Algorithmen und Datenstrukturen - Hausübung 09

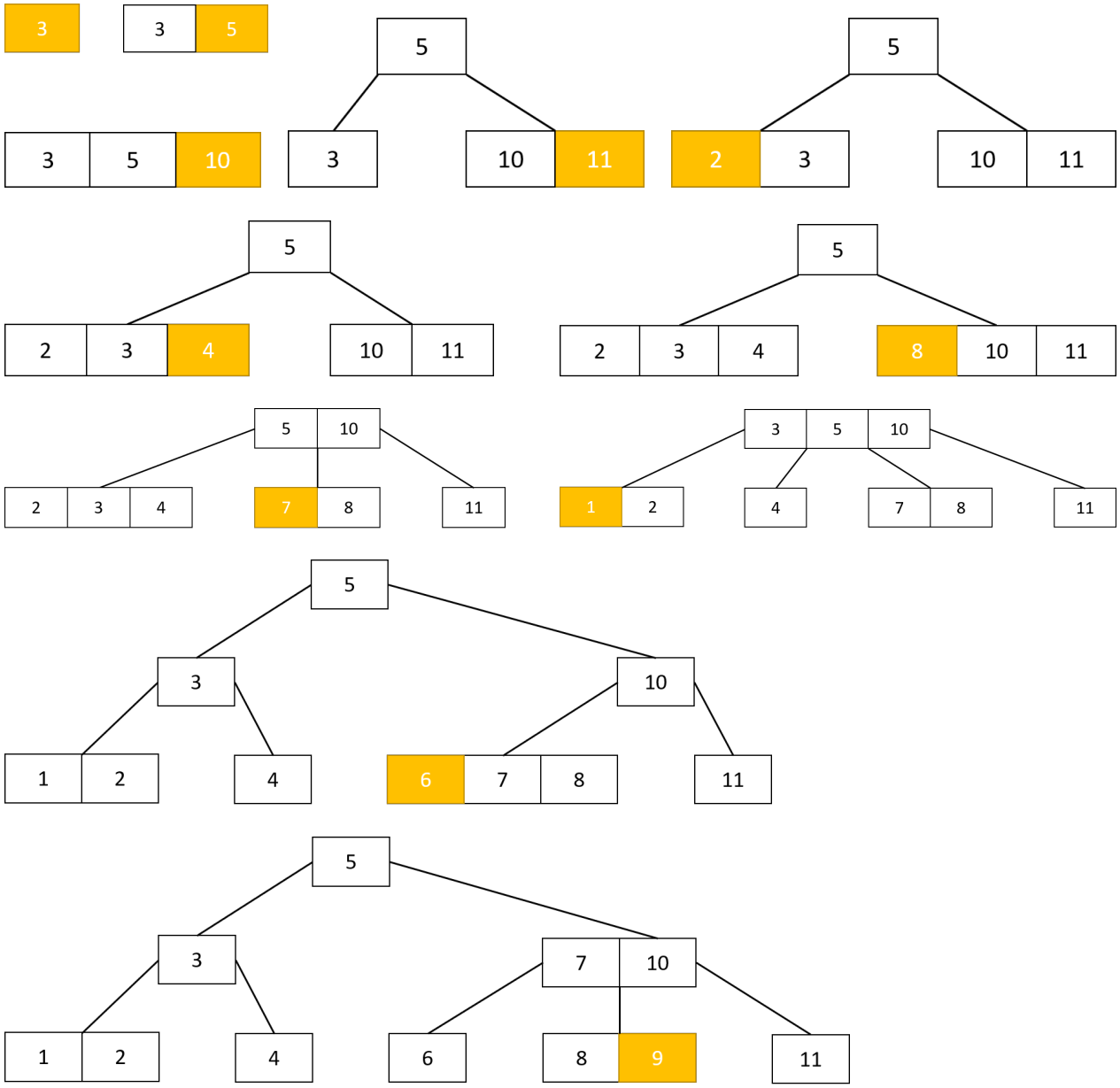
Gruppenmitglieder

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

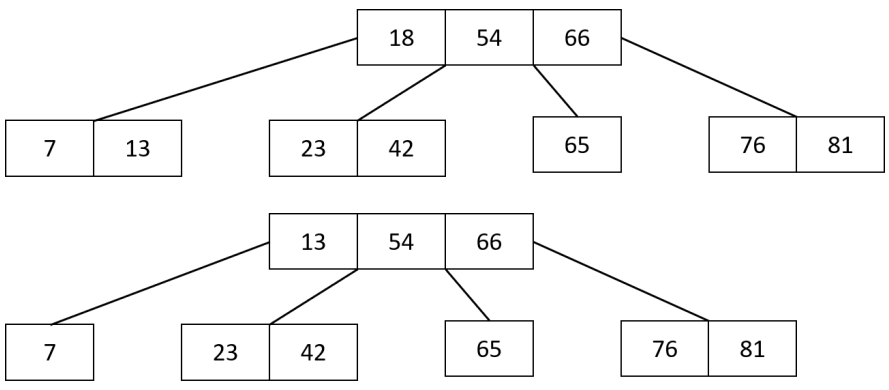
Inhaltsverzeichnis

H1

a)


3
3 5
3 5 10
5
3
10 11
5
2 3
10 11
5
2 3 4
10 11
5
2 3 4
8 10 11
5 10
2 3 4
7 8
11
3 5 10
1 2
4
7 8
11
5
3
10
1 2
4
6 7 8
11
5
3
7 10
1 2
4
6
8 9
11

b)


18 54 66
7 13
23 42
65
76 81
13 54 66
7
23 42
65
76 81

c)

	Einfügen	**Löschen**
Invariante:	Jeder Knoten hat maximal $2t-1$ Werte. Bis Auf die Wurzel hat jeder mindestens $t-1$.	Jeder Kindknoten auf Pfad des zu löschenden Knotens hat mindestens $t-1$ und maximal $2t-1$ 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 $2t-1$ Werten besitzen.	Falls ein zu besuchendes Kind nur $t-1$ Werte haben sollte und deren direkten Geschwister auch $t-1$ haben, werden diese mit dem Elternwert verschmolzen, sodass sich maximal/genau $2t-1$ Werte in einem Knoten befinden. Falls nun eines der Geschwister mehr Werte besitzt, wird ein Wert von ihm zum zu besuchenden rotiert, sodass dieses bei mindestens $t-1$ Werten bleibt.

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 neuentstandenen Elternwert rotiert, wird der Baum unvollständig.

Beim **Löschen** wird besonders das Verschmelzen problematisch, da Knoten mit $2t$ Werten immer noch nicht erlaubt sind und andernfalls durch das Löschen Knoten mit weniger als t Werten entstehen würden.

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 akuellen Position P der Figur F, welche durch Auswertung der x und y Koordinaten der in ihr gespeicherten Punkte, gegen den Uhrzeigersinn sortiert ist.
