

# Reexam for DIT181+182, LET375, DAT037+038, TDA416+417, DAT495, and DAT525

## Datastrukturer och algoritmer

Thursday, 2022-08-18, 8:30–12:30

## Solution suggestions

---

**Teachers**      **DIT182** and **DIT181**: Christian Sattler, 0723–526145  
**The rest**: Peter Ljunglöf, 0766–075561

**Allowed aids**   None

**Exam review**   When the exams have been graded, they are available for review in the CSE student office at Lindholmen (DIT181+182) and Johanneberg (the rest).

If you want to discuss the grading, please contact the course responsible and book a meeting. In that case, you should leave the exam in the student office until after the meeting.

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

There are **6 sections**, each containing **2 questions**, making **12 questions total**.

Each question is graded as **correct** or **incorrect**. Here is what you need to do to get each grade:

Grade	Sections with $\geq 1$ correct answer	Sections with both answers correct
3	5	—
4	5	2
5	6	4

**Good luck!**

# Section 1: Complexity

## Question 1A

The following function produces a copy of an array of values sorted according to a separately specified array of keys. We assume that the *keys* and *values* arrays have the same length  $n$ . We are interested in the asymptotic complexity of *sort\_values* in terms of  $n$ . We assume that comparing two keys takes constant time.

```
function sort_values(keys : Array, values : Array) → Array:
    map = new SortedMap()
    i = 0
    while i < length(keys):
        map.put(keys[i], values[i])
        i = i + 1
    result = new Array of size length(keys)
    i = 0
    for each key, value in the entries of map:
        result[i] = value
        i = i + 1
    return result
```

- a) What is the asymptotic complexity if SortedMap is a binary search tree (BST)?
- b) What is the asymptotic complexity if SortedMap is an AVL Tree?

Write your answer in  $O$ -notation. Be as exact and simple as possible. Justify why the complexity of the function has this order of growth.

### Answer:

- a) Both loops iterate over each element in the array, so  $n$  times. The first loop calls `map.put` for each element, and in the worst case BST is unbalanced and so `put` has linear complexity.

So the first loop is  $O(n) \cdot O(n) = O(n^2)$ , and the second is  $O(n)$ . In the end the complexity of *sort\_values* is quadratic,  $O(n^2)$

- b) This time the tree will always be balanced, so adding elements to it is logarithmic in the worst case. Therefore the first loop (and the whole program) is  $O(n) \cdot O(\log n) = O(n \log n)$

## Question 1B

Three more questions about the function *sort\_values*:

- a) Give an example that displays the worst-case complexity when SortedMap is a BST.  
Answer with example arrays *keys* and *values* that make *sort\_values* perform at the worst-case complexity you answered in 1A(a).

Note that since the complexity depends on the size  $n$  of the input, your example should be generic in  $n$ , that is, you should make it clear how the *keys* and *values* arrays would look for any value of  $n$ .

- b) The function will not work as intended if SortedMap was a hash table. Explain why.
- c) However, if we used a hash table, the function would still be able to run. What would the worst-case complexity of *sort\_values* be in that case (assuming a perfect hash function)?

### Answer:

- a) A simple example is if the *keys* array is sorted (in any order). Then the BST will be completely unbalanced (to the left or the right), and adding new elements will take  $O(n)$  time.

How the *values* array looks doesn't matter at all.

- b) The keys in a hash table are not stored in order, so when we iterate through the map entries we will get them in an unknown order. So *sort\_values* will not sort the values according to the keys.
- c) If we have a perfect hash function, adding elements to a hash table is constant time,  $O(1)$ . So the first loop becomes  $O(n) \cdot O(1)$ , and therefore the complexity of *sort\_values* will be  $O(n)$ .

## Section 2: Sorting

### Question 2A

Perform one level of split+sort+merge when **merge sorting** the following array:

42	12	54	72	36	82	99	66	6	27
0	1	2	3	4	5	6	7	8	9

Show the following:

- The result of splitting the input array. Explain how you do it.
- The result of merge sorting each of the splits.
- The result of merging the sorted sub-arrays. Explain how you do it.

