Comp 251: Mid-term examination #1

Instructor: Jérôme Waldispühl and David Becerra

September 30, 2020

- Answer the questions inside the provided space.
- Unless specified, all answers must be explained.
- Partial answers will receive credits.
- The clarity and presentation of your answers is part of the grading. Answers poorly presented may not be graded. This includes the clarity of the writing.
- This is an open book examination.
- This exam contains 10 pages.

Full name: _____

Student ID: _____

Question:	1	2	3	4	5	6	Total
Points:	20	20	26	16	10	8	100
Score:							

Short answers

1. True or False? Circle your answers. No justification. **Wrong answers will receive a penalty of -1**.

   (a) (2 points) The maximum capacity (i.e. number of nodes) of a heap of height h is 2^h.
   A. True **B. False**

   (b) (2 points) Performing one rotation always preserves the AVL property.
   A. True **B. False**

   (c) (2 points) In a red-black trees, at least half of the nodes in a path from the root to a leaf are black.
   A. True B. False

   (d) (2 points) The order of the vertices visited by the depth-first search algorithm (DFS) algorithm is always different than the order of the nodes returned by the breadth-first search algorithm (BFS) algorithm.
   A. True **B. False**

   (e) (2 points) A recursive algorithm has at most one base case.
   A. True **B. False**

   (f) (2 points) Merge sort is always faster than Insertion sort.
   A. True **B. False**

   (g) (2 points) In a proof of correctness of a recursive algorithm, the base case is equivalent to the termination condition.
   A. True **B. False**

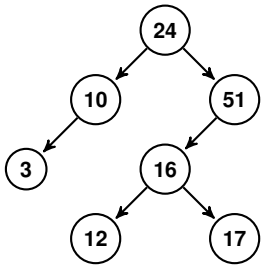
   (h) (2 points) A loop invariant is a property satisfied before and after each iteration of a loop.
   A. True B. False

   (i) (2 points) For a specific algorithm, its best case running time is always asymptotically strictly lower than its worst case running time.
   A. True **B. False**

   (j) (2 points) If two algorithms have the same worst-case running time $\mathcal{O}(n^2)$, then in the worst case they will perform the same number of primitives operations.
   A. True **B. False**

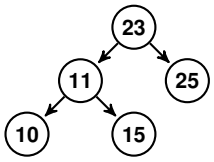
AVL trees

2. (a) (4 points) Consider the AVL tree below. Indicate which node violates the AVL property (give the key used to label the node). In one sentence, justify your answer.

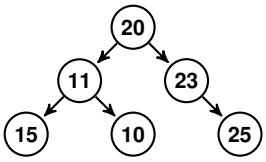


Solution: Node with key 51. The difference of height between the left and right subtrees is strictly larger than 1 (Reminder: we set the height of an empty tree to -1).

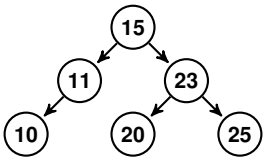
(b) (6 points) Consider the trees below. We want to insert a new key with value **20** in the first tree of this list (i.e. the tree labelled (A)), which is an AVL tree.



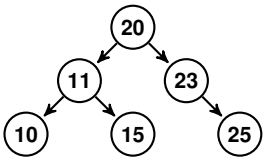
(A)



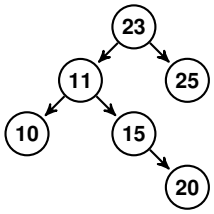
(B)



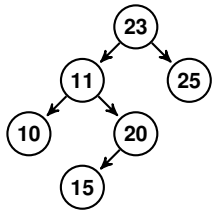
(C)



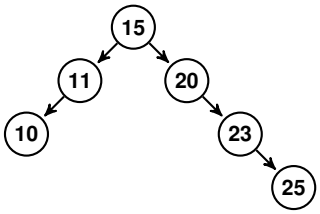
(D)



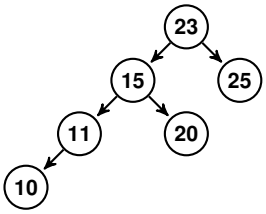
(E)



(F)



(G)



(H)

Give the sequence of trees that would result from the series of basic operations (i.e. BST insertion, rotation left and rotation right) as they would be executed by the AVL insertion method seen in class. For each tree, write the operation that has been executed (i.e. BST insertion, rotation left and rotation right) and the node used (e.g. root of the subtree for a rotation).

