1) Byte-Packing Stack.

WTD: Design a stack that efficiently stores 8-bit data in a continuous memory space. Ensure that the 32-bit words are packed without any wastage.
(e.g: I/P: Push 0x01, 0x02, 0x03, 0x04 ; O/P: Memory content - 0x04030201)

```c
#include <stdio.h>
#include <stdint.h>

#define STACK_SIZE 32  // Define the size of the stack in 32-bit words

typedef struct {
    uint32_t memory[STACK_SIZE];
    int SP;  // Stack Pointer
} BytePackingStack;

void initStack(BytePackingStack *stack) {
    for (int i = 0; i < STACK_SIZE; i++) {
        stack->memory[i] = 0;  // Initialize the memory to zeros
    }
    stack->SP = 0;  // Initialize the stack pointer to 0
}

void push(BytePackingStack *stack, uint8_t value) {
    // Ensure that the stack pointer is within bounds
    if (stack->SP < STACK_SIZE) {
        // Calculate the index of the 32-bit word in memory
        int wordIndex = stack->SP / 4;
        // Calculate the position of the 8-bit value within the word
        int byteOffset = (stack->SP % 4) * 8;
        // Pack the 8-bit value into the word using bitwise operations
        stack->memory[wordIndex] |= ((uint32_t)value << byteOffset);
        stack->SP++;  // Increment the stack pointer
    } else {
        printf("Stack overflow!\n");
    }
}

void printMemoryContent(BytePackingStack *stack) {
    printf("Memory content - ");
```

```
    for (int i = STACK_SIZE - 1; i >= 0; i--) {
        printf("%08X", stack->memory[i]);
    }
    printf("\n");
}

int main() {
    BytePackingStack stack;
    initStack(&stack);

    push(&stack, 0x01);
    push(&stack, 0x02);
    push(&stack, 0x03);
    push(&stack, 0x04);

    printMemoryContent(&stack);  // Output: Memory content - 04030201

    return 0;
}
```

2) String Message Queue.

WTD: Implement a queue that specializes in storing and retrieving string messages in a FIFO manner.

(e.g: I/P: Enqueue "HELLO", Enqueue "WORLD" ; O/P: Dequeue - "HELLO", Dequeue - "WORLD")

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

// Define a structure to represent a string message node
typedef struct MessageNode {
    char *message;
    struct MessageNode *next;
} MessageNode;

// Define a structure to represent the message queue
typedef struct {
```

```c
    MessageNode *front;  // Front of the queue
    MessageNode *rear;   // Rear of the queue
} StringMessageQueue;

// Initialize an empty message queue
void initStringMessageQueue(StringMessageQueue *queue) {
    queue->front = NULL;
    queue->rear = NULL;
}

// Enqueue a message into the queue
void enqueue(StringMessageQueue *queue, const char *message) {
    // Create a new message node
    MessageNode *newNode = (MessageNode *)malloc(sizeof(MessageNode));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }

    // Allocate memory for the message and copy it
    newNode->message = (char *)malloc(strlen(message) + 1);
    if (newNode->message == NULL) {
        printf("Memory allocation failed.\n");
        free(newNode);
        exit(1);
    }
    strcpy(newNode->message, message);

    newNode->next = NULL;

    if (queue->rear == NULL) {
        // If the queue is empty, set both front and rear to the new node
        queue->front = newNode;
        queue->rear = newNode;
    } else {
        // Otherwise, add the new node to the rear
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
}
```

```c
// Dequeue a message from the queue
char *dequeue(StringMessageQueue *queue) {
    if (queue->front == NULL) {
        // Queue is empty
        return NULL;
    }

    // Get the message from the front node
    char *message = queue->front->message;

    // Remove the front node and update the front pointer
    MessageNode *temp = queue->front;
    queue->front = queue->front->next;
    free(temp);

    // If the queue is now empty, update the rear pointer
    if (queue->front == NULL) {
        queue->rear = NULL;
    }

    return message;
}

// Free the memory used by the queue
void destroyStringMessageQueue(StringMessageQueue *queue) {
    while (queue->front != NULL) {
        MessageNode *temp = queue->front;
        queue->front = queue->front->next;
        free(temp->message);
        free(temp);
    }
}

int main() {
    StringMessageQueue queue;
    initStringMessageQueue(&queue);

    enqueue(&queue, "HELLO");
    enqueue(&queue, "WORLD");
```

```
    char *message1 = dequeue(&queue);
    char *message2 = dequeue(&queue);

    if (message1 != NULL) {
        printf("Dequeue - \"%s\"\n", message1);
        free(message1);
    }
    if (message2 != NULL) {
        printf("Dequeue - \"%s\"\n", message2);
        free(message2);
    }

    destroyStringMessageQueue(&queue);

    return 0;
}
```

3) Nested Statement Counter.

WTD: Use a stack to simulate the nested structure of programming constructs like loops or if-statements, and return their depth.
(e.g: I/P: { { } } ; O/P: Depth - 2)

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

// Define a stack structure for characters
#define MAX_STACK_SIZE 100

typedef struct {
    char items[MAX_STACK_SIZE];
    int top;
} CharStack;

// Initialize an empty stack
void initStack(CharStack *stack) {
    stack->top = -1;
```

```c
}

// Check if the stack is empty
bool isEmpty(const CharStack *stack) {
    return stack->top == -1;
}

// Push a character onto the stack
void push(CharStack *stack, char c) {
    if (stack->top < MAX_STACK_SIZE - 1) {
        stack->items[++stack->top] = c;
    } else {
        printf("Stack overflow!\n");
    }
}

// Pop a character from the stack
char pop(CharStack *stack) {
    if (!isEmpty(stack)) {
        return stack->items[stack->top--];
    } else {
        printf("Stack underflow!\n");
        return '\0';  // Return a default value (you can handle errors
differently)
    }
}

// Count the depth of nested statements
int countNestedDepth(const char *input) {
    CharStack stack;
    initStack(&stack);
    int depth = 0;

    for (int i = 0; input[i] != '\0'; i++) {
        if (input[i] == '{') {
            push(&stack, input[i]);
        } else if (input[i] == '}') {
            if (!isEmpty(&stack)) {
                pop(&stack);
                depth++;
```

```
            } else {
                printf("Mismatched closing brace at position %d\n", i);
                // Handle error: Unmatched closing brace
            }
        }
    }

    if (!isEmpty(&stack)) {
        printf("Unmatched opening brace(s) in the input\n");
        // Handle error: Unmatched opening brace(s)
    }

    return depth;
}

int main() {
    const char *input = "{{}}";  // Replace with your input string
    int depth = countNestedDepth(input);
    printf("Depth - %d\n", depth);  // Output: Depth - 2

    return 0;
}
```

4) Expression Validator.

