Assignment: Extra Questions

Q1. Implement queue using a stack and a function call stack.

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// --- Basic Stack Implementation ---

typedef struct Stack {

int top;

unsigned capacity;

int\* array;

} Stack;

Stack\* createStack(unsigned capacity) {

Stack\* stack = (Stack\*)malloc(sizeof(Stack));

if (!stack) return NULL;

stack->capacity = capacity;

stack->top = -1;

stack->array = (int\*)malloc(stack->capacity \* sizeof(int));

if (!stack->array) {

free(stack);

return NULL;

}

return stack;

}

int isStackEmpty(Stack\* stack) {

return stack->top == -1;

}

void push(Stack\* stack, int item) {

if (stack->top == (int)stack->capacity - 1) {

printf("Stack Overflow\n");

return;

}

stack->array[++stack->top] = item;

}

int pop(Stack\* stack) {

if (isStackEmpty(stack)) {

return INT\_MIN; // Error value

}

return stack->array[stack->top--];

}

// --- Queue Implementation using a Stack ---

typedef struct Queue {

Stack\* s1;

} Queue;

Queue\* createQueue(unsigned capacity) {

Queue\* q = (Queue\*)malloc(sizeof(Queue));

q->s1 = createStack(capacity);

return q;

}

// Enqueue is a simple push operation

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

push(q->s1, item);

printf("Enqueued: %d\n", item);

}

// Recursive helper function to find the bottom of the stack

int dequeueRecursive(Stack\* s) {

// Base case: If there's only one item, pop and return it

if (isStackEmpty(s)) {

return INT\_MIN; // Should not happen in the main dequeue logic

}

int item = pop(s);

if (isStackEmpty(s)) {

return item; // This is the bottom element

}

// Recursive step: go deeper

int bottom\_item = dequeueRecursive(s);

// After the recursive call returns, push the item back

// This happens on the way "up" the call stack

push(s, item);

return bottom\_item;

}

// Dequeue operation

int dequeue(Queue\* q) {

if (isStackEmpty(q->s1)) {

printf("Queue is empty\n");

return INT\_MIN;

}

return dequeueRecursive(q->s1);

}

// Free all allocated memory

void freeQueue(Queue\* q) {

free(q->s1->array);

free(q->s1);

free(q);

}

int main() {

Queue\* q = createQueue(10);

enqueue(q, 10);

enqueue(q, 20);

enqueue(q, 30);

printf("\nDequeued: %d\n", dequeue(q));

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

enqueue(q, 40);

printf("\nDequeued: %d\n", dequeue(q));

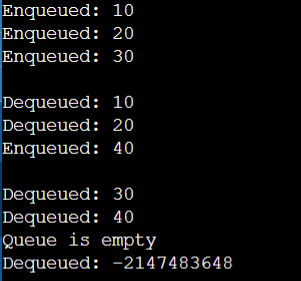
printf("Dequeued: %d\n", dequeue(q));

printf("Dequeued: %d\n", dequeue(q)); // Trying to dequeue from empty queue

freeQueue(q);

return 0;

}



Q2. Implement a stack using 2 queues.

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// --- Basic Queue Implementation ---

typedef struct QueueNode {

int data;

struct QueueNode\* next;

} QueueNode;

typedef struct Queue {

QueueNode \*front, \*rear;

int size;

} Queue;

Queue\* createQueue() {

Queue\* q = (Queue\*)malloc(sizeof(Queue));

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

q->size = 0;

return q;

}

int isQueueEmpty(Queue\* q) {

return q->size == 0;

}

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

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

temp->data = item;

temp->next = NULL;

q->size++;

if (q->rear == NULL) {

q->front = q->rear = temp;

return;

}

q->rear->next = temp;

q->rear = temp;

}

int dequeue(Queue\* q) {

if (isQueueEmpty(q)) {

return INT\_MIN;

}

QueueNode\* temp = q->front;

int item = temp->data;

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

if (q->front == NULL) {

q->rear = NULL;

}

free(temp);

q->size--;

return item;

}

// --- Stack Implementation using two Queues ---

typedef struct Stack {

Queue \*q1, \*q2;

} Stack;

Stack\* createStack() {

Stack\* s = (Stack\*)malloc(sizeof(Stack));

s->q1 = createQueue(); // The main queue

s->q2 = createQueue(); // The helper queue

return s;

}

int isStackEmpty\_sq(Stack\* s) {

return isQueueEmpty(s->q1);

}

// Push operation is a simple enqueue to q1

void push\_sq(Stack\* s, int item) {

enqueue(s->q1, item);

printf("Pushed: %d\n", item);

}

// Pop is the costly operation

int pop\_sq(Stack\* s) {

if (isStackEmpty\_sq(s)) {

printf("Stack is empty\n");

return INT\_MIN;

}

// Move all elements but the last one from q1 to q2

while (s->q1->size > 1) {

enqueue(s->q2, dequeue(s->q1));

}

// The last element in q1 is the one to be popped

int item = dequeue(s->q1);

