Laborationer i Java
TDDC91, TDDE22 & 725G97 - Datastrukturer och algoritmer

20 september 2018

Förord

Detta kompendium innehåller laborationer för kurserna TDDC91, TDDE22 samt 725G97: Datastrukturer och algoritmer (DALG) som ges vid Institutionen för datavetenskap (IDA), Linköpings universitet. Laborationerna är fyra till antalet och baseras på Java SE 7. Laborationerna bör genomföras i par, men samtliga i labbgruppen skall bidra till lösningen och också kunna motivera och förklara programkoden. För att bli godkänd krävs inte bara ett program som förefaller fungera på några testexempel utan koden skall vara rimligt effektiv och läsbar. Alla lösningar skall presenteras muntligen för laborationsassistenten i datorsalen och därefter skickas in med sendlab för rättning (kod samt i vissa fall teoretiska moment). Se även de allmänna regler som gäller för examinering av datorlaborationer vid IDA på nästföljande sida. Kodskeletten till de fyra laborationerna kan laddas ner från kurshemsidan eller kopieras från biblioteket

```
/home/TDDC91/code/lab{1..4}/
```

Observera: Det är inte meningen att man ska skriva om kodskeletten eller skriva helt egen kod; utan man ska utgå från den existerande koden och komplettera den med de delar som saknas, och som beskrivs i labbkompendiet. Assistenterna har inte möjlighet att lägga ned den tid som krävs för att sätta sig in i och rätta helt egna lösningar på uppgifterna.

De tre första laborationerna är baserade på laborationer i tidigare datastrukturkurser vid IDA, medan den fjärde laborationen härstammar från en dylik kurs vid KTH. Personer som på olika sätt bidragit till de nuvarande laborationerna är: Sven Moen, Lars Viklund, Tommy Hoffner, Johan Thapper, Patrik Hägglund, Daniel Karlsson, Daniel Persson, Dennis Andersson, Daniel Åström, Ulf Nilsson, Artur Wilk, Tommy Färnqvist, Mahdi Eslamimehr, Viggo Kann samt Hannes Uppman.

Lycka till!
Magnus Nielsen

Regler för examinering av datorlaborationer vid IDA

Datorlaborationer görs i grupp eller individuellt, enligt de instruktioner som ges för en kurs. Examinationen är dock alltid individuell.
Det är inte tillåtet att lämna in lösningar som har kopierats från andra studenter, eller från annat håll, även om modifieringar har gjorts. Om otillåten kopiering eller annan form av fusk misstänks, är läraren skyldig att göra en anmälan till universitetets disciplinnämnd.
Du ska kunna redogöra för detaljer i koden för ett program. Det kan också tänkas att du får förklara varför du har valt en viss lösning. Detta gäller alla i en grupp.
Om du förutser att du inte hinner redovisa i tid, ska du kontakta din lärare. Då kan du få stöd och hjälp och eventuellt kan tidpunkten för redovisningen senareläggas. Det är alltid bättre att diskutera problem än att, t.ex., fuska.
Om du inte följer universitetets och en kurs examinationsregler, utan försöker fuska, t.ex. plagiera eller använda otillåtna hjälpmedel, kan detta resultera i en anmälan till universitetets disciplinnämnd. Konsekvenserna av ett beslut om fusk kan bli varning eller avstängning från studierna.

Policy för redovisning av datorlaborationer vid IDA

För alla IDA-kurser som har datorlaborationer gäller generellt att det finns en bestämd sista tidpunkt, deadline, för inlämning av laborationer. Denna deadline kan vara under kursens gång eller vid dess slut. Om redovisning inte sker i tid måste, den eventuellt nya, laborationsserien göras om nästa gång kursen ges.
Om en kurs avviker från denna policy, ska information om detta ges på kursens webbsidor.

Redovisning

I normala fall går en redovisning till så att du demonstrerar programmet för assistenten, för att sedan skicka in kod och eventuella svar på teorifrågor. I år kräver vi att onlinedomaren **Kattis** har godkänt din labb innan du får redovisa den för din assistent. Kod samt svar ska skickas in via sendlab. Se instruktioner på `http://www.ida.liu.se/~TDDC91/current/info/labs.sv.shtml`. Se till att ni satt upp sendlab innan ni påbörjar laborationerna. Kattis bor på följande hemsida: `https://liu.kattis.com`.

