Belegaufgaben Smalltalk

Ausgabe: 21. April 1998

Abgabe: 31. Mai 1998

Abnahme der Implementationen Anfang Juni 1998 nach Vereinbarung.

Aufgabenbeschreibung:

- Wähle eines der Themen:
  - Doppelt verkettete Liste,
  - Binärer Suchbaum,
  - Sortierverfahren 1 oder 2,
  - Huffman-Codierung oder
  - Splaybaum.
- Arbeite Dich anhand der beiliegenden Informationen in das gewählte Thema ein.
- Implementiere ein Programm für die in dem ausgewählten Thema angesprochenen Methoden und teste dieses aus.
- Fertige dazu eine schriftliche Ausarbeitung an:
  - Beschreibe die implementierten Methoden und kommentiere diese **sinnvoll**.
  - Zeige die korrekte Funktion Deines Programms anhand eines Testdurchlaufs mit Ausgabe.

1 Doppelt verkettete Liste

Implementiere eine Klasse „DoppeltVerketteteListe", sowie ein Demo-Programm, das diese Datenstruktur und die Operationen darauf demonstriert. Folgende Operationen sind *mindestens* zu realisieren:

- Erzeugen einer leeren Liste
- Anfügen eines Elementes am Anfang der Liste
- Anfügen eines Elementes am Ende der Liste
- Einfügen eines Elementes vor/nach einem bestimmten Element
- Löschen eines Elementes
- Suchen nach einem Element
- Ausführen einer Operation (Block) auf allen Elementen in der vorliegendenReihenfolge sowie in umgekehrter Reihenfolge (`do: reverseDo:`)
- `select:`, `collect:`, `reject:`, `detect:`; – die ersten drei Methoden erzeugen dabei als Ergebnis wieder eine doppelt verkettete Liste

2 Binärer Suchbaum

Implementiere eine Klasse „BinaererSuchBaum", sowie ein Demo-Programm, das die Datenstruktur und die Operationen darauf demonstriert. Orientiere Dich dabei an dem Mechanismus des SortBlocks in SortedCollection. Folgende Operationen sind *mindestens* zu realisieren:

- Erzeugen eines leeren Baumes
- Einfügen eines Knotens
- Löschen eines Knotens
- Suchen nach einem Knoten
- Ausführen einer Operation (Block) auf allen Knoten, dabei `preOrderDo:`, `inOrderDo:` und `postOrderDo:`
- `select:`, `reject:`, `collect:`, `detect:` – die ersten drei Operationen erzeugen dabei wieder einen binaeren Suchbaum.
- Übersichtliche (TABs und Leerzeichen) Ausgabe des Baumes auf das Transcript oder in eine Textdatei

3 Sortierverfahren

Implementiere ein Demo- und Test-Programm für verschiedene Sortierverfahren.

Sortierverfahren 1: Implementiere folgende Verfahren:

- InsertionSort
- BubbleSort
- HeapSort
- SelectionSort
- QuickSort

Sortierverfahren 2: Implementiere folgende Verfahren:

- MergeSort
- SelectionSort
- RadixExchangeSort
- Distribution Counting
- QuickSort

Messe jeweils die Zeiten für die Sortierung einer (fast) sortierten Folge, einer umgekehrt sortierten Folge sowie einer zufällig erzeugten Folge. Biete dem Benutzer eine Auswahl der Algorithmen oder einen vollständigen Vergleich aller Algorithmen an. Stelle die Ergebnisse übersichtlich dar und erlaube die Auswal der Größe der zu sortierenden Folgen. Literatur zu den Verfahren RadixExchange und Distribution Counting ist z.B. im Buch „Algorithmen“ von R. Sedgewick zu finden.

4 Huffman-Codierung

4.1 Allgemeines

Gegeben sei ein Text T über einen Alphabet A. Für jeden Buchstaben $a \in A$ sei $H(a)$ die Häufigkeit mit der der Buchstabe a im Text T vorkommt. Es wird angenommen, daß $H(a) > 0$ für jeden Buchstaben $a \in A$ und daß A aus mindestens zwei Buchstaben besteht. Das Ziel ist die Konstruktion eines binären Präfix-Codes für T. Dabei soll der codierte Text T minimale Länge haben.

Ein binärer Code $c : A \to \{0,1\}^*$ weist jedem Buchstaben $a \in A$ einen Bitstring $c(a) \in \{0,1\}^*$ zu. Eine Codierung ist eine Präfix-Codierung, wenn für einen Code $c : A \to \{0,1\}^*$ gilt, daß $c(a)$ kein Präfix von $c(b)$ für $a, b \in A; a \neq b$ ist.

