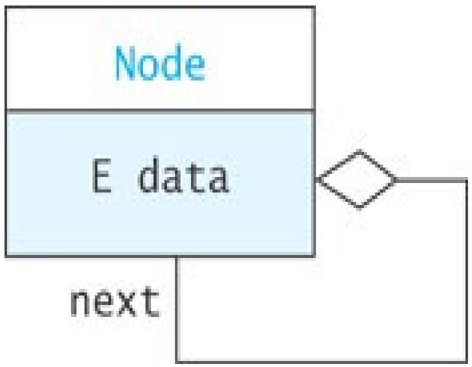
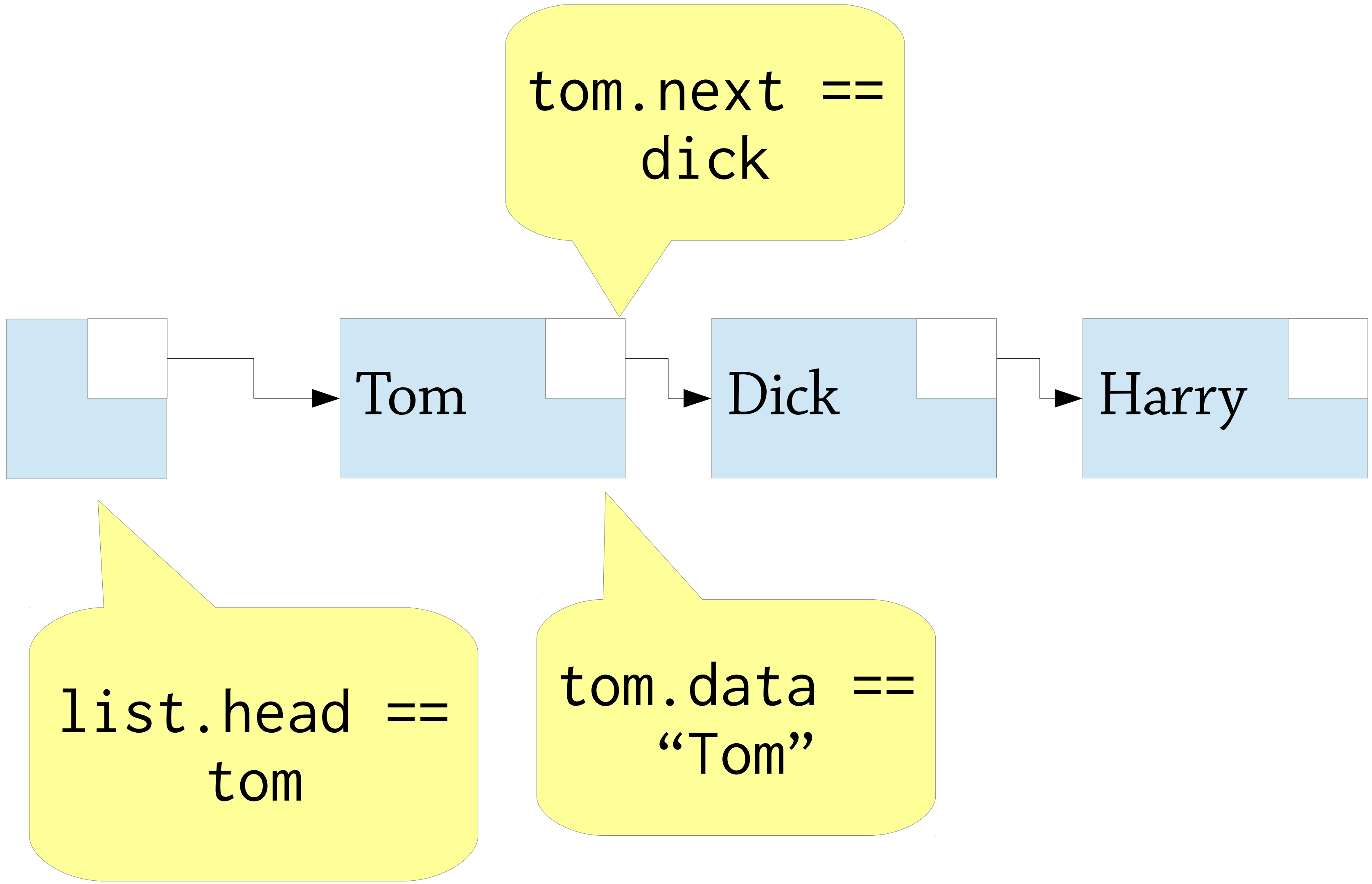
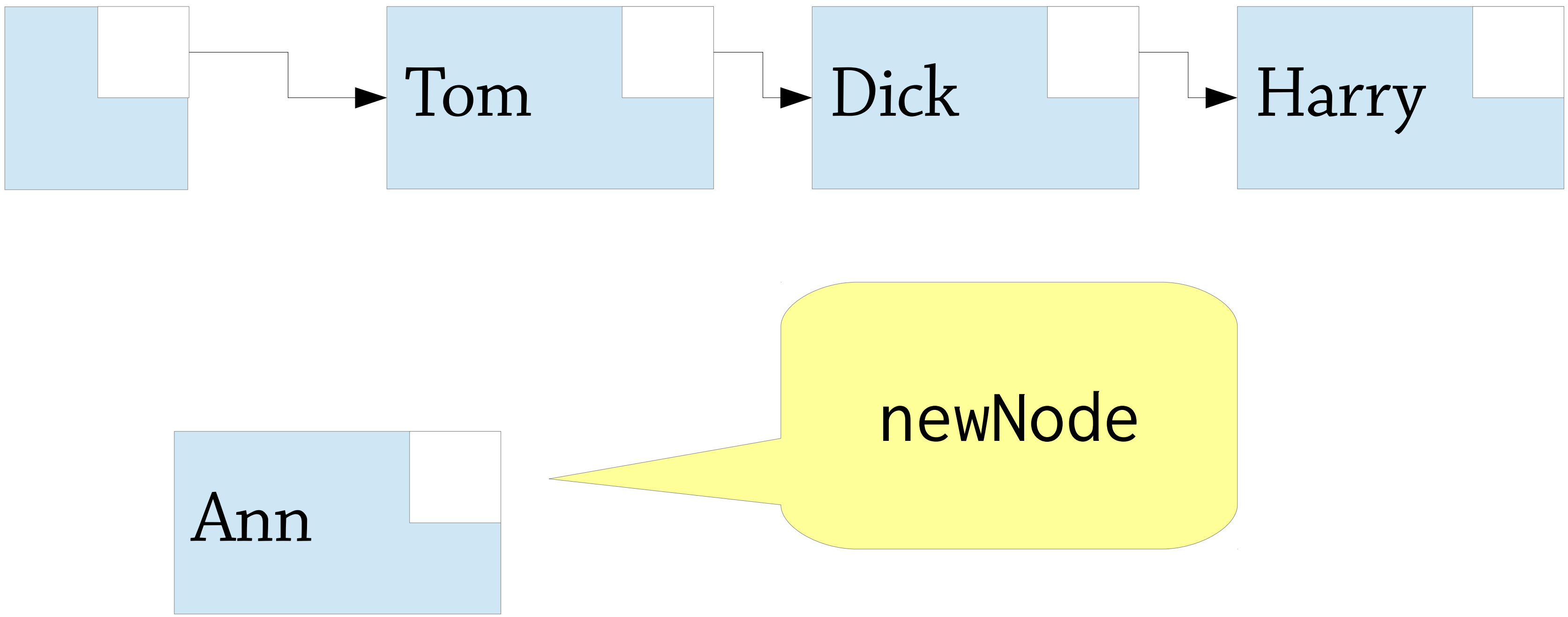
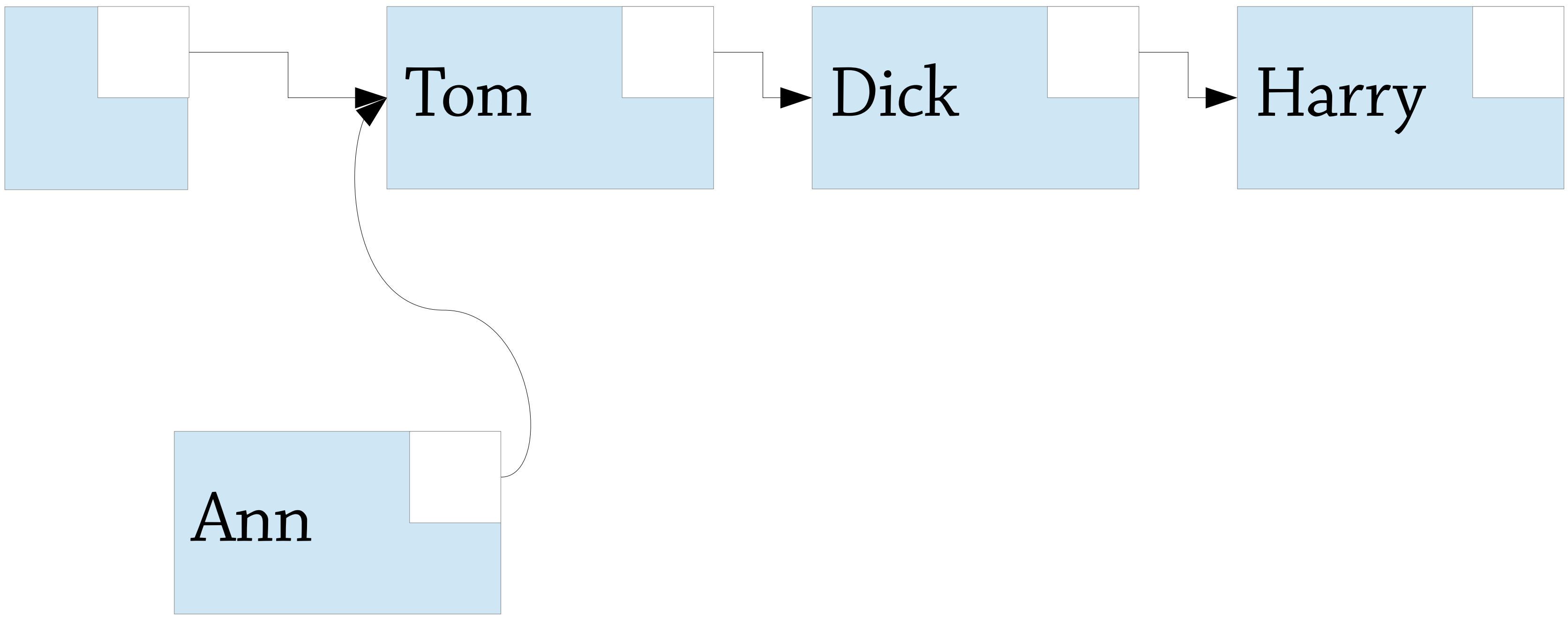
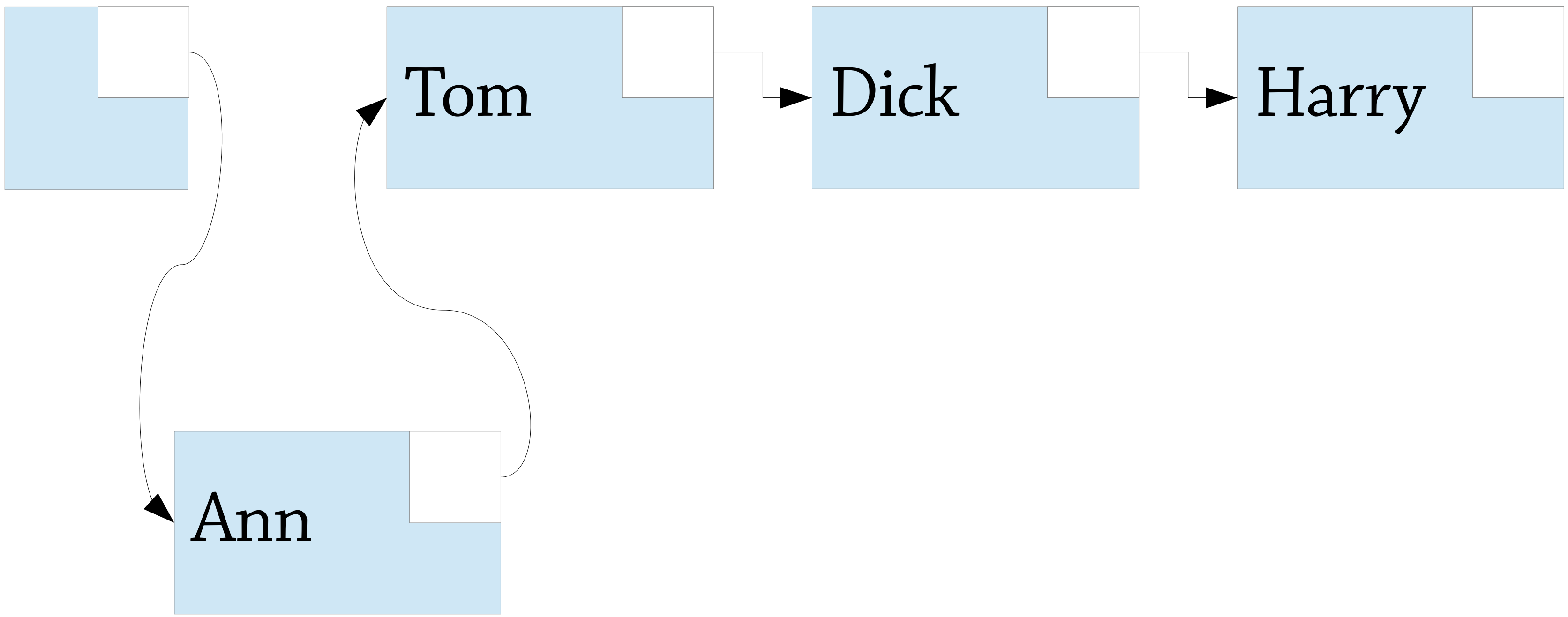
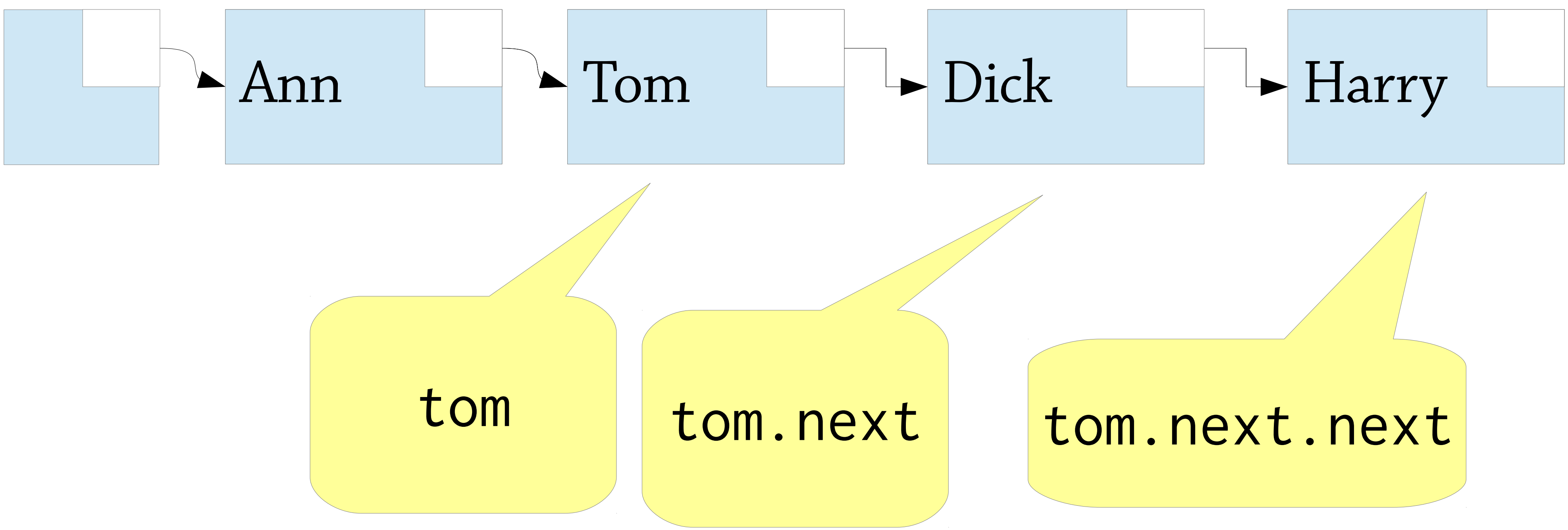
