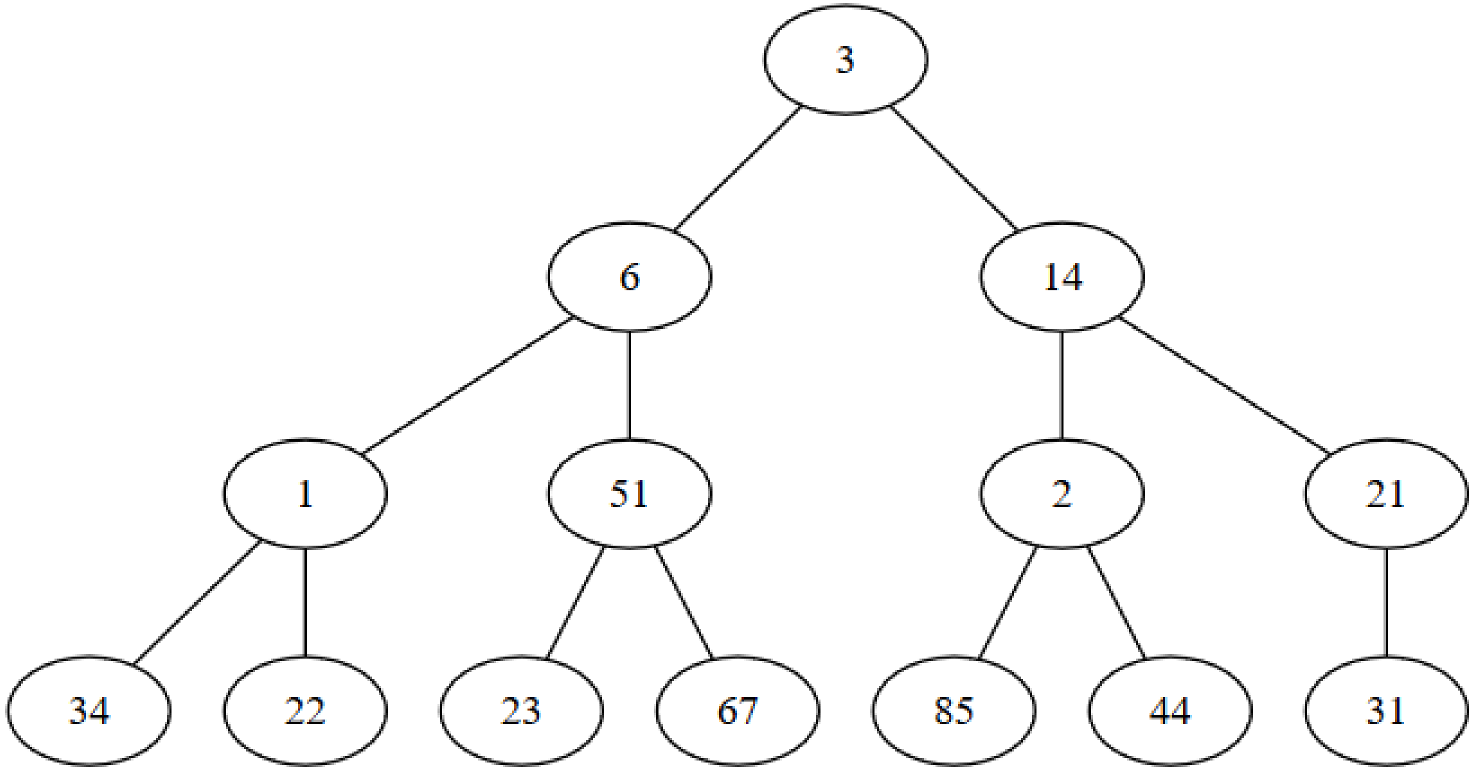
Datenstrukturen und Algorithmen

Vorlesung 8

Überblick

- Vorige Woche:
  - Doppelt verkettete Listen auf Arrays
  - Binärer Heap
- Heute betrachten wir:
  - Binärer Heap
  - Binomial-Heap

Heap - Wiederholung



3	6	14	1	51	2	21	34	22	23	67	85	44	31
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Heap - Wiederholung

- Ein binären Heap **ist ein Array**, das als Binärbaum visualisiert werden kann und, dass zusätzlich die ***Heap-Struktur*** und ***Heap-Eigenschaft*** besitzt

  - Heap-Struktur: ein Binärbaum, in welchem jeder Knoten genau zwei Kinder hat, außer der letzten zwei Niveaus, wo die Knoten von links nach rechts ausgefüllt werden

  - Heap-Eigenschaft: $a_i \geq a_{2*i}$ (falls $2 * i \leq n$) und $a_i \geq a_{2*i+1}$ (falls $2 * i + 1 \leq n$)

  - Ein Baum erfüllt die Heap-Eigenschaft bezüglich einer Vergleichsrelation „$\geq$“ auf den Schlüsselwerten genau dann, wenn für jeden Knoten *u* des Baums gilt, dass *u.wert* $\geq$ *v.wert* für alle Knoten *v* aus den Unterbäumen von *u* (**man kann jedwelche Vergleichsrelation auswählen**)

Naiver Heapaufbau (top-down Strategie)

- Der Heap wird von oben nach unten (top-down) aufgebaut, indem:
  - ein neues Element möglichst weit rechts eingefügt wird und
  - rekursiv nach oben getauscht wird, solange es größer als sein Elternknoten ist

- Diese Methode wird benutzt, wenn die Elemente des Heaps nicht von Anfang an bekannt sind (man erstellt also den Heap durch n-faches Einfügen eines Elementes)
- Zeitkomplexität der Einfüge-Operation ist $O(log_2 n)$

$\Rightarrow$ zum Aufbau eines Heaps mit *n* Elementen benötigt man $O(n log_2 n)$

Verbessertes Heapaufbau (bottom-up Strategie)

- Das Erstellen des Heaps kann verbessert werden wenn man alle Elemente des Heaps vom Anfang an kennt (ein Array, das nicht sortiert ist)
  - Man nimmt an, dass die zweite Hälfte des Arrays nur Blätter enthält
  - Anfangend von dem ersten Element in der Mitte des Arrays, das **nicht ein Blatt ist**, **bis zu dem ersten Element** des Arrays , ruft man *bubble-down* auf
- **Zeitkomplexität: *O(n)***

Heapsort

- Heapsort ist ein Sortierverfahren basierend auf Heap
- Nehmen wir an, dass folgende Sequenz in steigender Reihenfolge sortiert werden muss:

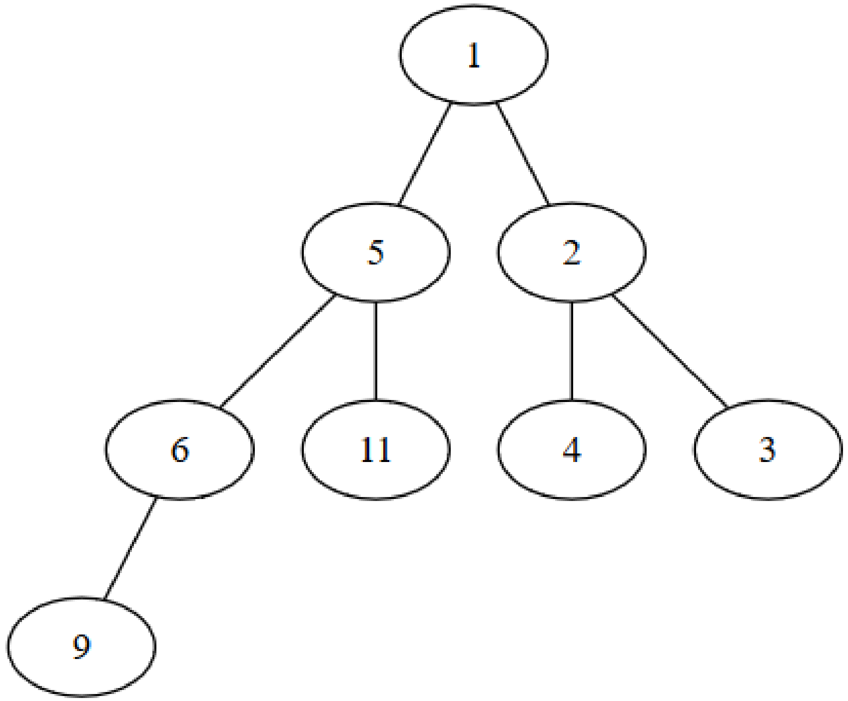
  6, 1, 3, 9, 11, 4, 2, 5

Heapsort – naive Vorgehensweise

- Idee:
  - Erstelle ein Min-Heap und füge alle Elemente der Sequenz ein (Naiver Heapaufbau)
  - Lösche alle Elemente aus dem Heap der Reihe nach: diese werden in aufsteigender Reihenfolge gelöscht, da immer nur die Wurzel gelöscht wird

Heapsort – naive Vorgehensweise

- Min-Heap mit den Elementen der Sequenz:



- Bei dem Löschen der Elemente werden diese in folgender Reihenfolge gelöscht: 1, 2, 3, 4, 5, 6, 8, 11

Heapsort – naive Vorgehensweise

- Welche ist die Zeitkomplexität der naiven Vorgehensweise?

