5.5 Halden und Prioritätswarteschlangen

Oft reicht ein so einfaches Konzept einer Warteschlange aber nicht aus. Von einer Prioritätswarteschlange spricht man, wenn die Elemente nach Priorität sortiert sind. Für einen Datensatz D schreiben wir jetzt $D.x$ für die Priorität von D. Die Priorität übernimmt also die Funktion des Schlüssels als Sortierkriterium. Wie nehmen allerdings nicht mehr an, dass die Prioritäten paarweise verschieden sind. Für eine Prioritätswarteschlange Q wollen wir zumindest die folgenden Operationen zur Verfügung stellen:

1. *Einfügen*(Q, D) fügt das Element D in Q (an geeigneter Stelle) ein.
2. *Maximum*(Q) gibt das Element maximaler Priorität zurück.
3. *ExtrahiereMaximum*(Q) löscht das Element maximaler Priorität aus Q und gibt es zurück.
4. *ErhöhePriorität*(Q, D, x) erhöht die Priorität des Elements D auf x wobei angenommen wird dass $x \geq D.x$.

Eine Prioritätswarteschlangen wird üblicherweise basierend auf der Datenstruktur der *Halde* (engl. *heap*) implementiert.

Definition 5.4. Eine Halde ist ein binärer Baum mit den folgenden Eigenschaften:

1. Alle bis auf das unterste Level sind vollständig aufgefüllt.
2. Das unterste Level ist von links bis zu einem bestimmten Punkt vollständig aufgefüllt.
3. Falls w ein Kind von v ist, dann ist Priorität$(w) \leq$ Priorität(v).

Daraus folgt unmittelbar dass die Wurzel des Baums maximale Priorität hat. Eine Halde kann (ähnlich wie ein Suchbaum) im Speicher mit Hilfe von Knoten abgelegt werden, die jeweils einen Zeiger *links* und einen Zeiger *rechts* haben. Wie wollen hier aber eine andere Darstellung angeben, die, je nach Anwendung, gewisse Vor- und Nachteile hat. Da alle Level bis auf das letzte vollständig ausgefüllt sind, können wir die Halde auf effiziente Weise in einem Datenfeld speichern das Level für Level von oben nach unten und, pro Level, von links nach rechts befüllt wird.

Beispiel 5.2. Die in Abbildung 5.4 als Baum dargestellte Halde hat als Datenfeld die folgende Form:

12	10	9	8	6	4	7	2	5	3

Um diese Darstellung einer Halde Q zu realisieren, speichern wir zusätzlich zu $Q.Länge$ auch noch die Haldenlänge $Q.HLänge$ die angibt bis zu welchem Punkt das Datenfeld befüllt ist. Wir werden bei allen Operationen annehmen dass $Q.Länge$ hinreichend groß ist. In der Praxis ist das z.B. dann erfüllt wenn im Vorhinein bekannt ist, wie viele Elemente Q höchstens enthalten wird. Falls diese Anzahl nicht im Vorhinein bekannt ist, muss bei einer Einfügeoperation gegebenenfalls die Länge des zugrundeliegenden Datenfelds erhöht werden, was mit zusätzlicher Komplexität verbunden sein kann.

Wir werden Elemente der Halde mit ihrem Index im zugrundeliegenden Datenfeld referenzieren. Dann hat das linke Kind des Elements mit Index i den Index $2i$, das rechte Kind den Index $2i+1$

