Datenstrukturen und Algorithmen
Übung 3 – Sortieren

Musterlösung

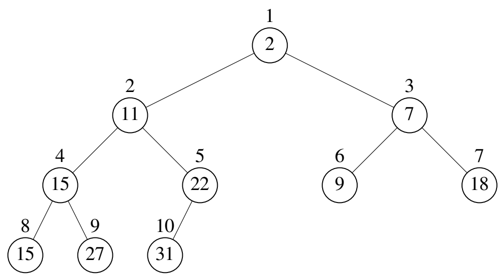
Theoretische Aufgaben

1. In der folgenden Tabelle ist ein Min-Heap in der üblichen impliziten Form gespeichert:

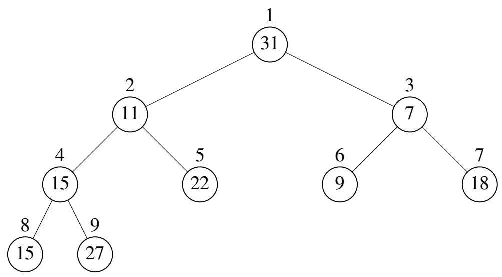
$$A = [2, 11, 7, 15, 22, 9, 18, 15, 27, 31].$$

Der Heap wird als *priority queue* verwendet. Wie sieht die Tabelle aus, nachdem HEAP-EXTRACT-MIN aufgerufen wurde, also das kleinste Element gelöscht und die Heap-Bedingung wieder hergestellt wurde? **(1 Punkt)**

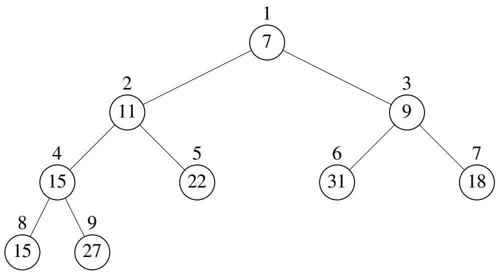
Initialer Heap:



Nach Tauschen/Löschen des minimalen Elements:



Nach Wiederherstellung der Heapeigenschaft:



Das Heap-Feld hat nun den Inhalt

$$\langle 7, 11, 9, 15, 22, 31, 18, 15, 27 \rangle$$

Hinweis: die Angabe des korrekten Feldes reicht, um den Punkt zu erlangen.

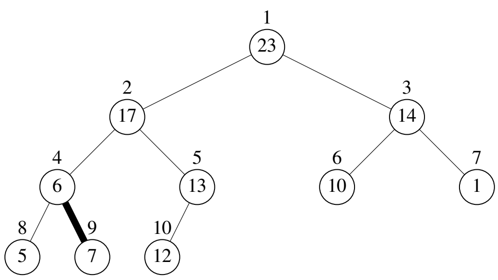
2. Beweisen Sie, dass in jedem Teilbaum eines Max-Heaps die Wurzel des Teilbaums den grössten Wert enthält, der in diesem Teilbaum vorkommt. Gehen Sie dafür nur von der Max-Heap-Eigenschaft aus. **(1 Punkt)**

Wir beweisen die Aussage per Induktion über die Höhe eines Max-Heap-Teilbaums.

Induktionsanfang Ein Teilbaum mit Höhe 1 enthält genau ein Element, daher gilt die Aussage trivialerweise.

