

6. Prioritätslisten (Priority Queues)

- Definition und Anwendungen
- Binäre Heaps
- Anwendung: HeapSort
- Index-Heaps mit change- und remove-Operation
- Binomiale Heaps mit merge-Operation

Prioritätslisten

- Eine **Prioritätsliste** speichert eine Liste von Elementen mit Prioritäten, für die eine **lineare Ordnung** (z.B. Vergleichsoperation auf Zahlen) definiert ist.
- Folgende Operationen werden effizient unterstützt:
 - **deleteMax()** : löscht das Element mit größter Priorität
 - **insert(x)**: fügt ein neues Element mit Priorität x ein.
 - **build(x[])**: Aufbau einer Prioritätsliste aus einem Feld von Elementen.
- Ebenfalls möglich: **deleteMin** statt **deleteMax** (einfach realisierbar durch geeignete Anpassung der Vergleichsoperation)
- Gleiche Prioritäten dürfen mehrfach vorkommen.
- In der Literatur wird manchmal auch von Schlüsseln statt Prioritäten gesprochen.

Evtl. weitere Operationen

- Elemente mit einer eindeutigen Nummerierung versehen und folgenden Operationen:
 - `change(i, x)`:
ändert beim Element mit Nummer i die Priorität auf x.
 - `remove(i)`:
löscht das Element mit Nummer i
- Verschmelzung von 2 Prioritätslisten:
 - `prioList1.merge(prioList2)`:

Anwendungen

- Grundlegend für **Greedy-Algorithmen** (gierige Algorithmen):
 - Löse ein Problem, indem mit einer einfachen Teillösung begonnen und diese schrittweise erweitert wird.
 - Wähle dabei immer den bestmöglichen Schritt (höchste Priorität) ohne Berücksichtigung zukünftiger Schritte:
„Nimm immer das größte Stück zuerst“
 - zahlreiche Beispiele für Greedy-Algorithmen:
Dijkstra-Algorithmus, Prim-Algorithmus,
Kruskal-Algorithmus, Datenkompression mit dem Huffman-Verfahren, etc.
- **HeapSort**:
Lösche aus einer Folge das jeweils größte Element und speichere es in die sortierte Folge ab.
- **Scheduler** in Betriebssystemen:
Elemente der Prioritätsliste sind Jobs.
Für schnelle Antwortzeiten erhalten kurze Jobs hohe Prioritäten.

Analyse naheliegender Implementierungen (1)

Datenstruktur	deleteMax()	insert(x)	build(x[])	merge(prioList)
verkettete Liste (unsortiert)	$O(n)$	$O(1)$	$O(n)$	$O(1)$
sortierte, verkettete Liste	$O(1)$	$O(n)$	$O(n \log n)$	$O(n)$
Ausgeglichener Suchbaum	$O(\log n)$	$O(\log n)$	$O(n \log n)$	$O(n \log n)$

- Alle Komplexitätsangaben für Prioritätslisten mit n Elementen.
- build bei sortierter Liste mit schnellem Sortierverfahren.
- Verkettete Liste mit Zeiger auf letztem Knoten, so dass bei unsortierter Liste merge in $O(1)$.

Analyse naheliegender Implementierungen (2)

Datenstruktur	change(i, x)	remove(i)
verkettete Liste mit effizienter Suchstruktur für Nummerierung	$O(\log n)$	$O(\log n)$
sortierte, verkettete Liste mit effizienter Suchstruktur für Nummerierung	$O(n)$	$O(\log n)$
Ausgeglichener Suchbaum mit effizienter Suchstruktur für Nummerierung	$O(\log n)$	$O(\log n)$

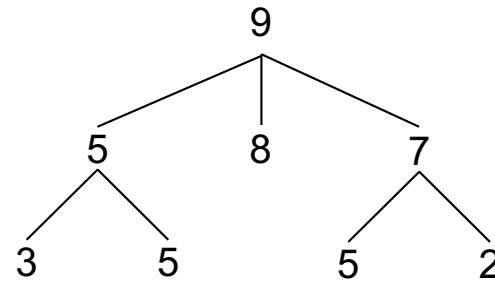
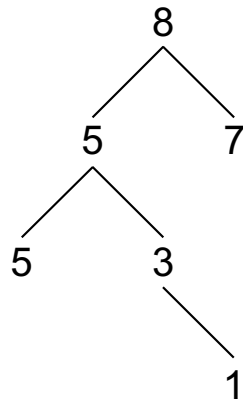
- Nummer i zur eindeutigen Identifizierung.
- Um bei $\text{change}(i, x)$ bzw. $\text{remove}(i)$ das Element mit Nummer i effizient (d.h. $O(\log n)$) finden zu können, wird zusätzlich eine schnelle Suchstruktur für die Nummer als Schlüssel benötigt.
- $\text{change}(i, x)$ bei sortierter Liste erfordert ein teures Umordnen in $O(n)$.

6. Prioritätslisten (Priority Queues)

- Definition und Anwendungen
- Binäre Heaps
- Anwendung: HeapSort
- Index-Heaps mit change- und remove-Operation
- Binomiale Heaps mit merge-Operation

Heaps

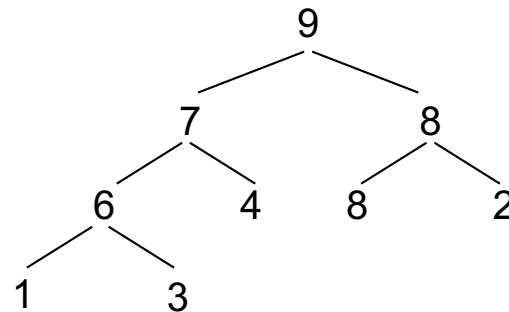
- Ein **Heap** ist ein Baum, für den die sogenannte **Heap-Ordnung** gilt:
Jeder Knoten ist größer als oder gleich seiner Kinder.
- Beispiele:**



- Das größte Element steht grundsätzlich an der Wurzel.
- In der Praxis wird versucht, die Heaps möglichst balanciert zu halten.

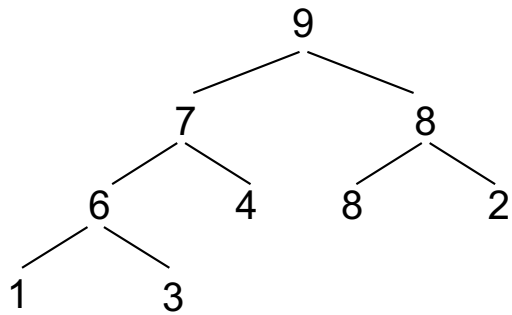
Binäre Heaps

- Ein **binärer Heap** ist ein vollständiger Binärbaum mit Heap-Ordnung.
- In einem vollständigen Binärbaum hat jeder Knoten maximal zwei Kinder und alle Ebenen sind vollständig gefüllt. Die letzte Ebene ist linksbündig gefüllt.
- **Beispiel:**



Binäre Heaps als Felder

- Ein binärer Heap lässt sich einfach als Feld implementieren, in dem die Elementen des Heaps ebenenweise in das Feld abgespeichert werden.

