CSCI 140 PA 7 Submission

Due Date: 10/14/2021 Late (date and time):_____

Name(s): Nero Li

Exercise 1 -- need to submit source code and I/O

-- check if completely done ✔ ; otherwise, discuss issues below

Source code below:

```
/*  Program: PA_7_exercise_1
    Author: Nero Li
    Class: CSCI 220
    Date: 10/14/2021
    Description:
        Use a List (C++ NodeList or Java LinkedPositionalList) from the
textbook
        to perform operations on a list of strings.

    I certify that the code below is my own work.

      Exception(s): N/A

*/

#include <iostream>
#include <string>

using namespace std;

typedef string Elem;                                        // list
base element type
class NodeList                                             // node-
based list
{
private:
    struct Node                                            // a node
of the list
    {
        Elem elem;                                         // element
value
        Node* prev;                                        //
previous in list
        Node* next;                                        // next in
list
    };

public:
```

```
    class Iterator                                          // an
iterator for the list
    {
    public:
        Elem& operator*();                                  //
reference to the element
        bool operator==(const Iterator& p) const;           // compare
positions
        bool operator!=(const Iterator& p) const;
        Iterator& operator++();                             // move to
next position
        Iterator& operator--();                             // move to
previous position
        friend class NodeList;                              // give
NodeList access
    private:
        Node* v;                                            // pointer
to the node
        Iterator(Node* u);                                  // create
from node
    };

public:
    NodeList();                                             // default
constructor
    int size() const;                                       // list
size
    bool empty() const;                                     // is the
list empty?
    Iterator begin() const;                                 //
beginning position
    Iterator end() const;                                   // (just
beyond) last position
    void insertFront(const Elem& e);                        // insert
at front
    void insertBack(const Elem& e);                         // insert
at rear
    void insert(const Iterator& p, const Elem& e);          // insert
e before p
    void eraseFront();                                      // remove
first
    void eraseBack();                                       // remove
last
    void erase(const Iterator& p);                          // remove
p

private:                                                    // data
members
    int n;                                                  // number
of items
    Node* header;                                           // head-
of-list sentinel
```

```
    Node* trailer;                                          // tail-
of-list sentinel
};

class NodeSequence : public NodeList {
public:
    void print();
    Iterator atIndex(int i);
    int indexOf(Elem n);
};

NodeList::Iterator::Iterator(Node* u)                       //
constructor from Node*
{ v = u; }

Elem& NodeList::Iterator::operator*()                       //
reference to the element
{ return v->elem; }

bool NodeList::Iterator::operator==(const Iterator& p) const    // compare
positions
{ return v == p.v; }

bool NodeList::Iterator::operator!=(const Iterator& p) const    // compare
positions
{ return v != p.v; }

NodeList::Iterator& NodeList::Iterator::operator++()        // move to
next position
{ v = v->next; return *this; }

NodeList::Iterator& NodeList::Iterator::operator--()        // move to
previous position
{ v = v->prev; return *this; }

NodeList::NodeList()                                        //
constructor
{
    n = 0;                                                  //
initially empty
    header = new Node;                                      // create
sentinels
    trailer = new Node;
    header->next = trailer;                                 // have
them point to each other
    trailer->prev = header;
}

int NodeList::size() const                                  // list
size
{ return n; }
```

```
bool NodeList::empty() const                                    // is the
list empty?
{ return (n == 0); }

NodeList::Iterator NodeList::begin() const                      // begin
position is first item
{ return Iterator(header->next); }

NodeList::Iterator NodeList::end() const                        // end
position is just beyond last
{ return Iterator(trailer); }

void NodeList::insert(const Iterator& p, const Elem& e)
{
    Node* w = p.v;                                              // p's
node
    Node* u = w->prev;                                          // p's
predecessor
    Node* v = new Node;                                         // new
node to insert
    v->elem = e;
    v->next = w; w->prev = v;                                   // link in
v before w
    v->prev = u; u->next = v;                                   // link in
v after u
    n++;
}

void NodeList::insertFront(const Elem& e)                       // insert
at front
{ insert(begin(), e); }

void NodeList::insertBack(const Elem& e)                        // insert
at rear
{ insert(end(), e); }

void NodeList::erase(const Iterator& p)                         // remove
p
{
    Node* v = p.v;                                              // node to
remove
    Node* w = v->next;                                          //
successor
    Node* u = v->prev;                                          //
predecessor
    u->next = w; w->prev = u;                                   // unlink
p
    delete v;                                                   // delete
this node
    n--;                                                        // one
fewer element
}
```

```
void NodeList::eraseFront()                                    // remove
first
{ erase(begin()); }

void NodeList::eraseBack()                                     // remove
last
{ erase(--end()); }

void NodeSequence::print()
{
    Iterator cur{begin()};
    while (cur != end())
    {
        cout << *cur << ' ';
        ++cur;
    }
    cout << endl;
}

NodeSequence::Iterator NodeSequence::atIndex(int i)
{
    Iterator cur{begin()};
    while (i--)
    {
        ++cur;
    }
    return cur;
}

int NodeSequence::indexOf(Elem n)
{
    Iterator cur{begin()};
    int i = 0;
    while (*cur != n)
    {
        ++cur;
        ++i;
    }
    return i;
}

int main()
{
    NodeSequence testSequence;

    testSequence.insertFront("Three");
    testSequence.insertBack("Four");
    testSequence.insertFront("Two");
    testSequence.insertBack("Five");
    testSequence.insertFront("One");
    testSequence.insertBack("Six");
```

```
    testSequence.print();
    cout << *testSequence.atIndex(0) << ' ';
    cout << *testSequence.atIndex(1) << ' ';
    cout << testSequence.indexOf("Three") << endl;
    testSequence.eraseFront();
    testSequence.eraseBack();
    testSequence.print();
    cout << *testSequence.atIndex(0) << ' ';
    cout << *testSequence.atIndex(1) << ' ';
    cout << testSequence.indexOf("Three") << endl;

    cout << "Modified by: Nero Li\n";
    return 0;
}
```

