Assignment One

Josh Seligman

joshua.seligman1@marist.edu

September 9, 2022

1 Singly Linked List

1.1 The Data Structure

A singly linked list is comprised of nodes which contain some form of data as well as a pointer to the next element within the list. As shown in Figure 1.1, the final node has a next of **null**, which marks the end of the list. In order to access a particular element, one has to start at the beginning and traverse through the list until the desired node is found. This causes data access to be on the magnitude of $O(n)$ as the time required to find an element has a linear relationship with the size of the list.
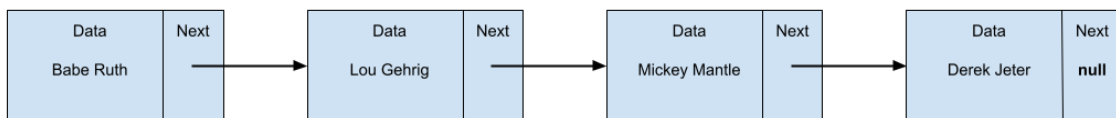

Figure 1.1: Example singly linked list of 4 Yankees legends.

1.2 Benefits of a Singly Linked List

1.2.1 Size

As previously mentioned in Section 1.1, the last node within a linked list has a next of **null**. This characteristic enables linked lists to have no size restrictions barring memory capacity. As a result, this feature makes linked lists preferred over arrays, which have a fixed length, when the size of the data is frequently changing and has an unkown maximum. For instance, as demonstrated in Section 5.4 starting on line 10 within *main.cpp*, the size of the linked list is only limited by our needs and, if needed, more nodes are able to easily be added to the list with their creation as done on lines 13-15 and linking as shown on lines 18 and 19. On the other hand, if the list was made with an array, the size of the array would have to be provided at the time of the creation of the array, and it would not be easy to change the size if additional data have to be added to the array.

1.2.2 DATA TYPE FLEXIBILITY

Linked lists do not have to be restricted to be able to store a specific data type. Instead, with the use of generics (C++ templates), the definition of a node is independent of the data type that the user wants to store within the linked list. This provides flexibilty and reusability for many use cases. As demonstrated in Section 5.1 in *node.h*, the definition of a node uses a generic T as the type of data being stored, which prevents any assumptions of the data and ensures compatibility with all data types. However, due to how the C++ linker works and to prevent all the code from being written within a single header file, the allowed types have to be stated on lines 13 and 14 of *node.cpp*. This is a C++ specific issue and is not present in other languages such as Java. Regardless, although they have to be specified for C++, any data type can still be stored within a node and a linked list. A demonstration of the user defining which data type is stored in a node is in Section 5.4 on lines 13-15 within *main.cpp*. Instead of the Node class defining the data type, the user is able to specify the type of data they want to store, which is a string in this situation but can be anything they want.

2 STACK

2.1 THE DATA STRUCTURE

A stack uses a last in, first out (LIFO) approach to storing data. The most common analogy for stacks is a stack of plates. Each plate is placed on top of each other to build the stack when being stored, but the plate on top is always the first to be used. In other words, the most recently plate that was put away is also the first plate that is taken out. As displayed in Figure 2.1, the stack has a variable called top, which points to the first item in the stack. Additionally, as shown by the arrows between each element, linked lists are used in the implementation of stacks. Stacks have 3 primary functions: push, pop, and isEmpty. Push adds a new element to the top of the stack, pop removes the top item from the stack and returns the data, and isEmpty returns whether or not the stack is empty.
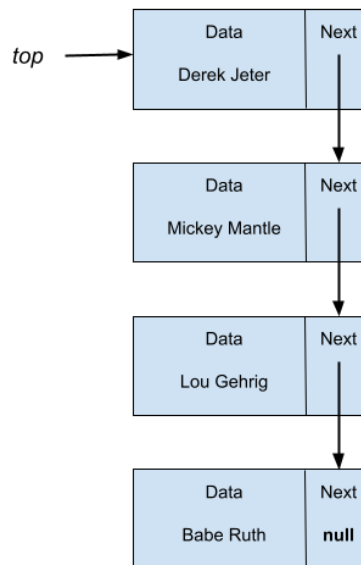

Figure 2.1: Example stack of 4 Yankees legends.

