Datenstrukturen und Algorithmen

Übung 3

FS 2018

Programm von heute

1. Feedback letzte Übung
2. Wiederholung Theorie
3. Programmieraufgabe

Eier werfen

- Strategie für beliebig viele Eier?

Eier werfen

- Strategie für beliebig viele Eier?
  - Binäre Suche, höchstens $\log_2 n$ Versuche.

Eier werfen

- Strategie für beliebig viele Eier?
  - Binäre Suche, höchstens $\log_2 n$ Versuche.
- Strategie mit nur einem Ei?

Eier werfen

- Strategie für beliebig viele Eier?
  - Binäre Suche, höchstens $\log_2 n$ Versuche.
- Strategie mit nur einem Ei?
  - Von unten anfangen. n Versuche.

Eier werfen

- Strategie für beliebig viele Eier?
  - Binäre Suche, höchstens $\log_2 n$ Versuche.
- Strategie mit nur einem Ei?
  - Von unten anfangen. n Versuche.
- Strategie mit zwei Eiern:

Eier werfen

- Strategie für beliebig viele Eier?
  - Binäre Suche, höchstens $\log_2 n$ Versuche.
- Strategie mit nur einem Ei?
  - Von unten anfangen. n Versuche.
- Strategie mit zwei Eiern:
  - Versuche, mit s Versuchen auszukommen.
  - Kleiner werdende Intevalle
  - $s + (s-1) + (s-2) + \cdots + 2 + 1 = \sum_{i=1}^{n} i = \frac{s(s+1)}{2} \geq 100$. Daher $s = 14$.

Eier werfen

- Strategie für beliebig viele Eier?
  - Binäre Suche, höchstens $\log_2 n$ Versuche.
- Strategie mit nur einem Ei?
  - Von unten anfangen. n Versuche.
- Strategie mit zwei Eiern:
  - Versuche, mit s Versuchen auszukommen.
  - Kleiner werdende Intevalle
  - $s + (s-1) + (s-2) + \cdots + 2 + 1 = \sum_{i=1}^{n} i = \frac{s(s+1)}{2} \geq 100$. Daher $s = 14$.
  - $\sqrt{n}$

Selection-Algorithmus

- Was passiert bei vielen gleichen Elementen?
- $99, 99, \ldots, 99$, Pivot 99, kleiner Partition leer, grösser Parition hat $n - 1$ mal 99.
- Kann Laufzeit auf n^2 verschlechtern
- Solutions?

Selection-Algorithmus

- Bei Gleichheit mit Pivot, wechsle Partition ab.

Selection-Algorithmus

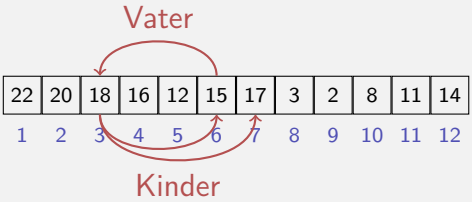
- Bei Gleichheit mit Pivot, wechsle Partition ab.
- Erweitere Algorithmus um explizit Anzahl gleicher Elemente zu behandeln.

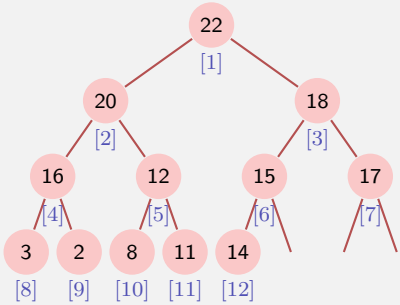
2. Wiederholung Theorie

Heap und Array

Baum $\rightarrow$ Array:

- Kinder$(i) = \{2i, 2i+1\}$
- Vater$(i) = \lfloor i/2 \rfloor$





Abhängig von Startindex![1]

[1] Für Arrays, die bei 0 beginnen: $\{2i, 2i+1\} \rightarrow \{2i+1, 2i+2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i-1)/2 \rfloor$

Algorithmus Versickern(A, i, m)

Input : Array A mit Heapstruktur für die Kinder von i. Letztes Element m.
Output : Array A mit Heapstruktur für i mit letztem Element m.

```
while 2i ≤ m do
    j ← 2i; // j linkes Kind
    if j < m and A[j] < A[j + 1] then
        j ← j + 1; // j rechtes Kind mit grösserem Schlüssel
    if A[i] < A[j] then
        swap(A[i], A[j])
        i ← j; // weiter versickern
    else
        i ← m; // versickern beendet
```

Algorithmus HeapSort(A, n)

Input : Array A der Länge n.
Output : A sortiert.

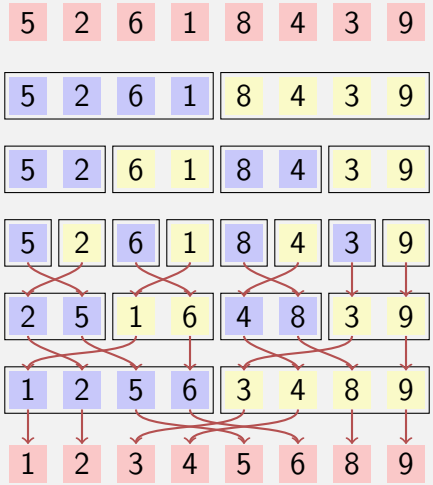
for $i \leftarrow n/2$ **downto** 1 **do**
    Versickere(A, i, n);

// Nun ist A ein Heap.

for $i \leftarrow n$ **downto** 2 **do**
    swap$(A[1], A[i])$
    Versickere$(A, 1, i-1)$

// Nun ist A sortiert.

