Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** 24AFAIPCL309

Class: SY AIML & Data Science **Expt. No. :** 03(B)

Title : Write a progra to implement a Doubly linked list

Problem Staement : To implement insertion, deletion, and traversal operations on a Doubly Linked List .

Software Required CodeBlock

Theory :

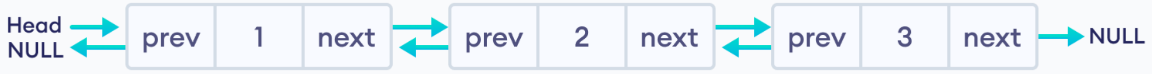
Doubly Linked List

A Doubly Linked List in C is a versatile data structure that connects nodes in a sequential chain using pointers. Each node in this list consists of three essential components:

- **Data**: Stores the actual value or information of the node.
- **Previous Pointer**: Contains the address of the preceding node in the sequence, allowing backward traversal.
- **Next Pointer**: Holds the address of the next node, enabling forward traversal.

Unlike a **Singly Linked List**, which can only be traversed in the forward direction, a doubly linked list allows traversal in **both directions** (forward and backward). This makes certain operations, like deletion, easier and more efficient.

Representation of Doubly Linked List :



Insertion Example :

Suppose we insert: **10, 20, 30**

Step 1: Insert 10 (Head = 10)

NULL <- [10] -> NULL

Step 2: Insert 20

NULL <- [10] <-> [20] -> NULL

Step 3: Insert 30

NULL <- [10] <-> [20] <-> [30] -> NULL

Deletion Example

Delete node at position **2** (value = 20):

Before: NULL <- [10] <-> [20] <-> [30] -> NULL
After : NULL <- [10] <-> [30] -> NULL

Advantages of Doubly Linked List

- Traversal is possible in both directions.
- Deletion is more efficient since we have access to the previous node.
- Insertions and deletions in the middle are faster compared to singly linked lists.

Disadvantages

- Requires extra memory for the prev pointer.

- More complex implementation compared to singly linked list.

Applications

- Used in implementing navigation (forward & backward in browsers).
- Undo/redo functionality in text editors.
- Memory management systems.
- Deques and advanced data structures.

Operations in Doubly Linked List :

1. Insertion at End

- A new node is created and inserted at the end of the list.
- If the list is empty, the new node becomes the **head**.
- Otherwise, traversal is done to the last node, and the new node is linked after it.

Algorithm:

1. Create a new node with given data.
2. If the list is empty:
   - Set head = newNode.
3. Else:
   - Traverse to the last node.
   - Update last node’s next pointer to newNode.
   - Set newNode’s prev to last node.

2. Deletion of a Node (by Position)

- The node at a given position is removed.
- If it is the first node, the head pointer is updated.
- If it is a middle or last node, pointers of adjacent nodes are updated.

Algorithm:

1. If the list is empty, display "List is empty".
2. If position = 1:
   - Update head to head->next.
   - If new head exists, set head->prev = NULL.
   - Free the first node.
3. Else:

- Traverse to the node at given position.
- Update its previous node's next to its next.
- Update its next node's prev to its prev.
- Free the node.

3. Display

- Traverse from head to the end, printing each node's data.
- Connections are shown using <->.

Code :

```
#include <stdio.h>
#include <stdlib.h>
// Structure of a node
struct Node {
   int data;
   struct Node* next;
   struct Node* prev;
};
struct Node* head = NULL;  // Global head pointer
// Insert element at end
void insertElement(int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = NULL;
   if (head == NULL) {
      newNode->prev = NULL;
      head = newNode;
   } else {
      struct Node* temp = head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
      newNode->prev = temp;
   }
}

// Delete node by position
void deleteNode(int pos) {
   if (head == NULL) {
      printf("List is empty\n");
      return;
   }
   struct Node* temp = head;
   if (pos == 1) {
      head = temp->next;
```

```
        if (head != NULL) {
            head->prev = NULL;
        }
        free(temp);
        return;
    }

    for (int i = 1; i < pos && temp != NULL; i++) {
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Position out of range!\n");
        return;
    }

    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    }
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }

    free(temp);
}

// Display the list
void display() {
    struct Node* temp = head;
    if (temp == NULL) {
        printf("List is empty\n");
        return;
    }
    printf("Doubly Linked List: ");
```

