Programmiertechnik 2

Unit 10: Suchen

Ablauf

- Symboltabellen und binäre Suchbäume
  - Symboltabellen
  - Binäre Suchbäume
  - Balanzierte Bäume
  - 2-3-4-Bäume
  - Rot-Schwarz-Bäume
  - AVL-Bäume
  - Skiplisten
- Datenstrukturen – Hash-Tabellen
  - Hashfunktionen
  - String-Hashing
  - Offene Adressierung
- Hash-Tabellen und Binäre Suchbäume

Symboltabellen

- Suche nach Werten (items), die unter einem Schlüssel (key) gefunden werden können
  - Bankkonten: Schlüssel ist Kontonummer
  - Flugreservierung: Schlüssel ist Flugnummer, Reservierungsnummer, ...
- Symboltabelle: Abstrakter Datentyp
  - Einfügen von neuen Werten unter einem Schlüssel
    - Optional Ändern des Wertes, welcher bereits unter diesem Schlüssel abgelegt ist
  - Auffinden/Ermitteln des Wertes, welches unter einem Schlüssel gespeichert ist
  - Optional: Löschen eines Schlüssels
  - Optional: Ermittlung aller Schlüssel-Wert-Paare
  - Anderer Name: Dictionary, Mapping
  - Typisch: Einfügen kommt weit häufiger vor als bei gedrucktem Wörterbuch

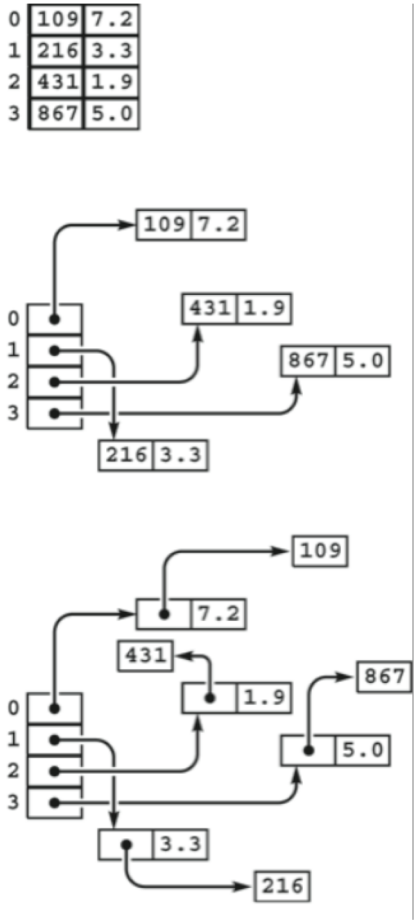
Symboltabellen als Felder/Listen

- Strategie 1: Unsortiertes Feld von Schlüssel-Wert-Paaren
  - Suche in linearer Zeit (lineare Suche)
  - Einfügen evtl. in konstanter Zeit
    - aber: Vergrößerung des Felds
- Strategie 2: Sortiertes Feld
  - Einfügen in linearer Zeit
  - Binäre Suche: logarithmische Zeit
- Strategie 3: Indiziertes Feld
  - Spezialfall von Zahlen als Schlüsseln
  - Schlüssel ist Index
  - Einfügen/Suchen in konstanter Zeit
  - Speicherbedarf linear mit Wertebereich

ST.java

DeDup.java

Schlüssel-indizierte Suche



- Ganzzahlige Schlüssel, Werte sind Gleitkommazahlen
- Drei Darstellungen:
  - Nach Schlüsseln geordnetes Array
  - Array mit Verweisen auf Objekte einer Elementklasse mit zwei einfachen Feldern
  - Array mit Verweisen auf Objekte einer Elementklasse die auf Schlüsselobjekte verweist
- Einfügen:
  - Größere Objekte im Array verschieben
- Suchen:
  - Array durchlaufen

KEY.java

ST_KEY.java

ST_LIST.java

Stringindex

	0	1	2	3	4	5	6	7	8	9
index	27	0	42	8	46	5	37	31	16	21

```
 0 call me sihmael some...
 5 me ishmael some year...
 8 ishmael some years a...
16 some years ago never...
21 years ago never mind...
27 ago never mind how I...
31 never mind how long...
37 mind how long...
42 how long...
46 long...
   ...
```

From Herman Mellville „Moby Dick“, „Call me Ishmael....“

- Schlüsselwortsuche in einem Textstring
  - Definieren Stringschlüssel für jedes Wort im Text
  - Ordnen Schlüssel sortiert in Tabelle an
  - Wenden binäre Suche auf unsere Tabelle mit Stringschlüsseln an
- Beispiel: Suchen nach never
  - Beginnen in der Mitte der Tabelle (4, 46): long < never
  - Suchen im rechten Teilbaum weiter (7, 31): finden Übereinstimmung

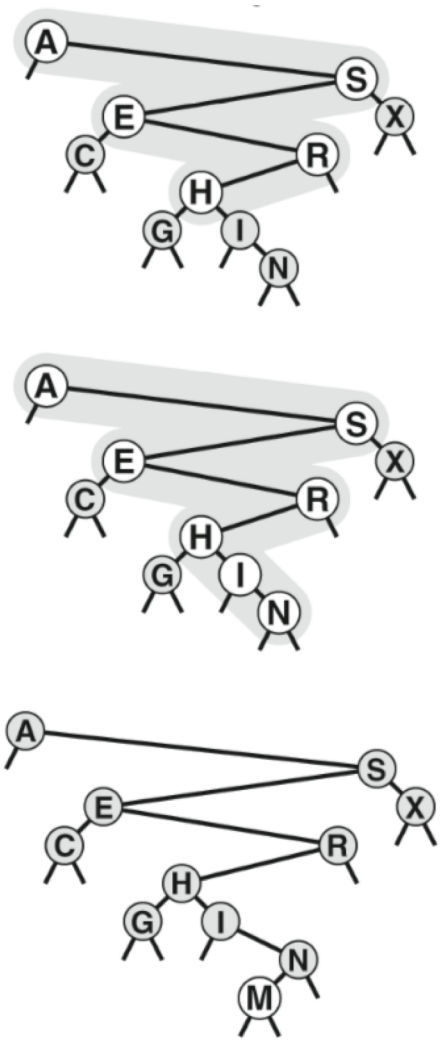
TI.java

TextSearch.java

Binäre Suchbäume

- Ziel: besser-als-linear für Einfügen und Suchen
- Binärer Suchbaum:
  - Binärer Baum: Knoten mit zwei Kindern
    - Kindknoten sind eventuell nicht vorhanden (*null*)
  - Jeder Knoten enthält Schlüssel-Wert-Paar
  - Schlüssel im linken Teilbaum alle kleiner, Schlüssel im rechten Teilbaum alle größer
    - doppelte Schlüssel? Links, rechts, verbieten?
- Suche: beginnend bei Wurzel, rekursiv
- Komplexität der Suche gleich Tiefe des Baums
  - best case: lg N (vollständiger Binärbaum)
  - average case (Gleichverteilung der Schlüssel): 2 ln N (≈ 1.39 lg N)
  - worst case: N (entarteter Baum: Liste)

Suchen und Einfügen in binary search trees (BSTs)



Bei einer erfolgreichen Suche nach H *in diesem Beispielbaum (oben) gehen wir an der Wurzel nach rechts (da* H *größer als* A *ist), dann nach links im rechten Teilbaum der Wurzel (da* H *kleiner als* S *ist). Auf diese Weise durchlaufen wir den Baum weiter nach unten, bis der Knoten mit dem* H *erreicht ist.*
Bei einer nicht erfolgreichen Suche nach M *in diesem Beispielbaum (Mitte) gehen wir an der Wurzel nach rechts (da* M *größer als* A *ist), dann nach links im rechten Teilbaum der Wurzel (da* M *kleiner als* S *ist). Auf diese Weise durchlaufen wir den Baum weiter nach unten, bis wir auf eine externe Verbindung in der untersten Ebene links vom* N *treffen.*
Um M *nach dem Suchfehler einzufügen, ersetzen wir einfach die Verbindung, die die Suche terminiert hat, durch eine Verbindung zu* M *(unten).*

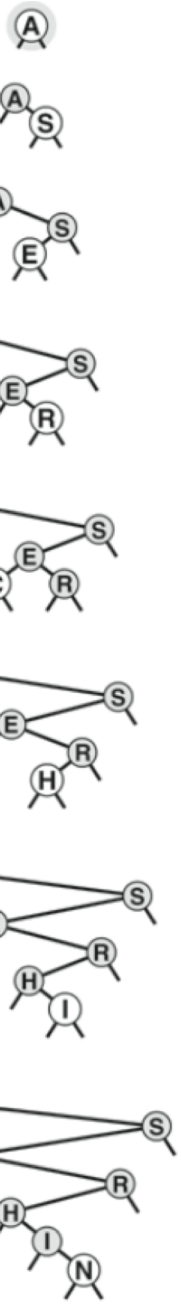
Einfügen in Binäre Suchbäume

- Naive Lösung: neue Knoten werden immer als Blätter (ohne Kindknoten) eingefügt
  - Suche beginnend bei Wurzel
  - Absteigen zu Kindknoten entsprechend Ordnung, bis Kindzeiger null
  - Problem: Baum kann entarten

