University of Waterloo
CS240 Spring 2024
Assignment 3

Due Date: Tuesday, June 18 at 5:00pm

Please read the following link for guidelines on submission:

`https://student.cs.uwaterloo.ca/~cs240/w24/assignments.phtml#guidelines`

Each question must be submitted individually to MarkUs as a PDF with the corresponding file names: `a3q1.pdf`, `a3q2.pdf`, ... It is a good idea to submit questions as you go so you aren't trying to create several PDF files at the last minute.

Late Policy: Assignments are due at **5:00pm**, with the grace period until 11:59pm.

Problem 1 [6 marks]

Generalize *quick-select* (Module 3 - Slide 13) to work on two input arrays. Let the resulting algorithm be called *quick-select-2arrays*(A, B, k). Arrays A and B are of size n and m, respectively, and $k \in \{0, 1, ..., n + m - 1\}$. Algorithm *quick-select-2arrays*(A, B, k) should return the item that would be in $C[k]$ if C was the array resulting from merging arrays A and B and C was sorted in non-decreasing order.

Your algorithm *quick-select-2arrays*(A, B, k) must be in-place, i.e. only $O(1)$ auxiliary space is allowed per recursive function call. Briefly and informally (one or two sentences) argue that the time complexity of your algorithm is the same as of *quick-select*, i.e., $O(v)$ in the average case where v is the total number of elements in A and B, i.e., $v = n + m$. Hint: use the same pivot-value for partitioning both arrays.

Problem 2 [2+3+3=8]

A clever student (let's call him Max) thinks he can avoid the worst-case behaviour of *quick-sort* by employing the following pivot-selection procedure. First, compute the mean $\bar{M}$ of the elements in the array. Then choose as the pivot the element x of the array, such that $|x - \bar{M}|$ is minimized, i.e., pick the element closest to the average value in the array. Everything else is the same as *quick-sort*. He calls the modified *quick-sort* algorithm *MX-sort*.

a) Write down the recurrence for running time $T(n)$ of *MX-sort*. In doing so, assume x is placed at index i of the partitioned array. The recurrence relation may be expressed in terms of n and i.

b) Assume that the elements of the array form an arithmetic sequence (i.e., have the form a, $a + k$, $a + 2k$, $a + 3k$, $\ldots$, $a + (n - 1)k$), scrambled in some order. Show that, under this distribution of array elements, *MX-sort* always runs in $\Theta(n \log n)$ time.

c) Unfortunately, Max's scheme is not as clever as it looks. Give an example of an array where *MX-sort* achieves its worst case runtime of $\Theta(n^2)$ and briefly explain why this example requires this time.

Problem 3 [8 marks]

Given an array $A[0 \ldots n-1]$ of numbers, such that $A[i] \geq A[i-j]$ for all $0 \leq i \leq n-1$ and $\log n \leq j \leq i$, design an algorithm to sort A in $O(n \log \log n)$ time.

Hint: Partition A into contiguous blocks of size $(\log n)$; i.e. the first $(\log n)$ elements are in the first block, the next $(\log n)$ elements are in the second block, and so on. Then, establish a connection between the elements within two blocks, which are separated by another block.

Problem 4 [2+2+4=8 marks]

Consider the problem of finding the location of a given item k in an array of n distinct integers. The following randomized algorithm selects a random index and checks whether its entry is the desired value. If it is, it returns the index; otherwise, it recursively calls itself.

Recall that *random(n)* returns an integer from the set of $\{0, 1, 2, \ldots, n-1\}$ uniformly and at random.

```
find-index(A,n,k)
   i = random(n)
   if A[i] = k then
      return i
   else
      return find-index(A,n,k)
   end if
```

In your answers below, be as precise as possible. You may use order notation when appropriate. Briefly justify your answers.

a) What is the **best-case** running time of `find-index`?

b) What is the **worst-case** running time of `find-index`?

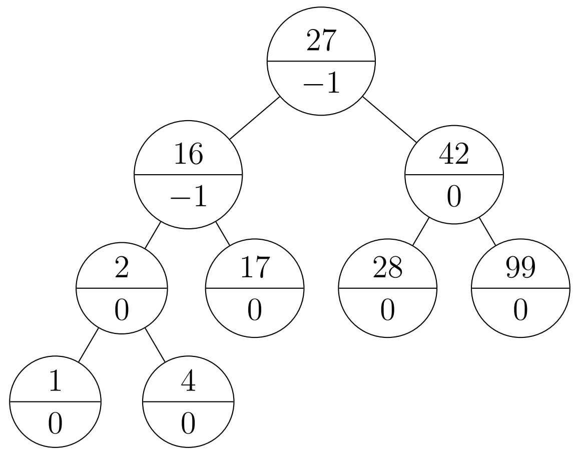
c) Let $T(n)$ be the expected running time of `find-index`.
Write a recurrence relation for $T(n)$ and then solve it.

