303105201 - Design of Data Structures

UNIT-3 Linked List

->Linked List

Definition

Representation of linked lists in Memory

Memory allocation

Garbage Collection.

-> Linked list operations:

Traversing

Searching

Insertion

Deletion

->Doubly Linked lists

->Circular linked lists

->Header linked list

->Linked Stacks and Queues

->Applications of Linked list

Linked List

Definition:

A linked list is a linear data structure in which elements are stored in nodes. Each node contains:

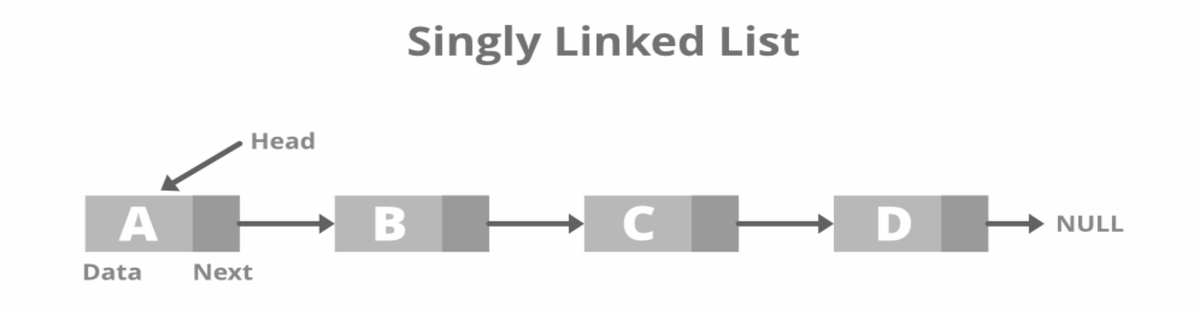
Data: The value stored in the node.

Pointer (or reference): A link to the next node in the sequence.

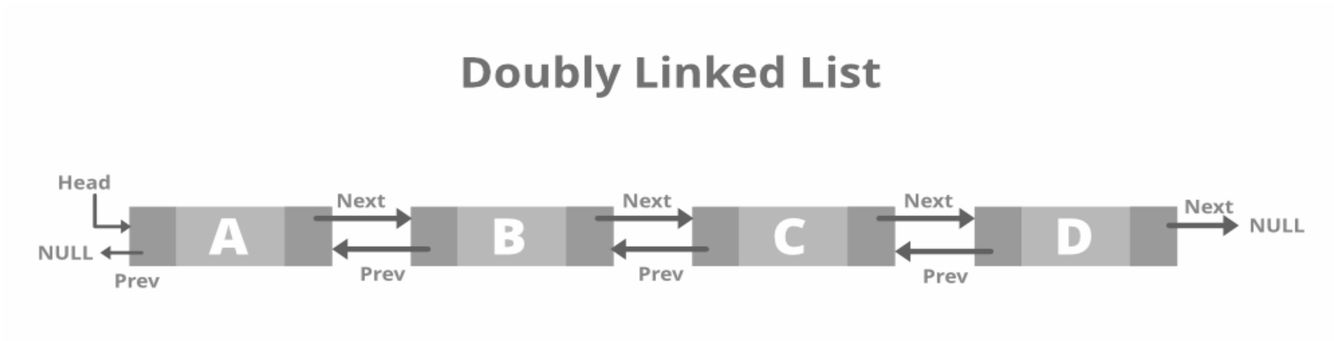
Unlike arrays, linked lists do not require contiguous memory locations. Instead, each element points to the next, forming a chain. Linked lists are dynamic, meaning they can grow and shrink in size as needed, which makes them flexible for various operations.

Types of Linked Lists

Singly Linked List: Each node points to the next node in the sequence.

