Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Filip Strömbäck och Magnus Nielsen

29 oktober 2020

Datastrukturer och algoritmer Lösningsförslag till tentamen 2020-10-29

1. Jag har läst och förstått tentans regler, och jag lovar att jag har följt dem under tentans gång.
2. En gastronomisk hantering:
   (a) Insättning implementeras lämpligtvis som följer:
      1: Eftersom rätten är ny har den inga beställningar ännu. Vi kan därför sätta in den i början av den länkade listan. Tidskomplexiteten är $\mathcal{O}(1)$ eftersom vi bara behöver uppdatera *head*-pekaren.
      2: Eftersom arrayen är osorterad kan vi helt enkelt sätta in rätten i slutet av arrayen. Tidskomplexiteten är $\mathcal{O}(1)$ amorterad tid om vi dubblerar storleken på arrayen när det inte finns plats för fler element.
      3: Vi kan helt enkelt sätta in ett element med nyckel 0 i sökträdet. Detta innebär att vi börjar i rootnoden och följer *left*-pekaren tills vi kommer till en lövsnod. Vi kan där sätta in vår nya rätt (0 kommer alltid vara minst), och balansera trädet på vägen tillbaka i rekursionen. Detta har tidskomplexiteten $\mathcal{O}(\log n)$.

   (b) Borttagning givet rättens namn implementerar jag som följer:
      1: Vi måste iterera genom den länkade listan eftersom den inte är sorterad efter namn. För varje element, se om namnet stämmer överens med det vi letar efter. Stämmer det så kan vi länka bort elementet ur listan. Totalt blir tidskomplexiteten $\mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$.
      2: Även här måste vi iterera genom arrayen tills vi hittar elementet. När vi hittar rätt element måste vi dessutom flytta resterande element ett steg framåt i arrayen (elementen måste ligga bredvid varandra, och vi har ingen representation av ett ”hål”). Tidskomplexitet: $\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$.
      3: I och med att sökträdet inte är sorterat på rättens namn måste vi traversera hela trädet för att hitta rätt. Detta kan vi exempelvis göra genom att rekursivt traversera trädet (inorder), och när vi har hittat rätt element tar vi bort det på vanligt sätt (dvs. byt eventuellt plats med inorder efterföljare, balansera sedan på vägen upp i trädet). Totalt blir tidskomplexiteten $\mathcal{O}(n) + \mathcal{O}(\log n) = \mathcal{O}(n)$.

   (c) För att öka antalet beställningar krävs följande:
      1: Hitta rätten på samma sätt som i (b). Öka sedan antalet beställningar i elementet, flytta sedan elementet framåt i listan tills det hamnar på rätt ställe (innan ett element med fler beställningar, eller i slutet av listan). Tidskomplexitet: $\mathcal{O}(n)$
      2: Hitta rätten på samma sätt som i (b), öka sedan antalet beställningar i elementet. Vi behöver inte göra något mer eftersom arrayen inte är sorterad. Tidskomplexitet: $\mathcal{O}(n)$
      3: Hitta och ta bort rätten på samma sätt som i (b). Sätt sedan in ett nytt element med rätt antal beställningar. Detta görs liknande som i (a), men med skillnaden att vi inte alltid går längst till vänster. I stället utnyttjar vi sökträdsegenskapen och väljer antingen vänster eller höger tills vi hittar en lämplig plats. Tidskomplexiteten blir fortfarande $\mathcal{O}(n) + \mathcal{O}(\log n) = \mathcal{O}(n)$. Det går också att flytta elementet som ska uppdateras, men tidskomplexiteten blir densamma.

   (d) För att hitta de k billigaste rätterna:
      1: Eftersom listan är sorterad efter antal beställningar kan vi helt enkelt iterera från slutet av den länkade listan (om den är dubbellänkad) och returnera de k sista elementen. I så fall blir tidskomplexiteten $\mathcal{O}(k)$. I en enkellänkad lista behöver vi traversera hela listan för att hitta de k sista elementen, vilket ger en tidskomplexitet av $\mathcal{O}(n)$ i stället.

2: En möjlighet är att sortera arrayen och sedan ta de sista k elementen. Detta har tidskomplexiteten $\mathcal{O}(k + n \log n) = \mathcal{O}(n \log n)$ (ordningen spelar ingen roll, så det är okej att sortera arrayen). En anna möjlighet med bättre tidskomplexitet är att iterera genom arrayen. Vi lägger varje element i en min-heap. I varje steg så tar vi bort element från heapen tills den innehåller maximalt k element. Efter detta innehåller heapen de k största elementen i arrayen. Tidskomplexiteten blir då: $\mathcal{O}(n \log k)$.

3: Traversera trädet inorder, men börja från höger i stället med från vänster. Returnera sedan de k första elementen i den traverseringen. Detta får tidskomplexiteten $\mathcal{O}(\log n + k)$ ($\mathcal{O}(\log n)$ om k är väldigt litet i förhållande till n).

