CS 161 (Stanford, Fall 2025) Section 4

Binary Search and Red-Black Trees

Definitions

A Binary Search Tree (BST) is a binary tree where each node has a key where:

For any node n, n.key is larger than all of the keys in the subtree under n's left child, and smaller than all of the keys in the subtree under n's right child.

More informally:

- Every LEFT descendant of a node has key less than that node.
- Every RIGHT descendant of a node has key larger than that node

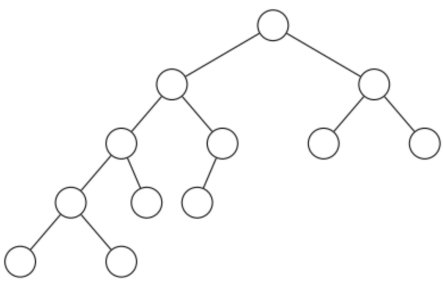
A Red-Black Tree is a type of self-balancing Binary Search Tree (BST) that maintains balance by ensuring that black nodes are evenly distributed and that there aren't too many consecutive red nodes. It obeys the following rules:

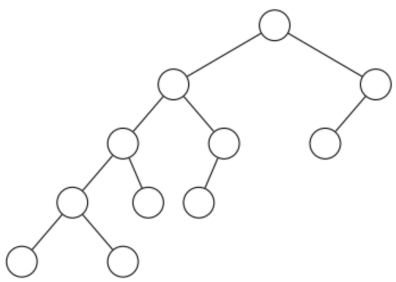
- Every node is colored red or black.
- The root node is a black node.
- NIL children count as black nodes.
- Children of a red node are black nodes.
- For all nodes x: all paths from x to NIL's have the same number of black nodes.

1 Red-Black Trees

1.1

For each of the following examples, if the nodes can be colored red or black to make a legitimate red-black tree, then give such a coloring. If not, then say that they cannot.





2 Binary Search Trees

2.1 Randomly Built BSTs

In this problem, we prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\log n)$. A *randomly built binary search tree* with n nodes is one that arises from inserting the n keys in random order into an initially empty tree, where each of the $n!$ permutations of the input keys is equally likely. Let $d(x, T)$ be the depth of node x in a binary tree T (The depth of the root is 0). Then, the average depth of a node in a binary tree T with n nodes is

$$\frac{1}{n}\sum_{x \in T} d(x, T).$$

1. Let the *total path length* $P(T)$ of a binary tree T be defined as the sum of the depths of all nodes in T, so the average depth of a node in T with n nodes is equal to $\frac{1}{n}P(T)$. Show that $P(T) = P(T_L) + P(T_R) + n - 1$, where T_L and T_R are the left and right subtrees of T, respectively.

2. Let $P(n)$ be the expected total path length of a randomly built binary search tree with n nodes. Show that $P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n - i - 1) + n - 1)$.

3. Show that $P(n) = O(n \log n)$. You may cite a result previously proven in the context of other topics covered in class.

4. Design a sorting algorithm based on randomly building a binary search tree. Show that its (expected) running time is $O(n \log n)$. Assume that a random permutation of n keys can be generated in time $O(n)$.

3 Hashing

3.1 Pattern matching with a rolling hash

In the Pattern Matching problem, the input is a *text* string T of length n and a *pattern* string P of length $m < n$. Our goal is to determine if the text has a (consecutive) substring[1] that is exactly equal to the pattern (i.e. $T[i \ldots i + m - 1] = P$ for some i).

1. Design a simple $O(mn)$-time algorithm for this problem.

2. Can we find a more efficient algorithm using hash functions? One naive way to do this is to hash P and every length-m substring of T. What is the running time of this solution?

3. Suppose that we had a universal hash family H_m for length-m strings, where each

[1] In general, *subsequences* are not assumed to be consecutive, but a *substring* is defined as a consecutive subsequence.

