Reexam, DAT038/TDA417 (and DAT037/TDA416)

Data structures and algorithms (solution suggestions)

Thursday, 2020-04-30, 8:30–12:30, in Canvas

Examiner(s) Peter Ljunglöf and Nick Smallbone.

Allowed aids Course literature, other books, the internet.
You are not allowed to discuss the problems with anyone!

Submitting Submit your answers ***in one single PDF or Word or OpenOffice document***.
Number every answer! Use page breaks between answers!

Pen and paper Some questions are easier to answer using pen and paper, which is fine!
In that case, take a photo of your answer and paste into your answer document.
Make sure the photo is readable!

Assorted notes You may answer in English or Swedish.
Excessively complicated answers might be rejected.
Write legibly – we need to be able to read and understand your answer!

Exam review If you want to discuss the grading, please contact Peter or Nick via email.

There are **6 basic questions**, and **3 advanced questions**, and **two points per question**.
So, the highest possible mark is 18 in total (12 from the basic questions and 6 from the advanced questions). Here is what you need to do to get each grade:

- To pass the exam, you must get 8 out of 12 on the basic questions.
- To get a four, you must get 9 out of 12 on the basic questions,
  and also get 2 out of 6 on the advanced questions.
- To get a five, you must get 10 out of 12 on the basic questions,
  and also get 4 out of 6 on the advanced questions.

Grade	Total points	Basic points	Advanced
3	⩾ 8	⩾ 8	—
4	⩾ 11	⩾ 9	⩾ 2
5	⩾ 14	⩾ 10	⩾ 4

Good luck!

Detailed instructions

During the exam (8:30–12:30)

ID checking:

- At the beginning of the exam you will be invited to a "breakout room" in Zoom, where your ID will be checked.
- If you are kicked out of the Zoom meeting, then enter again as quickly as possible by using the same link as earlier. You will be ID checked again before being let in.

Do's and don'ts:

- You are allowed to use books or the internet.
- Don't communicate with anyone else during the exam, neither orally nor in any other format. This includes posting questions in chat forums, etc.
- Don't wear earbuds or earphones.
- Don't leave your seat.

Contacting the guard or the examiner:

- If you need to go to the toilet, inform the exam guard via the Zoom chat.
  - Write "**Toabesök**" / "**Toilet visit**" before you go to the toilet.
  - Write "**Toabesök SLUT**" / "**Toilet visit ENDED**" when you're back.
- Write "**Fråga till examinator**" / "**Question for the examiner**" if you have a question. You will be invited to a breakout Zoom room, but it might take some time.
- If you need to contact the exam guard, write "**Kontakt**" / "**Contact**" in the Zoom chat. You will be invited to a breakout Zoom room, but it might take some time.

If you finish earlier:

- You have to stay logged in to Zoom until you are permitted to leave.
- If you finish earlier, write "**Scannar lösningar**" / "**Scanning solutions**" in the chat before you scan your solutions and merge them into one single document.
- When your solution is submitted, write "**Lämnat in i Canvas**" / "**Submitted to Canvas**".
- When the exam guard has ticked you off and added ##**KLAR**## before your Zoom name, you can leave the Zoom meeting. *But not before that*!

When the exam finishes (12:30–12:50)

- The exam finishes at 12:30 sharp. After that you are not allowed to continue working.
- You have 20 more minutes to scan your answers and submit them to Canvas.
  You have to submit before 12:50, otherwise you automatically fail the exam.
- Submit your answers in Canvas as a single file upload to the examination assignment. Make sure that you:
  - Submit *one single document* in PDF, Word or OpenOffice/LibreOffice format. Scanned answers should be inserted as images in the document.
  - Name the document *"reexam-YYYYMMDD-NNNN.pdf"*, with *YMDN* replaced by your personnummer (and the correct suffix instead of *.pdf*, if you use Word or OpenOffice).
  - Start every answer on a new page in this document.
- After submitting, write "**Lämnat in i Canvas**" / "**Submitted to Canvas**" in the chat.
- If you have problems submitting to Canvas or creating a single document in the correct format, you may send your solutions in an email to Peter (peter.ljunglof@cse.gu.se). Use the subject "**Reexam solutions**", and write your name and personnummer in the email.
  - **You must still do this before 12:50!**

Basic question 1: Complexity

The following program takes two sets *A* and *B* and computes their union $A \cup B$:

```
function Set union(Set A, Set B):
    if number of elements in A > number of elements in B:
        for each x in B:
            A.add(x)
        return A
    else:
        for each x in A:
            B.add(x)
        return B
```

a) What is the worst-case time complexity of this program? Assume that *A* and *B* are balanced binary search trees (e.g. red-black trees), and that *A* contains *m* elements and *B* contains *n* elements. You should also assume that comparing two elements takes O(1) time, and that finding the number of elements in a set takes O(1) time.

