Probeklausur zur Vorlesung

Algorithmen und Datenstrukturen

SoSe 2025

Wolfgang Mulzer

Bearbeitungszeit: mindestens 90 Minuten

Aufgabe 1 Datenstrukturen *4+3+3 Punkte*

Sei U eine total geordnete Menge. Wir wollen Teilmengen $S \subseteq U$ speichern, so dass folgende Operationen möglich sind:

- `insert`(x): Füge x zu S hinzu.
- `deleteMin`(): Voraussetzung: S ist nicht leer. Das kleinste Element aus S soll gelöscht werden.
- `deleteMax`(): Voraussetzung: S ist nicht leer. Das größte Element aus S soll gelöscht werden.

Sie dürfen annehmen, dass wir zwei Elemente aus U in konstanter Zeit vergleichen können. Für jede der folgenden drei Datenstrukturen, beschreiben Sie jeweils kurz, wie man die Operationen `insert` und `deleteMax` möglichst effizient implementieren kann, und geben Sie möglichst gute asymptotische obere Schranken für die Laufzeit. Erklären Sie gegebenenfalls, welche zusätzlichen Annahmen nötig sind.

(a) AVL-Baum;

(b) Hashtabelle mit linearem Sondieren;

(c) unkomprimierter Trie.

Lösung Aufgabe 1:

Gesucht:

- Operationen insert und delteMax
- obere Schranke der Laufzeit (Worst-case O-Notation)
- Vergleiche passieren in Konstanter Zeit also $O(1)$

AVL-Baum:

insert(k):

(a) Suche nach dem Schlüssel k im Baum
  - Beginne bei der Wurzel
  - Vergleiche k mit n.k
    - k == n.k: Wert gefunden
    - k < n.k: gehe in den linken Teilbaum
    - k > n.k: gehe in den rechten Teilbaum
  - Falls $NULL$ erreicht: k nicht vorhanden
  - return die Position von $NULL$ oder k

(b) Einfügen als neues Blatt oder Updaten von dem Vorhanden Wert an stelle k
  - beim Updaten passiert nichts weiter mit der AVL-Eigenschaft, da sich keine Höhen ändern → wir sind fertig

(c) Wenn wir ein neues Blatt einfügen → überprüfe den Pfad vom neuen Blatt bis zur Wurzel den Balance-factor der Knoten
  - BF = |Höhe des rechten Teilbaums| − |Höhe des linken Teilbaums|
  - Wenn der BF > 1 müssen wir eine Rotation ausführen um den BF wieder herzustellen
  - mögliche Rotationen: links, rechts, linksrechts, rechtslinksrotation
    - Baumstruktur links-links → R-Rotation
    - Baumstruktur rechts-rechts → L-Rotation
    - Baumstruktur links-rechts → LR-Rotation
    - Baumstruktur rechts-links → RL-Rotation

deleteMax():

- Von der Wurzel ausgehen verfolge den rechten Teilbaumpfad, bis zum ende, dort befindet sich in einem Binary Search Tree das größte Element(AVL-Baum ist eine Binary Search Tree)
- Lösche das Element
- Analog wie bei `insert(k)` den BF überprüfen und ggf. Rotationen ausführen

Laufzeit: im Worst-case Fall

- Suche(k) - O(log n) - Da der ein AVL-Baum selbstbalancierend ist, kann er nicht so stark degenerieren, das wir ein links- oder rechtsgerichteten Baum erhalten
- insert(k) - O(log n) - wie beim Suchen ähnliche Struktur
- deleteMax() - $O(\log n)$ - Suchen $O(\log n)$ Löschen $O(1)$
- Rebalancieren - $O(\log n)$

Hashtabelle mit linearem Sondieren:

insert(k):

- Berechne den Hashwert von h(k) und gehen an die Stelle im Array
- Wenn die Stelle $NULL$ ist einfügen von (k,v)
- Wenn die Stelle belegt ist gehe im Array weiter bis `NULL` gefunden wird
  - Dabei Speichern wir die erste Position von `DELETED`
- Am ende schauen wir ob wir eine `DELETED` position haben, wenn ja fürgen wir k dort ein, wenn nicht am ende wo wir `NULL` vorgefunden haben

deleteMax(): Hashtabellen gehören zu Dictionaries und die sind Ungeordnet, deshalb ist es uneffizient den Maximalwert des gesamten Arrays rauszufinden, wir müssten alle Entries durchgehen und Vergleichen was auf eine Laufzeigt von O(größe der Tabelle) hinauslaufen würde

