ASSIGNMENT-2

DATA STRUCTERES USING C

NAME: ZISHAN CHAUDHARY ROLL NO:24\SCA\BCA(AI&ML)077

Q1. "Inorder traversal of Binary Search Tree gives elements in sorted order". Justify this statement using an example.

Ans1. Understanding Inorder Traversal

First, let's recall what inorder traversal means for a binary tree:

Visit the left subtree.

Visit the root node.

Visit the right subtree.

Understanding Binary Search Tree (BST) Property

The defining property of a BST is as follows:

For every node in the tree:

All nodes in its left subtree have values less than the value of the current node.

All nodes in its right subtree have values greater than the value of the current node.

There are no duplicate keys (this can be a simplifying assumption, and variations exist for handling duplicates).

Example BST

Let's consider the following Binary Search Tree:

4

/ \

2 6

/ \ / \

1 3 5 7

Performing Inorder Traversal

Now, let's perform an inorder traversal on this BST, following the steps:

Visit the left subtree of 4: This takes us to the subtree rooted at 2.

Visit the left subtree of 2: This takes us to node 1. (No left subtree, so we visit 1) -> 1

Visit the root of the left subtree: We visit node 2 -> 1, 2

Visit the right subtree of 2: This takes us to node 3. (No subtrees, so we visit 3) -> 1, 2, 3

Visit the root node: We visit node 4 -> 1, 2, 3, 4

Visit the right subtree of 4: This takes us to the subtree rooted at 6.

Visit the left subtree of 6: This takes us to node 5. (No subtrees, so we visit 5) -> 1, 2, 3, 4, 5

Visit the root of the right subtree: We visit node 6 -> 1, 2, 3, 4, 5, 6

Visit the right subtree of 6: This takes us to node 7. (No subtrees, so we visit 7) -> 1, 2, 3, 4, 5, 6, 7

Q2. "Binary Tree can be represented in memory using Structure". Do you agree with this statement? Give reasons in support of your answer.

Ans2. **1. Node Representation as a Structure (or Class):**

* At the fundamental level, each **node** in a binary tree needs to store at least three pieces of information:
  + The **data** or value held by the node.
  + A **pointer (or reference)** to its left child node.
  + A **pointer (or reference)** to its right child node.
* A **structure** (in C/C++) or a **class** (in Java, Python, etc.) is the perfect way to group these related pieces of data together into a single unit. This allows us to define the blueprint for how each node in the binary tree will be organized in memory.

**Example (Conceptual Structure in C-like syntax):**

C

struct TreeNode {

int data;

struct TreeNode \*left;

struct TreeNode \*right;

};

**Example (Conceptual Class in Java):**

Java

class TreeNode {

int data;

TreeNode left;

TreeNode right;

}

**2. Dynamic Memory Allocation:**

* Binary trees often have a dynamic structure, meaning their size and shape can change during program execution. Structures (or classes) facilitate the use of dynamic memory allocation (e.g., using malloc in C/C++ or new in Java) to create individual nodes as needed.
* We can create the root node and then dynamically allocate memory for its left and right children, and so on, building the tree structure in memory as required. The pointers within the TreeNode structure/class allow us to link these dynamically allocated nodes together.

**3. Representing Relationships:**

* The core of a tree structure lies in the parent-child relationships. The pointers (left and right) within the TreeNode structure/class explicitly represent these relationships in memory. By following these pointers, we can traverse the tree, access different nodes, and understand the hierarchical organization of the data.

**4. Flexibility and Adaptability:**

* Using structures/classes allows for flexibility in how we represent the tree. We can easily add more data fields to the TreeNode structure/class if needed (e.g., a pointer to the parent node, flags for specific purposes, etc.).
* Different types of binary trees (like Binary Search Trees, Heaps, etc.) can all be implemented using this fundamental structural representation of nodes with left and right child pointers, with the specific properties enforced through algorithms and operations.

**5. Foundation for Tree Operations:**

* Algorithms for various tree operations (traversal, insertion, deletion, searching) rely on the ability to access the left and right children of a node. The structure-based representation provides this essential access through the left and right pointers.

Q3. Apply Kruskal's algorithm to find the minimum spanning tree on the following graph:

Ans3. Vertices: A, B, C, D, E (5 vertices)

Edges and Weights:

Edge Weight

(A, B) 1

(D, E) 2

(B, C) 3

(C, D) 4

(A, E) 5

(A, C) 7

(A, D) 10

Export to Sheets

Sorted Edges:

Rank Edge Weight

1 (A, B) 1

2 (D, E) 2

3 (B, C) 3

4 (C, D) 4

5 (A, E) 5

6 (A, C) 7

7 (A, D) 10

Export to Sheets

Building the MST:

Consider edge (A, B) with weight 1: Adding this edge does not create a cycle (A and B are initially in different components).

MST Edges: {(A, B)}

Connected Components: {A, B}, {C}, {D}, {E}

Consider edge (D, E) with weight 2: Adding this edge does not create a cycle (D and E are initially in different components).