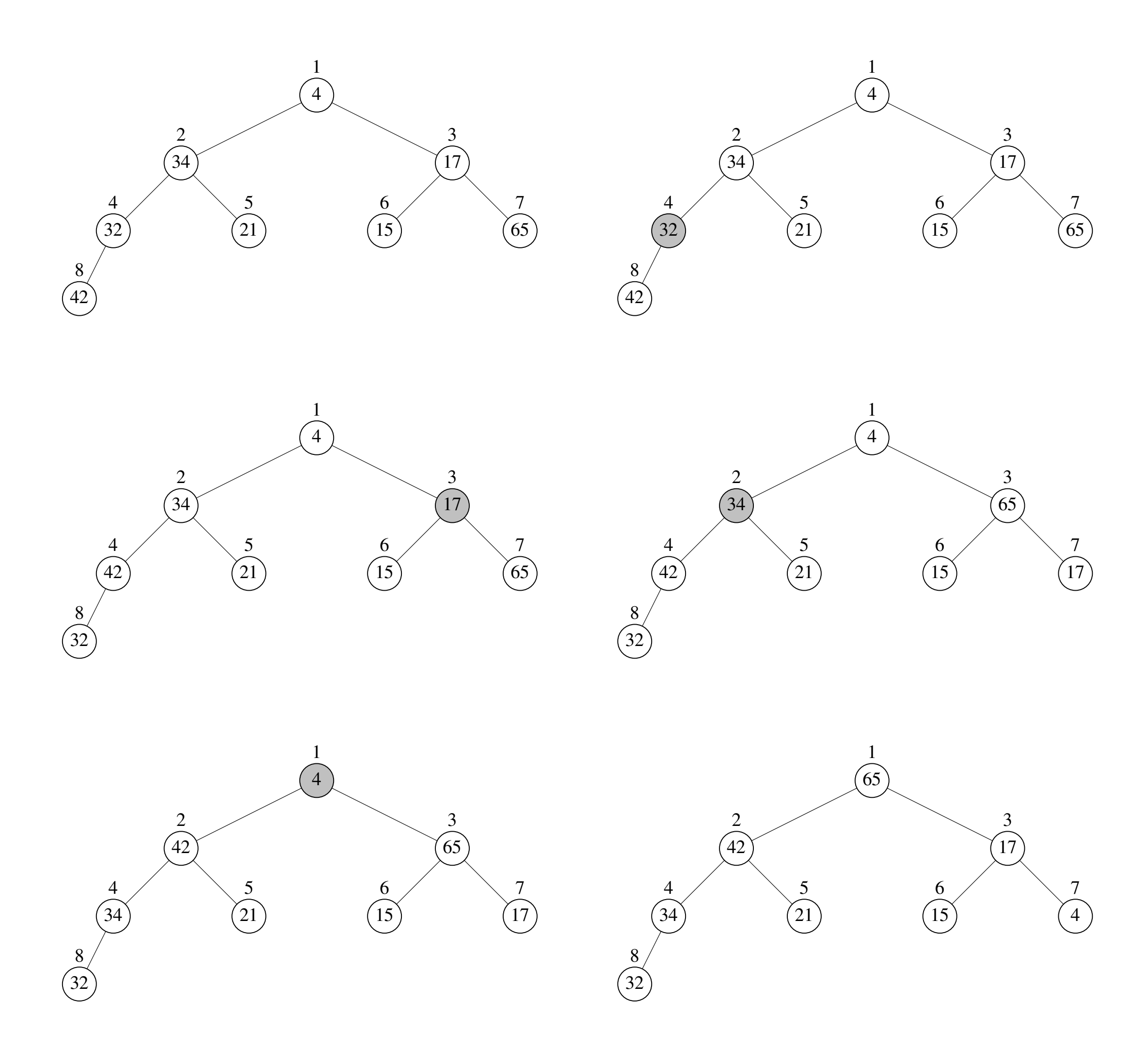
Induktionsschritt Nehmen wir an, dass die Ausage für alle Teilbäume mit Höhe $n' < n$ gilt, und zeigen Sie für Höhe n. Sei T ein Teilbaum eines Heaps mit Höhe n. Jeder Kindknoten k der Wurzel r ist Wurzel eines Teilbaums mit Höhe $n-1$ oder $n-2$, für die nach Induktionsannahme unsere Aussagt gilt. Also ist $A[k]$ jeweils der grösste Wert des jeweiligen Kind-Teilbaums. Da die Max-Heap-Eigenschaft im gesamten Heap gilt, ist nun auch jeweils $A[r] \geq A[k]$ - also ist $A[r]$ tatsächlich der grösste Wert des in r verwurzelten Teilbaums.

