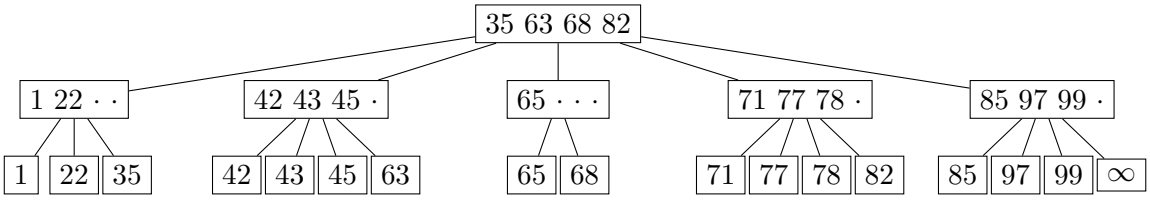
Übungsblatt 8

Abgabe via Moodle.
Deadline Fr. 30ter Juni

Aufgabe 1 ((a,b)-*Bäume,* 6 *Punkte*)

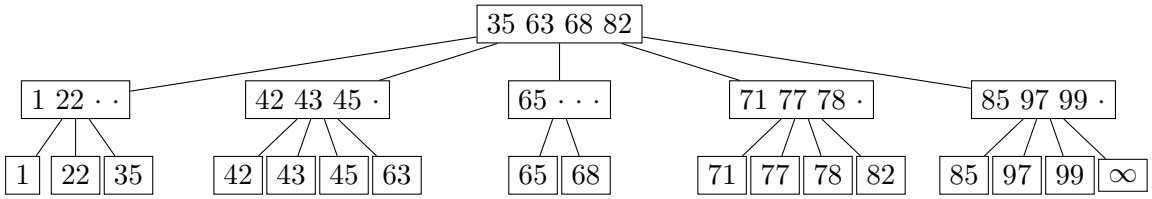
Führen Sie auf dem folgenden $(2,5)$-Baum die geforderten Operationen in der angegebenen Reihenfolge durch. Zeichnen Sie den Endzustand des Baums nach jeder der angegebenen Operationen!



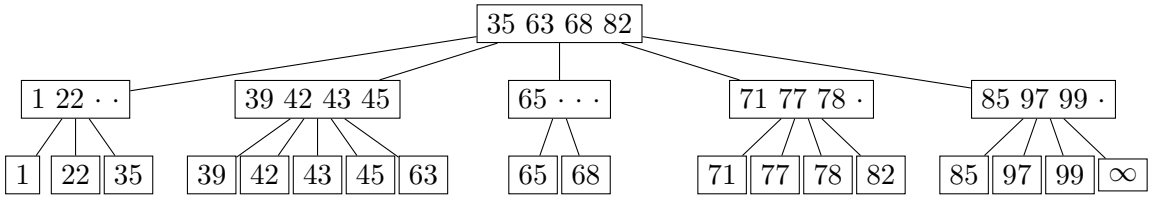
1. Einfügen von 39, Einfügen von 70, Einfügen von 67
2. Löschen von (wiederum vom Ausgangszustand): 42, Löschen von 63, Löschen von 77

Lösung:

