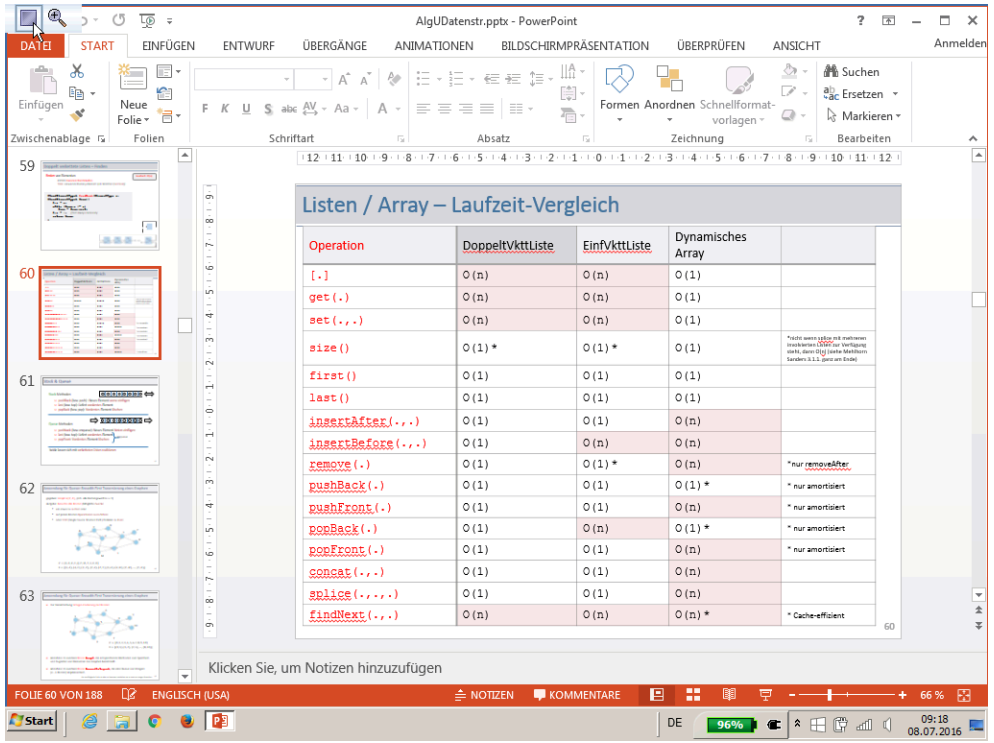
Script generated by TTT

Title: groh: profile1 (08.07.2016)

Date: Fri Jul 08 09:18:53 CEST 2016

Duration: 88:27 min

Pages: 70



Hashing

- leider in Praxis: viele leere Tabelleneinträge, Kollisionen (keys werden auf gleichen Index abgebildet)
- Wahrscheinlichkeit von Kollisionen: Annahme: randomisierte Hash-Funktion, n keys sollen auf m Indices verteilt werden:

$$P(\text{keine Kollision beim i}_{\text{ten}}\text{Schlüssel}) = \frac{m-(i-1)}{m}$$

$$P(\text{keine Kollsion bei n Schlüsseln}) = \prod_{i=1}^{n} \frac{m-(i-1)}{m} = \prod_{i=0}^{n-1}(1-\frac{i}{m})$$

Bsp: für $n = 23$ und $m = 365$ ist $P(\text{keine Kollision}) < 0.5$

Konstruktion von Hashfunktionen

Beispiel für eine Familie von guten Hashfunktionen h_a
parametrisiert durch Vektor a von g Integer-Zahlen: $a = (a_1, a_2, \dots, a_g)$:

- Wähle m als Primzahl (Grund: siehe [10],[11])
- Interpretiere Schlüssel von f Bits als Tupel von g Binärzahlen $(x_1, x_2, \dots, x_g)$

  Bsp. f=32, g = 4: $\underbrace{11011010}_{x_1=218}\ \underbrace{01001011}_{x_2=75}\ \underbrace{01101101}_{x_3=109}\ \underbrace{00110111}_{x_4=55}$

- h_a ist dann über Skalarprodukt und modulo-m-Operation definiert:
  $$h_a(x) = (a * x)\ mod\ m$$
- → beweisbar wenig Kollisionen (siehe [10],[11])

Konstruktion von Hashfunktionen

Beispiel für eine Familie von guten Hashfunktionen h_a
parametrisiert durch Vektor a von g Integer-Zahlen: $a = (a_1, a_2, \dots, a_g)$:

- Wähle m als Primzahl (Grund: siehe [10],[11])
- Interpretiere Schlüssel von f Bits als Tupel von g Binärzahlen $(x_1, x_2, \dots, x_g)$

  Bsp. f=32, g = 4: 11011010 01001011 01101101 00110111

  x_1=218 x_2=75 x_3=109 x_4=55

- h_a ist dann über Skalarprodukt und modulo-m-Operation definiert:

  $$h_a(x) = (a * x) \bmod m$$

- → beweisbar wenig Kollisionen (siehe [10],[11])

Konstruktion von Hashfunktionen

Beispiel für eine Familie von guten Hashfunktionen h_a
parametrisiert durch Vektor a von g Integer-Zahlen: $a = (a_1, a_2, \dots, a_g)$:

- Wähle m als Primzahl (Grund: siehe [10],[11])
- Interpretiere Schlüssel von f Bits als Tupel von g Binärzahlen $(x_1, x_2, \dots, x_g)$

  Bsp. f=32, g = 4: 11011010 01001011 01101101 00110111

  x_1=218 x_2=75 x_3=109 x_4=55

- h_a ist dann über Skalarprodukt und modulo-m-Operation definiert:

  $$h_a(x) = (a * x) \bmod m$$

- → beweisbar wenig Kollisionen (siehe [10],[11])

126

Konstruktion von Hashfunktionen

Beispiel für eine Familie von guten Hashfunktionen h_a
parametrisiert durch Vektor a von g Integer-Zahlen: $a = (a_1, a_2, \dots, a_g)$:

- Wähle m als Primzahl (Grund: siehe [10],[11])
- Interpretiere Schlüssel von f Bits als Tupel von g Binärzahlen $(x_1, x_2, \dots, x_g)$

  Bsp. f=32, g = 4: 11011010 01001011 01101101 00110111

  x_1=218 x_2=75 x_3=109 x_4=55

- h_a ist dann über Skalarprodukt und modulo-m-Operation definiert:

  $$h_a(x) = (a * x) \bmod m$$

- → beweisbar wenig Kollisionen (siehe [10],[11])

126

Konstruktion von Hashfunktionen

Beispiel für eine Familie von guten Hashfunktionen h_a
parametrisiert durch Vektor a von g Integer-Zahlen: $a = (a_1, a_2, \dots, a_g)$:

- Wähle m als Primzahl (Grund: siehe [10],[11])
- Interpretiere Schlüssel von f Bits als Tupel von g Binärzahlen $(x_1, x_2, \dots, x_g)$

  Bsp. f=32, g = 4: 11011010 01001011 01101101 00110111

  x_1=218 x_2=75 x_3=109 x_4=55

- h_a ist dann über Skalarprodukt und modulo-m-Operation definiert:

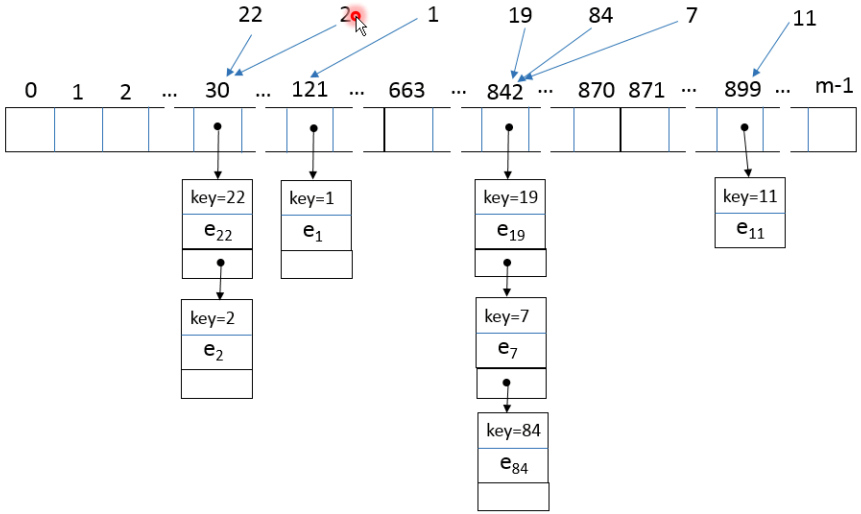
  $$h_a(x) = (a * x) \bmod m$$

- → beweisbar wenig Kollisionen (siehe [10],[11])

126

Kollisionen: Lösung 1: Chaining

Idee: statt Array von Elementen bzw. (Key, Element)-Items
nun Array von Sequenzen (bspw. verkettete Listen) von
Elementen bzw. (Key, Element)-Items



Hashing

- Anforderungen an Hashfunktion:
  - platzsparend (bspw. u.a. im Idealfall surjektiv)
  - gute Streuung / Verteilung über Tabelle
  - effizient berechenbar
  - ...
- Idealfall: h in O(1) berechenbar und jedes Element e alleine unter Index h(key(e)) gespeichert → find, insert, remove in O(1) realisierbar

