1.1 Abstrakte Datentypen
1.2 Lineare Strukturen
1.3 Bäume
1.4 Prioritätsschlangen
1.5 Graphen

Lineare Strukturen

- Sequenz $\{x_1, \ldots, x_n\}$ von beliebigen Datenobjekten x_i
- Typische Operationen
  - Füge y am Anfang / am Ende / hinter x_i ein
  - Ersetze x_i durch y
  - Entferne x_i
  - Lese das i-te Element

Lineare Strukturen

- Typische Operationen
  - Verknüpfe zwei Sequenzen
  - Zerlege eine Sequenz
  - Bestimme die Länge der Sequenz
  - Teste, ob ein Element y vorhanden ist
  - Sortiere die Elemente x_i
  - …

1.2 Lineare Strukturen

1.2.1 Listen

1.2.2 Warteschlangen

1.2.3 Stacks

1.2.1 Listen

- $L = \{x_1, \ldots, x_n\}$
- Zugriff auf beliebige Elemente x_i
  - Per Index (random access)
    - Get(i)
  - Per Marker (sequential access)
    - GetFirst()
    - GetNext()
    - GetPrevious()

Random Access

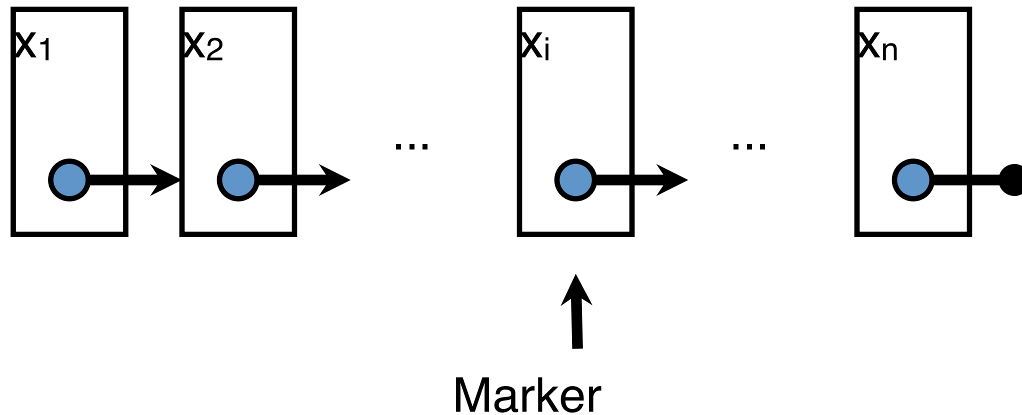
- Implementierung durch Arrays L[ ]
- Get(i) = L[i]
- Nachteile
  - Elemente löschen erzeugt Lücken oder alle Elemente mit höherem Index müssen verschoben werden (garbage collection)
  - Statische Obergrenze für Listenlänge

Sequential Access

- Implementierung durch Pointer oder Container
- Marker zeigt auf aktuelle Position
- Nachteil: Elementzugriff erfordert lineare Suche (kann jedoch meistens vermieden werden, z.B. „for each").
- Vorteil: beliebiges Erweitern und Löschen

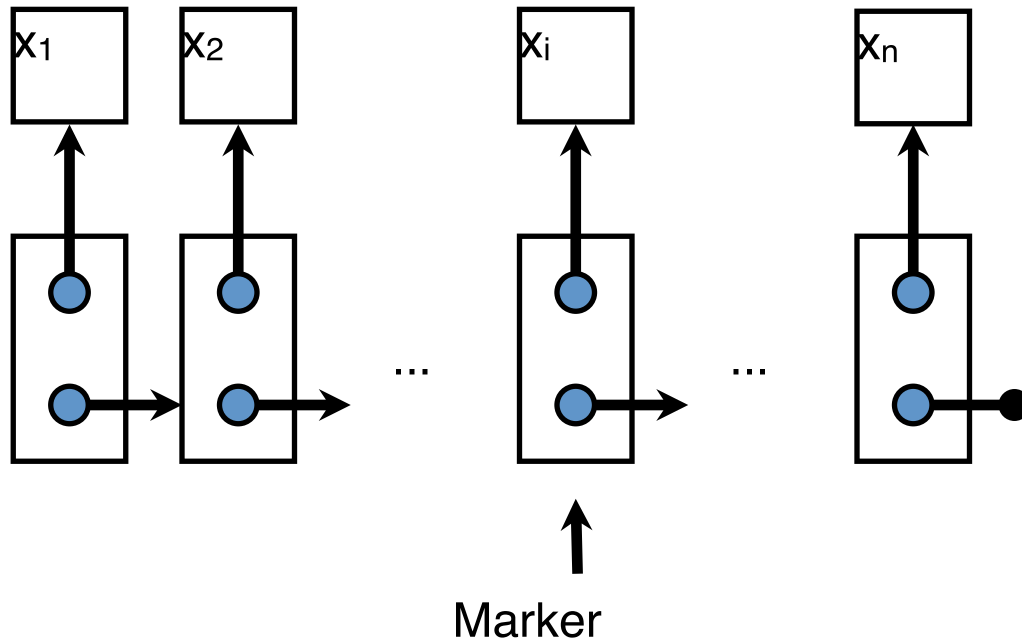
Sequential Access

- Pointer



Sequential Access

- Container



Listen

- Wertebereich : $L = \{\,\} \cup W \cup W^2 \cup W^3 \cup \ldots$
- Create : $\rightarrow L$
- Get : $L \rightarrow W$
- Reset : $L \rightarrow L$
- Next : $L \rightarrow L$
- Insert : $W \times L \rightarrow L$
- Delete : $L \rightarrow L$
- Empty : $L \rightarrow$ Bool
- IsLast : $L \rightarrow$ Bool

Listen

- Achtung: der Marker beschreibt einen Zustand, was sich mit funktionalen Axiomen schlecht formulieren läßt.

- Standard-Trick: führe ein zusätzliches Prädikat Insert* ein, dass aber keine weitere Listen-Funktion darstellt.

Listen

• Empty(Create())	= true
Empty(Insert(x,z))	= false
Empty(Insert*(x,z))	= false
• IsLast(Create())	= true
IsLast(Insert(x,z))	= false
IsLast(Insert*(x,z))	= IsLast(z)
• Get(Insert(x,z))	= x
Get(Insert*(x,z))	= Get(z)
Insert(x,Insert*(y,z))	= Insert*(y,Insert(x,z))

Listen

- Delete(Create()) = Create()
  Delete(Insert(x,z)) = z
  Delete(Insert*(x,z)) = Insert*(x,Delete(z))
- Next(Create()) = Create()
  Next(Insert(x,z)) = Insert*(x,z)
  Next(Insert*(x,z)) = Insert*(x,Next(z))
- Reset(Create()) = Create()
  Reset(Insert(x,z)) = Insert(x,z)
  Reset(Insert*(x,z)) = Insert(x,Reset(z))

Listen

- Satz: Jede Liste hat die Form

  $\text{Insert}^*(x_1,..\text{Insert}^*(x_i,\text{Insert}(x_{i+1},..\text{Insert}(x_n,\text{Create}())))))$

- Beweis durch vollständige Induktion über die Anzahl der verwendeten Operationen
  - Create() … n=0
  - Jede andere Operation erhält die Form

Get()

- Get(Insert*(x_1, … Insert*(x_i,Insert(x_{i+1},…Insert(x_n,Create()))))))
- Get(… Insert*(x_i,Insert(x_{i+1}, … Insert(x_n,Create())))))
- Get(Insert*(x_i,Insert(x_{i+1}, … Insert(x_n,Create())))))
- Get(Insert(x_{i+1}, … Insert(x_n,Create())))
- x_{i+1}

Next()

- $\text{Next}(\text{Insert}^*(x_1,\ldots\text{Insert}^*(x_i,\text{Insert}(x_{i+1},\ldots\text{Insert}(x_n,\text{Create}()))))))$
- $\text{Insert}^*(x_1,\text{Next}(\ldots\text{Insert}^*(x_i,\text{Insert}(x_{i+1},\ldots\text{Insert}(x_n,\text{Create}()))))))$
- $\text{Insert}^*(x_1,\ldots\text{Next}(\text{Insert}^*(x_i,\text{Insert}(x_{i+1},\ldots\text{Insert}(x_n,\text{Create}()))))))$
- $\text{Insert}^*(x_1,\ldots\text{Insert}^*(x_i,\text{Next}(\text{Insert}(x_{i+1},\ldots\text{Insert}(x_n,\text{Create}()))))))$
- $\text{Insert}^*(x_1,\ldots\text{Insert}^*(x_i,\text{Insert}^*(x_{i+1},\ldots\text{Insert}(x_n,\text{Create}())))))$

Insert()

- Insert(y,Insert*(x_1,..Insert*(x_i,Insert(x_{i+1},..Insert(x_n,Create()))))))
- Insert*(x_1,Insert(y,..Insert*(xi,Insert(x_{i+1},..Insert(x_n,Create()))))))
- Insert*(x_1,..Insert(y,Insert*(x_i,Insert(x_{i+1},..Insert(x_n,Create()))))))
- Insert*(x_1,..Insert*(x_i,Insert(y,Insert(x_{i+1},..Insert(x_n,Create()))))))

Delete()

- Delete(Insert*(x_1,...Insert*(x_i,Insert(x_{i+1},...Insert(x_n,Create()))))))
- Insert*(x_1,Delete(...Insert*(x_i,Insert(x_{i+1},...Insert(x_n,Create()))))))
- Insert*(x_1,...Delete(Insert*(x_i,Insert(x_{i+1},...Insert(x_n,Create()))))))
- Insert*(x_1,...Insert*(x_i,Delete(Insert(x_{i+1},...Insert(x_n,Create()))))))
- Insert*(x_1,...Insert*(x_i,...Insert(x_n,Create())))

Weitere Funktionen

- Overwrite(y,Insert(x,z)) = Insert(y,z)
  Overwrite(y,Insert*(x,z)) = Insert*(x,Overwrite(y,z))
- Join(Create(),z) = z
  Join(Insert(x,y),z) = Insert*(x,Join(y,z))
  Join(Insert*(x,y),z) = Insert*(x,Join(y,z))
- …

Prof. Dr. Leif Kobbelt, Thomas Ströder, Fabian Emmes, Sven Middelberg, Michael Kremer
RWTH AACHEN UNIVERSITY

Implementierung

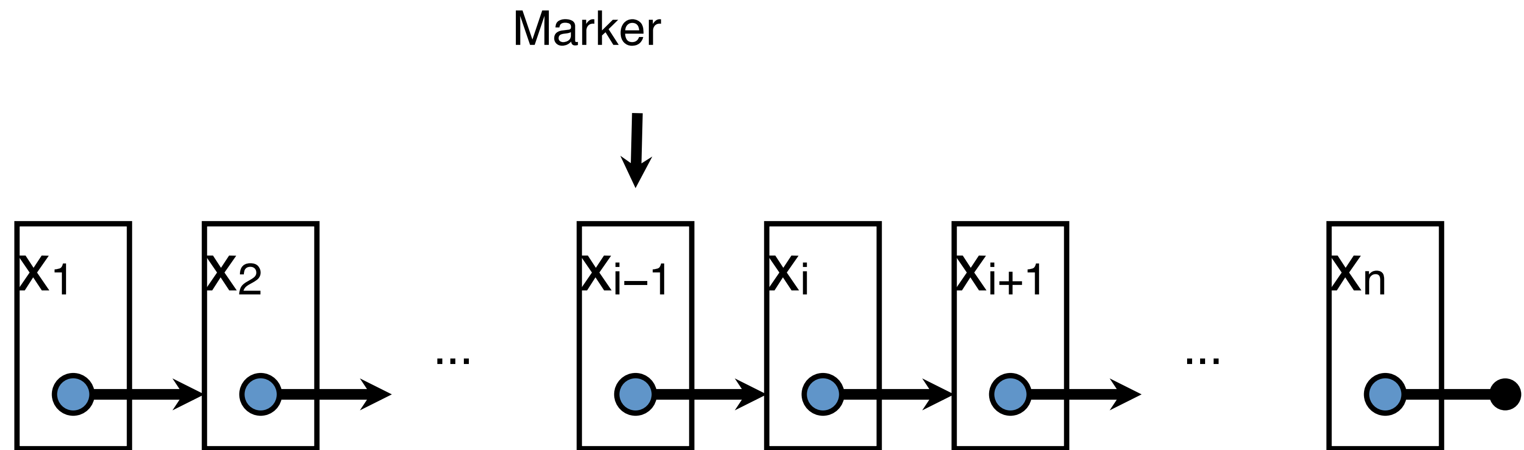
- Finden, Lesen, Überschreiben
- Löschen (Sonderfälle bei leerer Liste)
- Einfügen (Sonderfälle an den Enden)
- Anchor, Sentinel
- Einfach / Doppelt verkettete Listen

