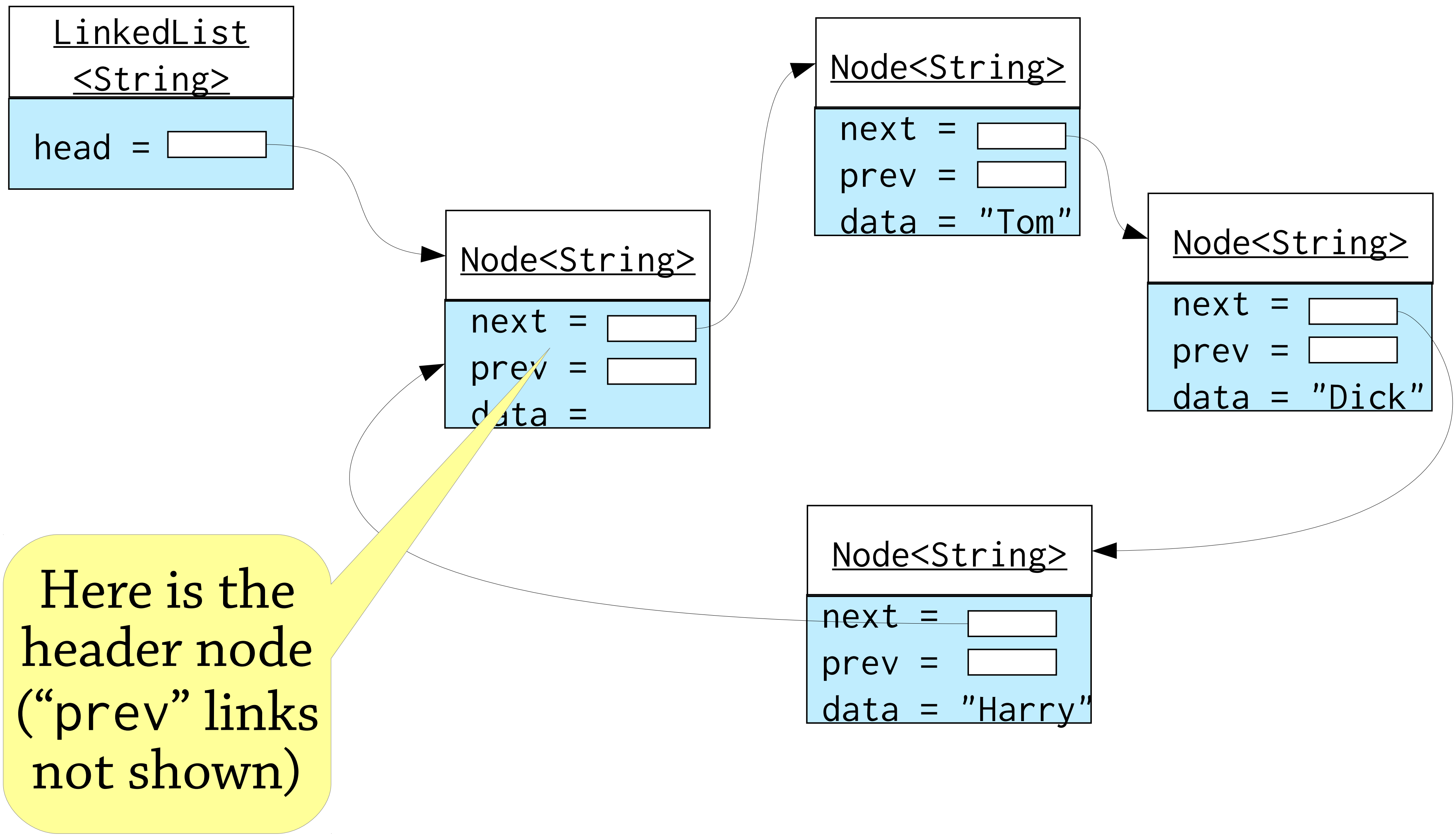
How can we get rid of these special cases?

