Abschnitt 3.3

Algorithmen auf Listen und Bäumen

Algorithmen auf Listen und Bäumen

Sortieren und Suchen sind elementare Aufgaben, die in den meisten Programmen anfallen.

Verfahren zum Suchen und Sortieren spielen eine zentrale Rolle in der Algorithmik.

Lernziele:

- Intuitiver Algorithmusbegriff
- Kenntnis wichtiger/klassischer Algorithmen und Algorithmenklassen
- Zusammenhang Algorithmus und Datenstruktur
- Wege vom Problem zum Algorithmus
- Implementierungstechniken für Datenstrukturen und Algorithmen (vom Algorithmus zum Programm)

Bemerkung:
Wir führen in den Bereich Algorithmen und Datenstrukturen ausgehend vom Problem ein. Andere Möglichkeit wäre gemäß der benutzten Datenstrukturen (Listen, Bäume, etc.).

Unterabschnitt 3.3.1

Sortieren

Sortieren

Sortieren ist eine Standardaufgabe, die Teil vieler speziellerer, umfassenderer Aufgaben ist.

Untersuchungen zeigen, dass „mehr als ein Viertel der kommerziell verbrauchten Rechenzeit auf Sortiervorgänge entfällt“ (Ottmann, Widmayer: Algorithmen und Datenstrukturen, Kap. 2).

Begriffsklärung: (Sortierproblem)

Gegeben ist eine Folge $s_1, \ldots, s_N$ von sogenannten Datensätzen.

Jeder Satz s_j hat einen Schlüssel k_j. Wir gehen davon aus, dass die Schlüssel ganzzahlig sind.

Aufgabe des **Sortierproblems** ist es, eine Permutation π zu finden, so dass die Umordnung der Sätze gemäß π folgende Reihenfolge auf den Schlüsseln ergibt:

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \cdots \leq k_{\pi(N)}$$

Bemerkung:

Offene Aspekte der Formulierung des Sortierproblem:

- Was heißt, eine Folge ist „gegeben“?
- Ist der Bereich der Schlüssel bekannt?
- Welche Operationen stehen zur Verfügung, um π zu bestimmen?
- Was genau heißt „Umordnung“?

Aufgabenstellung:

Wir benutzen Datensätze folgenden Typs

```
type Dataset = (Int, String)
```

mit Vergleichsfunktion:

```
leq:: Dataset -> Dataset -> Bool
leq (kx, dx) (ky, dy)  =  (kx<=ky)
```

Aufgabenstellung: (2)

Entwickle eine Funktion

```
sort :: [ Dataset ] -> [ Dataset ]
```

so dass das Ergebnis von `sort xl` für alle Eingaben `xl` aufsteigend sortiert ist und die gleichen Elemente enthält wie `xl` (mehrere Einträge mit gleichem Schlüssel sind nicht ausgeschlossen).

Aufgabenstellung: (3)

Wir betrachten:

- Sortieren durch Auswahl (engl. *selection sort*)
- Sortieren durch Einfügen (engl. *insertion sort*)
- Bubblesort
- Sortieren durch rekursives Teilen (*quick sort*)
- Sortieren durch Mischen (*merge sort*)
- Heapsort

Selection Sort

Sortieren durch Auswahl (selection sort)

Algorithmische Idee:

- Entferne einen minimalen Eintrag `min` aus der Liste.
- Sortiere die Liste, aus der `min` entfernt wurde.
- Füge `min` als ersten Element an die sortierte Liste an.

Sortieren durch Auswahl (selection sort) (2)

```
select :: Dataset -> [Dataset] -> Dataset
-- Hilfsfunktion: liefert einen minimalen Eintrag der
-- Liste bzw. x, falls x minimal

select x []     = x
select x (y:yl) = if x `leq` y then select x yl
                               else select y yl
```

Sortieren durch Auswahl (selection sort) (3)

```
delete :: Dataset -> [Dataset] -> [Dataset]
-- Hilfsfunktion: loescht ein Vorkommen von x aus der
-- Liste, falls solches vorhanden

delete x []     = []
delete x (y:yl) = if (x==y) then  yl
                            else  y:(delete x yl)
```

