Algorithmen und Datenstrukturen

Suchen

Problemstellung Suchen

- Eine Art Wörterbuch mit **langen Schlüsseln** und **großen Einträgen**:
  - *Erstellen* – Aufbau eines neuen leeren Wörterbuchs
  - *Suchen* – Auffinden eines Eintrags / Nichtfinden
  - *Einfügen* – Neuen Eintrag aufnehmen
  - *Löschen* – Entfernen eines Eintrags
  - *Vereinen* – Zwei Wörterbücher zusammenfassen
  - *Sortieren* – Alle Einträge nach Schlüsseln sortieren

Interface der Datenstruktur

```
class Dict key value coll where
  empty     :: coll key value
  search    :: key -> coll key value -> Maybe value
  add       :: key -> value -> coll key value -> coll key value
  delete    :: key -> coll key value -> coll key value

class (Dict key value coll) => DictS key value coll where
  searchM  :: (MonadState (coll key value) m) => key -> m (Maybe value)
  addM     :: (MonadState (coll key value) m) => key -> value -> m ()
  deleteM  :: (MonadState (coll key value) m) => key -> m ()
```

IKS

Doppelten Schlüsseln

- Variante 1: Verbot doppelter Schlüssel
  - Dank Programmierdisziplin werden keine erwartet
  - Beim *add* wird der Eintrag verworfen
  - Beim *add* wird der Eintrag überschrieben
- Variante 2: Mehrfacheintrag
  - Bei *search* wird irgendeine Fundstelle geliefert
  - Bei *search* werden alle Fundstellen geliefert
- Wir fordern eindeutige Schlüssel

Sequentielle Suche

- Neueinträge einfach anfügen (1 Schritt)
- Suche per Durchlauf:
  - N+1 Schritte im Fehlerfall
  - N/2 Schritte im Erfolgsfall (average)
- Löschen durch Entfernen des Elements
- Trick: Immer das Gesuchte an den Anfang stellen
  - Automatische Anpassung an Suchanfragen
  - Nur 40% langsamer als beste Alternative (average)

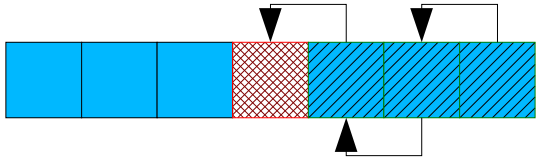
Sequentielle Suche

```
newtype SequentialSearch k v =
    SS { fromSS :: [(k,v)] }


instance Eq k => Dict k v
    SequentialSearch where

  empty    = SS []

  add k v  = SS . ((k,v):) . fromSS

  search k = lookup k . fromSS

  delete k =

      SS . filter ((k/=).fst) . fromSS
```

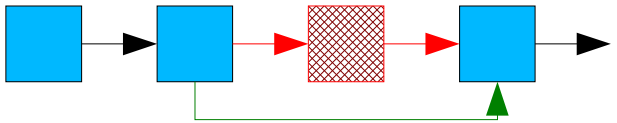
```
instance Eq k => DictM k v
    SequentialSearch where

  searchM k = do

      (SS d) <- get

      case break ((k==).fst) d of

          (_, []) -> return Nothing
          (d1,e:d2) -> do
              put $ SS (e : d1 ++ d2)
              return $ Just (snd e)
```

Feld, Liste oder Doppelt verkettet ?

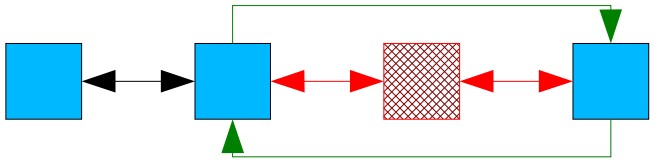
- Wie teuer ist eigentlich *delete*?
- Feld: ca. N/2 Elemente verschieben (teuer)



- Liste: Zeiger umsetzen (Suche nach Vorgänger)



- Doppelte verkettete Liste: Zeiger umsetzen



Einfach verkette Listen in C

```
struct SingleLinkedListNode {
        struct SingleLinkedListNode * next;
        int key, value; /* Datenelemente */
};

void addS(int key, int value, struct SingleLinkedNode ** start) {
        struct SingleLinkedNode * neu = malloc(sizeof(struct SingleLinkedNode));
        neu->key = key;         neu->value = value;       /* Datenwerte eintragen */
        neu->next = *start;     *start = neu;             /* Verlinkung setzen */
}
```

Anlegen mit malloc(3) und Entfernen mit free(3).

