Container, verkettete Listen, Stapel, Warteschlange, sortierte Liste, Fallstudie Point-In-Polygon Algorithmus (Jordankurven), Bresenham Algorithmus, Polygonfüllalgorithmus

5. DYNAMISCHE DATENSTRUKTUREN

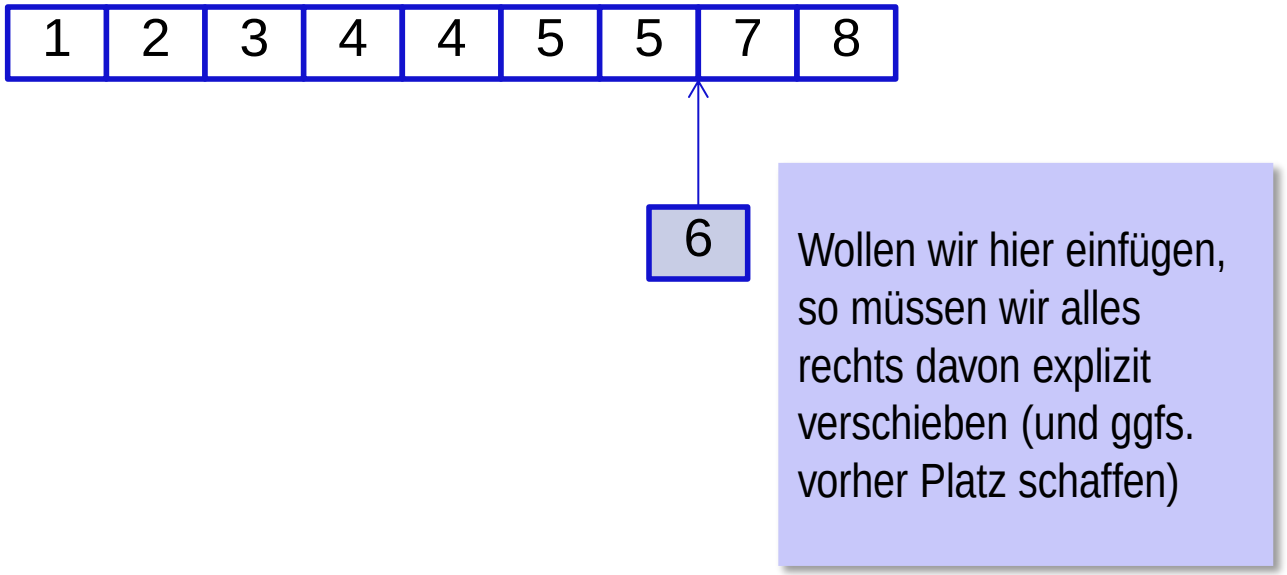
Container für eine Folge gleichartiger Daten

- Bisher: Arrays
- Zusammenhängender Speicherbereich, wahlfreier Zugriff (auf i-tes Element)

- Einmal alloziert ist die Länge fixiert

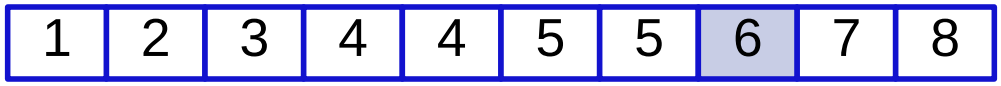
Probleme mit Arrays

- Einfügen oder Löschen von Elementen "in der Mitte" ist aufwändig



Probleme mit Arrays

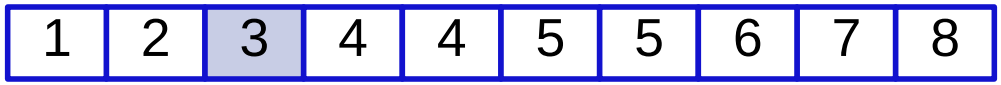
- Einfügen oder Löschen von Elementen "in der Mitte" ist aufwändig



Wollen wir hier einfügen, so müssen wir alles rechts davon explizit verschieben (und ggfs. vorher Platz schaffen)

Probleme mit Arrays

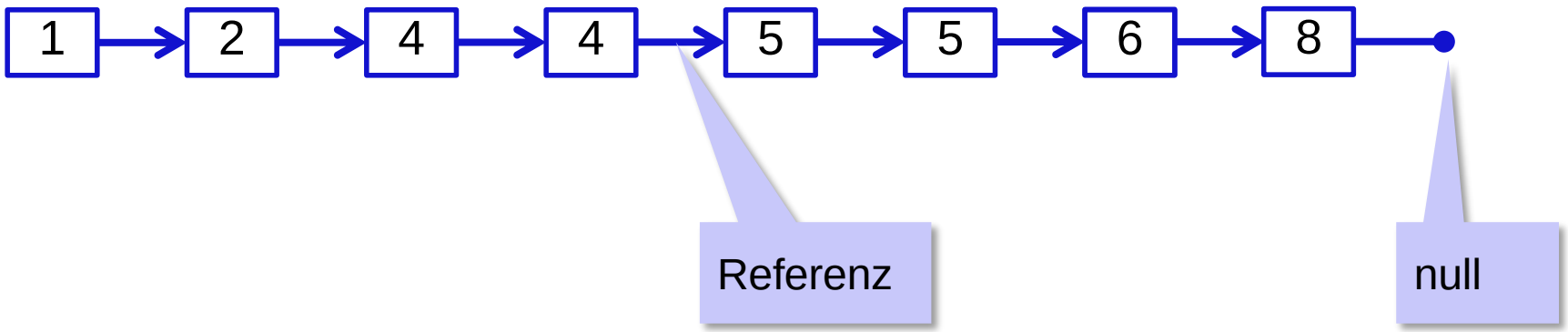
- Einfügen oder Löschen von Elementen "in der Mitte" ist aufwändig