WTD: Develop a stack-based mechanism to validate the correctness of arithmetic expressions by checking for balanced parentheses and proper operator placement.
(e.g: I/P: "(a+b) * (c-d)" ; O/P: Valid Expression)

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

// Define a stack structure for characters
#define MAX_STACK_SIZE 100

typedef struct {
    char items[MAX_STACK_SIZE];
    int top;
```

```c
} CharStack;

// Initialize an empty stack
void initStack(CharStack *stack) {
    stack->top = -1;
}

// Check if the stack is empty
bool isEmpty(const CharStack *stack) {
    return stack->top == -1;
}

// Push a character onto the stack
void push(CharStack *stack, char c) {
    if (stack->top < MAX_STACK_SIZE - 1) {
        stack->items[++stack->top] = c;
    } else {
        printf("Stack overflow!\n");
    }
}

// Pop a character from the stack
char pop(CharStack *stack) {
    if (!isEmpty(stack)) {
        return stack->items[stack->top--];
    } else {
        printf("Stack underflow!\n");
        return '\0';  // Return a default value (you can handle errors
differently)
    }
}

// Validate the correctness of an arithmetic expression
bool isValidExpression(const char *expression) {
    CharStack stack;
    initStack(&stack);

    for (int i = 0; expression[i] != '\0'; i++) {
        char c = expression[i];
        if (c == '(') {
```

```c
            push(&stack, c);
        } else if (c == ')') {
            if (isEmpty(&stack) || pop(&stack) != '(') {
                return false; // Unmatched closing parenthesis
            }
        }
    }

    // Check if there are any unmatched opening parentheses
    if (!isEmpty(&stack)) {
        return false;
    }

    return true;  // All parentheses are balanced
}

int main() {
    const char *expression = "(a+b) * (c-d)"; // Replace with your
expression
    bool isValid = isValidExpression(expression);

    if (isValid) {
        printf("Valid Expression\n");
    } else {
        printf("Invalid Expression\n");
    }

    return 0;
}
```

5) Command Parser.

WTD: Develop a command parser using a stack that can handle nested commands. The parser should be able to distinguish between different types of commands and their nesting levels. Implement a mechanism to return the depth of the nested commands for debugging or other purposes.
(e.g: I/P: "{CMD1 {CMD2}}" ; O/P: Depth - 2)

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

```c
#include <stdbool.h>
#include <string.h>

// Define a stack structure for characters
#define MAX_STACK_SIZE 100

typedef struct {
    char items[MAX_STACK_SIZE];
    int top;
} CharStack;

// Initialize an empty stack
void initStack(CharStack *stack) {
    stack->top = -1;
}

// Check if the stack is empty
bool isEmpty(const CharStack *stack) {
    return stack->top == -1;
}

// Push a character onto the stack
void push(CharStack *stack, char c) {
    if (stack->top < MAX_STACK_SIZE - 1) {
        stack->items[++stack->top] = c;
    } else {
        printf("Stack overflow!\n");
    }
}

// Pop a character from the stack
char pop(CharStack *stack) {
    if (!isEmpty(stack)) {
        return stack->items[stack->top--];
    } else {
        printf("Stack underflow!\n");
        return '\0';  // Return a default value (you can handle errors
differently)
    }
}
```

```c
// Parse and count the depth of nested commands
int countNestedDepth(const char *command) {
    CharStack stack;
    initStack(&stack);
    int depth = 0;

    for (int i = 0; command[i] != '\0'; i++) {
        char c = command[i];
        if (c == '{') {
            push(&stack, c);
        } else if (c == '}') {
            if (!isEmpty(&stack)) {
                pop(&stack);
                depth++;
            } else {
                printf("Mismatched closing brace at position %d\n", i);
                // Handle error: Unmatched closing brace
            }
        }
    }

    // Check if there are any unmatched opening braces
    if (!isEmpty(&stack)) {
        printf("Unmatched opening brace(s) in the command\n");
        // Handle error: Unmatched opening brace(s)
    }

    return depth;
}

int main() {
    const char *command = "{CMD1 {CMD2}}"; // Replace with your input
command
    int depth = countNestedDepth(command);
    printf("Depth - %d\n", depth);  // Output: Depth - 2

    return 0;
}
```

6) Palindrome Checker using Stack.

WTD: Create a function that uses a stack to determine whether a given string is a palindrome. Push each character of the string onto a stack and then pop them off to compare with the original string.
(e.g: I/P: "RACECAR" ; O/P: Palindrome)

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h> // Include for character manipulation functions

// Define a stack structure for characters
#define MAX_STACK_SIZE 100

typedef struct {
    char items[MAX_STACK_SIZE];
    int top;
} CharStack;

// Initialize an empty stack
void initStack(CharStack *stack) {
    stack->top = -1;
}

// Check if the stack is empty
bool isEmpty(const CharStack *stack) {
    return stack->top == -1;
}

// Push a character onto the stack
void push(CharStack *stack, char c) {
    if (stack->top < MAX_STACK_SIZE - 1) {
        stack->items[++stack->top] = c;
    } else {
        printf("Stack overflow!\n");
    }
}
```

```c
// Pop a character from the stack
char pop(CharStack *stack) {
    if (!isEmpty(stack)) {
        return stack->items[stack->top--];
    } else {
        printf("Stack underflow!\n");
        return '\0';  // Return a default value (you can handle errors differently)
    }
}

// Check if a string is a palindrome
bool isPalindrome(const char *str) {
    CharStack stack;
    initStack(&stack);

    // Push lowercase characters onto the stack after stripping non-alphanumeric characters
    for (int i = 0; str[i] != '\0'; i++) {
        char c = tolower(str[i]);
        if (isalnum(c)) {
            push(&stack, c);
        }
    }

    // Pop characters from the stack and compare them with the original string
    for (int i = 0; str[i] != '\0'; i++) {
        char c = tolower(str[i]);
        if (isalnum(c)) {
            if (c != pop(&stack)) {
                return false; // Not a palindrome
            }
        }
    }

    // If the stack is empty, it's a palindrome
    return isEmpty(&stack);
}
```

```c
int main() {
    const char *str = "RACECAR"; // Replace with your input string
    bool isPalin = isPalindrome(str);

    if (isPalin) {
        printf("Palindrome\n");
    } else {
        printf("Not a Palindrome\n");
    }

    return 0;
}
```

7) Queue-based Sequence Generator.

WTD: Design a queue-based system that can generate the Fibonacci sequence up to n numbers. The queue should be used to store intermediate Fibonacci numbers and help in generating subsequent numbers in the sequence.
(e.g: I/P: n = 5 ; O/P: 0, 1, 1, 2, 3)

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

// Define a structure for a queue node
typedef struct QueueNode {
    int data;
    struct QueueNode *next;
} QueueNode;

// Define a structure for a queue
typedef struct {
    QueueNode *front;
    QueueNode *rear;
} Queue;

// Initialize an empty queue
void initQueue(Queue *queue) {
    queue->front = NULL;
    queue->rear = NULL;
```

```c
}

// Check if the queue is empty
int isEmpty(Queue *queue) {
    return (queue->front == NULL);
}

// Enqueue a number into the queue
void enqueue(Queue *queue, int data) {
    QueueNode *newNode = (QueueNode *)malloc(sizeof(QueueNode));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;

    if (isEmpty(queue)) {
        queue->front = newNode;
        queue->rear = newNode;
    } else {
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
}

// Dequeue a number from the queue
int dequeue(Queue *queue) {
    if (!isEmpty(queue)) {
        QueueNode *temp = queue->front;
        int data = temp->data;
        queue->front = temp->next;
        free(temp);
        return data;
    } else {
        printf("Queue is empty.\n");
        exit(1);
    }
}
```

```c
// Generate and print the Fibonacci sequence up to n numbers
void generateFibonacci(int n) {
    Queue queue;
    initQueue(&queue);

    int a = 0, b = 1;

    for (int i = 0; i < n; i++) {
        enqueue(&queue, a);
        printf("%d, ", a);

        int next = a + b;
        a = b;
        b = next;
    }

    printf("\n");

    // Dequeue and print any remaining numbers in the queue (if any)
    while (!isEmpty(&queue)) {
        int data = dequeue(&queue);
        printf("%d, ", data);
    }
}

int main() {
    int n = 5;  // Replace with the desired number of Fibonacci numbers
    generateFibonacci(n);
    return 0;
}
```

8) Function Call Logger.

WTD: Implement a stack that logs function calls during the runtime of a program. This stack should allow for backtracking, enabling the user to trace the sequence of function calls and understand the flow of execution.

(e.g: I/P: Call FuncA, Call FuncB, Return ; O/P: Current function - FuncA)

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