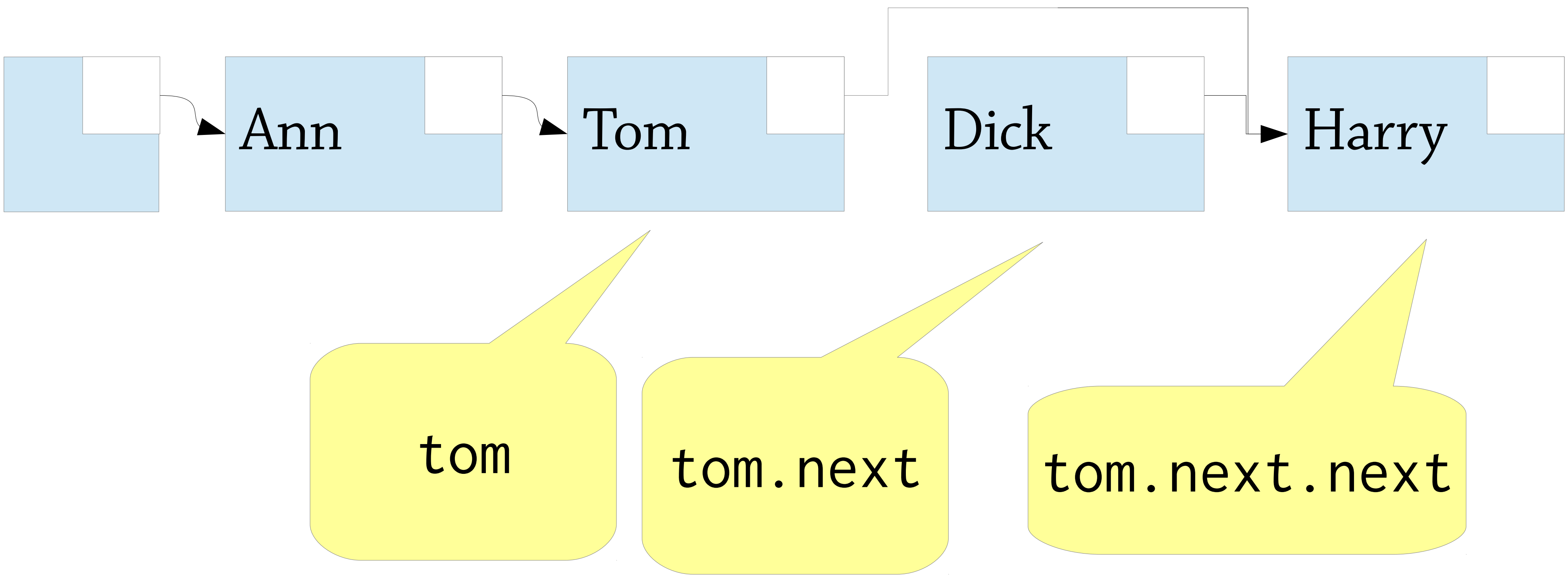
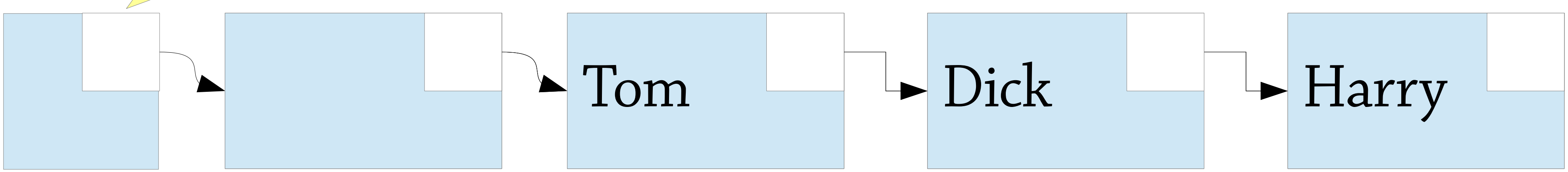
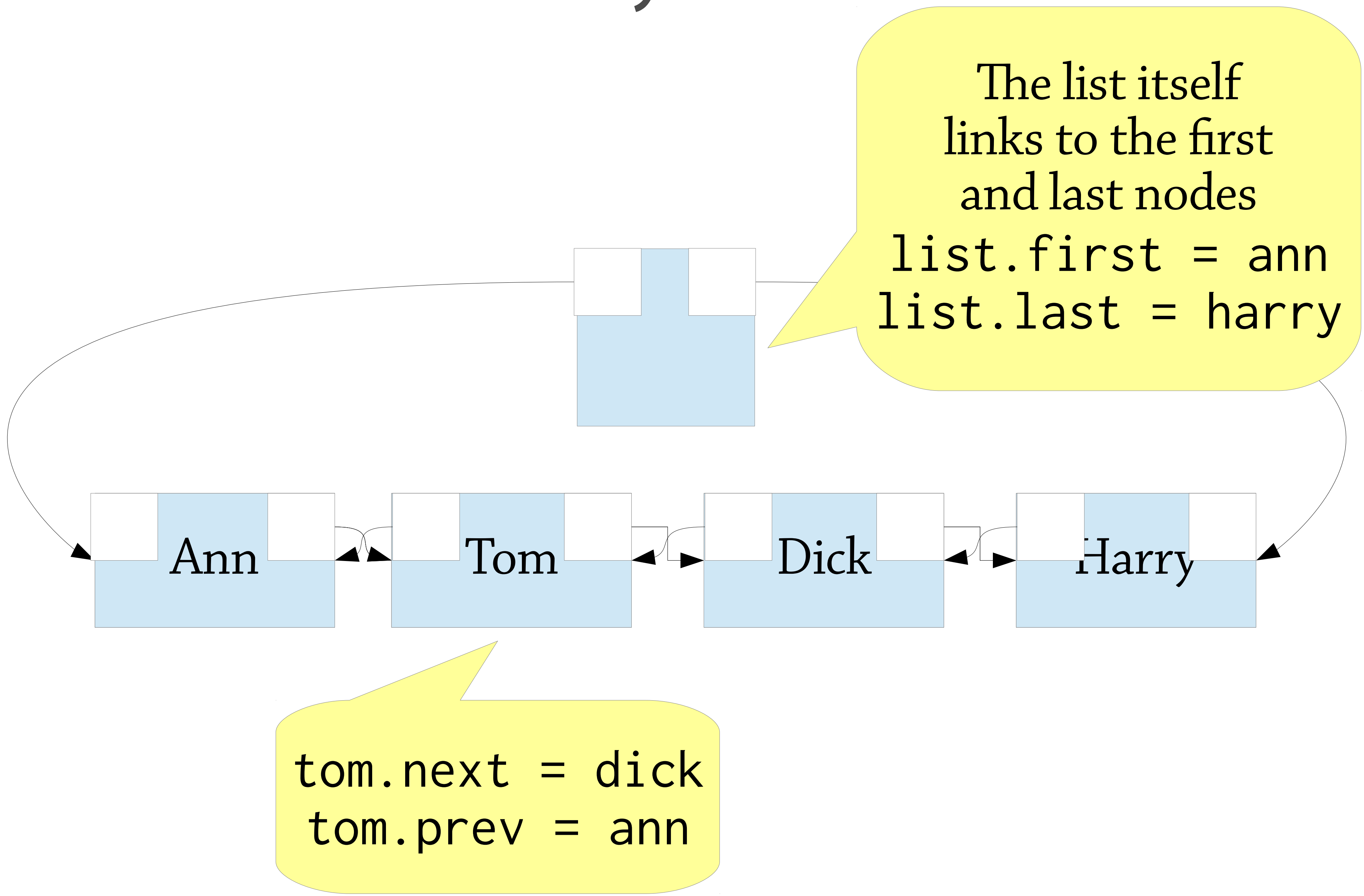
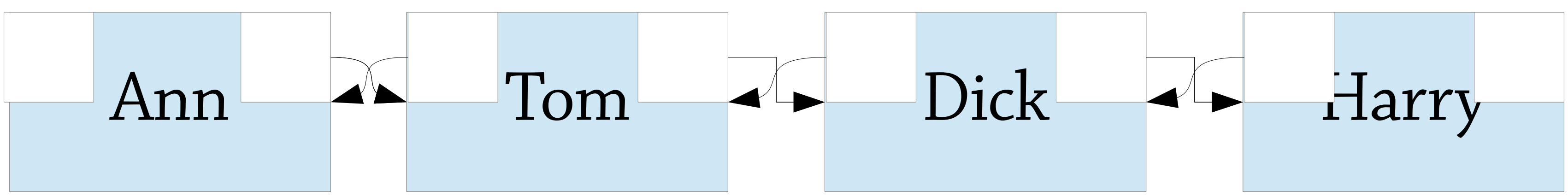
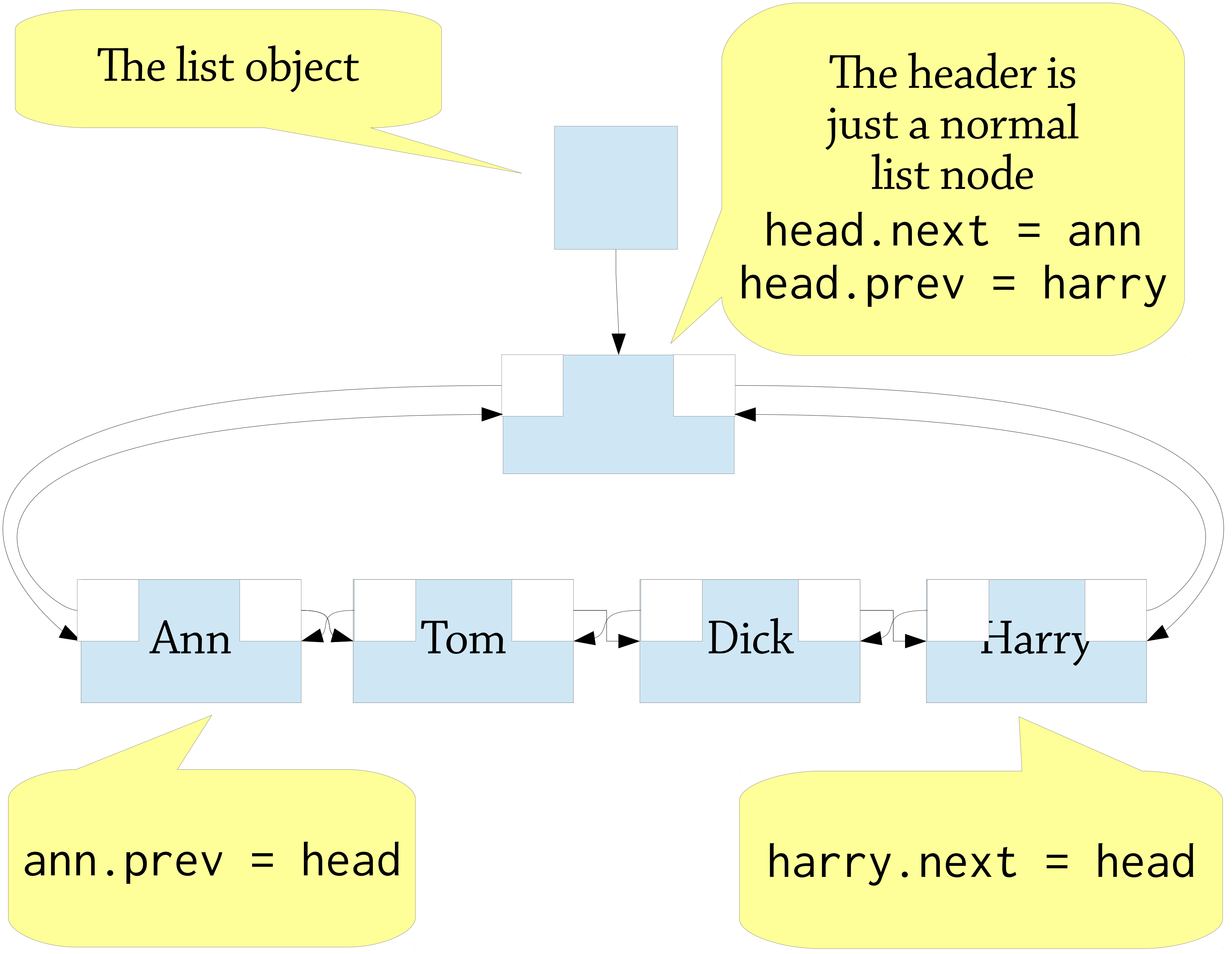
