Theorie

Sie haben bereits einfach verkettete Listen und als Spezialfall auch die sortierten verketteten Listen kennen gelernt. Dabei haben Sie gesehen, dass die Operation *Suchen* eine entscheidende Rolle spielt, da sie sowohl für das *Entfernen* wie auch für das *Einfügen* an einer bestimmten Stelle benötigt wird. Wenn es gelingen würde, diese Operation schneller zu realisieren, wäre es vielleicht möglich, die ganze Datenstruktur zu beschleunigen.

Aus dieser Einführung ist unschwer zu erahnen, dass die Skip-Liste genau das schafft. Es ist eine *sortierte, einfach verkettete Liste*, die uns aber ein schnelleres Suchen von Elementen in der Datenstruktur erlaubt. Doch auf welche Kosten? Sind nun die Operationen *Einfügen* und/oder *Entfernen* langsamer? Dies wird eine der wichtigen Fragen sein, welche wir klären müssen.

- Sie lernen eine einfache und effiziente Datenstruktur kennen.
- Sie lernen, einen konkreten Algorithmus zu analysieren.
- Sie lernen eine Java-Implementierung dieser Datenstruktur kennen.

1. Idee der Skip-Liste

In einer sortierten, verketteten Liste müssen wir jedes Element einzeln durchlaufen bis wir das gewünschten Element gefunden haben (*Figur 1a*). Wenn wir nun aber in der sortierten Liste auf jedem zweiten Element eine zusätzliche Referenz auf zwei Elemente weiter hinten setzen, dann reduziert sich die Anzahl zu besuchender Elemente auf einen Schlag um rund die Hälfte (*Figur 1b*). Genau betrachtet müssen wir nie mehr als $(n/2) + 1$ Elemente besuchen (n ist die Länge der Liste). Denken wir diese Idee weiter, dann kommt sofort die Idee, zusätzlich auf jedem vierten Element noch eine Referenz auf vier Elemente weiter hinten zu setzen (*Figur 1c*). Dann müssen wir beim Suchen nur noch $(n/4) + 2$ Elemente besuchen. Allgemein ausgedrückt, möchten wir folgendes tun:

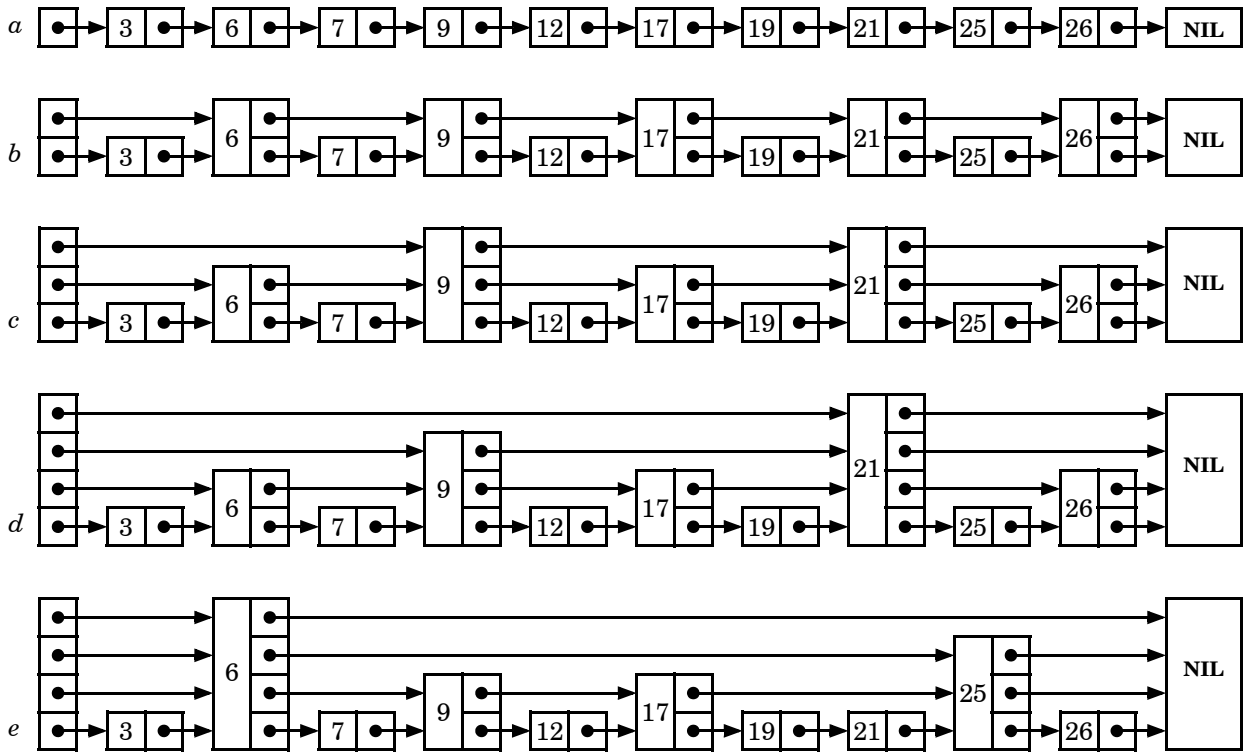
- Jedes Element ist durch eine Level-0-Referenz mit dem nächstfolgenden Element verbunden.
- Jedes zweite Element ist durch eine zusätzliche Level-1-Referenz mit dem übernächsten Element verbunden.
- Diese Idee wird weiterentwickelt, so dass jedes 2^i-te Element eine Referenz auf Level i hat, die 2^i Elemente weiter nach hinten zeigt. Somit hat jedes Element $i + 1$ Referenzen.

