Linköpings Tekniska Högskola
Institutionen för Datavetenskap
Tommy Färnqvist

5 januari 2016

TDDC91/TDDI16 Datastrukturer och algoritmer
Datortentamen (DAT1)
2016-01-05, 8–12

Examinator: Tommy Färnqvist
Jour: Tommy Färnqvist (telefon 013-282479).
Max poäng: 111 poäng (betyg 5 = 109p, 4 = 106p, 3 = 100p)
Hjälpmedel: Inga hjälpmedel tillåtna, förutom OpenDSA!

VÄNLIGEN IAKTTAG FÖLJANDE

- Lösningar till olika problem skall placeras enkelsidigt på separata blad. Skriv inte två lösningar på samma papper.
- Sortera lösningarna innan de lämnas in.
- MOTIVERA DINA SVAR ORDENTLIGT: avsaknad av, eller otillräckliga, förklaringar resulterar i poängavdrag. Även felaktiga svar kan ge poäng om de är korrekt motiverade.
- Om ett problem medger flera olika lösningar, t.ex. algoritmer med olika tidskomplexitet, ger endast optimala lösningar maximalt antal poäng.
- SE TILL ATT DINA LÖSNINGAR/SVAR ÄR LÄSBARA.
- Lämna plats för kommentarer.

Lycka till!

1. OpenDSA (100 p)

Efter inloggning i datortentasystemet finns i startmenyn för Linux Mint:

- "OpenDSA" (öppnar URL för tentamensversion av OpenDSA i Chromium) och
- "Inloggningsuppgifter OpenDSA" (öppnar sida med inloggningsuppgifter till tentamensversion av OpenDSA i Chrome).

Starta tentamensversionen av OpenDSA, logga in med inloggningsuppgifterna enligt ovan och lös de anvisade uppgifterna. Genom att klicka på ditt inloggningsnamn kan du se i betygsboken hur många av de poänggivande uppgifterna du löst hittills. När du har full poäng i betygsboken är du färdig med den här uppgiften och kan logga ut. Du behöver inte skicka in något via tentamenssystemet. Om du inte löser samtliga poänggivande uppgifter får du noll poäng på den här tentamensuppgiften.

2. Datastrukturdesign (4 p)

En *LRU-cache* är en datastruktur som lagrar upp till N *distinkta* nycklar. Om datastrukturen är full när en nyckel som inte redan finns i cachen läggs till tar LRU-cachen först bort den nyckel som cachades för längst tid sedan (eng. *least reccently cached*).

Redogör för en datastruktur som understödjer följande gränssnitt:

`LRU(int N)`	skapa en tom LRU-cache med kapacitet N
`void cache(Key key)`	om det finns N nycklar i cachen och den givna nyckeln inte redan finns i cachen, så (i) ta bort nyckeln som varit oanvänd som argument till `cache()` längst tid och (ii) lägg till den givna nyckeln till LRU-cachen
`bool inCache(Key key)`	finns nyckeln i LRU-cachen?

Exempel:

```
LRU<String> lru(5);          // LRU cache (in order of when last cached)
lru.cache("A");              // A         (add A to front)
lru.cache("B");              // B A       (add B to front)
lru.cache("C");              // C B A     (add C to front)
lru.cache("D");              // D C B A   (add D to front)
lru.cache("E");              // E D C B A (add E to front)
lru.cache("F");              // F E D C B (remove A from back; add F to front)
bool b1 = lru.inCache("C"); // F E D C B (true)
bool b2 = lru.inCache("A"); // F E D C B (false)
lru.cache("D");              // D F E C B (move D to front)
lru.cache("C");              // C D F E B (move C to front)
lru.cache("G");              // G C D F E (remove B from back; add G to front)
lru.cache("H");              // H G C D F (remove E from back; add H to front)
bool b3 = lru.inCache("D"); // H G C D F (true)
```

Operationerna `cache()` och `inCache()` ska använda förväntad konstant tid under antagandet att eventuella hashfunktioner sprider ut nycklarna jämt över respektive tabeller.

Beskriv vilka datastrukturer du använder och hur LRU-cachens operationer är implementerade samt analysera deras tidskomplexitet. Visa också tillståndet för dina datastrukturer omedelbart efter sekvensen av operationer i exemplet ovan.

3. Divide and conquer (3 p)

Betrakta följande tre algoritmer:

- Algoritm 1 löser problem av storlek N genom att rekursivt dela upp dem i 2 delproblem av storlek $N/2$ och kombinera resultatet i tid c (där c är någon konstant).
- Algoritm 2 löser problem av storlek N genom att lösa ett delproblem av storlek $N/2$ rekursivt och utföra någon form av beräkning som använder tid c (där c är någon konstant).
- Algoritm 3 löser problem av storlek N genom att rekursivt dela upp dem i 2 delproblem av storlek $N/2$ och utföra en linjär mängd (dvs, cN där c är någon konstant) extra arbete.

(a) Skriv ned ett rekursivt uttryck $T(N)$ för varje algoritm som visar hur exekveringstiden för en instans av storlek N beror på exekveringstiden för en mindre instans. (1,5)

(b) Vad är den asymptotiska tidskomplexiteten för respektive algoritm? Motivera ditt svar kortfattat. (1,5)

4. Algoritmkonstruktion (4 p)

Beskriv en algoritm som, givet k sorterade arrayer innehållande totalt N nycklar, avgör om det finns någon nyckel som uppträder mer än en gång. Din algoritm ska använda $\mathcal{O}(k)$ extra minne i värsta fallet. För full poäng krävs att exekveringstiden för din algoritm är $\mathcal{O}(N \log k)$ i värsta fallet. Motivera tids- och minneskomplexiteten för din algoritm kortfattat.