Information

Solutions to the exam in Algorithms and Data Structures I (1DL210), Uppsala University, held on Monday, October 24, 2022.

Problem 1

a) True

b) True

c) False

d) False

e) True

f) False

g) True

h) False (for example, consider $f(n) = 2n$ and $g(n) = n$)

i) True

j) False

Problem 2

a) $\Theta(n^2)$

b) $\Theta(n \log n)$

c) $\Theta(n \log n)$

d) It would take at most $k + 1$ comparisons when the array size is doubled.

e) It could solve instances of size $m + 1$ in one hour on the faster computer.

Problem 3

a) Two words are anagrams if they contain exactly the same number of copies of each letter. We can find out if this is the case by, for example, sorting the characters in both character arrays and then comparing the sorted arrays to see if they are identical. The sorting can be done in $O(n \log n)$ worst-case time, and comparing two arrays in $O(n)$ time, therefore our algorithm has $O(n \log n)$ worst-case running time. When sorting, it does not really matter which ordering we use (for example, we can sort the characters in alphabetical order or in order of their ASCII codes). The pseudocode of this algorithm is as follows.

```
ANAGRAM(Word1, Word2)
▷ We assume here that the words are represented as arrays of characters.

▷ If the words have different length, then they are not anagrams.
if Word1.length ≠ Word2.length
   then return False

▷ We can use any sorting algorithm with $O(n \log n)$ worst-case running time.
MERGE-SORT(Word1)
MERGE-SORT(Word2)

▷ Check if the sorted arrays are identical.
for i ← 1 to Word1.length
     do
        if Word1[i] ≠ Word2[i]
           then return False

return True
```

b) In this case we can find out if the words are anagrams in $O(n)$ worst-case time. If we know that the words only contain the possible characters A–Z, then there are at most 26 possible values for each array element. We can represent character "A" by integer 1, character "B" by integer 2, etc. One way to solve the problem is then as follows:

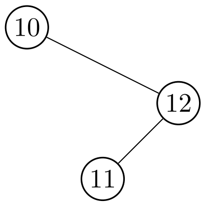
- Create an array *Count* of 26 integers where each element is initialized to 0.
- Loop over the first character array and add 1 to *Count*[i] every time we see the ith character. For example, for every "B" we see, we add 1 to *Count*[2].
- Loop over the second character array and subtract 1 from *Count*[i] every time we see the ith character. For example, for every "B" we see, we subtract 1 from *Count*[2].
- Loop over *Count* and return False if we find any non-zero element of *Count*. Otherwise return True.

Each of the three loops takes $O(n)$ time in the worst case, so the algorithm has a worst-case running time of $O(n)$. (In fact, it is also $\Theta(n)$.)

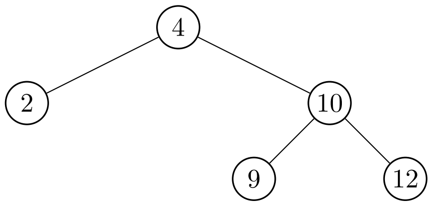
The same algorithm idea of course works as long as we have a *constant* number of possible values for the array elements. It does not matter if it is characters A–Z, characters A–Ö, or some other constant-sized set of possible values.

Problem 4

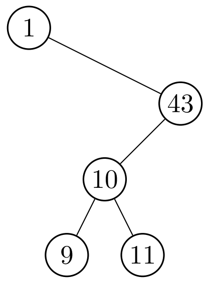
a)



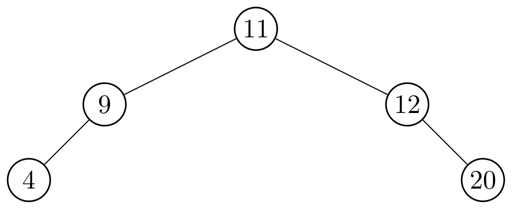
b)



c)



d)



Problem 5

Let n be the length of the array A. We know that INORDER-TREE-WALK(T) always takes $\Theta(n)$ time if T has n nodes, so any difference between the best- and worst-case running times of TREE-SORT has to be due to the calls to INSERT.

We know that INSERT(T, x) takes $\Theta(h)$ time in the worst case, where h is the height of T. It will in fact take $\Theta(h)$ time if the inserted element x gets inserted into a level lower than any other element in T. With these observations we can answer the questions below.

