Algorithmik-Praktikum

Informatik

Wintersemester 2019/2020

Binär- Heaps und Binomial-Heaps

Teilnehmer (Luca Stamos, Stefan Steinhauer, Dimitri Osokin, Marc Kevin Zenzen)

Inhaltsverzeichnis

Einleitung

Binär- und Binomial Heaps sind Datenstrukturen, welche den abstrakten Datentyp Prioritätswarteschlangen implementieren. Das bedeutet, dass jedem einzelnen Element des Heaps eine Priorität zugewiesen wird. Im nachfolgenden werden wir das notwendige Grundwissen behandeln und uns anschließend tiefer mit den einzelnen Teilthemen beschäftigen. Dabei gehen wir auf die verschiedenen Operationen der Heaps ein, wie z.B. Insert (Element hinzufügen), ExtractMin (entnimmt das Element mit dem niedrigsten Schlüssel), HeapifyUp (sortiert ein Element von unten nach oben in den Tree ein)/HeapifyDown (sortiert ein Element von oben nach unten in den Tree ein) oder Merge (Teilbäume zusammenfügen). Die Laufzeit der Heaps spielt hierbei auch eine entscheidende Rolle, da die Effektivität des Heaps unter anderem durch sie bestimmt wird. Zuletzt überlegen wir, wie wir die Datenstrukturen in einem praxisrelevanten Beispiel implementieren können.

Recherche / Grundlagen

Bevor wir auf die Binär-Heaps und Binomial-Heaps zu sprechen kommen, gehen wir auf den abstrakten Datentyp “Priority Queue” und Baumstrukturen ein, da diese die Grundlagen für Heaps darstellen.

Erstmalig wurden Priority Queues und dessen Implementierung in Heaps von im Jahr 1964 von Williams [2] in seiner Publikation “Algorythm 232” eingeführt. Im gleichen Jahr wurde der Algorithmus von Floyd [3] verbessert indem er vorschlug eine BUILD-MAX-HEAP Methode hinzuzufügen. 1978 beschrieb Jean Vuillemin [4] erstmals die Binomial-Heaps.

Heaps bieten sich immer dann an, wenn man es mit Prioritätswarteschlangen (“Priority Queues”) zu tun hat, wie sie bei Servern oder Betriebssystemen zum Einsatz kommen um die Ausführungsreihenfolge von Aufgaben festzulegen. Heaps stellen außerdem eine ideale Datenstruktur für Algorithmen dar, die schrittweise lokale optimierte Entscheidungen treffen (“Greedy Algorithmen”). Ein Beispiel dafür ist das Sortierverfahren “Heapsort” bei dem ein binärer Heap zum sortieren eingesetzt wird.

Definition der Lernziele

Sie verstehen den Aufbau von binären und binomialen Bäumen und deren Unterschiede.
Sie verstehen den Sinn von Heap-Strukturen und dessen Einsatz in der Programmierwelt.
Sie können die verschiedenen Funktionen (Operationen) eines Heaps verstehen und

implementieren.
Sie kennen die Laufzeitunterschiede zwischen den aufgeführten Heap-Varianten.

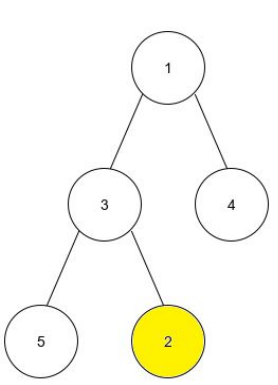
Thema

In folgendem Abschnitt gehen wir genauer auf die beiden Themen „Binär-Heaps“ und „Binomial-Heaps“ ein. Wir werden hier genauer auf die Verwendung der Prioritätswarteschlangen und auf die Unterschiede zwischen den Heaps eingehen. Wo die Vor- und Nachteile der Heaps liegen oder wo Sie sich zu anderen Sortieralgorithmen unterscheiden.
Die binär Heaps und binomial Heaps implementieren beide die abstrakte Klasse “Priority Queues”, welche die im späteren Verlauf verwendeten Funktionen zur Verfügung stellt.

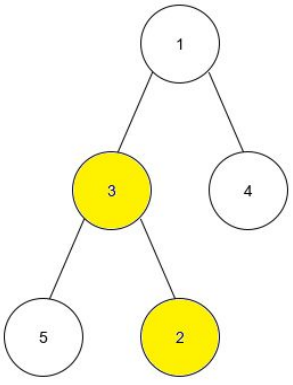
Binär-Heaps:

Die Datenstruktur des binären Heaps versteht man als einen fast vollständigen binären Baum. Ein Knoten eines binären Baums hat maximal zwei nachfolgende Knoten (Child). Durch die Verwendung der Prioritätswarteschlange (Priority Queues) werden die Elemente in einem binären Baum nach Priorität sortiert. Der Wert mit der höchsten Priorität liegt immer in der Wurzel, bei einem MinHeap ist dies der kleinste Wert und entsprechend im MaxHeap der größte. Darauf folgend werden alle Knoten von der Wurzel an mit einem Index *i* beschriftet (von oben nach unten und von links nach rechts), was uns im späteren Verlauf den Zugriff auf ein Childelement ermöglicht.