Einfach verkettete Liste

- class Element {
  Datentyp X
  Element next
  }

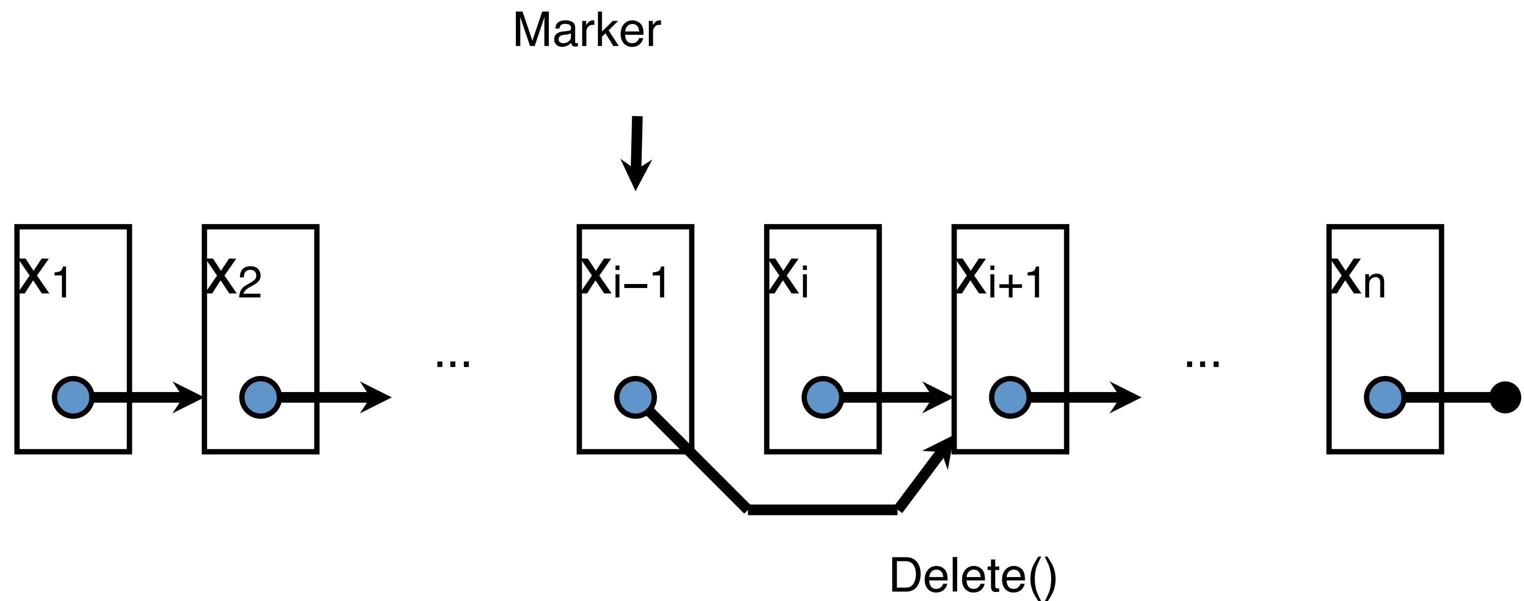
Sequential Access

- Delete



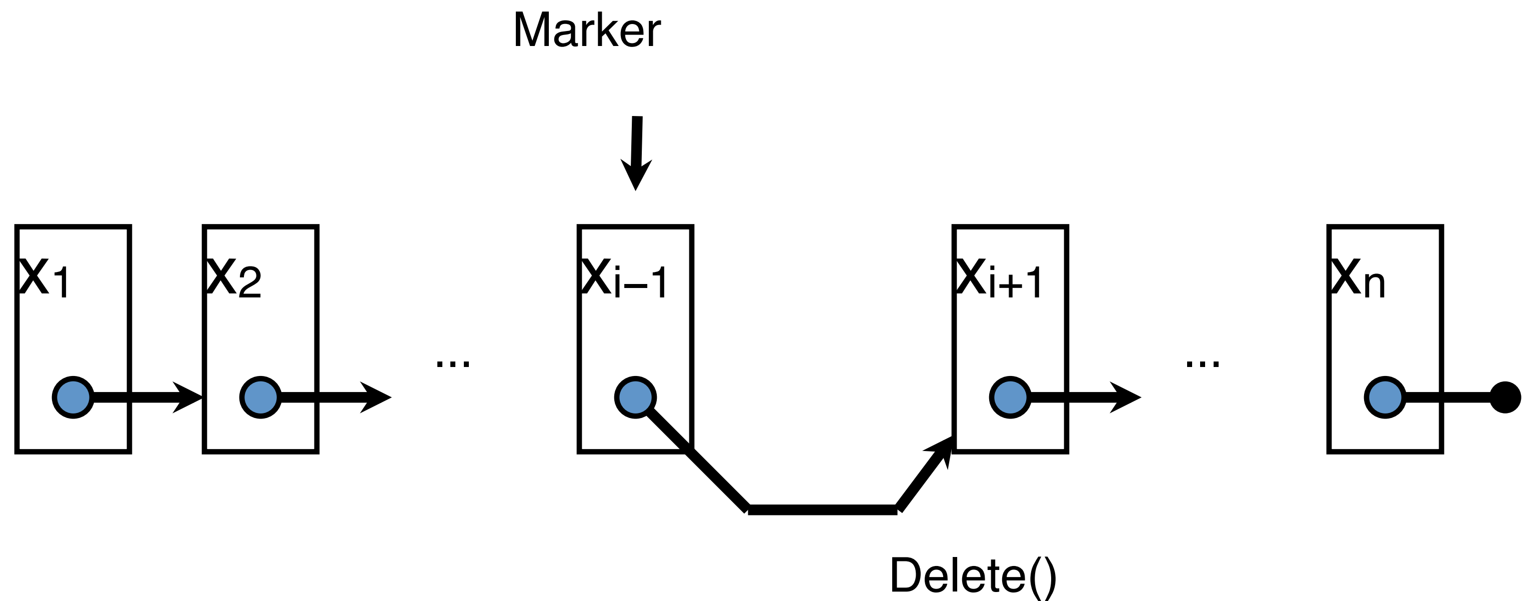
Sequential Access

- Delete



Sequential Access

- Delete

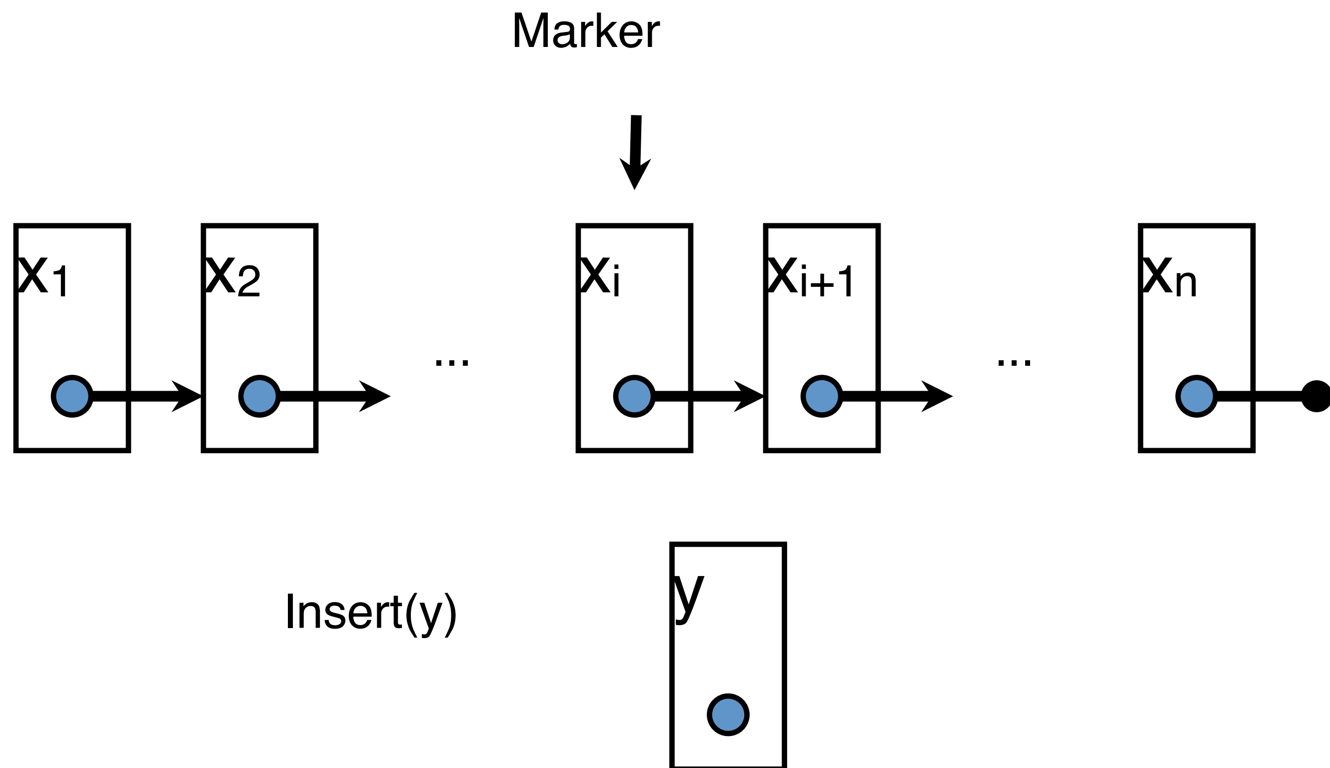


Einfach verkettete Listen

- Delete()
  Marker.next ← Marker.next.next

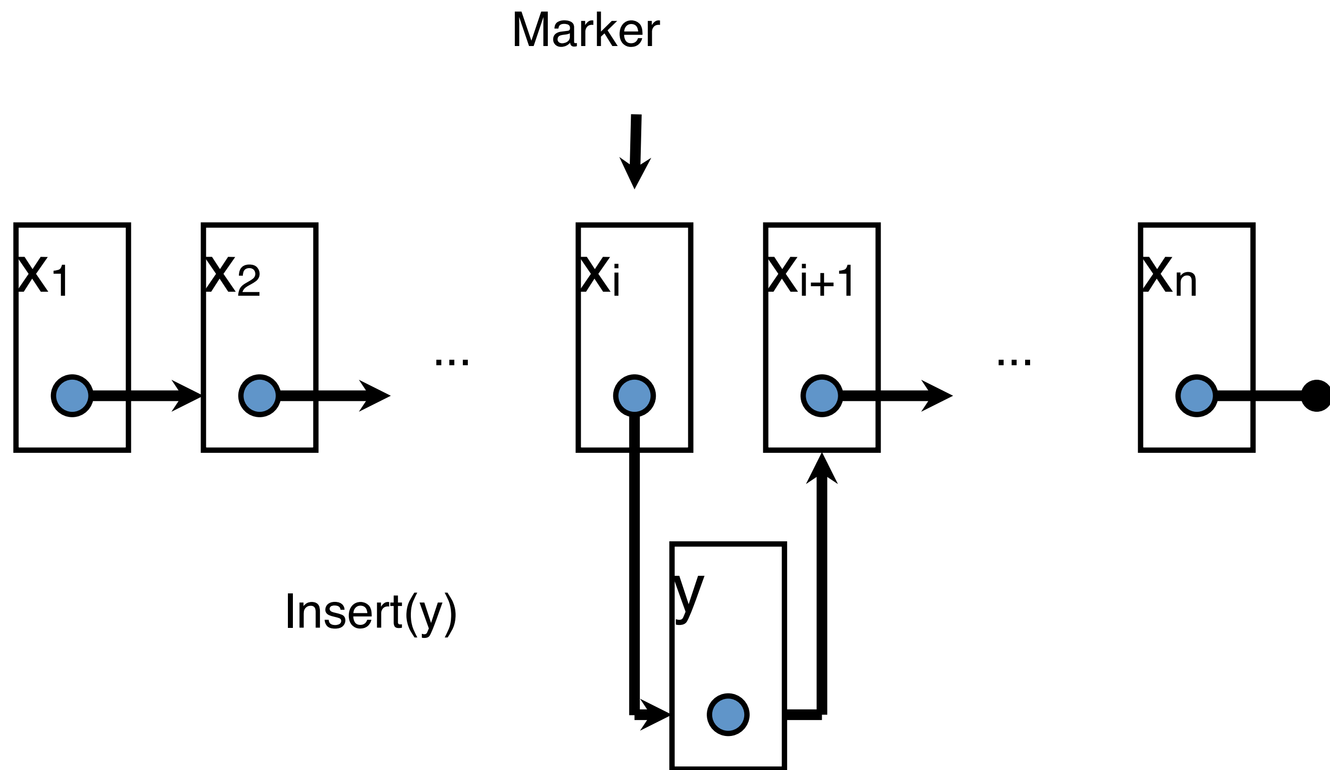
Sequential Access

- Insert



Sequential Access

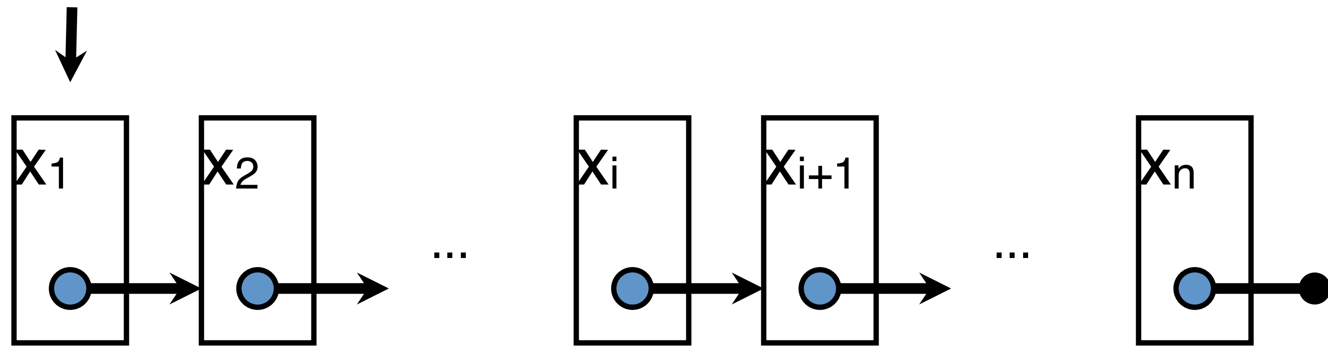