2.2 Benefits of a Stack

2.2.1 Time of Individual Data Access

Stacks are incredibly fast when it comes to the implementation of the push and pop methods. As demonstrated in *stack.cpp* in Section 5.2, the push method on lines 21-28 creates the new node for the stack (line 24), points its next element to the current top of the stack (line 26), and then update the top of the stack to point to the newly created node (line 28). These 3 steps will always run and do not depend on the current size of the stack, which means that the push method runs in $O(1)$ time. The pop method on lines 30-46 is very similar in that it also executes the same few steps regardless of the current size of the stack. These steps are check if the stack is empty for safety (lines 33-36), grab the data from the top of the stack (line 39), update the top pointer to point to the second item in the stack (line 40), and return the data (line 44). There is no need to traverse the list because the push method made the new node the first element of the list, which is equivalent to the top of the stack. Therefore, just like the push method, the pop method also runs in $O(1)$ time.

2.2.2 Compatibility with the Node Class

As mentioned in Section 2.1 and illustrated in Figure 2.1, the stack is implemented using a singly linked list. Therefore, the Stack class in the code has to utilize the Node class written for singly linked lists, which can be found in Section 5.1 and was analyzed in Section 1. In Section 5.2, line 9 in *stack.h* shows that the Stack class has an instance variable that points to the node on the top of the stack. Since the Node class supports any data type (see Section 1.2.2), the Stack class has to be able to provide a data type for each node to store. For the same reasons as described in Section 1.2.2, the Stack class also utilizes C++ templates to allow the user to decide which data type gets stored within the stack. The Stack class is then able to take the data type that the user requests, such as **char** on line 32 of *main.cpp* in Section 5.4, and pass it down to the data type that is stored within each node. This concept is demonstrated in Section 5.2 in *stack.cpp* on line 24 when a new node is created and on line 9 of *stack.h* when defining the top variable for the stack.

3 Queue

3.1 The Data Structure

A queue uses a first in, first out (FIFO) approach to handling data. One analogy to understand how a queue works is a line to buy movie tickets. The first person that is in line for these tickets is the first to be assisted at the ticket counter. On the other hand, the last person to enter the line will also be the last one to buy their ticket. Similar to stacks, queues are implemented using a singly linked list. As displayed in Figure 3.1, queues have a head variable that points to the first node within the linked list, which equates to the front of the line for the movie tickets. Queues have 3 primary functions: enqueue, dequeue, and isEmpty. Enqueue adds a new element to the end of the queue, dequeue removes and returns the first element in the queue, and isEmpty checks to see if the queue is empty or not.
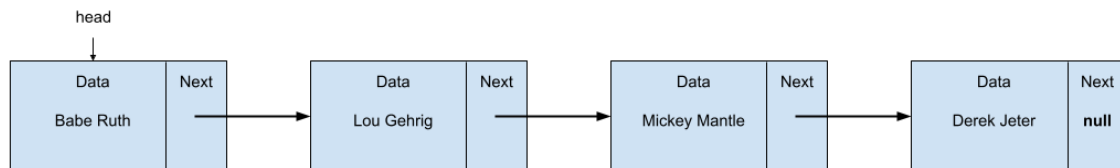

Figure 3.1: Example queue of 4 Yankees legends.

3.2 Benefits of a Queue

3.2.1 Compatibility with the Node Class

The Queue class, just like the Stack class, was implemented using templates to be able to support the use of the Node class, which requires an input of a data type upon creation. As described in Section 1.2.2, C++ templates enable flexibilty and support for many different data types. Additionally, it is also very safe to use because creating a Queue (or a Stack or Node) of a given type will result in a data structure that only accepts that type in the future. For instance, if a Queue is created to support strings, one cannot attempt to enqueue an integer as there will be a type mismatch and the code will not compile. For instance, line 74 of *main.cpp* in Section 5.4 is commented out because the compiler cannot convert a string to a single character, which causes a compilation error as the queue cannot support the string.

3.2.2 Error Safety

In addition the the type safety of the templates as described in Section 3.2.1, the Queue class also has an error measure to prevent users from breaking the program by doing something that is not allowed. In the case of the Queue class, the error safety occurs if the user tries to dequeue from an empty queue. As displayed on lines 43-45 of *queue.cpp* in Section 5.3, the function checks if the queue is empty and throws an error if it is. This is really important as, without the check, there would be a runtime error that crashes the program from attempting to work with a null pointer. In the test program for the queue in *main.cpp* of Section 5.4, lines 67-72 use a try-catch block to make sure the programmed error is thrown when attempting to dequeue from an empty queue. It is of importance to note that this feature is also implemented in the Stack class when attempting to pop from an empty stack (see lines 33-35 of *stack.cpp* in Section 5.2).

