Assignment 2 - Set C: Theoretical Questions and Answers

Advanced BST Operations and Analysis

Question 1

Which data structure is required to display nodes of Binary Search Tree depth wise?

Answer

Queue data structure is required to display nodes of Binary Search Tree depth-wise.

Explanation

To display nodes level by level (depth-wise) in a Binary Search Tree, we use the Breadth-First Search (BFS) or Level Order Traversal algorithm, which requires a Queue.

Why Queue?

- Queue follows First-In-First-Out (FIFO) principle
- It helps process nodes level by level from left to right
- When we visit a node, we enqueue its children (left then right)
- This ensures that all nodes at level L are processed before nodes at level L+1

Algorithm

1. Start with root node and enqueue it

2. While queue is not empty:

- Dequeue a node
- Print/process the node
- Enqueue its left child (if exists)
- Enqueue its right child (if exists)

Example

```
Tree:       5
           / \
          3   7
         /   / \
        2   6   8

Level Order (using Queue): 5 3 7 2 6 8
```

Queue Operations

```
Initial: [5]
Step 1: Dequeue 5, print 5, enqueue 3,7 → [3,7]
Step 2: Dequeue 3, print 3, enqueue 2 → [7,2]
Step 3: Dequeue 7, print 7, enqueue 6,8 → [2,6,8]
Step 4: Dequeue 2, print 2 → [6,8]
Step 5: Dequeue 6, print 6 → [8]
Step 6: Dequeue 8, print 8 → []
```

Question 2

Write a C program to display nodes of Binary Search Tree depth wise.

Answer

```
#include
#include

typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

typedef struct QueueNode {
    Node *treeNode;
    struct QueueNode *next;
} QueueNode;

typedef struct Queue {
    QueueNode *front, *rear;
} Queue;

Queue* createQueue() {
    Queue *q = (Queue*)malloc(sizeof(Queue));
    q->front = q->rear = NULL;
    return q;
}

void enqueue(Queue *q, Node *node) {
    QueueNode *temp = (QueueNode*)malloc(sizeof(QueueNode));
    temp->treeNode = node;
    temp->next = NULL;

    if (q->rear == NULL) {
        q->front = q->rear = temp;
        return;
    }
    q->rear->next = temp;
    q->rear = temp;
}
```

```
Node* dequeue(Queue *q) {
    if (q->front == NULL) return NULL;

    QueueNode *temp = q->front;
    Node *node = temp->treeNode;
    q->front = q->front->next;

    if (q->front == NULL) q->rear = NULL;

    free(temp);
    return node;
}

int isEmpty(Queue *q) {
    return q->front == NULL;
}

void levelOrderTraversal(Node *root) {
    if (root == NULL) return;

    Queue *q = createQueue();
    enqueue(q, root);

    printf("Level Order Traversal (Depth-wise): ");

    while (!isEmpty(q)) {
        Node *current = dequeue(q);
        printf("%d ", current->data);

        if (current->left != NULL) enqueue(q, current->left);
        if (current->right != NULL) enqueue(q, current->right);
    }
    printf("\n");
}
```

Question 3

Write a C program to compare two binary search trees (node wise comparison).

Answer

```
int compareNodeWise(Node *tree1, Node *tree2) {
    // Both trees are empty
    if (tree1 == NULL && tree2 == NULL) {
        return 1;
    }

    // One tree is empty, other is not
    if (tree1 == NULL || tree2 == NULL) {
        return 0;
    }

    // Compare current node data
    if (tree1->data != tree2->data) {
        return 0;
    }

    // Recursively compare left and right subtrees
    return compareNodeWise(tree1->left, tree2->left) &&
           compareNodeWise(tree1->right, tree2->right);
}
```

Explanation

This function performs a node-wise comparison of two binary search trees by:

1. **Base Cases:**

- Both nodes are NULL → trees are equal (return 1)
- One node is NULL, other is not → trees are not equal (return 0)

2. **Recursive Case:**

- Compare data of current nodes
- If data doesn't match → return 0
- If data matches → recursively compare left and right subtrees
- Return 1 only if both subtrees are also equal

Time Complexity: O(n) where n is number of nodes

Space Complexity: O(h) where h is height (recursion stack)

Example

```
Tree 1:    5          Tree 2:    5
          / \                   / \
         3   7                 3   7

compareNodeWise(Tree1, Tree2) returns 1 (Equal)

Tree 1:    5          Tree 2:    5
```

```
          / \                    / \
         3   7                  3   8

compareNodeWise(Tree1, Tree2) returns 0 (Not Equal)
```

Question 4

How to implement mirror() and copy() functions without recursion?

