

Binäre Suchbäume

Mengen, Funktionalität, Binäre Suchbäume, Heaps, Treaps

Mengen

- **Ziel:** Aufrechterhalten einer Menge (hier: ganzer Zahlen) unter folgenden Operationen:
 - Einfügen eines Elements
 - Löschen eines Elements
 - Suchen eines Elements
- **Anforderung:** effizient auch bei grossen Mengen!

Mengen: Anwendungen (I)

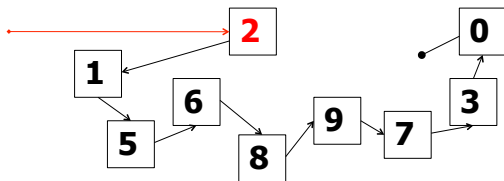
- Telefonbuch (Menge von Namen mit zugehörigen Telefonnummern)
 - **Einfügen:** Neue Telefonanschlüsse
 - **Löschen:** Aufgegebene Telefonanschlüsse
 - **Suchen:** Telefonnummer einer Person

Mengen: Anwendungen (II)

- Nutzerverwaltung (Menge von Nutzern mit Passwörtern und/oder weiteren Informationen)
 - **Einfügen:** Neue Nutzer
 - **Löschen:** Ex-Nutzer
 - **Suchen:** Einloggen
- **Effizienz:** Keine Wartezeit beim Einloggen, auch bei Millionen von Nutzern.

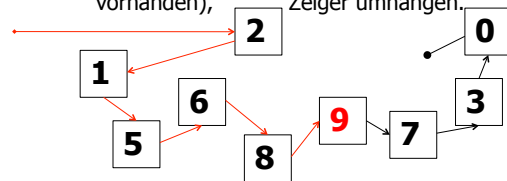
Mengen: Lösung mit Listen

- Menge wird als Liste gespeichert.
 - **Einfügen:** zum Beispiel vorne anfügen (effizient)



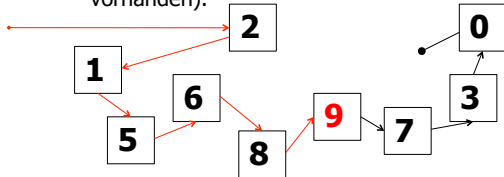
Mengen: Lösung mit Listen

- Menge wird als Liste gespeichert.
 - **Löschen:** Durchlaufen der Liste bis zum Element (oder bis zum Ende, falls nicht vorhanden), Zeiger umhängen.



Mengen: Lösung mit Listen

- Menge wird als Liste gespeichert.
 - **Suchen:** Durchlaufen der Liste bis zum Element (oder bis zum Ende, falls nicht vorhanden).



Mengen: Lösung mit Listen

- Falls die Menge n Elemente enthält, so ist der Aufwand fürs Suchen eines Elements im schlimmsten Fall proportional zu n .
- Facebook: Zeit zum Einloggen wäre proportional zur Anzahl der Nutzer.



Mengen: Lösung mit Listen

- Beispiel: Suche nach den letzten 100 Elementen in einer Liste von 10,000,000 Elementen

```
List l;
for (int i=0; i<10000000; ++i)
  l.push_front(i);

// ...and search for the 100 first ones
for (int i=0; i<100; ++i)
  l.find(i);
```

Facebook hat
1,000,000,000
Nutzer, eine
Suche würde
6.5 Sekunden
dauern!

6.5 Sekunden

Mengen: Lösung mit Listen

- Beispiel: Suche nach den letzten 100 Elementen in einer Liste von 10,000,000 Elementen

- Finden eines Elements:

```
const Node* List::find (int key)
{
  for (const Node* p = head_; p != 0; p = p->get_next())
    if (p->get_key() == key) return p;
  return 0;
}
```

Mengen: Lösung mit Binären Suchbäumen.

- Falls die Menge n Elemente enthält, so ist der Aufwand fürs Suchen eines Elements im schlimmsten Fall proportional zu $\log_2 n$.
- Facebook: Zeit zum Einloggen ist proportional zum **Logarithmus** der Anzahl der Nutzer.

$\log_2(1,000,000,000) \approx 30$.

Mengen: Lösung mit Binären Suchbäumen.

- Falls die Menge n Elemente enthält, so ist der Aufwand fürs Suchen eines Elements im schlimmsten Fall proportional zu $\log_2 n$.
- Auch Einfügen und Löschen geht in Zeit proportional to $\log_2 n$.

