# Kongu Engineering College KONGU ENGINEERING COLLEGE Kongu Engineering College (Autonomous)

(Autonomous)

Perundurai,Erode – 638060

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**Insertion Sort on Singly Linked List**

**A MICRO PROJECT REPORT**

**FOR**

**DESIGN AND ANALYSIS OF ALGORITHMS (22ITT31)**

**SUBMITTED BY**

**GOWRI D (23ITR048)**

** KONGU ENGINEERING COLLEGE**  (Autonomous)

Perundurai,Erode – 638060

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**Insertion Sort on Singly Linked List**

**A MICRO PROJECT REPORT**

**FOR**

**DESIGN AND ANALYSIS OF ALGORITHMS(22ITT31)**

**SUBMITTED BY**

**GOWRI D (23ITR048)**

# Kongu Engineering College KONGU ENGINEERING COLLEGE Kongu Engineering College (Autonomous)

# (Autonomous)

Perundurai,Erode – 638060

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**BONAFIDE CERTIFICATE**

|  |  |
| --- | --- |
| Name | : GOWRI D |
| Course Code | : 23ITR048 |
| Course Name | : DESIGN AND ANALYSIS OF ALGORITHMS |
| Semester | : IV |

Certified that this is a bonafied record of work for application project done by the above student for 22ITT31-DESIGN AND ANALYSIS OF ALGORITHMS during the academic year 2024-2025.

Submitted for the Viva Voice Examination held on \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Faculty Incharge Head of the Department

## ABSTRACT

## The Insertion Sort on Linked List project is a practical implementation and visualization of the classic insertion sort algorithm applied to singly linked lists. This project highlights how the use of linked lists—rather than arrays—can optimize insertions during sorting by avoiding element shifting. It offers a side-by-side analysis of both linked list and array-based sorting to explore efficiency differences and memory handling. The project provides an interactive web interface allowing users to input data, visualize the list, and observe the sorting process in real time. By combining theoretical algorithm concepts with a hands-on visual demonstration, this project enhances understanding of data structure-specific behavior, time complexity trade-offs, and algorithmic thinking, making it a valuable educational resource for learners studying data structures and sorting algorithms.

## TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| **CHAPTER No** | **TITLE** | **PAGE No** |
|  | **ABSTRACT** | **ix** |
| **1.** | **INTRODUCTION** | 6 |
|  | 1.1 PURPOSE | 7 |
|  | 1.2 OBJECTIVE | 7 |
|  | 1.3 METHODOLOGY OVERVIEW | 8 |
| **2.** | **PROBLEM STATEMENT** | 9 |
| **3.** | **METHODOLGY** | 10 |
|  | 3.1 Input & Initialization | 10 |
|  | 3.2Sorting Logic & Execution | 10 |
|  | 3.3 Result Processing | 10 |
|  | 3.4 Visualization & Output | 10 |
| **4.** | **IMPLEMENTATION** | 11 |
|  | 4.1 Code Structure Overview | 11 |
|  | 4.2 Insertion Sort (Linked List) | 14 |
|  | 4.3 Comparison with Arrays | 16 |
|  | 4.4 Web-based Visualization | 18 |
| **5.** | **RESULTS** | 23 |

## INTRODUCTION

The **Insertion Sort algorithm** is a foundational technique in sorting, where elements are inserted into their correct positions one at a time to build a sorted sequence. While commonly applied to arrays, this project explores the application of insertion sort on a **singly linked list**, a dynamic data structure that offers efficient insertions without the overhead of shifting elements. The aim is to demonstrate how linked lists can improve flexibility and memory usage in sorting scenarios, especially for datasets that change frequently.

This project introduces an interactive visualization tool that allows users to input values, build a linked list dynamically, and observe the insertion sort process in real time. Developed using **HTML5, CSS3, and JavaScript**, the interface enables users to sort values and visually track how each element is repositioned. For comparative analysis, a **Java-based implementation** is also included to benchmark sorting performance on both arrays and linked lists. The project not only highlights the core logic of insertion sort but also provides insights into the performance trade-offs between static and dynamic data structures—bridging theory and application for enhanced algorithm comprehension.