### Answer:

- The easiest way is to split in half, so you get the two arrays:

**[42, 12, 54, 72, 36] and [82, 99, 66, 6, 27]**

- Calling merge sort on each half gives us two sorted lists:

**[12, 36, 42, 54, 72] and [6, 27, 66, 82, 99]**

- Merging compares the heads of each list and puts the smallest in a new list:

- [12, 36, 42, 54, 72] + [6, 27, 66, 82, 99] → []**
- [12, 36, 42, 54, 72] + [ 27, 66, 82, 99] → [6]**
- [ 36, 42, 54, 72] + [ 27, 66, 82, 99] → [6, 12]**
- [ 36, 42, 54, 72] + [ 66, 82, 99] → [6, 12, 27]**
- [ 42, 54, 72] + [ 66, 82, 99] → [6, 12, 27, 36]**
- [ 54, 72] + [ 66, 82, 99] → [6, 12, 27, 36, 42]**
- [ 72] + [ 66, 82, 99] → [6, 12, 27, 36, 42, 54]**
- [ 72] + [ 82, 99] → [6, 12, 27, 36, 42, 54, 66]**
- [ ] + [ 82, 99] → [6, 12, 27, 36, 42, 54, 66, 72]**
- [ ] + [ 99] → [6, 12, 27, 36, 42, 54, 66, 72, 82]**
- [ ] + [ ] → [6, 12, 27, 36, 42, 54, 66, 72, 82, 99]**

## Question 2B

A *run* in a list of integers is a maximal non-empty contiguous subsequence of increasing numbers:

- *Contiguous* means that the sequence cannot have gaps. It is specified by a *start* index and an *end* index in the list. In Java, this is called a sublist; in Python, it is called a slice.
- *Increasing* means that if  $x$  comes before  $y$ , then  $x \leq y$ .
- *Maximal* means that the (contiguous, increasing) sequence cannot be extended.

Every list of integers decomposes uniquely into a list of successive runs. For example, the array in Question 2A has these runs:  $\langle\langle 42 \rangle\rangle$ ,  $\langle\langle 12\ 54\ 72 \rangle\rangle$ ,  $\langle\langle 36\ 82\ 99 \rangle\rangle$ ,  $\langle\langle 66 \rangle\rangle$ ,  $\langle\langle 6\ 27 \rangle\rangle$ .

Describe an algorithm, in pseudocode or your favourite programming language, that finds the runs of a given array. It should return the **list of start indices** of the runs in order:

**function** *find\_runs*(*list* : Array)  $\rightarrow$  List of integers

For example, when called on the array in Question 2A, it should return  $[0, 1, 4, 7, 8]$ .

*Note:* You can use a dynamic array to implement the result list. For example, you can initialise the list using  $\langle\langle \text{runs} = \text{new DynamicArray}() \rangle\rangle$  and add an element at the end using  $\langle\langle \text{runs.append}(x) \rangle\rangle$ . Don't forget to return *runs* when your function is finished.

**Answer:**

```
function find_runs(list : Array)  $\rightarrow$  List of integers:
    runs = new DynamicArrayList()
    for i from 0 to length(list):
        if  $i-1 \geq 0$  and  $\text{list}[i-1] \leq \text{list}[i]$ :
            // index i belongs to a previous run
        else:
            // index i starts a new run
            runs.append(i)
    return runs
```

*Note (not relevant for solving the problem):* Merge sort can be modified to merge the runs of an array instead of splitting the array in half. This is the basis for TimSort, the built-in sorting algorithm in Python, which is efficient on many real-life examples of lists. Example code for merging the runs is shown on the next page. You absolutely don't need it to solve the problem, it's only there for your amusement.

## Code for merging the runs (not relevant for solving this exam)

Here is one possible implementation for merging the runs that you returned in Question 2B.

```
function run_based_merge_sort(list : Array) → Array:
    runs = find_runs(list)
    return merge_runs(list, runs)

function merge_runs(list : Array, run_starts : List of integers) → Array:
    result = []
    run_ends = run_starts[1:] + [length(list)]
    for (start, end) in zip(run_starts, run_ends):
        result = merge(result, list[start : end])
    return result
```

This code calls the standard *merge* function which you used in Question 2A(c).

This is not the most efficient implementation of run-based merge sort! The better versions, such as Python's built-in TimSort, use heuristics for selecting which runs to merge and when.