Binärbäume, Definition und Terminologie

- Ein binärer Baum ist entweder leer,...
- oder er besteht aus einem **Knoten** (Kreis)
 - mit einem **Schlüssel** (hier: ganze Zahl),...
 - und einem **linken** und **rechten Teilbaum**, die jeweils Binärbäume sind.

Binäre Suchbäume

- Ein Binärbaum heisst **binärer Suchbaum**, wenn er leer ist, oder wenn folgende Bedingungen gelten:
 - Sowohl der linke als auch der rechte Teilbaum sind binäre Suchbäume.
 - Der Schlüssel der Wurzel ist
 - $>$ alle Schlüssel im linken Teilbaum, und
 - \leq alle Schlüssel im rechten Teilbaum.

Binäre Suchbäume: Beispiel

Binäre Suchbäume: Suchen nach einem Schlüssel

5 ?

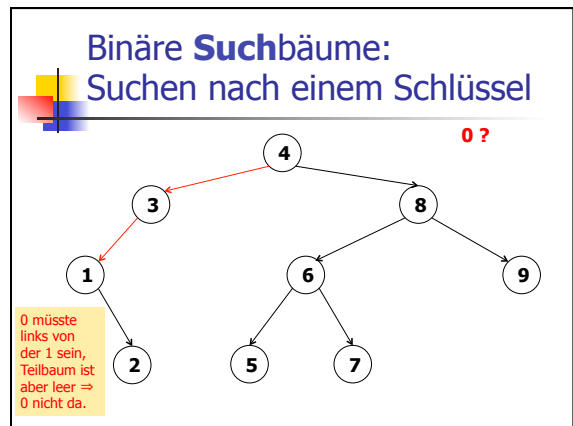
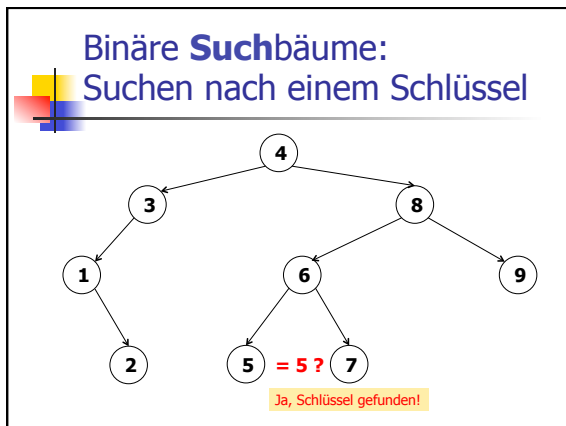
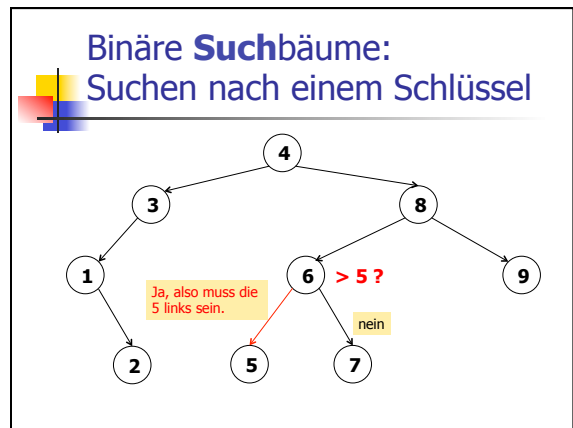
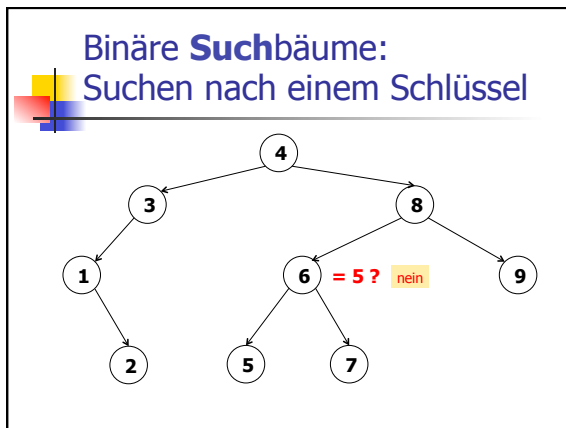
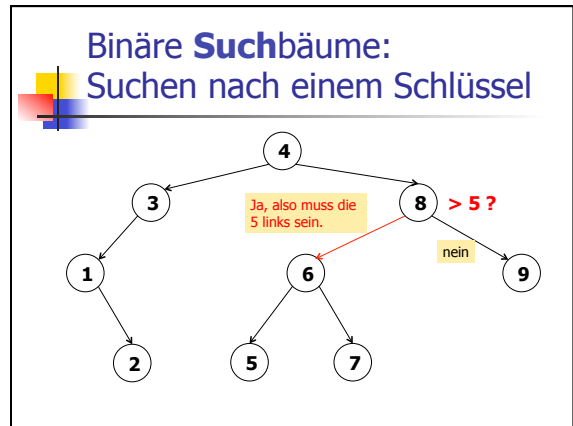
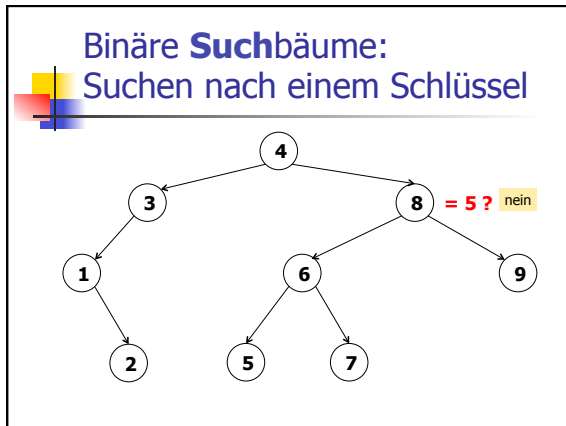
Binäre Suchbäume: Suchen nach einem Schlüssel

