Department of Information Science and engineering

Course: Design and Analysis of Algorithms (18CS43)

Report On

INSERTION SORT VS TREE SORT

Submitted by

Sahil Sharma (1RV19IS046)

Ayush Gautam (1RV19IS011)

Course in-charge:

Dr. Rajashekara Murthy S

Associate Professor,

Dept. of Information Science and Engineering

RV College of Engineering®

2020-2021

INSERTION SORT

1. Introduction:

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- Simple implementation: Jon Bentley shows a three-line C version, and a five-line optimized version
- Efficient for (quite) small data sets, much like other quadratic sorting algorithms.
- More efficient in practice than most other simple quadratic (i.e., O(n^2)) algorithms such as selection sort or bubble sort.
- Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is O(kn) when each element in the input is no more than k places away from its sorted position.
- Stable; i.e., does not change the relative order of elements with equal keys.
- In-place; i.e., only requires a constant amount O(1) of additional memory space.

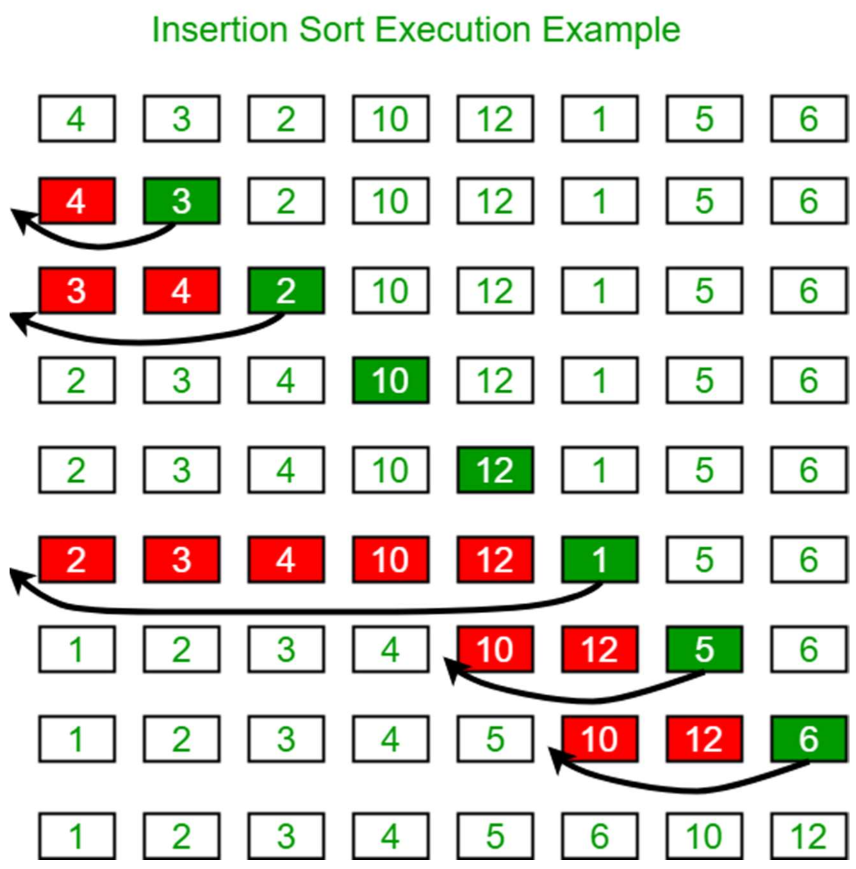
2. Algorithm, Time & Space complexity

Insertion sort iterates, consuming one input element each repetition, and grows a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

To sort an array of size n in ascending order:

1. Iterate from arr[1] to arr[n] over the array.
2. Compare the current element (key) to its predecessor.
3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.