Einfach verkette Listen in C

```
int * searchS(int key, struct SingleLinkedNode ** start) {
        struct SingleLinkedNode * p;
        for(p = *start; p != NULL; p = p->next)          /* Knoten ablaufen */
                if(key == p->key)  return &(p->value);   /* Zeiger zurück */
        return NULL;                                     /* Nicht gefunden */
}
void deleteS(int key, struct SingleLinkedNode ** start) {
        struct SingleLinkedNode * p;
        for(; (p = *start) != NULL; start = &(p->next))
                if(key == p->key) {
                        *v = p->next; free(p); return;
                }
}
```

Doppelt verkettete Listen in C

```
struct DoubleLinkedNode {
        struct DoubleLinkedNode * prev, * next;
        int key, value;  /* Datenelemente */
};
void addD(int key, int value, struct DoubleLinkedNode ** start) {
        struct DoubleLinkedNode * neu = malloc(sizeof(struct DoubleLinkedNode));
        neu->key = key; neu->value = value;                          /* Datenwerte setzen */
        if(*start == NULL) {neu->next = neu; neu->prev = neu; }      /* */
        else {neu->next = *start; neu->prev = neu->next->prev;       /* */
                neu->next->prev = neu; neu->prev->next = neu; }      /* */
        *start = neu;                                                /* Als erstes Element */
}
```

IKS

Doppelt verkettete Listen in C

```
int * searchD(int key, struct DoubleLinkedNode ** start) {
        struct DoubleLinkedNode * p;
        if(*start == NULL)  return NULL;     /* Keine Elemente vorhanden */
        p = *start;                          /* Mit erstem Element beginnen */
        do {                                 /* Erstes Element immer bearbeiten */
                if(p->key == key)            /* Vergleichen */
                        return &(p->value);  /* Gefunden */
                p = p->next;                 /* Zum nächsten Element */
        } while(p != *start);                /* Abbruch, wenn wieder vorn angekommen */
        return NULL;
}
```

IKS

Doppelt verkettete Listen in C

Löschen unterscheidet sich vom Suchen durch die Aktion beim gefundenen Element:

```
if(p == p->next)          /* Letztes Element in der Liste? */
        *start = NULL;    /* Liste leeren */
else {                    /* Ausklinken aus der Liste */
        p->next->prev = p->prev;
        p->prev->next = p->next;
        if(p == *start)  *start = p->next;       /* Listenbegin evtl. verschieben */
}
free(p);
return;
```

Zyklisch doppelt verkettete Listen sind sehr häufig.

Sortierte Suche

- Problem: Fehlersuche dauert zu lange (N+1)
- Lösung: Daten werden sortiert gehalten
- Suche per Durchlauf:
  - N/2 Schritte im Fehlerfall (average)
  - N/2 Schritte im Erfolgsfall (average)
- Löschen durch Entfernen des Elements
- Einfügen durch Insertion(Sort)
  - Liste empfehlenswert, da Einfügen sonst teuer

Sortierte Suche

```
newtype SortedSearch a b = Sort { fromSort :: [(a,b)] }

instance Ord a => Dict a b SortedSearch where

    empty = Sort []

    search = doSortSearch $ \v _ _ -> v

    add k v = doSortSearch (\_ d1 d2 -> Sort (d1 ++ (k,v) : d2)) k

    delete = doSortSearch $ \_ d1 d2 -> Sort (d1 ++ d2)


doSortSearch f k (Sort d) = case break ((k<=).fst) d of

    (d1,(t,v):d2) | t == k     -> f (Just v) d1 d2

    (d1,d2)                    -> f Nothing  d1 d2
```

Binäre Suche

- Problem: Suche dauert mit (N/2) zu lange
- Lösung: Sortierte Felder(!) halbieren
- Endrekursiver Abstieg nach Vergleich mit Mitte
- Höchstens ln N + 1 Schritte für jede Suche
- Einfügen und Löschen: N/2 Verschiebungen
- Trick: Interpoliert statt x = (l+r)/2
  - Suchzeit im Mittel bei ln ln N + 1 (Praxis: < 6)
  - Gleichmäßige Schlüsselverteilung nötig