Solution: A (initial tree), E (BST insertion of 20), H (rotation left at node 11), C (rotation right at 23).

(c) (6 points) Consider two AVL trees `A` and `B` such that all keys of `A` are lower or equal to the keys of `B`. Assume that these two AVL trees `A` and `B` have the same heights. Propose a method `mergeAVL(A,B)` to merge 2 AVL trees `A` and `B` into a single valid AVL tree. Your solution should be as efficient as possible (i.e. with the lower worst case running time possible). You can write your solution as a pseudo-code or a series of instructions. Be as concise a possible. No justification needed (i.e. you do not need to provide the technical details of each operation).

Solution:

1. remove the rightmost node x of A. Alternatively, you can also use the leftmost node of B. Note: it may require a bit of work if x is an internal node (not covered in class).
2. use this node as the new root, such as A becomes left subtree of x and B the right subtree of x.
3. One conflict can occurs in the sub-tree in which you remove the node. Fix this conflict using the similar principles as delete if it occurs.

Note: There's an alternate and more general solution that would consist to merge the arrays and heapify. It would require more operations. though.

(d) (2 points) What is the worst case running time of this algorithm (Hint: You can assume that the worst case running time for deleting a node in an AVL tree is $O(\log n)$). Show your work.

Solution: $O(\log n)$. $O(\log n)$ for removing the node + $O(1)$ to create new tree.

(e) (2 points) Why is it important that all keys of the AVL tree `A` are lower or equal to the keys stored in `B` (or the opposite)?

Solution: Otherwise, we cannot guarantee that the merged tree satisfy the BST properties.

Binary search

3. Recall the recursive version of the binary search algorithm to find a key x in a sorted array A. The algorithm returns the index of the key if it finds it and -1 otherwise.

Algorithm 1 bsearch(int[] A, int i, int j, int x)

```
1:  if i <= j then
2:      e ← ⌊(i+j)/2⌋
3:      if A[e]>x then
4:          return bsearch(A,i,e-1,x)
5:      else if A[e]<x then
6:          return bsearch(A,e+1,j,x)
7:      else
8:          return e
9:      end if
10: else
11:     return -1
12: end if
```

(a) (4 points) First, we want to design a variant of this algorithm (`firstkey(A,i,j,x)`) that returns the lowest index of a key (i.e. the index of the first occurence in the sorted array). For instance, if we search for the key 3 in the array $[1, 3, 3, 3, 6, 8]$ it will return 1 (i.e. the indices start at 0). Explain briefly (3 sentences max), how you can modify the binary search algorithm above to solve this problem.

Solution: Do not stop when you find the key. Once we found the key, we continue to search on the left.

(b) (4 points) Indicate which line(s) of the algorithm above you would need to modify to implement your solution and briefly explain the changes. Note: If needed, you are allowed to update the signature of the function (e.g. add one variable to the list of arguments).

Solution:

- add one variable (say k) in the signature that stores the latest index the key was found (-1 by default). bsearch(int[] A, int i, int j, int x, int k)
- line 8 should be return bsearch(A,i,e-1,x,i) (last variable is k).
- line 11 should return the last index the key was found (i.e. k).

(c) (4 points) What is this worst case running time of this algorithm (`firstkey(A,i,j,x)`)? Justify.

Solution: $O(\log n)$. Same as binary search since the worst case is the same as binary search finding the key at the last iteration.

(d) (4 points) How would you change this algorithm to search for the largest index this time? Give

the worst case running time complexity again. No justification needed.

Solution: Once we found the key, we continue to search on the right. $O(\log n)$.

(e) (6 points) Next, we want to design a novel (and efficient) algorithm (`countkey(A,i,j,x)`) that computes the number of times a key is found in a sorted array. For instance, if we search for the key 3 in the array $[1, 3, 3, 3, 6, 8]$ it will return 3 (i.e. the key 3 occurs 3 times in the array). This algorithm **must** have the best worst case running time complexity possible. Briefly explain (3 sentences max) the principle of this algorithm.

