

Algorithmen & Datenstrukturen 2

Jan Fässler

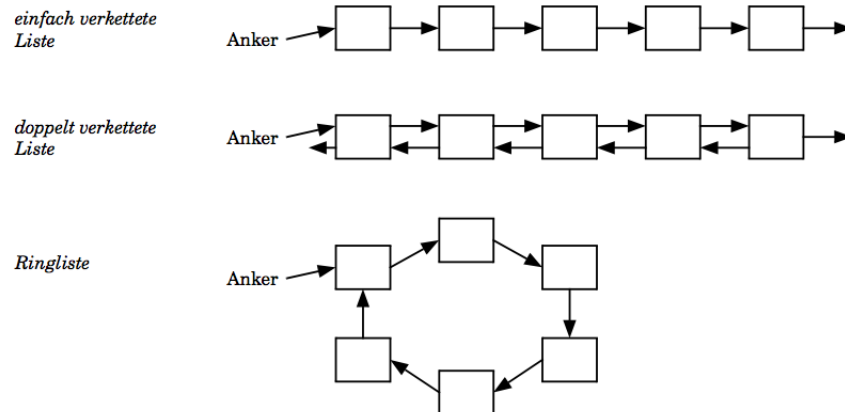
3. Semester (HS 2012)

Inhaltsverzeichnis

1	Listen	1
1.1	Stack	2
1.2	Erweiterte Liste	2
1.2.1	Iterators	4
1.2.2	Merge Sort	5
1.3	Skip-Liste	5
1.3.1	Suchen	5
1.3.2	Einfügen	6
1.3.3	Entfernen	6

1 Listen

Eine verkettete Liste (linked list) ist eine dynamische Datenstruktur zur Speicherung von Objekten. Sie eignen sich für das Speichern einer unbekannten Anzahl von Objekten, sofern kein direkter Zugriff auf die einzelnen Objekte benötigt wird. Jedes Element in einer Liste muss neben den Nutzinformationen auch die notwendigen Referenzen zur Verkettung enthalten. Es gibt drei verschiedene Arten von Listen:



Listing 1: einfache Linked List

```
1 public class LinkedList<T> {
    private Element<T> head = null;
    private Element<T> last = null;
    public void add(T data) {
        last.next = new Element<T>(data);
6        last = last.next;
    }
    public void remove(T data) {
        Element<T> current = head;
        while (current != null && current.data != data) {
11        current = current.next;
            if (current != null && current.data == data) {
                current.last = current.next;
                current = null;
            }
16    }
    }
    public T getFirst() {
        return head.data;
    }
21    public T getLast() {
        return last.data;
    }
    public class Element<E> {
        public Element<E> next;
26        public Element<E> last;
        public E data;
        public Element(E input) {
            data = input;
        }
31    }
}
```

1.1 Stack

Der Stack ist eine dynamische Datenstruktur bei der man nur auf das oberste Element des Stabels zugreifen (top), ein neues Element auf den Stabel legen (push) oder das oberste Element des Stapels entfernen (pop) kann.

Listing 2: Implementierung eines Stacks

```
public class Stack<T> extends LinkedList<T> {
    public T top() {
3      return this.getLast();
    }
    public void push(T data) {
        this.add(data);
    }
8   public T pop() {
        T last = this.getLast();
        this.remove(last);
        return last;
    }
13  public boolean isEmpty() {
        return this.getFirst() == null ? true : false;
    }
}
```

1.2 Erweiterte Liste

Dies ist mal eine mögliche und vor allem nur teilweise Implementierung einer doppelt verlinkten Liste. Die Implementierung des Iterators und der Sortierung sind ausgeklammert in Unterkapitel.

