3 Suchalgorithmen

Es gibt viele Anwendungen, deren Kern-Anforderung die Realisierung einer schnellen Suche ist. Tatsächlich ist überhaupt einer der wichtigsten Einsatzzwecke eines Computers die Speicherung großer Datenmengen in sog. Datenbanken und das schnelle Wiederfinden (engl: Retrieval) von Informationen in dieser Datenmenge.

Ungeschickt implementierte Suchfunktionen kommen schon bei einigen Gigabyte an Daten an ihre Grenzen und werden bei sehr großen Datenmengen vollkommen nutzlos. Und wir haben es mit zunehmend riesigen Datenmengen zu tun, die noch vor 10 Jahren unvorstellbar waren. Ein Vergleich mit der größten Bibliothek der Welt – der British Library, deren Lesesaal in Abbildung 3.2 zu sehen ist,– kann ein „Gefühl“ dafür geben, mit welchen Datenmengen wir es zu tun haben: Die British Library hat mehr als 150 Mio. Exemplare (also Bücher, Zeitschriften usw.).

Abb. 3.1: *Ein Karteikartensystem. Datenbank- und Information-Retrieval-Systeme sind digitale „Nachbauten“ solcher (und ähnlicher) Systeme.*

Gehen wir von 1500 Byte an Daten pro Buchseite aus, und einer durchschnittlich 300

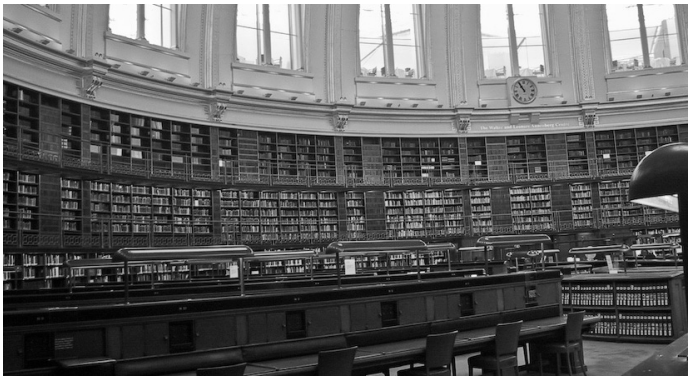


Abb. 3.2: *Der Lesesaal der berühmten „British Library“ – der größten Bibliothek der Welt mit einem Bestand von mehr als 150 Mio Exemplaren.*

Seiten pro Buch, so überschlagen wir, dass die British Library etwa 75000 Gigabyte oder 75 Terabyte an Daten gespeichert hat. Das Unternehmen Google, dagegen, unterhält weltweit laut groben Schätzungen über eine Million Server auf denen, davon können wir

ausgehen, durchschnittlich mehrere Terabyte an Daten gespeichert sind; wir können also grob schätzen, dass Google deutlich mehr als 1000000 Terabyte, also mehr als 1000 Petabyte an Daten auf den Firmen-internen Servern gespeichert hat, d. h. deutlich über 10000 mal, vielleicht sogar 100000 mal, mehr Daten als sich in der gesamten British Library befinden; Abbildung 3.3 deutet einen graphischen Vergleich dieser Datenmengen an. Ferner geht eine Studie von Cisco davon aus, dass in 2 bis 3 Jahren *täglich* mehr

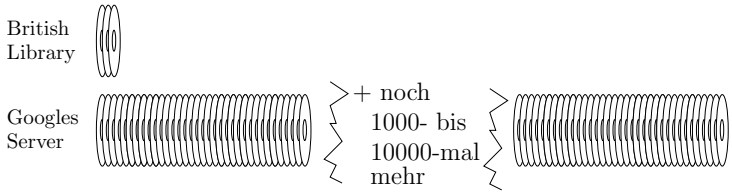


Abb. 3.3: *Große Datenmengen im Vergleich.*

als 2000 Petabyte an Daten übers Internet verschickt werden.

Das Durchsuchen einer einfachen Liste der Länge n benötigt $O(n)$ Schritte. Sind über die Liste keine besonderen Eigenschaften bekannt, kommt man nicht umhin, die ganze Liste einfach linear von „vorne“ bis „hinten“ zu durchsuchen. Hat man es mit einer großen Datenmenge zu tun – etwa mit einer Größe von mehreren Giga-, Tera- oder Petabyte – so ist ein Algorithmus mit Suchdauer von $O(n)$ vollkommen nutzlos.

Aufgabe 3.1

Angenommen, ein (nehmen wir sehr recht schneller) Rechner kann ein Byte an Daten in 50 ns durchsuchen. Wie lange braucht der Rechner, um eine Datenbank einer Größe von 100 GB / 100 TB / 100 PB zu durchsuchen, wenn der Suchalgorithmus

(a) ... eine Laufzeit von $O(n)$ hat?

(b) ... eine Laufzeit von $O(\log(n))$ hat – nehmen Sie an, die Laufzeit wäre proportional zu $\log_2 n$ (was durchaus sinnvoll ist, denn meistens werden bei solchen Suchen binäre Suchbäume verwendet)?

In diesem Kapitel lernen wir die folgenden Suchtechniken kennen:

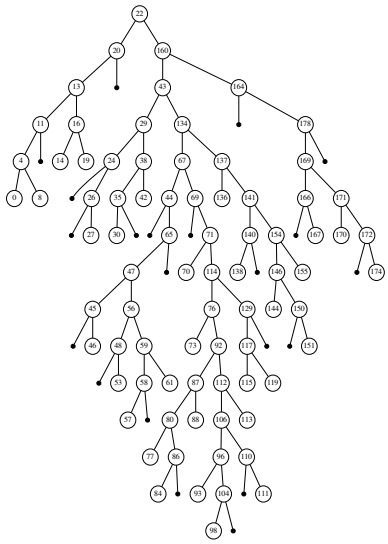
1. Suchen mittels binären Suchbäumen. Mittlere Suchlaufzeit (vorausgesetzt die Bäume sind balanciert) ist hier $O(\log n)$.
2. Suchen mittels speziellen balancierten binären Suchbäumen: den AVL-Bäumen und den rot-schwarz-Bäumen. Worst-Case-Suchlaufzeit ist hier $O(\log n)$.
3. Suchen mittels Hashing. Die Suchlaufzeit ist hier (unter gewissen Voraussetzungen) sogar $O(1)$.
4. Unterstützung von Suchen mittels eines Bloomfilters, einer sehr performanten randomisierten Datenstruktur die allerdings falsche (genauer: falsch-positive) Antworten geben kann.

5. Suchen mittels Skip-Listen. Eine Skip-Liste ist eine randomisierte Datenstruktur, deren Struktur (auf den ersten Blick) einer verketteten Liste gleicht. Die Suchlaufzeit ist hier allerdings $O(\log n)$.
6. Suchen mittels Tries und Patricia. Diese Datenstrukturen sind besonders für textbasierte Suchen geeignet und in vielen Suchmaschinen verwendet. Die Suchlaufzeit ist hier nicht abhängig von der Anzahl der enthaltenen Datensätze sondern alleine von der Länge des zu suchenden Wortes und beträgt $O(\text{Wortlänge})$.

3.1 Binäre Suchbäume

Binäre Suchbäume stellen die wohl offensichtlichste, zumindest am längsten bekannte Art und Weise dar, Schlüssel-Wert-Paare so zu ordnen, dass eine schnelle Suche nach Schlüsselwerten möglich ist. Binäre Suchbäume wurden Ende der 50er Jahre parallel von mehreren Personen gleichzeitig entdeckt und verwendet.

Die Performanz der Suche kann jedoch beeinträchtigt sein, wenn der binäre Suchbaum zu unbalanciert ist, d. h. wenn sich die Höhe des linken Teilbaums zu sehr von der Höhe des rechten Teilbaums unterscheidet – der Knoten mit der Markierung „44“ in dem rechts dargestellten binären Suchbaum ist etwa recht unbalanciert: Die Höhe des linken Teilbaums ist 0; die Höhe des rechten Teilbaums ist dagegen 6.



Ein binärer Suchbaum ist ein Baum, dessen Knoten Informationen enthalten. Jeder Knoten erhält einen eindeutigen Wert, auch *Schlüssel* genannt, über den man die enthaltenen Daten wiederfinden kann. Wir nehmen also an, dass in einem Suchbaum jedem Knoten v ein bestimmter Schlüsselwert $v.key$ zugeordnet ist. Ein *binärer Suchbaum* ist ein Suchbaum mit folgenden beiden Eigenschaften:

1. Jeder Knoten hat höchstens zwei Kinder.
2. Für jeden inneren Knoten v, d. h. Knoten mit Kindern, gilt: für *jeden* Knoten l des linken Teilbaums ist $l.key \leq v.key$ und für *jeden* Knoten r des rechten Teilbaums ist $r.key \geq v.key$.

Abbildung 3.4 zeigt ein Beispiel eines binären Suchbaums.

Ein binärer Suchbaum wird oft verwendet, um (den abstrakten Datentyp des) *Dictionaries* zu implementieren. Ein Dictionary enthält eine Sammlung von Schlüssel-Wert-Paaren und unterstützt effizient eine Suchoperation nach Schlüsseln, eine Einfügeoperation und eine Löschoperation. Pythons Dictionaries sind jedoch nicht über Suchbäume,

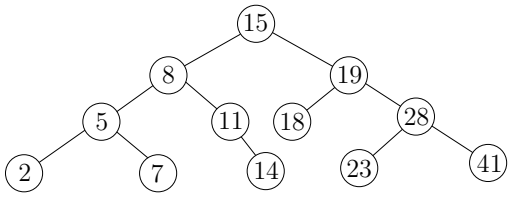


Abb. 3.4: *Beispiel eines binären Suchbaums. Man sieht, dass alle Schlüssel im linken Teilbaum eines jeden Knotens immer kleiner, und alle Werte im rechten Teilbaum eines jeden Knotens immer größer sind als der Wert des jeweiligen Knotens.*

sondern über Hash-Tabellen realisiert.

3.1.1 Repräsentation eines binären Suchbaums

Es gibt mehrere Möglichkeiten, Bäume, insbesondere binäre Bäume, in Python zu repräsentieren. Am einfachsten ist die Verwendung von geschachtelten Listen bzw. geschachtelten Tupeln oder geschachtelten Dictionaries – siehe auch Abschnitt 1.4 für weitere Details hierzu. So könnte beispielsweise das folgende geschachtelte Tupel den Binärbaum aus Abbildung 3.4 repräsentieren:

$$\begin{aligned} tSkript2 = (15,\ &(8,\ \ (5,\ 2,\ 7),\ \ (11,\ (),\ 14)\ \backslash \\ &(19,\ \ (18,(),()),\ \ (28,\ 23,\ 41) \end{aligned}$$

Dies ist eine einfache und übersichtliche Darstellung, die wir auch tatsächlich an anderer Stelle bei der Repräsentation von Binomial-Heaps so verwenden (siehe Abschnitt 4.2) die jedoch zwei entscheidende Nachteile hat, die in diesem Falle relativ schwer wiegen: Zum Einen ist sie wenig typsicher und bringt entsprechend viele Freiheitsgrade mit sich: Ob man beispielsweise ein Blatt als (18,(),()) , als (18,*None*,*None*) oder einfach als 18 repräsentiert, ist nicht direkt festgelegt. Zum Anderen ist sie schlecht erweiterbar: Möchte man etwa bestimmte Eigenschaften (wie etwa die Höhe oder die Farbe) eines Knoten mitverwalten, so läuft man hier Gefahr den gesamten Code ändern zu müssen.

Man kann die Repräsentation von Binärbäumen typsicherer gestalten, indem man eine eigens definierte Klasse verwendet. Wir nennen diese *BTree*; die Definition der Klasse zusammen mit der zugehörigen Konstruktorfunktion *__init__* ist in Listing 3.1 gezeigt.

```
1 class BTree(object):
2   def __init__ ( self , key,  ltree=None, rtree=None, val=None):
3       self . ltree  = ltree
4       self . rtree  = rtree
5       self .key = key
6       self . val  = val
```

Listing 3.1: *Ein Ausschnitt der Definition der Klasse BTree*

Hierbei sind die Parameter *ltree*, *rtree* und *val* der Funktion *__init__* sog. *benannte Parameter* (siehe Anhang A.3.4).

Ein einfacher Binärbaum, bestehend aus nur einem Knoten mit Schlüsselwert 15, kann folgendermaßen erzeugt werden:

$b = BTree(15)$

Die benannten Parameter werden nicht spezifiziert und erhalten daher ihren Default-Wert „*None*“.

Der in Abbildung 3.4 dargestellte binäre Suchbaum könnte in Python durch folgenden Wert repräsentiert werden.

$$\begin{aligned} binTree = BTree(15,&BTree(8,\ BTree(5,\ BTree(2),\ BTree(7)),\\ &\quad BTree(11,\ None,\ BTree(14))),\\ &BTree(19,\ BTree(18),\\ &\quad BTree(28,\ BTree(23),\ BTree(41)))) \end{aligned}$$

Der Einfachheit halber wurden den einzelnen Knoten nur Schlüsselwerte (das *key*-Attribut) gegeben, jedoch keine eigentlichen Daten (das *val*-Attribut).

Man sollte Zugriffs- und Updatefunktionen für die Klasse *BTree* hinzufügen, indem man entsprechende Instanzen der Klassenfunktionen *__getitem__* und *__setitem__* implementiert; zusätzlich könnte auch eine Instanz der Klassenfunktion *__str__* nützlich sein, die eine gut lesbare Form eines *BTree*s als String zurückliefert. Diese Implementierungsarbeit überlassen wir dem Leser.

Aufgabe 3.2

Implementieren Sie eine Instanz der Klassenfunktion *__str__*, die *BTree*s in einer gut lesbaren Form ausgeben kann.

Aufgabe 3.3

Implementieren Sie als Klassenfunktion von *BTree* eine Funktion *height*, die die Höhe des jeweiligen Binärbaums zurückliefert.

Aufgabe 3.4

Instanziieren Sie die Klassenfunktion *__len__* für die Klasse *BTree*, die die Anzahl der Knoten des jeweiligen *BTree*s zurückliefern soll.

3.1.2 Suchen, Einfügen, Löschen

Suchen. Am einfachsten kann die Suche implementiert werden. Angenommen der Schlüssel *key* soll gesucht werden, so wird zunächst der Schlüssel $r.key$ des Wurzelknotes r mit *key* verglichen. Falls *key* mit dem Schlüssel des Wurzelknotens übereinstimmt,

wird der im Wurzelknoten gespeicherte Wert $r.val$ zurückgegeben. Ist $key<r.key$, so muss sich aufgrund der Eigenschaften eines binären Suchbaums der Schlüsselwert im linken Teilbaum befinden, es wird also rekursiv im linken Teilbaum weitergesucht; ist $key>r.key$, wird rekursiv im rechten Teilbaum weitergesucht. Listing 3.2 zeigt eine Implementierung als Methode *search* der Klasse *BTree*.

```
1  class BTree(object):
2      ...
3    def search( self , key):
4        if key==self.key:
5            return self    # Rek.Abbr.: s gefunden.
6        elif key < self.key:
7            if self.ltree==None:
8                return None # Rek.Abbr.: s nicht gefunden.
9            else:
10               return self.ltree.search(key) # Rekursiver Aufruf
11       elif key > self.key:
12           if self.rtree==None:
13               return None # Rek.Abbr.: s nicht gefunden.
14           else:
15               return self.rtree.search(key) # Rekursiver Aufruf
```

Listing 3.2: *Implementierung der Suche im Binärbaum durch die Klassenfunktion BTree.search(key);*

In Zeile 4 wird getestet, ob der Schlüssel der Wurzel des aktuellen Binärbaums gleich dem zu suchenden Schlüssel ist; dann wird der Wert des Knotens $self.val$ zurückgeliefert. Falls der Schlüssel kleiner als der Schlüssel des aktuellen Knotens ist (Zeile 6), wird rekursiv im linken Teilbaum $self.ltree$ weitergesucht. Falls der Suchschlüssel größer ist (Zeile 11), wird rekursiv im rechten Teilbaum $self.rtree$ weitergesucht. Falls es keinen linken bzw. rechten Teilbaum mehr gibt, so wurde der Schlüssel nicht gefunden und es wird *None* zurückgeliefert (Zeile 8 und Zeile 12).

Aufgabe 3.5

Schreiben Sie die Funktion *search* iterativ.

Aufgabe 3.6

Schreiben Sie eine Methode *BinTree.minEl*() und eine Methode *BinTree.maxEl*(), die effizient das maximale und das minimale Element in einem binären Suchbaum findet.

Einfügen. Soll der Schlüssel *key* in einen bestehenden Binärbaum eingefügt werden, so wird der Baum von der Wurzel aus rekursiv durchlaufen – ähnlich wie bei der in Listing 3.2 gezeigten Suche. Sobald dieser Durchlauf bei einem Blatt v angekommen ist, wird

ein neuer Knoten an dieses Blatt angehängt; entweder als linkes Blatt, falls $v.key > key$, oder andernfalls als rechtes Blatt. Listing 3.3 zeigt die Implementierung als Methode $insert(key, val)$ der Klasse $BTree$.

```
1  class BTree(object):
2      ...
3      def insert( self ,key, val ):
4          if key < self.key:
5              if self. ltree == None:
6                  self. ltree = BTree(key,None,None,val) # Rek.Abbr: key wird eingefügt
7              else: self. ltree . insert(key, val)
8          elif key > self.key:
9              if self. rtree == None:
10                 self. rtree = BTree(key,None,None,val) # Rek.Abbr: key wird eingefügt
11             else: self. rtree . insert(key, val)
```

Listing 3.3: *Implementierung der Einfüge-Operation im Binärbaum durch die Methode insert (key, val).*

Falls der einzufügende Schlüssel *key* kleiner ist, als der Schlüssel an der Wurzel des Baumes *self .key*, und noch kein Blatt erreicht wurde, wird im linken Teilbaum *self . ltree* durch einen rekursiven Aufruf (Zeile 7) weiter nach der Stelle gesucht, an die der einzufügende Schlüssel passt. Falls der einzufügende Schlüssel *key* größer ist, als der Schlüssel an der Wurzel des Baumes und noch kein Blatt erreicht wurde, so wird im rechten Teilbaum (Zeile 11) weiter nach der passenden Einfügestelle gesucht. Falls die Suche an einem Blatt angelangt ist (falls also gilt *self . ltree*==*None* bzw. *self . rtree*==*None*), so wird der Schlüssel *key* als neues Blatt eingefügt – zusammen mit den zugehörigen Informationen *val*, die unter diesem Schlüssel abgelegt werden sollen. Dies geschieht in Listing 3.3 in den Zeilen 6 und 10.

Aufgabe 3.7

(a) In den in Abbildung 3.4 dargestellten binären Suchbaum soll der Schlüssel 22 eingefügt werden. Spielen Sie den in Listing 3.3 gezeigten Algorithmus durch; markieren Sie diejenigen Knoten, mit denen der Schlüsselwert 22 verglichen wurde und stellen Sie dar, wo genau der Schlüsselwert 22 eingefügt wird.

(b) Fügen Sie in den in Abbildung 3.4 dargestellten binären Suchbaum nacheinander die Werte $4 - 13 - 12 - 29$ ein. Spielt die Einfügereihenfolge eine Rolle?

(c) Fügen Sie in den in Abbildung 3.4 dargestellten binären Suchbaum nacheinander derart 8 Werte so ein, so dass der Baum danach eine Höhe von 10 hat.

Aufgabe 3.8

Der in Listing 3.2 gezeigte Algorithmus zum Einfügen in einen Binärbaum berücksichtigt nicht den Fall, dass der einzufügende Schlüssel x bereits im Baum vorhanden ist.
Erweitern Sie die Methode *insert* so, dass dieser Fall sinnvoll angefangen wird.

Aufgabe 3.9

Schreiben Sie die Methode *insert* iterativ.

Löschen. Welches Verfahren zum Löschen eines Knotens v in einem binären Suchbaum angewendet wird, hängt davon ab, ob der zu löschende Knoten ein Blatt ist, *ein* Kind besitzt oder *zwei* Kinder besitzt:

- Handelt es sich bei dem zu löschenden Knoten um ein Blatt, so wird dieses einfach gelöscht.
- Hat der zu löschende Knoten ein Kind, so wird einfach dieses Kind an die Stelle des zu löschenden Knotens gesetzt.
- Hat der zu löschende Knoten zwei Kinder – dies ist der schwierigste Fall – so geht man wie folgt vor: Man ersetzt den zu löschenden Knoten mit dem minimalen Knoten des rechten Teilbaums. Dieser minimale Knoten des rechten Teilbaums hat höchstens ein (rechtes) Kind und kann somit einfach verschoben werden – analog wie beim Löschen eines Knotens mit nur einem Kind.

In Abbildung 3.5 ist der Löschvorgang für die beiden Fälle, in denen der zu löschende Knoten Kinder hat, graphisch veranschaulicht.

Es gibt hier, wie in vielen anderen Fällen auch, grundsätzlich zwei Möglichkeiten, das Löschen zu implementieren: nicht-destruktiv oder destruktiv. Bei einer nicht-destruktiven Implementierung bleibt der „alte“ binäre Suchbaum unangetastet. Stattdessen wird als Rückgabewert ein „neuer“ binärer Suchbaum konstruiert (der durchaus Teile des „alten“ Suchbaums enthalten kann), der das zu löschende Element nicht mehr enthält. Eine Funktion, die nicht-destruktive Updates verwendet entspricht also am ehesten einer mathematischen Funktion: Sie bekommt einen Eingabewert (hier: einen zu modifizierenden Binärbaum) und produziert einen Ausgabewert (hier: einen Binärbaum, aus dem das gewünschte Element gelöscht wurde). Nicht-destruktive Implementierungen sind häufig anschaulich und kompakt; ein Nachteil ist jedoch der höhere Speicherplatzverbrauch. Ein guter Compiler und ein raffiniertes Speichermanagement kann diesen jedoch in Grenzen halten. Listing 3.4 zeigt die Implementierung als Methode der Klasse *BTree*.

```
1 class BTree(object):
2     ...
3     def deleteND(self,key):
4         if self.key==key:
5             if self.ltree==self.rtree==None: return None  # 0 Kinder
6             elif self.ltree==None: return self.rtree      # 1 Kind
7             elif self.rtree==None: return self.ltree
```

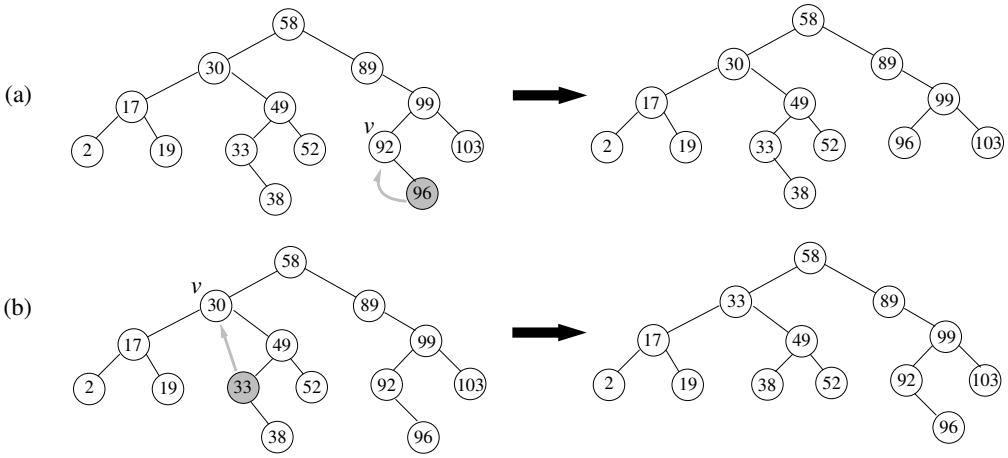


Abb. 3.5: *Löschen eines Knotens in einem binären Suchbaum. Abbildung (a) zeigt das Löschen eines Knotens $v = 92$, der nur ein Kind besitzt. Hier wird einfach das Kind von v (nämlich der Knoten mit dem Schlüssel 96) an dessen Stelle gesetzt. Abbildung (b) zeigt das Löschen des Knotens $v = 30$, der zwei Nachfolger besitzt. Hier wird der minimale Knoten des rechten Teilbaums von v – das ist in diesem Fall der Knoten mit dem Schlüssel 33 – an die Stelle von v gesetzt. Man sieht, dass der minimale Knoten selbst noch ein Kind hat; dieser wird, wie in Fall (a) beschrieben, an dessen Stelle gesetzt.*

```
8         else:                                          # 2 Kinder
9           z=self.rtree.minEl()
10          return BTree(z.key,self.ltree, self.rtree.deleteND(z.key), z.val)
11      else:
12        if key<self.key:
13          return BTree(self.key, self.ltree.deleteND(key), self.rtree, self.val)
14        elif key>self.key:
15          return BTree(self.key, self.ltree, self.rtree.deleteND(key), self.val)
```

Listing 3.4: *Implementierung der Lösch-Operation im Binärbaum durch die Klassenfunktion BTree.deleteND(key).*

Entspricht der Schlüssel *self.key* des aktuellen Knotens nicht dem zu löschenden Schlüssel *key*, so wird weiter nach dem zu löschenden Knoten gesucht – entweder im linken Teilbaum (Zeile 13) oder im rechten Teilbaum (Zeile 15). Falls jedoch der Schlüssel des aktuellen Knotens dem zu löschenden Schlüssel entspricht, so wird dieser Knoten gelöscht (Zeile 4–10). Ist der Knoten ein Blatt, so wird er einfach gelöscht (Zeile 5). Besitzt er ein Kind, so wird dieses Kind, also *self.ltree* bzw. *self.rtree*, an dessen Stelle gesetzt (Zeile 6 und 7). In Zeile 9 und 10 befindet sich der Code, um einen Knoten mit zwei Kindern zu löschen: Das minimale Element des rechten Teilbaums (hier: *self.rtree.minEl*(); siehe Aufgabe 3.6) wird an die Stelle des aktuellen Knotens gesetzt. Zusätzlich wird dieser minimale Knoten durch einen rekursiven Aufruf (*self.rtree.deleteND*(*z.key*)) von seiner ursprünglichen Position gelöscht.

