Mittuniversitetet
MID SWEDEN UNIVERSITY

DT046G, DT064G Datastruktuer och algoritmer

Martin Kjellqvist

lab.tex 406 2020-10-23 14:23:33Z martin

Innehåll

1 Introduktion

Hur man söker i datastrukturer beror på hur strukturen är organiserad. Om man exempelvis vill hitta ett speciellt avsnitt i en vänder man sig till innehållsförteckningen och går igen kapitel för kapitel, avsnitt för avsnitt. Om man däremot vill hitta ett speciellt ord i en bok, måste man (förutsatt att det inte finns ett index) leta sida för sida ord för ord, tills man hittar det sökta ordet.

Sökningar i datorprogram kan vara sökningar av allt ifrån komplexa objekt till enkla heltalsvärden. Oavsett vad som söks finns det några grundläggande sätt att göra dessa sökningar. Vilka sättt som är möjliga att tillämpa beror på hur vi kan organisera datat.

Om vi har en oordnad lista av data är en enkel linjärsökning ofta tillräcklig, i c++ är detta implementerat i exempelvis `std::find(first, last,`

`value)`. Om linjärsökningar blir för krävande kan datat organiseras på bättre vis. Om datat är sorterat kan sökningen börja i mitten av mängden, då man jämför det sökta elementet med mittenelementet kan man enkelt halvera sökmängden och rekursivt fortsätta med sökningen. Detta är i c++ implementerat i `std::binary_search` och `std::lower_bound`. Om datat kan omorganiseras görs detta med fördel till antingen balanserade sökträd (BST) eller uppslagstabeller. C++ har där `std::set/std::map` respektive `std::unordered_set/std::unordered_map`.

2 Syfte

Vi kommer i labben att i detalj undersöka hur olika datastrukturer erbjuder möjligheter att söka bland dess element. Vi kommer att använda heltalselement i i datastrukturerna och sökningen görs på heltalselement.

Undersökningen består i att samla och sammanställa data från tidsmätningar för sökning av olika sekvenser för olika sökmetoder.

En beskrivning av sökalgoritmerna finner du i kurslitteraturen eller i referenserna.

Som nämnts finns det olika sökmetoder implementerade i standardbiblioteket. Under normala omständigheter är det troligast att du vill använda någon av de metoder som finns i standardbiblioteket, inte skriva något eget. Vi använder egna implementationer här så att vi är säkra på att vi faktiskt undersöker de algoritmer vi ska.

3 Uppgift

Du skall jämföra fyra egenhändigt implementerade sökalgoritmer.

Din kod ska vara strukturerad så att funktioner är samlade i relaterade headerfiler med separata cpp-filer. Koden ska vara lättläst med förtydligande kommentarer både för funktionsdeklarationer och programkod. Var noggran med namngivningen av dina variabler. `a, i , x` är olämpliga namn. Namn ska vara beskrivande och hjälpa läsbarheten i din programkod.

Förslagsvis gör du en upppdelning i

1. Datagenerering - skapa känt indata till sökalgoritmerna. Organisera datat så att det lämpar sig för sökning.
2. Sökalgoritmer - din implementation av de olika sökmetoderna.
3. Tidsmätning - generera mätdata för analys.
4. Datarapportering - visualisering och analys av mätdatat.

4 Datagenerering

Samtliga sökalgoritmer arbetar mot samma ursprungliga data. Det som söks är slumptal. Det data vi valt i labben är en primtalsserie. Anledningen är att en primtalsserie har flera praktiska fördelar.

- Det ger en förutsägbar mängd träffar och missar. Det är praktiskt för att kontrollera riktigheten i sökmetoderna.
- Den är sorterad i stigande ordning. Praktiskt för binärsökningen. Underlättar skapandet av ett balanserat sökträd.
- Den är oordnad så att uppslagstabellen blir lagom lastad.

Metoden som rekommenderas för att generera en större mängd primtal kallas Eratosthenes såll (Sieve of Eratosthenes) vilket är ett enkelt sätt att peka ut primtal i ett intervall. Se 9.

5 Sökalgoritmer

Du skall utföra mätningar på följange metoder:

- Sekvensiell sökning. (Linjärsökning) **?** , Kap 6.1]
- Binärsökning. **?** , Kap 6.2.1]
- Uppslag i binärt sökträd. **?** , Kap 6.2.2]
- Uppslag i hashtabell. (Uppslagstabell) **?** , Kap 6.4]

5.1 Binärsökning

Prova gärna varianten interpoleringssökning, sökdatat bör vara ganska lämplig för metoden. Du behöver emellertid inte redovisa denna metod utan gör den för skojs skull.

