Skiplisten

20. März 2021

Dieses Kurzskript orientiert sich an der Originalveröffentlichung von William Pugh [Pug90] sowie an Vorlesungsskripten von Petra Mutzel, Markus Chimani, Carsten Gutwenger und Karsten Klein [MCGK06] und Ingo Wegener und Thomas Hofmeister [WH04].

1 Einleitung

Wir beschäftigen uns in diesem Text mit einer dynamischen Suchdatenstruktur, die unter dem Namen *Skipliste* bekannt ist. Dieser Name enthält offenbar das Wort *Liste*. Wir beginnen daher damit, uns in Erinnerungzu rufen, wie gut sich elementare Datenstrukturen als dynamische Suchdatenstrukturen einsetzen lassen. Wir gehen (hier und auch später) davon aus, dass wir uns nur um die Verwaltung von eindeutigen Schlüsseln aus dem Zahlenbereich $\{1, \ldots, U\}$ kümmern müssen und die tatsächlichen Daten passend verwaltet werden. Die Anzahl der (aktuell) zu verwaltenden Schlüssel bezeichnen wir mit n. Wir möchten die Operationen `INIT()` (initialisiert die Datenstruktur), `SEARCH(s)` (liefert das Element mit Schlüssel s zurück, wenn vorhanden), `INSERT(s)` (fügt ein Element mit Schlüssel s ein, wenn noch nicht vorhanden) und `DELETE(s)` (löscht das Element mit Schlüssel s, wenn vorhanden) realisieren.

Arrays lassen sich als *statische* Suchdatenstrukturen einsetzen. Dazu sortieren wir zunächst die Schlüssel, die wir abspeichern wollen, aufsteigend. Für jede Suchanfrage führen wir dann in Zeit $O(\log n)$ eine binäre Suche durch, um festzustellen, ob der Schlüssel im Array enthalten ist und wenn ja, an welcher Position. Wollen wir die Schlüsselmenge jedoch verändern, so kann dies lange dauern. Ein neuer Schlüssel muss möglicherweise in der Mitte eingefügt werden, so dass wir eine lineare Anzahl von Verschiebungen durchführen müssen und auch das Löschen eines Schlüssels kann zu einer linearen Anzahl von Verschiebungen führen.

Sortierte Listen hingegen lassen sich leicht anpassen, wenn Elemente gelöscht oder hinzugefügt werden sollen – allerdings nur, wenn man die richtige Position in der Liste bereits gefunden hat. Denn Listen sind uns als Datenstruktur bekannt, in der das Suchen nach Elementen lineare Zeit benötigen kann, unabhängig davon, ob die Liste sortiert ist oder nicht. Können wir dies irgendwie umgehen und eine Liste entwerfen, in der wir ähnlich suchen können wie in einem Array (also in logarithmischer Zeit und mit einer Art binärer Suche)? Dies ist das Thema dieses Kurzskripts.

Die Idee zur Umsetzung ist überraschend. Wir betrachten dazu Abbildung 1.

In der obersten Zeile sehen wir eine gewöhnliche (einfach) verkettete Liste, wobei die Elemente aufsteigend nach Schlüsseln sortiert sind. In der zweiten Zeile gibt es zwei verkettete Listen, und jedes zweite Element ist Teil beider Listen. Suchen wir nun ein Element, so können wir solange der oberen Liste folgen, bis wir ein größeres Element gefunden haben (oder das gesuchte Element). Erst dann wechseln wir auf die untere Liste. Auf diese Weise können wir ungefähr die Hälfte der Suchanfragen einsparen. In der untersten Zeile ist eine Datenstruktur mit drei Listen realisiert, bei der die oberste nur jedes vierte Element enthält. Nun können wir zunächst der oberen Liste folgen, dann der mittleren, und schließlich der untersten.