Natürlich gibt es sofort offene Fragen. Wie speichere ich diese zusätzlichen Referenzen? Die einfache Lösung ist, dass jedes Element ein Array von Referenzen besitzt, das jeweils $i + 1$ lang ist. In dieses Array werden dann die Referenzen gespeichert (*Figur 1*). Leider schränken wir auf diese Weise die Dynamik der Liste ein. Arrays haben eine fixe Länge und wenn wir die Skip-Liste anwachsen lassen kommt der Moment, wo ein Element in der Liste einen neuen Speicherplatz für eine zusätzliche Referenz benötigt. Arrays sind aber keine dynamischen Datenstrukturen und deshalb nicht einfach erweiterbar. Eine Alternative wäre, anstatt Arrays verkettete Listen zu verwenden. Damit hätten wir wieder eine vollständig dynamische Datenstruktur.

Es gibt aber ein noch viel gravierenderes Problem: Die Operationen *Einfügen* und *Entfernen* werden mit einer solchen Referenz-Struktur extrem aufwendig, falls die Regelmässigkeit der Referenzstruktur aufrecht erhalten bleiben muss. Denn bei jeder Operation müssten wir praktisch die gesamte Referenz-Struktur umbauen und würden uns so das schnellere *Suchen* viel zu teuer erkaufen.

Wenn jedes 2^i-te Element eine Referenz auf 2^i Elemente weiter hinten hat, dann sind die Levels ganz einfach verteilt: 50% der Elemente haben nur Level 0, 25% Level 1, usw. Was wäre, wenn die Level der Elemente zufällig gewählt würden, einfach proportional gleich verteilt wie vorhin beschrieben (*Figur 1e*)? Die i-te Referenz eines Elementes muss nicht mehr strikt auf das Element 2^i weiter hinten zeigen sondern einfach auf das nächste Element mit Level i oder höher. Damit müssen zum *Entfernen* und *Einfügen* nur noch lokale Änderungen vorgenommen werden und der vorgängig erwähnte Nachteil verschwindet. Der Einfachheit halber führen wir einen `MaxLevel` ein, welcher die maximale Höhe des Levels eines Elementes vorschreibt. Dadurch können wir die einfachere Lösung mit den Arrays für die Referenzen benutzen.

