

Algorithmen und Datenstrukturen - Hausübung 09

Gruppenmitglieder

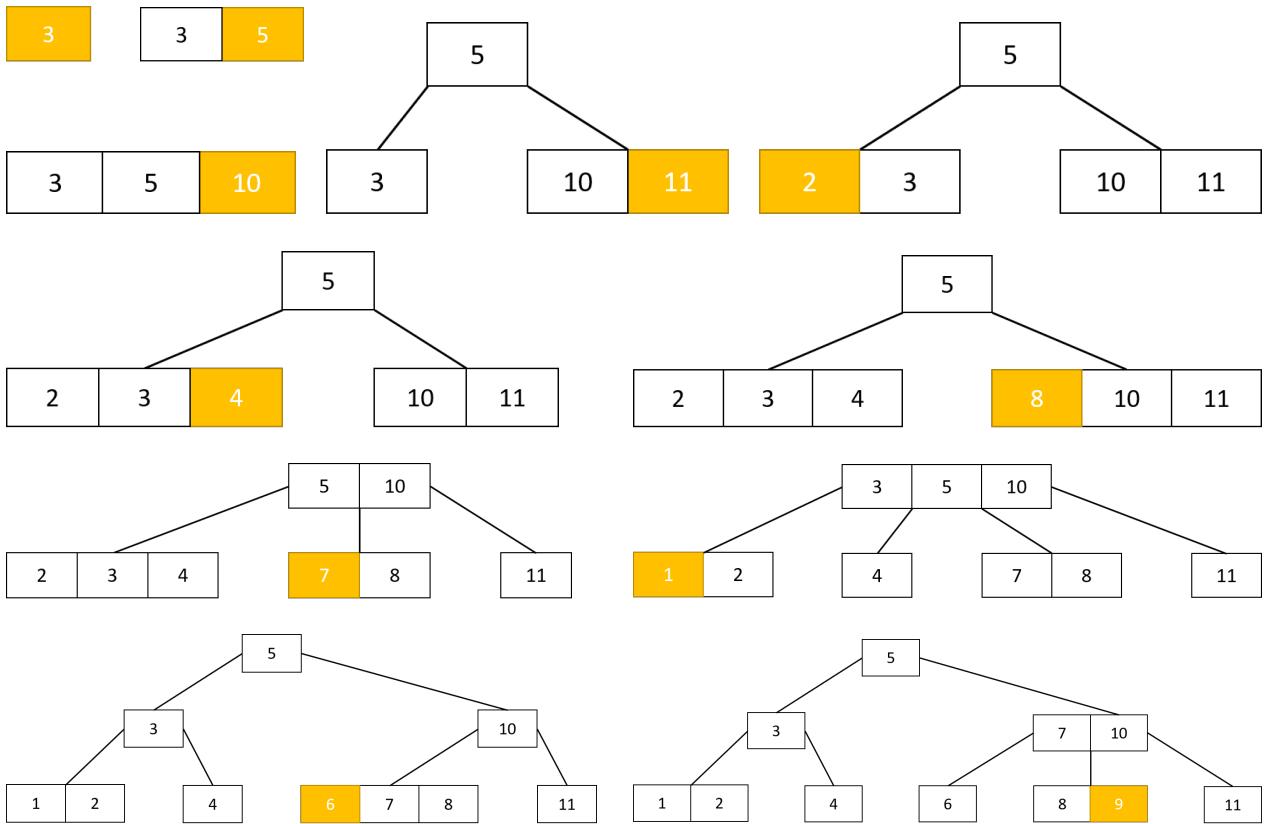
- Emre Berber (2957148)
- Christoph Berst (2743394)
- Jan Braun (2768531)

Inhaltsverzeichnis

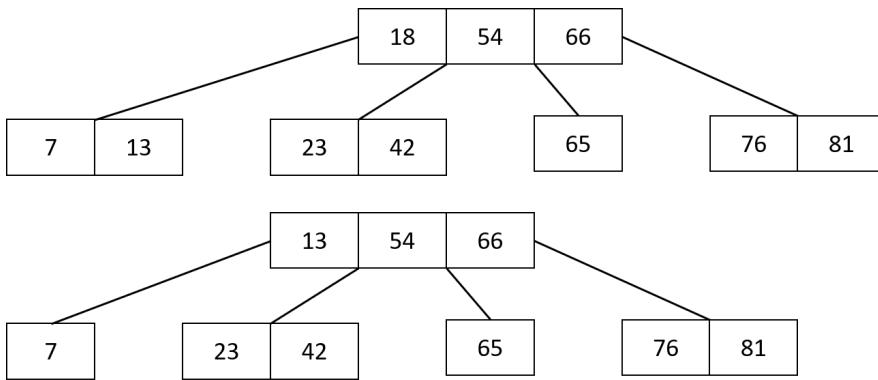
H1	1
a)	1
b)	1
c)	1
d)	2
H2	2
H3	2
a)	2
i)	2
ii)	2
iii)	2
b)	3
i)	3
ii)	3

H1

a)



b)



c)