Laufzeit:

- Suchen - Worst-case O(länge des Arrays), da wir bei linearem Sondieren, wenn ein Kollision auftritt den Eintrag immer weiterschieben, bis wir eine leere Position finden. Wenn das Array sehr voll ist kann es vorkommen das wir einmal das gesamte Array durchlaufen müssen.
- insert(k) - $O(n)$ Laufzeit hängt wie beim Suchen vom Füllgrad ab
- deleteMax() - siehe erklärung bei `deleteMax()`

unkomprimierter Trie:

insert(k):

- Für jedes Zeichen c im Schlüssel s
  - Prüfe: gibt es eine ausgehende Kante für Symbol c?
    - Wenn ja: Folge der Kante zum nächsten Knoten
    - Wenn nein:

        - Erstelle einen neuen Knoten
        - Füge eine Kante mit Symbol c hinzu, die zum neuen Knoten führt.
- Nach dem letzten Zeichen:
  - Markiere den Knoten als `Ende eines Schlüssels` mit z.B.: $

deleteMax():

- Vom Wurzelknoten aus:
  - Folge immer der größten möglichen ausgehenden Kante, also das höchste Zeichen
  - solange wie möglich nach rechts unten gehen
- Sobald man bei einem Ende-Knoten landet z.B.: bei $ → ist dies der größte gespeicherte Schlüssel
- Diesen löschen wir dann:
  - Entferne den Ende-Marker
  - Fall der Knoten von keinen anderen Schlüssel mehr benutzt wird, löschen wir den Knoten

Laufzeit:

- insert(k): $O(|s| \cdot |\Sigma|)$
- deleteMax(): $O(|s_{max}| \cdot |\Sigma|)$

Aufgabe 2 Hashing *1+5+4 Punkte*

(a) Nennen Sie zusätzlich zu Hashing mit Verkettung noch zwei weitere Möglichkeiten der Kollisionsbehandlung in einer Hashtabelle.

(b) Fügen Sie nacheinander die Schlüssel 5, 28, 19, 15, 20, 33, 12, 17, 10 in eine Hashtabelle der Größe 9 ein. Die Hashfunktion sei $h(k) = k \bmod 9$. Die Konflikte werden mit linearem Sondieren gelöst. Verwenden Sie dafür das Schema auf der nächsten Seite.

(c) Beschreiben Sie einen Weg, wie man Kuckucks-Hashing mit *drei* Hashfunktionen implementieren kann. Geben Sie Pseudocode für die Einfüge-Operation.

Lösung Aufgabe 2:

a:

(a) Hashing with linear probing

(b) Cuckoo Hashing

b:

Ich füge hier einfach das endergebnis ein:

0	1	2	3	4	5	6	7	8
10	28	19	20	12	5	15	33	17

c:

```
put(k, v):
 // if k is present, we simply update the value
 if (T[h1(k)].k == k)
  T[h1(k)].v <- v
  return
 if (T[h2(k)].k == k)
  T[h2(k)].v <- v
  return
 if (T[h3(k)].k == k)
  T[h3(k)].v <- v
  return
 // if not, we must insert (k, v)
 // if the table is already full, we cannot insert $k$
 if (|S| == N)
  throw TableFullException
 // we try to insert the entry at the first position
 pos <- h1(k)
 for i := 1 to N do
  // if the position is empty, we insert the entry
  // and are done
  if (T[pos] == NULL)
   T[pos] <- (k,v)
   return
  // otherwise, we exchange the current entry and
  // determine the second possible position
  (k,v) <-> T[pos]
  if (pos == h1(k))
   pos <- h2(k)
  else if (pos == h2(k))
   pos <- h3(k)
```

```
 else
  pos <- h1(k)
// if the insertion does not succeed, we need to rebuild the table
Choose new hash functions and rebuild the table
put(k,v)
```

Aufgabe 3 Vermischtes *2+2+2+2+2 Punkte*

(a) Nennen Sie zwei Eigenschaften und zwei mögliche Anwendungen von kryptographischen Hashfunktionen.

