Algorithmen und Datenstrukturen

Übungsgruppe 14

Utz Pöhlmann
4poehlma@informatik.uni-hamburg.de
6663579

Louis Kobras
4kobras@informatik.uni-hamburg.de
6658699

Rene Ogniwek
reneogniwek@gmx.net
6428103

Steffi Kaussow
s.kaussow@gmail.com
6414862

7. Dezember 2015

Punkte für den Hausaufgabenteil:

5.1	5.2	5.3	5.4	Σ

Zettel vom 26. 10. – Abgabe: 07. 12.

Übungsaufgabe 5.1

[| 4]

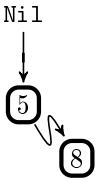
Aufgabe 5.1.1

Führen Sie die in der Vorlesung behandelte Einfügeoperation `TreeInsert` für binäre Suchbäume nacheinander mit den Elementen 5, 8, 9, 6, 7 aus. Zu Anfang sei der Baum leer. Geben Sie den nach jeder Einfügeoperation erhaltenen Suchbaum an. (1 Pkt.)

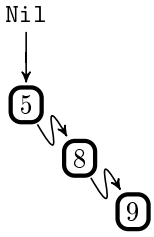
Im Folgenden wurden die `nil`-Blätter der Übersicht halber nicht mitgeführt.

Füge 5 ein:

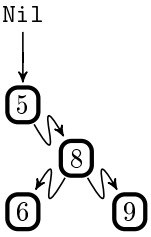
Füge 8 ein:



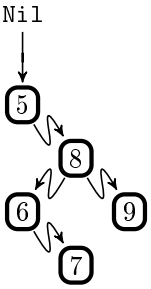
Füge 9 ein:



Füge 6 ein:

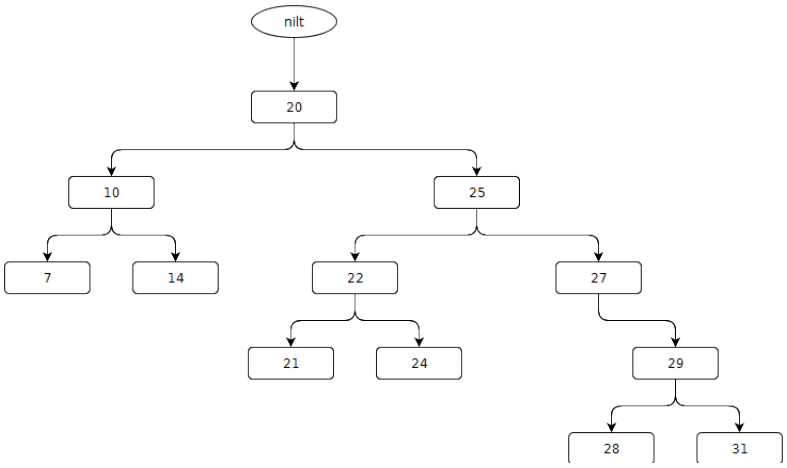


Füge 7 ein:



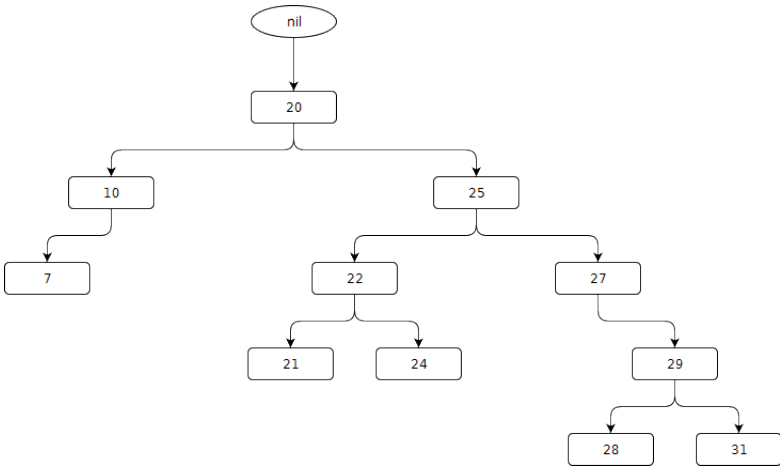
Aufgabe 5.1.2

Gegeben sei folgender binärer Suchbaum:

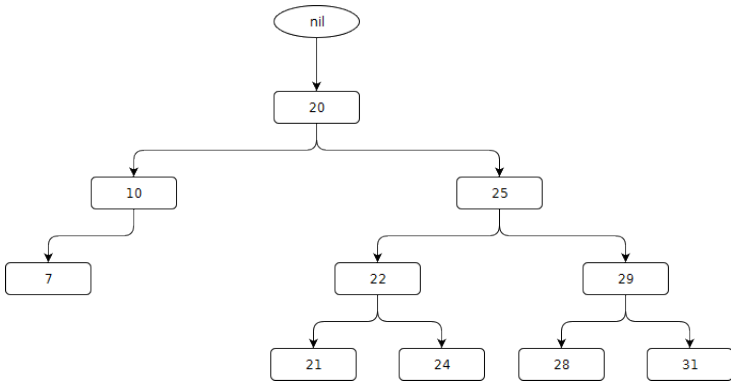


Führen Sie nacheinander die in der Vorlesung behandelte Löschoperation für binäre Suchbäume mit den Elementen 14, 27, 25 aus. Geben Sie den nach jeder Löschoperation erhaltenen Suchbaum an. (3 Pkt.)

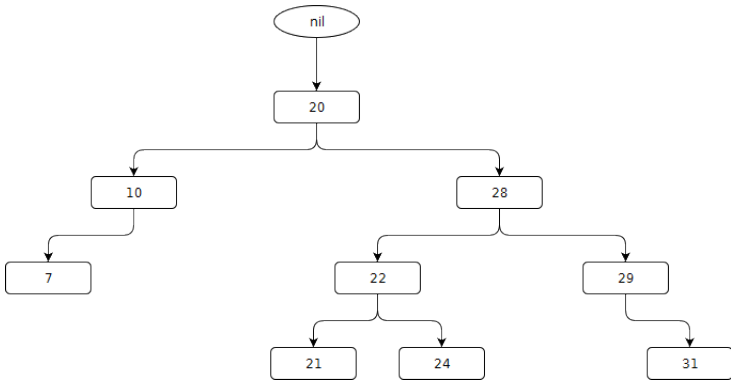
Nach Entfernen der 14:



Nach Entfernen der 27:



Nach Entfernen der 25:



Übungsaufgabe 5.2

[| 4]

Geben Sie für die Routine `InorderTreeWalk` einen nichtrekursiven Algorithmus (3 Pkt.) und für die Routine `TreeMinimum` einen rekursiven Algorithmus (1 Pkt.) an. Erläutern Sie in beiden Fällen unbedingt die Arbeitsweise ihres Algorithmus.

(Hinweis: Beide Routinen wurden in der Vorlesung behandelt. `InorderTreeWalk` *allerdings in rekursiver,* `TreeMinimum` *in nichtrekursiver Darstellung. Für die erste Teilaufgabe hilft ein Vergleich mit der Breiten-*

und Tiefensuche bei Graphen. Für die zweite Teilaufgabe betrachte man die Routine `TreeSearch` *in ihren beiden Varianten.)*

```
inorderTreeWalk_nichtRekursiv(x)
1  AusgabeArray = null
2  TreeMinimum(x)                    ; s. Vorlesung 7 Folie 37
3  AusgabeArray.add(x)
4  while TreeSucessor(x) ≠ nil       ; s. Vorlesung 7 Folie 47
5        x = TreeSucessor(x)
6        AusgabeArray.add(x)
7        return AusgabeArray
```

Dieser Algorithmus sucht sich das Minimum eines Baumes und packt es in ein Array. Danach wird sein Nachfolger gesucht und in das Array gepackt. Sollte ein Knoten irgendwann keinen Nachfolger mehr haben, terminiert der Algorithmus und wir geben das Array aus.

```
TreeMinimum_rekursiv(x)
1  if links[x] ≠ nil :
2  TreeMinimum_rekursiv(links[x])
3  else return x
```

Dieser Algorithmus prüft, ob der aktuelle Knoten ein linkes Kind hat. Ist dies der Fall, wird die Methode rekursiv am linken Kind aufgerufen (das linke Kind wird von seinem Vater gefragt, was denn sein Minimum sei). Sollte dies allerdings nicht so sein und der aktuelle Knoten besitzt kein linkes Kind, dann ist er selbst sein eigenes Minimum und damit auch das Minimum aller Knoten über ihm und wird ausgegeben.