Sortieren durch Auswahl (selection sort) (4)

```
selectionsort :: [Dataset] -> [Dataset]
-- Sortieren durch Auswahl
--    mnm:  ein minimaler Eintrag in Liste xl
--    rest: die Liste xl ohne min

selectionsort []     = []
selectionsort (x:xl) =
      let  mnm  = select x xl
           rest = delete mnm (x:xl)
      in   mnm : (selectionsort rest)
```

Insertion Sort

Sortieren durch Einfügen (insertion sort)

Algorithmische Idee:

- Sortiere zunächst den Rest der Liste.
- Füge dann den ersten Eintrag in die sortierte Liste ein.

Sortieren durch Einfügen (insertion sort) (2)

```
insert :: Dataset -> [Dataset] -> [Dataset]
-- Hilfsfunktion: fuegt Argument in sortierte Liste ein
-- Ergebnis: sortierte Liste

insert x []     =  [x]
insert x (y:yl) =  if (x `leq` y)
                     then  x : (y:yl)
                     else  y : (insert x yl)
```

Sortieren durch Einfügen (insertion sort) (3)

```
insertionsort :: [Dataset] -> [Dataset]
-- Sortieren durch Einfuegen

insertionsort []     = []
insertionsort (x:xl) = insert x (insertionsort xl)
```

Bubble Sort

Bubblesort

Algorithmische Idee:

- Schiebe einen Eintrag nach rechts heraus:
  - Beginne dazu mit dem ersten Eintrag x.
  - Wenn Schieben von x auf einen gleichen oder größeren Eintrag y stößt, schiebe y weiter.
  - Ergebnis: maximaler Eintrag `mxe` und Liste ohne `mxe`
- Sortiere die Liste ohne `mxe` und hänge `mxe` an.

Bubblesort (2)

```
bubble:: [Dataset] -> Dataset -> [Dataset]
                              -> ([Dataset], Dataset)
-- Hilfsfunktion: liefert einen maximalen Eintrag der
-- Liste und die Liste ohne den maximalen Eintrag

bubble rl x []     = (rl,x)
bubble rl x (y:yl) = if (x `leq` y)
                       then bubble (rl++[x]) y yl
                       else bubble (rl++[y]) x yl
```

Bubblesort (3)

```
bubblesort :: [Dataset] -> [Dataset]
-- Sortieren durch Herausschieben der maximalen
--   Elemente

bubblesort []     = []
bubblesort (x:xl) = let (rl,mxe) = bubble [] x xl
                     in (bubblesort rl)++[mxe]
```

Quicksort

Quicksort: Sortieren durch Teilen

Algorithmische Idee:

- Wähle einen beliebigen Datensatz mit Schlüssel k aus, das sogenannte *Pivotelement*.
- Teile die Liste in zwei Teile:
  - 1. Teil enthält alle Datensätze mit Schlüsseln $< k$
  - 2. Teil enthält die Datensätze mit Schlüsseln $\geq k$
- Wende `quicksort` rekursiv auf die Teillisten an.
- Hänge die resultierenden Listen und das Pivotelement zusammen.

Quicksort: Sortieren durch Teilen (2)

```
split :: Dataset -> [Dataset] -> ([Dataset],[Dataset])
-- Hilfsfkt.: teilt Liste in zwei Listen (below,above)
--  below: alle Elemente in kleiner p
--  above: alle Elemente groesser gleich p

split p []     = ([],[])
split p (x:xr) =
    let (blw,abv) = split p xr
    in  if p `leq` x then (blw,x:abv)
                     else (x:blw,abv)
```

Quicksort: Sortieren durch Teilen (3)

```
qsort :: [Dataset] -> [Dataset]
-- Sortieren nach der Strategie ``Teile und Herrsche''

qsort []         = []
qsort (p:rest) = let (below,above) = split p rest
                 in (qsort below) ++ [p] ++ (qsort above)
```

Bemerkung:

Quicksort ist ein typischer Algorithmus gemäß der Divide-and-Conquer-Strategie:

- Zerlege das Problem in Teilprobleme.
- Wende den Algorithmus auf die Teilprobleme an.
- Füge die Ergebnisse zusammen.

