Datenstrukturen & Algorithmen 4. März, 2015

Handout 2

Sebastian Millius, Sandro Feuz, Daniel Graf

Thema: Divide & Conquer (Mergesort, Binäre Suche), Hashing

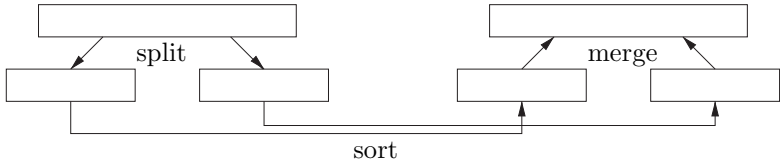
"Divide et impera"
"Veni, vidi, vici"
– Julius Caesar

Divide & Conquer

Divide & Conquer (dt. "Teile und Herrsche") ist ein algorithmisches Design Prinzip / Löseansatz. Das eigentliche Problem wird so lange in kleinere und einfachere Teilprobleme zerlegt (*Divide*), bis man diese lösen (*Conquer*) kann. Anschliessend wird aus diesen Teillösungen eine Lösung für das Gesamtproblem (re)konstruiert. (*Combine*)

1. *Divide*: Teilen des Problems in (unabhängige) Teilprobleme (der gleichen Art)
2. *Conquer*: Löse kleine Teilprobleme (die trivialen) ad hoc (direkt) (Verankerung). Anderfalls löse die Teilprobleme rekursiv.
3. *Combine*: Kombination der Teillösungen zu einer Lösung des Gesamtproblems

Klassische Beispiele für einen Divide & Conquer Ansatz sind **Binary Search** und **Mergesort**: die zu sortierende Liste wird in zwei Teile aufgetrennt, die jede einzeln sortiert wird. Danach werden die beiden sortierten Listen in einer Schleife in linearer Zeit zusammengeführt (*Merge*).



MergeSort($A[1 \dots n]$)
  Merge(MergeSort($A[1, \dots, \lfloor n/2 \rfloor]$), MergeSort($A[\lfloor n/2 \rfloor + 1, \dots, n]$))

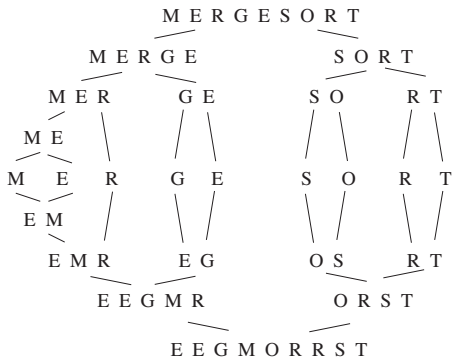


Abb.: Mergesort in Aktion

- *Vergleiche*: $\mathcal{O}(n \log n)$ (Bestcase: $\mathcal{O}(n \log n)$)
- *Verschiebungen*: $\mathcal{O}(n \log n)$ (Bestcase: $\mathcal{O}(n \log n)$)
- *in place?*: Nein (es ist möglich, Mergesort "in place" zu implementieren)
- *stabil?*: Ja

Aufgabe Gegeben ist ein $n \times n$ Grid, n ist eine 2er Potenz, eine Ecke fehlt. Zeige, dass es vollständig bedeckt werden kann mit Teilchen der folgenden Art