Den tänkta arbetsgången för en labb är alltså:

1. Skriv kod.
2. Skicka in koden till Kattis och bli godkänd. I lydelsen för respektive laboration finns angivet vilka filer du ska skicka in till Kattis.
3. Redovisa för assistenten i labbsal och bli godkänd.
4. Skicka in kod och eventuella svar på teorifrågor via sendlab.
5. Efter granskning beslutar assistenten om labben slutgiltigt är godkänd eller om kompletteringar måste göras. Bokföring av detta sköts i Webreg.

Tips

- Komma igång med Kattis:
  `https://liu.kattis.com/help/`
- Om du behöver skicka in samma labb många gånger till Kattis kan det vara skönt att använda sig av submit-skriptet istället för webbgränssnittet:
  `https://liu.kattis.com/help/submit`
- Observera att Kattis inte hanterar Javafiler med package-deklarationer. Om du har din labb i ett paket måste du alltså låta bli att ta med paketdeklarationen när du skickar filerna till Kattis.

Lab 1: Hashtabeller

Mål: *Efter den här laborationen ska du kunna göra en icketrivial implementation av den abstrakta datatypen ordlista (eng. dictionary eller map) genom att använda en sluten hashtabell med öppen adressering. Du ska också känna till något om varför det är viktigt med bra kollisionshantering i hashtabeller.*

Förberedelser: *Läs om hashtabeller med öppen adressering i OpenDSA.*

I kompilatorer utnyttjas en så kallad *symboltabell* för att lagra information om de identifierare (variabler, konstanter, metodnamn etc) som förekommer i källkoden för det program som kompileras. De attribut som sparas för en identifierare kan t.ex. vara information om dess typ, adress och räckvidd (eng. scope).

Syftet med den här laborationen är att implementera en enkel symboltabell. Varje identifierare har ett unikt namn och en typ. För att snabbt kunna söka i tabellen använder vi en sluten hashtabell med öppen adressering. Informationen lagras i `keys` och `vals`, två globala, parallella arrayer. En cell i `keys` är en referens till ett objekt av typen `String`, identifierarens namn, och cellerna i `vals` är referenser till `Character`-objekt, identifierarnas typ. Konstanten `null` används för att markera tomma platser i hashtabellen. Vi har följande deklarationer i pseudokod:

```
public class SymbolTable {
   ...
   /* The keys */
   private String[] keys;
   /* The values */
   private Character[] vals;
   ...
```

Vår symboltabell ska stödja följande operationer:

1. `Character get(String key)`
   Slå upp identifieraren `key` i symboltabellen och returnera dess typ. Om identifieraren inte finns i tabellen, returnera `null`. Resultatet är odefinierat då hashtabellen är full.

2. `void put(String key, Character val)`
   Sätt in identifieraren med namn `key` och typ `val` i symboltabellen. Om identifieraren redan finns i tabellen, ändra till det nya `val`-värdet. Ett anrop till `put` där `val` är `null` ska ge samma resultat som anropet `delete(key)`. Resultatet är odefinierat då hashtabellen är/blir full eller då `key` är `null`.

3. `void delete(String key)`
   Ta bort identifieraren `key` ur symboltabellen. Eventuella efterföljande element i sonderingssekvensen måste tas bort och sedan sättas in på nytt, så att dessa element kan hittas även i fortsättningen.

Uppgift: *Implementera metoderna `get`, `put` och `delete`. Kodskeletten återfinnes på kurshemsidan under menyvalet "Laborationer". För att kunna återskapa körexemplet nedan behöver hashfunktionen vara den som summerar ASCII-värdena för alla tecken i identifieraren modulo tabellängden.*

Detta är naturligtvis inte en hashfunktion som skulle användas i en verklig symboltabell, men för våra syften duger den. Kollisionshanteringen som kallas linjär sondering (eng. linear probing) i kursboken är tillräcklig för den här laborationen (fördelen med att använda en mer avancerad sonderingsteknik är att man kan fylla hashtabellen mer utan att få försämrade söktider). Flytta ut även andra eventuella hjälpfunktioner till egna metoder.

