Linked Lists

quarta-feira, 16 de outubro de 2024 13:48

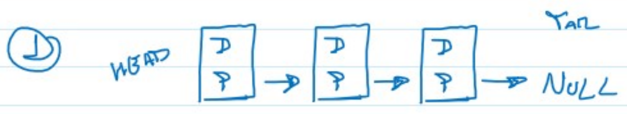
Linear Data Structure where elements are not storaged in a sequential way

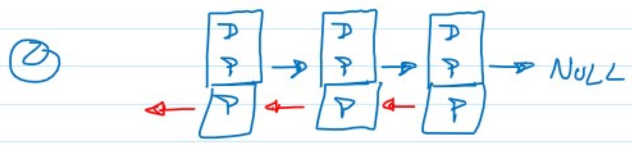
↳ Each element (called Node) contains

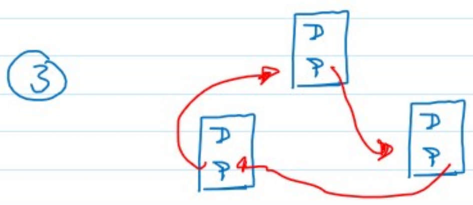
{ Data ⟶ detail information
{ Pointer Ref ⟶ Pointer to the next element

Types of Linked Lists

1. **Singly Linked List**: Each node points to the next node and the last node points to `null`.
2. **Doubly Linked List**: Each node has two pointers: one to the next node and another to the previous node.
3. **Circular Linked List**: The last node points back to the first node, making the list circular.







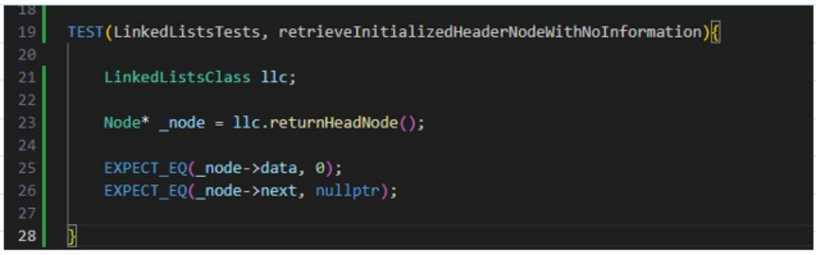
→ Approach for implementation

Approach:

- *Define a **structure Node** having two members **data** and a **next** pointer. The data will store the value of the node and the next pointer will store the address of the next node in the sequence. For doubly linked list you will have to add an addition pointer **prev** that will store the address of the prev node in the sequence.*
- *Define a class **LinkedList** consisting of all the member functions for the **LinkedList** and a **head** pointer that will store the reference of a particular linked list.*
- *Initialize the **head** to **NULL** as the linked list is empty initially.*
- *Implement basic functions like **insertAtBeginning**, **insertAtEnd**, **deleteFromBeginning**, **deleteFromEnd** that*

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

→ Defined as a pointer to the next structure



```
TEST(LinkedListsTests, retrieveInitializedHeaderNodeWithNoInformation){

    LinkedListsClass llc;

    Node* _node = llc.returnHeadNode();

    EXPECT_EQ(_node->data, 0);
    EXPECT_EQ(_node->next, nullptr);

}
```

→ Here is no need to do. Initialize the class with the head as a constructor

↳ The idea is to have both data = 0 and pointer to next

implement basic functions like
insertAtBeginning, insertAtEnd, deleteFromBeginning, deleteFromEnd that will manipulate the elements of the linked list.

