Karlsruher Institut für Technologie

Algorithmen 1 – Sommersemester 2018
Zuverlässige Softwaresysteme im Kontext der Automobilindustrie
`http://verialg.iti.kit.edu`

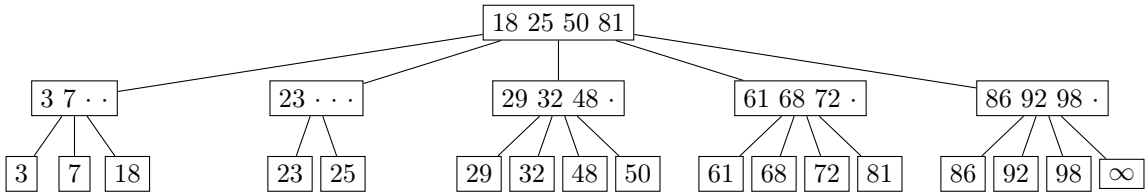
Prof. Dr. Carsten Sinz – Markus Iser

Übungsblatt 9

Ausgabe: 20.06.2018 – 15:30
Abgabe: 27.06.2018 – 13:00

A (a, b)-Bäume

Führen Sie auf dem folgenden $(2, 5)$-Baum die geforderten Operationen in der angegebenen Reihenfolge durch. Zeichnen Sie den Endzustand des Baums nach jeder der angegebenen Operationen.

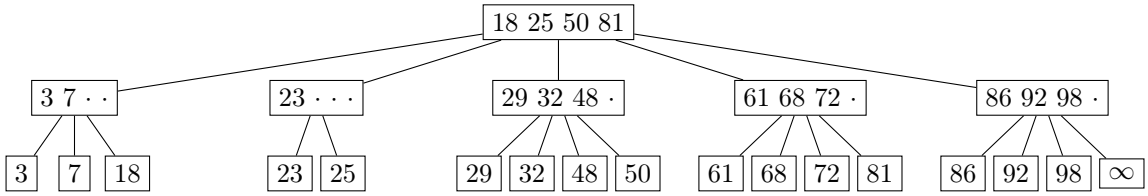


A.1 Einfügen (1,5 Punkte)

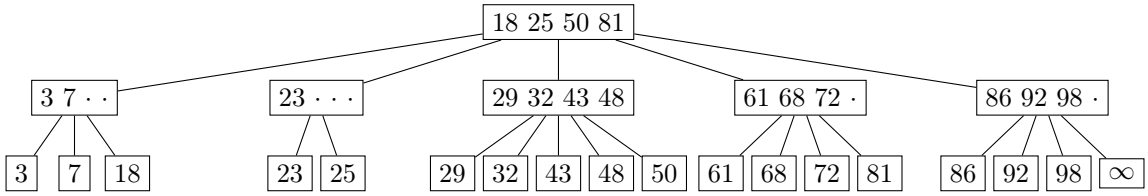
Einfügen von 43, Einfügen von 45, Einfügen von 38.

