

# Algorithms and Datastructures

## Searching Lists

Jacob Trier Frederiksen & Anders Kalhauge



Spring 2019

## Book Keeping

### Symbol Tables continued

Hashed

Search Trees

Balanced Search Trees

## Book Keeping

### Symbol Tables continued

Hashed

Search Trees

Balanced Search Trees

- Mentimeter [www.menti.com](<https://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

## 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.

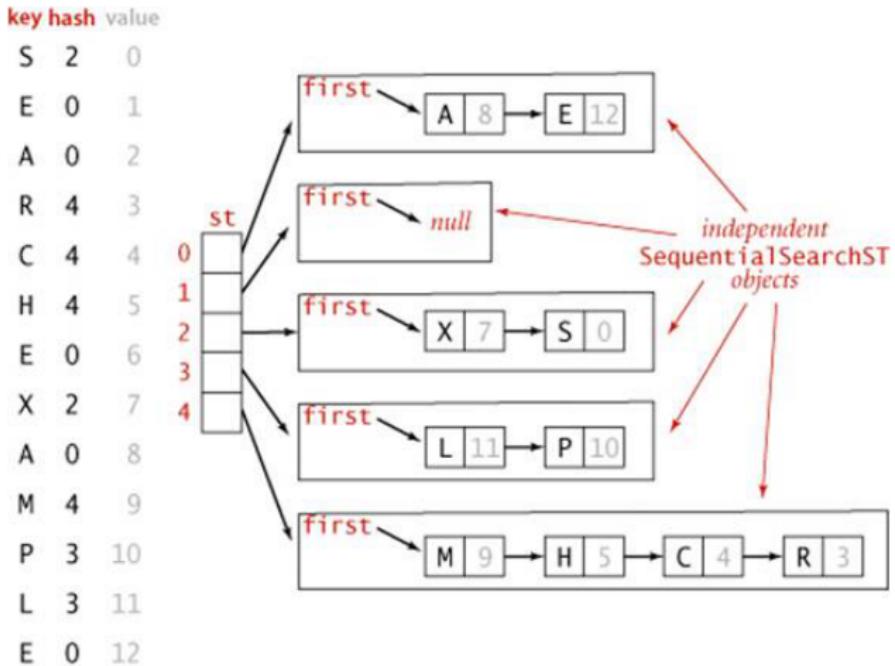
## Perfect solution

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

## Realistic Solution

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

## Chained hashing



Hashing with separate chaining for standard indexing client

# Chained hashing

## Linear probing

key	hash	value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	6	0			S													
E	10	1				S			0			E						
A	4	2					entries in red are new			S			E					
R	14	3						A									R	3
C	5	4							C	S			E				R	1
H	4	5								2	4	0						3
E	10	6								A	C	S	H		E		R	
X	15	7									2	4	0	S		6		3
A	4	8									A	C	S	H		E	R	X
M	1	9										8	4	0	S	6		3
P	14	10										M	A	C	S	H	E	R
L	6	11										9	8	4	0	S	6	X
E	10	12										P	M	A	C	S	H	L

Annotations:

- entries in red are new
- entries in gray are untouched
- keys in black are probes
- probe sequence wraps to 0
- keys[] ← keys[12]
- vals[] ← vals[12]

With:

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

You can:

```
public static void 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());
}
```

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
- `String's hashCode()` method

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

**Make a histogram of each hash function**

- Universal hashing
- Static tables
- Perfect hashing

Let's Take A Coffee Break.