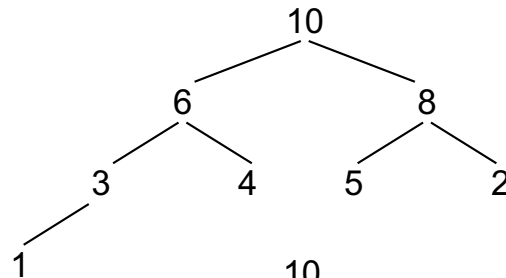


a	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	9	7	8	6	4	8	2	1	3

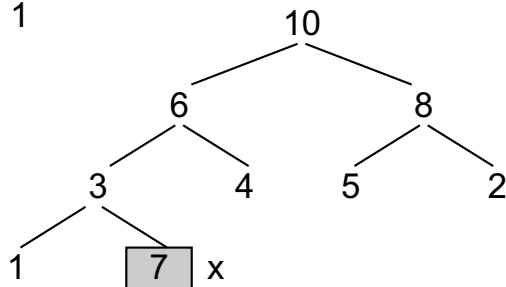
- Die Implementierung als Feld gestattet eine effiziente Baumtraversierung - sowohl von der Wurzel als auch von den Blättern aus:
 - Wurzel ist $a[0]$.
 - Die Kinder von $a[i]$ sind $a[2*i+1]$ und $a[2*i+2]$.
 - Der Elternknoten zu $a[i]$ ist $a[(i-1)/2]$ (ganzzahlige Division)
- Die Höhe eines Heaps mit n Elementen ist $\lfloor \log_2 n \rfloor$.

Operation insert und upheap (1)

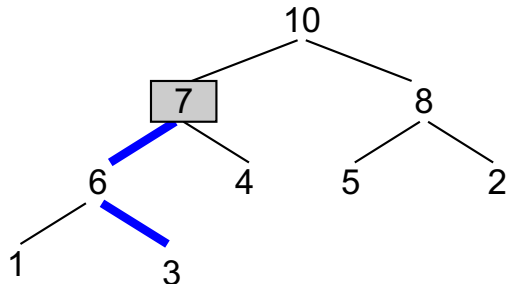
- Ein neues Element wird eingefügt, indem es an das Ende des Heaps abgespeichert wird.
- Im allgemeinen ist dann die Heap-Ordnung verletzt.
- Um die Heap-Ordnung wieder herzustellen, wird das neue Element nach oben verschoben (**Upheap**).



Heap zu Beginn.



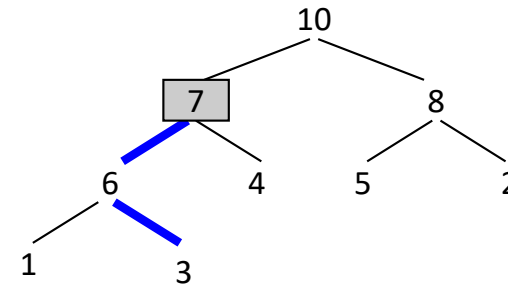
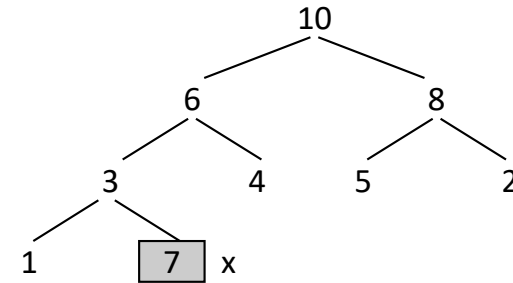
Einfügen von 7 am Heap-Ende.



Upheap auf 7 durchführen, bis Heap-Ordnung wieder erfüllt ist.

Operation insert und upheap (2)

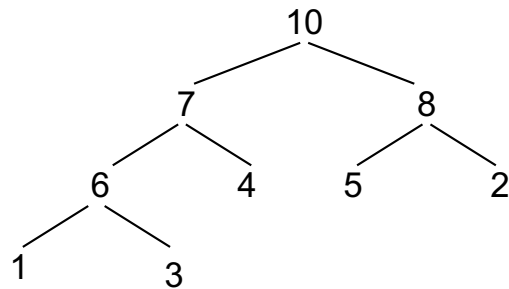
```
private void upheap(int k) {  
    int x = a[k];  
    while (k > 0 && a[(k-1)/2] < x) {  
        a[k] = a[(k-1)/2];  
        k = (k-1)/2;  
    }  
    a[k] = x;  
}  
  
public void insert(int x) {  
    a[n++] = x;  
    upheap(n-1);  
}  
  
private int[] a = new int[N]; // Heap  
private int n = 0; // Anzahl Elemente im Heap
```



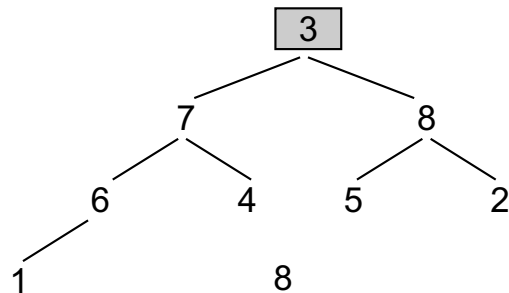
- Laufzeit von upheap und damit von insert:
 $O(\log n)$

Operation delMax und downheap (1)

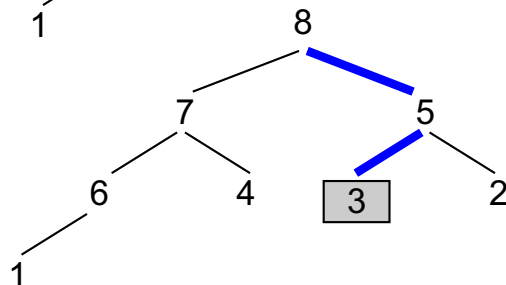
- Das größte Element $a[0]$ kann einfach gelöscht werden, indem es durch das letzte Element des Heaps ersetzt wird:
 $a[0] = a[n-1];$
- Im allgemeinen ist dann die Heap-Ordnung verletzt.
- Um die Heaps-Ordnung wieder herzustellen, wird $a[0]$ nach unten zum jeweils größerem Kind verschoben (**Downheap**).



Heap zu Beginn.



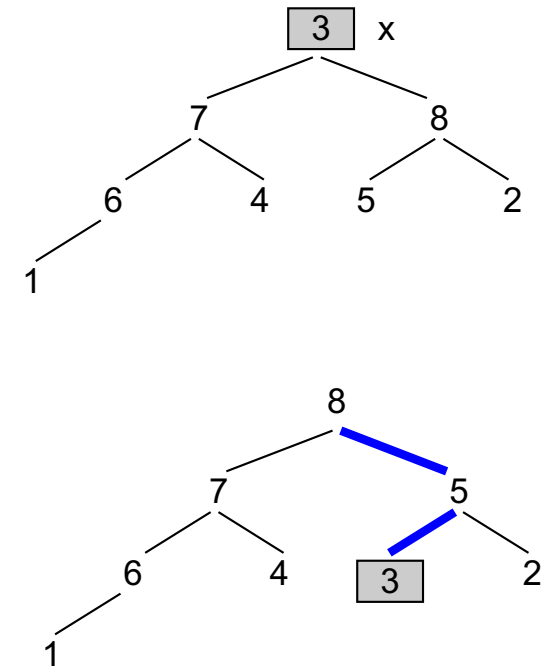
10 wird gelöscht und durch letztes Element 3 ersetzt.



Downheap auf 3 anwenden, bis Heap-Ordnung erfüllt ist.

