

Beispielblatt 2

186.813 VU Algorithmen und Datenstrukturen 1 VU 6.0

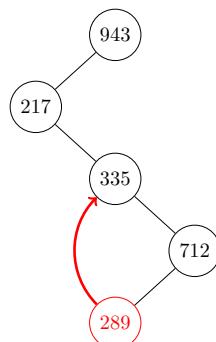
25. September 2013

Aufgabe 1 Gegeben sei ein binärer Suchbaum mit Werten im Bereich von 1 bis 1001. In diesem Baum wird nach der Zahl 363 gesucht. Überprüfen Sie für die angegebenen Folgen (a) bis (d), ob die Elemente in der angegebenen Reihenfolge Werte von Knoten repräsentieren können, die bei der Suche im binären Suchbaum traversiert wurden. Unterstützen Sie Ihre Argumentation jeweils mit einer Skizze:

- (a) $\langle 943, 217, 335, 712, 289, 397, 348, 363 \rangle$
 - (b) $\langle 1001, 199, 935, 240, 936, 273, 363 \rangle$
 - (c) $\langle 1000, 200, 898, 248, 889, 273, 361, 363 \rangle$
 - (d) $\langle 2, 398, 388, 211, 276, 385, 379, 278, 363 \rangle$
-

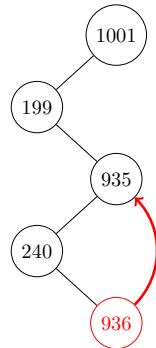
Lösung

- (a) Von den ersten Zahlen 943 – 712 der Zahlenfolge ist es durchaus möglich, dass die angegebene Folge Werte von Knoten in einem binären Suchbaum repräsentiert. Aber der Knoten mit dem Wert 289 ist falsch eingeordnet, denn dieser müsste sich eigentlich in dem linken Teilbaum von Knoten 335 befinden. \Rightarrow Für die gesamte Zahlenfolge $\langle 943, 217, 335, 712, 289, 397, 348, 363 \rangle$ ist es nicht möglich. (Der Wertebereich, den Knoten im linken Teilbaum von 712 annehmen können ist zwischen 335 – 712)

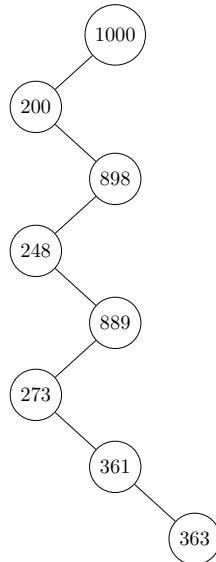


- (b) Von den ersten Zahlen 1001 – 240 der Zahlenfolge ist es durchaus möglich, dass die angegebene Folge Werte von Knoten in einem binären Suchbaum repräsentiert. Aber

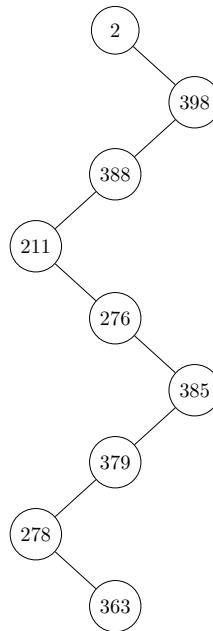
der Knoten mit dem Wert 936 ist falsch eingeordnet, denn dieser müsste sich eigentlich in dem rechten Teilbaum von Knoten 935 befinden. \Rightarrow Für die gesamte Zahlenfolge $\langle 1001, 199, 935, 240, 936, 273, 363 \rangle$ ist es nicht möglich. (Der Wertebereich, den Knoten im rechten Teilbaum von 240 annehmen können ist zwischen 240 – 935)



- (c) Ja, diese Traversierungsreihenfolge $\langle 1000, 200, 898, 248, 889, 273, 361, 363 \rangle$ ist durchaus möglich, wenn es sich um einen binären Suchbaum handelt.



- (d) Ja, diese Traversierungsreihenfolge $\langle 2, 398, 388, 211, 276, 385, 379, 278, 363 \rangle$ ist durchaus möglich, wenn es sich um einen binären Suchbaum handelt.



Listing 1: Check ob es sich um gültige Suchbäume handelt. (Für Klasse `bin_tree.h` siehe Listing 3)

```

1 //This Project is written in C++11, so use following command to compile it:
2 // g++ -ggdb -std=c++11 -o bsp1 bsp1.cpp
3
4 #include <iostream>
5 #include "bin_tree.h"
6
7 int main(){
8     std::cout << std::endl << "Bsp 1.a" << std::endl;
9     { //1.a
10         {
11             bin_tree<int> t;
12             static const int inserts[] = {943,217,335,712};
13             std::vector<int> ins (inserts, inserts + sizeof(inserts) / sizeof(int));
14             for(int i:ins) {
15                 t.insert(i);
16             }
17             t.insert_manually(289,712,true); //insert the node 289 as leftchild
18             if(t.check_binary_search_tree()) {
19                 std::cout << "RICHTIG" << std::endl;
20             } else {
21                 std::cout << "FALSCH" << std::endl;
22             }
23
24             std::cout << std::endl << t.to_string(3) << std::endl;
25         }
26         {
27             bin_tree<int> t;
28             static const int inserts[] = {943,217,335,712,289};

