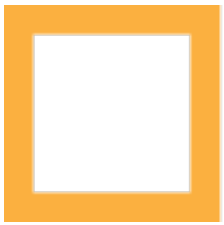
Algorithms and Datastructures
Searching Lists

Jacob Trier Frederiksen & Anders Kalhauge



cphbusiness

Spring 2019

Outline

cphbusiness
COPENHAGEN BUSINESS ACADEMY

Book Keeping

Symbol Tables continued

- Hashed
- Search Trees
- Balanced Search Trees

Book Keeping

Symbol Tables continued
- Hashed
- Search Trees
- Balanced Search Trees

Book Keeping, Week 8

cphbusiness
COPENHAGEN BUSINESS ACADEMY

- Mentimeter www.menti.com, weekly check-up.
- Infrastructure check. Are we well within our learning platform?
- Assignment #2, Q&A if needed.
- Review of exercises from last week, by peer presentation and discussion.
- Lectures & Exercises.

Book Keeping

Symbol Tables continued

- Hashed
- Search Trees
- Balanced Search Trees

cphbusiness
COPENHAGEN BUSINESS ACADEMY

Problem

We want to access a symbol table with n keys, using a key $k \in K$ as was it an index to an array with n elements.

Perfect solution

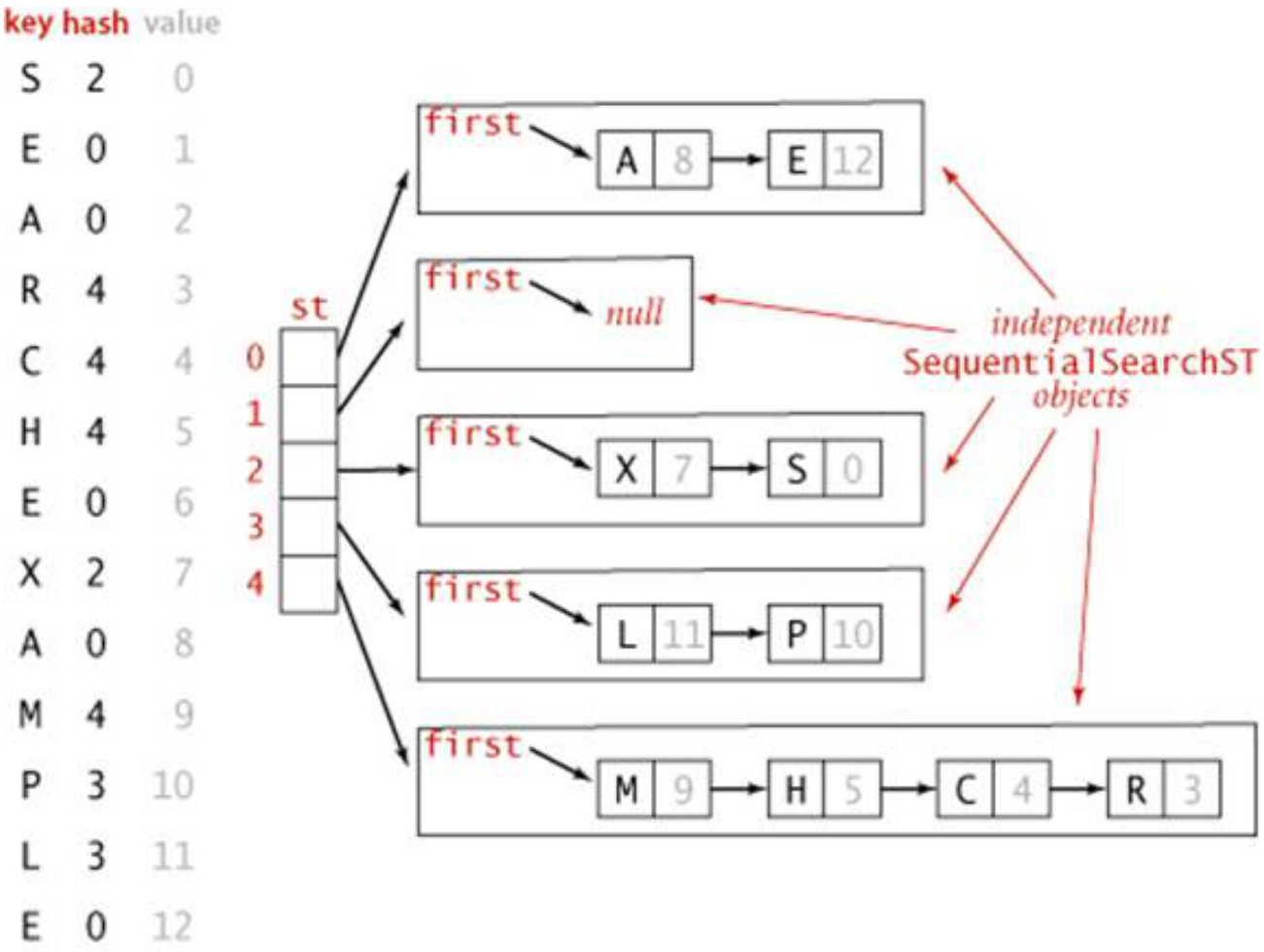
By **magic** we find a mapping from the keys K to $\{0, \ldots, n-1\}$.
We call it the hash function: $h : K \to \{0, \ldots, n-1\}$.

Realistic Solution

Find a hash function that maps the keys K uniformly over $\{0, \ldots, m-1\}$ where $m \approx n$.

