Esau Hervert
eah170630

Assignment 1

CSE 5343
June 15, 2019

Problem

Write a program to sort a list of numbers in ascending order. The list is a single linked list. You cannot use any library to implement this. In order to sort it, do not swap the values in the nodes - BUT swap the nodes themselves. The linked list must have at least 15 numbers - initially not sorted.

Submit the code. Submit screenshots of the following runs:

1. Travers the linked list before it is sorted.
2. Sort it, and traverse the linked list after the sort.

Code

Code 1: Sorting_Functions_Linked_List.h

```
#ifndef Sorting_Linked_List
#define Sorting_Linked_List

#include <iostream>

using namespace std;

// This is the code that creates the individual nodes.
struct Node {
        int val; // This will store the value of the node.
        Node *next; // This will store a pointer to the next node.
};

// here we will create a class which is the Linked List.
class LinkedList {
public:
        // This will initialize the Linked List to just be the
        // NULL Pointer.
        Node *head = NULL;
        // This function will let us add nodes to the end of the
        // linked list by passing values that the new nodes will have.
        void Add(int);
};

void LinkedList::Add(int v) {
        // Here we create a new node.
        Node *New = NULL;
        New = new Node();

        // Have this node contain the value that we want it to have.
        New->val = v;
```

```
        // Since it is at the end, we have that we want this node
        //to point to NULL.
        New->next = NULL;

        // For the case that this is the first node in the LL.
        if (head == NULL) {
                // The head of the Linked List will be the new node.
                head = New;
        }
        // For the case where there are already nodes in the LL.
        else {
                // We will define a new node so that we can traverse the
                // Linked List to find the final node.
                Node *temp = NULL;
                temp = head;
                while (temp->next != NULL) {
                        temp = temp->next;
                }
                // Once we find the Last node, all we need to do is have
                // this last node point to our new node.

                // Since we have that the new node already points to NULL,
                // we don't haveto do much.
                temp->next = New;
        }
}

Node* prev(Node *h, Node *N) {
        // Base case for if we find the node previous to N.
        if (h->next == N) {
                return h;
        }

        // The following two cases are just in case there is an error in
        // the call to prev.
        // In case we have that the Node is not in the list.
        if (h->next == NULL) {
                return h;
        }
        // In case the node is the head.
        if (h == N) {
                return h;
        }
        // Traverse the linked list until we find the node previous to N.
        prev(h->next, N);
}

Node* SwapNodes(Node *h, Node *N1, Node *N2) {
        /*
```

```
Given the following Linked List:

...->M_(i-1)-> N_1->M_(i+1)->...-> M_(j-1)->N_2->M_(j+1)->...

Where M_i = N_1 and M_j = N_2, i<j.
To swap elements N_1 and N_2, we need to do the following:

        i)              M_(j-1)-> N_1
        ii)             M_(i-1)-> N_2
        iii)    N_1-> M_(j+1) = N_2.next
        iv)             N_2-> M_(i+1) = N_1.next

        Note:   M_(j+1) = N_2.next
                        M_(i+1) = N_1.next

...->M_(i-1)->N_2->M_(i+1)->... ->M_(j-1)->N_1->M_(j+1)->...

To do this we do the following:
        1) Define a temp node to hold the pointer to M_(i+1),
           call this pointer temp.next.
        2) M_(j-1).next = N_1
        3) M_(i-1).next = N_2
                Note: If we have that M_(i-1) is nonexistant,
                i.e. N_1 is the head, then we do not have to
                do this step. The other steps will take care
                of the pointers.
        4) N_1.next = M_(j+1)
        5) N_2.next = temp.next
                Note: temp.next is the value of the original
                      N1.next.

The case where M_(j+1) = N_1 requires a different approach.
*/

if (prev(h, N2) != N1) {
        // This temp node will hold an intermediate value which
        // will be used to swap the nodes.
        Node *temp = NULL;
        temp = new Node();
        temp->next = N1->next;

        // Since we have that N2 is never the head node. we do
        // not have to worry about prev(h,N2) not being undefined.
        prev(h, N2)->next = N1;

        // If N1 is not the head.
        if (N1 != h)
        {
                prev(h, N1)->next = N2;
        }
```

```
                N1->next = N2->next;

                N2->next = temp->next;
        }
        /*
        For the case:

        ->...M_(i-1)->N_1->N_2->M_(j+1)->...

        We will do the following:
                i)              M_(i-1)->N_2
