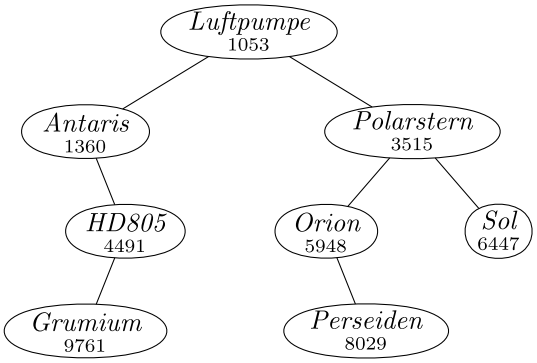
Datenstrukturen und Algorithmen, SS 2024
Prof. Dr. P. Rossmanith
D. Cloerkes, M. Gehnen, D. Mock, N. Speetzen

Abgabe: 03.05.2024, 14:00

Übungsblatt mit Lösungen 03

Aufgabe T7

Wir haben diesen Treap:



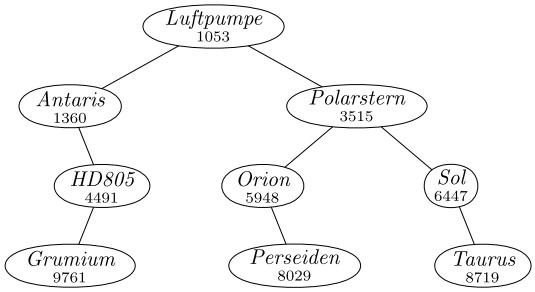
a) Fügen Sie den Schlüssel *Taurus* mit der Priorität 8719 ein.

b) Löschen Sie danach die *Luftpumpe*.

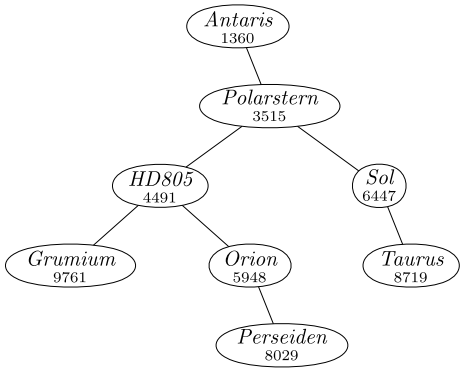
c) Fügen Sie jetzt die *Luftpumpe* wieder ein. Verwenden Sie die Priorität 2854.

Lösungsvorschlag

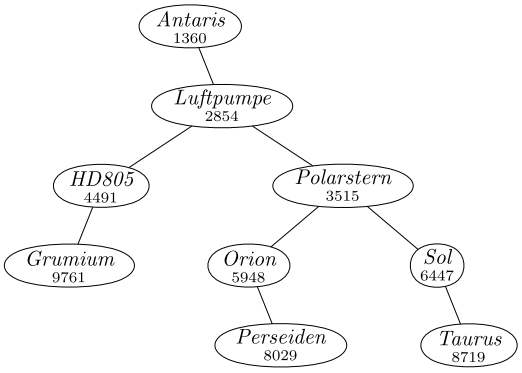
a)



b)

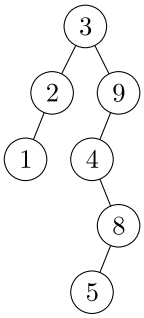


c)



Aufgabe T8

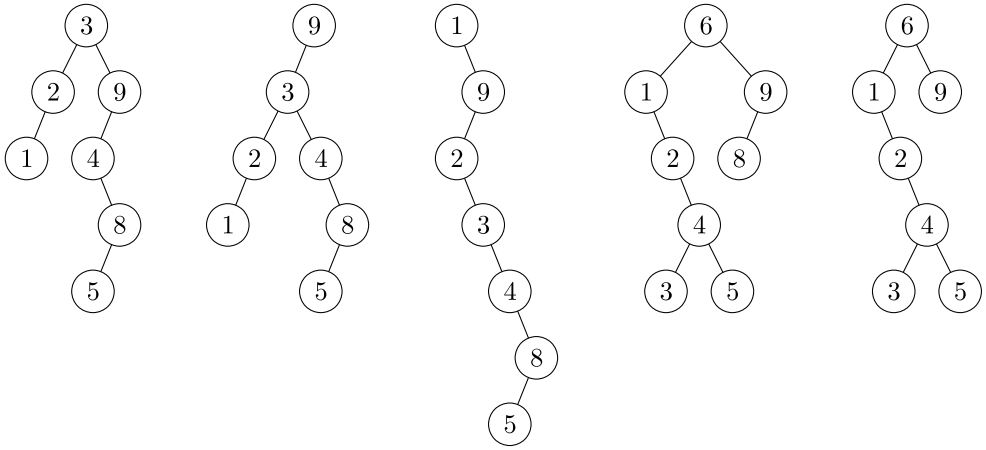
Wir betrachten folgenden Splay-Baum:



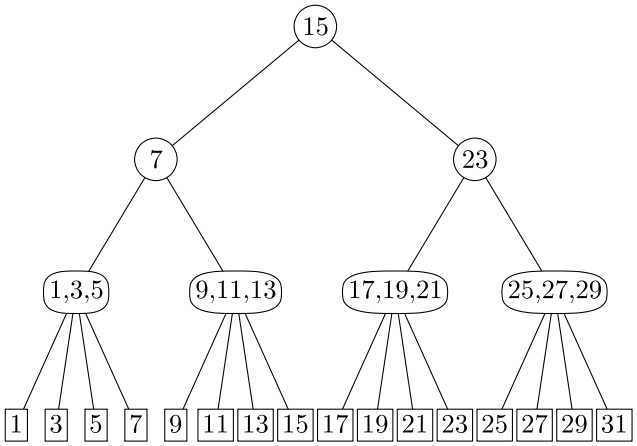
Was passiert, wenn wir in diesen Baum nach dem Schlüssel 10 suchen, dann nach 1 suchen, dann 6 einfügen und schließlich 8 löschen?

Lösungsvorschlag

Wir starten mit dem Baum links und suchen 10, suchen 1, fügen 6 ein und löschen schließlich die 8. Hier sind die Bäume, nach jedem dieser Schritte:



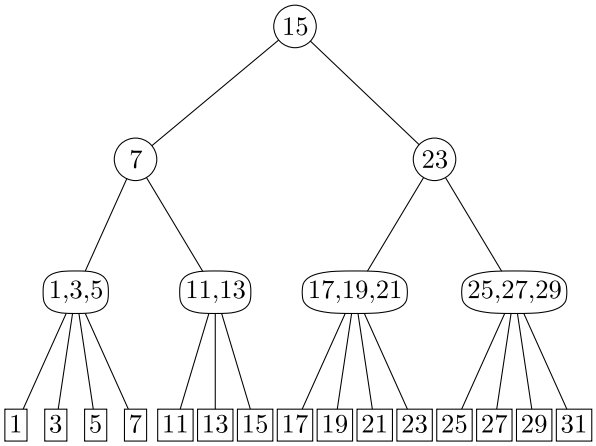
Aufgabe T9

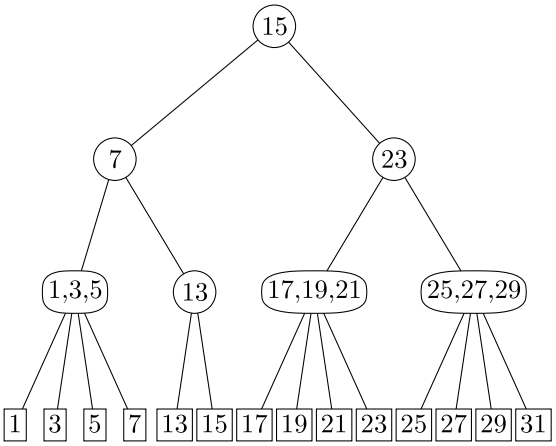


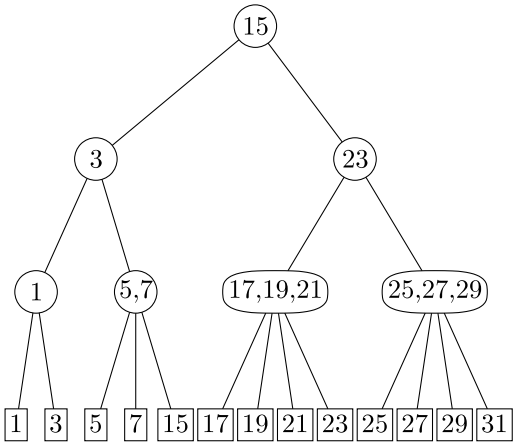
Wie sieht dieser $(2, 4)$-Baum jeweils nach dem Löschen der Schlüssel 9, 11, 13 und 17 aus? Zeichnen Sie den Baum nach jeder Löschoperation.

