CSCI 0500: Data Structures, Algorithms, and Intractability (Fall 2025)

Assignment 4

Due at 11:59pm ET, Monday, Nov 10

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

   For simplicity, you can assume all numbers are distinct. 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()`: returns 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.

   Each operation should run in worst-case $O(\log n)$ time, where n is the number of elements in the data structure.

   (b) Design a 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 run in expected $O(\log n)$ time, where n is the number of elements in the data structure.

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

   For simplicity, we consider only priorities and ignore the associated data.

   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 node 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 after each `extract_min`, so that there is at most one tree of each degree. The `extract_min` operation has three steps:

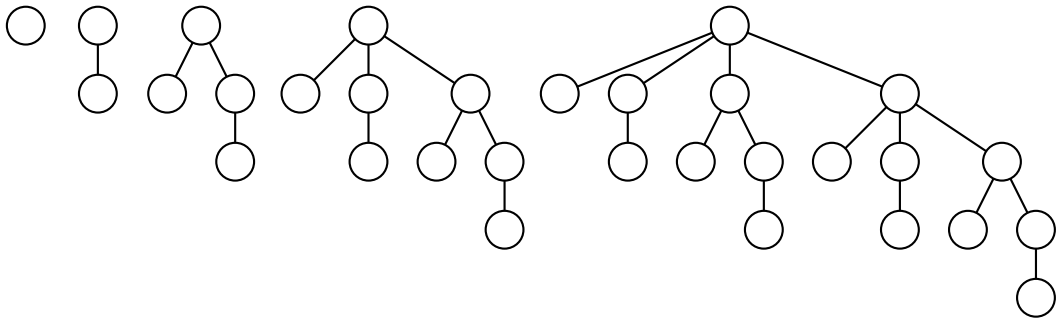
   1) Remove the minimum node and add all of its children as new trees.

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. Repeat until no two trees have the same degree.
3) Update the minimum node pointer.

Claim 1. *All trees in a Fibonacci heap are binomial trees.*

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 1 holds because insertion creates single-node trees, which are binomial trees. During `extract_min`, merging two binomial trees of degree $k-1$ gives a binomial tree of degree k.

(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` runs in $O(a + \log n)$ worst-case time, where a is the number of insertions since the previous `extract_min`, and n is the heap size before the operation.
(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 $O(\log n + a)$ 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 2 (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 3 (Balance Factor)**.** For a node v in a binary tree, the balance factor of v is defined as: $\mathrm{BF}(v) =$ (height of v's right subtree) $-$ (height of v's left subtree). [2]

[1] Some sources have slightly different definitions of depth and height, which could be off by 1.
[2] Some sources swap left and right in the balance factor definition, which is equivalent up to a sign change.

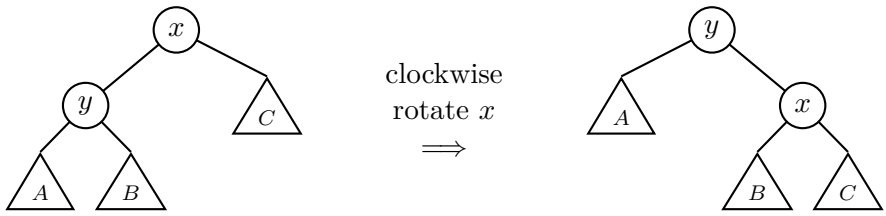
An AVL tree is a binary search tree where $\mathrm{BF}(v) \in \{-1, 0, 1\}$ for all nodes v. That is, the heights of the two child subtrees of any node differ by at most 1.

(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 a height-h AVL tree. Derive a recurrence for f_h in terms of f_{h-1} and f_{h-2}, and show that f_h grows exponentially in h.)

For simplicity, we focus on how an AVL tree maintains balance during insertion. Inserting a new key is done in two steps: First, insert the new key as in a standard binary search tree. Then, traverse upward from the inserted node to the root until a node with balance factor -2 or 2 is reached (if any), and then rebalance that node using rotations.

Let x be the leafmost node whose balance factor temporarily becomes -2 or 2. By symmetry, we can assume $\mathrm{BF}(x) = -2$, i.e., the left subtree of x is taller. Let y be the left child of x. Because x is the leafmost unbalanced node, we have $\mathrm{BF}(y) \in \{-1, 0, 1\}$.

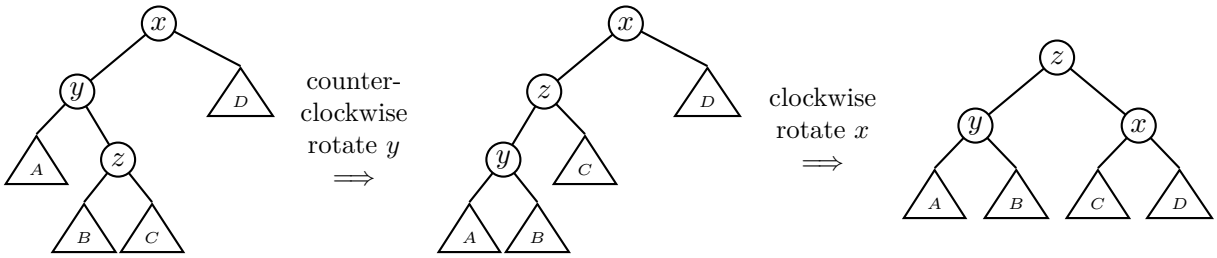
Case 1: $\mathrm{BF}(y) = -1$. A single clockwise rotation at x suffices.



(b) Prove that after a clockwise rotation at x, the balance factors of x and y are in $\{-1, 0, 1\}$.
(Hint: Let $h(V)$ be the height of the 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' be the height after the rotation. What are the values of $h'(A)$, $h'(B)$, $h'(C)$, and $h'(x)$?)

Case 2: $\mathrm{BF}(y) = 0$. This cannot happen because it would require the heights of both subtrees of y to increase during the initial insertion.

Case 3: $\mathrm{BF}(y) = 1$. This case requires two rotations, as illustrated below.



After the rotation(s), the height of the entire subtree (originally rooted at x) is the same as before the insertion. Therefore, no further upward traversal is needed after rebalancing x.

Each node maintains its subtree height and balance factor, which are updated during the initial insertion and any subsequent rotations. Overall, insertion takes $O(\log n)$ time because the height of the tree is $O(\log n)$: the initial insertion takes $O(\log n)$ time, the upward traversal takes $O(\log n)$ time, and there are at most two rotations, each of which takes $O(1)$ time.