So far...

Complexity analysis

- For recursive and iterative programs

Sorting algorithms

- Bubble, insertion, selection, quick, merge, counting

Binary search, dynamic arrays

Rest of course: lots of data structures!

Abstract data types and the Java collections framework (first half of chapter 6)

Abstract data type

Separate the *interface* of a data structure from the *implementation*

E.g., a set should support insertion, deletion and membership testing:

```
interface Set<E> {
    void add(E element);
    void remove(E element);
    bool contains(E element);
}
```

Abstract data type

An ADT can have several implementations. For Set<E>, there is

- A dynamic array: O(1) insert, O(n) delete, O(n) contains
- A sorted dynamic array: O(n) insert, O(n) delete, O(log n) contains
- A balanced binary search tree (later): all operations O(log n)

By using an ADT, you can easily switch to a different implementation.

Advantages of ADTs

Can program to an *interface* and choose or change the implementation later

Don't need to worry about implementation details: “what index should I insert this new element in?” Just use the methods the ADT defines.

Java collections framework

The *Java collections framework* is a large library of data structures implementing abstract data types

- Collection
- List – ArrayList, LinkedList, …
- Set – HashSet, TreeSet, …
- Map – HashMap, TreeMap, ConcurrentHashMap, …

Found under java.util.*

Collection<E>

A "collection" of stuff in unspecified order, e.g. a list, a set, …. Most Java data structures implement Collection<E>.

```
interface Collection<E> {
  boolean isEmpty();
  int size();
  boolean contains(E x);
  Iterator<E> iterator();
  // Optional
  boolean add(E x);
  boolean remove(E x);
  void clear();
}
```

An iterator

Iterators

An Iterator<E> lets us iterate over a collection:

```
interface Iterator<E> {
  // Are there any objects left?
  boolean hasNext();
  // Move to the next object.
  E next();
}
```

Using an iterator

If coll has type Collection<E>, then

```
for (E x: coll) { … }
```

which means the same as

```
Iterator<E> it = coll.iterator();
while (it.hasNext()) {
   E x = it.next();
   …
}
```

Iterators

```
class ArrayList<E> implements Collection<E> {
  E[] array;
  int size;
  …
  Iterator<E> iterator() {
    final E[] array = this.array;
    final int size = this.size;
    return new Iterator<E>() {
      int i = 0;
      boolean hasNext() {
        return i < size;
      }
      E next() {
        return array[i++];
      }
      void clear() { throw new UnsupportedOperationException(); }
    }
  }
}
```

Compare with for-loop:

```
for (int i = 0;
     i < size;
     i++) {
  …
}
```

Collection and Iterator

Collection<E>: unordered collection of stuff

- Operations: add, remove, check for membership, *iterate*

Iterator<E>: allows you to loop through a collection

- Built-in for-each syntax in Java

The two most important ADTs in the Java collections framework – most data structures extend Collection<E>

Linked lists
(chapters 6.5, 17)

Enkellänkade listor

Att lägga till och ta bort element inuti en ArrayList tar linjär tid O(n)

- vilket är oändligt långsamt i vissa applikationer

En länkad lista kan lägga till och ta bort varsomhelt i en lista i konstant tid, O(1)

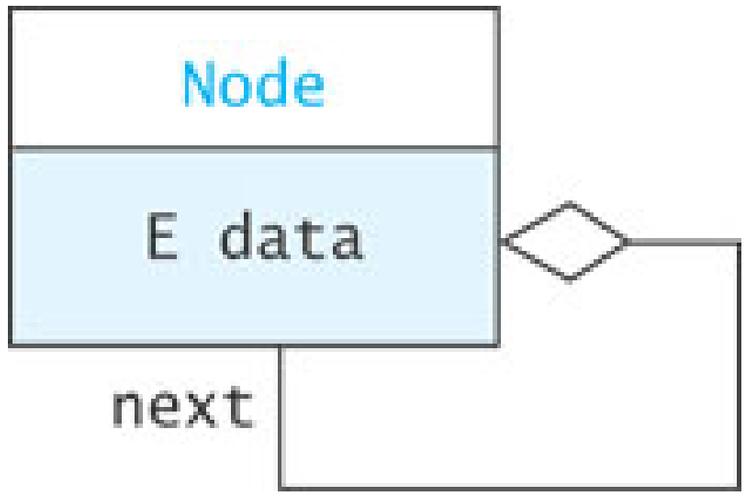
- å andra sidan tar det linjär tid O(n) att gå till en specifik position i en länkad lista, vilket är konstant i en ArrayList

I en länkad lista så är varje element (”nod”) länkad till det efterföljande elementet

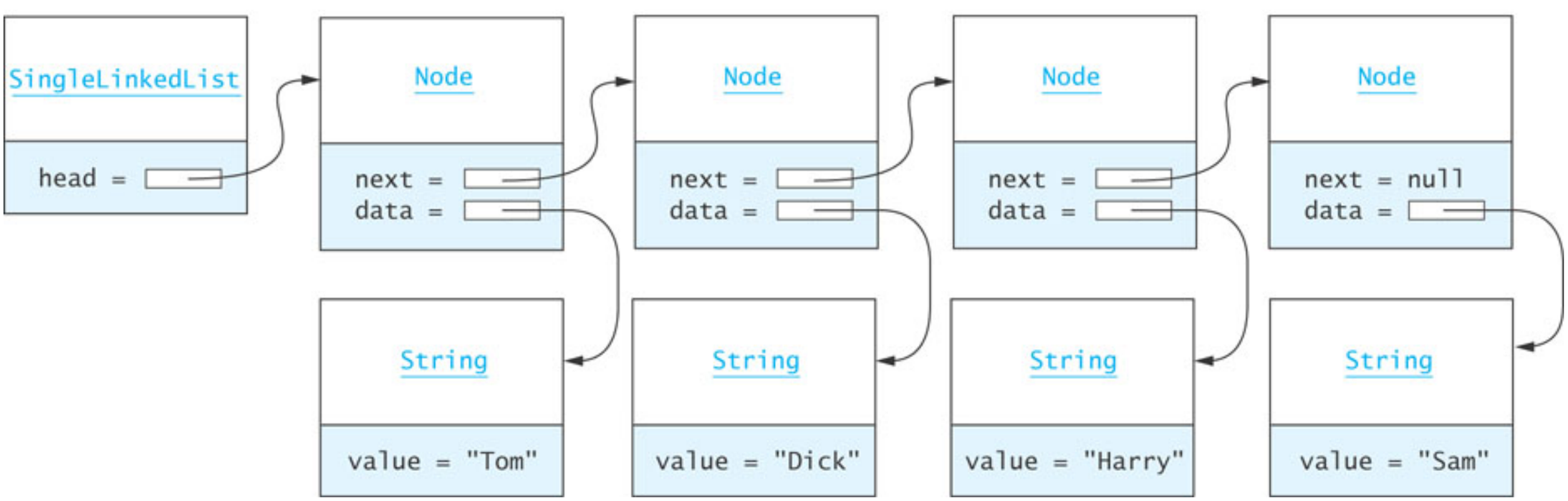
En enkellänkad listnod innehåller

- noddata, och
- en länk till efterföljaren

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



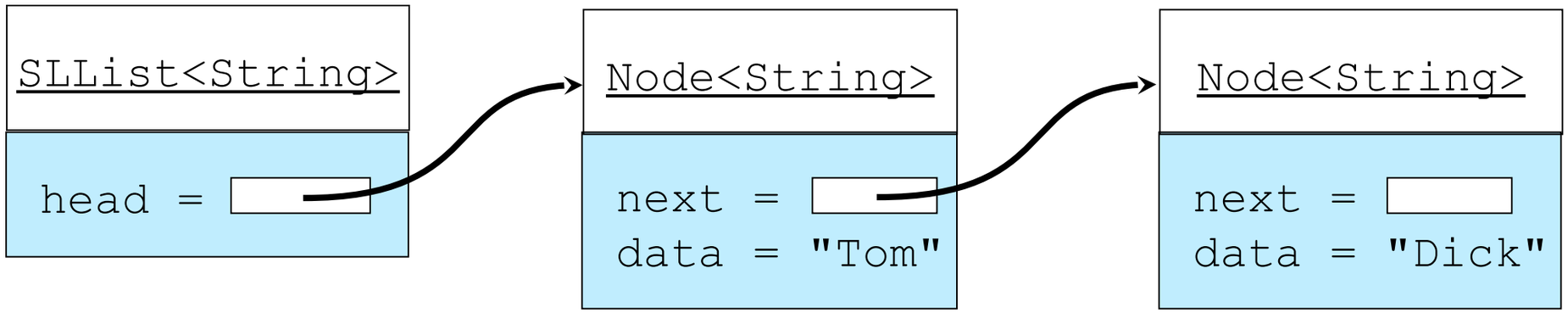
Länkad representation av listan {”Tom”, ”Dick”, ”Harry”, ”Sam”}



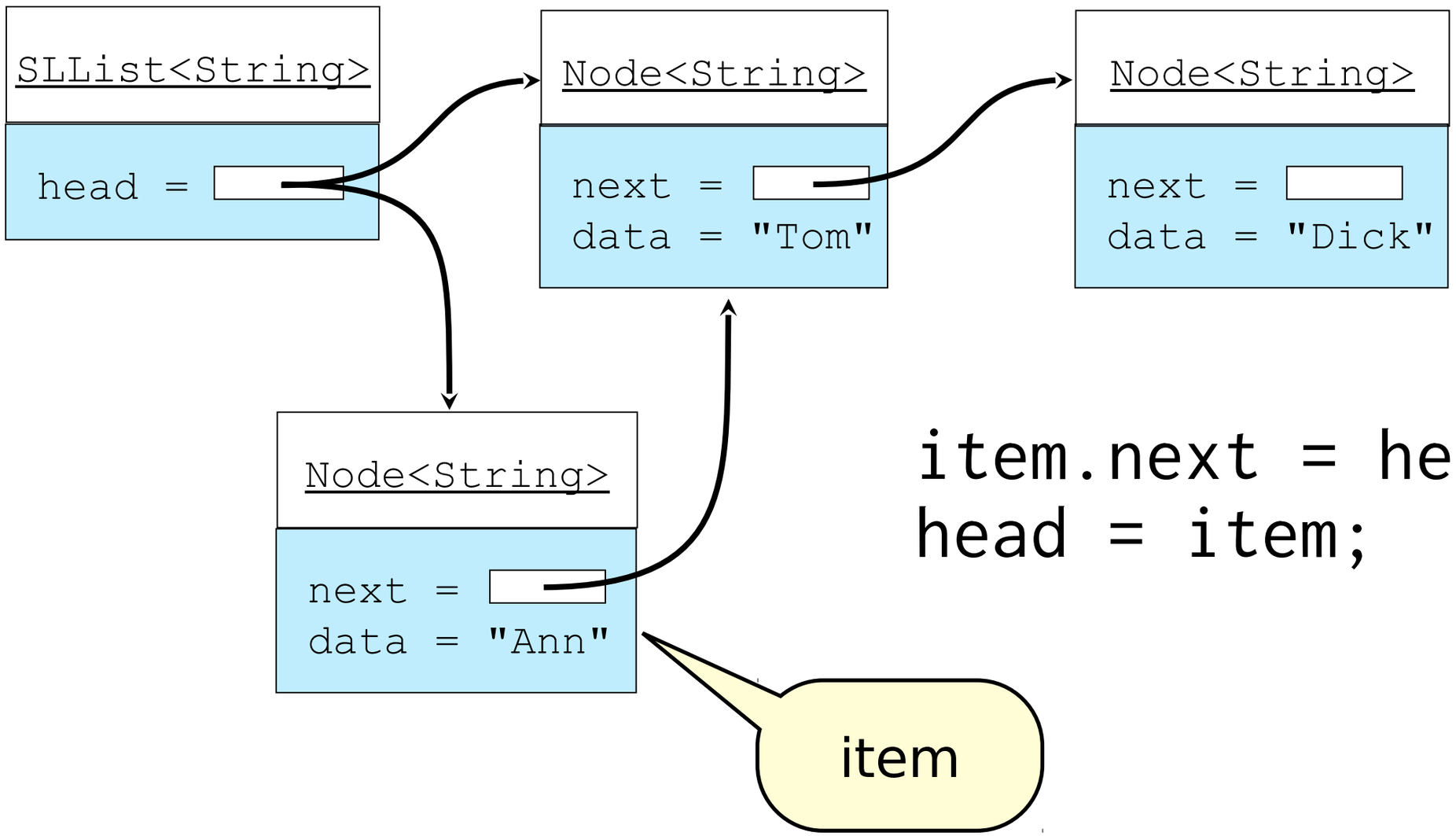
Operations on linked lists

```
// 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)
```