- Insert



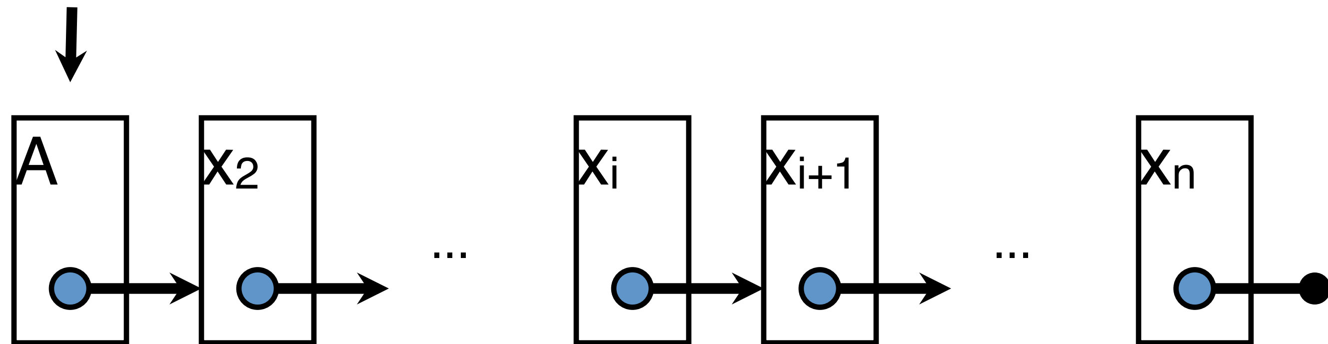
Einfach verkettete Listen

- Insert(Y)
  - Y.next ← Marker.next
  - Marker.next ← Y

Modifikation am Listenanfang



Modifikation am Listenanfang



Anchor
Erstes Element enthält keine Daten

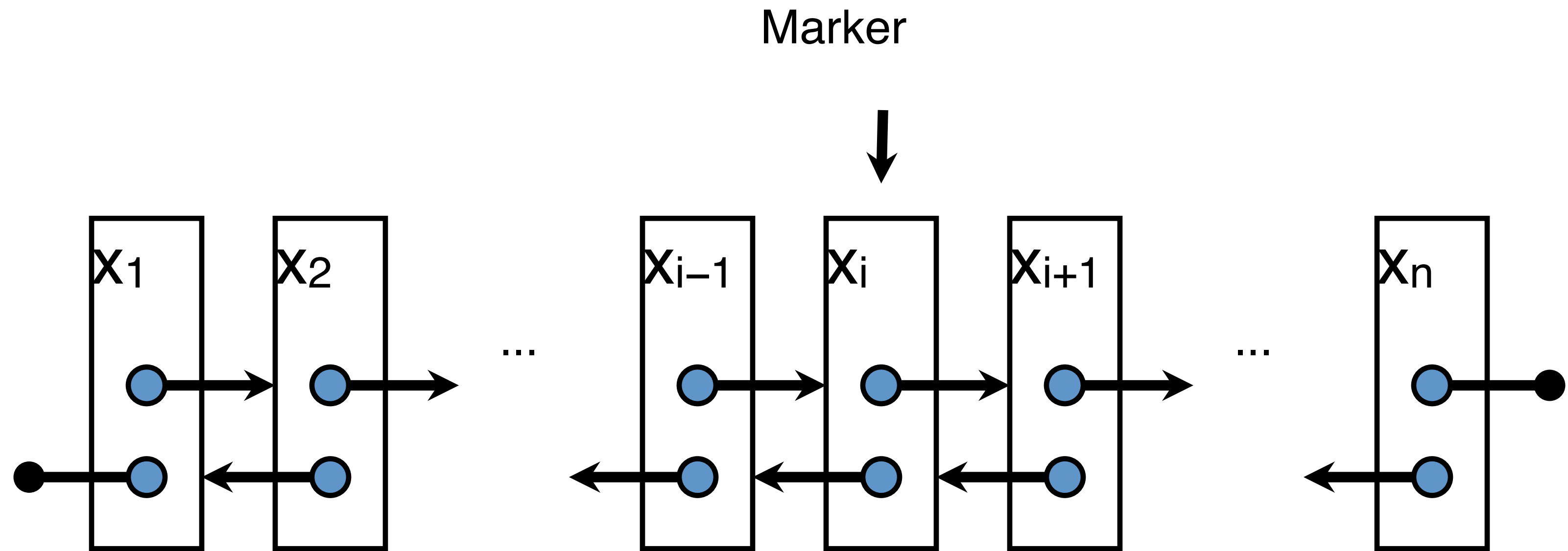
Doppelt verkettete Listen

- Flexiblerer Zugriff (upstream/downstream)
  - Next(), Prev()
- Bisher: Einfügen/Löschen nach dem Marker (wg. Pointer update)
- Jetzt: Marker zeigt auf aktuelles Element