Wollen wir hier löschen, müssen wir alles rechts davon explizit verschieben

Lösung: (Verkettete) Listen

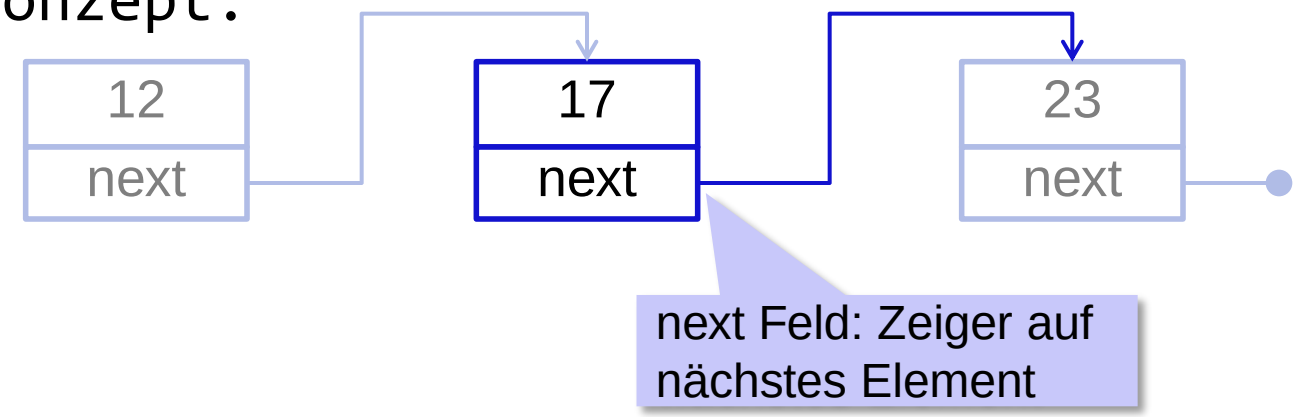
- Container für eine Folge von Daten des gleichen Typs
- Kein zusammenhängender Speicherbereich, kein wahlfreier Zugriff



(Einfach) verkettete Liste

```
class Node
{
   double value; // "Nutzlast" des Knoten
   Node next;    // Verkettung: Zeiger
                 // auf nächsten Knoten
   Node (double v, Node nxt) // Konstruktor
   {
    value=v; next = nxt;
   }
}
```

Konzept:



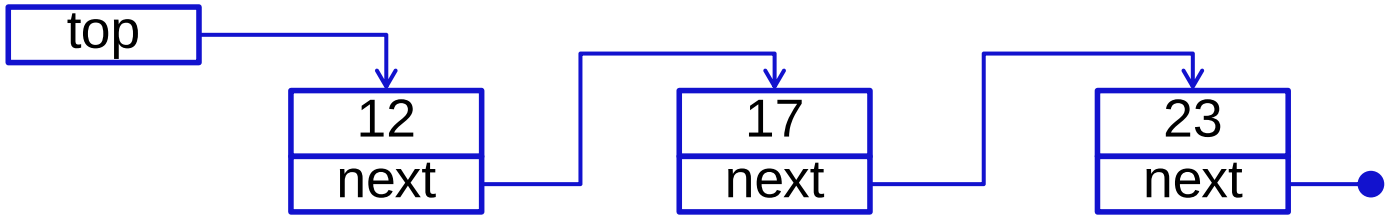
Stapel

Der Stapel ist das einfachste Beispiel einer Datenstruktur, welche man gut mit einer verketteten Liste implementieren kann.
Funktionalität:

- Knoten vorne einfügen: Push
- Knoten vorne herausnehmen: Pop

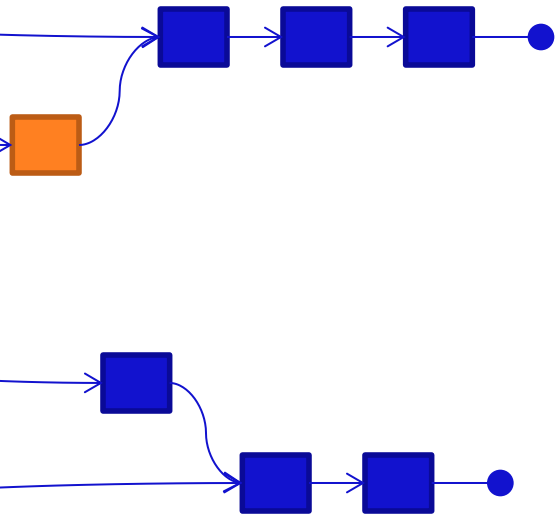
```
LIFO :
Last In
First Out
```

Erstes Element durch "top" repräsentiert



Stapel

```
public class Stack {

    private class Node {...} // wie oben implementiert

    Node top = null; // oberstes Element auf dem Stapel, initial 0

    public void Push(double val) {
         Node next = top;
         top = new Node(val, next);
    }

     public double Pop() {
         assert(top != null);
         double value = top.value;
         top = top.next;
         return value;
     }

}
```



Warteschlange (FIFO)

