Datenstrukturen

Verlinkte Listen

- Verlinkte Datenstruktur
- Ermöglicht effizientes einfügen an allen Positionen
- Nicht cache-effizient
- Kein wahlfreier Zugriff
- Slicing
- `size()` in $O(n)$ oder $O(1)$
- In C++: `std::list` und `std::forward_list`

Dynamische Arrays

- Handle + großer, zusammenhängender Speicher
- Potentiell sehr Cache-effizient (in Java nicht so sehr)
- Wahlfreier Zugriff
- `push_back()` in amortisiert $O(1)$
- Einfügen in der Mitte in $O(n)$
- Kein slicing
- `size()` in $O(1)$
- In C++: `std::vector`
- In Java: `ArrayList`

Zyklische Arrays

- Dynamische Arrays mit Modulo-Index
- Im Prinzip gleiches Verhalten, außer bei einfügen/entfernen vorne

Amortisierte Analyse

Prinzip

- Berechne Kosten für n Operationen
- Teile das Ergebnis durch n

Beispiel

- `push_back()` von dynamischen Arrays in amortisiert $O(n)$
- Kosten im Standardfall: 1
- Kosten bei Reallokation eines Arrays der Größe n: n + 1
- Reallokationen wenn: $\exists k \in N : n = 2^k$
- Beweis dass $n\times$ `push_back()` $\in O(n)$ an Tafel
- $O(n)/n = O(1)$ (nur hier so rechnen!)

Hashmaps

Übersicht

- Datenstruktur mit **erwarteter** Laufzeit, **nicht amortisierter**
- definiere `hash(ValueType) -> word`. Der Rückgabewert wird als Hashwert bezeichnet.
- Es wird ein Array von Containern und eine Funktion die für jeden Hashwert einen gültigen Index zurückgibt angelegt.
- Ein Container dient oft für mehr als einen Hashwert!
  $\Rightarrow$ Kollisionsgefahr
- Zugriff in erwartet $O(1)$, worst-case aber $O(n)$
- Gute Hashfunktion **essentiel**!
- Kollisionen von Hashmaps stellten und stellen reale Angriffsvektoren für Denial-of-Service-Attacken dar!

Beispiel

```
fn hash(s: string) -> uint // this algorithm is horrible!
    sum := 0
    for character in s:
        sum += cast_to<uint>(character)
    return sum

class hash_set:
private:
    array<array<string>> data;
    fn hash_to_index(hash: uint) -> uint
        return hash % data.size()
public:
    fn insert(str: string) -> void
        index := hash_to_index(hash(str))
        if data[index] contains str:
            throw exception{str + " already in set"}
        else: data[index].push_back(str)
```

Beispiel

```
fn main() -> int
    var set: hash_set
    for str in ["foo", "bar", "baz"] :
        map.insert(str)
```

```
fn main() -> int
    var set: hash_set
    for str in ["abc", "cab", "bca",
                "cba", "acb", "bac"] :
        map.insert(str)
```

Wahrscheinlichkeitstheorie

Wahrscheinlichkeitstheorie

- Sei $p \in [0, 1]$ die Wahrscheinlichkeit für ein Ereignis
- Die Wahrscheinlichkeit für n-maliges Eintreten in n Versuchen ist p^n
- Die Wahrscheinlichkeit für einmaliges Eintreten in n Versuchen ist $1 - (1 - p)^n$
- Zufall hat kein Gedächtnis!