Redigera filen `SymbolTable.java`. För att kompilera programmet, skriv sedan kommandot `javac SymbolTableTest.java`. Genom att ge kommandot `java SymbolTableTest` körs filen.

Använder du Eclipse kan du kompilera och köra programmet genom att högerklicka på klassen `SymbolTableTest.java` i paketutforskaren och sedan välja "Run As" och därefter "Java Application". Möjligen måste du också i menyn "Window" välja "Show View" och "Console".

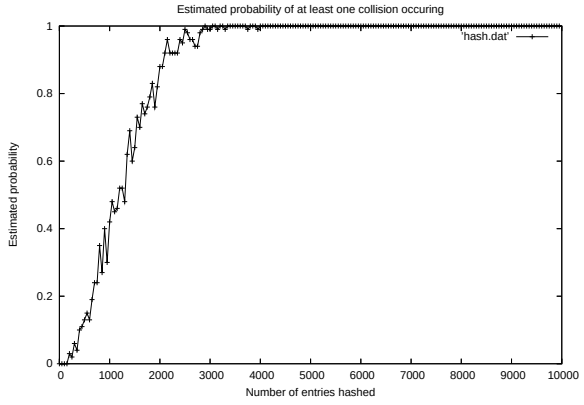
I testprogrammet är storleken på hashtabellen satt till 7 (så att körexemplet som återfinns nedan inte tar så stor plats). Där kan innehållet i hashtabellen dumpas via kommandot D. För varje cell i hashtabellen skrivs då följande ut: index, `val`-värdet och sist `key`-värdet tillsammans med dess hash-värde om `val`-värdet är skiljt från `null`.

Kattis: *När du har skrivit all kod och testat den utförligt skickar du in filen `SymbolTable.java` till Kattis för att få ditt program bedömt.*

Kollisioner

Hur ofta uppkommer det en kollision i en hashtabell? Om en hashtabell har storlek n visar det sig att sannolikheten för att minst en kollision uppstår är större än 0.5 även om mycket färre element än $n/2$ sätts in i tabellen. Detta faktum beror på den så kallade födelsedagsparadoxen (eng. birthday paradox). Du ska nu göra en liten undersökning av den här effekten.

Filen `Collisions.java` simulerar insättningar i en hashtabell av storlek 1000003 för att uppskatta risken för att en kollision uppstår vid insättning av mellan 50 och 10000 element i tabellen. Antagandet är att hashfunktionen lyckas sprida ut de insatta elementen med likformig sannolikhet över hela tabellen. I figuren nedan återfinns en graf över uppmätta sannolikheter från en typisk körning av programmet `Collisions`[1].



Uppgift: *Granska källkoden i `Collisions.java` och försök tolka det resulterande diagrammet i figuren ovan. Vad är den uppskattade sannolikheten för att minst en kollision uppstår när 2500 element sätts in i tabellen? Hur många element behöver sättas in i tabellen för att den uppskattade sannolikheten för en kollision ska överstiga 0.5 och går det att ge någon enkel förklaring till att det är så mycket färre insättningar än 500000 som krävs?*

[1] Programmet `hscript` som finns på samma ställe som kodskeletten utför mätningarna enligt ovan samt ritar en graf över de uppmätta sannolikheterna om någon skulle vilja verifiera experimentet. Grafen hamnar i `collisions.pdf`.

Redovisning: *Demonstrera programmet för assistenten samt lämna in* `SymbolTable.java` *via sendlab. Lämna också in en kort text (på egen fil) om de uppmätta kollisionssannolikheterna från* `Collisions` *samt dina tolkningar av dessa.*

