Assignment One

Josh Seligman

joshua.seligman1@marist.edu

September 8, 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.

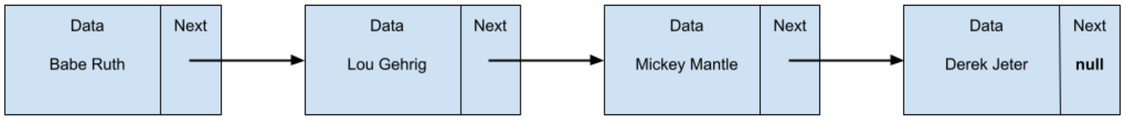

Figure 1.1: Example singly linked list of 4 Yankees legends.

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.

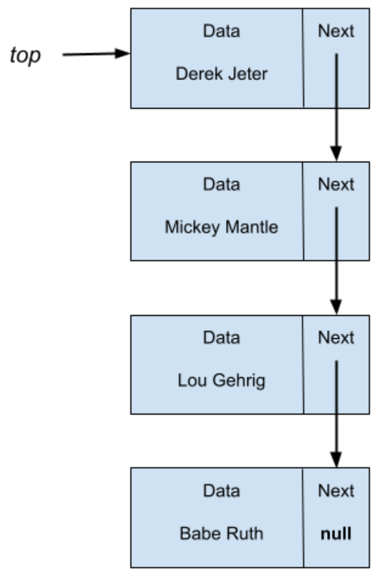

Figure 2.1: Example stack of 4 Yankees legends.

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

3.2 Benefits of a Stack

4 Main Program

4.1 Program Overview

4.2 Good Parts of the Program

5 Appendix

5.1 Singly Linked List

node.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>;
```

node.h

```
#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
```

5.2 Stack

stack.cpp

```
#include <string>

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

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

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

// Creates a new node and adds it to the stack
template <typename T>
void Stack<T>::push(T newData) {
    Node<T>* newNode = new Node(newData);
    // Set the next first so we do not lose the rest of the stack
    newNode->next = top;
    top = newNode;
}
```

```

// Removes the top node from the stack
template <typename T>
T Stack<T>::pop() {
    if (isEmpty()) {
        // Throw an exception if the stack is already empty
        throw std::invalid_argument("Tried to pop from an empty stack.");
    } else {
        // We need to collect the data in the node before removing it from the stack
        Node<T>* topNode = top;
        T topData = topNode->data;
        top = top->next;

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

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

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

stack.h

```
#pragma once

#include "node.h"

template <typename T>
class Stack {
private:
    // Top points to the top of the stack
    Node<T>* top;
public:
    // We need a constructor and destructor
    Stack();
    ~Stack();

    // Push adds a new element to the stack
    void push(T newData);

    // Pop removes the top element from the stack
    T pop();

    // isEmpty checks to see if the stack is empty
    bool isEmpty();
};
```

5.3 Queue

queue.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
template class Queue<std::string>;
template class Queue<char>;
```

queue.h

```cpp
#pragma once

#include "node.h"

template <typename T>
class Queue {
private:
    // Head points to the front of the queue
    Node<T>* head;
public:
    // We need a constructor and destructor
    Queue();
    ~Queue();

    // Enqueue adds a new element to the queue
    void enqueue(T newData);

    // Dequeue removes the front element from the queue
    T dequeue();

    // isEmpty checks to see if the queue is empty
    bool isEmpty();
};
```

5.4 Main Program

main.cpp

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

#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 add some data to it
    Stack<char> stack;
    stack.push('h');
    stack.push('s');
    stack.push('o');
    stack.push('J');

    // Print out the letters as we remove them from the stack
    while (!stack.isEmpty()) {
        std::cout << stack.pop();
    }
    std::cout << std::endl;

    try {
        // This should throw an error
        stack.pop();
    } catch (const std::invalid_argument& 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;
    queue.enqueue('J');
    queue.enqueue('o');
    queue.enqueue('s');
    queue.enqueue('h');

    // Print out the letters as we remove them from the queue
    while (!queue.isEmpty()) {
        std::cout << queue.dequeue();
    }
    std::cout << std::endl;

    try {
        // This should throw an error
        queue.dequeue();
    } catch (const std::invalid_argument& e) {
        std::cerr << e.what() << std::endl;
    }
}

// Function to check if a string is a palindrome, minus whitespace and capitalization
bool isPalindrome(std::string word) {
    // Initialize an empty stack and queue for the checks
    Stack<char> wordStack;
    Queue<char> wordQueue;

    // Iterate through each character in the word to populate the stack and queue
    for (int i = 0; i < word.length(); i++) {
        char character = word[i];
        if (character == ' ') {
            // Go to next character because we are ignoring whitespace
            continue;
        } else if (character >= 'a' && character <= 'z') {
            // Adjust the character to make it uppercase by taking the difference between
            // the start of the lowercase letters and the start of the uppercase letters
            character -= 'a' - 'A';
        }
        // Add the character to both the stack and the queue
        wordStack.push(character);
        wordQueue.enqueue(character);
    }

    while (!wordStack.isEmpty() && !wordQueue.isEmpty()) {
        // Get the character from the top of the stack and queue
        char charFromStack = wordStack.pop();
        char charFromQueue = wordQueue.dequeue();

        if (charFromStack != charFromQueue) {
            // We can return false because we already know that
            // the string is not a palindrome
            return false;
        }
    }

    // The string is a palindrome
    return true;
}

int main() {
    std::cout << "----- Testing Node class -----" << std::endl;
    testNode();
    std::cout << std::endl;

    std::cout << "----- Testing Stack class -----" << std::endl;
    testStack();
    std::cout << std::endl;

    std::cout << "----- Testing Queue class -----" << std::endl;
    testQueue();
    std::cout << std::endl;

    std::cout << "----- Testing isPalindrome -----" << std::endl;
    std::cout << isPalindrome("racecar") << std::endl; // 1
    std::cout << isPalindrome("RaCecAr") << std::endl; // 1
    std::cout << isPalindrome("ra   c e   car") << std::endl; // 1
    std::cout << isPalindrome("4") << std::endl; // 1
    std::cout << isPalindrome("") << std::endl; // 1
    std::cout << isPalindrome("ABC") << std::endl; // 0
    std::cout << std::endl;

    std::cout << "----- Magic Items -----" << std::endl;
    try {
        // Read the file and store it in an array
        StringArr* data = readFile("magicitems.txt");

        // Only print out the palindromes
        for (int i = 0; i < data->length; i++) {
            if (isPalindrome(data->arr[i])) {
                std::cout << data->arr[i] << std::endl;
            }
        }

        // Clean up memory
        delete data;
    } catch (const std::invalid_argument& e) {
        std::cerr << e.what() << std::endl;
    }

    return 0;
}
```