Exam IN2010 fall 2022

1. December 2022

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.
- Answers are entered in Inspera, and there is no possibility of uploading handwritten answers.
- No aids are allowed.

Comments and tips

- It is recommended to briefly read through the exam before you start. The full set of exercises is attached as a PDF.
- *Read the exercise very carefully*.
- Make sure you answer precisely what the exercise asks of you.
- 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*.
- An *easy to comprehend, unambiguous, and precise* natural language explanation, may yield more points than pseudocode that is difficult to comprehend, ambiguous or imprecise.

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.

Miscellaneous 10 poeng

(a) Binary trees consist of nodes that have up to two children.

(b) If an algorithm uses $\mathcal{O}(n)$ number of steps, we say it has linear runtime complexity.

(c) Runtime complexity of Mergesort is $\mathcal{O}(n \cdot \log(n))$.

(d) It is impossible to check if an array is sorted in $\mathcal{O}(n)$.

(e) Binary search on arrays is significantly faster than binary search on linked lists.

(f) Runtime complexity says something about how many steps an algorithm uses relative to the size of the input.

(g) If one algorithm has better runtime complexity than another algorithm, then it will always use fewer steps regardless of the size of the input.

(h) Huffman coding is used to compress data.

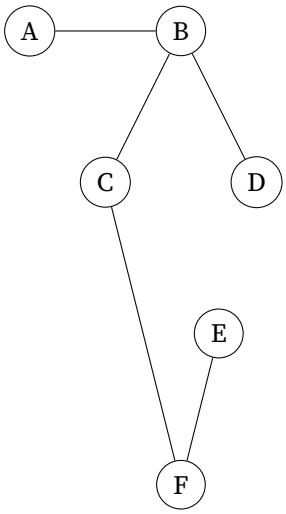
(i) The root node of a tree is the only node that does not have a parent.

(j) A graph with n nodes cannot have more than n edges.

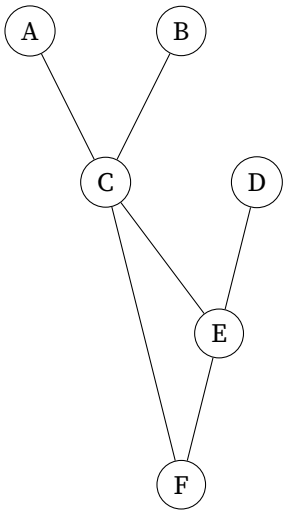
Graph properties 10 poeng

Here are two graphs:

G_1

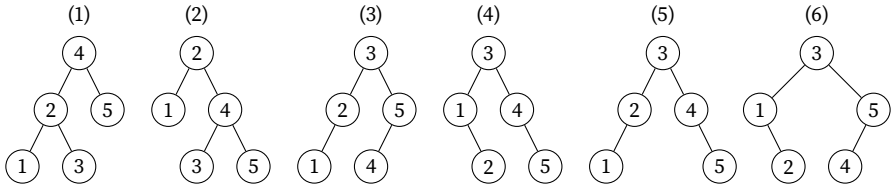


G_2



Balanced search trees 6 poeng

The following are all AVL trees that contain the numbers 1 to 5.



(a) Which of the AVL trees above do you get if you enter the numbers 1 to 5 in the order $2, 4, 3, 1, 5$? Enter the answer as a number between 1 and 6.

(b) How many of the trees above can be colored as a red-black tree?

(c) If we add 6 to all the AVL-trees above, in which trees will rotations occur? Enter the answer as the sum of all the trees in question. Thus, if only rotations occur in the first tree, the answer is 1, and if rotations occur in all the trees, the answer is 21 (because $1 + 2 + 3 + 4 + 5 + 6 = 21$).

Some questions about priority queues and binary heaps 10 poeng

By *heap* we mean arrays representing binary min-heaps.

(a) An array with n elements can be turned into a heap in $\mathcal{O}(n)$.

(b) A min-heap becomes a max-heap by reversing the array.

(c) All elements at depth d in a heap are smaller than all elements at depth $d+1$.

(d) Insertion into a heap with n elements is $\mathcal{O}(\log(n))$ at worst case.

(e) An AVL tree can be used as a priority queue with the same runtime complexity as a heap.

(f) The nodes along a path from the root node to a leaf node of a heap are ordered from smallest to largest.

(g) If we reorder two sibling nodes of a heap, it remains a heap.

(h) A priority queue with constant time on all operations would give Dijkstra a runtime complexity of $\mathcal{O}(|V|+|E|)$.

(i) One can find the *largest* element in a *min*-heap in $\mathcal{O}(\log(n))$.

