Assignment 2
Data Structures and Algorithms for AI
Computer Science Department, VU Amsterdam

Caspar Grevelhoerster

October 10, 2021

Instructions: This is the second assignment of the course. The assignment consists of four questions and they appear below. Your job is to answer the questions and submit your solutions in PDF format via Canvas.

Note 1. If you like to discuss something regarding the assignment, you are only allowed to discuss with your instructor.
Note 2. The deadline to submit your file is before the **end of Sunday October 10, 2021**. Late submissions will not be accepted.
Note 3. You can **use only** the algorithms and/or data structures that we have discussed in the course and **before publishing the assignment, which includes only the first seven lectures of the course**. Moreover, if you need, you can use such algorithms and/or data structures without discussing their details.
Note 4. If you prefer to present your algorithms in code, then they have to present them in pseudocode and NOT an actual code (e.g., Python).

QUESTION 1. [4 POINTS] Please answer the following short questions.

- Let H be a binary max-heap that stores a set S of keys. Where does in H the smallest key of S might live?

  In a heap, the values are ordered by the level they are placed in in the tree structure. There are max- and min-heaps that differ in prioritizing larger values and smaller values respectively. In a max-heap, the largest element is at the top of the tree (root node) and from there, at each level the values decrease. This means that each parent node will have children nodes that have values that are smaller than the parent node's value. According to this structural definition, the lowest of all values S placed in a max-heap H must be stored in one of the leaf nodes of the H. Also, it is not defined that this smallest child must live at the last level necessarily - it could also be a leaf node placed at the second-last level.

- Is the following statement TRUE or FALSE? Briefly explain your answer. Let T be a complete or almost complete binary tree. If we set the key of each node of T to be its index (i.e., location) in the preorder traversal of T, then T is a binary min-heap.

  Preorder traversal is a form of tree traversal where each parent node is visited before its children nodes. Further, a min-heap is a data structure where each parent node has a smaller value than its children. When combining these two statements, it can be noted that if a binary tree is explored using preorder traversal and the visited node's values are set in an increasing order, then, indeed, the tree will comply to the structural prerequisites of a binary min-heap. This is the case since all parent nodes are visited before its children and are therefore assigned a lower value than their descendants which shows the statement to be TRUE.

- Alice has a binary search tree (BST) that stores numbers between 1 and 1000, but we do not know how her tree looks like. We ask Alice to search for 363 in her tree and give us the sequence of numbers she will be visiting during the search. She claims that she has visited 925, 202, 911, 240, 912, 245, and finally 363, during her search for 363. Is Alice's claim true or false? Explain your answer. [*Hint.* After vising 911 she has visited 240, which is a smaller number than 911. What does this mean?]

  A binary tree is a tree where each internal node A has at most two children B and C (read from left to right, so B is the left and C the right child). If both of these children would be considered root nodes of the subtrees B'_A and C'_A respectively, then all values in B' must be smaller than X_A (X as the value of node A) and all values in C' must be greater than X_A by definition of a binary search tree (BST). This structural property enables fast lookup in the BST since for each visit of an internal node, it can be determined which of the two subtrees must the desired value if it exists. The sequence of nodes which are visited during binary search therefore set upper and lower boundaries of the remaining values since they show that all remaining values must be smaller than (in the case of B') or larger than (in the case of C') the value X of the respective visited node A. The first value is 925 (thus the root node A of the binary tree). Since the desired key is 363 and $363 < 925$, the right subtree (C'_{925}) becomes irrelevant (since it stores values larger than 925) and the left subtree (B'_{925}) is considered in the next iteration. This means that by definition, all the values in this subtree B'_{925} must be < 925. Because of this explanation, there can be two kinds of boundaries inferred (one of the two at each iteration): (1) If the desired value is smaller than X_A, X_A becomes an upper boundary of the remaining values in B'_A and (2) if on the other hand, the desired value is larger than X_A, X_A becomes a lower boundary of the remaining values in C'_A. Therefore, the following sequence of upper and lower boundaries of the remaining set of keys K arises due to the visited nodes: $K < 925$; $202 < K < 925$; $202 < K < 911$; $240 < K < 911$. After this last iteration, all remaining values must be within the most recent lower (240) and upper boundary (911). However, 912 is visited at next and it does not lie within the boundaries of the last iteration: $240 < 912 < 911$ does not hold. The value 912 was stored in the left subtree of the node with the value 911. Therefore, the sequence of keys determines that the tree of Alice does not comply to the structural properties of a binary search tree. Her claim is thereby proven to be FALSE.

