TDDE22/725G97 - laborationer 2020
magnus.nielsen@liu.se

Labbserie för TDDE22 & 725G97 Datastrukturer och algoritmer

Innehåll

Labbserie för TDDE22 & 725G97
Datastrukturer och algoritmer

Förord

Detta kompendium innehåller laborationer för kurserna TDDE22 samt 725G97. Datastrukturer och algoritmer (DALG) som ges vid Institutionen för datavetenskap (IDA), Linköpings universitet. Laborationerna är fem 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. Det är **mycket viktigt** att läsa varje laboration noga. För att bli godkänd krävs inte bara ett program som förefaller fungera på några testexempel utan koden skall kunna lösa generella fall samt vara rimligt effektiv och läsbar. Alla lösningar skall presenteras muntligen för laborationsassistenten i datorsalen (eller möjligen vid kursledares kontor enligt överenskommelse om speciella omständigheter råder) och därefter skickas in för rättning via sendlab (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 fem laborationerna kan laddas ner från kurshemsidan eller kopieras från biblioteket

```
/courses/TDDE22/code/lab{1..4 + X}/
```

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.

Viktigt: Tanken är att **ni ska göra egna, ordentliga, testprogram**. De givna ”testprogram” som tillhör vissa av laborationerna är främst för visualisering så att man ordentligt förstår vad som händer, och är således **inte** tillräckliga för att säkerställa att ni har gjort rätt. Fundera **noga** på varje laboration, hitta fall som bör vara korrekta, och fall som bör vara fel, och testa dessa. Givetvis kan ni använda er av det givna testprogrammet i fallet med hashtabellen, men egna testprogram är sannolikt både effektivare och snabbare. Tänk på att testa ”fringe-cases” (ex. ta bort första och sista elementen), låt stora delar av tabellen fyllas upp av element med samma hashvärde och bekräfta att ordningen på kvarvarande element bibehålls efter borttagning av element i mitten, slutet, början, etc. Tanken i denna kurs är att ni ska få träna på att testa er själva till stor del, och det gäller inte bara hashtabellen utan hela labbserien. Med det sagt är det självklart ok att ställa frågor om ni kör fast, även om svaren kan vara mindre specifika än ni är vana vid från introduktionskurser i programmering.

De tre första laborationerna är, i någon utsträckning, 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: Filip Strömbäck, Oskar Holmström, 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

Labbserie för TDDE22 & 725G97
Datastrukturer och algoritmer

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. Kod samt svar ska skickas in via sendlab. Se instruktioner på `http://www.ida.liu.se/~TDDE22/info/labs.sv.shtml`.

Den tänkta arbetsgången för en labb är alltså:

1. Skriv kod.
2. Testa ordentligt, både korrekta och inkorrekta fall, och se till att din lösning håller tiden.
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.

Labbserie för TDDE22 & 725G97
Datastrukturer och algoritmer

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. Man bör fortfarande försöka få ut det data som eftersöks, det som händer om man inte hittar det vid traversering av tabellen är odefinierat.
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`. ***OBS!*** *Glöm inte att storleken (bör) ändras vid vissa operationer. 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.* ***Observera*** *att ni inte ska skriva om / göra en ny hashfunktion utan använd den givna.*

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 även ut eventuella hjälpfunktioner till egna metoder.

Givna filer

- Hämta filerna genom att ställa dig i TDDE22-mappen och köra:
  `cp -r /courses/TDDE22/code/lab1 .` (inklusive punkten).
- `SymbolTable.java` är den fil du ska arbeta med.
- För att kompilera programmet, kör 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”.

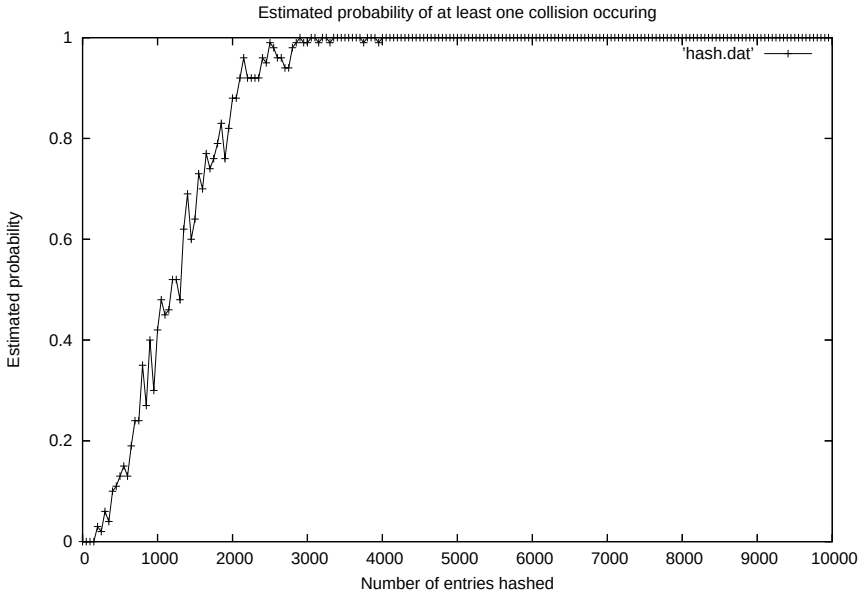
I visualiseringsprogrammet (SymbolTableTest) är storleken på hashtabellen satt till 7 (så att körexemplet som återfinns nedan inte tar så stor plats). Där kan innehållet i hashtabellen dumpas via kommandot D. För varje cell i hashtabellen skrivs då följande ut: index, `val`-värdet och sist `key`-värdet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