Lösung

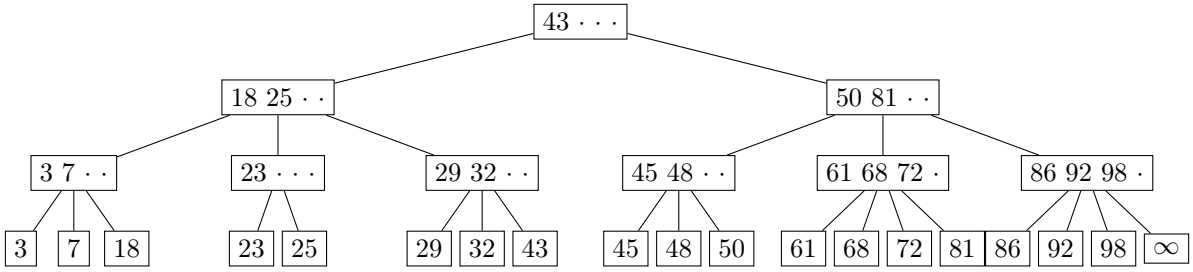
Originalzustand



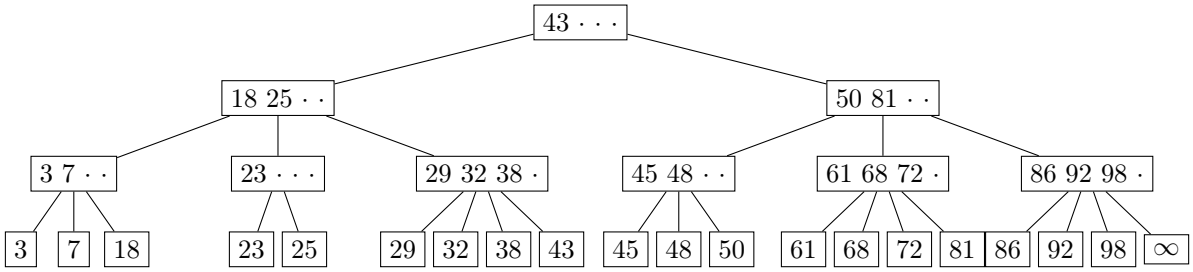
nach Einfügen von 43



nach Einfügen von 45



nach Einfügen von 38

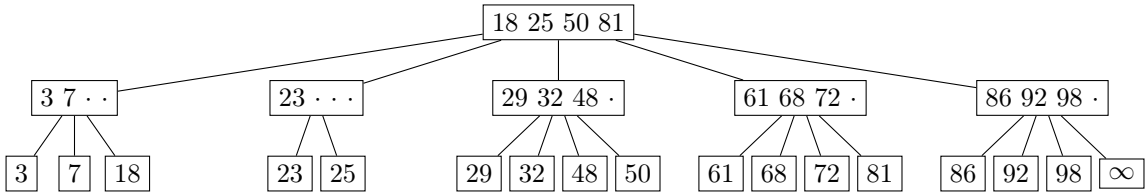


A.2 Löschen (1,5 Punkte)

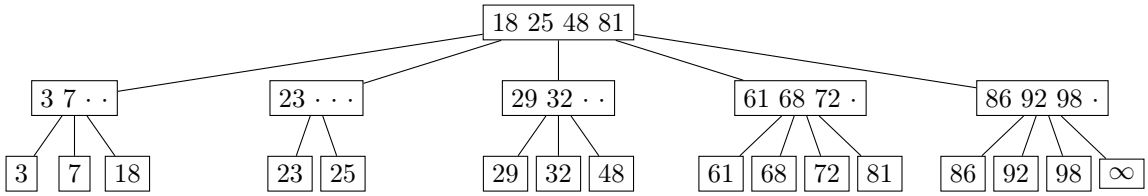
Beginnen Sie wieder *beim Ausgangszustand*. Löschen von 50, Löschen von 23, Löschen von 48.

Lösung

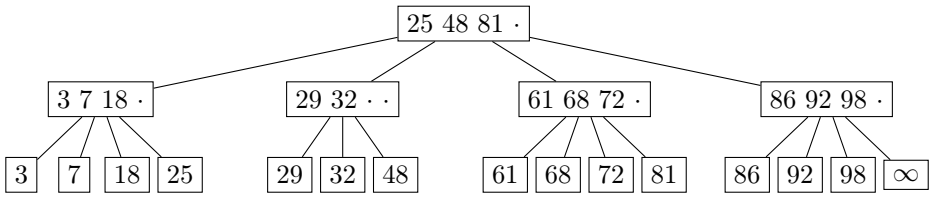
Originalzustand



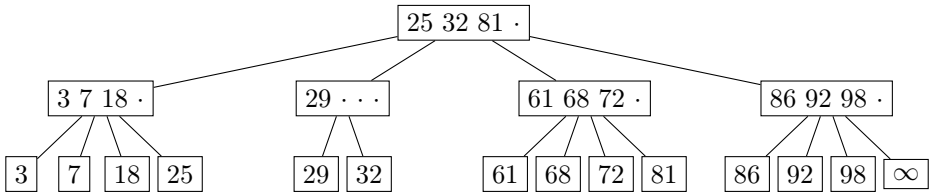
nach Löschen von 50



nach Löschen von 23



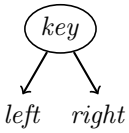
nach Löschen von 48



B Binäre Suchbäume

In der Vorlesung wurden Binärbäume mit einer zusätzlichen verketteten Liste an den Blättern vorgestellt. Im Folgenden betrachten wir ausschließlich binäre Suchbäume *ohne zusätzliche Datenliste*. Die Klasse BinaryNode für die Knoten ist vorgegeben, wobei left und right auf nil gesetzt werden können.

```
class BinaryNode(key : Element, left, right : BinaryNode) {
  Element key
  BinaryNode left //Alle über left erreichbare Elemente sind <= key
  BinaryNode right //Alle über right erreichbare Elemente sind > key
}
```