```c
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>

#define MAX_STACK_SIZE 100

typedef struct {
    char items[MAX_STACK_SIZE][256]; // Assuming function names are less
than 256 characters
    int top;
} FunctionCallStack;

// Initialize an empty function call stack
void initFunctionCallStack(FunctionCallStack *stack) {
    stack->top = -1;
}

// Check if the function call stack is empty
bool isFunctionCallStackEmpty(const FunctionCallStack *stack) {
    return stack->top == -1;
}

// Push a function call onto the stack
void pushFunctionCall(FunctionCallStack *stack, const char *functionName)
{
    if (stack->top < MAX_STACK_SIZE - 1) {
        strcpy(stack->items[++stack->top], functionName);
    } else {
        printf("Function call stack overflow!\n");
        exit(1);
    }
}

// Pop a function call from the stack
void popFunctionCall(FunctionCallStack *stack) {
    if (!isFunctionCallStackEmpty(stack)) {
        stack->top--;
    } else {
        printf("Function call stack underflow!\n");
        exit(1);
```

```
    }
}

// Get the current function being executed
const char *getCurrentFunction(const FunctionCallStack *stack) {
    if (!isFunctionCallStackEmpty(stack)) {
        return stack->items[stack->top];
    } else {
        return "No function";
    }
}

int main() {
    FunctionCallStack callStack;
    initFunctionCallStack(&callStack);

    // Simulate function calls and returns
    pushFunctionCall(&callStack, "FuncA");
    printf("Current function - %s\n", getCurrentFunction(&callStack));

    pushFunctionCall(&callStack, "FuncB");
    printf("Current function - %s\n", getCurrentFunction(&callStack));

    popFunctionCall(&callStack);
    printf("Returning from %s\n", getCurrentFunction(&callStack));

    popFunctionCall(&callStack);
    printf("Returning from %s\n", getCurrentFunction(&callStack));

    return 0;
}
```

9) Queue-based Text Filter.

WTD: Develop a queue-based text filter that removes specific words from a given text string. The words to be filtered out should be enqueued and then compared against the text for filtering.

(e.g: I/P: Text - "Hello world", Filter - "world" ; O/P: "Hello")

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

// Define a structure for a queue node
typedef struct QueueNode {
    char data[256]; // Assuming words are less than 256 characters
    struct QueueNode *next;
} QueueNode;

// Define a structure for a queue
typedef struct {
    QueueNode *front;
    QueueNode *rear;
} StringQueue;

// Initialize an empty queue
void initStringQueue(StringQueue *queue) {
    queue->front = NULL;
    queue->rear = NULL;
}

// Check if the queue is empty
bool isStringQueueEmpty(const StringQueue *queue) {
    return queue->front == NULL;
}

// Enqueue a string into the queue
void enqueueString(StringQueue *queue, const char *data) {
    QueueNode *newNode = (QueueNode *)malloc(sizeof(QueueNode));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }

    strcpy(newNode->data, data);
    newNode->next = NULL;
```

```c
    if (isStringQueueEmpty(queue)) {
        queue->front = newNode;
        queue->rear = newNode;
    } else {
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
}

// Dequeue a string from the queue
void dequeueString(StringQueue *queue) {
    if (!isStringQueueEmpty(queue)) {
        QueueNode *temp = queue->front;
        queue->front = temp->next;
        free(temp);
    }
}

// Free memory used by the queue
void destroyStringQueue(StringQueue *queue) {
    while (!isStringQueueEmpty(queue)) {
        dequeueString(queue);
    }
}

// Check if a word is in the queue
bool isInQueue(const StringQueue *queue, const char *word) {
    QueueNode *current = queue->front;
    while (current != NULL) {
        if (strcmp(current->data, word) == 0) {
            return true;
        }
        current = current->next;
    }
    return false;
}

// Filter words from the text
char *filterText(const char *text, StringQueue *filterWords) {
    char *filteredText = strdup(text);
```

```c
    if (filteredText == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }

    char *word = strtok(filteredText, " ");
    char *prevWord = NULL;

    while (word != NULL) {
        if (isInQueue(filterWords, word)) {
            if (prevWord != NULL) {
                size_t wordLen = strlen(word);
                // Replace word with spaces of the same length
                memset(prevWord + strlen(prevWord), ' ', wordLen);
            } else {
                // If the first word is filtered, replace it with spaces
                size_t wordLen = strlen(word);
                memset(word, ' ', wordLen);
            }
        }
        prevWord = word;
        word = strtok(NULL, " ");
    }

    return filteredText;
}

int main() {
    StringQueue filterWords;
    initStringQueue(&filterWords);

    // Add words to filter out
    enqueueString(&filterWords, "world");

    const char *text = "Hello world"; // Replace with your input text
    char *filteredText = filterText(text, &filterWords);

    if (filteredText != NULL) {
        printf("%s\n", filteredText);
        free(filteredText);
```

```
    }

    destroyStringQueue(&filterWords);

    return 0;
}
```

10) Recursive to Iterative Converter.

WTD: Create a function that uses a stack to convert a recursive algorithm into its iterative version. For example, convert a recursive Fibonacci function into an iterative one that uses a stack for storage.
(e.g: I/P: Fibonacci(5) ; O/P: 5)

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

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

// Define a structure for a stack
typedef struct {
    StackNode *top;
} Stack;

// Initialize an empty stack
void initStack(Stack *stack) {
    stack->top = NULL;
}

// Check if the stack is empty
int isStackEmpty(Stack *stack) {
    return stack->top == NULL;
}

// Push an element onto the stack
```

```
void push(Stack *stack, int data) {
    StackNode *newNode = (StackNode *)malloc(sizeof(StackNode));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = stack->top;
    stack->top = newNode;
}

// Pop an element from the stack
int pop(Stack *stack) {
    if (!isStackEmpty(stack)) {
        StackNode *temp = stack->top;
        int data = temp->data;
        stack->top = temp->next;
        free(temp);
        return data;
    } else {
        printf("Stack is empty.\n");
        exit(1);
    }
}

// Fibonacci function using an iterative approach with a stack
int iterativeFibonacci(int n) {
    if (n <= 1) {
        return n;
    }

    Stack fibStack;
    initStack(&fibStack);
    push(&fibStack, 1);
    push(&fibStack, 0);

    int result = 0;

    for (int i = 2; i <= n; i++) {
        int a = pop(&fibStack);
```

```c
        int b = pop(&fibStack);
        result = a + b;
        push(&fibStack, a);
        push(&fibStack, result);
    }

