Exam, DAT038/TDA417/DAT525
(and reexam for DAT037/TDA416/LET375)
Datastrukturer och algoritmer

Thursday, 2022-01-13, 14:00–18:00

Teachers LET375: Pelle Evensen, 0732–060056
Other courses: Peter Ljunglöf, 0766–075561; Nick Smallbone, 0707–183062

Teachers will visit around 15:00 and 16:30

Allowed aids None

Exam review When the exams have been graded they are available for review in the student office on floor 4 in the EDIT building.
If you want to discuss the grading, please come to the exam review:

Monday, 31 January, 10–12 and 13–15, in room 6128 (EDIT building 6th floor)

In that case, you should leave the exam in the student office until after the review. We will bring all exams to the review meeting.

Notes Write your anonymous code (not your name) on every page.
You may answer in English or Swedish.
Excessively complicated answers might be rejected.
Write legibly – we need to be able to read your answer!
You can write explanations on the question sheet or on a separate paper.

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 ⩾1 correct answer	Sections with both answers correct
3	5	—
4	5	2
5	6	4

Good luck!

Section 1: Complexity

Question 1A

Suppose we want to maintain a stack *without duplicate elements*. In this stack, if we push an element that is already on the stack then we just ignore it. Note that the duplicate element does not necessarily have to be the first element on the stack, i.e., if one of the elements on the stack is equal to the element we are trying to push, we don't add it to the stack. The following pseudocode snippets implement such a stack using different data structures:

A. Using a linked list:

```
list = new linked list

push(x):  if not list.contains(x):
              list.addFirst(x)

pop():    return list.removeFirst()
```

B. Using a combination of a balanced binary search tree and a linked list:

```
set = new balanced tree set
list = new linked list

push(x):  if not set.contains(x):
              set.add(x)
              list.addFirst(x)

pop():    x = list.removeFirst()
          set.remove(x)
          return x
```

C. Using a combination of a hash table and a linked list:

```
set = new hash set
list = new linked list

push(x):  if not set.contains(x):
              set.add(x)
              list.addFirst(x)

pop():    x = list.removeFirst()
          set.remove(x)
          return x
```

Question 1A (continued)

Your task is to give the worst-case asymptotic complexity of the `push` and `pop` functions in terms of the size *n* of the stack for the three different solutions. You should express the asymptotic complexity in the simplest form possible. Assume that comparisons take constant time and that the hash table uses a good (constant-time) hash function.

Answer:

A. *Linked list*

`push:` _____ `pop:` _____

B. *Balanced BST + linked list*

`push:` _____ `pop:` _____

C. *Hash table + linked list*

`push:` _____ `pop:` _____

Brief explanation:

Question 1B

Consider the following function:

```
f(n):
    s = 1
    m = n
    while m >= 1:
        for i in 1, 2, ..., m:
            s = s + 1
        m = m / 2
    return s
```

Your task is to describe the asymptotic complexity of the above function `f` expressed in terms of its argument `n`. You should express the asymptotic complexity in the simplest form possible. The answer needs to be well motivated!

Note: the runtime of the inner loop depends on `m`, but your answer should depend only on `n`.

Answer: _____

Explanation:

Section 2: Using data structures

Question 2A

The following algorithm computes the median element of an array. In the pseudocode, `numbers` is the input to the algorithm, which you can assume to be an array of integers without duplicates. The pseudocode uses a data structure `X`, but does not specify what sort of data structure `X` should be:

```
X = new empty data structure
for j in numbers:  // numbers is an array of integers
    X.add(j)       // Assume that there are no duplicate numbers

while X.size() > 2:
    remove smallest element from X
    remove largest element from X

return smallest element of X
```

Which of the following data structure(s) would be a suitable data structure to use for `X`?

linked list	dynamic array	binary search tree
hash table	red-black tree	binary heap

Circle **all** correct answers – there may be several. Assume that:

- The algorithm should run in worst-case O(n log n) time (where n is the length of `numbers`).
- The input array `numbers` does not contain duplicates.

Brief explanation of why the data structure(s) you chose are suitable (you do not need to explain why the other ones are unsuitable):

Question 2B

In a tree, the *level* of a node is defined as follows: the root of the tree is at level 0, the children of the root are at level 1, the children of level 1 nodes are at level 2, and so on.

A *min-max heap* is a binary tree where:

- Every node contains a value.
- If a node's level is *even*, its value is *less than or equal to* the values of its descendants.
- If a node's level is *odd*, its value is *greater than or equal to* the values of its descendants.

(A node's descendants are its children, grandchildren, great-grandchildren and so on.)

