Hash tables

(19.1 – 19.3, 19.5 – 19.6)

Hash tables naively

A hash table implements a set or map

The plan: take an array of some 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 naively, 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}

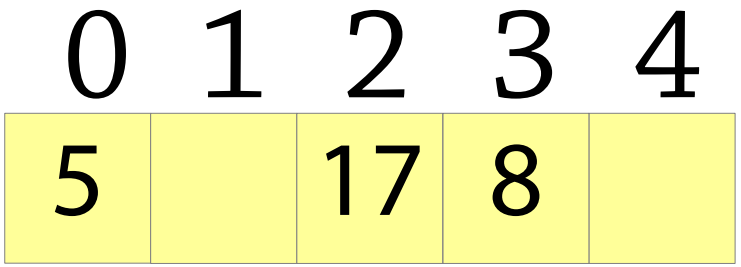
Inserting 14 gives:

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



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

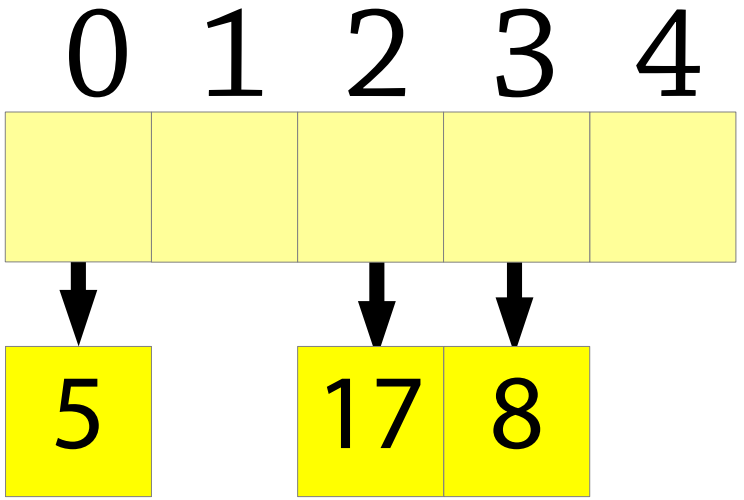
This is called a *collision*

Real hash tables are naive hash tables plus tricks for dealing with and avoiding collisions!

Handling collisions: chaining

Instead of an array of elements, have an array of *linked lists* (chains)

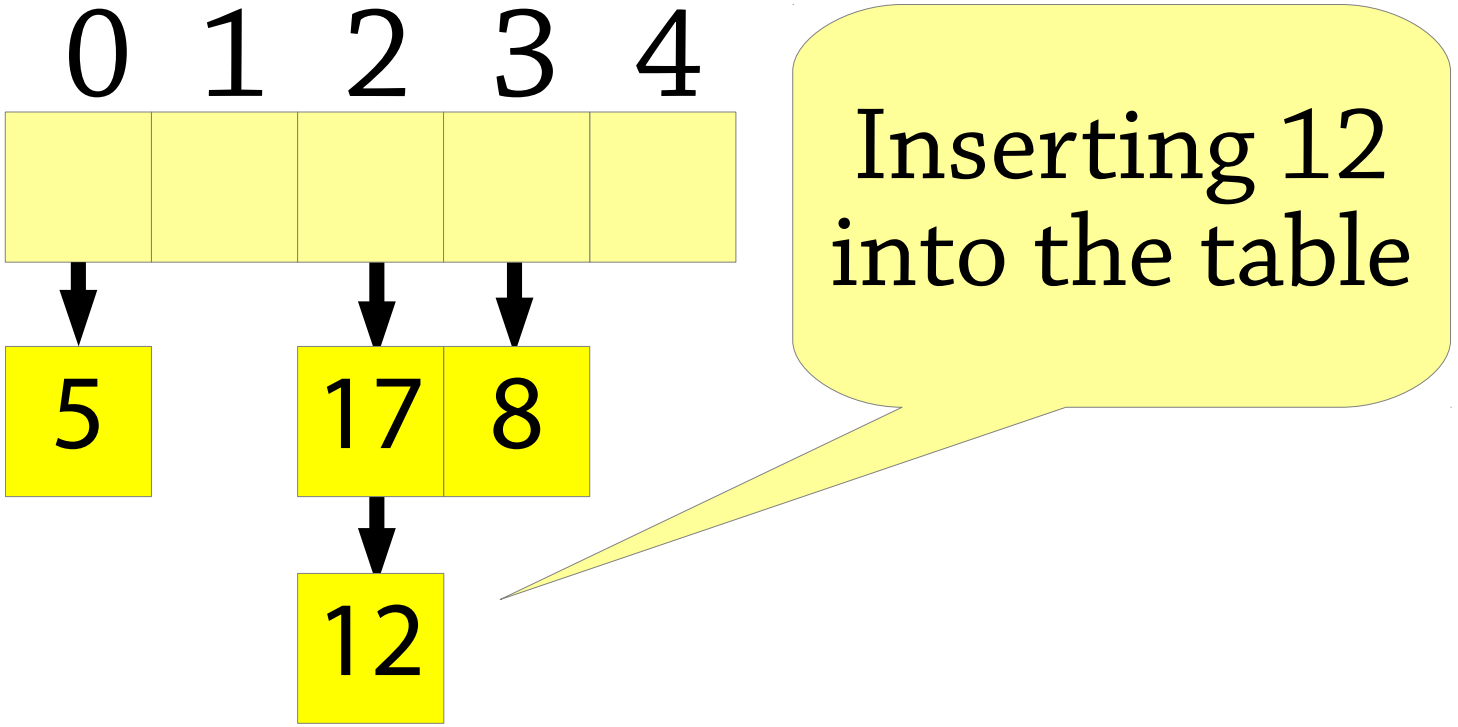
To add an element, calculate its hash and insert it into the list at that index