  $O(nlog_2n)$

  (n-faches Einfügen – $nlog_2n$, n-faches Löschen – $nlog_2n$)

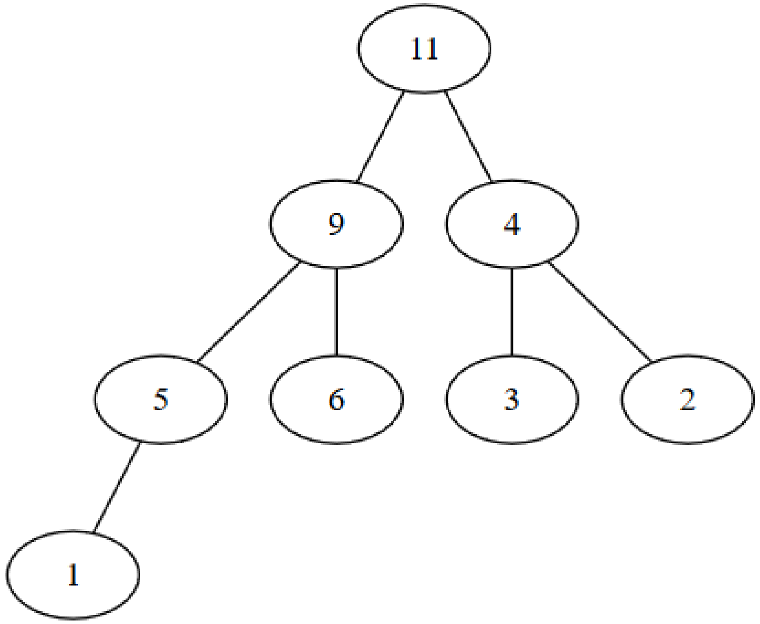
- Welche ist die Speicherkomplexität? Braucht man zusätzliches Speicherplatz?
  - man braucht ein zusätzliches Array – Θ(n)

Heapsort – verbesserte Vorgehensweise

- Wenn man eine Sequenz in **aufsteigender** Reihenfolge sortieren will, dann benutzt man einen **Max-Heap**

  ⇒ dann braucht man kein zusätzliches Speicherplatz für ein Array - Warum?

- Zusätzlich kann man den verbesserten Heapaufbau benutzen



Heapsort – verbesserte Vorgehensweise

- Welche ist die Zeitkomplexität der verbesserten Vorgehensweise?
  - die Komplexität für den Heapaufbau ist *O(n),*
  - aber die Komplexität für das Löschen der Elemente bleibt $O(n log_2 n)$

- Welche ist die Speicherkomplexität?
  - Man braucht kein zusätzliches Speicherplatz

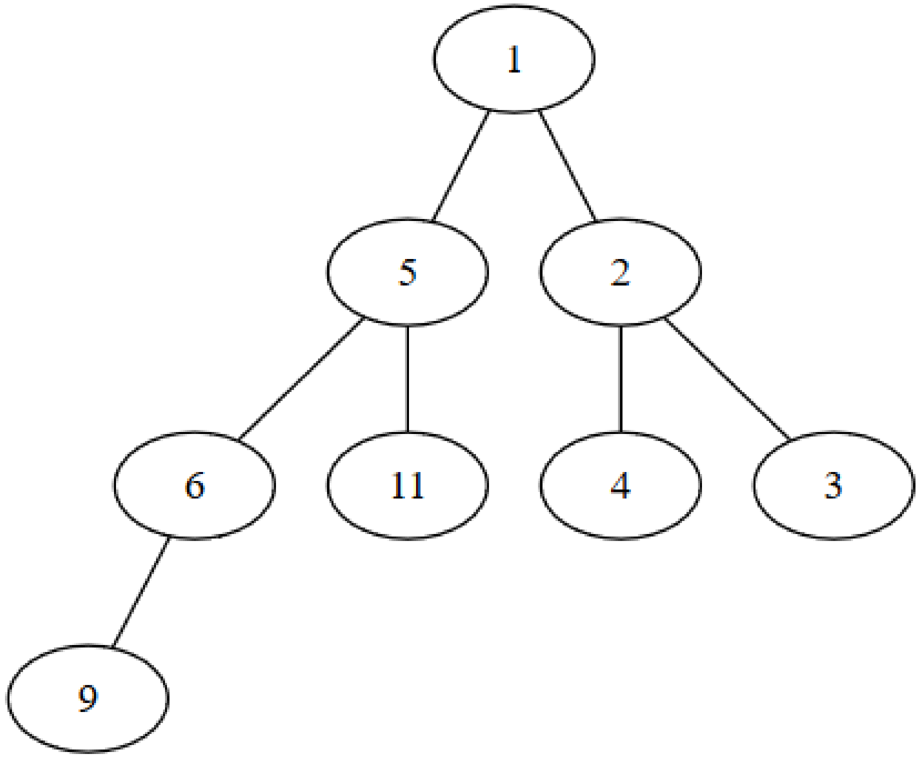
Heapsort im Vergleich

- Heapsort und Quicksort sind **nicht stabil**, Mergesort ist **stabil**
- Quicksort hat im schlimmsten Fall $\Theta(n^2)$ - wenn man das Pivot nicht richtig wählt (eher unwahrscheinlich in der Praxis)
- Mergesort benötigt zusätzlichen Speicherplatz für Arrays
- Mergesort ist sehr leicht zu parallelisieren
- Mergesort funktioniert auf verkettete Listen ohne zusätzlichen Speicherplatzbedarf (ähnlich Seminar 4)

Es gibt keinen "allgemein besten" Algorithmus, aber es gibt einen <u>passenden!</u>

ADT Prioritätsschlange – Repräsentierung auf binärem Heap

- Eine effiziente Repräsentierung der Prioritätsschlange ist mit Hilfe eines binären Heaps, in welchem das Element mit der höchsten Priorität in der Wurzel des Heaps gespeichert wird



ADT Prioritätsschlange – Repräsentierung auf binärem Heap

- Wenn ein Element in die Prioritätsschlange eingefügt wird, dann braucht man nur das Element in den Heap einzufügen

- Wenn ein Element aus der Prioritätsschlange entfernt wird, dann wird die Wurzel des Heaps entfernt

- Bei top wird die Wurzel zurückgeben

ADT Prioritätsschlange – Repräsentierungen

- Komplexitäten für die unterschiedlichen Repräsentierungen:

Operation	Sortiert	Nicht-sortiert	Heap
push	$O(n)$	$\Theta(1)$	$O(\log_2 n)$
pop	$\Theta(1)$	$\Theta(n)$	$O(\log_2 n)$
top	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

- Welche ist die totale Komplexität für folgende Sequenz von Operationen:
  - Man fängt mit einer leeren Prioritätsschlange an
  - Man fügt *n* Elemente in die Prioritätsschlange
  - Man entfernt *n* Elemente aus der Prioritätsschlange

Prioritätsschlange – Erweiterung

- Wir haben die Standard-Operationen aus dem Interface der Prioritätsschlange besprochen:
  - push
  - pop
  - top
  - isEmpty
  - init
- Manchmal sind folgende Operationen auch nützlich:
  - Die Priorität eines Elementes ändern
  - Ein beliebiges Element löschen
  - Merge zweier Prioritätsschlangen

Prioritätsschlange – Erweiterung

- Welche ist die Komplexität dieser Operationen wenn die Prioritätsschlange auf binären Heap repräsentiert ist:
  - Die Priorität eines Elementes erhöhen ist $O(log_2 n)$ wenn man die Position des Elementes kennt
  - Ein beliebiges Element löschen ist $O(log_2 n)$ wenn man die Position des Elementes kennt
  - Merge zweier Prioritätsschlangen hat die Komplexität $\Theta(n+m)$, wobei die Prioritätsschlangen *n* und *m* Elemente enthalten
    (oder $\Theta(n)$ falls beide *n* Elemente enthalten)

Anwendungen von Prioritätenschlangen

- Graphentheorie:
  - Finden des minimalen Spannbaums (Prim's Algorithmus)
  - Dijkstra's Algorithmus in einem gewichteten Graph
  - Aufgaben mit Graphen kommen sehr oft vor und sind sehr groß, z.B.:
    eine Firma will ein Logistikzentrum, Fabrik, usw. öffnen - wo?
    eine Band geht auf Tour, wie soll man das organisieren?
- Zugriff prioritisieren auf verschiedene begrenzte Ressourcen

Prioritätsschlange – andere Repräsentierungen

- Wenn man kein Merge zwischen Prioritätsschlangen braucht, dann sind binären Heaps gute Repräsentierungen

- Wenn man die Merge Operation braucht, dann gibt es andere Datenstrukturen, die als Repräsentierung für Prioritätsschlangen besser geeignet sind (kleinere Komplexität)