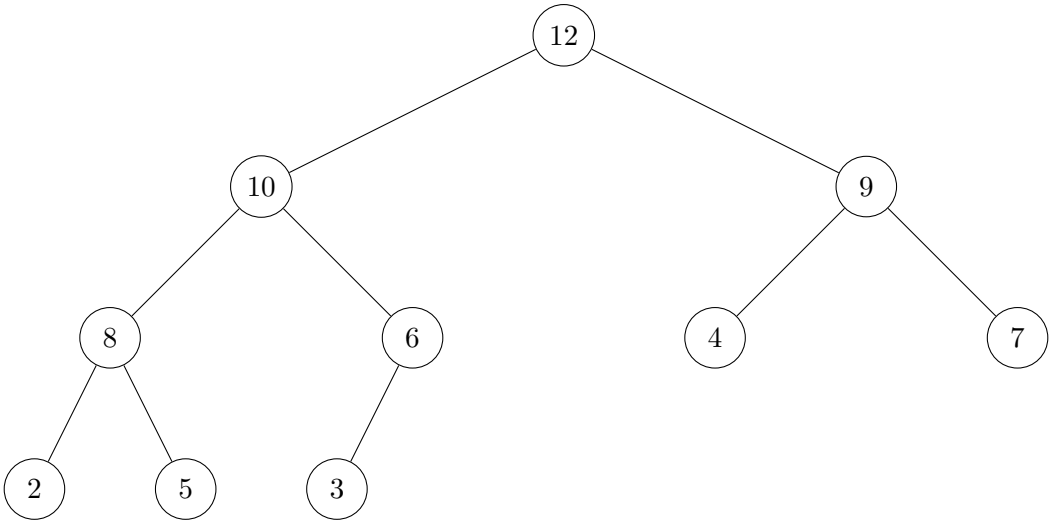

Abbildung 5.4: Eine Halde in Baumdarstellung

und der Vater den Index $\lfloor \frac{i}{2} \rfloor$. Dementsprechend definieren wir die Prozeduren LINKS$(i) := 2i$, RECHTS$(i) := 2i + 1$ und VATER$(i) := \lfloor \frac{i}{2} \rfloor$.

Das Einfügen eines neuen Elements geschieht dadurch, dass es zunächst an die letzte Stelle geschrieben wird und danach aufwärts an die richtige Stelle verschoben wird, siehe Algorithmus 20. Auf ähnliche Weise kann die Erhöhung der Priorität eines Elements implementiert werden, siehe Algorithmus 21. AUFWÄRTSKORRIGIEREN und damit auch EINFÜGEN und ERHÖHEPRIORITÄT benötigt in einer Halde mit n Elementen $O(\log n)$ Zeit.

Algorithmus 20 Einfügen in eine Prioritätswarteschlange

```
Prozedur EINFÜGEN(Q, D)
    Q.HLänge := Q.HLänge + 1          ▷ Annahme: Q.Länge ist hinreichend groß
    Q[Q.HLänge] := D
    AUFWÄRTSKORRIGIEREN(Q, Q.HLänge)
Ende Prozedur

Prozedur AUFWÄRTSKORRIGIEREN(Q, i)
    Solange i > 1 und Q[VATER(i)].x < Q[i].x
        Vertausche Q[i] und Q[VATER(i)]
        i := VATER(i)
    Ende Solange
Ende Prozedur
```

Zur Extraktion des Maximums geht man dual dazu vor: zunächst wird das Maximum (das sich ja an der Wurzel $Q[1]$ befindet) entfernt und durch das Element am Ende der Prioritätswarteschlange ersetzt. Danach wird dieses Element, das ja jetzt zu niedrige Priorität für seine Position hat, Stück für Stück durch Vertauschungen an eine korrekte Stelle (abwärts) verschoben, siehe Algorithmus 22. ABWÄRTSKORRIGIEREN und damit auch EXTRAHIEREMAXIMUM benötigt ebenfalls $O(\log n)$ Zeit.