Aufgabe 3.10

Man kann ein destruktives Löschen unter Anderem unter Verwendung einer „Rückwärtsverzeigerung“ implementieren, d. h. unter Verwendung einer Möglichkeit, den Vaterknoten eines Knotens v anzusprechen.

Implementieren Sie diese Möglichkeit, indem Sie die Klasse *BTree* um ein Attribut *parent* erweitern. Man beachte, dass dies weitere Änderungen nach sich zieht: Die Methode *insert* muss etwa angepasst werden.

Aufgabe 3.11

Implementieren Sie eine Methode $BTree.delete(v)$, die auf destruktive Art und Weise einen Knoten mit Schlüsselwert v aus einem binären Suchbaum löscht.

Aufgabe 3.12

Implementieren Sie eine Methode $insertND(v)$ der Klasse *BinTree*, die nicht-destruktiv einen Knoten in einen binären Suchbaum einfügt; ein Aufruf $bt.insertND(v)$ sollte bt nicht verändern, sondern einen neuen binären Suchbaum zurückliefern, der bt mit eingefügtem v entspricht.

3.1.3 Laufzeit

Die Suche braucht $O(h)$ Schritte, wobei h die Höhe[1] des binären Suchbaums ist, denn es wird mindestens ein Vergleich für jede Stufe des Baumes benötigt. Gleiches gilt für das Finden des maximalen bzw. minimalen Elements.

Was ist die Höhe eines binären Suchbaums? Das lässt sich nicht pauschal beantworten, denn die Höhe hängt von der Reihenfolge ab, in der Schlüssel in einen Baum eingefügt werden. Man kann zeigen, dass bei einer zufällig gewählten Einfügereihenfolge von n Zahlen im Durchschnitt ein binärer Suchbaum mit einer Höhe von $c \cdot \log_2 n$ entsteht, d. h. im Durchschnitt ist die Höhe eines binären Suchbaums, dessen Einfüge- und Löschoperationen wie oben beschrieben implementiert sind, in $O(\log n)$.
Bei einer ungünstigen Einfügereihenfolge ist es aber möglich, dass ein binärer Suchbaum der Höhe n entsteht, mit einer Strukur wie etwa in Abbildung 3.6 gezeigt.

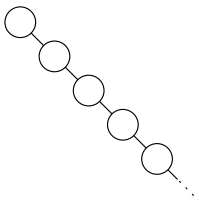


Abb. 3.6: *Ein „entarteter“ (extrem unbalancierter) binärer Suchbaum, wie er durch ungeschicktes Einfügen entstehen kann.*

[1] Die Höhe eines Baumes ist die Anzahl von Kanten von der Wurzel bis zu dem „tiefsten“ Blatt; siehe Anhang B.4.1 für mehr Details.

Aufgabe 3.13

Gegeben seien die Schlüssel $51, 86, 19, 57, 5, 93, 8, 9, 29, 77$.

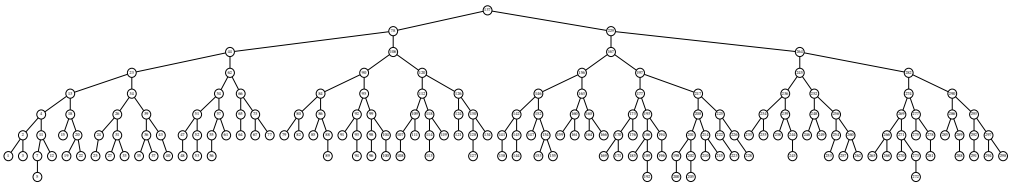
(a) Welche Höhe hat der Baum, wenn die Schlüssel in der oben angegebenen Reihenfolge in einen anfänglich leeren Baum eingefügt werden?

(b) Finden Sie eine Einfügereihenfolge, bei der ein Baum der Höhe 9 entsteht.

(c) Finden Sie eine Einfügereihenfolge, bei der ein Baum minimaler Höhe entsteht.

In den folgenden beiden Abschnitten werden Techniken vorgestellt, wie man binäre Suchbäume möglichst balanciert halten kann.

3.2 AVL-Bäume



AVL-Bäume sind balancierte binäre Suchbäume. Sie sind benannt nach den Erfindern, Georgi Adelson-Velski und Jewgeni Landis, zwei russischen Mathematikern und Informatikern, die 1962 erstmals beschrieben, wie binäre Suchbäume mittels sog. „Rotationen“ balanciert gehalten werden können.

Ein AVL-Baum ist ein binärer Suchbaum, für den gilt, dass sich die Höhe des linken Teilbaums und die Höhe des rechten Teilbaums eines jeden Knotens um höchstens einen Betrag von 1 unterscheiden darf. Wir gehen hier von der im letzten Abschnitt beschriebenen Implementierung eines binären Suchbaums aus und definieren zusätzlich für jeden Knoten v ein Attribut $v.\mathit{height}$, das die Höhe des Knotens speichert, und ein Attribut $v.\mathit{balance}$, das den Balance-Wert des Knotens speichert.

Seien $\mathit{lheight}$ die Höhe des linken Teilbaums und $\mathit{rheight}$ die Höhe des rechten Teilbaums eines Knoten v, dann sind die beiden Attribute $v.\mathit{height}$ und $v.\mathit{balance}$ wie folgt definiert:

$$v.\mathit{height} = 1 + \mathit{max}(\mathit{rheight}, \mathit{lheight}) \tag{3.1}$$

$$v.\mathit{balance} = -\mathit{lheight} + \mathit{rheight} \tag{3.2}$$

Die Tatsache, dass ein AVL-Baum balanciert ist, bedeutet, dass für jeden Knoten v eines AVL-Baums

$$v.\mathit{balance} \in \{-1, 0, 1\}$$

gelten muss.

Listing 3.5 zeigt die Implementierung der *__init__*-Methode der Klasse *AVLTree*, die von der im letzten Abschnitt vorgestellten Klasse *BTree* erbt. Diese *__init__*-Funktion führt dieselben Kommandos aus, wie die *__init__*-Funktion der Elternklasse *BTree* – dies wird durch den entsprechenden Aufruf in Zeile 4 sichergestellt. Zusätzlich werden die Höhen- und Balance-Werte des Knotens berechnet – dies geschieht durch den Aufruf der Funktion *_calcHeight* in Zeile 5.

```
1  class AVLTree(BTree):
2
3    def __init__ ( self , key, ltree=None, rtree=None, val=None):
4      BTree. __init__ ( self , key, ltree , rtree , val)
5      self . _calcHeight()
6
7    def _calcHeight( self ):
8      rheight = -1 if not self. rtree else self . rtree . height
9      lheight = -1 if not self. ltree else self . ltree . height
10     self . height = 1 + max(rheight,lheight)
11     self . balance = -lheight + rheight
```

Listing 3.5: *Implementierung der Klasse AVLTree, die von BTree – der Klasse, die unbalancierte binäre Suchbäume implementiert, – erbt.*

Die Funktion *_calcHeight* berechnet die Höhe und den Balance-Wert gemäß der in den Gleichungen (3.1) und (3.2) dargestellten Beziehungen. Das '_'-Zeichen, mit dem der Methodenname beginnt, deutet an, dass es sich hier um eine interne Methode handelt, die zwar von anderen Methoden verwendet wird, jedoch üblicherweise nicht von einem Benutzer der Klasse.

3.2.1 Einfügeoperation

Sowohl beim Einfügen als auch beim Löschen kann die Balance eines Knoten bzw. mehrerer Knoten auf dem Pfad von der Einfüge- bzw. Löschposition bis zurück zur Wurzel zerstört sein. Abbildung 3.7 veranschaulicht, welche Knoten re-balanciert werden müssen.

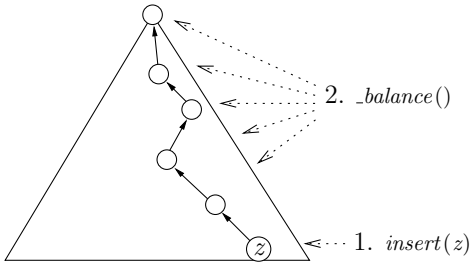


Abb. 3.7: *Nach einer Einfügeoperation müssen die Knoten auf dem Pfad von der Einfügeposition bis hin zur Wurzel rebalanciert werden.*

Wir gehen von einer – wie im letzten Abschnitt in Listing 3.3 beschriebenen – *insert*-Funktion aus. Stellen wir sicher, dass vor jedem Verlassen der *insert*-Funktion die Funktion *_balance*() aufgerufen wird, so erfolgt die Balancierung während des rekursiven Aufstiegs; dies entspricht genau der Rebalancierung der Knoten von der Einfügeposition bis hin zur Wurzel wie in Abbildung 3.7 gezeigt.

Das folgende Listing 3.6 zeigt die Implementierung:

```
1  def insert( self ,x, val=None):
2    if x < self.key:
3      if self. ltree == None:
4        self. ltree = AVLTree(x,None,None,val)
5      else:
6        self. ltree . insert(x, val)
7    elif x > self.key:
8      if self. rtree == None:
9        self. rtree = AVLTree(x,None,None,val)
10     else:
11       self. rtree . insert(x, val)
12   self. _calcHeight()
13   self. _balance()
```

Listing 3.6: *Implementierung der Einfügeoperation bei AVL-Bäumen.*

Beim rekursiven Aufstieg wird zunächst Höhe und Balance-Wert neu berechnet (Zeile 12) und dann (falls notwendig) rebalanciert (Zeile 13).

Aufgabe 3.14

Implementieren Sie nach ähnlichem Prinzip eine balancierende Löschfunktion

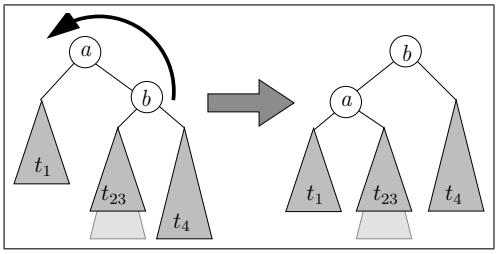
3.2.2 Grundlegende Balancierungsoperationen: Rotationen

Die Balancierungsoperationen werden *Rotationen*[2] genannt. Man unterscheidet zwischen Einfachrotationen und Doppelrotatonen, die prinzipiell die Hintereinanderausführung zweier Einfachrotationen darstellen.

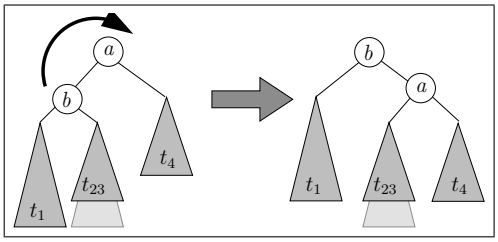
Man beachte, dass ein Knoten a immer genau dann rebalanciert wird, wenn sich die Höhe des rechten und die Höhe des linken Teilbaums um einen Betrag von genau 2 unterscheiden, d. h. wenn $a.balance \in \{-2, 2\}$. Der Grund dafür, dass der Betrag des Balance-Werts immer genau 2 beträgt, ist, dass wir sicherstellen, dass immer sofort nach dem Einfügen *eines* Knotens bzw. dem Löschen *eines* Knotens rebalanciert wird.

[2]Das Wort „Rotation“ wird in diesem Zusammenhang wohl eher deshalb verwendet, weil die Verwendung dieses Begriffs in der wissenschaftlichen Literatur zur Gewohnheit wurde und weniger weil es offensichtliche Analogien zu der Drehbewegung einer Rotation gibt.

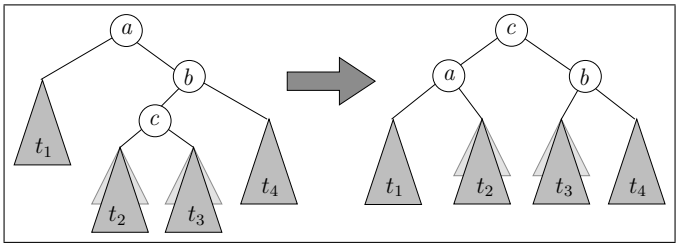
Abbildung 3.8 zeigt die vier verschiedenen Arten von Rotationen: Zwei symmetrische Fälle der Einfachrotationen in Abbildungen 3.8(a) (für den Fall $a.balance = 2$) und 3.8(b) (für den Fall $a.balance = -2$) und die zwei symmetrischen Fälle der Doppelrotationen in Abbildungen 3.8(c) (für und Fall $a.balance = 2$) und 3.8(d) (für den Fall $a.balance = -2$).



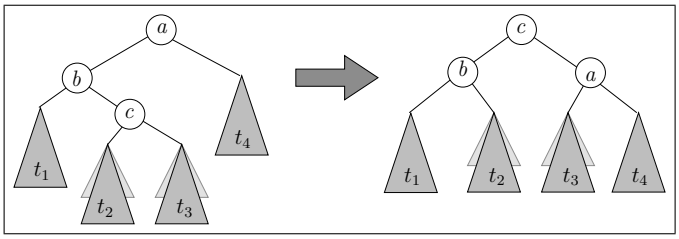
(a) Einfache Links-Rotation: $a.balance = 2$ und innerer Enkel nicht höher.



(b) Einfache Rechts-Rotation: $a.balance = -2$ und innerer Enkel nicht höher.



(c) Doppel-Links-Rotation: $a.balance = 2$ und innerer Enkel höher.



(d) Doppel-Rechts-Rotation: $a.balance = -2$ und innerer Enkel höher.

Abb. 3.8: *Die vier verschiedenen Rotationen: Zwei Einfach- und zwei Doppelrotationen.*

Die Einfachrotationen (Abbildungen 3.8(a) und 3.8(b)) können immer dann angewendet werden, wenn der innere, im höheren Teilbaum befindliche, Enkel von a nicht höher ist als der äußere Enkel. Doppelrotationen (Abbildungen 3.8(c) und 3.8(d)), die im Prinzip eine Hinteranderausführung von zwei Einfachrotationen darstellen, müssen entsprechend bei einer Rebalancierung angewendet werden, wenn der innere im höheren Teilbaum befindliche Enkel von a höher ist als der äußere Enkel. Die eben beschriebenen Kriterien, wann welche Rotation anzuwenden ist, sind in der in Listing 3.7 gezeigten Methode *_balance*() implementiert.

```
1  def _balance( self ):
2      if  self.balance == 2: # rechts höher
3          t23 = self.rtree.ltree  ;  t4  = self.rtree.rtree
4          if not t23:  self._simpleLeft()
5          elif  t4 and t23.height ≤ t4.height:  self._simpleLeft()
6          else:  self._doubleLeft()
7      if  self.balance == -2: # links höher
8          t23 = self.ltree.rtree  ;  t1  = self.ltree.ltree
9          if not t23:  self._simpleRight()
10         elif  t1 and t23.height ≤ t1.height:  self._simpleRight()
11         else:  self._doubleRight()
```

Listing 3.7: *Die Methode _balance() entscheidet, ob überhaupt balanciert werden muss und wenn ja, welche der vier Rotationen angewendet werden soll.*

Wir beschreiben im Folgenden exemplarisch zwei der vier verschiedenen Rotationen im Detail:

Einfache Linksrotation (Abbildung 3.8(a)): Hier ist der innere im höheren Teilbaum befindliche Enkel t_{23} von a nicht höher als der äußere Enkel t_4. Der schwach gezeichnete Teil der Abbildung deutet an, dass der innere Enkel auch gleich hoch sein kann als der äußere Enkel. Die Rotation „hebt“ nun as rechtes Kind b samt dessen rechten Teilbaum t_4 um eine Ebene nach oben, indem b zur neuen Wurzel gemacht wird. Entscheidend ist hier, dass t_4 – der Teilbaum, durch den der Höhenunterscheid von 2 entsteht – nach der Rotation eine Ebene höher aufgehängt ist als vor der Rotation. Der Knoten a wird zum linken Kind von b (da $a<b$ bleibt die Eigenschaft eines Suchbaums erhalten) und a behält seinen linken Teilbaum t_1; dadurch sinkt das Höhenniveau von t_1 durch die Rotation. Das ist jedoch unkritisch, da die Höhe von t_1 um 2 geringer war als die Höhe von t_2. Der Teilbaum t_{23} wird zum rechten Teilbaum von a. Da alle Schlüsselwerte in t_{23} kleiner als $a.key$ und größer als $b.key$ sind, bleibt auch hier die Eigenschaft des binären Suchbaums erhalten. Folgendes Listing zeigt eine entsprechende Implementierung in Form einer Methode *_simpleLeft()* der Klasse *AVLTree*:

```
1  def _simpleLeft( self ):
2      a = self  ;  b = self.rtree
3      t1   = a.ltree
4      t23  = b.ltree
5      t4   = b.rtree
6      newL = AVLTree(a.key, t1, t23, a.val)
7      self.key = b.key ;  self.ltree  = newL ; self.rtree  = t4 ;  self.val = b.val
```

Doppelte Linksrotation (Abbildung 3.8(c)): Hier ist der innere im höheren Teilbaum befindliche Enkel (der seinerseits aus t_2 und t_3 besteht) von a höher als der äußere Enkel t_4. Der schwach gezeichnete Teil der Abbildung deutet an, dass einer der

beiden Teilbäume des Enkels auch um eins niedriger sein kann als der andere Teilbaum. Hier wird zunächst eine Rechtsrotation des Teilbaums mit Wurzel *b* ausgeführt; dies bringt zwar noch nicht den gewünschten Höhenausgleich, jedoch wird so die Voraussetzung für die Ausführung einer Einfachrotation hergestellt: der innere Enkel ist nicht mehr höher als der äußere Enkel. Eine anschließende Linksrotation führt dann zum Erfolg. Folgendes Listing zeigt eine entsprechende Implementierung in Form einer Methode *_doubleLeft*() der Klasse *AVLTree*:

```
1  def _doubleLeft( self ):
2      a = self ; b = self.rtree ; c = self.rtree.ltree
3      t1 = a.ltree
4      t2 = c.ltree
5      t3 = c.rtree
6      t4 = b.rtree
7      newL = AVLTree(a.key, t1, t2, a.val)
8      newR = AVLTree(b.key, t3, t4, b.val)
9      self.key = c.key ; self.ltree = newL ; self.rtree = newR ; self.val = c.val
```

Aufgabe 3.15

Implementieren Sie ...

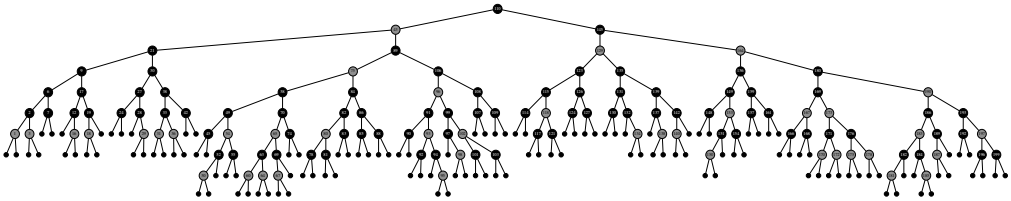
(a) ... eine Methode *_simpleRight* der Klasse *AVLTree*, die eine einfache Rechtsrotation realisiert.

(b) ... eine Methode *_doubleRight* der Klasse *AVLTree*, die eine Doppel-Rechts-Rotation realisiert.

Aufgabe 3.16

Um wie viel kann sich die Länge des längsten Pfades mit der Länge des kürzesten Pfades (von der Wurzel zu einem Blatt) eines AVL-Baums höchstens unterscheiden?

3.3 Rot-Schwarz-Bäume



Ein Rot-Schwarz-Baum, ist ein balancierter binärer Suchbaum; jeder Knoten in einem Rot-Schwarz-Baum enthält eine zusätzliche Information, die angibt, ob der Knoten rot oder schwarz ist. Rot-Schwarz-Bäume generieren im Vergleich zu AVL-Bäumen einen etwas geringeren Balancierungsaufwand, neigen auf der anderen Seite jedoch dazu, etwas größere Pfadlängendifferenzen aufzuweisen als AVL-Bäume.

Man kann einen Rot-Schwarz-Baum ganz ähnlich implementieren, wie einen gewöhnlichen binären Suchbaum, nur dass zusätzlich ein Attribut *self.c* mitgeführt wird, das die Farbe des jeweiligen Knotens speichert.

```
1 RED,BLACK = 0,1
2 class RBTree(object):
3   def __init__(self, color, key, ltree=None, rtree=None, val=None):
4     self.l = ltree
5     self.r = rtree
6     self.val = val
7     self.c = color
8     self.key = key
```

Der Übersichtlichkeit halber verzichten wir darauf, die Klasse *RBTree* von *BTree* erben zu lassen. Die Gemeinsamkeiten dieser beiden Klassen sind ohnehin etwas geringer als die Gemeinsamkeit zwischen *AVLTree* und *BTree*.

Für jeden Knoten eines Rot-Schwarz-Baumes müssen die folgenden beiden Invarianten gelten:

1. Invariante: Kein roter Knoten hat einen roten Elternknoten.
2. Invariante: Jeder Pfad von der Wurzel zu einem Blatt enthält die gleiche Anzahl schwarzer Knoten.

Diese Invarianten müssen ggf. nach einer Einfüge- oder Löschoperation wiederhergestellt werden.

Diese beiden Invarianten garantieren, dass sich die Höhen der beiden Teilbäume eines Knotens nicht zu stark unterscheiden können. Deshalb rechnet man Rot-Schwarz-Bäume auch der Klasse der balancierten Bäume zu. Zwei verschiedene Pfade von der Wurzel zu einem Blatt können sich um höchsten den Faktor „Zwei“ unterscheiden, da beide die gleiche Anzahl schwarzer Knoten enthalten müssen und zwischen je zwei schwarzen

Knoten auf diesem Pfad sich höchstens ein roter Knoten befinden kann. Die Höhe eines Rot-Schwarz Baumes ist daher auch im schlechtesten Fall $O(\log n)$; insofern kann man Rot-Schwarz-Bäume als balancierte bezeichnen.

