COEN 79

Object-Oriented Programming and Advanced Data Structures

Assignment #4 - Due: November 1st, 2018 – 11:59PM

Name: Jordan Murtiff **Date:** 10-30-18

- Number of questions: 10
- Points per question: 0.4
- Total: 4 points

---

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.*

```
1.  void list_remove_dups(node* head_ptr)
2.  // Postcondition: All the duplicates are removed from the linked list
3.  // Example: If the list contains 1,1,1,2, after running this function the list
4.  // contains 1,2
```

```
void list_remove_dups(node* head_ptr)
{
    node *cursor = head_ptr;
    while(cursor != NULL)
    {
        node* runner = head_ptr;
        while(runner->link() != NULL)
        {
            if(runner->data == runner->link()->data())
            {
                node* tempDelete = runner->link();
                node* tempLink = runner->link()->link();
                runner->set_link(tempLink);
                delete tempDelete;
            }
            else
            {
                runner = runner ->link();
            }
        }
    cursor = cursor->link();
    }
}
```

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 the data and link fields:
    void set_data(const value_type& new_data) { data_field = new_data; }
    void set_link(node* new_link) { link_field = new_link;  }

    // Constant member function to retrieve the current data:
    value_type data() const { return data_field; }

    // Two slightly different member functions to retreive the 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. (Note: *No toolkit function is available*. Only the node class is available.)

```
void list_copy (const node* source_ptr, node*& head_ptr, node*& tail_ptr)
// 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 pointed to by source_ptr
{
```

```
{
    head_ptr = NULL;
    tail_ptr = NULL;
    const node* cursor = source_ptr;
    while(cursor != NULL)
    {
        node* temp = new node (cursor->data(), NULL);
        if(head_ptr == NULL)
        {
            head_ptr = NULL;
            tail_ptr = NULL;
        }
        else
        {
            tail->set_link(temp);
            tail = temp;
        }
        cursor = cursor->link();
    }
}
```

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

Linked list toolkit functions are not member functions of the node class because otherwise if would be impossible to handle an empty linked list with node member functions. Because member functions need an object to activate the function, there would always have to be at least one node object to activate these functions. This would mean that it would be impossible to either insert, delete, or get a node of a linked list without one node already existing. With linked list toolkit functions being non-member functions we are able to modify an empty linked list with no issue and do not need pre-existing node objects.

4. In the following function, why `cursor` has 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 has been declared as a const variable because we only want to read/access each node of a linked list without modifying the contents of each node. A const pointer allows us to move between the different nodes of the linked list without ever changing the values of the nodes themselves. In the case of finding the length of a linked list, all we must do is travel between each node and increase a variable, there is no need to change any of the "data" values of any of the nodes of the linked list.

If we were to change cursor from a constant pointer to a non-const pointer, the code would not compile. In this case we would be increasing the power of the value parameter head_ptr since we would be making it a regular pointer (that can change the data of the nodes of the linked list) rather than a constant pointer.

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. }
```

The values printed out would be:

ctor = 1 cc = 0 dest = 0 asop = 0

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

ctor = 1 cc = 4 dest = 3 asop = 0

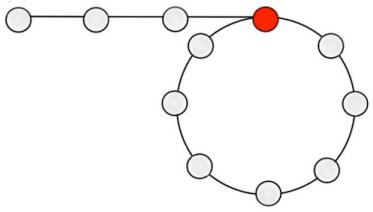
When the main function starts, the constructor, copy constructor, destructor, and assignment operator have not be used and therefore their values start at 0. However, once the first student object mySt1 has been declared, the constructor is called to set the name member variable to "scu" and increments the "ctor" value by 1.

Then, the next statement calls the "stuFunc" function and gives the parameter as the mySt1 object that was given in Line 44. This means that the copy constructor is called twice (once to create the value parameter "input" and another time when the value from the local variable "input" is copied to a temporary location before being returned at the end of the function). Additionally, once the "stuFunc" function ends, both local copies made in the function are then deallocated. This means that the value of "cc" will increase by 2 and the value of "dest" will also increase by 2.

Finally we have the last statement on line 50, using the copy constructor to assign a new student object "mySt2" to the return value of the "stuFunc" function. Since we know the "stuFunc" creates a local variable that is passed as a value paraemter and returns said variable, we know that the copy constructor will be called twice again. However, since we are assigning the return value of the "stuFunc" to "mySt2", the destructor will only be called once since we assign the return value given by the "stuFunc" function. We do not deallocate what we should assign to "mySt2".

So all in all, the normal constructor is called once when we create the student object "mySt1", the copy constructor is called twice each time the"stuFunc" is called (one for the parameter and again for the return statement) and the destructor is called twice when only the "stuFunc" is called alone and only once when the return value of "stuFunc" is assigned to the student object "mySt2".

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