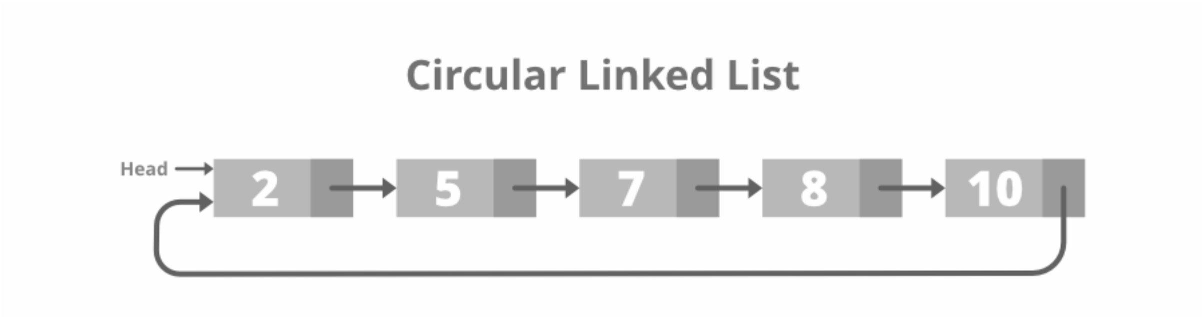


Doubly Linked List: Each node points to both the next and the previous node

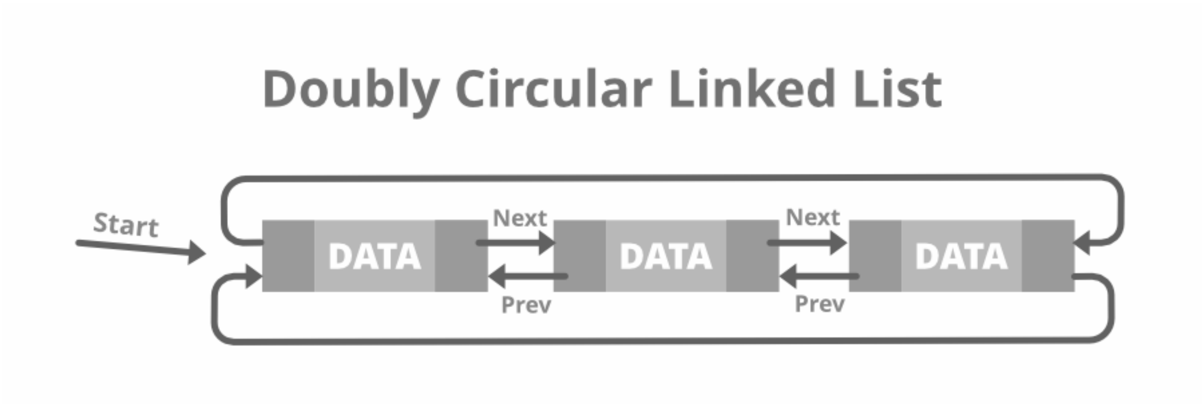


<u>Circular Linked List:</u> The last node points back to the first node, forming a circle.

Single circular:



Double circular:



Representation of Linked Lists in Memory

A linked list is a data structure consisting of nodes where each node contains data and a reference (or link) to the next node in the sequence.

<u>Singly Linked List</u>

In a singly linked list, each node points to the next node and the last node points to NULL.

Example:

```
#include <stdio.h>

#include <stdlib.h>

// Definition of the node

struct Node {

   int data;

   struct Node* next;

};

// Function to create a new node

struct Node* createNode(int data) {

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

   if (!newNode) {

      printf("Memory error\n");

      return NULL;

   }

   newNode->data = data;

   newNode->next = NULL;

   return newNode;

}

// Function to print the linked list
```

```
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int main() {
    struct Node* head = createNode(1);
    head->next = createNode(2);
    head->next->next = createNode(3);
    printList(head);
    return 0;
}
```

__Diagram:__

head -> [1 | next] -> [2 | next] -> [3 | next] -> NULL

__Output:__

```
1 -> 2 -> 3 -> NULL

--------------------------------
Process exited after 0.00571 seconds with return value 0
Press any key to continue . . .
```

Memory Allocation

Memory allocation in C is managed using functions from the C standard library, mainly malloc, calloc, realloc, and free.

malloc and free:

- malloc (memory allocation) allocates a specified number of bytes and returns a pointer to the allocated memory.
- free deallocates the memory previously allocated by malloc.