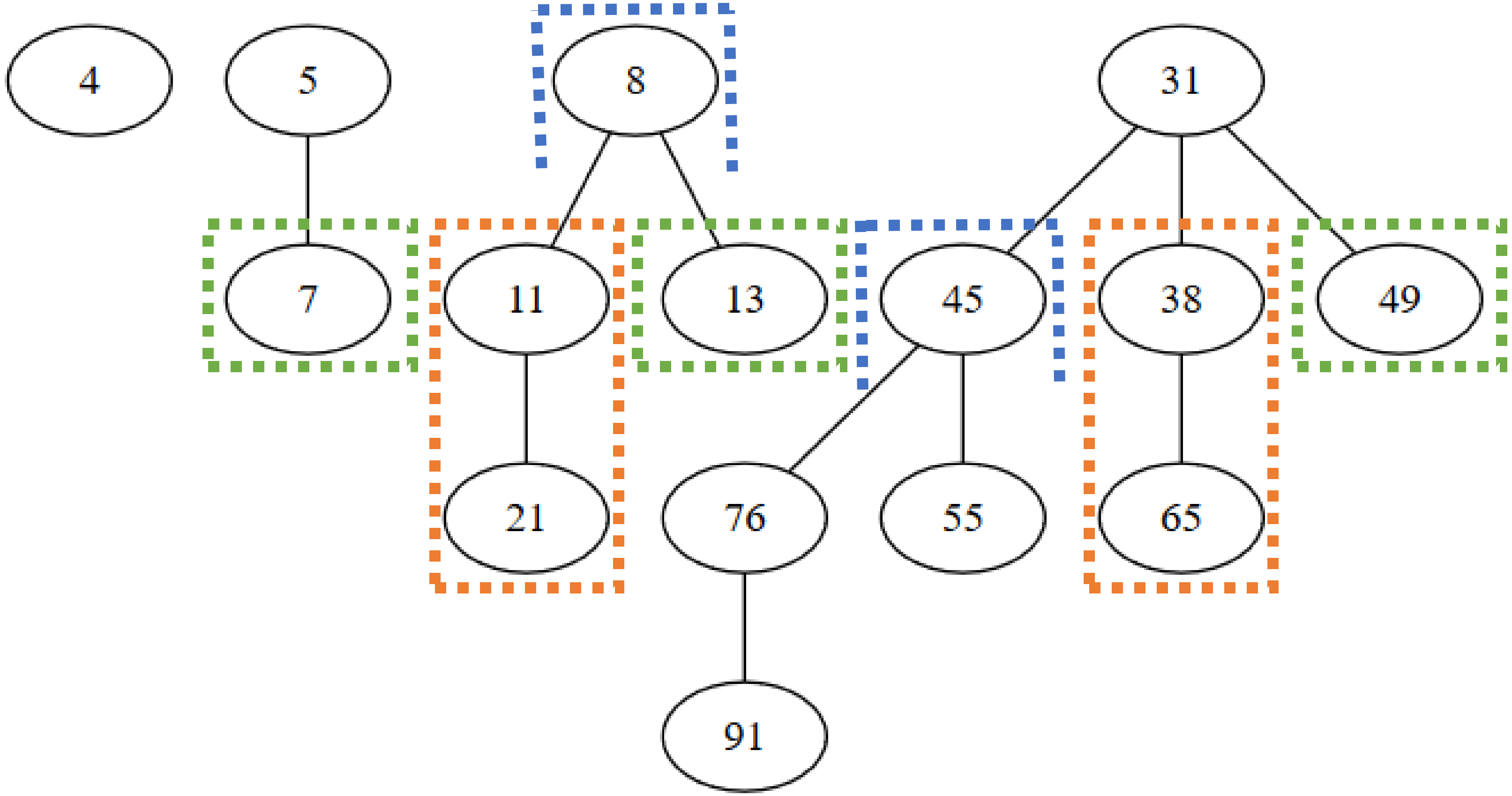
- Davon besprechen wir: *Binomial-Heap*

Binomial Heap

- Ein Binomial Heap besteht aus einer Sequenz von Binomial-Bäumen verschiedener Ordnung.

- Binomial-Bäume können wie folgt rekursiv definiert werden:
  - Ein *Binomial-Baum der Ordnung 0* besteht aus einem einzelnen Knoten.
  - Ein *Binomial-Baum der Ordnung k* besitzt eine Wurzel mit Grad *k,* deren Kinder genau die Ordnung *k*-1, *k*-2, ..., 0 (in dieser Reihenfolge) besitzen.

Binomialbaum – Beispiele



Binomialbäume der Ordnung 0, 1, 2 und 3

Binomial-Baum

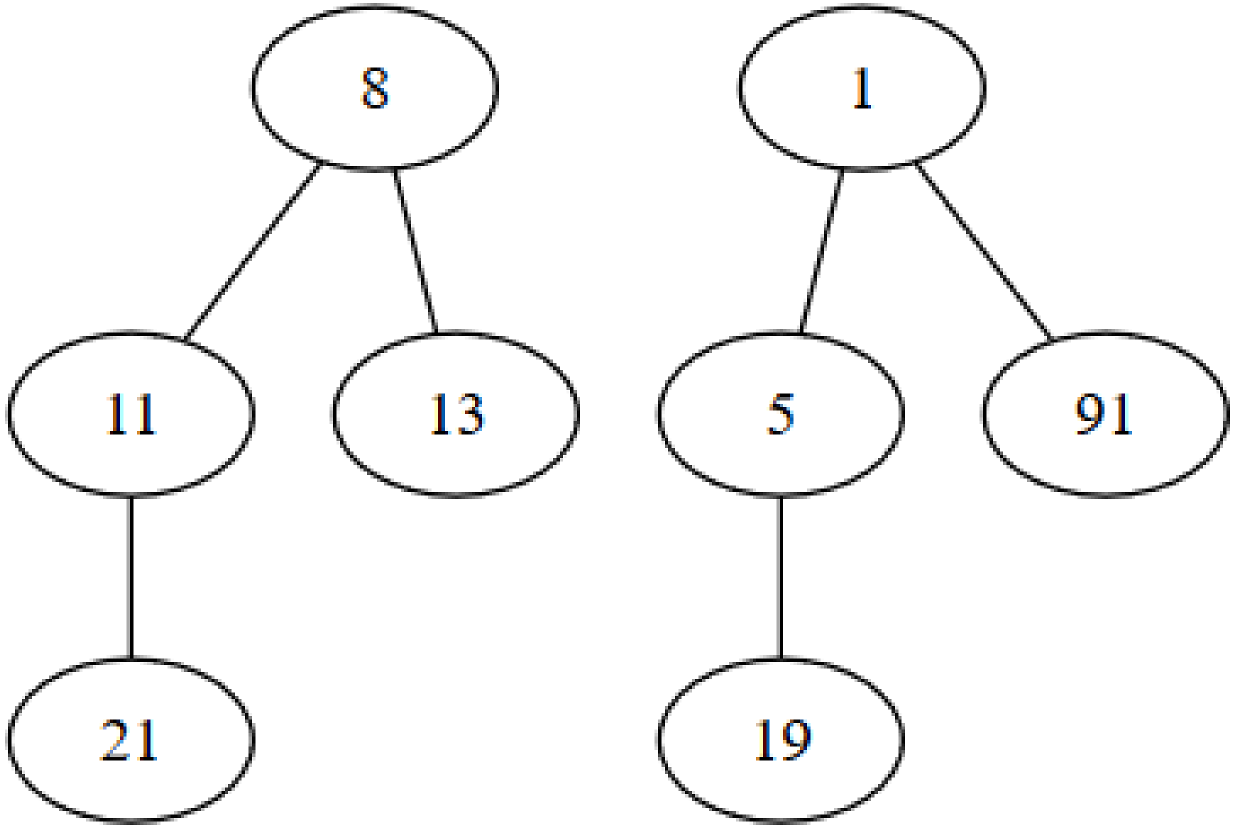
- Ein Binomial-Baum der Ordnung k hat genau 2^k Knoten

- Die Höhe eines Binomial-Baumes der Ordnung k ist k

- Wenn man die Wurzel eines Binomial-Baumes der Ordnung k löscht, dann kriegt man k Binomial-Bäume der Ordnung k-1, k-2, ..., 2, 1, 0.

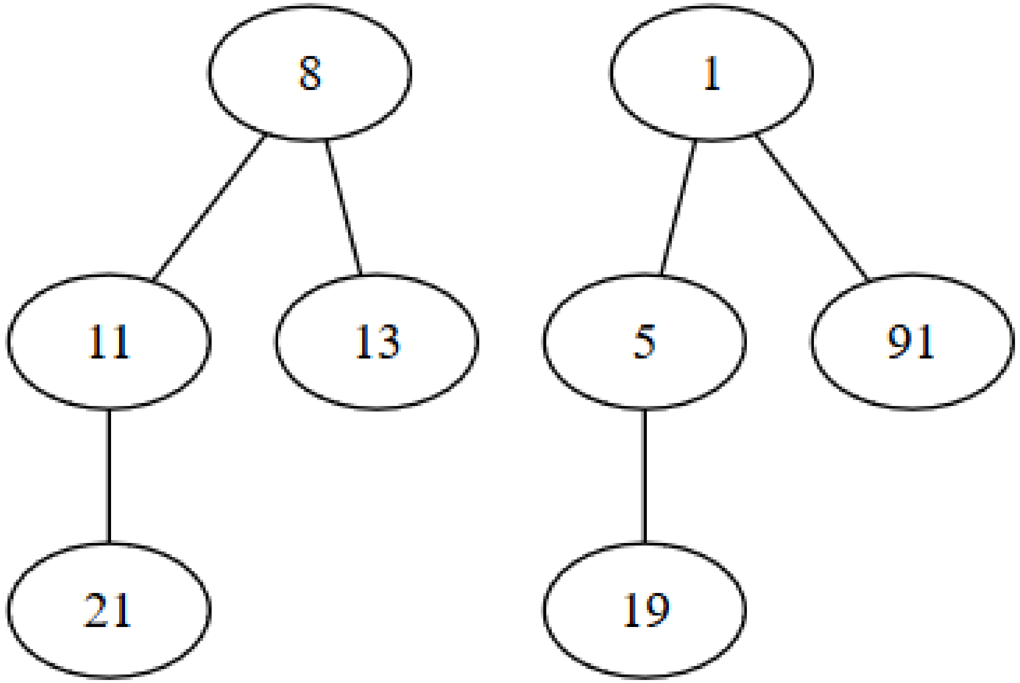
- Ein Binomial-Baum der Ordnung *k* lässt sich leicht aus zwei Binomial-Bäumen der Ordnung *k*-1 erstellen (merge), indem einer der beiden Bäume zum am weitesten links stehenden Kind des anderen gemacht wird.

