

Binär und Binomial Heaps

Von: Luca Stamos, Stefan Steinhauer, Dimitri
Osokin, Marc Kevin Zenzen

Inhaltsverzeichnis

- Einleitung / Allgemeines
- Lernziele
- Prioritätswarteschlange (ADT)
- Binär Heaps
- Binomial Heaps
- Laufzeiten
- Implementierung

Einleitung/Allgemeines

- ❑ Binär- und Binomial Heaps als Prioritätswarteschlange (ADT)
- ❑ Binär Heaps erstmals implementiert in 1964 von J. W. J. Williams
- ❑ Erweitert von Robert W. Floyd im selben Jahr
- ❑ Binomial Heaps erstmals beschrieben im Jahr 1978 von Jean Vuillemin
- ❑ Praktischer Einsatz in Servern und Betriebssystemen

Lernziele

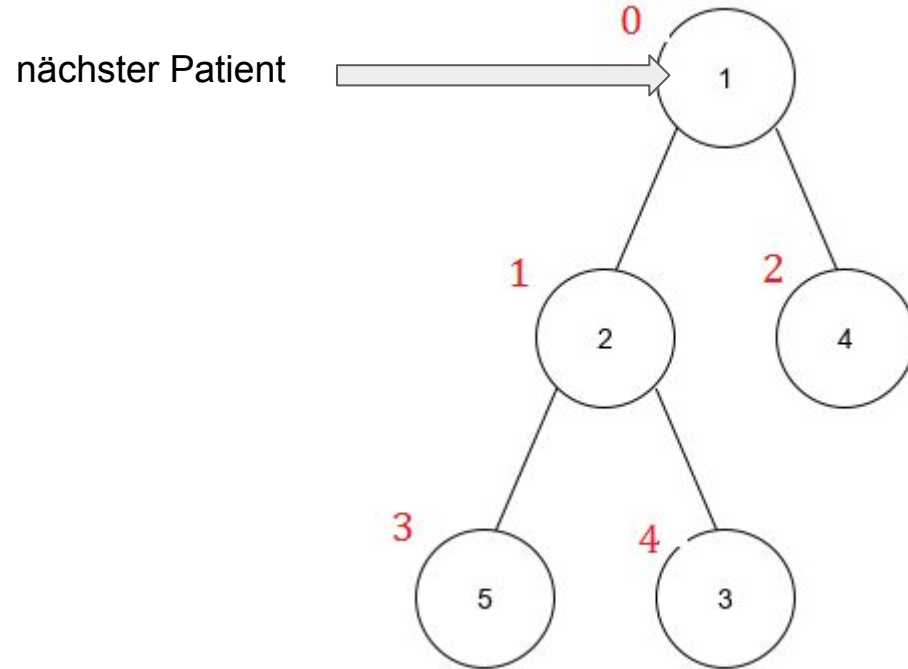
Sie verstehen:

- den Aufbau von Binären Heaps
- den Aufbau von Binomial Heaps
- die Unterschiede zwischen den beiden Heaps
- Sinn der Heap-Strukturen
- verschiedene Funktionen (Operationen) der Heaps
- den Unterschied / Vergleich der Laufzeiten von Heaps und z.B Quicksort

Prioritätswarteschlange (ADT)

- Warteschlange oder Stack (Stapel) mit Prioritätswert
- stellt Funktionen *isEmpty*, *Insert* und *extractMin/Max* zur Verfügung.
- Heaps -> implementieren -> Priority Queue
 - *merge*, *heapifyUp / Down*
- Vorteile:
 - *Binär Heap -> Element mit höchster Prio. immer in der Wurzel*
 - *Binomial Heap -> mergen geht sehr schnell ($O(\log n)$)*

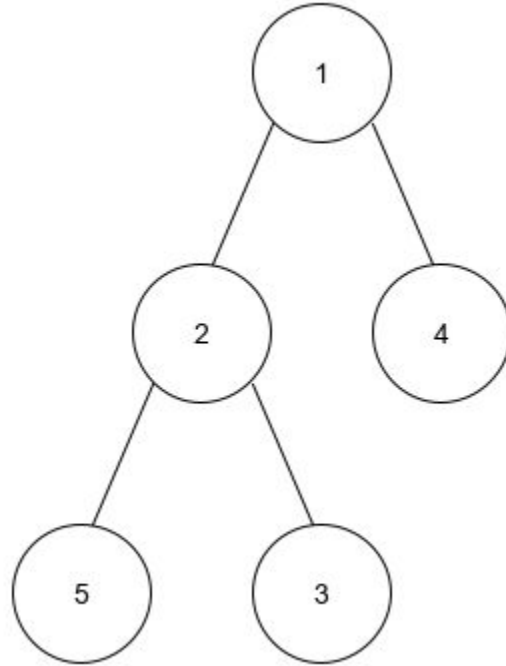
Anwendungsbeispiel (Organspende)