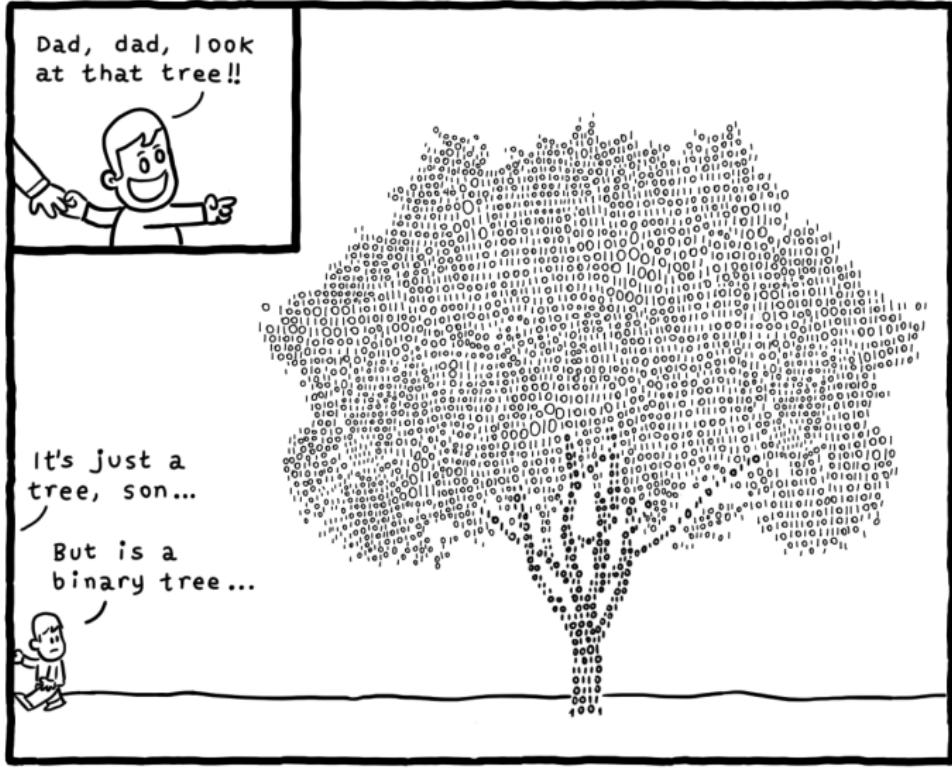
Last week: the `insert` operation was expensive (worst case) for both unordered linked lists, and ordered array symbol tables.

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	
<i>sequential search (unordered linked list)</i>	$N$	$N$	$N/2$	$N$	no
<i>binary search (ordered array)</i>	$\lg N$	$2N$	$\lg N$	$N$	yes

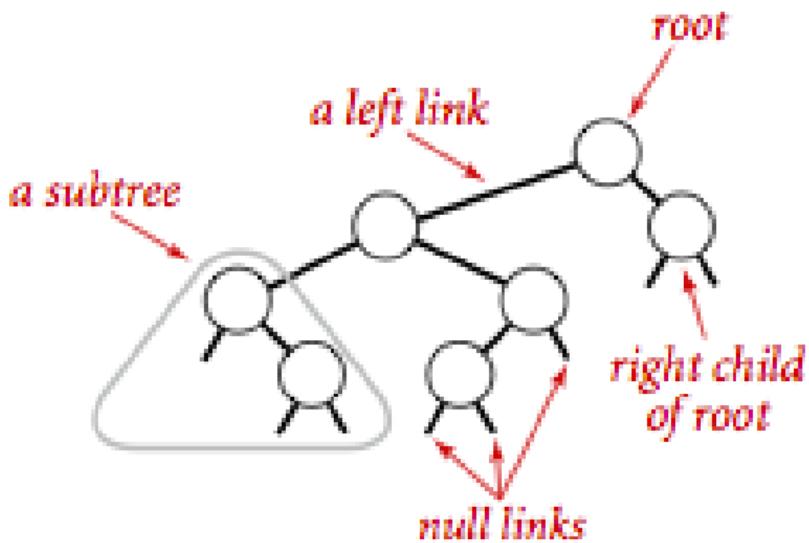
Cost summary for basic symbol-table implementations