- Let H_1 and H_2 be two binary min-heaps, each storing n different keys. In addition, assume that the structure of each of the two heaps is a complete binary tree. Briefly explain an algorithm (either in pseudocode or in text) that merges H_1 and H_2 into a single binary min-heap of size exactly $2n$. You do not need to analyze the running time of your algorithm, but it should run in time $O(\log n)$. The space requirement of your algorithm should be only $O(1)$.

  It is possible to merge two complete binary heaps H_1 and H_2 into a single binary heap using an algorithm that performs the following steps: (1) Compare the two roots and determine the root with the smaller value (the two trees will be handled as $H_{smaller}$ and H_{larger} based on this comparison). (2) $H_{smaller}$ (with the smaller root value) will be considered first: the top node of this heap is dequeued and stored in an extra variable old_root. The dequeue-operation involves the repositioning of one of the leaf nodes of $H_{smaller}$ as the new root of the tree (which will make $H_{smaller}$ lose the min-heap-property briefly). $H_{smaller}$'s min-heap property is restored by executing minHeapify() over $H_{smaller}$. This sub-algorithm is also included in the dequeue operation. At this point, there are two min-heaps $H_{smaller}$ and H_{larger} as well as the overall smallest node old_root. (3) These two heaps will be fused by making the old_root the root of both trees. The positioning of the two trees does not matter here since both root nodes are larger than the old_root. After setting roots of the two heaps as children of the old_root, both min-heaps have been merged to one in maximally $O(log(n))$ time.

QUESTION 2. [2 POINTS] Explain an algorithm in words that uses an (initially empty) AVL tree to sort n arbitrary given keys in $O(n \log n)$ time. Also, explain in words why the running time of your algorithm is $O(n \log n)$.

An AVL tree (here abbreviated as "AVLT") is a data structure similar to a binary tree with an additional balance condition. The latter defines that in an AVLT, each internal node's two children nodes can only differ in height by at most 1. When ordering n items using an AVLT, each item is individually considered and inserted into the tree. The insertion operation works as follows: If the AVLT is empty, the key is simply inserted and thereby made the root node of the tree. If, however, the tree is non-empty, then the tree is traversed using binary search until a "free spot" is found. (Binary search compares a visited node to the item; if the item is larger than the visited node's value, the algorithm will visit the right child and if it is lower, the left child in the next iteration. Afterwards, the next iteration restarts with comparing the item to the child which is now considered the root-node of its subtree.) When a "free spot" is reached (a node that is empty; a null-pointer node), then the item is inserted at this spot. After this insertion operation, the balance condition of the AVLT might be violated. To account for this, there are certain rotations performed that restore this balance condition. Since this algorithm needs to iterate over all items ($O(n)$) and insert them into the AVLT-structure ($O(log(n))$ because an AVLT has $log(n)$ levels), it has an overall time-complexity of $O(n * log(n))$.

QUESTION 3. [2 POINTS] Prove by induction on h, that there are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h (leaves are at height 0) in any n-item binary heap. [*Hint 1.* As a base case for $h = 0$, note that you need to show that the number of leaves in any n-item binary heap is at most $\lceil \frac{n}{2} \rceil$. To this end, you might find the answer to Exercise 1 of Week 4 useful. If so, then you can simply refer to that exercise without re-proving it. *Hint 2.* In the inductive step of your proof, consider removing all leaves of the binary heap H that you want to show its number of nodes at height $h + 1$ is at most $\lceil \frac{n}{2^{h+2}} \rceil$. Note that the number of nodes of H at height $h + 1$ is equal to the number of nodes of the binary heap that results after removing leaves of H at height h.]

