**Data Structure Lab Assignment**

1. **>> C program to implement insertion at the first position, last position, and any position of a singly linked list**.

:

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

// Define a structure for a node in the linked list struct Node {

int data;

struct Node\* next;

};

// Function to create a new node struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node)); if (newNode == NULL) {

printf("Memory allocation failed.\n"); exit(1);

}

newNode->data = data; newNode->next = NULL; return newNode;

}

// Function to insert a node at the beginning of the linked list struct Node\* insertAtFirst(struct Node\* head, int data) {

struct Node\* newNode = createNode(data); newNode->next = head;

return newNode;

}

// Function to insert a node at the end of the linked list struct Node\* insertAtLast(struct Node\* head, int data) {

struct Node\* newNode = createNode(data); if (head == NULL) {

return newNode;

}

struct Node\* current = head; while (current->next != NULL) {

current = current->next;

}

current->next = newNode; return head;

}

// Function to insert a node at a specific position in the linked list

struct Node\* insertAtPosition(struct Node\* head, int data, int position) { if (position < 1) {

printf("Invalid position.\n"); return head;

}

struct Node\* newNode = createNode(data); if (position == 1) {

newNode->next = head; return newNode;

}

struct Node\* current = head; int currentPosition = 1;

while (current != NULL && currentPosition < position - 1) { current = current->next;

currentPosition++;

}

if (current == NULL) { printf("Invalid position.\n"); return head;

}

newNode->next = current->next; current->next = newNode; return head;

}

// Function to print the linked list

void printLinkedList(struct Node\* head) { struct Node\* current = head;

while (current != NULL) { printf("%d -> ", current->data); current = current->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

// Insert at the first position head = insertAtFirst(head, 10); head = insertAtFirst(head, 5);

// Insert at the last position head = insertAtLast(head, 15); head = insertAtLast(head, 20);

// Insert at a specific position

head = insertAtPosition(head, 25, 3);

// Print the linked list printf("Linked List: "); printLinkedList(head);

return 0;

}

*Time Complexity Analysis :*

* Insertion at the first position (insertAtFirst) : O(1)
* Insertion at the last position (insertAtLast) : O(n), where n is the number of nodes in the list, as we need to traverse the entire list to find the last node.
* Insertion at a specific position (insertAtPosition) : O(n), where n is the position at which you want to insert the node. In the worst case, it may require traversing the entire list.

Overall, the time complexity of these insertion operations in the singly linked list is O(n) in the worst case, where n is the number of nodes in the list.

1. **>> C program to implement deletion at the first position, last position, and any position of a singly linked list**

***:***

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

// Define a structure for a node in the linked list struct Node {

int data;

struct Node\* next;

};

// Function to create a new node struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node)); if (newNode == NULL) {

printf("Memory allocation failed.\n"); exit(1);

}

newNode->data = data; newNode->next = NULL; return newNode;

}

// Function to insert a node at the beginning of the linked list struct Node\* insertAtFirst(struct Node\* head, int data) {

struct Node\* newNode = createNode(data); newNode->next = head;

return newNode;

}

// Function to insert a node at the end of the linked list struct Node\* insertAtLast(struct Node\* head, int data) {

struct Node\* newNode = createNode(data); if (head == NULL) {

return newNode;

}

struct Node\* current = head; while (current->next != NULL) {

current = current->next;

}

current->next = newNode; return head;

}

// Function to delete the first node of the linked list struct Node\* deleteAtFirst(struct Node\* head) {

if (head == NULL) {

printf("List is empty. Cannot delete.\n"); return NULL;

}

struct Node\* temp = head; head = head->next; free(temp);

return head;

}

// Function to delete the last node of the linked list struct Node\* deleteAtLast(struct Node\* head) {

if (head == NULL) {

printf("List is empty. Cannot delete.\n"); return NULL;

}

if (head->next == NULL) {

// If there's only one node in the list free(head);

return NULL;

}

struct Node\* current = head; struct Node\* previous = NULL; while (current->next != NULL) {

previous = current; current = current->next;

}

previous->next = NULL; free(current);

return head;

}

// Function to delete a node at a specific position in the linked list struct Node\* deleteAtPosition(struct Node\* head, int position) {

if (position < 1 || head == NULL) {

printf("Invalid position or empty list. Cannot delete.\n"); return head;

}

if (position == 1) {

struct Node\* temp = head; head = head->next; free(temp);

return head;

}

struct Node\* current = head; struct Node\* previous = NULL;

int currentPosition = 1;

while (current != NULL && currentPosition < position) { previous = current;

current = current->next; currentPosition++;

}

if (current == NULL) {

printf("Invalid position. Cannot delete.\n"); return head;

}

previous->next = current->next; free(current);

return head;

}

// Function to print the linked list

void printLinkedList(struct Node\* head) { struct Node\* current = head;

while (current != NULL) { printf("%d -> ", current->data); current = current->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

// Insert at the first position head = insertAtFirst(head, 10); head = insertAtFirst(head, 5);

// Insert at the last position head = insertAtLast(head, 15); head = insertAtLast(head, 20);

// Insert at a specific position head = insertAtLast(head, 25);

// Print the linked list printf("Linked List: "); printLinkedList(head);

// Delete at the first position head = deleteAtFirst(head);

printf("After deleting first node: "); printLinkedList(head);

// Delete at the last position head = deleteAtLast(head);

printf("After deleting last node: "); printLinkedList(head);

// Delete at a specific position head = deleteAtPosition(head, 3);

printf("After deleting node at position 3: "); printLinkedList(head);

return 0;

}

*Time Complexity Analysis :*

* Deletion at the first position (deleteAtFirst) : O(1)
* Deletion at the last position (deleteAtLast) : O(n), where n is the number of nodes in the list, as we need to traverse the entire list to find the last node.
* Deletion at a specific position (deleteAtPosition) : O(n), where n is the position from which you want to delete the node. In the worst case, it may require traversing the entire list.

Overall, the time complexity of these deletion operations in the singly linked list is O(n) in the worst case, where n is the number of nodes in the list.

