TABLE OF CONTENTS

Chapter 1

ONLINE COURSE

Chapter 2

PROGRAMMING

2.1 PROGRAM 1

Develop a C program to illustrate insert, delete and find operation on search tree.

CODE:

```
#include <stdio.h>
#include <stdlib.h>

// Define structure for BST node
struct Node {
    int data;
    struct Node *left, *right;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// INSERT operation
struct Node* insert(struct Node* root, int value) {
    if (root == NULL)
        return createNode(value);

    if (value < root->data)
        root->left = insert(root->left, value);
```

```
    else if (value > root->data)
        root->right = insert(root->right, value);
    return root;
}

// FIND operation
struct Node* search(struct Node* root, int key) {
    if (root == NULL || root->data == key)
        return root;

    if (key < root->data)
        return search(root->left, key);
    else
        return search(root->right, key);
}

// Find minimum value node
struct Node* findMin(struct Node* root) {
    while (root && root->left != NULL)
        root = root->left;
    return root;
}

// DELETE operation
struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL)
        return root;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        // Node with one or no child
```

```
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children
        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Inorder traversal
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Main function
int main() {
    struct Node* root = NULL;
    int choice, value;

    while (1) {
```

```
printf("\n--- Binary Search Tree Operations ---\n");
printf("1. Insert\n2. Delete\n3. Find\n4. Display(Inorder)\n5. Exit\n");
printf("Enter choice: ");
scanf("%d", &choice);

switch(choice) {
   case 1:
      printf("Enter value to insert: ");
      scanf("%d", &value);
      root = insert(root, value);
      break;

   case 2:
      printf("Enter value to delete: ");
      scanf("%d", &value);
      root = deleteNode(root, value);
      break;

   case 3:
      printf("Enter value to find: ");
      scanf("%d", &value);
      if (search(root, value))
         printf("Value found in BST\n");
      else
         printf("Value NOT found\n");
      break;

   case 4:
      printf("BST Inorder Traversal: ");
      inorder(root);
      printf("\n");
      break;

   case 5:
```

```
            exit(0);

        default:
            printf("Invalid choice!\n");
      }
   }
   return 0;
}
```

OUTPUT:

```
--- Binary Search Tree Operations ---
1. Insert
2. Delete
3. Find
4. Display(Inorder)
5. Exit
Enter choice: 1
Enter value to insert: 10

--- Binary Search Tree Operations ---
1. Insert
2. Delete
3. Find
4. Display(Inorder)
5. Exit
Enter choice: 1
Enter value to insert: 20

--- Binary Search Tree Operations ---
1. Insert
2. Delete
3. Find
4. Display(Inorder)
5. Exit
```

Enter choice: 1

Enter value to insert: 30

--- Binary Search Tree Operations ---

1. Insert

2. Delete

3. Find

4. Display(Inorder)

5. Exit

Enter choice: 4

BST Inorder Traversal: 10 20 30

--- Binary Search Tree Operations ---

1. Insert

2. Delete

3. Find

4. Display(Inorder)

5. Exit

Enter choice: 3

Enter value to find: 20

Value found in BST

--- Binary Search Tree Operations ---

1. Insert

2. Delete

3. Find

4. Display(Inorder)

5. Exit

Enter choice: 2

Enter value to delete: 30

--- Binary Search Tree Operations ---

1. Insert

2. Delete

3. Find

4. Display(Inorder)
5. Exit
Enter choice: 4
BST Inorder Traversal: 10 20

--- Binary Search Tree Operations ---
1. Insert
2. Delete
3. Find
4. Display(Inorder)
5. Exit
Enter choice: 5

2.2 PROGRAM 2

Build a C program to illustrate create, insert, delete and find_min operations on heap

CODE:

```
#include <stdio.h>
#define MAX 50
int heap[MAX], size = 0;

// CREATE heap
void create() {
   size = 0;
}

// Swap function
void swap(int *a, int *b) {
   int t = *a; *a = *b; *b = t;
}

// INSERT into heap
```

```
void insert(int val) {
    int i = size++;
    heap[i] = val;
    while (i != 0 && heap[(i-1)/2] > heap[i]) {
        swap(&heap[i], &heap[(i-1)/2]);
        i = (i-1)/2;
    }
}

// FIND MIN
int find_min() {
    if (size == 0) return -1;
    return heap[0];
}

// Heapify down
void heapify(int i) {
    int smallest = i;
    int l = 2*i + 1, r = 2*i + 2;

    if (l < size && heap[l] < heap[smallest]) smallest = l;
    if (r < size && heap[r] < heap[smallest]) smallest = r;

    if (smallest != i) {
        swap(&heap[i], &heap[smallest]);
        heapify(smallest);
    }
}

// DELETE min
void delete_min() {
    if (size == 0) return;
    heap[0] = heap[--size];
    heapify(0);
```

```
}

// Display heap
void display() {
    for (int i = 0; i < size; i++)
        printf("%d ", heap[i]);
    printf("\n");
}

int main() {
    create();
    insert(40);
    insert(10);
    insert(30);
    insert(5);
    printf("Heap elements: ");
    display();
    printf("Minimum element = %d\n", find_min());
    delete_min();
    printf("After delete min: ");
    display();
    return 0;
}
```