Hashed Symbol Tables

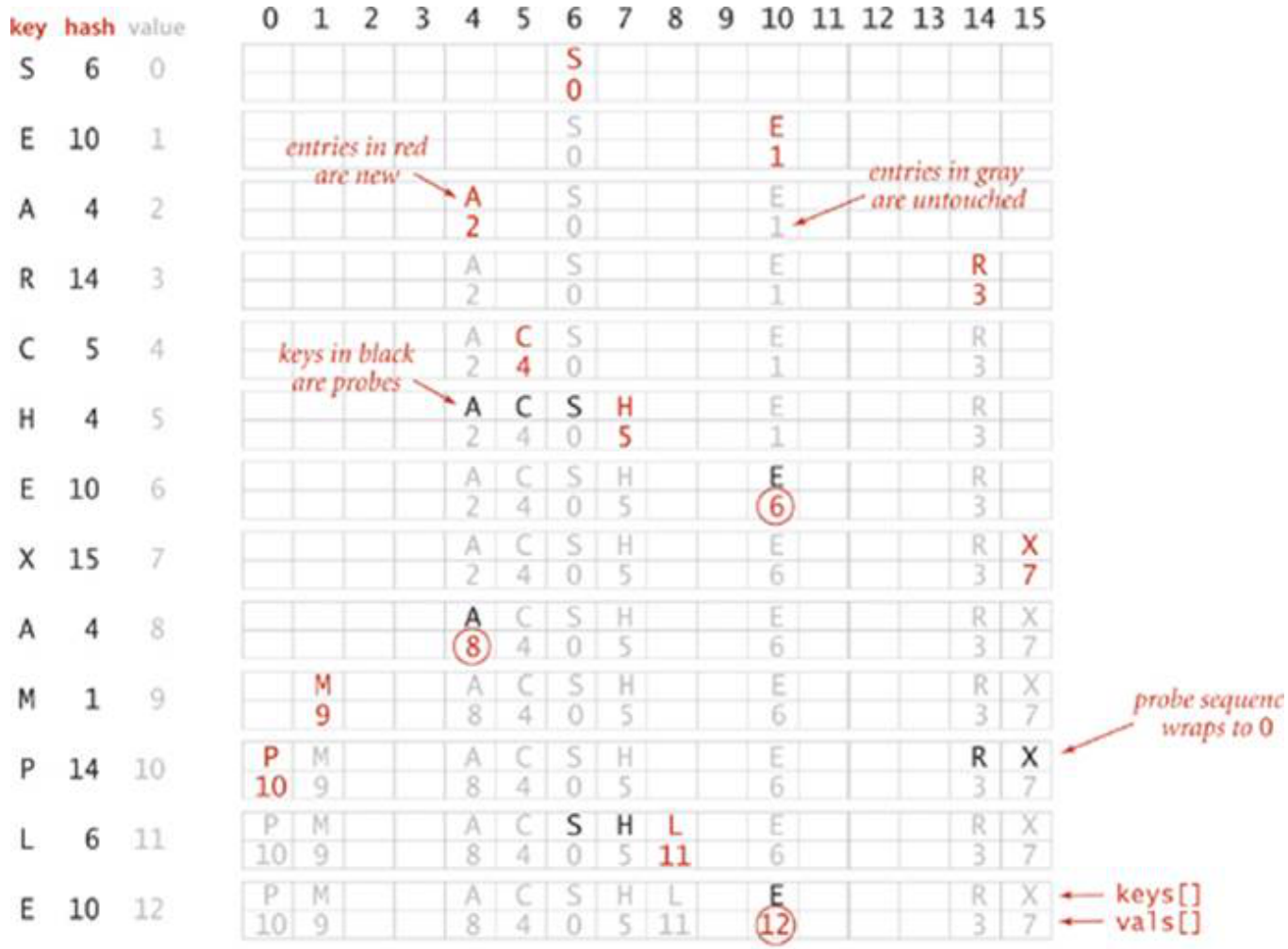
Chained hashing



Hashing with separate chaining for standard indexing client

cphbusiness
COPENHAGEN BUSINESS ACADEMY

Linear probing



Trace of linear-probing ST implementation for standard indexing client

With:

```
@FunctionalInterface
public interface HashFunction {
  int function(String key);
  }
```

You can:

```
public static test(String[] keys, HashFunction hash) {
  for (String key : keys)
      System.out.println(hash.function(key));
  }
public static void main(String... args) {
  String[] words =
      FileUtility.toStringArray("sp.txt", "[^A-Za-z]");
  test(words, k -> k.length());
  }
```

Exercise ?? - Hash function

cphbusiness
COPENHAGEN BUSINESS ACADEMY

Experiment with various hash implementation of hash functions. All functions should return a number between 0 and 31.

The signature of the hash function should be:

```
@FunctionalInterface
public interface HashFunction {
  int function(String key);
  }
```

Create hash functions that uses:

- the first character code
- the last character code
- the sum of character codes
- `Strings hashCode()` method

You can use `n%32` to fit `n` into the range of $0...31$

Make a histogram of each hash function

Cuckoo Hashing

cphbusiness
COPENHAGEN BUSINESS ACADEMY

- Insert $O(1)$ expected
- Delete, Query $O(1)$ Guaranteed
- Two hash functions h_1 and h_2

1. Compute $h_1(x)$
2. If $T[h_1(x)]$ is empty: done, otherwise if y occupies the spot, evict y and put x in $T[h_1(x)]$, Try to place y using $h_2(y)$
3. If $T[h_2(y)]$ is empty: done, otherwise evict $T[h_2(y)]$, y here and repeat this step with the evicted value
4. If more than $log(n)$ attempts - rehash

Other hashing aspects

- Universal hashing
- Static tables
- Perfect hashing

Let's Take A Coffee Break.