1. **>> C program to implement insertion and deletion at the first position, last position, and any position of a doubly linked list :**

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

// Define a structure for a node in the doubly linked list struct Node {

int data;

struct Node\* prev; struct Node\* next;

};

// Function to create a new node struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node)); if (newNode == NULL) {

printf("Memory allocation failed.\n"); exit(1);

}

newNode->data = data; newNode->prev = NULL; newNode->next = NULL; return newNode;

}

// Function to insert a node at the beginning of the doubly linked list struct Node\* insertAtFirst(struct Node\* head, int data) {

struct Node\* newNode = createNode(data); newNode->next = head;

if (head != NULL) {

head->prev = newNode;

}

return newNode;

}

// Function to insert a node at the end of the doubly linked list struct Node\* insertAtLast(struct Node\* head, int data) {

struct Node\* newNode = createNode(data); if (head == NULL) {

return newNode;

}

struct Node\* current = head; while (current->next != NULL) {

current = current->next;

}

current->next = newNode; newNode->prev = current; return head;

}

// Function to insert a node at a specific position in the doubly linked list struct Node\* insertAtPosition(struct Node\* head, int data, int position) {

if (position < 1) { printf("Invalid position.\n"); return head;

}

struct Node\* newNode = createNode(data); if (position == 1) {

newNode->next = head; if (head != NULL) {

head->prev = newNode;

}

return newNode;

}

struct Node\* current = head; int currentPosition = 1;

while (current != NULL && currentPosition < position - 1) { current = current->next;

currentPosition++;

}

if (current == NULL) { printf("Invalid position.\n"); return head;

}

newNode->next = current->next; newNode->prev = current;

if (current->next != NULL) { current->next->prev = newNode;

}

current->next = newNode; return head;

}

// Function to delete a node at the beginning of the doubly linked list struct Node\* deleteAtFirst(struct Node\* head) {

if (head == NULL) {

printf("List is empty. Cannot delete.\n"); return NULL;

}

struct Node\* temp = head; head = head->next;

if (head != NULL) { head->prev = NULL;

}

free(temp); return head;

}

// Function to delete a node at the end of the doubly linked list struct Node\* deleteAtLast(struct Node\* head) {

if (head == NULL) {

printf("List is empty. Cannot delete.\n"); return NULL;

}

if (head->next == NULL) {

// If there's only one node in the list free(head);

return NULL;

}

struct Node\* current = head; while (current->next != NULL) {

current = current->next;

}

current->prev->next = NULL; free(current);

return head;

}

// Function to delete a node at a specific position in the doubly linked list struct Node\* deleteAtPosition(struct Node\* head, int position) {

if (position < 1 || head == NULL) {

printf("Invalid position or empty list. Cannot delete.\n"); return head;

}

if (position == 1) {

struct Node\* temp = head; head = head->next;

if (head != NULL) { head->prev = NULL;

}

free(temp); return head;

}

struct Node\* current = head; int currentPosition = 1;

while (current != NULL && currentPosition < position) { current = current->next;

currentPosition++;

}

if (current == NULL) {

printf("Invalid position. Cannot delete.\n"); return head;

}

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

current->next->prev = current->prev;

}

free(current); return head;

}

// Function to print the doubly linked list

void printDoublyLinkedList(struct Node\* head) { struct Node\* current = head;

while (current != NULL) { printf("%d <-> ", current->data); current = current->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = NULL;

// Insert at the first position head = insertAtFirst(head, 10); head = insertAtFirst(head, 5);

// Insert at the last position head = insertAtLast(head, 15); head = insertAtLast(head, 20);

// Insert at a specific position

head = insertAtPosition(head, 25, 3);

// Print the doubly linked list printf("Doubly Linked List: "); printDoublyLinkedList(head);

// Delete at the first position head = deleteAtFirst(head);

printf("After deleting first node: "); printDoublyLinkedList(head);

// Delete at the last position head = deleteAtLast(head);

printf("After deleting last node: ");

printDoublyLinkedList(head);

// Delete at a specific position head = deleteAtPosition(head, 2);

printf("After deleting node at position 2: "); printDoublyLinkedList(head);

return 0;

}

*Time Complexity Analysis :*

* Insertion at the first position (insertAtFirst) : O(1)
* Insertion at the last position (insertAtLast) : O(1)
* Insertion at a specific position (insertAtPosition) : O(n), where n is the position at which you want to insert the node.
* Deletion at the first position (deleteAtFirst) : O(1)
* Deletion at the last position (deleteAtLast) : O(1)
* Deletion at a specific position (deleteAtPosition) : O(n), where n is the position from which you want to delete the node.

Overall, the time complexity of these insertion and deletion operations in the doubly linked list is O(n) in the worst case, where n is the position at which you want to perform the operation.

1. **>> C program to implement heap sort using arrays and then use the same heap data structure to implement a priority queue :**

#include <stdio.h>

// Function to heapify a subtree rooted at node i void heapify(int arr[], int n, int i) {

int largest = i; // Initialize the largest as the root int left = 2 \* i + 1; // Left child

int right = 2 \* i + 2; // Right child

// If the left child is larger than the root if (left < n && arr[left] > arr[largest])

largest = left;

// If the right child is larger than the largest so far if (right < n && arr[right] > arr[largest])

largest = right;

// If the largest is not the root if (largest != i) {

int temp = arr[i]; arr[i] = arr[largest]; arr[largest] = temp;

// Recursively heapify the affected subtree heapify(arr, n, largest);

}

}

// Function to perform heap sort void heapSort(int arr[], int n) {

// Build a max heap

for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);

// Extract elements from the heap one by one for (int i = n - 1; i > 0; i--) {

// Move the current root to the end int temp = arr[0];

arr[0] = arr[i]; arr[i] = temp;

// Call heapify on the reduced heap heapify(arr, i, 0);

}

}

// Function to insert an element into the max heap (priority queue) void insert(int arr[], int \*n, int value) {

// Increase the size of the heap (\*n)++;

int i = \*n - 1;

// Find the correct position for the new element while (i > 0 && value > arr[(i - 1) / 2]) {

arr[i] = arr[(i - 1) / 2]; i = (i - 1) / 2;

}

// Insert the new element at its correct position arr[i] = value;

}

// Function to remove and return the maximum element from the max heap (priority queue) int extractMax(int arr[], int \*n) {

if (\*n <= 0)

return -1; // Heap is empty

if (\*n == 1) { (\*n)--;

return arr[0];

}

// Store the maximum value and remove it from the heap int max = arr[0];

arr[0] = arr[(\*n) - 1]; (\*n)--;

