Basic question 1: Complexity

Here is a simple sorting algorithm that works on linked lists:

```
function sort(xs : list of integers) → list of integers:
    queue = new maximum priority queue (using a binary heap)
    for each x in xs:
        add x to queue

    result = new linked list of integers
    while queue is not empty:
        remove the maximum element y from queue
        insert y at the front of result

    return result
```

What is the asymptotic complexity of this function in the number of elements N of `xs`?

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

Complexity: $O(N \log(N))$

Justification (you can also add notes directly in the code above):

First note the following. Since we add exactly the elements of the list to the heap, the heap has size at most N throughout the program. That allows us to bound the operations on the heap.

Each heap insertion takes $O(\log(N))$ time and we do N insertions, so this part takes $O(N \log(N))$ time in total.

Removing the maximum element from the heap is $O(\log(N))$. Inserting it at the front of the result list is $O(1)$. So the body of the while-loop is $O(\log(N))$. Since it is executed N times, it takes $O(N \log(N))$ time.

Summing up the two program parts, we get $O(N \log(N))$.

Basic question 2: Sorting

Perform a quicksort partitioning of the following array, using the element at position 3 as pivot:

0	1	2	3	4	5	6	7	8
15	74	63	51	32	48	89	91	27

Note: quicksorting the left and right parts of the partition is not part of this question.

Answer

Note: By “position 3” we meant “index 3”, i.e. the value 51. But if you thought we meant the value 63 it’s also fine because that’s the 3rd value in the array.

Write down how the array looks after the partitioning:

0	1	2	3	4	5	6	7	8
32	27	48	15	51	63	89	91	74

What sequence of swaps did you make when partitioning the array?

Using the algorithm from the lectures:

- swap pivot 51 with first element: **swap(3, 0)**
- initialize lo = 1 and hi = 8
- both lo and hi are stuck
- **swap(1, 8)**, lo = 2, hi = 7
- lo is stuck, hi advances to 5
- **swap(2, 5)**, lo = 3, hi = 4
- lo advances to 5, hi is stuck
- lo and hi have crossed, swap pivot with hi: **swap(0, 4)**

If you used a different algorithm from that of your course, explain it here:

Basic question 3: Basic data structures

We have a class Queue that is implemented as a circular array. Internally, it has an array `data` and these integer variables:

- `front`: index of the front of the queue,
- `rear`: index of the rear of the queue,
- `size`: number of elements in the queue.

The class has two public methods, enqueue and dequeue:

```
class Queue:
    method enqueue(item : int)
    method dequeue() → int
```

If we create a queue q with a capacity of six, add the four elements a, b, c, and d, and then remove one, the internal representation looks like this:

front = 1   rear = 4   size = 3

	0	1	2	3	4	5
data =		b	c	d		

Now we execute the following sequence of method calls:

```
q.dequeue()
q.enqueue(e)
q.enqueue(f)
q.dequeue()
q.enqueue(g)
```

How does the internal state of q look after these operations?

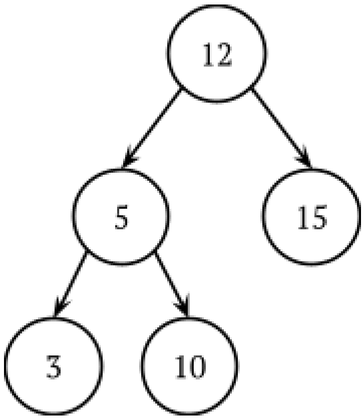
Answer

front = **3**   rear = **1**   size = **4**

	0	1	2	3	4	5
data =	**g**			**d**	**e**	**f**

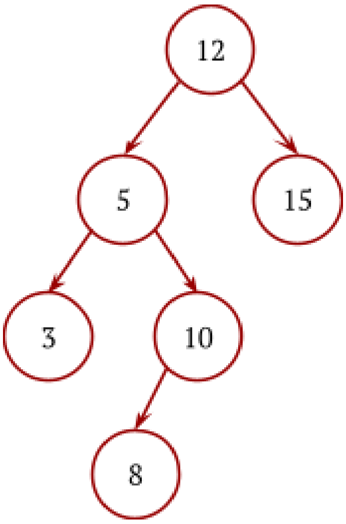
Basic question 4: Search trees

Insert 8 into the following AVL tree using the AVL insertion algorithm.



