HashTabell

Hiba Qutbuddin Habib

Oktober 2022

Introduktion

I denna uppgiften är syftet att man ska lära sig att organisera en uppsättning data som bör vara tillgänglig med en given nyckel. I denna uppgiften används en csv fil, där den innehåller ett antal postnummer, denna filen kommer att användas som data grund. För att uppnå uppgiftens syfte så kommer vi att tillämpa en antal olika metoder, såsom linjär sökning, binärsökning samt sökning med en given nyckel genom att tillämpa en hash funktion.

En tabell med postnummer

I denna delen kommer man att ha stor användning av csv filen med de olika postnummer. Då en "Zip" metoden implementeras för att kunna läsa av csv filen samt att infoga de olika postnummer i en vektor. Vi implementerar även två andra metoder " Linear" samt "Binary", dessa metoder kommer att ha i funktion att söka efter en specifik postnummer i en vektor. Metoden "Linear" kommer att söka linjärt genom en vektor efter en specifik postnummer medan metoden "Binary" kommer att tillämpa binär sökning i vektor.

För att kunna få en uppfattning av tiden, så kommer vi först att deklarerar postnummer av typen String och vi mäter tiden vid sökning efter två specifik postnummer nämlingen "11115" samt "98499". Sedan kommer datatypen av postnummer att ändras till Integer och då tid mätes återigen vid sökning efter sammma givna postnummer.

Resultat

Postnummer	**Linär**	**Binär**
111 15	53 ns	296 ns
984 99	28471 ns	348 ns

Table 1: Tid i nanosekund, postnummer av typen String

Postnummer	Linär	Binär
111 15	117 ns	104 ns
984 99	9312 ns	102 ns

Table 2: Tid i nanosekund, postnummer av typen Integer

Resultatet ovan visar tidmätning vid sökning efter två olika postnummer ”11115” samt ”98499” med två olika sökningsmetoder, linjär samt binär sökning. Det som är specifik med dessa två valde postnummer är att den första kommer nämligen att ligga i början av vektor medan den andra kommer att ligga i slutet. Vid sökning efter postnummer med data typen String så krävde det längre tid för den binära söknings algoritmen att hitta den postnummer som ligger nära början av vektor jämfört med den linjär sökningsalgrotimen, då vid sökning efter den postnummer som ligger vid slutet av vektor så vara det mycket snabbare för binära söknings algoritmen att hitta den jämfört med den linjära. Detta tyder att binär sökning är mycket effektivare när det kommer till sökning bland stora mängder, då den csv filen har redan från början ordnade postnummer, så när man söker binärt efter något som kan redan ligga i början av listan kommer algoritmen ändå att dela upp vektor i två dela och jämföra alla söknings steg, medan för den linjär sökning så kommer den direkt att hitta den postnummer eftersom man söker linjärt genom varje index.

När vi ändrade data typen från String till Integer så skede det en stor förbättring när det gäller tidmätning speciellt vid sökning efter den sista postnummer i listan då tiden för den linjär sökning blev ungefär 3 gånger mindre som man ser i tabellen ovan. Denna förbättring kan bero på att det är lättare för dator att arbeta med Integer då String gör att talet är en sekvens av karaktärer vilket gör att programmet kommer första behöva tid för att tolka dessa karaktärer med en viss encodings typ för att sedan kunna jämföra och hitta det som man söker efter.

Att använda nycklar som index

De postnummer kan också användas som nyckel, då varje stad har en specifik postnummer som är också Integer, dessa kan de utnyttjas som olika index i en vektor. När dessa används som index då kan vi implementera en metod ”Lookup” för att hitta de olika städer med hjälp av deras postnummer som är i sin tur index, metoden ser ut enligt:

```
public String lookup (Integer key){
    int index = key %  max;
    Node nxt = data[index];
     while (nxt != null){
        if(key.equals(nxt.code)){
            return nxt.name;}
```

```
            nxt = nxt.next;}
    return null;}
```

Metoden "Lookup" tar emot en postnummer som en nyckel, den användes denna nyckel i en modulos operation, då "max" är längden. Värdet av modulos operation används som index då en pekare kommer att peka åt just denna index, sen sker det en jämförelse.

