ETH zürich

Übung 9 - Datenstrukturen

Informatik II

21./22. April 2026

Heutiges Programm

Recap: Datenstrukturen

Hash-Tabellen

Vergleich: Duplikate eliminieren

Quadtrees

Code Example: Stack und Queue

Prüfungsaufgaben

Hausaufgaben

1. Recap: Datenstrukturen

Datenstrukturen aus Informatik I

- Vektor **`vector`**
- Verkettete Liste **`llvec`**
- Set (Menge) **`unordered_set<T>`**
- Stack (Stapel, LIFO) **`stack`**
- Baum **`tnode`**

Datenstrukturen aus Informatik I

- Vektor **`vector`**
- Verkettete Liste **`llvec`**
- Set (Menge) **`unordered_set<T>`**
- Stack (Stapel, LIFO) **`stack`**
- Baum **`tnode`**

Informatik II bis jetzt:

- Liste **`list`**
- Set (Menge) **`set`**
- Dictionary **`dict`**

Datenstrukturen aus Informatik I

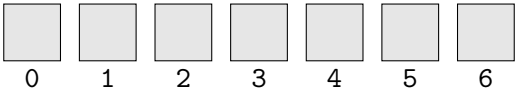
- Vektor **`vector`**
- Verkettete Liste **`llvec`**
- Set (Menge) **`unordered_set<T>`**
- Stack (Stapel, LIFO) **`stack`**
- Baum **`tnode`**

Informatik II bis jetzt:

- Liste **`list`**
- Set (Menge) **`set`**
- Dictionary **`dict`**

$\Rightarrow$ Potpourri von Datenstrukturen: Wähle die beste abhängig vom Kontext!
Heute werden wir die dafür nötigen Unterscheidungen lernen.

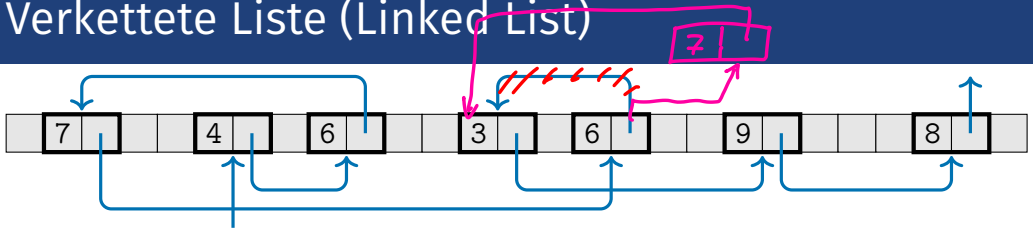
Liste



Operation	Laufzeit
Index-Zugriff	$\mathcal{O}(1)$
Suche (`in`)	$\mathcal{O}(n)$
Sortierte Suche	$\mathcal{O}(\log n)$
Einfügen	$\mathcal{O}(n)$
Entfernen	$\mathcal{O}(n)$

Sortierte Suche wird in Übung 6 behandelt

Verkettete Liste (Linked List)

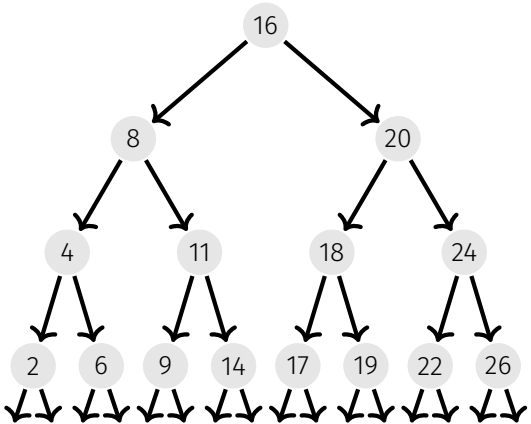


Operation	Laufzeit
Sortierte/Unsortierte Suche	$\mathcal{O}(n)$
Einfügen nach Knoten	$\mathcal{O}(1)$
Entfernen nach Knoten	$\mathcal{O}(1)$
Einfügen nach Wert	$\mathcal{O}(n)$
Entfernen nach Wert	$\mathcal{O}(n)$

Balancierter Binärer Suchbaum

Für jeden Knoten

- **linker Teilbaum**: kleinere Schlüssel
- **rechter Teilbaum**: grössere Schlüssel



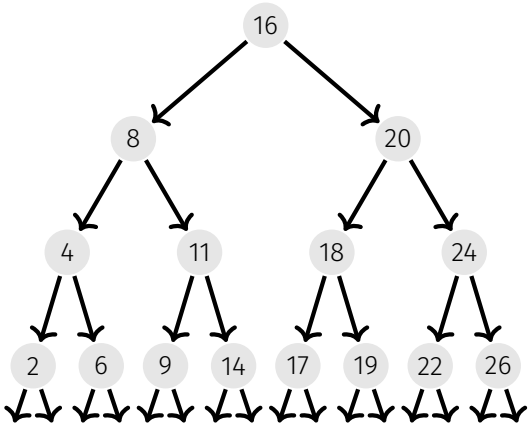
Balancierter Binärer Suchbaum

Für jeden Knoten

- **linker Teilbaum**: kleinere Schlüssel
- **rechter Teilbaum**: grössere Schlüssel

balanciert: Höhe $\mathcal{O}(\log n)$

Operation	Laufzeit
Suche	$\mathcal{O}(\log n)$
Einfügen	$\mathcal{O}(\log n)$
Entfernen	$\mathcal{O}(\log n)$

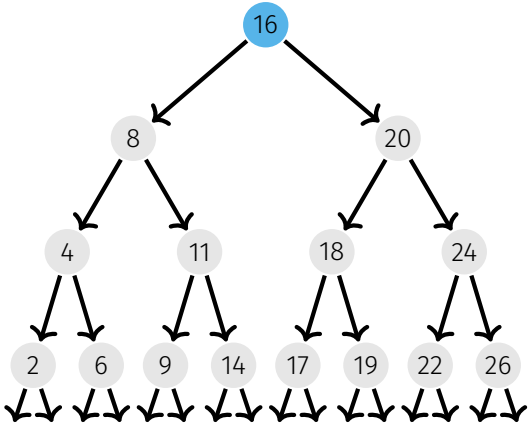


Balancierter Binärer Suchbaum

Für jeden Knoten

- **linker Teilbaum**: kleinere Schlüssel
- **rechter Teilbaum**: grössere Schlüssel

Suche nach 9

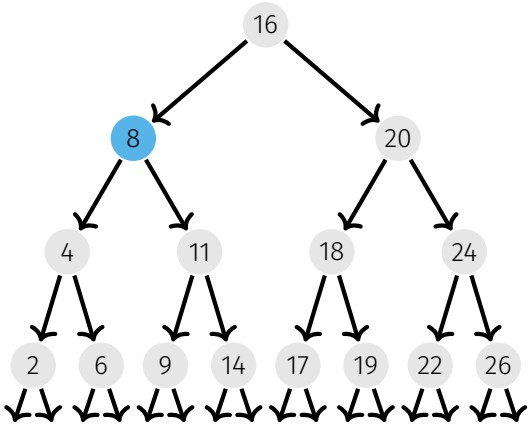


Balancierter Binärer Suchbaum

Für jeden Knoten

- **linker Teilbaum**: kleinere Schlüssel
- **rechter Teilbaum**: grössere Schlüssel