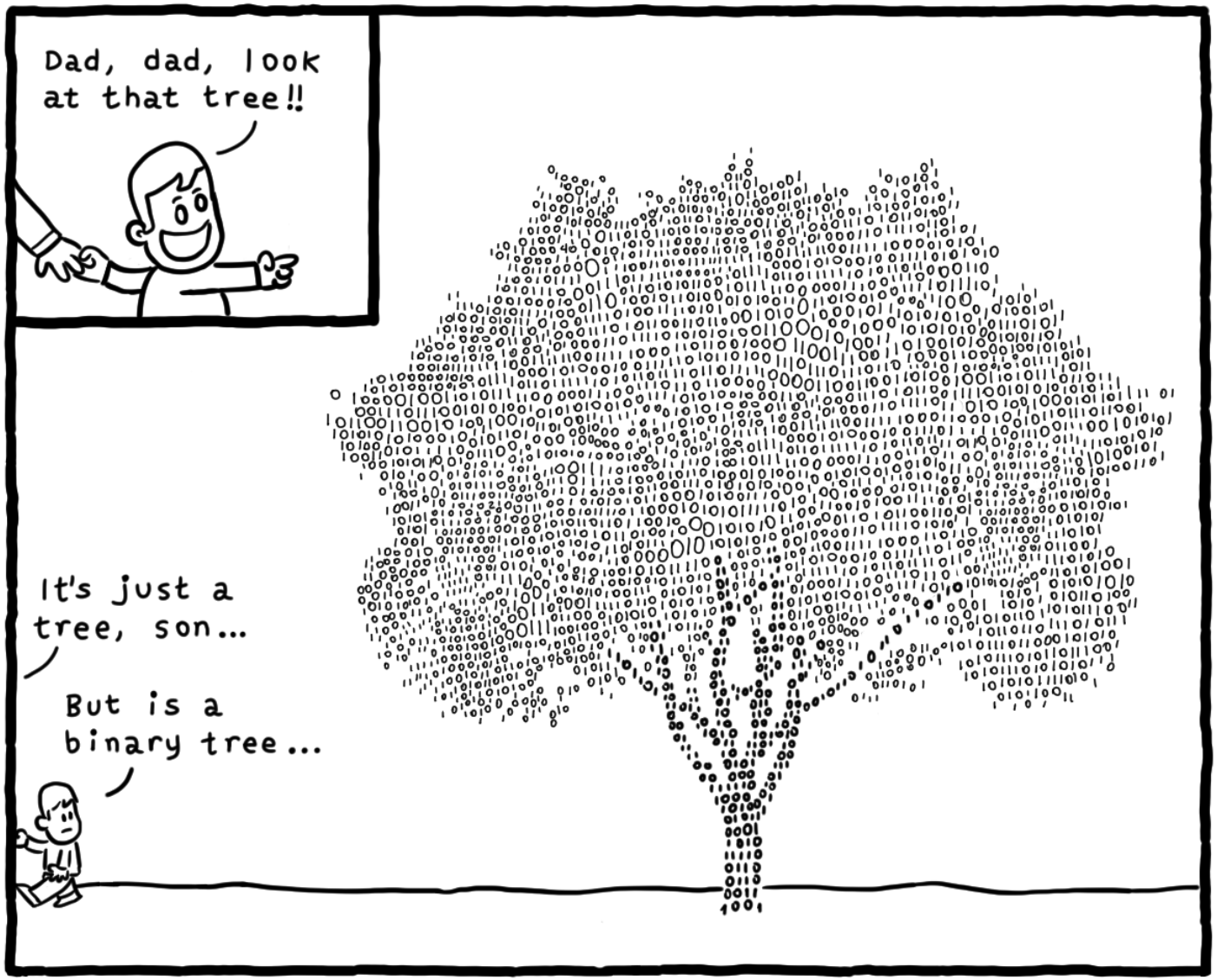
cphbusiness
COPENHAGEN BUSINESS ACADEMY

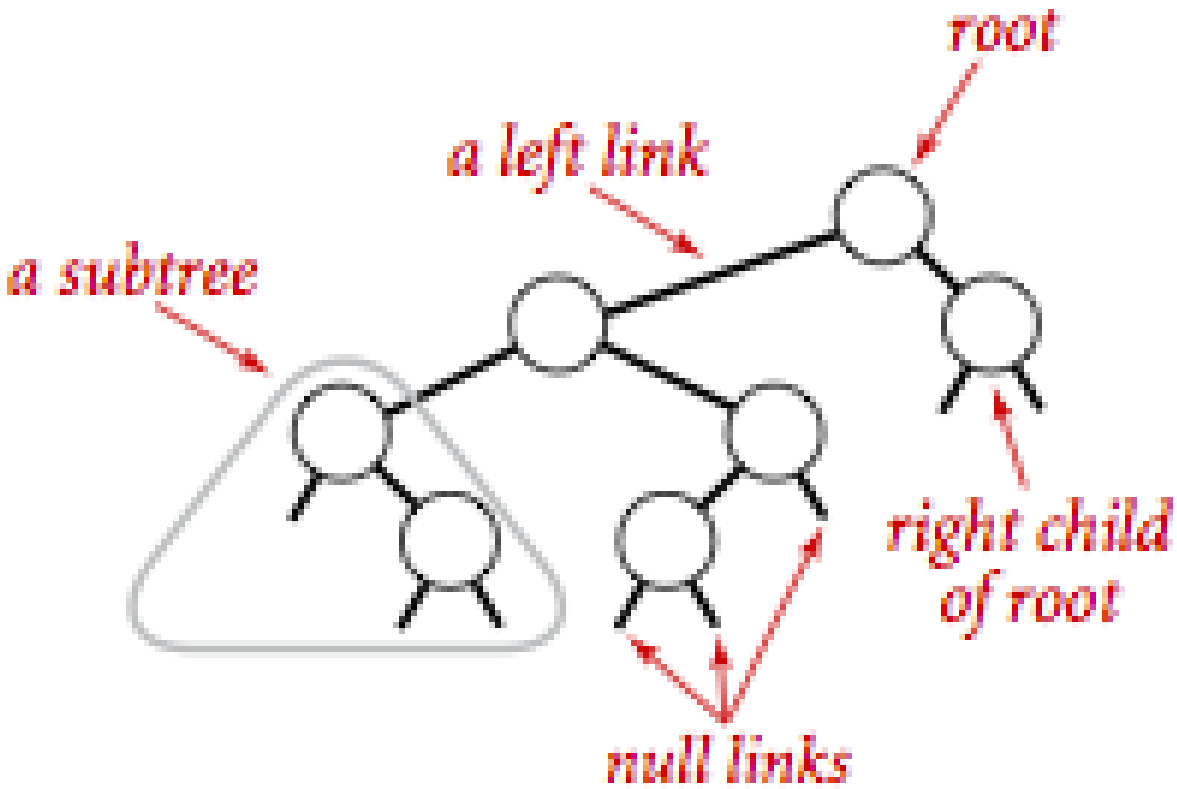
Last week: the `insert` operation was expensive (worst case) for both unordered linked lists, and ordered array symbol tables.

algorithm (data structure)	worst-case cost (after N inserts)		average-case cost (after N random inserts)		efficiently support ordered operations?
	search	insert	search hit	insert	
sequential search (unordered linked list)	N	N	$N/2$	N	no
binary search (ordered array)	$\lg N$	$2N$	$\lg N$	N	yes

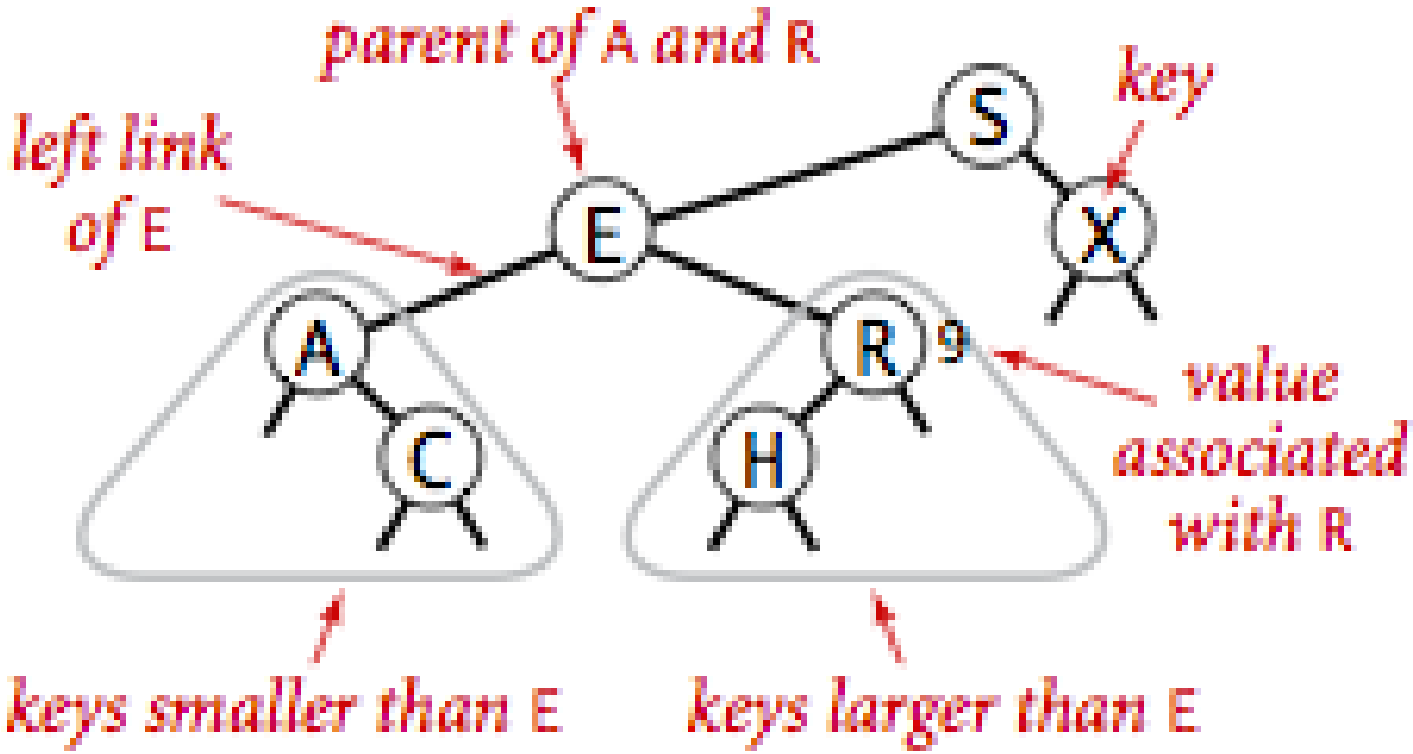
Cost summary for basic symbol-table implementations

