Datenstrukturen und Algorithmen
Heimübung 7

Eli Kogan-Wang (7251030)
David Noah Stamm (7249709)
Daniel Heins (7213874)
Tim Wolf (7269381)

28. Mai 2022

Aufgabe 1

Damit Prev(x) und Succ(x) in $\mathcal{O}(1)$ liegen, muss man eine zusätzliche Eigenschaft für alle Knoten im Baum definieren.
Jeder Knoten x im Baum erhält zusätzlich eine Verweis auf die Adresse seines Nachfolgers und Vorgängers mit N_x := die Adresse des Nachvolgers von x und V_x := die Adresse des Vorgängers von x
Damit dies Eigenschaft aufrecht erhalten wird müssen die Einfügen- und Löschenfunktion abgeändert werden.

Algorithm 1 EinfügenNeu(T,z)

```
y ← nil
x ← root[T]
while x ≠ nil do1
    y ← x
    if key[z] < key[x] then
        x ← lc[x]
    else
        x ← rc[x]
p[z] ← y
if y = nil then
    root[t] ← z
    V[z] ← nil
    N[z] ← nil
else
    if key[z] < key[y] then
        lc[y] ← z
        N_{V_y} ← z
        V_z ← V_y
        V_y ← z
        N_z ← y
    else
        rc[y] ← z
        V_{N_y} ← z
        N_z ← N_y
        N_y ← z
        V_z ← y
```

Algorithm 2 LöschenNeu(T,z)

if $lc[z] = nil \vee rc[z] = nil$ **then**
  $y \leftarrow z$
else
  $y \leftarrow N_z$
if $V_y \neq nil$ **then**
  $N_{V_y} \leftarrow N_z$
if $N_y \neq nil$ **then**
  $V_{N_y} \leftarrow V_z$
if $V_z \neq nil$ **then**
  $N_{V_z} \leftarrow N_z$
if $N_z \neq nil$ **then**
  $V_{N_z} \leftarrow V_z$
if $lc[y] \neq nil$ **then**
  $x \leftarrow lc[y]$
else
  $x \leftarrow rc[y]$
if $x \neq nil$ **then**
  $p[x] \leftarrow p[y]$
if $p[y] = nil$ **then**
  $root[T] \leftarrow x$
else
  if $y = lc[p[y]]$ **then**
    $lc[p[y]] \leftarrow x$
  else
    $rc[p[y]] \leftarrow x$
if $y \neq z$ **then**
  $key[y] \leftarrow key[z]$
Return y

Algorithm 3 Pred(x)

Return $key[V_x]$

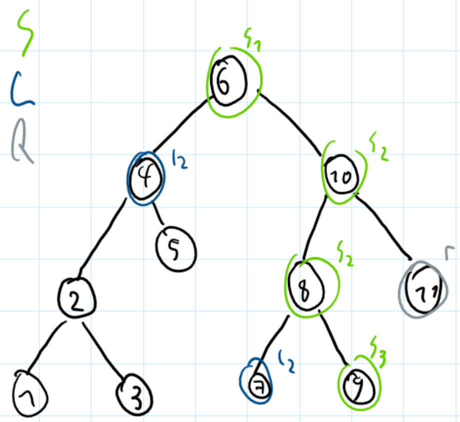
Algorithm 4 Succ(x)

Return $key[N_x]$

Aufgabe 2

Nr. 2)

Gegenbeispiel:



$$\Rightarrow \left.\begin{array}{l} S = \{s_1, s_2, s_3, s_4\} \\ L = \{l_1, l_2\} \\ R = \{r_1\} \end{array}\right\} \Rightarrow key(s_1) < key(l_2) < key(r_1)$$

$\Downarrow$

↯ Behauptung

$\Downarrow$

Aufgabe 3

a) Es wird Inorder-Tree-Walk(x), mit Vertauschung von lc[x] mit rc[x] und statt einer Ausgabe wird list-insert (L,x) aufgerufen.
Dadurch wird eine absteigende Sortierung der Schlüssel an list-insert übergeben wodurch eine aufsteigend sortierte Link-list entsteht. Zum Beginn des Algorithmus wird x auf root[T] gesetzt und die Liste L ist leer.

Algorithm 5 bin-Search-to-linked-list(T,L,x)

```
1: if x ≠ nil then
2:     bin-Search-to-linked-list(T,L,rc[x])
3:     List-Insert(L,x)
4:     bin-Search-to-linked-list(T,L,lc[x])
```

b) Z.z Der Algorithmus ist Korrekt $\iff$ Die Elemente in L sind aufsteigend sortiert.
o.B.d.A Ist dies der Fall, falls List-insert die Adressen der Elemente des Baumes und die Elemente in einer absteigenden sortierten Reihenfolge übertragen werden.
$\Rightarrow$ Es ist nur noch zu zeigen, dass die Elemente des Baumes in einer absteigenden sortierten Reihenfolge an List-Insert übertragen werden. (Das die Adressen dieser übertragen werden gilt nach Zeile 3).
Beweis durch Vollständige Induktion über die Anzahl der Element eines Baumes mit Hilfe der Suchbaumeigenschaft.
Sei T_n:= ein beliebiger binärer Suchbaum mit n Elementen
IA: n = 0: Trivial
n = 1: Trivial
n = 2: Trivial
IV: Es gelte: Der Algorithmus überträgt die Elemente eines Baumes T_{n-1} in einer absteigenden sortierten Reihenfolge korrekt.
IS: Z.Z Der Algorithmus überträgt die Elemente eines Baumes T_n in einer absteigenden sortierten Reihenfolge korrekt.
Es gilt: $T_n = T_{n-1}$ + Einfügen (T_{n-1},n)
Nach IV gilt bereits das alle Elemente ohne n Korrekt ausgegeben werden.
Folglich ist nur noch zu Zeigen, das n an der richtigen Stelle ausgeben wird.
Dies ist, aber trivialerweise der Fall, da beim Einfügen die Suchbaumeigenschaft erhalten bleibt.
$\Rightarrow$ n wird an der richtigen Stelle ausgeben. Nachdem Induktionsprinzip gilt, dass für einen Baum mit n elementen, die Elemente des Baumes in einer absteigenden sortierten Reihenfolge an List-Insert übertragen werden.
$\Rightarrow$ Der Algorthmus ist Korrekt. $\square$

c) bin-Search-to-linked-list ist eine abgeänderte Variante des aus der Vorlesung bekannten Algo. Inoder-Tree-Walk.
Es gibt zwei Unterschiede zwischen bin-Search-to-linked-list und Inoder-Tree-Walk, welche aber keinen Einfluss auf die Laufzeit haben.
Zum einen wurde rc[x] durch lc[x] ausgetauscht. (Dies hat trivialerweise keinen Einfluss auf die Laufzeit.) Zum anderen wurde return key[x] durch List-Insert(L,x) ausgetauscht. Diese haben beide eine Laufzeit von $\mathcal{O}(1)$ (Nach Lemma 10.10) Folglich bleibt die Laufzeit erhalten.
Da Inoder-Tree-Walk(x) nach Folie 45 eine Laufzeit von $\theta(n)$, hat bin-Search-to-linked-list folglich auch eine Laufzeit $\theta(n)$