```

```

29         std::vector<int> ins (inserts, inserts + sizeof(inserts) / sizeof(int));
30         for(int i:ins) {
31             t.insert(i);
32         }
33         std::cout << std::endl <<t.to_string(3) << std::endl;
34     }
35 }
36 std::cout << std::endl << "Bsp 1.b" << std::endl;
37 { //1.b
38 {
39     bin_tree<int> t;
40     static const int inserts[] = {1001, 199, 935, 240};
41     std::vector<int> ins (inserts, inserts + sizeof(inserts) / sizeof(int));
42     for(int i:ins) {
43         t.insert(i);
44     }
45     t.insert_manually(936,240,false); //insert the node 936 as right child of 240
46     if(t.check_binary_search_tree()) {
47         std::cout << "RICHTIG" << std::endl;
48     } else {
49         std::cout << "FALSCH" << std::endl;
50     }
51
52     std::cout << std::endl <<t.to_string(3) << std::endl;
53 }
54 {
55     bin_tree<int> t;
56     static const int inserts[] = {1001, 199, 935, 240, 936};
57     std::vector<int> ins (inserts, inserts + sizeof(inserts) / sizeof(int));
58     for(int i:ins) {
59         t.insert(i);
60     }
61     std::cout << std::endl <<t.to_string(3) << std::endl;
62 }
63 }
64 std::cout << std::endl << "Bsp 1.c" << std::endl;
65 { //1.c
66 {
67     bin_tree<int> t;
68     static const int inserts[] = {1000, 200, 898, 248, 889, 273, 361};
69     std::vector<int> ins (inserts, inserts + sizeof(inserts) / sizeof(int));
70     for(int i:ins) {
71         t.insert(i);
72     }
73     if(t.check_binary_search_tree()) {
74         std::cout << "RICHTIG" << std::endl;
75     } else {
76         std::cout << "FALSCH" << std::endl;

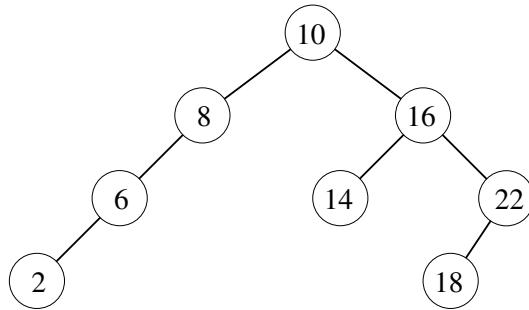
```

```

77     }
78
79     std::cout << std::endl <<t.to_string(3) << std::endl;
80 }
81 }
82 std::cout << std::endl << "Bsp 1.d" << std::endl;
83 { // 1.d
84 {
85     bin_tree<int> t;
86     static const int inserts[] = {2, 398, 388, 211, 276, 385, 379, 2
87     std::vector<int> ins (inserts, inserts + sizeof(inserts) / sizeof(
88     for(int i:ins) {
89         t.insert(i);
90     }
91     if(t.check_binary_search_tree()) {
92         std::cout << "RICHTIG" << std::endl;
93     } else {
94         std::cout << "FALSCH" << std::endl;
95     }
96
97     std::cout << std::endl <<t.to_string(3) << std::endl;
98 }
99 }
100 return 0;
101 }

```

Aufgabe 2 Gegeben sei folgender AVL-Baum direkt nach dem Einfügen eines neuen Elements:

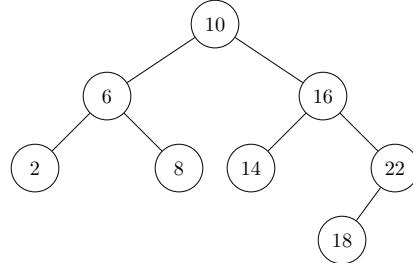


Ist die AVL-Eigenschaft in diesem Baum verletzt? Falls ja, nennen Sie den Knoten der zuletzt eingefügt wurde, führen Sie die notwendigen Maßnahmen durch, um die AVL-Bedingung wieder herzustellen, und zeichnen Sie den Baum in seinem korrigierten Zustand.

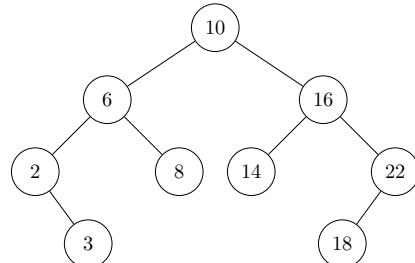
Führen Sie danach folgende Operationen in diesem AVL-Baum in der angegebenen Reihenfolge durch und zeichnen Sie den Baum nach jeder Operation. Achten Sie darauf, dass die AVL-Eigenschaft immer erhalten bleibt.

- Fügen Sie 3 in den AVL-Baum ein.
 - Fügen Sie 5 in den AVL-Baum ein.
 - Löschen Sie 6 aus dem AVL-Baum (verwenden Sie falls notwendig den Successor als Ersatzelement).
-

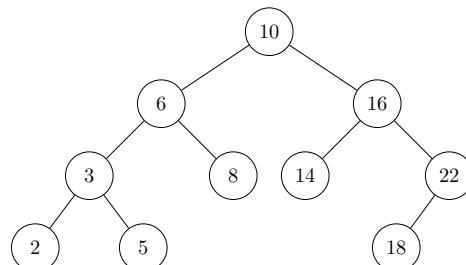
Lösung Ja, die AVL-Eigenschaft in diesem Baum ist verletzt. Der Knoten 2 wurde als letztes eingefügt. Dadurch erfüllt der Teilbaum mit der Wurzel 8 nicht mehr die AVL-Eigenschaft. \Rightarrow Der Teilbaum mit der Wurzel 8 muss nach rechts rotiert werden. Dadurch erhält der Teilbaum das Element 6 als neue Wurzel.



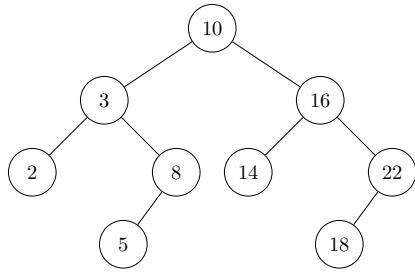
- **3 in den AVL-Baum einfügen:** Hier wird die 3 einfach als rechtes Kind von 2 eingefügt. Es sind keine weiteren Schritte notwendig.



- **5 in den AVL-Baum einfügen:** Hier wird die 5 als rechtes Kind von 3 eingefügt und dadurch wird die AVL-Bedingung verletzt. \Rightarrow Der Teilbaum mit der Wurzel 2 wird nach links rotiert, wodurch die 3 die neue Wurzel dieses Teilbaums wird.



- **6 aus dem AVL-Baum löschen:** Hier wird 6 aus dem Baum rausgelöscht und durch 8 ersetzt. Dadurch entsteht wieder ein ungültiger AVL-Baum, denn der Teilbaum mit der Wurzel 8 erfüllt nicht die AVL-Eigenschaft. \Rightarrow Der Teilbaum muss nach rechts rotiert werden. Dadurch wird die 3 zur neuen Wurzel und die 5 wird links an die 8 angehängt.



Listing 2: Check ob es sich um gültige Suchbäume handelt. (Für Klasse `avl_tree.h` siehe Listing 4)

```

1 //This Project is written in C++11, so use following command to compile it:
2 // g++ -ggdb -std=c++11 -o bsp2 bsp2.cpp
3
4 #include <iostream>
5 #include "avl_tree.h"
6
7 int main(){
8     avl_tree<int> t;
9     static const int inserts[] = {10, 8, 16, 6, 14, 22, 18};
10    std::vector<int> ins (inserts, inserts + sizeof(inserts) / sizeof(inserts));
11    for(int i:ins) {
12        t.insert(i);
13    }
14
15    std::cout << std::endl <<t.to_string(2) << std::endl;
16
17    t.insert(2);
18
19    std::cout << std::endl <<t.to_string(2) << std::endl;
20
21    t.insert(3);
22
23    std::cout << std::endl <<t.to_string(2) << std::endl;
24
25    t.insert(5);
26
27    std::cout << std::endl <<t.to_string(2) << std::endl;
28
29    return 0;
30}

```

Aufgabe 3 Vergleichen Sie die Datenstrukturen doppelt verkettete, zyklische Liste (aufsteigend sortiert), doppelt verkettete, zyklische Liste (nicht sortiert), natürlicher binärer Suchbaum und AVL-Baum bezüglich ihres Zeitaufwandes für die Suche nach einem Schlüssel k und die

Bestimmung des größten Elementes im Best- und Worst-Case in Θ -Notation in Abhängigkeit der Anzahl n der gespeicherten Elemente.

