

Tentamen

Datastrukturer (DAT036)

- Datum och tid för tentamen: 2011-12-16, 8:30–12:30.
- Ansvarig: Nils Anders Danielsson. Nås på 0700 620 602 eller anknytning 1680. Besöker tentamenssalarna ca 9:30 och ca 11:30.
- Godkända hjälpmedel: Kursboken (Data Structures and Algorithm Analysis in Java, Weiss, valfri upplaga), handskrivna anteckningar.
- För att få betyget n (3, 4 eller 5) måste du lösa minst n uppgifter. Om en viss uppgift har deluppgifter med olika gradering så räknas uppgiften som löst om du har löst *alla* deluppgifter med gradering n eller mindre.
- För att en uppgift ska räknas som “löst” så måste en i princip helt korrekt lösning lämnas in. Enstaka mindre allvarliga misstag kan *eventuellt* godkännas (kanske efter en helhetsbedömning av tentan; för högre betyg kan kraven vara hårdare).
- Lämna inte in lösningar för flera uppgifter på samma blad.
- Skriv din tentakod på varje blad.
- Lösningar kan underkännas om de är svårlästa, ostrukturerade eller dåligt motiverade. Pseudokod får gärna användas, men den får inte utelämnas för många detaljer.
- Om inget annat anges så kan du använda kursens uniforma kostnadsmodell när du analyserar tidskomplexitet (så länge resultaten inte blir uppenbart orimliga).

1. (*Lätt.*) Skriv ett effektivt program som tar en lista av strängar, beräknar varje strängs frekvens (hur många gånger den förekommer i listan), och skriver ut strängarna sorterade efter frekvens (i fallande ordning). Om två strängar har samma frekvens spelar ordningen ingen roll. Exempel:

- Lista: "elefant", "giraff", "lodjur", "elefant", "elefant"
- Utskrift:

```
elefant
lodjur
giraff
```

Förklara implementationen, och analysera programmets tidskomplexitet. Din analys ska ta hänsyn både till listans längd n , och antalet tecken w i den längsta strängen. Var tydlig med vilka antaganden du gör i din analys. Standarddatastrukturer och -algoritmer från kursen behöver inte förklaras, men däremot motiveras. Om pseudokod används måste den vara detaljerad.

2. (*Lätt.*) Betrakta följande Javaklass:

```
// Trädnode med föräldrapekare.
//
// Invariant: Om this.parent inte är null så gäller antingen
//   this.parent.left = this
// eller
//   this.parent.right = this.
public class Node<A> {
    public A contents;      // Innehåll.
    public Node<A> left;   // Vänster delträd.
    public Node<A> right;  // Höger delträd.
    public Node<A> parent; // Förälder.
}
```

Är följande implementation av enkelrotation korrekt? Motivera (gärna med figurer), och om något är fel, visa hur man kan åtgärda felet. (Eventuella problem har inget med Java att göra. Koden kompilerar utan problem.)

```

// Roterar upp noden child förbi sin förälder.
//
// Noden child samt dess förälder och förälderns förälder
// måste vara icke-null.
public static <A> void rotateUp(Node<A> child) {
    // Förfäder.
    Node<A> parent      = child.parent;
    Node<A> grandparent = parent.parent;

    // Uppdatera föräldrapekare.
    child.parent = grandparent;
    parent.parent = child;

    // Uppdatera barnpekare.
    if (parent.left == child) {
        parent.left = child.right;
        child.right = parent;
    } else {
        parent.right = child.left;
        child.left  = parent;
    }

    // Uppdatera pekaren från "farföräldern".
    if (grandparent.left == parent) {
        grandparent.left = child;
    } else {
        grandparent.right = child;
    }
}
}

```

3. Uppgiften är att konstruera en datastruktur som implementerar en variant av avbildnings-ADTn ("map-ADTn").

Anta att nycklar är totalt ordnade och att det finns en komparator som avgör hur två nycklar k_1 och k_2 är relaterade ($k_1 < k_2$, $k_1 = k_2$ eller $k_1 > k_2$). ADTn har följande operationer:

empty: Skapar en tom avbildning.

insert(k, v): Lägger till en bindning $k \mapsto v$ till avbildningen. Om en bindning $k \mapsto v'$ redan finns i avbildningen så tas den bort.

lookup(k): Om en bindning $k \mapsto v$ finns i avbildningen så ger den här operationen v som svar, och annars meddelas på något lämpligt sätt att en sådan bindning saknas.

remove-smaller(k): Tar bort alla bindningar $k' \mapsto v$ där $k' < k$. Övriga bindningar påverkas inte.

