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

Datum: 23.5.2019

Übungsblatt mit Lösungen 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?

Lösungsvorschlag

Folgender Codeabschnitt partitioniert das Eingabearray.

```
do
    do l := l + 1 while a[l] < p;
    do r := r - 1 while p < a[r];
    vertausche a[l] und a[r];
while l < r;
```

Wenn alle Elemente gleich sind, so werden `l` und `r` in jeder äußeren Schleife genau um eins erhöht. Somit wird das Array in zwei gleichgroße Hälften partitioniert. Quicksort braucht also $\Theta(n \log n)$.
Insertionsort und Heapsort sind in linearer Zeit fertig.

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.

Lösungsvorschlag

	Quicksort	Heapsort	Mergesort	Insertion-Sort	Straight-Radix	Radix-Exchange
in-place?	??	J	N	J	N	??
stabil?	N	N	J	J	J	N
Laufzeit (worst-case)	n^2	$n \log n$	$n \log n$	n^2	nw	nw
Laufzeit (Durchschnitt)	$n \log n$	$n \log n$	$n \log n$	n^2	nw	nw
vergleichsbasiert?	J	J	J	J	N	N

Bei den Radix-Sortierverfahren bezeichne w die Wortlänge. Quicksort und Radix-Exchange-Sort sind wegen des benötigten Stacks nicht in-place. Beide lassen sich jedoch so implementieren, dass der Stack in rekursiven Funktionsaufrufen „versteckt“ wird, während jeder einzelne Aufruf nur konstant viel Platz benötigt. Bei Sortierverfahren sagen manche, dass sie nicht in-place sind, wenn sie $\Omega(n)$ viel Platz zusätzlich brauchen, andere, schon bei mehr als konstant vielem Zusatzplatz.

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?

Lösungsvorschlag

Wir können beide Operationen genauso umsetzen, wie sie auch in Heapsort verwendet werden. Ein Element wird eingefügt, indem es an das Ende des Arrays angefügt wird und anschließend aufsteigt. Das kleinste Element finden wir in der Wurzel und können es durch das letzte Element im Array ersetzen und anschließend absteigen lassen. Beide Operationen benötigen nur $O(\log n)$ Zeit.

Balancierte Suchbäume benötigen ebenfalls $O(\log n)$ Zeit für beide Operationen. Sie brauchen aber mehr Speicherplatz und die Implementierung ist deutlich komplizierter und damit wahrscheinlich auch ein bisschen langsamer. Auf der anderen Seite sind Suchbäume „pointerbasiert“ (nicht die Elemente selbst werden bewegt, sondern nur Zeiger oder Referenzen auf sie) und die eigentlichen Elemente bewegen sich nicht. Ein Heap ist aber in Wirklichkeit ein Array und der Index eines Elements ändert sich natürlich, wenn der Heap verändert wird.

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.

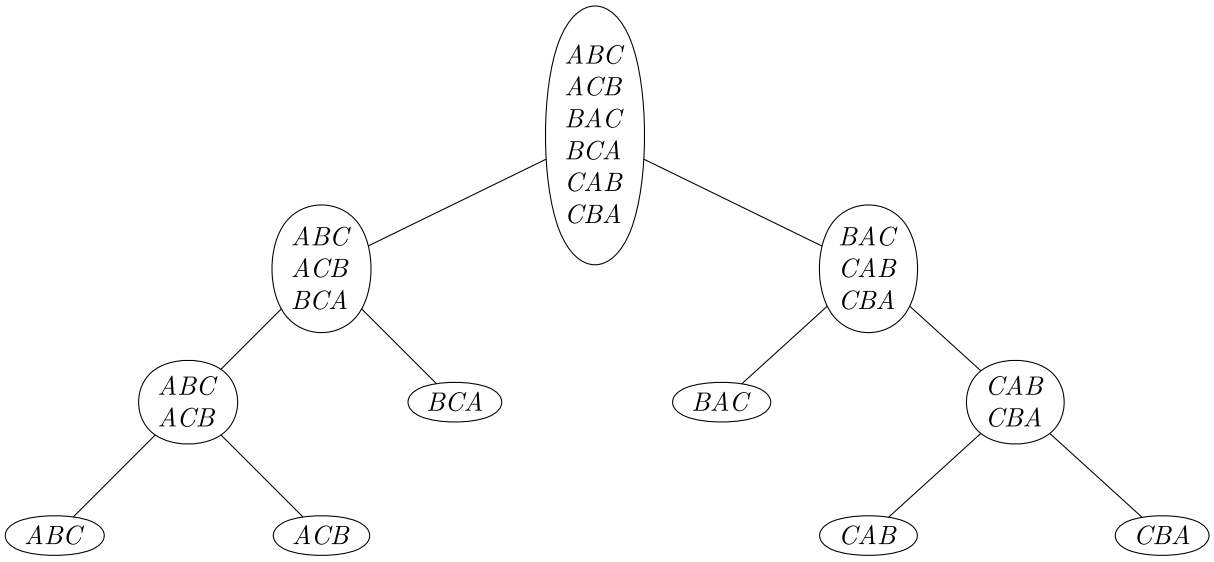
Ein Algorithmus versucht herauszufinden, welche der möglichen sechs Permutationen vorliegt indem er zunächst den Vergleich $x[0] \leq x[1]$, dann den Vergleich $x[0] \leq x[2]$ und schließlich – falls noch nötig – den Vergleich $x[1] \leq x[2]$ durchführt.