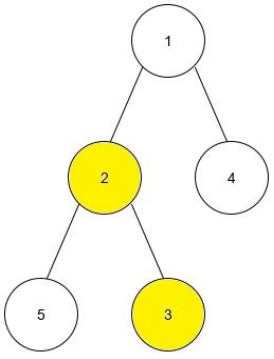
Im nachfolgenden gehen wir von einem MinHeap aus, d.h. es befindet sich immer das Element mit der höhste Priorität und dem kleinsten Wert (Key) in der Wurzel.
Beim einfügen (*insert*) eines neuen Elements in den bestehenden Baum wird dieses immer unten links angehangen [Abbildung 1]. Anschließend wird es mit dem Elternelement (Parent) verglichen [Abbildung 2]. Im Fall, dass das Elternelement größer ist, werden diese getauscht (swap) [Abbildung 3]. Dieser Vorgang wird solange wiederholt, bis das Element an der richtigen Stelle im Baum liegt. Diesen Prozess nennt man *heapifyUp*.



[Abbildung 1]



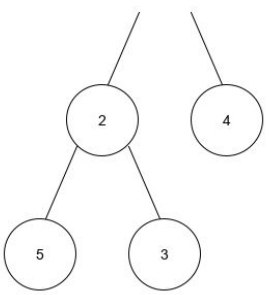
[Abbildung 2]



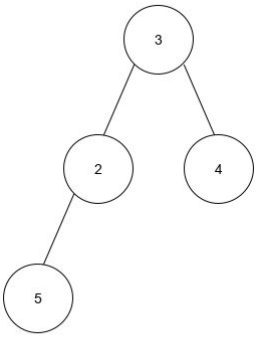
[Abbildung 3]

Die Operation *extractMin* entnimmt das Element aus der Wurzel [Abbildung 4] und liefert dieses zurück. Nachdem es entfernt wurde, wird das Element mit dem größten Index in die Wurzel verschoben [Abbildung 5]. Anschließend wird es mit den zwei Childelementen verglichen und mit dem kleineren Childelement getauscht [Abbildung 6, 7]. Dies wird

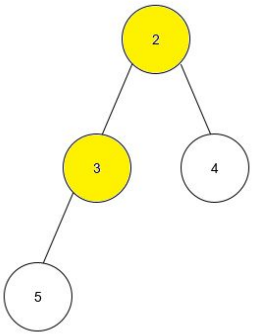
solange wiederholt, bis das Element an der richtigen Stelle ist. Diesen Vorgang nennt man *heapifyDown*, da das Element in dem Baum nach unten “wandert”.



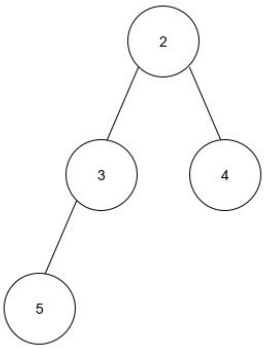
[Abbildung 4]



[Abbildung 5]

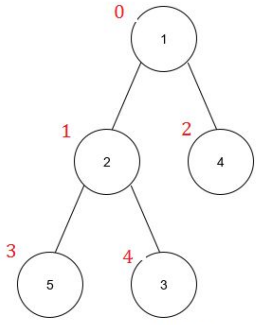


[Abbildung 6]



[Abbildung 7]

Um den sortierten Baum als Heap in einem Array zu speichern, wurden zuvor alle Knoten mit Indizes *i* beschriftet [Abbildung 8]. Diese werden entsprechend der Indizes in einem Array angeordnet [Abbildung 9]. Dies ermöglicht mit der Formel *i = i * 2 + 1* den Zugriff auf das linke Childelement, mit *i = i * 2 + 2* kommt man auf das rechte Childelement. Um von einem Childelement auf das dazugehörige Elternelement zu kommen benutzt man i = (i - 1) / 2.



[Abbildung 8]

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

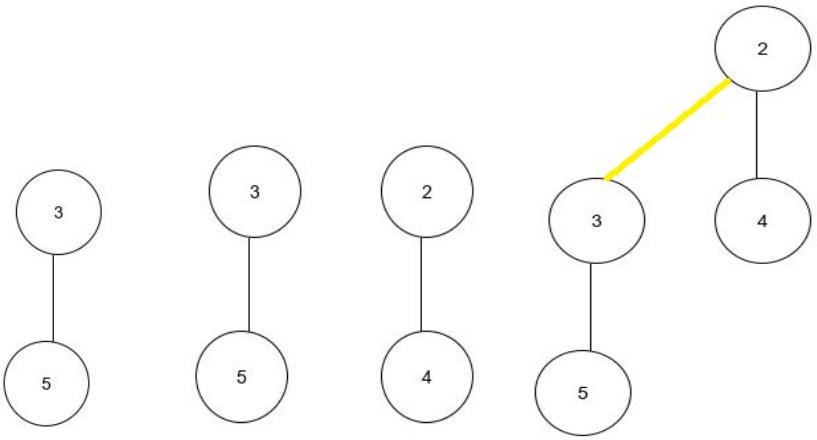
[Abbildung 9]

<u>Binomial-Heaps:</u>

Binomial Heaps beschreibt man als die Menge von Binomialbäumen, deren Wurzeln miteinander linear verknüpft sind. Binomialbäume definiert man mit zwei Bedingungen. Die erste Bedingung sagt aus, dass ein Baum mit genau einem Knoten als ein Baum 0-ten Grades mit der Tiefe Null beschrieben wird. Die zweite Bedingung sagt aus, dass Binomialbäume mit einem Grad höher als Null aus zwei Teilbäumen selben Grades niedrigerer Tiefe zusammengesetzt sind. Diese werden durch die Operation “merge” miteinander verbunden, wobei ein Teilbaum als der linkeste Nachfolger des anderen Teilbaumes zur Wurzel verknüpft wird, wodurch ein Binomialbaum höheren Grades entsteht.

