

Programmeringsteknik fördjupningskurs

Datorlaborationer

EDAA01

<http://cs.lth.se/edaa01ht>



LUNDS UNIVERSITET
Lunds Tekniska Högskola

HT 2021

Innehåll

| | | |
|---------------------|---------------------------------------|----|
| <i>Laboration 1</i> | använda abstrakta datatyper | 5 |
| <i>Laboration 2</i> | länkad lista | 15 |
| <i>Laboration 3</i> | grafiska användargränssnitt | 20 |
| <i>Laboration 4</i> | rekursion | 25 |
| <i>Laboration 5</i> | binära sökträd | 29 |
| <i>Laboration 6</i> | hashtabell | 32 |

Laboration 1 – använda abstrakta datatyper

Mål: Du ska träna på att implementera algoritmer med hjälp av de abstrakta datatyperna lista, mängd och map. Du ska också träna på att använda interface och klasser från Java Collection Framework som beskriver respektive implementerar dessa abstrakta datatyper.

Under laborationen kommer du att träffa på en hel del nya klasser. Meningen är att du ska få ett första smakprov av hur klassbiblioteket Java Collection Framework kan användas. Under kursen kommer vi sedan att se närmare på varje abstrakt datatyp och kommer då att diskutera hur motsvarande klasser i JFC ser ut under huven och hur man ska använda dem.

Ytterligare ett mål med laborationen är att du ska repetera/komma igång med att tyda felutskrifter samt använda en debugger för att hitta fel i program.

Förberedelser

F1. Den här uppgiften går ut på att tolka felutskrifter vid exekvering av ett program. Samtidigt kan det bli en repetition om du känner dig ringrostig när det gäller programmering och Java. Programmet, som är färdigt men innehåller några fel, ska lösa följande problem:

Ett antal arbetsuppgifter, "jobb", som tar olika lång tid ska fördelas på ett antal maskiner. Man vill göra ett schema för vilken maskin som ska göra vilket jobb på så sätt att alla jobb är klara så tidigt som möjligt. Exempel på ett schema med 3 maskiner (1-3) och 7 jobb (j1-j7), jobbets tidsåtgång står inom parentes:

| | | | |
|----------------|---------------------|--------------------|---------------------------------------|
| M ₁ | j ₄ (16) | | |
| M ₂ | j ₂ (14) | | j ₇ (3) |
| M ₃ | j ₅ (6) | j ₆ (5) | j ₃ (4) j ₁ (2) |

Ett enkelt sätt att lösa uppgifter av denna typ är att fördela jobben i avtagande ordning efter tidsåtgång och att ge varje jobb till den maskin som har minst att göra. Denna schemalägningsalgoritm kallas LPT-algoritmen (Longest Processing Time) och har använts för att skapa schemat i figuren. Algoritmen ger inte alltid den optimala lösningen, men ett tillräckligt bra resultat. Att räkna ut den optimala lösningen skulle ta alldeles för lång tid om antal maskiner och jobb är stort.

Bekanta dig med filerna finns i paketet 1pt i projektet för laborationen. Det finns fel i programmet som orsakar exekveringsfel. Kör programmet och studera det felmeddelande som dyker upp. På första raden visas vilken typ av fel (subklass till Exception) som inträffat. Därefter visas stacktrace – en sekvens av de metodanrop som var aktiva när felet inträffade. Fyll i information om felet i tabellen. Rätta felet.

| Feltyp | Felet uppträder i klassen | i metoden | på raden | Orsak till felet |
|---------------|---------------------------|-----------|----------|------------------|
| NPE | scheduler | multid | 19 | konstruktorn |
| out of bounds | -11- | print | 49 | for loop |


Kör programmet igen och fyll i informationen om det nya exekveringsfel i tabellen. I det andra felmeddelandet visas ett tal sist på raden med feltypen. Vad betyder detta tal? Rätta felet.

Svar: ?? rad? fattar inte vilken siftra som menar,

- F2. Programmet från förra uppgiften går nu att köra, men innehåller också några logiska fel. Utskriften från programmet ska bli:

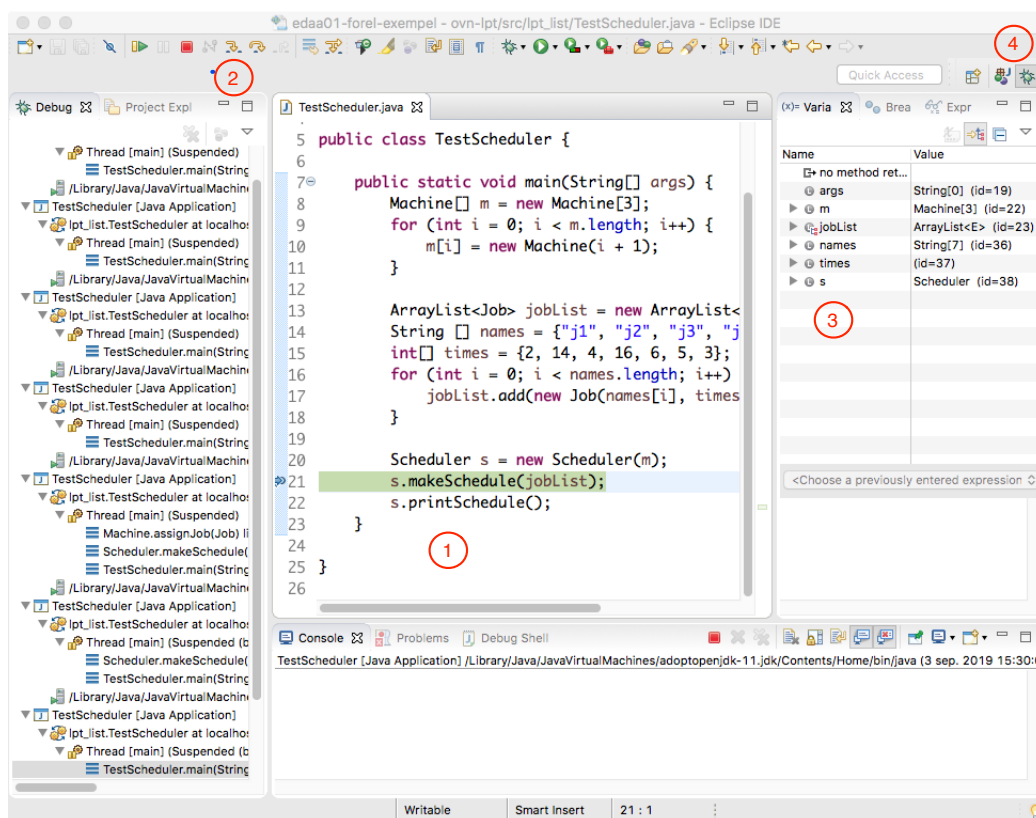
```
Maskin 1 har total schemalagd tid 16 [j4 16]
Maskin 2 har total schemalagd tid 17 [j2 14, j7 3]
Maskin 3 har total schemalagd tid 17 [j5 6, j6 5, j3 4, j1 2]
```






Kör programmet och se hur utskriften skiljer sig från det önskade resultatet. Du ska nu använda debuggern för att hitta fel. En debugger är ett verktyg för att köra programmet interaktivt. Man kan köra programmet stegvis eller till en viss brytpunkt, se variablers värde mm. Vi börjar med att "gissa" var felet finns och koncentrerar oss på metoden `makeSchedule`.

1. Man sätter en brytpunkt på en rad genom att dubbelklicka till vänster om raden. Sätt en brytpunkt på den rad i `main`-metoden där `makeSchedule` anropas. En blå prick ska då synas.
2. Starta debuggern. Det gör man genom att högerklicka på filen och välja `Debug As > Java Application`. Eller klicka på Debug-ikonen  i verktygsraden. Eclipse byter till debug-perspektivet. Fönstrets nya utseende visas i figur 1.

I mitten finns editorfönstret (1). Den rad som står i tur att exekveras är markerad. På verktygsraden (2) finns ikoner för kommandon. De viktigaste är:

Figur 1: Debug-perspektivet



-  Resume — fortsätt exekveringen till nästa brytpunkt eller till programmets slut.
-  Terminate — avsluta exekveringen av programmet.
-  Step over — exekvera en rad i programmet.
-  Step into — exekvera till nästa rad, gå in i metoder som anropas.
-  Step return — exekvera till metodens slut.

I fliken Variables (3) kan man se variabler och parametrar och deras värden. Om man byter till fliken Breakpoints visas de brytpunkter man satt.

Längst upp till höger (4) kan man byta mellan olika perspektiv (här mellan Java- och Debug-perspektivet).

3. Kontrollera i fliken Variables att variablerna `m`, `jobList` och `s` har initierats på rätt sätt. Om man klickar på pilen framför ett objekt ser man in i objektet. Om man istället klickar på dess namn visas innehållet nederst i Variables.
 - Klicka på `m` – Vektorn ska innehålla tre maskiner med schemalagd tid 0.
 - Klicka på `jobList` – listan ska innehålla sju jobb.
 - Klicka på `s` – Den utskrift som då visas är inte speciellt informativ. Det beror på att debuggern anropar objektets `toString`-metod. Eftersom `toString` inte är skuggad i klassen `Scheduler` används den `toString`-metod som ärvt från klassen `Object`. Klicka istället på pilen framför `s`. Sedan kan du klicka på dess attribut `machines` för att se att allt ser ok ut.
4. Allt verkar rätt så här långt. Byt till Java-perspektivet och tag bort brytpunkten genom att dubbelklicka på den. Öppna klassen `Scheduler` och sätt en ny brytpunkt på raden efter anropet av `sort` i metoden `makeSchedule`. Starta debuggern. Nu vill vi kontrollera om jobben sorterats på rätt sätt. Hur ser innehållet i `tempJobList` ut? Jobben ska sorteras efter avtagande tid, men det verkar inte bli så. Fundera var felet kan finnas och rätta det rätta felet. Vad orsakade felet?

Svar: `descTimeComp j1, j2`

Provkör programmet. Nu fungerar det bättre, men utskriften stämmer fortfarande inte.

5. Eftersom sorteringen nu verkar fungera är det dags att undersöka vad som händer i metoderna `machineWithLeastToDo` och `assignJob`. Kör programmet i debuggern och stega framåt (till en början med Step over) för att se var det går fel. Efter varje anrop av `machineWithLeastToDo` ska `m` innehålla en ref till den maskin som har minst att göra. Efter varje anrop av `assignJob` ska maskinens totala tid ha uppdateras och jobbet ska ha lagts till i listan. När du upptäcker något misstänkt går du nästa gång in i den metoden (med Step into) för att se vad som händer. Vad orsakade felet?

