Remove duplicates from a sorted linked list

Write a function that takes a list sorted in non-decreasing order and deletes any duplicate nodes from the list. The list should only be traversed once.

For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates( ) should convert the list to 11->21->43->60.

Algorithm:

Traverse the list from the head (or start) node. While traversing, compare each node with its next node. If the data of the next node is the same as the current node then delete the next node. Before we delete a node, we need to store the next pointer of the node

Implementation:

Functions other than removeDuplicates() are just to create a linked list and test removeDuplicates( ).

```
/* C++ Program to remove duplicates from a sorted linked list
#include <bits/stdc++.h>

using namespace std;

/* Link list node */
class Node
{
    public:
    int data;
    Node* next;
};

/* The function removes duplicates from a sorted list */
void removeDuplicates(Node* head)
{
    /* Pointer to traverse the linked list */
    Node* current = head;

    /* Pointer to store the next pointer of a node to be dele

    Node* next_next;

    /* do nothing if the list is empty */
    if (current == NULL)
    return;

    /* Traverse the list till last node */
    while (current->next != NULL)
    {
    /* Compare current node with next node */
    if (current->data == current->next->data)
    {
        /* The sequence of steps is important*/
        next_next = current->next->next;
        free(current->next);
        current->next = next_next;
    }
    else /* This is tricky: only advance if no deletion */
        current = current->next;
    }
    }
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the linked l
void push(Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = new Node();

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */
void printList(Node *node)
{
    while (node!=NULL)
    {
        cout<<" "<<node->data;
        node = node->next;
    }
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    Node* head = NULL;

    /* Let us create a sorted linked list to test the functio
    Created linked list will be 11->11->11->13->13->20 */
    push(&head, 20);
    push(&head, 13);
    push(&head, 13);
    push(&head, 11);
    push(&head, 11);
    push(&head, 11);

    cout<<"Linked list before duplicate removal ";
    printList(head);

    /* Remove duplicates from linked list */
    removeDuplicates(head);

    cout<<"\nLinked list after duplicate removal ";
    printList(head);

    return 0;
}
```

Output

```
 Linked list before duplicate removal  11 11 11 13 13 20
Linked list after duplicate removal  11 13 20
```

Time Complexity: O(n) where n is the number of nodes in the given linked list.

Space Complexity: O(1) , as there is no extra space used.

Recursive Approach :

```
/* C++ Program to remove duplicatesfrom a sorted linked list
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
class Node
{
    public:
    int data;
    Node* next;
};

/* The function removes duplicates from a sorted list */
void removeDuplicates(Node* head)
{
    /* Pointer to store the pointer of a node to be deleted*/
    Node* to_free;

    /* do nothing if the list is empty */
    if (head == NULL)
        return;

    /* Traverse the list till last node */
    if (head->next != NULL)
    {

        /* Compare head node with next node */
        if (head->data == head->next->data)
        {
            /* The sequence of steps is important.
            to_free pointer stores the next of head
            pointer which is to be deleted.*/
            to_free = head->next;
        head->next = head->next->next;
        free(to_free);
        removeDuplicates(head);
        }
        else /* This is tricky: only
        advance if no deletion */
        {
            removeDuplicates(head->next);
        }
    }
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at thebeginning of the linked li
void push(Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = new Node();

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}
```

}

```
/* Function to print nodes in a given linked list */
void printList(Node *node)
{
    while (node!=NULL)
    {
        cout<<" "<<node->data;
        node = node->next;
    }
}

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

    /* Let us create a sorted linked
    list to test the functions
    Created linked list will be
    11->11->11->13->13->20 */
    push(&head, 20);
    push(&head, 13);
    push(&head, 13);
    push(&head, 11);
    push(&head, 11);
    push(&head, 11);

    cout<<"Linked list before duplicate removal ";
    printList(head);

    /* Remove duplicates from linked list */
    removeDuplicates(head);

    cout<<"\nLinked list after duplicate removal ";
    printList(head);

    return 0;
}
```

Output

```
Linked list before duplicate removal  11 11 11 13 13 20
Linked list after duplicate removal  11 13 20
```

Time Complexity: O(n) where n is the number of nodes in the given linked list.

