Datenstrukturen und Algorithmen, SS 2019
Prof. Dr. P. Rossmanith
S. Dollase, J. Dreier, H. Lotze

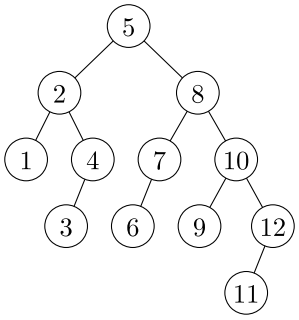
Datum: 2.5.2019

Übungsblatt mit Lösungen 04

Aufgabe T12

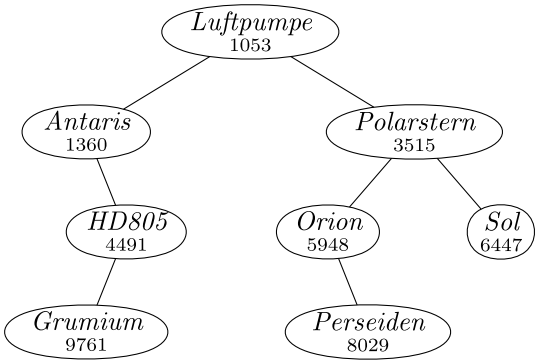
Geben Sie einen AVL-Baum der Höhe 5 an, der möglichst wenige Knoten enthält.

Lösungsvorschlag



Aufgabe T13

Wir haben diesen Treap:



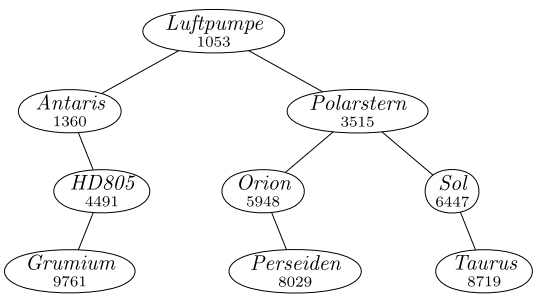
a) Fügen Sie den Schlüssel *Taurus* mit der Priorität 8719 ein.

b) Löschen Sie danach die *Luftpumpe*.

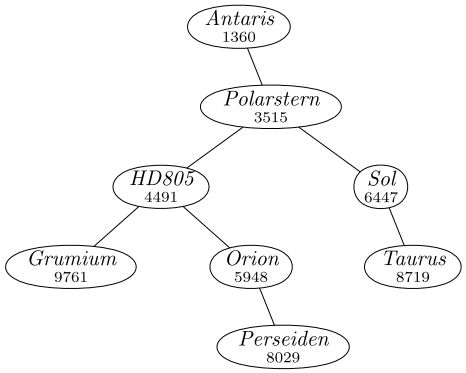
c) Fügen Sie jetzt die *Luftpumpe* wieder ein. Verwenden Sie die Priorität 2854.

Lösungsvorschlag

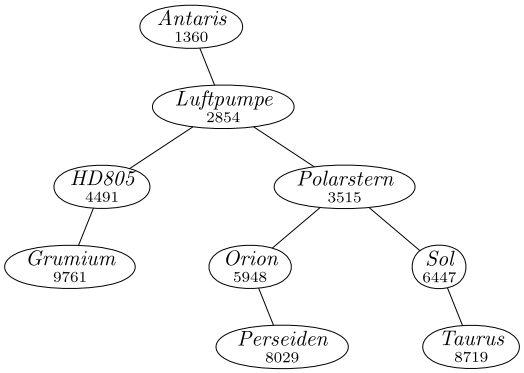
a)



b)



c)



Aufgabe T14

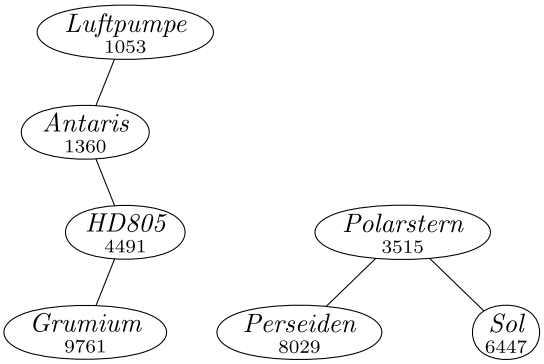
Wir möchten eine neue Operation *split*(*key*) für Treaps entwerfen. Sie soll den Treap in zwei neue Treaps aufteilen. Im ersten sollen alle Schlüssel enthalten sein, die kleiner als *key* sind, im anderen alle Schlüssel die größer als *key* sind. Der alte Treap kann dabei zerstört werden. Falls der Treap den Schlüssel *key* enthielt, dann kommt er in den beiden neuen Treaps nicht mehr vor.