Handling collisions: chaining

Instead of an array of elements, have an array of *linked lists* (chains)

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



Performance of chained hash tables

Chained hash tables are fast if the chains are small

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

But if the chains get big, everything gets slow

- Can degrade to O(n) in the worst case

There are two cases when this can happen!

Performance of chained hash tables

Case one: the hash table is too full

- If we try to store 1,000,000 values in an array of size 5, some chains will be 200,000 long

Solution: expand the hash table

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

Problem: h(x) 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
- Hash function of an integer will just be the integer itself

Performance of chained hash tables

Case two: the hash function is lousy

- Worst case: h(x) is a constant function, e.g. h(x) = 0
- Then all elements will end up in the same chain!
- h(x) needs to distribute values evenly

One helpful trick: make the hash table size always be a prime number

- If h(x) always returns an even number, and the hash table size is even, then the odd-numbered array indexes will never be used!
- In general: same problem if h(x) is too often divisible by *n*, and hash table size is divisible by *n*
- Using a prime number reduces the chance of this happening

Designing hash functions

A good hash function should distribute values evenly

- h(x) has a roughly equal chance of being any particular number
- That way, all chains will be roughly the same length!
- Also, similar values should not have similar hash codes

Defining good hash functions is a black art!

- Weird heuristics that are semi-backed-up by theory

Defining a good hash function

What is bad about 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)

Any anagrams will have the same hash code

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:

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

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!

An analogy

Suppose we want to define a hash function for lists of digits from 0-9:

- [0,9,3,4,2,1] etc.

Idea: write out the digits as a single number with a leading 1:

- hash([0,9,3,4,2,1]) = 1093421

(Without the leading 1 we would get the same hash for e.g. [0,1] and [1])

The hash function on strings is doing exactly this, only working in base 128 instead of base 10

The problem

For performance, we will calculate the hash using machine integers so the calculation

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

will happen modulo 2^{32} (*integer overflow*)

So the hash will only use the last few characters!

Solution: replace 128 with another number, e.g. 33

$$33^n + s_0 \cdot 33^{n-1} + s_1 \cdot 33^{n-2} + \ldots + s_{n-1}$$

This is (almost) what Java uses for strings

Hashing composite values

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

Use the same approach as for strings!

$$33^2 + 33 \times h(a) + h(b)$$

This comes out quite nicely in code too:

```
int hash = 1;
hash = hash*33 + a.hashCode();
hash = hash*33 + b.hashCode();
```

Hash functions

This is called *Bernstein hashing*, it's only one way of defining hash functions

- Bernstein discovered that using 33 as the constant gives good distribution
- Why? Nobody knows!

Many hash functions are inspired by random number generation algorithms

- The output of a good hash function should look random so there are many similarities

Often pretty ad hoc!

