Prof. Dr. Leif Kobbelt
Stefan Dollase, Ira Fesefeldt, Alexandra Heuschling, Gregor Kobsik

Lösung - Übung 6

Aufgabe 4 (Selektion des zweitkleinsten Elements): 7 Punkte

Entwerfen Sie einen Algorithmus, welcher aus n Elementen das zweitkleinste auswählt und dafür höchstens $n + \lceil \log_2(n) \rceil$ Vergleiche von Elementen benötigt. Dabei kann es hilfreich sein zusätzlich das kleinste Element zu finden. Begründen Sie Ihre Antwort.

Hinweis:

- Sie dürfen beliebig viele Vergleiche zwischen Booleans nutzen.

Lösung

Wir berechnen zunächst das Minimum indem wir unten stehenden Vergleichsbaum aufbauen. Die erste Ebene entsteht indem wir jeweils zwei direkt nebeneinander liegende Elemente miteinander vergleichen und jeweils das kleinere Element in die neue Ebene schreiben. Zusätzlich merken wir uns auf welcher Seite das kleinere Element stand. Außerdem merken wir uns ob wir gerade die erste Baumebene bauen. Die zweite Ebene entsteht aus der ersten Ebene in genau derselben Art und Weise. Damit haben wir $\frac{n}{2}$ Vergleiche auf der ersten Ebene, $\frac{n}{4}$ auf der zweiten, und so weiter. Zusammen ergibt das

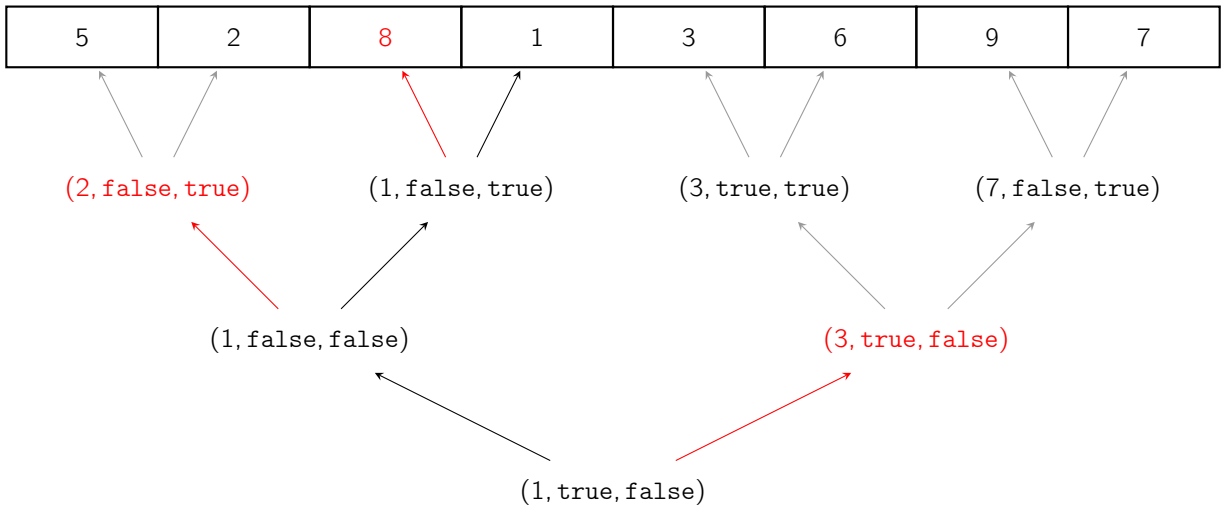
$$\sum_{i=1}^{\lceil \log_2(n) \rceil} \frac{n}{2^i} = -n + \sum_{i=0}^{\lceil \log_2(n) \rceil} \frac{n}{2^i} < n \cdot \frac{1}{0.5} - n = n \ .$$

Anschließend können wir mit folgender Funktion das zweitkleinste Element im Array finden, welche das Minimum der unten rot hervorgehobenen Werte berechnet, wobei wir Folgendes annehmen:

- `(x,y,z).value == x`
- `(x,y,z).minIsLeft == y`
- `(x,y,z).isLastLayer == z`

```
def min2(a):
    if a.isLastLayer:
        if a.minIsLeft:
            return a.right
        else:
            return a.left
    else:
        if a.minIsLeft:
            return min(a.right.value, min2(a.left))
        else:
            return min(a.left.value, min2(a.right))
```

Ohne Beschränkung der Allgemeinheit nehmen wir an, dass das Minimum im linken Teilbaum ist. Wir wissen, dass die Wurzel im rechten Teilbaum das kleinste Element des rechten Teilbaums ist. Rekursiv können wir nun noch das zweit kleinste Element im linken Teilbaum berechnen. Das kleinere dieser beiden muss nun auch das zweit kleinste Element sein.
Falls das Minimum im rechten Teilbaum ist, ist die Korrektheit symmetrisch begründet.
Für dieses Verfahren brauchen wir höchstens $\lceil \log_2(n) \rceil$ Vergleiche, wodurch wir insgesamt mit maximal $n + \lceil \log_2(n) \rceil$ Vergleichen zwischen Elementen auskommen.