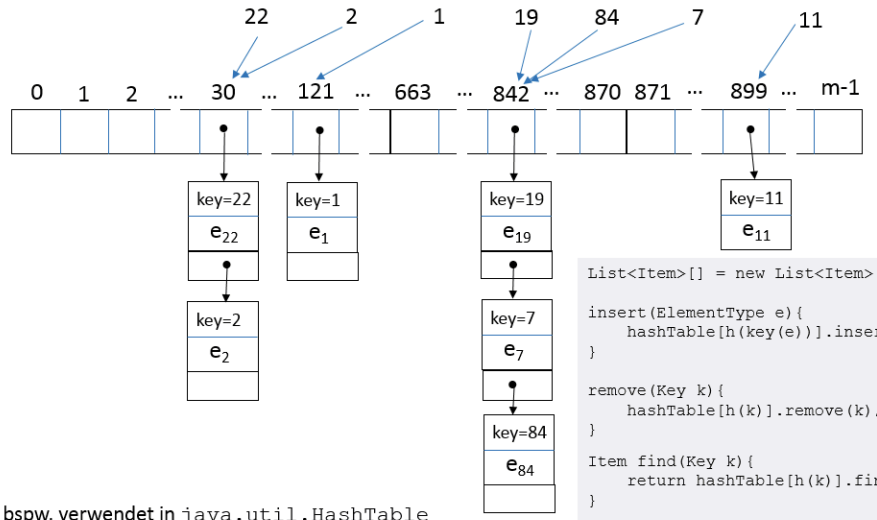
```
void insert(ElementType e){
    hashTable[h(key(e))] = e; (1)
}

void remove(Key k){
    hashTable[h(k)] = null;
}
```

```
Item<ElementType,Key> find(Key k){
    return hashTable[h(k)];
}
```

(1) Genauer müsste hier natürlich stehen: `hashTable[h(key(e))] = new Item(key(e),e);` dann sollte (um zweimaliges Berechnen zu vermeiden) key(e) natürlich zwischengespeichert werden.

Kollisionen: Lösung 1: Chaining

Idee: statt Array von Elementen bzw. (Key, Element)-Items
nun Array von Sequenzen (bspw. verkettete Listen) von
Elementen bzw. (Key, Element)-Items



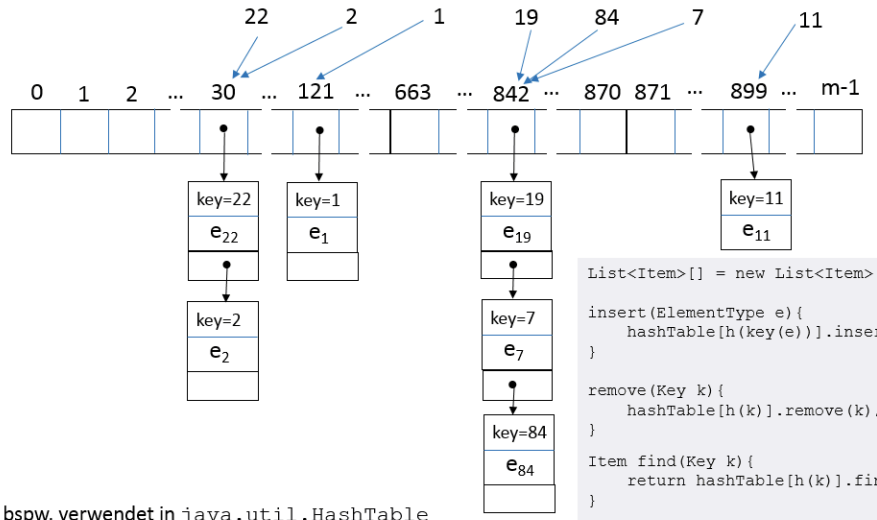

```
List<Item>[] = new List<Item>[m];

insert(ElementType e){
    hashTable[h(key(e))].insert(e); (1)
}

remove(Key k){
    hashTable[h(k)].remove(k);
}

Item find(Key k){
    return hashTable[h(k)].find(k);
}
```

(1) Siehe entsprechende Bemerkung auf Folie 119)

bspw. verwendet in `java.util.HashTable`

Kollisionen: Lösung 1: Chaining

Idee: statt Array von Elementen bzw. (Key, Element)-Items
nun Array von Sequenzen (bspw. verkettete Listen) von
Elementen bzw. (Key, Element)-Items



List<Item>[] = new List<Item>[m];

insert(ElementType e){
    hashTable[h(key(e))].insert(e); (1)
}

remove(Key k){
    hashTable[h(k)].remove(k);
}

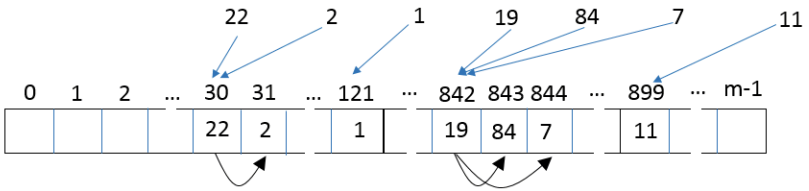
Item find(Key k){
    return hashTable[h(k)].find(k);
}

(1) Siehe entsprechende Bemerkung auf Folie 119)

bspw. verwendet in `java.util.HashTable`

Kollisionen: Lösung 2: (Linear) Probing

Idee: speichere Element e mit i==h(key(e)) am ersten freien Index i, i+1, i+2, ...

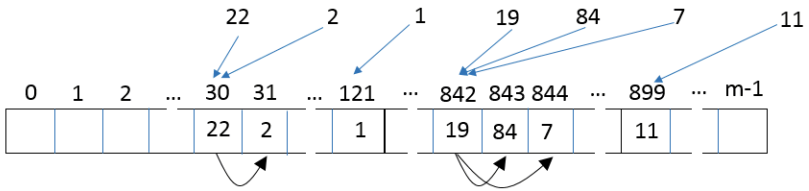



```
insert(ElementType e){
    i = h(key(e));
    while((hashTable[i] != null) && (hashTable[i].element != e))
        i = (i+1) % m;
    hashTable[i] = new Item(key(e),e);
}
```

```
Item find(Key k){
    i = h(k);
    while((hashTable[i] != null) && (hashTable[i].key != k))
        i = (i+1) % m;
    return hashTable[i];
}
```


Kollisionen: Lösung 2: (Linear) Probing

Idee: speichere Element e mit i==h(key(e)) am ersten freien Index i, i+1, i+2, ...

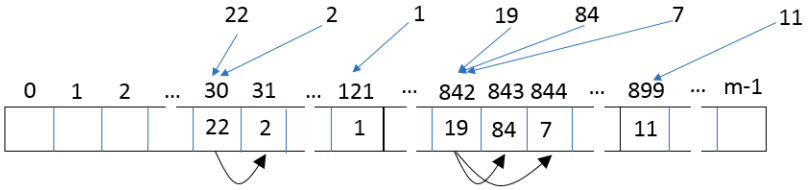



```
insert(ElementType e){
    i = h(key(e));
    while((hashTable[i] != null) && (hashTable[i].element != e))
        i = (i+1) % m;
    hashTable[i] = new Item(key(e),e);
}
```

```
Item find(Key k){
    i = h(k);
    while((hashTable[i] != null) && (hashTable[i].key != k))
        i = (i+1) % m;
    return hashTable[i];
}
```


Kollisionen: Lösung 2: (Linear) Probing

Idee: speichere Element e mit i==h(key(e)) am ersten freien Index i, i+1, i+2, ...

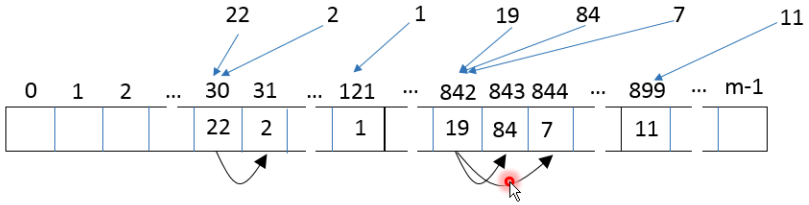



```
insert(ElementType e){
    i = h(key(e));
    while((hashTable[i] != null) && (hashTable[i].element != e))
        i = (i+1) % m;
    hashTable[i] = new Item(key(e),e);
}
```

```
Item find(Key k){
    i = h(k);
    while((hashTable[i] != null) && (hashTable[i].key != k))
        i = (i+1) % m;
    return hashTable[i];
}
```


Kollisionen: Lösung 2: (Linear) Probing

Idee: speichere Element e mit i==h(key(e)) am ersten freien Index i, i+1, i+2, ...



Achtung: remove erfordert ggf. lokale Anpassungen:
Für jedes Element e mit idealem Index i == h(key(e)), das am Index j gespeichert wurde, muss gesichert sein, dass die Indices i, i+1, ..., j besetzt sind.

Änderung der Hashtablegröße

- Damit Hashing möglichst kollisionsfrei: n < m, aber m nicht zu groß (Speicherverschwendung).

