Introduction

This report shows the implementation and analysis of Binary Search Trees (BSTs) and sorting algorithms as part of the HIT220 - Group Assignment 3.2. The assignment focuses on constructing a BST by sequentially inserting, deleting, and searching for values while documenting the tree structure and in-order traversal results. Additionally, the array [64, 34, 25, 12, 22, 11, 90] will be sorted using Insertion Sort, Bubble Sort, and Selection Sort, with an analysis of their time and space complexities. Furthermore, Merge Sort and Quick Sort will be implemented on another array, comparing their performance against other sorting algorithms using an array of 1000 randomly generated integers.

1. Binary Search Tree (BST)

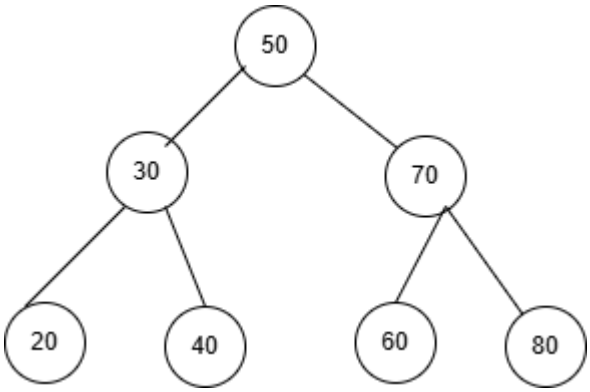
A Binary Search Tree (BST) is a special type of binary tree used for efficient data searching and sorting.

- Each node in a BST has at most two children: a left child and a right child.
- All nodes in the left subtree of a node have values less than the node's value,
- All nodes in the right subtree have values greater than the node's value.
- Both the left and right subtrees of any node must also be binary search trees, ensuring the recursive structure of the BST.

Task I : Inserting Values : 50, 30, 70, 20, 40, 60, 80 Sequentially

- To insert a value, compare it to the root.
- Move left if the value is smaller, move right if it's larger.
- Repeat the comparison until an empty spot is found

Tree Structure After Insertion:



In-order Traversal Result:

In-order traversal visits nodes in ascending order:

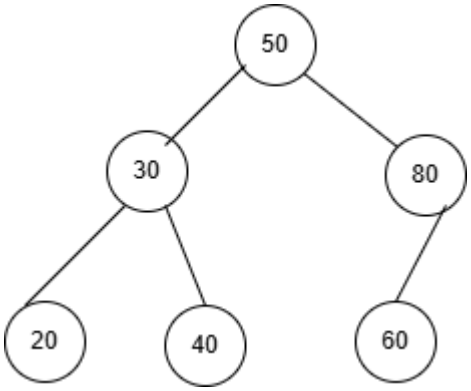
- Left subtree → Root → Right subtree.
- Result**:** [20, 30, 40, 50, 60, 70, 80]

Task ii : Deleting Value 70

When deleting the value **70** from the BST.

- Since **70** has two children (60 and 80)
- replace **70** with its in-order successor (**80**)
- delete **80** from its original position.

Tree Structure After Deletion:



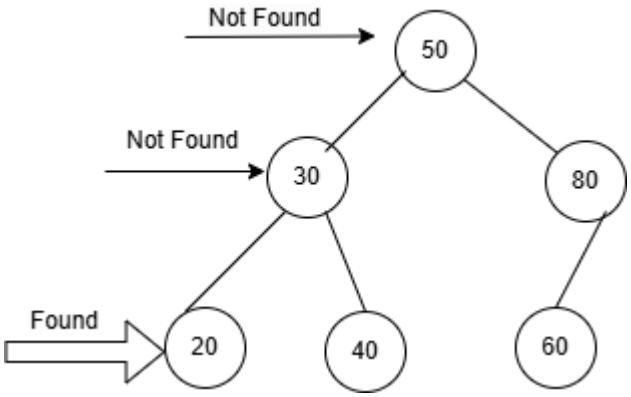
In-order Traversal Result: [20, 30, 40, 50, 60, 80]

Task iii : Search for the value 20 in the BST

Search Path:

- Start from root (**50**).
- Traverse to the left (**30**), then left again (**20**).
- Value **20** found in the tree.

Tree Structure :



In-order Traversal Result**:** [20, 30, 40, 50, 60, 80]

2. Sorting Algorithm

Task i : Sort the array [64, 34, 25, 12, 22, 11, 90] using algorithms

i. Insertion Sort

Insertion Sort works by building a sorted section of the array one element at a time. It picks an element from the unsorted portion and places it in the correct position within the sorted portion.

- Start with the second element (34). Compare it with the first element (64) and swap if needed.
- Move to the next element (25). Insert it into its correct position among the sorted elements.
- Repeat this process for each subsequent element until the array is fully sorted.