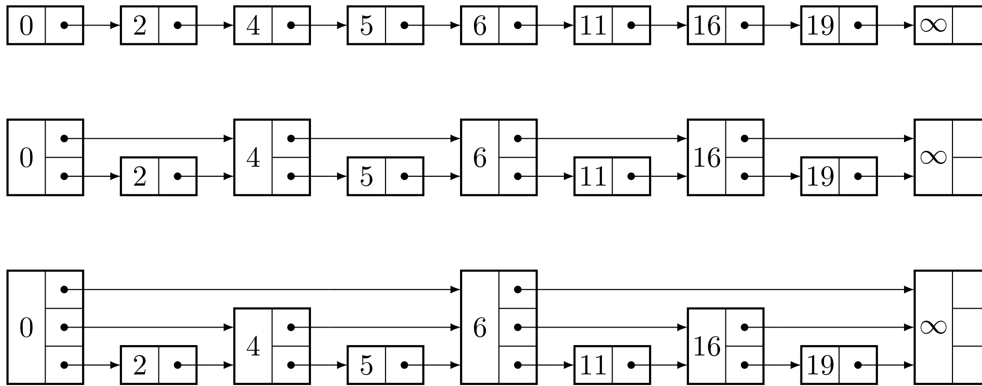

Abbildung 1: Eine einfach verkettete Liste, eine Liste mit zusätzlichen Zeigern und eine ‚perfekte' Skipliste mit sieben Elementen.

Wir stellen fest, dass eine Suche nun plötzlich einer binären Suche ähnelt: Zuerst identifizieren wir mit einer Suche in der obersten Liste, ob unser gesuchtes Element im linken oder rechten Teil liegt, dann mit der mittleren, ob wir uns im linken oder rechten Teil des eingeschränkten Suchbereichs befinden, und mit der untersten Liste finden wir schließlich das Element. Wenn die Liste mehr als die hier angegebenen sieben Elemente enthält, müssen wir natürlich auch mehr Listen verwenden, um dieses Prinzip aufrechtzuerhalten. Mit etwas Überlegen stellen wir fest, dass dafür etwa $\log n$ Listen benötigt werden. Die i. Liste enthält dabei jedes 2^{i-1}. Element, und wir beginnen die Suche immer in der Liste, die die wenigsten Elemente enthält.

Wir können uns die Details einer solchen *perfekten* Skipliste nun überlegen (z.B. in der Übung). Jedoch stoßen wir dabei recht schnell auf ein Problem: Wie halten wir eine solche Datenstruktur *dynamisch* aufrecht, d.h. bei Einfügen und Löschen von Elementen? Auch

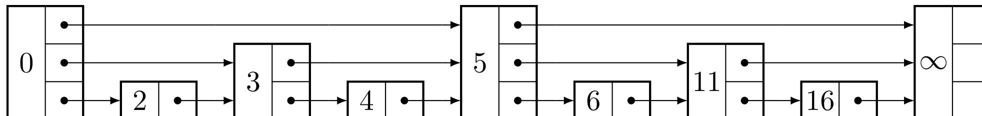

Abbildung 2: Perfekte Skipliste, die im Vergleich zu Abbildung 1 die 3 zusätzlich enthält und die 19 nicht mehr enthält.

ohne die genauen Lösch- und Einfügenoperationen zu spezifizieren, sehen wir beim Vergleich von Abbildung 1 und Abbildung 2, dass das Löschen von 19 mit anschließendem Einfügen von 3 die gesamte Listenstruktur verändert. An jedem Element müssen wir Zeiger verändern. Wir haben also (noch) nichts gegenüber einer normalen verketteten Liste gewonnen (und den Speicheraufwand erhöht).

