

# 9

## Suchbäume

*Algorithmen & Datenstrukturen · Sommersemester 2026*

Prof. Dr. Sebastian Wild

## 9 Sortierte Wörterbücher

- 9.1 Symbol Tables (Wörterbücher)
- 9.2 Konventionen in Java
- 9.3 Elementare Ideen für Symbol Tables
- 9.4 Priority Queues und Heaps
- 9.5 Operationen in binären Heaps
- 9.6 Binäre Suchbäume
- 9.7 Augmented BSTs
- 9.8 Balancierte Suchbäume: 2–3-Bäume
- 9.9 Balancierte Suchbäume: Red-Black-Trees

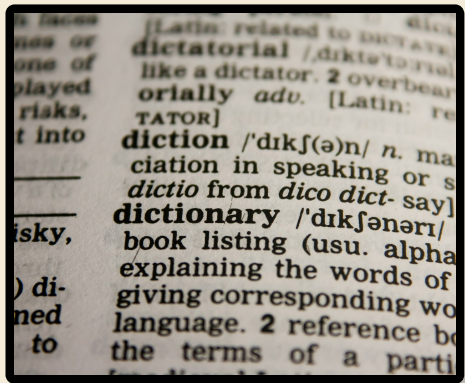
## 9.1 Symbol Tables (Wörterbücher)

# Symbol Table ADT

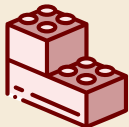
Symbol table (ST) / Dictionary / Map / Assoziatives Array / key-value store:

Java: `java.util.Map<K,V>`

Python `dict {k:v}`



- ▶ `put(k, v)`     Python dict: `d[k] = v`  
Füge key-value mapping  $(k, v)$  zur ST hinzu
- ▶ `get(k)`     Python dict: `d[k]`  
Gib Wert für key  $k$  zurück (falls vorhanden)
- ▶ `delete(k)`     Python dict: `del d[k]`  
Entferne key  $k$  (und assoziierten Wert) aus ST
- ▶ `contains(k)`     Python dict: `k in d`  
Ist key  $k$  in der ST?
- ▶ `isEmpty(), size()`
- ▶ `create()`



*Der wahrscheinlich wichtigste Baustein der Informatik!*

(Jede Programmiersprache liefert eine ST mit.)

# Symbol tables und mathematische Funktionen

- ▶ Symbol tables sind oberflächlich ähnlich zu (partiellen) mathematischen Funktionen
  - ▶ beide assoziieren (Funktions-) Werte zu Schlüssel(n) (Argumenten)
- ▶ aber: mathematische Funktionen sind *statisch/immutable* (unveränderlich; ändert Wert nicht)  
(Eine andere Zuordnung von Werten ergibt eine gänzlich *andere* Funktion)
- ▶ Symbol Table = *dynamisches* Mapping  
Also eine Funktion, die sich über die Zeit verändern kann

## Basic symbol table API

---

**Associative array abstraction.** Associate one value with each key.

```
public class ST<Key, Value>
```

```
    ST()
```

*create an empty symbol table*

```
    void put(Key key, Value val)
```

*put key-value pair into the table* ← a[key] = val;

```
    Value get(Key key)
```

*value paired with key* ← a[key]

```
    boolean contains(Key key)
```

*is there a value paired with key?*

```
    void delete(Key key)
```

*remove key (and its value) from table*

```
    boolean isEmpty()
```

*is the table empty?*

```
    int size()
```

*number of key-value pairs in the table*

```
    Iterable<Key> keys()
```

*all the keys in the table*

# Ordered symbol tables

Optional können Symbol Tables weitere Operationen unterstützen:

- ▶  $\text{min}()$ ,  $\text{max}()$   
Kleinster bzw. größter Schlüssel in ST
- ▶  $\text{floor}(x)$ ,  $\lfloor x \rfloor = \mathbb{Z}.\text{floor}(x)$   
Größter Schlüssel  $k$  in ST mit  $k \leq x$ .
- ▶  $\text{ceiling}(x)$   
Kleinster Schlüssel  $k$  in ST mit  $k \geq x$ .
- ▶  $\text{rank}(x)$   
Anzahl an Schlüssel  $k$  in ST  $k < x$ .
- ▶  $\text{select}(i)$   
Der  $i$ -kleinste Schlüssel in ST (Null-basiert, d.h.,  $i \in [0..n)$ )

Wie beim Sortieren benötigen wir hier eine Vergleichsfunktion (Comparable oder Comparator)

## Ordered symbol table API

---

```
public class ST<Key extends Comparable<Key>, Value>
```

```
...
```

```
Key min() smallest key
```

```
Key max() largest key
```

```
Key floor(Key key) largest key less than or equal to key
```

```
Key ceiling(Key key) smallest key greater than or equal to key
```

```
int rank(Key key) number of keys less than key
```

```
Key select(int k) key of rank k
```

```
void deleteMin() delete smallest key
```

```
void deleteMax() delete largest key
```

```
int size(Key lo, Key hi) number of keys between lo and hi
```

```
Iterable<Key> keys() all keys, in sorted order
```

```
Iterable<Key> keys(Key lo, Key hi) keys between lo and hi, in sorted order
```

## 9.2 Konventionen in Java

## Conventions

---

- Values are not null. ← Java allows null value
- Method `get()` returns null if key not present.
- Method `put()` overwrites old value with new value.

### Intended consequences.

- Easy to implement `contains()`.

```
public boolean contains(Key key)
{ return get(key) != null; }
```

- Can implement lazy version of `delete()`.

```
public void delete(Key key)
{ put(key, null); }
```

## Keys and values

---

**Value type.** Any generic type.

**Key type: several natural assumptions.**

- Assume keys are `Comparable`, use `compareTo()`.
- Assume keys are any generic type, use `equals()` to test equality.
- Assume keys are any generic type, use `equals()` to test equality; use `hashCode()` to scramble key.

specify `Comparable` in API.

built-in to Java  
(stay tuned)

**Best practices.** Use immutable types for symbol table keys.

- Immutable in Java: `Integer`, `Double`, `String`, `java.io.File`, ...
- Mutable in Java: `StringBuilder`, `java.net.URL`, `arrays`, ...

## Equality test

---

All Java classes inherit a method `equals()`.

**Java requirements.** For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is true.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is false.

} equivalence relation

do `x` and `y` refer to the same object?

**Default implementation.** `(x == y)`

**Customized implementations.** Integer, Double, String, `java.io.File`, ...

**User-defined implementations.** Some care needed.

# Implementing equals for user-defined types


---

Seems easy.

```
public class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Date that)
    {
        if (this.day != that.day ) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year ) return false;
        return true;
    }
}
```

check that all significant  
fields are the same



# Implementing equals for user-defined types

Seems easy, but requires some care.

typically unsafe to use equals() with inheritance  
(would violate symmetry)

```
public final class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    ...

    public boolean equals(Object y)
    {
        if (y == this) return true;
        if (y == null) return false;
        if (y.getClass() != this.getClass())
            return false;

        Date that = (Date) y;
        if (this.day != that.day ) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year ) return false;
        return true;
    }
}
```

must be Object.  
Why? Experts still debate.

optimize for true object equality

check for null

objects must be in the same class  
(religion: getClass() vs. instanceof)




cast is guaranteed to succeed

check that all significant  
fields are the same


## Equals design

---

### "Standard" recipe for user-defined types.

- Optimization for reference equality.
- Check against `null`.
- Check that two objects are of the same type and cast.
- Compare each significant field:
  - if field is a primitive type, use `==`  but use `Double.compare()` with `double` (or otherwise deal with `-0.0` and `NaN`)
  - if field is an object, use `equals()`  apply rule recursively
  - if field is an array, apply to each entry  can use `Arrays.deepEquals(a, b)` but not `a.equals(b)`

### Best practices.

- No need to use calculated fields that depend on other fields.  e.g., cached Manhattan distance
- Compare fields mostly likely to differ first.
- Make `compareTo()` consistent with `equals()`.