Listing 3: Liste mit Iterator

```
public class AdvancedComparableList<T> extends Comparable<T> implements
    Iterable<T> {
    private ListElement<T> head, foot;
    private int size = 0;
4   public T getFirst() { return head.data; }
    public T getLast() { return foot.data; }
    public int size() { return size; }
    public boolean contains(T data) {
        boolean found = false;
9       CLISTIterator<T> it = this.iterator();
        while (!found && it.hasNext()) if (it.next().equals(data)) found = true;
        return found;
    }
    public void add(T data) { add(size, data); }
14  public void add(int index, T data) {
        if (index > size) throw new IndexOutOfBoundsException();
        else if (!this.contains(data)) {
            ListElement<T> newElement = new ListElement<T>(data);
            ListElement<T> current = head;
19          if (size == 0) {
                head = newElement;
                foot = head;
            } else if (index == size) {
                newElement.last = foot;
24          foot.next = newElement;
                foot = newElement;
            } else if (index == 0) {
```

```

        newElement.next = current;
        current.last = newElement;
29     head = newElement;
    } else {
        for (int i=0; i<index; i++) current = current.next;
        newElement.next = current;
        newElement.last = current.last;
34     if (current.last != null) current.last.next = newElement;
        current.last = newElement;
    }
    size++;
}
39 }

public T remove() { return remove(size-1); }
public T remove(T data) throws Exception {
    if (this.contains(data)) return remove(this.indexOf(data));
    else throw new Exception("Element "+data+" does not exist.");
44 }

public T remove(int index) {
    if (index >= size) throw new IndexOutOfBoundsException();
    T element = get(index);
    if (index == 0) {
49     if (size > 1) head = head.next;
        else { head = null; foot = null; }
    } else if (index == (size-1)) {
        foot.last.next = null;
        foot = foot.last;
54     } else {
        ListElement<T> current = head;
        for (int i=0; i<index; i++) current = current.next;
        current.last.next = current.next;
        if (current.next != null) current.next.last = current.last;
59     current = null;
    }
    return element;
}

public int indexOf(T data) {
64     int index = -1;
    if (this.contains(data)) { index++; while(!this.get(index).equals(data))
        index++; }
    return index;
}

public T get(int index) {
69     T element = null;
    if (index >= 0 && index < size) element = this.iterator(index).next();
    return element;
}

public class ListElement<T extends Comparable<T>> implements Comparable<T>
{
74     public ListElement<T> next, last;
    public T data;
    public ListElement(T input) { data = input; }
    public int compareTo(T o) { return data.compareTo(o); }
    public String toString() { return String.valueOf(data) + ">" + String.
        valueOf(next); }
79 }
}

```

1.2.1 Iterators

Die Schnittstelle `java.util.Iterator`, erlaubt das Iterieren von Containerklassen. Jeder Iterator stellt Funktionen namens `next()`, `hasNext()` sowie eine optionale Funktion namens `remove()` zur Verfügung. Der folgende `ListIterator` stellt auch noch Funktionen für rückwärtsiterieren zur Verfügung, sowie die Möglichkeit den aktuellen Index abzufragen. Zudem kann damit noch direkt über den Iterator Elemente eingefügt oder ersetzt werden.

Listing 4: Iterators

```
public CListIterator<T> iterator() { return new CListIterator<T>(head,
    this); }
public CListIterator<T> iterator(int index) {
    if (index >= size) throw new IndexOutOfBoundsException();
    CListIterator<T> it = this.iterator();
5    for (int i=0; i<index; i++) it.next();
    return it;
}
public static class CListIterator<E extends Comparable<E>> implements
    ListIterator<E> {
    private ListElement<E> nextElement, prevElement, lastReturned;
10    private AdvancedComparableList<E> _list;
    private int index = 0;
    public CListIterator(ListElement<E> element, AdvancedComparableList<E>
        list) {
        _list = list;
        nextElement = element;
15    prevElement = (element == null?null:element.last);
        ListElement<E> current = element;
        while (current != null && current.last != null) {
            index++;
            current = current.last;
20    }
    }
    public boolean hasNext() { return nextElement != null; }
    public E next() {
        index++;
25    prevElement = nextElement;
        nextElement = nextElement.next;
        lastReturned = prevElement;
        return prevElement.data;
    }
30    public boolean hasPrevious() { return prevElement != null; }
    public E previous() {
        index--;
        nextElement = prevElement;
        prevElement = prevElement.last;
35    lastReturned = nextElement;
        return nextElement.data;
    }
    public int nextIndex() { return index; }
    public int previousIndex() { return index-1; }
40    public void add(E data) { _list.add(previousIndex(), data); lastReturned
        = null; }
    public void remove() { _list.remove(previousIndex()); lastReturned =
        null; }
    public void set(E e) { lastReturned.data = e; }
}
```