$h_m \in H_m$ the sum of hashes of characters in the string:

$$h_m(s) = h(S[0]) + \cdots + h(S[m-1]). \tag{1}$$

Explain how you would use this hash family to solve the pattern matching problem in $O(n)$ time.

(Hint: the idea is to improve over your naive algorithm by **reusing your work**.)

4. Unfortunately, a family of "additive" functions like the one in the previous item cannot be universal. Prove it.

5. The trick is to have a hash function that looks almost like (1): the hash function treats each character of the string is a little differently to circumvent the issue you discovered in the previous part, but they're still related enough that we can use our work. Specifically, we will consider hash functions parameterized by a fixed large prime p, and a random number x from $1, \ldots, p-1$:

$$h_x(S) = \sum_{i=0}^{m-1} S[i] \cdot x^i \pmod{p}.$$

For fixed pair of strings $S \neq S'$, the probability over random choice of x that the hashes are equal is at most m/p, i.e.

$$\Pr[h_x(S) = h_x(S')] \leq m/p.$$

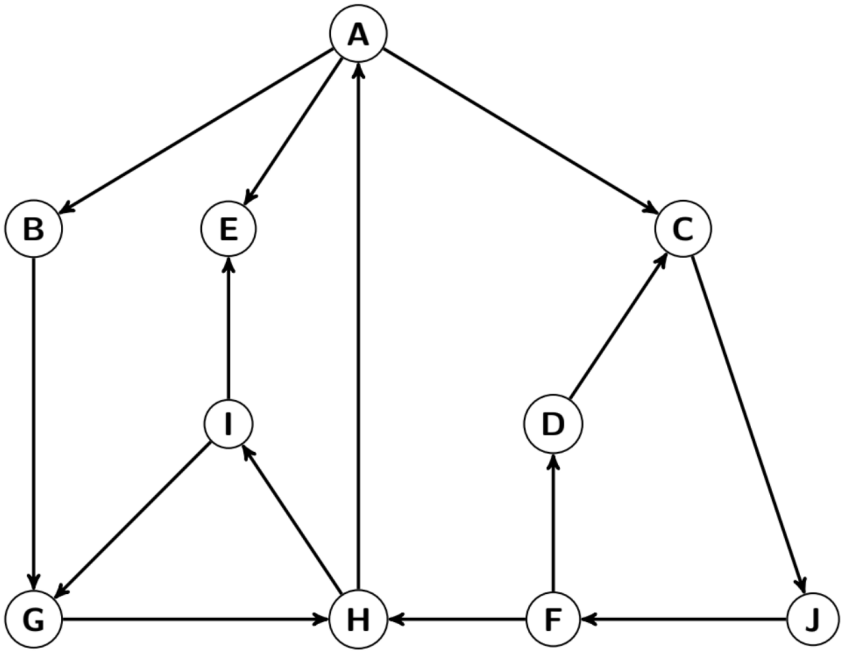
(This follows from the fact that a polynomial of degree $(m-1)$ can have at most m zeros. Do you see why?)

Design a randomized algorithm for solving the pattern matching problem. The algorithm should have worst-case run-time $O(n)$, but may return the wrong answer with small probability (e.g. $< 1/n$). (Assume that addition, subtraction, multiplication, and division modulo p can be done in $O(1)$ time.)

6. How would you change your algorithm so that it runs in *expected* time $O(n)$, but always return the correct answer?

4 Graphs, DFS, BFS

4.1 Graph Traversal



1. Perform DFS on the graph above starting from vertex A. Use lexicographical ordering to break vertex ties. As you go, label each node with the start time and the finish time. Highlight the edges in the tree generated from the search.

2. Perform BFS on the graph above starting from vertex A. Use lexicographical ordering to break vertex ties. As you go, label each node with the discovery order. Highlight the edges in the tree generated from the search.

3. Perform BFS on the graph above starting from vertex A. Use lexicographical ordering to break vertex ties. As you go, label each node with the discovery order. Highlight the edges in the tree generated from the search.