`x.equals(y)` if and only if `(x.compareTo(y) == 0)`

## ST test client for traces

---

Build ST by associating value  $i$  with  $i^{\text{th}}$  string from standard input.

```
public static void main(String[] args)
{
    ST<String, Integer> st = new ST<String, Integer>();
    for (int i = 0; !StdIn.isEmpty(); i++)
    {
        String key = StdIn.readString();
        st.put(key, i);
    }
    for (String s : st.keys())
        StdOut.println(s + " " + st.get(s));
}
```

<b>keys</b>	S	E	A	R	C	H	E	X	A	M	P	L	E
<b>values</b>	0	1	2	3	4	5	6	7	8	9	10	11	12

**output**

A	8
C	4
E	12
H	5
L	11
M	9
P	10
R	3
S	0
X	7

## ST test client for analysis

---

**Frequency counter.** Read a sequence of strings from standard input and print out one that occurs with highest frequency.

```
% more tinyTale.txt
```

```
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
it was the spring of hope
it was the winter of despair
```

```
% java FrequencyCounter 1 < tinyTale.txt
it 10
```

← tiny example  
(60 words, 20 distinct)

```
% java FrequencyCounter 8 < tale.txt
business 122
```

← real example  
(135,635 words, 10,769 distinct)

```
% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

← real example  
(21,191,455 words, 534,580 distinct)

## Frequency counter implementation

---

```
public class FrequencyCounter
{
    public static void main(String[] args)
    {
        int minlen = Integer.parseInt(args[0]);
        ST<String, Integer> st = new ST<String, Integer>();
        while (!StdIn.isEmpty())
        {
            String word = StdIn.readString();
            if (word.length() < minlen) continue;
            if (!st.contains(word)) st.put(word, 1);
            else
                st.put(word, st.get(word) + 1);
        }
        String max = "";
        st.put(max, 0);
        for (String word : st.keys())
            if (st.get(word) > st.get(max))
                max = word;
        StdOut.println(max + " " + st.get(max));
    }
}
```

create ST

ignore short strings

read string and update frequency

print a string with max freq

## 9.3 Elementare Ideen für Symbol Tables



## Elementary ST implementations: summary

---

ST implementation	guarantee		average case		key interface
	search	insert	search hit	insert	
sequential search (unordered list)	$N$	$N$	$N/2$	$N$	<code>equals()</code>

**Challenge.** Efficient implementations of both search and insert.

# Lineare Listen als ST

## Unsortierte (verkettete) Liste:


 Effizientes put

  $\Theta(n)$  Zeit für get

↪ Zu langsam um nützlich zu sein

# Lineare Listen als ST

## Unsortierte (verkettete) Liste:


 Effizientes put

  $\Theta(n)$  Zeit für get

↪ Zu langsam um nützlich zu sein

## Sortierte *verkettete* Liste:

  $\Theta(n)$  Zeit für put

  $\Theta(n)$  Zeit für get (kein Random Access!)

↪ Zu langsam um nützlich zu sein

# Lineare Listen als ST

## Unsortierte (verkettete) Liste:


 Effizientes put

  $\Theta(n)$  Zeit für get

↪ Zu langsam um nützlich zu sein

## Sortierte *verkettete* Liste:

  $\Theta(n)$  Zeit für put

  $\Theta(n)$  Zeit für get (kein Random Access!)

↪ Zu langsam um nützlich zu sein

↪ *Hilft die sortierte Reihenfolge überhaupt nichts?!*

## Binary search in an ordered array

**Data structure.** Maintain an ordered array of key-value pairs.

**Rank helper function.** How many keys  $< k$ ?

**successful search for P**

			keys[]									
lo	hi	m	0	1	2	3	4	5	6	7	8	9
			A	C	E	H	L	M	P	R	S	X
0	9	4	A	C	E	H	L	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	R	S	X
5	6	5	A	C	E	H	L	M	P	R	S	X
6	6	6	A	C	E	H	L	M	P	R	S	X

*entries in black are a[lo..hi]*

*entry in red is a[m]*

*loop exits with keys[m] = P: return 6*

**unsuccessful search for Q**

lo	hi	m	0	1	2	3	4	5	6	7	8	9
			A	C	E	H	L	M	P	R	S	X
0	9	4	A	C	E	H	L	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	R	S	X
5	6	5	A	C	E	H	L	M	P	R	S	X
7	6	6	A	C	E	H	L	M	P	R	S	X

*loop exits with lo > hi: return 7*

## Binary search: Java implementation

---

```
public Value get(Key key)
{
    if (isEmpty()) return null;
    int i = rank(key);
    if (i < N && keys[i].compareTo(key) == 0) return vals[i];
    else return null;
}
```

