Imperative Programmierung (IPR)

Kapitel 6: Tabellen

Univ.-Prof. Dr.-Ing. habil. Gero Mühl

Lehrstuhl für Architektur von Anwendungssystemen (AVA)
Fakultät für Informatik und Elektrotechnik (IEF)
Universität Rostock

Universität Rostock

Traditio et Innovatio

AVA

Inhalte

Kapitel 6.1

Definition

Die Tabelle

Definition 1 (Tabelle)

*Die **Tabelle** ist ein grundlegender Datentyp, der die Speicherung einer Menge von Schlüssel/Wert-Paaren (Einträgen) und den Zugriff auf die gespeicherten Werte mittels der eindeutigen Schlüssel ermöglicht.*

Beispiel 1

$insert(1, Elefant)$
$insert(3, Tiger)$
$insert(7, Giraffe)$
$insert(11, Schimpanse)$

#	Tier
1	Elefant
3	Tiger
7	Giraffe
11	Schimpanse

- $read(1) = Elefant$
- $isin(4) = False$

Typische Anwendungsbeispiele für Tabellen

- Speicherung von Personaldaten
  unter dem Schlüssel der Personalnummer
- Speicherung von Bestandsinformationen
  unter dem Schlüssel der Inventarobjektnummer
- Speicherung von Platzkarteninformationen
  unter dem Schlüssel der Kombination von Zugnummer und Datum
- Verwaltung von Variablen und ihrer Werte in einem Interpreter
- Symboltabelle eines Compilers

Kapitel 6.2

Spezifikation

Grundlegende Funktionen einer Tabelle

Funktion	Beschreibung der Funktion
`init`	Neue Tabelle
`insert(k,v,t)`	Fügt Wert v mit Schlüssel k ein
`read(k,t)`	Liefert Wert zu Schlüssel
`delete(k,t)`	Löscht Wert zu Schlüssel
`update(k,v,t)`	Aktualisiert Wert zu Schlüssel
`isin(k,t)`	Prüft, ob Schlüssel vorhanden
`empty(t)`	Prüft, ob Tabelle leer
`full(t)`	Prüft, ob Tabelle voll
`length(t)`	Liefert Länge der Tabelle

Schrittweise Entwicklung der Spezifikation

- Die Beschreibung des Datentyps `Table` greift zurück auf:

$$[\mathbb{B}, \mathbb{N}, \mathit{Key}, \mathit{Value}]$$

- Innerhalb von *Value* soll es zusätzlich einen Fehlerwert geben (z. B. für den Fall, dass mittels `read` auf die leere Tabelle zugegriffen wird):

$$\mid \quad \mathit{errorvalue} \in \mathit{Value}$$

- Zur einfacheren Spezifikation der Eigenschaften definieren wir noch:

$$\begin{array}{l} \mid \quad \mathit{Value}_V = \mathit{Value} \setminus \{\mathit{errorvalue}\} \\ \mid \quad \mathit{Table}_X = \{t \in \mathit{Table} \mid \mathit{full}(t)\} \end{array}$$

Schrittweise Entwicklung der Spezifikation

- Wir definieren die Menge von Tabellen:

  $$[Table]$$

- Tabellen können mit folgenden Funktionen erzeugt werden:

  $$\begin{array}{l} init : Table \\ insert : Key \times Value \times Table \longrightarrow Table \end{array}$$

- Schließlich gibt es eine Konstante für die maximale Anzahl von Einträgen (Schlüssel/Wert-Paaren) in einer Tabelle:

  $$maxentries \geq 1$$

- Jetzt können die Eigenschaften der weiteren Funktionen der Tabelle definiert werden.

Tabellenoperationen: `insert`

$$
\begin{array}{|l}
\hline
\mathit{insert} : \mathit{Key} \times \mathit{Value} \times \mathit{Table} \longrightarrow \mathit{Table} \\
\hline
\forall\, k : \mathit{Key};\ v : \mathit{Value}_V;\ t : \mathit{Table} \bullet \\
\qquad \mathit{insert}(k, \mathit{errorvalue}, t) = t \\
\qquad \mathit{isin}(k, t) = \mathit{True} \\
\qquad\qquad \Rightarrow \mathit{insert}(k, v, t) = t \\
\forall\, k : \mathit{Key};\ v : \mathit{Value}_V;\ t : \mathit{Table}_X \bullet \\
\qquad \mathit{insert}(k, v, t) = t \\
\end{array}
$$

- Hier wird also nur eine einfache Fehlerbehandlung realisiert, bei der das Einfügen in eine volle Tabelle diese unverändert lässt.
- Das erneute Einfügen eines Schlüssels lässt die Tabelle unverändert.
- Alternativ ließe sich auch eine erweiterte Fehlerbehandlung mit *overflow* realisieren.

Tabellenoperationen: `read`

$$read : Key \times Table \longrightarrow Value$$

$$\forall\, x, y : Key;\ v : Value_V;\ t : Table \setminus Table_X \mid x \neq y \bullet$$

$$read(x, init) = errorvalue$$
$$read(x, insert(x, v, t)) = v$$
$$read(x, insert(y, v, t)) = read(x, t)$$

Beispiel 2 (read)

$$read(\underbrace{3}_{x}, insert(\underbrace{1}_{y}, \underbrace{Elefant}_{v}, \underbrace{insert(3, Giraffe, insert(5, Kamel, init))}_{t}))$$
$$= read(\underbrace{3}_{x}, insert(\underbrace{3}_{x}, \underbrace{Giraffe}_{v}, \underbrace{insert(5, Kamel, init)}_{t}))$$
$$= Giraffe$$

Tabellenoperationen: `delete`

$$
\begin{array}{|l}
\hline
delete : Key \times Table \longrightarrow Table \\
\hline
\forall\, x, y : Key;\ v : Value_V;\ t : Table \setminus Table_X \mid x \neq y \bullet \\
\quad delete(x, init) = init \\
\quad delete(x, insert(x, v, t)) = t \\
\quad delete(x, insert(y, v, t)) = insert(y, v, delete(x, t))
\end{array}
$$

Beispiel 3 (delete)

$$
\begin{aligned}
& delete(\underbrace{3}_{x}, insert(\underbrace{1}_{y}, \underbrace{Elefant}_{v}, \underbrace{insert(3, Giraffe, insert(5, Kamel, init))}_{t})) \\
&= insert(1, Elefant, delete(\underbrace{3}_{x}, insert(\underbrace{3}_{x}, \underbrace{Giraffe}_{v}, \underbrace{insert(5, Kamel, init)}_{t}))) \\
&= insert(1, Elefant, insert(5, Kamel, init))
\end{aligned}
$$

Tabellenoperationen: update

$$update : Key \times Value \times Table \longrightarrow Table$$

$$\forall x, y : Key;\ v, w : Value_V;\ t : Table \setminus Table_X \mid x \neq y \bullet$$

$$update(x, errorvalue, t) = t$$
$$update(x, v, init) = init$$
$$update(x, v, insert(x, w, t)) = insert(x, v, t)$$
$$update(x, v, insert(y, w, t)) = insert(y, w, update(x, v, t))$$

Beispiel 4 (update)

$$update(\underbrace{3}_{x}, \underbrace{Kuh}_{v}, insert(\underbrace{1}_{y}, \underbrace{Elefant}_{w}, \underbrace{insert(3, Giraffe, insert(5, Kamel, init))}_{t}))$$

$$= insert(1, Elefant, update(\underbrace{3}_{x}, \underbrace{Kuh}_{v}, insert(\underbrace{3}_{x}, \underbrace{Giraffe}_{w}, \underbrace{insert(5, Kamel, init)}_{t})))$$

$$= insert(1, Elefant, insert(3, Kuh, insert(5, Kamel, init)))$$

Tabellenoperationen: `isin`

$$
\begin{array}{|l}
\hline
isin : Key \times Table \longrightarrow \mathbb{B} \\
\hline
\forall\, x, y : Key;\ v : Value_V;\ t : Table \setminus Table_X \mid x \neq y \bullet \\
\quad isin(x, init) = False \\
\quad isin(x, insert(x, v, t)) = True \\
\quad isin(x, insert(y, v, t)) = isin(x, t) \\
\hline
\end{array}
$$