Allerdings würde die Datenstruktur wohl immer noch recht gut funktionieren, wenn wir nicht auf der exakten Struktur beharren. Auch in der in Abbildung 3 abgebildeten Struktur lässt sich noch recht schnell suchen. Die zweite Idee ist nun, für das Einfügen und Löschen Randomisierung zu verwenden. Anstatt die Suchstruktur exakt aufrechtzuerhalten, werden Elemente mit einer zufälligen Höhe eingefügt (die Höhe ist die Anzahl der Listen, in denen ein Element enthalten ist). In einer perfekten Skipliste haben etwa die Hälfte der Elemente Höhe 1, ein Viertel Höhe 2, ein Achtel Höhe 3 und so weiter. Im Erwartungswert kann man eine ähnliche Verteilung auf einfache Weise folgendermaßen erreichen: Beim Einfügen eines Elements wirft man eine Münze und zählt die Anzahl Münzwürfe, bis zum ersten Mal Kopf

erscheint (den Münzwurf, bei dem Kopf erscheint, zählt man mit). Diese Anzahl ist die Höhe des Elements. In einer Skipliste haben Elemente also eine zufällige Höhe, basierend auf diesem einfachen Zufallsexperiment.

Damit haben wir die grundsätzlichen Ideen für den Entwurf von Skiplisten kennengelernt. In Abschnitt 2 besprechen wir die Details einer möglichen Skiplistenimplementierung in Pseudocode, und in Abschnitt 3 analysieren wir die Datenstruktur.

2 Eine Skiplistenimplementierung in Pseudocode

Die *Höhe* eines Elements entspricht der Anzahl der Listen, in denen es enthalten ist, also der Anzahl der ausgehenden Zeiger, die das Element hat. Alle Elemente mit Höhe i bilden das *Level* i. Wir speichern die Zeiger in Arrays, die ab 0 indiziert sind. Die Höhe ändert sich für normale Elemente nach dem Einfügen nicht mehr (d.h. wir sollten in einer tatsächlichen Implementierung für die Zeiger Arrays fester Größe verwenden), aber für das erste Elemente der Liste muss sie dynamisch anpassbar sein, daher verwenden wir hier der Einfachheit halber überall dynamische Arrays für die Zeiger. Unsere Datenstruktur hat nun folgende Komponenten:

- Einen Zeiger `head` sowie einen Zeiger `tail` auf das erste und das letzte Listenelement
- `Element`-Objekte für die einzelnen Elemente. Objekt `e` der Klasse `Element` enthält
  - die Höhe `e.height`,
  - den Schlüssel `e.key`, sowie
  - ein dynamisches Array `e.next` mit `e.height` vielen Elementen, wobei `e.next[i]` einen Zeiger auf den Nachfolger des Elements `e` auf Level i enthält.

Wir gehen davon aus, dass `new Element(`s, h`)` ein neues Element mit Schlüssel s und Höhe h erzeugt, d.h. es wird ein dynamisches Array angelegt, `height` auf h und `key` auf s gesetzt. In unserem Entwurf werden das erste und das letzte Element der Liste keine echten Daten erhalten. Das Endelement enthält einen Schlüssel, der größer ist als alle vorkommenden Elemente, was wir hier durch ∞ ausdrücken. Was das Startelement enthält, ist unerheblich, wir verwenden hier 0. Start- und Endelement bilden den Rahmen der Liste. Die Höhe des Startelements entspricht immer der maximal vorkommenden Elementhöhe, die Höhe des Endelements ist unerheblich. Wir initialisieren unsere Liste folgendermaßen.

```
INIT()
 1: head = new Element(0,1)
 2: tail = new Element(∞,1)
 3: head.next[0] = tail
```

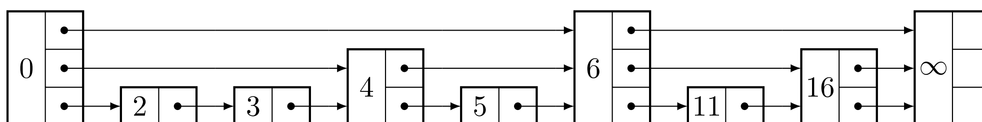

Abbildung 3: Eine fast balancierte Skipliste.

