Datastructures

Data Structures

- Datatype
  - A model of something that we want to represent in our program
- Data structure
  - A particular way of *storing* data
  - How? Depending on what we want to do with the data
- Today: Two examples
  - Queues
  - Tables

Another Datastructure: Tables

A *table* holds a collection of *keys* and associated *values*.

For example, a phone book is a table whose keys are names, and whose values are telephone numbers.

Problem: Given a table and a key, find the associated value.

John Hughes	1001
Mary Sheeran	1013
Koen Claessen	5424
Hans Svensson	1079

Table Lookup Using Lists

Since a table may contain any kind of keys and values, define a parameterised type:

E.g. [(”x”,1), (”y”,2)] :: Table String Int

type Table k v = [(k, v)]

lookup ”y” … $\longrightarrow$ Just 2

lookup :: Eq k => k -> Table k v -> Maybe v

lookup ”z” ... $\longrightarrow$ Nothing

How long does it take to look up a name?

John Hughes	1001
Mary Sheeran	1013
Koen Claessen	5424
Hans Svensson	1079

If the table has **n** entries and the name is in the table then on average it takes **n/2** steps

If the name is not in the table then we always take **n** steps.

We say that it is "Order n", written
O(n) – i.e. the number of steps grows linearly as n grows.

Finding Keys Fast

Finding keys by searching from the beginning is slow!

A better method:

look somewhere in the middle, and then look backwards or forwards depending on what you find.

(This assumes the table is sorted).

Claessen?

Aaboen A	
Nilsson Hans	
Östvall Eva	

Representing Tables

We must be able to break up a table fast, into:

- A smaller table of entries before the middle one,
- the middle entry,
- a table of entries after it.

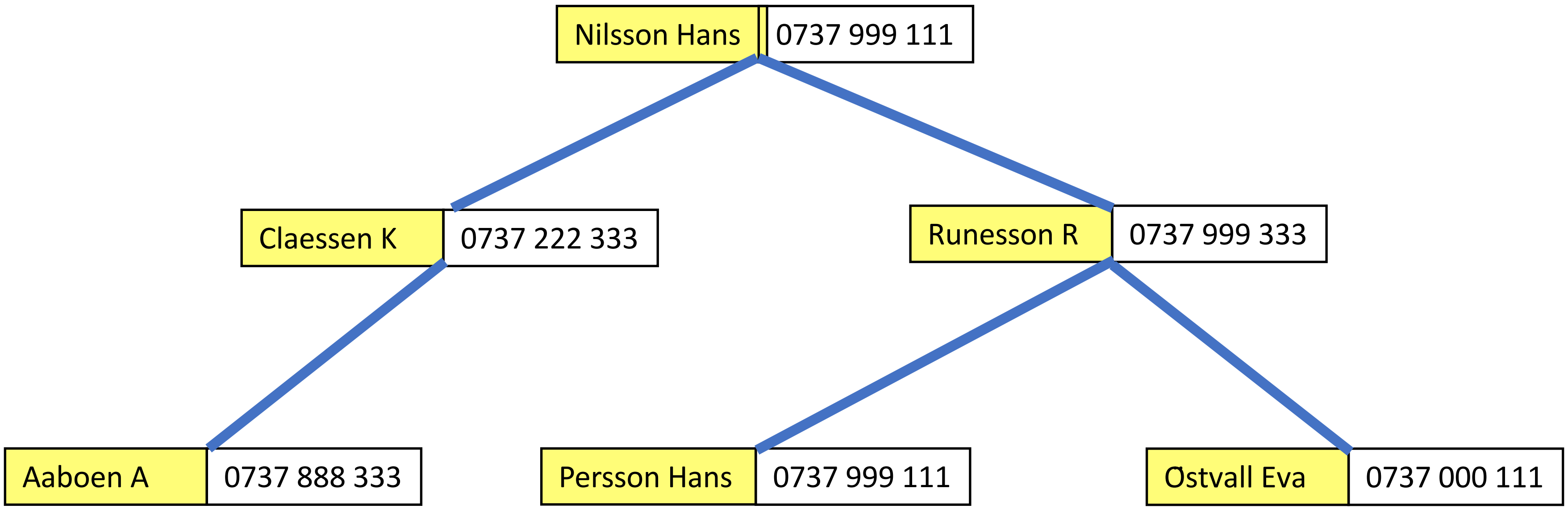
Aaboen A	

Nilsson Hans	

Östvall Eva	

data Table k v =

Join (Table k v) k v (Table k v)



Nilsson Hans
0737 999 111
Claessen K
0737 222 333
Runesson R
0737 999 333
Aaboen A
0737 888 333
Persson Hans
0737 999 111
Østvall Eva
0737 000 111

Quiz

What’s wrong with this (recursive) type?

data Table k v = Join (Table k v) k v (Table k v)

Quiz

What's wrong with this (recursive) type? No base case!

data Table k v = Join (Table k v) k v (Table k v)

| Empty

Add a base case.

