Exam IN2010 fall 2024

27. November 2024

About the exam

- The exam consists of a tiny warm-up, followed by two main parts.
- The first part consists of small exercises, which are automatically corrected. No distinction is made between leaving a question unanswered and providing an incorrect answer; this means there is no reason to leave a question unanswered.
- The second part consists of more comprehensive exercises, where you to a greater extent have to provide pseudocode and written explanations.
- No aids are allowed.
- The full set of exercises is attached as a PDF.
- An English translation of the full set of exercises is attached as a PDF.

Comments and tips

- *Read the exercise very carefully*.
- Make sure you answer precisely what the exercise asks of you.
- Explain any assumptions you make.
- Make sure that your delivery is clear, precise, and easy to understand, both in terms of structure and content.
- If you get stuck on a problem, then you might want to proceed to another problem and return later.
- All exercises that require an implementation should be answered in *pseudo-code*. The important thing is that the pseudo-code is *easy to comprehend, unambiguous, and precise*.
- A *easily understandable, unambiguous and precise* explanation in natural language, may be worth more points than pseudocode that is difficult to comprehend, ambiguous or imprecise.
- In implementation tasks, lower runtime complexity is worth more points.
- You may assume that algorithms and data structures known from the syllabus are available, unless otherwise specified.

Warm-up 2 poeng

(a) What is an algorithm? Keep your answer brief (maximum four sentences).

(b) What is a data structure? Keep your answer brief (maximum four sentences).

Here, we are not after a “textbook answer". We are interested in seeing your understanding of the terms. Any reasonable answer gives a full score.

Huffman Trees 10 poeng

We want to compress text strings consisting of the symbols `A`, `B`, `C` and `D` using Huffman coding. The relative frequencies are given by the following frequency table:

A	B	C	D
1	3	7	2

(a) What are the code lengths for the different symbols in the corresponding Huffman tree?

(b) What do the leaf nodes in the resulting tree contain?

Generally, in a Huffman tree, each node v has:

- v.`symbol` – The symbol of the node (or **`null`**)
- v.`freq` – The frequency of the node
- v.`left` – Left child of the node (or **`null`**)
- v.`right` – Right child of the node (or **`null`**)

A Huffman tree is built from a set of pairs (s, f) where s is a symbol and f is a frequency.

(c) Describe the algorithm for building a Huffman tree. You can outline it with pseudocode, but a good description in natural language works just as well.

Etymological Dictionary 10 poeng

An etymological dictionary is a reference work for the origin of a word. Here is an example from the category Mathematics:

> **hexagon:** six-angled figure. From Greek *hex* six"+ *gonía* "angle".
> *(Source: Norwegian Etymological Dictionary – thematically arranged, Yann De Caprona)*

Assume that a word object v only has the following fields:

- v.`string` – The word represented as a string
- v.`category` – A string indicating the category the word belongs to

You can assume that the category of a word is one of N categories, where $N < 100$.

(a) The array A contains the entire etymological dictionary arranged alphabetically (i.e., arranged by `string`). Write a procedure that *stably* sorts the array A *thematically* (i.e., arranged by `category`). The procedure shall not include calls to sorting algorithms.

```
Input: An array A with n word objects
Output: An array A with the same n word objects arranged by category
1 Procedure EtymologySort(A)
  | // ···
```

(b) Now assume that you have a dictionary K, which associates each category with an interval indicating where the category starts and ends in the array. For each word object v, $K[v.\texttt{category}] = [\texttt{low}, \texttt{high}]$, where `low` and `high` give the index of the first and the last word object that belongs to the category of v. You now want to search for a given word object in the array A as returned from `EtymologySort`.

Describe how you can use an algorithm known from the curriculum to check if a word object v exists in the array A. Also, provide the runtime complexity of the algorithm.