Beispiel 5 (isin)

$$
\begin{aligned}
& isin(\underbrace{4}_{x}, insert(\underbrace{1}_{y}, \underbrace{Elefant}_{v}, \underbrace{insert(3, Giraffe, insert(5, Kamel, init))}_{t})) \\
&= isin(\underbrace{4}_{x}, insert(\underbrace{3}_{y}, \underbrace{Giraffe}_{v}, \underbrace{insert(5, Kamel, init)}_{t})) \\
&= isin(\underbrace{4}_{x}, insert(\underbrace{5}_{y}, \underbrace{Kamel}_{v}, \underbrace{init}_{t})) \\
&= isin(\underbrace{4}_{x}, init) = False
\end{aligned}
$$

Tabellenoperationen: `empty`

$$
\begin{array}{|l}
\hline
empty : Table \longrightarrow \mathbb{B} \\
\hline
\forall\, k : Key;\ v : Value_V;\ t : Table \setminus Table_X \bullet \\
\quad empty(init) = True \\
\quad empty(insert(k, v, t)) = False
\end{array}
$$

Beispiel 6 (empty)

$$
empty(insert(\underbrace{1}_{k}, \underbrace{Elefant}_{v}, \underbrace{insert(3, Giraffe, insert(5, Kamel, init))}_{t}))
$$

$$
= False
$$

Tabellenoperationen: `length`

$$
\begin{array}{|l}
\hline
\mathit{length} : \mathit{Table} \longrightarrow \mathbb{N} \\
\hline
\forall\, k : \mathit{Key};\ v : \mathit{Value}_V;\ t : \mathit{Table} \setminus \mathit{Table}_X \bullet \\
\quad \mathit{length}(\mathit{init}) = 0 \\
\quad \mathit{length}(\mathit{insert}(k, v, t)) = 1 + \mathit{length}(t) \\
\hline
\end{array}
$$

Beispiel 7 (length)

$$\mathit{length}(\mathit{insert}(\underbrace{1}_{k}, \underbrace{\mathit{Elefant}}_{v}, \underbrace{\mathit{insert}(3, \mathit{Giraffe}, \mathit{insert}(5, \mathit{Kamel}, \mathit{init}))}_{t}))$$

$$= 1 + \mathit{length}(\mathit{insert}(\underbrace{3}_{k}, \underbrace{\mathit{Giraffe}}_{v}, \underbrace{\mathit{insert}(5, \mathit{Kamel}, \mathit{init})}_{t}))$$

$$= 1 + 1 + \mathit{length}(\mathit{insert}(\underbrace{5}_{k}, \underbrace{\mathit{Kamel}}_{v}, \underbrace{\mathit{init}}_{t}))$$

$$= 1 + 1 + 1 + \mathit{length}(\mathit{init}) = 1 + 1 + 1 + 0 = 3$$

Tabellenoperationen: `full`

$$
\begin{array}{|l}
full : Table \longrightarrow \mathbb{B} \\
\hline
\forall\, t : Table_X \bullet \\
\qquad full(t) = True \\
\forall\, t : Table \setminus Table_X \bullet \\
\qquad full(t) = False
\end{array}
$$

Ausführbare Spezifikation der Tabelle

```
module Table where
import Prelude hiding (init,read,length)

type Key = Int
type Value = String

errorvalue = "ERROR"

maxentries = 5

data Table = Empty | App(Key,Value,Table)
     deriving Show
```

Ausführbare Spezifikation der Tabelle

```
init    :: Table
insert  :: (Key,Value,Table) -> Table
isin    :: (Key,Table) -> Bool

init = Empty

insert(k,v,t) = if v == errorvalue then t
                else if isin(k,t) then t
                else if full(t) then t
                else App(k,v,t)

isin(x,Empty)       = False
isin(x,App(k,v,t)) = if x == k then True
                     else isin(x,t)
```

Ausführbare Spezifikation der Tabelle

```
read   :: (Key,Table) -> Value
empty  :: Table -> Bool
delete :: (Key,Table) -> Table

read(x,Empty)       = errorvalue
read(x,App(k,v,t)) = if x == k then v
                     else read(x,t)

empty(Empty)       = True
empty(App(k,v,t)) = False

delete(x,Empty)       = Empty
delete(x,App(k,v,t)) = if x == k then t
                       else App(k,v,delete(x,t))
```

Ausführbare Spezifikation der Tabelle

```
update :: (Key,Value,Table) -> Table
length :: Table -> Int
full   :: Table -> Bool

update(k,v,Empty)          = Empty
update(k,v,App(k2,v2,t)) =
   if k == k2 then App(k,v,t)
   else App(k2,v2,update(k,v,t))

length(Empty)       = 0
length(App(k,v,t)) = 1 + length(t)

full(t) = if length(t) == maxentries then True
          else False
```

Kapitel 6.3

Implementierung als Array

Implementierung als Array

- Die Einträge der Tabelle werden in einem Array gespeichert.
- Ein Element des Arrays enthält entweder eine Referenz auf einen Eintrag oder aber den Wert `null`.
- Ein Eintrag enthält dabei sowohl den Schlüssel (Typ `int`) als auch den Wert (Typ `char*`).
- Sofern die Tabelle nicht voll ist, wird beim Einfügen nach einem unbelegten Array-Element gesucht und der Eintrag dort eingefügt.

Implementierung als Array

- Der Array-Index, an dem zuletzt ein Eintrag gelöscht wurde, wird in der Variablen `last_delete` gespeichert.
- So kann zumindest der erste Einfügevorgang nach einem Löschvorgang schnell ausgeführt werden (Vorform einer **Freiliste**).
- Ansonsten wird die Suche an der Stelle im Array fortgesetzt, an der zuletzt ein Eintrag eingefügt wurde.
- Diese Stelle wird jeweils in der Variablen `last_insert` gespeichert.

Implementierung als Array

```
#define MAX_ELEMENTS 100
typedef char* element;

struct _node {
  element value;
  int key;
};
typedef struct _node node;

struct _table {
  int size;
  int last_insert;
  int last_delete;
  node* nodes[MAX_ELEMENTS];
};
typedef struct _table table;
```

Implementierung init und length

```
table* table_init() {
  table* t = malloc(sizeof(table));
  t->size = 0;
  t->last_insert = -1;
  t->last_delete = -1;
  for (int i = 0; i < MAX_ELEMENTS; i++)
    t->nodes[i] = NULL;
  return t;
}

int table_length(table* t) {
  return t->size;
}
```

Implementierung empty und full

```
int table_empty(table* t) {
  return table_length(t) == 0;
}

int table_full(table* t) {
  return table_length(t) == MAX_ELEMENTS;
}
```

Implementierung isin

```
int table_isin(int k, table* t) {
  if (table_empty(t)) {
    return 0;
  } else {
    for (int i = 0; i < MAX_ELEMENTS; i++) {
      if (t->nodes[i] != NULL) {
        if (t->nodes[i]->key == k)
          return 1;
      }
    }
  }
  return 0;
}
```

Implementierung insert

```
table* table_insert(int k, element e, table* t) {
  if (e == NULL || table_full(t) || table_isin(k, t))
    return t;

  node* n = malloc(sizeof(node));
  n->key = k;
  n->value = e;

  // continued on next slide
```

Implementierung insert

```
  // continued from previous slide

  if (t->last_delete != -1) {
    t->nodes[t->last_delete] = n;
    t->last_insert = t->last_delete;
    t->last_delete = -1;
  } else {
    do {
      t->last_insert++;
      t->last_insert %= MAX_ELEMENTS;
    } while (t->nodes[t->last_insert] != NULL);
    t->nodes[t->last_insert] = n;
  }
  t->size++;
  return t;
}
```

Implementierung read

```
element table_read(int k, table* t) {
  if (table_empty(t))
    return NULL;

  for (int i = 0; i < MAX_ELEMENTS; i++)
    if (t->nodes[i] != NULL)
      if (t->nodes[i]->key == k)
        return t->nodes[i]->value;

  return NULL;
}
```

