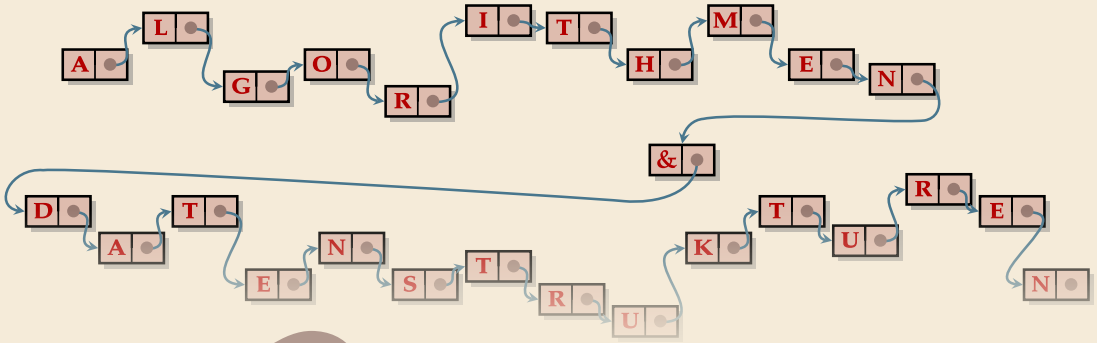


9 Suchbäume

Algorithmen & Datenstrukturen · Sommersemester 2026

Prof. Dr. Sebastian Wild

CS210 (Sommersemester 2026)
Philipps-Universität Marburg
version 2026-06-17 23:01 H

Outline

9 Suchbäume

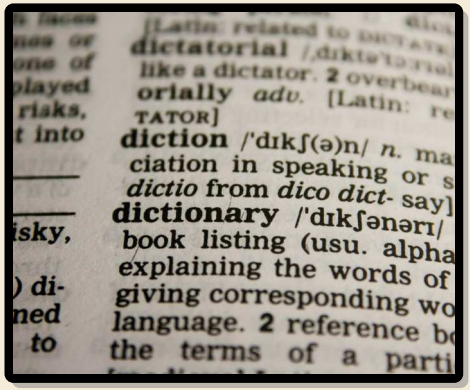
9.1 Symbol Tables (Wörterbücher)

Symbol Table ADT

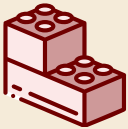
Java: `java.util.Map<K,V>`

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

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 Schlüsseln (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:

- `min()`, `max()`
  Kleinster bzw. größter Schlüssel in ST
- `floor`(x), $\lfloor x \rfloor = \mathbb{Z}.$`floor`$(x)$
  Größter Schüssel k in ST mit $k \leq x$.
- `ceiling`(x)
  Kleinster Schlüssel k in ST mit $k \geq x$.
- `rank`(x)
  Anzahl an Schlüsseln k in ST $k < x$.
- `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()`.

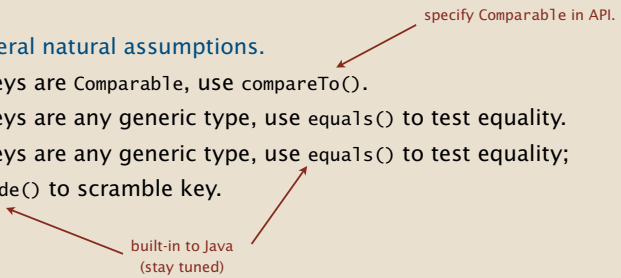
```
public void delete(Key key)
{  put(key, null);  }
```

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.



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^{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));
}
```

keys	S	E	A	R	C	H	E	X	A	M	P	L	E
values	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

% java FrequencyCounter 8 < tale.txt
business 122

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

tiny example
(60 words, 20 distinct)

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

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

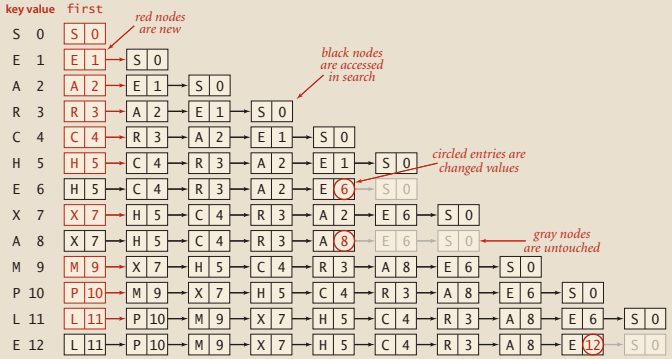
9.3 Elementare Ideen für Symbol Tables

Sequential search in a linked list

Data structure. Maintain an (unordered) linked list of key-value pairs.

Search. Scan through all keys until find a match.

Insert. Scan through all keys until find a match; if no match add to front.



Trace of linked-list ST implementation for standard indexing client

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	`equals()`

Challenge. Efficient implementations of both search and insert.

Lineare Listen als ST

Unsortierte (verkettete) Liste:

- 👍 Effizientes `put`
- 👎 $\Theta(n)$ Zeit für `get`
- $\rightsquigarrow$ Zu langsam um nützlich zu sein

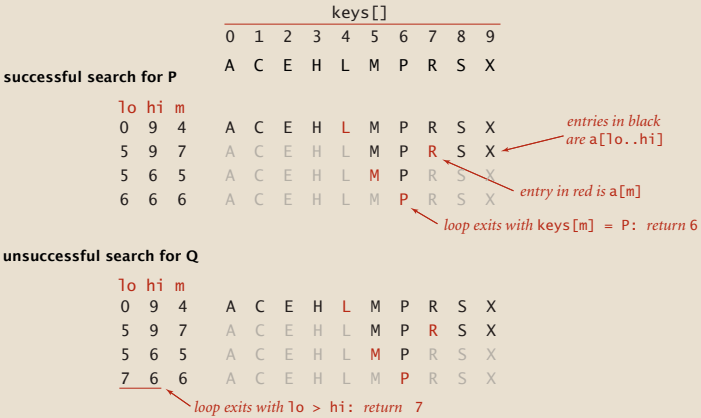
Sortierte *verkettete* Liste:

- 👎 $\Theta(n)$ Zeit für `put`
- 👎 $\Theta(n)$ Zeit für `get` (kein Random Access!)
- $\rightsquigarrow$ Zu langsam um nützlich zu sein
- $\rightsquigarrow$ *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 ?



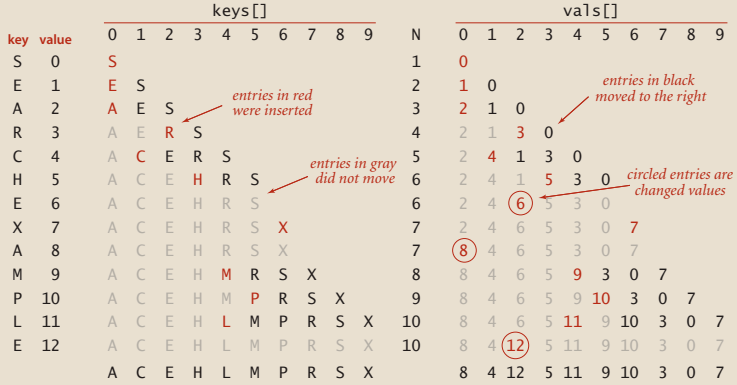
Binary search: Java implementation

```
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.