- Lots of experimentation involved

Linear probing

Another way of dealing with collisions is *linear probing*

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

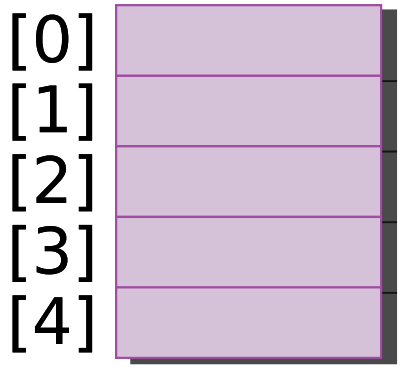
If you want to store a value at index i but it's full, store it in index $i+1$ instead!

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

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

Example of linear probing

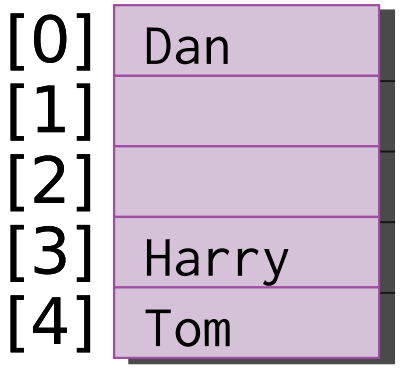
```
Tom  Dan  Harry  Sam  Pete
```



Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Example of linear probing

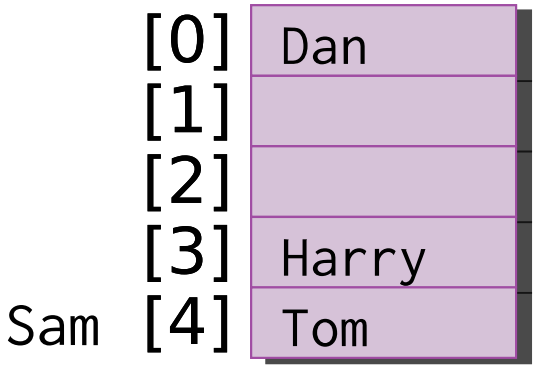
Sam Pete



Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Example of linear probing

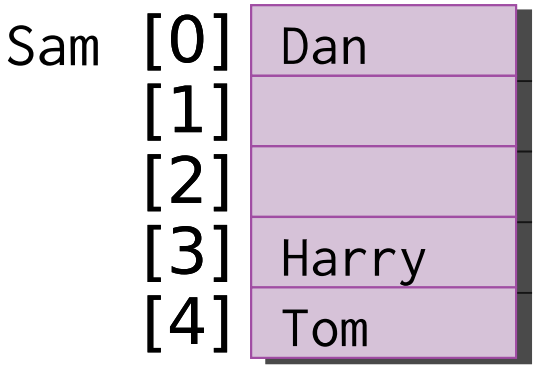
Pete



Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Example of linear probing

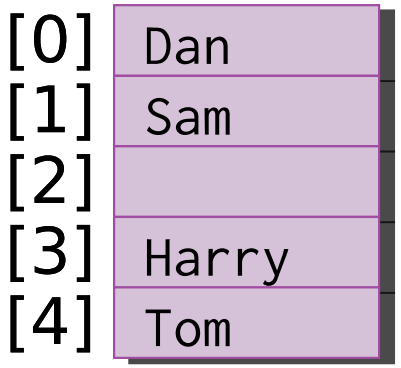
Pete



Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

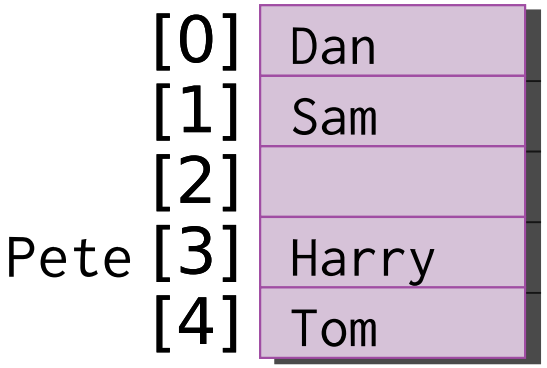
Example of linear probing

Pete



Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

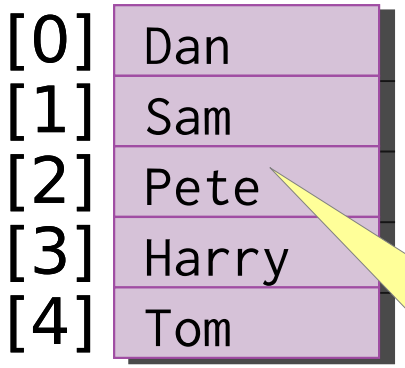
Example of linear probing



Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Example of linear probing

Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"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:

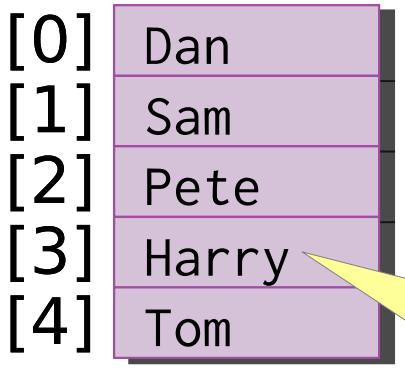
- Calculate the hash of the element, *i*
- Look at *array[i]*
- If it's the right element, return it!
- If there's no element there, fail
- If there's a *different* element there, search again at index *(i+1) % array.size*

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

Deleting with linear probing

Can't just remove an element...

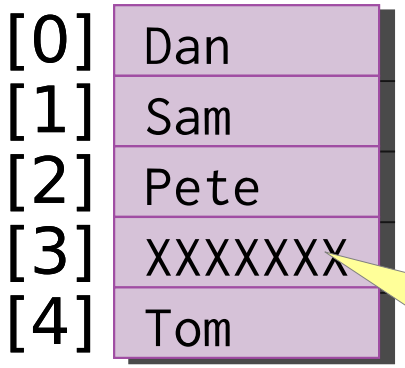
Name	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3



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	Hash	Hash % 5
"Tom"	84274	4
"Dan"	68465	0
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

The search algorithm should skip over XXXXXXX

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*

Naive 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 many values with the same hash, values with that hash become slower to access but other hashes are unaffected

In linear probing, you get a cluster and values with *nearby* hashes become slower to access too!

As the array gets close to 100% full, you get very long clusters in the hash table and performance becomes dreadful

Linear probing needs a much bigger array than linear chaining for the same performance

But: as you don't need to also create list nodes, you can create a bigger array in the same amount of memory

Probing vs 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
  - Other alternatives to linear probing, e.g. quadratic probing
  - Some sort of probing seems to be fastest

In return:

- *Expected* (average) O(1) performance if the hash function is random (there are no patterns)
- Better performance in practice than BSTs
- Disadvantage: hash tables are *unordered* so you can't get the elements in increasing order

Theoretical foundations a little shaky with common hash functions, but very good practical performance.

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*

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)

A recursive function

```
void rec(int n) {
  if (n > 0) {
    System.out.println(n);
    rec(n-1);
    rec(n-1);
  }
}
```

How much memory does this function use?

Don't forget to include the call stack!

A recursive function

```
void rec(int n) {
  if (n > 0)
    System.
    rec(n-1
    rec(n-1
  }
}
```

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!

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!

