C programming

Pointers and DataStructures

Kiran VVN 2/5/25 C programming

Foreword

Learning pointers and data structures in C is a journey that many programmers find both challenging and rewarding. The intricacies of memory management, the abstraction of linked lists, and the complexity of trees can often seem like insurmountable obstacles. This compilation is dedicated to all the programmers out there who aspire to master these foundational concepts but find themselves struggling along the way.

To those who have spent countless hours debugging segmentation faults, who have puzzled over NULL pointers, and who have felt the frustration of cryptic compiler errors—we understand your challenges. This guide is crafted with you in mind, aiming to demystify pointers and data structures through clear explanations, step-by-step instructions, and visual illustrations.

We believe that with patience, practice, and the right resources, anyone can grasp these essential programming constructs. This collection of programs and explanations is more than just code; it's a bridge to deeper understanding and proficiency in C programming.

May this guide serve as a beacon on your learning path, illuminating the concepts that once seemed elusive. Embrace the journey, persist through the difficulties, and celebrate each moment of clarity. Remember, every expert was once a beginner who kept learning.

Happy coding, and here's to your success in conquering pointers and data structures!

Contents

List of Figures

Compiling the programs in this book

gcc pointers_intro.c -o pointers_intro

gcc pointers_to_pointers.c -o pointers_to_pointers

gcc function_pointers.c -o function_pointers

gcc dynamic_memory_allocation.c -o dynamic_memory_allocation

gcc singly_linked_list.c -o singly_linked_list

gcc doubly_linked_list.c -o doubly_linked_list

gcc circular_linked_list.c -o circular_linked_list

gcc binary_tree.c -o binary_tree

gcc avl_tree.c -o avl_tree

gcc message_queue_simulation.c -o message_queue_simulation

gcc sender.c -o sender -lrt

gcc receiver.c -o receiver -lrt

gcc queue_linked_list.c -o queue_linked_list

Introduction

This document presents a complete reference to data structures with C programming language. It is intended to be a foundation for strong programming skill development with C. This book is for anyone who wishes to understand pointers either as a beginner or someone who is familiar with pointers to anyone who is just brushing up. The code in this book is generic and should work with any compiler that supports ANSI C standard. For the purpose of this book, we have used, UBUNTU and gcc compiler.

1. Basic Pointers – Introduction

Pointers are variables that hold memory addresses of other variables. They are powerful tools in C, enabling dynamic memory management, efficient array handling, and the creation of complex data structures like linked lists and trees.

Concept: A pointer is a variable that stores the memory address of another variable. Think of it as a signpost pointing to a location in memory.

Imagine a house (variable) with an address (memory location). The pointer is like a mailbox containing that address.

Copy

```
+----------------+          +----------------+
|    int num     |          |    int *ptr    |
|----------------|          |----------------|
|   Value: 42    |<---+     |   Address of   |
| Address: 0x001 |    |     |    num: 0x001  |
+----------------+    |     +----------------+
                      |
                      |
                      |
               +------+------+
               |   Pointer   |
               |   ptr -->   |
               |   0x001     |
               +-------------+
```

- `num` is an integer variable stored at memory address `0x001`.
- `ptr` is a pointer holding the address of `num`.

Figure 1 Pointers

Example Program: Understanding Basic Pointers

```c
#include <stdio.h>

int main() {

    int num = 42;       // An integer variable

    int *ptr = &num;    // Pointer to the integer variable


    printf("Value of num: %d\n", num);

    printf("Address of num: %p\n", (void*)&num);

    printf("Value of ptr (Address of num): %p\n", (void*)ptr);

    printf("Value pointed to by ptr: %d\n", *ptr);


    *ptr = 100;         // Modifying num using the pointer

    printf("New value of num: %d\n", num);


    return 0;

}
```

Explanation:

- **Declaration:** int *ptr; declares a pointer to an integer.
- **Initialization:** ptr = # assigns the address of num to ptr.
- **Dereferencing:** *ptr allows access to the value stored at the memory location ptr points to.
- **Modification:** Changing *ptr modifies the original variable num.

Key Concepts:

- **Address-of Operator (&):** Retrieves the memory address of a variable.

- **Dereference Operator (*):** Accesses or modifies the value at the pointed address.
- **Pointer Declaration:** Syntax type *pointer_name; where type is the data type of the variable the pointer points to.

2. Advanced Pointers

Advanced pointer concepts delve deeper into pointers' capabilities, including pointers to pointers, pointer arithmetic, function pointers, and dynamic memory allocation.

Copy

```
+----------------+
|     int **pp   |
|----------------|
| Address of *p  |---------+
|    (0x003)     |         |
+----------------+         |
                           |
                    +------v------+
                    |    int *p   |
                    |-------------|
                    | Address of  |---------+
                    |    num      |         |
                    |  (0x001)    |         |
                    +-------------+         |
                                            |
                                     +------v------+
                                     |    int num  |
                                     |-------------|
                                     |   Value:    |
                                     |     42      |
                                     +-------------+
```

- `pp` points to `p`, which in turn points to `num`.

Figure 2 - Pointers to Pointers

a) Pointers to Pointers

Pointers can also point to other pointers, creating multiple levels of indirection.

Example Program: Pointers to Pointers

```
#include <stdio.h>


int main() {
    int value = 2023;
    int *ptr = &value;      // Pointer to integer
    int **ptr2 = &ptr;      // Pointer to pointer


    printf("Value: %d\n", value);
    printf("Value via ptr: %d\n", *ptr);
    printf("Value via ptr2: %d\n", **ptr2);


    return 0;
}
```

Explanation:

- ptr stores the address of value.
- ptr2 stores the address of ptr.
- **ptr2 accesses value through two levels of indirection.

b) Function Pointers

Function pointers store the address of functions, allowing dynamic function calls and callbacks.

