```
    return head.getNext() == tail;
  }
}
```

Da für die Klasse `DList` die gleichen Methoden wie für die Klasse `List` aus Abschnitt 13.2 definiert sind, lassen sich die Datenstrukturen Stack und Queue in gleicher Weise unter Benutzung der doppelt verketteten Liste implementieren. Für die Queue ergibt sich daraus der Vorteil, dass auch das Herausnehmen eines Elementes vom Ende der Liste mit konstantem Aufwand möglich ist.

Die Datenstruktur der doppelt verketteten Liste wird manchmal auch als *Deque* für »double-ended queue« bezeichnet. Hierbei handelt es sich um eine Warteschlange, die das Einfügen und Herausnehmen von Elementen an beiden Enden erlaubt.

Deque

13.4 Skip-Listen

Ein Nachteil von verketteten Listen gegenüber Feldern ist, dass die binäre Suche nicht einsetzbar ist, da man nicht beliebig »springen« kann. Eine Alternative ist es, mehrere Listen übereinanderzulegen, die dieses Springen ermöglichen. Eine derartige Datenstruktur wird als *Skip-Liste* bezeichnet.

Abbildung 13–6 zeigt eine Skip-Liste, die direkt die binäre Suche realisiert. Die Ebenen der Skip-Liste sind mit L_i bezeichnet, wobei die Ebene L_0 die eigentliche, sortiert abgespeicherte Liste darstellt. Die Listen der Ebene L_i springen jeweils 2^i Schritte weit und realisieren damit die binäre Aufteilung.

Skip-Liste

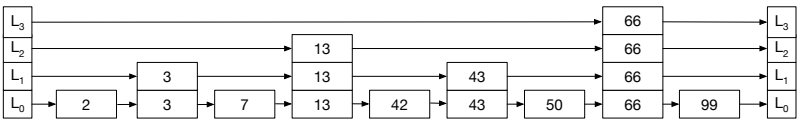

Abb. 13–6
Aufbau einer Skip-Liste

Es gibt jeweils einen initialen und einen terminalen Listenknoten, die die Werte $-\infty$ und $+\infty$ tragen. Grundsätzlich gibt es zwei Möglichkeiten zur Speicherung der Listenelemente auf mehreren Ebenen:

- Der Wert wird in separaten Listenelementen auf allen betroffenen Ebenen abgespeichert.
- Die Listenelemente werden verschmolzen, so dass Listenelemente mehrere Nachfolger haben können (ja nach abgedeckten Ebe-

nen). Auf diese Weise haben wir in der Abbildung diese Elemente schon zusammengeschoben.

In Abbildung 13–7 wird die Suche in einer solchen Liste skizziert. Man startet in der höchsten Ebene mit dem initialen Knoten. Wenn man einen Knoten ansieht, gibt es drei Möglichkeiten:

- Der aktuelle Wert ist der Suchwert. Dann ist man fertig.
- Der aktuelle Wert ist kleiner als der Suchwert. Dann wechselt man zum nächsten Knoten derselben Ebene.
- Der aktuelle Wert ist größer als der Suchwert. Befinden wir uns auf der Ebene L_0, ist der Suchwert nicht gespeichert. Befinden wir uns auf einer höheren Ebene, gehen wir zurück zum letzten Knoten und gehen dann eine Ebene tiefer.

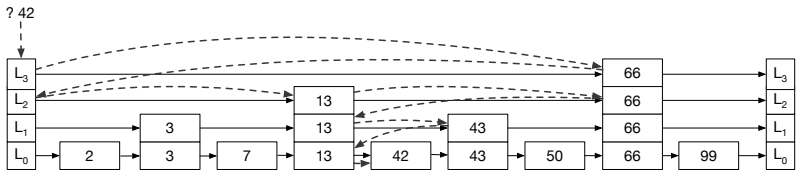

Abb. 13–7
Suche in einer Skip-Liste

Die bisher gezeigte Liste hat eine statische Struktur mit festen Schrittlängen, die durch das Einfügen und Löschen von Werten zerstört werden würde (etwa wenn die Werte 8 und 9 eingefügt werden). Allerdings ist der Algorithmus der Suche davon nicht betroffen: Selbst wenn 8 und 9 auf Ebene L_0 eingefügt würden, würde die Suche weiterhin funktionieren – jedoch bei der Suche nach der 9 mehrere Schritte auf der untersten Ebenen durchlaufen.

Daher reicht es aus, wenn wir nur eine gute Näherung an feste Schrittlängen haben. Eine Möglichkeit ist es dabei, neu eingefügte Werte zufällig einer Ebene zuzuweisen, wobei im Mittel die Hälfte der Werte der Ebene L_0 zugeordnet wird, ein weiteres Viertel der Ebene L_1, ein Achtel dann L_2 etc. Derartige Skip-Listen nennt man randomisiert. Abbildung 13–8 zeigt eine *randomisierte Skip-Liste*. Würde jetzt ein weiterer Wert hinzugefügt, würde er mit Wahrscheinlichkeit $\frac{1}{2^{i+1}}$ der Ebene L_i zugeordnet, wobei die Anzahl der Ebenen sich aus der Gesamtzahl der Werte ergibt.

