Data Structures and Algorithms (Spring2024)

Week06 (20/21/21-Mar-2024)

M Ateeq,
Department of Data Science, The Islamia University of Bahawalpur.

Linked List

Deletion

We have already covered representation, traversal, and insertion of elements in a linked list.To expand on your current implementation, let's discuss and implement deletion in a singly linked list. Deletion in a linked list involves removing a node from the list, which can occur in several contexts:

1. **Deleting the first node**: Involves changing the head of the list to the second node.
2. **Deleting a node other than the first**: Requires traversing the list to find the node before the one to be deleted, then adjusting pointers to bypass the deleted node.
3. **Deleting the last node**: Similar to deleting a non-first node, but you stop at the second-to-last node and set its next pointer to `nullptr`.

When deleting a node, it's crucial to ensure the memory occupied by it is properly deallocated to avoid memory leaks.

Here's how we can implement deletion functionality in your linked list:

- **Deleting at the beginning**: Remove the first node and update the head pointer.
- **Deleting at the end**: Traverse to the second-to-last node and update its next pointer to `nullptr`.
- **Deleting at a specific position**: Similar to insertion at a specific position, but instead of inserting a new node, you'll remove the node at the given position.

Let's add these functionalities to our code:

```
#include <iostream>

struct Node {
    int data;
    Node* next;
};

// Previous traverse function...
// Previous insert functions...
```

Delete At Beginning

```
// Function to delete the first node of the list
void deleteAtBeginning(Node*& head) {
    if (head == nullptr) return; // Empty list, nothing to delete

    Node* temp = head; // Store the current head node in a temporary variable
    head = head->next; // Update the head to the next node
    delete temp; // Deallocate the former head node
}
```

Delete At End

```
// Function to delete the last node of the list
void deleteAtEnd(Node*& head) {
    if (head == nullptr) return; // Empty list, nothing to delete

    // If there is only one node, delete it and update the head to nullptr
    if (head->next == nullptr) {
        delete head;
        head = nullptr;
        return;
    }
```

```
    // Traverse to the second-to-last node
        Node* current = head;
        while (current->next->next != nullptr) {
            current = current->next;
        }

        // Delete the last node and update the second-to-last node's next to nullptr
        delete current->next;
        current->next = nullptr;
    }
```

Delete At Position

```
    // Function to delete a node at a specific position
    void deleteAtPosition(Node*& head, int position) {
        if (head == nullptr || position < 0) return; // Invalid position or empty list

        Node* current = head;
        if (position == 0) { // Delete the first node
            head = head->next;
            delete current;
            return;
        }

        // Traverse to the node before the one to delete
        for (int i = 0; i < position - 1 && current->next != nullptr; ++i) {
            current = current->next;
        }




    // If the position is beyond the end of the list
        if (current == nullptr || current->next == nullptr) {
            std::cout << "Invalid position\n";
            return;
        }

        // Perform the deletion
        Node* temp = current->next; // Node to delete
        current->next = current->next->next; // Bypass the deleted node
        delete temp; // Deallocate the node
    }
```

```
int main() {
    // Initialize an empty linked list
    Node* head = nullptr;

    // Insert nodes into the linked list to prepare for demonstration
    insertAtBeginning(head, 10);  // List: 10
    insertAtEnd(head, 20);        // List: 10 -> 20
    insertAtEnd(head, 30);        // List: 10 -> 20 -> 30
    insertAtPosition(head, 2, 25);// List: 10 -> 20 -> 25 -> 30
    insertAtEnd(head, 40);        // List: 10 -> 20 -> 25 -> 30 -> 40



std::cout << "Initial list: ";
    traverseLinkedList(head);     // Expected: 10 -> 20 -> 25 -> 30 -> 40

    // Demonstrate deletion at the beginning
    deleteAtBeginning(head);
    std::cout << "After deleting at the beginning: ";
    traverseLinkedList(head);     // Expected: 20 -> 25 -> 30 -> 40

    // Demonstrate deletion at the end
    deleteAtEnd(head);
    std::cout << "After deleting at the end: ";
    traverseLinkedList(head);     // Expected: 20 -> 25 -> 30



// Demonstrate deletion at a specific position (e.g., position 1)
    deleteAtPosition(head, 1);
    std::cout << "After deleting at position 1: ";
    traverseLinkedList(head);     // Expected: 20 -> 30

    // Clean up memory by deallocating remaining nodes
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }

    return 0;
}
```

In this extended `main` function, we first insert five nodes into the list to create a sequence of elements for clear demonstration. We then sequentially perform and display the results of deletion operations:

1. **Delete at the beginning**: Removes the first node (10), showing the list starting from 20.
2. **Delete at the end**: Removes the last node (40), showcasing the trimmed list ending with 30.
3. **Delete at a specific position**: We choose position 1 (zero-indexed) for demonstration, removing 25 and showing the list as 20 -> 30.

After each deletion, we traverse and print the list to visualize the changes. Finally, we ensure to deallocate the remaining nodes to clean up allocated memory, preventing memory leaks.

Doubly Linked Lists

A Doubly Linked List is an advanced and versatile data structure that consists of nodes linked together in a sequence, where each node contains three components: the data, a pointer to the next node (often called `next`), and a pointer to the previous node (`prev`). This bidirectional linking offers significant advantages over its simpler counterpart, the singly linked list, by allowing traversal in both directions—forward and backward. The inclusion of a previous pointer in each node facilitates more efficient insertions and deletions, particularly in the middle of the list, by providing direct access to the preceding elements.

Characteristics of Doubly Linked Lists

- **Bidirectional Traversal**: Unlike singly linked lists, doubly linked lists can be traversed in both directions, which simplifies operations that require looking at or modifying previous nodes.
- **Dynamic Data Structure**: Doubly linked lists are dynamic, meaning they can grow and shrink at runtime by allocating and deallocating memory for nodes. This property makes them suitable for applications where the size of the data structure changes frequently.
- **Efficient Insertions and Deletions**: Nodes can be added or removed from both the beginning and the end of the list, as well as directly before or after a given node, with relative ease and without the need to traverse the entire list.