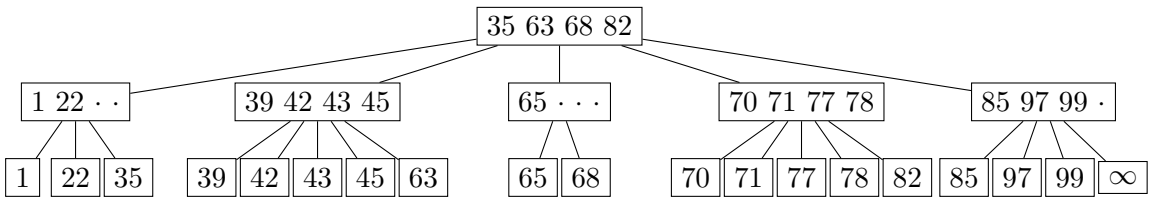
1. Originalzustand



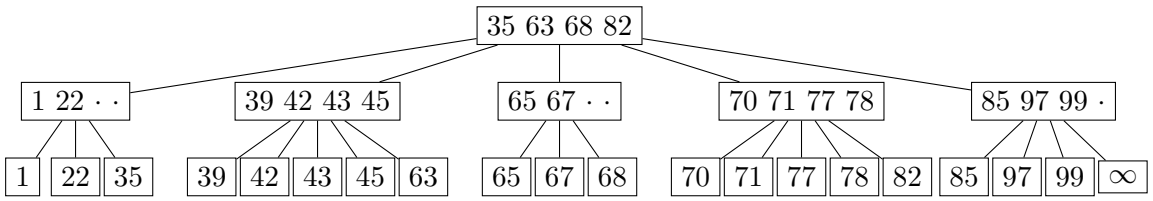
nach Einfügen von 39



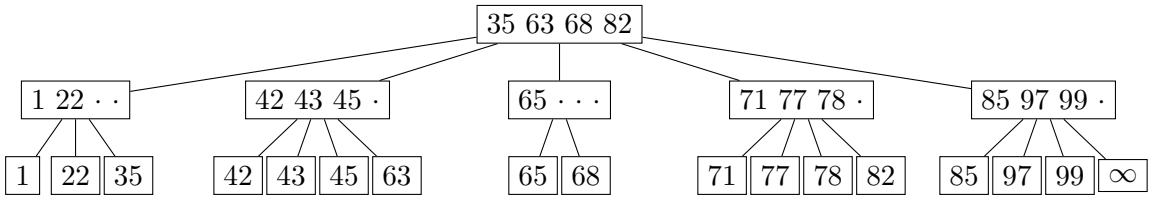
nach Einfügen von 70



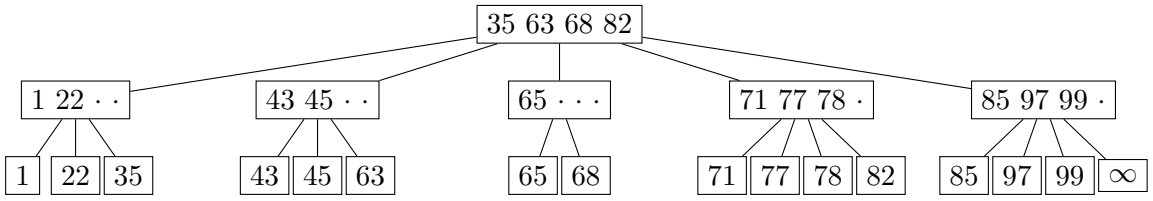
nach Einfügen von 67



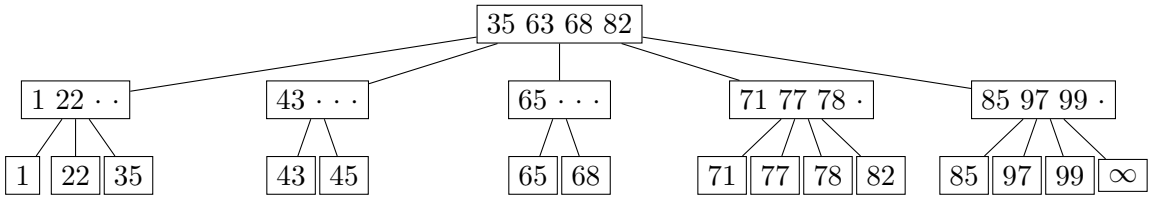
2. Originalzustand



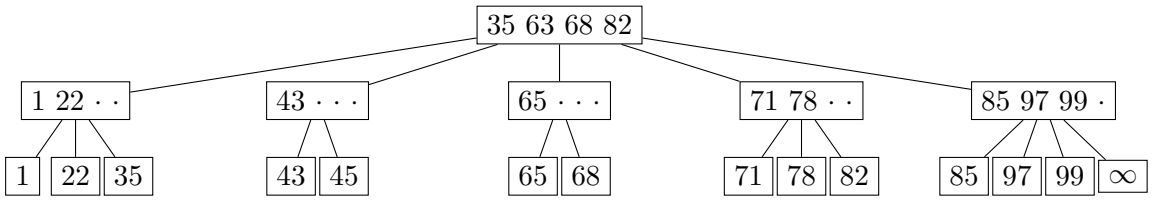
nach Löschen von 42



nach Löschen von 63



nach Löschen von 77



Aufgabe 2 **((a,b)-*Bäume,* $6+2$ *Punkte*)**

Betrachten Sie (a,b)-Bäume, die eine Operation *rank*(k: *Key*) unterstützen. Diese liefere die Position des ersten Elements mit Schlüssel k in der jeweiligen sortierten Liste (also den *Rang* von k) oder $\perp$ falls k nicht in der Liste vorkommt.