Suche nach 9

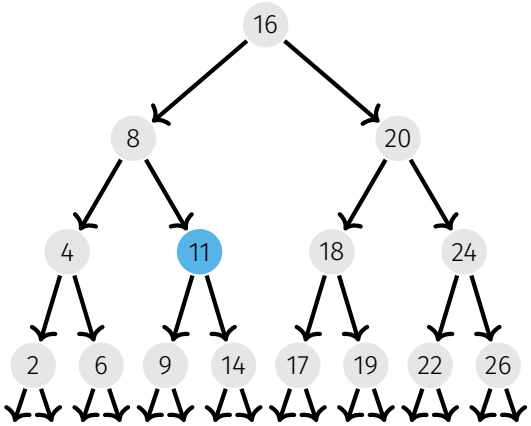


Balancierter Binärer Suchbaum

Für jeden Knoten

- **linker Teilbaum**: kleinere Schlüssel
- **rechter Teilbaum**: grössere Schlüssel

Suche nach 9

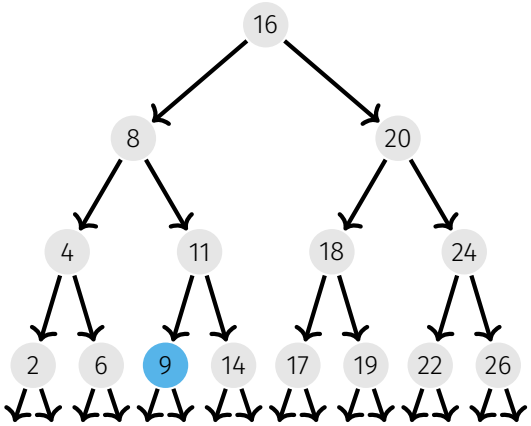


Balancierter Binärer Suchbaum

Für jeden Knoten

- **linker Teilbaum**: kleinere Schlüssel
- **rechter Teilbaum**: grössere Schlüssel

Suche nach 9

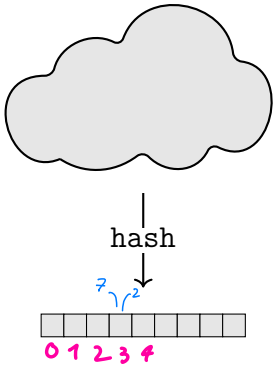


2. Hash-Tabellen

Hash-Tabelle

Genutzt, um ungeordnete Sammlungen einzigartiger Elemente zu speichern.

Operation	Erwartet	Schlechtester Fall
Suche	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Einfügen	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Entfernen	$\mathcal{O}(1)$	$\mathcal{O}(n)$



Hash-Funktionen

Welche Eigenschaften sollen Hash-Funktionen haben?

Hash-Funktionen

Welche Eigenschaften sollen Hash-Funktionen haben?

- konsistent (immer den gleichen Output für den gleichen Input)

Hash-Funktionen

Welche Eigenschaften sollen Hash-Funktionen haben?

- konsistent (immer den gleichen Output für den gleichen Input)
- möglichst kollisionsfrei: alle Inputs werden auf möglichst verschiedene Outputs gemappt, idealerweise gleichverteilt. Wenn mehrere Inputs beim gleichen Output überlappen, müssen wir Kollisions-Behandlung nutzen.

Hash-Funktionen

Welche Eigenschaften sollen Hash-Funktionen haben?

- konsistent (immer den gleichen Output für den gleichen Input)
- möglichst kollisionsfrei: alle Inputs werden auf möglichst verschiedene Outputs gemappt, idealerweise gleichverteilt. Wenn mehrere Inputs beim gleichen Output überlappen, müssen wir Kollisions-Behandlung nutzen.

Verschiedene Tabellen-Grössen: Nutze Index `hash(x) % M` für Tabelle der Grösse `M`

Kollisions-Behandlung

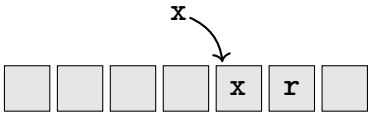
- Probing
  Wenn die Tabelle besetzt ist, wird der nächste freie Index gewählt

Kollisions-Behandlung

- Probing
  Wenn die Tabelle besetzt ist, wird der nächste freie Index gewählt

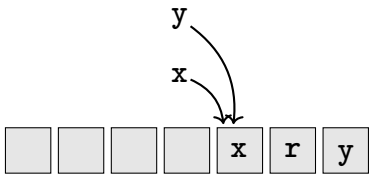
Kollisions-Behandlung

- Probing
  Wenn die Tabelle besetzt ist, wird der nächste freie Index gewählt



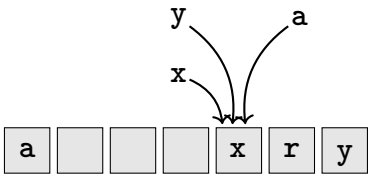
Kollisions-Behandlung

- Probing
  Wenn die Tabelle besetzt ist, wird der nächste freie Index gewählt



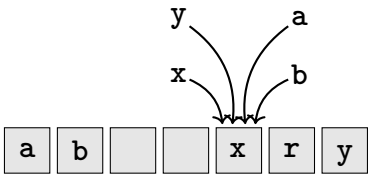
Kollisions-Behandlung

- Probing
  Wenn die Tabelle besetzt ist, wird der nächste freie Index gewählt



Kollisions-Behandlung

- Probing
  Wenn die Tabelle besetzt ist, wird der nächste freie Index gewählt

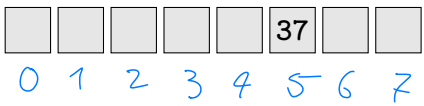


Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt

Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt



37 % 8

32 % 8 = 4

37 - 32 = 5

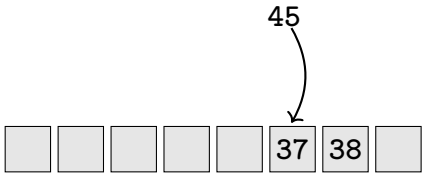
Modulo

Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt

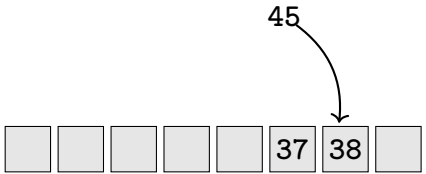
Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt



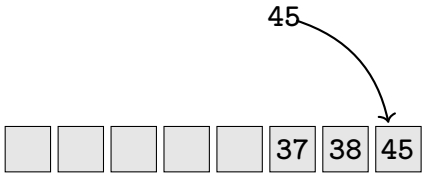
Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt



Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt



Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt

Kollisions-Behandlung

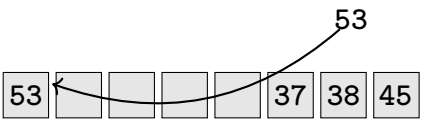
- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt

Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt

Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt



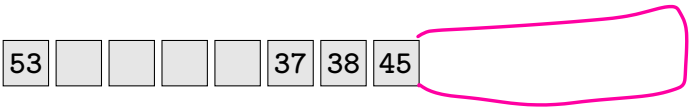
Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt

Wie viele Schritte braucht es, um 53 zu finden?

Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt



Wie viele Schritte braucht es, um 53 zu finden? 4 Schritte ausgehend von Index 5

Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt

Wie viele Schritte braucht es, um 53 zu finden? 4 Schritte ausgehend von Index 5
Wie viele Schritte braucht es, um sicherzustellen dass 61 NICHT enthalten ist?