Algorithmus 21 Erhöhung der Priorität in einer Prioritätswarteschlange

Vorbedingung: $x \geq Q[i].x$

```
Prozedur ErhöhePriorität(Q, i, x)
    Q[i].x := x
    AufwärtsKorrigieren(Q, i)
Ende Prozedur
```

Algorithmus 22 Extraktion des Maximums aus Prioritätswarteschlange

```
Prozedur ExtrahiereMaximum(Q)
    Falls Q.HLänge = 0 dann
        Antworte "Q ist leer"
    sonst
        max := Q[1]
        Q[1] := Q[Q.HLänge]
        Q.HLänge := Q.HLänge − 1
        AbwärtsKorrigieren(Q, 1)
        Antworte max
    Ende Falls
Ende Prozedur

Prozedur AbwärtsKorrigieren(Q, i)
    Solange i ≤ Q.HLänge
        Sei m ∈ {i, links(i), rechts(i)} ∩ [1, ..., Q.HLänge] so dass Q[m].x maximal ist
        Falls m = i dann                                   ▷ Abwärts-Korrektur beendet
            i := Q.HLänge + 1
        sonst
            Vertausche Q[i] und Q[m]
            i := m
        Ende Falls
    Ende Solange
Ende Prozedur
```

Anders als bei den Operationen für AVL-Bäume, die funktional und rekursiv implementiert waren, haben wir uns bei diesen Korrekturoperationen für eine imperative Implementierung entschieden. Der funktionale Programmierstil harmoniert üblicherweise gut mit Bäumen (und sonstigen rekursiv definierten Datenstrukturen), der imperative mit Datenfeldern (und anderen Datenstrukturen mit globalem Zugriff).

Auf Basis von Halden lässt sich auch ein Sortierverfahren angeben: *Haldensortieren* (engl. *heapsort*). Die Grundidee dieses Verfahrens besteht darin aus dem Eingabendatenfeld eine Halde aufzubauen und dann Schritt für Schritt jeweils das Maximum der Halde zu entfernen und an das Ende des Ausgabedatenfelds zu schreiben. Aufgrund der gewählten Implementierung einer Halde als Datenfeld können wir in diesem Verfahren für das Eingabedatenfeld, die Halde und das Ausgabedatenfeld den selben Speicherbereich benutzen, siehe Algorithmus 23. Man sieht sofort ein, dass ErzeugeHalde in Zeit $O(n \log n)$ läuft. Tatsächlich benötigt diese Prozedur sogar nur Zeit $O(n)$: Eine Halde mit n Elementen hat Höhe $\lfloor \log n \rfloor$ und höchstens $\lceil \frac{n}{2^{h+1}} \rceil$ Knoten der Höhe h. AbwärtsKorrigieren(A, i) benötigt Zeit $O(h)$ wobei h die Höhe von $A[i]$

Algorithmus 23 Haldensortieren (engl. *heapsort*)

```
Prozedur HaldensortierenHalde(A)
    ErzeugeHalde(A)
    Für i := A.Länge, ..., 2
        Vertausche A[1] und A[i]
        A.HLänge := A.HLänge − 1
        AbwärtsKorrigieren(A, 1)
    Ende Für
Ende Prozedur

Prozedur ErzeugeHalde(A)
    A.HLänge := A.Länge
    Für i := Vater(A.HLänge), ..., 1
        AbwärtsKorrigieren(A, i)
    Ende Für
Ende Prozedur
```

ist. Insgesamt erhalten wir also für die Laufzeit von ErzeugeHalde:

$$\sum_{h=0}^{\lfloor \log n \rfloor - 1} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor \log n \rfloor - 1} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h})$$

Aus $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ erhält man durch Differenzieren und Multiplikation mit x die Gleichung $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$ und damit für $x = \frac{1}{2}$:

$$O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n).$$

Insgesamt benötigt Haldensortieren dann Zeit $O(n \log n)$.

Statt einer Sortierung nach maximaler Priorität kann natürlich auch eine Prioritätswarteschlange mit Sortierung nach minimaler Priorität auf symmetrische Weise implementiert werden. Die hier beschriebenen Datenstrukturen heißen in der Literatur auch *max-priority queue* basierend auf *max-heaps*, die symmetrischen *min-priority queue* basierend auf *min-heaps*.