    return result;
}

int main() {
    int n = 5; // Replace with the desired Fibonacci number
    int result = iterativeFibonacci(n);
    printf("Fibonacci(%d) = %d\n", n, result);
    return 0;
}
```

11) Stack-based Text Editor.

WTD: Design a text editor that uses a stack to implement basic text editing features like undo and redo. Each operation should push the current state of the text onto the stack, allowing for easy rollback or re-application of changes.
(e.g: I/P: Add "Hello", Undo, Add "Hi" ; O/P: "Hi")

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

// Define a structure for a stack node
typedef struct StackNode {
    char* text;
    struct StackNode* next;
} StackNode;

// Define a structure for a stack
typedef struct {
    StackNode* top;
} TextEditorStack;
```

```c
// Initialize an empty stack
void initTextEditorStack(TextEditorStack* stack) {
    stack->top = NULL;
}

// Check if the stack is empty
bool isTextEditorStackEmpty(TextEditorStack* stack) {
    return stack->top == NULL;
}

// Push the current text onto the stack
void pushTextEditorState(TextEditorStack* stack, const char* text) {
    StackNode* newNode = (StackNode*)malloc(sizeof(StackNode));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    newNode->text = strdup(text);
    newNode->next = stack->top;
    stack->top = newNode;
}

// Pop the top state from the stack and return it
char* popTextEditorState(TextEditorStack* stack) {
    if (!isTextEditorStackEmpty(stack)) {
        StackNode* temp = stack->top;
        char* text = temp->text;
        stack->top = temp->next;
        free(temp);
        return text;
    } else {
        printf("Stack is empty.\n");
        exit(1);
    }
}

// Free memory used by the stack
void destroyTextEditorStack(TextEditorStack* stack) {
    while (!isTextEditorStackEmpty(stack)) {
        char* text = popTextEditorState(stack);
```

```c
        free(text);
    }
}

int main() {
    TextEditorStack editorStack;
    initTextEditorStack(&editorStack);

    char* currentText = NULL;
    char input[256]; // Assuming input lines are less than 256 characters

    while (1) {
        printf("Enter a command (Add/Undo/Redo/Quit): ");
        fgets(input, sizeof(input), stdin);

        if (strncmp(input, "Add", 3) == 0) {
            // Add operation
            if (currentText != NULL) {
                free(currentText);
            }
            currentText = (char*)malloc(strlen(input) - 4); // Exclude
"Add " prefix
            strcpy(currentText, input + 4);
            pushTextEditorState(&editorStack, currentText);
        } else if (strncmp(input, "Undo", 4) == 0) {
            // Undo operation
            if (!isTextEditorStackEmpty(&editorStack)) {
                char* undoneText = popTextEditorState(&editorStack);
                if (currentText != NULL) {
                    free(currentText);
                }
                currentText = undoneText;
            }
        } else if (strncmp(input, "Redo", 4) == 0) {
            // Redo operation
            // (You can implement redo functionality if needed)
        } else if (strncmp(input, "Quit", 4) == 0) {
            // Quit the editor
            break;
        } else {
```

```
            printf("Invalid command. Try again.\n");
        }

        // Print the current text
        printf("Current Text: %s", (currentText != NULL) ? currentText :
"(empty)\n");
    }

    // Clean up and exit
    if (currentText != NULL) {
        free(currentText);
    }
    destroyTextEditorStack(&editorStack);

    return 0;
}
```

12) Queue-based Logger.

WTD: Build a queue-based logging system that logs and retrieves various system events. The queue should have a predefined size, and when it gets full, the oldest log entry should be removed to make space for a new one.
(e.g: I/P: Log "Event1", Log "Event2", Retrieve ; O/P: "Event1")

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

#define MAX_QUEUE_SIZE 10

typedef struct {
  char *event;
} LogEntry;

typedef struct {
  LogEntry *entries[MAX_QUEUE_SIZE];
  int front;
  int rear;
} LogQueue;
```

```c
void log_init(LogQueue *queue) {
  queue->front = 0;
  queue->rear = 0;
}

void log_enqueue(LogQueue *queue, char *event) {
  if (queue->rear == MAX_QUEUE_SIZE) {
    // The queue is full, so remove the oldest log entry.
    queue->front = (queue->front + 1) % MAX_QUEUE_SIZE;
  }

  queue->entries[queue->rear] = event;
  queue->rear = (queue->rear + 1) % MAX_QUEUE_SIZE;
}

char *log_dequeue(LogQueue *queue) {
  if (queue->front == queue->rear) {
    // The queue is empty.
    return NULL;
  }

  char *event = queue->entries[queue->front];
  queue->front = (queue->front + 1) % MAX_QUEUE_SIZE;

  return event;
}

int main() {
  LogQueue queue;

  log_init(&queue);

  log_enqueue(&queue, "Event1");
  log_enqueue(&queue, "Event2");

  char *event = log_dequeue(&queue);
  printf("Retrieved event: %s\n", event);

  return 0;
}
```

13) Multi-stack Array.

WTD: Design a system that allows multiple stacks to be stored within a single array. The space utilization should be optimized so that as one stack grows, it can acquire more space without causing overflow errors for the other stacks.
(e.g: I/P: Push 1 to Stack1, Push 'a' to Stack2 ; O/P: Array - [1, 'a'])

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

// Define the maximum size of the array
#define MAX_SIZE 100

// Define the number of stacks (change as needed)
#define NUM_STACKS 2

// Calculate the size allocated for each stack
#define STACK_SIZE (MAX_SIZE / NUM_STACKS)

// Define the MultiStack structure
typedef struct {
    int data[MAX_SIZE];
    int stackTops[NUM_STACKS];
} MultiStack;

// Initialize the MultiStack
void initMultiStack(MultiStack *stack) {
    for (int i = 0; i < NUM_STACKS; i++) {
        stack->stackTops[i] = i * STACK_SIZE - 1;
    }
}

// Check if a stack is empty
bool isStackEmpty(MultiStack *stack, int stackNumber) {
    return stack->stackTops[stackNumber] < stackNumber * STACK_SIZE;
}
```

```c
// Check if a stack is full
bool isStackFull(MultiStack *stack, int stackNumber) {
    return stack->stackTops[stackNumber] >= (stackNumber + 1) * STACK_SIZE
- 1;
}

// Push an element onto a stack
void push(MultiStack *stack, int stackNumber, int value) {
    if (!isStackFull(stack, stackNumber)) {
        stack->stackTops[stackNumber]++;
        stack->data[stack->stackTops[stackNumber]] = value;
    } else {
        printf("Stack %d is full. Cannot push %d.\n", stackNumber, value);
    }
}

// Pop an element from a stack
int pop(MultiStack *stack, int stackNumber) {
    if (!isStackEmpty(stack, stackNumber)) {
        int value = stack->data[stack->stackTops[stackNumber]];
        stack->stackTops[stackNumber]--;
        return value;
    } else {
        printf("Stack %d is empty.\n", stackNumber);
        return -1; // Return a sentinel value to indicate an empty stack
    }
}

int main() {
    MultiStack stack;
    initMultiStack(&stack);

    // Push elements onto different stacks
    push(&stack, 0, 1); // Push 1 to Stack 0
    push(&stack, 1, 'a'); // Push 'a' to Stack 1

    // Pop elements from different stacks
    int poppedValue1 = pop(&stack, 0);
    int poppedValue2 = pop(&stack, 1);
```

```
    // Print the popped values
    printf("Popped from Stack 0: %d\n", poppedValue1);
    printf("Popped from Stack 1: %c\n", (char)poppedValue2);

    return 0;
}
```

14) Circular Queue.

WTD: Develop a circular queue that overwrites the oldest elements when the queue reaches its capacity. Implement wrap-around functionality to make sure that new elements are inserted at the start of the array when the end is reached.
(e.g: I/P: Enqueue 1, 2, 3, 4 (Size=3) ; O/P: 2, 3, 4)

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

#define MAX_SIZE 3

typedef struct {
    int *data;
    int front;
    int rear;
    int size;
} CircularQueue;

// Initialize the circular queue
void initCircularQueue(CircularQueue *queue) {
    queue->data = (int *)malloc(MAX_SIZE * sizeof(int));
    if (queue->data == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    queue->front = -1;
    queue->rear = -1;
    queue->size = 0;
}
```

```c
// Check if the circular queue is empty
bool isCircularQueueEmpty(CircularQueue *queue) {
    return queue->size == 0;
}

// Check if the circular queue is full
bool isCircularQueueFull(CircularQueue *queue) {
    return queue->size == MAX_SIZE;
}