In a min-max heap, which nodes do you need to search to find the *minimum* value stored in the tree? If you need to search several nodes, list all of them. (You can either describe in words which nodes to search, or use notation such as `root.left`.)

Answer: _____

_____

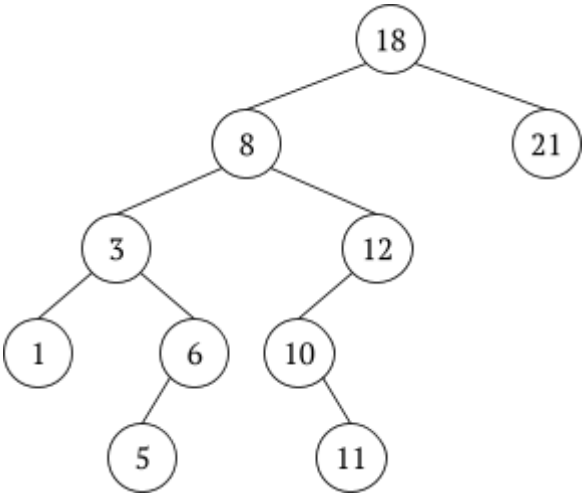
And which nodes do you need to search to find the *maximum* value stored in the tree?
If you need to search several nodes, list all of them.

Answer: _____

_____

Optional explanation:

Section 3: Search trees



Question 3A

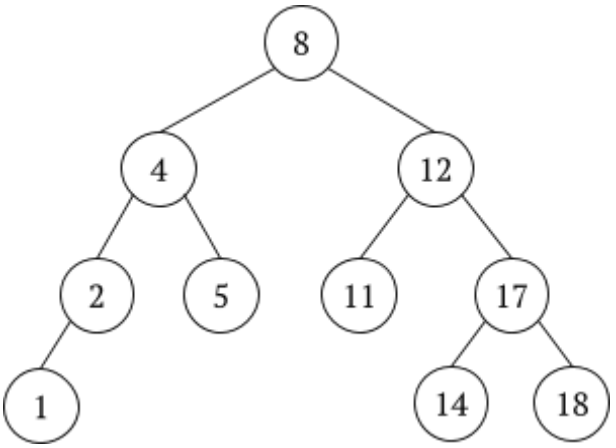
Delete **8** from the binary search tree on the right, using the BST deletion algorithm. How does the tree look afterwards?

Answer:

Brief explanation of how you did the deletion:

Question 3B

Add **15** to the AVL tree on the right, using the AVL insertion algorithm. How does the tree look afterwards?



Answer:

Brief explanation of how you did the insertion:

Section 4: Queues and priority queues

Question 4A

Assume you have a circular array queue which looks like this:

0	1	2	3	4	5	6	7	8	9	10	11
							the	*cat*	*sat*	*on*	

↑ `front` ↑ `rear`

What does the queue look like after performing the following operations?

```
enqueue("the") ; dequeue() ; enqueue("mat") ; dequeue() ; dequeue()
```

Don't forget to show where `front` and `rear` point afterwards.

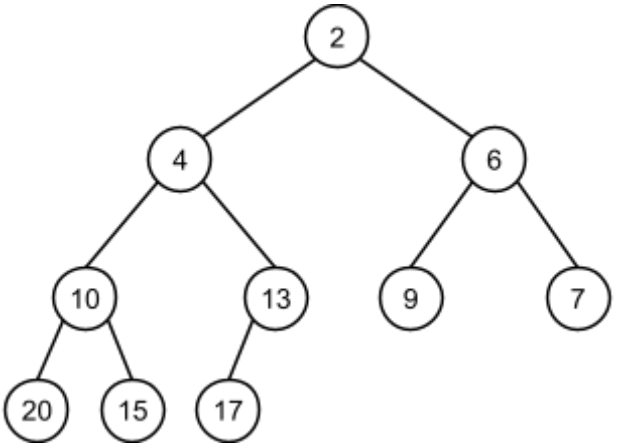
Answer:

0	1	2	3	4	5	6	7	8	9	10	11

Brief explanation:

Question 4B

Assume you have a minimum priority queue implemented as the binary heap to the right. What does this heap look like, in the standard array implementation?



Answer:

0	1	2	3	4	5	6	7	8	9	10	11

Now, remove the minimum element from the heap. What does the array look like after that?