Time Complexity: O(n^2)

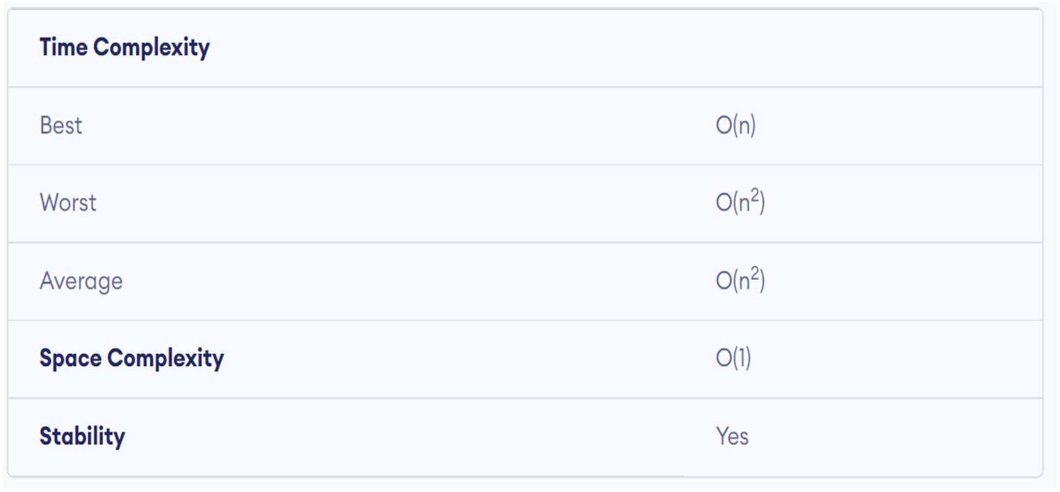
Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.
Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.



Time Complexity	
Best	O(n)
Worst	O(n^2)
Average	O(n^2)
Space Complexity	O(1)
Stability	Yes

ALGORITHM

```
i ← 1
while i < length(A)

             x ← A[i]

             j ← i - 1

             while j >= 0 and A[j] > x
```

```
                A[j+1] ← A[j]

                j ← j - 1

          end while

          A[j+1] ← x

          i ← i + 1

end while
```

- The new inner loop shifts elements to the right to clear a spot for x = A[i]

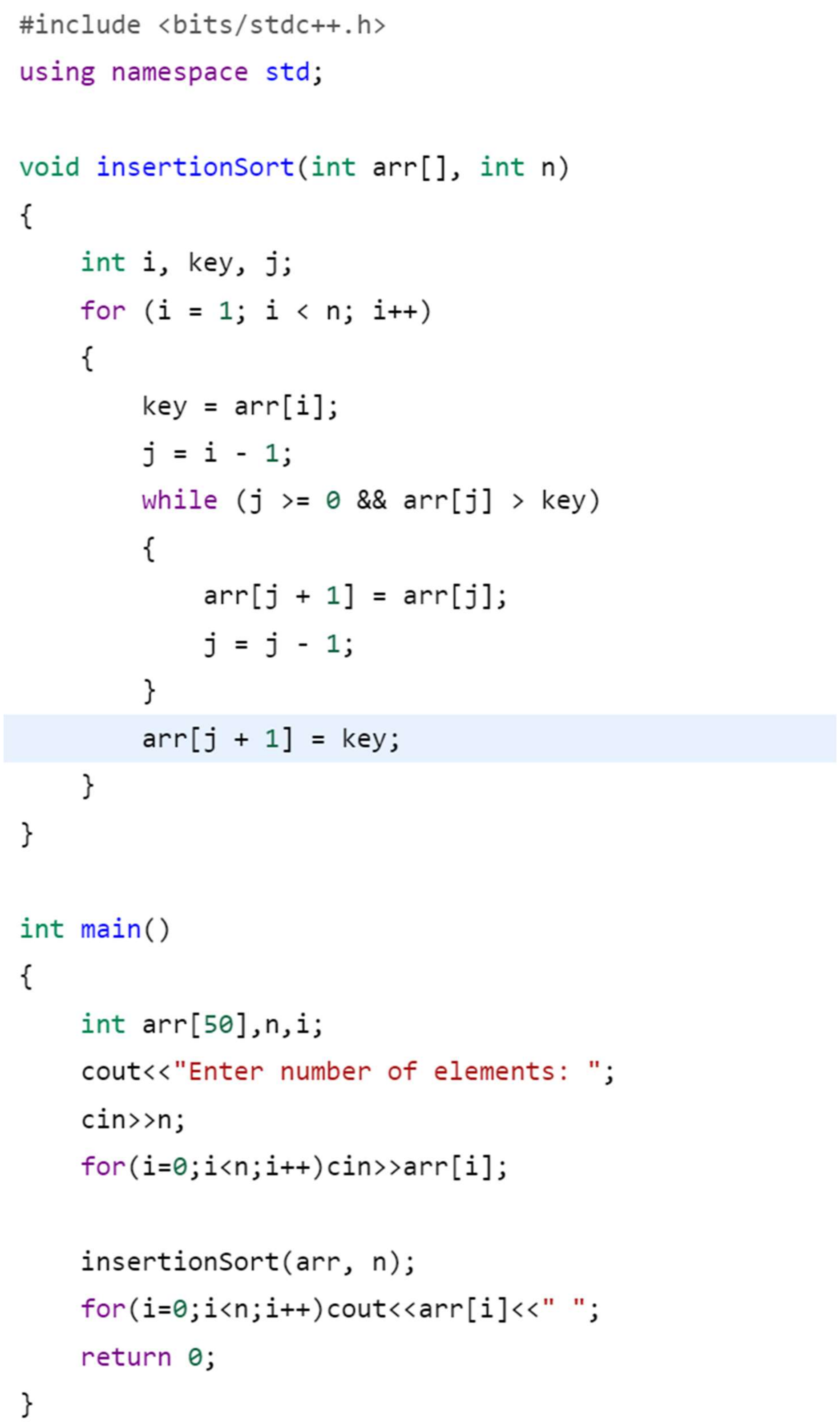
ALGORITHM RECURSIVE

```
function insertionSortR(array A, int n)

   if n > 0

         insertionSortR(A, n-1)

         x ← A[n]

         j ← n-1

         while j >= 0 and A[j] > x

               A[j+1] ← A[j]

               j ← j-1

         end while

         A[j+1] ← x

         end if

end function
```

