Hash tabeller

Mattias Sandberg

18 Okt 2022

Introduktion

I denna uppgift som kommer jag att gå igenom det mest naturliga sättet att organisera en uppsättning av element som är tillgängliga med en viss nyckel. Det finns både extremt effektiva hash algoritmer och lite långsammare och det är en avvägning som har att göra med hur mycket minne man är villig att allokera för algoritmen. I denna uppgift så ska vi försöka hitta en implementation som är såväl effektiv och sparsam för datorns minne.

En tabell med postnummer

I detta exempel så kommer jag att använda en tabell med svenska postkoder. Original filen med informationen ligger som CSV format och vi kommer att läsa av filen och överföra varje element till ett inträde i array. varje inträde innehåller postkoden, ortens namn samt populationen. Jag definierar sedan en nod klass med följande egenskaper (String code, String name, Integer pop). Sedan så ska jag skriva en linjär ”lookup” metod som använder en linjär sökning genom alla postkoder sökandes efter ett specifikt inträde. Nedan kommer koden som utför denna process:

```
public String linear(String zip){
  for(int p = 0; p < data.length; p++){
    if(zip.equals((data[p]).code))
      return(data[p]).name;
  }
  return null;
}
```

Eftersom postkoderna i filen är i ordning så kan jag skriva en ”binary search” metod som utför samma uppgift. Jag konstruerar sedan en benchmark som söker efter ”111 15” och ”994 99” och jämför ”linear search” med ”binary search. Nedan kommer en bit av ”binary search” koden och sedan benchmarks:

```
public String binary(String zip){
  int min = 0;
  int max = maxi;

  while (true){
  int indx = (min + max)/2;

  int comp = zip.compareTo(data[indx].code);
  if (comp == 0){
    return data[indx].name;
  }
  if(comp > 0 && indx < max){
      min = indx + 1;
      continue;
```

Operation/Stad	Postnummer	Exekveringstid (nanosekunder)
Linear Stockholm	111 15	83
Linear Pajala	984 99	39567
Binary Stockholm	111 15	226
Binary Pajala	984 99	263

Table 1: Tabell över benchmark från String (Linear and binary).

Eftersom vi redan vet att alla postnummer är siffror så kan vi redan från början konvertera dom till integers. Genom att skapa ett nytt Zip program där jag ändrar noden till att hålla integers som kod så är detta möjligt. Om jag nu utför samma benchmark med den nya implementationen så ser jag att exekveringstiden är snabbare och därmed mindre kostsam tidsmässigt vilket är vårt mål. Dock så blev Binary Pajala något långsammare vilket var underligare och värt att utforska mer, men benchmarken är inte alltid exakta utan ger en grov uppfattning om exekveringstider. I båda fallen är tidskomplexiteten Ordo O(1). Nedan kommer Resultatet från ”Integers”

Operation/Stad	Postnummer	Exekveringstid (nanosekunder)
Linear Stockholm	111 15	67
Linear Pajala	984 99	14756
Binary Stockholm	111 15	187
Binary Pajala	984 99	336

Table 2: Tabell över benchmark från Integer (Linear and binary).

Använd nyckel som index

Om vi har en zip kod som nyckel och en nyckel som en integer, så kan vi använda integern som ett index i en array. Eftersom den högsta möjliga nyckeln är 99999 så konstruerar jag en array som är hundra tusen element stor och sedan använder jag nyckeln som ett index. Jag blir tvungen att utöka storleken på data arrayen samt processen att fylla arrayen och sen implementerar jag en "lookup" metod och benchmarkar programmet. Resultatet som representeras nedan visar att denna implementation är substantiellt snabbare än vad "binary search" är och därmed en givande metod:

Operation/Stad	**Postnummer**	**Exekveringstid (nanosekunder)**
Lookup Stockholm	111 15	54
Lookup Pajala	984 99	55

Table 3: Tabell över benchmark från Integer (Linear and binary).