Disadvantages

- **Increased Memory Overhead**: Each node requires extra memory for the additional pointer to the previous node, which can be a drawback in memory-constrained environments.
- **Complexity**: The implementation of doubly linked lists is more complex than that of singly linked lists due to the additional pointer. Care must be taken to correctly update both `next` and `prev` pointers during insertions and deletions to maintain the integrity of the list.

Applications

Doubly linked lists are used in various applications where their unique features provide significant advantages, such as:

- Implementing complex data structures like dequeues, which require efficient operations at both ends.
- Browser history, where users can navigate forward and backward through visited pages.
- Undo functionality in software applications, where operations are stored in a list that can be traversed in reverse to undo recent changes.

Creation of a Doubly Linked List

Creating a doubly linked list involves initializing the list and defining a method for adding nodes to it. Initially, the list is empty, signified by a `nullptr` value for the head (the first node in the list). Nodes are then added to the list using various insertion methods, such as adding at the beginning, at the end, or after a specific node.

```
struct Node {
    int data;       // Data carried by this node.
    Node* next;     // Pointer to the next node in the list.
    Node* prev;     // Pointer to the previous node in the list.

    // Constructor to simplify node creation.
    Node(int data, Node* next = nullptr, Node* prev = nullptr)
        : data(data), next(next), prev(prev) {
        data = data;
            next = nullptr;
            prev = nullptr;
        }
};
```

Traversal of a Doubly Linked List

Traversal in a doubly linked list can be performed in two directions: forward (from the head to the end) and backward (from the end to the head). This flexibility is due to the presence of both next and prev pointers in each node.

Forward Traversal

Forward traversal starts from the head of the list and moves through each node by following the next pointers until reaching a node that points to `nullptr`, indicating the end of the list.

```
void forwardTraversal(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }
    std::cout << std::endl;
}
```

Backward Traversal

Backward traversal begins from the last node, which requires first traversing the list to find the end. Once the end is reached, traversal moves backward through the list by following the prev pointers until reaching the first node, which has a prev pointer of `nullptr`.

```
void backwardTraversal(Node* head) {
    Node* current = head;
    // First, find the last node
    while (current != nullptr && current->next != nullptr) {
        current = current->next;
    }
    // Now, traverse backward from the last node
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->prev;
    }
    std::cout << std::endl;
}
```

Insertion in a Doubly Linked List

Insertion in a doubly linked list is a fundamental operation that involves adding a new node to the list. This operation can be performed in several ways, depending on where the new node is to be placed: at the beginning of the list, at the end, after a specific node, or before a specific node. Each of these insertion methods enhances the flexibility and utility of doubly linked lists in various applications.

Insertion at the Beginning

To insert a new node at the beginning of the list:

1. **Create a new node** with the given data.
2. **Set the new node's next pointer** to the current head of the list.
3. **Update the current head node's prev pointer** to point to the new node, if the list is not empty.
4. **Update the head of the list** to be the new node.
5. **Set the new node's prev pointer** to `nullptr`.

This operation effectively places the new node as the first node of the list, with the entire operation being O(1), or constant time complexity.

```
// Function to insert a new node at the beginning of the list
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node(newData);
    newNode->next = head; // Make next of new node as head
    if (head != nullptr) {
        head->prev = newNode; // Move the prev of head node to new node
    }
    head = newNode; // Move the head to point to the new node
}
```

Insertion at the End

To add a new node at the end of the list:

1. **Create a new node** with the given data.
2. If the list is empty, **set the head to the new node**.
3. If the list is not empty, **traverse to the last node** of the list.
4. **Set the last node's next pointer** to the new node.
5. **Set the new node's prev pointer** to the last node.
6. **Set the new node's next pointer** to `nullptr`.

This method appends the new node to the list, requiring traversal from the head to the end of the list, making it an O(n) operation, where n is the number of elements in the list.

```
// Function to insert a new node at the end of the list
void insertAtEnd(Node*& head, int newData) {
    Node* newNode = new Node(newData);
    if (head == nullptr) {
        head = newNode; // If the list is empty, make the new node as head
        return;
    }
    Node* lastNode = head;
    while (lastNode->next != nullptr) {
        lastNode = lastNode->next; // Traverse till the last node
    }
    lastNode->next = newNode; // Change the next of last node
    newNode->prev = lastNode; // Make last node as previous of new node
}
```

Insertion After a Specific Node

Inserting a new node after a specific node involves:

1. **Locate the specific node** after which the new node will be inserted.
2. **Create a new node** with the given data.
3. **Set the new node's next pointer** to the specific node's next node.
4. **Update the next node's prev pointer** (if it is not `nullptr`) to point to the new node.
5. **Set the specific node's next pointer** to the new node.
6. **Set the new node's prev pointer** to the specific node.

This operation allows insertion at any point within the list and is particularly efficient because it does not require traversal of the entire list, assuming the specific node is already known.

```
// Function to insert a node after a given node
void insertAfter(Node* prevNode, int newData) {
    if (prevNode == nullptr) {
        cout << "The given previous node cannot be nullptr" << endl;
        return;
    }
    Node* newNode = new Node(newData);
    newNode->next = prevNode->next; // Make next of new node as next of prevNode
    prevNode->next = newNode; // Make the next of prevNode as newNode
    newNode->prev = prevNode; // Make prevNode as previous of newNode

    if (newNode->next != nullptr) {
        newNode->next->prev = newNode; // Change previous of newNode's next node
    }
}
```

Insertion Before a Specific Node

Inserting a new node before a specific node can be slightly more complex and involves:

1. **Locate the specific node** before which the new node will be inserted. If the specific node is the head, follow the insertion at the beginning steps.
2. **Create a new node** with the given data.
3. **Set the new node's next pointer** to the specific node.
4. **Set the new node's prev pointer** to the specific node's prev node.
5. **Update the specific node's prev node's next pointer** (if it is not the head) to point to the new node.
6. **Update the specific node's prev pointer** to point to the new node.