4 Main Program

4.1 Program Overview

The objective of the main program was to print out all the palindromes, ignoring whitespace and capitalization, within a file. More specifically, after normalizing a string for whitespace and capitalization, each character of the string should be both pushed onto a stack and enqueued on a queue. The stack will hold the characters in reverse order and the queue will hold each character in the order as they are in the string. Thus, checking to see if the string is a palindrome requires one to just pop the first character from the stack and dequeue the first character in the queue. If there is ever a time where the characters are different, then the string is not a palindrome. On the other hand, if every comparison is successful, then the string is a palindrome.

4.2 Good Parts of the Program

4.2.1 File Operations and Output

4.2.2 Time Complexity of isPalindrome

5 Appendix

5.1 Singly Linked List

node.cpp

```
1  #include <string>
2
3  #include "node.h"
```

```
 4
 5  template <typename T>
 6  Node<T>::Node(T initialData) {
 7      // Initialize the node with the data and without a next node in the linked list
 8      Node::data = initialData;
 9      Node::next = nullptr;
10  }
11
12  // Define acceptable data types that the Node can accept for the template
13  template class Node<std::string>;
14  template class Node<char>;
```

node.h

```
 1  #pragma once
 2
 3  // Node represents an item within a singly linked list and can store data of a given type
 4  template <typename T>
 5  class Node {
 6      public:
 7          // A node has the data it is storing (of a type defined by the user)
 8          // and a pointer to the next node
 9          T data;
10
11          // The pointer uses the template to make sure all elements of the linked list
12          // store the same data type
13          Node<T>* next;
14
15          // Nodes will be instantiated with some data and not have a next node
16          Node(T initialData);
17  };
18
19  // Super helpful resource on templates for c++
20  // https://isocpp.org/wiki/faq/templates#separate-template-fn-defn-from-decl
```

5.2 STACK

stack.cpp

```
 1  #include <string>
 2
 3  #include "stack.h"
 4  #include "node.h"
 5
 6  // Instantiate the stack with the top pointing to nothing
 7  template <typename T>
 8  Stack<T>::Stack() {
 9      top = nullptr;
10  }
11
12  template <typename T>
13  Stack<T>::~Stack() {
14      // Since the nodes were created on the heap, we have to
15      // make sure everything is cleared from memory
16      while (!isEmpty()) {
17          pop();
18      }
19  }
20
21  // Creates a new node and adds it to the stack
22  template <typename T>
23  void Stack<T>::push(T newData) {
```

```cpp
24      Node<T>* newNode = new Node(newData);
25      // Set the next first so we do not lose the rest of the stack
26      newNode->next = top;
27      top = newNode;
28  }
29
30  // Removes the top node from the stack
31  template <typename T>
32  T Stack<T>::pop() {
33      if (isEmpty()) {
34          // Throw an exception if the stack is already empty
35          throw std::invalid_argument("Tried to pop from an empty stack.");
36      } else {
37          // We need to collect the data in the node before removing it from the stack
38          Node<T>* topNode = top;
39          T topData = topNode->data;
40          top = top->next;
41
42          // Since the node was created on the heap, we have to free it from memory
43          delete topNode;
44          return topData;
45      }
46  }
47
48  // Checks to see if the stack is empty or not
49  template <typename T>
50  bool Stack<T>::isEmpty() {
51      return top == nullptr;
52  }
53
54  // Define acceptable data types that the Stack can accept for the template
55  template class Stack<std::string>;
56  template class Stack<char>;
```

stack.h

```cpp
1   #pragma once
2
3   #include "node.h"
4
5   template <typename T>
6   class Stack {
7   private:
8       // Top points to the top of the stack
9       Node<T>* top;
10  public:
11      // We need a constructor and destructor
12      Stack();
13      ~Stack();
14
15      // Push adds a new element to the stack
16      void push(T newData);
17
18      // Pop removes the top element from the stack
19      T pop();
20
21      // isEmpty checks to see if the stack is empty
22      bool isEmpty();
23  };
```

