# 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.
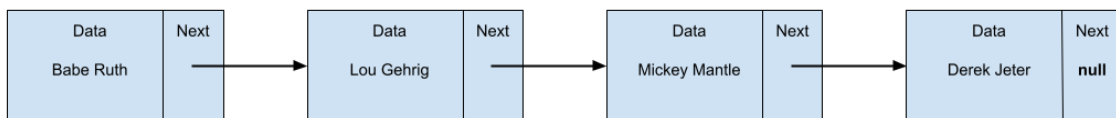


Figure 1.1: Example singly linked list of 4 Yankees legends.

### 1.2 Asymptotic Analysis

Assuming one only has a pointer to the first element in the list, doing any sort of work with the data requires one to start at the beginning and traverse through the list by using the next pointers within each node until the desired operation is complete. For example, as done in Listing 7 within Section 5.4 on lines 22-26, the only way to dynamically obtain the data for each node was to start at the beginning and iterate through each node by taking advantage of the links. Overall, since linked list operations, such as searching, adding, and removing, all function at a rate of a magnitude of the size of the list, their runtime is classified as $O(n)$.

### 1.3 Benefits of a Singly Linked List

#### 1.3.1 Limited Size Restrictions

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.

### 1.3.2 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.

### 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.
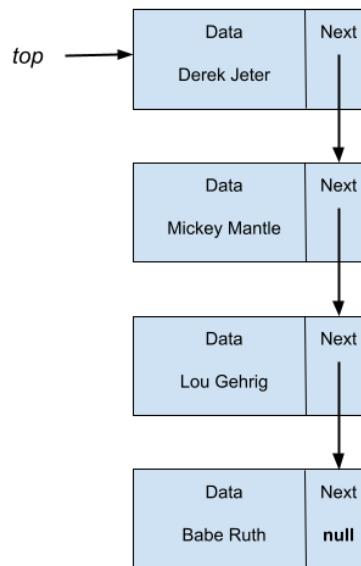
Figure 2.1: Example stack of 4 Yankees legends.

### 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 1.3.2), 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 1.3.2, 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.
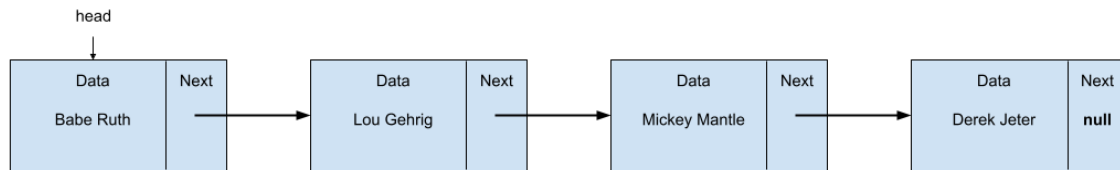
head

| Data | Next | | Data | Next | | Data | Next | | Data | Next |
| Babe Ruth | | | Lou Gehrig | | | Mickey Mantle | | | Derek Jeter | null |

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 1.3.2, 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

```cpp
#include <string>

#include "node.h"

template <typename T>
Node<T>::Node(T initialData) {
    // Initialize the node with the data and without a next node in the linked list
    Node::data = initialData;
    Node::next = nullptr;
}

// Define acceptable data types that the Node can accept for the template
template class Node<std::string>;
template class Node<char>;
```

Listing 1: node.cpp

```cpp
#pragma once

// Node represents an item within a singly linked list and can store data of a given type
template <typename T>
class Node {
    public:
        // A node has the data it is storing (of a type defined by the user) and a pointer
            to the next node
        T data;

        // The pointer uses the template to make sure all elements of the linked list store
            the same data type
        Node<T>* next;

        // Nodes will be instantiated with some data and not have a next node
        Node(T initialData);
};

// Super helpful resource on templates for c++
// https://isocpp.org/wiki/faq/templates#separate-template-fn-defn-from-decl
```

Listing 2: node.h

## 5.2 Stack

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

#include "stack.h"
#include "node.h"

// Instantiate the stack with the top pointing to nothing
template <typename T>
Stack<T>::Stack() {
    top = nullptr;
}

```