Merge Sort

Sortieren durch Mischen:

Algorithmische Idee:

- Hat die Liste mehr als ein Element, berechne die Länge der Liste div 2 (halfsize).
- Teile die Liste in zwei Teile der Länge halfsize (+1).
- Sortiere die Teile.
- Mische die Teile zusammen.

Bemerkung:
Mergesort ist auch effizient für das Sortieren von Datensätzen, die auf externen Speichermedien liegen und nicht vollständig in den Hauptspeicher geladen werden können.

Bemerkung:

```
merge :: [Dataset] -> [Dataset] -> [Dataset]
-- Hilfsfunktion: mischt zwei sortierte Listen
-- zu einer sortierten Liste zusammen

merge [] []           = []
merge [] yl           = yl
merge xl []           = xl
merge (x:xl) (y:yl) = if (x `leq` y)
                       then x : (merge xl (y:yl))
                       else y : (merge (x:xl) yl)
```

Bemerkung: (2)

```
mergesort :: [Dataset] -> [Dataset]
-- Sortieren durch Mischen
--   halfsize: Haelfte der Listenlaenge
--   front:    Vordere Haelfte der Liste
--   back :    Hintere Haelfte der Liste

mergesort []     = []
mergesort (x:[]) = [x]
mergesort xl     =
    let halfsize  = (length xl) `div` 2
        front     = take halfsize xl
        back      = drop halfsize xl
    in
        merge (mergesort front) (mergesort back)
```

Heapsort

LHS Programm

Unterabschnitt 3.3.2

Suchen

Suchen

Die Verwaltung von Datensätzen basiert auf *drei grundlegenden Operationen*:

- *Einfügen* eines Datensatzes in eine Menge von Datensätzen;
- *Suchen* eines Datensatzes mit Schlüssel k;
- *Löschen* eines Datensatzes mit Schlüssel k.

Bemerkung:
Weitere oft gewünschte Operationen sind: Sortierte Ausgabe, Suchen aller Datensätze mit bestimmten Eigenschaften, Bearbeiten von Daten ohne eindeutige Schlüssel, etc.

Schnittstell zum Suchen

Wir betrachten die folgende Schnittstelle:

```
module Dictionary (Dict,emptyDict,get,put,remove) where

type Dict = STree  -- type des Dictionarys

emptyDict :: Dict  -- leeres Dictionary

get :: Dict -> Int -> (Bool,String)
-- Nachschauen des Eintrags zu Schluessel i

put :: Dict -> Int -> String -> Dict
-- Einfuegen des Eintrags (i,s),
-- Ueberschreibt ggf. alten Eintrag zu i

remove :: Dict -> Int -> Dict
-- Loeschen des Eintrags zu Schluessel i
```

Bemerkung:

In der Literatur zur funktionalen Programmierung wir „get“ oft „lookup“ oder „search“, „put“ oft „insert“ und „remove“ oft „delete“genannt.

Um den Zusammenhang zu OO-Schnittstellen augenfälliger zu machen, benutzen wir die dort üblichen Namen.

Ziel ist es, Datenstrukturen zu finden, bei denen der Aufwand für obige Operationen gering ist. Wir betrachten hier die folgenden Dictionary-Realisierungen:

- lineare Datenstrukturen (Übung)
- (natürliche) binäre Suchbäume (Vorlesung)

Begriffsklärung: (Binärer Suchbaum)

Ein markierter Binärbaum B ist ein ***natürlicher binärer Suchbaum*** (kurz: binärer Suchbaum), wenn die Suchbaum-Eigenschaft gilt, d.h. wenn für jeden Knoten K in B gilt:

- Alle Schlüssel im linken Unterbaum von K sind echt kleiner als der Schlüssel von K.
- Alle Schlüssel im rechten Unterbaum von K sind echt größer als der Schlüssel von K.

Bemerkung:

- „Natürlich“ bezieht sich auf das Entstehen der Bäume in Abhängigkeit von der Reihenfolge der Einfüge-Operationen (Abgrenzung zu balancierten Bäumen).
- In einem binären Suchbaum gibt es zu einem Schlüssel maximal einen Knoten mit entsprechender Markierung.