This method allows for insertion directly before a known node, effectively inserting the new node into the desired position in the list without traversing it entirely if the specific node is known.

```
// Function to insert a new node before a specific node
void insertBefore(Node*& head, Node* nextNode, int newData) {
    if (nextNode == nullptr) {
        cout << "The given next node cannot be nullptr" << endl;
        return;
    }
    Node* newNode = new Node(newData);
    newNode->prev = nextNode->prev;
    newNode->next = nextNode;
    nextNode->prev = newNode;
    if (newNode->prev != nullptr) {
        newNode->prev->next = newNode;
    } else {
        head = newNode; // If the nextNode is head, then update the head to newNode
    }
}
```

Reflection MCQs

1. **What is a linked list?**
   - A) A data structure where elements are stored in contiguous memory locations.
   - B) A collection of elements stored in non-contiguous memory locations, linked using pointers.
   - C) A fixed-size collection of elements.
   - D) A dynamic array with automatic resizing.
   - **Correct Option:** B
2. **How does a singly linked list differ from a doubly linked list?**
   - A) It uses more memory.
   - B) Each node points to the next and previous nodes.
   - C) It only allows forward traversal.
   - D) It cannot store data.
   - **Correct Option:** C

1. **What is the time complexity of accessing an element by index in a linked list?**

   - A) O(1)
   - B) O(log n)
   - C) O(n)
   - D) O(n^2)
   - **Correct Option:** C
2. **Which of the following operations is more efficient in a linked list compared to a dynamic array?**

   - A) Accessing an element by index.
   - B) Insertion at the end.
   - C) Insertion at the beginning.
   - D) Searching for an element.
   - **Correct Option:** C

1. **In a singly linked list, how is the end of the list indicated?**

   - A) By a special end node.
   - B) By a NULL pointer.
   - C) By the largest value.
   - D) By looping back to the first node.
   - **Correct Option:** B
2. **What is an Abstract Data Type (ADT)?**

   - A) A specific implementation of a data structure in a programming language.
   - B) A low-level data type provided by a programming language.
   - C) A type of data structure that allows dynamic resizing.
   - D) A model for data types where the data type is defined by its behavior.
   - **Correct Option:** D

1. **Can arrays be considered an ADT?**

   - A) Yes, because they abstract away memory management.
   - B) No, because they are concrete implementations with fixed operations.
   - C) Yes, because they can grow and shrink dynamically.
   - D) No, because they are a primitive data type.
   - **Correct Option:** B
2. **Which data structure would be most suitable for implementing a browser's back button functionality?**

   - A) Static Array
   - B) Circular Linked List
   - C) Singly Linked List
   - D) Doubly Linked List
   - **Correct Option:** D

1. **What is the time complexity of inserting a new node at the beginning of a singly linked list?**

   - A) O(1)
   - B) O(log n)
   - C) O(n)
   - D) O(n^2)
   - **Correct Option:** A
2. **Which of the following is NOT a correct way to represent a node in a singly linked list using `struct` in C++?**

   - A) `struct Node { int data; Node* next; };`
   - B) `struct Node { int data; Node next; };`
   - C) `struct Node { int data; Node* next = nullptr; };`
   - D) `struct Node { int data; Node* next; Node(int x) : data(x), next(nullptr) {} };`
   - **Correct Option:** B

1. **For traversing a singly linked list to its last node, which of the following is true?**

   - A) You need to know the size of the list beforehand.
   - B) You traverse until you find a node where `node->next` is NULL.
   - C) You can directly access it in O(1) time.
   - D) Traversal is not possible in singly linked lists.
   - **Correct Option:** B
2. **In a singly linked list, what does the `next` pointer of the last node point to?**

   - A) The first node of the list.
   - B) A random node in the list.
   - C) It does not have a `next` pointer.
   - D) NULL.
   - **Correct Option:** D

1. **How do dynamic arrays handle resizing?**

    - A) They automatically delete excess capacity.
    - B) They maintain a fixed size regardless of the number of elements.
    - C) They allocate new memory with larger capacity and copy elements over.
    - D) They compress the existing elements to fit the new data.
    - **Correct Option:** C
2. **Which of the following best describes a circular linked list?**

    - A) A linked list where the `next` pointer of the last node points to NULL.
    - B) A linked list where each node has two pointers: one to the next node and one to the previous node.
    - C) A linked list where the `next` pointer of the last node points to the first node.
    - D) A linked list with nodes in a random order.
    - **Correct Option:** C

1. **What advantage does a doubly linked list have over a singly linked list?**

    - A) It uses less memory per node.
    - B) It allows for direct access to elements by index.
    - C) It allows traversal in both directions.
    - D) It has a fixed size.
    - **Correct Option:** C
2. **What is the primary use of the `NULL` pointer in linked lists?**

    - A) To add new nodes to the list.
    - B) To indicate the end of the list.
    - C) To store data within the nodes.
    - D) To connect the list in a circular manner.
    - **Correct Option:** B

1. **Which operation is typically more efficient in a linked list compared to an array?**
    - A) Accessing elements by index.
    - B) Random access to elements.
    - C) Insertion at the start of the data structure.
    - D) Searching for an element by its value.
    - **Correct Option:** C
2. **Why might a circular linked list be preferred in a scenario where the data needs to be accessed in a round-robin manner?**
    - A) Because it can store an infinite number of elements.
    - B) Because it allows easier reversal of the list.
    - C) Because it enables continuous looping through the elements without needing to reset to the beginning manually.
    - D) Because it automatically sorts the elements.
    - **Correct Option:** C

