Algorithmen und Datenstrukturen
Sommersemester 2022
Woche 8

Kevin Angele, Tobias Dick, Oskar Neuhuber,
Andrea Portscher, Monika Steidl, Laurin Wischounig

Abgabe bis 17.05.2022 23:59
Besprechung im PS am 19.05.2022

Aufgabe 1 (2 Punkte): Kollisionensbehandlung

Kollisionen sind unvermeidbar bei Hashing - deshalb gibt es verschiedene Methoden damit umzugehen. Gegeben ist folgende Hash-Funktion:

$$f1(k) = (3k + 3)\%11 \tag{1}$$

.
Zeichnen Sie das Array nach jedem Einfügen auf. Folgende Schlüssel werden eingefügt:
13-18-44-23-2-47-22-11.

- Verwenden Sie zur Kollisionsbehandlung lineare Sondierung.
- Verwenden Sie zur Kollisionsbehandlung externe Verkettung.
- Verwenden Sie zur Kollisionsbehandlung Doppeltes Hashing. Wählen Sie dafür eine passende Hash-Funktion.

Lösung:

- Berechnung von Hashwerten (f2(k)) & h(k,i) siehe Doppeltes Hashing:
  k → f1(k), f2(k), h(k,1), h(k,2), h(k,3)
  13 → 9, 1, 10
  18 → 2, 3, 5
  44 → 3, 5, 8
  23 → 6, 5, 0
  2 → 9, 5, 3, 8,2
  47 → 1, 2, 3, 5, 7
  22 → 3, 6, 9, 4, 10
  11 → 3, 3, 6, 9, 1, 7

- lineare Sondierung: | | 47 | 18 | 44 | 22 | 11 | 23 | | | 13 | 2 | |
- externe Verkettung: | | 47 | 18 | 11→22→44 | | | 23 | | | 2 →13 | | |
- Doppeltes Hashing: verwendete Hash-Funktion: f2(k) = q - (k % q) mit Primzahl q ≤ N → f2(k) = 7 - (k % 7)
  Hashfunktion h(k,i) = (f1(k)+i*f2(k))%m | | 47 | 18 | 44 | 22 | | 23 | 11 | 2 | 13 | |

Aufgabe 2 (3 Punkte): Mengenoperationen

Maps können verwendet werden, um Mengen zu modellieren, indem wir bei einem Key-Value-Paar nur den Key betrachten. In Java gibt es vorimplementierte Klassen für Mengen wie zB. HashSet. Implementieren Sie folgende übliche Operationen für Mengen: Schnittmenge (intersection), Vereinigungsmenge (union), Differenz (difference) und symmetrische Differenz (symmetricDifference). Vervollständigen sie dazu den vorgegebenen Code in `Sets.java`. Sie dürfen keine bereits implementierten Funktionen für diese Operationen (zB. `retainAll()`) verwenden.

Lösung:
Siehe: SetsSolution.java

Aufgabe 3 (3 Punkte): Datentyp, Datenstruktur & Algorithmus

Sie wollen einen möglichst effizienten Suchfilter implementieren, der Buchtitel in Ihrer Bibliothek verwaltet. Sie Suche soll wie folgt funktionieren:
Der/Die Nutzer/in tippt die ersten b Buchstaben des Buchtitels ein. Zurückgegeben wird die Anzahl n der Buchtitel, die genau mit der eingegebenen Buchstabensequenz beginnen. k ist eine voreingestellte Konstante die angibt wie viele Buchtitel ausgegeben werden sollen. Das bedeutet ist $n \leq k$, wird eine alphabetisch sortierte Liste der n Buchtitel ausgegeben.