* 1. **PURPOSE**

The main purpose of this project is:

* To illustrate how data structure choice affects sorting algorithm efficiency.
* To make the concept of insertion sort more intuitive by visualizing it in a pointer-based linked list environment.
* To compare sorting behavior and performance across linked lists and arrays using real-time input and execution.
* To provide an engaging and interactive platform for students and learners to deepen their understanding of sorting algorithms and linked list operations.

## OBJECTIVE

The key objectives of this microproject are:

**Core Objectives**

 **Educational Enhancement**

Provide a visual and interactive interface for understanding the behavior of insertion sort on different data structures.

 **Algorithm Comparison**

Develop and demonstrate insertion sort on a singly linked list in Java and JavaScript.

 **Performace Insight**   
Benchmark and compare the time complexity of insertion sort when applied to arrays vs. linked lists.

### ****1.3 METHODOLOGY OVERVIEW****

1. **User Input**  
   Users provide a sequence of numbers either through a graphical input box or via code initialization. These elements are added sequentially to a singly linked list.
2. **Sorting Logic Execution**  
   The list is processed by the insertion sort function, which traverses through the nodes, inserting each into the correct sorted position in a new list. A helper function manages the ordered insertion.
3. **Performance Comparison**  
   The same dataset is used for both array and linked list implementations. The execution time of each method is measured using Java’s System.nanoTime() and displayed for comparison.
4. **Visualization & Output**  
   On the web interface, the user can view:
   * The unsorted list.
   * The step-by-step sorting process.
   * The final sorted list.
   * Execution time in milliseconds.

## 2. PROBLEM STATEMENT

Sorting is a fundamental operation in computer science, essential for optimizing data processing, searching, and organization. Among various sorting techniques, **insertion sort** is widely recognized for its simplicity and effectiveness on small or nearly sorted datasets. However, its conventional implementation on arrays can be inefficient due to the need for shifting elements during insertion.

This project focuses on applying insertion sort to a **singly linked list**, where insertions are more efficient as they rely on pointer manipulation rather than index shifting. The linked list structure allows dynamic memory allocation, making it better suited for scenarios involving frequent insertions or deletions.

The primary goal of this project is to **implement, visualize, and analyze** the behavior of insertion sort when applied to linked lists. To provide deeper insights, the project also includes a **comparative performance analysis** between linked list-based and array-based insertion sort implementations. Through a real-time, interactive web interface and Java-based benchmarking, users can input data, observe sorting progress, and evaluate execution efficiency. The project emphasizes how **data structure selection impacts algorithm performance**, encouraging learners to think critically about algorithm design, time complexity, and structural trade-offs.

**3.0 Insertion Sort on Linked List Methodology**

**3.1 Input & Initialization**

Accept a sequence of integer inputs from the user.  
Allow users to either:

* Manually enter custom values into the linked list.
* Automatically generate large datasets for performance testing (e.g., 10 to 10,000+ elements).

Initialize the linked list by inserting each value at the end using pointer-based node creation.  
Also, create a copy of the same dataset in an array for comparative sorting.  
Set up performance timers using Java’s System.nanoTime() to track sorting times for both structures.

Prepare the visualization interface using HTML, CSS, and JavaScript to display the original and sorted linked list states.

**3.2 Sorting Logic & Execution**

For both the array and linked list:

* Apply the **Insertion Sort** algorithm using appropriate techniques:
  + **Linked List:** Traverse the list, and for each node, insert it into the correct position in a new sorted list using pointer manipulation.
  + **Array:** Use index-based comparisons and shifting for in-place insertion.
* Track and log the execution time for both implementations.
* Ensure the logic handles edge cases like duplicate values, single-element lists, and already sorted data.

**3.3 Result Processing**

After execution:

* Display the sorted result for both the array and the linked list.
* Log the time taken (in milliseconds) for each structure.
* Identify and store:
  + Time complexity behavior based on input size.
  + Data structure used (Array or Linked List).
  + Number of comparisons and pointer movements (conceptually).

If the input list is empty or contains non-integer values, notify the user with an appropriate message.