Answer:

0	1	2	3	4	5	6	7	8	9	10	11

Brief explanation of how you removed the minimum:

Section 5: Sorting

Question 5A

Suppose that we sort the following array in ascending order using the standard merge sort algorithm (also known as top-down merge sort).

```
[13 7 98 45 97 30 71 9]
```

In the table below, fill in the *comparisons* that the merge sort algorithm does while sorting the array, in the correct order. For example, the algorithm starts by comparing 13 against 7, so we write 13 in the *first value* column and 7 in the *second value* column. In the recursive calls to merge sort, sort the left part of the array before the right half.

Step	First value	Second value	Step	First value	Second value
1	`13`	`7`	**9**		
2			**10**		
3			**11**		
4			**12**		
5			**13**		
6			**14**		
7			**15**		
8			**16**		

Optional explanation:

Question 5B

Assume that you use quicksort to sort an array with 6 elements, and use the first element as pivot. How many pairwise comparisons will be made…

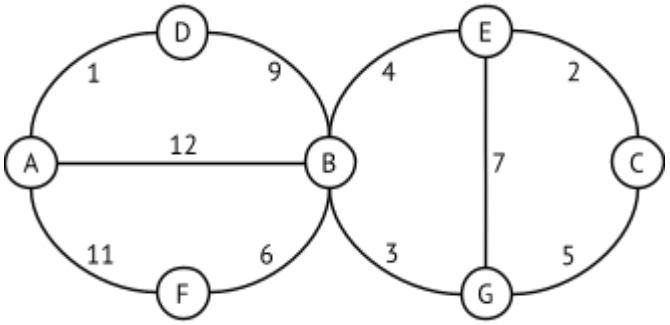
a) …in the worst case? *Answer:* _____

b) …in the best case? *Answer:* _____

Motivate your answers clearly.

Explanation:

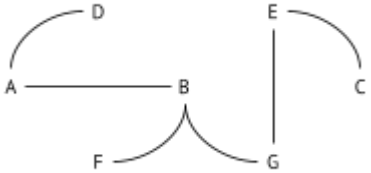
Section 6: Graphs



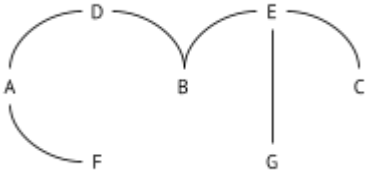
Question 6A

Which of the following denotes the minimum spanning tree (MST) of the graph on the right?

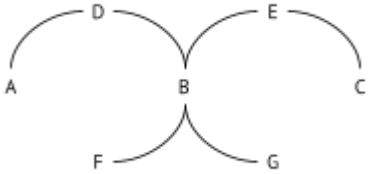
1



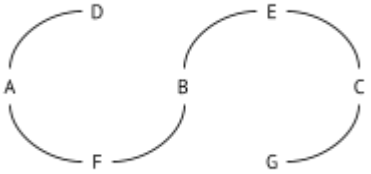
2



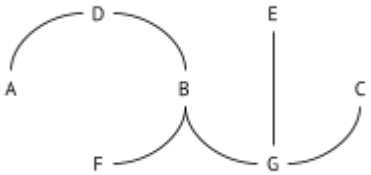
3



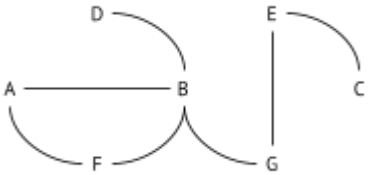
4



5



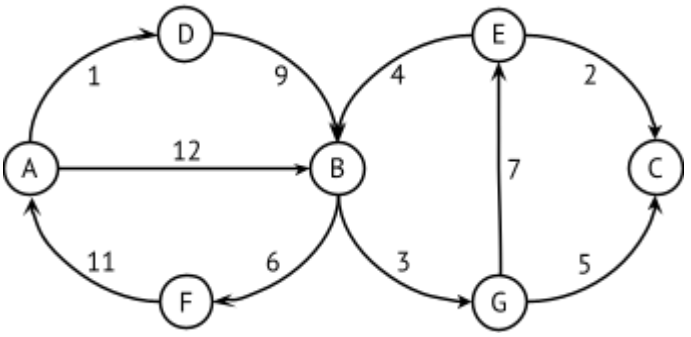
6



Answer: _____

Brief explanation:

Question 6B



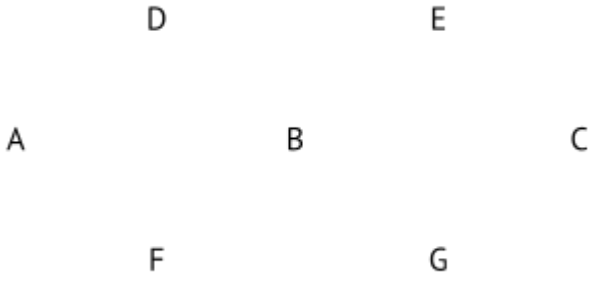
Perform Dijkstra's algorithm (also known as *uniform cost search*) on the directed graph to the right, starting in node **A**. Show each step of the algorithm: which node is removed, which node(s) are added to the priority queue, and how the priority queue looks like after each iteration. Also draw the final *shortest path tree* (SPT).

