Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Magnus Nielsen

23 oktober 2017

TDDC91
Datastrukturer och algoritmer
Datortentamen (DAT1)
2017-10-23, 14–18
Lösningsförslag

1. **Uppgift 2**
   A: - BubbleSort
   B: - SelectionSort
   C: - InsertionSort
   D: - QuickSort

2. **Uppgift 3**

   (a) Exempel på lösningar:
   5 7 15 1 8 3 18 9
   1 5 7 15 8 3 18 9
   1 3 5 7 15 8 18 9
   3 1 5 7 15 8 18 9

   (b) **Alternativ 1:**
   Vi tar bort 8 och hashar om probing kedjan från borttaget index till första nullvärde:
   $[null, 1, null, 3, null, 5, 15, 7, 18, 9]$
   Alternativ 2:
   Vi tar bort 8 och sätter en tombstone / delete token på den nyligen tömda platsen:
   $[9, 1, null, 3, null, 5, 15, 7, X, 18]$
   En poäng per korrekt svar.

3. **Uppgift 4**

   (a) Vi vet att det är stora mängder data som ska sorteras (åtminstone på sikt), och vi vet det är datum vi ska sortera. Med detta i åtanke vill vi se till att vi har en algoritm som fungerar väl för syftet. QuickSort eller MergeSort är bra val, men de är jämförelsebaserade och sålunda kan de inte bli snabbare än $O(n \log n)$. Vi har de räknebaserade algoritmerna RadixSort och BucketSort som båda är bra till stora datamängder och har tidskomplexitet $O(n)$, men de använder mycket mer minne än de jämförelsebaserade algoritmerna, och vi skulle bli tvungna att typkonvererta från datum till heltal.

   Ex 1: Då banken har god ekonomi och gör jag antagandet att de har (eller har råd med) riktigt bra datorer, och gör valet att hastigheten är viktigare än att vara försiktig med minnet och jag bedömer att det är tillräckligt stora datamängder på sikt att det kommer vara en tidsbesparing trots typkonvertering. Jag väljer alltså numerisk sortering, och jag väljer typkonvertera datumen till heltal på formen: YYYYMMDD. Att konvertera ett datum till heltal förutsätter vi tar konstant tid ($O(1)$), så att konvertera alla n datum borde rimligen ha tidskomplexiteten $O(n)$. RadixSort partitionerar data likt QuickSort (fast med en annan metod), men den är inte lika snabb på små datamängder, i synnerhet inte om det är stora tal, så jag väljer att använda mig av CountingSort för tillräckligt små datamängder. Brytpunkten avgör

jag med testning. (Jag har tagit hänsyn till tidskomplexitet, minne, samt både stora och små datamängder. Full poäng).

Ex 2: Jag anser att de räknebaserade algoritmerna skulle kräva för mycket minne, och jag tycker definitivt inte att det vore värt typkonverteringen det skulle kräva, så beslutar mig för att använda jämförelsebaserade algoritmer. Både QuickSort och MergeSort hade fungerat bra, men QuickSort har jag implementerat i labbserien, så jag är bekväm med, och förstår, den och valet faller därför på QuickSort. Då jag vet att QuickSort blir betydligt mindre effektiv på små datamängder väljer jag att köra tillräckligt små partitioner genom SelectionSort (eller InsertionSort eller någon annan algoritm som är speciellt effektiv på små datamängder). (Jag har tagit hänsyn till tidskomplexitet, minne, samt både stora och små datamängder. Full poäng).

(b) **Ex 1:** RadixSort och CountingSort har båda tidskomplexiteten $O(n)$
Ex 2: QuickSort har tidskomplexiteten $O(n^2)$ i värsta fallet, men förväntade fallet är $O(n \log n)$. Selection-/Insertion/etc-sort har tidskomplexitet $O(n^2)$, men då jag använder dem som en optimering på riktigt små datamängder för QuickSort som redan har en förväntad tidskomplexitet på $O(n \log n)$ kan jag med gott samvete säga att vår lösning har en tidskomplexitet $O(n \log n)$ i det förväntade fallet.

(c) Vi har nu en numerisk nyckel istället för datumklassen, och är kan sålunda använda mig fritt av vilken av de ovan nämnda sorteringsalgoritmerna utan typkonverteringar.
Ex 1: På så vis kan jag använda mig av en Radix-/CountingSort lösning liknande den innan. Jag bedömer att det är än mer värt det nu då jag slipper den tidigare overhead som typkonvereringen krävde. Tidskomplexiteten för lösningen kommer fortfarande vara $O(n)$.
Ex 2: Jag hade beslutat mig för jämförelsebaserade algoritmer, och anser även nu att de är bättre eftersom de tar mindre minne i anspråk. Numeriska heltal är jämförbara, vilket gör att QuickSort fortfarande är ett passande alternativ, i synnerhet kombinerat med Insertion-/Selection-/etc Sort. Eventuellt kan jämförelsen i sig till och med vara något billigare eftersom jag inte vet hur dyr jämförelsen mellan datum är. Baserat på detta samt vår motivation ovan anser jag fortfarande att QuickSort med SelectionSort för mindre partitioner är ett bättre alternativ. Jag bibehåller tidskomplexiteten $O(n \log n)$.
(Vi har i samtliga fall gjort grundliga nya utvärderingar baserat på de nya förutsättningarna och antingen gjort nya val, med motivering, eller inte gjort nya val baserat på motivering. Full poäng).

4. **Uppgift 5**

(a) Merge sort är en divide and conquer algoritm som partitionerar datamängden genom att halvera den. De nya partitionerna halveras sedan även de stegvis till vi bara har ett element i varje partition. Partitionerna slås sedan samman (merge) i sorterad ordning. När partitionerna är större än ett element slår vi samman dem genom att jämföra minsta elementet i första partitionen med minsta elementet i den andra partitionen och sedan sätta det minsta av dem först i sammanslagningen. Vi fortsätter på samma vis tills alla element är instoppade, och upprepar detta steg för alla partitionerna.
Merge sort har tidskomplexitet $O(n \log n)$.

(b) $\Theta(n)$

(c) $\Theta(n)$

(d) $\Theta(n \log n)$

5. **Uppgift 6**

   (a) Nej. Trädet är ett binärt träd (0-2 barn per nod) men ej ett binärt sökträd. För att det ska vara ett binärt sökträd krävs att alla noder i vänstra barnträdet från roten är mindre än roten, och alla noder i högra barnträdet är större än roten. Det samma gäller för samtliga noder i trädet utom de som inte har några barn.
   Rätt svar och stark motivering: 2p. Rätt svar och svag motivering: 1p. Fel / ingen motivering: 0p

   (b) Ja, då skillnaden i djup mellan lövnoder är maximalt 1/-1 (i fallet med detta träd är skillnaden i djup 0).
   Rätt svar och stark motivering: 2p. Rätt svar och svag motivering: 1p. Fel / ingen motivering: 0p

   (c) Trädet är en min-heap då den uppfyller att roten är mindre än båda sina barn, och det samma kan sägas för varje nod i trädet utom löv, samt att trädet är komplett (det finns inga hål mellan lövnoder).
   Rätt svar och stark motivering: 2p. Rätt svar och svag motivering: 1p. Fel / ingen motivering: 0p

6. **Uppgift 7**

   (a) Exempel på lösning:
   9 1 3 6 11 2 5 8 10 7

   (b) Exempel på lösning:
   9 1 2 5 7 10 11 8 3 6