- Wenn m zu klein oder zu groß: Reallokation:
  - Wähle neue Hashtabellengröße m‘ , wobei m‘ eine Primzahl sein sollte (Grund: siehe [11]) (ist amortisiert effizient möglich)
  - Wähle neue Hashfunktion h: $K \rightarrow [0, m' - 1]$
  - Übertrage Elemente (bzw. Items) in neue Hashtabelle

Perfektes Hashing

- Ziel: Sequenzen (im Fall Chaining) bzw. Folgen besetzter Positionen (im Fall lin. Probing) sollten möglichst kurz sein

- Wenn dies nicht der Fall ist: find kann zur linearen Suche über alle Elemente ausarten → worst case: O(n) (expected / average: O(1))

- → Ziel: perfektes Hashing ohne Kollisionen (dann → find: garantiert O(1))

- Idee (zumindest bei statischer Hashtabellengröße m, statischer Zahl von Elementen n): verwende zweistufiges Hashing:
  - h_1 mit wenig Kollisionen aber guter Ausnutzung der m Indices: Bildet auf Buckets (=kleinere Hashtabellen) von konstanter durchschnittlicher Größe ab
  - je ein h_2 ohne Kollisionen für jedes Bucket um in jedem die Kollisionen aus der 1. Stufe aufzulösen

Aufgabe Hashing → Hashing with Chaining

In einer **fiktiven Hashtabelle ohne Kollisionen** wären die Operationen insert, remove und find so implementierbar:

```
void insert(ElementType e){
    hashTable[h(key(e))] = e;
}

void remove(Key k){
    hashTable[h(k)] = null;
}

Item<ElementType,Key> find(Key k){
    return hashTable[h(k)];
}
```

In einer Hashtabelle **mit Chaining** müssen die **Operationen angepasst** werden (siehe umseitig). **Ergänzen Sie** die fehlenden Anweisungen an zwei gekennzeichneten Stellen entsprechend!

```
List<Item>[] = new List<Item>[m];

insert(ElementType e){
    hashTable[h(key(e))].insert(e);
}

remove(Key k){
    [          ];
}

Item find(Key k){
    return [          ];
}
```


Aufgabe Hashing → Hashing with Chaining

In einer **fiktiven Hashtabelle ohne Kollisionen** wären die Operationen insert, remove und find so implementierbar:

```
void insert(ElementType e){
    hashTable[h(key(e))] = e;
}

void remove(Key k){
    hashTable[h(k)] = null;
}

Item<ElementType,Key> find(Key k){
    return hashTable[h(k)];
}
```

In einer Hashtabelle **mit Chaining** müssen die **Operationen angepasst** werden (siehe umseitig). **Ergänzen Sie** die fehlenden Anweisungen an zwei gekennzeichneten Stellen entsprechend!

```
List<Item>[] = new List<Item>[m];

insert(ElementType e){
    hashTable[h(key(e))].insert(e);
}

remove(Key k){
    [          ];
}

Item find(Key k){
    return [          ];
}
```

Sortierte Sequenzen – Binäre Suche

- Idee: verwende sortiertes Array und binäre Suche zur Realisierung von locate → locate (ebenso wie find) in O(log n) realisierbar (insert und remove brauchen aber stets $\Theta(n)$ (wegen nötigem Umkopieren))

- Binäre Suche auf aufsteigend sortiertem Array: rekursives Prinzip:
  - Fange in der Mitte an zu suchen.
  - Wenn gesuchter Key größer: Wende binäre Suche auf rechte Teilliste an
  - Wenn gesuchter Key kleiner: Wende binäre Suche auf linke Teilliste an

Binäre Suche – Java Code von locate

```
public static int locate(int key, int[] a) { //a must be sorted ascendingly
        int lowerIndex = 0;
        int higherIndex = a.length - 1;
        if(key > a[higherIndex])
            return -1; //key is larger than the largest key in a --> treat this case extra
        int middleIndex = 0;
        while(lowerIndex <= higherIndex) { //while we are not finished with searching
            middleIndex = lowerIndex + (higherIndex - lowerIndex) / 2; //choose new mid
            if(key < a[middleIndex])
                higherIndex = middleIndex - 1; //continue search in lower half
            else if(key > a[middleIndex])
                lowerIndex = middleIndex + 1; //continue search in upper half
            else
                return middleIndex; //we found it!
        }
        //reaching this part of the code means that a does not contain key
        //--> return index of smallest key k' with k' > k :
        if(key <= a[middleIndex])
            return middleIndex;
        else
            return middleIndex + 1;
    }
```


Binäre Suche – Java Code von locate


```
public static int locate(int key, int[] a) { //a must be sorted ascendingly
        int lowerIndex = 0;
        int higherIndex = a.length - 1;
        if(key > a[higherIndex])
            return -1; //key is larger than the largest key in a --> treat this case extra
        int middleIndex = 0;
        while(lowerIndex <= higherIndex) { //while we are not finished with searching
            middleIndex = lowerIndex + (higherIndex - lowerIndex) / 2; //choose new mid
            if(key < a[middleIndex])
                higherIndex = middleIndex - 1; //continue search in lower half
            else if(key > a[middleIndex])
                lowerIndex = middleIndex + 1; //continue search in upper half
            else
                return middleIndex; //we found it!
        }
        //reaching this part of the code means that a does not contain key
        //--> return index of smallest key k' with k' > k :
        if(key <= a[middleIndex])
            return middleIndex;
        else
            return middleIndex + 1;
    }
```


Binäre Suche – Java Code von locate


```
public static int locate(int key, int[] a) { //a must be sorted ascendingly
        int lowerIndex = 0;
        int higherIndex = a.length - 1;
        if(key > a[higherIndex])
            return -1; //key is larger than the largest key in a --> treat this case extra
        int middleIndex = 0;
        while(lowerIndex <= higherIndex) { //while we are not finished with searching
            middleIndex = lowerIndex + (higherIndex - lowerIndex) / 2; //choose new mid
            if(key < a[middleIndex])
                higherIndex = middleIndex - 1; //continue search in lower half
            else if(key > a[middleIndex])
                lowerIndex = middleIndex + 1; //continue search in upper half
            else
                return middleIndex; //we found it!
        }
        //reaching this part of the code means that a does not contain key
        //--> return index of smallest key k' with k' > k :
        if(key <= a[middleIndex])
            return middleIndex;
        else
            return middleIndex + 1;
    }
```

Binäre Suche – Java Code von locate

```java
public static int locate(int key, int[] a) { //a must be sorted ascendingly
        int lowerIndex = 0;
        int higherIndex = a.length - 1;
        if(key > a[higherIndex])
            return -1; //key is larger than the largest key in a --> treat this case extra
        int middleIndex = 0;
        while(lowerIndex <= higherIndex) { //while we are not finished with searching
            middleIndex = lowerIndex + (higherIndex - lowerIndex) / 2; //choose new mid
            if(key < a[middleIndex])
                higherIndex = middleIndex - 1; //continue search in lower half
            else if(key > a[middleIndex])
                lowerIndex = middleIndex + 1; //continue search in upper half
            else
                return middleIndex; //we found it!
        }
        //reaching this part of the code means that a does not contain key
        //--> return index of smallest key k' with k' > k :
        if(key <= a[middleIndex])
            return middleIndex;
        else
            return middleIndex + 1;
    }
```


Binäre Suche – Java Code von locate

```java
public static int locate(int key, int[] a) { //a must be sorted ascendingly
        int lowerIndex = 0;
        int higherIndex = a.length - 1;
        if(key > a[higherIndex])
            return -1; //key is larger than the largest key in a --> treat this case extra
        int middleIndex = 0;
        while(lowerIndex <= higherIndex) { //while we are not finished with searching
            middleIndex = lowerIndex + (higherIndex - lowerIndex) / 2; //choose new mid
            if(key < a[middleIndex])
                higherIndex = middleIndex - 1; //continue search in lower half
            else if(key > a[middleIndex])
                lowerIndex = middleIndex + 1; //continue search in upper half
            else
                return middleIndex; //we found it!
        }
        //reaching this part of the code means that a does not contain key
        //--> return index of smallest key k' with k' > k :
        if(key <= a[middleIndex])
            return middleIndex;
        else
            return middleIndex + 1;
    }
```


Binäre Suche – Java Code von locate

```java
public static int locate(int key, int[] a) { //a must be sorted ascendingly
        int lowerIndex = 0;
        int higherIndex = a.length - 1;
        if(key > a[higherIndex])
            return -1; //key is larger than the largest key in a --> treat this case extra
        int middleIndex = 0;
        while(lowerIndex <= higherIndex) { //while we are not finished with searching
            middleIndex = lowerIndex + (higherIndex - lowerIndex) / 2; //choose new mid
            if(key < a[middleIndex])
                higherIndex = middleIndex - 1; //continue search in lower half
            else if(key > a[middleIndex])
                lowerIndex = middleIndex + 1; //continue search in upper half
            else
                return middleIndex; //we found it!
        }
        //reaching this part of the code means that a does not contain key
        //--> return index of smallest key k' with k' > k :
        if(key <= a[middleIndex])
            return middleIndex;
        else
            return middleIndex + 1;
    }
```


Binäre Suche – Java Code von locate

```java
public static int locate(int key, int[] a) { //a must be sorted ascendingly
        int lowerIndex = 0;
        int higherIndex = a.length - 1;
        if(key > a[higherIndex])
            return -1; //key is larger than the largest key in a --> treat this case extra
        int middleIndex = 0;
        while(lowerIndex <= higherIndex) { //while we are not finished with searching
            middleIndex = lowerIndex + (higherIndex - lowerIndex) / 2; //choose new mid
            if(key < a[middleIndex])
                higherIndex = middleIndex - 1; //continue search in lower half
            else if(key > a[middleIndex])
                lowerIndex = middleIndex + 1; //continue search in upper half
            else
                return middleIndex; //we found it!
        }
        //reaching this part of the code means that a does not contain key
        //--> return index of smallest key k' with k' > k :
        if(key <= a[middleIndex])
            return middleIndex;
        else
            return middleIndex + 1;
    }
```