Im Binomial Heap kann von jedem Grad nur ein Baum existieren. Ist also ein Binomialbaum ersten Grades gegeben [Abbildung 10], kann dieser als Heap dargestellt werden. Sobald ein weiterer Binomialbaum selben Grades hinzu kommt [Abbildung 11], muss die Merge Operation angewendet werden. Bei dem Merge von zwei Binomialbäumen wird die Wurzel

mit der niedrigeren Priorität, als linkes Childelement, der Wurzel mit der höheren Priorität untergeordnet [Abbildung 12]. Entsteht aus dem Merge ein Baum dessen Grad schon vorhanden ist, wird dieser Vorgang wiederholt bis die Integrität des Heaps wiederhergestellt ist. Im Falle eines Min-Heaps wird die höchste Priorität in der Wurzel, durch den niedrigsten Key-Wert angegeben, während bei einem Max-Heap der höchste Key-Wert in der Wurzel steht.



[Abbildung 10] [Abbildung 11] [Abbildung 12]

Binomial Heaps kann man auch binär darstellen, da die Menge der Knoten sich in binären Werten wieder geben lassen, anhand derer auch jede Tiefe und Anzahl der vorhandenen Binomialbäume ablesbar wäre. Somit wäre der Baum in Abbildung 12 in binär als 0100 dargestellt. Es ist ein einzelner Baum 3ten Grades, mit der Anzahl von 4 Knoten.

Der Binominal Heap enthält genau wie der Binär Heap die Funktionen Insert und Poll (*Extract_Min*). Um einen Insert durch zu führen wird ein Baum des Grades 0 dem Heap hinzugefügt. Für einen Poll muss die Wurzel des Baumes, dessen Wurzel die geringste Priorität hat entfernt werden. Die hinterbliebenen Teil-Bäume werden danach dem Heap hinzugefügt. Nach Beiden dieser Vorgänge muss die Integrität des Heaps, durch Mergen der doppelten Bäume, wiederhergestellt werden.

<u>Laufzeitvergleich:</u>

Im Vergleich zu Anderen Suchalgorithmen die nicht auf Basis einer Priority funktionieren, schwankt die Laufzeit des Heapsorts geringer. Dies liegt daran, dass andere Suchalgorithmen möglicherweise mit Keys arbeiten müssen die keine Zahlen sind, was die Laufzeit drastisch anhebt.

Die Laufzeit von Binär- und Binomialheaps verhält sich grundsätzlich gleich mit O(log n). Da ein Binomialheap aus mehreren Bäumen bestehen kann, ergibt sich bei *extractMin* eine höhere Laufzeit mit O(lg n) im Gegensatz zu O(1) im binären Heap, da hier nur die Wurzel genommen werden muss.

Pseudocode ausgehend von Max.Heap nach Cormen [1]:

Im folgenden Abschnitt werden die Laufzeiten der nötigen Funktionen anhand von Pseudocode nach Cormen [1] gezeigt.

Max-Heap-Insert, Heap-Extract-Max, Heap-Increase-Key und Heap-Maximum ←- O(lg n)
// Heap als 'Priority Queue' verwenden

Max-Heapify(A,*i*) ←- O(lg n) //Aufrechterhaltung der Heap-Eigenschaft
Laufzeit kann rekursiv beschrieben werden mit T(n) <= T(2n/3) + O(1) ----> O(lg n)

```
l = links(i)
r = rechts(i)
if l<=A.heap-größe und A[l] > A[i]
        maximum = l
else
        maximum = i
if r<= A.heap-größe und A[r] > A[maximum]
        maximum = r
if maximum != i
        vertausche A[i] mit A[maximum]
        Max-Heapify(A, maximum)
```

Build-Max-Heap(A) ←- O(n) //erzeugt einen Max-Heap aus einem ungeordneten Eingabefeld A

```
A.heap-größe = A.länge
for i = [A.länge / 2] downto 1
        do Max-Heapify(A, i)
```

Max-Heap-Insert(A, *schlüssel*) ←- O(lg n) bei n Elementen

```
A.heap-größe = A.heap-größe + 1
A[A.heap-größe] = new_Node (schlüssel)
```

Heap-Extract-Max(A)←- O(lg n)

if A.*heap-größe* < 1
    error “Heap-Unterlauf”
max = A[1]
A[1] = A[A.*heap-größe*]
A.*heap-größe* = A.*heap-größe* -1
Max-Heapify(A, 1)
return *max*

Heapsort(A) ←- O(n lg n) // (n-1) * Max-Heapify (O(lg n)) [mit Build-Max-Heap (O(n))]
// sortiert ein Feld in-place

build-Max-Heap(A)
for *i* = *A.länge* **downto** 2
    vertausche A[1] mit A[*i*]
    A.heap-größe = A.heap-größe -1
    Max-Heapify(A,1)

Implementierung

Für die Implementierung der Binär- und Binominal Heaps haben wir die Programmiersprache Kotlin gewählt, mit der Bibliothek LibGDX um eine Grafische Darstellung zu ermöglichen.

