CMPT 435 - Fall 2022 - Dr. Labouseur

Assignment One

Jayden Melendez
Jayden.Melendez1@Marist.edu

October 4, 2024

Contents

1 Introduction

This document presents the implementation of the Node class in C++. Below is the code demonstrating its structure and functionality.

2 Node Class

Here is the C++ code for the Node class:

```cpp
// AssignmentOne/Node.h
// This class is the outline for the a node within a linked list
#ifndef NODE_H
#define NODE_H

#include <string>
class Node
{
public:
    // Variables
    std::string data;
    Node *next;

    // Constructor
    Node(const std::string &value);

    // Function outline to set the next node
    void setNext(Node *nextNode);

    // Function outline to get the next node
    Node *getNext() const;

    // Function outline to get the node's data
    std::string getData() const;
};

#endif
```

Figure 1: Header file for the Node class in a linked list.

```cpp
1  #include "node.h"
2
3  // Constructor
4  Node::Node(const std::string &value) : data(value), next(nullptr) {}
5
6  // Method to set the next node
7  void Node::setNext(Node *nextNode) {
8      next = nextNode;
9  }
10
11 // Method to get the next node
12 Node *Node::getNext() const {
13     return next;
14 }
15
16 // Method to get data within a node
17 std::string Node::getData() const {
18     return data;
19 }
```

Figure 2: C++ file for the Node class in a linked list.

2.1 The Node Class Overview

The **Node** class is a representation of the linked list data structure. Its structure involves a chain of linked objects. Objects are connected to each other through pointers. In my iteration of a Node class, there is a **next** variable of type Node that is used to define the pointer to the next object. The **data** variable represents the data stored inside the node. Its datatype is a string. The class constructor is defined on line 4. It takes in a string value which defines the data of the node. On line 7, the **setNext()** method takes in a Node object as an argument and sets the next pointer of the current node to point to the object passed in. On line 12, the **getNext()** method is used to return the next object in the linked list. The last method in this class is getData(). This method is used to retrieve the data within a node.

2.2 Things I Learned From The Node Class

The implementation of a simple Node class that represents a linked list solidified my knowledge about the versatility of this data structure. In future sections, I will explain the **Stack** and **Queue** classes that are both very similar to the Node class. Conceptually they are very similar. The main differences between them are the number of pointers pointing into objects and the order in which objects enter and leave the structure.

3 Queue Class

3.1 Code

```cpp
// AssignmentOne/queue.h
// This class is the outline of a Queue
// "First in First Out"
#ifndef QUEUE_H
#define QUEUE_H
#include "node.h" // Include the header file for the Node class
#include <string>

class Queue {
private:
    Node *head; // The head (front) of the queue (line)
    Node *tail; // The tail (back) of the queue (line)

public:
    Queue(); // Constructor outline
    void enqueue(std::string data); // Enqueue function outline
    std::string dequeue(); // Dequeue functon outline
    bool isEmpty(); // Returns if the queue is empty ot not
};

#endif
```