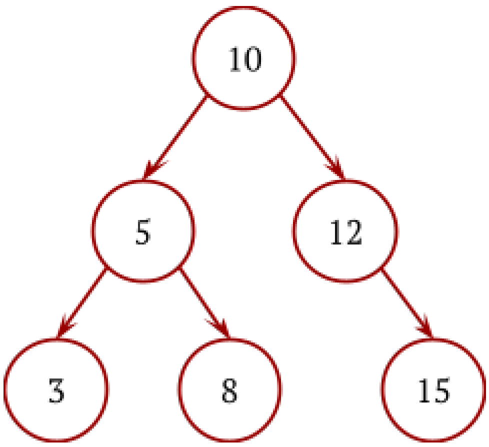
Answer

How does the tree look after you have inserted 8, but before rebalancing?



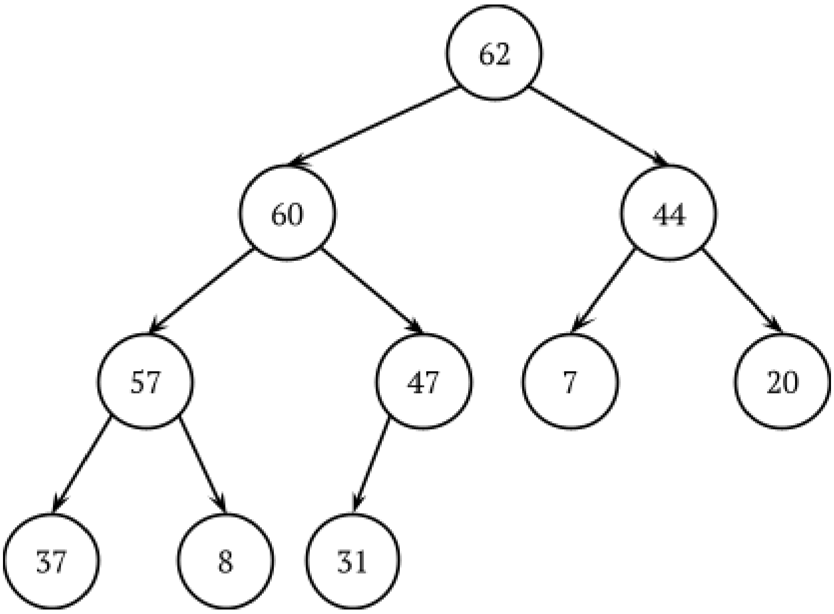
Node 12 is unbalanced (left-right case).

How does the tree look after rebalancing?



Basic question 5: Priority queues

You are given a maximum priority queue implemented as the following binary heap:



Answer

How does the above heap look when represented as an array?
You can choose the 0-based or 1-based version.

0	1	2	3	4	5	6	7	8	9	10
62	60	44	57	47	7	20	37	8	31	

Now remove the maximum element from the priority queue. How does the array look afterwards?

0	1	2	3	4	5	6	7	8	9	10
60	57	44	37	47	7	20	31	8		

Basic question 6: Hash tables

Suppose we have the following *open-addressing* hash table using *linear probing*.
Our hash function is the identity function *h(x) = x (mod 10)*.

0	1	2	3	4	5	6	7	8	9
8	1		23	14	4	5		18	9

The hash table was created by adding the elements 1, 4, 5, 8, 9, 14, 18, 23 **in an unknown order**.
The array has never been resized, and no elements have been deleted.
In what orders could the elements have been added?

A) 9, 8, 1, 14, 23, 4, 5, 18

B) 18, 23, 14, 1, 8, 4, 5, 9

C) 1, 14, 23, 18, 4, 9, 5, 8

D) 18, 14, 4, 1, 9, 8, 5, 23

E) 18, 9, 1, 23, 4, 5, 8, 14

Determine which of these orders are possible (there may be several, or even none).
For the others, explain why they are impossible.