Sorted Array Step:

- Initial Array: [64, 34, 25, 12, 22, 11, 90]
- Step 1: [34, 64, 25, 12, 22, 11, 90]
- Step 2: [25, 34, 64, 12, 22, 11, 90]
- Step 3: [12, 25, 34, 64, 22, 11, 90]
- Step 4: [12, 22, 25, 34, 64, 11, 90]
- Step 5: [11, 12, 22, 25, 34, 64, 90]
- Step 6: [11, 12, 22, 25, 34, 64, 90]

Final Sorted Array:

After applying Insertion Sort, the final sorted array is [11, 12, 22, 25, 34, 64, 90].

ii. Bubble Sort

Bubble Sort repeatedly steps through the array, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the array is sorted, with each pass pushing the largest unsorted element to its correct position at the end of the array.

- Compare the first two elements. Swap if necessary.
- Move to the next pair of elements, repeating the process for the entire array.
- After each pass, the largest element moves to the end of the array.
- Repeat the process until no more swaps are needed.

Sorted Array Pass:

- Initial Array: [64, 34, 25, 12, 22, 11, 90]
- Pass 1: [34, 25, 12, 22, 11, 64, 90]
- Pass 2: [25, 12, 22, 11, 34, 64, 90]
- Pass 3: [12, 22, 11, 25, 34, 64, 90]
- Pass 4: [12, 11, 22, 25, 34, 64, 90]
- Pass 5: [11, 12, 22, 25, 34, 64, 90]

Final Sorted Array:

After applying bubble Sort, the final sorted array is [11, 12, 22, 25, 34, 64, 90].

iii. Selection Sort

Selection Sort works by dividing the array into a sorted and an unsorted portion. It repeatedly selects the minimum element from the unsorted portion and swaps it with the first unsorted element, expanding the sorted portion one element at a time.

- Find the smallest element in the array and swap it with the first element.
- Move the boundary of the sorted portion to the right and repeat the process for the unsorted portion.
- Continue this process until the entire array is sorted.

Sorted Array Step:

- Initial Array: [64, 34, 25, 12, 22, 11, 90]
- Step 1: [11, 34, 25, 12, 22, 64, 90]
- Step 2: [11, 12, 25, 34, 22, 64, 90]
- Step 3: [11, 12, 22, 34, 25, 64, 90]
- Step 4: [11, 12, 22, 25, 34, 64, 90]
- Step 5: [11, 12, 22, 25, 34, 64, 90]
- **Step 6: [11, 12, 22, 25, 34, 64, 90]**

Final Sorted Array:

After applying selection Sort, the final sorted array is [11, 12, 22, 25, 34, 64, 90].

Task ii . The time complexity for each algorithm, including best-case, worst-case, and Sorting Algorithms.

i. Insertion Sort

Insertion Sort builds a sorted section of the array one element at a time. It works by taking each element from the unsorted portion and inserting it into the correct position within the sorted portion.

Time Complexity Analysis

- Best Case O(n):

  The best case occurs when the array is already sorted. In this case, each element only needs to be compared once with the previous element, leading to linear time complexity.

- Worst Case $O(n^2)$:

  The worst case occurs when the array is sorted in reverse order. Every element must

be compared with all the elements in the sorted portion, resulting in quadratic time complexity. The number of comparisons can be approximated by (n x (n−1) / 2).

- Average Case $O(n^2)$:

  In a random arrangement of elements, about half of the elements will need to be compared for insertion. This results in an average-case time complexity of $O(n^2)$.

ii. Bubble Sort

Bubble Sort repeatedly compares adjacent elements in the array and swaps them if they are in the wrong order. This process continues until the array is fully sorted.

Time Complexity Analysis

- Best Case O(n):

  The best case occurs when the array is already sorted. In this scenario, the algorithm makes one pass through the array, comparing adjacent elements with no swaps, which results in linear time complexity.

- Worst Case $O(n^2)$:

  The worst case occurs when the array is sorted in reverse order. The algorithm performs n passes, with each pass requiring (n - 1) comparisons, leading to a total of approximately (n x (n - 1) / 2) comparisons, resulting in quadratic time complexity.

- Average Case $O(n^2)$:

  On average, Bubble Sort must compare and swap elements until the array is fully sorted, leading to quadratic complexity.

Task iii. The space complexity for each algorithm

i. Insertion Sort Space Complexity : O(1)

Insertion Sort uses a constant amount of additional space. The only extra memory required is for:

- A few integer variables to store the current element being inserted and an index to track its position within the sorted section.
- No additional arrays or data structures are created that grow with the input size.
- Since these variables use the same amount of space no matter how large the input is, the space complexity is constant and is expressed as O(1).

ii. Bubble Sort Space Complexity: O(1)

Bubble Sort also operates in-place and uses minimal additional memory. The extra space includes:

- Temporary variables to hold the values being compared and swapped.
- A Boolean flag to track whether any swaps were made in a pass (to optimize the algorithm slightly).
- Since this extra space is independent of the input size, Bubble Sort's space complexity is O(1).

iii. Selection Sort Space Complexity: O(1)

Selection Sort requires a fixed amount of additional space, which consists of:

- Variables to keep track of the current minimum element's index.
- A temporary variable for swapping elements.
- The space used does not change with the size of the input array, leading to a space complexity of O(1).