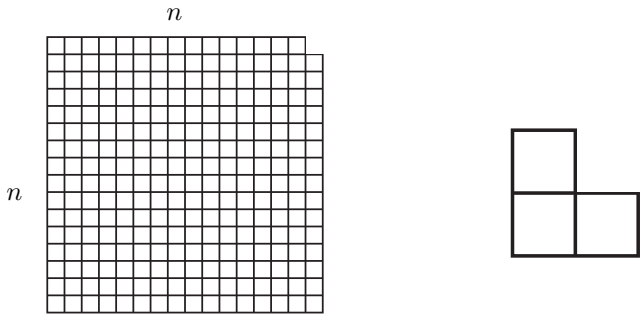


Abb.: *links*: Beispiel Brett, *rechts*: verfügbare Teile

Binäre Suche

Binäre Suche ist ein schneller Algorithmus für die Suche nach Schlüsseln in einem sortierten Array A. Um einen Schlüssel k in A zu suchen, vergleicht man k mit dem Median $A[\frac{n}{2}]$. Ist k vor $A[\frac{n}{2}]$ so muss es in der ersten Hälte liegen, ansonsten in der zweiten Hälfte des Arrays. Durch Wiederholung dieses Prozesses, kann der Schlüssel in $\mathcal{O}(\log n)$ gefunden werden.

BINARY SEARCH(A, k)

```
    // Binäre Suche im sortierten Array A nach dem Schlüssel k
 1  l = 1
 2  r = length[A]

 3  while l ⩽ r
 4        middle = (l + r)/2
 5        if A[middle] < k
 6              l = middle + 1
 7        else if A[middle] > k
 8              r = middle − 1
 9        else return "k gefunden"
10  return "k nicht gefunden"
```

Dies kann als Divide & Conquer Ansatz gesehen werden: erlaube konstanten Aufwand, werfe die Hälfte des Inputs weg und fahre auf dem Rest fort. Wir können die Laufzeit angeben als

$$T(n) \leqslant \begin{cases} T(n/2) + c & \text{falls } n \geqslant 2 \\ c & \text{sonst} \end{cases}$$

Aufgabe: Zeige, dass für obige Rekursionsgleichung gilt: $T(n) = \mathcal{O}(\log n)$.

Stop and Think: Varianten von Binärer Suche: Finde einen möglichst effizienten Algorithmus für folgende Probleme (alle lösbar in sublinearem/logarithmischem Zeitaufwand):

1. Zähle die *Anzahl* gleicher Elemente zu einem Schlüssel k in einem sortierten Array in $\mathcal{O}(\log n)$
2. Gegeben ist ein Array $A[1 \ldots n]$ von folgender Form: $[0, \ldots 0, 1 \ldots 1]$ (eine unbekannte Anzahl Nullen, gefolgt von 1en. Finde Übergangspunkt in einem solchen Array $A[1 \ldots n]$ in $\mathcal{O}(\log n)$
3. *One-Sided Binary Search I*: Gegeben ist ein Array A bestehend aus einem Run von 0en gefolgt von einer unbegrenzten Anzahl von 1en. Bestimme den Übergangspunkt p in $\mathcal{O}(\log p)$.

4. *One-Sided Binary Search II*: Gegeben ist eine unendliche sortierte Sequenz von Zahlen $A = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, ...)$. Entscheide ob ein Schlüssel k in der Sequenz vorkommt.

5. *Peak Finder I: unimodal search* Gegeben ist ein Array A mit n verschiedenen Einträgen. Das Array ist von folgender spezieller Form (*unimodal*): Für einen (unbekannten) Index p zwischen 1 und n gilt:

   · $A[i] < A[i+1]$ für $1 \leqslant i < p$ und
   · $A[i] > A[i+1]$ für $p \leqslant i < n$

   d.h. die Einträge wachsen zunächst an, erreichen einen Peak bei $A[p]$ und nehmen dann wieder ab.

   Finde den Peak in $\mathcal{O}(\log n)$.

6. *Peak Finder II*: Gegeben ist ein unsortiertes Array $A[1 \dots n]$ mit n Zahlen. Ein Peak ist ein "lokales Maximum", d.h. wenn der Einträg grösser gleich als seine Nachbarn ist

$$A[i] \text{ ist Peak wenn } \begin{cases} A[i-1] \leqslant A[i] \geqslant A[i+1] & \text{falls } 1 < i < n \\ A[1] \geqslant A[2] & \text{falls } i = 1 \\ A[n] \geqslant A[n-1] & \text{falls } i = n \end{cases}$$

   Entscheide ob (gibt es immer einen?) und finde einen Peak in A in $\mathcal{O}(\log n)$.