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	`equals()`
binary search (ordered array)	$\log N$	N	$\log N$	$N / 2$	`compareTo()`

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

⇝ *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 – min-oriented version

Erlauben hier nur Zugriff auf größtes Element

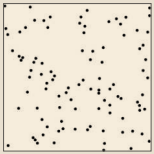
(Max-oriented) Priority Queue (MaxPQ):

- `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()`
  Element mit größter priority. (Verändert PQ nicht.)
- `delMax()`
  Entfernt Element mit größter Priorität.
- `changeKey(`x`,`p'`)`
  Aktualisiere die Priorität von x zu p'.
  Manchmal nur erlaubt für *höhere* 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]



Priority queue API

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

Key must be Comparable (bounded type parameter)

```
public class MaxPQ<Key extends Comparable<Key>>
```

	`MaxPQ()`	*create an empty priority queue*
	`MaxPQ(Key[] a)`	*create a priority queue with given keys*
`void`	`insert(Key v)`	*insert a key into the priority queue*
`Key`	`delMax()`	*return and remove a largest key*
`boolean`	`isEmpty()`	*is the priority queue empty?*
`Key`	`max()`	*return a largest key*
`int`	`size()`	*number of entries in the priority queue*

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`

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

Tatsächich: **Ja!** *Binäre Heaps.*

Array Sichtweise

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

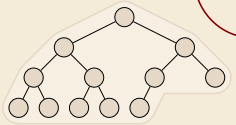
store nodes
in level order
in $A[1..n]$

Baum Sichtweise

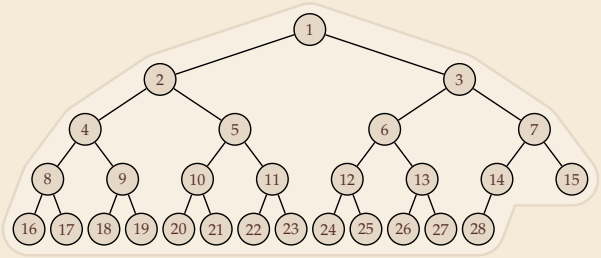
alle Level voll
letztes Level
„linksbündig"

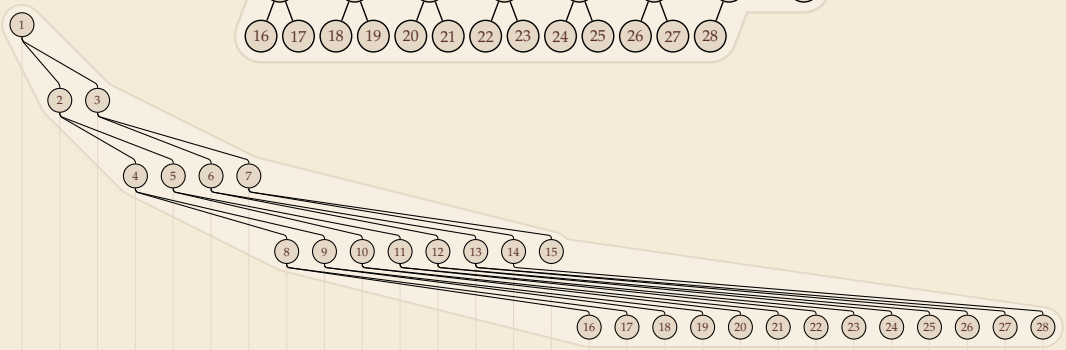
Heap = Baum mit
(i) vollständiger binärer Baum
(ii) heap-geordnet

Vater $\geq$ Kinder



Implizite Vollständige Binärbäume

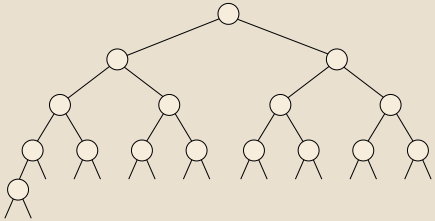




Complete binary tree

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

Binary heap: representation

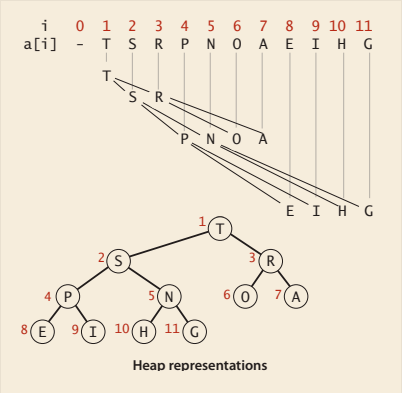
Binary heap. Array representation of a heap-ordered complete binary tree.

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!



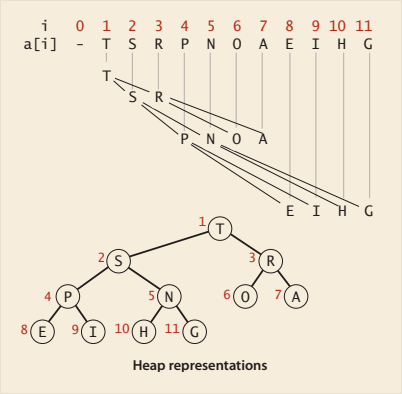
Heap representations

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`.



Heap representations

9.5 Operationen in binären Heaps

Binary heap: promotion

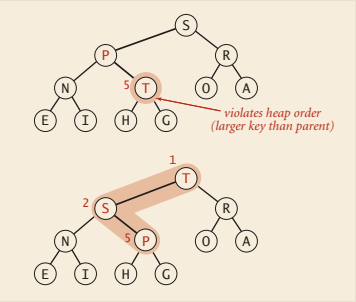
Scenario. A key becomes larger than its parent's key.

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



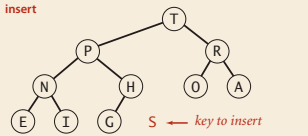
Peter principle. Node promoted to level of incompetence.

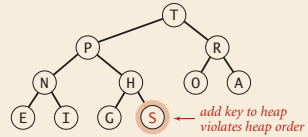
Binary heap: insertion

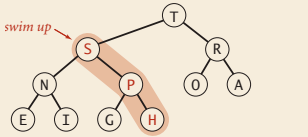
Insert. Add node at end, then swim it up.

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.

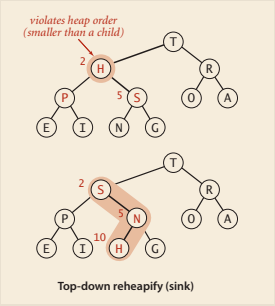
To eliminate the violation:

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.