```
private int rank(Key key) // number of keys < key
{
    int lo = 0, hi = N-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        int cmp = key.compareTo(keys[mid]);
        if (cmp < 0) hi = mid - 1;
        else if (cmp > 0) lo = mid + 1;
        else if (cmp == 0) return mid;
    }
    return lo;
}
```

## Binary search: trace of standard indexing client

**Problem.** To insert, need to shift all greater keys over.

		keys[]											vals[]									
key	value	0	1	2	3	4	5	6	7	8	9	N	0	1	2	3	4	5	6	7	8	9
S	0	S										1	0									
E	1	E	S									2	1	0								
A	2	A	E	S								3	2	1	0							
R	3	A	E	R	S							4	2	1	3	0						
C	4	A	C	E	R	S						5	2	4	1	3	0					
H	5	A	C	E	H	R	S					6	2	4	1	5	3	0				
E	6	A	C	E	H	R	S					6	2	4	6	5	3	0				
X	7	A	C	E	H	R	S	X				7	2	4	6	5	3	0	7			
A	8	A	C	E	H	R	S	X				7	8	4	6	5	3	0	7			
M	9	A	C	E	H	M	R	S	X			8	8	4	6	5	9	3	0	7		
P	10	A	C	E	H	M	P	R	S	X		9	8	4	6	5	9	10	3	0	7	
L	11	A	C	E	H	L	M	P	R	S	X	10	8	4	6	5	11	9	10	3	0	7
E	12	A	C	E	H	L	M	P	R	S	X	10	8	4	12	5	11	9	10	3	0	7
		A	C	E	H	L	M	P	R	S	X		8	4	12	5	11	9	10	3	0	7

*entries in red were inserted*  
*entries in gray did not move*  
*entries in black moved to the right*  
*circled entries are changed values*

## Elementary ST implementations: summary

---

ST implementation	guarantee		average case		key interface
	search	insert	search hit	insert	
sequential search (unordered list)	$N$	$N$	$N/2$	$N$	<code>equals()</code>
binary search (ordered array)	$\log N$	$N$	$\log N$	$N/2$	<code>compareTo()</code>

**Challenge.** Efficient implementations of both search and insert.

## Binary search: ordered symbol table operations summary

---

	sequential search	binary search
search	$N$	$\log N$
insert / delete	$N$	$N$
min / max	$N$	1
floor / ceiling	$N$	$\log N$
rank	$N$	$\log N$
select	$N$	1
ordered iteration	$N \log N$	$N$

**order of growth of the running time for ordered symbol table operations**

# Diskussion

*Symbol Table Dilemma: Auf Basis Linearer Listen haben wir scheinbar einen Widerspruch*

- ▶ Um schnell suchen zu können, brauchen wir sortierte Speicherung (und Random Access!)
- ▶ Um schnell einfügen zu können, dürfen wir keine Sortierung verlangen

# Diskussion

*Symbol Table Dilemma: Auf Basis Linearer Listen haben wir scheinbar einen Widerspruch*

- ▶ Um schnell suchen zu können, brauchen wir sortierte Speicherung (und Random Access!)
- ▶ Um schnell einfügen zu können, dürfen wir keine Sortierung verlangen

~> *Es scheint also ein etwas kniffligeres Problem zu sein . . .*

~> *Schauen wir uns erst einmal einen Spezialfall an!*

## 9.4 Priority Queues und Heaps

# Priority Queue ADT

Erlauben hier nur Zugriff auf größtes Element

**(Max-oriented) Priority Queue (MaxPQ):**

- ▶  $\text{construct}(A[0..n])$   
Konstruiere PQ für Elemente in  $A[0..n]$ .
- ▶  $\text{insert}(x, p)$   
Füge  $x$  mit Priorität  $p$  in PQ ein  $\approx \text{put}(p, x)$
- ▶  $\text{max}()$   
Element mit größter priority. (Verändert PQ nicht.)
- ▶  $\text{delMax}()$   
Entfernt Element mit größter Priorität.
- ▶  $\text{changeKey}(x, p')$   
Aktualisiere die Priorität von  $x$  zu  $p'$ .  
Manchmal nur erlaubt für *höhere* neue Priorität.
- ▶  $\text{isEmpty}()$

Spezialfall von ST mit vielen Anwendungen!



# Priority Queue ADT – min-oriented version

Erlauben hier nur Zugriff auf größtes Element

~~(Max-oriented)~~ <sup>Min</sup> Priority Queue (~~MaxPQ~~):

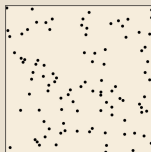
- ▶ `construct( $A[0..n]$ )`  
Konstruiere PQ für Elemente in  $A[0..n]$ .
- ▶ `insert( $x, p$ )`  
Füge  $x$  mit Priorität  $p$  in PQ ein  $\approx$  `put( $p, x$ )`
- ▶ ~~`max()`~~ <sup>min</sup>  
Element mit ~~größter~~ <sup>kleinster</sup> priority. (Verändert PQ nicht.)
- ▶ ~~`delMax()`~~ <sup>Min</sup>  
Entfernt Element mit ~~größter~~ <sup>kleinster</sup> Priorität.
- ▶ `changeKey( $x, p'$ )`  
Aktualisiere die Priorität von  $x$  zu  $p'$ .  
Manchmal nur erlaubt für ~~höhere~~ <sup>niedrigere</sup> neue Priorität.
- ▶ `isEmpty()`

Spezialfall von ST mit vielen Anwendungen!



## Priority queue: applications

- Event-driven simulation. [ customers in a line, colliding particles ]
- Numerical computation. [ reducing roundoff error ]
- Discrete optimization. [ bin packing, scheduling ]
- Artificial intelligence. [ A\* search ]
- Computer networks. [ web cache ]
- Operating systems. [ load balancing, interrupt handling ]
- Data compression. [ Huffman codes ]
- Graph searching. [ Dijkstra's algorithm, Prim's algorithm ]
- Number theory. [ sum of powers ]
- Spam filtering. [ Bayesian spam filter ]
- Statistics. [ online median in data stream ]




8	4	7
1	5	6
3	2	

## Priority queue API

---

**Requirement.** Items are generic; they must also be Comparable.

Key must be Comparable  
(bounded type parameter)



<code>public class MaxPQ&lt;Key&gt; extends Comparable&lt;Key&gt;&gt;</code>	
<code>MaxPQ()</code>	<i>create an empty priority queue</i>
<code>MaxPQ(Key[] a)</code>	<i>create a priority queue with given keys</i>
<code>void insert(Key v)</code>	<i>insert a key into the priority queue</i>
<code>Key delMax()</code>	<i>return and remove a largest key</i>
<code>boolean isEmpty()</code>	<i>is the priority queue empty?</i>
<code>Key max()</code>	<i>return a largest key</i>
<code>int size()</code>	<i>number of entries in the priority queue</i>

**Note.** Duplicate keys allowed; `delMax()` picks any maximum key.

# Primitive Ideen

Auch hier gilt: Lineare Listen tun's nicht ...

- ▶ unsortierte Liste  $\rightsquigarrow$   $\Theta(1)$  insert, aber  $\Theta(n)$  delMax
- ▶ sortierte Liste  $\rightsquigarrow$   $\Theta(1)$  delMax, aber  $\Theta(n)$  insert

# Primitive Ideen

**Auch hier gilt: Lineare Listen tun's nicht ...**

- ▶ unsortierte Liste  $\rightsquigarrow$   $\Theta(1)$  insert, aber  $\Theta(n)$  delMax
- ▶ sortierte Liste  $\rightsquigarrow$   $\Theta(1)$  delMax, aber  $\Theta(n)$  insert

*Können wir etwas zwischen diesen Extremen haben? Eine nur "ein Bisschen sortierte" Liste?*

# Primitive Ideen

Auch hier gilt: **Lineare Listen tun's nicht ...**

- ▶ unsortierte Liste  $\rightsquigarrow \Theta(1)$  insert, aber  $\Theta(n)$  delMax
- ▶ sortierte Liste  $\rightsquigarrow \Theta(1)$  delMax, aber  $\Theta(n)$  insert

*Können wir etwas zwischen diesen Extremen haben? Eine nur "ein Bisschen sortierte" Liste?*

Tatsächlich: **Ja!** *Binäre Heaps.*

*Array Sichtweise*

Heap = array  $A$  mit  
 $\forall i \in [n] : A[\lfloor i/2 \rfloor] \geq A[i]$

# Primitive Ideen

Auch hier gilt: **Lineare Listen tun's nicht ...**

- ▶ unsortierte Liste  $\rightsquigarrow \Theta(1)$  insert, aber  $\Theta(n)$  delMax
- ▶ sortierte Liste  $\rightsquigarrow \Theta(1)$  delMax, aber  $\Theta(n)$  insert

Können wir etwas zwischen diesen Extremen haben? Eine nur "ein Bisschen sortierte" Liste?

Tatsächlich: **Ja!** *Binäre Heaps.*

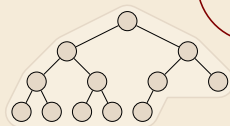
*Array Sichtweise*

Heap = array  $A$  mit  
 $\forall i \in [n] : A[\lfloor i/2 \rfloor] \geq A[i]$

≡  
↑  
store nodes  
in level order  
in  $A[1..n]$

*Baum Sichtweise*

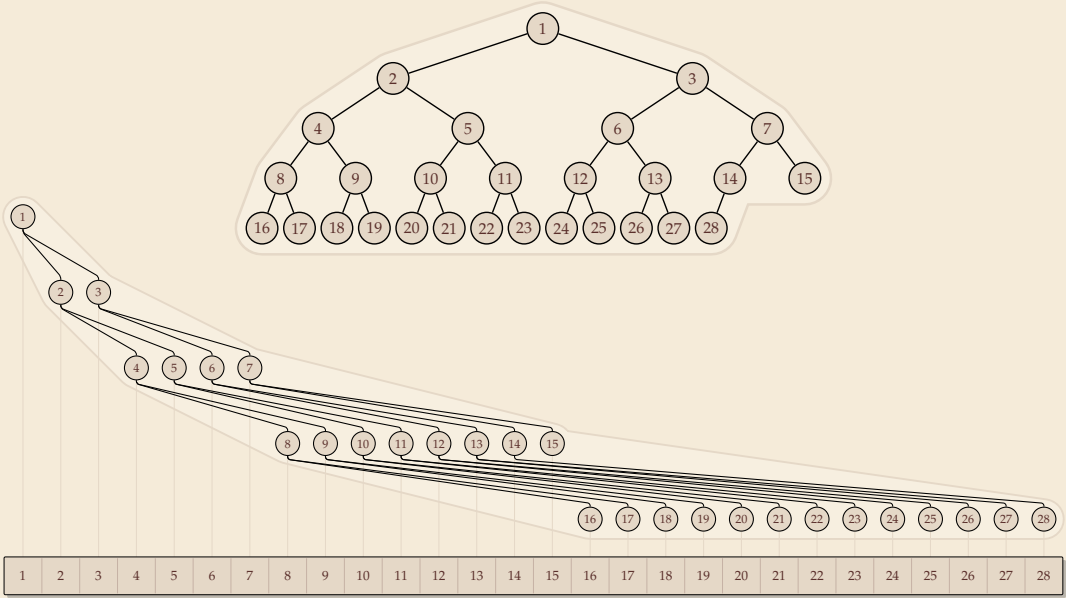
Heap = Baum mit  
(i) vollständiger binärer Baum  
(ii) heap-geordnet



Vater  $\geq$  Kinder

alle Level voll  
letztes Level  
„linksbündig“

# Implicit Complete Binary Trees

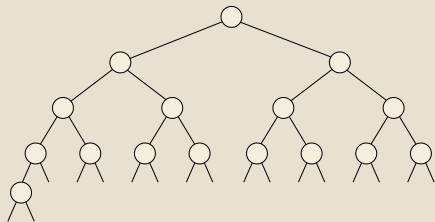


## Complete binary tree

---

**Binary tree.** Empty or node with links to left and right binary trees.

**Complete tree.** Perfectly balanced, except for bottom level.



complete binary tree with  $n = 16$  nodes (height = 4)

**Property.** Height of complete binary tree with  $n$  nodes is  $\lfloor \lg n \rfloor$ .

**Pf.** Height increases only when  $n$  is a power of 2.

## A complete binary tree in nature

---



Hyphaene Compressa - Doum Palm

© Shlomit Pinter

## Binary heap: representation

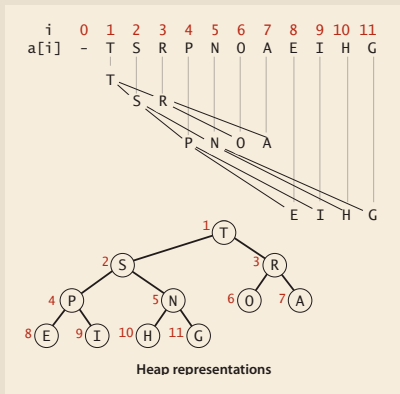
**Binary heap.** Array representation of a heap-ordered complete binary tree.

**Heap-ordered binary tree.**

- Keys in nodes.
- Parent's key no smaller than children's keys.

**Array representation.**

- Indices start at 1.
- Take nodes in **level** order.
- No explicit links needed!

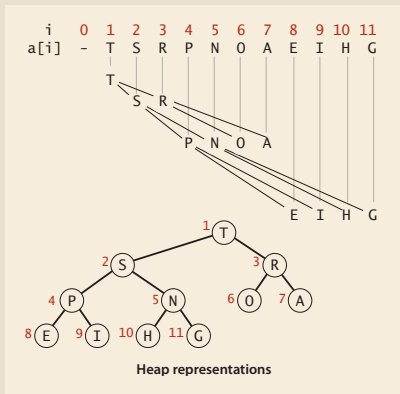


## Binary heap: properties

**Proposition.** Largest key is  $a[1]$ , which is root of binary tree.

**Proposition.** Can use array indices to move through tree.

- Parent of node at  $k$  is at  $k/2$ .
- Children of node at  $k$  are at  $2k$  and  $2k+1$ .



## 9.5 Operationen in binären Heaps

## Binary heap: promotion

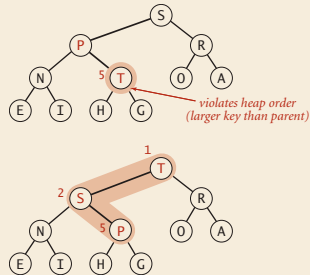
**Scenario.** A key becomes **larger** than its parent's key.

To eliminate the violation:

- Exchange key in child with key in parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
    while (k > 1 && less(k/2, k))
    {
        exch(k, k/2);
        k = k/2;
    }
}
```