Suche im Binärbaum

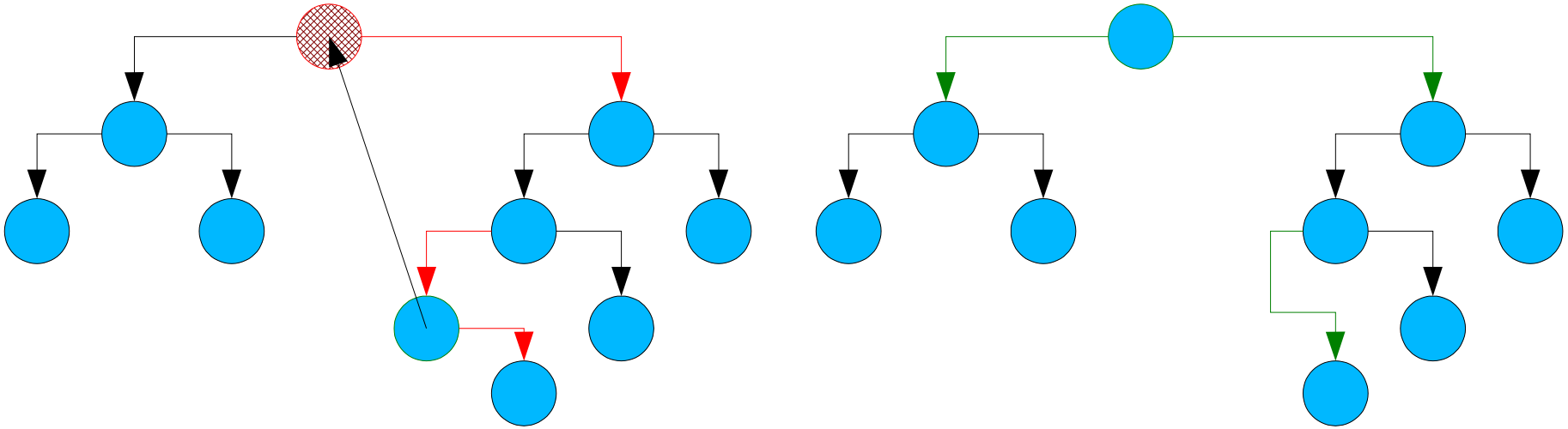
- Problem: Andere Operationen zu teuer
- Lösung: Baumstruktur statt Linearität
- Suche und Einfügen trivial:
  - Average: 2*ln N Schritte
  - Worst Case: N Schritte
- Löschen etwas schwieriger, Zeiten wie oben
- Empfindlich gegen vorsortierte Einfügungen

Binärbaum

```
data BinTree k v = BinLeaf |
   BinNode {
      key :: k,
      value :: v,
      left,right :: BinTree k v
   }


instance Ord k => Dict k v BinTree
    where
  empty = BinLeaf
```

```
search k BinLeaf = Nothing
search k n
        | k < key n = search k (left n)
        | k > key n = search k (right n)
        | otherwise = Just (value n)
add k v BinLeaf = BinNode k v BinLeaf
     BinLeaf
add k v n
        | k < key n = n { left  = add k v (left n) }
        | k > key n = n {right=add k v (right n)}
        | otherwise = n { key = k, value = v }
```

Löschen im Binärbaum



- Beobachtung: Inorder-Traversierung ist sortiert
- Ersetzung des zu löschenden Knotens durch seinen Nachfolger
- Korrektur der Zeiger