Jazz or Classical? 10 poeng

You want to invite friends to a concert but are unsure whether to invite them to *jazz* or *classical*, which are the only two genres you like. Through a small music network, you have a good overview of your own musical taste, the musical tastes of your friends, your friends' friends, your friends' friends' friends, etc. The musical taste of a friend (or a friend's friend, etc.) v is given by a dictionary M, where $M[v]$ is a set of genres. For example, $M[v]$ might be $\{$`"jazz"`, `"pop"`, `"folk"`$\}$.

(a) How can you represent the music network as a graph? What do the nodes and edges represent in the graph?

Although you want to bring as many people as possible to the concert, there is a limit to how peripheral you want your friends to be. You set a limit k which indicates how many levels of friends you are willing to go through. For example, if $k = 1$, you will only invite your own friends. If $k = 2$, you will invite your own friends and their friends.

(b) Write an algorithm that takes a graph $G = (V, E)$, a dictionary M, a start node $s \in V$, and a positive integer k as input. It should return `"jazz"` or `"classical"` depending on which genre most people like within k levels of friends. You can assume that the number of people who like jazz and classical is not the same.

Input: A graph $G = (V, E)$, a dictionary M, a start node $s \in V$, and an integer k
Output: A string `"jazz"` or `"klassisk"`

```
1 Procedure JazzOrClassical(G, M, s, k)
  | // ···
```

(c) Now you want to send out invitations. You want to *first* invite your own friends, *then* your friends' friends, and so on. Which algorithm, known from the curriculum, is suitable for sending out invitations in this way?

Python 3.6 10 poeng

Assume that Python dictionaries (`dict`), up to version 3.6, were implemented with linear probing as taught in IN2010. (This is a slight modification of the truth).

In December 2012, Raymond Hettinger proposed a change to the underlying representation of dictionaries. In the proposal, Hettinger described that the current representation was unnecessarily inefficient.

It organized the data in an array `entries`, where each element has space for a key k and a value v. The array has size N and a lot of empty space. Hettinger proposed reorganizing the data. The key-value pairs should instead be stored in a tightly packed, dynamic array `entries` with n elements, where n is the number of key-value pairs. The elements in `entries` can be referenced from another array `indices` with N elements (which still has a lot of empty space).

For example, the following dictionary:

```
d = {}
d["timmy"] = "red"
d["barry"] = "green"
d["guido"] = "blue"
```

was previously represented like this:

```
entries = [(_, _),
           ("barry", "green"),
           (_, _),
           (_, _),
           (_, _),
           ("timmy", "red"),
           (_, _),
           ("guido", "blue")]
```

where `(_, _)` indicates an empty space. The proposal is to represent it like this:

```
indices = [_, 1, _, _, _, 0, _, 2]
entries = [("timmy", "red"), ("barry", "green"), ("guido", "blue")]
```

The above proposal led to the implementation of dictionaries in Python being changed from version 3.6.

(a) Briefly explain, in natural language, how the algorithms for lookup and insertion with linear probing need to be changed to account for the new structure.

(b) What is a rehash? What causes a rehash?

(c) How will a rehash proceed in the new structure?

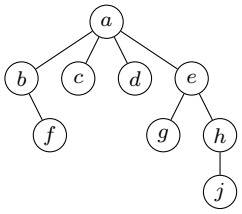
(d) What are the advantages of the new structure?

Lowest Common Ancestor — 12 poeng

The *lowest common ancestor* of two nodes u and v in a tree T is the lowest (i.e., the node with the highest depth) node that has both u and v as descendants.

Here are some examples from the tree on the right:

- The lowest common ancestor of a and a is a
- The lowest common ancestor of b and e is a
- The lowest common ancestor of g and j is e
- The lowest common ancestor of e and j is e



(a) Briefly explain why there must exist a common ancestor for every pair of nodes.

