Teaching Notes on Linked List Applications & Advanced Operations

4. Applications of Linked Lists

4.1 Stack using Linked List

Stack follows LIFO (Last In First Out) principle. Operations:

- push(x): Insert element at the beginning (top)
- pop(): Delete element from the beginning (top)

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

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

struct Node* top = NULL;

void push(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = top;
    top = newNode;
    printf("%d pushed to stack\n", value);
}

void pop() {
    if (top == NULL) {
        printf("Stack Underflow\n");
        return;
    }
    struct Node* temp = top;
    printf("%d popped from stack\n", top->data);
    top = top->next;
    free(temp);
}

void display() {
    struct Node* temp = top;
    printf("Stack: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    push(10);
    push(20);
    push(30);
    display();
    pop();
    display();
    return 0;
}
```

4.2 Queue using Linked List

Queue follows FIFO (First In First Out) principle. Operations:

- enqueue(x): Insert element at the end (rear)
- dequeue(): Delete element from the beginning (front)

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

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

struct Node* front = NULL;
struct Node* rear = NULL;

void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
    printf("%d enqueued to queue\n", value);
}

void dequeue() {
    if (front == NULL) {
        printf("Queue Underflow\n");
        return;
    }
    struct Node* temp = front;
    printf("%d dequeued from queue\n", front->data);
    front = front->next;
    if (front == NULL) rear = NULL;
    free(temp);
}

void display() {
    struct Node* temp = front;
    printf("Queue: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    dequeue();
    display();
    return 0;
}
```

5. Advanced Operation

5.1 Reversing a Singly Linked List

Reversal of a linked list is done using three pointers: prev, curr, next.

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

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

struct Node* insertEnd(struct Node* head, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (head == NULL) return newNode;
    struct Node* temp = head;
    while (temp->next != NULL) temp = temp->next;
    temp->next = newNode;
    return head;
}

struct Node* reverse(struct Node* head) {
    struct Node* prev = NULL;
    struct Node* curr = head;
    struct Node* next = NULL;
    while (curr != NULL) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

void display(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;
    head = insertEnd(head, 10);
    head = insertEnd(head, 20);
    head = insertEnd(head, 30);
    printf("Original List: ");
    display(head);
    head = reverse(head);
    printf("Reversed List: ");
    display(head);
    return 0;
}
```