Offensichtlich können auch jetzt noch Situationen entstehen, bei denen *Suchen* unglücklicherweise sehr aufwendig ist. Zum Glück aber sind solche Fälle sehr selten und man kann davon ausgehen, dass sie kaum entstehen. Auch der Wert *MaxLevel* kann sinnvoll gewählt werden, dass die *Suchen*-Operation auch bei anwachsender Skip-Liste schnell bleibt.



2. Implementierung

2.1 Datenstrukturen

In diesem Abschnitt geht es um die konkrete Realisierung der Skip-Liste und der drei Operationen *Suchen*, *Einfügen* und *Entfernen*. Um zwischen Elementen zu vergleichen und damit überhaupt eine Sortierung zu ermöglichen, gehen wir davon aus, dass jedes Element in der Skip-Liste einen Schlüssel `m_key` hat. Die Reihenfolge der Elemente wird aufgrund von diesem `m_key` bestimmt. Neben dem `m_key` haben die Elemente auch ein Array `m_next`, um die Referenzen zu speichern (aus der einfachen Referenz `m_next` der einfach verketteten Liste ist jetzt ein ganzes Array von Referenzen geworden). Die Länge dieses Arrays gibt auch gleich den Level an, der ein Element hat (der Level ist `m_next.length` – 1). Diese Länge wird beim Einfügen zufällig gewählt und darf den Wert `MaxLevel` der Skip-Liste nicht überschreiten.

```
class Element {
  private int m_key; // Vergleichsschlüssel
  private Element[] m_next; // Referenz-Speicher
  ...
}
```

Um die Implementierung der Operationen möglichst einfach zu halten, hat die Skip-Liste ein Dummy-Element für den Anfang der Skip-Liste, das zugleich der Anker ist, und ein Dummy-Element für den Schluss der Skip-Liste. Diese zwei Elemente haben den Level *MaxLevel* und den kleinstmöglichen respektive den grösstmöglichen Schlüssel.

```
class SkipListe {
  private Element m_headAnchor;    // kleinstmöglicher key, level = MaxLevel
  private Element m_tailAnchor;    // grösstmöglicher key, level = MaxLevel
  private int m_maxLevel;   // speichert den maximalen Level
}
```

2.2 Operationen

Suchen: Um ein Element zu finden, wandern wir so lange auf dem aktuellen Level den Referenzen entlang, wie das zu suchende Element grösser ist als das gerade aktuelle. Geht es auf dem aktuellen Level nicht mehr weiter, steigen wir einen Level ab und suchen wie beschrieben weiter. Wenn wir auf Level 0 nicht mehr weiterkommen, müssen wir gerade vor dem gesuchten Element stehen, oder es gibt dieses Element gar nicht.

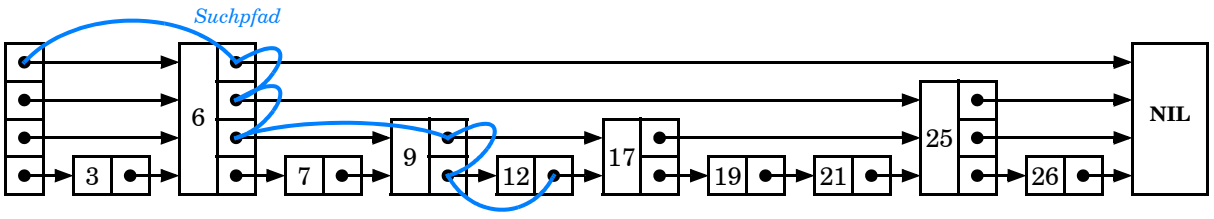
Einfügen: Um ein Element einzufügen, gehen wir genau gleich vor wie beim Suchen. Zusätzlich führen wir aber noch ein Array `update` mit, in dem all jene Elemente gespeichert werden, auf denen wir einen Level-Wechsel vorgenommen haben. Endet die Suche, haben wir entweder ein Element mit gleichem `m_key` gefunden und ersetzen es, oder wir haben den Ort gefunden, wo wir unser Element einfügen wollen. Beim Einfügen bestimmen wir zuerst den zufälligen Level des neuen Elementes. Danach setzen wir alle Referenzen des neuen Elementes so, dass sie dorthin zeigen, wo die Elemente in unserem `update`-Array hinzeigen. Zum Schluss müssen wir bloss noch die Referenzen der Elemente im `update`-Array auf das neue Element umbiegen.