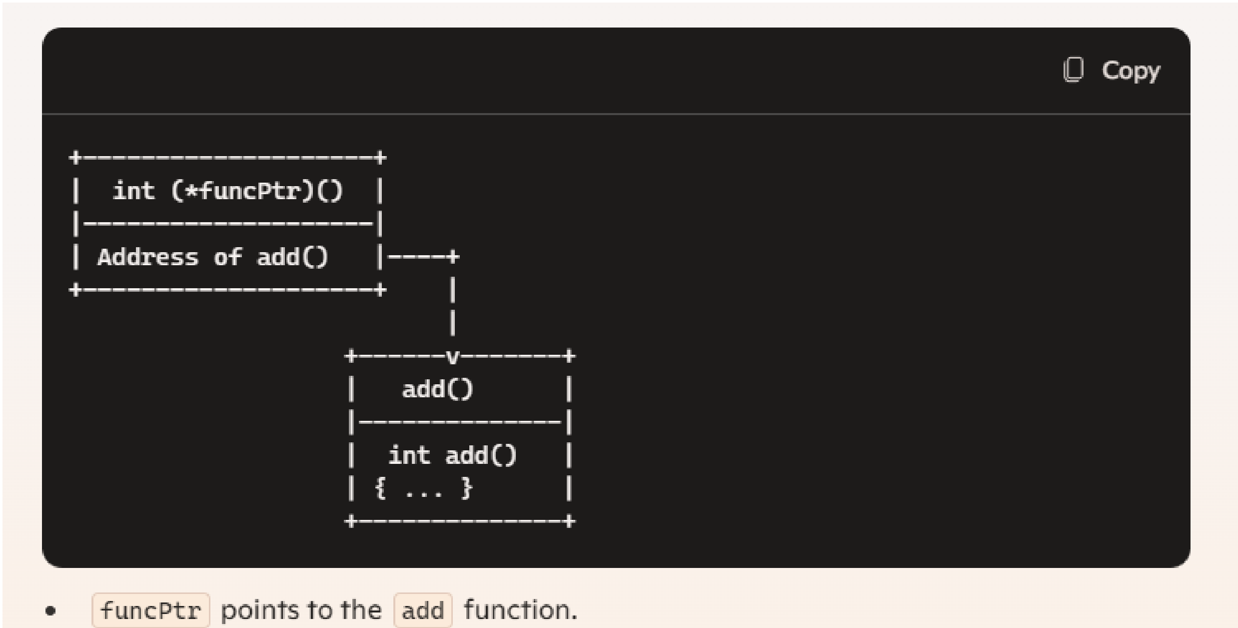


Figure 3 = Function Pointers

Example Program: Function Pointers

```
#include <stdio.h>

// Functions to perform arithmetic operations

int add(int a, int b) { return a + b; }

int multiply(int a, int b) { return a * b; }

int main() {

  // Function pointer declaration

  int (*operation)(int, int);

  // Assigning the 'add' function to the pointer

  operation = &add;

  printf("Addition: %d\n", operation(5, 3));

  // Assigning the 'multiply' function to the pointer

  operation = &multiply;
```

```
    printf("Multiplication: %d\n", operation(5, 3));


    return 0;
}
```

Explanation:

- int (*operation)(int, int); declares a pointer to a function taking two int and returning an int.
- The function pointer operation can point to different functions matching the signature.

c) Dynamic Memory Allocation

Dynamic memory allocation allows allocating memory at runtime using pointers, essential for creating flexible data structures.

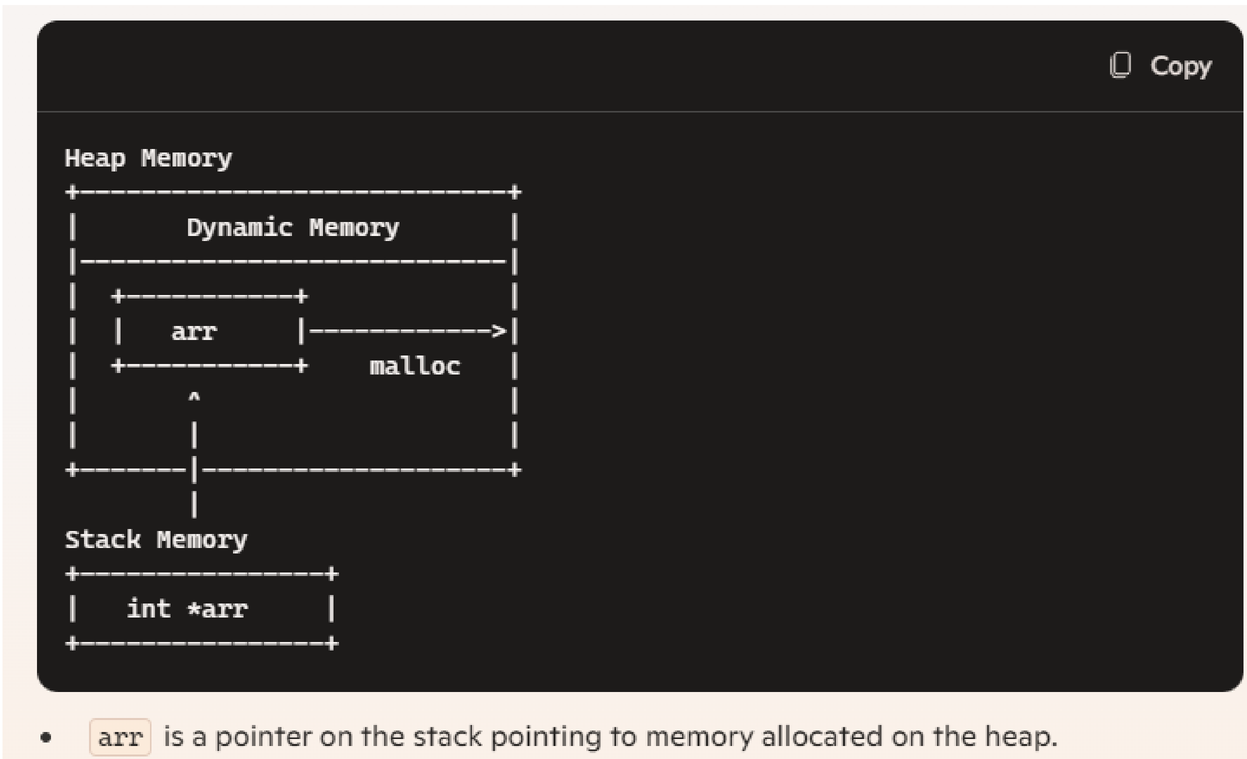


Figure 4 - Dynamic Memory Allocation

Example Program: Dynamic Memory Allocation with malloc

c

```
#include <stdio.h>

#include <stdlib.h>


int main() {

    int n = 5;

    int *arr = (int*)malloc(n * sizeof(int));  // Allocating memory for an array of 5 integers
```

```
  if (arr == NULL) {
    printf("Memory allocation failed!\n");
    return 1;
  }

  // Initializing and printing the array
  for (int i = 0; i < n; i++) {
    arr[i] = (i + 1) * 10;
    printf("arr[%d] = %d\n", i, arr[i]);
  }

  free(arr);  // Freeing allocated memory

  return 0;
}
```

Explanation:

- malloc allocates specified bytes of memory and returns a pointer to the beginning.
- Always check if malloc returns NULL to handle allocation failures.
- Use free to deallocate memory and prevent memory leaks.

3. Linked Lists – Basic to Advanced

Linked lists are dynamic data structures consisting of nodes connected through pointers. They are essential for implementing other complex structures.

a) Singly Linked List

In a singly linked list, each node contains data and a pointer to the next node.

Copy

```
Head
 |
 v
+--------+     +--------+     +--------+     +--------+
|  Data  |     |  Data  |     |  Data  |     |  Data  |
|   10   |     |   20   |     |   30   |     |   40   |
+--------+     +--------+     +--------+     +--------+
| Next---+---->| Next---+---->| Next---+---->| Next-->NULL
+--------+     +--------+     +--------+     +--------+
```

- Each node contains data and a reference to the next node.

Figure 5 = Single Linked List

Example Program: Singly Linked List Implementation

```
#include <stdio.h>

#include <stdlib.h>


// Define the node structure

typedef struct Node {

   int data;

   struct Node *next;

} Node;


// Function to create a new node

Node* createNode(int data) {

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

   if (!newNode) {

      printf("Memory allocation error!\n");

      exit(1);
```

```
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}


// Function to insert a node at the beginning
void insertAtBeginning(Node **head, int data) {
    Node *newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
}


// Function to display the list
void displayList(Node *head) {
    Node *temp = head;
    printf("Singly Linked List: ");
    while (temp) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
// Main function
int main() {
```

```
    Node *head = NULL;

    insertAtBeginning(&head, 30);

    insertAtBeginning(&head, 20);

    insertAtBeginning(&head, 10);


    displayList(head);

    // Free memory (not shown for brevity)

    return 0;

}
```

Explanation:

- **Node Structure:** Contains data and next pointer.
- **Insertion:** New nodes are inserted at the beginning for simplicity.
- **Traversal:** Displaying the list by following the next pointers.

b) Doubly Linked List

In a doubly linked list, each node has pointers to both the next and previous nodes.