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

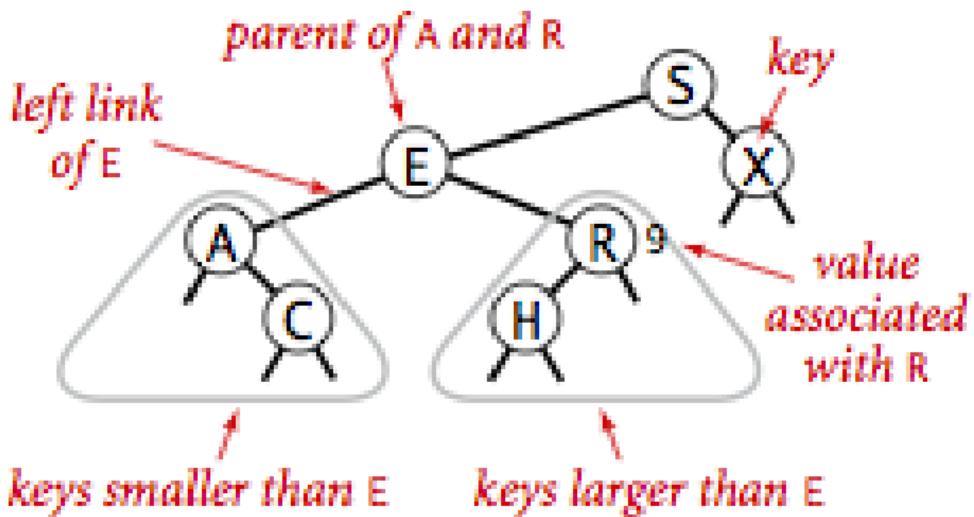
# Binary Trees



Daniel Storl {turnoff.us}



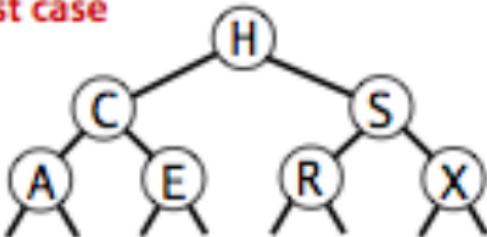
## Anatomy of a binary tree



## Anatomy of a binary search tree

## Analysis

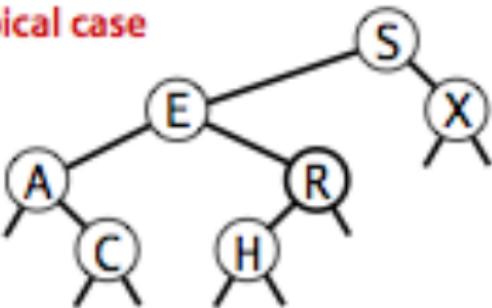
best case



$$O(\log n)$$

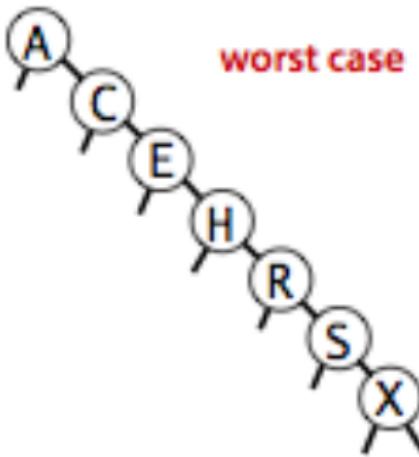
## Analysis

typical case



$$O(\log n)$$

## Analysis



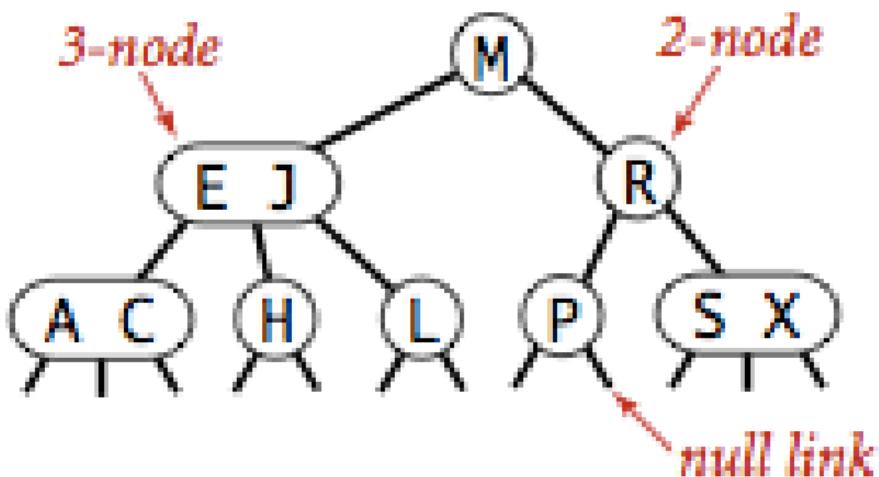
$$O(n)$$

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

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	$N$	$N$	$\frac{1}{2}N$	$N$	<code>equals()</code>
binary search (ordered array)	$\lg N$	$N$	$\lg N$	$\frac{1}{2}N$	<code>compareTo()</code>
BST	$N$ 	$N$ 	$1.39 \lg N$	$1.39 \lg N$	<code>compareTo()</code>