// Heapify the root heapify(arr, \*n, 0);

return max;

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int n = sizeof(arr) / sizeof(arr[0]);

// Perform heap sort heapSort(arr, n);

printf("Sorted array using heap sort: "); for (int i = 0; i < n; i++)

printf("%d ", arr[i]); printf("\n");

// Implementing a priority queue using a max heap int maxHeap[1000];

int heapSize = 0;

// Insert elements into the priority queue insert(maxHeap, &heapSize, 15);

insert(maxHeap, &heapSize, 10);

insert(maxHeap, &heapSize, 20);

// Extract and print the maximum element from the priority queue

printf("Max element from the priority queue: %d\n", extractMax(maxHeap, &heapSize));

return 0;

}

*Time Complexity Analysis :*

1. Heap Sort :

* Building the max heap : O(n)
* Extracting elements one by one : O(n \* log(n))
* Overall time complexity : O(n \* log(n))

1. Priority Queue Operations :

* Insertion : O(log(n)) on average (worst-case scenario)
* Extraction of the maximum element : O(log(n)) on average (worst-case scenario)

The overall time complexity of the priority queue operations (insertion and extraction) is O(log(n)) on average (worst-case scenario), where "n" is the number of elements in the heap.

1. **>> C programs to implement a circular queue using array and linked list :**

*A >> Circular Queue using Arrays :*

#include <stdio.h>

#include <stdlib.h> #define MAX\_SIZE 5

// Structure for the circular queue struct CircularQueue {

int data[MAX\_SIZE]; int front, rear;

};

// Initialize the circular queue

void initialize(struct CircularQueue\* queue) { queue->front = -1;

queue->rear = -1;

}

// Check if the queue is empty

int isEmpty(struct CircularQueue\* queue) {

return (queue->front == -1 && queue->rear == -1);

}

// Check if the queue is full

int isFull(struct CircularQueue\* queue) {

return ((queue->rear + 1) % MAX\_SIZE == queue->front);

}

// Enqueue (insert) an element into the circular queue void enqueue(struct CircularQueue\* queue, int value) {

if (isFull(queue)) {

printf("Queue is full. Cannot enqueue.\n"); return;

}

if (isEmpty(queue)) { queue->front = 0;

queue->rear = 0;

} else {

queue->rear = (queue->rear + 1) % MAX\_SIZE;

}

queue->data[queue->rear] = value;

}

// Dequeue (remove) an element from the circular queue int dequeue(struct CircularQueue\* queue) {

if (isEmpty(queue)) {

printf("Queue is empty. Cannot dequeue.\n");

return -1; // A sentinel value to indicate an empty queue

}

int removedValue = queue->data[queue->front]; if (queue->front == queue->rear) {

queue->front = -1;

queue->rear = -1;

} else {

queue->front = (queue->front + 1) % MAX\_SIZE;

}

return removedValue;

}

// Display the elements of the circular queue void display(struct CircularQueue\* queue) {

if (isEmpty(queue)) { printf("Queue is empty.\n"); return;

}

int i = queue->front; printf("Circular Queue: "); do {

printf("%d ", queue->data[i]); i = (i + 1) % MAX\_SIZE;

} while (i != (queue->rear + 1) % MAX\_SIZE); printf("\n");

}

int main() {

struct CircularQueue queue; initialize(&queue);

enqueue(&queue, 10);

enqueue(&queue, 20);

enqueue(&queue, 30);

enqueue(&queue, 40);

enqueue(&queue, 50); display(&queue);

printf("Dequeued: %d\n", dequeue(&queue)); printf("Dequeued: %d\n", dequeue(&queue));

enqueue(&queue, 60);

enqueue(&queue, 70); display(&queue);

return 0;

}

*B >> Circular Queue using Linked List :*

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

// Structure for a node in the circular queue struct Node {

int data;

struct Node\* next;

};

// Structure for the circular queue struct CircularQueue {

struct Node\* front; struct Node\* rear;

};

// Initialize the circular queue

void initialize(struct CircularQueue\* queue) { queue->front = NULL;

queue->rear = NULL;

}

// Check if the queue is empty

int isEmpty(struct CircularQueue\* queue) {

return (queue->front == NULL && queue->rear == NULL);

}

// Enqueue (insert) an element into the circular queue void enqueue(struct CircularQueue\* queue, int value) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node)); if (newNode == NULL) {

printf("Memory allocation failed. Cannot enqueue.\n"); return;

}

newNode->data = value; newNode->next = NULL;

if (isEmpty(queue)) {

queue->front = newNode; queue->rear = newNode;

newNode->next = newNode; // Circular linking

} else {

newNode->next = queue->front; queue->rear->next = newNode; queue->rear = newNode;

}

}

// Dequeue (remove) an element from the circular queue int dequeue(struct CircularQueue\* queue) {

if (isEmpty(queue)) {

printf("Queue is empty. Cannot dequeue.\n");

return -1; // A sentinel value to indicate an empty queue

}

int removedValue = queue->front->data; if (queue->front == queue->rear) {

free(queue->front); queue->front = NULL;

queue->rear = NULL;

} else {

struct Node\* temp = queue->front; queue->front = queue->front->next; queue->rear->next = queue->front; free(temp);

}

return removedValue;

}

// Display the elements of the circular queue void display(struct CircularQueue\* queue) {

if (isEmpty(queue)) { printf("Queue is empty.\n"); return;

}

struct Node\* current = queue->front; printf("Circular Queue: ");

do {

printf("%d ", current->data); current = current->next;

} while (current != queue->front);

printf("\n");

}

int main() {

struct CircularQueue queue; initialize(&queue);

enqueue(&queue, 10);

enqueue(&queue, 20);

enqueue(&queue, 30);

enqueue(&queue, 40);

enqueue(&queue, 50); display(&queue);

printf("Dequeued: %d\n", dequeue(&queue)); printf("Dequeued: %d\n", dequeue(&queue));

enqueue(&queue, 60);

enqueue(&queue, 70); display(&queue);

return 0;

}

*Time Complexity Analysis :*

1. Circular Queue using Arrays :

* Enqueue (insert) : O(1)
* Dequeue (remove) : O(1)
* Display : O(n), where n is the number of elements in the queue

1. Circular Queue using Linked List :

* Enqueue (insert) : O(1)
* Dequeue (remove) : O(1)
* Display : O(n), where n is the number of elements in the queue

Both implementations provide constant time complexity for enqueuing and dequeuing elements, making them efficient for typical use cases of circular queues. The display operation has a linear time complexity, as it involves traversing all elements in the queue.