Für einen Text $T = a_1, \ldots, a_n$ mit $a_1, \ldots, a_n \in A$ gilt $c(T) = c(a_1) \ldots c(a_n)$. Die Länge L des codierten Textes T sei $L = \sum_{a \in A} H(a) * |c(a)|$.

Ein Huffman-Code für T ist ein Präfix-Code minimaler Länge.

Ein Huffman-Code kann mittels eines binären Baums darstellt werden. Ein solcher Baum kann für die Decodierung hilfreich sein. Ein solcher Baum wird so organisiert, daß

- alle inneren Knoten genau 2 Kinder besitzen,
- jede Kante mit 0 oder 1 markiert ist,
- jeder Buchstabe $a \in A$ genau einem Blatt b_a zugewiesen ist, so daß die Markierungen des Weges von der Wurzel zu b_a mit dem Code c(a) übereinstimmen.

Allerdings stellt nicht jeder durch einen Binärbaum repräsentierte Code einen Huffman-Code dar.

4.2 Algorithmus

Der Algorithmus wird einen Huffman-Code ermitteln, der durch einen Binärbaum repräsentiert wird.

- Füge alle Buchstaben $a \in A$ in eine Liste ein, die nach dem kleinsten Häufigkeitswert geordnet wird.
- Solange die Liste mindestens zwei Buchstaben enthält, wiederhole:
  - Entferne die beiden Buchstaben α und β mit der geringsten Häufigkeit aus der Liste.
  - Füge einen neuen Buchstaben γ in die Liste gemäß $H(\gamma)$ ein, dabei wird $H(\gamma) = H(\alpha) + H(\beta)$ gesetzt.
  - Schaffe einen Knoten v_γ für γ und setze v_α und v_β als Kinder von v_γ. Markiere die Kante $\{v_\alpha, v_\gamma\}$ mit 0 und die Kante $\{v_\beta, v_\gamma\}$ mit 1.

4.3 Beispiel

Der Buchstabe C wird mit dem Codebaum in der Abbildung 1 als 011 codiert. Die Wurzel ist der Startpunkt, dann wird der Baum durchgegangen bis ein Blatt erreicht ist. Dabei wird der Weg anhand der inneren Knoten gewählt: Ist ein C im rechten Kind des Knotens, gehe zum rechten Kind, sonst zum linken. Die dabei durchlaufenden Kantenmarkierungen bilden das Codewort.
Die Eingabe 10100 wird als FA decodiert. Start ist wieder die Wurzel. Ein Weg wird anhand der jeweiligen Ziffer und der dazu passenden Kantenmarkierung gewählt. Wird ein Blatt erreicht, ist ein Buchstabe decodiert und es wird wieder mit bei Wurzel gestartet. In diesem Fall ist dies $101 \rightarrow F, 00 \rightarrow A$.

4.4 Smalltalk-Implementation

Implementiere eine Klasse „Huffman-Code“ mit folgenden Methoden.

Quellalphabet	A	B	C	D	E	F
Wahrscheinlichkeiten	0.4	0.1	0.1	0.1	0.2	0.1

Aufbau des Codierungsbaums:

Start: Liste: A - E - B - C -D -F

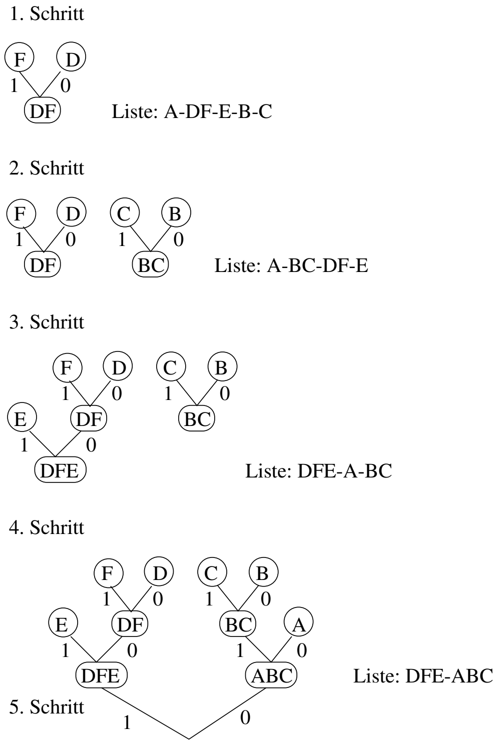


Abbildung 1: Beispiel für Aufbau eines Huffman-Codebaums

`initialize: alphabet with: probabilities`

`printCodeTree`

Die Methode `initialize` soll anhand einer Eingabe eines Alphabets (Array alphabet) und einer dazu gegebenen Häufigkeit (Array probabilities) einen Codierungsbaum aufbauen, die Methode `printCodeTree` soll diesen im Workspacefenster ausgeben (Einfache Ausgabe, keine Graphik).