                ii)             N_1->N_2.next
                iii)    N_2->N_1
        */
        else {
                if (N1 != h) // If N1 is not the head.
                {
                        prev(h, N1)->next = N2;
                }

                N1->next = N2->next;

                N2->next = N1;
        }

        if (N1 == h) {
                // For the case were we have that N_2 is now the head.
                return N2;
        }
        return h;
}

// Selection Sorting
// Function takes a Linked List starting with the head and returns the
// head of the sorted Linked List.
Node* SelectSortLL(Node *h) {
        Node *i = NULL; // This will be used to traverse the array.
        Node *j = NULL;
        Node *hh = NULL; // Modifyable Head.
        Node *smallest = NULL;
        i = h; // We will start at the head of the Linked List.

        // Here we will traverse the LL and sort as we go.
        while (i != NULL) {
                // We first set the start of the Linked List to be the
                // node with the smallest value.
                smallest = i;

                j = i->next; // We then start at the next node.
```

```
                // Traverse the rest of the Linked List.
                while (j != NULL) {
                        // If we find a value along the way that is
                        // smaller than before, we make it the
                        // new smallest.
                        if (j->val < smallest->val) {
                                smallest = j;
                        }
                        j = j->next; // Fowards on the Linked List.
                }
                // We have now traversed the rest of the Linked List.
                // We swap the current value with the lowest value and
                // then travel fowards on the Linked List.
                if (smallest != i) {
                        SwapNodes(h, i, smallest);

                        // For the case where we swapped the first Node,
                        // we must redefine the head of the Linked List.
                        if (h == i) {
                                // This is for the case where the smallest
                                // is now the new head.
                                h = smallest;
                        }
                        i = smallest->next; // Fowards on the Linked List.
                }
                // If no swap is needed, we just move fowards.
                else {
                        i = i->next;
                }
        }
        return h;
}

// This structure will be used so that we can return two values for the
// Bubble Sort: the head of the sorted list and number of passes.
struct BubbleValues {
        int Passes;
        Node *head;
};

BubbleValues* BubbleSort(Node *h) {
        // This will be a counter to see if we have to do any swaps
        // in the pass through the list.
        int Swaps = 0;

        // This node will be used to hold the index at current.
        Node *i = NULL;

        // This node will be used to switch with the adjacent node
        // if it turns out to be smaller.
```

```
        Node *j = NULL;

        i = h; // We start at the beggining of the list.

        // We initialize the structure so that we can return both the
        // number of passes and head of the sorted list.
        BubbleValues *result = new BubbleValues();
        result->head = h;


        while (i->next != NULL) // Check to not go out of bounds.
        {
                j = i->next; // Let j be the adjacent node.

                // If j is smaller than i, we swap them.
                if (i->val > j->val) {

                        // SwapNodes returns the head of the LL with
                        // swapped nodes i and j;
                        h = SwapNodes(h, i, j);
                        Swaps++; // This counter keeps track of swaps.
                }
                // If no swap is needed, we just proceed.
                else {
                        i = i->next;
                }
        }

        // This is the Base case, where we have that we traversed the
        // Linked List and we did no swaps, therefore done.
        if (Swaps == 0) {
                result->Passes = 1;
                return result;
        }

        // Otherwise we go ahead and do another pass recursively until
        // we are done.
        // Note the order in which we do this call matters.
        result = BubbleSort(h);
        result->Passes = 1 + result->Passes;
        result->head = result->head;
        return result;
}

// This function will travers the linked list starting at the head.
void traversal(struct Node *head)
{
        if (head == NULL)  // Base case.
        {
                cout << endl;
```

```
                return;
        }
        cout << head->val << " "; // Print value at current node.
        traversal(head->next); // Call the function at the next node.
}
#endif Sorting_Linked_List
```