Redovisning: *Demonstrera programmet för assistenten samt lämna in* `SymbolTable.java` *via sendlab.*

Frivillig teoriuppgift - 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.

Uppgift: *Granska källkoden i* `Collisions.java` *och försök tolka det resulterande diagrammet i figuren nedan. 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?*



Testfall

(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 >
```

Labbserie för TDDE22 & 725G97
Datastrukturer och algoritmer

Lab 2: AVL-träd

Mål: *Efter denna laboration ska du ha god förståelse för träd i allmänhet, och det självbalanserande AVL-trädet i synnerhet med dess olika rotationer.*

Förberedelser: *Läs om AVL-träd i OpenDSA. Titta även gärna på visualiseringslänkarna som finns på föreläsningssidan.*

Uppgift: *I den givna koden finns ett BST (vanligt binärt sökträd) implementerat. Din uppgift är att komplettera det och börja göra det till ett självbalanserande AVL-träd.* `Insert` *och* `remove` *är deklarerade i klassen* `AVLTree`*, där de anropar motsvarande funktioner i* `AVLTreeNode`*. För närvarande gör den insättning och borttagning utan att balansera. Uppgiften är att se till att balansering görs i:*

- `insert`*. Denna del är obligatorisk.*
- `remove`*. Denna del är frivillig.*

Operationen `remove` ska ta bort det angivna elementet (om det finns). Utgå från den givna funktionen remove i nodklassen som finns på filen `AVLTreeNode.java` och komplettera den med balansjustering. `Insert` bör sätta in på samma sätt som nu, men även här måste du se till att balansering utförs. Tänk noga igenom var en obalans ligger i förhållande till noden som har obalans (tips: rita), och hur man kan avgöra om en enkelrotation eller en dubbelrotation ska göras! På den punkten skiljer sig borttagning påtagligt från vad som gäller vid insättning. Tänk också på att borttagningen resulterar i att vissa pekare kan bli `null` i `remove` där vi i `insert` vet att de alltid är giltiga. Till din hjälp har du en mängd stödfunktioner i `AVLTreeNode`, bland annat de olika rotationerna (längst ner i filen), funktioner för att beräkna höjden, etc. Håll tidskomplexiteten i åtanke medan du arbetar så att din lösning är rimlig.

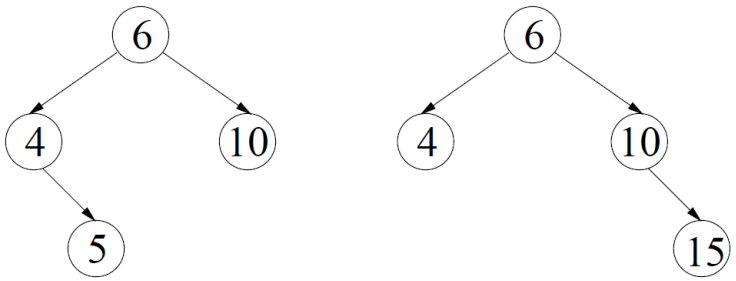
Givna filer

- Hämta filerna genom att ställa dig i din TDDE22-mapp och köra:
  `cp -r /courses/TDDE22/code/lab2 .` (inklusive punkt)
- Klassen `AVLTree.java` innehåller definitionen av trädet och de tillhörande metoderna i klassen `AVLTree`. Denna fil ska inte ändras.
- `AVLTreeNode.java` innehåller noden samt implementationen av de olika trädoperationerna baserat på hur nodtypen ser ut, inklusive rotationer. Alla förändringar du gör bör vara i denna fil.
- Filen `AVLTreeTest.java` innehåller ett visualiseringsprogram för AVL-trädet. Sätt in några värden i ett AVL-träd så kan en del av operationerna på AVL-trädet testas interaktivt.