↳ The idea is to have both data = 0 and pointer to next as null ptr

Basic Operations of a Linked List

Following are some basic operations which are required to manipulate the nodes of a linked list:

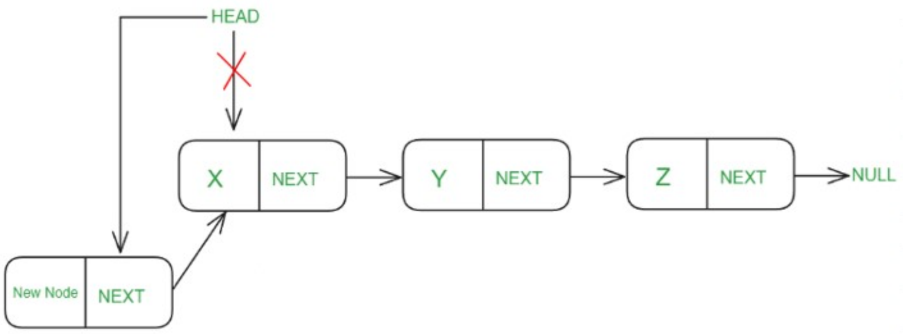
Operation	Description	Time Complexity	Space Complexity
insertAtBeginning	Attaches a new node at the start of the linked list.	O(1)	O(1)
insertAtEnd	Attaches a new node at the end of the linked list.	O(n)	O(1)
insertAtPosition	Attaches a new node at a specific position in the linked list.	O(n)	O(1)
deleteFromBeginning	Removes the Head of the linked list and updates the Head to the next node	O(1)	O(1)
deleteFromEnd	Removes the node present at the end of the linked list.	O(n)	O(1)
deleteFromPosition	Removes a node from a given position of the linked list.	O(n)	O(1)
Display	Prints all the values of the nodes present in the linked list.	O(n)	O(1)

Note: Here **n** denotes the number of nodes in the linked list.

Now let's learn how we can implement these basic functions of linked list in C++:

Algorithm for insertAtBeginning Implementation

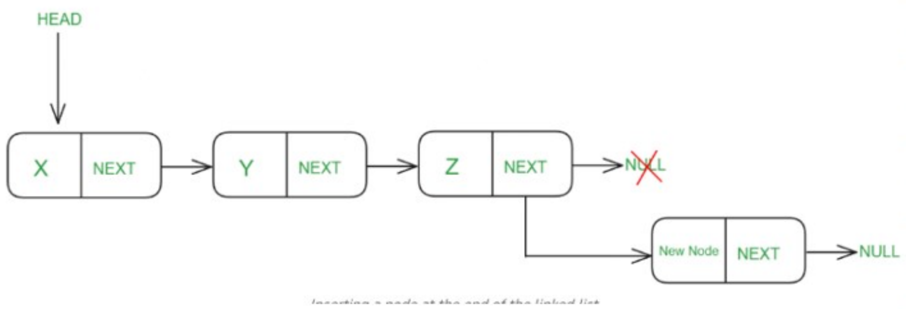
1. Create a new Node
2. Point the new Node's next pointer to the current head.
3. Update the head of the linked list as the new node.



Inserting a node at the first of the linked list

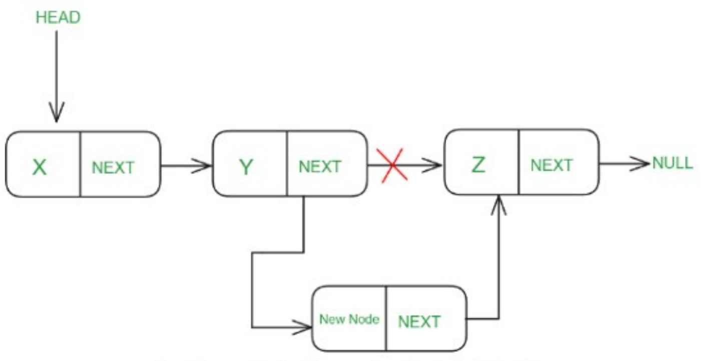
Algorithm for insertAtEnd Implementation

1. Create a new Node
2. If the linked list is empty, update the head as the new node.
3. Otherwise traverse till the last node of the linked list.
4. Update the next pointer of the last node from NULL to new node.



Algorithm for insertAtPosition Implementation

1. Check if the provided position by the user is a valid poistion.
2. Create a new node.
3. Find the node at position -1.
4. Update the next pointer of the new node to the next pointer of the current node.
5. Update the next pointer of the current node to new node.