Write the priority queue like this: "`X:4, Y:6, Z:8`", where the numbers are the priorities.

Answer:

Removed node	**Added node(s)**	**Priority queue after adding new nodes**
–	`A`	`A:0`

Shortest path tree from A:



Exam, DAT038/TDA417/DAT525

(and reexam for DAT037/TDA416/LET375)

Datastrukturer och algoritmer – solution suggestions

Thursday, 2022-01-13, 14:00–18:00

Teachers LET375: Pelle Evensen, 0732–060056
Other courses: Peter Ljunglöf, 0766–075561; Nick Smallbone, 0707–183062

Teachers will visit around 15:00 and 16:30

Allowed aids None

Exam review When the exams have been graded they are available for review in the student office on floor 4 in the EDIT building.
If you want to discuss the grading, please come to the exam review:

Monday, 31 January, 10–12 and 13–15, in room 6128 (EDIT building 6th floor)

In that case, you should leave the exam in the student office until after the review. We will bring all exams to the review meeting.

Notes Write your anonymous code (not your name) on every page.
You may answer in English or Swedish.
Excessively complicated answers might be rejected.
Write legibly – we need to be able to read your answer!
You can write explanations on the question sheet or on a separate paper.

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 ⩾1 correct answer	Sections with both answers correct
3	5	—
4	5	2
5	6	4

Good luck!

Section 1: Complexity

Question 1A

Suppose we want to maintain a stack *without duplicate elements*. In this stack, if we push an element that is already on the stack then we just ignore it. Note that the duplicate element does not necessarily have to be the first element on the stack, i.e., if one of the elements on the stack is equal to the element we are trying to push, we don't add it to the stack. The following pseudocode snippets implement such a stack using different data structures:

A. Using a linked list:

```
list = new linked list

push(x):  if not list.contains(x):
              list.addFirst(x)

pop():    return list.removeFirst()
```

B. Using a combination of a balanced binary search tree and a linked list:

```
set = new balanced tree set
list = new linked list

push(x):  if not set.contains(x):
              set.add(x)
              list.addFirst(x)

pop():    x = list.removeFirst()
          set.remove(x)
          return x
```

C. Using a combination of a hash table and a linked list:

```
set = new hash set
list = new linked list

push(x):  if not set.contains(x):
              set.add(x)
              list.addFirst(x)

pop():    x = list.removeFirst()
          set.remove(x)
          return x
```

Question 1A (continued)

Your task is to give the worst-case asymptotic complexity of the `push` and `pop` functions in terms of the size n of the stack for the three different solutions. You should express the asymptotic complexity in the simplest form possible. Assume that comparisons take constant time and that the hash table uses a good (constant-time) hash function.

Answer:

A. *Linked list*

`push:` O(n) `pop:` O(1)

B. *Balanced BST + linked list*

`push:` O(log n) `pop:` O(log n)

C. *Hash table + linked list*

`push:` O(1), but see below `pop:` O(1), but see below

Brief explanation:

- *Linked list*: `list.contains(x)` takes O(n) time, but `addFirst` and `removeFirst` take O(1) time.
- *Balanced BST + linked list*: `set.contains/add/remove` all take O(log n) time.
- *Hash table + linked list*: `set.contains/add/remove` take amortised O(1) expected time, because the hash function is of good quality. Although the internal array sometimes has to be resized, it will happen so rarely that it will be on average constant time in the long run, and therefore the amortised time is O(1).

  However, the "un-amortised" complexity is O(n) if the internal has to be resized. This answer is also accepted, if the explanation is clear that this happens because of resizing.

Question 1B

Consider the following function:

```
f(n):
    s = 1
    m = n
    while m >= 1:
        for i in 1, 2, ..., m:
            s = s + 1
        m = m / 2
    return s
```

Your task is to describe the asymptotic complexity of the above function `f` expressed in terms of its argument `n`. You should express the asymptotic complexity in the simplest form possible. The answer needs to be well motivated!

Note: the runtime of the inner loop depends on `m`, but your answer should depend only on `n`.

Answer: $O(n)$

Explanation:

The outer loop runs the following iterations:

- first with $m = n$,
- then with $m = n/2$,
- then with $m = n/4$,
- ...,
- and so on until $m = 1$.

