Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Magnus Nielsen

17 oktober 2020

TDDE22
Datastrukturer och algoritmer
Datortentamen (DAT1)
2020-08-25, 08–12 - Lösningsförslag

Håll i åtanke att dessa är *Lösningsförslag*. Det finns inte bara en bra lösning till majoriteten av uppgifterna.

Uppgift 2 – Storskalig tentahantering

a)

1. Vi traverserar listan från start tills vi hittar rätt kurskod och student. Tidskomplexitet: $O(n)$.
2. Då arrayen är sorterad kan vi nyttja binärsökning. Vi tittar först mitt i arrayen (index n/2). Om kombinationen vi söker är "mindre" än elementet vi tittar på upprepar vi samma sak på "vänstra" halvan av arrayen (alla index $<$ n/2), om det är "större" gör vi det i "högra" halvan (alla index $>$ n/2). Vi fortsätter halvera sökområdet tills vi hittar rätt element. Tidskomplexitet: $O(\log(n))$.
3. I ett balanserat sökträd avgör vi i varje individuellt fall om elementet vi söker är mindre än eller större än noden vi tittar på. Resultatet av jämförelsen avgör om vi ska traversera åt vänster eller höger i trädet, eller om vi har hittat rätt element. Sökning i (binära) sökträd begränsas av trädets höjd, och då trädet är balanserat är höjden logaritmisk mot antalet noder. Tidskomplexitet: $O(\log(n))$.
4. I en hashtabell kan vi förutsätta att hashfunktionen pekar ut rätt index på konstant tid ($O(1)$). Det står inget i uppgiften om att arrayen där tentorna lagras är sorterad, så vi antar att den inte är det. Vi måste alltså, i värsta fallet, traversera hela arrayen för den specifika kurskoden. Vi vet att antalet tentander per tenta är mindre än n, så vi har tidskomplexitet: $O(n)$.

b)

1. Vi använder oss av en dynamisk array (ex: ArrayList) för lagring av resultatet och itererar fram till första positionen i den länkade listan som har rätt kurskod, lägger till den tentan i arrayen, och fortsätter på samma sätt så länge kurskoden överensstämmer. Tidskomplexitet: $O(n)$.
2. Vi binärsöker, likt i a, fram tills vi hittar rätt kurskod. Därefter lägger vi till den tentan i en dynamisk array (ex: ArrayList). Vi itererar sedan åt både "vänster" och "höger", och lägger till samtliga tentor så länge kurskoder överensstämmer. Sökningen har tidskomplexitet $O(\log(n))$ för att hitta rätt kurskod, och att ta fram samtliga tentor har tidskomplexitet $O(n)$ då vi vet att maximala antalet tentor i en kurs bör vara signifikant mindre än n. Tidskomplexitet: $O(\log(n) + n) \subset O(n)$.
3. Ungefär samma situation som med binärsökning i array: I värsta fallet har vi $O(\log(n))$ för att hitta rätt kurskod (logaritmisk höjd på trädet), måste sedan fortsätta traverseringen och

spara undan varje tenta som har matchande kurskod. Maximala antalet tentor i en kurs har bör vara betydligt mindre än n, och vi har då tidskomplexitet: $O(\log(n) + n) \subset O(n)$.

4. Vi lagrar, för varje kurs, en lista med samtliga tentor för den kursen. Vi kan förutsätta $O(1)$ för uppslagning i hashtabellen. Tidskomplexitet: $O(1)$.

c)