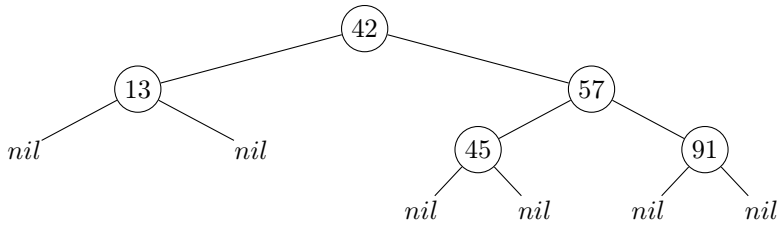
Wir betrachten in dieser Aufgabe die Liste $A = (42, 13, 57, 91, 45, 48, 6, 51, 10, 55, 13)$.

B.1 Naives Einfügen (2 Punkte)

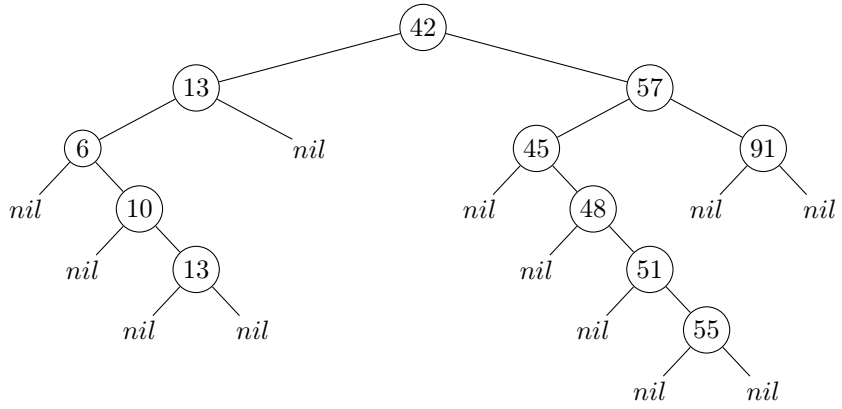
Fügen Sie schrittweise die Elementen aus Liste A in einen anfangs leeren binären Suchbaum mit der naiven Methode ein. Zeichen den binären Suchbaum nach Einfügen von 45 und 13. Welche Höhe hat der Baum nach Einfügen aller Elemente?

Lösung

Der Baum nach Einfügen von 45:



Der Baum nach Einfügen von 13:



Der Knoten mit 55 hat Höhe 5, der Baum hat also Höhe 5.

B.2 Von Sortiert zu Balanciert (2 Punkte)

Implementieren Sie einen Algorithmus, der aus einer beliebigen duplikatfreien sortierten Folge L der Länge n einen binären Suchbaum *minimaler Höhe* konstruiert. Begründen Sie kurz warum die von Ihrem Algorithmus konstruierten Suchbaum diese Eigenschaft haben.

Lösung

Wir bauen einen binären Suchbaum rekursiv auf, in dem wir aus einem vorsortierten Teilarray immer das mittlere Element als Wurzel wählen und aus den restlichen Elemente rekursiv einen Suchbaum konstruieren. Diese Idee wird in folgender Funktion umgesetzt:

Algorithm 1: Construct(L : Array $[0..n-1]$ Of Element) : BinaryNode

```
1 if n ≤ 0 then
2     return nil
3 s := ⌊n/2⌋
4 left := Construct(L[0..s − 1])
5 right := Construct(L[s + 1..n − 1])
6 return BinNode(L[s], left, right)
```

Der so konstruierte Baum hat Höhe $\lfloor \log_2 n \rfloor$, weil das Array nur ebenso oft halbiert werden kann (wie man es von der binären Suche her kennt). Jeder binäre Baum mit n Knoten hat aber mindestens Höhe $\lfloor \log_2 n \rfloor$ (siehe perfekte Binärbäume in der 6. Übung), daher ist das Ergebnis minimal.