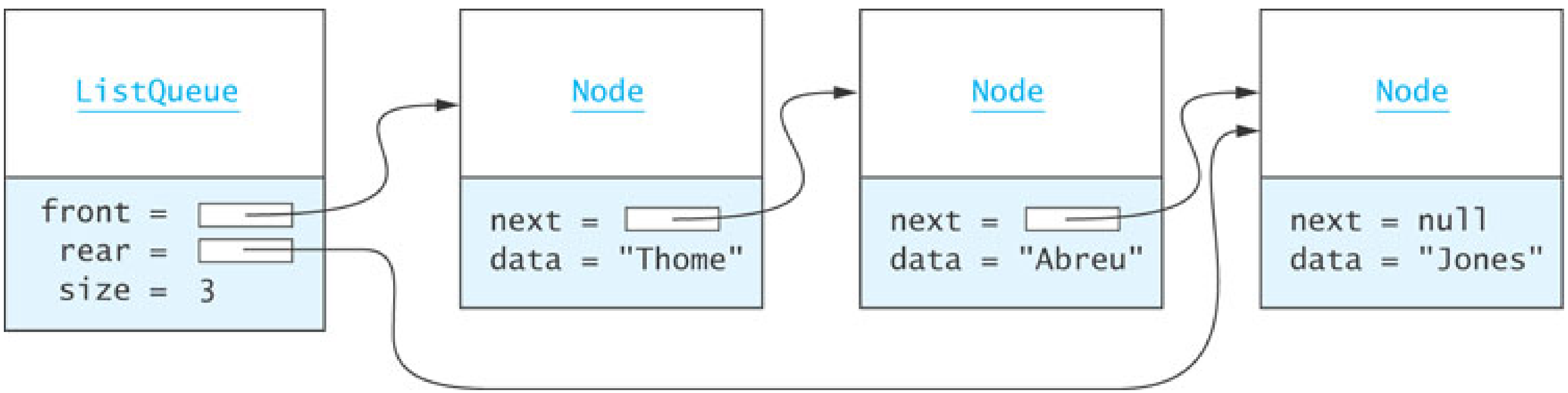
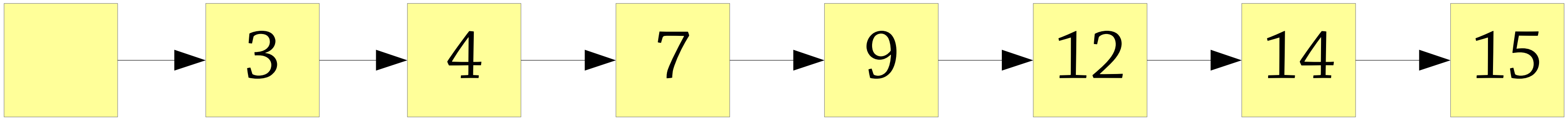
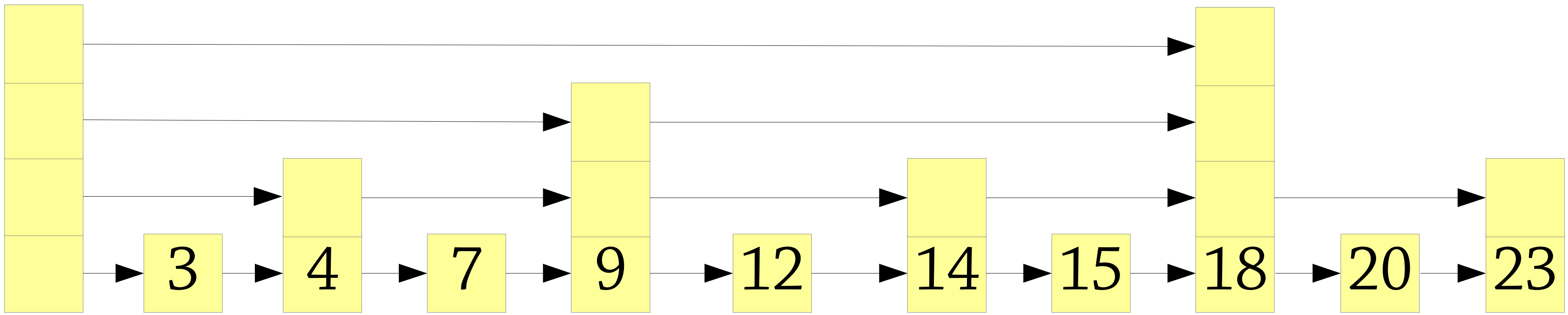
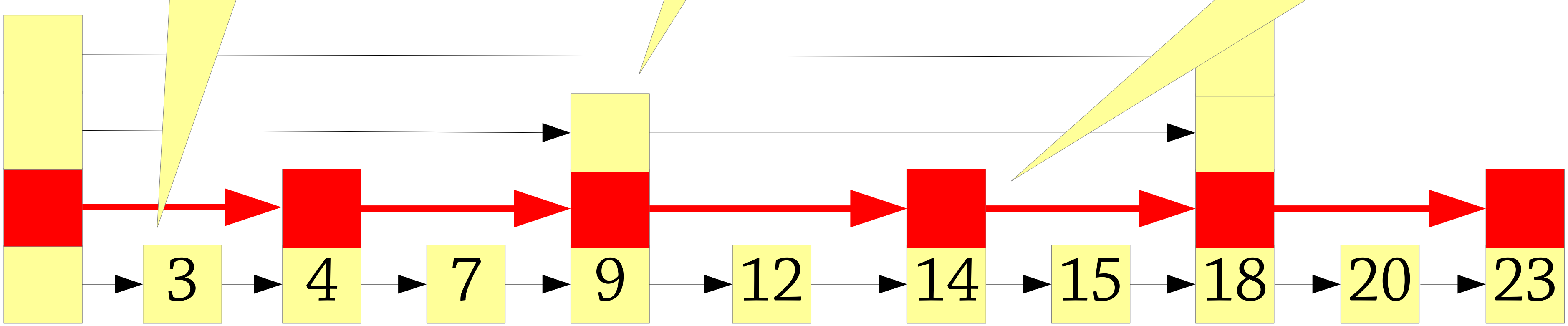
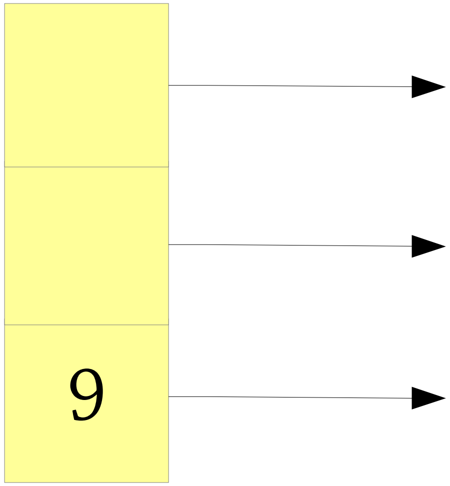
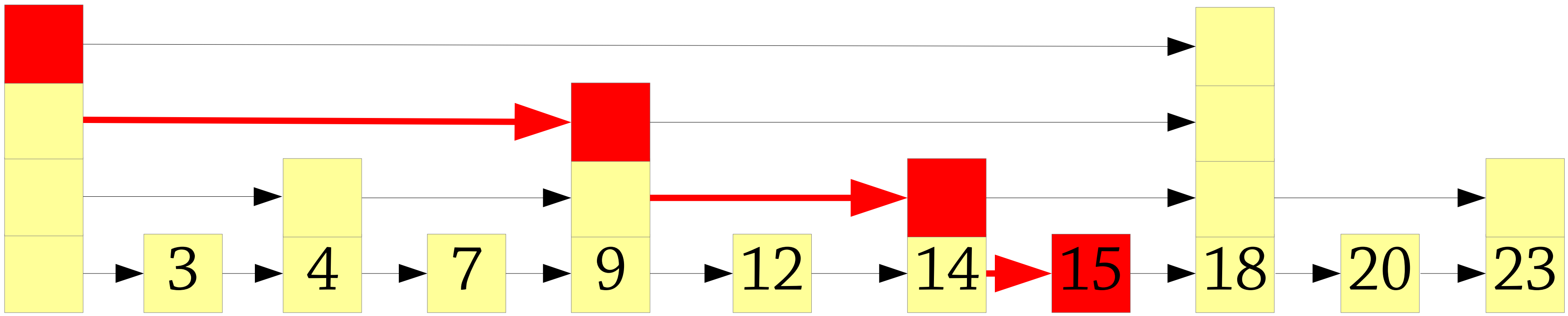
