Sökning i en sorterad och osorterad array

Adalet Adiljan

9 September 2023

Introduktion

Likt rubrikens lydelse kommer denna rapport att handla om att lära ut principen av datastrukturer och hur detta kan förbättra sökningseffektiviteten. Uppgiftens deluppgifter har haft i syfte att bygga förståelse för varför sorterad data är viktig innan sökoperationer utförs, något som har gett oss en bättre förståelse kring hur man implementerar effektiva sökalgoritmer för sorterade arrayer överlag.

Metoder

Till vår första uppgift har vi börjat att först definiera en sökfunktion för en osorterad array i olika storlek. Sedan skriva en optimering av linjära sökfunktionen för det fall som vi har en sorterad array och mäta tiden det tar för att hitta den matchande element med hjälp av vår benchmark metod som vi har ställt upp. Vår andra typ av sökfunktion som vi behövde implementera heter binärsökning, som implementeras enbart i det fall som arrayen är sorterad, detta då den är som mest effektiv. Vår tredje deluppgift ingick i att implementera en så kallad Even Better metod, som är designad för att kunna hitta dubletter av två sorterade arrayer.

Oordnad array

Till att börja med implementerade jag en sökfunktion kallad 'search unsorted' i den publikt deklarerade klassen 'Linear' som tar emot en oordnad array och en nyckel att söka efter i denna array. En enkel for-loop implementerades för att linjärt söka igenom alla element i arrayen, tills nyckeln hittas eller tills hela arrayen har genomsökts som det därpå returneras true annars false. Vår 'search unsorted' funktion kallas för en linjär typ av sökning vars tidskomplexitet är O(n), där n är antalet element i arrayen. Tillskillnad från den binära sökningen som redan kräver att elementen i arrayen redan är sorterade, är den linjära sökningen lämplig för att hantera osorterad data.

I vår arrayAssignment klass (där vi anropar den linjära och binära klassen) har vi definierat en privat metod kallad randomKeys som tar emot två parametrar av två int-värden. Vi har kallat den för loop respektive n, där loop motsvarar längden av den genererade arrayen och kommer att innehålla loop-antal heltal. En for-loop användes för att tilldela varje index i den tomma arrayen med ett slumpmässigt värde av nycklar med rnd.nextInt keyword på följande vis:

```
private static int[] randomKeys(int loop, int n){
Random rnd = new Random();
int[]indx = new int[loop];
for (int i = 0; i < loop; i++){
  indx[i] = rnd.nextInt(n*5);
}
return indx;
}
```

Sorterad array

För att optimera sökningen i en sorterad array, implementerade vi metoden 'sorted' som genererar en sorterad array utan duplicerade element. Tillskillnad från vår linjära sökning är vår search metod en binär sökalgoritm som är uppbyggd på att söka efter element i en redan sorterad array genom att upprepat dela upp sökområdet till hälften vid varje iteration. Jämfört med vår linjära sökningsmetod ligger tidskomplexiteten för binärsökning på O(log n), där n är antalet element i den sorterade arrayen.

Med den instanserade Random-klassen (Inbyggd Java-klass som tillhör paketet java.util) genererades slumpmässiga sekvens av heltal som ökades kumulativt (stigandes efter addition) för att skapa den sorterade arrayen. Jag skapade en ny sökfunktion kallad search från den publika deklarerade klassen Binary. Funktionen utnyttjade det faktum att arrayen är sorterad med ett villkor som avslutar söket så fort en element i arrayen visade sig vara större än den nyckel som hittades.

Benchmarking

Metoderna för prestandamätningen utfördes i main funktionen inom en yttre loop vars iteration kunde bestämmas fritt i olika storlek. I vår yttre loop anropas de olika sökalgoritmerna flera gånger för olika inmatningsstorlekar. Benchmarking-koden mäter specifikt tiden det tar för varje sökalgoritm att köra med olika inmatningsstorlekar.

Prestandamätningen utfördes generellt sett för sorterad sökning där vi vid implementationen av en förbättrad version fick göra en antagelse av att vi hanterade osorterad data.

Förbättrad sökfunktion

Slutligen hade vi i uppgift att implementera en förbättrad sökfunktion som har i syfte att optimera processen av dupletter i två sorterade arrayer.

```
  public static void evenBetter(int[] array, int[] array1){
  int countDuplicates = 0;
  int i = 0;
  int j = 0;

  while (i < array.length && j < array1.length){
    int key = array[i];
    int index2key = array1[j];
    if(array[i] == array1[j]){
        int countDuplicates++;
        i++;
        j++;
    } else if(key < index2key){
        i++;
    } else {
        j++;
    }
  }
}
```

AntalDupletter fungerar som en räknare för att hålla koll på antalet dubletter som hittas mellan de två arrayerna och definieras utanför while-loopen. Loopen fortsätter så länge i är mindre än längden på den första arrayen och j är mindre än längden på den andra arrayen. Inuti värdet hämtar vi värdet på den första pekaren från den första arrayen och värdet på den andra pekaren från den andra arrayen. Om de är lika ökar vi antalet dubletter med +1 och de två pekarna. Om den första nyckeln är mindre än den andra nyckeln är det osannolikt att vi hittar en dublett i framtiden av den första arrayen och vi går vidare till nästa element. Om denna nyckel (element) är större än den andra elementet, inkrementerar vi j med +1, eftersom att det är osannolikt att vi hittar en dublett i den andra arrayen. Processen fortsätter tills vi når slutet av respektive array, då loopen avslutas.