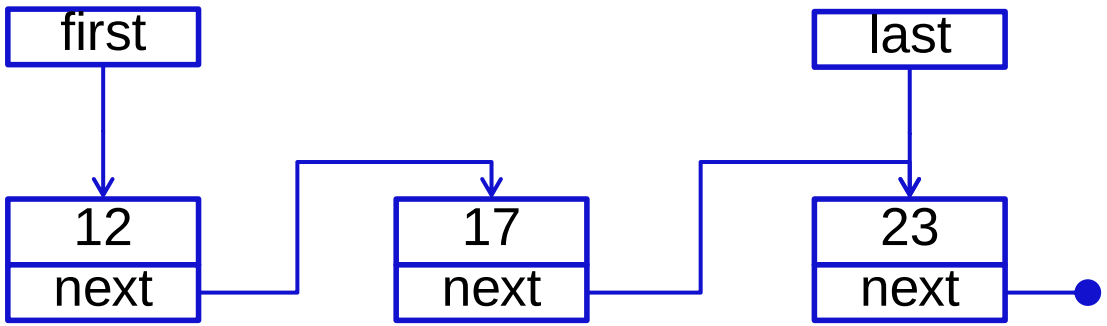
Wichtigste Operationen

- Knoten hinten einfügen: Put
- Knoten vorne herausnehmen: Get

FIFO:
First In
First Out

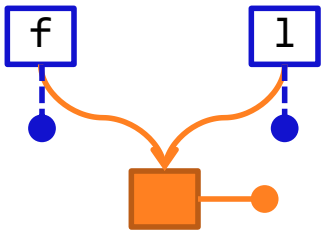
```
public class Queue {

    private class Node {...} // wie oben implementiert

    Node first= null; // erstes Element der Warteschlange
    Node last = null; // letztes Element der Warteschlange
```

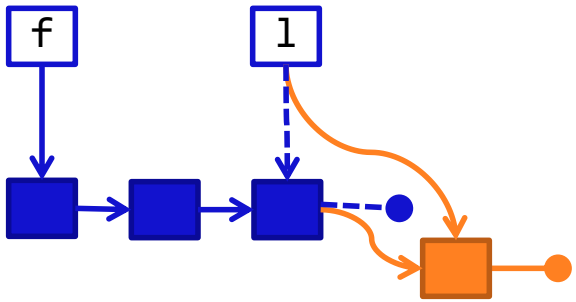
Invarianten:
Entweder
first = last = null oder
first != null, last != null



Warteschlange: Einfügen

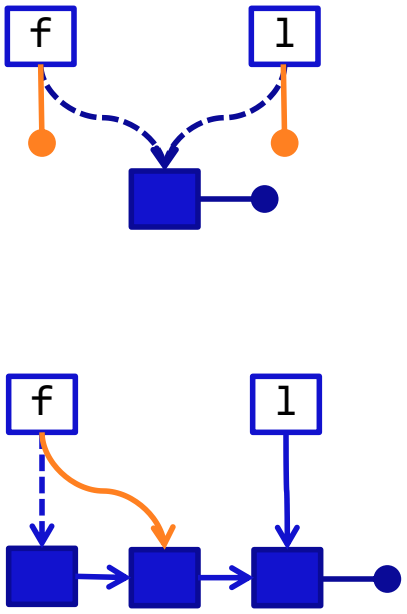
```
public class Queue {
...

public void Put(double val){
   if (last == null)
   {
      last = new Node(val, null);
      first = last;
   }
   else
   {
      last.next = new Node(val, null);
      last = last.next;
   }
}
...
```





Warteschlange: Herausnehmen

```
public class Queue {
...

public double Get(){
   assert(first != null);

   double value = first.value;
   if (first == last)
      last = null;
   first = first.next;
   return value;
}
...
```



Kosten

Für Warteschlange und Stapel

- Wenn Allokation hinreichend billig, dann
  - Einfügen: O(1)
  - Herausnehmen: O(1)
- Speicherkosten: Ein Listenelement pro Wert

Soweit war das auch einfach. Schwieriger ist Einfügen / Löschen in der Mitte.

Sortierte Liste

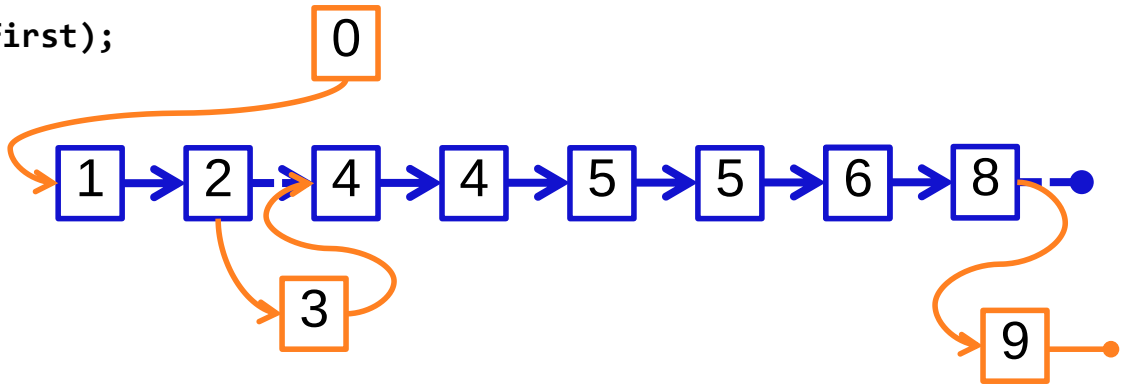
Wichtigste Operationen

- Wert sortiert einfügen: PutSorted
- Wert herausnehmen
- Iterieren

```
public class SortedList {

    private class Node {...} // wie oben implementiert

    Node first= null; // erstes Element der Liste
```

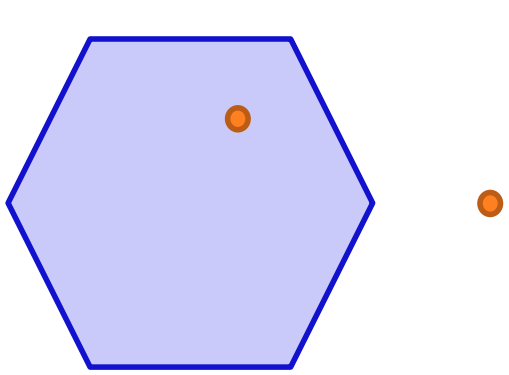
Sortierte Liste

```
public class SortedList {

public void InsertSorted(double val)
{
    if (first == null || val < first.value)
    {
        Node node = new Node(val, first);
        first = node;
    }
    else
    {
        Node prev = first;
        Node current = prev.next;
        while(current != null && current.value < val)
        {
            prev = current;
            current = current.next;
        }
        prev.next = new Node(val,current);
    }
}
```