Abbildung 3.9 zeigt ein Beispiel eines Rot-Schwarz-Baums.

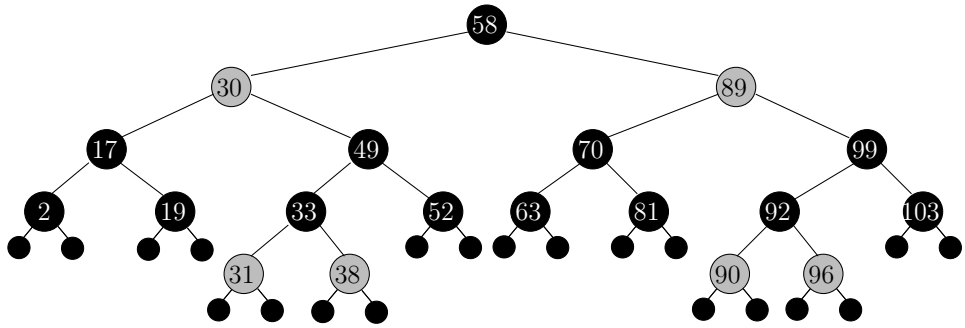


Abb. 3.9: *Beispiel eines Rot-Schwarz-Baumes; man sieht, dass es sich zunächst um einen binären Suchbaum handelt, man sieht, dass kein roter Knoten einen roten Elternknoten besitzt, und dass jeder Pfad von der Wurzel zu einem Blatt die gleiche Anzahl schwarzer Knoten enthält – in diese Falle sind dies drei schwarze Knoten (bzw. vier schwarze Knoten, wenn wir uns die leeren Knoten schwarzgefärbt denken). Außerdem ist angedeutet, dass wir uns die leeren Knoten alle als schwarze Knoten denken; folglich sind für die Blattknoten prinzipiell beide Farben möglich.*

3.3.1 Einfügen

Da Rot-Schwarz-Bäume binäre Suchbäume sind, ist die Suchfunktion bei Rot-Schwarz-Bäumen genau gleich wie die Suchfunktion bei binären Suchbäumen. Bei der Realisierung der Einfügeoperation muss jedoch darauf geachtet werden, dass durch das Einfügen eines neuen Knotens die beiden Invarianten nicht verletzt werden. Wir gehen beim Einfügen eines neuen Knotens v so vor, dass wir zunächst v als neuen *roten* Knoten so in den Rot-Schwarz-Baum einfügen, wie wir dies auch bei herkömmlichen binären Suchbäumen getan haben. Dadurch ist zwar Invariante 2 erhalten (da wir keinen neuen schwarzen Knoten einfügen, bleibt die Anzahl der schwarzen Knoten auf jedem Pfad unverändert), Invariante 1 könnte dadurch jedoch verletzt werden. Abbildung 3.10 zeigt als Beispiel die Situation, nachdem der Schlüsselwert „42“ in den Rot-Schwarz-Baum aus Abbildung 3.9 eingefügt wurde – als Folge wird dabei tatsächlich Invariante 1 verletzt.

Folgendermaßen eliminieren wir nach solch einer Einfügeoperation mögliche Verletzungen der Invariante 1: Wir laufen vom eingefügten Blatt bis hin zur Wurzel durch den Rot-Schwarz-Baum und eliminieren in $O(\log n)$ Schritten sukzessive alle Verletzungen der Invariante 1 auf diesem Pfad. Hierbei muss tatsächlich der ganze Pfad (der Länge $O(\log n)$) überprüft werden, da die Eliminierung einer Verletzung auf Höhe i eine weitere Verletzung auf Höhe $i-1$ nach sich ziehen kann.

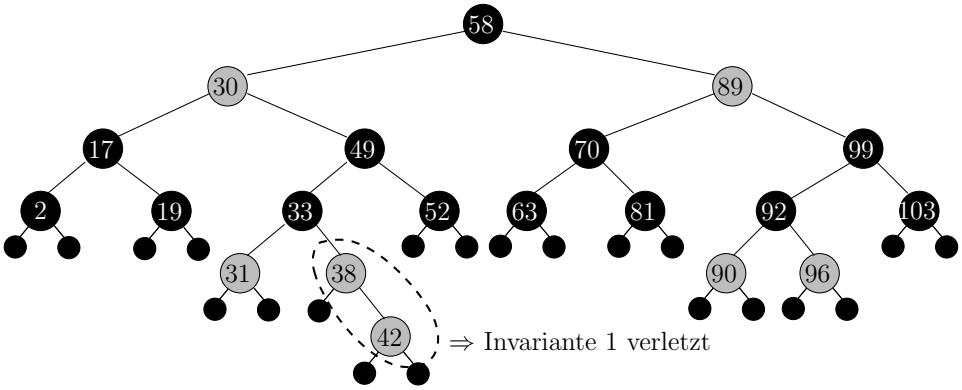


Abb. 3.10: *Situation, nachdem ein neuer (roter) Knoten mit Schlüssel $k = 42$ in den Rot-Schwarz-Baum aus Abbildung 3.9 wie in einen herkömmlichen binären Suchbaum eingefügt wurde. Wie man sieht, wird dadurch Invariante 1 verletzt.*

Abbildung 3.11 zeigt alle vier möglichen Konstellationen, die die Invariante 1 verletzen und die eine Rebalancierung erfordern.

In Abbildung 3.10 liegt an der „Stelle“, die zur Verletzung der Invariante 1 führt, die vierte Konstellation vor. Abbildung 3.12 zeigt den Rot-Schwarz-Baum nach Wiederherstellen der Invariante 1, die durch Abbildung der vierten Konstellation auf die einheitliche Form entsteht.

Implementierung. Listing 3.8 zeigt eine mögliche Implementierung der Einfüge-Operation *RBTree.insert*.

```
1  class RBTree(object):
2    ...
3    def insert(self,x,val=None):
4      self._insert(x,val)
5      self.c = BLACK
6
7    def _insert(self,x,val=None):
8      if x < self.key:
9        if not self.l:
10         self.l= RBTree(RED,x)
11       else:
12         self.l._insert(x)
13     elif x > self.key:
14       if not self.r:
15         self.r = RBTree(RED,x)
16       else:
17         self.r._insert(x)
18     self._balance()
```

Listing 3.8: *Implementierung der Einfüge-Operation in einen Rot-Schwarz-Baum*

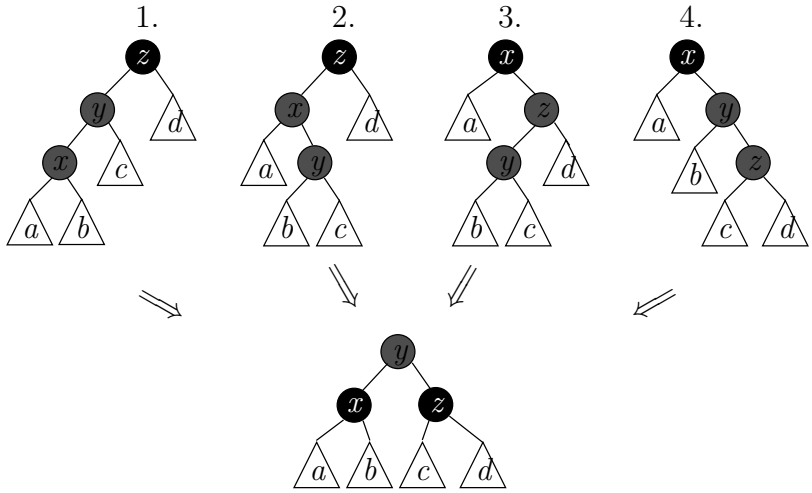


Abb. 3.11: *Alle vier Situationen, in denen beim rekursiven Aufstieg rebalanciert werden muss. Jede dieser vier Konstellationen kann durch Abbildung auf eine einheitliche – im Bild unten dargestellte – Form „repariert“ werden.*

Die Wurzel des Baumes wird nach Ausführung der Einfügeoperation stets schwarz gefärbt (Zeile 5). Die eigentliche rekursiv implementierte Einfügeoperation befindet sich in der Funktion *_insert*. Zunächst wird in Zeile 8 bzw, Zeile 13 überprüft, ob der einzufügende Schlüsselwert x in den linken Teilbaum (falls $x < self.key$) oder in den rechten Teilbaum (falls $x > self.key$) einzufügen ist. Ist der linke bzw. rechte Teilbaum leer (d. h. gilt **not** $self.l$ bzw. **not** $self.r$), so wird ein neuer ein-elementiger Rot-Schwarzbaum mit rotem Knoten erzeugt und als linkes bzw. rechtes Kind eingefügt – dies geschieht in Zeile 10 bzw. Zeile 15. Ist der linke bzw. rechte Teilbaum nicht leer, so wird *_insert* rekursiv aufgefrufen. Ganz am Ende der Einfügeprozedur – und damit beim rekursiven Aufstieg – wird die Funktion *_balance* aufgerufen, die bei Bedarf die Invarianten wiederherstellt und damit rebalanciert.

Listing 3.9 zeigt die Implementierung der *_balance*-Funktion, die die Invarianten wiederherstellt.

In den Zeilen 6, 9, 12 und 15 wird jeweils getestet, ob eine der in Abbildung 3.11 graphisch dargestellten vier Situationen zutrifft. Wir wählen für die weiteren Erklärungen als Beispiel den für die Situation 1 zuständigen Code aus; die drei anderen Fälle können analog erklärt werden. Situation 1 liegt genau dann vor, wenn ...

1. ... der linke Teilbaum von s und wiederum dessen linker Teilbaum nicht leer sind, d. h. wenn „**not** $s.l$“ und „**not** $s.l.l$“ gelten[3].
2. ... und wenn $s.l$ und $s.l.l$ rot gefärbt sind, wenn also gilt, dass „$s.l.c == s.l.l.c == RED$“.

[3] Pythons Wert „*None*“ entspricht in booleschen Formeln dem logischen Wert „Falsch“; daher kann mittels „**if not** $s.l$ ... “ überprüft werden, ob $s.l$ auf einen Rot-Schwarz-Baum zeigt, oder stattdessen einen *None*-Wert enthält.

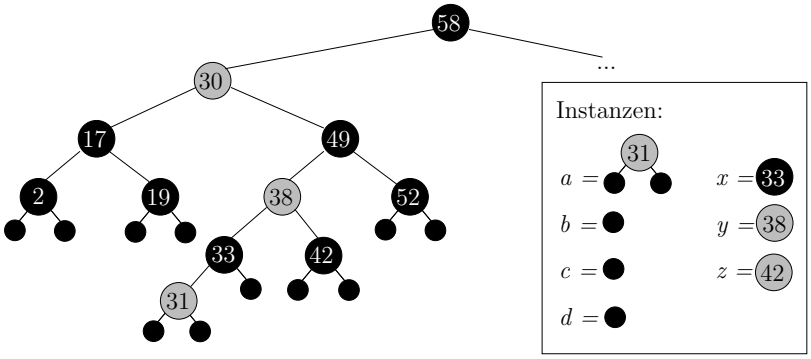


Abb. 3.12: *Der Rot-Schwarz-Baum nach Einfügen des Schlüsselwertes 42 und nach Wiederherstellen der Invariante 1. In diesem Falle ist hierfür nur eine einzige Rebalancierung notwendig. Rechts im Bild sind für den einen durchgeführten Rebalancierungs-Schritt – dieser entspricht Situation 4 – die notwendigen Instanzen für die in Abbildung 3.11 verwendeten Platzhalter angegeben, also für die Teilbäume a, b, c, d und für die Knoten x, y, z aus Abbildung 3.11.*

In Abbildung 3.13 ist nochmals die Situation 1 zusammen mit den darauf zu mappenden Zweigen des Baumes s dargestellt.

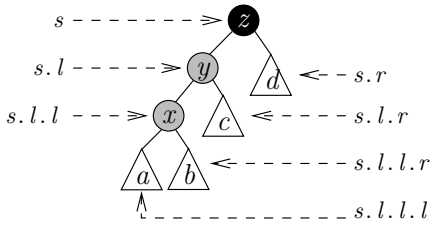


Abb. 3.13: *Die erste der vier möglichen Situationen, in denen rebalanciert werden muss. Die Abbildung stellt die Zuordnung der Variablen x, y, z und a, b, c, d auf die entsprechenden Knoten bzw. Teilbäume des Baumes s dar, die mittels Indizierung angesprochen werden können – natürlich aber nur, wenn die Methode __getitem__ entsprechend definiert wurde.*

Liegt Situation 1 vor, so wird also der Variablen x die in $s.l.l$ gespeicherten Werte, der Variablen y die in $s.l$ gespeicherten Werte und der Variablen z die in s gespeicherten Werte zugewiesen. Während die Variablen x, y und z Knoten-Werte (d. h. das *key*-Attribut als erste Komponente zusammen mit dem *val*-Attribut eines Knotens als zweite Komponente) enthalten, sollten den Variablen a, b, c und d ganze Teilbäume zugewiesen werden – dies ist auch aus der Darstellung in Abbildung 3.11 ersichtlich. Variable a erhält in Situation 1 den Wert $s.l.l.l$, Variable b erhält den Wert $s.l.l.r$, Variable c erhält den Wert $s.l.r$ und Variable d erhält den Wert $s.r$. Schließlich wird in den Zeilen 20 und 21 in Listing 3.9 gemäß den in Abbildung 3.11 gezeigten Regeln der neue linke und der neue rechte Teilbaum erzeugt. Schließlich wird in Zeile 22 die rebalancierte Variante des Rot-Schwarz-Baumes generiert.

```
class RBTree(object):
  ...
  def _balance( self ):
    s = self
    if s.c==RED: return s
    if s.l and s.l.l and s.l.c == s.l.l.c == RED: # Fall 1:
      y = (s.l.key, s.l.val) ; x = (s.l.l.key, s.l.l.val) ; z = (s.key, s.val)
      a = s.l.l.l ; b = s.l.l.r ; c = s.l.r ; d = s.r
    elif s.l and s.l.r and s.l.c == s.l.r.c == RED: # Fall 2:
      x = (s.l.key, s.l.val) ; y = (s.l.r.key, s.l.r.val) ; z = (s.key,s.val)
      a = s.l.l ; b = s.l.r.l ; c = s.l.r.r ; d = s.r
    elif s.r and s.r.l and s.r.c == s.r.l.c == RED: # Fall 3:
      x = (s.key, s.val) ; y = (s.r.l.key, s.r.l.val) ; z = (s.r.key, s.r.val)
      a = s.l ; b = s.r.l.l ; c = s.r.l.r ; d = s.r.r
    elif s.r and s.r.r and s.r.c == s.r.r.c == RED: # Fall 4:
      x = (s.key, s.val) ; y = (s.r.key, s.r.val) ; z = (s.r.r.key, s.r.r.val)
      a = s.l ; b = s.r.l ; c = s.r.r.l ; d = s.r.r.r
    else:
      return s
    newL = RBTree(BLACK, x[0], a, b, x[1])
    newR = RBTree(BLACK, z[0], c, d, z[1])
    self.c = RED ; self.key = y[0] ; self.l = newL ; self.r = newR ; self.val = y[1]
```

Listing 3.9: *Implementierung der Rebalancierung, d. h. Eliminierung von Verletzungen der Invariante 1, die beim Einfügen eines neuen roten Blattes in einen Rot-Schwarz-Baum entstehen können.*

Aufgabe 3.17

(a) Wie hoch wäre ein (fast) vollständiger binärer Suchbaum, der 300000 Elemente enthält?

(b) Wie hoch könnte ein Rot-Schwarz-Baum maximal sein, der 300000 Elemente enthält?

Aufgabe 3.18

Schreiben Sie eine Methode *RBTree.inv1Verletzt*, die testet, ob es einen Knoten im Rot-Schwarz-Baum gibt, für den die Invariante 1 verletzt ist, d. h. ob es einen roten Knoten gibt, dessen Vorgänger ebenfalls ein roter Knoten ist. Ein Aufruf von *inv1Verletzt* soll genau dann *True* zurückliefern, wenn die Invariante 1 für mindestens einen Knoten verletzt ist.

Aufgabe 3.19

Schreiben Sie eine Methode, die überprüft, ob die Invariante 2 verletzt ist.

(a) Schreiben Sie hierfür zunächst eine Methode *RBTree.anzSchwarzKnoten*, die ein Tupel (x, y) zurückliefern soll, wobei in x die minimale Anzahl schwarzer Knoten auf einem Pfad von der Wurzel zu einem Blatt und in y die maximale Anzahl schwarzer Knoten auf einem Pfad von der Wurzel zu einem Blatt zurückgegeben werden soll.

(b) Schreiben Sie nun eine Methode *RBTree.inv2Verletzt*, die genau dann *True* zurückliefern soll, wenn die Invariante 2 für den entsprechenden Rot-Schwarz-Baum verletzt ist.

Aufgabe 3.20

Vergleichen Sie die Performance des Python-internen *dict*-Typs mit der vorgestellten Implementierung von rot-schwarz Bäumen folgendermaßen:

(a) Fügen sie 1 Mio zufällige Zahlen aus der Menge $\{1, \ldots 10Mio\}$ jeweils in einen Python-*dict* und in einen Rot-Schwarz-Baum ein, messen sie mittels *time*() jeweils die verbrauchte Zeit und vergleichen sie.

(b) Führen sie nun 1 Mio Suchdurchgänge auf die in der vorigen Teilaufgabe erstellten Werte aus, und messen sie wiederum mittels *timeit* die verbrauchte Zeit und vergleichen sie.

3.3.2 Löschen

Das Löschen eines Knoten v in einem Rot-Schwarz-Baum besteht grundsätzlich aus drei Schritten:

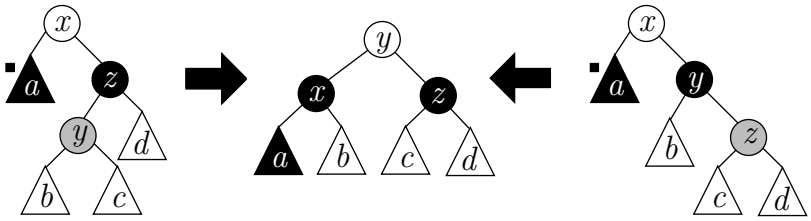
Schritt 1: Ein Knoten v in einem Rot-Schwarz-Baum wird zunächst gelöscht als wäre es ein Knoten in einem herkömmlichen binären Suchbaum: Besitzt der zu löschende Knoten zwei (nicht-leere) Kinder als Nachfolger, so ersetzt man das Schlüssel-Wert-Paar von v durch das Schlüssel-Wert-Paar des minimalen Knotens m des rechten Teilbaums und löscht anschließend m – dies entspricht der Darstellung von Fall (b) in Abbildung 3.5 auf Seite 55. Da m mindestens einen Blattknoten besitzt, kann man so das Problem auf das Löschen eines Knotens mit mindestens einem Blattknoten reduzieren.

Ist m ein schwarzer Knoten, so wird durch Löschen von m die Invariante 2 verletzt, die vorschreibt, dass jeder Wurzel-Blatt-Pfad in einem Rot-Schwarz-Baum die gleiche Anzahl schwarzer Knoten besitzen muss. Dies wird vorübergehend dadurch „ausgeglichen“, indem das eine schwarze Blatt von m einen doppelten Schwarz-Wert zugewiesen bekommt.

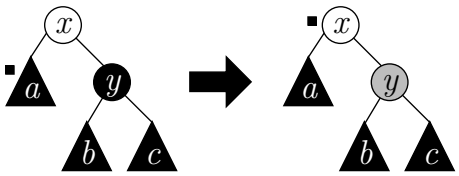
Schritt 2: Nun propagiert man doppelte Schwarz-Werte den Baum soweit durch Anwendung bestimmter Regeln (die unten aufgeführten drei Fälle) nach oben, bis diese

aufgelöst werden können. In der graphischen Darstellung dieser Regeln markieren wir Doppelschwarze Knoten hierbei durch eine zusätzliche Schwarz-Markierung (■). Ein roter Knoten mit einer Schwarz-Markierung kann durch schwarz-färben des roten Knotens aufgelöst werden. Man beachte dass in den im Folgenden aufgeführten drei Fällen der doppelschwarze Knoten immer das Linke Kind ist. Die Fälle, in denen der doppelschwarze Knoten das rechte Kind ist, sind symmetrisch, und nicht getrennt aufgeführt.

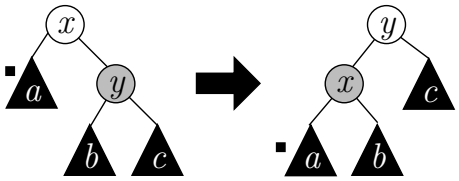
(a) Der Geschwisterknoten eines doppelschwarzen Knotens ist schwarz und es gibt einen roten Neffen. Dies ist der „günstigste“ Fall; die Schwarz-Markierung kann aufgelöst werden.



(b) Der Geschwisterknoten eines doppelschwarzen Knotens ist schwarz und beide Neffen sind schwarz. Durch folgende Rotation kann die Schwarz-Markierung nach oben weitergereicht werden.

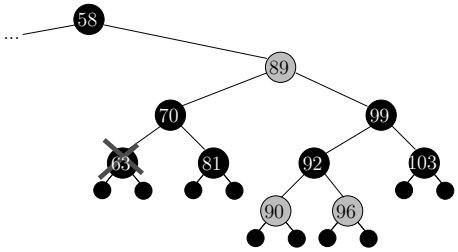


(c) Der Geschwisterknoten eines doppelschwarzen Knotens ist rot. Dies erfordert zunächst eine Rotation und verweist anschließend auf entweder Fall (a) oder Fall (b).

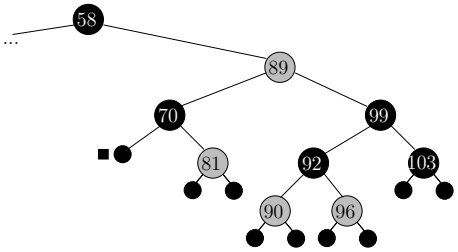


Schritt 3: Befindet sich die Schwarz-Markierung an der Wurzel, wird sie einfach gelöscht.

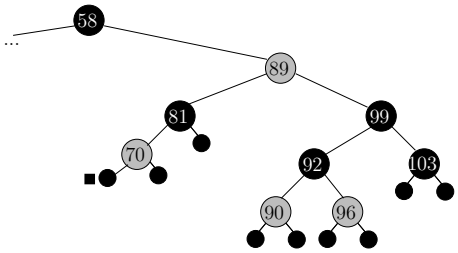
Abbildung 3.14 zeigt als Beispiel die Löschung eines schwarzen Knotens und das anschließende Rebalancieren gemäß obiger Regeln.



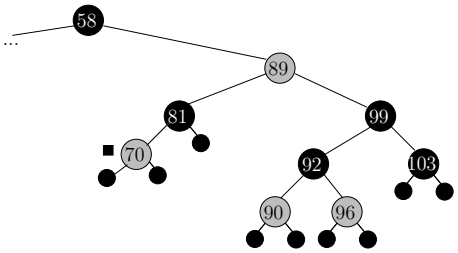
(a) Der Knoten mit Schlüsselwert „63“ soll gelöscht werden.



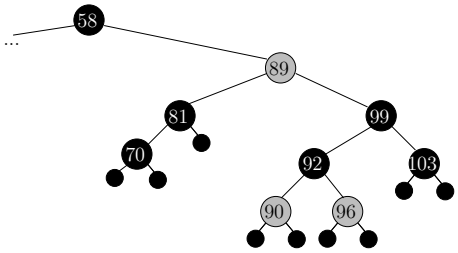
(b) Eines der Blätter dieses gelöschten Knotens wird doppelschwarz gefärbt. Der Bruder des doppelschwarzen Knotens ist rot; daher kann die Rotation aus Fall (c) angewendet werden.



(c) Der Geschwisterknoten des doppelschwarzen Knotens ist schwarz; Neffen existieren nicht. Fall (b) wird also angewendet und die Schwarz-Markierung wandert nach oben.