(b) Wahr oder falsch: Der Algorithmus von Dijkstra funktioniert auch in Graphen mit negativen Kantengewichten. Begründen Sie Ihre Antwort.

(c) Zeichnen Sie einen komprimierten Trie für die Wörter KLAUSUR, KLASSE, KLEEBLATT, KLEISTER.

(d) Wahr oder falsch: Ein binärer Baum der Höhe h besitzt immer mindestens 2^h Knoten. Begründen Sie Ihre Antwort. (Zur Erinnerung: Die *Höhe* bezeichnet die maximale Anzahl von Kanten von der Wurzel des Baumes bis zu einem Blatt.)

(e) Nennen Sie einen Vorteil und einen Nachteil von (a, b)-Bäumen gegenüber AVL-Bäumen.

Lösung Aufgabe 3:

Teilaufgabe (a):

Eigenschaften:

- Schnell berechenbar für beliebige Eingaben
- Kollisions resistent: Es ist praktisch unmöglich, zwei vverschiedene Eingaben zu finden, die denselben Hashwert haben

Anwendungen:

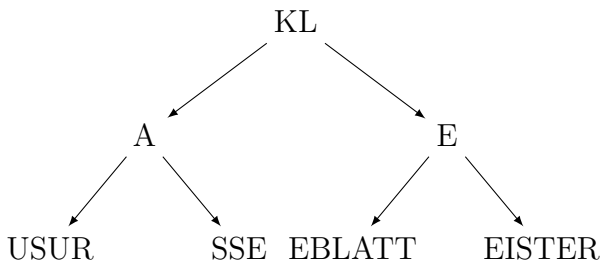
- Passwort-Hashing - man Speichert beim eingeben des Passworts nicht das Passwort in Klartext, sondern nur den Hash-Wert
- Hash-Referenzen und Blockchain - Durch das Speichern von Hashreferenzen in Objekten kann man Manipulation am Objekt sehr leicht erkennen, da die Hash-Werte nicht mehr übereinstimmen

Teilaufgabe (b):

Antwort: Falsch

Begründung: Dijkstra funktioniert nicht in Graphen mit negativen Kantengewichten, weil er Algorithmus einmal gefundene minimale Distanzen als endgültig markiert. Sollten doch negative Kanten existieren, könnten spätere kürzere Wege existieren, die Dijkstra nicht mehr berücksichtigt.

Teilaufgabe (c):



Teilaufgabe (d):

Antwort: Falsch

Begründung: Ein binärer Baum der Höhe h kann im Worst-case nur $h+1$ Knoten besitzen - z.B.: wenn wir einen degenerierten Baum in Kettenform gegeben haben, dort hätte jeder Knoten nur genau ein Nachfolger. Das Minimum wäre also linear und nicht exponentiell.

Teilaufgabe (e):

Vorteil: (a, b)-Bäume sind besser für externe Speicher, wie Festplatten, geeignet, weil sie weniger Knotenbesuche pro Suche benötigen -dadruch werden Speicherzugriffe reduziert.

Nachteil: Suchen und Updaten innerhalb eines Knotens, der viele Einträge(Schlüssel) enthält, sind aufwändiger als bei AVL-Bäumen. Da bei AVL-Bäumen jeder Knoten nur ein Schlüssel enthält. (a, b)-Bäume haben zusätzlich einen höheren, dafür aber konstanteren Speicherbedarf.

Aufgabe 4 Graphen *6+4 Punkte*

(a) Führen Sie im folgenden ungewichteten Graphen eine Breitensuche durch, um die kürzesten Wege ausgehend vom Knoten s zu ermitteln.

Verwenden Sie dafür das Schema auf der nächsten Seite. Der Pseudocode für BFS ist wie folgt:

```
Q <- new Queue
s.found <- true
s.d <- 0
s.pred <- NULL
Q.enqueue(s)
while not Q.isEmpty() do
    (*)
    v <- Q.dequeue()
    for w in v.outNeighbors() do
        if not w.found then
            w.found <- true
            w.d <- v.d + 1
            w.pred <- v
            Q.enqueue(w)
    (**)
```

In dem Schema sollen Sie jeweils vermerken: den Zustand der Warteschlange zu Beginn der `while`-Schleife (an der Stelle (*) im Pseudocode); den Knoten v, der aus der Schleife entfernt wird (next vertex); die Nachbarn w, die in der `for`-Schleife durchlaufen werden (neighbors); sowie den Zustand der `found` (f), d und `pred` (π) Attribute für jeden Knoten am Ende der `while`-Schleife (an der Stelle (**) im Pseudocode).

