5. Randomized Data Structures

- Deterministische Datenstrukturen:
  Bisher war alles deterministisch, Verhalten für identische Eingaben immer gleich
- Randomisierte Datenstrukturen:
  Nun hängt Verhalten auch von zufälligen Entscheidungen der Datenstruktur ab

Skip Lists

Idee

- Zweidimensionale Datenstruktur
- Füge rekursiv Express-Listen ein
- Diese haben weniger Elemente als die ursprüngliche Liste

Suche mittels Express-Listen

Beginne in Express-Liste:

- Wenn Element gefunden, ausgeben
- Wenn nächstes Element kleiner-gleich gesuchtem Element, weiter nach rechts
- Wenn nächstes Element in Express-Liste größer als gesuchtes Element, nach unten

Implementierung

- `L.head`: erstes/oberstes Element der Liste
- `L.height`: Höhe der Skiplist
- `x.key`: Wert
- `x.next`: Nachfolger
- `x.prev`: Vorgänger
- `x.down`: Nachfolger Liste unten
- `x.up`: Nachfolger Liste oben
- `nil`: kein Nachfolger / leeres Element

Suchalgorithmus

```
search(L,k)
        current=L.head;
        WHILE current != nil DO
                IF current.key == k THEN return current;
```

```
            IF current.next != nil AND current.next.key <= k
            THEN current=current.next
            ELSE current=current.down;
    return nil;
```

Laufzeit hängt von Expresslisten ab

Auswahl der Elemente für Express-Listen

Wähle jedes Element aus Liste mit Wahrscheinlichkeit p (z.B. $p = \frac{1}{2}$) für übergeordnete Liste
Durchschnittliche Höhe $h \in O(\log_{\frac{1}{p}} n)$

Durchschnittliche Laufzeit für Suchen

Im schlimmsten Fall wird Suche erst in unterster Liste beendet
Wenn Skip-Liste Höhe h, braucht man im Durchschnitt $\frac{h}{p} \in O(h) = O(\log n)$ viele Schritte
$\rightsquigarrow$ Durchschnittliche Laufzeit = $O(h)$

Einfügen

- Füge auf unterster Ebene ein und dann eventuell auf Ebenen darüber
- Zufällige Wahl mit Wahrscheinlichkeit p auf jeder Ebene
- Durchschnittliche Laufzeit = $O(h)$

Löschen

- Entferne Vorkommen des Elements auf allen Ebenen
- Durchschnittliche Laufzeit = $O(h)$

Laufzeiten und Speicherbedarf

- Einfügen $\Theta(\log_{\frac{1}{p}} n)$
- Löschen $\Theta(\log_{\frac{1}{p}} n)$
- Suchen $\Theta(\log_{\frac{1}{p}} n)$
- O-Notation versteckt Faktor $\frac{1}{p}$
- Speicherbedarf im Durchschnitt: $\frac{n}{1-p}$

Anwendung

Einfügen/Löschen unterstützen parallele Verarbeitung (z.B. Multi-Core-Systeme), da nur sehr lokale Änderungen
Bäume mit Re-Balacierung können dies nicht
Dafür logarithmische Laufzeit nur im Durchschnitt, also nicht garantiert

Hash Tables

Idee

$h : \text{Datenmenge} \to [0, T.length - 1]$ ist uniform und unabhängig verteilt
Datum x wird auf Arrayeintrag $h(x)$ in Hashtabelle/Array `T[]` abgebildet und dann dort gespeichert
Suche: ist `x` in `T[h(x)]` vorhanden?
Löschen: Lösche `x` aus `T[h(x)]`
Einfügen, Suchen, Löschen mit konstant vielen Array-Operationen

Kollisionsauflösung

- Wenn Array-Eintrag schon belegt, bilde verkettete Liste und füge neues Element vorne ein
- Es gibt weitere Arten der Kollisionsauflösung

Hash Tables mit verketteten Listen

- Einfügen immer noch konstante Anzahl Array-/Listen-Operationen
- Suchen/Löschen benötigen so viele Schritte, wie jeweilige Liste lang ist
- Wenn Hashfunktion uniform verteilt, dann hat jede Liste im Erwartungswert $\frac{n}{T.length}$ viele Einträge

Laufzeit

Bei uniform und unabhängig verteilten Hashwerten benötigen Suchen und Löschen im Durchschnitt $\Theta(\frac{n}{T.length})$ viele Schritte.
Einfügen benötigt im Worst-Case $\Theta(1)$ viele Schritte.
Wählt man $T.length \approx n$, ergibt sich im Durchschnitt konstante Laufzeit.