Aufgabe 5 (Selektionsalgorithmus mit 7er-Gruppen): 6 Punkte

In der Vorlesung haben Sie gesehen, dass der Median der Mediane in linearer Zeit berechnet werden kann. Dazu wurde mithilfe des Medians der Mediane ein Pivot-Element gewählt. Bei der Berechnung des Medians der Mediane wurde zunächst die Eingabe in 5er-Gruppe unterteilt, anschließend jeweils der Median jeder 5er-Gruppe gebildet und schließlich der Median all dieser Mediane gebildet.
Angenommen die Eingabe wäre nicht in 5er-Gruppen eingeteilt worden, sondern in **7er-Gruppen**. Der restliche Selektionsalgorithmus bleibt hingegen unverändert.
Arbeitet der Selektionsalgorithmus weiterhin in linearer Zeit? Beweisen Sie Ihre Antwort.

Lösung

Der Selektionsalgorithmus benötigt mit 7er-Gruppen ebenfalls lineare Zeit.
Zunächst versuchen wir den Beweis der Linearität aus der Vorlesung zu übertagen.
Dafür haben wir, dass mindestens die Hälfte der $\lfloor \frac{n}{7} \rfloor$ Mediane, also $\lceil \frac{1}{2} \lfloor \frac{n}{7} \rfloor \rceil$ Mediane, größer gleich sind als der Median of Medians. Mindestens die Hälfte der Elemente in einer Gruppe (das sind bei 7 genau 4) sind größer gleich deren Median. Also sind mindestens $4\lceil \frac{1}{2} \lfloor \frac{n}{7} \rfloor \rceil$ Elemente größer gleich der Median of Medians. Damit ist $|S_<| < n - 4\lceil \frac{1}{2} \lfloor \frac{n}{7} \rfloor \rceil < \lfloor \frac{11n}{14} \rfloor$ für $n \geq 50$. Symmetrisch ist damit auch $|S_>| < n - 4\lceil \frac{1}{2} \lfloor \frac{n}{7} \rfloor \rceil < \lfloor \frac{11n}{14} \rfloor$ für $n \geq 50$.
Folgende Laufzeitabschätzung lesen wir aus dem Algorithmus ab.

$$T(n) \leq \begin{cases} c_1, & \text{falls } n < 50 \\ c_2 n + T(\lfloor \frac{n}{7} \rfloor) + T(\lfloor \frac{11n}{14} \rfloor), & \text{falls } n \geq 50 \end{cases}$$

Wir benötigen $c_2 n$ Schritte für die Partitionierung, $T(\lfloor \frac{n}{7} \rfloor)$ Schritte um den Median der Mediane exakt zu bestimmen und müssen auf maximal $\frac{11n}{14}$ Werten im rekursiven Aufruf arbeiten.
Wir wollen zeigen, dass $T(n) \leq g(n)$ mit $g(n) = cn = O(n)$.

Induktionsanfang Angenommen $c_1 \leq c$, dann gilt $T(n) = c_1 \leq cn = g(n)$ für $n < 50$.

Induktionshypothese Angenommen es gilt $T(m) \leq cm$ für $m < n$.

Induktionsschritt Sei $n \geq 50$.

$$\begin{array}{rcll} T(n) & \leq & c_2 n + T(\lfloor \frac{n}{7} \rfloor) + T(\lfloor \frac{11n}{14} \rfloor) & \\ & \leq & c_2 n + c\lfloor \frac{n}{7} \rfloor + c\lfloor \frac{11n}{14} \rfloor & \text{(Induktionshypothese)} \\ & \leq & c_2 n + c\frac{n}{7} + c\frac{11n}{14} & \text{mit m = 1} \\ & = & c_2 n + cn - c\frac{n}{14} & \\ & = & cn + n(c_2 - c\frac{1}{14}) & \\ & \leq & cn & \text{falls } c_2 - c\frac{1}{14} \leq 0 \end{array}$$

Die Zusatzubedingung lässt sich wie folgt in eine untere Grenze für c umformen.

$$c_2 - c\frac{1}{14} \leq 0 \iff c_2 \leq c\frac{1}{14} \iff 14c_2 \leq c$$

Die Induktion funktioniert also für $max(c_1, 14c_2) \leq c$. Damit benötigt der Selektionsalgorithmus für 7er-Gruppen ebenfalls lineare Zeit.

