Extraföreläsning Datastrukturer DAT036

Nils Anders Danielsson

2012-03-16

Uppgift 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.

Uppgift 1

Exempel:

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

```
elefant
lodjur
giraff
```

Uppgift 1

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.

Lösning 1

”beräknar varje strängs frekvens”:

```
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)
```

Antagande: Perfekt hashfunktion ($O(w)$).

$$O(1) + nO(w) = O(nw)$$

Lösning 1

"sorterade efter frekvens":

```
sträng-frekvens-par =
  frekvenser.list-with-bindings()
sträng-frekvens-par.sort(descending)
```

$$\texttt{descending}((s_1, f_1), (s_2, f_2)) = \begin{cases} <, & f_1 > f_2, \\ =, & f_1 = f_2, \\ >, & f_1 < f_2 \end{cases}$$

Antagande: Lastfaktor $\geq$ konstant oberoende av n.

$$O(n) + O(n \log n) = O(n \log n)$$

Lösning 1

"skriver ut strängarna":

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

$\leq nO(w) = O(nw)$

Lösning 1

Totalt:

$$O(nw)+O(n\log n)+O(nw) = O(n(\log n+w))$$

Uppgift 2

(Lätt.) Betrakta följande Javaklass:

```
// Trädnoder 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.
}
```

Uppgift 2

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.)

Uppgift 2

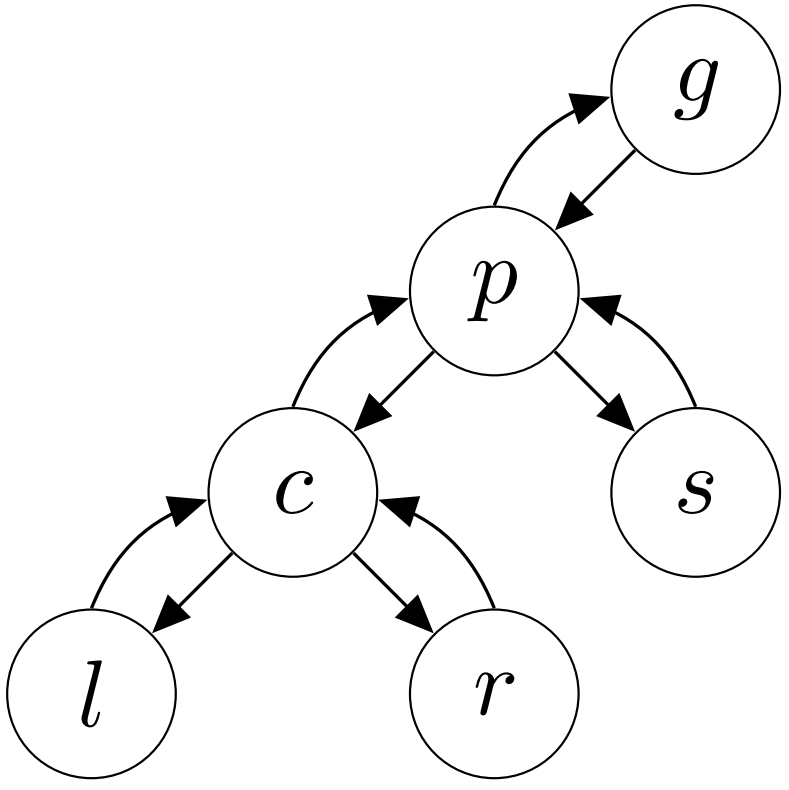
```
// Roterar upp noden child förbi sin förälder.
//
// Noden child samt dess förälder och förälderns
// förälder måste vara icke-null.
public static <A> void rotateUp(Node<A> child) {
    // Förfäder.
    Node<A> parent      = child.parent;
    Node<A> grandparent = parent.parent;

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

Uppgift 2

