Laborationer i C++
TDDI16
Datastrukturer och algoritmer

27 augusti 2017

Förord

Detta kompendium innehåller laborationer för kursen TDDI16 Datastrukturer och algoritmer som ges vid Institutionen för datavetenskap (IDA), Linköpings universitet. Kursens fyra laborationer baseras på C++.

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. Kodningsmässigt är laborationerna inte speciellt omfattande. I de flesta fall är det fråga om att använda och modifiera given kod. Detta är en i praktiken vanligt förekommande situation och därför en bra och realistisk övning då labbtiden i kursen dessutom är begränsad.

Programkod

Givna program finns tillgängliga i mappar för respektive uppgift i huvudmappen för laborationerna:

```
/home/TDDI16/kursbibliotek/lab/
```

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.

Tidsplanering

Någon direkt tidplan ges ej för utförandet av dessa uppgifter, utan du får själv planera hur laborationstiden bäst disponeras för de olika uppgifterna. Föreläsningarna styr när det tidigast kan vara lämpligt att göra respektive uppgift.

Redovisning

På kurshemsidan finns instruktioner för hur inlämning av labbar ska gå till. Se även respektive uppgift för hur den ska redovisas och följ dessa anvisningar! Se också de allmänna regler som gäller för examinering av datorlaborationer vid IDA på kurshemsidan.

Lycka till
Rita Kovordanyi

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.

Lab 1: AVL-träd

Mål: *Efter den här laborationen ska du kunna implementera borttagning av element i ett AVL-träd.*

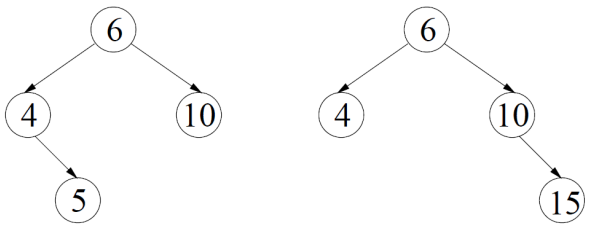
Förberedelser: *Läs på om AVL-träd.*

I mappen `/home/TDDI16/kursbibliotek/lab/lab1/` finns den givna koden för AVL-trädet:

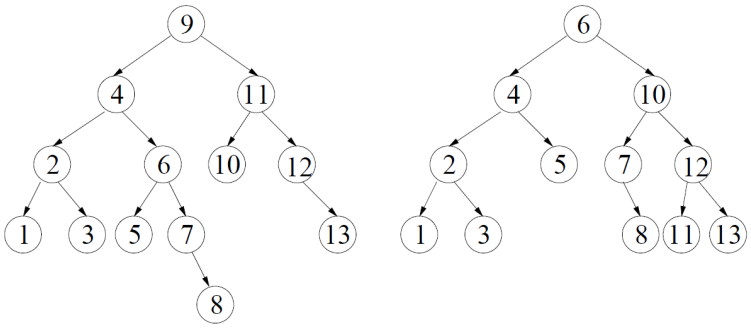
- Inkluderingsfilen `AVL_tree.h` innehåller definitionen för AVL-trädet och några tillhörande funktioner som ej är medlemmar (`swap` och `operator«`).
- Implementeringsfilen `AVL_tree.cc` innehåller definitionerna för medlemsfunktionerna i klassen `AVL_Tree` samt definitioner för den nodtyp som AVL-trädet implementeras med, `AVL_Tree_Node`.
- Ett testprogram finns på filen `avl_tree-test.cc`. Det sätter in några värden i ett AVL-träd och sedan kan en del av operationerna på AVL-trädet testas.
- En funktion for borttagning i ett enkelt binärt sökträd finns på filen `simple_remove.cc`.
- För att kompliera programmet finns en make-fil, `Makefile`. Kommandot `make` kör den.

Uppgift: *Din uppgift är att komplettera AVL-trädet med operationen `remove`, i analogi med hur övriga operationer är impementerade, t.ex. `insert`. I det givna progammet är `remove` deklarerad i klassen `AVL_Tree`, men kastar bara ett undantag om funktionen anropas. Någon motsvarande `remove()` i klassen `AVL_Tree_Node` är inte deklarerad (men det ska du göra).*

Operationen `remove` ska ta bort angivet element (om det finns). Utgå från den givna funktionen som finns på filen `simle_remove.cc`. Komplettera med balansjustering. Studera hur balanskontroll och justering görs i `insert` och gör i princip samma sak. Tänk dock noga igenom var en obalans ligger i förhållande till noden som har obalans och hur man kan avgöra om en enkel-rotation eller en dubbelrotation ska göras! På den punkten skiljer det påtagligt från vad som gäller vid insättning, även om koden ska bli snarlik. Nedan finns tre testfall din kod måste klara av. Observera att din kod naturligtvis måste fungera även i det generella fallet och inte bara på de tre testfallen, så tänk noga på vilka olika situationer som kan uppkomma och hur de ska hanteras.