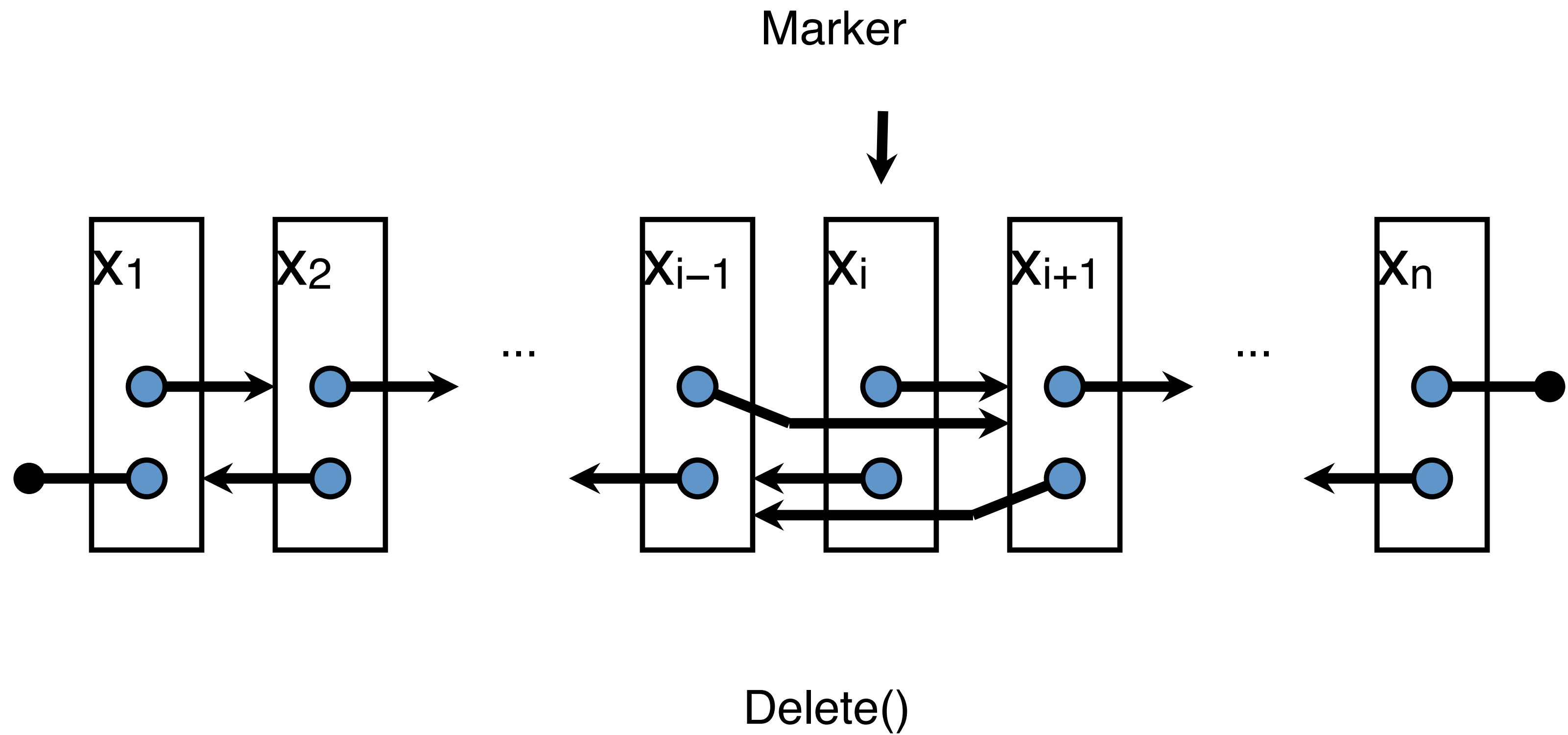
Doppelt verkettete Liste

- class Element {
  Datentyp X
  Element prev, next
  }

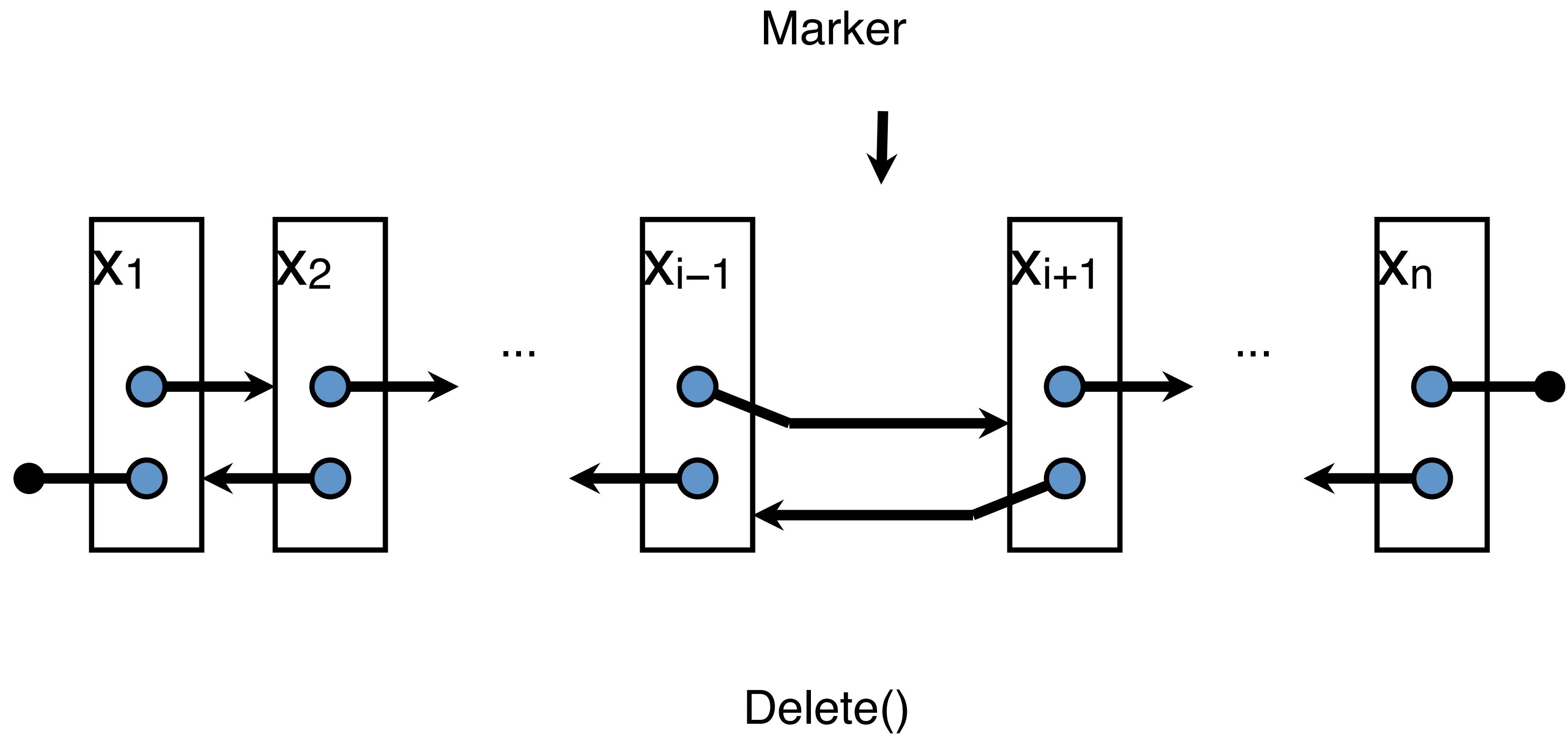
Doppelt verkettete Listen



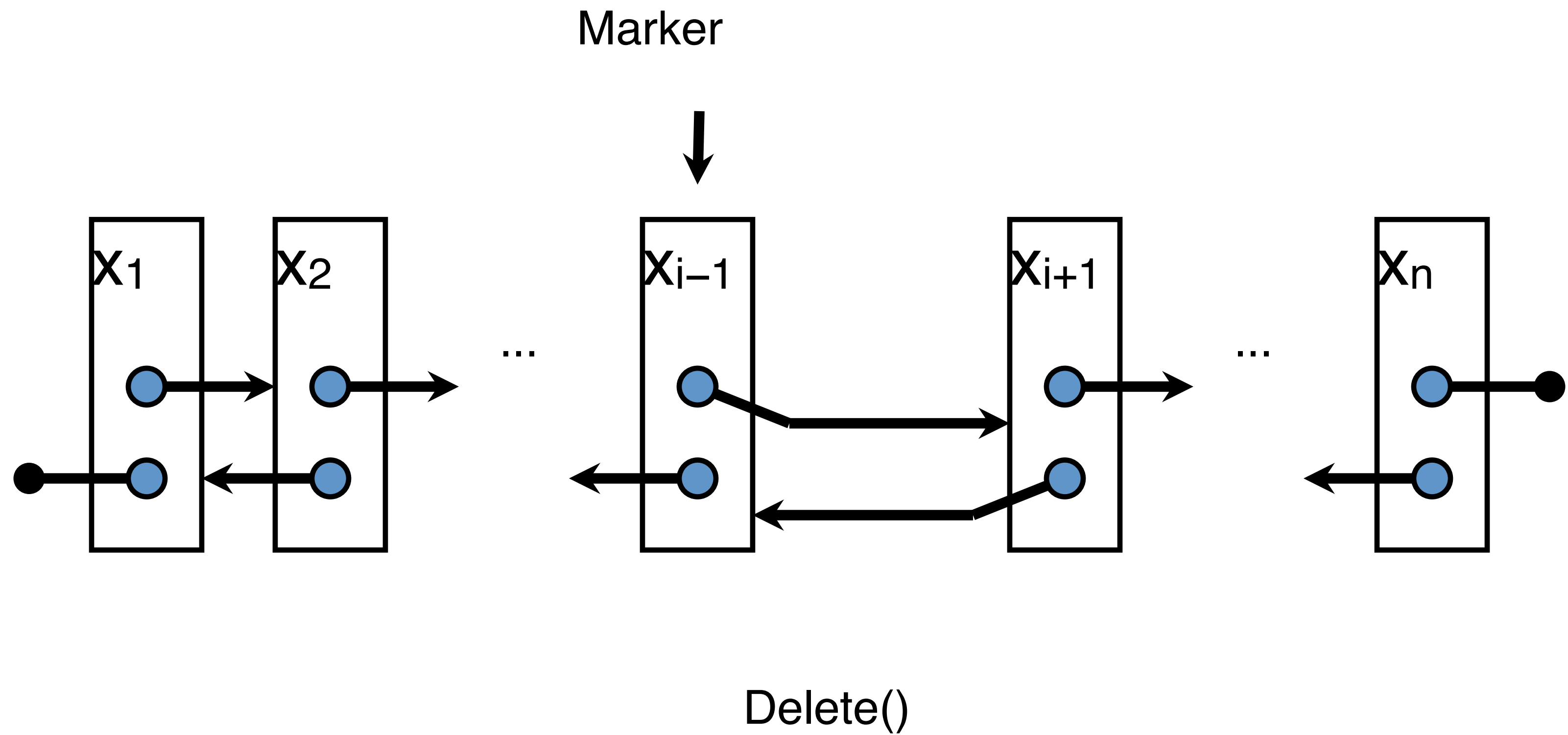
Doppelt verkettete Listen



Doppelt verkettete Listen



Doppelt verkettete Listen



Doppelt verkettete Listen

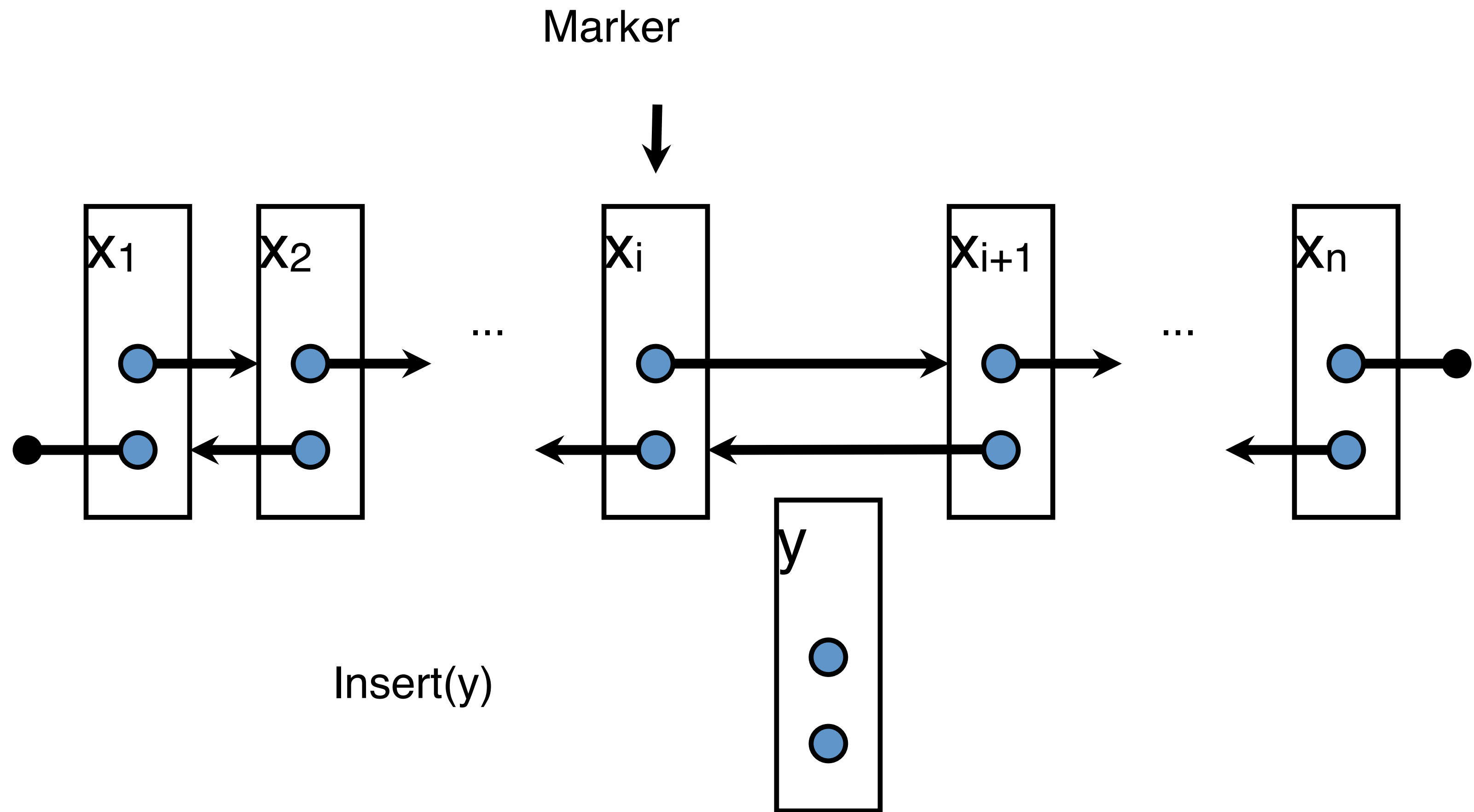
- Delete()