Binär Heaps

Insert()

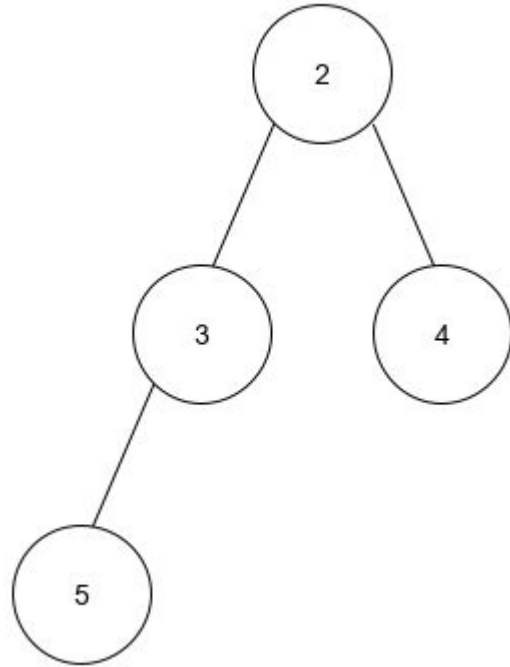
heapifyUp()



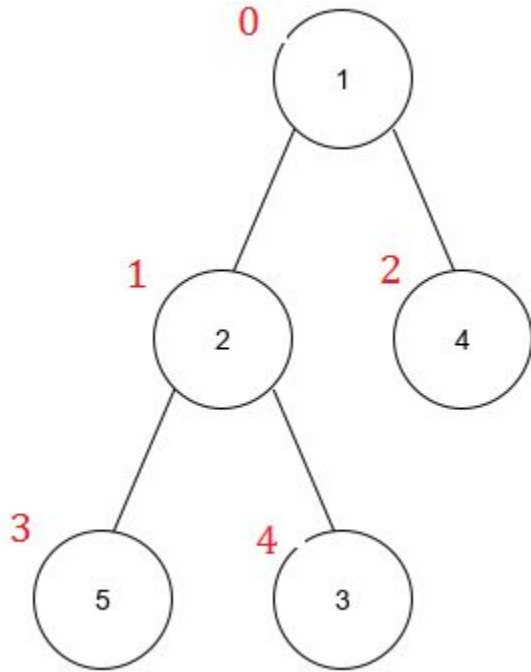
Binär Heaps

extractMin() / poll

heapifyDown()



Binär Heaps



Index	0	1	2	3	4
Prio	1	2	4	5	3

Linke Child:

$$i = i * 2 + 1$$

Rechte Child:

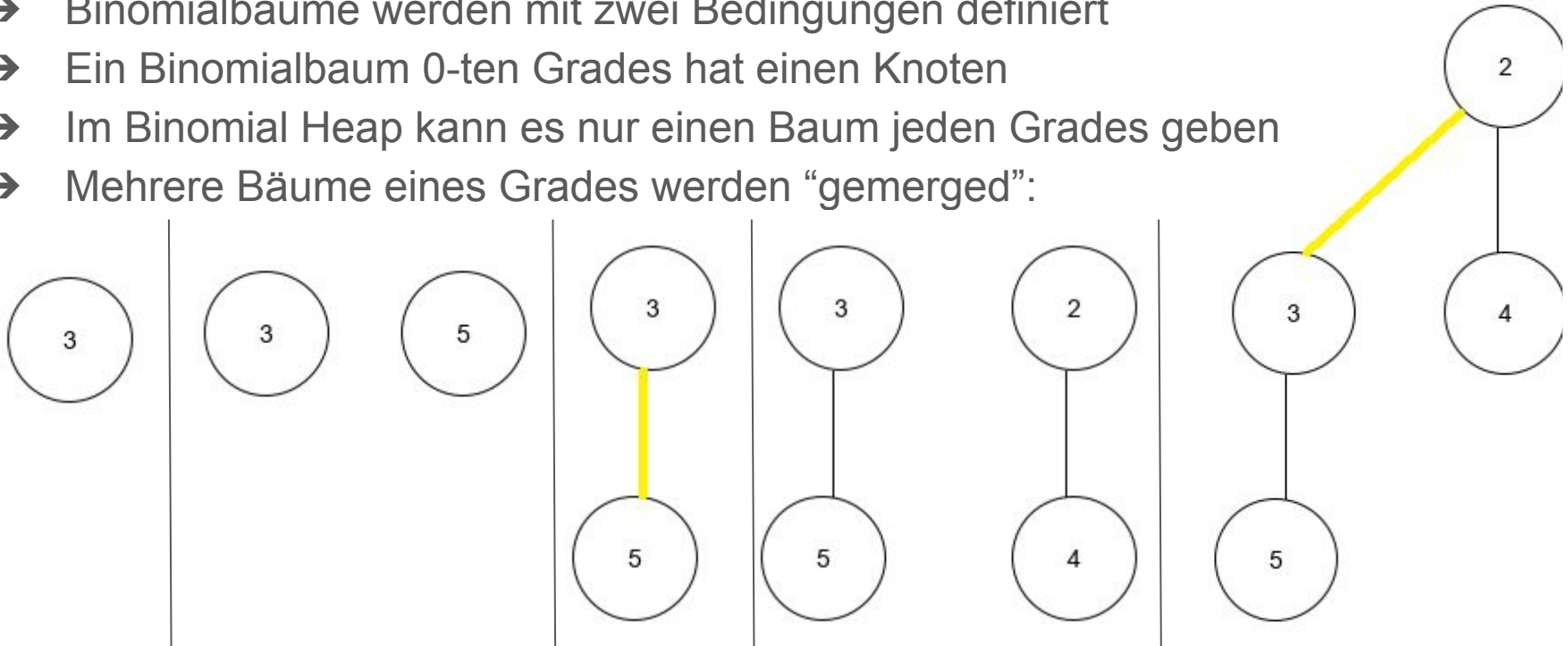
$$i = i * 2 + 2$$

Elternelement:

$$i = \text{floor}((i - 1) / 2)$$

Binomial Heaps

- Binomial Heaps beschreibt man als die Menge verbundener Binomialbäume
- Binomialbäume werden mit zwei Bedingungen definiert
- Ein Binomialbaum 0-ten Grades hat einen Knoten
- Im Binomial Heap kann es nur einen Baum jeden Grades geben
- Mehrere Bäume eines Grades werden “gemerged”:



Binomial Heaps

- Beim entfernen eines Knotens, entstehen Bäume aus den restlichen Teilbäumen
- Das entfernen erfolgt mit der Operation ExtractMin / poll
- Binomial Heaps lassen sich auch binär darstellen
 - ◆ Dafür stelle man die Anzahl der Knoten als Dualzahl da

Laufzeiten

Operation	Binär	Binomial
insert	$O(\log n)$	$O(\log n)$
peek	$O(1)$	$O(\log n)$
extractMin	$O(\log n)$	$O(\log n)$
merge	$O(n)$	$O(\log n)$

Implementierung

- Kotlin
- Grafische Darstellung mit LibGDX
- Binary Heap
- Binominal Heap

Klasse: Node

- Priority
- posX, posY (grafische Oberfläche)
- ggbf. Payload
- updatePosition(), draw(), drawText() (grafische Oberfläche)

Klasse: BinaryHeap

- heap : mutableListOf<Node>()
- addNode()

```
fun addNode(priority : Int){  
    // neuer Knoten wird dem Heap hinzugefügt  
    heap.add(Node(priority))  
    heapifyUp()  
}
```

- poll()

```
fun poll() : Node {  
    // liefert und entfernt die Wurzel des Baums  
    if (heap.size == 0) throw IllegalStateException()  
    val item : Node = heap[0]  
    heap[0] = heap[heap.size - 1]  
    heapifyDown()  
    return item  
}
```

Klasse: BinaryHeap

- peek()

```
fun peek() : Node {                                // liefert die Wurzel des Baums
    if (heap.size == 0) throw IllegalStateException()
    return heap[0]
}
```

