

# Algorithmen und Datenstrukturen SoSe25

## -Assignment 4-

Moritz Ruge

Matrikelnummer: 5600961

Lennard Wittenberg

Matrikelnummer: —

Mai 2025

## Problem 1: Rot-Schwarz Bäume und (2,4)-Bäume

In Aufgabe 3 auf dem 3. Aufgabenblatt wurden rot-schwarz Bäume definiert.

a) Zeigen Sie: Rot-schwarz Bäume und (2, 4)-Bäume sind äquivalent. Genauer: es gibt eine lokale Transformation, welche Gruppen von Knoten im rot-schwarz Baum in Knoten im (2, 4)-Baum überführt, und umgekehrt. Geben Sie eine solche Transformation an, und begründen Sie, dass Ihre Transformation die Bedingungen an rot-schwarz Bäume und an (2, 4)-Bäume erfüllt.

In einem (2,4)-Baum gilt:

min: 2 Kindknoten und max: 4 Kindknoten

min Einträge: 1 und max Einträge: 3

Bei der Convertierung eines (2,4)-Baums in einen Rot-Schwarz-Baum müssen betrachtet werden:

k: Elternknoten; w: Kinderknoten

Convertierung:

1-Knoten  $\rightarrow$  rot-schwarz: Bei einem Knoten mit nur 1. Eintrag, wird der Knoten zu einem Schwarzknoten.  $\rightarrow$

Ein 1-Knoten hat entweder 0 oder 2 Kindknoten

– 0 Kindknoten: Schwarzer Blattknoten

– 2 Kindknoten: für den 1-Knoten gilt:  $v = (w_1, k_1, w_2)$ , wobei  $w_1 < k_1 < w_2$  –  $> w_1$  wird zum linken Kindknoten von  $k_1$  und  $w_2$  wird zum rechten Kind von  $k_1$ . Die Farbe der Kindknoten wird vom Elternknoten bestimmt.

2-Knoten  $\rightarrow$  red-black: Bei einem Knoten mit 2 Einträgen, wird 1 Knoten zum Elternknoten und der andere Knoten wird zum Kindknoten

– Dabei gilt:  $v = (k_1, k_2)$ , wobei  $k_1 < k_2 \rightarrow$  Daraus entstehen zwei Möglichkeiten:

(i)  $k_1$ : Elternknoten und  $k_2$ : rechter Kindknoten

(ii)  $k_2$ : Elternknoten und  $k_1$ : linker Kindknoten

– Ein 2-Knoten hat 3 oder 0 Kindknoten. Für die Verteilung der Kindknoten gilt:

(i) Wenn  $v$  keine Kindknoten hat, dann ist  $v$  ein Blattknoten. Somit gilt für die Kinder von  $v$ , dass sie zu schwarzen Blattknoten werden, nachdem sie in den Rot-Schwarz-Baum überführt wurden.

(i)  $v = (w_1, k_1, w_2, k_2, w_3)$

Zur Veranschaulichung:

```

    ' ' '
      |A|B|
    C  D  E
    ' ' '
  
```

– Für  $v$  gibt es wieder zwei Möglichkeiten:

– es gilt  $w_1 < \dots < k_2 < w_3 \rightarrow$  Größenordnung von links nach rechts

(1.)  $k_1$  wird zum Elternknoten:

$w_1$ : linker Kindknoten von  $k_1$

$k_2$ : rechter Kindknoten von  $k_1$

$w_2$ : linker Kindknoten von  $k_2$

$w_3$ : rechter Kindknoten von  $k_2$   
 (2.)  $k_2$  wird zum Elternknoten:  
 $k_1$ : linker Kindknoten von  $k_2$   
 $w_3$ : rechter Kindknoten von  $k_2$   
 $w_1$ : linker Kindknoten von  $k_1$   
 $w_2$ : rechter Kindknoten von  $k_1$   
 – Die Farbe der Kindknoten wird vom Elternknoten bestimmt.