parent of node at k is at k/2



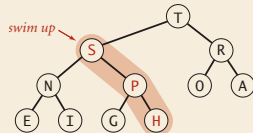
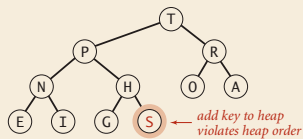
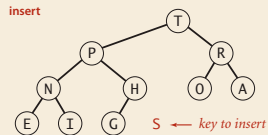
**Peter principle.** Node promoted to level of incompetence.

## Binary heap: insertion

**Insert.** Add node at end, then swim it up.

**Cost.** At most  $1 + \lg n$  compares.

```
public void insert(Key x)
{
    pq[++n] = x;
    swim(n);
}
```



## Binary heap: demotion

**Scenario.** A key becomes **smaller** than one (or both) of its children's.

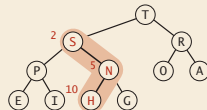
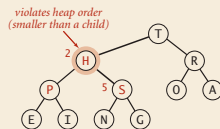
To eliminate the violation:

why not smaller child?

- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

```
private void sink(int k)
{
    while (2*k <= n)
    {
        int j = 2*k;
        if (j < n && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

children of node at k are  $2*k$  and  $2*k+1$



Top-down reheapify (sink)

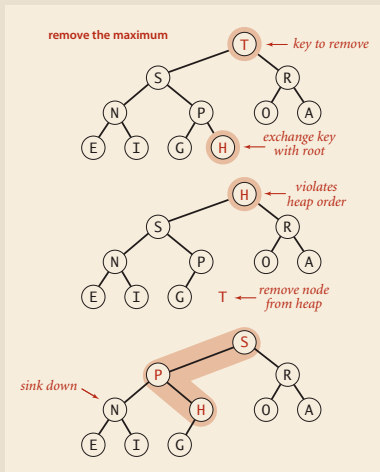
**Power struggle.** Better subordinate promoted.

## Binary heap: delete the maximum

**Delete max.** Exchange root with node at end, then sink it down.

**Cost.** At most  $2 \lg n$  compares.

```
public Key delMax()
{
    Key max = pq[1];
    exch(1, n--);
    sink(1);
    pq[n+1] = null; ← prevent loitering
    return max;
}
```



## Binary heap: Java implementation

---

```
public class MaxPQ<Key extends Comparable<Key>>
{
    private Key[] pq;
    private int n;

    public MaxPQ(int capacity)
    { pq = (Key[]) new Comparable[capacity+1]; }

    public boolean isEmpty()
    { return n == 0; }
    public void insert(Key key) // see previous code
    public Key delMax() // see previous code

    private void swim(int k) // see previous code
    private void sink(int k) // see previous code

    private boolean less(int i, int j)
    { return pq[i].compareTo(pq[j]) < 0; }
    private void exch(int i, int j)
    { Key t = pq[i]; pq[i] = pq[j]; pq[j] = t; }
}
```

← fixed capacity  
(for simplicity)

← PQ ops

← heap helper functions

← array helper functions

## Priority queue: implementations cost summary

---

implementation	insert	del max	max
<b>unordered array</b>	1	$n$	$n$
<b>ordered array</b>	$n$	1	1
<b>binary heap</b>	$\log n$	$\log n$	1

order-of-growth of running time for priority queue with  $n$  items

## Binary heap: considerations

---

### Underflow and overflow.

- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use resizing array.

leads to  $\log n$   
amortized time per op  
(how to make worst case?)

### Minimum-oriented priority queue.

- Replace `less()` with `greater()`.
- Implement `greater()`.

### Other operations.

- Remove an arbitrary item.
  - Change the priority of an item.
- can implement efficiently with `sink()` and `swim()`  
[ stay tuned for Prim/Dijkstra ]

### Immutability of keys.

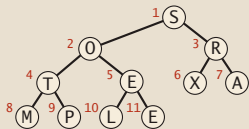
- Assumption: client does not change keys while they're on the PQ.
- Best practice: use immutable keys.

# Heapsort

## Basic plan for in-place sort.

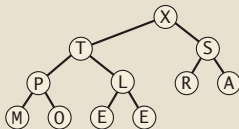
- View input array as a complete binary tree.
- Heap construction: build a max-heap with all  $n$  keys.
- Sortdown: repeatedly remove the maximum key.

keys in arbitrary order



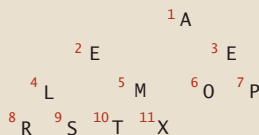
1	2	3	4	5	6	7	8	9	10	11
S	O	R	T	E	X	A	M	P	L	E

build max heap  
(in place)



1	2	3	4	5	6	7	8	9	10	11
X	T	S	P	L	R	A	M	O	E	E

sorted result  
(in place)

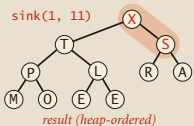
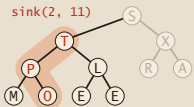
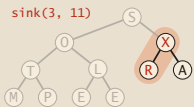
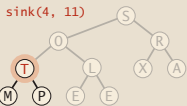
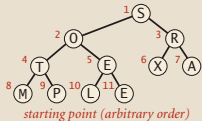


1	2	3	4	5	6	7	8	9	10	11
A	E	E	L	M	O	P	R	S	T	X

# Heapsort: heap construction

First pass. Build heap using bottom-up method.