Operation delMax und downheap (2)

```
private void downheap(int k) {  
    int x = a[k];  
    while (2*k+1 < n) { // a[k] hat Kind  
        int j = 2*k+1; // a[j] ist linkes Kind von a[k]  
        if (j+1 < n) // a[k] hat rechtes Kind  
            if (a[j+1] > a[j]) j++;  
        // a[j] ist jetzt das groesste Kind  
        if (x >= a[j])  
            break; // Schleifenabbruch  
        a[k] = a[j];  
        k = j;  
    }  
    a[k] = x;  
}  
  
public int deleteMax() {  
    int x = a[0];  
    a[0] = a[--n];  
    downheap(0);  
    return x;  
}
```



- Laufzeit von downheap und damit von deleteMax:
 $O(\log n)$

Aufbau eines Heaps: Operation build

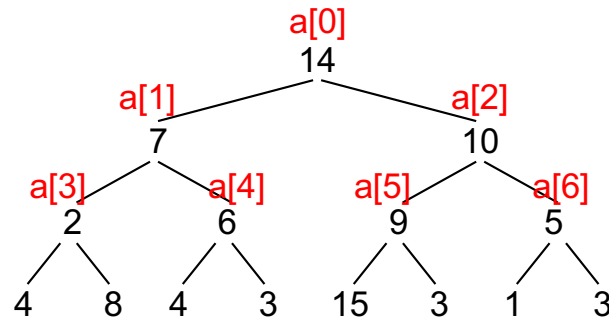
- Wende auf letzten Nicht-Blatt-Knoten (Index $n/2 - 1$; ganzzahlige Division) bis zur Wurzel jeweils eine downheap-Operation an.
Abwärtslaufende Zählschleife!

```
public void build(int[] x) {  
    System.arraycopy(x, 0, a, 0, x.size);  
    for (int i = n/2 - 1; i >= 0; i--)  
        downheap(i);  
}
```

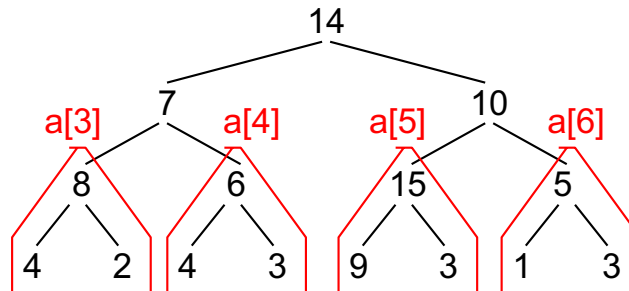
- Begründung, warum der letzte Nicht-Blatt-Knoten den Index $n/2 - 1$ hat:
 - Letzter Nicht-Blatt-Knoten muss Elternknoten des letzten Blattes sein.
 - Letztes Blatt hat Index $n-1$.
 - Elternknoten hat daher den Index $(n-2)/2 = n/2 - 1$ (ganzzahlige Division)

Beispiel zu Aufbau eines Heaps (1)

- Ziel: Umbau von $a = \{14, 7, 10, 2, 6, 9, 5, 4, 8, 4, 3, 15, 3, 1, 3\}$ in ein Heap.

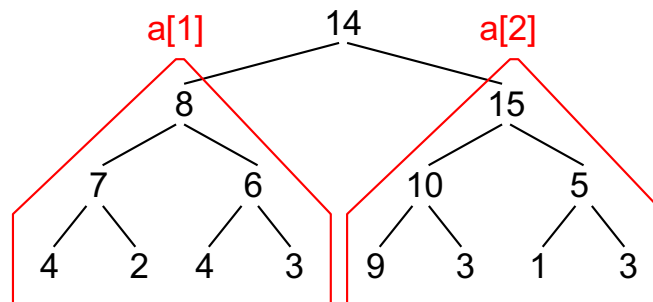


- Downheap angewandt auf Position $i = 6, 5, 4$ und 3 :

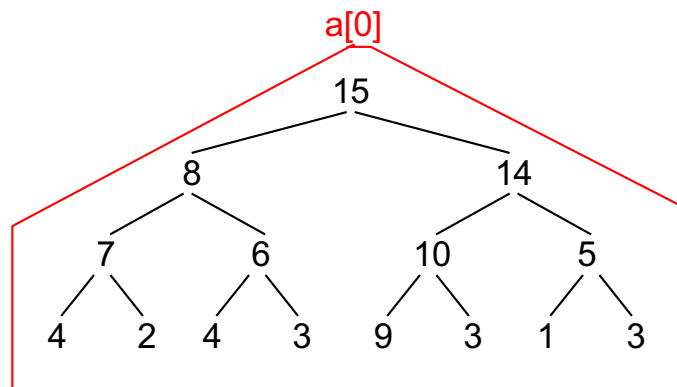


Beispiel zu Aufbau eines Heaps (2)

- Downheap angewandt auf Position $i = 2$ und 3:

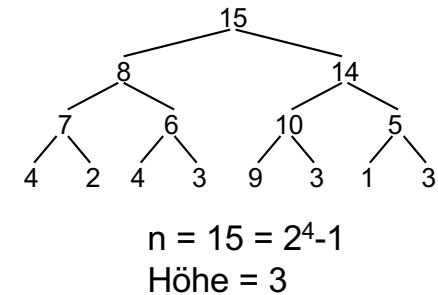


- Downheap angewandt auf Position $i = 0$:



Analyse des Aufbaus eines Heaps

- Wir nehmen einfachheitshalber an, dass $n = 2^k - 1$.
Damit ist der Heap ein vollständiger Binärbaum, der auch in der untersten Schicht vollständig gefüllt ist.
Die Höhe des Heaps ist $k-1$.



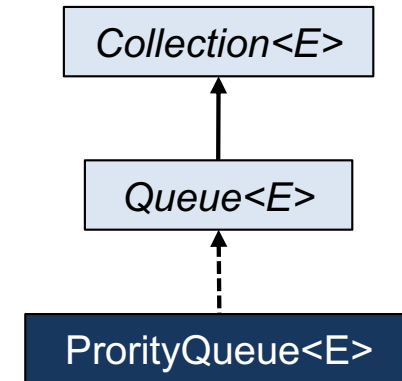
- Folgende Downheaps werden ausgeführt:
 - $(n+1)/4$ mal Heaps der Höhe 1
 - $(n+1)/8$ mal Heaps der Höhe 2
 - $(n+1)/16$ mal Heaps der Höhe 3
 - ...
 - $(n+1)/2^k = 1$ mal auf ein Heap der Höhe $k-1$
- Insgesamt ergibt sich damit folgende Laufzeit (c ist eine Konstante):

$$T(n) = (n+1) * \left(\underbrace{1/4 + 2/8 + 3/16 + \dots + (k-1)/2^k}_{\rightarrow 1} \right) * c$$

- Damit: $T(n) = O(n)$

PriorityQueue aus der Java-Collection

- Klasse `PriorityQueue` aus der Java-API bietet Implementierung einer Prioritätsliste an.
- Vergleichsoperation als Konstruktor-Parameter.
- Operationen in $O(\log n)$:
 - `offer` zum Einfügen eines Elements
 - `poll` zum Löschen des Elements mit höchster (bzw. niedrigster) Priorität
 - `peek` zum Lesen des Elements mit höchster (bzw. niedrigster) Priorität
- `PriorityQueue` ist als `binärer Heap` realisiert.
- Keine effiziente `change`-Methode!
Effiziente Implementierung mit `Index-Heaps` (nächster Abschnitt).

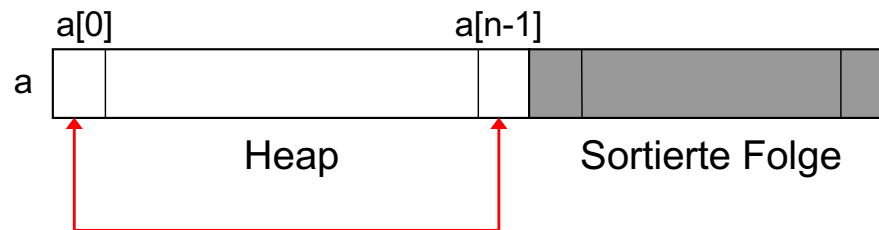


6. Prioritätslisten (Priority Queues)

- Definition und Anwendungen
- Binäre Heaps
- **Anwendung: HeapSort**
- Index-Heaps mit change- und remove-Operation
- Binomiale Heaps mit merge-Operation

Heapsort

Idee:



Implementierung:

```
public static void heapSort(int[] a) {
```

```
    // a in ein Heap umbauen (wie bei Operation build):
```

```
    for (int i = n/2 - 1; i >= 0; i--)
```

```
        downheap(a, n, i);
```

```
    // Heap abbauen und sortierte Folge aufbauen:
```

```
    while (n > 1) {
```

```
        // Im Heap sind noch mehr als 1 Element
```

```
        int t = a[0]; a[0] = a[n-1]; a[n-1] = t;
```

```
        n--;
```

```
        downheap(a, n, 0);
```

```
    }
```

```
}
```

wie `downheaps(i)`,
außer dass Heap `a` und Anzahl
Elemente `n` im Heap als
Parameter übergeben werden.