1. **What is a potential disadvantage of using a singly linked list?**
    - A) It can only store integer data types.
    - B) It cannot be used to implement stacks or queues.
    - C) It requires more memory due to the storage of pointers.
    - D) It allows for fast random access to elements.
    - **Correct Option:** C
2. **When inserting a new node at the end of a singly linked list, if you do not maintain a tail pointer, what is the time complexity?**
    - A) O(1)
    - B) O(log n)
    - C) O(n)
    - D) O(n^2)
    - **Correct Option:** C

Coding Exercise: Implementing a Singly Linked List in C++

Objective: In this exercise, you will implement a basic singly linked list using `struct` in C++. You will create functions for inserting a node at the beginning, inserting a node at the end, displaying the list, and searching for an element in the list.

Step 1: Define the Node Structure

Start by defining the structure for a list node. Each node should contain an integer data field and a pointer to the next node.

```
struct Node {
    int data;
    Node* next;
};
```

Step 2: Inserting at the Beginning

Implement a function that inserts a new node at the beginning of the list. This function should take a pointer to the head of the list and the data to be inserted as parameters.

```
void insertAtBeginning(Node** head, int newData) {
    // Your code here
}
```

Step 3: Inserting at the End

Implement a function that inserts a new node at the end of the list. This function should also take a pointer to the head of the list and the data to be inserted.

```
void insertAtEnd(Node** head, int newData) {
    // Your code here
}
```

Step 4: Displaying the List

Create a function to print the elements of the list. Traverse from the head to the end, printing each element's data.

```
void displayList(Node* node) {
    // Your code here
}
```

Step 5: Searching for an Element

Implement a function that searches for an element in the list. It should return true if the element is found and false otherwise.

```
bool search(Node* head, int key) {
    // Your code here
}
```

Instructions:

1. **Implement `insertAtBeginning`:** Create a new node with the given data, point it to the current head, and update the head of the list.
2. **Implement `insertAtEnd`:** If the list is empty, the new node becomes the head. Otherwise, traverse to the end of the list and link the new node.
3. **Complete `displayList`:** Traverse the list from the head while printing the data of each node until the end is reached.
4. **Fill in `search`:** Traverse the list. If a node with the given data is found, return true. If the end of the list is reached, return false.

Sample Main Function:

```
int main() {
    Node* head = nullptr; // Start with an empty list

    // Insert elements
    insertAtBeginning(&head, 10);
    insertAtEnd(&head, 20);
    insertAtBeginning(&head, 5);
    displayList(head);

    // Search for elements
    if(search(head, 20)) {
        std::cout << "Element found." << std::endl;
    } else {
        std::cout << "Element not found." << std::endl;
    }

    return 0;
}
```

Structure of a Doubly Linked List Node

The structure of a node in a doubly linked list is an essential concept that forms the foundation of understanding how doubly linked lists work. Unlike nodes in a singly linked list, which have a pointer to the next node, nodes in a doubly linked list contain two pointers: one pointing to the next node and another pointing to the previous node. This dual-linkage allows traversal in both directions—forward and backward—through the list.

Components of a Doubly Linked List Node

1. **Data**: The data component of a doubly linked list node stores the value that the node represents. This could be a simple data type like an integer or a float, or more complex data structures like strings or objects, depending on the application's requirements.
2. **Next Pointer**: The next pointer (`next`) of a node points to the subsequent node in the list. For the last node in a doubly linked list, this pointer is set to `nullptr` (or `NULL` in some languages), indicating the end of the list when traversing forward.
3. **Previous Pointer**: The previous pointer (`prev`) points to the preceding node in the list. In the case of the first node, this pointer is set to `nullptr`, signifying that there is no node before it. This pointer is what distinguishes a doubly linked list from a singly linked list and enables backward traversal.

Visual Representation

To visualize the structure, consider a doubly linked list containing three nodes with integer data values. Here's a simple representation:

```
nullptr <- prev [1] next -> <-> prev [2] next -> <-> prev [3] next -> nullptr
```

- The `[1]`, `[2]`, and `[3]` represent the data stored in each of the three nodes.
- Arrows pointing right (`->`) represent the next pointers, indicating the direction to the next node.
- Arrows pointing left (`<-`) represent the prev pointers, showing the connection to the previous node.
- The first node’s prev pointer and the last node’s next pointer both point to `nullptr`, indicating the boundaries of the list.

Implementation

In C++ or similar languages, the node structure can be implemented as follows:

```
struct Node {
    int data;       // The data part
    Node* next;     // Pointer to the next node
    Node* prev;     // Pointer to the previous node

    // Constructor for convenience
    Node(int data) : data(data), next(nullptr), prev(nullptr) {}
};
```

Importance of the `prev` Pointer

The presence of the `prev` pointer in each node facilitates several operations that are either more complex or not possible in a singly linked list. For example, operations like deletion or insertion before a given node can be performed more efficiently because the list can be traversed backward to find the previous node, eliminating the need to keep an external track of it. Additionally, the ability to traverse in both directions makes doubly linked lists suitable for applications that require bidirectional iteration, such as navigating through a web browser's history.

Creation of a Doubly Linked List

Creating a doubly linked list involves initializing the list and defining a method for adding nodes to it. Initially, the list is empty, signified by a `nullptr` value for the head (the first node in the list). Nodes are then added to the list using various insertion methods, such as adding at the beginning, at the end, or after a specific node.

Initializing a Doubly Linked List

A doubly linked list is initialized by setting the head pointer to `nullptr`, indicating that the list is empty. This head pointer will be updated as nodes are added to or removed from the list.