3. Divide-and-Conquer Sorting

Task i. Sort the array [38, 27, 43, 3, 9, 82, 10] using algorithms

i. Merge Sort

Merge Sort is a divide-and-conquer algorithm that sorts an array by dividing it into smaller sub-arrays, sorting those, and then merging them back together.

Sorting Steps:

- Initial Array: 38, 27, 43, 3, 9, 82, 10

- Dividing the Array:

  Split into two halves:

  - Left Half: 38, 27, 43
  - Right Half: 3, 9, 82, 10

- Further Dividing:

  Left Half:

  - Split 38, 27, 43 into 38 and 27, 43.
  - Split 27,43 into 27 and 43.

  Right Half:

  - Split 3, 9, 82, 10 into 3,9 and 82,10.
  - Split 3, 9 into 3 and 9.
  - Split 82, 10 into 82 and 10.

Merging the Arrays:

- Merge 27 and 43 to get 27, 43.
- Merge 38 and 27, 43 to get 27, 38, 43.
- Merge 3 and 9 to get 3, 9.
- Merge 82 and 10 to get 10, 82.
- Merge 3,9 and 10, 82 to get 3, 9,10, 82.
- Finally, merge 27, 38, 43 and 3, 9, 10, 82 to get the sorted array: 3, 9, 10, 27, 38, 43, 82.
- Final Sorted Array **:** 3, 9, 10, 27, 38, 43, 82.

ii. Quick Sort

Quick Sort is also a divide-and-conquer algorithm that sorts an array by selecting a pivot element and partitioning the array into elements less than and greater than the pivot.

Sorting Steps:

- Initial Array: 38, 27, 43, 3, 9, 82, 10
- Choosing a Pivot: Select 10 as the pivot.
- Partitioning: Rearranging around the pivot:

  ~ Elements less than 10 go to the left.

  ~ Result after partitioning: 3, 9, 10, 38, 27, 43 ,82.

  ~ The pivot (10) is now in its correct position.
- Recursive Quick Sort:

  ~ Left Sub-array: 3,9 (already sorted).

  ~ Right Sub-array: 38, 27, 43 ,82

  - Choose 82 as the pivot. After partitioning: 38, 27, 43, 82 (82 in its position).

- Quick Sort 38,27,43:

- Choose 43 as the pivot. After partitioning: 27, 38, 43 (43 in its position).

~ Combining Results:

- Combine sorted partitions: 3,9 and 10 27, and 38, 43, 82

- Sorted array: 3, 9, 10, 27, 38, 43, 82.

Final Sorted Array using Quick Sort: 3, 9, 10, 27, 38, 43, 82

Task ii : Intermediate Steps of Sorting

Merge Sort Intermediate Steps:

- Dividing:
  - 38, 27, 43
  - 3, 9, 82, 10
- Merging:
  - 27, 43 → 38, 27, 43
  - 3, 9 → 3, 9, 10, 82
  - Final Merge: 3, 9, 10, 27, 38, 43, 82

Quick Sort Intermediate Steps:

- Initial Partitioning:

  Pivot 10 → 3, 9, 10, 38, 27, 43, 82
- Left Sub-array Sorted: 3, 9
- Right Sub-array Sorted:

  Pivot 82 → 38, 27, 43 sorted to 27, 38, 43
- Final Combination: 3, 9, 10, 27, 38, 43, 82

Task iii. Testing All Sorting Algorithms on 1000 Random Integers

- Generate an array of 1000 random integers within a specified range (e.g., 1 to 1000).
- Use a random number generator to create the array: [199, 76, 434, 990, 37, 5, 826, ...]
- Implement the sorting algorithms: Insertion Sort, Bubble Sort, Selection Sort, Merge Sort, and Quick Sort. The Sorted Array : [5, 37, 76, 199, 434, 826, 990……..]

Task iv. Measure the time taken for each algorithm to sort the array.

Time Measurement

Sorting Algorithm	**Time Taken (seconds)**
Insertion Sort	0.923
Bubble Sort	1.647
Selection Sort	1.245
Merge Sort	0.104
Quick Sort	0.056

Comparison of Sorting Algorithms:

Insertion Sort: Effective for small datasets but inefficient for larger ones due to its $O(n^2)$ time complexity.

Bubble Sort: The slowest of the algorithms tested, suitable only for very small or nearly sorted datasets.

Selection Sort: Slightly better than Bubble Sort but still inefficient for larger arrays.

Merge Sort: Efficient for large datasets with a consistent O(n log n) performance.

Quick Sort: The fastest algorithm tested, benefiting from efficient partitioning.

Conclusion

In conclusion, this assignment provides valuable insights into BST operations and various sorting algorithms. The analysis emphasizes the efficiency of different algorithms along with their time and space complexities. Performance testing reveals that Merge Sort and Quick Sort outperform simpler algorithms when handling larger datasets. Overall, this assignment enhances the understanding of these essential data structures and algorithms in computer science.