(b) Sei $G = (V, E)$ ein gerichteter, ungewichteter Graph. Der *transponierte Graph* G^T ist der Graph, den wir aus G erhalten, indem wir die Richtungen aller Kanten in G umdrehen. Das heißt, es ist $G^T = (V, E^T)$, wobei

$$E^T = \{(w, v) \mid (v, w) \in E\}.$$

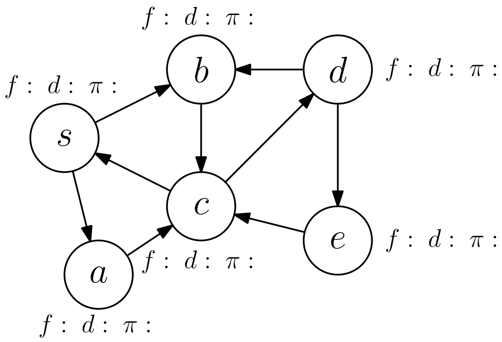
Geben Sie jeweils einen effizienten Algorithmus an, der G^T aus G konstruiert, wenn G (i) als Adjazenzliste und (ii) als Adjazenzmatrix gegeben ist.

Beschreiben Sie Ihren Algorithmus jeweils in Worten, und analysieren Sie die Laufzeit.

Queue:

next vertex

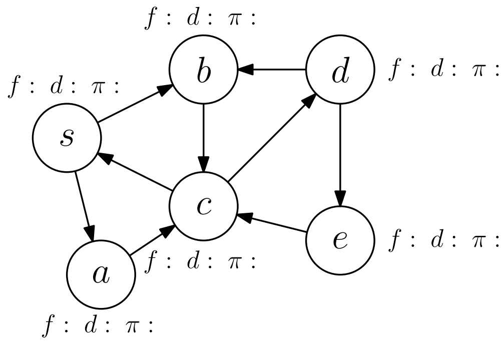
neighbors



Queue:

next vertex

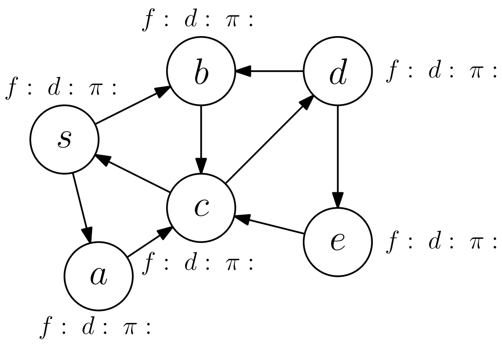
neighbors



Queue:

next vertex

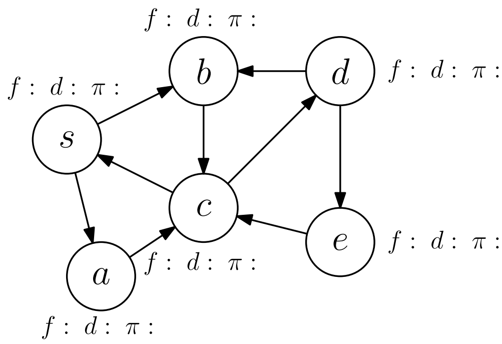
neighbors



Queue:

next vertex

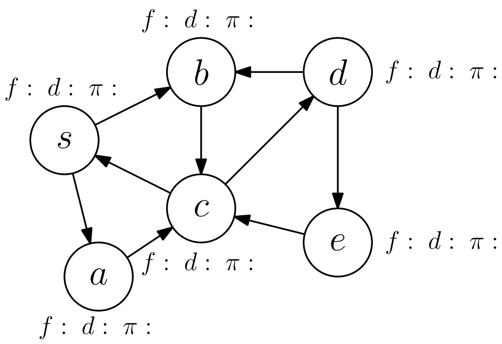
neighbors



Queue:

next vertex

neighbors



Queue:

next vertex

neighbors