Report by : - Siddhant Barua & Jinggui Tan
CWid : - 10439929 & 10427081
Assignment 2 Code Report

Assignment 2 Report

Remove duplicates from a Link List using STL Containers (Set).

The first implementation of the Linked List removes duplicate nodes in the Linked List using an additional STL container. We have used sets for this, and the following image describes our code.

```
// First implementation; uses additional data structure
void removeDupl_1(Node* head)
{
    // no Node or only one Node in the LinkedList, do nothing
    if(head == 0 || head->next == 0){
        return;
    }
    // set: no duplicate data
    set <int, greater<int>> tempSet;
    tempSet.insert(head->data);

    Node* currentNode = head->next;
    Node* lastNode = head;
    while(currentNode !=0){
        if(tempSet.count(currentNode->data) == 0){
            tempSet.insert(currentNode->data);
            lastNode = currentNode;
            currentNode = currentNode->next;
        }else{
            Node* tempNode = currentNode;
            lastNode->next = currentNode->next;
            currentNode = currentNode->next;
            delete tempNode;
        }
    }
}
```

In the above code we have a time complexity of - O(n).
We use set to help us find duplicates then we don’t need to find that one by one.

Report by : - Siddhant Barua & Jinggui Tan
CWid : - 10439929 & 10427081
Assignment 2 Code Report

Remove duplicates from a Linked List without using STL Containers

Now we do the same remove duplicate function but without any additional STL Containers. The following image describes our code.

```cpp
void removeDupl_2(Node* head)
{
    // take the ith node and compare with each one, remove the one who has same data.
    Node* currentNode = head;
    Node* lastNode;
    Node* compareNode;

    // no Node or only one Node in the LinkedList, do nothing
    if(head == 0 || head->next == 0){
        return;
    }

    // loop compare and remove until the last one (nullptr)
    while(currentNode!=0){
        //compareNode start from the next one of currentNode
        compareNode = currentNode->next;
        lastNode = currentNode;
        // compare with each Node after current Node
        while(compareNode!=0){
            // find the same one
            if(compareNode->data == currentNode->data){
                Node* temp = compareNode;
                lastNode->next = compareNode->next;
                compareNode = compareNode->next;
                delete temp;
            }else{
                lastNode = compareNode;
                compareNode = compareNode->next;
            }
        }
        // move currentNode to next one and start next loop
        currentNode = currentNode->next;
    }

}
```

The time complexity for this implementation is $O(n^2)$. This is because we have two pointers and hence have to use two loops to position them and check for duplicates.

Report by : - Siddhant Barua & Jinggui Tan
CWid : - 10439929 & 10427081
Assignment 2 Code Report

Then we tried to change the code and use recursion.

```
void removeDupl_2(Node* head)
{
    // take the ith node and compare with each one, remove the one who has same data.
    Node* currentNode = head;
    Node* lastNode;
    Node* compareNode;

    // no Node or only one Node in the LinkedList, do nothing
    if(head == 0 || head->next == 0){
        return;
    }

    // loop compare and remove until the last one (nullptr)
    if(currentNode!=0){
        //compareNode start from the next one of currentNode
        compareNode = currentNode->next;
        lastNode = currentNode;
        // compare with each Node after current Node
        while(compareNode!=0){
            // find the same one
            if(compareNode->data == currentNode->data){
                Node* temp = compareNode;
                lastNode->next = compareNode->next;
                compareNode = compareNode->next;
                delete temp;
            }else{
                lastNode = compareNode;
                compareNode = compareNode->next;
            }
        }
        // move currentNode to next one and start next loop
        currentNode = currentNode->next;
        removeDupl_2(currentNode);
    }

}
```

Recursion didn't help to reduce the time complexity because removeDupl_2 is still O(n) for the while loop and each call of removeDupl_2 would still call itself (n-1) times.
That's $n+(n-1)+(n-2)+...+ 1 = O(n^2)$.
And the tests also support our assumption.

```
jtan8@avalon:~/code/hw2/updated$ ../origin/prob1
List successfully deleted from memory
Origin Time used :5
jtan8@avalon:~/code/hw2/updated$ ./prob1
List successfully deleted from memory
Recursion Time used :5
```

We tested the algorithm with edge cases (add one element, add no element) and everything works fine.

Report by : - Siddhant Barua & Jinggui Tan
CWid : - 10439929 & 10427081
Assignment 2 Code Report

Queue Implementation using two stacks.

The image below describes our code, we have a enqueue and a dequeue function.
We first Insert data into a stack. We them pop each data from the stack and push it into another stack. For instance if we push data 1,2,3,4,5 into stack 2 then pop them and store in stack 1 we get 5,4,3,2,1.
We use Dequeue to pop out stack 1.
We use Enqueue to pop out current data into another stack and also to push new data onto the stack.