Wie schnell ist Ihr Verfahren?

Was erhalten wir, wenn wir *split*(`"Orion"`) auf den ursprünglichen Treap von T13 anwenden?

Lösungsvorschlag

Ist der Schlüssel s vorhanden, wird er zunächst gelöscht. Danach wird s mit der Priorität $-\infty$ (bzw. einer Priorität, die niedriger ist als die aller anderen Knoten) neu eingefügt und dabei natürlich zur Wurzel durchrotiert. Nun ergeben der linke und rechte Unterbaum die gewünschten Treaps. Die Laufzeit für das Löschen und Einfügen von s ist linear in der Höhe des ursprünglichen Treaps und damit $O(\log n)$ im Erwartungswert.



Aufgabe T15

In dieser Aufgabe betrachten wir Splaybäume.

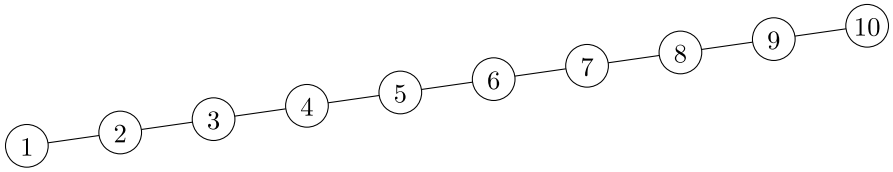
a) Was erhalten wir, wenn wir in einen anfangs leeren Splaybaum die Schlüssel $1, \ldots, 10$ in dieser Reihenfolge einfügen? Wie lange dauert es, wenn wir ähnlich mit den Schlüsseln $1, \ldots, n$ vorgehen?

b) Was erhalten wir, wenn wir in einen anfangs leeren Splaybaum wieder die Schlüssel $1, \ldots, 10$ in dieser Reihenfolge einfügen, aber diesmal jeden Schlüssel *zweimal hintereinander* einfügen?

c) Wie sieht der Baum aus, nachdem wir nach der 1 suchten?

Lösungsvorschlag

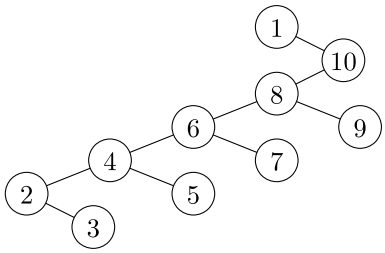
a) Bei jeder Einfügeoperation wird der eingefüge Schlüssel einmal nach links rotiert (einfaches *zag*). Nach dem Einfügen der Schlüssel $1, \ldots, 10$ sieht der Baum so aus:



Da jedes Einfügen in konstanter Zeit geschieht, ist die Laufzeit für das Einfügen der Schlüssel $1, \ldots, n$ linear.

b) Wird ein Schlüssel, der bereits in der Wurzel steht, ein zweites Mal eingefügt, wird die Wurzel ersetzt und es muss nicht rotiert werden. Deshalb ist das Ergebnis das gleiche.

c)



Aufgabe H10 (5×2 Punkte)
Betrachten Sie den Treap aus Aufgabe T13. Führen Sie folgende Operationen nacheinander auf ihm aus und zeichnen Sie jeweils die dabei entstehenden Treaps.

a) Fügen Sie *Deneb* mit Priorität 8719 ein.

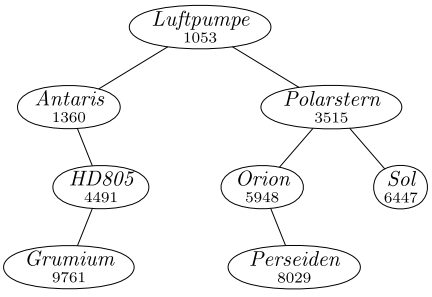
b) Löschen Sie *Antaris*.

c) Fügen Sie *Altair* mit Priorität 2854 ein.

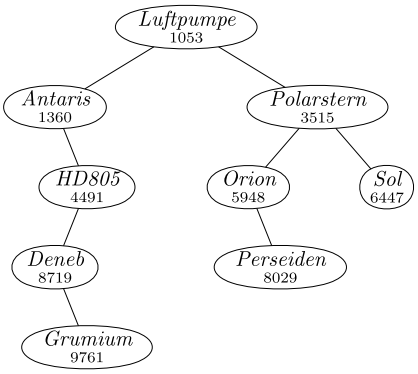
d) Löschen Sie *HD805*.

e) Fügen Sie *Wega* mit einer Priorität ein, die Sie selbst durch ein Zufallsexperiment erzeugen: Dabei wählen Sie eine Zahl zwischen 1 und 10000 gleichverteilt aus.