Adding Nodes

1. **At the Beginning**: To add a node at the beginning, a new node is created with its next pointer set to the current head of the list, and its prev pointer set to `nullptr`. The current head node's prev pointer is updated to point to the new node. Finally, the head of the list is updated to point to the new node.
2. **At the End**: Adding a node at the end requires traversing the list to the last node and inserting the new node after it. The new node's prev pointer is set to the last node, and the last node's next pointer is updated to point to the new node. For an empty list, the new node is simply set as the head.
3. **After a Specific Node**: To insert a new node after a given node, the new node's next pointer is set to point to the given node's next node, and its prev pointer is set to the given node. Adjustments are made to the next node of the given node and the prev pointer of the next node (if it exists) to incorporate the new node into the list.

Traversal of a Doubly Linked List

Traversal in a doubly linked list can be performed in two directions: forward (from the head to the end) and backward (from the end to the head). This flexibility is due to the presence of both next and prev pointers in each node.

Forward Traversal

Forward traversal starts from the head of the list and moves through each node by following the next pointers until reaching a node that points to `nullptr`, indicating the end of the list.

```
void forwardTraversal(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }
    std::cout << std::endl;
}
```

Backward Traversal

Backward traversal begins from the last node, which requires first traversing the list to find the end. Once the end is reached, traversal moves backward through the list by following the prev pointers until reaching the first node, which has a prev pointer of `nullptr`.

```
void backwardTraversal(Node* head) {
    Node* current = head;
    // First, find the last node
    while (current != nullptr && current->next != nullptr) {
        current = current->next;
    }
    // Now, traverse backward from the last node
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->prev;
    }
    std::cout << std::endl;
}
```

Insertion in a Doubly Linked List

Insertion in a doubly linked list is a fundamental operation that involves adding a new node to the list. This operation can be performed in several ways, depending on where the new node is to be placed: at the beginning of the list, at the end, after a specific node, or before a specific node. Each of these insertion methods enhances the flexibility and utility of doubly linked lists in various applications.

Insertion at the Beginning

To insert a new node at the beginning of the list:

1. **Create a new node** with the given data.
2. **Set the new node's next pointer** to the current head of the list.
3. **Update the current head node's prev pointer** to point to the new node, if the list is not empty.
4. **Update the head of the list** to be the new node.
5. **Set the new node's prev pointer** to `nullptr`.

This operation effectively places the new node as the first node of the list, with the entire operation being O(1), or constant time complexity.

```
// Function to insert a new node at the beginning of the list
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node(newData);
    newNode->next = head; // Make next of new node as head
    if (head != nullptr) {
        head->prev = newNode; // Move the prev of head node to new node
    }
    head = newNode; // Move the head to point to the new node
}
```

Insertion at the End

To add a new node at the end of the list:

1. **Create a new node** with the given data.
2. If the list is empty, **set the head to the new node**.
3. If the list is not empty, **traverse to the last node** of the list.
4. **Set the last node's next pointer** to the new node.
5. **Set the new node's prev pointer** to the last node.
6. **Set the new node's next pointer** to `nullptr`.

This method appends the new node to the list, requiring traversal from the head to the end of the list, making it an O(n) operation, where n is the number of elements in the list.

```
// Function to insert a new node at the end of the list
void insertAtEnd(Node*& head, int newData) {
    Node* newNode = new Node(newData);
    if (head == nullptr) {
        head = newNode; // If the list is empty, make the new node as head
        return;
    }
    Node* lastNode = head;
    while (lastNode->next != nullptr) {
        lastNode = lastNode->next; // Traverse till the last node
    }
    lastNode->next = newNode; // Change the next of last node
    newNode->prev = lastNode; // Make last node as previous of new node
}
```

Insertion After a Specific Node

Inserting a new node after a specific node involves:

1. **Locate the specific node** after which the new node will be inserted.
2. **Create a new node** with the given data.
3. **Set the new node's next pointer** to the specific node's next node.
4. **Update the next node's prev pointer** (if it is not `nullptr`) to point to the new node.
5. **Set the specific node's next pointer** to the new node.
6. **Set the new node's prev pointer** to the specific node.

This operation allows insertion at any point within the list and is particularly efficient because it does not require traversal of the entire list, assuming the specific node is already known.

```
// Function to insert a node after a given node
void insertAfter(Node* prevNode, int newData) {
    if (prevNode == nullptr) {
        cout << "The given previous node cannot be nullptr" << endl;
        return;
    }
    Node* newNode = new Node(newData);
    newNode->next = prevNode->next; // Make next of new node as next of prevNode
    prevNode->next = newNode; // Make the next of prevNode as newNode
    newNode->prev = prevNode; // Make prevNode as previous of newNode

    if (newNode->next != nullptr) {
        newNode->next->prev = newNode; // Change previous of newNode's next node
    }
}
```

Insertion Before a Specific Node

Inserting a new node before a specific node can be slightly more complex and involves:

1. **Locate the specific node** before which the new node will be inserted. If the specific node is the head, follow the insertion at the beginning steps.
2. **Create a new node** with the given data.
3. **Set the new node's next pointer** to the specific node.
4. **Set the new node's prev pointer** to the specific node's prev node.
5. **Update the specific node's prev node's next pointer** (if it is not the head) to point to the new node.
6. **Update the specific node's prev pointer** to point to the new node.