Lösung

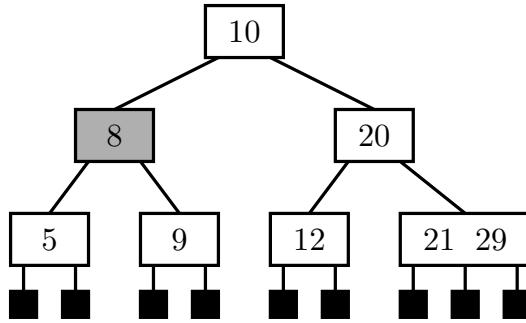
i) Suche nach dem Schlüssel k

- aufsteigend sortierte doppelt verkettete Zyklische Liste
Best-Case: $\Theta(1)$
Worst-Case: $\Theta(n/2) = \Theta(n)$
- unsortierte doppelt verkettete Zyklische Liste
Best-Case: $\Theta(1)$
Worst-Case: $\Theta(n)$
- natürlicher binärer Suchbaum
Best-Case: $\Theta(1)$
Worst-Case: $\Theta(n)$
- AVL-Baum
Best-Case: $\Theta(1)$
Worst-Case: $\Theta(\log(n))$

ii) Bestimmung des größten Elements

- aufsteigend sortierte doppelt verkettete Zyklische Liste
Best-Case: $\Theta(1)$
Worst-Case: $\Theta(1)$
- unsortierte doppelt verkettete Zyklische Liste
Best-Case: $\Theta(1)$
Worst-Case: $\Theta(n)$
- natürlicher binärer Suchbaum
Best-Case: $\Theta(1)$
Worst-Case: $\Theta(n)$
- AVL-Baum
Best-Case: $\Theta(1)$
Worst-Case: $\Theta(\log(n))$

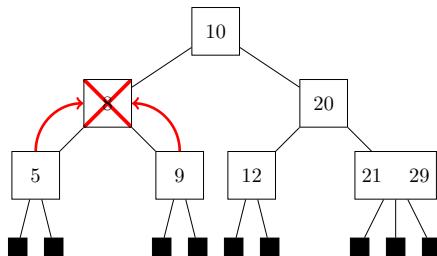
Aufgabe 4 Aus folgendem B-Baum der Ordnung 3 soll der Schlüssel 8 entfernt werden:



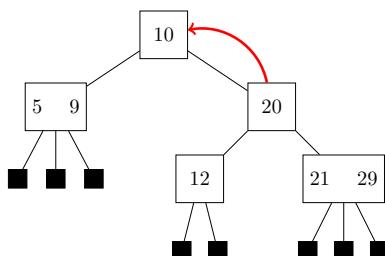
Zeichnen Sie den Baum nach jedem Schritt der Reorganisation, die notwendig ist, um wieder einen gültigen B-Baum zu erhalten (die leeren, schwarz dargestellten Blätter können bei der Zeichnung entfallen).

Lösung

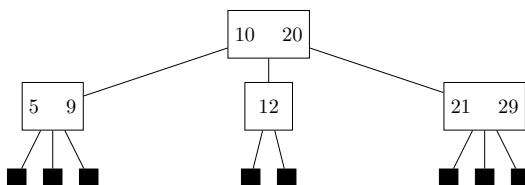
- i) Löschen des Elements mit dem Schlüssel 8 und erstellung eines neuen Knotens mit den beiden Schlüsseln 5 und 9:



- ii) Das Element mit dem Schlüssel 20 nach oben verschieben:



- iii) Die Schlüssel 10 und 20 sind nun die neue Wurzel:



Aufgabe 5 Geben Sie den Zustand einer Hashtabelle der Länge 13 an, wenn die Schlüssel

$$\langle 5, 1, 16, 22, 14, 29, 32, 27, 2 \rangle$$

in gegebener Reihenfolge in die anfangs leere Datenstruktur eingefügt werden und ein offenes Hashverfahren mit der Hashfunktion $h(k) = k \bmod 13$ sowie

- (a) lineares Sondieren (Schrittweite 2)
- (b) Double Hashing ($h'(k) = 1 + (k \bmod 5)$)

verwendet wird.

Vergleichen Sie die Anzahl der beim Einfügen betrachteten Hashtabellenplätze für die angegebenen Sondierungsfunktionen.

Lösung Schlüssel und Wert der Hashfunktion:

k	5	1	16	22	14	29	32	27	2
$h_1(k)$	5	1	3	9	1	3	6	1	2
$h_2(k)$	1	2	2	3	5	5	3	3	3

- (a) lineares Sondieren (Schrittweite 2)

i) Einfügen von 5 (da die Tabelle leer ist, ist dies ohne Probleme möglich) **1 Vergleich**:

0	1	2	3	4	5	6	7	8	9	10	11	12
					5							

ii) Einfügen von 1 (da die Tabelle fast leer ist, ist dies ohne Probleme möglich) **1 Vergleich**:

0	1	2	3	4	5	6	7	8	9	10	11	12
	1				5							

iii) Einfügen von 16 (da die Tabelle fast leer ist, ist dies ohne Probleme möglich) **1 Vergleich**:

0	1	2	3	4	5	6	7	8	9	10	11	12
		1		16		5						

iv) Einfügen von 22 (da die Tabelle fast leer ist, ist dies ohne Probleme möglich) **1 Vergleich**:

0	1	2	3	4	5	6	7	8	9	10	11	12
		1		16		5			22			

v) Einfügen von 14 (hier gibts Kollisionen mit 1, 3 und 5, aber 7 ist frei) **4 Vergleiche**:

0	1	2	3	4	5	6	7	8	9	10	11	12
	1		16		5		14		22			

vi) Einfügen von 29 (hier gibts Kollisionen mit 3, 5, 7 und 9, aber 11 ist frei) **5 Vergleiche**:

0	1	2	3	4	5	6	7	8	9	10	11	12
	1		16		5		14		22		29	

- vii) Einfügen von 32 (hier gibts keine Kollisionen, somit ist das Einfügen auf 6 ohne Probleme möglich) **1 Vergleich:**

0	1	2	3	4	5	6	7	8	9	10	11	12
	1		16		5	32	14		22		29	

- viii) Einfügen von 27 (hier gibts Kollisionen mit 1, 3, 5, 7, 9 und 11, aber 0 ist frei) **7 Vergleiche:**

0	1	2	3	4	5	6	7	8	9	10	11	12
27	1		16		5	32	14		22		29	

- ix) Einfügen von 2 (hier gibts keine Kollisionen, somit ist das Einfügen auf 2 ohne Probleme möglich) **1 Vergleich:**