Output:

```
--- Doubly Linked List Menu ---
1. Insert Element
2. Delete Node (by position)
3. Display List
4. Exit
Enter choice: 1
Enter value: 10

Enter choice: 1
Enter value: 20

Enter choice: 1
Enter value: 30
```

Enter choice: 3
Doubly Linked List: 10 <-> 20 <-> 30 <-> NULL

Enter choice: 2
Enter position to delete: 2

Enter choice: 3
Doubly Linked List: 10 <-> 30 <-> NULL

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** 24AFAIPCL309

Class: SY AIML & Data Science **Expt. No. :** 03(C)

Title : Write a program to implement a Circular Linked List

Problem Staement

To implement insertion, deletion, and traversal operations on a Circular Linked List .

Software Required

CodeBlock

Theory :

Circular Linked List

A **linked list** is a linear data structure where elements (called **nodes**) are connected using pointers. Unlike arrays, linked lists do not require contiguous memory allocation, which makes insertion and deletion operations more efficient.

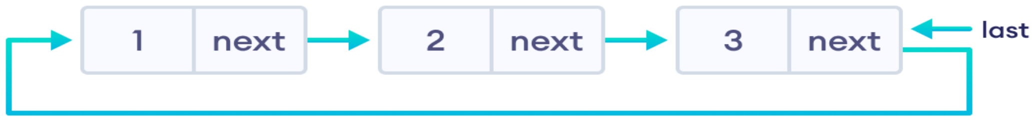
A **Circular Singly Linked List (CSLL)** is a special type of linked list where:

- Each node contains **data** and a **pointer** to the next node.
- The **last node** does not point to NULL, but instead points back to the **head node**, forming a **circle**.

This circular connection allows traversal to continue from the last node back to the first node, making the list suitable for applications where a circular traversal is needed (e.g., round-robin scheduling, circular queues).

1. Circular Singly Linked List

Here, the address of the last node consists of the address of the first node.



Advantages of Circular Singly Linked List

1. Efficient traversal: You can start at any node and reach all others in a circular manner.
2. Saves space compared to linear linked lists where last node points to NULL.
3. Useful in **applications requiring cyclic iteration** (e.g., multiplayer games, round-robin scheduling, buffers).

Disadvantages

1. Traversal is more complex compared to linear linked lists.
2. Extra care is needed while performing insertion and deletion to maintain circular structure.
3. No direct access to elements (sequential access only).

Algorithm :

Insertion at End

```
Step 1: Create newNode and assign value to data.
Step 2: If head == NULL:
           head = newNode
           newNode->next = head
       Else:
           temp = head
           While (temp->next != head):
              temp = temp->next
           temp->next = newNode
           newNode->next = head
```

	2. Deletion by Position

2. Deletion by Position

Step 1: If head == NULL → print "List is empty" and return.
Step 2: If pos == 1:
    If head->next == head:
        free(head)
        head = NULL
    Else:
        Find last node
        last->next = head->next
        temp = head
        head = head->next
        free(temp)
Step 3: Else:
    Traverse list until position
    If position invalid → print "Out of range"
    Else adjust prev->next = temp->next
        free(temp)

3. Display List

Step 1: If head == NULL → print "List is empty" and return.
Step 2: Start with temp = head
Step 3: Do
    Print temp->data
    temp = temp->next
  While (temp != head)
Step 4: Print (head) to indicate circular nature.

