Name:

Vorname:

Matrikelnummer:

Lösungsvorschlag

Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. P. Sanders 14.3.2014

Klausur Algorithmen I

Aufgabe 1.	Kleinaufgaben	16 Punkte
Aufgabe 2.	k-Cores	10 Punkte
Aufgabe 3.	Selektieren und Sortieren	8 Punkte
Aufgabe 4.	Palindrome	11 Punkte
Aufgabe 5.	Minimale Spannbäume	8 Punkte
Aufgabe 6.	Hashing mit linear Probing	7 Punkte

Bitte beachten Sie:

- Als Hilfsmittel ist nur **ein** DIN-A4-Blatt mit Ihren **handschriftlichen** Notizen zugelassen.
- **Schreiben** Sie auf **alle** Blätter Ihren Namen und Ihre Matrikelnummer.
- Merken Sie sich Ihre Klausur-ID für den Notenaushang.
- Die Klausur enthält 15 Blätter.
- Die durch Übungsblätter gewonnenen Bonuspunkte werden erst nach Erreichen der Bestehensgrenze hinzugezählt. Die Anzahl Bonuspunkte entscheidet nicht über das Bestehen.

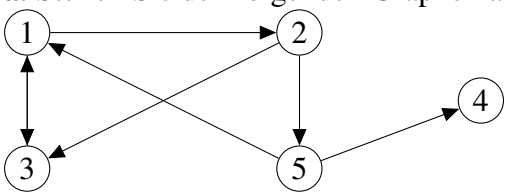
Aufgabe		1	2	3	4	5	6	Summe
max. Punkte		16	10	8	11	8	7	60
Punkte	EK							
	ZK							

Bonuspunkte:	Summe:	**Note:**

Lösungsvorschlag

Aufgabe 1. Kleinaufgaben [16 Punkte]

a. Stellen Sie den folgenden Graphen als Adjazenzfeld dar: [2 Punkte]



	1	2	3	4	5	6
V						

	1	2	3	4	5	6	7
E							

Lösung

	1	2	3	4	5	6
V	1	3	5	6	6	8

	1	2	3	4	5	6	7
E	2	3	3	5	1	1	4

Lösungsende

b. Nennen Sie zwei Operationen, die auf einer einfach verketteten Liste $O(1)$ Zeit, auf einem unbeschränkten Feld hingegen $O(n)$ Zeit benötigen. [1 Punkt]

Lösung

pushFront, popFront, concat, splice

Lösungsende

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

c. Lösen Sie die drei folgenden Rekurrenzen im Θ-Kalkül:

$$V(n) = 2014n + V(n/8), \qquad V(1) = 42$$
$$W(n) = n/2014 + 9W(n/8), \qquad W(1) = 5$$
$$X(n) = 8X(n/8) + V(n) \qquad X(1) = 1$$

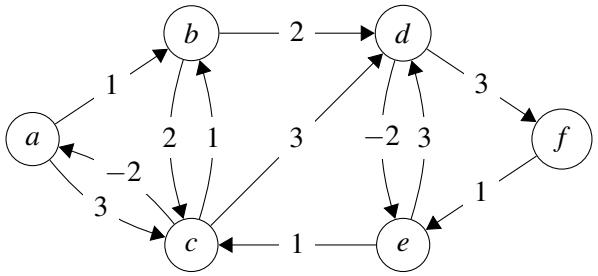
mit $n = 8^k$ und $k \in \mathbb{N}_{>0}$. [3 Punkte]

Lösung

$V(n) = \Theta(n)$, $W(n) = \Theta(n^{\log_8 9})$ und $X(n) = \Theta(n \log n)$.

Lösungsende

d. Wieviele kürzeste Wege von a nach f enthält folgender gerichtete gewichtete Graph? Begründen Sie kurz. [2 Punkte]



Lösung

Unendlich viele kürzeste Wege. Der kürzeste Distanz von a nach f ist 6. Wegen dem Zyklus $a \to b \to d \to e \to c \to a$, der das Gesamtgewicht 0 hat, gibt es aber unendlich viele Pfade von a nach d mit Gewicht 3.

