CSCI 255: Lab #3 Linked Lists

Darwin Jacob Groskleg

Tuesday, September 24th, 2019

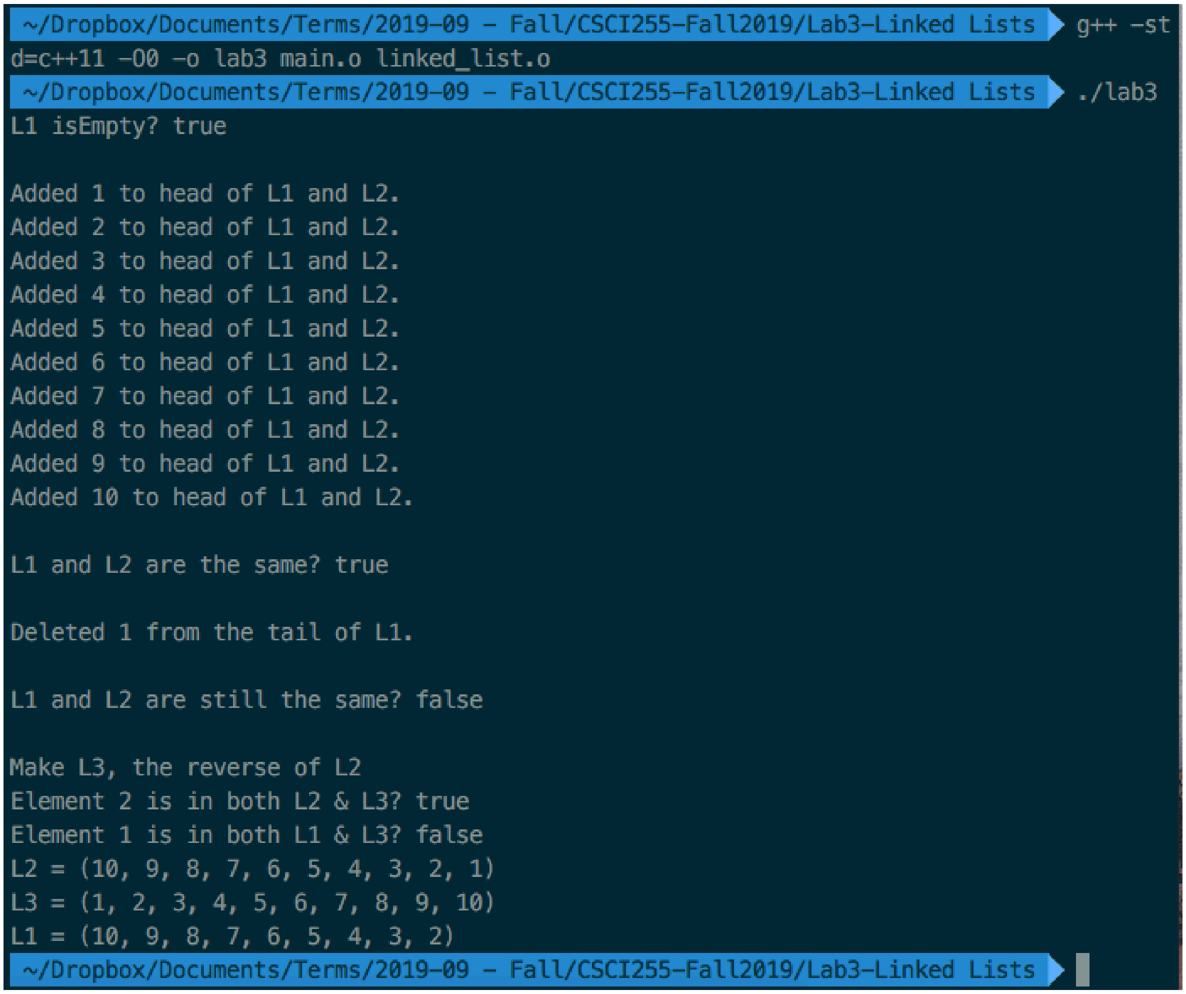
Contents

Questions

Time Complexity

Time complexity is indicated in the definition comments for all methods, see “linked_list.cpp”.

