Använda mängder och mappar

Tema: Användning av Javas klasser för mängder och mappar.

U 1. I `java.util` finns klasserna `TreeSet` och `TreeMap` som implementeras med hjälp av binära sökträd. I dokumentationen till dessa klasser talas det om att element jämförs enligt deras ”natural order”. Vad menas med det?

U 2. Antag att vi ska använda en av klasserna i Java Collections Framework för att hålla reda på böcker. En bok beskrivs av följande klass:

```
public class Book  {
    private String isbn;
    private String title;
    private String author;

    public Book(String isbn, String title, String author) {
        this.isbn = isbn;
        this.title = title;
        this.author = author;
    }
}
```

a) Vi vill snabbt snabbt kunna hitta en bok med hjälp av dess ISBN-nummer och väljer klassen `HashMap`. Skriv programrader för att skapa ett `HashMap`-objekt att lagra böcker i och för att sätta in ett par (påhittade) böcker i bokregistret.
   Gör också de eventuella förändringar som behövs i klassen `Book`.

b) I ett annat sammanhang vill man lagra böcker i ett `HashSet`-objekt. Skriv programrader som skapar ett `HashSet`-objekt att lagra böcker och sätter in ett par (påhittade) böcker i mängden.
   Gör också de eventuella förändringar som behövs i klassen `Book`.

U 3. Beskriv kort vad som händer vid sökning efter en bok i hashtabellen i uppgift U 2b, dvs. vad som görs i metoden `contains` i klassen `HashSet`. Redogör i beskrivningen för hur metoderna `equals` och `hashCode` används vid sökningen.

U 4. Vi ska fortsätta arbeta med klasserna från uppgift U 2.

a) Antag att vi istället för en hashtabell vill använda ett binärt sökträd. Vi byter därför ut `HashMap` mot `TreeMap` i uppgift U 2 a. Behövs det några ändringar i klassen `Book`?

b) Vilken klass är att föredra, `HashMap` eller `TreeMap`, om antal böcker är stort och det är viktigt att sökningarna går snabbt?

Tillämpningar

U 5. Antag att du har fått i uppgift att skriva ett program som läser en textfil, som är indelad i sidor, och ska producera en s.k. korsreferenslista över innehållet. En korsreferenslista är en lista över alla förekommande ord i alfabetisk ordning och för varje ord en uppräkning av de sidor där ordet förekommer. Exempel:

```
alla    1, 2, 5, 8, 19
att     3, 5, 7, 9
bara    2, 6, 7, 9
bo      5, 19
del     6
```

Sidnumren för varje ord ska vara ordnade i stigande följd. Även om samma ord finns flera gånger på samma sida ska sidnumret bara förekomma en gång.

Beskriv med ord hur man kan lösa problemet. I din beskrivning ska det framgå vilka algoritmer och datastrukturer som ska användas för att producera korsreferenslistan. Ange också om du kan använda någon eller några av de klasser i Java Collections Framework du träffat på under kursen.

Du kan anta att det finns en operation som hämtar nästa ord tillsammans med aktuellt sidnummer från filen.

U 6. En grupp av ord är anagram om de kan bildas av varandra genom att ändra ordningen på de ingående bokstäverna. T.ex. är de svenska orden ”avig” och ”viga” anagram till varandra. Ett bra sätt att hålla reda på anagram är att använda en map där värdena består av mängder med ord som är anagram till varandra. Som nycklar i mappen använder man den sträng man får om man sorterar bokstäverna i alfabetisk ordning. Om vi t.ex. skulle sätta in orden ”avig”, och ”viga” i en sådan (från början tom) map så skulle följande inträffa:

- När ordet ”avig” ska sättas in bildar vi först en sträng genom att sortera bokstäverna. Vi får då ”agiv”. Denna nyckel finns inte i tabellen (som är tom). Den sätts då in med sitt tillhörande värde, mängden [”avig”].
- När ”viga” ska sättas in bildar vi den sorterade strängen som igen blir ”agiv”. Denna nyckel finns och då läggs det nya ordet till dess värde som nu blir mängden [”avig”, ”viga”].

a) Implementera klassen `Anagram` som håller reda på ord och dess anagram:

```
public class Anagram {
    private Map<String, Set<String>> anagrams;

    /** Skapar ett objekt som hanterar anagram. */
    public Anagram() {
        // Fyll i egen kod.
    }



    /** Lägger till order word. */
    public void add(String word) {
        // Fyll i egen kod.
    }

    /**
     * Returnerar en mängd med alla ord som är anagram till ett visst ord.
     * word ska inte ingå i den mängd som returneras. Om word inte har
     * några insatta anagram ska en tom mängd returneras.
     */

    public Set<String> getAnagramsOf(String word) {
        // Fyll i egen kod.
    }

    /**
     * Returnerar en sträng med bokstäverna i word sorterade i
     * bokstavsordning.
     */
    private String alphabetize(String word) {
        // Färdig att använda.
    }
```

```
}
```

b) Vilken tidskomplexitet får metoden `add` i medelfall? Du kan anta att orden innehåller rimligt antal bokstäver och att metoden `alphabetize` därför har konstant tidskomplexitet.