August 2018, q1

Analyse the worst-case time complexity of the following code, as a function of n:

```
for (int i = 0; i < n; i++) {
  int x = q.dequeue();
  if (s.contains(x))
    l.addLast(x);
  else
    l.addFirst(x);
}
```

Use the course's uniform cost model and make the following assumptions:

- That n is a positive integer
- That `q` is a queue of integers that initially has n elements, implemented as a linked list
- That `s` is a set of integers implemented using an AVL tree, initially containing at most $2n$ elements
- That `l` is an initially empty list of integers implemented using a dynamic array

The analysis should make reference to the program code. You can assume that operations on standard data structures take their usual time. Your answer should be a simple big-O expression (not a sum, recurrence relation or similar). Unnecessarily imprecise answers may be rejected.

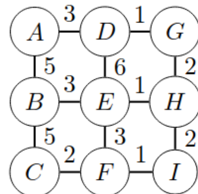
Answer

The loop executes n times. `q.dequeue` takes $O(1)$ time. `s.contains` takes $O(\log(2n)) = O(\log n)$ time. `l.addLast` takes $O(1)$ amortised time. `l.addFirst` takes $O(i) = O(n)$ time.

The total time complexity is $O(n(1 + \log n + n)) = O(n^2)$.

August 2018, q2

Suppose we use Dijkstra's algorithm to calculate the shortest path from node A to all other nodes in the following graph:



Write down the nodes in the order they are enumerated, together with their distances.

Answer

- Node A, distance 0
- Node D, distance 3
- Node G, distance 4
- Node B, distance 5
- Node H, distance 6
- Node E, distance 7
- Node I, distance 8
- Node F, distance 9
- Node C, distance 10

August 2018, q3

Design a data structure that represents a set of integers.

The data structure should support the following operations:

- `empty()`: creates an empty set.
- `add(x)`: adds the integer x to the set. If x is already present in the set, the set is unchanged.
- `remove(x)`: removes the integer x from the set. If x is not present in the set, the set is unchanged.
- `member(x)`: returns `true` if x is present in the set.
- `ithsmallest(i)`: returns the ith smallest value in the set, indexing from 0. That is, if the set contains $x_0, x_1 \ldots, x_{n-1}$, where $x_0 < x_1 < \ldots < x_{n-1}$, then this operation should return x_i. You may assume that $0 \leq i < n$, where n is the number of values in the set.

The operations should have the following complexity, where n is the number of values in the set:

- For a 3: `empty` should take $O(1)$ time and the other operations should take $O(n)$ time.
- For a 4 or 5: `empty` should take $O(1)$ time and the other operations should take $O(\log n)$ time.

You should motivate why your solution has the correct complexity.

You may use or adapt standard data structures and algorithms from the course, without describing them in detail. You do not need to write detailed code, but your solution should include enough detail to be able to analyse the complexity.

Answer for a 3

Store the elements in a sorted dynamic array. `add(x)` should check if `x` is present in the array, and if not, insert it in the correct place ($O(n)$ time). `remove(x)` should remove `x` from the sorted array ($O(n)$ time). `member` should use a linear search ($O(n)$ time). `ithsmallest(i)` can be implemented by returning `arr[i]` and takes $O(1)$ time.

Answer for a 4 or 5

Use a balanced binary search tree (e.g. an AVL tree or AA tree), but add to each node a `size` field which records the number of nodes in that subtree. This field can be easily maintained during insertions, deletions and rotations at a cost of $O(1)$ time per modified node. `add(x)` uses AVL insertion, `remove(x)` uses AVL deletion and `member(x)` uses binary search tree lookup (all $O(\log n)$ time).

`ithsmallest(i)` can then be implemented like this:

```
ithsmallest(i) = ithsmallestof(i, root)
ithsmallestof(i, node) {
  if (node.left == null)
    leftsize = 0;
  else
    leftsize = node.left.size;
  if (i < leftsize)
    return ithsmallestof(i, node.left);
  else if (i == leftsize)
    return node.value;
  else
    return ithsmallestof(i - leftsize - 1, node.right);
}
```

August 2018, q4

The following array represents a binary min-heap.

3	5	8	10	8	13	16	12	13	15	20	20	14	

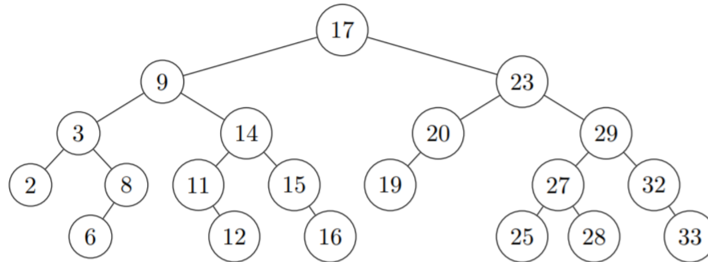
Suppose we now insert 2 into the heap. What does the array look like afterwards?