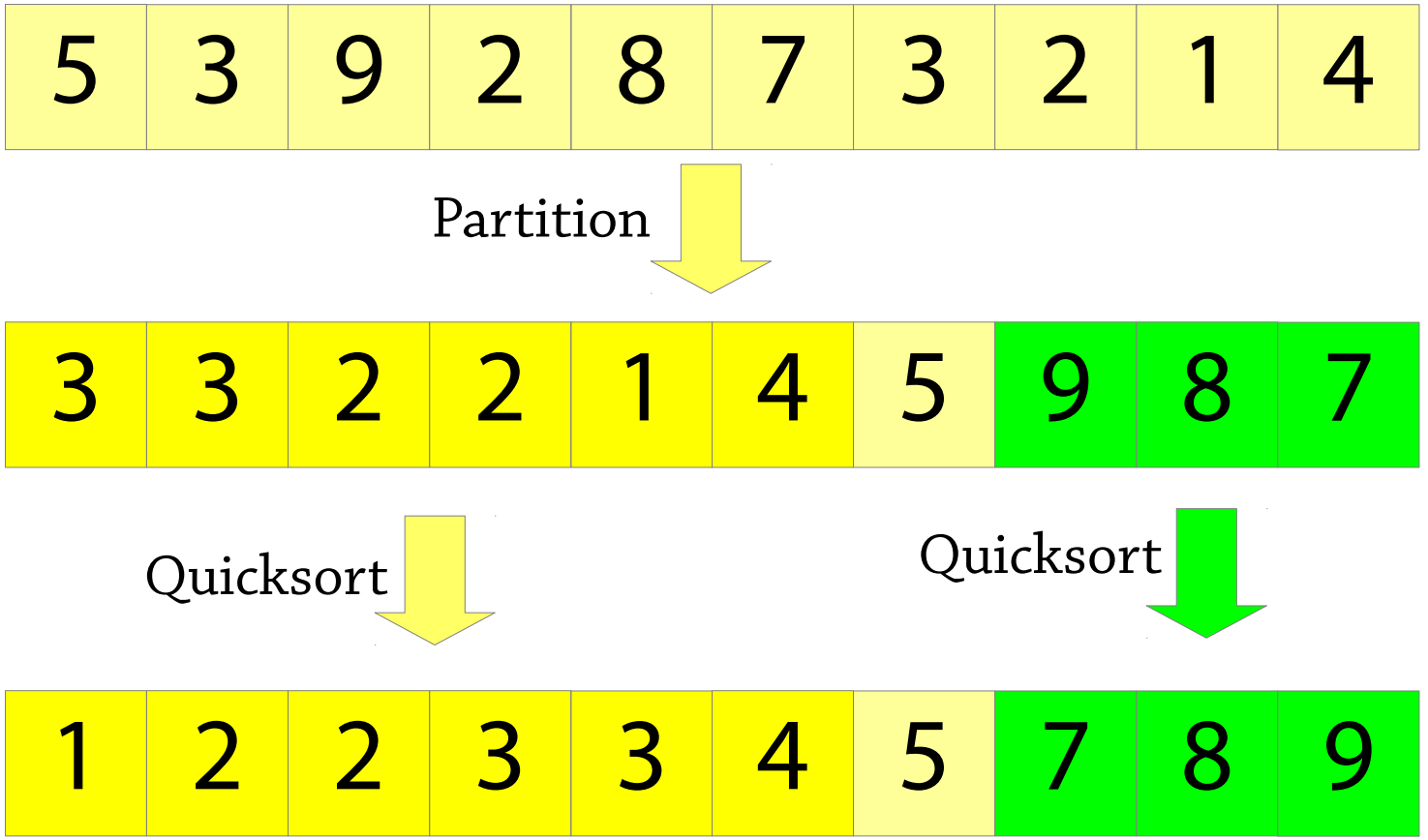
Tail calls

A tail-recursive function (one where all recursive calls are tail calls) takes **O(1)** stack space, in a language with TCO

If a function has a mixture of tail and non-tail calls, the amount of stack space is **O(maximum depth of non-tail recursive calls)**

In languages without TCO, you can transform tail recursion into a loop by hand to save 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
}
```

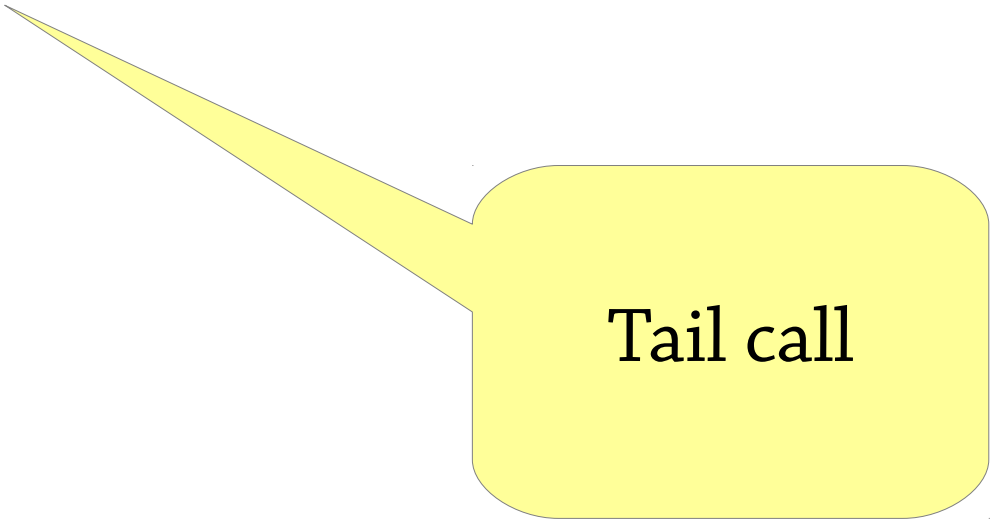
O(n), including the call stack!

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: always sort the *smaller* partition first, and the bigger partition second

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;
   }
   }
}
```