- Balanzierte Bäume: Nach Einfügen wird Baum wieder ausgeglichen

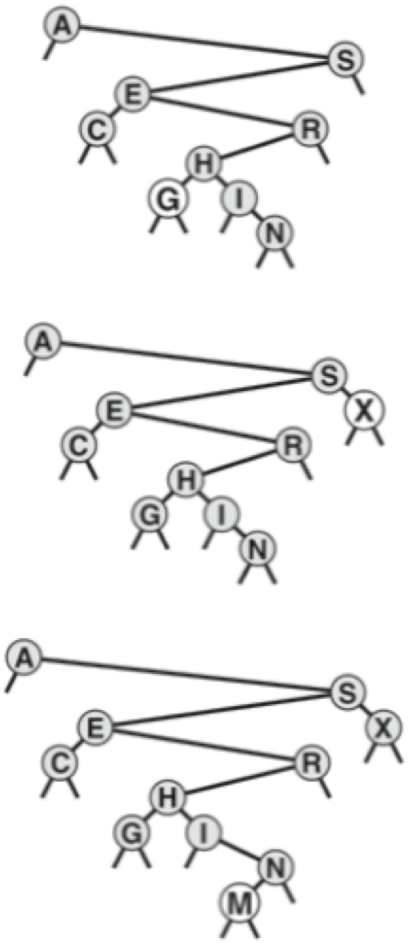
ST_BST.java

Konstruktion eines binären Suchbaums (Beispiel)

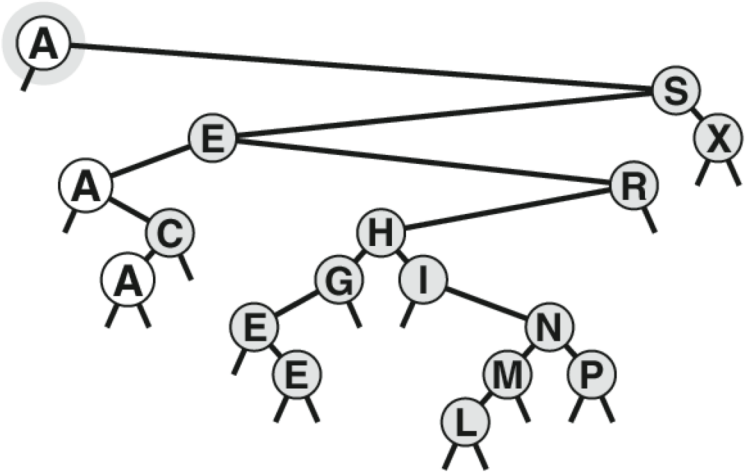


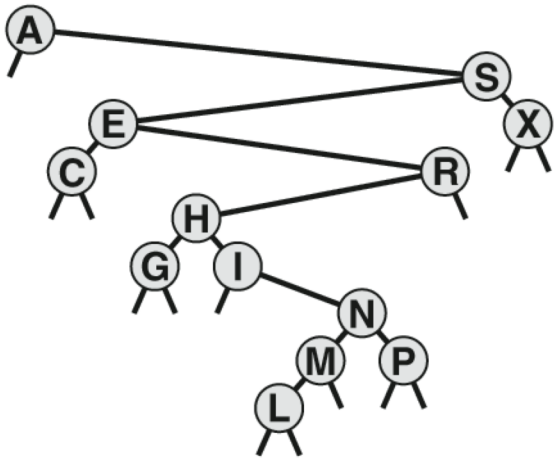
Diese Sequenz zeigt, wie die Schlüssel A S E R C H I N *in einen anfangs leeren binären Suchbaum eingefügt werden. Jedes Einfügen folgt auf einen Such-fehler auf der untersten Ebene des Baums.*

Rechts werden G X M eingefügt; der Rest der Folge P L E fehlt

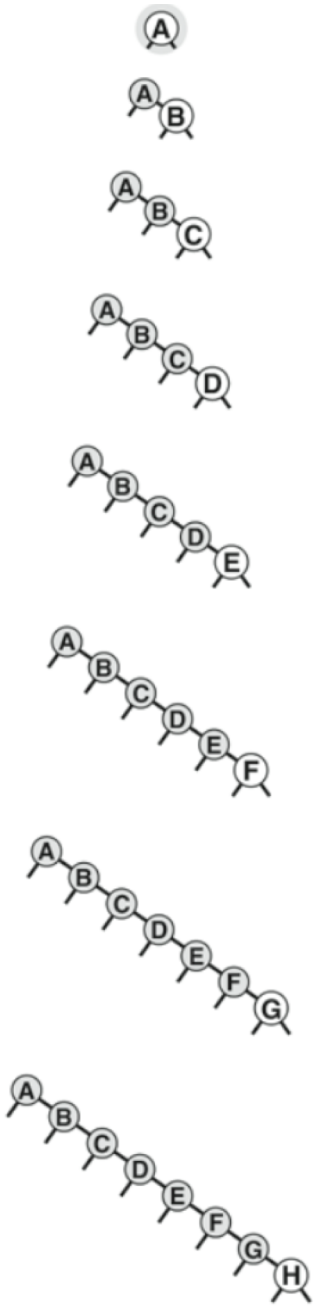


Problem doppelter Schlüssel





- Datensätze mit doppelten Schlüsseln sind über den Baum verstreut
- Werden gefunden wenn man bei Übereinstimmung Suche fortsetzt
- Abbruchkriterium: null-Zeiger

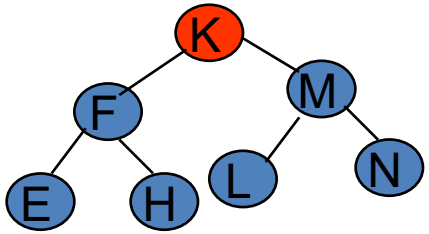


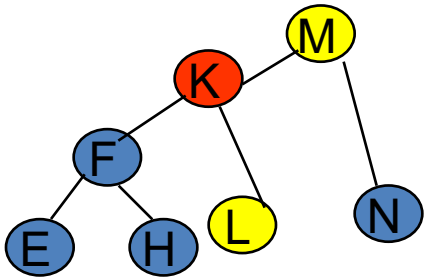
Problem Entartung

- Binärer Suchbaum entartet wenn Schlüssel in aufsteigender Reihenfolge eintreffen
- Dann:
  - Einfach verkettet Liste
  - Quadratische Konstruktionszeit
  - Lineare Suchzeit

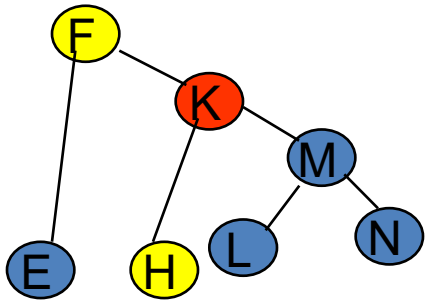
Rotation in Bäumen

- Linksrotation:
  Wurzel wird linkes Kind





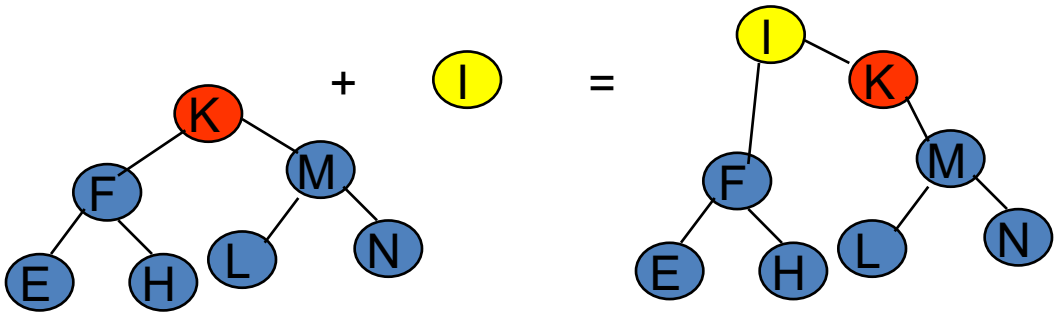
- Rechtsrotation:
  Wurzel wird rechtes Kind



- Rotation ist lokale Änderung
- Ermöglich Knotenverschiebung, ohne globale Ordnungseigenschaften zu verletzen
- Basis für Operationen Einfügen, Entfernen, Verknüpfen