The question arose: can we get a faster `insert` operation? (Answer is fortunately yes :-))


Dad, dad, look at that tree!!
It's just a tree, son...
But is a binary tree...
Daniel Stori {turnoff.us}



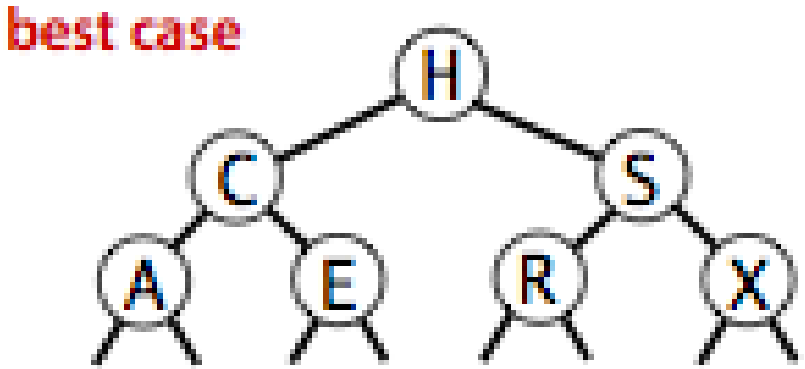
Anatomy of a binary tree



Anatomy of a binary search tree

cphbusiness
COPENHAGEN BUSINESS ACADEMY

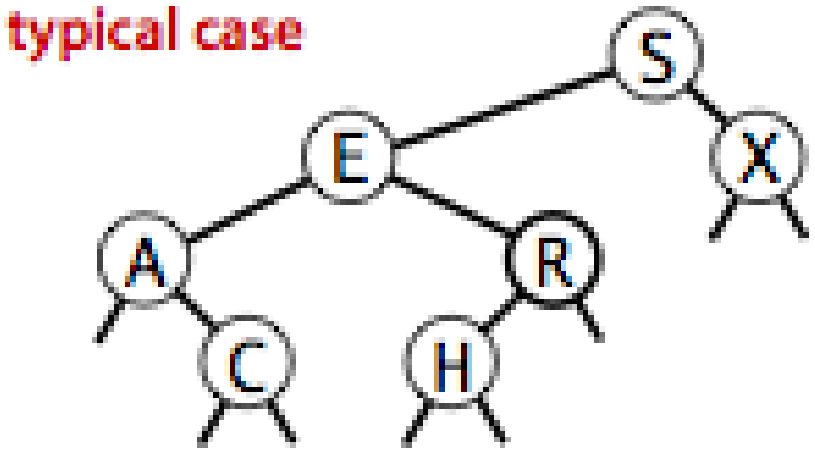
Analysis



$$O(\log n)$$

cphbusiness
COPENHAGEN BUSINESS ACADEMY

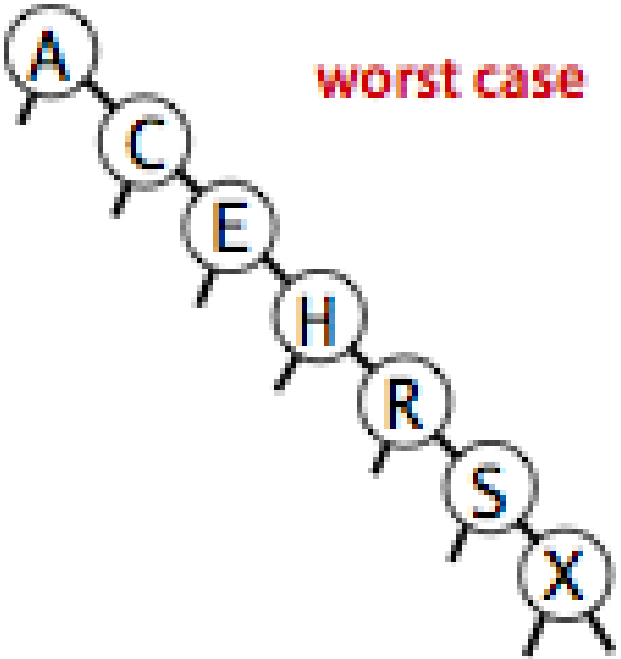
Analysis



$$O(\log n)$$

cphbusiness
COPENHAGEN BUSINESS ACADEMY

Analysis



$$O(n)$$

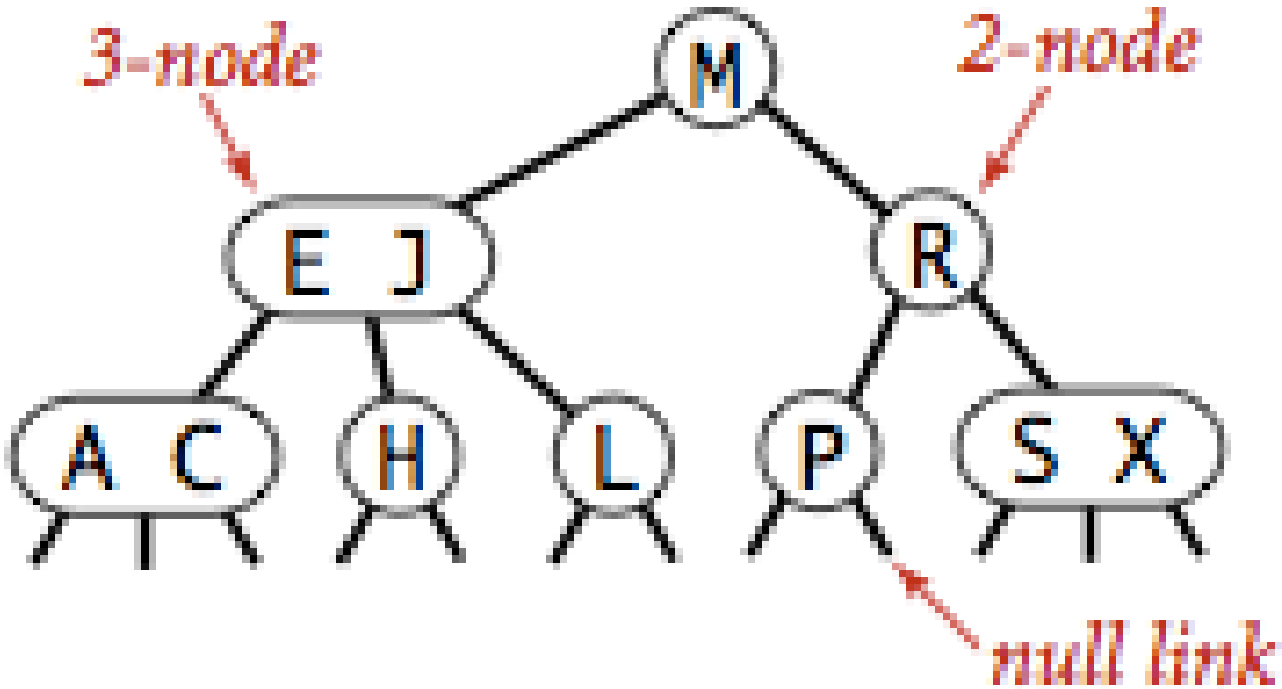
This week: the `insert` operation is still expensive (but only worst case) for binary search trees. Missing $logN$ for worst case....