One idea (see book): 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:

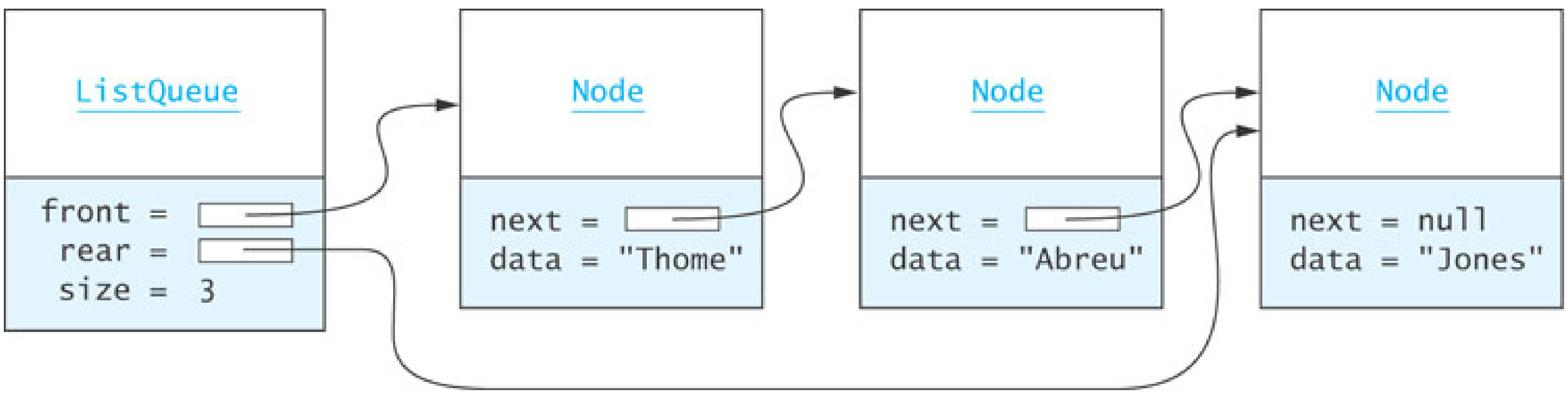
- push: add to front of list
- pop: remove from front of list

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

Hash tables
(19.1 – 19.3, 19.5 – 19.6)

Hash tables naïvely

A hash table implements a set or map

The plan: take an array of size k

Define a *hash function* that maps values to indices in the range {0,...,k-1}

- Example: if the values are integers, hash function might be $h(n) = n \bmod k$

To find, insert or remove a value x, put it in index $h(x)$ of the array

Hash tables naïvely, example

Implementing a set of integers, suppose we take a hash table of size 5 and a hash function $h(n) = n \bmod 5$

0	1	2	3	4
5		17	8	

This hash table contains {5, 8, 17}

Inserting 14 gives:

0	1	2	3	4
5		17	8	**14**

Similarly, if we wanted to find 8, we would look it up in index 3

A problem

This naïve idea doesn't work.
What if we want to insert 12 into the set?

0	1	2	3	4
5		17	8	

We should store 12 at index 2, but there's already something there!

