Algorithms and Data Structures

Final Exam

August 8, 2022

Last Name: ______________________________

First Name: ______________________________

Signature: ______________________________

1. (a) Order the following functions by their asymptotic growth rate:

6 4 2 3 1 5

$$n\,(n + \log n), n\,\log n, \log n, (\log n)^2, 1\,000\,000\,000, n^{3/2}$$

(b) Give a sharp Big-Oh upper bound and a sharp Big-Omega lower bound for the running time of the following Python function which takes as input a Python list `S` of length n.

```python
def mystery_function (S):
    for j in range(len(S)):
        for k in range(j+1, len(S)):
            if S[j] == S[k]:
                return False
    return True
```

(c) What is `mystery_function` good for?

(d) Can you suggest a different implementation of `mystery_function` which has a running time of $O(n \log n)$?

(5+5+5+5)

(b) Worst case: The if-condition is never true. Then, with $n = \mathrm{len}(S)$, the inner loop is executed

$$(n-1) + (n-2) + \dots + 1 = O(n^2)$$

times.

Best case: The if-condition is true the first time and the function is exited. Thus, the lower bound is $\Omega(1)$.

(c) Returns True if all entries of S are unique, otherwise False.

(d) First, sort the array at $O(n \log n)$ cost, then check if all elements are distinct from their nearest neighbor at $O(n)$ cost.

2. True or false? Explain your answer in 1–2 sentences.

   (a) A pre-order traversal of a tree with n nodes runs in $O(n)$ time.

   (b) Accessing the next element in a pre-order traversal of a tree can always be done in $O(1)$ time.

   (c) Accessing the next element in a breadth-first traversal of a binary tree can always be done in $O(1)$ time.

   (d) Inserting an element into a hash table can always be done in $O(1)$-time.

   (e) There are binary search trees where a search can take $\Omega(n)$ time.

(2+2+2+2+2)

(a) True. A pre-order traversal visits every edge twice, and the number of edges in a tree is $O(n)$.

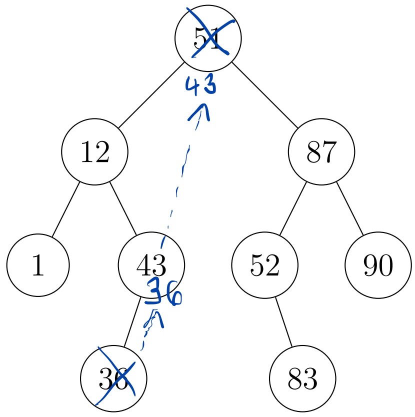
(b) False. Some steps require traversing $O(h)$ nodes, where h is the height of the tree.

(c) True. The processing of each node requires one dequeue and at most two enqueues, so it is $O(1)$ in every case.

(d) False. The may be, in the worst case, $O(n)$ collisions that need to be iterated over. (Only if insertion is done with separate chaining using a linked list, and if no checking for duplicate keys takes place, a guaranteed $O(1)$-insertion is possible.)

(e) False. The best case is when the key that is searched for is in the root of the tree, so that no iteration is required.
Note, however, that there are highly unbalanced search trees where the worst-case behavior is $O(n)$.

3. (a) Draw the resulting binary search tree when you remove the root node 51 from the following tree.

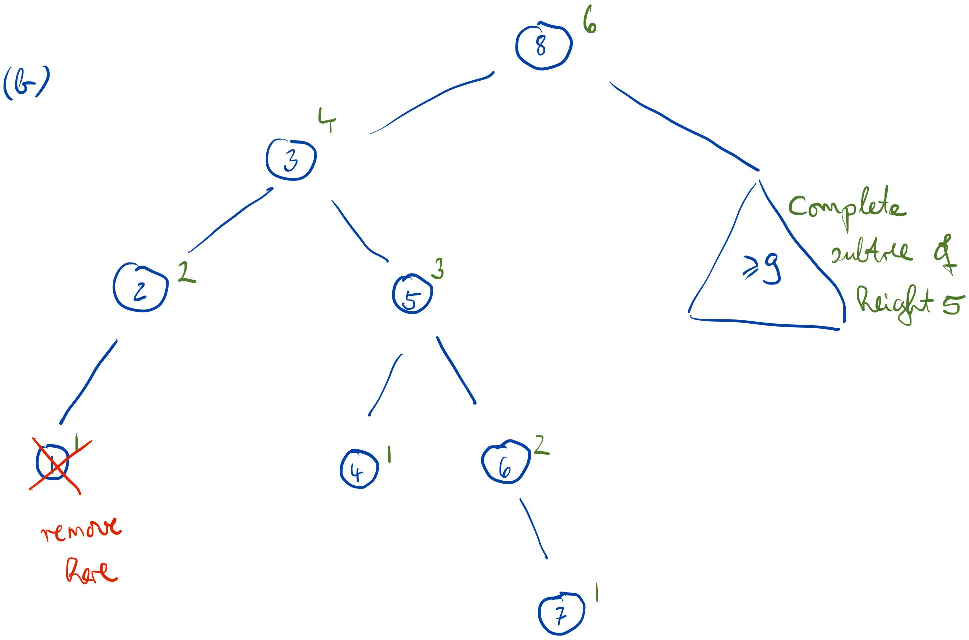


(b) Draw an example of an AVL-tree where the removal of an element triggers more than one rotation.