Implementierung delete

```
table* table_delete(int k, table* t) {
  if (table_empty(t))
    return t;
  for (int i = 0; i < MAX_ELEMENTS; i++) {
    if (t->nodes[i] != NULL) {
      if (t->nodes[i]->key == k) {
        free(t->nodes[i]);
        t->nodes[i] = NULL;
        t->last_delete = i;
        t->size--;
        break;
      }
    }
  }
  return t;
}
```

Implementierung update

```
table* table_update(int k, element e, table* t) {
  if (e == NULL || table_empty(t))
    return t;

  for (int i = 0; i < MAX_ELEMENTS; i++) {
    if (t->nodes[i] != NULL) {
      if (t->nodes[i]->key == k) {
        t->nodes[i]->value = e;
        break;
      }
    }
  }
  return t;
}
```

Implementierung print

```
void table_print(table* t) {
  int e = 0;
  node* n;
  for (int i = 0; i < MAX_ELEMENTS; i++) {
    n = t->nodes[i];
    if (n != NULL) {
      printf("insert(%i,%s,", n->key, n-> value);
      e++;
    }
  }
  printf("init");
  for (int i = 0; i < e; i++)
    printf(")");
}
```

Implementierung print

```
void table_destroy(table* t) {
  for (int i = 0; i < MAX_ELEMENTS; i++)
    if (t->nodes[i] != NULL)
      free(t->nodes[i]);
  free(t);
}
```

Exemplarische Implementierung `main`

```
int main(int argc, char* argv[]) {
  table* t = table_init();
  table_insert(5, "Affe", t);
  table_insert(8, "Giraffe", t);
  table_insert(2, "Ente", t);
  table_insert(3, "Tiger", t);
  printf("%i\n", table_isin(6, t)); // 0
  printf("%i\n", table_isin(8, t)); // 1
  printf("%s\n", table_read(6, t)); // NULL
  printf("%s\n", table_read(8, t)); // Giraffe
  table_delete(8, t);
  printf("%s\n", table_read(8, t)); // NULL
  table_update(5, "Katze", t);
  printf("%s\n", table_read(5, t)); // Katze
  table_print(t);
  table_destroy(t);
}
```

Diskussion der Implementierung als Array

- Der Hauptvorteil der Implementierung liegt in ihrer Einfachheit.
- Nachteilig ist, dass viele Operationen linearen Aufwand erfordern.
- Lediglich `empty`, `full` und `length` benötigen konstanten Aufwand.
- Durch Verwendung einer Freiliste kann auch der Aufwand für `insert` auf konstanten Aufwand gedrückt werden.
- Für den Anwendungsfall Tabelle gibt es aber deutlich effizientere Datenstrukturen.
- Hierzu gehören binäre Suchbäume, die nachfolgend besprochen werden, sowie Hashing, das im nächsten Semester behandelt wird.

Aufwand der Tabellenoperationen

Funktion	Aufwand
`init`	linear
`insert`	linear
`read`	linear
`delete`	linear
`update`	linear
`isin`	linear
`empty`	konstant
`full`	konstant
`length`	konstant

Kapitel 6.4

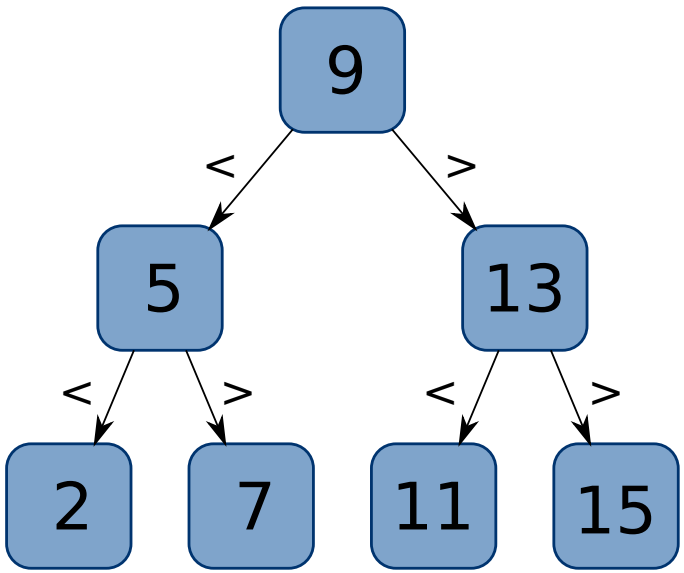
Implementierung als binärer Suchbaum

Binärer Suchbaum

Definition 2 (Binärer Suchbaum / engl. binary search tree (BST))

Ein **binärer Suchbaum** *ist ein binärer Baum, bei dem*

1. *die Knoten des linken Teilbaums jedes Knotens nur kleinere Schlüssel als der Knoten selbst besitzen und*
2. *die Knoten des rechten Teilbaums jedes Knotens nur größere Schlüssel als der Knoten selbst besitzen.*



- Hinweis: Wir betrachten hier nur BSTs ohne Duplikate, d. h. mit eindeutigen Schlüsseln.
- Die Inorder-Traversierung eines BSTs liefert stets die sortierte Folge der Schlüssel:

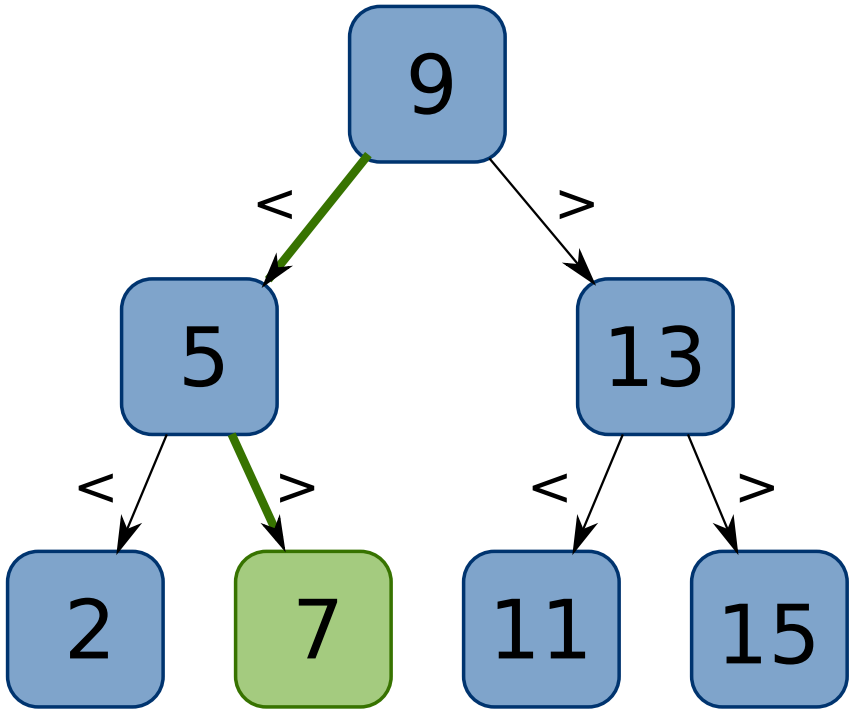
$$[2, 5, 7, 9, 11, 13, 15]$$

Suche in einem Binären Suchbaum

- Um einen Schlüssel in einem BST zu finden, wird dieser mit dem Schlüssel des Wurzelknotens verglichen.
- Sind beide Schlüssel gleich, so wurde der Schlüssel gefunden.
- Ist der gesuchte Schlüssel kleiner (größer), so wird die Suche rekursiv im linken (rechten) Teilbaum fortgesetzt.
- Die Suche terminiert, wenn der Schlüssel gefunden oder ein Knoten erreicht wird, bei dem die Suche nicht fortgesetzt werden kann.
- Letzteres ist der Fall, wenn der gesuchte Schlüssel kleiner (größer) als der des betrachteten Knotens ist und das linke (rechte) Kind dieses Knotens nicht existiert.
- In diesem Fall ist der gesuchte Schlüssel nicht im Baum enthalten.