Figure 3: Header file for Queue class.

```cpp
// AssignmentOne/queue.cpp
// This class is the outline of a queue built from
// the outline of a linked list using the node class
// "First in First Out"
#include "queue.h"
#include <iostream>
#include <stdexcept>

// Constructor that intializes the head and tail of the queue to nullptr
Queue::Queue() : head(nullptr), tail(nullptr) {}

// Method to add data from the tail (back) into the queue
void Queue::enqueue(std::string data) {
    Node *node = new Node(data); // Create a new node
    if (tail != nullptr) {
        tail->next = node; // Make the tail point at the next node
    }
    tail = node; // Make the new node the current tail
    if (head == nullptr) {
        head = node; // Make the node the head
    }
}

// Method to remove data from the head (front) of the queue
std::string Queue::dequeue() {
    if (head == nullptr) {
        throw std::out_of_range("Queue␣is␣empty");
    }
    std::string data = head->getData(); // Get the data
    Node *oldHead = head; // Placeholder for removing the old head
    head = head->getNext(); // Make the new head
    if (head == nullptr) {
        tail = nullptr;
    }
    delete oldHead; // delete the old head
    return data;
}

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

Figure 4: C++ file for the Queue class.

3.2 The Queue Class Overview

A Queue is another elementary data structure that can be represented by the code derived from the **Node** class. The **Queue** class contains the variables **Head** and **Tail**. The head points to the front of the Queue. This is where Node objects exit the queue. The tail points to the back of the Queue. This is where Node objects enter the Queue. This data structure is known as first-in-first-out because any element that enters the Queue before the other element, will exit the Queue before the newer element. On line 9 of Figure 4, the Head and Tail are initialized to null pointer because the Queue is empty on initialization. On line 12, the **enqueue()** method allows objects to be passed into the Queue. On line 13, a new Node object is created that will be added to the Queue. On line 14, the conditional logic defines that every element after the tail is defined will become the new tail with the old tail pointing to it using the **Next** variable. On line 17, every new node is defined as the current tail. On line 18, if there isn't already a head pointing to the object behind it, the current node passed into the method enqueue() will be the head. The **dequeue()** method is used to take objects out of the Queue. On line 25, the method handles if there are no more objects to dequeue() in the Queue. On line 29, the data of the head is collected because that is what the dequeue() will return. Next, on line 30 and 31, the method handles setting the head as the next object in the queue. On line 32, dequeue() then handles if the Queue becomes empty on a dequeue() by setting a null pointer on the tail if the head is a null pointer.

3.3 Things I Learned About Queues

I continued to use the versatility of the **Node** class and C++ syntax to build out my data structures. Although it got pretty annoying at some points... handling **pointers** in C++ made it easier to conceptualize what the program was doing. Because data structures are mainly taught using pointers, it was very easy to imagine the Next variable from the node class acting as the pointer for elements in the Queue. In addition, handling the removal of a head by passing the value of the old head to a variable called oldHead widened my mental scope that oldHead was a pointer pointing to the value of the head that was being dequeued. After the oldHead was deleted, the old Node object will be pointed to by the current head that was added in.

4 Stack Class

4.1 Code

```
// AssignmentOne/stack.h
// This class is the outline of a stack built from the outline of a linked list using the no
// "First in Last Out"
#ifndef STACK_H
#define STACK_H
#include "node.h"  // Include the header file for the Node class
#include <string>

class Stack {
private:
    Node* top;  // Pointer to the top of the stack

public:
    Stack();  // Constructor
    void push(std::string data);  // Method to push an element onto the stack
    std::string pop();  // Method to pop an element from the stack
    bool isEmpty();
};

#endif // STACK_H
```

Figure 5: Header file for the Stack class.

```
 1 // AssignmentOne/stack.h
 2 // This class is the outline of a stack
 3 // "First in Last Out"
 4 
 5 #include "stack.h"
 6 #include <stdexcept>
 7 
 8 // Constructor definition
 9 Stack::Stack() : top(nullptr) {}
10 
11 // Method to push an element onto the stack
12 void Stack::push(std::string data) {
13     Node* node = new Node(data);  // Creating a node
14     node->setNext(top); // Set the current top node
15     top = node; // Change the top pointer to point to the new node
16 }
17 
18 // Method to pop an element from the stack
19 std::string Stack::pop() {
20     // Check to see if the stack is empty
21     if (top == nullptr) {
22         throw std::out_of_range("Stack␣is␣empty");
23     }
24 
25     std::string data = top->getData(); // Get the top data
26     Node* poppedNode = top; // Hold the previous top's data
27     top = top->getNext(); // Update the top pointer
28     delete poppedNode; // Delete the old top node
29     return data; // Return the data
30 }
31 
32 // Method to check if the stack is empty
33 bool Stack::isEmpty() {
34     if (top == nullptr) {
35         return true;
36     } else {
37         return false;
38     }
39 }
```

Figure 6: C++ file for the Stack class.

4.2 The Stack Class Overview

A stack is a data structure that abides by **first-in-last-out** logic. If an object is passed into the **Stack** and other objects follow, the objects that came after will exit the Stack first. In order to keep track of which object should come out of the stack first, on line 9 in the Stack constructor, the **top** pointer variable is defined. Its default value on a Stack initialization is a null pointer to represent a Stack with no contents. In order to add **Node** objects to the stack, the **push()** method is defined on line 12. This method takes in data and adds the data to the data variable defined by the Node class. One line 13, the Node class is used to instantiate a new node for the Stack. On line 14 and 15, the new node points to the current top pointer object and then is redefined to become the new top. The next method on line 19 is the **pop()** method. This method removes the top element from the stack. The method first checks on line 21 if the top variable is pointing to another object, if not, then the stack is empty. In a stack, the object is always the object at the top of the stack. On line 26 to 28, the top pointer is removed from the popped object and defined to the next object in the stack. On line 33, the final method is **isEmpty()**. This method checks if the top is a nullptr and returns a boolean depending on whether it is or not. This is useful when implementing the Stack to prevent null pointer errors.

4.3 Things I Learned

I have a calling to the Stack data structure. Something about its simplicity is very interesting to me. It is similar to the Queue in that we have to handle the pointers but only requires one pointer to pop() objects off it. I started with this data structure and it helped me wrap my head around how to translate the logic of data structures into code. It also taught me the importance of handling pointers in C++ and keeping track of the movement of objects. Although it was a struggle, it was worth the learning experience. I truly believe that I would not have understood the logic as much if I decided to program in Java over C++!

5 Sorting Class

5.1 Sorting Class Code

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


class Sorting {

public:
    int comparisonCounter;
    Sorting();
    int doSelectionSort(std::string* items, int size);
    int doInsertionSort(std::string* items, int size);
    int doMergeSort(std::string *items, int size);
    std::string* doMerge(std::string *items,
    std::string *leftItems, std::string *rightItems,
    int leftSize, int rightSize);
    int setPartition(std::string* items,
    int itemFromLeft, int itemFromRight);
    int doQuickSort(std::string *items,
    int itemFromLeft, int itemFromRight);
    std::string* doKnuthShuffle(std::string* items, int size);
};
#endif
```