Figur 1: 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.



Figur 2: Testfall 3 är borttagning av nod 9 i trädet ovan till vänster. Trädet kan byggas genom att köra `avl_tree-test` 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.

Redovisning: *Koden för funktionen* `remove` *som du skrivit själv för klassen* `AVL_Tree_Node` *ska klippas ut och läggas in i en separat fil som du skickar in till din assistent. Städa mappen där du har programmet för AVL-trädet, gör en tar-fil av mappen, komprimera tar-filen med gzip och skicka med också den filen till din assistent. Se även instruktionen för inlämning av labbar på sid. 4 i detta kompendium.*

Ni bör vara beredda att kunna dema utan stöd från labbkompisen (om ni jobbar i par), och att individuellt kunna besvara frågor från labbassistenten, t.ex:

- *Hur tar ni/du hand om ett AVL-träd med flera obalanser?*
- *Vad är maximala antalet rotationer man kan behöva göra i ett AVL-träd?*
- *Hur kan ni/du garantera att AVL-trädet är balanserat efter att ert/ditt program körts?*
- *Beskriv och dema ett extra trixigt fall av oblanserat AVL-träd som ni/du kan hantera förutom de enkla exempel som anges i dessa labbinstruktioner.*
- *Beskriv hur en rekursiv algoritm fungerar. Vad är det som gör att den är rekursiv?*
- *Vad är ett 'basfall' för något? Ge exempel!*
- *Hur beräknas höjden av ett träd? Vad har t.ex. nod 6 i figur 2a ovan för höjd?*

Lab 2: Knäcka lösenord

Mål: *Efter den här laborationen ska du kunna använda symboltabeller för att implementera en strategi för att knäcka lösenord krypterade med* Subset Sum*-metoden.*

Förberedelser: *Läs på om symboltabeller.*

När ett systems inloggningshanterare ställs inför ett lösenord måste en kontroll utföras så att det lösenordet stämmer med användarens lösenord i systemets interna tabeller. En naiv metod vore att lagra lösenorden i en symboltabell med användarnas namn som nycklar, men den metoden är känslig mot att någon otillåtet får tillgång till systemets tabell och därmed exponerar alla lösenord. I stället använder de flesta system en säkrare metod där en symboltabell med *krypterade* lösenord för varje användare används. Krypteringen är en envägsmappning, dvs. mappningen ska inte gå inte att reversera. När en användare skriver in ett lösenord krypteras det lösenordet och kontrolleras mot det lagrade (krypterade) värdet. Vid överensstämmelse släpps användaren in i systemet.

För att den här lösningen på inloggningsproblemet ska fungera väl behövs en krypteringsmetod med två egenskaper: Att kryptera ett lösenord bör vara enkelt och att lista ut ett lösenord givet den krypterade versionen bör vara mycket svårt.

Subset Sum-kryptering

En enkel lösenordshanteringsmetod är följande: Lösenordslängden sätts till ett specifikt antal bitar, säg N. Systemet håller reda på en tabell T med N heltal. För att kryptera ett lösenord använder systemet lösenordet för att välja en delmängd av talen i T och lägger samman dem; summan (modulo 2^N) är det krypterade lösenordet.

Följande miniexempel illustrerar processen. Antag att vi har lösenordet `00101`, och att systemets tabell innehåller följande fem tal:

```
0. 10110      0
1. 01101      0
2. 00101      1
3. 10001      0
4. 01011      1
```

Med den här tabellen skulle lösenordet `00101` krypteras som `10000` eftersom det säger att summan av rad två och fyra i tabellen ska användas (och `00101 + 01011 = 10000`). I praktiken skulle förstås en mycket större tabell användas.