1. **>> C programs to implement a merge sort using array and linked list :**

*A >> Merge Sort using Arrays :*

#include <stdio.h>

// Function to merge two subarrays of arr[]

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r) { int i, j, k;

int n1 = m - l + 1; int n2 = r - m;

// Create temporary arrays int L[n1], R[n2];

// Copy data to temporary arrays L[] and R[] for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

// Merge the temporary arrays back into arr[l..r] i = 0; // Initial index of first subarray

j = 0; // Initial index of second subarray k = l; // Initial index of merged subarray while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i]; i++;

} else {

arr[k] = R[j]; j++;

} k++;

}

// Copy the remaining elements of L[], if there are any while (i < n1) {

arr[k] = L[i]; i++;

k++;

}

// Copy the remaining elements of R[], if there are any while (j < n2) {

arr[k] = R[j]; j++;

k++;

}

}

// Function to perform merge sort on arr[l..r] void mergeSort(int arr[], int l, int r) {

if (l < r) {

// Same as (l+r)/2, but avoids overflow for large l and r int m = l + (r - l) / 2;

// Sort first and second halves mergeSort(arr, l, m); mergeSort(arr, m + 1, r);

// Merge the sorted halves merge(arr, l, m, r);

}

}

// Function to print an array

void printArray(int arr[], int size) { for (int i = 0; i < size; i++)

printf("%d ", arr[i]); printf("\n");

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int arr\_size = sizeof(arr) / sizeof(arr[0]);

printf("Given array is \n"); printArray(arr, arr\_size);

mergeSort(arr, 0, arr\_size - 1);

printf("Sorted array is \n"); printArray(arr, arr\_size); return 0;

}

*B >> Merge Sort using Linked List :*

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

// Structure for a node in the linked list struct Node {

int data;

struct Node\* next;

};

// Function to merge two sorted linked lists into a single sorted list struct Node\* merge(struct Node\* left, struct Node\* right) {

struct Node\* result = NULL;

// Base cases

if (left == NULL) return right;

if (right == NULL) return left;

// Compare the data of the two nodes if (left->data <= right->data) {

result = left;

result->next = merge(left->next, right);

} else {

result = right;

result->next = merge(left, right->next);

}

return result;

}

// Function to perform merge sort on a linked list void mergeSort(struct Node\*\* headRef) {

struct Node\* head = \*headRef; struct Node\* left;

struct Node\* right;

// Base case: If the list is empty or has only one element, it's already sorted if (head == NULL || head->next == NULL)

return;

// Split the list into two halves struct Node\* slow = head; struct Node\* fast = head->next; while (fast != NULL) {

fast = fast->next;

if (fast != NULL) { slow = slow->next; fast = fast->next;

}

}

left = head;

right = slow->next; slow->next = NULL;

// Recursively sort the two halves mergeSort(&left); mergeSort(&right);

// Merge the sorted halves

\*headRef = merge(left, right);

}

// Function to insert a node at the end of the linked list void insert(struct Node\*\* headRef, int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node)); newNode->data = data;

newNode->next = NULL;

if (\*headRef == NULL) {

\*headRef = newNode; return;

}

struct Node\* current = \*headRef; while (current->next != NULL)

current = current->next; current->next = newNode;

}

// Function to print a linked list void printList(struct Node\* node) {

while (node != NULL) { printf("%d ", node->data); node = node->next;

}

printf("\n");

}

int main() {

struct Node\* head = NULL;

insert(&head, 12);

insert(&head, 11);

insert(&head, 13);

insert(&head, 5);

insert(&head, 6);

insert(&head, 7);

printf("Given linked list is \n"); printList(head);

mergeSort(&head);

printf("Sorted linked list is \n"); printList(head);

return 0;

}

*Time Complexity Analysis :*

1. Merge Sort using Arrays :

* Time Complexity : O(n \* log(n)) in the worst and average case, where "n" is the number of elements in the array.

1. Merge Sort using Linked List :

* Time Complexity : O(n \* log(n)) in the worst and average case, where "n" is the number of elements in the linked list.

Both implementations of merge sort have the same time complexity and are efficient for sorting large datasets. They are stable sorting algorithms and can handle various types of data.

1. ***>> C program that implements a hash table using arrays with linear probing and chaining to handle collisions :***

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

#define SIZE 10

// Structure for a node in the chaining linked list struct Node {

int key; int data;

struct Node\* next;

};

// Structure for a hash table entry struct HashEntry {

int key; int data;

};

// Initialize the hash table

void initialize(struct HashEntry\* hashTable) { for (int i = 0; i < SIZE; i++) {

hashTable[i].key = -1; // Initialize keys to -1 (indicating empty) hashTable[i].data = -1; // Initialize data to -1 (indicating empty)

}

}

// Hash function (simple modulo-based hash)

int hash(int key) { return key % SIZE;

}

// Function to insert a key-value pair into the hash table void insert(struct HashEntry\* hashTable, int key, int data) {

int index = hash(key);

// Linear probing to handle collisions while (hashTable[index].key != -1) {

index = (index + 1) % SIZE;

}

hashTable[index].key = key; hashTable[index].data = data;

}

// Function to retrieve the value associated with a key from the hash table int get(struct HashEntry\* hashTable, int key) {

int index = hash(key);

// Linear probing to find the key

while (hashTable[index].key != key) { index = (index + 1) % SIZE;

// Key not found in the table

if (hashTable[index].key == -1) { printf("Key not found.\n"); return -1;

}

}

return hashTable[index].data;

}

// Function to display the contents of the hash table void display(struct HashEntry\* hashTable) {

printf("Hash Table:\n");

for (int i = 0; i < SIZE; i++) { if (hashTable[i].key != -1) {

printf("Key: %d, Value: %d\n", hashTable[i].key, hashTable[i].data);

}

}

}

int main() {

struct HashEntry hashTable[SIZE]; initialize(hashTable);

// Insert key-value pairs into the hash table insert(hashTable, 10, 100);

insert(hashTable, 20, 200);

insert(hashTable, 30, 300);

insert(hashTable, 40, 400);

insert(hashTable, 50, 500);

insert(hashTable, 12, 120);

insert(hashTable, 22, 220);

insert(hashTable, 32, 320);

// Display the contents of the hash table display(hashTable);

// Retrieve values by key int keyToFind = 30;

int result = get(hashTable, keyToFind); if (result != -1) {

printf("Value for key %d: %d\n", keyToFind, result);

}

return 0;

}

*Time Complexity Analysis :*

1. Insertion and Retrieval :

* In the average case, the time complexity for insertion and retrieval is O(1) because, on average, the linear probing operation finds an empty slot quickly.
* In the worst case (when the hash table is nearly full), the time complexity can approach O(n) because linear probing may require scanning nearly the entire table.