MST Edges: {(A, B), (D, E)}

Connected Components: {A, B}, {C}, {D, E}

Consider edge (B, C) with weight 3: Adding this edge does not create a cycle (B and C are initially in different components).

MST Edges: {(A, B), (D, E), (B, C)}

Connected Components: {A, B, C}, {D, E}

Consider edge (C, D) with weight 4: Adding this edge does not create a cycle (C and D are initially in different components).

MST Edges: {(A, B), (D, E), (B, C), (C, D)}

Connected Components: {A, B, C, D, E}

We have now added 4 edges to our MST. Since the graph has 5 vertices, an MST will have 5 - 1 = 4 edges. We can stop here.

Minimum Spanning Tree Edges: {(A, B), (D, E), (B, C), (C, D)}

Total Weight of the MST: 1 + 2 + 3 + 4 = 10

The Minimum Spanning Tree consists of the edges (A, B), (D, E), (B, C), and (C, D) with a total weight of 10.

Q4. Q4 Apply Prim's algorithm to find the minimum spanning tree on the following graph:

Ans4. Let's start with vertex **0**.

**Vertices:** 0, 1, 2, 3, 4, 5, 6, 7

**Edges and Weights:** (From the visual representation)

|  |  |
| --- | --- |
| **Edge** | **Weight** |
| (0, 1) | 4 |
| (0, 7) | 8 |
| (1, 2) | 8 |
| (1, 7) | 11 |
| (2, 3) | 7 |
| (2, 8) | (This seems to be a typo or mislabeling in the image, assuming it connects to vertex 6 with weight 2) -> \*\*(2, 6) |
| (2, 5) | 4 |
| (3, 4) | 9 |
| (3, 5) | 14 |
| (4, 5) | 10 |
| (5, 6) | 6 |
| (6, 7) | 1 |
| (1, 7) | 11 |

Export to Sheets

**Building the MST:**

1. **Start with vertex 0.** **MST Vertices: {0}** **MST Edges: {}**
2. **Minimum edge connected to {0}: (0, 1) with weight 4.** **MST Vertices: {0, 1}** **MST Edges: {(0, 1)}**
3. **Minimum edge connected to {0, 1} (excluding cycles): (2, 6) with weight 2** (assuming the '8' connected to '2' is actually connecting to '6' with weight 2, as it leads to a smaller MST). **MST Vertices: {0, 1, 6}** **MST Edges: {(0, 1), (2, 6)}**
4. **Minimum edge connected to {0, 1, 6} (excluding cycles): (6, 7) with weight 1.** **MST Vertices: {0, 1, 6, 7}** **MST Edges: {(0, 1), (2, 6), (6, 7)}**
5. **Minimum edge connected to {0, 1, 6, 7} (excluding cycles): (1, 2) with weight 8.** **MST Vertices: {0, 1, 6, 7, 2}** **MST Edges: {(0, 1), (2, 6), (6, 7), (1, 2)}**
6. **Minimum edge connected to {0, 1, 6, 7, 2} (excluding cycles): (2, 3) with weight 7.** **MST Vertices: {0, 1, 6, 7, 2, 3}** **MST Edges: {(0, 1), (2, 6), (6, 7), (1, 2), (2, 3)}**
7. **Minimum edge connected to {0, 1, 6, 7, 2, 3} (excluding cycles): (3, 4) with weight 9.** **MST Vertices: {0, 1, 6, 7, 2, 3, 4}** **MST Edges: {(0, 1), (2, 6), (6, 7), (1, 2), (2, 3), (3, 4)}**
8. **Minimum edge connected to {0, 1, 6, 7, 2, 3, 4} (excluding cycles): (2, 5) with weight 4.** **MST Vertices: {0, 1, 6, 7, 2, 3, 4, 5}** **MST Edges: {(0, 1), (2, 6), (6, 7), (1, 2), (2, 3), (3, 4), (2, 5)}**

We have now included all 8 vertices in our MST.

**Minimum Spanning Tree Edges (based on the assumption about the edge (2, 6)):**

{(0, 1), (2, 6), (6, 7), (1, 2), (2, 3), (3, 4), (2, 5)}

**Total Weight of the MST:** 4 + 2 + 1 + 8 + 7 + 9 + 4 = **35**

**If we strictly follow the visual and assume the '8' near vertex 2 is an isolated label and there's no edge with weight 8 connected to vertex 2 that leads to a better MST, the process would be slightly different, and the resulting MST might have a higher weight.** However, the most logical interpretation for finding a *minimum* spanning tree involves considering the smallest possible edge weights connecting the growing MST to the remaining vertices. The assumption of an edge (2, 6) with weight 2 seems crucial for a minimal result.

**Final Answer (based on the assumption):** The edges of the Minimum Spanning Tree are {(0, 1), (2, 6), (6, 7), (1, 2), (2, 3), (3, 4), (2, 5)} with a total weight of 35.

Q5. Compare and Contrast the Linear probing and Chaining methods of collision resolution in Hashing. Also explain these techniques with the help of examples.