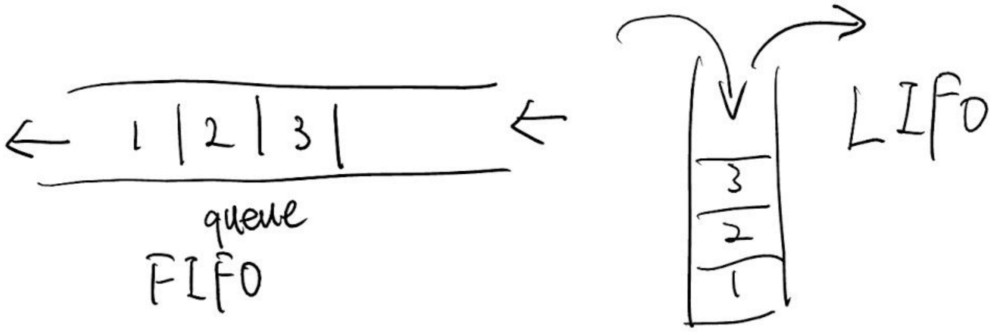
```
class Queue{
private:
    stack<int> stack1, stack2;
public:
    // step1:1st insert data in to stack2 -> 1,2,3,4,5
    // step2:then pop each data and push into stack1 -> 5,4,3,2,1
    // Dequeue just need to pop out stack1.
    // Enqueue need to :
    // step1:pop out current data(stack1) into stack2 and
    // step2:push new data into stack2, then do Enqueue step2 again.

    void EnQueue(int data){
        //make sure stack2 is empty
        while(!stack2.empty()){
            stack2.pop();
        }

        while(!stack1.empty()){
            stack2.push(stack1.top());
            stack1.pop();
        }
        stack2.push(data);

        while(!stack2.empty()){
            stack1.push(stack2.top());
            stack2.pop();
        }
    }

    int DeQueue(){
        // safe check
        if (stack1.size()==0) {
            cout << "ERR:Empty Queue detected!"<< endl;
            return 0;
        }
        int offer = stack1.top();
        stack1.pop();
        return offer;
    }

};
```

Report by : - Siddhant Barua & Jinggui Tan
CWid : - 10439929 & 10427081
Assignment 2 Code Report

The time complexity for the above code is O(3n)=O(n) just three while loops so it’ s O(n) for Enqueue.
For Dequeue it’s O(1).

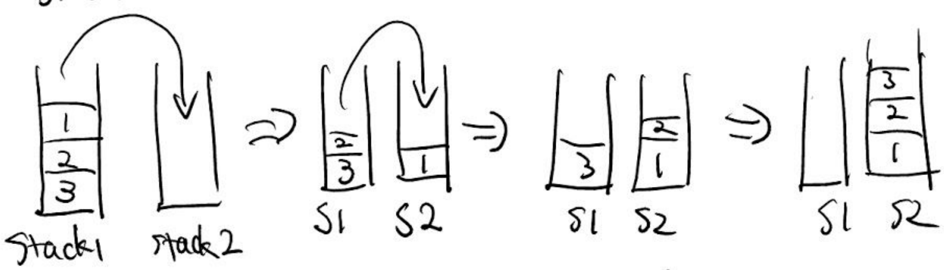
The images in the next page explains how Enqueue and Dequeue functions work.

Report by : - Siddhant Barua & Jinggui Tan
CWid : - 10439929 & 10427081
Assignment 2 Code Report

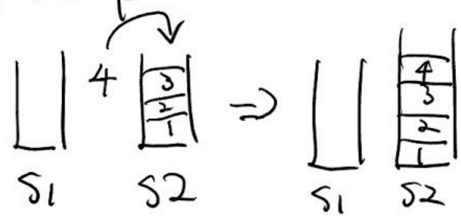


a. EnQueue. Assume we have 1, 2, 3 in the queue.

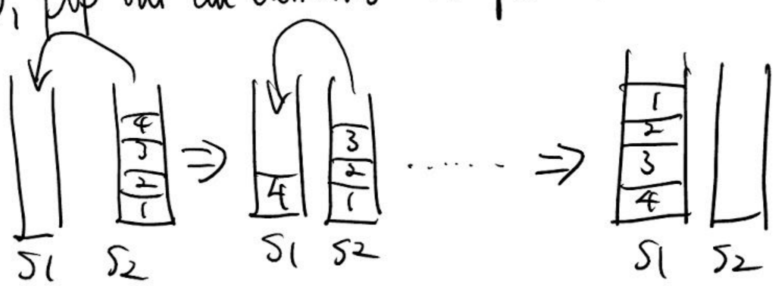
1. make sure stack2 is empty, pop out any existing element from Stack1 to stack 2.



2. then push new element into stack2.



3. pop out all elements and push into stack 1.



then we successfully reverse the stack.

Report by : - Siddhant Barua & Jinggui Tan
CWid : - 10439929 & 10427081
Assignment 2 Code Report

b. DeQueue.

Just pop ~~element~~ from Stack 1.