This is called a *collision*

The problem with naïve hash tables

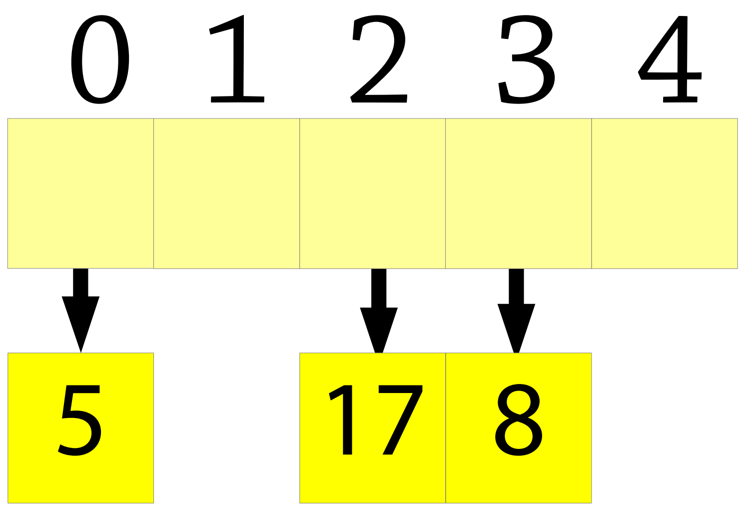
Naïve hash tables have two problems:

1. Sometimes two values have the same hash – this is called a *collision*
   - Two ways of avoiding collisions, *chaining* and *probing* – we will see them later
2. The hash function is specific to a particular size of array
   - Allow the hash function to return an arbitrary integer and then take it modulo the array size: *h(x) = x.hashCode() mod array.size*

Avoiding collisions: chaining

Instead of an array of elements, have an array of *linked lists*

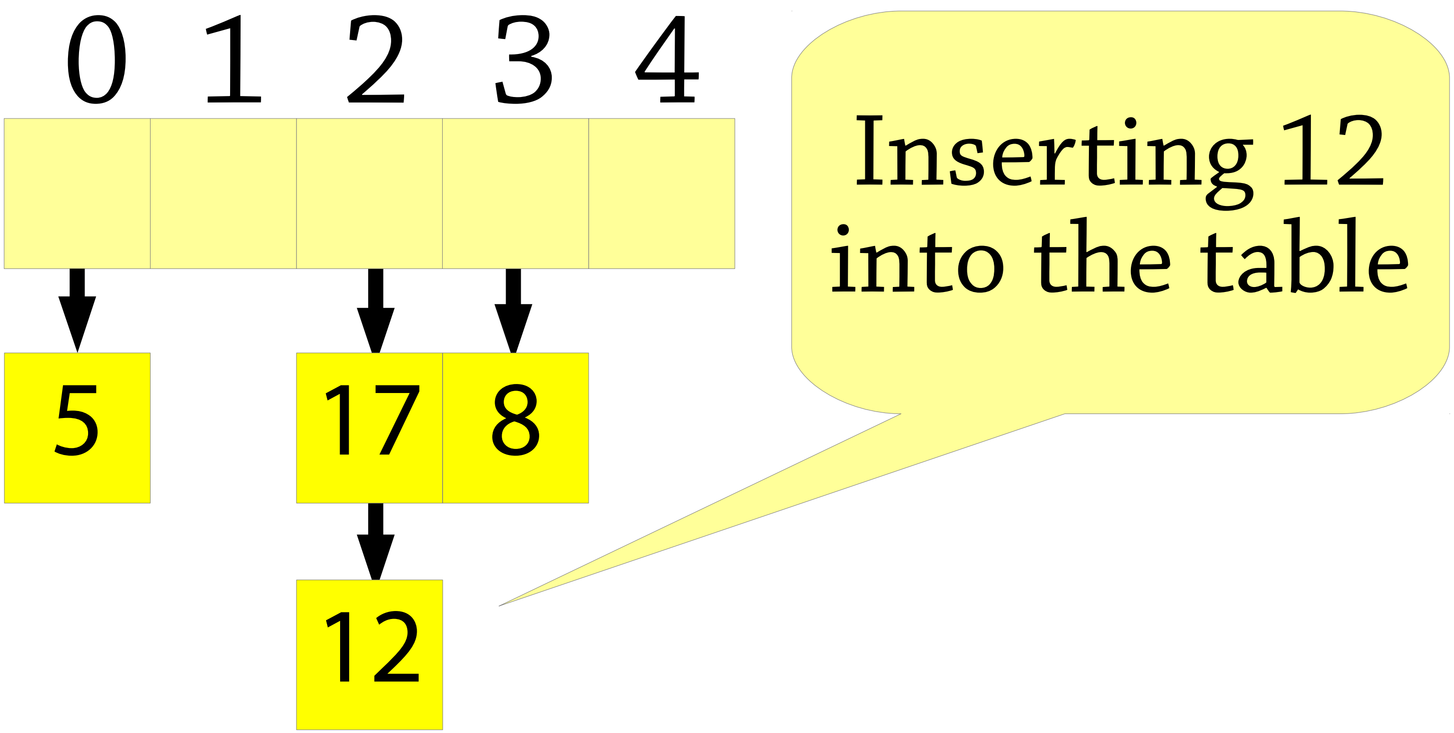
To add an element, calculate its hash and insert it into the list at that index



