Übungsblatt 4

Abgabe via Moodle.
Deadline Fr. 2ter Juni

Aufgabe 1 (*Hashing*, $4+1+2+1$ *Punkte*)

Gegeben sei eine Hashtabelle mit 10 Buckets. Dabei sei die Hashfunktion h definiert als die Funktion, welche eine Zahl auf die Einerstelle abbildet, zum Beispiel gilt $h(13) = 3$.

1. Verwenden Sie einerseits Hashing mit verketteten Listen, andererseits Hashing mit linearer Suche (zyklisches Array), um folgende Operationen durchzuführen:

   Einfügen von 19, 52, 25, 29, 62, 12, 88, 10, 53

   Geben Sie jeweils die Tabellen nach dem Einfügen von 62 und nach dem Einfügen von 53 an.

2. Geben Sie die Anzahl der beim Einfügen betrachteten Hashtabellenplätze für beide Verfahren an!

3. Nehmen Sie an, dass nach jedem Datum mit gleicher Wahrscheinlichkeit gesucht wird. Welche Kosten sind für das jeweilige Verfahren für eine erfolgreiche Suche zu erwarten?

4. Wie groß ist der Speicherverbrauch, wenn sowohl Zeiger als auch Element ein Maschinenwort benötigen, und mit einem Nullzeiger, das Ende einer Liste markiert werden kann?

Lösung:

1. **Nach dem Einfügen von 62**
   Hashing mit verketteten Listen (—∘ stellt einen NIL-Zeiger dar.):

0	1	2	3	4	5	6	7	8	9
—∘	—∘	→ 62 → 52 —∘	—∘	—∘	→ 25 —∘	—∘	—∘	—∘	→29 → 19 —∘

   Hashing mit linearer Suche:

0	1	2	3	4	5	6	7	8	9
29		52	62		25				19

   Nach dem Einfügen von 53
   Hashing mit verketteten Listen (—∘ stellt einen NIL-Zeiger dar.):

0	1	2	3	4	5	6	7	8	9
→10—∘	—∘	→12→ 62 → 52 —∘	→ 53 —∘	—∘	→ 25 —∘	—∘	—∘	→88—∘	→29 → 19 —∘

   Hashing mit linearer Suche:

0	1	2	3	4	5	6	7	8	9
29	10	52	62	12	25	53		88	19

2. Hashing mit verketten Listen: 9, Hashing mit linearer Suche:17

3. Hashing mit verketten Listen: 13/9 , Hashing mit linearer Suche: 17/9

4. Hashing mit verketten Listen (Zeiger in der Hashtabelle): 18 + 10 = 28
   Hashing mit linearer Suche: 10 Maschienenworte

Aufgabe 2 (*Rechnungssystem,* $6 + 2$ *Punkte*)

Nehmen Sie an Sie haben eine große Datei die aus Tripeln der Form (*transaction, price, customerID*) besteht.

1. Entwickeln Sie einen Algorithmus, der den Gesamtzahlbetrag pro Kunden in erwarteter linearer Laufzeit berechnet.
2. Begründen Sie die erreichte Laufzeit!

Lösung:

1. Man nehme eine Hashtabelle H mit verketteten Listen und einer zufälligen Hashfunktion aus einer universellen Familie.

   Speichere zu jeder *customerID* den Gesamtpreis in der Hashtabelle H mit *customerID* als Schlüssel. Dazu muss man als erstes die Datei einmal durchlaufen um zu zählen wie viele Einträge in der Datei vorhanden sind, um die Hashtabelle entsprechend zu initialisieren. Nun geht man die Datei ein zweites Mal durch um den Gesamtpreis für jeden Customer zu bestimmen. Das funktioniert folgendermaßen:
   Wenn die zum aktuellen Tripel gehörende CustomerID noch nicht in der Hashtabelle ist, so wird ein Eintrag in der Hashtabelle erzeugt und mit dem Preis aus dem Tripel initialsiert. Sollte die CustomerID schon in der Hashtabelle vorhanden sein, so wird der Preis der in der Hashtabelle gespeichert ist um den Preis aus dem Tripel erhöht. So akkumuliert sich Stück für Stück beim Durchgehen der Datei der Gesamtpreis eines jeden Customers.
   Zum Schluss gibt man alle Elemente der Hashtabelle aus. Dazu kann man den folgenden Algorithmus verwenden.