3. Ist das Feld mit den Werten $[23, 17, 14, 6, 13, 10, 1, 5, 7, 12]$ ein Max-Heap? Begründen Sie. **(1 Punkt)**



Nein, die Max-Heap-Eigenschaft ist an der Kante zwischen den Knoten 4 und 9 (Werte 6 und 7) verletzt.

4. Gegeben sei die Schlüsselfolge $[4, 34, 17, 32, 21, 15, 65, 42]$. Zeigen Sie den Ablauf der Funktion BUILD-MAX-HEAP ähnlich wie in Abbildung 6.3 im Buch. **(1 Punkt)**



5. Wie hoch ist die Laufzeit von HEAPSORT angewendet auf ein Feld A der Länge n, das bereits in aufsteigender Reihenfolge sortiert ist? Wie hoch ist sie, wenn das Feld A in absteigender Reihenfolge sortiert ist? Begründen Sie jeweils Ihre Aussagen. **(1 Punkt)**

Aufsteigend sortiert

- **Heap-Konstruktion**: $\mathcal{O}(n)$
- **Sortieren:** Wir tauschen $n-1$ mal das größte Element ans Ende eines schrumpfenden Heaps, Aufwand $\mathcal{O}(n \log n)$ (vgl. Vorlesung/Buch)
- **Total:** $\mathcal{O}(n) + (n \log n) = (n \log n)$

Absteigend sortiert

- **Heap-Konstruktion:** Ein absteigend sortieres Feld entspricht bereits einem Max-Heap. Wenn wir das ausnützen, können wir uns theoretisch [a] die Heapkonstruktion sparen: $\mathcal{O}(0)$
- **Sortieren:** Aufwand wieder $\mathcal{O}(n \log n)$ (s.o.)
- **Total:** $\mathcal{O}(n \log n)$

Eine in irgendeine Richtung sortierte Eingabe bringt uns also asymptotisch keine Vorteil, selbst wenn wir die Heap-Konstruktion auslassen könnten.

[a] Praktisch wäre das ohnehin selten sinnvoll — wenn wir bereits wissen, dass unsere Daten absteigend sortiert sind, könnten wir sie direkt in $\mathcal{O}(n)$ korrekt sortieren

6. Gegeben sei die Schlüsselfolge [19, 17, 3, 28, 60, 33, 20, 30, 2]. Geben Sie alle Aufrufe der Prozedur QUICKSORT und die Reihenfolge ihrer Abarbeitung an. Nehmen Sie an, dass das gesamte Feld sortiert werden soll. **(1 Punkt)**

QUICKSORT(A, 1, 9)	A=	19	17	3	28	60	33	20	30	2
Nach PARTITION(A, 1, 9)	A=	2	17	3	28	60	33	20	30	19
QUICKSORT(A, 1, 0)	A=	2	17	3	28	60	33	20	30	19
QUICKSORT(A, 2, 9)	A=	2	17	3	28	60	33	20	30	19
Nach PARTITION(A, 2, 9)	A=	2	17	3	19	60	33	20	30	28
QUICKSORT(A, 2, 3)	A=	2	17	3	19	60	33	20	30	28
Nach PARTITION(A, 2, 3)	A=	2	3	17	19	60	33	20	30	28
QUICKSORT(A, 2, 1)	A=	2	3	17	19	60	33	20	30	28
QUICKSORT(A, 3, 3)	A=	2	3	17	19	60	33	20	30	28
↪ *Ergebnis:*	A=	2	3	17	19	60	33	20	30	28
QUICKSORT(A, 5, 9)	A=	2	3	17	19	60	33	20	30	28
Nach PARTITION(A, 5, 9)	A=	2	3	17	19	20	28	60	30	33
QUICKSORT(A, 5, 5)	A=	2	3	17	19	20	28	60	30	33
QUICKSORT(A, 7, 9)	A=	2	3	17	19	20	28	60	30	33
Nach PARTITION(A, 7, 9)	A=	2	3	17	19	20	28	30	33	60
QUICKSORT(A, 7, 7)	A=	2	3	17	19	20	28	30	33	60
QUICKSORT(A, 9, 9)	A=	2	3	17	19	20	28	30	33	60
↪ *Ergebnis:*	A=	2	3	17	19	20	28	30	33	60
↪ *Ergebnis:*	A=	2	3	17	19	20	28	30	33	60
↪ *Ergebnis:*	A=	2	3	17	19	20	28	30	33	60
↪ *Ergebnis:*	A=	2	3	17	19	20	28	30	33	60