**3.4 Visualization & Output**

Display the following through the browser-based visualization panel:

* Step-by-step formation of the sorted linked list.
* Color-coded node highlights showing active insertion points.
* Final sorted list displayed in the format:  
  1 -> 2 -> 3 -> 5 -> 8 -> None
* Execution time comparison for:
  + Linked List Insertion Sort
  + Array Insertion Sort
* Provide users with options to reset, regenerate, or input a new list for further experimentation.

**IMPLEMENTATION :**

**4.1 Input & Initialization**

<!-- HTML -->

<div class="input-group">

<label>Input Type:</label>

<div class="toggle-switch" role="radiogroup" aria-label="Input Type Selection">

<input type="radio" name="inputType" id="randomList" value="random" checked aria-label="Random Linked List">

<label for="randomList">Random Linked List</label>

<input type="radio" name="inputType" id="customList" value="custom" aria-label="Custom Linked List">

<label for="customList">Custom Linked List</label>

</div>

</div>

<div id="randomListControls">

<label for="listSize">List Size:</label>

<input type="range" id="listSize" min="1" max="20" value="10">

<output>10</output>

</div>

<div id="customListControls" style="display:none;">

<label for="customListInput">Enter comma-separated values:</label>

<input type="text" id="customListInput" placeholder="e.g. 5, 12, 3, 9">

</div>

<div id="result"></div>

<div id="executionTime"></div>

// JavaScript

// Node class for singly linked list

class ListNode {

constructor(value) {

this.value = value;

this.next = null;

}

}

// Generate random singly linked list of given size

function generateRandomList(size) {

if (size < 1) return null;

let head = new ListNode(Math.floor(Math.random() \* size \* 10));

let current = head;

for (let i = 1; i < size; i++) {

current.next = new ListNode(Math.floor(Math.random() \* size \* 10));

current = current.next;

}

return head;

}

// Convert array to singly linked list

function arrayToLinkedList(arr) {

if (!arr.length) return null;

let head = new ListNode(arr[0]);

let current = head;

for (let i = 1; i < arr.length; i++) {

current.next = new ListNode(arr[i]);

current = current.next;

}

return head;

}

// Get input linked list based on user selection

function getInputList() {

const inputType = document.querySelector('input[name="inputType"]:checked').value;

if (inputType === 'random') {

const size = parseInt(document.getElementById('listSize').value);

return generateRandomList(size);

} else {

const input = document.getElementById('customListInput').value;

const arr = input.split(',')

.map(num => parseInt(num.trim()))

.filter(num => !isNaN(num));

return arrayToLinkedList(arr);

}

}

// Update slider output display

function initializeEventListeners() {

document.querySelectorAll('input[name="inputType"]').forEach(radio => {

radio.addEventListener('change', function() {

const randomControls = document.getElementById('randomListControls');

const customControls = document.getElementById('customListControls');

randomControls.style.display = this.value === 'random' ? 'block' : 'none';

customControls.style.display = this.value === 'custom' ? 'block' : 'none';

});

});

const sizeSlider = document.getElementById('listSize');

sizeSlider.addEventListener('input', function() {

this.nextElementSibling.value = this.value;

});

}

// Clear previous results and initialize visualization placeholders

function initializeVisualization() {

document.getElementById('result').innerHTML = '';

document.getElementById('executionTime').innerHTML = '';

// Any additional initialization for your visualization library/chart can go here

}

// Wait for DOM and initialize event listeners + visualization

document.addEventListener('DOMContentLoaded', () => {

initializeEventListeners();

initializeVisualization();

});

**4.2 Divide & Compare**

function displayLinkedListResults(head, method, executionTime) {

// Convert linked list to array for display

function listToArray(node) {

const arr = [];

while (node) {

arr.push(node.value);

node = node.next;

}

return arr;

}

const sortedArr = listToArray(head);

document.getElementById('result').innerHTML = `

<div class="result-content">

<h3>Insertion Sort Result</h3>

<p>Method: ${method}</p>

<p>Sorted List: ${sortedArr.length > 20 ? sortedArr.slice(0, 20).join(', ') + '...' : sortedArr.join(', ')}</p>