(d) Hier trifft die Schwarz-Markierung auf einen roten Knoten und kann aufgelöst werden.



(e) Invariante 2 ist wiederhergestellt.

Abb. 3.14: *Beispiel für das Löschen eines Knotens aus einem Rot-Schwarz-Baum.*

3.4 Hashing

Auch das Hashing verfolgt (wie alle anderen in diesem Kapitel vorgestellten Suchalgorithmen) das Ziel, das Einfügen, Suchen und Löschen von Elementen aus einer großen Menge von Elementen effizient zu realisieren. Hashing verwendet jedoch ein im Vergleich zu den bisher vorgestellten Methoden vollkommen anderes und noch dazu einfach zu verstehendes Mittel, um diese Operationen zu implementieren. Die Methode des Hashing ist in vielen Situationen sehr performant. Mittels Hashing ist es möglich, das Einfügen, Suchen und Löschen[4] mit verhältnismäßig einfachen Mitteln mit einer Laufzeit von $O(1)$ zu implementieren. Auch die dem Python Typ *dict* zugrunde liegende Implementierung verwendet Hashing. Zur Veranschaulichung werden wir in diesem Abschnitt das dem *dict*-Typ zugrundeliegende Hashing nachprogrammieren und einem eigenen Typ *OurDict* zugrunde legen.

Für die Implementierung des Hashing ist es zunächst erforderlich, ein genügend großes Array (bzw. in Python: eine genügend große Liste der Länge n) zur Verfügung zu stellen, die sog. *Hash-Tabelle* t. Die Grundidee besteht darin, einen (Such-)Schlüssel k mittels einer sog. *Hash-Funktion* h auf einen Index $h(k)$ der Hash-Tabelle abzubilden; optimalerweise sollte dann der zu k gehörige Wert v an diesem Index der Tabelle gespeichert werden; mittels $t[h(k)]$ kann man also in konstanter Zeit – der Zeit nämlich, um den Rückgabewert von h zu berechnen – auf den Wert v zugreifen. Abbildung 3.15 zeigt diese Situation.

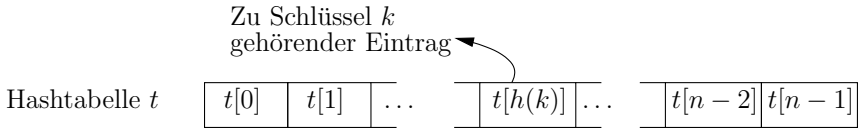


Abb. 3.15: *Hashtabelle t der Größe n. Der zum Suchschlüssel k passende Eintrag befindet sich (optimalerweise) an Tabellenposition $h(k)$, wobei h die verwendete Hashfunktion ist.*

Sind die Schlüssel allesamt ganze Zahlen, so wäre die einfachst denkbare Hashfunktion einfach die Identität, also $h(i) = i$, d. h. jeder Schlüssel k würde so auf den k-ten Eintrag in der Hashtabelle abgebildet werden. In der Praxis ist dies jedoch in der Regel nicht sinnvoll: werden etwa 64 Bit für die Darstellung einer Ganzzahl verwendet, so gibt es 2^{64} verschiedene Schlüssel. Würde man die Identität als Hash-Funktion wählen, so hätte diese auch 2^{64} verschiedene mögliche Werte und man müsste folglich eine Hash-Tabelle mit 2^{64} Einträgen zur Verfügung stellen. Dies entspricht einer Hash-Tabelle der Größe von ca. 16 Mio Terabyte, vorausgesetzt man veranschlagt nur ein Byte Speicherplatz pro Tabelleneintrag. Üblicherweise ist also der Wertebereich aller (sinnvollen und praktisch eingesetzten) Hash-Funktion viel kleiner als deren Definitionsbereich.

[4] Gelegentlich werden diese drei Operationen, nämlich Einfügen, Suchen, Löschen, auch als die sog. „Dictionary Operations“ bezeichnet.

3.4.1 Hash-Funktionen

Eine sinnvolle, praktisch einsetzbare Hashfunktion sollte folgende Eigenschaften besitzen:

1. Sie sollte jeden Schlüssel k auf einen Wert aus $\{0, \ldots, n-1\}$ abbilden.
2. Sie sollte „zufallsartig“ sein, d. h. sie sollte, um Kollisionen zu vermeiden, vorhandene Schlüssel möglichst gleichmäßig über die Indizes streuen.
3. Sie sollte möglichst einfach und schnell berechenbar sein.

Aufgabe 3.21

Welche dieser Eigenschaften erfüllt die „einfachst denkbare Hashfunktion“, also die Identität? Welche Eigenschaften werden nicht erfüllt?

Wir stellen im Folgenden zwei unterschiedliche Methoden vor, Hash-Funktionen zu entwerfen.

Die Kongruenzmethode. Zunächst wandelt man den Schlüssel k in eine Zahl $x = \mathit{integer}(k)$ um und stellt anschließend mittels Restedivision durch eine Primzahl p sicher, dass der berechnete Hashwert sich im Bereich $\{0, \ldots, p-1\}$ befindet, wobei optimalerweise p die Größe der zur Verfügung stehenden Hash-Tabelle ist. Es gilt also

$$h(k) = \mathit{integer}(k) \ \texttt{\%}\ p$$

(wobei „`%`“ Pythons Modulo-Operator darstellt). Und tatsächlich erfüllt diese Hash-Funktion die obigen drei Kriterien: Man kann zeigen, dass sie – vorausgesetzt p ist eine Primzahl – zufallsartig ist, sie bildet den Schlüssel auf den Indexbereich der Hash-Tabelle ab und sie ist einfach zu berechnen.

Da es oft der Fall ist, dass die Schlüsselwerte vom Typ String sind, betrachten wir als Beispiel die Umwandlung eines Strings in eine Zahl. Hat man es mit verhältnismäßig kurzen Strings zu tun, so könnte man die *integer*-Funktion einfach dadurch implementieren, dass man die ASCII-Werte der einzelnen Buchstaben „nebeneinander“ schreibt und dadurch eine (recht große) Zahl erhält, die man mittels Modulo-Rechnung in die Index-Menge $\{0, \ldots, p-1\}$ einbettet. So wäre etwa

$$\mathit{integer}(\texttt{'KEY'}) = 0b\ \underbrace{01001011}_{\mathit{ord}(\texttt{'K'})}\ \underbrace{01000101}_{\mathit{ord}(\texttt{'E'})}\ \underbrace{01011001}_{\mathit{ord}(\texttt{'Y'})} = 4932953$$

Wählt man für p etwa den Wert 163, erhält man so:

$$h(\texttt{'KEY'}) = 4932953 \ \texttt{\%}\ 163 = 84$$

Ein entsprechender Hash-Algorithmus mit zugrundeliegender Hash-Tabelle t mit $\mathit{len}(t) == 163$ würde somit den zum Schlüssel `'KEY'` gehörenden Wert in $t[84]$ suchen.

Folgendes Listing zeigt die Implementierung dieser Hash-Funktion in Python.

```
1 def hashStrSimple(s,p):
2   v=0
3   for i in range(len(s)):
4     j = len(s)-1-i
5     v += ord(s[j]) << (8*i)
6   return v % p
```

Listing 3.10: *Implementierung einer einfachen Hash-Funktion auf Strings*

Pythons „≪“-Operator schiebt alle Bits einer Zahl um eine bestimmte Anzahl von Positionen nach links. In der **for**-Schleife ab Zeile 3 lassen wir die Laufvariable i über alle Indexpositionen des Strings laufen und berechnen so die folgende Summe (wobei $n = len(s)$):

$$\sum_{i=0}^{n-1} ord(s_{n-1-i}) \ll (8*i) \tag{3.3}$$

$$= ord(s_{n-1}) \ll 0 + ord(s_{n-2}) \ll 8 + \ldots + ord(s_0) \ll (8*(n-1)) \tag{3.4}$$

zurückgeliefert wird diese Zahl modulo der übergeben Zahl p, die optimalerweise eine Primzahl sein sollte.

Aufgabe 3.22

Schreiben Sie mittels einer Listenkomprehension die in Listing 3.10 gezeigte Funktion *hashStrSimple* als Einzeiler.

Alternativ könnte der in Listing 3.10 implementierte Algorithmus durch das sog. Horner-Schema implementiert werden.

$$\sum_{i=0}^{n-1} ord(s_{n-1-i}) \ll (8*i)$$

$$= ord(s_{n-1}) + (ord(s_{n-2}) + (ord(s_{n-3}) + (\ldots) \ll 8) \ll 8) \ll 8$$

Beispielsweise könnte nun die Berechnung des Hash-Werts von `'longKey'` folgendermaßen erfolgen:

$$ord(\texttt{y}) + (ord(\texttt{e}) + (ord(\texttt{K}) + (ord(\texttt{g}) + (ord(\texttt{n}) + (ord(\texttt{o}) + ord(\texttt{l}) \ll 8) \ll 8) \ll 8) \ll 8) \ll 8) \% p$$

Das Horner-Schema kann man in Python elegant unter Verwendung der *reduce*-Funktion implementieren:

```
1 def horner(l,b):
2   return reduce(lambda x,y: y+(x<<b), l)
```

Listing 3.11: *Implementierung des Horner-Schemas mittels der higher-order reduce-Funktion.*

Die *reduce*-Funktion ist eine higher-order-Funktion. Sie benutzt die als erstes Argument übergebene Funktion dazu, die Elemente der als zweites Argument übergebenen Sequenz zu verknüpfen. Das erste Argument x, der Argument-Funktion, steht hierbei für den bereits aus den restlichen Elementen berechneten Wert; das zweite Argument y der Argument-Funktion steht hierbei für ein Element aus l.

Aufgabe 3.23

Implementieren Sie das Horner-Schema in einer Schleife – anstatt, wie in Listing 3.11 die Python-Funktion *reduce* zu verwenden.

Während die Größe des berechneten Hashwerts beschränkt ist (denn: $h(k) \in \{0, \ldots, p-1\}$), können jedoch, je nach Länge des gehashten Strings, sehr große Zwischenergebnisse entstehen. Man könnte eine weitere Steigerung der Performance (und sei es nur Platz-Performance) erreichen, indem man das Entstehen sehr großer Zwischenergebnisse vermeidet. Dazu können die folgenden Eigenschaften der Modulo-Funktion ausgenutzt werden:

$$(a+b) \,\%\, p = (a\%p + b\%p) \,\%\, p$$
$$(a*b) \,\%\, p = (a\%p * b\%p) \,\%\, p$$

Man kann also, ohne das Endergebnis zu beeinflussen, in jedem Schleifendurchlauf auf das Zwischenergebnis eine Modulo-Operation anwenden und so sicherstellen, dass keine Zahlen entstehen, die größer als p sind. Listing 3.12 zeigt eine Python-Implementierung des Horner-Schemas, die zusätzlich die eben beschriebene Eigenschaft der Modulo-Funktion ausnutzt.

```
1 def horner2(l,b,p):
2   return reduce(lambda x,y: y+(x<<b)%p, l) %p
```

Listing 3.12: *Implementierung einer für lange Strings performanteren Hash-Funktion unter Verwendung des Horner-Schemas und der eben vorgestellten Eigenschaften der Modulo-Funktion*

Mittels *horner2* kann eine im Vergleich zu der in Listing 3.10 gezeigten Funktion *hashStrSimple* performantere Hash-Funktion geschrieben werden:

```
1 def hashStr(s,p):
2   return horner2(map(ord,s),8,p)
```

Aufgabe 3.24

Verwenden Sie, statt *reduce* und *map*, eine Schleife, um die in Listing 3.12 gezeigte Funktion *hashStr* zu implementieren.

Aufgabe 3.25

Ganz offensichtlich ist nicht, welche der Funktionen *horner* und *horner2* tatsächlich schneller ist – auf der einen Seite vermeidet *horner2* die Entstehung großer Zahlen als Zwischenergebnisse; andererseits werden in *horner2* aber auch sehr viel mehr Operationen (nämlich Modulo-Operationen) ausgeführt als in *horner*.

Ermitteln Sie empirisch, welcher der beiden Faktoren bei der Laufziet stärker ins Gewicht fällt. Vergleichen Sie die Laufzeiten der beiden Funktionen *horner* und *horner2* mit Listen der Länge 100, die Zufallszahlen zwischen 0 und 7 enthalten, mit Parameter $b = 3$ und einer dreistelligen Primzahl. Verwenden Sie zur Zeitmessung Pythons *timeit*-Modul.

Empirisches „Bit-Mixen“. Die Kongruenzmethode liefert zwar i. A. gute Resultate, in der Praxis sieht man jedoch des öfteren andere, theoretisch zwar weniger gut abgesicherte (bzgl. der „Zufälligkeit“) jedoch sehr performante und sich gut bewährende Hash-Funktionen. Eine solche Hash-Methode verwendet Python intern für das Hashing in *dict*-Objekten. Listing 3.13 zeigt eine Nachimplementierung [14] des Algorithmus den Python für das Hashing von Strings verwendet:

```
1  class string:
2      def __hash__( self ):
3          if not self: return 0 # Der leere String
4          value = ord( self [0]) << 7
5          for char in self:
6              value = c_mul(1000003, value) ^ ord(char)
7          return value ^ len( self )
8
9  def c_mul(a, b):
10     return eval(hex((long(a) * b) & 0xFFFFFFFFL)[:-1])
```

Listing 3.13: *Implementierung des dem Python dict-Datentyp zugrundeliegenden Hash-Algorithmus für Strings*

Hierbei soll die Funktion *c_mul* eine übliche C-Multiplikation simulieren, die zwei 32-Bit Ganzzahlen multipliziert. Die Funktion *__hash__* liefert eine 32-Bit-Zahl zurück, deren Bits (hoffentlich) möglichst gut „durchgewürfelt“ wurden. Der ^-Operator verknüpft seine beiden Argumente bitweise durch eine logische XOR-Funktion; bitweise XOR-Verknüpfungen sind ein häufig angewandtes Mittel, um die Bits einer Zahl möglichst durcheinander zu würfeln.

Um später sicherzustellen, dass ein bestimmter Hashwert auch tatsächlich ein gültiger Index-Wert aus der gegebenen Hashtabelle t darstellt, also im Bereich $\{0, \ldots, len(t)\}$ liegt, werden wir später die i niederwertigsten Bits aus dem Hashwert extrahieren. Dafür ist es jedoch auch notwendig, dass die Größe der Hash-Tabelle nicht eine Primzahl p, sondern immer eine Zweierpotenz 2^i ist. Wir zeigen später in diesem Kapitel, wie eine Implementierung dies mit einfachen Mitteln sicherstellen kann.

3.4.2 Kollisionsbehandlung

Die „Zufälligkeit“ der Hash-Funktion soll sicherstellen, dass unterschiedliche Schlüssel k und k' mit $k \neq k'$ mit möglichst geringer Wahrscheinlichkeit auf den selben Index abgebildet werden, d. h. dass mit möglichst geringer Wahrscheinlichkeit $h(k) = h(k')$ gilt. Nehmen wir an, die Hash-Tabelle t habe eine Größe von n Einträgen und m Einträge sind bereits besetzt. Je größer der *Belegungsgrad* $\beta = m/n$ einer Hashtabelle, desto wahrscheinlicher werden jedoch Kollisionen – auch bei einer Hash-Funktion die eine optimale „Zufälligkeit“ garantiert.

Als Kollision wollen wir die Situation bezeichnen, in der ein neu einzufügender Schlüssel k durch die Hash-Funktion auf einen bereits belegten Eintrag in der Hashtabelle abgebildet wird, also $t[h(k)]$ bereits mit dem Wert eines anderen Schlüssls k' belegt ist, für den $h(k) = h(k')$ gilt.

Es gibt mehrere Möglichkeiten, wie man mit dem Problem möglicher Kollisionen umgehen kann. Wir stellen zwei davon vor: Getrennte Verkettung und einfaches bzw. doppeltes Hashing.

Getrennte Verkettung. Bei der getrennten Verkettung legt man jeden Eintrag der Hash-Tabelle als Liste an. Tritt eine Kollision ein, so wird der Eintrag einfach an die Liste angehängt. Abbildung 3.16 zeigt ein Beispiel einer Hash-Tabelle der Größe $n = 11$, die eine bestimmte Menge von Schlüsselwerten (vom Typ „String“) enthält, die mittels getrennter Verkettung eingefügt wurden. Der Index der Schlüssel wurde dabei jeweils mittels der Hash-Funktion $h(k) = hashStr(k,11)$ bestimmt.

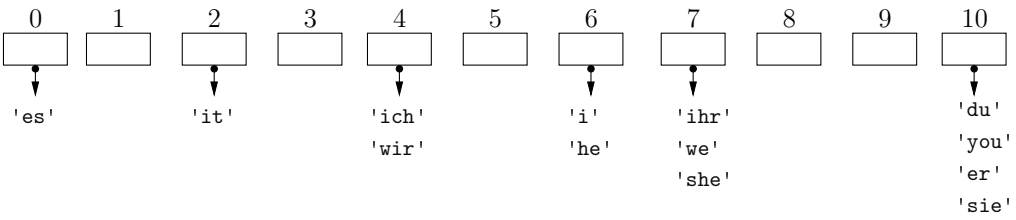


Abb. 3.16: *Eine Hash-Tabelle der Größe $n = 11$, gefüllt mit den String-Werten ['ich','du','er','sie','es','wir','ihr','sie','i','you','he','she', 'it','we']. Als Hash-Funktion wurde die in Listing 3.12 beschriebene Funktion hashStr verwendet. Der Belegungsfaktor ist in diesem Fall $\beta = 13/11$.*

Anders als beim einfachen bzw. doppelten Hashing ist bei der getrennten Verkettung theoretisch ein beliebig großer Belegungsfaktor möglich. Man kann über stochastische Methoden zeigen, dass bei zufällig gewählten Schlüsseln, die *durchschnittliche* Länge der Listen β beträgt, also gleich dem Belegungsfaktor ist. Das bedeutet, dass die Laufzeit für eine...

... erfolglose Suche nach einem Schlüssel $c + \beta$ beträgt, wobei c die Laufzeit für die Berechnung des Hash-Wertes des zu suchenden Schlüssel ist. Die an einem Eintrag befindliche Liste muss vollständig durchsucht werden, bis festgestellt werden kann, dass der Schlüssel sich nicht in der Hash-Tabelle befindet.

... erfolgreiche Suche nach einem Schlüssel $c + \beta/2$ beträgt, denn im Durchschnitt muss die Liste, die sich an einem Eintrag befindet, bis zur Hälfte durchsucht werden, bis der gesuchte Wert gefunden wurde.

Aufgabe 3.26

Wie groß ist die durchschnittliche Listenlänge für die Hashtabelle aus Abbildung 3.16 in der Theorie und konkret am Beispiel?

Einfaches und Doppeltes Hashing. Beim Einfachen bzw. Doppelten Hashing wird bei einer Kollision ein alternativer freier Tabellenplatz gesucht. Das hat zur Folge, dass bei diesen beiden Verfahren der Belegungsfaktor höchstens 1 sein kann, dass also stets $\beta \leq 1$ gelten muss.

Das einfache Hashing geht folgendermaßen vor: Soll der Schlüssel k gespeichert werden und ist die Hash-Tabellenposition $h(k)$ bereits belegt, so wird versucht, k in der Tabellenposition $(h(k)+1) \% n$ zu speichern; ist diese wiederum belegt, so wird versucht k in der Tabellenposition $(h(k)+2) \% n$ zu speichern, usw.

Bei der getrennten Verkettung werden bei der Suche nach einem Schlüssel k evtl. auch weitere Schlüssel k' untersucht, aber nur solche, die auf die gleiche Tabellenposition gehasht werden; beim einfachen Hashing jedoch, kann es vorkommen, dass auch noch Schlüssel mituntersucht werden, die auf andere Tabellenpositionen gehasht werden. Außerdem hat das einfache Hashing den Nachteil, dass eine starke Tendenz zur „Clusterung“ der belegten Einträge besteht; insbesondere unter diesen Clustern kann die Suchperformance sehr leiden. Im Falle des einfachen Hashing beträgt die Laufzeit ...

- ... für eine erfolglose Suche nach einem Schlüssel $\frac{1}{2} + \frac{1}{2(1-\beta)^2}$ Schritte,
- ... für eine erfolgreiche Suche nach einem Schlüssel $\frac{1}{2} + \frac{1}{2(1-\beta)}$ Schritte.

wobei β jeweils den Belegungsfaktor der verwendeten Hash-Tabelle bezeichnet. Zur Begründung hierfür wäre eine aufwändige stochastische Rechnung notwendig, die wir hier der Einfachheit halber nicht aufführen.

Aufgabe 3.27

(a) Fügen Sie mittels einfachem Hashing und Hash-Funktion $h(k) = \mathit{hashStr}(k,11)$ die folgenden Schlüssel in der angegebenen Reihenfolge in eine Hash-Tabelle der Größe 11 ein:

`'er'`, `'ihr'`, `'es'`, `'we'`, `'he'`, `'it'`, `'ich'`

(b) Wie viele Schritte braucht man danach, um nach dem Schlüssel `'ord'` zu suchen?

(c) Wie viele Schritte braucht man danach, um nach dem Schlüssel `'le'` zu suchen?

Beim sog. doppelten Hashing versucht man diese Cluster-Bildung zu vermeiden. Tritt in Tabellenposition $h(k)$ eine Kollision beim Suchen oder Einfügen von Schlüssel k auf, wird hierbei, statt bei der Position $(h(k)+1)\ \%\ p$ fortzufahren, an der Position $(h(k)+u)\ \%\ k$ fortgefahren. Hierbei kann h(k) = $(k+u)\ \%\ p$ als *zweite* Hash-Funktion betrachtet werden, weshalb dieses Verfahren sich doppeltes Hashing nennt. Man kann tatsächlich auch zeigen, dass doppeltes Hashing im Durchschnitt weniger Tests erfordert als lineares Austesten.

3.4.3 Implementierung in Python

Wir wollen die Funktionsweise des Python *dict*-Typs, der intern doppeltes Hashing verwendet, hier nachprogrammieren. Wir erreichen dabei natürlich nicht die Performance des *dict*-Typs, denn dieser ist in der Programmiersprache C implementiert; Python-Code ist, da interpretiert, zwar nicht deutlich, aber immer noch etwas langsamer als auf Performance optimierter C-Code.

Zunächst kann man für die Einträge der Hash-Tabelle eine eigene Klasse definieren; Listing 3.14 zeigt eine passende Klassendefinition zusammen mit deren Konstruktorfunktion *__init__*.

```
1 class Entry(object):
2   def __init__(self):
3     self.key   = None
4     self.value = None
5     self.hash  = 0
```

Listing 3.14: *Definition der Klasse Entry für die Einträge in die Hash-Tabelle*

Jeder Eintrag besteht also aus einem Schlüssel, dem zugehörigen Wert und dem für den Schlüssel berechneten Hash-Wert; aus Performance-Gründen ist es durchaus sinnvoll, sich diesen zu merken anstatt ihn jedesmal neu zu berechnen.

Aufgabe 3.28

Definieren Sie sich eine Instanz der Methode *__str__*, um sich die für den Benutzer relevanten Daten von Objekten vom Typ *Entry* anzeigen zu lassen.

Listing 3.15 zeigt einen Teil der Deklaration der Klasse *OurDict*. Unser Ziel ist es, durch diese Klasse *OurDict* die Funktionsweise des Python-internen Typs *dict* nachzuprogrammieren.

```
1 MINSIZE = 8
2 class OurDict(object):
3   def __init__(self):
4     self.used = 0
5     self.table=[]
6     while len(self.table)<MINSIZE:
```

```
7         self.table.append(Entry())
8       self.mask = 7
9       self.size = MINSIZE
```

Listing 3.15: *Definition der Klasse OurDict*

Das Attribut *used* soll immer angeben, wie viele Schlüssel-Wert-Paare sich in der Hash-Tabelle befinden; das Attribut *table* speichert die eigentliche Hash-Tabelle; diese wird in den Zeilen 6 und 7 initialisiert indem sie mit leeren Einträgen (die mittels *Entry*() erzeugt werden) gefüllt wird. Das Attribut *mask* enthält eine Bit-Maske, die später dazu verwendet wird, den zur Hash-Tabellengröße passenden Teil eines Hash-Wertes zu selektieren; dazu später mehr.

