II.3.2 Rekursive Datenstrukturen

Freitag, 17. November 2017 09:30

Rekursion kann nicht nur für Algorithmen benutzt werden, sondern auch zur Definition v. Datenstrukturen.
(z.B. Syntaxdiagramm für Ausdrücke in Java)

- Rekursiv def. Datenstrukturen eignen sich zur Repräsentation, deren Größe vorher nicht bekannt ist bzw. deren Größe sich ändert (<u>dynamische</u> Datenstrukturen).

- Bsp: Liste (Vorteil gegenüber Arrays: Größe kann sich beliebig ändern)
  Bäume, Graphen, ...

- Man muss jeweils Algorithmen schreiben, um in solchen Datenstrukturen
  - Elemente einzufügen
  - " zu löschen
  - " zu suchen
  - etc.

- Generelle Idee: Benutzt Referenzen (Zeiger/Pointer), so dass Elemente auf andere Elemente zeigen können. Hierdurch kann man Zeigerkette bel. verlängern u. Objekte können bel. groß werden.

- Bsp: Liste
  - Jedes Listenelement hat einen Wert und einen Nachfolger
  - Liste kann auch leer sein.

(Grammatik: Liste = Element Liste | leer)

- Klasse Element ist "rekursiv", da sie ein Attribut next hat, das wieder vom Typ Element ist.
- Nachteil, falls man Listen nur mit der Klasse "Element" implementiert: Leere Liste wird durch null dargestellt. Problem, wenn man (nicht-statische) Methoden schreibt, die auch auf der leeren Liste arbeiten sollen.

  z.B. e.fuegeVorneEin(...) ist nicht möglich falls e null ist.

• Lösung: Definiere weitere Klasse Liste, mit Attribut Kopf vom Typ Element (für das 1. Element der Liste).

  ⇒ Leere Liste ist jetzt ein Objekt l vom Typ Liste mit l.kopf = null.

• Datenkapselung: Attribut kopf sollte private sein

  (Hier: Klasse Element ist nur im gleichen Paket sichtbar, wert + next sollten private sein u. über entspr. Selektoren zugreifbar sein) ← aus Platzgründen nicht auf der Folie

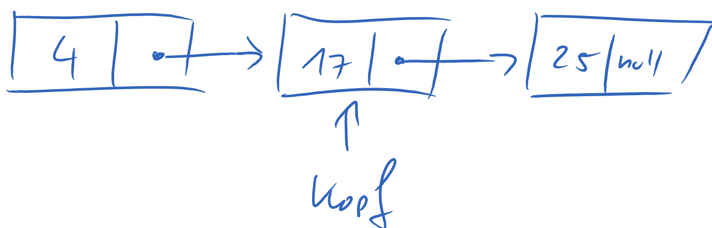
Rekursive Algorithmen eignen sich besonders gut zur Bearbeitung v. rek. Datenstrukturen.

Entwerfe erst Schnittstellendokumentation mit den wichtigsten

Operationen auf Listen.
Ausschließend Implementierung mit rek. Algorithmen.

Suche: sucht nach 1. Element in der Liste mit vorgeg. Wert.
Wenn es kein solches El. gibt, wird null zurückgegeben.

- benutze statische Hilfsmethode suche(wert, e), die nach "wert" in der Teilliste, die mit Element e beginnt. Bsp: suche(17, kopf)



Strukturelle Rekursion:

Nicht-rek. Attribute werden im Methodenrumpf behandelt.
Rek. Attribute werden im rek. Aufruf d. Methode behandelt.

toString / durchlaufe / drucke

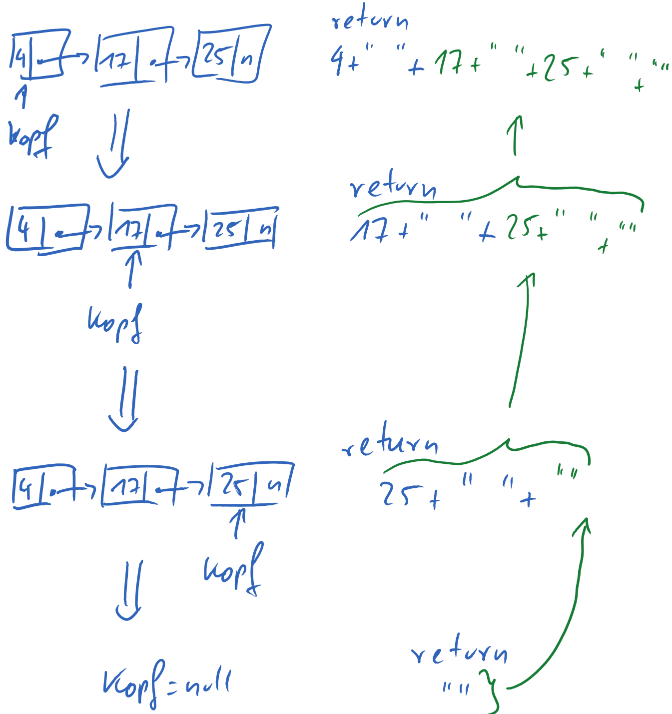
toString: erzeugt einen String
(....)
Elemente d. Liste

