Assignment One

Alex Tocci

Mason.Tocci1@Marist.edu

October 4, 2023

1 Stack and Queue

1.1 Creating Node Class

Listing 1: Node Class

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  // Include for std::random_device
5  #include <random>
6
7  using namespace std;
8
9  // Define a Node class for the stack and queue
10 class Node {
11 public:
12     char data;
13     Node* next;
14
15     Node(char value) : data(value), next(nullptr) {}
16 };
```

Lines 1-5: These lines include the necessary C++ standard libraries for our program to run.
Line 7: This allows us to use standard C++ functions without needing to prefix them with 'std::'.
Lines 10-13: Define a C++ class called 'Node', containing two member variables, 'char data' and 'Node* next'. 'Char data' stores the data associated with the node. 'Node* next' points to the next node in the linked structure or is set to nullptr if there is no next node.
Line 15: This is a constructor for the Node class. It initializes a Node object with the given character value 'value' and sets the next pointer to 'nullptr'.

1.2 Creating Stack Class

Listing 2: Stack Class

```
17 // Create a stack using the Node class
18 class Stack {
19 private:
20     // Point to the top of the stack
21     Node* top;
22 
23 public:
24     Stack() : top(nullptr) {}
25 
26     // Push a value onto the stack
27     void push(char value) {
28         Node* newNode = new Node(value);
29         newNode -> next = top;
30         top = newNode;
31     }
32 
33     // Pop the top value from the stack
34     char pop() {
35         if (isEmpty()) {
36             cerr << "Stack is empty." << endl;
37             return -1;
38         }
39         char value = top->data;
40         Node* temp = top;
41         top = top -> next;
42         delete temp;
43         return value;
44     }
45 
46     // Check if the stack is empty
47     bool isEmpty() {
48         return top == nullptr;
49     }
50 };
```

Lines 18-21: Define a class called 'Stack', containing a 'Node* top' member variable. This top pointer is used to keep track of the top element in the stack.
Line 24: This is the constructor of the Stack class. It initializes the top pointer to 'nullptr', indicating an empty stack.
Lines 27- 31: This function is used to push (add) a character value onto the stack. It creates a new Node with the given character value. It sets the 'next' pointer of the new 'Node' to point to the current top of the stack, linking the new Node to the previous top element. It updates the 'top' pointer to point to the new 'Node', making it the new top of the stack.
Lines 34-44: This function is used to pop (remove) the top character value from the stack. It checks if the stack is empty by calling the 'isEmpty' method. If the stack is not empty, it retrieves the character value stored in the current top Node, creates a temporary pointer (temp) to the current top Node to avoid losing reference to it, updates the 'top' pointer to point to the next Node in the stack, removing the current top Node, deletes the previous top Node to free the memory it occupied, finally, it returns the popped character value.
Lines 47- 49: This function is used to check if the stack is empty. It returns 'true' if the 'top' pointer is 'nullptr', indicating an empty stack, and 'false' otherwise.

1.3 Creating Queue Class

Listing 3: Queue Class

```
51 // Create a queue using the Node class
52 class Queue {
53 private:
54     Node* front;
55     Node* back;
56
57 public:
58     Queue() : front(nullptr), back(nullptr) {}
59
60     // Enqueue a value into the queue
61     void enqueue(char value) {
62         Node* newNode = new Node(value);
63         if (isEmpty()) {
64             front = back = newNode;
65         }
66         else {
67             back -> next = newNode;
68             back = newNode;
69         }
70     }
71
72     // Dequeue a value from the queue
73     char dequeue() {
74         if (isEmpty()) {
75             cerr << "Queue is empty." << endl;
76             return -1;
77         }
78         char value = front->data;
79         Node* temp = front;
80         front = front -> next;
81         delete temp;
82         return value;
83     }
84
85     // Check if the queue is empty
86     bool isEmpty() {
87         return front == nullptr;
88     }
89 };
```

Lines 52-55: Define a class called 'Queue', containing two member varables 'Node* front' and 'Node* back'. The front pointer is used to keep track of the front element in the queue. The back pointer is used to keep track of the back element in the queue.
Line 58: This is the constructor of the Queue class. It initializes both 'front' and 'back' pointers to 'nullptr', indicating an empty queue.
Lines 61-70: This function is used to enqueue (add) a character value into the queue. It creates a new Node with the given character value. If the queue is empty, it sets both 'front' and 'back' pointers to point to the new 'Node'. This makes the new 'Node' both the front and back of the queue. If the queue is not empty, it adds the new 'Node' to the back of the queue by updating the 'next' pointer of the current back element to

point to the new 'Node', and then updates the 'back' pointer to the new 'Node'.
Lines 73-82: This function is used to dequeue (remove) the front character value from the queue. It checks if the queue is empty by calling the 'isEmpty' method. If the queue is not empty, retrieves the character value stored in the current front Node, creates a temporary pointer (temp) to the current front Node to avoid losing reference to it, updates the 'front' pointer to point to the next Node in the queue, removing the current front Node, deletes the previous front Node to free the memory it occupied, finally, it returns the dequeued character value.
Lines 86-89: This function is used to check if the queue is empty. It returns 'true' if the 'front' pointer is 'nullptr', indicating an empty queue, and 'false' otherwise.

1.4 Check for palindrome

Listing 4: Check for Palindrome

