Datastrukturer och Algoritmer 2015-03-16 Lösningsskisser

Problem 1. *Småfrågor:* Avgör om vart och ett av följande påståenden är sant eller falsk eller svara på frågan. För att få poäng måste du ange en kortfattat *motivation* till ditt svar.

Jag har kompletterat svaren med en del bra studentsvar

a) se OH bilderna om hur man använder variablerna lastReturned, expectedModCount, modCount

Man använder en heltalsvariabek modCount som räknas upp varje gång strukturen modifieras. I iteratorn har man en expectedModCount som initieras till modCount när iteratorn skapas. Sedan kan man jämföra dessa varje gång iteratorn anropas.

b) Den empiriska komplexiteten är lika med ”klock”-tid dvs man implementerar sin algoritm och kör den och mäter tiden. Här påverkar saker som programmerarens skicklighet och vilket programspråk man använder samt datorns egenskaper.
Den teoretiska komplexiteten försöker bortse från dessa saker genom att bara titta på indatastorlek och algoritmen självt. Att bara svara att teoretisk beräknas och empirisk mäts vid provkörning är lite tunns svar tycker jag, jag vill ju veta skillnaden mellan dem och då bör man nämna att den teoretiska försöker bortse från vissa saker. Man skulle också kunna nämna att dom normalt skall ge samma asymptotiska resultat även om konstanterna kan skilja åt.

c) 4 gånger. $(2n)^2 = 4n^2$

d) Falskt. Komplexiteten ändras inte, den är alltid (n-1) + (n-2) + … Det går inte som i tex bubbelsort att lägga in nån test som gör att man kan sluta tidigare.

e) Falskt. Antalet är n(n-1)/2 dvs ((n-1) + (n-2) + ...+1)

f) Falskt. Med en dålig hashfunktion kan alla elementen hamna i en rak lista och då blir det O(n).

g) Sant. Trädet kan vara skevt. Sök tex efter alla bågar i stigande ordning och därefter efter minsta elementet. Eller fyll trädet med sorterade data.

h) Falskt, Dijkstra klarar inte negativa bågar. ”Det kanske inte finns några bågar/vägar” har många svarat men om det inte finns någon väg så returnerar Dijkstra att längden är oändlig vilket ju är helt rätt.

Problem 2. *Testar: hashtabeller*

a) De två vanligaste är att man lagrar kollisioner på annan plats i det befintliga fältet eller att man lagrar i länkade listor.

Ett axplock av fördelar/nackdelar (vissa kan vara både för och nackdelar, tex kan fast storlek vara bra – man vet ju precis hur mycket plats tabellen tar. Å andra sidan tar den alltid upp plats som om den var full.)

befintligt fält: ”fast” storlek, kan ge långa sökvägar vid kollisioner, svårigheter att hantera borttagna element,

listor: blir i princip inte full bara långsammare, pekarna tar plats, vanligen kortare sökvägar vid kollisioner, tar mindre plats när den innehåller få element,

se mer utförligt i boken / OH bilder

b) Hur ser hashfunktionerna för strängar och heltal ut i Java?

För heltal är det heltalet självt. För strängar är det, se OH för en mer detaljerad förklaring

```
for (int i = 0; i < len; i++) {
  h = 31*h + val[off++];
```

Problem 3. Lösningsskiss:

Använd facksortering dvs ha ett booleskt fält 0..5, gå igenom listan och ändra false i fältet till true. Gå sedan igenom fältet igen och leta efter den enda true markeringen. Komplxitet: 2 genomgångar av listan är O(n).

Det fungerar lika bra med heltal såklart bara man initialiserar fältet till nåt negativt.

Här finns fler lösningar,

- summan skall vara n(n+1)/2, summera alla elementen och subtrahera från den summan så har vi det saknade talet.