Auxiliary Space: O(n)

Another Approach: Create a pointer that will point towards the first occurrence of every element and another pointer temp which will iterate to every element and when the value of the previous pointer is not equal to the temp pointer, we will set the pointer of the previous pointer to the first occurrence of another node.

Below is the implementation of the above approach:

```
// C++ program to remove duplicates
// from a sorted linked list
#include <bits/stdc++.h>
using namespace std;

// Linked list Node
struct Node {
    int data;
    Node* next;
    Node(int d)
    {
        data = d;
        next = NULL;
    }
};

// Function to remove duplicates
// from the given linked list
Node* removeDuplicates(Node* head)
{
    // Two references to head temp will iterate to the whole
    // Linked List prev will point towards the first
    // occurrence of every element
    Node *temp = head, *prev = head;
    // Traverse list till the last node
    while (temp != NULL) {
        // Compare values of both pointers
        if (temp->data != prev->data) {
            // if the value of prev is not equal to the
            // value of temp that means there are no more
            // occurrences of the prev data-> So we can set
            // the next of prev to the temp node->*/
            prev->next = temp;
            prev = temp;
        }
        //Set the temp to the next node
        temp = temp->next;
    }
    // This is the edge case if there are more than one
    // occurrences of the last element
    if (prev != temp)
        prev->next = NULL;
    return head;
}

Node* push(Node* head, int new_data)
{
    /* 1 & 2: Allocate the Node & Put in the data*/
    Node* new_node = new Node(new_data);
    /* 3. Make next of new Node as head */
    new_node->next = head;
    /* 4. Move the head to point to new Node */
    head = new_node;
    return head;
}

/* Function to print linked list */
void printList(Node* head)
{
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

/* Driver code */
int main()
{
    Node* llist = NULL;
    llist = push(llist, 20);
    llist = push(llist, 13);
    llist = push(llist, 13);
    llist = push(llist, 11);
    llist = push(llist, 11);
    llist = push(llist, 11);
    cout << ("List before removal of duplicates\n");
    printList(llist);
    cout << ("List after removal of elements\n");
    llist = removeDuplicates(llist);
    printList(llist);
}
```

Output

```
 List before removal of duplicates
11 11 11 13 13 20
List after removal of elements
11 13 20
```

Time Complexity: O(n) where n is the number of nodes in the given linked list.

Auxiliary Space: O(1)

Another Approach: Using Maps

The idea is to push all the values in a map and printing its keys.

Below is the implementation of the above approach:

```
// CPP program for the above approach
#include <bits/stdc++.h>
using namespace std;
```

```
/* Link list node */
struct Node {
    int data;
    Node* next;
    Node()
    {
        data = 0;
        next = NULL;
    }
};

/* Function to insert a node at
the beginning of the linked * list */
void push(Node** head_ref, int new_data)
{

    /* allocate node */
    Node* new_node = new Node();

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off
    the new node */
    new_node->next = (*head_ref);

    /* move the head to point
    to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */
void printList(Node* node)
{
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
}

// Function to remove duplicatesvoid
removeDuplicates(Node* head)
{
    unordered_map<int, bool> track;
    Node* temp = head;
    while (temp) {
        if (track.find(temp->data) == track.end()) {
            cout << temp->data << " ";
        }
        track[temp->data] = true;
        temp = temp->next;
    }
}

// Driver Codeint main()
{
    Node* head = NULL;

    /* Created linked list will be
    11->11->11->13->13->20 */
    push(&head, 20);
    push(&head, 13);
    push(&head, 13);
    push(&head, 11);
    push(&head, 11);
    push(&head, 11);

    cout << "Linked list before duplicate removal ";
    printList(head);

    cout << "\nLinked list after duplicate removal ";
    removeDuplicates(head);

    return 0;
}
```

Output

```
Linked list before duplicate removal 11 11 11 13 13 20
Linked list after duplicate removal 11 13 20
```

Time Complexity: O(n) where n is the number of nodes in the given linked list.

Space Complexity: O(n)