Exempellista

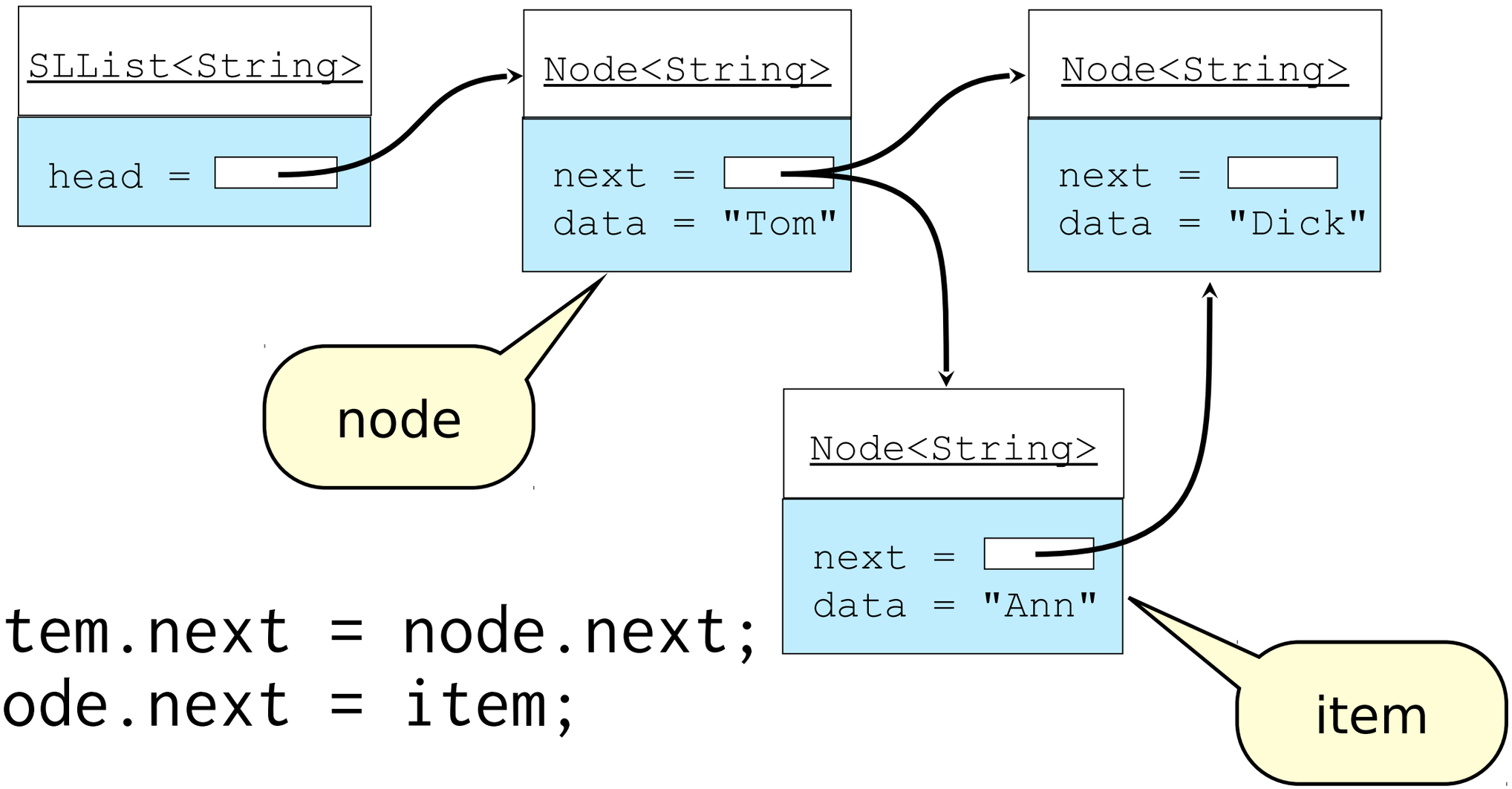


Metoden addFirst(E item)



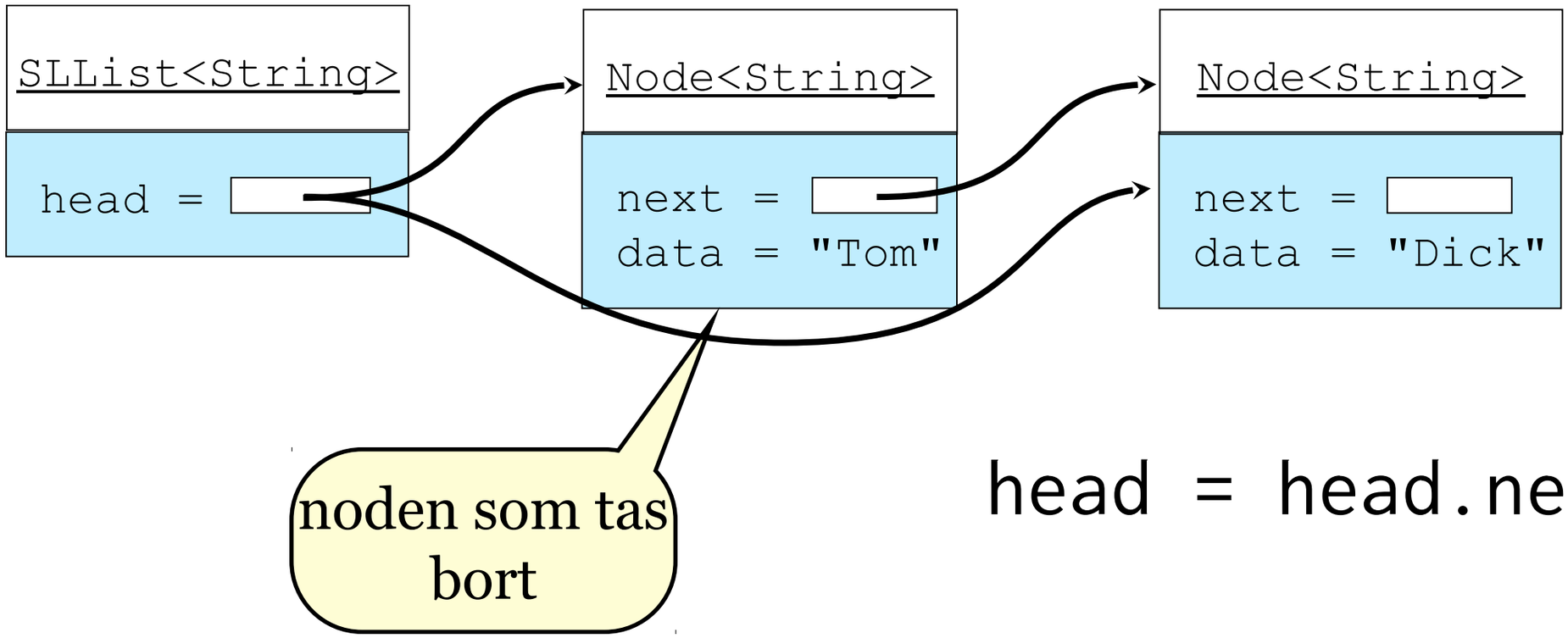

```
item.next = head;
head = item;
```

addAfter(Node<E> node, E item)



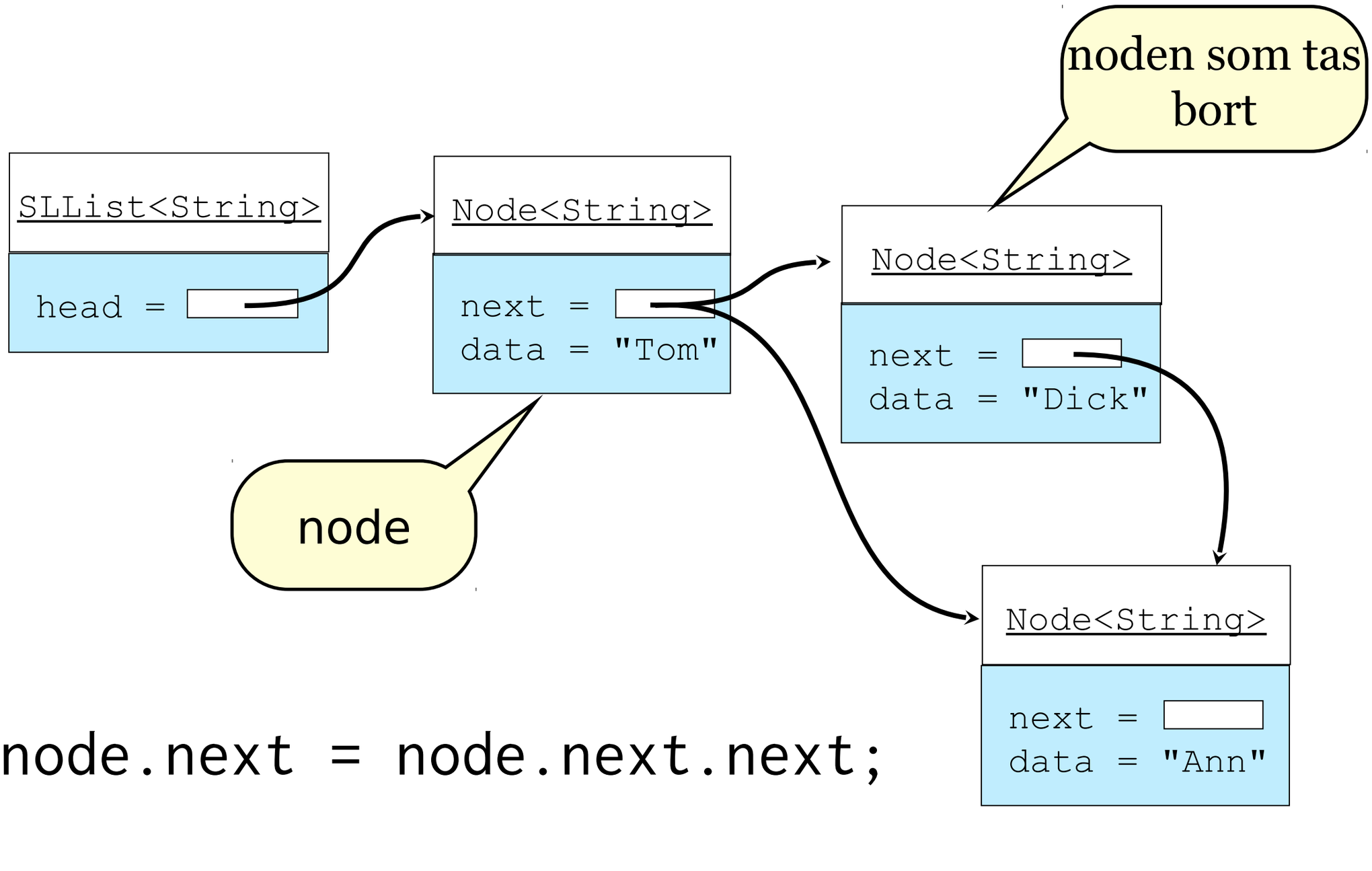

```
item.next = node.next;
node.next = item;
```

removeFirst()