(j) *Over half* of the elements of a heap are located on the *two deepest* levels.

Running time on graph algorithms 8 poeng

For each graph algorithm, check the (lowest) correct runtime complexity.

	$O(1)$	$O(\|V\| + \|E\|)$	$O((\|V\| + \|E\|) \cdot \log(\|V\|))$	$O(\|V\| \cdot \|E\|)$
`DFSFull`				
`Prim`				
`TopSort`				
`BellmanFord`				

Short descriptions of the algorithms:

- `DFSFull`: Visits all vertices in a graph exactly once (depth-first)
- `Prim`: Finds a minimal spanning tree of a given graph
- `TopSort`: Provides a topological sort of the vertices in a given graph
- `BellmanFord`: Finds shortest paths from one to all other vertices

Linear probing 10 poeng

(a) We start with an empty array of size 10.

0	1	2	3	4	5	6	7	8	9

The hash function to use is $h(k, N) = k \mod N$, as for this example becomes the same as $h(k, 10) = k \mod 10$. Thus, a number hashes to its last digit.

Use *linear probing* to insert these numbers into the array in the given order:

$$21, 54, 82, 10, 20, 44$$

Fill in the table as it looks after all the numbers have been entered with linear probing. Write the answer as a comma-separated list, where _ can be used to indicate an empty space.

(b) Explain briefly how the algorithm for insertion by linear probing works, and sketch out the algorithm with pseudocode. You can assume that the input array is not full, and that you have a hash function h such that $h(k, N)$ gives a number between 0 and $N - 1$ for an arbitrary key k. Free space in the array is indicated by **`null`**.

```
Input: An array A of size N, a key k and value v
Output: An array containing (k, v)
1 Procedure LinearProbingInsert(A, k, v)
  | // ...
```

Garbage collection 8 poeng

Many modern programming languages have a *garbage collector*, which is a procedure that frees memory that is guaranteed not to be used in the program anymore. Your task is to derive a simple algorithm for garbage collection.

We can assume that everything that is stored are *objects*, where an object can refer to other objects. We let $G = (V, E)$ be an object graph, where V represents all the objects created, and a (directed) edge from u to v means that the object u has a reference to the object v. In addition, we have a set R with all the objects that can be referenced directly (typically objects which are referenced by program variables). All objects in R are also in V. Objects to which there is *no* reference via objects in R are guaranteed not to be used in the program, and should be freed.

This means that none of the objects in R should be freed, nor should any objects that are reachable through references from objects in R. The objects that should be freed are not in R, and cannot be reached from any object in R.

Suppose you have a procedure `Free` that frees an object. You should give a procedure `GarbageCollect` which takes a graph $G = (V, E)$ and a set R as input, and calls `Free` on all objects that *cannot* be reached from R.

Input: An object graph $G = (V, E)$ and a set R of objects
Output: Free all objects to which there is no reference

```
1 Procedure GarbageCollect(G, R)
  | // ···
```

Auto complete 6 poeng

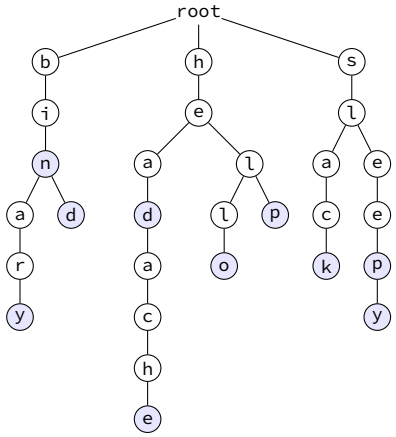
Most text editors have functionality for "auto completion", where a small box appears with possible ways to complete a started word. Below we provide two possible data structures that can be used to implement a simple mechanism for completing a started word.

We show examples of how the data structures look if they store the words: `bin`, `binary`, `bind`, `head`, `headache`, `hello`, `help`, `slack`, `sleep`, `sleepy`.

Strategy 1

The first strategy is to use a *prefix-tree* that holds strings. A prefix tree has a root. If a string s is inserted into the tree, the root node has a child with the first letter of s. That node in turn has a pointer to the second letter of s, and so on. The last letter of s is marked as a *terminal* node (the terminal nodes are colored blue in the example). That is, if you follow a path from the root node to a terminal node, then you have spelled a word that is in the tree.

From a prefix tree it is easy to find all ways to complete a started word. It is done by returning all paths from the root node that begin with the letters from a given word and ends in a terminal node.