// Enqueue an element into the circular queue
void enqueue(CircularQueue *queue, int value) {
    if (isCircularQueueFull(queue)) {
        // If the queue is full, overwrite the oldest element
        queue->front = (queue->front + 1) % MAX_SIZE;
    }

    queue->rear = (queue->rear + 1) % MAX_SIZE;
    queue->data[queue->rear] = value;

    if (isCircularQueueFull(queue)) {
        queue->front = (queue->front + 1) % MAX_SIZE;
    } else if (isCircularQueueEmpty(queue)) {
        queue->front = 0;
    }

    if (queue->size < MAX_SIZE) {
        queue->size++;
    }
}

// Dequeue an element from the circular queue
int dequeue(CircularQueue *queue) {
    if (!isCircularQueueEmpty(queue)) {
        int value = queue->data[queue->front];
        queue->front = (queue->front + 1) % MAX_SIZE;
        queue->size--;
        return value;
    } else {
        printf("Queue is empty.\n");
```

```c
        exit(1);
    }
}

// Clean up and destroy the circular queue
void destroyCircularQueue(CircularQueue *queue) {
    free(queue->data);
    queue->front = -1;
    queue->rear = -1;
    queue->size = 0;
}

int main() {
    CircularQueue queue;
    initCircularQueue(&queue);

    // Enqueue elements
    enqueue(&queue, 1);
    enqueue(&queue, 2);
    enqueue(&queue, 3);
    enqueue(&queue, 4);

    // Dequeue and print elements
    while (!isCircularQueueEmpty(&queue)) {
        int value = dequeue(&queue);
        printf("%d, ", value);
    }

    printf("\n");

    // Clean up and destroy the circular queue
    destroyCircularQueue(&queue);

    return 0;
}
```

15) Min-Element Stack.

WTD: Construct a stack that can return the minimum element from the stack in constant time O(1). Use an auxiliary stack or any other data structure to keep track of the minimum element. (e.g: I/P: Push 4, Push 2, Push 8 ; O/P: Min - 2)

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

// Define the maximum size of the stack
#define MAX_SIZE 100

typedef struct {
    int data[MAX_SIZE];
    int top;
} MinElementStack;

typedef struct {
    int data[MAX_SIZE];
    int top;
} MinElementAuxStack;

// Initialize the main stack
void initMinElementStack(MinElementStack *stack) {
    stack->top = -1;
}

// Initialize the auxiliary stack for minimum elements
void initMinElementAuxStack(MinElementAuxStack *auxStack) {
    auxStack->top = -1;
}

// Check if the main stack is empty
bool isMinElementStackEmpty(MinElementStack *stack) {
    return stack->top == -1;
}

// Check if the auxiliary stack for minimum elements is empty
bool isMinElementAuxStackEmpty(MinElementAuxStack *auxStack) {
    return auxStack->top == -1;
```

```c
}

// Push an element onto the main stack
void pushMinElement(MinElementStack *stack, MinElementAuxStack *auxStack,
int value) {
    if (stack->top >= MAX_SIZE - 1) {
        printf("Stack overflow.\n");
        exit(1);
    }

    stack->top++;
    stack->data[stack->top] = value;

    // Update the auxiliary stack for minimum elements
    if (isMinElementAuxStackEmpty(auxStack) || value <=
auxStack->data[auxStack->top]) {
        auxStack->top++;
        auxStack->data[auxStack->top] = value;
    }
}

// Pop an element from the main stack
int popMinElement(MinElementStack *stack, MinElementAuxStack *auxStack) {
    if (isMinElementStackEmpty(stack)) {
        printf("Stack underflow.\n");
        exit(1);
    }

    int poppedValue = stack->data[stack->top];
    stack->top--;

    // Check if the popped value is the minimum element
    if (poppedValue == auxStack->data[auxStack->top]) {
        auxStack->top--;
    }

    return poppedValue;
}

// Get the minimum element from the main stack
```

```
int getMinElement(MinElementAuxStack *auxStack) {
    if (isMinElementAuxStackEmpty(auxStack)) {
        printf("Stack is empty.\n");
        exit(1);
    }

    return auxStack->data[auxStack->top];
}

int main() {
    MinElementStack stack;
    MinElementAuxStack auxStack;
    initMinElementStack(&stack);
    initMinElementAuxStack(&auxStack);

    // Push elements onto the main stack
    pushMinElement(&stack, &auxStack, 4);
    pushMinElement(&stack, &auxStack, 2);
    pushMinElement(&stack, &auxStack, 8);

    // Get the minimum element from the main stack
    int minElement = getMinElement(&auxStack);
    printf("Min - %d\n", minElement);

    return 0;
}
```

16) Stack Sorting.

WTD: Design a method to sort a stack. You are allowed to use only one additional stack as a temporary storage. No other data structures should be used.
(e.g: I/P: Stack - [4, 3, 1] ; O/P: Stack - [1, 3, 4])

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

// Define the maximum size of the stack
#define MAX_SIZE 100
```

```
typedef struct {
    int data[MAX_SIZE];
    int top;
} Stack;

// Initialize the stack
void initStack(Stack *stack) {
    stack->top = -1;
}

// Check if the stack is empty
bool isStackEmpty(Stack *stack) {
    return stack->top == -1;
}

// Check if the stack is full
bool isStackFull(Stack *stack) {
    return stack->top >= MAX_SIZE - 1;
}

// Push an element onto the stack
void push(Stack *stack, int value) {
    if (isStackFull(stack)) {
        printf("Stack overflow.\n");
        exit(1);
    }

    stack->top++;
    stack->data[stack->top] = value;
}

// Pop an element from the stack
int pop(Stack *stack) {
    if (isStackEmpty(stack)) {
        printf("Stack underflow.\n");
        exit(1);
    }

    int poppedValue = stack->data[stack->top];
```

```c
    stack->top--;
    return poppedValue;
}

// Sort a stack in ascending order using one additional stack
void sortStack(Stack *inputStack) {
    Stack auxStack;
    initStack(&auxStack);

    while (!isStackEmpty(inputStack)) {
        int temp = pop(inputStack);

        while (!isStackEmpty(&auxStack) && auxStack.data[auxStack.top] >
temp) {
            push(inputStack, pop(&auxStack));
        }

        push(&auxStack, temp);
    }

    // Transfer elements back to the input stack for ascending order
    while (!isStackEmpty(&auxStack)) {
        push(inputStack, pop(&auxStack));
    }
}

int main() {
    Stack stack;
    initStack(&stack);

    // Push elements onto the stack
    push(&stack, 4);
    push(&stack, 3);
    push(&stack, 1);

    // Sort the stack
    sortStack(&stack);

    // Print the sorted stack
    printf("Stack - [");
```

```
    while (!isStackEmpty(&stack)) {
        printf("%d", pop(&stack));
        if (!isStackEmpty(&stack)) {
            printf(", ");
        }
    }
    printf("]\n");

