Algorithmen

Michael Kaufmann

8/11/2021 und 9/11/2021
Prioritätswarteschlangen - Hashing

Prioritätswarteschlangen:
Anwendung der bisherigen Datenstrukturen (Arrays, Listen, Heaps)

Prioritätswarteschlangen (Priority Queues)

Szenario: Notaufnahme im Krankenhaus
– Immer neue Patienten kommen rein. Jeder erhält eine 'Priorität', je nach Dringlichkeit. Die Fälle werden in der Reihenfolge ihrer Prioritäten behandelt.
Beachte dass Prioritäten sich auch ändern können (falls sich der Zustand eines Patienten verschlechtert) !

Szenario: Queue auf einem Clusterrechner
– Jobs kommen unregelmäßig und andauernd an.
Normalerweise wird die Reihenfolge als Warteschlange organisiert. Manchmal gibt es aber Jobs mit höherer Priorität, die vorgezogen werden.

Prioritätswarteschlangen

Was wollen wir ? Formal:

- Datenstruktur, um eine dynamische Menge von Elementen zu verwalten.
- Jedes Element hat einen Prioritätswert.
- Operationen sind: Einfügen eines Elements, Erhöhen des Prioriätswertes, evtl. Erniedrigen des Prioritätswertes, Streichen das Element mit dem höchsten Prioritätswert

Wie realisieren wir diese Datenstruktur ?

Realisierung

als ungeordnetes Array oder Liste

- Einfügen geht in $O(1)$ ans Ende der Liste/Arrays
- Streichen des Max-Elements geht in $O(n)$ (Durchlaufen)
- DecreaseKey und IncreaseKey geht in $O(1)$, sofern die Position des Elements bekannt ist

Realisierung

als geordnetes Array oder Liste

- Erstmal Sortieren des Arrays in $O(n \log n)$ (siehe später)
- Löschen des Max-Elements in $O(1)$, steht am Anfang/Ende des Arrays
- Einfügen muss an die richtige Stelle: In Liste mit Durchlaufen in $O(n)$
  In Array mit binärer Suche in $O(\log n)$, dann aber alle restl. Einträge um 1 Stelle verschieben ($O(n)$).
- DecreaseKey und IncreaseKey durch Verschieben des Elements an die richtige Stelle. $O(n)$.

Realisierung

als Heap

- Erstmal HeapAufbau in $O(n)$
- Löschen des Max-Elements in $O(1)$, Reparieren in $O(\log n)$
- Einfügen in $O(\log n)$
- DecreaseKey und IncreaseKey in. $O(\log n)$.

Zeigt Vorteile des Heaps

Ist irgendwie zwischen sortiertem und ungeordnetem Array
Alle Operationen in $O(\log n)$, also bisschen teurer als $O(1)$,
aber keine richtig teuer ($O(n)$)

II.5 Hashing

Hashing: Szenario

- Haben einen Online Shop und verwalten die Daten der Kunden, die in den letzten 20 Minuten auf unsere Seiten geklickt haben
- Kunden werden über IP Adressen identifiziert
- Menge an Personen, die online sind, ändert sich dauernd.
- Wollen schnellen Zugriff auf die Kundendaten

Erster Versuch:

- Allokiere ein Array für alle möglichen IP Adressen (je 32 Bits)
- Fülle Einträge für die Kunden, die gerade aktiv sind
- Dann geht der Zugriff in $O(1)$

Nachteil: Anzahl aktiver Kunden ist VIEL kleiner als die Zahl aller möglichen IP Adressen. Speicherplatzverschwendung !!

Verwaltung der aktiven Kunden

2. Versuch: Erzeuge Liste der IP Adressen
– Effizient bzgl. Speicherplatz. Aber lange Zugriffszeit für einzelnen Kunden

3. Versuch: IP Adressen hintereinander in ein Array
Aber: Welche Adresse steht wo? Zugriff auch ganz langsam.

Lösung: Hashing

Hashing: 2. Szenario

Problem aus dem Programmierprojekt 'KI für Halma':

Haben ca. 100 Felder und 10 Spielfiguren. Speichere alle erreichbaren Stellungen ab.

1. das sind viele
2. wollen schnellen Zugriff

Methode der Wahl: Hashing

Hashing: Idee

Benutzen ein kleines, aber nicht zu kleines Array (Ansatz 3)

Aber anstatt fortlaufend Einträge zu schreiben, benutzen wir Funktion, die uns sagt, wo das Element steht. (HASH Funktion)

Beachte: Es können Fehler auftreten, nämlich wenn zwei Elemente den gleichen Hashfunktionswert haben.

