CS 61B Spring 2022

Balanced Search and Graphs

Exam Prep Discussion 13: April 18, 2022

1 LLRBs

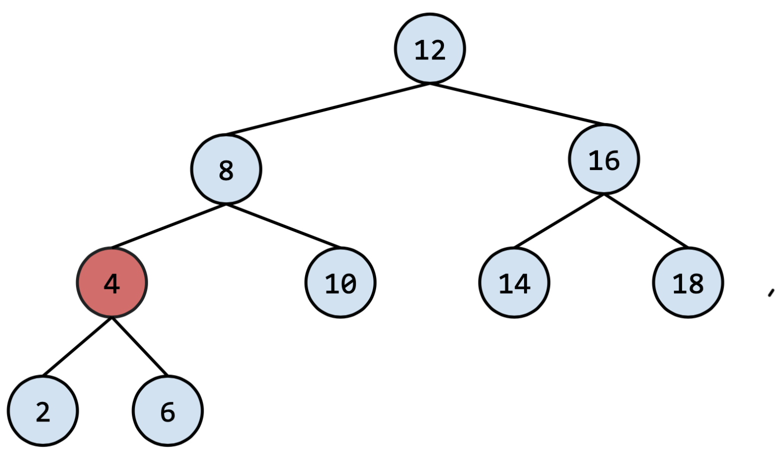
a) (2 Points). Perform the following insertions on the Left Leaning Red Black Tree (LLRB) given below. For each insertion, give the fix up operations needed. Recall a fix up operation is one of the following:

- `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.

For all insertions - insert a red node, treating the LLRB as a BST, then maintain LLRB properties.

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	

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.

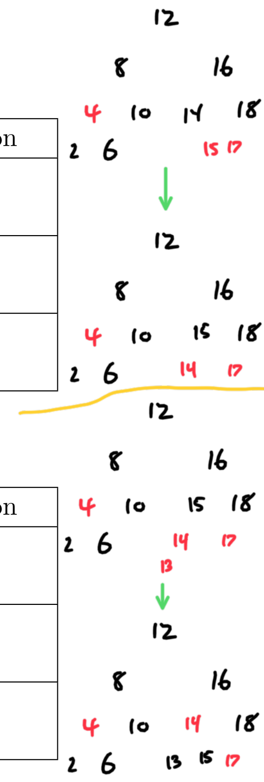
	Operation applied	Node to apply on
1st operation	● `rotateLeft()` ○ `rotateRight()` ○ `colorFlip()` ○ change root to black ○ `None`	14
2nd operation	○ `rotateLeft()` ○ `rotateRight()` ○ `colorFlip()` ○ change root to black ● `None`	
3rd operation	○ `rotateLeft()` ○ `rotateRight()` ○ `colorFlip()` ○ change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

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`	15
2nd operation	○ `rotateLeft()` ○ `rotateRight()` ● `colorFlip()` ○ change root to black ○ `None`	14
3rd operation	○ `rotateLeft()` ○ `rotateRight()` ○ `colorFlip()` ○ change root to black ● `None`	

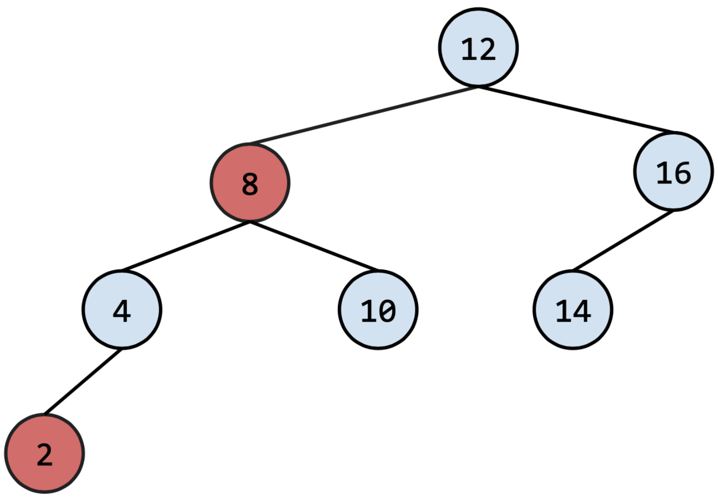
iv) (0.75 Points). Insert 19. Note that insertions are dependent, so insert 19 into the state of the LLRB after the insertion of 13.

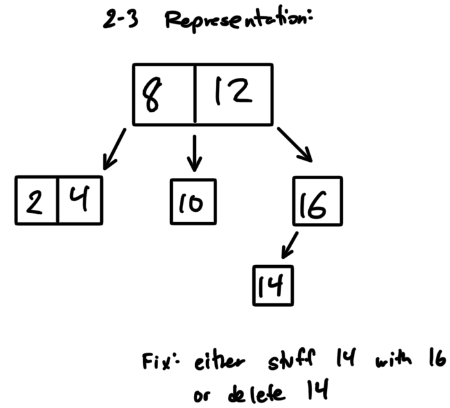
	Operation applied	Node to apply on
1st operation	○ `rotateLeft()` ○ `rotateRight()` ● `colorFlip()` ○ change root to black ○ `None`	18
2nd operation	○ `rotateLeft()` ○ `rotateRight()` ● `colorFlip()` ○ change root to black ○ `None`	16
3rd operation	● `rotateLeft()` ○ `rotateRight()` ○ `colorFlip()` ○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



Try drawing this one! A red 19 is added to the right of 18; do two color flips then satisfy LL property by rotating left.

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?

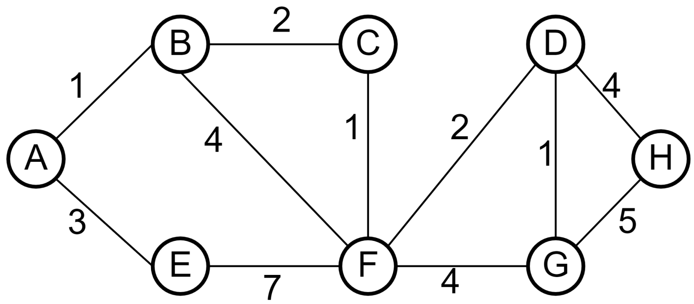
○ 2 ○ 4 ○ 8 ○ 10 ○ 12 ● 14 ○ 16 ○ impossible

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

○ 2 ○ 4 ○ 8 ○ 10 ○ 12 ● 14 ○ 16 ○ impossible

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.

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

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

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

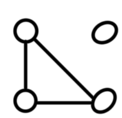
(e) Give the path A* search would return, starting from A and with G as a goal. Let $h(u, v)$ be the valued returned by the heuristic for nodes u and v.

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

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- may not be connected

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

True - matrices have constant access time, lists would require iteration

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

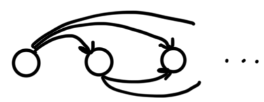
True - the 2 vertices of the edge will look at it

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 - BFS goes fully through layer i, adds all layer i+1 nodes then adds i+2 once all layer i nodes done.

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

False - complete directed graph



Each node is connected to all nodes after it

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).

DFS.

keep track of the node prior to visiting a given node (to exclude it in moving forward).

If a node is revisited, then there is a cycle.

Once all nodes are visited, algorithm concludes no cycle - which is worst case, and it runs in $\Theta(V)$ time.