Auswählen nach Rang (Selektion)

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Geg.: Folge X von n Schlüsseln, eine Zahl k mit $1 \leq k \leq n$
Ges.: ein k-kleinster Schlüssel von X, also den Schlüssel x_k für X sortiert als $x_1 \leq x_2 \leq \cdots \leq x_n$

trivial lösbar in Zeit $O(kn)$ (k mal Minimum Entfernen), oder auch in Zeit $O(n \cdot \log n)$ (Sortieren)

Ziel: $O(n)$ Zeit Algorithmus für beliebiges k (z.B. auch $k=n/2$, "*Median* von X")

Vereinfachende Annahme für das Folgende: alle Schlüssel in X sind verschieden, also für sortiertes X gilt $x_1 < x_2 < \cdots < x_n$

Übung: Adaptieren Sie die folgenden Algorithmen, sodass diese Annahme nicht notwendig ist und die asymptotischen Laufzeiten erhalten bleiben.

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Geg.: Folge X von n Schlüsseln, eine Zahl k mit $1 \leq k \leq n$
Ges.: ein k-kleinster Schlüssel von X, also den Schlüssel x_k für X sortiert als $x_1 \leq x_2 \leq \cdots \leq x_n$

Idee: Dezimiere!

Wähle irgendein $z \in X$ und berechne $X_{<z} = \{x \in X \mid x < z\}$ und $X_{>z} = \{x \in X \mid x > z\}$
(z.B. durch Partitionsfunktion aus der letzten Vorlesung)

Es gilt dann $z = x_h$ mit $h-1 = |X_{<z}|$.

Fall $h=k$: $\Rightarrow z$ ist das gesuchte x_k

Fall $h>k$: $\Rightarrow x_k$ liegt in $X_{<z}$ und ist darin der k-kleinste Schlüssel ($X_{>z}$ ist irrelevant)

Fall $h<k$: $\Rightarrow x_k$ liegt in $X_{>z}$ und ist darin der $(k-h)$-kleinste Schlüssel ($X_{<z}$ ist irrelevant)

Also – x_k wird bei gegebenem z entweder sofort gefunden, oder man kann es rekursiv in $X_{<z}$ oder $X_{>z}$ finden. Welcher Fall für gewähltes z eintritt ist a priori nicht bekannt. Es wäre also günstig, wenn sowohl $X_{<z}$ als auch $X_{>z}$ "wenig" Schlüssel enthalten.

Sei $\frac{1}{2} < \alpha < 1$:
Wir nennen $z \in X$ einen **α-guten Splitter** für X, wenn sowohl
$|X_{<z}| \leq \alpha|X|$ als auch $|X_{>z}| \leq \alpha|X|$ gilt.

Algorithmus zum Finden des k-kleinsten Schlüssel in X (bei festgelegtem α)

Select(X , k)
1. If $|X|$ klein (z.B. $|X| \leq 50$) then verwende eine triviale Methode.
2. Finde einen α-guten Splitter $z \in X$ für X
3. Berechne $X_{<z} = \{x \in X | x < z\}$ und $X_{>z} = \{x \in X | x > z\}$ und bestimme $h = |X_{<z}| + 1$.
4. If $h=k$ then return z
   else if $h>k$ then return **Select**($X_{<z}$, k)
   else (* $h<k$ *) return **Select**($X_{>z}$, $k-h$)

Laufzeitanalyse: $T(n)$ Laufzeit von **Select**(X , k), wobei $n=|X|$
$S_\alpha(n)$ (erwartete) Laufzeit um α-guten Splitter zu finden

1. $a \cdot n$ für eine Konstante a
2. $S_\alpha(n)$
3. $c \cdot n$ für eine Konstante c
4. $T(\alpha n)$

$T(n) \leq a \cdot n$	wenn $n \leq 50$
$T(n) \leq c \cdot n + S_\alpha(n) + T(\alpha n)$	wenn $n > 50$

Wie findet man einen α-guten Splitter für X ?

Methode 1: Randomisiert

Wie findet man einen α-guten Splitter für X ?

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Methode 1: Randomisiert

Methode 2: Deterministisch (Blum, Floyd, Pratt, Rivest, Tarjan) für $\alpha = 7/10$

i) Teile X in n/5 Gruppen zu je 5 Schlüssel auf
ii) Bestimme für jede 5-er Gruppe den Median (3.-kleinsten Schlüssel)
iii) Verwende verschränkt rekursiv **Select**()
um den Median z dieser n/5 Mediane zu bestimmen

Behauptung: z ist ein α-guter Splitter für $\alpha = 7/10$.

Beweis:
eine Hälfte der 5er-Gruppen (die mit den Medianen $\geq$ z) enthält jeweils mindestes 3 Schlüssel größer als z.
$\Rightarrow$ es gibt mindestens $3\cdot(n/5)/2 = (3/10)n$ Schlüssel größer als z
$\Rightarrow |X_{>z}| \geq (3/10)n \Rightarrow |X_{<z}| \leq (7/10)n$

eine Hälfte der 5er-Gruppen (die mit den Medianen $\leq$ z) enthält jeweils mindestes 3 Schlüssel kleiner als z.
$\Rightarrow$ es gibt mindestens $3\cdot(n/5)/2 = (3/10)n$ Schlüssel kleiner als z
$\Rightarrow |X_{<z}| \geq (3/10)n \Rightarrow |X_{>z}| \leq (7/10)n$

Wie findet man einen α-guten Splitter für X ?

