Script **generated by TTT**

Title: Seidl: GAD (24.05.2016)

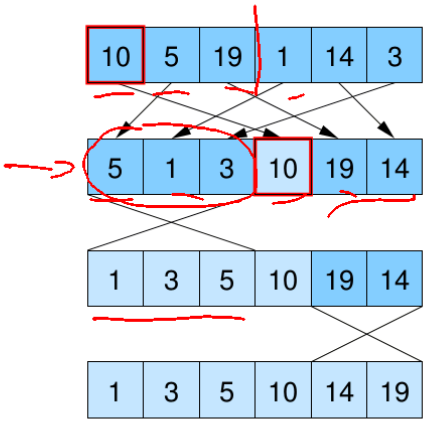
Date: Tue May 24 14:22:14 CEST 2016

Duration: 87:55 min

Pages: 70

QuickSort

Idee:
Aufspaltung in zwei Teilmengen, aber nicht in der Mitte der Sequenz wie bei MergeSort, sondern getrennt durch ein Pivotelement



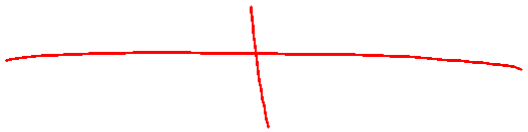
Implementierung: effizient und in-place

```
void quickSort(Element[ ] a, int ℓ, int r) {
    // a[ℓ ... r]: zu sortierendes Feld
    if (ℓ < r) {
        p = a[r];   // Pivot
        int i = ℓ − 1;   int j = r;
        do {   // spalte Elemente in a[ℓ, ..., r − 1] nach Pivot p
            do { i++ } while (a[i] < p);
            do { j−− } while (j ≥ ℓ ∧ a[j] > p);
            if (i < j) swap(a, i, j);
        } while (i < j);
        swap (a, i, r);   // Pivot an richtige Stelle
        quickSort(a, ℓ, i − 1);
        quickSort(a, i + 1, r);
    }
}
```


QuickSort: Laufzeit

Problem:

- im Gegensatz zu MergeSort kann die Aufteilung in Teilprobleme unbalanciert sein (also nur im Optimalfall eine Halbierung)
- im worst case quadratische Laufzeit
  (z.B. wenn Pivotelement immer kleinstes oder größtes aller Elemente ist)



Implementierung: effizient und in-place

```
void quickSort(Element[ ] a, int ℓ, int r) {
    // a[ℓ ... r]: zu sortierendes Feld
    if (ℓ < r) {
        p = a[r];   // Pivot
        int i = ℓ − 1;   int j = r;
        do {   // spalte Elemente in a[ℓ, ..., r − 1] nach Pivot p
            do { i++ } while (a[i] < p);
            do { j−− } while (j ≥ ℓ ∧ a[j] > p);
            if (i < j) swap(a, i, j);
        } while (i < j);
        swap (a, i, r);   // Pivot an richtige Stelle
        quickSort(a, ℓ, i − 1);
        quickSort(a, i + 1, r);
    }
}
```


QuickSort: Laufzeit

Problem:

- im Gegensatz zu MergeSort kann die Aufteilung in Teilprobleme unbalanciert sein (also nur im Optimalfall eine Halbierung)
- im worst case quadratische Laufzeit
  (z.B. wenn Pivotelement immer kleinstes oder größtes aller Elemente ist)

Lösungen:

- wähle zufälliges Pivotelement:
  Laufzeit $O(n \log n)$ mit hoher Wahrscheinlichkeit
- berechne Median (mittleres Element):
  mit Selektionsalgorithmus, später in der Vorlesung

Sortieren QuickSort

Implementierung: effizient und in-place

void quickSort(Element[ ] a, int ℓ, int r) {
    // a[ℓ ... r]: zu sortierendes Feld
    if (ℓ < r) {
        p = a[r];   // Pivot
        int i = ℓ − 1;   int j = r;
        do {   // spalte Elemente in a[ℓ, ..., r − 1] nach Pivot p
            do { i++ } while (a[i] < p);
            do { j−− } while (j ≥ ℓ ∧ a[j] > p);
            if (i < j) swap(a, i, j);
        } while (i < j);
        swap (a, i, r);   // Pivot an richtige Stelle
        quickSort(a, ℓ, i − 1);
        quickSort(a, i + 1, r);
    }
}

H. Seidl (TUM) GAD SS'16 219

Sortieren QuickSort

QuickSort: Laufzeit

Problem:

- im Gegensatz zu MergeSort kann die Aufteilung in Teilprobleme unbalanciert sein (also nur im Optimalfall eine Halbierung)
- im worst case quadratische Laufzeit
  (z.B. wenn Pivotelement immer kleinstes oder größtes aller Elemente ist)

Lösungen:

- wähle zufälliges Pivotelement:
  Laufzeit $O(n \log n)$ mit hoher Wahrscheinlichkeit
- berechne Median (mittleres Element):
  mit Selektionsalgorithmus, später in der Vorlesung

H. Seidl (TUM) GAD SS'16 220

QuickSort

Laufzeit bei zufälligem Pivot-Element

- Zähle Anzahl Vergleiche (Rest macht nur konstanten Faktor aus)
- $\bar{C}(n)$: erwartete Anzahl Vergleiche bei n Elementen

Satz

Die erwartete Anzahl von Vergleichen für QuickSort mit zufällig ausgewähltem Pivotelement ist

$$\bar{C}(n) \leq 2n \ln n \leq 1.39 n \log_2 n$$

QuickSort

Beweis.