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

removeAfter(Node<E> node)




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

A problem

It's annoying to have two versions of every operation: addFirst/addAfter, removeFirst/removeAfter

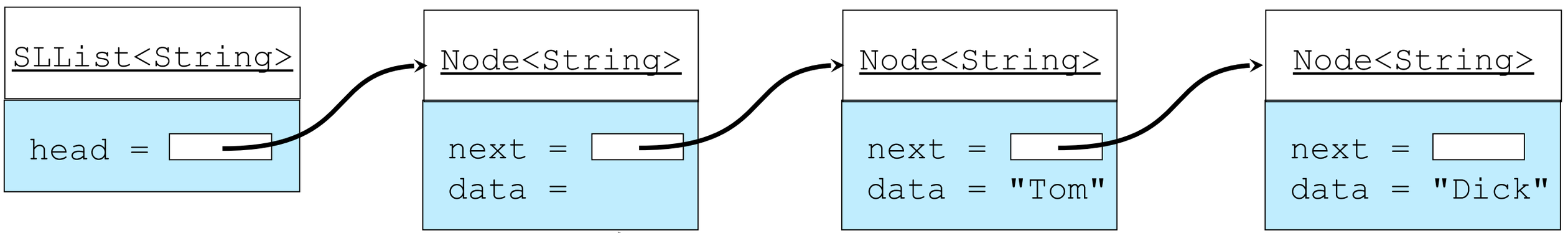
Why do we need addFirst/removeFirst?

- We cannot use addAfter/removeAfter, because the first node is not after any other node

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

Instead of addFirst(x), you can do addAfter(head, x)

List with header node (17.1.1)



Here is the header node –
it does not contain any data

Doubly-linked lists

With singly-linked lists you can only go *forwards* through the list

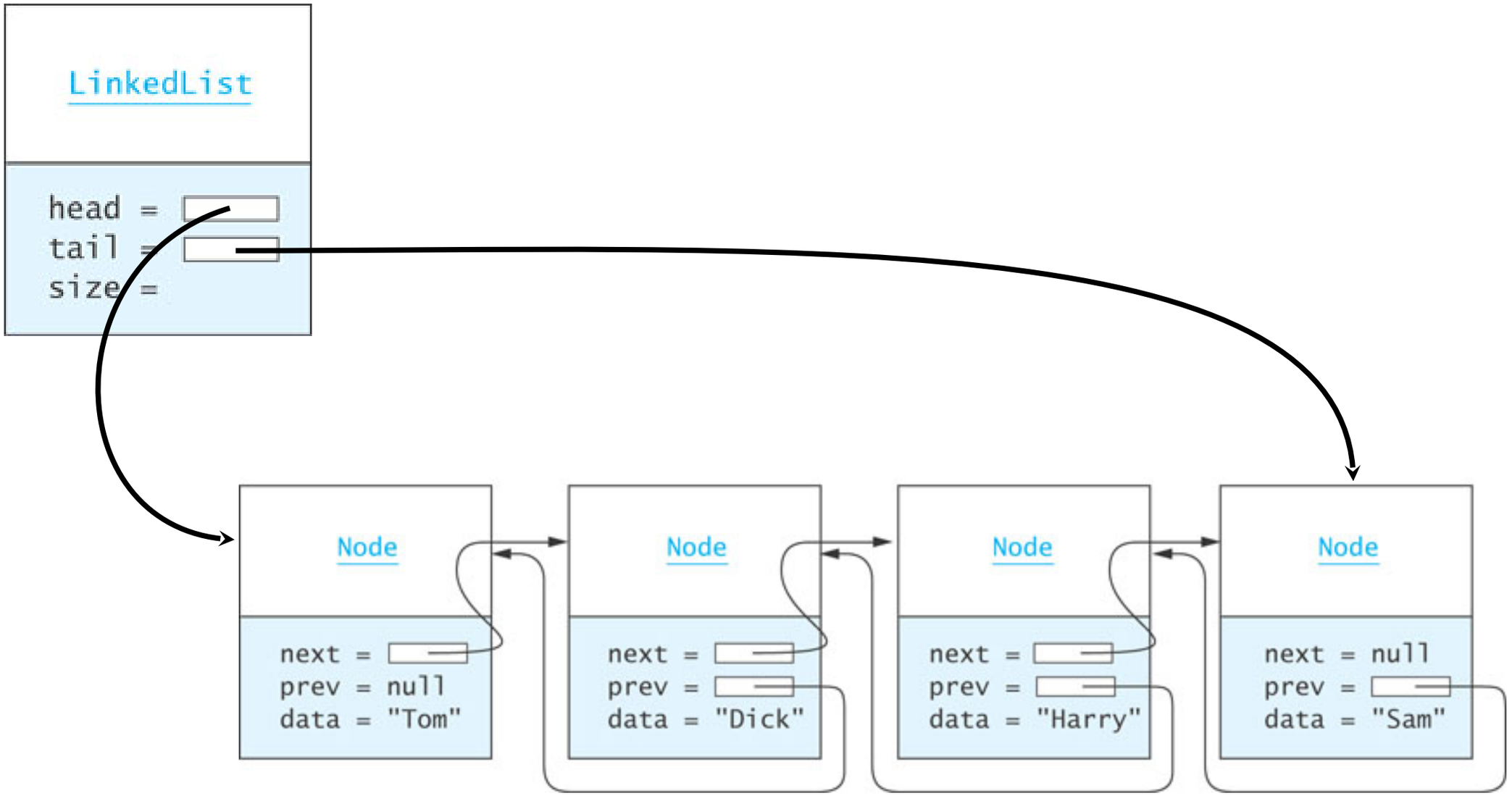
Doubly-linked list: each node has a link to the next *and the previous* node

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 best data structure for sequential access!

Dubbellänkade listor



Operations on doubly-linked lists

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

Deleting a node:

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

But this will CRASH if we try to delete the first node, since then `node.prev == null`! Same for the last node!

We need lots of special cases in all our operations to work around this.

```
if (node == head) …
else if (node == tail) …
else ...
```

Operations on doubly-linked lists

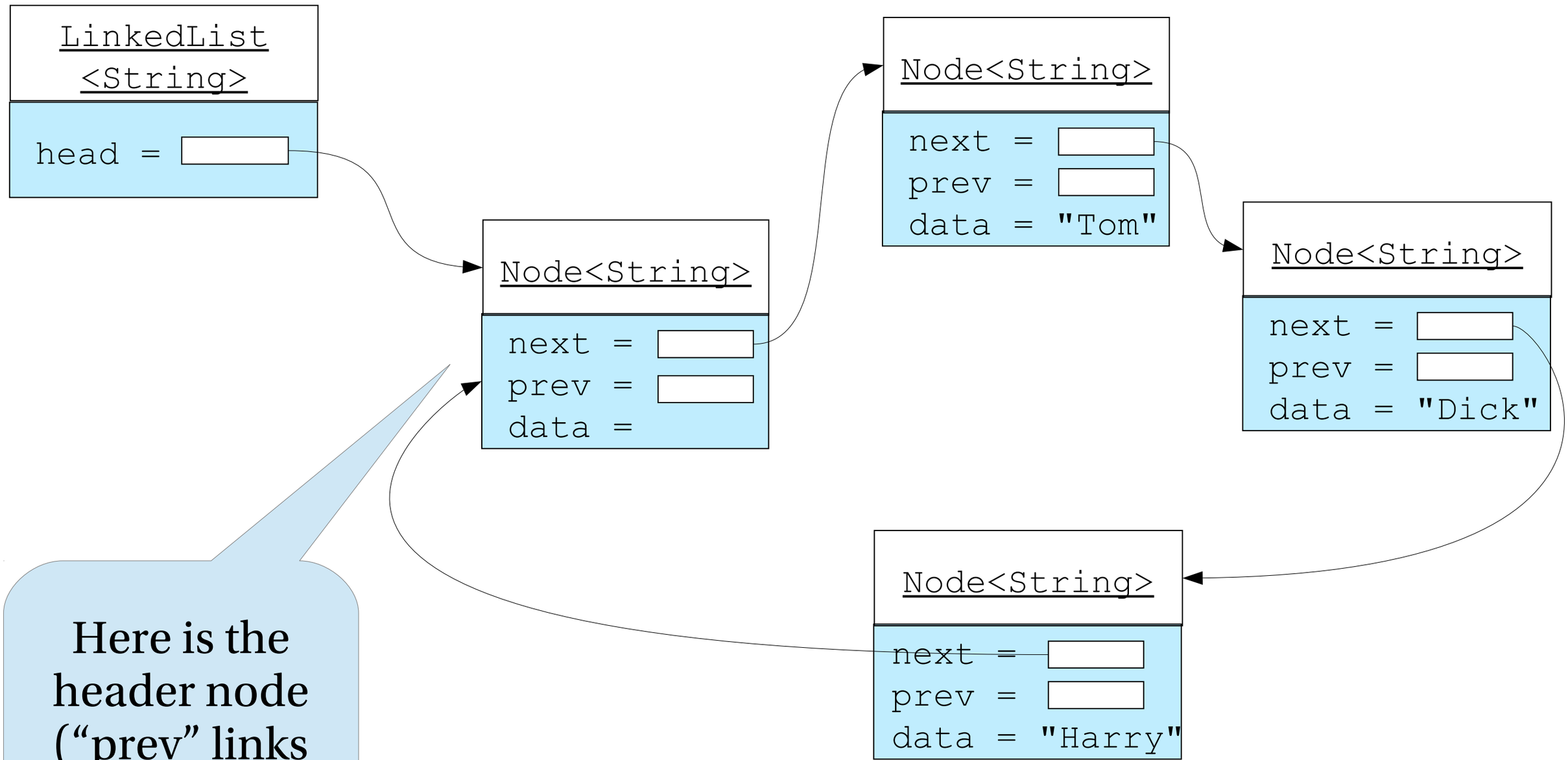
How can we get rid of all the special cases?