```
        // 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;
        }
    }
```

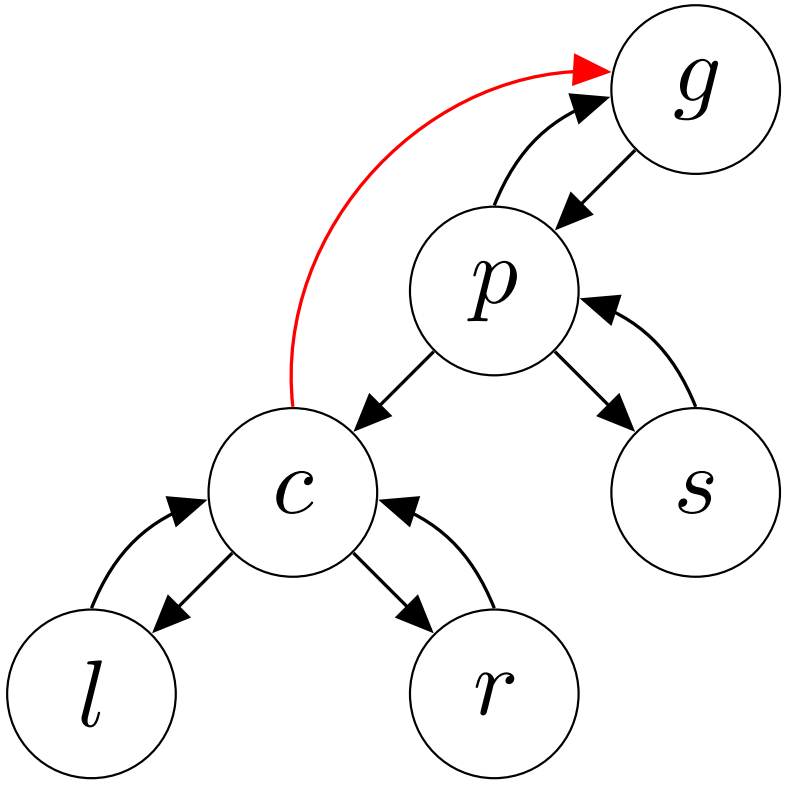
Lösning 2

```
child.parent  = grandparent;
parent.parent = child;

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

if (grandparent.left == parent) {
    grandparent.left  = child;
} else {
    grandparent.right = child;
}
```



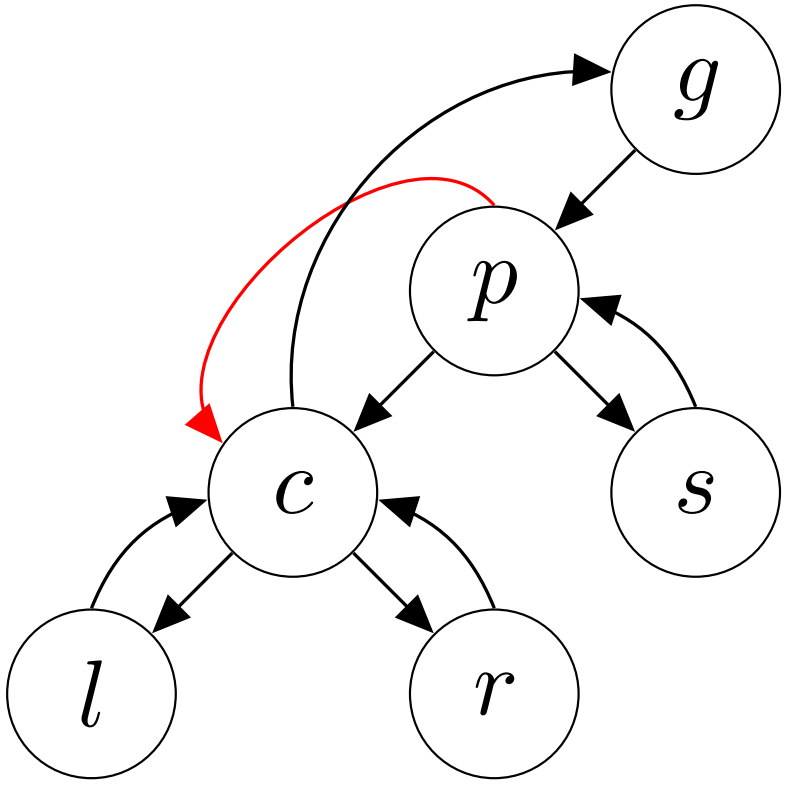
Lösning 2