  Marker.prev.next ← Marker.next

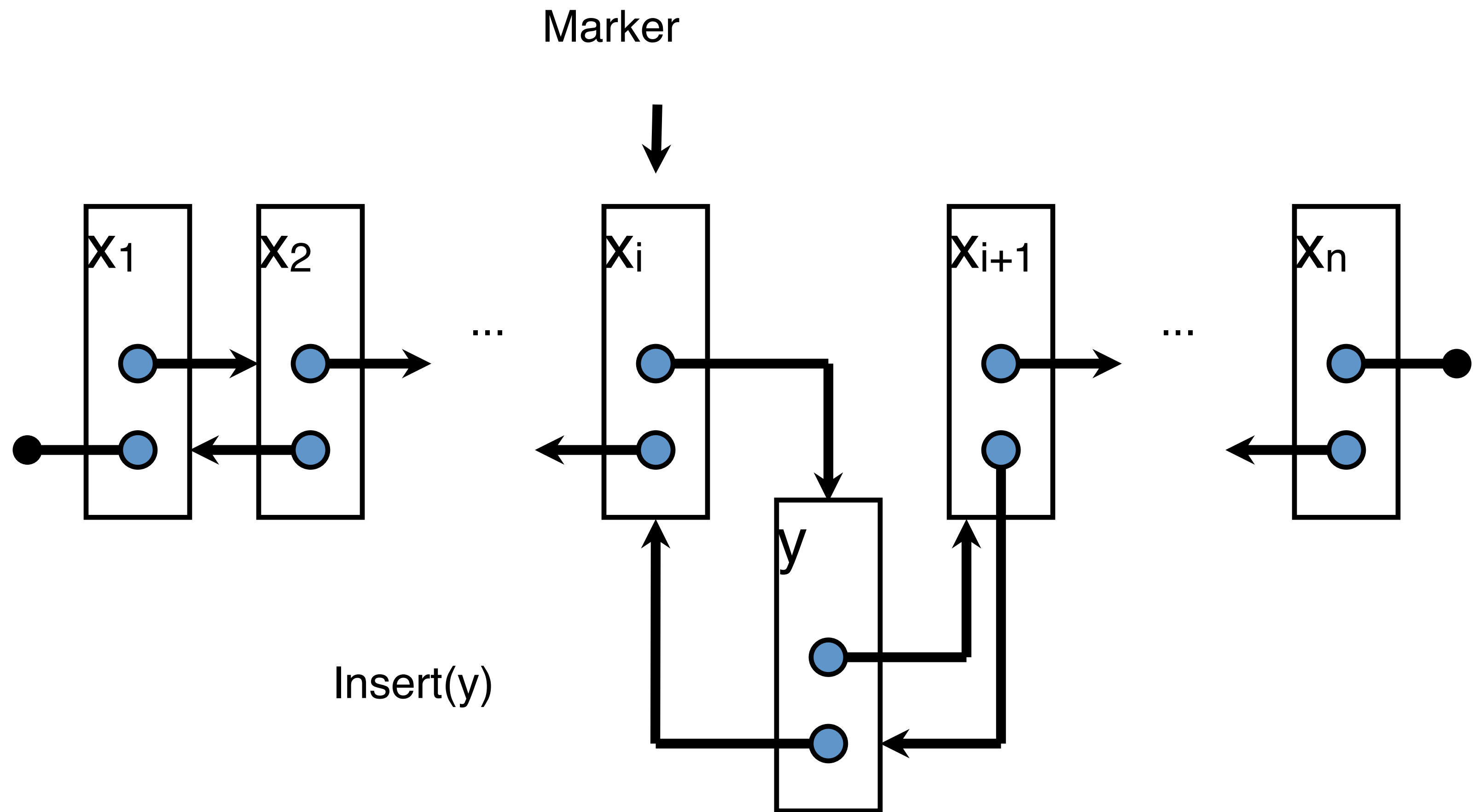
  Marker.next.prev ← Marker.prev

  Marker ← Marker.prev

Doppelt verkettete Listen



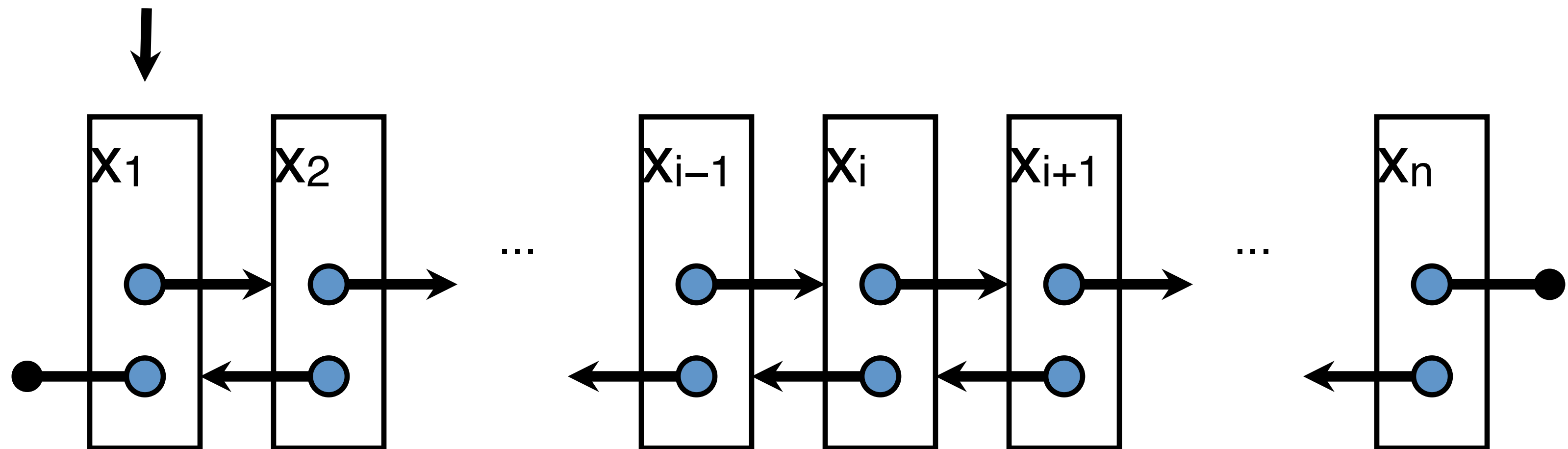
Doppelt verkettete Listen



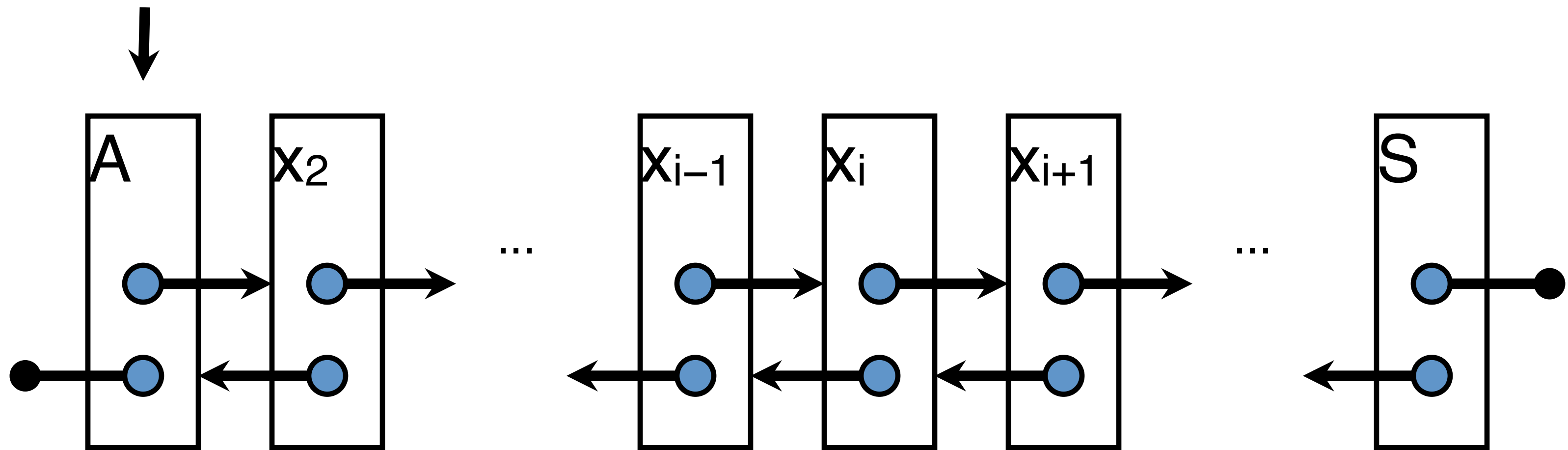
Doppelt verkettete Listen

- Insert(Y)
  Y.prev← Marker
  Y.next← Marker.next
  Y.prev.next← Y
  Y.next.prev← Y

Modifikation an den Enden



Modifikation an den Enden



Anchor/Sentinel (keine Daten)

1.2.2 Warteschlange

- Liste mit eingeschränkter Funktionalität
- Einfügen nur am Ende
- Auslesen/Entfernen nur am Anfang
- „First in, first out“ (FIFO)

Warteschlange

- Wertebereich : $L = \{\ \} \cup W \cup W^2 \cup W^3 \cup \ldots$
- Create : $\rightarrow L$
- Enq : $W \times L \rightarrow L$
- Deq : $L \rightarrow L$
- Get : $L \rightarrow W$
- Empty : $L \rightarrow Bool$

Warteschlange

•	Empty(Create())	= true
	Empty(Enq(x,z))	= false
•	Deq(Enq(x,Create()))	= Create()
	Deq(Enq(x,z))	= Enq(x,Deq(z)) if z ≠ { }
•	Get(Enq(x,Create()))	= x
	Get(Enq(x,z))	= Get(z) if z ≠ { }

Warteschlange

- Satz: Jede Warteschlange hat die Form

  $Enq(x_n,Enq(x_{n-1},\ldots Enq(x_1,Create())))$

- Beweis durch vollständige Induktion über die Anzahl der verwendeten Operationen
  - Create() … n=0
  - Jede andere Operation erhält die Form

Warteschlange

- $Get(Enq(x_n,Enq(x_{n-1}, \ldots Enq(x_1,Create())))))$
- $Enq(x_n,Get(Enq(x_{n-1}, \ldots Enq(x_1,Create())))))$
- $Enq(x_n,Enq(x_{n-1},Get(\ldots Enq(x_1,Create())))))$
- $Enq(x_n,Enq(x_{n-1}, \ldots Get(Enq(x_1,Create())))))$
- x_1

Warteschlange

- Deq(Enq(x_n,Enq(x_{n-1}, … Enq(x_1,Create())))))
- Enq(x_n,Deq(Enq(x_{n-1}, … Enq(x_1,Create())))))
- Enq(x_n,Enq(x_{n-1},Deq(… Enq(x_1,Create())))))
- Enq(x_n,Enq(x_{n-1}, … Deq(Enq(x_1,Create())))))
- Enq(x_n,Enq(x_{n-1}, … Create()))

Array-Implementierung

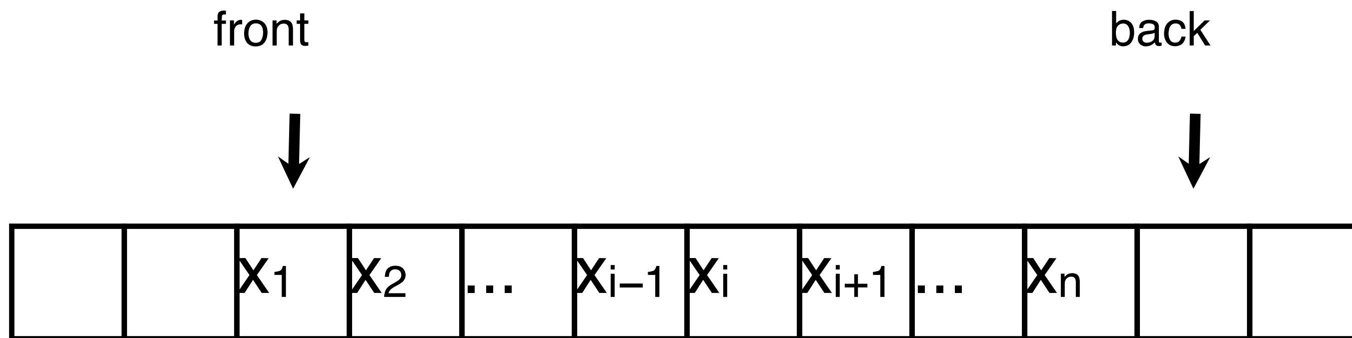
- Da nur am Anfang eingefügt und am Ende gelöscht wird, ist keine garbage collection notwendig.

- Maximale Länge wird als bekannt vorausgesetzt.