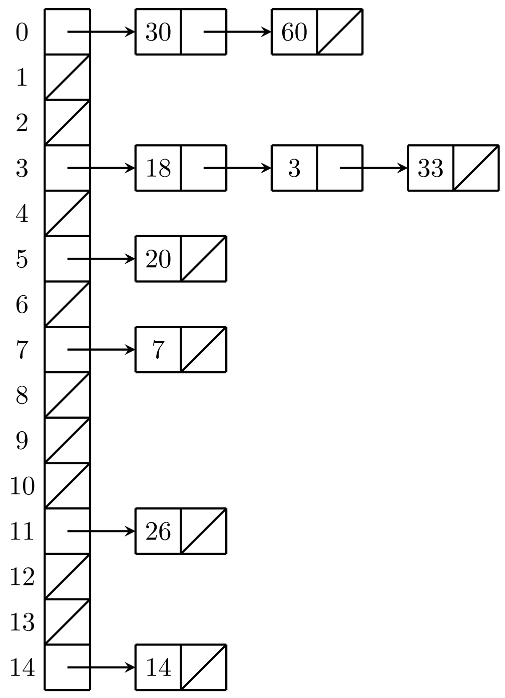
a) The worst-case running time for TREE-SORT is seen for inputs that create a BST of maximal height. This happens, for example, if the array A is already sorted. Inserting the ith element then takes $\Theta(i)$ time, and the total running time of the loop in lines 4–5 is $\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$.

b) The best-case running time for TREE-SORT is seen for inputs that create a BST of minimal height. This happens if the array A is sorted in such a way that the created BST is a complete binary tree. For this to happen, the root of the BST (which is the first element inserted) has to be the median element of A, the left child of the root should be the median of the elements less than the root, etc. For example, inserting the elements of $A = [4, 2, 6, 1, 3, 5, 7]$ in order would create a complete binary tree.

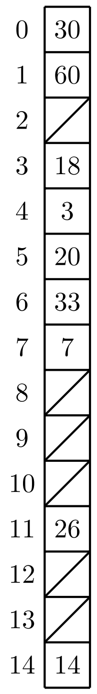
The height of T will be $\Theta(\log n)$ in this case. We can find matching upper and lower bounds for the loop in lines 4–5: Since there are n elements to insert, and each insertion takes $O(\log n)$ time, the loop must take $O(n \log n)$ time. We can also note that $\Omega(n)$ of the nodes in T are leaves, and inserting the elements that ended up as leaves must have taken $\Omega(\log n)$ time, so the loop must take $\Omega(n \log n)$ time. Combining the upper and lower asymptotic bounds, we see that the loop takes $\Theta(n \log n)$ time.

Problem 6

a)



b)



c) When expressed in öre, many prices seen in practice (like the example prices in the question) are divisible by 5 (i.e., they end with either the digit "0" or "5").

Using the hash function $h(k) = k \mod 15$ is not very suitable in this case. Since 15 is a multiple of 5, we will have that $h(k)$ is divisible by 5 whenever k is divisible by 5. In other words, all keys that are divisible by 5 will be mapped to one of three slots in the hash table: 0, 5, or 10.

As an example, consider the "typical" prices mentioned in the question:

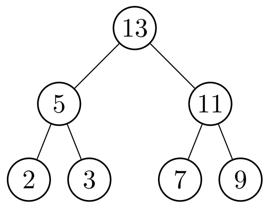
$$\begin{aligned} h(500) &= 5 \\ h(1995) &= 0 \\ h(1000) &= 10 \\ h(995) &= 5 \\ h(1045) &= 10 \\ h(1600) &= 10 \\ h(1395) &= 0 \end{aligned}$$

If most of our keys are in fact divisible by 5, such as in the example, then most keys will be mapped to only 3 of the 15 slots in the hash table, creating a lot more collisions that necessary. A better hash function would be one that does not preserve simple patterns like this in the hashed values. There are many types of hash functions that tend to work better, but if we keep to the simple division method, where $k(h) = k \mod m$, then we typically avoid these type of issues by picking a prime number as the hash table size m. The prime numbers closest to 15 are 13 and 17, so these would be good choices.
For example, let $h'(k) = k \mod 17$ and consider the prices from the question again:

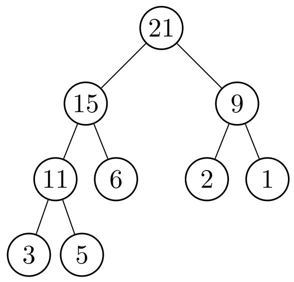
$$\begin{aligned} h'(500) &= 7 \\ h'(1995) &= 6 \\ h'(1000) &= 14 \\ h'(995) &= 9 \\ h'(1045) &= 8 \\ h'(1600) &= 2 \\ h'(1395) &= 1 \end{aligned}$$

Problem 7

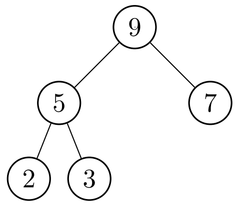
a)



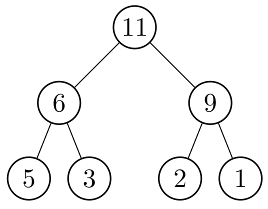
b)



c)



d)



e) The only possible value is $x = 8$.

f) This question forgot to specify which running time (best or worst) that was meant. Both Insert and Extract-Max have $\Theta(\log n)$ worst-case running time and $\Theta(1)$ best-case running time.

Problem 8

a) a, h, c, i, j

b) a, f, e, h, c, i, g, j

c) $d, b, c, e, g, i, a, h, j, f$

d) $d, b, c, i, g, j, e, a, f, h$

Problem 9

Let π be a shortest path from u to w that goes through v. Our goal is to find π if it exists. It must be the case that π is a concatenation of two paths π_1 and π_2 such that π_1 is a shortest path from u to v, and π_2 is a shortest path from v to w.

We know that BFS will find a shortest path from a given start node to all other reachable nodes. We can therefore implement Shortest-Path-With-Stop to find π by using two calls to BFS, once starting from u and once starting from v. From the two calls to BFS we extract a shortest path $u \rightsquigarrow v$ and a shortest path $v \rightsquigarrow w$, and print one after the other. If either call to BFS does not find any path (to v or w, respectively), then the path π does not exist and we print `NIL` instead.

Since BFS takes $O(|V| + |E|)$ time in the worst case, and since Shortest-Path-With-Stop only does two calls to BFS and some simple printing, it follows that Shortest-Path-With-Stop takes $O(|V| + |E|)$ time in the worst case as well.

Problem 10

The high-level problem with the proof attempt is that it is repeatedly moving the goal post.

In more detail, the claim that the maximum element can be found with a worst-case running time of $O(1)$ means that we claim that there exists an algorithm to solve the problem with worst-case running time $T(n) = O(1)$. That, in turn, means that we claim that there exists constants c and n_0 such that $T(n) \leqslant c$ for all $n \geqslant n_0$.

A proper proof by induction would be along the following lines, for some constant c and n_0.

Base case: Show that $T(n_0) \leqslant c$.

Induction step: Assume as the induction hypothesis that $T(n-1) \leqslant c$, for some $n > n_0$. Prove that $T(n) \leqslant c$.

The problem with the flawed proof attempt is that it used as induction hypothesis the assumption that the maximum element of the first $n-1$ elements can be found in $O(1)$ time. We then compare that maximum element with the final element and take the maximum of that, also in $O(1)$ time. The reasoning that $O(1) + O(1) = O(1)$ is correct, but that is not what we were supposed to show! We are moving the goal post by stating that there exists some functions that are $O(1)$ so that the induction step works out, but we have to stick to the function $T(n)$ and constants c and n_0 from the claim, not pick new ones for the induction step.

The actual claim is that there exist constants c and n_0 such that $T(n) \leqslant c$ for all $n \geqslant n_0$. To prove this claim the induction hypothesis should be $T(n-1) \leqslant c$ and the induction step should be to show that $T(n) \leqslant c$. But now it doesn't add up any more. We can't show that $T(n) \leqslant c$ simply by noting that $T(n) = T(n-1) + O(1)$, since we can't show that $T(n-1) + O(1) \leqslant c$ using only the hypothesis $T(n-1) \leqslant c$, no matter which c and n_0 we have.