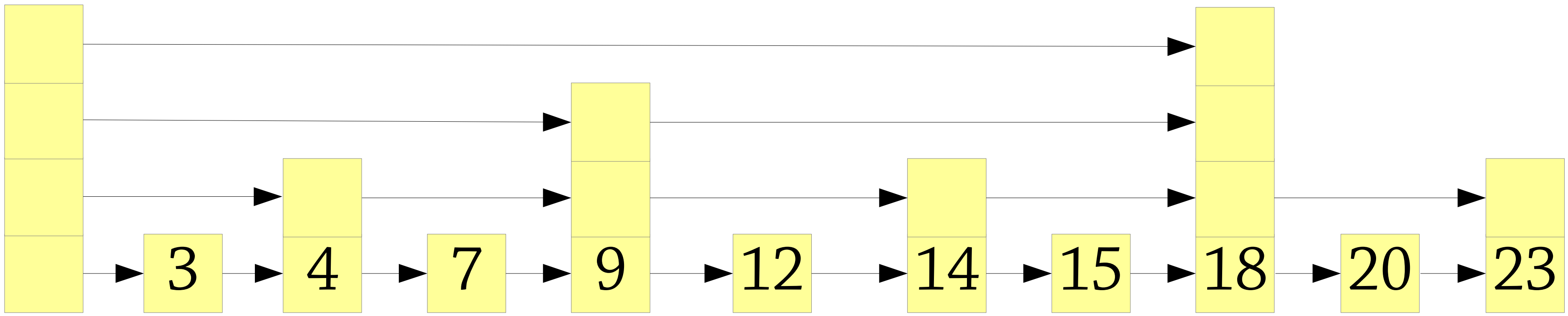
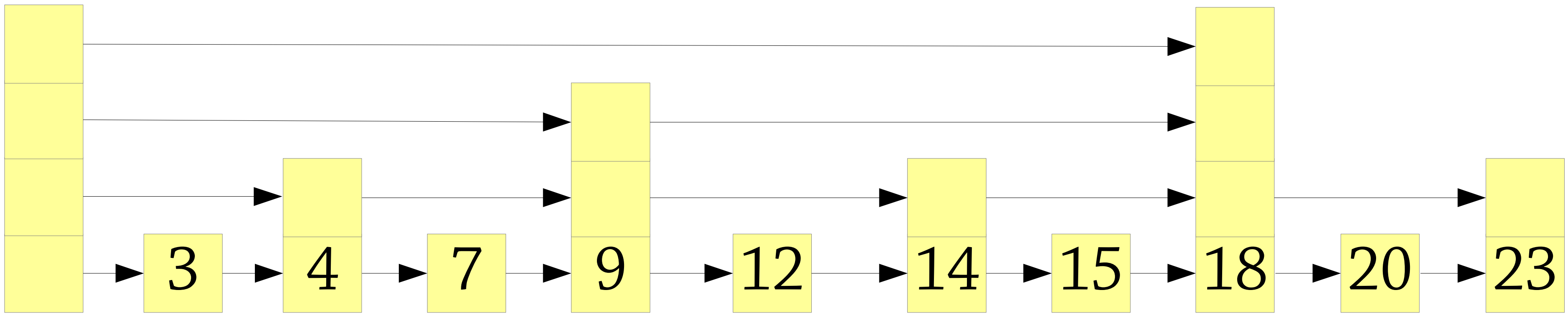
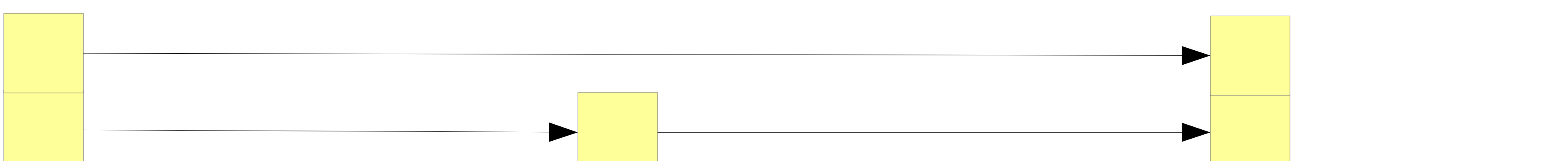
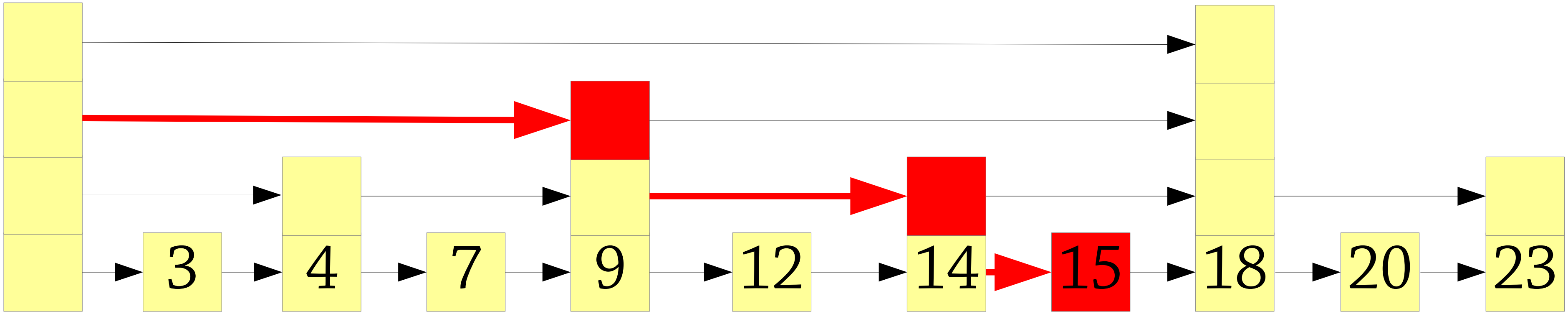
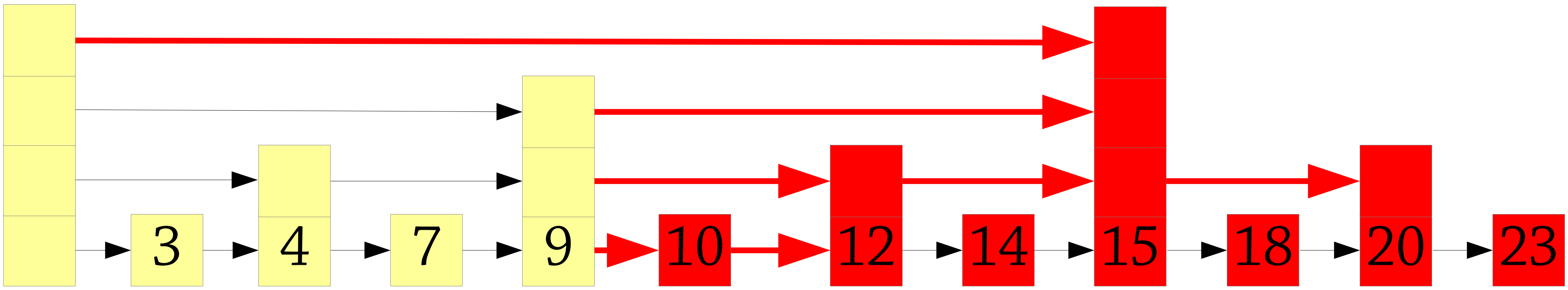
Iterators in Java