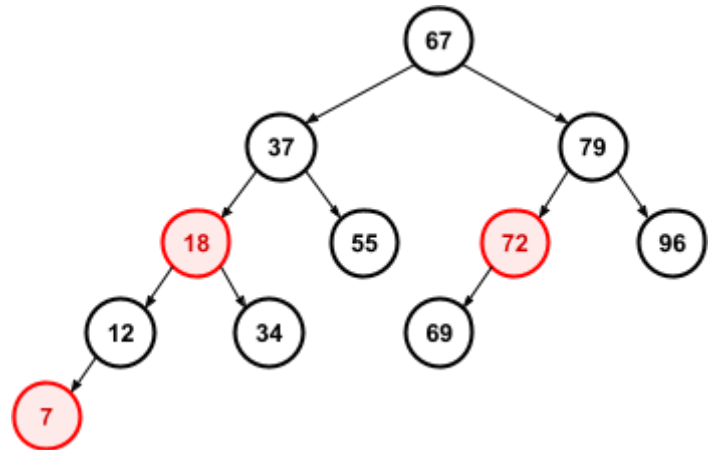
*Note:* Our pseudocode borrows some expressions from Python:

- $a[j : k]$  means the  $(k-j)$  element sublist  $[a_j, a_{j+1}, \dots, a_{k-1}]$ , and  $a[j:] == a[j : \text{length}(a)]$ ,
- to concatenate two lists  $a$  and  $b$ , we write  $a + b$ ,
- the *zip* function takes two lists of the same length and returns a corresponding list of pairs of the same length.

## Section 3: Search trees

### Question 3A

The tree to the right is neither a valid AVL tree nor a valid red-black tree.



- a) Why is it not a valid red-black tree?

Explain how the invariant is invalidated (and where in the tree).

**Answer:** No it's not. All paths from the root to a leaf passes through 3 black nodes, except the path  $67 \rightarrow 79 \rightarrow 72$ ! Note that this is a leaf path, since the right child of 72 is a leaf.

- b) Is it possible to recolour the nodes to make it a valid red-black tree?

Either give the new colouring (e.g., by listing all nodes that should be red), or explain why the tree cannot be made red-black.

**Answer:** Colour the node 69 red instead of 72. That is, the nodes 7, 18 and 69 should be red.

- c) Why is it not a valid AVL tree?

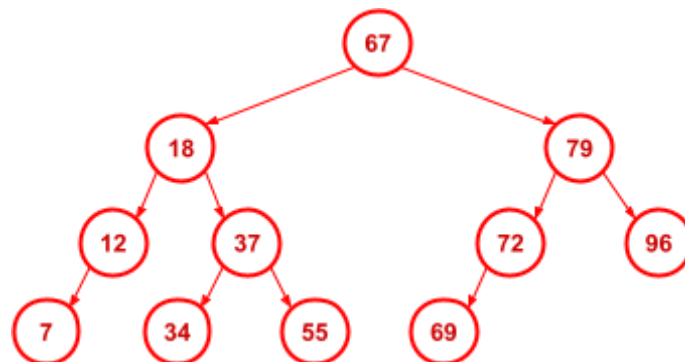
Explain how the invariant is invalidated (and where in the tree).

**Answer:** The node 37 is imbalanced since its left child has height 3 and its right child has height 1.

- d) Perform one AVL rotation to make it balanced, and draw the new resulting tree.

Explain which rotation you performed, and where in the tree.

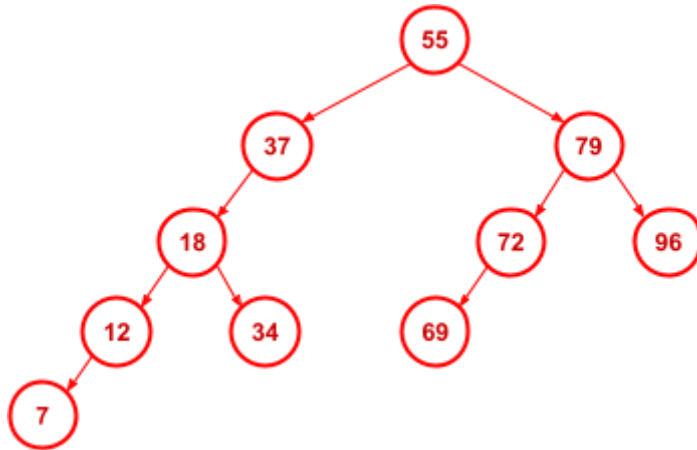
**Answer:** Perform a single right rotation at node 37. This results in the following AVL tree:



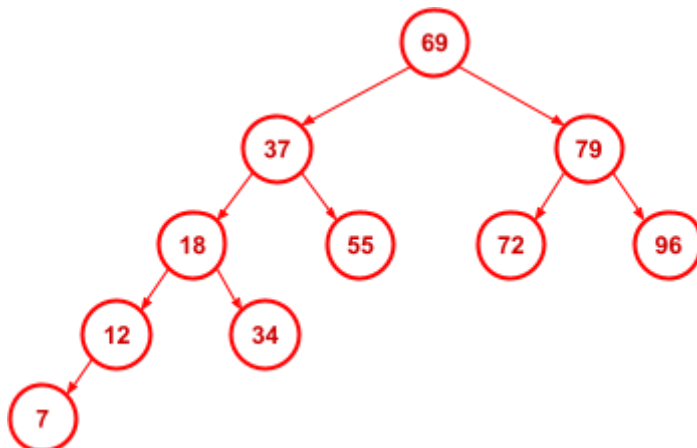
### Question 3B(a, b)

Now consider the tree above as a plain binary search tree (BST).

- a) Delete the root node (67) from the tree, and show the resulting tree.



- b) There are two possible ways to delete a node which has two children.  
Now show the resulting tree after deleting 67 from the initial tree in the alternative way.





### Question 3B(c)

- c) Implement the method *delete* for the class BST, which removes the given value from the tree (if it exists). You can pick any of the two possible ways to delete nodes with two children.

**class** Node:

```
value : int
left  : Node
right : Node
```

**class** BST:

```
root : Node
```

**method** delete(*value* : int):

```
root = delete_internal(root, value)
```

**method** delete\_internal(*node* : Node, *value* : int) → Node:

```
// The value is not in the tree.
```

```
if node is null:
```

```
    return null
```

```
// Go down the tree to the node we want to delete.
```

```
if value < node.value:
```

```
    node.left = delete_internal(node.left, value)
```

```
else if value > node.value:
```

```
    node.right = delete_internal(node.right, value)
```

```
// Now we have found the node to delete.
```

```
else if node.left is null:
```

```
    node = node.right
```

```
else if node.right is null:
```

```
    node = node.left
```

```
// The node to delete has two children.
```

```
else:
```

```
    node.left = delete_rightmost(node, node.left)
```

```
return node
```

```
// Delete the rightmost descendant, using its value for node.
```

**method** delete\_rightmost(*node* : Node, *descendant* : Node) → Node:

```
if descendant.right is null:
```

```
    node.value = descendant.value
```

```
return null
```

```
descendant.right = delete_rightmost(node, descendant.right)
```

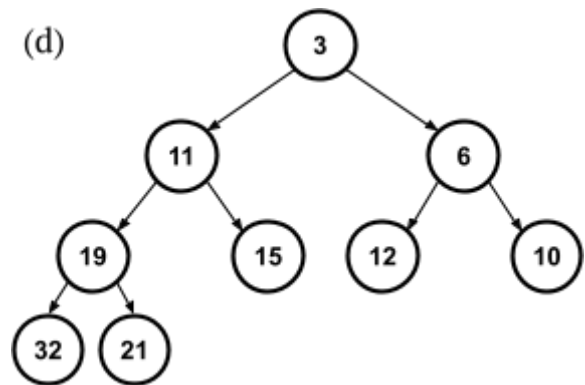
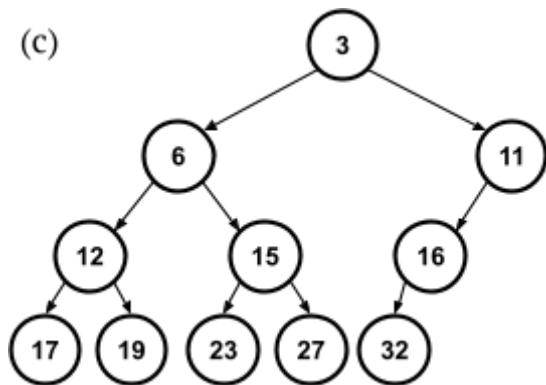
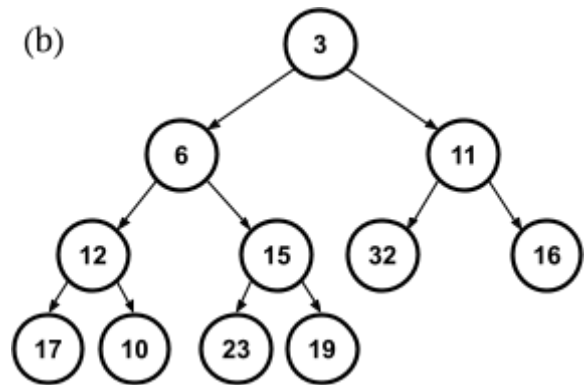
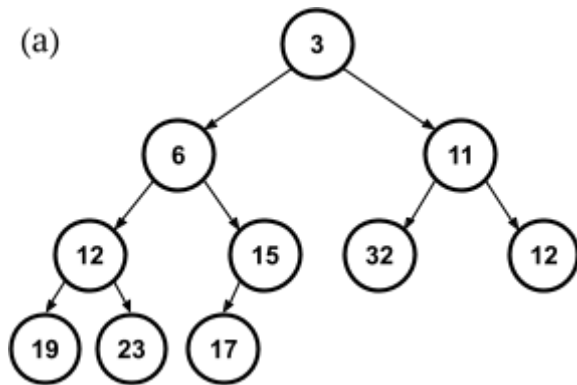
```
return descendant
```