Gute Hash-Funktionen

"Universelle" Hash-Funktion

- Interpretiere (Binär-)Daten als Zahlen zwischen 0 und $p-1$, p ist prim, $p >> T.length$
- Wähle zufällige $a, b \in [0, p-1], a \neq 0$, setze
  $h_{a,b}(x) := ((a \cdot x + b) \mod p) \mod T.length$
- Verteilung und Unabhängigkeit/Kollisionsresistenz gewährleistet
  Kryptographische Hash-Funktionen wie MD5, SHA1: $\{0,1\}^* \to \{0,1\}^{160}$
  MD5, SHA1 nicht sicher, besser SHA2/SHA3 verwenden
  Setze $h(x) = MD5(x) \mod T.length$

Anwendungen

z.B. MySQL

Hash Tables vs. Bäume

- Hash Table:
  - Nur Suche nach bestimmten Wert möglich
  - In der Regel Hashtable größer als zu erwartende Anzahl Einträge
- Baum:
  - Schnelles Traversieren möglich (z.B. nächstkleinerer Wert), auch Bereichssuche

Laufzeiten, Speicherbedarf

- Einfügen: $\Theta(1)$ im Worst-Case
- Löschen: $\Theta(1)$ im Durchschnitt
- Suchen: $\Theta(1)$ im Durchschnitt
- Speicherbedarf in der Regel größer als n, üblicherweise ca. $1,33 \cdot n$

Bloom-Filter

Speicherschonende Wörterbücher mit kleinem Fehler

Beispiel: Schlechte Passwörter vermeiden

1. Speichere schlechte Passwörter in Bloom-Filter
2. Prüfe, ob eingegebenes Passwort im Bloom-Filter ist
   Starke Passwörter, die fälschlicherweise dem Wörterbuch zugeordnet werden, sind ärgerlich, aber nicht sehr schlimm

Anwendungen

- NoSQL-Datenbanken: Abfragen für nicht-vorhandene Elemente verhindern
- Bitcoin: Prüfen von Transaktionen ohne gesamte Daten zu laden
- Früher auch Chrome-Browser: Erkennen schädlicher Webseiten

Erstellen

Gegeben:

- n Elemente $x_0, x_1, \ldots, x_{n-1}$ beliebiger Komplexität
- m Bits speicher, üblicherweise in einem Bit-Array
- k "gute" Hash-Funktionen $H_0, \ldots, H_k - 1$ mit Bildbereich $0, 1, \ldots, m - 1$
- Empfohlene Wahl: $k = \frac{m}{n} \cdot \ln 2$ ergibt Fehlerrate von ca. 2^{-k}, üblicherweise $k \in [5, 20]$

```
initBloom(X,BF,H) // H array of functions H[j], BF bit-array, X array of objects
        FOR i=0 TO BF.length-1 DO BF[i]=0; // initialise array with 0-entries
        FOR i=0 TO X.length-1 DO
                FOR j=0 TO H.length-1 DO
                        BF[H[j](X[i])]=1;
```

- Schreibe für jedes Element in jede Bit-Position $H_0(x_i), \ldots, H_{k-1}(x_i)$ eine 1
- Eventuell werden dabei Einträge mehrmals auf 1 gesetzt

Suchen

```
searchBloom(BF,H,y) // H array of functions H[j]
        result=1;
        FOR j=0 TO H.length-1 DO
                result=result AND BF[H[j](y)];
        return result;
```

- Gibt an, dass `y` im Wörterbuch ist, gdw. alle k Einträge für `y` in `BF==1` sind
- Wenn `y` nicht im Wörterbuch, kann Algorithmus eventuell trotzdem 1 zurückgeben
- Daher "gute" Hash-Funktionen und Größe des Filters nicht zu klein wählen
- Keine false negatives, nur false positives
- Wenn BF nur bis zur Hälfte mit 1en gefüllt und Hash-Funktionen uniforme und unabhängige Werte liefern, dann Fehler $\leq 2^{-k}$

Beispielrechnung

$n = 100.000$ Passwörter, je 10 ASCII-Zeichen

- Baumstruktur:
  - Speicherbedarf: 8.000.000 Bits + Baumstruktur
  - Suchen: ca. $\log_2 100.000 \approx 17$ Elemente betrachten
- Bloom-Filter mit $k = 7, m = k \cdot n \cdot \ln 2$
  - Speicherbedarf: ca. 1.000.000 Bits
  - Suchen: $k = 7$ Mal hashen und $k = 7$ Array-Zugriffe