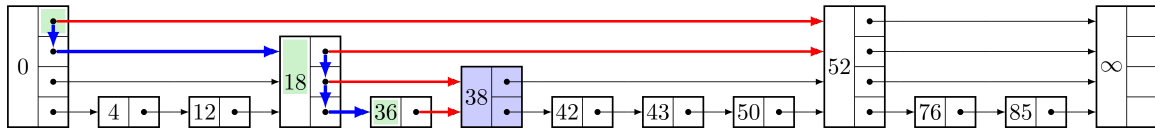

Abbildung 4: Erfolgreiches Suchen nach Schlüssel 38 oder erfolgloses Suchen nach Schlüssel 37.

Suchen Mit der Methode `SEARCH(s)` suchen wir das Element s in der Skipliste.[1] Die Beschreibung der Methode findet sich unter dem Pseudocode.

```
SEARCH(s)
 1: p = head
 2: predecessor = array(head.height)
 3: for i = head.height-1 downto 0 do
 4:   while p.next[i].key < s do
 5:     p = p.next[i]
 6:   predecessor[i] = p
 7: p = p.next[0]
 8: return (p, predecessor)
```

Zunächst folgen wir den Zeigern auf dem höchsten Level, bis wir an einem Element angekommen sind, dessen Nachfolger größer oder gleich s ist. Dann steigen wir ein Level ab. Erneut suchen wir das erste Element, dessen Nachfolger die Bedingung $\geq s$ erfüllt. Wir wiederholen dies, bis wir auf der untersten Ebene das erste Element gefunden haben, dessen Nachfolger $\geq s$ erfüllt (dies ist spätestens erfüllt, wenn der Nachfolger das Endelement ist, da dieses als Schlüssel ∞ enthält). Am Schluss ist der gefundene Nachfolger entweder das gesuchte Element, oder s ist in der Liste nicht enthalten.

Abbildung 4 veranschaulicht das Vorgehen für eine erfolgreiche Suche nach 38 oder eine erfolglose Suche nach 37. In beiden Fällen enthält p am Ende das Element mit dem Schlüssel 38. Die grün hinterlegten Flächen bilden zusammen das Array `predecessor`. Dieses Array benötigen wir für die Suche nach s nicht, wir werden es jedoch beim Löschen und Einfügen von Elementen benötigen. Es enthält für alle Höhen einen Zeiger auf das größte Element in der entsprechenden Liste, das kleiner ist als s (das ist gerade das Element, dessen Nachfolger das erste Element ist, für das $\geq s$ gilt).

Löschen Mit der Methode `DELETE(s)` entfernen wir das Element mit dem Schlüssel s aus der Liste, wenn es enthalten ist (und geben einen Zeiger auf das Element zurück).

```
DELETE(s)
 1: (p,predecessor) = SEARCH(s)
 2: if p != s
 3:    return NULL
 4: for i = 0 to p.height-1 do
 5:   predecessor[i].next[i] = p.next[i]
```

[1] Im Folgenden werden wir uns hin und wieder zur Vereinfachung erlauben, Elemente mit den gespeicherten Schlüssel gleichzusetzen. Wir suchen natürlich eigentlich das Element mit dem Schlüssel s.

```
 6: h' = head.height
 7: while head.next[h'-1] == tail and h' > 1 do
 8:   h' = h'-1
 9: head.resize(h')
10: return s
```

Zunächst rufen wir `SEARCH(s)` auf. Wir erfahren so, ob s in der Liste enthalten ist, und erhalten das `predecessor`-Array. Wenn das Element p den Schlüssel s enthält, löschen wir es aus der Liste. Dies lässt sich einfach realisieren, indem von allen Vorgängern in allen Listen die Zeiger auf die entsprechenden Nachfolger von p gesetzt werden (Zeile 4 und 5). Zeilen 6 bis 9 dienen nun lediglich dazu, die neue maximale Elementhöhe zu ermitteln und sowohl `head.height` als auch `head.next` entsprechend anzupassen, wobei darauf geachtet wird, dass die Höhe mindestens 1 bleibt. Die Methode `resize(h)` setzt `height` auf h und die Größe von `next` ebenfalls (führt hier gegebenenfalls zum Abschneiden von Elementen).