- För att kompilera kör du `javac AVLTest.java` och kör programmet med `java AVLTest`, precis som vanligt, alternativt kör i Eclipse (eller den IDE du väljer att använda).

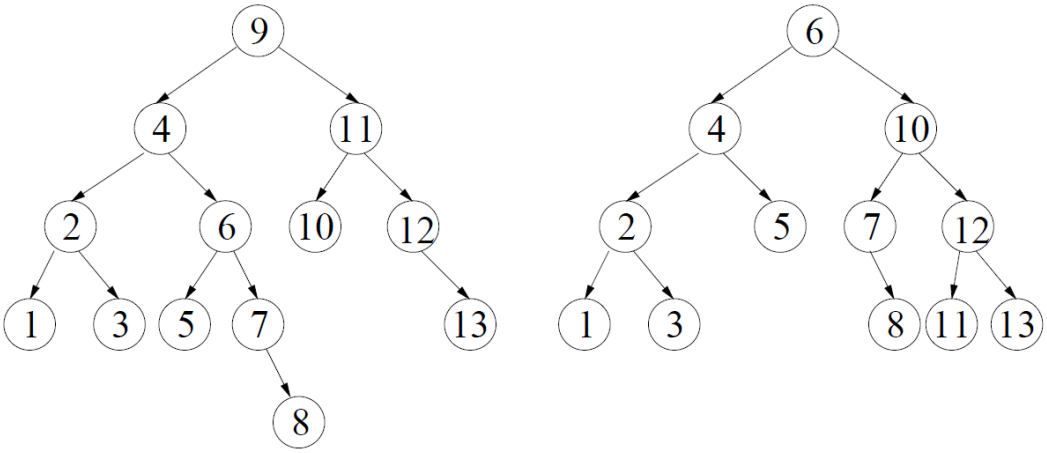
Redovisning: *Demonstrera programmet för assistenten samt lämna in* `AVLTreeNode.java` *via sendlab.*

Testfall

Nedan finns tre testfall din kod bör klara om du gör `båda` funktionerna. Observera att din kod naturligtvis måste fungera även i det generella fallet och inte bara på de tre testfallen, så tänk noga igenom vilka situationer som kan uppkomma och hur de ska hanteras när du gör dina testprogram! Gör du bara en av funktionerna måste du vara noga med att fundera ut bra testfall.



Testfall 1 är borttagning av nod 10 i trädet ovan till vänster och testfall 2 är borttagning av nod 4 i trädet ovan till höger. Rita och fundera på hur resultatet av dessa borttagningar bör se ut innan du kör testet för att kunna bekräfta att du har gjort rätt.



Testfall 3 är borttagning av nod 9 i trädet ovan till vänster. Trädet kan byggas genom att köra `AVLTreeTest` med följande indata: `1 9 1 4 1 11 1 2 1 6 1 10 1 12 1 1 1 3 1 5 1 7 1 13 1 8`. Resultatet av borttagningen ska bli som i trädet ovan till höger.

Lab 3: Mönsterigenkänning

Bakgrund

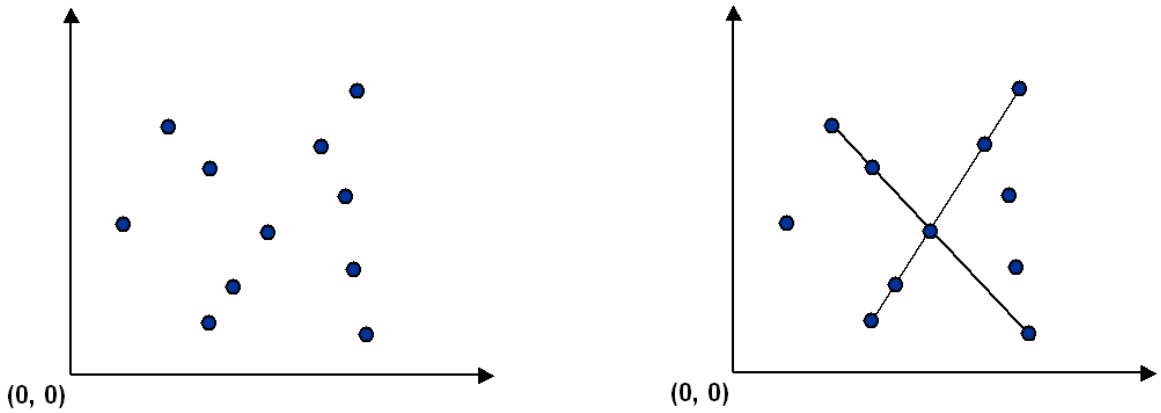
Du har kommit över ett antal filer som innehåller en mängd till synes slumpmässiga punkter. Du tror dock att vissa av dem innehåller gömda meddelanden, och du vill därför skriva ett program som kan hjälpa dig att hitta dessa. Du vet att de gömda meddelandena består av linjesegment, och att minst fyra punkter från varje linje finns med i punktmängden (de två ändpunkterna av linjesegmentet, och två till). För att rekonstruera linjerna vill du alltså hitta alla uppsättningar om fyra eller fler punkter som ligger på samma linje och rita ut dessa.

Tillämpningar

Viktiga komponenter i datorseende är att använda mönsteranalys av bilder för att rekonstruera de verkliga objekt som genererat bilderna. Denna process delas ofta upp i två faser: *feature detection* och *pattern recognition*. I *feature detection* väljs viktiga områden hos bilden ut; i *pattern recognition* försöker man känna igen mönster i områdena. Här får ni chansen att undersöka ett särskilt rent mönsterigenkänningsproblem rörande punkter och linjesegment. Den här typen av mönsterigenkänning dyker upp i många andra tillämpningar som t.ex. statistisk dataanalys.

Uppgift:

I de givna filerna finns programmet `brute`*, som löser ovanstående problem. Programmet läser in en mängd diskreta punkter i planet från fil, och hittar alla (maximala) linjesegment som innehåller en delmängd av fyra eller flera av punkterna, vilka ritas ut på skärmen.*