Avoiding collisions: chaining

Instead of an array of elements, have an array of *linked lists*

To add an element, calculate its hash and insert it into the list at that index



Performance of chained hash tables

If the linked lists are small, chained hash tables are fast

- If the size is bounded, operations are O(1) time

But if they get big, everything gets slow

Observation 1: the array must be big enough

- If the hash table gets too full (a high *load factor*), allocate a new array of about twice the size (*rehashing*)

Observation 2: the hash function must *evenly distribute* the elements!

- If everything has the same hash code, all operations are O(n)

Defining a good hash function

What is wrong with the following hash function on strings?

Add together the character code of each character in the string
(character code of a = 97, b = 98, c = 99 etc.)

Maps e.g. *bass* and *bart* to the same hash code! (s + s = r + t)

Similar strings will be mapped to nearby hash codes – does not distribute strings evenly

A hash function on strings

An idea: map strings to integers as follows:

$$s_0 \cdot 128^{n-1} + s_1 \cdot 128^{n-2} + \ldots + s_{n-1}$$

where s_i is the code of the character at index i

If all characters are ASCII (character code 0 – 127), each string is mapped to a different integer!

The problem

In many languages, when calculating

$s_0 \cdot 128^{n\text{-}1} + s_1 \cdot 128^{n\text{-}2} + \ldots + s_{n\text{-}1}$,

the calculation happens modulo 2^{32} (*integer overflow*)

So the hash will only use the last few characters!

Solution: replace 128 with 37

$s_0 \cdot 37^{n\text{-}1} + s_1 \cdot 37^{n\text{-}2} + \ldots + s_{n\text{-}1}$

Use a *prime number* to get a good distribution

This is what Java uses for strings

Hashing a pair

```
class C { A a; B b; }
```

One way: multiply the two hash codes by different prime numbers and add the results, then add a constant:

```
int hashCode() {
    return 31 * a.hashCode() +
           37 * b.hashCode() + 1;
}
```

Hash functions

A good hash function must distribute elements evenly to avoid collisions

Defining really good hash functions is a black art – but the two techniques above give you decent hash functions

Last trick: make the hash table size a prime number – this helps mask patterns in the hash function

- e.g., if the hash function always returns an even number, if the array size is a power of two then all the odd indexes will be empty

Linear probing

Another way of dealing with collisions is *linear probing*

Uses an array of values, like in the naïve hash table

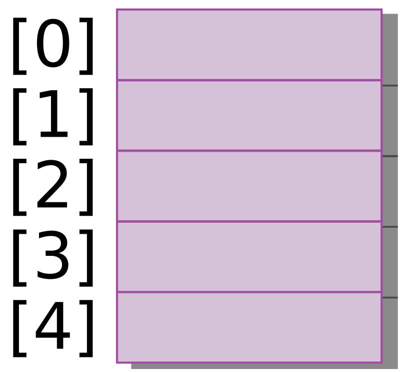
If you want to store a value at index i but it's full, store it in index $i+1$ instead!