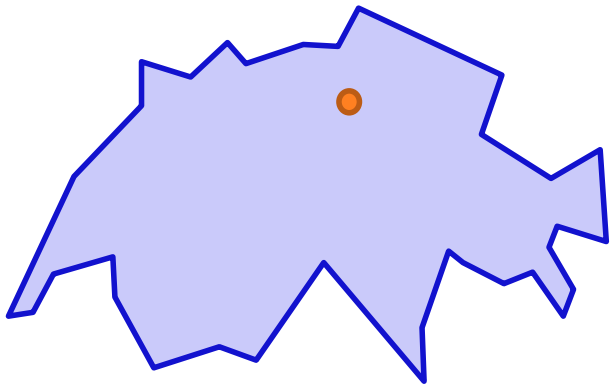


Fallstudie: Point-In-Polygon Algorithmus

- Annahme: wir haben ein abgegrenztes Gebiet auf einer Landkarte.
  - Wie stellen wir das Gebiet dar?
  - Wie entscheiden wir effizient ob ein Punkt «innen» liegt?
  - Wie füllen wir das Gebiet mit einer Farbe?



Ein konvexes Gebiet



Ein nicht konvexes Gebiet

Hintergrund: Jordankurven

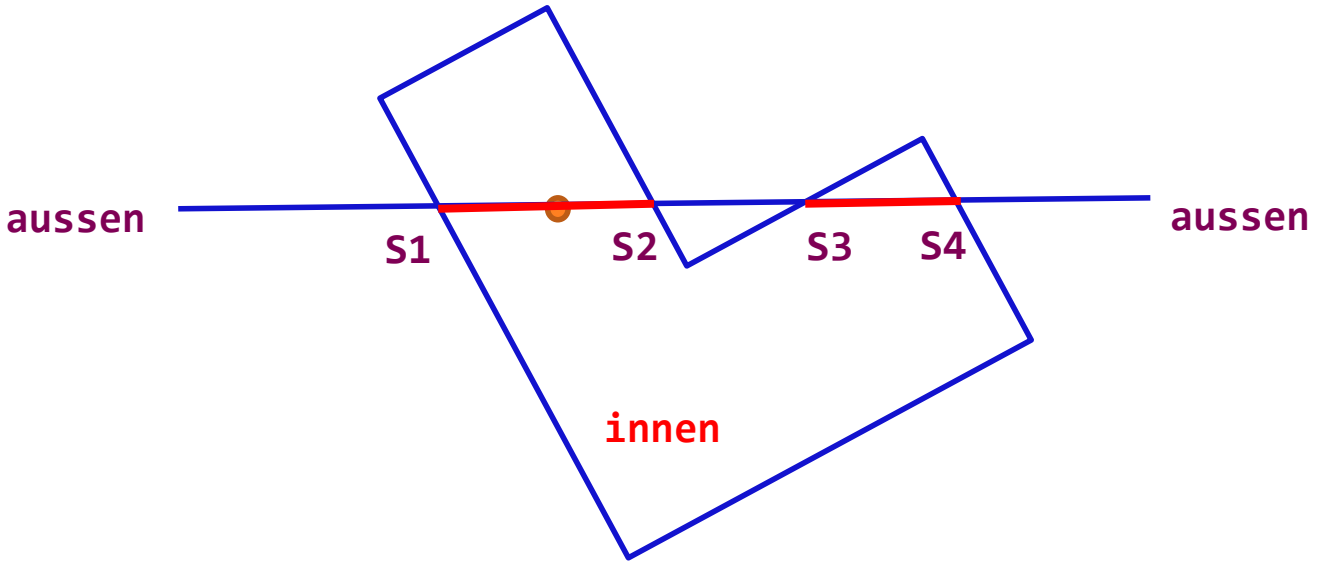
- Schnittfreie Polygone sind *Jordankurven* und zerlegen die Ebene in zwei Gebiete: das Innere und das Äussere
  - Das Innere ist beschränkt und
  - Das Äussere ist unbeschränkt.

Der allgemeine strikt mathematische Beweis (algebraische Topologie, Abbildung der Einheitskugel) dieser Aussage ist relativ aufwändig und soll uns nicht weiter aufhalten.

- Das kann man ausnutzen, indem man das Polygon mit Geraden schneidet. Man weiss dann, dass der unbeschränkte Teil der Geraden aussen liegt. Daher funktioniert die folgende Abzählmethode.

Abzählmethode

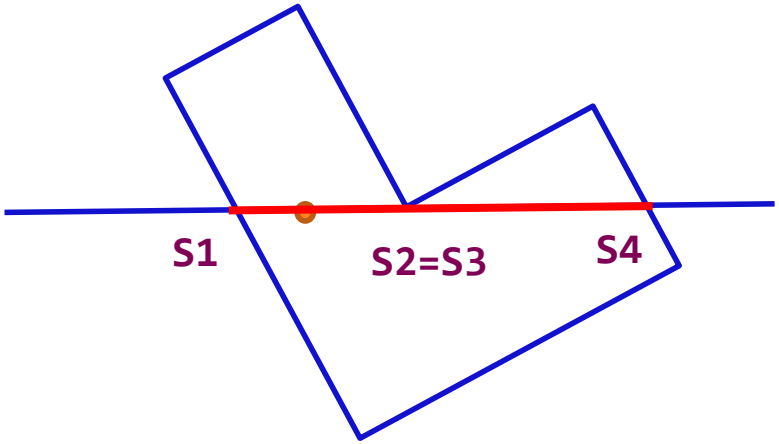
- Zähle auf einer beliebigen Geraden (die den fraglichen Punkt enthält), von unendlich fern kommend, die Anzahl *echte* Schnittpunkte mit dem Polygon
- Alle Bereiche der Gerade zwischen ungeradzahligen und geradzahligen Schnittpunkten liegen innen