Den här uppgiften har graderade deluppgifter:

För trea: Implementera ovanstående ADT, förklara hur din implementation fungerar, och analysera operationernas tidskomplexitet. Standarddatastrukturer och -algoritmer från kursen behöver inte beskrivas i detalj. För att bli godkänd måste du se till att operationerna har följande tidskomplexiteter (antingen i värsta fall, eller amorterat):

empty: $O(1)$.

insert: $O(\log n)$.

lookup: $O(\log n)$.

remove-smaller: $O(n \log n)$.

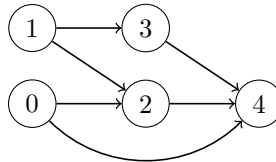
Här är n antalet bindningar i avbildningen. Du kan anta att element kan jämföras på konstant tid.

För fyra: Som för trea, men **remove-smaller** ska ha tidskomplexiteten $O(\log n)$.

4. Man kan sortera riktade acykliska grafer (DAGs) topologiskt genom att göra en djupet först-sökning och pusha noderna på en stack i postordning.

För trea: Implementera den här algoritmen. Skriv en funktion (metod) som tar en graf och ger tillbaka en lista med grafens noder i topologisk ordning. Exempel:

- Graf:



- Möjligt resultat: 0, 1, 2, 3, 4.

Var tydlig med hur du representerar grafer, och hur djupet först-sökningen utförs. Du behöver inte beskriva implementationer av stackar och listor i detalj.

För fyra: Analysera din implementations tidskomplexitet (noggrant, det räcker inte att skriva “djupet först-sökning tar si och så lång tid”, du måste analysera din egen implementation).

5. (*Svår.*) Betrakta följande variant av list-ADTn:

empty: Skapar en tom lista.

insert(x): Läger till x sist i listan.

delete-last: Tar bort och ger tillbaka listans sista element. Krav: Listan får inte vara tom.

Man kan implementera den här ADTn med en dynamisk array på följande sätt:¹

empty: Allokerar en array med längd ett.

insert(x): Om arrayen är full: fördubbla arrayens storlek. Stoppa in x på den första lediga positionen.

delete-last: Ta bort det sista elementet från arrayen. Om arrayens längd c är minst 4, och den innehåller $c/4$ element, halvera arrayens storlek. Ge tillbaka det sista elementet.

Notera att arrayens längd hela tiden är 2^k för något $k \in \mathbb{N}$.

Din uppgift är att bevisa att alla operationer tar amorterat konstant tid.

¹Notera att den här pseudokoden *inte* är detaljerad.

6. (*Svår.*) Handelsresandeproblemet kan formuleras så här: Givet en komplett,² viktad, oriktad graf, hitta en kortaste enkel cykel. Med andra ord, hitta en väg som besöker alla noder exakt en gång, förutom att den börjar och slutar i samma nod, och som är minst lika kort som alla andra sådana vägar.

Lös följande variant av handelsresandeproblemet *effektivt*:

- Kanternas vikter är icke-negativa. Du kan anta att de är naturliga tal.
- Grafen uppfyller triangelolikheten: Låt $d(u, v)$ stå för vikten av kanten mellan noderna u och v . I så fall gäller för alla noder u, v och w att $d(u, w) \leq d(u, v) + d(v, w)$.
- Du behöver inte hitta en kortaste enkel cykel, det räcker att hitta en enkel cykel som är max dubbelt så lång som de kortaste.

Standarddatastrukturer och -algoritmer från kursen behöver inte förklaras. Motivera utförligt varför din lösning är korrekt och effektiv.

Tips: Använd ett minsta uppspannande träd.

²I en komplett graf finns det, för varje par av noder, en kant mellan noderna.

Lösningförslag för tentamen i Datastrukturer (DAT036) från 2011-12-16

Nils Anders Danielsson

1. Låt oss benämna indatalistan `strängar`.

Vi kan börja med att beräkna varje strängs frekvens genom att använda en hashtabell som mappar strängar till frekvenser:

```
frekvenser ← new hashtabell
för varje sträng s i strängar
  om s → f finns i frekvenser
    frekvenser.insert(s, f + 1)
  annars
    frekvenser.insert(s, 1)
```

Låt oss säga att listan innehåller n strängar, och att den längsta strängen innehåller w tecken. Det är då rimligt att anta att hashfunktionen kan beräknas på $O(w)$ steg. Om vi vidare antar att hashfunktionen är perfekt¹ så tar insättning/uppslagning i hashtabellen också $O(w)$ tidsenheter (amorterat), och koden ovan tar totalt $O(wn)$ tidsenheter.

