**Due: Tuesday, February 28, 2023, 11:59pm**

Submit online at (clickable link): `https://forms.gle/nT1gLPDT9weu6NqMA`

## 1 A balanced binary-search tree

The elegance of red-black trees and splay trees may sometimes leave us with a sense of awe. How did they come up with such a nifty scheme to keep the binary search tree balanced? It is actually not that hard to balance a binary search tree. Many simple schemes can give us $\Theta(\log n)$ amortized running time. They may be lacking in elegance, but quite easy to understand.

Consider the following scheme. First, we store at every node $x$ the number of items in the subtree rooted $x$. This allows us to determine when the binary search tree becomes unbalanced. Let's define a node $x$ to be *horribly unbalanced* if the left subtree of $x$ has $\geq 5$ times the number of nodes than in its right subtree, or vice versa. All we will do in this scheme is to "fix up the subtree" if we detect that $x$ is horribly unbalanced.

1. Suppose that none of the nodes in a binary search tree $T$ with $n$ items is horribly unbalanced. Argue that the height of $T$ is $O(\log n)$. *Hint:* use recursion/induction.

2. Suppose that during an insertion, we detect that some node $x$ is horribly unbalanced. We can fix the subtree rooted at $x$ by taking apart the subtree and adding every item from this subtree into a balanced tree that is as balanced as possible. Describe how to accomplish this re-balancing in $O(m)$ worst case actual time, where $m$ is the number of items in the subtree rooted at $x$.

   *Note:* Do not use pseudocode to describe your algorithm. For example, if you want to sort some numbers in an array $A$, just say "Sort the values in $A$."

3. (revised) When you a insert a new key $k$, you may encounter several nodes that are horribly unbalanced nodes on the path from the root to where $k$ gets inserted. Where should you perform a re-balance first? at the horribly unbalanced node closest to the root? or closest to $k$? Justify your response.

4. Devise an amortized analysis for this data structure using the accounting method. Your analysis must show that the amortized running time of insert and search are $O(\log n)$.

   *Note:* You must state an invariant for your accounting scheme and argue that the invariant is maintained after each insert and search operation, including operations that perform re-balancing operations.

## 2 Sorted Array with Sidekick

Storing items in a sorted array has the advantage that binary search can find an item in $\Theta(\log n)$ time. However, insertion into a sorted array is cumbersome and takes $\Theta(n)$ time.

A Sorted Array with Sidekick (SAwS) combines a sorted array with a smaller unsorted array, the sidekick. New items inserted in a SAwS are added to the sidekick array, which takes $O(1)$ actual time since the sidekick is not sorted. When the number of items in the sidekick grows beyond $\sqrt{n}$, the sidekick is sorted and merged into the sorted array. After this merger, the new sidekick is empty. To search in a SAwS simply do a binary search in the main array and a linear search in the sidekick.

**Notes:**

- Throughout this problem, let $n$ be the number of items currently stored in the SAwS. This means $n$ changes when items are added to or removed from the SAwS. You should analyze the running times in terms of $n$.
- Just assume that the arrays have enough space allocated. It is possible to combine this data structure with dynamic tables, but let's keep it simple.
- (revised) Again, for simplicity, just assume that in our use case we will never insert duplicate keys into this data structure. (It is possible to handle duplicates, but it gets rather complicated when we delete a key that has duplicates still in the SAwS.)
- You know nothing about the type, distribution or range of the keys. So you cannot use counting sort, bucket sort, radix sort or any linear-time sorting algorithm.
- You may assume without further comment that in the worst case sorting $m$ items takes $\Theta(m \log m)$ time, binary search in a sorted array of $m$ items takes $\Theta(\log m)$ time, finding the median of an unsorted array of $m$ items takes $\Theta(m)$ time and merging two sorted arrays of $m$ and $\ell$ items takes $\Theta(m + \ell)$ time.
- Do not write pseudo-code. If you want to sort an array $A$, just say "Sort the array $A$."

**Questions:**

1. Briefly describe how the sidekick array can be merged into the sorted array in $\Theta(n)$ actual time.
2. Using the accounting method, show that the insert and search operations in a SAwS can be accomplished in $O(\sqrt{n})$ amortized time. (Ignore the delete operation for now.)

   *Note:* State an invariant for your accounting scheme and argue that the invariant is maintained after each insert and search operation, including operations that merge the sidekick into the sorted array.
3. Describe how to implement the delete operation in a SAwS so that search, insert and delete each takes $O(\sqrt{n})$ amortized time.

   The delete operation is given a key $x$ and must find then remove $x$ from the SAwS data structure. For sorted arrays, it is convenient to simply mark the location of $x$ as "removed". (Assume the programming language supports this.) For unsorted arrays, you can either mark $x$ as removed or move the last item to the vacated location. However, $n$ is still the number of items in the SAwS and does not include the locations marked "removed".

   Note that a delete makes either the main array or the sidekick array smaller by 1 and that repeated deletions reduce the values of $n$ and $\sqrt{n}$ significantly. In particular, the size of the sidekick array might exceed $\sqrt{n}$ not because we added items to the sidekick, but because the value of $n$ is reduced.

   *Note:* You should revise the invariant for your accounting scheme and argue that the invariant is maintained after each insert, delete and search operation, including operations that merge the sidekick into the sorted array.