Each time, the inner loop runs m times. So the inner loop runs in total

$$n + n/2 + n/4 + \ldots + 1 \leqslant n\,(1 + ½ + ¼ + ⅛ + \ldots) = n \cdot 2 = O(n)$$

Section 2: Using data structures

Question 2A

The following algorithm computes the median element of an array. In the pseudocode, `numbers` is the input to the algorithm, which you can assume to be an array of integers without duplicates. The pseudocode uses a data structure `X`, but does not specify what sort of data structure `X` should be:

```
X = new empty data structure
for j in numbers:  // numbers is an array of integers
    X.add(j)       // Assume that there are no duplicate numbers

while X.size() > 2:
    remove smallest element from X
    remove largest element from X

return smallest element of X
```

Which of the following data structure(s) would be a suitable data structure to use for `X`?

linked list	dynamic array	binary search tree
hash table	red-black tree	binary heap

Circle **all** correct answers – there may be several. Assume that:

- The algorithm should run in worst-case O(n log n) time (where n is the length of `numbers`).
- The input array `numbers` does not contain duplicates.

Brief explanation of why the data structure(s) you chose are suitable (you do not need to explain why the other ones are unsuitable):

In a red-black tree, you can add items, remove items, and find the smallest/largest in O(log n) time, giving a total runtime of O(n log n).

The other data structures are not suitable because:

- A linked list or an array is not suitable because finding the smallest/largest takes O(n) time.
- A BST is not suitable because if it becomes unbalanced the runtime will be O(n^2).
- A hash table is not suitable because finding the smallest/largest takes O(n) time.
- A min-heap or max-heap is not suitable since we need to find both the smallest and largest.

Question 2B

In a tree, the *level* of a node is defined as follows: the root of the tree is at level 0, the children of the root are at level 1, the children of level 1 nodes are at level 2, and so on.

A *min-max heap* is a binary tree where:

- Every node contains a value.
- If a node's level is *even*, its value is *less than or equal to* the values of its descendants.
- If a node's level is *odd*, its value is *greater than or equal to* the values of its descendants.

(A node's descendants are its children, grandchildren, great-grandchildren and so on.)

In a min-max heap, which nodes do you need to search to find the *minimum* value stored in the tree? If you need to search several nodes, list all of them. (You can either describe in words which nodes to search, or use notation such as `root.left`.)

Answer: The `root` is always the minimum.

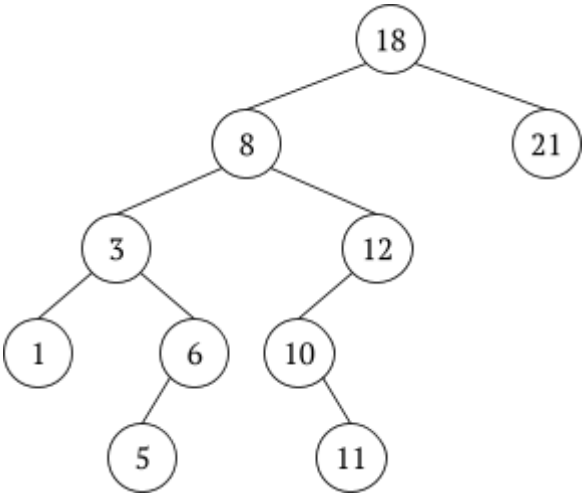
And which nodes do you need to search to find the *maximum* value stored in the tree?
If you need to search several nodes, list all of them.

Answer: The maximum is either `root.left` or `root.right`.
(Optional extra: if the heap has only one element, then the maximum is the `root`.)

Optional explanation:

Since the `root` has level 0, it is less than or equal to its descendants, i.e. all other values in the tree, which makes it the minimum. Since `root.left` and `root.right` have level 1, `root.left` is the largest value in the left subtree and `root.right` is the largest value in the right subtree. One of them must therefore be the largest value in the whole tree.