Körexempel: (Användarens indata skrivs i kursiv stil.)

```
>  java SymbolTableTest
+--- Hash tables ---
r : Reset all
H : Hash
l : Lookup
i : Insert
d : Delete
D : Dump hashtable
q : Quit program
h : show this text

lab >  H
Hash string:  het
Hash(het) => 6

lab >  i
Insert string:  het
With type:  c

lab >  i
Insert string:  the
With type:  d

lab >  D
0 d the (6)
1 null -
2 null -
3 null -
4 null -
5 null -
6 c het (6)

lab >  l
Lookup string:  the
Lookup(the) => d

lab >  i
Insert string:  the
With type:  i

lab >  D
0 i the (6)
1 null -
2 null -
3 null -
4 null -
5 null -
6 c het (6)
```

```
lab >  H
Hash string:  info
hash(info) => 1

lab >  i
Insert string:  info
With type:  d

lab >  H
Hash string:  fusk
hash(fusk) => 0

lab >  i
Insert string:  fusk
With type:  c

lab >  D
0 i the (6)
1 d info (1)
2 c fusk (0)
3 null -
4 null -
5 null -
6 c het (6)

lab >  d
Delete string:  het

lab >  D
0 c fusk(0)
1 d info (1)
2 null -
3 null -
4 null -
5 null -
6 i the (6)

lab >  l
Lookup string:  het
lookup(het) => null

lab >
```

Lab 2: Binära sökträd

Mål: *Efter den här laborationen ska du i ett binärt sökträd kunna implementera borttagning av element samt traversering i preorder och levelorder av ett träd med hjälp av s.k. iteratorer i Java.*

Förberedelser: *Studera avsnitten om binära sökträd i OpenDSA. Repetera vid behov avsnitten om stackar och köer.*

Ett binärt sökträd som implementerar ADTn "map" är ett binärt träd där noderna innehåller par av nycklar och värden arrangerade så att för varje nod med nyckel `key` gäller att alla nycklar i dess vänstra delträd är mindre än (eller lika med) `key` och att alla nycklar i dess högra delträd är större än (eller lika med) `key`. För enkelhets skull antar vi att nycklarna är av typen `int` och att varje nyckel har ett associerat värde `val` som är av typen `String`. En trädnod har då följande schematiska utseende:

```
public class Node {
  public int key;
  public String val;
  public Node left, right;
  ...
}
```

Vi gör vidare antagandet att en nyckel aldrig förekommer flera gånger i trädet (dvs. trädet är en avbildning eller "map" på engelska). Våra binära träd har följande instansvariabler:

```
public class BST {
  private Node root;
  private int size;
  ...
}
```

I motsats till kursboken lagrar vi värden också i trädets löv eftersom vi på detta sätt kan halvera minnesåtgången i vissa sitationer.

Vårt binära sökträd implementerar bl.a. följande operationer:

- `public String find(int key)`
  Sök och returnera det värde som associeras med nyckeln `key`. Returnera värdet `null` om nyckeln inte finns i trädet.
- `public void insert(int key, String val)`
  Sätt in en nod med nyckel `key` och värdet `val` på rätt plats i sökträdet. Om det redan finns en nod med nyckeln `key` så skriver vi över dess värde med det nya värdet `val`.
- `public void remove(int key)`
  Sök och ta bort den nod vars nyckel är `key`. Om noden som ska tas bort har två barn *måste* ersättning med föregångaren i inorder användas. Gör ingenting om nyckeln inte finns i trädet.

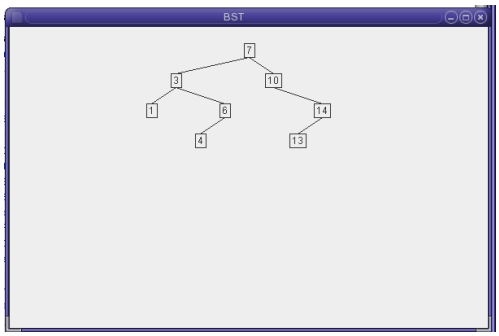
Dessutom ska vårt binära sökträd implementera följande traverseringsoperationer (med hjälp av iteratorer):

- `public PreorderIterator preorder()`
  Returnerar en iterator som itererar över trädets noder i ”preorder”.

- `public LevelorderIterator levelorder()`
  Returnerar en iterator som itererar över trädets noder i ”levelorder” (också kallat ”breddenförst”).