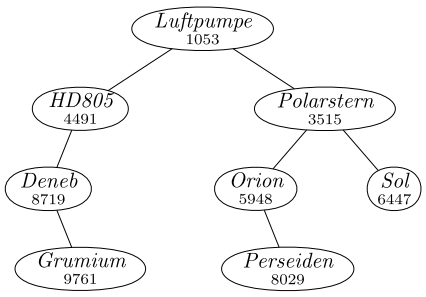
Lösungsvorschlag
Der ursprüngliche Treap steht links:



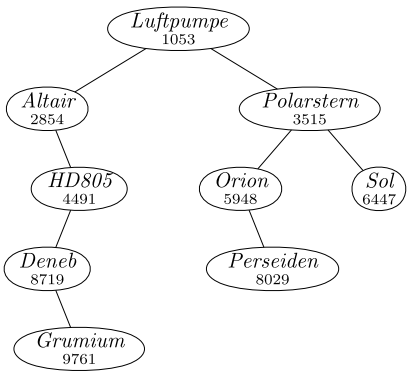
a)



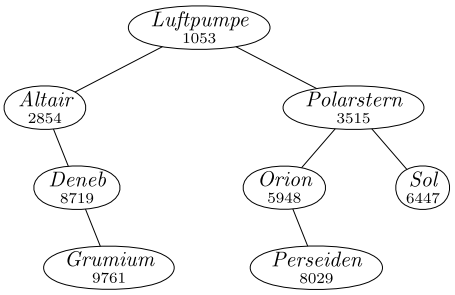
b)



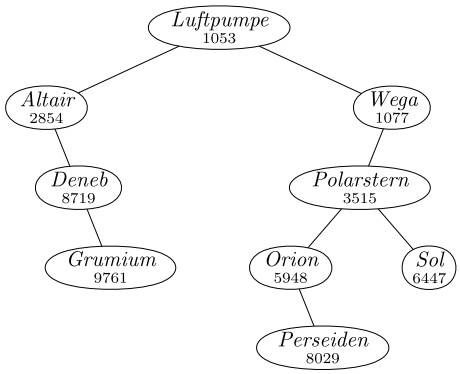
c)



d)



e)



Aufgabe H11 (10 Punkte + 5 Bonuspunkte)
Entwerfen Sie einen effizienten Algorithmus für *split*(*key*) analog zu T14 für Splaybäume.

Bonusfrage: Was können Sie über die amortisierte Laufzeit sagen, wenn wir folgende Operationen in beliebiger Reihenfolge und Kombination durchführen: Einfügen, Löschen, Suchen und *split*?

Lösungsvorschlag

Wir geben einen Algorithmus an, der auf einem Splaybaum B die Operation *split*(k) ausführt. Wir führen auf B ein *splay*(k) aus, was den Schlüssel k in die Wurzel befördert, oder, falls k in B nicht vorkommt, den nächstkleineren oder nächstgrößeren Schlüssel.

Jetzt können wir einfach aus dem linken Unterbaum der Wurzel B_1 machen und aus dem rechten Unterbaum B_2. Falls die Wurzel k ist, ignorieren wir sie. Ansonsten, fügen wir sie als Wurzel von B_1 oder B_2 hinzu.

Was passiert wenn wir folgende Operationen in beliebiger Reihenfolge und Kombination durchführen: Einfügen, Löschen, Suchen und *split*?

Da wir es mit mehreren Splaybäumen zu tun haben, können wir als Potentialfunktion einfach die Summe der Potentialfunktionen aller Bäume nehmen. Dann sieht man leicht, dass die amortisierten Kosten $O(\log n)$ sind, weil bei einem *split* nur eine *splay*-Operation ausgeführt wird und anschließend beim Zerteilen in zwei Bäume die Gesamtpotentialfunktion sogar abnimmt. Bonuspunkte bekommt, wer dies im Detail durchrechnet.