Code :

```
#include <stdio.h>
#include <stdlib.h>
// Structure of a node
struct Node {
   int data;
   struct Node* next;
};

struct Node* head = NULL;  // Global head pointer
// Function to insert element at the end
void insertElement(int value) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
```

```
    if (head == NULL) {  // First node
        head = newNode;
        newNode->next = head;  // Circular link
    } else {
        struct Node* temp = head;
        while (temp->next != head) {  // Traverse till last node
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;  // Point back to head
    }
}

// Function to delete node by position
void deleteNode(int pos) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }

    struct Node* temp = head;

    // Case 1: Delete first node
    if (pos == 1) {
        if (head->next == head) {  // Only one node
            free(head);
            head = NULL;
            return;
        }

        struct Node* last = head;
        while (last->next != head) {  // Find last node
            last = last->next;
        }

        last->next = head->next;
        head = head->next;
        free(temp);
        return;
    }

    // Case 2: Delete at other position
    struct Node* prev = NULL;
    for (int i = 1; i < pos && temp->next != head; i++) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == head) {  // Position out of range
        printf("Position out of range!\n");
```

```
        return;
    }

    prev->next = temp->next;
    free(temp);
}
// Function to display the list
void display() {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* temp = head;
    printf("Circular Singly Linked List: ");

    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);

    printf("(head)\n");
}

// Main function
int main() {
    int choice, value, pos;

    while (1) {
        printf("\n--- Circular Singly Linked List Menu ---\n");
        printf("1. Insert Element\n");
        printf("2. Delete Node (by position)\n");
        printf("3. Display List\n");
        printf("4. Exit\n");
        printf("Enter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insertElement(value);
                break;
            case 2:
                printf("Enter position to delete: ");
                scanf("%d", &pos);
                deleteNode(pos);
                break;
            case 3:
                display();
                break;
```

```
        case 4:
          exit(0);
        default:
          printf("Invalid choice!\n");
      }
   }

   return 0;
}
```

Output

```
--- Circular Singly Linked List Menu ---
1. Insert Element
2. Delete Node (by position)
3. Display List
4. Exit
Enter choice: 1
Enter value: 10

--- Circular Singly Linked List Menu ---
1. Insert Element
2. Delete Node (by position)
3. Display List
4. Exit
Enter choice: 1
Enter value: 20

--- Circular Singly Linked List Menu ---
1. Insert Element
2. Delete Node (by position)
3. Display List
4. Exit
Enter choice: 1
Enter value: 30

--- Circular Singly Linked List Menu ---
1. Insert Element
2. Delete Node (by position)
3. Display List
4. Exit
Enter choice: 3
Circular Singly Linked List: 10 -> 20 -> 30 -> (head)

--- Circular Singly Linked List Menu ---
1. Insert Element
2. Delete Node (by position)
3. Display List
4. Exit
Enter choice: 2
```

```
Enter position to delete: 2

--- Circular Singly Linked List Menu ---
1. Insert Element
2. Delete Node (by position)
3. Display List
4. Exit
Enter choice: 3
Circular Singly Linked List: 10 -> 30 -> (head)

--- Circular Singly Linked List Menu ---
1. Insert Element
2. Delete Node (by position)
3. Display List
4. Exit
Enter choice: 1
Enter value: 40

--- Circular Singly Linked List Menu ---
1. Insert Element
2. Delete Node (by position)
3. Display List
4. Exit
Enter choice: 3
Circular Singly Linked List: 10 -> 30 -> 40 -> (head)
--- Circular Singly Linked List Menu ---
1. Insert Element
2. Delete Node (by position)
3. Display List
4. Exit
Enter choice: 4
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** 24AFAIPCL309

Class: SY AIML & Data Science **Expt. No. :** 04

Title : Write a program to implement a Stack Using linked list such that the push and pop operation of the stack still take O(1) time.

Problem Staement	To implement a Stack Using linked list
Software Required	CodeBlock
Theory :	Stack Using Linked List

What is a Stack?

A **stack** is a **linear data structure** that follows the **Last In First Out (LIFO)** principle. This means the element added last is removed first.

Stack Operations:

- **Push** – Insert an element onto the stack
- **Pop** – Remove the top element
- **Peek/Top** – View the top element without removing it
- **isEmpty** – Check if the stack is empty

Why Use Linked List for Stack?

- Dynamic size: No need to predefine size as in array-based stack
- Efficient: Memory is allocated only when needed

Linked List Stack Structure:

Each node contains:

- `data` – value of the element
- `next` – pointer to the next node

The **top** of the stack is represented by the head of the linked list.

Algorithm

PUSH(x)

1. Create a new node
2. Assign data = x
3. Set new_node->next = top
4. Update top = new_node

POP()

1. If top is NULL
   - Stack Underflow (Nothing to pop)
2. Else
   - Temp = top
   - top = top->next
   - Delete temp

PEEK()

1. If top is NULL
   - Stack is empty
2. Else
   - Return top->data

isEmpty()

Return true if top == NULL, else false

Code :

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

// Define structure for a stack node
struct Node {
    int data;
    struct Node* next;
};

// Initialize stack as empty
struct Node* top = NULL;

// Function to push an element onto the stack
void push(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Stack Overflow! Unable to allocate memory.\n");
        return;
    }
    newNode->data = value;
```

```c
    newNode->next = top;  // Link new node to previous top
    top = newNode;        // Make new node the top
    printf("%d pushed to stack.\n", value);
}

// Function to pop an element from the stack
void pop() {
    if (top == NULL) {
        printf("Stack Underflow! No elements to pop.\n");
        return;
    }
    struct Node* temp = top;
    printf("%d popped from stack.\n", top->data);
    top = top->next;
    free(temp);
}

// Function to peek at the top element
void peek() {
    if (top == NULL) {
        printf("Stack is empty! Nothing to peek.\n");
    } else {
        printf("Top element is: %d\n", top->data);
    }
}

// Function to display elements of the stack
void display() {
    if (top == NULL) {
        printf("Stack is empty.\n");
        return;
    }
    struct Node* temp = top;
    printf("Stack elements (top to bottom): ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function to perform stack operations
int main() {
    int choice, value;

    while (1) {
        printf("\n--- Stack using Linked List ---\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
```

```
                printf("Enter value to push: ");
                scanf("%d", &value);
                push(value);
                break;
            case 2:
                pop();
                break;
            case 3:
                peek();
                break;
            case 4:
                display();
                break;
            case 5:
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

Output :