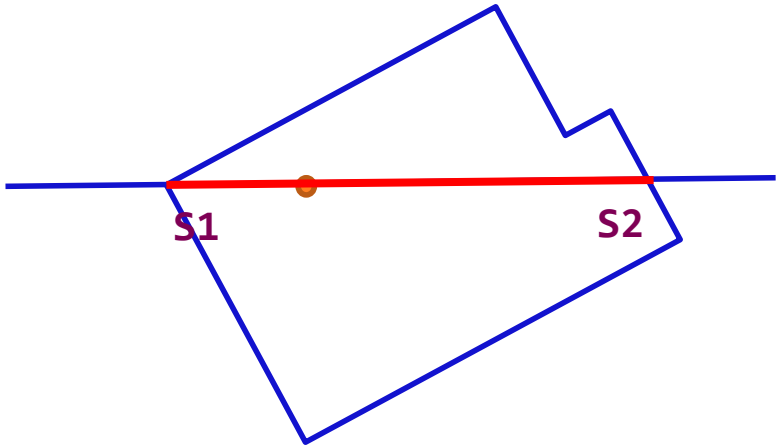


- Das funktioniert mit jeder Geraden, wir suchen uns die einfachen Fälle raus (also horizontal oder vertikal)

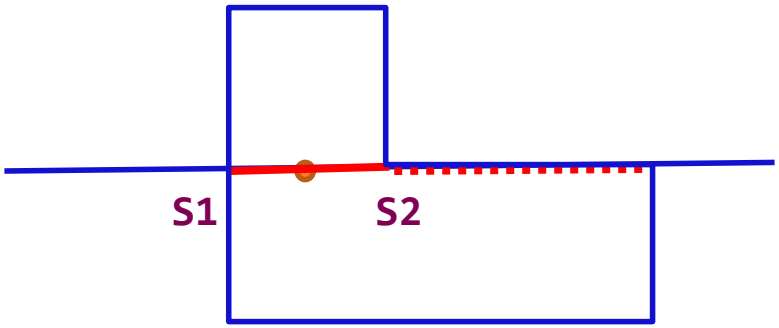
Spezialfälle

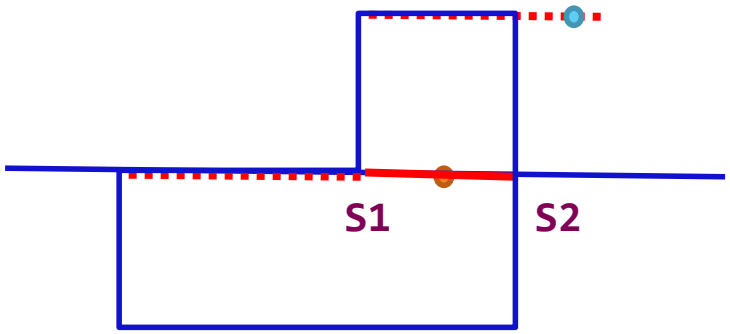
- Schnittpunkt = Eckpunkt





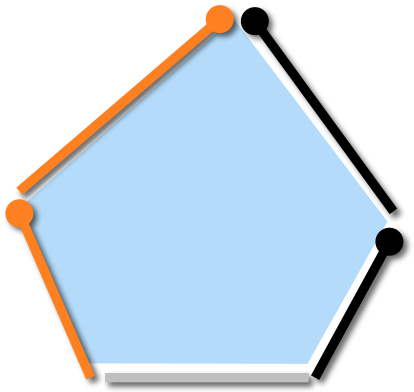
- «Schnittpunkt» = Strecke

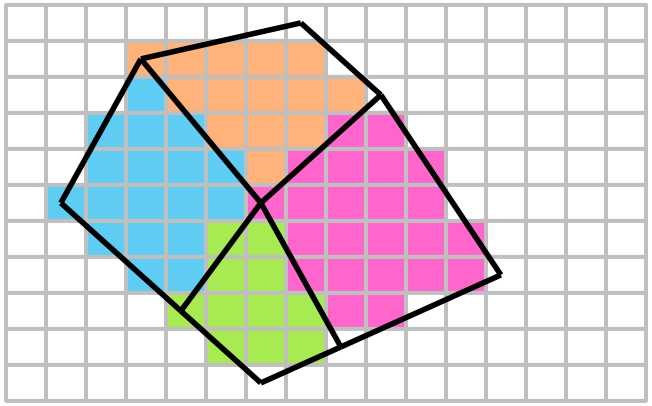




Implementationstrick

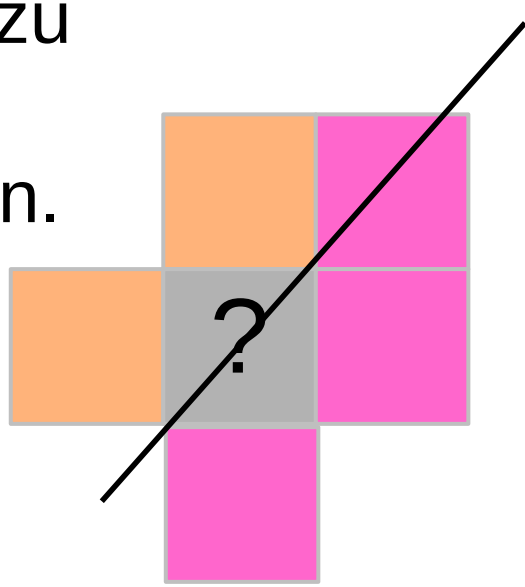
- Behandle Eckpunkte einer begrenzenden Kante richtungsabhängig
  - Es werden z.B. nur die oberen Eckpunkte einer Kante mit vertikaler Komponente berücksichtigt
  - horizontale Kanten werden ignoriert
  - Für eindeutige Segmentierung bei Kachelung: Entscheidung für die Hinzunahme der einen oder anderen Richtung oben/unten, rechts/links (optional: behandle Kanten separat)





Weitere Tricks

- Interessiert man sich für Punkte auf einem Gitter (z.B. in der Pixelgrafik), so ist es günstig alle Algorithmen weitgehend ganzzahlig zu formulieren und den Gebrauch der Fliesskommaarithmetik zu minimieren.
  - Effizienz
  - Genauigkeit



- Beispiel: Bresenham-Algorithmus für das Linienzeichnen (kommt gleich…).