```
13  // Creates a new node and adds it to the stack
14  template <typename T>
15  void Stack<T>::push(Node<T>* newNode) {
16      // Set the next first so we do not lose the rest of the stack
17      newNode->next = top;
18      top = newNode;
19
20      // printStack();
21  }
22
23  // Removes the top node from the stack
24  template <typename T>
25  Node<T>* Stack<T>::pop() {
26      if (isEmpty()) {
27          // Throw an exception if the stack is already empty
28          throw std::invalid_argument("Stack underflow exception. Tried to pop from an empty
                  stack.");
29      } else {
30          // We need to collect the data in the node before removing it from the stack
31          Node<T>* topNode = top;
32          top = top->next;
33
34          // We have to remove whatever next is pointing to because the node is no longer
35          // a part of the linked list for the stack
36          topNode->next = nullptr;
37
38          return topNode;
39      }
40  }
41
42  // Checks to see if the stack is empty or not
43  template <typename T>
44  bool Stack<T>::isEmpty() {
45      return top == nullptr;
46  }
47
48  template <typename T>
49  void Stack<T>::printStack() {
50      // Get the top of the stack and iterate through, printing the data in each node
51      Node<T>* cur = top;
52      while (cur != nullptr) {
53          std::cout << cur->data << " -> ";
54          cur = cur->next;
55      }
56      // Finish the queue printing
57      std::cout << "nullptr" << std::endl;
58  }
59
60  // Define acceptable data types that the Stack can accept for the template
61  template class Stack<std::string>;
62  template class Stack<char>;
```

Listing 3: stack.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
```

```
12      Stack();
13
14      // Push adds a new element to the stack
15      void push(Node<T>* newNode);
16
17      // Pop removes the top element from the stack
18      Node<T>* pop();
19
20      // isEmpty checks to see if the stack is empty
21      bool isEmpty();
22
23      // Prints the stack out
24      void printStack();
25  };
```

Listing 4: stack.h

## 5.3 QUEUE

```
1  #include <string>
2  #include <iostream>
3
4  #include "queue.h"
5  #include "node.h"
6
7  // Instantiate the queue with the head pointing to nothing
8  template <typename T>
9  Queue<T>::Queue() {
10     head = nullptr;
11     tail = nullptr;
12  }
13
14  // Creates a new node and adds it to the queue
15  template <typename T>
16  void Queue<T>::enqueue(Node<T>* newNode) {
17     if (isEmpty()) {
18         // Immediately set the head and tail to be the new node if we are empty
19         head = newNode;
20         tail = newNode;
21     } else {
22         // Have the old tail to point to the new node and then update the pointer
23         tail->next = newNode;
24         tail = newNode;
25     }
26     // printQueue();
27  }
28
29  // Removes the front node from the queue
30  template <typename T>
31  Node<T>* Queue<T>::dequeue() {
32     if (isEmpty()) {
33         // Throw an exception if the queue is already empty
34         throw std::invalid_argument("Tried_to_dequeue_from_an_empty_queue.");
35     } else {
36         // We need to collect the data in the node before removing it from the queue
37         Node<T>* frontNode = head;
38         head = head->next;
39         if (head == nullptr) {
40             // Tail has to become nullptr because the queue is now empty
41             tail = nullptr;
42         }
43
44         // We have to remove whatever next is pointing to because the node is no longer
45         // a part of the linked list for the stack
46         frontNode->next = nullptr;
```

```
47
48          return frontNode;
49      }
50 }
51
52 // Checks to see if the queue is empty or not
53 template <typename T>
54 bool Queue<T>::isEmpty() {
55      return head == nullptr;
56 }
57
58 template <typename T>
59 void Queue<T>::printQueue() {
60      // Get the head of the queue and iterate through, printing the data in each node
61      Node<T>* cur = head;
62      while (cur != nullptr) {
63          std::cout << cur->data << " -> ";
64          cur = cur->next;
65      }
66      // Finish the queue printing
67      std::cout << "nullptr" << std::endl;
68 }
69
70 // Define acceptable data types that the Queue can accept for the template
71 template class Queue<std::string>;
72 template class Queue<char>;
```

Listing 5: queue.cpp

```
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
11      // Tail points to the end of the queue (tradeoff to get O(1) enqueue)
12      Node<T>* tail;
13 public:
14      // We need a constructor and destructor
15      Queue();
16
17      // Enqueue adds a new element to the queue
18      void enqueue(Node<T>* newNode);
19
20      // Dequeue removes the front element from the queue
21      Node<T>* dequeue();
22
23      // isEmpty checks to see if the queue is empty
24      bool isEmpty();
25
26      // Prints the entire queue out
27      void printQueue();
28 };
```