- Betrachte sortierte Sequenz $\langle e'_1, \ldots, e'_n \rangle$
- nur Vergleiche mit Pivotelement
- Pivotelement ist nicht in den rekursiven Aufrufen enthalten

$\Rightarrow$ e'_i und e'_j werden höchstens einmal verglichen und zwar dann, wenn e'_i oder e'_j Pivotelement ist

QuickSort

Beweis.

- Zufallsvariable $X_{ij} \in \{0, 1\}$
- $X_{ij} = 1 \quad \Leftrightarrow \quad e'_i$ und e'_j werden verglichen

$$\begin{aligned}
\bar{C}(n) &= \mathbb{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathbb{E}\left[X_{ij}\right] \\
&= \sum_{i<j} 0 \cdot \Pr\left[X_{ij} = 0\right] + 1 \cdot \Pr\left[X_{ij} = 1\right] \\
&= \sum_{i<j} \Pr\left[X_{ij} = 1\right]
\end{aligned}$$

QuickSort

Lemma

$$\Pr[X_{ij} = 1] = 2/(j - i + 1)$$

Beweis.

- Sei $M = \{e'_i, \ldots, e'_j\}$
- Irgendwann wird ein Element aus M als Pivot ausgewählt.
- Bis dahin bleibt M immer zusammen.
- e'_i und e'_j werden genau dann *direkt* verglichen, wenn eines der beiden als Pivot ausgewählt wird
- Wahrscheinlichkeit:

$$\Pr[e'_i \text{ oder } e'_j \text{ aus } M \text{ ausgewählt}] = \frac{2}{|M|} = \frac{2}{j - i + 1}$$

□

QuickSort

Beweis.

$$\begin{aligned}
\bar{C} &= \sum_{i<j} \Pr\left[X_{ij}=1\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1} \frac{2}{k} = 2\sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1} \frac{1}{k} \\
&\le 2\sum_{i=1}^{n-1}\sum_{k=2}^{n} \frac{1}{k} = 2(n-1)\sum_{k=2}^{n} \frac{1}{k} = 2(n-1)(H_n-1) \\
&\le 2(n-1)(1+\ln n-1) \le 2n\ln n = 2n\ln(2)\log_2(n)
\end{aligned}$$

$\ln n = \log_2 n \cdot \ln 2$

□

Sortieren QuickSort

QuickSort

Beweis.

$$\begin{aligned}
\bar{C} &= \sum_{i<j} \Pr\left[X_{ij}=1\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1} \frac{2}{k} = 2\sum_{i=1}^{n-1}\sum_{k=2}^{n-i+1} \frac{1}{k} \\
&\le 2\sum_{i=1}^{n-1}\sum_{k=2}^{n} \frac{1}{k} = 2(n-1)\sum_{k=2}^{n} \frac{1}{k} = 2(n-1)(H_n-1) \\
&\le 2(n-1)(1+\ln n-1) \le 2n\ln n = 2n\ln(2)\log_2(n)
\end{aligned}$$

□

H. Seidl (TUM) GAD SS'16 225

QuickSort

Verbesserte Version ohne Check für Array-Grenzen

```
void qSort(Element[ ] a, int ℓ, int r) {
    while (r − ℓ ≥ n0) {
        j = pickPivotPos(a, ℓ, r);
        swap(a, ℓ, j);      p = a[ℓ];
        int i = ℓ;      int j = r;
        repeat {
            while (a[i] < p)    i++;
            while (a[j] > p)    j−−;
            if (i ≤ j) { swap(a, i, j);   i++;   j−−; }
        } until (i > j);
        if (i ≤ (ℓ + r)/2) { qSort(a, ℓ, j);   ℓ = i; }
        else               { qSort(a, i, r);   r = j; }
    }
    insertionSort(a, ℓ, r);
}
```


Rang-Selektion

- Bestimmung des kleinsten und größten Elements ist mit einem einzigen Scan über das Array in Linearzeit möglich
- Aber wie ist das beim k-kleinsten Element, z.B. beim $\lfloor n/2 \rfloor$-kleinsten Element (Median)?

Problem:

Finde k-kleinstes Element in einer Menge von n Elementen

Rang-Selektion

- Bestimmung des kleinsten und größten Elements ist mit einem einzigen Scan über das Array in Linearzeit möglich
- Aber wie ist das beim k-kleinsten Element, z.B. beim $\lfloor n/2 \rfloor$-kleinsten Element (Median)?

Problem:

Finde k-kleinstes Element in einer Menge von n Elementen

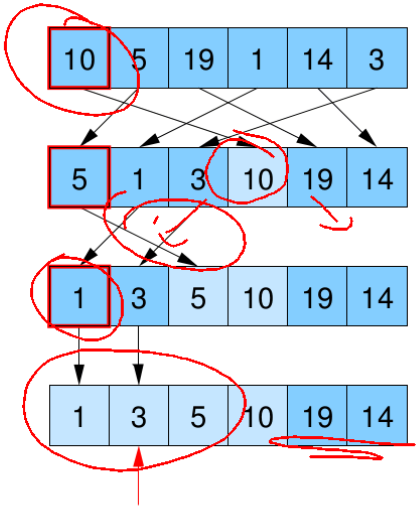
Naive Lösung: Sortieren und k-tes Element ausgeben

$\Rightarrow$ Zeit $O(n \log n)$

Geht das auch schneller?