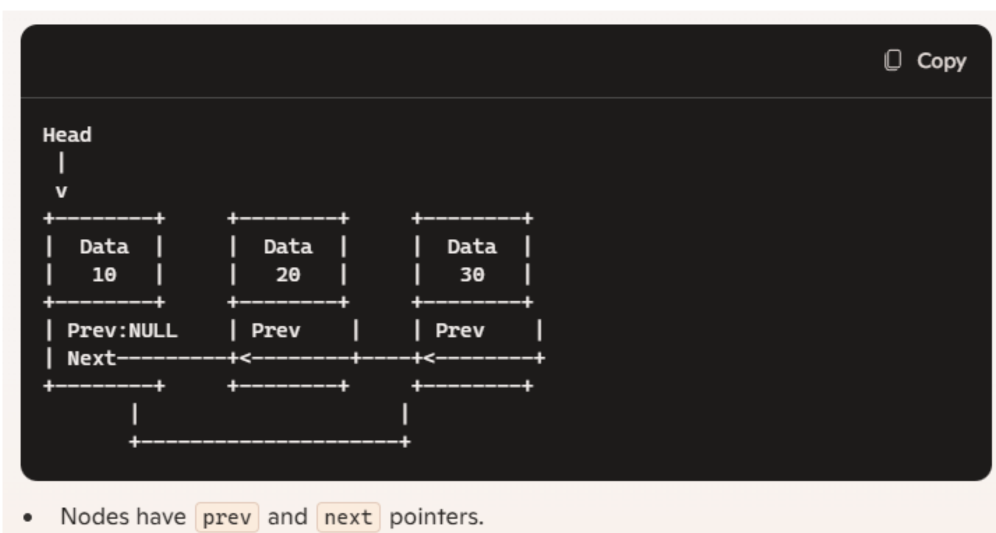


Figure 6 Doubly LInked List

Example Program: Doubly Linked List Implementation

c

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

// Define the node structure
typedef struct DNode {
   int data;
   struct DNode *prev;
   struct DNode *next;
} DNode;

// Function to create a new node
DNode* createDNode(int data) {
   DNode *newNode = (DNode*)malloc(sizeof(DNode));
   if (!newNode) {
      printf("Memory allocation error!\n");
      exit(1);
   }
   newNode->data = data;
   newNode->prev = newNode->next = NULL;
   return newNode;
}

// Function to insert at the end
```

```
void insertAtEnd(DNode **head, int data) {
   DNode *newNode = createDNode(data);
   if (*head == NULL) {
      *head = newNode;
      return;
   }
   DNode *temp = *head;
   while (temp->next)
      temp = temp->next;
   temp->next = newNode;
   newNode->prev = temp;
}


// Function to display the list forward
void displayForward(DNode *head) {
   DNode *temp = head;
   printf("Doubly Linked List Forward: ");
   while (temp) {
      printf("%d <=> ", temp->data);
      temp = temp->next;
   }
   printf("NULL\n");
}


// Function to display the list backward
```

```
void displayBackward(DNode *head) {
   DNode *temp = head;
   if (!temp) return;
   while (temp->next)
      temp = temp->next;
   printf("Doubly Linked List Backward: ");
   while (temp) {
      printf("%d <=> ", temp->data);
      temp = temp->prev;
   }
   printf("NULL\n");
}


// Main function
int main() {
   DNode *head = NULL;
   insertAtEnd(&head, 10);
   insertAtEnd(&head, 20);
   insertAtEnd(&head, 30);

   displayForward(head);
   displayBackward(head);

   // Free memory (not shown for brevity)
   return 0;
```

}

Explanation:

- **Prev and Next Pointers:** Each node points to both its predecessor and successor.
- **Bidirectional Traversal:** Can traverse the list forward and backward.
- **Insertion at End:** Nodes are added at the end for demonstration.

c) Circular Linked List

Circular linked lists have the last node pointing back to the first node, forming a circle.

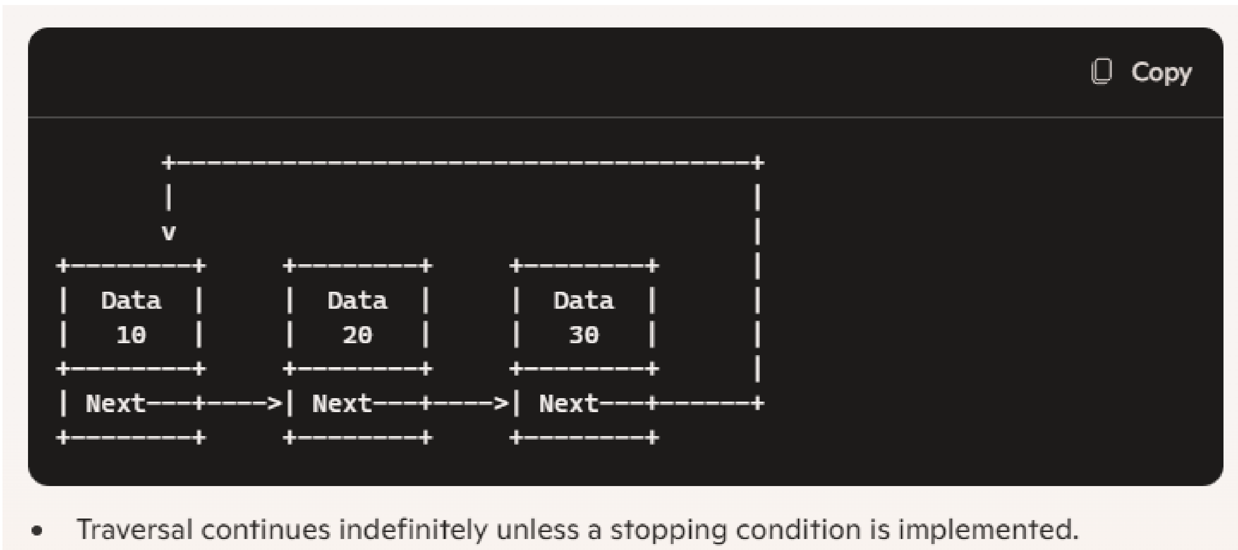


Figure 7 Circular Linked List

Example Program: Circular Singly Linked List

c

```c
#include <stdio.h>

#include <stdlib.h>



// Define the node structure

typedef struct CNode {

   int data;

   struct CNode *next;
```

```
} CNode;


// Function to create a new node
CNode* createCNode(int data) {
  CNode *newNode = (CNode*)malloc(sizeof(CNode));
  if (!newNode) {
    printf("Memory allocation error!\n");
    exit(1);
  }
  newNode->data = data;
  newNode->next = newNode; // Points to itself initially
  return newNode;
}


// Function to insert into circular linked list
void insert(CNode **last, int data) {
  CNode *newNode = createCNode(data);
  if (*last == NULL) {
    *last = newNode;
  } else {
    newNode->next = (*last)->next;
    (*last)->next = newNode;
    *last = newNode;
  }
}
```

```
// Function to display the list
void displayList(CNode *last) {
    if (!last) {
        printf("List is empty.\n");
        return;
    }
    CNode *temp = last->next;
    printf("Circular Linked List: ");
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != last->next);
    printf("(back to start)\n");
}

// Main function
int main() {
    CNode *last = NULL;
    insert(&last, 10);
    insert(&last, 20);
    insert(&last, 30);

    displayList(last);
```

```
  // Free memory (not shown for brevity)

  return 0;

