Lab-Report

Course Title: Data Structure Lab Report

Course Code: CSE-135

Name: Pallab Mondol

ID: 0242220005101380

Section: 63-E

Department of : CSE

Daffodil International University

Date of Submission: 08/06/23

Overview

The "Data Structure Lab" course is designed to provide students with a comprehensive understanding of data structures and their practical application in coding. Led by the esteemed Course Instructor, Umme Sanzida Afroz, and supported by the lab assistant, Samyo Acharjee, this course offers a valuable opportunity to delve into the world of data structures.

Throughout the course, students are introduced to fundamental data structures and gain hands-on experience in implementing them using the C programming language. The following are some of the key topics covered in the lab:

1. **Array:**
   - Understanding the concept of arrays and their manipulation techniques.
2. **Singly Linked List:**
   - Exploring the structure and operations of singly linked lists, including insertion, deletion, and searching.
3. **Doubly Linked List:**
   - Examining the properties and functionalities of doubly linked lists, which enable efficient traversal in both directions.
4. **Circular Linked List:**
   - Understanding the intricacies of circular linked lists, where the last node connects back to the first node, forming a circular structure.
5. **Operations of Linked List:**
   - Gaining proficiency in various operations performed on linked lists, such as insertion, deletion, and searching for elements.
6. **Stack and its Operations:**
   - Immersing into the concept of stacks, a Last-In-First-Out (LIFO) data structure, and mastering stack operations like push, pop, and peek.
7. **Queue and its Operations:**
   - Exploring the queue data structure, which follows a First-In-First-Out (FIFO) approach, and understanding queue operations such as enqueue and dequeue.
8. **Recursion Basics:**
   - Grasping the fundamentals of recursion, a powerful technique in which a function calls itself, allowing for elegant and efficient problem-solving.
9. **Binary Search Tree Code:**
   - Diving into binary search trees, a hierarchical data structure that enables efficient searching, insertion, and deletion operations.
10. **Graph (Adjacency Matrix):**
    - Introducing graphs and their representation using an adjacency matrix, which enables the depiction of relationships between nodes.

By immersing themselves in these topics and engaging in practical implementation using the C language, students develop a solid foundation in data structures. This knowledge equips them with the skills necessary to tackle real-world coding challenges, optimize program efficiency, and explore advanced concepts in algorithms and data management.

The "Data Structure Lab" offers a dynamic and interactive learning environment where students can strengthen their coding abilities and deepen their understanding of data structures.

Array

In this program, we declare an array `array` of size `SIZE`, which is set to 5 using a `#define` directive. The array is initialized with values 10, 20, 30, 40, and 50.

The program then prints the elements of the array using a `for` loop and the `printf` function. Each element is displayed on the console.

After that, another `for` loop is used to calculate the sum of all the elements in the array. The sum is accumulated in the variable `sum`.

Finally, the program prints the sum of the array elements using the `printf` function.

To run this program, follow the steps:

```c

#include <stdio.h>

#define SIZE 5

int main() {
    int array[SIZE] = {10, 20, 30, 40, 50}; // Initialize an array of size 5 with some values
    int i, sum = 0;

    printf("Elements in the array are: ");
    for (i = 0; i < SIZE; i++) {
        printf("%d ", array[i]); // Print elements of the array
    }

    for (i = 0; i < SIZE; i++) {
        sum += array[i]; // Calculate the sum of array elements
    }

    printf("\nSum of the elements is: %d\n", sum);

    return 0;
}


```

1. Open a text editor and create a new file. For example, you can name it `array_sum.c`.
2. Copy the above code and paste it into the `array_sum.c` file.
3. Save the file.
4. Open a command prompt or terminal and navigate to the directory where you saved the `array_sum.c` file.
5. Compile the C program using a C compiler (such as GCC) with the following command:

```
gcc -o array_sum array_sum.c
```

This command compiles the `array_sum.c` file and generates an executable named `array_sum`.
6.Run the compiled program by executing the generated executable file:

- On Windows:

```
array_sum.exe
```

On macOS or Linux:

```
./array_sum
```

7.The program will be executed, and the output will be displayed in the command prompt or terminal:

```
Elements in the array are: 10 20 30 40 50
Sum of the elements is: 150
```

The program calculates the sum of the elements in the array and displays the result. You can modify the array values or its size and rerun the program to observe different outcomes.