QuickSelect

Ansatz: ähnlich zu QuickSort, aber nur eine Seite betrachten



QuickSelect

Methode analog zu QuickSort

```
Element quickSelect(Element[] a, int l, int r, int k) {
  // a[l...r]: Restfeld,   k: Rang des gesuchten Elements
  if (r == l) return a[l];
  int z = zufällige Position in {l,...,r}; swap(a,z,r);
  Element p = a[r];  int i = l - 1;  int j = r;
  do {  // spalte Elemente in a[l,...,r-1] nach Pivot p
    do i++ while (a[i] < p);
    do j-- while (a[j] > p && j ≠ l);
    if (i < j) swap(a,i,j);
  } while (i < j);
  swap (a,i,r);  // Pivot an richtige Stelle
  if (k < i) return quickSelect(a, l, i-1, k);
  if (k > i) return quickSelect(a, i+1, r, k);
  else return a[k];  // k == i
}
```

QuickSelect

Alternative Methode

```
Element select(Element[] s, int k) {
  assert(|s| ≥ k);
  Wähle p ∈ s zufällig (gleichverteilt);

  Element[] a := {e ∈ s : e < p};
  if (|a| ≥ k)
    return select(a,k);

  Element[] b := {e ∈ s : e = p};
  if (|a| + |b| ≥ k)
    return p;

  Element[] c := {e ∈ s : e > p};
  return select(c,k - |a| - |b|);
}
```

QuickSelect

Alternative Methode

```
Element select(Element[ ] s, int k) {
  assert(|s| ≥ k);
  Wähle p ∈ s zufällig (gleichverteilt);

  Element[ ] a := {e ∈ s : e < p};
  if (|a| ≥ k)
    return select(a,k);

  Element[ ] b := {e ∈ s : e = p};
  if (|a| + |b| ≥ k)
    return p;

  Element[ ] c := {e ∈ s : e > p};
  return select(c,k − |a| − |b|);
}
```


Sortieren Selektieren

QuickSelect

Alternative Methode

```
Element select(Element[ ] s, int k) {
  assert(|s| ≥ k);
  Wähle p ∈ s zufällig (gleichverteilt);

  Element[ ] a := {e ∈ s : e < p};
  if (|a| ≥ k)
    return select(a,k);

  Element[ ] b := {e ∈ s : e = p};
  if (|a| + |b| ≥ k)
    return p;

  Element[ ] c := {e ∈ s : e > p};
  return select(c,k − |a| − |b|);
}
```


H. Seidl (TUM) GAD SS'16 230

Sortieren Selektieren

QuickSelect

Alternative Methode

Beispiel

s	k	a	b	c
⟨3,1,4,1,5,9,2,6,5,3,5,8,9⟩	7	⟨1,1⟩	⟨2⟩	⟨3,4,5,9,6,5,3,5,8,9⟩
⟨3,4,5,9,6,5,3,5,8,9⟩	4	⟨3,4,5,5,3,5⟩	⟨6⟩	⟨9,8,9⟩
⟨3,4,5,5,3,5⟩	4	⟨3,4,3⟩	⟨5,5,5⟩	⟨⟩

In der sortierten Sequenz würde also an 7. Stelle das Element 5 stehen.

Hier wurde das mittlere Element als Pivot verwendet.

Sortieren Selektieren

QuickSelect

teilt das Feld jeweils in 3 Teile:

- a Elemente kleiner als das Pivot
- b Elemente gleich dem Pivot
- c Elemente größer als das Pivot

$T(n)$: erwartete Laufzeit bei n Elementen

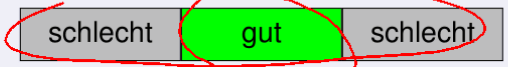
Satz

Die erwartete Laufzeit von QuickSelect ist linear: $T(n) \in O(n)$.

QuickSelect

Beweis.

- Pivot ist gut, wenn weder a noch c länger als 2/3 der aktuellen Feldgröße sind:



$\Rightarrow$ Pivot ist gut, falls es im mittleren Drittel liegt

$$p = \Pr[\text{Pivot ist gut}] = 1/3$$

QuickSelect

Beweis.

- Pivot ist gut, wenn weder a noch c länger als 2/3 der aktuellen Feldgröße sind:

$\Rightarrow$ Pivot ist gut, falls es im mittleren Drittel liegt

$$p = \Pr[\text{Pivot ist gut}] = 1/3$$

Erwartete Zeit bei n Elementen

- linearer Aufwand außerhalb der rekursiven Aufrufe: cn
- Pivot gut (Wsk. 1/3): Restaufwand $\leq T(2n/3)$
- Pivot schlecht (Wsk. 2/3): Restaufwand $\leq T(n-1) < T(n)$

QuickSelect

Beweis.

$$\begin{aligned}
T(n) &\leq cn + p \cdot T(n \cdot 2/3) + (1-p) \cdot T(n) \\
p \cdot T(n) &\leq cn + p \cdot T(n \cdot 2/3) \\
T(n) &\leq cn/p + T(n \cdot 2/3) \\
&\leq cn/p + c \cdot (n \cdot 2/3)/p + T(n \cdot (2/3)^2) \\
&\ldots \text{ wiederholtes Einsetzen} \\
&\leq (cn/p)(1 + 2/3 + 4/9 + 8/27 + \ldots) \\
&\leq \frac{cn}{p} \cdot \sum_{i \geq 0} (2/3)^i \qquad \sum_{i=0}^{\infty} q^i = \frac{1}{1-q} \text{ für } q < 1 \\
&\leq \frac{cn}{1/3} \cdot \frac{1}{1-2/3} = 9cn \in O(n)
\end{aligned}$$

□

QuickSelect

Beweis.