Figure 7: Header file for Sorting Algorithms

5.2 Selection Sort Code

```
 1 #include "sorting.h"
 2 #include <string>
 3 
 4 /** This file contains all method calls for different sorting algortihms
 5  * @author Jayden Melendez
 6  * @date October 2nd, 2024
 7 */
 8 
 9 // Constructor
10 Sorting::Sorting() {
11     comparisonCounter = 0; // Counting the number of comparisons
12 }
13 
14 // Selection Sort
15 int Sorting::doSelectionSort(std::string *items, int size) {
16     // Iterate through magic items
17     for (int i = 0; i < size; ++i) {
18 
19         int minIndex = i; // Minimum starts at current index
20         std::string min = items[i]; // Minimum starts at the current index
21         // Iterate through magic items following the current index
22         for (int j = i + 1; j < size; j++) {
23             comparisonCounter++; // Increment the comparison
24             // Compare each string to the current minimum
25             if (items[j] < min) {
26                 min = items[j]; // Define the new minimum
27                 minIndex = j;   // Define location of the new minimum
28             }
29         }
30         // Swap current string with the smallest string found
31         std::string temp = items[i];
32         items[i] = items[minIndex];
33         items[minIndex] = temp;
34     }
35     return 0;
36 }
```

Figure 8: Method for Selection Sort

5.3 Selection Sort Overview

The **Sorting** class is defined by four different sorts (so far). The only variable in the constructor on lines 10-12 is the **comparisonCounter**. It is responsible for keeping track of the number of times two elements are compared in a sort. The first sorting algorithm is **Selection Sort**. In my opinion, this was the simplest sorting algorithm to conceptually visualize. The method for selection sort takes in the pointer to items **items** on line 15, which is an array of strings of **magic items**. On line 17, the algorithm begins by entering a **for** loop that iterates through each element of the array. On line 19, the minIndex will start as the first index of the array. The value of the first index is also assigned to a variable, **min**. These values will dynamically change as we iterate through the array and make comparisons with other elements. The inner for loop on line 22 then iterates through all elements starting at the index defined in the outer for loop +1 aka (j = i + 1). On line 23, the global variable comparisonCounter is incremented to keep track that an element from the outer loop is compared with an element in an inner loop. The inner loop is constantly checking to see if the value at index **j** is smaller than the value at the **min** value at the line 25 conditional. When the entire array, following the starting point of the outer loop, is iterated through, the most minimum value is set and swapped with the value at i. This process continues until the outer loop reaches the end of the array and the current **min** for comparison, not the minimum for the entire array, is the last element.

5.4 Things I learned About Selection Sort

Selection Sort is one of the easiest sorts to implement. The space complexity is determined by the fact that selection sort is an in-place sort. Therefore, its space complexity is:

$$O(n)$$

However, the number of comparisons is determined by the fact that for every element in the array (order n times), each following element needs to be compared another order n times. Therefore, the time complexity is:

$$O(n^2)$$

This logic transcends many algorithms! When there is a double for loop, we are comparing every element to every other element in an array. This at least makes the algorithm order n squared time before considering other complexities that might supersede order n squared. I also learned and found it interesting that Selection Sort has a static number of comparisons for the same array! By same array I mean, the same size but elements are in a different order. Because every element needs to be compared to every other element, their order does not matter.