java.util.Iterator

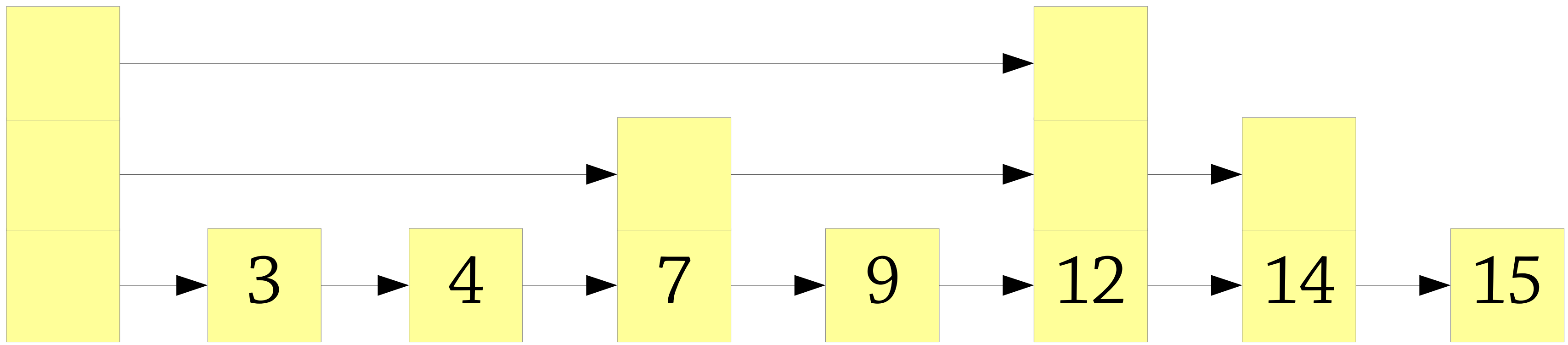
- `Iterator<E>` is an interface which provides a uniform way to enumerate all elements in a `Collection<E>`, e.g. a `List<E>` or a `Set<E>`.
- Minimal implementation is:
  - `boolean hasNext()` – is there another element?
  - `E next()` – give me next element
- There is also an optional method for removing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

Linked lists

Linked lists

Linked lists are a data structure designed for *sequential access* to a list

- Move forwards (and backwards) through the list, one element at a time
- Read or write the element at the current position
- Insert or delete elements at the current position
- all in O(1) time

The downside: getting to a specific position in the list takes O(n) time

- Linked lists are bad for *random access*

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;
}
```



The list itself is a reference to the first node:

```
class List<E> {
  Node<E> head;
}
```

Singly-linked lists

The list [Tom, Dick, Harry] as a linked list:



Modifying a linked list

```
// 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)
```

Calling list.addFirst(“Ann”)



First create a new list node

Calling list.addFirst("Ann")



Then set `newNode.next = list.head`

Calling list.addFirst("Ann")



Then set `list.head = newNode`

Done!

`addAfter` is very similar

Calling list.deleteAfter(tom)



To remove `tom.next` from the list,
set `tom.next = tom.next.next`

Calling list.deleteAfter(tom)



Done!

`deleteFirst` is very similar

Header nodes

It's not good to have *two* versions of each list operation (e.g. `addFirst` vs `addAfter`):

- The API gets twice as big
- Code using the list library will need special cases when it modifies the front of the list
- Twice as much code to write

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

We can get rid of `addFirst(x)` and do `addAfter(headerNode, x)` instead

List with header node

nn” before “Tom”, we can
, “Ann”)

We could even get rid of this list object now



The header node!

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
- This also means we can't delete the current node (would need to update its predecessor's `next` field)

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 Tom in the list below:

```
dick.prev = ann;
ann.next = dick;
```



In general we can do:

```
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 there are lots of special cases!

- What if the node is the first node?
  This code crashes, since `node.prev == null`
  We also need to update `list.first`
- 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?

Solution: circular linked list!


The list object
The header is
just a normal
list node
head.next = ann
head.prev = harry
Ann
Tom
Dick
Harry
ann.prev = head
harry.next = head

Circularly-linked list with header node

An extra header node, "in between" the first and last elements in the list

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`

No special cases in insertion or deletion!

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

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 (disadvantage: more memory use)

Compared to dynamic arrays:

- *random access* takes O(n) instead of O(1) time
- insert/delete are O(1) – once you find the node
- worse constant factors
  (extra memory needed for list nodes, cache-unfriendly)

Comparing dynamic array and doubly linked list as list implementations

operation	Dynamic array	Doubly linked list
add(x)	O(1) amortized	O(1)
add(x,i)	O(n)	O(n)
remove(x)	O(n)	O(n)
remove(i)	O(n)	O(n)
get(i)	O(1)	O(n)
set(i,x)	O(1)	O(n)
contains(x)	O(n)	O(n)
size	O(1)	O(1)
iterator	O(1)	O(1)
hasNext/next	O(1)	O(1)
iter.remove	O(n)	O(1)

Skip lists

Linked lists are bad at random access

We can use a sorted linked list to implement a set:



But finding an element takes O(n) time

Notice it is only *finding the right place in the list* that's slow

- Once you've found the right place to insert/delete, you can modify the list in O(1) time

Basic skip lists

The idea of skip lists: take a linked list and give some nodes *extra* forward links which skip further ahead in the list

- Each node has a *level* – e.g. a level 3 node has 3 forward links
- Each level skips further forward than the level before
- The bottom level lets you go through the list one by one as in a normal linked list

Can view this as several linked lists, which skip through different amounts of the whole list



Basic skip lists

The idea of skip lists: take[…] give some nod[…]d links[…]er ahead in the lis[…]

- […]- e.g. a […]vard links
- […]er forward[…] the level befo[…]
- […]you go th[…]gh the list one by[…]l lin[…]st

Can […]v this as several l[…]ked lists, which […] diffe[…]t amounts of th[…] whole list



Skip list nodes

A node in a skip list has some data and an array of forward links:

```
class SkipNode<E> {
  E data;
  SkipNode<E> links[];
}
```

The level is the size of this array



Basic skip lists

We can find things efficiently in the skip list by using the extra levels to “skip ahead”

- Start at the highest level of the list
- Go right as far as you can without going past the node you're looking for
- Then repeat the process one level down

e.g. finding 15:



Naive skip lists

How many levels should we have?
And what level should each node have?

In *naive skip lists*:

- the level 1 list contains all nodes
- the level 2 list contains every second node
- the level 3 list contains every fourth node
- each level skips twice as many nodes as the level before



Naive skip lists

Formally, between any two nodes of level ≥ n+1, there is a node of level n

- Between all level ≥ 2 nodes there is a level 1 node



Naive skip lists

Formally, between any two nodes of level ≥ n+1, there is a node of level n

- Between all level ≥ 2 nodes there is a level 1 node
- Between all level ≥ 3 nodes there is a level 2 node

Naive skip lists

Formally, between any two nodes of level ≥ n+1, there is a node of level n

- Between all level ≥ 2 nodes there is a level 1 node
- Between all level ≥ 3 nodes there is a level 2 node
- Between all level ≥ 4 nodes there is a level 3 node



Naive skip lists

Why arrange the nodes like this?

Because, when searching in the list...

- The highest level skips through half the list
- The next level skips through a quarter
- and so on...

so search takes O(log n) time! (Compare to binary search.)



Naive skip lists

But updating a naive skip list takes O(n) time! For example, here we have inserted 10, and the parts of the list that changed are highlighted in red...



Naive skip lists – the invariant

Each node in the skip list has a *level*

- Level 1 contains every element of the skip list
- Level 2 contains every 2^{nd} element
- Level 3 contains every 4^{th} element
- Level k contains every 2^{k-1}th element

We can search in O(log n) time

But insertion/delete takes O(n) time

- Have to update too much of the list

Probabilistic skip lists

The solution: *probabilistic* skip lists!

- Level 1 contains every element of the skip list
- Level 2 contains **roughly ½** of the elements
- Level 3 contains **roughly ¼** of the elements
- Level k contains **roughly $1/2^{k-1}$** of the elements

