Hashtabeller

viktor björkén

October 2022

1 Introduktion

I denna rapport ska vi lära och skriva om olika typer av så kallade hashtabeller. Hashtabeller Är ett sätt som vi använder oss utav när vi har element med en viss nyckel. Vi kommer att skapa olika impplementationer och jämföra med andra alogritimer som vi har skrivit om i tidigare rapoorter.

2 Postnummer

Vi kommer börja med att skapa en metod som gör att vi kan läsa av en csv fil. Denna funktion ska då också ta dessa värden och stoppa in de i en array. Sedan skriver vi en ”lookup” funktion som då söker igenom arrayen efter det postnummer vi är ute efter. Vi kommer att testa skriva dessa metoder på olika sätt, vi kommer börja med en linjär. Det denna då gör är att den börjar i början av arrayen och sedan söker igenom varje värde för att hitta det vi söker. Sist men inte minst kan vi skriva en metod med hjälp av ett binärt sök system, detta kan vi göra då listan med alla postnummer är sorterad.

Vi kan skriva den binära sökning på följande sätt.

```
public String binary(String zip){
int first = 0;
int last;
while (true){
int index = (first + last)/2;
int comp = zip.compareTo(data[index].code);
if (comp == 0){
return data[index].name;
}
if(comp > 0 && index < last){
first = index + 1;
continue;
```

Gör vi sedan ett benchmark på dessa 2 olika implementatoner kan vi se föjande svar.

sök metod	Postnummer	tid (ns)
binär sök	111 15	103
binär sök	984 99	50234
linjär sök	111 15	443
linjär sök	984 99	523

Det vi sedan ska göra är att gör om denna kod så att vi konerverterar våra postnummer till "Integers". Detta kan vi göra då vi vet att alla postnummer endast är nummer. Vi skriver om koden för att det ska fungera med Integers istället och för sedan fram en tabell med de resultat vi får efter våran benchmark

```
Integer code = Integer.valueOf(row[0].replaceAll("\\s",""));
data[i++] = new Node(code, row[1], Integer.valueOf(row[2]))
```

När vi skrivit om denna kod testar vi om våra benchmarks och kan tydligt se att vi har fått en bättre exekveringstid än den tidigare Detta får vi då vi inte behöver lägga tid på att konvertera våra Strings till Integers då vi nu har Integers från första början. Därav den snabbare exekveringstiden. Däremot har vi samma tidskomplexitet som är

$$O(1)$$

sök metod	Postnummer	tid (ns)
binär sök	111 15	90
binär sök	984 99	24042
linjär sök	111 15	295
linjär sök	984 99	534

3 Nyckel i index

I nästa deluppgift är uppgiften att vi sätter våra postnummer från listan som våran "nyckel", sedan använder vi oss av dessa nycklar som index i våran array.

Det vi gör när vi implementerar denna nya metod är att vi först och främst ökar arrayens storlek så vi får plats för med alla postnummers index. Ett problem som vi kommer stöta på är att arrayen inte kommer att fyllas helt därav kommer vi ha tomma "null" platser i arrayen.

Postnummer	tid (ns)
111 15	65
984 99	67

Nyckel som index. nya loopup

Postnummer	tid (ns)
111 15	234
984 99	474

Lookup binär sök med Integers

4 Storlekens roll

för att lösa detta föregånende problem är en lösning är att istället implementera en mindre array där vi gör om våran nyckel till indexet av denna mindre aray.

Vi implementerar denna metod som gör om våran nyckel till index med hjälp av en hash funktion. En sådan kan se ut på följande sätt.

Vi börjar med att ta modulo med värdet på våran nyckel samt ett annat "eget" värde detta kommer ge oss ett nytt unikt mindre värde. Som man kanske kan räkna ut kan detta bidra till att problem uppstår. Dessa problem är när vi då får samma värden av att ta modulo av våra nyckel värden. När detta händer kallas det att vi får en "krock".

Det man får tänka på när man olika hashfunkioer är att får ta i åtanke och försöka hitta en balans mellan storleken på arrayen samt antalet kollisioner. Om vi har en större array kommer vi att få flera kollisioner samt mer uttryme som går förlorad.

Därefter testar vi olika värde som vi tar modulo av och ritar upp en benchmark.

Modulo	Kollisioner (genomsnitt)	Kollisioner (max)
10000	1.4	12
12345	0.9	7
20000	1.3	13

Det vi kan kan se är att det finns en skilland mellan när vi använder oss utav mer "vanliga" jämt emot när vi tar ett modulo värde på t.ex 12345.

Dessa vanliga värden är något högre än vårat 12345 värde. Min hypotes är att detta beror på att när vi tar modulo av ett mer "slumpat" värde får vi ett mer unika index.

5 Hantera kollitioner

Desto mindre "krockar" desto bättre för att handskas med detta problem vill vi få ner dessa kollisioner så mycket vi bara kan. En lösning för detta är att använda oss utav så kallade "buckets"

Dessa "buckets" funkar så att element som har samma hash värde som då har en standad storlek på ett element. Vi börjar med att uppdatera vårat tidigare program med dessa buckets. Vi börjar med en tom array och när vi får in ett värde allocerar vi den minska bucketen.

En viktigt sak att tänka på när skriver denna nya lookup är att vi ser till att vi faktist hittar ett giligt postnummer. Risken finns att vi annars kan hitta ett hashvärde som ger index till ett giligt postnummer även om vi sökte efter ett som ej existerar.

För att göra detta ännu bättre kan vi implementera en annan "bucket" funktion. Denna går då ut på att vi kommer starta med vårat hashade index för att sedan fortsätta i arrayen för att hitta rätt.

Detta är fördelaktigt då när våran lookup funktion hittar ett tomt värde i arraten akn den stanna. Dock kommer detta bestyda att våran array kommer behöva vara större än annars.

Metod/ stad	Array	Element	Tid
Stockholm	1811	1	40
Stockholm (bättre)	1811	1	35
Stockholm	7919	1	240
Stockholm (bättre)	7919	1	190
Uppsala	1811	5	100
Uppsala (bättre)	1811	3	90
Uppsala	7919	1	160
Uppsala (bättre)	7919	1	130