Datenstruktur Binary-Heap:
Die Basis des Binary Trees bildet die Klasse Node.

```
open class Node(var priority : Int, var posx : Float = 0f, var posy : Float = 0f){
```

Hierbei bilden posx und posy zusammen die Position der Node auf dem Bildschirm für die Grafische Darstellung.

Eine Liste mit allen Nodes wird in der Oberklasse des Binary Heaps gespeichert.

```
class BinaryHeap(){

    var heap = mutableListOf<Node>()
```

In dieser Implementierung haben wir uns für einen Array ähnlichen Ansatz anstatt eines Objektorientierten für den Binary Tree entschieden.
Für diesen Ansatz sind einige Funktionen nötig um von einem Node bzw. des Indexes dessen auf die Parent- und Child Nodes zu schließen.

```
fun getLeftChildIndex(index : Int) = 2 * index + 1                    // liefert den linken Child-Index der Node
fun getRightChildIndex(index : Int) = 2 * index + 2                   // liefert den rechten Child-Index der Node
fun getParentIndex(childIndex: Int) = (childIndex - 1) / 2            // liefert den Parent-Index der Node

fun hasLeftChild(index : Int) = getLeftChildIndex(index) < heap.size  // liefert True wenn linkes Child vorhanden
fun hasRightChild(index : Int) = getRightChildIndex(index) < heap.size // liefert True wenn rechtes Child vorhanden
fun hasParent(index: Int) : Boolean = getParentIndex(index) >= 0      // liefert True wenn Parent vorhanden

fun getLeftChildX(index : Int) = heap[getLeftChildIndex(index)].posx  // liefert die X Position des linkes Childs
fun getLeftChildY(index : Int) = heap[getLeftChildIndex(index)].posy  // liefert die Y Position des linkes Childs

fun getRightChildX(index : Int) = heap[getRightChildIndex(index)].posx // liefert die X Position des rechten Childs
fun getRightChildY(index : Int) = heap[getRightChildIndex(index)].posy // liefert die Y Position des rechten Childs

fun leftChild(index: Int) : Int = heap[getLeftChildIndex(index)].priority // liefert die Priority des linken Childs
fun rightChild(index: Int) : Int = heap[getRightChildIndex(index)].priority//liefert die Priority des rechten Childs
fun parent(index: Int) : Int = heap[getParentIndex(index)].priority   // liefert die Priority des Parent Node
```

Nodes können mit einer *Add* Funktion hinzugefügt werden.

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

Mit einer *Poll* Funktion kann die Node mit dem geringsten Wert, also der Root entnommen werden. *Peek* gibt auch den Root zurück aber entfernt ihn nicht.

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


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

}
```

Nach solchen Funktionen die die Integrität des Binary Heaps schädigen muss entsprechend die HeapifyUp- oder HeapifyDown Funktion aufgerufen werden um die Integrität wieder herzustellen.

```
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)
    }

}


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

    }


}
```

Datenstruktur Binominal-Heap:

Für die Implementation des Binominal Heaps benutzen wir als Basis dieselbe Node Klasse die wir auch für den Binary Heap benutzt haben.
Desweiteren haben wir uns für eine Implementation entschieden, die Teilbäume benutzt um den Binominal Tree zu formen. Jeder Baum hat einige Grundattribute gemeinsam, allerdings

unterscheiden wir zwischen Bäumen des 0-ten Grades und allen darüber, weil nur dieser eine Node als Payload braucht. Die Restlichen Teil-Bäume sorgen für die Struktur des Baums.

```
//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){

}
```

Die Oberklasse enthält hier wieder eine Liste mit vollständigen Bäumen und die Anzahl aller Nodes in Diesen. Die Liste ‘Lines’ enthält alle zu zeichnenden Linien für die grafische Darstellung.

```
class BinominalHeap (){
    //Die Liste alle Trees im Heap
    var heap = mutableListOf<Tree>()
    var lines = mutableListOf<Vector2>()
    var nodeCount = 0
```

Auch für den Binominal Heap haben wir eine *Add* und *Poll* Funktion eingebaut.
Für die *Poll* Funktion wird eine *Split* Funktion benötigt die den Root eines Baumes entfernt und die übriggebliebenen Teilbäume dem Heap hinzufügt.

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



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

    //herrausfinden 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)
}
```

```
//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
}
```

Da diese Funktionen auch hier die Integrität des Heaps schädigen muss in diesem Fall die Funktion *CorrectTree* ausgeführt werden. Diese Funktion prüft, ob zwei Bäume des gleichen Grades im Heap sind, wenn ja, werden diese durch die Funktion *merge* verbunden und der neue Baum dem Heap hinzugefügt. Dies wird so lange wiederholt bis nur noch ein Baum von jedem Grad vorhanden ist.

```
//Funktion zum Kontrollieren, ob der Heap von jedem Baum Grad nur einen hat
open fun isCorrect() : Boolean{
    var correct = true
    for(t in heap){
        for(t2 in heap){
            if(t.degree == t2.degree && !t.equals(t2)) correct = false
        }
    }
    return correct
}

//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
                }
            }
        }
    }
}
```

```
//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)
}
```

Kurzfassung der Grafischen Oberfläche:

Da der Fokus dieser Dokumentation nicht auf der Grafischen Oberfläche liegt, folgt hier nur eine Kurzfassung dieser.