Console Output



```
 ~/Dropbox/Documents/Terms/2019-09 - Fall/CSCI255-Fall2019/Lab3-Linked Lists  g++ -st
d=c++11 -O0 -o lab3 main.o linked_list.o
 ~/Dropbox/Documents/Terms/2019-09 - Fall/CSCI255-Fall2019/Lab3-Linked Lists  ./lab3
L1 isEmpty? true

Added 1 to head of L1 and L2.
Added 2 to head of L1 and L2.
Added 3 to head of L1 and L2.
Added 4 to head of L1 and L2.
Added 5 to head of L1 and L2.
Added 6 to head of L1 and L2.
Added 7 to head of L1 and L2.
Added 8 to head of L1 and L2.
Added 9 to head of L1 and L2.
Added 10 to head of L1 and L2.

L1 and L2 are the same? true

Deleted 1 from the tail of L1.

L1 and L2 are still the same? false

Make L3, the reverse of L2
Element 2 is in both L2 & L3? true
Element 1 is in both L1 & L3? false
L2 = (10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
L3 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
L1 = (10, 9, 8, 7, 6, 5, 4, 3, 2)
 ~/Dropbox/Documents/Terms/2019-09 - Fall/CSCI255-Fall2019/Lab3-Linked Lists 
```

Figure 1: Compile and run.

main.cpp

```
/* main.cpp
 * ---------
 * Authors: Darwin Jacob Groskleg
 * Date:    Tuesday, September 24, 2019
 * CSCI 255
 * Lab 3: Linked Lists
 *
 * QUESTION: What is the Big-O time complexity (in terms of the number of nodes
 *           that have to be traversed) for each of those methods?
 *
 * ANSWER:  time complexity is indicated in the definition comments for all
 *          methods, mostly ”linked_list.cpp”.
 *
 * PROGRAM OUTPUT:
 * -----------------------------------------------------------------------------
 * L1 isEmpty? true
 *
 * Added 1 to head of L1 and L2.
 * Added 2 to head of L1 and L2.
 * Added 3 to head of L1 and L2.
 * Added 4 to head of L1 and L2.
 * Added 5 to head of L1 and L2.
 * Added 6 to head of L1 and L2.
 * Added 7 to head of L1 and L2.
 * Added 8 to head of L1 and L2.
 * Added 9 to head of L1 and L2.
 * Added 10 to head of L1 and L2.
 *
 * L1 and L2 are the same? true
 *
 * Deleted 1 from the tail of L1.
 *
 * L1 and L2 are still the same? false
 *
 * Make L3, the reverse of L2
 * Element 2 is in both L2 & L3? true
 * Element 1 is in both L1 & L3? false
 * L2 = (10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
 * L3 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
 * L1 = (10, 9, 8, 7, 6, 5, 4, 3, 2)
 * -----------------------------------------------------------------------------
 */
#include <iostream>
#include <iomanip>
#include <stack>

#include ”linked_list.hpp”

using namespace std;

void printList(LinkedList &ll, string label);

int main() {
    LinkedList L1, L2;
    cout << ”L1 isEmpty? ” << boolalpha << L1.isEmpty() << ”\n\n”;

    for (int i=1; i<=10; i++) {
        L1.addToHead(i);
        L2.addToHead(i);
        cout << ”Added ” << i << ” to head of L1 and L2.\n”;
    }

    cout << ”\nL1 and L2 are the same? ” << boolalpha << L1.same(L2) << ”\n\n”;

    int n = L1.deleteFromTail();
    cout << ”Deleted ” << n << ” from the tail of L1.\n\n”;

    cout << ”L1 and L2 are still the same? ” << boolalpha << L1.same(L2)
        << ”\n\n”;

    cout << ”Make L3, the reverse of L2\n”;
    LinkedList L3;
    for (int i=10; i>=1; i--) L3.addToHead(i);

    cout << ”Element 2 is in both L2 & L3? ” << L2.isInBothList(2, L3) << '\n';
    cout << ”Element 1 is in both L1 & L3? ” << L1.isInBothList(1, L3) << '\n';

    printList(L2, ”L2”);
    printList(L3, ”L3”);
    printList(L1, ”L1”);

    return 0;
}

void printList(LinkedList &ll, string label=”ll”) {
    cout << label << ” = (”;
    stack<int> st;
    while (!ll.isEmpty())
        st.push(ll.deleteFromTail());
    if (!st.empty()) {
        cout << st.top();
        st.pop();
    }
    while (!st.empty()) { cout << ”, ” << st.top(); st.pop(); }
    cout << ”)\n”;
}
```