Laufzeit von HeapSort im Vergleich zu anderen Sortierverfahren

- Laufzeitanalyse von HeapSort: $T(n) = O(n \log n)$
- Laufzeitmessungen (Zeiten in msec):

Sortierverfahren	n = 20000	n= 40000	n = 60000	n = 80000	n = 100000
quickSort	2.12	3.83	5.95	8.10	10.31
quickSort3Median	1.79	3.65	5.65	7.71	9.82
mergeSort	2.71	5.78	8.86	12.13	15.05
heapSort	2.39	5.10	8.04	11.09	14.25

- Messbedingungen:
 - Die CPU-Zeiten sind in msec angegeben und wurden auf einem iMac 2.8 GHz Intel Core 2 Duo und NetBeans 6.8 gemessen.
 - Die Zeitmessungen wurden für 30 zufällig initialisierte int-Felder mit den verschiedenen Sortierverfahren sortiert und anschließend die Zeiten gemittelt.

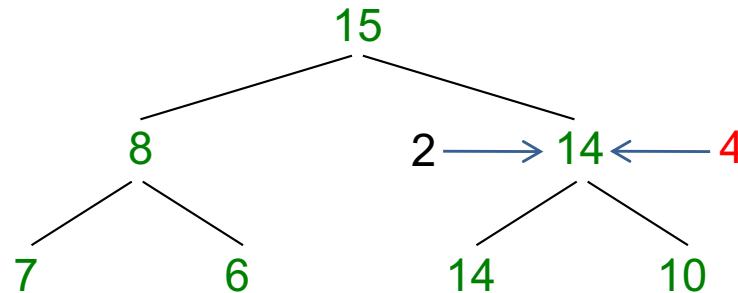
6. Prioritätslisten (Priority Queues)

- Definition und Anwendungen
- Binäre Heaps
- Anwendung: HeapSort
- Index-Heaps mit change- und remove-Operation
- Binomiale Heaps mit merge-Operation

Ziel: weitere Operationen change und remove

- Elemente haben nun einen **Prioritätswert** und eine **Nummer** (Index) 0,1, ..., n-1.
- Bisherige Operationen (leichte Änderung bei insert und build) werden beibehalten:
 - **deleteMax()** : löscht das Element mit größter Priorität
 - **insert(i, x)**: fügt neues Element mit Nummer i und Priorität x ein.
 - **build(i[], x[])**: Aufbau einer Prioritätsliste aus Feld von Elementen.
- Weitere Operationen:
 - **change(i, x)**: Priorität des Elements mit Nummer i auf x setzen.
 - **remove(i)**: Löschen des Elements mit Nummer i.
- Effiziente Lösung mit einem **Index-Heap**.

Einfacher Heap löst Problemstellung nicht effizient



Position p	0	1	2	3	4	5	6
heap[p]	15	8	14	7	6	14	10

Ein einfaches heap-Feld enthält direkt die Prioritätswerte.

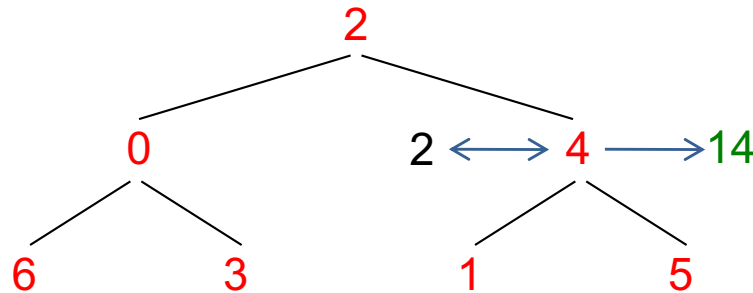
Nummer i	0	1	2	3	4	5	6
prio[i]	8	14	15	6	14	10	7

Das prio-Feld ordnet jeder Nummer den zugehörigen Prioritätswert zu.

- Das Element mit Nummer 4 hat den Prioritätswert 14 und steht im Heap-Feld an der Position 2.
- **Problem:** wie kann beispielsweise bei Element mit Nummer 4 der Prioritätswert effizient von 14 auf 20 erhöht werden. Element muss im Heap-Feld gefunden werden!

Index-Heap löst das Problemstellung effizient

- Das heap-Feld enthält nun die Nummern (Indizes) der Elemente statt den Prioritätswerten.
- Es wird nun zusätzlich ein pos-Feld benötigt, das für jede Nummer die Position des Elements im Heap angibt.



Position p	0	1	2	3	4	5	6
heap[p]	2	0	4	6	3	1	5

heap-Feld enthält Nummer statt Prioritätswert

Nummer i	0	1	2	3	4	5	6
prio[i]	8	14	15	6	14	10	7
pos[i]	1	5	0	4	2	6	3

Das prio-Feld ordnet jeder Nummer den zugehörigen Prioritätswert zu.

Das Pos-Feld gibt für jede Nummer die Position im Heap-Feld an.

- Es gilt: $\text{pos}[\text{heap}[p]] = p$ und $\text{heap}[\text{pos}[i]] = i$ (pos ist die inverse Abbildung zu heap).

Operation insert

- Die Algorithmen für insert, deleteMax und build müssen leicht angepasst werden, wie am Beispiel insert zu sehen ist.
- Die Laufzeit von allen Operationen bleibt bei $O(\log n)$.

```
private void upheap(int k) {  
    int x = heap[k];  
    while (k > 0 && heap[(k-1)/2] < x) {  
        heap[k] = heap[(k-1)/2];  
        k = (k-1)/2;  
    }  
    heap[k] = x;  
}
```

```
public void insert(int x) {  
    heap[n++] = x;  
    upheap(n-1);  
}
```

Fügt Element
mit Priorität x
in den Heap
ein

insert und upheap mit
einfachem Binär-Heap

```
private void upheap(int k) {  
    int i = heap[k]; // i ist hier eine Nummer  
    while (k > 0 && prio[heap[(k-1)/2]] < prio[i]) {  
        heap[k] = heap[(k-1)/2];  
        pos[heap[k]] = k;  
        k = (k-1)/2;  
    }  
    heap[k] = i;  
    pos[i] = k;  
}
```

```
public void insert(int i, int x) {  
    if (pos[i] != -1) return;  
    prio[i] = x;  
    heap[n] = i;  
    pos[i] = n;  
    n++;  
    upheap(n-1);  
}
```

Fügt Element mit
Nummer i und Priorität x
in den Heap ein

Falls i bereits im Heap
vorhanden ist, dann
mache nichts.

insert und upheap mit Index-Heap

Operation change und remove

```
public int change(int i, int x) {  
    if (pos[i] == -1)  
        return;  
    int oldPrio = prio[i];  
    prio[i] = x;  
    upheap(pos[i]);  
    downheap(pos[i]);  
    return oldPrio;  
}
```

Ändert den Vorrangswert des Elements mit Nummer i auf den neuen Wert x.

i muss bereits im Heap vorhanden sein.

Je nach neuer Priorität wird das Element nach unten oder nach oben verschoben

```
public void remove(int i) {  
    if (pos[i] == -1)  
        return;  
    int remPos = pos[i];  
    pos[i] = -1;  
    heap[remPos] = heap[n-1];  
    pos[heap[remPos]] = remPos;  
    n--;  
    upheap(remPos);  
    downheap(remPos);  
}
```

Löscht das Element mit Nummer i aus dem Heap.

i muss bereits im Heap vorhanden sein.

i aus Heap löschen.

