Gianna Julio

CMPT435 - Algorithm Analysis and Design
Assignment 1 - Data Structures

Assignment1.cpp

Node Class

```cpp
// node class
class Node
{
public:
    char value;
    Node* pointer;
};
```

The Node class is very simple. It contains a char value and a pointer attribute that points to the next Node in the list.

SinglyLinkedList Class

```cpp
// singly linked list
class SinglyLinkedList
{
public:
    Node* head;
    Node* tail;

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

    // add front
    void addFront(char n)
    {
        // initialize character as a Node
        Node* chNode = new Node();
        chNode->value = n;

        // front is empty - first value
        if (head == nullptr)
        {
            head = chNode;
            tail = chNode;
        }
        // otherwise, new node points to old node and replaces head
        else
```

```
        {
            chNode->pointer = head;
            head = chNode;
        }
    }

    // add end
    void addEnd(char n)
    {
        // initialize character as a Node
        Node* chNode = new Node();
        chNode->value = n;
        // if no tail - either both head and tail are empty or just tail is null
        if (tail == nullptr)
        {
            // if the list is empty, add as a head
            if (head == nullptr)
            {
                addFront(n);
            }
            // just tail is null
            else
            {
                head->pointer = chNode;
                tail = chNode;
                chNode->pointer = nullptr;
            }
        }
        // otherwise, tail pointer now points to new node and new node replaces tail
        else
        {
            tail->pointer = chNode;
            chNode->pointer = nullptr;
            tail = chNode;
        }
    }

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

    // remove end
    Node* removeEnd()
    {
        // initialize temp and previous nodes
        Node* temp = head, * prev = nullptr;
        // while temp is not the tail
```

```
84          while (temp->pointer != nullptr)
85          {
86              // save prev value
87              prev = temp;
88              // iterate through nodes
89              temp = temp->pointer;
90          }
91          // reallocate memory
92          delete temp;
93          // set prev pointer to null and replace tail with prev
94          prev->pointer = nullptr;
95          tail = prev;
96          return tail;
97      }
98
99      // print
100     void print()
101     {
102         Node* temp = head;
103
104         // if list is empty
105         if (head == nullptr)
106         {
107             cout << "Empty list" << endl;
108             return;
109         }
110
111         // iterate through nodes and print on same line, separated by spaces
112         while (temp != nullptr)
113         {
114             cout << temp->value << " ";
115             temp = temp->pointer;
116         }
117         // new line
118         cout << endl;
119
120     }
121
122     // return length for easy access - traverse list with counter
123     int length()
124     {
125         // start at head with length 0
126         Node* temp = head;
127         int len = 0;
128
129         // traverse and increment
130         while (temp != nullptr)
131         {
132             len++;
133             temp = temp->pointer;
134         }
135
136         return len;
137     }
```

```
138 };
```

The Singly Linked List class keeps track of a head and tail Node. The default constructor (line 9) initializes both the head and tail to a null pointer value. This class contains the functions to add and remove from front and end (lines 16, 38, 69, 80), as well as print (line 101) and length functions (line 124).

The addFront method creates a new Node and either places it at the start of the list or replaces the head Node, altering its pointer to point to the old head Node.

The addEnd method also creates a new Node, and either utilizes the addFront function to place it as the first Node, places it as the tail if the tail is null, or replaces the tail and makes the previous Node's pointer point to the new Node.

The removeFront method stores the pointer to the head node in a temporary Node, reassigns the head the its own pointer, or the next Node in the list, and deletes the value pointed to by the temporary Node, the old head. The memory is open to be reallocated and the new head is returned.

The removeEnd method iterates through each Node in the list, storing a temporary pointer to the current, as well as storing a pointer to the previous Node. When the tail is reached, the temporary pointer pointing to the last value is deleted from memory, and the tail is reassigned to point to the stored previous value, which is then returned.

The print function simply iterates through the Nodes until it reaches a null pointer at the tail, printing each value on the same line separated by a space to the console. If the list is empty, detected by a null head, 'Empty list' is instead printed to console.

The length function iterates through each Node in the list, incrementing a counter variable each time, and returning the total length at the end.

Stack Class

```
1  // stack - first in, last out
2  class Stack: public SinglyLinkedList
3  {
4  public:
5      // push - adds node to top of stack
6      void push(char n)
7      {
8          addFront(n);
9      }
10
11     // pop - removes node from top of stack
12     Node* pop()
13     {
14         return removeFront();
15     }
16 };
```

The Stack class extends the Singly Linked List class, utilizing the addFront and removeFront functions in its push and pop functions (lines 6 and 12), giving it first in, last out functionality.

Queue Class

```
1  // queue - first in, first out
2  class Queue: public SinglyLinkedList
3  {
4  public:
5      // enqueue - adds node to end of queue
6      void enqueue(char n)
7      {
8          addEnd(n);
9      }
10
11     // dequeue - removes node from front of queue
12     Node* dequeue()
13     {
14         return removeFront();
15     }
16 };
```

The Queue class extends the Singly Linked List class as well, utilizing the addEnd and removeFront functions in its enqueue and dequeue functions (lines 6 and 12 again), giving it first in, first out functionality.

Loading Items From File

```
1  // load magic items into array from file
2  void loadItems()
3  {
4      // initialize file
5      fstream itemFile;
6      itemFile.open("magicitems.txt", ios_base::in);
7
8      if (itemFile.is_open())
9      {
10         // counter
11         int i = 0;
12
13         // while file still has items to read
14         while (itemFile.good())
15         {
16             string item; // initialize item string
17             getline(itemFile, item); // get line
18             items[i] = item; // store item string
19             i++; // increment counter
20         }
21         itemFile.close();
22     }
23 }
```

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 global array, then closes the file.

Main Function

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

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

// MAIN FUNCTION
// ------------------------------------------------------------------------------
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);

                s.push(ch);
                q.enqueue(ch);
            }
        }

        // assume each item is a palindrome
        bool palindrome = true;

        // while stack (and queue - same length) is not empty
        while (s.length() > 1)
        {
            // grab each head value
            char stackChar = s.pop()->value;
            char queueChar = q.dequeue()->value;

            // compare - if there are any that dont match the word is not a palindrome
            if (stackChar != queueChar)
                palindrome = false;
        }

        // if the bool is still true after the loop, all letters matched
        if (palindrome == true)
        {
```

```
53              // print out the palindrome item
54              cout << item << endl;
55          }
56      }
57      return 0;
58  }
```

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, 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.