3-Knoten – > red-black: Bei einem Knoten mit 2 Einträgen gilt:  $v = (k_1, k_2, k_3)$ , dabei wird  $k_2$  zum Schwarzen Elternknoten,  $k_1$  zum Roten linken Kindknoten von  $k_2$  und  $k_3$  zum rechten Kindknoten von  $k_2$ .

– Ein 3-Knoten hat 4 oder 0 Kindknoten. Für diese gilt:  
 –  $w_1 < k_1 < w_2 < k_2 < w_3 < k_3 < w_4$   
 – Daraus folgt:

$w_1$ : linker Kindknoten von  $k_1$   
 $w_2$ : rechter Kindknoten von  $k_1$   
 $w_3$ : linker Kindknoten von  $k_3$   
 $w_4$ : rechter Kindknoten von  $k_3$

– Wenn  $v$  keine Kindknoten hat, dann ist  $v$  ein Blattknoten. Somit gilt für die Kinder von  $v$ , dass sie zu schwarzen Blattknoten werden, nachdem sie in den Rot-Schwarz-Baum überführt wurden.  
 – Die Farbe der Kindknoten wird vom Elternknoten bestimmt.

Wurzel:

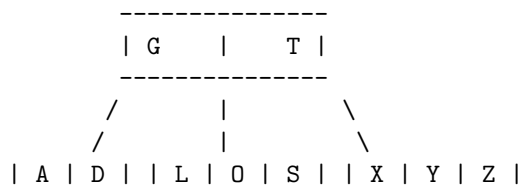
– Die Wurzel verhält sich je nach Baum, genau wie ein Innerer Knoten oder ein Blatt Knoten und ist immer Schwarz. Eine spezielle Betrachtung ist daher nicht nötig. Die Farbwahl aller Knoten in einem in Rot-Schwarz überführten (2,4)-Baum geht von der Wurzel aus.

Unter Anwendung dieser Regeln, lässt sich jeder (2,4) Baum in einen Rot-Schwarz-Baum überführen.

Beispiel: (2,4)-Baum aus der Aufgabe 2b -> Rot-Schwarz-Baum

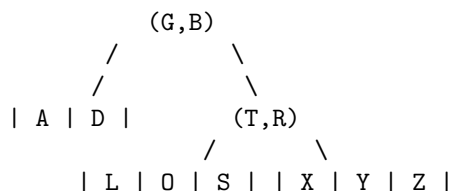
Ich stelle Knoten im Rot-Schwarz-Baum als Tupel (key,colour)

“ “ “

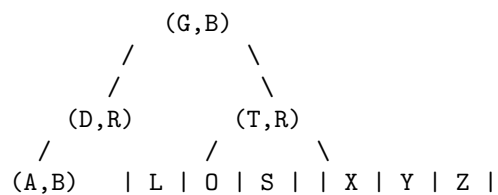


“ “ “

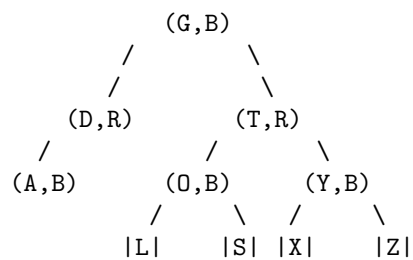
“ “ “ 1. Wurzel -> Rot-Schwarz-Knoten Wähle G als Wurzel im Rot-Schwarz-Baum



““2. Linkes Kind von G -> Rot-Schwarz-Knoten Wähle D als Elternknoten von A.