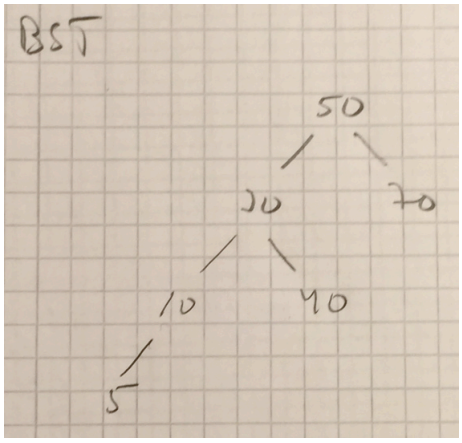
- någon har använd xor, kanske kommer den lösningen här.

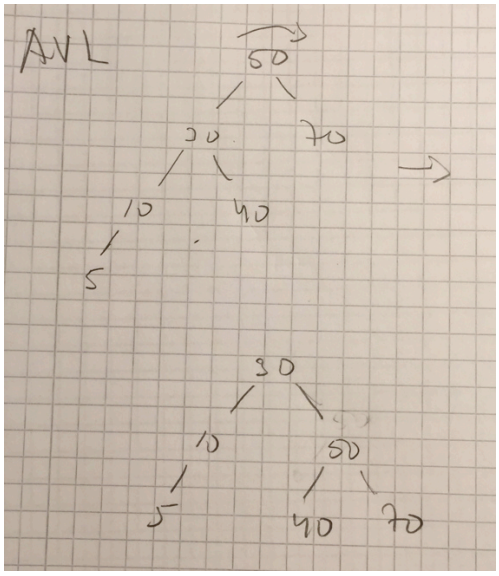
Problem 4. Testar: träd

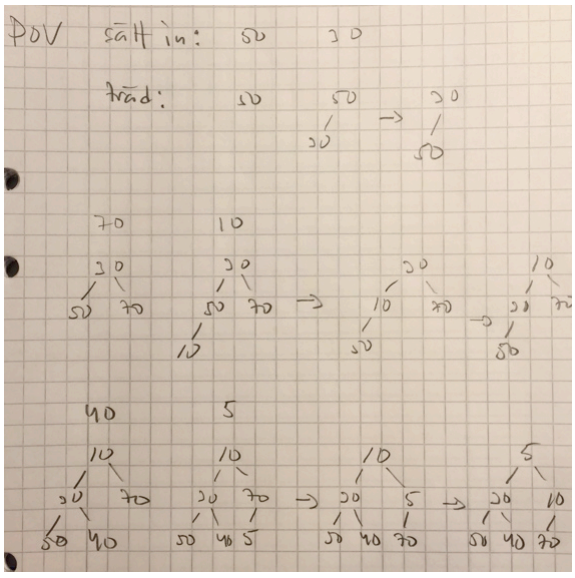
Lösningsskiss:

a) Egenskaper: se bok / OH-bilder.
Jag förväntar mig ganska korta svar typ ”ett AVL träd är ett balanserat BST som balanserar sig självt med hjälp av rotationer som görs när … osv.”

b)







Problem 5. Lösningsskiss:

Algoritm **med sortering**:

sortera med någon bra metod tex qs samt gå igenom listan en gång och ”räkna lika grannar” säg att det är n st, skriv sedan ut n(n-1)2 många dubletter. Dvs antag 4447777789.... Det är 3st 4or: skriv 3 x (4,4), sen är det 5st 7or: skriv 10 x (7,7) osv

Man kan också jämföra grannar, ungefär som min algoritm i uppgiften ovan, men vi kan ju gå vidare så fort vi hittar en icke dublett, se nedan.

Alla lika: kan ge snabbare sortering beroende på val av sorteringsmetod. Idealiskt vore en variant av qs som slutar om alla är lika, tex den vi gått igenom. Då blir sorteringen O(n). Tyvärr blir genomgången/utskriften dyrbar när alla är lika så sorteringen spelar ingen roll här. Om vi har n indata så skall vi ju skriva n(n-1)/2 dubletter och det blir ju $O(n^2)$. Totalt $n+n^2 = O(n^2)$.