}
```

Explanation:

- **Circularity:** The next pointer of the last node points to the first node.
- **Traversal:** Starts at last->next and loops until it reaches the starting point again.

4. Binary Trees – Basic to Advanced

Binary trees are hierarchical structures where each node has up to two children. They are foundational for many algorithms and applications.

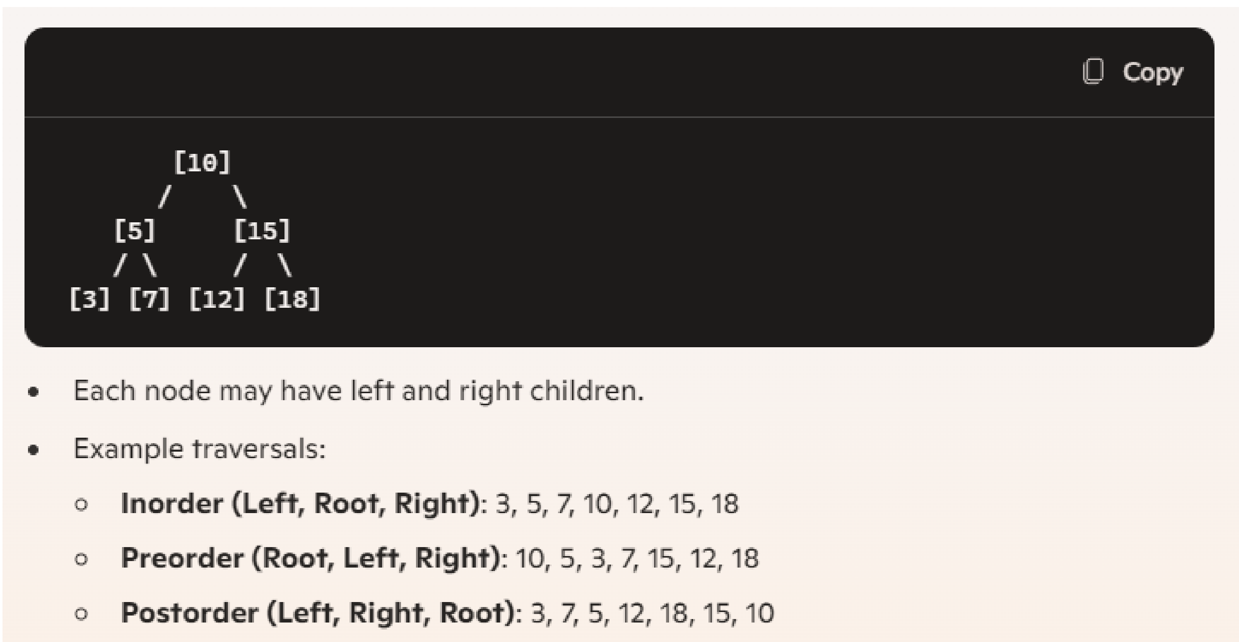


Figure 8 - Binary Tree

Example Program: Binary Tree Implementation and Traversal

```
#include <stdio.h>

#include <stdlib.h>


// Define the node structure
```

```
typedef struct BNode {
   int data;
   struct BNode *left;
   struct BNode *right;
} BNode;

// Function to create a new node
BNode* createBNode(int data) {
   BNode *newNode = (BNode*)malloc(sizeof(BNode));
   if (!newNode) {
      printf("Memory allocation error!\n");
      exit(1);
   }
   newNode->data = data;
   newNode->left = newNode->right = NULL;
   return newNode;
}

// Functions for tree traversals
void inorderTraversal(BNode *root) {
   if (root) {
      inorderTraversal(root->left);
      printf("%d ", root->data);
      inorderTraversal(root->right);
   }
```

```
}

void preorderTraversal(BNode *root) {
  if (root) {
    printf("%d ", root->data);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
  }
}

void postorderTraversal(BNode *root) {
  if (root) {
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d ", root->data);
  }
}

// Main function
int main() {
  // Creating a simple binary tree
  BNode *root = createBNode(1);
  root->left = createBNode(2);
  root->right = createBNode(3);
  root->left->left = createBNode(4);
```

```
    root->left->right = createBNode(5);


    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");


    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");


    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");


    // Free memory (not shown for brevity)
    return 0;
}
```

Explanation:

- **Node Structure:** Contains data, left, and right pointers.
- **Tree Traversals:**
    - **Inorder (Left, Root, Right):** Produces sorted output for BST.
    - **Preorder (Root, Left, Right):** Useful for copying the tree.
    - **Postorder (Left, Right, Root):** Used for deleting the tree.

Advanced Binary Tree Topics:

- **Binary Search Trees (BST):** Left child < parent < right child.

- **Insertion and Deletion Algorithms:** Maintaining tree properties.
- **Balanced Trees:** AVL and Red-Black trees ensure operations remain efficient.
- **Tree Traversal Algorithms:** Breadth-First Search (BFS) using queues.

5. AVL Trees

AVL trees are self-balancing binary search trees. They maintain a balance factor to ensure operations remain in O(log n) time.

```
Initial Insertion:

        [30]
       /    \
    [20]   [40]
    /
 [10]

After Inserting [25]:

        [30]                         [30]
       /    \                       /    \
    [20]   [40]       --->       [25]   [40]
    /  \                         /   \
 [10] [25]                    [20] [10]

Rotation occurs to maintain balance.
```

- **Balance Factor**: Height of left subtree minus height of right subtree.
- **Rotations**: Performed when balance factor is not within -1 to 1.

Figure 9 AVL Tree

Example Program: AVL Tree Implementation

Implementing AVL trees involves complex rotations to maintain balance. Here's a concise implementation demonstrating insertion and balancing.

c

```
#include <stdio.h>

#include <stdlib.h>


// Node structure for AVL tree

typedef struct AVLNode {

   int data;

   struct AVLNode *left;

   struct AVLNode *right;

   int height;

} AVLNode;


// Function prototypes

AVLNode* insert(AVLNode* node, int data);

AVLNode* createNode(int data);

int getHeight(AVLNode* node);

int getBalance(AVLNode* node);

AVLNode* rightRotate(AVLNode* y);

AVLNode* leftRotate(AVLNode* x);

int max(int a, int b);


// Create a new node

AVLNode* createNode(int data) {

   AVLNode* node = (AVLNode*)malloc(sizeof(AVLNode));

   if (!node) {

      printf("Memory allocation error!\n");
```

```
        exit(1);
    }
    node->data = data;
    node->left = node->right = NULL;
    node->height = 1;  // New node initially at height 1
    return node;
}

// Get node height
int getHeight(AVLNode* node) {
    return node ? node->height : 0;
}

// Get maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Right rotate subtree rooted with y
AVLNode* rightRotate(AVLNode* y) {
    AVLNode* x = y->left;
    AVLNode* T2 = x->right;

    // Rotation
    x->right = y;
```

```
    y->left = T2;

    // Update heights
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

    return x;  // New root
}

// Left rotate subtree rooted with x
AVLNode* leftRotate(AVLNode* x) {
    AVLNode* y = x->right;
    AVLNode* T2 = y->left;

    // Rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

    return y;  // New root
}
```

```
// Get balance factor
int getBalance(AVLNode* node) {
   return node ? getHeight(node->left) - getHeight(node->right) : 0;
}


// Insert a node and balance the tree
AVLNode* insert(AVLNode* node, int data) {
   // Normal BST insertion
   if (!node)
      return createNode(data);


   if (data < node->data)
      node->left  = insert(node->left, data);
   else if (data > node->data)
      node->right = insert(node->right, data);
   else  // Duplicate data not allowed
      return node;


   // Update height
   node->height = 1 + max(getHeight(node->left), getHeight(node->right));


