**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.

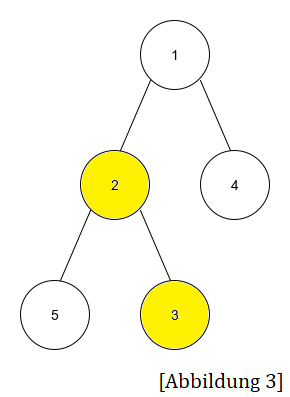
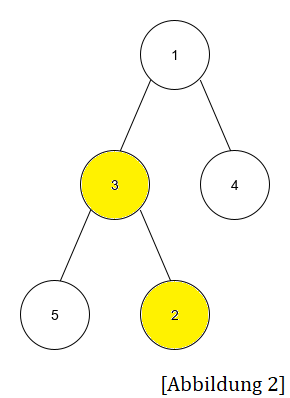
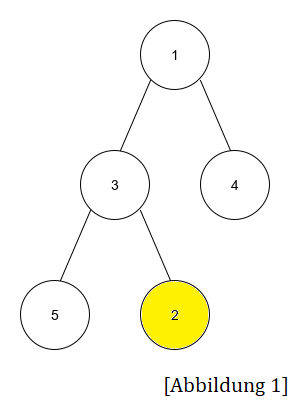
**Thema**

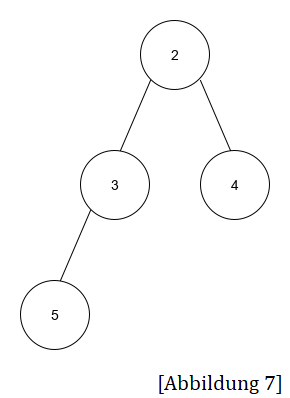
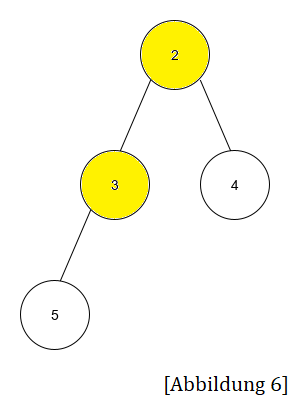
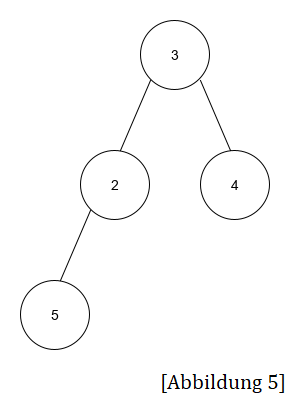
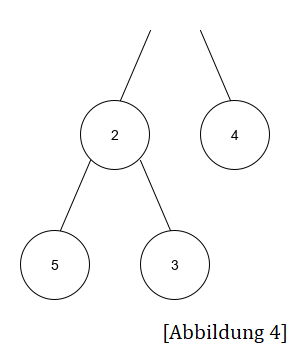
In folgendem Abschnitt gehen wir genauer auf die beiden Themen „Binär-Heaps“ und „Binomial-Heaps“ ein. Wir werden hier genauer auf die Verwendung der Prioritätswarteschlangen und auf die Unterschiede zwischen den Heaps eingehen. Wo die Vor- und Nachteile der Heaps liegen oder wo Sie sich zu anderen Sortieralgorithmen 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:

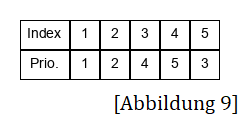
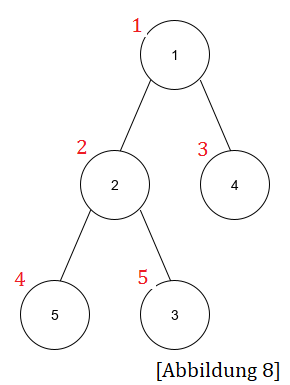
Die Datenstruktur des binären Heaps versteht man als einen fast vollständigen binären Baum. Ein Knoten eines binären Baums hat maximal zwei nachfolgende Knoten (Child). Durch die Verwendung der Prioritätswarteschlange (Priority Queues) werden die Elemente in einem binären Baum nach Priorität sortiert. 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*.

  
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”.



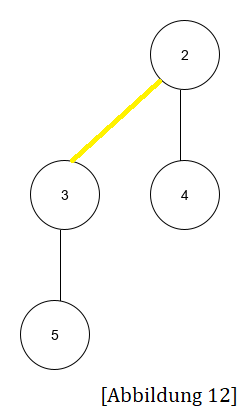
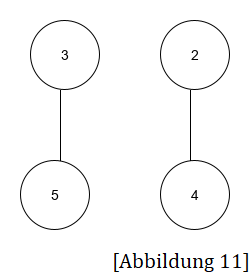
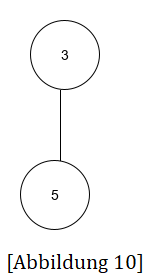
Um den sortierten Baum als Heap in einem Array zu speichern, wurden zuvor alle Knoten mit Indizes *i* beschriftet [Abbildung 8]. Diese werden entsprechend der Indizes in einem Array angeordnet [Abbildung 9]. Dies ermöglicht mit der Formel *i = i \* 2 + 1* den Zugriff auf das linke Childelement, mit *i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



Binomial-Heaps:

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.



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.

Laufzeitvergleich:

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.

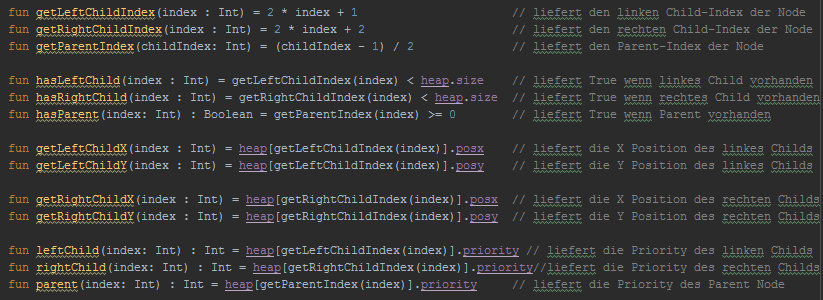


Hierbei bilden posx und posy zusammen die Position der Node auf dem Bildschirm für die Grafische Darstellung.

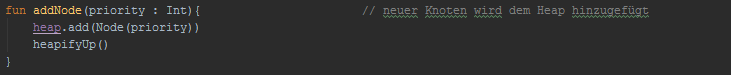
Eine Liste mit allen Nodes wird in der Oberklasse des Binary Heaps gespeichert.



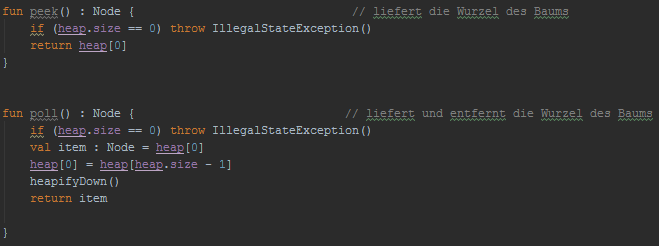
In dieser Implementierung haben wir uns für einen Array ähnlichen Ansatz anstatt eines Objektorientierten für den Binary Tree entschieden.  
Für diesen Ansatz sind einige Funktionen nötig um von einem Node bzw. des Indexes dessen auf die Parent- und Child Nodes zu schließen.

****

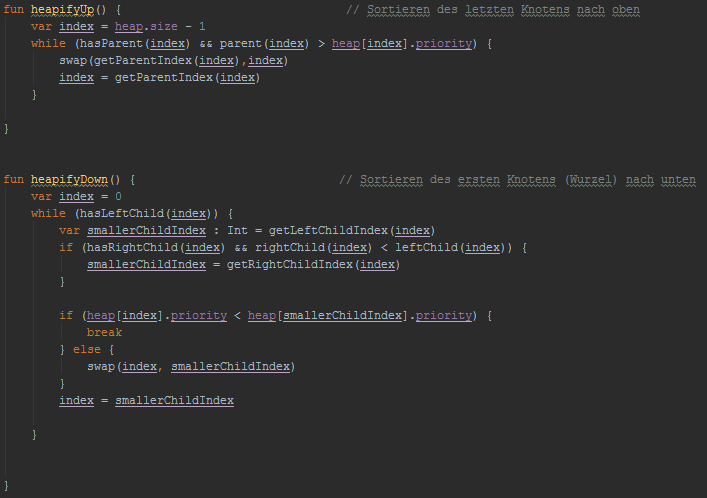
Nodes können mit einer *Add* Funktion hinzugefügt werden.



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.

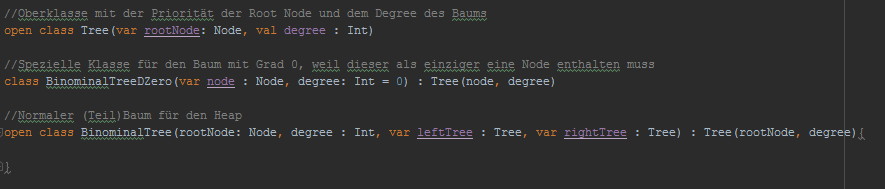


Nach solchen Funktionen die die Integrität des Binary Heaps schädigen muss entsprechend die HeapifyUp- oder HeapifyDown Funktion aufgerufen werden um die Integrität wieder herzustellen.

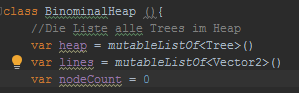


Datenstruktur Binominal-Heap:

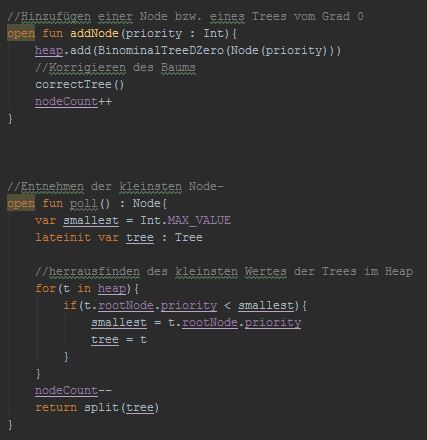
Für die Implementation des Binominal Heaps benutzen wir als Basis dieselbe Node Klasse die wir auch für den Binary Heap benutzt haben.  
Desweiteren haben wir uns für eine Implementation entschieden, die Teilbäume benutzt um den Binominal Tree zu formen. Jeder Baum hat einige Grundattribute gemeinsam,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

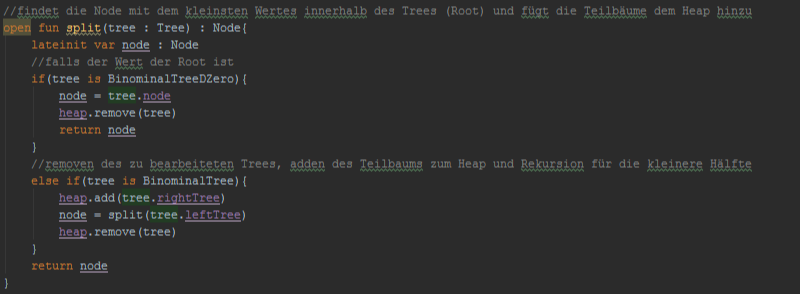


Die Oberklasse enthält hier wieder eine Liste mit vollständigen Bäumen und die Anzahl aller Nodes in Diesen. Die Liste ‘Lines’ enthält alle zu zeichnenden Linien für die grafische Darstellung.

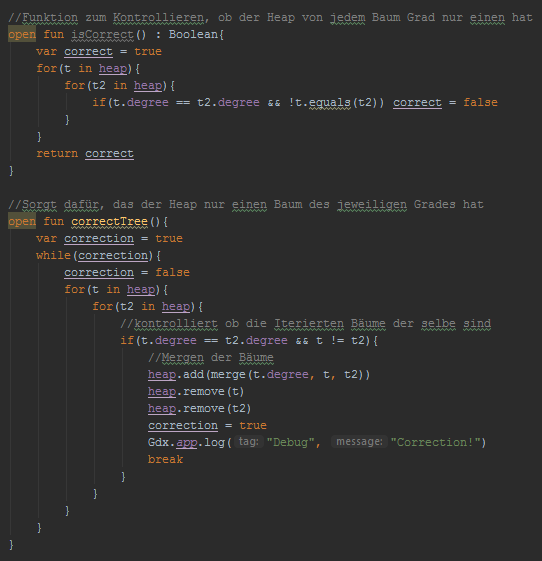


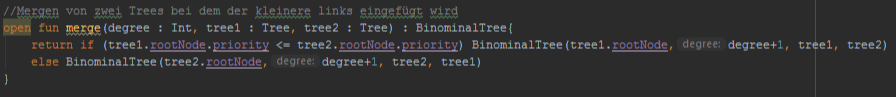
Auch für den Binominal Heap haben wir eine *Add* und *Poll* Funktion eingebaut.  
Für die *Poll* Funktion wird eine *Split* Funktion benötigt die den Root eines Baumes entfernt und die übriggebliebenen Teilbäume dem Heap hinzufügt.





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.



****

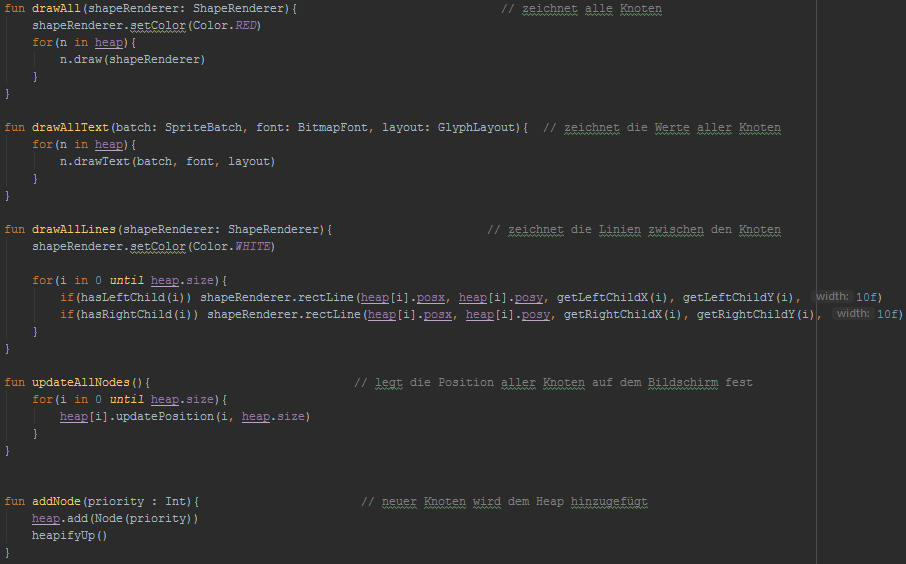
Kurzfassung der Grafischen Oberfläche:

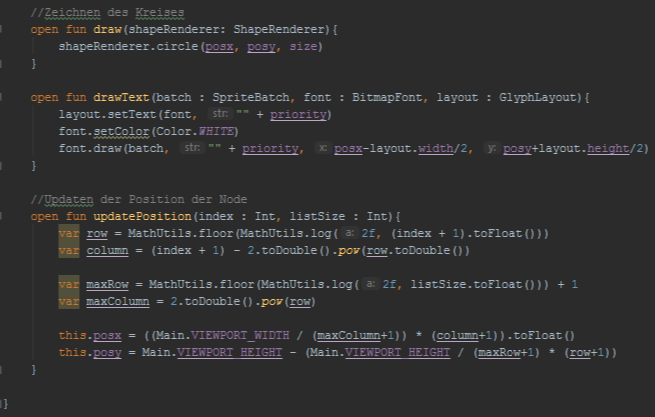
Da der Fokus dieser Dokumentation nicht auf der Grafischen Oberfläche liegt, folgt hier nur eine Kurzfassung dieser.

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.

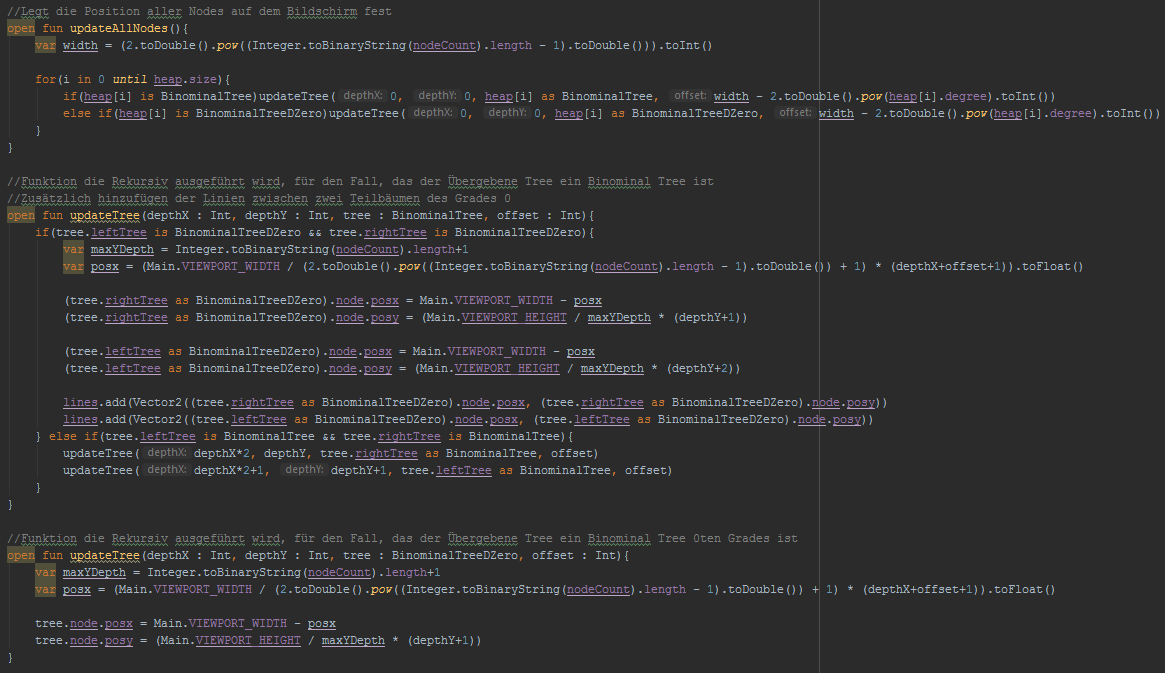
Die einzelnen Positionen der Nodes werden rechnerisch bestimmt und müssen nach einem *Add* oder *Poll* erneut durch eine Update Funktion berechnet werden.

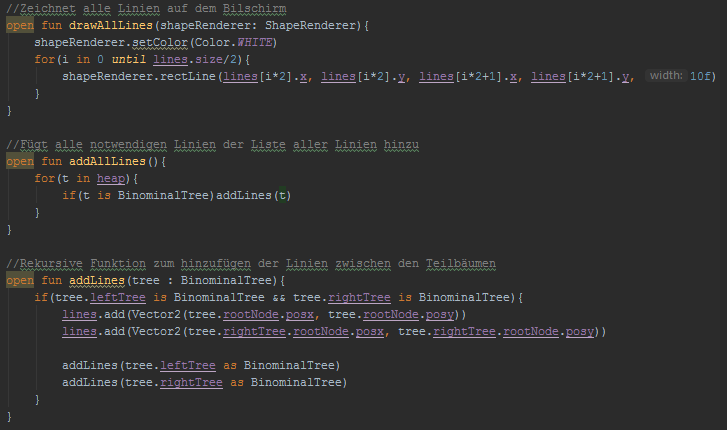
Zeichnen des Binary Heaps:

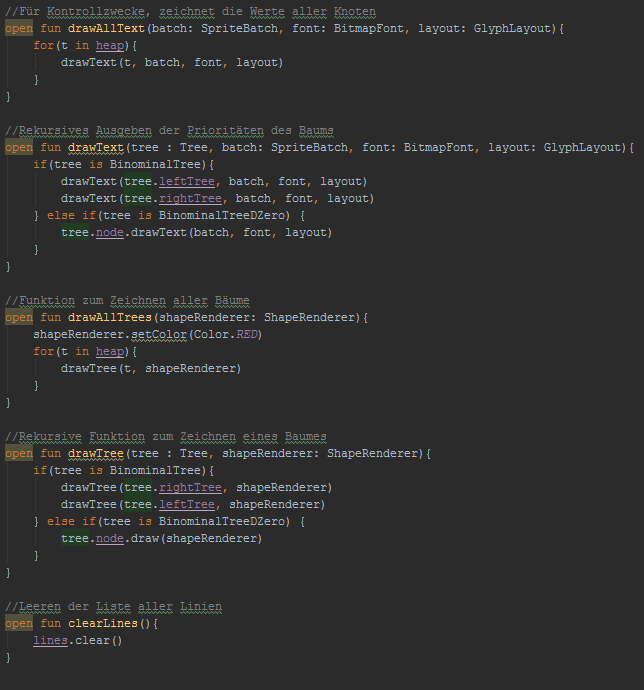




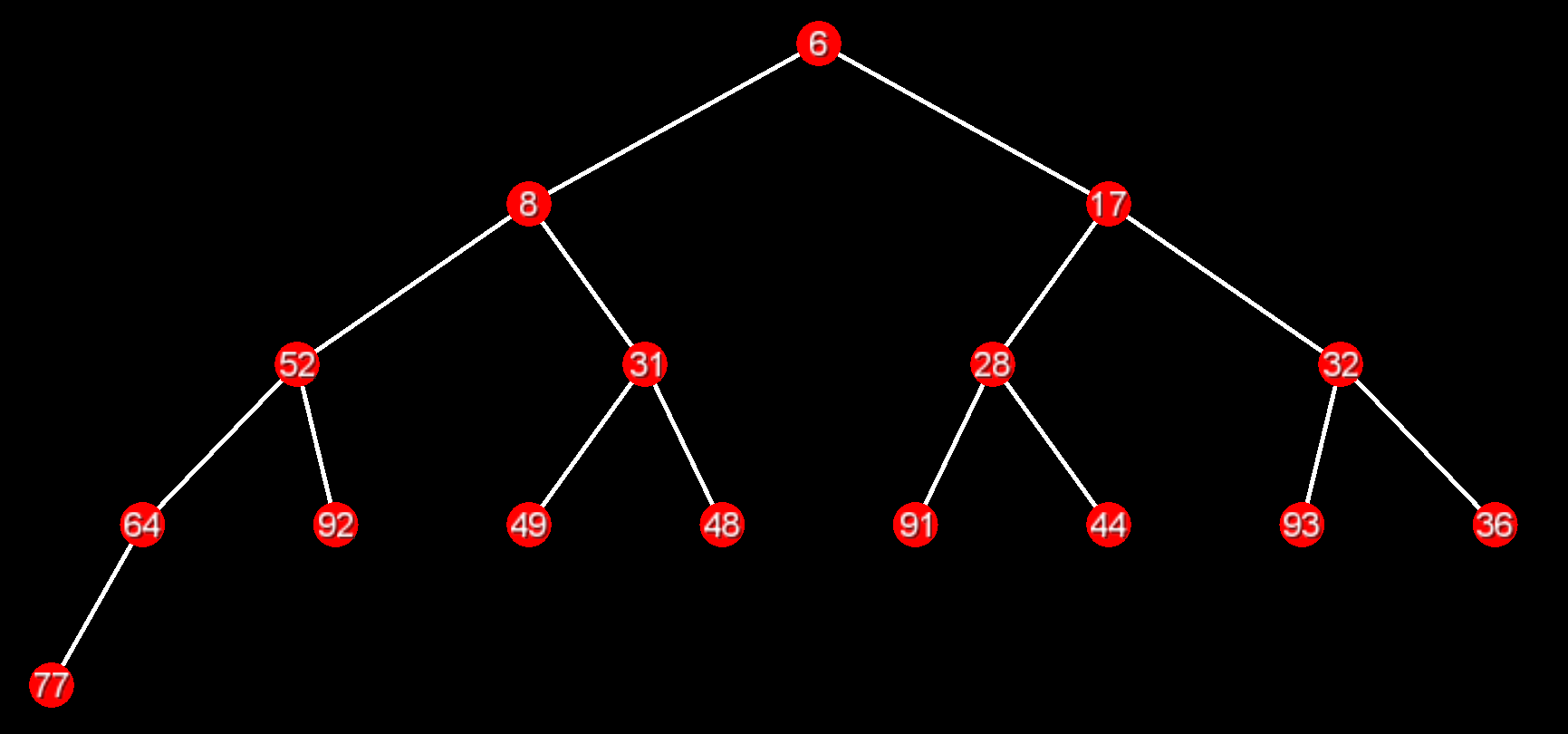
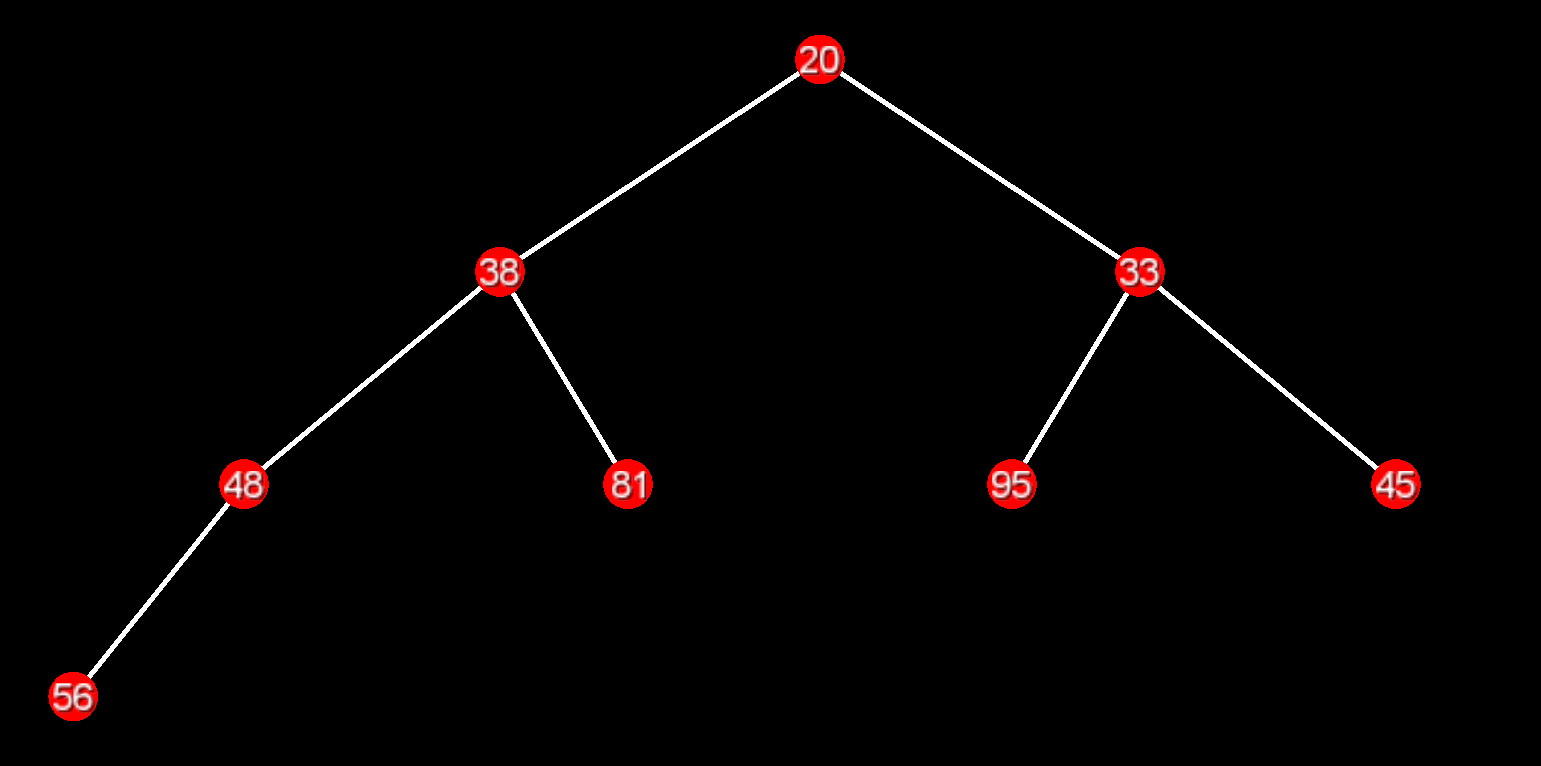
Zeichnen des Binominal Heaps:

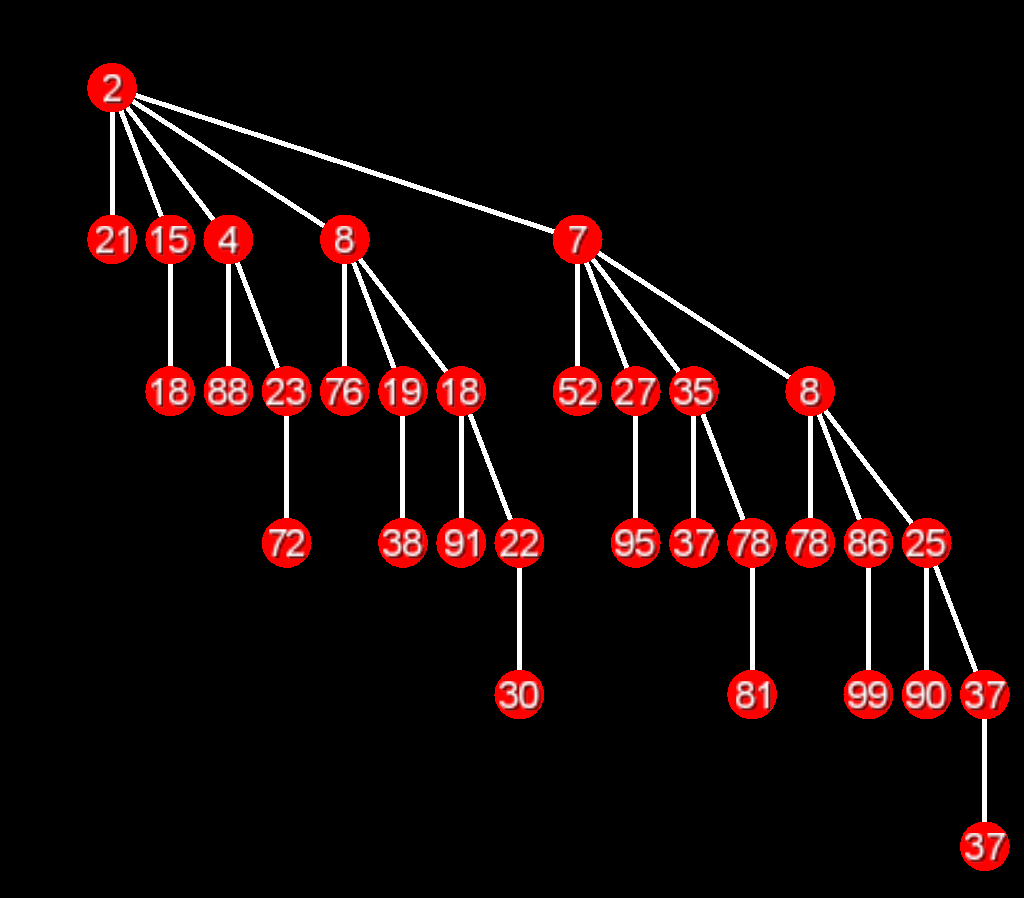


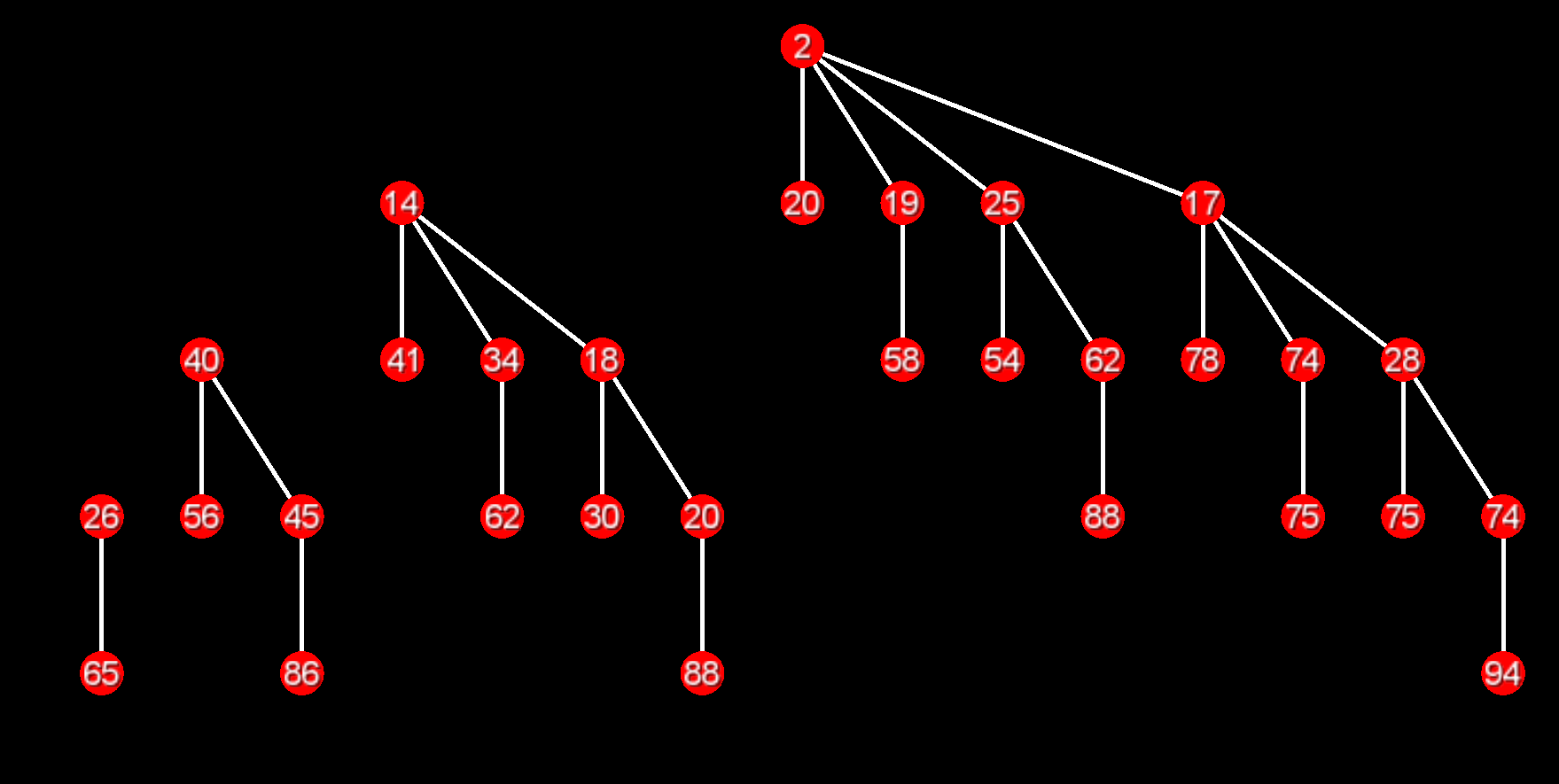




Screenshots des Programms in Ausführung:







**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