Aufgabe 3.29

In den Zeilen 6 und 7 in Listing 3.15 werden die Einträge der Hash-Tabelle zunächst mit leeren `Entry()`-Werten initialisiert. Was spricht dagegen, statt der `while`-Schleife, dazu den `*`-Operator auf Listen zu verwenden, d. h. die Zeilen 5, 6 und 7 in Listing 3.15 zu ersetzen durch

$$self.table = [Entry()] * MINSIZE \quad ?$$

Den zu einem Schlüssel gehörenden Wert kann man mittels der in Listing 3.16 gezeigten Methode *_lookup* nachschlagen.

```
1  class OurDict(object):
2    ...
3    def _lookup(self, key):
4      hashKey = hashStr(key)
5      i = hashKey &self.mask # Selektion der benötigten Bits
6      entry = self.table[i]
7      if entry.key==None or entry.key==key: # gefunden!
8        return entry
9
10     # Falls entry.key != key: wende zweite Hashfunktion an.
11     perturb = hashKey
12     while True:
13       i = (i<<2) + i + perturb+1
14       entry = self.table[i & self.mask]
15       if entry.key==None or entry.key==key:
16         return entry
17       perturb = perturb >> PERTURB_SHIFT
```

Listing 3.16: *Implementierung der _lookup-Methode, die einen gegebene Schlüssel im Dictionary nachschlägt und den Eintrag zurückliefert*

Zeile 4 berechnet zunächst den Hash des Schlüssels und verwendet dazu den in Listing 3.13 angegebenen Algorithmus. In Zeile 5 selektieren wir mittels der bitweisen Und-Operation „`&`“ die benötigten Bits des Hashs. Welche Bits aktuell benötigt werden, hängt wiederum von der momentanen Größe der Hash-Tabelle ab. In den Zeilen 7 und 8 wird schließlich der in $self.table[i]$ befindliche Eintrag zurückgeliefert, falls entweder der Schlüssel dieses Eintrags mit dem Suchschlüssel übereinstimmt, oder der Eintrag noch leer war; stimmt der Schlüssel jedoch nicht mit dem Suchschlüssel überein, so könnte es sich um eine Kollision handeln, und es wird mittels einer zweiten Hash-Funktion weiter nach einem Eintrag gesucht, der zu dem Schlüssel passt. Hierbei gilt für die zweite Hash-Funktion ein ähnliches pragmatisches Prinzip wie für die „erste“ Hash-Funktion: die Bits müssen möglichst gut durcheinandergewürfelt werden, um eine optimale Streuung zu gewährleisten, um Clusterung zu vermeiden.

Aufgabe 3.30

Angenommen, unsere Hash-Tabelle hat eine Größe von 2^{20} und enthält 900 000 Werte. Angenommen, wir würden *keine* zweite Hash-Funktion verwenden wollen, sondern stattdessen einfaches Hashing.

(a) Passen Sie hierfür die **while**-Schleife in Zeile 12 aus Listing 3.16 so an, dass sie den Schlüssel *key* unter der Annahme sucht, dass die Hash-Tabelle mit linearem Hashing befüllt wurde.

(b) Wie oft müsste die so implementierte **while**-Schleife im Durchschnitt durchlaufen werden, bis ein in der Hash-Tabelle befindlicher Schlüssel gefunden wird?

(c) Wie oft müsste die so implementierte **while**-Schleife im Durchschnitt durchlaufen werden, bis die *_lookup*-Funktion „merkt“, dass der zu suchende Schlüssel *key* sich nicht in der Hash-Tabelle befindet?

Aufgabe 3.31

Passt man nicht genau auf, so kann es passieren, dass die **while**-Schleife in Zeile 12 aus Listing 3.16 eine Endlosschleife wird. Wie könnte dies passieren und wie genau kann man sicherstellen, dass diese Schleife immer terminiert?

Wie werden aber die relevanten Bits des Hashs selektiert? Starten wir mit einem leeren Dictionary, so hat zu Beginn die Hash-Tabelle 8 Einträge (siehe Zeile 1 und 9 in Listing 3.15); um einen Schlüssel auf eine Hash-Tabellenposition abzubilden, müssen wir hier die letzten 3 Bits selektieren ($self.mask$ müsste in diesem Fall also den Wert 7 haben). Nehmen wir beispielsweise an, der Hash-Wert eines Schlüssel würde sich zu $hashKey = 18233$ ergeben. Schreibt man nun den Inhalt von $hashKey$, $self.mask$ und i in Binärdarstellung auf, so sieht man leicht, dass sich für den Wert von i durch bitweise Und-Verknüpfung der Wert „1“ ergibt:

$hashKey$	=	18233	=	0100 0111 0011 1001
$self.mask$	=	7	=	0000 0000 0000 0111 &
i	=			0000 0000 0000 0001

Allgemein kann man durch Wahl von $self.mask = 2^i - 1$ mittels $hashKey \& self.mask$ die niederwertigsten i Bits von *haskKey* selektieren und diese Selektion als gültigen Index in einer 2^i-großen Hash-Tabelle interpretieren. Wichtig hierfür ist, sicherzustellen, dass die Größe n der Hash-Tabelle immer eine Zweierpotenz ist, d. h. dass $n = 2^i$ für eine $i \in \mathbb{N}$ gilt.

Aufgabe 3.32

Die Selektion der i niederwertigsten Bits entspricht eigentlich der Operation „% 2^i“. Dies widerspricht eigentlich der Empfehlung aus Abschnitt 3.4.1, man solle als Hash-Funktion „% p“ mit p als Primzahl verwenden. Argumentieren Sie, warum dies hier durchaus sinnvoll ist.

Mit Hilfe der *_lookup*-Funktion ist, wie in Listing 3.17 zu sehen, das Einfügen eines neuen Elements bzw. Ersetzen eines bestehenden Elements relativ einfach zu realisieren:

```
1 class OurDict(object):
2   ...
3   def _insert( self,key,value):
4     entry = self._lookup(key)
5     if entry.value==None: self.used += 1
6     entry.key = key
7     entry.hash = hashStr(key)
8     entry.value = value
```

Listing 3.17: *Die _insert-Methode ist eine „interne“ Funktion, mit der ein Element in eine OurDict-Objekt eingefügt werden kann*

Die Funktion *_lookup* liefert denjenigen Eintrag zurück, der mit dem einzufügenden Schlüssel-Wert-Paar zu füllen ist. In Zeile 5 wird der „Füllstandsanzeiger“ der Hash-Tabelle *self.used* angepasst, aber nur dann, wenn auch tatsächlich ein neuer Wert eingefügt (und nicht ein alter ersetzt) wurde. Die *_insert*-Methode sollte jedoch nicht direkt vom Benutzer verwendet werden; die Schnittstelle für das Einfügen eines Elementes bietet die *__setitem__*-Methode; Listing 3.18 zeigt eine Implementierung. Stellt eine Klasse eine Implementierung der *__setitem__*-Methode zur Verfügung, so wird eine Zuweisung der Form *x*[*key*]=*value* automatisch in einen Aufruf der Form *x*. *__setitem__*(*key*, *value*) überführt. In Zeile 5 in Listing 3.18 findet das eigentliche Einfügen des übergebenen Schlüssel-Wert-Paares statt. Wozu aber der Code in den Zeilen 7 und 8?

Ein Problem beim Hashing besteht darin, dass die Größe der Hash-Tabelle eigentlich fest vorgegeben werden sollte. Bei der Deklaration und Verwendung eines *dict*-Objektes „weiß“ Python jedoch nicht im Voraus, wie viele Werte in der Hash-Tabelle gespeichert

```
class OurDict(object):
  ...
  def __setitem__(self,key,value):
    oldUsed = self.used
    self._insert(key,value)
    # Muss die Hashtabellengröße angepasst werden?
    if (self.used>oldUsed and self.used*3 >=(self.mask+1)*2):
      self._resize(2*self.used)
```

Listing 3.18: *Mit Hilfe der __setitem__ -Methode kann der Benutzer komfortabel ein Schlüssel-Wert-Paar in ein Objekt vom Typ OurDict einfügen.*

werden sollen; diese Information kann nicht *statisch*[5] bestimmt werden, sondern sie ergibt sich erst *dynamisch*, also während das Programm ausgeführt wird (sprich: zur „Ausführungszeit“).

Aufgabe 3.33

Die Implementierung des Python-internen *dict*-Typs unterscheidet bei der Anpassung der Größe der Hash-Tabelle die folgenden beiden Fälle:

(a) Ist die Länge der momentanen Hash-Tabelle größer als 4096, so wird, falls erforderlich die Größe immer verdoppelt.

(b) Ist die Länge der momentanen Hash-Tabelle kleiner als 4096, so wird, falls erforderlich, die Größe immer vervierfacht.

Passen Sie die in Listing 3.18 gezeigte Implementierung entsprechend an.

Listing 3.19 zeigt die Implementierung der Größenanpassung der Hash-Tabelle.

```
class OurDict(object):
  ...
  def _resize(self, minused):
    newsize=MINSIZE
    while newsize<=minused and newsize>0: newsize=newsize<<1
    oldtable = self.table
    newtable = []
    while len(newtable) < newsize:
      newtable.append(Entry())
```

[5] Der Informatiker spricht von „statisch“, wenn er meint: vor der Ausführung eines Programms bzw. zur „Compilezeit“, also während der Analyse des Programmcodes. Es gibt viele Informationen, die vor Ausführung des Programms nur sehr schwer oder auch gar nicht bestimmt werden können. So ist es i. A. unmöglich statisch zu bestimmen, ob ein Programm anhalten wird oder in eine Endlosschleife läuft – dies wird in der Literatur häufig als das sog. „Halteproblem“ bezeichnet.

```
10      self.table = newtable
11      self.used = 0
12      for entry in oldtable:
13          if entry.value==None:
14              self._insert_init(entry)
15      self.mask = newsize-1
16      self.size = newsize
```

Listing 3.19: *Mit Hilfe der _resize -Methode kann die Länge der Hash-Tabelle, falls notwendig, vergrößert werden.*

Man sieht, dass diese Größenanpassung der Hash-Tabelle ein kritischer Punkt in der Performance des *dict*-Typs bzw. des *OurDict*-Typs ist. Denn hier wird eine neue Tabelle mit mindestens doppelter Größe der alten Tabelle neu angelegt (Zeilen 4–9) und anschließend *alle* vorhandenen Einträge aus der alten Tabelle in die neue Tabelle kopiert (Zeilen 11–14). Die Funktion *_resize* hat offensichtlich eine Laufzeit von $O(n)$, wobei n die Größe der Hash-Tabelle ist, was bei sehr großen Hash-Tabellen durchaus kritisch sein kann. Aus Performance-Gründen wird beim Einfügen der Einträge aus der alten Tabelle in die Neue nicht die *_insert*-Funktion verwendet, sondern eine eigens für diese Situation geschriebene Einfüge-Funktion *_insert_init* ; diese berechnet die (schon berechneten) Hash-Werte der Einträge nicht neu, sondern verwendet die schon vorhandenen *entry.hash*-Werte; außerdem vermeidet *_insert_init* zur weiteren Optimierung Funktionsaufrufe.

Aufgabe 3.34

Programmieren Sie die Funktion *_insert_init* .

Aufgabe 3.35

Definieren Sie für den *OurDict*-Typ die Methode *__getitem__*, mit deren Hilfe man einfach den Wert eines Schlüssels durch Indizierung erhält.

Aufgabe 3.36

Implementierung Sie für den *OurDict*-Typ eine Möglichkeit, Elemente zu löschen, d. h. definieren Sie eine Instanz der Methode *__delitem__*. Auf was müssen Sie dabei besonders achten?

Aufgabe 3.37

Warum ist es nicht sinnvoll, dem Python-Typ *list* eine Implementierung der *__hash__*-Methode zu geben? In anderen Worten: warum können Listen nicht als Index eines *dict*-Objekt dienen? Was könnte schief gehen, wenn man auf ein Element mittels eine Liste zugreifen möchte, wie etwa in folgendem Beispiel:

```
>>> lst = [1,2,3]
>>> d = { lst:14, 'Hugo':991 }
```

3.5 Bloomfilter

Die erstmals von Burton Bloom [2] vorgestellte Datenstruktur, bietet (ähnlich wie die später beschriebene Union-Find-Datenstruktur) eine sowohl sehr platz- als auch zeiteffiziente Möglichkeit, zu testen, ob sich ein Datensatz in einer bestimmten Datenmenge befindet. Bloomfilter bieten lediglich zwei Operationen an: das Hinzufügen eines Datensatzes und einen Test, ob ein bestimmter Datensatz bereits enthalten ist – im Weiteren auch oft mit *Membership-Test* bezeichnet. Es ist dagegen nicht möglich, ein Element aus einem Bloomfilter zu löschen.

Ein Bloomfilter ist eine probabilistische Datenstruktur und kann falsche Antworten auf einen Membership-Test liefern: Befindet sich ein Datensatz in der Menge, so antwortet das Bloomfilter immer korrekt. Befindet sich jedoch ein Datensatz *nicht* in der Menge, so kann (i. A. mit geringer Wahrscheinlichkeit) das Bloomfilter eine falsch-positive Antwort liefern.

3.5.1 Grundlegende Funktionsweise

Ein Bloomfilter besteht aus einem Array A der Größe m mit booleschen Einträgen. Das einzufügende Element e wird auf eine Familie von k Hash-Funktionen $h_0, \ldots h_{k-1}$ angewendet. Um e schließlich einzufügen, werden die Array-Einträge an den Positionen $h_0(e)\,\%\,m, \ldots, h_{k-1}(e)\,\%\,m$ des Arrays A auf *True* gesetzt.

Nehmen wir als Beispiel an, wir hätten zwei Hashfunktionen h_0 und h_1, ein Array mit 10 Positionen, und wir wollten die Strings `eine`, `Einführung` und `Informatik` einfügen. Nehmen wir folgende Hash-Werte der Strings an:

$$\begin{array}{lll} h_0(\texttt{eine}) = 3, & h_0(\texttt{Einführung}) = 1, & h_0(\texttt{Informatik}) = 6 \\ h_1(\texttt{eine}) = 1, & h_1(\texttt{Einführung}) = 8, & h_1(\texttt{Informatik}) = 7 \end{array}$$

Abbildung 3.17 zeigt, was beim Einfügen der drei Strings in das Bloomfilter geschieht.

Will man überprüfen, ob ein Element e im Bloomfilter enthalten ist, so überprüft man, ob $A[h_0(e)] = A[h_1(e)] = \ldots = A[h_{k-1}(e)] = \mathit{True}$ gilt. Es gibt zwei Fälle:

- Mindestens einer der Einträge $A[h_0(e)], \ldots A[h_{k-1}(e)]$ hat den Wert *False*. In diesem Fall können wir sicher davon ausgehen, dass e bisher noch nicht in das

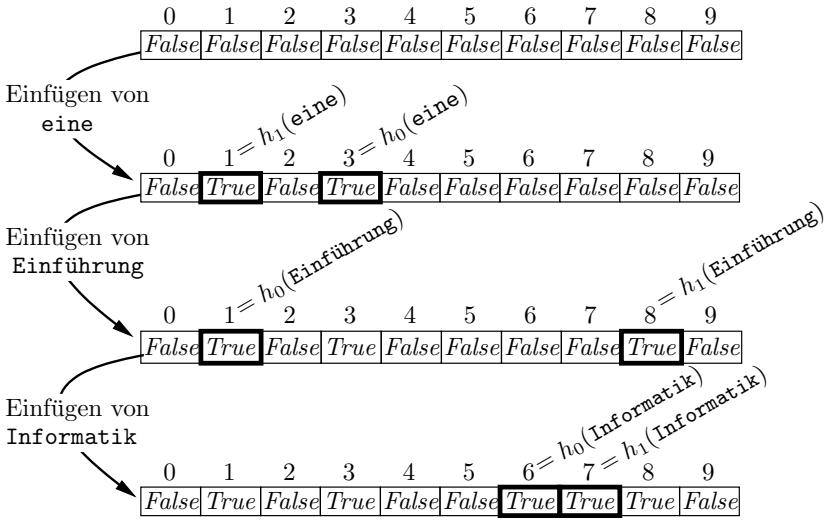


Abb. 3.17: *Einfügen der drei Elemente* `eine`*,* `Einführung` *und* `Informatik` *in ein Bloomfilter der Länge 10 unter Verwendung der beiden Hash-Funktionen h_0 und h_1.*

Bloomfilter eingefügt wurde; andernfalls hätten nämlich alle diese Einträge den Wert *True*.

- Alle Einträge $A[h_0(e)], \ldots A[h_{k-1}(e)]$ haben den Wert *True*. In diesem Fall können wir annehmen, dass e schon in das Bloomfilter eingefügt wurde. Diese Annahme entspricht jedoch nicht mit 100%-Wahrscheinlichkeit der Wahrheit. Es kann vorkommen, dass alle diese Einträge zufällig in Folge anderer Einfügeoperationen schon auf *True* gesetzt wurden.

Nehmen wir obiges Beispiel und überprüfen das durch Einfügen von `eine`, `Einführung` und `Informatik` entstandenen Bloomfilter daraufhin, ob es die beiden Strings `Algorithmik` und `praktisch` enthält. Wir gehen von folgenden Hash-Werten aus:

$$h_0(\texttt{Algorithmik}) = 9, \quad h_0(\texttt{praktisch}) = 1,$$
$$h_1(\texttt{Algorithmik}) = 3, \quad h_1(\texttt{praktisch}) = 7,$$

Abbildung 3.18 zeigt wie es zu falsch-positiven Antworten kommen kann. Das Bloomfilter liefert fälschlicherweise die Aussage, dass der String `praktisch` bereits ins Bloomfilter eingefügt wurde (denn $h_0(\texttt{praktisch}) = \mathit{True}$ und $h_1(\texttt{praktisch}) = \mathit{True}$).

Aufgabe 3.38

Worin unterscheidet sich einfaches Hashing von einem Bloomfilter mit $k = 1$?

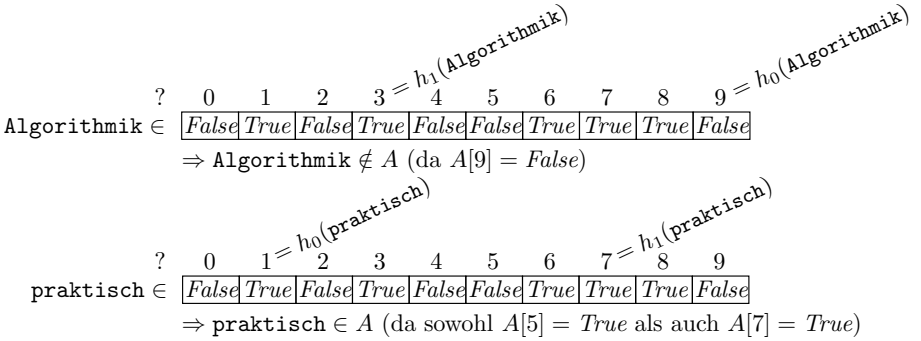


Abb. 3.18: *Zwei Membership-Tests des Bloomfilters aus Abbildung 3.17 auf die Strings* `Algorithmik` *und* `praktisch`*. Der zweite Test, der prüft, ob* `praktisch` *bereits ins Bloomfilter eingefügt wurde, liefert ein falsches Ergebnis.*

3.5.2 Implementierung

Listing 3.20 zeigt die Implementierung eines Bloomfilters in Python:

```
1  class BloomFilter(object):
2    def __init__(self, h, m):
3      self.k = len(h) ; self.h = h
4      self.A = [False]*m
5      self.m = m
6
7    def insert(self,x):
8      for hashFkt in self.h: self.A[hashFkt(x)] = True
9
10   def elem(self,x):
11     return all([self.A[hashFkt(x)] for hashFkt in self.h])
```

Listing 3.20: *Implementierung eines Bloomfilters.*

Das Bloomfilter wird durch die Klasse *BloomFilter* implementiert. Die Liste *self.h* speichert die k Hashfunktionen; die Liste *self.A* enthält die Array-Einträge des Bloomfilters; alle Einträge werden in Zeile 4 mit *False* initialisiert.

Die Einfüge-Operation ist durch die Methode *insert* implementiert. Die **for**-Schleife in Zeile 8 durchläuft in der Variablen *hashFkt* die k Hashfunktionen des Bloomfilters; der Ausdruck *hashFkt*(x) definiert eine der k Positionen des Arrays *self.A*, die im Zuge der Einfüge-Operation auf *True* gesetzt werden müssen.

Ähnlich einfach ist die Implementierung der Methode *elem*, die testet, ob ein Element x sich im Bloomfilter befindet. Die Listenkomprehension in Zeile 13 sammelt alle k relevanten Einträge von *self.A* in einer Liste auf; haben alle Werte dieser Liste den Wahrheitswert *True*, so wird angenommen, dass x sich im Bloomfilter befindet.

Aufgabe 3.39

(a) Erklären Sie, warum folgender Methode der Klasse *BloomFilter nicht* geeignet ist, ein Element aus dem Bloomfilter zu entfernen:

```
def delete( self ,x):
  for i in range(0, self .k):  self .A[self .h[i](x) % self .m] = False
```

(b) Schreiben Sie die Methode *delete* so um, dass sie ebenfalls das Element x löscht, jedoch mit möglichst wenig „Seiteneffekten“.

(c) Warum ist selbst die in der letzten Teilaufgabe programmierte Lösch-Funktion in vielen Fällen nicht sinnvoll?

Aufgabe 3.40

Eine bessere Möglichkeit (als die in Aufgabe 3.39 vorgestellte), eine Lösch-Funktion in einem Bloomfilter zu implementieren, besteht darin, sich die gelöschten Elemente in einem zweiten Bloomfilter zu merken.

(a) Schreiben Sie eine Methode *deleteSB* die eine solche Lösch-Funktion implementiert. Passen Sie dabei, wenn nötig, die Klasse *BloomFilter* an; passen Sie die Methode *elem* entsprechend an.

(b) Kann durch das Löschen mittels *deleteSB* auch eine Situation entstehen, in der falsch-negative Antworten auf Membership-Tests gegeben werden? Vergleichen Sie diese Lösch-Funktion mit der in Aufgabe 3.39 vorgestellten Lösch-Funktion.

Aufgabe 3.41

Eine bessere Möglichkeit (als die in Aufgabe 3.40 vorgestellte), eine Lösch-Funktion zu implementieren, ist die Verwendung eines sog. *Countingfilters*. Ein Countingfilter ist ein Bloomfilter, dessen Einträge keine Bitwerte (d. h. *True* oder *False*) sind, sondern Zähler. Anfänglich sind alle Einträge 0; mit jeder Einfüge-Operation werden die durch die Hash-Funktion bestimmten Einträge des Bloomfilter-Arrays jeweils um Eins erhöht.

(a) Implementieren Sie, angelehnt an die in Listing 3.20 gezeigte Implementierung der Klasse *BloomFilter*, eine Klasse *CountingFilter*, die einen Countingfilter implementiert. Implementieren Sie eine Methode *insert*, die ein Element einfügt, und eine Methode *elem*, die testet, ob ein Element in dem Bloomfilter enthalten ist.

(b) Implementieren Sie eine Methode *delete*, die ein Element in einem Bloomfilter löscht.

Aufgabe 3.42

Gegeben seien zwei Bloomfilter B_1 und B_2, mit $len(B_1.array) == len(B_2.array)$ (d. h. die Arrays der Bloomfilter haben die gleiche Länge) und $B_1.h == B_2.h$ (d. h. die beiden Bloomfilter verwenden die gleiche Menge von Hash-Funktionen).

(a) Erklären Sie, wie man die Mengen, die die beiden Bloomfilter B_1 und B_2 repräsentieren, in einem neuen Bloomfilter vereinigen kann.
Schreiben Sie eine entsprechende Python-Funktion $unionBF(B_1, B_2)$, die diese Vereinigung implementiert.