Lösungsende

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

e. Zeigen Sie oder widerlegen Sie, dass $3^n = \Omega(3^{2n})$ gilt. [2 Punkte]

Lösung

Die Behauptung gilt nicht.
Angenommen die Behauptung gilt. Dann gibt es ein $c > 0$ und ein $n_0 \in \mathbb{N}$, so dass für alle $n \leq n_0$ gilt: $3^n \geq c \cdot 3^{2n}$, also

$$\frac{3^n}{3^{2n}} \geq c \quad \Leftrightarrow \quad \frac{1}{3^n} \geq c \; .$$

Es gilt aber $1/3^n \to 0$ für $n \to \infty$. Dies führt zu einem Widerspruch.

Lösungsende

f. Gegeben sei eine Vorfahrendatenbank in Form eine Folge D von Paaren (ElternID, KindID), wobei Eltern und Kinder durch Personen-IDs aus $\mathbb{N}$ dargestellt werden.

Geben Sie einen Algorithmus an, der in erwartet $O(|D|)$ Zeit feststellt, ob die Person mit ID $a \in \mathbb{N}$ Vorfahre der Person mit ID $b \in \mathbb{N}$ ist? Korrekte Lösungen die dies noch in $O(|D| \log |D|)$ Zeit erreichen erhalten noch 2 Punkte. [3 Punkte]

Lösung

Die ElternIDs werden mit Hilfe von Hashing gruppiert und ein Graph in Adjazenzfeld-Darstellung wird aufgebaut. Dannach macht man eine Breitensuche von dem Knoten mit ID a aus und gibt die Lösung entsprechend aus.

Lösungsende

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

g. Gegeben ist das folgende Parent-Array aus dem *Basis-Algorithmus* Union-Find der Vorlesung (kein Union-by-Rank, ohne Pfadkompression) :

v	1	2	3	4	5	6
parent[v]	4	2	2	4	2	1

Geben Sie eine Folge von *Union Operationen* des Basis-Union-Find Algorithmus an, so dass das gegebene parent-Array erzeugt wird. Geben Sie außerdem nach jeder Union Operation an, welche *link* Operation dadurch ausgeführt wird und wie sich das Parent-Array dadurch ändert. *Hinweis:* In der Vorlesung steht $link(i, j)\ \{\ parent[i] := j\ \}$. [3 Punkte]

v	1	2	3	4	5	6
parent[v]	1	2	3	4	5	6
parent[v]						
parent[v]						
parent[v]						
parent[v]						
parent[v]						
parent[v]						

Lösung

union(6,1); union(5,2); union(3,5); union(6,4) (die Operationen werden zu link(6,1) [parent[6] = 1], link(5,2) [parent[5] = 2], link(3,2) [parent[3]=2], link(1,4) [parent[1]=4]. Die Bäume die zu dem Parent-Array gehören, lass sich folgendermaßen darstellen:

Lösungsende

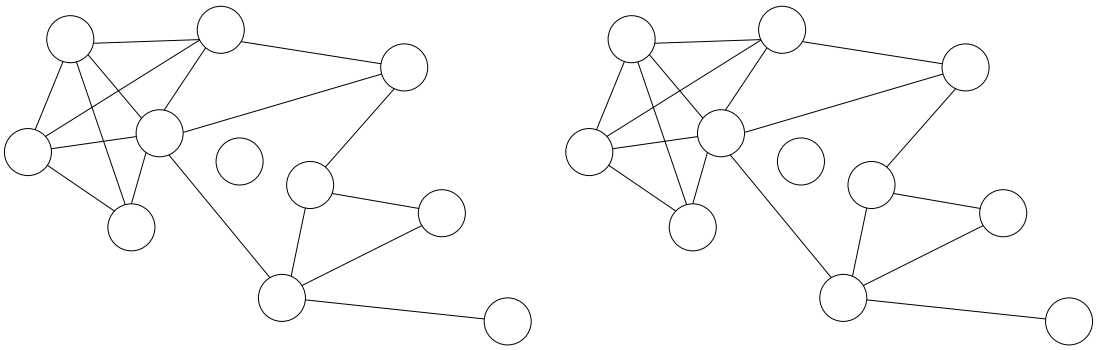
Lösungsvorschlag

Aufgabe 2. k-Cores [10 Punkte]

Der k-Core eines ungerichteten Graphen $G = (V,E)$ ist gegeben durch die größte Teilmenge $V' \subseteq V$, so dass alle Knoten im knoteninduzierten Teilgraph[1] $G' := G[V']$ einen Knotengrad $deg_{G'}(v)$ größer gleich k haben. Die Core-Struktur $\mathscr{C} : V \to \mathbb{N}_{\geq 0}$ gibt für jeden Knoten v die größte Nummer x an, so dass v noch im x-Core enthalten ist.