Einfügen in Wurzel

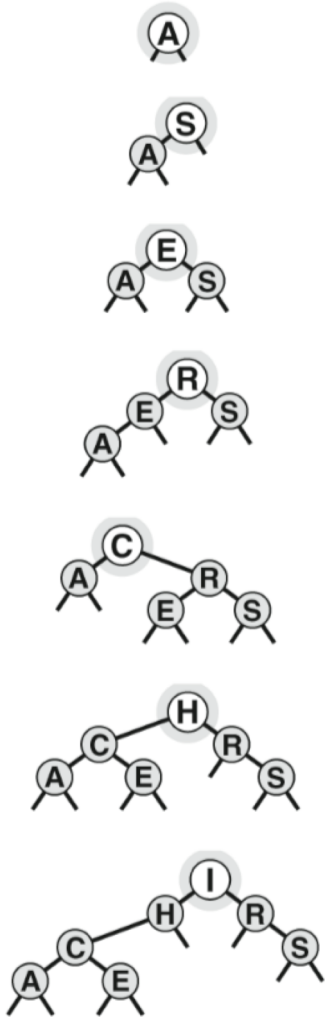
- naives Einfügen: neuer Knoten als Blatt
- Einfügen in Wurzel: neuer Knoten wird Wurzel
  - Suche nach frisch eingefügten Schlüsseln geht schneller
- Problem: Binäre Ordnung wird potentiell verletzt
- Lösung: Rotieren des Baums nach Einfügen in Teilbaum
  - Rechtsrotation nach Einfügen in linken Teilbaum
  - Linksrotation nach Einfügen in rechten Teilbaum



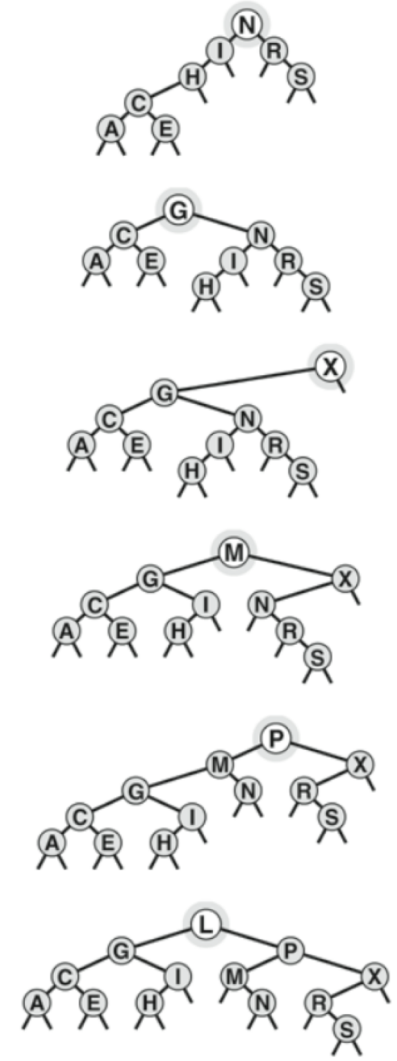
Einfügen in Wurzel (2)

```
static Node insert(Node h, ITEM x)
{
  if (h==null) return new Node(x);
  if (x.key().less( h.item.key())) {
     h.l = insert(h.l, x); h = rotR(h);
  } else {
     h.r = insert(h.r, x); h = rotL(h);
  }
  return h;
}
private Node rotR(Node h)
  { Node x = h.l; h.l = x.r; x.r = h; return x; }
private Node rotL(Node h)
  { Node x = h.r; h.r = x.l; x.l = h; return x; }
```

Konstruktion eines BST durch Einfügen an der Wurzel



Diese Sequenz zeigt, wie die Schlüssel A S E R C H I *in einen anfangs leeren binären Suchbaum nach dem Verfahren Einfügen an der Wurzel eingefügt werden. Jeder neue Knoten wird an der Wurzel eingefügt, wobei die Verbindungen entlang seines Suchpfads geändert werden, um einen richtigen binären Suchbaum zu erhalten.*



Diese Sequenz zeigt, wie die Schlüssel N G X M P L *in den gemäß Abbildung 12.16 begonnenen binären Suchbaum eingefügt werden.*

Zerlegen eines binären Suchbaums

- k+1-kleinsten Schlüssel an die Wurzel bringen
- Rekursive Operation

```
Node partR(Node h, int k) {
    int t = (h.l == null) ? 0 : h.l.N;
    if (t > k) {
        partR(h.l, k); h = rotR(h);
    }
    if (t < k) {
        partR(h.r, k-t-1); h = rotL(h);
    }
    return h;
}
```

Binären Suchbaum ausgleichen

- Rekursive Methode
- Gleicht Baum in linearer Zeit perfekt aus
- Bringt Median-Knoten an die Wurzel
  - Dieser Schritt wird für die Teilbäume (rekursiv) wiederholt

```
private Node balanceR(Node h) {
  if ((h == null) || (h.N == 1)) return h;
  h = partR(h, h.N/2);
  h.l = balanceR(h.l);
  h.r = balanceR(h.r);
  // Größe N für alle Teilbäume neu berechnen
  fixN(h.l); fixN(h.r); fixN(h);
  return h;
}
```

Balancierte Bäume

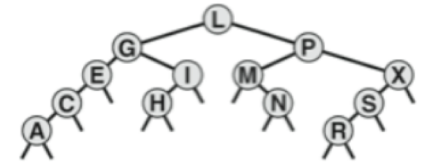
- Ziel: entartete Bäume sollen vermieden werden
- Lösung 1: probabilistischer Algorithmus
  - Schlüssel werden in randomisierter Reihenfolge eingefügt
- Lösung 2: amortisierende Algorithmen
  - einzelne Einfügeoperationen eventuell teuer durch Neubalancierung
  - Mittelwert für viele Operationen soll “günstig” sein
- Lösung 3: optimale Algorithmen
  - Performancegarantie für jede Operation
  - Buchhaltung über Struktur des Baums erforderlich

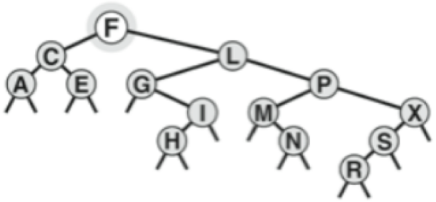
Einfügen in randomisierten binären Suchbaum

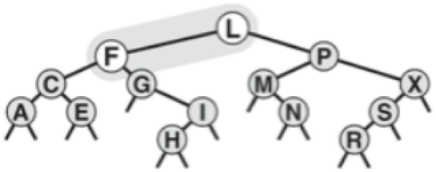
- Jeder Knoten kann mit gleicher Wahrscheinlichkeit Wurzel sein
  - Zufällig an der Wurzel (mit Rotation) oder tiefer im Baum (Standardverfahren) einfügen
  - Algorithmus liefert garantierte Leistung in probabilistischem Sinne

```
private Node insertR(Node h, ITEM x) {
   if (h == null) return new Node(x);
   if (Math.random() * h.N < 1.0) return insertT(h, x);
   if (x.key().less( h.item.key() )) h.l = insertR(h.l, x);
   else h.r = insertR(h.r, x);
   h.N++;
   return h;
}
void insert(ITEM x)  { head = insertR(head, x); }
```

Beispiel randomisierter Suchbaum

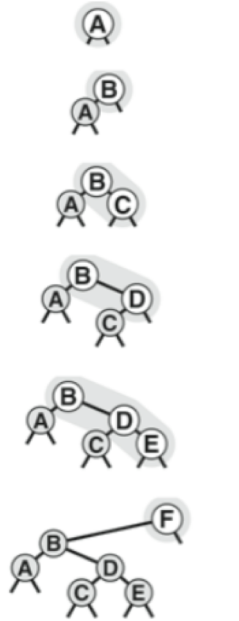






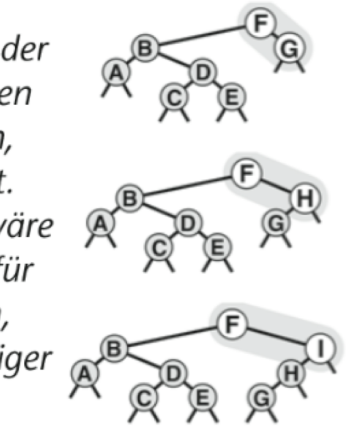
Einfügen:

Die letzte Position eines neuen Datensatzes in einem randomisierten binären Suchbaum kann sich überall im Suchpfad des Datensatzes befinden, je nach dem Ergebnis zufälliger Entscheidungen, die während der Suche getroffen wurden. Diese Abbildung zeigt jede der möglichen Endpositionen für einen Datensatz mit dem Schlüssel F, *wenn der Datensatz in den Beispielbaum (oben) eingefügt wird.*



Konstruktion:

Diese Sequenz zeigt das Einfügen der Schlüssel A B C D E F G H I *in einen anfangs leeren binären Suchbaum, wobei das Einfügen zufällig erfolgt. Der untere Baum sieht so aus, als wäre er mit dem Standardalgorithmus für binäre Suchbäume erstellt worden, wobei dieselben Schlüssel in zufälliger Reihenfolge eingefügt wurden.*



Komplexität

- Erstellen eines randomisierten binären Suchbaums ist äquivalent zum Erstellen eines BST aus einer zufälligen Anfrangspermutation der Schlüssel
  - 2 N ln N Vergleiche für Konstruktion
  - 2 ln N Vergleiche für Suchoperationen