- heapifyUp()

```
fun heapifyUp() {                                   // Sortieren des letzten Knotens nach oben
    var index = heap.size - 1
    while (hasParent(index) && parent(index) > heap[index].priority) {
        swap(getParentIndex(index), index)
        index = getParentIndex(index)
    }
}
```


Klasse: BinaryHeap

- heapifyDown()

```
fun heapifyDown() {                                // Sortieren des ersten Knotens (Wurzel) nach unten
    var index = 0
    while (hasLeftChild(index)) {
        var smallerChildIndex : Int = getLeftChildIndex(index)
        if (hasRightChild(index) && rightChild(index) < leftChild(index)) {
            smallerChildIndex = getRightChildIndex(index)
        }

        if (heap[index].priority < heap[smallerChildIndex].priority) {
            break
        } else {
            swap(index, smallerChildIndex)
        }
        index = smallerChildIndex
    }
}
```

Klasse: BinominalTree(s)

- Oberklasse Tree -> rootNode : Node, degree : Int
- Unterklasse BinominalTreeDZero -> node : Node, degree : Int = 0
- Unterklasse BinominalTree -> leftTree : Tree, rightTree : Tree

```
//Oberklasse mit der Priorität der Root Node und dem Degree des Baums
open class Tree(var rootNode: Node, val degree : Int)

//Spezielle Klasse für den Baum mit Grad 0, weil dieser als einziger eine Node enthalten muss
class BinominalTreeDZero(var node : Node, degree: Int = 0) : Tree(node, degree)

//Normaler (Teil)Baum für den Heap
open class BinominalTree(rootNode: Node, degree : Int, var leftTree : Tree, var rightTree : Tree) : Tree(rootNode, degree){
}
}
```

Klasse: BinominalHeap

- heap : mutableListOf<Tree>()
- lines : mutableListOf<Vector2>(), nodeCount : Int (grafische Oberfläche)
- addNode()

```
//Hinzufügen einer Node bzw. eines Trees vom Grad 0
open fun addNode(priority : Int){
    heap.add(BinominalTreeDZero(Node(priority)) )
    //Korrigieren des Baums
    correctTree()
    nodeCount++
}
```

Klasse: BinominalHeap

- correctTree()

```
//Sorgt dafür, das der Heap nur einen Baum des jeweiligen Grades hat
open fun correctTree(){
    var correction = true
    while(correction){
        correction = false
        for(t in heap){
            for(t2 in heap){
                //kontrolliert ob die iterierten Bäume der selbe sind
                if(t.degree == t2.degree && t != t2){
                    //Mergen der Bäume
                    heap.add(merge(t.degree, t, t2))
                    heap.remove(t)
                    heap.remove(t2)
                    correction = true
                    Gdx.app.log( tag: "Debug", message: "Correction!")
                    break
                }
            }
        }
    }
}
```

Klasse: BinominalHeap

- merge()

```
//Mergen von zwei Trees bei dem der kleinere links eingefügt wird
open fun merge(degree : Int, tree1 : Tree, tree2 : Tree) : BinominalTree{
    return if (tree1.rootNode.priority <= tree2.rootNode.priority) BinominalTree(tree1.rootNode, degree: degree+1, tree1, tree2)
    else BinominalTree(tree2.rootNode, degree: degree+1, tree2, tree1)
}
```

Klasse: BinominalHeap

- poll()

```
//Entnehmen der kleinsten Node-
open fun poll() : Node{
    var smallest = Int.MAX_VALUE
    lateinit var tree : Tree

    //herausfinden des kleinsten Wertes der Trees im Heap
    for(t in heap){
        if(t.rootNode.priority < smallest){
            smallest = t.rootNode.priority
            tree = t
        }
    }
    nodeCount--
    return split(tree)
}
```

Klasse: BinominalHeap

- split()

```
//findet die Node mit dem kleinsten Wertes innerhalb des Trees (Root) und fügt die Teilbäume dem Heap hinzu
open fun split(tree : Tree) : Node{
    lateinit var node : Node
    //falls der Wert der Root ist
    if(tree is BinominalTreeDZero){
        node = tree.node
        heap.remove(tree)
        return node
    }
    //removen des zu bearbeiteten Trees, adden des Teilbaums zum Heap und Rekursion für die kleinere Hälfte
    else if(tree is BinominalTree){
        heap.add(tree.rightTree)
        node = split(tree.leftTree)
        heap.remove(tree)
    }
    return node
}
```

Fragen ???