Entfernen: Wiederum gehen wir genau gleich vor wie beim Suchen. Auch diesmal führen wir zusätzlich ein Array `update` mit, in dem alle Elemente gespeichert werden, auf denen wir einen Level-Wechsel vorgenommen haben. Da wir in der Skip-Liste einen `MaxLevel` haben, ist die Länge des Arrays `MaxLevel` + 1. Ist das zu entfernende Element gefunden, müssen wir alle Referenzen umbiegen, die auf dieses Element gerichtet sind. Dazu gehen wir unser `update`-Array durch und biegen alle Referenzen so um, dass sie dorthin zeigen, wo unser zu entfernendes Element hinzeigt. Danach können wir das zu entfernende Element einfach aus der Liste entfernen.

Bestimmen des zufälligen Levels: Beim Einfügen eines neuen Elementes wird für dieses zufällig ein Level bestimmt. Dabei muss aber die eingangs besprochene Verteilung der Levels erhalten bleiben (Level-0 → 100%, Level-1 → 50%, Level-2 → 25%, usw). Wenn wir nun zufällig entscheiden möchten, ob ein Element eine solche zusätzliche Referenz haben soll, können wir eine Münze werfen. «Kopf» bedeutet, das neue Element hat eine zusätzliche Referenz, «Zahl» bedeutet, dass das neue Element keine solche zusätzliche Referenz hat. Allgemein gesagt werfen wir so lange die Münze, bis das erste Mal «Zahl» kommt. Die Anzahl Münzwürfe bestimmt den Level des neuen Elementes. Die Formel lautet: $Level = AnzahlWürfe - 1$. (Tip: `Math.random()`).

2.3 Analyse

Die Operation *Suchen* ist entscheidend für das *Einfügen* und *Löschen*, weshalb wir uns auf die Analyse des Suchvorganges beschränken wollen. Die Skip-Liste ist zufällig aufgebaut und somit können wir nicht die generelle Länge des Suchpfades bestimmen, sondern nur seine *durchschnittlich erwartete* Länge Erfahrungswert).



Um den Erwartungswert für die Länge des Suchpfades zu berechnen, verfolgen wir den Suchpfad *rückwärts*, beginnend bei dem Element, das den gesuchten `m_key` enthält. Falls der gesuchte `m_key` nicht vorkommt, starten wir einfach nächst grösseren Element. Kurz: wir starten dort, wo die Operation *Suchen* aufhört. Allgemein ausgedrückt nehmen wir an, dass wir uns auf einem Element e befinden, das den Level i hat (mit $i = 0$).

E(k) ist die erwartete Anzahl Schritte, die man vom gesuchten Element e aus beim Zurückverfolgen des Suchpfades benötigt, um erstmals k Levels an Höhe zu gewinnen. Als Schritte zählen wir dabei jeweils das Hinaufklettern um einen Level und das Laufen nach links.

Wir haben angenommen, dass wir uns auf Knoten e mit Level i befinden. Aufgrund unseres Verfahrens zur Erzeugung eines zufälligen Levels eines Elements hat das Element e mit Wahrscheinlichkeit ½ einen höheren Level als i und mit Wahrscheinlichkeit ½ genau Level i. Diese zwei Fälle betrachten wir gesondert:

Fall 1: [i = `e.level`] impliziert, dass der zurückverfolgte Suchpfad vom Element e zu einem Element mit mindestens Level i geht und von diesem Knoten an immer noch k Level hinaufgeklettert werden müssen.