Svar: `assignJob`

Rätta felet. Nu bör programmet fungera som det ska.

En kommentarer: Observera att vi i och med detta inte alls kan garantera att programmet är korrekt. Det enda vi visat är att programmet fungerar för just detta exempel.

- F3. Under kursens gång ska vi lära oss saker som gör att klassen `Scheduler` kan förbättras (kortare kod, snabbare). Bland annat måste vi upprepade gånger söka efter den maskin

som har minst att göra. Finns det någon abstrakt datatyp som kan vara till hjälp för detta så att vi kan stryka metoden machineWithLeastToDo?

Svar:

Sorteringen kan också förenklas genom användning av lambda-uttryck och klassen DescTimeComp kan rationaliseras bort.

F4. Besvara frågorna med hjälp av kurslitteraturen, föreläsningbilderna och/eller dokumentationen för respektive interface.

a) Betrakta följande kodavsnitt:

```
List<Integer> nbrs = new ArrayList<Integer>();
for (int i = 0; i < 100; i += 10) {
    nbrs.add(i);
    nbrs.add(i);    // notera: talet läggs till två gånger
}

for (int a : nbrs) {
    System.out.println(a);
}
```

• Hur många rader skrivs ut?

Svar:

• Kan vi vara säkra på talens ordning i utskriften?

Ja ☒ Nej ☐

b) Anta att vi byter ut den första raden mot följande:

```
Set<Integer> nbrs = new HashSet<Integer>();
```

• Hur många rader skrivs nu ut?

Svar:

• Kan vi vara säkra på talens ordning i utskriften?

Ja ☐ Nej ☒

c) Betrakta följande kodavsnitt:

```
Map<String, Integer> m = new HashMap<String, Integer>();
m.put("albatross", 12);
m.put("pelikan", 27);
m.put("lunnefågel", 19);
m.put("albatross", 7);
System.out.println(m.get("albatross"));
```

• Fyll i typargument på de streckade raderna ovan, så att typerna stämmer överens med koden. (Du ska inte använda Object här.)

• Vad skrivs ut i exemplet?

Svar:

d) I interfacet Map finns en metod för att undersöka om en given nyckel förekommer. Vilken då?

Svar:

F5. Läs igenom texten under rubriken "Bakgrund".

F6. Lös uppgifterna D1, D2, D3 och D4.

F7. Läs igenom övriga uppgifter.

Bakgrund

Historiker och litteraturvetare använder ibland datorbaserade metoder för att få översiktlig information om stora textmassor. Genom att exempelvis räkna förekomster av ortnamn (och liknande) kan man skapa sig en uppfattning om den geografi som beskrivs i texten.¹

Med en sådan metod kan vi undersöka Selma Lagerlöfs bok *Nils Holgerssons underbara resa genom Sverige* och räkna förekomster av landskapens namn. På så sätt kan vi förstå något om den bild av Sverige som tecknas i boken. Lagerlöfs bok gavs ut 1906–1907, under den nationalromantiska perioden, då många av 1900-talets föreställningar om den svenska nationen tog form.

När vi undersöker en text, som Lagerlöfs bok, räknar vi alltså förekomster av vissa ord, som exempelvis platser. I den här uppgiften kommer du att konstruera klasser för att räkna ord på olika sätt. Vi kommer särskilt att fokusera på vilka abstrakta datatyper man kan använda.

Datorarbete

- D1. Börja med att bekanta dig med interfacet `TextProcessor`. Det innehåller metoder för att behandla inläst text, ett ord i taget, samt presentera ett resultat. Vi kommer att använda detta interface för att hantera olika slags textanalyser på ett enhetligt sätt.

```
public interface TextProcessor {

    /**
     * Anropas när ett ord lästs in.
     * Metoden ska uppdatera statistiken därefter.
     */
    void process(String w);

    /**
     * Anropas när samtliga ord i sekvensen lästs in.
     * Metoden ska skriva ut en sammanställning av statistiken.
     */
    void report();
}
```

(I just detta fall skulle vi även kunna använda en abstrakt klass, men interface passar bättre eftersom `TextProcessor` inte har några egna attribut eller metoder.)

- D2. I projektet finns också en klass `SingleWordCounter`, som implementerar interfacet ovan. Denna klass är till för att räkna hur många gånger ett givet ord förekommer.

Klassen innehåller ett fel, som gör att antalet alltid blir 0 (noll). Finn felet och åtgärda det.

- D3. Projektet innehåller därtill ett program `Holgersson.java`. Där skapas ett `SingleWordCounter`-objekt, som ska räkna antalet förekomster av ordet "nils". Vi stavar namnet med små bokstäver (gemener), eftersom programmet `Holgersson` omvandlar alla ord till gemener vid inläsning (med hjälp av `String`-metoden `toLowerCase`).

Därefter läses boken in. Bokens text finns i textfilen `nilsholg.txt`, och inleds med ett par dikter innan kapitel 1. Öppna gärna filen och se hur den ser ut.

Alla filens ord går igenom och vårt `SingleWordCounter`-objekt uppdateras. Slutligen skrivs resultatet ut.

¹ Ett intressant exempel på en sådan historisk undersökning, större än vad vi har möjlighet att göra här, är: Cameron Blevins, "Space, Nation, and the Triumph of Region: A View of the World from Houston": *Journal of American History*, vol. 101, no. 1, 2014.

Kör programmet och kontrollera att det utskrivna resultatet stämmer. Namnet "Nils" förekommer 75 gånger i Lagerlöfs roman. Om ditt resultat inte stämmer kan det bero på felet i föregående uppgift.

Kommentar till raden `s.useDelimiter(...)`: här konfigureras Scanner-objektet så att skiljetecknen `,. : ; ! ? ' "` filtreras bort från de inlästa orden. Anropet ser lite märkligt ut, eftersom vi använt ett *reguljärt uttryck* för att ange att alla dessa tecken ska betraktas som skiljetecken.² Du behöver inte bry dig om hur denna rad fungerar.

- D4. Inför fortsättningen vill vi utöka programmet Holgersson så att det kan hantera flera TextProcessor-implementationer på samma text.

Ändra programmet så att det har **en lista** av TextProcessor-objekt. Till att börja med ska listan bara innehålla det enda objekt som finns sedan tidigare (och som räknar "Nils").

Varje gång ett ord lästs in ska alla TextProcessor-objekt i listan få sin `process`-metod anropad. När all text har lästs in ska alla TextProcessor-objekt skriva ut sina respektive resultat.

Lägg till en rad i ditt Holgersson-program, så att även antalet förekomster av ordet "norge" räknas. Din lista ska alltså innehålla två TextProcessor-objekt, och du ska få följande resultat:

```
nils: 75
norge: 1
```

- D5. Hittills har vi behövt skapa ett nytt objekt för varje ord vi räknar. Nu vill vi införa en ny typ av textanalys, där inte bara ett enda ord räknas, utan flera. Vi ska räkna hur många gånger de olika svenska landskapen nämns i boken.

Skapa en ny klass MultiWordCounter, som implementerar interfacet TextProcessor och fungerar enligt följande:

- Konstruktorn ska ta en vektor av strängar som parameter. Vektorn innehåller de ord vi vill räkna. Följande exempel visar hur en sådan konstruktor ska fungera:

```
String[] landskap = { "blekinge", "bohuslän" /* , ... */ };
TextProcessor r = new MultiWordCounter(landskap);
```

- Din klass MultiWordCounter ska ha exakt **ett** attribut, och det attributet ska vara av typen Map (med lämpliga typargument – jämför med förberedelseuppgifterna). Detta Map-attribut används för att hålla reda på hur många gånger de sökta orden (landskapsnamn i exemplet ovan) förekommer. Inledningsvis innehåller denna Map värdet 0 (noll) för varje sökt ord (landskapsnamn). Metoden `process` ökar antalet om ett givet ord är ett av de sökta orden.
- Även om attributet har typen Map, så ska det objekt som skapas för det vara av den konkreta klassen HashMap. På så vis blir ditt program, så långt som möjligt, oberoende av vilken implementation av Map-interfacet som faktiskt används. Namnet HashMap ska alltså bara förekomma på **ett** ställe i klassen (förutom ev. import-satser).
- Metoden `report` ska skriva ut alla nycklar och respektive värden i din Map.

² Om du är särskilt intresserad kan du läsa mer om denna Scanner-finess här: <https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html>. Du kan även läsa mer om reguljära uttryck på exempelvis Wikipedia (https://en.wikipedia.org/wiki/Regular_expression).

Tips: Man kan gå igenom alla nycklar i en Map så här (om vi antar att `m` är en Map vars nycklar är av typ `String`):

```
for (String key : m.keySet()) {  
    // gör något med key och m.get(key)  
}
```

- D6. Lägg till ett `MultiWordCounter`-objekt för landskapen i din lista i programmet `Holgersson`. Notera att det finns en användbar strängvektor given i programmet.

Kör programmet. I resultatet ser vi att gränstrakterna (Skåne, Lappland) nämns relativt ofta. Kanske var det angeläget att visa att Sverige ännu var rätt stort, trots att Norge lämnat unionen året innan? (Norge nämns ju endast en gång.)

- D7. Vi ska nu ta fram information om boken på ett annat sätt. Genom att räkna alla ord, inte bara landskap, kan vi skapa oss en uppfattning om bokens innehåll. Vi måste emellertid utesluta vissa vanliga ord, som "och", "ett" och "att", för att få ett meningsfullt resultat. Vi behöver alltså återigen en tabell av ord, men nu för att räkna *alla* ord, *utom* ett antal undantagsord.

Skapa en klass `GeneralWordCounter`, som implementerar interfacet `TextProcessor` och fungerar enligt följande:

- Konstruktorn ska ta en mängd (`Set`) som parameter. Denna mängd innehåller undantagsord lästa från filen `undantagsord.txt`:

```
Scanner scan = new Scanner(new File("undantagsord.txt"));  
  
Set<String> stopwords = ...;           // en lämplig mängd skapas  
  
... ..                               // orden läses in från Scannern  
... ..                               // 'scan' och lagras i mängden  
  
TextProcessor r = new GeneralWordCounter(stopwords);
```

Filen `undantagsord.txt` finns i ditt Eclipse-projekt. Öppna gärna den och se hur den ser ut.

- Använd en Map för att hålla reda på hur många gånger respektive ord förekommer.
 - Även här ska du använda `HashMap`, men även här ska namnet `HashMap` bara förekomma på ett ställe i klassen (förutom ev. import-satser). För din mängd (`Set`) kan du välja implementation själv.
 - Metoden `process` räknar alla ord, såvida de inte finns i mängden av undantagsord. Första gången ett nytt ord upptäcks läggs det till med antalet 1, och påföljande gånger samma ord upptäcks ökas dess antal med ett.
 - Metoden `report` ska skriva ut alla ord som förekommer 200 gånger eller fler.
- D8. Lägg till ett `GeneralWordCounter`-objekt i din lista i programmet `Holgersson`. Kör programmet. Vilka är de vanligaste orden i Lagerlöfs bok?