$$\begin{aligned}
T(n) &\leq cn + p \cdot T(n \cdot 2/3) + (1-p) \cdot T(n) \\
p \cdot T(n) &\leq cn + p \cdot T(n \cdot 2/3) \\
T(n) &\leq cn/p + T(n \cdot 2/3) \\
&\leq cn/p + c \cdot (n \cdot 2/3)/p + T(n \cdot (2/3)^2) \\
&\ldots \text{ wiederholtes Einsetzen} \\
&\leq (cn/p)(1 + 2/3 + 4/9 + 8/27 + \ldots) \\
&\leq \frac{cn}{p} \cdot \sum_{i \geq 0} (2/3)^i \\
&\leq \frac{cn}{1/3} \cdot \frac{1}{1-2/3} = 9cn \in O(n)
\end{aligned}$$

□

Sortieren schneller als $O(n \log n)$

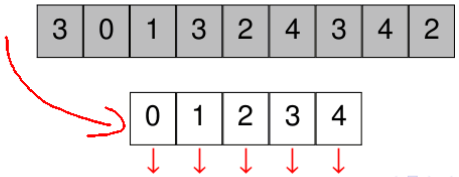
Buckets

- mit paarweisen Schlüsselvergleichen: nie besser als $O(n \log n)$
- Was aber, wenn die Schlüsselmenge mehr Struktur hat?
  z.B. Zahlen / Strings bestehend aus mehreren Ziffern / Zeichen
- Um zwei Zahlen / Strings zu vergleichen reicht oft schon die erste Ziffer / das erste Zeichen.
  Nur bei gleichem Anfang kommt es auf mehr Ziffern / Zeichen an.

Sortieren schneller als $O(n \log n)$

Buckets

- mit paarweisen Schlüsselvergleichen: nie besser als $O(n \log n)$
- Was aber, wenn die Schlüsselmenge mehr Struktur hat?
  z.B. Zahlen / Strings bestehend aus mehreren Ziffern / Zeichen
- Um zwei Zahlen / Strings zu vergleichen reicht oft schon die erste Ziffer / das erste Zeichen.
  Nur bei gleichem Anfang kommt es auf mehr Ziffern / Zeichen an.
- Annahme: Elemente sind Zahlen im Bereich $\{0, \ldots, K-1\}$
- Strategie: verwende Feld von K Buckets (z.B. Listen)



Sortieren schneller als $O(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[ ] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```


Sortieren schneller als $O(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[ ] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n + K)$ Problem: nur gut für $K \in o(n \log n)$
Speicher: $\Theta(n + K)$

Sortieren schneller als $O(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[ ] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n+K)$ Problem: nur gut für $K \in o(n \log n)$
Speicher: $\Theta(n+K)$

- wichtig: kSort ist stabil, d.h. Elemente mit dem gleichen Schlüssel behalten ihre relative Reihenfolge

$\Rightarrow$ Elemente müssen im jeweiligen Bucket *hinten* angehängt werden

RadixSort

- verwende K-adische Darstellung der Schlüssel
- Annahme:
  Schlüssel sind Zahlen aus $\{0,\ldots,K^d-1\}$ repräsentiert durch d Stellen von Ziffern aus $\{0,\ldots,K-1\}$
- sortiere zunächst entsprechend der niedrigstwertigen Ziffer mit kSort und dann nacheinander für immer höherwertigere Stellen
- behalte Ordnung der Teillisten bei

RadixSort

```
radixSort(Sequence<Elem> s) {
    for (int i = 0; i < d; i++)
        kSort(s,i);      // sortiere gemäß key_i(x)
                         // mit key_i(x) = (key(x)/K^i) mod K
}
```

Verfahren funktioniert, weil kSort stabil ist:

Elemente mit gleicher i-ter Ziffer bleiben sortiert bezüglich der Ziffern $i-1\ldots0$ während der Sortierung nach Ziffer i

Laufzeit: $O(d(n+K))$ für n Schlüssel aus $\{0,\ldots,K^d-1\}$

Sortieren schneller als $O(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[ ] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n+K)$ Problem: nur gut für $K \in o(n \log n)$
Speicher: $\Theta(n+K)$

- wichtig: kSort ist stabil, d.h. Elemente mit dem gleichen Schlüssel behalten ihre relative Reihenfolge

$\Rightarrow$ Elemente müssen im jeweiligen Bucket *hinten* angehängt werden

RadixSort

```
radixSort(Sequence<Elem> s) {
    for (int i = 0; i < d; i++)
        kSort(s,i);      // sortiere gemäß key_i(x)
                         // mit key_i(x) = (key(x)/K^i) mod K
}
```


RadixSort

```
radixSort(Sequence<Elem> s) {
    for (int i = 0; i < d; i++)
        kSort(s,i);      // sortiere gemäß key_i(x)
                         // mit key_i(x) = (key(x)/K^i) mod K
}
```

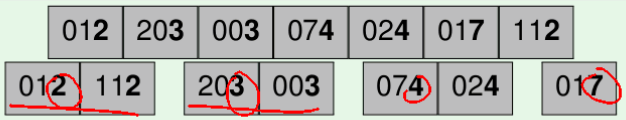
Verfahren funktioniert, weil kSort stabil ist:

Elemente mit gleicher i-ter Ziffer bleiben sortiert bezüglich der Ziffern $i-1 \ldots 0$ während der Sortierung nach Ziffer i