    return 0;
}
```

17) Queue from Stacks.

WTD: Implement a queue using two stacks. Use one stack for enqueuing and another for dequeuing. Make sure the oldest element gets dequeued.
(e.g: I/P: Enqueue 1, Enqueue 2 ; O/P: Dequeue - 1)

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

// Define the maximum size of the stacks
#define MAX_SIZE 100

typedef struct {
    int data[MAX_SIZE];
    int top;
} Stack;

typedef struct {
    Stack enqueueStack;
    Stack dequeueStack;
} Queue;

// Initialize a stack
void initStack(Stack *stack) {
    stack->top = -1;
}
```

```
// Check if a stack is empty
bool isStackEmpty(Stack *stack) {
    return stack->top == -1;
}

// Check if a stack is full
bool isStackFull(Stack *stack) {
    return stack->top >= MAX_SIZE - 1;
}

// Push an element onto a stack
void push(Stack *stack, int value) {
    if (isStackFull(stack)) {
        printf("Stack overflow.\n");
        exit(1);
    }

    stack->top++;
    stack->data[stack->top] = value;
}

// Pop an element from a stack
int pop(Stack *stack) {
    if (isStackEmpty(stack)) {
        printf("Stack underflow.\n");
        exit(1);
    }

    int poppedValue = stack->data[stack->top];
    stack->top--;
    return poppedValue;
}

// Initialize the queue
void initQueue(Queue *queue) {
    initStack(&(queue->enqueueStack));
    initStack(&(queue->dequeueStack));
}

// Enqueue an element into the queue
```

```c
void enqueue(Queue *queue, int value) {
    push(&(queue->enqueueStack), value);
}

// Dequeue an element from the queue
int dequeue(Queue *queue) {
    if (isStackEmpty(&(queue->dequeueStack))) {
        // If the dequeue stack is empty, transfer elements from enqueue
stack
        while (!isStackEmpty(&(queue->enqueueStack))) {
            int temp = pop(&(queue->enqueueStack));
            push(&(queue->dequeueStack), temp);
        }
    }

    if (!isStackEmpty(&(queue->dequeueStack))) {
        return pop(&(queue->dequeueStack));
    } else {
        printf("Queue is empty.\n");
        exit(1);
    }
}

int main() {
    Queue queue;
    initQueue(&queue);

    // Enqueue elements
    enqueue(&queue, 1);
    enqueue(&queue, 2);

    // Dequeue and print elements
    int dequeuedValue = dequeue(&queue);
    printf("Dequeue - %d\n", dequeuedValue);

    return 0;
}
```

18) Stack-based Calculator.

WTD: Build a simple calculator to evaluate postfix expressions. Use a stack to keep track of operands and apply operators as they appear.
(e.g: I/P: "5 1 2 + 4 * + 3 -" ; O/P: 14)

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

// Define the maximum size of the stack
#define MAX_SIZE 100

typedef struct {
    double data[MAX_SIZE];
    int top;
} Stack;

// Initialize the stack
void initStack(Stack *stack) {
    stack->top = -1;
}

// Check if the stack is empty
bool isStackEmpty(Stack *stack) {
    return stack->top == -1;
}

// Check if the stack is full
bool isStackFull(Stack *stack) {
    return stack->top >= MAX_SIZE - 1;
}

// Push an element onto the stack
void push(Stack *stack, double value) {
    if (isStackFull(stack)) {
        printf("Stack overflow.\n");
        exit(1);
    }
```

```
    stack->top++;
    stack->data[stack->top] = value;
}

// Pop an element from the stack
double pop(Stack *stack) {
    if (isStackEmpty(stack)) {
        printf("Stack underflow.\n");
        exit(1);
    }

    double poppedValue = stack->data[stack->top];
    stack->top--;
    return poppedValue;
}

// Evaluate a postfix expression
double evaluatePostfix(char *expression) {
    Stack operandStack;
    initStack(&operandStack);

    char *token = strtok(expression, " ");
    while (token != NULL) {
        if (isdigit(token[0])) {
            // If the token is a number, push it onto the operand stack
            push(&operandStack, atof(token));
        } else {
            // If the token is an operator, pop the top two operands,
apply the operator, and push the result
            double operand2 = pop(&operandStack);
            double operand1 = pop(&operandStack);
            double result;

            switch (token[0]) {
                case '+':
                    result = operand1 + operand2;
                    break;
                case '-':
                    result = operand1 - operand2;
                    break;
```

```
                case '*':
                    result = operand1 * operand2;
                    break;
                case '/':
                    result = operand1 / operand2;
                    break;
                default:
                    printf("Invalid operator: %s\n", token);
                    exit(1);
            }

            push(&operandStack, result);
        }

        token = strtok(NULL, " ");
    }

    // The final result should be at the top of the operand stack
    return pop(&operandStack);
}

int main() {
    char expression[] = "5 1 2 + 4 * + 3 -";
    double result = evaluatePostfix(expression);
    printf("Result: %.2lf\n", result);

    return 0;
}
```

19) Priority Queue using Heap.

WTD: Create a priority queue using either a max-heap or a min-heap. Implement methods for insertion and removal of elements based on their priority.
(e.g: I/P: Insert 3, Insert 1, Insert 4 ; O/P: RemoveMax - 4)

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

// Define the maximum size of the heap
```

```c
#define MAX_SIZE 100

typedef struct {
    int *data;
    int size;
    int capacity;
} MaxHeap;

// Initialize the max-heap
MaxHeap *initMaxHeap(int capacity) {
    MaxHeap *heap = (MaxHeap *)malloc(sizeof(MaxHeap));
    if (heap == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }

    heap->data = (int *)malloc(capacity * sizeof(int));
    if (heap->data == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }

    heap->size = 0;
    heap->capacity = capacity;

    return heap;
}

// Swap two elements in the heap
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Heapify the max-heap (downward)
void heapify(MaxHeap *heap, int index) {
    int largest = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;
```

```
    if (left < heap->size && heap->data[left] > heap->data[largest]) {
        largest = left;
    }

    if (right < heap->size && heap->data[right] > heap->data[largest]) {
        largest = right;
    }

    if (largest != index) {
        swap(&heap->data[index], &heap->data[largest]);
        heapify(heap, largest);
    }
}

// Insert an element into the max-heap
void insert(MaxHeap *heap, int value) {
    if (heap->size >= heap->capacity) {
        printf("Heap overflow.\n");
        exit(1);
    }

    int index = heap->size;
    heap->data[index] = value;
    heap->size++;

    // Fix the max-heap property by moving the element up
    while (index > 0 && heap->data[index] > heap->data[(index - 1) / 2]) {
        swap(&heap->data[index], &heap->data[(index - 1) / 2]);
        index = (index - 1) / 2;
    }
}

// Remove and return the maximum element from the max-heap
int removeMax(MaxHeap *heap) {
    if (heap->size <= 0) {
        printf("Heap underflow.\n");
        exit(1);
    }
```

```
    int max = heap->data[0];
    heap->data[0] = heap->data[heap->size - 1];
    heap->size--;

    // Restore the max-heap property by heapifying the root
    heapify(heap, 0);

    return max;
}

// Clean up and destroy the max-heap
void destroyMaxHeap(MaxHeap *heap) {
    free(heap->data);
    free(heap);
}

int main() {
    MaxHeap *heap = initMaxHeap(MAX_SIZE);

    // Insert elements into the max-heap
    insert(heap, 3);
    insert(heap, 1);
    insert(heap, 4);

    // Remove and print the maximum element
    int max = removeMax(heap);
    printf("RemoveMax - %d\n", max);

    // Clean up and destroy the max-heap
    destroyMaxHeap(heap);

    return 0;
}
```

20) Radix Sort using Queue.

WTD: Implement Radix Sort using a queue. Use additional queues to sort each digit from the least significant to the most significant.

