ALP III — Bonuszettel — Gruppe 1.8

Tobias Lohse, Marvin Kleinert, Anton Drewing — 8. Januar 2015

Aufgabe 1

(a) Skipliste

Da die Skipliste auf unterster Ebene eine sortierte Liste enthält, ist es sehr leicht den Median zu finden. Wir speichern uns einfach einen Pointer auf den Median `m`, welcher in O(1) verfügbar ist.

Nur bei `insert()` und `delete()` müssen wir beachten, ob sich der Median verschiebt. Dazu führen wir einen mit 0 initialisierten Hilfsparameter `l=0` ein. Bei `insert(k)` müssen wir nun falls `k > m` gilt `l` inkrementieren und sonst dekrementieren. Bei `delete(k)` tuen wir es genau umgekehrt. außerdem rufen wir am Ende von `delte()` und `insert()` die Hilfsfunktion `shift_mdeian()` auf.

```
shift_median():
    if l == 2:
        m = m.right
        l = 0
    if l == -2:
        m = m.left
        l=0
```

Es ist sehr leicht den Median in einer Skipliste zu finden. Wir haben ohne die Laufzeit der Operationen `insert()` und `delete()` zu verschlechtern eine Möglichkeit gefunden den Median in O(1) zu finden.

(b) Trie

Wir speichern in jedem Knoten `node`, die Anzahl `node.size` der Blätter des Teilbaums, welcher der Anzahl der Wörter mit diesem Buchstaben an dieser Stelle entspricht. Dazu muss bei `insert()` nur in jedem Knoten auf dem Weg `node.size` inkrementiert werden und bei `delete()` in jedem Knoten auf dem Weg `node.size` dekrementiert werden.

Dann können wir den Median in O(log n) finden mit folgender Funktion:

```
median():
    m = root.size/2
    node = root
    while m != 0:
        i = 0
        while node.letter[i].size < m:
            m -= node.letter[i].size
            i++
        node = node.letter[i]
    return node
```

Alternativ können wir auch direkt einen Pointer auf den Median speichern. Dazu gehen wir genau wie bei der Skipliste vor und speichern einen Hilfsparameter `l=0`. Bei `insert()` und `delete()` vergleichen wir das einzufügende Wort gegen den Median und inkrementieren oder dekrementieren `l` entsprechend. Bei $|l| \geq 2$ verschieben wir den Median nach links oder rechts. Dabei gehen wir wie folgt vor um zum Vorgänger zu gehen — um zum Nachfolger zu gelangen gehen wir analog vor.

- Starte im Knoten
- Iteriere:
  - Gehe zum Vaterknoten.
  - Von dort gehe in den nächst rechtesten Knoten ausgehend vom Startknoten oder von ganz rechts.
  - Wenn ein Knoten gefunden wurde, brich ab.
- Solange wir nicht in einem Endknoten sind:
    - gehe ins rechteste Kind des Knoten
- Gib den gefundenen Endknoten zurück

(c) Hashing

Wenn wir nicht eine Ordnungserhaltende Hashfunktion haben, was in den meisten Fällen unpraktikabel ist. Bleibt uns nichts anderes übrig als ein Array `K` mit allen Schlüsseln zu erstellen. Dies kann entweder bei `insert()` in O(1) und `delete()` mit O(n) geschehen, oder erst beim Aufruf von `median()`.

Um den Median zu finden, müssen wir dann nur `K` mit Mergesort oder ähnlichem sortieren und das mittleren Element von `K` zurückgeben `find(K[K.len/2])`. Das braucht O(n log n). Es gibt auch Algorithmen, die den Median in O(n) finden.

(d) 2–3-Baum

Wir gehen genauso vor wie bei der Skipliste und beim Trie und setzen eine Hilfsvariable `l`. Bei `insert()` und `delete()` vergleichen wir das einzufügende Wort gegen den Median und inkrementieren oder dekrementieren `l` entsprechend. Wenn $|l| \geq 2$, verschieben wir den Median jeweils nach links oder rechts. Dazu verwenden wir dieselbe `prev()` und `next()` Funktion, wie beim Trie.

Aufgabe 2

(a) Binärbaum

Wenn `A` der Baum mit den kleineren Elementen ist und `B` der Baum mit den größeren Elementen, können wir diese einfach zusammen fügen, indem wir `B` rechts ans größte Element (rechteste Blatt) von `A` hängen.

(b) AVL-Baum

Wenn wieder `A` der Baum mit den kleineren und `B` der mit den größeren Elementen ist, dann nehmen wir das kleinste Element (linkeste Blatt) `x` von `B` als neue Wurzel und wissen, dass `A` links angehängt

werden kann und `B` rechts. Nun müssen wir testen ob der Teilbaum in `x` balanciert ist, ist dies nicht der Fall, so müssen wir eine rotation bzw. eine Doppelrotation ausführen und danach wieder den Teilbaum in `x` — welches nun verschoben ist — testen und rotieren, solange bis der Teilbaum in `x` balanciert ist.