durchlaufe(e): erzeugt String aller Werte derjenigen Liste,

die mit e beginnt

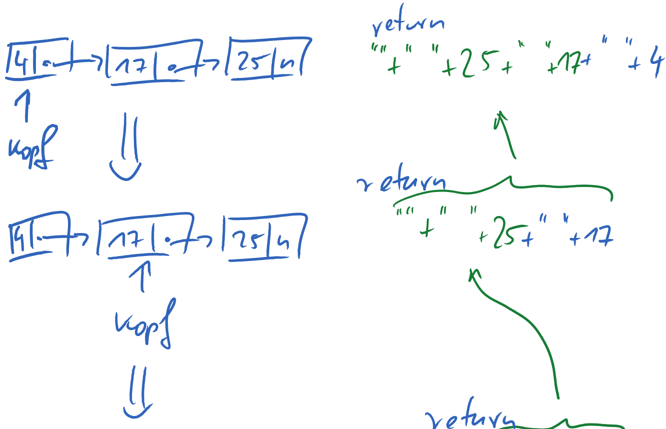
l.drucke() : ruft System.out.
println(l) auf. Dieses
führt l.toString() aus.

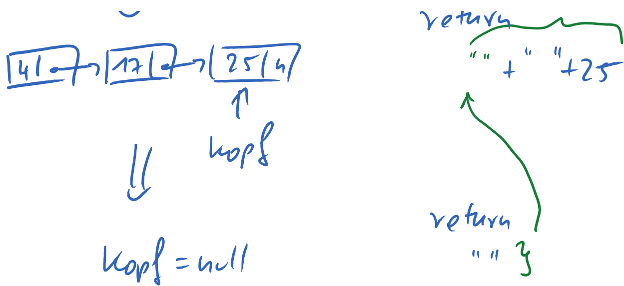
Methode "durchlaufe" ist rekursiv:



toStringRueckwaerts / durchlaufeRueckw.
druckeRueckw.

Soll Elemente von hinten n. vorn
ausgeben
Wie arbeitet durchlaufeRueckwaerts?



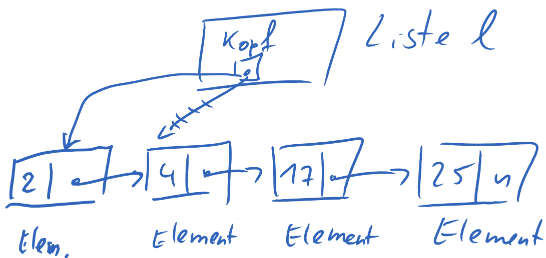


fuegeVorneEin

fügt ein Element mit vorgegebenem Wert vorne in Liste ein

Falls Liste bisher nicht leer war:

l.fuegeVorneEin(2):



"kopf = ..." ist nötig, damit das kopf-Attribut der Liste auf das neue Element zeigt.

l.fuegeSortiertEin(x)

fügt den Wert x vor dem ersten Element in der Liste l ein, das einen größeren Wert als x hat. Sonst wird x am Ende eingefügt.

⇒ Wenn l vorher aufsteigend sortiert war, dann ist l hinter-

her immer noch sortiert.

Statische Hilfsmethode

fuegeSortiertEin (x, e):

- fügt x vor dem ersten größeren Element in der Liste ein, die mit e beginnt
- Ergebnis ist das erste Element dieser Liste, die durch Einfügen von x entsteht.