   Give your answer in terms of *m* and *n*, and explain your reasoning. In your answer, you may use the functions *min(x,y)* and *max(x,y)*, which give the minimum and maximum of *x* and *y*.

b) The program uses an *if-then-else* to choose between two different algorithms for calculating set union. Explain why this is more efficient than using a simpler algorithm such as the following:

```
function Set union(Set A, Set B):
   for each x in B:
      A.add(x)
   return A
```

Your answer should contain the following:

a) The worst-case complexity, either in O-notation or as *order-of-growth*.
b) An explanation of why the program is more efficient than the simpler algorithm in part (b).

Solution suggestion

a) Suppose that m > n, so that the "then"-branch of the if-then-else runs.
Then the loop runs n times, inserting a value into A each time.
Since A is a balanced BST, each insertion should take O(log (size of A)) time.
The size of A is initially m, but grows throughout the loop to a maximum of m+n.
But since m > n we have m+n < 2m so the size of A is O(m).
Putting this together, we have a loop that runs n times, where the loop body takes O(log m) time. Therefore, when m > n, the total runtime is O(n log m).

If m <= n, then the "else"-branch runs instead.
A similar argument shows that this takes O(m log n) time.

Therefore, the complexity is **O(m log n) if m <= n**, and **O(n log m) if m > n**.

We can also write the complexity using the *min* and *max* functions as
O(min(m, n) log max(m, n)) or **O(min(m, n) log (m+n))** or **O(min(m log n, n log m))**.
All of these answers were accepted.

b) The algorithm in part b always takes O(n log m) time. This is worse than the algorithm in part a whenever n > m, that is, when B is bigger than A.

For example, if A has only 1 element, and B has a million elements, then the program in part a will do one BST insertion (taking time proportional to log 1000000), while the program in part b will do 1000000 insertions (taking a approximately million times as long).

Basic question 2: Sorting

Transform your own 12-digit personnummer into an array with 11 cells, by summing every adjacent pair of digits. For example, if your personnummer is 19711031–1234, you will get the list of numbers [1+9, 9+7, 7+1, 1+1, 1+0, 0+3, 3+1, 1+1, 1+2, 2+3, 3+4], and create the following array:

10	16	8	2	1	3	4	2	3	5	7

Write down the array that you have created. In the rest of the question, you will show part of what happens when the array is **quicksorted.**

First, partition your array using the first element as pivot, and show what the array looks like afterwards. Mark the two partitions, e.g., by using braces, or arrows, or different colours.
If you use colours it might look something like this, but with numbers instead of dots (and maybe with different sizes for the partitions):

🟨 .	🟨 .	🟨 .	🟨 .	🟨 .	🟨 .	⬜ .	🟩 .	🟩 .	🟩 .	🟩 .

After partitioning, quicksort makes two recursive calls to sort the two partitions. Show what happens in the **partitioning** step of those two recursive calls. That is, take each partition of the array, and partition it, again using the first element of the partition as pivot. After you've done this, show the new array and mark the new partitions (note that some partitions could become empty). For example:

🟥 .	🟥 .	⬜ .	🟨 .	🟨 .	🟨 .	⬜ .	⬜ .	🟩 .	🟩 .	🟩 .

Now repeat the same procedure once more: take each partition created in the previous step and apply the partitioning algorithm to it, using the first element as pivot. Show the new array and mark the new partitions. For example:

🟥 .	⬜ .	⬜ .	🟨 .	⬜ .	🟨 .	⬜ .	⬜ .	🟩 .	🟩 .	⬜ .

Also describe how you do the partitioning, e.g., by giving pseudo-code or by giving a good explanation.

Your answer should contain the following:

a) The starting array.
b) The three partitioned arrays as explained above.
c) A description of your partitioning strategy or algorithm.