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	½ N	N	`equals()`
binary search (ordered array)	lg N	N	lg N	½ N	`compareTo()`
BST	N	N	1.39 lg N	1.39 lg N	`compareTo()`

New question: can we get a *true* $\sim logN$ `insert` operation? (Answer is fortunately still yes :-))

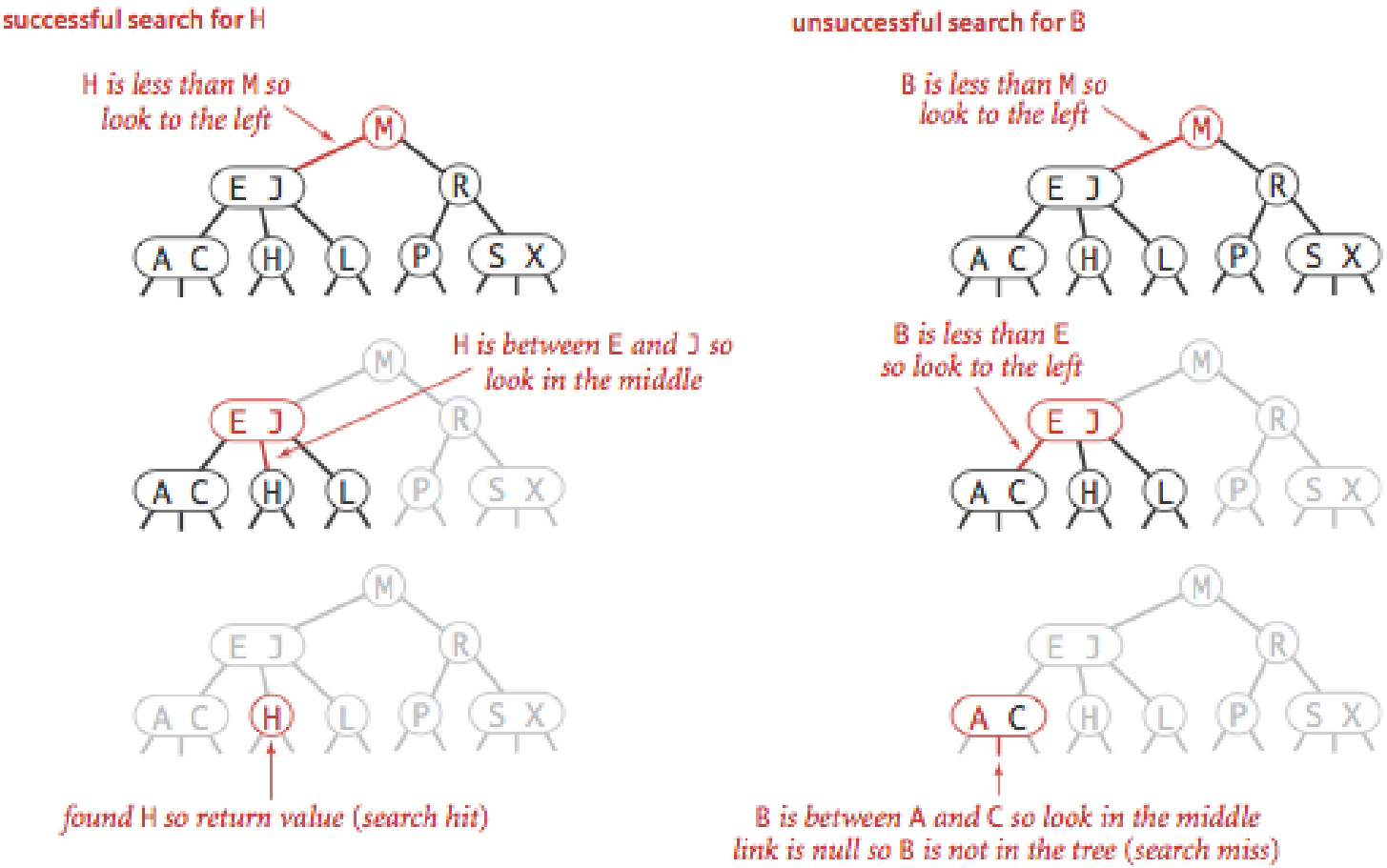
cphbusiness
COPENHAGEN BUSINESS ACADEMY

Anatomy



