Algorithmen und Datenstrukturen

Ulrich Köthe

29. Mai 2008

Inhaltsverzeichnis

0.1 weiteres Beispiel zu Heaps

1. Heapbedingung (MaxHeap): Wurzel hat größere Priorität als Kinder (MinHeap heißt kleinste Priorität oben)

2. Upheap: repariert den Heap (Heap-Bedingung) falls Element k eine zu hohe Priorität hat (d.h. vertauschen mit Parent)

3. Downheap: repariert den Heap falls Element k eine zu niedrige Priorität hat (d.h. vertauschen mit mit Kind der höchsten Priorität)

4. Komplexität O(log n) (einfügen und Element höchster Priorität entfernen)

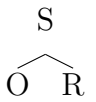
0.1.1 Beispiel am Wort "SORTING"

Anmerkung: Priorität entspricht der Reihenfolge im Alphabet, d.h. A hat die kleinste und Z die größte Priorität
Anfang, S eingefüen):

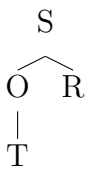
S

O einfügen:

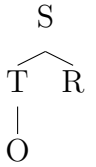
R einfügen:



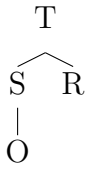
T einfügen:



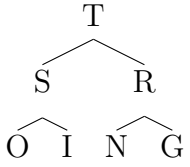
T sitzt jedoch zu tief, darum up-heap:



und nochmal upheap:



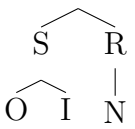
I, N, G einfügen:



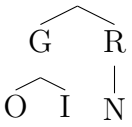
Elemente entfernen:

Wurzel entfernen, Element mit geringster Warscheinlichkeit nach oben setzen, Heap wieder herstellen

$\not{T}$



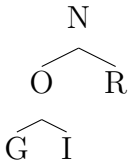
(Heapbedingung nicht ergefüllt)



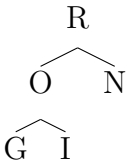
(nach down heap)

(down heap)

$\not{S}$



(gleiche Vorgehensweise wie oben)



(down heap)

0.2 andere Heapvarianten:

- Min-Max-Priority Queue ("Deap", Double-Ended-Heap)

- Binomialer Heap, effiziente Operation “merge Heap” O(n log (iN_1+N_2)
  (Beweis durch binomial Koeffizienten, Zusammenführen zweier Prioritätslisten)
- Fibonacci-Heap: Einfügen in amortisierter Zeit O(1)
  (Beweis durch Fibonacci Zahlen)

1 Assotiative Arrays

- ähnlich Arrays X = a[i], a[i] = x, wobei der Typ von i beliebig sein kann
  (beim Array: i $\in$ 0...N-1
- z.B. Wörterbuch x = toEnglish[word], typ[word] = = string (word ist in diesem Fall ja i)
  (häufigster Fall String als indices)
- offensichtliche Realisierung: zurückführen auf Suchen
  Datenstruktur muss so erweitert werden, dass mehr Informationen gespeichert werden können, z.B. erweiterte Node-Klasse um Feld 'data'
  node: key, node: data
  verwende sequentielle Suche oder Baumsuche um die beiden Zugriffsoperatoren zu implementieren (a[i]
  Sequentielle Suche a[i] $\in$ O (n)
  Baumsuche a[i] $\in$ O (log n))
- Bsp. für Realisierung mit Baumsuche, std-map in C++

Es stellt sich die Frage, ob das auch schneller, d.h. O(1) geht. - Ja, mit einer Hashtabelle bzw. Hashing

- gegeben: Schlüssel aus Universum U, U = Menge allem was als Schlüssel legal ist
  Hashfunktion: $f : u \to 0 \dots M-1 \subset N$
- Definition einer Kollision, wenn $f(u_1 \in U) = f(u_2 \in U)$ d.h. zwei Schlüssel zeigen auf die selbe Zahl
  Eigenschaft: Falls $|U| > M$ sind Kollisionen unvermeidlich
- Aufgabe: Entwerfen einer Hash-Funktion mit möglichst wenigen Kollisionen
  $\to$ Hashfunktion ähnlich eines Zufallszahlengenerators
  jede $k \in 0 \cdots M-1$ soll mit gleicher Wahrscheinlichkeit herauskommen
  (= uniformes Hashing)

