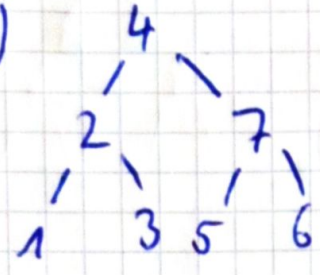
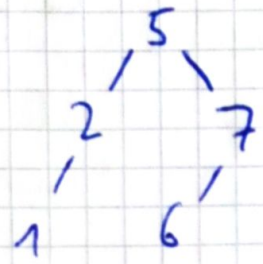
8 + 6 + 6½ + 10 = 30½ / 32

5.1a)

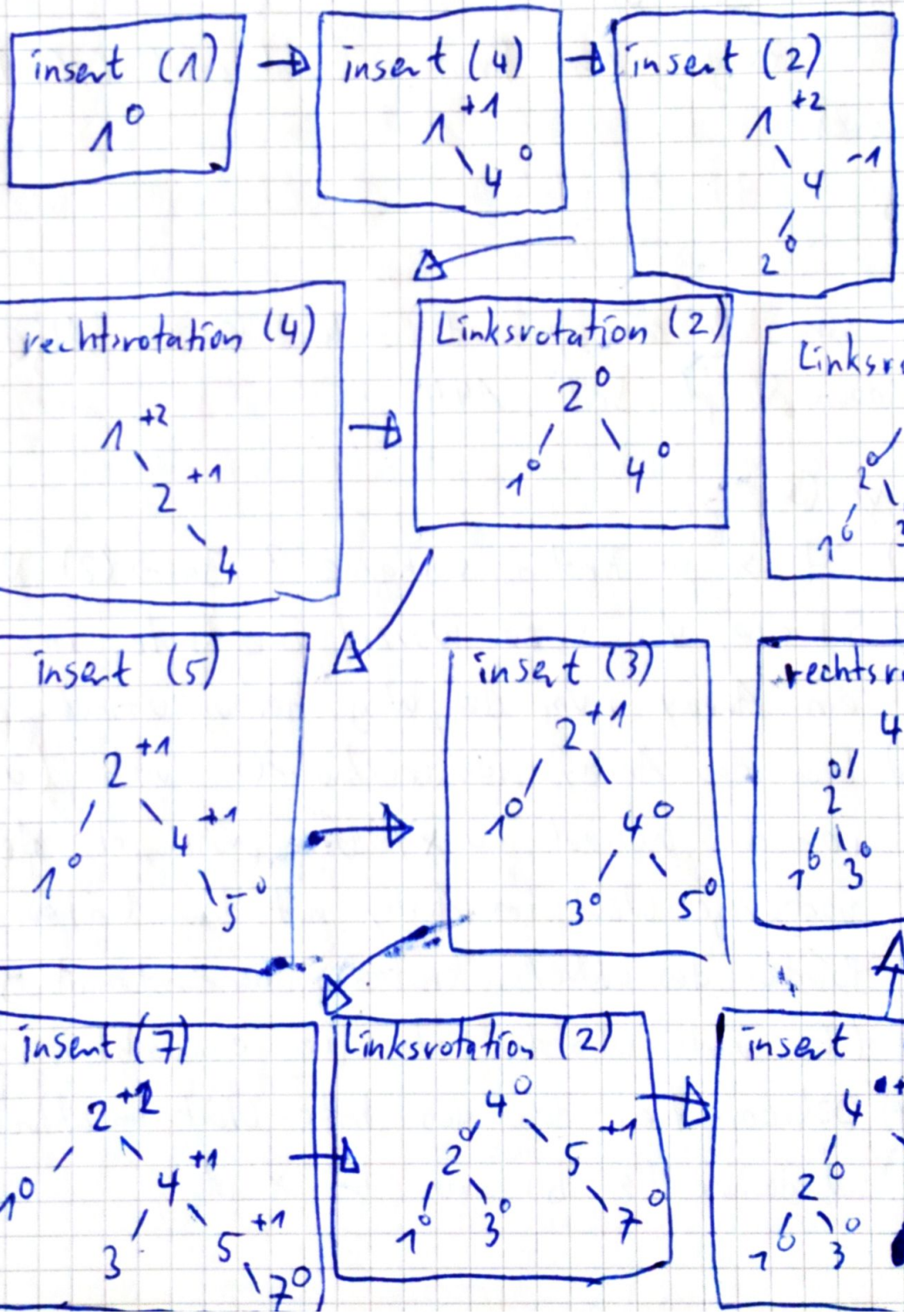


Nach insert (4)
" (2)
" (7)
" (1)
" (3)
" (5)
" (6)



Nach remove (3)
" (4)

b)



5.2a)
```
insert (int schluessel, int info)
{ ... while (...)
  { Eltern = Zeiger;
    if ( x < Zeiger -> schluessel)
       { Zeiger -> info += 1;
         Zeiger = Zeiger -> links;
       }
    else { Zeiger = Zeiger -> rechts; }
  }
  if (Zeiger == 0)
     { Zeiger = new Knoten (schluessel, 0, 0, 0);
     }
  :
}
```

Dies sind die anzupassenden Zeilen für den Algorithmus aus dem Skript, Seite 144.

remove (x)

(1) ist x im Knoten vorhanden? Ja → (2) Nein → return
(2) Schreibe von x zur Wurzel alle Knoten in ein Array, um den Weg heraus zu finden