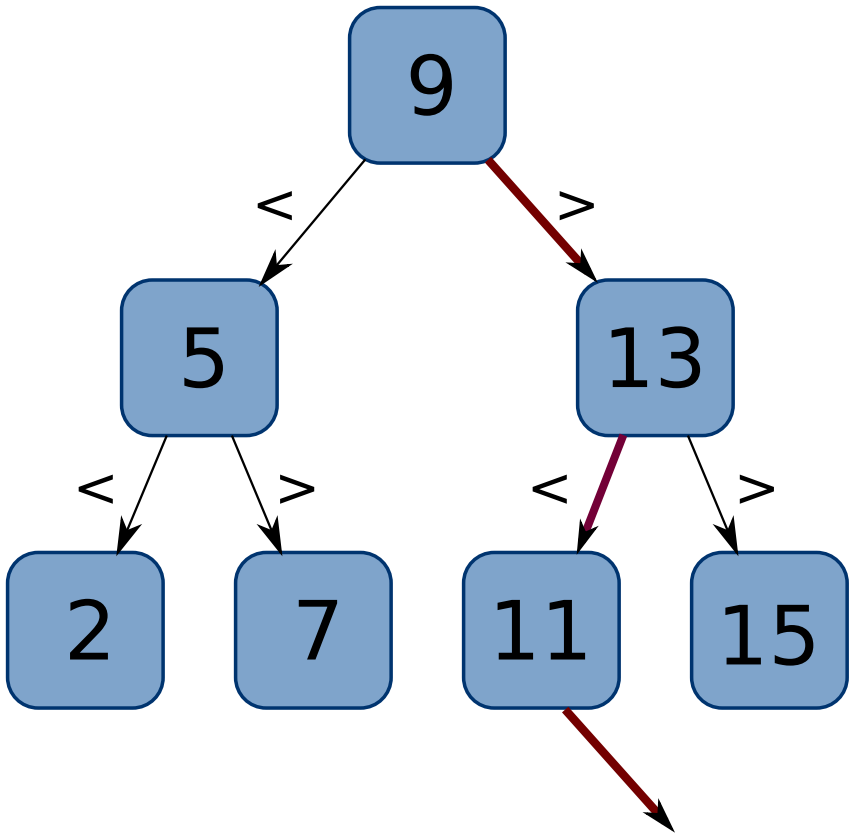
Suche in einem Binären Suchbaum

- Die Suche nach dem Schlüssel 7 führt vom Wurzelknoten aus in den linken Teilbaum, da $7 < 9$.
- Der nächste Schritt führt in den rechten Teilbaum, da $7 > 5$, und findet den Schlüssel 7.



Suche in einem Binären Suchbaum

- Die Suche nach dem Schlüssel 12 führt vom Wurzelknoten aus in den rechten Teilbaum, da $12 > 9$.
- Der nächste Schritt führt in den linken Teilbaum, da $12 < 13$.
- Dort terminiert die Suche erfolglos, da $12 > 11$ ist, aber das rechte Kind dieses Knotens nicht existiert.



Suche in einem Binären Suchbaum

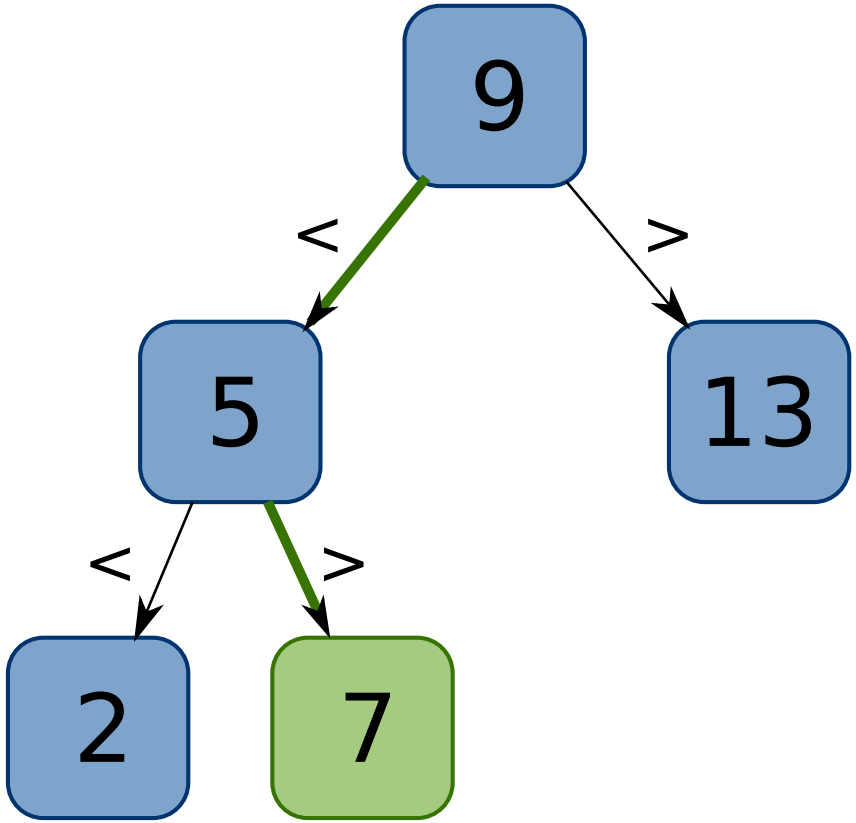
- Der Aufwand für die Suche in einem binären Suchbaum beträgt $O(h)$ und ist damit abhängig von der Höhe des Baums h.
- Die Höhe eines binären Baums beträgt im besten Fall $O(\log n)$ bei einem balancierten Baum und im schlechtesten Fall $O(n)$, zum Beispiel bei einem zu einer Kette entarteten Baum.
- Die Höhe eines binären Suchbaums hängt von der Vorgehensweise beim Einfügen sowie beim Entfernen von Schlüsseln ab.
- Durch eine geeignete Vorgehensweise kann eine Balancierung sichergestellt werden.
- Wir betrachten im Folgenden aber ein einfacheres Verfahren.

Einfügen in einen Binären Suchbaum

- Beim Einfügen eines Schlüssels in einen binären Suchbaum wird zunächst so vorgegangen, wie bei der Suche nach diesem Schlüssel.
- Die Suche terminiert dann bei einem Knoten, weil eine Fortsetzung der Suche nicht möglich ist, da das entsprechende Kind nicht existiert.
- Sie terminiert also immer bei einem Blatt (beide Kinder fehlen) oder einem Halbblatt (ein Kind fehlt).
- Bei diesem Knoten wird dann der neue Schlüssel als linkes (rechtes) Kind als Blatt eingefügt, wenn der neue Schlüssel kleiner (größer) als der Schlüssel des Knotens ist.

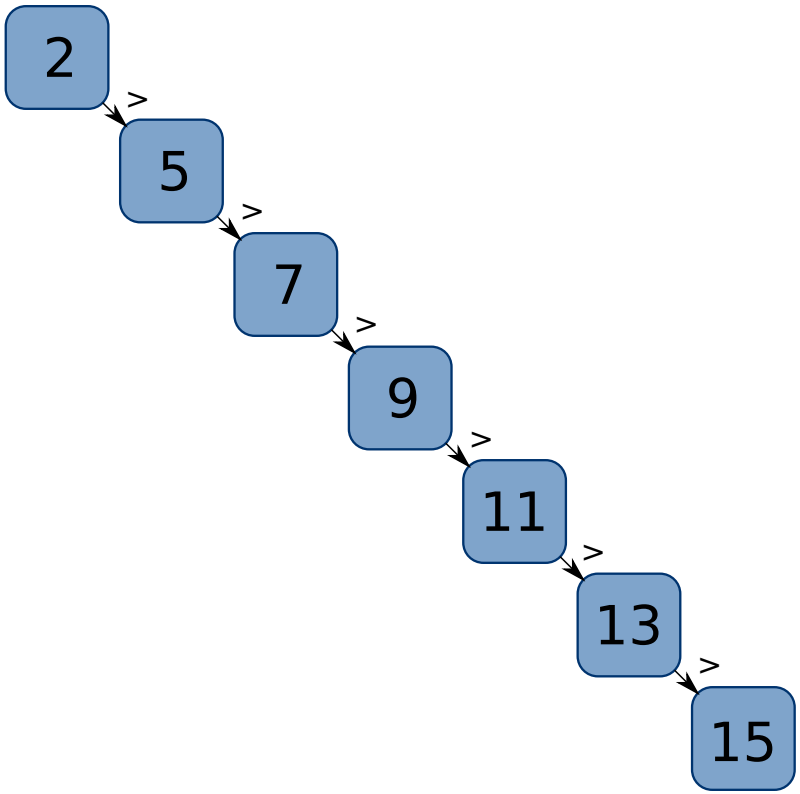
Einfügen in einen Binären Suchbaum

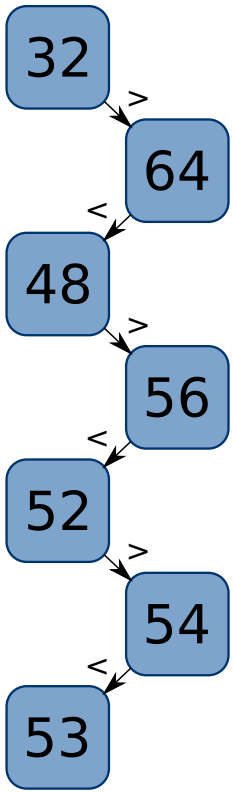
- Der Schlüssel 7 wird als rechtes Kind des Knotens mit dem Schlüssel 5 einfügt, da $7 < 9$ und $7 > 5$ ist und das rechte Kind des Knotens mit Schlüssel 5 nicht existiert.