Alla olika: då tar sorteringen O(nlogn) men genomlöpningen sker snabbt. Om vi har listan {4,5,6,7} så löper vi igenom en gång dvs O(n) jämf. Totalt nlogn + n = O(nlogn).

Några dubletter: tex listan ovan efter sortering {4,4,4,5,6,7,7}. Detta ligger mellan nlogn och n^2 beroende på antalet dubletter.

Algoritm **med binärt sökträd**:

Låt noderna i trädet innehålla dels talet vi sätter in och dels en räknare som anger hur många av talet vi satt in. Detta behövs för att hålla reda på multipla dubletter tex för 4 ovan. När första 4:an kommer så sätts den bara in. Andra fyran skall ge *en* utskrift (räknaren är 1) och ökar räknaren till 2. Tredje 4:an skall ge *två* utskrifter (räknaren är 2) och ökar räknaren osv. Skriver vi direkt, som i algoritmen nedan, så kommer utskrifterna i oordning men vi kan ju också fylla trädet först och sedan gå igenom det och skriva i efterhand vilket nog är bättre.

Sätt in talen i trädet med en ”insert” som reagerar på dubletter enligt ovan. Antag att talen finns i fältet `arr` och att trädet finns i `t`, n = arr.length

```
for i=1 to n insert(arr(i), t)
```

där insert ser ut så här

```
insert (int e, tree t)
    if t == null
        sätt in ny nod i trädet, räknaren = 1
    else if e == t.rot
        dublett funnen, skriv ut dubletten “räknaren” gånger
        öka räknaren
    else if e<t.value
        insert(e, t.left)
    else
        insert(e, t.right)
    end if
end insert
```

Alla lika: Då sker inga rekursiva anrop utan man kommer bara till roten, O(n) men det blir många utskrifter, $(n^2-n)/2=O(n^2)$

Alla olika: här måste vi se till två fall

fall 1: indata sorterat eller nästan sorterat => obalanserat träd => $O(n^2)$.

fall 2: osorterat: => troligen välbalanserat träd => nlogn

Några dubletter: för slumpmässiga träd blir det O(nlogn) eftersom utskrifterna är få. Sorterade får samma problem som ovan.

Problem 6. **Lösningsskiss**:

Idé / pseudokod

För alla noder v

Beräkna kortaste vägar från v till alla i `thePlaces i` tur och ordning med varianten på Dijkstra från labben.

och spara undan summan av alla dessa vägarna, detta är kostnaden för orten v

Tag sedan den nod som har minsta summan.

Komplexitet:

thePlaces kan vara n (där n är antalet noder i grafen) så får vi n^2 anrop av shortestPath som har komplexiteten (e*loge * kopiera listan) där e är antalet bågar. Totalt $O(n^2*e^2*loge)$ men det är i överkant. Det kostar ju tex inte e att kopiera listan varje gång och man kan ju anta att thePlaces innehåller mindre än n deltagare.

```
// ====================================
public int findMostCentralPlace(List<Integer> thePlaces) {
    int    minNodeSoFar = -1;
    double minCostSoFar = Double.MAX_VALUE;
    // gå igenom alla noder i grafen och beräkna deras
    // kostnad till dom andra
    for ( int possiblePlace = 0;
                    possiblePlace<neighbours.length;
                        possiblePlace++ ) {
        double cost = 0;
        // gå igenom alla orter med deltagare
        for ( int place : thePlaces ) {
            cost = cost + getTotalPathCost(
                    shortestPath(place, possiblePlace ));
        }
        if ( cost < minCostSoFar ) {
            minNodeSoFar = possiblePlace;
            minCostSoFar = cost;
        }
    }
    return minNodeSoFar;
} // end findMostCentralPlace

private double getTotalPathCost( Iterator<E> path ) {
    double cost = 0.0;
    while ( path.hasNext() ) {
        cost = cost + path.next().getWeight();
    }
    return cost;
}
}
```