(3) Lese das Array nun rückwärts durch (also von der Wurzel zu x) und prüfe, ob der vorherige Wert im Array mit dem linken Pointer des aktuellen Knotens übereinstimmt. Ja → (4) Nein → (5)
(Es wurde also der linke Weg eingeschlagen)
(4) Dekrementiere dann den Info-Wert im aktuellen Knoten
(5) Wiederhole (3) bis man bei x ist.

b) select (k) {
    find (root, k)
}

find (Bsbaum current, int rank){
    if (current → info == rank) return current;
    if (current → info > rank) find (current → links , rank);
    if (current → info < rank)
        find (current → rechts , rank - current→info - 1);
}

Die Idee hierbei ist, sich an der Anzahl der Knoten im linken Teilbaum zu orientieren.

Die Rekursion stammt von de Seite algs4.cs.princeton.edu

5.3a) Die Laufzeit ist im Worst-Case O (Tiefe (T))

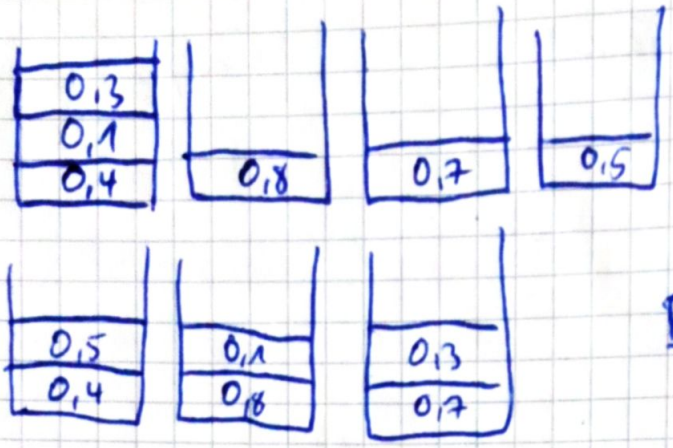
join (T1, T2) { Idee? — 1/2 P.