Aufgabe 3

Folgender Algorithmus berechnet die minimal nötige Anzahl an Wächtern um alle Bilder, deren Orte im Array `pic_pos` gespeichert sind, zu überwachen.

```
count_guards(pic_pos):
    guard_pos = [];
    for x in pic_pos:
        if !(x > guard_pos[-1]+1 && x < guard_pos[-1]-1):
            gurad_pos += x+1
    return guard_pos.len
```

Jeder neue Wächter wird soweit rechts wie möglich positioniert um das nächste bisher noch unbewachte Bild grade noch zu bewachen und soviel weiteren Platz wie Möglich nach rechts zu überwachen. Also sind alle Wächter ideal positioniert und somit erhalten wir die minimal nötige Anzahl an Wächtern.

Aufgabe 5

Tabelle des in der Vorlesung gegebenen Algorithmus:

```
  | $ S t u d e n t
------------------
$ | 0 1 2 3 4 5 6 7
M | 1 1 2 3 4 5 6 7
e | 2 2 2 3 4 4 5 6
n | 3 3 3 3 4 5 4 5
s | 4 4 4 4 4 5 5 5
a | 5 5 5 5 5 5 6 6
```

Es braucht 6 Operationen um ‘Student’ in ‘Mensa’ umzuwandeln.

Aufgabe 6

(a) Doppelter Eulerpfad

Wir laufen wie bei einer Tiefensuche durch den Graph, dabei markieren wir, in welche Richtungen ein Pfad bereits gelaufen wurde. Der nächste zu laufende Pfad wird nach folgenden in absteigender Priorität sortieren zusätzlichen Kriterien bestimmt:

1. Pfade die noch gar nicht gegangen wurden, haben immer Priorität.
2. Einen Pfad zurück zu-laufen hat Priorität.
3. Nach normaler Tiefensuche nächster Pfad.

(b) Azyklizität eines gerichteten Graphen

Wir starten eine Tiefensuche in einem Knoten, der ein Blatt ist (keine ausgehenden Kanten hat); falls es einen solchen Knoten nicht gibt, so hat der Graph einen Kreis. In jedem Knoten, den wir während der Tiefensuche erreichen, überprüfen wir, ob es eine Verbindung zu einem Knoten gibt, den wir bereits besucht haben; ist dies der Fall, so hat der Graph einen Kreis. Wenn der Graph nicht zusammenhängend ist, müssen wir die Prozedur auf jeder Zusammenhangskomponente ausführen. Sollten wir am Ende keinen Kreis gefunden haben, so ist der Graph azyklisch.

(c) Satz von Euler-Hierholzer

Ein ungerichteter Graph enthält einen Eulerkreis (Kreis, der alle Kanten genau einmal durchläuft), genau dann wenn der Graph zusammenhängend ist und jeder Knoten geradzahligen Grad hat.

Die Hinrichtung ist leicht zu zeigen. Auf der Eulertour wird jeder Knoten, der betreten wird auch wieder verlassen, also muss jeder Knoten von gradem Grad sein. Außerdem ist es offensichtlich, dass in einem nicht zusammenhängenden Graph kein Eulerkreis existieren kann.

Für die Rückrichtung verwenden wir Induktion. Der Induktionsanfang ist ein Graph mit drei Knoten. Wenn dieser zusammenhängend ist und jeder Knoten graden Grad hat, erhalten wir notwendigerweise ein Dreieck, welches offensichtlich einen Eulerkreis enthält. Im Induktionsschritt betrachten wir einen Graphen mit k Knoten mit gradem Grad. Dieser Graph muss einen Kreis enthalten, da er wegen des graden Grads aller Knoten kein Blatt enthalten kann und somit kein Baum sein kann. Dieser Kreis kann entweder ein Eulerkreis sein, oder nicht. Wenn nicht, dann entfernen wir alle Kanten des Kreises aus dem Graphen. Es bleiben ein oder mehrere zusammenhängende Graphen übrig, deren Knoten immer noch alle von gradem Grad sind, da wir ja beim entfernen des Kreises den Grad jedes Knotens um eine grade Anzahl verringern mussten. Mit der Induktionsannahme folgt, dass die verbleibenden Zusammenhängenden Graphen mit $\leq k$ Knoten jeweils einen Eulerkreis besitzen. Somit haben wir den Graphen in mehrere sich berührende Kreise aufgeteilt. Den Eulerkreis konstruieren wir, indem wir einen Kreis durchlaufen und an jedem Berührungspunkt, den wir erreichen, zunächst den berührenden Kreis durchlaufen, bevor wir unseren äußeren Kreis zu Ende durchlaufen. Wir haben also einen Eulerkreis gefunden.

Damit ist es nun leicht zu überprüfen, ob ein Graph einen Eulerkreis enthält. Wir starten in einem beliebigen Knoten und führen eine Tiefensuche aus, in jedem Knoten überprüfen wir, ob der Grad grade ist, wenn dies in einem Knoten nicht der Fall ist, so gibt es keinen Eulerkreis. Können wir durch die Tiefensuche nicht alle Knoten erreichen, so ist der Graph nicht zusammenhängend und hat damit auch keinen Eulerkreis.