Basic question 1: Complexity

Here is a simple function that takes two *linked lists* as input and returns a linked list that only contains elements that are present in *both* input lists:

```
function both(xs : list, ys : list) -> list:
    set = new empty set (using a self-balancing BST)
    res = new linked list

    for each x in xs:                       // The loop is O(N log(N))
        add x to set                        // Adding to set is O(log(N))

    for each y in ys:                       // The loop is O(N log(N))
        if set is empty:                    // This test is constant, O(1)
            return res

        if y in set:                        // Test membership in set is O(log(N))
            add y to the front of res       // Adding to the front is O(1)
            remove y from set               // Removing from set is O(log(N))

    return res
```

Assume that the input lists `xs` and `ys` have the same number of elements N. We want to know what the asymptotic worst-case time complexity of the function ’`both`‘ is in terms of the number of elements N. 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):

See the comments to the code. Note that `add` and `remove` and membership check for `set` are O(log(N)) because set contains O(N) elements throughout the program.

Basic question 2: Sorting

Perform a quicksort partitioning of the following array, using the *median of all elements* present in the array as pivot:

0	1	2	3	4	5	6	7	8
48	55	64	83	4	20	54	25	61

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

What is the asymptotic worst-case complexity of the partitioning algorithm? Write your answer in O-notation and be as exact and simple as possible. Explain why the complexity of partitioning an array has this order of growth.

Answer

What is the median value you used as a pivot? 54

Write down how the array looks after the partitioning:

Here, we use the Hoare partitioning scheme taught in the course. If you used another algorithm, that's fine as long as you say which one you used.

0	1	2	3	4	5	6	7	8
4	25	48	20	54	83	64	55	61

Complexity: O(n) (in the length n in the array)

Justification:

The partitioning itself is linear, O(n): the two pointers which together visit each array cell once.

If you regarded that finding the median was part of the partitioning algorithm, then the complexity is O(n log(n)). Finding the median in a list is O(n log(n)), and this is difficult to improve because it requires some kind of sorting.

The question was a bit unclear whether or not to include finding the median, so we accept both answers.

Basic question 3: Hash tables

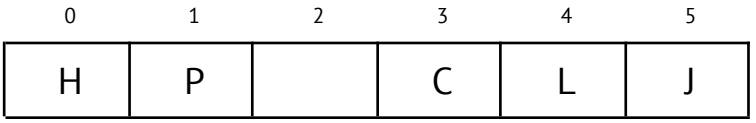
Here is a simple data structure that models a teacher:

```
class Teacher:
    name : String
    office : int
```

Suppose we have defined the following teacher objects:

```
L = new Teacher("Lex", 6114)
C = new Teacher("Christian", 6467)
H = new Teacher("Hazem", 6476)
J = new Teacher("Jonas", 6108)
P = new Teacher("Peter", 6125)
```

and have inserted them in the following hash table, which is implemented using *linear probing*. You can assume that the table hasn't been resized, and no elements have been deleted.



The hash function is the *length of a teacher's name* modulo 6:

$h(t)$ = `t.name.size()` mod 6

for example

h(C) = `C.name.size()` mod 6 = "`Christian`".`size()` mod 6 = 9 mod 6 = 3

a) Which object(s) could have been the first one(s) to be inserted into the table?
b) Why is this a particularly bad hash function?

Answers

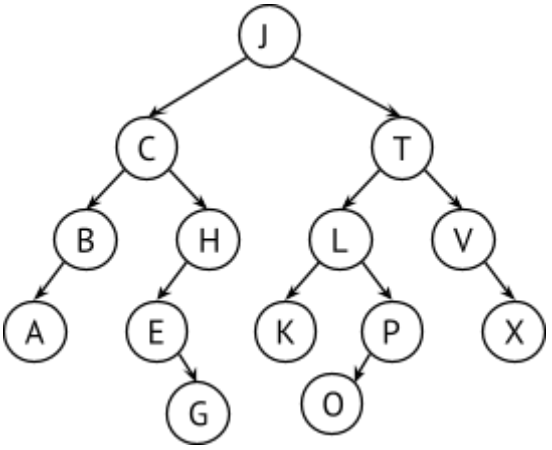
List the object(s) that could have been inserted first: C, J
(*Note*: there might be several and you must list all of them)

Why is this a particularly bad hash function?

Many teacher names have the same length, resulting in a lot of collisions.

Basic question 4: AVL trees

Consider the AVL tree to the right.



Someone just made an insertion, but forgot to perform the necessary rotation(s) to restore the AVL balance. It is your job to figure out which value got added, which node now violates the AVL property, and perform the necessary tree rotation(s) to restore the AVL balance.

