Inhalt

3.3.7 Heap (Haufen)

- Motivation
  - für manchen Anwendungen nur partielle Ordnung der Elemente statt vollständiger nötig, z.B.
    - Prioritätsschlange: nur das minimale (oder maximale) Element
    - k Maximalwerte aus Menge mit n >> k Elementen (Rangfolgebestimmung von Suchmaschinen!)
  - Die typische Operationen:

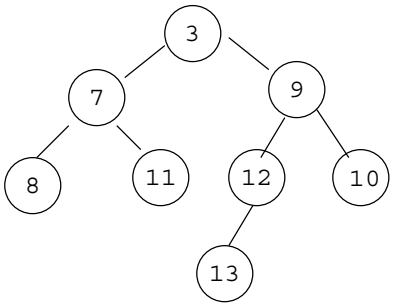
    `enqueue()`

    `findMin()`

    `dequeueMin()`
  - Gesucht Datenstruktur, die alle Operationen auch im schlechtesten Fall in logarithmischer Zeit O(log n) ausführt
- Verschiedene Realisierungen, hier binärer Heap

Heap (2)

- Partiell geordneter Baum
  - ist ein knotenmarkierter Binärbaum mit
  - jeder Knotenwert ist kleiner oder gleich den Werten seiner Nachfolger



Heap (3)

- Modell und Invariante

```
data Ord t => POTree t = E | N(POTree t) t (POTree t)

inv E = True
inv(N l x r) = (if l /= E then x<=root l else True) &&
               (if r /= E then x<=root r else True)

--Operations:
  dequeueMin :: Ord t => POTree t -> (min, POTree t)
  dequeueMin pt
-- requires pt != E
-- effects : minimal value deleted from Argument
--          returns pair of minimal value vmin and
--                              argument tree without vmin
....
```

Heap (funktional)

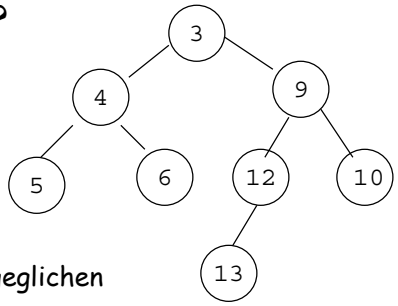
```
> dequeueMin :: Ord t => POTree t -> (t, POTree t)
> dequeueMin ( N l v r) = (v, merge2 l r)

> merge2 E r = r
> merge2 l E = l
> merge2 E E = E
> merge2 (N ll vl rl) (N lr vr rr)
>   | vl <= vr  =  N (merge2 ll rl) vl (N lr vr rr)
>         | otherwise =  N (N ll vl rl) vr (merge2
  lr rr)

> t = N (N (N E 70 E) 50 (N E 80 E)) 15  (N (N E
  90 E) 70 E)
```


Binärer Heap

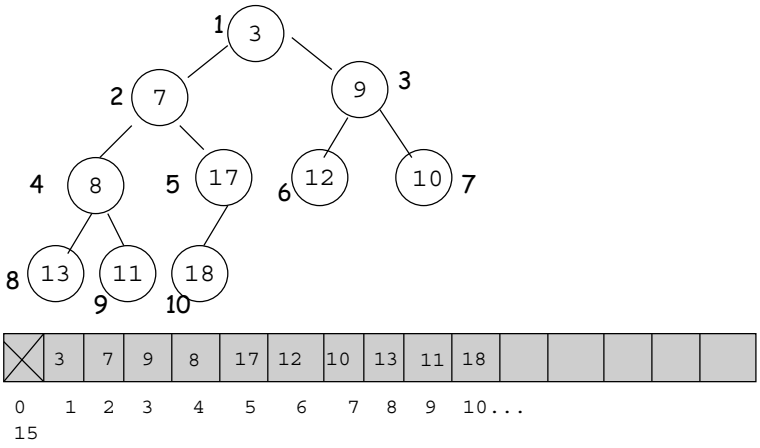
- Logarithmische Laufzeit?
  - Kann Baum entarten?
  - Ja:
    `dequeue 3..9`



  - Ziel:
    Baum möglichst ausgeglichen halten
    z.B. Binominal Heap, Fibonacci Heap
    hier: Binärer Heap – Baum ist vollständig