(e.g: I/P: [170, 45, 75, 90, 802, 24, 2, 66] ; O/P: [2, 24, 45, 66, 75, 90, 170, 802])

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

// Define the maximum size of the queue and the number of digits (0-9)
#define MAX_SIZE 100
#define NUM_DIGITS 10

typedef struct {
    int data[MAX_SIZE];
    int front, rear;
} Queue;

// Initialize the queue
void initQueue(Queue *queue) {
    queue->front = queue->rear = -1;
}

// Check if the queue is empty
bool isQueueEmpty(Queue *queue) {
    return queue->front == -1;
}

// Check if the queue is full
bool isQueueFull(Queue *queue) {
    return (queue->rear + 1) % MAX_SIZE == queue->front;
}

// Enqueue an element into the queue
void enqueue(Queue *queue, int value) {
    if (isQueueFull(queue)) {
        printf("Queue overflow.\n");
        exit(1);
    }

    if (isQueueEmpty(queue)) {
        queue->front = queue->rear = 0;
    } else {
```

```c
        queue->rear = (queue->rear + 1) % MAX_SIZE;
    }

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

// Dequeue an element from the queue
int dequeue(Queue *queue) {
    if (isQueueEmpty(queue)) {
        printf("Queue underflow.\n");
        exit(1);
    }

    int dequeuedValue = queue->data[queue->front];

    if (queue->front == queue->rear) {
        // Queue has only one element
        queue->front = queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % MAX_SIZE;
    }

    return dequeuedValue;
}

// Find the maximum number of digits in an array of numbers
int findMaxDigits(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }

    int digits = 0;
    while (max > 0) {
        digits++;
        max /= 10;
    }
```

```
    return digits;
}

// Perform Radix Sort on an array of numbers
void radixSort(int arr[], int n) {
    int maxDigits = findMaxDigits(arr, n);
    Queue digitQueues[NUM_DIGITS];

    for (int i = 0; i < NUM_DIGITS; i++) {
        initQueue(&digitQueues[i]);
    }

    int divisor = 1;
    for (int digitPosition = 1; digitPosition <= maxDigits;
digitPosition++) {
        // Distribute numbers into digit queues based on the current digit
position
        for (int i = 0; i < n; i++) {
            int digit = (arr[i] / divisor) % 10;
            enqueue(&digitQueues[digit], arr[i]);
        }

        // Collect numbers from digit queues back into the original array
        int j = 0;
        for (int i = 0; i < NUM_DIGITS; i++) {
            while (!isQueueEmpty(&digitQueues[i])) {
                arr[j] = dequeue(&digitQueues[i]);
                j++;
            }
        }

        divisor *= 10;
    }
}

int main() {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Input Array: ");
```

```
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    radixSort(arr, n);

    printf("Sorted Array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

21) Queue with Two Priorities.

WTD: Design a queue data structure that handles two levels of priority (high and low). Ensure that elements with high priority are dequeued before those with low priority.
(e.g: I/P: Enqueue 1 (High), Enqueue 2 (Low) ; O/P: Dequeue - 1)

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

// Define the maximum size of the queue
#define MAX_SIZE 100

typedef struct {
    int data[MAX_SIZE];
    int front, rear;
} Queue;

// Initialize the queue
void initQueue(Queue *queue) {
    queue->front = queue->rear = -1;
}
```

```c
// Check if the queue is empty
bool isQueueEmpty(Queue *queue) {
    return queue->front == -1;
}

// Check if the queue is full
bool isQueueFull(Queue *queue) {
    return (queue->rear + 1) % MAX_SIZE == queue->front;
}

// Enqueue an element into the queue with the specified priority
void enqueue(Queue *queue, int value, bool highPriority) {
    if (isQueueFull(queue)) {
        printf("Queue overflow.\n");
        exit(1);
    }

    if (isQueueEmpty(queue)) {
        queue->front = queue->rear = 0;
    } else {
        queue->rear = (queue->rear + 1) % MAX_SIZE;
    }

    if (highPriority) {
        // Move all elements to the right to make space for the
high-priority element
        int i = queue->rear;
        while (i > queue->front) {
            queue->data[(i + 1) % MAX_SIZE] = queue->data[i];
            i = (i - 1 + MAX_SIZE) % MAX_SIZE;
        }
        queue->rear = (queue->rear + 1) % MAX_SIZE;
        queue->data[queue->front] = value;
    } else {
        queue->data[queue->rear] = value;
    }
}

// Dequeue an element from the queue
int dequeue(Queue *queue) {
```

```
    if (isQueueEmpty(queue)) {
        printf("Queue underflow.\n");
        exit(1);
    }

    int dequeuedValue = queue->data[queue->front];

    if (queue->front == queue->rear) {
        // Queue has only one element
        queue->front = queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % MAX_SIZE;
    }

    return dequeuedValue;
}

int main() {
    Queue queue;
    initQueue(&queue);

    // Enqueue elements with different priorities
    enqueue(&queue, 1, true);  // High priority
    enqueue(&queue, 2, false); // Low priority

    // Dequeue and print elements
    int dequeuedValue = dequeue(&queue);
    printf("Dequeue - %d\n", dequeuedValue);

    return 0;
}
```

22) Undo and Redo Stack.

WTD: Implement a system that allows for undo and redo operations. Use two stacks to keep track of all states, one for undo and another for redo.
(e.g: I/P: Write "Hello", Undo, Write "Hi" ; O/P: "Hi")

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

```c
#include <stdlib.h>
#include <stdbool.h>

#define MAX_SIZE 100

typedef struct {
    char data[MAX_SIZE];
    int top;
} Stack;

void initStack(Stack *stack) {
    stack->top = -1;
}

bool isStackEmpty(Stack *stack) {
    return stack->top == -1;
}

bool isStackFull(Stack *stack) {
    return stack->top == MAX_SIZE - 1;
}

void push(Stack *stack, char value) {
    if (isStackFull(stack)) {
        printf("Stack overflow.\n");
        exit(1);
    }
    stack->data[++stack->top] = value;
}

char pop(Stack *stack) {
    if (isStackEmpty(stack)) {
        printf("Stack underflow.\n");
        exit(1);
    }
    return stack->data[stack->top--];
}

// Undo and Redo stacks
Stack undoStack;
```

```c
Stack redoStack;

void undo() {
    if (!isStackEmpty(&undoStack)) {
        char value = pop(&undoStack);
        push(&redoStack, value);
    }
}

void redo() {
    if (!isStackEmpty(&redoStack)) {
        char value = pop(&redoStack);
        push(&undoStack, value);
    }
}

void write(char value) {
    push(&undoStack, value);
    // Clear redo stack when a new action is performed
    initStack(&redoStack);
}

int main() {
    initStack(&undoStack);
    initStack(&redoStack);

    // Simulate user actions
    write('H');
    write('e');
    write('l');
    write('l');
    write('o');

    printf("Current Text: ");
    while (!isStackEmpty(&undoStack)) {
        char value = pop(&undoStack);
        printf("%c", value);
    }
    printf("\n");
```

```c
    // Undo and redo operations
    undo();
    undo();

    printf("After Undo: ");
    while (!isStackEmpty(&undoStack)) {
        char value = pop(&undoStack);
        printf("%c", value);
    }
    printf("\n");

    redo();

    printf("After Redo: ");
    while (!isStackEmpty(&undoStack)) {
        char value = pop(&undoStack);
        printf("%c", value);
    }
    printf("\n");

    return 0;
}
```

23) Post-order Traversal using Stack.

WTD: Implement post-order traversal of a binary tree using a stack. Make sure to visit the left subtree, then the right subtree, and finally the root node.
(e.g: I/P: Tree - 1->2->3 ; O/P: 2, 3, 1)

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

// Define a binary tree node
typedef struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
} TreeNode;
```

```
// Define a structure to represent a stack node for tree nodes
typedef struct StackNode {
    TreeNode* node;
    struct StackNode* next;
} StackNode;

// Initialize the stack
void initStack(StackNode** stack) {
    *stack = NULL;
}

// Check if the stack is empty
bool isStackEmpty(StackNode* stack) {
    return stack == NULL;
}

// Push a tree node onto the stack
void push(StackNode** stack, TreeNode* node) {
    StackNode* newNode = (StackNode*)malloc(sizeof(StackNode));
    if (newNode == NULL) {
        printf("Memory allocation error.\n");
        exit(1);
    }
    newNode->node = node;
    newNode->next = *stack;
    *stack = newNode;
}

// Pop a tree node from the stack
TreeNode* pop(StackNode** stack) {
    if (isStackEmpty(*stack)) {
        printf("Stack underflow.\n");
        exit(1);
    }
    TreeNode* node = (*stack)->node;
    StackNode* temp = *stack;
    *stack = (*stack)->next;
    free(temp);
    return node;
```

```
}

// Post-order traversal using a stack
void postOrderTraversal(TreeNode* root) {
    if (root == NULL) return;

    StackNode* stack1;
    StackNode* stack2;

    initStack(&stack1);
    initStack(&stack2);

    push(&stack1, root);

    while (!isStackEmpty(stack1)) {
        TreeNode* node = pop(&stack1);
        push(&stack2, node);

        if (node->left) push(&stack1, node->left);
        if (node->right) push(&stack1, node->right);
    }

    while (!isStackEmpty(stack2)) {
        TreeNode* node = pop(&stack2);
        printf("%d ", node->data);
    }
}

// Create a new binary tree node
TreeNode* createNode(int data) {
    TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));
    if (newNode == NULL) {
        printf("Memory allocation error.\n");
        exit(1);
    }
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