1. Vi itererar fram till rätt position i den länkade listan, skapar in en ny nod med den nya tentan och pekar om relevanta pekare. Peka om pekare tar konstant tid. I värsta fallet måste vi iterera hela listan. Tidskomplexitet: $O(n)$.
2. Vi söker fram rätt position med binärsökning. Vi måste skapa utrymme genom att flytta samtliga efterföljande tentor ett steg längre fram i arrayen, och sätter sedan in tentan på den nyligen friade platsen. Tidskomplexitet: $\log(n) + n \in O(n)$.
3. Vi traverserar trädet som vanligt tills vi hittar en lämplig tom lövnod där vi sätter in tentan. På tillbakavägen uppdaterar vi balansinformation och, vid behöver, utför rotationer för att återställa balansen i trädet. Tidskomplexitet: $O(\log(n))$.
4. I en hashtabell kan vi förutsätta att hashfunktionen pekar ut rätt index på konstant tid ($O(1)$). Det står inget i uppgiften om att arreyen där tentorna lagras är sorterad, så vi antar att den inte är det. Då kan vi sätta in nya tentor sist (första lediga plats) vilket också tar konstant tid. Tidskomplexitet: $O(1)$.

d)

Antaganden: Vi kan inte förutsätta någon speciell insättningsordning, så vi kan inte optimera baserat på det.

Hämtning: Examinatorn kommer att vilja hämta samtliga tentor för en given kurskod i en och samma hämtning.

1. Insättning: n stycken insättningar ger tidskomplexitet: $n * O(n) = O(n^2)$.

   Hämtning: Vi vet att antalet kurskoder $<$ n, så vi får tidskomplexitet: $O(n^2)$.

   Totalt: $O(n^2 + n^2) \subset O(n^2)$.

2. Insättning: n stycken insättningar ger tidskomplexitet: $O(n^2)$.

   Hämtning: Vi vet att antalet kurskoder $<$ n, så vi får tidskomplexitet $O(n^2)$.

   Totalt: $O(n^2 + n^2) \subset O(n^2)$

3. Insättning: n stycken insättningar ger tidskomplexitet: $O(n \log(n))$.

   Hämtning: Vi vet att antalet kurskoder $<$ n, så vi får tidskomplexitet: $O(n^2)$.

   Totalt: $O(n^2 + n \log(n)) \subset O(n^2))$

4. Insättning: n stycken insättningar ger tidskomplexitet: $O(n)$

   Hämtning: Vi vet att antalet kurskoder $<$ n, så vi får $O(n)$

   Totalt: $O(n + n) \subset O(n)$

e)

1. Fördelar: Lätt att implementera, relativt effektivt utnyttjande av minne. Vi har bara exakt så många noder som vi behöver i listan. Minimal overhead för pekare i varje nod som håller reda på nästa element i listan.

   Nackdelar: Långsam / dålig tidskomplexitet.

2. Fördelar: Snabb sökning, i synnerhet efter individuella tentor.

   Nackdelar: Långsam insättning och dålig minnesanvändning. Vi måste antingen avsätta tid till att allokera mer minne och kopiera existerande array varje gång arrayen blir full, eller allokera tillräckligt mycket minne från start för att vara säkra på att ha utrymme för samtliga tentamina.

3. Fördelar: Snabb insättning och söktid efter individuella tentamina, och bra balans mellan de två. Bra minnesanvändning med bara lite overhead för pekare (om vi har länkad implementation).

   Nackdelar: Rekursiv traversering.

4. Fördelar: Snabbast insättning och hämtning av alla tentor i en kurs, vilket är den vanligaste användningen.

   Nackdelar: -

Svar: Hashtabell (alternativ 4) då den är snabbast i samtliga fall av normal användning. Inte nödvändigtvis snabbast vid hämtning av individuella tentamina, men det är ett undantagsfall.

f) Om vi ändrar från

```
HashTable<String, ArrayList<Exam>>
```

till

```
HashTable<String, HashTable<String, ArrayList<Exam>>>
```

Vi hashar först på kurskod och hittar då en hashtabell som hashar på student-id. Det ger oss en söktid (1) även när vi söker enstaka studenter. Insättningstiden är opåverkad.

g) Då vi vet att alla insättningar kommer att vara gjorda innan vi behöver ha en sorterad datamängd kan vi utnyttja detta till att, vid insättning, bara stoppa in tentan sist i arrayen. När alla tentorna är insatta kan vi sortera datamängden *en* gång. Totala insättningstiden går då från $O(n^2)$ till $O(n + n\log(n)) \subset O(n\log(n))$