- Die Wahrscheinlichkeit, dass Konstruktionskosten eines randomisierten BST um das α-fache über dem Durchschnitt liegt, ist geringer als $e^{-\alpha}$

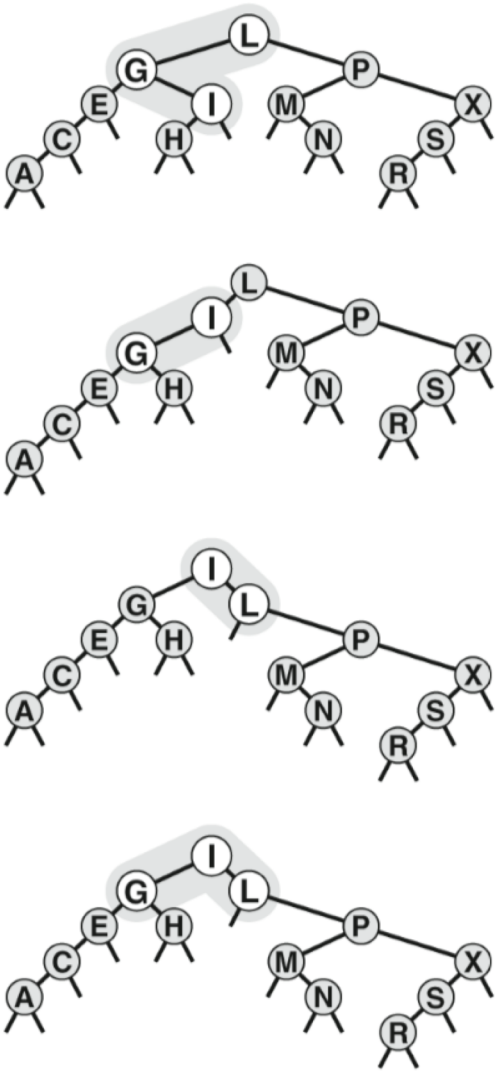
  - 2,3 Mio Vergleiche für Erstellung eines randomisierten BST mit 100.000 Knoten
  - Wahrscheinlichkeit, dass Anzahl der Vergleiche > 23 Mio ist kleiner als 0.01 %

(C. Martínez, Salvador Roura; „Randomization of Search Trees by Subtree Size“, 4th Europ. Symp. on Alg., Barcelona,1996)

Splay Trees

- D. Sleator und R. E. Tarjan 1985
  - „Self-adjusting binary search trees“, JACM 32, 1985
- Umordnung des Baums sowohl beim Einfügen als auch beim Suchen
  - gesuchter Schlüssel wird stets Wurzel des Baums
  - to splay: “spreizen, teilen”
- Umsortieren durch Rotieren – Rotationen sollen Baum auch ausgleichen
  1. Knoten N, Elternknoten P, Großelternknoten G
  2. N links von P, links von G: doppelte Rechtsrotation
  3. N rechts von P, rechts von G: doppelte Linksrotation
  4. N links von P, rechts von G: rechts-dann-links
  5. N rechts von P, links von G: links-dann-rechts
- Amortisierte Komplexität: M Operationen (Einfügen oder Suchen) auf einem Baum mit N Knoten benötigen O((N+M) log (N+M))
  - Worst-case für eine einzelne Operation: O(N+M)

Splay-Einfügen

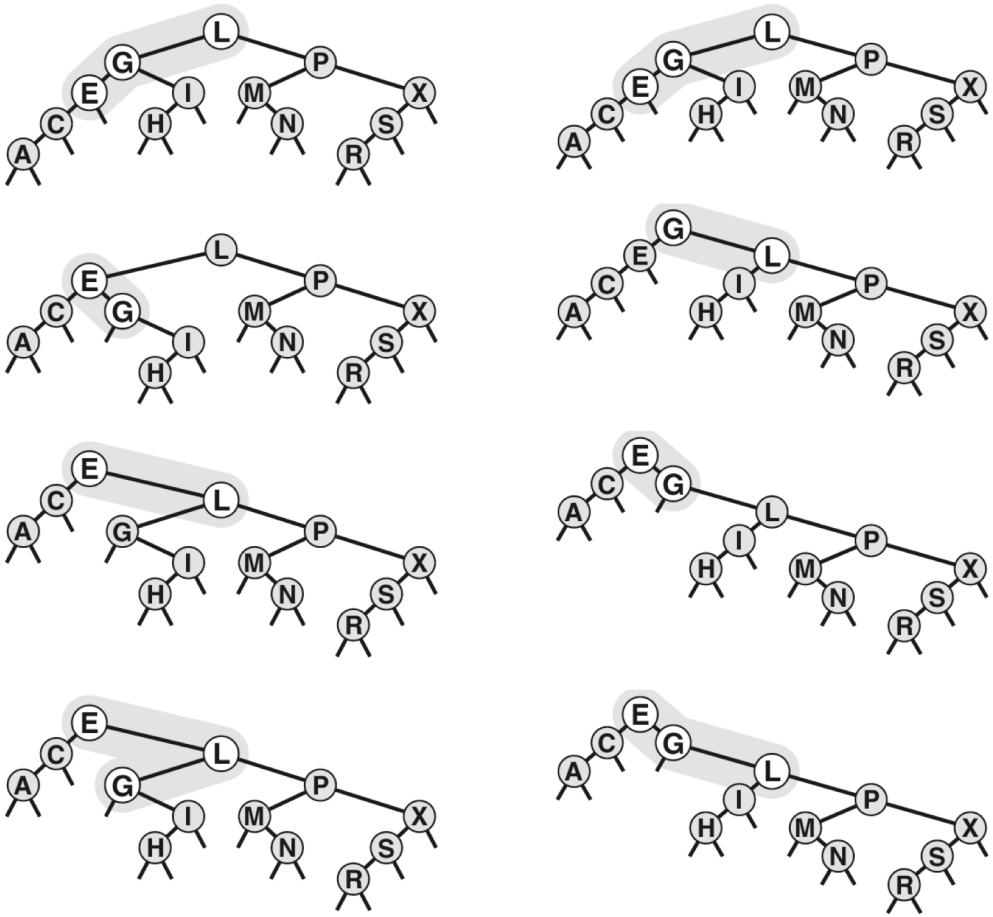


In diesem Beispielbaum (oben) bringt eine Linksrotation bei G *gefolgt von einer Rechtsrotation bei* L *den Knoten* I *zur Wurzel (unten). Diese Rotationen können das Einfügen an der Wurzel bei einem Standard- oder Splay-BST abschließen.*

Verbindungen von I zur Wurzel sind unterschiedlich orientiert

- Betrachten 2 Rotationen
  (anstelle rekursiver Rotationen)
- Bringen einzufügenden Knoten aus Position des „übernächsten Nachfolgers“ der Wurzel zur Spitze des Baumes

Splay-Einfügen (2)



Hier:

- Beide Verbindungen in einer Doppelrotation sind in gleicher Richtung orientiert

2 Optionen:

- Einfügen an der Wurzel: untere Rotation zuerst ausführen (links)
- Splay-Einfügen: höhere Rotation zuerst ausführen (rechts)

Suchen:

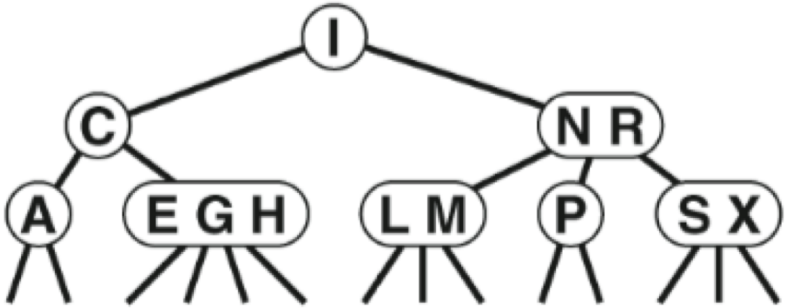
- Auch hierbei werden Rotationen ausgeführt
- Dadurch wird die Länge der Suchpfade verkürzt

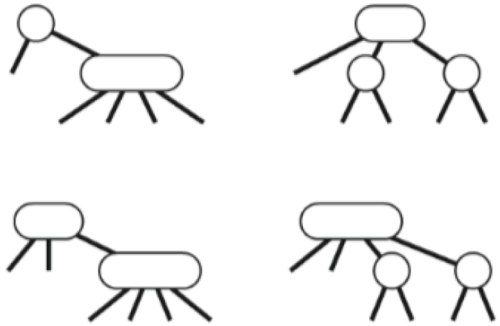
2-3-4-Bäume

- Drei Arten von Knoten
  - 2-Knoten: Ein Schlüssel, zwei Kindknoten
  - 3-Knoten: Zwei Schlüssel, drei Kindknoten
  - 4-Knoten: Drei Schlüssel, vier Kindknoten
- Balancierter 2-3-4-Baum: Alle Blätter haben den gleichen Abstand von der Wurzel
- Einfügen in balancierten Baum erhält Balance
  - Einfügen neuer Schlüssel immer in Blättern
  - Einfügen in 2-Blatt erzeugt 3-Blatt
  - Einfügen in 3-Blatt erzeugt 4-Blatt
  - Zerlegen von 4-Knoten auf dem Weg von der Wurzel
    - 2-Knoten mit 4-Knoten als Kind wird zu 3-Knoten mit 2 weiteren 2-Kindern
    - 3-Knoten mit 4-Knoten als Kind wird zu 4-Knoten mit 2 weitern 4-Kindern
    - Ist die Wurzel ein 4-Knoten, wird sie in 2-Knoten zerlegt – Tiefe des Baums steigt