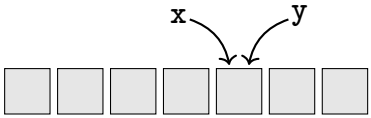
Kollisions-Behandlung

- Probing
  Beispiel: Die Hash-Funktion ist $H(x) = (x \mod 8)$
  Wenn Tabelle besetzt ist, wird der nächste freie Index gewählt

Wie viele Schritte braucht es, um 53 zu finden? 4 Schritte ausgehend von Index 5
Wie viele Schritte braucht es, um sicherzustellen dass 61 NICHT enthalten ist?
5 Schritte, beginnend bei Index 5 bis ein leerer Knoten erreicht wird

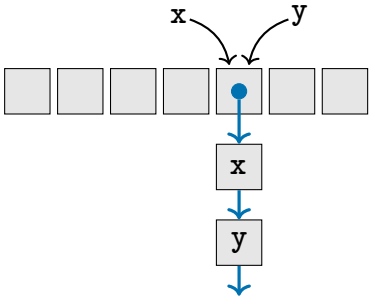
Kollisions-Behandlung

- Chaining:
  eine (verkettete) Liste von Elementen pro Tabellen-Eintrag



Kollisions-Behandlung

- Chaining:
  eine (verkettete) Liste von Elementen pro Tabellen-Eintrag



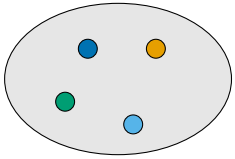
Kollisions-Behandlung

Probing und Chaining bieten beide gewisse Vorteile:

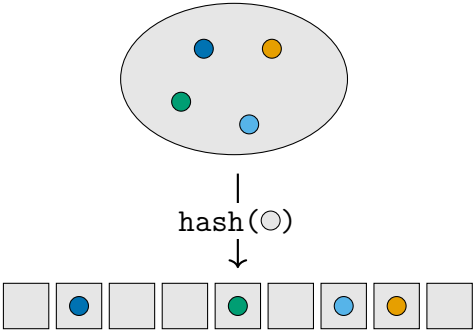
- Probing: Alle Daten bleiben im Array (Cache-freundlich)
- Chaining: Ein oft wiederholter Hash füllt nicht die ganze Hash-Tabelle, Zugriff auf andere Hashes bleibt schnell

Trotzdem haben beide Taktiken eine Worst-Case-Laufzeit von $\mathcal{O}(n)$.

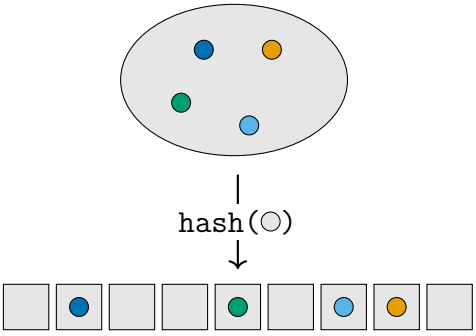
`set` als Hash-Tabelle



`set` als Hash-Tabelle



`set` als Hash-Tabelle



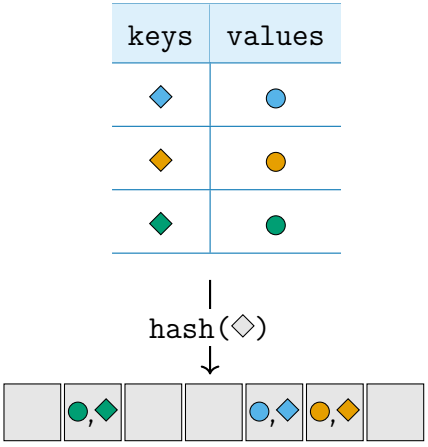
Operation	E	W
Suche	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Einfügen	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Entfernen	$\mathcal{O}(1)$	$\mathcal{O}(n)$

E: Erwartete Laufzeit
W: Laufzeit im Worst Case

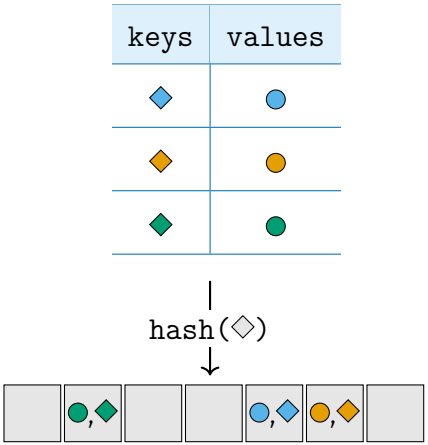
`dict` als Hash-Tabelle

keys	values
	
	
	

`dict` als Hash-Tabelle



`dict` als Hash-Tabelle



Operation	E	W
Suche	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Einfügen	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Entfernen	$\mathcal{O}(1)$	$\mathcal{O}(n)$

E: Erwartete Laufzeit
W: Laufzeit im Worst Case

Set/Dict als BBST?

- Könnte man ein Set auch als balancierten binären Suchbaum implementieren?
  Falls ja, wie und mit welcher Laufzeit? Falls nein, weshalb nicht?

Set/Dict als BBST?

- Könnte man ein Set auch als balancierten binären Suchbaum implementieren?
  Falls ja, wie und mit welcher Laufzeit? Falls nein, weshalb nicht?
  Ja! Jedes Element in der Menge ist ein Knoten im Suchbaum. $\mathcal{O}(\log n)$ für Einfügen/Entfernen/Suchen. Besser worst-case-Garantie als Hash-Tabellen.

Set/Dict als BBST?

- Könnte man ein Set auch als balancierten binären Suchbaum implementieren?
  Falls ja, wie und mit welcher Laufzeit? Falls nein, weshalb nicht?
  Ja! Jedes Element in der Menge ist ein Knoten im Suchbaum. $\mathcal{O}(\log n)$ für Einfügen/Entfernen/Suchen. Besser worst-case-Garantie als Hash-Tabellen.
- Und ein Dictionary?

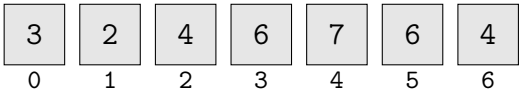
Set/Dict als BBST?

- Könnte man ein Set auch als balancierten binären Suchbaum implementieren?
  Falls ja, wie und mit welcher Laufzeit? Falls nein, weshalb nicht?
  Ja! Jedes Element in der Menge ist ein Knoten im Suchbaum. $\mathcal{O}(\log n)$ für Einfügen/Entfernen/Suchen. Besser worst-case-Garantie als Hash-Tabellen.
- Und ein Dictionary?
  Ja! Genau gleich, ausser dass ein Knoten ein Tupel `(k, v)` ist (sortiert nach Schlüssel).