Låt oss nu extrahera en lista med par (s, f) av strängar och frekvenser:

```
sträng-frekvens-par ← frekvenser.list-with-bindings()
```

Den här konverteringen tar $O(n)$ tidsenheter, om hashtabellens lastfaktor $\lambda \geq c$ för någon konstant c oberoende av n .

Nu kan vi sortera listan `m` a p andra komponenten, i fallande ordning:

```
sträng-frekvens-par.sort(descending)
```

Här är `descending` en komparator som, då $p_1 = (s_1, f_1)$ och $p_2 = (s_2, f_2)$ jämförs, säger " $p_1 < p_2$ " om $f_1 > f_2$, " $p_1 = p_2$ " om $f_1 = f_2$, och " $p_1 > p_2$ " om $f_1 < f_2$. Sorteringen kan utföras med $O(n \log n)$ jämförelser, och jämförelserna tar konstant tid (givet att vi använder en uniform kostnadsmodell), så tidskomplexiteten blir $O(n \log n)$.

¹Knappast realistiskt, men förhoppningsvis inte alltför långt från sanningen med realistiska indata.

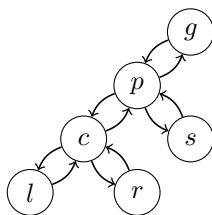
Till sist kan vi skriva ut frekvenserna:

för varje par (s, f) i sträng-frekvens-par
skriv ut s följt av en radbrytning

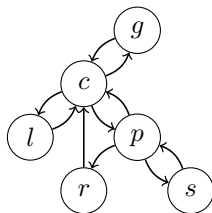
Om vi antar att tiden för en utskrift är $O(c)$, där c är antalet tecken i strängen, så är tidskomplexiteten för utskriften $O(wn)$.

Den totala tidskomplexiteten blir $O(n(w + \log n))$.

2. Implementationen är inte korrekt. Betrakta följande situation:



Efter ett anrop av `rotateUp(c)` får vi följande situation:



Notera att vi inte har uppdaterat föräldrapekaren för noden r , vilket betyder att invarianten för `Node` är bruten: $r.\text{parent} \neq \text{null}$, men ändå har vi varken $r.\text{parent}.\text{left} = r$ eller $r.\text{parent}.\text{right} = r$.

Man kan åtgärda felet genom att alltid uppdatera föräldrapekare och barnpekare samtidigt (se inlämningsuppgift 6.3 från 2011 års upplaga av kursen).

3. Man kan implementera ADTn med hjälp av ett splayträd. Operationerna `empty`, `insert` och `lookup` är de vanliga splayträdoperationerna, och har de vanliga tidskomplexiteterna (amorterat):

- `empty`: $O(1)$.
- `insert`: $O(\log n)$.
- `lookup`: $O(\log n)$.

Det återstår att implementera `remove-smaller(k)`. Vi kan göra det på följande sätt:

- Försök hitta minsta nyckeln större än eller lika med k .
- Om det inte finns någon sådan nyckel så är alla nycklar mindre än k , och då kan vi avsluta genom att skapa ett nytt, tomt träd.
- Om det finns någon sådan nyckel, splaya upp motsvarande nod till trädets rot, och radera den nya rotens vänstra delträd (som nu innehåller alla nycklar mindre än k).

Med detaljerad rekursiv² pseudokod:

```
// Hittar den nod, i delträdet vars rot är r, som har den
// minsta nyckeln som är större än eller lika med k. Om
// det inte finns någon sådan nod ges istället null som
// svar. Roten r får vara null.
Node find-closest-larger-key(Key k, Node r) {
    if (r == null) {
        return null;
    }

    if (r.key == k) {
        return r;
    } else if (r.key < k) {
        return find-closest-larger-key(k, r.right);
    } else { // r.key > k
        Node n = find-closest-larger-key(k, r.left);
        if (n == null) return r;
        else return n;
    }
}

remove-smaller(k) {
    // Antag att trädklassen innehåller exakt en
    // tillståndsvariabel, root, som pekar på rotnoden.
    Node larger = find-closest-larger-key(k, root);
    if (larger == null) {
        root = null;
    } else {
        splay(larger);
        larger.left = null;
    }
}
```

²En iterativ implementation kan vara bättre, eftersom den undviker problem med "stack overflow".