Input/output below:

```
One Two Three Four Five Six
One Two 2
Two Three Four Five
Two Three 1
Modified by: Nero Li
```

Exercise 2 (**with extra credit**) -- need to submit source code and I/O
-- check if completely done ✔; otherwise, discuss issues below
Source code below:

```
/*  Program: PA_7_exercise_2
    Author: Nero Li
    Class: CSCI 220
    Date: 10/14/2021
    Description:
        Set up TextEditor class, which can be used to store, edit, and
display a
        string of characters using a List from exercise 1.

    I certify that the code below is my own work.

      Exception(s): N/A

*/

#include <iostream>
#include <string>

using namespace std;

typedef char Elem;                                          // list
base element type
class NodeList                                              // node-
based list
{
```

```
private:
    struct Node                                             // a node
of the list
    {
        Elem elem;                                          // element
value
        Node* prev;                                         //
previous in list
        Node* next;                                         // next in
list
    };

public:
    class Iterator                                          // an
iterator for the list
    {
    public:
        Elem& operator*();                                  //
reference to the element
        bool operator==(const Iterator& p) const;           // compare
positions
        bool operator!=(const Iterator& p) const;
        Iterator& operator++();                             // move to
next position
        Iterator& operator--();                             // move to
previous position
        friend class NodeList;                              // give
NodeList access
    private:
        Node* v;                                            // pointer
to the node
        Iterator(Node* u);                                  // create
from node
    };

public:
    NodeList();                                             // default
constructor
    int size() const;                                       // list
size
    bool empty() const;                                     // is the
list empty?
    Iterator begin() const;                                 //
beginning position
    Iterator end() const;                                   // (just
beyond) last position
    void insertFront(const Elem& e);                        // insert
at front
    void insertBack(const Elem& e);                         // insert
at rear
    void insert(const Iterator& p, const Elem& e);          // insert
e before p
```

```
    void eraseFront();                                          // remove
first
    void eraseBack();                                           // remove
last
    void erase(const Iterator& p);                              // remove
p

private:                                                        // data
members
    int n;                                                      // number
of items
    Node* header;                                               // head-
of-list sentinel
    Node* trailer;                                              // tail-
of-list sentinel
};

class NodeSequence : public NodeList {
public:
    void print();
    Iterator atIndex(int i);
    int indexOf(Elem n);
    void printWithCursor(int cursor);
};

NodeList::Iterator::Iterator(Node* u)                           //
constructor from Node*
{ v = u; }

Elem& NodeList::Iterator::operator*()                           //
reference to the element
{ return v->elem; }

bool NodeList::Iterator::operator==(const Iterator& p) const    // compare
positions
{ return v == p.v; }

bool NodeList::Iterator::operator!=(const Iterator& p) const    // compare
positions
{ return v != p.v; }

NodeList::Iterator& NodeList::Iterator::operator++()            // move to
next position
{ v = v->next; return *this; }

NodeList::Iterator& NodeList::Iterator::operator--()            // move to
previous position
{ v = v->prev; return *this; }

NodeList::NodeList()                                            //
constructor
{
```

```
    n = 0;                                                  //
initially empty
    header = new Node;                                      // create
sentinels
    trailer = new Node;
    header->next = trailer;                                 // have
them point to each other
    trailer->prev = header;
}

int NodeList::size() const                                  // list
size
{ return n; }

bool NodeList::empty() const                                // is the
list empty?
{ return (n == 0); }

NodeList::Iterator NodeList::begin() const                  // begin
position is first item
{ return Iterator(header->next); }

NodeList::Iterator NodeList::end() const                    // end
position is just beyond last
{ return Iterator(trailer); }

void NodeList::insert(const Iterator& p, const Elem& e)
{
    Node* w = p.v;                                          // p's
node
    Node* u = w->prev;                                      // p's
predecessor
    Node* v = new Node;                                     // new
node to insert
    v->elem = e;
    v->next = w; w->prev = v;                               // link in
v before w
    v->prev = u; u->next = v;                               // link in
v after u
    n++;
}