Einfügeoperationen erhalten Balance

Jeder 4-Knoten, der kein Nachfolger eines 4-Knoten ist, lässt sich Teilen durch Übergeben des mittleren Datensatzes an Vorgänger





Diese Abbildung zeigt einen 2-3-4-Baum, der die Schlüssel A S R C H I N G E X M P L *enthält. In einem derartigen Baum können wir einen Schlüssel finden, indem wir anhand der Schlüssel im Wurzelknoten die Verbindung zu einem Teilbaum bestimmen. Um zum Beispiel in diesem Baum nach* P *zu suchen, folgen wir der rechten Verbindung von der Wurzel, da* P *größer als* I *ist, folgen der mittleren Verbindung vom rechten Nachfolger der Wurzel, da* P *zwischen* N *und* R *liegt und schließen dann erfolgreich die Suche am 2-Knoten ab, der* P *enthält.*

2-3-4-Bäume: Analyse

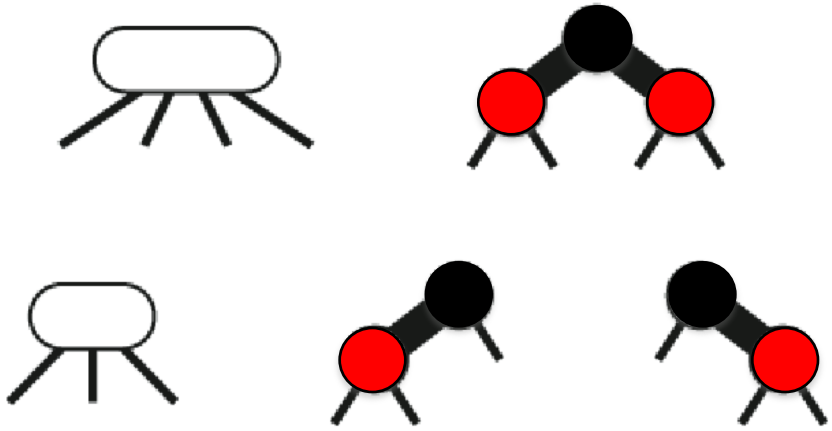
- Baum mit N Knoten enthält weniger als 3N Schlüssel
- Suchen in Baum mit N Knoten: höchstens lg N + 1 Knoten werden inspiziert
- Einfügen in Baum mit N Knoten: höchstens lg N + 1 Teilungen
  - average case bisher nicht analysiert
  - Vermutung: im Mittel weniger als eine Teilung pro Einfügeoperation

Rot-Schwarz-Bäume

- erfunden von Bayer 1972
- Binärer Suchbaum
- Knoten haben “Farbe”: rot oder schwarz
  - oft repräsentiert in einem Bit
  - Darstellung der Farbe entweder in Knoten oder in Verweis auf den Knoten
- Wurzel ist immer schwarz
- Jeder Pfad im Baum hat gleich viele schwarze Knoten
- Jeder rote Knoten hat nur schwarze Kinder
  - Gesamttiefe des Baums höchstens 2x Zahl der schwarzen Knoten
- Interpretation des 2-3-4-Baums als Rot-Schwarz-Baum
  - Knoten mit schwarzen Kindern: 2-Knoten
  - Knoten mit einem roten Kind: 3-Knoten
  - Knoten mit zwei roten Kindern: 4-Knoten

Rot-Schwarz-Bäume

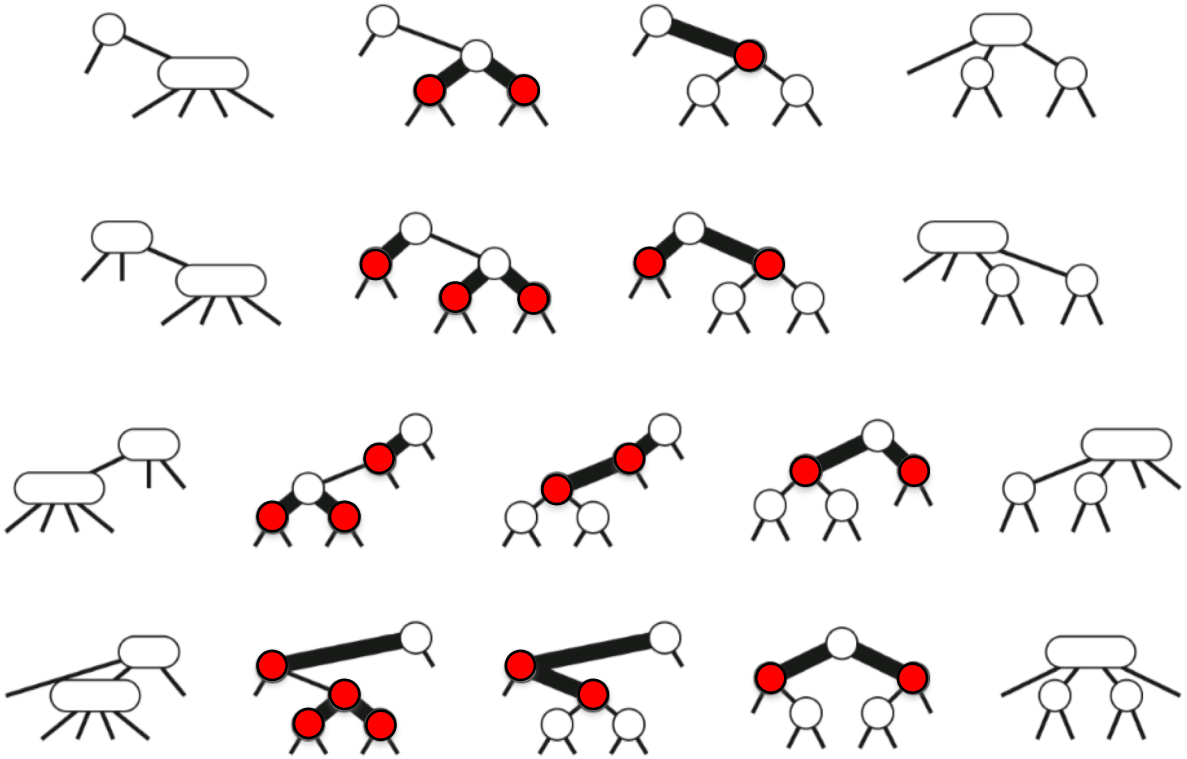
- Stellen 2-3-4-Baum als binären Rot-Schwarz-Baum dar



Ein 4-Knoten (links oben) wird durch einen ausgeglichenen Teilbaum von drei 2-Knoten dargestellt, zwischen denen rote Verbindungen bestehen (rechts oben). Beide haben drei Schlüssel und vier schwarze Verbindungen. Ein 3-Knoten (links unten) wird durch einen 2-Knoten dargestellt, der mit einer einzelnen roten Verbindung (entweder nach rechts oder nach links) zu einem anderen 2-Knoten verbunden ist (rechts unten). Alle haben zwei Schlüssel und drei schwarze Verbindungen.

