CSCI 0500: Data Structures, Algorithms, and Intractability (Fall 2026)

Assignment 4

Due at 11:59pm ET, Friday, Oct 23

1. (1 point) Consider the problem of finding the k-th smallest element in a dynamic set.

   For simplicity, suppose all numbers are distinct. In both parts, let n denote the number of elements currently stored in the data structure. You can use any data structures discussed in class without implementing them. You do not need to prove the correctness or analyze the runtime of your algorithms.

   (a) Design a deterministic data structure that supports the following operations:
   - `insert(x)`: add the number x to the data structure.
   - `find_median()`: return the median of all numbers in the data structure. If there are an even number of elements, return the smaller of the two middle values. You can assume there is at least one element in the data structure.

   Each operation should have worst-case runtime $O(\log n)$.

   (b) Design a (possibly randomized) data structure that supports the following operations:
   - `insert(x)`: add the number x to the data structure.
   - `find_kth_element(k)`: return the k-th smallest number in the data structure. You can assume there are at least k elements in the data structure.

   Each operation should have worst-case (expected) runtime $O(\log n)$.

2. (1 point) In this question, we study Fibonacci heaps, a data structure that implements a priority queue. We consider a simplified version that supports only the following operations:
   - `insert(x)`: insert a new item with priority x.
   - `extract_min()`: remove and return the item with the smallest priority.

   For simplicity, we ignore the associated data and assume all priorities are distinct.

   A Fibonacci heap is a collection of (not necessarily binary) trees that satisfy the min-heap property: the priority of a child is at least the priority of its parent. We maintain a pointer to the node with minimum priority, which must be the root of one of the trees.

   Insertion takes $O(1)$ worst-case time: `insert(x)` creates a single-node tree with priority x, adds it to the collection, and updates the minimum pointer if needed.

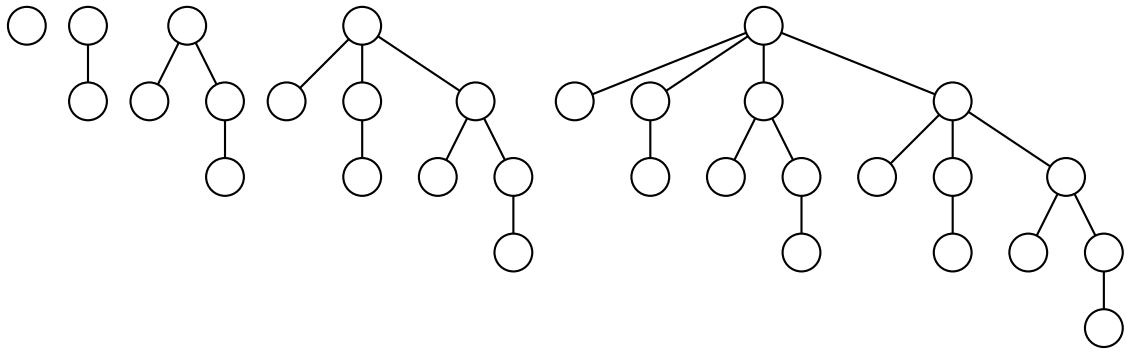
   The degree of a tree is the number of children of its root. A key idea of Fibonacci heaps is to merge trees during `extract_min` so that afterward there is at most one tree of each degree. The `extract_min` operation has three steps:

   1) Remove the minimum node and add all of its children as new trees.

2) While there are two trees with the same degree, merge them by making the root with larger priority the rightmost child of the root with smaller priority.
3) Update the minimum pointer.

Definition 1 (Binomial Trees)**.** A binomial tree of degree 0 is a single node. A binomial tree of degree k has a root whose children are binomial trees of degree $0, \ldots, k-1$ (in this order).

The figure below shows binomial trees of degrees $0, \ldots, 4$.



Claim 2. *In this simplified version of Fibonacci heaps, all trees are binomial trees.*

Proof. We show that every operation preserves the property that all trees are binomial trees. The `insert` operation adds a single-node tree, which is a binomial tree of degree 0. The `extract_min` operation removes the minimum node in Step 1, leaving child subtrees that are binomial trees by definition. In Step 2, merging two binomial trees of degree k gives a binomial tree of degree $k+1$, because one root becomes the rightmost child of the other. □

(a) Prove that after an `extract_min` operation, there are $O(\log n)$ trees and each tree has degree $O(\log n)$, where n is the heap size after the operation.
(Hint: Prove by induction that a binomial tree of degree k has exactly 2^k nodes.)

(b) Prove that `extract_min` can be implemented to run in $O(a + \log n)$ time, where a is the number of insertions since the previous `extract_min`, and n is the heap size before this `extract_min`.
(Hint: The heap had at most n nodes after the previous `extract_min`. By Part (a), the heap had $O(\log n)$ trees, each of degree $O(\log n)$. After a insertions, the heap has $O(a + \log n)$ trees, each of degree $O(\log n)$. Consider how each step of `extract_min` changes the number of trees and analyze its runtime.)

Consequently, starting from an empty Fibonacci heap, any sequence of a `insert` and b `extract_min` operations takes $O(a + b \log n)$ total time, where n is the maximum heap size.