```
--- Stack using Linked List ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 10
10 pushed to stack.

--- Stack using Linked List ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push: 20
20 pushed to stack.

--- Stack using Linked List ---
Enter your choice: 4
Stack elements (top to bottom): 20 -> 10 -> NULL

Enter your choice: 3
Top element is: 20

Enter your choice: 2
20 popped from stack.

Enter your choice: 4
```

	Stack elements (top to bottom): 10 -> NULL

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** 24AFAIPCL309

Class: SY AIML & Data Science **Expt. No. :** 7

Title : Write a program to implement a Insertion Sorting

Problem Staement

To implement insertion sort.

Software Required

CodeBlock

Theory :

Insertion Sort is one of the simplest and most intuitive sorting algorithms. It is based on the idea of **building a sorted list one element at a time** by comparing and inserting each new element into its correct position in an already sorted part of the list.

Concept and Working Principle

- The array is divided into two parts:
    - **Left side (sorted part)** – elements that are already arranged in order.
    - **Right side (unsorted part)** – elements that are yet to be arranged.
- The algorithm starts from the **second element** (index 1), assuming that the first element is already sorted.
- It picks the current element (called **key**) and compares it with the elements on the left side.
- All elements that are **greater than the key** are **shifted one position to the right** to make space for the key.
- The key is then placed in its correct sorted position.

- This process is repeated for all elements in the array until the entire list becomes sorted.

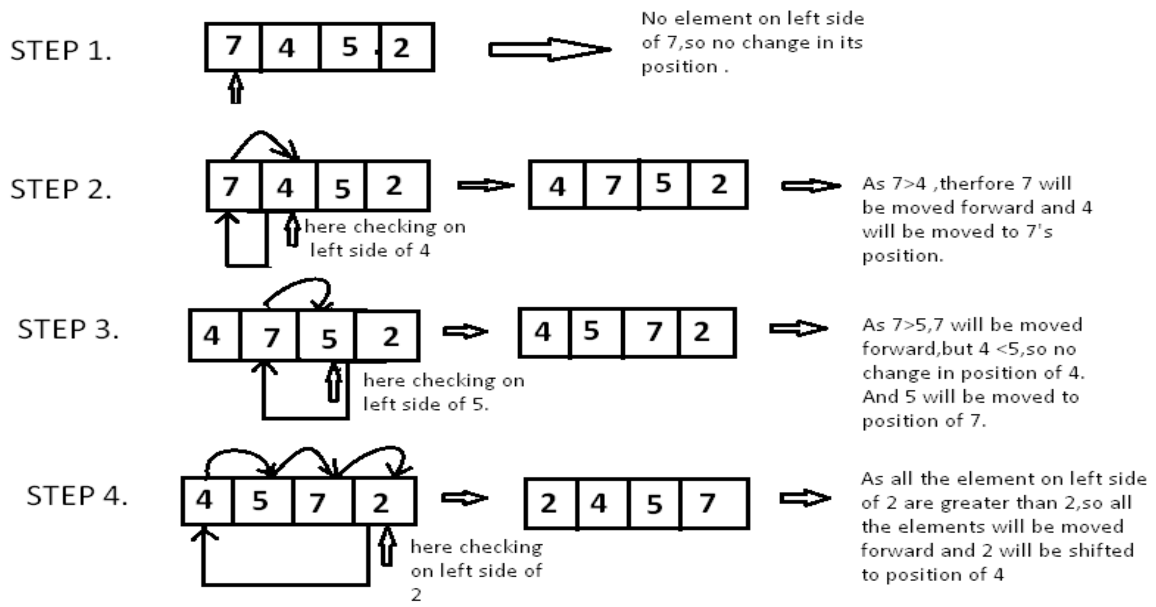
Advantages:

1. Simple and easy to understand.
2. Efficient for small datasets.
3. Performs well for nearly sorted data.
4. Requires no additional memory (in-place algorithm).
5. Stable sort — preserves relative order of equal elements.

5. Disadvantages:

1. Inefficient for large datasets (O(n²) time complexity).
2. More comparisons and shifts as the number of elements increases.
3. Not suitable for large lists compared to algorithms like Merge Sort or Quick Sort.

Example :
Take Array[]=[7,4,5,2]



Code :	**ALGORITHM:** 1. Start 2. Read the number of elements and the array elements. 3. Repeat for i = 1 to n-1: ○ Set temp = arr[i] ○ Set j = i - 1 ○ While j >= 0 and arr[j] > temp: ▪ Move arr[j] to position j + 1 ▪ Decrement j by 1 ○ Insert key at position j + 1 4. Display the sorted array. 5. Stop