Idea (see book): use a *header node* like for singly-linked lists, and also a footer node. `head` and `tail` will point at the header and footer node. Since you never delete the header or footer node, there are no special cases

Problem: allocates two extra nodes per list

Cute solution: *circularly-linked list with header node*

Circularly-linked list with header node



Circularly-linked lists with header

`head.next` is the first element in the list, and `head.prev` is the last element

No special cases, since `node.next` and `node.prev` are never null, and you never need to delete `head`

Quite simple to implement!

Also possible to do it without a header node – then you sometimes have to update *head*, so there are some special cases

- But we only need a pointer to the first element, not the last, so there are fewer special cases (the last node is `head.prev`)

Linked lists vs dynamic arrays

Dynamic arrays:

- have O(1) random access (get and set)
- have *amortised* (average) 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

Complement each other!

Both implement the List interface.

List

A List is a Collection that supports random and sequential access:

```
interface List<E>
extends Collection<E> {
  // get or set index idx
  E get(int idx);
  E set(int idx, E x);
  // add or remove at index idx
  boolean add(int idx, E x);
  void remove(int idx);
  ListIterator<E>
    listIterator(int idx);
}
```

ListIterator

A ListIterator is like an iterator, but you can go backwards, and you can add and remove elements at the current position:

```
interface ListIterator<E>
extends Iterator<E> {
  // go back
  boolean hasPrevious();
  E previous();
  // add or remove an element
  void add(E x);
  void set(E x);
  void remove(); // actually in Iterator
}
```

A ListIterator points at a space *between* two elements! When you call `add`, the new element will go in this space. See the documentation for details.

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

Efficient data structure for *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 ADT

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

But unlike arrays, random access is O(n)

LinkedList<E> class in Java

Stacks and queues
(chapters 6.6, 16)

Stackar

Methods	Behavior
`boolean empty()`	Returns **`true`** if the stack is empty; otherwise, returns **`false`**.
`E peek()`	Returns the object at the top of the stack without removing it.
`E pop()`	Returns the object at the top of the stack and removes it.
`E push(E obj)`	Pushes an item onto the top of the stack and returns the item pushed.

Det ”översta” elementet är det senaste

- LIFO = Last-In-First-Out

Stackar används överallt i datorer

- t.ex. vid funktions-/metodanrop (run-time stack, call stack)

Balanserade parenteser

Vi behöver en stack för att kontrollera ifall ett uttryck har balanserade parenteser:

- speciellt om uttrycket kan innehålla olika sorters parenteser
- `( a + b * [ c / { d - e } ] ) + { d / e }`

Algoritmidé:

- när vi läser (, [eller { så lägger vi den på stacken
- när vi läser),] eller } så poppar vi stacken och kontrollerar att parenteserna stämmer

Balanserade parenteser, iterativt

```
private static final String PARENS  = "()[]{}";

public static void parseParensIterative(String str) throws ParseException {
        Stack<Integer> parenStack = new Stack<Integer>();
        for (int i = 0; i < str.length(); i++) {
                char c = str.charAt(i);
                int paren = PARENS.indexOf(c);
                if (paren >= 0) {
                        if (paren % 2 == 0) {
                                parenStack.push(paren);
                        } else {
                                if (parenStack.empty())
                                        throw new ParseException("Too many close parentheses", i);
                                int openParen = parenStack.pop();
                                if (paren != openParen + 1)
                                        throw new ParseException("Unbalanced parentheses", i);
                        }
                }
        }
        if (!parenStack.empty())
                throw new ParseException("Too many open parentheses", str.length());
}
```

vi behöver en stack för att hålla reda på parenteserna

detta förutsätter att ([{ ligger på jämna index i PARENS

detta förutsätter att (och) ligger efter varandra i PARENS

Implementera en stack

Vi kan implementera stacken som ett fält:

- vi måste kunna omallokera fältet
- det blir väldigt likt en ArrayList

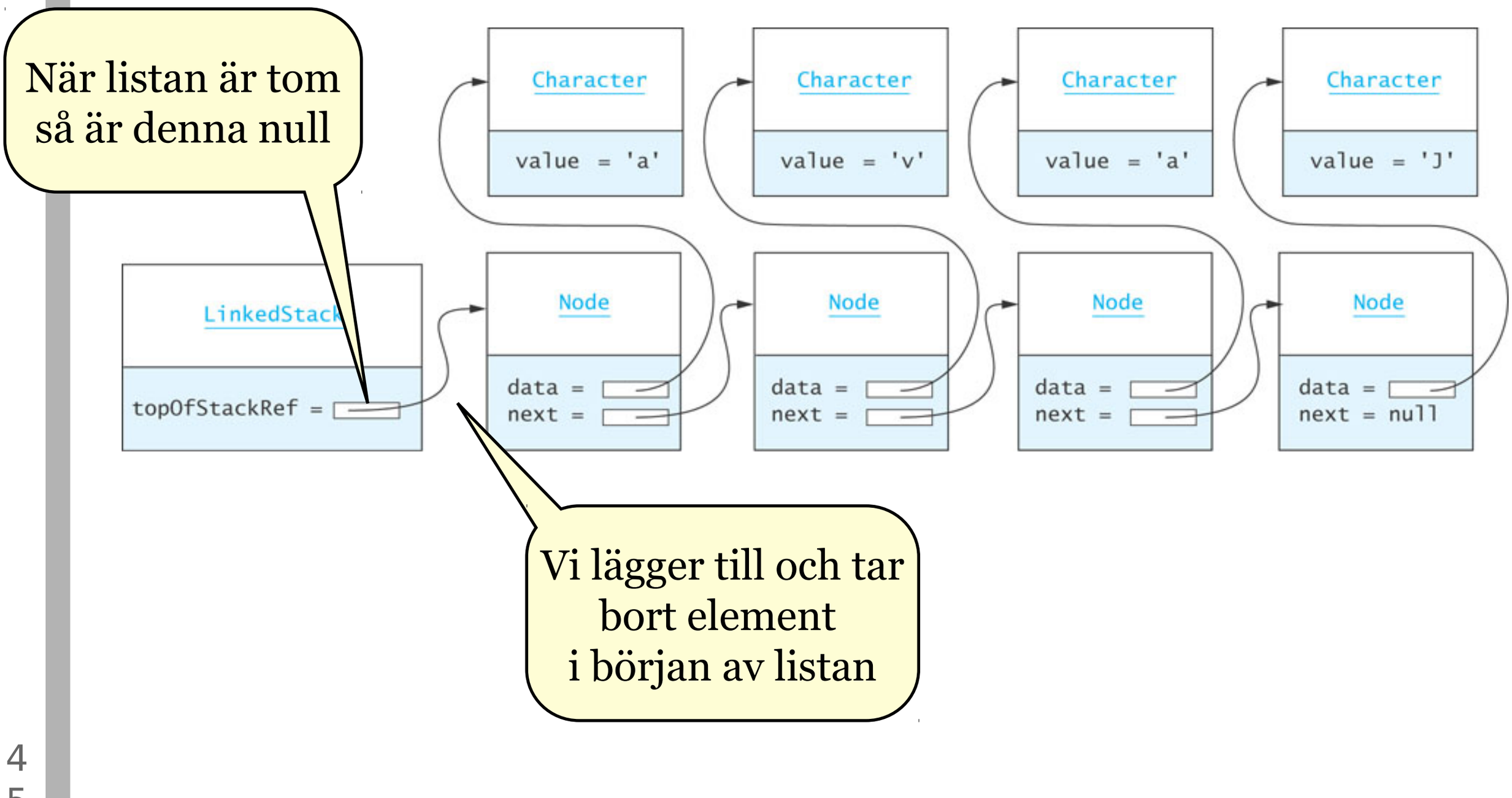
Vi kan implementera stacken som en länkad lista:

- det blir väldigt likt en LinkedList

Vi kan också använda intern privat lista:

- t.ex. ArrayList eller LinkedList
- stackmetoderna (t.ex. `push(E)`) kommer då att anropa den interna listans metoder (t.ex. `add(E)`)
- det kallas *delegering*

En stack som en länkad lista



A stack as a linked list

Really easy!

```
LinkedList<E> list = new …
void push(E x) {
  list.addFirst(x);
}
E pop() {
  return list.removeFirst();
}
E peek() {
  return list.getFirst();
}
```

Do we need a separate stack ADT?

Since it's so easy to implement a stack as a linked list, why do we want a separate Stack<E> interface?

- Using Stack<E> allows us to switch to a different implementation (like an array)
- We also get *only* the operations that we need

But in Java Stack<E> is a *class* extending Vector<E>! This is a mistake. Use Deque<E> (later) instead

Komplexitet för stackmetoder

Den stora fördelen med stackar är att de är väldigt effektiva:

- alla operationer har konstant tidskomplexitet, O(1)
- dvs, push(E), pop(), peek() och empty()

Köer

Method	Behavior
`boolean offer(E item)`	Inserts `item` at the rear of the queue. Returns **`true`** if successful; returns **`false`** if the item could not be inserted.
`E remove()`	Removes the entry at the front of the queue and returns it if the queue is not empty. If the queue is empty, throws a `NoSuchElementException`.
`E poll()`	Removes the entry at the front of the queue and returns it; returns **`null`** if the queue is empty.
`E peek()`	Returns the entry at the front of the queue without removing it; returns **`null`** if the queue is empty.
`E element()`	Returns the entry at the front of the queue without removing it. If the queue is empty, throws a `NoSuchElementException`.

”Översta” elementet är det som har väntat längst

- FIFO = First-In-First-Out

Köer är ett vanligt alternativ till stackar:

- används i operativsystem för att hålla koll på processer som väntar på en resurs, t.ex. en skrivarkö
- flera algoritmer använder köer för att implementera *bredden-först*-sökning

LinkedList är en Queue

Java har ett gränssnitt för köer:

- java.util.Queue är en utökning av java.util.Collection
- java.util.LinkedList implementerar java.util.Queue

För att skapa en kö i Java:

```
Queue<String> names = new LinkedList<String>();
```

- detta skapar en LinkedList, men eftersom variabeln är deklarerad som en Queue så kan vi bara använda kömetoderna

Java har **inget** gränssnitt för stackar:

- java.util.Stack är en klass, inte ett gränssnitt
- **DÅLIG** design!

