University of Waterloo
CS240E, Winter 2024
Assignment 2

Due Date: Tuesday, February 6, 2024 at 5:00pm ~~Wednesday, February 15, 2023, at 5pm~~

Be sure to read the assignment guidelines (`http://www.student.cs.uwaterloo.ca/~cs240e/w23/guidelines/guidelines.pdf`). Submit your solutions electronically as individual PDF files named a2q1.pdf, a2q2.pdf, ... (one per question).

For some questions you may find helper-routines from predecessor-courses helpful. If the helper-routine is in the course notes (even if we have not gone over it), you may use it without giving details.

Also, a reminder: we **cannot compute logarithms or exponentiation in constant time**; we compute these from scratch as we need them.

Grace period: submissions made before 11:59PM on Feb. 6, will be accepted without penalty. Please note that submissions made after 11:59PM **will not be graded** and may only be reviewed for feedback.

Question 1 [6 marks]

Let $A[0..n-1]$ be an unsorted array that stores integers, where each entry is in the range $[0, n^5)$. Entries in A are not necessarily unique. Design an algorithm to test whether there are indices i and j such that $|A[i] - A[j]| = 10$. The worst-case run-time and the auxiliary space must be in $O(n)$.
Remark: You are *not* allowed to use hashing for this question. We have not even covered hashing yet, but even if we had, its worst-case run-time is **not fast enough**.

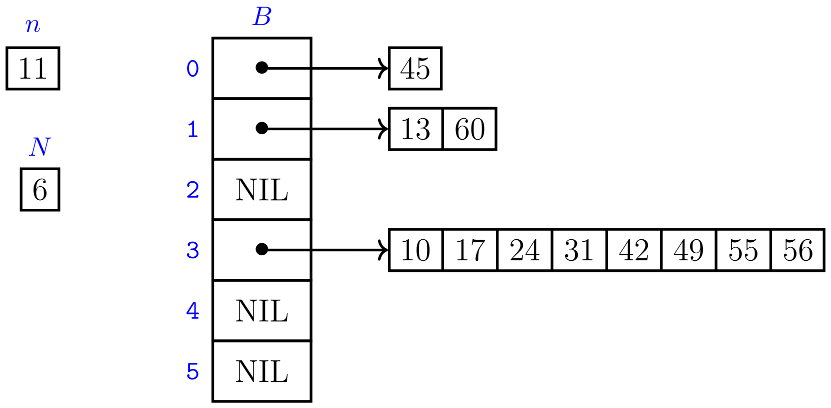
Question 2 [5+3=8 marks]

a) Assume that you are given a non-empty list L that contains n key-value pairs in sorted order, where $n + 1$ is a Fibonacci-number. Design an algorithm to build an AVL-tree that contains the key-value pairs from L and has the maximum possible height. (In other words: build a Fibonacci-tree containing the items of L.) The run-time and the auxiliary space must be $O(n)$.

b) Prof. Quirky thinks that he can do the above with a comparison-based algorithm even if list L is in unsorted order. Show that this is not possible.

Question 3 [4+1+6+6(+5)=17(+5) marks]

Consider the following realization (called *buckets of sorted arrays*) of ADT Dictionary. We have an array $B[0 \dots N{-}1]$ for some $N \geq 0$. Entry $B[j]$ is either NIL, or it stores an array of size exactly 2^j, with items in sorted order. The following is an example.



a) Design an algorithm that does *search*(k) in buckets of sorted arrays in $O(\log^2 n)$ time. You may assume that you know n and N, but N could be arbitrarily large relative to n. You need not argue correctness, but explain your idea well. Recall: For this assignment you can **not** compute log() in constant time.

b) Insert key 40 into the example and show the resulting B. There are multiple possible answers; show the one that moves as few items as possible into a different bucket.

c) Design an algorithm to do *insert*(k, v) in such a way that as few items as possible are moved. Your algorithm must have worst-case run-time $O(1+s)$, where s is the number of items that were moved. You need not argue correctness. You may assume that N is sufficiently big so that the new item will fit.