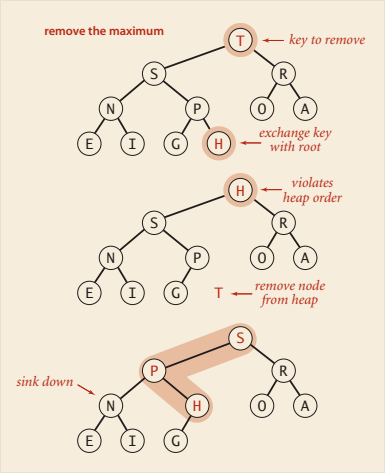
Binary heap: delete the maximum

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;
   return max;
}
```

prevent loitering



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
unordered array	1	n	n
ordered array	n	1	1
binary heap	$\log n$	$\log n$	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.

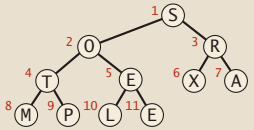
- Assumption: client does not change keys while they're on the PQ.
- Best practice: use immutable keys.

Heapsort

Basic plan for in-place sort.

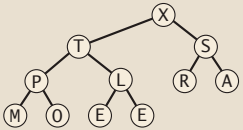
- View input array as a complete binary tree.
- Heap construction: build a max-heap with all n keys.
- Sortdown: repeatedly remove the maximum key.

keys in arbitrary order



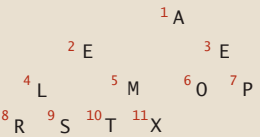
1	2	3	4	5	6	7	8	9	10	11
S	O	R	T	E	X	A	M	P	L	E

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)

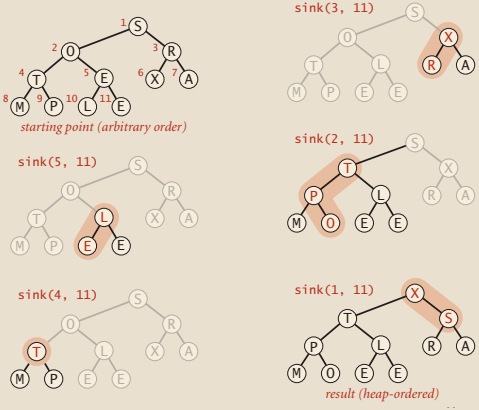


1	2	3	4	5	6	7	8	9	10	11
A	E	E	L	M	O	P	R	S	T	X

Heapsort: heap construction

First pass. Build heap using bottom-up method.

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

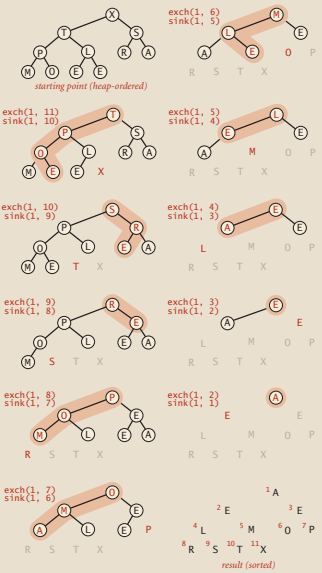


Heapsort: sortdown

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
`construct(`$A[1..n]$`)`	$O(n)$
`max()`	$O(1)$
`insert(`x`,`p`)`	$O(\log n)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

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.

- $\rightsquigarrow$ Erlaube beliebige Form von binären Bäumen

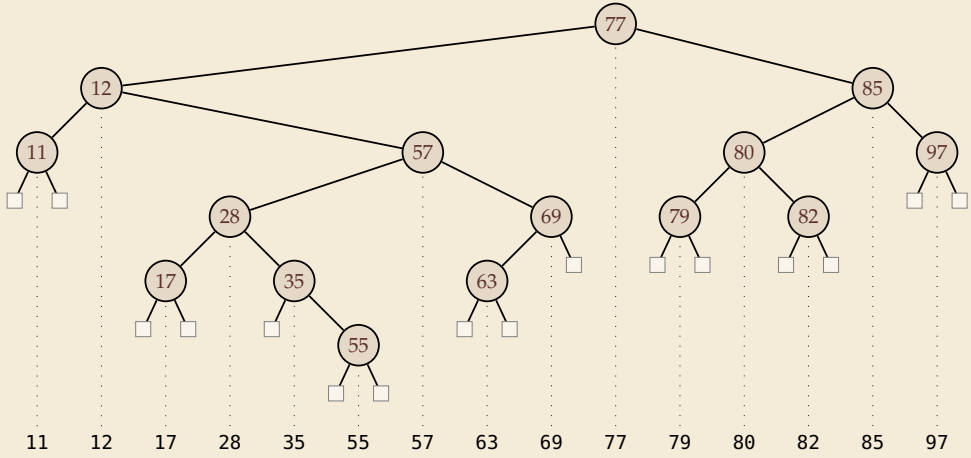
Umgekehrt ist die Heap-Ordnung nicht streng genug.

- 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:

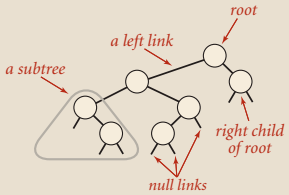
alle Schlüssel im **linken** Teilbaum $\leq$ Schlüssel in **Wurzel** $\leq$ Schlüssel im **rechten** Teilbaum



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

Binary search trees

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

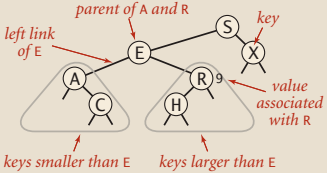


A binary tree is either:

- 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

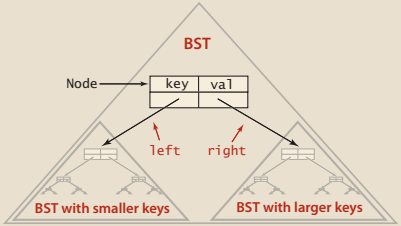
Java definition. A BST is a reference to a root `Node`.

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;
   }
}
```



Binary search tree

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;
}
```

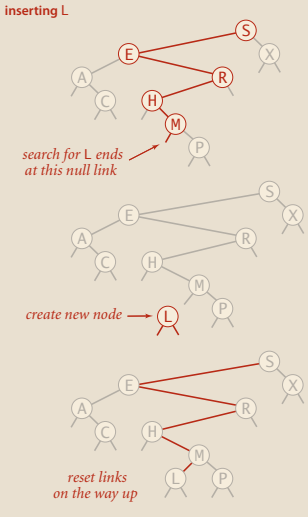
Cost. Number of compares is equal to 1 + depth of node.

BST insert

Put. Associate value with key.

Search for key, then two cases:

- Key in tree ⇒ reset value.
- Key not in tree ⇒ add new node.



Insertion into a BST

BST insert: Java implementation

Put. Associate value with key.

concise, but tricky, recursive code; read carefully!

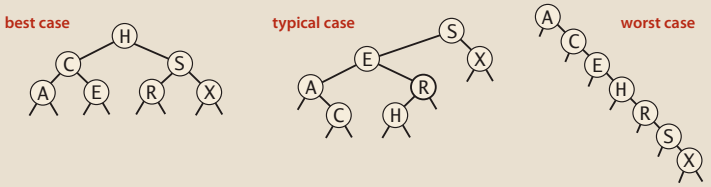
```
public void put(Key key, Value val)
{  root = put(root, key, val);  }

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;
}
```

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

Tree shape

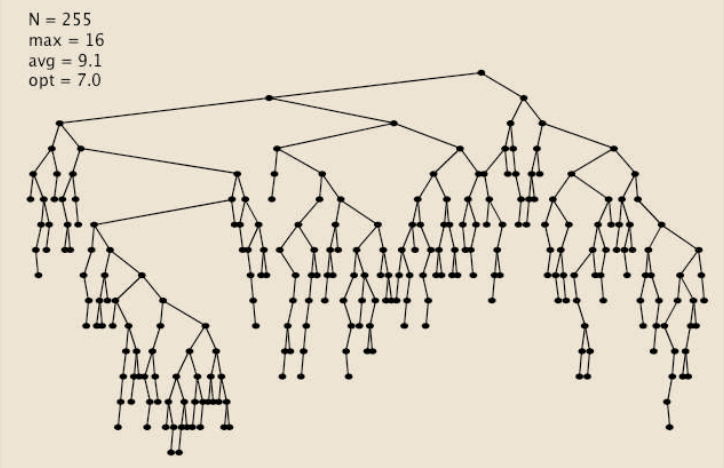
- Many BSTs correspond to same set of keys.
- Number of compares for search/insert is equal to 1 + depth of node.



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.
```

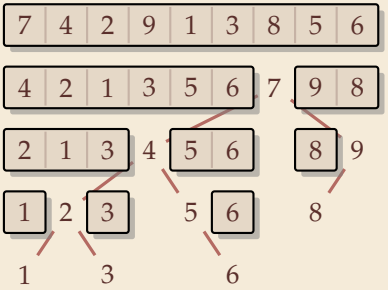
A. It's not a sorting algorithm (if there are duplicate keys)!

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

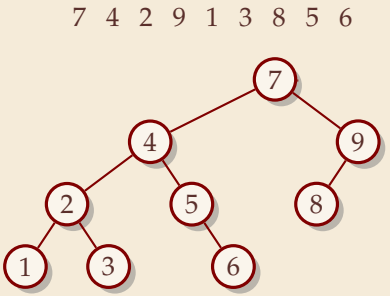
Q. What are its properties?

Quicksort & Binary Search Trees

Quicksort



Binary Search Tree (BST)



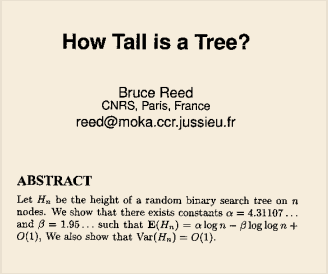
- recursion tree of quicksort = binary search tree from successive insertion
- comparisons in quicksort = comparisons to built BST
- comparisons in quicksort $\approx$ comparisons to search each element in BST

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 $\alpha = 4.31107\ldots$ and $\beta = 1.95\ldots$ such that $\mathrm{E}(H_n) = \alpha \log n - \beta \log \log n + O(1)$, We also show that $\mathrm{Var}(H_n) = O(1)$.