Resultat

I denna kapitel av rapporten diskuterar vi resultaten som givits för tabell 1, tabell 2 och slutgiltningsvis tabell 3. Tabell 1 visar oss hur lång tid det tar för de sorterade och sorterade fall för search unsorted metoden, där vi

kan se en sämre prestanda för det osorterade fallet, trots att funktionen passar bäst för att hantera osorterade arrayer. Tabell 2 motsvarar prestandan hos den linjära respektive binära funktionen i ett sorterat fall, där vi ser att den binära funktionen påvisar bättre tidsresultat i än den linjära funktionen ända upp till 1 miljon element. Binära sökningen har en tidskomplexitet på O(log(n)) medan den linjära ligger på O(m), vilket förklarar varför den binära funktionen presterar bättre ju större array storlek vi har. En tredje tabellen påvisar hur vår bättre sorteringsmetod har sorterat för större arrayvärden, där vi kan se en signifikant skillnad i söktiden. Då den binära sökmetoden är en effektiv algoritm med en logaritmisk komplexitet kan vi säga att den upp till 64M element kommer den att ständigt dela upp sökområdet till hälften (vid varje iteration) och påvisa en alltmer större prestanda för ju ökade värden.

Even Better

Even Better är precis som vad namnet lyder, bättre alternativ som sökfunktion jämfört med vår linjära- respektive binära sökfunktion vad gäller dess effektivitet och minskade komplexitet vad gäller dess funktionalitet och optimala minneskomplexitet. I ett redan sorterat fall skulle vi jämföra varje element i den första arrayen med element i den andra med ett tidskomplexitet på O(m*n), där m och n representerar längderna på de två arrayerna medan Even Better har O(m+n), vilket gör den betydligt snabbare. Jämfört med den binära sökningen vid dublett-sökning, skulle den kräva flera sökningar (en för varje element) i den första arrayen, vilket motsvarar en tidskomplexitet på O(m*log(n)) där m är längden på 1:a arrayen och n är längden på 2:a andra arrayen. Här jämför vår förbättrade version direkt element i båda arrayerna och har en tidskomplexitet i det totala antalet element. Koden är även mindre komplex och undviker ”onödiga” jämförelser genom att redan peka framåt på nästa element under jämförelsen. Samtidigt har även en fördelaktig minneskomplexitet då den bollar med mindre antal variabler (i vårt fall enbart 2 stycken: en räknare respektive en pekare). Om det är värt att sortera arrayerna innan vi söker efter dubletter beror bland annat på arrayernas storlek och datafördelningen, som vi har märkt. Det är ganska självklart att sorteringen är tidskrävande för ju större arrayer vi har trots vilken sökmetod vi implementerar. Vad gäller tidskomplexitet, så är sorterad data ofta fördelaktigt. Prestandan för sökmetoder vars den ena är mer anpassningsbar än den andra kommer att påvisa en alltmer ökad prestanda vad gäller sökning. Dessutom kräver sorterad data ofta extra minnesuttrymme. De flesta sorteringsalgoritmer är inte anpassade för osorterad data (inte ”in-place”) och kräver en separat kopia av datan för att utföra sorteringen, vilket kan öka minneskomplexiteten. Sammantaget, om vi behöver utföra sökoperationer på datan och om tidskomplexiteten är central, kan det vara värt att sortera datan (dock i bekostnad med ökad

minensanvändning). Om minnesanvändningen är viktig och sökoperationer inte utförs ofta, kan osorterad data vara ett rimligt alternativ. I annat fall, bör vi med effektivare idéer kunna mötas i mitten, där både minnes- och tidskomplexiteten är bra.

array storlek	**linjär sorterad**	**linjär osorterad**
100	0,02	0,02
300	0,05	0,05
900	0,2	0,2
2700	0,4	0,5
8100	1,3	1,3
24300	3,8	4,0
72900	11,6	12,0
218700	36,0	9,4
656100	110,0	115,0
1000000	170,9	171,0

Table 1: motsvarar prestandan i mikrosekunder hos search unsorted funktion i en sorterad respektive osorterad fall.

array storlek	**linjär**	**binär**
100	0,02	0,008
300	0,05	0,1
900	0,2	0,1
2700	0,4	0,3
8100	1,3	0,05
24300	3,9	0,05
72900	11,8	0,06
218700	36,7	0,08
656100	108,0	0,09
1000000	167,4	0,1

Table 2: Prestandan under sorterat fall

array storlek	**linjär**	**binär**	**förbättring**
100	0,02	0,03	0,004
300	0,02	0,01	0,001
900	0,16	0,01	0,003
2700	1,5	0,05	0,01
8100	13,0	0,2	0,02
24300	116,6	0,8	0,06
72900	1050,1	2,7	0,2
218700	9804,5	9,0	0,7

Table 3: Motsvarar vårt resultat när vi tillämpar vår evenBetter metod för sorterade arrayer.