Je nach Priorität wird das Element unten oder nach oben verschoben.

Übersicht über Laufzeiten

Datenstruktur	deleteMax()	insert(i, x)	build(i[], x[])	change(i, x)	remove(i)
Binäre Heaps (Java API PriorityQueue)	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
Index-Heaps	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$

- Alle Komplexitätsangaben für Prioritätslisten mit n Elementen.
- Bei binären Heaps werden die Operationen $\text{change}(i, x)$ und $\text{remove}(i)$ nicht direkt unterstützt. Daher muss das Heap-Feld zuerst durchlaufen werden, um das Element x zu lokalisieren.

Implementierung bei beliebigem Schlüsseltyp

- Elemente haben statt Nummerierung 0,1, ... n-1 einen beliebigen aber eindeutigen Schlüssel (key).
- Speichere im Heap-Feld Schlüssel ab.
- Schlüssel, Prioritätswert und Position im Heap werden in einer HashMap abgespeichert.

Position p	0	1	2	3	4	5	6
heap[p]	k2	k0	k4	k6	k3	k1	k5

heap-Feld enthält
Schlüssel (keys)

Schlüssel	k0	k1	k2	k3	k4	k5	k6
Priorität	8	14	15	6	14	10	7
Position im Heap	1	5	0	4	2	6	3

HashMap mit **Schlüssel**
und **Prioritätswert** und
Heap-Position als Nutzdaten
(value)

6. Prioritätslisten (Priority Queues)

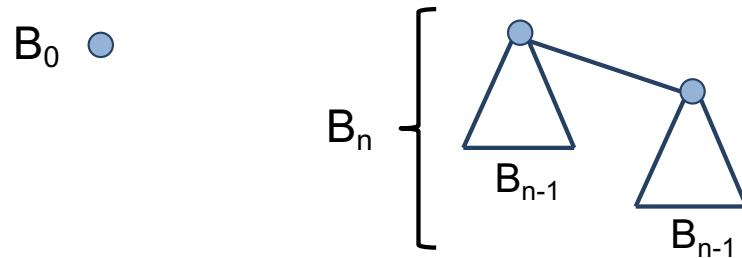
- Definition und Anwendungen
- Binäre Heaps
- Anwendung: HeapSort
- Index-Heaps mit change- und remove-Operation
- Binomiale Heaps mit merge-Operation

Effizientes Verschmelzen von Prioritätslisten

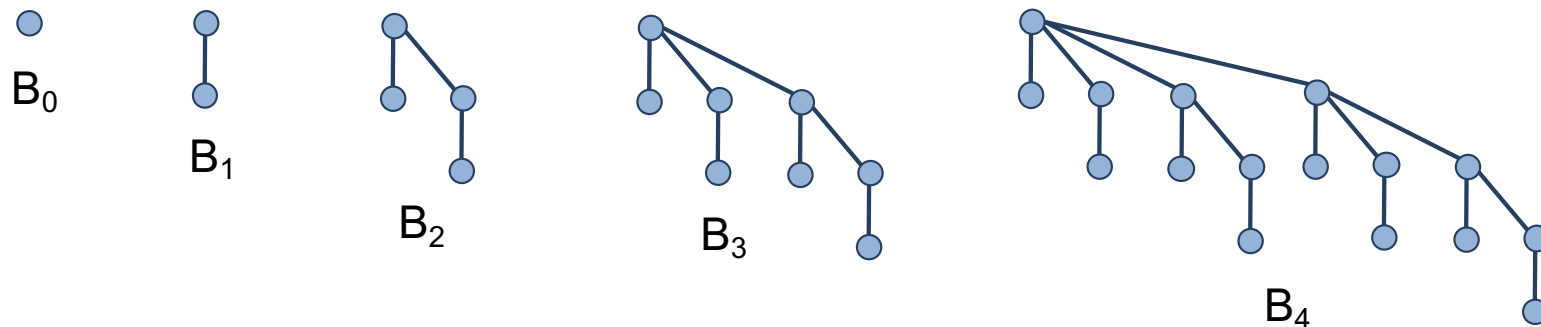
- Verschmelzung von 2 Prioritätslisten:
 - `prioList1.merge(prioList2)`
- Weiterhin:
 - `deleteMax()` : löscht Element mit größter Priorität
 - `insert(x)`: fügt neues Element mit Priorität x ein.
 - `build(x[])`: Aufbau einer Prioritätsliste aus Feld von Elementen.
- Lösung mit einem **Binomialen Heap**.

Binomiale Bäume

- Ein **binomialer Baum** B_n wird rekursiv wie folgt definiert:
 - B_0 besteht aus einem einzigen Knoten
 - ein binomialer Baum B_n wird gebildet, indem an die Wurzel eines binomialen Baums B_{n-1} ein weiterer binomialer Baum B_{n-1} als rechtes Kind gehängt wird.

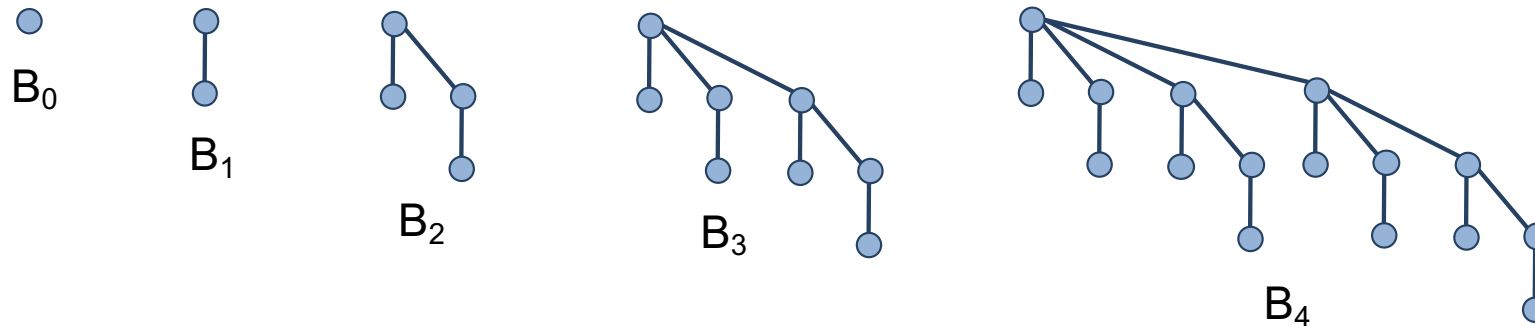


- Die binomialen Bäume B_0 bis B_4 sehen damit wie folgt aus:



Eigenschaften binomialer Bäume (1)

- Die Wurzel von B_n hat die Kinder B_0, B_1, \dots, B_{n-1} .
(Beweis induktiv über n)
- B_n hat die Höhe n .
- B_n besteht aus genau 2^n Knoten.