Answer

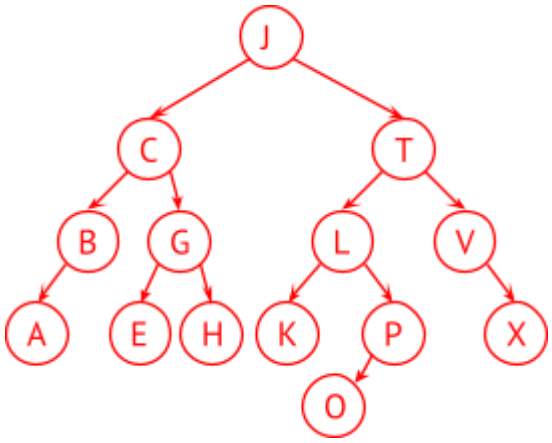
Which value got added? G

Which node violates the AVL balance property? H, the difference in height must be one at most, and is two in this node.

What rotations does the AVL algorithm do?

This is a left-right case and we can fix this with a left rotation at E followed by right rotation at H.

Resulting tree after the rotation(s):



Basic question 5: Binary heaps

Here is a minimum-heap `h`, implemented with a binary heap, with integer values:

0	1	2	3	4	5	6	7	8
2	12	5	16	20	8	10	33	17

We have defined an `update` function on a minimum-heap:

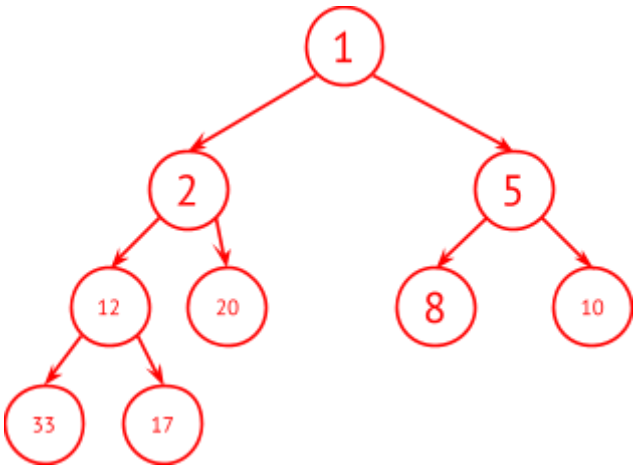
```
function update(h : array, oldVal : int, newVal : int):
    int i = find(h, oldVal)
    if i >= 0:
        h[i] = newVal
        if newVal > oldVal:  sink(h, i)   // sift new value down
        else:                swim(h, i)   // sift new value up

function find(h : array, val : int) -> int:
    for i from 0 to h.length - 1:
        if h[i] == val:
            return i
    return -1
```

The update function tries to find a given value (`oldVal`) and replaces it with a new one (`newVal`) in the underlying array (`h`). It subsequently calls the correct binary heap function (`sink` or `swim`) to restore the heap invariant.

Suppose we update the given heap `h` with: `update(h, 16, 1)`, draw the resulting heap as a tree.

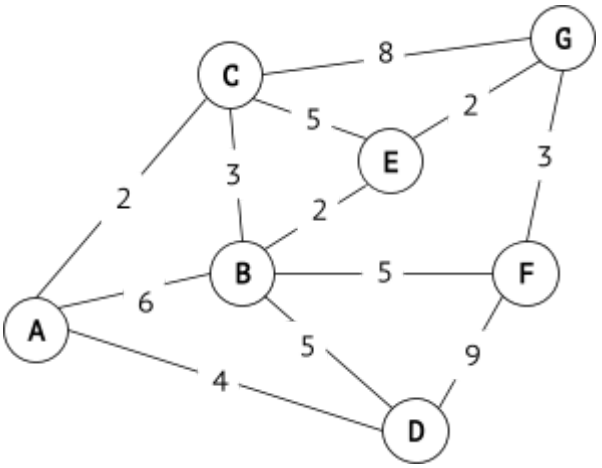
Answer

Draw the updated min-heap as a tree.



Basic question 6: Graphs

We can represent any graph using an *adjacency list*. Your task is to write down such a representation for the *undirected*, *weighted* graph on the right. Note that the weights must be present in the adjacency lists.



In addition, perform Dijkstra's algorithm on the graph to the right, starting in node **A**. In which order does the algorithm visit the nodes, and what is the computed distance to each of them?