Intresset för gränstrakterna framgår indirekt även här. Ledargåsen Akka är döpt efter ett lappländskt fjällmassiv, och pojken kommer från Skåne. Kanske skymtar vi bland orden även nationalromantikens fascination för den vilda naturen?

Mer om interface

- D9. När du implementerade metoden `report` ovan drog vi gränsen vid 200 ord. En sådan fix gräns är ju otillfredsställande: för en längre bok kanske utskriften blir ohanterligt lång, och om vi analyserar en kortare text kanske vi inte får något resultat alls. Istället vore det bättre att skriva ut, säg, de fem vanligaste orden.

För detta ska vi använda Javas inbyggda sorteringsalgoritm för listor. Det är en effektiv algoritm, som låter oss tillhandahålla en egen jämförelse för elementens ordning. Denna jämförelse beskrivs med hjälp av ett särskilt interface, och i denna och de följande två deluppgifterna ska du få se närmare hur detta fungerar.

Först behöver vi en lista av ord med tillhörande antal. Från vår `Map` finns inget sätt att skapa en lista, men vi kan få en *mängd* (`Set`) av sådana ord-antal-par med följande metod:

```
/** Returns a Set view of the mappings contained in this map. */
Set<Map.Entry<K,V>> entrySet();
```

Vi får då ut en mängd med objekt av typ `Map.Entry<String,Integer>`. Varje sådant objekt innehåller ett ord och tillhörande antal. När vi väl har en sådan mängd kan vi placera elementen i en lista, genom att skicka in mängden som parameter till listans konstruktor.

- Skriv in följande rader i klassen `GeneralWordCounter`, metoden `report`. Ta bort (eller kommentera ut) den kod som fanns där sedan tidigare.

```
Set<Map.Entry<String, Integer>> wordSet = counts.entrySet();
List<Map.Entry<String, Integer>> wordList = new ArrayList<>(wordSet);
```

- Lägg till kod i `report` (några rader) så att listans (`wordList`) fem första element skrivs ut.

Testa programmet och se att du får fem rader i utskriften från `GeneralWordCounter`. De fem raderna är inte de fem vanligaste orden, utan bara de fem ord som råkar hamna först i `Map`-implementationens ordning.

(Listan är ännu inte sorterad i rätt ordning, så detta är bara ett mellansteg, där vi kontrollerar att steget från `Map` till `List` fungerar.)

- D10. Nu ska vi använda Javas inbyggda sorteringsalgoritm. Vi kommer att göra det genom att göra små förändringar, steg för steg. I flera av stegen uppstår kompileringsfel (som sedan löses i nästa steg).

Försök förstå vad kompileringsfelen betyder! Fråga gärna din handledare.

- Skapa en ny, tom klass i Eclipse. Kalla klassen `WordCountComparator`.
- Lägg till följande rad efter kodraderna ovan:

```
wordList.sort(new WordCountComparator());
```

Vi anger alltså att ett `WordCountComparator`-objekt ska användas för att ange ordningen. Vi ska strax komma till hur det fungerar.

Du får ett kompileringsfel. Vad beror det på?

- Som framgår av kompileringsfelet behöver vår klass `WordCountComparator` implementera interfacet `Comparator`. Ändra i den nya klassen `WordCountComparator`, så att den börjar så här:

```
package textproc;

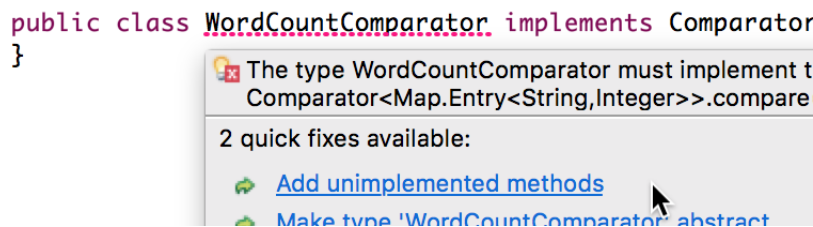
import java.util.*;

public class WordCountComparator implements Comparator<Map.Entry<String,Integer>>
```

Vi lägger alltså till ett interface för klassen (och tillhörande import-sats). På så vis anger vi att WordCountComparator kan användas för det slags jämförelser som sorteringen behöver. Vi lovar, med andra ord, att klassen har en metod compare, som är den som sorteringen behöver. (Att metoden heter compare beror på att det är så den heter i interfacet Comparator.)

Typargumentet ovan, Map.Entry<String,Integer>, betyder att vår klass WordCountComparator kan jämföra just sådana Map.Entry-objekt som vi har i vårt program.

Kompileringsfelet för sort försvinner. Istället får vi ett nytt kompileringsfel i klassen WordCountComparator, eftersom den ännu inte implementerar själva jämförelsen ("WordCountComparator must implement method..."). Klassen uppfyller ännu inte löftet.



Figur 2: Kompileringsfel, med "quick fix"-förslag, i Eclipse.

- Håll muspilen över kompileringsfelet i WordCountComparator och välj "Add unimplemented methods" i den lilla "quick fix"-listan som dyker upp (se figur 2 ovan).

Du får nu en automatgenererad compare-metod för jämförelsen. Eclipse har givetvis ingen aning om *vilken* ordning vi vill ha, men metoden går att kompilera.

Provkör programmet. Ordningen är fortfarande fel, men nu återstår endast en eller några få kodrader innan vi får en sorterad utskrift.

- D11. I metoden compare ska två parametrar, o1 och o2, jämföras. Javas sorteringsalgoritm anropar metoden upprepade gånger för två ord-antal-par, och sorterar elementen utifrån den ordning som vår compare-metod anger.

Metoden ska returnera ett värde enligt följande mönster:

- större än 0 om det första elementet är "större" än det andra, och
- mindre än 0 om det första elementet är "mindre" än det andra,
- exakt 0 om de två är att betrakta som "lika".

Här syftar "mindre" och "lika" på den önskade sorteringsordningen. Du känner säkert igen idén från metoden compareTo i klassen String.

Implementera metoden compare så att elementen sorteras med avseende på antal förekomster, i fallande ordning. Kör programmet och kontrollera resultatet.

Tips: Undrar du hur du får fram heltalet ur ett Map.Entry-objekt? Titta då gärna i specifikationen.³ Kom ihåg att det rör sig om ett par från en Map, där nycklarna är strängar och värdena heltal.

³ <https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html>

- D12. Justera din `compare`-metod (från uppgift D11) så att elementen sorteras i första hand på antal förekomster, och i andra hand i bokstavsordning.

Testa! Öka först antal ord som skrivs ut i metoden `report` i `GeneralWordCounter` så att du ser att din ändring i `compare` spelar någon roll.

Jämförelse av Map-implementationer

- D13. Man kan mäta exekveringstiden för ett Java-avsnitt med hjälp av `System.nanoTime`. Denna metod returnerar antalet nanosekunder som förflutit sedan någon ospecificerad tidpunkt. Genom att subtrahera två sådana tidpunkter kan man få ett mått på förfluten tid, exempelvis i millisekunder:

```
long t0 = System.nanoTime();
... // kod vars exekveringstid vi vill mäta
long t1 = System.nanoTime();
System.out.println("tid: " + (t1 - t0) / 1000000.0 + " ms");
```

5053

Justera programmet `Holgersson` så att tiden för programmet skrivs ut, så som ovan. Kör programmet tre gånger och välj medianvärdet av exekveringstiderna (det mellersta). Notera detta medianvärde.

- D14. Justera dina klasser `GeneralWordCounter` och `MultiWordCounter`, så att den använder `TreeMap` istället för `HashMap`. Om du har gjort rätt så räcker det ändra på ett ställe.

- Fungerar ditt program fortfarande?
- Hur påverkas ordningen i det utskrivna resultatet?
- Hur påverkas exekveringstiden? (Beräkna även här medianvärdet från tre körningar.)

54/58

- D15. Fundera igenom följande, och diskutera med din handledare:

- Vad är det för skillnad på `Map` och `HashMap`?
- Vad är det för skillnad på `HashMap` och `TreeMap`? Vad beror skillnaderna på?
- På laborationen har du bl.a. använt de abstrakta datatyperna `mängd` och `map`. Vad har de för speciella egenskaper som gör dem lämpliga att använda på det sätt som har gjorts?
- Du har även använt Javas inbyggda sortering med hjälp av en sorteringsordning du själv angivit. Hur fungerar det? Och vilken funktion har interfacet `Comparator` i detta?

Laboration 2 – länkad lista

Mål: Du ska lära dig implementera den abstrakta datatypen kö (FIFO queue) på två olika sätt; dels genom att delegera till Javas klass LinkedList och dels genom att implementera en från grunden med en länkad datastruktur. Du ska också lära dig att testa en klass genom att skriva testmetoder och använda testverktyget JUnit.

Förberedelser

F1. Läs igenom texten under rubriken "Bakgrund".

F2. Läs texten till uppgift D3 och studera fig. 1. Svara på följande frågor:

- a) Vilket värde ska attributet last ha i en tom lista ? Svar:
- b) Antag att listan inte är tom. Skriv klart följande tilldelningssats så att variabeln n refererar till första noden i kön. QueueNode<E> n =
- c) Antag att listan inte är tom. Skriv klart följande tilldelningssats så att variabeln e refererar till första elementet i kön. E e =

F3. Läs PM om JUnit som finns på kursens webbsida. Svara på följande frågor med hjälp av dokumentationen av JUnits klass Assert. Länk finns på kursens webbsida.

- a) Vilken metod är lämplig att använda om man vill kontrollera att ett logiskt uttryck har värdet false? Svar:
- b) Vilken metod är lämplig att använda om man vill kontrollera att två vektorer har identiskt innehåll? Svar:

F4. Lös uppgifterna D1, D2, D3 och D4.

F5. Läs igenom övriga uppgifter.

Bakgrund

På den här laborationen ska du på två olika sätt implementera en generisk klass `FifoQueue` med följande klassrubrik:

```
public class FifoQueue<E> extends AbstractQueue<E> implements Queue<E>
```

Klassen representerar en kö och ska implementera interfacet `Queue` i klassbiblioteket `java.util`.