Answer

Which orders are possible? only C and D

Explain why the others are impossible:

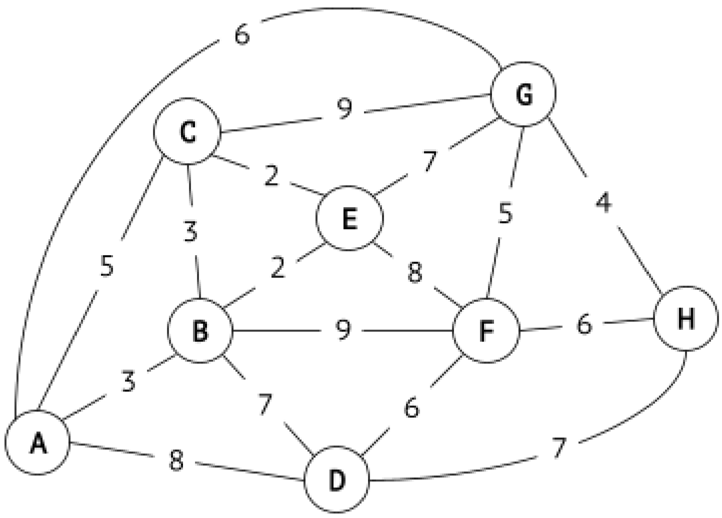
We look at each cluster and make the following observations:

- 8 is displaced by 18 and 9, so has to be inserted after them.
- 4 is displaced by 14, so has to be inserted after it.
- 5 is displaced by 4, so has to be inserted after it.

That eliminates A, B, E.

Basic question 7: Graphs

You are given the graph to the right.

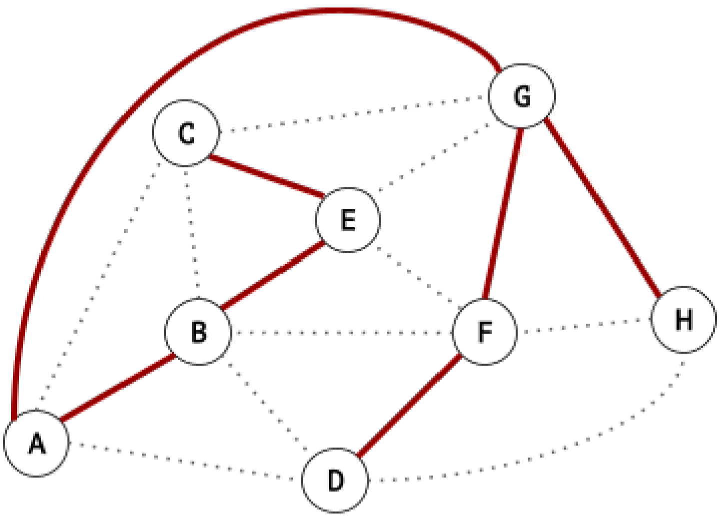


Draw the minimum spanning tree (MST).

Choose either Prim's or Kruskal's algorithm. In which order does the algorithm add the edges to the MST?

Answer

Draw the MST by filling the dotted edges:



Chosen algorithm (Prim or Kruskal): Kruskal (for example)

Starting vertex (only if needed): This is only needed in Prim's algorithm

MST edges added in order (not all cells have to be used):

1	BE or CE	4	GH	7	DF or AG
2	CE or BE	5	FG	8	
3	AB	6	AG or DF	9	

Basic question 8: Abstract data types

The following code implements a common abstract data type (ADT) using a common data structure.

```
class MysteriousDataStructure:
    a : Node

    method f() -> int:
        x = this.a.value
        this.a = this.a.next
        return x

    method g(y : int):
        n = new Node
        n.value = y
        n.next = this.a
        this.a = n

class Node:
    value : int
    next : Node
```

Answer

Which ADT is implemented here? A stack (implemented as a linked list)

Give the methods `f` and `g` better names:

`f =` pop

`g =` push

Advanced question 1: Complexity

The following is a very meaningless function F that takes an integer n as input:

```
function F(n):
    S = new set (using an AVL tree)
    for i from 0 to n:
        for j from 0 to n:
             S.add(i + j)

    for each a in S:
        for each b in S:
            doSomething(a, b)
```

Asymptotically, what is the **space complexity** and the **time complexity** of F in the input n? Assume that `doSomething` is O(1) in both time and space.

