Datenstrukturen und Algorithmen, SS 2024
Prof. Dr. P. Rossmanith
D. Cloerkes, M. Gehnen, D. Mock, N. Speetzen

Abgabe: 16.05.2024

Übungsblatt mit Lösungen 05

Aufgabe T15

Gegeben ist folgendes Array: $[8, 6, 3, 7, 4, 2, 20, -45]$.

(a) Wie viele Inversionen hat es?

(b) Wie viele Läufe hat es?

(c) Was macht Quicksort in der ersten Partitionierungsphase daraus?

(d) Was macht Mergesort in der letzten Mischphase?

(e) Wie sieht das Array nach der Konstruktion des Max-Heaps aus, wenn Heapsort angewandt wird? Wie sieht es nach der ersten Bubble-Operation aus?

Lösungsvorschlag

(a) Gesucht ist die Anzahl der Index-Paare (i, j) mit $i < j$, so dass der Wert an Stelle i größer ist als der Wert an Stelle j. Für die gegebene Zahlenfolge sind das genau die Paare

$$\begin{aligned} &\{(1,2),(1,3),(1,4),(1,5),(1,6),(1,8),(2,3),(2,5),(2,6),(2,8)\} \\ \cup\ &\{(3,6),(3,8),(4,5),(4,6),(4,8),(5,6),(5,8),(6,8),(7,8)\}, \end{aligned}$$

also hat sie 19 Inversionen.

(b) Das Array hat 6 Läufe:

- 8
- 6
- $3, 7$
- 4
- $2, 20$
- -45

(c)
- 8 wird als Pivotelement gewählt
- Vertauschen von 20 und -45: $[8, 6, 3, 7, 4, 2, -45, 20]$
- Vertauschen von -45 und 20: $[8, 6, 3, 7, 4, 2, 20, -45]$
- Abbruch der Schleife: $l > r$ (l zeigt auf -45, r auf 20)
- Wert an Stelle l rückt nach vorne, Wert an Stelle r rückt an Stelle l und das Pivotelement rückt an Stelle r: $[-45, 6, 3, 7, 4, 2, 8, 20]$

(d) In der letzten Mischphase sind die linke und die rechte Teilfolge bereits sortiert: $3, 6, 7, 8, -45, 2, 4, 20$. Gemischt werden sollen $[3, 6, 7, 8]$ und $[-45, 2, 4, 20]$.

- $3 > -45 \Rightarrow b[0] = -45$
- $3 > 2 \Rightarrow b[1] = 2$
- $3 \leq 4 \Rightarrow b[2] = 3$
- $6 > 4 \Rightarrow b[3] = 4$
- $6 \leq 20 \Rightarrow b[4] = 6$
- $7 \leq 20 \Rightarrow b[5] = 7$
- $8 \leq 20 \Rightarrow b[6] = 8$
- linker Counter hat linke Teilfolge verlassen $\Rightarrow b[7] = 20$

(e) In der langsamen Methode der Vorlesung wird zuerst der Heap durch sequentielles Hinzufügen der Elemente erstellt.

- Es werden nach einander 8,3,6 in den Heap hinzugefügt. Sie erhalten die Heapeigenschaft, ohne das gebubblet werden müsste: $[8, 3, 6]$.
- Das Hinzufügen von 7 verletzt die Heapeigenschaft. Bubbleup ergibt $[8, 7, 3, 6]$.
- Hinzufügen von 4 und 2 ist ok: $[8, 7, 3, 6, 4, 2]$.
- Bei 20 muss zweimal bubble-up ausgeführt werden: $[20, 7, 8, 6, 4, 2, 3]$
- -45 hinzufügen, Heap erstellt: $[20, 7, 8, 6, 4, 2, 3, -45]$

Nun wird das oberste (also maximale Element) aus dem Heap entnommen und durch das hinterste Element ersetzt. Dieses verletzt die Heapeigenschaft und wird durch bubbledown an seinen richtigen Platz befördert:

- 20 wird durch -45 ersetzt: $[-45, 7, 8, 6, 4, 2, 3|20]$
- Zweimal Bubbledown: $[7, 6, 8, -45, 2, 3|20]$

Aufgabe T16

Stellen Sie sich vor, Sie befinden sich in folgender Situation: Sie wollen ein Terabyte (1024 Gigabyte) an Strings aufsteigend ordnen. Alle Strings sind genau 16 Byte groß. Dafür steht Ihnen ein Computer mit fünf Festplatten, die jeweils ein Terabyte groß sind, zur Verfügung. Eine der Festplatten enthält die Strings und ist *read-only*. Die weiteren vier sind frei benutzbar. Man kann von verschiedenen Festplatten parallel schreiben/lesen. Nehmen Sie an, der Computer hat 16 Gigabyte an Arbeitsspeicher, den Sie frei benutzen können.