1. Displaying the Hash Table :

* The time complexity for displaying the hash table is O(n), where "n" is the size of the hash table.

Overall, the performance of this hash table implementation with linear probing depends on the distribution of keys and the load factor of the table. If the load factor is kept low, the average-case performance remains close to O(1). However, in the worst-case scenario with high load and clustering of keys, the performance can degrade to O(n).

1. ***>> C program that implements binary search and linear search in arrays :***

#include <stdio.h>

// Linear search function

int linearSearch(int arr[], int size, int target) { for (int i = 0; i < size; i++) {

if (arr[i] == target) {

return i; // Return the index where the target element is found

}

}

return -1; // Return -1 if the target element is not found

}

// Binary search function (assuming the array is sorted) int binarySearch(int arr[], int size, int target) {

int left = 0;

int right = size - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target) {

return mid; // Return the index where the target element is found

} else if (arr[mid] < target) {

left = mid + 1; // Target is in the right half

} else {

right = mid - 1; // Target is in the left half

}

}

return -1; // Return -1 if the target element is not found

}

int main() {

int arr[] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

int size = sizeof(arr) / sizeof(arr[0]); int target = 12;

// Linear search

int linearResult = linearSearch(arr, size, target); if (linearResult != -1) {

printf("Linear Search: Element %d found at index %d.\n", target, linearResult);

} else {

printf("Linear Search: Element %d not found.\n", target);

}

// Binary search (requires a sorted array)

int binaryResult = binarySearch(arr, size, target); if (binaryResult != -1) {

printf("Binary Search: Element %d found at index %d.\n", target, binaryResult);

} else {

printf("Binary Search: Element %d not found.\n", target);

}

return 0;

}

*Time Complexity Analysis :*

1. Linear Search :

* In the worst case, the time complexity of linear search is O(n), where "n" is the size of the array. This is because it may need to examine all elements in the array to find the target.

1. Binary Search :

* In the worst case, the time complexity of binary search is O(log n), where "n" is the size of the sorted array. Binary search repeatedly divides the search interval in half, leading to a logarithmic time complexity.

Binary search is significantly faster than linear search for large arrays, especially when the array is sorted. However, binary search requires the array to be sorted, whereas linear search does not have this requirement. The choice between these search algorithms depends on the specific problem and the characteristics of the data.

1. ***>> C program that implements a stack and a queue using arrays :***

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

#define MAX\_SIZE 100

// Stack implementation using an array struct Stack {

int arr[MAX\_SIZE]; int top;

};

// Initialize the stack

void initializeStack(struct Stack\* stack) { stack->top = -1;

}

// Check if the stack is empty

int isStackEmpty(struct Stack\* stack) { return (stack->top == -1);

}

// Check if the stack is full

int isStackFull(struct Stack\* stack) { return (stack->top == MAX\_SIZE - 1);

}

// Push an element onto the stack

void push(struct Stack\* stack, int value) { if (isStackFull(stack)) {

printf("Stack is full. Cannot push.\n"); return;

}

stack->arr[++(stack->top)] = value;

}

// Pop an element from the stack int pop(struct Stack\* stack) {

if (isStackEmpty(stack)) {

printf("Stack is empty. Cannot pop.\n");

return -1; // A sentinel value to indicate an empty stack

}

return stack->arr[(stack->top)--];

}

// Queue implementation using an array struct Queue {

int arr[MAX\_SIZE]; int front, rear;

};

// Initialize the queue

void initializeQueue(struct Queue\* queue) { queue->front = -1;

queue->rear = -1;

}

// Check if the queue is empty

int isQueueEmpty(struct Queue\* queue) { return (queue->front == -1);

}

// Check if the queue is full

int isQueueFull(struct Queue\* queue) {

return ((queue->rear + 1) % MAX\_SIZE == queue->front);

}

// Enqueue (insert) an element into the queue void enqueue(struct Queue\* queue, int value) {

if (isQueueFull(queue)) {

printf("Queue is full. Cannot enqueue.\n"); return;

}

if (isQueueEmpty(queue)) { queue->front = 0;

queue->rear = 0;

} else {

queue->rear = (queue->rear + 1) % MAX\_SIZE;

}

queue->arr[queue->rear] = value;

}

// Dequeue (remove) an element from the queue int dequeue(struct Queue\* queue) {

if (isQueueEmpty(queue)) {

printf("Queue is empty. Cannot dequeue.\n");

return -1; // A sentinel value to indicate an empty queue

}

int removedValue = queue->arr[queue->front]; if (queue->front == queue->rear) {

queue->front = -1;

queue->rear = -1;

} else {

queue->front = (queue->front + 1) % MAX\_SIZE;

}

return removedValue;

}

int main() {

struct Stack stack; initializeStack(&stack);

struct Queue queue; initializeQueue(&queue);

// Push elements onto the stack push(&stack, 10);

push(&stack, 20);

push(&stack, 30);

// Pop elements from the stack

printf("Popped from stack: %d\n", pop(&stack)); printf("Popped from stack: %d\n", pop(&stack));

// Enqueue elements into the queue enqueue(&queue, 100);

enqueue(&queue, 200);

enqueue(&queue, 300);

// Dequeue elements from the queue

printf("Dequeued from queue: %d\n", dequeue(&queue)); printf("Dequeued from queue: %d\n", dequeue(&queue));

return 0;

}

*Time Complexity Analysis :*

1. Stack Operations :

* Push : O(1)
* Pop : O(1)
* Checking if the stack is empty or full : O(1)

1. Queue Operations :

* Enqueue : O(1)
* Dequeue : O(1)
* Checking if the queue is empty or full : O(1)

Both the stack and queue implementations using arrays provide constant time complexity for their fundamental operations, making them efficient for typical use cases. However, the arrays have a fixed maximum size (in this example, `MAX\_SIZE`), which limits the number of elements they can hold.