3. Vergleich: Duplikate eliminieren

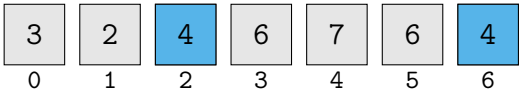
Duplikate entfernen mit einer Liste

Input: eine Liste `a`



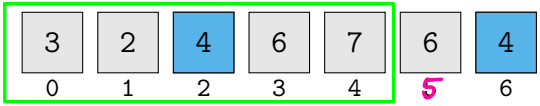
Duplikate entfernen mit einer Liste

Input: eine Liste `a`
Output: `True` falls ein Element mehrmals in `a` vorkommt, `False` sonst



Duplikate entfernen mit einer Liste

Input: eine Liste `a`
Output: `True` falls ein Element mehrmals in `a` vorkommt, `False` sonst



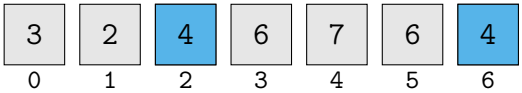

```
1  def has_duplicates(a):
2      """Return True if a has duplicates."""
3    for index, x in enumerate(a):
4      if x in a[:index]:
5        return True
6    return False
```

index = 5
x = 6

6 in a[:5]

Duplikate entfernen mit einer Liste

Input: eine Liste `a`
Output: `True` falls ein Element mehrmals in `a` vorkommt, `False` sonst




```
1  def has_duplicates(a):
2      """Return True if a has duplicates."""
3    for index, x in enumerate(a):
4      if x in a[:index]:
5        return True
6    return False
```

Laufzeit für $n = \texttt{len(a)}$: $\leq \sum_{i=1}^{n} \mathcal{O}(i) \in \mathcal{O}(n^2)$

Duplikate entfernen mit einem Set

```
1 def has_duplicates(a):
2     """Return True if a has duplicates."""
3   seen_elements = set()
4   for x in a:
5     if x in seen_elements:
6       return True
7     else:
8       seen_elements.add(x)
9   return False
```

Duplikate entfernen mit einem Set

```
1 def has_duplicates(a):
2     """Return True if a has duplicates."""
3   seen_elements = set()
4   for x in a:
5     if x in seen_elements:
6       return True
7     else:
8       seen_elements.add(x)
9   return False
```

Laufzeit für $n =$ `len(a)`:

Duplikate entfernen mit einem Set

```
1 def has_duplicates(a):
2     """Return True if a has duplicates."""
3   seen_elements = set()
4   for x in a:
5     if x in seen_elements:
6       return True
7     else:
8       seen_elements.add(x)
9   return False
```

O(1)

Laufzeit für $n = \texttt{len(a)}$:
erwartet: $\leq \sum_{i=1}^{n} c \in \mathcal{O}(n)$

Duplikate entfernen mit einem Set

```
1  def has_duplicates(a):
2      """Return True if a has duplicates."""
3    seen_elements = set()
4    for x in a:
5      if x in seen_elements:
6        return True
7      else:
8        seen_elements.add(x)
9    return False
```

Laufzeit für $n = \texttt{len(a)}$:
erwartet: $\leq \sum_{i=1}^{n} c \in \mathcal{O}(n)$
Worst Case: $\leq \sum_{i=1}^{n} c \cdot i \in \mathcal{O}(n^2)$

Duplikate entfernen mit einem BBST

```
1  def has_duplicates(a):
2      """Return True if a has duplicates."""
3    # create empty binary search tree
4    seen_elements = BST() # assuming class BST exists
5    for x in a:
6      if seen_elements.search(x):
7        return True
8      else:
9        seen_elements.add(x)
10   return False
```

Duplikate entfernen mit einem BBST

```
1  def has_duplicates(a):
2      """Return True if a has duplicates."""
3    # create empty binary search tree
4    seen_elements = BST() # assuming class BST exists
5    for x in a:
6      if seen_elements.search(x):
7        return True
8      else:
9        seen_elements.add(x)
10   return False
```

$O(\log(n))$

Laufzeit für $n = \texttt{len(a)}$: $\leq \sum_{i=1}^{n} c \cdot \log i \in \mathcal{O}(n \log n)$

4. Quadtrees

QuadTrees

Wozu brauchen wir sie?

- Schnellere Suche nach Punkten innerhalb eines Rechtecks im Vergleich zum naiven Ansatz.
- Effizient für die Abfrage mehrerer Rechtecke. $\mathcal{O}(\log n)$

Notiz: Die Konstruktion eines Quadtree braucht Zeit.

Anwendungsbeispiele

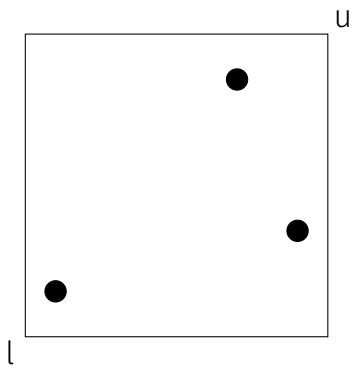
- Kann für Bildkompression und -optimierung verwendet werden.
- Ermöglicht eine effiziente räumliche Indizierung und Suche in geografischen Informationssystemen (GIS)
- Kann für eine effiziente Wegfindung in Robotik und autonomen Fahrzeugen verwendet werden.
- Nützlich zur Optimierung der Laufzeit der Finite Elemente Methode.

QuadTrees

Ein *Quadtree* ist ein Objekt mit den folgenden Attributen:

- Eine untere linke Ecke $l \in \mathbb{R}^2$.
- Eine obere rechte Ecke $u \in \mathbb{R}^2$.
- Eine maximale Kapazität $m \in \mathbb{N}$.
- Eine Liste p von Punkten in $\mathbb{R}^2$.
- Vier Referenzen auf Quadtree-Objekte: c_0, c_1, c_2, c_3.

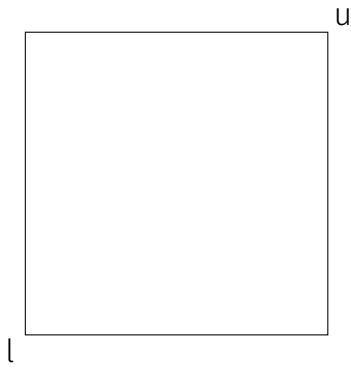
Ein Quadtree mit $m = 3$.



Idee

```
1  class QuadTree:
2    def __init__(self, l, u,
3    max_cap):
4      self.l = l
5      self.u = u
6      self.m = max_cap
7      self.points = []
8      self.children = None
9      self.count = 0
```

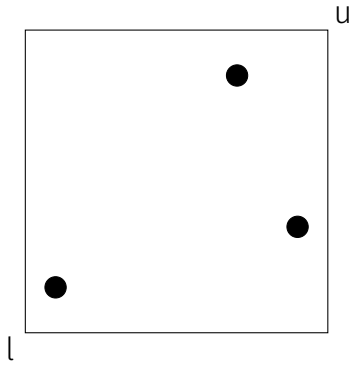
Ein Quadtree ist leer am Anfang.



Idee

- Man kann Punkte zu einem Quadtree q einfügen, welche in seinen Grenzen liegen.
- Die Punkte werden zu $q.p$ hinzugefügt, solange die Länge von $q.p$ seine maximale Kapazität $q.m$ nicht überschreitet.

Ein Quadtree mit $m = 3$.

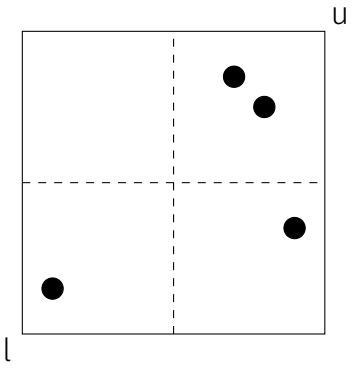


Idee

Sobald die Länge von $q.p$ die maximale Kapazität $q.m$ überschreitet, teilen wir q wie folgt in vier Quadtrees auf.

- Wir erstellen vier neue Quadtrees, die q wie in der Abbildung partitionieren.
- Wir speichern diese Quadtrees in $q.c[0]$, $q.c[1]$, $q.c[2]$, $q.c[3]$.
- Wir fügen jeden Punkt in $q.p$ zu dem Quadtree in $q.c$ hinzu, der den Punkt enthält.
- $q.p$ wird geleert ($q.p = []$).