Übungsaufgabe 5.3

[| 4]

Gegeben sei ein ungerichteter Graph $G = (V, E)$ auf n Knoten. Ferner seien $s, t \in V$ Knoten, deren Distanz (ihr kürzester Abstand) echt größer als $\frac{n}{2}$ sei, d.h. $d(s, t) > n/2$. Zeigen Sie zunächst, dass dann ein von s und t verschiedener Knoten v existieren muss, dessen Löschung alle s-t-Pfade zerstört. (Der Graph G', der aus G entsteht, wenn man v löscht, enthält also keine Pfade von s zu t mehr.)

Geben Sie dann einen Algorithmus an, der den Knoten v findet. Können Sie einen Algorithmus angeben, der in Zeit $O(n + m)$ arbeitet? Erläutern Sie die Arbeitsweise Ihres Algorithmus und begründen Sie seine Korrektheit sowie seine Laufzeit.

(Anmerkung: Die obige Fragestellung tritt bei der Untersuchung von Netzwerken auf. Sie besagt, dass, wenn zwei Knoten zu weit auseinander sind (nur durch viele Zwischenknoten verbunden sind), diese eine anfälligere Verbindung haben als Knoten die dichter beieinander sind. Obiges Resultat zeigt nämlich gerade, dass der Ausfall eines Knotens v genügt, um s und t zu trennen. Allerdings ist die Aussage nicht, dass irgendein Knoten v ausfallen kann und dann sofort s und t getrennt wären, sondern nur, dass (mindestens) ein solcher Knoten existiert.)

Sei G = (V, E) ein Graph. Dieser Graph besitzt die Knoten s, t und v und die Bedingung, dass d(s, t) > |V|/2 gilt. Damit enthält der kürzeste Pfad von s nach t mindestens die Hälfte aller Knoten (Genauer |V|/2 + 2 Knoten, wobei |V|/2 abgerundet wird und die + 2 durch s und t zustandekommen). Um nun einen weiteren Pfad dieser Länge zu kreieren, muss ein Teil des bisherigen Pfades wiederverwertet werden, da sonst nicht

mehr genügend freie Knoten zur Verfügung stünden. Und für jeden weiteren Pfad werden jeweils Teile der alten Pfade wieder verwertet. Irgendwann gehen die nicht benutzten Knoten aus und auch die Möglichkeiten übrige Knoten zu verbinden sind ausgeschöpft dann können weitere Pfade nicht generiert werden. Alle Knoten, die nun in der Schnittmenge aller Pfade liegen, sind potentielle v Kandidaten, da jeder mögliche Pfad sie enthält - sie sozusagen die Schnittstellen aller möglichen kürzesten Pfade sind und so in jedem Pfad vorkommen müssen, damit er ein Pfad von t nach s ist.

Wir setzen an jeden Knoten eine hochlaufende Zahl an, so dass jeder Knoten eine eigene individuelle Zahl erhält ($(O(n)$).
Wir führen eine Breitensuche aus, um den kleinsten Pfad zu bestimmen ($O(n+m)$).
Wir führen eine Tiefensuche aus, um irgendeinen anderen Pfad zu bestimmen ($O(n+m)$).
Da ein Pfad von s nach t existiert, werden beide Varianten dieser Suchen einen Pfad finden.
Wir drehen die die Reihenfolge der Adjazenzlisten um (Für jeden Knoten n müssen alle seine Nachbarn bearbeitet werden. Da eine Kante durch 2 "Nachbarpaareïn der Adjazenzlsite dargestellt wird, ergibt sich eine Laufzeit von $O(n+2*m) = O(n+m)$).
Wir führen eine zweite Tiefensuche durch, um einen Pfad maximal verschieden zum Zweiten zu erhalten ($O(n+m)$).

Nun werden die 3 Listen der Suchen jeweils nach der hochlaufenden Zahl sortiert ($O(Stelligkeit*(Listenlnge+Anzahlverschiedener Stellen))$).[1]

Die Stelligkeit dieser Zahlen ist maximal die Stelligkeit von n. Das Herausfinden dieser Stelligkeit geht mit konstantem Zeitaufwand.

Jede Stelle hat von Natur aus 10 mögliche Werte (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

Die Listenlänge ist im worst-case für jede Liste n.