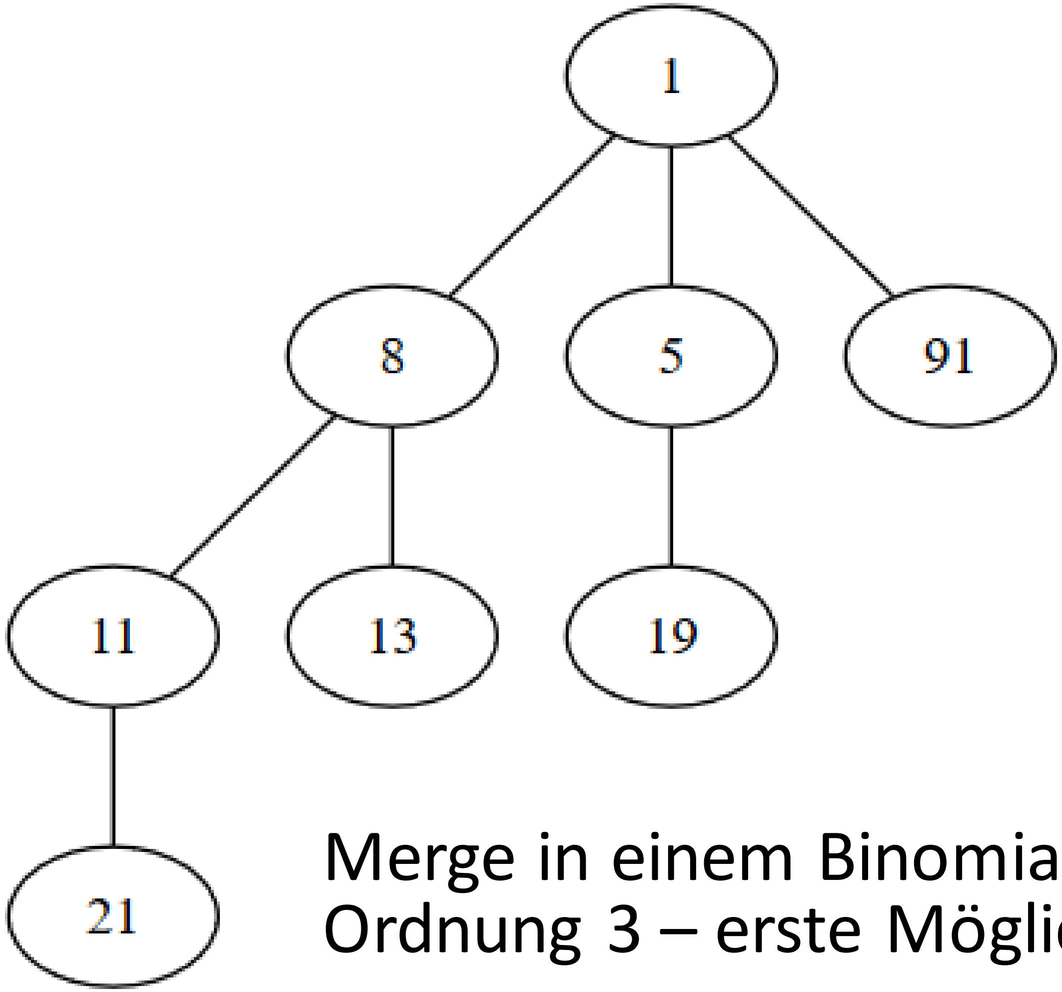
Binomial-Baum – Merge



Zwei Binomial-Bäume der Ordnung 2 (vor dem Merge)