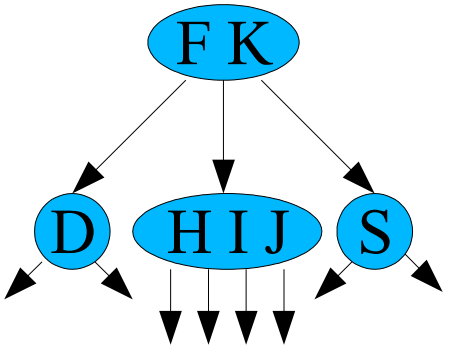
Löschen im Binärbaum

```
delete k BinLeaf = BinLeaf
delete k n| k < key n = n { left  = delete k (left n) }
          | k > key n = n { right = delete k (right n) }
          | BinLeaf == right n = left n
          | otherwise = case removeLeftmost (right n) of
                          (m, t) -> m { left = left n, right = t }

removeLeftmost n  | left n == BinLeaf = (n, right n)
                  | otherwise = case removeLeftmost (left n) of
                                (m, t) -> (m, n { left = t } )
```

Ausgeglichene Bäume

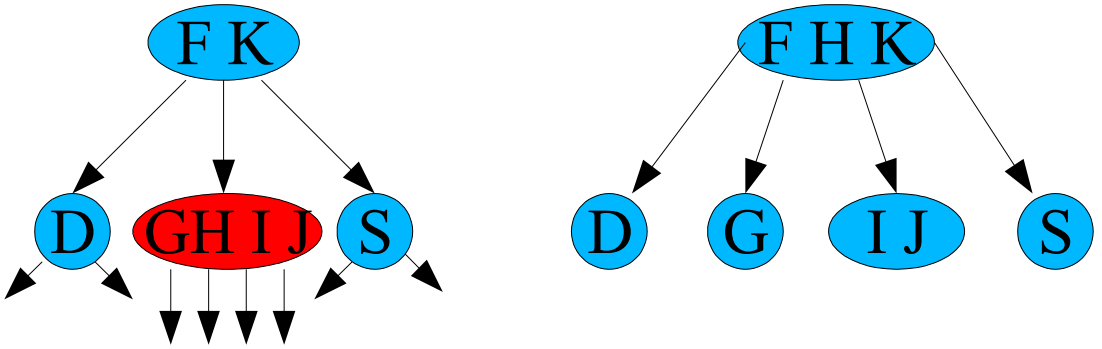
- Problem: Worst Case ist linear
- Lösung: Ausgleich der Bäume erzwingen
- Zusatzkosten beim Einfügen und Löschen
- Einfaches Modell ist der 2-3-4 Baum
- Suche sollte hinterher wie im Binärbaum gehen
- Worst Case für alle Opertionen: O(ln N)
- Meist nicht mehr so schnell im Mittel

2-3-4 Bäume



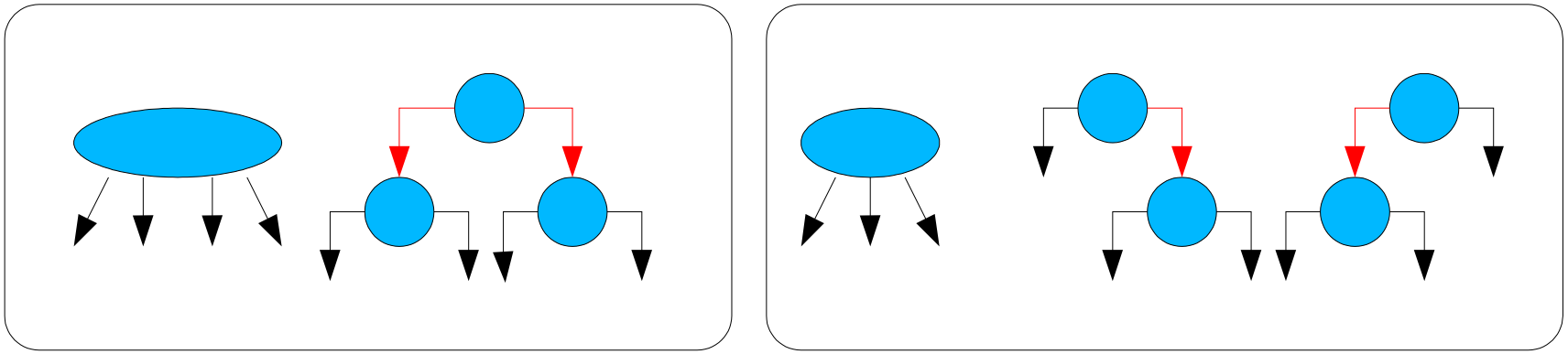
- Statt zwei Verzweigungen auch drei oder vier
- Flexibilität beim Umbau des Baumes
- Einfügen vergrößert Knoten, Löschen verkleinert
- Höhe des Baumes ändert sich nur bei Überläufen