Looking Up a Key

To look up a key in a table:

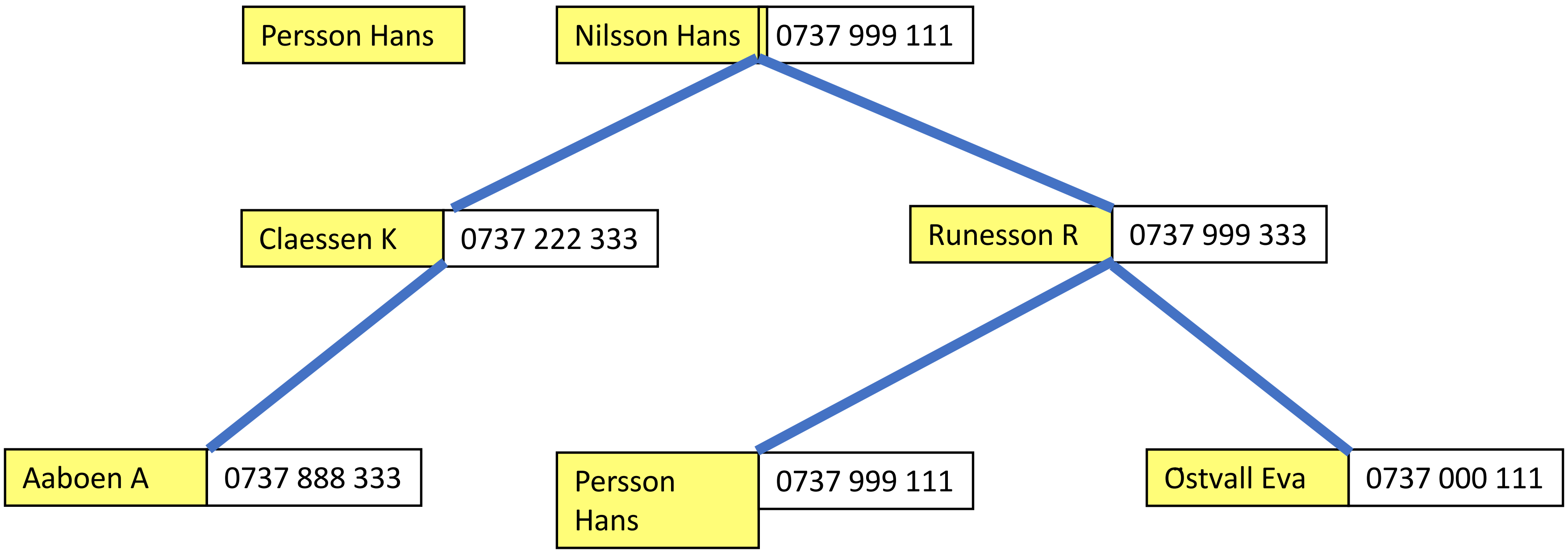
•If the table is empty, then the key is not found.

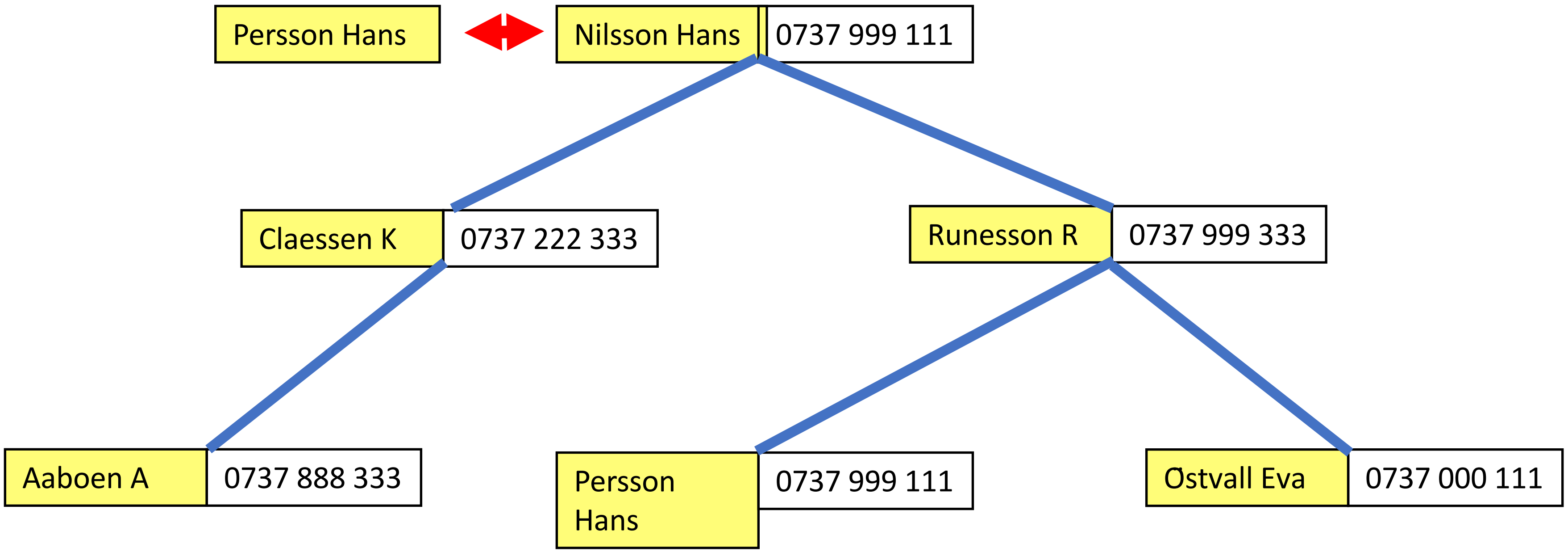
•Compare the key with the key of the middle element.