Die Klasse Node enthält sowohl die Position wo sie gezeichnet werden muss, als auch die Größe des zu zeichnenden Kreises. Zudem wird in jeden Kreis seine Priorität reingeschrieben, die zuvor zufällig zwischen 0 und 100 bestimmt wurde.

Die einzelnen Positionen der Nodes werden rechnerisch bestimmt und müssen nach einem *Add* oder *Poll* erneut durch eine Update Funktion berechnet werden.

Zeichnen des Binary Heaps:

```
fun drawAll(shapeRenderer: ShapeRenderer){                              // zeichnet alle Knoten
    shapeRenderer.setColor(Color.RED)
    for(n in heap){
        n.draw(shapeRenderer)
    }
}

fun drawAllText(batch: SpriteBatch, font: BitmapFont, layout: GlyphLayout){  // zeichnet die Werte aller Knoten
    for(n in heap){
        n.drawText(batch, font, layout)
    }
}

fun drawAllLines(shapeRenderer: ShapeRenderer){                       // zeichnet die Linien zwischen den Knoten
    shapeRenderer.setColor(Color.WHITE)

    for(i in 0 until heap.size){
        if(hasLeftChild(i)) shapeRenderer.rectLine(heap[i].posx, heap[i].posy, getLeftChildX(i), getLeftChildY(i),  width: 10f)
        if(hasRightChild(i)) shapeRenderer.rectLine(heap[i].posx, heap[i].posy, getRightChildX(i), getRightChildY(i),  width: 10f)
    }
}

fun updateAllNodes(){                                 // legt die Position aller Knoten auf dem Bildschirm fest
    for(i in 0 until heap.size){
        heap[i].updatePosition(i, heap.size)
    }
}


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

```
    //Zeichnen des Kreises
    open fun draw(shapeRenderer: ShapeRenderer){
        shapeRenderer.circle(posx, posy, size)
    }

    open fun drawText(batch : SpriteBatch, font : BitmapFont, layout : GlyphLayout){
        layout.setText(font,  str: "" + priority)
        font.setColor(Color.WHITE)
        font.draw(batch,  str: "" + priority,  x: posx-layout.width/2,  y: posy+layout.height/2)
    }

    //Updaten der Position der Node
    open fun updatePosition(index : Int, listSize : Int){
        var row = MathUtils.floor(MathUtils.log( a: 2f, (index + 1).toFloat()))
        var column = (index + 1) - 2.toDouble().pow(row.toDouble())

        var maxRow = MathUtils.floor(MathUtils.log( a: 2f, listSize.toFloat())) + 1
        var maxColumn = 2.toDouble().pow(row)

        this.posx = ((Main.VIEWPORT_WIDTH / (maxColumn+1)) * (column+1)).toFloat()
        this.posy = Main.VIEWPORT_HEIGHT - (Main.VIEWPORT_HEIGHT / (maxRow+1) * (row+1))
    }

