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

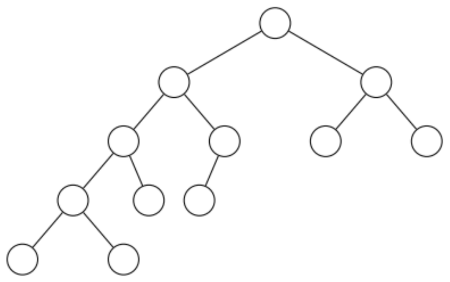
A Red-Black Tree is a type of self-balancing Binary Search Tree (BST) that maintains balance by ensuring that black nodes are evenly distributed and that there aren't too many consecutive red nodes. It obeys the following rules:

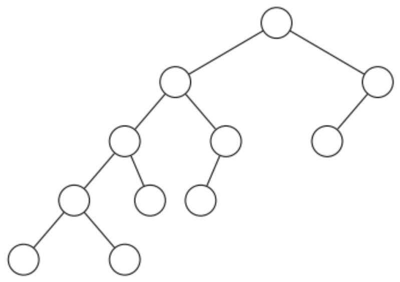
- Every node is colored red or black.
- The root node is a black node.
- NIL children count as black nodes.
- Children of a red node are black nodes.
- For all nodes x: all paths from x to NIL's have the same number of black nodes.

1 Red-Black Trees

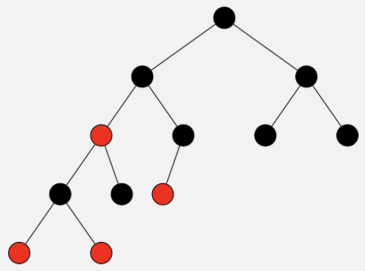
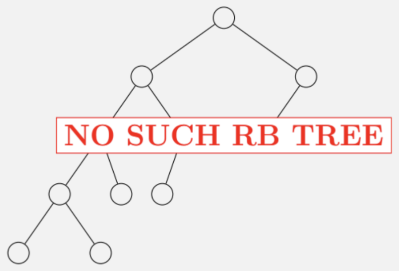
1.1

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.





Solution

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.

Solution

Let $r(T)$ denote the root of tree T. Note the depth of node x in T is equal to the length of the path from $r(T)$ to x. Hence, $P(T) = \sum_{x\in T} d(x, T)$. For each node x in T_L, the path from $r(T)$ to x consists of the edge $(r(T), r(T_L))$ and the path from $r(T_L)$ to x. The same reasoning applies for nodes x in T_R. Equivalently, we have

$$d(x, T) = \begin{cases} 0, & \text{if } x = r(T) \\ 1 + d(x, T_L), & \text{if } x \in T_L \\ 1 + d(x, T_R), & \text{if } x \in T_R \end{cases}$$

Then,

$$\begin{aligned}\sum_{x\in T} d(x,T) &= d(r(T),T) + \sum_{x\in T_L} d(x,T) + \sum_{x\in T_R} d(x,T)\\ &= 0 + \sum_{x\in T_L}[1+d(x,T_L)] + \sum_{x\in T_R}[1+d(x,T_R)]\\ &= |T_L| + |T_R| + \sum_{x\in T_L} d(x,T_L) + \sum_{x\in T_R} d(x,T_R)\\ &= n-1+P(T_L)+P(T_R).\end{aligned}$$

It follows that $P(T) = P(T_L) + P(T_R) + n - 1$.

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

Solution

Let T be a randomly built binary search tree with n nodes. Without loss of generality, we assume the n keys are $\{1, \ldots, \text{n}\}$. By definition, $P(n) = \mathbb{E}_{\mathbb{T}}[P(T)]$. Then, $P(n) = \mathbb{E}_{\mathbb{T}}[P(T_L) + P(T_R) + n - 1] = n - 1 + \mathbb{E}_{\mathbb{T}}[P(T_L)] + \mathbb{E}_{\mathbb{T}}[P(T_R)]$, where T_L and T_R are the left and right subtrees of T, respectively. Note

$$\mathbb{E}_{\mathbb{T}}[P(T_L)] = \sum_{i=1}^{n} \mathbb{E}_{\mathbb{T}}[P(T_L)|r(T)=i]\cdot Pr(r(T)=i).$$