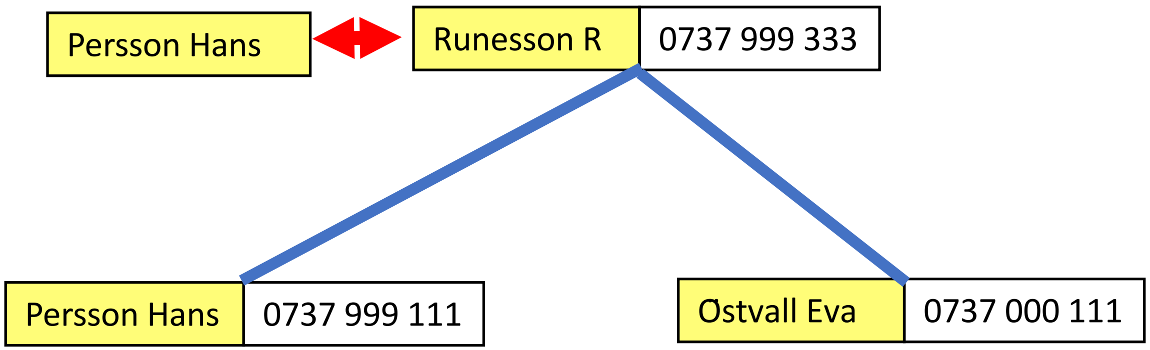
•If they are equal, return the associated value.

•If the key is less than the key in the middle, look in the first half of the table.

•If the key is greater than the key in the middle, look in the second half of the table.

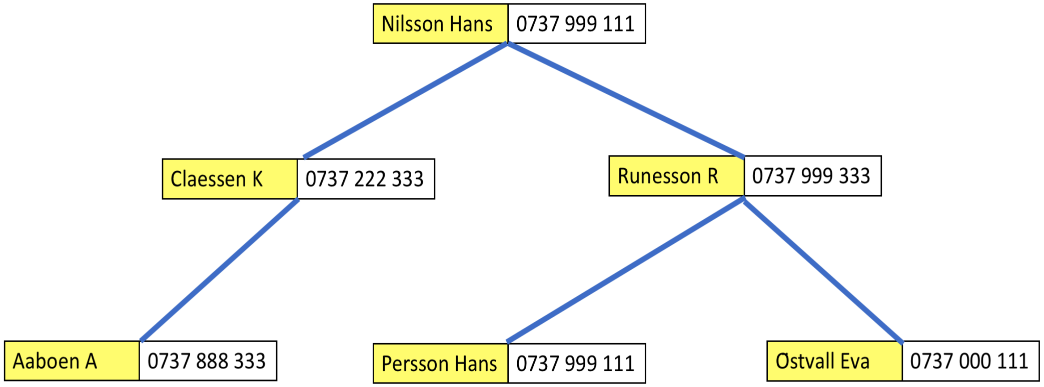

Persson Hans
Nilsson Hans
0737 999 111
Claessen K
0737 222 333
Runesson R
0737 999 333
Aaboen A
0737 888 333
Persson Hans
0737 999 111
Østvall Eva
0737 000 111


Persson Hans
Nilsson Hans
0737 999 111
Claessen K
0737 222 333
Runesson R
0737 999 333
Aaboen A
0737 888 333
Persson Hans
0737 999 111
Østvall Eva
0737 000 111


Persson Hans
Runesson R
0737 999 333
Persson Hans
0737 999 111
Østvall Eva
0737 000 111


Persson Hans
Persson Hans
0737 999 111

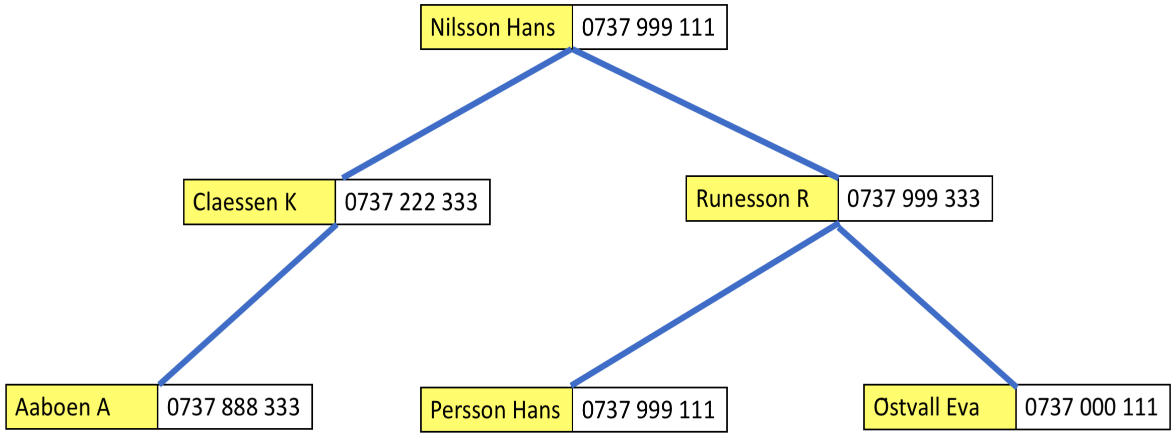
How long does it take to look up a name?



If the height of the table is **h** then it takes at most **h** steps.

How long does it take to look up a name?

