|  |  |
| --- | --- |
| **Assignment No:3** | |
| **Aim:** | Write a program in C++ to implement and manipulate a singly linked list using separate functions for creating nodes, displaying the list, inserting at the front, inserting at the rear, inserting after a specific location, deleting the first node, deleting the last node, and deleting after a specific location. |
| **Objective:** | Comprehend the structure and functionality of linked lists by explaining the purpose and operation of each function used to create, insert, display, and delete nodes within a linked list. Demonstrate an understanding of how linked lists differ from arrays in terms of memory management and flexibility. |
| **Theory:** | A linked list is a fundamental data structure used in computer science for storing a sequence of elements, where each element points to the next. This structure allows for efficient insertion and deletion of elements.  **Advantages of Linked Lists Over Arrays:**  **Dynamic Size:**  Linked lists can grow or shrink in size dynamically, making them more flexible than arrays, which require a predefined size. This allows efficient memory utilization, especially when the number of elements is unpredictable.  **Efficient Insertions and Deletions:**  In linked lists, elements can be easily inserted or deleted without reorganizing the entire data structure. In contrast, inserting or deleting an element in an array often requires shifting multiple elements, which can be time-consuming.  **No Wasted Space:**  Since linked lists allocate memory as needed, there’s no wasted space. Arrays, on the other hand, require allocation of a fixed amount of memory, which can lead to wasted space if the array is not fully utilized or insufficient space if the array is full.  **Ease of Implementing Data Structures:**  Linked lists are well-suited for implementing other dynamic data structures such as stacks, queues, and graphs. Their structure allows for simpler and more efficient implementations of these data structures compared to arrays.  **Memory Management:**  Linked lists do not require contiguous memory allocation, making them a better choice in situations where memory is fragmented. Arrays require a contiguous block of memory, which may not always be available.  Flexibility in Data Storage:  Elements in a linked list can be easily re-arranged or manipulated without needing to move other elements. This flexibility is particularly useful in applications where elements need to be frequently inserted, deleted, or rearranged.  The program will include the following operations:  createnode(int n): This function creates a new node with the given data and adds it to the end of the list.  display(): This function prints the elements of the linked list.  front(int n): This function inserts a new node with the given data at the front of the list.  ins\_rear(int n): This function inserts a new node with the given data at the end of the list.  ins\_after(int n, int loc): This function inserts a new node with the given data after a specified location in the list.  del\_first(): This function deletes the first node of the list.  del\_rear(): This function deletes the last node of the list.  del\_after(int loc): This function deletes the node after a specified location in the list.  Each of these operations is critical for understanding how linked lists work and how data structures can be manipulated efficiently.  **Applications of linked lists:**   * Operating system process scheduling * Memory management systems * Real-time task management * Undo/Redo functionality in software applications * Real-time event handling in interactive systems * Dynamic file systems * Managing network packet buffers * Real-time data streams in multimedia applications * Real-time sensor data processing in embedded systems * Browser history management |
| **Algorithm:** | **1. Algorithm for Creating a Node:**   * **Start** by creating a new element to store the given value. * Set the new element’s value to the provided input. * Set the pointer of the new element to indicate it is the last in the sequence. * **If** the sequence is empty:   + Make this new element the first and last element in the sequence. * **Else**:   + Link the last element in the sequence to the new element.   + Update the reference to the last element to this new element. * **End** the process.   **2. Algorithm for Displaying the Sequence:**   * **Start** by setting a reference to the first element in the sequence. * **If** the sequence is empty:   + Display a message indicating the sequence is empty. * **Else**:   + **While** the end of the sequence has not been reached:     - Display the value of the current element.     - Move to the next element in the sequence. * **End** the process.   **3. Algorithm for Inserting at the Front:**   * **Start** by creating a new element to store the given value. * Set the new element’s value to the provided input. * Link the new element to the current first element in the sequence. * Update the reference to the first element to this new element. * **End** the process.   linkedlist_insert_at_start  **4. Algorithm for Inserting at the Rear:**   * **Start** by creating a new element to store the given value. * Set the new element’s value to the provided input. * Set the pointer of the new element to indicate it is the last in the sequence. * **If** the sequence is empty:   + Make this new element the first element in the sequence. * **Else**:   + Traverse the sequence to find the last element.   + Link the last element to the new element. * **End** the process.   linkedlist_insert_last  **5. Algorithm for Inserting After a Specified Position:**   * **Start** by creating a new element to store the given value. * Set the new element’s value to the provided input. * Traverse the sequence to reach the specified position. * Link the new element to the element following the specified position. * Link the specified position element to the new element. * **End** the process.   linkedlist_insert_middle  **6. Algorithm for Deleting the First Element:**   * **Start** by referencing the first element in the sequence. * **If** the sequence is empty:   + There is nothing to delete. * **Else**:   + Update the reference to the first element to the next element in the sequence.   + Remove the original first element.   + Display a message indicating the deletion was successful. * **End** the process.     **7. Algorithm for Deleting the Last Element:**   * **Start** by referencing the first element and initializing a second reference. * **If** the sequence is empty:   + There is nothing to delete. * **Else**:   + Traverse the sequence to reach the last element.   + Update the pointer of the second-to-last element to indicate it is now the last.   + Display the value of the deleted element.   + Remove the last element. * **End** the process.     **8. Algorithm for Deleting After a Specified Position:**   * **Start** by referencing the first element and initializing a second reference. * Traverse the sequence to reach the specified position. * **If** there is no element following the specified position:   + There is nothing to delete. * **Else**:   + Link the specified position element to the element after the one to be deleted.   + Remove the element after the specified position. * **End** the process.   linkedlist_deletion |
| **Program:** | #include <iostream>  using namespace std;  struct Node {  int data;  Node\* next;  };  class SinglyLinkedList {  private:  Node\* head;  public:  SinglyLinkedList() {  head = nullptr;  }  Node\* createNode(int value) {  Node\* newNode = new Node();  newNode->data = value;  newNode->next = nullptr;  return newNode;  }  void displayList() {  if (head == nullptr) {  cout << "The list is empty." << endl;  return;  }  Node\* current = head;  while (current != nullptr) {  cout << current->data << " -> ";  current = current->next;  }  cout << "nullptr" << endl;  }  void insertAtFront(int value) {  Node\* newNode = createNode(value);  newNode->next = head;  head = newNode;  }    void insertAtRear(int value) {  Node\* newNode = createNode(value);  if (head == nullptr) {  head = newNode;  return;  }  Node\* current = head;  while (current->next != nullptr) {  current = current->next;  }  current->next = newNode;  }  void insertAfterPosition(int value, int position) {  Node\* newNode = createNode(value);  Node\* current = head;  for (int i = 0; i < position && current != nullptr; i++) {  current = current->next;  }  if (current == nullptr) {  cout << "Position does not exist." << endl;  delete newNode;  return;  }  newNode->next = current->next;  current->next = newNode;  }  void deleteFirstNode() {  if (head == nullptr) {  cout << "The list is empty. Nothing to delete." << endl;  return;  }  Node\* temp = head;  head = head->next;  delete temp;  cout << "Deleted the first node." << endl;  }  void deleteLastNode() {  if (head == nullptr) {  cout << "The list is empty. Nothing to delete." << endl;  return;  }  if (head->next == nullptr) {  delete head;  head = nullptr;  cout << "Deleted the last node." << endl;  return;  }  Node\* current = head;  while (current->next->next != nullptr) {  current = current->next;  }  delete current->next;  current->next = nullptr;  cout << "Deleted the last node." << endl;  }  void deleteAfterPosition(int position) {  Node\* current = head;  for (int i = 0; i < position && current != nullptr; i++) {  current = current->next;  }  if (current == nullptr || current->next == nullptr) {  cout << "No node exists after position " << position << "." << endl;  return;  }  Node\* temp = current->next;  current->next = temp->next;  delete temp;  cout << "Deleted node after position " << position << "." << endl;  }  ~SinglyLinkedList() {  while (head != nullptr) {  deleteFirstNode();  }  }  };  int main() {  SinglyLinkedList list;  list.insertAtRear(10);  list.insertAtRear(20);  list.insertAtRear(30);  cout << "Initial list: ";  list.displayList();  list.insertAtFront(5);  cout << "After inserting 5 at front: ";  list.displayList();  list.insertAfterPosition(25, 1);  cout << "After inserting 25 after position 1: ";  list.displayList();  list.deleteFirstNode();  cout << "After deleting the first node: ";  list.displayList();  list.deleteLastNode();  cout << "After deleting the last node: ";  list.displayList();  list.deleteAfterPosition(1);  cout << "After deleting node after position 1: ";  list.displayList();  return 0;  } |
| **Output:** | Initial list: 10 -> 20 -> 30 -> nullptr  After inserting 5 at front: 5 -> 10 -> 20 -> 30 -> nullptr  After inserting 25 after position 1: 5 -> 10 -> 25 -> 20 -> 30 -> nullptr  After deleting the first node: 10 -> 25 -> 20 -> 30 -> nullptr  After deleting the last node: 10 -> 25 -> 20 -> nullptr  After deleting node after position 1: 10 -> 25 -> nullptr |
| **Conclusion:** | |
| **Date:** | |
| **Staff Sign:** | |
| \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***END**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* | |