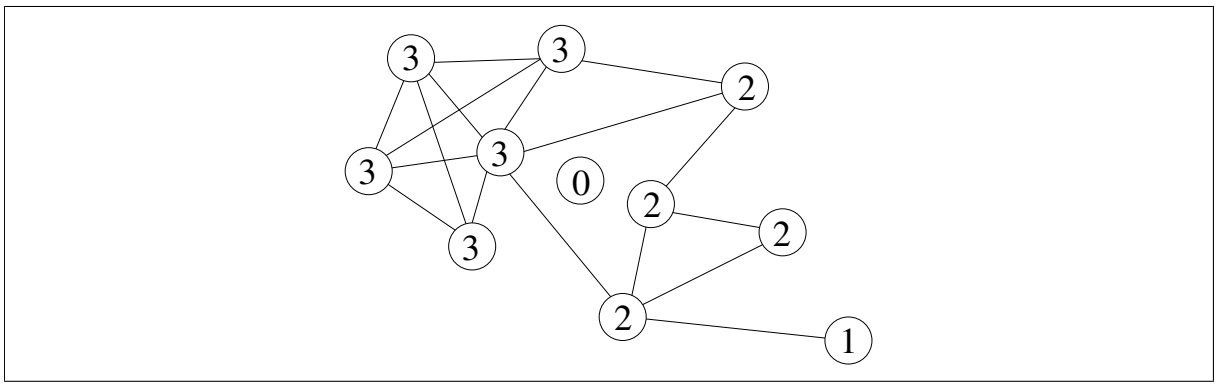
a. Geben Sie für folgenden Graphen die Core-Struktur $\mathscr{C}$ an. Schreiben Sie dazu in jeden Knoten v den Wert $\mathscr{C}(v)$.

Für die **Lösung:** Kopie zum **Rechnen:**



[2 Punkte]

Lösung



Lösungsende

b. Geben ist ein ungerichteter Graph $G = (V,E)$. Geben Sie einen Algorithmus an, der die Core-Struktur $\mathscr{C}$ in Zeit $O(|V| + |E|)$ berechnet und ausgibt. Begründen Sie kurz das Laufzeitverhalten ihres Algorithmus.
Hinweis: Lösungen die die Aufgabe in Zeit $\Theta((|V| + |E|)\log|V|)$ lösen erhalten 6 Punkte. Beschreiben Sie Datenstrukturen die nicht in der Vorlesung behandelt wurden genau. [8 Punkte]

[1] $G[V'] = (V', E' := \{\{u,v\} \in E \mid u,v \in V'\})$

(mehr Platz zum Schreiben finden Sie auf dem nächsten Blatt)

Lösungsvorschlag

Fortsetzung von Aufgabe 2

Lösung

Wir geben zunächst einen Algorithmus in Pseudocode an der das Problem löst:

```
deg : Array[1..n] of ℕ                          // Array für den Knotengrad, mit 0 initialisiert
active : Array[1..n] of {0,1}                   // Array für aktive Knoten, mit 1 initialisiert
𝒞 : Array[1..n] of ℕ                            // Array für die Core-Struktur, mit 0 initialisiert
Q : Addressable Priority Queue

foreach {u,v} ∈ E do                            // Knotengrad berechnen
    deg(u) += 1
    deg(v) += 1

foreach v ∈ V do                                // Prioritätsliste initialisieren
    Q.insert(v, deg(v))

while (!Q.empty())
    v := Q.extractMin()
    𝒞[v] := deg(v)                              // Core-Nummer zuweisen
    active(v) := 0                              // Knoten wurde entfernt
    foreach {v,u} ∈ E and active[v] do          // Knotengrad der Nachbarn updaten
        deg[u] -= 1
        Q.decreaseKey(u, deg[u])
return 𝒞
```

