

# Aufgaben und Lösungen

## Aufgabe 1 - UNION-FIND

8x vorgekommen

Beschreiben Sie jeweils eine Lösung für das Union-Find-Problem mit Laufzeit

1.  $O(\log n)$  (amortisiert) für UNION und  $O(1)$  für FIND
2.  $O(1)$  für UNION und  $O(\log n)$  für FIND

wobei  $n$  die Anzahl der Elemente ist. Begründen Sie in beiden Fällen die entsprechenden Laufzeiten.

### Lösung

#### $O(\log n)$ (amortisiert) für UNION und $O(1)$ für FIND

Feld name[1...n]: name[x] = Name des Blocks der x enthält.  $1 \leq x \leq n$

size[1..n]: size[A] = Anzahl Elemente im Block A, initialisiert mit 1

L[1..n]: L[A] = Liste aller Elemente in Block A, initialisiert  $L[i] = \{i\}$

**Initialisierung:**

```
begin
  for i := 1 to n do
    name[i] = i
    size[i] = 1
    L[i] = {i}
  end
end
```

**FIND(x):**

```
begin
  return name[x]
end
```

**UNION(A,B):**

```
begin
  if size[A] ≤ size[B] then
    foreach i in L[A] do
      name[i] = B
    end
    size[B] += size[A]
    L[B] = L[B].concat(L[A])
  else
    foreach i in L[B] do
      name[i] = A
    end
    size[A] += size[B]
    L[A] = L[A].concat(L[B])
  end
end
```

**Laufzeit:** FIND(x):  $O(1) \rightarrow$  Einfacher Zugriff auf ein Feld  
 UNION:  $O(\log n) \rightarrow$  x kann maximal  $\log(n)$  mal seinen Namen ändern, da es sich nach jeder Namensänderung in einer doppelt so großen Menge befindet.  
 (Die kleinere Menge wird umbenannt)

### **$O(1)$ für UNION und $O(\log n)$ für FIND**

Feld name[1...n]: name[x] = Name des Blocks mit Wurzel x (hat nur Bedeutung, falls x Wurzel)  
 Feld vater[1...n]:  $\text{vater}[x] = \begin{cases} \text{Vater von } x \text{ in seinem Baum} \\ 0, \text{ falls } x \text{ Wurzel} \end{cases}$   
 Feld wurzel[1...n]: wurzel[x] = Wurzel des Blocks mit Namen x  
 Feld size[1..n]: size[x] = Anzahl Knoten im Unterbaum mit Wurzel x

#### **Initialisierung:**

```
begin
  for i := 1 to n do
    vater[i] = 0
    name[i] = i
    wurzel[i] = i
  end
end
```

#### **FIND(x):**

```
begin
  while vater[x] != 0 do
    x = vater[x]
  end
  return name[x]
end
```

#### **UNION(A,B,C):**

```
begin
  r1 = wurzel[A]
  r2 = wurzel[B]
  if size[r1] < size[r2] then
    vater[r1] = r2
    name[r2] = C
    wurzel[C] = r2
    size[r2] += size[r1]
  else
    vater[r2] = r1
    name[r1] = C
    wurzel[C] = r1
    size[r1] += size[r2]
  end
end
```

#### **Laufzeit:**

FIND(x):  $O(\log n) \rightarrow$  Weighted UNION Rule  
 UNION:  $O(1) \rightarrow$  Nur Pointer ändern

**Warum hat der Baum logarithmische Höhe/Tiefe?** Im Worst-Case wird ein UNION auf zwei gleich große und gleich tiefe Bäume ausgeführt. Dabei ist die Größe von C doppelt so groß wie die ursprünglichen Bäume, jedoch ist die Tiefe nur um 1 gewachsen ( $\log(\text{size}(x)) \geq \text{Hoehe}(x)$ )

## Aufgabe 2 - Hashing

8x vorgekommen

Entwickeln Sie eine Datenstruktur zur Speicherung von  $n$  Schlüsseln aus dem Universum  $\{1, \dots, N\}$  (wobei  $n \ll N$ ), die eine Zugriffszeit von  $O(1)$  garantiert. Sie dürfen dabei  $O(n^2)$  Speicherplatz verwenden.

5x vorgekommen

(Perfektes Hashing) Verbessern Sie die Datenstruktur aus Aufgabe , so dass nur noch Speicherplatz  $O(n)$  benutzt wird.

Hashig durch Verkettung und mit offener Adressierung (Linear Probing: Wie funktioniert Delete())

## Lösung

### Hashing mit Verkettung

löse Kollisionen nicht auf, speichere mehrere Schlüssel an der gleichen Position

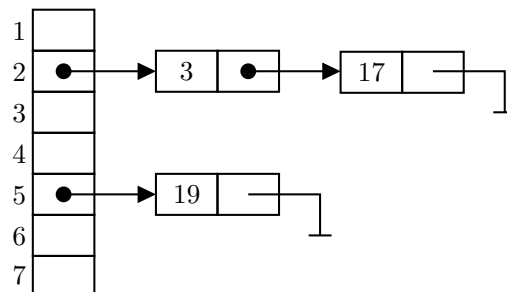
Speichere für jedes Ergebnis der Hashfunktion  $h$  eine Liste

**Lookup(x):** lineare Suche in Liste  $T[h(x)]$

- Worst Case: alle Keys in derselben List  $\rightarrow O(n)$
- erwartete Zeit:  $O(\frac{n}{m})$
- Belegungsfactor  $\beta = \frac{n}{m} \leftarrow$  erw. Länge einer Liste  $T[x]$
- wenn  $m \geq n$ , d.h.  $\beta \leq 1$  dann  $\rightarrow$  erw. Laufzeit  $O(1)$

**Insert(x):**  $x \notin S$ . Füge x an erst freie Stelle in  $T[h(x)]$  ein

**Delete(x):** Entferne x aus  $T[h(x)]$



meist wird als Hashfunktion einfaches Modulo verwendet.