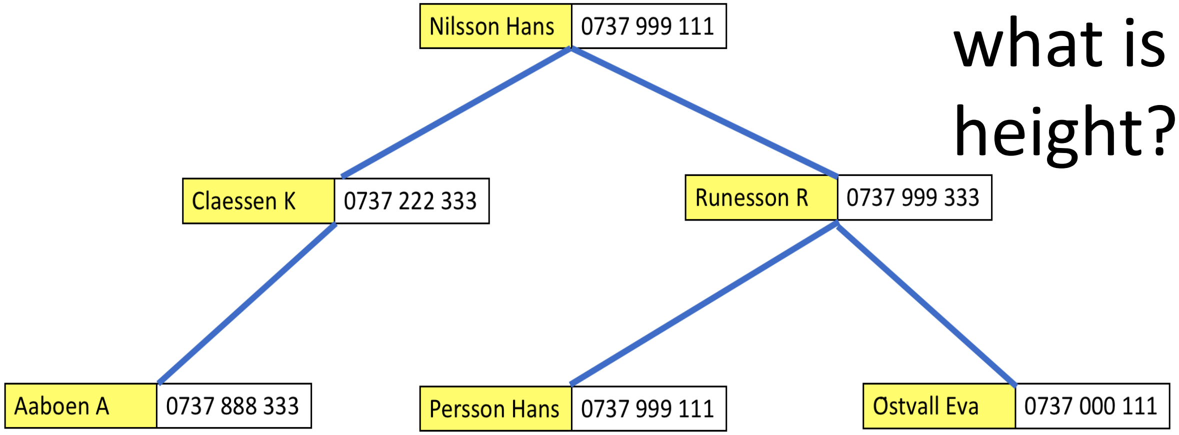
If the height of the table is **h** then it takes at most **h** steps.



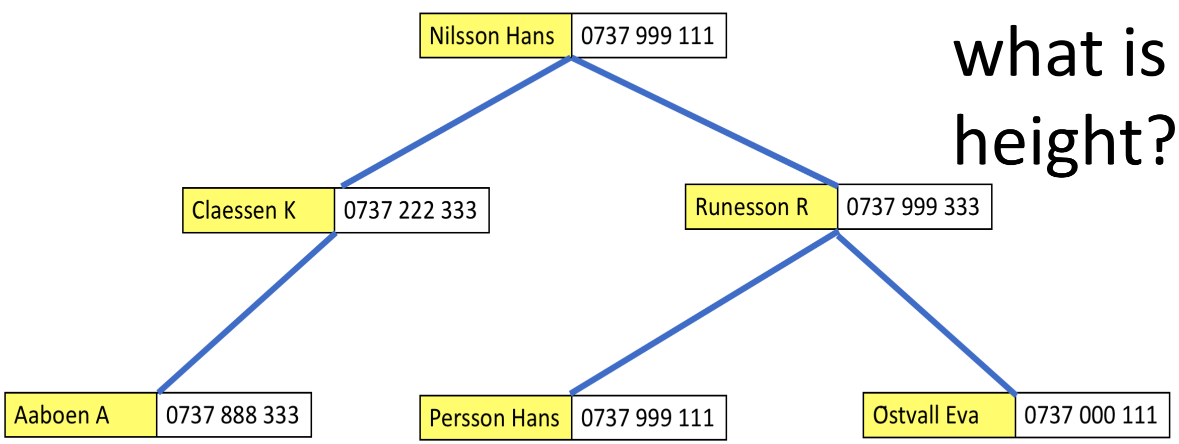
How long does it take to look up a name?

If the height of the table is **h** then it takes at most **h** steps.

If the table has **n** entries, what is the "best" height?



How long does it take to look up a name?