Answer

Adjacency lists:

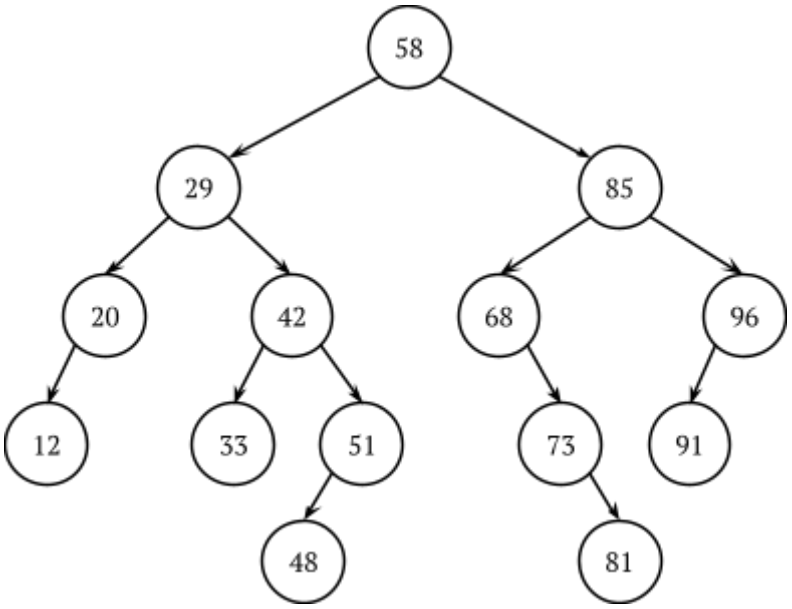
Node	Adjacency list
A	B:6, C:2, D:4
B	A:6, C:3, D:5, E:2, F:5
C	A:2, B:3, E:5, G:8
D	A:4, B:5, F:9
E	B:2, C:5, G:2
F	B:5, D:9, G:3
G	C:8, E:2, F:3

Dijkstra:

	first visited						*last visited*
vertex	A	C	D	B	E	G	F
distance from A	0	2	4	5	7	9	10

Advanced question 7: Binary search trees

The figure below shows a binary search tree with integers:



Your task is to define a function that finds the closest value to a given value in a *balanced* binary search tree. The value x is closest to y when |x - y| is minimum. For example, the value closest 85 is 81, and 20 is closest to 12. You may assume that the given value is present in the tree, and if there are multiple nodes that are equally close, you may just choose one. Your function should be written in either pseudocode, Haskell, Java or Python.

To get awarded one point your solution should have a linear complexity, $O(n)$, or better. For two points your solution should have a logarithmic complexity, $O(\log\ n)$.

Explain the complexity of your solution.

Notes:

- The phrasing of the closest value should have been more precise: the closest value different from the given one. That is, given y, the node value x **different from y** such that |x - y| is minimum. This was clear from the given examples.
- The definition of *n* is the size of the BST, as usual.

Answer

There are many ways to do this. Here is just one. It takes the root node of a tree to search, the value whose closest neighbour is to be found, and a preliminary result (defaulting to null).

```
function nearest(node, value: Integer, result = null: Optional<Integer>) → Optional<Integer>:
    if node is null:
        return result
    if node.value ≠ value and (result is null or |node.value - value| < |result - value|):
        result = node.value
    if node.value ⩽ value:
        result = nearest(node.left, value, result)
    if node.value ⩾ value:
        result = nearest(node.right, value, result)
    return result
```

We claim that the call

```
nearest(node, value)
```

takes $O(\log(n))$ in the number of nodes n of the BST rooted at node.

To see this, note that the function makes at most one recursive call, except it does two if value equals node.value, which can only happen once. Therefore, it searches at most two branches of the tree. Since the tree is balanced, every branch contains $O(\log(n))$ nodes. The time spent examining each node is $O(1)$. In total, this makes the complexity $O(\log(n))$.

Advanced question 8: Cyclic order binary search

Let **A** be an array of $n \geqslant 1$ integers. Assume that **A** is in *strict cyclic order*: $\mathbf{A}[i] < \mathbf{A}[(i + 1) \bmod n]$ for all indices i except one. (Note that this implies that all elements of **A** are distinct.)

Part A

Find an algorithm that computes the index of the minimum of **A** and runs in time $O(\log n)$.

```
int findMinIndex(int[] A)
```

Part B

Find an algorithm that checks if **A** contains an element v and runs in time $O(\log n)$:

```
boolean contains(int[] A, int v)
```

Give your algorithms in pseudocode or a suitable programming language, which we mentioned before. Justify why your algorithms work, and why they have these complexities.

Solving part A will give one point, solving both part A and B gives two points.