Der dargestellte Algorithmus löst die Aufgabe in $\Theta((|V|+|E|)\log|V|)$, falls ein binary Heap als Priority Queue verwendet wird. In diesem Fall kostet jede der verwendeten Priority Queue Operationen $O(\log n)$. Der dominierende Faktor in dem Algorithmus ist damit die letzte While-Schleife. Die asymptotische Laufzeit ergibt sich, da jeder Knoten genau einmal aus der Prioritätsliste mit extractMin geholt wird und für jede Kante höchstens einmal decreaseKey aufgerufen wird. Insgesammt gab es dafür 6 Punkte.

Die letzten zwei Punkte gab es, wenn man eine Bucket Priority Queue verwendet hat und damit die geforderte Laufzeit aus der Aufgabenstellung erreicht. Die Bucket Priority Queue musste genau erläutert werden. Eine Bucket Priority Queue kann mit doppelt verketteten Listen aus der Vorlesung wie folgt umsetzen (es waren auch weniger detailierte Antworten zulässig):

```
Class BucketPQ
    buckets : Array[0..n−1] of DoublyLinkedLists
    element_pointer : Array[0..n−1] of Handle   // Pointer, mit NULL initialisiert
    element_key : Array[0..n−1] of ℕ≥0          // Keys, mit 0 initialisiert
    min = n : ℕ≥0                               // Zeiger auf das kleinste nicht leere Bucket

    Procedure insert(v : V, k : Key)
        Handle h := buckets[k].insertFront(v)
        element_pointer[v] := h
        element_key[v] = k
        min = k < min ? k : min
```

```
Procedure decreaseKey(v : V, k : Key)
    Key old_key := element_key[v]
    Handle h := element_pointer[v]
    buckets[old_key].remove(h)          // v aus dem aktuellen Bucket entfernen
    insert(v,k)                         // v wieder in die Queue einfügen

Procedure extractMin()
    Handle h = buckets[min].head;
    buckets[min].remove(h)              // v aus dem aktuellen Bucket entfernen
    update min to smallest non-empty bucket
    return h.v

Procedure empty()
    track size in other methods
    return if PQ is empty
```

Insgesamt reduziert sich die Laufzeit der insert und decreaseKey Operation durch die Verwendung dieser Datenstruktur auf $O(1)$. Die Laufzeit der extractMin Operation ist abhängig davon, ob das gerade angeschaute Bucket danach leer wird. Ist dies der Fall muss das nächst kleinste Bucket gefunden werden. Nehmen wir an, das nächst kleinste Bucket gehört zu dem Key k. Der nächste Aufruf von extractMin liefert uns dann einen Knoten, der noch mindestens Grad k hat. Dadurch werden noch k Kanten angeschaut und die Kosten werden dadurch amortisiert. Damit erhalten wir eine Gesamtlaufzeit von $O(|V|+|E|)$.

Lösungsende

Lösungsvorschlag

Aufgabe 3. Selektieren und Sortieren [8 Punkte]

a. Gegeben sei die Sequenz $S = \langle 65,28,50,33,21,56,22,95,50,12,90,53,28,77,39\rangle$. Bestimmen Sie mit Hilfe des *Quickselect*-Algorithmus aus der Vorlesung das Element mit Rang $k = 8$. Geben Sie jeweils für jeden Rekursionsschritt die aktuell betrachtete Teilsequenz s, den aktuellen Wert für k, sowie die Sequenzen $a := \langle e \in s : e < p\rangle$, $b := \langle e \in s : e = p\rangle$ und $c := \langle e \in s : e > p\rangle$ an. Das Pivot-Element p ist für jeden Schritt vorgegeben. [4 Punkte]