Att implementera en kö

Vi kan implementera kön som en *dubbellänkad lista*:

- det är så det är gjort i Java, eftersom LinkedList är en dubbellänkad lista

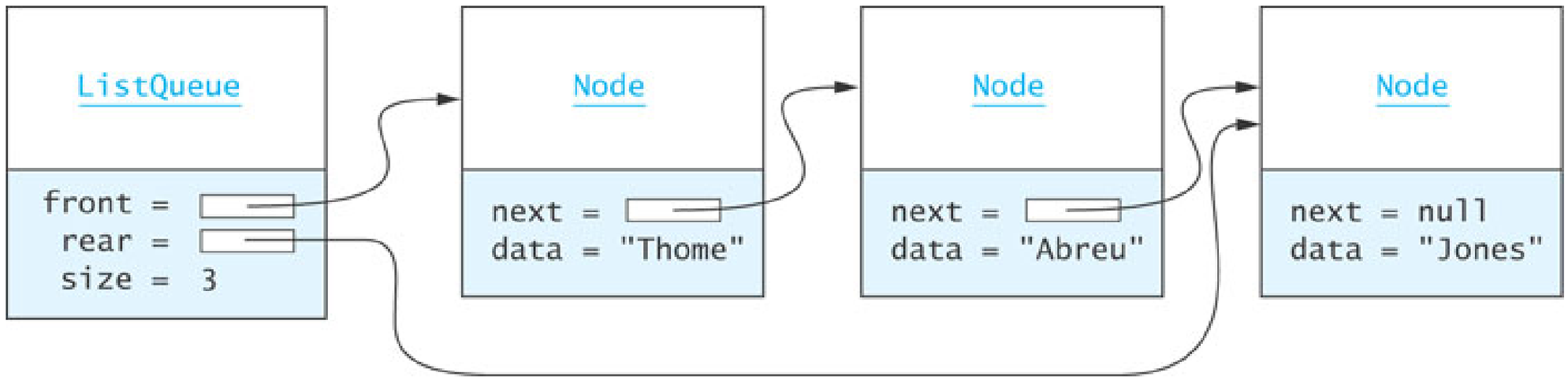
Men det är egentligen onödigt – det går precis lika bra med en *enkellänkad lista*:

- vi måste ha en pekare till huvudet och en pekare till svansen
- vi stoppar in element i svansen och tar ut dem i huvudet

Det funkar också att implementera som ett *cirkulärt fält*:

- vi måste ha två heltalsvariabler – som säger var början resp. slutet av kön är
- precis som en ArrayList måste vi kunna omallokera fältet när kön blir för stor

En kö som en enkellänkad lista



Insättningar görs sist i listan:

- ”Jones” är alltså det senast tillagda elementet
- vi behöver peka om rear.next och sedan rear till den nya noden

Element tas bort från början av listan:

- ”Thome” är alltså det element som har väntat längst
- vi behöver peka om front till front.next

En kö som ett fält

Vi kan implementera en kö som ett fält.

Antingen med ”översta” elementet först:

- insättning i slutet = O(1)
- borttagning i början = O(n)

Eller med ”översta” elementet sist:

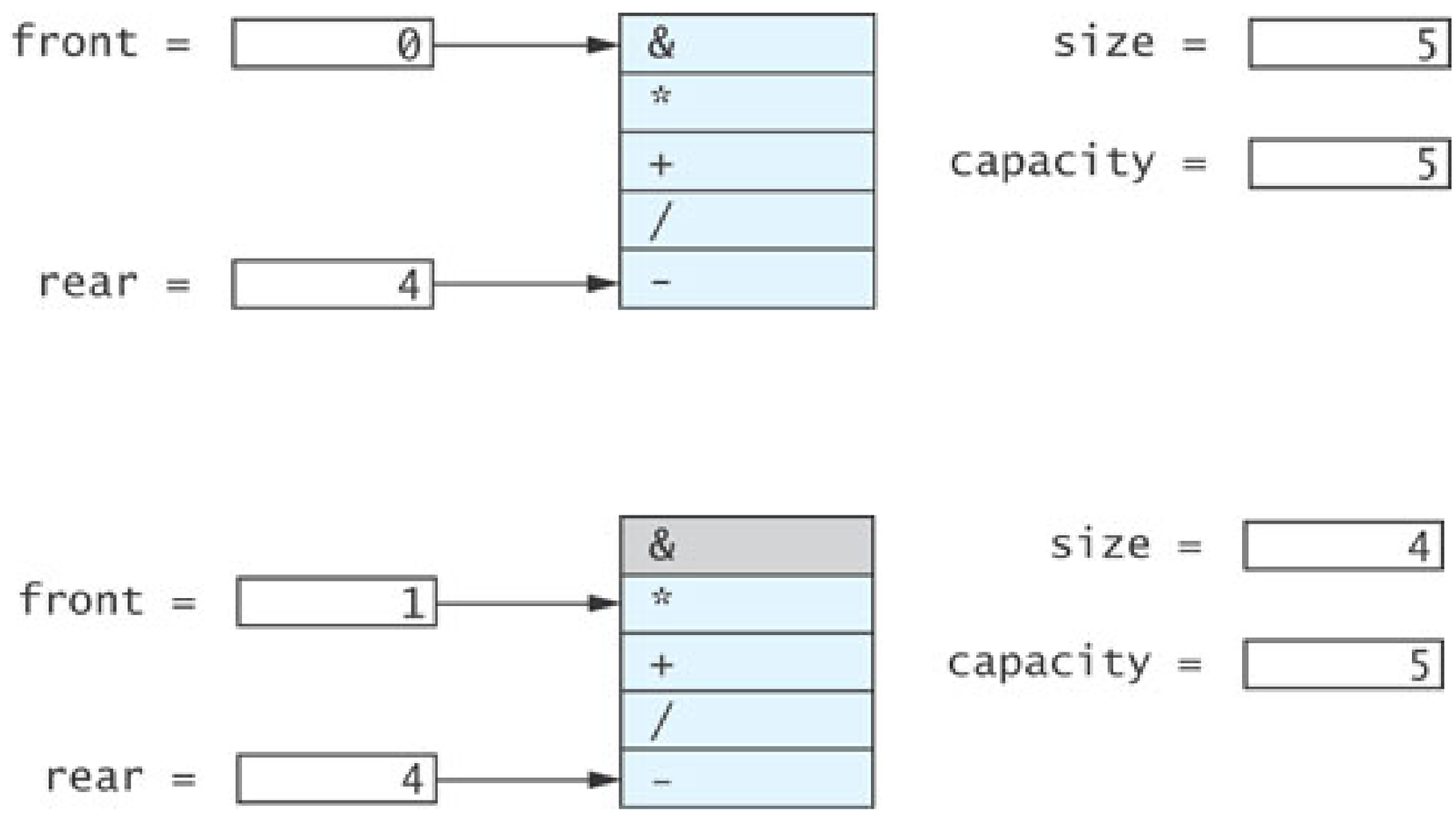
- insättning i början = O(n)
- borttagning i slutet = O(1)

Dvs, alldeles för ineffektivt.

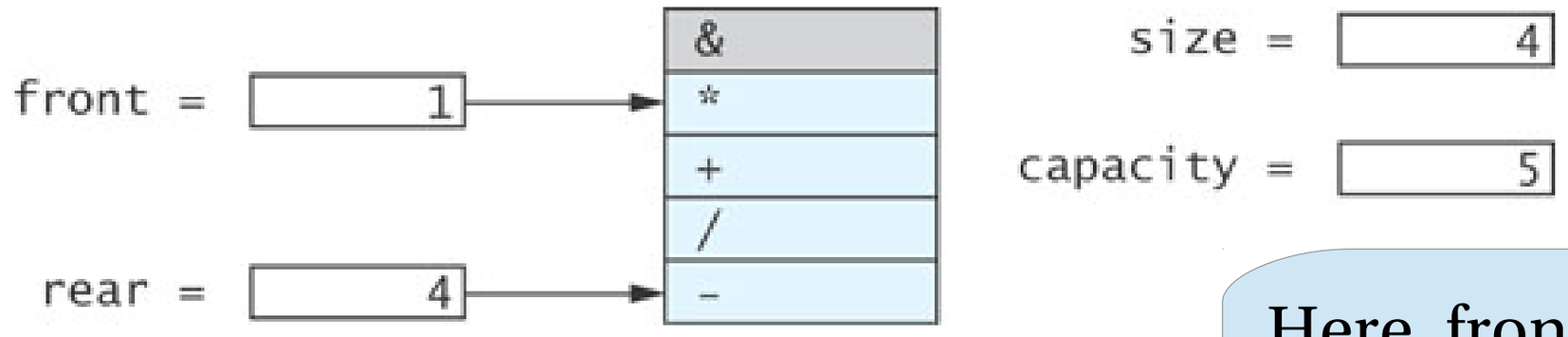
Lösningen är att använda ett cirkulärt fält.

En kö som ett cirkulärt fält

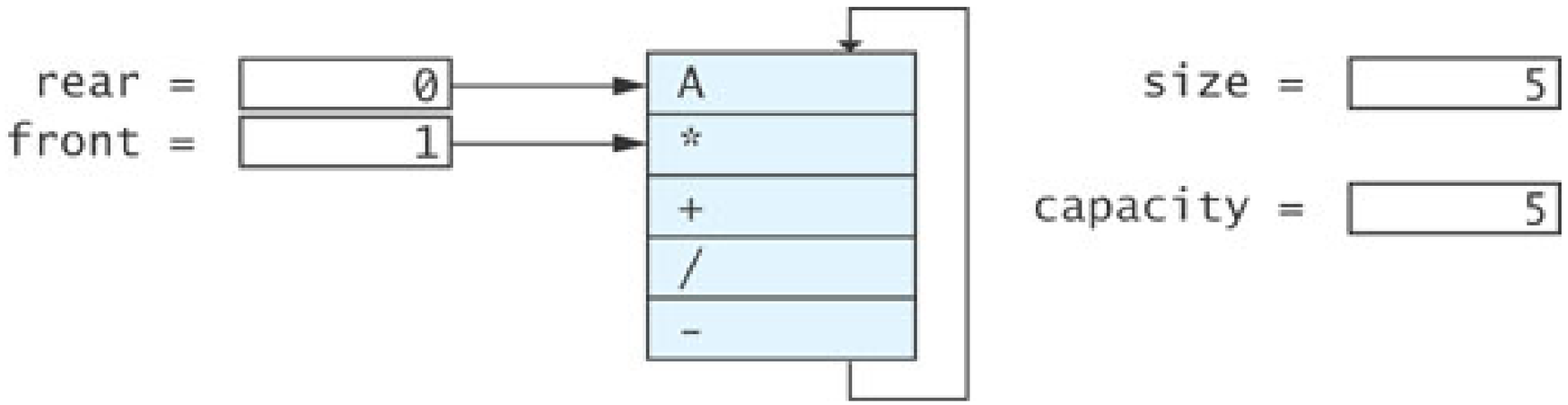
Queue contains everything between front and rear: & * + / -