*Note: This is just one possible implementation! There are many different variants, recursive or imperative.*

## Section 4: Binary heaps

### Question 4A

Which of the following trees represent a binary min-heap?



For each of the trees:

- if it is not a binary heap, explain why not – i.e. point out a concrete specific problem with it,
- if it is a binary heap, write it in array notation.

### Answer:

- a) This is a binary heap and the array looks like: [3, 6, 11, 12, 15, 32, 12, 19, 23, 17].
- b) This is not a binary heap: the node 10 is smaller than its parent 12.
- c) This is not a binary heap: it is not a complete tree.
- d) This is a binary heap and the array looks like: [3, 11, 6, 19, 15, 12, 10, 32, 21].

## Question 4B

- a) Explain how to represent a binary min-heap as an array. Make sure to describe all invariant(s).
- b) Give pseudocode for inserting a new element into a heap represented as a dynamic array.

### Answer:

- a) With the 0-based array indexing convention, an array  $x$  represents a heap exactly if for every valid array index  $i$ , we have  $x[(i - 1) / 2 \text{ (rounded down)}] \leq x[i]$ .
- b) Call the dynamic array  $x$  and the element to be inserted  $a$ . Then we do:

```
 $i = \text{size}(x)$ 
 $x.\text{append}(a)$ 
loop:
     $j = (i - 1) / 2 \text{ (rounded down)}$ 
    if (not  $j \geq 0$ ) or  $x[j] \leq x[i]$ :
        break out of loop
    swap positions  $j$  and  $i$  of  $x$ 
     $i = j$ 
```

# Section 5: Hash tables

## Question 5A

For each of the following questions, select the correct answer (1, 2 or 3).

Assume that you don't know anything at all about the hash function except that it takes constant time to calculate hash values.

The correct answers are coloured red:

- a) What is the worst-case asymptotic time complexity, in terms of  $n$ , of adding one element to a separate-chaining hash table containing  $n$  elements?
1. Constant,  $O(1)$
  2. Logarithmic,  $O(\log n)$
  3. **Correct:** Linear,  $O(n)$  — because the hash function could be very bad.
- b) What is the most efficient way of resizing an open-addressing hash table that is close to full?
1. **Correct:** Create a new table, roughly twice as large. Add each element from the old table to the new table, using normal hash table insertion.
  2. Create a new table, roughly twice as large. Add each element from the old table to the cell that is indicated by the new hash value.
  3. Create a new table, roughly twice as large. Since the hash values are stable, you can simply put the elements in the same positions as they were in the old table.
- c) Which expression most accurately describes the worst-case time complexity of searching for an element in a separate-chaining hash table, with table size  $m$ , containing  $n$  elements, and buckets as linked lists of size at most  $k$ ?
1. Linear in the table size,  $O(m)$
  2. Linear in the number of elements,  $O(n)$
  3. **Correct:** Linear in the bucket size,  $O(k)$
- d) How can you delete an element  $d$  in an open-addressing hash table using linear probing?
1. Find the element  $d$ , and clear the cell.
  2. **Correct:** Find the element  $d$ , and set the value of the cell to a special “deleted” value — this is the idea with “lazy deletion”.
  3. Find the element  $d$ , then continue searching for the last element  $x$  in the cluster. Move  $x$  to the cell where  $d$  is (thus deleting  $d$ ), and then clear the old  $x$  cell.

