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.

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.

insert 5 $h(5) =$

0	1	2	3	4	5	6	7	8

insert 28 $h(28) =$

0	1	2	3	4	5	6	7	8

insert 19 $h(19) =$

0	1	2	3	4	5	6	7	8

insert 15 $h(15) =$

0	1	2	3	4	5	6	7	8

insert 20 $h(20) =$

0	1	2	3	4	5	6	7	8

insert 33 $h(33) =$

0	1	2	3	4	5	6	7	8

insert 12 $h(12) =$

0	1	2	3	4	5	6	7	8

insert 17 $h(17) =$

0	1	2	3	4	5	6	7	8

insert 10 $h(10) =$

0	1	2	3	4	5	6	7	8

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.

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

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

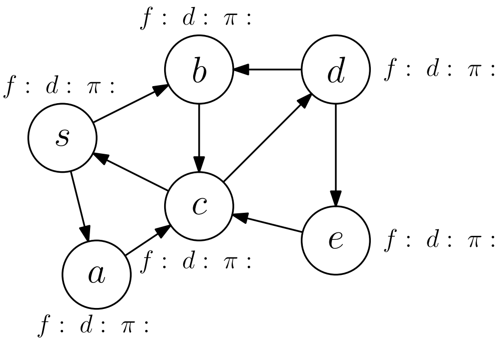
Geben Sie jeweils einen effizienten Algorithmus an, der G^T aus G konstruiert, wenn G (i) als Adjazenzliste und (ii) als Adjazenzmatrix gegeben ist.

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

Queue:

next vertex

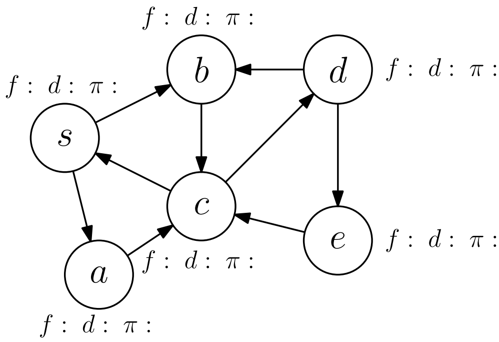
neighbors



Queue:

next vertex

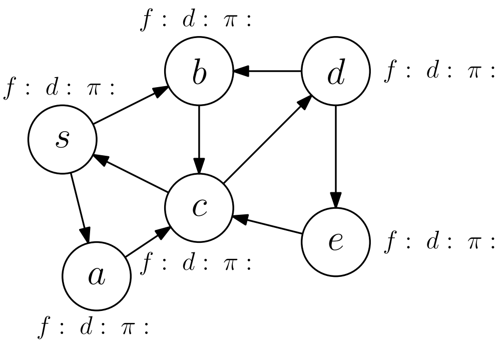
neighbors



Queue:

next vertex

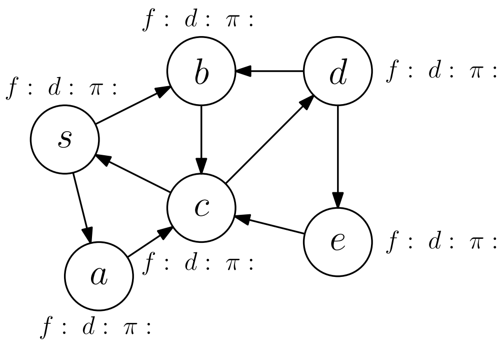
neighbors



Queue:

next vertex

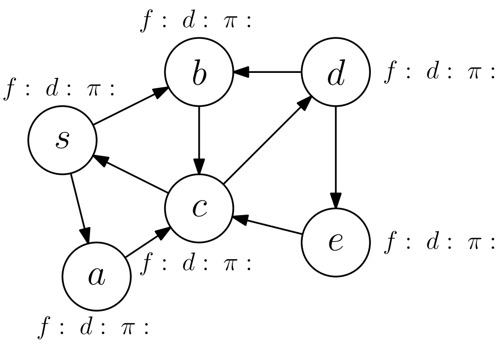
neighbors



Queue:

next vertex

neighbors



Queue:

next vertex

neighbors