It does not make the code any shorter, it also doesn't reduce the execution time, but it increases the additional memory consumption from

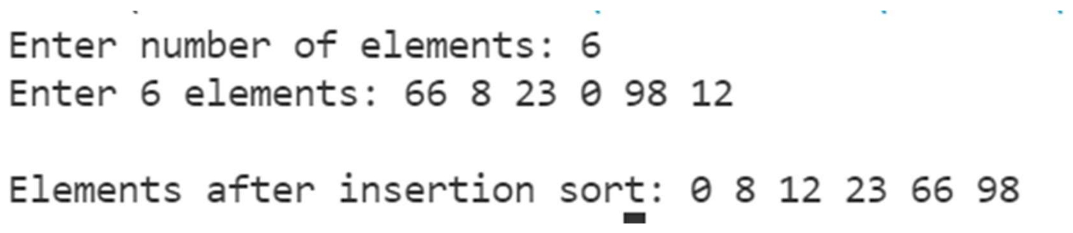
O(1) to O(N) (at the deepest level of recursion the stack contains N references to the A array, each with accompanying value of variable n from N down to 1).

3. Code and Output

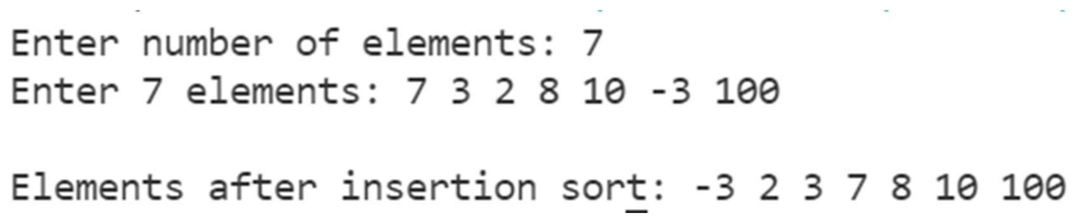


```
#include <bits/stdc++.h>
using namespace std;

void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

int main()
{
    int arr[50],n,i;
    cout<<"Enter number of elements: ";
    cin>>n;
    for(i=0;i<n;i++)cin>>arr[i];

    insertionSort(arr, n);
    for(i=0;i<n;i++)cout<<arr[i]<<" ";
    return 0;
}
```

OUTPUT



```
Enter number of elements: 6
Enter 6 elements: 66 8 23 0 98 12

Elements after insertion sort: 0 8 12 23 66 98
```



```
Enter number of elements: 7
Enter 7 elements: 7 3 2 8 10 -3 100

Elements after insertion sort: -3 2 3 7 8 10 100
```

4. Applications

An application of Insertion sort happens everyday in real life while playing cards.

To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence.