```
child.parent  = grandparent;
parent.parent = child;

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

if (grandparent.left == parent) {
    grandparent.left  = child;
} else {
    grandparent.right = child;
}
```



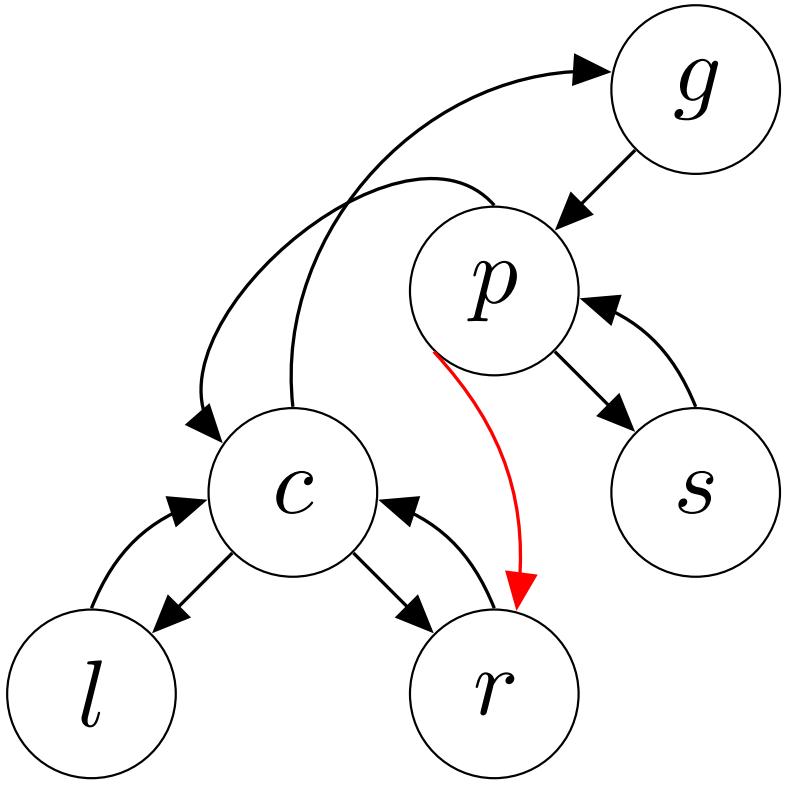
Lösning 2

```
child.parent  = grandparent;
parent.parent = child;

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

if (grandparent.left == parent) {
    grandparent.left  = child;
} else {
    grandparent.right = child;
}
```



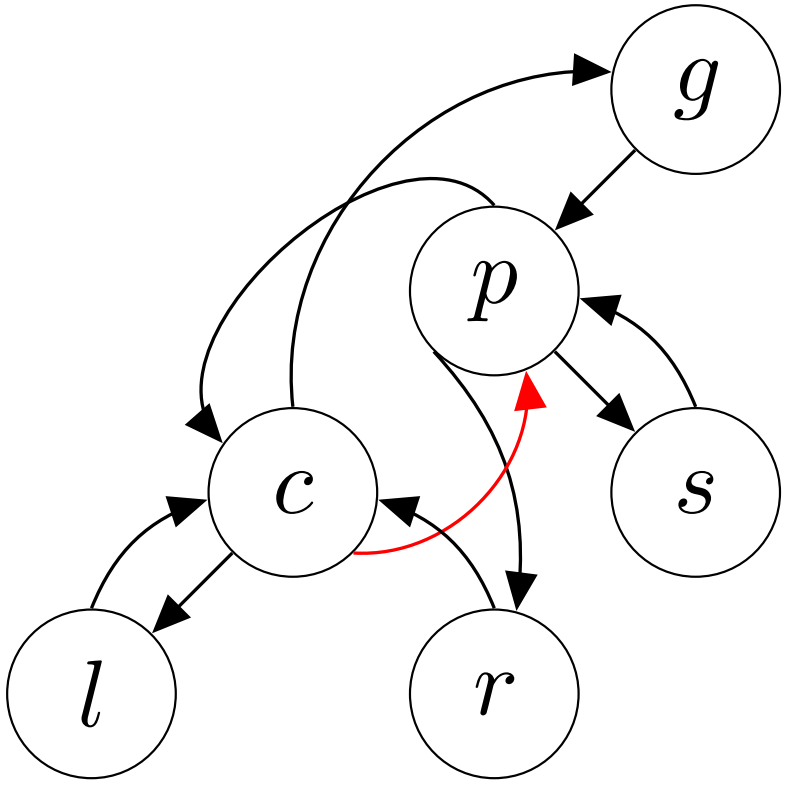
Lösning 2