0	1	2	3	4	5	6	7	8	9	10	11	12
27	1	2	16		5	32	14		22		29	

⇒ **22 Vergleiche sind bei diesem offenen Hashverfahren notwendig.**

- (b) Double Hashing ($h'(k) = 1 + (k \bmod 5)$)

- i) Einfügen von 5 (da die Tabelle leer ist, ist dies ohne Probleme möglich) **1 Vergleich:**

0	1	2	3	4	5	6	7	8	9	10	11	12
					5							

- ii) Einfügen von 1 (da die Tabelle fast leer ist, ist dies ohne Probleme möglich) **1 Vergleich:**

0	1	2	3	4	5	6	7	8	9	10	11	12
	1				5							

- iii) Einfügen von 16 (da die Tabelle fast leer ist, ist dies ohne Probleme möglich) **1 Vergleich:**

0	1	2	3	4	5	6	7	8	9	10	11	12
		1		16		5						

- iv) Einfügen von 22 (da die Tabelle fast leer ist, ist dies ohne Probleme möglich) **1 Vergleich:**

0	1	2	3	4	5	6	7	8	9	10	11	12
		1		16		5			22			

- v) Einfügen von 14 (hier gibts eine Kollision mit 1, aber 6 ist frei) **2 Vergleiche:**

0	1	2	3	4	5	6	7	8	9	10	11	12
	1		16		5	14			22			

- vi) Einfügen von 29 (hier gibts eine Kollision mit 3, aber 8 ist frei) **2 Vergleiche:**

0	1	2	3	4	5	6	7	8	9	10	11	12
		1		16		5	14		29	22		

- vii) Einfügen von 32 (hier gibts Kollisionen mit 6 und 9, aber 12 ist frei) **3 Vergleich:**

0	1	2	3	4	5	6	7	8	9	10	11	12
	1		16		5	14		29	22			32

viii) Einfügen von 27 (hier gibts eine Kollision mit 1, aber 4 ist frei) **2 Vergleiche**:

0	1	2	3	4	5	6	7	8	9	10	11	12
	1		16	27	5	14		29	22			32

ix) Einfügen von 2 (hier gibts keine Kollisionen, somit ist das Einfügen auf 2 ohne Probleme möglich) **1 Vergleich**:

0	1	2	3	4	5	6	7	8	9	10	11	12
	1	2	16	27	5	14		29	22			32

⇒ **14 Vergleiche sind bei diesem offenen Hashverfahren notwendig.**

⇒ Die Sondierungsmethode Double-Hashing ist wesentlich Effizienter in diesem Fall (wie sonst meist auch), denn die 2. von der 1. unabhängigen Hashfunktion bewirkt eine viel bessere Streuung der Werte in der Tabelle.

Aufgabe 6 Breitensuche ist ein Verfahren zum Durchsuchen bzw. Durchlaufen von Knoten eines Graphen ähnlich der in der Vorlesung behandelten Tiefensuche. Auch hier geht man von einem Startknoten u aus, allerdings unterscheiden sich Tiefen- und Breitensuche hinsichtlich der Reihenfolge, in der weitere Knoten des Graphen abgearbeitet bzw. besucht werden. Wir gehen im Folgenden von einem ungerichteten Graphen aus.

Beginnend mit dem Startknoten u werden bei der Breitensuche zunächst alle zu u adjazenten Knoten besucht, d.h. alle Knoten v , für die eine Kante (u, v) im Graphen existiert; zusätzlich werden alle diese Knoten v in einer Warteschlange gespeichert. Die Breitensuche bearbeitet also zuerst immer alle direkt benachbarten Knoten und folgt nicht – wie die Tiefensuche – gleich einem Pfad in die Tiefe.

Nachdem nun alle adjazenten Knoten von u betrachtet wurden, wird der erste Knoten der Warteschlange entnommen und für diesen das Verfahren wiederholt. Dies wird nun so lange fortgesetzt, bis entweder die Warteschlange leer ist oder bis – wenn man nach einem bestimmten Knoten sucht – dieser gefunden wurde. Wie auch bei der Tiefensuche werden durch Markieren bereits bearbeiteter Knoten Mehrfachbesuche verhindert.

Gegeben sei nun die Datenstruktur Queue (Warteschlange), welche eine beliebige Menge an Objekten aufnehmen kann und diese gemäß der Reihenfolge ihres Einfügens zurück liefert. Folgende Operationen werden von einer Queue Q zur Verfügung gestellt:

- $Q.isEmpty()$: Liefert true zurück, falls die Queue Q keine Elemente enthält, und false sonst.
- $Q.put(x)$: Fügt das Element x der Queue Q hinzu.
- $Q.get()$: Entfernt das älteste Element in der Queue Q und liefert dieses zurück.

Benutzen Sie die Queue, um eine nicht-rekursive Version der Breitensuche zu entwerfen. Beschreiben Sie erst in wenigen Worten den Ablauf Ihres Algorithmus und geben Sie diesen dann

in Pseudocode an. Die Queue können Sie dabei als „Black Box“ betrachten, d.h., Sie können sie benutzen, ohne die genaue Funktionsweise explizit als Pseudocode ausarbeiten zu müssen.

Lösung

1. Alle eventuell markierten Knoten entmarkieren und die Queue leeren.
2. Anschließend wird der Startknoten in die Queue gelegt und markiert.
3. Das Erste Element wird aus der Queue genommen und für diesen Knoten werden alle Nachbarknoten bestimmt, die dann sofern sie nicht bereits markiert wurden in die Queue gelegt und markiert werden.
4. Es wird solange der Schritt 3 wiederholt, bis die Queue leer ist.

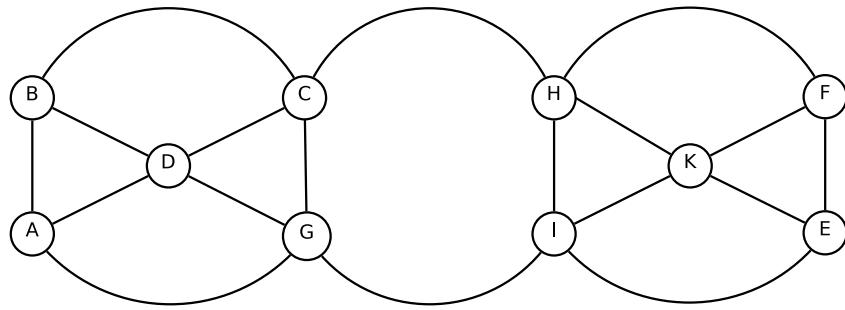
Algorithmus Breitensuche(G, s):

Eingabe: Graph $G = (V, E)$; Knoten $s \in V$

Ausgabe: alle Knoten G , die von s aus erreichbar sind

```
1: Initialisierung: setze markiert[ $v$ ] = 0 für alle  $v$ ;  
   // Löschen der Elemente in der Queue  
2: solange  $Q.isEmpty() == \text{Falsch}$  {  
3:    $Q.get()$ ;  
4: }  
5:  $Q.put(s)$ ;  
6: markiert[ $s$ ] = 1;  
7: solange  $Q.isEmpty() == \text{Falsch}$  {  
8:    $v = Q.get()$ ;  
9:   Ausgabe von  $v$ ;  
10:  für alle Knoten  $w$  aus  $N(v)$  {  
11:    falls markiert[ $w$ ] == 0 dann {  
12:       $Q.put(w)$ ;  
13:      markiert[ $w$ ] = 1;  
14:    }  
15:  }  
16: }
```

Aufgabe 7 Auf dem gegebenen Graphen wird die aus dem Skriptum bekannte **Tiefensuche** und die aus dem vorigen Übungsbeispiel bekannte **Breitensuche** durchgeführt.