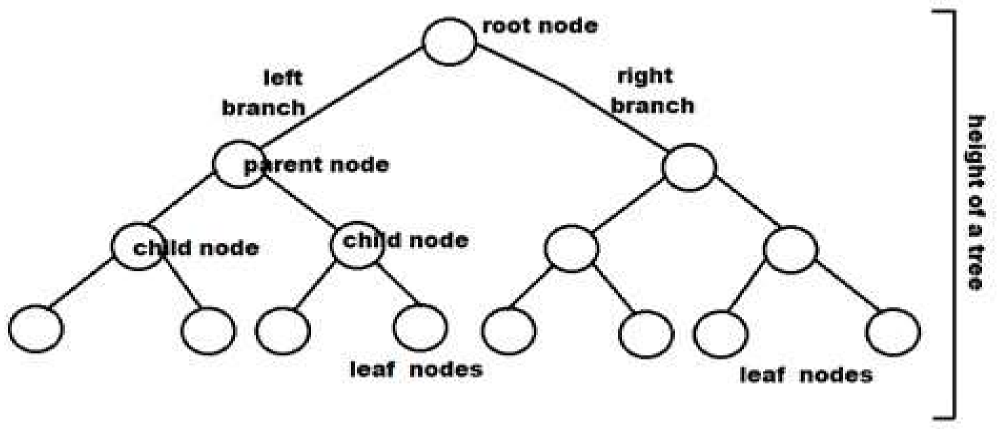
TREE SORT

1. Introduction

Tree sort is a sort algorithm that builds a binary search tree from the elements to be sorted, and then traverses the tree (**in-order**) so that the elements come out in sorted order. Its typical use is sorting elements online: after each insertion, the set of elements seen so far is available in sorted order.

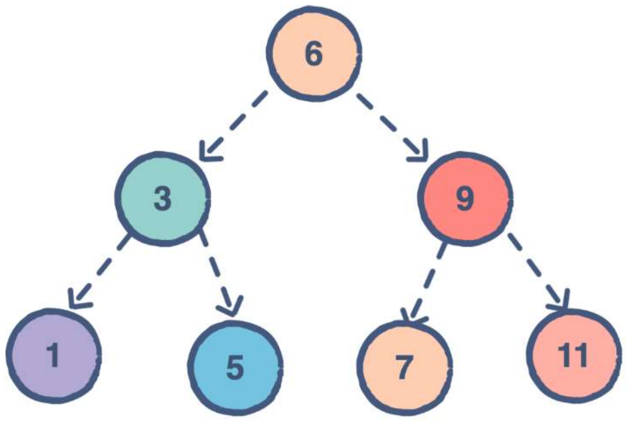
Tree sort can be used as a one-time sort, but it is equivalent to quicksort as both recursively partition the elements based on a pivot, and since quicksort is in-place and has lower overhead, it has few advantages over quicksort. It has better worst case complexity when a self-balancing tree is used, but even more overhead.

- **Binary Tree:** A binary tree is a tree data structure in which each node has at most two child nodes. The child nodes are called the left child and right child.



- **Binary Search Tree:** Binary Search Tree, Binary Sorted Tree or BST, is a binary tree satisfies the following condition:
    - For each node, the left sub-tree contains only nodes with values less than that node’s value.

- For each node, the right sub-tree contains only nodes with values greater than that node's value.
- Left and right sub-trees are also BSTs.



An example of a binary search tree

- BST keeps the keys in sorted order so that searching for a key or finding a place to insert a key is done in **O(n)** in the worst case, and **O(log n)** in the average case, where **n** is a tree size.

2. Algorithm, Time & Space complexity

Steps involved are:

- Takes the elements input in an array.
- Creates a binary search tree by inserting data items from the array into the tree.
- Performs in-order traversal on the tree to get the elements in sorted order.