(b) Erklären Sie, wie man die Mengen, die die beiden Bloomfilter B_1 und B_2 repräsentieren, in einem neuen Bloomfilter schneiden kann.
Schreiben Sie eine entsprechende Python-Funktion $intersectBF(B_1, B_2)$, die diesen Schnitt implementiert.

3.5.3 Laufzeit und Wahrscheinlichkeit falsch-positiver Antworten

Sowohl das Einfügen, als auch der Membership-Test benötigen jeweils $O(k)$ Schritte, um die k Hash-Funktionen zu berechnen. Die Laufzeit ist also – und das ist das eigentlich Bemerkenswerte an einem Bloomfilter – unabhängig von der Anzahl n der im Bloomfilter enthaltenen Einträge.

Eine entscheidende Frage bzgl. der Performance eines Bloomfilters bleibt jedoch: Wie groß ist die Wahrscheinlichkeit eines falsch-positiven Membership-Tests? Wir gehen im Folgenden von der (nur näherungsweise korrekten) Annahme aus, die Funktionswerte der k Hash-Funktionen seien alle unabhängig und perfekt pseudo-zufällig verteilt. Dann können wir annehmen, dass die Wahrscheinlichkeit, ein bestimmtes Bit aus den m Einträgen des Bit-Arrays würde durch eine bestimmte Hash-Funktion h_i gesetzt, genau $1/m$ ist; die Gegenwahrscheinlichkeit, d. h. die Wahrscheinlichkeit, dass dieses Bit *nicht* gesetzt wird, ist entsprechend $1 - 1/m$. Die Wahrscheinlichkeit, dass dieses Bit durch keine der k Hashfunktionen $h_0, \ldots h_{k-1}$ gesetzt wird, ist also $(1 - 1/m)^k$. Befinden sich bereits n Elemente im Bloomfilter, so ist die Wahrscheinlichkeit, dass dieses Bit durch keine der n Einfügeoperationen gesetzt wurde $(1 - 1/m)^{kn}$. Die Gegenwahrscheinlichkeit, d. h. die Wahrscheinlichkeit, dass dieses Bit durch eine der n Einfügeoperationen gesetzt wurde, ist $1 - (1 - 1/m)^{kn}$.

Die Wahrscheinlichkeit FPT eines falsch-positiven Tests, d. h. die Wahrscheinlichkeit dass *alle* für einen Eintrag relevanten k Bits bereits gesetzt wurden ist also

$$FPT = \left(1 - (1 - 1/m)^{kn}\right)^k$$

Für den Designer eines Bloomfilters stellen sich zwei entscheidende Fragen:

1. Welcher Wert sollte für k gewählt werden?. Wie viele Hash-Funktionen sollten optimalerweise für ein Bloomfilter der Größe m und einer erwarteten Anzahl von n

Einträgen verwendet werden, d. h. welche Anzahl k von Hash-Funktionen minimiert die Wahrscheinlichkeit falsch-positiver Aussagen?

Um diese Fragen zu beantworten, müssen wir zunächst den Ausdruck der Wahrscheinlichkeit eines falsch-positiven Tests etwas vereinfachen. Da $(1-1/m)^x \approx e^{-x/m}$ (durch Taylorreihenentwicklung einfach nachzuvollziehen.) gilt für die Wahrscheinlichkeit FPT eines falsch-positiven Tests:

$$FPT = \left(1-(1-1/m)^{kn}\right)^k \approx \left(1-e^{-kn/m}\right)^k =: FPT_{\approx}$$

Will man das Minimum dieses Ausdrucks – betrachtet als Funktion nach k – finden, so sucht man die Nullstellen der Ableitung; leichter ist es jedoch (was sich erst nach einiger Rechnerei herausstellt), den Logarithmus dieses Ausdruck zu minimieren. Leiten wir zunächst den Logarithmus von FPT nach k ab

$$ln(FPT_{\approx})' = \left[k \cdot \ln(1-e^{-kn/m})\right]' = \ln(1-e^{-kn/m}) + \frac{kn}{m} \cdot \frac{e^{-kn/m}}{1-e^{-kn/m}}$$

Eine Nullstelle liegt bei $k = (\ln 2) \cdot \frac{m}{n}$, und man kann auch tatsächlich zeigen, dass dies ein Minimum ist.

2. Welcher Wert sollte für m gewählt werden?. Oft möchte man die Fehlerrate eines Bloomfilters begrenzen. Die entscheidende Frage hierzu ist: Wie groß sollte das Bloomfilter-Array gewählt werden, wenn man – bei einer erwarteten Anzahl von n Einträgen – sicherstellen möchte, dass die Wahrscheinlichkeit falsch-positiver Aussagen höchstens p sein wird?

Die Herleitung einer entsprechenden Formel ist noch aufwändiger als obige Herleitung der optimalen Wahl von k und wir überlassen es dem interessierten Leser sich in entsprechender Literatur [4] darüber zu informieren. Folgende Formel liefert die Mindestgröße m eines Bloomfilters mit n gespeicherten Elementen, die die Wahrscheinlichkeit falsch-positiver Aussagen auf höchstens p beschränkt.

$$m \geq n \log_2(1/p)$$

Aufgabe 3.43

Beantworten Sie die folgenden Fragen:

(a) Wie viele Hash-Funktionen sollte man verwenden, bei einem Bloomfilter der Größe 1 MBit, das etwa 100000 Elemente speichern soll?

(b) Wie viele Bits pro gespeichertem Eintrag werden von einem Bloomfilter benötigt, dessen Fehlerrate höchstens bei 1% liegen soll?

(c) Wie viele Bits pro gespeichertem Eintrag werden von einem Bloomfilter benötigt, dessen Fehlerrate höchstens bei 0.1% liegen soll?

Aufgabe 3.44

(a) Erklären Sie, wie man mit Hilfe eines Bloomfilters eine schnelle und speichereffiziente Rechtschreibprüfung implementieren kann.

(b) Gehen wir von einem Wörterbuch mit 100000 Einträgen aus; wir wollen sicherstellen dass die Rechtschreibprüfung nur bei höchstens jedem 1000sten Wort einen Fehler begeht. Wie groß muss das Bloomfilter gewählt werden? Wie viele Hash-Funktionen sollten optimalerweise verwendet werden?

(c) Implementieren Sie die Rechtschreibprüfung. Verwenden Sie die in Listing 3.20 gezeigte Implementierung von Bloomfiltern. Recherchieren Sie, welche Hash-Funktionen sinnvoll sein könnten und verwenden Sie diese; evtl. ist es auch sinnvoll aus einer einzelnen Hash-Funktion durch Gruppierung der Bits mehrere Hash-Funktionen zu generieren. Implementieren Sie eine Funktion $\mathit{richtig}(s)$, die mit Hilfe des Bloomfilters testet, ob der String s sich im Wörterbuch befindet.

3.5.4 Anwendungen von Bloomfiltern

Sehr beliebt ist der Einsatz von Bloomfiltern, um die Antwortzeiten von Datenbanken oder langsamen Massenspeichern zu beschleunigen. Ferner gibt es eine wachsende Zahl von Anwendungen, deren Anwendungsfälle nicht auf das klassische Paradigma einer relationalen Datenbank passen; hierzu wurde in neuster Zeit der Begriff *NoSQL* (für: „Not only SQL“) geprägt. Bloomfilter stellen eine häufig gewählte Technik dar, um Daten in nicht-relationalen Datenbanken, wie etwa dokumentenorientiert verteilte Datenbanken, zu strukturieren.

Um eines (von sehr vielen) Beispielen zu geben: Bloomfilter werden in Googles BigTable [5], einem verteilten Ablagesystem für unstrukturierte Daten, verwendet, um die Anzahl von Suchaktionen zu reduzieren. Hierbei wird jede Anfrage an die Datenbank zunächst an ein Bloomfilter gegeben, das alle in der Datenbank enthaltenen Schlüsselwerte enthält. Befindet sich ein Schlüssel nicht in der Datenbank, so antwortet das Bloomfilter korrekt (und sehr schnell, nämlich mit konstanter Laufzeit) und die Anfrage muss nicht weiter von der langsameren Datenbank bearbeitet werden. Befindet sich der Schlüssel im Bloomfilter, so muss allerdings direkt auf die Datenbank bzw. den Massenspeicher zugegriffen werden (zum Einen um auszuschließen, dass das Bloomfilter eine falsch-positive Antwort gegeben hat; zum Anderen um den zum Schlüssel passenden Wert aus der Datenbank zu holen und zurückzuliefern). Abbildung 3.19 zeigt diese Technik nochmals graphisch.

Es gibt eine Reihe von Netzwerk-Anwendungen, in denen die Verwendung eines Bloomfilters sehr sinnvoll sein kann. Wir geben eines (von vielen möglichen) Beispielen – die Implementierung eines sog. *Web-Proxys*. Die Hauptaufgabe eines Web-Proxys ist die Reduktion von Web-Traffics, also der über das Netzwerk bzw. Internet verschickten Datenmenge. Wird diese Datenmenge verringert, so kann damit i. A. die Zugriffsgeschwindigkeit auf Web-Seiten verbessert werden. Diese Geschwindigkeitserhöhung wird durch *Caching* häufig genutzter Seiten erreicht, d. h. auf den Proxys befindliche sog. *Web-Caches* speichern häufig genutzte Web-Dokumente und sind so für Rechner die

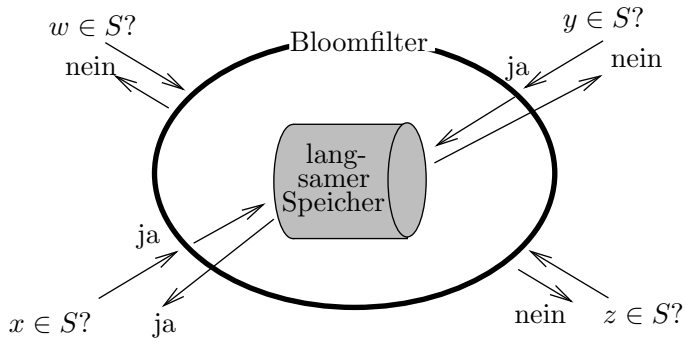


Abb. 3.19: *Ein Bloomfilter kann dazu verwendet werden, die Zugriffe auf einen langsamen Massenspeicher (wie etwa eine Festplatte oder ein noch langsameres Bandlaufwerk) zu reduzieren. In den meisten Fällen, in denen sich ein Element nicht auf dem langsamen Speicher S befindet, kann so bei einer Anfrage der Zugriff auf S vermieden werden; in dem Beispiel ist dies bei den Anfragen „$w \in S$?“ und „$z \in S$?“ der Fall. Nur wenn das Bloomfilter eine positive Antwort liefert, muss direkt auf S zugegriffen werden, zum Einen um auszuschließen, dass es sich bei der Antwort des Bloomfilters um eine falsch-positive Aussage handelte (das ist im Beispiel bei der Anfrage „$y \in S$?“ der Fall); zum Anderen natürlich um die angefragten Informationen aus S zu holen und dem Benutzer zurückzuliefern.*

den Web-Proxy nutzen schneller erreichbar als wenn sie von Ihrer ursprünglichen Quellen geladen werden müssten. Dieses Proxy-Konzept kann noch um ein Vielfaches effektiver gestaltet werden, wenn sich Web-Proxys untereinander Informationen über den Inhalt ihrer Caches austauschen: Im Falle eines *Cache-Miss*[6] versucht der Web-Proxy das angeforderte Web-Dokument aus dem Cache eines anderen Web-Proxys zu beziehen. Hierzu müssen Proxys über den Inhalt der Caches anderer Proxy bescheid wissen. Anstatt aber die kompletten Inhalte der Caches über das Internet auszutauschen (was aufgrund deren Größe sehr teuer wäre), werden in regelmäßigen zeitlichen Abständen Bloomfilter verschickt, die die Einträge der Caches beinhalten. Der prominente „Squid Web Proxy Cache“ verwendet beispielsweise Bloomfilter.

Aufgabe 3.45

Erklären Sie, warum in diesem Falle der Implementierung eines Web-Proxys die Eigenschaft der Bloomfilter, mit einer gewissen Wahrscheinlichkeit falsch-positive Antworten zu geben, vollkommen unproblematisch ist.

[6] Mit Cache-Miss bezeichnet man die Situation, dass sich eine angeforderte Seite nicht im Cache des jeweiligen Proxys befindet.

3.6 Skip-Listen

Die erst 1990 von William Pugh [16] eingeführten *Skip-Listen* bilden eine einfache und in vielen Fällen sehr effiziente Implementierung der Dictionary-Operationen „Einfügen“, „Suchen“ und „Löschen“. Tatsächlich erweisen sich Skip-Listen oft als die einfachere und effizientere Alternative zu einer Implementierung über balancierte Baumstrukturen. Skip-Listen stellen eine sog. *randomisierte* Datenstruktur dar: Beim Aufbau einer Skip-Liste bzw. beim Einfügen von Elementen in eine Skip-Liste werden gewisse Zufallsentscheidungen getroffen, auf die wir später genauer eingehen werden.

Ähnlich wie bei einfachen verketteten Listen sind die Einträge in einer Skip-Liste durch Zeiger verkettet. Es besteht jedoch ein wesentlicher Unterschied zu verketteten Listen: Ein Element einer Skip-Liste kann mehrere Vorwärtszeiger enthalten. Abbildung 3.20 zeigt ein Beispiel. Die Anzahl der Vorwärtszeiger eines Eintrags bezeichnen wir als die

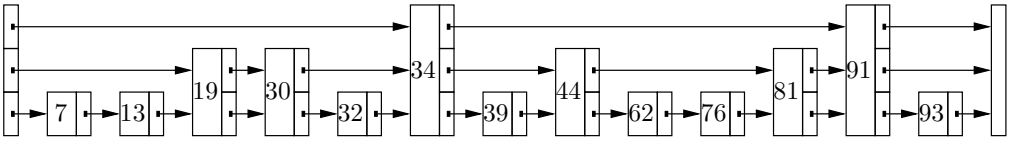


Abb. 3.20: *Beispiel einer Skip-Liste der Höhe 3.*

Höhe des Knotens. Als die *Höhe einer Skip-Liste* bezeichnen wir die maximale Höhe eines Eintrags der Liste (ausgenommen des initialen Eintrags).

Eine Skip-Liste muss folgende Eigenschaft besitzen: Greift man einen Eintrag aus einer Skip-Liste zufällig heraus, so sollte die Wahrscheinlichkeit, auf einen Eintrag mit i Vorwärtszeigern zu treffen, genau $p^{i-1} \cdot (1-p)$ sein, wobei $0 < p < 1$ eine vorher festgelegte Wahrscheinlichkeit ist. Das bedeutet, jeder $\frac{1}{p}$-te Eintrag mit i Vorwärtszeigern hat auch (mindestens) $i+1$ Vorwärtszeiger. Wählt man etwa $p = 1/2$, so hätte durchschnittlich jeder 2. Eintrag zwei Vorwärtszeiger (entspricht der Wahrscheinlichkeit $(1/2)^1$), jeder 4. Eintrag drei Vorwärtszeiger (entspricht der Wahrscheinlichkeit $(1/2)^2$), jeder 8. Eintrag vier Vorwärtszeiger (entspricht der Wahrscheinlichkeit $(1/2)^3$), usw. Die folgende Python-Funktion *randHeight*() erzeugt eine zufällige Höhe für einen neuen Eintrag genau so, dass obige Eigenschaften gelten.

```
1 from random import random
2 p = ...   # feste Wahrscheinlichkeit mit 0 < p < 1
3 def randHeight():
4   i=1
5   while random()≤p: i+=1
6   return min(i,MaxHeight)
```

Listing 3.21: *Die Funktion randHeight() erzeugt mit einer vorher festgelegten Konstanten* $0 < p < 1$ *eine zufällige Höhe.*

Die Funktion *random*() erzeugt normalverteilt (d. h. alle Zahlen sind gleichwahrscheinlich) eine zufällige Gleitpunktzahl zwischen 0 und 1. Aus der Tatsache, dass alle Gleitpunktzahlen gleichwahrscheinlich sind, folgt, dass *random*()$\leq p$ mit Wahrscheinlichkeit

p gilt. Die Wahrscheinlichkeit, dass *randHeight* den Wert 1 zurückliefert ist also $1-p$, die Wahrscheinlichkeit, dass *randHeight* den Wert 2 zurückliefert entspricht der Wahrscheinlichkeit, dass $random() \leq p$ beim ersten Durchlauf und $random() > p$ beim zweiten Durchlauf gilt, was mit einer Wahrscheinlichkeit von $p \cdot (1-p)$ der Fall ist, usw.

3.6.1 Implementierung

Wir definieren eine Klasse *SLEntry*, die einen einzelnen Eintrag in einer Skip-Liste repräsentiert, bestehend aus einem Schlüssel *key*, einem Wert *val* und einer Liste *ptrs* von Vorwärtszeigern.

```
1 class SLEntry(object):
2   def __init__(self, key, ptrs=[], val=None):
3     self.key = key; self.ptrs = ptrs; self.val = val
```

Listing 3.22: *Definition der Klasse SLEntry, die einen Eintrag der Skip-Liste repräsentiert.*

Des Weiteren definieren wir eine Klasse *SkipList*, die eine Skip-Liste repräsentiert.

```
1 class SkipList(object):
2   def __init__(self):
3     self.tail = SLEntry(Infty)
4     self.head = SLEntry(None,[self.tail for _ in range(MaxHeight+1)])
5     self.height = 0
```

Listing 3.23: *Definition der Klasse SkipList, die eine Skip-Liste repräsentiert.*

Eine Skipliste *sl* besitzt ein spezielles Anfangselement *sl.head*, das eine *MaxHeight* lange Liste von Vorwärtszeigern enthält, die anfänglich alle auf das Ende-Element *sl.tail* zeigen. Das spezielle Ende-Element *sl.tail* hat als Schlüssel den Wert „∞“[7] und ist ansonsten leer.

Suche. Am einfachsten ist die Implementierung der Suche. Listing 3.24 zeigt die Implementierung der Suche nach einem Eintrag mit Schlüssel *key*.

```
1 class SkipList(object):
2   ...
3   def search(self, key):
4     x = self.head
5     for i in range(self.height, -1, -1):
6       while x.ptrs[i].key < key: x = x.ptrs[i]
7     x = x.ptrs[0]
8     if x.key == key: return x.val
9     else: return None
```

Listing 3.24: *Implementierung der Suche nach einem Eintrag mit Schlüssel key*

[7] Der Wert „∞“ kann in Python durch den Ausdruck *float*(`'inf'`) erzeugt werden.

Zunächst werden die Vorwärtszeiger auf der höchstmöglichen Stufe, also auf Stufe *self.height*, solange gelaufen, bis der Suchschlüssel kleiner ist als der Schlüssel des momentanen Elements; dies bewirkt die **while**-Schleife in Zeile 6. Anschließend werden die Vorwärtszeiger auf der nächstniedrigeren Stufe entsprechend lange gelaufen, usw. Ist schließlich die unterste Stufe 0 erreicht, so befindet sich die Suche direkt vor dem gesuchten Eintrag – vorausgesetzt natürlich, der Schlüssel *key* befindet sich überhaupt in der Skip-Liste.

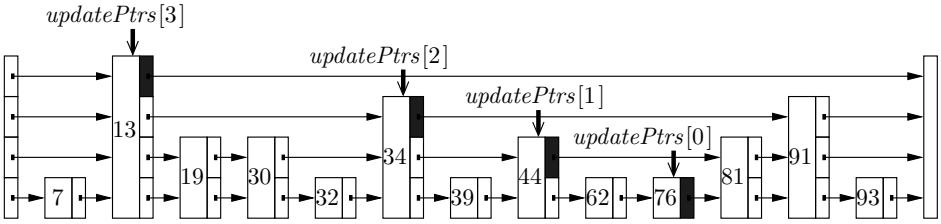
Einfügen. Beim Einfügen eines Elementes in eine Skip-Liste wählen wir die Höhe dieses Elementes durch eine Zufallsentscheidung, die wir schon oben durch die Funktion *randHeigth* implementiert haben. Die Struktur der Skip-Liste ist nicht alleine durch die einzufügenden Elemente determiniert, sondern wird zusätzlich durch Zufallsentscheidungen beim Aufbau der Liste bestimmt. Die Zuweisung in Zeile 13 in Listing 3.25 ist auch tatsächlich das einzige Kommando in den präsentierten Algorithmen über Skip-Listen, das mit Zufallszahlen arbeitet.

```
1  class SkipList( object ):
2    ...
3    def insert( self ,key, val ):
4      updatePtrs = [self.head]*(MaxHeight+1)
5      x = self.head
6      for i in range( self.height, -1, -1):
7        while x.ptrs[i].key < key: x = x.ptrs[i]
8        updatePtrs[i] = x
9      x = x.ptrs[0]
10     if x.key == key: # bestehenden Eintrag verändern
11       x.val = val
12     else:            # neuen Eintrag einfügen
13       newheight = randHeight()
14       self.height = max(self.height, newheight)
15       entry = SLEntry(key,[updatePtrs[i].ptrs[i] for i in range(newheight)],val)
16       for i in range(0,newheight+1):
17         updatePtrs[i].ptrs[i] = entry
```

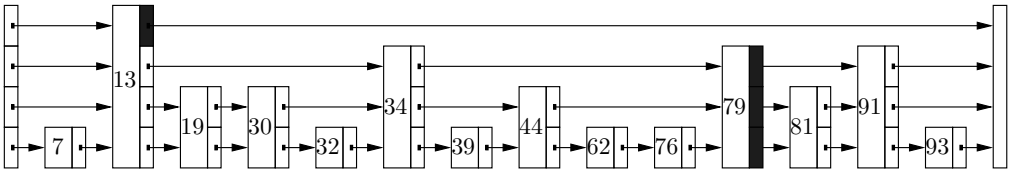
Listing 3.25: *Implementierung der Einfüge-Operation eines Schlüssel-Wert-Paares in eine Skip-Liste*

Bis zur Zeile 12 wird, ähnlich wie in der *search*-Methode, nach der richtigen Einfügeposition gesucht. Zusätzlich werden in der Liste *updatePtrs* diejenigen Elemente der Skip-List gespeichert, deren Vorwärtszeiger bei einem Einfügen möglicherweise angepasst werden müssen; Abbildung 3.21 zeigt dies an einer Beispielsituation; die in *updatePtrs* befindlichen Vorwärtszeiger sind hierbei dunkel markiert. In Zeile 13 und 14 wird durch eine Zufallsentscheidung eine Höhe für das einzufügende Element bestimmt und, falls diese Höhe größer als die bisher maximale Höhe eines Elementes in der Skip-Liste ist, die Höhe der Skip-Liste angepasst. In Zeile 15 wird der neue Eintrag erzeugt. Der i-te Vorwärtszeiger des neuen Eintrags ist hierbei der i-te Vorwärtszeiger des i-ten Eintrags

in *updatePtrs* für $0 \leq i \leq newheight$; dies ist in Abbildung 3.21(b) an der Beispielsituation veranschaulicht. Schließlich werden die Zeiger der in *updatePtrs* befindlichen Elemente so angepasst, dass sie auf den neu erzeugten Eintrag zeigen; dies geschieht in den Zeilen 16 und 17.



(a) Die Situation vor der Einfüge-Operation: In *updatePtrs*[*i*] ist das Element der Skip-Liste gespeichert, dessen *i*-ter Vorwärtszeiger, in der Abbildung grau eingefärbt, beim Einfügen des neuen Elementes angepasst werden muss.



(b) Die Situation nach der Einfüge-Operation: Die *i*-ten Vorwärtszeiger von *updatePtrs*[*i*] für $0 \leq i \leq newheight$ sind jetzt die *i*-ten Vorwärtszeiger des neu eingefügten Elements.