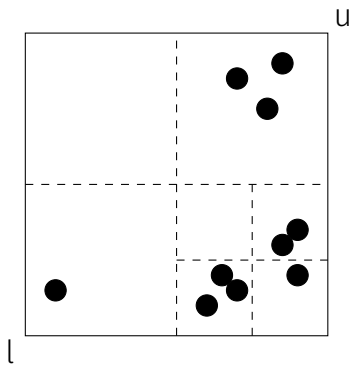
Ein Quadtree mit $m = 3$.



Idee

Das Hinzufügen weiterer Punkte kann zu einer weiteren Unterteilung der Quadtrees führen.

Ein Quadtree mit $m = 3$.

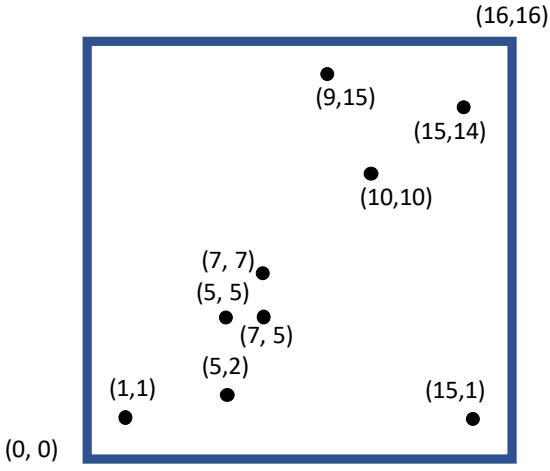


Übung: Erstellen eines Quadtree

Erstelle einen Quadtree mit den folgenden Informationen:

Max capacity : 2

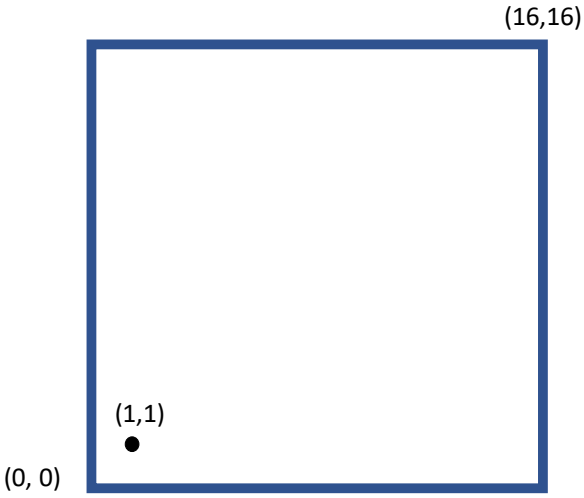
Points to insert:
(1,1)
(5,2)
(5,5)
(7,7)
(7,5)
(15,1)
(10,10)
(9,15)
(15,14)



Lösung: Erstellen eines Quadtree

Max capacity : 2

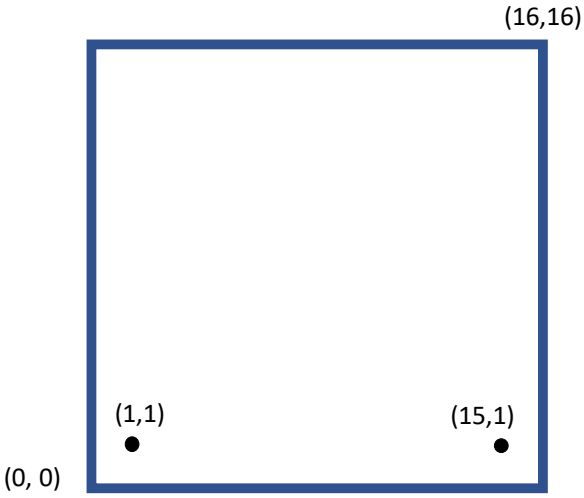
Points to insert:
(1,1)
(15,1)
(5,2)
(5,5)
(7,7)
(7,5)
(10,10)
(9,15)
(15,14)



Lösung: Erstellen eines Quadtree

Max capacity : 2

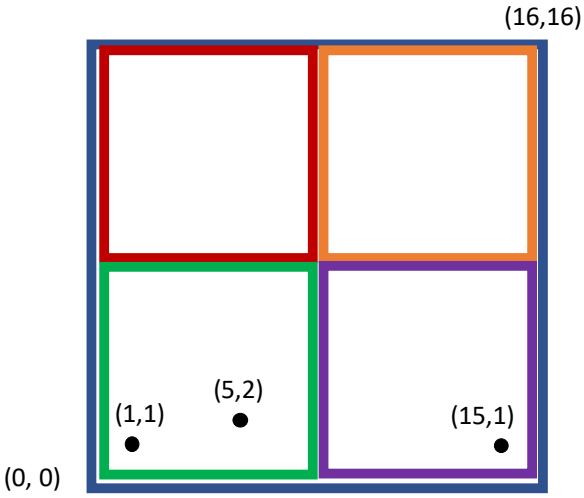
Points to insert:
(1,1)
(15,1)
(5,2)
(5,5)
(7,7)
(7,5)
(10,10)
(9,15)
(15,14)



Lösung: Erstellen eines Quadtree

Max capacity : 2

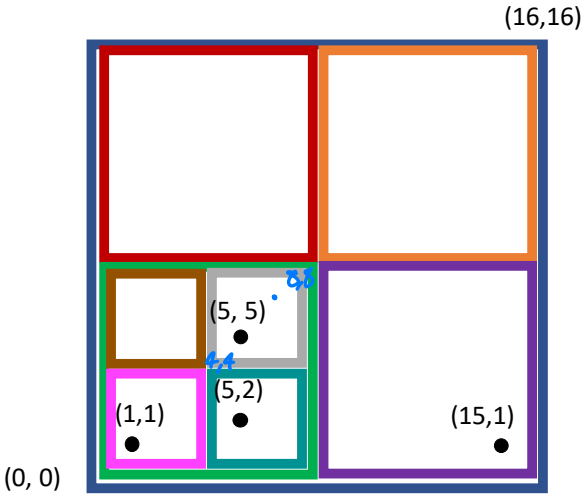
Points to insert:
(1,1)
(15,1)
(5,2)
(5,5)
(7,7)
(7,5)
(10,10)
(9,15)
(15,14)



Lösung: Erstellen eines Quadtree

Max capacity : 2

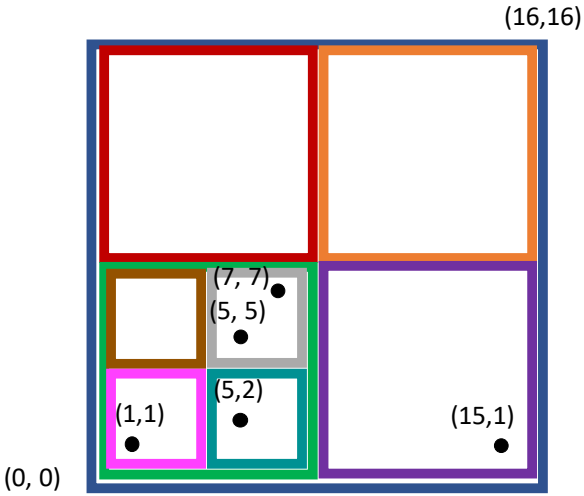
Points to insert:
(1,1)
(15,1)
(5,2)
(5,5)
(7,7)
(7,5)
(10,10)
(9,15)
(15,14)