Einfügen in einen Binären Suchbaum

- Der Aufwand beim Einfügen ist wieder $O(h)$.
- Donald E. Knuth wies 1973 nach, dass durch das Einfügen zufällig sortierter Elemente ein BST mit logarithmischer Höhe entsteht.
- Durch wiederholtes Einfügen von aufsteigend (oder absteigend) sortierten Schlüsseln kann der Baum hingegen zu einer Kette entarten.





Löschen in einem Binären Suchbaum

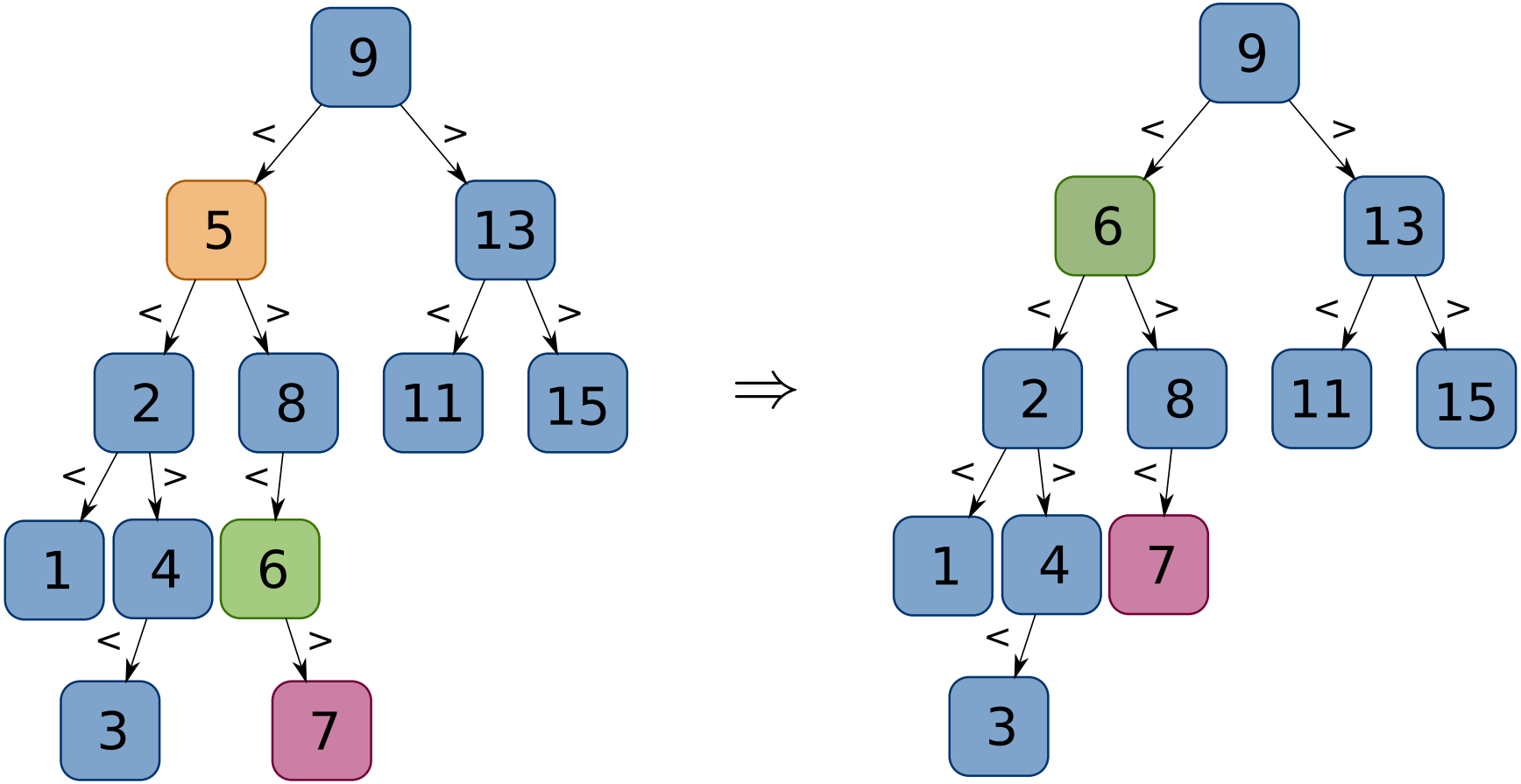
- Hier werden drei Fälle unterschieden, je nachdem ob der zu löschende Knoten ein Blatt, ein Halbblatt oder ein innerer Knoten ist.
  1. Ist der zu löschende Knoten ein Blatt, dann wird der Knoten einfach ersatzlos aus dem Baum entfernt.
  2. Ist der zu löschende Knoten ein Halbblatt, dann wird das (einzige) Kind des zu löschenden Knotens an die Stelle dieses Knotens gesetzt.
  3. Ist der zu löschende Knoten hingegen ein innerer Knoten, hat er also zwei Kinder, so ist das Löschen komplizierter (siehe folgende Folien).

Löschen eines inneren Knotens aus einem BST

- Hier wird der zu löschende Knoten durch den *minimalen* Knoten seines *rechten* Teilbaums ersetzt. Der evtl. nicht-leere rechte Teilbaum des Ersatzknotens tritt dann an die Stelle des Ersatzknotens.
- Hinweis: Der linke Teilbaum des minimalen Knotens ist immer leer!
- Alternativ kann der Knoten auch durch den *maximalen* Knoten seines *linken* Teilbaums ersetzt werden. Dann tritt dessen potentiell nicht-leerer linke Teilbaum an die Stelle des Ersatzknotens.
- Hinweis: Der rechte Teilbaum des maximalen Knotens ist immer leer!
- Der Ersatzknoten ist also der nächstgrößere bzw. der nächstkleinere Knoten im Teilbaum des zu löschenden Knotens.
- Beide Alternativen sollten alternierend angewendet werden, um dem Entarten des Baums entgegenzuwirken.
- Der Aufwand für das Löschen beträgt $O(h)$.