Abb. 3.21: *Einfügen eines neuen Elements mit Schlüsselwert 79 und der (mittels der Funktion randHeight() zufällig erzeugten) Höhe 3 in eine Skip-Liste.*

Löschen. Beim Löschen werden in der Such-Phase ebenfalls diejenigen Elemente gemerkt, deren *i*-ter Vorwärtszeiger eventuell angepasst werden muss. Listing 3.26 zeigt die Implementierung der Lösch-Funktion.

```
1  class SkipList( object ):
2    ...
3    def delete( self ,key):
4      updatePtrs = [self.head]*(MaxHeight+1)
5      x = self.head
6      for i in range( self.height, -1, -1):
7        while x.ptrs[i].key < key: x = x.ptrs[i]
8        updatePtrs[i] = x
9      x = x.ptrs[0]       # x ist das zu löschende Element
10     if x.key == key:
11       heightx = len(x.ptrs)-1
12       for i in range(0,heightx+1):
13         updatePtrs[i].ptrs[i] = x.ptrs[i]
14       while self.height >= 0 and self.head.ptrs[ self.height] == self.tail:
15         self.height -= 1
```

Listing 3.26: *Implementierung der Lösch-Funktion.*

Die Methode *delete* funktioniert sehr ähnlich wie die Methode *insert*. Einer Erwähnung Wert sind allenfalls die Zeilen 14 und 15, in der die Höhe des Skip-Liste genau dann angepasst wird, wenn das Element mit der maximalen Höhe gelöscht wurde. Hierbei genügt es nicht, die Höhe einfach um Eins zu erniedrigen, denn der Höhenunterschied zum nächst tieferen Element könnte mehr als Eins betragen. Stattdessen muss das *head*-Element der Skip-Liste untersucht werden: Der höchstgelegene Zeiger, der *nicht* auf das *tail*-Element zeigt, ist die neue Höhe der Skip-Liste.

Aufgabe 3.46

Implementieren Sie die Funktion *__str__*, so dass Skip-Listen folgendermaßen ausgegeben werden:

```
>>> print skiplist
>>> [ (30|1), (33|4), (40|3), (77|1), (98|1), (109|1), (193|3) ]
```

Ausgegeben werden soll also der Schlüssel jedes Elements zusammen mit der Höhe des Elements.

Aufgabe 3.47

(a) Schreiben Sie eine Methode *keys*(), die eine Liste der in der Skip-Liste gespeicherten Schlüsselwerte zurückliefert.

(b) Schreiben Sie eine Methode *vals*(), die eine Liste der in der Skip-Liste gespeicherten Werte zurückliefert.

Aufgabe 3.48

Oft wird eine effiziente Bestimmung der Länge einer Skip-Liste benötigt. Erweitern Sie die Klasse *SkipList* um ein Attribut *length*, passen Sie entsprechend die Methoden *insert* und *delete* an und geben Sie eine Implementierung der Methode *__len__* an, so dass die *len*-Funktion auf Skip-Listen anwendbar ist.

Aufgabe 3.49

(a) Schreiben Sie eine Funktion *numHeights*(h), die die Anzahl der Elemente mit Höhe n zurückliefert.

(b) Schreiben Sie eine Funktion *avgHeight*(s), die die durchschnittliche Höhe eines Elementes der Skip-Liste s berechnet.

3.6.2 Laufzeit

Für alle Operationen auf einer Skip-Liste dominiert immer die Laufzeit der Suche nach der richtigen Einfüge- bzw. Löschposition. Es genügt also, wenn wir uns bei der Laufzeitanalyse auf die Untersuchung der Laufzeit der Suche in einer Skip-Liste beschränken.

Die erwartete Höhe einer Skip-Liste. Wir werden im Folgenden sehen, dass die Höhe einer Skip-Liste entscheidend für die Laufzeit der Suche ist. Da Skip-Listen eine randomisierte Datenstruktur darstellen, ist die Höhe einer Skip-Liste keine vorherbestimmte Größe. Mathematisch lässt sich die Höhe als Zufallsvariable[8] $H(n)$ modellieren.

Wir können also nie mit Sicherheit vorhersagen, welche Höhe eine bestimmte Skip-Liste haben wird. Wir beschränken uns daher darauf, zu fragen, was die erwartete[9] Höhe $H(n)$ einer Skip-Liste mit n Elementen ist. Wir führen hierzu den Ausdruck $numHeights(h)$ ein, der die durchschnittliche Anzahl von Elementen einer n-elementigen Skip-Liste repräsentiert, die eine Höhe von $\geq h$ haben. Es gilt:

$$numHeights(0) = n \cdot p^0, \quad numHeights(1) = n \cdot p^1, \quad numHeights(2) = n \cdot p^2, \quad \ldots$$

Wir setzen dies nun soweit fort, bis wir eine Höhe h gefunden haben, für die es durchschnittlich weniger als ein Element mit dieser Höhe gibt:

$$numHeights(\log_{1/p}(n) + 1) = n \cdot p^{\log_{1/p}(n)} \cdot p = \frac{n}{n} \cdot p = p < 1$$

Aus der Tatsache, dass es durchschnittlich weniger als ein Element in der Skip-Liste gibt, das eine Höhe von mindestens $\log_{1/p}(n) + 1$ aufweist, können wir schließen, dass $\log_{1/p}(n)$ die Höhe (d. h. die maximale Höhe eines Elementes der Skip-Liste) einer durchschnittlichen Skip-Liste mit n Elementen ist, also:

$$H(n) \approx \log_{1/p}(n) + 1$$

Für die häufig gewählte Wahrscheinlichkeit $p = \frac{1}{2}$ gilt $H(n) \approx \log_2(n) + 1$.

Aufgabe 3.50

(a) Schreiben Sie eine Methode *numHeights*(*h*) der Klasse *SkipList*, die von einer gegebenen Skipliste die Anzahl der Elemente mit Höhe n zurückliefert.

(b) Schreiben Sie eine Funktion *avgHeight*() der Klasse *SkipList*, die die durchschnittliche Höhe eines Elementes der Skip-Liste berechnet.

Die erwartete Länge eines Suchpfades. Die entscheidende Idee, die durchschnittliche Länge eines Suchpfades zu ermitteln, besteht darin, den Suchpfad rückwärts zu betrachten. Wir starten beginnend vom Zeiger der Höhe 0, der direkt auf das gesuchte

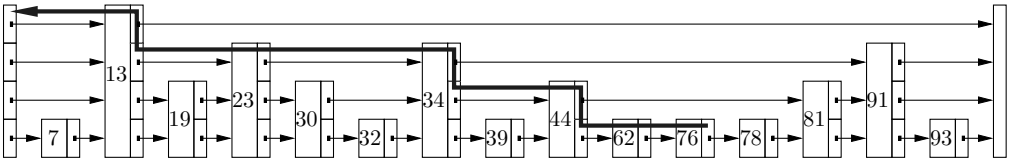


Abb. 3.22: *Darstellung eines Suchpfades. Wie man sieht, gibt es für jeden Schritt immer zwei mögliche „Richtungen“ um sich auf dem Pfad vom zu suchenden Element hin zum Listenkopf zu bewegen: entweder horizontal oder nach oben. Die Frage die bleibt ist: Wie viele der horizontalen Schritte müssen durchschnittlich gegangen werden, bis man „oben“ ankommt?*

Element zeigt und laufen *rückwärts* bis wir am Kopf der Skip-Liste angelangt sind. Abbildung 3.22 zeigt solch einen „Rückwärts“-Pfad ausgehend vom zu suchenden Element, in diesem Fall „76“, zum Kopf der Liste.

Befindet man sich bei einem Element der Höhe i so geht der Rückwärts-Pfad solange in vertikaler Richtung weiter, bis er auf ein Element der Skip-Liste stößt, das eine Höhe von (mindestens) $i+1$ besitzt. Dann geht der Pfad um eine Stufe nach oben, usw. Es ist hilfreich, sich folgenden (fiktiven) Algorithmus vorzustellen, der diesen Rückwärts-Pfad durchläuft:

```
x = zuSuchendesElement
while x ≠ self.head
  if nachOben(x) moeglich:
    x = nachObenWeiter(x)
  else:
    x = vertikalWeiter(x)
```

Listing 3.27: *Fiktiver Algorithmus zum Durchlaufen des Rückwärts-Pfades*

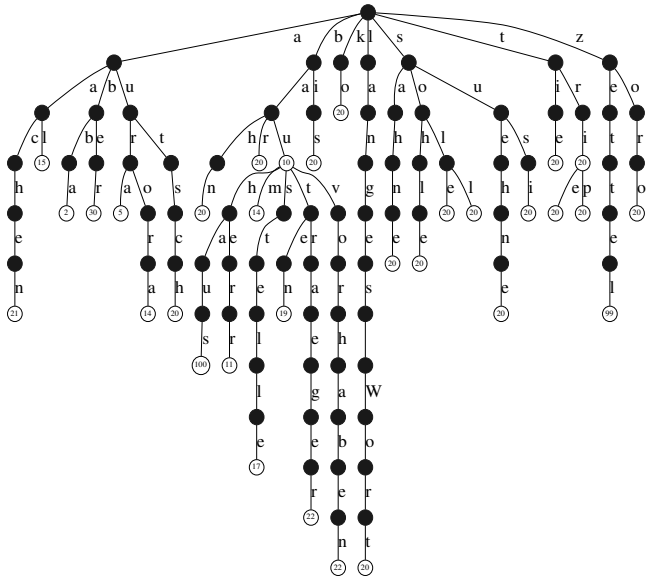
Wie oben beschrieben, wird in diesem Algorithmus in jedem **while**-Schleifendurchlauf das Kommando *nachObenWeiter*(x) mit Wahrscheinlichkeit p ausgeführt und das Kommando *vertikalWeiter*(x) mit Wahrscheinlichkeit $1-p$. Die entscheidende Frage ist: Wie viele *vertikalWeiter*(x)-Schritte werden (durchschnittlich) gegangen, bis der Pfad sich eine Ebene nach oben bewegt? Bezeichnen wir als X die Zufallsvariable, die die Anzahl der *verticalWeiter*(x)-Schritte angibt, bis die Höhe ansteigt. Diese Zufallsvariable ist geometrisch verteilt mit $E[X] = 1/p$. Wir wissen, dass auf dem Rückwärtspfad sich die Höhe $H(n)$-mal erhöhen muss bis das *head*-Element erreicht ist. Die Laufzeit der Suche beträgt also

$$\frac{1}{p} \cdot H(n) \approx \frac{1}{p} \cdot \log_{1/p} n + 1 = O(\log n)$$

[8] Für weitere Details zu Zufallsvariablen siehe Abschnitt B.3.3.

[9] Gefragt ist hier also der *Erwartungswert*. Der Erwartungswert $H(n)$ der Höhe ist nichts anderes als die durchschnittliche Höhe einer Skip-Liste, gemittelt über sehr viele Skip-Listen.

3.7 Tries



Tries sind Bäume, deren Kanten mit Buchstaben (bzw. sequentialisierten Teilen der Suchschlüssel) beschriftet sind. Besonders dann, wenn Suchschlüssel aus langen Zeichenketten bestehen sind Tries vielen anderen in diesem Kapitel vorgestellten Datenstrukturen überlegen, sowohl was die Suchzeit als auch was die Speichereffizienz betrifft.
Besonders interessant ist, dass die Laufzeit zur Suche eines Schlüssels *nicht* von der Gesamtzahl der Einträge im Trie abhängt, sondern alleine von der Länge des Suchschlüssels.
Die Suche nach einem kurzen String benötigt in einem Trie also immer die gleiche Anzahl (weniger) Schritte, unabhängig davon, ob sich im Trie 1000, 100 000 oder mehrere Millarden Einträge befinden.
Daher werden Tries und Trie-ähnliche Datenstrukturen sehr häufig bei der Implementierung von (Text-)Suchmaschinen eingesetzt. Außerdem werden sie oft verwendet, um effiziente Lookups in Routing-Tabellen zu implementieren, die beispielsweise für die Funktionsweise des Internets unerlässlich sind.

3.7.1 Die Datenstruktur

Binäre Suchbäume bewähren sich in der Praxis genau dann sehr gut, wenn sich Schlüssel effizient vergleichen lassen. Dies ist im Allgemeinen dann der Fall, wenn Werte eines „einfachen“ Typs verglichen werden, wie etwa Integer-Werte oder einzelne Zeichen; jedoch kann ein Vergleich „teuer“ werden, wenn Werte komplexerer zusammengesetzter Typen verglichen werden, wie etwa (möglicherweise lange) Zeichenketten; aber selbst einfache Vergleiche können in objektorientierten Sprachen verhältnismäßig teuer sein, da der Vergleichsoperator üblicherweise überladen ist und, bevor der eigentliche Vergleich ausgeführt wird, zunächst die für die verwendeten Typen passende Methode dynamisch (also während der Laufzeit) ausgewählt werden muss. Dieser sog. *dynamic dispatch* ist verhältnismäßig rechenaufwändig.

Handelt es sich bei den Schlüsselwerten also um komplexe Werte etwa eines zusammengesetzten Typs, insbesondere um Strings, so ist die sog. *Trie*[10]-Datenstruktur, ei-

[10] Der Name „Trie“ leitet sich ab aus dem englischen Wort „retrieval“, dem Finden bzw. Wiederfinden von Informationen.

ne Baumstruktur, oft die beste Wahl Schlüssel-Wert-Paare effizient zu speichern und wieder zu finden. Anders als bei Suchbäumen, sind in den Knoten eines Tries nicht die Schlüssel selbst gespeichert, sondern die Position des Knotens innerhalb des Trie-Baums bestimmt, welcher Schlüssel im Knoten gespeichert ist. Angenommen die Strings, aus denen die Schlüsselwerte bestehen, setzen sich zusammen aus Kleinbuchstaben zwischen a und z. Dann sind alle Kinder $v.children$ eines Knoten v im Trie markiert mit einem Element aus $\{a,\ldots,z\}$. Den zu einem Schlüsselwert passenden Eintrag in einem Trie findet man nun einfach dadurch, dass man von der Wurzel beginnend den mit den Zeichen im String markierten Kanten nachläuft. Abbildung 3.23 zeigt einen einfachen Trie, der die Schlüsselwerte `'bahn'`, `'bar'`, `'bis'`, `'sole'`, `'soll'`, `'tri'`, `'trie'` und `'trip'` speichert.

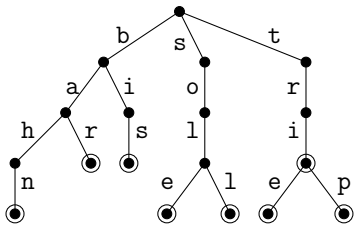


Abb. 3.23: *Beispiel eines Tries, der die Strings* `'bahn'`, `'bar'`, `'bis'`, `'sole'`, `'soll'`, `'tri'`, `'trie'` *und* `'trip'` *speichert. Nur die Knoten mit doppelter Umrandung entsprechen einem enthaltenen Schlüssel und können tatsächlich einen Wert speichern.*

Aufgabe 3.51

Zeichnen Sie einen Trie, der die Schlüssel `gans`, `ganz`, `galle`, `leber`, `lesen`, `lesezeichen`, `zeichnen`, `zeilenweise`, `adam`, `aaron` speichert und beantworten Sie die folgenden Fragen:

(a) Wie viele Schritte benötigt eine Suche in diesem Trie minimal?

(b) Wie viele Schritte benötigt eine Suche in diesem Trie maximal?

Aufgabe 3.52

Beantworten Sie die folgenden Fragen:

(a) Wie viele Character-Vergleiche benötigt eine Suche in einem Trie höchstens, der 1 Mio verschiedene Schlüsselwerte mit einer Länge von höchstens 14 enthält?

(b) Wie viele Character-Vergleiche benötigt eine Suche in einem binären ausgeglichenen Suchbaum, der 1 Mio verschiedene Schlüsselwerte mit einer Länge von höchstens 14 enthält?

Listing 3.28 zeigt die Definition der Python-Klasse *Trie*:

```
1 class Trie( object ):
2   def __init__ ( self ):
3     self . children = {}
4     self . val = None
```

Listing 3.28: *Klasse Trie mit der __init__ -Methode*

Jeder Trie t enthält also ein Attribut $t.val$, das die im jeweiligen Knoten befindliche Information speichert, und ein Attribut $t.children$, das die Menge der Kinder des Knotens speichert. Diese Kinder-Menge wird in Listing 3.28 als *dict*-Wert repräsentiert, der Kantenmarkierungen auf Kinder-Tries abbildet. Es wäre wohl auch die Repräsentation als Liste denkbar, die aus Kantenmarkierungen und Kinder-Tries bestehende Tupel enthält, jedoch erweist sich die Verwendung eines *dict*-Wertes als effizientere Wahl.

3.7.2 Suche

Die beiden wichtigsten Operationen auf einen Trie sind das Einfügen und das Suchen. Beginnen wir, da einfacher, mit der Implementierung der Suche. Listing 3.29 zeigt eine rekursive Implementierung der Methode *search*:

```
1 class Trie( object ):
2   ...
3   def search( self ,key):
4     if not key: return self. val
5     c = key[0]
6     if c not in self . children : return None
7     return self. children [c] . search( key [1:])
```

Listing 3.29: *Rekursive Implementierung der Suche in einem Trie.*

Zeile 4 in Listing 3.29 implementiert den Rekursionsabbruch, der dann eintritt, wenn der Suchstring *key* leer ist. In diesem Fall gehen wir davon aus, dass die Suche am Ziel angelangt ist und liefern daher als Rückgabewert den im momentanen Knoten gespeicherten Wert *self . val* zurück. Andernfalls versuchen wir dem Zweig nachzulaufen, der mit dem ersten im Suchschlüssel gespeicherten Zeichen, also mit $key[0]$ bzw. c, markiert ist. Falls kein solcher Zweig vorhanden ist, d. h. falls c nicht im Dictionary *self . children* enthalten ist, wird einfach *None* zurückgeliefert. Andernfalls fahren wir in Zeile 8 rekursiv mit der Suche des verbleibenden Suchschlüssels $key[1:]$ fort, solange eben, bis der Suchschlüssel leer ist.

3.7.3 Einfügen

Nur wenig schwieriger ist die in Listing 3.30 gezeigte Implementierung der Einfüge-Operation:

```
1 class Trie( object ):
2   ...
3   def insert( self ,key, val ):
4     if not key:
5       self . val=val
6     else:
7       if key[0] not in self . children:
8         self . children [key[0]] = Trie()
9       self . children [key[0]] . insert(key[1:], val)
```

Listing 3.30: *Rekursive Implementierung einer Funktion insert, die ein neues Schlüssel-Wert-Paar in einen Trie einfügt.*

Ist der Schlüsselstring leer, so wurde bereits an die passende Stelle des Tries navigiert und der als Parameter übergebene Wert *val* kann eingefügt werden – dies geschieht in Zeile 5. Andernfalls wird, wie bei der Suche auch, das nächste Zeichen des Schlüsselstrings (also $key[0]$) dazu benutzt um sich der passenden Stelle im Trie weiter zu nähern – dies geschieht in Zeile 9 in Listing 3.30: Gibt es noch keinen $key[0]$-Eintrag im *children*-Dictionary, so wird ein solcher Eintrag erzeugt. Anderfalls wird dem $key[0]$-Eintrag des im aktuellen Knoten gespeicherten *children*-Dictionaries gefolgt und für den dort gespeicherten Trie die *insert*-Methode rekursiv mit dem restlichen Schlüssel $key[1:]$ aufgerufen.

Aufgabe 3.53

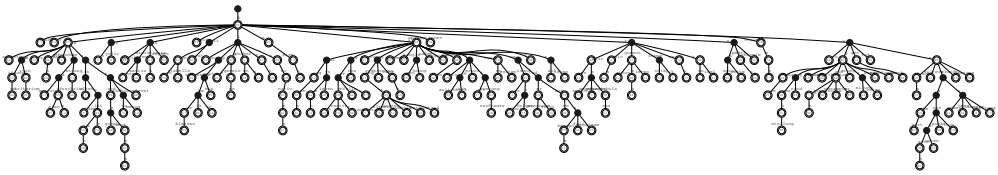
Implementieren Sie eine Methode *keys*(), die eine Liste aller in einem Trie befindlichen Schlüsselwerte zurückliefert.

Dass die vorgestellte Trie-Datenstruktur sehr effizient ist, zeigt ein (natürlich sehr Rechner-abängiger) Laufzeit-Vergleich für das Suchen von 1000 Wörter der Länge 100 einmal in einem Trie und einmal in einer Instanz des Python-internen *dict*-Typs:

Implementierung	Laufzeit
dict-Typ	0.348
Trie-Typ	0.353

Man beachte jedoch: Die *dict*-Implementierung ist dadurch ungleich bevorteilt, dass sie direkt in C implementiert. Die Tatsache, dass die Laufzeiten der beiden Implementierungen trotzdem in etwa gleich sind, deutet darauf hin, dass die Trie-Struktur für diesen Anwendungsfall prinzipiell die effizientere Methode ist.

3.8 Patricia-Tries



Ein Patricia (auch häufig als Patricia-Trie bezeichnet) ist einem Trie sehr ähnlich, nur dass ein Patricia auf eine kompaktere Darstellung Wert legt. Dies geht zwar etwas auf Kosten der Laufzeit – die Einfügeoperation und die Löschoperation werden etwas langsamer und die Implementierung komplexer. In vielen Fällen werden diese Nachteile aber wenig ins Gewicht fallen, und der Vorteil der kompakteren Speicherung überwiegt. Oben dargestellter Patricia speichert etwa die lexikographisch ersten 200 in diesem Buch vorkommenden Wörter.

3.8.1 Datenstruktur

Es gibt den einen problematischen Fall, dass sich viele Wörter in einem Trie (bzw. in einem Teilbaum des Tries) befinden, die sich einen langen gemeinsamen Präfix teilen, d. h. die alle mit der gleichen Buchstabenkombination beginnen. In diesem Fall „beginnt“ der Baum mit einer langen Kette von Knoten, wobei jeder Knoten jeweils nur ein Kind hat. Abbildung 3.24(a) zeigt einen Trie, dessen Einträge alle den Präfix `'bau'` haben. Patricia-Tries stellen eine Optimierung der Tries dar. Man kann nämlich Knoten mit Grad 1 (also mit nur einem Kind) in denen sich keine Informationen befinden mit dem jeweiligen Kind-Knoten verschmelzen und so eine kompaktere Darstellung eines Tries erhalten. Die verbleibenden Knoten speichern dann den gemeinsamen Präfix aller im entsprechenden Teilbaum befindlichen Knoten. Abbildung 3.24(b) zeigt ein Beispiel eines Patricia-Trie:

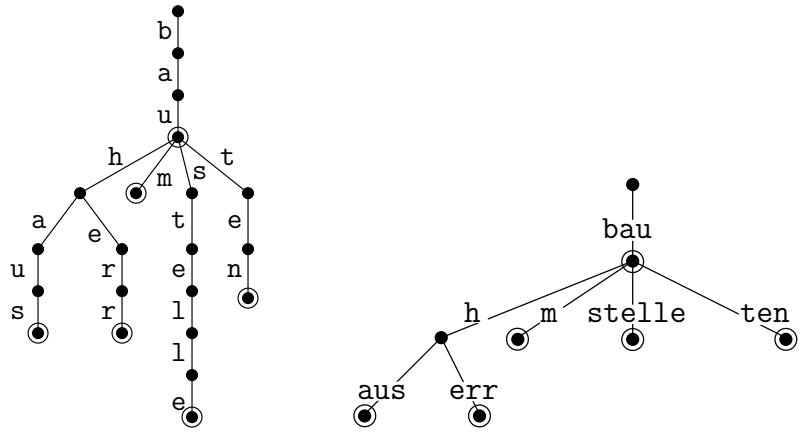
Aufgabe 3.54

Fügen Sie in den Patricia-Trie aus Abbildung 3.24(b) die Schlüsselwerte `baustellplatz` und `bauträger` ein.

Wir implementieren Patricia-Tries als Klasse *Patricia*. Die Konstruktor-Funktion *__init__* ist mit der Konstruktorfunktion der Klasse *Trie* identisch.

```
1 class Patricia( object ):
2   def __init__ ( self ):
3       self . children = {}
4       self . val = None
```

Listing 3.31: *Klassendefinition Patricia*



(a) Ein Trie. **(b)** Ein Patricia-Trie.