Anatomy of a 2-3 search tree

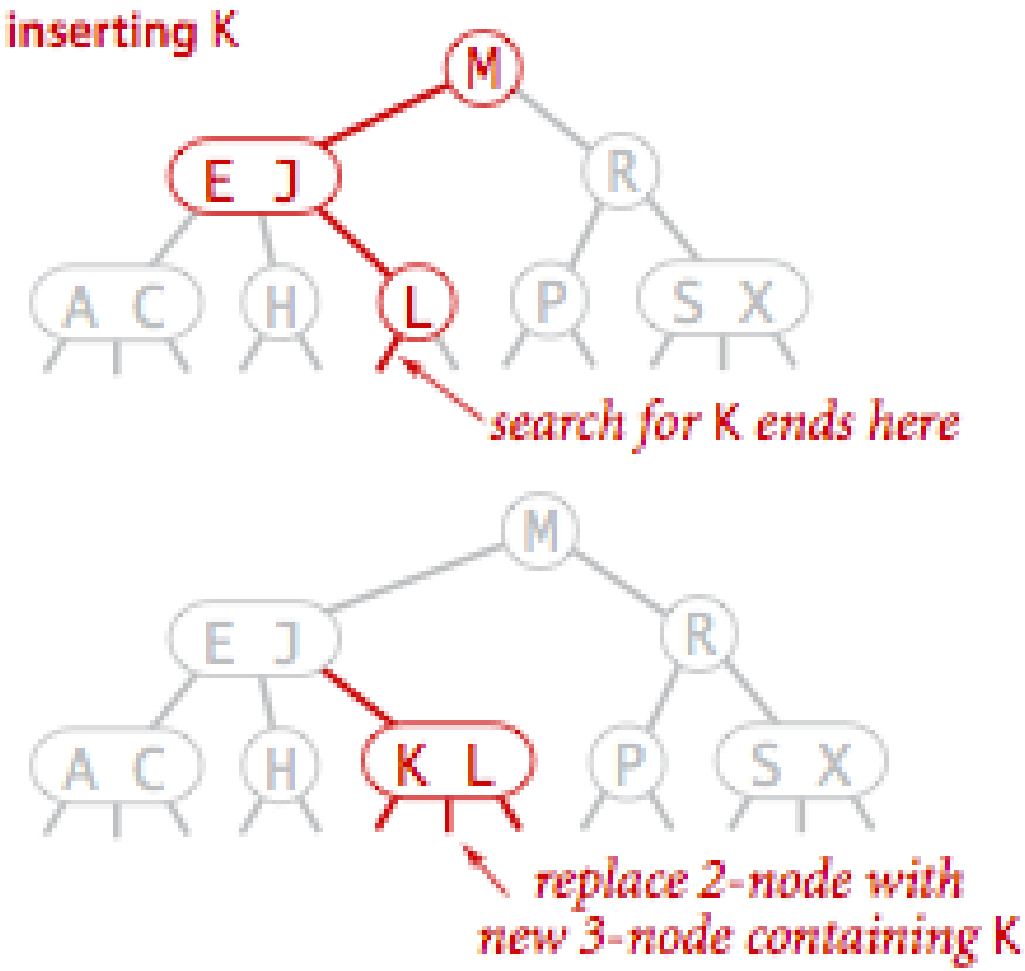
Searching



Search hit (left) and search miss (right) in a 2-3 tree

cphbusiness
COPENHAGEN BUSINESS ACADEMY

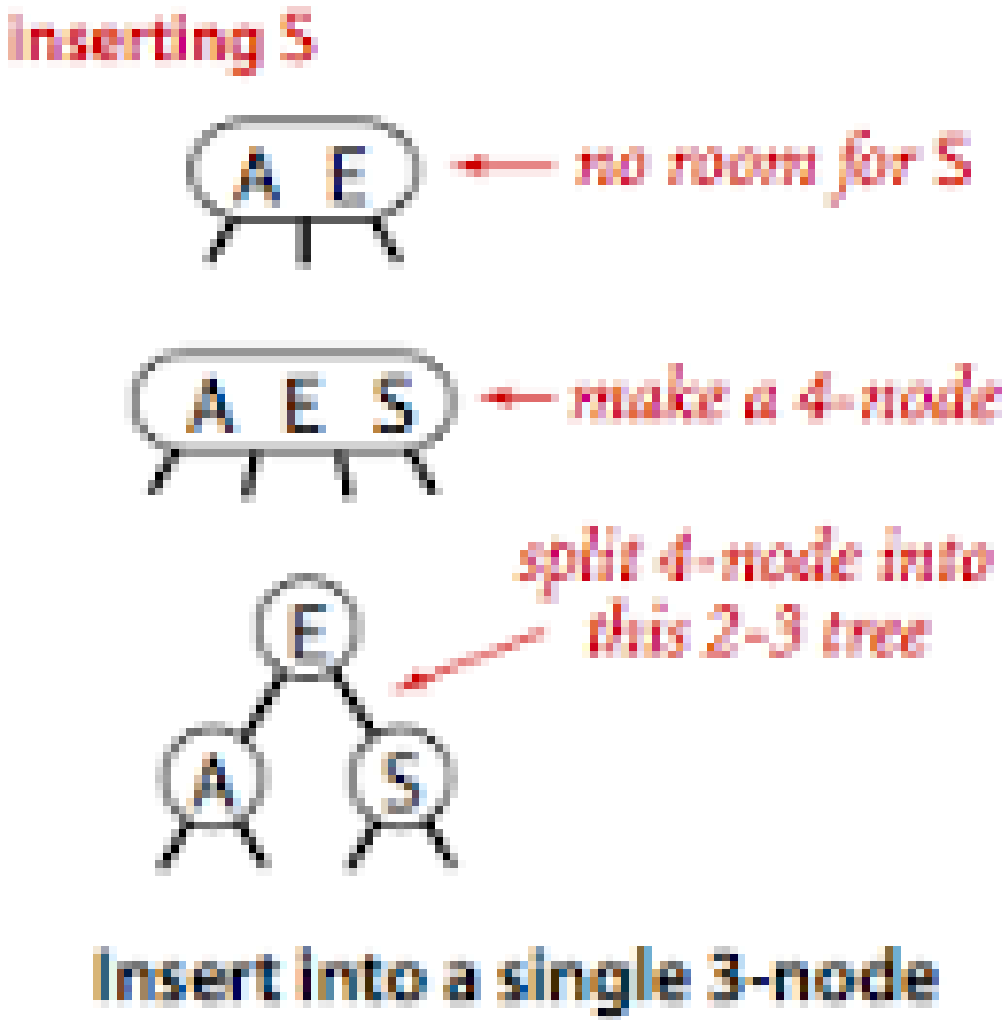
Inserting



Insert into a 2-node

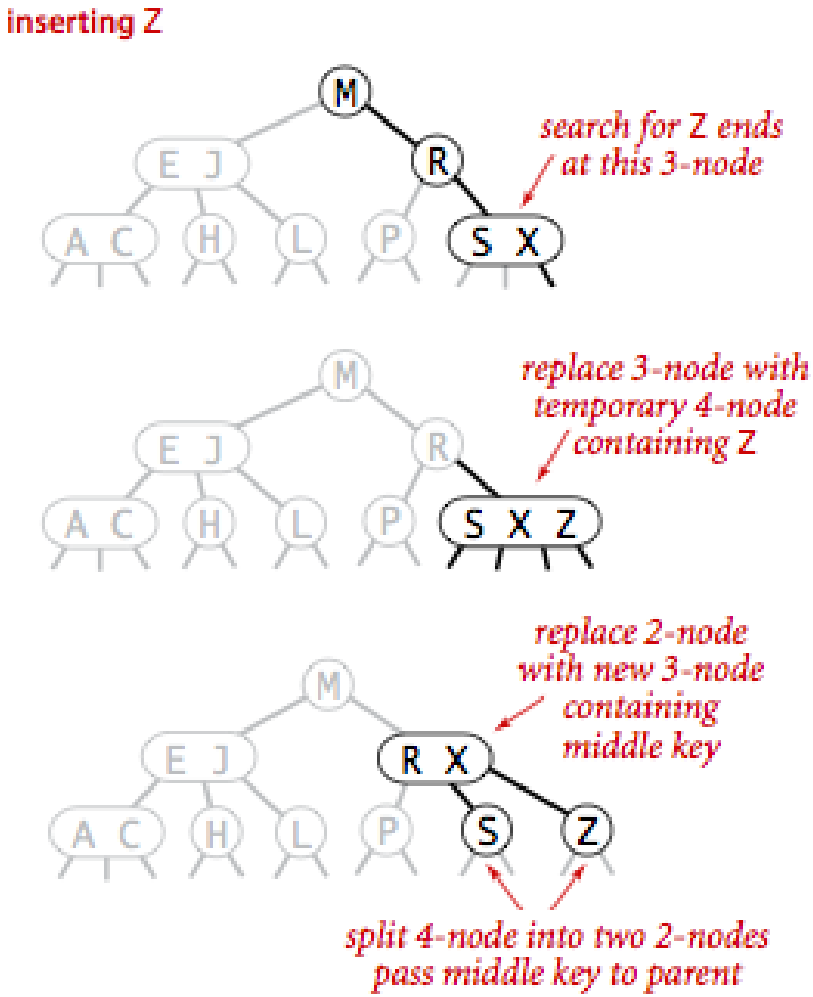
cphbusiness
COPENHAGEN BUSINESS ACADEMY

Inserting



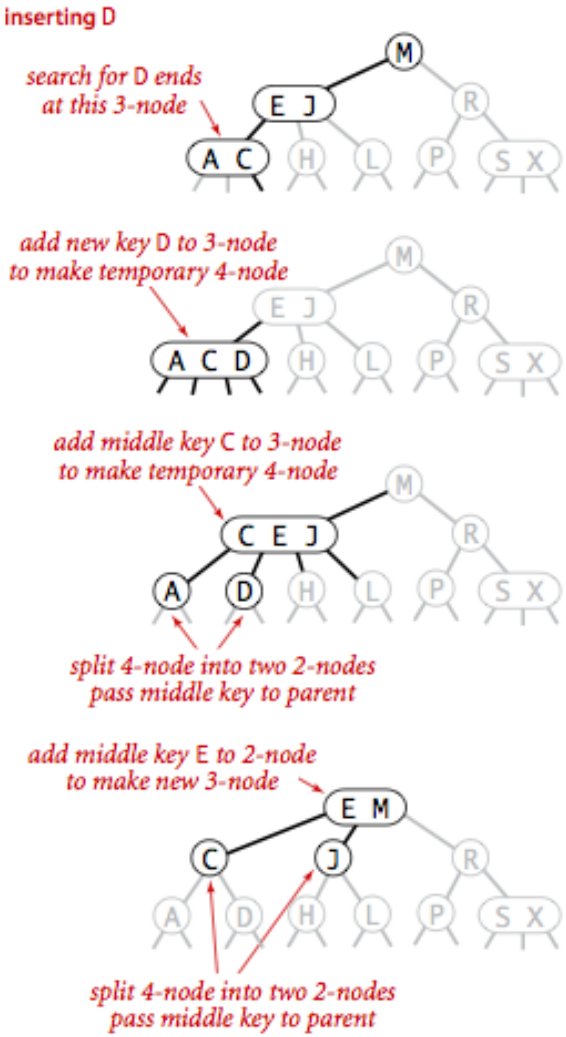