Datenstruktur für Suchbäume:

Wir stellen Dictionaries als Binärbäume mit Markierungen vom Typ `Dataset` dar:

```
data STree =  Node Dataset STree STree
            |  Empty
     deriving (Eq, Show)

emptyDict = Empty
```

Die Konstante `emptyDict` repräsentiert das leere Dictionary.

Invariante für Suchbäume:

Binärbäume, die als Dictionary verwendet werden, müssen die Suchbaum-Eigenschaft erfüllen.

Alle Funktionen, die Dictionaries als Parameter bekommen, gehen davon aus, dass die Suchbaum-Eigenschaft für die Parameter gilt.

Die Funktionen müssen garantieren, dass die Eigenschaft auch für Ergebnisse gilt.

Man sagt:
Die Suchbaum-Eigenschaft ist eine ***Datenstrukturinvariante*** von Dictionaries.

Wir garantieren die Datenstrukturinvariante u.a. dadurch, dass wir Nutzern des Moduls `Dictionary` keinen Zugriff auf die Konstruktoren geben.

Suchen eines Eintrags:

Wenn kein Eintrag zum Schlüssel existiert, liefere `(False,"")`; sonst liefere `(True,s)`, wobei s der String zum Schlüssel ist:

```
get Empty k          = (False,"")
get (Node (km,s) l r) k
  | k < km     =   get l k
  | km < k     =   get r k
  | otherwise  =   (True,s)
```

Einfügen eines Eintrags:

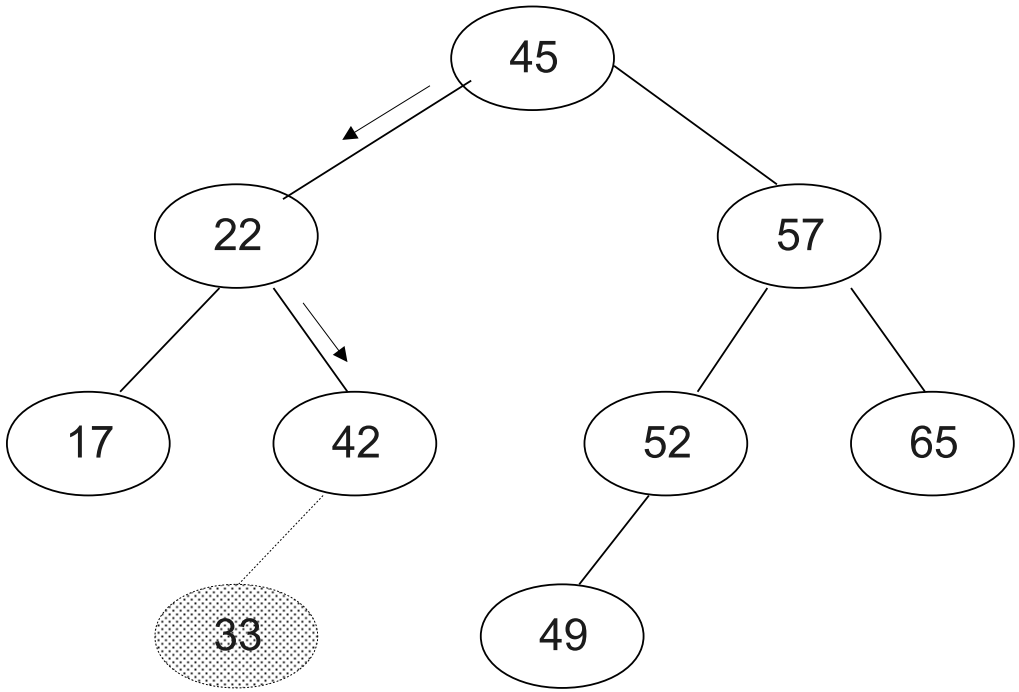
Algorithmisches Vorgehen:

- Neue Knoten werden immer als Blätter eingefügt.
- Die Position des Blattes wird durch den Schlüssel des neuen Eintrags festgelegt.
- Beim Aufbau eines Baumes ergibt der erste Eintrag die Wurzel.
- Ein Knoten wird
  - in den linken Unterbaum der Wurzel eingefügt, wenn sein Schlüssel kleiner ist als der Schlüssel der Wurzel;
  - in den rechten, wenn er größer ist.