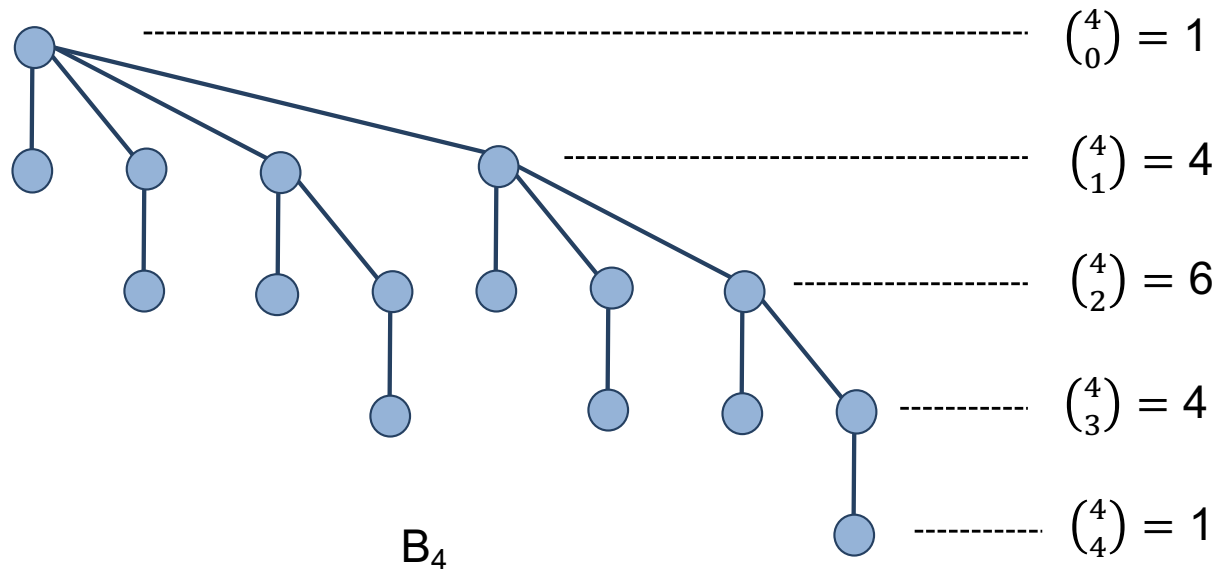


Beispiel:

- B_4 hat B_0, B_1, B_2 und B_3 als Kinder.
- B_4 hat die Höhe 4.
- B_4 besteht aus genau $2^4 = 16$ Knoten.

Eigenschaften binomialer Bäume (2)

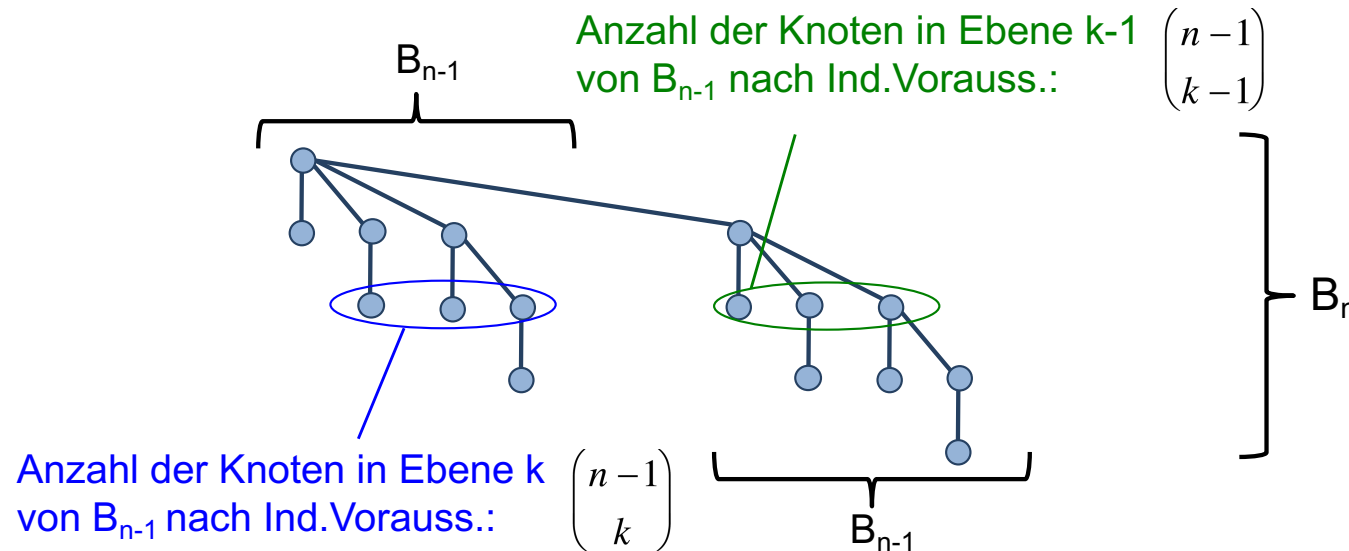
- B_n hat in der Ebene k genau $\binom{n}{k}$ Knoten.
Daher auch der Name **binomiale** Bäume.



Eigenschaften binomialer Bäume (3)

- B_n hat in der Ebene k genau $\binom{n}{k}$ Knoten.

Beweis durch vollständige Induktion über n :



Die Anzahl der Knoten von B_n in der Ebene k ist damit:

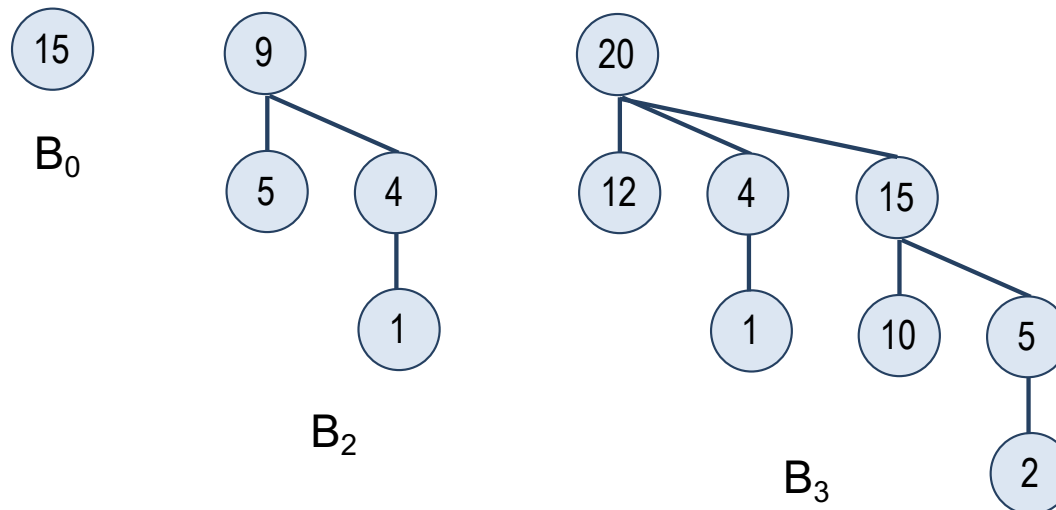
$$\binom{n-1}{k} + \binom{n-1}{k-1} = \binom{n}{k}$$

Binomiale Heaps

- Ein **binomialer Heap** ist eine Folge von binomialen Bäumen (Wald) mit folgenden Eigenschaften:
 - die binomialen Bäume erfüllen die Heap-Ordnung (Eltern \geq Kinder),
 - die binomialen Bäume haben unterschiedliche Größen,
 - die binomialen Bäume sind der Größe nach aufsteigend sortiert.
- Ein binomialer Heap mit n Elementen hat höchstens $\lceil \log_2(n) \rceil$ viele Binomialbäume.

