# Assignment One

Ethan Ondreicka

Ethan.Ondreicka1@Marist.edu

October 7, 2023

## 1 Part One: Node initialization

### 1.1 Developing the Node Class

```
// Constructs the node
struct Node {
    char data;
    Node* next;

    Node(char ch) : data(ch), next(nullptr) {}
};
```

Listing 1: Construction of the Node Class

## 2 Part Two: Palindromes

### 2.1 Creating Stacks

```
// Stack implementation using linked list
template <typename T>
class MyStack {
private:
    struct Node {
        T data;
        Node* next;
        Node(const T& item) : data(item), next(nullptr) {}
    };
    Node* head;

public:
    MyStack() : head(nullptr) {}

    void push(const T& item) {
        Node* newNode = new Node(item);
        newNode->next = head;
        head = newNode;
    }
```

```
20
21      void pop() {
22          if (head) {
23              Node* temp = head;
24              head = head->next;
25              delete temp;
26          }
27      }
28
29      T& peek() const {
30          if (head) {
31              return head->data;
32          }
33          throw std::out_of_range("Stack is empty");
34      }
35
36      bool empty() const {
37          return head == nullptr;
38      }
39  };
```

Listing 2: Creation of the Stack class

Through the class MyStack, I was able to create a template for creating a stack. Lines 15 - 19, each character in the string of text gets put into the stack. And in lines 21 - 27, each character is removed from the stack.

## 2.2 Creating Queues

```
1   // Queue implementation using linked list
2   template <typename T>
3   class MyQueue {
4   private:
5       struct Node {
6           T data;
7           Node* next;
8           Node(const T& item) : data(item), next(nullptr) {}
9       };
10      Node* head;
11      Node* tail;
12
13  public:
14      MyQueue() : head(nullptr), tail(nullptr) {}
15
16      void enqueue(const T& item) {
17          Node* newNode = new Node(item);
18          if (tail) {
19              tail->next = newNode;
20          } else {
21              head = newNode;
22          }
23          tail = newNode;
24      }
25
26      void dequeue() {
27          if (head) {
28              Node* temp = head;
29              head = head->next;
30              delete temp;
31              if (!head) {
32                  tail = nullptr;
33              }
34          }
35      }
```

```
36
37      T& headElement() const {
38          if (head) {
39              return head->data;
40          }
41          throw std::out_of_range("Queue is empty");
42      }
43
44      bool empty() const {
45          return head == nullptr;
46      }
47  };
```

Listing 3: Creation of the Queue Class

With the creation of the MyQueue template, I was able to put my entire array into a queue and push and pop each character of every string.

## 2.3 Comparing the Stacks and Queues

```
1   bool isPalindrome = true;
2
3           while (currentNode != nullptr) {
4               charStack.push(currentNode->data); // Push onto the stack
5               charQueue.enqueue(currentNode->data); // Enqueue into the queue
6               currentNode = currentNode->next;
7           }
8
9           while (!charStack.empty() && !charQueue.empty()) {
10              char stackTop = charStack.peek();
11              char queuehead = charQueue.headElement();
12
13              if (stackTop != queuehead) {
14                  isPalindrome = false;
15                  break; // Not a palindrome, no need to continue checking
16              }
17
18              charStack.pop();
19              charQueue.dequeue();
20          }
```

Listing 4: Comparing each character to see if it is a palendrome

Through these while loops in my main function, I am able to compare the heads of both the stack and the queue. Lines 4-5 in the first while loop push each character node onto the stack and queue respectively. In the second while loop, if neither the stack or queue is empty, it will initialize the variables on lines 10-11 and in the for loop below it will compare character by character.

# 3 Part Three: Sorting Algorithms

## 3.1 Selection Sort

```
1     int selectionSortCount = 0;
2
3     // Selection sort
4     for (int i = 0; i < lineCount - 1; i++) {
5         int smallestLine = i;
6         for (int j = i + 1; j < lineCount; j++) {
7             // Counts the amount of comparisons
8             selectionSortCount ++;
9             if (shuffled_lines[j] < shuffled_lines[smallestLine]) {
10                smallestLine = j;
11            }
12        }
13        if (smallestLine != i) {
14            std::swap(shuffled_lines[i], shuffled_lines[smallestLine]);
15        }
16    }
```

Listing 5: Selection Sort Algorithm

The Selection Sort algorithm repeatedly goes through the sorted array over and over again from the beginning until every item in the array is sorted.

Selection sort has an asymptotic running time of $O(n^2)$ because no matter what, it will continue to go through every element in the array once, whether it is sorted or unsorted already.