Problem 5 [0+2+2=4 marks]

a) **Practice** (not worth any marks): Starting with an empty AVL tree, insert the following keys in order: 27 99 17 28 42 16 1 2 4.

You should obtain the AVL tree given in the next part.

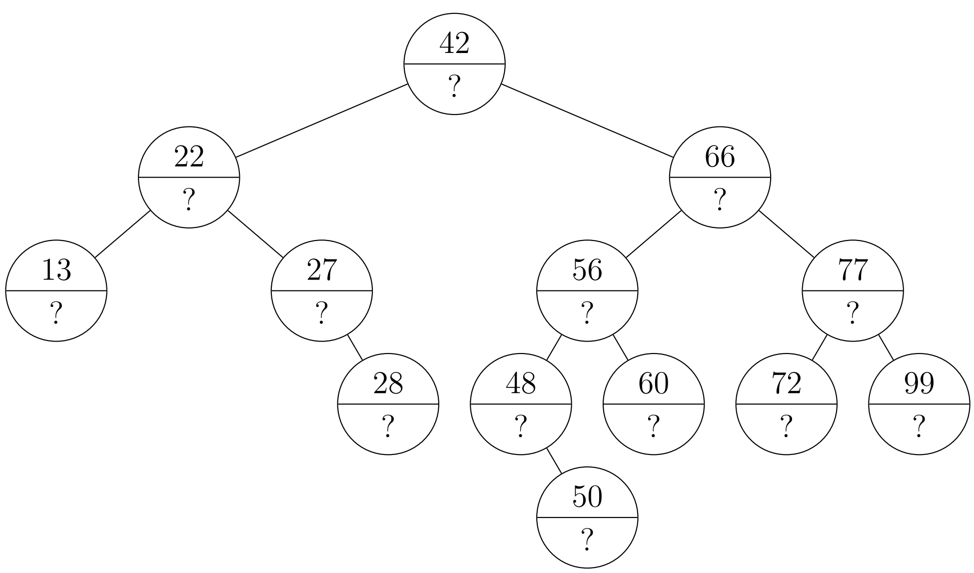
b) Given the following AVL tree:
Note: this tree shows balance factors instead of height.



Insert the following keys in order: $8^\star$, 22, 21, $18^\star$.

Show the resulting AVL trees with **balance factors** (not height) for each node after the elements marked with star ($\star$) are inserted.
Note: you should only show 2 trees.

c) Consider the following AVL tree:



Given the above tree, delete the following keys in order:

$$66,\ 13^\star,\ 72,\ 77,\ 56^\star,\ 42^\star$$

Show the resulting AVL trees with **balance factors** (not height) for each node after the elements marked with star ($\star$) are deleted. If you have a choice of which element to move up, pick the inorder successor.
Note: you should only show 3 trees.

Problem 6 AVL Trees [4+6 marks]

In this question, we want to modify an AVL tree to support an operation `ithSuccessor`, in addition to the standard operations `insert, delete, find`. The operation `ithSuccessor` has two parameters, x and $i \geq 0$, and returns the ith inorder successor of the node x. If $i = 0$, then the node x itself is returned. You may assume that all input is valid; i.e. the successor exists (but may not be in the subtree rooted at x).

We assume that the nodes have the following fields:

- `key` – the key of the node;
- `left` – pointer to the left child;
- `right` – pointer to the right child;
- `balance` – balance factor of the node;
- `parent` – pointer to the parent of the node;
- `isLeft` – is true if the node is a left child of its parent;
- `isRight` – is true if the node is a right child of its parent;
- `numLeft` – holds the number of nodes in the left subtree of the node;
- `numRight` – holds the number of nodes in the right subtree of the node.

a) Give an algorithm `ithNode(x, i)` which returns the ith inorder node in the subtree rooted at x. For example, suppose the subtree contains m nodes, when $i = 1$, the minimum element in the subtree is returned and when $i = m$ the maximum element in the subtree is returned. You may assume that the subtree has at least i elements. Your algorithm should take worst-case $O(\log(m))$ time. Briefly justify that your algorithm achieves this runtime.

b) Give the algorithm for `ithSuccessor(x, i)` if n is the number of nodes in the given AVL tree. Your algorithm should take worst-case $O(\log(n))$ time and must use `ithNode(x, i)` from above. Briefly justify that your algorithm achieves this runtime.