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.

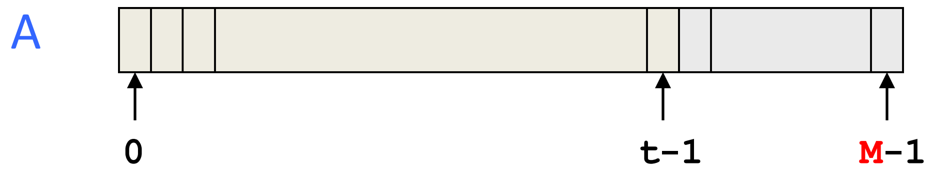
Adressierbarer Stack (Stapel)

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]
```

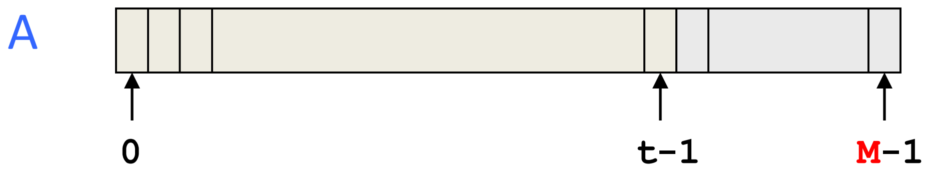
```
stück S.pop()
  if isempty()
    then error
    else t--
         return A[t]
```

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

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]
```

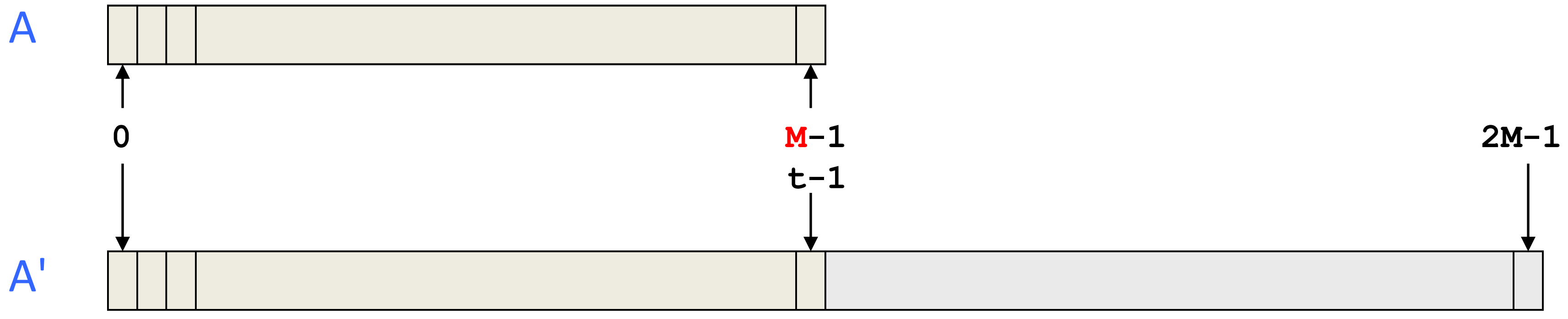
```
void S.push( stück x )
  if (t<M)
    then A[t]:=x
         t++
    else error
```

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.