5.5 Insertion Sort Code

```
1  // Insertion Sort
2  int Sorting::doInsertionSort(std::string *items, int size) {
3      // Iterate through magic items array
4      for (int i = 1; i < size; i++) {
5          std::string currentWord = items[i]; // Current word
6          int j; // Hold value for insertion
7          // Compare current word with sorted elements until it's smaller
8          for (j = i - 1; j >= 0 && items[j] > currentWord; j--) {
9              comparisonCounter++; // Increment the comparison
10             items[j + 1] = items[j]; // Shift right
11         }
12         items[j + 1] = currentWord; // Insert the current word
13     }
14     return 0;
15 }
```

Figure 9: Method for Insertion Sort

5.6 Insertion Sort Overview

Insertion Sort shares the same complexity as Selection Sort which is big-oh of n squared. This can be determined because of the nested for loop starting at line 4 and the inner for loop starting at line 8. The algorithm starts on line 2 with the method call of **doInsertionSort()**. This method takes in the array of items as strings and the size of the array. On line 4, the outer for loop increments until it has iterated through the array. On line 5, the currentWord is defined as the string that is going to be compared to the values at the previous index in the array. On line 6, the variable j is declared to represent the index going backwards from the index of currentWord. The inner loop logic on line 8 works a little differently than Selection Sort. The logic of this loop shifts elements to the right of the array if it was larger than the current word. Once the condition in the for loop is false, this means that currentWord is greater than the value it was compared to index j. At line 12, the currentWord takes the empty place that follows the index of the value that it is greater than.

5.7 Things I Learned About Insertion Sort

I found it interesting to see the next "simplest" approach to sorting. I learned that insertion sort conceptualizes sorting with a sorted part and unsorted part. The **currentWord** is shifted to the right and looks at all the elements behind it. This makes it so there is always a "sorted" part of the array. However, like selection sort, insertion sort iterates through all of the elements in the array. It then compares all previous elements to the current element. This is seen as another "iteration" through the array. In the worst case scenario, sorted in decreasing order, every element would need to be compared and swapped. This leads to the complexity:

$$O(n^2)$$

5.8 Merge Sort Code

```cpp
// Merge Sort
int Sorting::doMergeSort(std::string *items, int size) {
    // Return the base case
    if (size <= 1) {
        return 0;
    }
    int middle = size / 2; // Determine the middle index
    std::string *leftItems = new std::string[middle]; // Define the left
    std::string *rightItems = new std::string[size - middle]; // right

    // Populate left items from the left side of items
    for (int i = 0; i < middle; i++) {
        leftItems[i] = items[i];  // Add the items to the left array
    }
    // Populate right items from the right side of items
    for (int i = 0; i < size - middle; i++) {
        rightItems[i] = items[i + middle];  // Add the items on the right
    }
    // Recursively call method until base case is reached
    doMergeSort(leftItems, middle);  // Handle the left branch
    doMergeSort(rightItems, size - middle);  // Handle the right branch

    // Merge leftItems and rightItems until the full array is rebuilt
    // sorted
    std::string* mergedItems = doMerge(items, leftItems, rightItems,
    middle, size - middle);

    // Open up pointers
    delete[] leftItems;
    delete[] rightItems;

    return 0;
}
```