Antag nu att du fått tillgång till systemtabellen T och att du också lyckas snappa upp användarnamn och det lagrade krypterade lösenordet. Den här informationen ger dig inte omedelbar tillgång till systemet; för att knäcka ett lösenord behöver du hitta en delmängd av T som summerar till det lagrade krypterade lösenordet. Säkerheten hos systemet beror på svårigheten i det här problemet (som på engelska går under namnet Subset Sum). Uppenbarligen måste ursprungliga lösenordet vara tillräckligt långt för att hindra dig från att helt enkelt prova alla möjligheter men samtidigt tillräckligt kort så att användare kan komma ihåg sitt lösenord. Med N bitar finns det 2^N olika delmängder, så man skulle kunna tänka sig att 40 eller 50 bitar borde räcka...

Detaljer

I stället för att använda heltal som lösenord är det brukligt att använda någon behändig översättning från det användaren skriver in till heltal. Här har vi valt att använda ett alfabet med 32 tecken (små engelska bokstäver samt de första sex siffrorna) i lösenorden och att koda varje tecken som 5-bitars heltal (`unsigned chars`) med 00000 för att koda "a", 00001 för att koda "b", 00010 för att koda "c", och så vidare. Om ett lösenord består av C tecken, måste vår krypteringstabell innehålla N rader, där $N = 5 \cdot C$.

På katalogen `/home/TDDI16/kursbibliotek/lab/lab2/` finns programmet `encrypt.cc`, med tillhörande `Makefile`, som en systemadministratör skulle kunna använda för att kryptera en användares lösenord. Programmet använder `Key.h` för att utföra addition av C-siffrors heltal i bas 32. Notera att omvandling till binär representation behövs för att kunna adressera rader i krypteringstabellen T. Programmet läser in tabellen T och använder sedan bitarna i lösenordet för att välja ord ur tabellen att addera och skriver sedan ut summan, dvs. det krypterade lösenordet. I katalogen finns ett flertal krypteringstabbeller förberedda: `easy5.txt`, `easy8.txt`, `rand5.txt` och `rand8.txt`. Den första och tredje är för nycklar med 5 `unsigned chars` (25 bitar), den andra och den fjärde är för nycklar med 8 `unsigned chars` (40 bitar). De första två har mycket struktur medan de andra två är slumpmässigt genererade.

Med åttabokstävers lösenord (`C = 8`) blir resultatet följande för lösenordet `password`:

```
>./encrypt password < rand8.txt
   password  15  0 18 18 22 14 17  3   0111100000100101001010110011101000100011
 1 gobxmqkt   6 14  1 23 12 16 10 19   0011001110000011011101100100000101010011
 2 qdrvjxwz  16  3 17 21  9 23 22 25   1000000011100011010101001101111011011001
 3 joobqxtz   9 14 14  1 16 23 19 25   0100101110011100000110000101111001111001
 4 xnoixmnk  23 13 14  8 23 12 13 10   1011101101011100100010111011000110101010
10 tcixtvem  19  2  8 23 19 21  4 12   1001100010010001011110011101010010001100
13 lqtsdtca  11 16 19 18  3 19  2  0   0101110000100111001000011100110001000000
15 zlptzlfp  25 11 15 19 25 11  5 15   1100101011011111001111001010110010101111
18 gmjuvyqw   6 12  9 20 21 24 16 22   0011001100010011010010101110001000010110
20 uoqrdhwp  20 14 16 17  3  7 22 15   1010001110100001000100011001111011001111
22 ltdkzndz  11 19  3 10 25 13  3 25   0101110011000110101011001011010001111001
23 btezrznq   1 19  4 25 17 25 13 16   0000110011001001100110001110010110110000
26 bujilqno   1 20  9  8 11 16 13 14   0000110100010010100000101111000001101011110
27 qgaicljl  16  6  0  8  2 11  9 11   1000000110000000100000010010110100101011
28 yyefwcld  24 24  4  5 22  2 11  3   1100011000001000010110110000100101100011
30 gnvowyjk   6 13 21 14 22 24  9 10   0011001101101010111010110110000100101010
34 aynzobxh   0 24 13 25 14  1 23  7   0000011000011011100101110000011011100111
38 lxwewfhh  11 23 22  4 22  5  7  7   0101110111101100010010110001010011100111
39 aenipbjd   0  4 13  8 15  1  9  3   0000000100011010100001111000010100100011
   vbskbezp  21  1 18 10  1  4 25 15   1010100001100100101000001001001100101111
```