Fall 2: [i < `e.level`] impliziert, dass der zurückverfolgte Suchpfad beim Element e mindestens einen Level hinaufklettert anstatt einen Schritt nach links läuft. Also muss der Suchpfad von diesem neuen Level $i + 1$ aus noch $k - 1$ Levels hinaufklettern, um beim Zurückverfolgen insgesamt k Levels hinaufzusteigen. Damit erhalten wir folgende Rekursionsgleichung für $E(k)$: $E(k) = ½$ („Kosten um einen Zeiger nach links zu laufen“ + $E(k)$) + ½ («Kosten um ein Level hinaufzuklettern» + $E(k - 1)$)

Wenn wir die Schrittkosten auf 1 setzen und vereinfachen, erhalten wir: $E(k) = 2 + E(k - 1)$

Da die Anzahl Schritte beim Zurückverfolgen eines Pfades, der 0 Levels hinaufklettert, 0 ist gilt: $E(0) = 0$

Nach dem Auflösen der Rekursionsgleichung erhalten wir die einfache Lösung: $E(k) = 2k$

Kurz zur Wiederholung: Diese Formel gibt uns die erwarteten Kosten, um auf einem Suchpfad k Levels hinaufzuklettern. Wir verwenden diese Formel jetzt, um den Erwartungswert für die Kosten eines Suchpfades, abhängig von der Anzahl Elemente der Skip-Liste (und nicht der Levels) zu berechnen.

Bei der Idee der Skip-Liste sind wir zuerst von einer regelmässigen Referenz-Struktur ausgegangen. Wir haben dort gesagt, dass jedes 2^i-te Element eine Referenz auf Level i hat. Nehmen wir einfachheitshalber an, bei einer solchen perfekten Skip-Liste sei die Anzahl Elemente n eine Zweierpotenz. Das würde bedeuten, dass das letzte Element (Element Nr. n) den maximalen Level 2^i hat. Wenn wir die Gleichung $2^i = n$ nach i auflösen, ergibt das $i = \log_2(n)$. In unserer nicht perfekten Skip-Liste streben wir genau diese regelmässige Verteilung an und somit gilt diese Gleichung im Mittel auch für nicht perfekte Skip-Listen. Das heisst, wir haben eine Gleichung gefunden, die uns angibt, wie hoch unser `MaxLevel` in Abhängigkeit von der Anzahl der Elemente im Idealfall sein sollte.

Zurück zu der Rekursionsgleichung. $E(k)$ gibt uns die Kosten für das *Suchen* in Abhängigkeit des Levels an. Nun können wir einfach den `MaxLevel` einsetzen und erhalten die Formel, in Abhängigkeit der Anzahl Elemente n. Die Formel E[n], die uns die erwarteten Kosten für das Suchen eines Elementes in Abhängigkeit der Anzahl Elemente n in einer Skip-Liste angibt, ist also: $\mathrm{E}[n] = 2\ \mathrm{ld(n)}$

Aus den früheren Anmerkungen ist klar, dass auch die Kosten für das *Einfügen* und *Entfernen* von Elementen in Skip-Listen von derselben Grössenordnung sind.

Aufgaben

3.1 Entfernen

Studieren Sie die vorgegebene Implementierung der Skip-Liste (`SkipListe.java`). Die Implementierung ist fast vollständig, nur die Methode `void entfernen(int key)` fehlt noch. Implementieren Sie diese Methode. Zum Testen können Sie das Programm entweder normal als Applikation ausführen (über `main()`) oder aber als Applet, wobei Ihnen dann ein hübsches graphisches Rendering der aktuellen Situation zur Verfügung steht.

3.2 Entartung von Skip-Listen

Theoretisch kann eine Skip-Liste durch eine unglückliche Abfolge von Operationen zu einer linearen Liste entarten (nur noch Referenzen auf das direkt nächste Element vorhanden). Wieso ist dies ein in der Praxis vernachlässigbares Problem?

3.3 Verschmelzung von Skip-Listen

Formulieren Sie einen einfachen Algorithmus in Pseudocode, der zwei gegebene Skip-Listen zu einer neuen Skip-Liste vereinigt.