Implementation

Polygon

- Kanten: Folge von Eckpunkten,
  z.B. implementiert als verkettete Liste

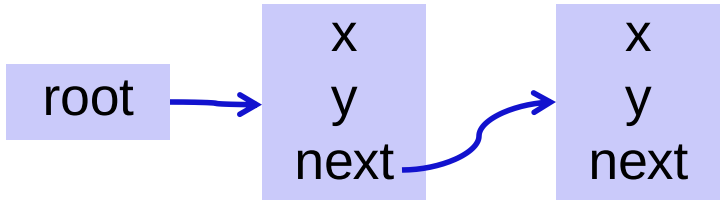
Benötigte Methoden auf dem Polygon

- Hinzufügen von Eckpunkten
- Abzählen von Schnittpunkten des Polygons mit einer Geraden, dargestellt durch Punkt und Richtung (für die Entscheidung ob Punkt innen liegt)
- Aufzählen von inneren Punkten (für das Füllen)

Implementation

Polygon dargestellt als verkettete Liste von Punkten

```
public class Polygon {

    private class Vertex
    {
        int x,y;  // Koordinate des Eckpunkts
        Vertex next;       // Verkettung

        Vertex(int px, int py, Vertex nxt)
        {
            x=px;
            y=py;
            next =nxt;
        }
    }

    Vertex root;

    public void AddPoint(int x, int y)
    {
        root = new Vertex(x,y,root); // Stapel von Punkten
    }

    public boolean PointInPolygon(int x, int y){...}

}
```



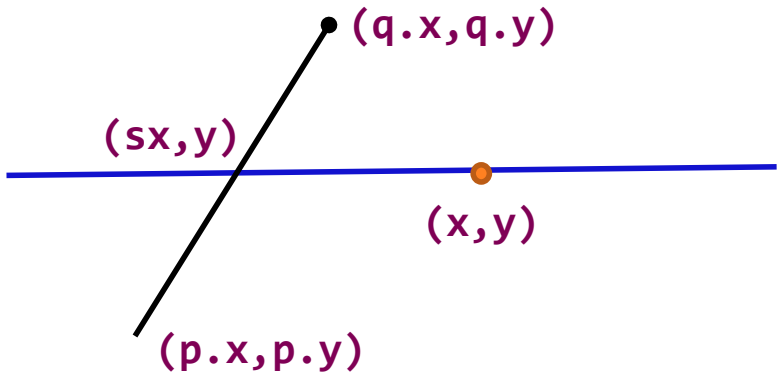
PointInPolygon

```
public boolean PointInPolygon(int x, int y)
{
    if (root==null) return false;
    Vertex p = root;
    Vertex q = p.next;
    boolean inside = false;

    while (q != null) // Traversiere alle Kanten
    {
        if (y <= p.y && y > q.y || y > p.y && y <= q.y) // Schnittpunkt mit Kante
        {
            double sx = p.x + (q.x - p.x) * (double)(y-p.y) / (q.y-p.y);

            if (sx <= x) inside = !inside; // Schnittpunkt links von x
        }

        p = q;   // nächste Kante
        q = p.next;
    }
    return inside;
}
```

Horizontale durch (x,y) schneidet Kante p — q

Ausgenommen: unterer Punkt der Kante p — q

Impliziert: p.y != q.y



Test

```
public class TestPoly {

public static void main(String[] args) {
    Polygon p=new Polygon();

    p.AddPoint(0,0);
    p.AddPoint(0,15);
    ...
    p.AddPoint(10,5);
    p.AddPoint(0,0);

    for (int y=30; y>=0; --y){
         for(int x=0; x<30; ++x){
           if (p.PointInPolygon(x, y))
             System.out.print("+");
           else
             System.out.print("-");
         }
         System.out.println();
    }
}
}
```

Das separate Testen jedes Punktes ist wenig effizient.

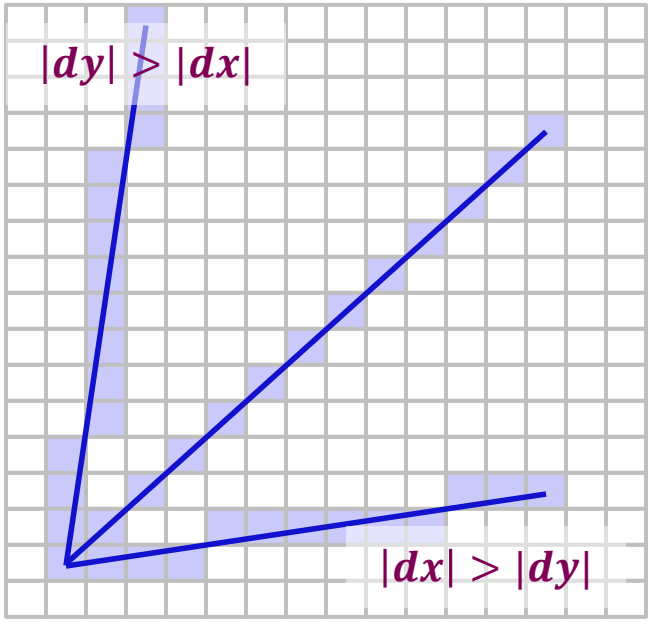
Bessere Möglichkeit?