On insertion, we choose the level of the new node *at random*, maintaining the distribution above

- ```
  level = 1;
  while (coin flip gives heads) level = level + 1;
  ```
  ```

Probabilistic skip lists

Here is how a probabilistic skip list might look:

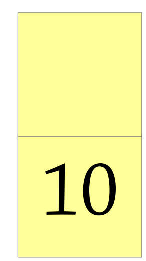


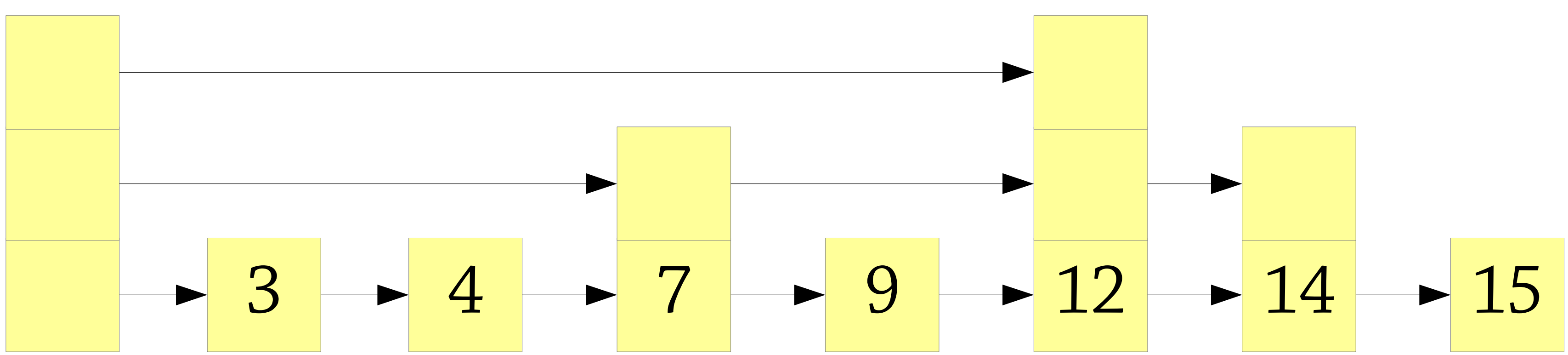
Probabilistic skip lists

Inserting 10. First choose the level:

- Level 1: yes
- Level 2: coin flip, heads, yes
- Level 3: coin flip, tails, no

We make it a level 2 node:

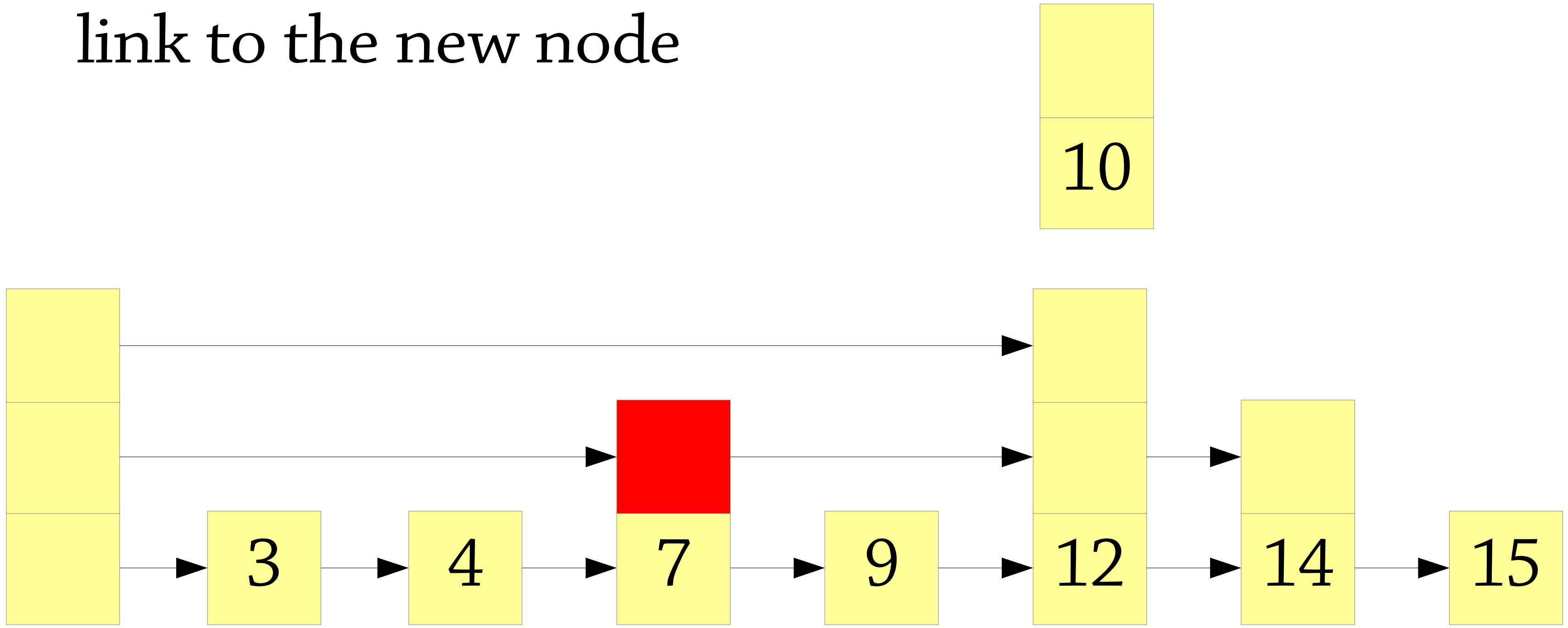




Probabilistic skip lists

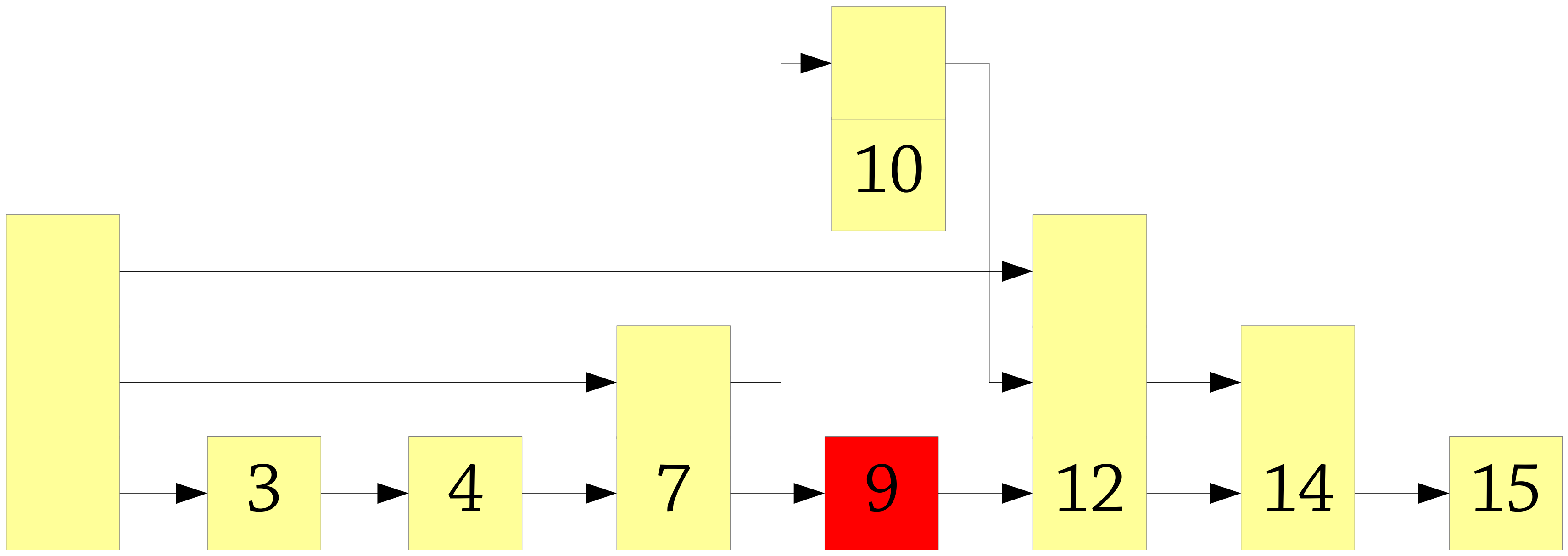
Next step: find the predecessor level 2 node (the greatest level 2 node that's less than the new node)

- This node should have a link to the new node



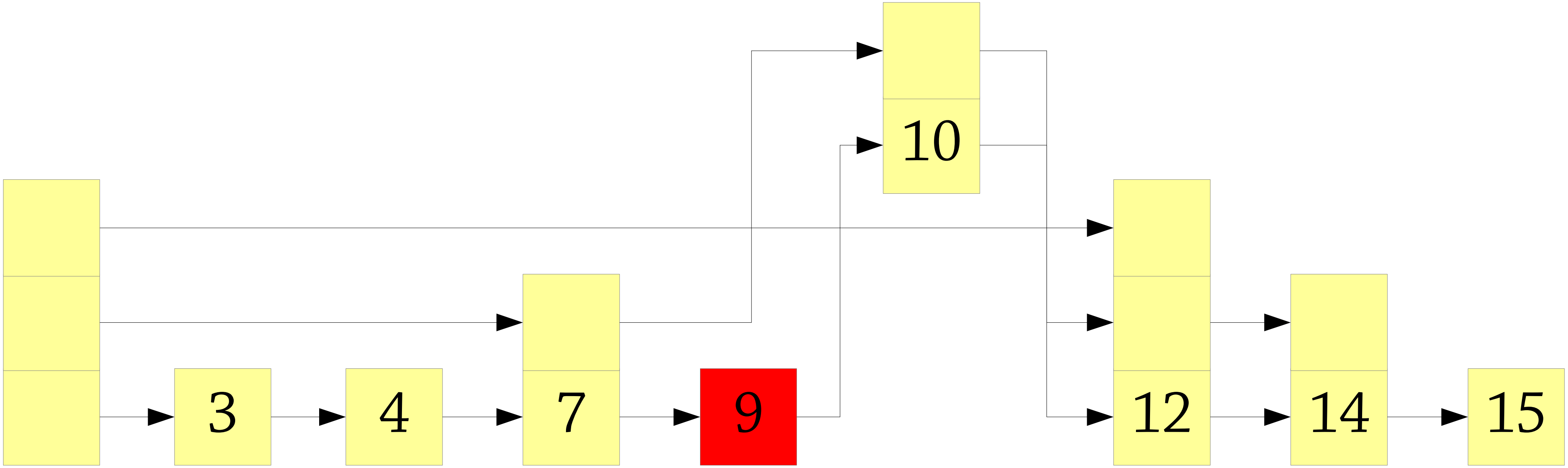
Probabilistic skip lists

Now we insert the new node into the level 2 list, go down to level 1 and repeat the process



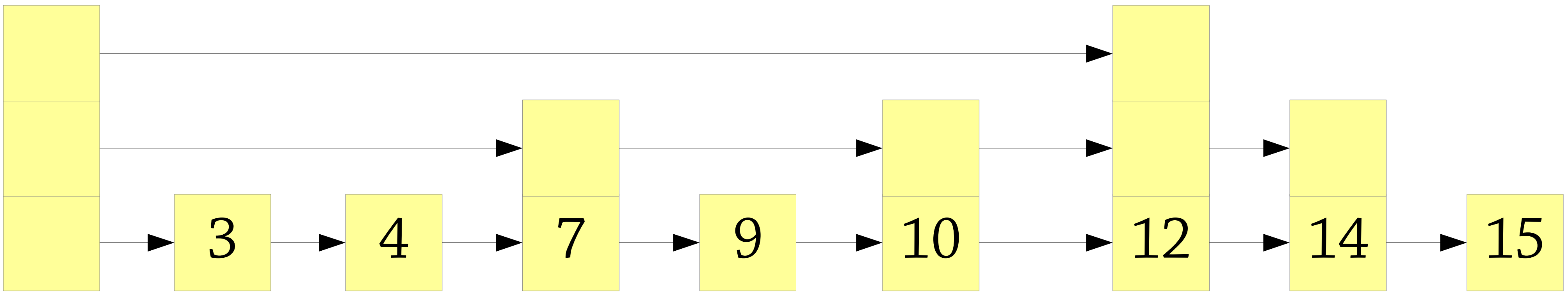
Probabilistic skip lists

Now we insert the node into the level 1 list, and we're finished



Probabilistic skip lists

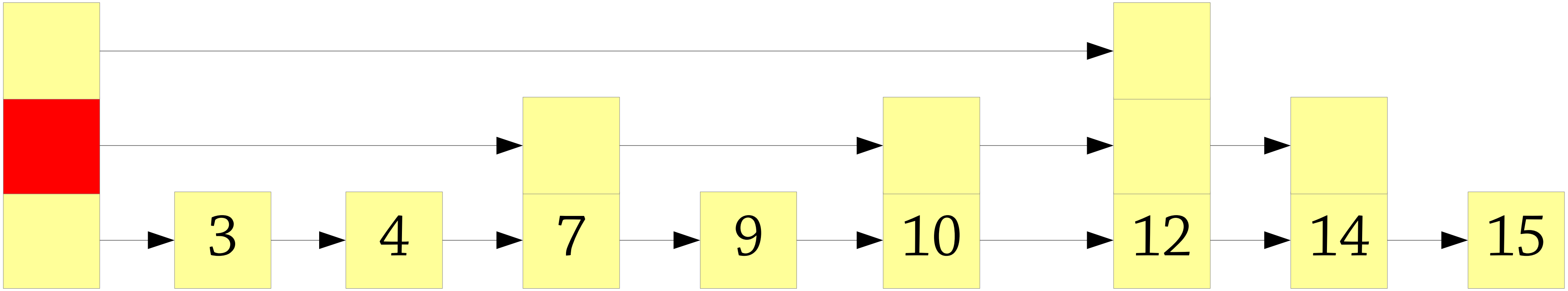
Done!



Probabilistic skip lists

Deletion: simply remove the node from the list – e.g., deleting 7, a level 2 node:

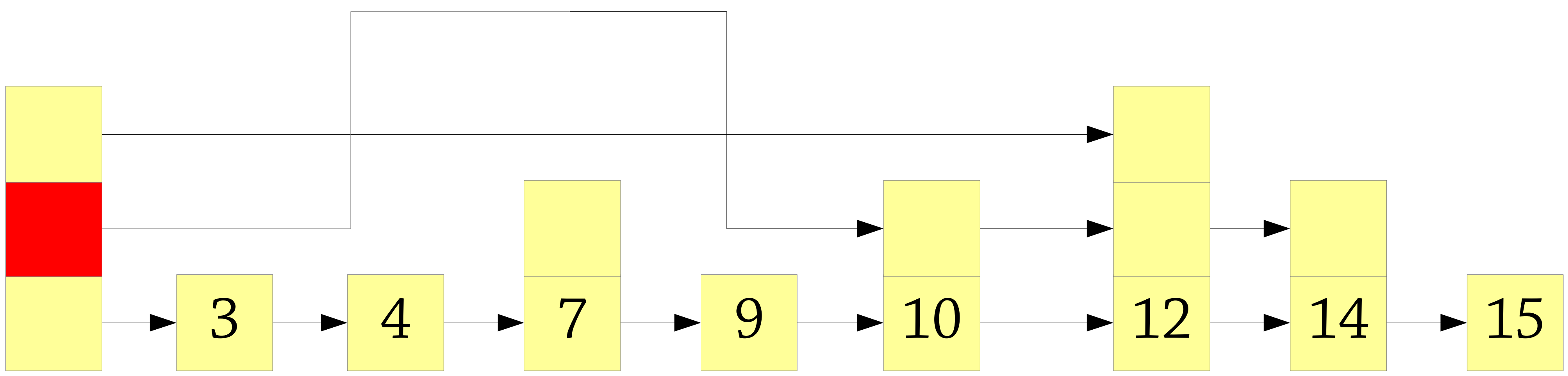
- Find level 2 predecessor



Probabilistic skip lists

Deletion: simply remove the node from the list – e.g., deleting 7, a level 2 node:

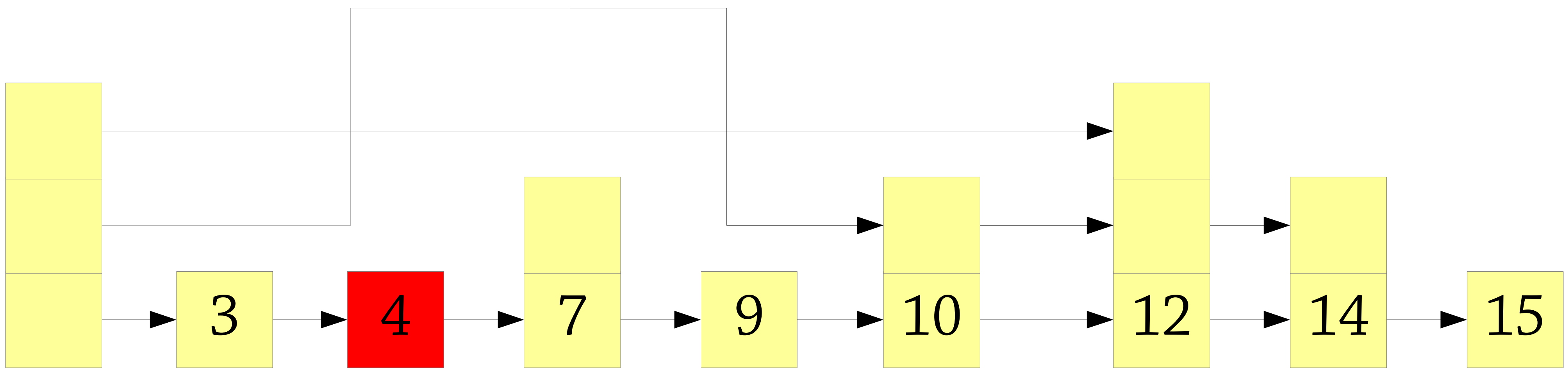
- Find level 2 predecessor
- Remove node from level 2



Probabilistic skip lists

Deletion: simply remove the node from the list – e.g., deleting 7, a level 2 node:

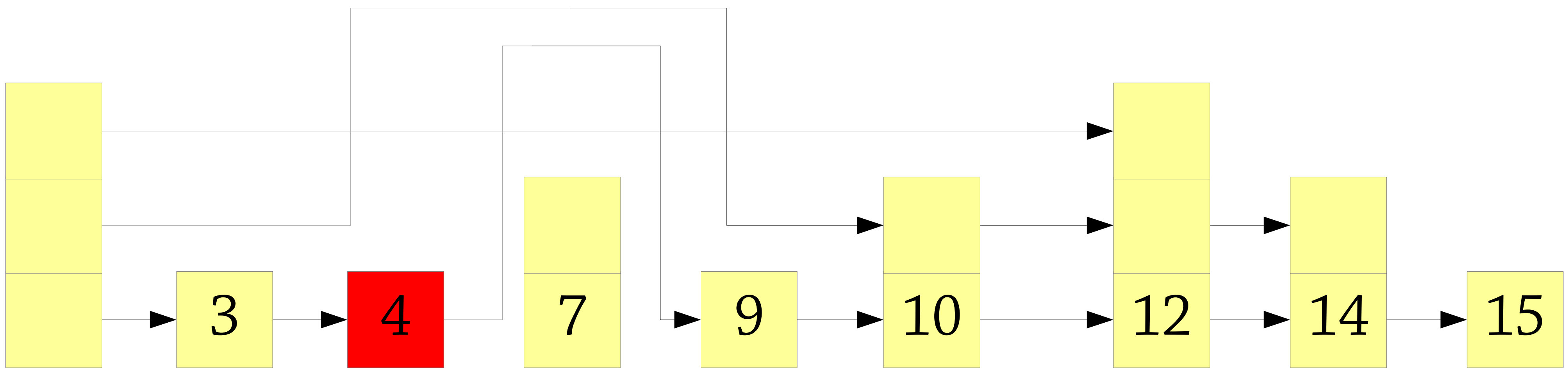
- Find level 1 predecessor



Probabilistic skip lists