}
```

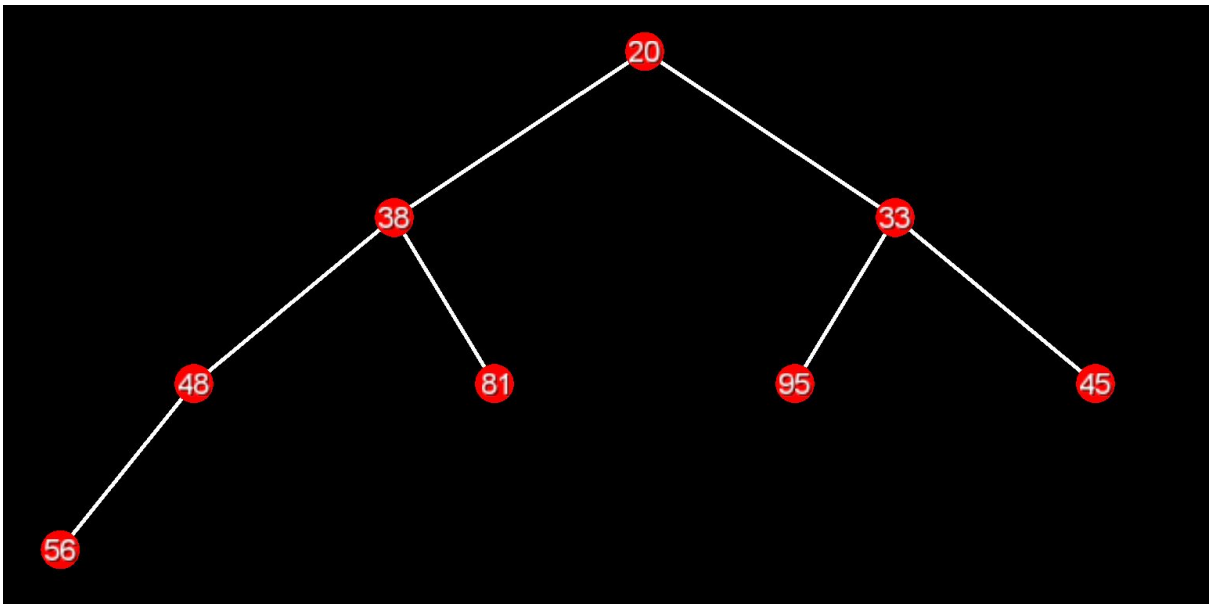
Zeichnen des Binominal Heaps:

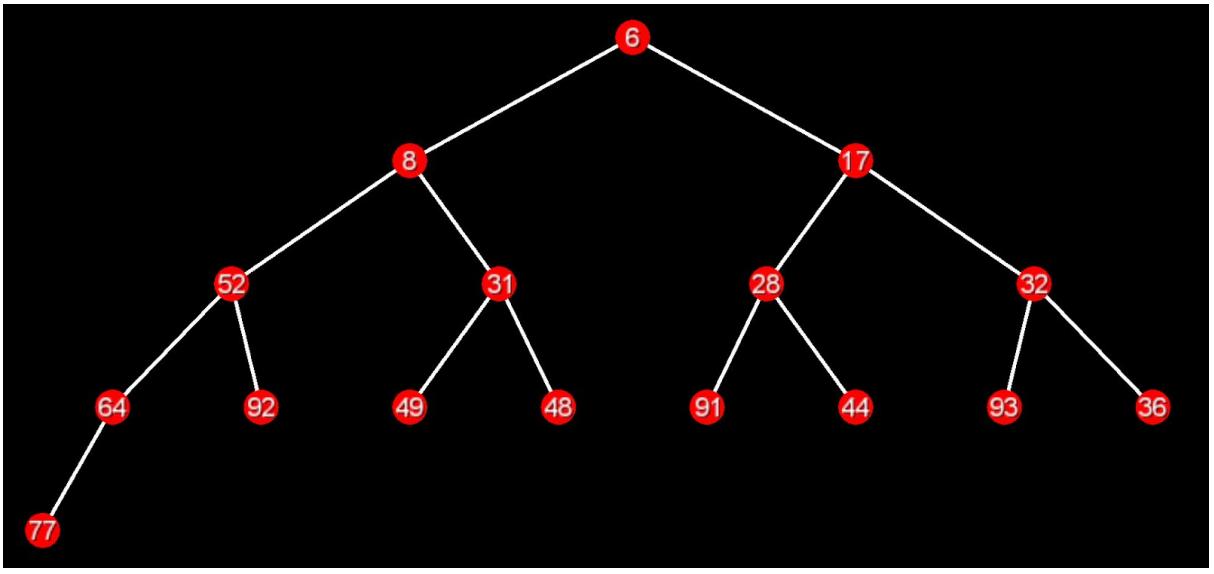
```
//Legt die Position aller Nodes auf dem Bildschirm fest
open fun updateAllNodes(){
    var width = (2.toDouble().pow((Integer.toBinaryString(nodeCount).length - 1).toDouble())).toInt()

    for(i in 0 until heap.size){
        if(heap[i] is BinominalTree)updateTree( depthX: 0,  depthY: 0, heap[i] as BinominalTree,  offset: width - 2.toDouble().pow(heap[i].degree).toInt())
        else if(heap[i] is BinominalTreeDZero)updateTree( depthX: 0,  depthY: 0, heap[i] as BinominalTreeDZero,  offset: width - 2.toDouble().pow(heap[i].degree).toInt())
    }
}

//Funktion die Rekursiv ausgeführt wird, für den Fall, das der Übergebene Tree ein Binominal Tree ist
//Zusätzlich hinzufügen der Linien zwischen zwei Teilbäumen des Grades 0
open fun updateTree(depthX : Int, depthY : Int, tree : BinominalTree, offset : Int){
    if(tree.leftTree is BinominalTreeDZero && tree.rightTree is BinominalTreeDZero){
        var maxYDepth = Integer.toBinaryString(nodeCount).length+1
        var posx = (Main.VIEWPORT_WIDTH / (2.toDouble().pow((Integer.toBinaryString(nodeCount).length - 1).toDouble()) + 1) * (depthX+offset+1)).toFloat()

        (tree.rightTree as BinominalTreeDZero).node.posx = Main.VIEWPORT_WIDTH - posx
        (tree.rightTree as BinominalTreeDZero).node.posy = (Main.VIEWPORT_HEIGHT / maxYDepth * (depthY+1))

        (tree.leftTree as BinominalTreeDZero).node.posx = Main.VIEWPORT_WIDTH - posx
        (tree.leftTree as BinominalTreeDZero).node.posy = (Main.VIEWPORT_HEIGHT / maxYDepth * (depthY+2))

        lines.add(Vector2((tree.rightTree as BinominalTreeDZero).node.posx, (tree.rightTree as BinominalTreeDZero).node.posy))
        lines.add(Vector2((tree.leftTree as BinominalTreeDZero).node.posx, (tree.leftTree as BinominalTreeDZero).node.posy))
    } else if(tree.leftTree is BinominalTree && tree.rightTree is BinominalTree){
        updateTree( depthX: depthX*2, depthY, tree.rightTree as BinominalTree, offset)
        updateTree( depthX: depthX*2+1,  depthY: depthY+1, tree.leftTree as BinominalTree, offset)
    }
}