Abb. 3.24: *Ein Trie und ein Patricia-Trie, die die jeweils gleichen Schlüsselwerte gespeichert haben, nämlich* `'bau'`, `'bauhaus'`, `'bauherr'`, `'baum'`, `'baustelle'`, `'bauten'`. *Jeder Knoten des Patricia-Trie hält den Präfix gespeichert, den alle in seinem Teilbaum gespeicherten Schlüsselwerte gemeinsam haben.*

3.8.2 Suche

Wie man leicht sieht, ist sowohl das Einfügen, *insert*, als auch das Suchen, *search*, im Falle der Patricia-Tries komplizierter zu implementieren als im Falle der Tries. Das liegt daran, dass nun nicht mehr sofort klar ist, welchen Zweig man eigentlich zu laufen hat – man muss nach passenden Zweigen erst suchen. Listing 3.32 zeigt die Implementierung der Suchfunktion.

```
1  class Patricia( object ):
2    ...
3    def search( self , key):
4      if not key:
5        return self. val
6      prefixes = [k for k in self . children if key. startswith (k) ]
7      if not prefixes :
8        return None
9      else:
10       prefix = prefixes [0]
11       return self. children [ prefix ]. search( key[len( prefix ): ])
```

Listing 3.32: *Implementierung der Suchfunktion für Patricias.*

Man beachte zunächst, dass die Implementierung rekursiv ist. Der Rekursionsabbruch erfolgt, wenn der zu suchende Schlüsselwert *key* der leere String ist, also **not** *key* gilt. In diesem Fall gehen wir davon aus, dass der gesuchte Knoten des Patricia-Tries erreicht wurde und geben einfach den darin gespeicherten Wert *self . val* zurück – dies geschieht

in Zeile 5. Andernfalls suchen wir in *self.children* nach einem Schlüsselwert, der ein Präfix von *key* ist. Gibt es kein solches Attribut (das ist der Fall, wenn *prefixes*==[] bzw. **not** *prefixes*), so gilt der Schlüsselwert *key* als nicht gefunden, die Suche wird abgebrochen und *None* zurückgeliefert – dies geschieht in Zeile 8. Andernfalls wird der mit dem gefundenen Schlüsselwert beschrifteten Kante *self.children[prefix]* nachgelaufen und die Suchprozedur mit entsprechend verkürztem Schlüssel *key[len(prefix):]* rekursiv aufgerufen – dies geschieht in Zeile 11.

3.8.3 Einfügen

Insbesondere die Implementierung der Einfügeoperation ist für Patricias komplexer als für einfache Tries. Listing 3.33 zeigt die Implementierung eines Patricia-Tries.

```
1  class Patricia(object):
2      ...
3    def insert(self,key,val):
4      v = self
5      prefx = [k for k in v.children.keys() if k.startswith(key[0])] if key else []
6      if prefx != []:
7        prefx = prefx[0]
8        if not key.startswith(prefx): # Fall 3 ⇒ umstrukturieren
9          i = prefixLen(key, prefx)
10         t1 = v.children[prefx]
11         del(v.children[prefx])
12         v.children[key[:i]] = Patricia()
13         v.children[key[:i]].children[prefx[i:]] = t1
14         if key[i:]==[]:
15           v.children[key[:i]].val = val
16           return
17         v.children[key[:i]].children[key[i:]] = Patricia()
18         v.children[key[:i]].children[key[i:]].val = val
19       else: # Fall 2 ⇒ einfach weiterlaufen
20         key = key[len(prefx):]
21         if key==[]:
22           v.val = val
23           return
24         v = v.children[prefx]
25         v.insert(key, val)
26     else: # Fall 1 ⇒ neuen Eintrag generieren
27       v.children[key] = Patricia()
28       v.children[key].val = val
```

Listing 3.33: *Implementierung eines Patricia-Trie*

Auch die Implementierung von *insert* ist rekursiv; der rekursive Aufruf ist in Zeile 25 in Listing 3.33 zu sehen. In jedem Aufruf von *insert* auf einen Knoten v sind drei Fälle zu unterscheiden:

1. Fall: Es gibt keinen Eintrag in $v.\,children$ dessen Schlüssel einen mit *key* gemeinsamen Präfix hat. Dies ist der einfachste Fall. Es muss lediglich ein neuer Eintrag in $v.\,children$ erzeugt werden mit Schlüssel *key* dessen Wert *val* ist. Dieser Fall wird in den Zeilen 27 und 28 aus Listing 3.33 behandelt.

2. Fall: Es gibt in $v.\,children$ einen Eintrag *commonPrae* der ein Präfix von *key* ist d. h. für den gilt, dass $commonPrae == key[:i]$ (mit $i = len(commonPrae)$). In diesem Fall muss einfach dieser mit $key[:i]$ markierten Kante nachgelaufen werden und anschließend mit dem verbleibenden Suffix von *key* weitergesucht werden. Dieser Fall wird in den Zeilen 20 bis 25 in Listing 3.33 behandelt.

3. Fall: Es gibt in $v.\,children$ einen Eintrag *prefx*, der zwar kein vollständiger Präfix von *key* ist; jedoch haben *prefx* und *key* einen *gemeinsamen* Präfix, d. h. es gibt ein $0 < i < len(prefx)$ mit $prefx[:i] == key[:i]$. Dies ist der aufwändigste Fall, denn hier muss der bisherige Patricia umgebaut werden. Die Beschriftung *prefx* muss zunächst zu $prefx[:i]$ verkürzt werden. An dem durch $prefx[:i]$ erreichten Knoten werden zwei Zweige erzeugt. Der eine wird mit $prefx[i:]$ beschriftet und enthält die Informationen, die auch vorher unter dem Schlüssel *prefix* erreichbar waren – also den Teilbaum t_1. Der andere Zweig wird mit $key[i:]$ beschriftet und enthält den Wert zum neu eingefügten Schlüssel *key*.

   Abbildung 3.25 zeigt nochmals bildlich, was zu tun ist und was entsprechend auch in Listing 3.33 zwischen den Zeilen 9 und 18 implementiert ist.

Aufgabe 3.55

Beantworten Sie die folgenden beiden Fragen bzgl. der Suche nach allen in *children* enthaltenen Schlüsselwerten, die ein Präfix von *key* sind:

(a) In Zeile 5 in Listing 3.33 gezeigten Listenkomprehensionen werden alle Schlüsseleinträge im Kantendictionary *children* gesucht, die mit dem Anfangsbuchstaben des Schlüssels, also mit $key[0]$, beginnen. Argumentieren Sie, warum sich darunter die gesuchten Schlüsseleinträge befinden müssen.

(b) Argumentieren Sie, warum die in Zeilen 5 in Listing 3.33 und in Zeile 6 in Listing 3.32 verwendeten Listenkomprehensionen entweder leer oder einelementig sein müssen.

Der Vorteil des Patricia-Tries gegenüber der einfachen Trie-Datenstruktur besteht aber darin, dass eine kompaktere Repräsentation möglich wird; die Geschwindigkeit leidet darunter, jedoch nur geringfügig, wie folgende Tabelle zeigt:

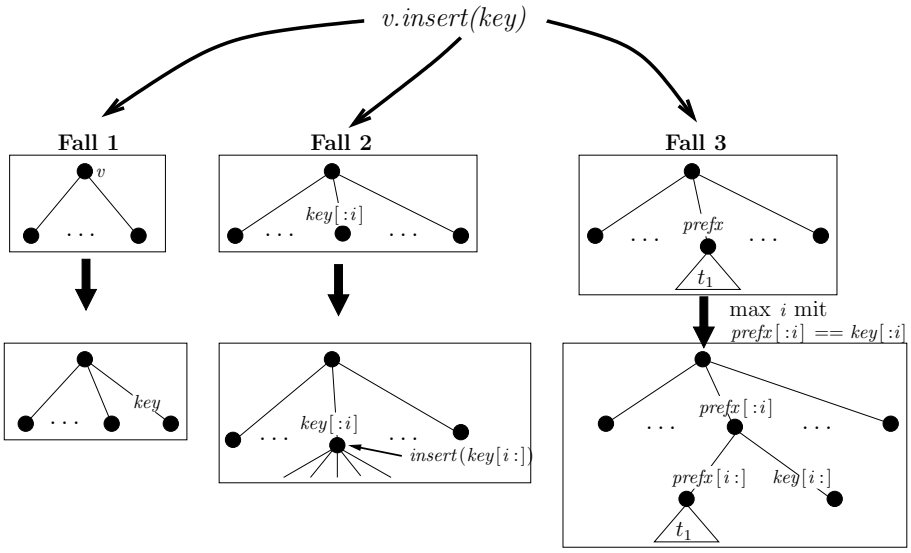


Abb. 3.25: *Grafische Darstellung der drei verschiedenen Fälle die beim Einfügen in einen Patricia-Trie zu unterscheiden sind.*

Implementierung	Laufzeit
`dict`-Typ	0.348
`Patricia`-Typ	0.3924

3.9 Suchmaschinen

Suchmaschinen verwenden Methoden des Information Retrieval, einem Forschungsgebiet mit mittlerweile langer Tradition, das sich allgemein mit der Wiedergewinnung (engl: „Re-Trieval“) von Informationen beschäftigt, die in großen Datenbeständen für den Benutzer ansonsten praktisch „verloren“ wären. Wir beschäftigen uns hier jedoch nur mit einem sehr kleinen Teil des Information Retrieval, mit rein lexikalischen (also rein textbasierten) Suchtechniken. Viele Suchmaschinen verwenden darüberhinaus semantische Suchtechniken, die Informationen aus verschiedenen Wissensbereichen mit einfließen lassen und mit Hilfe dieser Zusatzinformationen das Suchen effektiver gestalten können.

Eine für das Programmieren von Suchmaschinen sehr nützliche Datenstruktur ist der *Trie* und dessen Verfeinerung, der *Patricia*.

3.9.1 Aufbau einer Suchmaschine

Abbildung 3.26 zeigt den typischen Aufbau einer Suchmaschine. Der *Crawler* läuft hierbei über die Dokumentenbasis, der *Indexer* parst die Dokumente und extrahiert die zu indizierenden Elemente, i. A. Wörter oder Phrasen, und die Suchanfrage-Bearbeitung

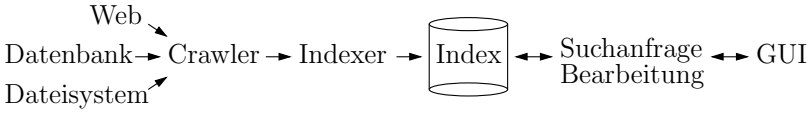


Abb. 3.26: *Typischer Aufbau einer Suchmaschine.*

extrahiert die für die Anfrage notwendigen Daten aus der Indexstruktur.

In realen Suchmaschinen können die einzelnen Teile sehr komplex werden: oft arbeitet der Crawler über verschiedene Rechner verteilt. Der Indexer muss möglichst viele Dokumente erkennen können und womöglich in der Lage sein, die Dokumentenstruktur (also: was ist Überschrift, was ist einfacher Text, ...) erkennen können usw. Außerdem muss er ein sog. Stemming betreiben, d. h. nur die Wortstämme sollten berücksichtigt werden und nicht etwa für Akkusativ, Dativ oder Mehrzahl verschiedene Index-Einträge des eigentlich gleichen Wortes erzeugt werden.

3.9.2 Invertierter Index

Der sog. *invertierte Index* bildet das „Herz“ jeder Suchmaschine; diese Datenstruktur ermöglicht das schnelle Finden von Wörtern und Suchbegriffen. Dieser Index ordnet jedem Wort von Interesse Informationen über dessen Position in der Dokumentenbasis zu. Oft wird hierbei jedem Wort aus dem Index die Liste aller Vorkommen dieses Wortes (i. A. ist dies eine Liste von Dokumenten) zugeordnet. Jedes Dokument, auf das hierbei referenziert wird, besitzt innerhalb des Systems eine eindeutige Identifikationsnummer. Jedes dieser Vorkommen ihrerseits könnte wiederum eine Liste von Positionen innerhalb des Dokuments referenzieren, in denen das Wort auftaucht. Abbildung 3.27 zeigt die Struktur eines solchen invertierten Indexes nochmals graphisch.

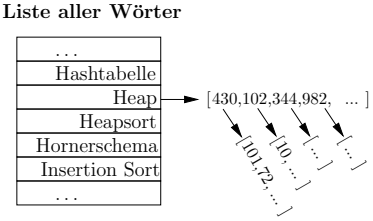


Abb. 3.27: *Darstellung des Invertierten Indexes*

3.9.3 Implementierung

Es gibt mehrere Möglichkeiten, den Index in Python zu implementieren. Wir verwenden der Einfachheit halber hier Pythons *dict*-Typ, um den invertierten Index zu implementieren. Jedes Wort w des Indexes stellt hierbei einen Schlüssel des *dict*-Objekts *ind* dar – dies ist in Listing 3.34 zu sehen. In einem Eintrag $ind[w]$ werden nun alle Dokumente gespeichert, in denen das Wort w auftaucht. Wir wollen uns zusätzlich auch noch alle

Positionen innerhalb eines Dokuments merken. Diese könnten wir prinzipiell als Liste in $ind[w]$ hinterlegen. Wir wollen jedoch zusätzlich für jedes Dokument uns alle Positionen, in unserem Falle zunächst nur Zeilennummern, innerhalb des Dokuments merken, in denen das entsprechende Wort vorkommt. Folglich ist es am günstigsten als Einträge in $ind[w]$ wiederum *dict*-Objekte zu wählen, die jedem Dokument in dem w vorkommt, die relevanten Positionen innerhalb des Dokuments zuordnen.

```
1  import os
2
3  class Index(object):
4    def __init__(self, path=''):
5      self.docId = 0
6      self.ind = {}
7      self.docInd = {}
8      if path≠'0': self.crawl(path)
9
10   def toIndex(self, (word,pos), docId):
11     if word not in self.ind:
12       self.ind[word] = { docId : [pos] }
13     elif docId not in self.ind[word]:
14       self.ind[word][docId] = [pos]
15     else:
16       self.ind[word][docId].append(pos)
17
18   def addFile(self, file, tmp=''):
19     def tupl(x,y): return (x,y)
20     if tmp=='': tmp=file
21     print "Adding", file
22     self.docInd[self.docId] = file
23     fileHandle = open(tmp) ; fileCont = fileHandle.readlines() ; fileHandle.close()
24     fileCont = map(tupl, xrange(0,len(fileCont)), fileCont)
25     words = [(word.lower(),pos) for (pos, line) in fileCont
26                                 for word in line.split()
27                                 if len(word) ≥3 and word.isalpha() ]
28     for word,pos in words:
29       self.toIndex((word,pos), self.docId)
30     self.docId+=1
31
32   def crawl(self, path):
33     for dirpath, dirnames, filenames in os.walk(path):
34       for file in filenames:
35         f = os.path.join(dirpath, file)
36         if isPdf(f):
37           tmpFile = os.path.join(dirpath, 'tmp.txt')
38           os.popen('pdftotext \'' +f+'\' ' +'\''+tmpFile+'\'')
```

```
39              self.addFile(f,tmpFile)
40              os.popen('rm \'' + tmpFile + '\'') # und wieder loeschen ...
41            if isTxt(f):
42              self.addFile(f)
43
44    def ask(self, s):
45      if s in self.ind:
46        return [self.docInd[d] for d in self.ind[s].keys()]
47      else: return []
```

Listing 3.34: *Die Klasse Index implementiert eine sehr einfache Suchmaschine unter Verwendung von Dictionaries*

Das „Herz“ der Implementierung stellt die Funktion *toIndex* dar, die ein Wort dem Index hinzufügt. Jeder Eintrag des Indexes enthält zum Einen Informationen, in welchen Dokumenten das entsprechende Wort vorkommt und zum Anderen enthält es Informationen an welchen Positionen im jeweiligen Dokument es vorkommt; dies entspricht genau dem in Abbildung 3.27 dargestellten doppelt invertierten Index. Beim Einfügen eines Wortes *word* in den Index sind die folgenden drei Fälle zu beachten: 1. Es gibt noch keinen Eintrag *word*; dann muss zunächst ein neues Dictionary angelegt werden mit einem Eintrag. 2. Es gibt schon einen Eintrag *word*, jedoch gibt es noch keinen *docId*-Eintrag für *word*; dann muss ein neuer Eintrag *docId* in *ind*[*word*] angelegt werden mit einem Positionseintrag. 3. Es gibt schon einen Eintrag *word* und für *word* einen Eintrag *docId*; dann muss die neue Positionsinformation an die Liste der schon bestehenden Positionen angehängt werden.

Die Funktion *addFile* erzeugt für alle relevanten Wörter des übergebenen Textfiles *file* Einträge im Index. Der Parameter *tmp* wird nur dann mit übergeben, wenn eine temporäre Datei erzeugt wurde – dies ist beispielsweise bei der Verarbeitung von PDF-Dateien der Fall, die mittels eines externen Programms in Textdateien umgewandelt werden. In der in Zeile 25 in Listing 3.34 mittels einer Listenkomprehension erzeugten Liste *words* befinden sich alle Wörter von *file* die dem Index hinzugefügt werden sollen.

Die Funktion *crawl* implementiert den Crawler; in unserem Fall läuft der Crawler über die Verzeichnisstruktur und fügt alle Dateien, die textuelle Information enthalten, dem Index hinzu; in dieser einfachen Variante kann *crawl* lediglich pdf- und Textdateien indizieren.

Aufgabe 3.56

Erklären Sie die Listenkomprehension in Zeile 25 in Listing 3.34: wozu die beiden **for**-Schleifen, wozu die **if**-Anweisung?

3.9.4 Erweiterte Anforderungen

Erweiterte Anforderungen, die im Rahmen der Aufgaben noch nicht angedacht wurden, die aber von den „großen“ Suchmaschinen, unter anderem vom Opensource Framework Lucene [13] und Google's Suchmaschinenalgorithmen verwendet werden.

1. Insbesondere dann, wenn die Anzahl der zu indizierenden Dokumente und folglich auch die Größe des Indexes die Ressourcen eines einzelnen Rechners übersteigt, muss man darüber nachdenken den Crawler, Indexer und die Indizes verteilt über mehrere Maschinen arbeiten zu lassen. Das von Google beschriebene MapReduce-Framework bietet hierfür eine nützliche Schnittstelle [7, 15].

2. Wenn man die Usability[11] verbessern will, dann ist es hilfreich, einen Dokumenten-Cache[12] mit zu verwalten, d. h. kleine Textteile, die einen möglichst repräsentativen Auszug aus einem Dokument bilden, werden für den schnellen Zugriff effizient gespeichert.

3. Um die Qualität der Suchergebnisse zu verbessern könnte man die Textstruktur beim Indizieren mit berücksichtigen: So könnte man etwa Vorkommen eines Wortes in Überschriften anders gewichten, als die Vorkommen eines Wortes in einem Paragraphen.

Aufgabe 3.57

Die in Listing 3.34 vorgestellte Implementierung einer Suchmaschine verwendet als Datenstruktur für den Index den Python-Typ *dict*, d. h. Hashtabellen. Reale Suchmaschinen verwenden dagegen sehr oft Tries bzw. Patricia-Tries.

(a) Verwenden Sie statt dem Python *dict*-Typ für *self.ind* besser den im vorigen Abschnitt vorgestellten Trie. Vergleichen Sie nun Laufzeit und Größe der als Index entstehenden Datenstruktur bei Verwendung von *dict* und bei Verwendung von *Trie*.

(b) Verwenden Sie statt dem Python *dict*-Typ für *self.ind* besser den im vorigen Abschnitt vorgestellten Patricia-Trie. Vergleichen Sie nun Laufzeit und Größe der als Index entstehenden Datenstruktur bei Verwendung von *dict* und bei Verwendung von *Patricia*.

Aufgabe 3.58

Erweitern Sie den Indexer so, dass auch die Position innerhalb einer Zeile mit berücksichtigt wird.

[11] Als Usability bezeichnet man oft auch in der deutschsprachigen Literatur die Benutzbarkeit aus Anwendersicht; dazu gehören Eigenschaften wie Verständlichkeit, Fehlertoleranz, Übersichtlichkeit, usw.

[12] Als Cache bezeichnet man in der Informatik in der Regel einen schnellen kleinen Speicher, der diejenigen Teile eines größeren Datenspeichers zwischenspeichert, von denen zu erwarten ist, dass sie momentan bzw. in Zukunft oft verwendet werden; viele Festplatten verwenden Cache-Speicher und auch viele Rechner verwenden schnelle Cache-Speicher um die Zugriffsperformance auf den Hauptspeicher zu optimieren.

Aufgabe 3.59

Implementieren Sie eine Methode *Index.askHTML* so, dass ein HTML-Dokument zurückgeliefert wird, in dem die Treffer als Hyperlinks auf die jeweiligen Dokumente dargestellt sind.

Aufgabe 3.60

(a) Die Methode *Index.ask* gibt die Treffer für ein Suchwort beliebig zurück. Modifizieren Sie *Index.ask* so, dass die Treffer (also die Dokumente, in denen das das Suchwort enthalten ist) nach Gewicht sortiert ausgegeben werden. Hierbei soll das Gewicht gleich der Anzahl der Vorkommen des Suchworts im jeweiligen Dokument sein.

(b) Geben Sie die Treffer nun sortiert nach der relativen Häufigkeit des Vorkommens des Suchworts zurück. Bei der relativen Häufigkeit wird einfach die Größe des Dokuments noch mit berücksichtigt, d. h.

$$\textit{rel. Häufigkeit} = \frac{\textit{Häufigkeit}}{\textit{Dokumentengröße}}$$

Aufgabe 3.61

(a) Programmieren Sie eine Funktion *Index.remove*(*file*), mit der man eine im Index befindliche Datei wieder entfernen kann.

(b) Programmieren Sie eine Funktion *Index.update*(*file*), mit der man eine im Index befindliche womöglich veraltete Datei auf den neusten Stand bringen kann.

Aufgabe 3.62

Implementieren Sie ein einfaches Stemming, indem Sie die häufigsten Endungen `'ung'`, `'ungen'`, `'en'`, `'er'`, `'em'` und `'e'` abschneiden. Dies muss dann natürlich auch bei der Suche mit berücksichtigt werden, d. h. Suchwörter müssen vor der eigentlichen Suche mit dem selben Algorithmus gestemmt werden.

Aufgabe 3.63

Suchmaschinen indizieren aus Effizienzgründen üblicherweise nicht alle Wörter. Viele kurze Wörter, die es nahezu in jedem Dokument gibt, werden ignoriert. Diese Wörter werden im Information Retrieval oft als Stoppwörter bezeichnet. Erweitern Sie die Methode *Index.addtoIndex* so, dass ein Wort nur dann eingefügt wird, wenn es nicht aus einer vorgegebenen Menge von Stoppwörtern stammt.
Tipp: Verwenden Sie als Stoppwörter entweder einfach die wichtigsten bestimmten und unbestimmten Artikel, Präpositionen, Konjunktionen und Negationen; oder, alternativ, besorgen Sie sich aus Quellen wie etwa [1] eine Liste von Stoppwörtern.

Aufgabe 3.64

Bisher wurden lediglich pdf-Dateien und reine Textdateien indiziert.

(a) Parsen und indizieren sie zusätzlich HTML-Dateien.

(b) Parsen und indizieren sie zusätzlich Openoffice-Dateien.

(c) Parsen und indizieren sie zusätzlich MS-Office-Dateien.

(d) Parsen und indizieren sie zusätzlich TEX-Dateien.

Aufgabe 3.65

Realisieren Sie die Möglichkeit den erzeugten Index abzuspeichern und einen abgespeicherten Index wieder zu laden; je größer der Index, desto sinnvoller ist es, ihn persistent, d. h. dauerhaft und über die Laufzeit des Programms hinausgehend, zu speichern. Python stellt hierfür die Module *pickle*, *shelve* und/oder *marshal* zur Verfügung.