Solution: Find the first and last occurrence of the key and return the difference.

(f) (4 points) What is this worst case running time of this algorithm (`countkey(A,i,j,x)`)? Briefly explain your answer.

Solution: $O(\log n)$. Finding the first and last occurrence of the key costs $O(\log n)$ for each, the computing the difference of the index is $O(1)$. Thus, the total complexity is $O(\log n) + O(\log n) + O(1) = O(\log n)$

Heapsort

4. (a) (6 points) Consider the following array $[3, 4, 8, 9, 12, 15]$. Build a Max Heap from this array using the algorithm seen in class. Show your work (i.e. your solution should include one array for each step and specify on which nodes you call the heapify function. Indicate the swaps as well). **You must use the array notation** (i.e. you must not draw trees).

Solution:
$[3, 4, 8, 9, 12, 15]$
$[3, 4, 15, 9, 12, 8]$ // heapify index 2
$[3, 12, 15, 9, 4, 8]$ // heapify index 1 (swap 12 and 4)
$[15, 12, 3, 9, 4, 8]$ // heapify index 0 (swap 3 and 15)
$[15, 12, 8, 9, 4, 3]$ // heapify index 0 (swap 3 and 8)

(b) (10 points) You will show the execution of the Heapsort algorithm on the array $[12, 6, 8, 1, 4, 7]$. This array has already been process and satisfy the heap properties. Show all steps of the algorithm and remember that you should sort-in-place (i.e. indeed you do not need any more space than what is already occupied by the array). At the end, the array should be sorted in increasing order of the keys. **You must use the array notation** (i.e. you must not draw trees).

Solution:
$[7, 6, 8, 1, 4], [12]$
$[8, 6, 7, 1, 4], [12]$
$[4, 6, 7, 1], [8, 12]$
$[7, 6, 4, 1], [8, 12]$
$[1, 6, 4], [7, 8, 12]$
$[6, 1, 4], [7, 8, 12]$
$[4, 1], [6, 7, 8, 12]$
$[1], [4, 6, 7, 8, 12]$
$[], [1, 4, 6, 7, 8, 12]$

Big-Oh notation

5. (a) (5 points) Show that $\frac{1}{2} \cdot n^2 + 8 \cdot \sqrt{n} + 1$ is $O(n^2)$. Find the two constants c and n_0 and demonstrate that the Big O definition is satisfied.

Solution:

(b) (5 points) Someone claimed that "If $f(n) = O(g(n))$ then $g(n) = O(f(n))$"... Is it true or false? Justify your answer, either by proving it is correct, or if it is not by showing a counterexample.

Solution: This is false as demonstrated by the counterexample, $f(n) = n$ and $g(n) = n^2$. Note that $n = O(n^2)$ but $n^2! = O(n)$.

Loop invariant

6. Consider the following algorithm:

Algorithm 2 mystery(int[] A)

```
1: n ← len(A)
2: for k ← 1 to n − 1 do
3:     if A[k]<A[k-1] then
4:         A[k] ↔ A[k − 1]
5:     end if
6: end for
7: return A[n − 1]
```

Note 1: the instruction $A[k] \leftrightarrow A[k-1]$ indicates that we swap the values at index k and $k-1$.
Note 2: The **for** loop is inclusive (i.e. the last value taken by k is $n-1$).

We want to prove the following loop invariant property: *"The key at index k is the largest that can be found in the (sub-)array $A[0:k]$"*.

(a) (2 points) Prove the initialization step.

Solution: When $k = 1$ either $A[1]$ is already larger than $A[0]$ or we swap it.

(b) (4 points) Prove the Maintenance.

Solution: Assume that $A[k-1]$ is the largest key of $A[0:k-1]$. Then, either $A[k]$ is already larger than $A[k-1]$ and thus than all keys in $A[0:k-1]$. Or we swap $A[k]$ and $A[k-1]$ and $A[k]$ becomes the largest of $A[0:k]$.

(c) (2 points) Prove the termination step.

Solution: We make a finite number of iteration (n-1).