Figure 10: Recursive Method for Merge Sort

5.9 Merge Sort Overview

Merge sort is known as a **Divide-And-Conquer** algorithm. This recursive method is responsible for taking an array and breaking it down until it gets to many single element arrays. A single element array is sorted, so we can treat their values as individual elements. On line 2, similar to the previous sorts, the pointer to the array of items and the size of the array are passed in as parameters. One line 4, a conditional defines the base case. When the method is called over and over, the size of the array is halved. This is seen on line 7. Therefore, we can use the value of size to gauge when the many arrays all reach a length of one. In addition to halving to represent the divide portion of the algorithm, lines 7 and 8 split the array into the left and right side. When the method is called over and over, the left and the right arrays are constantly halved into more left and rights. This continues until the base case is reached. On line 12, we begin to iterate through the left array to add all items on the left side of the items array to the previous left array. The same is done for the right on line 16. The first recursive call on line 20 handles the next split for the current **left** of the array of items that was currently passed on. The next recursive call on line 21 is used to split the right sides. Line 24 is the call to the **doMerge()** method that will rebuild the arrays into a sorted array. In order to prevent pointer overlap, lines 27 and 28 delete any pointers for the left and right arrays.

```
 1 // Method to merge many arrays for Merge Sort
 2 std::string* Sorting::doMerge(std::string *items,
 3 std::string *leftItems, std::string *rightItems,
 4 int leftSize, int rightSize) {
 5     // Define helper variables for comparison
 6     int i = 0;
 7     int left = 0;
 8     int right = 0;
 9 
10     // Compare left and right items for merging
11     while (left < leftSize && right < rightSize) {
12         comparisonCounter++; // Increment the comparison counter
13         if (leftItems[left] < rightItems[right]) {
14             items[i] = leftItems[left];
15             left++;
16         } else {
17             items[i] = rightItems[right];
18             right++;
19         }
20         i++;
21     }
22 
23     // Handle remaining elements from leftItems
24     while (left < leftSize) {
25         items[i] = leftItems[left];
26         left++;
27         i++;
28     }
29 
30     // Handle remaining elements from rightItems
31     while (right < rightSize) {
32         items[i] = rightItems[right];
33         right++;
34         i++;
35     }
36 
37     return items;  // Return the pointer to the merged array
38 }
```

Figure 11: Recursive Method for Merging In Mergesort

5.10 Merge Overview

Now that the array was divided into many one long arrays, it is time to merge them back together. This method takes in many arguments. It takes in the item's pointer to merge the left and right arrays back into the items array. The left items and right items are the two arrays that need to be compared and put in the correct order based on their contents. The left and right size parameters are used to keep track of the size of the left and right array. The **while** loop, starting on line 11, is the basic logic that will compare all the divided arrays and merge them back together. The logic allows the code to iterate through the **leftItems** array and **rightItems** array until it reaches the end. The conditional on line 13 is then very simple. When an item in the left array is less than an item in the right array at their respective indices, the left item is placed in the current main array. The opposite happens starting on line 16 for when an element in the right array is less than an element in the left array. When there are an odd number of elements, it is possible that they are not handled from the first while loop. The while loops on line 24 and 32 deal with this for the left and right arrays respectively.

5.11 Things I Learned About Merge Sort

Recursion has never been my strong suit. It was hard to map out in my head and even harder to debug. However, merge sort allowed me to conceptually practice it. Merge sort was a prime example of recursion because it was simple to understand that smaller and smaller arrays are passed into recursive calls that continue to split until they reach the base case. I also learned that this algorithm saves time but takes up more space when compared to selection sort and insertion sort. Merge sort is not in place sorting. This means that we need to create new arrays to do our dividing. This takes up more memory. However, the nature of the recursive calls of halving an array with a left and right forms a **tree structure**. Each split of an array is represented as a certain position, level, on the tree. Each position can be represented by n for the number of elements that are compared to one another when merging. The height of the tree structure is represented by:

$$log_2(n)$$

Therefore, we multiply the time complexity of each level by the height of the tree and we get the worst case complexity of merge sort which is

$$n * log_2(n)$$

5.12 Quick Sort Code

```
1 int Sorting::doQuickSort(std::string *items, int start, int end) {
2     if (start < end) {
3         int partition = setPartition(items, start, end); // Partition array
4         doQuickSort(items, start, partition - 1);  // Recursively sort left
5         doQuickSort(items, partition + 1, end);  // Recursively sort right
6     }
7     return 0;
8 }
```

Figure 12: Code for Quicksort

5.13 Quick Sort Overview

Ahhhh **Quicksort**. I would say it was fun... I mean... of course it was fun! However, I know now that the name didn't come from a college student making it from scratch for the first time because this process wasn't quick. I'll talk more about it later. The method above is recursive. It takes in the items array pointer, the start index, and the end index. The base case of this algorithm, found on line 2 is when the start index is greater than the end index. This means that we iterated through the entire array. Line 3 holds the value of the partition. The partition is the value that all other elements are compared against so they can be moved to the correct side. Line 4 is the first recursive call of quick sort that deals with all elements to the left of the partition. The second is on line 5 is for the right part. The partition, the pivot value in the helper method, will constantly decrement in the first recursive call as the left array gets smaller and smaller. The right partition increments until it gets to the start value gets to the end of the right array. This is when the arrays are of size 1.

```
 1 int Sorting::setPartition(std::string *items, int start, int end) {
 2     std::string temp = "";  // Temp variable for swapping
 3     std::string pivotValue = items[end];  // Pivot is the last element
 4     int i = start - 1;  // Helper index starts before the first element
 5 
 6     // Iterate through to the second last element
 7     for (int j = start; j < end; j++) {
 8         comparisonCounter++;
 9         if (items[j] <= pivotValue) {
10             i++;  // Increment helper index
11             // Swap items[i] and items[j]
12             temp = items[i];
13             items[i] = items[j];
14             items[j] = temp;
15         }
16     }
17     // Place the pivot in its correct position
18     temp = items[i + 1];
19     items[i + 1] = items[end];
20     items[end] = temp;
21     return i + 1;  // Return pivot position
22 }
```