3. (1 point) In this question, we study AVL trees, a type of binary search tree that guarantees worst-case $O(\log n)$ height, where n is the number of nodes in the tree.

Definition 3 (Depth and Height)**.** The depth of a node v is the number of nodes on the path from v to the root. The height of a tree is the maximum depth of its nodes. [1]

For example, the height of an empty tree is 0, and the height of a single-node tree is 1.

Definition 4 (Balance Factor)**.** For a node v in a binary tree, the balance factor of v is $\mathrm{BF}(v) = (\text{height of } v\text{'s right subtree}) - (\text{height of } v\text{'s left subtree})$. [2]

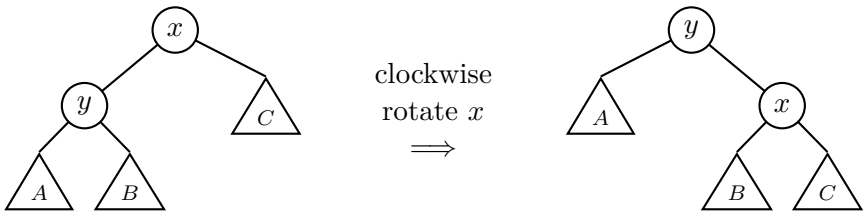
An AVL tree is a binary search tree where $\mathrm{BF}(v) \in \{-1, 0, 1\}$ for all nodes v. That is, the heights of the two child subtrees of any node differ by at most 1.

(a) Prove that the height of an n-node AVL tree is $O(\log n)$.
(Hint: Let f_h be the minimum number of nodes in an AVL tree of height h. Derive a recurrence for f_h in terms of f_{h-1} and f_{h-2}, and use it to show that f_h grows at least exponentially in h.)

For simplicity, we focus on how an AVL tree maintains balance during insertion. We first insert the new key as in a standard binary search tree. We then traverse upward from the inserted node toward the root. If we reach a node with balance factor -2 or 2, we rebalance it using rotations and stop.

Let x be the first unbalanced node reached during the upward traversal. By symmetry, we can assume $\mathrm{BF}(x) = -2$, so the left subtree of x is taller. Let y be the left child of x. Because x is the lowest unbalanced node, we have $\mathrm{BF}(y) \in \{-1, 0, 1\}$.

Case 1: $\mathrm{BF}(y) = -1$. A single clockwise rotation at x restores balance, as illustrated below. The triangles labeled A, B, and C represent (possibly empty) subtrees.

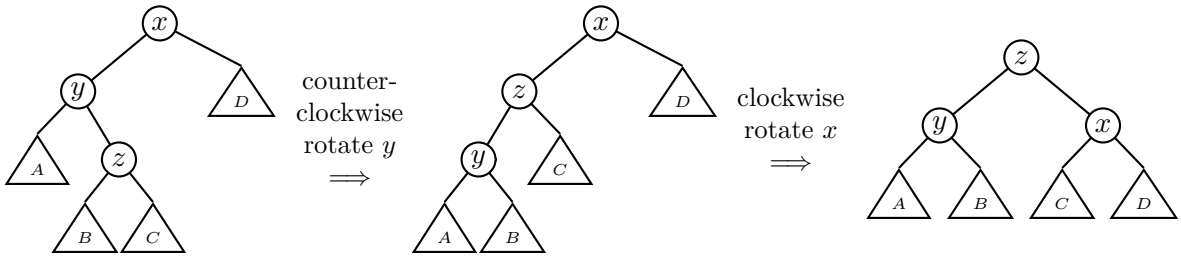


(b) Prove that after the rotation, the balance factors of x and y are in $\{-1, 0, 1\}$. Also prove that the subtree originally rooted at x has the same height as before the insertion.
(Hint: Let $h(V)$ be the height of subtree V, and let $h(v)$ be the height of the subtree rooted at v. Let $k = h(C)$. Because $\mathrm{BF}(x) = -2$ and $\mathrm{BF}(y) = -1$, we have $h(y) = k+2$, $h(A) = k + 1$, and $h(B) = k$. Let h' denote heights after the rotation. What are the values of $h'(x)$ and $h'(y)$? What was the height of the subtree rooted at x before the insertion?)

Case 2: $\mathrm{BF}(y) = 0$. This case cannot occur during insertion. For x to become unbalanced, the subtree rooted at y must increase in height. If $\mathrm{BF}(y) = 0$, this would require both child subtrees of y to increase in height, but the new key is added to only one of them.

[1] Some sources count edges instead of nodes, giving values smaller by 1.
[2] Some sources subtract in the opposite order, which flips the sign.

Case 3: $\mathrm{BF}(y) = 1$. Let z be the right child of y. A counterclockwise rotation at y followed by a clockwise rotation at x restores balance, as illustrated below.



In both Cases 1 and 3, the subtree originally rooted at x has the same height after rebalancing as before the insertion. Therefore, the heights and balance factors of nodes above this subtree are unchanged, and no further upward traversal is needed.

Each node stores its subtree height and balance factor, which we update during the upward traversal and rotations. Because the tree has height $O(\log n)$, the standard binary search tree insertion and the upward traversal each take $O(\log n)$ time. There are at most two rotations, each taking $O(1)$ time, so insertion takes $O(\log n)$ time overall.