`code: string`

`decode: string`

Die Methode `code` soll einen String mit Zeichen aus dem Alphabet gemäß des geschaffenen Codierungsbaum codieren, die Methode `decode` einen String aus Einsen und Nullen decodieren. Dabei soll der codierte/decodierte String im Workspacefenster angezeigt werden.

Es ist sinnvoll für die Blätter des Binärbaum eine eigene (kleine) Klasse zu implementieren. Desweiteren dürfen die Klassen SortedCollection, IdentityDictonary und Association benutzt werden. Die Liste (s. Algorithmus) soll nicht extra implementiert werden, sondern mit Hilfe der genannten Klassen organisiert werden.

5 Splay-Bäume

Splay-Bäume sind eine effiziente Datenstruktur für Wörterbuchabfragen. Splay-Bäume „passen sich den Daten an": Inhalte (oder auch Schlüssel) einzelner Knoten, die relativ oft abgefragt werden, werden zur Wurzel des Baumes bewegt (selbstorganisierende Datenstruktur). Im allgemeinen wird nicht erwartet, daß alle Schlüssel mit gleicher Wahrscheinlichkeit abgefragt werden, vielmehr werden einige Schlüssel öfter als andere abgefragt. Splay-Bäume stellen sich auf solche Fälle ein.
Die Struktur eines Splay-Baumes ist die eines binären Suchbaumes. Es wird pro Knoten nur der Schlüssel (und ggf. die Information) abgespeichert. Die Idee der Selbstorganisation ist: Wenn nach einem Schlüssel x gefragt wird, dann wird dieser Schlüssel mit Rotationen zur Wurzel gebracht. Nachfolgende Aktionen für andere Schlüssel lassen Schlüssel x wieder im Baum langsam „nach unten sacken". Wird aber Schlüssel x nach kurzer Zeit wieder abgefragt, wird Schlüssel x relativ nah zur Wurzel sein, und es nicht viel Zeit kosten, ihn wieder mittels binärer Suche zu finden.
Splay-Bäume arbeiten mit einer sogenannten splay-Operation. Für einen Schlüssel x verläuft splay(x) wie folgt:

- Zuerst wird eine binäre Suche nach x gestartet. Die binäre Suche wird mit einem Schlüssel y terminieren. Falls die Suche erfolgreich war, ist x=y, sonst ist y der kleinste Schlüssel, der größer ist als x, bzw. der größte Schlüssel, der kleiner ist als x.
- Der Schlüssel y wird neue Wurzel.

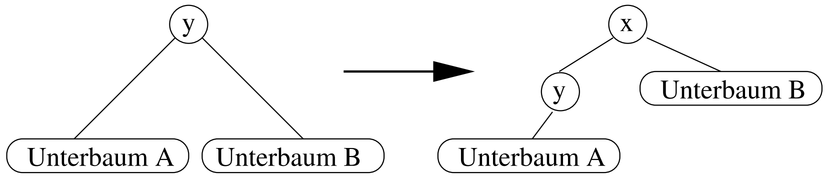


Abbildung 2: Einfügen des Schlüssel x in den Splay-Baum

Die folgenden Operationen benutzen die splay-Operation:

- lookup(x) sieht nach, ob der Schlüssel x im Baum ist: Führt splay(x) aus und überprüft die Wurzel.
- insert(x, info) fügt einen Schlüssel x mit dazugehörender Information info in den Baum ein: Führt splay(x) durch. Falls die Wurzel den Schlüssel speichert, wird der alte Informationteil mit dem neuen überschrieben, ansonsten wird die Wurzel einen Schlüssel y speichern. Falls $y < x$, dann füge x wie in Abbildung 2 gezeigt ein, analog ist $y > x$ zu behandeln.