## 3.2 Insertion Sort

```
1     // Insertion sort
2     int insertionSortCount = 0;
3     for (int i = 1; i < lineCount; i++) {
4         std::string valueToInsert = shuffled_lines[i];
5         int j = i - 1;
6
7         while (j >= 0 && shuffled_lines[j] > valueToInsert) {
8             shuffled_lines[j + 1] = shuffled_lines[j];
9             j = j - 1;
10            insertionSortCount++;
11        }
12        shuffled_lines[j + 1] = valueToInsert;
13    }
```

Listing 6: Insertion Sort Algorithm

The Insertion Sort algorithm works by iteratively selecting elements from the unsorted portion of the array and inserting them into their correct positions in the sorted portion. The while loop starting on line 7 shifts elements in the sorted portion of the array to the right until the correct position for "valueToInsert" is found.

Insertion Sort has an asymptotic running time of $O(n^2)$, because it goes through a list one element at a time and repeats until all items are sorted.

### 3.3 Merge Sort

```
    // Creates the merge function
void merge(std::string arr[], int start, int middle, int end, int& comparisonCount) {
    int leftSize = middle - start + 1;
    int rightSize = end - middle;

    // Create temporary arrays for left and right
    std::string left[leftSize];
    std::string right[rightSize];

    for (int i = 0; i < leftSize; i++) {
        left[i] = arr[start + i];
    }
    for (int j = 0; j < rightSize; j++) {
        right[j] = arr[middle + 1 + j];
    }

    // Merge the two halves back into the original array
    int i = 0, j = 0, k = start;
    while (i < leftSize && j < rightSize) {
        comparisonCount++;
        if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        } else {
            arr[k] = right[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements of left[] and right[] if any
    while (i < leftSize) {
        arr[k] = left[i];
        i++;
        k++;
    }
    while (j < rightSize) {
        arr[k] = right[j];
        j++;
        k++;
    }
}
```

Listing 7: Merge Function

```
    // Merge sort function
void mergeSort(std::string arr[], int start, int end, int& comparisonCount) {
    if (start < end) {
        int middle = (start + end) / 2;

        // Recursively sort the left and right halves
        mergeSort(arr, start, middle, comparisonCount);
        mergeSort(arr, middle + 1, end, comparisonCount);

        // Merge the sorted halves
        merge(arr, start, middle, end, comparisonCount);
    }
}
```

Listing 8: Merge Sort Algorithm

The Merge Sort algorithm is split into two different functions, one for splitting the array, and the other for merging. This is called "dividing and conquering", as it takes one big problem and makes it into a bunch of smaller problems that are then much easier to solve. The first function splits the shuffled array in half into two temporary arrays. The second function merges the arrays back together again comparing the values in each array to determine where they fit.

Merge Sort has an asymptotic running time of $O(n - nlog_2 n)$, this is because it splits the inital array in half into smaller problems, making it easier to produce a fully sorted list in less time.

## 3.4 Quick Sort

```
    // Split the array into two subarrays and return the pivot value
int split(std::string arr[], int low, int high, int& comparisonCount) {
    // Choose the pivot as the middle element
    int middle = low + (high - low) / 2;
    std::string pivot = arr[middle];

    // Move the pivot element to the end
    std::swap(arr[middle], arr[high]);

    int i = low - 1; // Index of the smaller element

    for (int j = low; j < high; j++) {
        comparisonCount++; // Count each comparison
        if (arr[j] <= pivot) {
            i++;
            std::swap(arr[i], arr[j]);
        }
    }

    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}
```

Listing 9: Splitting the Arrays

```
    // Quick sort function with comparison counter
void quickSort(std::string arr[], int low, int high, int& comparisonCount) {
    if (low < high) {
        int pivotIndex = split(arr, low, high, comparisonCount);

        // Recursively sort the subarrays
        quickSort(arr, low, pivotIndex - 1, comparisonCount);
        quickSort(arr, pivotIndex + 1, high, comparisonCount);
    }
}
```

Listing 10: Quick Sort Algorithm

The Quick Sort algorithm also has two functions that work together in the same "divide and conquer" methodology. The Split function divides the shuffled array in half much like the Merge Sort algorithm, but instead uses a pivot value (determined by the middle element of the array) to compare. The quicksort function calls the split (Line 4) function to get the pivot value and split the arrays. The arrays are then recursively sorted (lines 7-8) by continuously calling the quicksort function until each sub-array only contains one element.

The Quick Sort algorithm's asymptotic running time is $O(nlogn)$. This is because it shares the same divide and conquer method and is able to break down larger data sets into smaller pieces, but with the help of a pivot value which can help further.

## 3.5 Results Table

| Sorting Algorithm | Comparisons | Asymptotic Running Time |
|---|---|---|
| Selection Sort | 221445 | $On^2$ |
| Insertion Sort | 113,441 | $On^2$ |
| Merge Sort | 5438 | $O(n - nlog_2n)$ |
| Quick Sort | 6476 | $nlogn$ |

Number of Comparisons for Insertion, Merge, and Quick sorts calculated by getting the avg from running the code 10x.