Please note that this example is a simple demonstration of using arrays in C programming. In a complete course on arrays, you would typically cover a broader range of topics, including more complex operations, multidimensional arrays, sorting algorithms, searching algorithms, and various practical applications of arrays in problem-solving.

Singly Linked List

In this program, we define a `struct Node` representing a node in a singly linked list. Each node contains an integer data value and a pointer to the next node.

The program includes two functions:

1. `insertAtBeginning`: This function inserts a new node at the beginning of the linked list. It takes a pointer to the head of the list (`struct Node** head`) and the data value to be inserted. It creates a new node, assigns the data value to it, and updates the `next` pointer to point to the current head node. Finally, it updates the head pointer to the new node.
2. `displayList`: This function traverses the linked list starting from the head and prints the data values of each node.

In the `main` function, we initialize the head pointer as `NULL` and call the `insertAtBeginning` function three times to insert nodes with values 15, 10, and 5 at the beginning of the linked list. Then, we call the `displayList` function to print the elements of the linked list.

To run this program, follow the steps:

```c

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

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

// Function to insert a new node at the beginning of the linked list
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}

// Function to display the elements of the linked list
void displayList(struct Node* head) {
    struct Node* current = head;
    printf("Linked List: ");
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main() {
    struct Node* head = NULL;

    // Insert nodes at the beginning of the linked list
    insertAtBeginning(&head, 5);
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 15);

    // Display the linked list
    displayList(head);

    return 0;
}

```

1. Open a text editor and create a new file. For example, you can name it `singly_linked_list.c`.
2. Copy the above code and paste it into the `singly_linked_list.c` file.
3. Save the file.
4. Open a command prompt or terminal and navigate to the directory where you saved the `singly_linked_list.c` file.
5. Compile the C program using a C compiler (such as GCC) with the following command:

```
gcc -o singly_linked_list singly_linked_list.c
```

This command compiles the `singly_linked_list.c` file and generates an executable named `singly_linked_list`.

6. Run the compiled program by executing the generated executable file:

On Windows:

```
gsingly_linked_list.exe
```

On macOS or Linux:

```
./singly_linked_list
```

7. The program will be executed, and the output will be displayed in the command prompt or terminal:

```
Linked List: 15 10 5
```

The program demonstrates the basic implementation of a singly linked list, where nodes are dynamically allocated and linked together using pointers. The `insertAtBeginning` function inserts new nodes at the beginning of the list,

Doubly Linked List

In this program, we define a `struct Node` representing a node in a doubly linked list. Each node contains an integer data value, a pointer to the previous node (`prev`), and a pointer to the next node (`next`).

The program includes two functions:

1. `insertAtBeginning`: This function inserts a new node at the beginning of the doubly linked list. It takes a pointer to the head of the list (`struct Node** head`) and the data value to be inserted. It creates a new node,

assigns the data value to it, updates the `prev` pointer to `NULL`, and updates the `next` pointer to point to the current head node. If the list is not empty, it also updates the `prev` pointer of the current head node to point to the new node. Finally, it updates the head pointer to the new node.
2. `displayList`: This function traverses the doubly linked list starting from the head and prints the data values of each node.

In the `main` function, we initialize the head pointer as `NULL` and call the `insertAtBeginning` function three times to insert nodes with values 15, 10, and 5 at the beginning of the doubly linked list. Then, we call the `displayList` function to print the elements of the doubly linked list.