5.2 Binärt sökträd

Vi har ingen avsikt att implementera någon självbalanserande struktur. Det är däremot enkelt att skapa ett balanserat träd om man konstruerar en sökmängd med storleken $N^2 - 1$. Roten blir då mittenelementet. Vänster barn genereras rekursivt av mängden til vänster, höger barn sammalunda.

En skillnad i implementationen som är av vikt är om det binära sökträdet representeras av den traditionella trädrepresentationen

```
struct node{
  T data;
  node* left; // nullptr leaf
  node* right; // nullptr leaf
} root;
```

eller om man istället väljer en kontinuerlig representation ex.

```
size_t parent(size_t node);
size_t left_child(size_t node);
size_t right_child(size_t node);

std::vector<T> binary_search_tree; // root at 0.
```

Välj den representation du tror är bäst. Motivera varför du gjort det valet.

Det binära sökträdet generaras utifrån primtalslistan som skapats för linjärsökning.

5.3 Hashtabell

Implementera hashtabellens uppslagsfunktion med en enkel modulusoperation. Välj storleken på hashtabellen så att lasten ligger under 50%.

Kollisioner hanteras enklast med en länkad lista. Det är lämpligt att beräkna hur djup den djupaste listan är. Om djupet är för stort minskar du lasten i hashtabellen.

```
struct node{  int data; node* next};

std::vector<node*> hashtable(reserved_size);

// Alternativt med std::unique_ptr
std::vector<std::unique_ptr<node>> hashtable(reserved_size);
// med motsvarande i node.
auto& item = hashtable[hash];
auto new_node = std::make_unique<node>({data,
    std::move(item);}
hashtable[hash].reset(new_node);
```

6 Tidsmätningar

För lämligt stora dataserier mäter du tiden för varje sökningsoperation. Målet är att kunna göra en trovärdig analys av algoritmens komplexitet utifrån dina mätdata. Denna tid bokförs av 7.

Nogrannhet måste iakktas vid dina tidsmätningar. Det som måste balanseras är noggrannhet och tiden det tar för en programkörning.

Använder du `C++` så finns `std::chrono::` ur `<chrono>`.

Enstaka sökningar är alldeles för kortlivade för att kunna mäta på ett meningsfullt vis. Det finns två tekniker för att komma runt detta.

- Gör lämpligt många sökningar, dividera resultatet med antalet sökningar. Lämpligt många sökningar är olika för olika sökmängder och olika sökmetoder.
- Sök under en bestämd tidsperiod. Resultatet är tidsperioden / antalet gjorda sökningar. Denna metod är ofta stabilare än den ovanstående.

En mätserie för varje indata måste bestå av minst 10 tidsmätningar. Fler är bättre. Ju fler mätningar desto mindre mätfel. Undersök standardavvikelsen i dina mätserier.

Vanliga misstag är

- Man gör sökningen för ett enstaka element. Då du itererar din sökalgoritm ska du i varje iteration söka efter ett nytt värde. Om du söker efter samma element i alla iterationer mäter du inte egenskaperna för metoden, utan du mäter egenskaperna för att söka det speciella värdet.

- Din kompilator optimerar bort sökningarna helt om du inte behandlar resultatet på något vis. Kompliatorn kan göra det eftersom om du ignorerar resultatet har sökningen igen effekt på programkörningen. Att förbruka tid är inte en effekt kompilatorn tar hänsyn till.
- Man genererar sökvärden som inte relateras till sökmängden. Om du söker bland 1000 primtal ska värdet du söker vara av samma storlek.
- Man mäter inte endast söktiden utan även initierande programkod. En vanlig fallgrop är att kopiera dataserien till sökfunktionen istället för att ge pekare eller iteratorer till serien. Man kan också tillhandahålla sökserien som en global variabel.

7 Datarapportering

Din programkörning ska lagra programkörningsresultatet på en enkel och lättillgänglig textform. Du kommer för redovisningen och rapporteringen att använda ett grafritningsverktyg som illustrerar alla dina resultat på ett överskådligt sätt.