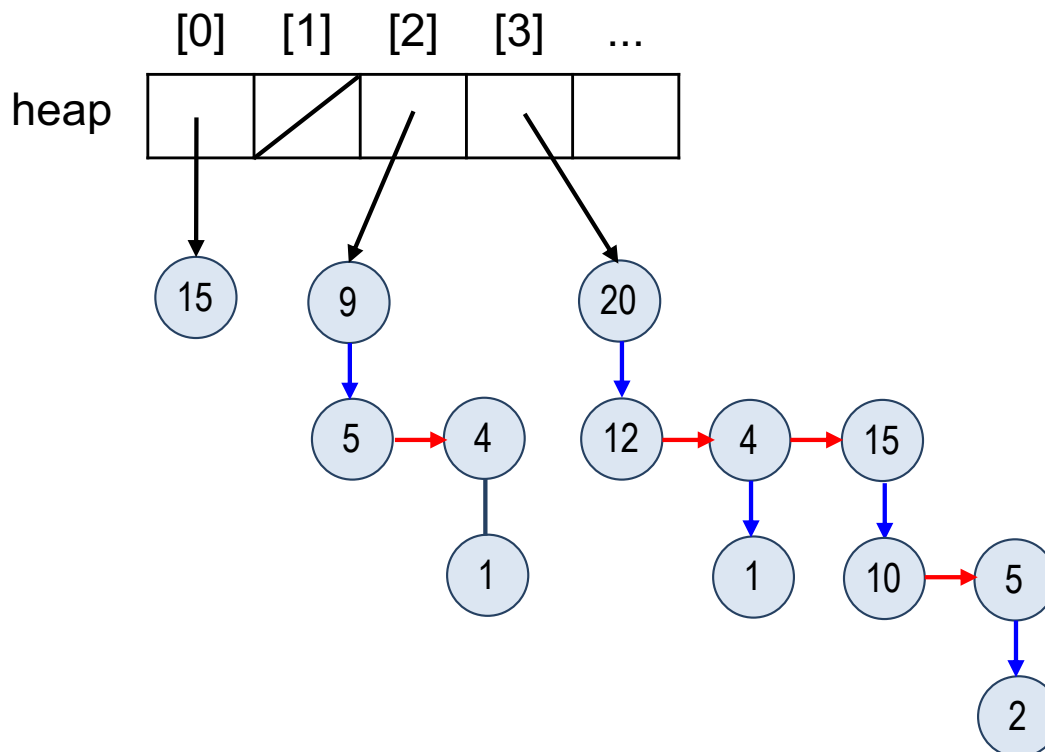
Beispiel:

Ein binomialer Heap mit den drei heap-geordneten Binomialbäumen B_0 , B_2 und B_3 und insgesamt 13 Elementen.



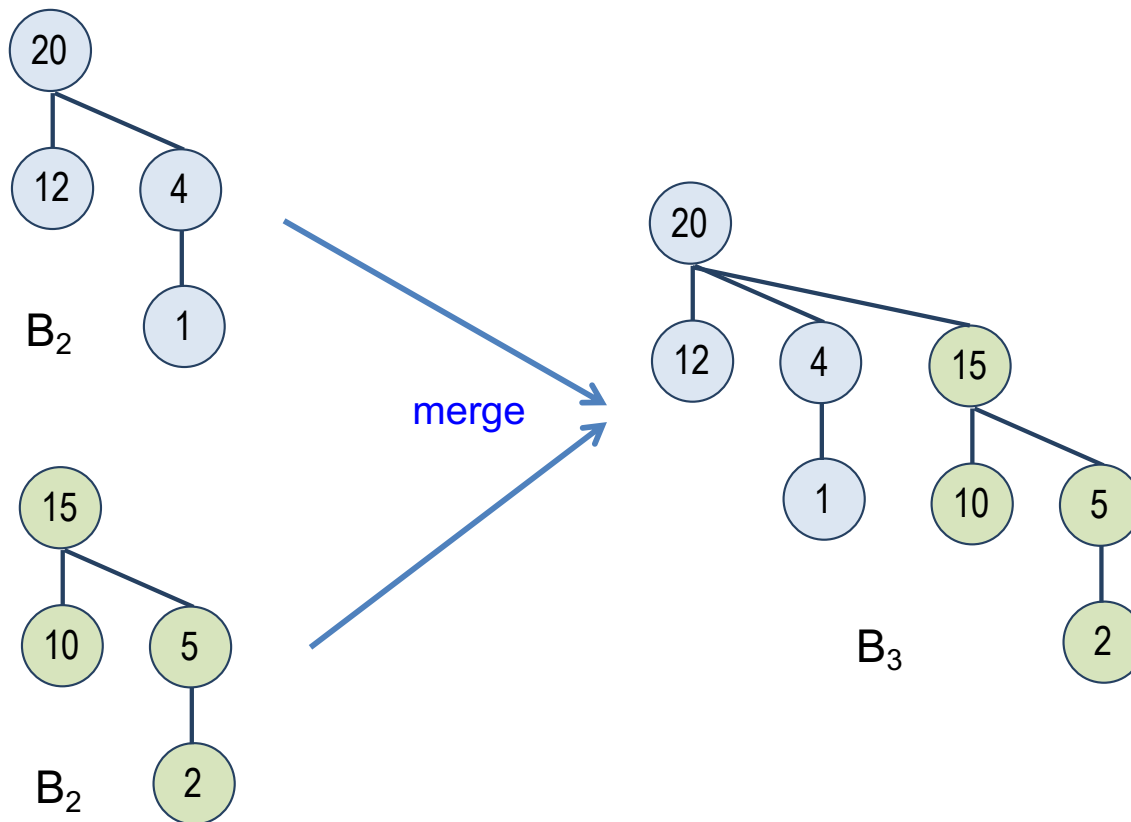
Implementierung von binomialen Heaps

- Die Folge von Binomialbäumen wird in einem Feld heap gehalten, wobei $\text{heap}[k]$ eine Referenz auf den Binomialbaum B_k ist.
- Die einzelnen Binomialbäume werden als verzeigte Struktur realisiert, wobei jeder Knoten einen Verweis auf sein **erstes Kind** und einen Verweis auf seinen **rechten Geschwisterknoten** (sofern vorhanden) hat.
- Der Binomiale Heap von voriger Seite ließe sich damit wie folgt implementieren:



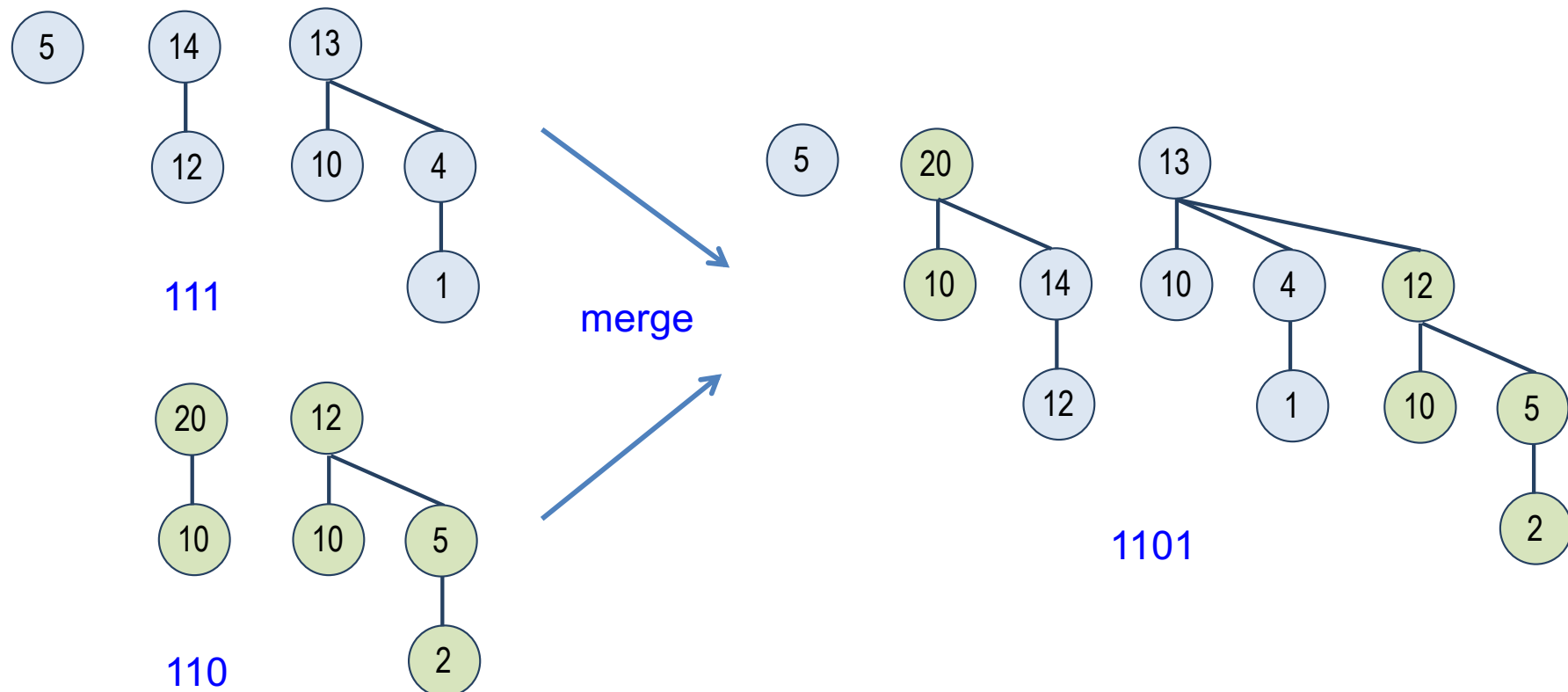
Merge-Operation bei Heap-geordneten Binomialbäumen