```
for (int k = n/2; k >= 1; k--)  
    sink(a, k, n);
```

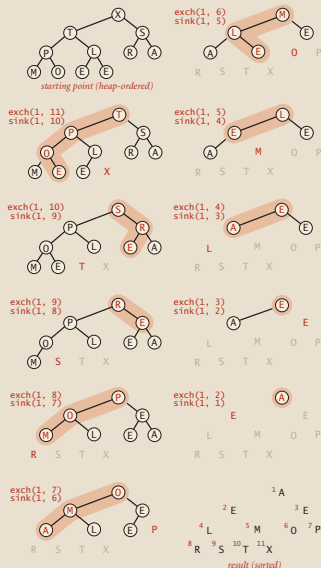


# Heapsort: sortdown

## Second pass.

- Remove the maximum, one at a time.
- Leave in array, instead of nulling out.

```
while (n > 1)
{
    exch(a, 1, n--);
    sink(a, 1, n);
}
```



## Heapsort: Java implementation

```
public class Heap
{
    public static void sort(Comparable[] a)
    {
        int n = a.length;
        for (int k = n/2; k >= 1; k--)
            sink(a, k, n);
        while (n > 1)
        {
            exch(a, 1, n);
            sink(a, 1, --n);
        }
    }
}

private static void sink(Comparable[] a, int k, int n)
{ /* as before */ }

private static boolean less(Comparable[] a, int i, int j)
{ /* as before */ }

private static void exch(Object[] a, int i, int j)
{ /* as before */ }
}
```

but make static (and pass arguments)

but convert from 1-based indexing to 0-base indexing

## Sorting algorithms: summary

	inplace?	stable?	best	average	worst	remarks
selection	✓		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$n$ exchanges
insertion	✓	✓	$n$	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small $n$ or partially ordered
shell	✓		$n \log_3 n$	?	$c n^{3/2}$	tight code; subquadratic
merge		✓	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
timsort		✓	$n$	$n \lg n$	$n \lg n$	improves mergesort when preexisting order
quick	✓		$n \lg n$	$2 n \ln n$	$\frac{1}{2} n^2$	$n \log n$ probabilistic guarantee; fastest in practice
3-way quick	✓		$n$	$2 n \ln n$	$\frac{1}{2} n^2$	improves quicksort when duplicate keys
heap	✓		$3 n$	$2 n \lg n$	$2 n \lg n$	$n \log n$ guarantee; in-place
?	✓	✓	$n$	$n \lg n$	$n \lg n$	holy sorting grail

# Binary heap – Zusammenfassung

Operation	Laufzeit
<code>construct(<math>A[1..n]</math>)</code>	$O(n)$
<code>max()</code>	$O(1)$
<code>insert(<math>x, p</math>)</code>	$O(\log n)$
<code>delMax()</code>	$O(\log n)$
<code>changeKey(<math>x, p'</math>)</code>	$O(\log n)$
<code>isEmpty()</code>	$O(1)$
<code>size()</code>	$O(1)$

## 9.6 Binäre Suchbäume

## Zurück zu Symbol Tables

*Binäre Bäume scheinen die nötige Flexibilität mitzubringen!*

*Allerdings ist die Form von Heaps für volle ST zu starr.*

↪ Erlaube beliebige Form von binären Bäumen

## Zurück zu Symbol Tables

*Binäre Bäume scheinen die nötige Flexibilität mitzubringen!*

*Allerdings ist die Form von Heaps für volle ST zu starr.*

↪ Erlaube beliebige Form von binären Bäumen

*Umgekehrt ist die Heap-Ordnung nicht streng genug.*

▶ Maximum ist in der Wurzel, aber wo ist das Minimum? der Median?  $x$ ?

## Zurück zu Symbol Tables

*Binäre Bäume scheinen die nötige Flexibilität mitzubringen!*

*Allerdings ist die Form von Heaps für volle ST zu starr.*

↪ Erlaube beliebige Form von binären Bäumen

*Umgekehrt ist die Heap-Ordnung nicht streng genug.*

- ▶ Maximum ist in der Wurzel, aber wo ist das Minimum? der Median?  $x$ ?
- ▶ Müssen zurück zu **sortierter Reihenfolge** für effiziente Suche

## Zurück zu Symbol Tables

*Binäre Bäume scheinen die nötige Flexibilität mitzubringen!*

*Allerdings ist die Form von Heaps für volle ST zu starr.*

↪ Erlaube beliebige Form von binären Bäumen

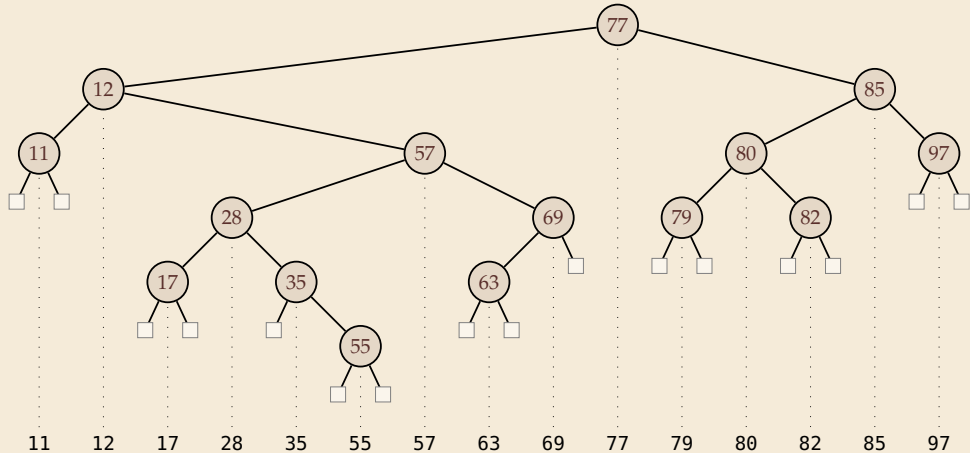
*Umgekehrt ist die Heap-Ordnung nicht streng genug.*

- ▶ Maximum ist in der Wurzel, aber wo ist das Minimum? der Median?  $x$ ?
- ▶ Müssen zurück zu **sortierter Reihenfolge** für effiziente Suche
- ▶ *Aber was heißt denn sortiert in einem Baum?*

# Binärer Suchbaum

Suchbaumeigenschaft:

alle Schlüssel im **linken** Teilbaum  $\leq$  Schlüssel in **Wurzel**  $\leq$  Schlüssel im **rechten** Teilbaum



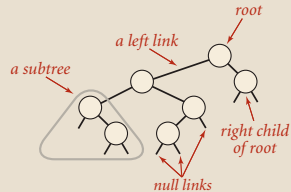
*Wir können (eine Art von) binärer Suche in diesem Baum machen!*

# Binary search trees

**Definition.** A BST is a **binary tree in symmetric order**.

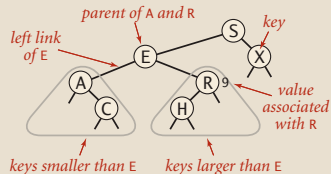
A binary tree is either:

- Empty.
- Two disjoint binary trees (left and right).



**Symmetric order.** Each node has a key, and every node's key is:

- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.



## BST representation in Java

**Java definition.** A BST is a reference to a root Node.

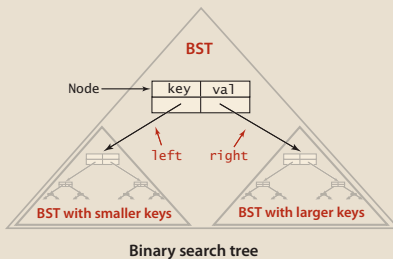
A Node is composed of four fields:

- A Key and a Value.
- A reference to the left and right subtree.

↑ smaller keys      ↑ larger keys