Programmet löser problemet genom att göra en totalsökning över alla kombinationer av 4 punkter och för varje kombination kontrollera ifall de ligger på en linje. Om så är fallet ritas ett linjesegment mellan ändpunkterna ut. Detta upprepas tills alla kombinationer har undersökts. En liten optimering

som är implementerad är att inte undersöka om 4 punkter är linjärt beroende ifall de 3 första punkterna som har valts ut inte är det.

1. *Undersök programmet* `brute` *och gör en analys av dess tiskomplexitet. Redogör dina resultat i filen* `readme.txt`*.*
2. *Programmet* `brute` *är på tok för långsamt för att kunna användas för större datamängder. Du ska därför implementera en bättre lösning,* `fast`*, som ska klara av att hitta alla linjesegment i en mängd med 12800 punkter på under 1 minut på IDA:s datorer (notera att ThinLinc-servern är något långsammare än datorerna i datorsalarna på den här typen av beräkningar). Se den sista rubriken för idéer. Analysera också din snabbare lösning på samma sätt som för* `brute` *och fyll i* `readme.txt`*.*

OBS! *Då årets upplaga av kursen går på distans kan vi inte ge er några vettiga mätvärden vad det gäller tid för körning. Det är därför extra viktigt att ni resonerar kring vad som är rimligt när ni gör denna laboration. ThinLinc-servern uppdateras ofta, och körtid varierar med belastning, så vi kan tyvärr inte ge er några rimliga körtider för den heller.*

Givna filer

Hämta filerna genom att ställa dig i din TDDE22-mapp och köra:
`cp -r /courses/TDDE22/code/lab3 .` (inklusive punkt)

Följande filer är givna för labben:

`Point.java` Dessa filer innehåller en implementation av en punktdatatyp som används för att lagra punkterna som undersöks. Punkten har i vanlig ordning medlemsvariabler för x- och y-koordinater. Punkter kan också jämföras med lessThan eller greaterThan, vilken ger en lexikografisk ordning av punkterna (undersök gärna implementationen). Utöver det finns medlemsfunktionen `slopeTo`, vilken beräknar lutningen från denna punkten till en annan punkt (som en lutningskoefficient, kan vara $\pm\infty$). Till sist finns också den fria funktionen `render_line` som ritar en linje mellan två punkter på skärmen.

`Brute.java` Implementation av en bruteforce-lösning till problemet.

`Fast.java` Kodskelett där du kan implementera en snabbare lösning till problemet. Koden sköter inläsning av data, utritning av punkter och tidtagning.

`readme.txt` Readme-fil som ska fyllas i med resultaten av din undersökning av `brute`, samt en analys av din snabbare lösning.

`data/` Innehåller givna testfall, se rubriken Testfall för detaljer.

Testfall

Mappen `data/` innehåller ett stort antal testfall i form av textfiler som du kan använda för att testa din lösning. Varje indatafil börjar med ett heltal som anger antalet punkter i filen, följt av x- och

y-koordinater för punkterna. I och med att programmen förväntar sig att läsa indata från fil körs testfallen lämpligt enligt följande:

```
java brute data/filnamn.txt
```

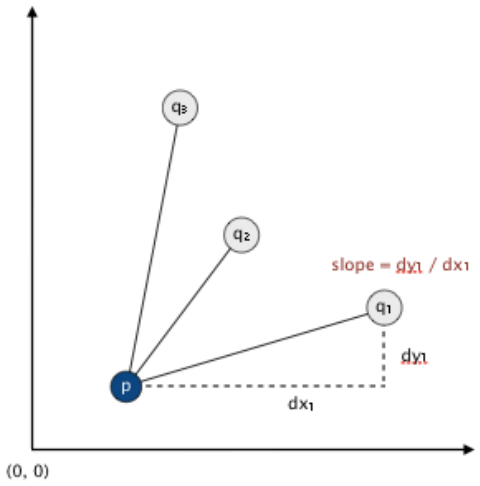
En del av testfallen har en `png`-fil som innehåller den förväntade utdatan från en korrekt lösning. Så länge `brute` inte har modifierats kan även `brute` användas för att generera en referensbild.

Följande fall kan vara intressanta att titta närmare på:

- `grid4x4`, `grid5x5`, `grid6x6` (jämför med utdatan från `brute`)
- `input12800.txt` (för att testa prestandan hos er lösning, innehåller 12800 punkter)
- `mystery10089.txt` (bra prestandatest med utfall som är enkelt att verifiera)

En snabbare, sorteringsbaserad, lösning

Vi kan lösa problemet mycket snabbare än vad `brute` gör genom att använda oss av sortering. Idén är som följer:



1. Välj en punkt p av alla punkterna i datamängden.
2. Tänk på p som origo.
3. Sortera alla andra punkter efter lutningen de har gentemot p.
4. Kontrollera om tre (eller fler) punkter ligger på samma linje som p.

Dessa steg gör att vi hittar alla linjesegment som går igenom p, så om vi upprepar dessa steg för *alla* möjliga p så hittar vi alla linjesegment. Du kan välja att antingen skriva en egen sorteringsfunktion eller en komparator för att använda med Collections.sort (se exempelvis Brute.java).

Labbserie för TDDE22 & 725G97
Datastrukturer och algoritmer

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 `/courses/TDDE22/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 `/courses/TDDE22/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 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 en mycket kort tid.* ***Obs!*** *Normalt har vi här en tidsgräns, men då kursen går på distans kan vi inte ge några rimliga gränser. ThinLinc-servern uppdateras regelbundet och körtid varierar baserat på belastning så det går inte att ge några rimliga tider här heller. Kortfattat bör det absolut röra sig om sekunder, inte minuter eller timmar, för att köra det stora exemplet. Bra testfall att testa ditt program med finns på* `/courses/TDDE22/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.*

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 *makeChildren*? 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.

Labbserie för TDDE22 & 725G97
Datastrukturer och algoritmer

LabX: Kollisioner och Einstein!

Förord - Viktigt!

Denna laboration skiljer sig något från de andra laborationerna. Det finns inga labbpass bokade för denna laboration utan tanken är att den ska genomföras på egen tid. Var extra noga med att inte diskutera lösningen med dina kamrater då det krävs extremt lite kodande för att lösa den, och för mycket hjälp kommer garanterat att förstöra hela syftet med laborationen. Tanken är att illustrera hur viktigt det är att vara noga när man väljer sina datastrukturer och algoritmer, samt hur lite som kan krävas för att göra väldigt stora förbättringar (eller försämringar) beroende på vilket syfte vi har med programmet. Större delen av tiden ni ägnar åt denna laboration kommer mycket sannolikt att bestå av att läsa given källkod och experimentera med simuleringen.

Den här laborationen är delvis baserad på en C++-laboration vi har använt i andra DALG-kurser, och den är i sin tur en anpassad version av en mycket mer omfattande labbserie i Java som ges vid Princeton University av Robert Sedgewick.

Mål: *När du gjort den här laborationen ska du känna till hur man kan simulera rörelserna hos N partiklar som kolliderar elastiskt med hjälp av händelsedriven simulering. Specifikt ska du behärska användandet av prioritetsköer vid sådan simulering. Den här typen av simulering är mycket vanligt förekommande när man försöker förstå och förutsäga egenskaper hos fysikaliska system på molekyl- och partikelnivå. Detta inkluderar rörelserna hos gasmolekyler, dynamiken i kemiska reaktioner, diffusion på atomnivå, stabiliteten hos ringarna runt Saturnus, och fasövergångarna i olika grundämnen. Samma tekniker används i andra tillämpningsområden, som datorgrafik, datorspel och robotik, för att simulera olika partikelsystem.*

Förberedelser: *Läs på om prioritetsköer i OpenDSA.*

Givna filer

Givna filer kommer ni åt genom att ställa er i er TDDE22-mapp och köra (copy paste till terminalen): `cp -r /courses/TDDE22/code/labx .` (inklusive punkt)

Därefter kan ni i Eclipse köra new -> java project och döp projektet till labx. Alla filerna bör därefter dyka upp Eclipse i vanlig ordning. Arbetar ni i Emacs eller liknande är det bara att köra på som vanligt.

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 lydelsen. Assistenterna har inte möjlighet att lägga ned den tid som krävs för att sätta sig in i och rätta helt egna lösningar på uppgifterna.

Simulering av partikelkollisioner i Java

Vår implementation innefattar följande klasser: `MinPQ`, `Particle`, `Event`, `StdDraw` och `CollisionSystem`. `Event`-klassen representerar en kollisionshändelse. Det finns fyra sorters händelser: kollision med vertikal vägg, kollision med horisontell vägg, kollision mellan två partiklar och en händelse som betyder att det är dags att rita om bilden av partikelsystemet. Vi har valt följande enkla (men inte särskilt eleganta) lösning: För att kunna avgöra om en händelse är giltig sparar `Event` antalet kollisioner relevanta partiklar varit inblandade i när händelsen skapades. Varje Event har två huvudpartiklar (de två som förutses kollidera). När händelsen bearbetas svarar den mot en fysisk kollision om antal kollisioner för vardera partikel är samma som när händelsen skapades, dvs de inblandade partiklarna har inte varit inblandade i några andra kollisioner från det att händelsen förutsågs tills det den bearbetas.

Katalogen `/courses/TDDE22/labs/labx` innehåller, förutom klasserna ovan, filen `Simulator.java` som i sin tur innehåller ett program som driver simuleringen. Utöver källkodsfilerna innehåller katalogen även några indatafiler: `billiards10.txt` vilket är en väldigt liten simulering, `diffusion.txt` som är lite större och mer intressant, och slutligen `brownian.txt` vilket är den stora simuleringen. Frånsett testprogrammet (TestMinPQ.java) är det enklast att manuellt kompilera och köra programmet i terminal. Om du navigerar till src mappen kan du kompilera genom att köra:

```
>javac Simulator.java
```

Du kan nu ge kommandot:

```
>java Simulator N
```

för att simulera N partiklar med slumpmässiga egenskaper i 10000 sekunder. Kommandot

```
>cat data/filnamn | java Simulator
```

läser in partikeldata från filen *`file`* och kör simuleringen. Det är nu dags att du experimenterar lite med programmet och sätter dig in i källkoden för att få en översiktlig bild av hur simuleringen går till. Kör definitivt minst en gång med `brownian.txt` som indatafil enligt ovan innan ni börjar hacka kod.

Uppgift: *I filen* `MinPQ.java` *finns den prioritetskö simuleringsprogrammet använder sig av. Din uppgift är att implementera* heap-insättning *i prioritetskön. I nuläget görs lat insättning, vilket innebär att vi bara stoppar in event i kön på första bästa lediga plats, vilket i sin tur innebär att vi måste testa och återställa heap-egenskapen för hela heapen varje gång vi tar ut min-elementet. För denna insättning används metoden toss i* `MinPQ`*.*

Funktionen `insert` *kastar bara ett undantag, men ska istället göra insättning enligt principen för heapar. Metoder för att återställa heap-egenskapen finns givna, det är dock* ***inte*** *ok att anropa fixHeap() i insert då detta är extremt ineffektivt (och inte speciellt utmanande). Om du anser dig behöva göra andra ändringar i* `MinPQ.java` *för att implementationen ska bli korrekt bör du se till att det går att ändra tillbaka och köra med pq.toss i CollisionSystem (förklaras nedan). I labbkatalogen finns också filen* `TestMinPQ.java` *som innehåller ett program som testar ifall prioritetskön fungerar som den ska. Anropet* `javac TestMinPQ.java` *kompilerar testprogrammet och du kan sedan köra det med* `java TestMinPQ` *alternativt köra programmet i Eclipse.*

När du fått prioritetskön att fungera med heap-insättning är det dags att ersätta alla anrop till `pq.toss` *med* `pq.insert` *i* `CollisionSystem.java` *och kontrollera att simuleringsprogrammet fortfarande fungerar som det ska. Observera att* `toss` *och övriga metoder i* `MinPQ` *fortfarande måste fungera korrekt trots att heap-insättning kan ha gjorts.*

Datafilerna för partikeldata har följande format: Första raden innehåller antalet partiklar N. Resterande N rader innehåller vardera 6 reella tal (position, hastighet, massa och radie) följt av tre heltal (rött, grönt och blått färgvärde). Alla positionskoordinater ska ligga mellan 0 och 1 och färgvärdena mellan 0 och 255. Ingen partikel får heller ha något överlapp med någon annan partikel eller väggarna.

