Übung 06: Hashtabellen, Radixsort

Aufgabe 1: Radixsort

Skizzieren Sie wie in der Vorlesung den Ablauf von Radixsort für ein Array *A*, das die folgenden englischen Wörter enthält. In jeder Runde wird eine Buchstabenposition sortiert. Geben Sie die Ordnung der Wörter nach *jeder* Runde an!
A = <COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX>

Aufgabe 2: Hashtabellen – Verkettung der Überläufer

Gegeben ist eine Hashtabelle der Größe $m = 9$. Also Hashfunktion wird die Divisionsmethode eingesetzt („*modulo*“). Kollisionen werden durch Verkettung aufgelöst. Neue Schlüssel werden immer am Ende einer Liste hinzugefügt.

a) Illustrieren Sie durch eine Skizze, wie die Hashtabelle aussieht, wenn sie zu Beginn leer ist und wenn man der Reihe nach die folgenden Schlüssel einfügt: 5, 28, 19, 15, 20, 33, 12, 17, 10.

b) Wie viele **Schlüsselvergleiche**[1] benötigen Sie im konkreten Beispiel bei einer ***erfolgreichen Suche*** im Best Case, im Worst Case und im Average Case? Nehmen Sie für den Average Case an, dass jeder Schlüssel mit der gleichen Wahrscheinlichkeit gesucht wird.

Aufgabe 3: Hashtabellen – Lineares Sondieren

Gegeben sei eine Hashtabelle der Größe $\boldsymbol{m = 11}$. Zur Kollisionsauflösung wird ***lineares Sondieren*** verwendet. Die Hashtabelle sei zu Beginn leer.

a) Es werden der Reihe nach die Schlüssel 10, 22, 31, 4, 15, 28, 59 eingefügt. Geben Sie die Belegung der Hashtabelle nach dem Einfügen aller Schlüssel an.

b) Nun Schlüssel 4 mit dem Verfahren der Vorlesung („*Eager Delete*“) gelöscht, siehe Code rechts.

   Führen Sie diesen Code gedanklich aus und geben Sie die Arraybelegung von `keys` am Ende jeder Iteration der **zweiten** `while`-Schleife an.

```
int i = hash(key);
while (!key.equals(keys[i])) {
    i = (i + 1) % m;
}
keys[i] = null;
vals[i] = null;
i = (i + 1) % m;
while (keys[i] != null) {
    Key   keyToRehash = keys[i];
    Value valToRehash = vals[i];
    keys[i] = null;
    vals[i] = null;
    n--;
    put(keyToRehash, valToRehash);
    i = (i + 1) % m;
}
n--;
```

Aufgabe 4: Lineares Sondieren – *Lazy Delete*

Anders als in 3b) sollen die beim Löschen benötigten Maßnahmen nicht sofort erledigt werden, sondern auf später verschoben werden („*Lazy Delete*“). Modifizieren Sie die vorgegebene Java-Klasse `HashtableProbingLazyDelete.java`[2].
Beim „*Eager Delete*“ war es nicht erlaubt, dass ein existierender Key `null` als Value speichert. Beim „*Lazy Delete*“ soll nun beim Löschen eines Key-Value-Paares der Value auf `null` gesetzt werden, der entsprechende Key aber nicht gelöscht werden. Der Key wird dadurch als ungültig/gelöscht markiert und erst beim nächsten `resize()` aus der Hashtabelle entfernt.

Hinweise:

- Die Suche (`get`) muss weiterhin alle gültigen Einträge finden.
- Überschreitet die Anzahl der gültigen Tabelleneinträge 50%, soll die Tabelle verdoppelt werden, unterschreitet die Anzahl der gültigen Tabelleneinträge 12,5% soll die Tabelle halbiert werden. Achten Sie ferner darauf, dass die Anzahl der gültigen UND ungültigen Tabelleneinträge insgesamt nicht mehr als 50% beträgt. Am besten Sie zählen mit, wie viele Einträge aktuell ungültig sind.

(umblättern)

[1] Als Schlüsselvergleich zählt nur, wenn Sie tatsächlich 2 Integer miteinander vergleichen.
[2] Die Dateien sind einzeln oder als IntelliJ-Projekt im Gitlab `Uebungen/Uebung06/` vorgegeben.

- `put` soll Einträge mit ungültigen Schlüsseln ggfs. für das Einfügen verwenden.
- Der Code darf beliebig modifiziert werden, die Schnittstelle muss erhalten bleiben. Es sind gar nicht so viele Änderungen notwendig. Der bisherige Code für `delete` („Eager Delete“) ist auskommentiert, Sie können diesen nehmen und anpassen.
- Testen Sie mit der mitgelieferten Testklasse.