Ovan ser vi överst lösenordet i klartext och underst det krypterade lösenordet. Däremellan ser vi de rader ur systemtabellen T som lösenordet plockar ut för att adderas. En attackerare som har kommit över tabellen ställs inför uppgiften att lista ut vilka rader som använts för att få den slutliga summan på nedersta raden. Hade vi använt en mer regelbunden tabell `easy8.txt` hade attackerarens uppgift blivit betydligt lättare (se nästa sida):

```
>./encrypt password < easy8.txt
   password  15  0 18 18 22 14 17  3   01111000001001010010101100111010001000011
 1 aaaaaaac   0  0  0  0  0  0  0  2   00000000000000000000000000000000000000010
 2 aaaaaaae   0  0  0  0  0  0  0  4   00000000000000000000000000000000000000100
 3 aaaaaaai   0  0  0  0  0  0  0  8   00000000000000000000000000000000000001000
 4 aaaaaaaz   0  0  0  0  0  0  0 25   00000000000000000000000000000000000011001
10 aaaaabaa   0  0  0  0  0  1  0  0   00000000000000000000000000000010000000000
13 aaaaaiaa   0  0  0  0  0  8  0  0   00000000000000000000000000010000000000000
15 aaaabaaa   0  0  0  0  1  0  0  0   00000000000000000000000010000000000000000
18 aaaaiaaa   0  0  0  0  8  0  0  0   00000000000000000000010000000000000000000
20 aaabaaaa   0  0  0  1  0  0  0  0   00000000000000000010000000000000000000000
22 aaaeaaaa   0  0  0  4  0  0  0  0   00000000000000000100000000000000000000000
23 aaaiaaaa   0  0  0  8  0  0  0  0   00000000000000001000000000000000000000000
26 aacaaaaa   0  0  2  0  0  0  0  0   00000000000001000000000000000000000000000
27 aaeaaaaa   0  0  4  0  0  0  0  0   00000000000010000000000000000000000000000
28 aaiaaaaa   0  0  8  0  0  0  0  0   00000000001000000000000000000000000000000
30 abaaaaaa   0  1  0  0  0  0  0  0   00000000100000000000000000000000000000000
34 azaaaaaa   0 25  0  0  0  0  0  0   00000110010000000000000000000000000000000
38 iaaaaaaa   8  0  0  0  0  0  0  0   01000000000000000000000000000000000000000
39 zaaaaaaa  25  0  0  0  0  0  0  0   11001000000000000000000000000000000000000
   b0onjjbh   1 26 14 13  9  9  1  7   00001110100111001101010010100100000100111
```

Attackeraren hade med den här mer regelbunden tabellen kunnat nysta upp summan på sista raden, genom att gå bakifrån i summan och bit för bit identifiera de rader i T vars 1:a matchar motsvarande bit i summan. Börjar man med 0 som delsumma, kan man gå bit för bit, och för en given delsumma peka ut rätt rad i tabellen som måste adderas till den för att få rätt bit på nästa position i totalsumman. Det fungerar även att jobba direkt med heltalskoderna. T.ex. för att få en 7:a, behöver man addera siffrorna $(2 + 4 + 8 + 25)$ mod 32. Man får då lägga en 1:a på minnet när föregående siffra i den delsumma man bygger upp hamnat över 32.

En lösning

I generella fallet kan man inte anta att krypteringstabellen T är regelbunden. Programmet `brute.cc` är ett första försök att lösa problemet att givet ett krypterat lösenord och en tabell hitta ursprungslösenordet. Lösningen består helt enkelt i att testa att summera alla möjliga delmängder (kombinationer av rader) i tabellen.

Med C definierad till 5, blir resultatet följande för lösenordet `passw`:

```
>make encrypt
>./encrypt passw < rand5.txt
passw  15  0 18 18 22   01111000001001010010110
exvx5   4 23 21 23 31   00100101111010110111111111

>make brute
>./brute exvx5 < rand5.txt
i0ocs
passw
```

Observera att det inte nödvändigtvis finns en unik lösning, så programmet skriver ut allihop. Notera också att den här lösningen inte går att använda för längre lösenord. För till exempel lösenord med 10 bokstäver finns det 32^{10} (över 1 biljard) möjliga delmängder att testa.

