Zakho Technical College

Department of Computer Information System

Data Structures and Algorithms

5. Analysis (Searching Methods)

Lecturers:

Sipan M. Hameed

Ahmed Jamil

www.sipan.dev

2024-2025

5. Table of Contents

5. Table of Contents 2

5.1 1. Linear (Sequential) Search

5.1.1 Overview

- **Purpose**: Find a target value in a list by checking each element sequentially.
- **Applicability**: Works on **sorted or unsorted arrays**.
- **Time Complexity**:
  - **Best Case**: `O(1)` (element is at the first index).
  - **Worst Case**: `O(n)` (element is at the last index or not present).
- **Space Complexity**: `O(1)` (no extra memory required).

5.1.2 C# Code Example

csharp

```
public class LinearSearch
{
    public static int Search(int[] array, int target)
    {
        for (int i = 0; i < array.Length; i++)
        {
            if (array[i] == target)
                return i; // Return index if found
        }
        return -1; // Target not found
    }
}

// Usage:
int[] numbers = { 5, 2, 9, 1, 3 };
int index = LinearSearch.Search(numbers, 9); // Returns 2
```

5.2 2. Binary Search

6. Overview

- **Purpose**: Efficiently find a target in a **sorted array** by repeatedly dividing the search interval in half.
- **Applicability**: Requires a **sorted array** (ascending or descending).
- **Time Complexity**:

    - **Best/Worst/Average Case**: `O(log n)`.
- **Space Complexity**:
    - **Iterative**: `O(1)`.
    - **Recursive**: `O(log n)` (call stack depth).

C# Code Example

Iterative Approach

csharp

```
public class BinarySearch
{
    public static int IterativeSearch(int[] sortedArray, int target)
    {
        int low = 0;
        int high = sortedArray.Length - 1;

        while (low <= high)
        {
            int mid = low + (high - low) / 2; // Avoid overflow
            if (sortedArray[mid] == target)
                return mid;
            else if (sortedArray[mid] < target)
                low = mid + 1; // Search right half
            else
                high = mid - 1; // Search left half
        }
        return -1; // Target not found
    }
}

// Usage:
int[] sortedNumbers = { 1, 3, 5, 7, 9 };
int index = BinarySearch.IterativeSearch(sortedNumbers, 5); // Returns
2
```

5.3 3. Binary Search Tree (BST)

7. Overview

- **Purpose**: A tree-based data structure where each node has at most two children, maintaining the property:
  - Left child < Parent < Right child.
- **Operations**:
  - **Insertion**: `O(log n)` (average), `O(n)` (worst, if tree is skewed).
  - **Search**: `O(log n)` (average), `O(n)` (worst).
  - **Deletion**: `O(log n)` (average), `O(n)` (worst).
- **Space Complexity**: `O(n)` (stores all nodes).

8. BST Node Class

csharp

```
public class BSTNode
{
    public int Data { get; set; }
    public BSTNode Left { get; set; }
    public BSTNode Right { get; set; }

    public BSTNode(int data)
    {
        Data = data;
        Left = Right = null;
    }
}
```

5.4 BST Operations

9. Insertion

csharp

```
public class BST
{
    public BSTNode Root { get; private set; }

    public void Insert(int data)
    {
```

```
        Root = InsertRec(Root, data);
    }

    private BSTNode InsertRec(BSTNode root, int data)
    {
        if (root == null)
            return new BSTNode(data);

        if (data < root.Data)
            root.Left = InsertRec(root.Left, data);
        else if (data > root.Data)
            root.Right = InsertRec(root.Right, data);

        return root;
    }
}

// Usage:
BST bst = new BST();
bst.Insert(10);
bst.Insert(5);
bst.Insert(15);
```

10. Search

csharp

```
public class BST
{
    public bool Search(int target)
    {
        return SearchRec(Root, target);
    }

    private bool SearchRec(BSTNode root, int target)
    {
```

```
        if (root == null) return false;
        if (root.Data == target) return true;

        return (target < root.Data)
            ? SearchRec(root.Left, target)
            : SearchRec(root.Right, target);
    }
}

// Usage:
bool exists = bst.Search(15); // Returns true
```

11. Deletion

csharp

```
public class BST
{
    public void Delete(int data)
    {
        Root = DeleteRec(Root, data);
    }

    private BSTNode DeleteRec(BSTNode root, int data)
    {
        if (root == null) return root;

        if (data < root.Data)
            root.Left = DeleteRec(root.Left, data);
        else if (data > root.Data)
            root.Right = DeleteRec(root.Right, data);
        else
        {
            // Case 1: No child or one child
            if (root.Left == null)
                return root.Right;
```

```
            else if (root.Right == null)
                return root.Left;

            // Case 2: Two children
            root.Data = MinValue(root.Right);
            root.Right = DeleteRec(root.Right, root.Data);
        }
        return root;
    }

    private int MinValue(BSTNode root)
    {
        int min = root.Data;
        while (root.Left != null)
        {
            min = root.Left.Data;
            root = root.Left;
        }
        return min;
    }
}

// Usage:
bst.Delete(10); // Deletes node with value 10
```

---

12. 4. Key Differences and Use Cases

Feature	Linear Search	Binary Search	Binary Search Tree
Input Requirement	Any array (sorted/unsorted)	Sorted array	Dynamic sorted structure
Time Complexity	O(n)	O(log n)	O(log n) avg, O(n) worst
Space Complexity	O(1)	O(1) (iterative)	O(n)
Use Case	Small/unsorted data	Static sorted data	Dynamic data with frequent insertions/deletions

13. 5. Summary

- **Linear Search**: Simple but slow for large datasets. Use for unsorted or small data.
- **Binary Search**: Extremely fast for sorted arrays. Ideal for static datasets.
- **BST**: Balances insertion, deletion, and search efficiently. Use for dynamic data (if kept balanced with AVL/Red-Black trees).