Förklaring till varför klassen `FifoQueue` ärver `AbstractQueue`: Ibland kan man implementera vissa metoder med hjälp av andra metoder. Ex:

```
public boolean isEmpty () {  
    return size() == 0;  
}
```

För att underlätta för den som ska implementera det (stora) interfacet `Queue` finns den abstrakta klassen `AbstractQueue` som innehåller många av `Queue`-metoderna implementerade enligt detta mönster. Det som återstår att göra i klassen `FifoQueue` är att implementera metoderna `offer`, `size`, `peek`, `poll` och `iterator`.

```
/**  
 * Inserts the specified element into this queue, if possible.  
 * post: the specified element is added to the rear of this queue.
```

```

    * @param x the element to insert
    * @return true if it was possible to add the element to this queue, else false
    */
    boolean offer(E x);

    /**
     * Returns the number of elements in this queue.
     * @return the number of elements in this queue
     */
    int size();

    /**
     * Retrieves, but does not remove, the head of this queue,
     * or returns null if this queue is empty.
     * @return the head of this queue, or null if the queue is empty
     */
    E peek();

    /**
     * Retrieves and removes the head of this queue,
     * or returns null if this queue is empty.
     * post: the head of the queue is removed if the queue was not empty
     * @return the head of this queue, or null if the queue is empty
     */
    E poll();

    /**
     * Returns an iterator over the elements in this queue.
     * @return an iterator over the elements in this queue
     */
    Iterator<E> iterator();

```

En kommentar till metoden `offer`: enligt specifikationen ska det element som är parameter sätts in enbart om det möjligt. I beskrivningen av interfacet `Queue` i Java-dokumentationen kan man utläsa att det är tillåtet att införa begränsningar på köer, t ex att en kö bara får innehålla ett visst antal element. Om man anropar metoden `offer` när kön redan innehåller det maximalt tillåtna antalet ska i sådana fall ingen insättning göras och metoden ska returnera `false`. I vår implementering ska inte någon sådan begränsning göras. Metoden ska därför i klassen `FifoQueue` alltid sätta in elementet och returnera `true`.

Datorarbete

- D1. Först ska du implementera kö-klassen genom att delegera till klassen `LinkedList` i paketet `java.util`. I projektet för laborationen finns det i paketet `queue_delegate` en fil med namnet `FifoQueue.java`. I klassen finns attributet `list` som du ska använda för att hålla reda på elementen i kön.

Implementera alla metoderna. Du ska inte behöva lägga till mer än en rad i varje metod. Läs dokumentationen av `Queue` på nätet så att du väljer rätt metoder.

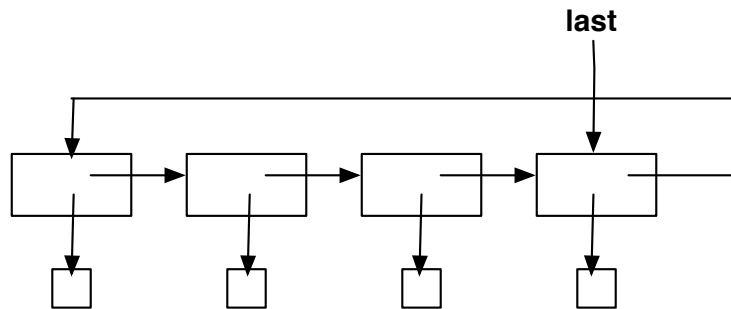
- D2. I filen `TestFifoQueue` i paketet `testqueue` finns det testmetoder som kontrollerar funktionaliteten hos de metoder som implementeras i denna uppgift. Bekanta dig med testklassen så att du förstår vad som testas.

Testa din kö-klass och rätta till den tills du får grönt ljus.

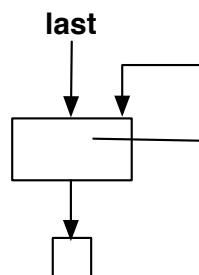
- D3. Nu ska du göra en ny implementering av klassen `FifoQueue`. En enkellänkad lista ska användas för elementen i kön. Noderna i listan ska representeras av följande privata nästlade klass (deklarerad i klassen `FifoQueue`):


```
private static class QueueNode<E> {
    E element;           // refererar till elementet
    QueueNode<E> next;   // refererar till efterföljande nod

    /* Konstruktör */
    QueueNode(E element) {
        this.element = element;
        next = null;
    }
}
```



Cirkulär lista att använda för att implementera en kö.
 next-referenser och referenser till insatta element är utritade.
 Noden längst till vänster innehåller det äldsta elementet i kön. Noden längst till höger, som attributet last refererar till, innehåller det senast insatta elementet.



Kö med ett enda element

Figur 1: Kö som representeras av cirkulär enkellänkad lista.

Listan ska vara *cirkulär*, dvs i den sista listnoden är inte referensen (next) till efterföljaren null utan i stället refererar den till det äldsta (första) listnoden i listan. I en tom kö har attributet last värdet null. Kön representeras i klassen `FifoQueue` av ett attribut (last), som refererar till den listnod som innehåller det sist insatta elementet. Se fig. 1. Det är *inte* tillåtet att lägga till ett extra attribut `first` i köklassen.

Observera att det bara är implementeringen av listan som är cirkulär. Utifrån sett är det en vanlig lista med början och slut.

I paketet `queue_singlelinkedlist` finns en fil med namnet `FifoQueue.java`. I klassen finns attributet last, metoderna som ska implementeras samt klassen `QueueNode`. Dessutom finns det ytterligare ett attribut `size` som representerar antalet element i kön.

Implementera alla metoder utom `iterator()`, som vi återkommer till i nästa uppgift. Testa metoderna parallellt. OBS! Ändra i testklassen så att du importerar `FifoQueue` från rätt paket.

Du bör börja med att implementera och testa metoderna `offer` och `size`. När du känner dig säker på att insättning fungerar kan du gå vidare till metoderna `peek` och `poll`.

- D4. I denna uppgift ska `iterator()` implementeras. Metoden ska returnera ett objekt av en klass som implementerar interfacet `Iterator<E>`. En skiss över hur detta kan göras följer:

```
public class FifoQueue<E> extends AbstractQueue<E> implements Queue<E> {
    ...
    public Iterator<E> iterator () {
        return new QueueIterator();
    }

    private class QueueIterator implements Iterator<E> {
        private QueueNode<E> pos;
        ...
        /* Konstruktör */
        private QueueIterator() {...}

        public boolean hasNext() {...}

        public E next() {...}
    }
}
```

Lägg märke till att i förslaget ovan är klassen `QueueIterator` en privat inre klass i klassen `FifoQueue`. Det innebär att man i `QueueIterator` har tillgång till alla attribut i sitt omgivande objekt av typen `FifoQueue`. Man kan alltså inne i ett objekt av typen `QueueIterator` använda attributen i klassen `FifoQueue`. Klassen `QueueIterator` och dess konstruktör kan vara privata eftersom det bara är den omgivande klassen som kommer att använda dem.

Läs specifikationen för metoderna i interfacet `Iterator<E>` i Javas dokumentation på nätet. Lägg in klassen `QueueIterator` i klassen `FifoQueue` enligt ovan och implementera konstruktorn, `hasNext` och `next`.

Testa!

- D5. Ibland behöver man slå samman (konkatenera) två köer `q1` och `q2` till en kö bestående av alla element i `q1` följda av alla element i `q2`. Om man bara har tillgång till de befintliga metoderna på listan kan man successivt ta ut elementen ur `q2` med metoden `poll` och sätta in dem i `q1` med metoden `offer`. Om det finns n element i `q2` anropas alltså båda metoderna n gånger.

OBS: Du ska göra en effektivare lösning genom att i stället utföra konkateneringen i en metod i klassen `FifoQueue`. Utnyttja den interna datastrukturen hos `FifoQueue` istället för att använda metoderna `offer` och `poll`.

Implementera följande metod i `FifoQueue`:

```
/**
 * Appends the specified queue to this queue
 * post: all elements from the specified queue are appended
 * to this queue. The specified queue (q) is empty after the call.
 * @param q the queue to append
 * @throws IllegalArgumentException if this queue and q are identical
 */
public void append(FifoQueue<E> q);
```

I kommentaren står att `IllegalArgumentException` ska genereras som `queue` och `q1` är identiska. Med det menas att man inte ska kunna slå ihop en kö med sig själv:

```
q1.append(q1);
```

Det betyder alltså *inte* att köernas innehåll ska jämföras. Tips! Utnyttja `this` i den jämförelse som behövs.

- D6. Skapa i paketet `testqueue` en fil `TestAppendFifoQueue.java` genom att i menyn `File` välja `New` → `JUnit TestCase`. Ange gärna i dialogen att den klass som ska testas är `queue.FifoQueue` så får du automatiskt inlagt en importsats i filen. (Om du inte anger detta kan du manuellt lägga till `import queue_singlelinkedlist.FifoQueue` i början av testklassen). Lägg i denna fil in test för `append`-metoden. Testen ska åtminstone täcka in följande fall för konkatenering:

- två tomma köer
- tom kö som konkateneras till icke-tom kö
- icke-tom kö som konkateneras till tom kö
- två icke-tomma köer
- försök att slå ihop en kö med sig själv

I testen ska du både kontrollera storlek *och* att elementen hamnat i rätt ordning. Glöm inte att kontrollera att den andra kön är tom efter sammanslagningen.

Kör testen och korrigera eventuella fel i `append`-metoden tills alla test lyckas.

- D7. Fundera igenom följande, och diskutera med din handledare.

- Skulle du lika gärna kunna använda `ArrayList` för att lagra elementen i `FifoQueue` i uppgift D1?
- Jämför de två olika sätten att implementera `FifoQueue` (uppgift D1 resp. uppgift D3). Fördelar/nackdelar?
- Istället för att implementera en egen kö-klass skulle man helt enkelt kunna använda någon av kö-klasserna i `java.util` (`LinkedList` eller `ArrayDeque`). Ofta är det klokt att återanvända en befintlig implementering på detta sätt. I vilka situationer kan det vara olämpligt?
- Vid testning av dina klass `FifoQueue` får du grönt ljus. Kan du då vara säker på att din klass är felfri?

Laboration 3 – grafiska användargränssnitt

Mål: Du ska träna på att skapa grafiska användargränssnitt och mönstret Model-View-Controller. Du ska också träna på att formulera lambdauttryck och använda dem för att bland annat hantera användarinteraktion på ett smidigt sätt.

Förberedelser

F1. Läs igenom texten under rubriken "Bakgrund".

F2. Sök upp dokumentationen för `Map.Entry`.⁴ Interfacet används för att beskriva ett par, av en nyckel och ett värde, i en `Map`. Bekanta dig med vilka metoder som finns i interfacet.

Det finns en metod i `Map` för att hämta en mängd (Set) av sådana `Map.Entry`-objekt. Vad heter metoden?