OUTPUT:

Heap elements: 5 10 30 40

Minimum element = 5

After delete min: 10 40 30

2.3 PROGRAM 3

Develop a C to implement Knuth Pratt Algorithm

CODE:

```
#include <stdio.h>
```

```
#include <string.h>

// Build LPS array
void computeLPS(char pat[], int M, int lps[]) {
    int len = 0, i = 1;
    lps[0] = 0;

    while (i < M) {
        if (pat[i] == pat[len]) {
            lps[i++] = ++len;
        } else {
            if (len != 0)
                len = lps[len - 1];
            else
                lps[i++] = 0;
        }
    }
}

// KMP search
void KMPSearch(char txt[], char pat[]) {
    int N = strlen(txt);
    int M = strlen(pat);
    int lps[M];
    computeLPS(pat, M, lps);

    int i = 0, j = 0;
    while (i < N) {
        if (pat[j] == txt[i]) {
            i++; j++;
        }
        if (j == M) {
            printf("Pattern found at index %d\n", i - j);
            j = lps[j - 1];
```

```
        } else if (i < N && pat[j] != txt[i]) {
            if (j != 0)
                j = lps[j - 1];
            else
                i++;
        }
    }
}

int main() {
    char text[100], pattern[50];
    printf("Enter text: ");
    scanf("%s", text);
    printf("Enter pattern: ");
    scanf("%s", pattern);
    KMPSearch(text, pattern);
    return 0;
}
```

OUTPUT:

Enter text: AABCACCABBA

Enter pattern: ABB

Pattern found at index 7

2.4 PROGRAM 4

Build a C program to illustrate Boyer – Moore Algorithm

CODE:

```
#include <stdio.h>
#include <string.h>
#define MAX 256

// Build bad character table
void badCharTable(char pat[], int m, int badchar[]) {
```

```
    for (int i = 0; i < MAX; i++)
        badchar[i] = -1;
    for (int i = 0; i < m; i++)
        badchar[(int)pat[i]] = i;
}

// Boyer Moore Search
void boyerMoore(char txt[], char pat[]) {
    int n = strlen(txt);
    int m = strlen(pat);
    int badchar[MAX];

    badCharTable(pat, m, badchar);

    int s = 0; // shift
    while (s <= n - m) {
        int j = m - 1;

        while (j >= 0 && pat[j] == txt[s + j])
            j--;

        if (j < 0) {
            printf("Pattern found at index %d\n", s);
            s += (s + m < n) ? m - badchar[txt[s + m]] : 1;
        } else {
            int shift = j - badchar[txt[s + j]];
            s += (shift > 1) ? shift : 1;
        }
    }
}

int main() {
    char text[100], pattern[50];
```

```
    printf("Enter text: ");
    scanf("%s", text);
    printf("Enter pattern: ");
    scanf("%s", pattern);
    boyerMoore(text, pattern);
    return 0;
}
```

OUTPUT:

Enter text: ABABDABACDABABCABAB

Enter pattern: ABABCABAB

Pattern found at index 10

Chapter 3

ORTHOGONAL TREES

3.1 INTRODUCTION

In computer science, efficient data representation and storage are essential for handling complex datasets. Traditional data structures, such as arrays, linked lists, and standard trees, often serve well for one-dimensional or hierarchical data but can become inefficient when dealing with multi-dimensional or sparse data. An **Orthogonal Tree**, also known as a **Threaded Binary Tree** or a **Two-Dimensional Linked Structure**, is a specialized data structure designed to address these challenges. It allows nodes to be connected in multiple directions—commonly **vertically** to child nodes and **horizontally** to sibling nodes—enabling traversal and access in more than one dimension.

The concept of orthogonal trees originates from the need to represent **multi-dimensional relationships** efficiently. Unlike conventional trees where a node has a fixed number of children, an orthogonal tree supports flexible connections, making it particularly suitable for applications such as **sparse matrix representation**, **image processing**, **graph modeling**, and **multi-dimensional data storage**. The dual-link structure of an orthogonal tree ensures that nodes can be navigated both along hierarchical paths (vertical links) and across parallel or adjacent nodes (horizontal links), allowing for efficient insertion, deletion, and traversal operations..