Since each element is equally likely to be the root of T, $\Pr(r(T) = i) = \frac{1}{n}$ for all i. Conditioned on the event that element i is the root, T_L is a randomly built binary search tree on the first $i-1$ elements. To see this, assume we picked element i to be the root. From the point of view of the left subtree, elements $1, \ldots, i-1$ are inserted into the subtree in a random order, since these elements are inserted into T in a random order and subsequently go into T_L in the same relative order. Hence, $\mathbb{E}_{\mathbb{T}}[P(T_L)|r(T) = i] = P(i-1)$. Putting these together, we get

$$\mathbb{E}_{\mathbb{T}}[P(T_L)] = \sum_{i=1}^{n}\frac{1}{n}P(i-1).$$

Similarly, we get $\mathbb{E}_{\mathbb{T}}[P(T_R)] = \sum_{i=1}^{n}\frac{1}{n}P(n-i)$. Then,

$$\begin{aligned}P(n) &= n-1+\mathbb{E}_{\mathbb{T}}[P(T_L)]+\mathbb{E}_{\mathbb{T}}[P(T_R)]\\ &= n-1+\frac{1}{n}\sum_{i=1}^{n}[P(i-1)+P(n-i)]\\ &= n-1+\frac{1}{n}\sum_{i=0}^{n-1}[P(i)+P(n-i-1)],\end{aligned}$$

where we changed the indexing of the sumamtion in the last equality.

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.

Solution

This is the same recurrence that appears in the analysis of Quicksort.

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 $O(n)$.

Solution

The algorithm is 1) construct a randomly built binary search tree T by inserting given elements in a random order; and 2) do the inorder traversal on T to get a sorted list. Note step 2 can be done in $O(n)$ time. We argue that step 1 takes $O(n \log n)$ time in expectation. We observe that given the final state of tree T, we can compute the amount of work spent to construct T. To insert a node x at depth d, we traversed exactly the path from the root to the parent of x, at depth $d - 1$, to insert it. Hence, we can upper bound the total work done to construct T by $O(P(T))$. From part (c), we know that $P(T) = O(n \log n)$ in expectation. It follows that Step 1 takes $O(n \log n)$ time in expectation. Overall, the algorithm runs in $O(n \log n)$ in expectation.

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. $T[i \dots i + m - 1] = P$ for some i).

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

Solution

Compare P to each length m substring of T starting from index 0 to $n - m$. Return true if any substring matches exactly, and false otherwise.
This algorithm iterates through $O(n)$ substrings of T, and each check against P takes $O(m)$, making the algorithm $O(mn)$.

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

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?

   Solution

   Hashing every length-m substring of T takes $O(m)$ for each substring, with a total of $O(n)$ substrings. This overall is still $O(mn)$.

3. Suppose that we had a universal hash family H_m for length-m strings, where each $h_m \in H_m$ 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**.)

   Solution

   Each time we hash the next substring, subtract the hash of the character that was removed and add the hash of the character that was added. This takes $O(1)$ for each substring, so the overall runtime becomes $O(n)$.

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

   Solution

   Consider a 2 character string S, and another 2 character string S' with the characters of S in reverse order. For any $h_m \in H_m$ we have $P(h_m(S) = h_m(S')) = 1$ and $S' \neq S$.

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.)

Solution

Our algorithm uses the same idea from part 3, but applies this polynomial rolling hash function instead. The key insight is that if we have $h_x(T[k \dots k+m-1])$ then we have

$$h_x(T[k+1 \dots k+m]) = (h_x(T[k \dots k+m-1]) - T[k])/x + T[k+m] \cdot x^{m-1} \pmod p$$

Algorithm 1: PatternMatch(T, P)

```
$p_h \leftarrow h_x(p)$
**for** *all substrings* $s_k \in T$ **do**
    **if** $k = 0$ **then**
        $hash \leftarrow h_x(s_k)$
    **else**
        $hash \leftarrow (hash - s_{k-1}[0])/x + s_k[m] \cdot x^{m-1}$
    **if** $hash = p_h$ **then**
        return True
return False
```

Runtime: Computing the hash for a substring from scratch takes $O(m)$ time. However, we compute the entire hash only for P and the first substring of T. Remaining hashes requires computing x^m, but we can precompute and store this value. This makes computing hash for each subsequent substring $O(1)$, making the algorithm $O(n)$.

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

Solution

Modify the algorithm so that whenever the hashes match, before returning "True" it also checks that the pattern P actually matches to the substring (and if not continue the loop).
Checking takes $O(m)$ time, and in expectation we would only have to check $O(n \cdot m/p)$ times. (That's $O(n)$ hash comparisons $\times$ probability m/p of false positive each hash comparison). When $p = \Omega(n \cdot m)$, that's $O(1)$ checks.