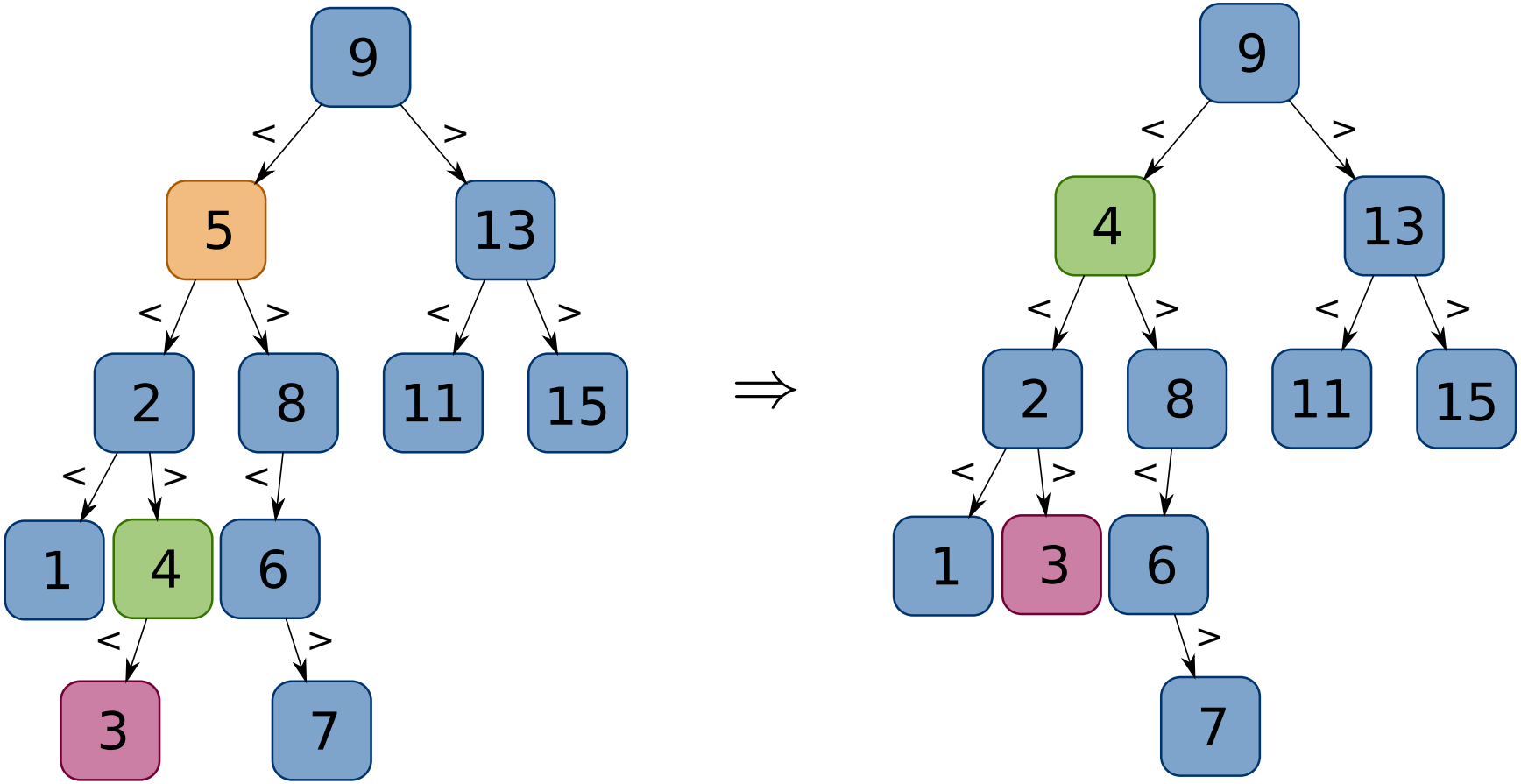
Löschen eines inneren Knotens aus einem BST

- Im folgenden BST soll der Knoten mit Schlüssel 5 gelöscht werden.
- Bei Wahl des Knotens mit minimalem Schlüssel (6) des rechten Teilbaums ergibt sich folgende Transformation:



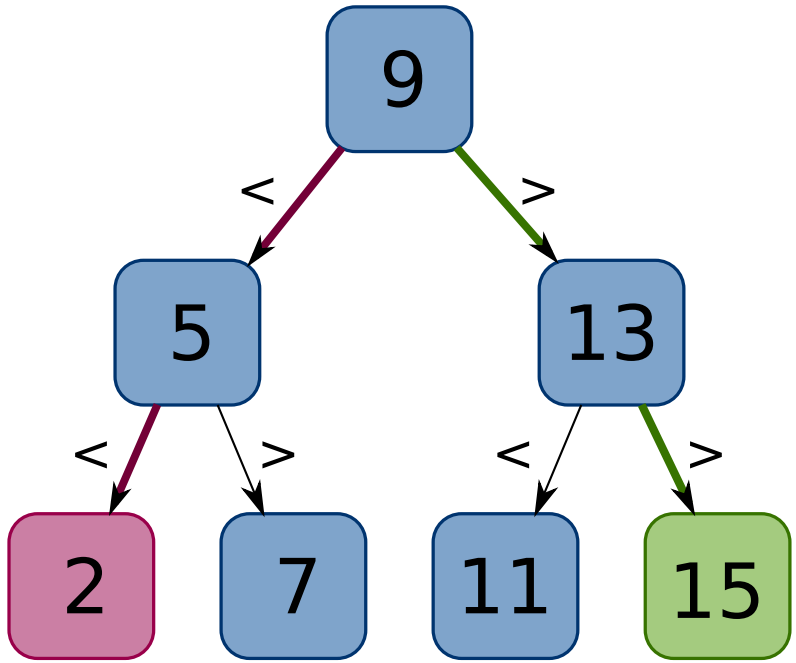
Löschen eines inneren Knotens aus einem BST

- Im folgenden BST soll der Knoten mit Schlüssel 5 gelöscht werden.
- Bei Wahl des Knotens mit maximalem Schlüssel (4) des linken Teilbaums ergibt sich folgende Transformation:



Finden des minimalen/maximalen Schlüssels

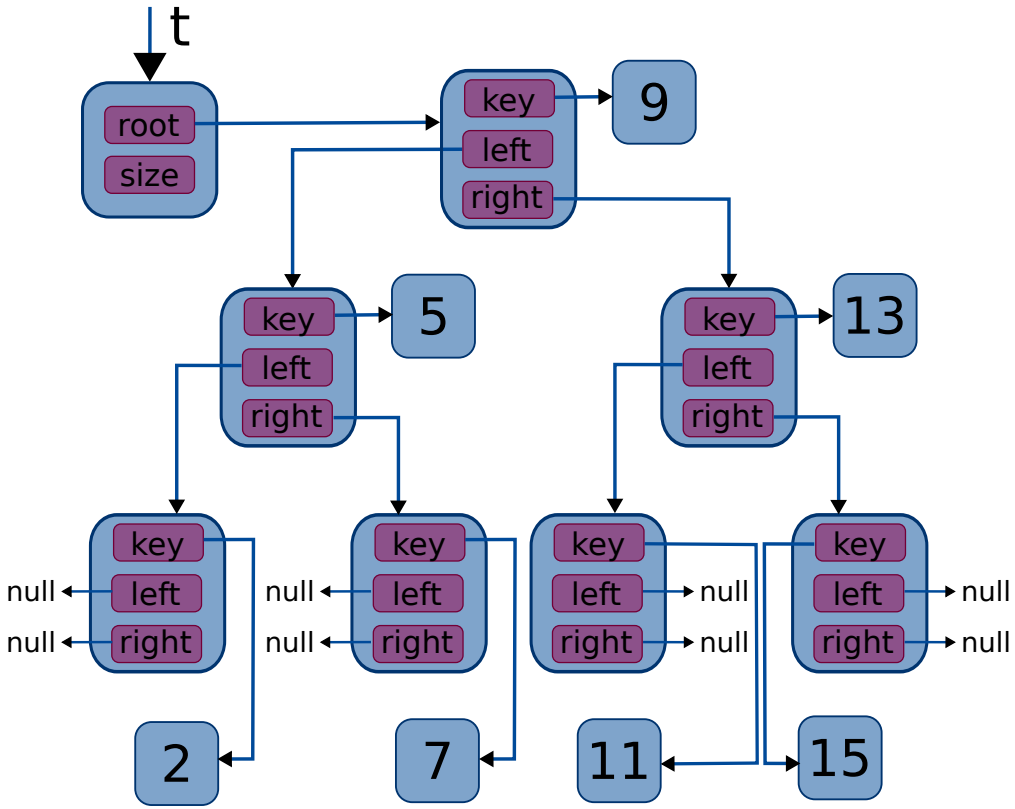
- Der minimale Schlüssel in einem BST wird gefunden, indem von der Wurzel aus solange wie möglich immer nur nach links gegangen wird.
- Der maximale Schlüssel kann analog durch den Abstieg immer nach rechts gefunden werden.
- Beide Operation benötigen einen Aufwand von $O(h)$.



