

PROJECT: COMPUTER PROGRAMMING

SUBMITTED TO: DR. ABDUL QADAR KARA

BY: SAMI UR REHMAN , BISMA SHAIKH AND

EBAD ULLAH

ID: FA23-BSCS-0022 , FA23-BSCS-0023 AND

FA23-BSCS-0035

SEMESTER NO. 01

***Linked List Project Documentation***

**1. Introduction**

The Linked List project is a C++ implementation of a fundamental data structure - the Linked List. The primary objective is to provide hands-on experience to students in understanding and implementing basic operations associated with linked lists, including insertion, deletion, traversal, and memory management.

**2. Project Structure**

**2.1 Node Structure**

The project defines a Node struct with two members:

**Data**: to store the value of the node (integer type).

**Link**: a pointer to the next node.

**2.2 Functions**

The project includes the following functions:

**2.2.1 Basic Operations**

**insertAtHead:** Inserts a node with the given data at the beginning of the linked list.

**Explanation :** This function is responsible for inserting a new node with the provided data at the beginning of the linked list. It starts by creating a new node using the createNode function. Then, it ensures that the new node's link points to the current head of the list, effectively linking the new node to the existing list. Finally, the head is updated to point to the newly inserted node, making it the new starting point of the linked list.

**Algorithm insertAtHead(head, value):**

1. Create a new node (newNode) with the given value.

2. Set newNode's link to the current head.

3. Update head to newNode.

**insertAtTail:** Inserts a node with the given data at the end of the linked list.

**Explanation :** This function handles the insertion of a new node with the given data at the end of the linked list. It begins by creating a new node. If the list is empty, the new node becomes the head. If not, it traverses the list until it reaches the last node and appends the new node to it, ensuring that the linkage is correctly maintained.

**Algorithm insertAtTail(head, value):**

1. Create a new node (newNode) with the given value.

2. If head is NULL:

a. Set head to newNode.

3. Else:

a. Initialize temp to head.

b. While temp's link is not NULL:

i. Move temp to temp's link.

c. Set temp's link to newNode.

**insertAtPosition**: Inserts a node with the given data at a specific position in the linked list.

**Explanation :** This function is designed to insert a new node with the provided data at a specific position within the linked list. It first checks if the specified position is either at the beginning or if the list is empty. If true, it delegates the task to the insertAtHead function. If the position is beyond the length of the list, it defaults to inserting the new node at the tail. Otherwise, it traverses to the position before the specified one, and the new node is appropriately inserted in the sequence.

**Algorithm insertAtPosition(head, value, pos):**

1. If pos is 0 or head is NULL:

a. Call insertAtHead(head, value).

2. Else:

a. Create a new node (newNode) with the given value.

b. Initialize temp to head and currentPos to 0.

c. While currentPos is less than pos - 1 and temp is not NULL:

i. Move temp to temp's link.

ii. Increment currentPos by 1.

d. If temp is not NULL:

i. Set newNode's link to temp's link.

ii. Set temp's link to newNode.

e. Else:

i. Call insertAtTail(head, value).

**deleteAtHead**: Deletes the node at the beginning of the linked list.

**Explanation :** The purpose of this function is to delete the node at the beginning of the linked list. It checks if the list is not empty, then stores the current head in a temporary variable, updates the head to the next node in the list, and finally deallocates the memory occupied by the previous head.

**Algorithm deleteAtHead(head):**

1. If head is not NULL:

a. Store head in temp.

b. Update head to temp's link.

c. Delete temp.

**deleteAtEnd**: Deletes the node at the end of the linked list.

**Explanation :** This function facilitates the removal of the last node from the linked list. It verifies if the list is not empty. If there is only one node, it deletes the head. Otherwise, it traverses the list to find the second-last node, adjusts its link to NULL, and then deletes the last node.

**Algorithm deleteAtEnd(head):**

1. If head is not NULL:

a. If head's link is NULL:

i. Delete head.

ii. Set head to NULL.

b. Else:

i. Initialize temp to head.

ii. While temp's link's link is not NULL:

- Move temp to temp's link.

iii. Delete temp's link.

iv. Set temp's link to NULL.