But... Worst-case height is N.

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

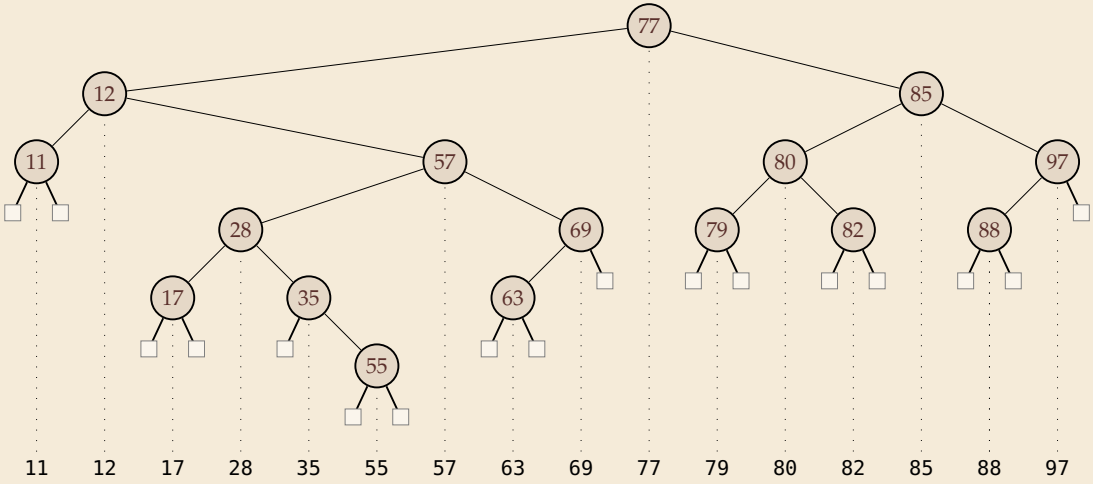
ST implementations: summary

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

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

BST delete

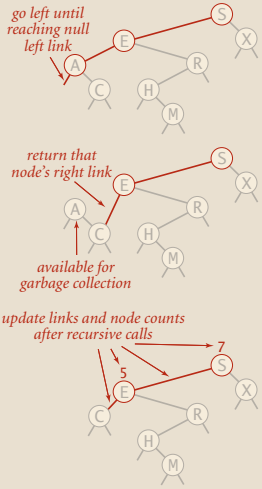
- Easy case: remove leaf, e. g., 11 ⇝ replace by `null`
- Medium case: remove unary, e. g., 69 ⇝ replace by unique child
- Hard case: remove binary, e. g., 85 ⇝ swap with predecessor, recurse



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);
   else if (cmp > 0) x.right = delete(x.right, key);
   else {
      if (x.right == null) return x.left;
      if (x.left  == null) return x.right;

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.count = size(x.left) + size(x.right) + 1;
   return x;
}
```

search for key

no right child

no left child

replace with successor

update subtree counts

9.7 Augmented BSTs

Rank-basierte Operationen

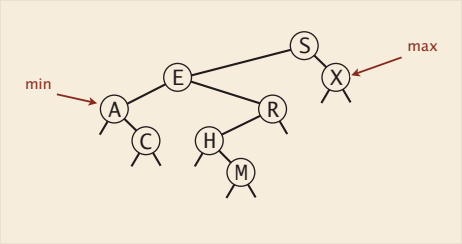
Bisher: `insert`, `get/put`, `delete`

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

Minimum and maximum

Minimum. Smallest key in table.

Maximum. Largest key in table.

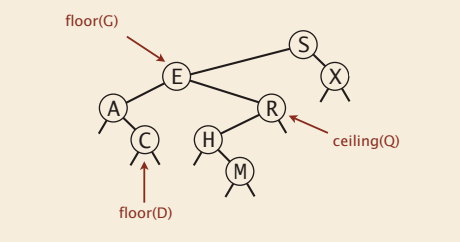


Q. How to find the min / max?

Floor and ceiling

Floor. Largest key ≤ a given key.

Ceiling. Smallest key ≥ a given key.



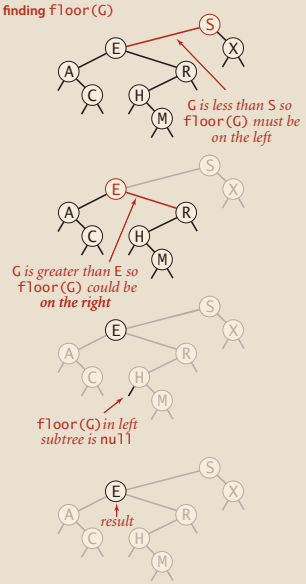
Q. How to find the floor / ceiling?

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.



Computing the floor

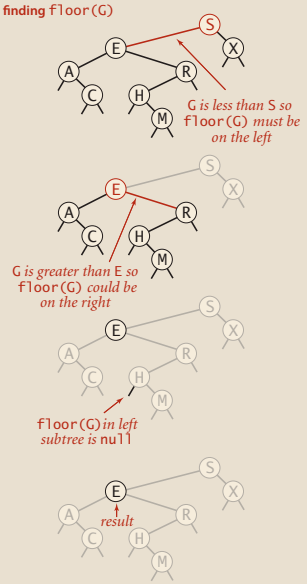
```
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}
private Node floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);

   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;

}
```



Rank-basierte Operationen

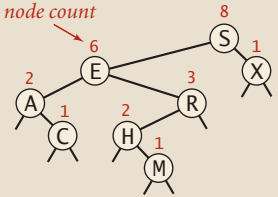
Soweit, so gut.

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

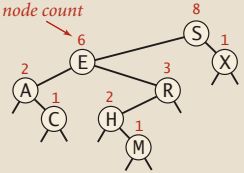
```
private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val, 1);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0) x.val = val;
   x.count = 1 + size(x.left)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

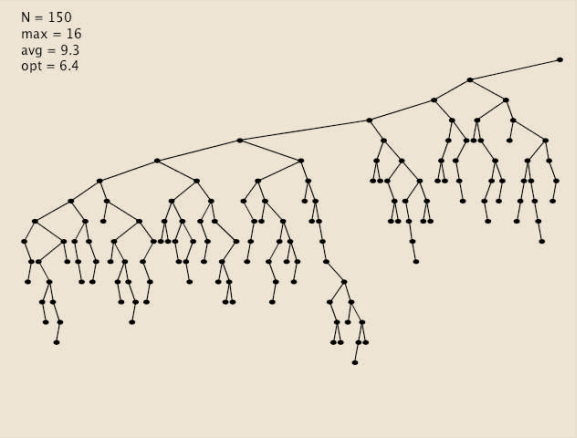
Augmentierte BSTs – Zusammenfassung

Mit h der Höhe des BST erhalten wir

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

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	$\frac{1}{2} N$		`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()`

other operations also become √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		
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()`
goal	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	$\log N$	✔	`compareTo()`

Challenge. Guarantee performance.

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

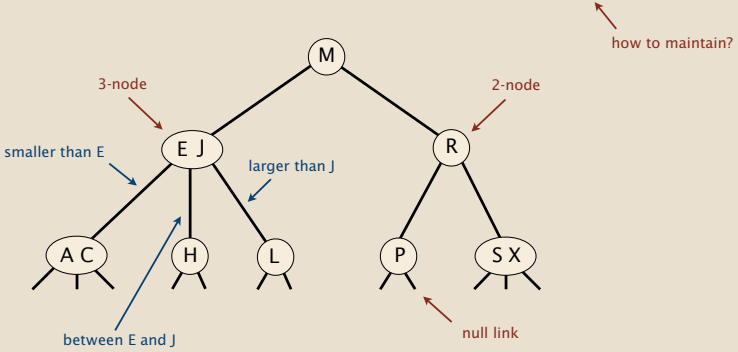
2-3 tree

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.