Strategy 2

The second strategy is to use a hashmap, where we let each prefix of a word map to a list of all possible ways to complete the word. If `H` is a hashmap, and s is a string to be added, we consider each prefix p of s and add s to the list `H[`p`]`. In the example to the right, you see how the hashmap might look if all the example words have been added (note that some of the lists are shortened, indicated by "...").

From such a hashmap it is very easy to find all ways to complete a beginning words. This is done by looking up the started word in the hash map and return the list.

Nøkkel	Verdi
""	"bin", ..., "sleepy"
"b"	"bin", "binary", "bind"
"bi"	"bin", "binary", "bind"
"bin"	"bin", "binary", "bind"
"bina"	"binary"
"binar"	"binary"
"binary"	"binary"
"bind"	"bind"
"h"	"head", ... "help"
"he"	"head", ... "help"
"hea"	"head", "headache"
"head"	"head", "headache"
"heada"	"headache"
"headac"	"headache"
"headach"	"headache"
"headache"	"headache"
"hel"	"hello", "help"
"hell"	"hello"
"hello"	"hello"
"help"	"help"
"s"	"slack", "sleep", "sleepy"
"sl"	"slack", "sleep", "sleepy"
"sla"	"slack"
"slac"	"slack"
"slack"	"slack"
"sle"	"sleep", "sleepy"
"slee"	"sleep", "sleepy"
"sleep"	"sleep", "sleepy"
"sleepy"	"sleep"

Discuss the advantages and disadvantages of the two different strategies both with regards to runtime and memory usage

Bucket queue 10 poeng

A *bucket queue* (or a bucket queue) is a data structure, inspired from *bucket sort*, and can be used as a *priority queue*.

In a bucket queue, items can be added with a priority between 0 and $N - 1$, where N is a positive integer determined when the queue is created. A call to `RemoveMin` should remove and return an element with the lowest possible priority; if there are several elements with the same priority, it does not matter which element is removed and returned.

(a) Explain *briefly* which data structure is well suited for a bucket queue.

(b) Give an algorithm for `Insert` for a bucket queue.

```
Input: A bucket queue Q with priorities from 0 to N − 1, and an element x with priority p
Output: Q with x inserted with priority p
1 Procedure Insert(Q, x, p)
  | // ···
```

(c) Give an algorithm `RemoveMin` for a bucket queue. You can assume that the queue is not empty.

```
Input: A bucket queue Q with priorities from 0 to N − 1
Output: A x with the lowest possible priority, which is removed from Q
1 Procedure RemoveMin(Q)
  | // ···
```

(d) Assume that $N = 100$. Specify the runtime complexity of `Insert` and `RemoveMin` for a queue of n elements.

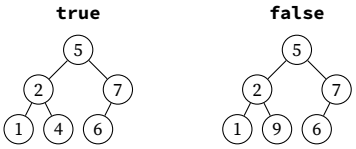
Is the binary tree a search tree? 10 poeng

Suppose you are given a binary tree B of unique integers. If v is a node in the binary tree, then

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

We want to check if the binary tree is also a binary *search tree*. Below you will find the specification for the algorithm, and two examples of trees that respectively should return **`true`** and **`false`**.

```
Input: The root node v of a binary tree B
Output: Returns true if the binary tree is a binary search
        tree, false otherwise
1 Procedure CheckBST(v)
  | // ···
```



There are several good solutions to this problem. The following can be helpful when thinking of a solution:

- You can assume that you have procedures `FindMin` and `FindMax` which respectively finds the smallest and largest number in the binary tree. Since the tree is not guaranteed to be a binary search tree (or balanced) then these procedures will have linear time.
- It may be a good idea to split the algorithm by creating a help procedure.

(a) Write down the property a binary tree must have to be called a binary *search tree*.

(b) Complete the procedure above. A lower runtime complexity gives a higher score.

(c) Give the runtime complexity of your algorithm with respect to the number of nodes in the binary tree.

Minimal front page 10 poeng

You are given a graph $G = (V, E)$ that represents a domain, where every vertex represents a page in the given domain. If there is an edge from u to v, then there is a link from page u that links to v.

You have been assigned to make a front page (resulting in a new vertex in the graph), that must satisfy the following:

- Every page must be reachable from the front page.
- The front page must contain as few links as possible.

You must describe algorithms that calculate how many links the new front page has to contain, depending on if the graph is:

(a) An undirected graph. Provide a short description of the algorithm with natural language (you may refer to algorithms in the course).

(b) A directed and acyclic graph. Provide a short description of the algorithm with natural language (you may refer to algorithms in the course).

(c) A directed graph (that may contain cycles). Provide a short description of the algorithm with natural language (you may refer to algorithms in the course).