1.2.2 Merge Sort

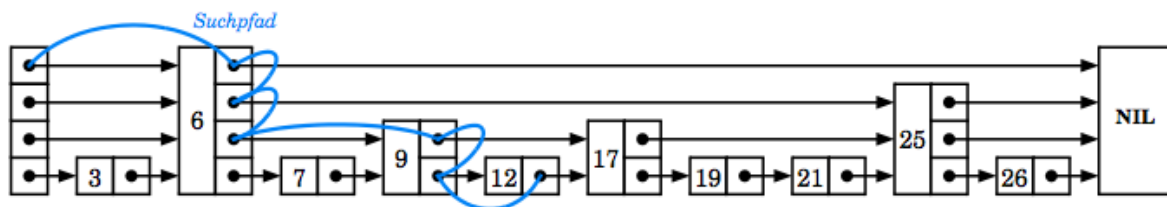
Listing 5: Merge Sort

```
/*      sorting      */
2  public void sort() { mergesort(0, this.size() - 1); }
   private void mergesort(int low, int high) {
       if (low < high) {
           if (high-low > 1) {
               int middle = (low + high) / 2;
               mergesort(low, middle);
               mergesort(middle + 1, high);
               merge(low, middle, high);
           } else {
               if (this.get(low).compareTo(this.get(high)) > 0) {
12                  T tmp = this.get(high);
                     this.remove(high);
                     this.add(low, tmp);
               }
           }
17     }
   }

   private void merge(int low, int middle, int high) {
       int iLeft = low, iRight = middle+1;
       while (iLeft <= high && iRight <= high) {
22         T right = get(iRight);
           if (get(iLeft).compareTo(right) > 0) {
               remove(iRight);
               add(iLeft, right);
               iRight++;
27         } else iLeft++;
       }
   }
}
```

1.3 Skip-Liste

Die Skip-Liste ist eine sortierte, einfach verkettete Liste, die uns aber ein schnelleres Suchen von Elementen in der Datenstruktur erlaubt. In einer sortierten, verketteten Liste müssen wir jedes Element einzeln durchlaufen bis wir das gewünschte Element gefunden haben. 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. Genau betrachtet müssen wir nie mehr als $(n/2) + 1$ Elemente besuchen (n ist die Länge der Liste).



1.3.1 Suchen

Um ein Element zu finden, gehen wir so lange auf dem momentanen Level den Referenzen entlang, wie das zu suchende Element immer noch grösser ist als das gerade aktuelle. Geht es auf dem momentanen Level nicht mehr weiter, gehen wir einfach einen Level tiefer und suchen

wie beschrieben weiter. Wenn wir auf Level 0 nicht mehr weiterkommen, müssen wir gerade vor dem gesuchten Element sein, oder es gibt dieses Element gar nicht.

1.3.2 Einfügen

Um ein Element einzufügen, gehen wir genau gleich wie beim Suchen vor. 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 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.

1.3.3 Entfernen

Wiederum gehen wir genau gleich wie beim Suchen vor. Auch diesmal führen wir zusätzlich noch ein Array update mit, in dem all jene 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 $\text{MaxLevel} + 1$. 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 entnehmen.