Hashtabellslösning

Uppgift: *Din uppgift är att skriva ett snabbare program för att knäcka lösenord,* `decrypt.cc` *som är funktionellt ekvivalent med* `brute.cc`*, men klarar av lösenord med 10 bokstäver på 400 s.*

Observera att din lösning måste använda datatypen `Key`*, assistenterna har inte möjlighet att lägga ned den tid som krävs för att sätta sig in i och rätta helt egna lösningar på uppgifterna.*

För att få till lösningen i uppgiften ovan ska du använda en hashtabell. Grundidén är att ta en delmängd S av tabellen T, beräkna alla möjliga delmängdssummor som kan skapas från S, sätta in de värdena i en symboltabell och använda den symboltabellen för att kolla alla de möjligheterna med bara en uppslagning.

När man betraktar strategin vi just skissat dyker flera frågor upp: Hur stor bör S vara? Exakt hur ska den här en-uppslagningskontrollen fungera? Vilken ADT bör symboltabellen baseras på? Vilken algoritm skulle vara bäst? Att handskas med de här detaljerna utgör den största delen av ditt arbete med den här uppgiften.

Det är lämpligt att debugga din lösning för lösenord med 6 bokstäver, för att sedan gå vidare till lösenord med 8 respektive 10 bokstäver. Ditt mål är, naturligtvis, att kunna dekryptera godtyckligt lösenord som krypterats med `encrypt.cc`, som till exempel:

```
>make decrypt
>./decrypt vbskbezp < rand8.txt
koaofbmx
password
xvbyofnz
1p1ngsgg
```

Den här strategin fungerar naturligtvis inte när lösenordslängden ökar, men den minskar arbetsmängden tillräckligt för att göra system som använder den här krypteringsmetoden sårbara.

Följande lösenord har krypterats med `rand8.txt`, `rand10.txt`, respektive `rand12.txt`:

`xwtyjjin`	`h554tkdzti`	`uz1nuyric5u3`
`rpb4dnye`	`oykcetketn`	`xnsriqenxw5p`
`kdidqv4i`	`bkzlquxfnt`	`4l4dxa3sqwjx`
`m5wrkdge`	`wixxliygk1`	`wuupk1ol3lbq`

Redovisning: *Skicka in ditt program* `decrypt.cc` *tillsammans med en ifylld* `readme.txt` *där du svarat på frågor om din implementation (mall finns på kursens labbkatalog). Se även instruktionen för inlämning av labbar på sid. 4 i detta kompendium.*

Ni bör vid själva demotillfället vara beredda att kunna dema utan stöd från labbkompisen (om ni jobbar i par), och att individuellt kunna besvara frågor från labbassistenten, t.ex:

- *Har ni lyckats sänka tidskomplexiteten (i genomsnittfallet) hos er lösning, jämfört med brutelösningen?*
  - *Om ja, på vilket sätt?*
  - *Om nej, varför inte?*
- *På vilket sätt har ni mött kravet att använda en hashtabell i er lösning? Gå muntligt igenom i koden hur er hashtabell fungerar.*
  - *Vad har ni använt för hashfunktion?*
- *Vad är minneskomplexiteten hos er lösning (i genomsnittfallet)?*
  - *Hur kan man härleda / motivera denna analys?*
- *Vad är den springande punkten i er lösning, som gör att den går att köra på rimlig tid?*
- *Får ni alltid samma körningstid (mätt i sekunder) på samma problem (dvs. samma input)?*
  - *Om det finns variationer, vad kan det bero på?*

Lab 3: Simulering

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.*

Den här laborationen är en anpassad version av en mycket mer omfattande labbserie i Java som ges vid Princeton University av Robert Sedgewick.

Förberedelser: *Läs på om prioritetsköer i OpenDSA.*

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.

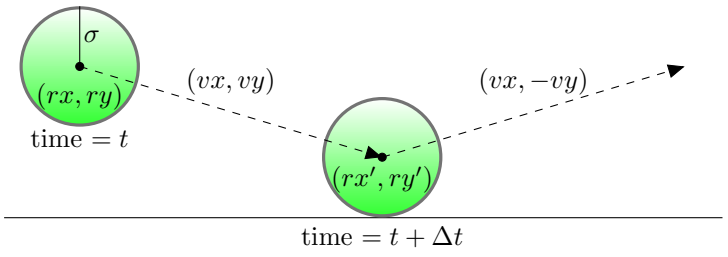
- *Tidsdriven simulering.* Diskretisera tiden i kvanta av storlek dt. Uppdatera varje partikels position efter dt tidssteg och kolla efter överlapp. Om det finns två överlappande partiklar, backa klockan till tidpunkten för kollisionen och förtsätt simuleringen. Det här sättet att simulera innebär att 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 an 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.