cphbusiness
COPENHAGEN BUSINESS ACADEMY

Inserting

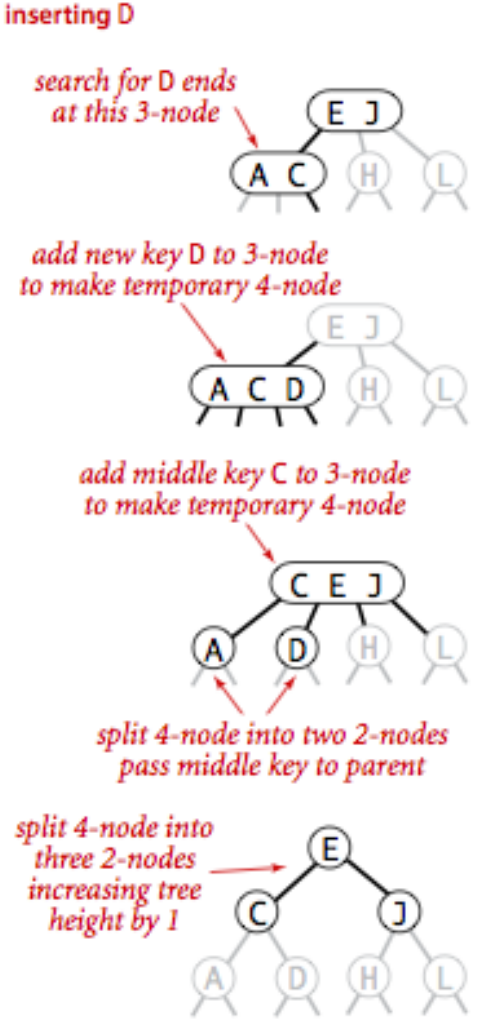


Insert into a 3-node whose parent is a 2-node

Inserting



Insert into a 3-node whose parent is a 3-node

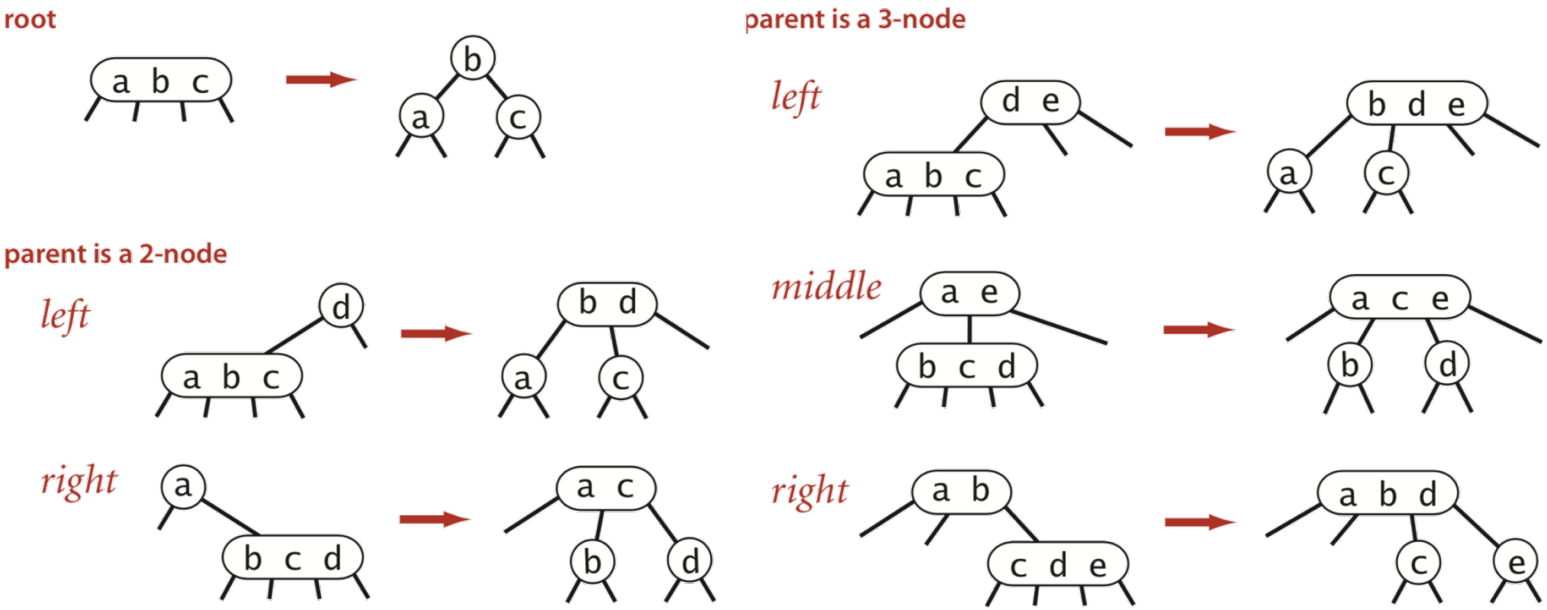


Splitting the root

cphbusiness
COPENHAGEN BUSINESS ACADEMY

Summing up

The restructuring of the 2-3-Tree leaves perfect symmetry (almost).....

