CMPT 435 - Spring 2019 - Dr. Labouseur

Assignment One – Palindrome Checker
Checking for Palindromes With Stacks and Queues

Sam Alcosser
Samuel.Alcosser1@Marist.edu

September 16, 2020

Contents

1 Introduction

1.1 What is this project?

The task of this project was to use stacks and queues, which we wrote ourselves, to be able to test whether or not each item of a 666 line text document is a palindrome.

1.2 About This Document

The use of this extensive documentation is meant to be a cleaner way of explaining how the assignment function works. Because of this, the only code comments in the program itself briefly explain the functions of each class within their respective header files. Other than these, there are no actual comments.

2 Logic and Functionality

Stacks, Queues, and Palindromes

To test the palindromes, the basic structure of the logic is to put each character, excluding spaces, into a stack and into a separate queue at the same time. The opposing last in first out (LIFO) nature of stacks and first in first out (FIFO) system of queues allow for this process to work. Because of the ordering of the stack and the queue, the characters of the line can be read from the outside in.

2.1 Analysis of Class and Header Files

2.1.1 `Node.h` :The Node Class

```
1  #pragma once
2  #include <string>
3  #include <iostream>
4  class Node{        // defining class for the node that has the pointer and the data
5  public:
6          std::string data;        // stores data for the node
7          Node* next;                        // stores pointer for the node
8          Node() {};                        // constructor for Node class
9  };
```

The node class is a very simple object. So simple in fact, that there is no `.cpp` class file to go along with it. In reality, this class could have been turned into a `struct`, although to be honest the former option was easier to implement. After some assistance from Dr. Labouseur, I was able to grasp how to correctly deal with pointers to objects. The contents of a `Node` object, as defined by me, are `std::string data`, `Node* next`, and the constructor `Node(){}`.

The first one, `std::string data` , is simply the attribute that holds the data of the node in a string. `Node* next` is a pointer to another Node. The ability to have an attribute of this type is somewhat new to me, although I do see

the use of it. With it, and with a general understanding of how it works, it is possible to have code that more accurately represents what the compiler will be doing. Something like this would never even be a thought when using another language such as Java. Lastly, I learned to use the `#pragma once` preprocessor directive to avoid using header guards on all classes. This is seen throughout the rest of the classes as well.

2.1.2 `Stack.h` :The Stack Class

```
#pragma once
#include <iostream>
#include <string>
#include "Node.h"
class Stack
{
public:
        Node *top;                                          //points to the top node of the stack
        bool isEmpty();                                    //tests whether or not the stack is empty
        void push(std::string arg); //function to add a node to the top of the stack
        Node pop();                                          //function to take something off of the stack
};
```

Above is the header file for the `Stack` class that I made. Due to the extensive comments, I will now move on to reviewing its `.cpp` companion file with the definitions of the methods. Starting with the `isEmpty()` function.

```
bool Stack::isEmpty()
{
    if (top == nullptr)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

This is a simple method to check whether or not the stack is empty or not. All that the function does is check whether the `top` pointer of the stack is pointing to anything, and returns the answer as a boolean. This method is actually used inside of another one of `Stacks`' methods called `pop()`.

```
Node Stack::pop()
{
    if (isEmpty())
    {
        throw "Memory underflow Error";
    }
    else
    {
        Node retVal = Node(*top);
```

```
10          Node *newTop = top->next;
11          top = newTop;
12          return retVal;
13      }
14  }
```

This method is used to take an item off of the top of the stack. To account for the possibility of an attempt to pop off an empty stack, the `isEmpty()` method is used. As long as this function returns `false`, the function proceeds to take off of the top of the stack. It does so by copying the top `node` and saving it to a local variable (line 9), setting the `top` to the address of the top `nodes`' next pointer (lines 10,11), and then finally returning the local variable. The final method of this class, `push(std::string arg)` is shown below.

