CS 61B Spring 2022

Balanced Search and Graphs

Exam Prep Discussion 13: April 18, 2022

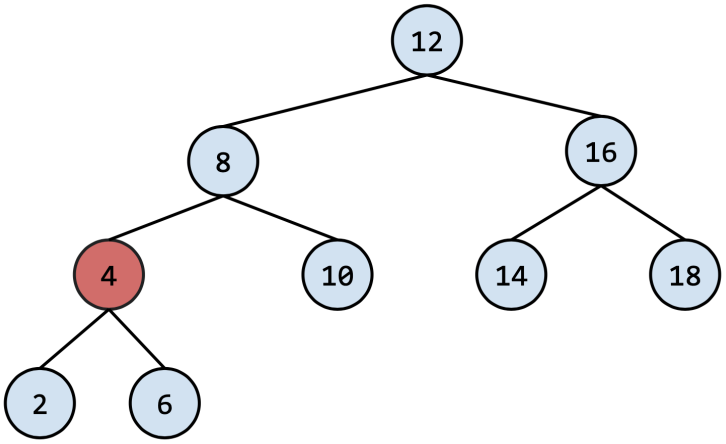
1 LLRBs

a) (2 Points). Perform the following insertions on the Left Leaning Red Black Tree (LLRB) given below. For each insertion, give the fix up operations needed. Recall a fix up operation is one of the following:

- `rotateLeft`
- `rotateRight`
- `colorFlip`
- change the root node to black.

Note that insertions are **dependent**. If only two operations are necessary, pick "None" for the third operation. If only one operation is necessary, pick "None" for the second and third operation. If no operations are necessary, pick "None" for all three operations.

If you put "None" for the "Operation applied", **leave the "Node to apply on" blank.** (Summer 2021 MT2)



i) (0.5 Points). Insert 17

	Operation applied	Node to apply on
1st operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	
2nd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	
3rd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	

Solution:

	Operation applied	Node to apply on
1st operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black √ **None**	
2nd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black √ **None**	
3rd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black √ **None**	

Explanation: 17 is inserted as the left child of 18. No fixes are required at this point.

ii) (0.5 Points). Insert 15. Note that insertions are dependent, so insert 15 into the state of the LLRB after the insertion of 17.

	Operation applied	Node to apply on
1st operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	
2nd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	
3rd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	

Solution:

	Operation applied	Node to apply on
1st operation	√ **`rotateLeft()`** ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	14
2nd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black √ **None**	
3rd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black √ **None**	

Explanation: 15 is inserted as the right child of 14. This requires a left rotation of 14 to maintain the left-leaning invariant.

iii) (0.75 Points). Insert 13. Note that insertions are dependent, so insert 13 into the state of the LLRB after the insertion of 15.

	Operation applied	Node to apply on
1st operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	
2nd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	
3rd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	

Solution:

	Operation applied	Node to apply on
1st operation	◯ `rotateLeft()` √ **`rotateRight()`** ◯ `colorFlip()` ◯ change root to black ◯ `None`	15
2nd operation	◯ `rotateLeft()` ◯ `rotateRight()` √ **`colorFlip()`** ◯ change root to black ◯ `None`	14
3rd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black √ **`None`**	

Explanation: 13 is inserted as the left child of 14. This requires a right rotation on 15, since you cannot have 2 left red nodes in a row; then you must color flip 14 to break up the 4-node.

iv) (0.75 Points). Insert 19. Note that insertions are dependent, so insert 19 into the state of the LLRB after the insertion of 13.

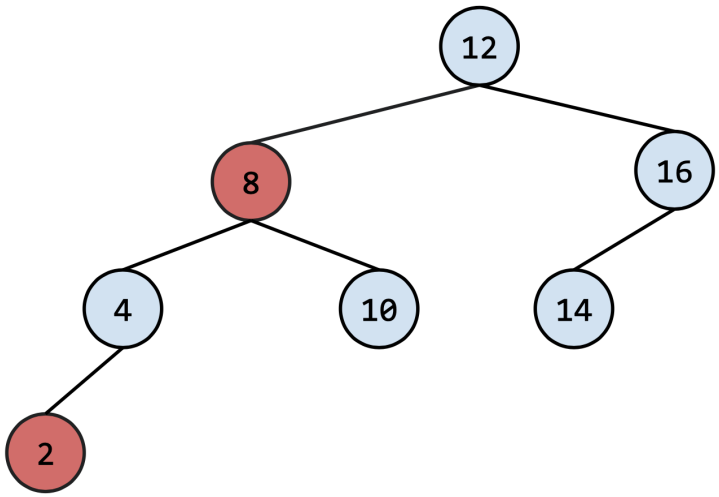
	Operation applied	Node to apply on
1st operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	
2nd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	
3rd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	