Korrekturhinweis: Die PARTITION- und *Ergebnis*-Zeilen sind nicht notwendig, sie dienen nur dem besseren Verständnis.

Praktische Aufgaben

Sortieren ist für die praktische Informatik von zentraler Bedeutung. In dieser Serie sollen Sie sich näher mit praktischen Implementationen von Sortieralgorithmen in Java befassen. Damit Sie sich auf das Wesentliche konzentrieren können, stellen wir Java-Code zur Verfügung, den Sie auf Ilias herunterladen können. Der Code enthält eine Implementation des QUICKSORT Algorithmus, der in der Vorlesung besprochen wurde.

In der Vorlesung wurden Sortieralgorithmen am Beispiel von ganzzahligen Feldern als Eingabedaten vorgestellt. Das Ziel einer praktischen Implementation soll aber sein, dass beliebige Daten sortiert werden können, solange es möglich ist, Elemente paarweise zu vergleichen. Der Java Code erreicht dies mittels zwei Konzepten: Erstens werden sogenannte *generische* Klassen und Methoden verwendet, und zweitens werden *Comparator* Objekte definiert, um Daten paarweise zu vergleichen. Um sich für diese Aufgabe vorzubereiten, sollten Sie sich mit diesen Konzepten bekannt machen. Dazu bietet sich beispielhaft folgende Beschreibung an, welche beide Themen beschreibt:

http://www.theserverside.de/java-generics-generische-methoden-klassen-und-interfaces/.
Nehmen Sie sich Zeit, die Anwendung der Konzepte in unseren bereitgestellten Java Code zu studieren. Bearbeiten Sie dann folgende Aufgaben:

1. Erstellen Sie eine neue Klasse `NameVornameComparator`, welche zwei Objekte der Klasse `StudentIn` lexikographisch hinsichtlich Name und Vorname (in dieser Reihenfolge) vergleichen kann.

   Bsp: 'Meier Anna' ist lexikographisch kleiner als 'Meier Beat'.

   Orientieren Sie sich an der bereits fertigen Klasse `MatrikelNrComparator`. Beachten Sie, dass Ihr `NameVornameComparator` das Interface `java.util.Comparator` aus dem Java API implementieren muss. Testen Sie Ihre Implementation mittels des Programms `MiniTestApp.java`. Geben Sie Ihren Code für `NameVornameComparator` sowie die Ausgabe von `MiniTestApp.java` ab. **(2 Punkte)**

```java
/**
 * File: NameVornameComparator.java
 *
 * Klasse zum Vergleichen zweier Objekte (Records) vom Typ StudentIn
 * lexikographisch bezüglich Name und Vorname
 *
 */

public class NameVornameComparator implements java.util.Comparator<StudentIn>
{
  public int compare(StudentIn a, StudentIn b) {
    int res = a.getName().compareTo(b.getName());
    if (res != 0)
        return res;
    return a.getVorname().compareTo(b.getVorname());
  }

}
```