Section 3: Search trees

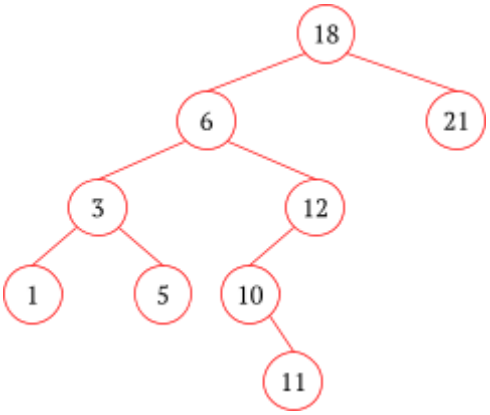


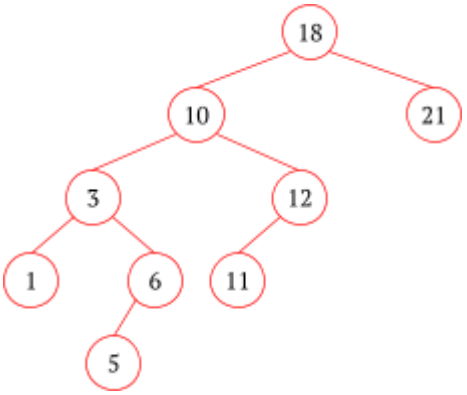
Question 3A

Delete **8** from the binary search tree on the right, using the BST deletion algorithm. How does the tree look afterwards?

Answer:

You can answer with either of the following trees:





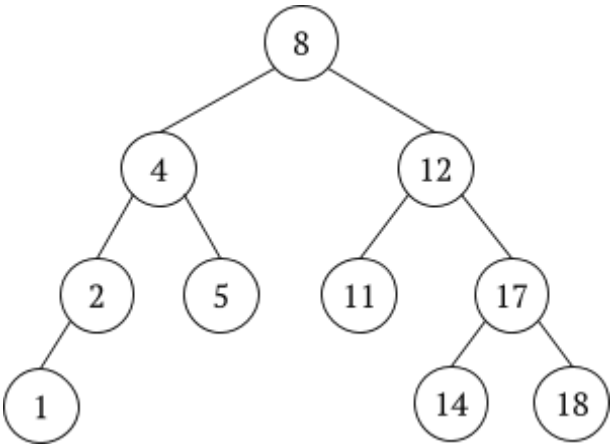
Brief explanation of how you did the deletion:

If you answered with the tree on the left:
Replace 8 with 6 (the largest value in the left subtree) then replace 6 with 5 (its only child).

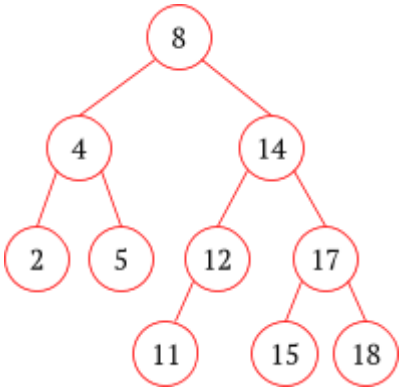
If you answered with the tree on the right:
Replace 8 with 10 (the smallest value in the right subtree) then replace 10 with 11.

Question 3B

Add **15** to the AVL tree on the right, using the AVL insertion algorithm. How does the tree look afterwards?



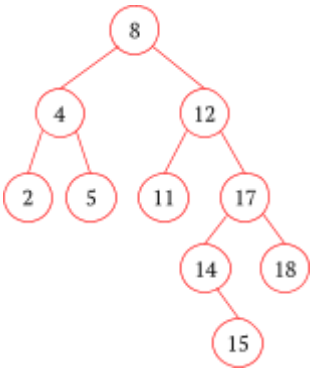
Answer:



Brief explanation of how you did the insertion:

Inserting 15 unbalances node 12. It's a right-left case, so rotate node 17 right (bringing 14 up), then rotate 14 left (bringing 14 up again).

Here's how the tree looks like before rebalancing:



Section 4: Queues and priority queues

Question 4A

Assume you have a circular array queue which looks like this:

0	1	2	3	4	5	6	7	8	9	10	11
							the	*cat*	*sat*	*on*	

↑ front ↑ rear

What does the queue look like after performing the following operations?

```
enqueue("the") ; dequeue() ; enqueue("mat") ; dequeue() ; dequeue()
```

Don't forget to show where `front` and `rear` point afterwards.

Answer:

0	1	2	3	4	5	6	7	8	9	10	11
mat							~~*the*~~	~~*cat*~~	~~*sat*~~	*on*	*the*

↑ rear ↑ front

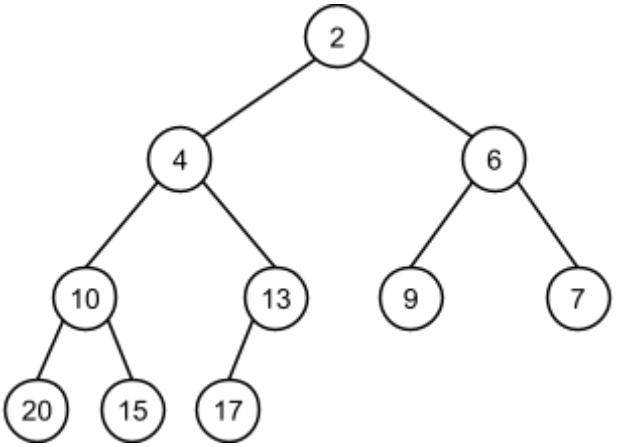
(Crossed-out cells can also be empty)

Brief explanation:

Enqueue moves "`rear`" on by 1, dequeue moves "`front`" on by 1.