Mergesort



Split

Split

Split

Merge

Merge

Merge

Algorithmus Rekursives 2-Wege Mergesort(A, l, r)

Input : Array A der Länge n. $1 \leq l \leq r \leq n$
Output : Array $A[l, \ldots, r]$ sortiert.

if $l < r$ **then**

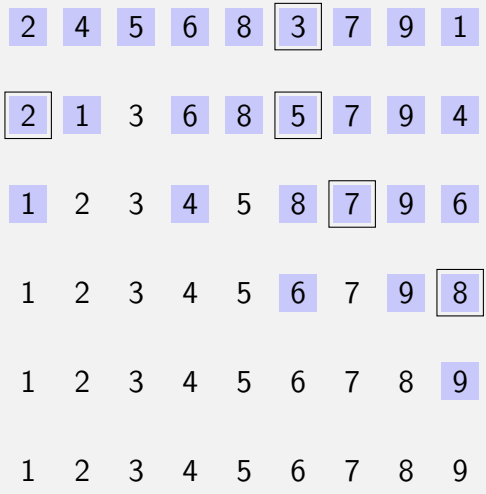
- $m \leftarrow \lfloor (l + r)/2 \rfloor$ // Mittlere Position
- Mergesort(A, l, m) // Sortiere vordere Hälfte
- Mergesort$(A, m + 1, r)$ // Sortiere hintere Hälfte
- Merge(A, l, m, r) // Verschmelzen der Teilfolgen

Algorithmus NaturalMergesort(A)

Input : Array A der Länge $n > 0$
Output : Array A sortiert

```
repeat
    r ← 0
    while r < n do
        l ← r + 1
        m ← l; while m < n and A[m + 1] ≥ A[m] do m ← m + 1
        if m < n then
            r ← m + 1; while r < n and A[r + 1] ≥ A[r] do r ← r + 1
            Merge(A, l, m, r);
        else
            r ← n
until l = 1
```

Quicksort (willkürlicher Pivot)



Algorithmus Quicksort($A[l, \ldots, r]$

Input : Array A der Länge n. $1 \leq l \leq r \leq n$.
Output : Array A, sortiert zwischen l und r.
if $l < r$ **then**
    Wähle Pivot $p \in A[l, \ldots, r]$
    $k \leftarrow$ Partition$(A[l, \ldots, r], p)$
    Quicksort$(A[l, \ldots, k-1])$
    Quicksort$(A[k+1, \ldots, r])$

Quicksort mit logarithmischem Speicherplatz

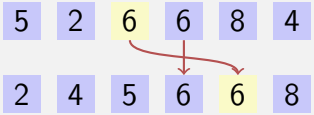
Input : Array A der Länge n. $1 \leq l \leq r \leq n$.
Output : Array A, sortiert zwischen l und r.

while $l < r$ **do**
    Wähle Pivot $p \in A[l, \ldots, r]$
    $k \leftarrow$ Partition$(A[l, \ldots, r], p)$
    if $k - l < r - k$ **then**
        Quicksort$(A[l, \ldots, k-1])$
        $l \leftarrow k + 1$
    else
        Quicksort$(A[k+1, \ldots, r])$
        $r \leftarrow k - 1$

Der im ursprünglichen Algorithmus verbleibende Aufruf an Quicksort$(A[l, \ldots, r])$ geschieht iterativ (Tail Recursion ausgenutzt!): die If-Anweisung wurde zur While Anweisung.

Stabile und in-situ-Sortieralgorithmen

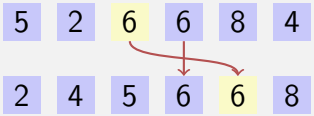
- Stabile Sortieralgorithmen ändern die relative Position von zwei gleichen Elementen nicht.



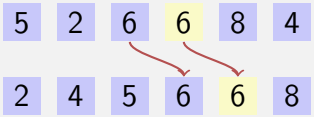
nicht stabil

Stabile und in-situ-Sortieralgorithmen

- Stabile Sortieralgorithmen ändern die relative Position von zwei gleichen Elementen nicht.



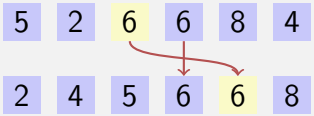
nicht stabil



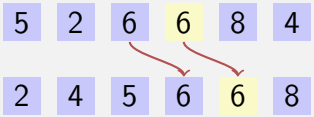
stabil

Stabile und in-situ-Sortieralgorithmen

- Stabile Sortieralgorithmen ändern die relative Position von zwei gleichen Elementen nicht.



nicht stabil



stabil

- In-situ-Algorithmen brauchen nur konstant viel zusätzlichen Speicher.

3. Programmieraufgabe

Typen als Template-Parameter

```
template <typename ElementType>
class vector{
        size_t size;
        T* elem;
public:
        ...
        vector(size_t s):
        size{s},
        elem{new ElementType[s]}{}
        ...
        ElementType& operator[](size_t pos){
                return elem[pos];
        }
        ...
}
```

Funktions-Templates

```
template <typename T> // square number
T sq(T x){
        return x*x;
}
template <typename Container, typename F>
void  apply(Container& c, F f){ // x <- f(x) forall x in c
        for(auto& x: c)
        x = f(x);
}
int main(){
        std::vector<int> v={1,2,3};
        apply(v,sq<int>);
        output(v); // 1 4 9
}
```

Fragen?

Fragen?

Let's get to work.