DSA

**1. Queue (FIFO - First-In-First-Out):**

* **Concept:** A queue follows a strict order where elements are added at the back (enqueue) and removed from the front (dequeue). It's like a waiting line where the first person in line gets served first.

There are four main types of queues commonly used in computer science:

1. **Simple Queue (Linear Queue):**
   * This is the most basic type of queue, following the First-In-First-Out (FIFO) principle.
   * Elements are added at the back (enqueue) and removed from the front (dequeue).
   * Imagine a line of people waiting for service; the first person in line gets served first.
2. **Circular Queue:**
   * Similar to a simple queue, but with a circular structure.
   * The last element in the queue points back to the first, creating a loop.
   * This allows for more efficient memory usage when the queue size is fixed.
   * Applications include buffer management and implementing round-robin scheduling algorithms.
3. **Priority Queue:**
   * Unlike simple queues, elements in a priority queue have an associated priority value.
   * Elements with higher priority are removed first, regardless of their insertion order.
   * This is useful for tasks with varying importance levels, like processing urgent jobs first.
4. **Double-Ended Queue (Deque):**
   * A deque (pronounced "deck") is more flexible than a simple queue.
   * Elements can be added and removed from both the front and the back (end) of the queue.
   * This allows for operations like reversing a sequence or implementing a browser's back/forward functionality.

#include <stdio.h>

#include <stdlib.h>

typedef struct QueueNode {

int data;

struct QueueNode\* next;

} QueueNode;

typedef struct Queue {

QueueNode\* front, \*rear;

} Queue;

void initializeQueue(Queue\* q) {

q->front = q->rear = NULL;

}

int isEmpty(Queue\* q) {

return q->front == NULL;

}

int isFull(Queue\* q) { // Not applicable for linked list implementation

return 0; // Always able to allocate a new node

}

void enqueue(Queue\* q, int data) {

QueueNode\* newNode = (QueueNode\*)malloc(sizeof(QueueNode));

newNode->data = data;

newNode->next = NULL;

if (isEmpty(q)) {

q->front = q->rear = newNode;

} else {

q->rear->next = newNode;

q->rear = newNode;

}

}

int dequeue(Queue\* q) {

if (isEmpty(q)) {

printf("Queue underflow\n");

return -1;

}

QueueNode\* temp = q->front;

int data = temp->data;

q->front = q->front->next;

if (q->front == NULL) { // Handle case of removing last element

q->rear = NULL;

}

free(temp);

return data;

}

**2. Stack (LIFO - Last-In-First-Out):**

* **Concept:** A stack functions like a stack of plates, where elements are added (push) and removed (pop) from the top. The last element added is the first one to be retrieved.

#include <stdio.h>

#include <stdlib.h>

typedef struct StackNode {

int data;

struct StackNode\* next;

} StackNode;

typedef struct Stack {

StackNode\* top;

} Stack;

void initializeStack(Stack\* s) {

s->top = NULL;

}

int isEmptyStack(Stack\* s) {

return s->top == NULL;

}

int isFullStack(Stack\* s) { // Not applicable for linked list implementation

return 0; // Always able to allocate a new node

}

void push(Stack\* s, int data) {

StackNode\* newNode = (StackNode\*)malloc(sizeof(StackNode));

newNode->data = data;

newNode->next = s->top;

s->top = newNode;

}

int pop(Stack\* s) {

if (isEmptyStack(s)) {

printf("Stack underflow\n");

return -1;

}

StackNode\* temp = s->top;

int data = temp->data;

s->top = s->top->next;

free(temp);

return data;

}

**3. Singly Linked List:**

* **Concept:** A linear data structure where elements (nodes) are connected using pointers. Each node contains data and a reference (next) to the next node in the sequence.

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* next;

} Node;

void initializeList(Node\*\* head) {

\*head = NULL;

}

void insertAtBeginning(Node\*\* head, int data) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->data = data;

newNode->next = \*head;

\*head = newNode;

}

void insertAtEnd(Node\*\* head, int data) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->data = data;

newNode->next = NULL;

if (\*head == NULL) {

\*head = newNode;

return;

}

Node\* last = \*head;

while (last->next != NULL) {

last = last->next;

}

last->next = newNode;

}

// Additional functions for traversal, deletion, etc. can be implemented

**4. Doubly Linked List:**

* **Concept:** Similar to a singly linked list, but each node has two pointers: one to the next node and one to the previous node. This allows for efficient navigation in both directions.

#include <stdio.h>

#include <stdlib.h>

typedef struct DNode {

int data;

struct DNode\* prev;

struct DNode\* next;

} DNode;

typedef struct DoublyLinkedList {

DNode\* head;

DNode\* tail;

} DoublyLinkedList;

void initializeDoublyList(DoublyLinkedList\* list) {

list->head = list->tail = NULL;

**5. Circular Linked List:**

* **Concept:** A variation of a singly linked list where the last node's next pointer points back to the first node, creating a circular structure.

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

int data;

struct Node\* next;

} Node;

void initializeList(Node\*\* head) {

\*head = NULL;

}

// Insert at the beginning (can be modified for insertion at any position)

void insertAtBeginning(Node\*\* head, int data) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->data = data;

if (\*head == NULL) {

newNode->next = newNode; // Link to itself for the first node

\*head = newNode;

return;

}

// Find the last node

Node\* last = \*head;

while (last->next != \*head) {

last = last->next;

}

// Insert at the beginning and update last node's next

newNode->next = \*head;

last->next = newNode;

\*head = newNode;

}

void printList(Node\* head) {

if (head == NULL) {

printf("List is empty\n");

return;

}

Node\* temp = head;

do {

printf("%d ", temp->data);

temp = temp->next;

} while (temp != head);

printf("\n");

}

int main() {

Node\* head = NULL;

initializeList(&head);

insertAtBeginning(&head, 10);

insertAtBeginning(&head, 20);

insertAtBeginning(&head, 30);

printf("Circular Linked List: ");

printList(head);

return;

}