To run this program, follow the steps:

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

// Node structure
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Function to insert a new node at the beginning of the doubly linked list
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = *head;

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

    *head = newNode;
}

// Function to display the elements of the doubly linked list
void displayList(struct Node* head) {
    struct Node* current = head;
    printf("Doubly Linked List: ");

    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }

    printf("\n");
}

int main() {
    struct Node* head = NULL;

    // Insert nodes at the beginning of the doubly linked list
    insertAtBeginning(&head, 5);
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 15);

    // Display the doubly linked list
    displayList(head);
```

1. Open a text editor and create a new file. For example, you can name it `doubly_linked_list.c`.
2. Copy the above code and paste it into the `doubly_linked_list.c` file.
3. Save the file.

4. Open a command prompt or terminal and navigate to the directory where you saved the `doubly_linked_list.c` file.
5. Compile the C program using a C compiler (such as GCC) with the following command:

```
gcc -o doubly_linked_list doubly_linked_list.c
```

This command compiles the `doubly_linked_list.c` file and generates an executable named `doubly_linked_list`.

6. Run the compiled program by executing the generated executable file:

On Windows:

```
doubly_linked_list.exe
```

On macOS or Linux:

```
./doubly_linked_list
```

7. The program will be executed, and the output will be displayed in the command prompt or terminal:

```
Doubly Linked List: 15 10 5
```

The program demonstrates the basic implementation of a doubly linked list, where nodes are dynamically allocated and linked together using pointers. The `insertAtBeginning` function inserts new nodes at the beginning of the list by adjusting the pointers of the new node, the current head node, and the previous head node (if applicable). The `displayList` function traverses the list and prints the data values of each node.

Please note that this example provides a basic understanding of a doubly linked list. In a complete course on doubly linked lists, you would typically cover more advanced operations, such as inserting nodes at specific positions, deleting nodes, traversing in both directions, and implementing other common functionalities.

Circular Linked List

Data-Structure-Lab-(1380)

In this program, we define a `struct Node` representing a node in a circular linked list. Each node contains an integer data value and a pointer to the next node (`next`).

The program includes three functions:

1. `createNode`: This function dynamically allocates memory for a new node, assigns the data value to it, and initializes the `next` pointer to `NULL`. It returns the pointer to the newly created node.
2. `insertAtBeginning`: This function inserts a new node at the beginning of the circular linked list. If the list is empty (head is `NULL`), it assigns the newly created node as the head and makes it circular by pointing its `next` pointer to itself. Otherwise, it traverses the list to find the last node, updates its `next` pointer to the new node, and makes the new node circular by pointing its `next` pointer to the previous head node. Finally, it updates the head pointer to the new node.
3. `displayList`: This function traverses the circular linked list starting from the head and prints the data values of each node. It continues to traverse until it reaches the head node again.

In the `main` function, we initialize the head pointer as `NULL` and call the `insertAtBeginning` function three times to insert nodes with values 15, 10, and 5 at the beginning of the circular linked list. Then, we call the `displayList` function to print the elements of the circular linked list.

To run this program, follow the steps:

```c
}

// Function to insert a new node at the beginning of the circular linked list
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);

    if (*head == NULL) {
        *head = newNode;
        newNode->next = *head;
    } else {
        struct Node* current = *head;
        while (current->next != *head) {
            current = current->next;
        }
        current->next = newNode;
        newNode->next = *head;
        *head = newNode;
    }
}

// Function to display the elements of the circular linked list
void displayList(struct Node* head) {
    struct Node* current = head;
    printf("Circular Linked List: ");

    if (head != NULL) {
        do {
            printf("%d ", current->data);
            current = current->next;
        } while (current != head);
    }

    printf("\n");
}

int main() {
    struct Node* head = NULL;

    // Insert nodes at the beginning of the circular linked list
    insertAtBeginning(&head, 5);
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 15);

    // Display the circular linked list
    displayList(head);
```

1. Open a text editor and create a new file. For example, you can name it `circular_linked_list.c`.
2. Copy the above code and paste it into the `circular_linked_list.c` file.
3. Save the file.
4. Open a command prompt or terminal and navigate to the directory where you saved the `circular_linked_list.c` file.
5. Compile the C program using a C compiler (such as GCC) with the following command:

```
gcc -o circular_linked_list circular_linked_list.c
```

   This command compiles the `circular_linked_list.c` file and generates an executable named `circular_linked_list`.

6. Run the compiled program by executing the generated executable file:

   On Windows: `circular_linked_list.exe`

On macOS or Linux:
```
./circular_linked_list
```

7. The program will be executed, and the output will be displayed in the command prompt or terminal:
```
Circular Linked List: 15 10 5
```

The program demonstrates the basic implementation of a circular linked list, where nodes are dynamically allocated and linked together using pointers. The `insertAtBeginning` function inserts new nodes at the beginning of the list by adjusting the pointers of the new node, the current head node, and the last node. The `displayList` function traverses the list and prints the data values of each node.

Please note that this example provides a basic understanding of a circular linked list. In a complete course on circular linked lists, you would typically cover more advanced operations, such as inserting nodes at specific positions, deleting nodes, traversing the list in both directions, and implementing other common functionalities.

Operations of Linked List

In this program, we define a `struct Node` representing a node in a singly linked list. Each node contains an integer data value and a pointer to the next node (`next`).

The program includes several functions:

1. `createNode`: This function dynamically allocates memory for a new node, assigns the data value to it, and initializes the `next` pointer to `NULL`. It returns the pointer to the newly created node.
2. `insertAtBeginning`: This function inserts a new node at the beginning of the linked list. It creates a new node using the `createNode` function, sets its `next` pointer to the current head, and updates the head pointer to point to the new node.
3. `insertAtEnd`: This function inserts a new node at the end of the linked list. If the list is empty (head is `NULL`), it assigns the newly created node as the head. Otherwise, it traverses the list to find the last node and appends the new node to its `next` pointer.
4. `deleteNode`: This function deletes a node with a given data value from the linked list. It handles cases where the node to be deleted is the head node or an intermediate node. It traverses the list until it finds the node with the specified data value, updates the `next` pointer of the previous node to skip the node to be deleted, and frees the memory occupied by the deleted node.
5. `displayList`: This function traverses the linked list from the head node and prints the data values of each node until it reaches the end of the list.

In the `main` function, we initialize the head pointer as `NULL` and call the `insertAtBeginning` function three times to insert nodes with values 15, 10, and 5 at the beginning of the linked list. Then, we call the `displayList` function to print the elements of the linked list.

Next, we call the `insertAtEnd` function twice to insert nodes with values 20 and 25 at the end of the linked list. Again, we call the `displayList` function to print the updated elements of the linked list.

Finally, we call the `deleteNode` function to delete the node with a value of 10 from the linked list. Once more, we call the `displayList` function to print the final elements of the linked list after deletion.

To run this program, follow the steps:

```c
        return;
    }

    prev->next = current->next;
    free(current);
}

// Function to display the elements of the linked list
void displayList(struct Node* head) {
    struct Node* current = head;
    printf("Linked List: ");

    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }

    printf("\n");
}

int main() {
    struct Node* head = NULL;

    // Insert nodes at the beginning of the linked list
    insertAtBeginning(&head, 5);
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 15);

    // Display the linked list
    displayList(head);

    // Insert nodes at the end of the linked list
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 25);

    // Display the linked list
    displayList(head);

    // Delete a node from the linked list
    deleteNode(&head, 10);

    // Display the linked list
    displayList(head);

    return 0;
}
```

1. Open a text editor and create a new file. For example, you can name it **`linked_list_operations.c`**.
2. Copy the above code and paste it into the **`linked_list_operations.c`** file.
3. Save the file.
4. Open a command prompt or terminal and navigate to the directory where you saved the **`linked_list_operations.c`** file.
5. Compile the C program using a C compiler (such as GCC) with the following command:

```
gcc -o linked_list_operations linked_list_operations.c
```

This command compiles the **`linked_list_operations.c`** file and generates an executable named **`linked_list_operations`**.

06. Run the compiled program by executing the generated executable file:
    - On Windows:

```
linked_list_operations.exe
```

On macOS or Linux:

```
./linked_list_operations
```

6. The program will be executed, and the output will be displayed in the command prompt or terminal:

```
Linked List: 15 10 5
Linked List: 15 10 5 20 25
Linked List: 15 5 20 25
```

The program demonstrates the basic operations on a singly linked list, including inserting nodes at the beginning and end of the list and deleting a specific node. The `displayList` function is used to print the elements of the linked list at different stages of the program.

Please note that this example provides a basic understanding of linked list operations. In a complete course on linked lists, you would typically cover more advanced operations, such as inserting nodes at specific positions, deleting nodes at specific positions, searching for nodes, and implementing other common functionalities.

Stack and its Operations

In this program, we define a `struct Stack` to represent a stack data structure. The stack is implemented as an array `items` with a fixed maximum size of `MAX_SIZE` and a `top` variable that keeps track of the index of the top element in the stack.

The program includes several functions:

1. `initialize`: This function initializes the stack by setting the `top` index to `-1`, indicating an empty stack.
2. `isEmpty`: This function checks if the stack is empty by checking if the `top` index is `-1`.
3. `isFull`: This function checks if the stack is full by checking if the `top` index is equal to `MAX_SIZE - 1`.
4. `push`: This function pushes an element onto the stack. It first checks if the stack is full. If it is, it displays an error message. Otherwise, it increments the `top` index and assigns the element to the corresponding position in the `items` array.
5. `pop`: This function pops an element from the stack. It first checks if the stack is empty. If it is, it displays an error message and returns `-1`. Otherwise, it retrieves the element at the `top` index, decrements the `top` index, and returns the popped element.
6. `peek`: This function returns the top element of the stack without removing it. It first checks if the stack is empty. If it is, it displays an error message and returns `-1`. Otherwise, it returns the element at the `top` index.

In the `main` function, we create a `stack` object and initialize it using the `initialize` function. We then demonstrate the stack operations by pushing elements (`10`, `20`, and `30`) onto the stack using the `push` function, popping elements from the stack using the `pop` function, and getting the top element using the `peek` function.

To run this program, follow these steps:

```c
// Function to pop an element from the stack
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack Underflow. Cannot pop element from an empty stack.\n");
        return -1;
    }

    int item = stack->items[stack->top--];
    printf("Popped element %d from the stack.\n", item);
    return item;
}

// Function to get the top element of the stack without removing it
int peek(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty. Cannot peek.\n");
        return -1;
    }

    return stack->items[stack->top];
}

int main() {
    struct Stack stack;
    initialize(&stack);

    // Push elements onto the stack
    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);

    // Pop elements from the stack
    pop(&stack);
    pop(&stack);

    // Get the top element of the stack
    int topElement = peek(&stack);
    if (topElement != -1) {
        printf("Top element of the stack: %d\n", topElement);
    }

    return 0;
}
```

1. Open a text editor and create a new file. For example, you can name it **`stack_operations.c`**.
2. Copy the above code and paste it into the **`stack_operations.c`** file.
3. Save the file.
4. Open a command prompt or terminal and navigate to the directory where you saved the **`stack_operations.c`** file.
5. Compile the C program using a C compiler (such as GCC) with the following command:

```
gcc -o stack_operations stack_operations.c
```

   This command compiles the **`stack_operations.c`** file and generates an executable named **`stack_operations`**.

6. Run the compiled program by executing the generated executable file:

   On Windows:

```
stack_operations.exe
```

   On macOS or Linux:

```
./stack_operations
```

7. The program will be executed, and the output will be displayed in the command prompt or terminal:

```
Pushed element 10 onto the stack.
Pushed element 20 onto the stack.
Pushed element 30 onto the stack.
Popped element 30 from the stack.
Popped element 20 from the stack.
Top element of the stack: 10
```

The program demonstrates the basic operations of a stack, including pushing elements onto the stack, popping elements from the stack, and peeking at the top element. The program also includes checks for stack overflow and underflow conditions to handle scenarios where the stack is already full or empty.

Please note that this example provides a basic understanding of stack operations. In a complete course on stack data structures, you would typically cover more advanced operations, such as evaluating postfix expressions, implementing stack-based algorithms, and exploring additional applications of stacks in computer science.

Queue and its Operations

In this program, we define a `struct Queue` to represent a queue data structure. The queue is implemented as an array `items` with a fixed maximum size of `MAX_SIZE`, along with `front` and `rear` variables that keep track of the indices of the front and rear elements in the queue.

The program includes several functions:

1. `initialize`: This function initializes the queue by setting both the `front` and `rear` indices to `-1`, indicating an empty queue.
2. `isEmpty`: This function checks if the queue is empty by checking if the `front` index is `-1`.
3. `isFull`: This function checks if the queue is full by checking if the next index of the `rear` index (modulo `MAX_SIZE`) is equal to the `front` index.
4. `enqueue`: This function adds an element to the rear of the queue. It first checks if the queue is full. If it is, it displays an error message. Otherwise, it updates the `rear` index, assigns the element to the corresponding position in the `items` array, and adjusts the `front` index if the queue was previously empty.
5. `dequeue`: This function removes the element from the front of the queue. It first checks if the queue is empty. If it is, it displays an error message and returns `-1`. Otherwise, it retrieves the element at the `front` index, updates the `front` and `rear` indices accordingly (adjusting to an empty queue if necessary), and returns the dequeued element.
6. `front`: This function returns the front element of the queue without removing it. It first checks if the queue is empty. If it is, it displays an error message and returns `-1`. Otherwise, it returns the element at the `front` index.
7. `rear`: This function returns the rear element of the queue without removing it. It first checks if the queue is empty. If it is, it displays an error message and returns `-1`. Otherwise, it returns the element at the `rear` index.

In the `main` function, we create a `queue` object and initialize it using the `initialize` function. We then demonstrate the queue operations by enqueueing elements (`10`, `20`, and `30`) into the queue using the `enqueue` function, dequeuing elements from the queue using the `dequeue` function, and getting the front and rear elements using the `front` and `rear` functions.

To run this program, follow these steps:

```c
// Function to get the front element of the queue without removing it
int front(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot get the front element.\n");
        return -1;
    }

    return queue->items[queue->front];
}

// Function to get the rear element of the queue without removing it
int rear(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot get the rear element.\n");
        return -1;
    }

    return queue->items[queue->rear];
}

int main() {
    struct Queue queue;
    initialize(&queue);

    // Enqueue elements into the queue
    enqueue(&queue, 10);
    enqueue(&queue, 20);
    enqueue(&queue, 30);

    // Dequeue elements from the queue
    dequeue(&queue);
    dequeue(&queue);

    // Get the front and rear elements of the queue
    int frontElement = front(&queue);
    if (frontElement != -1) {
        printf("Front element of the queue: %d\n", frontElement);
    }

    int rearElement = rear(&queue);
    if (rearElement != -1) {
        printf("Rear element of the queue: %d\n", rearElement);
    }

    return 0;
}

```

1. Open a text editor and create a new file. For example, you can name it `queue_operations.c`.
2. Copy the above code and paste it into the `queue_operations.c` file.
3. Save the file.
4. Open a command prompt or terminal and navigate to the directory where you saved the `queue_operations.c` file.
5. Compile the C program using a C compiler (such as GCC) with the following command:

```
gcc -o queue_operations queue_operations.c
```

This command compiles the `queue_operations.c` file and generates an executable named `queue_operations`.

6. Run the compiled program by executing the generated executable file:

On Windows:

```
queue_operations.exe
```

On macOS or Linux:

```
./queue_operations
```

07. The program will be executed, and the output will be displayed in the command prompt or terminal:

```
Enqueued element 10 into the queue.
Enqueued element 20 into the queue.
Enqueued element 30 into the queue.
Dequeued element 10 from the queue.
Dequeued element 20 from the queue.
Front element of the queue: 30
Rear element of the queue: 30
```

The program demonstrates the basic operations of a queue, including enqueuing elements into the queue, dequeuing elements from the queue, and getting the front and rear elements. The program also includes checks for queue overflow and underflow conditions to handle scenarios where the queue is already full or empty.

Please note that this example provides a basic understanding of queue operations. In a complete course on queue data structures, you would typically cover more advanced operations, such as implementing circular queues, exploring different queue implementations, and examining additional applications of queues in computer science.

Recursion Basics

In this program, we demonstrate two common examples of recursive functions: calculating the factorial of a number and finding the nth Fibonacci number.

1. **`factorial`**: This recursive function calculates the factorial of a given number `n`. It uses a base case where the factorial of 0 is 1. In the recursive case, it multiplies `n` with the factorial of `(n-1)`. The function keeps calling itself with a smaller value until it reaches the base case and returns the final result.
2. **`fibonacci`**: This recursive function calculates the nth Fibonacci number. It has two base cases where the Fibonacci of 0 is 0 and the Fibonacci of 1 is 1. In the recursive case, it calculates the Fibonacci of `(n-1)` and `(n-2)` and adds them together. The function keeps calling itself with smaller values until it reaches the base case and returns the final Fibonacci number.

In the `main` function, we prompt the user to enter a positive integer. We then calculate and display the factorial of the number using the `factorial` function and the nth Fibonacci number using the `fibonacci` function.

To run this program, follow these steps:

```c
#include <stdio.h>

// Recursive function to calculate the factorial of a number
int factorial(int n) {
    // Base case: factorial of 0 is 1
    if (n == 0) {
        return 1;
    }

    // Recursive case: factorial of n is n multiplied by factorial of (n-1)
    return n * factorial(n - 1);
}

// Recursive function to calculate the nth Fibonacci number
int fibonacci(int n) {
    // Base cases: fibonacci of 0 is 0 and fibonacci of 1 is 1
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    }

    // Recursive case: fibonacci of n is the sum of fibonacci of (n-1) and fibonacci of (n-2)
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int num;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    // Calculate and display the factorial of the number
    int fact = factorial(num);
    printf("Factorial of %d is %d.\n", num, fact);

    // Calculate and display the nth Fibonacci number
    int fib = fibonacci(num);
    printf("The %dth Fibonacci number is %d.\n", num, fib);

    return 0;
}
```

1. Open a text editor and create a new file. For example, you can name it `recursion_basics.c`.
2. Copy the above code and paste it into the `recursion_basics.c` file.
3. Save the file.
4. Open a command prompt or terminal and navigate to the directory where you saved the `recursion_basics.c` file.
5. Compile the C program using a C compiler (such as GCC) with the following command

```
gcc -o recursion_basics recursion_basics.c
```

   This command compiles the `recursion_basics.c` file and generates an executable named `recursion_basics`.

6. Run the compiled program by executing the generated executable file:

   On Windows:

```
recursion_basics.exe
```

On macOS or Linux:

```
./recursion_basics
```

7. The program will be executed, and it will prompt you to enter a positive integer. After entering the number, the program will display the factorial and the nth Fibonacci number based on the provided input:

```
Enter a positive integer: 5
Factorial of 5 is 120.
The 5th Fibonacci number is 5.
```

The program demonstrates the basics of recursion by calculating the factorial and Fibonacci numbers recursively. It showcases the power of recursion to solve problems by breaking them down into smaller, simpler subproblems.

Binary Search Tree

In this program, we demonstrate the implementation of a binary search tree (BST) and perform an inorder traversal to display the elements in sorted order.

The `Node` structure represents a node in the BST. It has three members: `data` to store the node's value, `left` to point to the left child node, and `right` to point to the right child node.

The `createNode` function creates a new node with the specified data and returns a pointer to it.

The `insert` function inserts a new node with the given data into the BST. It follows the property of a BST, where the nodes in the left subtree are less than the parent node, and the nodes in the right subtree are greater than or equal to the parent node. The function recursively traverses the BST to find the appropriate position for insertion.

The `inorderTraversal` function performs an inorder traversal of the BST. It visits the left subtree, prints the value of the current node, and then visits the right subtree. This results in the elements being displayed in ascending order.

In the `main` function, we create an empty BST (`root` is initially set to `NULL`) and insert several nodes with different values. We then perform an inorder traversal using the `inorderTraversal` function and display the elements in sorted order.

To run this program, follow these steps:

1. Open a text editor and create a new file. For example, you can name it `binary_search_tree.c`.
2. Copy the above code and paste it into the `binary_search_tree.c` file.
3. Save the file.
4. Open a command prompt or terminal and navigate to the directory where you saved the `binary_search_tree.c` file.
5. Compile the C program using a C compiler (such as GCC) with the following command:

```
gcc -o binary_search_tree binary_search_tree.c
```

   This command compiles the `binary_search_tree.c` file and generates an executable named `binary_search_tree`.

6. Run the compiled program by executing the generated executable file:

   On Windows:

```
binary_search_tree.exe
```

On macOS or Linux:

```
./binary_search_tree
```

07. The program will be executed, and it will display the elements of the binary search tree in sorted order:

```
Inorder traversal of the binary search tree: 20 30 40 50 60 70 80
```

The program demonstrates the basic functionality of a binary search tree, including node insertion and inorder traversal. It showcases the property of a BST where the elements are stored in a sorted order, allowing efficient searching, insertion, and deletion operations.

Graph(Adjacency Matrix)

In this program, we demonstrate the implementation of a graph using an adjacency matrix.

The `addEdge` function takes the adjacency matrix and two nodes as input and adds an edge between the two nodes by setting the corresponding matrix entries to 1. Since the graph is undirected, we set both `adjacencyMatrix[startNode][endNode]` and `adjacencyMatrix[endNode][startNode]` to 1.

The `displayGraph` function takes the adjacency matrix and the number of nodes as input and displays the matrix representation of the graph. It iterates over the matrix and prints the values of each entry.

In the `main` function, we prompt the user to enter the number of nodes and edges in the graph. We create an adjacency matrix of size `MAX_NODES` and initialize it to all zeros. Then, we prompt the user to enter the edges as node pairs and use the `addEdge` function to update the adjacency matrix accordingly. Finally, we call the `displayGraph` function to display the adjacency matrix.

To run this program, follow these steps:

1. Open a text editor and create a new file. For example, you can name it `adjacency_matrix_graph.c`.
2. Copy the above code and paste it into the `adjacency_matrix_graph.c` file.
3. Save the file.
4. Open a command prompt or terminal and navigate to the directory where you saved the `adjacency_matrix_graph.c` file.
5. Compile the C program using a C compiler (such as GCC) with the following command:

```
gcc -o adjacency_matrix_graph adjacency_matrix_graph.c
```

   This command compiles the `adjacency_matrix_graph.c` file and generates an executable named `adjacency_matrix_graph`.

6. Run the compiled program by executing the generated executable file:

   On Windows:

   ```
   adjacency_matrix_graph.exe
   ```

   On macOS or Linux:

   ```
   ./adjacency_matrix_graph
   ```

7. The program will be executed, and it will prompt you to enter the number of nodes and edges in the graph, followed by the edges as node pairs. After that, it will display the adjacency matrix representation of the graph based on the provided input.

```
Enter the number of nodes in the graph: 4
Enter the number of edges in the graph: 5
Enter the edges (node pairs):
0 1
0 2
1 2
2 3
3 0
Adjacency Matrix:
0 1 1 0
1 0 1 0
1 1 0 1
0 0 1 0
```

The program demonstrates the implementation of a graph using an adjacency matrix. The adjacency matrix provides a compact representation of the graph's connections, with 1s indicating the presence of an edge between two nodes and 0s indicating no edge. This representation allows for efficient access to node relationships and is commonly used in graph algorithms and applications.