Welche der folgenden Listen von besuchten Knoten können dabei in genau dieser Reihenfolge entstehen? Hinweis: Die Nachbarn eines Knotens können in beliebiger Reihenfolge abgearbeitet werden.

Reihenfolge	Tiefensuche	Breitensuche
A B C D E F G H I K		
A B D C G H I K F E		
B A D C G H I K F E		x
A B C D H K F E I H		
K E F H I G C D A B	x	x
C G I K E F H B D A	x	

Aufgabe 8 Gegeben ist ein gerichteter Graph $G = (V, E)$, ein Startknoten $s \in V$ und ein Zielknoten $t \in V$. Die Distanzen zwischen den Knoten sind in der Adjazenzmatrix A gegeben.

Entwerfen Sie einen **Greedy-Algorithmus** in Pseudocode, der (mit der Motivation einen möglichst kurzen Weg vom Startknoten s zum Zielknoten t zu finden) wie folgt vorgeht: Ausgehend vom Startknoten s wählt dieser Algorithmus bei jedem Knoten jeweils immer die kürzeste noch nicht verwendete Kante, um dem Zielknoten t auf einem zusammenhängenden Pfad näher zu kommen.

Liefert dieser Algorithmus immer den kürzesten Pfad? Begründen Sie Ihre Antwort.

Veranschaulichen Sie die Funktionsweise des Algorithmus anhand eines Beispiels (Graph mit 4 – 5 Knoten), bzw. geben Sie ein möglichst einfaches Gegenbeispiel an.

Lösung Der Folgende Algorithmus verwendet eine einfach verkettete Liste L , die alle Kanten enthält, die der Algorithms auf dem Weg von Knoten s nach e gefunden hat. In die Liste kann mittels $L.push_back(e)$ ein Element eingefügt werden. Die Liste kann mit $L.clear()$ gelöscht werden.

Algorithmus Breitensuche(G, s):

Eingabe: Graph $G = (V, E)$ mit Gewichten $w_e \forall e \in E$; Knoten $s, e \in V$

Ausgabe: möglichst kurzer Pfad von s nach e (ist aber nicht unbedingt der kürzeste)

```

1: Initialisierung: setze markiert[i] = 0 für alle i;  

   // Löschen der Elemente in der Liste  

2: L.clear();  

3: k = s;  

4: solange k ≠ e {  

5:   sdist=NULL;  

6:   nextk=NULL;  

7:   für alle Knoten l aus N(k) {  

8:     edge = e(k,l); // edge enthält die Kante zwischen den Knoten k und l  

9:     falls markiert[edge] == 0 UND (sdist == NULL ODER w(edge) < sdist) dann {  

10:      sdist=w(edge); // sdist enthält die Distanz zwischen den Knoten k und l  

11:      nextk=l;  

12:    }  

13:  }  

14:  falls sdist == NULL dann {  

15:    Fehler, es existiert kein Weg von s nach e.  

16:  }  

17:  k = nextk;  

18:  markiert[edge] = 1;  

19:  L.push_back(edge);  

20: }

```

Der in Abbildung 1 gezeigte Graph ist z.B. ein Paradebeispiel dafür, dass der Algorithmus nicht immer den kürzesten Pfad vom Knoten *s* nach *e* findet. (Denn der optimalste Weg wäre der direkte zwischen *s* und *e*.)

Das Problem an diesem Algorithmus ist, dass er sich immer nur die lokalen Bestwerte vergleicht und keinen Blick fürs ganze hat.

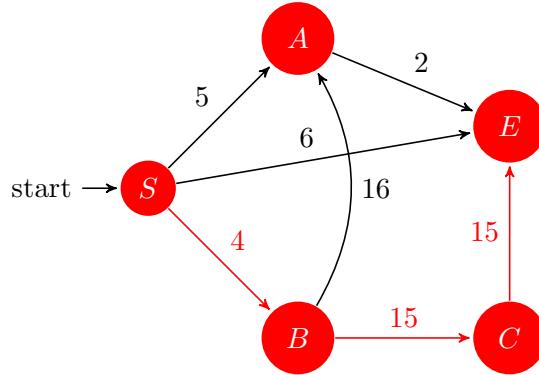


Abbildung 1: Beispiel für einen Graph, der sich nicht gut für den Greedy-Algorithmus eignet.

A. Bin-Tree

A.1. bin_tree.h

Listing 3: Klasse bin_tree (Für Klasse avl_tree.h siehe Listing 4)

```
1 #ifndef BIN_TREE_H
2 #define BIN_TREE_H
3
4 #include <memory>
5 #include <stdexcept>
6 #include <string>
7 #include <sstream>
8 #include <vector>
9 #include <iomanip>
10
11 #include <math.h>
12
13 template <typename T>
14 class bin_tree{
15     protected:
16         template <typename U>
17         class element{
18             private:
19                 typedef U data_type;
20             public:
21                 element() = delete;
22                 element(data_type _data);
23                 element(data_type _data, element *_parent);
24                 element(data_type _data, element *_parent, element *_left, e
25                     data_type data;
26                     int height;
27                     element *left;
28                     element *right;
29                     element *parent;
30                     ~element();
31             };
32             template <typename U>
33             class limit{
34                 private:
35                     typedef U data_type;
36                     data_type lim;
37                     bool undef;
38                 public:
39                     limit();
40                     limit(data_type _limit);
41                     data_type get();
42                     bool is_undef();
43                     void set(data_type _limit);
44                     void set_undef();
45             };
46         public:
```

```

48     typedef T data_type;
49     typedef element<data_type> elem;
50     bin_tree();
51     bin_tree(data_type data);
52     ~bin_tree();
53
54     virtual void insert(data_type data);
55     void insert_manually(data_type data,
56                           data_type parent_data, bool left_or_right);
57     void del(data_type data);
58
59     std::string to_string(int width=2) const;
60     std::string inorder() const;
61     std::string preorder() const;
62     std::string postorder() const;
63     int get_height() const;
64     bool check_binary_search_tree() const;
65     /*void insert_manually(data_type data, data_type parent_data, bool l
66     data_type search(data_type data);
67     void del(data_type data);
68     */
69 protected:
70     int get_height(const elem *current) const;
71     void calc_height(elem *current);
72     void calc_height_recursive(elem *current);
73
74     void set_left(elem *current, elem *left);
75     void set_right(elem *current, elem *right);
76     void set_parent(elem *current, elem *parent);
77
78     std::string inorder(const elem *current) const;
79     std::string preorder(const elem *current) const;
80     std::string postorder(const elem *current) const;
81
82     elem *root;
83
84 private:
85     virtual void insert(data_type data, elem *parent);
86     void to_vector(std::vector<std::vector<std::shared_ptr<data_type>>>
87                     const elem *current) const;
88     elem *search(data_type data, elem *current) const;
89
90     elem *minimum(elem *current) const;
91     elem *maximum(elem *current) const;
92     elem *successor(elem *current) const;
93     elem *predecessor(elem *current) const;
94
95     bool check_binary_search_tree(const elem *, limit<data_type> ulimit