Binärer Heap

- Vollständiger Baum: Feldrepräsentation



Feldrepräsentation vollständiger binärer Bäume

```
Comparable[] arrayBT;
int count;            //number of nodes
int size;             //array size
int parent(int i){
  // requires 0 < i <= count
  // effects: - , returns index of parent
  return i/2;
}
int leftChild (int i){
  // requires 0 < i <= count
  // effects: -, return index of left child, if
   //             isLeaf(i) return 0
  if (2*i > count) return 0; else return 2*i;
}
int rightChild(int i) { // ...
  if (2*i+1 > count) return 0; else return 2*i+1;}
```

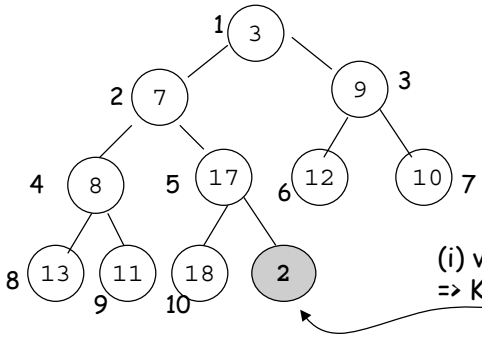
Abstraktionsfunktion und Invariante

```
// abstr::Comparable arrayT->POTree(abstr' (Comparable))
// abstr a    if (count==0) then  E
// abstr a = abstr'' root(a)
// abstr'' a[j] = if (a[j] == null ) then E
// abstr'' a[i] = N ( l  a'[i] r)
//                  a'[i] = abstr'a[i]
//                  l = abstr'' a[2*i]
//                  r = abstr'' a[2*i+1]

// inv:  a[0] == null && for all 1 <= i <= count (a[i] != null)
//                    && for all 1 <= i < count/2
//                        (a[2*i] = null \/ a[i] <= a[2*i]
//                        && a[2*i+1 = null \/ a[i] <= a[2*i+1])
//                       && count <= size-1
```


Binärer Heap

- Einfügen (enqueue)

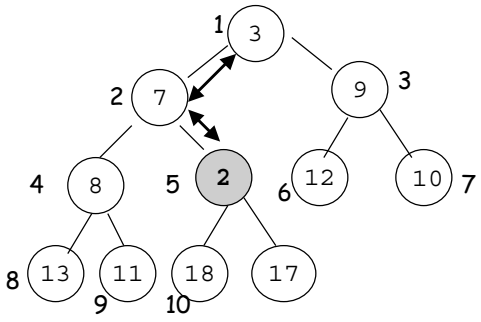


(i) vollständiger Baum
=> Knoten hier einfügen

(ii) Heap-Eigenschaft
wiederherstellen

Binärer Heap

- enqueue()




```
while (parent > val) {exchange();}
```

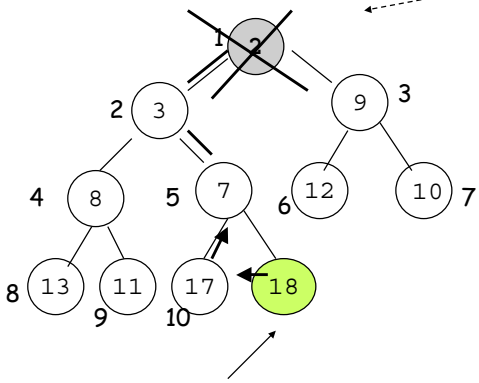
Maximal log n Vertauschungen

Binary heap: enqueue

```
public void enqueue(Comparable val) throws
   java.lang.Exception  {
// requires : -
// effects: inserts val into this as a leaf into the
// only possible position which maintains completeness
// of tree and maintains heap invariant
// if count > size throws an execption
  if (count== arrayBT.length -1) throw new
     java.lang.Exception("heap full");
  count++;
  int i = count;
  while (i > 1  && arrayBT[parent(i)].compareTo(val)>0)
  {  arrayBT[i] = arrayBT[parent(i)];
     i /= 2;
  }
  arrayBT[i] = val;
}
```

Binärer Heap: dequeue

- dequeue



"Loch" wandert nach unten zum kleinern seiner Kinder, es sein denn, dies ist kleiner als das letzte Element arrayBT[count] (hier 18)

