Assignment 8 - Hash tabeller

Jesper Hesselgren

Oktober 20, 2024

Introduktion

I den här rapporten ska vi utforska datastrukturen hashtabeller och undersöka hur de kan användas för att organisera data på ett effektivt sätt. Utgångspunkten är en CSV-fil som innehåller 9675 postnummer i Sverige. Vi kommer att implementera tre olika lösningar för att hantera dessa postnummer: en enkel men ineffektiv lösning, en mycket snabb men minneskrävande lösning, samt en lösning som balanserar både tids- och minnesanvändning, i form av en hashtabell. Vi kommer också att göra en enklare prestandaanalys av varje lösning genom att mäta söktider och antalet kollisioner.

Hash-tabellen

En hashtabell är en datastruktur där data lagras som *nyckel-värde-par*. Nyckeln används för att snabbt hitta och hämta ett värde. Genom en så kallad hash-funktion omvandlas nyckeln till en unik plats i tabellen, vilket gör sökningen snabb. I bästa fall får varje värde en unik nyckel som mappas till en unik plats i tabellen, men om hash-funktionen genererar samma nyckel för två olika värden uppstår en kollision. Detta kan dock hanteras med hjälp av lite olika metoder.

Kodstrukturen

I vår implementation representeras postnummerinformationen genom klassen Zip, som innehåller en array av typen Area. Varje Area objekt i arrayen representerar ett postnummer och lagrar relevant information såsom postkod, områdesnamn och befolkningsstorlek. Klassen Area är en intern klass till Zip och fungerar som en behållare för varje postnummer. Arrayens storlek sätts initialt till 10 000, men vi kommer justera den siffra i de olika lösningar vi implementerar. I de lösningar vi kommer att utforska hämtas data från en CSV-fil med alla postnummer, och genom konstruktorn i Zip-klassen placeras dessa i Area arrayen på lite olika sätt.

```
public class Zip {

Area[] postnr;
int max = 10000;

// Intern klass som innehåller data om vårat postnummer
public class Area {
    String zipCode;
    String name;
    Integer popu;

    public Area(String zip, String name, Integer population) {
        this.zipCode = zip;
        this.name = name;
        this.popu = population;
    }

public Zip(String file) {
    this.postnr = new Area[this.max];
    try (BufferedReader br = new BufferedReader(new FileReader(file))) {
        String line;
        int i = 0;
        while ((line = br.readLine()) != null && i < this.max) {
            String[] row = line.split(",");
            postnr[i++] = new Area(row[0], row[1], Integer.valueOf(row[2]));
        }
        this.max = i;
    } catch (Exception e) {
        System.out.println("File " + file + " not found");
    }
}
}
```

En ineffektiv lösning

Genom vetskapen att vår CSV-fil innehåller 9675 postnummer som är sorterade i stigande ordning, kan vi enkelt läsa in filen och placera postnumren i en array där det lägsta postnumret placeras först och det högsta sist. Med denna struktur kan vi använda två olika sökmetoder, linjärsökning och binärsökning. Linjärsökning går igenom alla poster en i taget, vilket är ineffektivt för stora datamängder. Binärsökning, som delar upp sökningen i halvor, är betydligt mer effektiv än linjärsökning, men som vi senare kommer att se i våra implementationer, är denna metod ändå inte optimal i detta sammanhang.

En effektiv men minneskrävande lösning

Ett alternativt sätt att implementera en lösning är att utnyttja det faktum att våra postnummer sträcker sig från 11115 till 98499 och är unika. Genom att utöka storleken på vår Area-array till 100 000 och placera varje postnummer på samma index som sitt numeriska värde, kan vi uppnå en söktid med tidskomplexiteten *O(1)* för varje element i datastrukturen. Detta innebär att vi kan slå upp ett postnummer direkt, utan att behöva söka igenom arrayen.

Denna lösning kommer dock med en relativt stor kostnad i form av minnesanvändning. Eftersom vi endast har 9675 postnummer, men använder en array med storleken 100 000, utnyttjar vi faktiskt mindre än 10 procent av det minne som arrayen kräver. Detta gör lösningen ineffektiv ur ett minnesperspektiv, trots dess snabba åtkomsttider.

Hash-tabellen

Den mest effektiva lösningen på vårt problem är att använda en hash-tabell. Genom en hash-funktion kan vi mappa varje postnummer till ett index, vilket ger snabba söktider och gör att vi kan använda minnet på ett effektivt sätt i förhållande till mängden data. Däremot måste vi hantera fallet när hash-funktionen ger samma index för två olika postnummer, vilket leder till en kollision. För att lösa detta justerar vi vår datastruktur genom att implementera en internklass Bucket, som hanterar dessa kollisioner genom att länka samman poster som hamnar på samma index.

I vår implementation består hashtabellen av en array av typen Bucket. Varje Bucket innehåller ett Area-objekt som lagrar information om postnumret, såsom själva postkoden, namnet på området och befolkningsantalet. Om en kollision uppstår, används en länkad lista där varje Bucket kan peka på nästa Bucket. På så sätt kan vi lagra flera poster på samma index utan att förlora någon data.

```
public class Zipper {
    Bucket[] postnr;
    int max = 13513;

    //Internklass som hanterar postkods datan
    public class Area {
        Integer zipCode;
        String name;
        Integer popu;
```

```
        public Area(Integer zip, String name, Integer population) {
            this.zipCode = zip;
            this.name = name;
            this.popu = population;
        }
    }
    //Internklass som används för att hanter kollisioner
    public class Bucket {
        Area area;
        Bucket next; // Hanterar kollisioner

        public Bucket(Area area) {
            this.area = area;
            this.next = null;
        }
    }
```

I vår implementation använder vi en enkel hash-funktion som tar ett postnummer och applicerar modulo-operationen med ett tal som representerar arrayens storlek. Efter att ha testat olika värden valde vi storleken 13 513, då detta ger en relativt liten mängd kollisioner. Hash-funktionen genererar ett index där varje postnummer lagras i en "bucket", vilket gör att vi snabbt kan komma åt rätt postnummer samtidigt som vi effektivt hanterar eventuella kollisioner.

Denna struktur ger oss en genomsnittlig tidskomplexitet på *O(1)* för både insättningar och sökningar. Hash-tabellen erbjuder en optimal lösning genom att vara både tids- och minneseffektiv.

Slutsats och diskussion

I denna rapport har vi undersökt tre olika lösningar för att hantera och söka postnummer med hjälp av olika datastrukturer, däribland en enkel array, en effektiv men minneskrävande lösning, och tillsist en hash-tabell. Genom våra implementeringar har vi kunnat jämföra prestanda i form av söktider och minnesanvändning. Den sista lösningen, i form av en hash-tabell visade sig vara den mest optimala, med snabba söktider och effektiv hantering av minne genom användning av en länkad lista för att lösa kollisioner. Trots vissa minnesmässiga kompromisser ger hash-tabellen en balanserad och skalbar lösning som klarar av att hantera stora datamängder effektivt.