```
#include <stdio.h>

int main() {
    int a[100], n, i, j, temp;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    // Insertion Sort logic
    for (i = 1; i < n; i++) {
        temp = a[i];
        j = i - 1;
        while (j >= 0 && a[j] > temp) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = temp;
    }
```

	`` printf("Sorted array:\n"); for (i = 0; i < n; i++) printf("%d ", a[i]); return 0; } ``` Output: Enter number of elements: 5 Enter 5 elements: 5 2 4 6 1 Sorted array: 1 2 4 5 6

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Data Structure Lab **Subject Code :** 24AFAIPCL309

Class: SY AIML & Data Science **Expt. No. :** 8

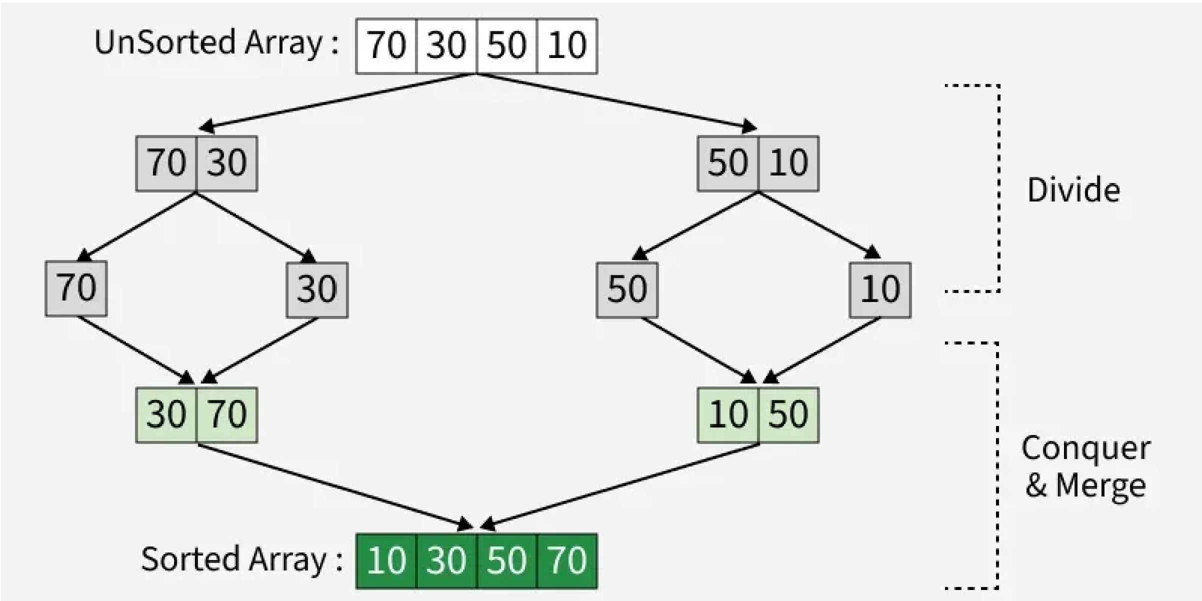
Title : Write a program to implement a Merge Sort

Problem Staement	To implement merge sort
Software Required	CodeBlock
Theory :	**Theory:** **Merge Sort** is a **divide and conquer** sorting algorithm. It divides the unsorted array into smaller subarrays until each subarray contains a single element, then merges these subarrays to produce new sorted subarrays until there is only one sorted array remaining. **Working Steps:** 1. **Divide:** Split the array into two halves. 2. **Conquer:** Recursively sort the two halves. 3. **Combine:** Merge the two sorted halves into a single sorted array. **Concept of Divide and Conquer:** The **Divide and Conquer** approach breaks a problem into smaller subproblems, solves each subproblem independently, and then combines the results to obtain the final solution. Merge Sort applies this idea in three main steps: 1. **Divide:** The array is divided into two halves (approximately equal parts).

This division continues recursively until each subarray contains a single element.

2. **Conquer (Sort):**
   Each of these smaller subarrays is considered sorted because a single element is trivially sorted.
3. **Combine (Merge):**
   The sorted subarrays are merged together in a way that the resulting array is also sorted.
   This merging process continues until all subarrays are combined into one fully sorted array.

Example to Understand Merge Sort:



Let’s consider an example array:

[70,30,50,10]

Step 1 – Divide:
Divide the array into two halves recursively:

[70,30]   and   [50,10]

And again:

[70] [30] [50] [10]

Step 2 – Conquer (Merge):
Now merge pairs of subarrays while sorting them:

[30,70] [10,50]

Step 3 – Combine:
Continue merging the sorted subarrays:

[10,30,50,70]

Finally:

[10,30,50,70]

Thus, the array is completely sorted.

Advantages:

1. Very efficient for large datasets.
2. Guarantees **O(n log n)** time complexity for all cases.
3. It is a **stable** sorting algorithm.
4. Performs well on linked lists (does not require random access).
5. Can be easily implemented using recursion.

Disadvantages:

1. Requires **extra memory** (O(n)) for temporary arrays during the merge process.
2. Not suitable for small datasets compared to simpler algorithms like **Insertion Sort** or **Bubble Sort**.
3. Recursive calls increase overhead and may cause stack overflow for very large arrays.

Algorithm: MergeSort(A, low, high)

1. If low < high
    1. Find mid = (low + high) / 2
    2. Call MergeSort(A, low, mid)
    3. Call MergeSort(A, mid + 1, high)
    4. Call Merge(A, low, mid, high)

Code :	**Algorithm: Merge(A, low, mid, high)** 1. Create temporary arrays for left and right subarrays. 2. Compare elements of both subarrays and copy the smaller one to the main array. 3. Copy the remaining elements of left or right subarray (if any).

```
#include <stdio.h>

void merge(int a[], int left, int mid, int right) {
    int temp[100];
    int i = left;      // start of first half
    int j = mid + 1;   // start of second half
    int k = left;      // start of temp array

    // Compare and copy smaller element into temp
    while (i <= mid && j <= right) {
        if (a[i] < a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }

    // Copy remaining elements (if any) from first half
    while (i <= mid)
        temp[k++] = a[i++];

    // Copy remaining elements (if any) from second half
    while (j <= right)
        temp[k++] = a[j++];

    // Copy back to original array
    for (i = left; i <= right; i++)
        a[i] = temp[i];
}

void mergeSort(int a[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;

        // Divide the array into two halves
```

```
        mergeSort(a, left, mid);
        mergeSort(a, mid + 1, right);

        // Merge the two sorted halves
        merge(a, left, mid, right);
    }
}

int main() {
    int a[100], n, i;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    // Call merge sort
    mergeSort(a, 0, n - 1);

    printf("Sorted array:\n");
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);

    return 0;
}
```

Output :

Enter number of elements: 6
Enter 6 elements:
10 3 5 2 8 1

Sorted array:
1 2 3 5 8 10