Ett cirkulärt fält

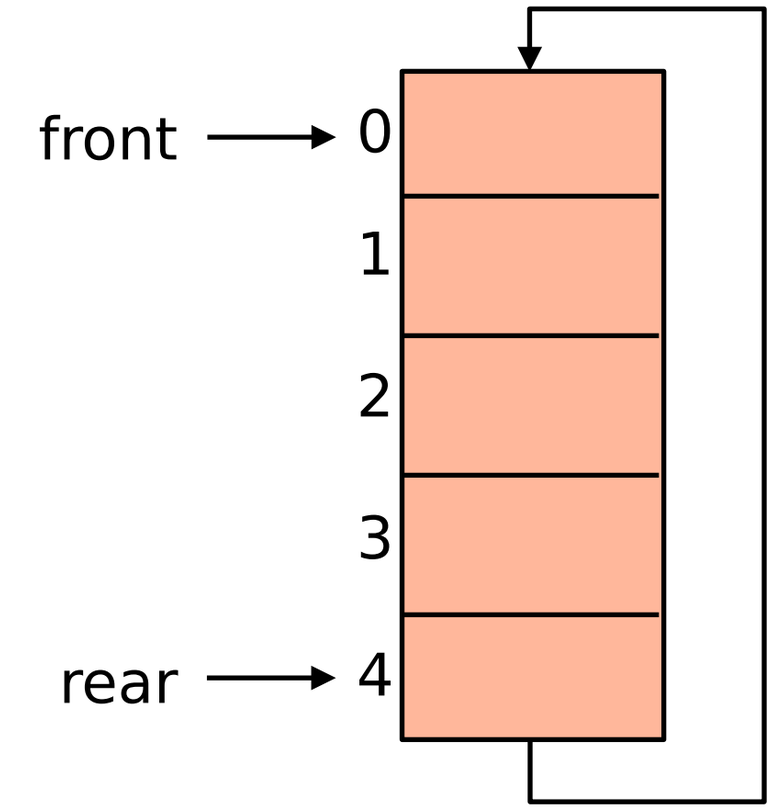


Here, front comes *after* rear! Array is circular: queue contains * + / - A



Exempel

ArrayQueue q = new ArrayQueue(5);

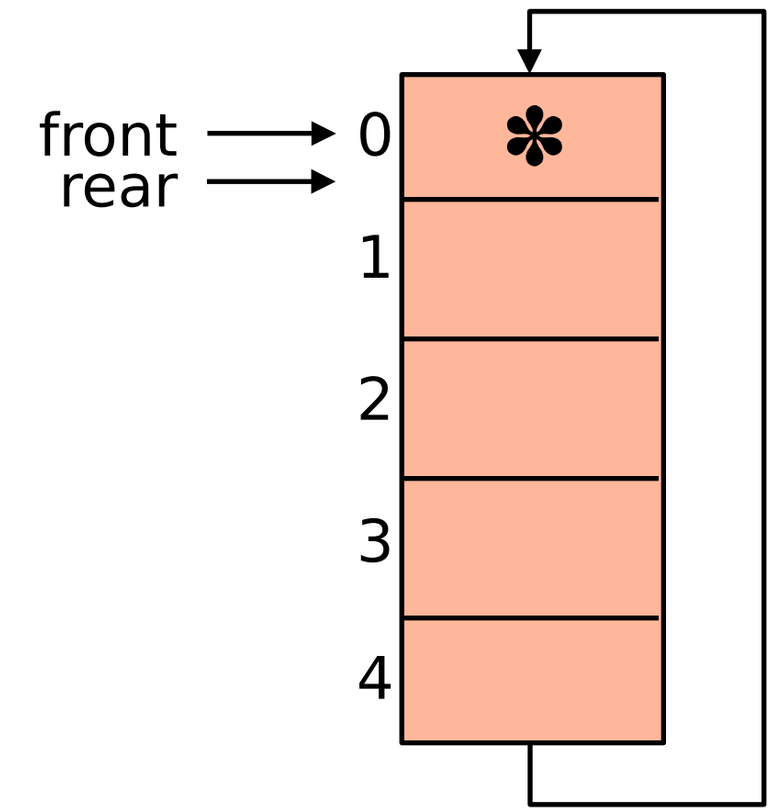


size = 0

capacity = 5

```
public ArrayQueue(int initCapacity) {
        capacity = initCapacity;
        theData = (E[]) new Object[capacity];
        front = 0;
        rear = capacity – 1;
        size = 0;
}
```

Exempel



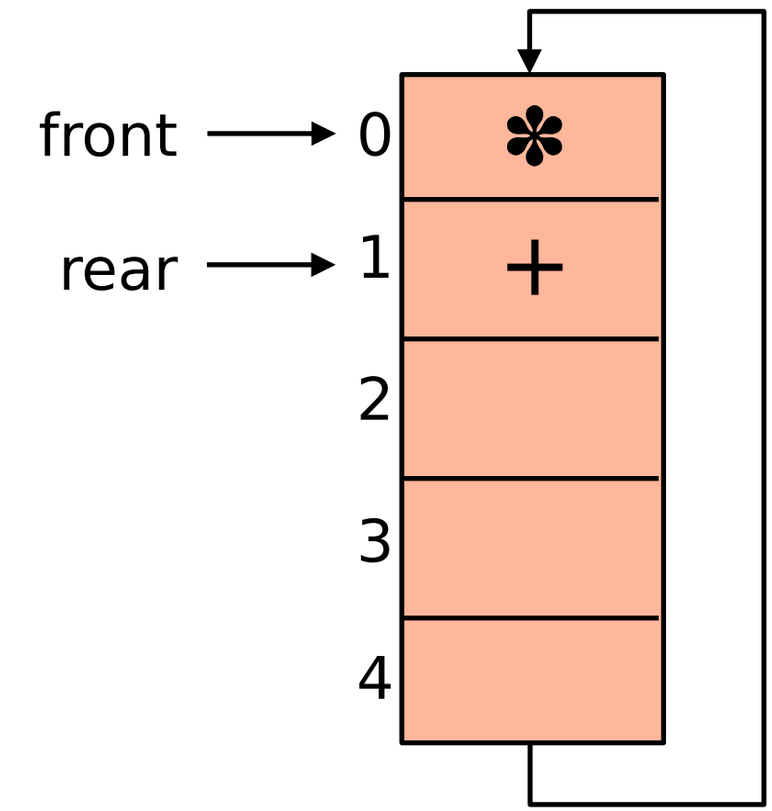
size = 1

capacity = 5

```
ArrayQueue q = new ArrayQueue(5);
q.offer(”✽”);
q.offer(”+”);
q.offer(”/”);
q.offer(”–”);
q.offer(”A”);
next = q.poll();
next = q.poll();
q.offer(”B”);
q.offer(”C”);
q.offer(”D”);
```

```
public boolean offer(E item) {
        if (size == capacity)
                reallocate();
        size++;
        rear = (rear + 1) % capacity;
        theData[rear] = item;
        return true;
}
```

Exempel



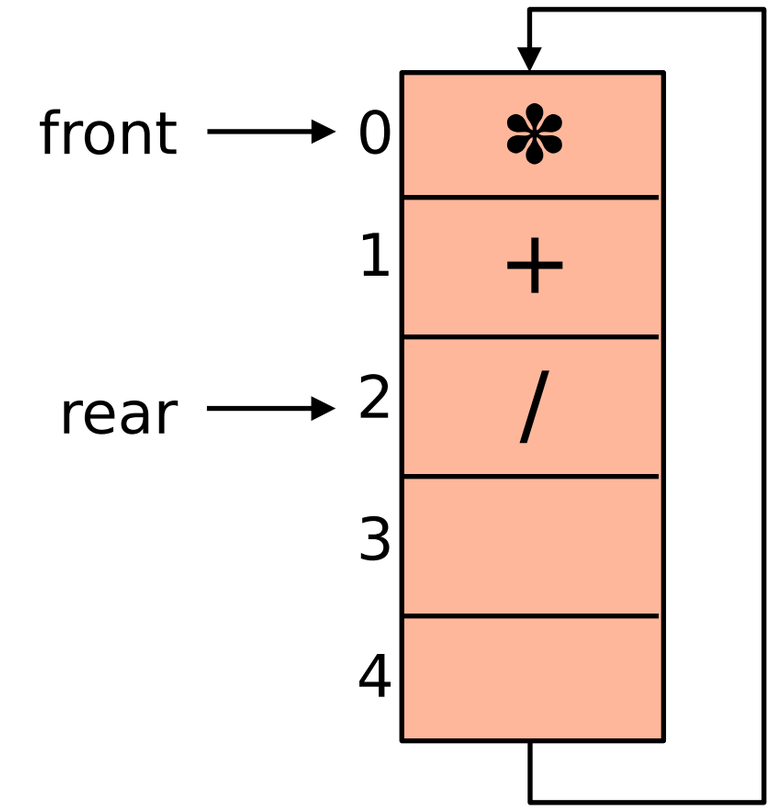
size = 2

capacity = 5

```
ArrayQueue q = new ArrayQueue(5);
q.offer(”✽”);
q.offer(”+”);
q.offer(”/”);
q.offer(”–”);
q.offer(”A”);
next = q.poll();
next = q.poll();
q.offer(”B”);
q.offer(”C”);
q.offer(”D”);
```

```
public boolean offer(E item) {
        if (size == capacity)
                reallocate();
        size++;
        rear = (rear + 1) % capacity;
        theData[rear] = item;
        return true;
}
```

Exempel



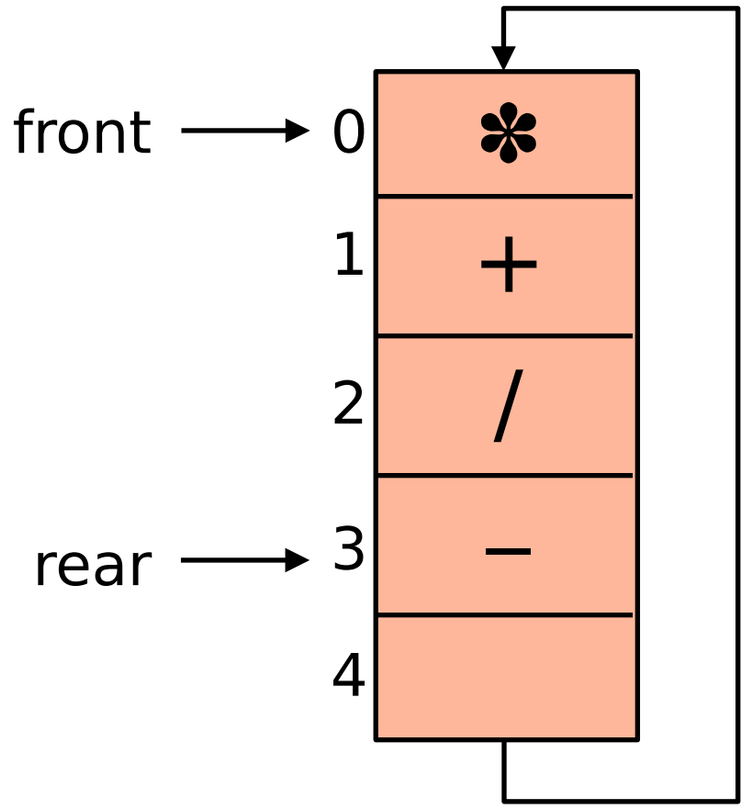
size = 3

capacity = 5

```
ArrayQueue q = new ArrayQueue(5);
q.offer(”✽”);
q.offer(”+”);
q.offer(”/”);
q.offer(”–”);
q.offer(”A”);
next = q.poll();
next = q.poll();
q.offer(”B”);
q.offer(”C”);
q.offer(”D”);
```

```
public boolean offer(E item) {
        if (size == capacity)
                reallocate();
        size++;
        rear = (rear + 1) % capacity;
        theData[rear] = item;
        return true;
}
```