//Funktion die Rekursiv ausgeführt wird, für den Fall, das der Übergebene Tree ein Binominal Tree 0ten Grades ist
open fun updateTree(depthX : Int, depthY : Int, tree : BinominalTreeDZero, offset : Int){
    var maxYDepth = Integer.toBinaryString(nodeCount).length+1
    var posx = (Main.VIEWPORT_WIDTH / (2.toDouble().pow((Integer.toBinaryString(nodeCount).length - 1).toDouble()) + 1) * (depthX+offset+1)).toFloat()

    tree.node.posx = Main.VIEWPORT_WIDTH - posx
    tree.node.posy = (Main.VIEWPORT_HEIGHT / maxYDepth * (depthY+1))
}
```

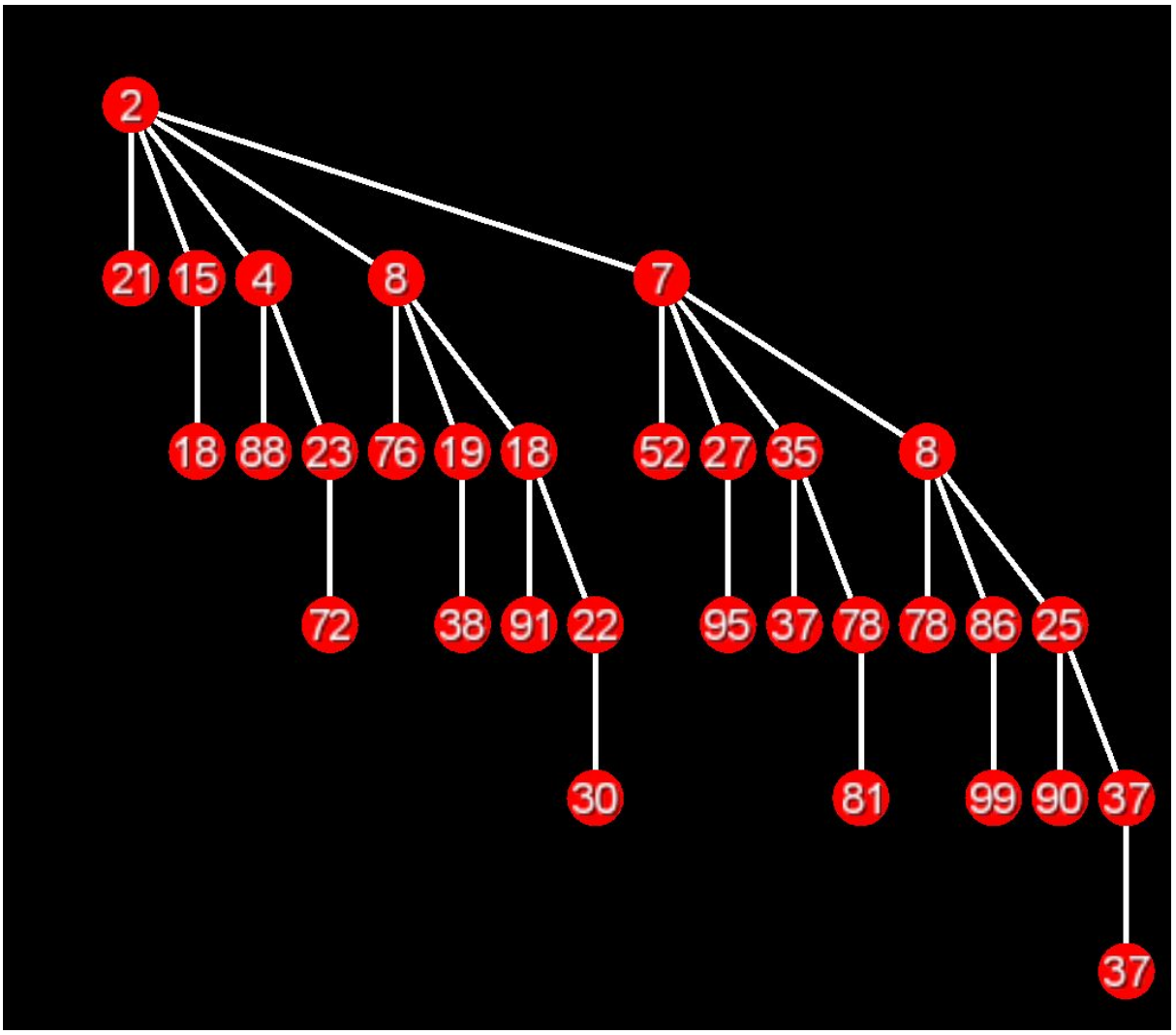
```
//Zeichnet alle Linien auf dem Bilschirm
open fun drawAllLines(shapeRenderer: ShapeRenderer){
    shapeRenderer.setColor(Color.WHITE)
    for(i in 0 until lines.size/2){
        shapeRenderer.rectLine(lines[i*2].x, lines[i*2].y, lines[i*2+1].x, lines[i*2+1].y,  width: 10f)
    }
}

//Fügt alle notwendigen Linien der Liste aller Linien hinzu
open fun addAllLines(){
    for(t in heap){
        if(t is BinominalTree)addLines(t)
    }
}