2. Versuchen Sie nun das Programm `SortTestApp.java` auszuführen. Dieses Programm erzeugt verschieden grosse Eingabefelder mit zufälligen Daten. Die Grösse der Eingabefelder geht bis zu mehr als einer Million Elemente. Die Felder werden dann *zwei Mal* mit QUICKSORT sortiert. Das heisst, zuerst wird die unsortierte Eingabe sortiert, und dann wird versucht, die bereits sortierten Daten noch einmal zu sortieren. Was beobachten Sie bei der Ausführung des Programms? Warum unterscheidet sich die Laufzeit des jeweils ersten und zweiten Sortiervorgangs? Finden Sie heraus und erklären Sie, was ein *Stack Overflow Error* ist und warum er in diesem Beispiel auftritt. Beschreiben Sie Ihre Erkenntnisse in ein paar Sätzen. **(1 Punkt)**

   Beim (rekursiven) Aufruf von Funktionen werden auf dem Call-Stack Werte abgelegt, die erst beim Zurückkehren aus diesem Aufruf wieder „entfernt" werden. Die Größe des Stacks hängt somit linear mit der Rekursionstiefe zusammen. In unserer Implementierung von QUICKSORT ist bei sortierter Eingabe die Rekursionstiefe in $\Theta(n)$. Da der Stack-Platz und damit die Rekursionstiefe begrenzt ist, stürzt das Programm beim Erreichen des Limits ab.

   Im Optimalfall wird nur eine Stacktiefe von $\log n$ benötigt.

   Hinweis: In Sprachen/Laufzeitumgebungen mit Unterstützung für Endrekursion (*Tail* *recursion*) lässt sich das Problem auch elegant umgehen, indem man die kleinere Array-Hälfte per regulärem rekursivem Aufruf sortiert, und die größere per endrekursivem Aufruf.

3. Modifizieren Sie den QUICKSORT Algorithmus so, dass das Problem vermieden wird. Geben Sie den Code des modifizierten QUICKSORT ab. Stellen Sie sicher, dass Ihr Algorithmus korrekt funktioniert, indem Sie ihn mit `MiniTestApp.java` testen. Hinweis: Verwenden Sie *randomisierten* QUICKSORT. **(1 Punkt)**

```java
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Random;

/**
 * File: RandomizedQuickSort.java
 */
public class RandomizedQuickSort {
    /**
     * expect pivat at 'right' element
     */
    public static <T> int partition(ArrayList<T> array, int left, int right, Comparator<T> comp)
    {
        T temp;         // temporaere Hilfsvariable zum swappen
        // *** 1. Pivotelement selektieren:
        T pivot = array.get(right);

        // *** 2. Aufteilung in Subsequenzen durchfuehren:
        int l = left-1;
        int r = right;
        do {
            do l++; while (comp.compare(array.get(l),pivot)<0);
            do r--; while (r>l && comp.compare(array.get(r),pivot)>0);

            // swap(array, l, r):
            temp = array.get(l);
            array.set(l, array.get(r));
            array.set(r, temp);
        } while (r>l);

        array.set(r, array.get(l));    // Korrektur: einmal zuviel getauscht

        // Pivotelement in sortierte Position bringen:
        array.set(l, pivot);
        array.set(right, temp);
        return l;
    }

    public static <T> int randomized_partition(ArrayList<T> array, int left, int right, Comparator<T> comp) {
```

```java
        Random random = new Random(); // besser wäre, dieses Objekt nur 1x zu erstellen
        int pivot = left + random.nextInt(right-left + 1);
        T temp = array.get(right);
        array.set(right, array.get(pivot));
        array.set(pivot, temp);
        return partition(array, left, right, comp);
    }

    /**
     * Sortiert den array zwischen den Indizes left und right (mit Grenzen).
     */
    public static <T> void quickSort(ArrayList<T> array, int left, int right, Comparator<T> comp)
    {
        int l = left-1;
        int r = right;
        if (right>left) {  // Abbruchbedingung der Rekursion
            l = randomized_partition(array, left, right, comp);

            // *** 3. Rekursiv die beiden Subarrays sortieren:
            quickSort(array, left, l-1, comp);
            quickSort(array, l+1, right, comp);
        }
    }

}
```

Vergessen Sie nicht, Ihren Sourcecode innerhalb der Deadline über die Ilias-Aufgabenseite einzureichen.