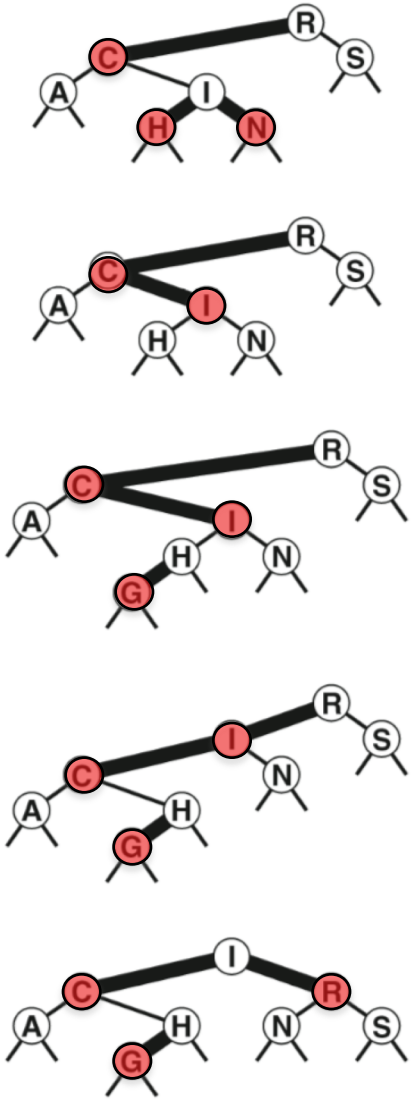
- Suchverfahren berücksichtigt Knotenfarbe nicht → keine zusätzliche Zeit für Standardsuchverfahren
- Kein Aufwand für Ausgleichen während Suche
- Nur Einfügen muss Knoten mit zwei roten Kindern berücksichtigen

Teilen von 4-Knoten in Rot-Schwarz-Baum



In einem Rot-Schwarz-Baum implementieren wir die Teilung eines 4-Knotens, der kein Nachfolger eines 4-Knotens ist, indem wir die Knotenfarben in den drei Knoten, aus denen sich der Knoten aufbaut, wechseln und dann gegebenenfalls eine oder zwei Rotationen durchführen. Wenn der Vorgänger ein 2-Knoten ist (oben) oder ein 3-Knoten, der eine geeignete Orientierung aufweist (zweite Zeile von oben), sind keine Rotationen erforderlich. Befindet sich der 4-Knoten an der mittleren Verbindung des 3-Knotens (unten), ist eine Doppelrotation notwendig; andernfalls genügt eine einfache Rotation (dritte Zeile von oben).

Einfügen in Rot-Schwarz-Bäume



Diese Zeichnung zeigt das Ergebnis (unten) für das Einfügen eines Datensatzes mit dem Schlüssel G in den Rot-Schwarz-Baum (oben). In diesem Fall besteht der Einfügevorgang aus folgenden Schritten: Aufteilen des 4-Knotens bei I mit einem Farbenwechsel (zweite Zeichnung von oben), danach Hinzufügen des neuen Knotens in der untersten Ebene (dritte Zeichnung von oben), dann (bei Rückkehr zu jedem Knoten auf dem Suchpfad im Code nach den rekursiven Methodenaufrufen) Ausführen einer Linksrotation bei C und einer Rechtsrotation bei R, um das Aufteilen des 4-Knotens abzuschließen.

RSTree.java

Analyse

- Eine Suche in einem Rot-Schwarz-Baum mit N Knoten benötigt weniger als 2 ld N + 2 Vergleiche
  - Nur Teilungen, die einem 3-Knoten, der mit einem 4-Knoten im 2-3-4-Baum verbunden ist, entsprechen, erfordern eine Rotation
  - Ungünstigster Fall: Pfad zum Einfügepunkt besteht aus alternierenden 3-Knoten und 4-Knoten

- Suche im Rot-Schwarz-Baum mit N Knoten benötigt 1,002 ld N Vergleiche
  - Wenn Baum aus zufälligen Schlüsseln aufgebaut ist

- In einem ausgeglichenen Rot-Schwarz-Baum enthalten alle Pfade von Blättern zur Wurzel die gleiche Anzahl schwarzer Knoten

AVL-Bäume

- Г. М. Адельсон-Вельский, Е. М. Ландис. Один алгоритм организации информации
  - Доклады АН СССР. 1962. Т. 146, № 2. С. 263–266.
  - Adelson-Velsky, Landis
- Forderung: Höhe des linken Teilbaums unterscheidet sich von Höhe des rechten Teilbaums höchstens um 1
  - Speicherung der Höhendifferenz im Elternknoten (-1/0/+1)
- Neubalancierung nach Verletzung der AVL-Bedingung:
  - Höhendifferenz ist +/-2 nach Einfügen oder Löschen
  - betrachte wieder drei Knoten N, P, G, so, dass N tieferes Kind (+1) von P und P tieferes Kind von G
  - N links von P links von G: Rechtsrotation von G
  - N rechts von P links von G: Linksrotation von P, dann Rechtsrotation von G
  - andere 2 Fälle symmetrisch

AVL-Bäume (2)

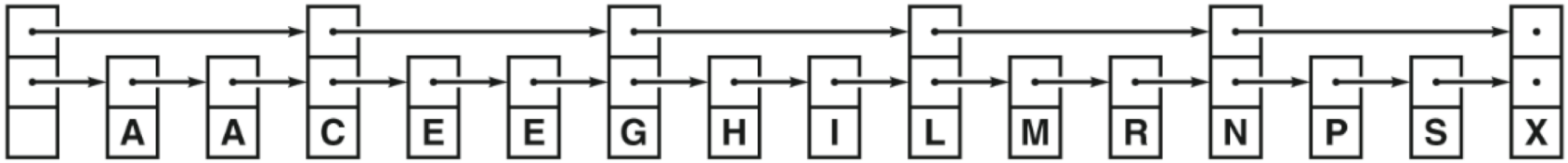
- Einfügen
  - füge in Blatt ein
  - danach: aktualisiere Höhenangaben, rebalanciere
- Löschen
  - eines Blatts: entferne Blatt
  - eines inneren Knotens
    - Finde wahlweise kleinsten folgenden Schlüssel (kleinsten Knoten im rechten Kind) oder größten vorhergehenden Schlüssel; gefundener Knoten hat höchstens einen Kindknoten
    - Ersetze zu löschenden Knoten mit gefundenem
  - danach: aktualisiere Höhenangaben, rebalanciere

Analyse

- Aktualisierung von Höhenangaben:
  - jede Operation (Einfügen, Löschen) meldet Höhenänderung
    - Höhenunterschied 0 kann sofort "nach oben" gemeldet werden
  - addiere zu aktuellem Höhenunterschied
  - +/-2: rebalanciere
  - Einfügen: Höhenunterschied +/-1 bedeutet insgesamt Höhenzunahme
  - Löschen: Höhenunterschied 0 bedeutet Höhenverlust
- Komplexität
  - worst-case für Tiefe des Baums: ca 1.44 log N
  - Suche: O(log N)
  - Einfügen: maximal 1 Doppelrotation nötig; O(log N)
  - Löschen: im worst case 1 Rebalancierung pro Knoten auf Pfad; O(log N)

Skiplisten

- Verkettete Liste wird um weitere Ebenen ergänzt
- i-te Ebene überspringt Elemente mit weniger als i Verbindungen
- Ausgeglichene Skipliste: In i-ter Ebene werden (mindestens) i Elemente übersprungen

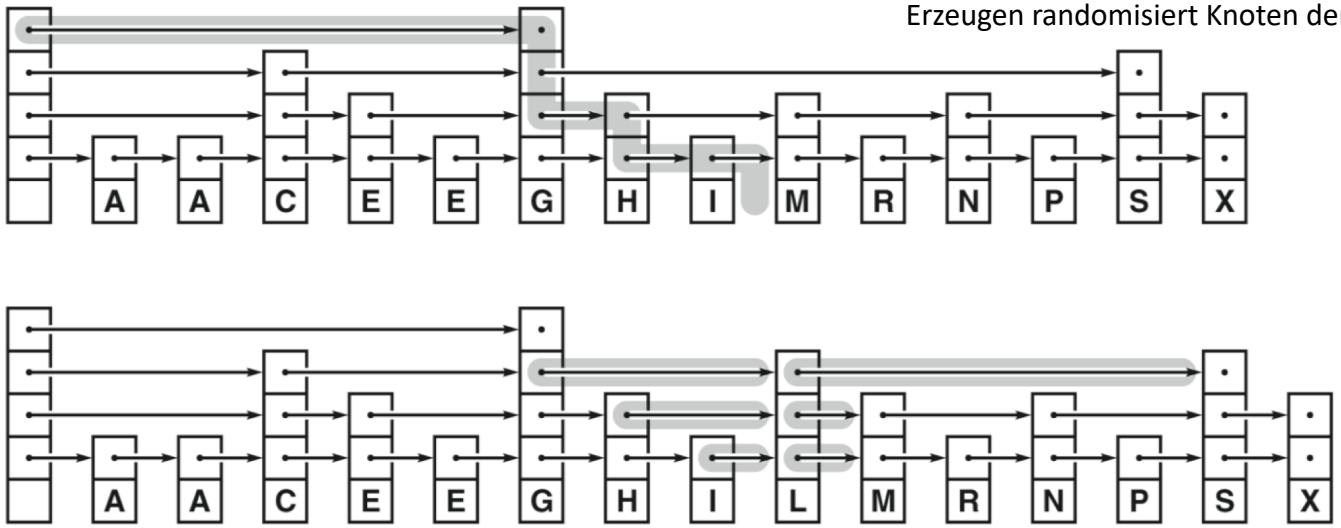


Jeder dritte Knoten in dieser Liste hat eine zweite Verbindung, sodass wir mit nahezu der dreifachen Geschwindigkeit durch die Liste springen können, wenn wir den ersten Verbindungen folgen. Zum Beispiel können wir vom Beginn der Liste an zum zwölften Knoten in der Liste, dem P, *gelangen, indem wir lediglich fünf Verbindungen folgen: den jeweils zweiten Verbindungen zu* C, G, L, N *und dann über die erste Verbindung von* N *zu* P.