Uppgift: *Operationerna* `find` *och* `insert` *är redan implementerade. Det återstår att implementera operationen* `remove` *och dess eventuella hjälpfunktioner, samt de två iteratorerna ovan. Observera att när du implementerar* `remove` *är det* inte *meningen att du ska lägga till några egna datamedlemmar i klasserna. För iteratorn* `preorder` *ska du använda dig av ADTn stack och för iteratorn* `levelorder` *ADTn kö som båda återfinns i kodskeletten.*

Komplettera skeletten med dina metoder och kompilera programmet med hjälp av kommandot `javac BSTTest.java` eller, i Eclipse, aktivera filen `BSTTest.java` och välj därefter ”Run As” och ”Java Application”. Testprogrammet körs genom att skriva kommandot `java BSTTest` (i Eclipse startar programmet automatiskt om kompileringen lyckades). När programmet startar öppnas ett fönster med en representation av de nycklar som för tillfället är insatta i trädet. I figuren nedan visas ett exempel på hur det kan se ut efter att insättning av nycklarna 8, 3, 1, 6, 4, 7, 10, 14, 13 följt av borttagning av nyckel 8 gjorts:



Kattis: *Skicka in filerna* `BST.java`*,* `PreorderIterator.java` *och* `LevelorderIterator.java` *till Kattis för att få din kod bedömd.*

Redovisning: *Demonstrera programmet för assistenten samt lämna in källkoden via sendlab.*

Lab 3: Quicksort

Mål: *När du gjort den här laborationen ska du veta hur sorteringsalgoritmen Quicksort fungerar. Du ska också känna till hur man kan kombinera Quicksort och Insertionsort.*

Förberedelser: *Läs om Quicksort och Insertionsort i OpenDSA.*

Sorteringsalgoritmen Quicksort har en extremt kort inre loop vilket gör den mycket snabb i praktiken. En noggrannt trimmad version av Quicksort löper ofta snabbare i medelfallet än någon annan generell sorteringsalgoritm. Nackdelarna med algoritmen är att den kräver $O(n^2)$ operationer i värsta fallet och att den är ömtålig. Ett litet svårupptäckt misstag i implementationen kan ge dålig prestanda för vissa indata.

En vanligt förekommande modifiering av grundalgoritmen för Quicksort är att se till att sorteringen endast använder lite extra minne (förutom indatasekvensen), så kallad "in-place" sortering. Nedan följer ett exempel på hur pseudokod för partitioneringssteget i en sådan variant av Quicksort skulle kunna se ut (hämtad från Wikipedia):

```
function partition(array, left, right, pivotIndex)
    pivotValue := array[pivotIndex]
    swap array[pivotIndex] and array[right] //  Move pivot to end
    storeIndex := left
    for i from left to right - 1
        if array[i] ≤ pivotValue
            swap array[i] and array[storeIndex]
            storeIndex := storeIndex + 1
    swap array[storeIndex] and array[right] //  Pivot to final place
    return storeIndex
```

Fundera över om detta är en bra partitioneringsstrategi. Undersök hur koden gör för några små exempel. Gör koden några onödiga operationer? Finns det vissa typer av indata som gör att strategin fungerar sämre?

Mycket arbete har lagts ner på att försöka förbättra Quicksort. Trots att grundalgoritmen nu är 50 år gammal publiceras fortfarande nya varianter och förbättringar för speciella indata. En populär förbättring är till exempel att använda Quicksort och någon enkel sorteringsalgoritm i kombination. Ett bra sätt att implementera denna idé är att låta Quicksort ignorera alla delarrayer under en viss konstant storlek och lämna över arbetet till Insertionsort. Schematiskt skulle det kunna se ut på följande sätt:

```
static void sort(Comparable[] a, int lo, int hi, int m) {
    if (hi <= lo + m) { insertionsort(a, hi, lo); return; }
    int j = partition(a, lo, hi);
    sort(a, lo, j-1, m);
    sort(a, j+1, hi, m);
}
```

En annan vanlig förbättring är att välja pivotelementet som medianen av tre element.