Array-Implementierung

- class Schlange {
    Datentyp S[Länge]
    int front, back
  }

Array-Implementierung



Array-Implementierung

- Create()
  - front ← 0
  - back ← 0
- Empty()
  - return (front = back)
- Get()
  - if front ≠ back then
    - return S[front]
  - else
    - Q-EMPTY-ERROR

Array-Implementierung

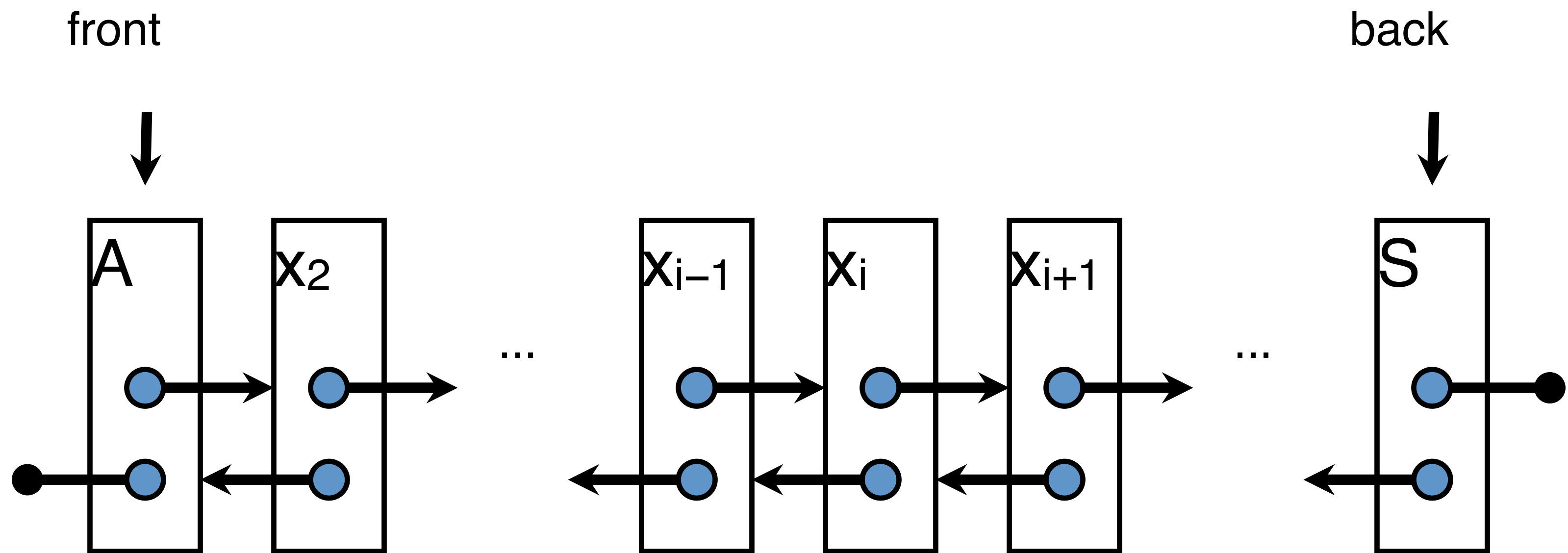
- Deq()

```
if front ≠ back then
    front ← (front + 1) % Länge
else
    Q-EMPTY-ERROR
```

- Enq(x)

```
S[back] ← x
back ← (back + 1) % Länge
if front = back then
    Q-FULL-ERROR
```

Pointer-Implementierung



Interner Marker entfällt!

1.2.3 Stack

- Liste mit eingeschränkter Funktionalität
- Einfügen nur am Anfang
- Auslesen/Entfernen nur am Anfang
- „Last in, first out" (LIFO)

Stack

- Wertebereich : $L = \{\,\} \cup W \cup W^2 \cup W^3 \cup \ldots$
- Create: $\rightarrow L$
- Push : $W \times L \rightarrow L$
- Pop : $L \rightarrow L$
- Top : $L \rightarrow W$
- Empty : $L \rightarrow \text{Bool}$

Stack

- Empty(Create()) = true
  Empty(Push(x,z)) = false
- Pop(Push(x,z)) = z
- Top(Push(x,z)) = x

Stack

- Satz: Jeder Stack hat die Form

  $Push(x_1,Push(x_2, \dots Push(x_n,Create())))$

- Beweis durch vollständige Induktion über die Anzahl der verwendeten Operationen
  - Create() ... n=0
  - Jede andere Operation erhält die Form

Array-Implementierung

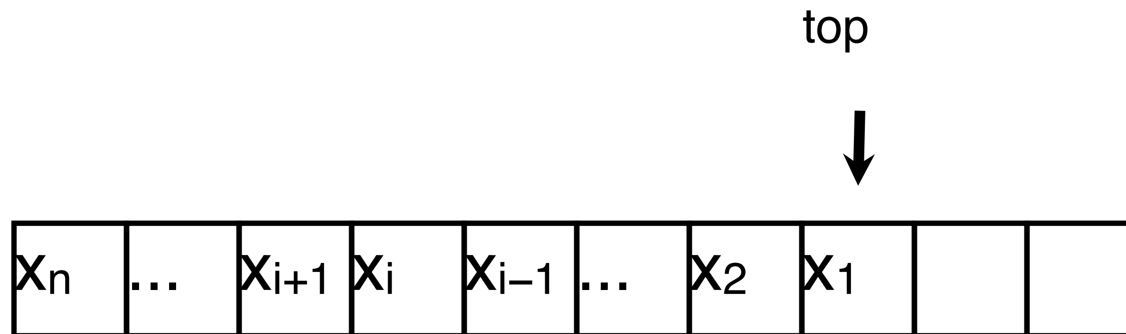
- Da nur am Anfang eingefügt und gelöscht wird, ist keine garbage collection notwendig.

- Maximale Größe wird als bekannt vorausgesetzt.

Array-Implementierung

- class Stack {
    Datentyp S[Größe]
    int top
  }

Array-Implementierung



Array-Implementierung

- Create()

```
top ← −1
```

- Empty()

```
return (top = −1)
```

- Top()

```
if top ≠ −1 then
    return S[top]
else
    STACK-EMPTY-ERROR
```

Array-Implementierung

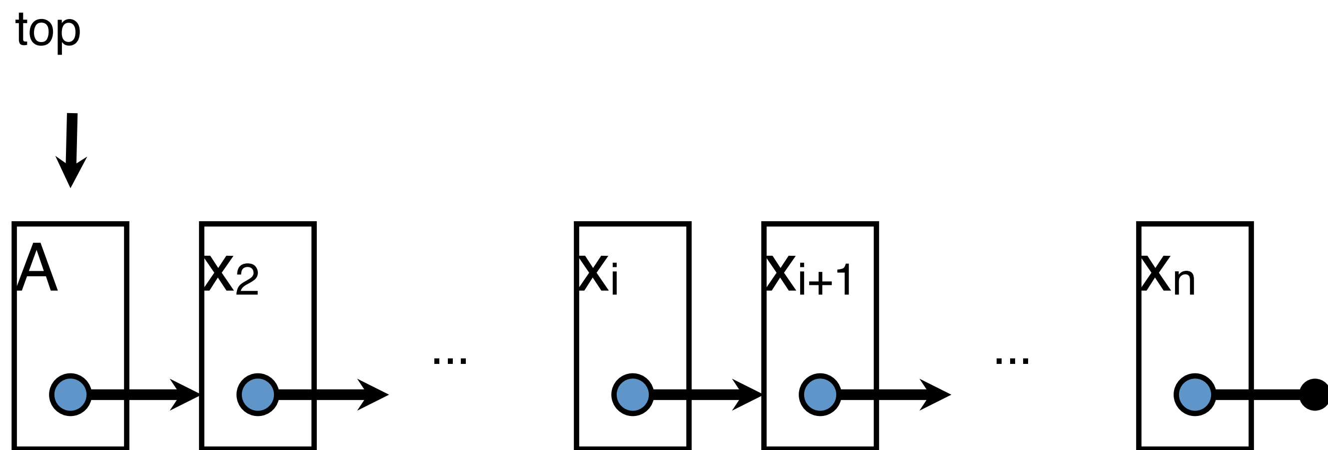
- Push(x)

```
top ← top+1
if top < Größe then
    S[top] ← x
else
    STACK-FULL-ERROR
```

- Pop()

```
if top ≠ −1 then
    top ← top−1
else
STACK-EMPTY-ERROR
```

Pointer-Implementierung



Suche im Labyrinth

- Geg:
  - Spielfläche $W = [0..n] \times [0..m]$
  - mögliche Positionen $P = (x,y) \in W$
  - Bewegungsrichtungen: $R = \{N,S,W,O\}$
  - Labyrinth ... $L : W \times R \rightarrow \text{Bool}$
  - Startposition P_{begin}
  - Zielposition P_{end}
- Ges: Pfad $S \in R^k$ von P_{begin} nach P_{end}

Suche im Labyrinth

- Funktion $Go: W \times R \rightarrow W$ beschreibt einen Schritt $Go(P,r)=Q$ von P in die Richtung r nach Q.

- Wenn $L(P_{begin},r) = true$ und $Go(P_{begin},r) = Q$ und es existiert ein Pfad $r_1,..,r_n$ von Q nach P_{end} dann ist $r,r_1,..,r_n$ ein Pfad von P_{begin} nach P_{end}.

Suche im Labyrinth

- Annahme: Labyrinth hat keine Zyklen
- Pfad = Liste
- Erweiterung nur am Anfang der Liste
  → Pfad = Stack
- Rekursive Formulierung

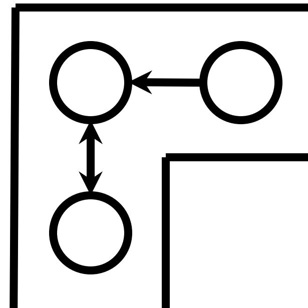
Rekursiver Algorithmus

- Bool Suche(P,S,Q)

```
if P = Q then
    S ← Create()
    return true
else
    for r ∈ {N,S,W,O} do
        if L(P,r) and Suche(Go(P,r),S,Q) then
            S ← Push(r,S)
            return true
    return false
```

Rekursiver Algorithmus

- Problem: unendliche Suche
- Lösung: verbiete Schritte zurück.
- „Negative" Richtung:
  −N = S, −S = N, −W = O, −O = W



Rekursiver Algorithmus

- Bool Suche(P,S,Q,r')

```
if P = Q then
    S ← Create()
    return true
else
    for r ∈ {N,S,W,O} \ r' do
        if L(P,r) & Suche(Go(P,r),S,Q,−r) then
            S ← Push(r,S)
            return true
    return false
```

Iterativer Algorithmus

- Rekursion steuert die Suche
- Direkte Lösung unter Verwendung eines Stack
- Ordnung der Richtungen N < O < S < W

  (next(N) = O, next(O) = S, ...)

- Sortiere Pfade lexikographisch

Lexikographische Reihenfolge

- N
- NN
- NNN
- ...
- NNNNO
- NNNNON
- ...
- NNNNOO
- ...
- NNNNOS
- ..
- NNNNOW
- ...
- NNNNS
- ...
- NNNO
- ...
- WW

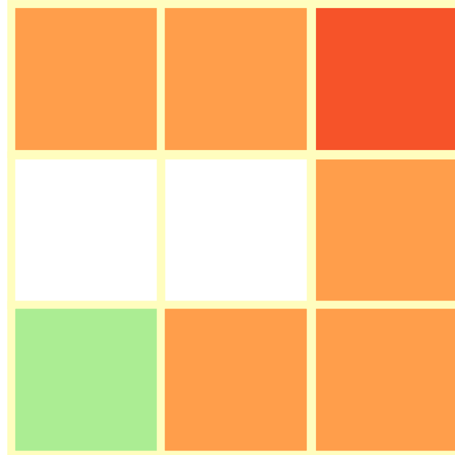
Iterativer Algorithmus

```
•   P ← P_begin
    S ← Create()
    r ← ´N´
    while P ≠ P_end and (L(P,r) or r < ´W´ or  not Empty(S)) do
      while P ≠ P_end and L(P,r) do
          S = Push(r,S)
          P ← Go(P,r)
          r ← ´N´
      if P ≠ P_end then
          while r = ´W´ and not Empty(S) do
              r = Top(S)
              P ← Go(P,−r)
              S ← Pop(S)
          if r < ´W´ then
              r ← next(r)
```

Iterativer Algorithmus

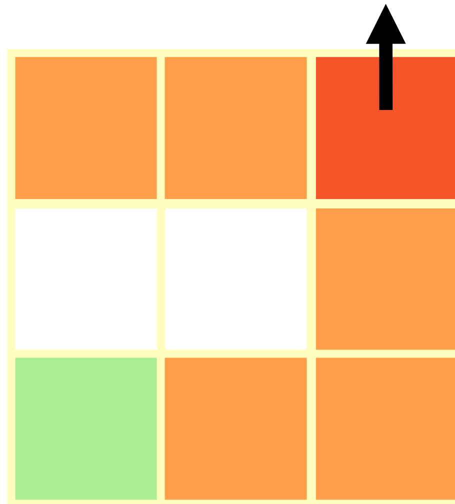
- $P \leftarrow P_{begin}$
  $S \leftarrow$ Create()
  $r \leftarrow$ ´N´
  while $P \neq P_{end}$ and (L(P,r) or $r <$ ´W´ or not Empty(S)) do
    while $P \neq P_{end}$ and L(P,r) and $r \neq -$Top(S) do
      S = Push(r,S)
      $P \leftarrow$ Go(P,r)
      $r \leftarrow$ ´N´
    if $P \neq P_{end}$ then
      while r = ´W´ and not Empty(S) do
        r = Top(S)
        $P \leftarrow$ Go(P,−r)
        $S \leftarrow$ Pop(S)
        if $r <$ ´W´ then
          $r \leftarrow$ next(r)

3 x 3 grid ...