```
private class Node
{
    private Key key;
    private Value val;
    private Node left, right;
    public Node(Key key, Value val)
    {
        this.key = key;
        this.val = val;
    }
}
```

Key and Value are generic types; Key is Comparable



## BST implementation (skeleton)

---

```
public class BST<Key extends Comparable<Key>, Value>
{
    private Node root;

    private class Node
    { /* see previous slide */ }

    public void put(Key key, Value val)
    { /* see next slides */ }

    public Value get(Key key)
    { /* see next slides */ }

    public void delete(Key key)
    { /* see next slides */ }

    public Iterable<Key> iterator()
    { /* see next slides */ }
}
```

← root of BST

## BST search: Java implementation

---

**Get.** Return value corresponding to given key, or null if no such key.

```
public Value get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```

**Cost.** Number of compares is equal to 1 + depth of node.

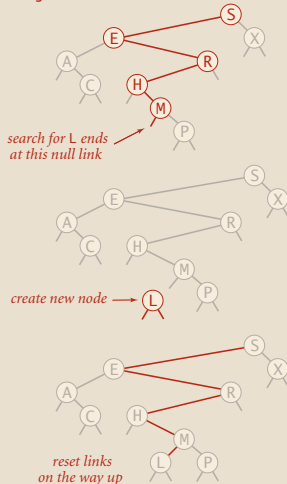
## BST insert

**Put.** Associate value with key.

Search for key, then two cases:

- Key in tree  $\Rightarrow$  reset value.
- Key not in tree  $\Rightarrow$  add new node.

inserting L



Insertion into a BST

## BST insert: Java implementation

---

**Put.** Associate value with key.

```
public void put(Key key, Value val)
{ root = put(root, key, val); }

private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = put(x.left, key, val);
    else if (cmp > 0)
        x.right = put(x.right, key, val);
    else if (cmp == 0)
        x.val = val;
    return x;
}
```

concise, but tricky,  
recursive code;  
read carefully!

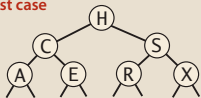
**Cost.** Number of compares is equal to  $1 + \text{depth of node}$ .

## Tree shape

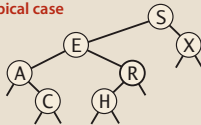
---

- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to  $1 + \text{depth of node}$ .

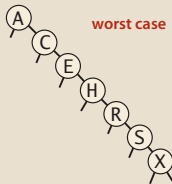
best case



typical case



worst case

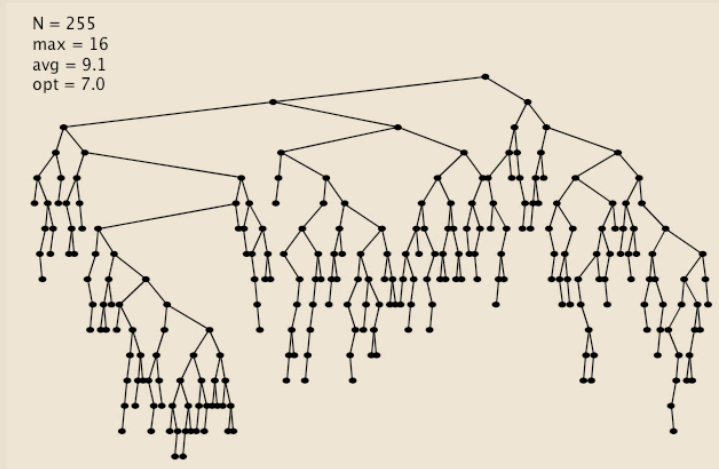


**Bottom line.** Tree shape depends on order of insertion.

## BST insertion: random order visualization

---

Ex. Insert keys in random order.



## Sorting with a binary heap

---

Q. What is this sorting algorithm?

0. Shuffle the array of keys.
1. Insert all keys into a BST.
2. Do an inorder traversal of BST.

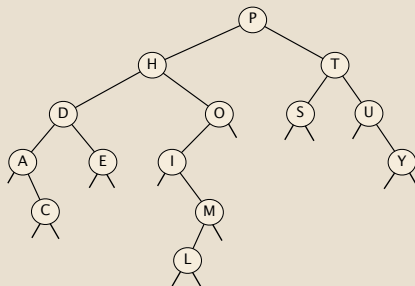
A. It's not a sorting algorithm (if there are duplicate keys)!

Q. OK, so what if there are no duplicate keys?

Q. What are its properties?

## Correspondence between BSTs and quicksort partitioning

0	1	2	3	4	5	6	7	8	9	10	11	12	13
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
P	S	E	U	D	O	M	Y	T	H	I	C	A	L
H	L	E	A	D	O	M	C	I	P	T	Y	U	S
D	C	E	A	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	O	M	L	I	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	M	L	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	T	Y	U	S
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y
A	C	D	E	H	I	L	M	O	P	S	T	U	Y



**Remark.** Correspondence is 1-1 if array has no duplicate keys.

## BSTs: mathematical analysis

---

**Proposition.** If  $N$  distinct keys are inserted into a BST in **random** order, the expected number of compares for a search/insert is  $\sim 2 \ln N$ .

**Pf.** 1–1 correspondence with quicksort partitioning.

**Proposition.** [Reed, 2003] If  $N$  distinct keys are inserted in random order, expected height of tree is  $\sim 4.311 \ln N$ .

### How Tall is a Tree?

Bruce Reed  
CNRS, Paris, France  
reed@moka.ccr.jussieu.fr

#### ABSTRACT

Let  $H_n$  be the height of a random binary search tree on  $n$  nodes. We show that there exists constants  $\alpha = 4.31107 \dots$  and  $\beta = 1.95 \dots$  such that  $E(H_n) = \alpha \log n - \beta \log \log n + O(1)$ . We also show that  $\text{Var}(H_n) = O(1)$ .

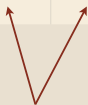
**But...** Worst-case height is  $N$ .