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

### Anatomy

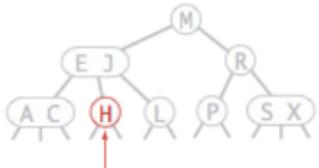
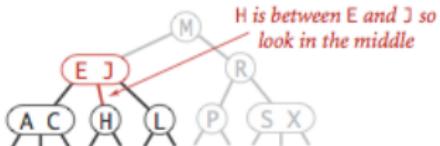
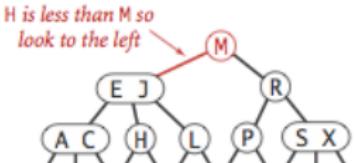


**Anatomy of a 2-3 search tree**

# 2-3 Search trees

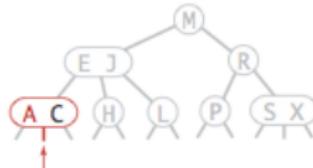
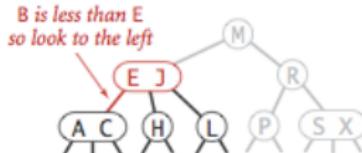
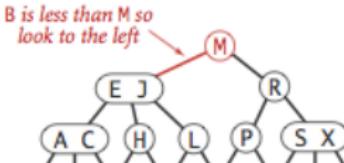
## Searching

successful search for H



found H so return value (search hit)

unsuccessful search for B

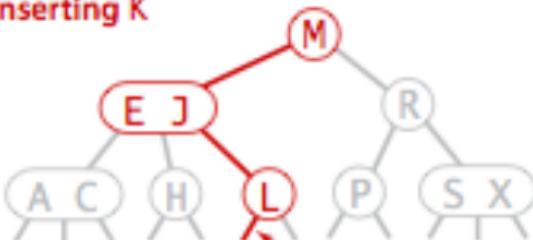


B is between A and C so look in the middle  
link is null so B is not in the tree (search miss)

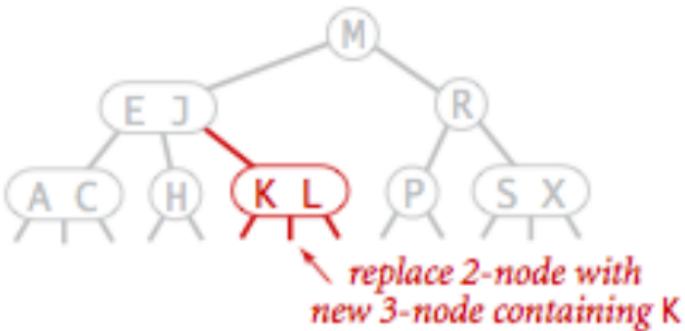
Search hit (left) and search miss (right) in a 2-3 tree