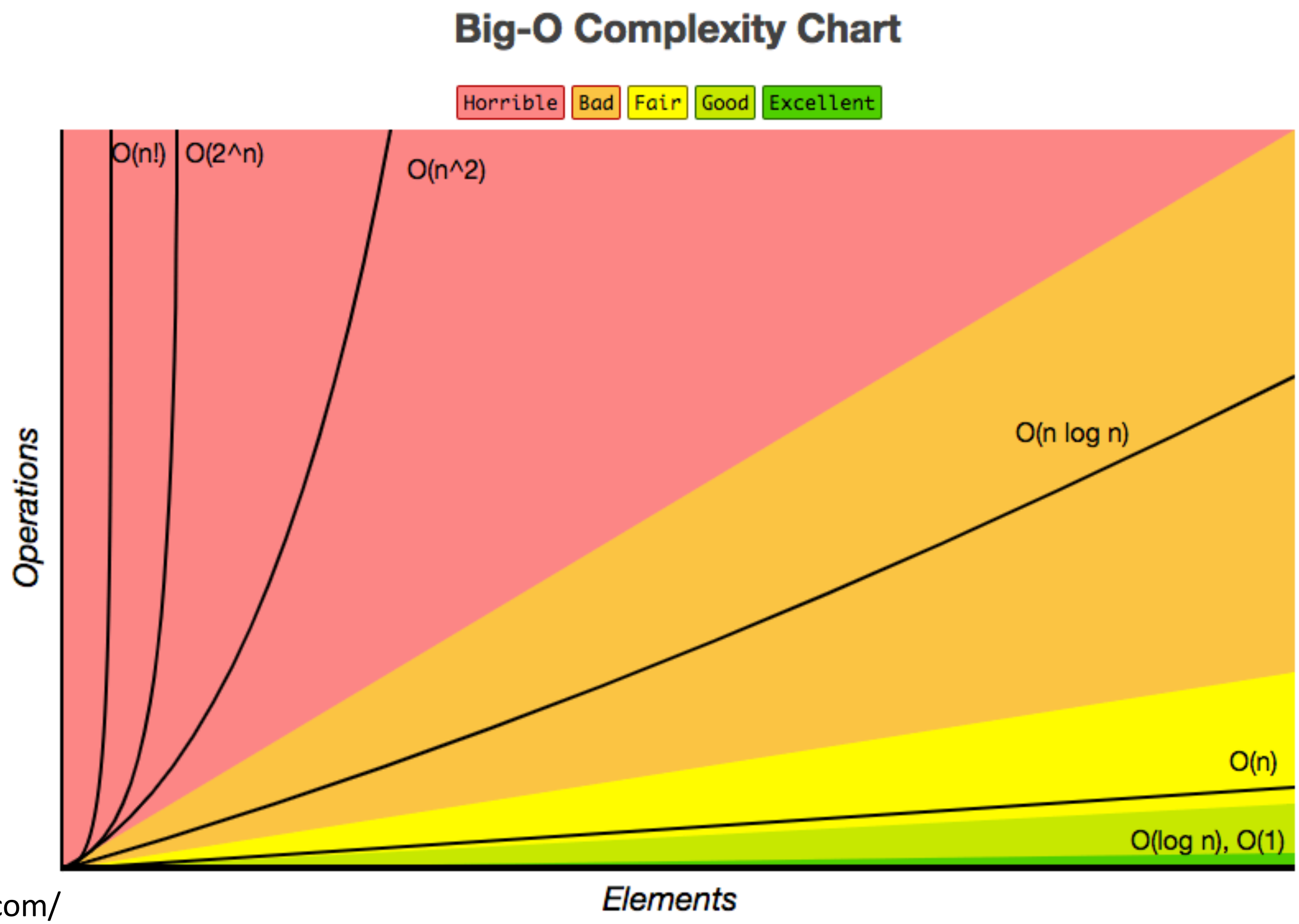


If the height of the table is **h** then it takes at most **h** steps.

If the table has **n** entries, what is the “best” height?

$\log_2 n$

O(n) vs O(log n)



http://bigocheatsheet.com/

n	Log n
100	7
1000	9
10000	14
100000	17
1000000	20
10000000	24
100000000	27

Inserting a New Key

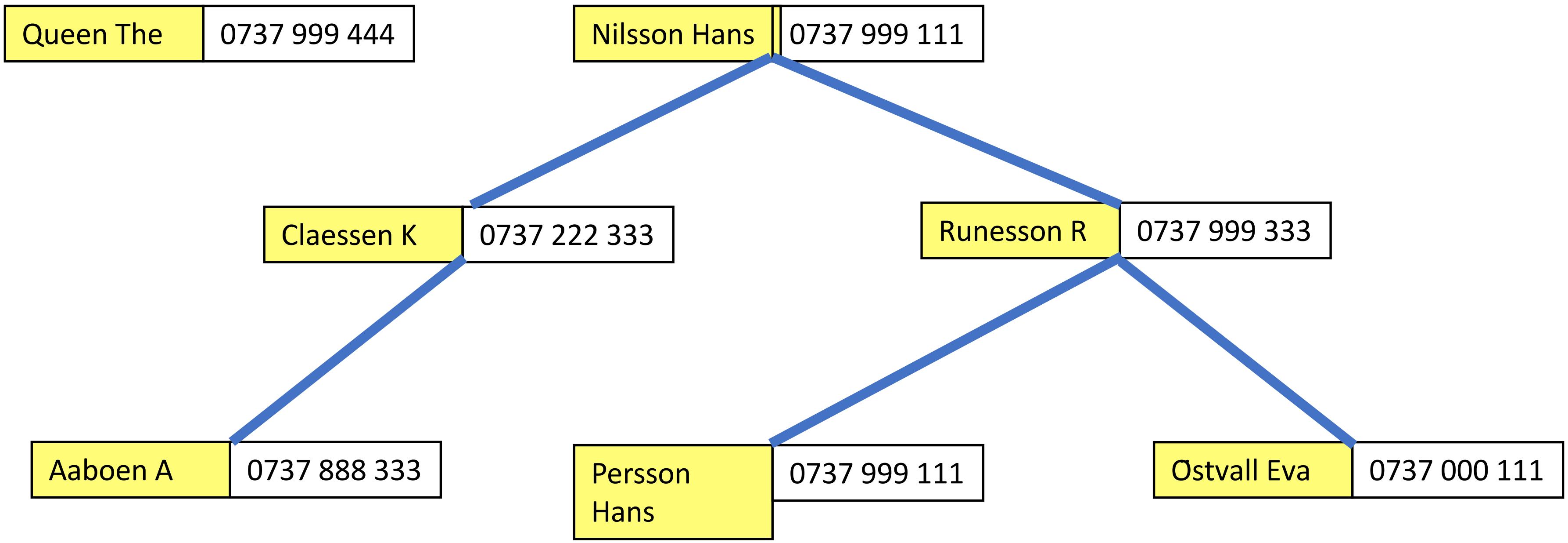
We also need a function to build tables. We define