[ exponentially small chance when keys are inserted in random order ]

## ST implementations: summary

---

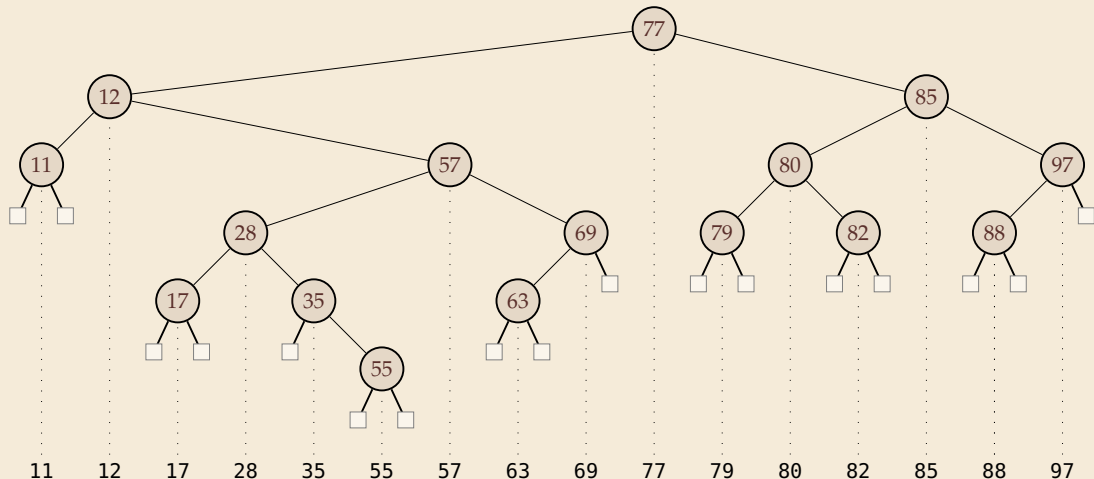
implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	$N$	$N$	$\frac{1}{2} N$	$N$	<code>equals()</code>
binary search (ordered array)	$\lg N$	$N$	$\lg N$	$\frac{1}{2} N$	<code>compareTo()</code>
BST	$N$	$N$	$1.39 \lg N$	$1.39 \lg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



Why not shuffle to ensure a (probabilistic) guarantee of  $4.311 \ln N$ ?

# BST delete

- ▶ Easy case: remove leaf, e. g., 11  $\rightsquigarrow$  replace by null
- ▶ Medium case: remove unary, e. g., 69  $\rightsquigarrow$  replace by unique child
- ▶ Hard case: remove binary, e. g., 85  $\rightsquigarrow$  swap with predecessor, recurse

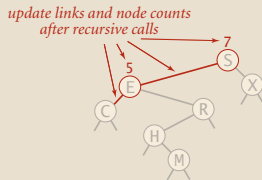
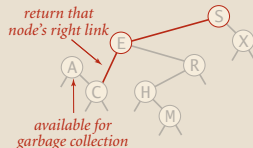
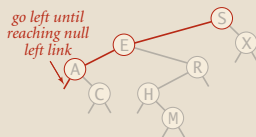


## Deleting the minimum

To delete the minimum key:

- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

```
public void deleteMin()  
{ root = deleteMin(root); }  
  
private Node deleteMin(Node x)  
{  
    if (x.left == null) return x.right;  
    x.left = deleteMin(x.left);  
    x.count = 1 + size(x.left) + size(x.right);  
    return x;  
}
```



## Hibbard deletion: Java implementation

```
public void delete(Key key)
{ root = delete(root, key); }

private Node delete(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = delete(x.left, key); ← search for key
    else if (cmp > 0) x.right = delete(x.right, key);
    else {
        if (x.right == null) return x.left; ← no right child
        if (x.left == null) return x.right; ← no left child

        Node t = x;
        x = min(t.right); ← replace with
        x.right = deleteMin(t.right); ← successor
        x.left = t.left;

    }
    x.count = size(x.left) + size(x.right) + 1; ← update subtree
    return x; ← counts
}
}
```

## 9.7 Augmented BSTs

# Rank-basierte Operationen

Bisher: insert, get/put, delete

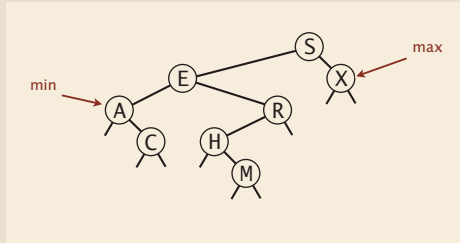
Was ist mit den anderen Ordered-Symbol-Table-Operationen?

## Minimum and maximum

---

**Minimum.** Smallest key in table.

**Maximum.** Largest key in table.



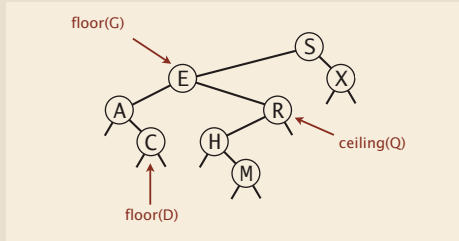
Q. How to find the min / max?

## Floor and ceiling

---

**Floor.** Largest key  $\leq$  a given key.

**Ceiling.** Smallest key  $\geq$  a given key.



**Q.** How to find the floor / ceiling?

# Computing the floor

Case 1. [ $k$  equals the key in the node]

The floor of  $k$  is  $k$ .

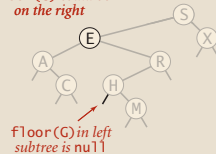
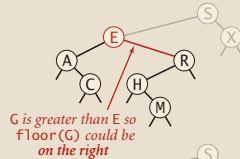
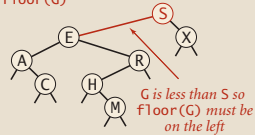
Case 2. [ $k$  is less than the key in the node]

The floor of  $k$  is in the left subtree.

Case 3. [ $k$  is greater than the key in the node]

The floor of  $k$  is in the right subtree  
(if there is any key  $\leq k$  in right subtree);  
otherwise it is the key in the node.

finding floor( $G$ )



# Computing the floor

```
public Key floor(Key key)
{
    Node x = floor(root, key);
    if (x == null) return null;
    return x.key;
}

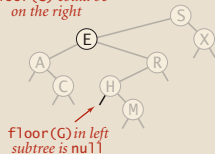
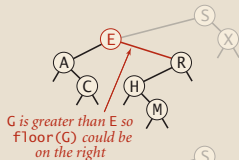
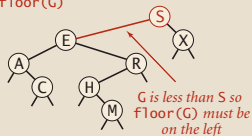
private Node floor(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);

    if (cmp == 0) return x;

    if (cmp < 0) return floor(x.left, key);

    Node t = floor(x.right, key);
    if (t != null) return t;
    else return x;
}
```

finding floor(G)



# Rank-basierte Operationen

Soweit, so gut.

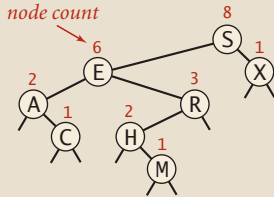
*Aber wie finden wir das 42. kleinste Element??*

## Rank and select

---


Q. How to implement `rank()` and `select()` efficiently?

A. In each node, we store the number of nodes in the subtree rooted at that node; to implement `size()`, return the count at the root.



## BST implementation: subtree counts


```
private class Node
{
    private Key key;
    private Value val;
    private Node left;
    private Node right;
    private int count;
}
```



number of nodes in subtree


```
public int size()
{ return size(root); }
```

```
private int size(Node x)
{
    if (x == null) return 0;
    return x.count;
}
```



ok to call  
when x is null

```
private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val, 1);
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = put(x.left, key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else if (cmp == 0) x.val = val;
    x.count = 1 + size(x.left) + size(x.right);
    return x;
}
```

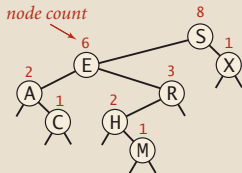


initialize subtree  
count to 1

# Rank

Rank. How many keys  $< k$ ?