4 = 5 ? nein

Binäre Suchbäume: Suchen nach einem Schlüssel

4 > 5 ? Nein, also muss die 5 rechts sein.

ja



Mengen: Lösung mit binären Suchbäumen

- Speichere die Elemente der Menge als Schlüssel in einem binären Suchbaum!
- Suchzeit ist proportional zur **Höhe** des Baums (Länge des längsten Pfades von der Wurzel bis zu einem Blatt).

Mengen: Lösung mit binären Suchbäumen

Mengen: Lösung mit binären Suchbäumen

- Speichere die Elemente der Menge als Schlüssel in einem binären Suchbaum!
- Suchzeit ist proportional zur **Höhe** des Baums (Länge des längsten Pfades von der Wurzel bis zu einem Blatt).
- Einfügen und Löschen von Elementen?

Binäre Suchbäume: Einfügen

- Neues Element suchen und als Blatt an der passenden Stelle anhängen!

Binäre Suchbäume: Einfügen

Binäre Suchbäume: Einfügen

Binäre Suchbäume: Einfügen

- Neues Element suchen und als Blatt an der passenden Stelle anhängen!
- Auch die Einfügezeit ist proportional zur Höhe des Baums.

Binäre Suchbäume: Löschen

- Element suchen, seinen Knoten durch **Rotationen** zu einem Blatt machen, Blatt löschen!
- Rotation: lokale Operation, welche die Positionen von Knoten verändert, **nicht** jedoch die Suchbaumeigenschaften.

Rotation eines Teilbaums

Rechtsrotation

Linksrotation

$A < t \leq B < s \leq C$ = $A < t \leq B < s \leq C$

Binäre Suchbäume: Löschen

8 löschen

Binäre Suchbäume: Löschen

8 löschen

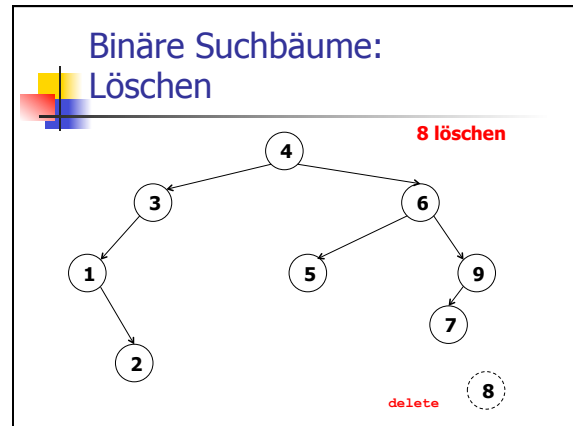
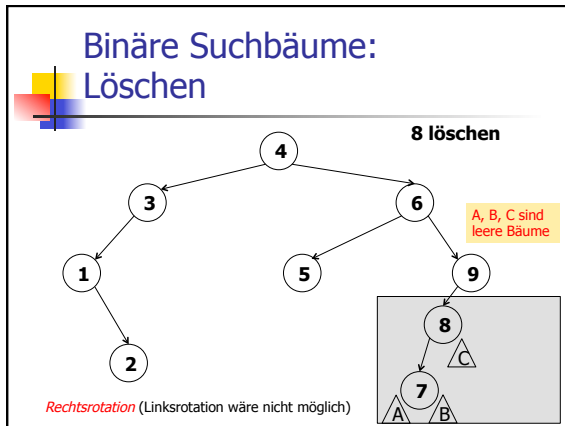
Rechtsrotation (Linksrotation wäre auch möglich)