Figure 13: Partition Method for Quicksort

This method is the meat and potatoes of the algorithm. In my algorithm, I chose the last element of the array as the pivot value. Although this isn't the most optimal value, it was the one that I was able to effectively implement. On line 2, the temp variable is defined for switching values. On line 3, the pivot value is set to be the last item in the array. The helper index is the value right before the start. The for loop on line 7 iterates through every element except the final element because that is the pivot value. When an item is less than the pivot value during the loop, the value at the helper index and the current index swap. When the loop finishes, on line 18, The new pivot value is the value of the next helper index. This is also known as the end of the current split array.

6 Code For Splitting Strings

```
 1 // Method to split strings
 2 vector<char> split(string str, char del)
 3 {
 4     vector<char> characters; // Characters in the string
 5     string temp = "";         // Temporary string
 6 
 7     for (int i = 0; i < (int)str.size(); i++)
 8     {
 9         if (str[i] == '␣')
10         {
11             continue; // Skip spaces
12         }
13         // If the character is not the "delimeter", add it to  temp string
14         if (str[i] != del)
15         {
16             temp += str[i];
17         }
18         else
19         {
20             // Add the characters to the characters vector
21             for (int j = 0; j < (int)temp.size(); j++){
22                 characters.push_back(temp[j]);
23             }
24             temp = ""; // Reset the temporary string
25         }
26     }
27 
28     // Loop just in case the string doesn't end in the delimeter
29     for (int j = 0; j < (int)temp.size(); j++){
30         characters.push_back(temp[j]); // Add each character of  last word
31     }
32 
33     return characters;
34 }
```

Figure 14: Method String Splitting

In C++, there was no split method in the basic library to call. Well, maybe there is somewhere but I was too lazy to look for it. The method returns a vector of characters that are characters found within the string that is passed into the method. The delimiter is the value that string is being split on. One line 4 and 5, the characters vector and temp string for parsing out unneeded values defined by the delimiter. The loop on line 7 loops through the string, skips spaces on line 11, adds characters that are not the delimiter to the characters temp string, and finally on lines 20-30, another 2 loops handles adding characters to the characters vector.

7 Code For Finding Palindromes

```
 1 // Method to search for palindromes in an array
 2 int searchForPalindromes(std::string *items, int size) {
 3     // Vector to store discovered palindromes
 4     std::vector<std::string> palindromes;
 5     // Loop through each word in the array
 6     for (int i = 0; i < size; i++) {
 7         Stack stack; // Define the stack for reverse string reading
 8         Queue queue; // Define the queue for forward string reading
 9         bool isPalindrome = true;
10         char del = '␣';
11         // Split on each character
12         vector<char> characters = split(items[i], del); // Call the split
13 
14         // Loop through the characters vector for adding characters
15         for (int j = 0; j < (int)characters.size(); j++) {
16             stack.push(std::string(1, characters[j]));
17             // Push onto the stack
18             queue.enqueue(std::string(1, characters[j])); // Enqueue
19         }
20         // Loop through the characters vector for comparing characters
21         for (int k = 0; k < (int)characters.size(); k++) {
22             // Take out the next character from the queue
23             std::string queueValue = queue.dequeue();
24             // Take out the next character from the stack
25             std::string stackValue = stack.pop();
26             queueValue[0] = std::tolower(queueValue[0]);
27             stackValue[0] = std::tolower(stackValue[0]);
28             if (queueValue != stackValue) {
29                 isPalindrome = false;
30                 std::cout << "Mismatch␣found.␣Exiting
31 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣comparison." << std::endl; // Debug output
32                 break;
33             }
34         }
35         // If the word was a palindrome, add it to the vector
36         if (isPalindrome){
37             palindromes.push_back(items[i]);
38         }
39         cout << endl; // New line after each item
40     }
```

