Uppgift 10

Simon Moradbakti

HT 2024

Introduktion

I den här uppgiften var syftet att visa det mest naturliga sättet att organisera en datamängd som ska vara tillgänglig via en nyckel. Vi kommer först visa en lösning som inte är speciellt effektiv, och sedan en variation som är betydligt bättre och ger oss en mer effektiv tidskomplexitet.

A table of zip codes

I denna deluppgift fick man en lista med en rad svenska postnummer i en CSV (comma separated value) fil i Excel. Varje enskilt postnummer blev en post i vår framtida array. Detta kunde helt enkelt göras genom att läsa av filen med den givna koden i uppgiften.

Vi skrev en metod som kallas lookup, precis som uppgiften vill att vi ska göra. Den fungerar som en standard sök algoritm, som fungerar på det sättet att den söker av hela listan tills den hittar värdet den söker, alltså O(n)! Lookup metoden implementerades såhär:

```
public boolean lookUp(String entry) {
  for (int i = 0; i < data.length; i++) {
    if (data[i] == null) break;
    if (data[i].code.equals(entry)) return true;
  }
  return false;
}
```

Uppgiften ville även att man skulle köra benchmarks för att jämföra våran linjära och binära sökmetod och se vad som är bäst. Datan som skulle matas in var den den första och den sista postkoden i vår CSV fil. Det var detta som gjordes och resultatet blev som följande:

Elementen	Lookup	Binärt
111 15	249	999
984 99	196000	875

Table 1: Benchmark, lookup mot binary sökning. (Zip är string).

Kollar man på resultatet av dessa benchmarks så ser man att linjär (Lookup) sökning är det bästa valet för elementet 111 15 då det hittas direkt. Den binära motsvarigheten börjar som vi vet mitt i listan och läser av från mitten ut åt, därav den längre tiden för att hitta det första elementet. Men om man istället väljer det sista elementet 984 99 blir det binära alternativet mycket bättre så den bara söker av från mitten ut, medans vårt linjära alternativ måste gå igenom hela listan. I praktiken betyder det att våran binära metod ger oss tidskomplexiteten O(nlogn). När vi istället gjorde om postnumren från strängar till heltal, blev båda algoritmerna betydligt effektivare.

Elementen	Lookup	Binärt
111 15	124	249
984 99	62000	124

Table 2: Benchmark, lookup mot binary sökning. (Zip är integer).

Detta då vi slipper konveratera våra stringar till integers vilket sparar tid.

Use key as index

För att göra det ännu snabbare kan vi använda postnumren som index i vår array. Eftersom det största postnumret är 99 999 kan vi skapa en array med 100 000 element och därför bara direkt använda postnumret som index. Vi kan prova detta och se vad vi får ut:

Elementen	Lookup	Binärt
111 15	0	124
984 99	0	124

Table 3: Benchmark, lookup mot binary sökning. (Zip är integer key är index).

Som man kan se så ger vår Lookup oss ett konstant värde som är 0. Det har egentligen ingen betydelse vad detta värde är sålänge det är en konstant. Detta ger tidskomplexiteten O(1).

Size matters

En nackdel med den implementation vi har gjort hittills är att vi har allokerat mycket minnesutrymme, men vi använder bara cirka 10 procent av det. Vi har

alltså en array med 100 000 element, men använder bara runt 10 000 av dem. I vissa små projekt kan detta vara okej att göra, eftersom det bara handlar om några få bytes. Men i större projekt skulle det vara ett stort problem.

Lösningen på detta är att omvandla nyckeln till ett index i en mindre array. En funktion som gör detta kallas för en hashfunktion. En enkel hashfunktion tar nyckeln och delar den med ett värde n (modulo), och hoppas att indexen blir relativt unika. Om två nycklar hamnar på samma index kallas det en kollision, något vi måste hantera senare. Ju färre kollisioner, desto bättre. Vi kan nu göra benchmarks och se hur det går:

modulo n	1	2	3	4	5	6	7	8	9
10000	4465	2414	1285	740	406	203	104	48	9
12345	7145	2149	345	34	1	50	0	0	0
13513	6234	1765	501	87	16	9	3	1	0
13600	6120	1800	498	90	19	5	2	0	0
14000	6890	1598	423	65	12	4	1	0	0

Table 4: Benchmark där 1-10 är antal kollisioner av varje typ

man kan komma fram till att primtal som modulo n ger färre kollioner. Gör vi dom större får vi färre kollisioner men tar upp mer plats. Primtal är mer effektiva då de endast kan delas på sig själv och 1.

handling collisions

För att hantera kollisioner kan vi använda ”buckets”, där varje bucket rymmer element med samma hashvärde. Buckets behöver inte vara stora – ett element räcker i början, vilket ger minimal overhead. Om en kollision inträffar, allokeras en större bucket för att rymma fler element, vilket bara sker när nya poster läggs till.

Implementera detta genom att använda en array av buckets som är tom från början och skapas vid behov. Vid lookup måste vi alltid kontrollera att postnumret i bucketen verkligen är det rätta, särskilt om flera postnummer delar samma bucket.

```
public void insert(Node nodeToInsert) {
    int index = nodeToInsert.code % size;
    Node current = data[index];
    if (current == null) {
        data[index] = nodeToInsert;
    } else {
        while (current.next != null) {
            current = current.next;
        }
        current.next = nodeToInsert;
    }
```

slightly better?

I denna deluppgift skulle vi förbättra vår existerande bucket implementation genom att starta med vårt hashade index och flytta oss fram i vår array för att hitta rätt startpunkt. Detta skulle sedan jämföras med orginalet. Detta gjordes och resultatet av detta blev följande:

modulo n	1	2	3	4	5	6	7	8	9
10000	2980	1600	857	500	270	140	65	30	6
12345	4760	1400	230	23	0	32	0	0	0
13513	4200	1200	334	58	11	6	1	0	0
13600	4080	1200	334	60	13	3	0	0	0
14000	4600	1070	282	43	8	1	0	0	0

Table 5: Benchmark, slightly better

Som man kan se gav detta viss bättre resultat. Jämförs detta med det orginella resultatet så har vi en förbättring på flera tusen kollisioner i vissa fall. Det går därför att dra slutsatsen att detta är bättre.