Uppgift: *Skriv ett Quicksortprogram som sorterar en heltalsarray i stigande ordning och som pivot-element får ni göra ett eget val. "Medianen av tre"* ***bör*** *vara det bästa alternativet, men så är inte alltid fallet i Java beroende på vilken JVM som används och därmed släpper vi detta tidigare krav. Låt programmet lämna alla tillräckligt små delarrayer till Insertionsort. För att Quicksortprogrammet ska bli fullständigt behövs också en implementation av Insertionsort. Bestäm empiriskt vilket värde på brytpunkten* `m` *som ger det snabbaste programmet.*

Programskelettet `QSort.java` återfinns i vanlig ordning på kurshemsidan och i kursens filbibliotek. I filen `QSortTest.java` finns konstanterna `firstM` och `lastM`. Det är mellan dessa värden `m` kommer att varieras. För att mäta hur lång tid sorteringen tar mäts en tid med noggrannheten 1 ns. Precisionen i dessa tidsanrop beror på begränsningar i hårdvara, operativsystem och implementationen av den virtuella Java-maskinen, vilket gör att den uppmätta tiden varierar en aning från körning till körning. Ett anrop till `javac QSortTest.java` kompilerar programmet.

Till skillnad från laboration 1 och 2 används inte en speciell testmeny. Däremot finns det möjlighet att köra programmet i debug-mode. Då genereras en sekvens av 300 heltal. Vilken sekvens det blir styrs av fröet `seed`. Programmet kör sedan Quicksort tre gånger; en gång för `m = 1`, en gång för `m = 10` och en gång för `m = 300`. Resultatet av sorteringen skrivs sedan ut. Detta fås om man kör `java QSortTest` med flaggan `-d`.

```
> java QSortTest -d | more
```

Att kontrollera att sekvensen verkligen är sorterad kan vara en ganska enformig uppgift. Därför skriver programmet ut tal som ligger i fel ordning tillsammans med deras index.

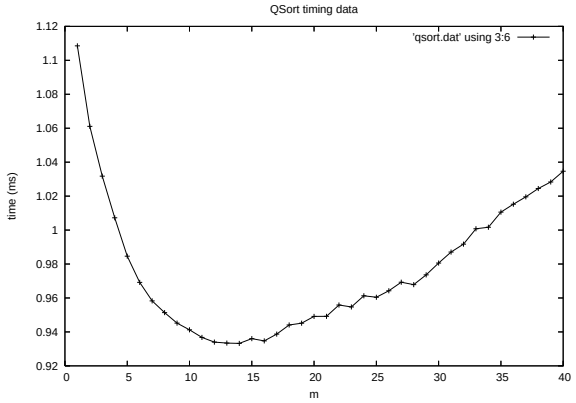
Efter att slutligen ha försäkrat dig om att ditt program sorterar rätt kör du det igen, med flaggan `-s`, för att leta efter den bästa brytpunkten för heltalsarrayer av längd 10000.

```
> java QSortTest -s
```

För att få bättre noggrannhet på de uppmätta tiderna sorteras nu talen 2000 gånger för varje värde på `m`. Endast de uppmätta tiderna skrivs ut. Det går att få fram en graf över de uppmätta tiderna i `timing.pdf` genom att ge följande kommandon:

```
> chmod a+x gscript
> ./gscript
```

Hur resultatet kan se ut visas nedan:



Kattis: *Skicka in filen* `QSort.java` *till Kattis.*

Redovisning: *Lämna in diagram*[2] *och empiriska resultat för Quicksort tillsammans med källkoden via sendlab. Glöm inte att demonstrera programmet för assistenten.*

[2]Det är naturligtvis fritt att generera diagrammet med andra hjälpmedel än `gscript`, så länge mätdata kommer från `QSortTest`.

Lab 4: Ordkedjor

Mål: *Efter den här laborationen skall du ha lärt dig mer om hur viktigt valet av datastrukturer och algoritmer kan vara för effektiviteten hos ett program. Den här uppgiften härstammar från Viggo Kann, KTH.*

Förberedelser: *Läs om sökning i grafer i OpenDSA.*

I katalogen `/home/TDDC91/code/lab4` finns ett Javaprogram som löser nedanstående problem. Din uppgift är att snabba upp programmet så att det går ungefär 10000 gånger snabbare.

