Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn
P. Bamberger
P. Schneider

Algorithmen und Datenstrukturen
Sommersemester 2020
Musterlösung Übungsblatt 6

Abgabe: Mittwoch, 24.06.2020, 16:00 Uhr.

Aufgabe 1: Binärer Suchbaum - Range Queries *(10 Punkte)*

(a) Implementieren Sie die Datenstruktur des binären Suchbaumes (BST) und die `insert` Operation. Sie können dazu die Vorlage `BST.py` benutzen. *(4 Punkte)*

(b) Erweitern Sie Ihren BST um die Operation `getrange`(x_{min}, x_{max}) welche alle Schlüssel $x \in$ BST mit $x_{min} \leq x < x_{max}$ ausgibt. *(4 Punkte)*

Bemerkung: Ähnlich zu, aber nicht exakt gleich wie in Vorlesung Woche 6 Folie 21.

(c) Wir können auch Wörter über den Zeichen $\{a, \ldots, z\}$ bezüglich der *lexikographischen Ordnung*[1] in einen Binärbaum einfügen. Tun Sie dies für alle Wörter in der gegebenen Datei `inputs.txt`. Benutzen Sie diese Datenstruktur um alle Wörter mit einem bestimmten Präfix *effizient* auszugeben. Führen Sie als Unit-Test eine Suche nach allen Wörtern mit dem Präfix "qw" durch. Kopieren Sie die Ausgabe in Ihre `erfahrungen.txt`. *(2 Punkte)*

Musterlösung

Für die Teilaufgaben (a) und (b) siehe `BST.py` unserer Musterlösung. Für die Teilaufgabe (c) reichte es `getrange('qw','qx')` auf einen korrekten BST gefüllt mit den Wörtern aus `input.py` durchzuführen. Die gewünschte Ausgabe ist `['qwb', 'qwdjbcsm', 'qwel', 'qwgconj', 'qwgzykg', 'qwivkay', 'qwlybcn', 'qwmwwi', 'qwo', 'qwohudf', 'qwpoh', 'qwqrn', 'qwrmxd', 'qwtq', 'qwxpyjl', 'qwxrm', 'qwyiwh']`.

Aufgabe 2: Binärer Suchbaum - Operationen *(10 Punkte)*

(a) Beschreiben Sie eine Funktion, welche die Tiefe eines binären Suchbaums ausgibt, und analysieren Sie die Laufzeit. *(2 Punkte)*

(b) Beschreiben Sie eine Funktion, welche für einen gegebenen binären Suchbaum mit n Knoten und ein gegebenes $k \leq n$ eine Liste mit den k kleinsten Schlüsseln ausgibt. Analysieren Sie die Laufzeit in Abhängigkeit von k und der Tiefe d des Baumes. *(4 Punkte)*

(c) Beschreiben Sie eine Funktion, welche als Eingabe einen binären Suchbaum B und einen Schlüssel x erhält und folgende Ausgabe generiert:

- Falls es ein Element v in B gibt mit $v.key = x$, gebe v zurück.

[1] Python unterstützt Vergleiche von Wörtern bzgl. der lexikographischen Ordnung nativ.

- Andernfalls gebe ein Paar (u, w) zurück, wobei u das Baum-Element mit dem nächstkleineren und w das Element mit dem nächstgrößeren Schlüssel ist. Dabei soll $u =$ None falls x kleiner als alle Schlüssel im Baum ist und $w =$ None falls x größer als alle Schlüssel im Baum ist.

Die Beschreibung der Funktion kann in Pseudo-Code oder durch eine hinreichend genaue textliche Beschreibung erfolgen. Analysieren Sie die Laufzeit Ihrer Funktion. *(4 Punkte)*

Musterlösung

(a)

Algorithm 1 `return-closest`(x)

```
v ← find(x)
if v ≠ None then
    return v
else
    insert(x)
    (p, s) ← (pred(x), succ(x))
    delete(x)
    return (p, s)
```

Die benutzten Funktionen (`find`, `insert`, `pred`, `succ`) sind aus der Vorlesung bekannt und haben Laufzeit $\mathcal{O}(d)$ wobei d die Tiefe des Baums ist. Die Gesamtlaufzeit ist daher $\mathcal{O}(d)$.