    Knoten root = new Knoten ;
    root = Linkestes (T2); delete_Linkestes (T2);
    root → Links = T1. root
    root → rechts = T2. root
}

Linkestes (Bsbaum T){
    while (T → links) T = T → links;
    return T;
}

delete_Linkestes (Bsbaum T){
    if (Linkestes (T) → rechts == null) remove (linkestes (T));
    else { linkestes (T). Elter → rechts = linkestes → rechts;
        remove linkestes (T); } }

5.4 a)



Worst-Fit

Best-Fit

b) Idee zum Worst-Fit:

Verwendung der Datenstruktur Max-Heap.
Die Schlüssel im Heap stehen für die Restkapazität der Behälter.
An der Wurzel steht also stets der Behälter mit dem größten freien Rest.
Für jedes Element, das nun eingefügt wird, wird ein passender Behälter aus dem Heap genommen ($O(\log(n))$). Dieser Behälter wird entfernt ($O(\log(n))$).
Danach wird ein neuer Schlüssel mit der Differenz von dem einst freien Platz und der eingefügten Menge eingefügt ($O(\log(n))$)
Sollte kein passender Behälter gefunden werden, so wird ein neuer Knoten eingefügt, der (1 - Kapazität) Restkapazität hat. ($O(n \cdot \log(n))$)
Beim Finden eines passenden Behälters beträgt die Laufzeit $O(3 \cdot n \cdot \log(n)) \triangleq O(n \log(n))$)

ii) Idee zum Best-Fit:

Verwendung der Datenstruktur AVL-Baum

Die Schlüssel im Baum stehen für die Restkapazität der Behälter.

Die Suche nach einem geeigneten Behälter verläuft wie folgt:

Man vergleicht den einzufügenden Wert mit de Wurzel. Ist der Wert kleiner als der in der Wurzel, vergleicht man den einzufügenden Wert mit dem Wert des linken Kindes der Wurzel. Ist diesen Wert größer als der einzufügende Wert, so ist die Wurzel der kleinste geeignete Behälter.

Ist der Wert der Wurzel größer als der einzufügende Wert, wird das oben beschriebene Verfahren rekursiv auf das rechte Kind der Wurzel angewendet.

Da im Allgemeinen bei jedem Durchlauf des Schverfahrens die Anzahl der verbleibenden Behälter mindestens halbiert wird, ist die Laufzeit $O(\log(n))$

Dieser Behälter wird dann entfernt ($O(\log(n))$)

Danach wird ein neuer Knoten eingefügt ($O(\log(n))$), der als Schlüssel den Wert des entfernten Knotens minus den Wert des einzufügenden Wertes hat.

Sollte das rechteste Kind im AVL-Baum kleiner als der einzufügende Wert sein, wird ein neuer Knoten eingefügt ($O(\log(n))$)

Im schlimmsten Fall hat der Algorithmus eine Laufzeit von $O(3 \cdot n \cdot \log(n)) \leq O(n \cdot \log(n))$

5.3b) split (T_1, x)

initialisiere Leere Binäre-Suchbäume T_1, T_2

aktuellerKnoten = T.wurzel

Solange (aktuellerKnoten → links || aktuellerKnoten → rechts)

falls (aktuellerKnoten → Wert ≥ x)

insert (aktuellerKnoten, T_2)

hilf = aktuellerKnoten → links

aktuellerKnoten → links = null

aktuellerKnoten = hilf // hier wird die Wurzel mit dem rechten Teilbaum in T_2 eingefügt

falls (aktuellerKnoten → Wert < x)

insert (aktuellerKnoten, T_1)

hilf = aktuellerKnoten → rechts

aktuellerKnoten → rechts = null

aktuellerKnoten = hilf // hier wird die Wurzel mit linkem Teilbaum in T_1 eingefügt

falls (aktuellerKnoten → links != 0 && aktuellerKnoten → rechts != 0)

falls (aktuellerKnoten → Wert ≥ x)

insert (aktuellerKnoten, T_2)

sonst (insert (aktuellerKnoten, T_1))

// Die Idee hinter dem Algorithmus ist, alle Knoten, die sich rechts vom rechtesten Knoten, dessen Wert echt-kleiner x ist, in T_2 packen. Alle anderen Knoten landen in T_1.

Korrekt?

-1p.