Einfügen Für das Einfügen gehen wir ähnlich vor. Wir benötigen zunächst eine Methode, die für ein neues Element eine Höhe bestimmt. Folgender Pseudocode setzt das in der Einleitung beschriebene Münzwurfexperiment um.

```
RANDOMHEIGHT()
1: h = 1
2: while random(0,1) == 1
3:   h = h + 1
4: return h
```

Die Höhe ist also mindestens 1. Es gibt keine obere Schranke für die Höhe, die garantiert eingehalten wird. Dies deutet schon an, dass wir für Skiplisten keine oberen Schranken für Laufzeit oder Speicherplatzbedarf im Worst-Case-Modell zeigen können. Wir besprechen nun die Einfügeoperation.

```
INSERT(s)
 1: (p,predecessor) = SEARCH(s)
 2: if p == s
 3:   return
 4: h = RANDOMHEIGHT()
 5: p = new Element(s,h)
 6: for i = 0 to head.height-1 do
 7:   p.next[i] = predecessor[i].next[i]
 8:   predecessor[i].next[i] = p
 9: oldheight = head.height
10: head.resize(max(old-height,h))
11: for i = oldheight to h-1 do
12:   head.next[i] = p
13:   p.next[i] = tail
```

Das Einfügen beginnt auch mit einer Suche. Ist die Suche erfolgreich, so müssen wir nichts tun (außer gegebenenfalls die gespeicherten Informationen zu aktualisieren). Nach einer erfolglosen Suche verwenden wir die obige Funktion, um eine zufällige Höhe zu bestimmen. Dann wird ein neues Element `p` erzeugt. Das `predecessor`-Array enthält genau die Elemente, hinter denen `p` eingefügt werden muss. Wir nutzen es, um die Zeiger auf `p` umzuleiten und die ursprünglichen Nachfolger als Nachfolger von `p` einzusetzen. Zeilen 9 bis 13 werden für den Fall benötigt, dass h größer ist als die bisherige maximale Elementhöhe. In diesem Fall müssen nicht nur `head.height` und die Größe von `head.next` korrigiert werden, sondern es müssen auch die entsprechenden Zeiger von `head` auf `p` bzw. von `p` auf `tail` gesetzt werden, um die neuen Listen zu erzeugen. Wenn `h` nicht größer ist als `old-height`, so haben diese Zeilen keinen Effekt.

3 Analyse

Die Analyse von Skiplisten hat viel mit der Analyse von Münzwurfexperimenten zu tun. Wir wollen die dahinterliegende Mathematik hier nicht näher besprechen, aber auf die benötigten Ergebnisse verweisen. Ein Experiment, das nur zwei Ausgänge kennt, die wir mit 0 und 1 bezeichnen, und das den Wert 1 mit Wahrscheinlichkeit w annimmt, ist eine *Bernoulli-Experiment* mit Parameter w. Führt man nun n unabhängige Bernoulli-Experimente mit demselben Parameter w durch und definiert eine Zufallsvariable, die der Anzahl der Versuche bis zum ersten Auftreten von 1 entspricht (inklusive des Versuchs, bei dem die 1 auftritt), so ist diese Zufallsvariable *geometrisch verteilt* mit Parameter w.