(a) Wie sortiert man die Strings am besten, sodass am Ende eine der Festplatten eine Liste aller Strings in aufsteigender Reihenfolge enthält?

(b) Gehen Sie davon aus, dass t_1 die Zeit ist, die man braucht, um 16GB an Strings zu sortieren, t_2 um ein Terabyte von einer Festplatte zu lesen, und t_3 die Zeit, um ein Terabyte auf eine Festplatte zu schreiben. Geben Sie eine möglichst gute Abschätzung der Laufzeit Ihres Verfahrens mit Hilfe dieser Werte.

(c) Was denken Sie sind realistische Werte für t_1, t_2 und t_3?

Lösungsvorschlag

(a) Ein gutes Verfahren ist Bottom-up-Mergesort:

Wir numerieren die schreibbaren Festplatten von 1 bis 4. Wir lesen 16 GB von der *read-only* Festplatte, sortieren diese Unterliste im Speicher, und schreiben diesen abwechselnd auf Festplatte 1 und 2 bis wir das Ende der *read-only* Festplatte erreichen. Jetzt enthalten Festplatten 1 und 2 genau 512 GB an Strings in jeweils 16 GB großen sortierten Blöcken.

Wir nehmen dann einen 16 GB Block aus Festplatte 1 und einen 16 GB Block aus Festplatte zwei und durch *mischen* schreiben wir abwechselnd auf Festplatte 3 und 4 bis wir alle Blöcke gemischt haben. Jetzt enthalten Festplatten 3 und 4 je 512 GB an Strings in jeweils 32 GB großen sortierten Blöcken. Festplatten 1 und 2 können wir jetzt problemlos überschreiben.

Wir wiederholen jetzt diese Prozedur, wobei wir in jedem Schritt die Blockgröße verdoppeln, bis am Ende eine finale *merge*-Operation von zwei 512 GB großen Blöcken zu dem erwünschten Ergebnis führt.

(b) Der erste Schritt um die zwei Listen aus geordneten 16 GB Blöcken zu schreiben braucht $1\text{TB}/16\text{GB} \cdot t_1 + t_2 + t_3 \approx 64t_1 + t_2 + t_3$. Beim Mischen können wir von zwei Festplatten parallel lesen, also braucht jede Mischoperation $t_2/2 + t_3$ Zeit. Die Blockgrößen wachsen exponentiell, also $16, 32, 64, 128, 256, 512, 1024$ und es gibt somit sechs *merges*. Somit ist die Gesamtlaufzeit ungefähr $64t_1 + 4t_2 + 7t_3$.

(c) Durch Testen kann man herausfinden, dass 16 GB zu sortieren etwa 10min dauert.

Eine andere möglichkeit wäre es die Formel

$$C_n = 1.4n \cdot \log n$$

heranzuziehen (etwa vier Instruktionen pro Schleifendurchlauf bei Quicksort). Wenn wir schätzen, dass ein Vergleich $15\,\text{ns} = 1.5{\cdot}10^{-8}\,\text{s}$ dauert, bekommen wir mit $n = 16\,\text{GB}/16\,\text{byte} = 10^9$

$$C_n = 1.4 \cdot 10^9 \cdot \log 10^9 \cdot 1.5 \cdot 10^{-8}\,\text{s} \approx 7\,\text{m}.$$

Wählen wir $t_1 = 7\,\text{m} = 0.12\,\text{h}$

Mechanische Festplatten mit 7200 rpm, wie sie häufig verbaut werden, bieten Lesegeschwindigkeiten von ungefähr 128 MB/s und Schreibgeschwindigkeiten von ungefähr 120 MB/s. Damit ergibt sich für $t_2 = 1\,\text{TB}/128\,\text{MB/s} = 2.1\,\text{h}$ und für $t_3 = 2.3\,\text{h}$

Die Gesamtzeit nach der Formel aus b) ist somit ungefähr 32.5 Stunden um alle Strings zu sortieren.