Binäre Suche – Java Code von locate

```java
public static int locate(int key, int[] a) { //a must be sorted ascendingly
        int lowerIndex = 0;
        int higherIndex = a.length - 1;
        if(key > a[higherIndex])
            return -1; //key is larger than the largest key in a --> treat this case extra
        int middleIndex = 0;
        while(lowerIndex <= higherIndex) { //while we are not finished with searching
            middleIndex = lowerIndex + (higherIndex - lowerIndex) / 2; //choose new mid
            if(key < a[middleIndex])
                higherIndex = middleIndex - 1; //continue search in lower half
            else if(key > a[middleIndex])
                lowerIndex = middleIndex + 1; //continue search in upper half
            else
                return middleIndex; //we found it!
        }
        //reaching this part of the code means that a does not contain key
        //--> return index of smallest key k' with k' > k :
        if(key <= a[middleIndex])
            return middleIndex;
        else
            return middleIndex + 1;
    }
```


Binäre Suche

lowerIndex middleIndex higherIndex

locate key 147

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	28	32	40	80	120	128	140	141	144	147	152	159	163	173	189	193	198

⇩

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	28	32	40	80	120	128	140	141	144	147	152	159	163	173	189	193	198

⇩

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	28	32	40	80	120	128	140	141	144	147	152	159	163	173	189	193	198

⇩

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	28	32	40	80	120	128	140	141	144	147	152	159	163	173	189	193	198

Binäre Suche

lowerIndex middleIndex higherIndex

locate key 147

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	28	32	40	80	120	128	140	141	144	147	152	159	163	173	189	193	198

⇩

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	28	32	40	80	120	128	140	141	144	147	152	159	163	173	189	193	198

⇩

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	28	32	40	80	120	128	140	141	144	147	152	159	163	173	189	193	198

⇩

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	28	32	40	80	120	128	140	141	144	147	152	159	163	173	189	193	198

Binäre Suche

lowerIndex middleIndex higherIndex

locate key 147

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	28	32	40	80	120	128	140	141	144	147	152	159	163	173	189	193	198

⇩

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	28	32	40	80	120	128	140	141	144	147	152	159	163	173	189	193	198

⇩

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	28	32	40	80	120	128	140	141	144	147	152	159	163	173	189	193	198

⇩

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	4	28	32	40	80	120	128	140	141	144	147	152	159	163	173	189	193	198

Binäre Suche – Java Code von locate

```
public static int locate(int key, int[] a) { //a must be sorted ascendingly
        int lowerIndex = 0;
        int higherIndex = a.length - 1;
        if(key > a[higherIndex])
            return -1; //key is larger than the largest key in a --> treat this case extra
        int middleIndex = 0;
        while(lowerIndex <= higherIndex) { //while we are not finished with searching
            middleIndex = lowerIndex + (higherIndex - lowerIndex) / 2; //choose new mid
            if(key < a[middleIndex])
                higherIndex = middleIndex - 1; //continue search in lower half
            else if(key > a[middleIndex])
                lowerIndex = middleIndex + 1; //continue search in upper half
            else
                return middleIndex; //we found it!
        }
        //reaching this part of the code means that a does not contain key
        //--> return index of smallest key k' with k' > k :
        if(key <= a[middleIndex])
            return middleIndex;
        else
            return middleIndex + 1;
    }
```


Binäre Suche

- locate: Analyse Laufzeit:
  entscheidende Zeilen:

```
…
while(lowerIndex <= higherIndex) { //while we are not finished with searching
            middleIndex = lowerIndex + (higherIndex - lowerIndex) / 2; //choose new mid
…
```

Wie oft kann man eine Liste der Länge n halbieren? Antwort: $\lfloor \log n \rfloor$ mal
→ Laufzeit ist O(log n)

- Weiterhin Nachteil von sortiertem Array: insert und remove brauchen O(n) Zeit (wegen nötigem Umkopieren)

  Alternative zu Array: verkette Liste: insert und remove brauchen O(1) Zeit, dafür braucht der Index Operator [.] O(n) Zeit → log(n) kaputt ☹

- → Idee: Verkette sortierte Liste mit zusätzlicher Suchstruktur

Binäre Suche

- locate: Analyse Laufzeit:
  entscheidende Zeilen:

```
…
while(lowerIndex <= higherIndex) { //while we are not finished with searching
            middleIndex = lowerIndex + (higherIndex - lowerIndex) / 2; //choose new mid
…
```

Wie oft kann man eine Liste der Länge n halbieren? Antwort: $\lfloor \log n \rfloor$ mal
→ Laufzeit ist O(log n)

- Weiterhin Nachteil von sortiertem Array: insert und remove brauchen O(n) Zeit (wegen nötigem Umkopieren)

  Alternative zu Array: verkette Liste: insert und remove brauchen O(1) Zeit, dafür braucht der Index Operator [.] O(n) Zeit → log(n) kaputt ☹

- → Idee: Verkette sortierte Liste mit zusätzlicher Suchstruktur

145

Binäre Suche

- locate: Analyse Laufzeit:
  entscheidende Zeilen:

```
…
while(lowerIndex <= higherIndex) { //while we are not finished with searching
            middleIndex = lowerIndex + (higherIndex - lowerIndex) / 2; //choose new mid
…
```

Wie oft kann man eine Liste der Länge n halbieren? Antwort: $\lfloor \log n \rfloor$ mal
→ Laufzeit ist O(log n)

- Weiterhin Nachteil von sortiertem Array: insert und remove brauchen O(n) Zeit (wegen nötigem Umkopieren)

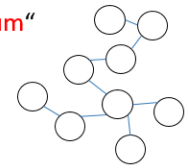
  Alternative zu Array: verkette Liste: insert und remove brauchen O(1) Zeit, dafür braucht der Index Operator [.] O(n) Zeit → log(n) kaputt ☹

- → Idee: Verkette sortierte Liste mit zusätzlicher Suchstruktur

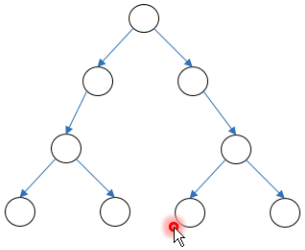
145

Bäume

- Ungerichteter, zusammenhängender, kreisfreier Graph: „Baum“
  (äquivalent: Es ex. genau ein Pfad zwischen je zwei Knoten)

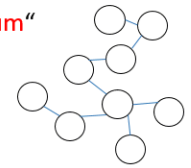


- „Gerichteter Baum“: Gerichteter, zusammenhängender, Graph, der
  - ein Baum ist, wenn Richtungen der Kanten ignoriert werden
  - Es existiert ein eindeutiger Wurzelknoten, so dass jeder Knoten im Baum auf eindeutigem Pfad vom Wurzel-Knoten erreichbar ist

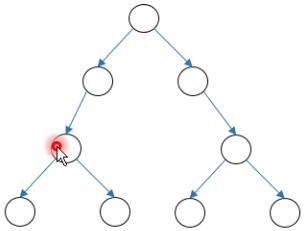


Bäume

- Ungerichteter, zusammenhängender, kreisfreier Graph: „Baum“
  (äquivalent: Es ex. genau ein Pfad zwischen je zwei Knoten)



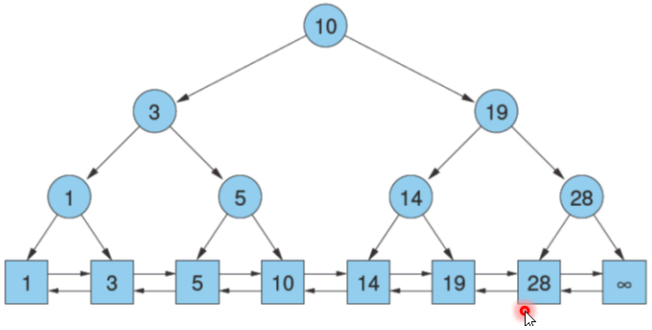
- „Gerichteter Baum“: Gerichteter, zusammenhängender, Graph, der
  - ein Baum ist, wenn Richtungen der Kanten ignoriert werden
  - Es existiert ein eindeutiger Wurzelknoten, so dass jeder Knoten im Baum auf eindeutigem Pfad vom Wurzel-Knoten erreichbar ist



- Gerichtete Bäume: Begriffe:
  - Wurzel: Knoten mit Eingangsgrad 0
  - Blatt: Knoten mit Ausgangsgrad 0
  - Innerer Knoten: Eingangsgrad = 1 & Ausgangsgrad ≥ 1

Binäre Suchbäume

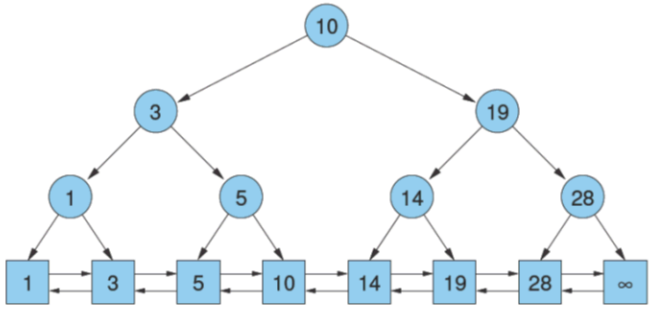
- Als *Suchstruktur*:
  Binärer Suchbaum: Gerichteter Baum für den gilt:
  - jeder Knoten hat höchstens 2 Kinder („Grad des Baumes“ ≤ 2)
  - Für jeden Knoten n_k mit Key k gilt:
    Alle Schlüssel im linken Teilbaum unter n_k sind kleiner als k,
    alle Schlüssel im rechten Teilbaum unter n_k sind größer als k



[4]