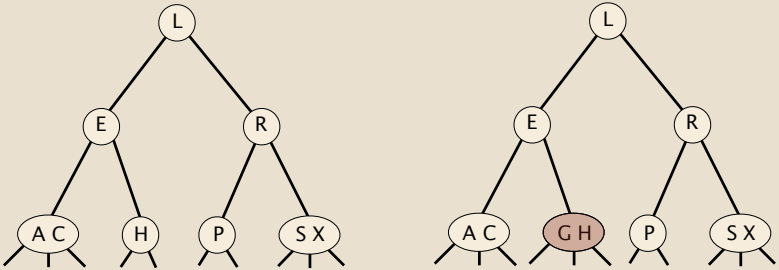


Insertion into a 2-3 tree

Insertion into a 2-node at bottom.

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

insert G

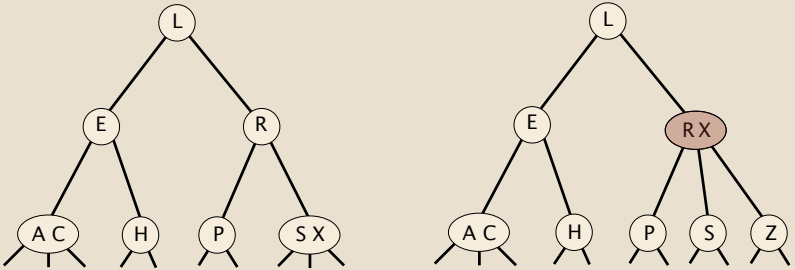


Insertion into a 2-3 tree

Insertion into a 3-node at bottom.

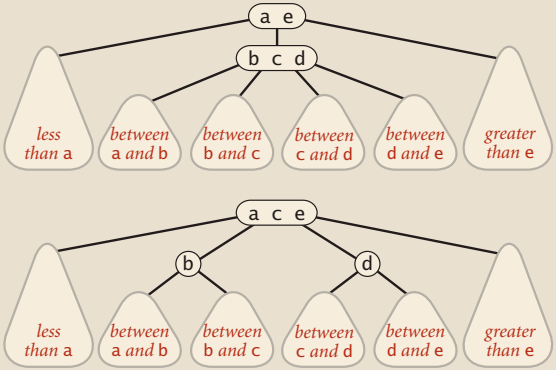
- Add new key to 3-node to create temporary 4-node.
- Move middle key in 4-node into parent.
- Repeat up the tree, as necessary.
- If you reach the root and it's a 4-node, split it into three 2-nodes.

insert Z



Local transformations in a 2-3 tree

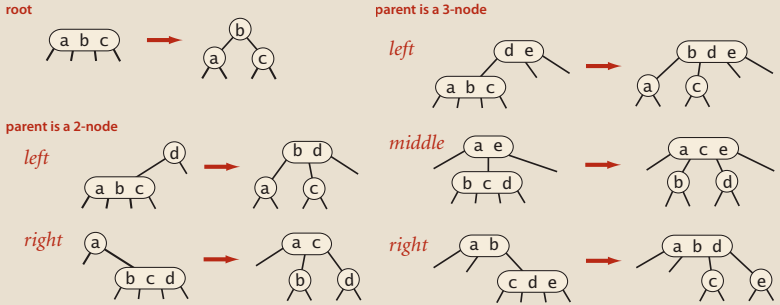
Splitting a 4-node is a local transformation: constant number of operations.



Global properties in a 2-3 tree

Invariants. Maintains symmetric order and perfect balance.

Pf. Each transformation maintains symmetric order and perfect balance.



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		
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()`

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.

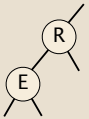
9.9 Balancierte Suchbäume: Red-Black-Trees

How to implement 2-3 trees with binary trees?

Challenge. How to represent a 3 node?

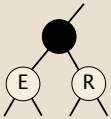
Approach 1: regular BST.

- No way to tell a 3-node from a 2-node.
- Cannot map from BST back to 2-3 tree.



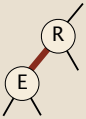
Approach 2: regular BST with "glue" nodes.

- Wastes space, wasted link.
- Code probably messy.



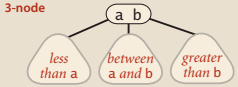
Approach 3: regular BST with red "glue" links.

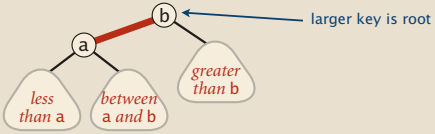
- Widely used in practice.
- Arbitrary restriction: red links lean left.

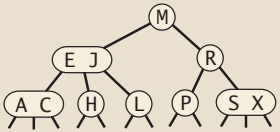


Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

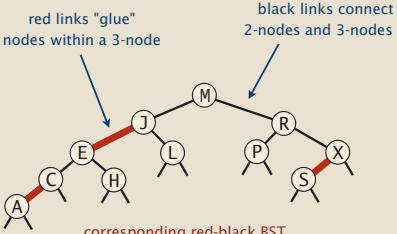
1. Represent 2–3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3–nodes.







2-3 tree



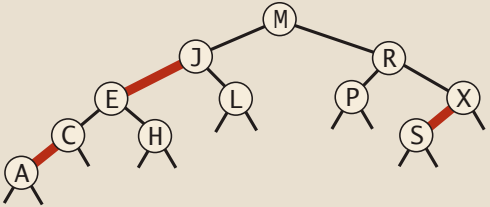
corresponding red-black BST

An equivalent definition

A BST such that:

- No node has two red links connected to it.
- Every path from root to null link has the same number of black links.
- Red links lean left.

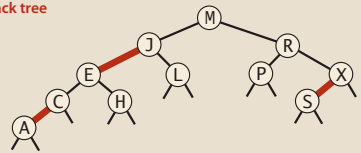
"perfect black balance"



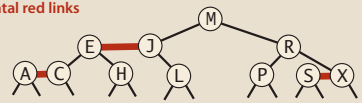
Left-leaning red-black BSTs: 1-1 correspondence with 2-3 trees

Key property. 1–1 correspondence between 2–3 and LLRB.

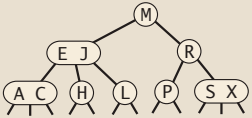
red–black tree



horizontal red links



2-3 tree

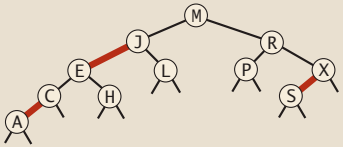


Search implementation for red-black BSTs

Observation. Search is the same as for elementary BST (ignore color).

but runs faster
because of better balance

```
public Val 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;
}
```



Remark. Most other ops (e.g., floor, iteration, selection) are also identical.

Red-black BST representation

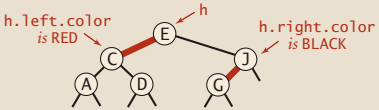
Each node is pointed to by precisely one link (from its parent) ⇒
can encode color of links in nodes.

```
private static final boolean RED   = true;
private static final boolean BLACK = false;

private class Node
{
   Key key;
   Value val;
   Node left, right;
   boolean color;   // color of parent link
}

private boolean isRed(Node x)
{
   if (x == null) return false;
   return x.color == RED;
}
```

null links are black



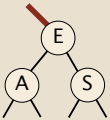
Insertion in a LLRB tree: overview

Basic strategy. Maintain 1-1 correspondence with 2-3 trees.

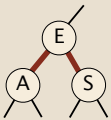
During internal operations, maintain:

- Symmetric order.
- Perfect black balance.

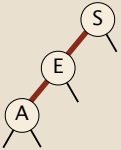
  [but not necessarily color invariants]



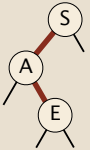
right-leaning red link



two red children (a temporary 4-node)



left-left red (a temporary 4-node)

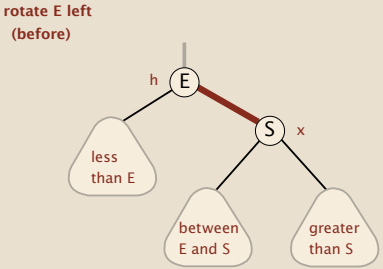


left-right red (a temporary 4-node)

How? Apply elementary red-black BST operations: rotation and color flip.

Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

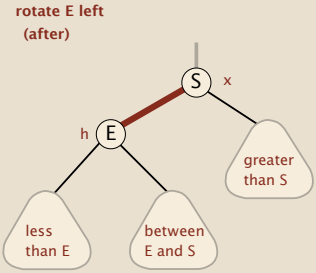



```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

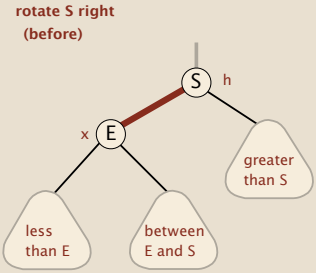



```
private Node rotateLeft(Node h)
{
   assert isRed(h.right);
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

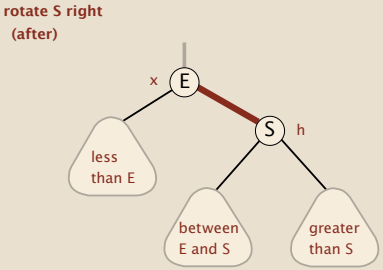



```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

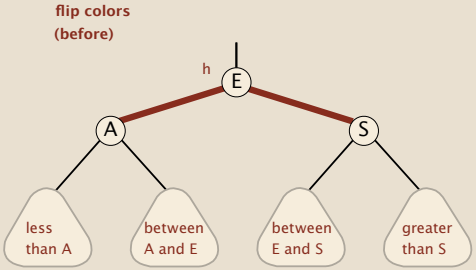



```
private Node rotateRight(Node h)
{
   assert isRed(h.left);
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

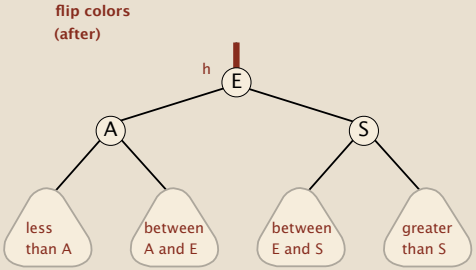



```
private void flipColors(Node h)
{
   assert !isRed(h);
   assert isRed(h.left);
   assert isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

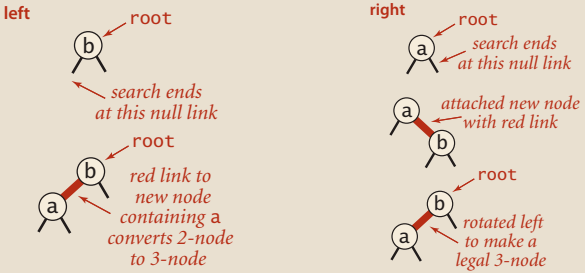



```
private void flipColors(Node h)
{
   assert !isRed(h);
   assert isRed(h.left);
   assert isRed(h.right);
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Insertion in a LLRB tree

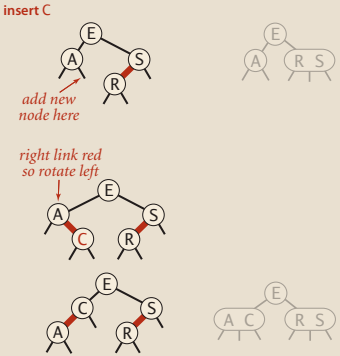
Warmup 1. Insert into a tree with exactly 1 node.



Insertion in a LLRB tree

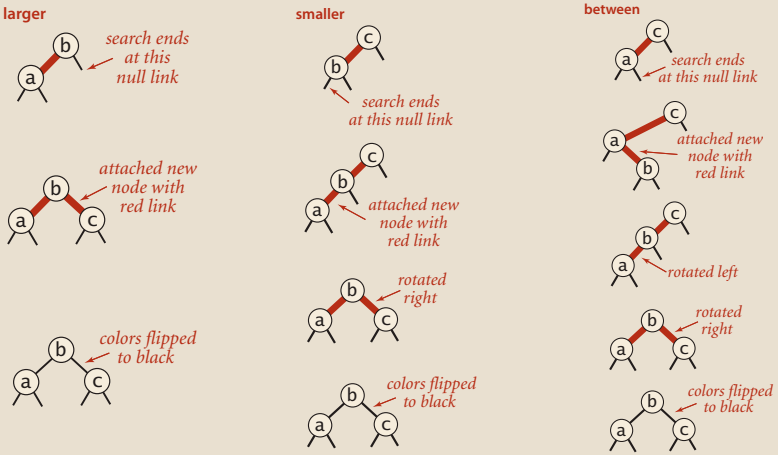
Case 1. Insert into a 2-node at the bottom.

- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- If new red link is a right link, rotate left. ← to fix color invariants



Insertion in a LLRB tree

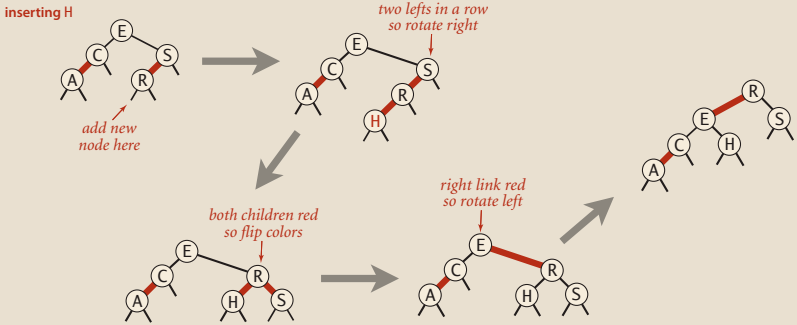
Warmup 2. Insert into a tree with exactly 2 nodes.



Insertion in a LLRB tree

Case 2. Insert into a 3-node at the bottom.

- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed). ← to fix color invariants

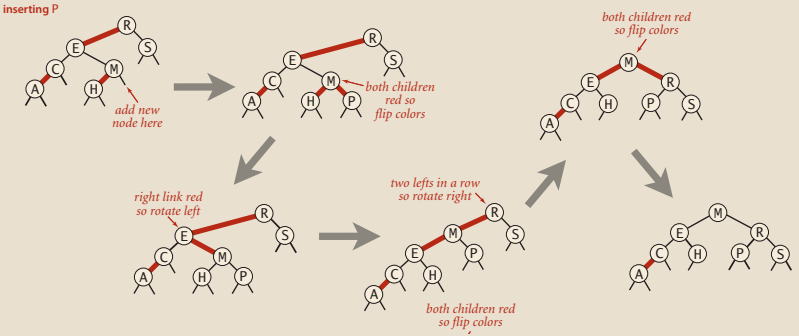


Insertion in a LLRB tree: passing red links up the tree

Case 2. Insert into a 3-node at the bottom.

- Do standard BST insert; color new link red. ← to maintain symmetric order and perfect black balance
- Rotate to balance the 4-node (if needed).
- Flip colors to pass red link up one level.
- Rotate to make lean left (if needed).
- Repeat case 1 or case 2 up the tree (if needed).

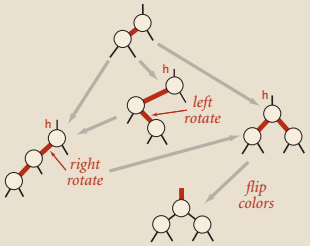
← to fix color invariants



Insertion in a LLRB tree: Java implementation

Same code for all cases.

- Right child red, left child black: rotate left.
- Left child, left-left grandchild red: rotate right.
- Both children red: flip colors.