7. *Peak Finder III: Where's the max?* Gegeben ist ein Array A von n sortierten Zahlen, die *circular shifted* um k Positionen nach rechts wurden. Also zum Beispiel $\{35, 42, 5, 15, 27, 29\}$ ist ein sortiertes Array das um $k = 2$ zirkulär verschoben wurde, während $\{27, 29, 35, 42, 5, 15\}$ um $k = 4$ verschoben wurde.

   · Unter der Annahme das k bekannt ist, finde das Maximum (einfach!)
   · Unter der Annahme das k nicht bekannt ist, finde das Maximum in $\mathcal{O}(\log n)$

8. Sei $A[1 \dots n]$ ein Array von n verschiedenen ganzen Zahlen, sortiert, so dass $A[1] < A[2] < \cdots < A[n]$. Jedes $A[i]$ kann daher positiv, negativ oder Null sein. Finde einen effizienten Algorithmus (in $\mathcal{O}(\log n)$) der ein i zurückgibt so dass $A[i] = i$, falls ein solches existiert.

9. Sei nun $A[1 \dots n]$ ein sortiertes Array von n verschiedenen *positiven* ganzen Zahlen; jedes $A[i]$ ist aus $\{1, 2, \dots\}$. Finde einen schnelleren Algorithmus der ein i zurückgibt, so dass $A[i] = i$, falls ein solches existiert. (*Tipp: einfach denken*)

10. Gegeben ist ein sortiertes Array mit n verschiedenen Zahlen aus $\{1, \dots, m\}$, wobei $n < m$. Gebe einen $\mathcal{O}(\log n)$ Algorithmus an, der ein Element aus $\{1, \dots, m\}$ findet, welches nicht im Array vorkommt.

Hashing

Links

· Hashing: MIT Open Courseware, Lectures 8-10 (https://www.youtube.com/watch?v=0M_kIqhwbFo)

Hash Tabellen sind eine Datenstruktur um ein Wörterbuch zu realisieren. Beim Hash-Verfahren wird versucht, durch Berechnung festzustellen, wo der Datensatz mit Schlüssel k gespeichert ist. Die Idee ist, dass das Nachschauen eines Elementes in einem Array $\Theta(1)$ ist, wenn man seinen Index weiss. Bei einer Suchanfrage wird der Schlüssel mitgegeben und falls das Schlüssel/Wert Paar existiert, wird der Wert zurückgegeben. Man kann damit aber beispielsweise auch einfache Mengen modellieren. Hier kann man Elemente einfügen und testen ob ein bestimmtes Element vorhanden ist.

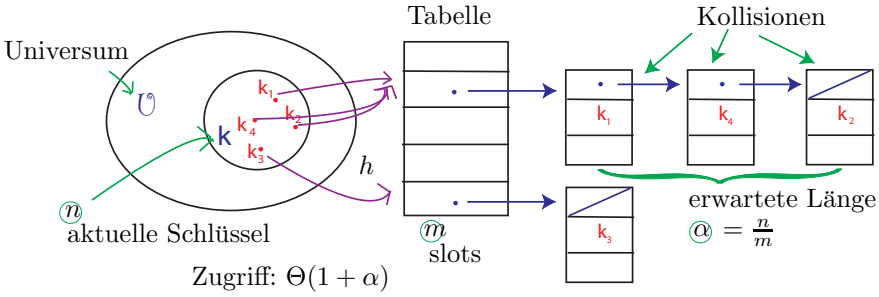
Dabei wird versucht Schlüssel aus einem grossen Schlüsselbereich/Universum $\mathcal{U}$ in einer Hashtabelle der Grösse m abzuspeichern. Eine Hash-Table ist grundsätzlich mal ein Array. Das Array sollte genügend gross sein, so dass nur bis zu ca. 75% der Plätze belegt sind. Je nach Implementation (z.B. in Java) wird die Arraygrösse auch dynamisch angepasst, was aber ziemlich aufwändig ist, da normalerweise alle Elemente neu eingefügt werden müssen.