Answer

Part A

There are several possible solutions, but all are variants of binary search. The main idea is to select the interval where the first element is larger than the last – because the smallest element must be in that interval:

```
function findMinIndex(int[] A) → int:
    // First check if the array is ordered.
    if A[0] ⩽ A[A.length - 1]:
        return 0

    // Invariant: left < right, A[left] > A[right]
    left = 0
    right = A.length - 1

    // Goal: right = left + 1. Then right is the index of the minimum.
    while right - left > 1:
        mid = (left + right) / 2  // rounding in an arbitrary direction
        if A[left] > A[mid]:      // case A[left] > A[mid] < A[right]
            right = mid
        else:                     // case A[left] < A[mid] > A[right]
            left = mid

    return right
```

Part B

First we find the index of the smallest element using part A. Then we can compare the searched element v with the first array element A[0] to decide which part of the array we have to search in. This part is then guaranteed to be sorted, so we can use standard binary search to look for v.

```
function contains(int[] A, int v) → boolean:
    orderStart = findMinIndex(A)
    if v ⩾ A[0]:
        return binarySearch(A, 0, orderStart - 1)
    else:
        return binarySearch(A, orderStart, A.length - 1)
```

There are many other possibilities, but all involve binary search in some way. E.g.:

- It's possible to tweak the part A solution directly instead of calling it.
- Binary search in the *virtual list* (an example of a *wrapper*) of same size as **A** that sends index i to **A**[(orderStart + i) mod n]

Advanced question 9: Design a matrix data structure

Your task is to design a data structure for storing a *matrix*, or two-dimensional array, of integers. The data structure should support the following operations:

- `zero(n, m)` Create an n-by-m matrix. *Initially, all values in the matrix should be zero*.
- `get(i, j)` Return the value at row `i`, column `j` of the matrix. Assume that rows and columns are numbered starting from 0.
- `set(i, j, x)` Set the value at row `i`, column `j` of the matrix to `x`.

All three operations must have asymptotic worst-case time complexity $O(\log\ N)$, where N is the maximum number of integers: $N = n \times m$. Implementing these operations will give you one point.

Note! The creation of an array of length n takes $O(n)$ time.

To get two points, your data structure should support two more operations:

- `scale(k)` multiply every number `x` in the matrix with `k`.
- `setRow(i, x)` set all values at row `i` to `x`.

These operations should be $O(\log\ N)$ as well, or better.

Your answer should specify what design you have chosen, how each operation is implemented, and motivate the time complexity of your solution.

You should write your answer in pseudocode, Haskell, Java or Python. You may freely use standard data structures and algorithms from the course without explaining how they are implemented. Furthermore, you may assume that a type or class exists that represents a pair of numbers, and that comparisons take $O(1)$ time.

Answer

The trick is to compute the actual cell values lazily (only when requested). To avoid creating arrays, we store cell/row values using maps instead. For the scaling operation, we just accumulate a scaling factor. To get both scaling and setting values to work logarithmically, we store the scaling factor at the time of setting the value with the value (making it essentially a fraction). Note that the specification does not require us to store the matrix dimensions.

The implementation is shown on the following page. Notes:

- It uses inner helper classes for structuring. Instances of those classes have an implicit pointer to an instance of the outer class, hence have access to its instance variables.
- The map method *get _init* initialises the value for the key if it does not exist. It can be implemented in terms of *contains*, *get*, *put*.
- All maps have size O(N), so their operations are O(log(N)). The only loop has constantly many iterations. Therefore, the complexity of each matrix operation is O(log(N)).

```
class Matrix:
    scaling: Integer, initialised as 1

    class Value:
        scaled: Integer
        value: Integer

        constructor Value(x):
            scaled = scaling
            value = x

        function effective() → Integer:
            if  (scaling, scaled) = (0, 0): return 0
            return (scaling / scaled) · value

    class Row:
        value: Optional<Value>
        cells: Map<Integer, Value> implemented using self-balancing BST

    rows: Map<Integer, Row> implemented using self-balancing BST

    zero(m n: Integer):
        scaling = 1
        rows = new map

    get(i j: Integer) → Integer:
        row = rows.get(i)
        if row is not null:
            for v in [row.cells.get(j), row.value]:
                if v is not null: return v.effective()
        return 0

    set(i j: Integer, x: Integer):
        // The map method get _init initialises the value for the key if it does not exist.
        rows.get_init(i).cells.put(j, Value(x))

    scale(k : Integer):
        scaling = scaling · k

    setRow(i : Integer, x: Integer):
        row = rows.get_init(i)
        row.value = Value(x)
        row.cells = new map
```