   ```
   1: procedure billing_system(F : File)
   2:     initialize Hashtable H with |File| Slots and a random hashfunction from a universal Family
   3:     for (tid, price, customerID) ∈ File do
   4:         if H.contains(customerID) then H.find(customerID) += price
   5:         else H.insert(customerID), H.find(customerID) = price
   6:     for (key, value) ∈ H do print "Customer mit ID $key muss $value zahlen!"
   7: return
   ```

2. Das Durchlaufen der Datei und das Initialisieren der Hashtabelle kostet jeweils O (|File|). Jede Hashtabellen-Operation in Zeile 4 und 5 hat erwartete Laufzeit O (1) (da die Hashtabelle $\Omega(|\text{File}|)$ Slots hat), die erwartete Laufzeit ist somit O (|File|). Das Ausgeben aller Elemente der Hashtabelle geht ebenfalls in O (|File|) (siehe Übung). Damit ergibt sich eine erwartete Gesamtlaufzeit von O (|File|).

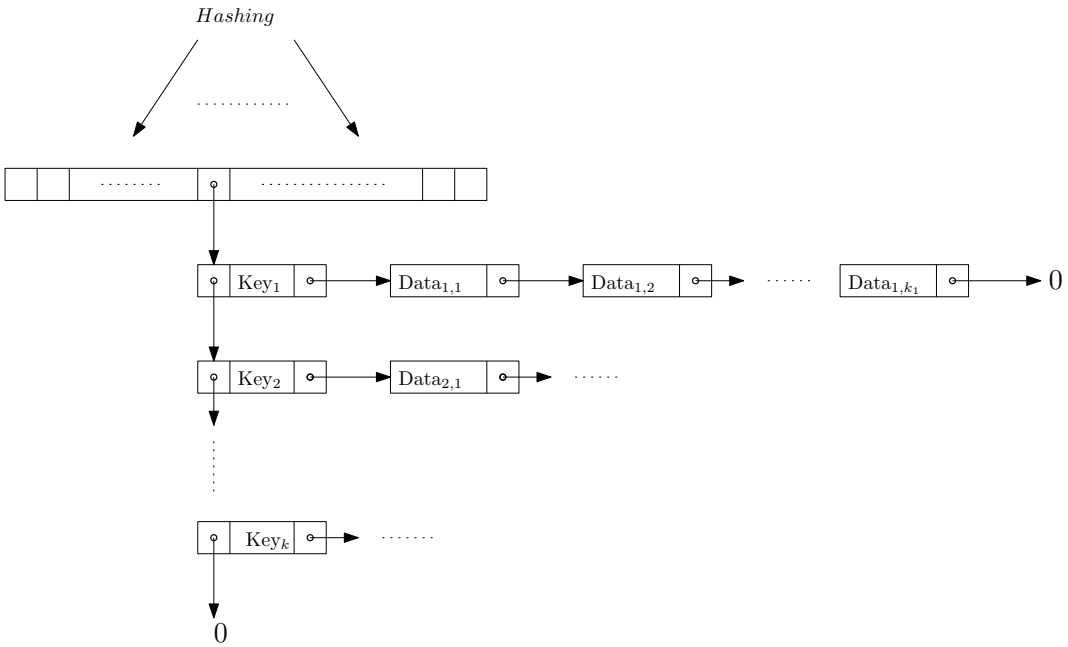
Aufgabe 3 (*Entwurf einer Datenstruktur,* 8 *Punkte*)

Gegeben sei ein Datentyp D mit Elementen der Form (*Schlüssel*, *Nutzdaten*). Die Größen von Schlüssel und Nutzdaten seien konstant. Die maximale Anzahl n der verschiedenen Schlüssel, die zu einem bestimmten Zeitpunkt in einem etwaigen System existieren, sei von vornherein bekannt. Die maximale Anzahl m der verschiedenen *Tupel* sei jedoch *nicht* von vornherein bekannt. Es können zudem auch Tupel mit dem gleichen Schlüssel vorkommen (im Extremfall können sogar alle vorhandenen Tupel den gleichen Schlüssel haben). Entwerfen Sie nun eine Datenstruktur für den Typ D, die folgendes leistet:

- Es existiert eine Operation *Insert*(x_s, x_d), die ein Tupel (x_s, x_d) aus Schlüssel und Nutzdaten in die Datenstruktur einfügt. Ein Tupel wird dabei auch dann eingefügt, wenn in der Datenstruktur bereits ein Tupel mit dem selben Schlüssel existiert.
- Es existiert eine Operation *Remove*(x_s), die für einen Schlüssel x_s ein Tupel mit passendem Schlüssel zurückgibt und aus der Datenstruktur entfernt. Falls kein solches Tupel in der Datenstruktur existiert, wird NIL zurückgegeben.
- Beide Operationen benötigen erwartet $\mathrm{O}(1)$ Zeit.
- Die Datenstruktur hat einen Speicherbedarf von höchstens $\mathrm{O}(\max\{n, m\})$.