```

```

96 } ;
97
98 template <typename T>
99 bin_tree<T>::bin_tree(data_type data)
100 :root(new elem(data)){}
101
102 template <typename T>
103 bin_tree<T>::bin_tree():root(nullptr){}
104
105 template <typename T>
106 bin_tree<T>::~bin_tree(){
107     if(root) {
108         delete root;
109     }
110 }
111
112 template <typename T>
113 int bin_tree<T>::get_height(const elem *current) const{
114     if(current) {
115         return current->height;
116     } else {
117         return -1;
118     }
119 }
120
121 template <typename T>
122 bool bin_tree<T>::check_binary_search_tree() const{
123     return check_binary_search_tree(root, limit<data_type>(), limit<data_type>());
124 }
125
126 template <typename T>
127 bool bin_tree<T>::check_binary_search_tree(const elem *e, limit<data_type> u,
128                                              limit<data_type> l) {
129     if(!l.limit.is_undef() && e->data <= l.limit.get()) {
130         return false;
131     }
132     if(!u.limit.is_undef() && e->data > u.limit.get()) {
133         return false;
134     }
135     if(e->left) {
136         bool res(check_binary_search_tree(e->left, limit<data_type>(e->data));
137         if(!res) {
138             return false;
139         }
140     }
141     if(e->right) {
142         return res(check_binary_search_tree(e->right, u.limit, limit<data_type>(e->data));
143     }
144     return true;

```

```

144 }
145
146 template <typename T>
147 int bin_tree<T>::get_height() const{
148     return get_height(root);
149 }
150
151 template <typename T>
152 void bin_tree<T>::calc_height(elem *current){
153     int height = -1;
154     height = get_height(current->left);
155     if(get_height(current->right) > height){
156         height = get_height(current->right);
157     }
158     current->height = ++height;
159 }
160
161 template <typename T>
162 void bin_tree<T>::calc_height_recursive(elem *current){
163     if(current->left) {
164         calc_height_recursive(current->left);
165     }
166     if(current->right) {
167         calc_height_recursive(current->right);
168     }
169     calc_height(current);
170 }
171
172 template <typename T>
173 void bin_tree<T>::insert(data_type data){
174     if(!root) {
175         root = new elem(data);
176     } else {
177         insert(data, root);
178     }
179 }
180
181 template <typename T>
182 void bin_tree<T>::insert_manually(data_type data,
183     data_type parent_data, bool left_or_right){
184     elem *parent_node(search(parent_data, root));
185     if(parent_node) {
186         if(left_or_right) {
187             set_left(parent_node, new elem(data));
188         } else {
189             set_right(parent_node, new elem(data));
190         }
191     } else {

```

```

192         std::stringstream ss;
193         ss << "Element " << parent_data << " doesn't exist.";
194         throw std::runtime_error(ss.str());
195     }
196 }
197
198 template <typename T>
199 void bin_tree<T>::insert(data_type data, elem *parent) {
200     if(data < parent->data) {
201         if(!parent->left) {
202             set_left(parent, new elem(data, parent));
203         } else {
204             insert(data, parent->left);
205         }
206     } else {
207         if(!parent->right) {
208             set_right(parent, new elem(data, parent));
209         } else {
210             insert(data, parent->right);
211         }
212     }
213 }
214
215 template <typename T>
216 void bin_tree<T>::set_left(elem *current, elem *left){
217     current->left = left;
218     calc_height(current);
219     if(current->parent) {
220         for(elem *p(current->parent); p;p=p->parent) {
221             calc_height(p);
222         }
223     }
224 }
225
226 template <typename T>
227 void bin_tree<T>::set_right(elem *current, elem *right){
228     current->right = right;
229     calc_height(current);
230     if(current->parent) {
231         for(elem *p(current->parent); p;p=p->parent) {
232             calc_height(p);
233         }
234     }
235 }
236
237 template <typename T>
238 void bin_tree<T>::set_parent(elem *current, elem *parent){
239     current->parent = parent;

```

```

240     calc_height(current);
241     for(elem *p(current->parent);p;p=p->parent) {
242         calc_height(p);
243     }
244 }
245
246 template <typename T>
247 std::string bin_tree<T>::to_string(int width) const{
248     if(root) {
249         //generate an empty-string for empty-elements
250         std::stringstream data_stream;
251         data_stream << std::setfill(' ');
252         data_stream.width(width);
253         data_stream << "";
254         std::string empty(data_stream.str());
255
256         std::vector<std::vector<std::shared_ptr<data_type>>> svec;
257         std::stringstream ret;
258
259         to_vector(svec, root);
260
261         int step=1;
262         int chars_last_line = pow(2,root->height)*(width+1)-1;
263         for(typename std::vector<std::vector<std::shared_ptr<data_type>>>::const_iterator it=svec.begin(); it != svec.end(); ++it,++step) {
264             int pos=pow(2,root->height-step+1)*width;
265             for(typename std::vector<std::shared_ptr<data_type>>::const_iterator tit != it->end(); ++tit) {
266                 data_stream.str("");
267                 if(*tit) {
268                     data_stream << std::setw(width)
269                                 << std::setfill('0') << **tit;
270                 } else {
271                     data_stream << empty;
272                 }
273                 ret << std::setw(pos) << data_stream.str();
274                 ret << std::setw(pos) << " ";
275             }
276             ret << std::endl;
277         }
278     }
279
280     return ret.str();
281 } else {
282     return "empty Tree";
283 }
284 }
285 }
286
287 template <typename T>