linked_list.hpp

```
/* linked_list.hpp
 * ----------------
 * Authors: Darwin Jacob Groskleg, Man Lin
 * Date:    Tuesday, September 24, 2019
 * CSCI 255
 * Lab 3: Linked Lists
 *
 * Purpose: declare the interface for a linked list of integers.
 */
#ifndef LINKED_LIST_HPP_INCLUDED
#define LINKED_LIST_HPP_INCLUDED

class Node {
  public:
    int info;
    Node *next;
    // O(1) - simple assignment
    Node(int el = 0, Node* n = nullptr) {
        info = el;
        next = n;
    }
    ~Node();
};

class LinkedList {
  public:
    // O(1) - simple assignment
    LinkedList() { head = tail = nullptr; }

    ~LinkedList();

    // O(1) - single comparison
    // Returns true if empty.
    bool isEmpty() const { return head == nullptr; }

    // Inserts a new node with the given value to the head of the list.
    void addToHead(int el);

    // Deletes the node at the tail of the list (if any) and returns
    // its value.
    int  deleteFromTail();

    // Determines if a particular value el (argument 1) is in this list and also
    // in another list (argument 2).
    // Returns TRUE if it is in the list, FALSE otherwise.
    bool isInBothList (int el, const LinkedList& another) const;

    // Checks whether the current linked list has the same contents of another.
    //
    // For example, if L1 and L2 are objects of type LinkedList, then
    // L1.same(L2) will return TRUE if L1 has the same contents as L2,
    // and FALSE otherwise.
    bool same(const LinkedList&) const;

  protected:
    Node *head;
    Node *tail;
    bool isInList(int el) const;
};

#endif // LINKED_LIST_HPP_INCLUDED
```

linked_list.cpp

```
/* linked_list.cpp
 * ---------------
 * Authors: Darwin Jacob Groskleg
 * Date:    Tuesday, September 24, 2019
 * CSCI 255
 * Lab 3: Linked Lists
 */
#include "linked_list.hpp"

// Node Destructor
//
//  O(n) - depending on number n nodes are in the tail
//
// NOTE: Will delete the node it points to, and so on.
Node::~Node() {
    if (next != nullptr)
        delete next;
}

// LinkedList Destructor
//
//  O(n) - each node must be deleted first
//
// Deletes all nodes starting from the head. Each node then deletes the next.
LinkedList::~LinkedList() {
    if (head != nullptr)
        delete head;
}

// Method: addToHead
//
//  O(1) - no traversal necessary, only/always touches one end.
//
// Mutates: head, first element in the list.
void LinkedList::addToHead(int el) {
    Node *new_element = new Node(el, head);
    if (isEmpty())
        tail = new_element;
    head = new_element;
}

// Method: deleteFromTail
//
//  O(n) - must travel up the list to reach the 2nd last node to be the tail.
//
// Mutates: tail, last node
// Note:    If empty, will return 0.
int LinkedList::deleteFromTail() {
    int value = 0;
    if (!isEmpty()) {
        value = tail->info;
        if (head == tail) {
            delete tail;
            head = tail = nullptr;
        } else {
            Node *new_tail = head;
            while (new_tail->next != tail)
                new_tail = new_tail->next;
            new_tail->next = nullptr;
            delete tail;
            tail = new_tail;
        }
    }
    return value;
}

// Method: isInList
//
//  O(n) - Must travel down the list, touching at most all n nodes.
//
// Mutates: nothing
bool LinkedList::isInList(int el) const {
    Node *node = head;
    while (node != nullptr) {
        if (node->info == el)
            return true;
        node = node->next;
    }
    return false;
}

// Method: isInBothList
//
//  O(n + m) - Linear, has to travel down both lists from the head to find a
//             match.
//
// Mutates: nothing
bool LinkedList::isInBothList(int el, const LinkedList& another) const {
    return isInList(el) && another.isInList(el);
}

// Method: same
//
//  O(n + m) - Linear, has to travel down both lists from the head, comparing
//             the node values at each point.
//
// Mutates: nothing
bool LinkedList::same(const LinkedList& another) const {
    Node *current = head;
    Node *other   = another.head;

    while (current != nullptr && other != nullptr) {
        if (current->info != other->info)
            return false;
        current = current->next;
        other   = other->next;
    }
    // if different sizes then 1 is not null and they won't have same contents
    if (current != nullptr || other != nullptr)
        return false;
    return true;
}
```