Aufgabe H12 (10 Punkte)

Einst schickte Frau Mutter, mit eiligen Worten,
klein Timmi zum Markt: „Es gibt neue Waren!
Strukturen für Daten! Verschiedenste Sorten!
Bring mir eine Queue—doch müssen wir sparen.
Drei Groschen für eine, das sollte schon passen."
So lief Timmi fort, den Marktplatz voraus
doch kaum war er dort, konnt' er sich nicht lassen
„Zwei Groschen reichen doch sicherlich aus!"
Von Gier besiegt und mit Eis in der Hand
erschrak Timmi heftig, als er sich besann
„Drei Groschen die Queue" las er dort am Stand—
er zerbrach sich den Kopf und das Eis zerann.
Mit zwei Stacks im Rucksack kehrt er schließlich heim
—wegen reichlicher Ernte bekam er sie beide—
Doch scholt' ihn die Mutter „Das kann doch nicht sein!
Stacks gehen verkehrt, weshalb ich sie meide!"
„Eine Queue, liebe Mutter, bau ich drumherum:
Enqueue-en werd' ich nur in den ersten der beid'
Und will ich dequeue-en, so füll ich sie um.
Amortisiert wird das klappen—in konstanter Zeit."

Hilf klein Timmi! Seine simulierte Queue funktioniert wie folgt:

```
int dequeue() {
  if(right.isEmpty()) {
    while(!left.isEmpty())
      right.push(left.pop());
  }
  return right.pop();
}

void enqueue(int x) {
  left.push(x);
}
```

enqueue legt also nur Elemente auf den linken Stack, *dequeue* dreht dann bei Bedarf den Inhalt des linken Stacks um: Es entfernt sukzessive alle Elemente und legt sie auf den rechten Stack. Solange der rechte Stack jetzt noch Elemente enthält, nimmt *dequeue* sie schlicht von diesem.

Kann Timmi auf diese Weise eine Queue *effizient* simulieren?

Nehmen Sie an, dass die simulierte Queue leer ist und dann n beliebige legale Operationen auf ihr ausgeführt werden (das bedeutet, dass *dequeue* nur aufgerufen wird, wenn die Queue nicht leer ist). Zeigen Sie mittels amortisierter Analyse, dass die Gesamtlaufzeit für alle Operationen $O(n)$ ist. Was ist eine geeignete Potentialfunktion? Wie ändert sie sich bei den beiden Operationen? Wie groß ist sie am Anfang und am Ende?

Lösungsvorschlag

Wir analysieren die Laufzeit mit Hilfe der amortisierten Analyse. Als Potenzialfunktion Φ wählen wir die Größe des linken Stacks l_i nach der $i$ten Operation, d.h. $\Phi(i) = c \cdot l_i$. Die Konstante $c > 0$ spezifizieren wir später. Angenommen, die $i$te Operation dauert $f(i)$ Zeitschritte. Wir zeigen, dass $f(i) + \Phi(i) - \Phi(i-1) = O(1)$ ist, denn dann ist die Gesamtlaufzeit

$$\sum_{i=1}^{n} f(i) \leq \Phi(n) - \Phi(0) + \sum_{i=1}^{n} f(i) = \sum_{i=1}^{n} f(i) + \Phi(i) - \Phi(i-1) = \sum_{i=1}^{n} O(1) = O(n).$$

Wir betrachten nun die verschiedenen Operationen.

Enqueue:
Die reellen Kosten der Einfügeoperation sind $f(i) = O(1)$, d.h. wir bekommen

$$O(1) + \Phi(i) - \Phi(i-1) = O(1) + c \cdot l_i - c \cdot l_{i-1} = O(1) + c \cdot l_i - c \cdot (l_i - 1) = O(1).$$

Dequeue:
Wir betrachten 2 Fälle: Der, wenn der rechte Stack Elemente hat, und wenn er leer ist.

1. Fall: Rechter Stack ist nicht leer.
Die reellen Kosten sind $f(i) = O(1)$ und $l_i = l_{i-1}$, d.h. wir haben

$$O(1) + c \cdot l_i - c \cdot l_{i-1} = O(1) + c \cdot l_i - c \cdot l_i = O(1).$$

2. Fall: Rechter Stack ist leer.
Hier müssen wir das "Umschwanken" beachten, welches lineare Zeit benötigt, d.h. die reellen Kosten sind $f(i) = c' \cdot l_{i-1} + O(1)$ und wir bekommen

$$c' \cdot l_{i-1} + O(1) + c \cdot l_i - c \cdot l_{i-1} = c' \cdot l_{i-1} + O(1) + 0 - c \cdot l_{i-1} = O(1).$$

Hier nehmen wir an, dass das c größer als c' gewählt wurde.

Da alle Operationen durch $O(1)$ beschränkt sind, erhalten wir für n beliebige Operationen eine Laufzeit von $O(n)$.