Listing 6: queue.h

## 5.4 MAIN PROGRAM

```
1 #include <iostream>
2 #include <string>
3
```

```cpp
#include "node.h"
#include "stack.h"
#include "queue.h"
#include "fileUtil.h"
#include "util.h"

// Function to test the Node class
void testNode() {
    // Create the nodes on the stack, so we do not have to delete later
    Node<std::string> n1("node_1");
    Node<std::string> n2("node_2");
    Node<std::string> n3("node_3");

    // Set up the links
    n1.next = &n2;
    n2.next = &n3;

    // Print out the data of each node in the linked list
    Node<std::string>* cur = &n1;
    while (cur != nullptr) {
        std::cout << cur->data << std::endl;
        cur = cur->next;
    }
}

// Function to test the Stack class
void testStack() {
    // Create a stack and the nodes
    Stack<char> stack;
    Node<char> hNode('h');
    Node<char> sNode('s');
    Node<char> oNode('o');
    Node<char> jNode('J');

    // Push each node onto the stack
    stack.push(&hNode);
    stack.push(&sNode);
    stack.push(&oNode);
    stack.push(&jNode);

    // Print out the letters as we remove them from the stack
    while (!stack.isEmpty()) {
        Node<char>* poppedElem = stack.pop();
        std::cout << poppedElem->data << '\t' << poppedElem->next << std::endl;
    }
    std::cout << std::endl;

    try {
        // This should throw an error
        stack.pop();
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
    }
}

// Function to test the Queue class
void testQueue() {
    // Create a queue and add some data to it
    Queue<char> queue;
    Node<char> jNode('J');
    Node<char> oNode('o');
    Node<char> sNode('s');
    Node<char> hNode('h');

    queue.enqueue(&jNode);
```

```
69      queue.enqueue(&oNode);
70      queue.enqueue(&sNode);
71      queue.enqueue(&hNode);
72
73      // Print out the letters as we remove them from the queue
74      while (!queue.isEmpty()) {
75          Node<char>* dequeuedElem = queue.dequeue();
76          std::cout << dequeuedElem->data << '\t' << dequeuedElem->next << std::endl;
77      }
78      std::cout << std::endl;
79
80      try {
81          // This should throw an error
82          queue.dequeue();
83      } catch (const std::invalid_argument& e) {
84          std::cerr << e.what() << std::endl;
85      }
86
87      // Proves type safety with templates and fails to compile
88      // Node<std::string> mismatchTest("Hello");
89      // queue.enqueue(mismatchTest);
90 }
91
92 // Function to check if a string is a palindrome, minus whitespace and capitalization
93 bool isPalindrome(std::string word) {
94      // Initialize an empty stack and queue for the checks
95      Stack<char> wordStack;
96      Queue<char> wordQueue;
97
98      // Iterate through each character in the word to populate the stack and queue
99      for (int i = 0; i < word.length(); i++) {
100         char character = word[i];
101         if (character == '_') {
102             // Go to next character because we are ignoring whitespace
103             continue;
104         } else if (character >= 'a' && character <= 'z') {
105             // Adjust the character to make it uppercase by taking the difference between
106             // the start of the lowercase letters and the start of the uppercase letters
107             character -= 'a' - 'A';
108         }
109         // Add the character to both the stack and the queue
110         Node<char>* charNodeStack = new Node<char>(character);
111         wordStack.push(charNodeStack);
112
113         Node<char>* charNodeQueue = new Node<char>(character);
114         wordQueue.enqueue(charNodeQueue);
115     }
116
117     while (!wordStack.isEmpty() && !wordQueue.isEmpty()) {
118         // Get the character from the top of the stack
119         Node<char>* nodeFromStack = wordStack.pop();
120         char charFromStack = nodeFromStack->data;
121
122         // Get the character from the top of the queue
123         Node<char>* nodeFromQueue = wordQueue.dequeue();
124         char charFromQueue = nodeFromQueue->data;
125
126         // We have to delete the nodes from the heap now that we are done with them
127         delete nodeFromStack;
128         delete nodeFromQueue;
129
130         if (charFromStack != charFromQueue) {
131             // We can return false because we already know that the string is not a
                    palindrome
132             return false;
```

```cpp
133            }
134        }
135
136        // The string is a palindrome
137        return true;
138 }
139
140 int main() {
141        std::cout << "————_Testing_Node_class_————" << std::endl;
142        testNode();
143        std::cout << std::endl;
144
145        std::cout << "————_Testing_Stack_class_————" << std::endl;
146        testStack();
147        std::cout << std::endl;
148
149        std::cout << "————_Testing_Queue_class_————" << std::endl;
150        testQueue();
151        std::cout << std::endl;
152
153        std::cout << "————_Testing_isPalindrome_————" << std::endl;
154        std::cout << isPalindrome("racecar") << std::endl; // 1
155        std::cout << isPalindrome("RaCecAr") << std::endl; // 1
156        std::cout << isPalindrome("ra___c_e___car") << std::endl; // 1
157        std::cout << isPalindrome("4") << std::endl; // 1
158        std::cout << isPalindrome("") << std::endl; // 1
159        std::cout << isPalindrome("ABC") << std::endl; // 0
160        std::cout << std::endl;
161
162        std::cout << "————_Magic_Items_————" << std::endl;
163        try {
164            // Read the file and store it in an array
165            StringArr* data = readFile("magicitems.txt");
166
167            // Only print out the palindromes
168            for (int i = 0; i < data->length; i++) {
169                if (isPalindrome(data->arr[i])) {
170                    std::cout << data->arr[i] << std::endl;
171                }
172            }
173
174            // Clean up memory
175            delete data;
176        } catch (const std::invalid_argument& e) {
177            std::cerr << e.what() << std::endl;
178        }
179
180        return 0;
181 }
```

Listing 7: main.cpp