Lösung:
1. Idee: Verwende eine Hashtabelle mit verketteten Listen.
Da der Speicherbedarf auf $\mathrm{O}(\max\{n, m\})$ beschränkt ist, kann eine Hashtabelle maximal die eine Größe in $\Theta(n)$ haben. Um für ein $m \gg n$ noch eine erwartete Zugriffszeit von $\mathrm{O}(1)$ zu haben, reicht eine Hashtabelle der Größe $\mathrm{O}(n)$ jedoch nicht aus.

Eigentliche Lösungsidee: Verwende eine Hashtabelle mit verketteten Listen für die Schlüssel und "hänge" an jeden Schlüssel x_s eine verkettete Liste mit den Nutzdaten der Tupel mit dem gleichen Schlüssel.



Man erstellt eine Hashtabelle der Größe n und wählt zufällig eine geeignete Hashfunktion (universelles Hashing).

Für die Operation *Insert*(x_s, x_d) greift man auf das Listenelement mit Schlüssel x_s (falls es existiert) zu und fügt x_d vorne an die Liste der Datenelemente hinzu. Falls in der Hashtabelle mit verketteten Listen kein Listenelement für x_s existiert, erstellt man eines am Ende der verketteten Liste zum

entsprechenden Hash-Slot, und fügt x_d als erstes Element in die verkettete Liste der Daten ein. Aus Satz 1 der Vorlesung ist bekannt, dass erwartet $\mathrm{O}(1)$ Schlüssel mit x_s kollidieren. Daher ist der Aufwand in beiden Fällen in $\mathrm{O}(1)$.

Für die Operation *Remove*(x_s) greift man auf das Listenelement mit Schlüssel x_s (falls es existiert) zu und entnimmt das erste Element x_d der Liste der Datenelemente. Falls in der Hashtabelle mit verketteten Listen kein Listenelement für x_s existiert, gibt man NIL aus. Aus Satz 1 der Vorlesung ist bekannt, dass erwartet $\mathrm{O}(1)$ Schlüssel mit x_s kollidieren. Daher ist der Aufwand in beiden Fällen in $\mathrm{O}(1)$.

Aufgabe P4 (*Modified Fibonacci, optional*)

Für die praktischen Übungen verwenden wir die Plattform `www.hackerrank.com`. Hier müssen Sie sich registrieren um an den Übungen teilzunehmen. Unter dem Link

`https://www.hackerrank.com/adsi-2023`

finden die praktischen Übungen in der Form eines Programmierwettbewerbs statt.

In der vierten Challenge geht es um eine modifizierte Fibonacci Folge.

Die Vorschrift zur Berechnung der Folge ist

$$t_{i+2} = (t_i)^2 - t_{i+1} \tag{1}$$

Die Anfangswerte t_0 und t_1 die für die Berechnung notwendig sind werden als Eingabe bereitgestellt.

Zusätzlich ist die Anzahl der zu berechnenden Folgenglieder n gegeben. Das heißt, die Rückgabe der zu vervollständigenden Funktion **newFibonacci** soll gerade t_n sein.

Um die Effizienz ihres Algorithmus zu testen gibt es wie letzte Woche einen großen Test (Case 3) auf Hackerrank. Wenn Ihr Algorithmus effizient genug ist, wird der Test in der vorgegebenen Zeit von 2 Sekunden meistens durchlaufen. Leider funktioniert es nicht immer dass dieser Test durchläuft, selbst bei der optimalen Lösung. Am besten submitten Sie ihren Algorithmus mehrmals.

Falls dieser Test trotzdem immer aufgrund von Zeitmangel fehlschlägt, gibt es eventuell die Möglichkeit Ihren Algorithmus noch zu verbessern.

Eine genauere Beschreibung, sowie ein Beispiel finden Sie auf HackerRank.