Aufgabe T17

Beweisen Sie, dass es keinen vergleichsbasierten Sortieralgorithmus gibt, welcher ein beliebiges Array der Größe fünf mit nur sechs Vergleichen sortieren kann.

Lösungsvorschlag

In der Vorlesung wurde bewiesen, dass wir mindestens $\log(5!) = \log(120) \approx 6.9$ Vergleiche brauchen.

Aufgabe T18

Wir untersuchen in dieser Aufgabe, wie man binäre Suchbäume zum Sortieren verwenden kann.

(a) Entwerfen Sie einen Algorithmus, der in $O(n)$ Schritten die n Schlüssel aus einem binären Suchbaum in aufsteigender Reihenfolge ausgeben kann.

(b) Beschreiben Sie, wie man mit binären Suchbäumen möglichst effizient sortieren kann.

(c) Wie schnell ist Ihr Verfahren aus (b) im worst-case mit gewöhnlichen binären Suchbäumen, mit AVL-Bäumen und mit Splay-Bäumen?

(d) Beantworten Sie Frage (c) auch für den Spezialfall, dass die Eingabe bereits sortiert bzw. umgekehrt sortiert ist.

Lösungsvorschlag

a) Wir nutzen eine Inorder-Traversierung des Baumes:

   Falls der aktuelle Knoten ein linkes Kind hat, steigen wir zu diesem hinab und gehen für diesen nach gleichem Prinzip vor. Anschließend geben wir den Schlüssel des aktuellen Knotens aus. Schließlich überprüfen wir, ob es ein rechtes Kind gibt, und falls ja, gehen wir auch für dieses nach gleichem Prinzip vor.

   Dabei wird jeder Knoten genau einmal besucht, wobei ein Besuch konstante Laufzeit hat. Die Laufzeit des Algorithmus is somit $O(n)$.

b) Man kann einen binären Suchbaum zum Sortieren verwenden, indem man alle Schlüssel der Reihe nach einfügt und anschließend mithilfe der Inorder-Traversierung ausgibt. Damit die Einfügeoperationen möglichst effizient durchgeführt werden können und der Baum nicht degeneriert, können wir dafür zum Beispiel AVL-Bäume verwenden.

c)
   - Gewöhnliche binäre Suchbäume: Im schlimmsten Fall haben wir eine bereits sortierte (oder umgekehrt sortierte) Folge von Zahlen, sodass die Höhe des Baumes linear in der Anzahl der Schlüssel ist. Damit braucht das Einfügen lineare Zeit und die gesamte Laufzeit ist $O(n^2)$.
   - AVL-Bäume: Hier ist die Laufzeit für Einfügeoperationen $O(\log(n))$, da sowohl die Höhe des Baumes als auch das Rebalancieren im worst-case logarithmisch in der Anzahl der Schlüssel sind. Deshalb ergibt sich beim Einfügen von n Schlüsseln eine Laufzeit von $O(n \log(n))$.
   - Splay-Bäume: Amortisiert ist auch hier die Laufzeit für Einfügeoperationen $O(\log(n))$. Deshalb ergibt sich insgesamt wieder eine Laufzeit von $O(n \log(n))$.

d) Falls die Folge (umgekehrt) sortiert ist, ergeben sich folgende Fälle:

- Gewöhnliche binäre Suchbäume: Der nächste Schlüssel wird im sortierten Fall immer rechts unten (bzw. im umgekehrt sortierten Fall immer links unten) hinzugefügt, sodass sich ein degenerierter Baum mit linearer Höhe bildet. Damit braucht eine Einfügeoperationen lineare Zeit und $\Omega(n)$ viele Operationen brauchen $\Omega(n)$ Zeit. Damit ist die gesamte Laufzeit in $\Theta(n^2)$.
- AVL-Bäume: Da es sich hier um balancierte Bäume handelt, ist die Länge eines Pfades zwischen Wurzel und einem beliebigen Blatt immer $\Omega(\log(n))$. Da man beim Einfügen eines neuen Schlüssels bis zu einem Blatt hinabsteigt, ergibt sich also in jedem Fall eine Laufzeit von $\Theta(n \log(n))$.
- Splay-Bäume: Der nächste Schlüssel wird im sortierten Fall immer rechts von der Wurzel eingefügt und anschließend mit einem einfachen *zag* hochrotiert. Das Einfügen eines Elementes geschieht also in konstanter Zeit. Im umgekehrt sortierten Fall werden die Schlüssel analog links von der Wurzel eingefügt und mit einem *zig* rotiert. Insgesamt ergibt sich also eine Laufzeit von nur $\Theta(n)$.