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++
```

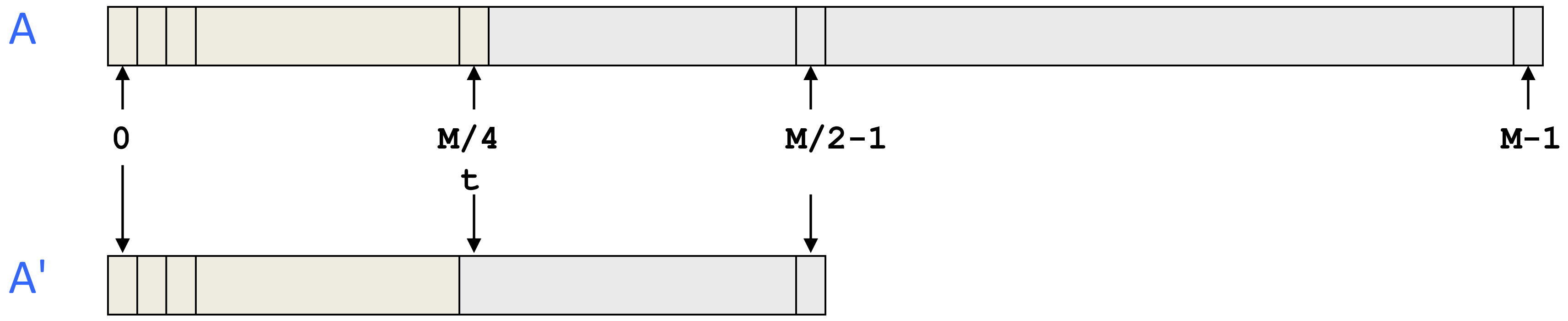
Zeit: $O(M)$ bei Re-allocation
$O(1)$ sonst

Also im *schlechtesten* Fall $O(M)$.

Wieviel Zeit im "Normalfall"/"Durchschnitt"?

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

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

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 $O(1)$.

Wörterbücher

Dynamische Menge an Elementen, die über Schlüssel definiert sind:

Paare(Schlüssel, Wert)

Operationen:

element search*(schlüssel* x*)*
void insert*(element* e*)*
void delete(*schlüssel* x)
element max/min()
void traverse()

Realisirung:

- Verketette Liste
- Feld
- BitVector mit Kapazität m

	Search	Insert	Delete	Max/Min	Traverse
Liste	O(n)	O(n)	O(n)	O(n)	$O(n^2)$
Sortiertes Feld	O(log n)	O(n)	O(n)	O(1)	O(n)
BitVector	O(1)	O(1)	O(1)	O(m)	O(m)

Amortization

Worst Case Performance measures of a data structure DS:

S(n)	maximal space needed, when DS holds at most n items
Q(n)	maximal query time, when DS holds at most n items
I(n)	maximal time for insertion, when DS holds at most n items

Amortization

Worst Case Performance measures of a data structure DS:

$S(n)$	maximal space needed, when DS holds at most n items
$Q(n)$	maximal query time, when DS holds at most n items
$I(n)$	maximal time for insertion, when DS holds at most n items

Upper bounds that hold for **every** possible operation.

In reality some particular operations cheap, some more expensive.

Sequence of m_I insertions and m_Q queries:

$$m_I \cdot I(n) + m_Q \cdot Q(n)$$

time bound may be a much too pessimistic bound.

Example: Power$_{\text{of 2}}$dictionary

$S = \{ 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 \}$ $n = 12$

Example: Power$_{\text{of } 2}$dictionary

S = { 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }   n = 12

2^0:	[ ]
2^1:	[ ]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

Example: Power$_{\text{of }2}$dictionary

S = { 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 } n = 12

2^0:	[ ]
2^1:	[ ]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

Set S is stored in distinct-sized sorted arrays, each has size a power of 2.

Example: Power$_{\text{of }2}$dictionary

S = { 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }   n = 12

2^0:	[ ]
2^1:	[ ]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

Set S is stored in distinct-sized sorted arrays, each has size a power of 2.

Query for x: binary search in each of the $\leq \log n$ arrays

$\Rightarrow$ $Q(n) = O(\log^2 n)$.

Example: Power$_{\text{of 2}}$dictionary

S = { 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 } n = 12

2^0:	[ ]
2^1:	[ ]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

Insert 27

Example: Power$_{\text{of }2}$dictionary

S = { 27 ,23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }

n = 13

2^0:	[27]
2^1:	[ ]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

Example: Power$_{\text{of 2}}$dictionary

S = { 27 ,23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }

n = 13

2^0:	[27]
2^1:	[ ]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

Insert 17

Example: Power$_{\text{of }2}$dictionary

S = { 17 , 27 , 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }
n = 14

2^0:	[27] [17]
2^1:	[ ]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

Example: Power$_{\text{of }2}$dictionary

S = { 17 , 27 , 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }

n = 14

2^0:	[27] [17]
2^1:	[ ]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

2^0:	[ ]
2^1:	[17 , 27]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

Example: Power$_{\text{of } 2}$dictionary

S = { 17 , 27 , 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }

n = 14

2^0:	[ ]
2^1:	[17 , 27]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

Insert 21

Example: Power$_{\text{of }2}$dictionary

S = { 21 , 17 , 27 , 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }

n = 15

2^0:	[21]
2^1:	[17 , 27]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

Example: Power$_{\text{of }2}$dictionary

S = { 21 , 17 , 27 , 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }
n = 15

2^0:	[21]
2^1:	[17 , 27]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

Insert 63

S = { 63 , 21 , 17 , 27 , 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }
n = 16

2^0:	[21] [63]
2^1:	[17 , 27]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

S = { 63 , 21 , 17 , 27 , 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }

n = 16

2^0:	[21] [63]
2^1:	[17 , 27]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

2^0:	[ ]
2^1:	[17 , 27] [21 , 63]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

S = { 63 , 21 , 17 , 27 , 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }
n = 16

2^0:	[ ]
2^1:	[17 , 27] [21 , 63]
2^2:	[11 , 23 , 45 , 84]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

2^0:	[ ]
2^1:	[ ]
2^2:	[11 , 23 , 45 , 84] [17 , 21 , 27 , 63]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

S = { 63 , 21 , 17 , 27 , 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }
n = 16

2^0:	[ ]
2^1:	[ ]
2^2:	[11 , 23 , 45 , 84] [17 , 21 , 27 , 63]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61]

2^0:	[ ]
2^1:	[ ]
2^2:	[ ]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61] [11 , 17 , 21 , 23 , 27 , 45 , 63 84]

S = { 63 , 21 , 17 , 27 , 23 , 11 , 84 , 33 , 20 , 45 , 29 , 41 , 61 , 39 , 55 , 16 }

n = 16

2^0:	[ ]
2^1:	[ ]
2^2:	[ ]
2^3:	[16 , 20 , 29 , 33 , 39 , 41 , 55 , 61] [11 , 17 , 21 , 23 , 27 , 45 , 63 84]

2^0:	[ ]
2^1:	[ ]
2^2:	[ ]
2^3:	[ ]
2^4:	[11 , 16 , 17 , 20 , 21 , 23 ,27 , 29 , 33 , 39 , 41 , 45 , 55 , 61 , 63 , 84]

Example: Power$_{\text{of 2}}$dictionary

Set S is stored in distinct-sized sorted arrays, each has size a power of 2.

Inserting x:

```
create new array of size 1 for $x$
while there are two arrays of the same size do
    merge them
```

Time proportional to the size of last merged array, which in the worst case is n.

$$\Rightarrow \; I(n) = O(n).$$

Singe insertion worst case time $I(n)=O(n)$

Sequence of m insertions w.c. time $m \cdot I(n) = O(m \cdot n)$?

Singe insertion worst case time $I(n)=O(n)$

Sequence of m insertions w.c. time $m \cdot I(n) = O(m \cdot n)$?

Much too pessismistic !!

Only few insertions take linear time.
Most insertions take only constant time !!

The cost of the few expensive insertions can be **“amortized”** over the many cheap insertions.

$I_A(n)$ amortized insertion time bound

$Q_A(n)$ amortized query time bound

if

every sequence of m_I insertions and m_Q queries takes at most time

$$m_I \cdot I_A(n) + m_Q \cdot Q_A(n)$$

where n is the maximum size of the data structure in the sequence

$I_A(n)$ amortized insertion time bound

$Q_A(n)$ amortized query time bound

if

every **sufficiently long** sequence of m_I insertions and m_Q queries takes at most time

$$m_I \cdot I_A(n) + m_Q \cdot Q_A(n)$$

where n is the maximum size of the data structure in the sequence

$I_A(n)$ amortized insertion time bound

$Q_A(n)$ amortized query time bound

if

every sequence of m_I insertions and m_Q queries takes at most time

$$m_I \cdot I_A(n) + m_Q \cdot Q_A(n) + \mathbf{f(n)}$$

where n is the maximum size of the data structure in the sequence

$I_A(n) = g(n)$ and $Q_A(n) = h(n)$ means

Considering **any** (sufficiently long) sequence,

the average insertion time in that sequence must be at most $g(n)$

the average query time in that sequence must be at most $h(n)$

Claim: Power$_{\text{of 2}}$dictionary has amortized performance

$$I_A(n) = O(\log n) \quad \text{and} \quad Q_A(n) = O(\log^2 n)$$

Claim: Power$_{\text{of 2}}$dictionary has amortized performance

$I_A(n) = O(\log n)$ and $Q_A(n) = O(\log^2 n)$

"bank account" proof: sequence of n insertions

time is money: 1 € pays for two steps in a merge

Each insert operation brings $k = \lfloor \log n \rfloor$ € into the system and places 1 € into bank account of each array.

When an array of size 2^k is merged there will be exactly 2^k € in its bank account to pay for the merge.

Amortized insertion cost = money put into the system by each insertion = $O(\log n)$