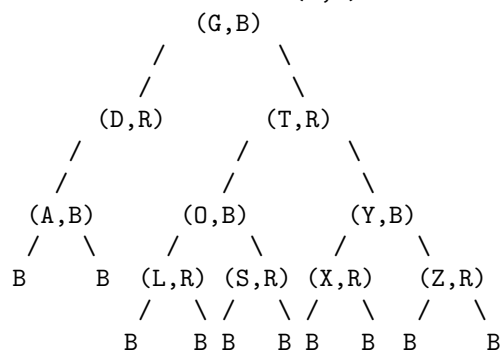


““3. Linkes und Rechtes Kind von R -> Rot-Schwarz-Knoten Wähle O als Elternknoten von L,S und Y a



““

““4. Wandle verbliebende (2,4)-Blätter in Rot-Schwarz-Blätter um.



““

## Problem 2: (2,3)-Bäume und (2,4)-Bäume

a) Fügen Sie die Schlüssel A, L, G, O, D, T, S, X, Y, Z in dieser Reihenfolge in einen anfangs leeren (2, 3)-Baum ein. Löschen Sie sodann die Schlüssel Z, A, L. Zeichnen Sie den Baum nach jedem Einfüge- und Löschvorgang, und zeigen Sie die Modifikation, welche durchgeführt werden.

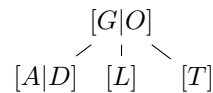
Ein (2,3)-Baum hat folgende Grenzen:

- min children: 2, max children: 3
- min entries: 1, max entries: 2
- A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z

1. insert(A):

[A]

⇒ Das rechte Blatt [L|O|T] hat zu viele Einträge → Rebalance



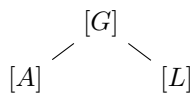
2. insert(L):

[A|L]

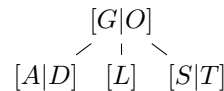
3. insert(G):

[A|G|L]

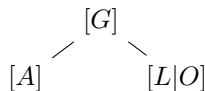
⇒ Die Wurzel hat zu viele Einträge → Split-



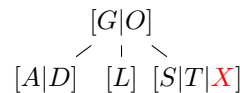
7. insert(S):



4. insert(O):

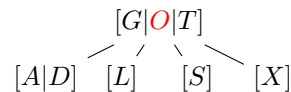
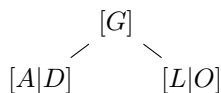


8. insert(X):



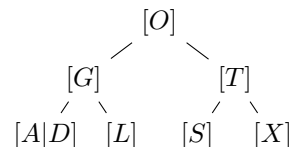
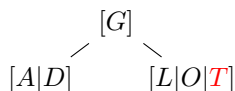
⇒ Das rechte Blatt [S|T|X] hat zu viele Einträge → Rebalance

5. insert(D):

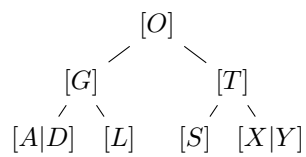


⇒ Die Wurzel hat zu viele Einträge & zu viele Kinder → Rebalance

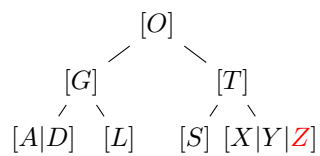
6. insert(T):



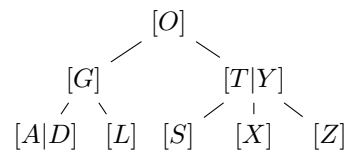
9. insert(Y):



10. insert(Z):



⇒ Der Knoten  $[X|Y|Z]$  hat zu viele Einträge → Balancieren



11. remove(Z):

12. remove(A):

13. remove(L):

b) Wiederholen Sie die Teilaufgabe (a) mit einem (2, 4)-Baum.

min children: 2, max children: 4

min entries: 1, max entries: 3

A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z

insert(A):

```

-----
| A |
-----

```

insert(L):

```

-----
| A | L |
-----

```

insert(G):

```

-----
| A | G | L |
-----

```

insert(O): first split

```

-----
| G |
-----
 /      \
| A |      | L | O |

```

insert(D):

```

-----
| G |
-----
 /      \
| A | D |  | L | O |

```

insert(T): second split

```

-----
| G | O |
-----
 /      |      \
| A | D |  | L |  | T |