$\Rightarrow$ Laufzeit: $3*c*(n+10)$. Das liegt in $O(n)$

Nun wird durch die 3 Listen gelaufen und jeweils verglichen, ob die Elemente in jeder dieser Liste drin sind. Da es sich bei diesen Elementen um sortierte Zahlen handelt, die sich innerhalb einer Liste nicht wiederholen, muss sich der Algorithmus jedes Element jeder Liste im worst-case genau einmal anschauen (Laufzeit: $3*n$. Dies liegt in $O(n)$).
Das erste gefundene Element/jedes gefundene Element (im Fall "jedes Element"muss der obige Vergleichsalgorithmus sich immer jedes Element jeder Liste anschauen, verändert aber nicht seine Laufzeit) wird in einen Knoten umgewandelt und ausgegeben (Laufzeit: $c+c$. Dies liegt in $O(c)$).

Gesamtlaufzeit: $O(n)+O(n+m)+O(n+m)+O(n+m)+O(n+m)+O(n)+O(n)+O(c) = O(n+m)$
Idee vollständig?: Ja, da durch die 2 Tiefensuchen mit Umkehrung der Adjazenzliste 2 maximal voneinander unabhängige Pfade gefunden werden. Die Schnittmenge von diesen wird in beiden Pfaden benötigt, um von s nach t zu kommen. Durch die Breitensuche wird dann auch noch garantiert, dass es sich um einen kürzesten Pfad handelt und somit um die minimale Anzahl der Knoten. Dies führt dazu, dass die Schnittmenge von diesem Pfad und den obigen beiden minimal ist. Und in dieser minimalen Schnittmenge ist nun jeder Knoten ein v-Kandidat.

Pseudocode:

```
ein_v_knoten(Graph G, Knoten s, Knoten t)
    //gibt einen möglichen v-Knoten aus
1   List Vs = v_knoten(G,s,t)
2   index = random(Vs.length)
3   return Vs[index]
```

[1] Quelle: s. Vorlesung 5 Folie 46 http://www.informatik.uni-hamburg.de/TGI/lehre/vl/WS1516/AuD/Folien/aud_v5.pdf

```
v_knoten(Graph G, Knoten s, Knoten t)
    //gibt eine Liste mit allen v-Knoten aus
 1  inti = 0
 2  for KnotenninG
 3       n.zahlenwert = i
 4       i = i + 1
 5  List A = sucheKleinstenPfad(G, s, t)
 6  List B = sucheAnderenPfad(G, s, t)
 7  GraphG' = G
 8  G'.Adjazenzliste = reverse(G.Adjazenzliste)
 9  List C = sucheAnderenPfad(G', s, t)
10  List A' = RadixSort(A nach Knoten.zahlenwert)
    // s. Vorlesung 5 Folie 46; Link: http://www.informatik.uni-hamburg.de/TGI/lehre/vl/
    // WS1516/AuD/Folien/aud_v5.pdf
11  List B' = RadixSort(B nach Knoten.zahlenwert)
12  List C' = RadixSort(C nach Knoten.zahlenwert)
13  List ErgebnisListe = new List
14  aktuellerKnoten = new Knoten
15  indexA = 0
16  indexB = 0
17  indexC = 0
18  while A'.length > indexA and B'.length > indexB and C'.length > indexC :
    //echt größer, da der Index ab 0 beginnt
19       aktuellerKnoten = A'[indexA]
20       if aktuellerKnoten < B'[indexB]
21            indexA = indexA + 1
22       else if aktuellerKnoten > B'[indexB]
23            indexB = indexB + 1
24       else
              //falls aktuellerKnoten == B'[indexB] ist
25            if aktuellerKnoten < C'[indexC]
26                 indexA = indexA + 1
27            else if aktuellerKnoten > C'[indexC]
28                 indexC = indexC + 1
29            else
                   //falls aktuellerKnoten == C'[indexC] ist
30                 ErgebnisListe.add(aktuellerKnoten)
31                 indexA = indexA + 1
32  return ErgebnisListe

sucheAnderenPfad(Graph G, Knoten s, Knoten t)
    //gibt einen Pfad von s nach t aus aus
1   Tiefensuche(G, s)
2   return bauePfad(G, s, t)
```

```
bauePfad(Graph G, Knoten s, Knoten t)
    //baut gegeben einen durchsuchten Graphen einen Pfad von s nach t
    //(s und t sind nicht in dem Pfad enthalten um später nicht in der Schnittmenge zu landen)
1   Knoten n = t.parent
2   List A = newList
3   while !(n == s)
4   A.add(n)
5   n = n.parent
6
7   Return A
```

```
sucheKleinstenPfad(Graph G, Knoten s, Knoten t)
    // gibt einen kürzesten Pfad von s nach t aus
1   Breitensuche(G, s)
2   return bauePfad(G, s, t)
```

```
Tiefensuche(Graph G, Knoten wurzel)
    // führt eine Tiefensuche auf dem Graphen durch
 1  for Knoten n in G
 2      n.entdeckt = false
 3      n.parent = NIL
 4  StackS = newStack
 5  S.push(wurzel)
 6  Knoten aktuellerKnoten = NULL
 7  while !(Sisempty)
 8      aktuellerKnoten = S.pop()
 9      if aktuellerKnoten.entdeckt == false
10          aktuellerKnoten.entdeckt = true
11      for Knoten m in aktuellerKnoten.AdjazenzListe
12          S.push(m)
13      m.parent = aktuellerKnoten
```

```
Breitensuche(Graph G, Knoten wurzel)
    // führt eine Breitensuche auf dem Graphen durch
 1  for Knoten n in G
 2      n.distanz = INFINITY
 3      n.parent = NIL
 4  Queue Q = new Queue
 5  wurzel.distanz = 0
 6  q.enqueque(wurzel)
 7  Knoten aktuellerKnoten = NULL
 8  while !(Qisempty)
 9      aktuellerKnoten = Q.dequeue()
10      for Knoten m in aktuellerKnoten.Adjazenzliste
11          if m.distanz == INFINITY
12              m.distanz = aktuellerKnoten.distanz + 1
13              m.parent = aktuellerKnoten
14              q.enqueue(m)
```