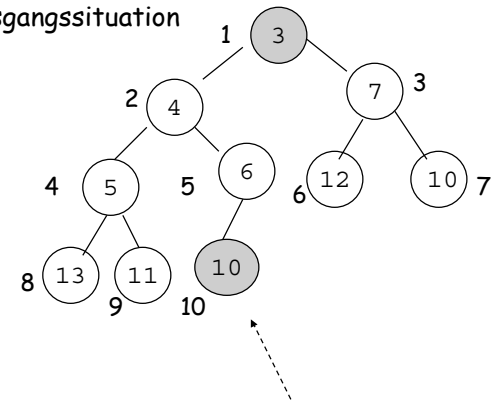
muss hier entfernt (Vollständigkeit)
und passend eingefügt werden (Heap-Eigenschaft)

Binärer Heap: dequeue

- dequeue()

2. Beispiel, Ausgangssituation

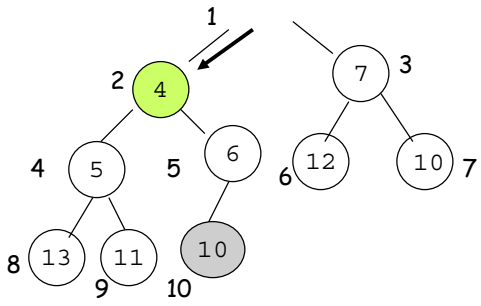
Der Knoten wird entfernt, Rückgabewert von dequeue()



Der Knoten muss "in den Baum " verschoben werden, Vollständig mit Heapeigenschaft!

Binärer Heap: dequeue

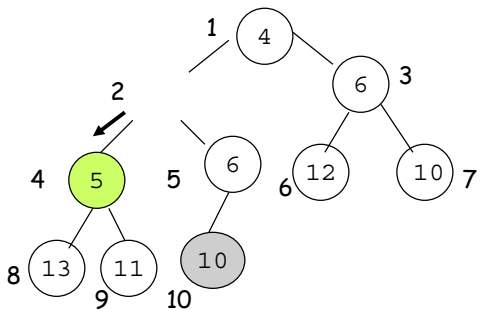
Schritt 2



4 < 7 → linker Teilbaum, 4 < 10: weiter

Binärer Heap: dequeue

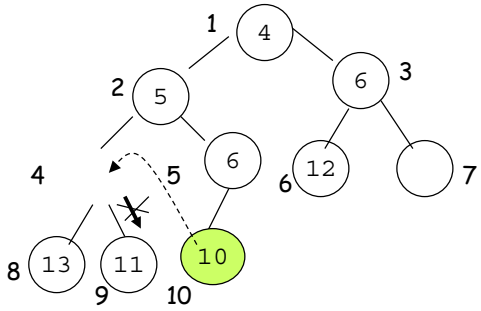
Schritt 3



5 < 6 → linker Teilbaum, 5 < 10: weiter

Binärer Heap: dequeue

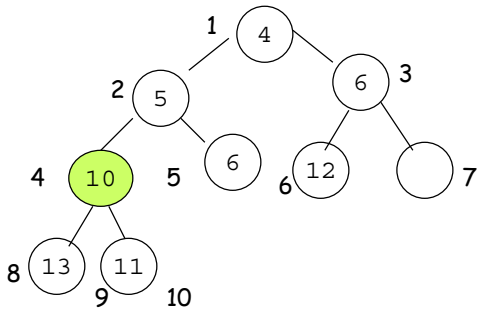
Schritt 4



11 < 13 → linker Teilbaum, aber 10 < 11 : 10 ins Loch, fertig!

Binärer Heap: dequeue

Endzustand nach dequeue()



Binärer Heap: dequeue

```
public Comparable dequeueMin() throws java.lang.Exception
{  // ...
  Comparable min = arrayBT[1];
  Comparable last = arrayBT[count];
  count--;
  int i = 1;
  while (2*i < count+1) {
    int child = 2* i;         // left child

   if ((child+1 < count+1)  //check right child
   && (arrayBT[child+1].compareTo( arrayBT[child]) < 0))
    child = child+1;

  if (last.compareTo(arrayBT[child]) < 0 )
      break;
    arrayBT[i] = arrayBT[child]; i = child;
  }}
```