### Verbesserung Verdopplungs-Strategie:

- Immer wenn  $\beta > 2$ , verdopple Tafelgröße  $\rightarrow$  1 sehr teures Insert (da alle Elemente mit neuer Hashfunktion umgespeichert werden), im Schnitt aber weiter  $O(1)$
- Bei Delete und kleinem  $\beta$ : Tabelle kann halbiert werden  $\rightarrow$  Ein sehr teures Delete, im Schnitt aber weiter  $O(1)$

### Zusatzaufgabe: Perfektes Hashing

## Aaaaalte Aufgabe 5 - Maximales Matching

Erklären Sie die Grundidee des in der Vorlesung behandelten Algorithmus zur Berechnung eines maximalen Matchings eines bipartiten Graphen

## Randomisierter Suchbaum

### Definition: Randomized Search Tree (RST)

Sei  $S = \{x_1, \dots, x_n\}$  eine Menge von  $n$  Schlüsseln. Jedem  $x_i$  wird eine zusätzlich eine Zufallszahl (auch Priorität genannt)  $prio(x_i)$  zugeordnet.  $prio(x_i)$  sind gleichverteilte reelle Zufallszahlen  $\in [0, 1]$  (Implementierung wären int-Zahlen, zB 32-bit).

Ein RST für  $S$  ist ein binärer Suchbaum für die Paare  $(x_i, prio(x_i))$ ,  $1 \leq i \leq n$ , sodass

1. normaler Knoten-orientierter Suchbaum für die Schlüssel  $x_1, \dots, x_n$
2. Maximumsheap bzgl der Prioritäten. d.h.  $prio(v) \geq prio(u)$ , falls  $v$  Parent. (( $u, v$ ) sind Knoten in einem Baum).  $\Rightarrow$  Wurzel enthält maximale Priorität.

**Existenz** durch Algorithmus zum Aufbau (rekursiv).

- Wurzel enthält  $(x_i, p_i)$  mit  $p_i = prio(x_i)$  maximal
- Linker Unterbaum: RST für  $\{(x_i, p_i) | x_j < x_i\}$
- Rechter Unterbaum: RST für  $\{(x_k, p_k) | x_k > x_i\}$

Beispiel:  $S = \{1, \dots, 10\}$

- Schreibe Tabelle mit Prioritäten und Werten.
- Teile die Tabelle beim Maximum und schreibe es in die Wurzel. Wiederhole, bis alle Elemente geschrieben.

$\Rightarrow$  Wenn sich die Prioritäten genauso oder umgekehrt, wie die Schlüssel verhalten, erhält man einen degenerierten Baum. (bzgl  $\leq$ ). zB  $prio(x_i) = x_i$ . Dieser Fall ist sehr unwahrscheinlich, wenn sich bei der Priorität um gleichverteilte Zufallszahlen handelt.

## Operationen

- **Lookup(x)**: normale suche in binärem Baum. Kosten  $\mathcal{O}(\text{Höhe}(T))$
- **Insert(x)**: Füge einen neuen Knoten  $v$  als Blatt  $(x, \text{prio}(x))$  gemäß des Schlüssels in den binären Baum ein, wobei  $\text{prio}(x)$  neue Zufallszahl (kann die Prio-Ordnung zerstören). Dann: Rotiere  $v$  nach oben, bis die Heap-Eigenschaft gilt, also  $\text{prio}(v) \leq \text{prio}(\text{parent}(v))$ .  
 Kosten:  $\mathcal{O}(\# \text{Rotationen}) = \mathcal{O}(\text{Höhe}(T))$ . Alternativ: normales einfügen in binären Baum in absteigender Reihenfolge der Prioritäten.
- **DELETE(x)**: Sei  $v$  der knoten mit Schlüssel  $x$  ( $v = \text{Lookup}(x)$ ).
  1. Rotiere  $v$  nach unten, bis  $v$  ein Blatt ist.  $R$  = linkes Rückgrat des rechten Unterbaums von  $v$ .  $L$  = rechtes Rückgrat des linken Unterbaums.
  2. Entferne das Blatt.
 Kosten:  $\mathcal{O}(\# \text{Rotationen}) = \mathcal{O}(1 + |L| + |R|)$
- **Split(y)**  $\rightarrow S_1 = \{x \in S | x \leq y\}, S_2 = \{x \in S | x \geq y\}$  (Teile den Baum, indem  $y$  mit maximaler Priorität zur Wurzel rotiert wird)
  1. Insert( $y + \epsilon$ ) mit Priorität  $\infty$
  2. Entferne die Wurzel
- **Join( $T_1, T_2$ )**:  $S \leftarrow S_1 \cup S_2$ .  $T_1$  RST für  $S_1$  und  $T_2$  RST für  $S_2$ 
  1. Konstruiere  $T$  (Füge  $y$  zwischen  $\text{Max}(S_1)$  und  $\text{Min}(S_2)$  ein. Voraussetzung:  $\text{Max}(S_1) \leq \text{Min}(S_2)$ )
  2. Lösche die Wurzel (Durch runterrotieren des eingefügten Knotens  $y$ )