Simulering av partikelkollisioner i C++

Vår implementation innefattar följande klasser: `MinPQ`, `Particle`, `Event` 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. Händelsen svarar mot en fysisk kollision om aktuellt antal kollisioner för partiklarna är samma som när händelsen skapades, dvs de inblandade partiklarna har inte varit inblandade i några störande kollisioner.

Katalogen `/home/TDDI16/kursbibliotek/lab/lab3/` innehåller, förutom klasserna ovan, filen `simulation.cc` som innehåller ett program som driver simuleringen, samt en make-fil. Kommandot `make` kompilerar simuleringsprogrammet. Du kan nu ge kommandot

`>./simulation` *`N`*

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

`>./simulation <` *`file`*

läser in partikeldata från filen *`file`* och kör simuleringen. I labbkatalogen finns tre datafiler: `billiards10.txt`, `diffusion.txt` och `brownian.txt`. 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.

Uppgift: *På filen `MinPQ.h` finns den prioritetskö simuleringsprogrammet använder sig av. Din uppgift är att implementera* lat insättning *i prioritetskön.*

Funktionen `toss` kastar bara ett undantag, men ska istället göra lat insättning. Det betyder att `toss(x)` bara ska sätta in `x` på första lediga plats i heaparrayen (utan att fixa till heapen) och eventuellt uppdatera instansvariabeln `orderOK` som håller reda på ifall heaparrayen uppfyller heapegenskapen eller ej. Du kommer också att behöva ändra på flera platser i `MinPQ.h` för att implementationen ska bli korrekt. I labbkatalogen finns också filen `TestMinPQ.cc` som innehåller ett program som testar ifall prioritetskön fungerar som den ska. Anropet `g++ TestMinPQ` kompilerar testprogrammet.

När du fått prioritetskön att fungera med lat insättning är det dags att ersätta alla anrop till `pq.insert` med `pq.toss` i `CollisionSystem.cc` och kontrollera att simuleringsprogrammet fortfarande fungerar som det ska. Observera att `insert` och övriga metoder i `MinPQ.h` fortfarande måste fungera korrekt trots att lat 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: *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.
- En röd partikel i rörelse, N blå partiklar i vila.
- 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.

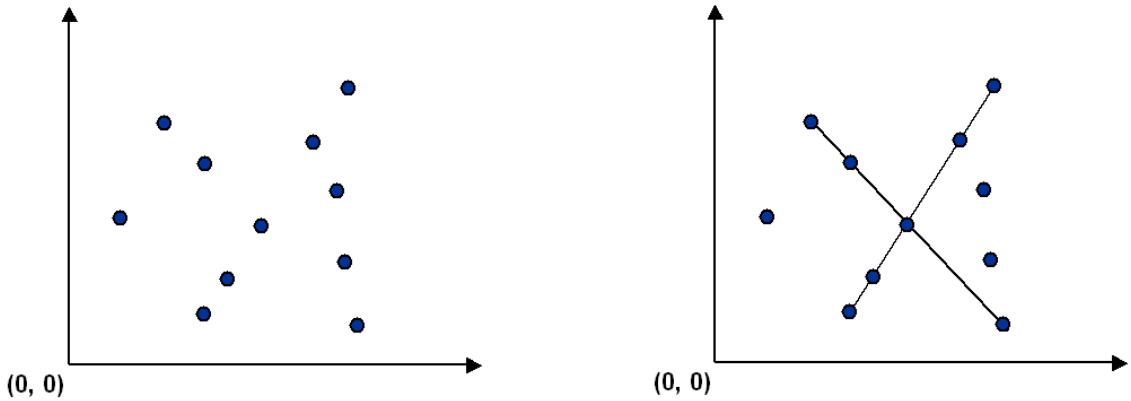
Redovisning: *Följ instruktionen för inlämning av labbar på sid. 4 i detta kompendium. Du behöver bara redogöra för dina ändringar av `MinPQ.h`, inte de ändringar du gjort i `CollisionSystem.cc`. Skicka också med dina partikeldatafiler till din assistent. Ge en kort beskrivning av varje datafil i inskickningen.*

Lab 4: Mönsterigenkänning

Mål: *Efter den här laborationen skall du kunna använda sortering för att konstruera ett effektivt program som känner igen linjemönster i en given mängd av punkter.*

Förberedelser: *Läs avsnitten om sortering i OpenDSA. Läs särskilt på om* `sort` *i STL.*

Givet en mängd av `N` distinkta punkter i planet, hitta alla (maximala) linjesegment som innehåller en delmängd av 4 eller fler av punkterna.