- d(L1,L2) ist die Distanz-Funktion d aus der Aufgabenstellung, wobei L1, L2 $\in L^2$
- Unser Qualitycheck findet durch Zeile 6 und Zeile 13 statt: Es wird versucht eine Nachbarposition mit einem geringern Abstand zu finden
- Es wird versucht nicht auf den letzte Position rückkehren zu können, um somit in Sackgassen nicht stecken zu bleiben, in denen man sonst immer wieder gegen die Wand laufen würde.
- RandomNextInt(m) gibt einen Wert r $\in [0, m) \subseteq \mathbb{N}$ zurück.
- 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 einen 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 getBiggestNeighbor(L,S,f,ε) 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 Qualiätsfunktion 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)

$$S_0: \quad f((-2,-4)) = 0 \quad \rightarrow \quad f((-2,-4)) = 0$$
$$S_1: \quad f((4,4)) = 0 \quad \rightarrow \quad f((3,3)) = 0.094 \quad \rightarrow \quad f((2,2)) = 1.508 \quad \rightarrow \quad f((1,2)) = 3.0625$$

H2

```
subsetN(M,q)
1   IF N.length == q
2       RETURN N
3   ELSE
4       FOREACH m in M
5           IF !contains(N,m)
6               add(N,m)
7           ELSE
8               continue
9           IF subsetN(M,q) == false
10              continue
11          ELSE
12              RETURN N
13      RETURN FALSE
```

```
contains(N,m)
1   FOREACH n in N
2       IF getX(m) == getX(n)
3           RETURN true
4       IF getY(m) == getY(n)
5           RETURN true
6       IF getZ(m) == getZ(n)
7           RETURN true
8   RETURN false
```