```

```

288 void bin_tree<T>::to_vector(
289     std::vector<std::vector<std::shared_ptr<data_type>>> &svec,
290     const elem *current) const{
291     std::vector<std::shared_ptr<data_type>> tmp_vec;
292     tmp_vec.push_back(std::shared_ptr<data_type>(new data_type(current->data
293     svec.push_back(tmp_vec);
294
295     if(current->left || current->right) {
296         if(current->left && current->right) {
297             std::vector<std::vector<std::shared_ptr<data_type>>> leftvec,rightvec;
298             to_vector(leftvec, current->left);
299             to_vector(rightvec, current->right);
300             typename std::vector<std::vector<std::shared_ptr<data_type>>>::iterator rit(rightvec.begin());
301             rit(rightvec.begin()), lit=leftvec.begin();
302             for(int i=0; i<current->height; i++) {
303                 //because first element is set above...
304                 std::vector<std::shared_ptr<data_type>> tmp;
305                 if(lit != leftvec.end()) {
306                     tmp.insert(tmp.end(),lit->begin(),lit->end());
307                     lit++;
308                 } else {
309                     for(int j=0; j<pow(2,i)/2; j++) {
310                         tmp.push_back(std::shared_ptr<data_type>());
311                     }
312                 if(rit != rightvec.end()) {
313                     tmp.insert(tmp.end(),rit->begin(),rit->end());
314                     rit++;
315                 } else {
316                     for(int j=0; j<pow(2,i)/2; j++) {
317                         tmp.push_back(std::shared_ptr<data_type>());
318                     }
319                 }
320                 svec.push_back(tmp);
321             }
322         } else if(current->left) {
323             std::vector<std::vector<std::shared_ptr<data_type>>> leftvec;
324             to_vector(leftvec, current->left);
325             for(typename std::vector<std::vector<std::shared_ptr<data_type>>>::iterator
326                 lit(leftvec.begin());lit!=leftvec.end();lit++) {
327                 std::vector<std::shared_ptr<data_type>> tmp;
328                 tmp.insert(tmp.end(),lit->begin(),lit->end());
329                 for(int i=0; i<lit->size(); i++) {
330                     tmp.push_back(std::shared_ptr<data_type>());
331                 }
332                 svec.push_back(tmp);
333             }
334         } else if(current->right) {

```

```

335         std::vector<std::vector<std::shared_ptr<data_type>>> rightvec;
336         to_vector(rightvec, current->right);
337         for(typename std::vector<std::vector<std::shared_ptr<data_type>>>
338             rit(rightvec.begin()); rit!=rightvec.end(); rit++) {
339             std::vector<std::shared_ptr<data_type>> tmp;
340             for(int i=0; i<rit->size(); i++) {
341                 tmp.push_back(std::shared_ptr<data_type>());
342             }
343             tmp.insert(tmp.end(), rit->begin(), rit->end());
344             svec.push_back(tmp);
345         }
346     }
347 } else {
348     //befuelle die restliche hoehe mit leeren zeichen...
349     if(current->parent) {
350         elem *tmp(current->parent);
351         int depth = 0;
352         while(tmp) {
353             depth++;
354             tmp = tmp->parent;
355         }
356         for(int i=depth+1; i<=root->height; i++) {
357             std::vector<std::shared_ptr<data_type>> tmp;
358             int chars_line = pow(2, i-depth);
359             for(int j=0; j<chars_line; j++) {
360                 tmp.push_back(std::shared_ptr<data_type>());
361             }
362             svec.push_back(tmp);
363         }
364     }
365 }
366 }
367
368 template <typename T>
369 std::string bin_tree<T>::inorder() const{
370     if(root) {
371         return inorder(root);
372     } else {
373         return "empty Tree";
374     }
375 }
376
377 template <typename T>
378 std::string bin_tree<T>::inorder(const elem *current) const{
379     std::stringstream ret;
380     if(current->left) {
381         ret << inorder(current->left);
382     }

```

```

383     ret << current->data << ", ";
384     if(current->right) {
385         ret << inorder(current->right);
386     }
387     return ret.str();
388 }
389
390 template <typename T>
391 std::string bin_tree<T>::preorder() const{
392     if(root) {
393         return preorder(root);
394     } else {
395         return "empty Tree";
396     }
397 }
398
399 template <typename T>
400 std::string bin_tree<T>::preorder(const elem *current) const{
401     std::stringstream ret;
402     ret << current->data << ", ";
403     if(current->left) {
404         ret << preorder(current->left);
405     }
406     if(current->right) {
407         ret << preorder(current->right);
408     }
409     return ret.str();
410 }
411
412 template <typename T>
413 std::string bin_tree<T>::postorder() const{
414     if(root) {
415         return postorder(root);
416     } else {
417         return "empty Tree";
418     }
419 }
420
421 template <typename T>
422 std::string bin_tree<T>::postorder(const elem *current) const{
423     std::stringstream ret;
424     if(current->left) {
425         ret << postorder(current->left);
426     }
427     if(current->right) {
428         ret << postorder(current->right);
429     }
430     ret << current->data << ", ";

```

```

431     return ret.str();
432 }
433
434 template <typename T>
435 typename bin_tree<T>::elem *bin_tree<T>::search(data_type data, elem *current)
436     if(current->data == data) {
437         return current;
438     } else if(current->left && data < current->data){
439         return search(data, current->left);
440     } else if(current->right){
441         return search(data, current->right);
442     } else {
443         return nullptr;
444     }
445 }
446
447 template <typename T>
448 typename bin_tree<T>::elem *bin_tree<T>::minimum(elem *current) const{
449     if(current->left) {
450         return minimum(current->left);
451     } else {
452         return current;
453     }
454 }
455
456 template <typename T>
457 typename bin_tree<T>::elem *bin_tree<T>::maximum(elem *current) const{
458     if(current->right) {
459         return maximum(current->right);
460     } else {
461         return current;
462     }
463 }
464
465 template <typename T>
466 typename bin_tree<T>::elem *bin_tree<T>::successor(elem *current) const{
467     if(current->right) {
468         return minimum(current->right);
469     } else {
470         if(current->parent && current->parent->right == current) {
471             return successor(current->parent);
472         } else {
473             return current->parent;
474         }
475     }
476 }
477
478 template <typename T>

```

```

479 typename bin_tree<T>::elem *bin_tree<T>::predecessor(elem *current) const{
480     if(current->left) {
481         return maximum(current->left);
482     } else {
483         if(current->parent && current->parent->left == current) {
484             return predecessor(current->parent);
485         } else {
486             return current->parent;
487         }
488     }
489 }
490
491 template <typename T>
492 void bin_tree<T>::del(data_type data){
493     elem *del=search(data, root);
494     if(del) {
495         std::unique_ptr<elem> r;
496
497         if(!del->left || !del->right) {
498             r.reset(del);
499         } else {
500             r.reset(successor(del));
501             del->data = r->data;
502         }
503
504         elem *p;
505         if(r->left) {
506             p = r->left;
507         } else {
508             p = r->right;
509         }
510         if(p) {
511             p->parent = r->parent;
512         }
513         if(!r->parent) {
514             root = p;
515             calc_height(root);
516         } else {
517             if(r.get()==r->parent->left) {
518                 set_left(r->parent, p);
519             } else {
520                 set_right(r->parent, p);
521             }
522         }
523
524 //this is importaint, if you forget this, every child would be deleted
525 r->left=nullptr;
526 r->right=nullptr;