Aufgabe 6 (Hashing): 2 + 3 + 2 = 7 Punkte

a) Wir speichern n Objekte in einer einfach verketteten Liste mit Schlüssel key[o] und Hashwert h(key[o]) für Objekte o. Die Schlüssel sind beliebig lange Zeichenketten mit minimal Länge d und die Hashwerte haben maximal Länge m mit $m << d$.

Wie können Sie die Hashwerte ausnutzen, um ein Objekt o in der Liste zu suchen? Begründen Sie ihre Antwort.

b) In dieser Aufgabe nehmen wir einfaches uniformes Hashing an. Das heißt für eine zufällige Eingabe x und eine Hashfunktion h ist jeder Hashwert für $h(x)$ gleich wahrscheinlich.

Berechnen Sie die erwartete Anzahl an Kollisionen beim Hashen der verschiedenen Werte $k_1, \ldots k_n$, das heißt berechnen Sie $\mathbb{E}(\ |\{\{k_i, k_j\} \mid k_i \neq k_j \text{ und } h(k_i) = h(k_j)\}|\ )$.

Hinweise:

- Beachten Sie, dass wir z.b. bei vier Elementen k_1, k_2, k_3, k_4 mit $h(k_1) = h(k_2) = h(k_3) = h(k_4)$ die 6 Kollisionen $\{k_1, k_2\}, \{k_1, k_3\}, \{k_1, k_4\}, \{k_2, k_3\}, \{k_2, k_4\}, \{k_3, k_4\}$ haben.
- Wegen einfachen uniformen Hashings einer Hashfunktion $h : A \to B$ folgt, dass für $x \neq y$ die Wahrscheinlichkeit, dass x und y gleich gehasht werden, uniform ist, also dass $\mathbb{P}(h(x) = h(y)) = \frac{1}{|B|}$.
- Nutzen Sie aus, dass der Erwartungswert linear ist, also dass $\mathbb{E}(\sum_{i=0}^{n} X_i) = \sum_{i=0}^{n} \mathbb{E}(X_i)$ gilt.

c) Angenommen wir benutzen offenes Hashing und wollen verhindern, dass bei einem zu hohen Füllgrad (= Anzahl der eingefügten Elemente geteilt durch die Größe der Hash-Tabelle) die Listen zu lang werden und dadurch die Suche in den Listen den Zugriffsaufwand dominiert.

Wir könnten dies erreichen, indem wir die Größe der Tabelle dynamisch anpassen, d.h. jedesmal, wenn der Füllgrad der Tabelle z.B. den Wert 2 übersteigt, erzeugen wir eine neue Tabelle mit doppelter Größe und überführen alle Element aus der alten Tabelle in die neue.

Ist dies eine sinnvolle Lösung?

Lösung

a) Da der Hashwert der Objekte in der Liste bereits gespeichert sind, können wir einmalig den Hashwert h(key[o]) der Eingabe o mit Schlüssel key[o] berechnen. Nun vergleichen wir die Hashwerte der Objekte in der Liste mit dem Hashwert der Eingabe. Sind die Hashwerte gleich, vergleichen wir anschließend noch deren Schlüssel. Damit reduzieren wir die Worstcase Laufzeitkomplexität der Suche in der Liste, da wir statt für jedes Objekt ein Vergleich mit Komplexität $O(d)$ ein Vergleich der Komplexität $O(m)$ in vielen Fällen vornehmen können.

b) Wir berechnen zuerst, mit wie vielen Elementen der Wert k_i kollidieren wird, das heißt wir berechnen $\mathbb{E}(\ |\{\{k_j, k_i\} \mid j > i \text{ und } h(k_j) = h(k_i)\}|\ )$ und addieren anschließend alle Erwartungswerte zusammen für den gewünschten Erwartungswert. Dies können wir tun, da $\{\{k_j, k_i\} \mid n \geq j > i \text{ und } h(k_j) = h(k_i)\}$ für jedes i eine Partition der Menge $\{\{k_i, k_j\} \mid k_i \neq k_j \text{ und } h(k_i) = h(k_j)\}$ darstellt.