s	k	p	Sequenzen
$\langle 65,28,50,33,21,56,22,$ $95,50,12,90,53,28,77,39\rangle$	**8**	**28**	$a : \langle 21,22,12\rangle$ $b : \langle 28,28\rangle$ $c : \langle 65,50,33,56,95,50,90,53,77,39\rangle$
$\langle 65,50,33,56,95,50,90,53,77,39\rangle$	3	**90**	$a : \langle 65,50,33,56,50,53,77,39\rangle$ $b : \langle 90\rangle$ $c : \langle 95\rangle$
$\langle 65,50,33,56,50,53,77,39\rangle$	3	**39**	$a : \langle 33\rangle$ $b : \langle 39\rangle$ $c : \langle 65,50,56,50,53,77\rangle$
$\langle 65,50,56,50,53,77\rangle$	1	**56**	$a : \langle 50,50,53\rangle$ $b : \langle 56\rangle$ $c : \langle 65,77\rangle$
$\langle 50,50,53\rangle$	1	**50**	$a : \langle\rangle$ $b : \langle 50,50\rangle$ $c : \langle 53\rangle$

Lösung

Das Element mit Rang $k = 8$ ist 50.

Lösungsende

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 3

b. Was ist die worst-case Laufzeit und die erwartete Laufzeit von Quickselect? [2 Punkt]

Lösung

Die erwartete Laufzeit von Quickselect ist $O(n)$ auf einer Sequenz der Länge n. Die worst-case Laufzeit ist $O(n^2)$, wie bei Quicksort.

Lösungsende

c. Ist Quick**sort** *in-place*? Wenn nein, mit wie viel zusätzlichem Platz kommt man in der besten Variante aus der Vorlesung aus? [2 Punkt]

Lösung

Quicksort ist nicht in-place, da auf dem Stack bis zu $O(\log n)$ Rekursions-Ebenen abgespeichert werden müssen. Die einfachste Quicksort Variante aus der Vorlesung benötigt sogar bis zu $O(n)$ Ebenen, mit der Rekursion in den nur kleineren Teil erhält man höchstens $O(\log n)$ Ebenen.

Lösungsende

Lösungsvorschlag

Aufgabe 4. Palindrome [11 Punkte]

Ein *Palindrom* ist ein Wort w, das rückwärts geschrieben das gleiche Wort bildet. Beispiele hierfür sind 'anna', 'kayak' und 'reittier'.
Jedes Wort w kann in eine *Sequenz von Palindromen* zerlegt werden: 'ababba' lässt sich in 'a|b|abba', 'aba|bb|a', 'a|bab|b|a' oder 'a|b|a|b|b|a' zerlegen, also in 3–6 Palindrome.
Wir bezeichnen die *minimale Anzahl* von Palindromen in die w zerlegt werden kann mit $p(w)$.

a. Zerlegen Sie das Wort $w =$'abbababaabac' in eine Sequenz von $p(w)$ Palindrome und notieren Sie $p(w)$. Markieren Sie deutlich welche Zerlegung zu werten ist. [2 Punkte]

$w =$ a b b a b a b a a b a c

$p(w) =$ ☐

$w =$ a b b a b a b a a b a c

Lösung

'abba|b|abaaba|c' und $p(w) = 4$.

Lösungsende

b. Entwerfen Sie einen Algorithmus, der für ein Wort w die Zahl $p(w)$ in $O(n^2)$ berechnet, wobei n die Länge von w ist. Begründen Sie kurz seine Korrektheit und analysieren Sie die Laufzeit. Die Zerlegung selbst brauchen Sie nicht ausgeben.

Hinweise: Sei $w[i..j]$ das Teilwort von w, das den i-ten bis j-ten Buchstaben enthält. Konstruieren Sie zuerst ein Array $L[i,j]$, das angibt, ob $w[i..j]$ ein Palindrom ist.
Lösungen in $\omega(n^2)$ geben höchstens 5 Punkte. [9 Punkte]

(mehr Platz zum Schreiben finden Sie auf dem nächsten Blatt)