Übungsaufgabe 5.4

[| 4]

Aufgabe 5.4.1

Im Problem `Big-Clique` ist ein ungerichteter Graph G gegeben. Die Frage ist, ob G eine Clique enthält, die aus mindestens $\frac{n}{2}$ Knoten besteht (wobei n die Anzahl der Knoten von G ist). Zeigen Sie, dass `Big-Clique` NP-vollständig ist. (2 Pkt.)

Big-Clique liegt in NP.
Verifikationsalgorithmus, der die Richtigkeit eines gegebenen Zertifikates in polynomialzeit überprüft:
Unser Zertifikat: Eine Menge an Knoten - unsere Clique - und der Graph
Wir schauen in der Adjazenzliste nach, ob zu jedem gegebenen Knoten, zu allen anderen gegebenen Knoten eine Kante existiert. Ist dies der Fall, ist das Zertifikat richtig, sonst nicht.
Big-Clique liegt in NPH.
Reduktion eines Algorithmusses aus NPH auf Big-Clique:
Wir nehmen Clique(G, k): G = (V, K) Graph, k = Größe der Clique
Wenn unser k größer oder gleich |V|/2 ist, geben wir das Problem einfach in Big-Clique.
Ist es allerdings kleiner, fügen wir für jeden Knoten, der uns auf |V|/2 fehlt, 2 Knoten an den Graphen an, die wir jeweils mit Kanten zu allen anderen Knoten versehen.
Diesen neuen Graphen fügen wir in den Automaten ein, der Big-Clique löst und geben die Lösung als unsere Lösung für unser Clique-Problem aus.
Da wir beides obere haben, folgt daraus: Big-Clique liegt in NPC

Aufgabe 5.4.2

Geben Sie unter der Annahme $P = NP$ einen deterministischen Algorithmus an, der in polynomieller Zeit zu einem ungerichteten Graphen G eine maximal große Clique bestimmt. (2 Pkt.)

Verifikationsalgorithmus, der die Richtigkeit eines gegebenen Zertifikates in polynomialzeit überprüft:
Unser Zertifikat: Eine Menge an Knoten - unsere größte mögliche Clique - und der Graph
Wir geben unseren Graph und die Zahl k = |Menge an Knoten| + 1 in Clique.
Wenn Clique wahr ausgibt, ist das Zertifikat falsch, da eine größere Clique existiert.
Dies läuft in polynomialzeit, da P = NP angenommen wird.
Durch diesen Verifikationsalgorithmus ist unser Problem in NP.
Unter der Annahme P = NP existiert allerdings auch ein P-Algorithmus und dieser ist deterministisch.