Also: Mache das Array nicht zu klein ! Wähle Hashfunktion geschickt, so dass Fehler möglichst nicht oft auftreten

Hashing

Gegeben Universum $U = [0, ..., N-1]$ (alle möglichen IP Adr.)
$S \subset U$ sei die zu verwaltende Menge. $|S| = n$ (Kunden online)

Operationen: Zugriff(a, S); Einfügen(a, S), Streichen(a, S)

- Hashtafel $T[0, ..., m-1]$ der Größe m
- Hashfunktion $h : U \to [0, ..., m-1]$ bildet Elemente $a \in U$ auf Tafel $T[h(a)]$ ab.
- $\frac{n}{m} = \beta$ heißt Belegungsfaktor

Bsp: $N = 50, m = 3, S = \{2, 21\}, \quad h(x) = x \bmod 3$

Problem:
Kollisionen! Was tun falls $x \neq y$ aber $h(x) = h(y)$?

Hashing mit Verkettung

Jedes Element $T[i]$ der Hashtafel $(0 \leq i \leq m-1)$ besteht aus einer Liste, die alle $x \in S$ mit $h(x) = i$ enthält.

worst case: alle $x \in S$ in einer Liste $\rightarrow O(n)$ Laufzeit

Mittlere Analyse liefert was viel!!! besseres:

Wahrscheinlichkeitsannahmen:

1. $h(x)$ wird in $O(1)$ berechnet.
2. h verteilt Elemente von U gleichmäßig auf T, also $|h^{-1}(i)| = \frac{|U|}{m}$ für alle $0 \leq i \leq m-1$.
3. Für n Operationen auf den Elementen gilt: $prob(x = \text{Elem. der } j\text{-ten Operat}) = \frac{1}{N}$ für $x \in U$ (Operationen unabhängig, gleichverteilt)

Für x_k Arg, der k-ten Oper.: $prob(h(x_k) = i) = \frac{1}{m}$ (Hashwerte gleichverteilt)

Hashing mit Verkettung

Def. $\delta_h(x, y) = 1$ falls $h(x) = h(y)$ für $x \neq y$ und
$= 0$ sonst

$\delta_h(x, S) = \sum_{y \in S} \delta_h(x, y)$
entspricht $1 +$ Länge der Liste in $T[h(x)]$

$\Rightarrow$ Kosten der Operation $XYZ(x) = O(1 + \delta_h(x, S))$

Satz:
Erwartete Kosten von $XYZ(x)$ sind $1 + \beta = 1 + \frac{n}{m}$

Erwartete Kosten bei Hashing mit Verkettung

Beweis:
Sei $h(x) = i$ und sei p_{ik} die Wahrsch., dass Liste i genau k Elemente enthält.
Dann ist $p_{ik} = \binom{n}{k}(\frac{1}{m})^k(1-\frac{1}{m})^{n-k}$.
Die erwarteten Kosten sind dann

$$\begin{aligned}\sum_k p_{ik}(1+k) &= \sum_k p_{ik} + \sum_k k\binom{n}{k}(\tfrac{1}{m})^k(1-\tfrac{1}{m})^{n-k} \\ &= \dots \\ &= 1 + \tfrac{n}{m}(\tfrac{1}{m} + (1-\tfrac{1}{m}))^{n-1} \\ &= 1 + \tfrac{n}{m} = 1+\beta\end{aligned}$$

□

Also: Im Mittel sind die Listen alle kurz.

Wie lang ist die längste Liste (erwartet)?

Erwartete Länge der längsten Liste:

Idee: Wir wählen S zufällig aus U. Also
$prob(h(x_k) = i) = \frac{1}{m}$ für $k \in S, i \in [0, ..., m-1]$

Schritte:

1. Schätze ab, ob Liste i die Länge $\geq j$ hat.
2. Schätze ab, ob die längste Liste die Länge $\geq j$ hat
3. Bilde Erwartungswert der längsten Liste L.
4. Schätze ihn ab:
   $\Rightarrow E(L) = O\left(\frac{\log n}{\log \log n}\right)$.

Wahl von β: Wir wollen Laufzeit $O(1)$!

Wahl von $\beta = n/m$

- Ist $\beta \leq 1$, ist die Laufzeit gut!
- Ist $\beta \geq 1/4$, ist die Platzausnutzung gut!
- Alles andere ist schlecht!
  $\rightarrow$ schlechte Laufzeit oder schlechte Platzausnutzung

Problem: Durch Einfügen/Streichen verändert sich β.
Vielleicht schnell zu klein oder zu gross.

Rehashen!!!