Binäre Suchbäume: Löschen

8 löschen

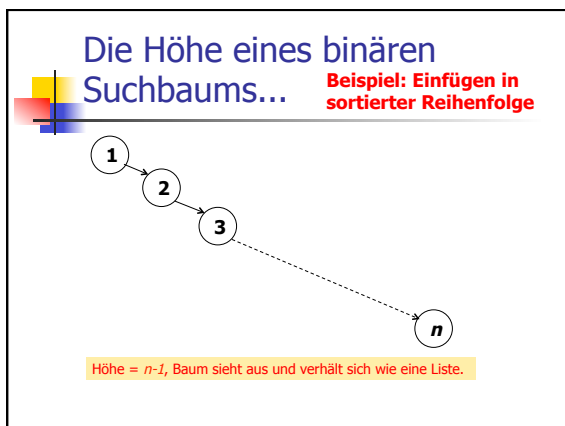
B und C sind leere Bäume

Linksrotation (Rechtsrotation wäre auch möglich)



- ### Binäre Suchbäume: Löschen
- Element suchen, seinen Knoten durch **Rotationen** zu einem Blatt machen, Blatt löschen!
 - Rotation: lokale Operation, welche die Positionen von Knoten verändert, **nicht** jedoch die Suchbaumeigenschaften.
 - Auch die Löszeit ist proportional zur Höhe des Baums (nicht mehr ganz so offensichtlich)

- ### Die Höhe eines binären Suchbaums...
- bestimmt die Laufzeit des
 - Suchens,
 - Einfügens,
 - Löschens.
 - kann aber bereits bei einer ungünstigen Einfügereihenfolge sehr gross werden!
 - Laufzeit im schlimmsten Fall nicht besser als bei einer Liste!



- ### Abhilfe: Balancierung
- Beim Einfügen und Löschen stets darauf achten, dass der Baum „balanciert“ ist, (d.h. keine grossen Höhenunterschiede zwischen linken und rechten Teilbäumen)
 - Verschiedene Möglichkeiten (mehr oder weniger kompliziert):
 - AVL-Bäume, Rot-Schwarz-Bäume,...
 - Hier: **Treaps** (randomisierte Suchbäume)

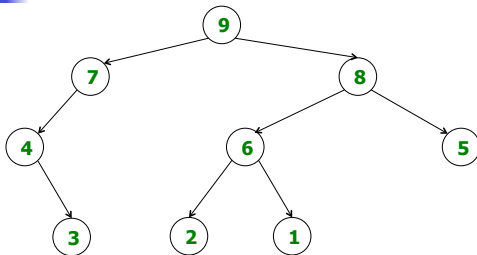
Treap = Tree + Heap

- Ein Binärbaum heisst *Heap*, wenn er leer ist, oder wenn folgende Bedingungen gelten:
 - Sowohl der linke als auch der rechte Teilbaum sind *Heaps*.
 - Der Schlüssel der Wurzel ist das Maximum aller Schlüssel.
 - das Maximum aller Schlüssel.

Heap: Schlüssel heissen Prioritäten

- Ein Binärbaum heisst *Heap*, wenn er leer ist, oder wenn folgende Bedingungen gelten:
 - Sowohl der linke als auch der rechte Teilbaum sind *Heaps*.
 - Die *Priorität* der Wurzel ist das Maximum aller *Prioritäten*.
 - das Maximum aller *Prioritäten*.

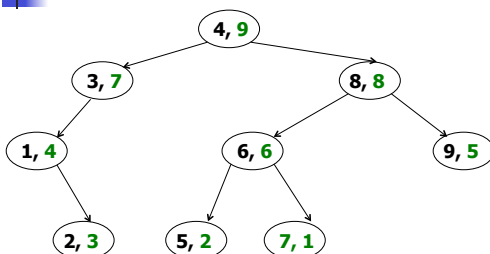
Heap: Beispiel



Treaps

- Ein Binärbaum heisst *Treap*, wenn jeder Knoten einen Schlüssel *und* eine Priorität hat, so dass folgende Eigenschaften gelten:
 - Der Baum ist ein binärer Suchbaum bzgl. der Schlüssel.
 - Der Baum ist ein *Heap* bzgl. der Prioritäten.

