Gianna Julio

CMPT435 - Algorithm Analysis and Design
Assignment 1 - Data Structures

Node Class

```
class Node
{
public:
    char value;
    Node* pointer;

    // default constructor
    Node()
    {
        value = '\0';
        pointer = NULL;
    }

    // overloaded constructor
    Node(char v)
    {
        value = v;
        pointer = NULL;
    }
};
```

The node class contains a value and a pointer attribute, as well as two constructors. The default constructor is not used, but would assign a null char value to the Node value.

SinglyLinkedList Class

```
class SinglyLinkedList
{
public:
    Node* head;
    Node* tail;

    // default constructor
    SinglyLinkedList()
    {
        head = NULL;
        tail = NULL;
    }

    // add front
    void addFront(Node* n)
    {
        // front is empty - first value
        if (head == NULL)
```

```
        {
            head = n;
            tail = n;
        }
        else
        {
            n->pointer = head;
            head = n;
        }
    }

    // add end
    void addEnd(Node* n)
    {
        // if no tail - either both head and tail are empty or just tail is null
        if (tail == NULL)
        {
            if (head == NULL)
            {
                addFront(n);
            }
            // just tail is null
            else
            {
                head->pointer = n;
                tail = n;
                n->pointer = NULL;
            }
        }
        else
        {
            tail->pointer = n;
            n->pointer = NULL;
            tail = n;
        }
    }

    // remove
    void remove(int pos)
    {
        if (head == NULL)
        {
            cout << "Empty list" << endl;
            return;
        }

        // Check if the position is greater than length
        if (length() < pos)
        {
            cout << "Index out of range" << endl;
            return;
        }

        // Declare temp1
```

```
        Node* temp1 = head, * temp2 = NULL;

        // Traverse the list to find the node to be deleted.
        while (pos-- > 1)
        {
            // Update temp2
            temp2 = temp1;

            // Update temp1
            temp1 = temp1->pointer;
        }

        // Change the pointer of previous node
        temp2->pointer = temp1->pointer;

        // Delete the node
        delete temp1;
    }

    // remove front
    void removeFront()
    {
        Node* temp = head;
        // Update head
        head = head->pointer;
        delete temp;
        return;
    }

    // remove end
    void removeEnd()
    {
        Node* temp = head, * prev = NULL;
        while (temp->pointer != NULL)
        {
            prev = temp;
            temp = temp->pointer;
        }
        delete temp;
        prev->pointer = NULL;
        tail = prev;
        return;
    }

    // print
    void print()
    {
        Node* temp = head;

        // if list is empty
        if (head == NULL)
        {
            cout << "Empty list" << endl;
            return;
```

```
127         }
128 
129         while (temp != NULL)
130         {
131             cout << temp->value << endl;
132             temp = temp->pointer;
133         }
134 
135     }
136 
137     // return length for easy access - traverse list with counter
138     int length()
139     {
140         // start at head with length 0
141         Node* temp = head;
142         int len = 0;
143 
144         // if list is empty
145         if (head == NULL)
146         {
147             return 0;
148         }
149 
150         // traverse and increment
151         while (temp != NULL)
152         {
153             len++;
154             temp = temp->pointer;
155         }
156 
157         return len;
158     }
159 };
```

The singly linked list keeps track of a head and tail value. The default constructor initializes both head and tail to null. This class contains the functions to add and remove from front and end (lines 15, 31, 51, 93, 103), as well as length and print functions (lines 138 and 117).

Stack Class

```
1  class Stack: public SinglyLinkedList
2  {
3  public:
4      Node* top;
5  
6      Stack()
7      {
8          top = NULL;
9      }
10 
11     Stack(Node* n)
12     {
13         top = n;
```

```
14      }
15
16      // push - adds node to top of stack
17      void push(Node* n)
18      {
19          addFront(n);
20      }
21
22      // pop - removes node from top of stack
23      Node* pop()
24      {
25          return top;
26          removeFront();
27      }
28  };
```

The stack class extends the singly linked list class, utilizing the add front and remove front functions in its push and pop functions (lines 17 and 23), giving it the first in, last out functionality.

Queue Class

```
1   class Queue: public SinglyLinkedList
2   {
3   public:
4       Node* first;
5
6       Queue()
7       {
8           first = NULL;
9       }
10
11      Queue(Node* n)
12      {
13          first = n;
14      }
15
16      // enqueue - adds node to end of queue
17      void enqueue(Node* n)
18      {
19          addEnd(n);
20      }
21
22      // dequeue - removes node from front of queue
23      Node* dequeue()
24      {
25          return first;
26          removeFront();
27      }
28  };
```

The queue class extends the singly linked list class as well, utilizing the add end and remove front functions in its push and pop functions (lines 17 and 23 again), giving it the first in, first out functionality.

Loading Items From File

```
void loadItems()
{
    // initialize file
    fstream itemFile;
    itemFile.open("magicitems.txt", ios_base::in);

    if (itemFile.is_open())
    {
        // counter
        int i = 0;

        // while file still has items to read
        while (itemFile.good())
        {
            string item; // initialize item string
            getline(itemFile, item); // get line
            items[i] = item; // store item string
            i++; // increment counter
        }
        itemFile.close();
    }
}
```

This function loads all of the items on the magic list into an array of strings. It opens the file, gets each item, saves it to the array, then finally closes the file at the end.

Main Function

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

// initialize array as global variable
string items[666];

int main()
{
    loadItems();

    // for each string item in the list of magic items
    for (string item : items)
    {
        // initialize a Stack and Queue
        Stack s;
        Queue q;

        // for each character in the item name string
        for (char c : item)
        {
            // ignore spaces and punctuation
```

```
            if (isalpha(c))
            {
                // convert to lowercase to ignore capitalization
                char ch = tolower(c);

                // initialize character as a Node
                Node* chNode = new Node(ch);
                s.push(chNode);
                q.enqueue(chNode);
            }
        }

        bool palindrome = true;

        for (char c : item)
        {
            if (s.pop()->value != q.dequeue()->value)
                palindrome = false;
        }

        if (palindrome == true)
        {
            cout << item << endl;
        }
    }

    return 0;
}
```

The main function runs through each character in each item, adding it to both a stack and a queue. Once the lists are full, it goes back through each list, popping and dequeueing and comparing each character. If every character matches, the word is a palindrome and is printed out. If not, the word is not printed.