Solution:

	Operation applied	Node to apply on
1st operation	◯ `rotateLeft()` ◯ `rotateRight()` √ **`colorFlip()`** ◯ change root to black ◯ `None`	18
2nd operation	◯ `rotateLeft()` ◯ `rotateRight()` √ **`colorFlip()`** ◯ change root to black ◯ `None`	16
3rd operation	√ **`rotateLeft()`** ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	12

Explanation: 19 is inserted as the right child of 18. This requires a color flip on 18 to break up the 4-node, then a color flip on 16 which not has 2 red children. After this, a left rotation on 12 is required since it has a red right child.

b) (1.5 Points). The tree below is **not** a valid LLRB (hint: to see why this is the case, draw the corresponding 2-3 tree) but it's close! In this part, we will try to *transform* it into a valid LLRB in two different ways. Note that each way acts **independently** of the previous. If a way isn't possible, put `impossible`. Recall that LLRBs **cannot** have duplicates.



i) (0.75 Points). Way 1: Remove a **single leaf** node from the tree. Which leaf node?

◯ 2 ◯ 4 ◯ 8 ◯ 10 ◯ 12 ◯ 14 ◯ 16 ◯ impossible

Solution:

◯ 2 ◯ 4 ◯ 8 ◯ 10 ◯ 12 √ **14** ◯ 16 ◯ impossible

Explanation: A LLRB always has the same "black height" (number of black nodes from root to leaf). Note that the left child has a "black height" of 2 but the right has a black height of 3; thus deleting 14 makes this a valid LLRB.

ii) (0.75 Points). Way 2: Flip the color of a **single** node. Which node?

◯ 2 ◯ 4 ◯ 8 ◯ 10 ◯ 12 ◯ 14 ◯ 16 ◯ impossible

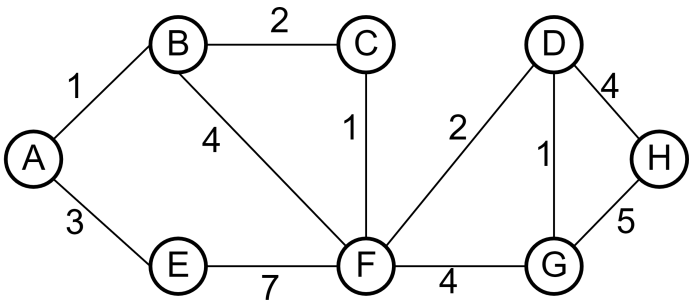
Solution:

◯ 2 ◯ 4 ◯ 8 ◯ 10 ◯ 12 √ **14** ◯ 16 ◯ impossible

Explanation: Like above, flipping 14 decreases the black height of the right child by 1, making it valid.

2 DFS, BFS, Dijkstra's, A*

For the following questions, use the graph below and assume that we break ties by visiting lexicographically earlier nodes first.



(a) Give the depth first search preorder traversal starting from vertex A.

A, B, C, F, D, G, H, E

Explanation: Preorder visits the current node, then recursively calls on each of its children. The chain of calls looks like this:

```
dfs(A)
    dfs(B)
        dfs(C)
            dfs(F)
                dfs(D)
                    dfs(G)
                        dfs(H)
                dfs(E)
```

(b) Give the depth first search postorder traversal starting from vertex A.

H, G, D, E, F, C, B, A

Explanation: Postorder recurses on all children then visits the current node. See above for the order of calls.

(c) Give the breadth first search traversal starting from vertex A.

A, B, E, C, F, D, G, H

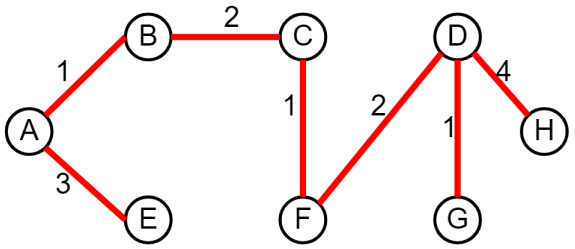
Explanation: BFS visits nodes in increasing distance (by number edges) from the source (ties broken alphabetically as given in the instructions). The groups in increasing distance from A are: 0: (A), 1: (B, E), 2: (C, F), 3: (D, G), 4: (H). Alternatively, draw out the queue to confirm this ordering for yourself.