1. Welchen abstrakten Datentyp und welche Datenstruktur würden Sie für diesen Suchfilter von Buchtitel verwenden, wenn sie die Abfrage möglichst effizient gestalten wollen? Begründen Sie Ihre Antwort.
2. Ihr gewählter abstrakter Datentyp ist auf Basis eines Arrays implementiert. Ist hier ein sortiertes oder unsortiertes Array besser geeignet?
3. Welche asymptotische Laufzeitkomplexität (als Groß-O-Funktion von k, n und der Gesamtanzahl N der gespeicherten Buchtitel) beansprucht Ihr Algorithmus, um festzustellen, ob $n \leq k$ ist oder nicht? Begründen Sie Ihre Antwort.
4. Ist $n \leq k$, welche asymptotische Laufzeitkomplexität (als Funktion von k, n und N) beansprucht Ihr Algorithmus, um die Liste der n Buchtitel zu erstellen? Begründen Sie Ihre Antwort.

Lösung:

1. Sortierte Zuordnungstabelle, Binärer Suchbaum; `subMap(k1, k2)` mit `k1` der b-Buchstaben-Sequenz und `k2` derselben Sequenz, wobei der letzte Buchstabe durch seinen alphabetischen Nachfolger ersetzt ist; jeder Knoten speichert die Anzahl u der Knoten seines Unterbaums (ohne asymptotische Laufzeitkosten)
2. Bei einer Suchabfrage für eine Bibliothek wird der Zugriff auf (*get*) die Zuordnung die Anzahl der Hinzufüge-Operationen (*put*) um einiges übersteigen. Daher ist es in diesem Fall besser ein sortiertes Array zu verwenden. Das Einfügen in ein sortiertes Array dauert hierbei länger, dafür sind *get*-Operationen deutlich schneller (bspw. durch binäre Suche).
3. $O(h)$ mit $h = \log N$ - suche erstes Element und Nachfolger des letzten Elements; addiere bzw. subtrahiere die entsprechenden u entlang der beiden Pfade. (Diese Antwort gilt auch für eine Implementierung auf Basis eines sortierten Arrays, wobei h die Anzahl rekursiver Schritte der Binärsuche ist.)

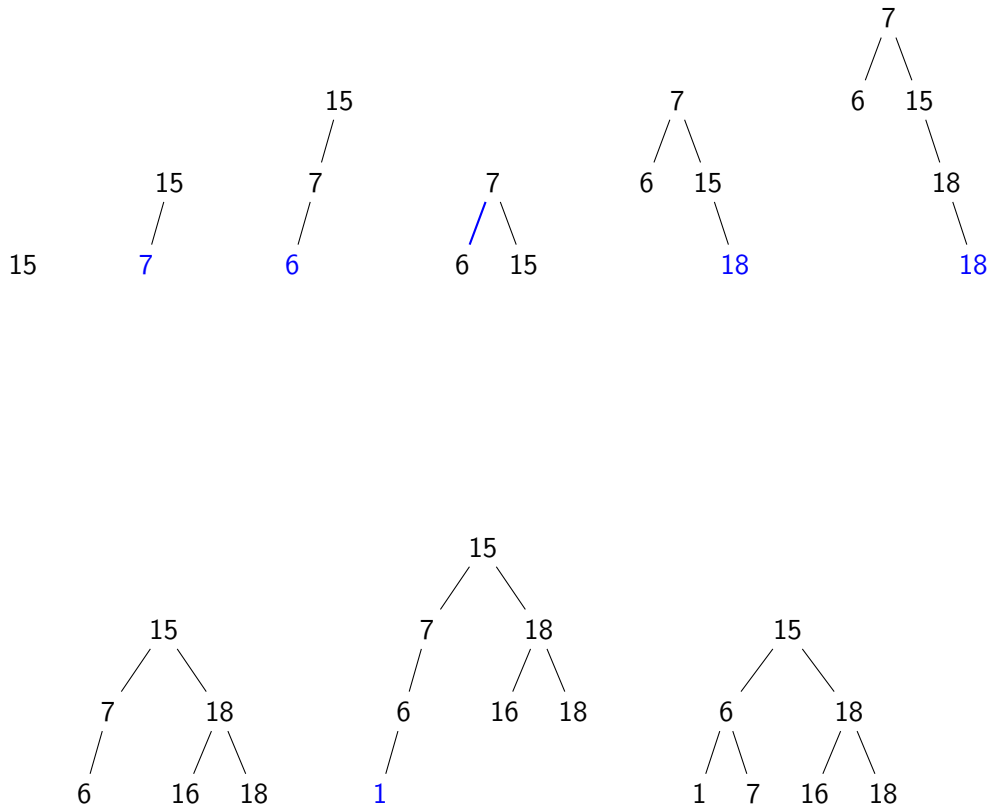
4. Argumentation basierend auf Basis eines sortierten Arrays wo h die Anzahl rekursiver Schritte der Binärsuche definiert: $O(n+h)$ (denjenigen Teil des Baums traversieren, dessen Schlüssel im gegebenen Intervall liegen) (Diese Antwort gilt auch für eine Implementierung auf Basis eines sortierten Arrays, wobei h die Anzahl rekursiver Schritte der Binärsuche ist.)