```
child.parent  = grandparent;
parent.parent = child;

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

if (grandparent.left == parent) {
    grandparent.left  = child;
} else {
    grandparent.right = child;
}
```



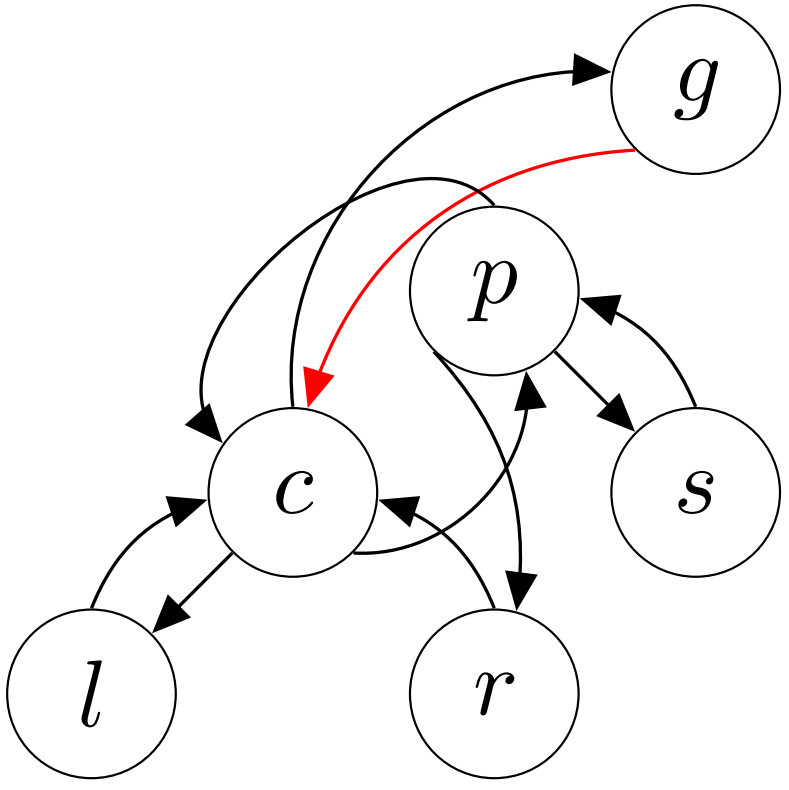
Lösning 2

```
child.parent  = grandparent;
parent.parent = child;