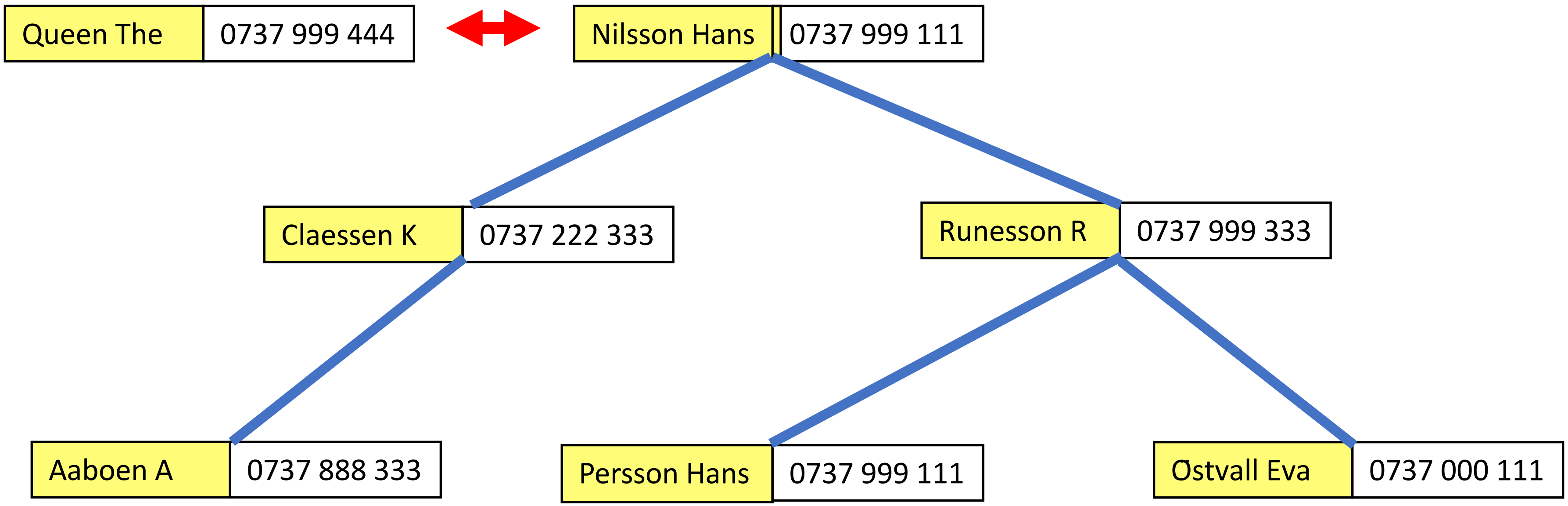
```
insertT :: Ord k => k -> v -> Table k v -> Table k v
```

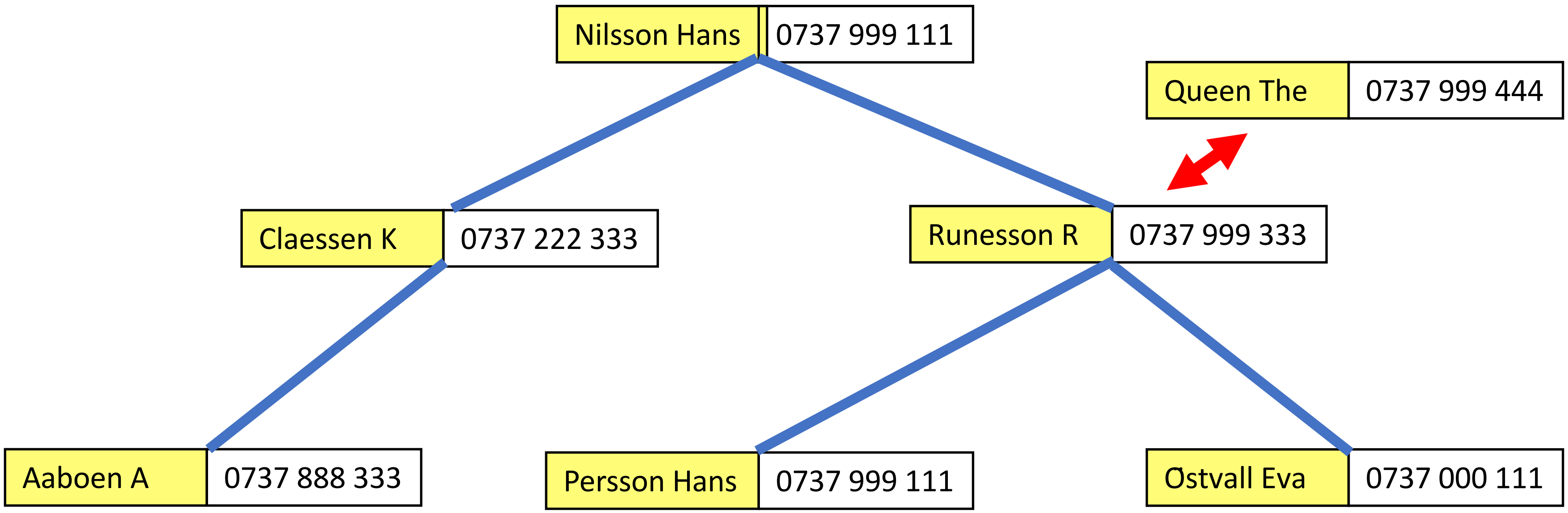
to insert a new key and value into a table.