Lösung: Erstellen eines Quadtree

Max capacity : 2

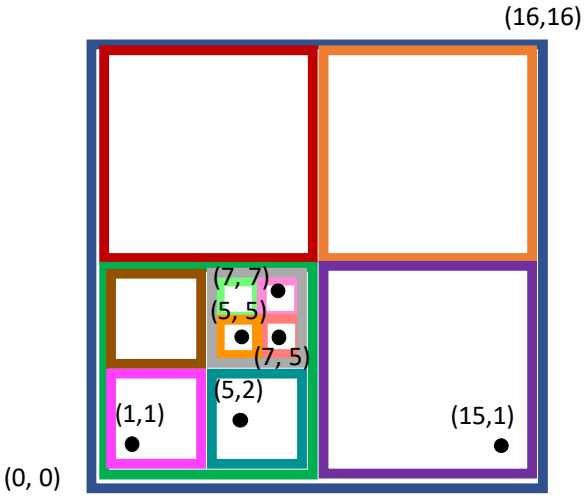
Points to insert:
(1,1)
(15,1)
(5,2)
(5,5)
(7,7)
(7,5)
(10,10)
(9,15)
(15,14)



Lösung: Erstellen eines Quadtree

Max capacity : 2

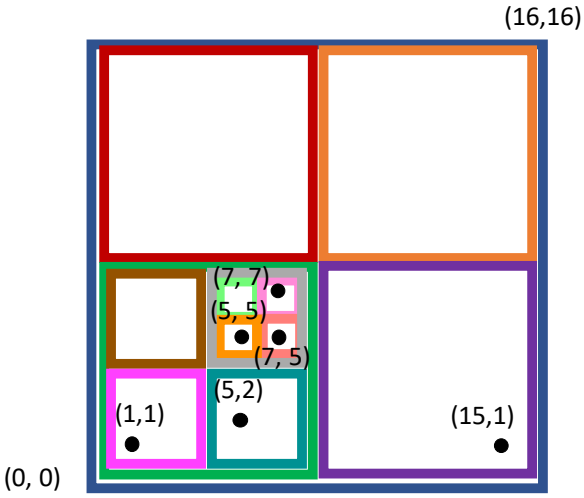
Points to insert:
(1,1)
(15,1)
(5,2)
(5,5)
(7,7)
(7,5)
(10,10)
(9,15)
(15,14)



Lösung: Erstellen eines Quadtree

Max capacity : 2

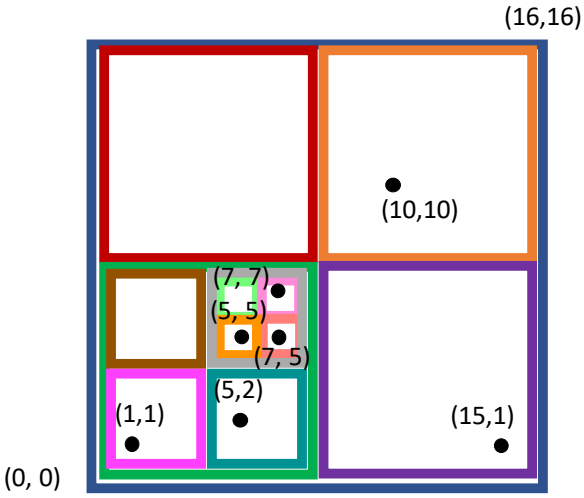
Points to insert:
(1,1)
(15,1)
(5,2)
(5,5)
(7,7)
(7,5)
(10,10)
(9,15)
(15,14)



Lösung: Erstellen eines Quadtree

Max capacity : 2

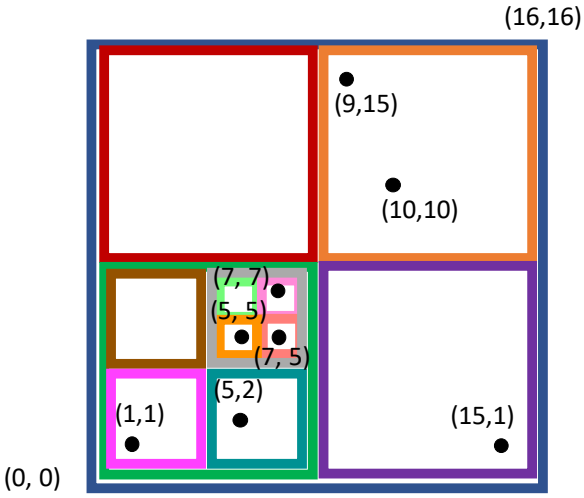
Points to insert:
(1,1)
(15,1)
(5,2)
(5,5)
(7,7)
(7,5)
(10,10)
(9,15)
(15,14)



Lösung: Erstellen eines Quadtree

Max capacity : 2

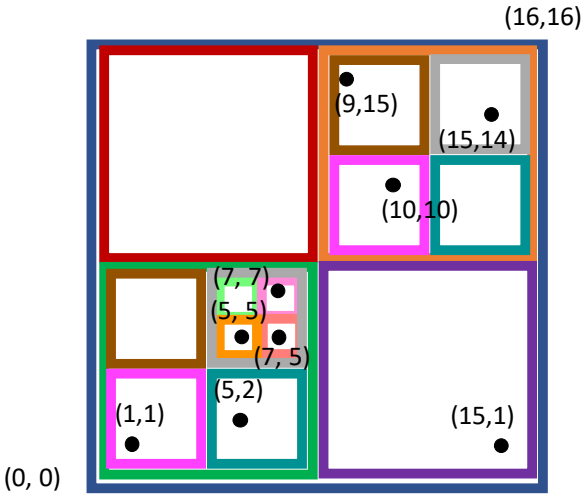
Points to insert:
(1,1)
(15,1)
(5,2)
(5,5)
(7,7)
(7,5)
(10,10)
(9,15)
(15,14)



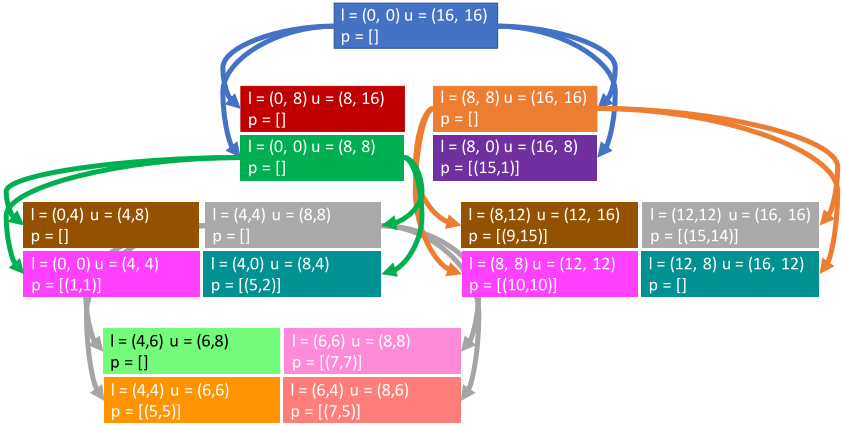
Lösung: Erstellen eines Quadtree

Max capacity : 2

Points to insert:
(1,1)
(15,1)
(5,2)
(5,5)
(7,7)
(7,5)
(10,10)
(9,15)
(15,14)