Treap: Beispiel

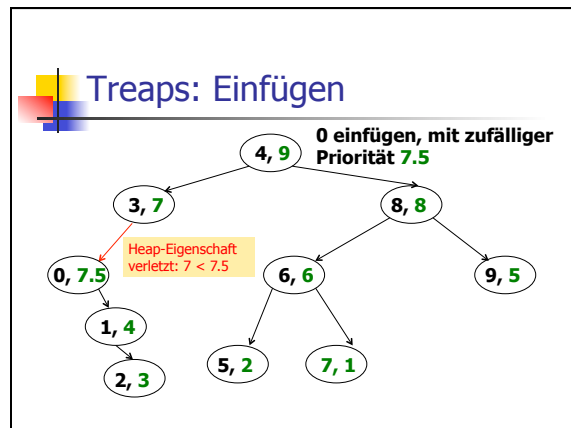
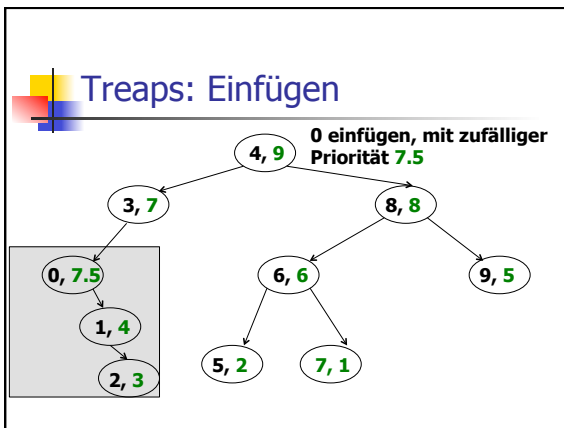
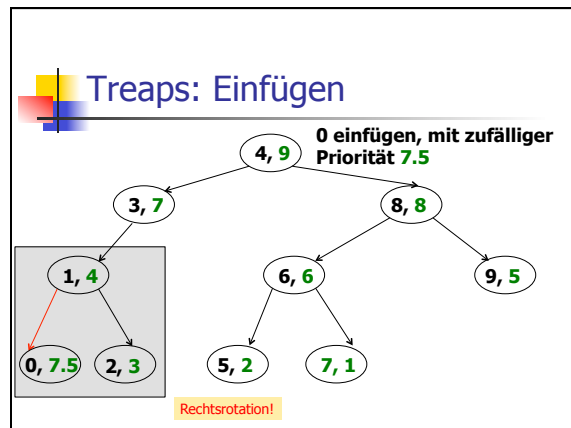
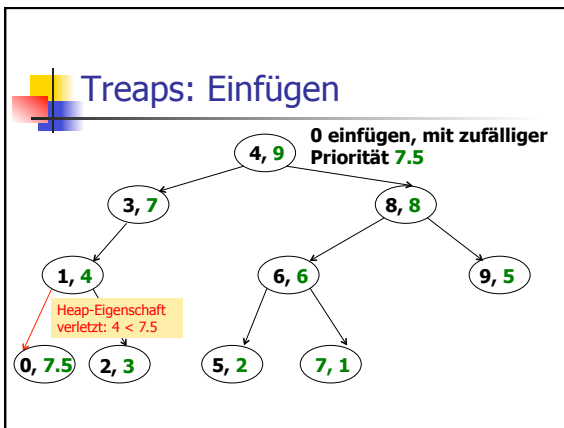
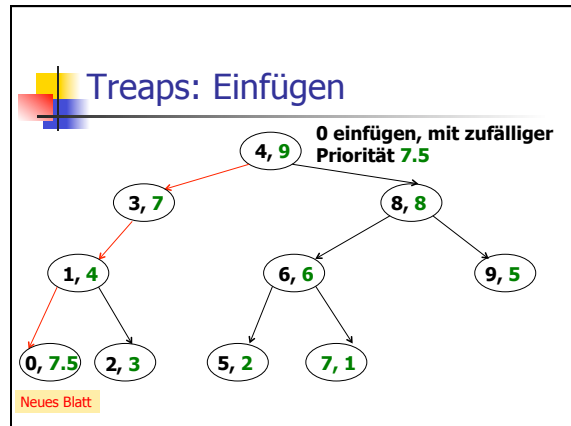