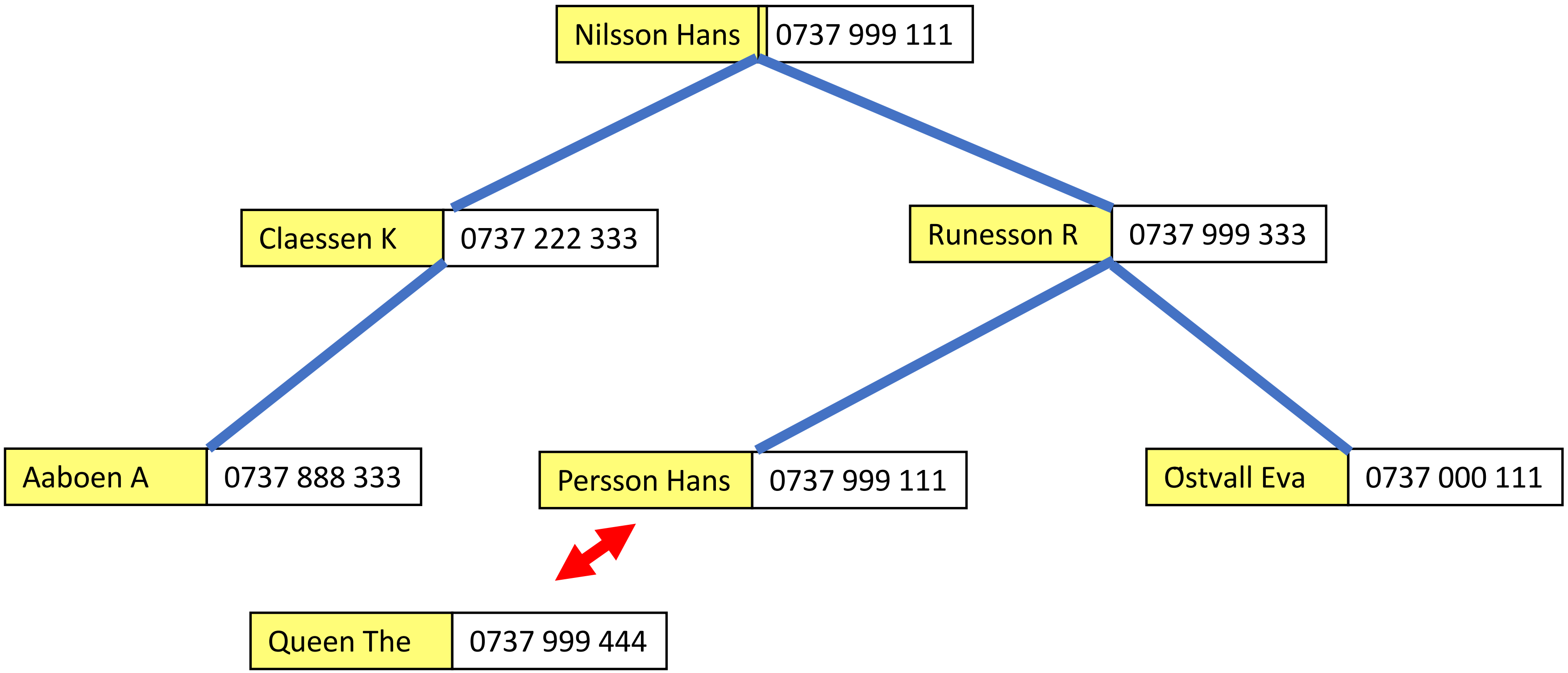
We must be careful to insert the new entry in the right place, so that the keys remain in order.

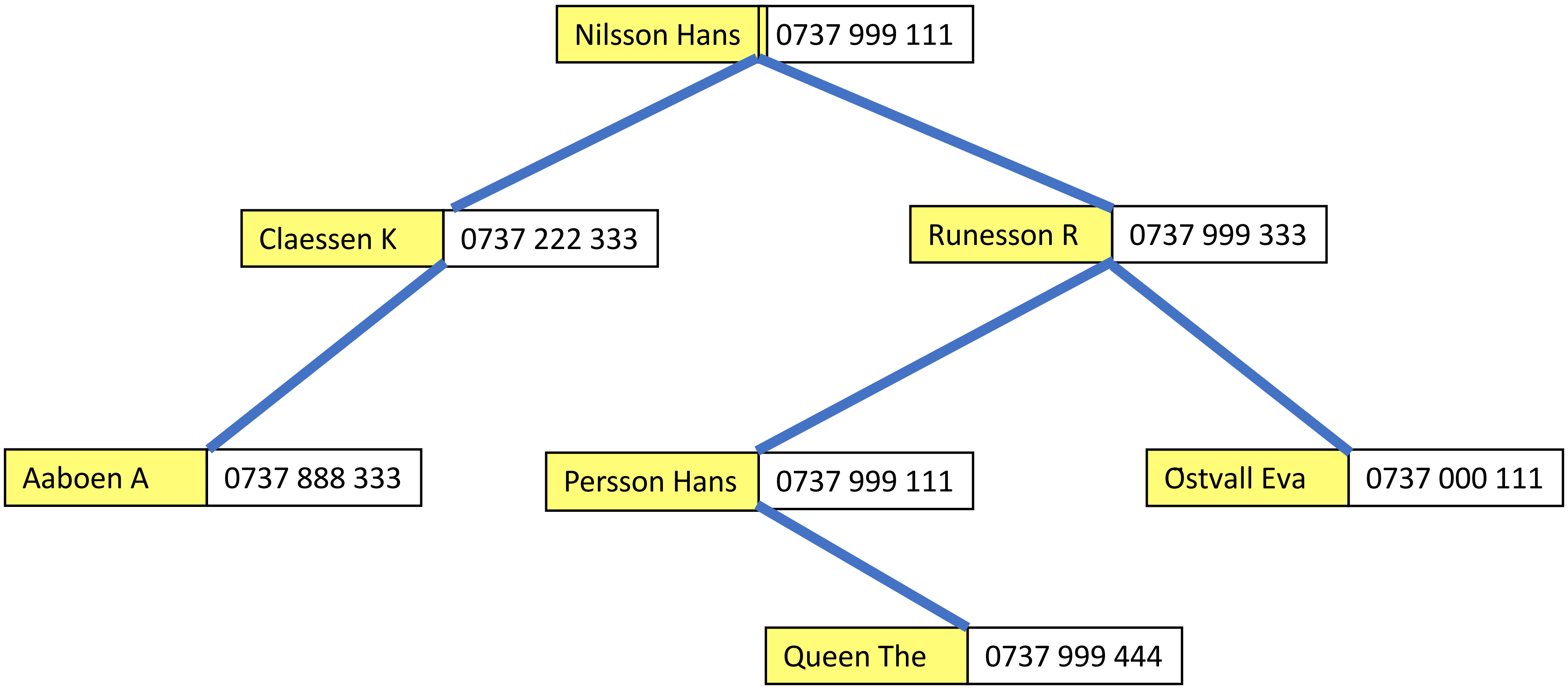
Idea: Compare the new key against the middle one. Insert into the first or second half as appropriate.