```
N
rx ry vx vy mass radius r g b
rx ry vx vy mass radius r g b
rx ry vx vy mass radius r g b
rx ry vx vy mass radius r g b
```

Uppgift: *Skriv en kort reflektion över de förändringar du har gjort. Vilken version är bättre? Varför? Finns det fall då den "sämre" implementationen är bättre än den du valde ut?*
Frivilligt: Skapa minst en ny partikeldatafil som simulerar någon intressant situation. Om fantasin tryter ger vi några förslag nedan.

Förslag på situationer att simulera:

- En partikel i rörelse.
- Två partiklar i rörelse rakt emot varandra.
- Två partiklar, en i vila, som kolliderar i vinkel.
- N partiklar i ett rutnät med slumpmässiga initiala riktningar (men samma fart), så att den totala kinetiska energin motsvarar en fix temperatur T och totalt rörelsemängdsmoment $= 0$.
- Diffusion: N väldigt små partiklar av samma storlek nära mitten av behållaren med slumpmässiga hastigheter.

- Diffusion av partiklar från en fjärdedel av behållaren (jämför `diffusion.txt`).
- N stora partiklar, så att det inte finns så mycket rörelseutrymme.
- 10 partiklar i en rad som inte kolliderar med 9 partiklar i en kolumn.
- 9 partiklar i rad som kolliderar med 9 partiklar i en kolumn.
- Andra biljardsituationer än den i `billiards.txt`.
- En liten partikel trängs mellan två stora partiklar.
- Newtons pendel.

Redovisning: *Assistenterna kommer i vanlig ordning att ha frågor som ska besvaras vid muntlig redovisning. Du behöver bara redogöra för dina ändringar av* `MinPQ.java`*, inte de ändringar du gjort i* `CollisionSystem.java`*. Efter godkänd redovisning ska MinPQ.java samt ert reflektionsdokument manuellt skickas via sendlab. Det räcker med att skicka in MinPQ.java, samt ett reflektionsdokument på valfritt (rimligt) filformat (lämpliga exempel: .txt, .md, .odt).*

Teoretisk / vetenskaplig bakgrund

Frivillig läsning. Det är inget krav att man förstår den vetenskapliga bakgrunden eller formlerna som används för att göra beräkningarna. Detta avsnitt kan vara lämpligt att skumma igenom för kontext, eller studera för den nyfikne, men det är inget krav att man har läst det för att klara uppgiften.

Stela sfär-modellen

Stela sfär-modellen är en idealiserad modell av rörelsen hos atomer och molekyler i en behållare. Vi kommer att fokusera på den tvådimensionella versionen, kallad *stela skiv-modellen*. De väsentliga egenskaperna hos denna modell listas nedan.

- N partiklar i rörelse, inneslutna i en låda med sidor av enhetslängd.
- Partikel i har känd position (rx_i, ry_i), hastighet (vx_i, vy_i), massa m_i och radie σ_i.
- Partiklar interagerar genom elastiska kollisioner med varandra och den reflekterande gränsytan.
- Inga andra krafter förekommer. Detta betyder att partiklar rör sig i räta linjer med konstant fart mellan kollisioner.

Den här enkla modellen har en central roll i *statistisk mekanik*, ett vetenskapligt område där man försöker koppla samman makroskopiska fenomen (som temperatur, tryck eller diffusionskonstant) med rörelser och dynamik på atom- och molekylnivå. Maxwell och Boltzmann använde modellen för att härleda ett samband mellan temperatur och fördelningen av olika farter hos interagerande molekyler; Einsten använde den för att förklara den Brownska rörelsen hos pollenkorn i vatten.

Simulering

Det finns två naturliga sätt att försöka simulera partikelsystem.

- *Tidsdriven simulering.* Diskretisera tiden i kvanta av storlek dt. Uppdatera varje partikels position efter dt tidssteg och kolla efter överlapp. Om det finns två överlappande partiklar, backa klockan till tidpunkten för kollisionen och förtsätt simuleringen. Det här sättet att simulera innebär att vi måste göra N^2 överlappskontroller per tidskvantum och att, om dt är för stort, riskerar vi att missa kollisioner eftersom kolliderande partiklar inte överlappar när vi tittar efter. För att få tillräcklig noggrannhet måste vi alltså välja dt väldigt litet, vilket saktar ner simuleringen.