```

insert(S):

```

-----
| G | O |
-----
 /      |      \
| A | D |  | L |  | S | T |

```

insert(X):

```

-----
| G | O |

```

```

      -----
     /      |      \
    /      |      \
   | A | D | | L |   | S | T | X |
insert(Y): rebasing the root -> third split

```

```

      -----
     | G      |      T |
      -----
    /      |      \
   /      |      \
  | A | D | | L | O | S | | X | Y |
insert(Z):

```

```

      -----
     | G      |      T |
      -----
    /      |      \
   /      |      \
  | A | D | | L | O | S | | X | Y | Z |
'''

```

Starting tree:

```

      -----
     | G      |      T |
      -----
    /      |      \
   /      |      \
  | A | D | | L | O | S | | X | Y | Z |

```

delete(Z): delete from leaf

```

      -----
     | G      |      T |
      -----
    /      |      \
   /      |      \
  | A | D | | L | O | S | | X | Y |

```

delete(A): delete from leaf -> min 1 key required -> condition holds true

```

      -----
     | G      |      T |
      -----
    /      |      \
   /      |      \
  | D |      | L | O | S | | X | Y |

```

delete(L): delete from Leaf -> 2 keys pressed -> node requirements satisfied ->  $G < O$  and  $S < T$  -> Order condition satisfied.

```

      -----
     | G      |      T |
      -----

```



| D |     /     |     |     \     |  
| O | S |     | X | Y |

### Problem 3: (a,b)-Bäume

a) Beschreiben Sie, wie man in einem (a, b)-Baum mit  $n$  Schlüsseln die Operation  $\text{succ}(k)$  implementieren kann. Was ist die Laufzeit?

⇒  $\text{succ}(k)$  - finde den Nachfolge vom Schlüssel  $k$

1. Suche den Knoten " $i$ ", der den Eintrag  $k$  enthält
2. Wenn  $k$  nicht das größte Element in  $u$  ist, schaue ob es noch ein Teilbaum zwischen dem Element  $k$  und seinem direkten Nachfolger gibt.
  - Wenn Nein → dann gib den direkten Nachfolger von  $k$  zurück
  - Wenn ja, gehe in den Teilbaum und gebe das kleinste Element zurück
3. Wenn  $k$  das größte Element in  $u$  ist:
  - wenn  $i$  ein rechten Teilbaum besitzt, gehe in den rechten Teilbaum und gebe das kleinste Element zurück
  - ansonsten gehe zum Elternknoten und suche das erste Element, das größer als  $k$  ist und gebe es zurück

⇒ Die Laufzeit beträgt  $O(\log n)$ , da die Höhe eines (a,b)-Baums  $O(\log n)$  ist.

b) Beschreiben Sie, wie man in einem (a, b)-Baum mit  $n$  Schlüsseln die Operation  $\text{findRange}(k_1, k_2)$  implementieren kann, die alle Schlüssel  $k$  liefert, für die  $k_1 \leq k \leq k_2$  ist. Die Laufzeit soll  $O(\log n + s)$  betragen. Dabei ist  $s$  die Anzahl der gelieferten Schlüssel.

`findRange(k_1, k_2):`

Sei  $T$  ein (a,b)-Baum mit Knoten  $v$ .

Jeder Knoten besteht aus einem oder mehreren keys  $k_i$ , dabei sind die keys innerhalb eines Knoten von links(kleinsten) nach rechts(größter) key sortiert.

Jeder key hat zusätzlich einen Verweis (ab hier pointer genannt) auf seine Kinder- und Elternknoten.

Pseudocode `findRange(k_1, k_2):`

- Wenn gilt:  $k_1 == k_2$  → wenn die gesuchte Reichweite nur einen key beinhaltet gibt die Position des keys mittels `get(k_2)` zurück, wenn sie existiert.