```
1   void Stack::push(std::string arg)
2   {
3
4       Node *n = new Node();
5       n->data = arg;
6       Node *oldTop = top;
7       n->next = oldTop;
8       top = n;
9
10      return;
11  }
```

This method is used to add new items to the top of the stack. The process for doing this starts by making a new node named `n` and giving the string argument to its `data` field (lines 4,5).After the node is made, a copy of the top pointer from the stack is created (line 6). Finally, the newly created node is then given the copy of the old top pointer and the stack's top pointer is now applied to the new node. (line 7, 8).

2.1.3 `Queue.h` :The Queue Class

```
1   #pragma once
2   #include "Node.h"
3   #include <string>
4   class Queue
5   { // header class to define the queue class
6   public:
7           Node *head;                                         //pointer to the head node
8           Node *tail;                                         //pointer to the tail node
9           void enQueue(std::string s); //function which adds a node to the queue
10          Node deQueue();                                   //function to take a node off  of the queue
11          bool isEmpty();                                   //tests if the queue is empty
12  };
```

Here is the header file for the `Queue` class. Generally, it should look very similar to the header file for the `stack` class. The only differences being `pop()` and `push(std::string arg)` are swapped for `enQueue(std::string s)` and `deQueue()`. Other than those changes, you will also see that there are now pointers for the `head` and the `tail`. The `head` is most similar to the `top` pointer of the `stack`. Although, with the addition of the `tail`, processes are much quicker.

```
1  bool Queue::isEmpty()
2  {
3          if (tail == nullptr)
4          {
5                  return true;
6          }
7          else
8          {
9                  return false;
10         }
11 }
```

This simple method `isEmpty()` is used to check if the `Queue` is empty. The `Queue` does this simply by checking if the tail is a null pointer and returning the answer as a boolean.

```
1  void Queue::enQueue(std::string s)
2  {
3
4          if (isEmpty())
5          {
6
7                  Node *n = new Node();
8                  n->data = s;
9                  tail = n;
10                 head = n;
11         }
12         else
13         {
14
15                 Node *t = new Node();
16                 t->data = s;
17                 tail->next = t;
18                 tail = t;
19         }
20         return;
21 }
```

For the `enQueue()` method, there are two possibilities of what will happen. Both options start by creating a new `node` and giving it the data that was passed to the function as a string (lines 7-8, 15-16). Now, what happens next depends on whether or not the `queue` was tested as empty or not. If the `queue` was empty, then the newly created `node` is pointed at by the `head` and `tail` pointers (lines 9-10).

Otherwise, The current `tail` pointer's next pointer is directed at the new `node`, and the tail pointer is pointed at the new node (lines 17-18). An analogy for this would be that the last person in the line would first point at the new person, putting them in line, before assigning the title of last in line to the new person. The last method of the `Queue` class is the `deQueue()` method.

```
1  Node Queue::deQueue()
2  {
3          if (isEmpty())
4          {
5                  throw "Memory Underflow Error";
6          }
7          else
8          {
9                  Node n = Node(*head);
10                 if (head->next != nullptr)
11                 {
12                         Node *newHead = head->next;
13                         head = newHead;
14                         return n;
15                 }
16                 else
17                 {
18                         head = nullptr;
19                         tail = nullptr;
20                         return n;
21                 }
22         }
23 }
```

The `deQueue()` method is the most complicated out of all previously reviewed. To begin, the method tests to ensure that the `queue` is not empty, throwing an error if this is true. (lines 3-6). Then, after making a copy of the current head called `n` which will later be returned (line 9), the function checks if there is a `node` after the `head` (line 10). The reason for this check is that the head pointer must be updated if there are still more elements. This procedure starts by making a new pointer called `*newHead`, as this will point at the next element after the current head (line 12). After this, `newHead` is assigned as the memory address / pointer for `head`, and the current head copy we called `n` is returned (lines 13-14). This proecess is much simpler if it is determined that the current head is the only item in the `queue`. If this is true, the `head` and `tail` are reset to null pointers, and the copy of the head is returned (lines 18-20).

2.2 Using Stacks and Queues To Create The Procedure

A Quick Note

Within this section, the logic of how the palindromes are found is discussed. The source code for this process is found on the `Main.cpp` file. Within this file, you can also find other functions such as `linkedListDemo()`, `StackDemo()`,

`QueueDemo()`, and `printList()`. Although these functions were useful to display the capabilities of the `stack` and the `queue`, they serve no function to help find palindromes and therefore are outside of the scope of this document. Instead, we will focus on the function called `AlanParse()`, aptly named as my professor, Dr. Alan labouseur, introduced the algorithm.

Organization

Below is the function in its' entirety. Due to the fact that this function is relatively complex, The logic will be broken up and discussed by level of scope.