Figure 15: Method String Splitting

This method is used for looking for palindromes. On line 4, the vector that will store all palindromes is defined. On line 6, there is a loop that iterates through the array of all items. On line 7 and 8, the stack and queue are defined. The stack will be the data structure used to read characters of a string in reverse order. A queue is a data structure that will read characters in a string forwards. The boolean on line 9 is a tag that will only add a string if the flag is still true. The character vector on line 12 splits the current item at the out loop index. The first inner loop pushes characters onto a stack on line 16 and queues elements into a queue at line 18. The loop on line 21 iterates through the vector array to capture the amount of times it takes to iterate through all the characters of the current word to compare each character. On line 23, the queue is dequeued once and on line 25, the stack is popped once. On line 26 and 27, the characters are made lowercase to ensure an accurate comparison. There is then a conditional on line 28 to check whether the dequeued character and the popped character are the same. If they are the same, the **isPalindrome** boolean remains true and the loop repeats without going into the condition. When the characters are different, the condition is entered and isPalindrome becomes false because the word is not a palindrome.

8 Knuth Shuffle

8.1 Knuth Shuffle Code

```
 1 // Knuth Shuffle
 2 std::string *Sorting::doKnuthShuffle(std::string *items, int size) {
 3     // Iterate through the array backwards
 4     for (int i = size - 1; i > 0; i--)
 5     {
 6         int randomIndex = std::rand() % (i + 1); // Generate random number for index
 7         // Swapping
 8         std::string temp = items[i];
 9         items[i] = items[randomIndex];
10         items[randomIndex] = temp;
11     }
12     return items; // Return the pointer to the same array
13 }
```

Figure 16: Method String Splitting

8.2 Knuth Shuffle Overview

The **The Knuth Shuffle** is a very easy algorithm for a shuffle. It involves choosing random numbers to move elements to random indices in the array. On line 2, the **doKnuthShuffle()** method takes in the items array and its size as parameters. On line 4, the array is iterated through backwards. One line 6, a random number is chosen for the random index. Now the algorithm just takes the item at the index that loop is on and swap it with the item at the random index. The method returns the shuffled array at line 12.

9 Test Cases

Table 1: Comparison of the Number of Comparisons for Each Sorting Algorithm

Trial	Selection Sort$O(n^2)$	Insertion Sort$O(n^2)$	Merge Sort$O(n \log n)$	Quick Sort$O(n \log n)$
1	221,445	106,003	5,409	7,419
2	221,445	107,534	5,430	7,037
3	221,445	111,715	5,394	7,491
4	221,445	115,594	5,426	7,141
5	221,445	107,516	5,424	7,591
6	221,445	108,216	5,411	7,442
7	221,445	110,329	5,423	7,471
8	221,445	108,986	5,405	7,573
9	221,445	113,201	5,427	7,537
10	221,445	109,510	5,419	6,916
Average	221,445	109,060	5,416	7,362

10 Insights on Complexity and Comparisons

Starting with **Selection Sort**, the most notable characteristic about all of the comparisons is that they are all the same for all of the test cases. Selection sort has the same time complexity for the best, average, and worst case. This is due to the algorithm always having to make the same number of comparisons regardless of the order the elements are in the array. Selection Sort is an in-place sorting algorithm. Even if the elements are already sorted, the algorithm still needs to compare every element to each other. This means that it always requires a constant amount of memory to sort its contents, regardless of the size of the array. For each test case, the number of comparisons for selection sort is 221,445. By mathematical definition, the average running time of selection sort is

$$n(n-1)/2$$

In Selection Sort,n represents the number of items in the array that is being sorted. Selection sort represents an arithmetic series. This is because for each iteration, we are multiplying n by the number of elements that are "left" to sort in the array. This is know as an **arithmetic series**. The equation S represents the sum of K integers.

$$S = k(k+1)/2$$

In the case of selection sort, K is equal to (n-1). When simplified, we find that the complexity of selection sort is

$$S = n(n-1)/2$$

This arithmetic series represents the insertion sort comparisons of less and less elements. However, the current number of elements following the "current" element are all compared to the "current" element. Therefore, we can translate the series to English by stating, compare a starting element with the preceding element (n-1) on the first iteration. On the next iteration, compare the starting element with the preceding 2 elements (n-2) and so on. This is the worst case because in some scenarios, the current element is already greater than the first preceding element. This means that the algorithm does not have to continue down the chain of decreasing n. The following is a mathematical representation of this process.