Binäre Suchbäume

- Als *Suchstruktur*:
  Binärer Suchbaum: Gerichteter Baum für den gilt:
  - jeder Knoten hat höchstens 2 Kinder („Grad des Baumes“ ≤ 2)
  - Für jeden Knoten n_k mit Key k gilt:
    Alle Schlüssel im linken Teilbaum unter n_k sind kleiner als k,
    alle Schlüssel im rechten Teilbaum unter n_k sind größer als k

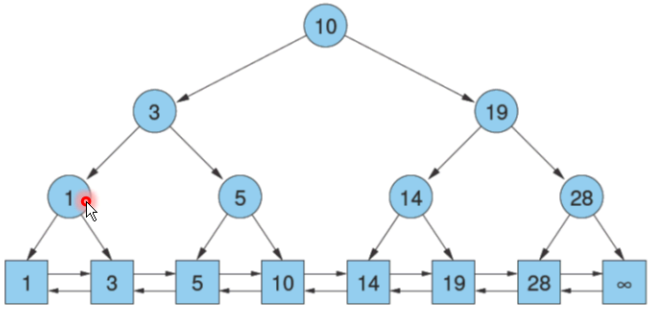


[4]

150

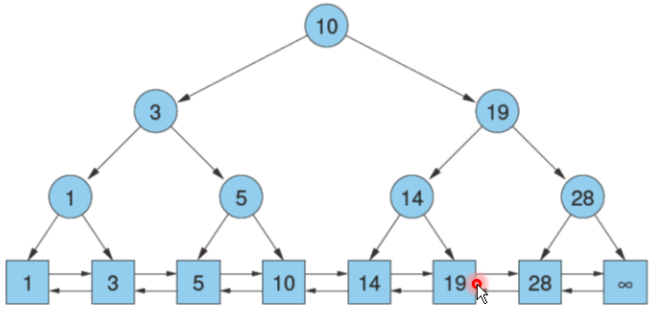
Binäre Suchbäume

- Als *Suchstruktur*:
  Binärer Suchbaum: Gerichteter Baum für den gilt:
  - jeder Knoten hat höchstens 2 Kinder („Grad des Baumes“ ≤ 2)
  - Für jeden Knoten n_k mit Key k gilt:
    Alle Schlüssel im linken Teilbaum unter n_k sind kleiner als k,
    alle Schlüssel im rechten Teilbaum unter n_k sind größer als k



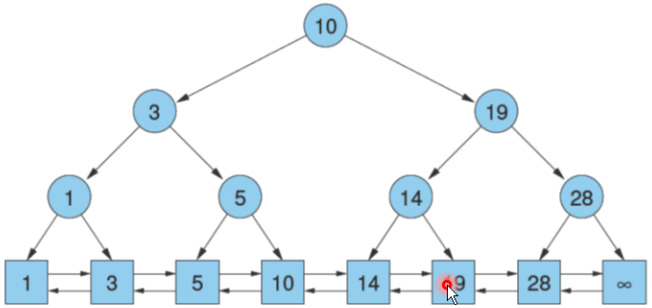
Binäre Suchbäume

- Als *Suchstruktur*:
  Binärer Suchbaum: Gerichteter Baum für den gilt:
  - jeder Knoten hat höchstens 2 Kinder („Grad des Baumes“ ≤ 2)
  - Für jeden Knoten n_k mit Key k gilt:
    Alle Schlüssel im linken Teilbaum unter n_k sind kleiner als k,
    alle Schlüssel im rechten Teilbaum unter n_k sind größer als k



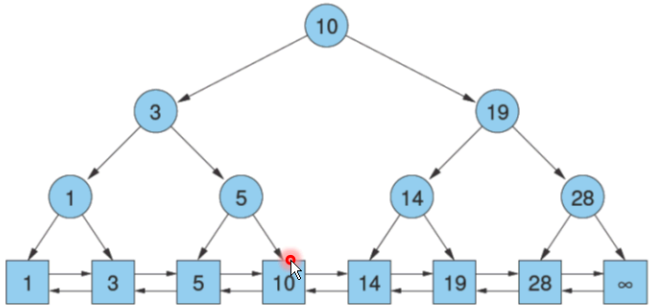
Binäre Suchbäume

- Als *Suchstruktur*:
  Binärer Suchbaum: Gerichteter Baum für den gilt:
  - jeder Knoten hat höchstens 2 Kinder („Grad des Baumes“ ≤ 2)
  - Für jeden Knoten n_k mit Key k gilt:
    Alle Schlüssel im linken Teilbaum unter n_k sind kleiner als k,
    alle Schlüssel im rechten Teilbaum unter n_k sind größer als k



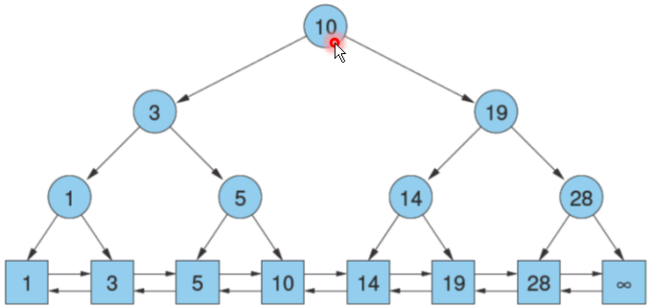
Binäre Suchbäume

- Als *Suchstruktur*:
  Binärer Suchbaum: Gerichteter Baum für den gilt:
  - jeder Knoten hat höchstens 2 Kinder („Grad des Baumes“ ≤ 2)
  - Für jeden Knoten n_k mit Key k gilt:
    Alle Schlüssel im linken Teilbaum unter n_k sind kleiner als k,
    alle Schlüssel im rechten Teilbaum unter n_k sind größer als k



Binäre Suchbäume

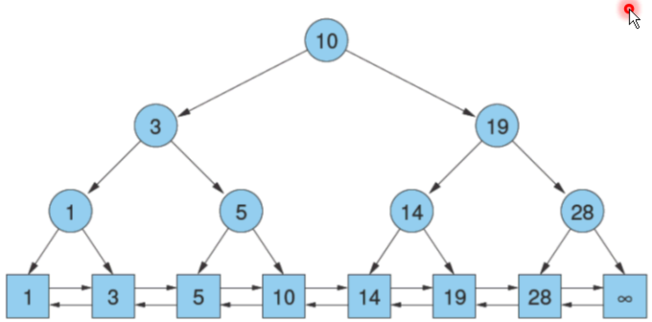
- Als *Suchstruktur*:
  Binärer Suchbaum: Gerichteter Baum für den gilt:
  - jeder Knoten hat höchstens 2 Kinder („Grad des Baumes“ ≤ 2)
  - Für jeden Knoten n_k mit Key k gilt:
    Alle Schlüssel im linken Teilbaum unter n_k sind kleiner als k,
    alle Schlüssel im rechten Teilbaum unter n_k sind größer als k



[4]

Binäre Suchbäume

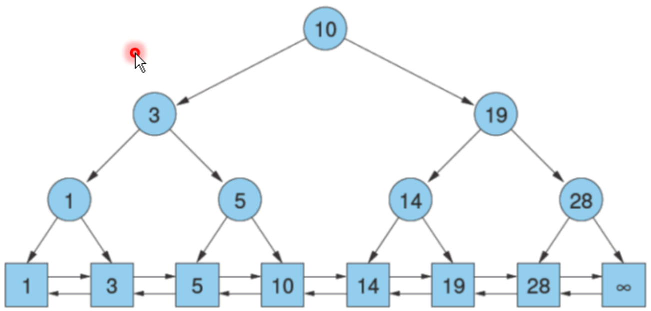
- Als *Suchstruktur*:
  Binärer Suchbaum: Gerichteter Baum für den gilt:
  - jeder Knoten hat höchstens 2 Kinder („Grad des Baumes“ ≤ 2)
  - Für jeden Knoten n_k mit Key k gilt:
    Alle Schlüssel im linken Teilbaum unter n_k sind kleiner als k,
    alle Schlüssel im rechten Teilbaum unter n_k sind größer als k



[4]

Binäre Suchbäume

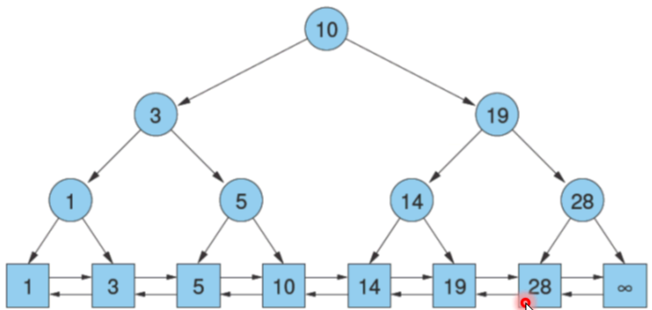
- Als *Suchstruktur*:
  Binärer Suchbaum: Gerichteter Baum für den gilt:
  - jeder Knoten hat höchstens 2 Kinder („Grad des Baumes“ ≤ 2)
  - Für jeden Knoten n_k mit Key k gilt:
    Alle Schlüssel im linken Teilbaum unter n_k sind kleiner als k,
    alle Schlüssel im rechten Teilbaum unter n_k sind größer als k



[4]

Binäre Suchbäume

- Als *Suchstruktur*:
  Binärer Suchbaum: Gerichteter Baum für den gilt:
  - jeder Knoten hat höchstens 2 Kinder („Grad des Baumes“ ≤ 2)
  - Für jeden Knoten n_k mit Key k gilt:
    Alle Schlüssel im linken Teilbaum unter n_k sind kleiner als k,
    alle Schlüssel im rechten Teilbaum unter n_k sind größer als k