<p>List Size: ${sortedArr.length}</p>

</div>

`;

document.getElementById('executionTime').innerHTML = `

<div class="time-content">

<h3>Performance</h3>

<p>Execution time: ${executionTime.toFixed(4)} ms</p>

</div>

`;

}

**4.4 Visualization & Output**

function displayLinkedListResults(head, method, executionTime) {

// Convert linked list to array for display

function listToArray(node) {

const arr = [];

while (node) {

arr.push(node.value);

node = node.next;

}

return arr;

}

const sortedArr = listToArray(head);

document.getElementById('result').innerHTML = `

<div class="result-content">

<h3>Insertion Sort Result</h3>

<p>Method: ${method}</p>

<p>Sorted List: ${sortedArr.length > 20 ? sortedArr.slice(0, 20).join(', ') + '...' : sortedArr.join(', ')}</p>

<p>List Size: ${sortedArr.length}</p>

</div>

`;

document.getElementById('executionTime').innerHTML = `

<div class="time-content">

<h3>Performance</h3>

<p>Execution time: ${executionTime.toFixed(4)} ms</p>

</div>

`;

}

**Insertion Sort Algorithms on Singly Linked List**

// Iterative Insertion Sort - O(n²)

function insertionSortIterative(head) {

if (!head || !head.next) return head;

let sorted = null;

let current = head;

while (current) {

const next = current.next;

sorted = sortedInsert(sorted, current);

current = next;

}

return sorted;

}

// Recursive Insertion Sort - O(n²)

function insertionSortRecursive(head) {

if (!head || !head.next) return head;

let sorted = insertionSortRecursive(head.next);

head.next = null;

return sortedInsert(sorted, head);

}

// Helper function to insert a node into sorted list

function sortedInsert(sortedHead, node) {

if (!sortedHead || node.value <= sortedHead.value) {

node.next = sortedHead;

return node;

}

let current = sortedHead;

while (current.next && current.next.value < node.value) {

current = current.next;

}

node.next = current.next;

current.next = node;

return sortedHead;

}

**BRUTE FORCE APPROACH ALGORITHM:**

function insertionSortIterative(head) {

let sorted = null;

let current = head;

while (current !== null) {

let next = current.next;

if (sorted === null || current.value <= sorted.value) {

current.next = sorted;

sorted = current;

} else {

let temp = sorted;

while (temp.next !== null && temp.next.value < current.value) {

temp = temp.next;

}

current.next = temp.next;

temp.next = current;

}

current = next;

}

return sorted;

}

**DIFFERENCE BETWEEN BRUTE FORCE AND DIVIDE AND CONQUER FOR INSERTION SORT ON SINGLY LINKED LIST**

**Brute Force:**

**Concept:**

* Insert each node from the original list into a new sorted list one by one.
* For each node, traverse the sorted list to find its correct position.

**How it works:**

1. Start with an empty sorted list.
2. Take the first node from the input list and add it to sorted list.
3. For each subsequent node, find the correct place in the sorted list by comparing values.
4. Insert the node at the right position.
5. Repeat until all nodes are inserted and sorted.

**Time Complexity:**

* Worst-case: O(n²)
* Each node can require traversing the sorted list (up to n nodes) for insertion.

**Divide and Conquer Approach:**

**Concept:**

* Break the list into two halves recursively.
* Sort each half separately (recursively).
* Merge the two sorted halves to get the final sorted list.

**How it works:**

1. If the list has 0 or 1 node, it's already sorted.
2. Split the list into two halves.
3. Recursively sort each half.
4. Merge the two sorted halves into a single sorted list.
5. Continue until the entire list is sorted.

**Time Complexity:**

* O(n log n)
* Efficiently divides the problem, sorts smaller parts, and merges them.