5.3 QUEUE

queue.cpp

```cpp
#include <string>

#include "queue.h"
#include "node.h"

// Instantiate the queue with the head pointing to nothing
template <typename T>
Queue<T>::Queue() {
    head = nullptr;
}

template <typename T>
Queue<T>::~Queue() {
    // Since the nodes were created on the heap, we have to
    // make sure everything is cleared from memory
    while (!isEmpty()) {
        dequeue();
    }
}

// Creates a new node and adds it to the queue
template <typename T>
void Queue<T>::enqueue(T newData) {
    Node<T>* newNode = new Node(newData);

    if (isEmpty()) {
        // Immediately set the head to be the new node if we are empty
        head = newNode;
    } else {
        // Traverse to the back of the queue
        Node<T>* cur = head;
        while (cur->next != nullptr) {
            cur = cur->next;
        }
        // Insert the new node in the back of the queue
        cur->next = newNode;
    }
}

// Removes the front node from the queue
template <typename T>
T Queue<T>::dequeue() {
    if (isEmpty()) {
        // Throw an exception if the queue is already empty
        throw std::invalid_argument("Tried to dequeue from an empty queue.");
    } else {
        // We need to collect the data in the node before removing it from the queue
        Node<T>* frontNode = head;
        T frontData = frontNode->data;
        head = head->next;

        // Since the node was created on the heap, we have to free it from memory
        delete frontNode;
        return frontData;
    }
}

// Checks to see if the queue is empty or not
template <typename T>
bool Queue<T>::isEmpty() {
    return head == nullptr;
}

// Define acceptable data types that the Queue can accept for the template
```

```
65 template class Queue<std::string>;
66 template class Queue<char>;
```

queue.h

```
1  #pragma once
2
3  #include "node.h"
4
5  template <typename T>
6  class Queue {
7  private:
8      // Head points to the front of the queue
9      Node<T>* head;
10 public:
11     // We need a constructor and destructor
12     Queue();
13     ~Queue();
14
15     // Enqueue adds a new element to the queue
16     void enqueue(T newData);
17
18     // Dequeue removes the front element from the queue
19     T dequeue();
20
21     // isEmpty checks to see if the queue is empty
22     bool isEmpty();
23 };
```