Exempel



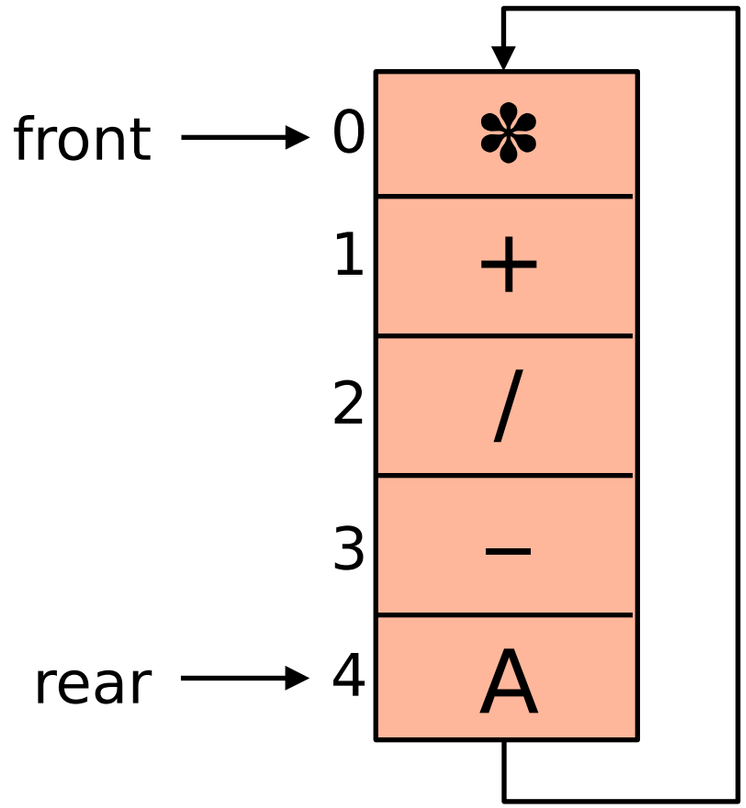
size = 4

capacity = 5

```
ArrayQueue q = new ArrayQueue(5);
q.offer(”✽”);
q.offer(”+”);
q.offer(”/”);
q.offer(”–”);
q.offer(”A”);
next = q.poll();
next = q.poll();
q.offer(”B”);
q.offer(”C”);
q.offer(”D”);
```

```
public boolean offer(E item) {
        if (size == capacity)
                reallocate();
        size++;
        rear = (rear + 1) % capacity;
        theData[rear] = item;
        return true;
}
```

Exempel



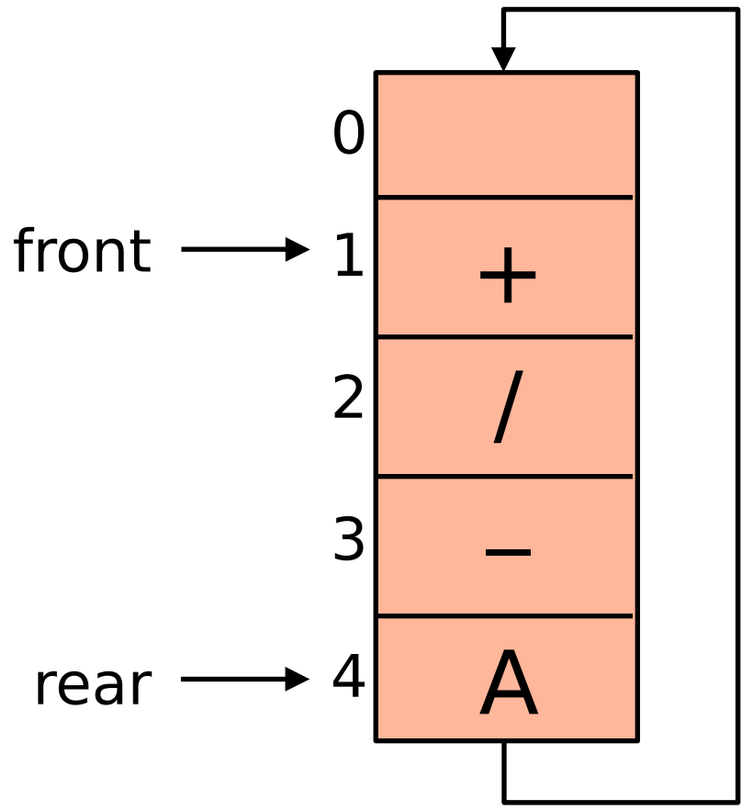
size = 5

capacity = 5

```
ArrayQueue q = new ArrayQueue(5);
q.offer(”✽”);
q.offer(”+”);
q.offer(”/”);
q.offer(”–”);
q.offer(”A”);
next = q.poll();
next = q.poll();
q.offer(”B”);
q.offer(”C”);
q.offer(”D”);
```

```
public boolean offer(E item) {
        if (size == capacity)
                reallocate();
        size++;
        rear = (rear + 1) % capacity;
        theData[rear] = item;
        return true;
}
```

Exempel



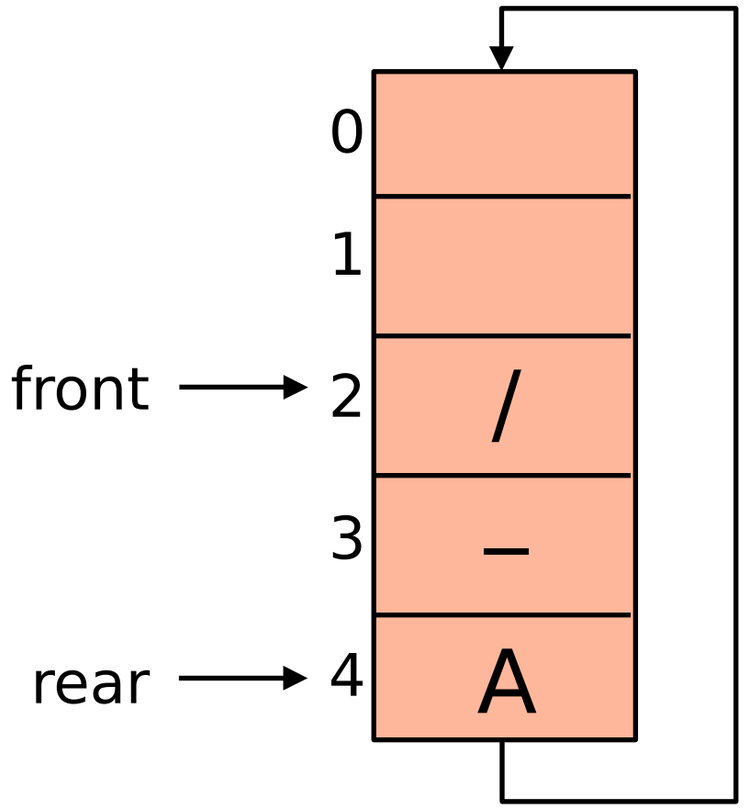
size = 4

capacity = 5

```
ArrayQueue q = new ArrayQueue(5);
q.offer(”✽”);
q.offer(”+”);
q.offer(”/”);
q.offer(”–”);
q.offer(”A”);
next = q.poll();
next = q.poll();
q.offer(”B”);
q.offer(”C”);
q.offer(”D”);
```

```
public E poll() {
        if (size == 0)
                return null;
        E result = theData[front];
        front = (front + 1) % capacity;
        size--;
        return result;
}
```

Exempel



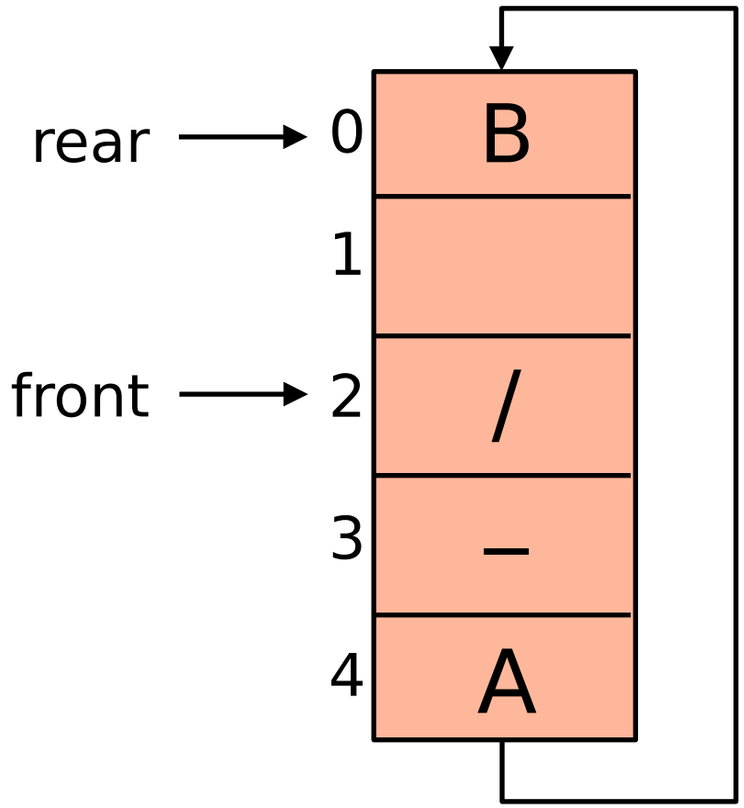
size = 3

capacity = 5

```
ArrayQueue q = new ArrayQueue(5);
q.offer(”✽”);
q.offer(”+”);
q.offer(”/”);
q.offer(”–”);
q.offer(”A”);
next = q.poll();
next = q.poll();
q.offer(”B”);
q.offer(”C”);
q.offer(”D”);
```

```
public E poll() {
        if (size == 0)
                return null;
        E result = theData[front];
        front = (front + 1) % capacity;
        size--;
        return result;
}
```

Exempel



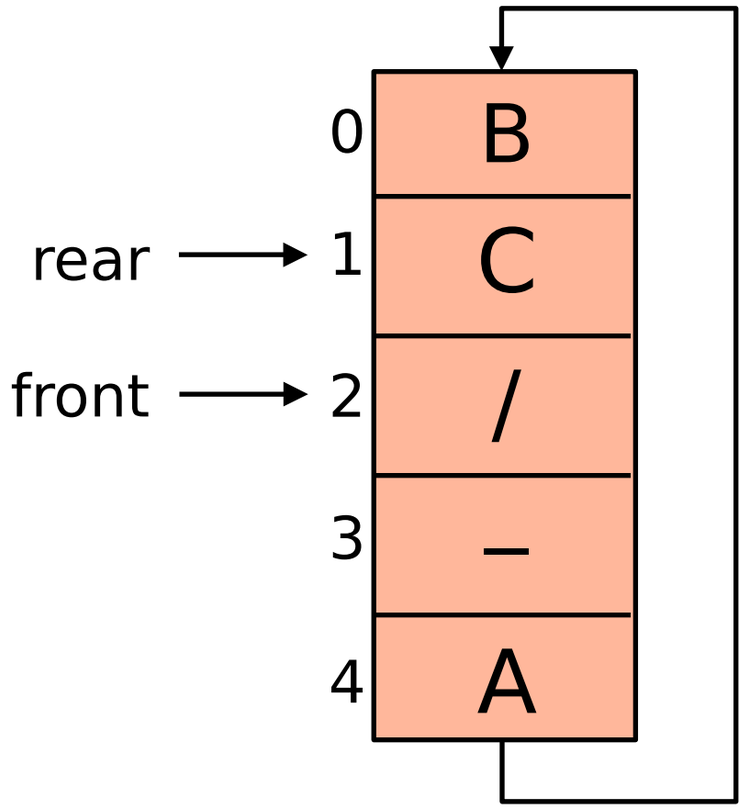
size = 4

capacity = 5

```
ArrayQueue q = new ArrayQueue(5);
q.offer(”✽”);
q.offer(”+”);
q.offer(”/”);
q.offer(”–”);
q.offer(”A”);
next = q.poll();
next = q.poll();
q.offer(”B”);
q.offer(”C”);
q.offer(”D”);
```

```
public E poll() {
        if (size == 0)
                return null;
        E result = theData[front];
        front = (front + 1) % capacity;
        size--;
        return result;
}
```