   // Balance factor
   int balance = getBalance(node);


   // Balancing the tree
```

```
    // Left Left Case
    if (balance > 1 && data < node->left->data)
        return rightRotate(node);
    // Right Right Case
    if (balance < -1 && data > node->right->data)
        return leftRotate(node);
    // Left Right Case
    if (balance > 1 && data > node->left->data) {
        node->left =  leftRotate(node->left);
        return rightRotate(node);
    }
    // Right Left Case
    if (balance < -1 && data < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;  // Return unchanged node
}

// Inorder traversal
void inorderTraversal(AVLNode* root) {
    if (root) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
```

```
        inorderTraversal(root->right);
    }
}


// Main function
int main() {
    AVLNode* root = NULL;


    // Inserting nodes
    int arr[] = {20, 4, 15, 70, 50, 100, 85};
    int n = sizeof(arr)/sizeof(arr[0]);
    for(int i = 0; i < n; i++)
        root = insert(root, arr[i]);


    printf("Inorder traversal of the AVL tree:\n");
    inorderTraversal(root);
    printf("\n");


    // Free memory (not shown for brevity)
    return 0;
}
```

Explanation:

- **Height Maintenance:** Each node keeps track of its height.
- **Balance Factor:** Calculated as the difference between the heights of left and right subtrees.

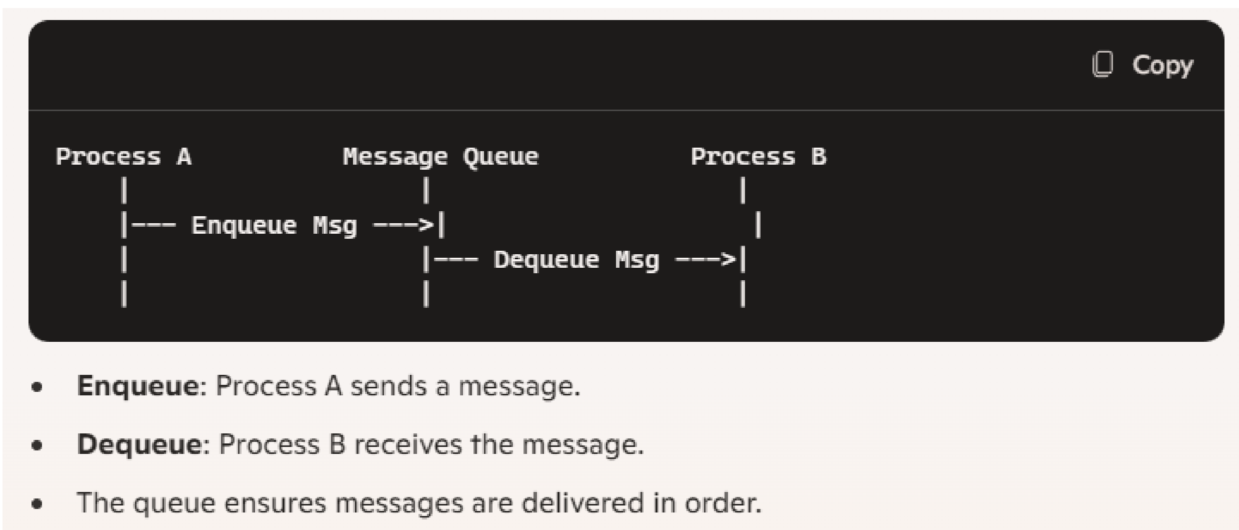
- **Rotations:** Performed to rebalance the tree when necessary.

Key Concepts:

- **Self-Balancing:** Ensures operations remain efficient (O(log n)).
- **Rotations:** Critical for maintaining tree balance.
- **Applications:** Used in databases and file systems for quick data retrieval.

6. Message Queues

Message queues facilitate communication between processes, allowing them to exchange data asynchronously.

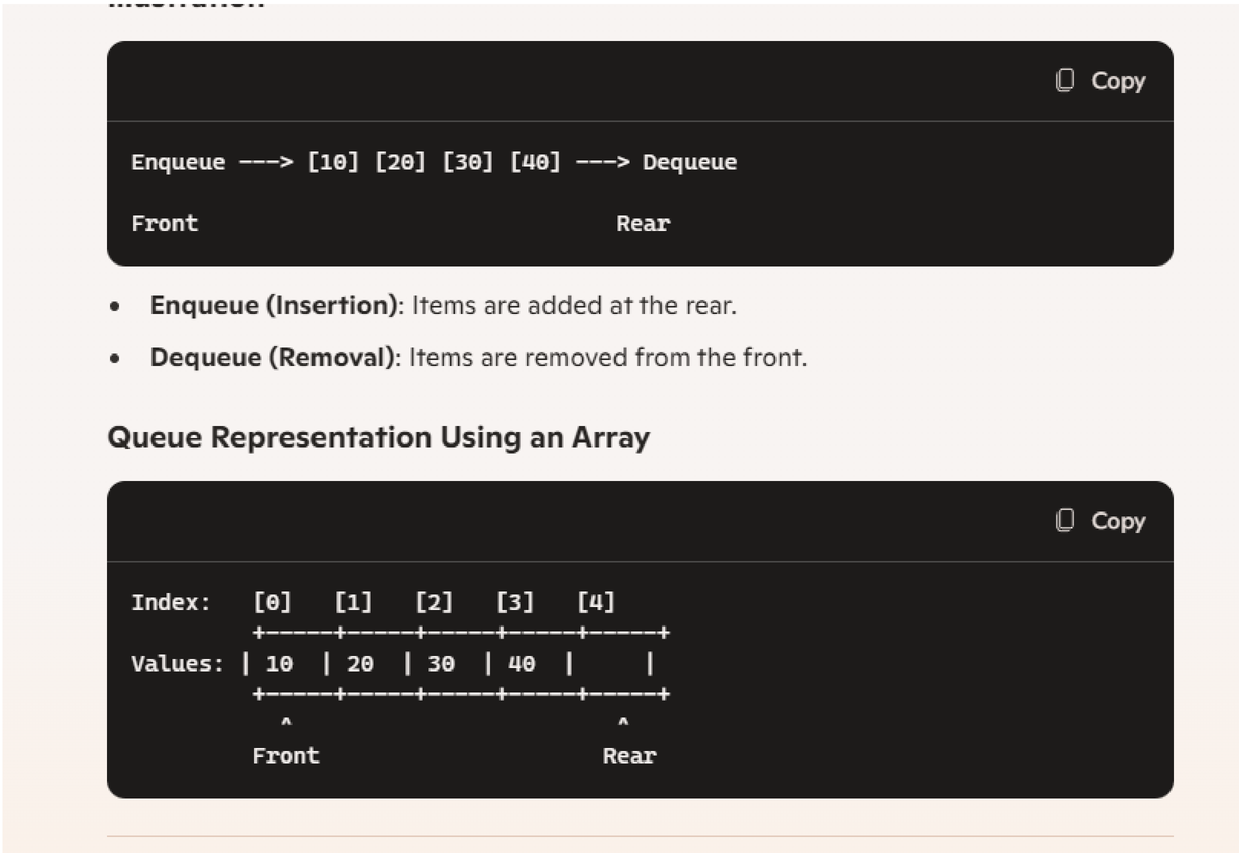


Copy

```
Process A          Message Queue          Process B
    |                    |                    |
    |--- Enqueue Msg --->|                     |
    |                    |--- Dequeue Msg --->|
    |                    |                    |
```

- **Enqueue**: Process A sends a message.
- **Dequeue**: Process B receives the message.
- The queue ensures messages are delivered in order.

Figure 10 Message Queue for processes