Lösungsvorschlag

Fortsetzung von Aufgabe 4

Lösung

Wir verwenden dynamische Programmierung und geben die Rekurrenzen, Reihenfolge und Sentinels an.
Das Array $L[i,j]$ enthalte *true*, wenn $w[i..j]$ ein Palindrom der Länge $j-i+1$ ist. Man kann L berechnen indem man $L[i,j] := \mathit{true}$ für $i = j$ und $i-1 = j$ festgelegt und

for $j = 1,\ldots,n$ **do** // Schleife über das Teilwort $w[1..j]$
  for $i = j-1,\ldots,1$ **do** // Schleife über die Palindrom-Länge
    $L[i,j] := (w[i] = w[j] \text{ und } L[i+1,j-1])$

durchläuft. Die Werte $i = j$ und $i-1 = j$ sind Sentinels und bezeichnen Palindrome der Länge Eins (ein einzelner Buchstabe) und Null (kein Buchstabe). Längere Palindrome werden dadurch berechnet, dass man den ersten $w[i]$ und letzten Buchstaben $w[j]$ vergleicht und prüft ob die dazwischenliegenden Buchstaben ein Palindrom sind. In der Schleife wird schrittweise das Wort w durch Anhängen eines Buchstabens $w[j]$ aufgebaut, und dabei längere Palindrome aus kürzen konstruiert.
Mit L kann man nun $p(w)$ mit dynamischer Programmierung über die Teilworte $w[1..j]$ berechnen. Man kann $M[j] = p(w[1..j])$ errechnen, in dem man $M[0] = 0$ als Sentinel festlegt und dann

for $j = 1,\ldots,n$ **do** $\quad M[j] := \min_{i=1,\ldots,j} \{M[i-1]+1 \mid L[i,j] = \mathit{true}\}$

durchläuft. Für jedes $M[j]$ durchläuft i dabei alle möglichen Zerlegungs-Schnitte in $w[1..j]$. Falls $w[i..j]$ ein Palindrom ist, kann man auf $p(w[1..(i-1)])$ des übrigen Worts zurückgreifen und Eins dazu zählen. Um $p(w[1..j])$ zu ermitteln, bestimmt man das Minimum über alle möglichen Schnitte.
Der gesamte Algorithmus liegt in $O(n^2)$, da sowohl die Berechnung von L in $O(n^2)$ als auch diejenige von M in $O(n^2)$ liegen.

Lösungsende

Lösungsvorschlag

Aufgabe 5. Minimale Spannbäume [8 Punkte]

a. Betrachten Sie zuerst zwei Spannbäume $T_1, T_2 \subseteq E$ in einem ungerichteten Graphen $G = (V, E)$. Zeigen Sie, dass es für jede Kante $e_1 \in T_1$ eine Kante $e_2 \in T_2$ gibt, so dass sowohl $(T_1 \setminus \{e_1\}) \cup \{e_2\}$ als auch $(T_2 \setminus \{e_2\}) \cup \{e_1\}$ ein Baum ist. [4 Punkte]

Lösung

Seien T_1 und T_2 zwei Bäume und $e_1 \in T_1$ eine Kante.
Ist $e_1 \in T_2$, so braucht man nichts zu machen: $e_2 = e_1$ genügt den Anforderungen trivial.
Wir betrachten also $e_1 \notin T_2$. Dann schließt e_1 mit T_2 einen Kreis $C \subseteq T_2 \cup \{e_1\}$. Von diesem Kreis ist mindestens eine Kante $e_2 \in C$ nicht in T_1 enthalten, da T_1 kreisfrei ist. Es ist auch $e_2 \neq e_1$, da $e_1 \in T_1$. Wir haben nun $e_1 \in T_1 \setminus T_2$ und $e_2 \in T_2 \setminus T_1$, es ist also $|T_1| = |T_1 \setminus \{e_1\} \cup \{e_2\}| = |T_2 \setminus \{e_2\} \cup \{e_1\}| = |T_2|$. In $T_1 \cup \{e_2\} \setminus \{e_1\}$ schließt e_2 einen Kreis (nämlich C) in T_1, der von e_2 wieder aufgelöst wird. In $T_2 \cup \{e_1\} \setminus \{e_2\}$ schließt e_1 einen Kreis (auch C) in T_1, der von e_2 wieder aufgelöst wird. Beide sind also kreisfrei und haben die gleiche Anzahl von Kanten wie die Bäume, damit sind die beiden Konstrukte Bäume.