Pseudocode:

For the second step of the algorithm, the function keeps looking for the right position for the input element recursively by comparing it to the root of each subtree. The following pseudocode, Insert(Node node, Key value) shows the details of that function:

```
Algorithm 1: insert(Node node, Key value): Inserts a new node in the BST
  Data: node: The input node object
        value: The value of node key
  Result: Returns the inserted node object
  if node = null then
      return Node(value);
  end
  else if value <= node.value then
      node.left ← insert(node.left, value);
  end
  else
      node.right ← insert(node.right, value);
  end
  return node;
```

For the third step, the function goes through all the tree recursively to get the left side of each subtree printed before the right side. This guarantees to print the tree in order. The following pseudocode, traverseInOrder(Node node) shows the details of that function:

```
Algorithm 2: traverseInOrder(Node node): Traverses and prints the BST in order
  Data: node: The input node object
  Result: Prints the final BST in ascending order
  if node != null then
      traverseInOrder(node.left);
      print(node.value);
      traverseInOrder(node.right);
  end
```

Time Complexity:

Given the following two points:

- Inserting a new node in a BST takes **O(log n)** in the average case and **O(n)** in the worst case.
- The input array to tree sort algorithm is of size n;

So the time complexity for constructing the BST is the time of inserting **n** nodes, which results in total time complexity of **O(n log n)** in the average case, and **O(n^2)** in the worst case. We will name it **T(BST Insertion).**
For the traversal time complexity, it takes steps equal to the tree size to read and print all the nodes, so it

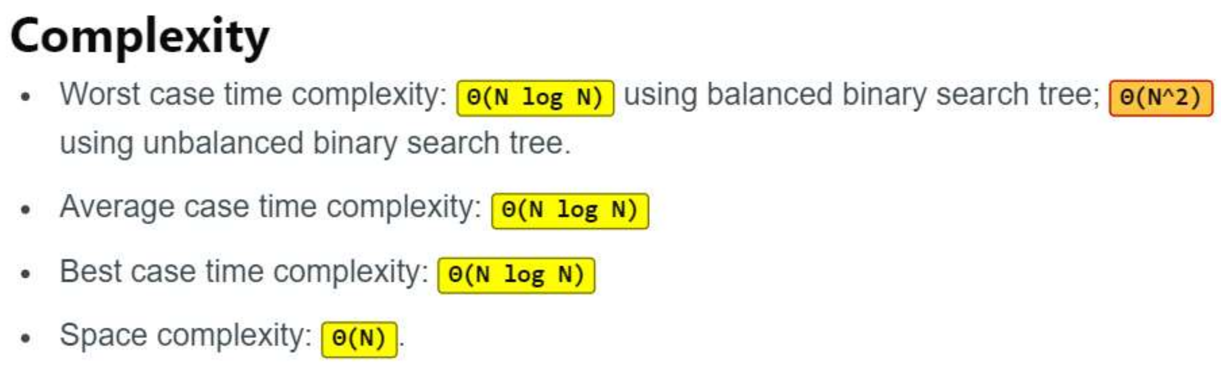
takes **n** steps. So that the time complexity of traversing and printing the BST in order is **O(n)** and we will name it **T(BST Traversal).**
Finally, the worst-case time complexity of sorting a binary tree using the steps of the tree sort algorithm is as follows:

$$T(SortingBinaryTree) = T(BST\ Insertion) + T(BST\ Traversal) = O(n^2) + O(n) = O(n^2)$$

The calculations of the worst-case assume an unbalanced BST. To maintain the average case, **a balanced BST is needed.**

➢ To build a balanced binary search tree, we need a different data structure to maintain the balancing, and we call it a self-balancing (or height balancing) binary search tree. Most known implementations of the self-balancing BST are AVL Tree, B-Tree, Red-Black Tree, and Splay Tree

<u>Space Complexity:</u>
As we need to create **n** nodes for **n** elements during all the steps, so the space complexity is **O(n).**