Svar: `entrySet()`

F3. Anta nu att vi har en lista (inte en mängd) av `Map.Entry` enligt ovan, med typargumenten `<String, Integer>`. Vi vill använda den inbyggda metoden `sort` för att sortera listan, i fallande ordning, med avseende på värden (inte nycklar).

Fyll i lambdauttrycket nedan så att ordningen blir den önskade.

```
List<Map.Entry<String, Integer>> list = ...; // listan skapas
list.sort((e1, e2) -> e1.getValue() - e2.getValue());
```

Du ska **inte** använda metoden `comparingByValue` i `Map.Entry`, utan ta tillfället att träna på att skriva lambdauttryck själv.

Tips: lambdauttrycket ska ge ett värde som är `<0`, `==0` eller `>0` beroende på hur de två jämförda elementen (`e1` och `e2`) förhåller sig till varandra. Det rör sig alltså om samma slags jämförelsevärde som exempelvis i metoden `compareTo` för strängar.

F4. Sök upp dokumentationen för följande Java-klasser på nätet:⁵ `JButton`, `TextField`, `JPanel` och `JList`.

Du behöver inte läsa alla detaljer, men försök skapa dig en idé om vad klasserna är till för. (**Tips:** jämför med bilden i figur 1 i följande avsnitt.) Svara också på följande frågor:

Med vilken metod anges vilken kod som ska köras när en `JButton` klickas? (**Tips:** titta i superklassen.)

Svar: `addActionListener`

Vilken metod returnerar texten som användaren har skrivit i ett `TextField`? (**Tips:** titta i superklassen.)

Svar: `getText()`

Med vilken metod lägger man till ett element i en `JPanel`? (**Tips:** titta i superklasserna.)

Svar: `add()`

Vilken metod använder man för att markera en viss rad i en `JList`?

Svar: `setSelectedIndex()`

F5. Lös uppgifterna D1, D2, D3 och D4.

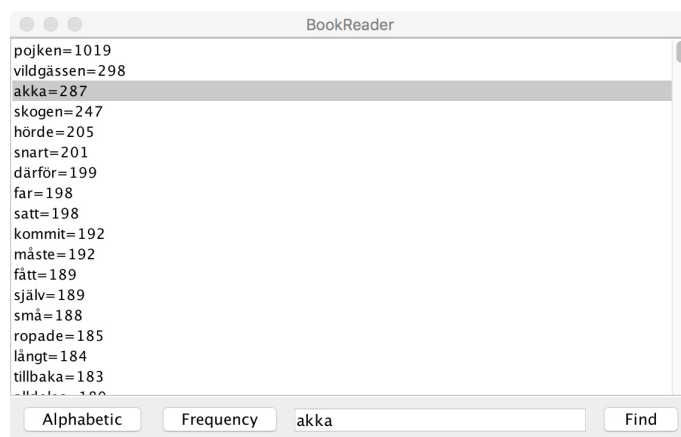
⁴ <https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html>

⁵ <https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/swing/JButton.html>
<https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/swing/TextField.html>
<https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/swing/JPanel.html>
<https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/swing/JList.html>

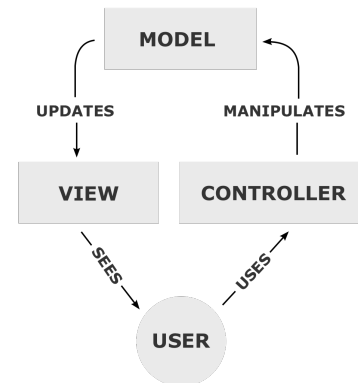
F6. Läs igenom övriga uppgifter.

Bakgrund

I den här laborationen kommer du att skapa ett grafiskt användargränssnitt till det textanalysprogram du konstruerade i laboration 1. En bild på hur det kan se ut visas i figur 1 nedan.



Figur 1: Grafiskt användargränssnitt för vårt textanalysprogram.



Figur 2: Idén bakom mönstret Model-View-Controller. (Källa: Wikipedia, public domain.)

Huvuddelen av fönstret upptas av en listvy (JList), där bokens ord och deras respektive antal finns listade. På varje rad står ett ord och motsvarande antal.

Under listvyn finns några knappar och ett textfält. De två knapparna till vänster låter oss välja hur listan ska sorteras (alfabetiskt eller efter antal förekomster). I textfältet kan man skriva in ett ord, och när man trycker på knappen "Find" söks motsvarande ord i listan upp och markeras.

En lista av ord och deras respektive antal kan vi hämta från klassen `GeneralWordCounter` från laboration 1. För listan behöver vi bestämma oss för en elementtyp. Vi kommer att se i datoruppgifterna nedan att det är praktiskt att använda typen `Map.Entry` (med lämpliga typargument) till detta. När innehållet i denna lista ändras, t ex genom att sorteras, vill vi att listvyn i fönstret uppdateras. För att lösa detta måste listvyn kopplas ihop med en listmodell. Listmodellens uppgift är att hålla reda på vår lista med ord-antal-par samt meddela listvyn när innehållet i listan är förändrat. Vi kommer att lösa det genom skriva en subklass till klassen `AbstractListModel`⁶.

Ett sätt att strukturera grafiska applikationer

Program med grafiska användargränssnitt struktureras ofta enligt ett mönster som kallas *Model-View-Controller* (se figur 2 ovan).⁷ Vi delar då in klasserna i tre olika delsystem, där varje delsystem består av en eller flera klasser:

⁶ <https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/swing/AbstractListModel.html>

⁷ Mer att läsa finns exempelvis på Wikipedia: <https://en.wikipedia.org/wiki/Model-view-controller>

Modellen hanterar data och algoritmer i applikationen. Det är alltså en central del av applikationen. Vi eftersträvar att göra modellen *oberoende* av användargränssnittet (vy och kontroll), så att samma modell kan återanvändas även med ett annat användargränssnitt.

I vårt system utgörs modellen av den klass `GeneralWordCounter`, som du implementerade i laboration 1. Då använde du denna modell med ett textbaserat gränssnitt.

Vyn används för att visa modellens innehåll grafiskt. I applikationen du ska skriva visas t.ex. orden och deras frekvenser i fönstret.

Kontrollen hanterar indata från användaren och låter på så användaren interagera med modellen.

Vyns klasser består alltså av fönster, listvy, knappar, inmatningsfält och annat. Sådana klasser finns ofta tillgängliga i färdiga paket, som JavaFX, Android eller Swing. I denna laboration använder vi Swing, ett standardpaket i Java. Kontrollen utgörs här av den klass du ska skriva: ett grafiskt användargränssnitt som använder komponenter i Swing-paketet.

Det är inte alltid helt lätt att dra skarpa gränser mellan de tre delsystemen, och exakt hur mönstret Model-View-Controller tillämpas skiljer från applikation till applikation. (Hör t.ex. listmodell-klassen till modellen eller kontrollen?) Indelningen i modell, vy och kontroll är ändå generell och användbar, och tillämpas ofta när man konstruerar applikationer med grafiska användargränssnitt. Swing-paketet, liksom de flesta andra ramverk av samma slag, är konstruerat utifrån detta synsätt.

Datorarbete

- D1. Öppna din klass `GeneralWordCounter` från laboration 1. Vi behöver utöka klassen med en metod för att få tillgång till en lista med ord-antal-par. Metoden finns i den klassen eftersom vi kommer att behöva det `Map`-attribut du införde där i laboration 1.

Lägg till följande metod i klassen `GeneralWordCounter`:

```
public List<_____> getWordList() {
    return _____;
}
```

Den första streckade luckan ska ersättas med ett lämpligt typargument. För den andra luckan behövs bl.a. mängden av ord-antal-par. Mängden och listan har samma typargument.

Tips: förberedelseuppgifterna ger dig ledtrådar till luckornas innehåll.

- D2. Vi ska nu påbörja en klass som ska fylla rollen av Controller i Model-View-Controller-mönstret. Skapa en klass med namnet `BookReaderController`. Klassen ska ha följande struktur:

```
public class BookReaderController {

    public BookReaderController(GeneralWordCounter counter) {
        SwingUtilities.invokeLater(() -> createWindow(counter, "BookReader", 100, 300));
    }

    private void createWindow(GeneralWordCounter counter, String title,
                              int width, int height) {
        JFrame frame = new JFrame(title);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container pane = frame.getContentPane();
    }
}
```

```
// pane är en behållarkomponent till vilken de övriga komponenterna
// (listvy, knappar etc.) ska läggas till.

frame.pack();
frame.setVisible(true);
    }
}
```

- D3. Skapa en klass, `BookReaderApplication` som ska innehålla `main`-metoden. Inför satser i `main` som skapar ett `GeneralWordCounter`-objekt och använder det för att räkna ord i *Nils Holgerssons underbara resä*, så som gjordes i laboration 1. Vi använder alltså samma text och samma undantagsord som där. (Det går bra att kopiera satserna för inläsningen från programmet `Holgersson` i den laborationen.)

Skapa också `BookReaderController`-objektet.

Provkör programmet och kontrollera att ett tomt fönster visas.

- D4. Nästa steg är att visa orden i fönstren genom att skapa listvyn (`JList`) och knyta den till en listmodell. Det finns en färdigskriven klass i projektet, `SortedListModel`, som ska användas listmodell. Den lista som ska skickas med till konstruktorn är listan med ord-antal-par som hämtas från `GeneralWordCounter`.

För att listan ska vara skrollbar behövs ytterligare en komponent av typen `JScrollPane`. Denna ska innehålla listvyn och läggas till i fönstret.

Provkör programmet. Du ska nu få upp ett fönster med orden och deras antal (i någon ordning). Prova att scrolla och ändra storleken på fönstret.

- D5. Vi vill nu även få plats med en rad i fönstrets nederkant, där knapparna och textfältet kan få plats. För detta ändamål ska vi använda ett `JPanel`-objekt:

- Skapa ett `JPanel`-objekt.
- Skapa två knappar med etiketterna "Alphabetic" respektive "Frequency" (eller något liknande).
- Lägg till knapparna i din `JPanel`.
- Lägg din `JPanel` i fönstrets nederkant:

Provkör programmet. Kontrollera att fönstret innehåller ordlistan och två knappar. Prova gärna att trycka på knapparna.

- D6. Nu ska vi få något att hända när man trycker på knapparna. Lägg till en rad i ditt program så att en enkel textsträng skrivs ut när man trycker på en av knapparna. Använd ett lambdauttryck med en `System.out.println`-sats.

Kör programmet och kontrollera att du får det förväntade resultatet när du trycker på knappen.

Tips: Lambdauttrycket ersätter en metod. Därför måste det ha lika många parametrar som metoden ifråga, även om parametrarna inte används i uttrycket.