`return get(k_2)`

- erstelle eine leere Liste: `found_keys = []`

- erstelle einen counter für keys: `left_boarder = k_1`

- Finde den key  $k_i \geq k_1$  mit der Funktion `get(left_boarder)`, um die Linke Grenze zu bestimmen. `get(left_boarder)` ist notwendig, falls  $k_1$  nicht Teil des (a,b)-Baumes ist.

- solange kein  $k_i$  gefunden wurde wiederhole diesen Funktionsaufruf, bis ein  $k_i$  gefunden wurde. Inkrementiere Dabei vor jeder nächsten Iteration den counter `left_boarder`.

```

- Wenn gilt:  $k_i == k_2$  -> dann liegt die gesuchte Reichweite nicht im Baum
return err

- Wenn gilt:  $k_i >= k_1$  -> hänge den key der Liste an und verlasse den Loop.
found_keys.append(k_i)
break

- // Diese Suche hat im worst-case eine Zeitkomplexität von  $O(b \cdot \log_a(n))$ .  $O(\text{get}) = O(\log(n))$ 
und dies muss maximal  $b$ -mal wiederholt werden, da der startkey  $k_i$  minimal
der kleinste key innerhalb eines Knotens  $v$  sein kann. Spätestens nach der  $b$ -ten
Inkrementierung ist der key  $k_i == \text{Elternkey des Knotens } v$ , in dem sich  $k_i$  zu Beginn
befindet und dieser ist größer als  $k_1$ .
- Nun kann man iterative linear alle keys, für die gilt:  $k_j = \text{found\_nodes}[0]$ ,
 $k_j \leq k_2$ , an die Liste der gefundenen keys anhängen.
- für Knoten  $v$  gilt:  $v = (k_1, \dots, k_l)$ 
- solange  $k_j \leq k_2$ :
- solange  $k_j \leq k_l$  -> hänge  $k_j$  and die Liste der gefundenen keys an.
found_keys.append(k_j)

-  $k_j = \text{Elternkey von } v$ 
- return found_keys // gibt die Liste aller keys zurück, die gefunden wurden
found_keys[0] =  $k_1$  und  $\text{found\_keys}[\text{len}(\text{found\_keys})-1] = k_2$ 
- Die Zeitkomplexität dieser Suche ist  $O(s \cdot \log(s))$ , wobei  $s$  die Anzahl der
besuchten keys ist.  $s$ -mal, da insgesamt  $s$  keys betrachtet wurden und
 $\log(s)$ -mal, da innerhalb eines bestimmten Knoten nur ein bestimmter Teil von  $s$ 
betrachtet wird.
Die Dominate Klasse ist linear  $O(s)$ 
- Für die Gesamt Zeitkomplexität gilt:  $O(b \cdot \log_a(n) + s)$ , weil die beiden Teile
der Funktion sequentiell ablaufen und nicht ineinander verschachtelt sind.

```

c) Seien  $T_1$  und  $T_2$  zwei  $(a, b)$ -Bäume, und sei  $S_1$  die Schlüsselmenge von  $T_1$  und  $S_2$  die Schlüsselmenge von  $T_2$ . Sei  $x$  ein weiterer Schlüssel. Alle Schlüssel in  $S_1$  sind kleiner als  $x$ , und alle Schlüssel in  $S_2$  sind größer als  $x$ . Beschreiben Sie eine Operation join, die aus  $T_1$ ,  $T_2$  und  $x$  einen  $(a, b)$ -Baum für die Schlüsselmenge  $S_1 \cup x \cup S_2$  erzeugt. Die Laufzeit sollte  $O(b \log_{\max} |S_1|, |S_2|)$  betragen. Hinweis: Betrachten Sie zunächst den Fall, dass  $T_1$  und  $T_2$  die gleiche Höhe haben. Achten Sie darauf, dass hinterher die  $(a, b)$ -Baum Eigenschaften wieder hergestellt werden.