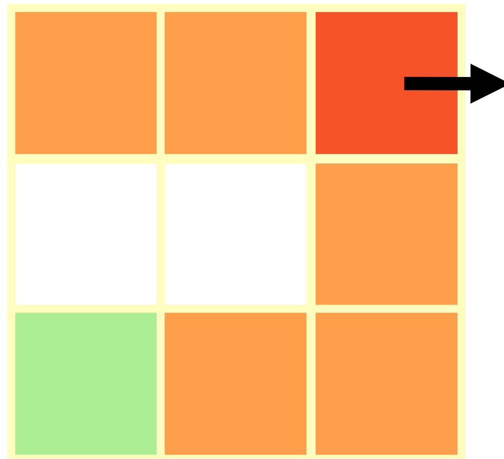
N
O
S
SO
SS
SSO
SSS
SSW
SSWN
SSWS
SSWW

3 x 3 grid ...



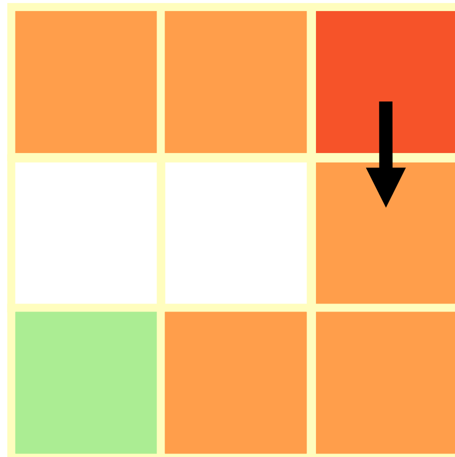
N
O
S
SO
SS
SSO
SSS
SSW
SSWN
SSWS
SSWW

3 x 3 grid ...



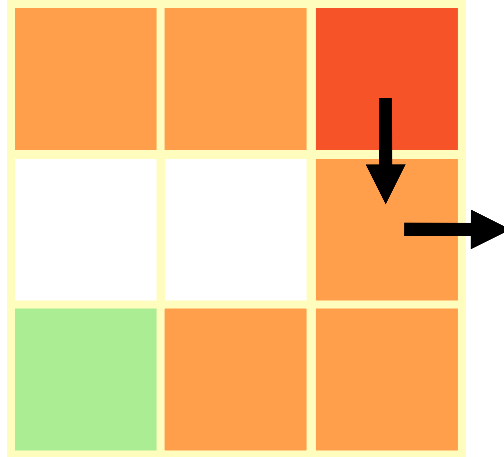
N
O
S
SO
SS
SSO
SSS
SSW
SSWN
SSWS
SSWW

3 x 3 grid ...



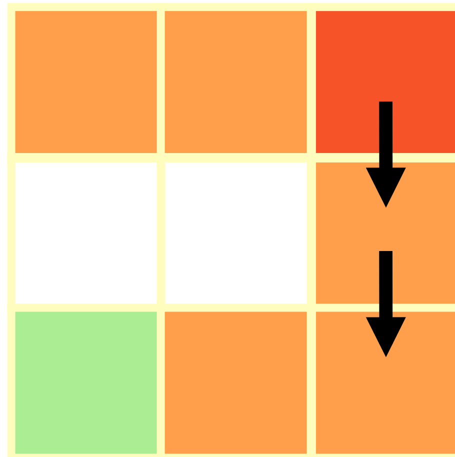
N
O
S
SO
SS
SSO
SSS
SSW
SSWN
SSWS
SSWW

3 x 3 grid ...



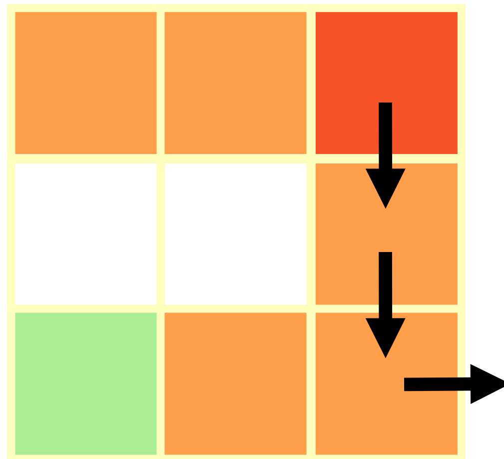
N
O
S
SO
SS
SSO
SSS
SSW
SSWN
SSWS
SSWW

3 x 3 grid ...



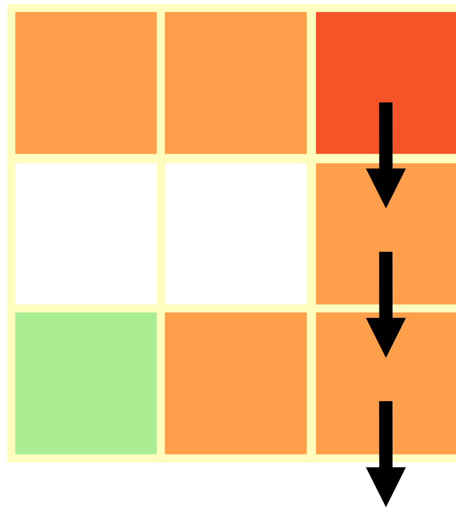
N
O
S
SO
SS
SSO
SSS
SSW
SSWN
SSWS
SSWW

3 x 3 grid ...



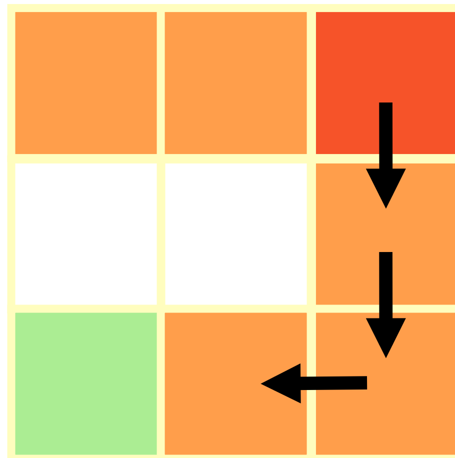
N
O
S
SO
SS
SSO
SSS
SSW
SSWN
SSWS
SSWW

3 x 3 grid ...



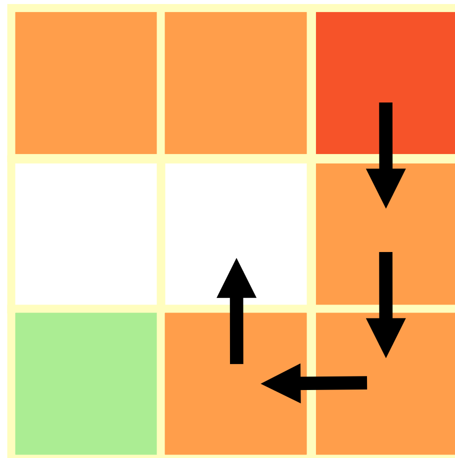
N
O
S
SO
SS
SSO
SSS
SSW
SSWN
SSWS
SSWW

3 x 3 grid ...



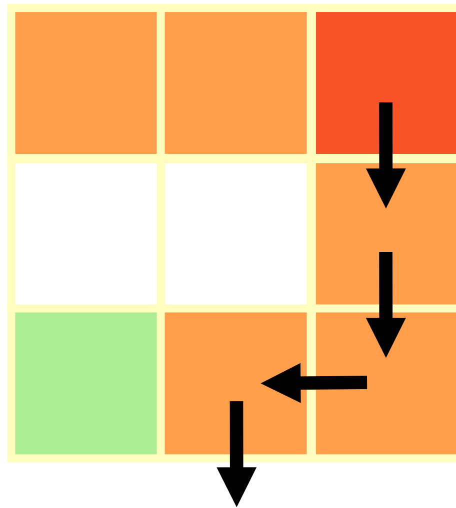
N
O
S
SO
SS
SSO
SSS
SSW
SSWN
SSWS
SSWW

3 x 3 grid ...



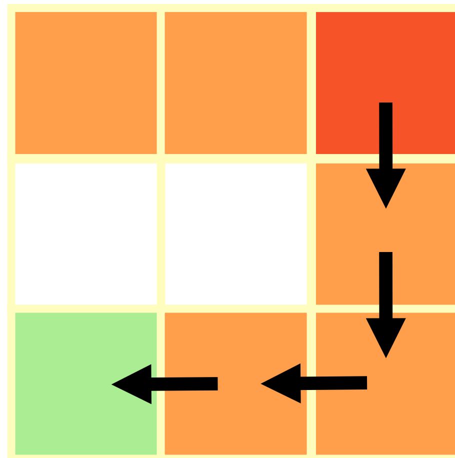
N
O
S
SO
SS
SSO
SSS
SSW
SSWN
SSWS
SSWW

3 x 3 grid ...



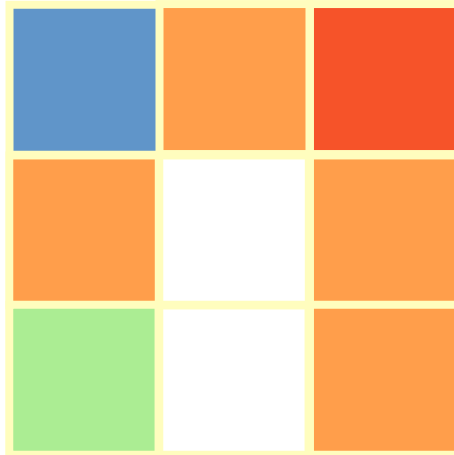
N
O
S
SO
SS
SSO
SSS
SSW
SSWN
SSWS
SSWW

3 x 3 grid ...



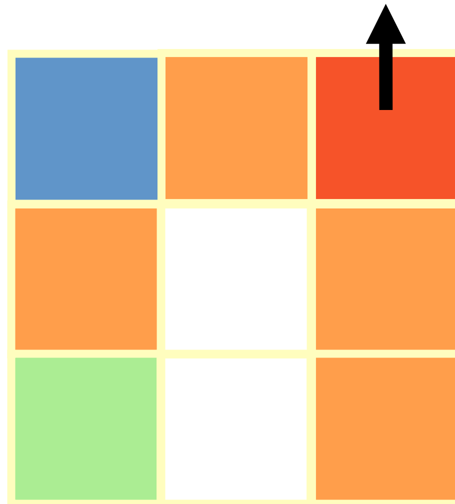
N
O
S
SO
SS
SSO
SSS
SSW
SSWN
SSWS
SSWW

3 x 3 grid ...