Lösungsende

b. Nennen und *beweisen* Sie die Schnitteigenschaft für minimale Spannbäume. [4 Punkte]

Lösung

Für einen Knotenmenge $S \subseteq V$ betrachte die Schnittkanten $C = \{\{u, v\} \in E \mid u \in S, v \in V \setminus S\}$.
Die Schnitteigenschaft lautet: „Die leichteste Kante e in C kann in einem MST verwendet werden.".
Angenommen T sei ein minimaler Spannbaum und $e \in C$ eine leichteste Kante eines beliebigen Schnittes C.
Liegt $e \in T$, so haben wir die Eigenschaft gezeigt, da e in einem MST verwendet wird.
Liegt jedoch $e \notin T$, dann schließt e mit T einen Kreis $K \subseteq T \cup \{e\}$. Da K ein Kreis ist, gibt es mindestens eine weitere Kante $f \in C \cap K$ mit $f \neq e$. Dann ist $T' = T \setminus \{e\} \cup \{f\}$ ebenfalls ein Baum und dessen Gewicht ist nicht größer als das von T, da e eine leichteste Kante in C ist. Da aber T minimales Gewicht hat, haben T und T', und damit auch e und f, beide jeweils das gleiche Gewicht.

Lösungsende

Lösungsvorschlag

Aufgabe 6. Hashing mit linear Probing [7 Punkte]

Wir betrachten in dieser Aufgabe Hashtabellen mit n Buckets und zugehörigen Hashfunktionen,

$$h_n(x) = x \text{ MOD } n\,.$$

Beispielsweise ist $h_7(42) = 0$. Zur Kollisionsauflösung wird lineare, zyklische Suche angewendet. Folgende Hashtabelle hat die Größe $n = 10$ und Hashfunktion h_{10}:

0	1	2	3	4	5	6	7	8	9
99	10		43	33	35	63			49

a. Sei nun eine leere Hashtabelle mit $n = 10$ und Hashfunktion h_{10} gegeben.
Geben Sie eine Folge von *insert*-Operationen an, so dass die Tabelle nach Ausführen dieser Operationsfolge den obigen Zustand hat. Wie viele Lesezugriffe auf das Array werden bei Ihrer Operationsfolge ausgeführt? [2 Punkte]

Lösung

Eine mögliche Lösung: 49, 99, 10, 43, 33, 35, 63
In jedem Fall: 13 Lesezugriffe

Lösungsende

(Teilaufgaben **b.** und **c.** auf dem nächsten Blatt)

Lösungsvorschlag

Fortsetzung von Aufgabe 6

b. Geben Sie für eine leere Hashtabelle der Größe n mit Hashfunktion h_n eine Folge von zuerst n **verschiedenen** *insert* Operationen und anschließend n **verschiedenen** *find* Operationen an, so dass jede einzelne *insert* Operationen zwar nur genau einen Lesezugriff auf die Tabelle benötigt, die Laufzeit jeder der *find* Operationen aber linear in der Tabellengröße ist. Begründen Sie kurz, warum Ihre Folge das gewünschte Verhalten liefert. [3 Punkte]

Lösung

Die Idee ist, mit den *insert*s die Tabelle kollisionsfrei komplett zu füllen, und dann mit den *find*s immer wieder nach einem Element zu suchen, welches nicht vorhanden ist. Also:
Betrachte die Folge *insert(i)* mit $i = 0,\ldots,n-1$ und *find(i)* mit $i = n, n+1\ldots,2n-1$. Die *insert* Operationen brauchen jeweils nur einen Lesezugriff, da alle Elemente verschieden und kleiner als n sind und dementsprechend keine Kollisionen stattfinden. Nach den *inserts* sind alle Einträge belegt. Die n *find*-Operationen fragen nach Elementen, welche nicht in der Tabelle enthalten sind. Da die Tabelle aber voll ist, muss jeder der Operationen $O(n)$ Lesezugriffe tätigen, bis sie dies feststellen kann.

Lösungsende

c. Nennen Sie zwei Vorteile von Hashing mit linearer Suche gegenüber Hashing mit verketteten Listen. [2 Punkte]

Lösung

- geringerer Speicherverbrauch, da keine Zeiger benötigt werden
- Lokalität der Speicherzugriffe
- unter Umständen einfachere Implementierung (je nach benötigten Methoden)

Lösungsende