  Dieses Verfahren wird rekursiv fortgesetzt, bis die Einfügeposition bestimmt ist.

Beispiel:

Einfügen von 33:



Implementierung von put:

Die algorithmische Idee lässt sich direkt umsetzen.

Beachte aber, dass das Dictionary nicht verändert wird, sondern ein neues erzeugt und abgeliefert wird:

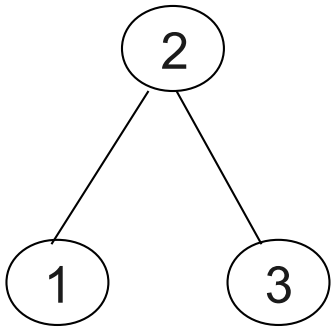
```
put Empty k s      =  Node (k,s) Empty Empty
put (Node (km,sm) l r) k s
  |  k == km    =  Node (k,s) l r
  |  k < km     =  Node (km,sm) (put l k s) r
  |  otherwise  =  Node (km,sm) l (put r k s)
```

Bemerkungen:

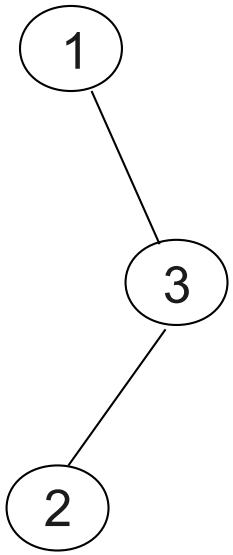
- Die Reihenfolge des Einfügens bestimmt das Aussehen des binären Suchbaums:

  Reihenfolgen:

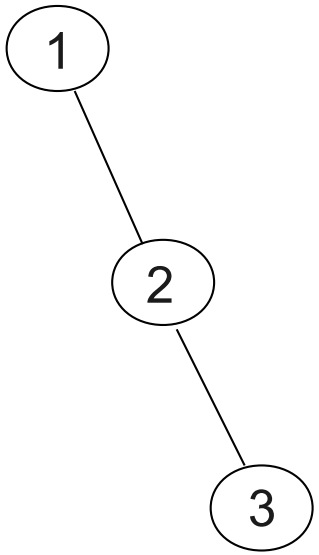
2 ; 3 ; 1



1 ; 3 ; 2



1 ; 2 ; 3



Bemerkungen: (2)

- Es gibt sehr viele Möglichkeiten, aus einer vorgegebenen Schlüsselmenge einen binären Suchbaum zu erzeugen.
- Bei sortierter Einfügereihenfolge entartet der binäre Suchbaum zur linearen Liste.
- Der Algorithmus zum Einfügen ist schnell, insbesondere weil keine Ausgleichs- oder Reorganisationsoperationen vorgenommen werden müssen.

Löschen:

Löschen ist die schwierigste Operation, da

- ggf. innere Knoten entfernt werden und dabei
- die Suchbaum-Eigenschaft erhalten werden muss.