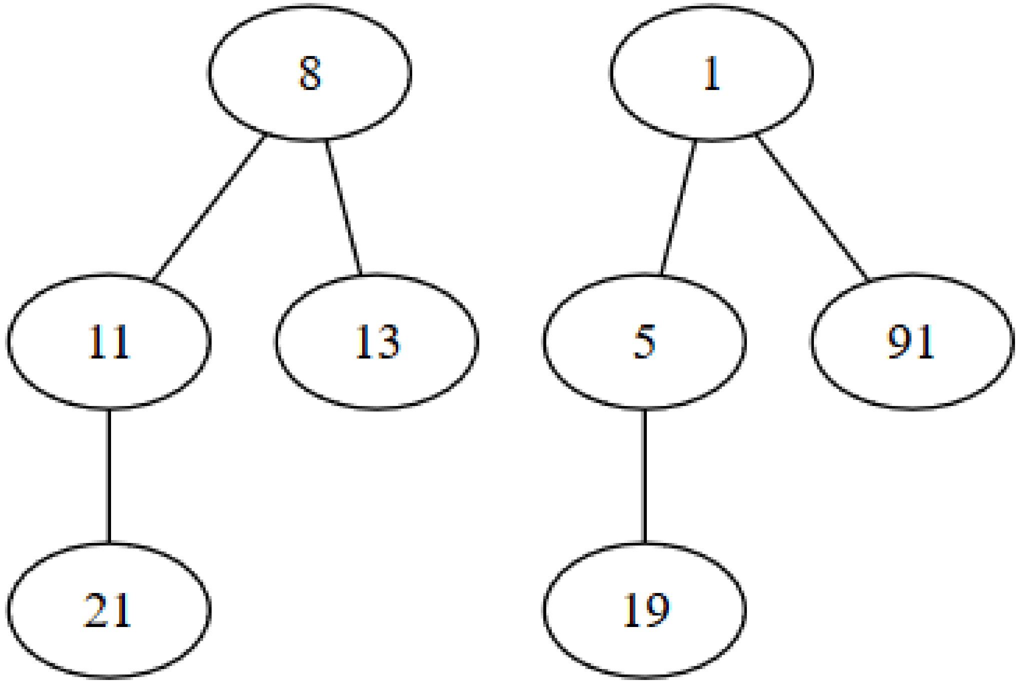
Binomial-Baum – Merge

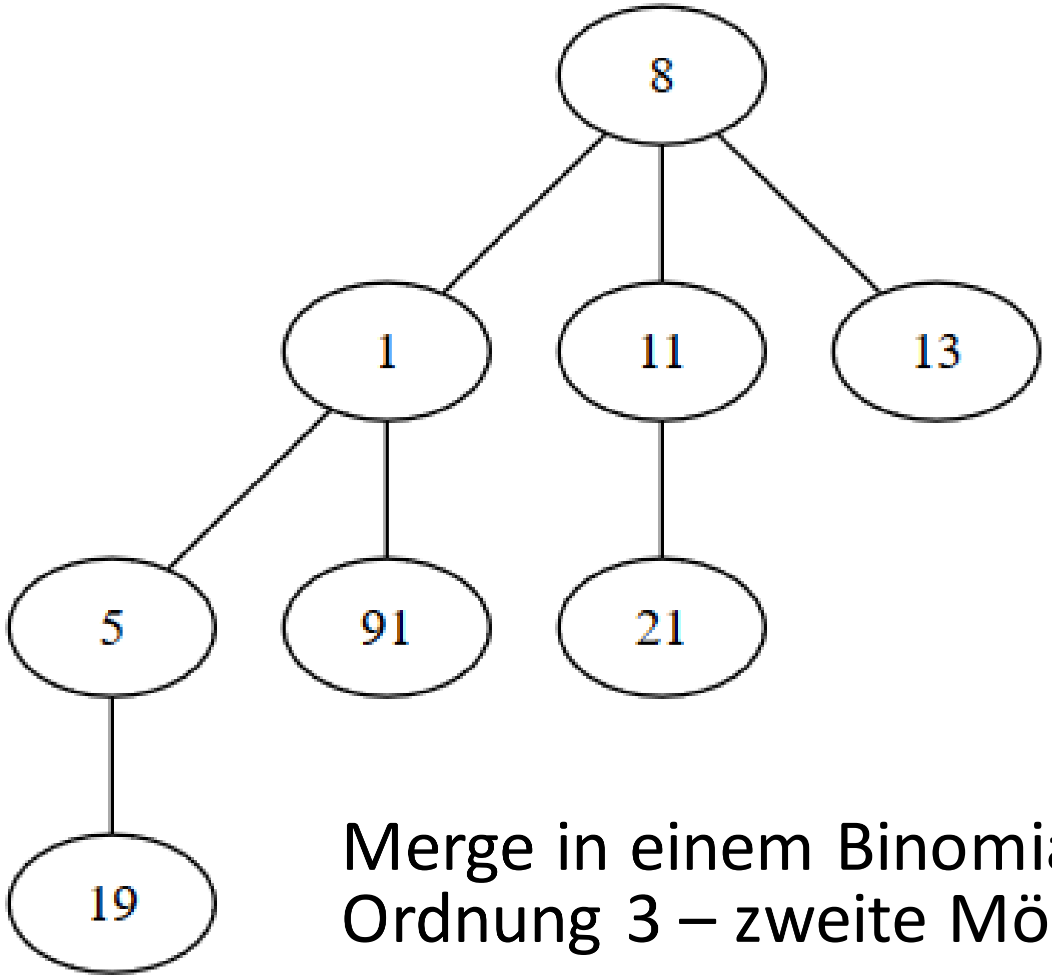




Merge in einem Binomial-Baum der Ordnung 3 – erste Möglichkeit

Binomial-Baum – Merge





Merge in einem Binomial-Baum der Ordnung 3 – zweite Möglichkeit

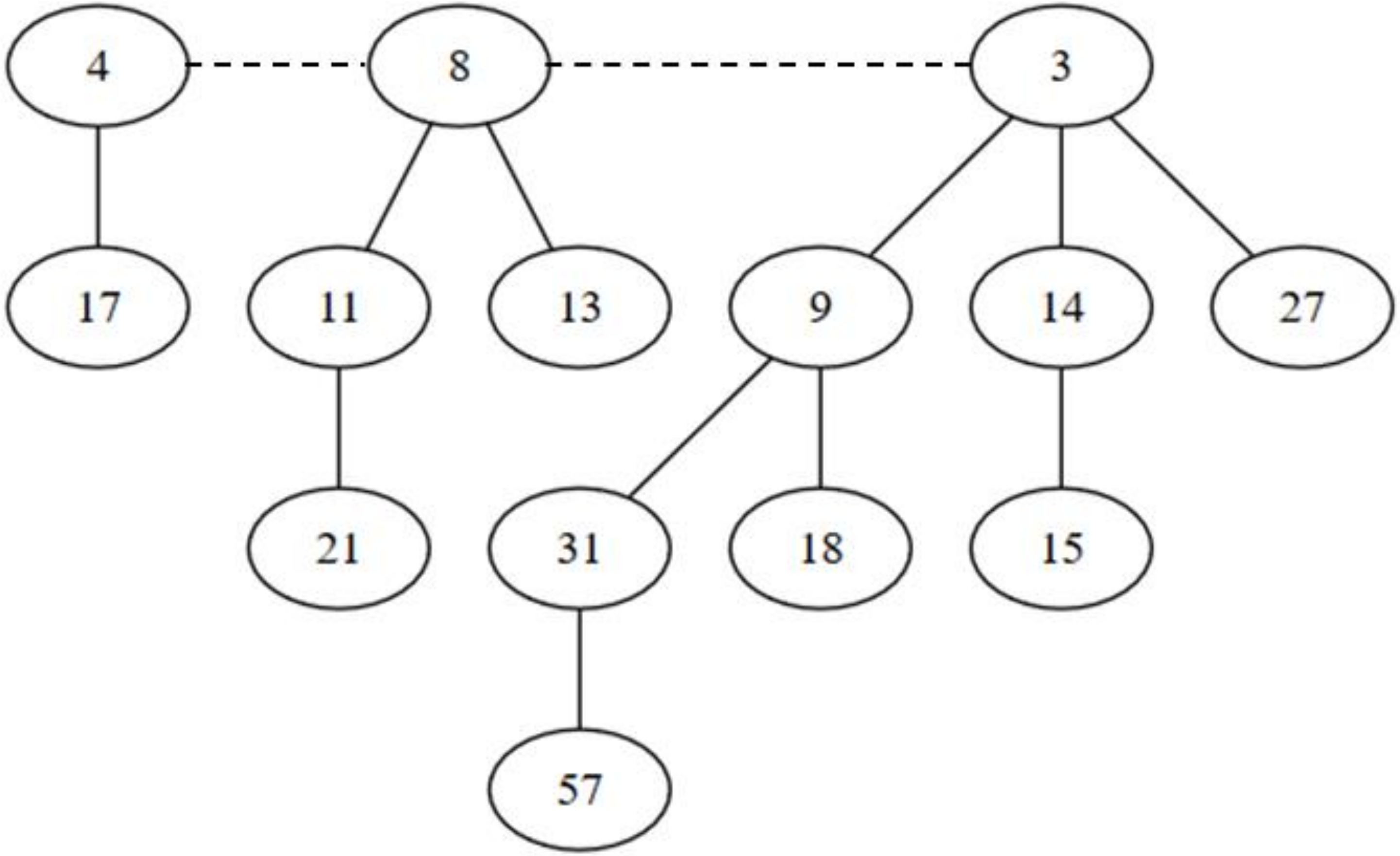
Binomial-Baum – Repräsentierung

- Um einen Binomial-Baum zu implementieren kann man folgende Repräsentierung benutzen:
  - Man braucht eine Struktur für die Knoten. Für jeden Knoten speichert man Folgendes:
    - Die Information aus dem Knoten
    - Die Adresse des Vaterknotens
    - Die Adresse des ersten Kind-Knotens
    - Die Adresse des nächsten Bruder-Knotens
  - Für den Baum speichert man die Adresse des Wurzelknotens (und vielleicht die Ordnung des Baumes)

Binomial-Heap

- Ein **Binomial-Heap** besteht aus einer **Sequenz von Binomial-Bäumen** mit folgenden Eigenschaften:
    1. Jeder Binomial-Baum erfüllt die **Heap-Eigenschaft**: für jeden Knoten ist der Wert kleiner als die Werte seiner Kinder (für Min-Heap)
    2. Es gibt höchstens ein Binomial-Baum der Ordnung k
    3. Als Repräsentierung, ist ein Binomial-Heap meistens eine sortierte verkettete Liste, wobei jeder Knoten ein Binomial-Baum enthält und die Knoten **nach der Ordnung der Bäume sortiert** sind

Binomial-Heap – Beispiel



Binomial-Heap mit 14 Knoten; besteht aus 3 Binomial-Bäume der Ordnung 1, 2 und 3

Binomial-Heap

- Für eine gegebene Anzahl von Elementen *n*, ist die Struktur des Binomial-Heaps (die Anzahl der Binomial-Bäume und deren Ordnung) eindeutig

- Die Struktur der Binomial-Heap wird von der binären Darstellung der Zahl *n* bestimmt
- Zum Beispiel $14 = 1110_2 = 2^3 + 2^2 + 2^1$, d.h. ein Binomial-Heap mit 14 Knoten enthält Binomial-Bäume der Ordnung 3, 2 und 1 (sortiert in umgekehrter Reihenfolge 1, 2,3)
- Zum Beispiel $21 = 10101_2 = 2^4 + 2^2 + 2^0$, d.h. ein Binomial-Heap mit 21 Knoten enthält Binomial-Bäume der Ordnung 4, 2 und 0

Binomial-Heap

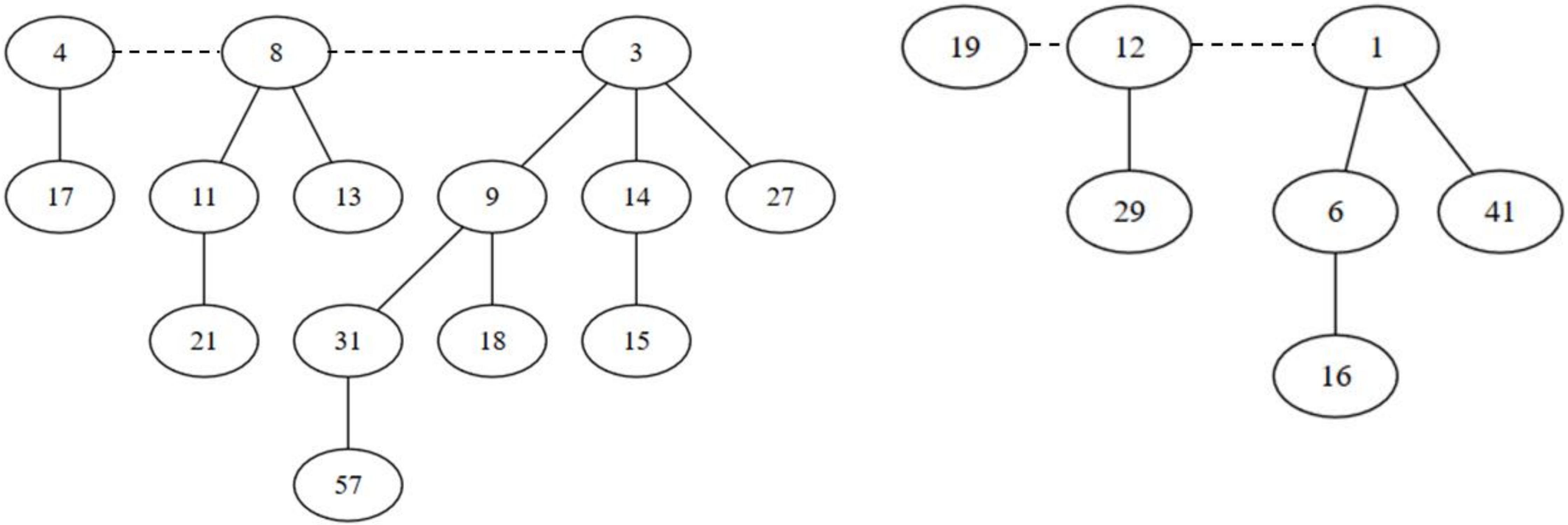
- Ein Binomial-Heap mit n Elementen enthält höchstens $log_2 n$ Binomial-Bäume