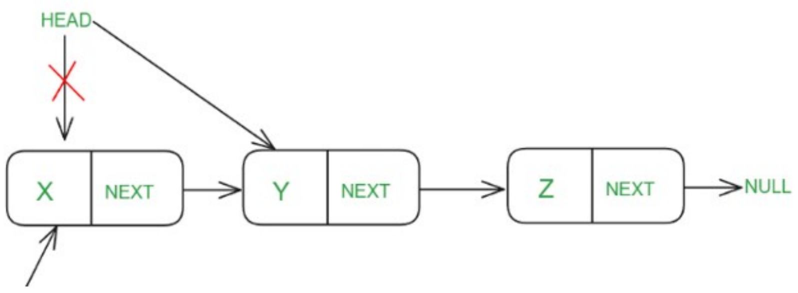
Inserting a node after the second position in the linked list

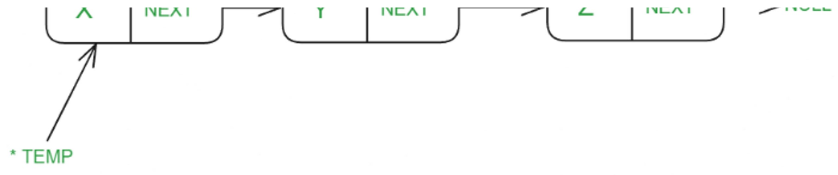
Algorithm for deleteFromBeginning Implementation

1. Check whether the **Head** of the linked list is not NULL. If Head is equal to NULL return as the linked list is empty, there is no node present for deletion.
2. Store the head of the linked list in a temp pointer.
3. Update the head of the linked list to next node.
4. Delete the temporary node stored in the temp pointer.

When removing or doing any other thing with the head pointer as parameter

remember to reference it back to the original head

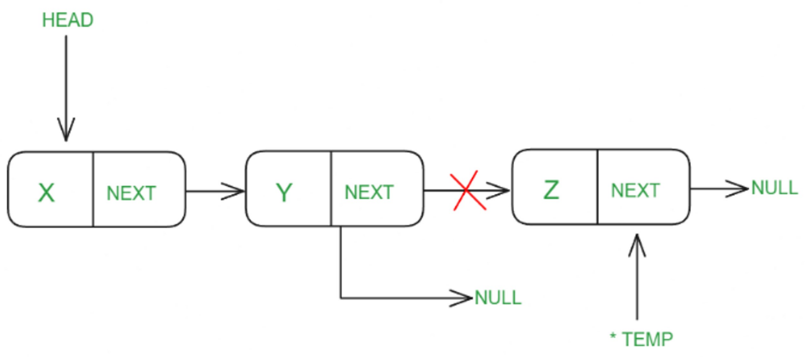




Deleting the first node of the linked list

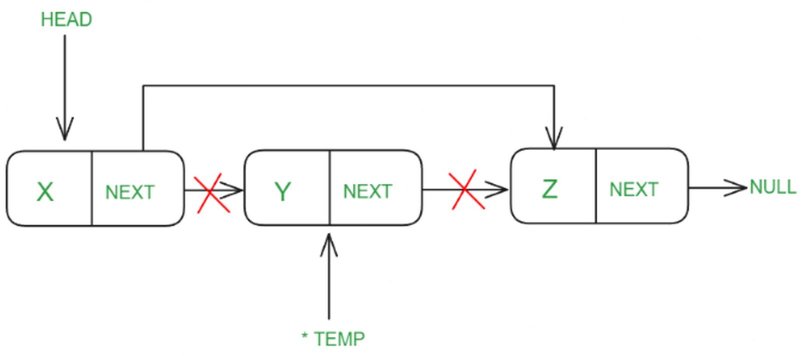
Algorithm for deleteFromEnd Implementation

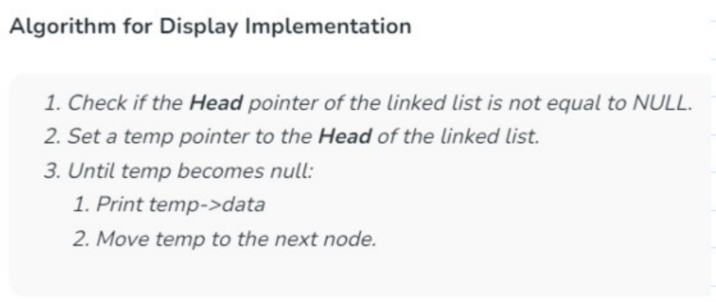
1. *Verify whether the linked is empty or not before deletion.*
2. *If the linked list has only one node, delete head and set head to NULL.*
3. *Traverse till the second last node of the linked list.*
4. *Store the last node of the linked list in a temp pointer.*
5. *Pointer the next pointer of the second last node to NULL.*
6. *Delete the node represented by the temp pointer.*