1. Fall Wenn e leer ist, dann ist das Ergebnis:

| x | null |

2. Fall Wenn der Wert y des ersten Elements e größer ist als der einzufügende Wert x, dann ist das Ergebnis:

| x | ⟶ | y | ⟶ | | ⟶ ...
↑ e (zeigt auf y)

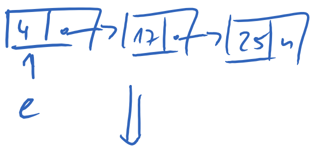
3. Fall Der einzufügende Wert x wird hinter dem ersten Element e eingefügt. Ergebnis ist:

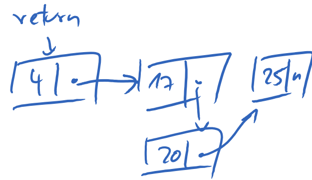
| | ⟶ | | ⟶ ..... ⟶ | x | ⟶ | | ⟶ ....
↑ e (zeigt auf das erste Element)

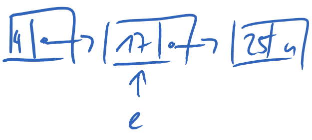
fuegeSortiertEin (x, e.next) (für den Teil ab dem zweiten Element)

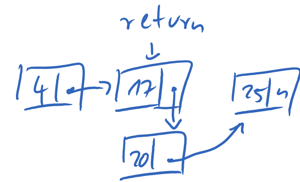
Bsp: Rufe fuegeSortiertEin auf,

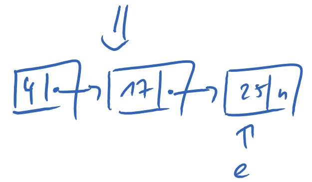
um 20 in die Liste (4, 17, 25) einzufügen.

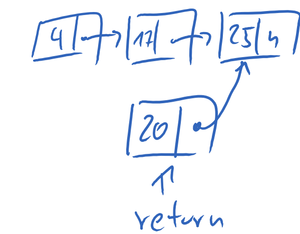










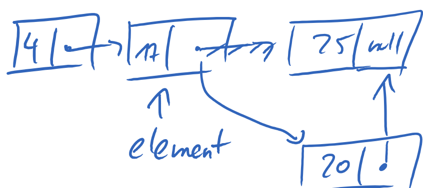


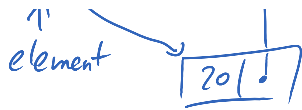
Man kann auch iterative Algorithmen auf rek. Datenstrukturen schreiben.

z.B. iterative Version von

fuegeSortiertEin

- nutzt aus, dass bei der Disjunktion || das 2. Argument nur ausgewertet wird, wenn 1. Argument false ist
- while-Schleife: element durchläuft die Liste, bis es auf das Element zeigt, hinter dem der neue Wert eingefügt werden muss.





wert = 20

l.loesche()

löscht die gesamte Liste

l.loesche(x)

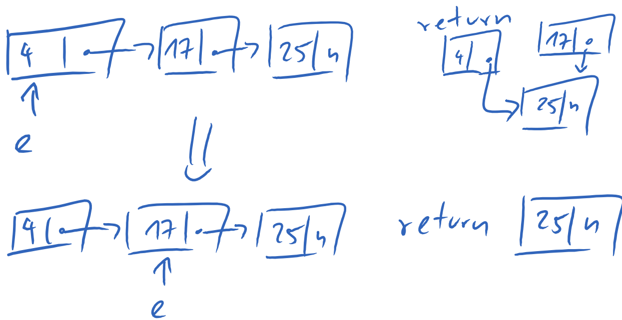
löscht das erste Vorkommen von x aus der Liste l.

benutzt statische Hilfsmethode

loesche(x, e)

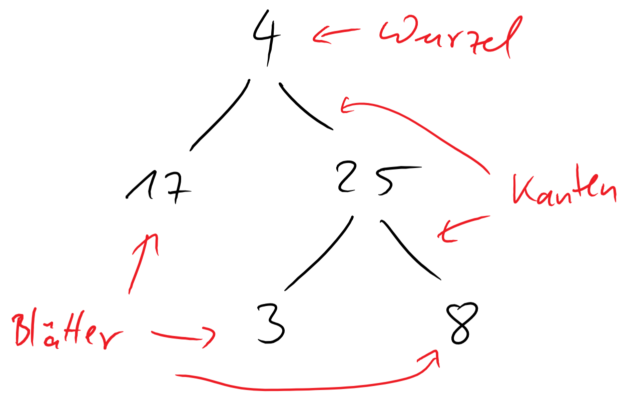
- löscht das 1. Vorkommen von x aus der Liste, die mit e beginnt
- Ergebnis: 1. Element der dadurch entstehenden Liste

Bsp: loesche 17 aus (4, 17, 25):



Mit Referenzen lassen sich nicht nur Listen, sondern viele andere dynam. Datenstrukturen realisieren.

z.B. Binärbäume.



Kanten verbinden Knoten

Knoten: Wurzel, Blätter, innere Knoten