- Die Höhe des Binomial-Heaps ist höchstens $log_2 n$

Binomial-Heap – Merge

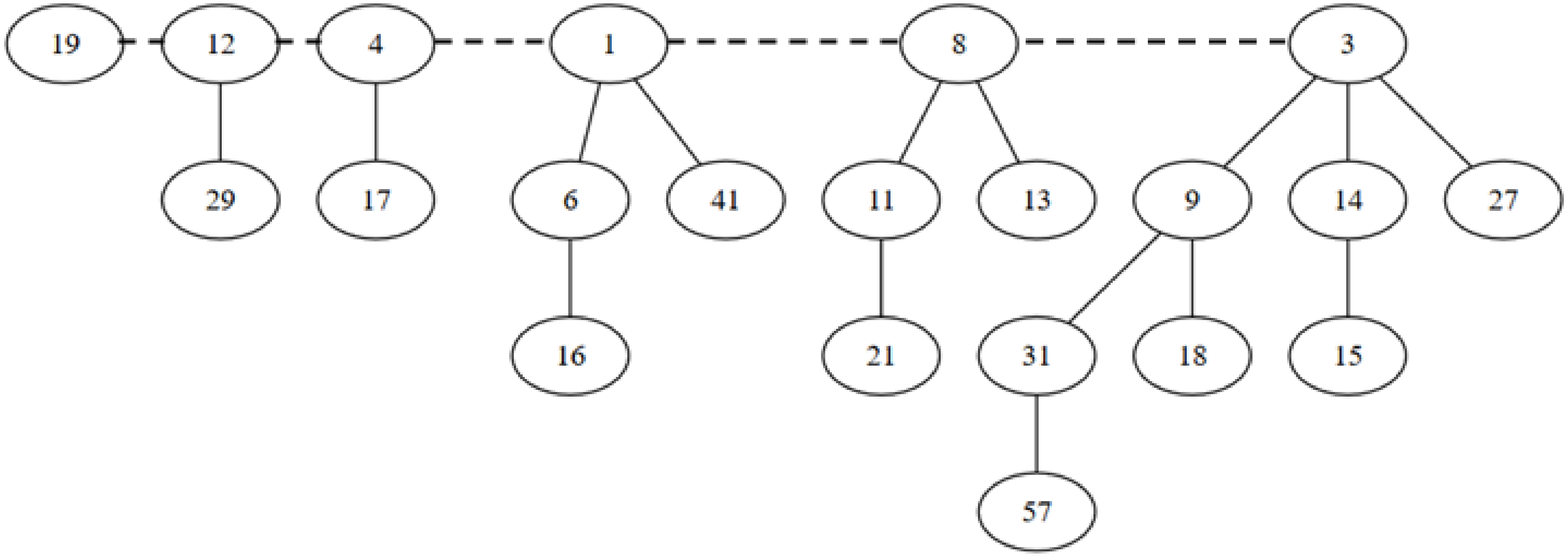
- Die interessante Operation des Binomial-Heaps ist die Merge Operation, die auch von anderen Operationen benutzt wird
- Da beide Binomial-Heaps sortierte verkettete Listen sind, besteht der erste Schritt der Merge-Operation aus dem *Merge* der zwei sortierten Listen
- Das Ergebnis des Merges kann zwei Binomial-Bäume derselben Ordnung enthalten. Man muss also die Liste iterieren und Binomial-Bäume derselben Ordnung *k* werden in einem Binomial-Baum der Ordnung *k+1* vereinigt (merge).
- Bei dem Merge der Binomial-Bäume muss die Heap-Eigenschaft aufbewahrt werden

Binomial-Heap – Merge Beispiel



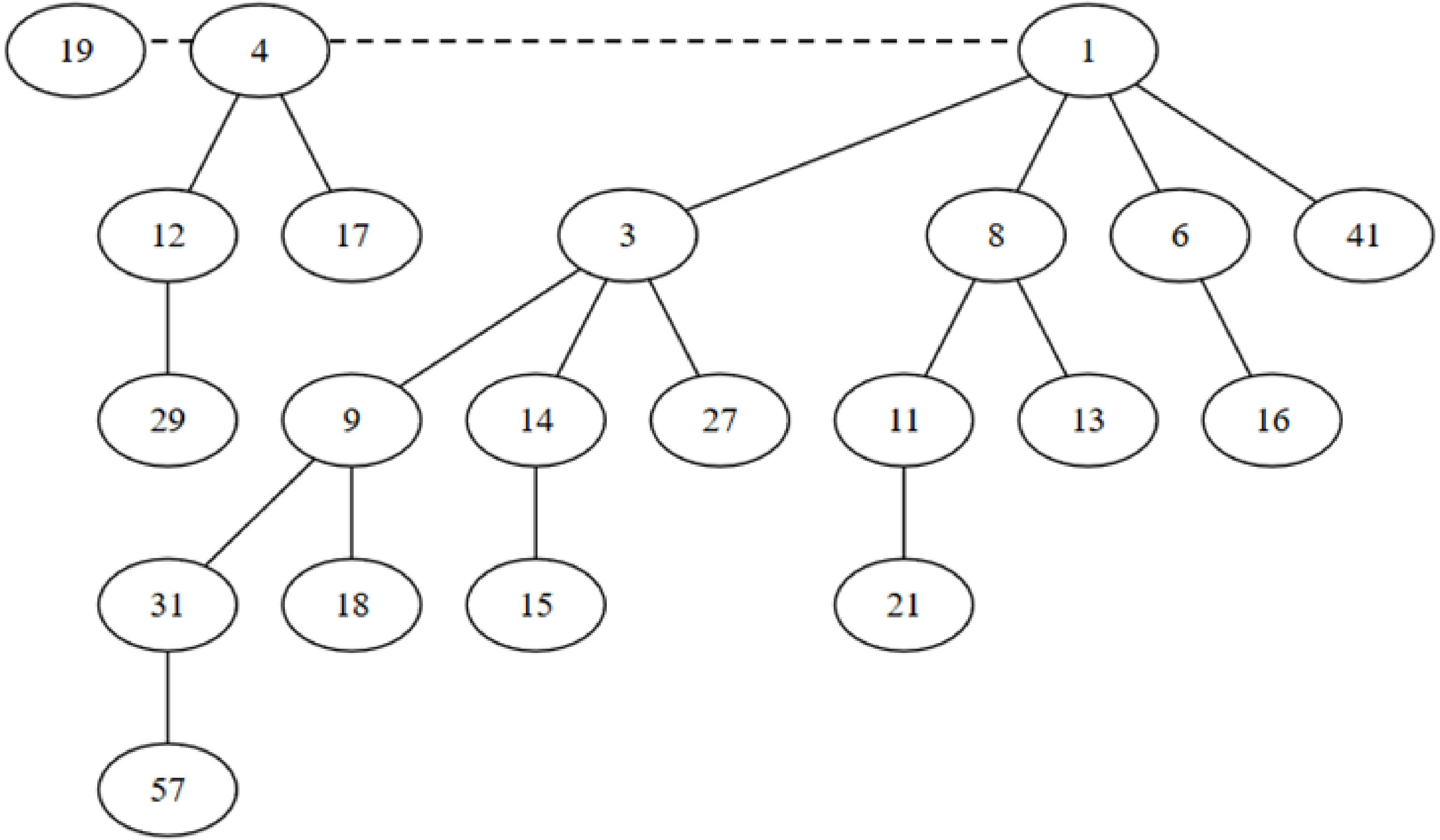
Binomial-Heap – Merge Beispiel

- Nach dem Merge der zwei verketteten Listen:



Binomial-Heap – Merge Beispiel

- Nach dem Merge der Binomial-Bäume derselben Ordnung:



Binomial-Heap – Merge Operation

- Wenn beide Binomial-Heaps *n* Elemente enthalten, dann ist die Komplexität des Merges $O(log_2n)$ (die maximale Anzahl der Binomial-Bäume aus einem Binomial-Heap mit *n* Elementen ist log_2n)

Binomial-Heap – andere Operationen

- Fast alle anderen Operationen für Binomial-Heap benutzen die Merge Operation

- *Push Operation:* Um ein neues Element einzufügen erstellt man einen Binomial-Heap, der dieses Element enthält, und man berechnet den Merge mit dem ursprünglichen Binomial-Heap.
  - Komplexität: $O(\log_2 n)$ im schlimmsten Fall ($\Theta(1)$ amortisiert)

- *Top Operation:* Das kleinste Element aus der Binomial-Heap (das Element mit der höchsten Priorität) ist die Wurzel einer der Binomial-Bäumen. Um das Minimum zurückzugeben muss man über die Wurzeln iterieren, also die Komplexität ist $O(\log_2 n)$

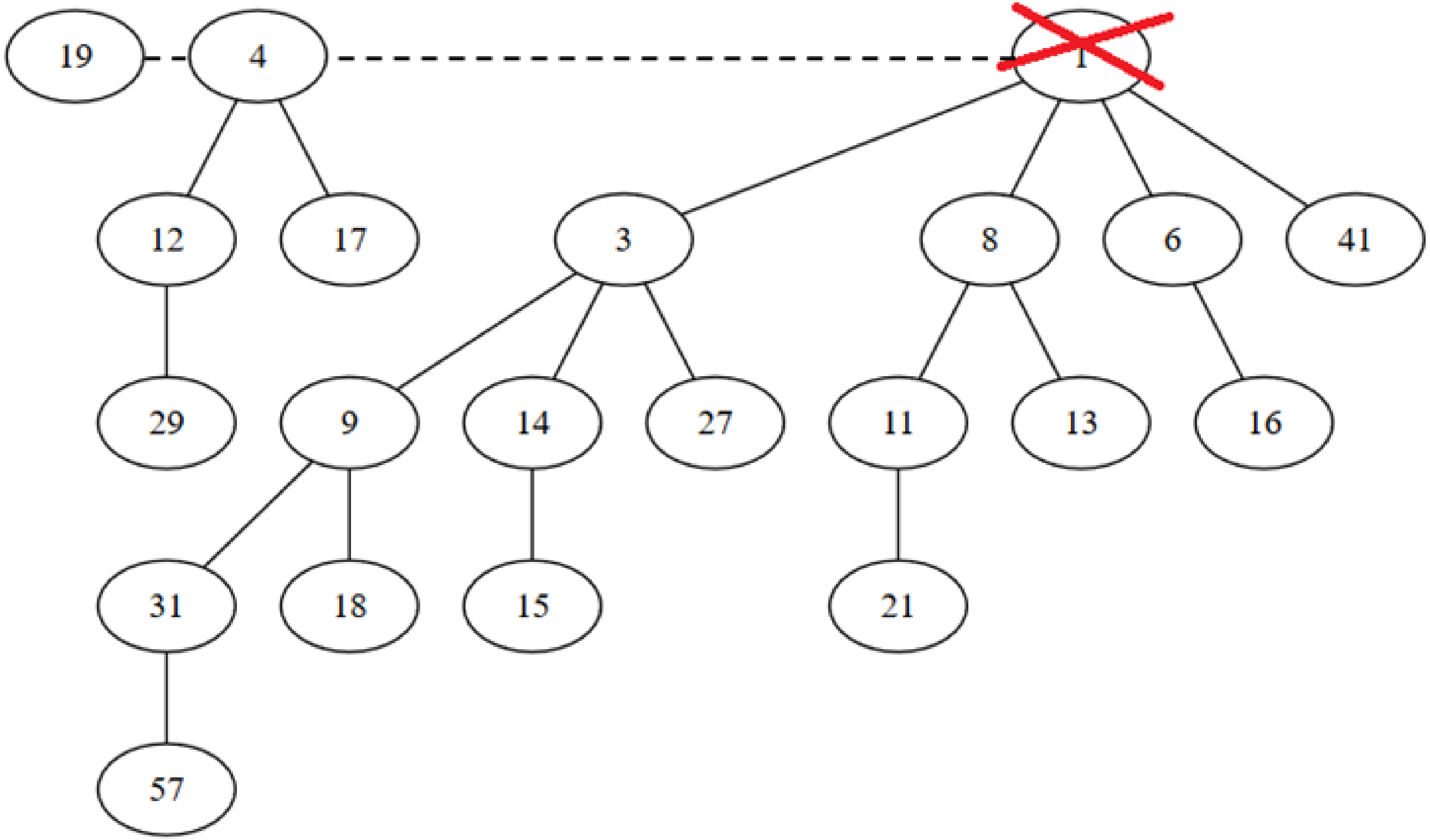
Binomial-Heap – andere Operationen

- *Pop Operation:*
  - Das Minimum entfernen heißt die Wurzel einer der Binomial-Bäumen zu entfernen.
  - Wenn man die Wurzel aus einem Binomial-Baum löscht, dann erhält man eine Sequenz von Binomial-Bäumen.
  - Diese Bäume werden als Binomial-Heap betrachtet (in umgekehrter Reihenfolge).
  - Man berechnet den Merge zwischen den neuen Binomial-Heap und den Rest der Elementen aus dem ursprünglichen Binomial-Heap

