Simple Datenstrukturen

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

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

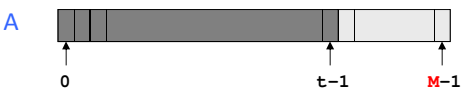
Wir nehmen an, dass es ein "Memory Management" gibt, das Listenelemente wie auch Felder fester Größe zur Verfügung stellt und verwaltet. Wir nehmen an, das "zur Verfügung Stellen" (memory allocation) geschieht in konstanter Zeit.

Stack (Stapel)

Datenstruktur, die eine Menge von Stücken verwaltet, sodass immer das zuletzt eingefügte, aber noch nicht entfernte Stück "sichtbar" ist.
(LIFO --- Last In First Out)

Operationen:
bool S.isempty()
stück S.top()
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)
```

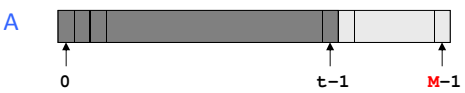
```
stück S.top()
  if isempty()
    then error
    else return A[t-1]
```

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

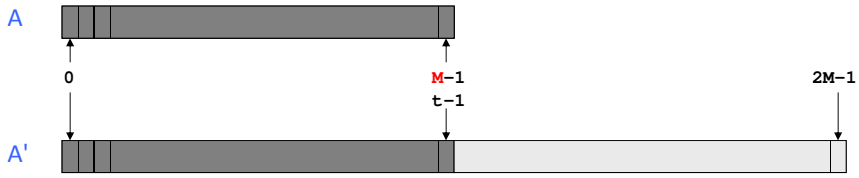
```
void S.push( stück x )
  if (t<M)
    then A[t]:=x
         t++
    else error
```

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++
```

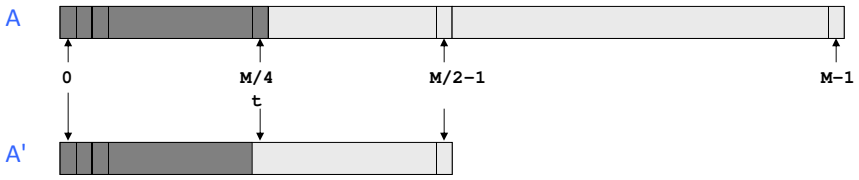
Zeit: O(M) bei Re-allocation
O(1) sonst

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·M/4 (Warum nicht t·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·(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).

Amortisierte Komplexität

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

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

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