Satz 1. *Sei X eine geometrisch verteilte Zufallsvariable mit Parameter $w \in (0, 1]$. Dann gilt:*

- *Die Wahrscheinlichkeit, dass X den Wert $i \in \mathbb{N}^{\geq 1}$ annimmt, ist $w^{i-1} \cdot w$. Die Wahrscheinlichkeit, dass X* mindestens *den Wert $i \in \mathbb{N}^{\geq 1}$ annimmt, ist w^{i-1}.*
- *Der Erwartungswert von X ist $1/w$.*
- *Der Erwartungswert $E[Y]$ der Zufallsvariable $Y = \max_{i=1,\dots,n} X$, also der erwartete maximale Wert von X, erfüllt*

$$E[Y] < 2 + \frac{w}{1-w} + \log_{1/w} n \in O(\log n)$$

Beweis. Die erste Aussage gilt, da wir genau dann i als Anzahl erhalten, wenn $p-1$ Mal 0 und einmal 1 das Ergebnis des Experiments ist. Die zweite Aussage ist recht intuitiv und der Beweis ist in Standardwerken zu randomisierten Algorithmen enthalten, zum Beispiel in Abschnitt 2.4 in [MU17]. Die dritte Aussage beweisen wir (Beweis aus [WH04]):

$$\begin{aligned} E[Y] &= \sum_{i=1}^{\infty} h \cdot \text{Prob}(Y = h) = \sum_{i=1}^{\infty} \text{Prob}(Y \geq h) \\ &= \sum_{i=1}^{\lceil \log_{1/w} n \rceil + 1} \text{Prob}(Y \geq h) + \sum_{i=\lceil \log_{1/w} n \rceil + 2}^{\infty} \text{Prob}(Y \geq h) \end{aligned}$$

$$\leq \sum_{i=1}^{\lceil \log_{1/w} n \rceil + 1} 1 + \sum_{i=\lceil \log_{1/w} n \rceil + 2}^{\infty} n \cdot w^{i-1} \leq \log_{1/w} n + 2 + n \cdot w^{\log_{1/w} n} \sum_{i=1}^{\infty} w^i$$

$$= \log_{1/w} n + 2 + \frac{w}{1-w}. \qquad \square$$

Was haben diese Aussagen nun mit der Analyse der Skiplistendatenstruktur zu tun? Wir bemerken, dass die Höhe eines neu einzufügenden Elements geometrisch verteilt ist. Der Parameter ist $w = 1/2$. Dann wissen wir aus Satz 1, dass der Erwartungswert für die Höhe eines neuen Elements $1/w = 1/(1/2) = 2$ ist, und dass die erwartete maximale Höhe eines Elements nach n Einfügeoperationen durch $O(\log n)$ beschränkt ist (wir erhalten aus Satz 3 als Schranke $3 + \log_2 n$). Damit ist die Gesamthöhe der Liste im Erwartungswert durch $O(\log n)$ beschränkt.

Dies gilt nicht nur nach n Einfügeoperationen, sondern auch, wenn nach mehrfachem Löschen und Einfügen noch n Elemente in der Skipliste sind. Skiplisten haben nämlich eine interessante Eigenschaft: Sie sind gedächtnislos. Das Aussehen einer Skipliste ist nicht abhängig von bereits wieder gelöschten Elementen, da alle Höhen unabhängig zufällig gezogen werden und die Struktur nur von den Höhen abhängt.

Die beschränkte erwartete Höhe erlaubt uns bereits, einen großen Teil der Laufzeitanalyse für die Operationen zu erledigen. Abgesehen von dem Aufruf von `SEARCH(s)` führen `DELETE(s)` und `INSERT(s)` nur einzelne Operationen aus und Schleifen, deren Iterationenzahl von der Höhe der Liste oder des neu eingefügten Elements abhängt. Der erwartete Aufwand dafür ist asymptotisch beschränkt durch die erwartete Höhe und damit in $O(\log n)$. Wir müssen also nur noch die Laufzeit der Operation `SEARCH(s)` analysieren. Auch hier spielt die erwartete Höhe der Liste eine Rolle, hinzu kommt jedoch die erwartete Länge des *Suchpfads*.