Question 4B

Assume you have a minimum priority queue implemented as the binary heap to the right. What does this heap look like, in the standard array implementation?



Answer:

Note, you can also use a 1-based array i.e. have element 0 empty and start the tree at element 1:

0	1	2	3	4	5	6	7	8	9	10	11
2	4	6	10	13	9	7	20	15	17		

Now, remove the minimum element from the heap. What does the array look like after that?

Answer:

0	1	2	3	4	5	6	7	8	9	10	11
4	10	6	15	13	9	7	20	17			

Brief explanation of how you removed the minimum:

Move 17 to the root. Then 17 swaps with 4, then 10, then 15.

Section 5: Sorting

Question 5A

Suppose that we sort the following array in ascending order using the standard merge sort algorithm (also known as top-down merge sort).

```
[13 7 98 45 97 30 71 9]
```

In the table below, fill in the *comparisons* that the merge sort algorithm does while sorting the array, in the correct order. For example, the algorithm starts by comparing 13 against 7, so we write 13 in the *first value* column and 7 in the *second value* column. In the recursive calls to merge sort, sort the left part of the array before the right half.

Step	First value	Second value	Step	First value	Second value
1	13	7	**9**	97	71
2	45	98	**10**	7	9
3	7	45	**11**	13	9
4	13	45	**12**	13	30
5	97	30	**13**	45	30
6	71	9	**14**	45	71
7	30	9	**15**	98	71
8	30	71	**16**	98	97

Optional explanation:

```
[13] + [7] -> [7 13]                    13<7
[98] + [45] -> [45 98]                  98<45
[7 13] + [45 98] -> [7 13 45 98]        7<45, 13<45
[97] + [30] -> [30 97]                  97<30
[71] + [9] -> [9 71]                    71<9
[30 97] + [9 71] -> [9 30 71 97]        30<9, 30<71, 97<71
[7 13 45 98] + [9 30 71 97] -> [7 9 13 30 45 71 97 98]
                      7<9, 13<9, 13<30, 45<30, 45<71, 98<71, 98<97
```

Question 5B

Assume that you use quicksort to sort an array with 6 elements, and use the first element as pivot. How many pairwise comparisons will be made…

a) …in the worst case? *Answer:* **15** (5+4+3+2+1) and **20** (6+5+4+3+2) are accepted

b) …in the best case? *Answer:* **8** (5+2+1), **10** (6+3+1) and **11** (6+3+2) are accepted

Motivate your answers clearly.

Explanation:

There are three possible answers, all of which get a point:

1. If implemented carefully, partitioning an array of N elements does N-1 comparisons (each element gets compared with the pivot once).

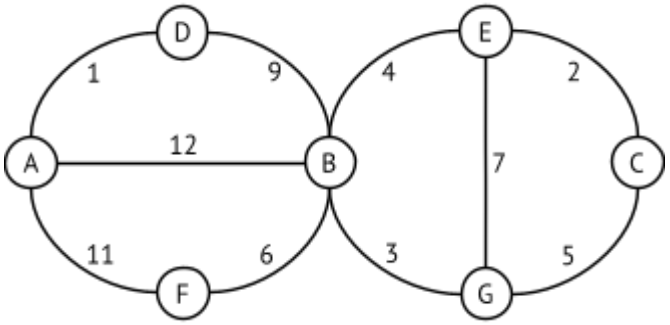
   In the worst case, quicksort makes recursive calls of size 5, 4, 3, 2 and 1, as well as the original call of size 6. This gives 5+4+3+2+1+0 = 15 comparisons.

   In the best case, the array gets split into arrays of size 3 and 2. In the recursive calls, the array of size 3 gets split (in the best case) into two arrays of size 1. In that case the number of comparisons is 5 + 2 (for the array of size 3) + 1 (for the array of size 2) = 8.

2. The partitioning algorithm from the course does one extra comparison: the element at the position where lo and hi "cross over" gets compared against the pivot twice, once when increasing lo and once when decreasing hi. This gives answers of 6+5+4+3+2=20 (worst case) and 6+3+2=11 (best case).