Answer

1. Mirror Function Without Recursion

```
void mirrorIterative(Node *root) {
    if (root == NULL) return;

    Queue *q = createQueue();
    enqueue(q, root);

    while (!isEmpty(q)) {
        Node *current = dequeue(q);

        // Swap left and right children
        Node *temp = current->left;
        current->left = current->right;
        current->right = temp;

        // Enqueue children for processing
        if (current->left != NULL) enqueue(q, current->left);
        if (current->right != NULL) enqueue(q, current->right);
    }
}
```

Approach:

- Uses Queue for level-order traversal
- At each node, swap left and right children
- Continue until all nodes are processed
- Time Complexity: O(n)
- Space Complexity: O(w) where w is maximum width of tree

2. Copy Function Without Recursion

```
Node* copyIterative(Node *root) {
    if (root == NULL) return NULL;

    Node *newRoot = createNode(root->data);
    Queue *q1 = createQueue();  // For original tree
    Queue *q2 = createQueue();  // For new tree

    enqueue(q1, root);
    enqueue(q2, newRoot);

    while (!isEmpty(q1)) {
        Node *current1 = dequeue(q1);
        Node *current2 = dequeue(q2);

        // Copy left child
        if (current1->left != NULL) {
            current2->left = createNode(current1->left->data);
            enqueue(q1, current1->left);
            enqueue(q2, current2->left);
        }

        // Copy right child
        if (current1->right != NULL) {
            current2->right = createNode(current1->right->data);
            enqueue(q1, current1->right);
            enqueue(q2, current2->right);
        }
    }

    return newRoot;
}
```

Approach:

- Uses two parallel queues for original and new tree
- Process nodes level by level
- Create new nodes and link them appropriately
- Time Complexity: O(n)
- Space Complexity: O(w) where w is maximum width

Key Difference from Recursion

- **Recursion:** Uses call stack (implicit)
- **Iterative:** Uses queue (explicit)

- Both have same time complexity but iterative avoids stack overflow for deep trees

Question 5

How to convert singly linked list to binary search tree?

Answer

Method 1: Simple Approach (O(n))

```
Node* sortedListToBST(int arr[], int start, int end) {
    if (start > end) return NULL;

    int mid = (start + end) / 2;
    Node *root = createNode(arr[mid]);

    root->left = sortedListToBST(arr, start, mid - 1);
    root->right = sortedListToBST(arr, mid + 1, end);

    return root;
}

Node* linkedListToBST(LinkedListNode *head) {
    // Step 1: Count nodes in linked list
    int count = 0;
    LinkedListNode *temp = head;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }

    // Step 2: Copy linked list data to array
    int arr[count];
    temp = head;
    for (int i = 0; i < count; i++) {
        arr[i] = temp->data;
        temp = temp->next;
    }

    // Step 3: Build BST from sorted array
    return sortedListToBST(arr, 0, count - 1);
}
```

Time Complexity: O(n)

Space Complexity: O(n) for array

Method 2: Optimized In-place Approach (O(n))

```
Node* listToBSTUtil(LinkedListNode **head, int n) {
    if (n <= 0) return NULL;

    // Recursively construct left subtree
    Node *left = listToBSTUtil(head, n / 2);

    // Create root node with current linked list node
    Node *root = createNode((*head)->data);
    root->left = left;

    // Move to next linked list node
    *head = (*head)->next;

    // Recursively construct right subtree
    root->right = listToBSTUtil(head, n - n / 2 - 1);

    return root;
}

Node* linkedListToBSTOptimized(LinkedListNode *head) {
    int count = 0;
    LinkedListNode *temp = head;

    // Count nodes
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }

    return listToBSTUtil(&head, count);
}
```

Time Complexity: O(n)

Space Complexity: O(log n) for recursion stack

Algorithm (Method 2)

1. Count total nodes in linked list

2. Use middle element as root (simulated by recursion)

3. Recursively build left subtree with first half

4. Create root with current list node

5. Recursively build right subtree with remaining nodes

Example

```
Linked List: 1 -> 2 -> 3 -> 4 -> 5

Resulting BST:
        3
       / \
      1   4
       \   \
        2   5
```

This creates a balanced BST from a sorted linked list.

Note: If linked list is unsorted, first sort it or insert elements one by one into BST.