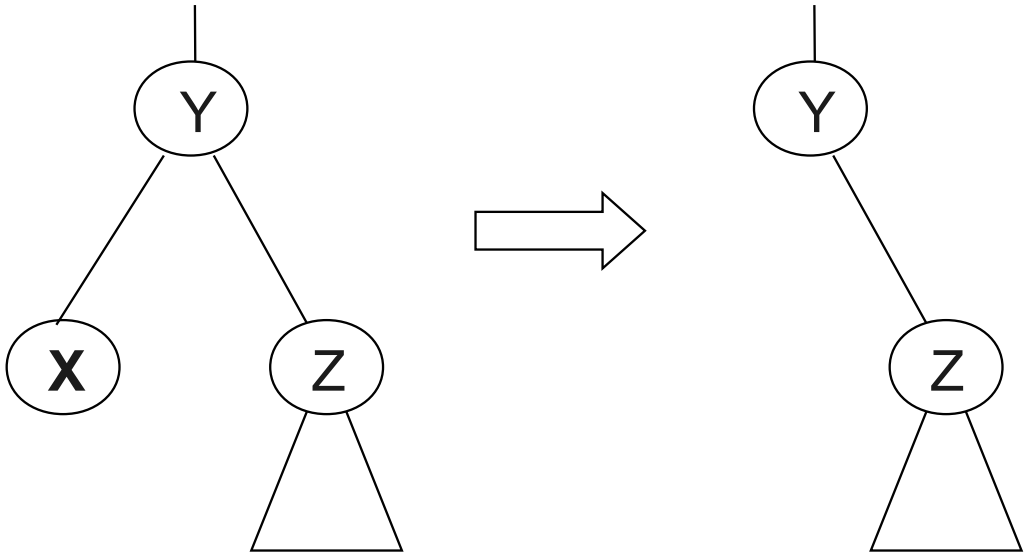
Algorithmisches Vorgehen:

- Die Position eines zu löschenden Knotens K mit Schlüssel X wird nach dem gleichen Verfahren wie beim Suchen eines Knotens bestimmt.
- Dann sind drei Fälle zu unterscheiden:

Löschen: (2)

1. Fall: *K* ist ein Blatt.

Lösche *K*:

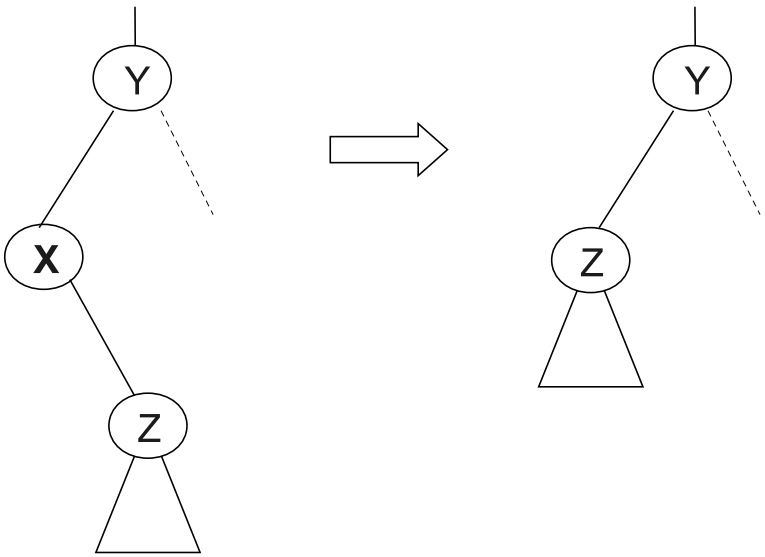


Entsprechend, wenn Knoten mit Schlüssel *X* in rechtem Unterbaum.

Löschen: (3)

2. Fall: *K* mit Schlüssel *X* hat genau einen Unterbaum.

K wird im Eltern-Knoten durch sein Kind ersetzt und gelöscht:



Die anderen links-rechts-Varianten entsprechend.

Löschen: (4)