## Question 5B

In Python, dictionaries (i.e. key-value maps) are implemented as open-addressing hash tables. Prior to version 3.6 dictionaries were implemented in one big table of key-value pairs:<sup>1</sup>

	pointer to key	pointer to value
0	—	—
1	→ “grass”	→ “green”
2	—	—
3	—	—
4	—	—
5	→ “sun”	→ “yellow”
6	—	—
7	→ “sky”	→ “blue”

On a standard 64-bit computer, every pointer and hashcode takes 64 bits, which is 8 bytes. So each row in the table uses 16 bytes.

But the table will always contain quite many empty rows (in this case 5 of them), which means that it should be possible to optimise. Raymond Hettinger (a Python guru) came up with a solution that is the standard implementation since Python 3.6. The hash table is split into two arrays, one single integer array of indices and one compact array of the same kind of key-value pairs as before, where the indices in the first array give the positions of key-value pairs in the second array:

	0	1	2	3	4	5	6	7
index	—	1	—	—	—	0	—	2

	pointer to key	pointer to value
0	→ “sun”	→ “yellow”
1	→ “grass”	→ “green”
2	→ “sky”	→ “blue”

This implementation can save quite a lot of memory, depending on the load factor.<sup>2</sup>

*(Continued on next page)*

<sup>1</sup> Actually, Python also stores the 64-bit hash value in the table to optimise searching and rehashing, so this description is a bit simplified. But you don’t have to care about that now.

<sup>2</sup> And usually we don’t need 64-bit integers for the index cells. E.g., if the index array has size  $\leq 2^{16}$  we can make do with 16-bit integers, which saves even more space. But you don’t have to care about that now.

## Question 5B (continued)

Here is an incomplete implementation of a key-value map using the hash table design just described. Your job is to implement the *get* method.

Here are some notes about the implementation:

- The integer array of indices is stored in the field called *indices*. The table of key/value pairs is split into two arrays called *keys* and *values*. Using the example on the previous page, *indices*[5] would contain 0, *keys*[0] would contain “sun” and *values*[0] would contain “yellow”.
- The hash table uses linear probing to handle collisions.
- The hash table uses *lazy deletion*. When a key is deleted, the *indices* array is not changed, but the entries in the *keys* and *values* array are set to *null*.<sup>3</sup>

Make sure that your *get* method handles deleted keys correctly.

```
class HashMap[Key, Value]:
```

```
  indices : Array[int]
```

```
  keys    : Array[Key]
```

```
  values  : Array[Value]
```

```
  method get(key : Key) → Value:
```

```
    // Look up the given key in the hash table, returning a value.
```

```
    // Returns null (or None) if the key is not present in the hash table.
```

```
    h = hash(key) mod length(indices)
```

```
    while true:
```

```
      k = indices[h]
```

```
      if keys[k] is null:
```

```
        return null
```

```
      else if key == keys[k]:
```

```
        return values[k]
```

```
      h = (h + 1) mod length(indices)
```

---

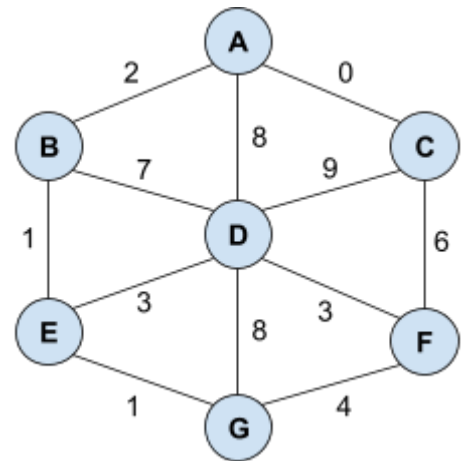
<sup>3</sup> For Python programmers, *null* is the same as *None*.