- D7. Ändra ditt program så att ordlistan sorteras alfabetiskt när man trycker på "Alphabetic", samt på antal förekomster när man trycker på "Frequency".

Ändringar i listan i `SortedListModel` förmedlas till listvyn. Kör programmet och verifiera att listans ordning förändras på rätt sätt när du trycker på knapparna.

- D8. Lägg nu till två element i vyn, ett textfält och en knapp, för att göra det möjligt att söka upp ett ord i listan. Om det inskrivna ordet finns i listan markeras det som valt och listan

scrollas listan så att ordet blir synligt (metoden `ensureIndexIsVisible` i `JList`), annars görs ingenting.

Tips: Det är praktiskt att använda ett lambdauttryck även här, eftersom du då har tillgång till alla lokala variabler i `start`. För att söka upp rätt rad i listan behöver du säkert använda mer än en Java-sats. Ett lambdauttryck med flera Java-satser kan kapslas in i ett block, så här:

```
obj.someMethod(e -> {  
    ... ;  
    ... ;  
    ... ;  
});
```

- D9. Lös tre uppgifter ur avsnittet "valbara uppgifter" nedan. Välj själv tre (eller fler) uppgifter som du tycker verkar intressanta. Uppgifterna är oberoende och kan göras i valfri ordning. Uppgifterna bygger (liksom tidigare deluppgifter) på att du själv söker i dokumentationen på nätet.
- D10. Fundera igenom följande, och diskutera med din handledare.
- Swing (liksom många motsvarigheter, som Android och JavaFX) bygger på en princip som kallas för *Hollywoodprincipen*: "Don't call us – we'll call you!"
Kan du se i ditt program vad som menas med detta?
 - Varför passar lambdauttryck särskilt bra ihop med denna princip?
 - När kontrolldelen av ditt program ändrar ordningen i ordlistan uppdateras användargränssnittet automatiskt. Kan du förstå något om hur det hänger ihop?

Valbara uppgifter

- V1. I ditt användargränssnitt kan man skriva in ett ord att söka efter. Om användaren råkar inleda eller avsluta ordet med ett eller fler mellanslag fungerar inte sökningen. Om användaren på samma sätt råkar skriva in versaler fungerar inte heller sökningen. Ändra programmet så att sökningen fungerar, även om det inmatade ordet börjar/slutar på mellanslag eller innehåller versaler. Du har nytta av ett par lämpliga metoder i klassen `String`. ✓
- V2. När man söker efter ett ord som inte finns i boken vore det bra med en ruta som meddelar användaren detta. Använd klassen `JOptionPane` för att visa en sådan ruta. ✓
- V3. Gör så att knappen "Find" aktiveras (trycks) automatiskt när man trycker Return. ✓
- V4. För de två sorteringsknapparna passar det bra att använda s.k. radioknappar. En sådan markeras när den är intryckt, och bara en knapp i samma grupp kan vara intryckt i taget. Läs om klassen `JRadioButton` och använd den för sorteringsknapparna. ✓
- V5. Programmet vore mer användbart om man kunde välja vilken textfil som ska analyseras. Låt användaren välja en fil att analysera. Använd klassen `JFileChooser`.

Laboration 4 – rekursion

Mål: Att ge träning i att skriva program med rekursiva algoritmer.

Förberedelser

F1. Läs igenom texten under rubriken "Bakgrund".

F2. Antag att följande anrop görs: `fractalLine(4, 810, 0);`

a) Hur många gånger nås basfallet, dvs. hur många linjer ritas?

Svar:

b) Hur långa är linjerna?

Svar:

F3. Lös uppgift D1, D2, D3 och D4.

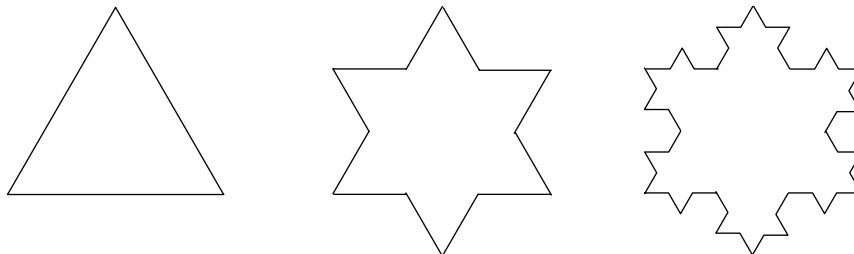
F4. Läs igenom de övriga uppgifterna.

Bakgrund

Uppgiften i denna laboration är att skriva ett program som ritar fraktala figurer. Fraktal, en term som myntades av Mandelbrot 1975, är benämningen på bilder som i motsats till t.ex rätta linjer, cirklar och trianglar är starkt sönderbrutna. De är uppbyggda av olika element med samma struktur.

Det finns många exempel på fraktaler i naturen såsom berglandskap, kustlinjer och virvelbilning i vattenfall. Inom matematiken använder man fraktaler för att beskriva sådana verkliga fenomen.

Studera fig 1 som visar fraktalen Kochs snöflinga (uppkallad efter den svenska matematikern Helge von Koch).



Figur 1: Kochs fraktal av ordning 0, 1 och 2.

Varje ny figur har åstadkommit genom att varje linje ersatts med en figur bestående av fyra nya linjer.

För att rita fraktalen Kochs snöflinga utgår man från en liksidig triangel. För att få en figur av ordning 1 ersätter man var och en av de tre linjerna med fyra nya linjer enligt fig 2. För att få en figur av ordning 2 ersätts varje linje i figuren av ordning 1 med fyra nya linjer osv.



Figur 2: En linje ersätts med fyra nya linjer.

En linje med längden `length` och riktningen `alpha` (vinkeln mellan linjen och x-axeln) ersätts alltså med fyra nya linjer som har följande längd och riktning:

- length/3, alpha
- length/3, alpha - 60°
- length/3, alpha + 60°
- length/3, alpha

Följande metod och tillhörande rekursiva hjälpmetod (i pseudokod) ritar Kochs snöflinga av en godtycklig ordning:

```
public void draw(int order, double length) {
    fractalLine(order, length, 0);
    fractalLine(order, length, 120);
    fractalLine(order, length, 240);
}

private void fractalLine(int order, double length, double alpha) {
    if (order == 0) {
        "rita en linje med längden length och riktningen alpha"
    } else {
        fractalLine(order-1, length/3, alpha);
        fractalLine(order-1, length/3, alpha-60);
        fractalLine(order-1, length/3, alpha+60);
        fractalLine(order-1, length/3, alpha);
    }
}
```

För att kunna rita olika fraktaler och fraktaler av olika ordning (grad av sönderbrytning) under laborationen finns det ett grafiskt användargränssnitt. Avsikten är att man som användare av det färdiga programmet ska kunna välja vilken fraktal man vill se ur en meny och kunna påverka fraktalens ordning genom att klicka på knappar. Användargränssnittet är i stora delar färdigt. Dock finns det bara en enda typ av fraktal att välja i menyn. Under laborationen kommer gränssnittet att behöva kompletteras så att man kan välja ytterligare en fraktaltyp.

D1. I projektet för laborationen finns tre paket: `fractal`, `koch` och `mountain`. I paketet `fractal` finns klasser för det grafiska användargränssnittet. Där finns bland annat en abstrakt klass `Fractal` som ska vara superklass till de egna "fraktalklasser" du skapar. Vidare finns klassen `TurtleGraphics` med metoder för att rita linjer i användargränssnittets fönster. De övriga klasserna i paketet beskriver användargränssnittets fönster med dess meny och knappar.

Huvudprogrammet finns i klassen `FractalApplication`. Kör detta program. Då öppnas ett fönster på skärmen:

- Fönstret har en meny med namnet *Fraktaler* och texten "Kochs triangel ordning 0" syns på fönstret.
- Om du öppnar menyn så syns det ett val: *Kochs triangel*. Om du väljer detta alternativ ur menyn så händer det ingenting. Det beror på att programmet vid detta val försöker rita Kochs fraktal av ordning 0, men denna metod gör ingenting förrän du själv kompletterat koden (uppgift D2).
- Fönstret har också knappar för att öka resp. minska den valda fraktalens ordning och rita den på nytt. Klicka på knappen pil uppåt. Texten i fönstret ändras då till "Kochs triangel ordning 1" men fortfarande ser man ingen fraktal av de skäl som nämnts ovan.

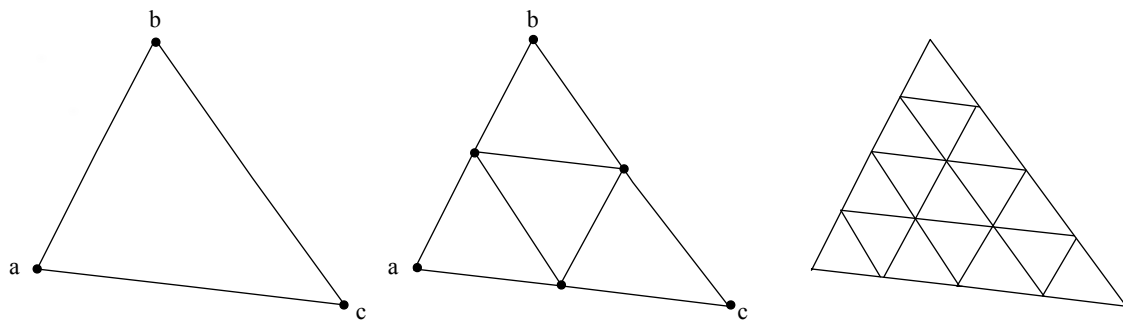
Om du inte ser några knappar längst ned i fönstret så justera fönstrets höjd (attributet `fractalHeight` i klassen `FractalApplication`).

- D2. I paketet `koch` finns en påbörjad klass `Koch` med metoder för att rita Kochs snöflinga. Fyll i de rader som saknas i metoden `fractalLine`.

Observera att koden för att rita Kochs snöflinga här i texten är pseudokod och att du i den riktiga koden även behöver ha med ett objekt av klassen `TurtleGraphics` som parameter för att rita linjer.

Kör huvudprogrammet. Nu ska du se Kochs fraktal av ordning 0 på fönstret då programmet startar och du ska kunna se samma fraktal av högre ordningar genom att använda knapparna.

- D3. I denna uppgift ska du lägga till ännu en fraktal till ditt tidigare program. Denna fraktal ska åskådliggöra ett bergsmassiv. En figur av ordning 0 utgörs av en triangel (gärna något sned). För att få nästa ordning ersätts varje triangel av fyra nya trianglar enligt fig 3.



Figur 3: Bergfraktal av ordning 0, 1 och 2.