(b) Assume that if v is a node, v has only the following fields:

- v.`parent` – parent node of v
- v.`children` – children nodes of v
- v.`depth` – depth of v

Provide an algorithm that, given two nodes u and v, finds and returns their lowest common ancestor.

```
Input: Two nodes u and v
Output: Lowest common ancestor of u and v
1 Procedure LowestCommonAncestor(u, v)
  | // ···
```

(c) Now assume that the tree is a *binary search tree of unique integers*. If v is a node, then v has only the following fields:

- v.`element` – the integer stored in the node
- v.`left` – left child of v
- v.`right` – right child of v

Provide an algorithm that, given the root of the tree and the elements x and y, returns the lowest common ancestor of the nodes that have the elements x and y, where $x < y$. You can assume that both elements x and y exist in the tree.

```
Input: A node v, and two values x and y where x < y
Output: Lowest common ancestor of the nodes that have the elements x and y
1 Procedure LowestCommonAncestorBST(v, x, y)
  | // ···
```

(d) What is the runtime complexity of your algorithm from (c) for a tree with n nodes in the worst case? What is the runtime complexity if you assume that the tree is balanced?

If you did not answer (c), you can provide the answer for your solution to (b) instead.

Median Queue 12 poeng

A *median queue* is an abstract data type for a collection that supports the following operations:

- `insert`(Q, x) – inserts an element x into the queue
- `removeMedian`(Q) – removes and returns the *median* element from the queue

The median is the "middleelement of the collection when the elements are arranged from smallest to largest. If we insert the numbers $8, 3, 1, 5$, and 7 into a median queue, then:

- the first call to `removeMedian` will give 5
- the second call to `removeMedian` will give 3
- the third call to `removeMedian` will give 7
- the fourth call to `removeMedian` will give 1
- the fifth call to `removeMedian` will give 8

Note that we choose the smaller of the two middle elements when the queue contains an even number of elements, as shown in the example above.

(a) If we insert the numbers $4, 1, 8, 3, 2$ into a median queue, what will the first three calls to `removeMedian` give? Provide the answer as a comma-separated list of numbers.

(b) Explain *briefly* how you would represent a median queue.

(c) Explain how you would implement the operations `insert` and `removeMedian`. You can use pseudocode as part of your explanation, but it is not a requirement. All data structures mentioned in the curriculum are available to you.

(d) Based on your explanation in (c), what can you say about the runtime complexity of `insert` and `removeMedian`?

(e) If we take n elements from a median queue using `removeMedian`, and insert them into a binary search tree, one by one, what will the resulting binary search tree look like? Explain briefly.

Round Trip with the Norwegian Trekking Association (DNT) 12 poeng

The student representative committee at the Department of Informatics has decided to organize a round trip in a beautiful Norwegian mountain landscape for all IFI students. This coincides with the launch of a new app by the Norwegian Trekking Association, together with public datasets, making it possible to make a well-informed route selection. Since most students at IFI have a fondness for graphs, it is inevitable that they will use the dataset to build a graph that captures enough information to algorithmically determine the best round trip.

The graph built is a directed graph $G = (V, E)$, where each node $u \in V$ represents a location, and an edge $(u, v) \in E$ represents a hiking path from u to v. No nodes have an edge to themselves.

(a) It turns out that the graph has a very special property: *Each node is part of at most one cycle*. Each cycle represents a *round trip*.

Provide an algorithm that finds all the round trips in the graph and state the runtime complexity. You may call algorithms known from the curriculum.

(b) One of the round trips from (a) is selected as the destination, where t is a node in the round trip. The trip starts at a *departure point*, which is easy to reach by public transport. A departure point is identified by nodes with in-degree 0. The last thing remaining in the planning is to choose a suitable departure point to start the trip in order to reach t.

Fortunately, the dataset also contains information about the average time a hiker takes along a hiking path. This is given as a weight function w, such that for each edge $(u, v) \in E$, $w(u, v)$ indicates the time taken as a positive integer. We will use the weights to determine which departure point is best suited to reach the destination.

The input is a directed and weighted graph $G = (V, E)$ with weight function w, a set of departure points U, and a node t in the destination. The output is the node $s \in U$ such that the distance between s and t is as low as possible.

Provide an algorithm that effectively finds the departure point with the lowest distance to t in the destination, and state the runtime complexity. You may call algorithms known from the curriculum.