## Inserting

inserting K



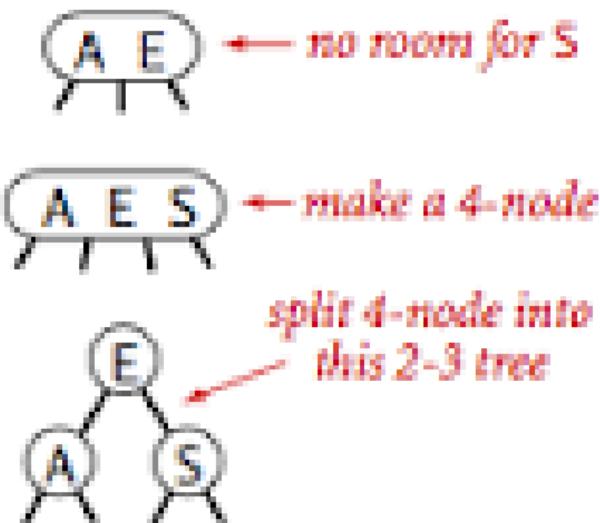
search for K ends here

replace 2-node with  
new 3-node containing K

Insert into a 2-node

### Inserting

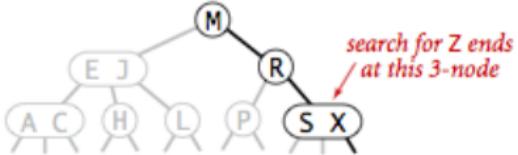
#### Inserting 5



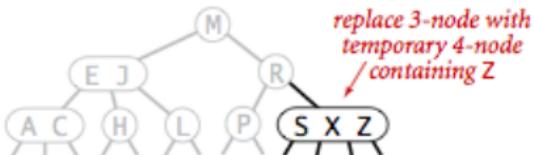
Insert into a single 3-node

## Inserting

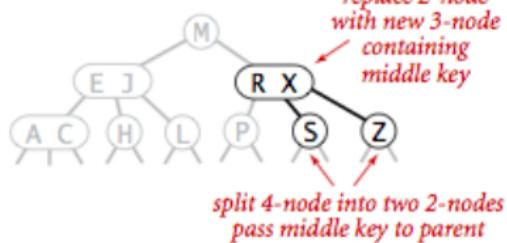
inserting Z



search for Z ends  
at this 3-node



replace 3-node with  
temporary 4-node  
containing Z



replace 2-node  
with new 3-node  
containing  
middle key

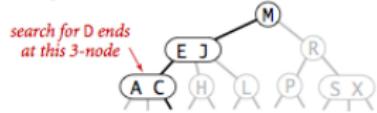
split 4-node into two 2-nodes  
pass middle key to parent

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

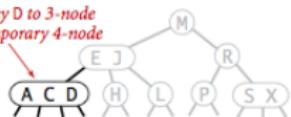
# 2-3 Search trees

## Inserting

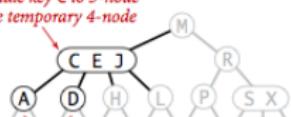
inserting D



add new key D to 3-node to make temporary 4-node

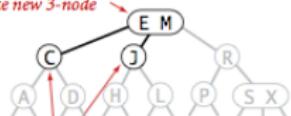


add middle key C to 3-node to make temporary 4-node



split 4-node into two 2-nodes  
pass middle key to parent

add middle key E to 2-node to make new 3-node

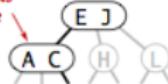


split 4-node into two 2-nodes  
pass middle key to parent

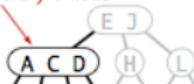
Insert into a 3-node whose parent is a 3-node

inserting D

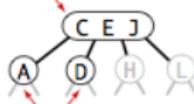
search for D ends at this 3-node



add new key D to 3-node to make temporary 4-node

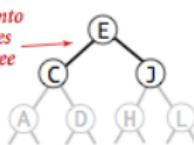


add middle key C to 3-node to make temporary 4-node



split 4-node into two 2-nodes  
pass middle key to parent

split 4-node into three 2-nodes  
increasing tree height by 1

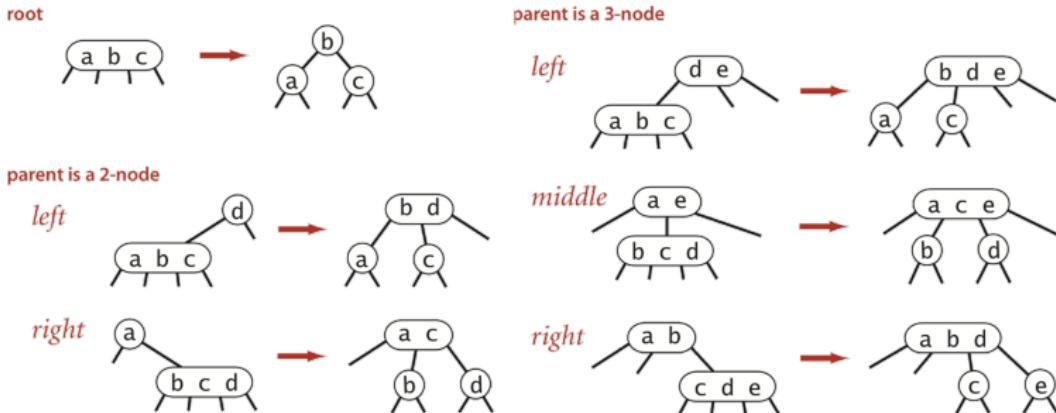


Splitting the root

# 2-3 Search trees

## Summing up

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

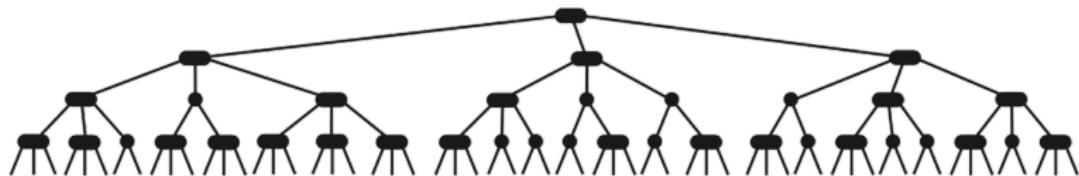


## 2-3 Search trees

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  $\log N$  for worst case....