Solution suggestion

a. The starting array:

10	16	8	2	1	3	4	2	3	5	7

b. The first partitioning:

Only 16 and 7 need to be swapped. Then the pivot (10) is swapped with 5. The resulting array:

5	7	8	2	1	3	4	2	3	10	16

b. The second partitioning:

In the left (yellow) partition, 7 and 3 are swapped, and 8 and 2 are swapped. Then the pivot (5) is swapped with 4. The right (green) partition only has one element.

4	3	2	2	1	3	5	8	7	10	16

b. The third partitioning:

The left (red) partition has pivot 4, which is larger than everything in the partition. So we only have to swap the pivot with the rightmost element (3). The middle (yellow) partition gets pivot 8, which has to swap with the remaining element (7).

3	3	2	2	1	4	5	7	8	10	16

c. The partitioning strategy:

We use the same strategy as in the course book (page 291). The main idea is that you have two indices, one going from the left and one from the right. Here's one version of the pseudocode:

```
int partition(A, low, high):
    pivot = A[low]
    i = low+1; j = high
    while i ⩽ j:
        if A[i] < pivot:
            i++
        else if A[j] > pivot:
            j--
        else:
            swap A[i] and A[j]
            i++; j--
    swap A[low] with A[j]
    return j
```

Basic question 3: Binary search trees

(You may answer this question using either red-black trees, AVL trees or AA trees.)

Show with an example that deleting an item from a red-black/AVL/AA tree, *using the deletion algorithm for ordinary unbalanced BSTs*, does not always produce a valid red-black/AVL/AA tree.

In other words: (1) start with a valid red-black/AVL/AA tree, (2) select an internal node to delete, (3) delete the node using the deletion algorithm for unbalanced BSTs, and (4) show that the resulting tree is not a valid red-black/AVL/AA tree.

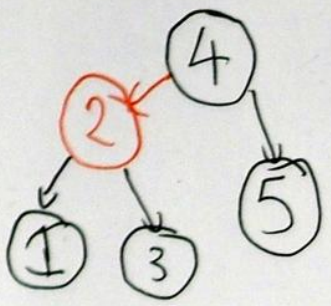
Important: In your example, the item that you delete must be an internal (non-leaf) node, meaning that it must have at least one child which is not `null`.

Your answer should contain the following:

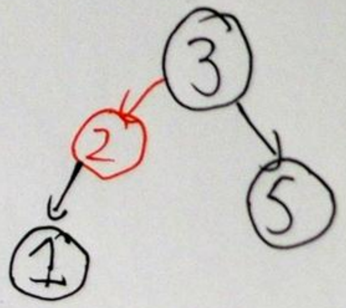
a) Which option you have chosen: red-black trees, AVL trees or AA trees.
b) The initial red-black/AVL/AA tree, and the item you have chosen to delete from it.
c) How the tree looks after the deletion.
d) An explanation of why the resulting tree is not a valid red-black/AVL/AA tree.

Solution suggestion (solution given for both red-black trees and AVL trees)

Red-black tree

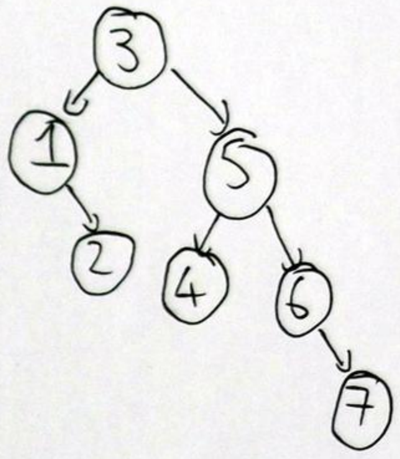


Delete 4 :

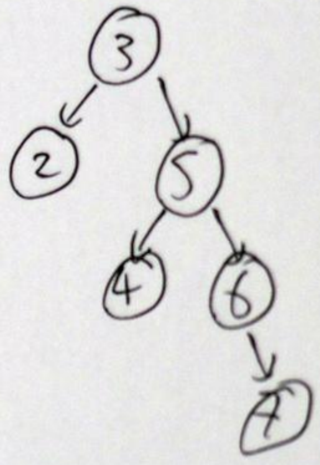


Not a R-B tree because
3 → 2 → 1 goes through 2 black nodes
3 → 2 → null goes through 1 black node
So perfect black balance is broken

