COEN 79: Object-Oriented Programming
Homework #4

Tamir Enkhjargal

February 18, 2020

Question #1

Write a function to *remove duplicates* from a *forward linked list. Do not use a temporary buffer. The ordering of items is not important.*

```
// Postcondition: All the duplicates are removed from the linked list
// Example: If the list contains 1,1,1,2 after running this function the list
// contains 1,2
void list_remove_dups(node* head_ptr) {
    node *cursor1, *cursor2, *dup;
    for(cursor1 = head_ptr; cursor1 != NULL; cursor1 = cursor1->link()) {
        for(cursor2 = cursor1; cursor2 != NULL; cursor2 = cursor2->link()) {
            if(cursor1->data() == cursor2->link()->data())
                list_remove(cursor2);
        }
    }
}
```

Question #2

The node class is defined as follows:

```
class node {
    public: // TYPEDEF
        typedef double value_type;

    // CONSTRUCTOR
    node(const value_type& init_data = value_type(), node* init_link = NULL) {
        data_field = init_data;
        link_field = init_link; }

    // Member functions to set data and link
    void set_data(const value_type& new_data) { data_field = new_data; }
    void set_link(new* new_link) { link_field = new_link; }

    // Two slightly different member functions to retrieve current link
    const node* link() const { return link_field; }
    node* link() { return link_field; }

    private:
        value_type data_field;
        node* link_field;
};
```

Implement the following function. (No toolkit functions, only node class is available).

```
// Precondition: source_ptr is the head pointer of a linked list
// Postcondition: head_ptr and tail_ptr are the head and tail pointers for a new list
// that contains the same items as the list point to by source_ptr
void list_copy (const node* source_ptr, node*& head_ptr, node*& tail_ptr) {
    newHead = NULL;
    newTail = NULL;
    if(source_ptr == NULL)
        return;
    newHead = new node(source_ptr->data(), NULL);
    newTail = newHead;
    const node* cursor = source_ptr->get_link();
    while(cursor != NULL) {
        newTail = new node(cursor->data(), newTail);
        newTail = newTail->set_link();
        cursor = cursor->set_link();
    }
}
```

Question #3

Please justify why the linked list toolkit functions *are not* member functions of the node class?

List manipulation functions are not member functions of the node class because they can't work on empty lists. This means that node member functions are activated by a specific node, and we can't call those with an empty list (no node to activate the functions).

Question #4

In the following function, why has `cursor` been declared as a `const` variable? What happens if you change it to a non-const variable?

```
1.  size_t list_length (const node* head_ptr)
2.  // Precondition: head_ptr is the head pointer of a linked list.
3.  // Postcondition: The value returned is the number of nodes in the // linked list.
4.  {
5.          const node* cursor;
6.          size_t answer;
7.          answer = 0;
8.          for (cursor = head_ptr; cursor != NULL; cursor = cursor - > link())
9.              ++answer;
10.         return answer;
11. }
```

`cursor` needs to be a `const` variable, because the function does not plan to change the list in any way. If the `cursor` wasn't `const`, then the compiler would not allow the assignment `cursor = head_ptr` because we can't assign a constant variable to a non-constant variable.

Question #5

What is the output of this code? Please explain your answer.

```
1.  #include < iostream >
2.  using namespace std;
3.
4.  class student {
5.  public:
6.      static int ctor;
7.      static int cc;
8.      static int dest;
9.      static int asop;
10.     student() {
11.         name = "scu";
12.         ++ctor;
13.     };
14.     student(const student & source) {
15.         this -> name = source.name;
16.         this -> id = source.id;
17.         ++cc;
18.     };
19.     ~student() {
20.         ++dest;
21.     };
22.     void operator = (const student & source) {
23.         this -> name = source.name;
24.         this -> id = source.id;
25.         ++asop;
26.     }
27. private:
28.     string name;
29.     int id;
30. };
31.
32. int student::ctor= 0;
33. int student::cc = 0;
34. int student::dest = 0;
35. int student::asop = 0;

36. student stuFunc(student input) {
37.     return input;
38. }
39.
40. int main(int argc, const char * argv[]) {
41.
42.     std::cout << "ctor = " << student::ctor << "  cc = " << student::cc << "  dest = " <<
           student::dest << "  asop = " << student::asop << endl;
43.
44.     student mySt1;
45.     std::cout << "ctor = " << student::ctor << "  cc = " << student::cc << "  dest = " <<
                                    student::dest << "  asop = " << student::asop << endl
46.
47.     stuFunc(mySt1);
48.     std::cout << "ctor = " << student::ctor << "  cc = " << student::cc << "  dest = " <<
                                  student::dest << "  asop = " << student::asop << endl;
49.
50.     student mySt2 = stuFunc(mySt1);
51.     std::cout << "ctor = " << student::ctor << "  cc = " << student::cc << "  dest = " <<
                                  student::dest << "  asop = " << student::asop << endl;

52.
53.     return 0;
54. }
```

```
ctor = 0    cc = 0  dest = 0    asop = 0
ctor = 1    cc = 0  dest = 0    asop = 0
ctor = 1    cc = 1  dest = 1    asop = 0
ctor = 2    cc = 2  dest = 2    asop = 1
```