2-3-4 Bäume



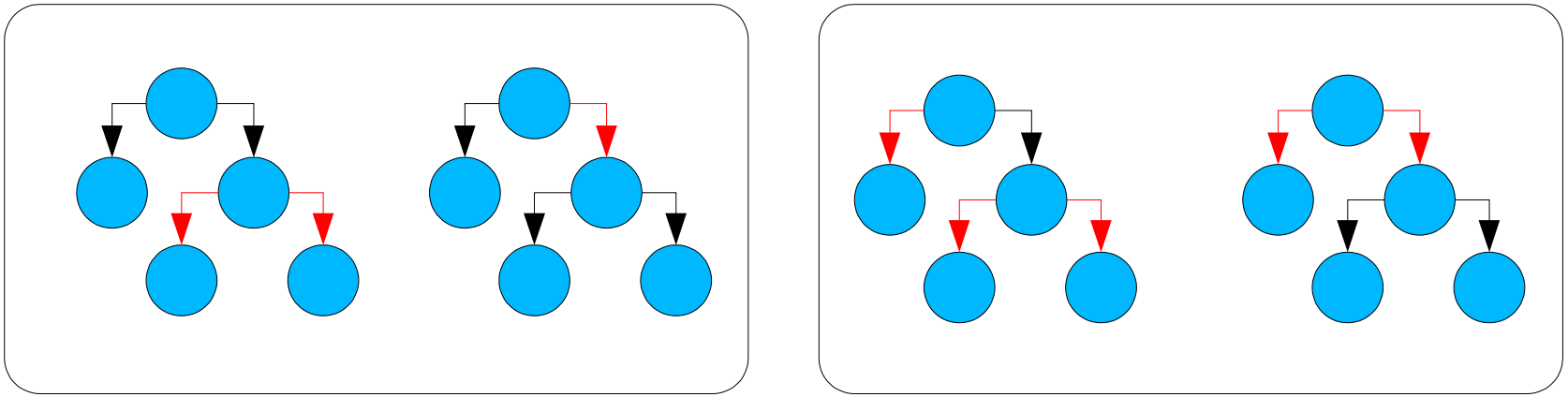
- Einfügen von „G“ zerstört Baumknoten
- Spalten und Verschieben nach oben
- Wenn Viererknoten übereinander, weitermachen
- Überlauf oben: Baum wächst in die Höhe

Rot-Schwarz Bäume



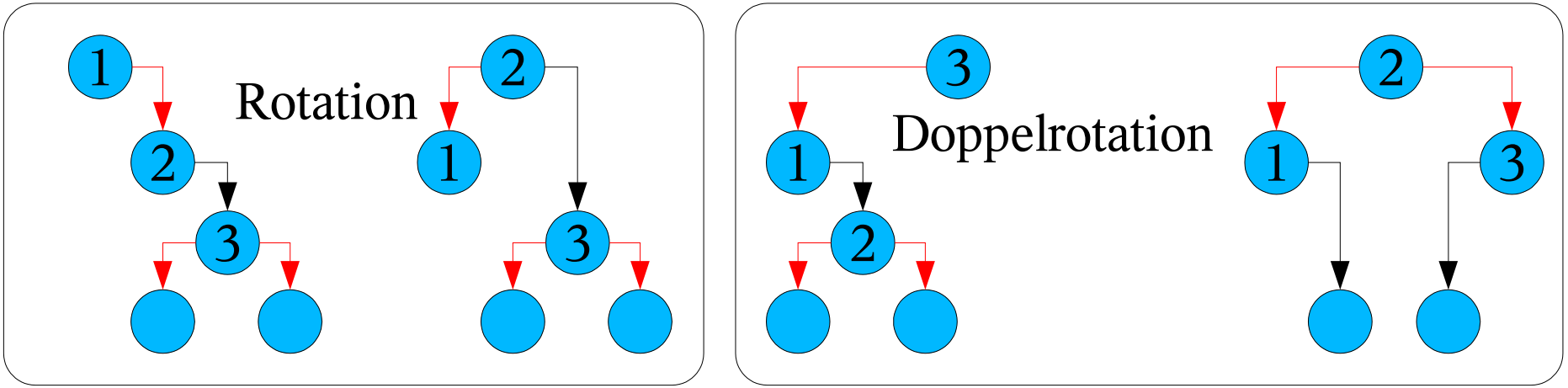
- Implementation von 2-3-4 Bäumen als Binäre
- Farbbit zur Kennzeichnung der Großknoten
- Baumtiefe doppelt zu groß wie beim 2-3-4 Baum
- Wahlfreiheit für Darstellung der 3-Knoten