Write your answers in O-notation. Be as exact and simple as possible. Explain the key reasons why each complexity has the order of growth you give.

Part 1

Space complexity: $O(n)$

Time complexity: $O(n^2 \log(n))$

Justification: The crucial step is to establish a good bound on the size of the set S. This will also tell us the space complexity of the function as everything else is just using constant space. Adding two integers each ranging from 0 to n gives the integers from 0 to $2n-1$, which are $O(n)$ many.

Inserting into an AVL tree of size $O(n)$ takes $O(\log(n))$ time. With that, the first part of the program takes $O(n^2 \cdot \log(n))$ time. Knowing that the set S has $O(n)$ elements, the second part takes $O(n^2)$ time. Summing both parts up, the first part dominates.

Part 2

Suppose we replace `i + j` by `i*n + j` in the line where we add to S. What is your answer now?

Space complexity: $O(n^2)$

Time complexity: $O(n^4)$

Justification: The expression $i \cdot n + j$ with i and j ranging from 0 to n gives exactly the integers from 0 to n^2. So the set S has size $O(n^2)$. Inserting into an AVL tree of size $O(n^2)$ takes $O(\log(n^2)) = O(2 \log(n)) = O(\log(n))$ time. So the first part of the program takes $O(n^2 \log(n))$ time and the second part takes $O(n^2 \cdot n^2) = O(n^4)$ time. The second part dominates.

Advanced question 2: Hash tables

Here is an *incorrect* algorithm for deleting an item in a hash table using *linear probing/open addressing:*

- Find the position of the item to be deleted. Suppose it is at index *i* in the array.
- If the item is the *last* item in the cluster, just remove it (i.e., put *null/None* there).
- Otherwise, find the last item in the cluster and move it to index *i*.

For example, suppose we have the following hash table, using the hash function *h(x) = x (mod 10)*:

0	1	2	3	4	5	6	7	8	9
28	38		3		5	15		8	18

The original exam said 18 which was a typo.

If we delete 8, it will move 38 from position 1 (the end of the cluster) to position 8, producing:

0	1	2	3	4	5	6	7	8	9
28			3		5	15		38	18

Or if we delete 3, the 3 will be replaced by a *null/None*.

This algorithm is incorrect. After deleting an item, the hash table may no longer be valid. Your task is to find an example where the algorithm goes wrong. That means:

- You start with a valid hash table
- You delete an item using the algorithm above
- Afterwards, an item is stored in an incorrect position

Once you find an example, you can ***write your answer on the next page***.

Advanced question 2, answer

There are plenty of possible answers, below is just the simplest one. Any cluster where the final item is already in its correct location would do. (And there are numerous other answers too).

Which (valid) hash table do you start from?
Use the hash function *h(x) = x (mod 10)*.

0	1	2	3	4	5	6	7	8	9
0	**1**								

Which item do you delete? **0**

How does the hash table look afterwards?

0	1	2	3	4	5	6	7	8	9
1									

Which item is now in the wrong place? **1**

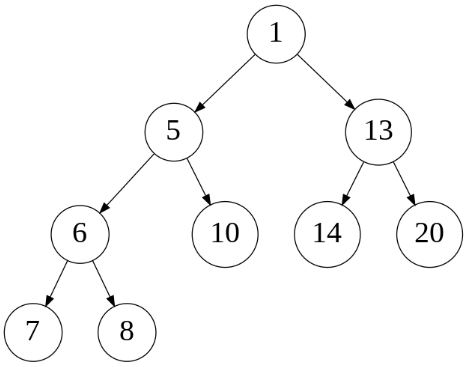
Advanced question 3: Strange trees

A *strange tree* is a data structure storing a set of keys. It is a binary tree, but not a binary search tree.

A strange tree is either *null/None* or has a key and left and right children satisfying the following:

- The key is *less than* the keys in its subtrees (i.e., a strict version of the heap property holds).
- All the keys in the left subtree are *less than* all the keys in the right subtree.

Here is an example of a strange tree (pointers to *null/None* nodes are omitted):



Your task is to write a function:

```
function contains(node : StrangeTree, key : int) → bool
```

which *searches* for a key in a strange tree. It should return *true* if the key is found or *false* otherwise.