- Aber: für jede Hashfunktion gibt es ungünstige Konstellationoen ($U_f \subset Uf(U_f)$) hat viele Kollisionen.
  Bsp: Hashfunktion Geburtstage → Menschen. Bei einer Party, zu der nur Leute eingeladen werden, die an einem 8ten Geburtstag haben, steigt die Wahrscheinlichkeit, dass zwei Leute am selben Tag Geburtstag haben, also einer Kollisioni, stark. So werden ja überhaupt nur 12 von 365 (bzw. 366) Möglichkeiten betrachtet.
  ⇒ Wahl einer guten Hashfunktion ist eine "Kunst"
  ⇒ Analyse der Daten für ein gutes f

- falls man die erlaubten Schlüssel schon kennt:
  $U_f \subset U$ kommt nur $U_f vor$ → "perfekte" Hashfunktion

1.1 Beispiel: Monatsnamen

U ist Strings der Länge 9 → 30^9 mögliche Strings, 12 werden Benutzt
U_f ={"Januar", "Februar", ..., "Dezember"}

- Hashfunktion Anfangsbuchstabe des Monatsnamen:
  5bit M = 32 (2^5 → J,F,M,A,M,J,J,A,S,N,D → offensichtliche Kollisionen bei J, M und A (⇒ keine gute Hashfunktion)

- Hashfunktion erste 3 Buchstaben → 15bit → M = 2^{15}
  ⇒ Jan, Feb, Mär, Apr, Mai, Jun, Jul, Aug, Sep, Nov, Dez
  ⇒ Keine Kollision, aber M ist groß

Aufgabe: ⇒ minimale, perfekte Hashfunktion $|U_f| = M$ finden ⇒ siehe bungsaufgabe in 3 Wochen (Algortihmus beruht auf Graphentheorie)

- universelles Hashing: wähle die Hashfunktion per Zufallszahl ⇒ Wahrscheinlichkeit, dass die Hashfunktion für die Schlüssel ungünstig ist, wird minimiert

- Cryptographische Hash-Funktionen wie z.B. md5, SHA1 u.a. für Verschlüsselung, verschlüsselte Kommunikation

- Anforderungen:
  - Kollision sehr unwahrscheinlich
  - Hash darf nicht invertierbar sein
    ⇒ M sehr groß (z.B. M = 2^{128})

1.2 Beliebte Hashfunktionen

- Assotiatives Array M ist festgelegt (Größe des Arrays)
  $f(U) = f'(U)\%M \Rightarrow \in 0 \ldots M-1$
  $f'(U)$ falls U unsigned int (32bit int Datentyp)
  U signed int $\Rightarrow |U| f'(U) = U$ oder Typkonvertierung (in C: (unsigned int) u)

- andere Schlüsseltypen interpretiert man als Array of Byte $f'(U)$ konvertiert Array of Byte $\rightarrow$ unsigned int

- Beliebte Beispiele: Bernsteinfunktion

```
def bHash(u): #U: Array of Byte
        h = 0
        for k in U:
                k = 33 * h + k          (*)
        return h
```

- modifiziertes Bernsteinhash: wie oben (*) h = 33 * h $\hat{}$ k # $\hat{=}$ bitweises xor

- Shift-Add-Xor-Funktion:

```
def saxHash(U):
        h=0
        for k in U:
                h ^= (h << 5) + (h >> 2) + k
        return h
```

- Fowler/Novel/Vo-Funktion:

```
def FNVHash(U):
        h = 2166136261
        for k in U:
                h = (h * 16777619) ^ k
        return h
```

- ...

$\rightarrow$ Hashtabelle: Kollision unvermeidlich wenn Schlüssel nicht bekannt

- besteht aus einer Hashfunktion + Array der größe M

- Insert (naiv):

```
def insert(u):
        index = hash(u) % M
        array[index] = u
```

→ funktioniert so nicht, weil Kollisionen zum überschreiben führen

1.3 geschickte Behandlung von Kollision: Zwei Ansätze:

- lineare Verkettung
- offene Adressierung

1.3.1 lineare Verkettung:

- jedes Arrayfeld kann dadurch beliebig viele Datenelemente speichern

```
def insert(u):
        index = hash(u) % M
        array[index].append(u)
                #insert() O(1)

def get(u):
        index = hash(u) % M                          # O (1)
        k = sequentialsearch(array[index],u)         # O (a)
                                                     # ges. O(1-
        if k == -1:
                return none
        else:
                return Array[index][k]
```

Füllstand $\alpha = \frac{N}{M}$ wobei N die Gre der Hashtabelle und M die GGre des Arrays ist
Wenn alles klappt
bei uniformen Hashing ist die Länge der Listen $O(\alpha)$ bei gleichmäßiger Verteilung alle Listen $\frac{N}{M}$ lang
Bsp: C++ std.hashmap

⇒ nächste Woche offene Addressierung