- Zwei heap-geordnete Binomialbäume mit der gleichen Größe B_n lassen sich zu einem heap-geordneten Binomialbaum B_{n+1} verschmelzen, indem der Baum mit der kleineren Wurzel als rechtes Kind an die Wurzel des anderen Baums gehängt wird.
- Beispiel:

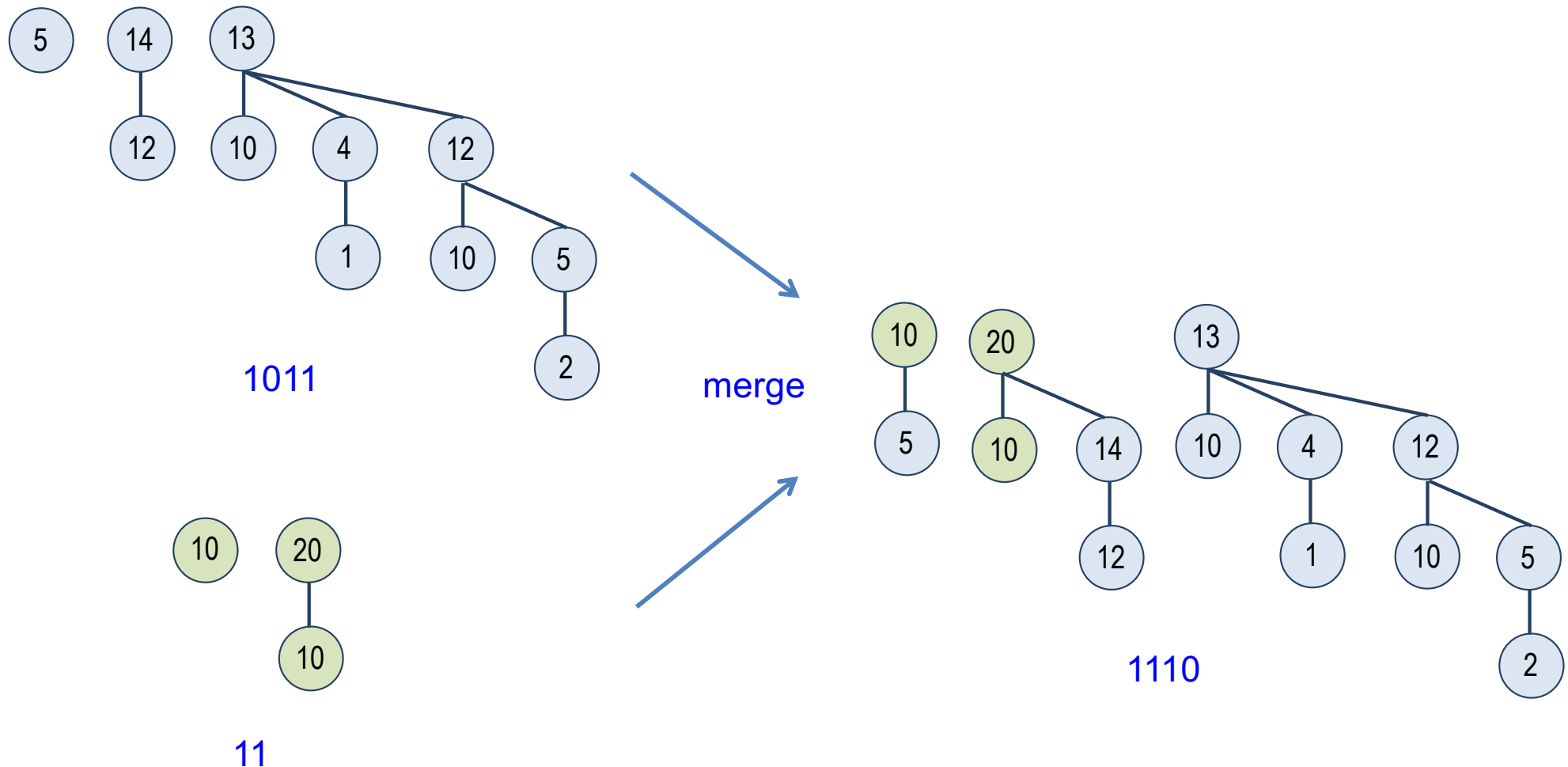


Merge-Operation bei binomialen Heaps

- Zwei binomiale Heaps werden miteinander verschmolzen, indem jeweils Binomialbäume der gleichen Größe miteinander verschmolzen werden.
- Binomiale Heaps lassen sich zu Binärzahlen $B_n \dots B_1 B_0$ abstrahieren, wobei die Ziffer B_k für das Vorhandensein des Binomialbaums B_k steht. Die Merge-Operation entspricht dann der Addition von Binärzahlen.
- Da ein Heap mit n Elementen aus maximal $\log(n)$ Binomialbäumen besteht, benötigt merge $O(\log n)$.



Weiteres Beispiel für Merge-Operation bei binomialen Heaps



Operation insert und delMax

- insert und delMax werden auf die Operation merge zurückgeführt und benötigen daher ebenfalls $O(\log n)$.

```
public void merge(BinomialHeap p) {  
    verschmelze Binomial-Heap p zu diesem  
    Binomial-Heap wie auf Seite 6-40 beschreiben;  
}
```

```
public void insert(int x) {  
    erstelle einen BinomialHeap p mit einem Element x;  
    this.merge(p);  
}
```

```
public int delMax() {  
    bestimme aus diesem Binomial-Heap den  
    Binomialbaum B mit maximaler Wurzel;  
    lösche B aus diesem Binomial-Heap;  
    bilde aus den Kindern von B ein Binomial-Heap p;  
    this.merge(p);  
    return Wert der Wurzel von B;  
}
```

Übersicht über Laufzeiten

Datenstruktur	deleteMax	insert	build	merge
Binäre Heaps	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$
Binomiale Heaps	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$

- Alle Komplexitätsangaben für Prioritätslisten mit n Elementen.
- merge wird bei binären Heaps nicht direkt unterstützt und kann wie die Operation build realisiert werden.
- build wie bei binären Heaps in Bottom-Up-Weise:
 - (1) Beginne mit n heapgeordneten Binomialbäumen jeweils der Größe 1.
 - (2) Verschmelze jeweils 2 heapgeordnete Binomialbäume, so dass $n/2$ viele heapgeordnete Binomialbäume der doppelten Größe entstehen.
(Falls Anzahl Binomialbäume ungerade, bleibt ein Binomialbäume übrig, der nicht verschmolzen wird)
 - (3) Wiederhole Schritt (2), bis keine Verschmelzung mehr möglich.
 - (4) Ergebnis ist ein binomialer Heap.