Laufzeit: $O(d(n+K))$ für n Schlüssel aus $\{0,\ldots,K^d-1\}$

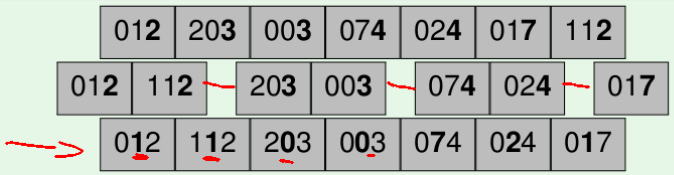
RadixSort

Beispiel



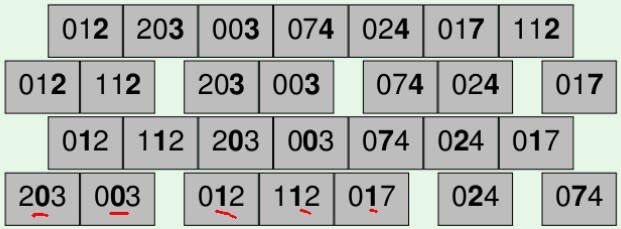
RadixSort

Beispiel



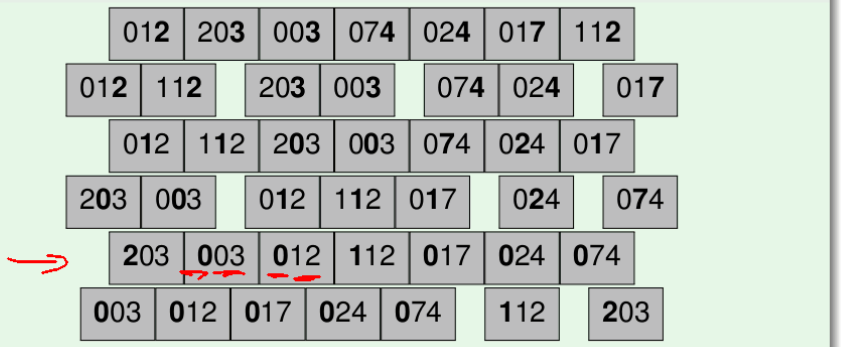
RadixSort

Beispiel



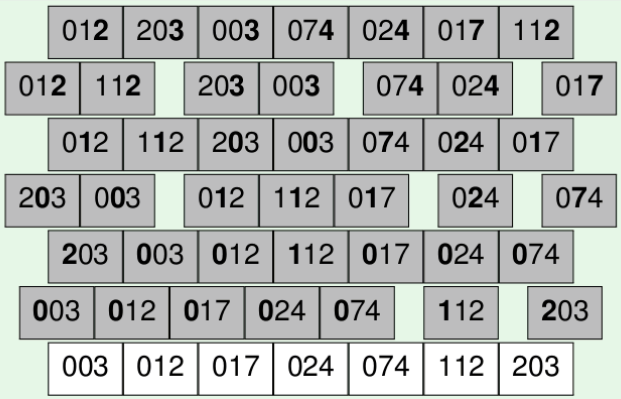
RadixSort

Beispiel



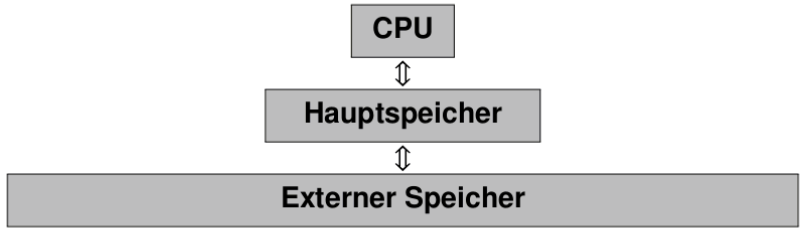
RadixSort

Beispiel



Externes Sortieren

Heutige Computer:



- Hauptspeicher hat Größe M
- Transfer zwischen Hauptspeicher und externem Speicher mit Blockgröße B

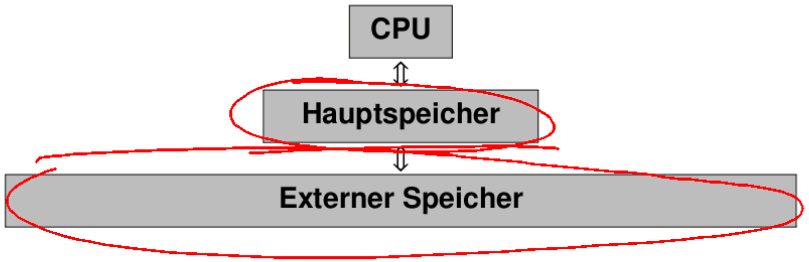
RadixSort

Beispiel

012	203	003	074	024	017	112
012	112	203	003	074	024	017
012	112	203	003	074	024	017
203	003	012	112	017	024	074
203	003	012	112	017	024	074
003	012	017	024	074	112	203
003	012	017	024	074	112	203

Externes Sortieren

Heutige Computer:



- Hauptspeicher hat Größe M
- Transfer zwischen Hauptspeicher und externem Speicher mit Blockgröße B

Externes Sortieren

Problem:

Minimiere Anzahl Blocktransfers zwischen internem und externem Speicher

Anmerkung:

Gleiches Problem trifft auch auf anderen Stufen der Hierarchie zu (Cache)

Externes Sortieren

Problem:

Minimiere Anzahl Blocktransfers zwischen internem und externem Speicher

Anmerkung:

Gleiches Problem trifft auch auf anderen Stufen der Hierarchie zu (Cache)

Lösung: Verwende MergeSort

Vorteil:

MergeSort verwendet oft konsekutive Elemente (Scanning)
(geht auf Festplatte schneller als Random Access-Zugriffe)

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar ($B \mid M$)
  (sonst z.B. Auffüllen mit maximalem Schlüssel)

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar ($B \mid M$)
  (sonst z.B. Auffüllen mit maximalem Schlüssel)

Run Formation Phase:

- Lade wiederholt Teilfeld der Größe M in den Speicher,
- sortiere es mit einem in-place-Sortierverfahren,
- schreibe sortiertes Teilfeld (Run) wieder zurück auf die Festplatte

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar ($B \mid M$)
  (sonst z.B. Auffüllen mit maximalem Schlüssel)

Run Formation Phase:

- Lade wiederholt Teilfeld der Größe M in den Speicher,
- sortiere es mit einem in-place-Sortierverfahren,
- schreibe sortiertes Teilfeld (Run) wieder zurück auf die Festplatte

$\Rightarrow$ benötigt n/B Blocklese- und n/B Blockschreiboperationen
Laufzeit: $2n/B$ Transfers

- ergibt sortierte Bereiche (Runs) der Größe M

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar ($B \mid M$)
  (sonst z.B. Auffüllen mit maximalem Schlüssel)

Run Formation Phase:

- Lade wiederholt Teilfeld der Größe M in den Speicher,
- sortiere es mit einem in-place-Sortierverfahren,
- schreibe sortiertes Teilfeld (Run) wieder zurück auf die Festplatte

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar ($B \mid M$)
  (sonst z.B. Auffüllen mit maximalem Schlüssel)

Run Formation Phase:

- Lade wiederholt Teilfeld der Größe M in den Speicher,
- sortiere es mit einem in-place-Sortierverfahren,
- schreibe sortiertes Teilfeld (Run) wieder zurück auf die Festplatte

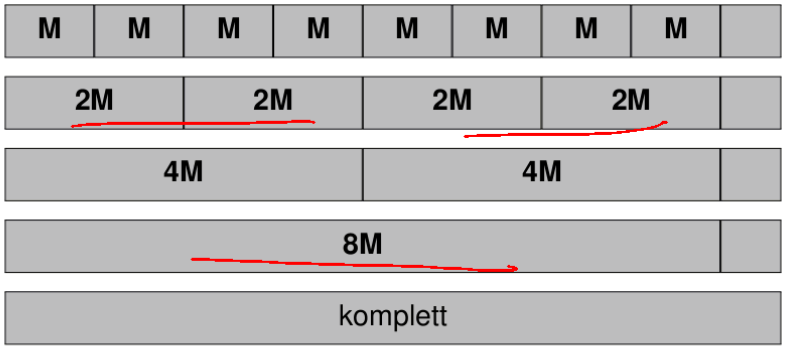
$\Rightarrow$ benötigt n/B Blocklese- und n/B Blockschreiboperationen
Laufzeit: $2n/B$ Transfers

- ergibt sortierte Bereiche (Runs) der Größe M

Externes Sortieren

Merge Phasen

- Merge von jeweils 2 Teilfolgen in $\lceil \log_2(n/M) \rceil$ Phasen
- dabei jeweils Verdopplung der Größe der sortierten Teile



Merge von zwei Runs

- von jedem der beiden Runs und von der Ausgabesequenz bleibt ein Block im Hauptspeicher (3 Puffer: 2× Eingabe, 1× Ausgabe)
- Anfang: beide Eingabepuffer mit B Elementen (1 Block) laden, Ausgabepuffer leer
- Dann: jeweils führende Elemente der beiden Eingabepuffer vergleichen und das kleinere in den Ausgabepuffer schreiben
- Wenn Eingabepuffer leer $\Rightarrow$ neuen Block laden
- Wenn Ausgabepuffer voll $\Rightarrow$ Block auf Festplatte schreiben und Ausgabepuffer leeren

Merge von zwei Runs

- von jedem der beiden Runs und von der Ausgabesequenz bleibt ein Block im Hauptspeicher (3 Puffer: 2× Eingabe, 1× Ausgabe)
- Anfang: beide Eingabepuffer mit B Elementen (1 Block) laden, Ausgabepuffer leer
- Dann: jeweils führende Elemente der beiden Eingabepuffer vergleichen und das kleinere in den Ausgabepuffer schreiben
- Wenn Eingabepuffer leer $\Rightarrow$ neuen Block laden
- Wenn Ausgabepuffer voll $\Rightarrow$ Block auf Festplatte schreiben und Ausgabepuffer leeren

Merge von zwei Runs

- von jedem der beiden Runs und von der Ausgabesequenz bleibt ein Block im Hauptspeicher (3 Puffer: 2× Eingabe, 1× Ausgabe)
- Anfang: beide Eingabepuffer mit B Elementen (1 Block) laden, Ausgabepuffer leer
- Dann: jeweils führende Elemente der beiden Eingabepuffer vergleichen und das kleinere in den Ausgabepuffer schreiben
- Wenn Eingabepuffer leer $\Rightarrow$ neuen Block laden
- Wenn Ausgabepuffer voll $\Rightarrow$ Block auf Festplatte schreiben und Ausgabepuffer leeren

- In jeder Merge-Phase wird das ganze Feld einmal gelesen und geschrieben

$\Rightarrow (2n/B)(1 + \lceil \log_2(n/M) \rceil)$ Block-Transfers

Multiway-MergeSort

- Verfahren funktioniert, wenn 3 Blöcke in den Speicher passen
- Wenn mehr Blöcke in den Speicher passen, kann man gleich $k \geq 2$ Runs mergen.
- Benutze Prioritätswarteschlange (Priority Queue) zur Minimumermittlung, wobei die Operationen $O(\log k)$ Zeit kosten
- $(k+1)$ Blocks und die PQ müssen in den Speicher passen