1. Geben Sie einen Algorithmus an, der *rank*(k) in $\mathrm{O}(\log n)$ Zeit berechnet (dabei sei n die Anzahl der Elemente in der sortierten Liste). Modifizieren Sie die (a,b)-Bäume gegebenenfalls in geeigneter Weise. Verwenden Sie **Pseudocode.**

2. Begründen Sie kurz, warum Ihr Verfahren das gewünschte Laufzeitverhalten aufweist.

Lösung:

1. Die Blätter bekommen zusätzlich eine Operation *size*, die immer den Wert 1 zurückliefert, wenn das gespeicherte Element einen Wert $\neq \infty$ enthält und 0 sonst.

   Die inneren Knoten sehen aus wie auf den Vorlesungfolien haben aber eine zusätzliche Variable *size*, die die Anzahl der Datenelemente speichert, die sich in der sortierten Teilliste unter dem jeweiligen Knoten befinden):

   ```
   1: class ABNodeWithSize is a ABNode
   2:     size: ℕ≥0
   3:     invariant size = Σ_{j=1}^{d} c[j] → size
   4: end class
   ```

   Der Algorithmus:

   ```
   1: rank(k: Key, root: Handle of ABNodeWithSize): ℕ≥0 ∪ {⊥}
   2:     return rank_rec(k, 0, root)
   ```

   Der eigentliche rekursive Algorithmus:

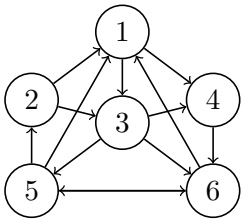
   ```
   1: rank_rec(k: Key, counter′: ℕ≥0, u: Handle of ABNodeWithSize): ℕ≥0 ∪ {⊥}
   2:     counter := counter′: ℕ≥0
   3:     for i = 1..d do
   4:         invariant k kommt nicht vor or
   5:             counter < Rang von k ≤ counter + Σ_{j=i}^{min{ℓ | u→S[ℓ]≥k}} u → c[j] → size
   6:         if u → S[i] < k then
   7:             counter := counter + u → c[i] → size
   8:             continue
   9:         end if
   10:        assert i = min{ℓ | u → S[ℓ] ≥ k}
   11:        assert k kommt nicht vor or counter < Rang von k ≤ counter + u → c[i] → size
   12:        if u → c[i] ist Blatt then
   13:            assert c[i] = 1
   14:            assert counter < Rang von k ≤ counter + 1
   15:            if key(u → c[i] → e) = k then return counter + 1
   16:            else return ⊥
   17:        end if
   18:        return rank_rec(k, counter, u → c[i])
   19:    end for
   ```

2. Jeder Aufruf der Rekursion verursacht (ohne Tochteraufrufe) maximal $\mathrm{O}(b) = \mathrm{O}(1)$ Zeit. Die Anzahl dieser Aufrufe liegt in O (Pfadlänge) für den Pfad von der Wurzel zu dem Blatt mit dem ersten Vorkommen von k. Diese Pfadlänge liegt bei (a, b)-Bäumen jedoch stets in $\mathrm{O}(\log n)$.

Aufgabe 3 (*Graphen,* $1 + 1 + 6 + 2$ *Punkte*)

Der zu einem gerichteten Graph $G = (V, E)$ **transponierte Graph** ist der Graph $G^T = (V, E^T)$ mit $E^T = \{(v, u) \in V \times V \mid (u, v) \in E\}$. In G^T sind also gegenüber G die Richtungen der aller Kanten vertauscht.

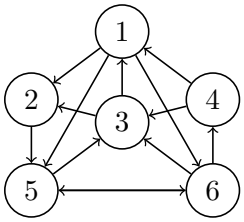
1. Geben Sie die Adjazenzfeldrepräsentation des rechts abgebildeten gerichteten Graph in Form der Felder V und E an.
2. Zeichnen Sie G^T des rechts abgebildeten Graphen.
3. Geben Sie einen Algorithmus mit Laufzeit $\mathrm{O}(n+m)$ an, der zu einem Graph in Adjazenzfeld Darstellung seinen transponierten Graphen G^T in Adjazenzfelddarstellung bestimmt. Geben Sie Pseudocode an!
4. Begründen Sie die Laufzeit des Algorithmus.



Lösung:

1.

V	1	3	5	8	9	12	14
E	3	4	1	3	4	5	6	6	1	2	6	1	5

2. 