Resultat

Metoden "lookup" används för att söka efter samma använda postnummer ovan "11115" samt "98499". I jämförelse med den binära sökningen ovan får vi följande resultat: Från tabellen ovan ser vi att det tar mycket mindre tid

Postnummer	**Lookup**	**Binär**
111 15	126 ns	104 ns
984 99	41 ns	102 ns

Table 3: Tid i nanosekund, postnummer av typen Integer

för "lookup" att hitta de postnummer som ligger vid slutet jämfört med den binärsäkningen, det eftersom för "lookup" används själv postnummer som index, alltså om vi tar den första postummer 98499 och räkna modulos 100000 som är i sin tur storleken av vekotor som innehåller samtliga postnummer så kommer vi får resultat 98499 och om vi sätter en pekare till den index så är vi redan vi just den index som innehåller den postnummer vi söker efter, däremot med binärsökning tar det längre tid eftersom man behöver dela upp vektor i två halvor som i sin tur kommer också att delas upp och därför tar det nästan dubblet så långt tid för binärsökning algoritm jämfört med metoden "Lookup" när vi söker efter något som ligger vid slutet.

Storleken har betydelse

Implementering av funktionen "lookup" ovan har i fördelan att den är tidsparande, då att hitta data som kan ligga långt ifrån början av vektor var en ganska snabb operation. Dock en nackdel med denna är uttrymmet om man resververar en plats i minnet som är 100000 stor för att sedan man använder endast 10 procent av den så har vi 90 procent av uttrymmet som inte ha kommit till nytta, ett sätt och kunna lösa detta problem på är att använda något som kallas för "Hash funktion". Hask algoritmen är en funktion som i vårt fall tar den givna postnummer och räkna den modulos en viss storlek man reserverar för en vektor, exampelviss 10000. Modulos räkna operationen kommer att ge en index, i denna index placeras då den postnummer som vi utförde operationen med. Denna algortimen är både utrymmes och tids sparande, då ju mindre uttrymme vi söker genom, desto snabbare går det.

Däremot Hask algoritmen har en nackdel och det är att det kan uppstå ett antal kollisioner vid placering i dem olika uträknade index med modulos operation. Då i vårt fall två postnummer eller fler kan få samma index, och problemet blir då att det går inte och placera två olika data i exakt samma plats.

Vi använde hash funktionen och vi räknade antal kollisioner som kan uppstå vid olika storlekar av vektor.

Resultat

m	**Antal kollision**
11000	4651
12145	2545
15000	4138
17555	3321
19000	3247
21111	1294
30000	2407

Table 4: Totala antal kollision för olika värde på m

I tabellen ovan ser vi resultat över total antal kollision för olika storlekar på den reserverade uttrymme. Något man kan tydligt observera är att större uttrymme verkar inte lösa problemet då vid 30000 har det uppstått mer antal kollisioner jämfört med 21111. Något annat kan man observera i tabellen ovan är att jämna tal verkar orsaka mer kollision jämfört med udda.

Hantera kollision

Ett sätt och kunna lösa kollision problem med är att använda något som kallas för "Bucket". Denna metoden kan kallas för kedje, implementering sker genom att först skapa en vektor, vid varje icke tom index så pekar den till huvudet av en länkade lista. Då om flera data skulle få samma index efter ett Hash funktion så kommer de att placeras i den länkade lista där den indexet pekar till, om den data finns inte redan med. I vårt fall för att att hitta en specifik stad så kommer man först tillämapa hash funktionen på den givna postnummer för att sedan hitta den länkade listan som indexet pekar till, den listan kommer då att skannas för att kolla om den postnummer existerar där.

Kan det blir bättre!

Sondering är ytterligare sätt som man kan lösa kollisions problem med. Denna metoden sker på plats alltså i en och samma vektor, då istället för att vi ska

skapa ytterligare flera länkade listor som varje index pekar till, så används samma vektor. För att detta ska fungera så behöver man en vektor som är större än mängd data vi har. Då om man ska lägga till en element i vektor och denna element skapar kollision så kommer lösning vara är att vi kommer söka bland de index med en viss interval efter den första lediga indexet, och när den hittas så placeras den data där. Om vi är vid den sista indexet så kan sökning börjar om från den första indexet. För att vi ska få en bild om hur denna fungerar så kommer vi att söka efter två olika postnummer och samla en viss statistik om hur många steg behöver man förflytta sig för att hitta det man söker efter för olika storlekar på vektor.

Resultat

m	**26 134**	**98 499**
11000	9	382
12145	1	3
15000	9	11
19000	0	47
21111	0	2

Table 5: Antal steg för att hitta de givna postnummer

Tabellen ovan visas antal steg som man behövde förflytta sig för att hitta den givna postnummer. Man kan se i tabellen att ju mindre vektor är desto fler steg behövs det, samt att när vi söker efter postnummer som ligger i början så kommer vi många gången hitta den direkt utan att någon förflyttning behöver ske, man kan också se att även om vi letar efter något som vanligt viss bör ligga i slutet så kan det hände att vi kan hitta den med bara några steg.