$\Rightarrow (k+1)B + O(k) \leq M$, also $k \in O(M/B)$

- Anzahl Merge-Phasen reduziert auf $\lceil \log_k(n/M) \rceil$

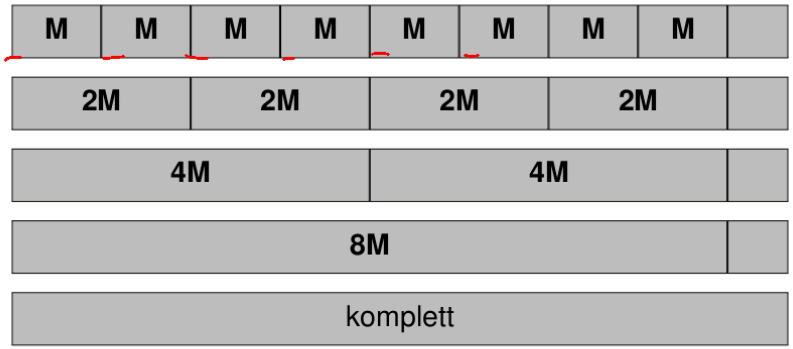
$\Rightarrow (2n/B)\left(1 + \left\lceil \log_{M/B}(n/M) \right\rceil\right)$ Block-Transfers

- In der Praxis: Anzahl Merge-Phasen gering
- Wenn $n \leq M^2/B$: nur eine einzige Merge-Phase (erst M/B Runs der Größe M, dann einmal Merge)

Externes Sortieren

Merge Phasen

- Merge von jeweils 2 Teilfolgen in $\lceil \log_2(n/M) \rceil$ Phasen
- dabei jeweils Verdopplung der Größe der sortierten Teile



Multiway-MergeSort

- Verfahren funktioniert, wenn 3 Blöcke in den Speicher passen
- Wenn mehr Blöcke in den Speicher passen, kann man gleich $k \geq 2$ Runs mergen.
- Benutze Prioritätswarteschlange (Priority Queue) zur Minimumermittlung, wobei die Operationen $O(\log k)$ Zeit kosten
- $(k+1)$ Blocks und die PQ müssen in den Speicher passen

$\Rightarrow (k+1)B + O(k) \leq M$, also $k \in O(M/B)$

- Anzahl Merge-Phasen reduziert auf $\lceil \log_k(n/M) \rceil$

$\Rightarrow (2n/B)\left(1 + \left\lceil \log_{M/B}(n/M) \right\rceil\right)$ Block-Transfers

- In der Praxis: Anzahl Merge-Phasen gering
- Wenn $n \leq M^2/B$: nur eine einzige Merge-Phase (erst M/B Runs der Größe M, dann einmal Merge)

Multiway-MergeSort

- Verfahren funktioniert, wenn 3 Blöcke in den Speicher passen
- Wenn mehr Blöcke in den Speicher passen, kann man gleich $k \geq 2$ Runs mergen.
- Benutze Prioritätswarteschlange (Priority Queue) zur Minimumermittlung, wobei die Operationen $O(\log k)$ Zeit kosten
- $(k+1)$ Blocks und die PQ müssen in den Speicher passen

$\Rightarrow$ $(k+1)B + O(k) \leq M$, also $k \in O(M/B)$

- Anzahl Merge-Phasen reduziert auf $\lceil \log_k(n/M) \rceil$

$\Rightarrow$ $(2n/B)\left(1 + \left\lceil \log_{M/B}(n/M) \right\rceil\right)$ Block-Transfers

- In der Praxis: Anzahl Merge-Phasen gering
- Wenn $n \leq M^2/B$: nur eine einzige Merge-Phase (erst M/B Runs der Größe M, dann einmal Merge)

Prioritätswarteschlangen

M: Menge von Elementen

prio(e): Priorität von Element e

Operationen:

- M.build($\{e_1, \ldots, e_n\}$): $M = \{e_1, \ldots, e_n\}$
- M.insert(Element e): $M = M \cup e$
- Element M.min(): gib ein e mit minimaler Priorität prio(e) zurück
- Element M.deleteMin():
  entferne Element e mit minimalem Wert prio(e)
  und gib es zurück

Adressierbare Prioritätswarteschlangen

Zusätzliche Operationen für adressierbare Priority Queues:

- Handle insert(Element e): wie zuvor, gibt aber ein Handle (Referenz / Zeiger) auf das eingefügte Element zurück
- remove(Handle h): lösche Element spezifiziert durch Handle h
- decreaseKey(Handle h, int k):
  reduziere Schlüssel / Priorität des Elements auf Wert k
  (je nach Implementation evt. auch um Differenz k)
- M.merge(Q): $M = M \cup Q$; $Q = \emptyset$;

Prioritätswarteschlangen mit Listen

Priority Queue mittels unsortierter Liste:

- build($\{e_1, \ldots, e_n\}$): Zeit $O(n)$
- insert(Element e): Zeit $O(1)$
- min(), deleteMin(): Zeit $O(n)$

Priority Queue mittels sortierter Liste:

- build($\{e_1, \ldots, e_n\}$): Zeit $O(n \log n)$
- insert(Element e): Zeit $O(n)$
- min(), deleteMin(): Zeit $O(1)$

$\Rightarrow$ Bessere Struktur als eine Liste notwendig!