**deleteAtPosition**: Deletes the node at a specific position in the linked list.

**Explanation :** The purpose of this function is to delete the node at a specific position within the linked list. If the position is at the beginning or the list is empty, it utilizes the deleteAtHead function. Otherwise, it traverses to the node before the specified position, updates its link to bypass the node to be deleted, and then deallocates the memory occupied by that node.

**Algorithm deleteAtPosition(head, pos):**

1. If pos is 0 or head is NULL:

a. Call deleteAtHead(head).

2. Else:

a. Initialize temp to head and currentPos to 0.

b. While currentPos is less than pos - 1 and temp's link is not NULL:

i. Move temp to temp's link.

ii. Increment currentPos by 1.

c. If temp is not NULL and temp's link is not NULL:

i. Store temp's link in toDelete.

ii. Set temp's link to toDelete's link.

iii. Delete toDelete.

**deleteByData**: Deletes the node with the given data from the linked list.

**Explanation :** This function aims to delete the first occurrence of a node with a given data value from the linked list. It traverses the list to find the node with the specified data. Once found, it adjusts the pointers to skip the node and deallocates its memory. Special consideration is given if the node to be deleted is the head of the list.

**Algorithm deleteByData(head, data):**

1. If head is not NULL:

a. Initialize current to head and previous to NULL.

b. While current is not NULL and current's data is not equal to data:

i. Set previous to current.

ii. Move current to current's link.

c. If current is not NULL:

i. If previous is not NULL:

- Set previous's link to current's link.

ii. Else:

- Set head to current's link.

iii. Delete current.

**print**: Traverses and prints the elements of the linked list.

**Explanation :** This function is responsible for traversing the linked list and printing the data values of each node. It begins at the head and iterates through each node, printing its data value, until reaching the end of the list.

**Algorithm print(head):**

1. Initialize temp to head.

2. While temp is not NULL:

a. Print temp's data followed by "->".

b. Move temp to temp's link.

3. Print "NULL".

**search**: Searches for a given data in the linked list.

**Explanation :** The search function is designed to locate the first occurrence of a node with a given data value within the linked list. It initializes a pointer to traverse the list, searching for the specified data. If the data is found, the function returns the pointer to that node; otherwise, it returns NULL.

**Algorithm search(head, data):**

1. Initialize temp to head.

2. While temp is not NULL and temp's data is not equal to data:

a. Move temp to temp's link.

3. Return temp.

**deleteList**: Deletes the entire linked list and frees memory.

**Explanation :** This function is responsible for deleting the entire linked list and freeing the associated memory. It iteratively traverses the list, deleting each node and updating the head pointer to NULL upon completion.

**Algorithm deleteList(head):**

1. Initialize current to head.

2. While current is not NULL:

a. Store current in temp.

b. Move current to current's link.

c. Delete temp.

3. Set head to NULL.

**Mainfunction :** All other function will be called from main.

**Explanation :** The main function serves as the entry point of the program. It initializes an empty linked list and presents a menu-driven interface for users to perform various linked list operations. The program continues to execute until the user chooses to exit, ensuring proper memory cleanup using the deleteList function before concluding.

**Algorithm main():**

1. Initialize head to NULL.

2. Initialize choice, data, position variables.

3. Do the following until choice is 0:

a. Display the menu for linked list operations.

b. Accept the user's choice.

c. Switch on the user's choice:

i. Case 1:

- Prompt user for data.

- Call insertAtHead(head, data).

ii. Case 2:

- Prompt user for data.

- Call insertAtTail(head, data).

iii. Case 3:

- Prompt user for position and data.

- Call insertAtPosition(head, data, position).

iv. Case 4:

- Call deleteAtHead(head).

v. Case 5:

- Call deleteAtEnd(head).

vi. Case 6:

- Prompt user for position.

- Call deleteAtPosition(head, position).

vii. Case 7:

- Prompt user for data.