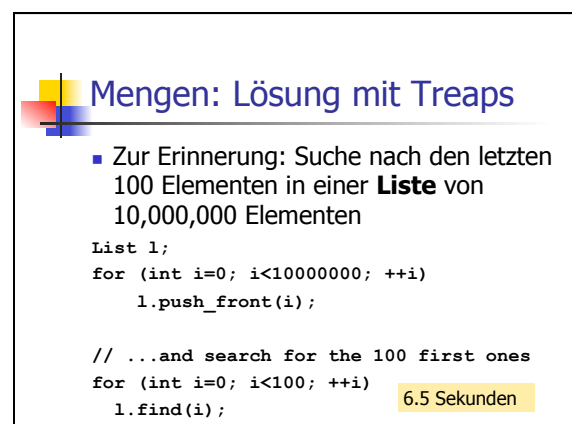
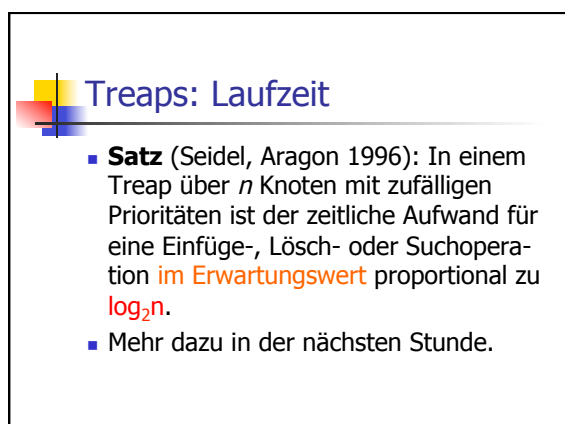
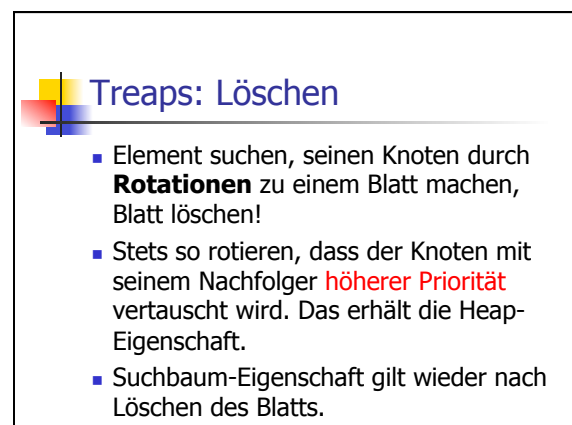
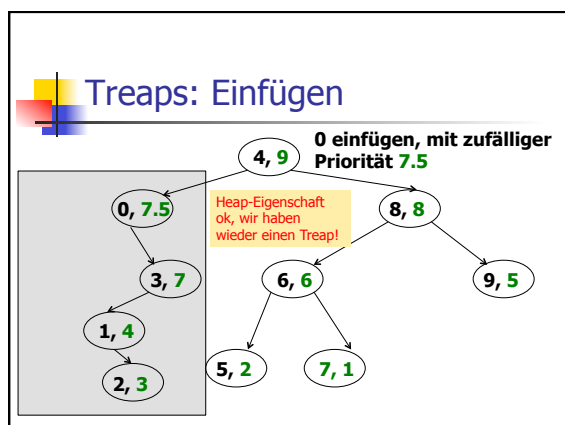
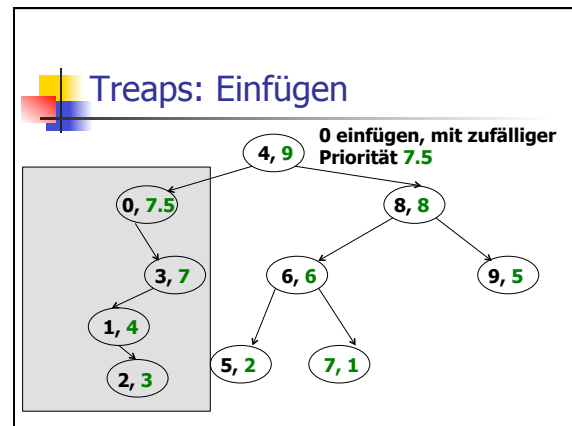
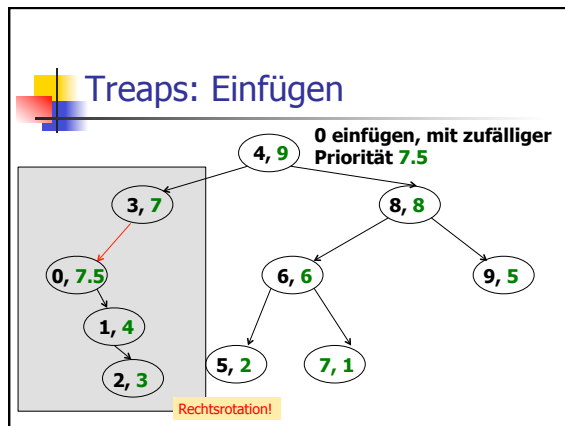
Treaps: Eindeutigkeit

- Satz:** Für n Knoten mit verschiedenen Schlüssel *und* Prioritäten gibt es genau einen *Treap* mit diesen Knoten.
- Beweis:** Die Wurzel ist der Knoten mit der höchsten Priorität; im linken Teilbaum sind alle Knoten mit kleineren und im rechten Teilbaum alle Knoten mit grösseren Schlüssel. Mit Induktion sind die Teilbäume auch eindeutig.