Implementierung einer Tabelle als binärer Suchbaum

- Realisierung des Baums als verkettete Datenstruktur.
  - Tabelle hat Referenz auf Wurzelknoten des Baums.
  - Knoten hat Referenz auf linken und rechten Kindknoten.

```
struct _table {
  node* root;
  int size;
};

struct _node {
  element value;
  int key;
  _node* left;
  _node* right
};
```



- Hinweis: Abb. zeigt nur die Schlüssel und *nicht* die Wertobjekte.

Implementierung `init`

```
#define MAX_ELEMENTS 100
typedef char* element;
struct _node {
        element value;
        int key;
        _node* left;
        _node* right;
};
typedef struct _node node;
struct _table {
        node* root;
        int size;
};
typedef struct _table table;

table* tableInit() {
  table* t = malloc(sizeof(table));
  t->root = NULL;
  t->size = 0;
  return t;
}
```

Implementierung `length`, `empty` und `full`

```
int tableLength(table* t) { return t->size; }

int tableEmpty(table* t) {
  return tableLength(t) == 0;
}

int tableFull(table* t) {
  return tableLength(t) == MAX_ELEMENTS;
}
```

Implementierung isin

```
int tableIsin(int k, table* t) {
  if (tableEmpty(t))
  return 0;

  node* n = t->root;
  while (n != NULL) {
    if (k == n->key) { // key found
      return 1;
        } else if (k < n->key) { // left subtree
      n = n->left;
        } else if (k > n->key) { // right subtree
          n = n->right;
        }
  }
  // key not found
  return 0;
}
```

Implementierung insert

```
table* tableInsert(int k, element e, table* t) {
  if (e == NULL || tableFull(t) || tableIsin(k, t))
    return t;
  node* m = malloc(sizeof(node));
  m->key = k;
  m->value = e;
  m->left = NULL;
  m->right = NULL;
  if (tableEmpty(t)) {
    t->root = m;
  } else {
  // continued on next slide
```

Implementierung `insert`

```
  node* n = t->root;
  while (n != NULL) {
    if (k < n->key) { // insert left
      if (n->left == NULL) {
        n->left = m;
        break;
      } else
        n = n->left;
      } else if (k > n->key) { // insert right
        if (n->right == NULL) {
          n->right = m;
          break;
        } else
          n = n->right;
        }
    }
  }
  t->size++;
  return t;
}
```

Implementierung read

```
element tableRead(int k, table* t) {
  if (tableEmpty(t))
    return NULL;
  node* n = t->root;
  while (n != NULL) {
    if (k == n->key) { // key found
          return n->value;
        } else if (k < n->key) { // left subtree
          n = n->left;
        } else if (k > n->key) { // right subtree
          n = n->right;
        }
  }
  return NULL;
}
```

Implementierung update

```
table* tableUpdate(int k, element e, table* t) {
  if (e == NULL || tableEmpty(t))
  return t;
  node* n = t->root;
  while (n != NULL) {
    if (k == n->key) { // key found
      n->value = e;
      break;
    } else if (k < n->key) { // left subtree
      n = n->left;
    } else if (k > n->key) { // right subtree
      n = n->right;
    }
  }
  return t;
}
```

Implementierung delete

```
table* tableDelete(int k, table* t) {
  if (tableEmpty(t) || !tableIsin(k, t))
  return t;
  t->root = tableDeleteNode(k, t->root);
  t->size--;
  return t;
}
```

Implementierung deleteNode

```
node* tableDeleteNode(int k, node* n) {
  if (k < n->key) {
    // node to be deleted is in left subtree;
    // if node to be deleted is left child,
    // a new left subtree is attached
    n->left = tableDeleteNode(k, n->left);
    return n;
  }

  if (k > n->key) {
    // node to be deleted is in right subtree
    // if node to be deleted is right child,
    // a new right subtree is attached
    n->right = tableDeleteNode(k, n->right);
    return n;
  }
  // continued on next slide
```

Implementierung deleteNode

```
// n->key == k
// node to be deleted was found

if (n->left == null) {
   // node is leaf
   // or half-leaf with right child
   return n->right;
}
// n->left != null => node is not a leaf
if (n->right == null) {
   // node is half-leaf with left child
   return n->left;
}
// continued on next slide
```

Implementierung deleteNode

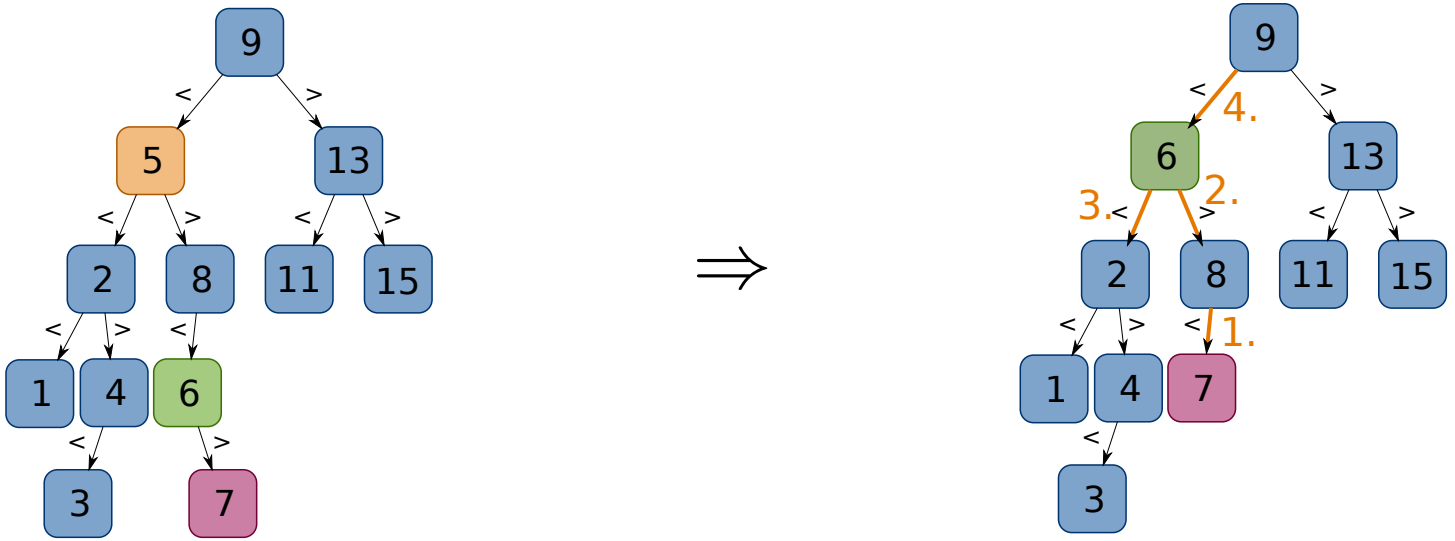
```
    // node is inner node

    // get min node from right subtree
    node* min = tableFindMin(n->right);

    // delete min node from right subtree
    // and attach right subtree of n to min
    min->right = tableDeleteNode(min->key, n->right);

    // attach left subtree of n to min
    min->left = n->left;

    // return min as replacement node for n
    return min;
}
```

Implementierung findMin

```
node* tableFindMin(node* n) {
  // find min node starting from node n
  if (n != NULL)
  while (n->left != NULL)
    n = n->left;
  return n;
}
```