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.

Punktdatatyp

I mappen `/home/TDDI16/kursbibliotek/lab/lab4/` finns datatypen `Point` som representerar en punkt i planet genom att implementera följande API (utvalda delar):

```
// konstruera punkten (x, y)
Point(unsigned int _x, unsigned int _y);
// lutningen mellan this och punkten that
double slopeTo(const Point& that) const;
// rita ut punkten på ytan s
```

```
void draw(SDL_Surface* s) const;
// rita linjen mellan this och punkten that på s
void lineTo(SDL_Surface* s, const Point& that) const;
// jämför punkterna p1 och p2 lexikografiskt
friend bool operator<(const Point& p1, const Point& p2);
```

Totalsökning

Programmet `brute.cc` finns också på labbens kurskatalog, tillsammans med en `Makefile` för att kompilera programmet. Dessutom finns en mängd testdata i katalogen. `brute.cc` läser in punkter från `stdin` och ritar upp dessa. Sedan söker programmet igenom alla kombinationer av 4 punkter och kontrollerar om de ligger på samma linjesegment. Om en sådan kombination hittas ritas en linje mellan ändpunkterna av linjesegmentet ut. En liten optimering är att låta bli att undersöka om 4 punkter är linjärt beroende om de 3 första inte är det. Programmet skriver också ut information om hur lång tid beräkningen tagit.

Det är nu dags att du undersöker källkoden, kompilerar programmet och experimenterar med några av indatafilerna. Nedanstående kommande kör programmet med punkterna som finns i `input150.txt`:

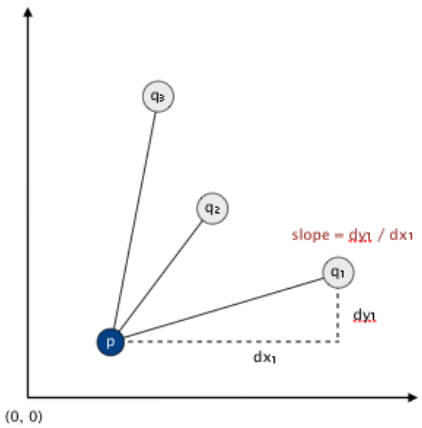
```
>./brute < input150.txt
```

Uppgift: *Experimentera med programmet och gör en analys av dess tidskomplexitet. Redogör för experimenten och den teoretiska analysen genom att fylla i filen* `readme.txt` *för totalsökningslösningen.*

En snabbare, sorteringsbaserad, lösning

Anmärkningsvärt nog går det att lösa problemet på ett mycket effektivare sätt än med totalsökningslösningen. Givet en punkt p, bestämmer följande metod om p finns med i en mängd av 4 eller fler linjärt beroende punkter.

- Tänk på p som origo.
- För varje annan punkt q, bestäm lutningen den har gentemot p.
- Sortera punkterna efter vilken lutning de har gentemot p.
- Kontrollera om 3 (eller fler) på varandra följande punkter i den sorterade ordningen har samma lutning gentemot p. Om det är så är dessa punkter, tilsammans med p, linjärt beroende.



Uppgift: *Skriv ett program* `fast.cc` *som implementerar ovanstående algoritm. Utför samma typ av analys som för totalsökningsalgoritmen och fyll i resultaten i* `readme.txt`*. Det är snyggare (och effektivare) om* `fast.cc` *inte ritar ut delsegment av linjesegment innehållande 5 eller fler punkter, men det är inte ett absolut krav.*

Tips

En komplicerande faktor är att jämförelsefunktionen i den sorteringsbaserade lösningen behöver ändras från sortering till sortering. En variant är att skapa ett funktionsobjekt som överlagrar `operator()` och som kan användas för att göra jämförelser. Fundera på om sorteringen behöver någon ytterligare egenskap.

Redovisning: *Koden i* `fast.cc` *som du skrivit själv ska klippas ut och läggas i en separat fil som du skickar in till din assistent tillsammans med den ifyllda* `readme.txt`*. Följ i övrigt instruktionen för inlämning av labbar på sid. 4 i detta kompendium.*