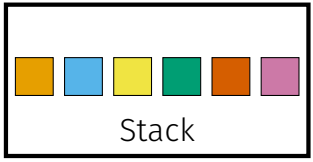


Die Lösung als Graph

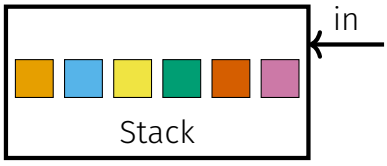


5. Code Example: Stack und Queue

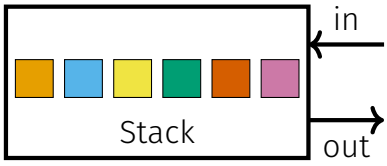
Stapel (Stack): Last In First Out



Stapel (Stack): Last In First Out

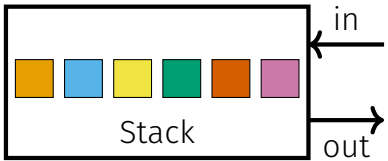


Stapel (Stack): Last In First Out

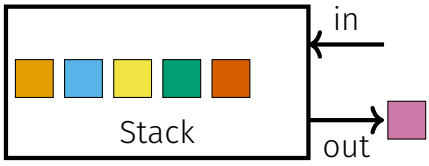


Stapel (Stack): Last In First Out

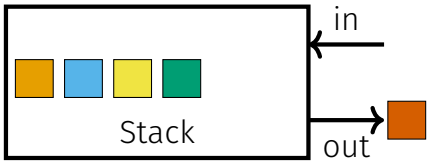
Im Folgenden ein Beispiel: Einige In-/Out-Aufrufe



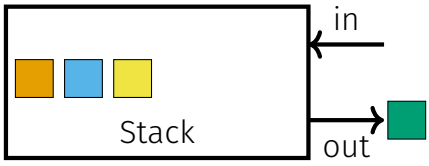
Stapel (Stack): Last In First Out



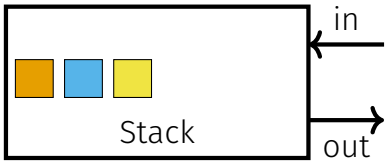
Stapel (Stack): Last In First Out



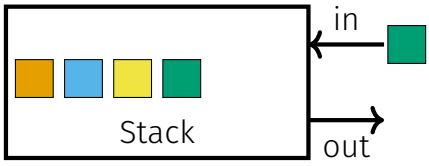
Stapel (Stack): Last In First Out



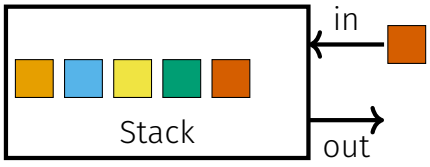
Stapel (Stack): Last In First Out



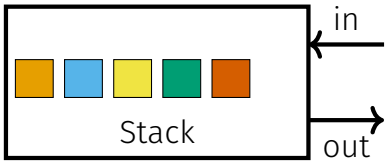
Stapel (Stack): Last In First Out



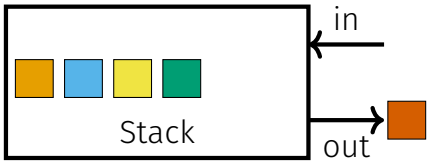
Stapel (Stack): Last In First Out



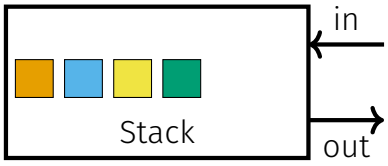
Stapel (Stack): Last In First Out



Stapel (Stack): Last In First Out



Stapel (Stack): Last In First Out



Code Example: Einen Stack implementieren

- Welche Operationen hat ein Stack?

Code Example: Einen Stack implementieren

- Welche Operationen hat ein Stack?
  Einfügen am Anfang, Entfernen am Anfang

Code Example: Einen Stack implementieren

- Welche Operationen hat ein Stack?
  Einfügen am Anfang, Entfernen am Anfang
- Welche Datenstruktur eignet sich dafür? Welche Laufzeit?

Code Example: Einen Stack implementieren

- Welche Operationen hat ein Stack?
  Einfügen am Anfang, Entfernen am Anfang
- Welche Datenstruktur eignet sich dafür? Welche Laufzeit?
  Verkettete Liste! $\mathcal{O}(1)$

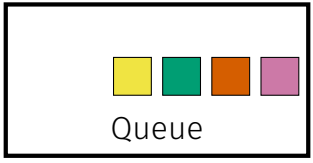
Gruppenübung 9: Stack

Implementiere einen Stack und teste seine Funktionalität.

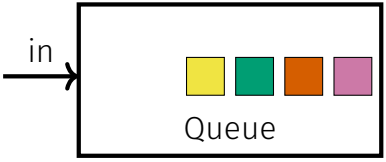
- Definiere die Klasse (Aufgabe 1 & 2);
- Implementiere "add" (Aufgaben 3)
- Implementiere "print" (Aufgabe 4);
- Implementiere "remove" (Aufgabe 5).

Mehr dazu in der detaillierten Aufgabenbeschreibung auf Code Expert.

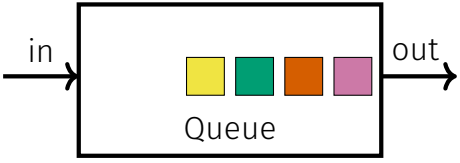
Warteschlange (Queue): First In First Out



Warteschlange (Queue): First In First Out

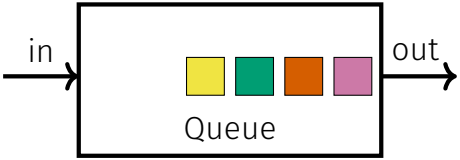


Warteschlange (Queue): First In First Out

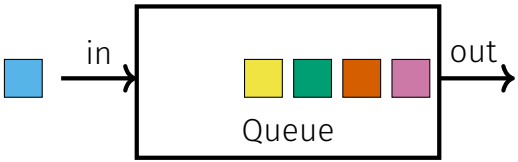


Warteschlange (Queue): First In First Out

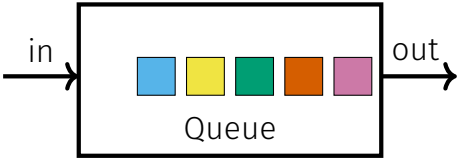
Im Folgenden ein Beispiel: Einige In-/Out-Aufrufe



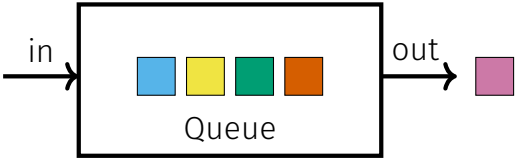
Warteschlange (Queue): First In First Out



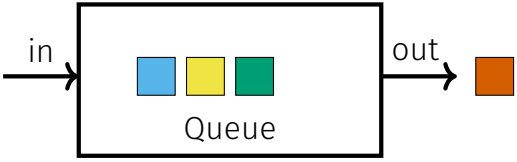
Warteschlange (Queue): First In First Out



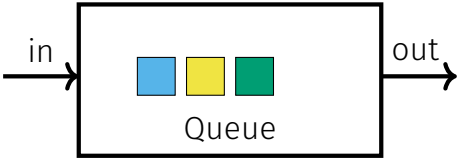
Warteschlange (Queue): First In First Out



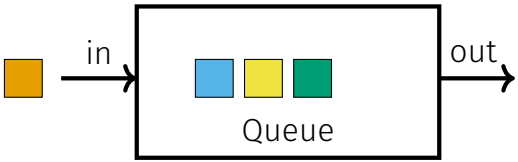
Warteschlange (Queue): First In First Out



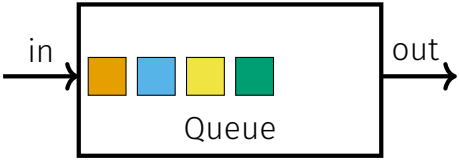
Warteschlange (Queue): First In First Out



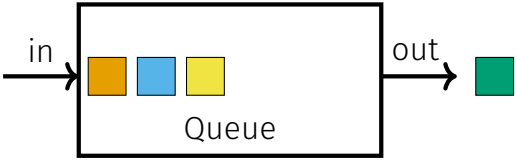
Warteschlange (Queue): First In First Out



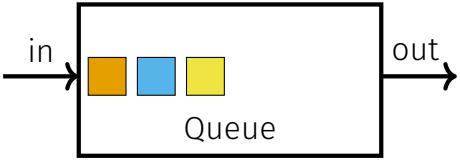
Warteschlange (Queue): First In First Out



Warteschlange (Queue): First In First Out



Warteschlange (Queue): First In First Out



Code Example: Queue implementieren

- Welche Operationen hat eine Queue?