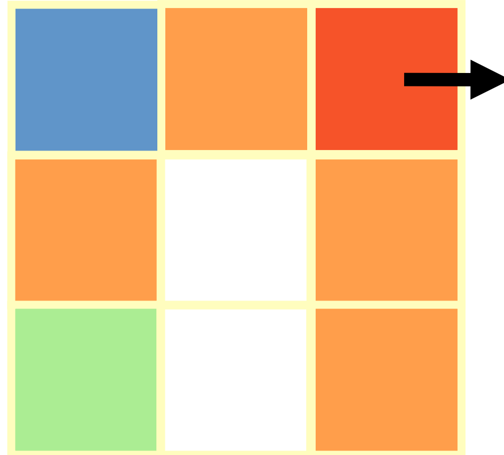
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



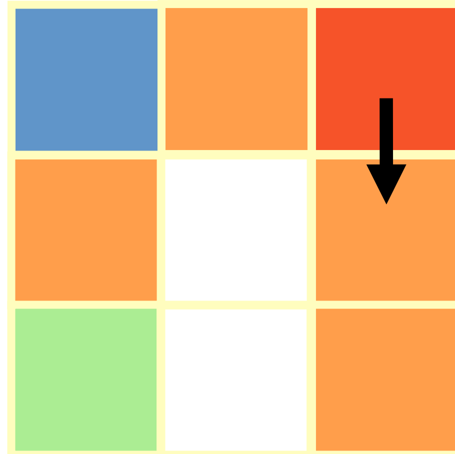
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



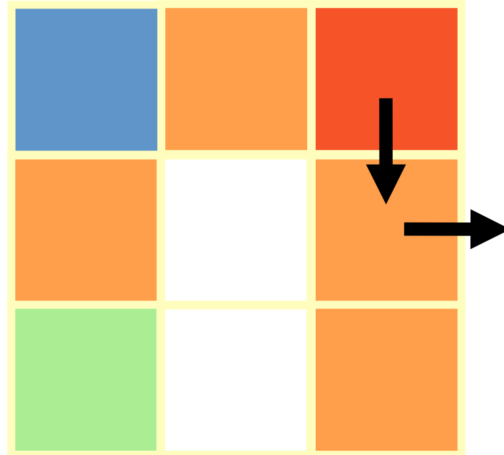
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



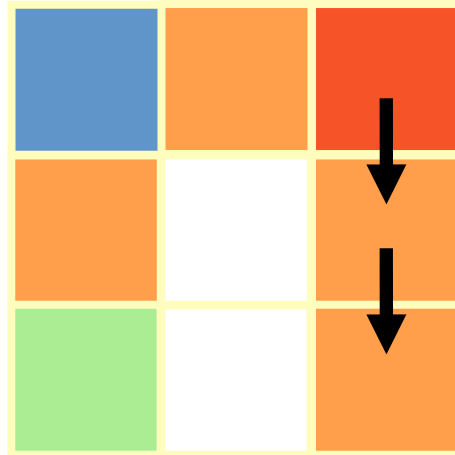
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



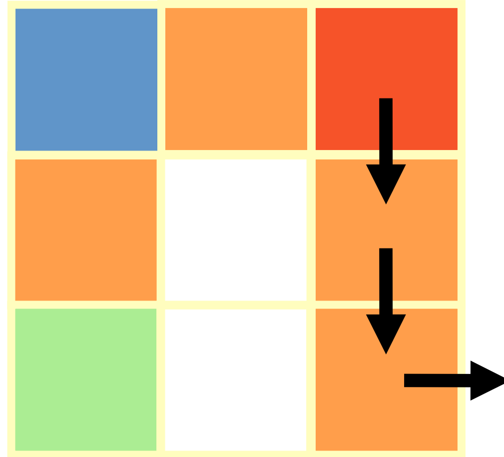
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



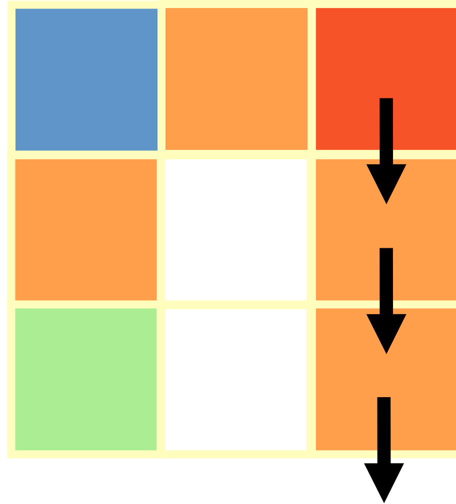
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



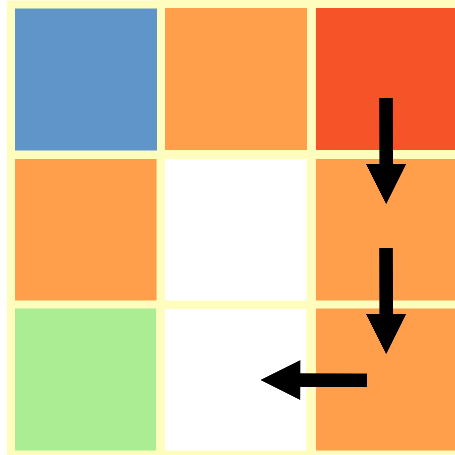
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



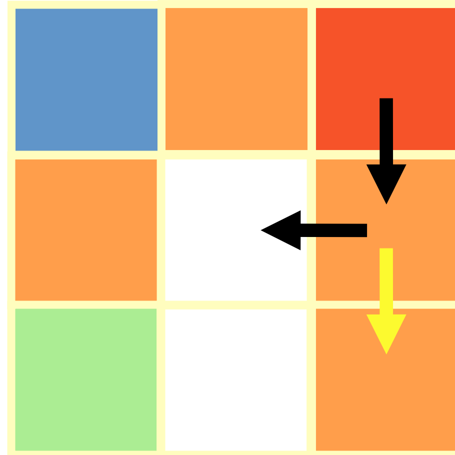
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



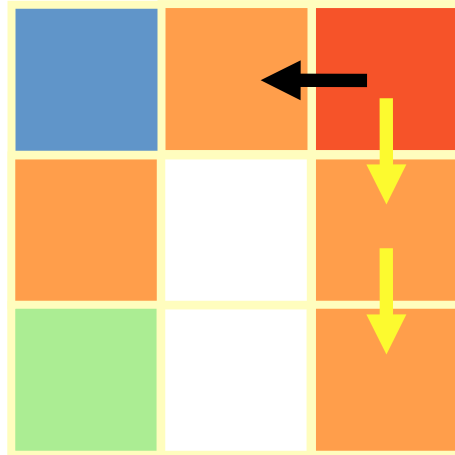
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



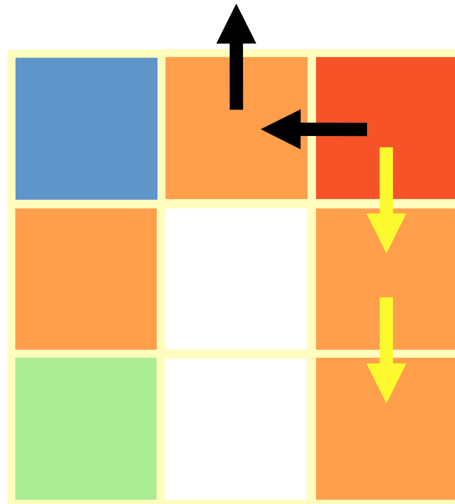
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



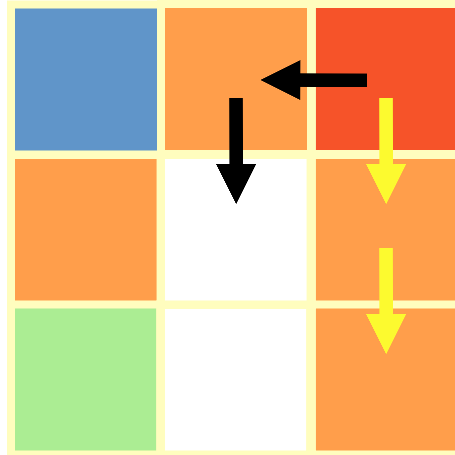
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



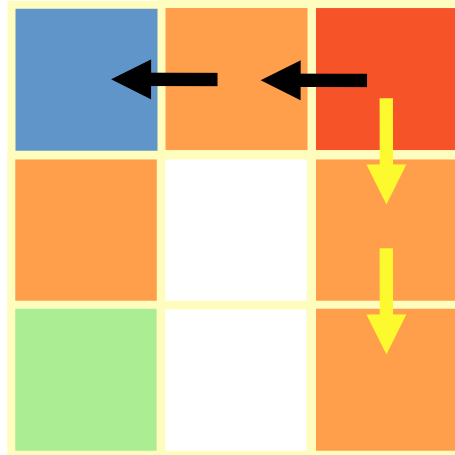
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



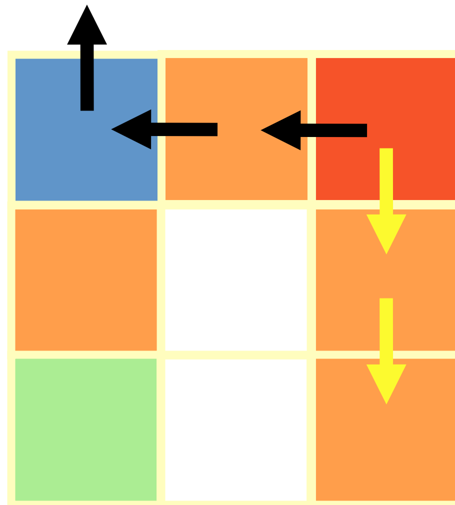
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



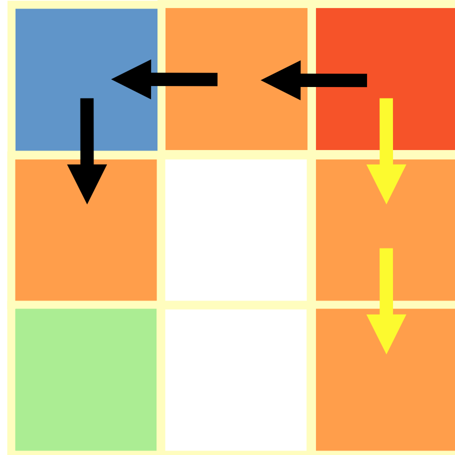
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



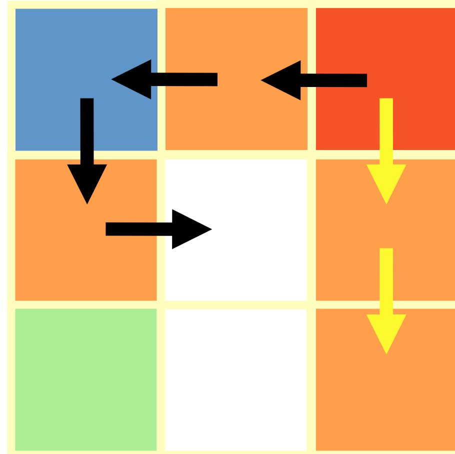
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



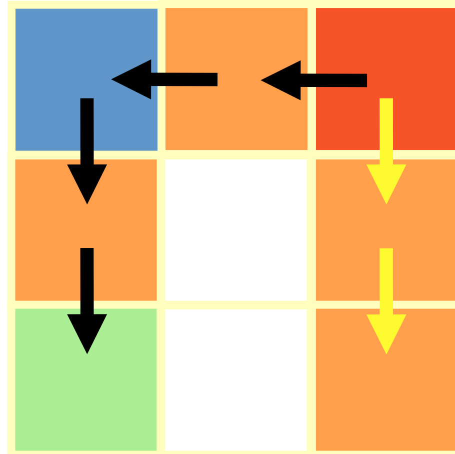
N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

3 x 3 grid ...



N
O
S
SO
SS
SSO
SSS
SSW
SW
W
WN
WS
WW
WWN
WWS
WWSO
WWSS

Stack

- Merke: Rekursive Algorithmen lassen sich in der Regel mit einer Stack-Datenstruktur auch iterativ formulieren.
- Das LIFO-Prinzip entspricht der Abarbeitungsreihenfolge der geschachtelten Prozeduren (was zuletzt aufgerufen wird, wird als erstes bearbeitet).
- Beispiel: systematische Suche in einem Labyrinth (bei Sackgassen zurückgehen).