Example:

```
#include <stdio.h>

#include <stdlib.h>

int main() {

  int* ptr;

  int n, i;

  // Number of elements

  n = 5;

  // Dynamically allocate memory using malloc

  ptr = (int*)malloc(n * sizeof(int));

  // Check if the memory has been successfully allocated by malloc

  if (ptr == NULL) {

    printf("Memory not allocated.\n");

    return 1;

  }

  // Memory has been successfully allocated

  printf("Memory successfully allocated using malloc.\n");

  // Initialize the elements

  for (i = 0; i < n; ++i) {

    ptr[i] = i + 1;
```

```
    }
    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d ", ptr[i]);
    }
    // Free the memory
    free(ptr);
    printf("\nMemory successfully freed.\n");
    return 0;
}
```

Output:

```
Memory successfully allocated using malloc.
The elements of the array are: 1 2 3 4 5
Memory successfully freed.

--------------------------------
Process exited after 0.006216 seconds with return value 0
Press any key to continue . . .
```

Garbage Collection

C does not have automatic garbage collection like higher-level languages (e.g., Java, Python). Memory management in C is manual, which means the programmer is responsible for allocating and freeing memory.

Manual Garbage Collection

In C, memory allocated with malloc, calloc, or realloc must be explicitly freed using the free function to avoid memory leaks.

Example with Linked List

```
#include <stdio.h>
#include <stdlib.h>
// Definition of the node
struct Node {
    int data;
    struct Node* next;
};
// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    if (!newNode) {
        printf("Memory error\n");
        return NULL;
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
// Function to free the linked list
void freeList(struct Node* head) {
    struct Node* temp;
    while (head != NULL) {
        temp = head;
```

```
        head = head->next;
        free(temp);
    }
}
int main() {
    struct Node* head = createNode(1);
    head->next = createNode(2);
    head->next->next = createNode(3);
    // Print the linked list
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
    // Free the linked list
    freeList(head);
    return 0;
}
```

Output:

```
1 -> 2 -> 3 -> NULL

--------------------------------
Process exited after 0.005536 seconds with return value 0
Press any key to continue . . .
```

Linked list operations:

Operations of Singly Linked List in C

It including common operations like insertion, deletion, and traversal.

Node Structure

Each node in the linked list is represented by a structure:

```
#include <stdio.h>

#include <stdlib.h>

// Definition of the node structure

struct Node {

   int data;

   struct Node* next;

};
```

Functions for Linked List Operations

1. *Insertion at the Beginning:*

- Insert a new node at the start of the linked list.

```
void insertAtBeginning(struct Node** head, int newData) {

   // Allocate memory for the new node

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

   // Assign data to the new node

   newNode->data = newData;

   // Make the new node point to the current head

   newNode->next = *head;

   // Move the head to point to the new node

   *head = newNode;
```

```
}
```

2. *<u>Insertion at the End:</u>*

- Insert a new node at the end of the linked list.

```
void insertAtEnd(struct Node** head, int newData) {
   // Allocate memory for the new node
   struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
   // Assign data to the new node
   newNode->data = newData;
   newNode->next = NULL;
   // If the linked list is empty, make the new node the head
   if (*head == NULL) {
      *head = newNode;
      return;
   }
   // Otherwise, traverse to the end and insert the new node
   struct Node* temp = *head;
   while (temp->next != NULL) {
      temp = temp->next;
   }
   temp->next = newNode;
}
```

3. *<u>Deletion of a Node:</u>*

- Delete a node with a specific value.

```
void deleteNode(struct Node** head, int key) {
    struct Node* temp = *head;
    struct Node* prev = NULL;
    // If the head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key) {
        *head = temp->next;
        free(temp);
        return;
    }
    // Search for the key to be deleted
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }
    // If the key was not found
    if (temp == NULL) return;
    // Unlink the node from the linked list
    prev->next = temp->next;
    free(temp);
}
```