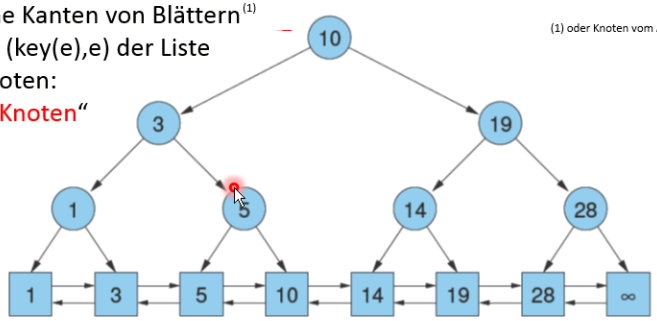
[4]

Binäre Suchbäume

- Als *Suchstruktur*:
  Binärer Suchbaum: Gerichteter Baum für den gilt:
  - jeder Knoten hat höchstens 2 Kinder („Grad des Baumes“ ≤ 2)
  - Für jeden Knoten n_k mit Key k gilt:
    Alle Schlüssel im linken Teilbaum unter n_k sind kleiner als k,
    alle Schlüssel im rechten Teilbaum unter n_k sind größer als k

- -- Knoten des Suchbaums sind mit Keys der Liste markiert.
  -- Jeder Key in der Liste kommt genau einmal im Baum vor.
  -- Zusätzliche Kanten von Blättern[(1)]
  zu Items (key(e),e) der Liste
  -- Innere Knoten:
  „Splitter Knoten“

(1) oder Knoten vom Ausgangsgrad 1

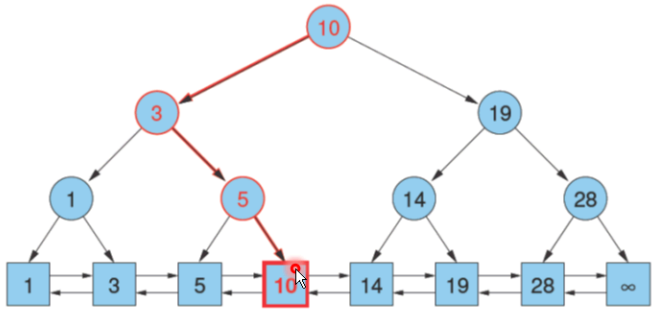


[4]

Binäre Suchbäume: locate

- locate(Key k) Operation:
  - starte in Wurzel
  - Für jeden erreichten Knoten: k ≤ Key k‘, gehe zum linken Kind-Knoten, ansonsten zum rechten.

Bsp: locate(9)

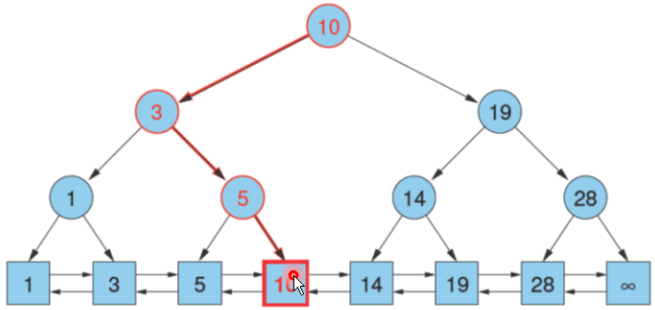


[4]

Binäre Suchbäume: locate

- locate(Key k) Operation:
  - starte in Wurzel
  - Für jeden erreichten Knoten: k ≤ Key k‘, gehe zum linken Kind-Knoten, ansonsten zum rechten.

Bsp: locate(9)



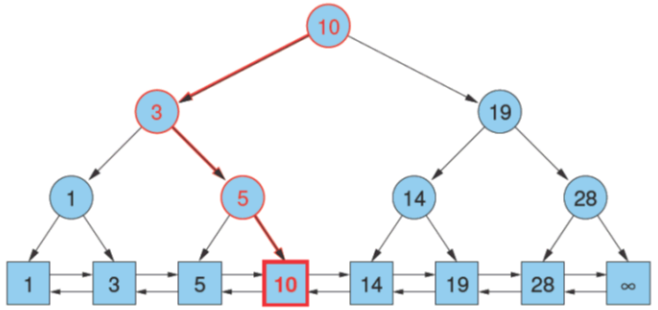
[4]

Binäre Suchbäume: locate

- locate(Key k) Operation:
  - starte in Wurzel
  - Für jeden erreichten Knoten: k ≤ Key k‘, gehe zum linken Kind-Knoten, ansonsten zum rechten.

Bsp: locate(9)

- Wenn Baum balanciert (d.h. hat Höhe $\lceil \log n \rceil$) → locate: O(log n)

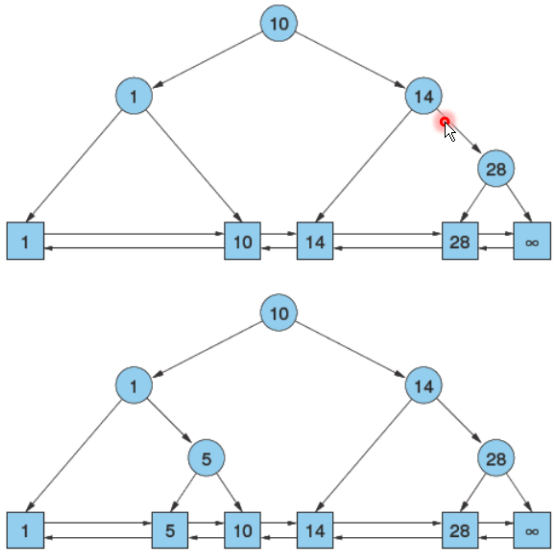


[4]

Binäre Suchbäume: insert

insert(ElementType e):
- zuerst wie locate(key(e)) bis Element e' in Liste erreicht
- falls key(e') > key(e): (d.h. Element ist noch nicht in Liste und Baum)
  - füge e vor e' in Liste ein
  - füge ein neues Suchbaumblatt mit key(e) ein

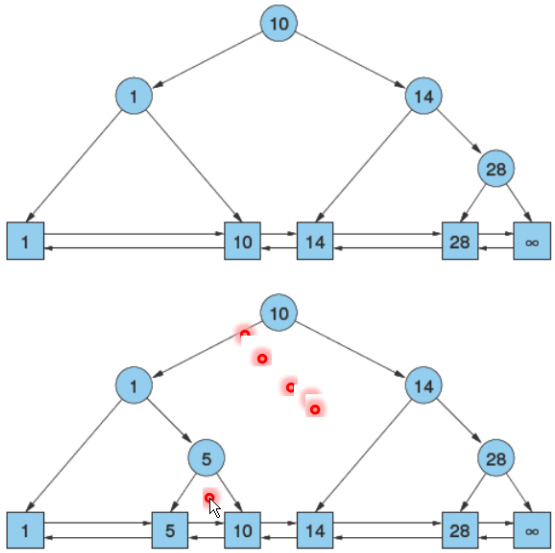
insert(5)



Binäre Suchbäume: insert

insert(ElementType e):
- zuerst wie locate(key(e)) bis Element e' in Liste erreicht
- falls key(e') > key(e): (d.h. Element ist noch nicht in Liste und Baum)
  - füge e vor e' in Liste ein
  - füge ein neues Suchbaumblatt mit key(e) ein

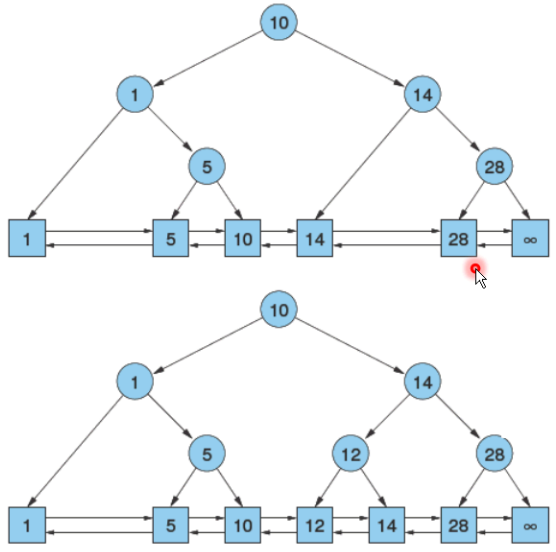
insert(5)



Binäre Suchbäume: insert

insert(ElementType e):
- zuerst wie locate(key(e)) bis Element e' in Liste erreicht
- falls key(e') > key(e): (d.h. Element ist noch nicht in Liste und Baum)
  - füge e vor e' in Liste ein
  - füge ein neues Suchbaumblatt mit key(e) ein

insert(12)



Binäre Suchbäume: remove

remove(Key k):
- zuerst wie locate(k) bis Element e in Liste erreicht
- falls key(e) = k: (d.h. Element ist überhaupt in Liste und Baum)
  - lösche e aus Liste
  - lösche Baum-Vater v von e aus dem Suchbaum
  - setze im Baumknoten w mit key(w) = k (sofern dieser nicht im Schritt zuvor gelöscht wurde) den neuen Wert key(w) = key(v)