Uppgift 3 – Vägvisande spelutveckling

Notera: Många av tidskomplexiteterna går att förenkla ytterligare. Vi har antalet noder n, och antalet bågar $m \leq 4n$ (fyra riktningar - väggar, så alla n+m kan förenklas till n). Egentligen vet vi till och med att $n = 100 * 100 = 10000$.

a) Jag rekommenderar att de använder en kö (först in, först ut). När en fiende dör stoppar vi in den i kön, och när nästa fiende ska skapas hämtar vi nästa från kön.

Insättning: $O(1)$ då vi sätter in sist.

Borttagning: $O(1)$, vi tar bort första elementet.

Detta löses generellt genom att hålla reda på första och sist använda index i en array.

b) Jag rekommenderar att vi istället använder oss av en prioritetskö. Om vi sätter olika prioritet beroende på hur starka fienderna är kan vi alltid se till att den starkaste i kön blir den första som återskapas. Det finns olika implementationer, men vi har pratat om heap-baserade prioritetsköer i kursen så jag rekommenderar det. Vi får då följade tidskomplexitet:

Insättning: $O(\log(n))$ då vi måste "bubbla upp" nyligen insatta element.

Borttagning: $O(\log(n))$ då vi måste återställa heapordningen.

c) Vi kan se spelplanen som en oriktad graf. Alltså kan vi köra Bredden Först sökning för att hitta kortaste vägar. Varje ruta blir en nod, och bågarna blir relationen att noderna är angränsande i matrisen. Väggar är inte passerbara, så vi hoppar över dessa i traverseringen.

Tidskomplexitet: $O(n+m)$ där n är antalet noder och m är antalet bågar (om vi vill: $O(n+m) \subset O(n)$ enligt "notera" ovan).

d) Jag rekommender att vi kan sätta en "vikt" på varje ruta som motsvarar friktion eller liknande faktor som avgör hur begränsad man är av terrängen. Vi inser snabbt att vi nu har en viktad graf och att BFS inte längre kommer att fungera, så jag föreslår Dijkstra's algoritm vilket i princip är BFS med en prioritetskö.

Tidskomplexitet: $O((n+m)\log n)(\subset (O(n\log(n))), se "notera'' ovan)$

e) Istället för att köra Dijkstra's för varje fiende ($O(f * n\log(n))$) kan vi utnyttja att en Dijkstra's med utgångspunkt i spelaren kommer att stöta på samtliga fiender. Vi sparar undan vägarna för varje fiende och låter sedan fienden traversera vägen baklänges.

Detta ger samma tidskomplexitet som i d) ($O(n\log(n))$), men med lite mer overhead då vi måste spara undan vägarna för varje fiende.

f) Den totala tidskomplexiteten blir mycket bättre. Vi skulle inte behöva köra Dijkstra's för varje förflyttning, eller alls. Dessvärre skulle lagringen ta upp väldigt mycket plats. Vi har ≈ 10000 rutor (ca 100x100), om matrisen i sin tur är n^2 stor har vi ≈ 100000000 positioner. Lagrar vi något alls (dvs 1 byte per element) i den är vi redan uppe i maximala mängden minne får vi använda (100 MB).

Svar: Jag rekommenderar mot att använda den lösningen.

g) Grafen i sig är inte speciellt stor, så vi kan motivera att använda oss av två grafer: en där dörrarna är öppna och en där de är stängda. Så länge vi har tillgängliga nycklar använder vi grafen där dörrarna är öppna, och så fort nycklarna tar slut byter vi graf. Varje gång vi passerar en låst dörr förbrukas ett lösenord.

Vi skulle även kunna komma undan utan att lagra två separata grafer, bara genom att låta positionen i algoritmen hålla reda på vilken version vi skulle ha nyttjat (om vi kan öppna dörrar eller ej).