**Pros and Cons**

| **Feature** | **Brute Force** | **Divide and Conquer** |
| --- | --- | --- |
| Strategy | Linear insertions one by one | Recursive splitting and merging |
| Time Complexity | O(n²) | O(n log n) |
| Efficiency | Less efficient for large lists | More efficient for large lists |
| Implementation | Simpler | More complex (needs splitting & merging) |
| Ideal for | Small or nearly sorted lists | Large or unsorted lists |

**Algorithm Analysis**

**Brute Force (Iterative Insertion Sort)**

**Input:** Linked list with values: 4 -> 3 -> 1 -> 2

**Process:**

* Start with empty sorted list.
* Insert 4 into sorted list → 4
* Insert 3 at correct position → 3 -> 4
* Insert 1 → 1 -> 3 -> 4
* Insert 2 → 1 -> 2 -> 3 -> 4

**Divide and Conquer (Merge Sort on Linked List)**

**Input:** Linked list with values: 4 -> 3 -> 1 -> 2

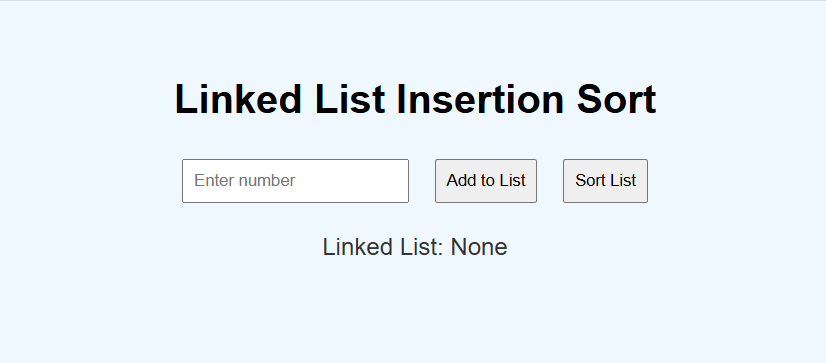
**Step1Divide**  
Split into two halves: 4 -> 3 and 1 -> 2

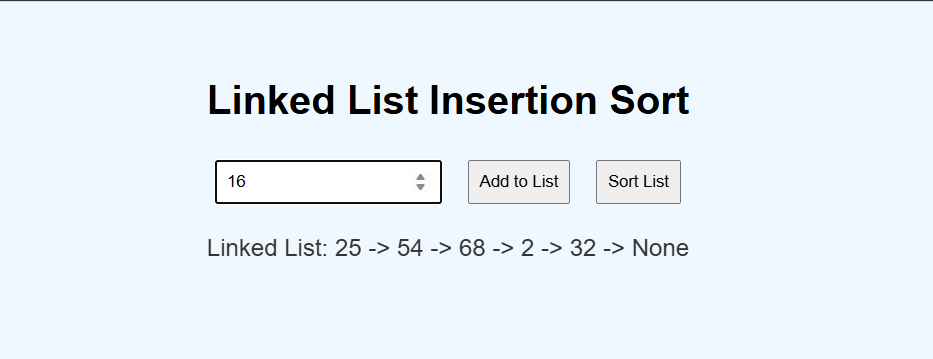
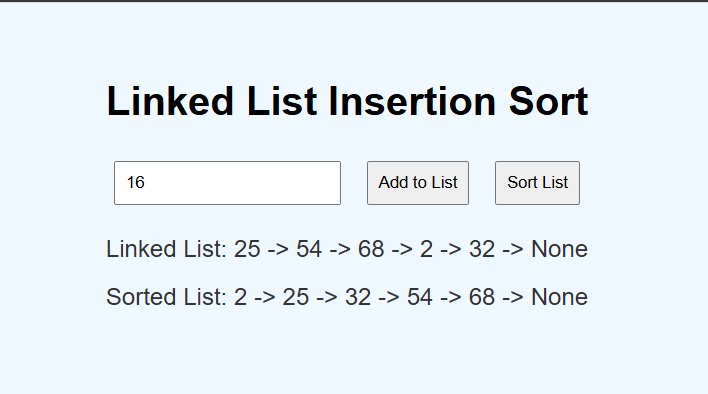
**Step 2: Conquer (Sort halves recursively)**

* Sort 4 -> 3 → 3 -> 4
* Sort 1 -> 2 → 1 -> 2

**Step3:Combine(Merge)**  
Merge 3 -> 4 and 1 -> 2 → 1 -> 2 -> 3 -> 4

**5.0. RESULTS:**



**GITHUB LINK: https://github.com/Vishnugopalsamy/DAA-microproject**