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

   if (isRed(h.right) && !isRed(h.left))     h = rotateLeft(h);
   if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight(h);
   if (isRed(h.left)  && isRed(h.right))     flipColors(h);

   return h;
}
```

insert at bottom (and color it red)

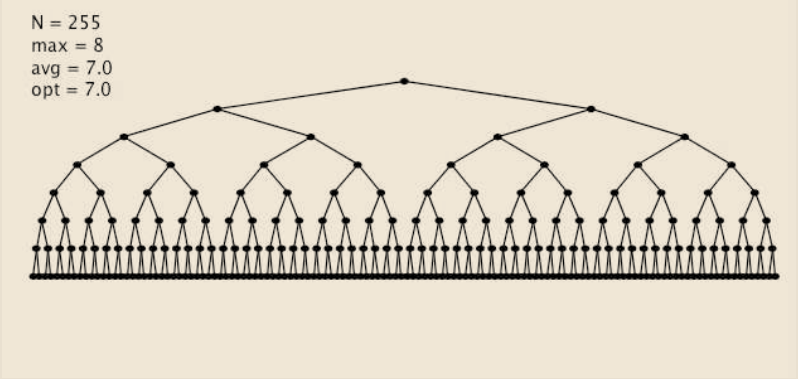
lean left

balance 4-node

split 4-node

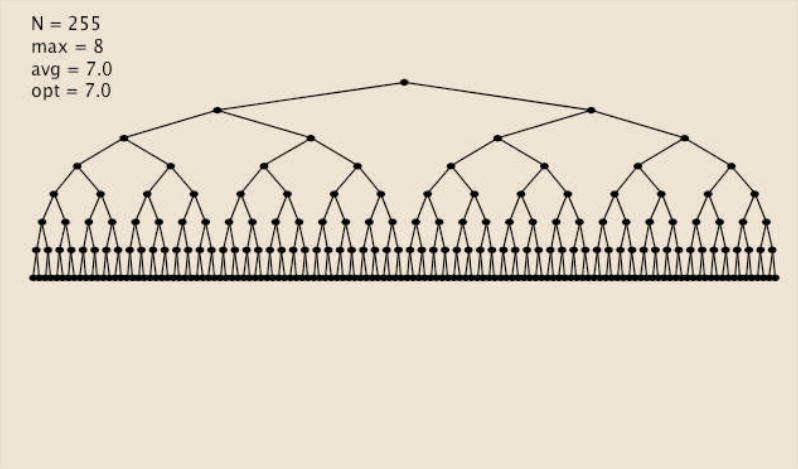
only a few extra lines of code provides near-perfect balance

Insertion in a LLRB tree: visualization



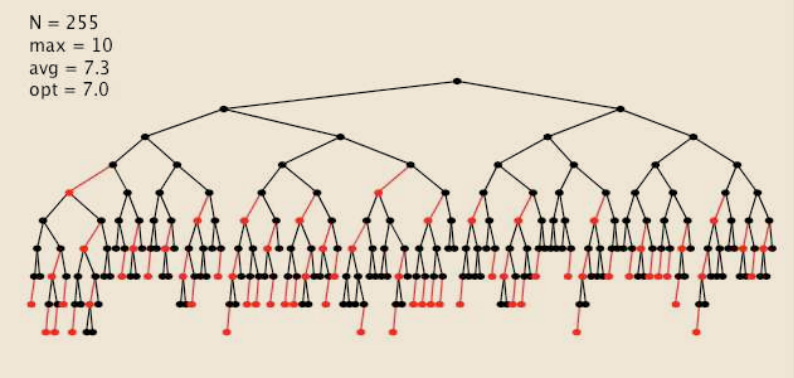
255 insertions in ascending order

Insertion in a LLRB tree: visualization



255 insertions in descending order

Insertion in a LLRB tree: visualization



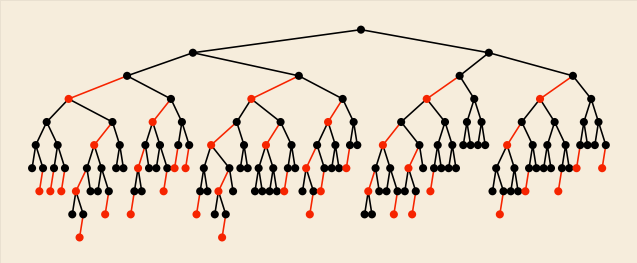
255 random insertions

Balance in LLRB trees

Proposition. Height of tree is $\leq 2 \lg N$ in the worst case.

Pf.

- Every path from root to null link has same number of black links.
- Never two red links in-a-row.



Property. Height of tree is $\sim 1.0 \lg N$ in typical applications.

ST implementations: summary

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()`
red-black BST	$2 \lg N$	$2 \lg N$	$2 \lg N$	$1.0 \lg N$ *	$1.0 \lg N$ *	$1.0 \lg N$ *	✔	`compareTo()`

* exact value of coefficient unknown but extremely close to 1

9.10 Balancierte Suchbäume: Löschen

Löschen

bisher: nur Einfügen

- unten im Baum anfangen, neuen Schlüssel einfügen
- temporäre 4-Knoten erlauben, aber sofort aufspalten in drei 2-Knoten

⇝ kann bis zur Wurzel propagieren und den Baum in der Höhe wachsen lassen

Wollen natürlich auch Löschen können!

- Wie in unbalancierten BST, falls zu löschender Knoten in innerem Knoten:
  - Tausche Schlüssel mit Nachfolger (Minimum in rechten Teilbaum)
  - Entferne dieses Knoten (kann kein linkes Kind haben)
  - ⇝ Im Folgenden beschrieben wir nur wie man einen nicht-binären Knoten löscht
  - ⇝ in 2-3-Baum ist das dann notwendigerweise ein **Blatt**!

⇝ *Es gibt unterschiedliche Ansätz, das Löschen zu realisieren*

Löschen in 2-3-Bäumen – Variante 1

Option 1 besteht darin, eine Fallunterscheidung in den Blättern zu machen

- Entfernen Schlüssel aus Blatt
- Unterscheide Fälle
  - (I) Falls vorher 3-Knoten $\rightsquigarrow$ jetzt 2-Knoten ✓
    - vorher 2-Knoten $\rightsquigarrow$ jetzt leer ϟ, d. h., müssen etwas tun
  - (II) vorher 2-Knoten, aber $\exists$ 3-Knoten Geschwister
    - $\rightsquigarrow$ Rotiere Schlüssel über Elternknoten um gelöschten zu ersetzen
  - (III) alle Geschwister 2-Knoten
    - $\rightsquigarrow$ schiebe Schlüsseln von Elternknoten in Blatt
    - $\rightsquigarrow$ rekursiv: Lösche aus Eltern (propagiere Löschung nach oben)

siehe z. B. `https://kubokovac.eu/gnarley-trees/23tree.html`

Nachteil von Variante 1: schon für 2-3-Bäume viele Fälle . . . für Red-Black-Trees noch mehr 😬

Löschen in 2-3-Bäumen – Variante 2

Variante 2: Konzeptionell sollte Löschen einfach sein: Einfügen rückgängig machen!

- Komplikation: Beim Einfügen fügen wir erst einen Knoten hinzu, dann wir rebalanciert

⇝ `delete(`x`)` muss das umgekehrt machen . . .

Invariante: Wenn wir in Suchpfad absteigen ist nächster Knoten **kein** 2-Knoten.

- dafür erlauben wir temporär auch 4-Knoten zu haben (nur bis Ende dieser Operation)
- Unterscheide Fälle
  - (a) Nächster Knoten ist 3-Knoten ⇝ Invariante erfüllt, gehe zum nächsten Knoten
  - (b) Nächster Knoten ist 2-Knoten ⇝ „drücke“ Schlüssel aus aktuellem Knoten runter
    - Invariante: aktueller Knoten **kein** 2-Knoten ⇝ hat Schlüssel „übrig“
    - Welchen Schlüssel runterdrücken?
      3-Knoten: linken Schlüssel bei mittlerem Kind; 4-Knoten: nie mittleren Schlüssel
    - Schlüssel runterdrücken = zwei Geschwister zu verschmelzen (plus extra Schlüssel)
      einer davon ist 2-Knoten; der andere könnte 3-Knoten sein

    ⇝ kann 5-Knoten erzeugen ⇝ teile direkt wieder auf in 2- und 3-Knoten
    aber mit 3-Knoten auf dem Suchpfad
- im rekursiven Aufstieg: 4-Knoten in drei 2-Knoten aufteilen (Vater immer 2-Knoten)

Löschen in Red-Black-Trees

Left-Leaning Red-Black verwenden Variante 2
Code kompakt, aber trickreich

Sedgewick: *Left-Leaning Red-Black Trees*, Report **2008**