I paketet `mountain` ska du lägga till en klass (liknande `Koch` i paketet `koch`) med metoder för att rita bergsfraktalen. Lämpliga parametrar till konstruktorn kan vara de tre startpunkterna. Till din hjälp finns den färdiga klassen `Point`, som beskriver en punkt.

OBS! Bergsfraktalen ritas på liknande sätt som Kochs snöflinga, men det finns ett par viktiga skillnader. Kochs snöflinga byggs upp av tre linjer. I varje rekursiv nivå ersätts en linje av fyra nya. En linje har en längd och en riktning. Bergsfraktalen består av en triangel. I varje rekursiv nivå ersätts en triangel av fyra nya trianglar. En triangel beskrivs av tre punkter.

För att din nya fraktal ska synas i användargränssnittets meny och kunna ritas upp behöver du bara ändra i `main`-metoden i klassen `FractalApplication`. Öka vektorn `fractals` storlek och lägg in ett objekt av din nya fraktalklass i den. När du provkör programmet kommer du att se att det i menyn dyker upp ett alternativ till med det namn som metoden `getTitle()` i den nya fraktalklassen returnerar. Välj detta alternativ för att testa ritning av bergsmassiv. Tips! Lägg in det nya fraktalobjektet först i vektorn så kommer det att visas när programmet startar. Du kommer nog att provköra det en hel del gånger.

- D4. Fraktalen i föregående uppgift blir för regelbunden för att likna ett bergsmassiv. Inför uppdelningen av en triangel i fyra nya, mindre trianglar ska därför mittpunkten förskjutas i y-led. Se fig. 4.

Förskjutningens storlek bestäms av funktionen `randFunc` som ger ett slumptal enligt en viss fördelning med avvikelsen `dev`:

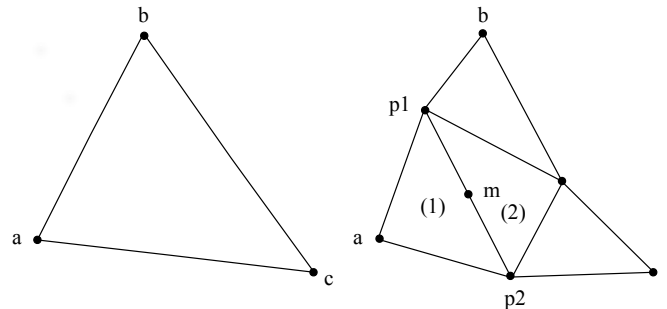
```
public static double randFunc(double dev) {
    double t = dev * Math.sqrt(-2 * Math.log(Math.random()));
    if (Math.random() < 0.5) {
        t = -t;
    }
    return t;
}
```

För varje nivå ska parametern `dev` till `randFunc` halveras. Om man glömmer att halvera denna parameter blir figuren för taggig. Låt gärna startvärdet på `dev` vara parameter till `Mountains` konstruktor.

De förskjutna mittpunkterna kommer att tillsammans med triangelns ursprungliga hörn att utgöra hörn i de fyra nya triangelarna.

Metoden `randFunc` är färdig att använda och finns i klassen `RandomUtilities`.

Berget kommer nu att ha lite mer oregelbundna former och se mer naturligt ut. Men det kommer att finnas vita fält här och var i figuren (löses i nästa deluppgift).



Figur 4: Bergfraktal av ordning 0 och 1. Mittpunkterna förskjuts - y-led innan en triangel delas upp i fyra nya trianglar.

- D5. Av den högra figuren i fig. 4 ser vi att en speciell svårighet uppstår genom att triangelarna har vissa sidor gemensamma. När triangel (1) ska delas in i fyra mindre trianglar så ska mittpunkten `m` förskjutas. När senare triangel (2) ska delas in får inte `m` förskjutas en gång till. Så här kan man göra för att klara av denna svårighet:

Implementera först en klass `Side` som håller reda på en triangelns ändpunkter.

Skapa i klassen `Mountain` en map av typen `HashMap<Side, Point>` där `Side`-objekt kan lagras tillsammans med sin beräknade mittpunkt.

När en mittpunkt ska beräknas söker man först i mappen efter en sida med ändpunkterna `p1`, `p2`. Om en sådan sida finns använder man den redan beräknade mittpunkten. I annat fall beräknar man mittpunkten på samma sätt som tidigare och lagrar sidan med `p1`, `p2` i mappen tillsammans med den beräknade mittpunkten.

Eftersom en sida bara används högst två gånger (första gången då mittpunkten beräknas och andra gången då den hittas i mappen) kan man ta bort sid-mittpunktsparet från mappen när man använt den. Sökningen blir snabbare då.

I mappen används klassen `Side` som nyckel. För att det ska fungera måste man där skugga metoderna `equals` och `hashCode` i klassen `Side`. Dessa två metoder används för att hitta nyckeln i mappen. (Hur mappen är implementerad och hur dessa två metoder används kommer att behandlas senare i kursen. När du tidigare har använt klassen `HashMap` har någon av Javas standardklasser används som nyckel. I dessa klasser är redan `equals` och `hashCode` skuggade.)

Metoden `hashCode` kan se ut så här:

```
@Override
public int hashCode() {
    return p1.hashCode() + p2.hashCode();
}
```

Inuti metoden `equals` är det sidornas ändpunkter som ska jämföras. Tänk på att man inte säkert vet ordningen på sidans ändpunkter.

Tips! `@Override` är ett direktiv till kompilatorn att kontrollera så att man skuggar en metod som verkligen finns i en superklass. Lägg till det före metoden `equals` så försäkras du dig om att metodnamnet är rättstavat och att parametern har rätt typ.

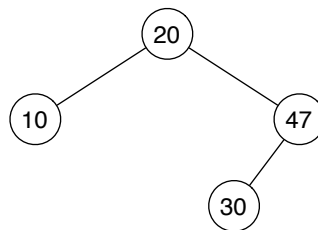
Laboration 5 – binära sökträd

Mål: Att ge träning i att implementera rekursiva algoritmer, speciellt för träd.

Förberedelser

F1. Betrakta trädet i fig. 1 och besvara följande frågor:

- a) Vad skrivs ut om trädets element skrivs ut i inorder? Svar:
- b) Vilken höjd har trädet? Svar:
- c) Vilken höjd får trädet om 42 sätts in? Svar:



Figur 1: Binärt sökträd.

F2. I uppgift D2 kommer du att skriva en rekursiv metod för att beräkna trädets höjd.

- a) Den rekursiva algoritmen kan uttryckas i pseudokod. Komplettera algoritmen:

Om trädet är tomt är höjden

annars är höjden 1 +

- b) Sök i dokumentationen av klassen Math på nätet. Med vilken metod kan man få det största värdet av två heltal? Svar:

F3. Repetera algoritmen för binärsökning från föreläsningen om rekursion. I uppgift D7 kommer du i att kunna hämta inspiration från den algoritmen. Den rekursiva metoden har parametrar `first` och `last` för att hålla reda på i vilken del av vektorn man söker. Hur beräknar man mitten i intervallet `[first, last]`?

`int mid =`

F4. Läs igenom texten under rubriken "Bakgrund".

F5. Lös uppgifterna D1- D5.

F6. Läs igenom övriga uppgifter.

Bakgrund

Under denna laboration kommer delar av en klass för hantering av binära sökträd att implementeras.

I projektet för laborationen finns ett paket `bst` med en fil `BinarySearchTree.java`. Här finns en påbörjad implementering för hantering av binära sökträd.

Noderna i trädet representeras av en statisk nästlad klass `BinaryNode`.

Datorarbete

- D1. Börja med att implementera konstruktörerna i klassen `BinarySearchTree`.

Tanken med de två konstruktörerna är att den som skapar träd-objekt ska kunna välja hur elementen i trädet ska jämföras. Väljer man att använda konstruktorn utan parameter måste klassen som ersätter typparametern `E` implementera `Comparable`. Väljer man den andra konstruktorn ska man istället skicka med ett lambdauttryck.

- D2. Implementera metoden

```
public int height();
```

som beräknar trädets höjd med rekursiv teknik.

Tips! Det är lämpligt att skriva en rekursiv privat metod som anropas i den publika metoden.

- D3. I denna uppgift ska en metod med följande rubrik implementeras i klassen `BinarySearchTree`:

```
public boolean add(E x);
```

Metoden ska lägga in elementet `x` i trädet om det inte redan finns. Metoden ska returnera `true` om insättningen kunde utföras, annars `false`. Implementeringen ska vara rekursiv. Tänk noga efter hur du ska jämföra två element för att upptäcka dubbletter resp. för att se om det nya elementet ska sättas in till vänster eller till höger.

Implementera också metoderna `size` och `clear`. Tips! Dessa två metoder blir korta (en respektive två rader var bör räcka).

- D4. Implementera i klassen `BinarySearchTree` metoden

```
public void printTree();
```

som skriver ut nodernas innehåll i inorder.

- D5. Testa metoderna `height`, `add`, `size` och `clear` genom att använda `JUnit`. Glöm inte att testa att din `add`-metod fungerar om man försöker sätta in dubbletter och att `add` alltid returnerar rätt resultat. Glöm inte heller att testa metoderna `height` och `size` i ett tomt träd.

Bägge konstruktörerna ska testas. Använd inte samma typ av element i alla testerna.

- D6. I klassen `BSTVisualizer` finns metoden `void drawTree(BinarySearchTree<?> bst)` som ritar ett binärt träd i ett fönster. (Inuti klassen `BSTVisualizer` används metoden `height` från uppgift D1 samt klasser i paketet `drawing`).

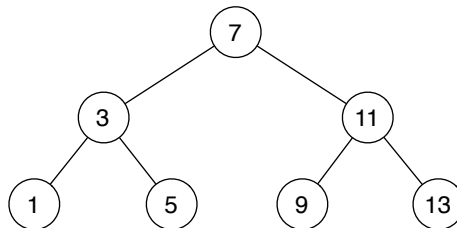
Skriv en `main`-metod i klassen `BinarySearchTree` som ritar trädet (anropa `drawTree`) samt skriver ut det (anropa `printTree`). Prova att skapa några träd av olika form, t. ex. ett skevt träd resp. träd med mer optimal form.

- D7. Ett träd kan bli snett (obalanserat) när man gör många insättningar och borttagningar. Ett sätt att undvika detta är att balansera trädet i samband med varje insättning/borttagning enligt den metod som vi gått igenom på föreläsningarna (s.k. AVL-träd). Ett annat sätt kan vara att "bygga om" trädet då och då när det blivit alltför snett förutsatt att detta inte sker alltför ofta. En algoritm som bygger om trädet till ett träd som har maximalt

antal noder på alla nivåer utom den som ligger längst bort från roten ska implementeras i denna uppgift.