- *Händelsedriven simulering.* I den här formen av simulering koncentrerar vi oss bara på de tidpunkter vid vilka någon intressant händelse äger rum. I stela skiv-modellen rör sig alla partiklar i räta linjer, med konstant fart, mellan kollisioner. Vår utmaning blir alltså att bestämma en ordnad sekvens av partikelkollisioner. Detta gör vi genom att upprätthålla en *prioritetskö* med framtida händelser, ordnade efter tid. Vid varje given tidpunkt innehåller prioritetskön alla framtida kollisioner som skulle äga rum, givet att varje partikel fortsätter längs en rät linje för all framtid. Allteftersom partiklar kolliderar och byter riktning blir vissa framtida händelser i prioritetskön ”ogiltiga” och svarar inte längre mot fysiska kollisioner. Vi

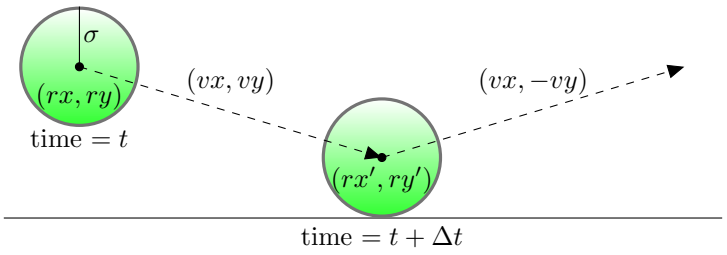
kommer att använda en lat strategi där vi lämnar sådana händelser kvar i prioritetskön för att identifiera dem och bortse från dem när de sedan plockas ut ur kön. Huvudloopen i simuleringen fungerar enligt följande:

- Ta bort den närmast förestående händelsen, dvs den med lägst prioritet t.
- Om händelsen motsvarar en ogiltig kollision, kasta bort den. Händelsen är ogiltig om en av partiklarna har deltagit i en kollision sedan händelsen sattes in i prioritetskön.
- Om händelsen svarar mot en fysisk kollision mellan partikel i och partikel j:
  * Flytta fram alla partiklar till tid t längs rätlinjiga rörelsebanor.
  * Uppdatera hastigheterna för de två kolliderande partiklarna i och j enligt lagarna för elastiska kollisioner.
  * Bestäm alla framtida händelser som skulle äga rum där antingen i eller j är inblandade, under antagandet att alla partiklar rör sig längs räta linjer från tid t och framåt. Sätt in dessa händelser i prioritetskön.
- Om händelsen svarar mot en fysisk kollision mellan partikel i och en vägg, gör motsvarande saker som ovan för partikel i.

Det här händelsedrivna sättet att simulera blir mer robust, exakt och effektivt än det tidsdrivna.

Att förutsäga kollisioner

Hur kan vi förutsäga framtida kollisioner? Partikelhastigheterna innehåller faktiskt all information vi behöver. Antag att en partikel har position (rx, ry), hastighet (vx, vy) och radie σ vid tid t och att vi vill avgöra om och när partikeln kolliderar med en vertikal eller horisontell vägg.



Eftersom alla koordinater är mellan 0 och 1 kommer en partikel i kontakt med en horisontell vägg vid tid $t + \Delta t$ om $ry + \Delta t \cdot vy$ är lika med antingen σ eller $(1 - \sigma)$. Om vi löser ut Δt får vi:

$$\Delta t = \begin{cases} (1 - \sigma - ry)/vy & vy > 0, \\ (\sigma - ry)/vy & vy < 0, \\ \infty & vy = 0. \end{cases}$$

Alltså kan vi sätta in en händelse i prioritetskön med prioritet $t + \Delta t$ (och information som beskriver kollisionen mellan partikel och vägg). En liknande ekvation förutsäger tidpunkten för kollision med en vertikal vägg. Beräkningarna för två partiklar som kolliderar är också snarlika, men mer komplicerade. Notera att allt som oftast leder beräkningen till att en kollision *inte* kommer att ske. I

så fall behöver vi inte sätt in något i prioritetskön. Det kan också hända att den förutsedda kollisionen ligger så långt fram i tiden att den inte är intressant. För att slippa lagra sådan information i kön håller vi reda på en parameter `limit` som ger en bortre gräns för vilka händelser vi är intresserade av.

Lösning av kollisioner

När en kollision väl äger rum behöver vi lösa upp den genom att applicera de formler från fysik som bestämmer beteendet hos en partikel efter en elastisk kollision med en reflekterande gränsyta eller med en annan partikel. I vårt exempel ovan, då partikeln träffar en horisontell vägg, ändras hastigheten från (vx, vy) till $(vx, -vy)$ vid tidpunkten för kollisionen. Sambanden för kollision mot vertikal vägg och kollision partikel mot partikel är snarlika.