```
Enqueue ---> [10] [20] [30] [40] ---> Dequeue

Front                             Rear
```

- **Enqueue (Insertion)**: Items are added at the rear.
- **Dequeue (Removal)**: Items are removed from the front.

Queue Representation Using an Array

```
Index:   [0]   [1]   [2]   [3]   [4]
        +-----+-----+-----+-----+-----+
Values: | 10  | 20  | 30  | 40  |     |
        +-----+-----+-----+-----+-----+
          ^                        ^
        Front                     Rear
```

Figure 11 - Queue representation using an Array

Simulated Message Queue Using a Circular Buffer

While system-level message queues involve inter-process communication (IPC) mechanisms, we can simulate a message queue using a circular buffer.

Example Program: Simple Message Queue Simulation

c

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define QUEUE_SIZE 5

#define MESSAGE_LENGTH 100
```

```
// Message Queue Structure
typedef struct {
    char messages[QUEUE_SIZE][MESSAGE_LENGTH];
    int front;
    int rear;
    int count;
} MessageQueue;

// Initialize the message queue
void initQueue(MessageQueue *mq) {
    mq->front = 0;
    mq->rear = -1;
    mq->count = 0;
}

// Enqueue a message
void enqueue(MessageQueue *mq, const char *message) {
    if (mq->count == QUEUE_SIZE) {
        printf("Queue is full. Cannot enqueue message.\n");
        return;
    }
    mq->rear = (mq->rear + 1) % QUEUE_SIZE;
    strncpy(mq->messages[mq->rear], message, MESSAGE_LENGTH);
    mq->count++;
```

```
}


// Dequeue a message
void dequeue(MessageQueue *mq, char *message) {
  if (mq->count == 0) {
    printf("Queue is empty. Cannot dequeue message.\n");
    return;
  }
  strncpy(message, mq->messages[mq->front], MESSAGE_LENGTH);
  mq->front = (mq->front + 1) % QUEUE_SIZE;
  mq->count--;
}


// Display the queue
void displayQueue(MessageQueue *mq) {
  if (mq->count == 0) {
    printf("Queue is empty.\n");
    return;
  }
  printf("Message Queue:\n");
  int index = mq->front;
  for (int i = 0; i < mq->count; i++) {
    printf("%d: %s\n", i + 1, mq->messages[index]);
    index = (index + 1) % QUEUE_SIZE;
  }
```

```
}


int main() {
    MessageQueue mq;
    initQueue(&mq);


    enqueue(&mq, "Message One");
    enqueue(&mq, "Message Two");
    enqueue(&mq, "Message Three");


    displayQueue(&mq);


    char msg[MESSAGE_LENGTH];
    dequeue(&mq, msg);
    printf("Dequeued: %s\n", msg);


    displayQueue(&mq);


    return 0;
}
```

Explanation:

- **Circular Buffer:** Simulates a queue where the rear and front wrap around.
- **Enqueue/Dequeue Operations:** Add and remove messages from the queue.
- **Usage Scenario:** Demonstrates basic message passing within a single process.

7. Queues

Queues are linear structures following the First-In-First-Out (FIFO) principle.

Example Program: Queue Implementation Using Linked List

Implementing a queue using a linked list allows dynamic memory management without fixed size limitations.

c

```
#include <stdio.h>

#include <stdlib.h>


// Define the node structure

typedef struct QNode {

   int data;

   struct QNode *next;

} QNode;


// Define the queue structure

typedef struct Queue {

   QNode *front;

   QNode *rear;

} Queue;


// Function to create a new node

QNode* createQNode(int data) {

   QNode *newNode = (QNode*)malloc(sizeof(QNode));

   if (!newNode) {

      printf("Memory allocation error!\n");
```

```
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to initialize the queue
void initQueue(Queue *q) {
    q->front = q->rear = NULL;
}

// Enqueue function
void enqueue(Queue *q, int data) {
    QNode *newNode = createQNode(data);
    if (q->rear == NULL) {
        q->front = q->rear = newNode;
        return;
    }
    q->rear->next = newNode;
    q->rear = newNode;
}

// Dequeue function
int dequeue(Queue *q) {
```

```
    if (q->front == NULL) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }
    QNode *temp = q->front;
    int data = temp->data;
    q->front = q->front->next;

    // If front becomes NULL, reset rear to NULL
    if (q->front == NULL)
        q->rear = NULL;

    free(temp);
    return data;
}

// Display the queue
void displayQueue(Queue *q) {
    if (!q->front) {
        printf("Queue is empty.\n");
        return;
    }
    QNode *temp = q->front;
    printf("Queue: ");
    while (temp) {
```

```
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function
int main() {
    Queue q;
    initQueue(&q);

    enqueue(&q, 100);
    enqueue(&q, 200);
    enqueue(&q, 300);

    displayQueue(&q);

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

    displayQueue(&q);

    // Free remaining nodes (not shown for brevity)
    return 0;
}
```

Explanation:

- **Dynamic Size:** The queue can grow and shrink as needed.
- **Enqueue at Rear:** New elements are added at the rear.
- **Dequeue from Front:** Elements are removed from the front.

Example Program: POSIX Message Queue

- **Sender**: Opens a message queue and sends a message.
- **Receiver**: Receives the message from the queue.
- **mq_open**, **mq_send**, **mq_receive**, **mq_close**, and **mq_unlink** are functions from the mqueue.h library.
- ***gcc sender.c -o sender -lrt***
- ***gcc receiver.c -o receiver -lrt***

Sender.c

```
// sender.c

#include <stdio.h>

#include <stdlib.h>

#include <mqueue.h>

#include <string.h>


#define QUEUE_NAME "/my_queue"

#define MAX_SIZE 1024


int main() {

  mqd_t mq;

  char buffer[MAX_SIZE];
```

```
    // Open the queue
    mq = mq_open(QUEUE_NAME, O_WRONLY | O_CREAT, 0644, NULL);
    if (mq == -1) {
        perror("mq_open");
        exit(1);
    }

    // Send a message
    printf("Enter a message: ");
    fgets(buffer, MAX_SIZE, stdin);
    if (mq_send(mq, buffer, strlen(buffer) + 1, 0) == -1) {
        perror("mq_send");
        exit(1);
    }

    // Close the queue
    mq_close(mq);
    return 0;
}
```

Receiver.c

```
// receiver.c
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
```

```
#define QUEUE_NAME "/my_queue"

#define MAX_SIZE 1024


int main() {

    mqd_t mq;

    char buffer[MAX_SIZE];


    // Open the queue

    mq = mq_open(QUEUE_NAME, O_RDONLY);

    if (mq == -1) {

        perror("mq_open");

        exit(1);

    }


    // Receive the message

    ssize_t bytes_read = mq_receive(mq, buffer, MAX_SIZE, NULL);

    if (bytes_read >= 0) {

        printf("Received: %s\n", buffer);

    } else {

        perror("mq_receive");

        exit(1);

    }


    // Close and unlink the queue
```

```
    mq_close(mq);

    mq_unlink(QUEUE_NAME);

    return 0;

}
```