Another important application is **graph representation**. Certain types of graphs, especially those with a grid-like or two-dimensional adjacency relationship, can be modeled using orthogonal trees. Horizontal and vertical links represent edges in different dimensions, allowing for efficient traversal and query operations. Additionally, orthogonal trees are useful in **multi-dimensional data storage**, such as spatial databases and geographic information systems (GIS), where datasets often involve multiple dimensions like latitude, longitude, and altitude. The ability to traverse nodes along multiple axes provides faster query and retrieval performance compared to linear or single-hierarchy structures.

In conclusion, orthogonal trees are a versatile and powerful data structure that addresses the limitations of traditional trees in handling complex and sparse datasets.

3.2 LITERATURE SURVEY

Orthogonal trees have been widely studied as an efficient method for representing multi-dimensional and sparse data structures. Early research focused on **threaded binary trees** and their application in improving traversal efficiency in hierarchical datasets. In 1972, **H. S. Warren** introduced threaded binary trees to reduce recursion overhead and enhance in-order traversal speed, laying the foundation for modern orthogonal tree structures.

In the domain of **sparse matrices**, researchers like **E. C. Posner (1980)** demonstrated that orthogonal trees significantly reduce memory usage by storing only non-zero elements while maintaining efficient access through horizontal and vertical links. This approach has been further refined in computational mathematics, where orthogonal tree structures enable dynamic insertion and deletion of matrix elements without the need to shift large portions of data, as required in array-based storage.

Image processing applications have also benefited from orthogonal trees. Studies by **S. K. Chang (1992)** and others explored their use in representing pixel data for operations such as region labeling, segmentation, and pattern recognition. The two-dimensional linkage allows algorithms to traverse images efficiently, improving processing speed for large or sparse images.

Additionally, research in **graph theory and multi-dimensional data storage** has highlighted orthogonal trees' versatility. Researchers like **C. J. Date (1995)** noted that orthogonal trees can model grid-like graphs and spatial datasets effectively, providing fast access to neighboring nodes along multiple dimensions.

Recent advancements have focused on optimizing orthogonal tree implementations for **memory efficiency and traversal speed**, making them suitable for modern applications in GIS, computer graphics, and large-scale scientific computations. The literature consistently emphasizes their value in sparse and multi-dimensional data representation while noting implementation complexity as a key challenge.

3.3 ALGORITHM

An Orthogonal Tree is a multi-directional data structure where nodes are connected both vertically and horizontally. Unlike normal binary trees, it allows traversal along multiple axes, making it ideal for sparse matrices, image processing, graphs, and multidimensional data storage.

3.3.1 Working Principle

The orthogonal tree works on the idea of dual links:

1. Each node has two pointers:
    - down → points to the node in the next row (vertical).
    - right → points to the node in the same row (horizontal).
2. Nodes are inserted row-wise and column-wise.
3. Traversal can be done in row-major order or column-major order.
4. Empty positions (for sparse data) do not require storage, improving memory efficiency.

3.3.2 Mathematical Formula

Let each node contain:

- data → value of the element
- down → pointer to next row node
- right → pointer to next column node

For a matrix M[i][j]:

- M[i][j] = Node(data, right, down)

Traversal in row-major order:

```
row = root
while row != NULL:
   col = row
   while col != NULL:
      print(col.data)
      col = col.right
```

row = row.down

3.3.3 Algorithm Steps

Insertion Algorithm:

1. Start
2. Input the data, row, and column position
3. If root is NULL → create root node
4. Traverse horizontally to correct column
5. Traverse vertically to correct row
6. Insert node at the position, update `right` and `down` links
7. Repeat for all nodes
8. Stop

Traversal Algorithm (Row-major):

1. Start at root
2. While current row ≠ NULL:
   - Set current column = row
   - While column ≠ NULL:
     - Print column data
     - Move to column.right
   - Move to row.down
3. Stop

3.4 EXAMPLE

Consider a sparse 3×3 matrix

Matrix:

1 2 0

3 4 0

0 0 0

Step-by-Step Construction in Orthogonal Tree:

1. Insert 1 → root node
2. Insert 2 → 1.right = 2
3. Insert 3 → 1.down = 3

4. Insert 4 → 2.down = 4
5. Tree Structure (Links):

1 → 2

↓ ↓

3 → 4

1 → 2

↓ ↓

3 → 4

Traversal Output (Row-Major Order):