B.3 Ausführung (1 Punkt)

Betrachten Sie anschließend die duplikatfreie, sortierte Liste $L = (6, 10, 13, 42, 45, 48, 51, 55, 57, 91)$ von Elementen aus A. Wenden Sie Ihren Algorithmus aus Teilaufgabe **??** auf L an, und bestimmen Sie die Höhe des erzielten Baums.

Lösung

Wir wenden unser Construct() auf $[\,6, 10, 13, 42, 45, 48, 51, 55, 57, 91\,]$ an. Die Indizes wie 0101 symbolisieren den Zweig im Rekursionsbaum.

Construct($[\,6, 10, 13, 42, 45, 48, 51, 55, 57, 91\,]$) mit $n = 10$, $s = 5$, $L[s] = 48$ berechnet rekursiv:

Construct$_0$($[\,6, 10, 13, 42, 45\,]$) mit $n_0 = 5$, $s_0 = 2$, $L[s] = 13$ berechnet rekursiv:

$\mathsf{Construct}_{00}([\,6, 10\,])$ mit $n_{00} = 2$, $s_{00} = 1$, $L[s] = 10$ berechnet rekursiv:

$\mathsf{Construct}_{000}([\,6\,])$ mit $n_{000} = 1$, $s_{000} = 0$, $L[s] = 6$ berechnet rekursiv:

$\mathsf{Construct}_{0000}([\,])$ liefert nil

$\mathsf{Construct}_{0001}([\,])$ liefert nil

$\mathsf{Construct}_{000}([\,6\,])$ liefert $\mathsf{BinNode}(6, nil, nil)$.

$\mathsf{Construct}_{001}([\,])$ liefert nil

$\mathsf{Construct}_{00}([\,6, 10\,])$ liefert $\mathsf{BinNode}(10, \mathsf{BinNode}(6, nil, nil), nil)$.

$\mathsf{Construct}_{01}([\,42, 45\,])$ mit $n_{01} = 2$, $s_{01} = 1$, $L[s] = 45$ berechnet rekursiv:

$\mathsf{Construct}_{010}([\,42\,])$ mit $n_{010} = 1$, $s_{010} = 0$, $L[s] = 42$ berechnet rekursiv:

$\mathsf{Construct}_{0100}([\,])$ liefert nil

$\mathsf{Construct}_{0101}([\,])$ liefert nil

$\mathsf{Construct}_{010}([\,42\,])$ liefert $\mathsf{BinNode}(42, nil, nil)$.

$\mathsf{Construct}_{011}([\,])$ liefert nil

$\mathsf{Construct}_{00}([\,42, 45\,])$ liefert $\mathsf{BinNode}(45, \mathsf{BinNode}(42, nil, nil), nil)$.

$\mathsf{Construct}_{0}([\,6, 10, 13, 42, 45\,])$ liefert $\mathsf{BinNode}(13, \mathsf{BinNode}(10, \mathsf{BinNode}(6, nil, nil), nil), \mathsf{BinNode}(45, \mathsf{BinNode}(42, nil, nil), nil))$.

$\mathsf{Construct}_{1}([\,51, 55, 57, 91\,])$ mit $n_{1} = 4$, $s_{1} = 2$, $L[s] = 57$ berechnet rekursiv:

$\mathsf{Construct}_{10}([\,51, 55\,])$ mit $n_{10} = 2$, $s_{10} = 1$, $L[s] = 55$ berechnet rekursiv:

$\mathsf{Construct}_{100}([\,51\,])$ mit $n_{100} = 1$, $s_{100} = 0$, $L[s] = 51$ berechnet rekursiv:

$\mathsf{Construct}_{1000}([\,])$ liefert nil

$\mathsf{Construct}_{1001}([\,])$ liefert nil

$\mathsf{Construct}_{100}([\,51\,])$ liefert $\mathsf{BinNode}(51, nil, nil)$.

$\mathsf{Construct}_{101}([\,])$ liefert nil

$\mathsf{Construct}_{10}([\,51, 55\,])$ liefert $\mathsf{BinNode}(55, \mathsf{BinNode}(51, nil, nil), nil)$.