- Algorithmen für verkettete Listen lassen sich leicht auf Skiplisten übertragen

Einfügen in Skiplisten

Erzeugen randomisiert Knoten der Ordnung 0..3



Indem wir die Struktur gemäß *Folie 37* *um weitere Ebenen ergänzen und den Verbindungen ermöglichen, eine variable Anzahl von Knoten zu überspringen, erhalten wir ein Beispiel für eine allgemeine Skipliste. Um nach einem Schlüssel in der Liste zu suchen, beginnen wir auf der höchsten Ebene und gehen jedes Mal eine Ebene tiefer, wenn wir einen Schlüssel finden, der nicht kleiner als der Suchschlüssel ist. Hier (oben) suchen wir* L, *indem wir auf Ebene 3 beginnen und dann folgende Bewegungen ausführen: hinüber zur ersten Verbindung, danach bei* G *nach unten (und dabei die Nullverbindung als Verbindung auf einen Markierungsschlüssel behandeln), als Nächstes hinüber zu* I, *dann nach unten auf Ebene 2, weil* S *größer als* L *ist, und schließlich nach unten auf Ebene 1, weil* M *größer als* L *ist. Um einen Knoten* L *mit drei Verbindungen einzufügen, klinken wir ihn in die drei Listen an genau den Plätzen ein, wo wir beim Suchen Verbindungen zu größeren Schlüsseln gefunden haben.*

Eigenschaften

- Suchen und Einfügen in einer randomisierten Skipliste mit Ordnung t benötigen im Durchschnitt $(t \log_t N) / 2 = (t / (2 \text{ ld } t))$ ld N Vergleiche
- Skiplisten enthalten im Durchscnitt (t / (t-1)) N Verbindungen
- Zugrundeliegende Datenstruktur ist eine alternative Darstellung eines ausgeglichenen Baumes...

Hash-Tabellen: Überblick

- Alle bislang betrachteten Verfahren funktionieren am besten bei eindeutigen Schlüsseln
  - → Wie kann man eindeutige Schlüssel erzeugen?
  - → Wie kann man große, dünn besetzte Wertebereiche „verdichten“

- Hashfunktionen: Abbildung von Schlüsseln auf Zahlen
  - Hashwert: Wert der Hashfunktion
- Hashtabelle: Symboltabelle, die mit Hashwerten indiziert ist
- Kollision: Paar von Schlüsseln mit gleichem Index in Hashtabelle
  - Kollisionsauflösung: Suche in kollidierenden Schlüsseln
- Kompromiss zwischen Speicherverbrauch und Rechenzeit
  - unbegrenzter Speicher: Schlüssel ist Index; Rechenzeit ist konstant
  - unbegrenzte Zeit: lineare Suche in Schlüsseln, kein zusätzlicher Speicher

Multiplikative Hashfunktion für Gleitkommaschlüssel

.513870656	51
.175725579	17
.308633685	30
.534531713	53
.947630227	94
.171727657	17
.702230930	70
.226416826	22
.494766086	49
.124698631	12
.083895385	8
.389629811	38
.277230144	27
.368053228	36
.983458996	98
.535386205	53
.765678883	76
.646473587	64
.767143786	76
.780236185	78
.822962105	82
.151921138	15
.625476837	62
.314676344	31
.346903890	34

Um Gleitkommazahlen zwischen 0 und 1 in Tabellenindizes für eine Tabelle der Größe 97 zu konvertieren, multiplizieren wir mit 97. In diesem Beispiel gibt es drei Kollisionen: bei 17, 53 und 76. Die höchstwertigen Bits der Schlüssel bestimmen die Hash-Werte; die niederwertigsten Bits der Schlüssel spielen keine Rolle. Ein Ziel beim Entwurf von Hashfunktionen besteht darin, solche Unausgeglichenheiten zu vermeiden, indem man möglichst jedem Datenbit eine Bedeutung zuweist.

Hashfunktionen

- Abbildung von Schlüsseln auf Werte von 0..M-1 (M: Größe der Tabelle)
  - Idealfall: Hashwerte sind gleichverteilt
- Hashfunktion hängt vom Schlüsseltyp ab
  - üblich: Berechnung des Hashwerts auf Basis der internen Darstellung des Schlüsselwerts
  - Java: int java.lang.Object.hashCode();
- Beispiel: Gleitkommazahlen f von B bis E
  - Abbildung auf 0..1, dann Multiplikation mit M
  - hash(f) = $\lfloor (f-B)/(E-B)*M \rfloor$
- Beispiel: Natürliche Zahlen z von 0 bis 2^w
  1. Abbildung auf Gleitkommazahlen von 0..1, dann wie Gleitkommazahlen
     - Gleichverteilung nur dann erreicht, wenn Zahlen gleichverteilt sind
  2. Modulo-Hashing: hash(z) = z % M
     - in vielen Fällen erscheinen die Reste modulo M wie zufällig

Hashfunktionen (2)

- Gleichverteilung: Alle Bits des Schlüssels sollten berücksichtigt werden
- Modulo-Hashing: falls M eine Zweierpotenz, werden nur die unteren log M Bits berücksichtigt
- üblich: M sollte eine Primzahl sein
  - mögliche Werte für M sind oft in Programm kodiert
    - etwa Mersenne-Primzahlen: 2^t-1 für t = 2, 3, 5, 7, 13, 17, 19, 31, ...
- Schnelles Hashing: Design von Hashfunktionen beachtet nicht nur Gleichverteilung, sondern auch effiziente Implementierbarkeit
- Sicheres Hashing (Kryptologie): Hash-Funktion sollte nicht umkehrbar sein
  - MD-5 (RFC 1321): Message Digest #5
  - SHA-224 (RFC 3874): Secure Hash Algorithm

String-Hashing

- Hashfunktionen für Strings: Berücksichtigung aller Zeichen
- Betrachtung des Strings als ganze Zahl, etwa zur Basis 256 (8-bit Zeichen) oder 65536 (16-Bit-Zeichen)
  - Integer-Arithmetik auf großen Zahlen ist aber ineffizient
- Modulo-Hashing mit Hilfe des Hornerschemas
  - Beispiel: String a,b,c,d, 8-bit-Zeichen (0..255)
  - $(a*256^3+b*256^2+c*256+d) \% M =$
    $(((a*256 + b)*256 + c)*256+d) \% M =$
    $(((a*256 + b) \% M * 256 + c) \% M *256 + d) \% M$
  - Zwischenergebnisse verlassen niemals den Wertebereich für int
- Verwendung von 256 als Faktor nicht formal erforderlich
  - Verteilung wird u.U. besser, wenn Faktor keine Zweierpotenz

String-Hashing (2)

```
static int hash(String s, int M){
  int h = 0, a = 31; // 31 ist Faktor im JDK
  for (int i = 0; i < s.length(); i++)
        h = (a*h + s.charAt(i)) % M;
  return h;
}
// verwenden pseudozufällig Koeffizienwerte
// Kollisionn treten nur mit Wahrscheinlichkeit 1/M auf
static int hashU(String s, int M){
  int h = 0, a = 31415, b = 27183;
  for (int i = 0; i < s.length(); i++){
        h = (a*h + s.charAt(i)) % M;
        a = a*b % (M-1);
  }
  return h;
}
```

String-Hashing (3)

- java.lang.String.hashCode:
  - Faktor ist 31
  - hashCode wird nur beim ersten Zugriff berechnet, danach gespeichert (cached)
- Wiederverwendung von Hash-Funktionen:
  - Objekte implementieren nur .hashCode()
    - Rückgabetyp ist z.B. int
  - Rufer von .hashCode() führen Modulo-Operation selbst aus

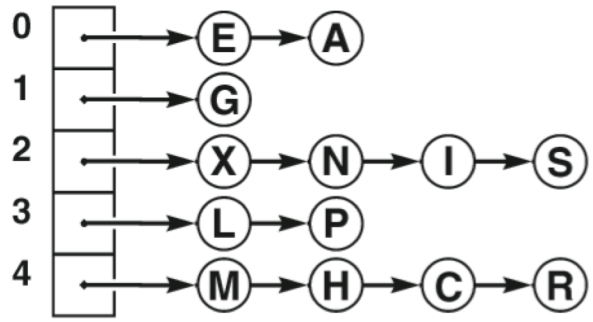
Perfektes Hashing

- Annahme: Menge M der Schlüssel ist vorab bekannt
  - Beispiel: Namen von Methoden eines Objekts
  - Beispiel: Liste von Schlüsselwörtern einer Programmiersprache
- Perfekte Hashfunktion: Funktionswerte kollidieren nicht
  - minimale perfekte Hashfunktion: Funktionswerte sind von 0..#M-1
- Fox, Heath, Chen, Daoud. Practical minimal perfect hash functions for large databases
  - CACM, 35(1):105-121, Januar 1992
- Schmidt. GPERF – A Perfect Hash Function Generator