Du har två alternativ för att åstadkomma det:

1. Spara samtliga datapunkter från programmet. Metod, storlek på sökmängden, tidsåtgång, serie. Du behöver då efterbehandla mätserierna.
2. Spara medelvärdet och standardavvikelsen för varje mätpunkt. Du behöver även spara antal mätningar och annan relevant data.

Utdatat som ditt program producerar måste vara lättbehandlat av den mjukvara du avser använda för dina grafer/sammanställning.

Ett förslag för alternativ två.

```
20000       0.1654       0.0032       50
40000       0.3284       0.0092       50
60000       0.4541       0.0232       50
80000       0.6210       0.0620       50
100000      0.7654       0.0419       50
...
```

Listning 1: linearsearch_samples.data

Den första kolumnen som representerar sökmängdens storlek ska täcka åtminstone en tiopotens; här 20000 → 200000.

För metoder som binärsökning och binära sökträd krävs sannolikt ett ännu större område.

Det sammanställda datat innehåller åtminstone

- Sorteringsmetod
- Varje mätpunkt med felmarginal

8 Redovisning

Redovisning av uppgiften görs med

- Förklarad och välstrukturerad programkod.
- Grafer som illustrerar uppmätta serier med felgränser. Graferna ska också innehålla en passning till den algoritmiska komplexitet som ges av litteraturen eller din implementation.
- En förklaring till varje graf som antingen bekräftar teoretisk gräns eller beskriver varför teori och verklighet skiljer sig.

Om du redovisar muntligen görs förklaringarna muntligen. Om du redovisar via inlämningslåda redovisar du en rapport. Bifoga då även dina mätdata på en läsbar form.

8.1 Felanalys

Utgångspunkten för felanalysen är du tar fram en standardavvikelse för varje mätpunktserie. Desto fler mätpunkter, desto lägre standardavvikelse. Standardavvikelsen är enkel att beräkna i programkod

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2} \tag{1}$$

N är antal mätningar per mätserie för en punkt
$\overline{x}$ är medelvärdet av mätserien
x_i är värdet för mätpunkten

```
value_type std_dev(std::valarray<value_type> measurements){
   value_type sum = std::accumulate(measurements.begin(),
                                    measurements.end(),
                                       value_type());
   auto n = measurements.size();
   auto avg = sum / measurements.size();

   std::valarray<value_type> dev_square =
       std::pow(measurements - avg, 2);

   value_type square_sum =
       std::accumulate(dev_squared.begin(),
                                    dev_squared.end(),
                                       value_type());

   return std::sqrt(square_sum * (1.0 / (N - 1)));

}
```

Standardavvikelsen ska presenteras på ett sådant sätt att det är tydligt att den innefattar den teoretiska passningen. Om du vill göra en automatisk passning till den teroetiska modellen finns metoden *minsta kvadratmetoden*[**?**]. Om man

har ett grafritningsverktyg som kan uppdatera grafer direkt i verktyget, som Calc och Excel, är det möjligt att höfta en rimlig passning genom att gissa.

8.2 Grafer

Ett lämpligt utseende för ett diagram som illustrerar mätpunkter och teoretiska modeller innehåller mycket information. Utforma dina grafer så att du har en balans mellan information i diagrammet och läsbarhet.

Graferna görs lämpligtvis med något av följande alternativ

- matplotlib.pyplot - Matlabinspirerat plotverktyg i python. Pyplot gör det enkelt att läsa in dina resultat och sammanställa grafer. Detta är lämpligt om du är bekant med python.
- Matlab/Octave - Är du bekant med Matlab eller Octave sedan tidigare är detta alternativ lämpligt.
- Gnuplot - Lämpligt om du vill sammanställa alla resultat med byggskript/makefiler.
- Calc/Excel - Kräver handpåläggning vid dataimporten till kalkylbladen. Om du inte är bekant med något av de tidigare alternativen är detta ett rimligt val.

9 Framtagning av primtal

Algorithm 1 Sieve of Erastosthenes

Require: En storlek N och en bool-array A med storlek på minst N. Array A initieras med true för samtliga element.

Ensure: Elementen i A är true för element vars index är primtal. False för alla andra element.

```
A_0 ← false
A_1 ← false
i ← 2
while i < N/2 do
    divisor ← i * 2
    while divisor < N do
        A_divisor ← false
        divisor ← divisor + i
    end while
    i ← i + 1                          ▷ Forsätt med nästa primtal
    while A_i = false do
        i ← i + 1
    end while
end while
```

Resultatet av algoritmen omvandlas sedan till lämplig form för din programkörning.