Queues using Linked Lists

Quell.c

```
#include <stdio.h>

#include <stdlib.h>


// Node structure

typedef struct QNode {

    int data;

    struct QNode *next;

} QNode;


// Queue structure

typedef struct Queue {

    QNode *front;

    QNode *rear;

} Queue;


// Create a new node

QNode* createQNode(int data) {

    QNode *newNode = (QNode*)malloc(sizeof(QNode));
```

```
    if (!newNode) {
        printf("Memory allocation error!\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

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

// Enqueue operation
void enqueue(Queue *q, int data) {
    QNode *newNode = createQNode(data);
    if (q->rear == NULL) {
        q->front = q->rear = newNode;
        return;
    }
    q->rear->next = newNode;
    q->rear = newNode;
}
```

```
// Dequeue operation
int dequeue(Queue *q) {
   if (q->front == NULL) {
      printf("Queue is empty. Cannot dequeue.\n");
      return -1;
   }
   QNode *temp = q->front;
   int data = temp->data;
   q->front = q->front->next;

   // If front becomes NULL, reset rear to NULL
   if (q->front == NULL)
      q->rear = NULL;

   free(temp);
   return data;
}

// Display the queue
void displayQueue(Queue *q) {
   if (!q->front) {
      printf("Queue is empty.\n");
      return;
   }
   QNode *temp = q->front;
```

```
    printf("Queue: ");
    while (temp) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Main function
int main() {
    Queue q;
    initQueue(&q);

    enqueue(&q, 100);
    enqueue(&q, 200);
    enqueue(&q, 300);

    displayQueue(&q);

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

    displayQueue(&q);
    // Memory cleanup (not shown)
    return 0;
}
```

Appendix A – Full System with Shared Memory and Linked List

A C program that reads user input to create a linked list and then adds it to shared memory using System V shared memory mechanisms. This allows multiple processes to access the linked list by attaching to the same shared memory segment.

System Overview

Plaintext Copy

```
+-----------------------------+
|         User Input          |
|-----------------------------|
| - Enters integers to form   |
|   a linked list             |
+--------------+--------------+
               |
               v
+-----------------------------+
|   Process 1 (Producer)      |
|-----------------------------|
| - Creates linked list       |
| - Allocates shared memory   |
| - Copies linked list to     |
|   shared memory             |
| - Detaches from shared      |
|   memory                    |
+--------------+--------------+
               |
     Shared Memory Segment (IPC)
               |
+--------------v--------------+
|                             |
|    Shared Memory Segment    |
|-----------------------------|
| - Stores linked list nodes  |
| - Accessible by multiple    |
|   processes                 |
|                             |
+--------------+--------------+
               |
               v
+-----------------------------+
|   Process 2 (Consumer)      |
|-----------------------------|
| - Attaches to shared memory|
| - Accesses linked list      |
| - Traverses and prints      |
|   linked list               |
| - Detaches from shared      |
|   memory                    |
+-----------------------------+
```

Figure 12 - System Overview

Program Overview

- **Linked List Creation**: The program reads integers from the user and creates a singly linked list.
- **Shared Memory Allocation**: It allocates shared memory to store the linked list.
- **Storing Linked List in Shared Memory**: Copies the linked list into the shared memory segment.
- **Demonstration**: Shows how another process can access and traverse the linked list from shared memory.

Important Notes

- **System V Shared Memory**: We're using System V shared memory (shmget, shmat, shmdt, shmctl).
- **Compiler Flags**: Compile with -pthread if needed.
- **Permissions**: You may need appropriate permissions to create shared memory segments.
- **Cleanup**: Ensure that shared memory is properly detached and removed after use to prevent memory leaks.

Process Flow Diagram

1. **Process 1 (Producer)**:
    - **Start**
    - ↓
    - **Read User Input**
    - ↓
    - **Create Linked List**
    - ↓
    - **Calculate Shared Memory Size**
    - ↓

- **Allocate Shared Memory (**shmget**)**
- ↓
- **Attach to Shared Memory (**shmat**)**
- ↓
- **Copy Linked List to Shared Memory**
- ↓
- **Detach from Shared Memory (**shmdt**)**
- ↓
- **End**

2. **Process 2 (Consumer)**:
    - **Start**
    - ↓
    - **Generate Same Key**
    - ↓
    - **Locate Shared Memory Segment (**shmget**)**
    - ↓
    - **Attach to Shared Memory (**shmat**)**
    - ↓
    - **Access Linked List from Shared Memory**
    - ↓
    - **Traverse and Print Linked List**
    - ↓
    - **Detach from Shared Memory (**shmdt**)**
    - ↓
    - **End**

Shared Memory Structure and Contents

Plaintext Copy

```
Shared Memory Contents:

+--------------------------------------------------------------+
| SharedMemory Structure                                       |
|--------------------------------------------------------------|
| - head: Pointer to the first node in shared memory           |
|                                                              |
| Linked List Nodes Stored Sequentially in Memory:             |
|                                                              |
| +------------+    +------------+    +------------+           |
| | Node 1     |    | Node 2     |    | Node 3     |   ...     |
| |------------|    |------------|    |------------|           |
| | data: 10   |    | data: 20   |    | data: 30   |           |
| | next: *----|--> | next: *----|--> | next: NULL |           |
| +------------+    +------------+    +------------+           |
|                                                              |
+--------------------------------------------------------------+
```

- **Node Structure**:
  - `data`: Integer value.
  - `next`: Pointer to the next node (adjusted for shared memory).

Program Code

Filename: shared_memory_linked_list.c

```c
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/types.h>

#include <unistd.h>


// Define the node structure

typedef struct Node {

   int data;

   struct Node* next;

} Node;


// Define a structure for shared memory that includes the head pointer

typedef struct SharedMemory {

   Node* head;

   char mem[1];  // Placeholder for start of variable-sized data

} SharedMemory;


// Function to create a new node
```

```
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        perror("malloc");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}


// Function to read user input and create a linked list
Node* createLinkedList() {
    Node* head = NULL;
    Node* temp = NULL;
    int data;
    char choice;


    printf("Create linked list (Enter integers). Press 'n' to stop.\n");
    while (1) {
        printf("Enter data: ");
        if (scanf("%d", &data) != 1) {
            // Clear invalid input
            while (getchar() != '\n');
            printf("Invalid input. Please enter an integer.\n");
```

```
            continue;
        }

        Node* newNode = createNode(data);

        if (head == NULL) {
            head = newNode;
            temp = head;
        } else {
            temp->next = newNode;
            temp = temp->next;
        }

        printf("Do you want to add another node? (y/n): ");
        getchar();  // Consume newline character
        choice = getchar();
        if (choice != 'y' && choice != 'Y') {
            break;
        }
    }
    return head;
}

// Function to calculate the size needed for shared memory
size_t calculateSize(Node* head) {
```

```
    size_t size = 0;
    Node* temp = head;
    while (temp) {
        size += sizeof(Node);
        temp = temp->next;
    }
    return size;
}


// Function to copy the linked list into a memory buffer
void copyLinkedListToBuffer(Node* head, void* buffer) {
    Node* temp = head;
    Node* prev = NULL;
    Node* current;
    void* ptr = buffer;

    while (temp) {
        current = (Node*)ptr;
        current->data = temp->data;
        if (prev) {
            prev->next = current;
        }
        prev = current;

        temp = temp->next;
```

```
        ptr += sizeof(Node);
    }
    if (prev) {
        prev->next = NULL;
    }
}

// Function to print the linked list
void printLinkedList(Node* head) {
    Node* temp = head;
    printf("Linked List: ");
    while (temp) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    key_t key;
    int shmid;
    size_t shmsize;
    SharedMemory* shm_ptr;

    // Step 1: Create the linked list
```

```
Node* head = createLinkedList();


// Step 2: Calculate the shared memory size needed
size_t list_size = calculateSize(head);
shmsize = sizeof(SharedMemory) - sizeof(char) + list_size;


// Step 3: Create shared memory segment
key = ftok("shmfile", 65);  // Generate a unique key
if (key == -1) {
   perror("ftok");
   exit(1);
}


shmid = shmget(key, shmsize, 0666 | IPC_CREAT);
if (shmid == -1) {
   perror("shmget");
   exit(1);
}


// Step 4: Attach to the shared memory
shm_ptr = (SharedMemory*)shmat(shmid, NULL, 0);
if (shm_ptr == (void*)-1) {
   perror("shmat");
   exit(1);
}
```

```
// Step 5: Copy the linked list into shared memory
void* data_ptr = shm_ptr->mem;
copyLinkedListToBuffer(head, data_ptr);

// Set the head pointer relative to shared memory
shm_ptr->head = (Node*)data_ptr;

printf("\nLinked list has been copied to shared memory.\n");

// For demonstration, let's print the linked list from shared memory
printf("\nTraversing linked list from shared memory:\n");
printLinkedList(shm_ptr->head);

// Step 6: Detach from shared memory
if (shmdt(shm_ptr) == -1) {
   perror("shmdt");
   exit(1);
}

// Note: In actual use, another process would attach to the shared memory
// and access the linked list using the known key.

// Step 7: Clean up local linked list
Node* temp;
```

```
  while (head) {
    temp = head;
    head = head->next;
    free(temp);
  }

  // Optionally remove the shared memory segment
  // Uncomment the following lines if you want to delete the shared memory
  /*
  if (shmctl(shmid, IPC_RMID, NULL) == -1) {
    perror("shmctl");
    exit(1);
  }
  printf("Shared memory segment deleted.\n");
  */

  return 0;
}
```