```

```

527     } else {
528         std::stringstream ss;
529         ss << "Element " << data << " doesn't exist.";
530         throw std::runtime_error(ss.str());
531     }
532 }
533
534
535
536
537
538
539 template <typename T>
540 template <typename U>
541 bin_tree<T>::element<U>::element(data_type _data)
542     :data(_data), parent(nullptr), left(nullptr), right(nullptr), height(0)
543
544 template <typename T>
545 template <typename U>
546 bin_tree<T>::element<U>::element(data_type _data, element *_parent)
547     :data(_data), parent(_parent), left(nullptr), right(nullptr), height(0)
548
549 template <typename T>
550 template <typename U>
551 bin_tree<T>::element<U>::element(data_type _data, element *_parent,
552         element *_left, element *_right)
553     :data(_data), parent(_parent), left(_left), right(_right), height(0) {}
554
555 template <typename T>
556 template <typename U>
557 bin_tree<T>::element<U>::~element(){
558     if(left) {
559         delete left;
560     }
561     if(right) {
562         delete right;
563     }
564 }
565
566
567 template <typename T>
568 template <typename U>
569 bin_tree<T>::limit<U>::limit()
570     :lim(), undef(true) {}
571
572 template <typename T>
573 template <typename U>
574 bin_tree<T>::limit<U>::limit(data_type _limit)

```

```

575     :lim(_limit), undef(false) {}
576
577         void set_undef();
578
579     template <typename T>
580     template <typename U>
581     bool bin_tree<T>::limit<U>::is_undef(){
582         return undef;
583     }
584
585     template <typename T>
586     template <typename U>
587     U bin_tree<T>::limit<U>::get(){
588         return lim;
589     }
590
591     template <typename T>
592     template <typename U>
593     void bin_tree<T>::limit<U>::set(data_type _limit){
594         undef = false;
595         lim = _limit;
596     }
597
598     template <typename T>
599     template <typename U>
600     void bin_tree<T>::limit<U>::set_undef(){
601         undef = true;
602     }
603
604 #endif

```

A.2. avl_tree.h

Listing 4: Klasse avl_tree

```

1 #ifndef AVL_TREE_H
2 #define AVL_TREE_H
3
4 #include "bin_tree.h"
5
6 template <typename T>
7 class avl_tree : public bin_tree<T>{
8     private:
9         typedef T data_type;
10        typedef typename bin_tree<T>::elem elem;
11    public:
12        avl_tree():bin_tree<T>(){}
13        avl_tree(data_type data):bin_tree<T>(data){}
14        virtual void insert(data_type data){bin_tree<T>::insert(data);}

```

```

15
16     private:
17         virtual void insert(data_type data, elem *parent);
18
19         elem *rotate_to_right(elem *current);
20         elem *rotate_to_left(elem *current);
21         elem *double_rotate_right_left(elem *current);
22         elem *double_rotate_left_right(elem *current);
23     };
24
25     template <typename T>
26     void avl_tree<T>::insert(data_type data, elem *parent) {
27         if(data < parent->data) {
28             if(!parent->left) {
29                 this->set_left(parent, new elem(data, parent));
30             } else {
31                 insert(data, parent->left);
32             }
33             this->calc_height(parent);
34
35             if(this->get_height(parent->right)-this->get_height(parent->left) ==
36                 elem *father(parent->parent);
37                 if(this->get_height(parent->left->left) > this->get_height(parent->left))
38                     elem *tmp(this->rotate_to_right(parent));
39
40                     this->set_parent(tmp, father);
41
42                     if(father && father->data > tmp->data){
43                         this->set_left(father, tmp);
44                     } else if(father && father->data <= tmp->data){
45                         this->set_right(father, tmp);
46                     } else {
47                         this->root = tmp;
48                     }
49             } else {
50                 elem *tmp(this->double_rotate_left_right(parent));
51                 this->set_parent(tmp, father);
52                 if(father){
53                     this->set_right(father, tmp);
54                 } else {
55                     this->root = tmp;
56                 }
57             }
58         }
59     } else {
60         if(!parent->right) {
61             this->set_right(parent, new elem(data, parent));
62         } else {

```

```

63         insert(data, parent->right);
64     }
65     this->calc_height(parent);
66
67     if(this->get_height(parent->right)-this->get_height(parent->left) ==
68         elem *father(parent->parent);
69         if(this->get_height(parent->right->right) > this->get_height(par-
70             elem *tmp(this->rotate_to_left(parent));
71             this->set_parent(tmp, father);
72             if(father){
73                 this->set_left(father, tmp);
74             } else {
75                 this->root = tmp;
76             }
77         } else {
78             elem *tmp(this->double_rotate_right_left(parent));
79             this->set_parent(tmp, father);
80             this->set_right(father, tmp);
81             if(father){
82                 this->set_right(father, tmp);
83             } else {
84                 this->root = tmp;
85             }
86         }
87     }
88 }
89 }
90
91 template <typename T>
92 typename avl_tree<T>::elem *avl_tree<T>::rotate_to_right(elem *current){
93     if(current && current->left) {
94         elem *tmp(current->left);
95         this->set_left(current, tmp->right);
96         if(current->left) {
97             this->set_parent(current->left, current);
98         }
99
100        this->set_right(tmp, current);
101        this->calc_height(current);
102        this->calc_height(tmp);
103
104        tmp->parent=nullptr;
105        current->parent=tmp;
106        current = tmp;
107
108    }
109    return current;
110 }

```

```

111
112 template <typename T>
113 typename avl_tree<T>::elem *avl_tree<T>::rotate_to_left(elem *current){
114     if(current && current->right) {
115         elem *tmp(current->right);
116
117         this->set_right(current, tmp->left);
118         if(current->right) {
119             this->set_parent(current->right, current);
120         }
121
122         tmp->left = current;
123         this->calc_height(current);
124         this->calc_height(tmp);
125
126         tmp->parent=nullptr;
127         current->parent=tmp;
128         current = tmp;
129     }
130     return current;
131 }
132
133 template <typename T>
134 typename avl_tree<T>::elem *avl_tree<T>::double_rotate_left_right(elem *curr
135     this->set_left(current, rotate_to_left(current->left));
136     return rotate_to_right(current);
137 }
138
139 template <typename T>
140 typename avl_tree<T>::elem *avl_tree<T>::double_rotate_right_left(elem *curr
141     this->set_right(current, rotate_to_right(current->right));
142     return rotate_to_left(current);
143 }
144
145 #endif

```