```c
int main() {
    // Create a binary tree
    TreeNode* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);

    // Perform post-order traversal
    printf("Post-order Traversal: ");
    postOrderTraversal(root);
    printf("\n");

    return 0;
}
```

24) Balanced Parentheses using Stack.

WTD: Check for balanced parentheses in a given expression. Use a stack to keep track of opening and closing brackets, braces, and parentheses.

(e.g: I/P: "{[()]}" ; O/P: Balanced)

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

// Define a structure for the stack
typedef struct {
    char* items;
    int top;
    int capacity;
} Stack;

// Initialize the stack
Stack* initStack(int capacity) {
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    if (stack == NULL) {
        printf("Memory allocation error.\n");
        exit(1);
    }
```

```c
    stack->items = (char*)malloc(sizeof(char) * capacity);
    if (stack->items == NULL) {
        printf("Memory allocation error.\n");
        free(stack);
        exit(1);
    }
    stack->top = -1;
    stack->capacity = capacity;
    return stack;
}

// Check if the stack is empty
bool isStackEmpty(Stack* stack) {
    return stack->top == -1;
}

// Push an element onto the stack
void push(Stack* stack, char item) {
    if (stack->top == stack->capacity - 1) {
        printf("Stack overflow.\n");
        exit(1);
    }
    stack->items[++stack->top] = item;
}

// Pop an element from the stack
char pop(Stack* stack) {
    if (isStackEmpty(stack)) {
        printf("Stack underflow.\n");
        exit(1);
    }
    return stack->items[stack->top--];
}

// Check if parentheses are balanced
bool areParenthesesBalanced(char* expression) {
    Stack* stack = initStack(strlen(expression));

    for (int i = 0; expression[i] != '\0'; i++) {
        char currentChar = expression[i];
```

```
        if (currentChar == '(' || currentChar == '[' || currentChar ==
'{') {
            push(stack, currentChar);
        } else if (currentChar == ')' || currentChar == ']' || currentChar
== '}') {
            if (isStackEmpty(stack)) {
                free(stack->items);
                free(stack);
                return false; // Unmatched closing parenthesis
            }
            char topChar = pop(stack);
            if ((currentChar == ')' && topChar != '(') ||
                (currentChar == ']' && topChar != '[') ||
                (currentChar == '}' && topChar != '{')) {
                free(stack->items);
                free(stack);
                return false; // Mismatched opening and closing
parentheses
            }
        }
    }

    bool balanced = isStackEmpty(stack); // Stack should be empty for
balanced parentheses
    free(stack->items);
    free(stack);
    return balanced;
}

int main() {
    char expression[] = "{[()]}";

    if (areParenthesesBalanced(expression)) {
        printf("Balanced\n");
    } else {
        printf("Not balanced\n");
    }

    return 0;
}
```

25) Queue-based Cache.

WTD: Implement a caching mechanism using a queue. When the cache is full, evict the least recently used item.

(e.g: I/P: Cache(2), Put 1, Put 2, Get 1, Put 3 ; O/P: Cache - [1, 3])

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

// Define a structure for a cache node
typedef struct CacheNode {
    int key;
    int value;
    struct CacheNode* next;
    struct CacheNode* prev;
} CacheNode;

// Define a structure for the cache
typedef struct {
    int capacity;
    int size;
    CacheNode* head; // Most recently used item
    CacheNode* tail; // Least recently used item
} LRUCache;

// Initialize the cache
LRUCache* initCache(int capacity) {
    LRUCache* cache = (LRUCache*)malloc(sizeof(LRUCache));
    if (cache == NULL) {
        printf("Memory allocation error.\n");
        exit(1);
    }
    cache->capacity = capacity;
    cache->size = 0;
    cache->head = NULL;
    cache->tail = NULL;
    return cache;
```

```c
}

// Remove a cache node from the list
void removeFromList(LRUCache* cache, CacheNode* node) {
    if (node->prev != NULL) {
        node->prev->next = node->next;
    } else {
        cache->head = node->next;
    }
    if (node->next != NULL) {
        node->next->prev = node->prev;
    } else {
        cache->tail = node->prev;
    }
}

// Move a cache node to the front of the list (most recently used)
void moveToFront(LRUCache* cache, CacheNode* node) {
    if (node == cache->head) {
        return; // Already at the front
    }
    removeFromList(cache, node);
    node->next = cache->head;
    node->prev = NULL;
    if (cache->head != NULL) {
        cache->head->prev = node;
    }
    cache->head = node;
    if (cache->tail == NULL) {
        cache->tail = node;
    }
}

// Evict the least recently used item from the cache
void evict(LRUCache* cache) {
    if (cache->tail != NULL) {
        CacheNode* tailPrev = cache->tail->prev;
        if (tailPrev != NULL) {
            tailPrev->next = NULL;
        }
```

```c
        free(cache->tail);
        cache->tail = tailPrev;
        if (cache->tail == NULL) {
            cache->head = NULL;
        }
        cache->size--;
    }
}

// Get a value from the cache
int get(LRUCache* cache, int key) {
    CacheNode* current = cache->head;
    while (current != NULL) {
        if (current->key == key) {
            moveToFront(cache, current);
            return current->value;
        }
        current = current->next;
    }
    return -1; // Key not found in cache
}

// Put a key-value pair into the cache
void put(LRUCache* cache, int key, int value) {
    CacheNode* current = cache->head;
    while (current != NULL) {
        if (current->key == key) {
            current->value = value;
            moveToFront(cache, current);
            return;
        }
        current = current->next;
    }

    // Key not found, add a new node
    CacheNode* newNode = (CacheNode*)malloc(sizeof(CacheNode));
    if (newNode == NULL) {
        printf("Memory allocation error.\n");
        exit(1);
    }
```

```c
    newNode->key = key;
    newNode->value = value;
    newNode->next = cache->head;
    newNode->prev = NULL;

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

    if (cache->tail == NULL) {
        cache->tail = newNode;
    }

    cache->size++;

    if (cache->size > cache->capacity) {
        evict(cache);
    }
}

int main() {
    LRUCache* cache = initCache(2);

    put(cache, 1, 1);
    put(cache, 2, 2);
    int val = get(cache, 1); // Returns 1
    printf("Get 1: %d\n", val);
    put(cache, 3, 3); // Evicts key 2
    val = get(cache, 2); // Returns -1 (not found)
    printf("Get 2: %d\n", val);
    put(cache, 4, 4); // Evicts key 1
    val = get(cache, 1); // Returns -1 (not found)
    printf("Get 1: %d\n", val);
    val = get(cache, 3); // Returns 3
    printf("Get 3: %d\n", val);
    val = get(cache, 4); // Returns 4
    printf("Get 4: %d\n", val);

    return 0;
```

```
}
```

—---Happy Learning —---