d) Define $\Phi := \sum_{j=0}^{N} \sum_{x \in B[j]} (N - j)$. Briefly verify that this is a potential function, and then show that with this potential function (and a suitable choice of time units), the amortized time of *insert* is $O(\log n)$. For this part you may assume that $N \in O(\log n)$.

e) **(Bonus)** Design a version of *insert*(k, v) that does *not* assume that N is sufficiently big (i.e., if needed, it increases the capacity of B and copies over if needed). Argue that the amortized run-time for *insert* is still $O(\log n)$. Clearly state what your potential function is (finding it is part of the problem), and by how much you increase the capacity of B when copying over.

While a correct answer to e) likely contains correct answers to c) and d), for ease of grading please write separate answers to c) and d) and refer to them in here as needed.

Question 4 [4+2+5+3+4=18 marks]

Motivation: Scapegoat trees as defined in class store at each node v the size of the subtree rooted at v. This is not actually required; this assignment will guide you towards a variation that operates without storing the size of the subtree.

Define a *light scapegoat tree* to be a binary search tree T that is *height-balanced*, i.e.,

$$height(T) \leq \lfloor \log_{4/3} n \rfloor .$$

You may assume that a light scapegoat tree knows its size n. It does not store its height, and its nodes know neither the size nor the height of their subtrees. You may assume that nodes have parent-references (the parent of the root is NIL).

Consider the following (incomplete) code to insert in such a tree:

Algorithm 1: *LightScapegoatTree::insert*(k, v)

```
// Current tree is height-balanced
1 $z \leftarrow BST::insert(k, v)$
2 if $notHeightBalanced(z)$ then
3     $p \leftarrow lowestSmallAncestor(z)$
4     completely rebuild the subtree at $p$ as a perfectly balanced subtree
```

a) Design algorithm *notHeightBalanced*(z). This must achieve the following: you are given a newly inserted node z that is a leaf, and you must determine whether the current binary search tree is still height-balanced. The run-time must be $O(\log n)$. Recall: For this assignment you can **not** compute log() in constant time.

b) Show that if the tree is not height-balanced after *BST::insert*, then there exists an ancestor p of z such that

$$size(p) < \left(\tfrac{4}{3}\right)^{d(p,z)} .$$

Here $size(p)$ denotes the number of items in the subtree rooted at p. (We write this as a function, rather than as a field, to emphasize that this is *not* stored with the tree.) For any ancestor v of z, $d(z, v)$ denotes the distance from z to v, i.e., the number of levels that v is above z. (So $d(z, z) = 0$, $d(parent(z), z) = 1$, etc.).

c) Design algorithm *lowestSmallAncestor*(z). This must achieve the following: You are given the newly inserted node z, and you know that the tree is now not height-balanced. You must find a node p such that $size(p) < \left(\tfrac{4}{3}\right)^{d(p,z)}$. Furthermore, your p must be the lowest ancestor of z that satisfies this, i.e., must minimize $d(p, z)$. The run-time must be $O(size(p) + \log n)$.

Hint: Recall that you do *not* have the sizes of subtrees stored at each node. But in time $O(size(p))$, you can compute the size of p. How can you organize this so that over all ancestors of z (until you have found the right one) you only spend time $O(size(p))$?

d) Let p be the node returned by *lowestSmallAncestor*. Show that after rebuilding the subtree at p to be perfectly balanced, the resulting binary search tree is height-balanced.

e) Let p be the node returned by *lowestSmallAncestor*. Show that before rebuilding the subtree at p we had

$$|\mathit{size}(p.\mathit{left}) - \mathit{size}(p.\mathit{right})| \in \Omega(\mathit{size}(p)).$$

For all parts, you may use results of previous parts even if you did not prove them.

Continuing the "motivation": We are *not* asking you do show that *LightScapegoatTree::insert* has $\Theta(\log n)$ amortized time, but it would be very easy to do so using the potential function from class and taking into account part (e) as well as the run-times that you have achieved.