Answer

2	5	3	10	8	13	8	12	13	15	20	20	14	16

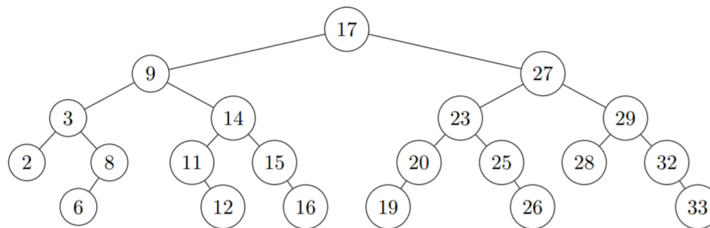
April 2018, q2

The following figure shows an AVL tree.



Show how the AVL tree looks after the value 26 has been inserted.

Answer



April 2016, q2

You are given a binary tree where the nodes have parent pointers (each node has an edge to its parent). However, some of the parent pointers may be pointing at the wrong nodes. Implement an algorithm that takes a binary tree with parent pointers, and checks if all the parent pointers are correct. **For a 4 or 5** you must also show that your algorithm takes linear time in the size of the tree.

You may assume that the tree class is defined as follows:

```
// Binary tree with parent pointers.
public class Tree<A> {
  // Tree nodes; null represents an empty tree.
  class Node {
    A contents;     // Value stored in the node.
    Node left;   // Left child.
    Node right;  // Right child.
```

```
    Node parent; // Parent (null for root); may be incorrect.
  }

  // The root.
  private Node root;

  // Your task.
  public boolean parentsAreCorrect() {
    ...
  }
}
```

Only **detailed code** (not necessarily Java) will be accepted. You may not use other methods or procedures apart from ones you have implemented yourself.

If you want, you can assume that the tree's height is $O(\log n)$, where n is the number of nodes in the tree.

Answer

The following recursive algorithm works:

```
public boolean parentsAreCorrect() {
  return parentsAreCorrect(root, null);
}

private boolean parentsAreCorrect(Node node, Node, parent) {
  if (node == null) {
    return true;
  } else if (node.parent != parent) {
    return false;
  } else {
    return (parentsAreCorrect(node.left, node) &&
            parentsAreCorrect(node.right, node));
  }
}
```

Since the algorithm visits each tree node once, and performs $O(1)$ work per node, the total time taken is $O(n)$.

DIT960, August 2014, q2

Design an algorithm that takes:

- An array containing n distinct natural numbers
- A number $k \leq n$

and computes the sum of the k largest numbers in the array.

For example, if the array is $\{3, 7, 5, 12, 6\}$, and $k = 3$, then the algorithm should return 25 $(12 + 7 + 6)$.

Write down your algorithm as **pseudocode** – you don't need to write fully detailed Java code. **You may freely use standard data structures and algorithms from the course in your solution, without explaining how they are implemented.**

For a 3: your algorithm should take $O(n \log n)$ time.

For a 4 or 5: your algorithm should take $O(n \log k)$ time.

Answer

For a 3: sort the array, then add up the first k elements.

For a 4/5: use a binary heap (specifically a min heap). The idea is that the heap will contain the k largest elements we have seen so far. When the heap grows to contain more than k elements, we eject the smallest one from the heap, so that it still contains the k largest elements. Pseudocode:

```
h = new min-heap
for each x in array
  add x to h
  if the size of h is > k then
    delete min element from x
return sum of all elements of h
```

DIT960, August 2016, q4

Design a data structure for storing a set of integers. It should support the following operations:

- `new()`: creates an empty set.
- `insert(x)`: adds the integer x to the set.
- `member(x)`: returns `true` if x is present in the set.
- `increasBy(x)`: add x to all the integers in the set.

For example, after calling `increaseBy(2)` on a set containing 1,2,3,4,5, the set afterwards should contain 3,4,5,6,7.

You may use or adapt standard data structures and algorithms from the course, without describing them in detail. You do not need to write detailed code, but your solution should include enough detail to be able to analyse the complexity.

The operations should have the following complexity, where n is the number of values in the set:

- For a 3: `new` should take $O(1)$ time, `insert` and `member` should take $O(\log n)$ time and `increaseBy` should take $O(n)$ time.
- For a 4 or 5: as for a 3, but the complexity of `increaseBy` should be $O(1)$.

You should motivate why your solution has the correct complexity.

Answer for a 3

We can use an AVL tree.

- new(): create a new AVL tree, which takes $O(1)$ time
- insert(x): use AVL insertion, which takes $O(\log n)$ time
- member(x): use BST membership testing, which takes $O(\log n)$ time
- increaseBy(x): use (e.g.) an inorder traversal to iterate through all nodes of the tree. For each node, add x to the value. Note that this preserves the relative order of all elements so will not break the AVL invariant. It takes $O(n)$ time because each node of the tree is visited once.

Answer for a 4/5

We maintain both an AVL tree and an integer k which stores the total of all calls to increaseBy.

- new: create a new AVL tree, set k=0. This takes $O(1)$ time.
- insert(n): insert n-k into AVL tree. This takes $O(\log n)$ time.
- member(n): check if n-k is contained in AVL tree. This takes $O(\log n)$ time.
- increaseBy(n): set k=k+n. This takes $O(1)$ time.