Rot-Schwarz Bäume



- Aufspaltung von 4-Knoten durch Umfärben
- Umfärbung ist sehr billig, deswegen 4-Knoten immer aufspalten
- Richtige Ausrichtung des 3-Knotens nötig

Rot-Schwarz Bäume

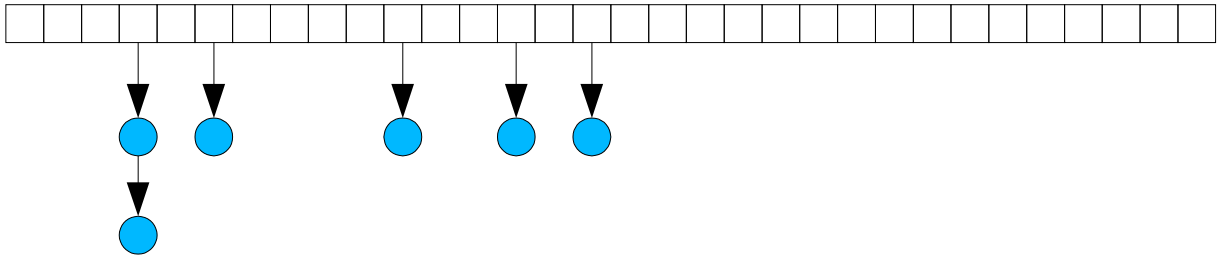


- Bei falsche Orientierung der 3-Knoten
- Einfache Rotation: 2 Knoten vertauschen
- Doppelte Rotation: 3 Knoten vertauschen
- Ergebnis ist ausgeglichen

Hashing

- Problem: Suche zu langsam
- Lösung: Zerhacken des Problem durch Rechnen
- Hashfunktion :: Key -> Hashwert
  Beispielsweise: Modulo Primzahl
- Hashwerte sind deutlich kleiner als Keys und uniform verteilt: Als Index in ein Feld benutzen
- Ideal: Operationen in O(1) = konstante Zeit
- Probleme: Kollisionen, Platzverschwendung

Hashing mit getrennter Verkettung



- Feld enthält Anfang einer Suchliste
- Einträge in der Suchliste nur wenige Elemente
- Zusätzlicher Platz für Listenverkettung nötig

Lineares Austesten

- Da meist nur ein Datum pro Hashwert vorhanden
  => Daten direkt in die Hashtabelle schreiben
- Bei Kollision nächsten freien Platz suchen
- Bei Suche bis zur Leerstelle suchen, dann Fehler
- Füllstand 80% => Erfolg: < 3 Tests, Fehler: < 10
- Löschen möglich mit Überprüfen der Folgewerte
- Suchzeiten steigen mit Füllstand schnell an
  Grund: Clusterbildung in der Hashtabelle

Doppeltes Hashing

- Clusterbildung vermeiden durch große Schritte
- Schrittweite muß relativ prim sein und pro Schlüssel unterschiedlich
- Zweite Hashfunktion benutzen, erster Hash prim
- Füllstand 80% => Erfolg: < 2 Tests, Fehler: < 5
- Füllstand 90% => Erfolg: < 5 Tests, Fehler: < 10
- Löschen nicht möglich: Regelmäßiger Neubau
- Brents Variation: Erfolg ist garantiert O(1)

Digitale Suchbäume

- Freier Binärbaum auf Basis von Bittests
- Levelorder: Gleiche Ebene = Gleiches Bit
- Bei Suche: Nur auf Gleichheit oder ein Bit nötig
- Vergleiche: ln N (Average), b (Worst)
- Baum ist unausgeglichen, abhängig von Schlüsselverteilung und Reihenfolge