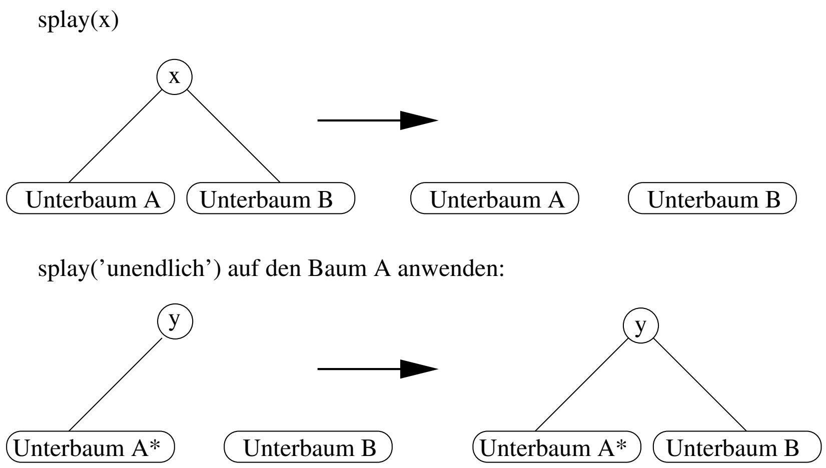


Abbildung 3: Entfernen des Schlüssel x aus dem Splay-Baum

- remove(x) entfernt den Schlüssel x aus dem Baum. Die Operation splay(x) wird ausgeführt, und falls ein Schlüssel x existiert, wird wie in Abbildung 3 mit einer weiteren splay-Operation splay(∞) vorgegangen.

Die einzelnen Fallunterscheidungen und Verfahrensweise der splay-Operation zeigt Abbildung 4.

5.1 Smalltalk-Implementation

Implementiere eine Klasse „Splay-Baum" mit folgenden Methoden:

`rotiereRechts: knoten`

`rotiereLinks: knoten`

`splay: knoten`

`insert: key`

`remove: key`

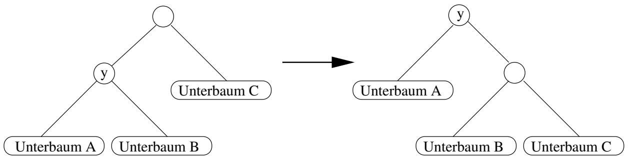
`lookup: key`

`depth`

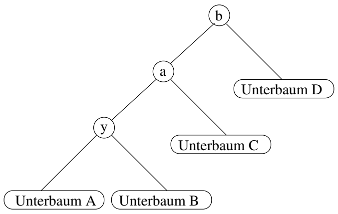
Die Methoden `rotiereRechts, rotiereLinks, splay, insert, remove` und `lookup` sollen die beschriebenen Funktionen ausführen. Die Methode `depth` soll die Tiefe des Splay-Baumes berechnen und ausgeben.

Eine Klasse Knoten ist hier sinnvoll. Ein Objekt Knoten sollte neben Variablen für den Schlüssel und der Kinder noch eine Variable für den Vater des Knotens im Baum haben (für die Wurzel ist dieser `nil`), dies kann die Implementation vereinfachen.
Ein Testdurchlauf soll die sortierte Folge 1,2, …100 in einen leeren Splay-Baum einfügen und für `lookup`: 1,2, …100 die Tiefe des Splay-Baumes ausgeben.

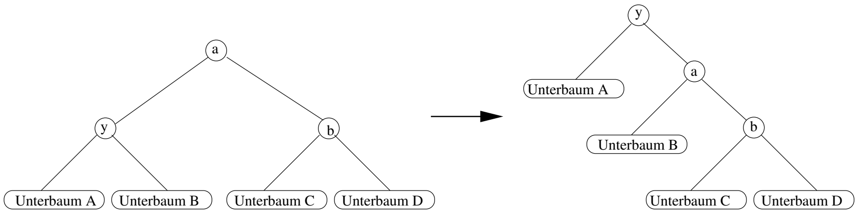
Fall 1: y ist Kind der Wurzel, entsprechende Rotation anwenden



Fall 2: y hat Tiefe mindestens 2 und folgendes Aussehen

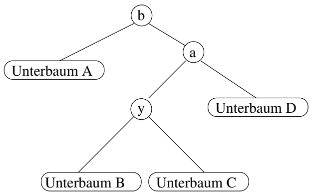


Also eine Rechtsrotation in b, gefolgt von einer Rechtsrotation in a

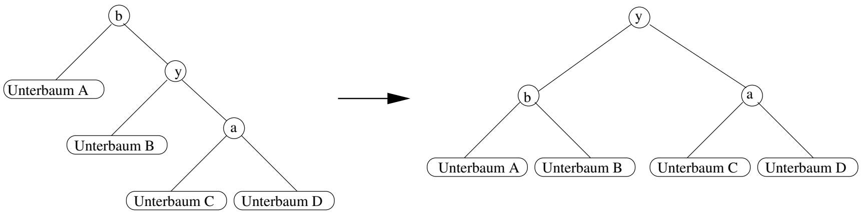


Fall 3: y hat Tiefe mindestens 2 und ist achsensymmetrisch zu Fall 2, analoges Vorgehen

Fall 4: y hat Tiefe mindestens 2 und folgendes Aussehen



Rechtsrotation in a, gefolgt von einer Linksrotation in b



Fall 5: y hat Tiefe mindestens 2 und ist achsensymmetrisch zu Fall 4, analoges Vorgehen

Abbildung 4: Aufruf von splay(y)