3. Fall: *K* mit Schlüssel *X* hat genau zwei Unterbäume.

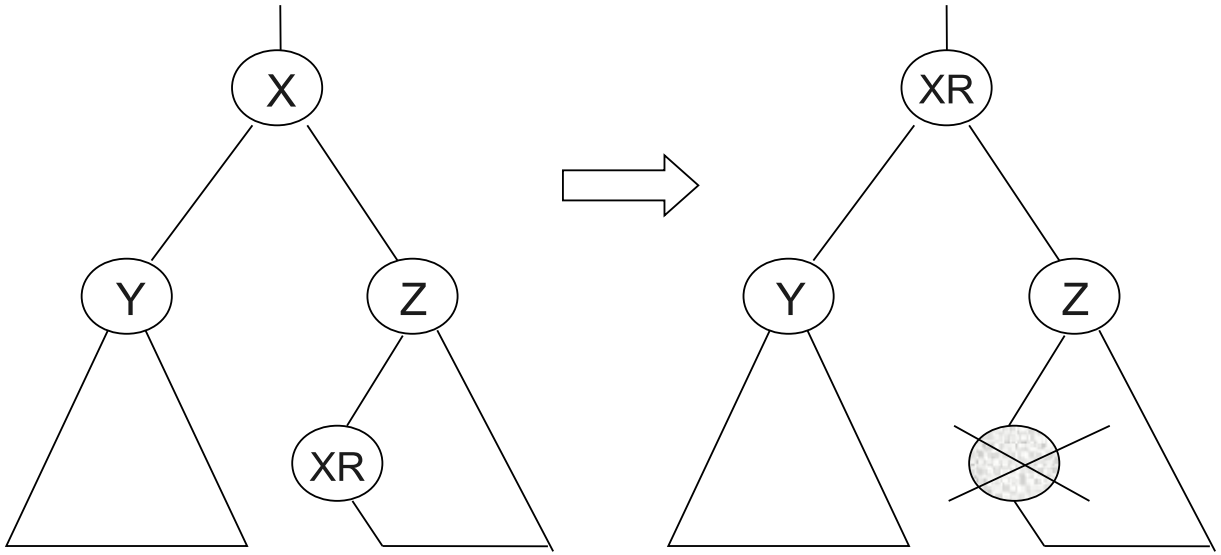
Problem:
Wo werden die beiden Unterbäume nach dem Löschen von *K* eingehängt?

Hier gibt es 2 symmetrische Lösungsvarianten:

- Ermittle den Knoten *KR* mit dem kleinsten Schlüssel im rechten Unterbaum, Schlüssel von *KR* sei *XR*.
- Speichere *XR* und die Daten von *KR* in *K*.
- Lösche *KR* gemäß Vorgehen zu Fall 1 bzw. 2, möglich da für *KR* einer der Fälle zutrifft.

(andere Variante: größten Schlüssel im linken UB)

Löschen: (5)



Umsetzung in Haskell

- Die Fälle 1 und 2 lassen sich direkt behandeln.
- Für Fall 3 realisiere Hilfsfunktion `removemin`, die
  - nichtleeren binären Suchbaum *b* als Parameter nimmt;
  - ein Paar (*mnm*, *br*) als Ergebnis liefert, wobei
    - *mnm* der kleinste Datensatz in *b* ist und
    - *br* der Baum ist, der sich durch Löschen von *mnm* aus *b* ergibt.

Umsetzung in Haskell (2)

```
removemin :: STree -> (Dataset,STree)
-- Parameter: nichtleerer binaerer Suchbaum b.
-- Liefert Eintrag d mit kleinstem Schluessel in b
-- und Baum nach Loeschen von d in b

removemin (Node d Empty r) = (d,r)
removemin (Node d l r) =
    let (mnm,ll) = removemin l
    in  (mnm, Node d ll r)
```

Umsetzung in Haskell (3)

```
remove Empty k                  = Empty
remove (Node (km,s) l r) k
   |  k < km      =  Node (km,s) (remove l k) r
   |  km < k      =  Node (km,s) l (remove r k)
   |  l == Empty  =  r
   |  r == Empty  =  l
   |  otherwise   =     -- k == km && l /= Empty /= r
         let (mnm,rr) = removemin r
         in  Node mnm l rr
```

Diskussion:

Der Aufwand für die Grundoperationen Einfügen, Suchen und Löschen eines Knotens ist proportional zur Tiefe des Knotens, bei dem die Operation aus- geführt wird.

Ist h die Höhe des Suchbaumes, ist der Aufwand der Grundoperationen im ungünstigsten Fall also $O(h)$, wobei

$$log(N+1) \quad \leq \quad h \quad \leq \quad N$$

für Knotenanzahl N.

Folgerung:
Bei degenerierten natürlichen Suchbäumen kann linearer Aufwand für alle Grundoperationen entstehen.

Im Mittel verhalten sich Suchbäume aber wesentlich besser. Zusätzlich versucht man durch gezielte Reorganisation eine gute Balancierung zu erreichen (siehe Kapitel 5).