1. ***>> C program that implements a circular queue using arrays :***

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

#define MAX\_SIZE 5

// Structure for the circular queue struct CircularQueue {

int data[MAX\_SIZE]; int front, rear;

};

// Initialize the circular queue

void initialize(struct CircularQueue\* queue) { queue->front = -1;

queue->rear = -1;

}

// Check if the queue is empty

int isEmpty(struct CircularQueue\* queue) {

return (queue->front == -1 && queue->rear == -1);

}

// Check if the queue is full

int isFull(struct CircularQueue\* queue) {

return ((queue->rear + 1) % MAX\_SIZE == queue->front);

}

// Enqueue (insert) an element into the circular queue void enqueue(struct CircularQueue\* queue, int value) {

if (isFull(queue)) {

printf("Queue is full. Cannot enqueue.\n"); return;

}

if (isEmpty(queue)) { queue->front = 0;

queue->rear = 0;

} else {

queue->rear = (queue->rear + 1) % MAX\_SIZE;

}

queue->data[queue->rear] = value;

}

// Dequeue (remove) an element from the circular queue int dequeue(struct CircularQueue\* queue) {

if (isEmpty(queue)) {

printf("Queue is empty. Cannot dequeue.\n");

return -1; // A sentinel value to indicate an empty queue

}

int removedValue = queue->data[queue->front]; if (queue->front == queue->rear) {

queue->front = -1;

queue->rear = -1;

} else {

queue->front = (queue->front + 1) % MAX\_SIZE;

}

return removedValue;

}

// Display the elements of the circular queue

void display(struct CircularQueue\* queue) { if (isEmpty(queue)) {

printf("Queue is empty.\n"); return;

}

int i = queue->front; printf("Circular Queue: "); do {

printf("%d ", queue->data[i]); i = (i + 1) % MAX\_SIZE;

} while (i != (queue->rear + 1) % MAX\_SIZE); printf("\n");

}

int main() {

struct CircularQueue queue; initialize(&queue);

enqueue(&queue, 10);

enqueue(&queue, 20);

enqueue(&queue, 30);

enqueue(&queue, 40);

enqueue(&queue, 50); display(&queue);

printf("Dequeued: %d\n", dequeue(&queue)); printf("Dequeued: %d\n", dequeue(&queue));

enqueue(&queue, 60);

enqueue(&queue, 70); display(&queue);

return 0;

}

*Time Complexity Analysis :*

1. Enqueue (Insertion) :

* In the average case, the time complexity for enqueuing an element is O(1) because it involves simple arithmetic operations.
* In the worst case (when the queue is almost full), the time complexity for enqueuing is still O(1), as the circular nature of the queue ensures that we can always find the next available slot with constant time operations.

1. Dequeue (Removal) :

* In the average case, the time complexity for dequeuing an element is O(1) for the same reasons as enqueuing.
* In the worst case (when the queue is almost empty), the time complexity for dequeuing is still O(1).

1. Checking if the queue is empty or full :

* These operations have constant time complexity, O(1).

The circular queue implementation using arrays provides constant time complexity for its basic operations, making it efficient for typical use cases. The circular nature of the queue ensures that enqueuing and dequeuing operations remain efficient even in the presence of many elements in the queue.

1. ***>> C program that implements a binary search tree (BST) and performs inorder, preorder, and postorder traversals :***

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

// Structure for a BST node struct TreeNode {

int data;

struct TreeNode\* left; struct TreeNode\* right;

};

// Function to create a new BST node struct TreeNode\* createNode(int data) {

struct TreeNode\* newNode = (struct TreeNode\*)malloc(sizeof(struct TreeNode)); if (newNode == NULL) {

printf("Memory allocation failed.\n"); exit(1);

}

newNode->data = data; newNode->left = NULL; newNode->right = NULL; return newNode;

}

// Function to insert a value into the BST

struct TreeNode\* insert(struct TreeNode\* root, int data) { if (root == NULL) {

return createNode(data);

}

if (data < root->data) {

root->left = insert(root->left, data);

} else if (data > root->data) {

root->right = insert(root->right, data);

}

return root;

}

// Function to perform inorder traversal of the BST void inorderTraversal(struct TreeNode\* root) {

if (root == NULL) { return;

}

inorderTraversal(root->left); printf("%d ", root->data); inorderTraversal(root->right);

}

// Function to perform preorder traversal of the BST void preorderTraversal(struct TreeNode\* root) {

if (root == NULL) { return;

}

printf("%d ", root->data); preorderTraversal(root->left); preorderTraversal(root->right);

}

// Function to perform postorder traversal of the BST void postorderTraversal(struct TreeNode\* root) {

if (root == NULL) { return;

}

postorderTraversal(root->left); postorderTraversal(root->right); printf("%d ", root->data);

}

int main() {

struct TreeNode\* root = NULL;

// Insert elements into the BST root = insert(root, 50); insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

// Perform inorder traversal printf("Inorder Traversal: "); inorderTraversal(root); printf("\n");

// Perform preorder traversal printf("Preorder Traversal: "); preorderTraversal(root); printf("\n");

// Perform postorder traversal printf("Postorder Traversal: "); postorderTraversal(root); printf("\n");

return 0;

}

*Time Complexity Analysis :*

* Insertion into the BST has a time complexity of O(h), where "h" is the height of the tree. In the worst case (unbalanced tree), the height "h" can be O(n), making insertion O(n).
* Inorder, preorder, and postorder traversals all have a time complexity of O(n) because they visit every node in the BST exactly once, where "n" is the number of nodes in the tree.

Note that the time complexity of these operations can vary depending on the structure of the tree. In the best case (balanced tree), the height "h" is O(log n), making all operations O(log n).

