Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Filip Strömbäck och Magnus Nielsen

28 oktober 2020

Datastrukturer och algoritmer Lösningsförslag till tentamen 2021-10-28

1. (a) Exempelvis:
   13, 24, 26, 6, 18, 19, 30, 8
   30, 19, 18, 8, 26, 6, 24, 13
   24, 13, 26, 6, 19, 30, 18, 8
   26, 6, 30, 19, 18, 8, 24, 13

   (b) Alternativ 1: Ersätt 26 med en tombstone (efter vanlig uppslagning för att hitta det). Detta element fungerar som ett null-element i beaktningen att det beskriver att inget element finns på den givna platsen. Däremot skiljer den sig från null i och med att uppslagningen inte kan avslutas när en tombstone hittas, precis som när ett ”vanligt” element hittas.

   Alternativ 2: Ta bort elementet (efter vanlig uppslagning för att hitta det). Fortsätt att iterera igenom arrayen tills ett null-element hittas. För varje element, ta bort elementet från arrayen och sätt in det genom att anropa ”insert”. Detta gör så att element som behöver flyttas hamnar på rätt plats.

   (c) När arrayen blev full kan implementationen ha allokerat en ny, dubbelt så stor, array och kopierat alla nycklar till den. I och med att hashfunktionen ändras innebär det att hashtabellen behöver ”hasha om” alla element under kopieringen (dvs. kör ”insert” på dem ett och ett på nytt).

2. (a) A, B, H, C, F, G, E

   (b) A, B, C, F, E, H, G

3. 1-a, 2-b, 3-c, 4-e, 5-d

4. (a) Loopen körs $n/2$ gånger, resterande operationer tar $\mathcal{O}(1)$ tid: $\mathcal{O}(n/2) = \mathcal{O}(n)$

   (b) Yttre loopen körs $\log(n)$ gånger, inre loopen körs n gånger, resterande operationer tar $\mathcal{O}(1)$ tid: $\mathcal{O}(n\log(n))$

   (c) Yttre loopen körs n gånger. Varje iteration tar $\mathcal{O}(i)$ tid eftersom `fun` anropas. Detta ger totalt: $\sum_{i=0}^{n} \mathcal{O}(i) = \mathcal{O}(n(n-1)/2) = \mathcal{O}(n^2/2) = \mathcal{O}(n^2)$. Alternativt kan vi tänka att vi kan arrangera det totala antalet operationer som en triangel (yttre loopen i x-led, inre i y-led exempelvis) och beräkna arean av den.

   (d) Yttre loopen körs n gånger. Koden inuti loopen innehåller 2 fall. Fall 1 tar $\mathcal{O}(n)$ tid, fall 2 tar $\mathcal{O}(1)$ tid. Fall 1 körs dock exakt 2 gånger (givet att $n > 2$). Innan loopen kör vi `funnier`, vilken tar $\mathcal{O}(n\log(n))$ tid. Totalt får vi: $\mathcal{O}(n\log(n)) + \mathcal{O}(2n) + n\mathcal{O}(1) = \mathcal{O}(n\log(n)) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n\log(n))$

5. (a) Enligt uppgiften behöver programmet inte lagra alla mätvärden. Vi behöver alltså bara lagra de mätvärden vi behöver för att kunna beräkna medelvärden. För detta räcker det att lagra de k senaste mätvärdena. Jag väljer att lagra dessa i en kö implementerad med en cirkulär array med storleken k, eftersom det ger $\mathcal{O}(1)$ insättning och borttagning. Jag behöver också en heltalsvariabel som innehåller summan.

   (b) Jag initierar kön med en maximal storlek k (så att implementationen inte behöver ändra storleken på arrayen). Jag lägger sedan till k stycken 0:or i kön. Min heltalsvariabel sätter jag till 0.

   (c) När programmet får in ett nytt mätvärde, x:

      i. Ta bort ett element från kön, lagra i z

      ii. Subtrahera z från summavariabeln

      iii. Lägg till x i kön

      iv. Addera x till summavariabeln

v. Skriv ut summa/k, detta är medelvärdet

(d) Tidskomplexiteten blir $\mathcal{O}(1)$ då köoperationerna tar $\mathcal{O}(1)$ tid. Bästa och värsta fall är båda $\mathcal{O}(1)$ då kön redan är initierad till rätt storlek vid programmets start.

(e) Det tar exakt $\mathcal{O}(1)$ tid att behandla varje element. Vi har inga amorterade eller värsta fall att ta hänsyn till, så totalt får vi $\mathcal{O}(n)$ tid.

(f) Minneskomplexiteten är $\mathcal{O}(k)$ eftersom kön lagrar de senaste k mätvärdena. Vi behöver inte lagra alla n mätvärden.

6. (a) Om vi betraktar vägnätet som en graf (där städer är noder och vägar är bågar) så blir problemet vi vill lösa: välj bågar så att det finns vägar mellan alla noder, vi vill välja bågar med så liten total vikt som möjligt. Vi vet att det räcker att välja $|V| - 1$ bågar för att det ska finnas en väg. Har vi exakt $|V| - 1$ bågar har vi ett träd. Vi ska alltså hitta det minsta uppspännande trädet i grafen. För att göra detta kan vi exempelvis använda Prims algoritm:

1 Skapa en prioritetskö p. Låt denna sortera sina element (bågar) baserat på bågens vikt.

2 Lagra `visited` = `false` för varje nod i grafen (exempelvis i en array eller en hashtabell)

3 Lägg till alla bågar från en startnod (exempelvis Glasgow) till prioritetskön.

4 Markera startnoden som `visited`

5 Repetera tills alla noder är besökta:

5.1 Plocka ut bågen med lägst vikt ur p.

5.2 Om bågen leder till en nod som är besökt, gå till 5.1

5.3 Annars, markera den nya noden som besökt, och lägg till alla bågar från den nya noden till p

(b) Prims algoritm tar $\mathcal{O}(|E| \log |V|)$ tid – vi har en prioritetskö som vi maximalt sätter in alla bågar i.

(c) Givet att den graf som produceras av (a) är ett träd kan vi helt enkelt traversera grafen med en rekursiv algoritm som vi kallar `traverse(n)`:

1 Markera n som besökt

2 Sätt $s = 0$

3 För varje båge från n:

3.1 Om bågen går till en ej besökt nod: anropa `traverse` på den noden, spara resultatet i x. Lagra att bågen behöver x plogar som körs längs vägen. Sätt också $s = s + x$.

4 Om $s = 0$ är detta en lövnod. Sätt i så fall $s = 1$.

5 Returnera s

Vi anropar `traverse`(Glasgow) för att beräkna hur alla plogar ska åka från Glasgow.

Detta ger en algoritm som hittar alla lövnoder, och ser till att skicka en plog dit. För resterande noder räknar algoritmen hur många lövnoder som finns ”under” noden för att komma fram till hur många plogar som måste åka genom dessa städer.

(d) Algoritmen har $\mathcal{O}(|V|)$ tid eftersom den traverserar alla noder i grafen exakt en gång. Eftersom grafen numera är ett träd finns exakt $|V| - 1$ bågar, så det tar ingen ”extra” tid att undersöka bågar.

(e) Lösningen i (c) ger en lösning som minimerar den totala sträckan som behöver skottas. Däremot tar lösningen inte hänsyn till att alla plogar startar i Glasgow. Det gör att fallet ofta är att vi behöver låta fler än en snöplog åka längs samma vägsegment för att de senare ska kunna dela på sig. Detta gemensamma vägsegment kan ibland vara väldigt långt, och utgör en onödig sträcka för en av snöplogarna.