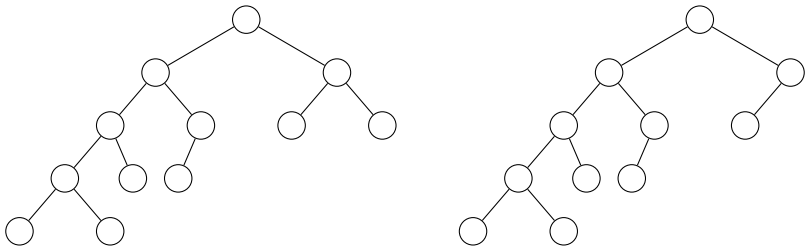
Red-Black Trees

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.



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 $O(n)$.

Pattern Matching with 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. $T[i \dots i+m-1] = P$ 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 $h_m \in H_m$ is the sum of hashes of characters (computed using a hash function h) in the string:

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

   Explain how you would hash all m-length substrings in T using this hash family 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 $1, \dots, p-1$:

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

   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?

7. Suppose that we had one fixed text T and many patterns $P_1, \dots P_k$ that we want to search in T. How would you extend your algorithm to this setting?

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

Hash Tables Gone Crazy

In this problem, we will explore *linear probing*. Suppose we have a hash table H with n buckets, universe $U = \{1, 2, \ldots, n\}$, and a *uniformly random* hash functions $h : U \to \{1, 2, \ldots, n\}$.

When an element u arrives, we first try to insert into bucket $h(u)$. If this bucket is occupied, we try to insert into $h(u) + 1$, then $h(u) + 2$, and so on (wrapping around after n). If all buckets are occupied, output **Fail** and don't add u anywhere. If we ever find u while doing linear probing, do nothing.

Throughout, suppose that there are $m \leq n$ distinct elements from U being inserted into H. Furthermore, assume that h is chosen *after* all m elements are chosen (that is, an adversary cannot use h to construct their sequence of inserts).

1. (Warmup) Can we ever output **Fail** while inserting these m elements?

2. Above, we gave an informal algorithm for inserting an element u. Your next task is to give algorithms for searching and deleting an element u from the table.

   Hint: Be careful that the search and delete algorithm work together!

3. In this part, we will analyze the runtime of linear probing, assuming no deletions occur.

   (a) Give an upper bound on the probability that $h(u) = h(v)$ for some u, v that are a part of these first m elements, assuming that $m = n^{1/3}$.

   Hint: You may need that for any $x > 0$, $(1 - x)^n \geq 1 - nx$.

   (b) When inserting an element, define the number of *probes* it does as the number of buckets it has to check, including the first empty bucket it looks at. For example, if $h(u), \ldots, h(u) + 10$ were occupied but $h(u) + 11$ was not then we would have to check 12 buckets.

   Prove that the expected number of total probes done when inserting $m = n^{1/3}$ elements is $O(m)$.

   (c) Suppose there are currently $m = \frac{n}{2}$ elements in our hash table, and we search for an existing element u (which is *not* adversarily chosen with respect to h). Prove that the expected number of probes done when searching for this element is $O(1)$.

   Hint: Recall that if $0 < \delta \leq 1$ and $Z \sim Binom(n, p)$ is a Binomial random variable with expectation μ, then $\mathbf{P}[Z \geq (1 + \delta)\mu] \leq \exp\left(-\frac{\delta^2 \mu}{3}\right)$.