Code 2: Assignment_1_Sorting_Linked_List.cpp

```
#include "Sorting_Functions_Linked_List.h"
#include <iostream>

using namespace std;

int main() {

        // Linked List 1:
        LinkedList LL1;
        LL1.Add(15);
        LL1.Add(14);
        LL1.Add(13);
        LL1.Add(12);
        LL1.Add(11);
        LL1.Add(10);
        LL1.Add(9);
        LL1.Add(8);
        LL1.Add(7);
        LL1.Add(6);
        LL1.Add(5);
        LL1.Add(4);
        LL1.Add(3);
        LL1.Add(2);
        LL1.Add(1);

        cout << "Linked List 1 is: ";
        traversal(LL1.head);

        cout << "The Select Sorted Linked List 1 is: ";
        LL1.head = SelectSortLL(LL1.head);
        traversal(LL1.head);
        cout << endl << endl;

        // Linked List 2:
        LinkedList LL2;
        LL2.Add(14);
        LL2.Add(4);
        LL2.Add(12);
        LL2.Add(5);
        LL2.Add(7);
        LL2.Add(6);
        LL2.Add(9);
```

```
        LL2.Add(11);
        LL2.Add(1);
        LL2.Add(2);
        LL2.Add(8);
        LL2.Add(3);
        LL2.Add(13);
        LL2.Add(10);
        LL2.Add(15);

        cout << "Linked List 2 is: ";
        traversal(LL2.head);

        // Define a structure that will hold the head of the
        // Bubble Sorted List and the number of passes.
        BubbleValues *result = NULL;
        result = BubbleSort(LL2.head);

        cout << "The Bubble Sorted Linked List 2 with " << result->Passes;
        cout << " Passes is: ";
        traversal(result->head);
}
```

Results

I went ahead and traversed the array and sorted it all in one run. Here is the screenshots of the final run.

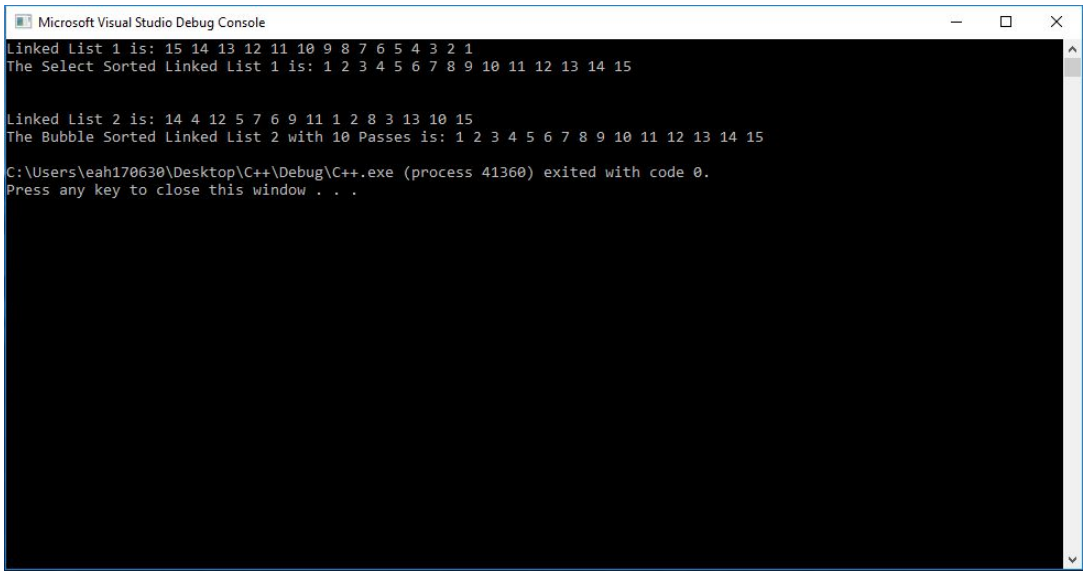

Figure 1: Results