# Lösningsförslag

### Använda mängder och mappar

U 1. Det betyder helt enkelt att man förutsätter att elementen implementerar interfacet `Comparable` så att man kan jämföra dem genom att använda operationen `compareTo`.

U 2. a)
```
hMap<String, Book> map = new HashMap<String, Book>();
map.put ("9113013645", new Book("9113013645", "De två tornen", "Tolkien"));
map.put ("9113016482", new Book("9113016482", "Hobbiten", "Tolkien"));
```

Inga förändringar behövs i klassen `Book`. Nyckeln utgöra av ISBN-numret som har typen `String`. Metoderna `hashCode` och `equals` är omdefinierade i klassen `String`.

b)
```
Set<Book> set = new HashSet<Book>();
set.add(new Book("9113013645", "De två tornen", "Tolkien"));
set.add(new Book("9113016482", "Hobbiten", "Tolkien"));
```

Skugga följande metoder i klassen `Book`:

```
public boolean equals(Object rhs) {
    if (rhs instanceof Book) {
        return isbn.equals(((Book) rhs).isbn);
    } else {
        return false;
    }
}

public int hashCode() {
    return isbn.hashCode();
}
```

U 3. Metoden `hashCode` använd för att hitta rätt vektorelement, dvs rätt lista. Därefter söks denna lista igenom efter rätt element. Vid denna sökning används `equals` vid jämförelserna.

U 4. a) Nycklarna i en `TreeMap` måste vara av en typ som implementerar interfacet `Comparable`. Nyckeln `isbn` är av typen `String` och klassen `String` implementerar `Comparable`. Därför behövs det inga förändringar i klassen `Book`.
(Om vi istället byter ut `HashSet` mot `TreeSet` i uppgift U 2 b måste vi se till att klassen `Book` implementerar `Comparable`. Eller också kan man använda ett lambdauttryck eller ett Comparator-objekt.)

b) Det går snabbare ett söka i en hashtabell än i ett binärt sökträd. (I alla fall om hashfunktionen är vettigt vald och hashtabellen har en rimlig fyllnadsgrad.) Därför är `HashMap` att föredra om antal böcker är stort och det är viktigt att det går snabbt att söka.

U 5. Man kan använda en hashtabell med ordet som nyckel och listan med sidnummer som värde. För varje ord i filen söker man i hashtabellen. Om ordet inte finns sätter man in det tillsammans med sidnumret. Om ordet redan finns ska aktuellt sidnummer sättas in sidnummerlistan om det inte redan finns med där. För att undvika dubbletter räcker det att jämföra aktuellt sidnummer med sista sidnumret i listan. Det är lämpligt att det finns en referens till sista elementet i listan så att man snabbt når detta.

Innan man skriver ut orden och dess sidnummerlista måste orden sorteras.

Som hashtabell kan man använda klassen `HashMap` och för sidnummerlistorna klassen `LinkedList`.

Istället för att lagra orden i en hashtabell kan de lagras i en ett balanserat binärt sökträd (t.ex. `TreeMap`). För att skriva ut korsreferenslistan går man igenom trädet i inorder.

Istället för att använda en lista för sidnumren skulle man kunna använda en sorterad mängd (t.ex. `TreeSet`).

U 6. a)

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

    /** Skapar ett objekt som hanterar anagram. */
    public Anagram() {
        anagrams = new HashMap<String, Set<String>>();
    }

    /** Lägger till ordet word */
    public void add(String word) {
        String sorted = alphabetize(word);
        Set<String> anagramSet = anagrams.get(sorted);
        if (anagramSet == null) {
            anagramSet = new HashSet<String>();
            anagrams.put(sorted, anagramSet);
        }
        anagramSet.add(word);
    }

    /**
     * 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) {
        Set<String> s = anagrams.get(alphabetize(word));
        if (s != null) {
            s = new HashSet<String>(s); // make a copy
            s.remove(word); // remove word, if present
        } else {
            s = new HashSet<String>();
        }
        return s;
    }

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

b) Vi antar att mappen innehåller $n$ ord och att antal anagram till ett ord är litet Ett anrop av metoder på en ords mängd med anagram anser vi därför ta konstant tid.

I `add` anropas `get` och ev. `put` på mappen. Eftersom vi använder en `HashMap` (hashtabell) får dessa metoder konstant tidskomplexitet och den totala tidskomplexiteten blir alltså $O(1)$.

Hade vi istället använt en `TreeMap` kostar metoderna `get` och ev. `put` $O(\log n)$ och den totala tidskomplexiteten blir $O(\log n)$.