// Swap the names of q1 and q2 to make q2 the main queue

Queue\* temp = s->q1;

s->q1 = s->q2;

s->q2 = temp;

return item;

}

int top\_sq(Stack\* s) {

if (isStackEmpty\_sq(s)) {

printf("Stack is empty\n");

return INT\_MIN;

}

// Similar logic to pop, but we must re-enqueue the last element

while (s->q1->size > 1) {

enqueue(s->q2, dequeue(s->q1));

}

int item = dequeue(s->q1);

enqueue(s->q2, item); // Put it back

Queue\* temp = s->q1;

s->q1 = s->q2;

s->q2 = temp;

return item;

}

// Free all allocated memory

void freeStack(Stack\* s) {

// Free any remaining nodes in both queues

while (!isQueueEmpty(s->q1)) dequeue(s->q1);

while (!isQueueEmpty(s->q2)) dequeue(s->q2);

free(s->q1);

free(s->q2);

free(s);

}

int main() {

Stack\* s = createStack();

push\_sq(s, 10);

push\_sq(s, 20);

push\_sq(s, 30);

printf("\nCurrent top: %d\n", top\_sq(s));

printf("Popped: %d\n", pop\_sq(s));

printf("Current top: %d\n", top\_sq(s));

printf("Popped: %d\n", pop\_sq(s));

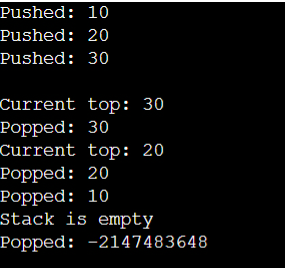
printf("Popped: %d\n", pop\_sq(s));

printf("Popped: %d\n", pop\_sq(s)); // Trying to pop from empty stack

freeStack(s);

return 0;

}



Q3. Write recursive function to determine if a number is a prime number.

#include <stdio.h>

#include <math.h> // Required for the sqrt() optimization

int is\_prime\_recursive(int n, int divisor) {

// Base case 1: If the divisor reaches 1, no factors were found. The number is prime.

if (divisor <= 1) {

return 1; // True

}

// Base case 2: If n is divisible by the current divisor, it's not prime.

if (n % divisor == 0) {

return 0; // False

}

// Recursive step: Call the function again with the next smaller divisor.

return is\_prime\_recursive(n, divisor - 1);

}

int main() {

int test\_nums[] = {17, 15, 2, 1, 0, -5, 29, 97, 100};

int num\_tests = sizeof(test\_nums) / sizeof(test\_nums[0]);

printf("--- Recursive Prime Number Checker ---\n");

for (int i = 0; i < num\_tests; i++) {

int n = test\_nums[i];

printf("Checking %d: ", n);

// Initial checks must be handled before the first recursive call.

if (n <= 1) {

printf("Not a prime number.\n");

} else if (n == 2) {

printf("Is a prime number.\n"); // 2 is a special prime case

} else {

// Start the recursive check. We only need to check up to sqrt(n).

if (is\_prime\_recursive(n, (int)sqrt(n))) {

printf("Is a prime number.\n");

} else {

printf("Not a prime number.\n");

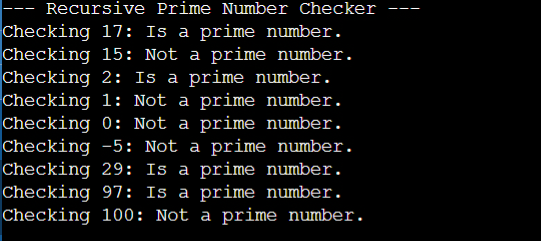
}

}

}

return 0;

}



Q4. Write recursive functions to assign the particular values to the array, and print the array in reverse and normal order.

#include <stdio.h>

#define MAX\_SIZE 5

void assign\_values\_recursive(int \*arr, int size) {

// Base case: If there are no elements to process, stop.

if (size <= 0) {

return;

}

// Recurse first to process the smaller, preceding part of the array.

assign\_values\_recursive(arr, size - 1);

// Assign value to the current last element on the way back up the call stack.

arr[size - 1] = size \* 10;

}

void print\_array\_normal(const int \*arr, int size) {

// Base case: If the array segment is empty, stop.

if (size <= 0) {

return;

}

// Print the first element, then recurse on the rest of the array.

printf("%d ", \*arr);

print\_array\_normal(arr + 1, size - 1);

}

void print\_array\_reverse(const int \*arr, int size) {

// Base case: If the array segment is empty, stop.

if (size <= 0) {

return;

}

// Recurse on the rest of the array first...

print\_array\_reverse(arr + 1, size - 1);

// ...then print the first element on the way back up the call stack.

printf("%d ", \*arr);

}

int main() {

int my\_array[MAX\_SIZE];

// 1. Assign values

assign\_values\_recursive(my\_array, MAX\_SIZE);

printf("Array has been assigned values.\n");

// 2. Print in normal order

printf("Array in normal order: ");

print\_array\_normal(my\_array, MAX\_SIZE);

printf("\n");

// 3. Print in reverse order

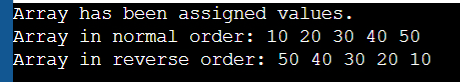
printf("Array in reverse order: ");

print\_array\_reverse(my\_array, MAX\_SIZE);

printf("\n");

return 0;

}



Q5. Write recursive function to print the given number in reverse order: 2015.

