**Batch: B-1 Roll No.:16010122104**

**Experiment No. 6**

**Grade: AA / AB / BB / BC / CC / CD /DD**

|  |
| --- |
| **Title: Implementation of various types of Linked List - doubly Linked List, circular Linked List , circular doubly Linked List etc** |

**Objective:** To understand the use of linked list as data structures for various application.

**Expected Outcome of Experiment:**

|  |  |
| --- | --- |
| **CO** | **Outcome** |
| **CO 2** | Apply linear and non-linear data structure in application development. |

**Books/ Journals/ Websites referred:**

**Introduction:**

A Linked List is a linear data structure which looks like a chain of nodes, where each node is a different element. Unlike Arrays, Linked List elements are not stored at a contiguous location. It is basically chains of nodes, each node contains information such as data and a pointer to the next node in the chain. In the linked list there is a head pointer, which points to the first element of the linked list, and if the list is empty then it simply points to null or nothing.

Why linked list data structure needed?

Dynamic Data structure: The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.

Ease of Insertion/Deletion: The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.

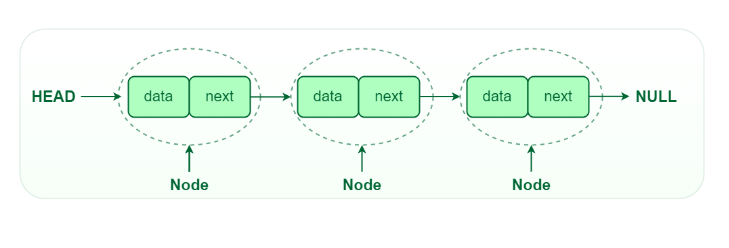
Efficient Memory Utilization: As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.

Implementation: Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

**Types of linked list:**

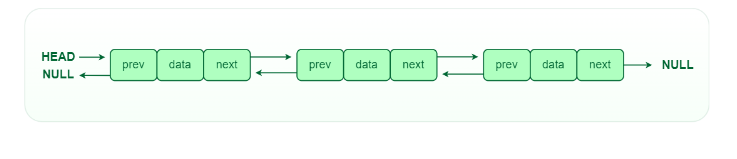
1. Singly-linked list

Traversal of items can be done in the forward direction only due to the linking of every node to its next node.

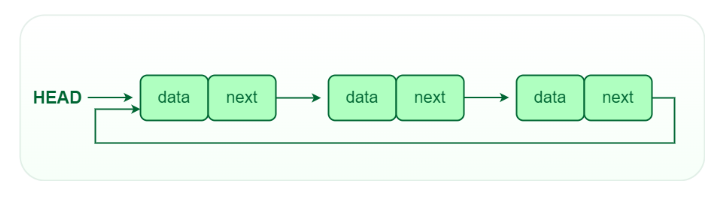


2. Doubly linked list

Traversal of items can be done in both forward and backward directions as every node contains an additional prev pointer that points to the previous node.



3. Circular linked lists

A circular linked list is a type of linked list in which the first and the last nodes are also connected to each other to form a circle, there is no NULL at the end.

**Program source code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

struct Node {

char title[100];

char genre[50];

int duration;

struct Node\* next;

struct Node\* prev;

};

struct Node\* initializeList() {

struct Node\* head = (struct Node\*)malloc(sizeof(struct Node));

strcpy(head->title, "Dummy");

head->next = head;

head->prev = head;

return head;

}

void addToList(struct Node\* head, char title[], char genre[], int duration) {

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

strcpy(newNode->title, title);

strcpy(newNode->genre, genre);

newNode->duration = duration;

newNode->next = head;

newNode->prev = head->prev;

head->prev->next = newNode;

head->prev = newNode;

}

void displayList(struct Node\* head) {

struct Node\* current = head->next;

if (current == head) {

printf("The list is empty.\n");

return;

}

printf("Continuous Watching List:\n");

while (current != head) {

printf("Title: %s\n", current->title);

printf("Genre: %s\n", current->genre);

printf("Duration: %d minutes\n", current->duration);

printf("-----------------------------\n");

current = current->next;

}

}

void removeFromList(struct Node\* head, char title[]) {

struct Node\* current = head->next;

while (current != head) {

if (strcmp(current->title, title) == 0) {

current->prev->next = current->next;

current->next->prev = current->prev;

free(current);

return;

}

current = current->next;

}

printf("'%s' not found in the list.\n", title);

}

int main() {

struct Node\* watchingList = initializeList();

int choice;

int numNodes;

char title[100];

char genre[50];

int duration;

while (1) {

printf("\nContinuous Watching List Menu:\n");

printf("1. Add show(s)/movie(s)\n");

printf("2. Remove a show/movie\n");

printf("3. Display the list\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the number of shows/movies to add: ");

scanf("%d", &numNodes);

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

printf("Enter title: ");

scanf(" %[^\n]s", title);

printf("Enter genre: ");

scanf(" %[^\n]s", genre);

printf("Enter duration (minutes): ");

scanf("%d", &duration);

addToList(watchingList, title, genre, duration);