Eine *Hash Funktion* ist eine mathematische Funktion die Schlüssel auf einen Index (*Hashadresse*) abbilden. Die Hashfunktion sollte schnell (sicher $\mathcal{O}(1)$) zu berechnen sein, und gleichzeitig die Schlüsselstruktur aufbrechen.

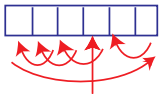
Im optimalen Fall kann somit nach einer Auswertung dieser Funktion sofort ein Element gespeichert oder darauf zugegriffen werden. Das Problem ist aber, dass die Hashfunktion mehrere Schlüssel auf die gleiche Stelle im Array abbilden kann, da es üblicherweise viel mehr mögliche Schlüssel als Plätze im Array hat. ($\rightsquigarrow$ Adresskollision, Synonyme)

Falls also beim Einfügen der Platz schon belegt ist oder sich beim Suchen ein anderer Schlüssel an diesem Ort befindet, muss ein Verfahren existieren das besagt, wo nun weiterzusuchen ist. Dabei gibt es verschiedene Möglichkeiten welche natürlich sehr unterschiedliche Eigenschaften haben:

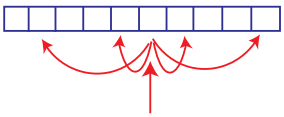
- *Verkettung*: In jedem Feld des Arrays befindet sich wiederum eine Datenstruktur die mehrere Elemente aufnehmen kann (z.B. Liste, Baum, ...)



- *Mehrere Hashfunktionen*: Bei einer Kollision wird einfach die nächste Hashfunktion probiert.
- *Offene Hashverfahren.* Anders als beim Verketten werden die Überläufer direkt in der Tabelle abgelegt. Eine Sondierungsreihenfolge gibt dabei an, wo Synonyme gespeichert werden.
  - *Lineares Sondieren*: Bei Kollisionen einfach die nächsten Felder probieren (absteigend oder aufsteigend).

    Ein Problem bei diesem Verfahren kann die *primäre Häufung* sein: Teile von bereits besetzten Bereichen werden immer grösser, so dass immer länger sondiert werden muss.
  - *Quadratisches Sondieren*: Anstatt die Felder linear aufzufüllen / zu durchsuchen wird mit quadratisch steigenden Abständen gesucht.

  - *Doppeltes Hashing*: Eine zweite Hashfunktion $h_0(x)$ wird benutzt um den Abstand beim Sondieren zu bestimmen. Die zusammengesetzte Funktion lautet dann: $h_i(x) = (h(x) - h_0(x) \cdot i) \mod m$

Es gibt auch noch ein paar weitere Methoden. Die wichtigsten sind aber oben aufgeführt.

Bemerkung: Ob jeweils (zuerst) nach unten oder nach oben sondiert wird ist Geschmackssache. Wichtig ist einfach, dass es immer konsequent gleich gemacht wird. Dies gilt insbesondere auch für die Prüfung (hier ev. noch hinschreiben in welche Richtung man sondiert..)

Aufgabe

Führen Sie die unten gegebene Sequenz von Einfüge-Operationen auf einer anfangs leeren Hashtabelle der Grösse 11 mit Hashfunktion $h(k) = k \mod 11$ aus. Verwenden Sie als Kollisions-Auflösungs-Strategie einmal lineares Sondieren, einmal quadratisches Sondieren und einmal Double Hashing mit $h_0(k) = 1 + (k \mod 9)$ aus. Zählen Sie die Anzahl betrachteter Einträge der Operationsfolge. Welche Variante ist für diese Sequenz die beste? Die Sequenz, die eingefügt werden soll, ist: $20, 34, 26, 33, 11, 16, 47, 4, 14, 17$.