Deletion: simply remove the node from the list – e.g., deleting 7, a level 2 node:

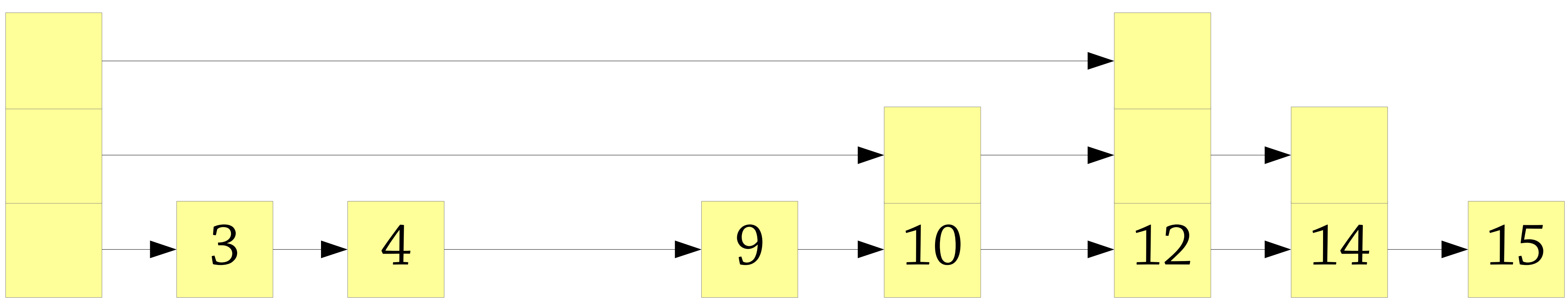
- Find level 1 predecessor
- Remove node from level 1



Probabilistic skip lists

Done!

Question: what happens if you delete all the nodes except the level 1 nodes?



Probabilistic skip lists

Deletion is dangerous...

- if you delete all nodes with level > 1, it degenerates to a linked list!

But, to do that you have to be extremely unlucky!

- When you delete a node, it has ½ chance of being level 2, ¼ chance of being level 4, etc., so you don't break the probabilistic behaviour
- The *probability distribution* of levels is the same before and after

So this is fine, *as long as* the user of the data structure can't see the level of each node

- Otherwise the probabilistic argument breaks down!

Probabilistic skip lists – summary

Give each node a random *level* when you create it

- Nodes with higher levels allow you to fast forward through the list

Insertion, deletion, lookup: O(log n) *expected* complexity

Code is pretty simple!

Can also be used to implement a *sequence* (array-like) datatype

Deterministic skip lists

Probabilistic skip lists are fast, but the lack of performance guarantee is a bit worrying

- e.g., if an attacker can see the random number seed, they can break the performance

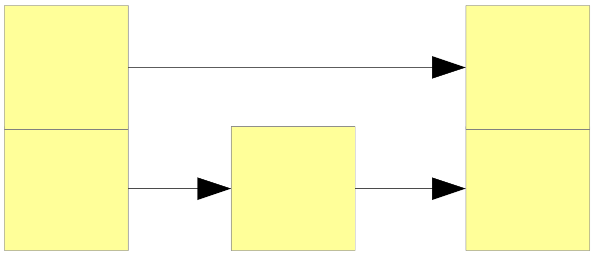
Deterministic skip lists have O(log n) time complexity whatever the situation

- Downside: deletion is a bit harder (we skip it)

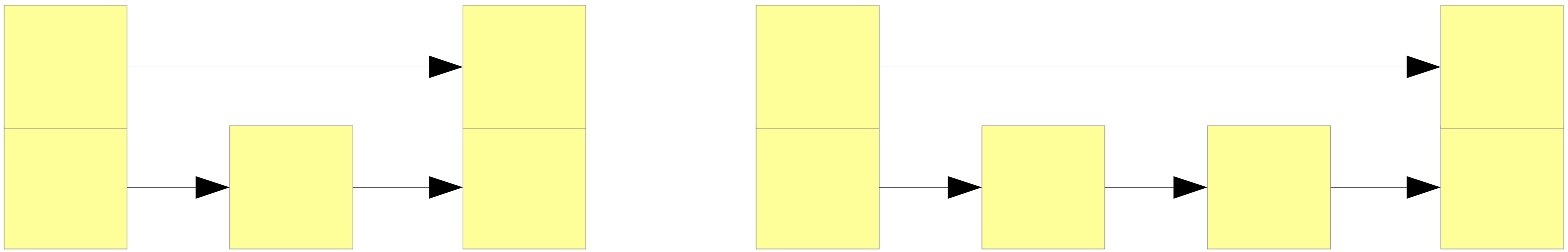
Inspired by 2-3 trees!

Deterministic skip lists

In a naive skip list, between each level n+1 node, there is only one level n node:

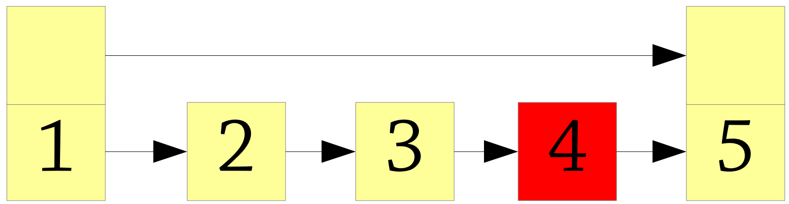


In a deterministic skip list, this can be either one or two nodes:

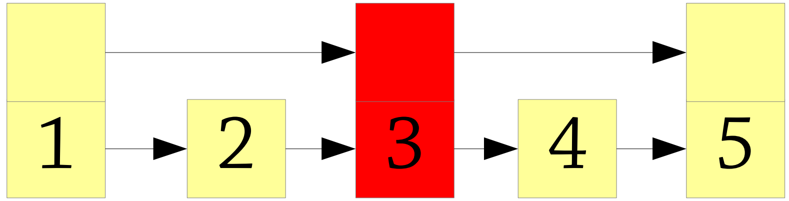


Deterministic skip lists

To insert into a deterministic skip list, first add a level 1 node:



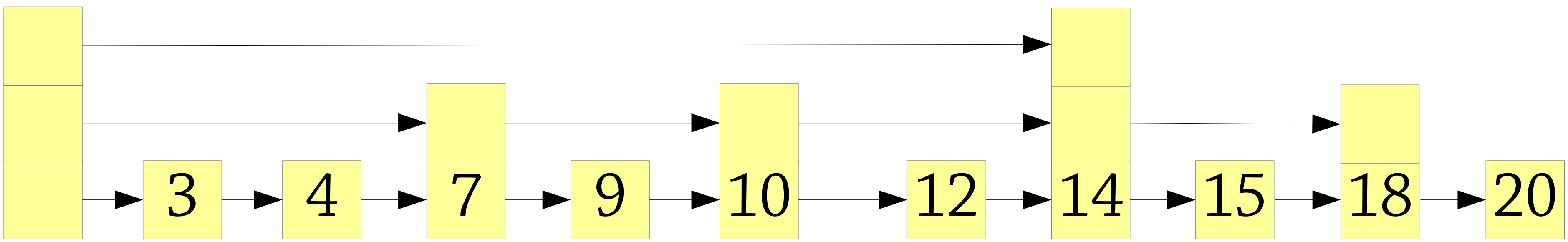
If this creates 3 level n nodes in a row, lift up the middle one to level n+1:



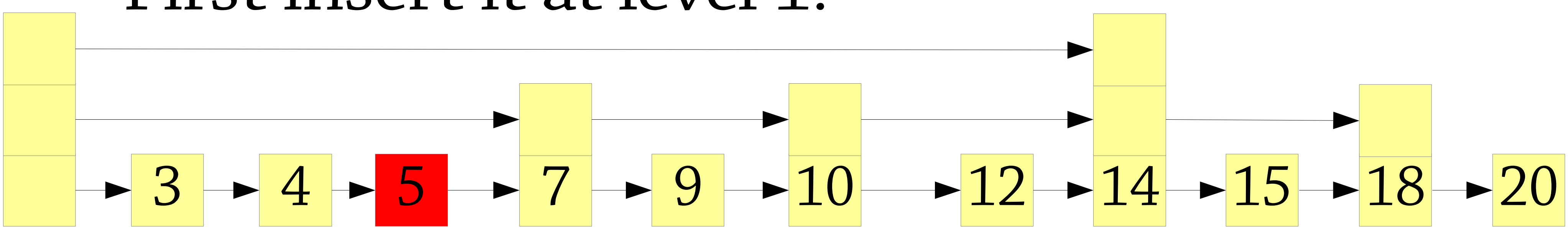
This might create three level n+1 nodes in a row, so continue up!

Insertion example

Inserting 5 into this skip list:



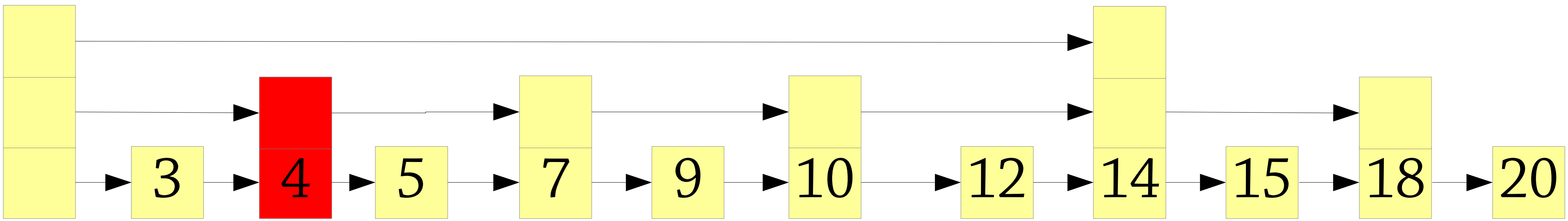
First insert it at level 1:



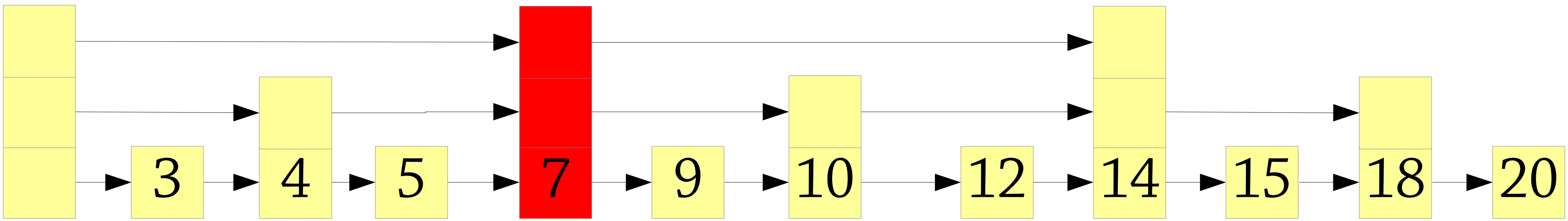
We've got three level 1 nodes without a level 2 node so promote 4 to level 2

Insertion example

4 has been promoted to level 2:



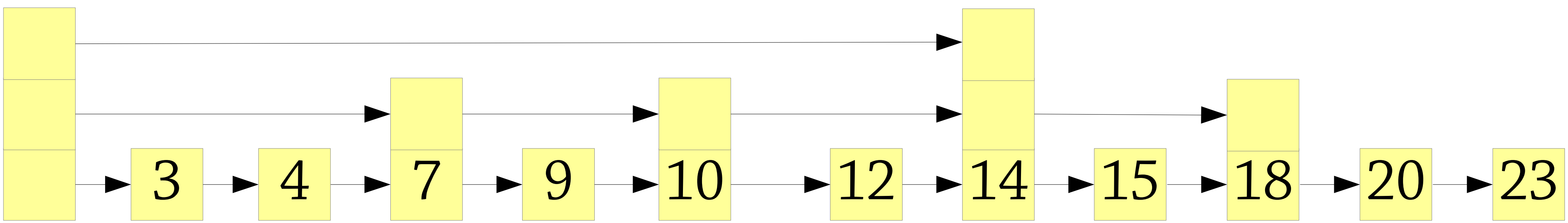
We've got three level 2 nodes (4, 7, 10) without a level 3 node so promote 7 to level 3:



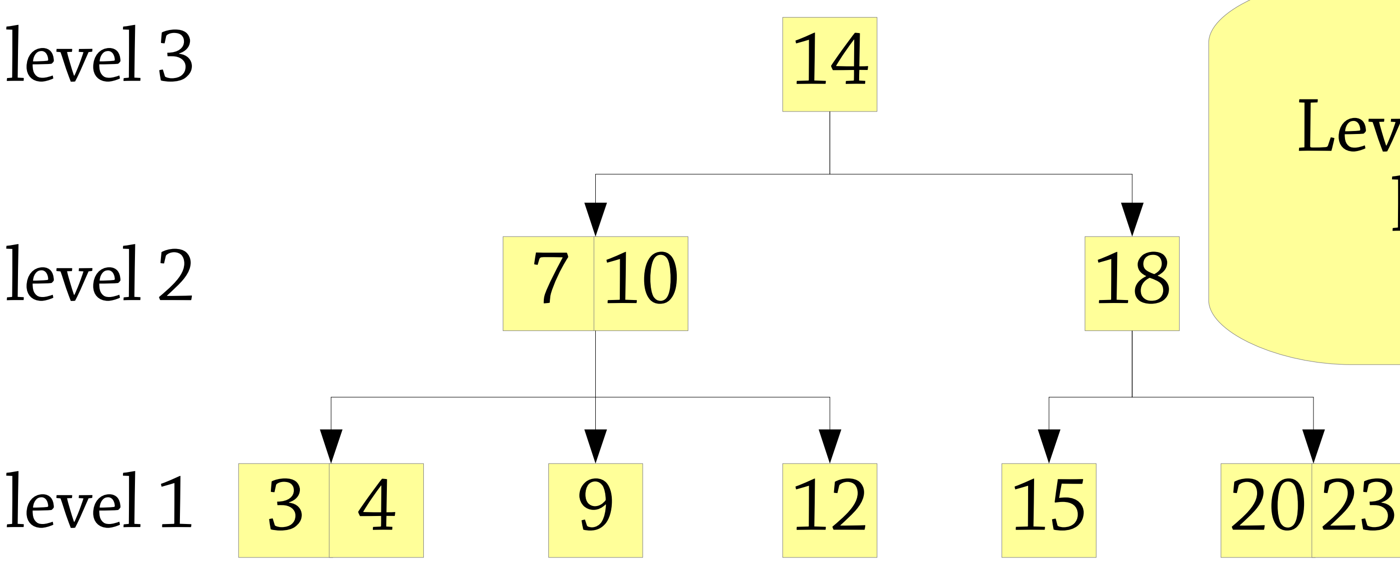
Done!

Relation to 2-3 trees

A deterministic skip list...



...and the corresponding 2-3 tree:



Level n skip list node =
level n tree node

Deterministic skip lists – summary

Allow either 1 or 2 level n nodes between each level n+1 nodes

- Can be seen as 2-3 trees, in fact *increasing the level* is very similar to *splitting the node*

What about deletion?

- Algorithm is inspired by 2-3 deletion
- Unfortunately gets rather complicated :(

Still, O(log n) cost for all operations, with relatively little code

But most skip lists are the probabilistic kind!

Skip lists versus trees

Skip list advantages:

- code is simpler
  (especially deletion in the probabilistic version)
- easy to iterate through the members of the list

Disadvantages:

- must be implemented as a *mutable* structure
  (bad in a functional language, or if you want to keep old versions around)
- only has probabilistic behaviour unless you use the more complicated version