Hashing mit direkter Verkettung

- Behandlung von Kollisionen durch verkettete Liste

A	S	E	R	C	H	I	N	G	X	M	P	L
0	2	0	4	4	4	2	2	1	2	4	3	3



Diese Zeichnung zeigt das Ergebnis nach dem Einfügen der Schlüssel A S E R C H I N G X M P L *in eine anfangs leere Hash-Tabelle mit direkter Verkettung (ungeordnete Listen) mit den oben angegebenen Hash-Werten. Zuerst kommt das* A *in Liste 0, dann* S *in Liste 2, dann* E *in Liste 0 (am Beginn eingefügt, um die Zeit für das Einfügen konstant zu halten), danach* R *in Liste 4 usw.*

HashST.java

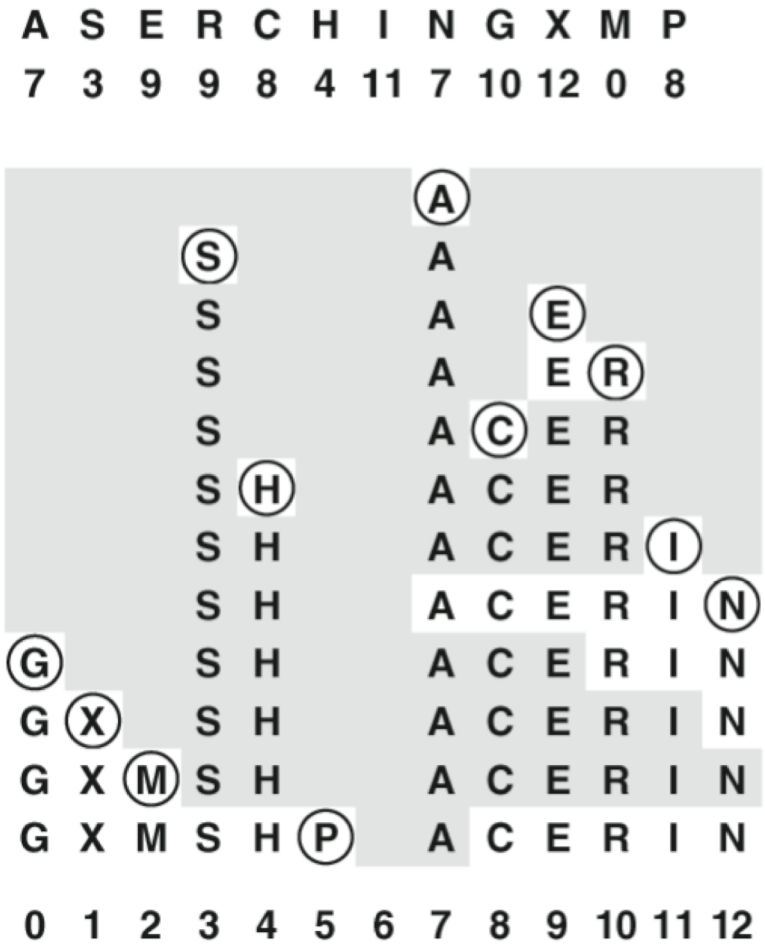
Statisches Hashing

- Hashtabelle T fester Größe (0..M-1)
- Schlüssel werden auf den Bereich 0..M-1 gehasht
- Bei Kollisionen werden Einträge mit gleichem Hashwert in verketteter Liste geführt
  - "overflow table"
- insert: Berechnung des Hashwerts h, Eintrag in T[h]
  - wahlweise am Anfang oder am Ende der verketteten Liste
- lookup: Berechnung des Hashwerts h, Suche nach Schlüssel in T[h]
  - Schlüsselvergleich in jedem Fall erforderlich, außer wenn T[h] leer ist
- Effizienz: lookup benötigt im Mittel N/M Vergleiche
  - Annahme: Hashwerte sind gleichverteilt
  - im Mittel O(1), falls N < M
    - O(1) im worst case, falls Hashfunktion perfekt

Offene Adressierung: Lineares Sondieren

- Problem des statischen Hashings: Speicherbedarf für *overflow table*
- Lösung: Hashing mit offener Adressierung (open-addressing)
  - Schlüssel hat nicht einen Hashwert, sondern viele
  - Hash-Funktionen h_0, h_1, h_2, ...
- Lineares Sondieren: Weitere Hashfunktionen ergeben sich durch lineare Verschiebung
  - $h_n(x) = (h(x)+n) \% M$
  - Von h(x) beginnend wird der nächste freie Slot gesucht
- Problem: mit steigender Tabellenbelegung werden Kollisionen wahrscheinlicher
  - Füllstand (load factor): N/M (offenes Hashing: N/M < 1)
- Problem: Clusterbildung
  - auch erfolglose Suche muss der Reihe nach alle Schlüssel testen

Lineares Sondieren



Diese Zeichnung zeigt das Einfügen der Schlüssel A S E R C H I N G X M P *in eine anfangs leere Hash-Tabelle der Größe 13 mit offener Adressierung mit den oben angegebenen Hash-Werten und Kollisionsbeseitigung durch lineares Sondieren. Zuerst kommt das* A *an Position 7, dann das* S *an Position 3 und das* E *an Position 9. Dann kommt das* R *an Position 10, nachdem eine Kollision an Position 9 aufgetreten ist, usw. Testsequenzen, die über das rechte Ende der Tabelle hinausgehen, fahren am linken Ende fort: Zum Beispiel liefert die Hashfunktion für den zuletzt eingefügten Schlüssel, das* P*, einen Hash-Wert für die Position 8, was schließlich zur Position 5 führt, nachdem Kollisionen an den Positionen 8 bis 12 und danach an 0 bis 4 aufgetreten sind. Alle nicht getesteten Tabellenpositionen sind grau unterlegt.*

Offene Adressierung: Quadratisches Sondieren

- Idee: alternative Adressen mit quadratischem Abstand von h(x)
- Verfeinerung: Abwechselnd positive und negative Offsets
  - h, h+1, h-1, h+4, h-4, h+9, h-9, ...
  - lookup mit anderem Hashwert muss nicht das ganze Cluster durchmustern
    - Etwa: h´ = h+1, durchsucht h+1, h+2, h, h+5, h-3, ...
- M prim, M = 4j+3: alternatives quadratisches Sondieren trifft letztlich alle Tabellenpositionen

Offene Adressierung: Löschen

- Problem: Suche in Hashtabelle bricht ab, wenn ein Eintrag leer ist
  - löscht man Index h_1 aus Folge h_0, h_1, h_2, dann kann h_2 nicht mehr gefunden werden
- Lösung für lineares Sondieren: Neuindizierung
  - Lösche Eintrag T[h]
  - danach Neueintragen von T[h+1], T[h+2], ... bis Null-Eintrag gefunden ist
- Quadratisches Sondieren: Liste neu zu indizierender Einträge ist schwer zu ermitteln
  - Lösung: Ersetze Eintrag durch Wächter
    - Suche wird über Wächter-Eintrag “hinwegsuchen”
    - Neueintrag kann Wächter durch gültigen Eintrag ersetzen

Dynamische Hashtabellen

- Mit wachsendem Füllstand wird Suche ineffizient
  - Übergang zu linearer Suche
  - offene Adressierung: Tabelle ist letztlich voll
- Lösung: Vergrößerung der Tabelle
  - etwa: Verdopplung (oder Vergrößerung um anderen Faktor)
  - Neuindizierung aller Einträge
    - Caching der Hashwerte?
- Wann soll Vergrößerung ausgelöst werden?
  - wenn Füllstand Grenzwert überschreitet
    - etwa: Tabelle ist halbvoll
  - wenn amortisierte Performanz sinkt
    - etwa: wenn mittlere Zahl der Vergleiche pro Operation größer als 4 wird
  - Berücksichtigung von Wächtereinträgen:
    - Verkleinerung der Tabelle, wenn Zahl der Wächtereinträge “groß” wird

Hashtabellen und Binäre Suchbäume

- Symboltabellen sind üblicherweise als Hashtabellen implementiert
  - für “gutartige” Schlüsselmengen O(1) falls Füllstand klein
  - Schlüssel müssen keiner Ordnungsrelation unterliegen
    - aber: Hashing muss unterstützt sein
    - OO: Hashwert darf sich über die Lebenszeit eines Objekts nicht ändern; gleiche Objekte müssen gleichen Hashwert liefern
- Binäre Bäume garantieren bessere Leistung im schlechten Fall
  - Bäume: O(log N); Hashtabellen können zu linearer Suche entarten
  - Verteilungseigenschaften der Hashfunktion kritisch für Suchzeiten
  - Echtzeitsysteme: Zugriffszeit muss vorhersagbar sein

Zusammenfassung

- Symboltabellen und binäre Suchbäume
  - Symboltabellen
  - Binäre Suchbäume
  - Balanzierte Bäume
  - 2-3-4-Bäume
  - Rot-Schwarz-Bäume
  - AVL-Bäume
  - Skiplisten
- Datenstrukturen – Hash-Tabellen
  - Hashfunktionen
  - String-Hashing
  - Offene Adressierung
- Hash-Tabellen und Binäre Suchbäume