This method allows for insertion directly before a known node, effectively inserting the new node into the desired position in the list without traversing it entirely if the specific node is known.

```
// Function to insert a new node before a specific node
void insertBefore(Node*& head, Node* nextNode, int newData) {
    if (nextNode == nullptr) {
        cout << "The given next node cannot be nullptr" << endl;
        return;
    }
    Node* newNode = new Node(newData);
    newNode->prev = nextNode->prev;
    newNode->next = nextNode;
    nextNode->prev = newNode;
    if (newNode->prev != nullptr) {
        newNode->prev->next = newNode;
    } else {
        head = newNode; // If the nextNode is head, then update the head to newNode
    }
}
```

`main()` Function for Insertion

```
#include <iostream>
using namespace std;

// Define the structure for a node in the doubly linked list
struct Node {
    int data;
    Node* prev;
    Node* next;

    // Constructor to create a new node
    Node(int data) : data(data), prev(nullptr), next(nullptr) {}
};

// All insertion functions
```

```
// Helper function to print the contents of the doubly linked list
void printList(Node* node) {
    Node* last = nullptr;
    cout << "Traversal in forward direction: ";
    while (node != nullptr) {
        cout << node->data << " ";
        last = node;
        node = node->next;
    }
    cout << endl;

    cout << "Traversal in reverse direction: ";
    while (last != nullptr) {
        cout << last->data << " ";
        last = last->prev;
    }
    cout << endl;
}



int main() {
    Node* head = nullptr; // Start with the empty list

    insertAtBeginning(head, 10); // List: 10
    insertAtEnd(head, 20);       // List: 10 <-> 20
    insertAtEnd(head, 30);       // List: 10 <-> 20 <-> 30
    insertAtBeginning(head, 5);  // List: 5 <-> 10 <-> 20 <-> 30

    Node* secondNode = head->next; // Assuming secondNode points to the node with d
ata 10
    insertAfter(secondNode, 15);   // List: 5 <-> 10 <-> 15 <-> 20 <-> 30

    Node* thirdNode = secondNode->next; // Node with data 15
    insertBefore(head, thirdNode, 12);  // List: 5 <-> 10 <-> 12 <-> 15 <-> 20 <->
30

    cout << "Created DLL is: " << endl;
    printList(head);

    return 0;
}
```

Deletion in a Doubly Linked List

Deletion in a doubly linked list is an operation that removes a node from the list. This process varies depending on the location of the node to be deleted—whether it is the first node, the last node, or a node located somewhere in the middle of the list. Due to the bidirectional nature of doubly linked lists, deletion is more straightforward compared to singly linked lists, as each node directly references both its predecessor and successor.

Deleting the First Node

To delete the first node in a doubly linked list:

1. Check if the head (first node) is not `nullptr` (i.e., the list is not empty).
2. Update the head to the next node in the list.
3. If the new head is not `nullptr`, set its `prev` pointer to `nullptr`.
4. Delete the old head node to free the allocated memory.

This operation removes the first node and updates the head of the list, ensuring that the list remains intact.

```
// Function to delete the first node of the list
void deleteFirstNode(Node*& head) {
    if (head == nullptr) return; // List is empty
    Node* temp = head;
    head = head->next; // Update head to the next node
    if (head != nullptr) {
        head->prev = nullptr; // Set the previous of new head to nullptr
    }
    delete temp; // Free the old head
}
```

Deleting the Last Node

To remove the last node:

1. Check if the list is not empty.
2. If there is only one node, delete it and set the head to `nullptr`.
3. Otherwise, traverse to the last node.
4. Set the `next` pointer of the second-to-last node to `nullptr`.
5. Delete the last node.

This method involves traversing to the end of the list, which has a linear time complexity relative to the length of the list.

```
// Function to delete the last node of the list
void deleteLastNode(Node*& head) {
    if (head == nullptr) return; // List is empty
    if (head->next == nullptr) { // List has only one node
        delete head;
        head = nullptr;
        return;
    }
    Node* last = head;
    while (last->next != nullptr) {
        last = last->next;
    }
    last->prev->next = nullptr; // Set the next of second last to nullptr
    delete last; // Delete the last node
}
```

Deleting a Node at a Specific Position

To delete a node located at a specific position in the list:

1. If the position is the first node, follow the procedure for deleting the first node.
2. Otherwise, locate the node at the given position by traversing the list.
3. Update the `next` pointer of the node's predecessor to point to the node's successor.
4. Similarly, update the `prev` pointer of the node's successor, if it exists, to point to the node's predecessor.
5. Delete the target node.

This process requires finding the node to be deleted, which involves traversing the list from the beginning.

```
// Function to delete a node at a specific position
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr || position < 0) return;
    Node* current = head;
    for (int i = 0; current != nullptr && i < position; i++) {
        current = current->next;
    }
    if (current == nullptr) return; // Position is out of bounds
    deleteNode(head, current);
}
```

Deleting a Given Node Directly

When a pointer or reference to the node to be deleted is available:

1. If the node is the first node, adjust the head pointer.
2. Update the `next` pointer of the node's predecessor, if it is not `nullptr`, to skip the current node and point to the node's successor.
3. Similarly, update the `prev` pointer of the node's successor, if it exists, to point to the node's predecessor.
4. Delete the node to free up the memory.

This operation is efficient as it does not require traversing the list to locate the node, assuming its address is already known.

```
// Function to delete a given node (when a pointer to the node is known)
void deleteNode(Node*& head, Node* delNode) {
    if (head == nullptr || delNode == nullptr) return;
    if (head == delNode) head = delNode->next;
    if (delNode->next != nullptr) delNode->next->prev = delNode->prev;
    if (delNode->prev != nullptr) delNode->prev->next = delNode->next;
    delete delNode;
}
```

`main()` Function for Deletion

```
#include <iostream>
using namespace std;

// Define the structure for a node in the doubly linked list
struct Node {
    int data;
    Node* prev;
    Node* next;

    // Constructor to create a new node
    Node(int data) : data(data), prev(nullptr), next(nullptr) {}
};
// Functions for insertion
// Function to print the list
```

```
int main() {
    Node* head = nullptr;
    // Populate the list
    insertAtEnd(head, 1);
    insertAtEnd(head, 2);
    insertAtEnd(head, 3);
    insertAtEnd(head, 4);
    insertAtEnd(head, 5);

    cout << "Original list: ";
    printList(head);



// Perform deletion operations
    deleteFirstNode(head);
    cout << "After deleting the first node: ";
    printList(head);

    deleteLastNode(head);
    cout << "After deleting the last node: ";
    printList(head);

    deleteNodeAtPosition(head, 1); // Delete node at position 1 (0-based indexing)
    cout << "After deleting a node at position 1: ";
    printList(head);

    // Clean up remaining nodes to prevent memory leaks
    while (head != nullptr) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }

    return 0;
}
```

Introduction to Circular Linked Lists

A Circular Linked List is a variation of the traditional linked list where the last node points back to the first node instead of pointing to `nullptr` or `NULL`. This creates a circular structure that allows for any node to be treated as the starting point, thereby offering unique advantages and applications over its linear counterparts.