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?

Lösungsvorschlag

Die einzelnen Vergleiche partitionieren die Menge der möglichen Permutationen wie in folgender Abbildung dargestellt.



Man sieht, dass tatsächlich ein Vergleichsbaum entsteht, der nur einzelne Permutationen für jedes Blatt übriglässt. Daher ist es in der Tat möglich, mit diesen Vergleichen die richtige Permutation zu bestimmen.

Wenn wir annehmen, dass jede Permutation mit gleicher Wahrscheinlichkeit, das heißt mit einer Wahrscheinlichkeit von 1/6, vorkommt, dann ist die erwartete Anzahl von Vergleichen $4 \cdot \frac{3}{6} + 2 \cdot \frac{2}{6} = \frac{8}{3}$.

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));
  }
}
```

Lösungsvorschlag

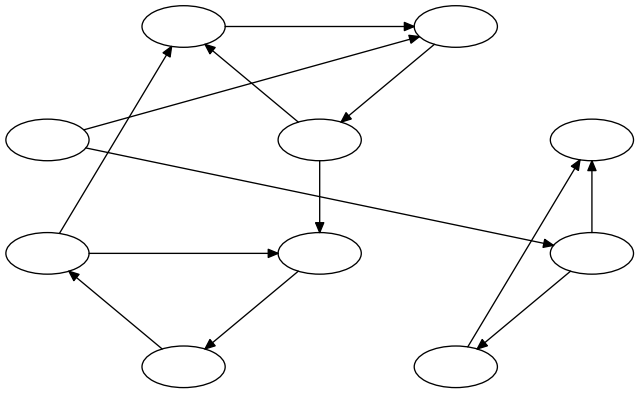
Das oberste Element des Stacks wird sofort am Anfang der Schleife entfernt, interessant ist also das erste der beiden eingefügten Intervalle. Bringen wir den Algorithmus dazu, als erstes Element ein sehr kleines Intervall (etwa der Größe eins) auf den Stack zu legen und dann gezwungenermaßen ein sehr großes Intervall als zweites, so benötigt Quicksort $\Omega(n)$ Iterationen der äußeren Schleife—eine Beispielinstanz dafür wäre schlicht ein bereits aufsteigend sortiertes Array. Da nun in jeder Iteration ein Element auf dem Stack verbleibt, erreicht der Stack eine Größe von $\Omega(n)$.

Dieses Verhalten kann verhindert werden, indem man immer das kürzere Intervall oben auf den Stack legt: so kann der Stack maximal eine Größe von $O(\log n)$ erreichen.

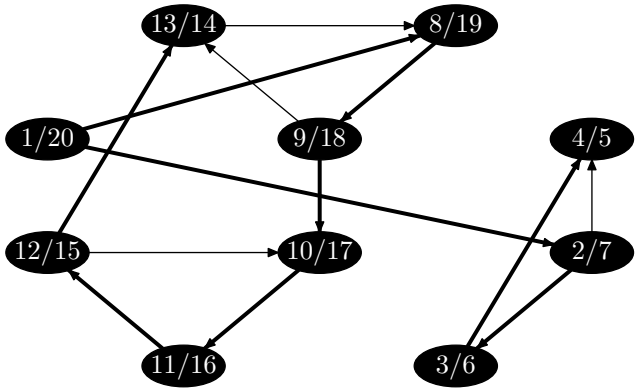
Nehmen wir o.b.d.A. an, die Eingabe bestehe aus einer beliebigen Permutation der Elemente $1 \dots n$ und alle diese Permutationen haben die gleiche Wahrscheinlichkeit. Ein einfaches Zählargument zeigt, dass die anfängliche Vertauschung Permutationen auf den Elementen $2 \dots n$ erzeugt, die wieder alle gleich wahrscheinlich sind: für jede der $(n-1)!$ vielen Permutationen können n verschiedene Permutation der Elemente $1 \dots n$ erzeugt werden, indem ein Element durch die 1 ersetzt und anschließend den anderen vorangestellt wird. Da dies niemals zwei gleiche Permutationen erzeugt, erhalten wir $n(n-1)! = n!$ viele Permutationen auf n Elementen—umgekehrt ist damit die Wahrscheinlichkeit, eine konkrete Permutation der Elemente $2 \dots n$ zu erhalten, $n\frac{1}{n!} = \frac{1}{(n-1)!}$

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.



Lösungsvorschlag



Die meisten Kanten sind Baumkanten. $2-4$ und $9-13$ sind Vorwärtskanten, $13-8$ und $12-10$ sind Rückwärtskanten. Es gibt keine Querkante.