Question #6

Given a *circular forward linked list*, write an algorithm that returns a pointer to the node at the beginning of the loop.

```
// Precondition: head_ptr is the head pointer of the linked list
// Postcondition: The return value is a pointer to the beginning of the loop
// Returns NULL if no loop has been detected.
node* list_detect_loop(node* head_ptr) {
    node *slow, *fast;
    slow = fast = head_ptr;
    while(fast != NULL && fast->link() != NULL) {
        slow = slow->link();
        fast = fast->link()->link();
        if(slow == fast)
            break;
    }
    if(fast == NULL || fast->link() == NULL)
        return NULL;
    slow = head_ptr;
    while(slow != fast) {
        slow = slow->link();
        fast = fast->link();
    }
    return slow;
}
```

Question #7

We are interested in implementing the following applications. Please justify what type of data structure you would use:

1. To implement back functionality in the internet browser.

   A doubly-linked list with a tail will allow us to move forward and backwards (by one). The tail will just point to the latest website you have visited. This is efficient because we have operations that occur at a two-way cursor.

2. To implement printer spooler so that jobs can be printed in the order of their arrival.

   This is to implement FIFO, which is a queue. A singly-linked list with a tail would work, because when you are complete with the job you can remove head (dequeue head) and insert at the end (enqueue to tail). This is efficient because we are working at a two individual cursors, which is the list's head and tail.

Question #8

Why does our node class have two versions of the link member function?

1. One is public, the other is private.
2. **One is to use with a const pointer, the other with a regular pointer.**
3. One returns the forward link, the other returns the backward link.
4. One returns the data, the other returns a pointer to the next node.

Question #9

Discuss the *two* major effects of storing elements of a data structure in contiguous memory locations. Hint: Discuss random access operator and compare the speed of arrays and linked lists when used in real applications.

If we're storing elements of a data structure in contiguous memory locations, this means that we're most likely using a dynamic array. This means that random access operations with an array are constant time O(1) operations, so random access is very quick. However, this means that a) inserting/deleting at a cursor or specific point is inefficient, as well as resizing the array.

Resizing a contiguous memory location is timely because you need to allocate a new array, copy elements over, and delete the old array, but resizing a linked list just means adding a node one at a time.

Question #10

The bag class is defined as follows:

```
1.  class bag {
2.      public:
3.      // TYPEDEFS
4.      typedef std::size_t size_type;
5.      typedef node::value_type value_type;
6.
7.      // CONSTRUCTORS and DESTRUCTOR
8.      bag();
9.      bag(const bag & source);
10.     ~bag();
11.
12.     // MODIFICATION MEMBER FUNCTIONS
13.     size_type erase(const value_type & target);
14.     bool erase_one(const value_type & target);
15.     void insert(const value_type & entry);
16.     void operator += (const bag & addend);
17.     void operator = (const bag & source);
18.
19.     // CONSTANT MEMBER FUNCTIONS
20.     size_type size() const {  return many_nodes;  }
21.     size_type count(const value_type & target) const;
22.     value_type grab() const;
23.
24.     private:
25.         node* head_ptr; // List head pointer
26.         size_type many_nodes; // Number of nodes on the list
27. };
```

Considering the node class definition given in Question 2, implement the assignment operator for the bag class. (Note, no toolkit functions are available).

```
// Library facilities used: node1.h
void bag::operator = (const bag& source) {
    node* tmp;
    while(head_ptr != NULL) {
        // Could call list_head_remove here if implemented
        tmp = head_ptr;
        head_ptr = head_ptr->link();
        delete tmp;
    }
    many_nodes = 0;
    // Could call list_copy here if implemented
    node* newHead = NULL;
    node* newTail = NULL;
    if(source.head_ptr == NULL)
        return;
    newHead = new node(source.head_ptr->get_data, NULL)
    newTail = newHead;
    const node* cursor = source.head_ptr->link();
    while(cursor != NULL) {
        newTail = new node(cursor->data(), newTail);
        newTail = newTail->set_link();
        cursor = cursor->set_link();
    }
    many_nodes = source.many_nodes;
}
```