void NodeList::insertFront(const Elem& e)                   // insert
at front
{ insert(begin(), e); }

void NodeList::insertBack(const Elem& e)                    // insert
at rear
{ insert(end(), e); }

void NodeList::erase(const Iterator& p)                     // remove
p
```

```
{
    Node* v = p.v;                                          // node to
remove
    Node* w = v->next;                                      //
successor
    Node* u = v->prev;                                      //
predecessor
    u->next = w; w->prev = u;                               // unlink
p
    delete v;                                               // delete
this node
    n--;                                                    // one
fewer element
}

void NodeList::eraseFront()                                 // remove
first
{ erase(begin()); }

void NodeList::eraseBack()                                  // remove
last
{ erase(--end()); }

void NodeSequence::print()
{
    Iterator cur{begin()};
    while (cur != end())
    {
        cout << *cur;
        ++cur;
    }
    cout << endl;
}

NodeSequence::Iterator NodeSequence::atIndex(int i)
{
    Iterator cur{begin()};
    while (i--)
    {
        ++cur;
    }
    return cur;
}

int NodeSequence::indexOf(Elem n)
{
    Iterator cur{begin()};
    int i = 0;
    while (*cur != n)
    {
        ++cur;
        ++i;
```

```
    }
    return i;
}

void NodeSequence::printWithCursor(int cursor)
{
    Iterator cur{begin()};
    int i{0};
    while (cur != end())
    {
        if (i == cursor)
        {
            cout << '>';
        }

        cout << *cur;
        ++cur;
        ++i;
    }
    if (i == cursor)
    {
        cout << '>';
    }
}

class TextEditor
{
public:
    TextEditor();
    void left();
    void right();
    void insertCharacter();
    void deleteCharacter();
    void getCurrentPosition();
    void moveToPosition();
    void display();
    void menu();
    int choice();
private:
    NodeSequence text;
    int cursor;
    int length;
};

TextEditor::TextEditor()
{
    string input;
    cout << "Enter a starting string: ";
    getline(cin, input);
    cout << "Editing document . . ." << endl << endl;;
    for (size_t i = 0; i < input.size(); ++i)
    {
```

```
        text.insertBack(input[i]);
    }
    cursor = input.size();
    length = input.size();
}
void TextEditor::left()
{
    if (cursor <= 0)
    {
        cout << "Cursor is at the begin (ignore)." << endl;
    }
    else
    {
        cout << "Moved cursor left." << endl;
        --cursor;
    }
    cout << endl;
}

void TextEditor::right()
{
    if (cursor >= length)
    {
        cout << "Cursor is at the end (ignore)." << endl;
    }
    else
    {
        cout << "Moved cursor right." << endl;
        ++cursor;
    }
    cout << endl;
}

void TextEditor::insertCharacter()
{
    char n;
    cout << "Enter a character: ";
    cin >> n;
    if (cursor == 0)
    {
        text.insertFront(n);
    }
    else if (cursor == length)
    {
        text.insertBack(n);
    }
    else
    {
        text.insert(text.atIndex(cursor), n);
    }
    ++length;
    cout << "Inserted character " << n << ".\n" << endl;
```

```
}

void TextEditor::deleteCharacter()
{
    if (cursor == 0)
    {
        text.eraseFront();
    }
    else if (cursor == length)
    {
        text.eraseBack();
    }
    else
    {
        text.erase(text.atIndex(cursor));
    }
    --length;
    cout << "Deleted one character.\n" << endl;
}

void TextEditor::getCurrentPosition()
{
    cout << "Current position: " << cursor << endl << endl;
}

void TextEditor::moveToPosition()
{
    int n;
    cout << "Enter a position: ";
    cin >> n;
    if (n > length || n < 0)
    {
        cout << "Cannot move to position " << n << " (ignore).\n";
    }
    else
    {
        cout << "Moved to position " << n << ".\n";
        cursor = n;
    }
    cout << endl;
}

void TextEditor::display()
{
    cout << "String: \"";
    text.printWithCursor(cursor);
    cout << '\"' << endl;
    cout << "Length: " << length << endl;
    cout << endl;
}

void TextEditor::menu()
```

```
{
    cout << "   Editing Menu\n" << endl;
    cout << "1. Left" << endl;
    cout << "2. Right" << endl;
    cout << "3. Insert character" << endl;
    cout << "4. Delete character" << endl;
    cout << "5. Get current position" << endl;
    cout << "6. Move to position" << endl;
    cout << "7. Display" << endl;
    cout << "8. Quit" << endl;
    cout << endl;
}

int TextEditor::choice()
{
    int n;
    cout << "Enter an option: ";
    cin >> n;
    if (n == 8)
        cout << "Thanks for using my editor program.\n";

    return n;
}

int main()
{
    TextEditor test;
    int n;
    test.menu();
    n = test.choice();
    while (n != 8)
    {
        switch (n)
        {
        case 1:
            test.left();
            break;
        case 2:
            test.right();
            break;
        case 3:
            test.insertCharacter();
            break;
        case 4:
            test.deleteCharacter();
            break;
        case 5:
            test.getCurrentPosition();
            break;
        case 6:
            test.moveToPosition();
            break;
```

```
        case 7:
            test.display();
            break;
        default:
            break;
        }
        n = test.choice();
    }

    cout << "Modified by: Nero Li\n";
    return 0;
}
```