Treaps mit zufälligen Prioritäten

- Wir benutzen einen Treap als binären Suchbaum.
- Beim Einfügen eines neuen Schlüssels wird seine Priorität zufällig gewählt.
- Wie bei normalen binären Suchbäumen: neuer Knoten wird neues Blatt.
- Zusätzlich muss die Treap-Eigenschaft wieder hergestellt werden!**





Mengen: Lösung mit Treaps

- Neu: Suche nach den letzten 100 Elementen in einem **Treap** von 10,000,000 Elementen

```
Treap t;
for (int i=0; i<10000000; ++i)
    t.insert(i);

// ...and search for the 100 first ones
for (int i=0; i<100; ++i)
    t.find(i);
```

0.017 Sekunden

Sortieren mit Treaps

- Folgerung: Durch Einfügen einer Folge von n Zahlen in einen Treap mit zufälligen Prioritäten kann die Folge in Zeit proportional zu $n \log_2 n$ sortiert werden.
- Gleicher optimaler Proportionalitätsfaktor wie Mergesort
- Das Treap-basierte Verfahren ist im wesentlichen äquivalent zu **Quicksort**.

Treaps in C++

- Aehnlich wie bei Listen:
 - Klasse **TreapNode** für die Knoten
 - Jeder **TreapNode** speichert ausser Schlüssel und Priorität zwei Zeiger auf **TreapNodes** (linkes und rechtes **Kind** = Wurzeln des linken und rechten Teilbaums)
 - Klasse **Treap**; jeder **Treap** ist durch einen Zeiger auf den Wurzelknoten repräsentiert.

Die Klasse TreapNode

```
class TreapNode
{
public:
    // constructor
    TreapNode( int key, int priority, TreapNode* left = 0, TreapNode* right = 0);

    // Getters
    int get_key() const;
    int get_priority() const;
    const TreapNode* get_left() const;
    const TreapNode* get_right() const;

private:
    int key_;
    double priority_;
    TreapNode* left_;
    TreapNode* right_;
    friend class Treap;
};
```

Wie bei Facebook: Freunde dürfen auf private Daten und Methoden (Mitgliedsfunktionen) zugreifen.

Die Klasse Treap Die „Viererbande“ + Wurzel

```
class Treap {
public:
    // default constructor:
    // POST: *this is an empty tree
    Treap();
    // copy constructor
    // POST: *this is a copy of other
    Treap(const Treap& other);
    // assignment operator
    // Post: other was assigned to *this
    Treap& operator= (const Treap& other);
    // Destructor
    ~Treap();
private:
    TreapNode* root_;
```

Konstruktor

Copy-Konstruktor

Zuweisungsoperator

Destruktor

Zeiger auf die Wurzel (0: leerer Treap)

Die Klasse Treap Einfügen

```
void Treap::insert(const int key)
{
    insert (root_, key, std::rand());
}
```

private Hilfsfunktion

Zufällige Priorität

Die Klasse Treap Einfügen

```

// PRE: current is a pointer in *this
// POST: a new TreapNode with key and priority is inserted into the
//       subtree hanging off current
void Treap::insert (TreapNode* current, const int key, const int priority)
{
    if (current == 0)
        current = new TreapNode (key, priority);
    else
        if (key < current->key_) {
            insert (current->left_, key, priority);
            if (priority > current->get_priority())
                rotate_right (current);
        }
        else {
            insert (current->right_, key, priority);
            if (priority > current->get_priority())
                rotate_left (current);
        }
}
    
```

9, 10

current verweist auf eine leere Blattposition.

Die Klasse Treap Einfügen

```

// PRE: current is a pointer in *this
// POST: a new TreapNode with key and priority is inserted into the
//       subtree hanging off current
void Treap::insert (TreapNode* current, const int key, const int priority)
{
    if (current == 0)
        current = new TreapNode (key, priority);
    else
        if (key < current->key_) {
            insert (current->left_, key, priority);
            if (priority > current->get_priority())
                rotate_right (current);
        }
        else {
            insert (current->right_, key, priority);
            if (priority > current->get_priority())
                rotate_left (current);
        }
}
    
```

9, 10

current verweist auf eine leere Blattposition.