Rehashing

- Benutze Folge von Hashtafeln $T_0, T_1, \ldots$ der Größe $m, 2m, 4m, \ldots$.
- Ist $\beta = 1$ bei Tafel T_i der Größe $2^i m$, kopiere Elemente in Tafel T_{i+1} der doppelten Größe
  $\rightarrow \beta$ nun $= 1/2$
- Ist $\beta \leq 1/4$, kopiere Elemente von T_i in T_{i-1}
  $\rightarrow \beta$ nun $= 1/2$

Beachte: Nach Rehashen hat man eine Zeitlang Ruhe. Rehashen ist teuer! Danach kommen aber viele billige Operationen
$\rightarrow$ amortisierte Analyse!
$\rightarrow$ n Operationen gehen erwartet in Zeit $O(n)$

Rehashing genauer: Nur Einfügungen

Nehmen an, wir sind grad von T_{i-1} zu T_i übergegangen und gehen nun von Tafel T_i der Größe $2^i m$ zu Tafel T_{i+1}:
Also T_i hat $\beta = 1/2$.
Dazu fügen wir $m2^i/2$ neue Elemente ein, bevor von $\beta = 1/2$ auf $\beta = 1$ steigt.

Also kostet Rehashen $O(m2^{i+1})$ (teuer), aber dazwischen gibt es immer $m2^{i-1} = O(m2^i)$ Operationen, die $O(1)$ kosten, d.h.

insgesamt kosten diese $m2^{i-1} + 1$ Operationen bis inklusive der nächsten Rehashing-Operation dann $O(m2^i)$, also Linearzeit.

Rehashing hat Laufzeit maximal verdoppelt/vervierfacht.
Das war amortisierte Analyse (siehe später).

Offene Adressierung

Ziel: Alle Tafeleinträge werden höchstens einmal belegt, keine Listen!

Ist beim Einfügen ein Feld schon belegt, so probiere ein anderes:
$\rightarrow$ Brauchen also eine Folge von Hashfunktionen
$\rightarrow$ Funktioniert die erste nicht, nimm die zweite!

Beispiele:

- $h_i = (h(x) + i) \bmod m$
  Linear probing: Probiere einfach das nächste Feld
- $h_i(x) = (h(x) + c_1 i + c_2 i^2) \bmod m$
  Quadratic probing

Beispiel zu Linear Probing

Linear Probing

Prinzip: Für Element k, berechne Hashwert $h(k)$. Versuche k dort abzulegen. Falls schon belegt, versuche $h(k)+1, h(k)+2, \ldots$ bis ein freies Feld gefunden ist.

Invariante: Falls k auf Feld $h(k)+x$ gespeichert ist, sind alle Felder $h(k), h(k)+1, \ldots h(k)+x$ belegt.

Zugriff auf Element k: Springe zu Position $h(k)$, laufe los, bis entweder k gefunden oder freies Feld gefunden (dann gibt es k nicht).

Wie geht Löschen ? ... ohne Invariante zu verletzen

Linear Probing - Löschen eines Elements

1. Lösung: Ersetze das Element durch Spezialsymbol, und behandele das Feld jeweils, also ob es belegt sei.
Nachteil: Hashtafel wird immer voller !
Ausweg: Reorganisiere die gesamte Menge von Zeit zu Zeit ohne die Spezialsymbole

2. Lösung: Nach dem Löschen teste die Einträge nach dem gelöschten Element k. Wenn sie nicht korrekt stehen (Test mit Hashwert $h(k)$), dann schiebe sie um 1 nach links.
Falls ein Eintrag k' gefunden ist, der an der richtigen Stelle steht, also an $h(k')$. suche weiter, bis entweder Lücke gefunden ist (STOP), oder ein Eintrag k'', dessen $h(k'') < h(k')$. In diesem Fall schiebe k'' in die Lücke bei $h(k') - 1$. (WEITER)

Satz (Offene Adressierung)
Unter der Annahme, dass alle Hashwerte gleichwahrscheinlich vorkommen, und dass β der Belegungsfaktor der Hashtafel angibt, gilt dass Einfügen/Streichen höchstens $1/(1-\beta)$ Schritte kosten im Erwartungswert..

Ein paar Werte eingesetzt:
$\beta = 0.5$ bedeutet $1/(1-\beta) = 2$
$\beta = 0.9$ bedeutet $1/(1-\beta) = 10$

Merke: Ist die Tashtafel relativ leer, ist offene Adressierung ne gute Idee. Falls zu voll, geht die Performanz in die Knie !

Schlussbemerkungen:

Es gibt keine Hashfunktion, die für alle Mengen von Schlüsseln am besten funktioniert !

Wenn man die Menge der Schlüssel kennt:
1. Universelles Hashing: Für eine gegebene Datenmenge funktionieren die 'meisten' Hashfunktionen gut. Wähle eine zufällig !
2. Perfektes Hashing: Gegeben Menge von festen Schlüsseln, so kann man eine kollisionsfreie (injektve) Hashfunktion konstruieren.

Siehe dazu weiterführende VL über Algorithmen oder Bücher!