(d) Give the order in which Dijkstra's Algorithm would visit each vertex, starting from vertex A. Sketch the resulting shortest paths tree.

A, B, C, E, F, D, G, H

Explanation: Dijkstra's visits nodes in increasing distance from the source (weighted). This order is: A: 0, B: 1, C: 3, E: 3, F: 4, D: 6, G: 7, H: 10. You

can confirm this ordering is the same as manually running Dijkstra's.



(e) Give the path A* search would return, starting from A and with G as a goal. Let $h(u, v)$ be the valued returned by the heuristic for nodes u and v.

u	v	$h(u, v)$
A	G	9
B	G	7
C	G	4
D	G	1
E	G	10
F	G	3
H	G	5

$A \to B, B \to C, C \to F, F \to D, D \to G$

Explanation: Note that this heuristic is not admissible so it may not return the shortest path ($h(u, v) > dist(A, G) = 7$). The A* execution trace is as follows:

(a) Pop off A.

```
pq = [B: (3, 8), E: (3, 13)] // (dist, dist + h)
prev = [B: A, E: A]
```

(b) Pop off B.

```
pq = [C: (3, 7), F: (5, 8), E: (3, 13)]
prev = [B: A, C: B, E: A, F: B]
```

(c) Pop off C.

```
pq = [F: (4, 7), E: (3, 13)]
prev = [B: A, C: B, E: A, F: C]
```

(d) Pop off F.

```
pq = [D: (6, 7), E: (3, 13)]
prev = [B: A, C: B, D: F, E: A, F: C]
```

(e) Pop off D.

```
pq = [G: (7, 7), H: (10, 15), E: (3, 13)]
prev = [B: A, C: B, D: F, E: A, F: C, G: D, H: D]
```

(f) Pop off G. Race the `prev` pointers to get the path.

3 Graph Conceptuals

Answer the following questions as either **True** or **False** and provide a brief explanation:

1. If a graph with n vertices has $n-1$ edges, it **must** be a tree.

   False. A tree **must** be connected.

2. The adjacency matrix representation is **typically** better than the adjacency list representation when the graph is very connected.

   True. The adjacency matrix representation is usually worse than the adjacency list representation with regards to space, scanning a vertex's neighbors, and full graph scans. However, when the graph is very connected, the adjacency matrix representation has roughly same asymptotic runtime in these operations, while "winning" in operations like hasEdge.

3. Every edge is looked at exactly twice in **every** iteration of DFS on a connected, undirected graph.

   True. The two vertices the edge is connecting will look at that edge when it's their turn.

4. In BFS, let $d(v)$ be the minimum number of edges between a vertex v and the start vertex. For any two vertices u, v in the fringe, $|d(u) - d(v)|$ is **always less than** 2.

   True. Suppose this was not the case. Then, we could have a vertex 2 edges away and a vertex 4 edges away in the fringe at the same time. But, the only way to have a vertex 4 edges away is if a vertex 3 edges away was removed from the fringe. We see this could never occur because the vertex 2 edges away would be removed before the vertex 3 edges away!

5. Given a fully connected, directed graph (a directed edge exists between every pair of vertices), a topological sort can never exist.

   False. Consider the graph constructed as follows: for all vertices i, j such that $i < j$, draw a directed edge from i to j. A valid topological ordering of this graph is simply enumerating the vertices: $1, 2, 3, \ldots . N$.

4 Cycle Detection

Given an undirected graph, provide an algorithm that returns true if a cycle exists in the graph, and false otherwise. Also, provide a Θ bound for the worst case runtime of your algorithm. You may use either an adjacency list or an adjacency matrix to represent your graph. (We are looking for an answer in plain English, not code).

We do a depth first search traversal through the graph. While we recurse, if we visit a node that we visited already, then we've found a cycle. Assuming integer labels, we can use something like a `visited` boolean array to keep track of the elements that we've seen, and while looking through a node's neighbors, if `visited` gives true, then that indicates a cycle.

However, since the graph is undirected, if an edge connects vertices `u` and `v`, then `u` is a neighbor of `v`, and v is a neighbor of `u`. As such, if we visit `v` after `u`, our algorithm will claim that there is a cycle since `u` is a visited neighbor of `v`. To address this case, when we visit the neighbors of `v`, we should ignore `u`. To implement this in code, one idea is using a `Map` to map each node to the node we took to get there, e.g. we would map `v` to `u` in the example described above.

In the worst case, we have to explore at most V edges before finding a cycle (number of edges doesn't matter). So, this runs in $\Theta(V)$.