Aufgabe H16 (8+8+8 Punkte)

(a) Sortieren Sie das folgende Array mithilfe von Quicksort aus der Vorlesung. Geben Sie dazu das Array nach jeder Partition-Operation an und markieren Sie das Pivot-Element und den Bereich, auf dem die Partition-Operation angewandt wurde.

$$[6, 8, 3, 9, 2, 1, 4, 5, 7]$$

(b) Sortieren Sie das folgende Array mithilfe von Mergesort aus der Vorlesung. Geben Sie dazu das Array nach jeder Merge-Operation an und markieren Sie den Bereich, auf dem die Merge-Operation angewandt wurde.

$$[3, 5, 2, 6, 1, 7, 4]$$

(c) Sortieren Sie das folgende Array mithilfe von Heapsort aus der Vorlesung. Geben Sie dazu das Array nach der jeder Bubble-Operation an. Sie können zusätzlich den Heap (der übrigen Elemente) grafisch angeben.

$$[3, 5, 2, 6, 1, 7, 4]$$

Lösungsvorschlag

(a)

$$[\underline{6, 8, 2, 9, 3, 1, 5, 4, 7}]$$
$$[\underline{1, 4, 2, 5, 3}, 6, 9, 8, 7]$$
$$[1, \underline{4, 2, 5, 3}, 6, 9, 8, 7]$$
$$[1, \underline{3, 2}, 4, 5, 6, 9, 8, 7]$$
$$[1, \underline{2}, 3, 4, 5, 6, 9, 8, 7]$$
$$[1, 2, 3, 4, 5, 6, \underline{9, 8, 7}]$$
$$[1, 2, 3, 4, 5, 6, \underline{7, 8}, 9]$$
$$[1, 2, 3, 4, 5, 6, 7, \underline{8}, 9]$$

(b) Die Sortierung von einelementigen Bereichen wurde weggelassen.

$$[3, 5, 2, 6, 1, 7, 4]$$
$$[3, \underline{2, 5}, 6, 1, 7, 4]$$
$$[\underline{2, 3, 5}, 6, 1, 7, 4]$$
$$[2, 3, 5, \underline{1, 6}, 7, 4]$$
$$[2, 3, 5, 1, 6, \underline{4, 7}]$$
$$[2, 3, 5, \underline{1, 4, 6, 7}]$$
$$[\underline{1, 2, 3, 4, 5, 6, 7}]$$

(c)

$$[5, 3|2, 6, 1, 7, 4]$$
$$[5, 3, 2|6, 1, 7, 4]$$
$$[6, 5, 2, 3|1, 7, 4]$$
$$[6, 5, 2, 3, 1|7, 4]$$
$$[7, 5, 6, 3, 1, 2|4]$$

Nach Build-heap (langsam, mit einzelnem Hinzufügen):

$$[7, 5, 6, 3, 1, 2, 4]$$
$$[6, 5, 4, 3, 1, 2|7]$$
$$[5, 3, 4, 2, 1|6, 7]$$
$$[4, 3, 1, 2|5, 6, 7]$$
$$[3, 2, 1|4, 5, 6, 7]$$
$$[2, 1|3, 4, 5, 6, 7]$$
$$[1, 2, 3, 4, 5, 6, 7]$$

Aufgabe H17 (10 Punkte)
Armin und Christoph unterhalten sich bei einem Glas Absinth über das Sortieren von Strings.[1] Voller Stolz und Freude teilt Armin seine Idee mit Christoph:

> Ein Algorithmus linearer Zeit,
> Strings der Länge n in Gesamtheit.
> In einen Trie sie passen,
> Sortiert sie ausspucken lassen,
> Zum MIT: nun eine echte Möglichkeit!