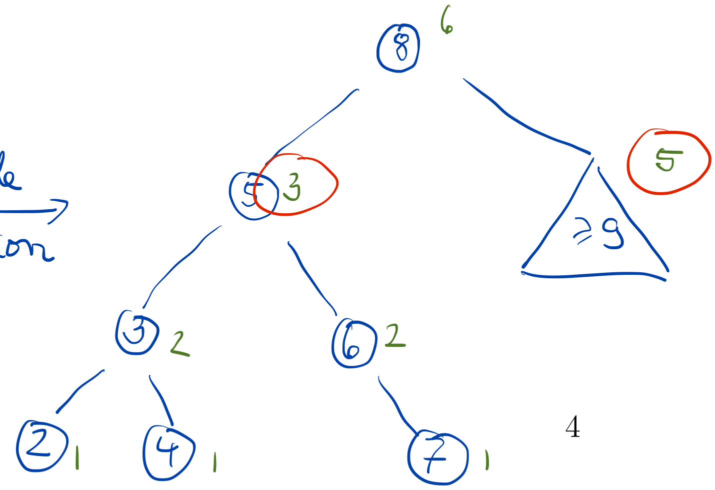
(c) Suppose you have an implementation of an AVL-tree. Argue that you can use it to sort a list in $O(n \log n)$ time.

(5+5+5)

(b)



double rotation



Now the root is unbalanced and triggers another rotation!

(c) Inserting an element into an AVL tree of size k can be done in $O(\log k)$ time.

Thus, putting all elements from the list into an initially empty AVL tree takes

$$O(1 + \log 2 + \log 3 + \dots + \log n) = O(n \log n)$$

time.

The sorted list can then be obtained by an in-order traversal of the tree at $O(n)$ time.

4. Let G be an undirected graph without weights. The graph distance between vertices u and v is then defined as the length of the shortest path between u and v. The following function computes the graph distance, where LinkedQueue is an implementation of the standard queue data type, and neighboring_vertices(w) is an iterator over all vertices connected to w by an edge.

```python
def distance(G, u, v):
    Q = LinkedQueue()
    Q.enqueue((u,0))
    while not Q.is_empty():
        w, d = Q.dequeue()
        if w == v:
            return d
        for s in G.neighboring_vertices(w):
            Q.enqueue((s, d+1))
```

   (a) How does this algorithm work? (You may argue by analogy with one of the tree traversal algorithms.)

   (b) In certain cases, this algorithm may fail spectacularly. Why and when?

   (c) What is the running time, in Big-Oh notation, of this algorithm if every vertex has an edge to 5 others? And if the graph is complete, i.e., every vertex has an edge to every other vertex?

   (d) Give a modification of the algorithm that reduces the running time to $O(n+m)$, where n is the number of vertices and m is the number of edges in the graph. (You may state the modification in precise language – no need to write Python code.) Give a brief reasoning why your modification respects the required running time bound.

   (e) Does your modification avoid the problem from part (b)? Why or why not?

(5+5+5+5+5)

(a) The code has the structure of a breadth-first traversal of a tree. When enqueueing a new vertex, we add a counter d which keeps track of the current distance to the starting vertex u. Since this counter is non-decreasing, the first time the final vertex v is encountered, d contains the length of the shortest path from u to v.

(b) If there is no path from u to v, the code may never exit with a queue that keeps growing unboundedly.

(c) If every vertex is connected to 5 others, every vertex will cause 5 more vertices to be enqueued. Thus, the running time is, in the worst case, $O(5^d)$ where d is the distance from u to v.

When the graph is complete, there is an edge from u to v, so $d=1$, for a worst-case running time $O(n)$.

(d) Mark each vertex with a Boolean marker when it is visited for the first time. Check this marker before enqueueing to never enqueue a vertex twice.

(e) Yes, because the algorithm will stop at the latest when every vertex was enqueued once.

5. Write an algorithm `search`, in pseudo-code or in Python, which returns the position of an element with key `k` in a binary search tree `T` if the element is found, and returns `None` otherwise.

(10)

```
def search (T, r, k):
    if T.key(r) == k:
        return r
    elif T.has_left_child(r) and T.key(r) < k:
        return search (T, T.left_child(r), k)
    elif T.has_right_child(r) and T.key(r) > k:
        return search (T, T.right_child(r), k)
    else:
        return False
```