AVL tree



Delete 1 :



Not an AVL tree because the root's children's heights differ by 2

Basic question 4: Hash tables

Assume you want to write a hash function for an array of 32-bit integers:

```
function int hashCode(int[] array)
```

Argue why the following two functions are bad hash functions. Give an example for each function where it fails to hash in a good and uniform way:

```
function int badHashCode1(int[] array):
    hash = 0
    for every element n in array:
        hash = hash + n
    return hash

function int badHashCode2(int[] array):
    hash = 0
    for every third element n in array:
        hash = 31 * hash + n
    return hash
```

Finally, suggest a better hash function `goodHashCode()` and argue why this is better than the previous ones.

Your answer should contain the following:

a) An example where `badHashCode1()` fails to hash in a good and uniform way
b) The same for `badHashCode2()`
c) A suggestion of a better hash function `goodHashCode()`
d) An explanation of why `goodHashCode()` is a better hash function

Solution suggestion

a) The function doesn't take the order between the elements into account, which means that all permutations of the same list will get the same hash code.

b) The function only looks at ⅓ of the elements, which means that you can change ⅔ of the elements however you want without changing the hash code. However, it multiplies the hash with a prime in every iteration, which means that the order is relevant, and you will get a better (almost) uniform distribution. (Since integers are 32- or 64-bits, the higher bits will be truncated on overflow, acting like an implicit "mod 2^{32}", which helps with the uniform distribution)

c) One way is to combine the two approaches – to multiply by 31, but on every element. In addition we can start with a non-zero number (e.g., the array length multiplied by another prime), so that not all zero-arrays will get the same hash code.

```
function int badHashCode2(int[] array):
    hash = 13 * array.length
    for every third element n in array:
        hash = 31 * hash + n
    return hash
```

d) By multiplying the hash with a prime number on every iteration, the hash will behave chaotic and therefore have a more uniform distribution. The position of the elements in the array will matter, and since we loop over all elements, the hash is affected by all elements.

e)

Basic question 5: Dynamic arrays or priority queues

Suppose that a program creates a **priority queue** and performs a series of *add* and *removeMin* operations. We are not told what order the program does the operations in, but we know that in total it calls *add* eight times and *removeMin* eight times.

The *add* operations add the following numbers in the order shown:

```
3  1  4  1  5  9  2  7
```

Every time the program calls *removeMin*, it prints out the value removed.

Which of the following sequences of numbers could the program print?
For each alternative, either give a sequence of *add* and *removeMin* operations that would give that sequence of numbers, or explain why the sequence is impossible.

a) `1  3  1  4  5  7  2  9`

b) `1  1  3  2  4  5  9  7`

c) `1  3  2  4  5  1  7  9`

Your answer should contain the following:

a) b) c) For each alternative:

- if it can be printed, give the sequence of *add* and *removeMin* operations;
- if it cannot be printed, explain why.

Solution suggestion

(a) is impossible. 2 is added before 7, so it is not possible for removeMin() to remove 7 before it removes 2.

(b) is possible. For example:

- add(3)
- add(1)
- add(4)
- add(1)
- removeMin() -> 1
- removeMin() -> 1
- removeMin() -> 3
- add(5)
- add(9)
- add(2)
- removeMin() -> 2
- removeMin() -> 4
- removeMin() -> 5
- removeMin() -> 9
- add(7)
- removeMin() -> 7

(c) is impossible. Two 1s are added before 2, so it is not possible for removeMin() to remove 2 before removing both 1s.

Basic question 6: Graphs

Draw a weighted undirected graph with the following properties:

- it contains exactly 6 nodes, named **A, B, C, D, E, F**
- every node is connected to at least 3 other nodes
- all edge weights are positive integers
- the minimal-weight (also called "shortest") path between **A** and **F** has weight equal to 10 + ***R***
- the minimum spanning tree has total weight 20 + ***R***

where ***R*** is *the digital root of your personnummer*:

https://www.thonky.com/nine-hours-nine-persons-nine-doors/digital-root

Here is a website where you can calculate digital roots:

https://www.thonky.com/digital-root-calculator/

Example: the digital root of personnummer 19711031-1234 is 6:

1+9+7+1+1+0+3+1+1+2+3+4 → 33 → 3+3 → 6

Your answer should contain the following:

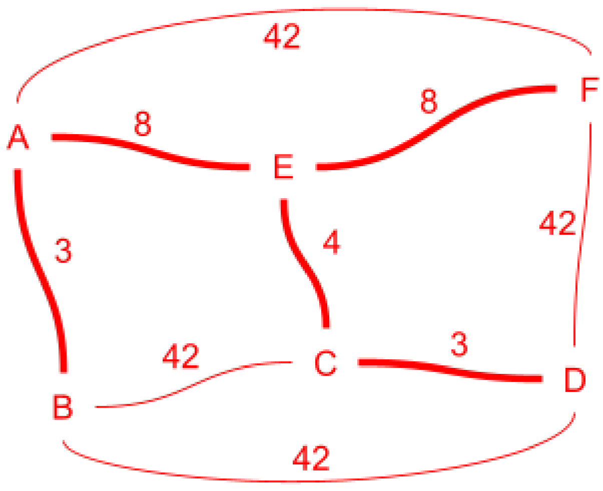
a) your personnummer
b) ***R*** – the digital root of your personnummer
c) the weight of the minimal-weight path between **A** and **F**
d) the weight of the minimum spanning tree
e) the graph, clearly marking the minimum spanning tree by e.g. using a different colour
f) the shortest path as a list of nodes: **A** → X → ··· → Z → **F**

Make sure the graph is readable!

Solution suggestion:

The shortest path is A → E → F, and has weight 8+8 = 16 = 10+***R***

The MST has total weight 8+8+3+4+3 = 26 = 20+***R***



Advanced question 7: Updatable meldable heap

A randomized meldable heap is a data structure that implements a priority queue using a binary tree. In a randomized meldable heap:

1. The tree always satisfies the *heap property*:
   the item stored in each node is less than (or equal to) the items stored in its children.
2. The tree is *not* forced to be balanced. In fact, it can become completely unbalanced!
   However, the algorithms used for updating the heap work fast even on unbalanced trees, by using randomness in a smart way.
   (You do not need to understand the use of randomness to solve this question!)

The main operation used in a randomized meldable heap is called `meld` (or merge), which combines two heaps into one and runs in *expected logarithmic time*, O(log *n*). The other operations are implemented in terms of `meld` and therefore also take logarithmic time (in the size of the heap).

For more information about randomized meldable heaps, you can read the following sources:

- Open Data Structures: http://opendatastructures.org/ods-java.pdf, section 10.2
- Wikipedia: https://en.wikipedia.org/wiki/Randomized_meldable_heap
- Lecture slides from the course: https://chalmers.instructure.com/courses/7567,
  see the lecture dated 3rd December

On the next page you can see a pseudocode implementation of a randomized meldable heap.
Please take a look at it now.

Now, your task is to implement the following method that *changes the priority* of a given node:

```
        public void updateKey(Node node, Item newKey)
```

Important: The `updateKey` operation must take *expected logarithmic time*, *O*(log *n*).
Give your answer in pseudocode similar to the one given on the next page.

Hint: Use the existing `meld` operation as much as possible.

Hint: As mentioned above, a randomized meldable heap does not try to keep the tree balanced. In fact, there is no restriction on the shape of the tree, and `meld` takes expected logarithmic time whatever the shape of the tree is. This means: 1) You can not assume that the tree is balanced. 2) You do not need to try to keep the tree balanced when changing the priority.

Your answer should contain:

a) Pseudocode for the `updateKey` operation.
b) An explanation why the operation takes expected logarithmic time O(log *n*)
   (you may assume without proof that `meld` takes expected logarithmic time).

Pseudocode for the class `RandomizedMeldablePQ`

Note that the inner class Node contains a parent pointer, which makes this implementation different from the lecture slides and the Wikipedia article.

```
class RandomizedMeldablePQ<Item>:
    Node root = null

    class Node:
        Item key
        Node left, right, parent
        public Node(Item key):
            this.key = key
            this.left = this.right = this.parent = null

    public void insert(Item x):
        root = meld(root, new Node(x))
        root.parent = null

    public Item delMin():
        Item item = root.key
        root = meld(root.left, root.right)
        if root is not null: root.parent = null
        return item

    public void meld(MeldablePQ<Item> other):
        root = meld(this.root, other.root)

    private Node meld(Node node1, Node node2):
        if node1 is null: return node2
        if node2 is null: return node1
        if node1.key > node2.key:
            swap node1 and node2
        if random coin toss == heads:
            node1.left = meld(node1.left, node2)
            node1.left.parent = node1
        else:
            node1.right = meld(node1.right, node2)
            node1.right.parent = node1
        return node1
```