```
1  void AlanParse()
2  {
3          fstream newfile;
4          string arr[1000];
5          string ln;
6          int count = 0;
7          int pCount = 0;
8          newfile.open("magicitems.txt", ios::in);
9          if (newfile.is_open())
10         {
11                 int cnt = 0;
12                 while (getline(newfile, ln))
13                         arr[cnt++] = ln;
14                 int placeHolder = 0;
15                 while (arr[placeHolder] != "")
16                 {
17
18                         //read data from file object and put it into string.
19                         string ogLine = arr[placeHolder];
20                         string line;
21                         Stack stack = Stack();
22                         Queue queue = Queue();
23                         int len = 0;
24                         for (char c : arr[placeHolder])
25                         {
26                                 if (c == ' ')
27                                 {
28                                         continue;
29                                 }
30                                 else
31                                 {
32                                         len++;
33                                         string temp;
34                                         temp += toupper(c);
35                                         stack.push(temp);
36                                         queue.enQueue(temp);
37                                 }
38                         }
39                         for (int i = 0; i < len; i++)
```

```
40                      {
41                              string st = stack.pop().data;
42                              string qu = queue.deQueue().data;
43
44                              if (st == qu)
45                              {
46                                      if (i == (len - 1))
47                                      {
48                                              pCount++;
49                                              std::cout << ++count << ": " << ogLine << ": is a palindrome." << endl;
50                                      }
51
52                                      continue;
53                              }
54                              else
55                              {
56                                      ++count;
57                                      //cout << ++count << ":" << endl;
58                                      //cout << st << " and " << qu << " dont match in " << line << ", so" << endl;
59                                      //        cout << ogLine << ": is not a palindrome." << endl;
60                                      break;
61                              }
62                      }
63                      placeHolder++;
64              }
65      }
66
67      //checking whether the file is open
68
69      //get size of file
70      // add to array
71
72      std::cout << "In Total, there were " << pCount << " palindromes." << endl;
73
74      return;
75  }
```

2.2.1 Initialization

```
1  void AlanParse()
2  {
3          fstream newfile;
4          string arr[1000];
5          string ln;
6          int count = 0;
7          int pCount = 0;
8          newfile.open("magicitems.txt", ios::in);
9          if (newfile.is_open())
10         {
11         ...
12         ...
13         ...
14     }
15         std::cout << "In Total, there were " << pCount << " palindromes." << endl;
16
17         return;
18 }
```

The outermost layer of this process begins with setting everything up for the process. a new `fstream` object is created to handle the source file `magicitems.txt`, and the file is opened (lines 3, 8). Also, four more variables are initialized of varying types on lines 4-7. The array `arr[]` is used to keep all of the lines while string variable `ln` maintains the current line. Lastly, the int variables `count` and `pCount` count the overall number of the line and the count of the palindromes respectively. `pCount` is seen at the bottom being used to display the total number of palindromes once the whole process is over. Now, let's go inside of the curly braces denoting what will take place if the file is open.

2.2.2 Storing Data In Memory