4. *__Traversal:__*

- Printing of all the elements in the linked list.

```
void printList(struct Node* node) {
    while (node != NULL) {
```

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

5. *Main Function*

- A main function to demonstrate the linked list operations:

```
int main() {
    struct Node* head = NULL;
    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtBeginning(&head, 5);
    insertAtEnd(&head, 30);
    printf("Linked list: ");
    printList(head);
    deleteNode(&head, 20);
    printf("Linked list after deletion: ");
    printList(head);
    return 0;
}
```

Output:

```
Linked list: 5 -> 10 -> 20 -> 30 -> NULL
Linked list after deletion: 5 -> 10 -> 30 -> NULL

--------------------------------
Process exited after 0.005972 seconds with return value 0
Press any key to continue . . .
```

Explanation

Node Definition: A node structure is defined with data and a next pointer.

Insertion Functions: Functions to insert nodes at the beginning and end of the list.

Deletion Function: Function to delete a node by key.

Traversal Function: Function to print the list.

Main Function: Demonstrates insertion, deletion, and printing of the list.

Doubly Linked List

A doubly linked list is a type of linked list in which each node contains a data part and two pointers, one pointing to the next node and another pointing to the previous node.

Example:

```
#include <stdio.h>

#include <stdlib.h>

// Definition of the node

struct Node {

   int data;

   struct Node* next;

   struct Node* prev;
```

```c
};
// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    if (!newNode) {
        printf("Memory error\n");
        return NULL;
    }
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}
// Function to print the doubly linked list
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int main() {
```

```
    struct Node* head = createNode(1);
    head->next = createNode(2);
    head->next->prev = head;
    head->next->next = createNode(3);
    head->next->next->prev = head->next;
    printList(head);
    return 0;
}
```

Output:

```
1 <-> 2 <-> 3 <-> NULL

--------------------------------
Process exited after 0.004254 seconds with return value 0
Press any key to continue . . .
```

Circular Linked List

In a circular linked list, the last node points to the first node, forming a circle.

Example:

```
#include <stdio.h>
#include <stdlib.h>
// Definition of the node
struct Node {
    int data;
    struct Node* next;
```

```
};
// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    if (!newNode) {
        printf("Memory error\n");
        return NULL;
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
// Function to print the circular linked list
void printList(struct Node* head) {
    if (head == NULL) return;
    struct Node* temp = head;
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(head)\n");
}
int main() {
```

```
    struct Node* head = createNode(1);
    head->next = createNode(2);
    head->next->next = createNode(3);
    head->next->next->next = head;
    printList(head);
    return 0;
}
```

Output:

```
1 -> 2 -> 3 -> (head)

--------------------------------
Process exited after 0.004347 seconds with return value 0
Press any key to continue . . .
```

Header Linked List

A header linked list is a variant where a special node, called the header, is placed at the beginning of the list. The header node typically does not contain meaningful data.

Example:

```
#include <stdio.h>
#include <stdlib.h>
// Definition of the node
struct Node {
    int data;
    struct Node* next;
```

```
};
// Function to create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
   if (!newNode) {
      printf("Memory error\n");
      return NULL;
   }
newNode->data = data;
   newNode->next = NULL;
   return newNode;
}
// Function to print the header linked list
void printList(struct Node* head) {
   struct Node* temp = head->next; // Skip the header node
   while (temp != NULL) {
      printf("%d -> ", temp->data);
      temp = temp->next;
   }
   printf("NULL\n");
}
int main() {
   struct Node* header = createNode(-1); // Header node
```

```
    header->next = createNode(1);

    header->next->next = createNode(2);

    header->next->next->next = createNode(3);

    printList(header);

    return 0;

}
```

Output:

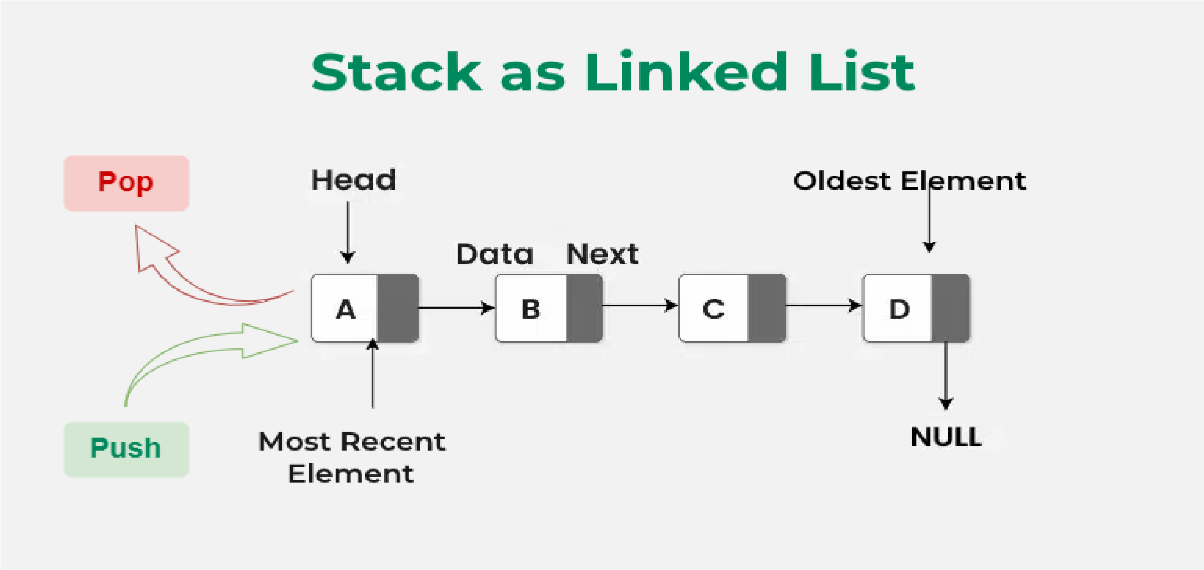
```
1 -> 2 -> 3 -> NULL

--------------------------------
Process exited after 0.004457 seconds with return value 0
Press any key to continue . . .
```

Linked Stacks and Queues

Stacks and queues can be efficiently implemented using linked lists.

Linked Stack

A stack is a data structure that follows the LIFO (Last In, First Out) principle.



Example:

```
#include <stdio.h>

#include <stdlib.h>

// Definition of the node

struct Node {

    int data;

    struct Node* next;

};

// Function to create a new node

struct Node* createNode(int data) {

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

    if (!newNode) {

        printf("Memory error\n");

        return NULL;

    }
```

```
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
// Push function
void push(struct Node** top, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *top;
    *top = newNode;
}
// Pop function
int pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack underflow\n");
        return -1;
    }
    struct Node* temp = *top;
    *top = (*top)->next;
    int popped = temp->data;
    free(temp);
    return popped;
}
// Function to print the stack
void printStack(struct Node* top) {
```

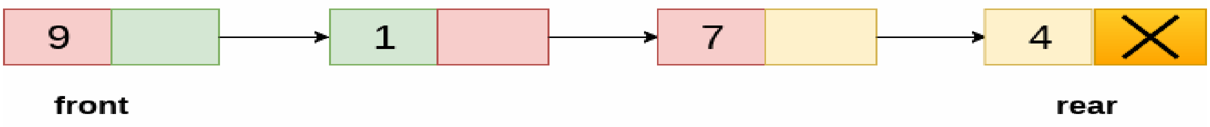
```
    while (top != NULL) {
        printf("%d -> ", top->data);
        top = top->next;
    }
    printf("NULL\n");
}
int main() {
    struct Node* stack = NULL;
    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);
    printStack(stack);
    printf("Popped: %d\n", pop(&stack));
    printStack(stack);
    return 0;
}
```

Output:

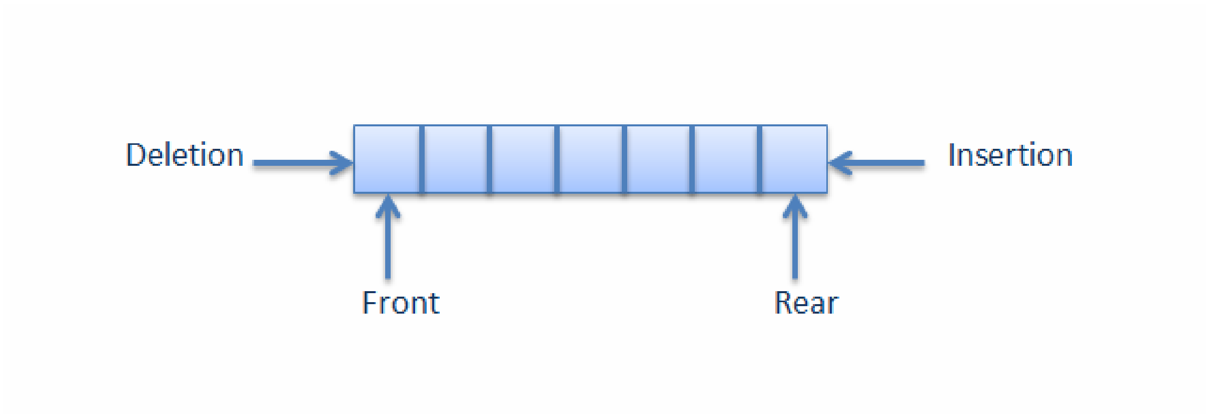
```
30 -> 20 -> 10 -> NULL
Popped: 30
20 -> 10 -> NULL

--------------------------------
Process exited after 0.005439 seconds with return value 0
Press any key to continue . . .
```

Linked Queue

A queue is a data structure that follows the FIFO (First In, First Out) principle.



Linked Queue



Example:

```
#include <stdio.h>

#include <stdlib.h>

// Definition of the node

struct Node {

    int data;

    struct Node* next;

};

// Function to create a new node

struct Node* createNode(int data) {

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

    if (!newNode) {

        printf("Memory error\n");

        return NULL;
```

```
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
// Enqueue function
void enqueue(struct Node** front, struct Node** rear, int data) {
    struct Node* newNode = createNode(data);
    if (*rear == NULL) {
        *front = *rear = newNode;
        return;
    }
    (*rear)->next = newNode;
    *rear = newNode;
}
// Dequeue function
int dequeue(struct Node** front, struct Node** rear) {
    if (*front == NULL) {
        printf("Queue underflow\n");
        return -1;
    }
    struct Node* temp = *front;
    *front = (*front)->next;
    if (*front == NULL) {
```

```
        *rear = NULL;
    }
    int dequeued = temp->data;
    free(temp);
    return dequeued;
}
// Function to print the queue
void printQueue(struct Node* front) {
    while (front != NULL) {
        printf("%d -> ", front->data);
        front = front->next;
    }
    printf("NULL\n");
}
int main() {
    struct Node* front = NULL;
    struct Node* rear = NULL;
    enqueue(&front, &rear, 10);
    enqueue(&front, &rear, 20);
    enqueue(&front, &rear, 30);
    printQueue(front);
    printf("Dequeued: %d\n", dequeue(&front, &rear));
    printQueue(front);
    return 0;
```

```
}
```

Output:

```
10 -> 20 -> 30 -> NULL
Dequeued: 10
20 -> 30 -> NULL

--------------------------------
Process exited after 0.005143 seconds with return value 0
Press any key to continue . . .
```

Applications of Linked Lists

Dynamic memory allocation: Linked lists are used to create flexible data structures that can grow and shrink dynamically.

Implementing stacks and queues: Linked lists provide an efficient way to implement these data structures.

Symbol tables in compilers: Linked lists are used to manage identifiers in symbol tables.

Adjacency lists in graphs: Linked lists are used to represent graph data structures.