Satz 2. *Die erwartete Laufzeit der Methoden* `SEARCH(s)`*,* `DELETE(s)` *und* `INSERT(s)` *ist durch* $O(\log n)$ *beschränkt.*

Beweis. Wie beschrieben müssen wir vor allem die erwartete Länge des Suchpfads, den `SEARCH(s)` auf der Suche nach `s` durchläuft, analysieren. Dies geht einfacher, wenn wir den Suchpfad *rückwärts* betrachten. Wir beginnen also bei dem Element `p`, an dem die Suche beendet wurde (unabhängig davon, ob dieses s wirklich enthält oder nicht). Von diesem Element laufen wir nun zum Kopf der Liste zurück. Dabei können wir uns vorstellen, dass die Höhen der Elemente erst während unseres Zurücklaufens zufällig bestimmt werden.

Der Suchpfad läuft in jedem Schritt entweder nach oben oder nach links (siehe Abbildung 4, von 36 verläuft der Pfad einen Schritt nach links, dann zwei nach oben, dann einen nach links und wieder einen nach oben). Der Pfad verläuft genau dann nach oben, wenn das Element, in dem wir uns befinden, auch in der Liste über uns existiert. (Beim Vorwärtssuchen betreten wir Elemente nämlich immer auf ihrer höchsten Ebene). Aber die Wahrscheinlichkeit dafür beträgt gerade 1/2! Denn die Wahrscheinlichkeit, dass ein Element, das Teil der Liste auf Level i ist, auch Teil der Liste auf Level $i + 1$ ist, ist ja gerade 1/2. Das bedeutet, dass wir mit Wahrscheinlichkeit 1/2 in die ‚gute' Richtung gehen, also nach oben, und mit Wahrscheinlichkeit 1/2 nach links. Wir haben es erneut mit einer geometrisch verteilten Zufallsvariable zu tun: Wir brauchen im Erwartungswert zwei Versuche (und damit zwei Schritte auf dem Suchpfad), um einmal nach oben zu gelangen.

Wir schätzen daher zunächst ab, dass wir maximal $2\lceil \log n \rceil + 2$ Schritte benötigen, um auf Level $\lceil \log n \rceil + 1$ aufzusteigen (oder das Startelement zu erreichen). Ab diesem Zeitpunkt laufen wir nur noch durch Elemente, deren Höhe mindestens $\lceil \log n \rceil + 1$ beträgt. Solche Elemente gibt es im Erwartungswert aber nicht viele! Die Wahrscheinlichkeit, dass ein Element mindestens Höhe $\lceil \log n \rceil + 1$ hat, beträgt nämlich laut Satz 1 gerade $(1/2)^{\lceil \log n \rceil} = 1/n$. Daher ist die Anzahl der Elemente auf dieser Höhe im Erwartungswert gleich 1. Wir laufen also im Erwartungswert nur noch durch ein Element, bis wir den Anfang der Liste erreicht haben.

Die Laufzeit von `SEARCH(s)` entspricht asymptotisch der Länge des Suchpfads und liegt daher in $O(\log n)$. Für die anderen Operationen haben wir bereits argumentiert, dass die Laufzeit im Erwartungswert $O(\log n)$ plus der Laufzeit von `SEARCH(s)` ist, also erhalten wir auch hier eine erwartete Laufzeit von $O(\log n)$. □

Literatur

[MCGK06] Mutzel, Petra ; Chimani, Markus ; Gutwenger, Carsten ; Klein, Karsten: *Datenstrukturen , Algorithmen und Programmierung II.* Sommersemester 2006. – Vorlesungsskript

[MU17] Mitzenmacher, Michael ; Upfal, Eli: *Probability and computing.* Cambridge University Press, 2017

[Pug90] Pugh, William: Skip Lists: A Probabilistic Alternative to Balanced Trees. In: *Communications of the ACM* 33 (1990), Nr. 6, 668–676. `http://dx.doi.org/10.1145/78973.78977`. – DOI 10.1145/78973.78977

[WH04] Wegener, Ingo ; Hofmeister, Thomas: *Datenstrukturen, Algorithmen und Programmierung 2.* Sommersemester 2004. – Vorlesungsskript