5.4 MAIN PROGRAM

main.cpp

```
1  #include <iostream>
2  #include <string>
3
4  #include "node.h"
5  #include "stack.h"
6  #include "queue.h"
7  #include "fileUtil.h"
8  #include "util.h"
9
10 // Function to test the Node class
11 void testNode() {
12     // Create the nodes on the stack, so we do not have to delete later
13     Node<std::string> n1("node 1");
14     Node<std::string> n2("node 2");
15     Node<std::string> n3("node 3");
16
17     // Set up the links
18     n1.next = &n2;
19     n2.next = &n3;
20
21     // Print out the data of each node in the linked list
22     Node<std::string>* cur = &n1;
23     while (cur != nullptr) {
24         std::cout << cur->data << std::endl;
25         cur = cur->next;
26     }
27 }
28
29 // Function to test the Stack class
```

```
30  void testStack() {
31      // Create a stack and add some data to it
32      Stack<char> stack;
33      stack.push('h');
34      stack.push('s');
35      stack.push('o');
36      stack.push('J');
37
38      // Print out the letters as we remove them from the stack
39      while (!stack.isEmpty()) {
40          std::cout << stack.pop();
41      }
42      std::cout << std::endl;
43
44      try {
45          // This should throw an error
46          stack.pop();
47      } catch (const std::invalid_argument& e) {
48          std::cerr << e.what() << std::endl;
49      }
50  }
51
52  // Function to test the Queue class
53  void testQueue() {
54      // Create a queue and add some data to it
55      Queue<char> queue;
56      queue.enqueue('J');
57      queue.enqueue('o');
58      queue.enqueue('s');
59      queue.enqueue('h');
60
61      // Print out the letters as we remove them from the queue
62      while (!queue.isEmpty()) {
63          std::cout << queue.dequeue();
64      }
65      std::cout << std::endl;
66
67      try {
68          // This should throw an error
69          queue.dequeue();
70      } catch (const std::invalid_argument& e) {
71          std::cerr << e.what() << std::endl;
72      }
73
74      // queue.enqueue("Hello");
75  }
76
77  // Function to check if a string is a palindrome, minus whitespace and capitalization
78  bool isPalindrome(std::string word) {
79      // Initialize an empty stack and queue for the checks
80      Stack<char> wordStack;
81      Queue<char> wordQueue;
82
83      // Iterate through each character in the word to populate the stack and queue
84      for (int i = 0; i < word.length(); i++) {
85          char character = word[i];
86          if (character == ' ') {
87              // Go to next character because we are ignoring whitespace
88              continue;
89          } else if (character >= 'a' && character <= 'z') {
90              // Adjust the character to make it uppercase by taking the difference between
91              // the start of the lowercase letters and the start of the uppercase letters
92              character -= 'a' - 'A';
93          }
94          // Add the character to both the stack and the queue
```

```cpp
95              wordStack.push(character);
96              wordQueue.enqueue(character);
97          }
98
99          while (!wordStack.isEmpty() && !wordQueue.isEmpty()) {
100             // Get the character from the top of the stack and queue
101             char charFromStack = wordStack.pop();
102             char charFromQueue = wordQueue.dequeue();
103
104             if (charFromStack != charFromQueue) {
105                 // We can return false because we already know that
106                 // the string is not a palindrome
107                 return false;
108             }
109         }
110
111         // The string is a palindrome
112         return true;
113     }
114
115     int main() {
116         std::cout << "----- Testing Node class -----" << std::endl;
117         testNode();
118         std::cout << std::endl;
119
120         std::cout << "----- Testing Stack class -----" << std::endl;
121         testStack();
122         std::cout << std::endl;
123
124         std::cout << "----- Testing Queue class -----" << std::endl;
125         testQueue();
126         std::cout << std::endl;
127
128         std::cout << "----- Testing isPalindrome -----" << std::endl;
129         std::cout << isPalindrome("racecar") << std::endl; // 1
130         std::cout << isPalindrome("RaCecAr") << std::endl; // 1
131         std::cout << isPalindrome("ra    c e    car") << std::endl; // 1
132         std::cout << isPalindrome("4") << std::endl; // 1
133         std::cout << isPalindrome("") << std::endl; // 1
134         std::cout << isPalindrome("ABC") << std::endl; // 0
135         std::cout << std::endl;
136
137         std::cout << "----- Magic Items -----" << std::endl;
138         try {
139             // Read the file and store it in an array
140             StringArr* data = readFile("magicitems.txt");
141
142             // Only print out the palindromes
143             for (int i = 0; i < data->length; i++) {
144                 if (isPalindrome(data->arr[i])) {
145                     std::cout << data->arr[i] << std::endl;
146                 }
147             }
148
149             // Clean up memory
150             delete data;
151         } catch (const std::invalid_argument& e) {
152             std::cerr << e.what() << std::endl;
153         }
154
155         return 0;
156     }
```

fileUtil.cpp

```cpp
#include <string>
#include <fstream>
#include <iostream>

#include "fileUtil.h"
#include "util.h"

StringArr* readFile(std::string filePath) {
    // File reading code from https://cplusplus.com/doc/tutorial/files/

    // We need to determine how many items there are, so a counter would work well
    int numItems = 0;

    // Create the string buffer for storing each line
    std::string line;
    // Attempt to open the file
    std::ifstream file(filePath);
    if (file.is_open()) {
        // Read each line from the file
        while (getline(file, line)) {
            // During the first read through, we only need to count the number of lines
            // because we do not know how big to make the array
            numItems++;
        }

        // Create the struct to output the data
        StringArr* outArr = new StringArr;
        outArr->arr = new std::string[numItems];
        outArr->length = numItems;

        // We will now reread the file and store the data now that the array has been
        // created with the correct size. Thus, we have to clear any error flags
        // and fix the pointer to go back to the top of the file
        file.clear();
        file.seekg(0);

        // i is a counter to keep track of the index we are on when reading from the file
        int i = 0;
        while (getline(file, line)) {
            // During the second read through, we will store the contents
            // of the line in the array
            outArr->arr[i] = line;
            i++;
        }

        // Close the file when done for safety
        file.close();

        return outArr;
    } else {
        // Throw an error if the file was not found
        throw std::invalid_argument("Unable to open file " + filePath);
    }
}
```

fileUtil.h

```cpp
#pragma once

#include "util.h"

// Reads a file and stores each line into an array
StringArr* readFile(std::string path);
```

util.h

```
1  #pragma once
2
3  // Struct to create string arrays and store their length at the same time
4  // Especially for use on the heap
5  struct StringArr {
6      std::string* arr;
7      int length;
8  };
```