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

if (grandparent.left == parent) {
    grandparent.left  = child;
} else {
    grandparent.right = child;
}
```



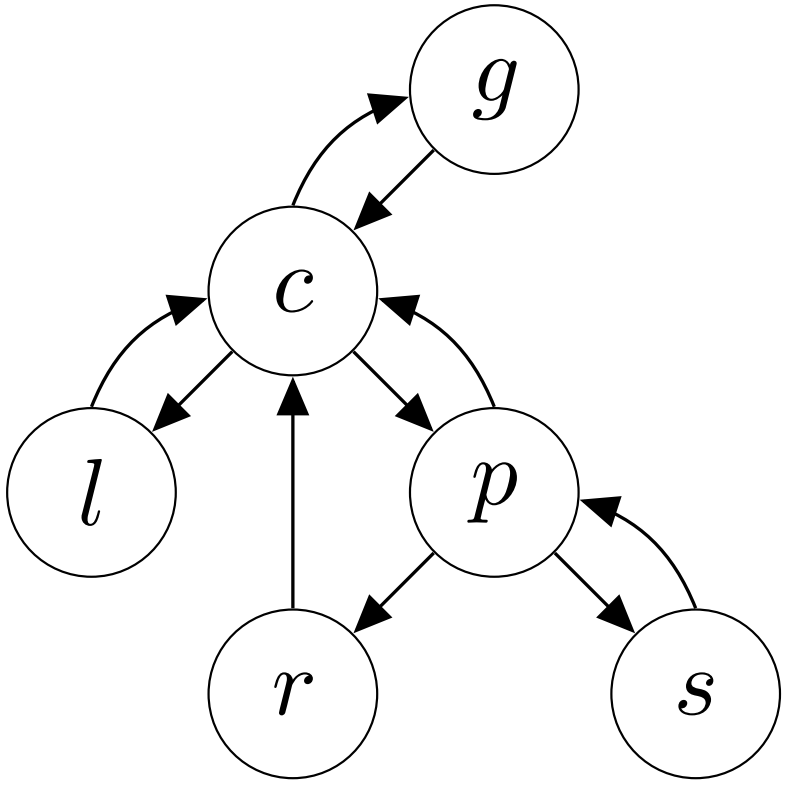
Lösning 2

```
child.parent  = grandparent;
parent.parent = child;

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

if (grandparent.left == parent) {
    grandparent.left  = child;
} else {
    grandparent.right = child;
}
```



Lösning 2

```
child.parent  = grandparent;
parent.parent = child;

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

if (grandparent.left == parent) {
    grandparent.left  = child;
} else {
    grandparent.right = child;
}
```



Lösning 2

Uppdatera föräldrapekare och barnpekare samtidigt:

```
public void setLeft(Node child) {
  left = child;
  if (child != null) {
    child.parent = this;
  }
}

public void setRight(Node child) {
  right = child;
  if (child != null) {
    child.parent = this;
  }
}
```

Lösning 2

```
if (parent.left() == child) {
  parent.setLeft(child.right());
  child.setRight(parent);
} else {
  parent.setRight(child.left());
  child.setLeft(parent);
}

if (grandparent.left() == parent) {
  grandparent.setLeft(child);
} else {
  grandparent.setRight(child);
}
```

Uppgift 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 $k_1 > k_2$).

Uppgift 3

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.

Uppgift 3

Den här uppgiften har graderade deluppgifter:

Uppgift 3

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`, `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.

Uppgift 3

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

Lösning 3

För trea:

- Dra nytta av $O(n \log n)$:
  Kan t ex konstruera nytt balanserat träd.
- Använd AVL-träd.
  (Klass med en tillståndsvariabel `map`.)
- `empty`, `insert`, `lookup`: De vanliga AVL-trädsoperationerna.
- `remove-smaller`: Implementera med hjälp av avbildnings-ADTn.

Lösning 3

```
remove-smaller(k) {
  List bindings = new LinkedList<Pair<Key, Value>>;
  for all k' → v in map {
    if (not (k' < k)) {
      bindings.add((k', v));
    }
  }
  map = empty();
  for all (k, v) in bindings {
    map.insert(k, v);
  }
}
```

$$\leq O(1)+nO(1)+O(1)+nO(\log n) = O(n \log n)$$

Lösning 3

För fyra:

- Kan använda splayträd.
- Kan använda skipplistor.
  (Dock förväntad komplexitet istället för amorterad.)

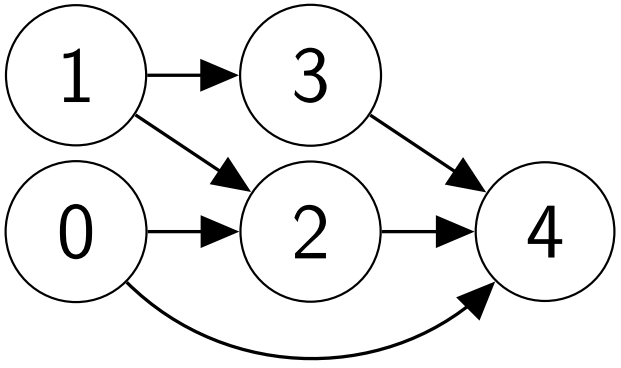
Uppgift 4

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

Uppgift 4

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.

Uppgift 4

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).

Lösning 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 position i innehåller en länkad lista med nod is direkta efterföljare.

Lösning 4