```
1  public void deleteMin() {
2      // both children of root black -> set to red
3      if (!isRed(root.left) && !isRed(root.right))
4          root.color = RED;
5      root = deleteMin(root);
6      if (!isEmpty()) root.color = BLACK;
7  }
8  private Node deleteMin(Node h) {
9      if (h.left == null) return null;
10     if (!isRed(h.left) && !isRed(h.left.left))
11         h = moveRedLeft(h);
12     h.left = deleteMin(h.left);
13     return balance(h);
14 }
15 public void delete(Key key) {
16     if (!contains(key)) return;
17     // both children of root black -> set root to red
18     if (!isRed(root.left) && !isRed(root.right))
19         root.color = RED;
20     root = delete(root, key);
21     if (!isEmpty()) root.color = BLACK;
22 }
```

```
23 private Node delete(Node h, Key key) {
24     if (key.compareTo(h.key) < 0)  {
25         if (!isRed(h.left) && !isRed(h.left.left))
26             h = moveRedLeft(h);
27         h.left = delete(h.left, key);
28     } else {
29         if (isRed(h.left))  h = rotateRight(h);
30         if (key.compareTo(h.key) == 0
31                 && (h.right == null))
32             return null;
33         if (!isRed(h.right) && !isRed(h.right.left))
34             h = moveRedRight(h);
35         if (key.compareTo(h.key) == 0) {
36             Node x = min(h.right);
37             h.key = x.key;
38             h.val = x.val;
39             h.right = deleteMin(h.right);
40         }
41         else h.right = delete(h.right, key);
42     }
43     return balance(h);
44 }
```

Hilfsmethoden

```java
private Node rotateRight(Node h) {
    Node x = h.left; h.left = x.right; x.right = h;
    x.color = h.color; h.color = RED;
    return x;
}
private Node rotateLeft(Node h) {
    Node x = h.right; h.right = x.left; x.left = h;
    x.color = h.color; h.color = RED;
    return x;
}
// flip the colors of a node and its two children
private void flipColors(Node h) {
    h.color = !h.color;
    h.left.color = !h.left.color;
    h.right.color = !h.right.color;
}
// Assuming h is red and both h.left and h.left.left
// are black, make h.left or one of its children red.
private Node moveRedLeft(Node h) {
    flipColors(h);
    if (isRed(h.right.left)) {
        h.right = rotateRight(h.right);
        h = rotateLeft(h);
        flipColors(h);
    }
    return h;
}
// Assuming h is red and both h.right and h.right.left
// are black, make h.right or one of its children red.
private Node moveRedRight(Node h) {
    flipColors(h);
    if (isRed(h.left.left)) {
        h = rotateRight(h);
        flipColors(h);
    }
    return h;
}

// restore red-black tree invariant
private Node balance(Node h) {
    if (isRed(h.right) && !isRed(h.left))
        h = rotateLeft(h);
    if (isRed(h.left) && isRed(h.left.left))
        h = rotateRight(h);
    if (isRed(h.left) && isRed(h.right))
        flipColors(h);
    return h;
}
```

Zusammenfassung

Balancierte Binäre Suchbäume (BBSTs)

Operation	Laufzeit
`construct(`$A[1..n]$`)`	$O(n \log n)$
`put(`k`,`v`)`	$O(\log n)$
`get(`k`)`	$O(\log n)$
`delete(`k`)`	$O(\log n)$
`contains(`k`)`	$O(\log n)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$
`min()` / `max()`	$O(\log n) \rightsquigarrow O(1)$
`floor(`x`)`	$O(\log n)$
`ceiling(`x`)`	$O(\log n)$
`rank(`x`)`	$O(\log n)$
`select(`i`)`	$O(\log n)$

~~**Binäre Heaps**~~ *Strict Fibonacci heaps*

Operation	Laufzeit
`construct(`$A[1..n]$`)`	$O(n)$
`insert(`x`,`p`)`	~~$O(\log n)$~~ $O(1)$
`delMax()`	$O(\log n)$
`changeKey(`x`,`p'`)`	~~$O(\log n)$~~ $O(1)$
`max()`	$O(1)$
`isEmpty()`	$O(1)$
`size()`	$O(1)$

und konstante Faktoren

- bis auf `construct` sind BBSTs genauso gut wie Heaps
- PQ Abstraktion trotzdem hilfreich
- und es gibt bessere PQs!

9.11 B-Bäume

File system model

Page. Contiguous block of data (e.g., a file or 4,096-byte chunk).
Probe. First access to a page (e.g., from disk to memory).

slow

fast

Property. Time required for a probe is much larger than time to access data within a page.

Cost model. Number of probes.

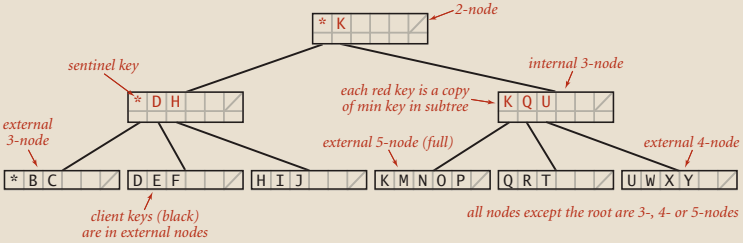
Goal. Access data using minimum number of probes.

B-trees (Bayer-McCreight, 1972)

B-tree. Generalize 2-3 trees by allowing up to $M - 1$ key-link pairs per node.

choose M as large as possible so that M links fit in a page, e.g., M = 1024

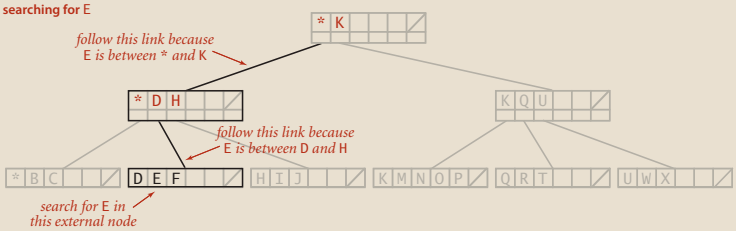
- At least 2 key-link pairs at root.
- At least $M / 2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.



Anatomy of a B-tree set (M = 6)

Searching in a B-tree

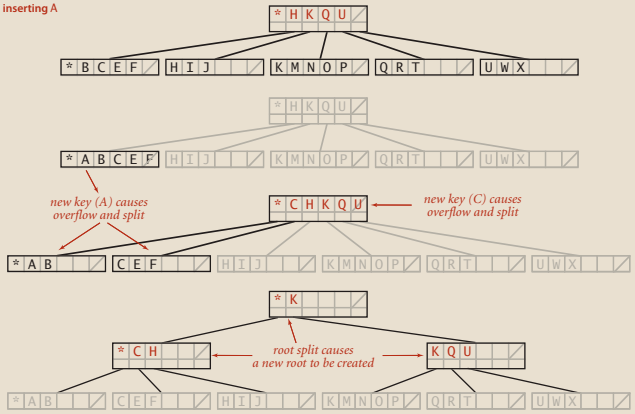
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Searching in a B-tree set (M = 6)

Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with M key-link pairs on the way up the tree.



Inserting a new key into a B-tree set

Balance in B-tree

Proposition. A search or an insertion in a B-tree of order M with N keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes.

Pf. All internal nodes (besides root) have between $M / 2$ and $M - 1$ links.

In practice. Number of probes is at most 4. ← M = 1024; N = 62 billion
$\log_{M/2} N \leq 4$

Optimization. Always keep root page in memory.

Balanced trees in the wild

Red-black trees are widely used as system symbol tables.

- Java: `java.util.TreeMap`, `java.util.TreeSet`.
- C++ STL: map, multimap, multiset.
- Linux kernel: completely fair scheduler, `linux/rbtree.h`.
- Emacs: conservative stack scanning.

B-tree variants. B+ tree, B*tree, B# tree, ...

B-trees (and variants) are widely used for file systems and databases.

- Windows: NTFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.