(b) Die folgende Funktion berechnet die Tiefe des Teilbaums mit Wurzel v.

Algorithm 2 `depth`(v)

```
if v = None then
    return -1   ▷ depth of a childless node must be 0, hence we define the depth of None as -1
else return max(depth(v.left)+1, depth(v.right)+1)
```

Um die Tiefe des Baums zu berechnen, rufen wir `depth(`r`)` auf, wobei r die Wurzel des Baums ist. Die Laufzeit beträgt $\mathcal{O}(n)$, da wir für jeden Knoten einen rekursiven Aufruf machen und jeder rekursive Aufruf kostet $\mathcal{O}(1)$.

Alternativ können wir auch ein BFS machen, was $\mathcal{O}(n)$ Zeit benötigt. Für den Knoten v, der als letztes besucht wurde, rufen wir dann folgende Funktion auf:

Algorithm 3 `traverse-up`(v)

```
d ← 0
while v.parent ≠ None do
    d ← d + 1
    v ← v.parent
return d
```

Dies benötigt $\mathcal{O}(d)$. Die Gesamtlaufzeit beträgt also $\mathcal{O}(n+d) = \mathcal{O}(n)$ (da $d \leq n$).

(c) Wir adaptieren die in-order Traversierung aus der Vorlesung:

Algorithm 4 `inorder_variant`(node) ▷ *Assume K is given globally, intially empty*

```
if node ≠ None then
    inorder_variant(node.left)
    if |K| ≥ k then
        return
    K.append(v.key)
    inorder_variant(node.right)
```

Sei K die Menge von k Knoten, die die k kleinsten Schlüssel im BST darstellen. Offensichtlich muss die In-Order-Traversierung alle Knoten in K einmal besuchen. Entsprechend der Vorlesung

fügt `inorder_variant`(root) die Schlüssel in aufsteigender Reihenfolge zu K hinzu. Somit enthält K am Ende die k kleinsten Schlüssel.

Sei A die Menge der Knoten die nicht in K sind aber auf die dennoch ein rekursiver Aufruf gestartet wird. Da die Rekursion abbricht sobald alle Knoten in K besucht sind, entspricht die Menge A genau den Knoten die Vorfahren von mindestens einem Knoten in K sind, sich aber nicht in K selbst befinden. Die Laufzeit ist also $\mathcal{O}(|A| + |K|)$ da ein rekursiver Aufruf (ohne Unteraufrufe) $\mathcal{O}(1)$ Zeit kostet.

Per Definition haben wir $|K| = k$, es bleibt also die Größe von A zu bestimmen. Wir behaupten, dass sich alle Knoten in A auf einem Pfad von der Wurzel zu einem Blatt befinden, d.h., $|A| \leq d$. Dies ist der Fall, wenn in A keine zwei Knoten vorhanden sind, sodass keiner der Vorfahren des anderen ist.

Nehmen wir für einen Widerspruch an, dass zwei solche Knoten u, v existieren, so dass weder u Vorfahr von v ist, noch umgekehrt. Angenommen (ohne Verlust der Allgemeinheit), dass $\texttt{key}(u) \leq \texttt{key}(v)$. Das heißt, u befindet sich im linken und v befindet sich im rechten Teilbaum eines gemeinsamen Vorfahren a von u und v.

Per Definition hat v einen Knoten $w \in K$ in seinem Teilbaum. Da sich v im rechten Teilbaum und u im linken Teilbaum von a befindet, haben wir $\texttt{key}(w) \geq \texttt{key}(u)$ und w hat eine höhere In-Order-Position. Aber dann hätten wir auch $u \in K$, ein Widerspruch zu $u \in A$.