Ans5. Comparing and Contrasting Linear Probing and Chaining in Hashing

Both Linear Probing and Chaining are popular techniques used to handle collisions in hash tables. A collision occurs when two or more keys map to the same index in the hash table. Let's compare and contrast these two methods:

Similarities:

Goal: Both techniques aim to resolve collisions and allow the insertion, search, and deletion of key-value pairs even when multiple keys map to the same index.

Impact on Performance: Both methods can degrade the performance of hash table operations if collisions become too frequent. The average time complexity for search, insertion, and deletion can move away from O(1) towards O(n) in the worst case.

Differences:

Feature Linear Probing Chaining (Separate Chaining)

Collision Handling Probes for the next available slot in the table. Maintains a linked list (or other data structure) at each hash table index to store colliding keys.

Data Structure Uses the hash table itself for storing all elements. Uses an auxiliary data structure (typically a linked list) to store colliding elements.

Space Usage Can utilize the empty slots within the hash table. Requires extra space for the linked lists (or other data structures).

Performance (Average) Can experience primary clustering, leading to longer probe sequences and reduced performance. Generally provides better average-case performance, especially with a good load factor.

Performance (Worst) Can degrade significantly to O(n) if the table becomes full and long clusters form. Can degrade to O(k) in the worst case, where k is the number of keys hashed to the same index (length of the chain).

Implementation Simpler to implement initially. Slightly more complex to implement due to the need for managing the auxiliary data structure.

Deletion Deletion requires careful handling (e.g., using tombstone values) to avoid breaking probe sequences for other elements. Deletion is straightforward; simply remove the node from the linked list.

Load Factor Impact Performance degrades rapidly as the load factor approaches 1 (table becomes full). Performance degrades more gracefully with increasing load factor, as long as the chain lengths remain relatively short.

Cache Performance Can have better cache locality due to contiguous storage of elements in the table. May have poorer cache locality due to the scattered nature of linked list nodes in memory.

Export to Sheets

Explanation with Examples:

Let's consider a hash table of size 10 (indices 0 to 9) and a simple hash function h(key) = key % 10.

1. Linear Probing:

Insertion:

Insert key 22: h(22) = 22 % 10 = 2. Place 22 at index 2.

Insert key 42: h(42) = 42 % 10 = 2. Index 2 is occupied. Probe the next slot (index 3), which is empty. Place 42 at index 3.

Insert key 32: h(32) = 32 % 10 = 2. Index 2 is occupied. Probe index 3 (occupied). Probe index 4 (empty). Place 32 at index 4.

Insert key 52: h(52) = 52 % 10 = 2. Indices 2, 3, 4 are occupied. Probe index 5 (empty). Place 52 at index 5.

The hash table would look like this:

| Index | Value |

| :---- | :---- |

| 0 | |

| 1 | |

| 2 | 22 |

| 3 | 42 |

| 4 | 32 |

| 5 | 52 |

| 6 | |

| 7 | |

| 8 | |

| 9 | |

Notice the primary clustering around index 2, where consecutive slots are filled due to collisions with the same initial hash value.

Search:

Search for 42: h(42) = 2. Check index 2 (not 42). Probe index 3 (is 42). Found.

Search for 32: h(32) = 2. Check index 2 (not 32). Probe index 3 (not 32). Probe index 4 (is 32). Found.

Search for 62: h(62) = 2. Check index 2 (not 62). Probe index 3 (not 62). Probe index 4 (not 62). Probe index 5 (not 62). Continue probing until an empty slot is found or the entire table is searched (not found).

Deletion:

Deleting 42 requires marking index 3 as a "tombstone" to indicate that it was occupied but is now empty. This is crucial because if we simply make it empty, a search for 32 (which initially hashed to 2) would stop prematurely at the empty slot.

2. Chaining (Separate Chaining):

Insertion:

Insert key 22: h(22) = 2. Create a linked list at index 2 and add 22.

Insert key 42: h(42) = 2. Go to the linked list at index 2 and add 42.

Insert key 32: h(32) = 2. Go to the linked list at index 2 and add 32.

Insert key 52: h(52) = 2. Go to the linked list at index 2 and add 52.

Insert key 15: h(15) = 5. Create a linked list at index 5 and add 15.

The hash table would conceptually look like this:

| Index | Linked List |

| :---- | :------------------ |

| 0 | null |

| 1 | null |

| 2 | 22 -> 42 -> 32 -> 52 |

| 3 | null |

| 4 | null |

| 5 | 15 -> null |

| 6 | null |

| 7 | null |

| 8 | null |

| 9 | null |

Search:

Search for 42: h(42) = 2. Go to the linked list at index 2 and traverse it to find 42.

Search for 15: h(15) = 5. Go to the linked list at index 5 and traverse it to find 15.

Search for 62: h(62) = 2. Go to the linked list at index 2 and traverse it. If 62 is not found after traversing the entire list, it's not in the table.

Deletion:

Deleting 42: h(42) = 2. Go to the linked list at index 2 and remove the node containing 42.