3. Der Algorithmus zur Erzeugung des **transponierten Graphen** funktioniert folgendermaßen: man beobachtet zunächst, dass die Eingangsgerade G gerade die Ausgangsgrade im transponierten Graphen sind. Um das neue Knotenarray aufzubauen benötigt man die Ausgangsgerade der Knoten in G. Die Eingangsgerade der Knoten im Eingabegraphen kann man in Linearzeit durch einfaches Iterieren über das Kantenarray berechnen. Mit diesen Informationen baut man dann das Knotenarray des transponierten Graphen bottom up auf. Dazu addiert man immer den Ausgangsgrad vom Vorgänger zum Vorgänger Startzeiger im Kantenarray und erhält damit den eigenen Startzeiger im Kantenarray, d.h. den Eintrag im Array V.
Die alten Quellknoten werden beim Drehen der Kanten zu neuen Zielknoten und andererseits werden die alten Zielknoten zu neuen Quellknoten. Dies kann man ausnutzen, um das neue Kantenarray aufzubauen.

```
1: function TRANSPOSE(V[1..n+1], E[1..m])
2: Ṽ = ⟨1, 0, ..., 0⟩ : Array[1..n+1] of ℕ
3: Ẽ = ⟨0, 0, ..., 0⟩ : Array[1..m] of ℕ
4: d_in = ⟨0, ..., 0⟩ Array[1..n] of ℕ
5: //compute indegrees to build Ṽ later (indegrees become outdegrees in G^T)
6: for e := 1 to m do
7:    d_in[E[e]] + +
8: end for
9: //build Ṽ
10: for i := 2 to n + 1 do
11:    Ṽ[i] = Ṽ[i − 1] + d_in[i − 1]
12: end for
13: //build Ẽ
14: for s := 1 to n do
15:    for e := V[s] to V[s + 1] − 1 do
16:       // old sources become new targets, old targets become new sources
17:       otarget = E[e]
18:       osource = s
19:       d_in[otarget] − −
20:       Ẽ[Ṽ[otarget] + d_in(otarget)] = osource
```

21: **end for**
22: **end for**
23: **return** $(\tilde{V}, \tilde{E})$

4. Der einzige kritische Punkt sind die Zeilen 14-22. Es handelt sich dabei um eine Traversierung des Graphen, bei dem für jeden Knoten alle Nachbarn angeschaut werden. Der Aufwand dieser Schleifen ergibt sich folgendermaßen: $\sum_{s=1}^{n} \sum_{e=V[s]}^{V[s+1]-1} c = c \sum_{s=1}^{n} d_{\text{out}}(s) = \mathrm{O}(|E|) = \mathrm{O}(m)$. Insgesamt ergibt sich damit ein Aufwand von $\mathrm{O}(n+m)$.

Aufgabe P8 (*Adjazenzfeldrepräsentation, optional*)

Für die praktischen Übungen verwenden wir die Plattform `www.hackerrank.com`. Hier müssen Sie sich registrieren um an den Übungen teilzunehmen. Unter dem Link

`https://www.hackerrank.com/adsi-2023`

finden die praktischen Übungen in der Form eines Programmierwettbewerbs statt.

In der achten Challenge geht es um die Adjazenzfeldrepresentation. Ihnen wird ein Graph als Eingabe gegeben. Dieser wird zunächst als Adjazenzliste gespeichert.
Der erste Teil Ihrer Aufgabe besteht darin in *build_adjacency_array_representation* diesen Graphen mittels Adjazenzfeldrepräsentaion (in $G.V$ und $G.E$) darzustellen. Im zweiten Teil der Aufgabe ist es Ihren Algorithmus aus Aufgabe 3 in der Funktion *tranpose* zu implementieren.

Anders als bei den früheren HackerRank Aufgaben wird Ihnen hierfür das Framework in Moodle zur Verfügung gestellt. Sie können einfach dieses Framework in die HackerRank Aufgabe *Adjazenzfeldrepräsentation* kopieren und dann Ihr Programm dort wie gewohnt testen und abgeben.

Je nachdem wie viele Teile der Aufgabe Sie erledigt haben und mithilfe der Laufzeit Ihres Algorithmus und wird ein Score errechnet. Dieser liegt zwischen 0 und 10, je nachdem, wie schnell Ihr Algorithmus ist. 10 entspricht hierbei einer Laufzeit von 0 Sekunden und kann nicht erreicht werden. Wird beispielsweise nur $G.V$ und $G.E$ berechnet, liegt der maximal erreichbare Score bei <5.