Algoritmer, datastrukturer och komplexitet, hösten 2011

Uppgifter till övning 2

Datastrukturer och undre gränser

På denna övning är det också **inlämning av skriftliga lösningar av teoriuppgifterna till labb 1** och muntlig redovisning av teoriuppgifterna. Teoriuppgifterna redovisas individuellt, till skillnad från labben som görs i par.

Samsortering Beskriv en algoritm som samsorterar k stycken var för sig sorterade listor i tid $O(n \log k)$ där n är det totala antalet element.

Datastruktur med max och min Konstruera en datastruktur som har följande operationer och komplexitet:

- `insert(x)` (insättning av ett element) tar tid $O(\log n)$,
- `deletemin()` (borttagning av det minsta elementet) tar tid $O(\log n)$,
- `deletemax()` (borttagning av det största elementet) tar tid $O(\log n)$,
- `findmin()` (returnerar minsta elementet) tar tid $O(1)$,
- `findmax()` (returnerar största elementet) tar tid $O(1)$.

n är antalet element som för närvarande finns lagrade. Du får anta att man kan jämföra två element och ta reda på vilket som är störst i tid $O(1)$.

Bloomfilter med borttagning Det finns bara två tillåtna operationer på ett vanligt Bloomfilter: Insert och IsIn. Hur kan man modifiera datastrukturen så att den också tillåter operationen Remove, som inte får ta längre tid än Insert?

Suffixregeluppslagning i Stava Stava har cirka 1000 suffixregler av typen

-orna ⟵ *-a, -an, -or*

När ett ord som *dockorna* ska kontrolleras kommer Stava att i tur och ordning gå igenom alla suffix efter den första stavelsen, det vill säga *-ϵ, -a, -na, -rna, -orna, -korna, -ckorna*. Hur ska suffixreglerna lagras för att det ska gå snabbt att slå upp suffixen?

Undre gräns för sökning i sorterad array Binärsökning i en sorterad array med n tal tar som bekant tiden $O(\log n)$. Bevisa att $\Omega(\log n)$ också är en undre gräns för antalet jämförelser som krävs för detta problem (i värsta fallet).

Laga julgransbelysning Jonas julgransbelysning med n seriellt kopplade lampor lyser inte. Det räcker att en lampa är trasig för att belysningen inte ska lysa. Han har skaffat n nya lampor som säkert fungerar. Han vill inte ersätta några hela lampor med nya, för det är dumt att slösa på nya lampor. Han vill naturligtvis göra så få lampprovningar som möjligt.

Konstruera och analysera en effektiv algoritm för hur Jonas ska få belysningen att fungera med så få lampprovningar (ur- och iskruvningar) som möjligt. Analysera värstafalletkomplexiteten för algoritmen exakt (inte bara ordoklass). En urskruvning följt av en iskruvning av en lampa tar tiden 1. Ju snabbare algoritm desto fler poäng.

Ge också en undre gräns för värstafalletkomplexiteten som matchar din algoritm.

Lösningar och ledningar

Ledning till Samsortering
Metod 1: Använd en heap där ett element från varje lista finns med och där det minsta elementet ligger i roten. Varje heapoperation tar $O(\log k)$ och det görs n insättningar i och n borttagningar ur heapen.
Metod 2: Samsortera listorna parvis. Nu finns det bara $k/2$ sorterade listor. Samsortera dessa listor parvis, och så vidare. Efter $\log k$ samsorteringsomgångar finns det bara en lista kvar. Varje samsorteringsomgång tar tid $O(n)$ eftersom det är totalt n element som ska samsorteras i varje omgång.

Lösning till Datastruktur med max och min
Antingen kan man lösa uppgiften med hjälp av två trappor (heapar) med samma element, men där den ena är ordnad med minsta elementet i roten och den andra med största elementet i roten eller också med hjälp av ett balanserat sökträd som utökas med två extra variabler, en för det minsta värdet i trädet och en för det största.

Implementationerna är i båda fallen enkla. I binärträdsfallet är det trivialt att skriva `findmin` och `findmax`. För dom tre övriga operationerna använder man motsvarande vanliga balanserade sökträdsoperationer men efterbehandlar genom att kopiera dom minsta och största värdena i trädet till dom två variablerna. □

Lösning till Bloomfilter med borttagning
Låt Bloomfiltret bestå av tal (till exempel 8- eller 16-bits heltal) istället för bitar. Talet noll motsvarar bitvärdet 0 och varje positivt tal motsvarar bitvärdet 1. Istället för att sätta en bit till 1 så ökar Insert istället motsvarande tal med ett. Remove fungerar precis som Insert men minskar med ett istället för att öka med ett. Nu räknar heltalen hur många element som är hashade till varje position. När ett heltal kommit ner till noll så betyder det att inget element längre hashas till den positionen.