Damit das klappt, muss current ein Alias des entsprechenden Zeigers im Baum sein, keine Kopie!

Die Klasse Treap Einfügen

```

// PRE: current is a pointer in *this
// POST: a new TreapNode with key and priority is inserted into the
//       subtree hanging off current
void Treap::insert (TreapNode* current, const int key, const int priority)
{
    if (current == 0)
        current = new TreapNode (key, priority);
    else
        if (key < current->key_) {
            insert (current->left_, key, priority);
            if (priority > current->get_priority())
                rotate_right (current);
        }
        else {
            insert (current->right_, key, priority);
            if (priority > current->get_priority())
                rotate_left (current);
        }
}
    
```

9, 10

current verweist auf eine leere Blattposition.

Heap-Eigenschaft ist noch verletzt; das wird „weiter oben“ in der Rekursion repariert.

Die Klasse Treap Einfügen

```

// PRE: current is a pointer in *this
// POST: a new TreapNode with key and priority is inserted into the
//       subtree hanging off current
void Treap::insert (TreapNode* current, const int key, const int priority)
{
    if (current == 0)
        current = new TreapNode (key, priority);
    else
        if (key < current->key_) {
            insert (current->left_, key, priority);
            if (priority > current->get_priority())
                rotate_right (current);
        }
        else {
            insert (current->right_, key, priority);
            if (priority > current->get_priority())
                rotate_left (current);
        }
}
    
```

9, 8

current verweist auf einen Knoten.

Die Klasse Treap Einfügen

```

// PRE: current is a pointer in *this
// POST: a new TreapNode with key and priority is inserted into the
//       subtree hanging off current
void Treap::insert (TreapNode* current, const int key, const int priority)
{
    if (current == 0)
        current = new TreapNode (key, priority);
    else
        if (key < current->key_) {
            insert (current->left_, key, priority);
            if (priority > current->get_priority())
                rotate_right (current);
        }
        else {
            insert (current->right_, key, priority);
            if (priority > current->get_priority())
                rotate_left (current);
        }
}
    
```

9, 10

current verweist auf einen Knoten.

Die Klasse Treap Einfügen

```

// PRE: current is a pointer in *this
// POST: a new TreapNode with key and priority is inserted into the
//       subtree hanging off current
void Treap::insert (TreapNode* current, const int key, const int priority)
{
    if (current == 0)
        current = new TreapNode (key, priority);
    else
        if (key < current->key_) {
            insert (current->left_, key, priority);
            if (priority > current->get_priority())
                rotate_right (current);
        }
        else {
            insert (current->right_, key, priority);
            if (priority > current->get_priority())
                rotate_left (current);
        }
}
    
```

9, 10

current verweist auf einen Knoten.

Die Klasse Treap Einfügen

```

// PRE: current is a pointer in *this
// POST: a new TreapNode with key and priority is inserted into the
//       subtree hanging off current
void Treap::insert (TreapNode* current, const int key, const int priority)
{
    if (current == 0)
        current = new TreapNode (key, priority);
    else
        if (key < current->key_) {
            insert (current->left_, key, priority);
            if (priority > current->get_priority())
                rotate_right (current);
        }
        else {
            insert (current->right_, key, priority);
            if (priority > current->get_priority())
                rotate_left (current);
        }
}
    
```

current verweist auf einen Knoten.

Die Klasse Treap Einfügen

```

// PRE: current is a pointer in *this
// POST: a new TreapNode with key and priority is inserted into the
//       subtree hanging off current
void Treap::insert (TreapNode* current, const int key, const int priority)
{
    if (current == 0)
        current = new TreapNode (key, priority);
    else
        if (key < current->key_) {
            insert (current->left_, key, priority);
            if (priority > current->get_priority())
                rotate_right (current);
        }
        else {
            insert (current->right_, key, priority);
            if (priority > current->get_priority())
                rotate_left (current);
        }
}
    
```

current verweist auf einen Knoten.

Die Klasse Treap Einfügen

```

// PRE: current is a pointer in *this
// POST: a new TreapNode with key and priority is inserted into the
//       subtree hanging off current
void Treap::insert (TreapNode* current, const int key, const int priority)
{
    if (current == 0)
        current = new TreapNode (key, priority);
    else
        if (key < current->key_) {
            insert (current->left_, key, priority);
            if (priority > current->get_priority())
                rotate_right (current);
        }
        else {
            insert (current->right_, key, priority);
            if (priority > current->get_priority())
                rotate_left (current);
        }
}
    
```

current verweist auf einen Knoten.