Ausgabe

```
------------------------------
----------+-------------------
---------+++------------------
---------+++++----------------
--------++++++---------------
--------+++++++--------------
-------++++++++--------------
-------+++++++++-------------
------++++++++++------------
------+++++++++++-----------
-----++++++++++++----------
-----+++++++++++++----------
-----+++++++++++++----------
-----++++++++++++++---------
-----++++++++++++++---------
+++++++++++++++++++---------
+++++++++++++++++++---------
+++++++++++++++++++---------
+++++++++++++++++++---------
+++++++++++++++++++---------
+++++++++----------------------
+++++++++----------------------
+++++++++----------------------
+++++++++----------------------
+++++++++----------------------
++++++++-----------------------
++++++-------------------------
++++-------------------------
++-------------------------
------------------------------
```

Linien zeichnen -- Diskretisierung

- Suche zu kontinuierlicher Linie l die «nebeneinander liegenden» Pixel, deren Mittelpunkte minimalen Abstand $d \leq 0.5$ zu l haben.
- Potentielle Mehrdeutigkeit bei $d = 0.5$
- Unterscheidung in steile und flache Linien.
- Angestrebte Vermeidung von Fliesskommarechnung!



Bresenham Algorithmus

- Ganzzahliges Verhältnis $\frac{dx}{dy}$ gegeben mit $dx = x_1 - x_0, dy = y_1 - y_0$.
- Starte in $(x_0, y_0) \in \mathbb{Z}^2$
- Geradengleichung $y = y_0 + (x - x_0) \cdot \frac{dy}{dx}$

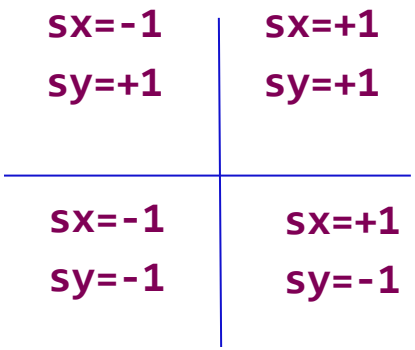
  anders geschrieben $dx(y - y_0) - dy(x - x_0) = 0$
- Ziel: Halte den Absolutwert des Minimierungsfehlers

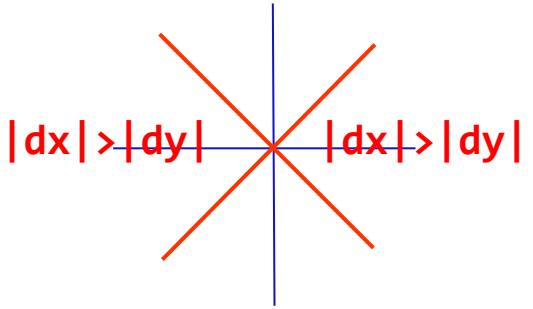
$$\text{err} = dx(y - y_0) - dy(x - x_0)$$

  klein
- Wenn z.B. $dx > dy > 0$ dann bleibt der Fehler beim Inkrementieren von x zuerst kleiner als beim Inkrementieren von y. Gilt dann $|err| > dx/2$ so kann er durch Inkrementieren von y wieder verkleinert werden.

Bresenham Algorithmus

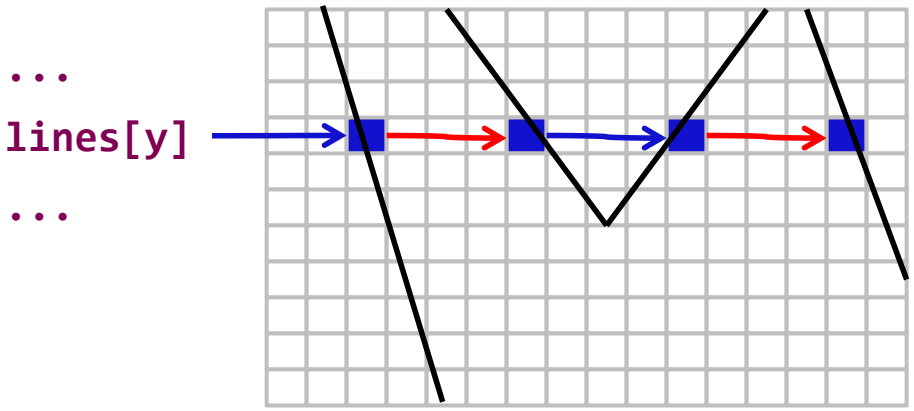
```
void line(int x0, int y0, int x1, int y1)
{
  int dx = Math.abs(x1-x0), sx = x0<x1 ? 1 : -1;
  int dy = Math.abs(y1-y0), sy = y0<y1 ? 1 : -1;
  int x= x0, y= y0;
  int err = 0;

  if (dx>dy)
   for (;;)
   {
      ZeichnePunkt(x,y);
      if (x==x1 && y==y1) break;
      x += sx;
      err += dy;
      if (err*2 > dx)
      {
         err -= dx;
         y += sy;
      }
  }
 else
   { ... } // symmetrisch in y
}
```





Polygone füllen

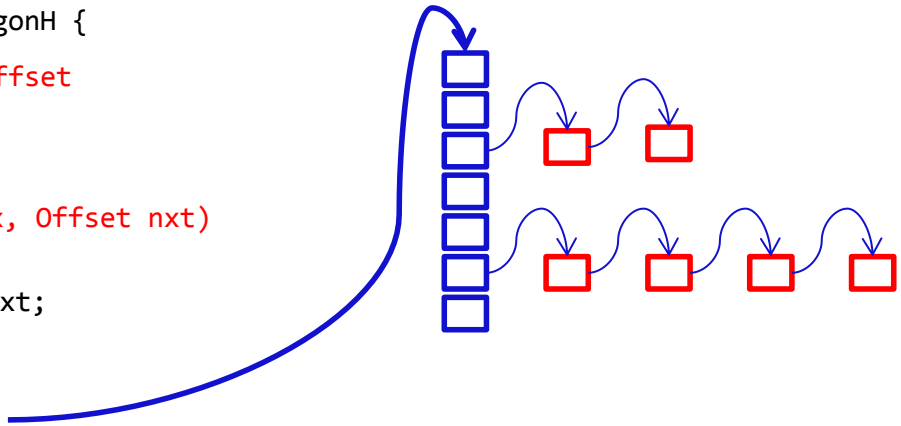
- Aufzählung der inneren Punkte durch Paare von (y,x0) – (y,x1) Koordinaten



- Zum Beispiel: Array über y, verkettete, sortierte Liste für jedes y über x
- Füllen der Datenstruktur durch modifizierten Bresenham-Algorithmus auf den Kanten des Polygons.