Binäre Suchbäume: remove

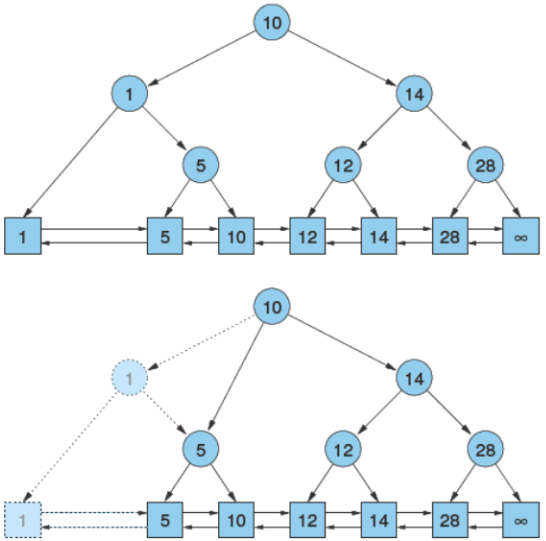
remove(Key k):

- zuerst wie locate(k) bis Element e in Liste erreicht
- falls key(e) = k: (d.h. Element ist überhaupt in Liste und Baum)
  - lösche e aus Liste
  - lösche Baum-Vater v von e aus dem Suchbaum
  - setze im Baumknoten w mit key(w) = k (sofern dieser nicht im Schritt zuvor gelöscht wurde) den neuen Wert key(w) = key(v)

Binäre Suchbäume: rem

remove(Key k):
- zuerst wie locate(k) bis Element e in Liste erreicht
- falls key(e) = k: (d.h. Element ist überhaupt in Liste und Baum)
  - lösche e aus Liste
  - lösche Baum-Vater v von e aus Suchbaum
  - setze im Baumknoten w mit key(w) = k (sofern dieser nicht im Schritt zuvor gelöscht wurde) den neuen Wert key(w) = key(v)

remove(1)

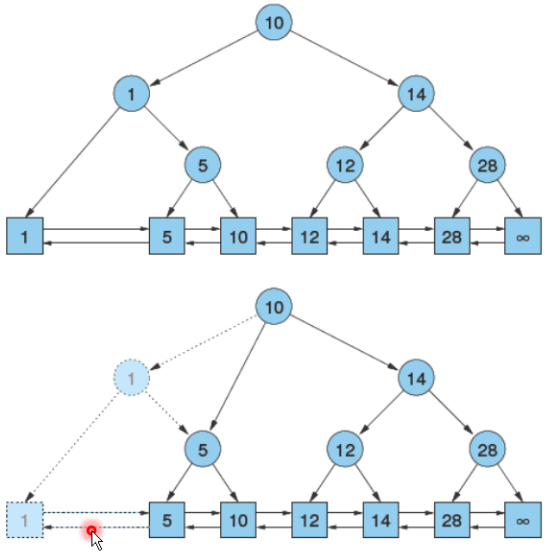


[4]

Binäre Suchbäume: rem

remove(Key k):
- zuerst wie locate(k) bis Element e in Liste erreicht
- falls key(e) = k: (d.h. Element ist überhaupt in Liste und Baum)
  - lösche e aus Liste
  - lösche Baum-Vater v von e aus Suchbaum
  - setze im Baumknoten w mit key(w) = k (sofern dieser nicht im Schritt zuvor gelöscht wurde) den neuen Wert key(w) = key(v)

remove(1)

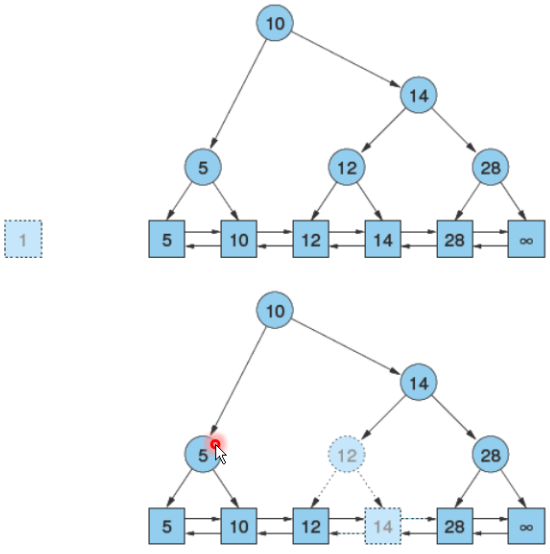


[4]

Binäre Suchbäume: rem

remove(Key k):
- zuerst wie locate(k) bis Element e in Liste erreicht
- falls key(e) = k: (d.h. Element ist überhaupt in Liste und Baum)
  - lösche e aus Liste
  - lösche Baum-Vater v von e aus Suchbaum
  - setze im Baumknoten w mit key(w) = k (sofern dieser nicht im Schritt zuvor gelöscht wurde) den neuen Wert key(w) = key(v)

remove(14)

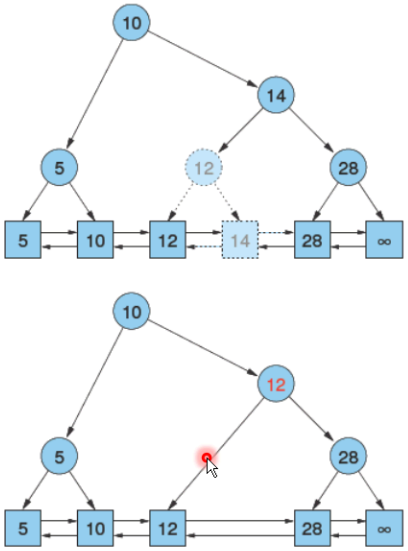


[4]

Binäre Suchbäume: rem

remove(Key k):
- zuerst wie locate(k) bis Element e in Liste erreicht
- falls key(e) = k: (d.h. Element ist überhaupt in Liste und Baum)
  - lösche e aus Liste
  - lösche Baum-Vater v von e aus Suchbaum
  - setze im Baumknoten w mit key(w) = k (sofern dieser nicht im Schritt zuvor gelöscht wurde) den neuen Wert key(w) = key(v)

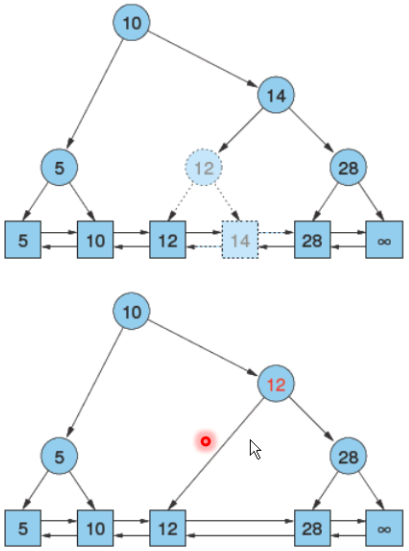
remove(14)



Binäre Suchbäume: rem

remove(Key k):
- zuerst wie locate(k) bis Element e in Liste erreicht
- falls key(e) = k: (d.h. Element ist überhaupt in Liste und Baum)
  - lösche e aus Liste
  - lösche Baum-Vater v von e aus Suchbaum
  - setze im Baumknoten w mit key(w) = k (sofern dieser nicht im Schritt zuvor gelöscht wurde) den neuen Wert key(w) = key(v)

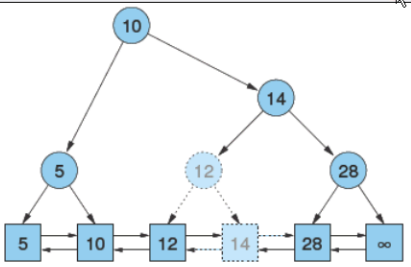
remove(14)

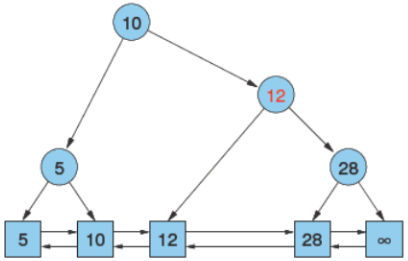


Binäre Suchbäume: rem

remove(Key k):
- zuerst wie locate(k) bis Element e in Liste erreicht
- falls key(e) = k: (d.h. Element ist überhaupt in Liste und Baum)
  - lösche e aus Liste
  - lösche Baum-Vater v von e aus Suchbaum
  - setze im Baumknoten w mit key(w) = k (sofern dieser nicht im Schritt zuvor gelöscht wurde) den neuen Wert key(w) = key(v)

remove(14)





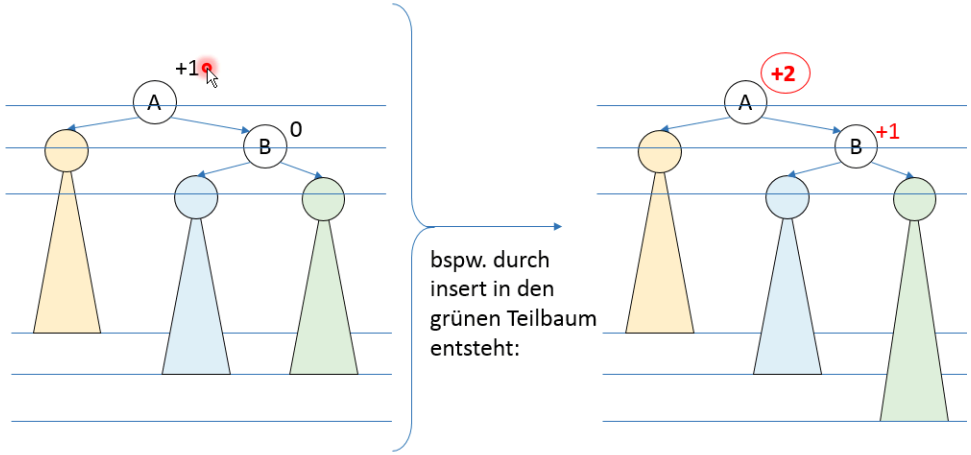
AVL Bäume

- Lösung: Balance des Baumes erhalten (Nach jeder Operation checken, ggf. Rebalancierung)
- viele verschiedene Ansätze: AVL Bäume, (a,b)-Bäume, rot-schwarz-Bäume etc.