$$(n-1) + (n-2) + (n-3) + \ldots + 1$$

Using this equation, we can calculate the predicted number of comparisons. The prediction is 221,555. The logic from the code varies slightly from the prediction because the number of comparisons does not include the final element. Selection sort constantly moves the **minimum** to a new element, therefore the last element does not need to be compared to anything. In this **Selection Sort** complexity equation, the start of the series, n, is represented by **n-1** in practice.

$$sum = (n-1) * (n-1)/2$$

Moving on to **Insertion Sort**, it has the same **AVERAGE** time complexity as selection sort. Insertion sort can be order n time if the array being sorted is **ALREADY** sorted. This is because, by definition, insertion sort compares an element with preceding elements starting at the second index. If the array is already sorted, the algorithm only has to compare each element to the preceding element once. This results in only having to iterate through the array once. In the worst-case scenario, the array is in reverse order. When the array is in reverse order, all elements need to be compared to all preceding elements, which results in an order of n squared time complexity. Not only does the algorithm have to iterate through the array, but it also has to iterate backward for each element and shift larger elements to the right until each element reaches its correct position. This makes Insertion Sort more versatile than Selection Sort because it is more likely that an array is already semi-sorted. In other words, the number of preceding comparisons is never the "max" for each element. This makes the average number of comparisons for insertion sort significantly lower than for selection sort. However, the worst-case scenario is a reversed order list. Therefore, it is possible to have a result of 221,445 comparisons for the worst case of big-O of n squared. The results in the table can be attributed to the shuffle algorithm. It is possible that the shuffle algorithm is prone to producing a semi-sorted list from an insertion sort perspective. Although the average number of comparisons for insertion sort is half the size of the list, the test cases showed it was closer to a representation like:

$$sum = (n) * (n - 1)/4$$

The first divide and conquer algorithm implemented was **Merge Sort**. The algorithm represents *n log n* complexity. As stated before, the tree structure formed when dividing an array is represented by log base 2 of n. When substituted for the number of elements, the expected average number of comparisons is 6,247. On average, the number of comparisons was better than the worst-case scenario of 6,247. This is due to the nature of sorting the array. On average, the sort will be good enough that it does not reach the worst-case scenario. Therefore, the average complexity can still be represented by:

$$sum = n * log_2(n)$$

The second divide and conquer algorithm, **Quick Sort**, has the same complexity as **Merge Sort**. However, there is a worst-case scenario that can lower the efficiency of Quick Sort. This algorithm involves choosing a pivot value that is compared to the other values in the array. If the array is already sorted and the pivot value is chosen as the last value, there will be a large imbalance in the algorithm. The left side of the array would be very large, while the right is very small. Therefore, the pivot would have to be compared to all the preceding elements to prove it is the largest, with every element becoming the new "pivot." This makes the complexity rise to:

$$O(n^2)$$

11 Insights on the Usefulness of AI

Artificial Intelligence has become a very useful tool in the realm of computer science. More than anything, I think it is the best debugging tool available. Although **Large Language Models** (LLMs) are not great at reasoning and deciding which answer is right, they excel at logical operations. So instead of typing, "Can you tell me where the logic in my selection sort method has gone wrong?", I would type, "Take this C++ method and define the parameters with this dataset [A, B, C, ...]. Walk through the method and tell me the current state of sorting for each comparison. Print out the final sorted array." Although this is more effort than just asking for the answer, it prevents the **hallucinations** that plague many LLMs. The model knows what is correct from a logical standpoint, so running through the logic of a function and producing an incorrect output convinces the LLM that something is wrong. At that point, I ask, "What are the possible reasons the final array is unsorted?" Nine times out of ten, it provides an accurate answer. Most of the time, it was an off-by-one error or calling the wrong variable—issues that can be difficult to spot at 2 AM. This allowed me to save much more time when working on divide and conquer sorting algorithms because the debugger makes it difficult to follow recursive structures.

12 Conclusion

I found C++ to be a very useful language for this project. Conceptually, managing pointers makes more sense than just defining a variable in Java. It was more enjoyable than not at times because I was simply applying what I already knew from coding in other languages. This project covered all the basic coding practices, so I can now say that I am decently proficient in C++. In my internship, I am still using Python for most of the tools I am working on. It has made me realize that I don't enjoy it as much anymore and prefer the structure of statically-typed languages. It is much more helpful to have an error message in my terminal than a blank one. This project was also really helpful for understanding basic algorithm and data structure concepts for interviews. I also experienced a shift in my thinking—it's almost as if I wasn't really a software developer until I completed this first project. My critical thinking skills have improved to the point where I will have much more confidence in my next technical interview.