Repetitorium 25.03.2021

Mittwoch, 24. März 2021 19:11

Beispiel: Stapel mit Push(key k), Pop() und Multipop(int k)

Push	O(1)
Pop	O(1)
Multipop	O(k)

Menge an n Operationen -> O(n) * WorstCase einer einzelnen Operation
= O(n) * O(n) = O(n^2)

Amortisierte Analyse

Menge an k Operationen => Bestimme "durchschnittliche", amortisierte Laufzeit

Aggregationsmethode

1. Alle Operationen betrachten
2. Alle Laufzeiten aufsummieren
3. Durch n teilen => amortisierte Laufzeit

Angenommen, die Operationenfolge enthält genau m Push Operationen. => n - m Pop() oder Multipop()-Operationen

Alle Push-Operationen zusammen haben Laufzeit O(m) * WorstCase von Push = O(m) * O(1) = O(m)

Alle Pop() und Multipop()-Operationen können höchsten die m gepushten Elemente wieder herunternehmen
=> Summe von Pop() und Multipop() liegt ebenfalls in O(m)

Die Summe aller Operationen liegt in O(m) liegt in O(n)

=> Die amortisierten Kosten einer einzelnen Operation liegen in O(1)

Buchhaltermethode

Idee: Billige Operationen zahlen für teure Operationen mit

1. Für alle Operationen amortisierte Kosten definieren ^c_i
2. Zeigen, dass Summe über c_i <= Summe über ^c_i
3. Zeigen, dass unser "Guthaben" nie unter 0 fällt
4. => ^c_i sind gültige amortisierte Kosten

Operation	c_i	^c_i
Push	1	2
Pop	1	0
Multipop(k)	Min(size, k)	0

Summe ^c_i >= Summe c_i, denn die Kosten für die Herausnahme werden bereits von Push mitbezahlt. Es können nie mehr Elemente herausgenommen werden, als gepusht wurden. Daher kann unser Kontostand auch nie unter 0 fallen.

=> Amortisierte Kosten jeder Operation liegen in O(1)

Potentialmethode

Idee: Verbinde eine Potentialfunktion mit einer "echten" Größe

1. Wähle eine Potentialfunktion Phi in Abhängigkeit von einer echten Größe
2. Zeige, dass Phi immer größer 0 ist, d.h. Phi(i) >= 0 für alle 0 <= i <= n
3. Berechne ^c_i = c_i + Phi(i) - Phi(i-1)

Phi(i) = Größe des Stacks nach der i-ten Operation
Phi(i) >= 0, denn der Stapel kann höchsten leer sein

Operation	c_i	DeltaPhi	^c_i
Push	1	1	2
Pop	1	-1	0
Multipop(k)	Min(size, k)	-Min(size, k)	0

=> die amortisierten Kosten jeder Operation liegen in O(1)

Dynamische Programmierung

Programmierprinzip: Aufteilung in kleinere Teilinstanzen, Lösung des großen Problems

Abgrenzung zu Teile und Herrsche:
- Überlappende Teilinstanzen statt disjunkte Teilinstanzen
- Bottom-up statt top-down
- n-dimensionales Array zur Zwischenspeicherung der Teilinstanzen statt direkte Übergaben, höchstens mit Caching
- Meist iterativ statt meist rekursiv

Greedy-Algorithmen
- Keine dynamische Bestimmung einer optimalen Lösung
- Kleine Teilschritte, die immer so fortfahren wie es im Augenblick optimal erscheint
- Die meisten Graphalgorithmen
- Bsp. Kruskal: Immer die leichteste Kante nehmen, die zwei Sets verbindet
- Sehr leicht zu implementieren

Fahrplan zu Dynamischem Programmieren
1. Struktur einer optimalen Lösung charakterisieren
2. Wert der optimalen Lösung rekursiv definieren
3. Erst kleinste Teilinstanzen lösen (und Lösung zwischenspeichern), dann mittels der rekursiven Definition immer größere Teilinstanzen lösen, bis optimale Lösung des Gesamtproblems gefunden ist
4. (Optimale Lösung rekonstruieren bzw. Algo so anpassen, dass er die Struktur mitspeichert)

Beispielaufgabe: Zwillinge haben gleichzeitig Geburtstag. Die Eltern kaufen eine Menge an Geschenken mit monetären ganzzahligen Werten W = {w_1, ..., w_n} und Gesamtwert v. Ist es möglich, die Geschenke fair zwischen den Zwillingen aufzuteilen?

1. optimale Lösung: Es ist möglich, d.h. es existiert eine Teilmenge T aus W mit Summe über T = v/2.
2. Für eine Teilmenge T ist die Summe s erstellbar, wenn ich aus T \ {w_i} die Summe s - w_i oder s bilden kann.
Wenn ich aus der Teilmenge T die Summe s erstellen kann, dann kann ich aus T vereinigt {w_i} die Summen s und s + w_i erstellen.

3. Wir beginnen bei der Teilmenge {} (leere Menge). Für diese kann ich die Summe 0 erstellen.
Nimm nun schrittweise die Geschenke w_1, w_2, ... w_n dazu und speichere nach jedem Schritt, welche Summen möglich sind. Dazu ein langes Array int values[v], wobei values[i] genau dann true ist, wenn die Summe i (mit den bisherigen) möglich ist.
Gib values[v/2] aus.

```
If(v % 2 != 2 0) return false
Int[] values = new int[v+1] //gefüllt mit false
Values[0] = true;
For(int w in W){
        For(int i = v+1 - w; i >= 0; i--){
                If(values[i] == true){
                        Values[i + w] = true;
                }
        }
}
Return values[v/2]
```

W = {3, 4, 2, 2, 1} => v = 12

1	0	0	0	0	0	0	0	0	0	0	0	0	0

w_1 hinzunehmen:

1	0	0	1	0	0	0	0	0	0	0	0	0	0

w_2 hinzunehmen:

1	0	0	1	1	0	0	1	0	0	0	0	0	0

w_3 hinzunehmen:

1	0	1	1	1	1	1	1	0	1	0	0	0	0

w_4 hinzunehmen:

1	0	1	1	1	1	1	1	1	1	0	1	0	

w_5 hinzunehmen:

1	1	1	1	1	1	1	1	1	1	1	1	1	1

Die Summe v/2 = 6 ist möglich => Die Geschenke lassen sich fair aufteilen

Hashing

h(k) = (2 * k + 5)%7

31	11	21	32	42	62	77

Verkettung

0	1	2	3	4	5	6
			62	31	77	32
					42	11
					21	

Linear

Wenn erster Platz besetzt, rücke weiter bis freie Stelle gefunden

h'(k, i) = (h(k) + i) % Array.length

h'(k,0) -> h'(k, 1) -> ...

0	1	2	3	4	5	6
32	42	77	62	31	21	11

Quadratisch

Wenn erster Platz besetzt, rücke in größer werdenden Schritten weiter, bis Platz gefunden

h'(k,i) = h(k) + 2i + 3i^2

Eigentlich h'(k,i) = (h(k) + 2i + 3 i^2) % Array.length

0	1	2	3	4	5	6
	32		42	31	21	11

Doppelt

Wenn Platz belegt, springe um festen Wert weiter
Sprungweite wird durch zweite Hashfunktion bestimmt

h'(k,i) = h(k) + i * ((4k + 3)%5)

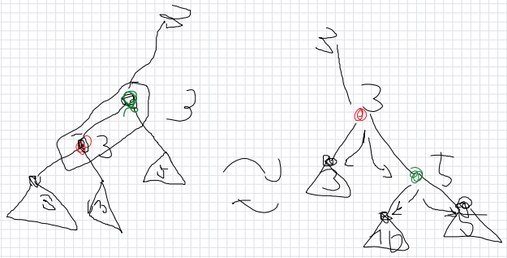
h'(k,i) = h(k) + i * (7 - 2 * (k % 3))

0	1	2	3	4	5	6
62	77	42	32	31	21	11

Gutes Hashen?
- Hashtabelle sollte größer als die Anzahl der zu erwartenden Werte, am besten prime Größe
- Primäre Hashfunktion: Faktor sollte teilerfremd zur Arraylänge sein (am besten prim)
- Hashfunktionen sollten schnell zu berechnen sein
- Sprungweite bei Linearem/Quadratischen/Doppeltem Hashing sollte teilerfremd zur Arraylänge sein

Blatt 3 Aufgabe 4

```
minGap(RSBaum tree){
        Return tree.root.minGap
}

Insert(Node n){

        Führe normales RBInsert aus, noch ohne FixUP

        s = successor(n)
        p = predecessor(n)

        If( | s - n| < | p - n| ){
                n.minGapNode = (n, s)
        } else {
                n.minGapNode = (p,s)
        }

        Node u = n;
        While( | u.minGap[0] - u.minGap[1] | <  |u.parent.minGap[0] - u.parent.minGap[1]| ){
                u.parent.minGap = u.minGap;
                u = u.parent;
        }

        Führe RBFixUP aus, halte minGap in O(1) aufrecht
        Dazu ist es nur nötig in den Rotationen die minGap-Werte der beiden rotierten Knoten mit ihren unteren Teilbäumen abzugleichen. Weiter oben kann sich nichts ändern, da die minGap des gesamten rotierten Teilbaums sich nicht ändert.

}
```



Cheatsheet - Cheatsheet

Laufzeiten, Stabilität, in-situ von Sortiertalgorithmen

Bedingungen für Linearzeitsortierung

Laufzeiten von anderen wichtigen (Graphen-)algorithmen
- Dijkstra
- BFS
- DFS
- Min. Spanbaum (Janik-Prim / Kruskal)
- Leichter Kreis
- TSP

Bekannte Summenformeln

Rekursionsgleichung
- Kleines Beispiel für Rekursionsbaummethode
- Meistermethode (3 Fälle + Regularitätsbedingung)

Datenstrukturen + Operationen + Laufzeiten
- Stapel
- Schlange
- Liste
- Heap + Max-Heapify, BuildHeap

5 RS-Baum Eigenschaften + evtl. die Rotationen

Fahrplan zu dynamischer Programmierung

Hashingtypen (+ Laufzeiten bei erfolgreicher/erfolglose Suche)

Quantordefinition für O, Omega, Theta, o, omega