$\mathsf{Construct}_{110}([\,91\,])$ mit $n_{110} = 1$, $s_{110} = 0$, $L[s] = 91$ berechnet rekursiv:

$\mathsf{Construct}_{1100}([\,])$ liefert nil

$\mathsf{Construct}_{1101}([\,])$ liefert nil

$\mathsf{Construct}_{110}([\,91\,])$ liefert $\mathsf{BinNode}(91, nil, nil)$.

$\mathsf{Construct}_{111}([\,])$ liefert nil

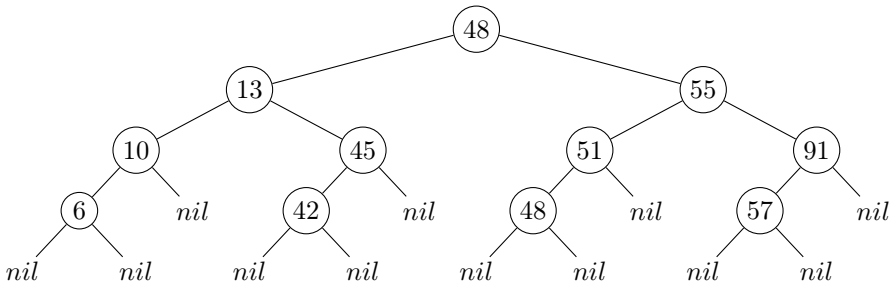
$\mathsf{Construct}_{10}([\,57, 91\,])$ liefert $\mathsf{BinNode}(91, \mathsf{BinNode}(57, nil, nil), nil)$.

$\mathsf{Construct}_{1}([\,51, 55, 57, 91\,])$ liefert $\mathsf{BinNode}(55, \mathsf{BinNode}(51, \mathsf{BinNode}(48, nil, nil), nil), \mathsf{BinNode}(91, \mathsf{BinNode}(57, nil, nil), nil))$.

Construct($[6, 10, 13, 42, 45, 48, 51, 55, 57, 91]$) liefert also

$$\begin{aligned}&\text{BinNode}(48, \text{BinNode}(13,\\ &\qquad \text{BinNode}(10, \text{BinNode}(6, nil, nil), nil),\\ &\qquad \text{BinNode}(45, \text{BinNode}(42, nil, nil), nil)),\\ &\quad \text{BinNode}(55,\\ &\qquad \text{BinNode}(51, \text{BinNode}(48, nil, nil), nil),\\ &\qquad \text{BinNode}(91, \text{BinNode}(57, nil, nil), nil)))\end{aligned}$$

Der in obiger Konstruktion-Vorschrift schwer lesbare Baum sieht folgendermaßen aus und hat Höhe 3:



B.4 Find Spezial (2 Punkte)

Implementieren Sie eine Funktion find(), die in einem binären Suchbaum für einen beliebigen Schlüssel k den Knoten x mit dem kleinsten Element $x.key \geq k$ zurückliefert und dabei höchstens h Kanten/Pointer left/right traversiert, wobei h die Höhe des Baums ist.

Lösung

Algorithm 2: find(k : Element, node : BinaryNode) : BinaryNode

```
1  if node = nil then
2      return nil
3  else
4      if node.key = k then
5          return node
6      else if node.key > k then
7          leftResult := find(k, node.left)
8          if leftResult = nil then
9              return node
10         else
11             return leftResult
12     else
13         return find(k, node.right)
```

Die Funktion find liefert in einem binären Suchbaum für einen beliebigen Schlüssel k den Knoten x mit dem kleinsten Element $x.key \geq k$ zurück. Die Suche endet in jedem Fall in einem inneren Knoten x auf Höhe h. Dies

ist der gesuchte Knoten, falls $k \leq x.key$ gilt. Andernfalls ist der gesuchte Knoten genau an der Stelle auf dem Pfad von der Wurzel zu x, an der zum *letzten* Mal dem $left$-Pointer gefolgt wurde, da dieser Knoten der letzte Knoten x mit $x.key \geq k$ war, der traversiert wurde. Aus diesem Grund merken wir uns in tmp beim Abstieg immer genau den letzten Knoten, an dem dem $left$-Pointer gefolgt wurde.