Tries

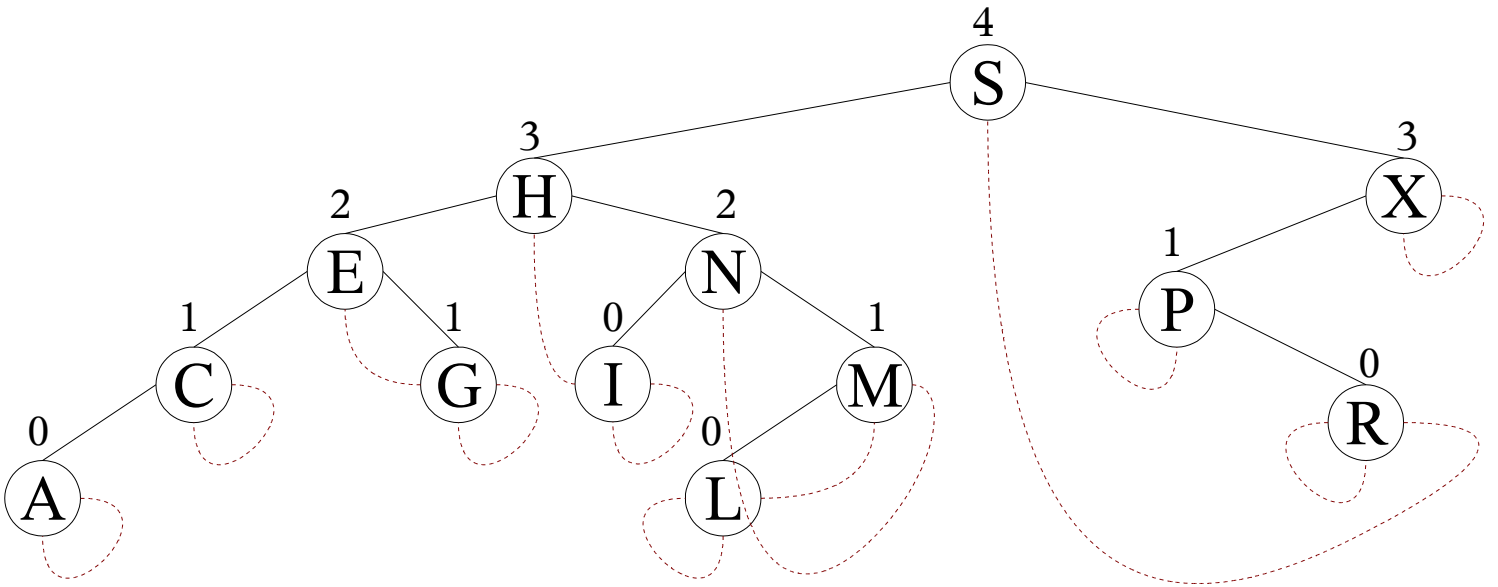
- Problem: Sehr lange Schlüssel
- Lösung: Verzicht auf Schlüssel im Knoten
- Daten nur noch im Blattknoten
- data Trie k v = TrieNode {left,right :: Trie k v}
  | TrieLeaf {key :: k, value :: v}
  | TrieEmpty
- Suchverzweigung anhand der Bits (je Level)
- Problem: Viele Knoten nur Linkliste in die Tiefe

Mehrwege-Tries

- Mehrere Bits zusammenfassen: Zeichenweise
- Verkettung von Hashs = Zeichen an Position x
- Sehr schnelle Suche: $\log_M N$
- Riesige Platzverschwendung
- Interessant als initiale Aufteilung in Klassen

Patricia

Practical Algorithm To Retrieve Information Coded In Alphanumeric



- Weglassen der Einwegverzweigungen im Trie
- Nur noch minimal notwenige Unterschiede testen
- Absteigende Bits, Rücklinks erkennbar

Patricia

- Testet die Unterschiede der bekannten Schlüssel
- Voller Schlüsselvergleich am Ende
- Einfügen an der ersten Bitstelle, die sich von der Fundstelle (und nur der) unterscheidet
- N Schlüssel: N Knoten und ln N Bitvergleiche
- Schnell, da nur wenige Bits zur Suche angeschaut

➔ Quintessenz des digitalen Suchens