Solution suggestion

The by far quickest and safest is to first delete the given node, and then insert it again.

```
public void updateKey(Node node, Item newKey):
    deleteNode(node)
    insert(newKey)
```

So now we only have to implement node deletion. Here's an explanation of how to do that:

> When you remove a node in the middle of the heap, the first thing you do is update the parent node so that the parent no longer points to the node being removed. Essentially, you now have two heaps: the original heap, and the sub-heap rooted at the node to be removed.
>
> Then, in the second heap (the one rooted at the node to be removed), you call remove. That removes the smallest item, which is the node you wanted to remove, and fixes up the heap.
>
> Finally, you merge that second heap with the main heap.
>
> (https://stackoverflow.com/a/53523650)

Here's code for that (where we don't call `delMin()` on the second heap, but instead meld the children directly):

```
public void deleteNode(Node node):

    // break the link from the parent to the node:
    if node.parent is null:
        root = null
    else if node.parent.left == node:
        node.parent.left = null
    else:
        node.parent.right = null

    // meld the children with the root of the heap:
    root = meld(root, meld(node.left, node.right))
    if root is not null:
        root.parent = null
```

What about efficiency?

- `deleteNode()` is logarithmic – because it calls `meld()` twice.
- `insert()` is also logarithmic – because it calls `meld()` only once.
- Therefore `updateKey()` is logarithmic, O(log *n*).

Advanced question 8: Choosing an appropriate data structure

The British National Corpus (BNC) is a database of English sentences, totalling 100 million words, taken from publications such as newspapers and books. The word *"in"* occurs 2 million times in the BNC. When the word *"in"* appears in a sentence, the next word in the sentence is most often *"the"* – the phrase *"in the"* occurs half a million times in the text of the BNC.

More generally, given a large text (such as the BNC), and a word (such as *"in"*), we may want to know which word most often follows it (in this case *"the"*). Your task is to design a class that computes this.

The class should be called `BigramStatistics` (a bigram is a pair of adjacent words such as *"in the"*) and should have the following methods:

- A **constructor** `BigramStatistics(String[] text)`.
  The argument **`text`** is the input text, given as an array of words. You may assume that all words are lowercase and that punctuation marks have already been removed.

- A **method** `String mostOftenFollows(String word)`.
  Returns the word which most often follows **`word`** in **`text`**.
  That is, if you find all places where **`word`** occurs in **`text`**, and look at the word that comes next in each case, then `mostOftenFollows(word)` should return the word that comes next in the highest number of cases.
  If several words come in equal first place, it may return any of them.
  If **`word`** does not occur in **`text`** or only occurs as the last word, it should return `null`.

See the next page for an example of how `BigramStatistics` should work!

The idea is that the user can create a `BigramStatistics` object for a text, and then call `mostOftenFollows` many times to ask about lots of different words. Therefore, it is OK if the constructor takes some time to run, but the `mostOftenFollows` method should run quickly. Specifically, the methods should have the following complexity, on a text of **n** words:

- The constructor should take at most **O(n log n)** time (linearithmic).
- `mostOftenFollows` should take at most **O(log n)** time (logarithmic).

(You may assume that comparing two strings or calculating their hashcode takes O(1) time.)

Write down your solution as a class, either in pseudocode or Java code. You may use existing data structures and algorithms in your solution, without explaining how they work.
Make sure it is clear **what fields your class uses**, and **how the constructor and method work**.

Hints:

- To loop over the bigrams, consider using a for-loop with an index `i`:
  `for (int i = 0; i < text.length-1; i++)`.
  You can then access the current and next word using `text[i]` and `text[i+1]`.
- You could start by building a table like the one shown on the next page.
  Then use that table to create a data structure for `mostOftenFollows` to use.