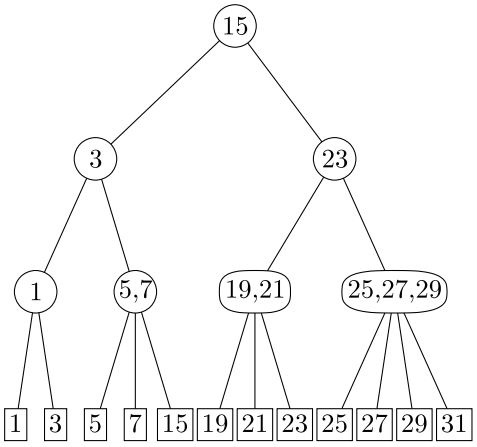
Lösungsvorschlag

Im Folgenden sehen Sie den Baum nach jeder Löschoperation.


15
7
23
1,3,5
11,13
17,19,21
25,27,29
1
3
5
7
11
13
15
17
19
21
23
25
27
29
31


15
7
23
1,3,5
13
17,19,21
25,27,29
1
3
5
7
13
15
17
19
21
23
25
27
29
31


15
3
23
1
5,7
17,19,21
25,27,29
1
3
5
7
15
17
19
21
23
25
27
29
31


15
3
23
1
5,7
19,21
25,27,29
1
3
5
7
15
19
21
23
25
27
29
31

Aufgabe T10

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();
}
```

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 Potentialfunktion 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 $O(n)$.

Aufgabe H8 (10 Punkte)
Betrachten Sie den Treap aus Aufgabe T7. Führen Sie folgende Operationen nacheinander auf ihm aus und zeichnen Sie jeweils die dabei entstehenden Treaps.

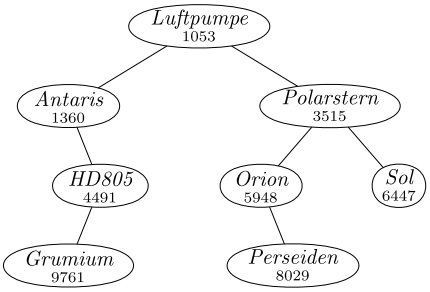
a) Fügen Sie *Aldebaran* mit Priorität 8719 ein.

b) Löschen Sie *Antaris*.

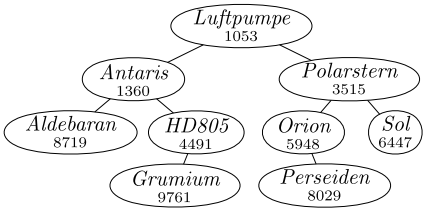
c) Fügen Sie *Dagobah* mit Priorität 2854 ein.

d) Löschen Sie *HD805*.

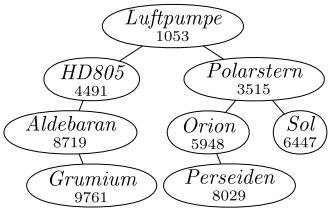
Lösungsvorschlag
Der Treap sah so aus:



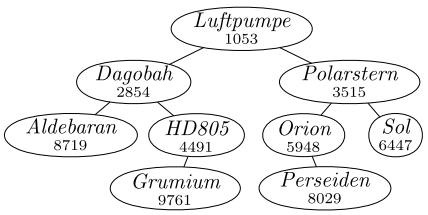
a)



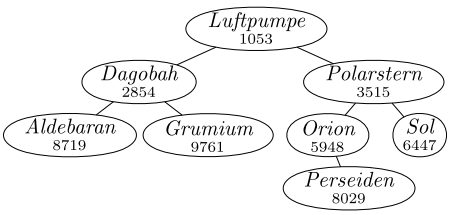
b)



c)



d)



Aufgabe H9 (10 Punkte)

Entwerfen Sie einen effizienten Algorithmus für $split(key)$, der für einen Splaybaum zwei Splaybäume ausgibt, einen mit allen Knoten kleiner key und einen mit allen Knoten größer key. 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.

Aufgabe H10 (10 Punkte)

In der Vorlesung wurde für (a, b)-Bäume verlangt, dass $b \geq 2a - 1$ gilt. Daher gibt es $(2, 3)$-Bäume, aber keine $(3, 4)$-Bäume.

Warum gilt diese Einschränkung? Was für Probleme könnte es beispielsweise mit $(3, 4)$-Bäumen geben?

Lösungsvorschlag

Aus der Definition von (a, b)-Bäumen wissen wir, dass die Wurzel zwischen 2 und b Kinder - hier also zwischen 2 und 4 - haben muss. Gleichzeitig verlangt die Definition, dass jeder innere Knoten außer der Wurzel zwischen a und b Knoten als Kinder hat - hier also zwischen 3 und 4.

Wir betrachten nun den Fall, dass wir genau fünf Schlüssel einfügen möchten - dies sollte in jedem (a, b)-Baum möglich sein. Da die Wurzel zwischen 2 und b Kinder hat, muss mindestens eines dieser Kinder ein innerer Knoten sein - wir können nicht alle 5 Schlüssel als Blätter an die Wurzel anhängen. Da alle Blätter den gleichen Abstand zur Wurzel haben müssen, folgt daraus, dass jedes Kind der Wurzel ein innerer Knoten sein muss. Da wiederum jeder innere Knoten zwischen 3 und 4 Kinder haben muss, wir also mindestens 6 Kinder bräuchten, jedoch nur 5 Elemente zur Verfügung haben, ist es damit nicht möglich, genau 5 Knoten in einen $(3, 4)$-Baum einzufügen.

Aufgabe H11 (10 Punkte)

An einem malerischen Ort südlich des Informatikzentrums gibt es zwei Pizzerien, die ein gemeinsames Problem haben: Braune Pilze, die scheinbar zufällig immer an den unpassendsten Stellen aus dem Boden sprießen, und sicherlich nicht gut für das Geschäft sind.