Hur effektiv är implementationen ovan? Analysen av splaying visar att en splayträdsoperation som innefattar sökning efter en nod och sedan splaying av den noden till trädets topp (plus $O(1)$ extra arbete för varje nod längs sökvägen) har tidskomplexiteten $O(\log n)$ (amorterat, och givet att jämförelser tar konstant tid). I `remove-smaller` utförs inte splayingen om `find-closest-larger-key` inte hittar någon lämplig nod, men det gör inget: Vi skulle kunna ändra koden ovan så att den splayar upp den sista noden som besöktes även om ingen lämplig nod hittas, men eftersom trädet raderas i nästa steg skulle det inte påverka slutresultatet – koden uppför sig *som om* splayingen faktiskt utfördes.

Raderingen av noder med små nycklar (genom att sätta `larger.left` eller `root` till `null`) tar konstant tid och ökar inte trädets potential, så den amorterade tidskomplexiteten är $O(1)$.

Slutsatsen blir att den amorterade tidskomplexiteten för `remove-smaller` är $O(\log n)$.

4. Grafrepresentation: Noder är numrerade från 0 till $n - 1$, där n är antalet noder. Grafstrukturen representeras med grannlistor: en array med storlek n , där n är antalet noder, och där position i innehåller en länkad lista med nod i s direkta efterföljare.

Vi kan implementera algoritmen i uppgiftsspecifikationen på följande sätt (pseudokod):

```
List<Integer> topological-sort(Graph g) {
    // Stacken.
    List<Integer> stack = new Stack<Integer>;

    // Array som visar om en viss nod besökts.
    Boolean visited[] = new Boolean[g.number-of-nodes];

    // Initialisering av arrayen.
    for (Integer i = 0; i < g.number-of-nodes; i++)
        visited[i] = false;

    // En lokal procedur med tillgång till variablerna
    // stack och visited.
    void dfs(Integer i) {
        visited[i] = true; // (A)

        for all immediate successors j of i in g { // (B)
            if (not visited[j])
                dfs(j);
        }

        stack.push(i); // (C)
    }
}
```

```

// Djupet först-sökning (D).
for (Integer i = 0; i < g.number-of-nodes; i++) {
    if (not visited[i])
        dfs(i);
}

// Den topologiskt sorterade listan.
return stack;
}

```

Tidskomplexitetsanalys (givet en graf med noder V och kanter E):

- Allokering av stack: $O(1)$.
- Allokering och initialisering av array: $O(|V|)$.
- Proceduren `dfs` anropas exakt en gång för varje nod, så raderna `A` och `C` körs båda $|V|$ gånger, och tar konstant tid varje gång: $O(|V|)$.
- Loopen `Bs` kropp körs en gång för varje kant, och varje iteration tar konstant tid (om man inte räknar tiden för `dfs(j)`): $O(|E|)$.
- Loopen `Ds` kropp tar också konstant tid (om man inte räknar tiden för `dfs(i)`): $O(|V|)$.

Totalt: $O(|V| + |E|)$.

5. Låt oss använda potentialmetoden med följande potentialfunktion (där c är arrayens längd, eller kapacitet, och n är listans längd):

$$\Psi(c, n) = k|c - 2n|.$$

Tanken är att potentialen är minst när arrayen är halvfull, och ökar ju närmare vi kommer en fördubbling/halvering av storleken; k är en positiv konstant vars värde förhoppningsvis faller ut från analysen nedan.

Som ett specialfall kan vi definiera att potentialen är 0 innan vi kört `empty`.

Är potentialfunktionen OK? Ja, i och med att potentialen är icke-negativ så är ursprungspotentialen mindre än eller lika med alla möjliga potentialer.