- AVL Bäume: in jedem Knoten Höhenunterschiede $\Delta h = h_r - h_l$ des rechten und linken Teilbaums speichern. Ziel: für alle Knoten soll stets $\Delta h \in \{-1,0,1\}$ sein.
- Bei insert und remove: Δh-Werte nach oben bis zur Wurzel anpassen
- Rebalancierungs-Methoden wenn $|\Delta h| \geq 2$:
  - Rotationen nach rechts oder links,
  - Doppelrotationen nach rechts oder links.

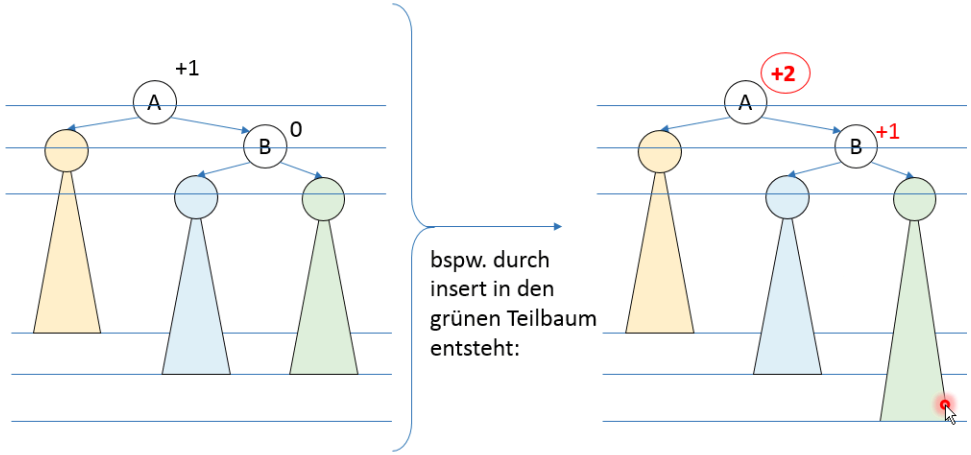
AVL Bäume: Rotationen

- Bsp. für Fall, dass Rotation nach links notwendig wird:



AVL Bäume: Rotationen

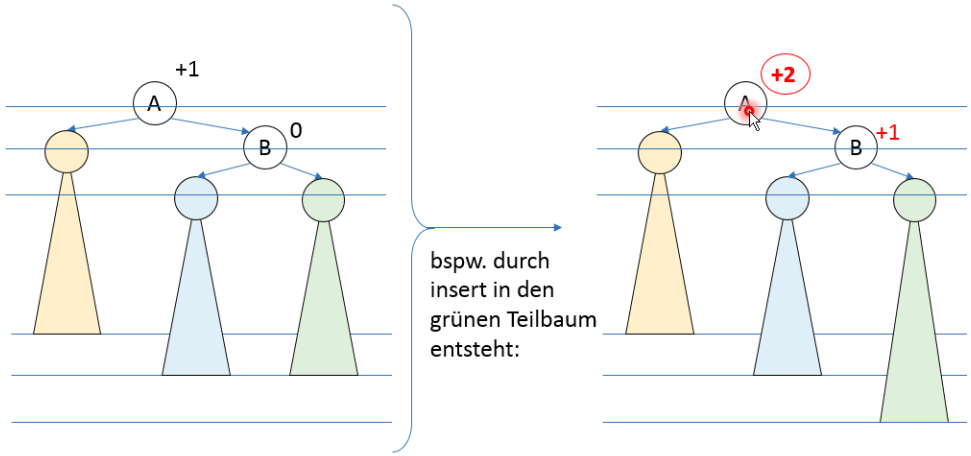
- Bsp. für Fall, dass Rotation nach links notwendig wird:



163

AVL Bäume: Rotationen

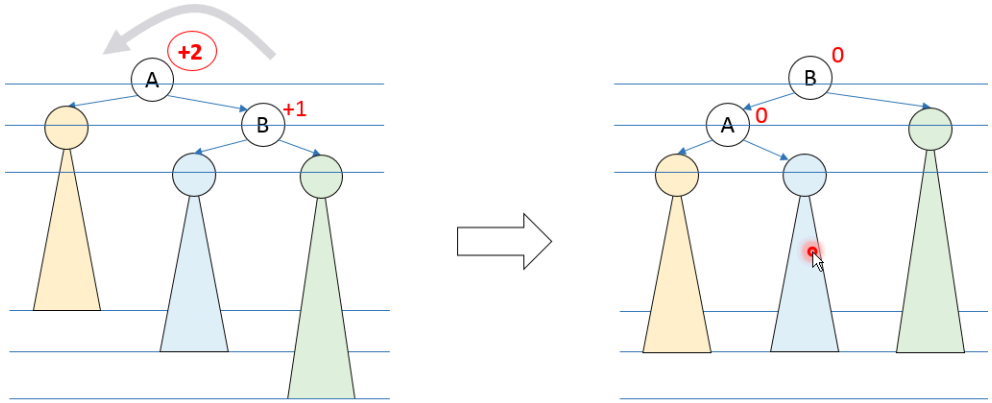
- Bsp. für Fall, dass Rotation nach links notwendig wird:



163

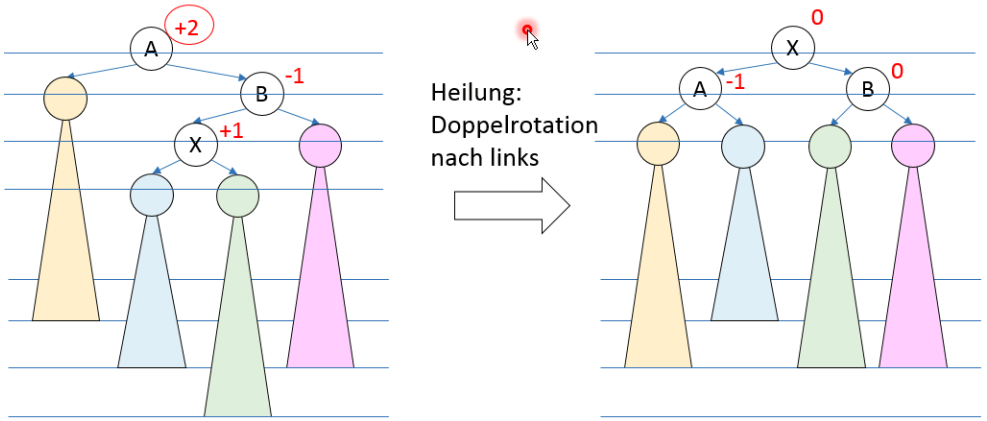
AVL Bäume: Rotationen

- Rotation nach links:



AVL-Bäume: Doppelrotationen

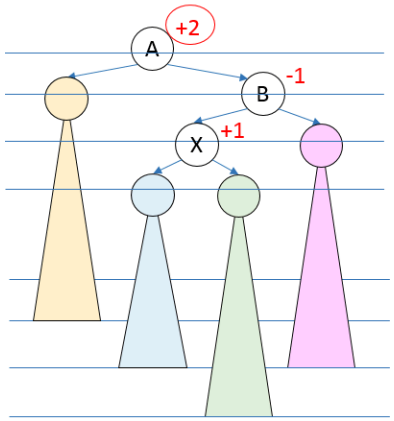
Bspw. durch insert in grünen TB entsteht diese Situation:



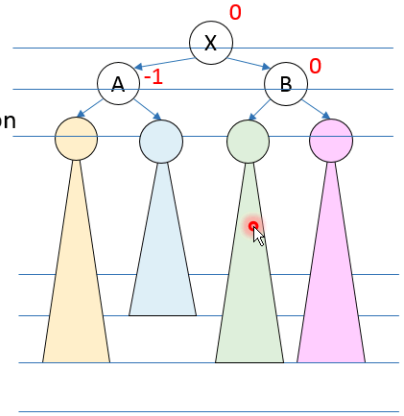
Heilung: Doppelrotation nach links

AVL-Bäume: Doppelrotationen

Bspw. durch insert in grünen TB entsteht diese Situation:



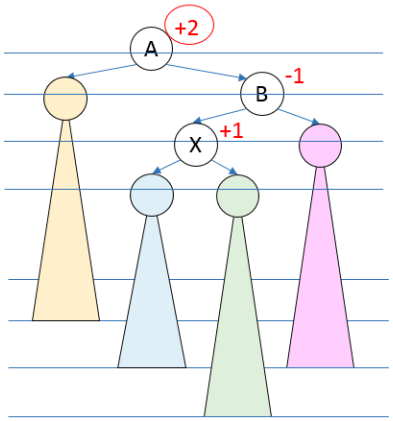
Heilung: Doppelrotation nach links



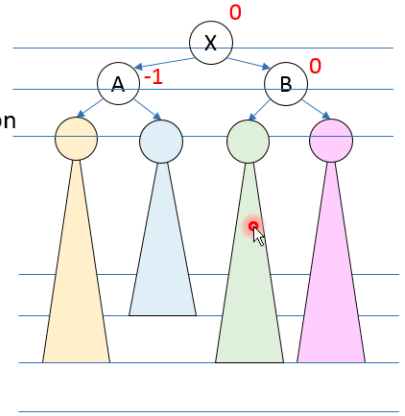
165

AVL-Bäume: Doppelrotationen

Bspw. durch insert in grünen TB entsteht diese Situation:



Heilung: Doppelrotation nach links



165