```
List<Integer> topological-sort(Graph g) {
  List<Integer> stack = new Stack<Integer>;
  Boolean visited[] = new Boolean[g.number-of-nodes];

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

  void dfs(Integer i) { ... }

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

  return stack;
}
```

Lösning 4

```
void dfs(Integer i) {
  visited[i] = true;

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

  stack.push(i);
}
```

Lösning 4

```
List<Integer> stack = new Stack<Integer>;
Boolean visited[] = new Boolean[g.number-of-nodes];

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

$$O(|V|)$$

Lösning 4

```
void dfs(Integer i) {
  visited[i] = true;

  ...

  stack.push(i);
}
```

Körs en gång per nod:

$$O(|V|)$$

Lösning 4

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

Körs en gång per kant:

$O(|E|)$ (exklusive rekursiva anrop)

Lösning 4

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

$O(|V|)$ (exklusive rekursiva anrop)

Lösning 4

Totalt:

$$O(|V| + |E|)$$

Uppgift 5

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

- `empty`: Skapar en tom lista.
- `insert`(x): Lägger till x sist i listan.
- `delete-last`: Tar bort och ger tillbaka listans sista element. Krav: Listan får inte vara tom.

Uppgift 5

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

- `empty`: Allokera 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 $k \in \mathbb{N}$.

Uppgift 5

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

Lösning 5

Potentialfunktion:

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

- c: arrayens längd.
- n: listans längd.
- k: positiv konstant.

Idé: Potentialen är minst när arrayen är halvfull, och ökar ju närmare vi kommer en fördubbling/halvering av storleken.

Lösning 5

- Specialfall: Potentialen 0 innan vi skapat listan.
- Potentialfunktionen OK?
  - Ja, ursprungspotentialen $\leq$ alla andra.

Lösning 5

`empty`:
Tar konstant tid, och ökar potentialen från 0 till k.
Den amorterade tidskomplexiteten är alltså
$O(1) + k$, vilket är $O(1)$ eftersom k är en konstant.

Lösning 5

`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 $O(1) + 2k = O(1)$.

Lösning 5

Om vi fördubblar arrayens kapacitet från c till $2c$ så är tidskomplexiteten för `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.

Lösning 5

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

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

Lösning 5

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

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

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

Lösning 5

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

Uppgift 6

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

(I en komplett graf finns det en kant mellan noderna u och v om och endast om $u \neq v$.)

Uppgift 6

Lös följande variant av handelsresandeproblemet *effektivt*:

- Kanternas vikter är ickenegativa. 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 hamiltonsk cykel, det räcker att hitta en hamiltonsk cykel som är max dubbelt så lång som de kortaste.

Uppgift 6

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

Tips: Använd ett minsta uppspännande träd.

Lösning 6

Algoritm:

- 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 den eftersökta cykeln.

Lösning 6

Korrekthet:
Notera först att p representerar en hamiltonsk cykel eftersom grafen är komplett och innehåller minst två noder. Det återstår att visa att cykelns längd är max $2w$, där w står för de kortaste hamiltonska cyklernas längd.

Lösning 6

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

Lösning 6

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 $v_1, v_2, ..., v_i$ $(i \geq 3)$ mot "genvägar" v_1, v_i (eftersom noderna $v_2, ..., v_{i-1}$ redan hade besökts). Triangelolikheten medför att de här genvägarna aldrig är längre än ursprungsvägsnutten:

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

Alltså är p max lika lång som p'.

Lösning 6

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

Beräkning av minsta uppspännande 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.