If that's full, try $i+2$, and so on

…if you get to the end of the array, wrap around to 0

Exempel: Insättning

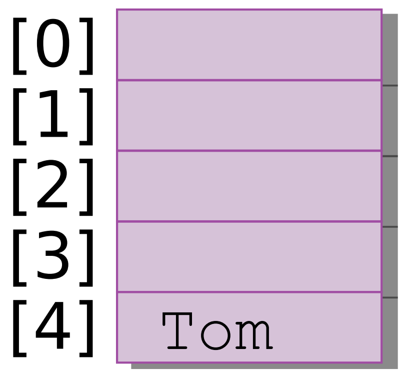
`Tom Dick Harry Sam Pete`



Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

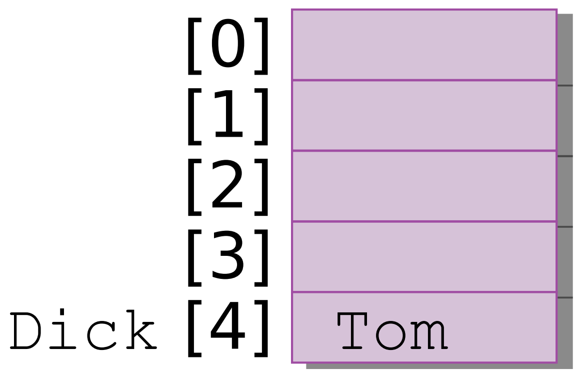
`Dick Harry Sam Pete`



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

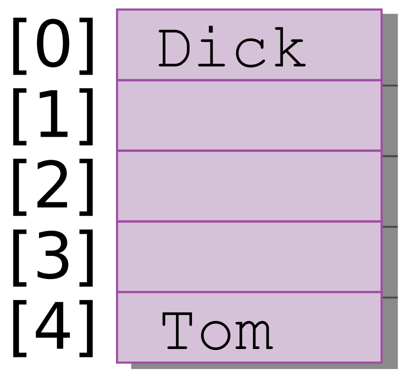
Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Harry Sam Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