Input/output below:

```
Enter a starting string: HHello word
Editing document . . .

   Editing Menu

1. Left
2. Right
3. Insert character
4. Delete character
5. Get current position
6. Move to position
7. Display
8. Quit

Enter an option: 7
String: "HHello word>"
Length: 11

Enter an option: 2
Cursor is at the end (ignore).

Enter an option: 1
Moved cursor left.

Enter an option: 3
Enter a character: l
Inserted character l.

Enter an option: 5
Current position: 10

Enter an option: 6
Enter a position: 0
Moved to position 0.

Enter an option: 4
Deleted one character.
```

```
Enter an option: 2
Moved cursor right.

Enter an option: 7
String: "H>ello world"
Length: 11

Enter an option: 8
Thanks for using my editor program.
Modified by: Nero Li
```

Answer for Question 1:

I prefer not to use an array to implement List ADT. First, although we can expand array maximum elements by creating a double-sized array, I still prefer the flexibility from the linked list. If I want to create List ADT with a limitation, I can create another variable to do that limitation. Using an array won't be as flexible as a linked list. Then, for creating an iterator, some of the operations will be easier and more efficient when using doubled linked list. For example, the "++" operator and "--" operator just need to change the current node to the current's next node without calling the element's index and checking the capacity for the current array. Finally, with an iterator, when I write down the function, I can use the linked list in the class as to how I am using the pointer. If I use an array, I can directly call each element in the array, and it will violate the rules for Abstract Data Type.

Answer for Question 2:

For sequence ADT, it is an Abstract Data Type based on list ADT, but for sequence ADT, two more functions allow you to find a variable by index or see a variable's index. There is not an ADT called sequence in the STL library in C++, but since we have an STL list for C++, we can create a class that contains a list variable and then add two functions to make it work as a sequence.