3. In fact, this extra comparison does not occur if the pivot is the smallest element, so the best case is actually 6+3+1=10 (this happens when, in the 2-element subarray, the pivot is the smallest element).

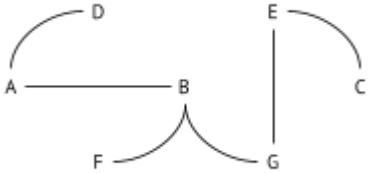
Section 6: Graphs



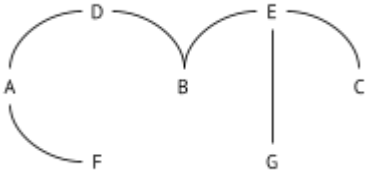
Question 6A

Which of the following denotes the minimum spanning tree (MST) of the graph on the right?

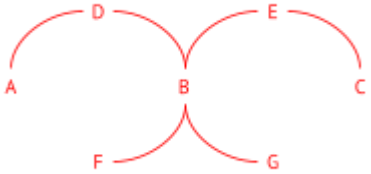
1



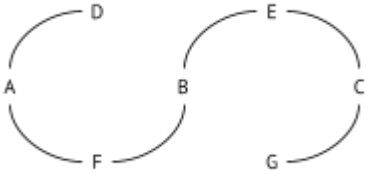
2



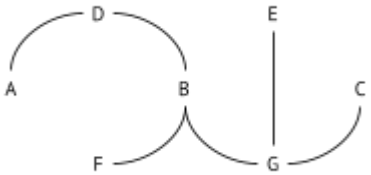
3



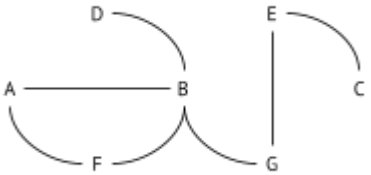
4



5



6

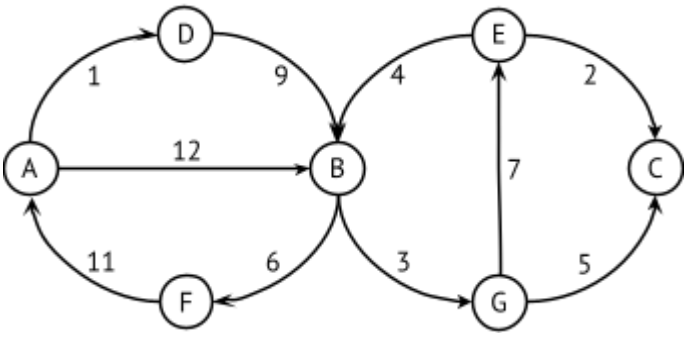


Answer: Option 3.

Brief explanation:

You can use Prim's or Kruskal's algorithm to find it.

Question 6B



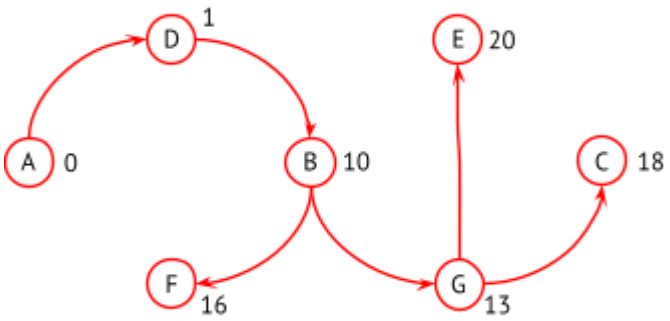
Perform Dijkstra's algorithm (also known as *uniform cost search*) on the directed graph to the right, starting in node **A**. Show each step of the algorithm: which node is removed, which node(s) are added to the priority queue, and how the priority queue looks like after each iteration. Also draw the final *shortest path tree* (SPT).

Write the priority queue like this: "`X:4, Y:6, Z:8`", where the numbers are the priorities.

Answer:

Note, this is the version of Dijkstra's algorithm where a node can appear multiple times in the priority queue, but we also accept the version where all nodes appear once and their weight gets updated:

Removed node	Added node(s)	Priority queue after adding new nodes
–	A	A:0
A	D B	D:1 B:12
D	B	B:10 B:12
B	G F	B:12 G:13 F:16
B	–	G:13 F:16
G	C E	F:16 C:18 E:20
F	A	C:18 E:20 A:27
C	–	E:20 A:27
E	C B	C:22 B:24 A:27
C	–	B:24 A:27
B	–	A:27

Shortest path tree from A:

(The total costs written beside each node are optional to write.)