```
1  {
2                  int cnt = 0;
3                  while (getline(newfile, ln))
4                          arr[cnt++] = ln;
5                  int placeHolder = 0;
6                  while (arr[placeHolder] != "")
7                  {
8                  ...
9                  ...
10                 ...
11                 }
```

Here the function is simply loading the whole file into memory. This is possible by using a while loop that will add lines from the file one by one as long as there is another line in the file (lines 3,4). Next, another while loop is started. This while loop will iterate continuously until it hits an index in the array that

is empty (lines 5,6). This is possible as I made the array so that the file size of 666 lines would not fully populate the it. Well, onto the next level!

2.2.3 Main Logic

Rereorganization

Here is the innermost layer of this function. For some perspective, at this point we have all of the lines stored in an array, and we are currently focusing on the line indicated by the `placeHolder` index counter. Abstractly speaking, here is where all of the actual processing is completed. The logic will be broken up into two sub sections: Pushing and enQueueing, and Comparing.

2.2.4 Pushing and enQueueing

```
1  //read data from file object and put it into string.
2  string ogLine = arr[placeHolder];
3  string line;
4  Stack stack = Stack();
5  Queue queue = Queue();
6  int len = 0;
7  for (char c : arr[placeHolder])
8  {
9      if (c == ' ')
10     {
11         continue;
12     }
13     else
14     {
15         len++;
16         string temp;
17         temp += toupper(c);
18         stack.push(temp);
19         queue.enQueue(temp);
20     }
21 }
22
23
```

To begin, the current line indicated as `arr[placeHolder]` is now assigned to the variable called `ogLine` (line 2), and a `Stack` and a `Queue` are initialized (lines 4,5). Next, Each character of the line that is not a space (lines 9-12) is added to a stack and a queue (lines 18,19). Also, a counter called `len` is used to keep track of the length of the list. If lines 16 and 17 are confusing, do not worry. These lines are only needed due to complications within the C++ language when it comes to modifying `strings` and `chars`.

2.2.5 Comparing

```
1  for (int i = 0; i < len; i++)
2  {
3      string st = stack.pop().data;
4      string qu = queue.deQueue().data;
5
6      if (st == qu)
7      {
8          if (i == (len - 1))
9          {
10             pCount++;
11             std::cout << ++count << ": " << ogLine << ": is a palindrome." << endl;
12         }
13
14         continue;
15     }
16     else
17     {
18         ++count;
19         //cout << ++count << ":" << endl;
20         //cout << st << " and " << qu << " dont match in " << line << ", so" << endl;
21         //        cout << ogLine << ": is not a palindrome." << endl;
22         break;
23     }
24 }
25 placeHolder++;
```

Finally The chars are `pop`ped and `dequeue`d off of their respective lists to be compared. the for loop at the top is used to iterate through the full length of both lists. In this case, the current item off of the stack is called `st`, and the item off of the queue is called `qu` (lines 3,4). Here is where the genius comes in. Because of how stacks and queues are built, they will effectivley be comparing each element of the list from the outside in. This is the exact process needed to check for palindromes. Because of this mirrored comparison, one would not want to waste compute cycles comparing chars once you cross the midpoint of the palindrome. This is the reason for the condition `if (i == (len-1))`.

With this, we can stop the checking once the for loop reaches the middle of the line with no faulty comparisons. At this point, the `pCount` variable used to keep track of the number of palindromes and the overall count variable are incremented. After this, there is a console print statement (lines 10,11). If it has not yet hit the midpoint, but the comparison was the same, the loop will `continue` onto the next letter. If not, and the characters don't match, the overall count variable is incremented, logs are printed, and the program `break`s out of the loop (lines 16 -22). Finally, after all of this, the `placeHolder` variable is incremented, and the next line is tested.

3 Closing Thoughts

Leave it to computer scientists to devise incredibly complex theorems and solutions to problems that we learned to solve in fifth grade. Either way, the process of experimenting with creating extensive documentation using LaTeX was quite fun overall. I can definitely see myself using it in the future. Also, I would like to see a 10 year old review 666 lines to check for palindromes in just under 250 milliseconds.

$\{S.A.\}$