CS 161 (Stanford, Fall 2025)
Optional Extra Section 4 Problems

1 Red-Black Trees

1.1

If the length of a path from the root of a red-black tree to one of the leaf NIL nodes is 50, what could be the length of another path from the root to some other NIL node?

Solution

Any value between ny value between 25 and 100 (inclusive).
In a red-black tree, all paths from the root to any NIL node must contain the same number of black nodes.
The shortest possible path has only black nodes, while the longest possible path alternates between red and black nodes, meaning its length is at most twice the black-height.
If one path has length 50, then its black-height is between 25 and 50. Hence, the length of another path from the root to a different NIL node could be any value between 25 and 100 (inclusive).

2 Binary Search Trees

2.1 True or False

Which of the following statements are true?

(a) The height of a binary search tree with n nodes cannot be smaller than $\Theta(\log n)$.

(b) All the operations supported by a binary search tree (except OutputSorted) run in $O(\log n)$ time.

Solution

(a) True. Statement (a) holds because there are at most 2^i nodes in the ith level of a binary tree, and hence at most $1+2+4+\cdots+2^i \leq 2^{i+1}$ nodes in levels 0 through i combined. Accommodating n nodes requires $2^{h+1} \geq n$, where h is the tree height, so $h = \Omega(\log n)$.

(b) False. Statement (b) holds for *balanced* binary search trees but is generally false for unbalanced ones.

3 Hashing

3.1 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?

   Solution

   No, as there are n spots in the table.

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!

   Solution

   When deleting, we should take special care that a previously occupied bucket is still marked as "previously occupied". Here's why: suppose $n = 3$ and $h(0) = 0, h(1) = 1, h(2) = 0$. Suppose elements were inserted in the order $0, 1, 2$. Then, $H = [0, 1, 2]$. What happens if we delete 1 and then search for 2? Well, after deleting 1, $H = [0, \square, 2]$ and so naively searching for 2 would return false, as the spot after 0 is empty.
   To get around this, we mark such deletions with a "tombstone" value so that search treats those as elements.

```
def Search(H, u):
    start = h(u)
    if H[start] == u:
        return True
    current = start + 1
    while current != start:
        if H[current] == u:
            return True
        elif H[current] == empty: # X is not empty!
            return False
```

```
        current = current % n + 1 #adds 1 mod n

def Delete(H, u):
    start = h(u)
    if H[start] == u:
        H[start] = X (tombstone)
        return
    current = start + 1
    while current != start:
        if H[current] == u:
            H[current] = X
            return
        elif H[current] == empty:
            return # u is not in H
        current = current % n + 1 #adds 1 mod n
```

3. In this part, we will analyze the expected runtime of linear probing assuming that $m = n^{1/3}$ and that 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 $\mathbf{P}[\textit{at least one of } E_1, \ldots, E_k \textit{ happens}] \leq \sum_{i \in [k]} \mathbf{P}[E_i]$ *given any random events* $E_1, \ldots, E_k$.

   Solution

   Number the elements $u_1, u_2, \ldots, u_m$. The probability that any pair of elements collide with each other is $\frac{1}{n}$, and hence, by union bound over all possible pairs, an upper bound on the probability of any collision is $\frac{m^2}{n} = n^{-1/3}$.

   (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 $O(m)$.

   Solution

   Let X be the total number of probes. Clearly $X \leq m^2$ (each inserted element can only be compared against the other inserted elements' positions). Let E be the event that there is at least one collision.

Using the law of total expectation and simple bounds:

$$\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}[X \mid E]\Pr[E] + \mathbb{E}[X \mid \overline{E}]\Pr[\overline{E}] \\
&\leq \mathbb{E}[X \mid E]\Pr[E] + \mathbb{E}[X \mid \overline{E}] \\
&\leq m^2 \cdot \frac{m^2}{n} \;+\; m \\
&= \frac{m^4}{n} + m \\
&= 2m \qquad (\text{since } n = m^3) \\
&= O(m).
\end{aligned}$$

Here we used $\Pr[E] \leq \frac{m^2}{n}$ (union bound over the $\binom{m}{2} \leq m^2/2$ pairs) and $\mathbb{E}[X \mid E] \leq m^2$. If there are no collisions ($\overline{E}$), each insertion probes exactly one bucket, so $\mathbb{E}[X \mid \overline{E}] = m$.

4 Graphs, DFS, BFS

4.1 True or False

1. If (u, v) is an edge in an undirected graph and during DFS, vertex v is completely explored before vertex u, then u is an ancestor of v in the DFS tree.

Solution

True. In DFS, a vertex is marked as "finished" only after all of its neighbors have been explored. If v is explored and finished before u, and there is an edge between them, that means DFS reached v while exploring from u and returned to u only after finishing v. Therefore, u must have discovered v, making u an ancestor of v in the DFS tree. The only other possibility is that v was explored and finished before u was ever visited—but since there is an edge between u and v, DFS would have explored v from u before finishing u, so that cannot happen.

2. In a directed graph, if there is a path from u to v and DFS visits u before visiting v, then u is an ancestor of v in the DFS tree.

Solution

False. Just because there is a path from u to v and DFS happens to visit u first does not mean that u will discover v directly. For example, if both u and v are reached from another vertex w, DFS might explore all of u's descendants before ever following the path that leads from u to v. In that case, u and v would be siblings (both children of w), not ancestor and descendant. The key difference is that this statement only

requires a path from u to v, not a direct edge—so v might be discovered independently in the DFS, not through u.

4.2 Bipartite Graphs

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) connects a vertex from U to V or a vertex from V to U. A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. In lecture, we saw an algorithm using BFS to determine where a graph is bipartite. Design an algorithm using DFS to determine whether or not an undirected graph is bipartite.

Solution

The algorithm is essentially the same as that of DFS, except at every node we visit, we either color it if it hasn't been visited before, or check its color if it has been visited before. The rough algorithm is as follows:

1. Start DFS from any node and color it RED.
2. Color the next node BLUE.
3. Continue coloring each successive node the opposite color until the end of the tree is reached.
4. If at any point a current node is the same color as one of its neighbors, then return false.
5. Once every node has been visited, if we haven't returned false, then the graph is bipartite.

Problem Solving Notes:

1. *Read and Interpret:* We are asked solve the bipartite graph problem using DFS instead of BFS. Since we aren't asked for a new algorithm, we simply need to figure out what additional bookkeeping we should do to DFS to solve this different problem.
2. *Information Needed:* Are the start/end times sufficient for determining whether the graph is bipartite? Is there a way to rule out with certainty a graph's bipartiteness given the procedure of our DFS algorithm?
3. *Solution Plan:* We follow the same coloring scheme procedure that BFS does by enforcing that no edge connects two vertices of the same color. Starting with an arbitrary color, each time we encounter a new node, we are forced to color that node a specific color to ensure bipartiteness. If we reach an impossible situation (i.e. where either coloring of a particular vertex will break the graph's bipartiteness), we can safely say that no such coloring exists.