Key Characteristics

- **Circularity**: The defining feature is that the list forms a circle, with each node connected to the next, and the last node linked back to the first.
- **No Natural End**: Unlike singly or doubly linked lists, circular linked lists do not have a natural starting or ending point, which can be both an advantage and a complexity in traversal and manipulation.
- **Efficiency**: Circular linked lists can be more efficient in scenarios where a list needs to be repeatedly looped over or when implementing cyclically repeating tasks.

Applications

- Circular linked lists are used in the implementation of data structures that require circular traversal, such as round-robin scheduling algorithms.
- They are suitable for applications needing a circular queue where the data needs to be processed and recycled, such as buffering streamed data.
- Circular linked lists are also used in multiplayer board games where the turn goes back to the first player after the last player.

Structure of a Circular Linked List Node

The node structure of a circular linked list is similar to that of a singly linked list, with a `data` field and a `next` pointer. However, in the circular variant, the `next` pointer of the last node points back to the first node, creating a closed loop.

Components

1. **Data**: Holds the value stored in the node. This could be of any data type, including integers, floating-point numbers, or more complex data structures.
2. **Next Pointer**: Points to the next node in the list. In the case of the last node, this pointer directs back to the first node, maintaining the circular structure.

Implementation Example

In C++ or similar languages, the basic structure of a node can be implemented as follows:

```
struct Node {
    int data; // The stored data
    Node* next; // Pointer to the next node

    // Constructor for creating a new Node with data
    Node(int data) : data(data), next(nullptr) {}
};
```

Circular Linked List Variants

- **Singly Circular Linked List**: This is the simplest form where each node points to the next node, and the last node points back to the first node, forming a single loop.
- **Doubly Circular Linked List**: In this more complex form, each node has two pointers: `next` and `prev`. `next` points to the next node in the list, and `prev` points to the previous node, with the last node's `next` pointing to the first node and the first node's `prev` pointing to the last node, creating a bidirectional loop.

Code Exercise: Implementing a Circular Linked List

This exercise involves implementing a basic circular linked list in C++. A circular linked list is a variation of the linked list where the last node points back to the first node, forming a loop. You will implement functions to add elements to the list, remove elements, and display the list's contents.

Node Structure

Begin with defining the structure of a node in the circular linked list:

```
struct Node {
    int data;
    Node* next;

    Node(int data) : data(data), next(nullptr) {}
};
```

Function Headers

Here are the function headers you will need to implement:

```
void insertAtEnd(Node*& head, int data);
void insertAtBeginning(Node*& head, int data);
void deleteNode(Node*& head, int key);
void displayList(Node* head);
```

Starter Code

Below is the starter code, including the `main()` function, where you can test the functionality of your circular linked list:

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;

    Node(int data) : data(data), next(nullptr) {}
};



void insertAtEnd(Node*& head, int data) {
    // TODO: Implement this function
}

void insertAtBeginning(Node*& head, int data) {
    // TODO: Implement this function
}

void deleteNode(Node*& head, int key) {
    // TODO: Implement this function
}

void displayList(Node* head) {
    // TODO: Implement this function
}
```

```
int main() {
    Node* head = nullptr; // Initialize the circular linked list as empty

    // Add elements to the list
    insertAtEnd(head, 1);
    insertAtEnd(head, 2);
    insertAtBeginning(head, 3);
    displayList(head); // Expected output: 3 1 2

    // Delete an element from the list
    deleteNode(head, 1);
    displayList(head); // Expected output: 3 2