//Rekursive Funktion zum hinzufügen der Linien zwischen den Teilbäumen
open fun addLines(tree : BinominalTree){
    if(tree.leftTree is BinominalTree && tree.rightTree is BinominalTree){
        lines.add(Vector2(tree.rootNode.posx, tree.rootNode.posy))
        lines.add(Vector2(tree.rightTree.rootNode.posx, tree.rightTree.rootNode.posy))

        addLines(tree.leftTree as BinominalTree)
        addLines(tree.rightTree as BinominalTree)
    }
}
```

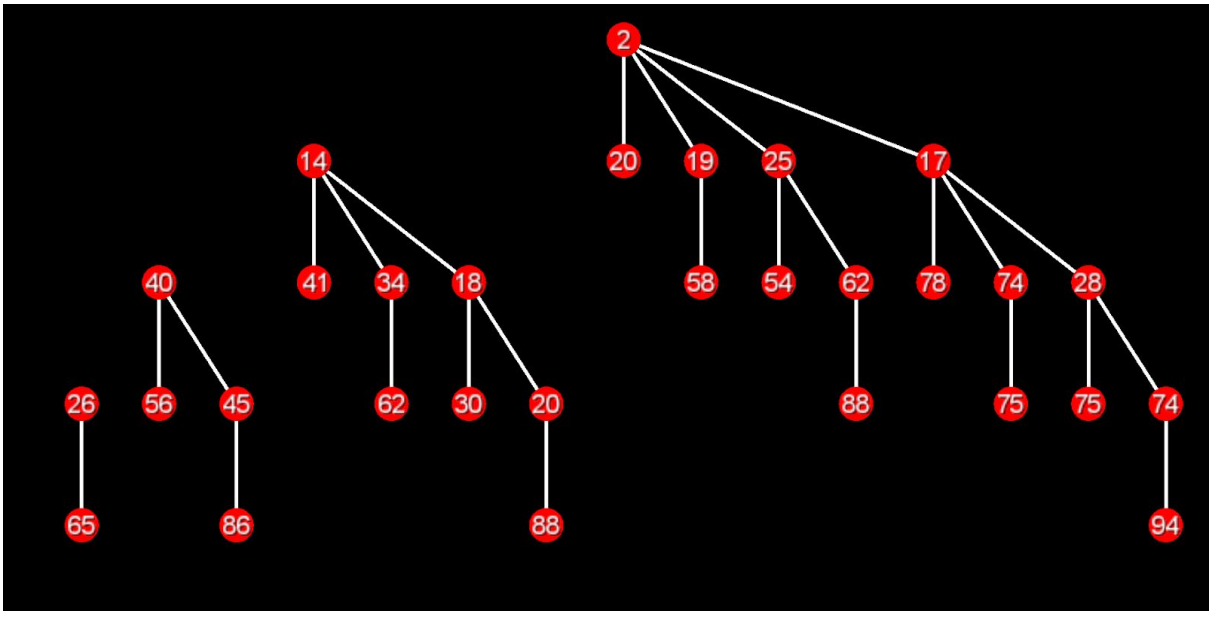
```
//Für Kontrollzwecke, zeichnet die Werte aller Knoten
open fun drawAllText(batch: SpriteBatch, font: BitmapFont, layout: GlyphLayout){
    for(t in heap){
        drawText(t, batch, font, layout)
    }
}

//Rekursives Ausgeben der Prioritäten des Baums
open fun drawText(tree : Tree, batch: SpriteBatch, font: BitmapFont, layout: GlyphLayout){
    if(tree is BinominalTree){
        drawText(tree.leftTree, batch, font, layout)
        drawText(tree.rightTree, batch, font, layout)
    } else if(tree is BinominalTreeDZero) {
        tree.node.drawText(batch, font, layout)
    }
}

//Funktion zum Zeichnen aller Bäume
open fun drawAllTrees(shapeRenderer: ShapeRenderer){
    shapeRenderer.setColor(Color.RED)
    for(t in heap){
        drawTree(t, shapeRenderer)
    }
}

//Rekursive Funktion zum Zeichnen eines Baumes
open fun drawTree(tree : Tree, shapeRenderer: ShapeRenderer){
    if(tree is BinominalTree){
        drawTree(tree.rightTree, shapeRenderer)
        drawTree(tree.leftTree, shapeRenderer)
    } else if(tree is BinominalTreeDZero) {
        tree.node.draw(shapeRenderer)
    }
}

//Leeren der Liste aller Linien
open fun clearLines(){
    lines.clear()
}
```

Screenshots des Programms in Ausführung:


20
38
33
48
81
95
45
56


6
8
17
52
31
28
32
64
92
49
48
91
44
93
36
77


2
21 15 4 8 7
18 88 23 76 19 18 52 27 35 8
72 38 91 22 95 37 78 78 86 25
30 81 99 90 37
37


2
20 19 25 17
14 58 54 62 78 74 28
40 41 34 18 88 75 75 74
26 56 45 62 30 20 94
65 86 88

Literaturliste

[1] Th. H. Cormen: Algorithmen - Eine Einführung Oldenburg 2. + 4. Auflage

Binär-Heaps:
[2] Williams, J. W. J. (1964), "Algorithm 232 - Heapsort", Communications of the ACM, Band 7 (Nr. 6, Juni 1964, S. 347–348)

[3] Floyd, Robert W. (1964), "Algorithm 245 - Treesort 3", Communications of the ACM, Band 7 (Nr. 12, Dezember 1964, S. 701)

Binomial-Heaps:
[4] Jean Vuillemin - A Data Structure for Manipulating Priority Queues

Literaturverzeichnis

Priority Queues:
https://en.wikipedia.org/wiki/Priority_queue

Binär Heaps:
https://de.wikipedia.org/wiki/Bin%C3%A4rer_Heap#

Binomial- Heaps
https://de.wikipedia.org/wiki/Binomial-Heap

https://www.geeksforgeeks.org/binomial-heap-2/

https://www.brilliant.org/wiki/binomial-heap/#minimum-functionalities