- Call deleteByData(head, data).

viii. Case 8:

- Call print(head).

ix. Case 9:

- Prompt user for data.

- Call search(head, data).

x. Case 0:

- Print exit message.

xi. Default:

- Print invalid choice message.

d. While choice is not 0, repeat the loop.

4. Call deleteList(head) to free memory.

5. Return 0.

**2.3 Memory Management**

Memory management is a critical aspect of the Linked List project, contributing to a deeper understanding of dynamic data structures and resource utilization in C++. The project employs dynamic memory allocation and deallocation using new and delete to create and manage nodes within the linked list.

**Dynamic Node Creation**: In functions like insertAtHead, insertAtTail, and insertAtPosition, the new keyword is utilized to dynamically allocate memory for a new node. This dynamic creation allows the linked list to adapt to changing data sizes, ensuring efficient utilization of memory resources.

**Node Deletion:** When nodes are deleted, the delete keyword is employed to release the memory occupied by the node. This ensures that memory is freed up and available for reuse.

**List Deletion:** The deleteList function iterates through the entire linked list, deleting each node and freeing up the associated memory. This careful memory management prevents memory leaks and ensures optimal use of system resources.

**3. User Interface**

The user interface of the Linked List project is designed with a menu-driven approach, allowing users to interact seamlessly with the linked list functionalities. The menu structure provides a clear and intuitive way for users to choose and execute various operations on the linked list.

**Menu-Driven Interaction:** The main function presents a menu with numbered options, each corresponding to a specific linked list operation. Users can easily select an option by entering the corresponding number, making the interface user-friendly and accessible.

**User Input Handling**: User input is obtained using cin, ensuring that the program can respond to user choices and execute the appropriate linked list operation based on the input.

**Error Handling:** The program incorporates error handling to validate user input, ensuring that invalid choices or data are appropriately handled, providing a robust and user-friendly experience.

**Informative Messages:** The program includes informative messages to guide the user, providing feedback on the execution of operations and the current state of the linked list.

The combination of a well-structured menu, user input handling, error validation, and informative messages collectively creates an interface that is both accessible to users and conducive to a positive learning experience..

**4. Evaluation Criteria**

**4.1 Functionality (60%)**

The program correctly implements all required linked list operations without logical errors.

**4.2 Code Quality (25%)**

The code is well-structured, readable, and well-commented. It uses meaningful variable names and adheres to coding standards.

**4.3 User Interface (10%)**

The user interface is intuitive, allowing users to easily interact with linked list functionalities.

**4.4 Documentation and Submission (5%)**

A brief report or documentation is provided explaining the approach, challenges faced, and any additional features implemented. The code is submitted following the provided guidelines.

**5. Conclusion**

The Linked List project serves as an invaluable educational experience, offering students a deep dive into the intricacies of working with pointers, structs, memory allocation, and fundamental linked list operations within the C++ programming language. Through the implementation of essential functionalities such as insertion, deletion, traversal, and search, students gain practical insights into the dynamic and flexible nature of linked lists.

This project not only provides hands-on experience but also lays a solid foundation for understanding key concepts in data structures and programming. The creation and manipulation of nodes, coupled with dynamic memory management, underscore the importance of precision and clarity in code implementation. Students are exposed to the nuances of algorithmic thinking, fostering a structured approach to problem-solving.

Moreover, the project's user interface, designed in a menu-driven format, enhances the learning experience by allowing users to interact with the linked list functionalities seamlessly. This approach ensures that students not only comprehend the underlying concepts but also develop practical skills in creating user-friendly programs.

As a culmination of these efforts, the Linked List project equips students with a toolkit of skills that extend beyond the specific scope of linked lists. The project lays the groundwork for more complex data structures and algorithms, setting the stage for future explorations in the vast landscape of computer science.

In conclusion, the Linked List project stands as a testament to the integration of theoretical knowledge with practical implementation, providing students with a solid foundation and sparking their curiosity for more advanced topics in the realm of programming and data structures.

THE END