Code Example: Queue implementieren

- Welche Operationen hat eine Queue?
  Einfügen am Anfang, Entfernen am Ende

Code Example: Queue implementieren

- Welche Operationen hat eine Queue?
  Einfügen am Anfang, Entfernen am Ende
- Zu welcher Laufzeit führt die verkettete Liste?

Code Example: Queue implementieren

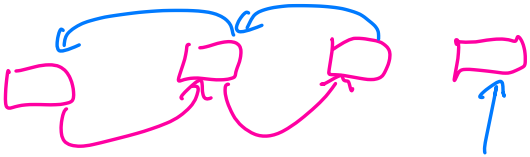
- Welche Operationen hat eine Queue?
  Einfügen am Anfang, Entfernen am Ende
- Zu welcher Laufzeit führt die verkettete Liste?
  Einfügen $\mathcal{O}(1)$, Entfernen $\mathcal{O}(n)$

Code Example: Queue implementieren

- Welche Operationen hat eine Queue?
  Einfügen am Anfang, Entfernen am Ende
- Zu welcher Laufzeit führt die verkettete Liste?
  Einfügen $\mathcal{O}(1)$, Entfernen $\mathcal{O}(n)$
- Welche Datenstruktur eignet sich besser dafür? Welche Laufzeit?

Code Example: Queue implementieren

- Welche Operationen hat eine Queue?
  Einfügen am Anfang, Entfernen am Ende
- Zu welcher Laufzeit führt die verkettete Liste?
  Einfügen $\mathcal{O}(1)$, Entfernen $\mathcal{O}(n)$
- Welche Datenstruktur eignet sich besser dafür? Welche Laufzeit?
  Doppelt verkettete Liste mit zusätzlichen Pointers in die andere Richtung!
  $\mathcal{O}(1)$



Gruppenübung 9: Queue

Implementiere eine Queue und teste ihre Funktionalität.

- Definiere die Klasse (Aufgabe 1 & 2);
- Implementiere "add" (Aufgaben 3)
- Implementiere "print" (Aufgabe 4);
- Implementiere "remove" (Aufgabe 5).

Mehr dazu in der detaillierten Aufgabenbeschreibung auf Code Expert.

Hash-Table

Gegeben eine Hash Tabelle mit n Elementen, was ist asymptotisch die schlechtest mögliche Anzahl Vergleiche die für das Hinzufügen eines Wertes nötig ist?

Given a hash table with n elements, what is the asymptotical worst case number of comparisons needed for inserting a value?

Anzahl Vergleiche? / *Number of comparisons?*

○ $\log n$ ○ $\sqrt{n}$ ⊗ n ○ $n \log n$ ○ n^2 ○ e^n ○ $n!$

Hash-Table 2

Gegeben die Hash Tabelle unten mit 11 Positionen, und dem Funktion $h(v) = v \bmod 11$, und 2 bereits eingetragenen Elementen. Fügen Sie die folgenden Schlüssel in der gegebenen Reihenfolge (von links nach rechts) in die Hash Tabelle ein. Kollisionen werden gelöst mittels Probing nach links (entgegengesetzte Richtung zur Vorlesung und Übungen!).

Given the hash-table below with 11 positions, using the hash map $h(v) = v \bmod 11$, with 2 elements inserted already. Enter the following keys in the given order (from left to right) into the hash-table. Collisions are resolved with probing to the left (opposite direction of what we did in class and exercises!).

Schlüssel / *Keys*: 26, 13, 3, 38

0	1	2	3	4	5	6	7	8	9	10
38	3	2	26	4	16					

Was ist der Index der Zelle in die 26 platziert wird? / *What is the index of the cell 26 is placed in?* 3

(3.f.2)

Was ist der Index der Zelle in die 13 platziert wird? / *What is the index of the cell 13 is placed in?* 2

(3.f.3)

Was ist der Index der Zelle in die 3 platziert wird? / *What is the index of the cell 3 is placed in?* 1

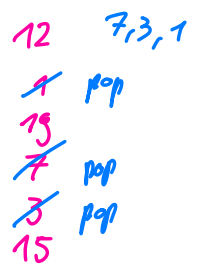
(3.f.4)

Was ist der Index der Zelle in die 38 platziert wird? / *What is the index of the cell 38 is placed in?* 0

Stack

Gegeben ein leerer Stapel (Stack) s, führen Sie die folgenden Operationen aus, und beantworten Sie anschliessend die Fragen zum resultierenden Stapel und der Ausgabe.

```
s.push(15)
s.push(3)
s.push(7)
print(s.pop())
print(s.pop())
s.push(19)
s.push(1)
print(s.pop())
s.push(12)
```



Was ist die resultierende Ausgabe? Tragen Sie diese Komma-getrennt ohne Leerschläge ein, also zum Beispiel `0,1,2`. / *What is the output created? Enter it comma-separated without spaces, e.g., `0,1,2`.*

7, 3, 1

(3.g.2)

Was ist der Inhalt des resultierenden Stapel? Tragen Sie diesen von oben nach unten (also das nächste auszugebende Element zuerst), und Komma-getrennt ohne Leerschläge ein, also zum Beispiel `0,1,2`. / *What is the content of the resulting stack? List it from top to bottom (so, first the element which would be output next) and enter comma-separated without spaces, e.g., `0,1,2`.*

12, 19, 15

Queue

Gegeben eine leere Warteschlange q, führen Sie die folgenden Operationen durch, und beantworten Sie dann die Fragen zur resultierenden Warteschlange und der Ausgabe. (Hinweis: "append" wurde in der Übung "add" genannt, und "pop" wurde "remove" genannt.)

```
q.append(9)
q.append(17)
print(q.pop())
q.append(2)
q.append(12)
q.append(4)
print(q.pop())
print(q.pop())
q.append(20)
```

Was ist die resultierende Ausgabe? Tragen Sie diese Komma-getrennt ohne Leerschläge ein, also zum Beispiel `0,1,2`. / *What is the output created? Enter it comma-separated without spaces, e.g., `0,1,2`.*

9, 17, 2

(3.g.2)

Was ist der Inhalt der resultierenden Warteschlange? Tragen Sie diesen von vorne nach hinten, und Komma-getrennt ohne Leerschläge ein, also zum Beispiel `0,1,2`. / *What is the content of the resulting queue? List it from front to back and enter comma-separated without spaces, e.g., `0,1,2`.*

12, 4, 20

7. Hausaufgaben

Aufgabe 8: Datenstrukturen

- Tisch-Reservierungssystem
- Hash-Tabelle mit Chaining
- Hash-Tabelle mit Probing
- Quadtrees

Fällig bis: **Montag, 27.04..2026, 20:00 CET**

KEINE HARTCODIERUNG

Fragen?

Feedback

https://jschultev.github.io/personal_website/Feedback