1. *>>* ***C program that implements both Depth-First Search (DFS) and Breadth-First Search (BFS) for a graph represented as an adjacency matrix :***

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

#define MAX\_NODES 100

// Structure for a queue (used in BFS) struct Queue {

int items[MAX\_NODES]; int front;

int rear;

};

// Structure for a stack (used in DFS) struct Stack {

int items[MAX\_NODES]; int top;

};

// Initialize the queue

void initializeQueue(struct Queue\* q) { q->front = -1;

q->rear = -1;

}

// Check if the queue is empty

int isQueueEmpty(struct Queue\* q) { return (q->front == -1);

}

// Enqueue an item into the queue

void enqueue(struct Queue\* q, int value) { if (q->rear == MAX\_NODES - 1) {

printf("Queue is full. Cannot enqueue.\n"); return;

}

if (q->front == -1) { q->front = 0;

}

q->rear++;

q->items[q->rear] = value;

}

// Dequeue an item from the queue int dequeue(struct Queue\* q) {

if (isQueueEmpty(q)) {

printf("Queue is empty. Cannot dequeue.\n");

return -1; // A sentinel value to indicate an empty queue

}

int removedValue = q->items[q->front]; if (q->front == q->rear) {

q->front = -1;

q->rear = -1;

} else {

q->front++;

}

return removedValue;

}

// Initialize the stack

void initializeStack(struct Stack\* s) { s->top = -1;

}

// Check if the stack is empty

int isStackEmpty(struct Stack\* s) { return (s->top == -1);

}

// Push an item onto the stack

void push(struct Stack\* s, int value) { if (s->top == MAX\_NODES - 1) {

printf("Stack is full. Cannot push.\n"); return;

}

s->items[++(s->top)] = value;

}

// Pop an item from the stack int pop(struct Stack\* s) {

if (isStackEmpty(s)) {

printf("Stack is empty. Cannot pop.\n");

return -1; // A sentinel value to indicate an empty stack

}

return s->items[(s->top)--];

}

// Perform Depth-First Search (DFS) for a graph represented as an adjacency matrix void dfs(int graph[MAX\_NODES][MAX\_NODES], int numNodes, int startNode) {

int visited[MAX\_NODES] = {0}; // Initialize all nodes as unvisited struct Stack stack;

initializeStack(&stack);

printf("DFS traversal starting from node %d: ", startNode);

push(&stack, startNode); visited[startNode] = 1;

while (!isStackEmpty(&stack)) { int currentNode = pop(&stack); printf("%d ", currentNode);

for (int i = 0; i < numNodes; i++) {

if (graph[currentNode][i] == 1 && !visited[i]) { push(&stack, i);

visited[i] = 1;

}

}

}

printf("\n");

}

// Perform Breadth-First Search (BFS) for a graph represented as an adjacency matrix void bfs(int graph[MAX\_NODES][MAX\_NODES], int numNodes, int startNode) {

int visited[MAX\_NODES] = {0}; // Initialize all nodes as unvisited struct Queue queue;

initializeQueue(&queue);

printf("BFS traversal starting from node %d: ", startNode); enqueue(&queue, startNode);

visited[startNode] = 1;

while (!isQueueEmpty(&queue)) {

int currentNode = dequeue(&queue); printf("%d ", currentNode);

for (int i = 0; i < numNodes; i++) {

if (graph[currentNode][i] == 1 && !visited[i]) { enqueue(&queue, i);

visited[i] = 1;

}

}

}

printf("\n");

}

int main() {

int numNodes, startNode;

int graph[MAX\_NODES][MAX\_NODES]; printf("Enter the number of nodes: ");

scanf("%d", &numNodes);

printf("Enter the adjacency matrix:\n"); for (int i = 0; i < numNodes; i++) {

for (int j = 0; j < numNodes; j++) { scanf("%d", &graph[i][j]);

}

}

printf("Enter the starting node for DFS and BFS: "); scanf("%d", &startNode);

dfs(graph, numNodes, startNode); bfs(graph, numNodes, startNode);

return 0;

}

*Time Complexity Analysis :*

* Both DFS and BFS visit each node in the graph once, so their time complexity is O(V + E), where "V" is the number of vertices (nodes) and "E" is the number of edges in the graph.

1. **>> C programs that implement selection sort, merge sort, insertion sort, bubble sort, and quick sort for arrays :**

*A >> Selection Sort :*

#include <stdio.h>

void selectionSort(int arr[], int size) { for (int i = 0; i < size - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < size; j++) { if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

if (minIndex != i) { int temp = arr[i];

arr[i] = arr[minIndex]; arr[minIndex] = temp;

}

}

}

int main() {

int arr[] = {64, 25, 12, 22, 11};

int size = sizeof(arr) / sizeof(arr[0]); selectionSort(arr, size);

printf("Selection Sort: "); for (int i = 0; i < size; i++) {

printf("%d ", arr[i]);

}

return 0;

}

*Time Complexity of Selection Sort :* O(n^2)

*B >> Merge Sort :*

#include <stdio.h>

void merge(int arr[], int left, int mid, int right) { int n1 = mid - left + 1;

int n2 = right - mid;

int leftArr[n1], rightArr[n2]; for (int i = 0; i < n1; i++) {

leftArr[i] = arr[left + i];

}

for (int i = 0; i < n2; i++) { rightArr[i] = arr[mid + 1 + i];

}

int i = 0, j = 0, k = left; while (i < n1 && j < n2) {

if (leftArr[i] <= rightArr[j]) { arr[k++] = leftArr[i++];

} else {

arr[k++] = rightArr[j++];

}

}

while (i < n1) {

arr[k++] = leftArr[i++];

}

while (j < n2) {

arr[k++] = rightArr[j++];

}

}

void mergeSort(int arr[], int left, int right) { if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid); mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int size = sizeof(arr) / sizeof(arr[0]); mergeSort(arr, 0, size - 1);

printf("Merge Sort: ");

for (int i = 0; i < size; i++) { printf("%d ", arr[i]);

}

return 0;

}

*Time Complexity of Merge Sort :* O(n log n)

*C >> Insertion Sort :*

#include <stdio.h>

void insertionSort(int arr[], int size) { for (int i = 1; i < size; i++) {

int key = arr[i]; int j = i - 1;

while (j >= 0 && arr[j] > key) { arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int size = sizeof(arr) / sizeof(arr[0]); insertionSort(arr, size);

printf("Insertion Sort: "); for (int i = 0; i < size; i++) {

printf("%d ", arr[i]);

}

return 0;

}

*Time Complexity of Insertion Sort :* O(n^2)

*D >> Bubble Sort :*

#include <stdio.h>

void bubbleSort(int arr[], int size) { for (int i = 0; i < size - 1; i++) {

for (int j = 0; j < size - i - 1; j++) { if (arr[j] > arr[j + 1]) {

int temp = arr[j]; arr[j] = arr[j + 1]; arr[j + 1] = temp;