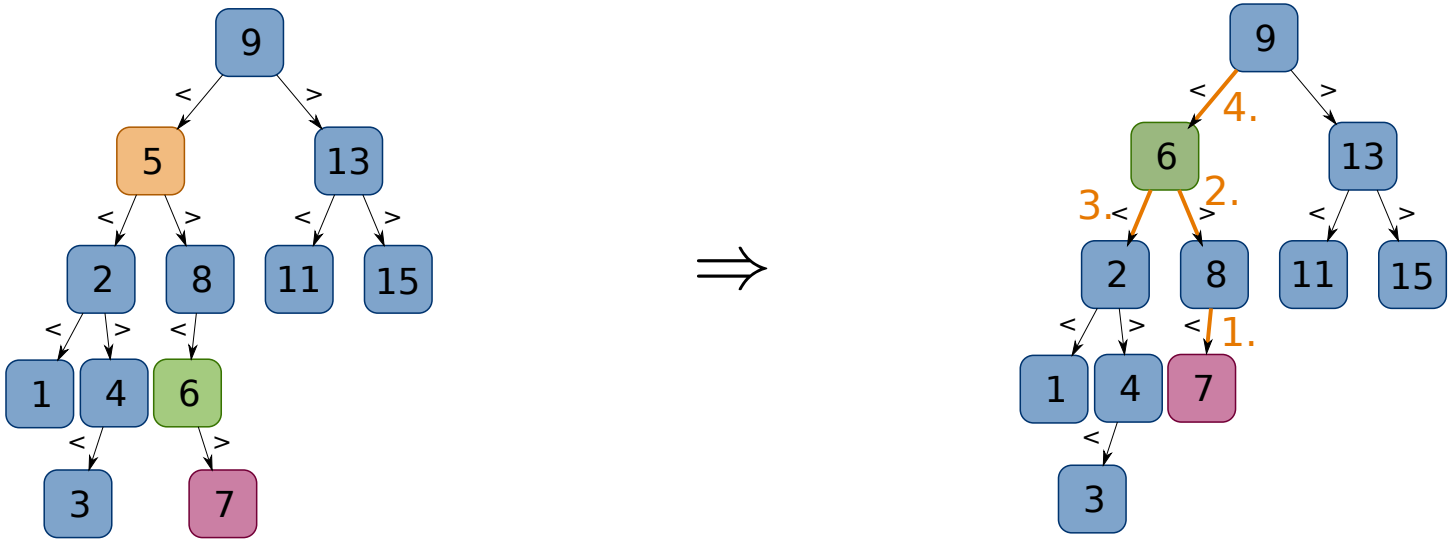
Beispiel deleteNode (Aufrufe)

```
deleteNode(5,"9")
    "9".left = deleteNode(5,"5")
                    min = findMin("5".right ≡ "8")
                              return "6"
                    "6".right = deleteNode(6,"5".right ≡ "8")
                                        "8".left = deleteNode("6".key ≡ 6,
                                                              "8".left ≡ "6")
                                                   return "7"
                                        return "8"
                    "6".left = "5".left ≡ "2"
                    return "6"
    return "9"
```



Beispiel `deleteNode` (resultierende Zuweisungen)

```
"8".left  = "7"    // attach right subtree of min node
                   // instead of min node

"6".right = "8"    // attach right subtree of deleted node
                   // to min node

"6".left  = "2"    // attach left subtree of deleted node
                   // to min node

"9".left  = "6"    // attach min node as new left subtree of
                   // root node
```



Alternative Implementierung von deleteNode

```
public static Table deleteNode(int k, Table t) {
    if (empty(t) || !isin(k, t)) return t;

    Node parent = findParentNode(k, t.root);
    Node n = null, right = null, left = null;
    boolean isLeftChild = false, isRoot = false;
    // find node n to be deleted
    if (k < parent.key) {
        isLeftChild = true;
        n = parent.left;
    } else if (k > parent.key)
        n = parent.right;
    else { // (k == parent.key)
        isRoot = true;
        n = parent;
    }
    // continued on next slide
```

Alternative Implementierung von deleteNode

```
if ((n.left != null) && (n.right != null)) {
    // n is inner node
    right = n.right;
    left = n.left;
    n = findMin(right);
    n.right = deleteMin(right);
    n.left = left;
} else {
   // n is leaf or half leaf
   n = (n.left != null) ? n.left : n.right;
}
// continued on next slide
```

Alternative Implementierung von deleteNode

```
        if (isRoot)
            t.root = n;
        else if (isLeftChild)
            parent.left = n;
        else
            parent.right = n;

        t.size--;

        return t;
    }
```

Implementierung von findParentNode

```
private static Node findParentNode(int k, Node n) {
    // returns parent node of node with key k
    // starting at node n
    if (n != null) {
        if (n.left != null) {
            if (k == n.left.key) return n;
            if (k < n.key)
                return findParentNode(k, n.left);
        }
        if (n.right != null) {
            if (k == n.right.key) return n;
            else if (k > n.key)
                return findParentNode(k, n.right);
        }
    }
    return n;
}
```

Implementierung von deleteMin

```
private static Node deleteMin(Node n) {
    if (n != null) {
        if (n.left != null)
            n.left = deleteMin(n.left);
        else
            return n.right;
    }
    return n;
}
```

Implementierung show

```
void tableShow2(node* n) {
  if (n == NULL)
  return;
  printf("%i, %s\n", n->key, n->value);
  tableShow2(n->left);
  tableShow2(n->right);
}

void tableShow(table* t) {
  printf("<<<<\n");
  if (!tableEmpty(t))
    tableShow2(t->root);
  else
    printf("empty");
  printf(">>>>\n");
}
```

Diskussion der Implementierung

- Durch Verwendung binärer Suchbäume wurde der Aufwand für `insert`, `read`, `delete`, `update` und `isin` auf $O(\log n)$ reduziert.
- Dies gilt allerdings nur für Suchbäume mit logarithmischer Höhe.

- Suchbäume, die durch das Einfügen zufälliger Schlüssel entstehen **zufällige Suchbäume**, haben im Durchschnitt logarithmische Höhe.
- Diese Eigenschaft bleibt auch unter Sequenzen von Einfüge- und Löschoperationen erhalten, wenn folgende Bedingungen erfüllt sind:
  1. Alle Permutationen von Einfüge- und Löschoperationen sind gleich wahrscheinlich.
  2. Bei Löschoperationen halten sich die Abstiege nach links und die nach rechts im Mittel die Waage.

- Balancierte Suchbäume haben immer logarithmische Höhe.
- Die Balance wird durch eine Erweiterung der vorgestellten Einfüge- und Löschoperationen sichergestellt.

Diskussion der Implementierung

- Sind die einzufügenden Schlüssel im vorhinein bekannt, so kann ein Suchbaum mit optimaler Höhe folgendermaßen erzeugt werden:
  - Wähle als Schlüssel des Wurzelelements des Suchbaums den Obermedian aller Schlüssel.
  - Konstruiere mit allen Schlüsseln, die kleiner als der Obermedian der Schlüssel sind, den linken Teilsuchbaum.
  - Konstruiere mit allen Schlüsseln, die größer als der Obermedian der Schlüssel sind, den rechten Teilsuchbaum.

Beispiel 8 (Konstruktion eines Suchbaums mit optimaler Höhe)

- Die Menge der Schlüssel sei $\{1, 3, 5, 6, 7, 8\}$.
- Die 6 wird die Wurzel. Aus der Schlüsselmenge $\{1, 3, 5\}$ wird der linke Teilbaum konstruiert, aus der Schlüsselmenge $\{7, 8\}$ der rechte.
- Die 3 wird die Wurzel des linken Teilbaum, die 1 ihr linkes und die 5 ihr rechtes Kind.
- Die 8 wird die Wurzel des rechten Teilbaums, die 7 ihr linkes Kind.

Aufwand der Tabellenoperationen

Funktion	Aufwand
`init`	konstant
`insert`	logarithmisch
`read`	logarithmisch
`delete`	logarithmisch
`update`	logarithmisch
`isin`	logarithmisch
`empty`	konstant
`full`	konstant
`length`	konstant

- Der logarithmische Aufwand gilt nur für Suchbäume mit logarithmischer Höhe, ansonsten ist der Aufwand linear.

Exemplarische Fragen zur Lernkontrolle

1. Wozu dient eine Tabelle?
2. Welche Operation bietet eine Tabelle typischerweise an?
3. Spezifizieren Sie alle grundlegenden Operationen der Tabelle!
4. Erläutern Sie die Implementierung der Tabelle als Array und als binärer Suchbaum!
5. Wie funktioniert das Einfügen, das Suchen und das Löschen eines Elements bei einem binären Suchbaum?
6. Unter welcher Annahme hat ein binärer Suchbaum die Höhe $O(\log n)$?
7. Welche Höhe hat ein entarteter binärer Suchbaum?
8. Wie kann bei bekannten Elementen ein binärer Suchbaum mit optimaler Höhe konstruiert werden?

Vielen Dank für Ihre Aufmerksamkeit!

Univ.-Prof. Dr.-Ing. Gero Mühl

gero.muehl@uni-rostock.de
https://www.ava.uni-rostock.de