	Einfügen	Löschen
Invariante:	Jeder Knoten hat maximal $2t - 1$ Werte.	Jeder Kindknoten auf Pfad des zu löschenen Knotens hat mindestens t Werte.
Beweis:	Jedes Mal, wenn der Einfüge-Algorithmus auf einen Knoten mit $2t - 1$ Werten stößt, wird dieser aufgesplittet, sodass dieser und seine Kinder definitiv weniger als $2t - 1$ Werte besitzen.	zu besuchende Kindknoten mit nur $t - 1$ Werten werden mit direkten Geschwistern, die nur $t - 1$ Werte haben, verschmolzen. Falls ein Geschwister mehr als $t - 1$ Werte besitzt, wird der Wert in Richtung dem zu besuchenden Kind rotiert.

d)

Beim **Einfügen** würde splitten nicht mehr ausreichen, da aus $2t - 1$ Werten mindestens ein Knoten mit $t - 1$ Knoten entsteht. Nur der Knoten, bei dem die Suche nach der Position zum Einfügen hätte eine Chance t Werte zu besitzen. Der beim splitten entstandene Elternwert würde dann einem Knoten mit sowieso schon t Werten hinzugefügt werden, womit das splitten häufiger auftreten könnte. Was mit dem Knoten geschieht, bei dem die Suche nicht weitergeht, ist ungewiss. Wenn man ihn mit $t - 1$ lässt, hält er sich unerlaubt in Baum auf. Wenn ihn über den neu entstandenen Elternwert rotiert, wird der Baum unvollständig.

Auf das **Löschen** hat dies im Prinzip keine Auswirkung siehe c) Invariante Löschen.

H2

"to be filled in"

H3

a)

i)

```
Labyrinth(L,S,Z)
1   P = S
2   ε = 1
3   WHILE d(P,Z) ≠ 0
4       FOR x=0 TO L.length-1
5           New(NL)
6           IF P ≠ L[x] ∧ A ≠ L[x] ∧ d(P,X) ≤ ε
7               Add(NL,L[x])
8           IF NL.length == 0
9               P = A
10              BREAK
11          A = P
12          IF NL.length == 1
13              P = NL[0]
14          ELSE
15              FOR n=0 TO NL.length-1
16                  IF d(P,Z) > d(NL[n],Z)
17                      P = NL[n]
18                  IF A == P
19                      P = NL[RandomnextInt(N.length)]
20      setPosition(F,P)
```

- NL ist die Liste der Nachbarn des aktuellen Position P der Figur F, welche durch Auswertung der x und y Koordinaten der in ihr gespeicherten Punkte, gegen den Uhrzeigersinn sortiert ist.
- $d(L_1, L_2)$ ist die Distanz-Funktion d aus der Aufgabenstellung, wobei $L_1, L_2 \in L^2$
- Unser Qualitycheck findet durch Zeile 6 und Zeile 13 statt: Es wird versucht eine Nachbarposition mit einem geringeren Abstand zu finden
- Es wird versucht nicht auf den letzten Position rückkehren zu können, um somit in Sackgassen nicht stecken zu bleiben, in denen man sonst immer wieder gegen die Wand laufen würde.
- $\text{RandomnextInt}(m)$ gibt einen Wert $r \in [0, m) \subseteq \mathbb{N}$ zurück.
- $\text{setPosition}(F, P)$ setzt die Figur F auf den Punkt P.

ii)

(5,1) → (5,2) → (4,2) → (4,3) → (4,4) → (3,4) → (3,5) → (2,5) → (1,5)

iii)

Während der Bearbeitung der i) haben wir unbemerkt unseren Algorithmus schon verbessert, in dem wir speichern auf welchem Feld er sich vorher befand (Variable A). Damit versuchen wir zu verhindern, dass die Figur andauernd versucht in Sackgassen ständig wieder zur Wand zu laufen. Sobald er an der Wand ist, sucht der Algorithmus nach einem Feld, welches immer noch nah genug ist, aber nicht das Feld ist auf dem er herkam. Somit versucht er vom lokalen Maximum zu dem globalen Maximum über qualitativ schlechtere Felder zu gelangen.

b)

i)

```
Bergsteiger(L,f,S)
1   best = S
2   ε = 1
3   REPEAT
4       S = best
5       R = getBiggestNeighbor(L,S,f,ε)
6       IF f(R) > f(S)
7           best = R
8   UNTIL S == best
9   RETURN best
```

Die Funktion $\text{getBiggestNeighbor}(L, S, f, \varepsilon)$ sucht einen Wert $X \in L \times L$, dessen wobei $d(S, X) \leq \varepsilon$ ist. All diese Werte gegen den Uhrzeigersinn durchgegangen in mit Hilfe von der Qualitätsfunktion f wird der Größte/Beste unter ihnen bestimmt.

Unser Algorithmus versucht ständig zum größtmöglichen Nachbarn zu gelangen bis der aktuelle Punkt keinen größeren Nachbarn mehr hat. Dieser Punkt ist dann ein lokales Maximum in der Funktion f .

ii)

$$S_0 : f((-2, -4)) = 0 \rightarrow f((-2, -4)) = 0$$

$$S_1 : f((4, 4)) = 0 \rightarrow f((3, 3)) = 0.094 \rightarrow f((2, 2)) = 1.508 \rightarrow f((1, 2)) = 3.0625$$