#include <stdio.h>

void print\_digits\_reverse(int n) {

// Handle negative numbers by printing the sign and proceeding with the positive version.

if (n < 0) {

printf("- ");

n = -n;

}

printf("%d ", n % 10);

if (n / 10 > 0) {

print\_digits\_reverse(n / 10);

}

}

int main() {

int num1 = 2015;

printf("The digits of %d in reverse are: ", num1);

print\_digits\_reverse(num1);

printf("\n");

int num2 = 98765;

printf("The digits of %d in reverse are: ", num2);

print\_digits\_reverse(num2);

printf("\n");

int num3 = -123;

printf("The digits of %d in reverse are: ", num3);

print\_digits\_reverse(num3);

printf("\n");

int num4 = 0;

printf("The digits of %d in reverse are: ", num4);

if (num4 == 0) {

printf("0");

} else {

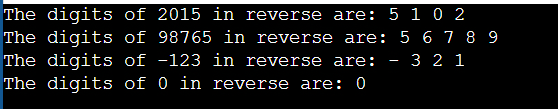
print\_digits\_reverse(num4);

}

printf("\n");

return 0;

}



Q6. Write a program to construct a spiral matrix using arrays.

#include <stdio.h>

#include <stdlib.h>

void printMatrix(int\*\* matrix, int rows, int cols) {

if (!matrix) return;

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

for (int j = 0; j < cols; j++) {

printf("%-4d", matrix[i][j]); // Print with padding

}

printf("\n");

}

}

void printSpiralOrder(int\*\* matrix, int rows, int cols) {

if (!matrix) return;

int top = 0, bottom = rows - 1;

int left = 0, right = cols - 1;

printf("[");

while (top <= bottom && left <= right) {

// 1. Print top row (left to right)

for (int i = left; i <= right; i++) {

printf("%d", matrix[top][i]);

if (top < bottom || i < right) printf(", ");

}

top++;

// 2. Print right column (top to bottom)

for (int i = top; i <= bottom; i++) {

printf("%d", matrix[i][right]);

if (left < right || i < bottom) printf(", ");

}

right--;

// 3. Print bottom row (right to left)

if (top <= bottom) {

for (int i = right; i >= left; i--) {

printf("%d", matrix[bottom][i]);

if (left < right || i > left) printf(", ");

}

bottom--;

}

// 4. Print left column (bottom to top)

if (left <= right) {

for (int i = bottom; i >= top; i--) {

printf("%d", matrix[i][left]);

if (i > top) printf(", ");

}

left++;

}

}

printf("]\n");

}

int\*\* generateSpiralMatrix(int rows, int cols) {

// Allocate memory for the matrix

int\*\* matrix = (int\*\*)malloc(rows \* sizeof(int\*));

if (!matrix) return NULL;

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

matrix[i] = (int\*)malloc(cols \* sizeof(int));

if (!matrix[i]) {

// Cleanup on failure

while (--i >= 0) free(matrix[i]);

free(matrix);

return NULL;

}

}

int top = 0, bottom = rows - 1;

int left = 0, right = cols - 1;

int num = 1; // The number to place in the matrix

while (top <= bottom && left <= right) {

// 1. Fill top row

for (int i = left; i <= right; i++) {

matrix[top][i] = num++;

}

top++;

// 2. Fill right column

for (int i = top; i <= bottom; i++) {

matrix[i][right] = num++;

}

right--;

// 3. Fill bottom row (check boundary)

if (top <= bottom) {

for (int i = right; i >= left; i--) {

matrix[bottom][i] = num++;

}

bottom--;

}

// 4. Fill left column (check boundary)

if (left <= right) {

for (int i = bottom; i >= top; i--) {

matrix[i][left] = num++;

}

left++;

}

}

return matrix;

}

void freeMatrix(int\*\* matrix, int rows) {

if (!matrix) return;

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

free(matrix[i]);

}

free(matrix);

}

int main() {

int rows = 4;

int cols = 5;

printf("--- Generating a %dx%d Spiral Matrix ---\n\n", rows, cols);

// 1. Generate the matrix

int\*\* spiral = generateSpiralMatrix(rows, cols);

// 2. Print the generated matrix in grid form

printf("Constructed Matrix:\n");

printMatrix(spiral, rows, cols);

// 3. Print the spiral traversal

printf("\nSpiral Traversal Output:\n");

printSpiralOrder(spiral, rows, cols);

// 4. Clean up allocated memory

freeMatrix(spiral, rows);

return 0;

}