You may assume that the `StrangeTree` class looks like this:

```
class StrangeTree:
    key   : int          // the key stored in this node
    left  : StrangeTree  // the left child
    right : StrangeTree  // the right child
```

Or, if you prefer Haskell:

```
data StrangeTree = Empty | Node Int StrangeTree StrangeTree
```

Note:

- Your solution should take *O(H)* time in terms of the height *H* of the tree.
- You may answer using pseudocode or Java/Python/Haskell.
- You can define helper functions if you want.

Write your answer on a separate sheet of paper.

Advanced question 3, answer

There are several possible solutions. The straightforward solution is this:

```
function contains(node : StrangeTree, key : int) → bool:
    if node is None:
        return False
    elif key < node.key: // optional
        return False     // optional
    elif key == node.key:
        return True
    elif node.right is not None and key >= node.right.key:
        return contains(node.right, key)
    else:
        return contains(node.left, key)
```

This solution ends up None-checking some nodes twice. To avoid this, we can extend the return type with an additional value to report in the negative case if the current subtree was the right place to look for the key (if so, we can stop looking elsewhere).

Solution using boolean exceptions

We can use bool-valued exceptions to signal a definite result (unit return type plus potential boolean exception are effectively three possible return values). We rely on exception propagation to simplify some control logic.

```
function raiseResult(node, key) [can raise bool]:
    if not (node is None and key ⩾ node.key):
        return
    raiseResult(node.left, key)
    raiseResult(node.right, key)
    raise (key = node.key)

function contains(node : StrangeTree, key : int) → bool:
    try:
        raiseResult(node, key)
    except v : bool:
        return v
    return False
```

Solution using the maybe-monad

Exceptional case(s) are inverted compared to the previous solution.

```
contains(root, key) := f(root) orElse False where
    f : StrangeTree → Maybe Bool
    f(node) := do
        guard (node ≠ None and key ⩾ node.key)
        asum [f(node.right), f(node.left), return (key = node.key)]
```

Advanced question 4: Data structure design

Two words are *anagrams* if you can rearrange the letters of one to get the other. Examples of anagrams are (1) *cat* and *act*; (2) *dare*, *dear* and *read*; (3) *wolves* and *vowels*; and (4) *algorithm* and *logarithm*.

Your task is to design a data structure for finding all anagrams of a given word. The data structure represents a collection of words. It should have two operations:

- `add(word : string)`: add a new word to the data structure
- `anagrams(word : string) → list[string]`: given a word, return a list of all words that are anagrams of it. The word itself should not be included in the list, only its anagrams.

Here is an example of using the data structure:

```
words = new anagram data structure
// add some words to the data structure
words.add("cat"); words.add("act")
words.add("dare"); words.add("dear"); words.add("read")
// now find some anagrams
words.anagrams("cat")    // should return the list ["act"]
words.anagrams("dear")   // should return the list ["dare", "read"]
words.anagrams("monkey") // should return an empty list []
```

The operations should have the following complexities (or better):

- add should take $O(\log(n))$ time where n is the size of the data structure
- anagrams should take $O(\log(n) + r)$ time where r is the number of anagrams returned

You can assume that the following function already exists – you don't need to implement it:

- `sortLetters(word : string) → string`: sort the letters of a string alphabetically. For example, `sortLetters("dear")` and `sortLetters("dare")` both return "`ader`".

Assume that all string operations (equality, comparison, `sortLetters`) take **constant time** $O(1)$.

You may use existing data structures and algorithms in your solution – you don't need to implement those yourself. You may answer using pseudocode or Java/Python/Haskell.

Write your answer on a separate sheet of paper.

Advanced question 4, answer

Here's one possible solution:

```
class Anagrams:
    words = new red-black tree
        (keys are strings, values are lists of strings)

    method add(word : string):
        key = sortLetters(word)
        if key not in words:
            words[key] = []
        words[key].append(word)

    method anagrams(word : string) -> list[string]:
        key = sortLetters(word)
        result = []
        if key in words:
            for anagram in words[key]:
                if word != anagram:
                    result.append(anagram)
        return result
```