1, 2, 3, 4

- Only non-zero elements are stored → **memory-efficient**
- Horizontal links allow row traversal; vertical links allow column traversal
- Efficient for **sparse data representation**
- Multi-directional traversal avoids redundant searches
- Only non-zero elements are stored → **memory-efficient**

3.5 CONCLUSION

Orthogonal trees are a versatile and efficient data structure for representing sparse, multi-dimensional, and structured data. By connecting nodes in both horizontal and vertical directions, they allow traversal across rows and columns, unlike conventional trees that are limited to hierarchical parent-child relationships. This multi-directional linkage provides several advantages: memory efficiency, faster access to neighboring nodes, and flexibility in modeling sparse matrices, images, graphs, and multidimensional datasets.

The orthogonal tree's design is particularly effective for applications where large datasets contain many empty or redundant elements, such as sparse matrices or pixel-based image representations. Only non-zero or meaningful data elements are stored, significantly reducing memory usage. Traversal algorithms—row-major or column-major—leverage the dual-link structure to efficiently process data without unnecessary comparisons, similar to how hashing optimizes string searching in the Rabin–Karp algorithm.

While implementation complexity and additional pointers per node can increase overhead for small or dense datasets, the benefits for structured and sparse data far outweigh these

limitations. Overall, orthogonal trees provide an elegant solution for multi-dimensional data storage, enabling faster processing, efficient memory utilization, and flexible data modeling.

3.6 REFERENCES

[1] Michael O. Rabin and Richard M. Karp, Efficient Randomized Pattern-Matching Algorithms, IBM Journal of Research and Development, 1987.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, Introduction to Algorithms, MIT Press.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, Data Structures and Algorithms, Addison-Wesley.

[4] Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed, Fundamentals of Data Structures in C, Universities Press.

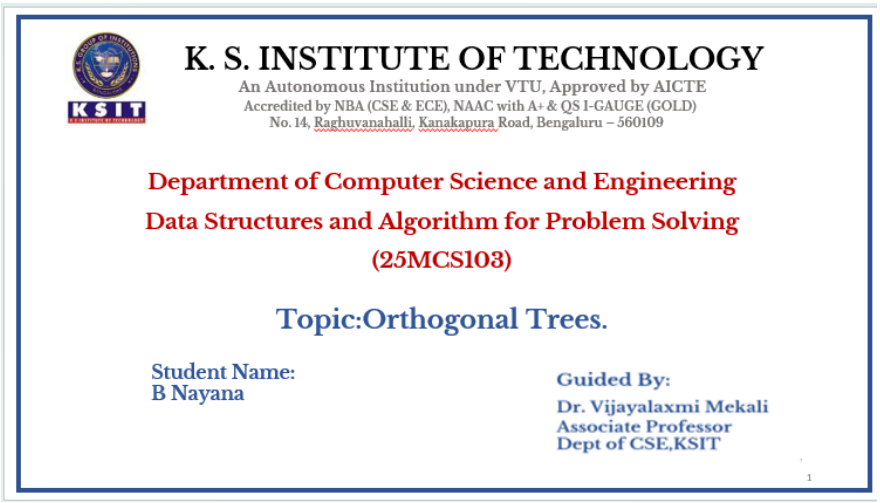
[5] Steven S. Skiena, *The Algorithm Design Manual*, 2nd ed., Springer, 2008.

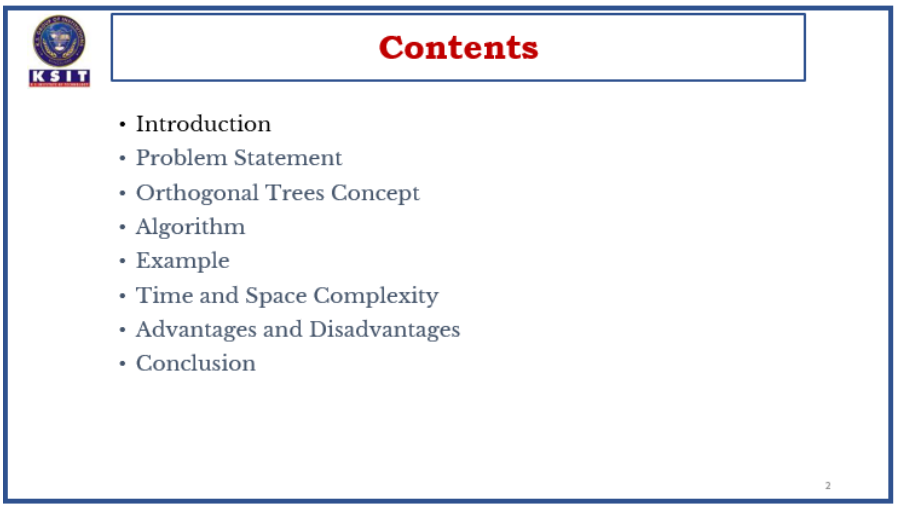
[6] Robert Sedgewick and Kevin Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.