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	$N$	$N$	$N$	$\frac{1}{2}N$	$N$	$\frac{1}{2}N$		equals()
binary search (ordered array)	$\lg N$	$N$	$N$	$\lg N$	$\frac{1}{2}N$	$\frac{1}{2}N$	✓	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  $\sim \log N$  insert operation?

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

**Proposition F:** Search and insert operations in a 2-3 tree with  $N$  keys by guarantee visits at most  $\log_2 N$  nodes.

**Proof:**

Identification of coefficients:  
 $A + C = 0, A + B + 2C + D = 0, A + C + 2D$   
 $\Rightarrow A + B = -C = \Omega(2), D = 0.$  Thus  
 $\frac{1}{(x^2+1)(x+1)^2} = \frac{1}{2(x+1)} + \frac{1}{2(x^2+1)}$

The height of an  $N$ -node 2-3 tree is between  $\log_3 N$  (all 3-nodes) and  $\log_2 N$  (all 2-nodes).

Complex case: $x^2 = e^{i\pi/2} \Leftrightarrow x = e^{i\pi/4} = e^{i\pi/4} (\cos y + i \sin y)$	
Inverses	$y = e^x \Leftrightarrow x = \ln y, \quad y = a^x \Leftrightarrow x = \frac{\ln y}{\ln a}$
Power functions	
$y = x^p, \quad y' = px^{p-1} \quad (p > 0)$	
Complex case: $z^p = e^{ip\theta} \Leftrightarrow z = e^{i\theta/p}$	
Inverses	$y = x^p \Leftrightarrow x = y^{1/p}$
Hyperbolic functions	
The hyperbolic functions are defined as follows.	
$y = \cosh x$	$y = \coth x$
$y = \sinh x$	$y = \tanh x$

**5.3 Logarithmic, Exponential, Power and Hyperbolic Functions**

Logarithmic functions

$y = \ln x, \quad y' = \frac{1}{x} \quad (x > 0)$

$\ln x = \frac{x}{\ln a}, \quad x > 0$

$a \log x = \frac{b \log x}{b \log a} = \frac{\ln x}{\ln a}$

$\log x^p = p \log x$

$\ln x^p = p \ln x$

$\log \frac{1}{x} = -\log x \quad \ln \frac{1}{x} = -\ln x$

$\log x = \frac{\ln x}{\ln a} = \frac{\ln x}{\ln \ln a}$

Complex case:  $\log z = \ln|z| + i \arg z$

Inverses

$y = \ln x \Leftrightarrow x = e^y, \quad y = \ln \log x \Leftrightarrow x = e^y = e^{\ln \log x}$

Exponential functions

\* Natural base  $e = \lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n = 2.71828 1825$

$y = e^x = \exp(x), \quad y' = e^x$

$y = a^x, \quad y' = a^x \ln a \quad (a > 0)$

$d^0 = 1, \quad \lim_{x \rightarrow -\infty} e^x = 0, \quad \lim_{x \rightarrow \infty} e^x = \infty$

\*  $(\lim_{N \rightarrow \infty} \left(-\frac{e^k}{N}\right)^N)^M = e^{-kM} \quad (\text{Teilfaktor } 118)$

**Proposition F:** Search and insert operations in a 2-3 tree with  $N$  keys by guarantee visits at most  $\log_2 N$  nodes.

**Proof:**

Identification of coefficients:  
 $A + C = 0, A + B + 2C + D = 0, A + C + 2D$   
 $\Rightarrow A + B = -C = \Omega(2), D = 0.$  Thus  
 $\frac{1}{(x^2+1)(x+1)^2} = \frac{1}{2(x+1)} + \frac{1}{2(x^2+1)}$

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