  - Die Komplexität ist $O\ (log_2 n)$

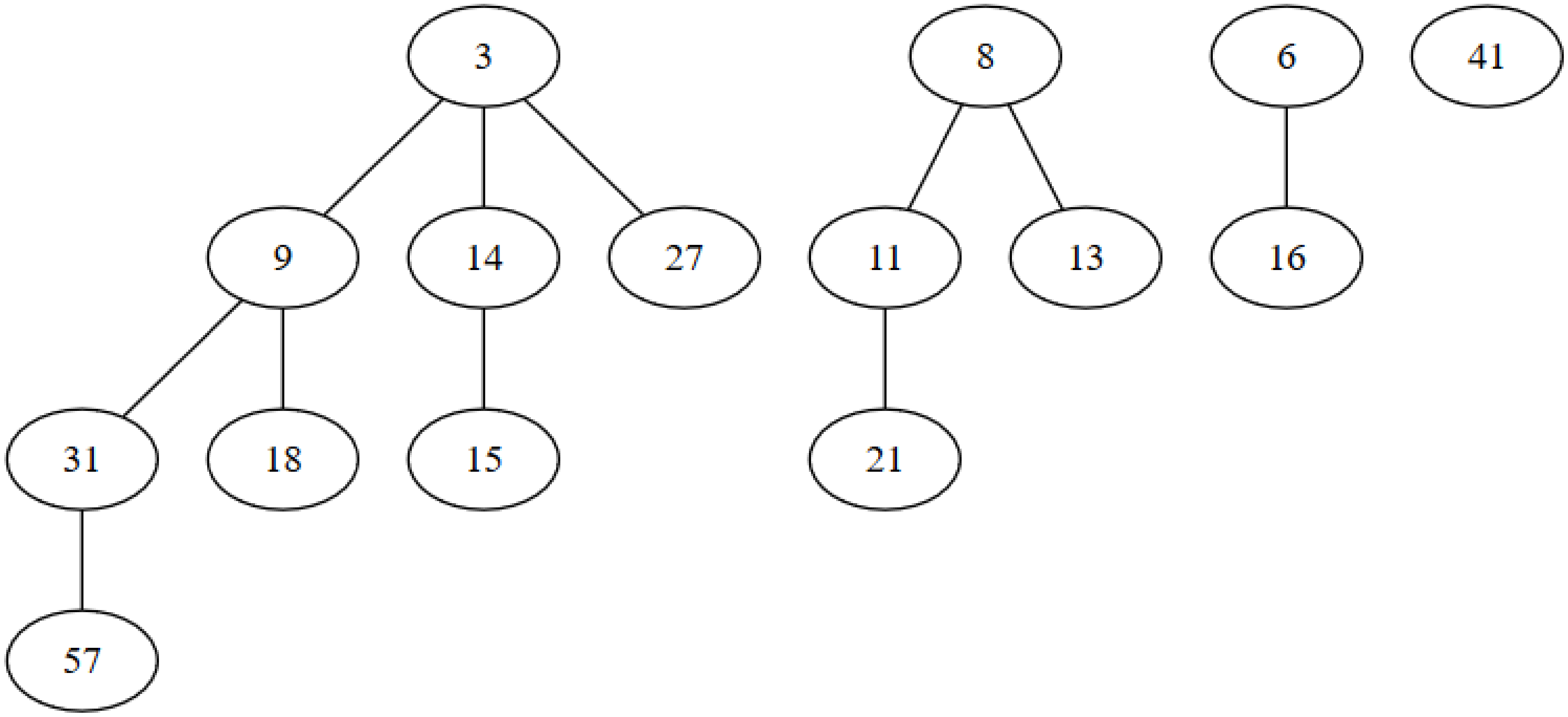
Binomial-Heap – pop

- Das Minimum ist in einer der Wurzeln



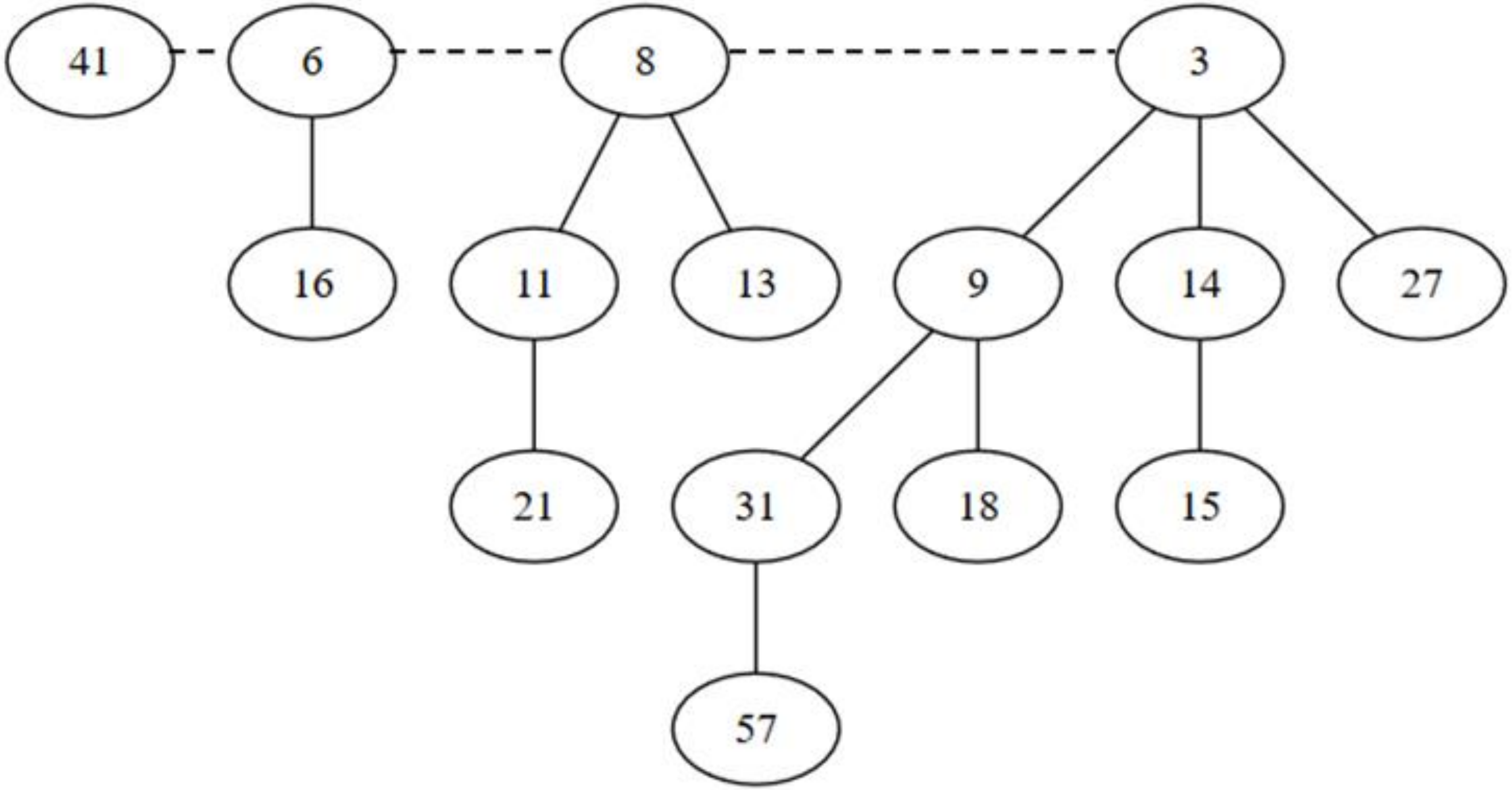
Binomial-Heap – pop

- Aus dem entsprechenden Binomial-Baum bleiben *k* Binomial-Bäume:



Binomial-Heap – pop

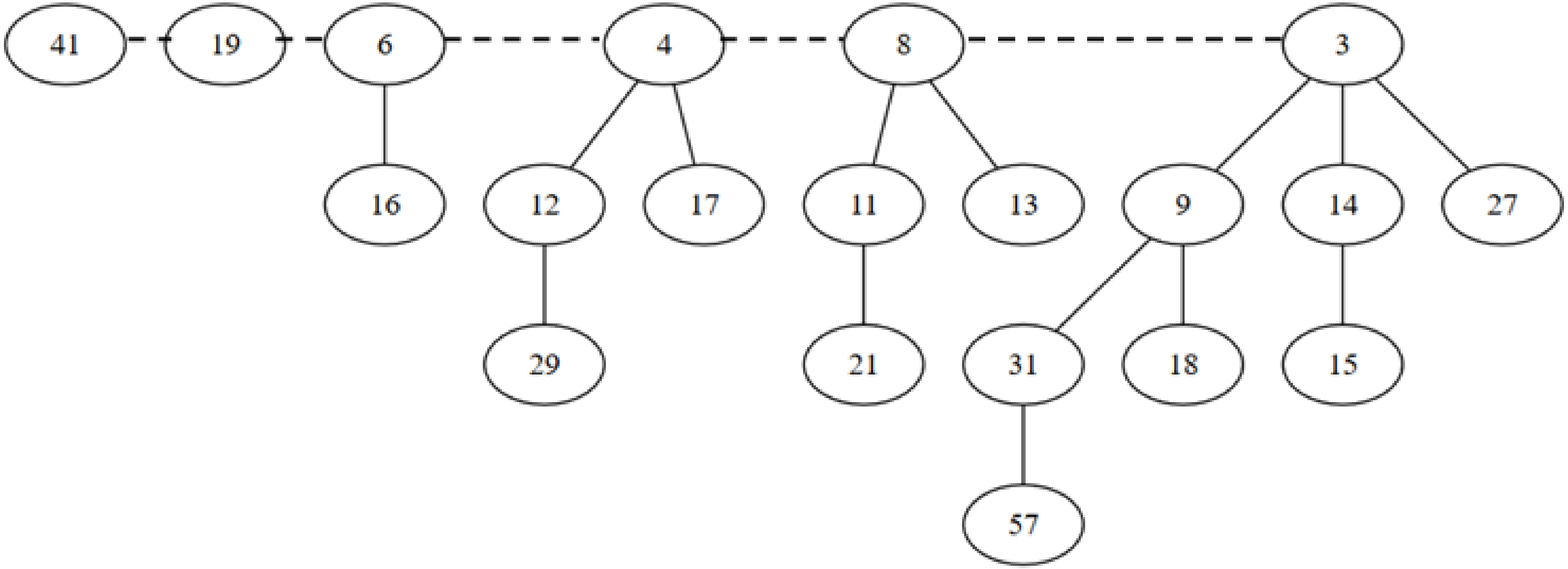
- Betrachte diese als ein Binomial-Heap...



Binomial-Heap – pop

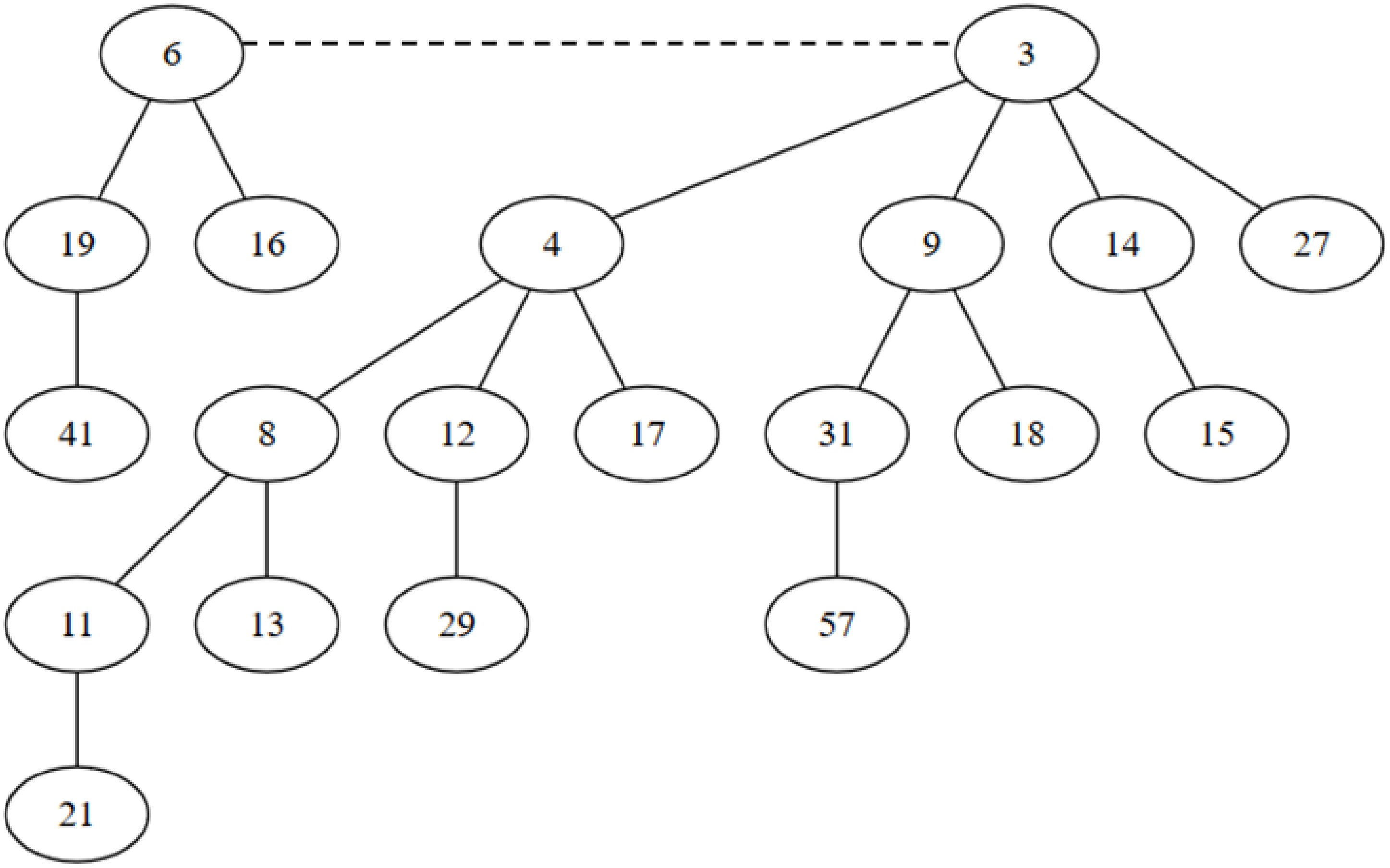
...und berechne den Merge zwischen den zwei Binomial-Heaps

- Erster Schritt der Merge-Operation:



Binomial-Heap – pop

- Zweiter Schritt der Merge-Operation:



Binomial-Heap – andere Operationen

- Priorität eines Elementes erhöhen:
  - Wenn man einen Zeiger zu dem Element hat, deren Priorität erhöht werden muss, dann kann man die Priorität ändern und dann *bubble-up* ausführen, falls die Priorität größer ist als die Priorität des Vaters (in den Beispielen hohe Priorität heißt kleine Zahl)
  - Komplexität: $O(\log_2 n)$

- Ein beliebiges Element löschen:
  - Wenn man ein Zeiger zu dem Element hat, das man löschen will, dann kann man seine Priorität auf $-\infty$ setzen (d.h. das Element wird bis zu dem Wurzel verschoben) und dann löscht man ein Element (also die Wurzel)
  - Komplexität: $O(\log_2 n)$