Exempel



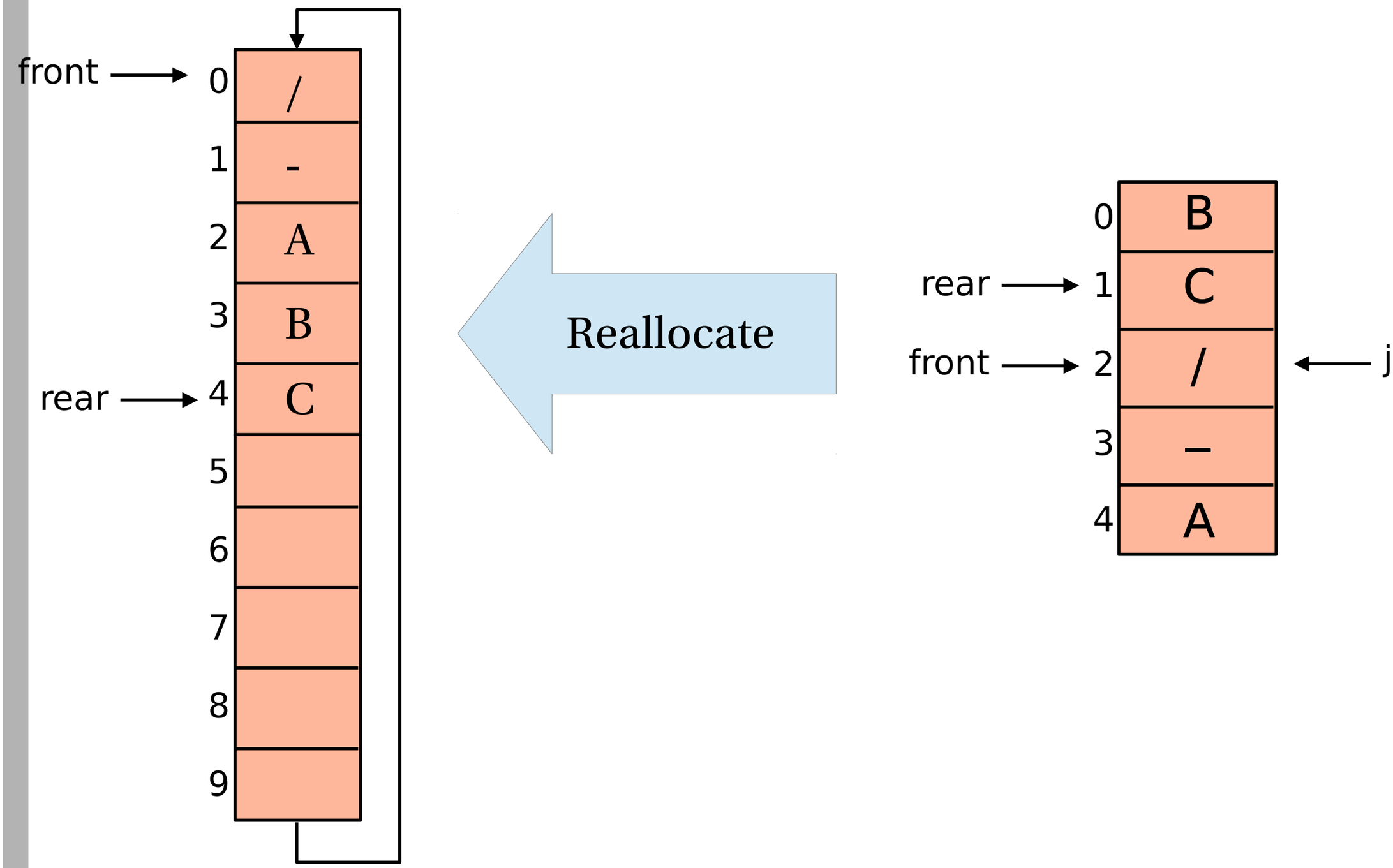
size = 4

capacity = 5

```
ArrayQueue q = new ArrayQueue(5);
q.offer(”✽”);
q.offer(”+”);
q.offer(”/”);
q.offer(”–”);
q.offer(”A”);
next = q.poll();
next = q.poll();
q.offer(”B”);
q.offer(”C”);
q.offer(”D”);
```

```
public E poll() {
        if (size == 0)
                return null;
        E result = theData[front];
        front = (front + 1) % capacity;
        size--;
        return result;
}
```

Exempel


front
0
/
1
-
2
A
3
B
rear
4
C
5
6
7
8
9
Reallocate
rear
0
B
1
C
front
2
/
j
3
–
4
A

Deque (double-ended queue)

En deque är både en stack och en kö, samtidigt:

- addFirst(E), addLast(E)
- pollFirst(), pollLast()

java.util.Deque är ett gränssnitt; det finns två färdiga implementeringar:

- java.util.ArrayDeque<E>
- java.util.LinkedList<E>

java.util.Deque

Method	Behavior
`boolean offerFirst(E item)`	Inserts `item` at the front of the deque. Returns **`true`** if successful; returns **`false`** if the item could not be inserted.
`boolean offerLast(E item)`	Inserts `item` at the rear of the deque. Returns **`true`** if successful; returns **`false`** if the item could not be inserted.
`void addFirst(E item)`	Inserts `item` at the front of the deque. Throws an exception if the item could not be inserted.
`void addLast(E item)`	Inserts `item` at the rear of the deque. Throws an exception if the item could not be inserted.
`E pollFirst()`	Removes the entry at the front of the deque and returns it; returns **`null`** if the deque is empty.
`E pollLast()`	Removes the entry at the rear of the deque and returns it; returns **`null`** if the deque is empty.
`E removeFirst()`	Removes the entry at the front of the deque and returns it if the deque is not empty. If the deque is empty, throws a `NoSuchElementException`.
`E removeLast()`	Removes the item at the rear of the deque and returns it. If the deque is empty, throws a `NoSuchElementException`.
`E peekFirst()`	Returns the entry at the front of the deque without removing it; returns **`null`** if the deque is empty.
`E peekLast()`	Returns the item at the rear of the deque without removing it; returns **`null`** if the deque is empty.
`E getFirst()`	Returns the entry at the front of the deque without removing it. If the deque is empty, throws a `NoSuchElementException`.
`E getLast()`	Returns the item at the rear of the deque without removing it. If the deque is empty, throws a `NoSuchElementException`.
`boolean removeFirstOccurrence(Object item)`	Removes the first occurrence of item in the deque. Returns **`true`** if the item was removed.
`boolean removeLastOccurrence(Object item)`	Removes the last occurrence of item in the deque. Returns **`true`** if the item was removed.
`Iterator<E> iterator()`	Returns an iterator to the elements of this deque in the proper sequence.
`Iterator<E> descendingIterator()`	Returns an iterator to the elements of this deque in reverse sequential order.

Exempel på deque-metoder:

Deque Method	Deque d	Effect
`d.offerFirst('b')`	b	'b' inserted at front
`d.offerLast('y')`	by	'y' inserted at rear
`d.addLast('z')`	byz	'z' inserted at rear
`d.addFirst('a')`	abyz	'a' inserted at front
`d.peekFirst()`	abyz	Returns 'a'
`d.peekLast()`	abyz	Returns 'z'
`d.pollLast()`	aby	Removes 'z'
`d.pollFirst()`	by	Removes 'a'

Functional stacks

```
type Stack a = ???
push :: a → Stack a → Stack a
pop :: Stack a → Stack a
peek :: Stack a → a
empty :: Stack a → Bool
```

Really easy!

```
type Stack a = [a]
push :: a → Stack a → Stack a
push x xs = x:xs

pop :: Stack a → Stack a
pop (x:xs) = xs

peek :: Stack a → a
peek (x:xs) = x

empty :: Stack a → Bool
empty [] = True
empty (x:xs) = False
```

Functional queues

```
type Queue a = ???
enqueue :: a → Queue a → Queue a
dequeue :: Queue a → (a, Queue a)
empty :: Queue a → Bool
```

One implementation...

```
type Queue a = [a]
enqueue :: a → Queue a → Queue a
enqueue x xs = xs ++ [x]

dequeue :: Queue a → (a, Queue a)
dequeue (x:xs) = (x, xs)

empty :: Queue a → Bool
empty [] = True
empty (x:xs) = False
```

But enqueue takes O(n) time!

Solution...

Represent a queue by a *pair* of lists (`xs`, `ys`)

- `xs`: elements that we can dequeue
- `ys`: elements that we have enqueued, *in reverse order*

For example, `([1,2,3],[6,5,4])` would represent the queue 1 2 3 4 5 6

Enqueue adds an element to `ys`, dequeue removes an element from `xs`

But this cannot possibly work – how do elements get from ys to xs?

Solution...

If we dequeue `([], [6,5,4])`, there is no element to remove from `xs`, but the queue is supposed to contain 4 5 6!

If `xs` is empty when we dequeue, reverse `ys` and replace `xs` with it...

([4,5,6], [])

...then perform the dequeue as normal:

([5,6], [])

The implementation

```
type Queue a = ([a], [a])
enqueue :: a → Queue a → Queue a
enqueue x (xs, ys) = (xs, x:ys)

dequeue :: Queue a → (a, Queue a)
dequeue (x:xs, ys) = (x, (xs, ys))
dequeue ([], ys) = (x, (xs, []))
  where x:xs = reverse ys

empty :: Queue a → Bool
empty ([],[]) = True
empty _ = False
```

The complexity?

Imagine the lifecycle of an element.

It starts by being enqueued (O(1) time).

Eventually it is dequeued (O(1) time).

In the middle it must be moved from the second list to the first list. This takes O(n) time if the second list has n elements – O(1) time per element.

We can say that dequeue is *amortised* O(1) time – some dequeues may take longer, if we have to reverse the list, but n operations take O(n) time.

Stacks and queues – summary

Both widely used and can be implemented efficiently

- Both easy to implement in an imperative language with linked lists or dynamic arrays
- Stacks easy to implement in a functional language, queues a bit harder and with O(1) *amortised* complexity – a bit less efficient

Both special cases of a deque:

- Stack: insert at beginning, remove from beginning
- Queue: insert at beginning, remove from end