Algorithm for deleteFromPosition Implementation

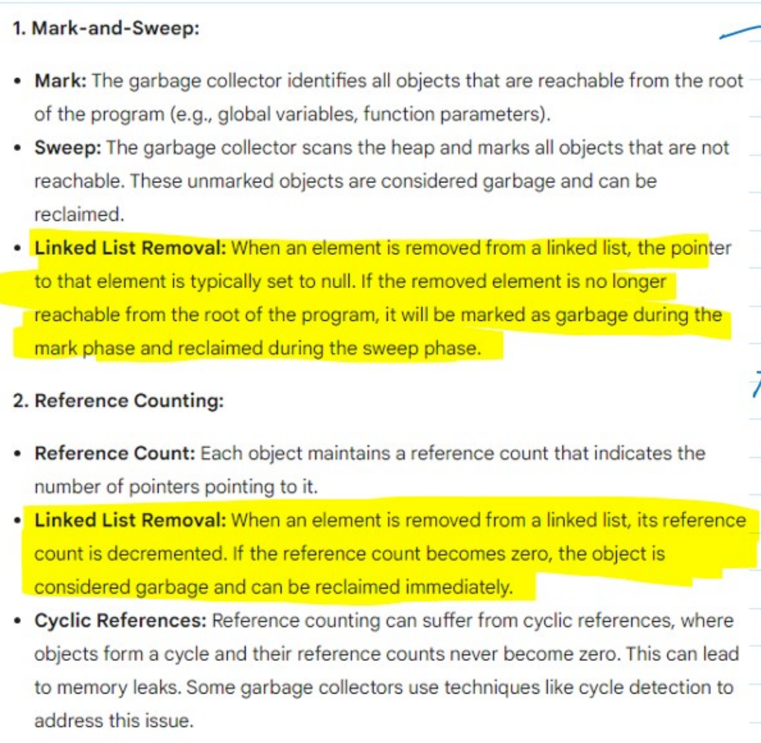
1. *Check if the provided postion by the users is a valid position in the linked list or not.*
2. *Find the node at position -1.*
3. *Save node to be deleted in a temp pointer.*
4. *Set the next pointer of the current node to the next pointer of the node to be deleted.*
5. *Set the next pointer of temp to NULL.*
6. *Delete the node represented by temp pointer.*





Algorithm for Display Implementation

1. *Check if the **Head** pointer of the linked list is not equal to NULL.*
2. *Set a temp pointer to the **Head** of the linked list.*
3. *Until temp becomes null:*
    1. *Print temp->data*
    2. *Move temp to the next node.*



1. Mark-and-Sweep:

- **Mark:** The garbage collector identifies all objects that are reachable from the root of the program (e.g., global variables, function parameters).
- **Sweep:** The garbage collector scans the heap and marks all objects that are not reachable. These unmarked objects are considered garbage and can be reclaimed.
- **Linked List Removal:** When an element is removed from a linked list, the pointer to that element is typically set to null. If the removed element is no longer reachable from the root of the program, it will be marked as garbage during the mark phase and reclaimed during the sweep phase.

2. Reference Counting:

- **Reference Count:** Each object maintains a reference count that indicates the number of pointers pointing to it.
- **Linked List Removal:** When an element is removed from a linked list, its reference count is decremented. If the reference count becomes zero, the object is considered garbage and can be reclaimed immediately.
- **Cyclic References:** Reference counting can suffer from cyclic references, where objects form a cycle and their reference counts never become zero. This can lead to memory leaks. Some garbage collectors use techniques like cycle detection to address this issue.

→ How the garbage collector works

↳ After some time it'll identify that there is no pointer to that memory adress !!