Khwopa College of Engineering

DSA Lab Sheet

II Year/ II Part

Faculty: Computer and Electronics

Labsheet #6

Objectives:

1. Implementation of static list.
2. Implementation of linked list: Singly and Doubly linked list.

Theory:

1. Static List (Array Implementation)

- Static List uses fixed-size arrays.
- Supports:
  - Insertion
  - Deletion
  - Traversal

Source Code:

```
#include <stdio.h>
#define SIZE 100
int list[SIZE];
int n = 0;
void insert(int pos, int value) {
    if (pos < 0 || pos > n || n == SIZE) {
        printf("Invalid position or overflow.\n");
        return;
    }
    for (int i = n; i > pos; i--) {
        list[i] = list[i - 1];
    }
    list[pos] = value;
    n++;
}
void delete(int pos) {
    if (pos < 0 || pos >= n) {
        printf("Invalid position.\n");
        return;
    }
    for (int i = pos; i < n - 1; i++) {
        list[i] = list[i + 1];
    }
    n--;
}
void display() {
    printf("List: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", list[i]);
    }
    printf("\n");
}
```

Task:

- Create an array-based list.
- Insert and delete at beginning, end, and specific position.
- **Display the list.**

2. Singly Linked List

- Each node has `data` and a pointer to the next node.
- Operations include insertions, deletions, and traversal.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* head = NULL;
void insertAtEnd(int value) {
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next)
            temp = temp->next;
        temp->next = newNode;
    }
}
void deleteAtBeginning() {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* temp = head;
    head = head->next;
    free(temp);
}
void displayList() {
    struct Node* temp = head;
    printf("Singly Linked List: ");
    while (temp) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

```
int main() {
    insertAtEnd(10);
    insertAtEnd(20);
    insertAtEnd(30);
    displayList();
    return 0;
}
```

Tasks

- Create a list with 5 elements.
- Perform insertions and deletions at various positions.
- Display list content.

3. Doubly Linked List

- Each node has data, prev, and next.
- Can be traversed in both forward and backward directions.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
struct DNode {
    int data;
    struct DNode* prev;
    struct DNode* next;
};
struct DNode* head = NULL;
void insertAtEnd(int value) {
    struct DNode* newNode = malloc(sizeof(struct DNode));
    newNode->data = value;
    newNode->next = NULL;
    newNode->prev = NULL;
    if (head == NULL) {
        head = newNode;
    } else {
        struct DNode* temp = head;
        while (temp->next)
            temp = temp->next;
        temp->next = newNode;
        newNode->prev = temp;
    }
}
void displayForward() {
    struct DNode* temp = head;
    printf("Forward: ");
    while (temp) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
void displayBackward() {
    struct DNode* temp = head;
    if (!temp) return;
    while (temp->next)
        temp = temp->next;
    printf("Backward: ");
    while (temp) {
        printf("%d <-> ", temp->data);
        temp = temp->prev;
    }
    printf("NULL\n");
}
```

Tasks

- Insert nodes at beginning and end.
- Display the list in both directions.
- Perform deletions from both ends.

4. Priority Queue (using Array or Linked List)

- A **Priority Queue** serves elements based on priority.
- Higher priority elements are dequeued before lower ones.
- Can be implemented using:
  - Array (with sorting)
  - Linked List (insertion in sorted order)

Structure:

```
struct Node {
     int data;
     int priority;
     struct Node* next;
};
```

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    int priority;
    struct Node* next;
};
struct Node* front = NULL;
void enqueue(int value, int priority) {
    struct Node* newNode = malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->priority = priority;
    newNode->next = NULL;
    if (front == NULL || priority < front->priority) {
        newNode->next = front;
        front = newNode;
    } else {
        struct Node* temp = front;
        while (temp->next && temp->next->priority <= priority)
            temp = temp->next;
        newNode->next = temp->next;
        temp->next = newNode;
    }
}
void dequeue() {
    if (front == NULL) {
        printf("Queue is empty.\n");
        return;
    }
    struct Node* temp = front;
    printf("Dequeued: %d (priority %d)\n", temp->data, temp->priority);
    front = front->next;
    free(temp);
}
void displayQueue() {
    struct Node* temp = front;
    printf("Priority Queue: ");
    while (temp) {
        printf("[%d|P:%d] -> ", temp->data, temp->priority);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

Tasks:

1. Insert elements with different priorities.
2. Dequeue (remove) the element with highest priority.
3. Display the queue based on priority order.