Randomisierte Skip-Liste

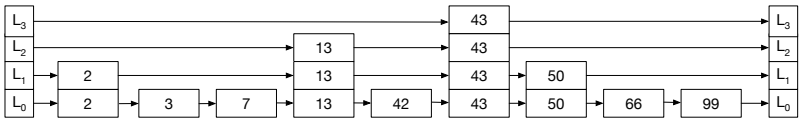

Abb. 13–8
Randomisierte Skip-Liste

Die zuerst eingeführten Skip-Listen mit Zweierpotenzen als festen Schrittlängen werden zur Abgrenzung von randomisierten Skip-Listen als *perfekte Skip-Listen* bezeichnet. Sie können insbesondere genutzt werden, wenn die Werte der Skip-Listen nicht mehr geändert oder ergänzt werden.

Betrachten wir eine einfache Implementierung der Skip-Liste. Zunächst benötigen wir eine Knotenklasse, die neben Vorgänger und Nachfolger auch die Verweise auf die Knoten der darunter liegenden Ebene L_{i-1} (down) und der darüber liegenden Ebene L_{i+1} (up) enthält. Diese Klasse ist in Programm 13.11 die Klasse SLItem, die als Element **int**-Werte enthalten kann. Wir implementieren dabei die Knotenvariante, bei der die Werte auf allen betroffenen Ebenen separat gespeichert sind:

Programm 13.11
Implementierung der Skip-Liste

```
public class SkipList {
  public static int negInf = Integer.MIN_VALUE; // -inf
  public static int posInf = Integer.MAX_VALUE; // +inf

  static class SLItem {
    public SLItem(int i) { element = i; }

    int element; // Element
    SLItem up, down, // Li+1 und Li-1
      prev, next; // Vorgänger, Nachfolger
  }

  SLItem head, tail;
  ...
  public boolean find(int o) {
    SLItem item = head;
    while (true) {
      // zunächst nach rechts suchen ...
      while (item.next.element != posInf &&
        item.next.element <= o)
        item = item.next;
      if (item.down != null)
        // eine Ebene nach unten
        item = item.down;
      else
        // Ebene L0 erreicht
        break;
    }
    return item.element == o;
  }
}
```

Weiterhin benötigen wir noch die `head`- und `tail`-Knoten sowie die speziellen Werte für $-\infty$ und $+\infty$, die wir als kleinsten bzw. größten `int`-Wert repräsentieren.

In Programm 13.11 ist nur die Suche über die Skip-Liste mit der Methode `find` dargestellt. Ausgehend vom `head`-Knoten wird die oberste Ebene zunächst nach rechts durchlaufen, solange der Knotenwert $\leq$ dem gesuchten Wert ist. Erreicht man einen Knoten, dessen Wert größer ist, wird versucht, nach unten zu gehen. Ist dies nicht mehr möglich, weil die L_0-Ebene erreicht ist, wird die Schleife mittels `break` abgebrochen. Da auf jeden Fall am Ende ein Knoten erreicht wird, muss abschließend noch geprüft werden, ob dessen Wert tatsächlich dem gesuchten Wert entspricht.

13.5 Das Iterator-Konzept

Durchwandern einer Kollektion

Die Implementierungen der Listen aus den vorigen Abschnitten weist noch ein Manko auf, welches die praktische Verwendbarkeit einschränkt. So ist es oft notwendig, eine Kollektion zu »durchwandern«, d.h. über alle Elemente zu navigieren. Dieses Navigieren ist zunächst abhängig von der Implementierung: Während beispielsweise ein Feld mittels einer Indexvariablen durchlaufen wird, ist für verkettete Listen das Verfolgen der `next`-Zeiger der einzelnen Knoten notwendig. Auch im Hinblick auf die Erhaltung des Prinzips der Kapselung ist daher ein Konzept wünschenswert, das eine einheitliche Behandlung des Navigierens unabhängig von der internen Realisierung unterstützt. In Java wird dieses Konzept durch *Iteratoren* verwirklicht. Hierbei handelt es sich um Objekte zum Iterieren über Kollektionen, deren Klasse die vordefinierte Schnittstelle `java.util.Iterator` implementiert. Ein Iterator verwaltet einen internen Zeiger auf die aktuelle Position in der zugrunde liegenden Datenstruktur. Auf diese Weise ist es möglich, dass mehrere Iteratoren gleichzeitig auf einer Kollektion arbeiten (Abbildung 13–9).

Iterator

Die Schnittstelle `java.util.Iterator` definiert die folgenden Methoden:

- **`boolean`** `hasNext()` prüft, ob noch weitere Elemente in der Kollektion verfügbar sind. In diesem Fall wird `true` geliefert. Ist dagegen das Ende erreicht, wird `false` zurückgegeben.
- `Object next()` liefert das aktuelle Element zurück und setzt den internen Zeiger des Iterators auf das nächste Element.