Queen The
0737 999 444
Nilsson Hans
0737 999 111
Claessen K
0737 222 333
Runesson R
0737 999 333
Aaboen A
0737 888 333
Persson Hans
0737 999 111
Østvall Eva
0737 000 111


Queen The
0737 999 444
Nilsson Hans
0737 999 111
Claessen K
0737 222 333
Runesson R
0737 999 333
Aaboen A
0737 888 333
Persson Hans
0737 999 111
Østvall Eva
0737 000 111


Nilsson Hans
0737 999 111
Queen The
0737 999 444
Claessen K
0737 222 333
Runesson R
0737 999 333
Aaboen A
0737 888 333
Persson Hans
0737 999 111
Østvall Eva
0737 000 111


Nilsson Hans
0737 999 111
Claessen K
0737 222 333
Runesson R
0737 999 333
Aaboen A
0737 888 333
Persson Hans
0737 999 111
Østvall Eva
0737 000 111
Queen The
0737 999 444


Nilsson Hans
0737 999 111
Claessen K
0737 222 333
Runesson R
0737 999 333
Aaboen A
0737 888 333
Persson Hans
0737 999 111
Østvall Eva
0737 000 111
Queen The
0737 999 444

Defining Insert

```
insertT key val Empty = Join Empty key val Empty
insertT key val (Join left k v right)
        | key <= k = Join (insertT key val left) k v right
        | key > k   = Join left k v (insertT key val right)
```

Many forget to join up the new right half with the old left half again.

Testing

- How should we test the Table operations?
  - By comparison with the list operations

```
prop_lookupT k t =
   lookupT k t == lookup k (contents t)
prop_insertT k v t =
   contents (insertT k v t) == insert (k,v) (contents t)
```

```
contents :: Table k v -> [(k,v)]
```

Generating Random Tables

- Recursive types need recursive generators

instance (Arbitrary k, Arbitrary v) =>
Arbitrary (Table k v) **where**

We can generate arbitrary Tables...

...provided we can generate keys and values

Generating Random Tables

- Recursive types need recursive generators

```
instance (Arbitrary k, Arbitrary v) =>
    Arbitrary (Table k v) where
    arbitrary = oneof [ return Empty,
                        do k <- arbitrary
                           v <- arbitrary
                           left <- arbitrary
                           right <- arbitrary
                           return (Join left k v right) ]
```

Quiz:
What is wrong with
this generator?

Controlling the Size of Tables

- Generate tables with *at most n elements*

```
table s = frequency [(1, return Empty),
                     (s, do k <- arbitrary
                            v <- arbitrary
                            l <- table (s `div` 2)
                            r <- table (s `div` 2)
                            return (Join l k v r))]
```

```
instance (Arbitrary k, Arbitrary v) =>
                     Arbitrary (Table k v) where
  arbitrary = sized table
```

Testing Table Properties

prop_lookupT k t = lookupT k t == lookup k (contents t)

Main> quickCheck prop_lookupT

Falsifiable, after 10 tests:

0

Join Empty 2 (-2) (Join Empty 0 0 Empty)

Main> contents (Join Empty 2 (-2) …)

[(2,-2),(0,0)]

What's wrong?

How to Generate Ordered Tables?

- Generate a random list,
  - Take the *first* (key,value) to be at the root
  - Take all the *smaller* keys to go in the left subtree
  - Take all the *larger* keys to go in the right subtree

Testing the Properties

- Now the invariant holds, but the properties don't!

```
Main> quickCheck prop_invTable
OK, passed 100 tests.
Main> quickCheck prop_lookupT
Falsifiable, after 7 tests:
-1
Join (Join Empty (-1) (-2) Empty) (-1) (-1) Empty
```

More Testing

```
prop_insertT k v t =
    insert (k,v) (contents t)
    == contents (insertT k v t)
```

```
Main> quickCheck prop_insertT
Falsifiable, after 8 tests:
0
0
Join Empty 0 (-1) Empty
```

What's wrong?

The Bug

```
insertT key val Empty = Join Empty key val Empty
insertT key val (Join left k v right) =
    | key <= k = Join (insertT key val left) k v right
    | key > k   = Join left k v (insertT key val right)
```

Inserts duplicate keys!

Testing Again

```
Main> quickCheck prop_insertT
Falsifiable, after 6 tests:
-2
2
Join Empty (-2) 1 Empty
```

Testing Again

```
Main> quickCheck prop_insertT
Falsifiable, after 6 tests:
-2
2
Join Empty (-2) 1 Empty

Main> insertT (-2) 2 (Join Empty (-2) 1 Empty)
Join Empty (-2) 2 Empty
```

Testing Again

```
Main> quickCheck prop_insertT
Falsifiable, after 6 tests:
-2
2
Join Empty (-2) 1 Empty

Main> insertT (-2) 2 (Join Empty (-2) 1 Empty)
Join Empty (-2) 2 Empty

Main> insert (-2,2) [(-2,1)]
[(-2,1),(-2,2)]
```

insert doesn't *remove* the old key-value pair when keys clash – the wrong model!

Fixing prop_insertT

- Ad hoc fix:

```
prop_insertT k v t =
  insert (k,v) [(k',v') | (k',v') <- contents t, k' /= k] ==
     contents (insertT k v t)
```

Data.Map

- The standard module Data.Map contains an advanced tree-based implementation of tables

Summary

- Recursive datatypes can store data in different ways
- Clever choices of datatypes and algorithms can improve performance dramatically
- Careful thought about *invariants* is needed to get such algorithms right!
- Formulating properties and invariants, and testing them, reveals bugs early