}

}

}

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int size = sizeof(arr) / sizeof(arr[0]); bubbleSort(arr, size);

printf("Bubble Sort: ");

for (int i = 0; i < size; i++) { printf("%d ", arr[i]);

}

return 0;

}

*Time Complexity of Bubble Sort :* O(n^2)

*E >> Quick Sort :*

#include <stdio.h>

void swap(int\* a, int\* b) { int temp = \*a;

\*a = \*b;

\*b = temp;

}

int partition(int arr[], int low, int high) { int pivot = arr[high];

int i = low - 1;

for (int j = low; j <= high - 1; j++) { if (arr[j] < pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]); return (i + 1);

}

void quickSort(int arr[], int low, int high) { if (low < high) {

int pivotIndex = partition(arr, low, high);

quickSort(arr, low, pivotIndex - 1); quickSort(arr, pivotIndex + 1, high);

}

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int size = sizeof(arr) / sizeof(arr[0]);

quickSort(arr, 0, size - 1); printf("Quick Sort: ");

for (int i = 0; i < size; i++) { printf("%d ", arr[i]);

}

return 0;

}

*Time Complexity of Quick Sort :* O(n^2)

Note : The time complexities mentioned above are for worst-case scenarios. Some sorting algorithms may perform better in practice, depending on the initial order of elements.

1. **>> C programs to implement Counting Sort and Radix Sort using arrays :** *A >> Counting Sort :*

#include <stdio.h>

void countingSort(int arr[], int size) { int max = arr[0];

for (int i = 1; i < size; i++) { if (arr[i] > max) {

max = arr[i];

}

}

int count[max + 1];

for (int i = 0; i <= max; i++) { count[i] = 0;

}

for (int i = 0; i < size; i++) {

count[arr[i]]++;

}

int output[size];

int outputIndex = 0;

for (int i = 0; i <= max; i++) { while (count[i] > 0) {

output[outputIndex] = i; outputIndex++;

count[i]--;

}

}

for (int i = 0; i < size; i++) { arr[i] = output[i];

}

}

int main() {

int arr[] = {4, 2, 2, 8, 3, 3, 1};

int size = sizeof(arr) / sizeof(arr[0]); countingSort(arr, size);

printf("Counting Sort: "); for (int i = 0; i < size; i++) {

printf("%d ", arr[i]);

}

return 0;

}

*B >> Radix Sort :*

#include <stdio.h>

// Function to find the maximum element in the array int findMax(int arr[], int size) {

int max = arr[0];

for (int i = 1; i < size; i++) { if (arr[i] > max) {

max = arr[i];

}

}

return max;

}

// Function to perform counting sort based on a specific digit's place (e.g., one's place, ten's place) void countingSortByDigit(int arr[], int size, int exp) {

int output[size];

int count[10] = {0};

for (int i = 0; i < size; i++) { count[(arr[i] / exp) % 10]++;

}

for (int i = 1; i < 10; i++) { count[i] += count[i - 1];

}

for (int i = size - 1; i >= 0; i--) { output[count[(arr[i] / exp) % 10] - 1] = arr[i]; count[(arr[i] / exp) % 10]--;

}

for (int i = 0; i < size; i++) { arr[i] = output[i];

}

}

// Function to perform Radix Sort void radixSort(int arr[], int size) { int max = findMax(arr, size);

for (int exp = 1; max / exp > 0; exp \*= 10) { countingSortByDigit(arr, size, exp);

}

}

int main() {

int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};

int size = sizeof(arr) / sizeof(arr[0]); radixSort(arr, size);

printf("Radix Sort: ");

for (int i = 0; i < size; i++) { printf("%d ", arr[i]);

}

return 0;

}

*Time Complexity Analysis :*

* Counting Sort : O(n + k), where "n" is the number of elements in the array, and "k" is the range of input values.
* Radix Sort : O(d \* (n + k)), where "n" is the number of elements, "k" is the range of input values, and "d" is the number of digits in the maximum number (typically log\_k(max)). Radix Sort is efficient when "d" is small compared to "n".

1. **>> C program to detect and remove loops in a linked list and print the linked list in reverse order using Floyd’s cycle detection algorithm** *:*

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

// Structure for a singly linked list node struct Node {

int data;

struct Node\* next;

};

// Function to create a new linked list node struct Node\* createNode(int data) {

struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node)); if (newNode == NULL) {

printf("Memory allocation failed.\n"); exit(1);

}

newNode->data = data; newNode->next = NULL; return newNode;

}

// Function to detect and remove loops in a linked list void detectAndRemoveLoop(struct Node\* head) {

struct Node\* slow = head; struct Node\* fast = head;

// Detect the loop using Floyd's Cycle Detection Algorithm while (slow && fast && fast->next) {

slow = slow->next;

fast = fast->next->next;

if (slow == fast) {

// Loop detected break;

}

}

if (slow == fast) {

// Loop exists, remove it slow = head;

while (slow->next != fast->next) { slow = slow->next;

fast = fast->next;

}

fast->next = NULL;

}

}

// Function to reverse a linked list

struct Node\* reverseLinkedList(struct Node\* head) { struct Node\* prev = NULL;

struct Node\* current = head; struct Node\* next = NULL;

while (current != NULL) { next = current->next; current->next = prev; prev = current;

current = next;

}

return prev; // New head of the reversed list

}

// Function to print a linked list

void printLinkedList(struct Node\* head) { struct Node\* current = head;

while (current != NULL) { printf("%d -> ", current->data); current = current->next;

}

printf("NULL\n");

}

int main() {

struct Node\* head = createNode(1); head->next = createNode(2);

head->next->next = createNode(3);

head->next->next->next = createNode(4);

head->next->next->next->next = createNode(5);

// Create a loop for testing

head->next->next->next->next->next = head->next->next; detectAndRemoveLoop(head);

printf("Linked List without loop: "); printLinkedList(head);

// Reverse the linked list

head = reverseLinkedList(head); printf("Linked List in reverse order: "); printLinkedList(head);

return 0;

}

*Time Complexity :*

* Detecting and removing loops in a linked list using Floyd's algorithm takes O(n) time, where "n" is the number of nodes in the list.
* Reversing a linked list also takes O(n) time as it iterates through all the nodes once.