Om man placerar alla element från trädet i växande ordning i en lista av typen `ArrayList` är det sedan enkelt att bygga ett träd där antalet noder i vänster respektive höger underträd aldrig skiljer sig med mer än ett och som därför är balanserat. Algoritmen är följande: Skapa en nod som innehåller mittelementet i listan. Bygg (rekursivt) ett träd som innehåller elementen till vänster om mittelementet och ett träd som innehåller elementen till höger om mittelementet. Låt dessa båda träd bli vänster respektive höger barn till roten.

Antag t.ex. att listan innehåller heltalen 1, 3, 5, 7, 9, 11, 13. Trädet som byggs får då det utseende som visas i fig. 2.



Figur 2: Binärt sökträd med nycklar 1, 3, 5, 7, 9, 11, 13 byggt enligt algoritmen i texten.

En metod med följande rubrik ska implementeras i klassen `BinarySearchTree`:

```

/**
 * Builds a balanced tree from the elements in the tree.
 */
public void rebuild();

```

Metoden ska implementeras så att den går igenom trädet i inorder och bildar en lista med innehållet i växande ordning. Sedan ska trädet byggas enligt algoritmen ovan. Följande rekursiva hjälpmetoder ska implementeras och användas inuti `rebuild`:

```

/*
 * Adds all elements from the tree rooted at n in inorder to the list sorted.
 */
private void toArray(BinaryNode<E> n, List<E> sorted)

/*
 * Builds a complete tree from the elements from position first to
 * last in the list sorted.
 * Elements in the list are assumed to be in ascending order.
 * Returns the root of tree.
 */
private BinaryNode<E> buildTree(List<E> sorted, int first, int last);

```

Observera att noder ska skapas och ett nytt träd ska byggas upp från scratch inuti `buildTree`. Metoden `add` ska inte användas.

Testa genom att skriva en main-metod som bygger ett snett (men inte helt snett) träd genom successiva `add`-anrop och som sedan anropar `rebuild`. Låt sedan main-metoden rita trädet och kontrollera att det blivit ett balanserat träd.

D8. Fundera igenom följande, och diskutera med din handledare.

- I vissa av de uppgifter du löst (t ex `height` och `add`) finns det en publik metod som anropar motsvarande rekursiva metod. Varför behövs bägge metoderna?
- I samband med ombyggnaden av trädet används en lista av typen `ArrayList` för att mellanlagra elementen. Skulle man lika gärna kunna använda en `LinkedList`?

Laboration 6 – hashtabell

Mål: Att ge förståelse för den abstrakta datatypen Map och datastrukturen hashtabell.

Förberedelser

Besvara frågorna med hjälp av dokumentationen för Javas interface Map<K, V>.

F1. Betrakta följande kodavsnitt:

```
public class TestMap {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        System.out.println(map.get("A") + " " + map.size());
        map.put("A", 43);
        System.out.println(map.get("A") + " " + map.size());
        map.put("A", 42);
        System.out.println(map.get("A") + " " + map.size());
    }
}
```

I map lagras nyckel-värde-par.

a) Vad har nycklarna för typ?

Svar:

b) Vad har värdena för typ?

Svar:

c) Vad skrivs ut?

Svar:

F2. Lös uppgifterna D1–D6.

F3. Läs igenom övriga uppgifter.

Datorarbete

I denna uppgift ska du göra en implementering av en *öppen hashtabell* ("separate chaining"). Listorna ska konstrueras från grunden med enkellänkade listor.

Din klass ska implementera gränssnittet `map.Map` som innehåller en delmängd av de metoder som finns i gränssnittet `java.util.Map`. Dokumentationen för `java.util.Map` på nätet beskriver metoderna.

```
package map;

interface Map<K,V> {
    static interface Entry<K,V> {
        K getKey();
        V getValue();
        V setValue(V value);
    }
    V get(Object arg0);
    boolean isEmpty();
    V put(K arg0, V arg1);
    V remove(Object arg0);
    int size();
}
```


Den nästlade klass som ska implementera `Map.Entry<K,V>` ska ha ett attribut som är en länk till nästa element i listan.

```
private static class Entry<K,V> implements Map.Entry<K,V> {
    private K key;
    private V value;
    private Entry<K,V> next;
    ...
}
```

Dessutom ska du skriva en metod som skriver ut hashtabellens innehåll.

Det kan vara lämpligt att implementera en sak i taget och testa när så är möjligt. I paketet `test` finns en färdig testklass `TestSimpleHashMap`.

D1. I projektet för laborationen finns ett paket `map`. Skapa i detta paket klassen

```
public class SimpleHashMap<K,V> implements Map<K,V>
```

Så här kan man göra i Eclipse:

- Markera paketet med höger musknapp och välj `New->Class`.
- Fyll i namnet på klassen (`SimpleHashMap<K,V>`) i fältet `Name`.
- Klicka på `Add-knappen` vid textfältet `Interfaces`. Då öppnas ett nytt dialogfönster. Fyll i interfacets namn (`map.Map`). Under "matching items" kommer medan du skriver förslag på interface som matchar ditt namn. Markera här interfacet (`map.Map`) och klicka på `OK`. Klicka på `Finish` i det första dialogfönstret (`New Java Class`).

Den nya klassen öppnas normalt i editorn och om du inte har ändrat inställningarna i Eclipse så bör nu din klass `SimpleHashMap` innehålla "stubbar" för de metoder som föreskrivs av interfacet (`map.Map`). Om det inte innehåller stubbar kan du skapa dem genom att på menyn `Source` välja alternativet `Override/Implement methods`.

Nu återstår det att lägga in den nästlade klassen

```
private static class Entry<K,V> implements Map.Entry<K,V>
```

Om du vill använda dialogen även för detta ska du

- Markera klassen `SimpleHashMap` och välja `New->Class` igen.
- I dialogen föreslås nu `SimpleHashMap` som `Enclosing type` (eftersom vi markerade den klassen).
- Kryssa i rutan vid `Enclosing type`.
- Fyll i namnet på den nästlade klassen (`Entry<K,V>`) i fältet `Name`.
- Markera att klassen ska vara statisk genom att kryssa i rutan `static`.
- Även den nästlade klassen ska implementera ett interface. I princip skulle samma teknik som beskrivits ovan för den omgivande klassen nu kunna användas. Det verkar dock som om Eclipse har vissa svårigheter med typparametrar för interface som implementeras av inre klasser. Därför föreslås att detta tillägg görs manuellt. Klicka alltså på `Finish`.

Skriv i filen in att den nästlade klassen implementerar interfacet `Map.Entry<K,V>`. Välj därefter från menyn `Source` alternativet `Override/Implement methods`. Du får då stubbar för de metoder som interfacet föreskriver.

- D2. Implementera konstruktorn och metoderna i den nästlade klassen `Entry`. Skugga också metoden `toString()` som ska returnera nyckel och värde med "=" emellan.
- D3. Bland attributen i `SimpleHashMap` ska det finnas en vektor (`table`) med `Entry`-element. Lägg in detta och andra lämpliga attribut och implementera följande två konstruktorer (som skapar vektorn):

```
/** Constructs an empty hashmap with the default initial capacity (16)
    and the default load factor (0.75). */
SimpleHashMap();

/** Constructs an empty hashmap with the specified initial capacity
    and the default load factor (0.75). */
SimpleHashMap(int capacity);
```

Man får inte använda en parametriserad typ när man skapar en vektor i Java. Gör därför så här:

```
(Entry<K,V>[]) new Entry[capacity];
```

- D4. För att kunna kontrollera att informationen lagras på rätt sätt ska du i klassen `SimpleHashMap` skriva en metod `String show()` som ger en sträng med innehållet på varje position i tabellen på egen rad.

```
0      key=value key=value etc.
1      key=value key=value etc.
...
```

- D5. Implementera `size()` och `isEmpty()`.
- D6. För att enkelt kunna implementera de övriga metoderna är det lämpligt att ha två privata hjälpmetoder:

```
private int index(K key)
private Entry<K,V> find(int index, K key)
```

`index(key)` ska returnera det index som ska användas för nyckeln `key`.

`find(index, key)` ska returnera det `Entry`-par som har nyckeln `key` i listan som finns på position `index` i tabellen. Om det inte finns något sådant ska metoden returnera `null`.

- D7. Implementera `put(K key, V value)`. Om det fanns ett gammalt värde ska detta returneras. Annars returneras `null`. Tänk på att fyllnadsgraden inte ska överstiga 0.75 och öka kapaciteten om så är fallet. Det är lämpligt att skriva en privat metod `rehash` för detta.
- D8. Implementera `get(Object object)`. Argumentet måste omvandlas till typen `K`. Om nyckeln inte finns returneras `null`.
- D9. Nu går det bra att testa. Öppna klassen `TestSimpleHashMap` och ta bort kommentarstecknen på de rader där `SimpleHashMap`-objekten skapas i metoden `setUp`. (De är bortkommenterade för att inte orsaka kompileringsfel innan klassen `SimpleHashMap` existerar.) En del tester i `TestSimpleHashMap` använder metoden `remove` som ej är implementerad ännu. Kör testen ändå och bortse från de fel som avser ännu inte implementerade metoder.
- Tips! Om det inte fungerar som det ska så kommentera bort koden med anropet av

rehash. Om programmet då går igenom testerna har du isolerat felet till koden för rehashingen.

- D10. Testerna i JUnit testar en hel del, men inte allt. Det är svårt att skriva ett fullständigt test av hashtabellen utan att förutsätta för mycket om hur den är implementerad. Skriv därför en `main`-metod där du skapar ett `SimpleHashMap`-objekt, och lägger in slumpmässigt valda element samt skriver ut innehållet med hjälp av metoden `show`. Om både nyckel och värde i varje par är samma `Integer`-värde och kapaciteten 10 blir det lätt att kontrollera resultatet. Använd både positiva och negativa tal. Öka antal element och kontrollera att ökningen av kapacitet (rehashing) fungerar som den ska. Kontrollera att listorna inte blir orimligt långa.
- D11. Implementera `remove(Object key)`. När man ska implementera `remove` bestämmer man först i vilken lista som nyckeln borde finnas. Följande fall måste hanteras:
1. Listan är `null`.
 2. `key` finns i det första elementet i listan.
 3. `key` finns senare i listan.
 4. `key` finns inte i listan.

Testa!

- D12. Fundera igenom följande, och diskutera med din handledare:
- Hashtabellen består av en vektor med listor innehållande element (i det här fallet nyckel-värde-par) som kolliderat. Vad är det som orsakar kollisioner?
 - Hur används metoderna `hashCode` och `equals` vid insättning i en hashtabell?
- D13. På laborationen har du implementerat en `map` med hjälp av en hashtabell. Finns det någon annan datastruktur som också kan användas för att effektivt implementera en `map`?