    return 0;
}
```

Reflection MCQs

1. **Which of the following statements is true for doubly linked lists?**
    - A. Each node contains two pointers: next and prev.
    - B. They cannot be traversed in reverse.
    - C. They use less memory than singly linked lists.
    - D. They do not allow for insertion at the end.
    - **Answer: A**
2. **In a circular linked list, the last node's next pointer will point to:**
    - A. NULL
    - B. The second node
    - C. The head node
    - D. Itself
    - **Answer: C**

1. **Which operation is more efficient in a doubly linked list compared to a singly linked list?**
    - A. Adding a node at the beginning
    - B. Deleting the last node
    - C. Accessing a middle node by index
    - D. Traversing the list forwards
    - **Answer: B**
2. **What makes circular linked lists particularly useful for implementing queues?**
    - A. They automatically prioritize elements.
    - B. The need for dynamic resizing is eliminated.
    - C. The last element directly points to the first element, facilitating circular traversal.
    - D. They can only store a fixed number of elements.
    - **Answer: C**

1. **Which of the following is a disadvantage of doubly linked lists over singly linked lists?**
    - A. Higher memory usage
    - B. Inability to traverse backwards
    - C. Slower insertion operations
    - D. Limited by a fixed size
    - **Answer: A**
2. **In the context of data structures, LIFO stands for:**
    - A. Last In, First Out
    - B. Linear In, First Out
    - C. Last In, Fast Out
    - D. Linear Interface, First Operation
    - **Answer: A**

1. **Which data structure would be most efficient for implementing an undo functionality in a text editor?**
    - A. Array
    - B. Singly linked list
    - C. Stack
    - D. Queue
    - **Answer: C**
2. **Circular linked lists are ideal for which scenario?**
    - A. When data needs to be stored in a strictly linear order
    - B. When you frequently need to process elements in a circular, repeating order
    - C. For applications that require fast random access to elements
    - D. When memory usage needs to be minimized at all costs
    - **Answer: B**

1. **What is the time complexity of inserting a node at the beginning of a doubly linked list?**

   - A. O(1)
   - B. O(n)
   - C. O(log n)
   - D. O(n^2)
   - **Answer: A**

2. **Which of the following is not a direct application of stacks?**

   - A. Browser history navigation
   - B. Round-robin scheduling
   - C. Syntax parsing for compilers
   - D. Evaluating postfix expressions
   - **Answer: B**

1. **In a doubly linked list, deleting a node (not the first or the last) requires updating:**

   - A. Only the next pointer of the preceding node
   - B. Only the prev pointer of the following node
   - C. Both the next pointer of the preceding node and the prev pointer of the following node
   - D. No pointers need to be updated
   - **Answer: C**

2. **A major benefit of circular linked lists over linear linked lists is:**

   - A. The ability to expand indefinitely without memory allocation
   - B. The elimination of the need for a head pointer
   - C. Efficient addition and removal of nodes from both ends
   - D. They do not require tracking the last node for circular traversal
   - **Answer: D**

1. **Which of the following scenarios is best suited for a circular linked list?**

   - A. Implementing a playlist that loops back to the first song after the last song finishes
   - B. A priority queue for managing tasks
   - C. Storing elements that require frequent, random access
   - D. Implementing a LIFO stack for a text editor's undo feature
   - **Answer: A**

2. **In which case would a doubly linked list not have an advantage over a singly linked list?**

   - A. When you need to traverse the list in both directions
   - B. When memory usage is a critical concern
   - C. When inserting elements at the list's end
   - D. When deleting the last element of the list
   - **Answer: B**

1. **A circular linked list can be used to implement:**

   - A. A stack
   - B. A queue
   - C. Both A and B
   - D. Neither A nor B
   - **Answer: C**
2. Which of the following operations is more efficient in a doubly linked list compared to a singly linked list?

   - A) Traversing to the last node
   - B) Deleting the last node
   - C) Inserting a node at the beginning
   - D) **Both B and C**

1. A circular doubly linked list is characterized by:

   - A) The prev pointer of the first node pointing to `nullptr`
   - B) **The next pointer of the last node pointing to the first node and the prev pointer of the first node pointing to the last node**
   - C) The absence of a head pointer
   - D) Nodes containing only data and a single pointer
2. In a circular linked list, adding a new node before the head node requires updating:

   - A) Only the new node's next pointer
   - B) Only the last node's next pointer
   - C) **Both the new node's next pointer and the last node's next pointer**
   - D) The head pointer only

1. The primary advantage of using a circular linked list over a singly linked list is:

   - A) Reduced memory usage
   - B) **Efficiency in implementing cyclically operating structures**
   - C) Simpler implementation
   - D) Better time complexity for insertion operations
2. Which of the following data structures is ideal for implementing a browser's back-forward feature?

   - A) Singly Linked List
   - B) **Doubly Linked List**
   - C) Circular Singly Linked List
   - D) Circular Doubly Linked List

1. A stack can be implemented using:

    - A) Only arrays
    - B) Only singly linked lists
    - C) **Both arrays and linked lists**
    - D) Neither arrays nor linked lists
2. Deleting a node in a doubly linked list when only a pointer to that node is given requires adjustment of:

    - A) Only the next pointers of adjacent nodes
    - B) Only the prev pointers of adjacent nodes
    - C) **Both next and prev pointers of adjacent nodes**
    - D) Neither next nor prev pointers

1. The time complexity of adding an element to a circular linked list is:

    - A) O(1)
    - B) **O(n) if adding to the end without a tail pointer**
    - C) O(log n)
    - D) O(n^2)
2. A circular linked list can be used to implement:

    - A) A stack
    - B) **A round-robin scheduler**
    - C) Binary Search Tree
    - D) Hash Table

1. The major drawback of a doubly linked list over a singly linked list is:

    - A) Lower time complexity for operations
    - B) **Higher memory consumption per node**
    - C) More difficult to implement traversal
    - D) It cannot be circular
2. In which scenario is a circular linked list not the best option?

    - A) Implementing a playlist that loops
    - B) **Storing data where the end of the list needs to be marked distinctly**
    - C) Implementing a game where the turn goes back to the first player after the last
    - D) Designing a real-time application that requires circular traversal of tasks

1. To reverse a doubly linked list, you must:

    - A) Swap the data in each node
    - B) **Swap the next and prev pointers in each node**
    - C) Create a new list with reversed order
    - D) It's not possible to reverse a doubly linked list
2. The deletion of a node in a singly linked list requires traversal from the head to:

    - A) The node to be deleted only
    - B) **The node immediately before the node to be deleted**
    - C) The last node of the list
    - D) The node immediately after the node to be deleted

1. Which of the following is NOT a direct application of circular linked lists?

    - A) Undo functionality in text editors
    - B) Managing running processes in an operating system
    - C) **Balancing parentheses in expressions**
    - D) Implementing a snake game where the snake can wrap around edges
2. Which operation generally has the same time complexity in singly linked lists, doubly linked lists, and circular linked lists?

    - A) Inserting a node at a specific position
    - B) Deleting the last node
    - C) **Inserting a node at the beginning**
    - D) Searching for a node with a specific value

Make sure that you have:

- **completed all the concepts**
- **ran all code examples yourself**
- **tried changing little things in the code to see the impact**
- **implemented the required programs**

Please feel free to share your problems to ensure that your weekly tasks are completed and you are not staying behind