Der Einfachheit halber nehmen wir an, dass die Restaurants lange Flure seien, wo es zwischen Eingang (links) und Küche (rechts) nur eine Dimension gibt. Außerdem nehmen wir an, dass nie an zwei Stellen gleichzeitig Pilze erscheinen. Ein Pilz erscheint also erst, nachdem der vorherige Pilz durch ein kompliziertes Verfahren („draufspringen“) entfernt wurde. Ignorieren der Pilze ist allerdings keine Option.

- Bei Francesco und Andrea geht stets derjenige zu dem Pilz, der am nähesten an ihm steht. Der andere bleibt stehen. Falls beide gleich weit weg sind, wird zufällig entschieden.
- Wenn Luigi und Mario in ihrer Pizzeria einen Pilz zwischen sich sehen, so laufen beide gleich schnell von beiden Seiten auf ihn zu, bis einer ihn erreicht hat. Ist der Pilz außerhalb der beiden, so geht nur der näheste zu dem Pilz und der weiter entfernte bleibt stehen.

Nehmen wir an, dass alle vier lauffaul sind: um welchen Faktor a muss das Personal der beiden Pizzerien mit den jeweiligen Strategien schlimmstenfalls mehr laufen, als wenn sie bereits die am Anfang des Tages gewusst hätten wo die Pilze wann auftauchen werden und einen perfekten Plan gemacht hätten (dies nennt man auch Approximationsgüte)?

Hinweis 1: Bei der Luigis und Marios Strategie bietet sich eine amortisierte Analyse an. Als Potentialfunktion der Strategie nach Pilz i eignet sich $\Phi(S_i) = 2 \cdot dist(L_i, L_i^{plan}) + 2 \cdot dist(R_i, R_i^{plan}) + dist(L_i, R_i)$, wobei L_i bzw. R_i die Positionen von der linken bzw. rechten Person in der Sitution nach Pilz i sei, und L_i^{plan} (bzw. R_i^{plan}) die Positionen nach einem perfekten Plan sind. $dist(a, b)$ sei die Distanz zwischen a und b.

Hinweis 2: Die Potentialfunktion ist am Anfang nicht null, sondern hat die Größe $dist(L_0, R_0)$. Damit können wir die Approximationsgüte mit einer additiven Konstante bestimmen (Mario und Luigi laufen also maximal a-mal so viel wie im perfekten Plan, plus einmalig zusätzlich bis zu $dist(L_0, R_0)$).

Hinweis 3: Für die Analyse nehmen wir an, dass sich immer erst der Kellner der perfekten Lösung bewegt, danach dann in unserer Strategie: wie häufig bei amortisierter Analyse bietet es sich an, eine Fallunterscheidung zu betrachten. Wie ändert sich die Potentialfunktion,

- wenn sich erst jemand in der perfekten Lösung bewegt?
- wenn der Pilz außerhalb von Mario und Luigi erscheint, und einer dorthin geht?
- wenn der Pilz innerhalb der beiden erscheint, und sich beide bewegen?

Nur wenn der sich jemand in perfekten Lösung sich bewegt, dürfen wir „Guthaben“in der Potentialfunktion aufbauen. Wenn wir danach jemanden nach unserer Strategie bewegen, müssen wir dies vom Guthaben bezahlen.

Lösungsvorschlag

Probleme wie diese, wo der Input erst über Zeit bekannt wird, werden als *Online-Probleme* bezeichnet. Dieses Online-Problem ist auch bekannt als *k-Server-Problem* (in unserem Fall mit $k = 2$, da wir zwei Server/Personen haben). Es wird auf verschiedenen Metriken angeguckt, beispielsweise auf Graphen, in unserem Fall ist dies eine endliche Linie. Die Strategie von Andrea und und Francesco ist die sogenannte *Greedy*-Strategie, die von Mario und Luigi das sogenannte *Double Coverage*.