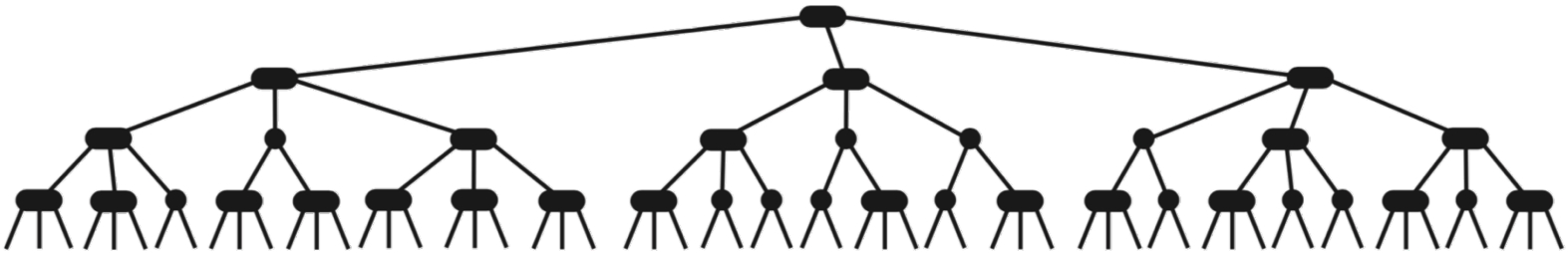


cphbusiness
COPENHAGEN BUSINESS ACADEMY

Summing up

... but guarantees depth of at most $log_2 N$!

Perfect balance. Every path from root to null link has same length.



Tree height.

- Worst case: $\lg N$. [all 2-nodes]
- Best case: $\log_3 N \approx .631 \lg N$. [all 3-nodes]
- Between 12 and 20 for a million nodes.
- Between 18 and 30 for a billion nodes.

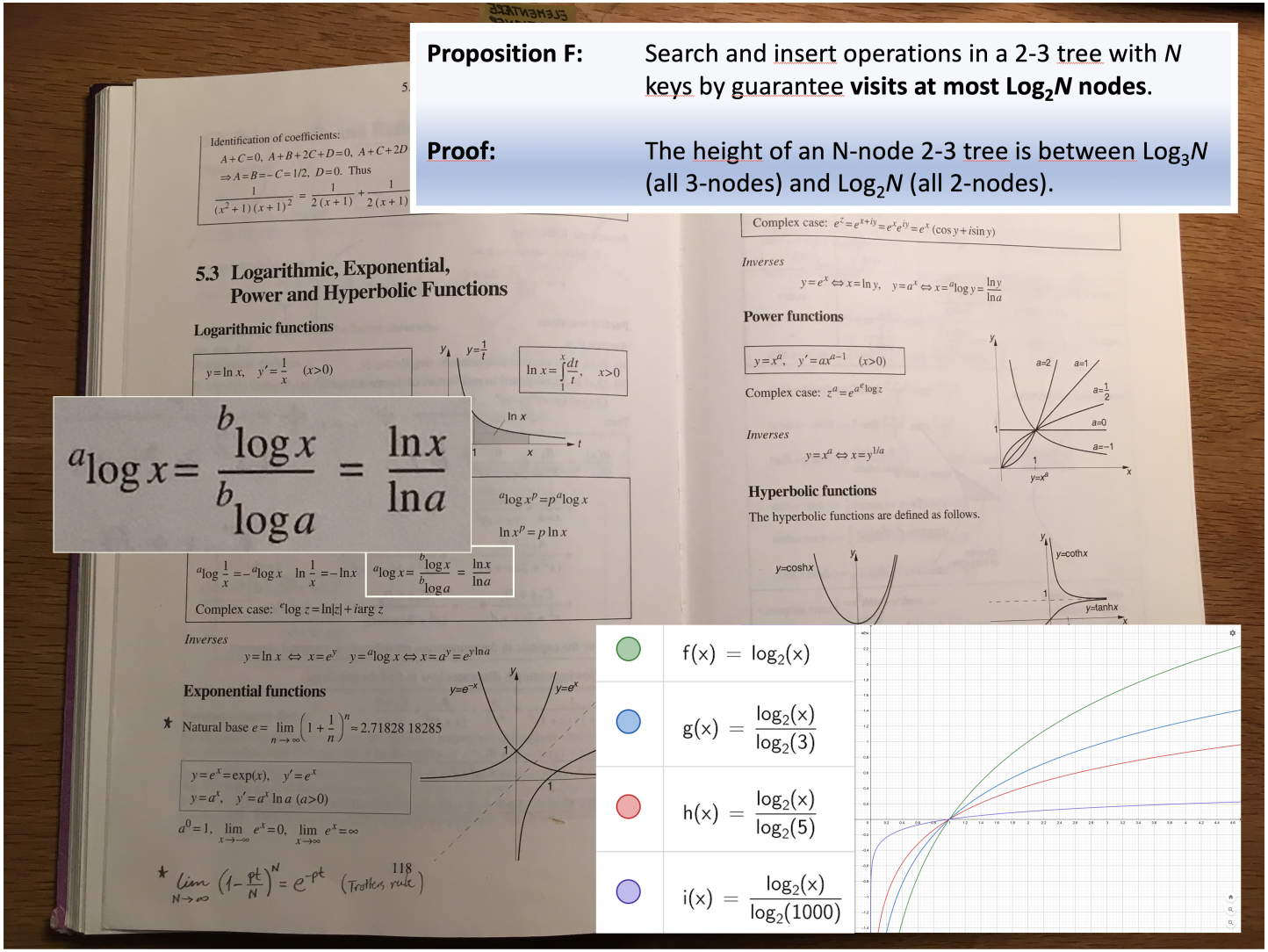
This week: we've found one structure that guarantees at least `insert` operation is still expensive (but only worst case) for binary search trees. Missing $logN$ for worst case....

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	½ N	N	½ N		`equals()`
binary search (ordered array)	$\lg N$	N	N	$\lg N$	½ N	½ N	✔	`compareTo()`
BST	N	N	N	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✔	`compareTo()`
2-3 tree	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✔	`compareTo()`

Question answered: we *can* get a true $\sim LogN$ `insert` operation?

A Curiosum ...

Just to be sure that all log-scaling does almost equally well in algorithmic complexity analysis.



Exercise ?? - BST Structure

cphbusiness
COPENHAGEN BUSINESS ACADEMY

Implement the structure for a binary search tree

OPTIONAL (but you might need it for the hand-in, anyway)

Implement the 2-3 Balanced Search Tree