```
1. node* list_detect_loop(node* head_ptr) {
2. // Pre-condition: head_ptr is the head pointer of the linked list
3. // Post-condition: The return value is a pointer to the beginning of the loop
4. // Returns NULL if no loop has been detected.
```

```
node* list_detect_loop(node* head_ptr)
{
    assert(head_ptr != NULL);
    node* slow_runner = head_ptr;
    node* fast_runner = head_ptr;
    while(fast_runner != NULL && fast_runner->link() != NULL)
    {
        slow_runner = slow_runner->link();
        fast_runner = fast_runner->link()->link();
        if(slow_runner == fast_runner)
        {
            break;
        }
    }
    if(fast_runner == NULL || fast_runner->link() == NULL)
    {
        retrun NULL;
    }
    slow_runner = head_ptr;
    while(slow_runner != fast_runner)
    {
        slow_runner = slow_runner->link();
        fast_runner = fast_runner->link();
    }
    return fast_runner;
}
```

7. Suppose `f` is a function with a prototype like this:

```
1. void f(________ head_ptr);
2. // Precondition: head_ptr is a head pointer for a linked list.
3. // Postcondition: The function f has done some manipulation of the linked list,
4. // and the list might now have a new head node.
```

What is the best data type for `head_ptr` in this function?

A. `node`
B. `node&`
C. `node*`
D. `node*&`

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

A. One is public, the other is private.
B. One is to use with a const pointer, the other with a regular pointer.
C. One returns the forward link, the other returns the backward link.
D. One returns the data, the other returns a pointer to the next node.

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.

Storing elements of a data structure in contiguous memory locations (like an array) can allow for two major benefits. One of these benefits is random acess operators. Since arrays are stored in contiguous memory locations, we are able to access and change data at any memory location within the array in O(1) time by just using the [] operator. Although we cannot easily increase the size of the array or shift elements within the array when itmes are added or deleted, we can still acess the data very easily.

The second benefit of storing elements in contiguous memory locations is that we can utilize the cache of the CPU within a computer or server. Becuase the array elemnts are next to each other in memory (within the RAM of the computer/server) then the CPU will most likely move the memory used in the array to the cache and process the data almost 1000x faster. So although linked lists may have adjustable sizes and be good when data is frequently inserted or deleted, this data will most likely not move from the RAM to the cache of a computer/server. Since a linked list is made up of multiple node objects in different parts of memory, it is often very unlikely that all or even a small portion of the linked list will be moved to the cache of a computer/server. So even though we have a data structure that can easily change size, it may still take O(n) or more time to access or change the data of a linked list.

Overall, arrays can be more quickly accessed and changed over a linked list, although it is still difficult to change the size of an array over a linked list. If you had to chose between these two data structures, however, it may be more efficient to chose an array over a linked list when thinking about quickly accessing and changing data within the data structure.

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 function is available.*)

```
1.  void bag::operator = (const bag & source)
2.  // Library facilities used: node1.h
3.  {
```

```
{
    if(&source == this)
    {
        return;
    }
    while(head_ptr != NULL)
    {
        node* temp = head_ptr;
        head_ptr = head->get_link();
        delete temp;
    }
    many_nodes = 0;
    node * temp_head_ptr = NULL;
    node *temp_tail_ptr = NULL;
    const node* cursor = source.head_ptr;
```

Code continued on right side of page ---------->>

```
    while(cursor != NULL)
    {
        node* tempNode = new node (cursor->data(), NULL);
        if(temp_head_ptr == NULL)
        {
            temp_head_ptr = NULL;
            temp_tail_ptr = NULL;
        }
        else
        {
            temp_tail_ptr->set_link(tempNode);
            temp_tail_ptr = tempNode;
        }
        cursor = cursor->link();
    }
    many_nodes = source.size();
}
```