What is the program doing

1. Linked List Creation

- createLinkedList(): Reads integers from the user to create a singly linked list.
    - Prompts the user to enter integers.
    - Continues until the user decides to stop.
    - Uses the createNode() function to create new nodes.

2. Shared Memory Allocation

- *Generating a Key*: Uses ftok() to generate a unique key for the shared memory segment.
    - ftok("shmfile", 65): Generates a key based on the file shmfile and project identifier 65.
    - If shmfile doesn't exist, create an empty file in the same directory:

      touch shmfile
- *Calculating Size*: The required shared memory size is the total size of all nodes plus the size of the shared memory structure without the placeholder array.
    - Uses calculateSize() to compute the total size needed for the linked list.
- *Creating Shared Memory Segment*: Uses shmget() to create the shared memory segment.
    - shmget(key, size, 0666 | IPC_CREAT): Creates a shared memory segment with read and write permissions.

3. Copying Linked List to Shared Memory

- **Attaching to Shared Memory**: Uses shmat() to attach the shared memory segment to the process's address space.
- **Copying Data**: copyLinkedListToBuffer() copies the linked list into the shared memory buffer.
    - Adjusts pointers within the shared memory to maintain the linked list structure.
    - Sets the head pointer in shared memory to point to the copied linked list.

4. Traversing the Linked List from Shared Memory

- **Printing the Linked List**: printLinkedList() traverses the linked list starting from the head pointer in shared memory.
    - Demonstrates that the linked list is accessible from shared memory.

5. Detaching and Cleaning Up

- **Detaching**: Uses shmdt() to detach from the shared memory segment.

- **Cleanup**: Frees the locally allocated linked list nodes.

6. Shared Memory Removal

- **Optional Cleanup**: The shared memory segment remains in the system unless explicitly removed.
    - Uncomment the shmctl() call to remove the shared memory segment after use.
    - Alternatively, manage the shared memory segment's lifecycle as needed for your application.

Compiling and Running the Program

1. Compilation

Compile the program using GCC:

gcc shared_memory_linked_list.c -o shared_memory_linked_list

2. Create the Key File

Ensure the key file used in ftok() exists:

touch shmfile

3. Running the Program

Execute the program:

./shared_memory_linked_list

4. Accessing Shared Memory from Another Process

You can create another program that attaches to the shared memory segment using the same key and reads the linked list.

Example:

c

```
// access_shared_memory.c

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <sys/types.h>


// Define the node structure (same as before)

typedef struct Node {

   int data;

   struct Node* next;

} Node;


typedef struct SharedMemory {

   Node* head;

   char mem[1];

} SharedMemory;


// Function to print the linked list

void printLinkedList(Node* head) {
```

```
    Node* temp = head;
    printf("Linked List from shared memory: ");
    while (temp) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    key_t key;
    int shmid;
    SharedMemory* shm_ptr;

    // Generate the same key
    key = ftok("shmfile", 65);
    if (key == -1) {
        perror("ftok");
        exit(1);
    }

    // Locate the shared memory segment
    shmid = shmget(key, 0, 0666);
    if (shmid == -1) {
        perror("shmget");
```

```
        exit(1);
    }

    // Attach to the shared memory
    shm_ptr = (SharedMemory*)shmat(shmid, NULL, 0);
    if (shm_ptr == (void*)-1) {
        perror("shmat");
        exit(1);
    }

    // Print the linked list from shared memory
    printLinkedList(shm_ptr->head);

    // Detach from shared memory
    if (shmdt(shm_ptr) == -1) {
        perror("shmdt");
        exit(1);
    }

    return 0;
}
```

Compile:

gcc access_shared_memory.c -o access_shared_memory

Run:

./access_shared_memory

Things to look at ?

Consistency of Data Structures:

- Ensure that the Node structure is identical in all programs accessing the shared memory.
- Any change in the structure requires recompilation of all programs accessing the shared memory.

Pointer Validity:

- Pointers within the shared memory segment are only valid within the context of the shared memory.
- Since absolute addresses may differ between processes, using relative pointers (offsets) can be safer.

Using Relative Pointers (Advanced):

- For production code, consider storing offsets relative to the shared memory base instead of raw pointers.
- This ensures that the pointers are valid across different processes.

Synchronization:

- If multiple processes will write to the shared memory, implement synchronization mechanisms (e.g., semaphores) to prevent race conditions.

Error Handling:

- Robust error handling is crucial for production code.
- Check return values of all system calls and handle errors appropriately.

Security:

- Be cautious with permissions (0666 allows all users to read and write to the shared memory segment).

- Adjust permissions as needed.

Cleanup:

- Remember to remove the shared memory segment when it's no longer needed to prevent resource leaks.
- Use shmctl(shmid, IPC_RMID, NULL) to delete the shared memory segment.

Step-by-Step Instructions to Run the Shared Memory Linked List System

Overview

The system consists of two programs:

1. **Producer Program (**shared_memory_linked_list.c**):**
   - Reads user input to create a linked list.
   - Allocates shared memory.
   - Copies the linked list into the shared memory segment.
2. **Consumer Program (**access_shared_memory.c**):**
   - Attaches to the existing shared memory segment.
   - Accesses the linked list stored in shared memory.
   - Traverses and prints the linked list.

How to run this System ?

Create and navigate to the working directory

mkdir shared_memory_example

cd shared_memory_example

Create the key file

touch shmfile

Create and edit the producer program

nano shared_memory_linked_list.c

```
# (Paste the producer code, save and exit)
# Create and edit the consumer program
nano access_shared_memory.c
# (Paste the consumer code, save and exit)
# Compile the producer program
gcc shared_memory_linked_list.c -o shared_memory_linked_list

# Compile the consumer program
gcc access_shared_memory.c -o access_shared_memory

# Run the producer program
./shared_memory_linked_list
# (Follow prompts to input data)

# Run the consumer program
./access_shared_memory

# List shared memory segments
ipcs -m

# Remove shared memory segment manually (if needed)
ipcrm -m <shmid>
```