Storlek spelar roll

Det finns dock en nackdel med implementationen som jag nu har konstruerat och det är att arrayen är 90 procent tom. Anledningen till detta är eftersom jag har en array som är hundra tusen element stor men det finns bara mindre än tio tusen postnummer. Detta är i vårt fall så är storleken relativt liten så det spelar inte lika stor roll, men vid större arrayer så blir detta problematiskt. Lösningen till detta problem är att omvandla original nyckeln till ett index i en mindre array. Jag konstruerar en funktion som tar zip kodnyckeln och returnerar ett index mellan 0 och 10000. Funktionen som omvandlar en nyckel till ett index kallas för hash funktion och är väldigt tidseffektiva vilket är elementärt då implementationen görs för att spara tid och utrymme. Efter att jag har beräknat nyckelns modulo så kan vi får kollisioner om två nycklar som mappar till samma index. Kollisioner är något som vi kan och kommer att lösa då det ej är ett önskvärt resultat. Jag ska nu göra ett experiment där programmet läser av alla postnummer och går igenom dom och skapar ett index modulo m för några värden på m som tex 10000 och 12345 och räknar antalet kollisioner och den genomsnittliga kollision träffen:

Kollision Typ	m Storlek	Antal Kollisioner
Max Kollision	10000	9
Genomsnittlig Kollision	10000	0.97
Max Kollision	12345	5
Genomsnittlig Kollision	12345	0.78

Table 4: Tabell över experiment från hash funktion.

När man jobbar med hashfunktioner som vi i denna uppgift gör så är det alltid en avvägning mellan storleken på arrayen, dvs det maximala indexet och antalet kollisioner. Detta beror på att ju större arrayen är desto färre kollisioner kommer att inträffa men i stället så kommer mycket utrymme och minne att slösas. Denna avvägning är hårfin och enligt mina mätvärden ovan ser vi att denna information stämmer, dvs att desto större array desto färre kollisioner generellt sätt.

Hantera kollisioner / Något bättre

En hash funktion med inte allt för många kollisioner går att hantera med hjälp av en array med "buckets". Varje "bucket" håller en liten uppsättning av element som har samma hash värde. Våran "bucket" ska ha en standardstorlek på ett element eftersom då kräver alla element som läggs till utan kollision endast en extra referens. I denna deluppgift så implementerar jag en version av det befintliga programmet som använder en array med "buckets". Arrayen är initialt tom och endast när en ett element läggs till allokerar vi minnet i den minimala "bucket". Det är även i denna implementation metoden "lookup" som kollar om elementet verkligen har rätt postnummer eller hittar elementet som har rätt postnummer och det finns flera stycken i "bucket". Även om det bara skulle vara ett element i "bucket" så kommer hash funktionen att säkerställa att det är det önskvärda elementet.

Det finns dock än ännu effektivare implementation av "bucket" som går ut på att använda arrayen i säg utan en inriktning för att separera "buckets". Idén är att börja med den hashade indexen och sedan gå framåt i arrayen för att hitta rätt inträde. Vår "lookup" metod kommer alltså att stanna så fort den hittar en tom plats i arrayen. Detta förutsätter att arrayen är tillräckligt stor, tex dubbelt så stor än den allra minsta möjliga. Problematiken om arrayen skulle vara för liten är att flera hundra element skulle behövas undersökas vilket skulle försämra prestandan ordentligt. Nedan kommer först kodsektioner från metoderna "lookup" och "Compares" och sedan en tabell med statistik över exekveringstid samt hur många element

som programmet behöver kolla på innan det hittar det rätta för array storlekarna 1811 och 7919 (både primtal). Enligt tabellen nedan så är denna implementation synnerligen effektivare än vad tex linear ”lookup” är:

```
public String lookup(Integer key){
  Integer index = key % mod;
  Node next = data[index];

  while (next != null){
    if(key.equals(next.code)){
      return next.name;
      }
    next = next.nexter;
  }
  return null;
}
public int compares (Integer key){
  Integer index = key % mod;
  Node next = data[index];
  int c = 0;

  while (next != null){
        c++;
        if(key.equals(next.code)){
          return c;
        }
        next = next.nexter;
  }
  return 0;
}
```

Operation / Stad	Array	Antal element	Exekveringstid (ns)
Stockholm	1811	1	35
Lovikka	1811	7	568
Gällivare	1811	4	104
Falun	1811	8	353
Stockholm	7919	1	144
Lovikka	7919	1	97
Gällivare	7919	1	67
Falun	7919	2	347

Table 5: Tabell över experiment från hash funktion med ”Bucket”.