Example of `BigramStatistics` class

Suppose we are given the text "*it was the best of times, it was the worst of times, it was the age of wisdom*". The following table shows how many times each word pair (bigram) occurs in the text:

bigram (word pair)	**frequency**
it was	3
was the	3
the best	1
best of	1
of times	2
times it	2
the worst	1
worst of	1
the age	1
age of	1
of wisdom	1

We can see from the table that, for example:

- The word *"it"* is most often followed by *"was"* (3 times)
- The word *"of"* is most often followed by *"times"* (twice)
  (and less often followed by *"wisdom"*, once*)*
- The word *"the"* is followed equally often by *"best"*, *"worst"* and *"age"* (once each)

Here are some examples of how the `BigramStatistics` class should work on this text:

```
// Input is an array of lowercase words, without punctuation
String[] text =
  {"it", "was", "the", "best", "of", "times", "it", "was", "the",
   "worst", "of", "times", "it", "was", "the", "age", "of", "wisdom"}

// Create a BigramStatistics object from the text
BigramStatistics stats = new BigramStatistics(text)

// Should print "was"
print(stats.mostOftenFollows("it"))

// Should print "times" (not "wisdom")
print(stats.mostOftenFollows("of"))

// Could print either "best" or "worst" or "age"
print(stats.mostOftenFollows("the"))

// Both of these should print null
print(stats.mostOftenFollows("wisdom"))
print(stats.mostOftenFollows("monkey"))
```

Solution suggestion

```
class BigramStatistics:
    // We create a map, where the key is a word and the value is the most common following word.
    HashMap<String, String> mostOftenFollowsMap = new HashMap()

    // To find the most common following word, we just look it up in the map
    public String mostOftenFollows(String word):
        return mostOftenFollowsMap.get(word);

    // The real work happens in the BigramStatistics constructor,
    // which has to populate mostOftenFollowsMap.
    public BigramStatistics(String[] text):
        // Idea: Build a map from bigrams to frequencies (like the table on the previous page).
        // But arrange it as a nested map where:
        //   The key is the first word
        //   The value is a map from the second word to the frequency
        // By doing so, we can easily find all words that follow any given word (and their frequencies).
        HashMap<String, HashMap<String, Integer>> bigramFrequencies =
            new HashMap()

        // Build the map shown above.
        for i from 0 to text.length - 1:
            String thisWord = text[i]
            String nextWord = text[i+1]

            // Add 1 to the frequency associated with the bigram (thisWord, nextWord).
            if thisWord not in bigramFrequencies.keys:
                bigramFrequencies[thisWord] = new HashMap()
            if nextWord not in bigramFrequencies[thisWord].keys:
                bigramFrequencies[thisWord][nextWord] = 0
            bigramFrequencies[thisWord][nextWord] += 1

        // Now go through the bigram frequency table, and for each word, find out which word follows
it
        // the highest number of times.
        for each thisWord in bigramFrequencies.keys:
            mostCommonNextWord = null
            maximumFrequency = 0
            for each (nextWord, frequency) in bigramFrequencies[thisWord]:
                if frequency > maximumFrequency:
                    mostCommonNextWord = nextWord
                    maximumFrequency = frequency

            // Finally, store the most common following word in mostOftenFollowsMap.
            mostOftenFollowsMap[word] = mostCommonNextWord
```

Advanced question 9: What does this program do?

Consider the following program, which takes as input a list L of numbers:

```
h1 = new max heap
h2 = new min heap
for each number x in L:
    if h1 is not empty and x < h1.max():
        h1.add(x)
        if h1.size() > h2.size():
            h2.add(h1.removeMax())
    else:
        h2.add(x)
        if h2.size() > h1.size() + 1:
            h1.add(h2.removeMin())
    print(h2.min())
```

As an example, on the input

```
L = {3, 1, 2, 4, 6, 5, 9, 8, 1, 1, 1}
```

the program prints the output

```
3, 3, 2, 3, 3, 4, 4, 5, 4, 4, 3
```

What does the program calculate? That is, what do the numbers in the output mean? Explain your answer.

Here are some hints:

- Whenever an element is removed from `h1`, it is added to `h2`, and vice versa
- At all points,
    - either `h1.size() == h2.size()`
    - or `h1.size() == h2.size() - 1`

Solution suggestion

The code and the hints tell us that:

- `h1` and `h2` together contains all elements that we have seen so far
- `h1` and `h2` contain the same number of elements (or `h2` contains one more element)
- everything in `h1` is smaller than everything in `h2`

This means that `h2.min()` is the middle of all the elements we have seen so far.

Or, in other words, the program prints the ***running median*** of the elements in the list L.