# Prioritätswarteschlange

Eine Prioritätswarteschlange
zur Verwaltung
der günstigsten Alternativen

# Prioritätswarteschlange

- Die Prioritätswarteschlange ist eine der Datenstrukturen, die zur Verwaltung von Alternativen bei Suchverfahren verwendet werden:
  - Stapel (**stack**) bei der Tiefensuche

    *last in first out*
  - Warteschlange (**queue**) bei der Breitensuche

    *first in first out*

  Beide ordnen die Elemente ***nach ihrem Auftreten*** ein.

# Prioritätswarteschlange

- Die Prioritätswarteschlange ist eine der Datenstrukturen, die zur Verwaltung von Alternativen bei Suchverfahren verwendet werden:
  - Stapel ...
  - Warteschlange ...
  - Prioritätswarteschlange (**priority queue**), beispielsweise bei Dijkstra

    Elemente werden ***geordnet nach ihrer Bewertung*** eingefügt.

# Prioritätswarteschlange

Allgemein

- Prinzipiell handelt es sich um ein objekt-orientiertes Vorgehen:
  Daten mit zugehörigen Zugriffsfunktionen
- Diese Variante wird hier für die betrachteten Wege bei der Suche des kürzesten Weges mit Dijkstra beschrieben

# Prioritätswarteschlange

## Konstruktor

- Im Konstruktor wird eine Wegeliste zur Verwaltung initialisiert.
  (*Die Klasse Weg für deren Elemente gehört ebenfalls zu diesem Projekt.*)

```
class prioSchlange():
    def __init__(self, initWeg):
        self.wege=[initWeg]
```

# Prioritätswarteschlange

## Methoden

- Das Sortieren benötigt ein Vergleichsprädikat ***vor*** , das gegebenenfalls vom erzeugenden Projekt kommt und in dem Fall dem Konstruktor übergeben werden muss.
  Hier ist ***vor*** als Methode vorgegeben.

```
def vor(self, erster, zweiter):
    return erster.gibLaenge()<zweiter.gibLaenge()
```

- Die Methode **`gibLaenge()`** wird von der Klasse Weg bereit gestellt. **`erster`** und **`zweiter`** sind also Instanzen dieser Klasse.

# Prioritätswarteschlange

## Methoden

```
def gibWege(self):
    return self.wege
def hatWege(self):
    return len(self.wege)>0

def holeErsten(self):
    if not self.hatWege(): return None
    return self.wege[0]

def entferneErsten(self):
    if not self.hatWege(): return None
    self.wege=self.wege[1:]
```

# Prioritätswarteschlange

Methoden

- fügt einen neuen Weg in die PrioWS ein

```
def fuegeEinenEin(self, neu):
    wege=[]+self.wege   ## echte Kopie
    pos=len(wege) # ggf hinten ran
    for weg in wege:
        if self.vor(neu,weg):
            pos=wege.index(weg)
            break
    self.wege.insert(pos,neu)
```

# Prioritätswarteschlange

Methoden

- fügt mehrere Wege in die PrioWS ein

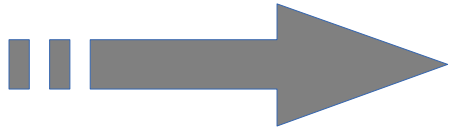
```
def fuegeEin(self, wege):
    for weg in wege:
        self.fuegeEinenEin(weg)
```

# Prioritätswarteschlange

Methoden

- entfernt alle Wege zum aktuellen Knoten bis auf den kürzesten

```
def entferneLaengere(self, aktuell):
    kuerzester=None
    wege=[]+self.wege    ## echte Kopie
    for weg in wege:     ## kuerzesten suchen
        if weg.gibZiel()==aktuell:
            if not kuerzester:
                kuerzester=weg
            elif self.vor(weg,kuerzester):
                kuerzester=weg
```



# Prioritätswarteschlange

Methoden

- entfernt alle Wege zum aktuellen Knoten bis auf den kürzesten.
  **`gibZiel()`** ist auch eine Methode der Klasse Weg.

```
def entferneLaengere(self, aktuell):
    ...
    for i in range(len(wege)):  # andere raus
        if wege[i].gibZiel()==aktuell:
            if wege[i]!=kuerzester:
                self.wege.remove(wege[i])
```

# Prioritätswarteschlange

Methoden

- Prüft, ob das Ziel erreicht ist.
  `letzter()` ist auch eine Methode der Klasse Weg.

```
def amZiel(self):
    if not self.hatWege(): return False
    aktuell=self.holeErsten()
    if aktuell.letzter()==aktuell.gibZiel():
            return True
    return False
```