Polygon Datenstruktur

```
public class PolygonH {

  private class Offset
  {
     int x;
     Offset next;

     Offset(int xx, Offset nxt)
     {
          x=xx;
          next = nxt;
     }
  }

  Offset[] lines;

  PolygonH()
  {
     lines = new Offset[16]; // initial vertical size
  }

  void Grow(int y) // ensure that array is large enough to include index y, e.g. binary growth
  {...}

  void AddPoint(int x, int y)  // insert x sorted into list at y
  {
     Grow(y);
     if (lines[y] == null || x <= lines[y].x)
          lines[y] = new Offset(x,lines[y]);
     else
     {
          Offset ofs=lines[y];
          while (ofs.next != null && ofs.next.x < x )
               ofs = ofs.next;
          ofs.next = new Offset(x, ofs.next);
     }
  }
…
}
```

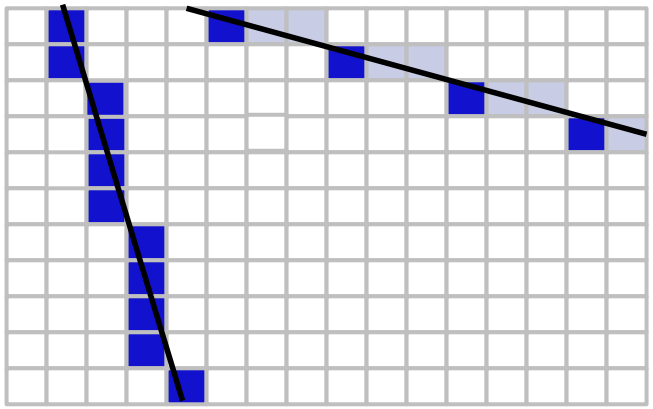


Fragen:

- Vorteile / Nachteile dieser Datenstruktur?
- Alternativen?
- **Warum haben wir uns gerade diese Richtung (horizontales Füllen) ausgesucht?**

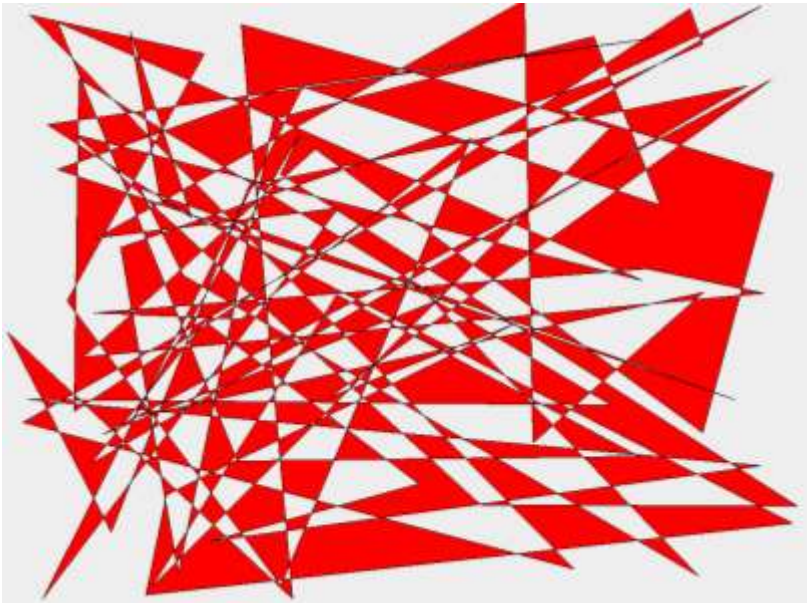
Einfügen einer Kante

```
public void addEdge(int x0, int y0, int x1, int y1)
{
  if (x0>x1) {
     int t=x0; x0=x1;x1=t;
     t=y0;y0=y1;y1=t;
  } // dx > 0

  int my = y0<y1 ? y0:y1;
  int dx = x1-x0;
  int dy = Math.abs(y1-y0), sy = y0<y1 ? 1 : -1;
  int x=x0;
  int y=y0;
  int err = 0;

  if (y != my) AddPoint(x,y); // lower most point omitted
  if (dx>dy) // flat lines
    do
    {
       x++;
       err += dy;
       if (err*2 > dx)
       {
          err -= dx;
          y += sy;
          if (y != my) AddPoint(x,y); // lower most point omitted
       }
    } while (y != y1);
  else // steep lines
    do
    {
       y += sy;
       err += dx;
       if (err*2 > dy)
       {
          err -= dy;
          x++; // right hand points omitted
        }
        if (y != my) AddPoint(x,y); // lower most point omitted
     } while (y != y1);
}
```



Füllen

```
public void fill(BufferedImage img, Color c)
{
  int color = c.getRGB();
  // Iteration über Zeilen
  for(int y=0; y<lines.length; ++y)
  {
    Offset ofs = lines[y];
    // Iteration über Paare von x-Werten
    while(ofs != null && ofs.next != null)
    {
     // Liniensegment füllen
    for (int x=ofs.x; x<ofs.next.x; ++x)
        img.setRGB(x, y, color);
      ofs = ofs.next.next;
    }
  }
}
```



- Das ganze Testprogramm wie immer auf der Vorlesungshomepage.