Easy recursive algorithm (3 cases!)




```
public int rank(Key key)
{ return rank(key, root); }

private int rank(Key key, Node x)
{
    if (x == null) return 0;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return rank(key, x.left);
    else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
    else if (cmp == 0) return size(x.left);
}
```

## BST: ordered symbol table operations summary

	sequential search	binary search	BST
search	$N$	$\lg N$	$h$
insert	$N$	$N$	$h$
min / max	$N$	1	$h$
floor / ceiling	$N$	$\lg N$	$h$
rank	$N$	$\lg N$	$h$
select	$N$	1	$h$
ordered iteration	$N \log N$	$N$	$N$

$h$  = height of BST  
(proportional to  $\log N$   
if keys inserted in random order)



**order of growth of running time of ordered symbol table operations**

# Augmentierte BSTs – Zusammenfassung

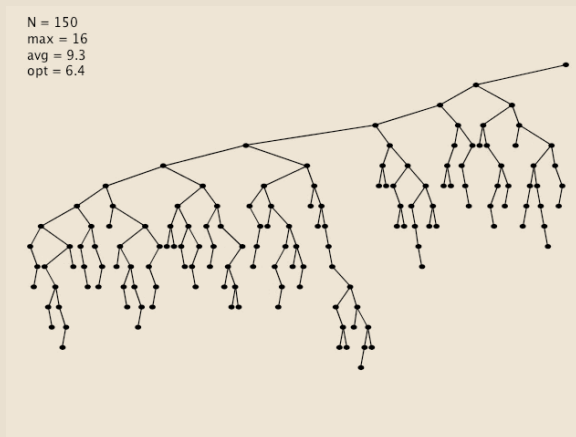
Mit  $h$  der Höhe des BST erhalten wir

Operation	Laufzeit
<code>construct(<math>A[1..n]</math>)</code>	$O(nh)$
<code>put(<math>k, v</math>)</code>	$O(h)$
<code>get(<math>k</math>)</code>	$O(h)$
<code>delete(<math>k</math>)</code>	$O(h)$
<code>contains(<math>k</math>)</code>	$O(h)$
<code>isEmpty()</code>	$O(1)$
<code>size()</code>	$O(1)$

## Hibbard deletion: analysis

---

Unsatisfactory solution. Not symmetric.



Surprising consequence. Trees not random (!)  $\Rightarrow \sqrt{N}$  per op.

Longstanding open problem. Simple and efficient delete for BSTs.

## ST implementations: summary

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

other operations also become  $\sqrt{N}$   
if deletions allowed

Next lecture. Guarantee logarithmic performance for all operations.

## 9.8 **Balancierte Suchbäume: 2–3-Bäume**

## Symbol table review

---

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
<b>sequential search (unordered list)</b>	$N$	$N$	$N$	$\frac{1}{2} N$	$N$	$\frac{1}{2} N$		<code>equals()</code>
<b>binary search (ordered array)</b>	$\lg N$	$N$	$N$	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	<code>compareTo()</code>
<b>BST</b>	$N$	$N$	$N$	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✓	<code>compareTo()</code>
<b>goal</b>	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✓	<code>compareTo()</code>

**Challenge.** Guarantee performance.

**This lecture.** 2-3 trees, left-leaning red-black BSTs, B-trees.

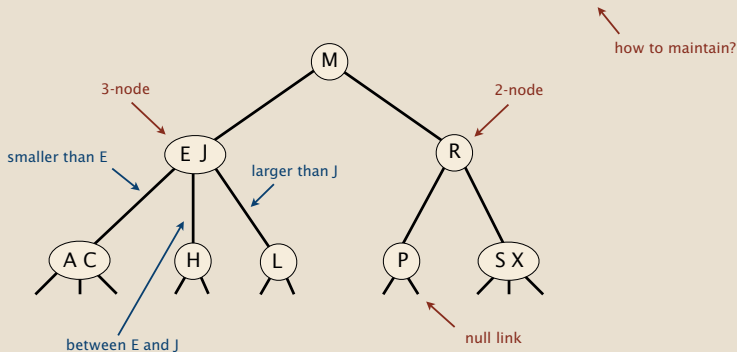
## 2-3 tree

Allow 1 or 2 keys per node.

- 2-node: one key, two children.
- 3-node: two keys, three children.

**Symmetric order.** Inorder traversal yields keys in ascending order.

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



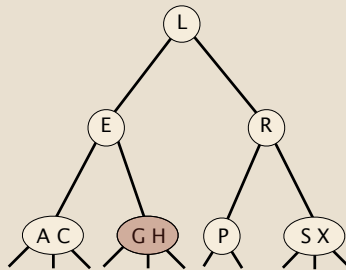
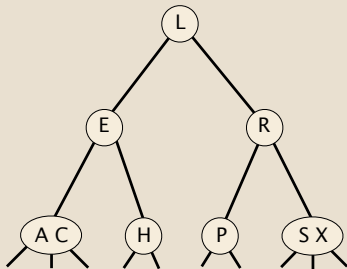
## Insertion into a 2-3 tree

---

### Insertion into a 2-node at bottom.

- Add new key to 2-node to create a 3-node.

insert G



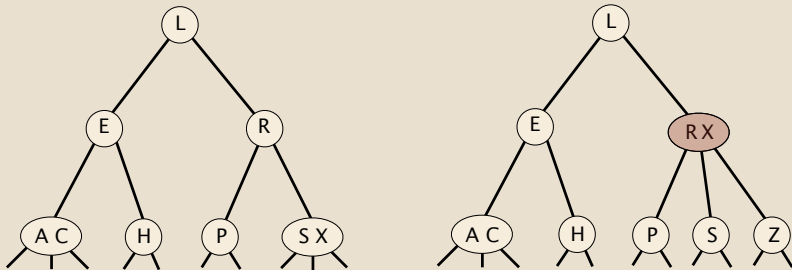
## Insertion into a 2-3 tree

---

### Insertion into a 3-node at bottom.

- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

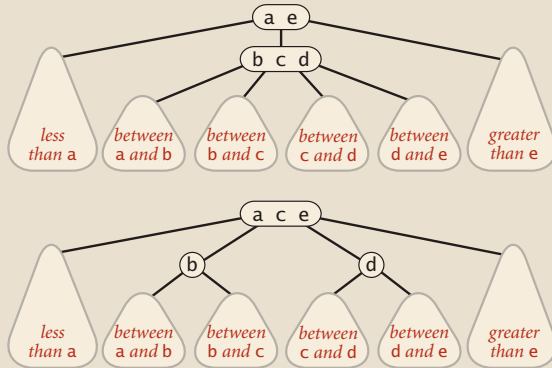
insert Z



## Local transformations in a 2-3 tree

---

Splitting a 4-node is a **local** transformation: constant number of operations.



## Global properties in a 2-3 tree

**Invariants.** Maintains symmetric order and perfect balance.

**Pf.** Each transformation maintains symmetric order and perfect balance.

*root*



*parent is a 3-node*

*left*



*parent is a 2-node*

*left*



*middle*



*right*



*right*



## 2-3 tree: performance

---

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



**Tree height.**

- Worst case:
- Best case:

## 2-3 tree: performance

---

**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.

**Bottom line.** Guaranteed **logarithmic** performance for search and insert.

## ST implementations: summary

implementation	guarantee			average case			ordered ops?	key interface
	search	insert	delete	search hit	insert	delete		
<b>sequential search (unordered list)</b>	$N$	$N$	$N$	$\frac{1}{2} N$	$N$	$\frac{1}{2} N$		<code>equals()</code>
<b>binary search (ordered array)</b>	$\lg N$	$N$	$N$	$\lg N$	$\frac{1}{2} N$	$\frac{1}{2} N$	✓	<code>compareTo()</code>
<b>BST</b>	$N$	$N$	$N$	$1.39 \lg N$	$1.39 \lg N$	$\sqrt{N}$	✓	<code>compareTo()</code>
<b>2-3 tree</b>	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	$c \lg N$	✓	<code>compareTo()</code>



constant  $c$  depend upon implementation

## 2-3 tree: implementation?

---

Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

fantasy code

```
public void put(Key key, Value val)
{
    Node x = root;
    while (x.getTheCorrectChild(key) != null)
    {
        x = x.getTheCorrectChildKey();
        if (x.is4Node()) x.split();
    }
    if (x.is2Node()) x.make3Node(key, val);
    else if (x.is3Node()) x.make4Node(key, val);
}
```

Bottom line. Could do it, but there's a better way.

## 9.9 **Balancierte Suchbäume: Red-Black-Trees**