Det ligger nära till hands att fråga sig hur man finner kortaste vägen från aula till labb genom att byta ut en bokstav i taget, till exempel så här:
aula → gula → gala → gama → jama→ jamb → jabb → labb
där alla mellanliggande ord måste finnas i ordlistan `word4` i labbens filkatalog.

För många ord i ordlistan existerar det en (kortaste) ordkedja från ordet själv till *labb*. Nu kan man fråga sig: vilket ord är det som har längst kortast ordkedja till *labb* och hur ser den kedjan ut? Det räcker att hitta och skriva ut en enda ordkedja av den maximala längden.

Specifikation

Indata består av två delar. Den första delen är ordlistan, som består av ett antal fyrbokstavsord, ett per rad. Denna del avslutas av en rad som bara innehåller ett `'#'`-tecken. Den andra delen är ett antal frågor, en per rad. En fråga är antingen på formen `'slutord'` eller på formen `'startord slutord'`, där bägge orden förekommer i ordlistan.

Programmet ska, för varje fråga på formen `'slutord'`, räkna ut hur lång den längsta kortaste kedjan är från något ord i ordlistan till slutordet och även skriva ut en sådan kedja. För frågor på formen `'startord slutord'` ska programmet räkna ut hur lång den kortaste kedjan är från startordet till slutordet, och även skriva ut en sådan kedja. Om det inte finns någon kedja från start- till slutord ska detta anges.

Exempel på körning

En ordlistefil finns i `/home/TDDC91/code/lab4/word4`. Du kan provköra ditt program genom att skriva in några testfrågor (t.ex. frågorna `'aula labb'` och `'sylt gelé'` och `'idén'`) på varsin rad i en fil (t.ex. `testord.txt`) och sedan köra

```
>cat /info/adk11/labb2/word4 testord.txt | java Main
aula labb: 8 ord
aula -> gula -> gala -> gama -> jama -> jamb -> jabb -> labb
sylt gelé: ingen lösning
idén: 15 ord
romb -> bomb -> bobb -> jobb -> jabb -> jamb -> jams -> kams
-> kaos -> klos -> klon -> klen -> ilen -> iden -> idén
```

Uppgift: *Det givna Javaprogrammet löser visserligen ovanstående problem, men det tar timmar att få fram svaret. Du ska effektivisera programmet så att det hittar svaret inom den tidsgräns som*

Kattis ger. Bra testfall att testa ditt program med finns på `/home/TDDC91/code/lab4/testfall/`. *De fyra teoriuppgifterna nedan ger uppslag om olika sätt att effektivisera programmet. Ditt optimerade program ska ha samma in- och utmatning som det givna programmet och det måste fortfarande vara Java.*

Kattis: *Skicka in alla* `.java`*-filer till Kattis. Glöm inte att ange vad klassen som innehåller* `main`*-metoden heter.*

Redovisning: *Svara på teorifrågorna, redovisa programmet för assistenten och lämna in koden samt skriftliga svar på teorifrågorna.*

Teorifrågor

1. Sätt dig in i hur det givna programmet fungerar. Svara speciellt på följande frågor: Vad används datastrukturen *used* till i programmet? Varför används just breddenförstsökning och inte till exempel djupetförstsökning? När lösningen hittats, hur håller programmet reda på vilka ord som ingår i ordkedjan i lösningen?

2. Både ordlistan och datastrukturen *used* representeras med klassen *Vector* i Java och sökning görs med metoden *contains*. Hur fungerar *contains*? Vad är tidskomplexiteten? I vilka lägen används datastrukturerna i programmet? Hur borde dessa två datastrukturer representeras så att sökningen går så snabbt som möjligt?

3. I programmet lagras varje ord som en String. Hur många Stringobjekt skapas i ett anrop av *MakeSons*? Att det är så många beror på att Stringobjekt inte kan modifieras. Hur borde ord representeras i programmet för att inga nya ordobjekt ska behöva skapas under breddenförstsökningen?

4. Det givna programmet gör en breddenförstsökning från varje ord i ordlistan och letar efter den längsta kedjan. Visa att det räcker med en enda breddenförstsökning för att lösa problemet.