## Section 6: Graphs

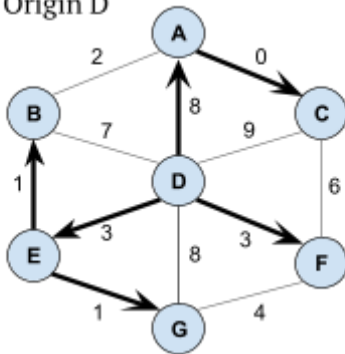
### Question 6A

Consider the weighted undirected graph to the right.

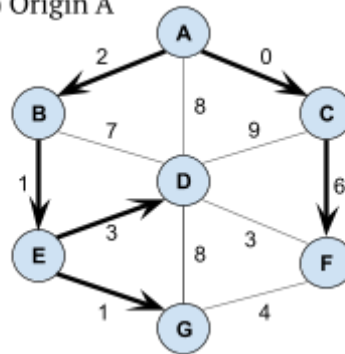
For each of the below subgraphs (indicated in bold), say if it is a **shortest path tree** starting at the **indicated origin**. If your answer is negative, justify it by pointing out a concrete problem.



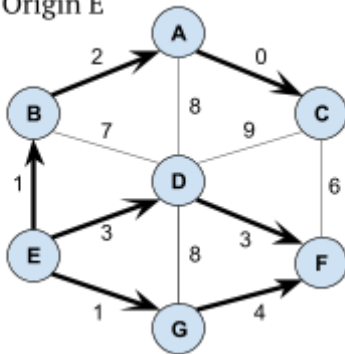
(a) Origin D



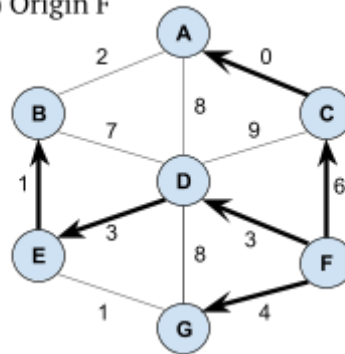
(b) Origin A



(c) Origin E



(d) Origin F



**Answer:**

- a) is not a SPT: the shortest path from D to A is D-E-B-A and not the direct edge D-A
- b) is a SPT
- c) is not a SPT: it is not even a tree since there are two paths from E to F
- d) is not a SPT: the shortest path from F to E is via G and not via D

## Question 6B

Describe an algorithm (in pseudocode or your second-favourite programming language) that detects if a directed graph contains a cycle.

Assume that the graph is given in adjacency list representation:

- you can iterate over each vertex (e.g. **for each vertex  $v$** ),
- for each vertex  $v$ , you can iterate over each outgoing edge (e.g. **for each edge  $e$  from  $v$** ).

Your algorithm should be reasonably efficient, close to  $O(V + E)$  in terms of  $V$  vertices and  $E$  edges. You can use any data structure from the course.

### Answer:

Here's an efficient solution. We use a variant of depth-first search that processes every vertex only the first time it is visited. With every vertex  $v$ , we store two Booleans flags (both initially false):

- **ancestor**: does  $v$  appear as an ancestor with respect to the current position in the graph?
- **visited**: have we already processed  $v$ ?

**function** has\_cycle()  $\rightarrow$  bool:

```
    unmark all vertices (as both visited and ancestor)
    for each vertex  $v$ :
        if vertex_has_cycle( $v$ ):
            return true
    return false
```

**function** vertex\_has\_cycle( $v$  : Vertex)  $\rightarrow$  bool:

```
    if  $v$  is marked as ancestor:
        return true
    if  $v$  is not marked as visited:
        mark  $v$  as visited
        mark  $v$  as ancestor
        for each edge  $e$  from  $v$ :
            if vertex_has_cycle(target of  $e$ ):
                return true
        unmark  $v$  as ancestor
    return false
```

We call vertex\_has\_cycle( $v$ ) for every vertex  $v$ . If this does not report a cycle, then no cycle exists.

We now check (not required) that this algorithm is  $O(V + E)$ . For every vertex  $v$ , the for-loop in the function vertex\_has\_cycle is executed only once. Thus, the loop body is executed only once for every edge. This also means that the visit function is called at most  $V + E$  many times ( $E$  from this loop body and  $V$  from the outer loop). The remaining statements in an invocation of vertex\_has\_cycle are constant time.