Storleken på heltalen måste avpassas till antalet element som beräknas ligga i Bloomfiltret. Man får nämligen problem om en räknare når heltalstypens tak. En möjlighet är att införa en extra bit per position som håller reda på om taket någon gång nåtts för den positionen. När en räknare med den biten satt kommer ner till noll så måste hela Bloomfiltret beräknas på nytt.

Ett alternativ är att den räknare som slår i taket får sitta fast i taket även om Remove anropas. Det betyder att den räknaren aldrig kommer att nå noll, även om alla element som hashas till den positionen tas bort. Det kommer förmodligen aldrig att inträffa eftersom det var så många element som hashats till den positionen, och om det inträffar så ökar sannolikheten för fel bara för dom ord som hashas dit, och det förmodligen inte så mycket. □

Lösning till Suffixregeluppslagning i Stava
Det man slår upp på är vänsterleden i reglerna. Vi kallar dessa för ingångssuffixen. Alla regler som har tomt ingångssuffix lagrar vi i en lista för sig. Övriga regler lagrar vi i en array sorterade efter ingångssuffixet läst baklänges. Vi använder latmannahashning på en bokstav, det vill säga vi har en bokstavsindexerad array som anger indexet till första ingångssuffixet som slutar på den bokstaven.

Vid sökning kollar man först i listan med regler med tomt ingångssuffix. Därefter slår man upp sista bokstaven (*a* i *dockorna*) med latmannahashning och går igenom dom regler som har den enda bokstaven som ingångssuffix. Sedan binärsöker man efter näst sista bokstaven (n) bland dom ingångssuffix som slutar med a och går igenom dom regler som har na som ingångssuffix. Därefter söker vi efter nästnäst sista bokstaven (r) bland dom ingångssuffix som slutar med na och går igenom dom regler som har rna som ingångssuffix och så vidare.

På detta sätt blir det färre och färre ingångssuffix att binärsöka bland. □

Ledning till Undre gräns för sökning i sorterad array
Konstruera ett beslutsträd för en godtycklig algoritm som söker i en sorterad array. Undersök hur många löv trädet måste ha och kom fram till hur högt det då måste vara.

Lösning till Laga julgransbelysning
Numrera julgransbelysningens lampplatser från 1 till n.

```
byt ut alla lampor utom den på plats 1 mot nya
lägg dom gamla lamporna i en säck
i ← 1
while (lampor kvar i säcken) do
      if (belysningen lyser) then
        i ← i + 1
        skruva ur den nya lampan på plats i
      else
          skruva ur den gamla trasiga lampan på plats i
      ta en lampa från säcken och skruva i den på plats i
if not (belysningen lyser) then
  skruva ur den gamla trasiga lampan på plats i
  skruva i en ny lampa på plats i
```

Det är lätt att se att algoritmen är korrekt med hjälp av följande invariant för slingan (invarianten är sann i början av varje varv i slingan): Alla lampor på plats $< i$ är gamla och fungerar, lampan på plats i är gammal, alla lampor på plats $> i$ är nya.

I värsta fall är alla lampor trasiga i belysningen, och då gör algoritmen $(n-1)+(n-1)+1 = 2n-1$ lampbyten.

Att $2n-1$ också är en undre gräns kan man inse så här: Enda sättet att avgöra ifall en viss lampa i belysningen är hel eller trasig är att försäkra sig om att alla andra lampor är hela och då se om belysningen lyser. En tuff motståndare ser till att belysningen fortfarande inte lyser efter att $n-1$ nya lampor skruvats i helt enkelt genom att låta den enda återstående originallampan vara trasig. Alla $n-1$ bortskruvade originallampor måste sedan provas igen, för den tuffa motståndaren har ännu inte tillåtit att någon som helst information om dessa lampor har getts, och det kräver $n-1$ lampbyten till. Slutligen ser motståndaren till att just den sist provade originallampan är trasig, och då krävs det ytterligare ett lampbyte för att fixa den.

Den tuffa motståndaren behöver alltså bara två trasiga lampor för att få algoritmen att ta $2n-1$ lampbyten! □