Um $\mathbb{E}(\ |\{\{k_j, k_i\} \mid j > i \text{ und } h(k_j) = h(k_i)\}|\ )$ zu berechnen, addieren wir die Wahrscheinlichkeiten, dass k_i und k_j kollidieren, multipliziert mit der Anzahl der Kollisionen die k_i produziert, das heißt wir haben

$$\mathbb{E}(\ |\{\{k_j, k_i\} \mid n \geq j > i \text{ und } h(k_j) = h(k_i)\}|\ ) = \sum_{n \geq j > i} \mathbb{P}(h(k_j) = h(k_i)) \cdot 1 = \sum_{n \geq j > i} \mathbb{P}(h(k_j) = h(k_i)) \ .$$

für jedes i-te Element schaut man sich nur die Elemente mit höherem Index an

Die Wahrscheinlichkeit für $\mathbb{P}(h(k_i) = h(k_j))$ ist wegen der Annahme einfacher uniformem Hashings genau $\frac{1}{m}$, wobei m die Größe des Bildes von h ist, daher haben wir

$$\sum_{n \geq j > i} \mathbb{P}(h(k_j) = h(k_i)) = \sum_{n \geq j > i} \frac{1}{m} = \frac{n - i}{m} \ .$$

n = 3, m = 3
=> das 1. Element kollidiert mit Wahrscheinlichkeit 2/3 mit einem Element, das einen höheren Index hat
=> das 2. Element kollidiert mit Wahrscheinlichkeit 1/3 mit einem Element, das einen höheren Index hat
=> das 3. Element kollidiert mit Wahrscheinlichkeit 0 ...

Addieren wir nun alle Erwartungswerte zusammen haben wir

$$\begin{aligned} &\mathbb{E}(\ |\{\{k_i, k_j\} \mid k_i \neq k_j \text{ und } h(k_i) = h(k_j)\}|\ ) \\ &= \sum_{i=1}^{n} \mathbb{E}(\ |\{\{k_j, k_i\} \mid n \geq j > i \text{ und } h(k_j) = h(k_i)\}|\ ) \\ &= \sum_{i=1}^{n} \frac{n - i}{m} = \frac{n^2 - \frac{n(n+1)}{2}}{m} = \frac{n^2 - n}{2m} \ . \end{aligned}$$

c) In der Tat ist dies eine sinnvolle Idee. Durch das Einfügen würden wir nun eine Laufzeit von $2m$ für das $2m$te Element haben. Dies ist jedoch in Ordnung, da wir in den letzten m vielen Einfügeoperationen (so viele gab es bevor wir das letzte mal die Hashtabelle verdoppelt haben) immer sehr schnellen Zugriff hatten, kann diese Effizientverbesserung der letzten m Einfügeoperationen genutzt werden, um nun beim $2m$ten Element mehr Laufzeitkomplexität zu haben. Tatsächlich würde sich das ausgleichen: m viele Operationen in $O(1)$ und 1 Operation in $O(m)$ ergibt miteinander „verrechnet“ eine „verrechnete“ Laufzeit von $O(1)$ pro Operation.

Man kann das vorherige Argument auch formal in der so genannten Amortisierten Analyse[a] nutzen und bekommt bei diesem Verfahren tatsächlich eine erwartete amortisierte Laufzeit von $O(1)$ pro Operation.

Aufgabe 7 (Programmierung in Python - Binäre Suchbäume):

4 + 6 + 3 + 3 + 4 = 20 Punkte

Bearbeiten Sie die Python Programmieraufgaben. In dieser praktischen Aufgabe werden Sie sich mit Bäumen, genauer mit binären Bäumen und binären Suchbäumen, auseindersetzen. Diese Aufgabe dient dazu einige Konzepte der Vorlesung zu wiederholen.
Zum Bearbeiten der Programmieraufgabe können Sie einfach den Anweisungen des Notebooks *blatt06-python.ipynb* folgen. Das Notebook steht in der .zip-Datei zum Übungsblatt im Lernraum zur Vergügung.
Ihre Implementierung soll immer nach dem *# YOUR CODE HERE* Statement kommen. Ändern Sie keine weiteren Zellen.
Laden Sie spätestens bis zur Deadline dieses Übungsblatts auch Ihre Lösung der Programmieraufgabe im Lernraum hoch. Die Lösung des Übungsblatts und die Lösung der Programmieraufgabe muss im Lernraum an derselben Stelle hochgeladen werden. Die Lösung des Übungsblatts muss dazu als .pdf-Datei hochgeladen werden. Die Lösung der Programmieraufgabe muss als .ipynb-Datei hochgeladen werden.

Übersicht der zu bearbeitenden Aufgaben:

a) Binäre Suchbäume

- insert()
- is_binary_search_tree()
- bin_tree_2_list()
- list_2_bin_tree()
- bin_tree_2_bin_search_tree()

Lösung

Die Lösung der Programmieraufgaben finden Sie im Lernraum. Die Datei trägt den Namen *blatt06-python-solution.ipynb*.