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Methode 2: Deterministisch (Blum, Floyd, Pratt, Rivest, Tarjan) für $\alpha = 7/10$

i) Teile X in n/5 Gruppen zu je 5 Schlüssel auf
ii) Bestimme für jede 5-er Gruppe den Median (3.-kleinsten Schlüssel)
iii) Verwende verschränkt rekursiv **Select**()
um den Median z dieser n/5 Mediane zu bestimmen

Behauptung: z ist ein α-guter Splitter für $\alpha = 7/10$.

Laufzeit:
Median einer 5-er Gruppe Bestimmen braucht konstant viel Zeit, O(1).
$\Rightarrow$ Schritt ii) braucht $(n/5)\cdot O(1) = O(n)$ Zeit.
Schritt i) brauch O(n) Zeit
Schritt iii) braucht T(n/5) Zeit

$\Rightarrow S_\alpha(n) \leq D\cdot n + T(n/5)$ für irgendeine Konstante D, wobei $\alpha = 7/10$.

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Sei $\frac{1}{2} < \alpha < 1$:
Wir nennen $z \in X$ einen **α-guten Splitter** für X, wenn sowohl
$|X_{<z}| \leq \alpha|X|$ als auch $|X_{>z}| \leq \alpha|X|$ gilt.

Laufzeitanalyse: $T(n)$ Laufzeit von **Select**(X , k), wobei $n=|X|$
$S_\alpha(n)$ (erwartete) Laufzeit um α-guten Splitter zu finden

$T(n) \leq a \cdot n$ wenn $n \leq 50$
$T(n) \leq c \cdot n + S_\alpha(n) + T(\alpha n)$ wenn $n > 50$

$\Rightarrow S_\alpha(n) \leq D \cdot n + T(n/5)$ für irgendeine Konstante D, wobei $\alpha = 7/10$.

$T(n) \leq a \cdot n$ wenn $n \leq 50$
$T(n) \leq c \cdot n + D \cdot n + T((1/5)n) + T((7/10)n)$
$= (c+D) \cdot n + T((1/5)n) + T((7/10)n)$ wenn $n > 50$

$\Rightarrow$ $T(n) \leq 10E \cdot n = O(n)$ mit $E = \max\{ a , c+D \}$
mit Induktion

Auswahl nach Rang kann in $O(n)$ „worst case“ Laufzeit gelöst werden.

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Simple Datenstrukturen

Wir setzen folgende Datenstrukturen und die grundlegenden Operationen darauf als bekannt voraus:

- **einfach verkettete Liste**
- **doppelt verkettete Liste**
- **Feld (array)**

Wir nehmen an, dass es ein "Memory Management" gibt, das Listenelemente wie auch Felder fester Größe zur Verfügung stellt und verwaltet. Wir nehmen an, das "zur Verfügung Stellen" (memory allocation) geschieht in konstanter Zeit.

Adressierbarer Stack (Stapel)

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Datenstruktur, die eine Folge von Stücken verwaltet, sodass immer nur am (rechten) Ende der Folge ein Stück eingefügt oder entfernt werden kann. Das letzte Element der Folge ("top") soll direkt zugreifbar sein wie auch das i-te Element, also das Element auf Position i.

Operationen:
bool S.isempty()
stück S.top()
stück S.pos(*int* i)
void S.push(*stück* x)
stück S.pop()

Realisierung durch Feld A[0..M-1]:

Invariante: die `t` Stücke des Stacks stehen in A[] an den Stellen `A[0..t−1]` mit `A[t−1]` das derzeit sichtbare Stück (top)

```
bool S.isempty()
  return (t==0)
```

```
stück S.top()
 if isempty()
  then error
  else return A[t-1]
```

```
stück S.pos( int i )
 if i<0 or i>= t
  then error
  else return A[i]
```

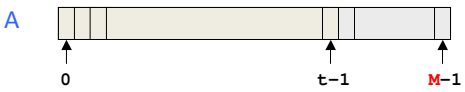
```
stück S.pop()
  if isempty()
   then error
   else t--
        return A[t]
```

```
void S.push( stück x )
  if (t<M)
   then A[t]:=x
        t++
   else error
```


SAARLAND UNIVERSITY
COMPUTER SCIENCE

Operationen:
bool S.isempty()
stück S.top()
stück S.pos(*int* i)
void S.push(*stück* x)
stück S.pop()

Realisierung durch Feld A[0..M-1]:



Invariante: die `t` Stücke des Stacks stehen in A[] an den Stellen `A[0..t−1]` mit `A[t−1]` das derzeit sichtbare Stück (top)

```
bool S.isempty()
  return (t==0)
```

```
stück S.top()
 if isempty()
  then error
  else return A[t-1]
```

```
stück S.pos( int i )
 if i<0 or i>= t
  then error
  else return A[i]
```

```
stück S.pop()
  if isempty()
   then error
   else t--
        return A[t]
```

```
void S.push( stück x )
  if (t<M)
   then A[t]:=x
        t++
   else error
```

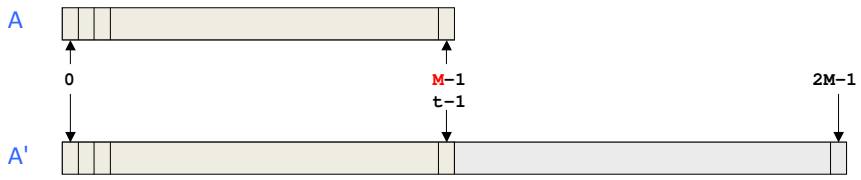
Jede Operation benötigt konstant viel Zeit.

Problem: Begrenzung der Größe des Stapels durch M ist unnatürlich

Abhilfe durch Re-allozieren von A[] auf doppelte Größe.

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Abhilfe durch Re-allozieren von A[] auf doppelte Größe.




```
void S.push( stück x )
  if (t<M)
   then A[t]:=x
        t++
   else A':=newarray of size 2M
        for (i=1;i<M;i++) A'[i]:=A[i]
        free A
        A:=A';  M:=2M;
        A[t]:=x
        t++
```

Zeit: O(M) bei Re-allocation
O(1) sonst

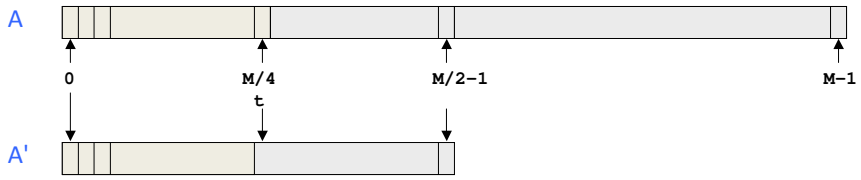
Also im *schlechtesten* Fall O(M).

Wieviel Zeit im "Normalfall"/"Durchschnitt"?

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Re-allozieren von A[] auf halbe Größe, wenn **zu wenig** Platz in A[] verwendet wird.

"zu wenig" .. $t \leq M/4$ (Warum nicht $t \leq M/2$?)




```
stück S.pop( )
  if isempty() then error
  else if t==M/4 then
       A':=newarray of size M/2
       for (i=1;i<t;i++) A'[i]:=A[i]
       free A
       A:=A';  M:=M/2;
  t --
  return A[t]
```

Zeit: O(M) bei Re-allocation
O(1) sonst

Also im *schlechtesten* Fall O(M).

Wieviel Zeit im "Normalfall"/"Durchschnitt"?

Amortisierte Zeitanalyse

SAARLAND UNIVERSITY
COMPUTER SCIENCE

push() und pop() brauchen meist O(1) Zeit, aber

O(M) Zeit be Re-allocation.

Zwischen 2 Re-allocationen gibt es mindestens M/4 pop() oder push() Operationen.

D.h. im Durchschnitt (über eine Folge von Operationen) braucht jedes pop() oder push() nur O(1) Zeit. Warum?

Idee: Zeit = Geld.
Jede Rechenoperationsdurchführung muss bezahlt werden.
Jedes pop() und push() bringen 5€ ins System (nehmen wir an)
1€ bezahlt für die normalen konstanten Kosten
4€ werden im System gespart
Bei Re-allocation sind dann mindestens $4 \cdot (M/4) = M$ Euros im System, mit denen die O(M) Kosten der Re-allocation bezahlt werden können.

Amortisierte Zeitanalyse

SAARLAND UNIVERSITY
COMPUTER SCIENCE

FAZIT: Wenn push() und pop() jeweils mit 5€ (= O(1) Zeit) ausgestattet werden, dann gibt es immer genug Geld im System, um die Durchführung dieser Operationen zu bezahlen.

D.h. Wenn eine Folge von n push() und pop() Operationen durchgeführt werden, dann brauchen sie ingesamt höchstens O(n) Zeit, bzw. im Durchschnitt (über die Folge) braucht jede Operation O(1) Zeit.

Man sagt: Die **amortisierte** Laufzeit von push() und pop() ist O(1).

Amortisierte Komplexität

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Seien $op_1,\ldots,op_k$ Update Operationen auf einer Datenstruktur.
Seien $f_1,\ldots,f_k$ nicht fallende Funktionen.

Man sagt, $op_1,\ldots,op_k$ haben amortisierte Kosten $f_1,\ldots,f_k$, wenn jede hinreichend lange Folge von Updateoperationen mit n_i Operationen vom Typ op_i, insgesamt Zeit $O(n_1 \leq f_1(N) + \ldots + n_k \leq f_k(N))$ braucht.

Dabei ist N die maximale Größe der Datenstruktur während der Updatefolge.