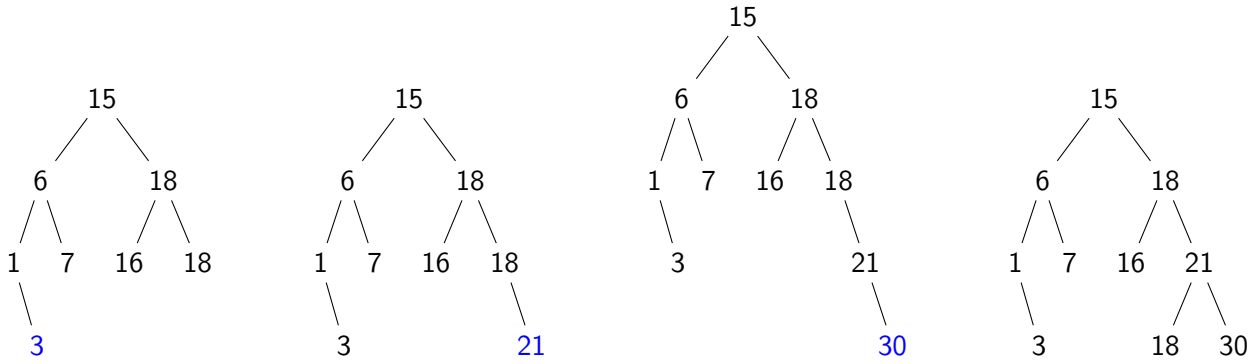
Aufgabe 4 (2 Punkte): AVL-Baum

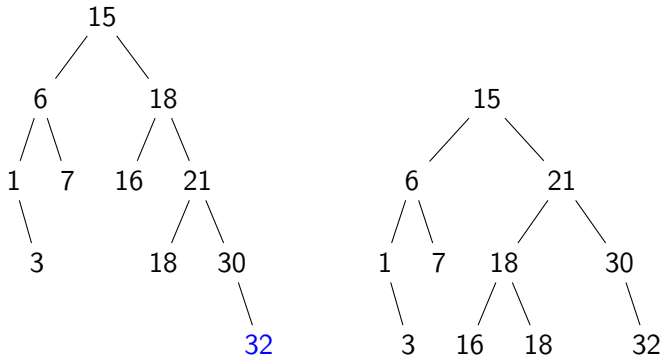
1. Fügen Sie die angegebenen Elemente in einem leeren AVL Baum ein. Zeichnen Sie den Baum nach jeder Operation auf und balancieren Sie ihn, falls notwendig.
   15-7-6-18-18-16-1-3-21-30-32
2. Was ist die Höhe des resultierenden Baumes?
3. Führen Sie eine Inorder-Traversierung durch - was fällt Ihnen bei der Ausgabe der Zahlen auf?
4. Löschen Sie die folgenden Elemente aus dem erzeugten AVL Baum. Zeichnen Sie den Baum nach jeder Operation auf und balancieren Sie ihn falls notwendig.
   18-15-1-3-30

Lösung:

1. EINFÜGEN:







2. Höhe: 4

3. Inorder: 1-3-6-7-15-16-18-18-21-30-32 (Zahlen werden aufsteigend ausgegeben)

4. LÖSCHEN: