Aufgabe 1

Selectionsort

Da bei Selectionsort immer der nicht sortierte Teil durchlaufen werden muss, um das Maximum zu finden, werden selbst im Best Case $\frac{n(n-1)}{2} = \Omega(n^2)$ Operationen benötigt. Die Laufzeit ist unabhängig von F in $\Theta(n^2)$.

Anmerkung: Wenn man eine Version von Selectionsort benutzt, die das Maximum nicht mit dem letzten Element austauscht, sondern die Struktur des unsortierten Teils erhält, so lässt sich die Anzahl der einzigartigen Vergleiche auf $\mathcal{O}(n + F)$ beschränken.

Heapsort

Die Laufzeit von Heapsort ist unabhängig von F in $\Theta(n \log n)$. Für die Analyse der Best Case Laufzeit kann man die Laufzeit der `heapify` Operationen für die initialen Blätter betrachten.

Nehmen wir an, n Elemente bilden einen vollständigen Heap der Höhe $h = \lfloor \log n \rfloor$. (Keine Beschränkung der Allgemeinheit, da für jedes n' ein vollständiger Heap der Größe mindestens $n'/2$ auftritt.) Betrachtet man nun die $\lceil n/2 \rceil$ vielen Blätter, so werden diese eines nach dem anderen an die Stelle der Wurzel gesetzt, bevor sie mit `heapify` an eine gültige Position sinken. Die Laufzeit von `heapify` entspricht dabei der Höhe dieser gültigen Position.

Wir betrachten nun den Teilbaum der Höhe $h - 3$ (Den Heap ohne seine drei untersten Ebenen). Ein initiales Blatt kann sich nach den $n/2$ `heapify` Operation für die initialen Blätter entweder in diesem Teilbaum befinden oder es wurde als ein Maximum entfernt oder aber es befindet sich im Heap mit einer Höhe größer als $h - 3$.

Die Anzahl aller Knoten in diesem Teilbaum ist $n/8$, also befinden sich maximal $n/8$ viele initialen Blätter im Teilbaum.

Sei k die Anzahl der Blätter die den Heap als Maximum verlassen haben, das bedeutet k Blätter waren unter den $n/2$ größten Elementen. Alle initialen Vorfahren dieser Blätter sind aber größer als mindestens eines dieser Blätter. Betrachtet man den Baum der nur aus den k Blättern und ihren Vorfahren besteht, so handelt es sich um einen Baum, bei dem alle inneren Knoten Grad kleiner gleich 2 besitzen und der über k Blätter verfügt. Dieser Baum hat mindestens $k - 1$ innere Knoten. Somit befinden sich unter den $n/2$ größten Knoten (die Knoten die entfernt werden) maximal $n/4$ Blätter.

Es verbleiben demnach mindestens $n/8$ Blätter, die sich im Heap auf einer Höhe größer $h - 3$ befinden. Da diese Blätter durch andere `heapify` Operation nur aufsteigen können, wurden sie durch `heapify` auf mindestens dieser Höhe platziert. Dies benötigt mindestens $\frac{1}{8}n(h - 2) = \Omega(n \log n)$ viele Operationen.

Insertionsort

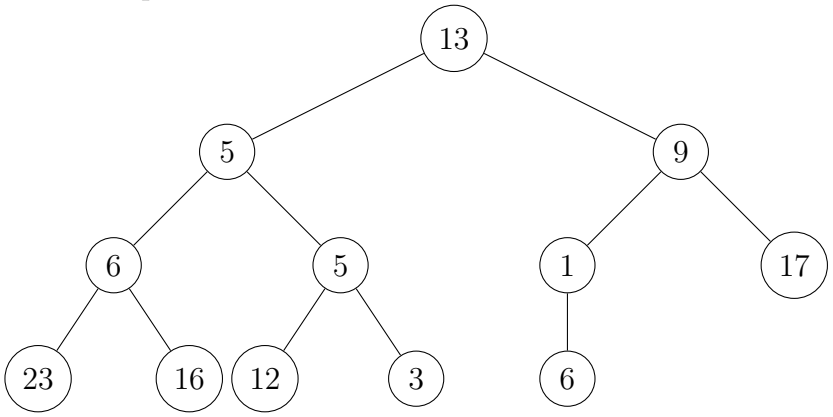
Insertionsort benötigt im Allgemeinen $\mathcal{O}(n + F)$ Operationen.

Im Folgenden sei $A = \langle a_1, \ldots, a_n \rangle$ das Array. Wir definieren die zu A gehörende Inversionssequenz als $(v_i, \ldots, v_n) \in \mathbb{N}^n$ wobei $v_j = \left| \left\{ (i,j) \middle| i < j \wedge a_i > a_j \right\} \right|$. Offensichtlich ist $\sum_{i=1}^{n} v_i = F$. Wir behaupten, dass bei Insertionsort in der i-ten Iteration der While-Schleife genau $v_i + 1$ Vergleiche durchführt werden. Nehmen wir an die ersten $i - 1$ Iterationen sind bereits abgeschlossen. Dann ist das Array $(a_0, \ldots, a_{i-1})$ sortiert und wir vergleichen solange paarweise, bis ein j gefunden ist mit $a_j < a_i$. Wir vergleichen also mit allen größeren Elementen, welche links von a_i sind und mit dem ersten Element, dass kleiner ist. Die Anzahl größerer Elemente ist v_i (siehe oben), also benötigen wir $v_i + 1$ Vergleiche.

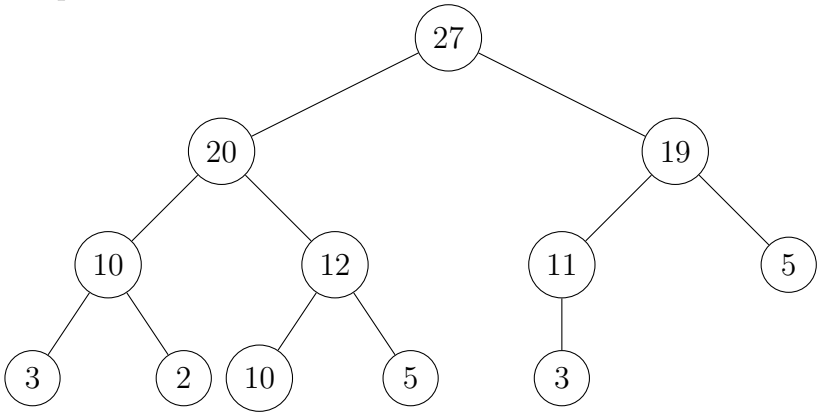
Insgesamt benötigen wir für den inneren Teil der While-Schleife $\mathcal{O}(F)$ Operationen (über alle Elemente gerechnet), da wir hier für jeden Vergleich Operationen, welche in konstanter Zeit ausführbar sind, durchführen. Die Iteration über alle Elemente des Arrays benötigt zusätzlich $\mathcal{O}(n)$ Zeit, was insgesamt zur Laufzeit $\mathcal{O}(n + F)$ führt.
Das bedeutet, wenn F in $\mathcal{O}(n)$ liegt, können wir mit Insertionsort in $\mathcal{O}(n)$ sortieren. Im Allgemeinen ergibt es Sinn Insertionsort zu verwenden, wenn F in $\mathcal{O}(n \log(n))$ liegt.

Aufgabe 2

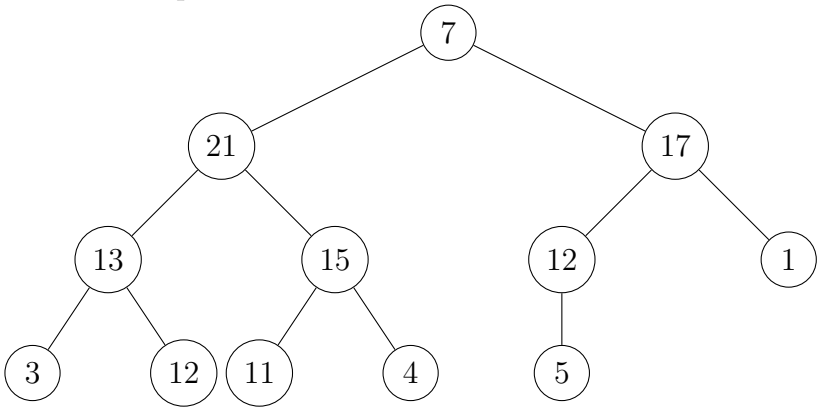
a) Nicht Heap



b) Heap

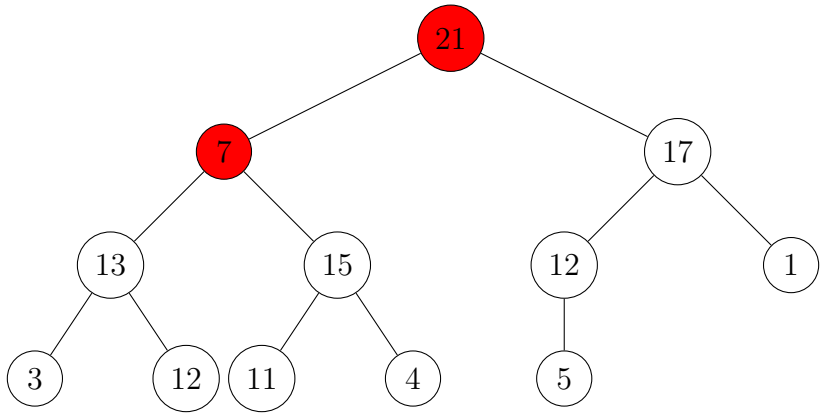


c) Beinahe Heap

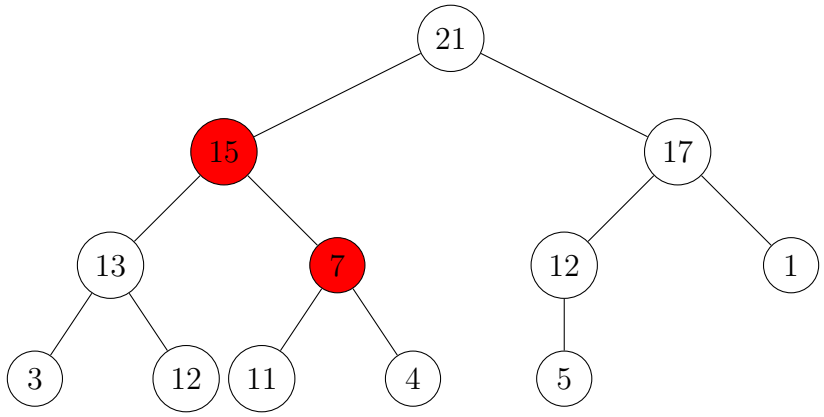


Heapify:

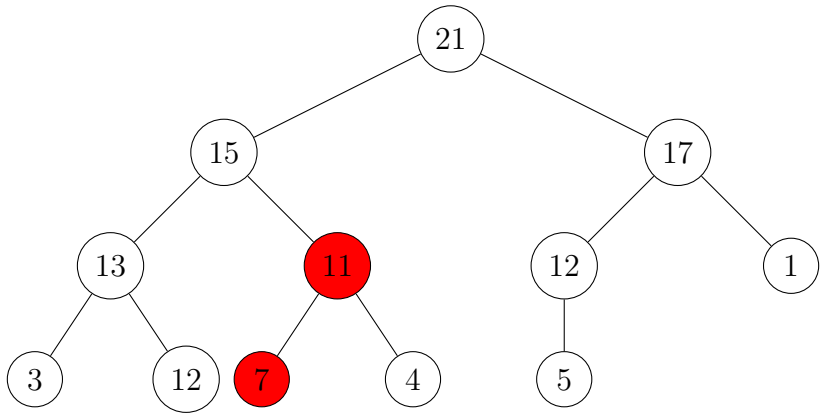
1)



2)



3)



Aufgabe 3

Erwartete Lösung in $h + \lceil \log_2 h \rceil$

Wenn wir ein Element in den Heap hineinsinken lassen, so führen wir immer 2 Vergleichsoperationen pro Ebene aus. Eine um, festzustellen ob wir das Element noch weiter nach unten tauschen müssen und eine um festzustellen ob nach links oder nach rechts. Wir möchten hier einige Vergleiche sparen.

Wir nennen den Pfad, der von der Wurzel ausgehend jeweils zum maximalen Kind führt den *Pfad der Größten*. So wäre im Beinahe-Heap aus Aufgabe 2 der Pfad $7 - 21 - 15 - 11$ der Pfad der Größten. Egal ob wir eine 2, eine 7, oder eine 31 einfügen würden, wir würden immer alle Täusche auf einem Teilpfad durchführen (da wir, sollten wir tauschen immer das Maximum der Kinder mit Wurzel des Teilbaums tauschen).

Wir wissen aber noch nicht, bis wohin wir das neue Element einsinken lassen müssen. Wir würden solange entlang des Pfades tauschen, bis das erste mal ein Element kleiner unserer einzufügenden Zahl ist. Wenn wir dieses Element gefunden haben, können wir ohne weitere Vergleiche die Elemente nach oben tauschen und die aktuelle Wurzel an der richtigen Position einfügen. Mit einer linearen Suche würden wir im Worst-Case (z.B. bei Einfügen der 7 im oberen Heap) weiterhin n Vergleiche brauchen, da aber die Elemente

auf dem Pfad (bis auf die Wurzel) absteigend sortiert sind, können wir mit binärer Suche die Position finden und benötigen somit nur noch $h + \log_2 h$ viele Vergleiche.

Die Komplexität der Vergleiche bei diesem Ansatz beträgt $h + \lceil \log_2 h \rceil <_{\text{fü}} 2h$.

```
procedure Sift-Down(H)
      path ← PathOfBiggest(H)
      position ← PositionInSorted(H[1], path[2...h], H)
      InsertPushUp(H[1], path[position], H)

// Return the path of the biggest elements in the heap
PathOfBiggest(H):
      path ← new array of size h
      path[1] = 1 // Der Pfad beginnt immer bei der Wurzel
      parent = 1 // Der „aktuelle Parent\ ist die Wurzel
      for i from 2 through h do
            if RightChild(parent) < LeftChild(parent)
                  nextParent ← LeftChild(parent)
            else
                  nextParent ← RightChild(parent)

            path[i] = nextParent // Pfad führt durch das größere Kind
            parent = nextParent
      return path

// Find where value would be inserted in given path
// Can return any value from a − 1 through b
PositionInSorted(value, path[a...b], H)
      if a > b
            return b
      m ← ⌊(a+b)/2⌋
      // Heap ist von groß nach klein, also der Pfad auch
      if value > H[path[m]]
            return PositionInSorted(value, path[a...m], H)
      else
            return PositionInSorted(value, path[m + 1...b], H)

// Insert at the specified position, pushing everything
// upwards in the heap, popping out the root
InsertPushUp(what, where, H)
      popped ← H[where]
      H[where] = what
      if where > 1
            InsertPushUp(popped, Parent(where), H)
```

Dieser Ansatz braucht für den Fall $h = 2$ maximal 4 Vergleiche, für $h = 3$ maximal 6.

Etwas schnellere Lösung in $h + \log^*(h) + O(1) <_{\text{füü}} h + \lceil \log_2 h \rceil$

Anstatt den gesamten Pfad von Anfang an zu berechnen, können auch die ersten $h - \log_2(h)$ Elemente des Pfades berechnet werden, gefolgt von einem Vergleich mit dem letzten Element:

$\in$ Pfad: Dann sind nur noch $\log_2(h - \log_2(h))$ Vergleiche (zur Binärsuche) notwendig. Die Vergleiche zur Festlegung des restlichen Pfades entfallen.

$\notin$ Pfad: In diesem Fall kann der Algorithmus rekursiv absteigen, und das Ende des Pfades als neue Wurzel betrachten. Selbst wenn hier der obige „langsame“ Algorithmus verwendet wird, wurden einige Vergleiche gespart.

Dieses Verhalten führt zu folgender Rekurrenz-Gleichung bei den Kosten:

$$C(h) \leq h - \log_2(h) + 1 + \max\{\log_2(h - \log_2(h)), C(\log_2(h))\}$$

Wir definieren[1] $\log^\star(h) = 0$, wenn $h \leq 1$, und $1 + \log^\star(\log(h))$, wenn $h > 1$.

Diese Optimierung ist aber eher marginal. Kurz ausgerechnet: Für 8000000 (und $C(0) = 0$) ergibt diese Rekurrenz 8000005. Der obige „langsame“ Ansatz braucht 8000023. Heaps dieser Höhe (also mit $2^{8\text{Mio.}}$ Elementen) haben da ganz andere Probleme. Optimierungen werden an anderen Stellen eher gebraucht.

Zu schnelle Lösungen

Sublineare Ansätze sind grundsätzlich falsch: Damit ließe sich ein Heapsort in $o(n \log n)$ basteln, Widerspruch.

[1] `http://en.wikipedia.org/wiki/Iterated_logarithm`

Aufgabe 4

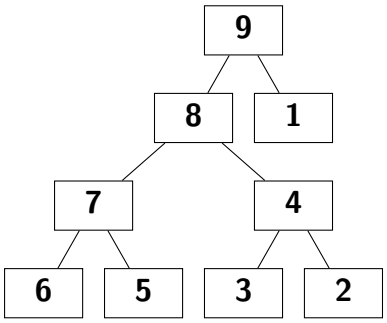
Das **linke Kind** eines Knoten i können wir mit $\texttt{lc}(i) = i + 1$ berechnen. Es ist nicht möglich ohne andere Informationen zum rechten Kind zu navigieren. Wir benötigen die Tiefe des Kindes und die Gesamttiefe des Baumes. Alternativ für diese beiden Werte kann auch die Höhe des induzierten Teilbaumes jedes Knoten verwendet werden. Wir definieren $d(i)$ als Tiefe des Knotens i und h als die Gesamthöhe des Baumes. Somit können wir das **rechte Kind** mit $\texttt{rc}(i) = i + 2^{h-d(i)}$ berechnen.

Für die Funktionsweise von **heapsort** ist es nötig, dass das Element mit dem größten Index, jenes das im Speicher an letzter Stelle steht freigegeben wird. Wenn man Preorder benutzt, so erfolgt dieses Freigeben nicht von unten nach oben. Es gilt aber trotzdem, dass der Baum zusammenhängend bleibt, da die Kinder eines Elements einen größeren Index besitzen und damit bereits freigegeben wurden bevor ein Knoten selbst entfernt wird. Außerdem soll gelten: Wenn immer ein Knoten ein rechtes Kind besitzt, so ist der linke Teilbaum vollständig. Diese Invariante bleibt erhalten, da alle Knoten im linken Teilbaum einen kleineren Index als das rechte Kind besitzen.

In der abgewandelten **Makeheap** Funktion gilt es darauf zu achten, dass die linken Teilbäume alle vollständig sind, wenn ein rechtes Kind existiert.

```
makeheap(A, i, j, h)
        n = j - i + 1
        if n < 3 then
                return
        m = i + (1 << h) - 1
        m = min(m, j)
        makeheap(A, i+1, m, h-1)
        makeheap(A, m+1, j, h-1)
        heapify(A, i, j, h)
end
```

Die entstehenden Heaps sind nicht balanciert beispielsweise würde die Liste $[9, 8, 7, 6, 5, 4, 3, 2, 1]$ in den folgenden Heap verwandelt:



Da der linke Teilbaum aber balanciert ist, und der rechte Teilbaum nicht höher ist als der linke, liegt die Höhe weiterhin in $\mathcal{O}(\log n)$.

Wir speichern uns die Höhe des Baums und schleifen diese bei den **heapify** Operationen

mit. Die berechneten Positionen der rechten Kinder entsprechen dabei nicht der Position die ein zusätzlich eingefügtes rechtes Kind tatsächlich besäße, sondern ergeben nur Sinn solange ein rechtes Kind existiert, der linke Teilbaum also vollständig ist. Wenn kein rechtes Kind existiert, erhalten wir aber immer eine Position die zu groß ist.

```
heapify(A, i, j, h)
        l = i + 1
        r = i + (1 << (h-1))
        m = i + (1 << h) - 1
        m = min(m, j)
        if l > j then
                return
        end
        if r > j then
                if A[i] > A[l]
                        return
                else
                swap A[i], A[l]
                heapify(A, l, m, h-1)
        end
        if A[l] > A[r] then
                if A[i] > A[l] then
                        return
                else
                        swap A[i], A[l]
                        heapify(A, l, m, h-1)
                end
        else
                if A[i] > A[r] then
                        return
                else
                        swap A[i], A[r]
                        heapify(A, m+1, j, h-1)
                end
        end
end

heapsort(A, i, j)
        n = j - i + 1
        h = log n
        makeheap(A, i, j, h)
        for k from j to i+1 do
                swap A[k], A[i]
                heapify(A, i, k, h)
        end
```

Heapify lässt sich also auch mit Preorder in-place implementieren.

Lokales Navigieren

Betrachten wir nun lokales Navigieren mit Preorder im Allgemeinen.

Für den **Elter (Parent)** müssen wir uns ohne weitere Informationen eines Tests bedienen. Dazu definieren wir $i' = i - 2^{h-d(i)+1}$. Falls $rc(i') = i$, geben wir $p(i) = i'$ zurück, ansonsten $p(i) = i - 1$. Alternativ können wir auch prüfen ob $d(i-1) = d(i) - 1$ gilt: Falls ja, so ist $p(i) = i-1$, sonst gilt $p(i) = i'$ mit i' wie eben. Diese Fallunterscheidung kann offensichtlich vermieden werden, wenn wir zusätzlich für jeden Knoten ein Flag speichern, das angibt, ob der Knoten linkes oder rechtes Kind ist.

Mit *Preprocessing* können wir die Tiefe für alle Knoten in insgesamt $\mathcal{O}(n)$ Zeit rekursiv berechnen und in $\mathcal{O}(1)$ nachschlagen. Um den Wert *dynamisch* zu ermitteln, benötigt der naive Ansatz $\mathcal{O}(h) = \mathcal{O}(\log n)$ Zeit für jedes Element, da der gesamte Baum (meistens) in seiner Tiefe durchlaufen werden muss.

Für den Knoten i lässt sich diese Laufzeit relativ leicht auf $\mathcal{O}(\log i)$ reduzieren: Falls $i \leq \log n$ liegt i auf dem linkesten Ast des Baumes[2] und wir können in $\mathcal{O}(1)$ i als Tiefe ausgeben. Ansonsten liegt i in einem der rechten Teilbäume eines Knoten j dieses Astes (möglicherweise ist j die Wurzel).
Wir stellen fest, dass die Knoten im rechten Teilbaum von j Nummern $\geq j + 2^{h-j}$ haben. Wir suchen nun den Teilbaum der i enthält, also das größte j (den tiefsten Knoten j), sodass $j + 2^{h-j} \leq i$ gilt. Dies impliziert wiederum, dass $h - j \leq \log i$. Somit gibt es nur $\mathcal{O}(\log i)$ Kandidaten für j (nämlich h bis $h - \log i$) und $\mathcal{O}(\log i)$ Vergleiche reichen zum finden von j aus. Außerdem besitzt der Teilbaum von j nur Tiefe $h - j$ und wir können die Tiefe von i in Zeit $\mathcal{O}(h - j) = \mathcal{O}(\log i)$ naiv berechnen. Der Beweis welche Laufzeit erzielt werden kann, wenn die Prozedur nur rekursiv angewendet wird, sei jedem selbst überlassen.

Darüber hinaus kann abhängig vom Modell eine Laufzeit von $\mathcal{O}(\log \log i)$ folgendermaßen erreicht werden:
Wir nehmen an, dass unser Binärbaum vollständig ist und lassen zur besseren Anschauung unsere Preorder bei $-h$ beginnen. Das bedeutet, dass das nullte Blatt den Preorder Index 0 besitzt. Generell ist für das k-te Blatt der Preorder Index $a(k)$ immer gleich (sofern er existiert).

Für diese Abbildung von Blattnummer zu Preorder Index a gilt $a(k) = 2k - \texttt{HW}(k)$ dabei ist $\texttt{HW}(k)$ die Hamming Weight, die Anzahl der Einsen in der Binärdarstellung von k. Da $0 \leq \texttt{HW}(k) \leq \log(k)$, erhalten wir die Schranken $2k - \log k \leq a(k) \leq 2k$. Entsprechend gilt für die inverse Abbildung von Index zu Blattnummer $\frac{i}{2} \leq a^{-1}(i)$ und $a^{-1}(i) \leq \frac{i}{2} + \log k \leq \frac{i}{2} + \log i$.

Der Algorithmus basiert darauf für einen Knoten i das nächstgrößere Blatt zu finden. Da dies der $d(i)$-fache linke Nachfahre ist, hat er den Index $i + d(i)$ und wir können die Tiefe $d(i)$ durch Subtraktion bestimmen. Gemäß der Schranken liegt die Blattnummer

[2] Der Pfad bestehend aus der Wurzel und jeweils dem linken Kind eines Knoten im Pfad.

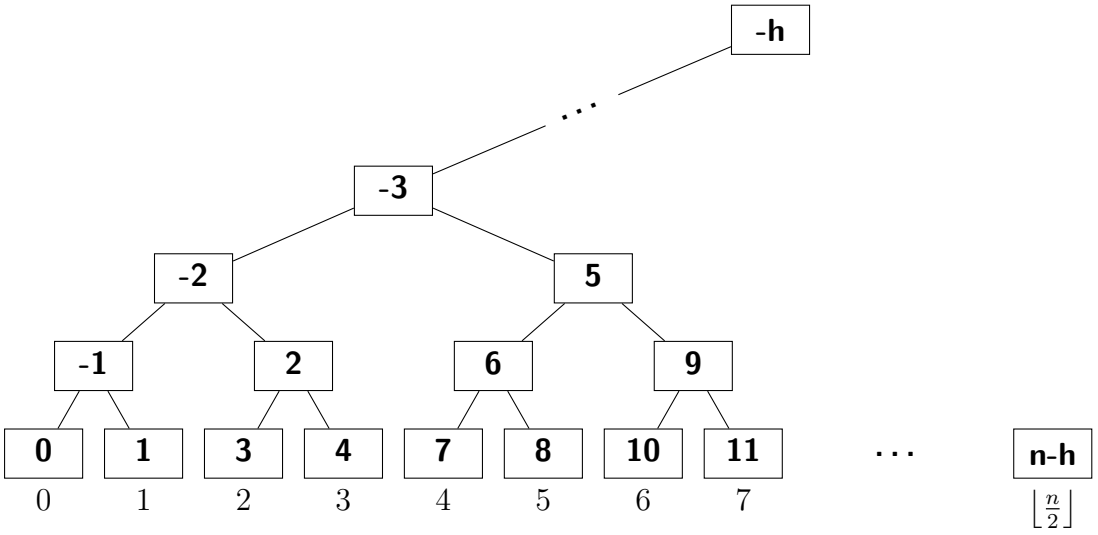


Abbildung 1: Knoten nummeriert nach verschobener Preorder und Blätter annotiert mit Blattnummer

des gesuchten Blatts zwischen $\frac{i}{2} \leq \frac{i+d(i)}{2}$ und $\frac{i+d(i)}{2} + \log(i + d(i)) \leq \frac{i}{2} + \frac{3}{2}\log i + 1$. Um das gesuchte Blatt zu finden wenden wir nun Binäre Suche über diese Blattnummern an, wobei wir in jedem Schritt vergleichen ob der Index $a(k)$ der aktuellen Blattnummer k größer als i ist.

Wir nehmen an, dass in unserem Modell `HW` in konstanter Zeit also so schnell wie Additionen, Stellenverschiebungen und logische Operationen berechnet werden kann. Diese Entscheidung lässt sich u.a. dadurch unterstützen, dass der entsprechende Schaltkreis gleiche oder bessere Tiefe sowie Größe besitzt und reale Rechner `HW` als primitive Operation unterstützen. Dadurch hat jeder Schritt der Binären Suche konstante Laufzeit.

Da wir ein Intervall der Länge $\mathcal{O}(\log i)$ durchsuchen, benötigt die Binäre Suche eine Laufzeit von $\mathcal{O}(\log\log i)$.

Für die Berechnung von $\log i$ (die Länge der Binärdarstellung von i) haben wir eine Laufzeit von $\mathcal{O}(\log\log i)$ zur Verfügung. Hier können wir erneut auf das Rechenmodell verweisen oder eine Form von Exponentieller und Binärer Suche anwenden. Wir können diesen Schritt aber auch umgehen indem wir stattdessen einfach h benutzen, was die Laufzeit der Binären Suche auf $\mathcal{O}(\log\log n)$ erhöht.