(e) Om vi antar att det är relativt vanligt att hitta de k mest beställda rätterna så är alternativ 1 relativt bra (givet att listan är dubbellänkad), eftersom den inte är sämre än de andra, men har den trevliga egenskapen att det är enkelt att hitta de populäraste rätterna.

Optimalt vill vi nog kombinera listan med ett sökträd eller en hashtabell som innehåller namn och pekare till noder i den länkade listan. Då går det att hitta ett element i listan snabbt genom att söka efter namnet i hashtabellen. Sedan kan vi uppdatera strukturen relativt snabbt. Samma sak fungerar även i ett balanserat sökträd.

3. Skrattletarna:

(a) Givet två listor s (skepp) och k (kanoner), vill vi hitta två element $s[i]$ och $k[j]$ så att $s[i]+k[j] \leq b$, där b är vår budget. Vi vill också att $s[i] + k[j]$ ska vara så stort som möjligt.

Vi kan exempelvis lösa detta med sortering:

- Sortera s och k (exempelvis med quicksort, förväntad tidskomplexitet $\mathcal{O}(n \log n)$).
- Sätt $j = m - 1$, $r = 0$, $i_r = 0$ och $j_r = 0$
- För varje i från 0 till n:
  - Minska j tills $s[i] + k[j] \leq b$.
  - Om $s[i] + k[j] \leq r$, sätt $r = s[i] + k[j]$, $i_r = i$, $j_r = j$
- Bästa priset finns sedan i r, bästa skeppet är den med index i_r i den sorterade arrayen, bästa kanonerna är de med index j_r i den sorterade arrayen.

Totalt tar detta $\mathcal{O}(n \log n + m \log m)$ tid, då loopen i lösningen ovan tar $\mathcal{O}(n + m)$ tid att köra. Det går även att lösa detta genom att binärsöka fram j i loopen, då tar loopen i stället $\mathcal{O}(n \log m)$ tid att köra, vilket ger ungefär samma tidskomplexitet som lösningen ovan.

(b) Eftersom vi ”bara” har $n = 100$ kandidater så kan vi lösa problemet med en array:

- Skapa en array a av par med 100 platser.
- För varje plats i i arrayen a, sätt platsen till $(0, i)$.
- Iterera sedan igenom arrayen från mjukvaran. För varje element e:
  - Öka första elementet i paret på plats i i arrayen a med ett (dvs. `a[i].first++`)
- Skapa sedan en prioritetskö i form av en min-heap.
- För varje element e i arrayen a:
  - Lägg till e på prioritetskön.
  - Om det är fler än $k = 20$ element i prioritetskön, ta bort ett element.
- Prioritetskön innehåller nu $k = 20$ element, där andra delen av paret innehåller numret på de $k = 20$ bästa kandidaterna.

Detta har tidskomplexiteten: $\mathcal{O}(n) + \mathcal{O}(n \log k) = \mathcal{O}(n \log k)$. Andra delen går även att lösa med sortering, vilket ger $\mathcal{O}(n \log n)$ totalt.

Har vi många olika pirater kan vi i stället för ett array använda en hashtabell för a ovan. Resten av lösningen blir densamma i det fallet.

(c) En möjlig lösning är att bygga en graf av alla grottor och alla vägar (lämpligtvis representerad med en grannlista). Vi kan sedan använda en modifierad BFS för att konstruera ett vägnät:

- Skapa en tom mängd där vi kan lagra besökta noder (*visited*)
- Skapa en tom mängd där vi kan lagra bågar (*edges*)
- Skapa en kö q, lägg till alla startpunkter (alla noder utan inkommande bågar, vi kan enkelt hitta dessa genom att i varje nod lagra antalet bågar som slutar i noden, och sedan välja noderna med 0 bågar till dem).
- Lägg till startpunkterna från q i *visited* (utan att ta bort dem).
- Så länge q inte är tom:
  - Ta bort första elementet ur q. Spara det i *current*.
  - För varje båge från *current* till *to*:
    - Om *to* inte finns i *visited*: lägg *to* i *visited*, lägg till *to* i q, samt lägg till båden i *edges*.
- Nu innehåller *edges* ett träd som motsvarar de vägar som måste passeras för att komma till alla grottor. Dock saknas vägar från den sista grottan till slutpunkten. De kan vi enkelt lägga till med en djupet-först-sökning i de bågar som *edges* innehåller. När sökningen hittar en nod utan utgående bågar behöver den helt enkelt lägga till en båge från den noden till slutnoden (en av grottorna kommer att ha en väg till slutnoden, så vi kommer inte alltid behöva lägga till en väg).
- Om vi vill så kan vi också räkna hur många pirater som behöver skjutas upp. Vi kan helt enkelt göra en djupet-först-sökning av bågarna i *edges* och rekursivt summera hur många vägar det finns till slutnoden.

Detta tar totalt: $\mathcal{O}(|V| + |E|)$ tid. BFS tar $\mathcal{O}(|V| + |E|)$ tid, likaså att hitta alla startpunkter och lägga till vägar till slutpunkten.