}

break;

case 2:

printf("Enter the title of the show/movie to remove: ");

scanf(" %[^\n]s", title);

removeFromList(watchingList, title);

break;

case 3:

displayList(watchingList);

break;

case 4:

printf("Thank you for using the Continuous Watching List!\n");

return 0;

default:

printf("Invalid choice. Please select a valid option.\n");

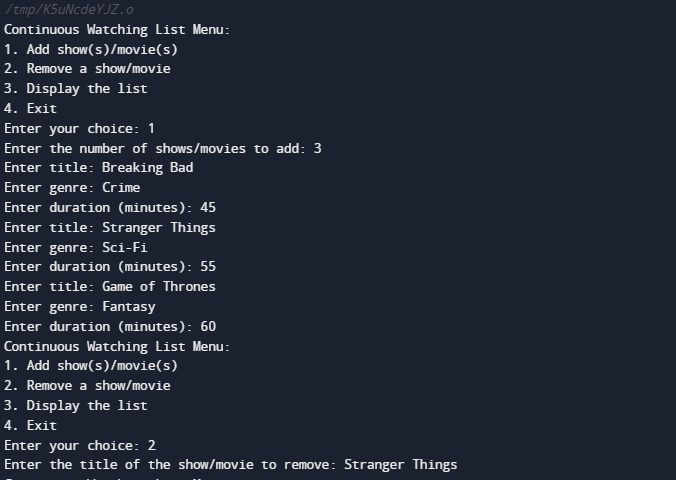
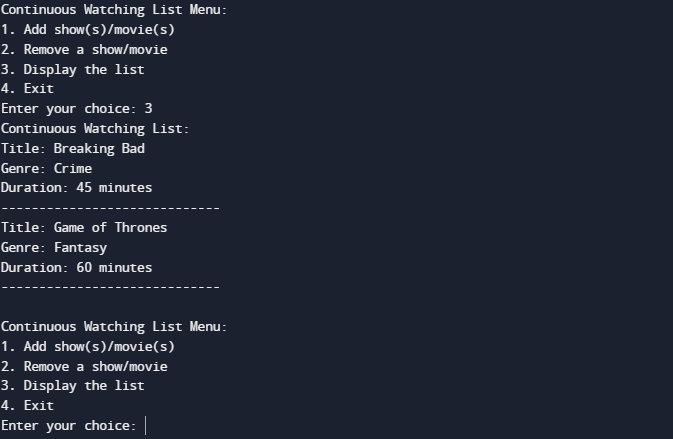
}

}

return 0;

}

**Output:**



**Conclusion:-**

We understood the use of linked list as data structures for various application.

**Post lab questions:**

1. Compare and contrast different types of linked list.
2. **Singly Linked List**:
   * **Structure**: Each node points to the next node in the list.
   * **Traversal**: Forward only.
   * **Memory Efficiency**: Lower memory overhead since it only stores one reference per node.
   * **Operations**: Insertion and deletion at the head are efficient (O(1)).
   * **Use Cases**: Suitable when you need simple, one-way traversal, and memory efficiency is a concern.
3. **Doubly Linked List**:
   * **Structure**: Each node points to both the next and the previous node in the list.
   * **Traversal**: Forward and backward.
   * **Memory Efficiency**: Higher memory overhead due to storing two references per node.
   * **Operations**: Insertion and deletion at both ends are efficient (O(1)).
   * **Use Cases**: Useful when you need bidirectional traversal and faster insertions and deletions at both ends.
4. **Circular Linked List**:
   * **Structure**: Similar to singly or doubly linked lists, but the last node points back to the first node.
   * **Traversal**: Continuous loop, can start from any node.
   * **Memory Efficiency**: Similar to singly or doubly linked lists.
   * **Operations**: Efficient for operations that involve looping through the list.
5. **Singly vs. Doubly Linked List**:
   * Singly linked lists are more memory-efficient.
   * Doubly linked lists allow bidirectional traversal.
   * Doubly linked lists are more versatile but use more memory.
6. **Singly vs. Circular Linked List**:
   * Singly linked lists have a clear end.
   * Circular linked lists are continuous and can be advantageous in specific scenarios like implementing circular buffers.
7. **Doubly vs. Circular Linked List**:
   * Both support bidirectional traversal.
   * Circular linked lists don't have a true end, which can be an advantage or disadvantage depending on the use case.
8. **Skip List**:
   * **Structure**: Consists of multiple linked lists where each level skips over a fixed number of elements.
   * **Traversal**: Efficient searching using multiple levels.
   * **Memory Efficiency**: Moderate memory overhead.
   * **Operations**: Efficient for searching (logarithmic time complexity).
   * **Use Cases**: Useful when you need fast searching, similar to balanced trees but with simpler implementation.
9. **Self-adjusting Lists** (e.g., Move-To-Front Lists):
   * **Structure**: Rearranges elements to improve access times.
   * **Traversal**: Depends on the adjustment strategy.
   * **Memory Efficiency**: Similar to regular linked lists.
   * **Operations**: Improves search times for frequently accessed elements.
   * **Use Cases**: Used in situations where frequently accessed items should be moved to the front to optimize access patterns.