Andrew Mackay

ID:3441988

Assignment 3

Question1 :

A red-black tree must satisfy the following properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (null node) is black.
4. If a node is red, then both its children must be black (no two red nodes can be adjacent).
5. Every simple path from a node to a descendant leaf must have the same number of black nodes.

Any AVL tree can be transformed into a red-black tree by following the next steps:

1. We start by coloring all nodes in the AVL tree as black. This ensures that properties 2, 3, and 4 of a red-black tree are satisfied.
2. Since red-black trees cannot have two consecutive red nodes, we need to ensure that this property is maintained. Therefore, we traverse the AVL tree, and for each node that has a red child, we apply the following steps:
   1. If the parent node and its red child are both left children or both right children, we perform a rotation (either a single or double rotation) to make them alternate children. This operation preserves the AVL property while ensuring that consecutive red nodes become non-adjacent.
   2. After the rotation(s), we recolor the nodes involved in the rotation. The parent node becomes black, and its original child becomes red. The child's other child (which remains black) also retains its color.
   3. We repeat this process for all nodes in the AVL tree until no consecutive red nodes exist.
3. Finally, we need to ensure that the black-height property (property 5) is satisfied for all paths in the tree. Since we initially colored all nodes black, we might need to adjust the colors to achieve this property while preserving the AVL property.

For example, the AVL Tree in Figure 1 can be transformed into the red-black tree in Figure 2.

A screenshot of a diagram

Description automatically generated

Figure 1: AVL Tree

A diagram of a diagram

Description automatically generated

Figure 2 : Red-Black Tree

Question 2 :

Let **y** be the node at which AVL imbalance occurs.

Let **z** be the parent of **y**.

Let **x** be the child of y with a height **h**.

Example of binary search tree T1:

A diagram of a tree

Description automatically generated

We can apply a simple left rotation to T1 to obtain a balanced AVL tree T2 with the same elements:

A diagram of a triangle with circles and letters

Description automatically generated

Similarly, a simple right rotation to a binary search tree T1 can result into an AVL tree T2 with the same elements.

Question 3 :

Answer in Code

Question 5 :

1. Sequence of vertices visited using depth-first search traversal starting at vertex g:  
   **g, d, c, b, a, e, i, j, f, m, n, o, p, l, h, k.**
2. Sequence of vertices visited using depth-first search traversal starting at vertex g:  
   **g, d, j, k, c, i, f, o, b, e, m, a, n, h, p, l.**
3. Adjacency list Vs Adjacency Matrix

A screenshot of a diagram

Description automatically generated A grid of numbers and letters

Description automatically generated

+ Advantage of Adjacency List: Adjacency lists are very space-efficient for sparse graphs because they only store information about the existing edges, not all possible edges.

- Disadvantage of Adjacency List: Checking whether an edge exists between two vertices in an adjacency list can require iterating through a whole list.

+ Advantage of Adjacency Matrix: Checking whether an edge exists between two vertices is very efficient with an adjacency matrix as it can be done in O(1) time.

- Disadvantage of Adjacency Matrix: adjacency matrices can be very space-inefficient since they require O(V^2) space, where V is the number of vertices.

d. Algorithm to find a path in a graph G that goes through every edge exactly once in each direction:

Algorithm(Graph G):

if G is not connected or has vertices with odd degrees:

return "No such path exists"

Initialize an empty stack S and an empty list C (for the circuit).

Start at any vertex v in G.

while there are unexplored edges in G:

find any unexplored edge e from v to another vertex u

mark edge e as explored

push u onto stack S

set v = u

if all edges in G are explored and v is the starting vertex:

while S is not empty:

pop vertex u from S

add u to the beginning of C

return C //C is the wanted path

else:

return "No such path exists"

Question 6 :

While the splice(w) operation removes the node w from the tree, it's the “u.parent = w.parent” assignment that ensures that the parent-child relationships are updated correctly, ensuring the Red-Black Tree properties are maintained.

Question 7 :

Answer in code.

Question 8 :

To prove that a binary tree with k leaves has a height of at least log k, we can use mathematical induction:

* Base Case: For a binary tree with one leaf (k = 1), the height is 0. Since log(1) = 0, the base case holds.
* Inductive Hypothesis: We assume that for any binary tree with k leaves, its height is at least log k.
* Inductive Step: Let's consider a binary tree with k+1 leaves. We want to show that its height is at least log(k+1). We start with the binary tree with k+1 leaves and remove one leaf from the tree. This results in a binary tree with k leaves. By our inductive hypothesis, the height of the binary tree with k leaves is at least log k. Now, add back the leaf that was removed. This will create a new leaf node at the bottom of one of the subtrees of the tree with k leaves. So, the height of the binary tree with k+1 leaves is now one more than the height of the binary tree with k leaves because we added a new level to the tree.   
    
  Therefore, the height of the binary tree with k+1 leaves is at least log k + 1. Since we know that the height of a binary tree with k leaves is at least log k (inductive hypothesis), and we've shown that when we add one more leaf, the height becomes at least log k + 1, we have proven that a binary tree with k+1 leaves has a height of at least log(k+1).

By induction, we have established that for any positive integer k, a binary tree with k leaves has a height of at least log k.