BASE CASE. At the base case of $h = 0$, for n items, the number of leaves equate to $\lceil \frac{n}{2^{0+1}} \rceil = \lceil \frac{n}{2} \rceil$. This was proven by induction in exercise 1 of week 4.

INDUCTION HYPOTHESIS. Given the induction hypothesis, it is assumed that at a height-level of h in a binary tree of n items, there are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes if all leaf-nodes are removed.

INDUCTIVE STEP. In a binary tree at any level of p parent-nodes, there are at most $2p$ children of these items since one parent can store at most two children. This means that if the height-level h increases to $h + 1$ (so to the height of the level above), the number of nodes n of that level halves.
Now, if all leaf-nodes are removed from H, it has $\lceil \frac{n}{2^{h+1}} \rceil$ nodes at level h. One level higher at $h + 1$, the number of nodes must at most halve, so $\lceil \frac{n}{2^{h+1}} \rceil * \frac{1}{2}$ which reduces to $\lceil \frac{n}{2^{h+1}*2} \rceil = \lceil \frac{n}{2^{h+1+1}} \rceil = \lceil \frac{n}{2^{h+2}} \rceil$. This proves the induction.

QUESTION 4. [2 POINTS] An array A of numbers is said to be k-sorted if every number in A is at most k numbers far from its location if A was sorted. For instance, the array $2, 3, 4, 1, 7, 5, 9, 6, 8$ is 3-sorted, because each of the numbers in the given array has a distance of at most 3 from its location if the array was sorted. Design an efficient algorithm that, given a number $k > 0$ and a k-sorted array A, outputs A in sorted order. Also, do not forget to analyze the running time and space complexities of your algorithm. The running time of your algorithm should depend on both k and n, and the space requirement of your algorithm should depend on k only. [*Hint.* The smallest number is among the first $k+1$ numbers, namely in $A[0 \dots k]$, the second smallest number is in $A[1 \dots k+1]$, the third smallest number is in $A[2 \dots k + 2]$, and so on. Maintain $A[0 \dots k]$—as the set of candidates for the smallest number—in an appropriate data structure and then update the data structure properly as you read the rest of the input.]

Algorithm:

Algorithm 1 Input: A k-sorted array H of n integers.

```
$minHeap$ = construct minHeap()
for $i = 0$ to $k + 1$ do
    $minHeap$.enqueue($H[\ i\ ]$)
end for
for $j = 0$ to $n - (k + 1)$ do
    $H[\ j\ ]$ = $minHeap$.dequeue( root node of $minHeap$ )
    $minHeap$.enqueue($H[\ j + (k + 1)\ ]$)
end for
for $l = n - (k + 1)$ to $n - 1$ do
    $H[\ l\ ]$ = $minHeap$.dequeue( root node of $minHeap$ )
end for
return $H$
```

TIME COMPLEXITY. The designed algorithm returns the array H in sorted order. It first creates an empty min-heap of $k + 1$ elements, since the smallest number is by definition of the k-ordered list H in the first $k + 1$ elements of H. The creation of that heap takes constant $\Theta(1)$ (for the creation) and $\Theta(log(k))$ (for the insertion of k elements) time. When this for-loop ends, another for-loop starts which iterates over the rest of the elements in H (iterating over those items takes $\Theta(n)$ time). At each iteration of the latter loop, the lowest element (which sits at the root) of the min-heap is dequeued and the H-array is updated at the correct position. After this step (still in the same for-loop), the next item of H is enqueued into the into the min-heap structure. This for loop takes $\Theta(n * log(k))$ time; $\Theta(n)$ for iterating over each missing item and $\Theta(log(k) + log(k))$ for the en- and dequeuing action respectively. The last for loop starts when the last item of the array H has been enqueued into the heap-structure; it iterates over the heap-structure until it is empty by dequeuing at every iteration (which takes $\Theta((k+1) * log(k))$ time). Overall, the time complexities reduce to $\Theta(n * log(k))$.

SPACE COMPLEXITY. The designed algorithm creates a min-heap data structure with a length of $k+1$ elements. Other than that, there is nothing new stored but instead, the given array is simply updated. Overall, this makes for a space-complexity of $\Theta(k)$.