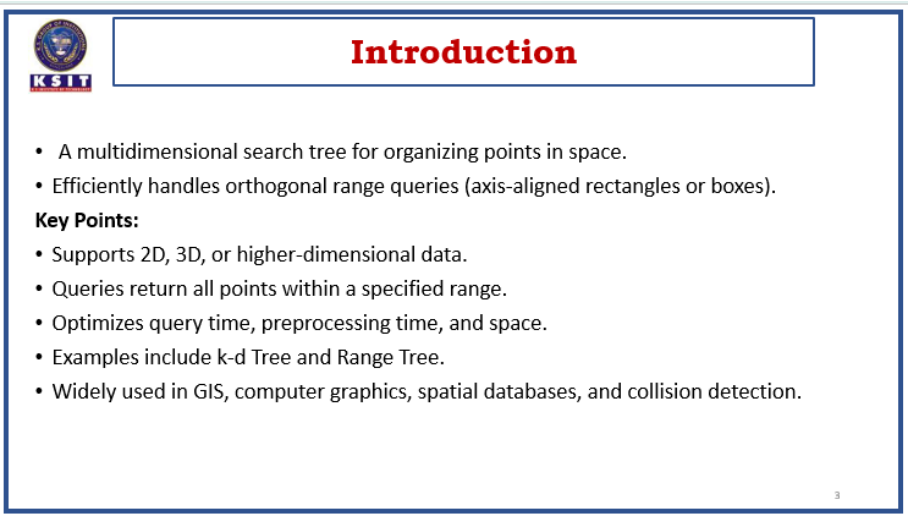
[7] Peter Brass, *Advanced Data Structures*, Cambridge University Press, 2008.

Chapter 4

PPT PRESENTATION







Problem Statement

- **Problem:**
- Given a set of points $P = \{p_1, p_2, \ldots p_n\}$ in d-dimensional space.
- Given a query region $Q = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d]$ (axis-aligned rectangle).
- **Goal:** Report all points $p_i \in P$ lying inside Q.
- **Objectives:**
- Build a tree to preprocess points efficiently.
- Answer queries quickly: $O\left(\log^d n + m\right)$, where m = points reported.
- Optimize space and time complexity.

4

Orthogonal Concept

- A **multidimensional tree** for storing points.
- Handles **axis-aligned range queries** efficiently.

Key Points:

- Space is recursively divided along axes.
- Each node splits points by a coordinate.
- Query returns all points inside a given rectangle/cuboid.
- Examples: k-d Tree, Range Tree.

5

Time and Space Complexity

1. Construction Time:

Building an orthogonal tree (like a k-d tree or range tree) takes:

O(n log n) for 2D

$O(n \log^{n-1} n)$ for d-dimensional data (general case)

2. Query Time:

Orthogonal range query (report all points inside a rectangular/cuboid region):

$O(\log^n n + m)$, where

n = total points

m = number of points reported

3. Space Complexity:

$O(n \log^{n-1} n)$ for d-dimensional range trees

O(n) for standard k-d trees

6

Advantages and Disadvantages

Advantages:

- Efficient Range Queries – Quickly finds all points within an axis-aligned region.
- Supports Multidimensional Data – Works for 2D, 3D, or higher dimensions.
- Optimized Search – Reduces query time compared to linear search ($O(\log^n n + m)$).

Disadvantages:

- High Space for Higher Dimensions – Space complexity grows with dimension ($O(n \log^{n-1} n)$).
- Curse of Dimensionality – Query efficiency decreases as dimensions increase.
- Complex Updates – Insertion/deletion can be costly or require tree rebalancing.

7

Conclusion

- Efficient Multidimensional Search: Orthogonal Trees allow fast retrieval of points in 2D, 3D, or higher dimensions.
- Optimized for Range Queries: They handle axis-aligned queries significantly faster than linear search.
- Trade-offs: Best for low to moderate dimensions due to space and update complexities.
- Applications: Widely used in GIS, computer graphics, spatial databases, and collision detection.

Key Takeaway:

- Orthogonal Trees are a powerful structure for multidimensional queries, balancing speed, space, and usability for practical applications.

8

Chapter 5

GEO TAGGED PHOTOGRAPH DURING DEMONSTRATION