Datenstrukturen und Algorithmen, SS 2019
Prof. Dr. P. Rossmanith
S. Dollase, J. Dreier, H. Lotze

Datum: 23.5.2019

Übungsblatt 07

Aufgabe T24

Analysieren Sie die Laufzeit von Quicksort für den Spezialfall, dass *alle* Elemente identisch sind. Wie verhalten sich Insertionsort und Heapsort in diesem Fall?

Aufgabe T25

	Quicksort	Heapsort	Mergesort	Insertion-Sort	Straight-Radix	Radix-Exchange
In-place?						
Stabil?						
Laufzeit (worst-case)						
Laufzeit (Durchschnitt)						
Vergleichsbasiert?						

Beantworten Sie die Fragen für alle Sortierverfahren. Gehen Sie davon aus, dass ein Vergleich in konstanter Zeit durchgeführt wird und die Anzahl der zu sortierenden Elemente n beträgt. Für Laufzeiten tragen Sie eine Funktion $f(n)$ in die Tabelle ein, um eine Laufzeit von $O(f(n))$ auszudrücken.

Aufgabe T26

Eine *Prioritätswarteschlange* ist eine Datenstruktur, welche folgende Operationen erlaubt:

1. *extract-min*: Gebe das kleinste gespeicherte Element zurück und lösche es aus der Warteschlange.
2. *insert*(x): Füge das Element x ein.

Wie können diese Operationen mithilfe von Heaps umgesetzt werden? Was ist die Laufzeit? Sie können davon ausgehen, dass anfangs bekannt ist, wie viele Elemente sich höchstens gleichzeitig in der Warteschlange befinden können.
Welche Vor- und Nachteile hat ein Heap gegenüber einer Implementierung basierend auf balancierten Suchbäumen?

Aufgabe H19 (7+3 Punkte)

Wir betrachten die paarweise verschiedenen drei Schlüssel A, B und C, welche wir in einem Array x der Größe drei in einer unbekannten Reihenfolge vorfinden.
Ein Algorithmus versucht herauszufinden, welche der möglichen sechs Permutationen vorliegt indem er zunächst den Vergleich $x[0] \leq x[1]$, dann den Vergleich $x[0] \leq x[2]$ und schließlich – falls noch nötig – den Vergleich $x[1] \leq x[2]$ durchführt.
a) Ist es möglich auf diese Weise die richtige Permutation zu finden? Wie sieht der entsprechende Vergleichsbaum aus?
b) Wie viele Vergleiche würde ein auf dieser Methode basierender Sortieralgorithmus im Durchschnitt verwenden und warum?

Aufgabe H20 (3+4+3 Punkte)

In der Vorlesung wurde Quicksort auch iterativ mit Hilfe eines expliziten Stacks implementiert. Da Speicherverbrauch immer ein wichtiger Faktor ist, sind wir an der maximalen Höhe dieses Stacks interessiert:

a) Finden Sie ein Beispiel, in welchem die gegebene Quicksort-Implementation $\Omega(n)$ Paare gleichzeitig im Stack speichert.

b) Überlegen Sie dann, wie der Algorithmus abgeändert werden kann, um diesen schmerzhaften Speicherverbrauch auf $O(\log(n))$ zu senken.

c) Um immer korrekt zu sein, bestimmt die vorliegende Implementation zunächst das minimale Element $a[\min]$ des Eingabearrays. Sie vertauscht es mit dem ersten Element $a[0]$ desselben und die verbleibenden Elemente $a[1], \ldots, a[a.length - 1]$ werden dann wie gehabt sortiert. Die Analyse von Quicksort setzt jedoch voraus, dass jede mögliche Permutation der zu sortierenden Schlüssel gleich wahrscheinlich ist. Sind nach der obigen Veränderung der Eingabe alle Permutationen der verbleibenden Elemente immer noch gleich wahrscheinlich?

```
public void quicksort(int a[]) {
  Stack<Pair<Integer, Integer>> stack = new Stack<>();
  stack.push(new Pair<Integer, Integer>(1, a.length - 1));
  int min = 0;
  for(int i = 1; i < a.length; i++) if(a[i] < a[min]) min = i;
  int t = a[0]; a[0] = a[min]; a[min] = t;
  while(!stack.isempty()) {
    Pair<Integer, Integer> p = stack.pop();
    int l = p.first(), r = p.second();
    int i = l - 1, j = r, pivot = j;
    do {
      do { i++; } while(a[i] < a[pivot]);
      do { j--; } while(a[j] > a[pivot]);
      t = a[i]; a[i] = a[j]; a[j] = t;
    } while(i < j);
    a[j] = a[i]; a[i] = a[r]; a[r] = t;
    if(r - i > 1) stack.push(new Pair<Integer, Integer>(i + 1, r));
    if(i - l > 1) stack.push(new Pair<Integer, Integer>(l, i - 1));
  }
}
```

Aufgabe H21 (10 Punkte)

Führen Sie auf folgendem Graphen eine Tiefensuche aus. Beginnen Sie auf dem oberen der beiden ganz linken Knoten. Notieren Sie die *discovery*- und *finish*-Zeiten. Benennen Sie die Baum-, Quer-, Vorwärts- und Rückwärtskanten.