Die Greedy-Strategie ist eine eher simple Strategie, die leider auch anfällig für simple Instanzen ist: angenommen, es gibt zwei Stellen direkt neben dem linken Server, an denen abwechselnd immer wieder Pilze erscheinen. Dann würde die Greedy-Strategie den einen Server immer wieder hin- und her bewegen, damit also unbegrenzt lange gehen, während eine perfekte Strategie einfach beide Server genau auf die beiden Stellen platziert (und damit nur konstante Kosten hat). Die Approximationsgüte kann also beliebig schlecht werden.

Mithilfe der im Hinweis angegebenen Potentialfunktion (und der Betrachtung, dass sich erst die Server der perfekten Lösung bewegen, und dann unsere der Online Strategie), gibt es folgende Änderungen der Potentialfunktion beim Double Coverage:

- Bewegt sich ein Server (o.B.d.A: der linke Server) in der optimalen Lösung um eine Entfernung d, so wird $dist(L_i, L_i^{plan})$ um maximal d größer. Durch den Vorfaktor 2 der Potentialfunktion steigt die Potentialfunktion also um bis zu $2d$. Diese 2 ist auch am Ende die gesuchte Approximationsgüte.

- Erscheint ein Pilz außerhalb der beiden Server (o.B.d.A.: links der Server) in der Double-Coverage Strategie der Aufgabe, so bewegt sich der nächste Server (der linke) dorthin, der rechte bleibt stehen. Wenn sich der linke Server hierfür um d bewegt, so wächst $dist(L_i, R_i)$ um d, und $2 \cdot dist(L_i, L_i^{plan})$ sinkt um $2d$ (da in der perfekten Lösung ja auch ein Server auf dem Pilz stehen muss, muss der linke Server der perfekten Lösung ja entweder dort oder noch weiter links stehen - der Abstand zwischen dem linken Server in der perfekten Lösung und unserer Strategie sinkt also um d). Insgesamt sinkt die Potentialfunktion also um d, genau der Betrag, den wir für die Bewegung des linken Servers bezahlen müssen.

- Erscheint der Pilz innerhalb der beiden Server, so bewegen sich beide um eine Distanz d dorthin. Wir müssen also die Potentialfunktion um $2d$ verkleinern: Dies ist kein Problem, da ja schon $dist(L_i, R_i)$ um $2d$ sinkt. Da wieder ein Server der perfekten Lösung genau auf dem Pilz stehen muss (egal ob L oder R), sinkt der Abstand zwischen diesem Server und dem Server in unserer Strategie ebenfalls um d. Damit können wir den (eventuell um bis zu d) gestiegenen Abstand zwischen dem anderen Server und seiner Position in der perfekten Lösung kompensieren, da sich dies im schlimmsten Fall genau ausgleicht.

Insgesamt schreiben wir uns also für jede Bewegung der perfekten Lösung die doppelte Entfernung als Guthaben auf, und stellen durch die Analyse und der Potentialfunktion sicher, dass die Double-Coverage Strategie mit diesem Guthaben auskommt. Damit erhalten wir die Approximationsgüte von 2 (die bei Online-Algorithmen auch manchmal *Competitive Ratio* genannt wird).
Die Strategie lässt sich auch für $k > 2$ verwenden, dann wird die Potentialfunktion allerdings ein wenig komplizierter, und der Algorithmus liefert keine Approximationsgüte von 2, sondern von k. Bei Interesse: dies ist in so ziemlich jedem Buch zu Online-Algorithmen erklärt, beispielsweise von Allan Borodin oder Dennis Komm.