Doch Christoph scheint nicht überzeugt:

> Theorie deutlich,
> $n \log n$, unüberwindbar,
> Algorithmus fort.

Können Sie aufklären, wer, Armin oder Christoph, recht hat?
Gegeben seien Strings beliebiger Länge (über dem Alphabet $\{0, 1\}$), getrennt durch ein Trennzeichen (z.B. \$), sodass die Gesamtlänge n ist. Kann man diese Strings in Zeit $O(n)$ sortieren, indem man sie in einen anfangs leeren Trie hinzufügt und im Anschluss passend ausgibt?
Wenn ja, begründen Sie die Laufzeit von $O(n)$, geben Sie an, wie aus dem Trie die sortierte Folge in linear Zeit ausgegeben werden kann und argumentieren Sie, wieso diese Laufzeit nicht der unteren Schranke aus der Vorlesung fürs Sortieren widerspricht. Wenn nein, zeigen Sie, wo dieses Verfahren mehr als $O(n)$ Zeit braucht oder fehlerhaft ist.

Lösungsvorschlag
In der Tat hat Armin Recht: Es dauert $O(s)$ Zeit, um einen String der Länge s in einen Trie hinzuzufügen. Um alle Strings hinzuzufügen, dauert es also genau $\sum_s O(s) = O(n)$ Zeit, da n die Summe der Längen aller Strings ist.
Entsprechend ist der resultierende Trie auch $O(n)$ groß und die Preorder-Traversierung kann linear in der Größe des Baumes ausgeführt werden.
Die untere Schranke aus der Vorlesung kommt aus zwei Gründen nicht zu tragen: Erstens ist das Verfahren in dem Sinne nicht vergleichsbasiert. Das sieht man daran, dass die Eingabestrings nicht ihrer Ordnung nach vergleicht werden, sondern dass ihr „Wert“ genutzt, um sie in den Trie einzuordnen. Zweitens ist n in der Schranke die Anzahl der (verschiedenen) Elemente, nicht deren Gesamtlänge. Dieser subtile Unterschied ist wichtig!
Als Beispiel könnte man sich vorstellen, dass die Länge jedes Wortes fest ist, sagen wir ℓ und die Gesamtlänge n ist. Wenn n deutlich größer als ℓ ist (genauer, $\geq 2^\ell$), dann gibt es viele Duplikate, was auch die Anzahl der möglichen Inputs verringert (statt $(n/\ell)!$).
Radixtrees sind natürlich eine offensichtliche Verbesserung (aus praktischer, nicht asymptotischer Sicht). Radix-Sort bietet sich hier auch an (kommt später in der Vorlesung).

[1] Dialog unterstützt durch ChatGPT.

Aufgabe H18 (6 Punkte)
Stellen Sie sich vor, dass ein sortiertes Array der Länge n durch $\log\log n$ Vertauschungen benachbarter Elemente verändert wird.

(a) Wie sortieren Sie das entstandene Array möglichst schnell? Können Sie eine Laufzeit von $\Omega(n \log n)$ vermeiden?

(b) Wie viele Vertauschung verkraftet ihre Methode, bevor sie nicht mehr schneller als der allgemeine Fall von $\Theta(n \log n)$ Zeit ist?

Lösungsvorschlag
Mit Insertionsort kann man in $O(n+i)$ Zeit sortieren, wobei k die Anzahl der Inversionen sind. Durch i Vertauschung benachbarter Elemente verursacht man maximal k Inversionen. Diese Methode ist für $i \in o(n \log n)$ Inversionen schneller als $O(n \log n)$, also z.B. Mergesort, dass immer diese Laufzeit hat.
Man kann die Laufzeit von $\Theta(n + i)$ für Insertionsort ganz einfach zeigen. Für jede Inversion wird genau damit verbundene Stelle im Array angeschaut und ein Tausch gemacht. Sonst finden keine Vergleiche (bis auf $O(n)$ für die äußere Schleife) statt. (Eine Laufzeit von $\Omega(n + i)$ folgt aus der Vorlesung.)
Selection Sort hat immer eine Laufzeit von $\Theta(n^2)$ und profitiert nicht von einer kleinen Anzahl Inversionen.