→ Datenstrukturen Übersicht

↳ Vielzahl an Datenstrukturen / Containers etc. je nach dem was man erreichen möchte
- → wie teuer ist Rechenleistung?
- → wie teuer ist Datentransfer?
- → wie teuer ist Speicherplatz?
- → welche Operationen (search, insert, delete, max, min etc...) sollen vorhanden sein? Für welche soll die Datenstruktur optimiert werden?

Kurze (nicht erschöpfende) Übersichten
- → Allg. Übersicht: https://medium.com/omarelgabrys-blog/diving-into-data-structures-6bc71b2e8f92
- → C++ Übersicht: https://www.programiz.com/cpp-programming/stl-containers
- → Python Übersicht: https://www.analyticsvidhya.com/blog/2021/06/datatypes-and-containers-in-python-a-complete-guide/

→ Es lohnt sich sehr oft kurz in der Dokumentation einer Library oder einfach Google nach einem Container zu suchen, um zu sehen welche Fkt. bereits implementiert sind.

→ Balanced BST

(Bsp. AVL trees)

→ O(h) ist meist die worst-time complexity ⇒ h soll möglichst nah an log n sein!

→ Hash Tables / Functions

Wir fügen ein Element x in eine Tabelle von Grösse M ein:

input x ↓

hash(x) → Wert → index = Wert % M

↳ stellt sicher, dass der Index innerhalb der Liste/des Arrays ist.

→ Problem: Was machen falls zwei Elemente an die gleiche Stelle geschrieben werden sollen? ≙ Wie gehen wir mit Kollisionen um?

falls das Key-set von Beginn an bekannt, kann h(k) perfekt gewählt werden.

- → open hashing: Einträge ausserhalb der Tab. erlaubt ~ dynamisch
- → closed hashing: Einträge ausserhalb Tab nicht erlaubt
- → open addressing: h-Index nicht der gleiche für Kollisionen
- → closed addressing: h(ki) gleich ≠ Kollisionen

(In der Vorlesung kamen nur die mit grünem Pfeil soweit ich weiss)

→ linear probing => bei Kollision einfach den Hashindex des Key den wir einfügen +=1 rechnen

=> Nachteil: primäre Häufung; ähnliche (Schlüssel, haben ähnliche) Hashadressen haben ähnliche Sondierungsfolge d.h. es bilden sich lange zusamh. volle Bereiche in der Hashtabelle.

Primary Clustering heisst d. falls der Hashindex in einem bereits existenten Cluster gelandet wäre, dass sich das Cluster dann vergrössert.

→ Quadratic Probing => Hashindex um 1, 4, 9, 16 ... verschieben.

Nachteil: secondary clustering; führt zu cluster vergrösserung nicht in der Nähe des orginal hashindex

k_1, k_2 haben nur die gleiche Sondierungsfolge, wenn der orginal Hashindex gleich war.

→ double hashing => zweite, unabhängige H wählen.

Bsp $h(k) = k \bmod 7$, $h_2(k) = 1 + k \bmod 5$

=> hashindex $s(k, j) = h(k) \pm j\, h_2(k)$, $j \in \mathbb{N}$

↳ ⊖ falls nach links sondieren

→ Cuckoo hashing => zwei Indextabellen, falls Kollision in T_1 → auf T_2 wechseln mit $h_2(k)$.

→ chaining: Elemente e.g. als linked list an die korrekte Stelle anhängen.

→ weiterführende Links:

https://iq.opengenus.org/time-complexity-of-hash-table/

https://khalilstemmler.com/blogs/data-structures-algorithms/hash-tables/

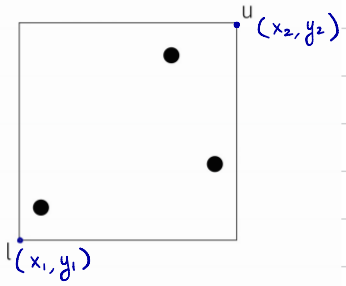
→ Quadtrees

A *quadtree* is an object with the following attributes:

- A lower left corner $l \in \mathbb{R}^2$.
- An upper right corner $u \in \mathbb{R}^2$.
- A maximum capacity $m \in \mathbb{N}$.
- A list p of points in $\mathbb{R}^2$.
- Four references to quadtree objects: c_0, c_1, c_2, c_3.

max_cap

A quadtree with $m = 3$.



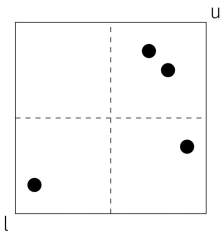

```
1  class QuadTree:
2    def __init__(self, l, u, max_cap):
3      self.l = l
4      self.u = u
5      self.m = max_cap
6      self.points = []
7      self.children = None
8      self.count = 0
```

subdivide:

As soon as $q.p$'s length exceeds $q.m$, we divide q into four as follows.

- We create four new quadtrees that partition q as shown in the figure.
- We store those quadtrees in $q.c[0]$, $q.c[1]$, $q.c[2]$, $q.c[3]$.
- We add each point in $q.p$ to the quadtree in $q.c$ that contains the point.
- $q.p$ is emptied ($q.p = []$).

A quadtree with $m = 3$.



- Schnellere Suche nach Punkten innerhalb eines Rechtecks im Vergleich zum naiven Ansatz.
- Effizient für die Abfrage mehrerer Rechtecke.
- 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.

Anmerkung: Der Aufbau des QuadTrees kann zeitaufwendig sein, aber die Vorteile einer effizienten Abfrage und räumlichen Indizierung überwiegen die Nachteile des Aufbaus in vielen Anwendungen.