7. Suppose that we had one fixed text T and many patterns $P_1, \dots P_k$ that we want to

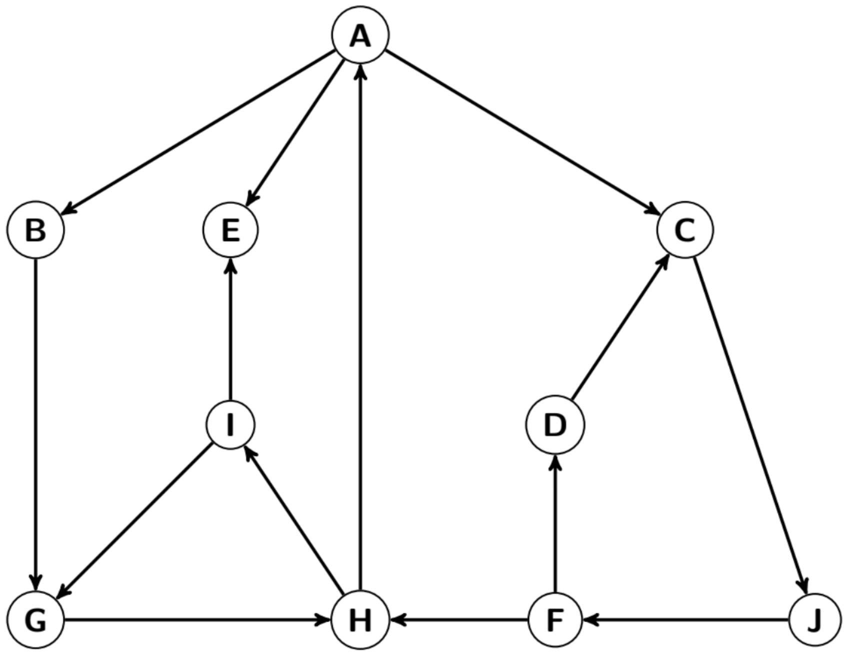
search in T. How would you extend your algorithm to this setting?

Solution

We can extend our algorithm simply by hashing each of $P_1, \ldots, P_k$ and checking the hash of each substring against this set of hashes.

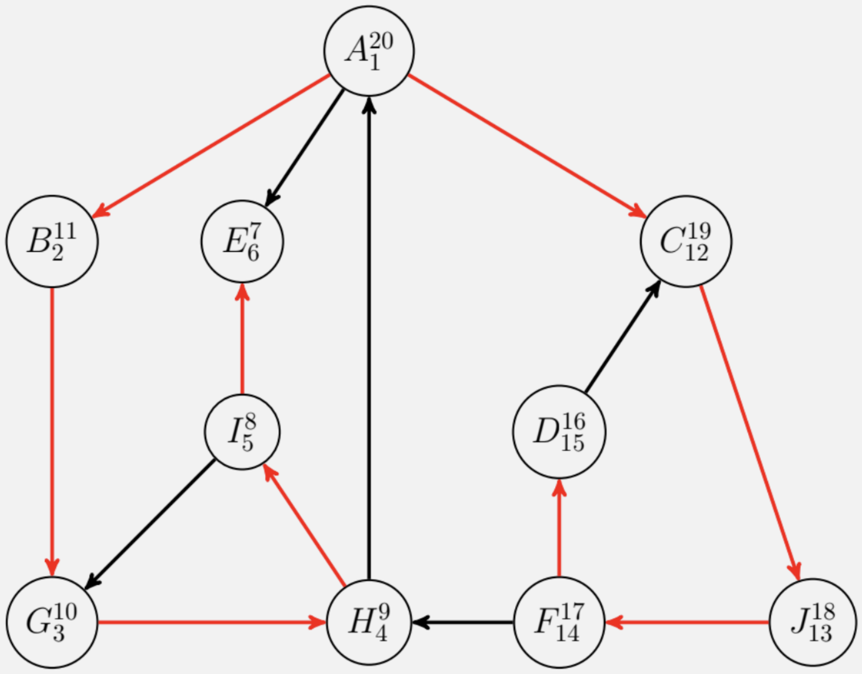
4 Graphs, DFS, BFS

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.

Solution



Problem Solving Notes:

(a) *Read and Interpret:* We are asked to perform DFS with an explicit condition for breaking ties. Following lecture examples on this is a good place to start.

(b) *Information Needed:* When should we update our counter to track start and end times? What conditions must be met before marking a node as completed?

(c) *Solution Plan:* Beginning at node A, proceed to the children of A in lexigraphical order, recursively calling DFS on the children nodes. We can mark a node as finished once each of its children have at least been marked as started (in most cases, they will all be marked as finished; see exception in True or False q.2). Our counter is updated any time we mark the start or end of a particular node in the graph.

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.