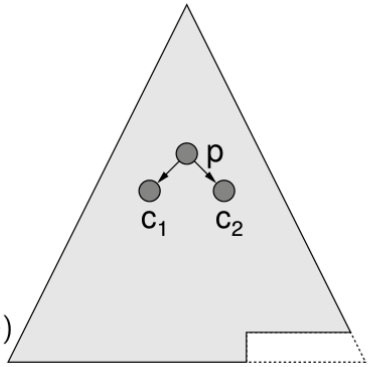
Binärer Heap

Idee: verwende Binärbaum

Bewahre zwei Invarianten:

- Form-Invariante: fast vollständiger Binärbaum
- Heap-Invariante:

  $\text{prio}(p) \leq \min\{\text{prio}(c1), \text{prio}(c2)\}$



Priority Queues Heaps

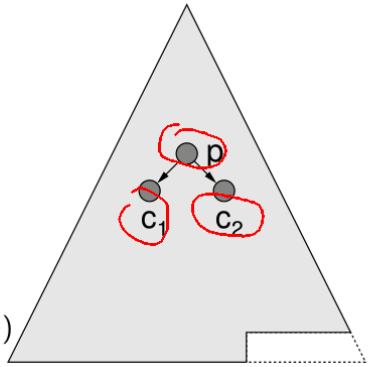
Binärer Heap

Idee: verwende Binärbaum

Bewahre zwei Invarianten:

- Form-Invariante: fast vollständiger Binärbaum
- Heap-Invariante:

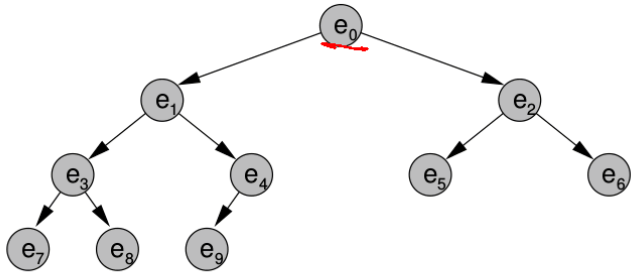
  $\text{prio}(p) \leq \min\{\text{prio}(c1), \text{prio}(c2)\}$

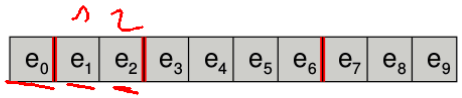


H. Seidl (TUM) GAD SS'16 249

Priority Queues Heaps

Binärer Heap als Feld





- Kinder von Knoten $H[i]$ in $H[2i+1]$ und $H[2i+2]$
- Form-Invariante: $H[0] \ldots H[n-1]$ besetzt
- Heap-Invariante: $H[i] \leq \min\{H[2i+1], H[2i+2]\}$

Priority Queues Heaps

Binärer Heap als Feld

insert(e)

- Form-Invariante: $H[n] = e$; siftUp(n); n++;
- Heap-Invariante:

  vertausche e mit seinem Vater bis

  $\text{prio}(H[\lfloor (k-1)/2 \rfloor]) \leq \text{prio}(e)$ für e in $H[k]$ (oder e in $H[0]$)

```
siftUp(i) {
  while (i > 0 ∧ prio(H[⌊(i − 1)/2⌋]) > prio(H[i])) {
    swap(H, i, ⌊(i − 1)/2⌋);
    i = (i − 1)/2;
  }
}
```

- Laufzeit: $O(\log n)$

Binärer Heap als Feld

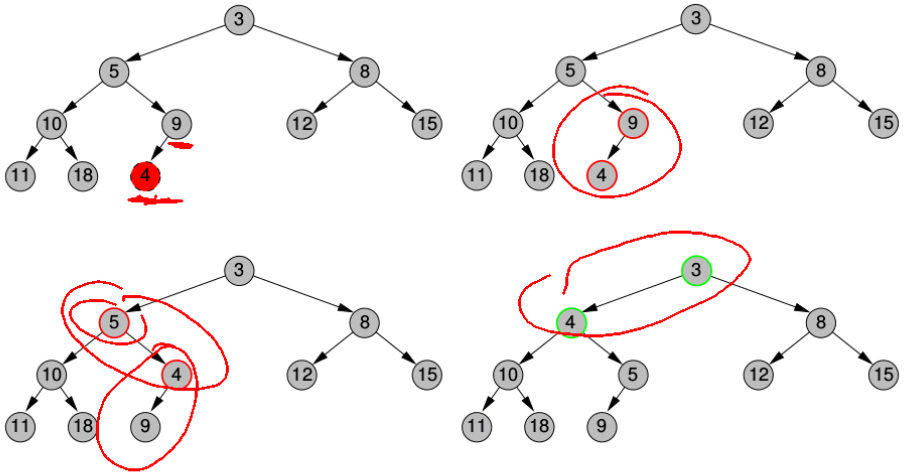
insert(e)

- Form-Invariante: $H[n] = e$; siftUp(n); n++;
- Heap-Invariante:

  vertausche e mit seinem Vater bis
  prio($H[\lfloor (k-1)/2 \rfloor]$) $\leq$ prio(e) für e in $H[k]$ (oder e in $H[0]$)

  siftUp(i) {
  while ($i > 0 \ \wedge$ prio($H[\lfloor (i-1)/2 \rfloor]$) > prio($H[i]$)) {
  swap($H, i, \lfloor (i-1)/2 \rfloor$);
  $i = (i-1)/2$;
  }
  }
- Laufzeit: $O(\log n)$

Heap - siftUp()



Binärer Heap als Feld

deleteMin()

- Form-Invariante:
  $e = H[0]$;
  n−−;
  $H[0] = H[n]$;
  siftDown(0);
  return e;
- Heap-Invariante: (siftDown)
  vertausche e (anfangs Element in $H[0]$) mit dem Kind, das die kleinere Priorität hat, bis e ein Blatt ist oder
  prio(e) $\leq \min\{\text{prio}(c_1(e)), \text{prio}(c_2(e))\}$.
- Laufzeit: $O(\log n)$