Complexity

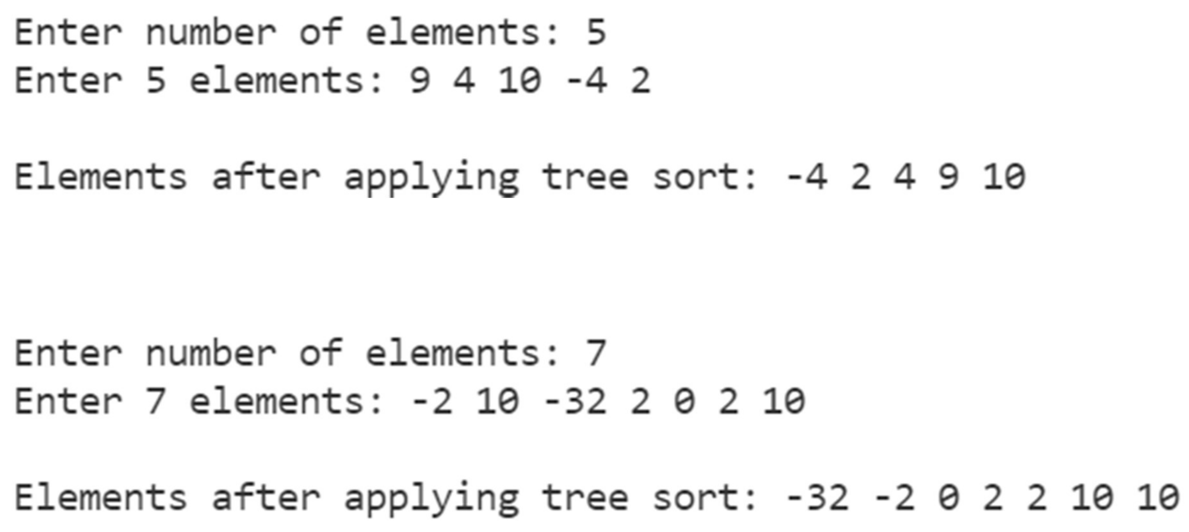
- Worst case time complexity: `Θ(N log N)` using balanced binary search tree; `Θ(N^2)` using unbalanced binary search tree.
- Average case time complexity: `Θ(N log N)`
- Best case time complexity: `Θ(N log N)`
- Space complexity: `Θ(N)`.

3. Code and Output

```cpp
#include<bits/stdc++.h>
using namespace std;
struct Node
{
    int key;
    struct Node *left, *right;
};

struct Node *newNode(int item)
{
    struct Node *temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

void storeSorted(Node *root, int arr[], int &i)
{
    if (root != NULL)
    {
        storeSorted(root->left, arr, i);
        arr[i++] = root->key;
        storeSorted(root->right, arr, i);
    }
}


Node* insert(Node* node, int key)
{
    if (node == NULL) return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    return node;
}
```

```
void treeSort(int arr[], int n)
{
    struct Node *root = NULL;
    root = insert(root, arr[0]);
    for (int i=1; i<n; i++)
        root = insert(root, arr[i]);
    int i = 0;
    storeSorted(root, arr, i);
}

int main()
{
    int arr[50],n,i;
    cout<<"Enter number of elements: ";
    cin>>n;
    cout<<"Enter "<<n<<" elements: ";
    for(i=0;i<n;i++)cin>>arr[i];
    treeSort(arr, n);
    cout<<"\nElements after applying tree sort: ";
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

OUTPUT



```
Enter number of elements: 5
Enter 5 elements: 9 4 10 -4 2

Elements after applying tree sort: -4 2 4 9 10



Enter number of elements: 7
Enter 7 elements: -2 10 -32 2 0 2 10

Elements after applying tree sort: -32 -2 0 2 2 10 10
```

4. Applications

- Its typical use is sorting elements online: after each insertion, the set of elements seen so far is available in sorted order.
- When using a splay tree as the binary search tree, the resulting algorithm (called splay sort) has the additional property that it is an adaptive sort, meaning that its running time is faster than O(n log n) for inputs that are nearly sorted.

TREE SORT VS INSERTION SORT

TREE SORT	INSERTION SORT
Best case TC: O(n*log n)	**Best case TC: O(n)**
Worst case TC: O(n*log n)(balanced)	**Worst case TC: O(n^2)**
Average case TC: O(n*log n)	**Average case TC: O(n^2)**
Space Complexity: O(n)	**Space Complexity: O(1)(i.e. constant)**
Stable: YES	**Stable: YES**
Class: Comparison Sort	**Class: Comparison Sort**
Method: Insertion	**Method: Insertion**
Sorting In Place: No	**Sorting In Place: Yes**
Online: Yes	**Online: Yes**
Data Structures: Array, Linked List	**Data Structures: Array**
Uses: Input could be small or large	**Uses: Useful when input is small**

References

- https://www.geeksforgeeks.org/tree-sort/
- https://www.geeksforgeeks.org/insertion-sort/
- https://en.wikipedia.org/wiki/Insertion_sort
- https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm
- https://www.programiz.com/dsa/insertion-sort
- https://en.wikipedia.org/wiki/Tree_sort#:~:text=A%20tree%20sort%20is%20a,is%20available%20in%20sorted%20order.
- https://iq.opengenus.org/tree-sort/https://www.baeldung.com/cs/sorting-binary-tree