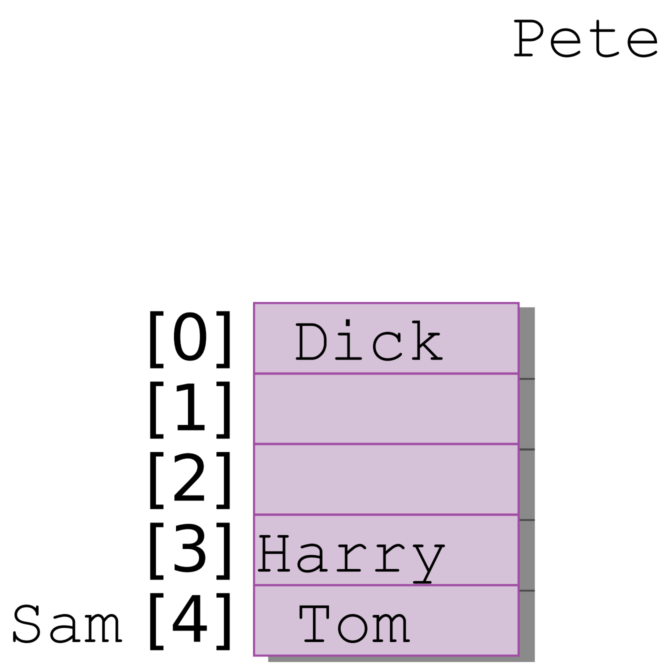
Exempel: Insättning

Sam Pete

[0]	Dick
[1]	
[2]	
[3]	Harry
[4]	Tom

Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

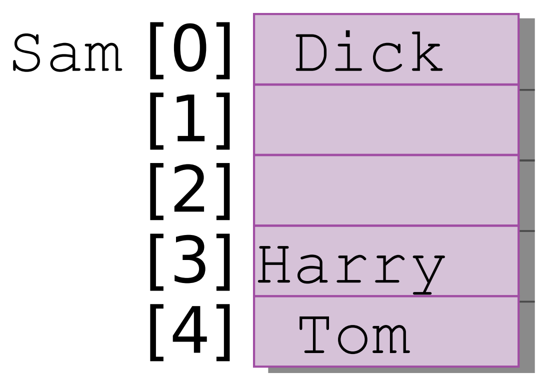
Exempel: Insättning



Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning

Pete



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

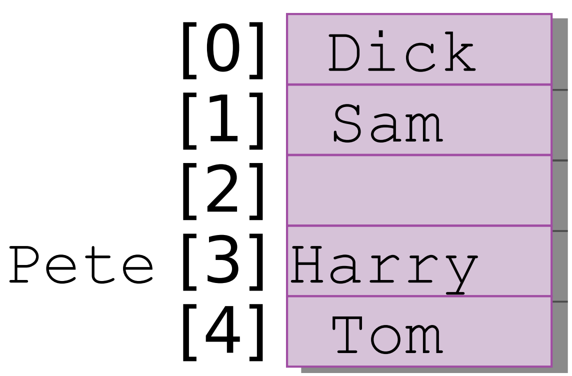
Exempel: Insättning

Pete

[0]	Dick
[1]	Sam
[2]	
[3]	Harry
[4]	Tom

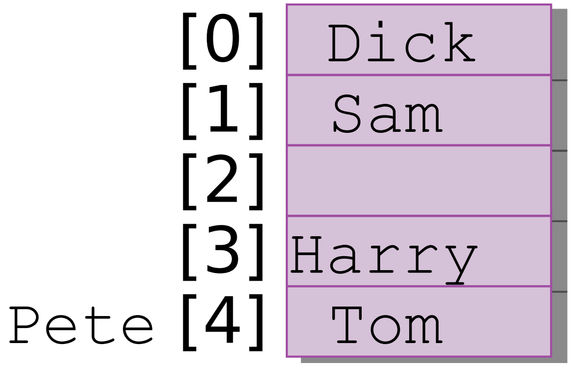
Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning



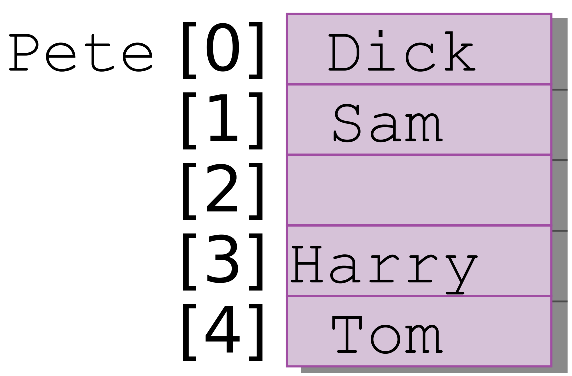
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning



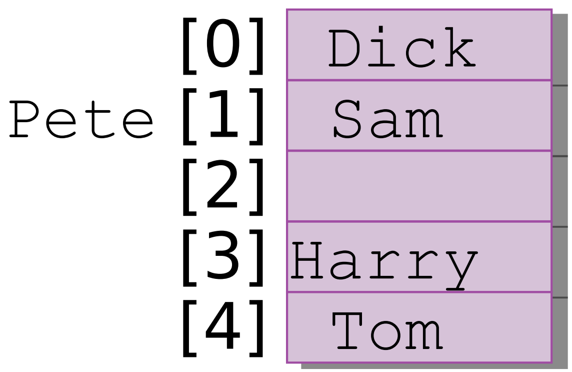
Name	**hashCode()**	**hashCode() %5**
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning



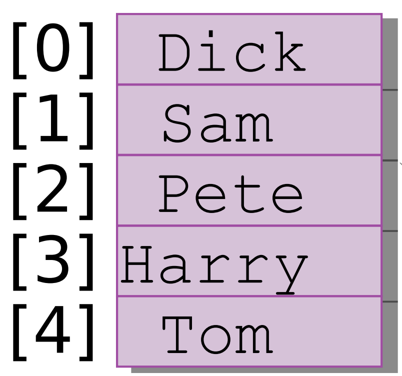
Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning



Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Exempel: Insättning



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

To find "Pete" (hash 3), you must start at index 3 and work your way all the way around to index 2

Searching with linear probing

To find an element under linear probing:

- Calculate the hash of the element, *i*
- Look at *array[i]*
- If it's the right element, return it!
- If there's no element there, fail
- If there's a *different* element there, search again at index *(i+1) % array.size*

We call a group of adjacent non-empty indices a *cluster*

Deleting with linear probing

Can't just remove the element...

Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

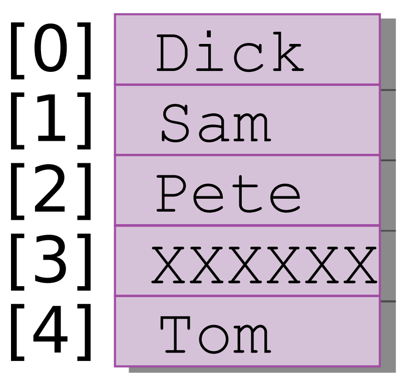
[0]	Dick
[1]	Sam
[2]	Pete
[3]	Harry
[4]	Tom

If we remove Harry,
Pete will be in the wrong cluster
and we won't be able to find him

Deleting with linear probing

Instead, mark it as deleted (*lazy deletion*)

Name	hashCode()	hashCode() %5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



The search algorithm will skip over XXXXXX

Deleting with linear probing

It's useful to think of the invariant here:

- Linear *chaining*: each element is found at the index given by its hash code
- Linear *probing*: each element is found at the index given by its hash code, *or a later index in the same cluster*

Naïve deletion will split a cluster in two, which may break the invariant

Hence the need for an empty value that does not mark the end of a cluster

Linear probing performance

To insert or find an element under linear probing, you might have to look through a whole cluster of elements

Performance depends on the size of these clusters:

- Small clusters – expected O(1) performance
- Almost-full array – O(n) performance
- If the array is full, you can't insert anything!

Thus you need:

- to expand the array and *rehash* when it starts getting full
- a hash function that distributes elements evenly

Same situation as with linear chaining!

Linear probing vs linear chaining

In linear chaining, if you insert several elements with the same hash *i*, those elements become slower to find

In linear probing, elements with hash *i+1*, *i+2*, etc., will belong to the same cluster as element *i*, and will also get slower to find

If the load factor is too high, this tends to result in very long clusters in the hash table – a phenomenon called *primary clustering*

Probing vs chaining

Linear probing is more sensitive to high load

On the other hand, linear probing uses less memory for a given load factor, so you can use a bigger array than you would with chaining

load factor (#elements / array size)	**#comparisons (linear probing)**	**#comparisons (linear chaining)**
0 %	1.00	1.00
25 %	1.17	1.13
50 %	1.50	1.25
75 %	**2.50**	1.38
85 %	3.83	1.43
90 %	5.50	1.45
95 %	10.50	1.48
100 %	—	1.50
200 %	—	2.00
300 %	—	**2.50**

Summary of hash table design

Several details to consider:

- *Rehashing*: resize the array when the load factor is too high
- *A good hash function*: need an even distribution
- *Collisions*: either chaining or probing

Hash tables have *expected* (average) O(1) performance if the hash function is random (there are no patterns) – but it's normally not!

Nevertheless, performance is O(1) in practice with decent hash functions.

So – theoretical foundations a little shaky, but very good practical performance.

Hash tables versus BSTs

Hash tables: O(1) performance in practice (O(n) if very unlucky), BSTs: O(log n) if balanced

Hash tables are *unordered*: you can't e.g. get the elements in increasing order

But they are normally *faster* than balanced BSTs, despite the theoretical O(n) worst case