Solution

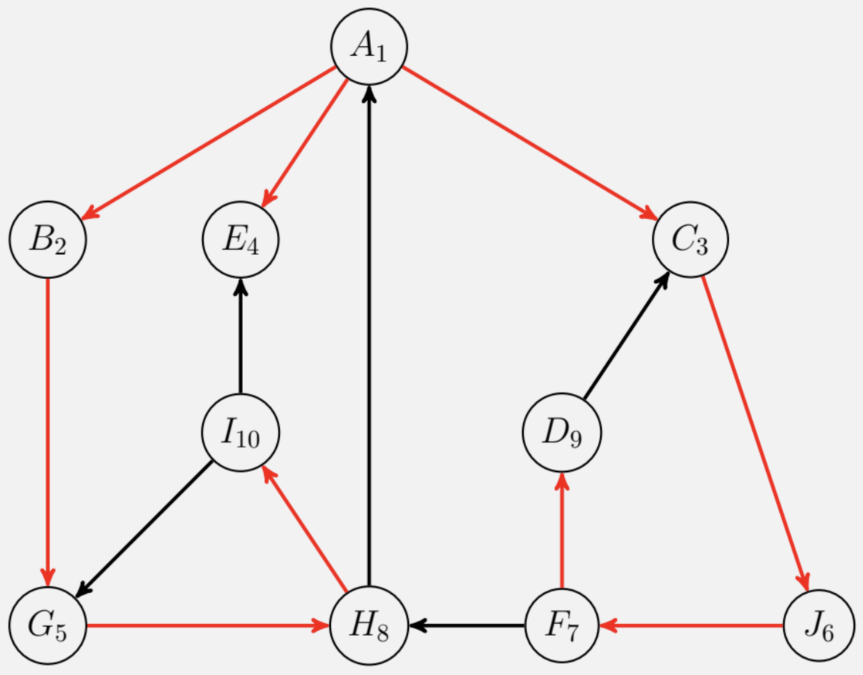


Figure 1: BFS traversal from *A* with discovery order (red edges = BFS tree).

Problem Solving Notes:

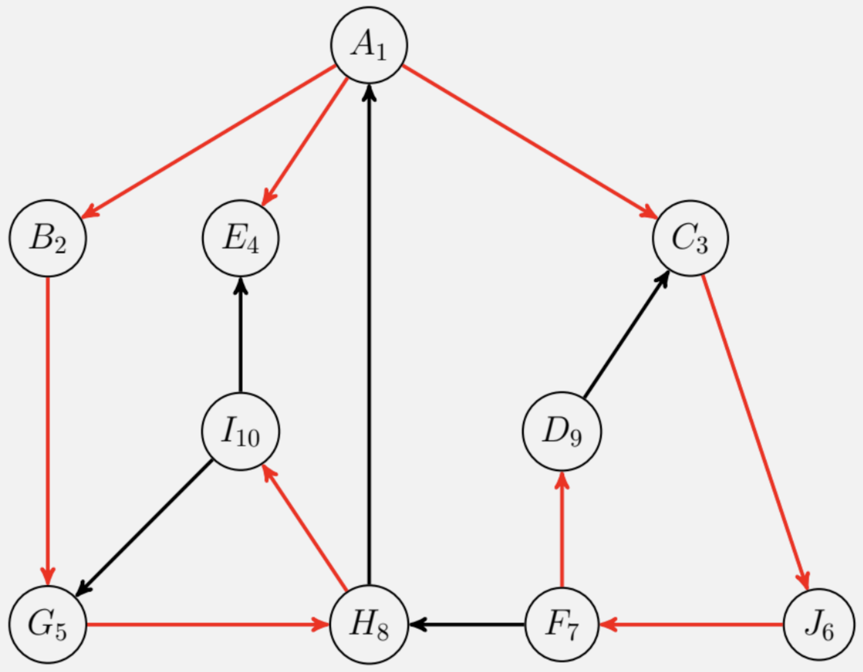
(a) *Read and Interpret:* We are asked to perform BFS with a stated tie-breaker (lexicographic).

(b) *Information Needed:* When do we increment the discovery counter? How does queue order (FIFO) change exploration compared with DFS?

(c) *Solution Plan:* Begin at *A*, enqueue its neighbors in lexicographical order and mark each as discovered upon enqueue. Repeatedly dequeue the front vertex, explore its undiscovered neighbors (again in lexicographical order), and add tree edges to the BFS tree until the queue is empty.

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.

Solution



Problem Solving Notes:

(a) *Read and Interpret:* We are asked to perform BFS with an explicit condition for breaking ties. Following the lecture examples on this is a great place to start!

(b) *Information Needed:* When should we be updating our discovery order counter? How does this traversal differ from DFS?

(c) *Solution Plan:* Beginning at node A, proceed to the children of A in lexicographical order, immediately marking each node as it's discovered. Repeat this procedure for each of the child nodes, continuing until all nodes are explored.