Låt oss nu analysera operationerna:

- `empty`: Tar konstant tid, och ökar potentialen från 0 till 1. Den amorterade tidskomplexiteten är alltså $O(1)$.
- `insert`: Om vi inte fördubblar arrayen så tar `insert` konstant tid, och potentialen ökar med

$$\Psi(c, n + 1) - \Psi(c, n) = k|c - 2(n + 1)| - k|c - 2n|,$$

där c och n är kapaciteten och längden *innan* vi satt in det nya elementet. En enkel fallanalys ($c \leq 2n$, $c = 2n + 1$ och $c \geq 2(n + 1)$)

visar att ökningen är som mest $2k$. Den amorterade tidskomplexiteten är då som mest $O(1) + 2k$, vilket är $O(1)$ eftersom k är en konstant. Om vi fördubblar arrayens kapacitet från c till $2c$ så är tidskomplexiteten av **insert** $O(c)$ (vi kopierar c element och sätter in ett nytt). Potentialförändringen är

$$\Psi(2c, c+1) - \Psi(c, c) = k|2c - 2(c+1)| - k|c - 2c| = -k(c-2).$$

Den amorterade tidskomplexiteten är $O(c) - k(c-2)$, vilket är $O(1)$ om k kan väljas tillräckligt stor.

- **delete-last**: Borttagningsoperationen kan analyseras på motsvarande sätt. Om kapaciteten inte halveras är den amorterade tidskomplexiteten (notera att $n \geq 1$)

$$\begin{aligned} O(1) + \Psi(c, n-1) - \Psi(c, n) &= \\ O(1) + k|c - 2(n-1)| - k|c - 2n| &\leq \\ O(1) + 2k &= \\ O(1). \end{aligned}$$

Om kapaciteten halveras är den amorterade tidskomplexiteten (notera att c är en tvåpotens och $c \geq 4$)

$$\begin{aligned} O(c) + \Psi\left(\frac{c}{2}, \frac{c}{4}\right) - \Psi\left(c, \frac{c}{4} + 1\right) &= \\ O(c) + k\left|\frac{c}{2} - 2\frac{c}{4}\right| - k\left|c - 2\left(\frac{c}{4} + 1\right)\right| &= \\ O(c) - \frac{3}{2}kc. \end{aligned}$$

Här får vi återigen att den amorterade tidskomplexiteten är $O(1)$ om k kan väljas tillräckligt stor.

Ma o blir den amorterade tidskomplexiteten för alla operationer $O(1)$ om vi bara låter k vara tillräckligt stor.

6. Algoritm (för grafer med minst två noder):

- Beräkna ett minsta uppspännande träd T . Ett sådant träd existerar eftersom grafen är sammanhängande.
- Gör en djupet först-sökning i T (inte grafen), med början i en godtycklig nod v , och lägg varje nod sist i en länkad lista p första gången noden besöks. Om man sedan lägger till v sist i listan så representerar p en enkel cykel (eftersom grafen är komplett). Den här cykeln är algoritmens resultat.

Korrekthet Observera först att trädets totala vikt inte kan vara större än den kortaste enkla cykelns vikt w , för om man tar bort en kant från cykeln så får man ett uppspannande träd, och alla kanter har icke-negativa vikter. Notera också att djupet först-sökningen i algoritmens andra steg följer varje kant i T exakt³ två gånger. Beteckna motsvarande väg med p' . Den här vägens totala vikt är max $2w$.

Allt som återstår är att visa att vägen p inte är tyngre/längre än p' . Vägen p har i princip konstruerats genom att utgå från p' och byta ut vissa vägsnuttar v_1, v_2, \dots, v_i ($i \geq 3$) mot "genvägar" v_1, v_i (eftersom noderna v_2, \dots, v_{i-1} redan hade besökts). Triangelolikheten medför att de här genvägarna aldrig är längre än ursprungsvägsnuttan:

$$d(v_1, v_i) \leq d(v_1, v_2) + d(v_2, v_i) \leq \dots \leq \sum_{j=1}^{i-1} d(v_j, v_{j+1}).$$

Alltså är p max lika lång som p' .

Effektivitet Antag att grafen har n noder. Då har den $\theta(n^2)$ kanter (eftersom den är komplett).

Beräkning av minsta uppspannande träd med Prims algoritm tar $O(n^2)$ steg (om man väljer rätt variant av algoritmen, och givet att det tar konstant tid att jämföra två vikter), och djupet först-sökningen tar $O(n)$ steg (eftersom trädet innehåller $n - 1$ kanter och vi bara gör konstant mycket extra arbete per nod och kant). Alltså är tidskomplexiteten $O(n^2)$, vilket är effektivt eftersom grafen innehåller $\theta(n^2)$ kanter.

³Eftersom T är ett träd.