```
90  // Check if a string is a palindrome (ignoring spaces and capitalization)
91  bool isPalindrome(const string& str) {
92      Stack stack;
93      Queue queue;
94
95      // Push characters to the stack and queue (ignoring spaces and converting to lowercase)
96      for (char i : str) {
97          if (!isspace(i)) {
98              stack.push(tolower(i));
99              queue.enqueue(tolower(i));
100         }
101     }
102
103     // Compare characters from the stack and queue
104     while (!stack.isEmpty() && !queue.isEmpty()) {
105         if (stack.pop() != queue.dequeue()) {
106             // Not a palindrome
107             return false;
108         }
109     }
110
111     return true;
112 }
```

Lines 91-93: Two data structures are created: a 'Stack' object named 'stack' and a 'Queue' object named 'queue'.
Lines 96-101: Check if the current character 'i' is not a space character using the 'isspace' function. If 'i' is not a space, it converts the character 'i' to lowercase using the 'tolower' function, pushes the lowercase character onto the 'stack' using the 'push' method of the 'Stack' object, and enqueues the lowercase character into the 'queue' using the 'enqueue' method of the 'Queue' object.
Lines 104-109: This while loop continues as long as both the 'stack' and the 'queue' are not empty. It compares the characters at the front of the 'stack' (using 'stack.pop()') and the front of the 'queue' (using 'queue.dequeue()'). If the characters are not equal, it means that the input string is not a palindrome, and the function returns 'false'.

1.5 Shuffle

Listing 5: Shuffle

```
113 // Shuffle the array using the Knuth shuffle algorithm with std::random_device seeding
114 void shuffle(string* arr, int size) {
115     random_device rd;
116     srand(rd());
117
118     for (int i = size - 1; i > 0; --i) {
119         // Generate a random index between 0 and i
120         int j = rand() % (i + 1);
121         // Swap elements at i and j
122         swap(arr[i], arr[j]);
123     }
124 }
```

Lines 114-116: Declare a random device named 'rd'. Random device is used to generate random numbers. Then, we call 'srand(rd())'. 'srand' is a function that initializes the random number generator with a seed value.
Lines 118-124: This 'for' loop is used to shuffle the elements of the input array 'arr' using the Knuth shuffle. It starts with 'i' initialized to 'size - 1', which is the index of the last element in the array. The loop continues as long as 'i' is greater than 0. In each iteration of the loop, it generates a random index 'j' between 0 and 'i' using the 'rand()' function. Then, it swaps the elements 'i' and 'j' in the array 'arr' using the 'swap' function.

2 Sorting

2.1 Selection Sort

Listing 6: Selection Sort

```
125 // Selection sort
126 void selectionSort(string* arr, int size, int& comparisons) {
127     for (int i = 0; i < size - 1; ++i) {
128         int index = i;
129         for (int j = i + 1; j < size; ++j) {
130             // Convert both strings to lowercase for comparison
131             for (char& ch : arr[j]) {
132                 ch = tolower(ch);
133             }
134             for (char& ch : arr[index]) {
135                 ch = tolower(ch);
136             }
137             comparisons++;
138             // Compare the lowercase strings
139             if (arr[j] < arr[index]) {
140                 index = j;
141             }
142         }
143         // If a smaller element was found, swap it with the current element
144         if (index != i) {
145             swap(arr[i], arr[index]);
146         }
147     }
148 }
```

Lines 126-128: This for loop iterates over the elements of the array from the beginning to the second-to-last element ('size - 1'). The purpose of this loop is to select the minimum element in the unsorted portion of the array and place it in its correct position.
Lines 129-142: This for loop iterates from 'i + 1' to 'size - 1', comparing each element in the unsorted portion of the array with the element at index 'i'. It converts two strings (the one at index 'j' and the one at index 'index') to lowercase using a loop to make the comparison case-insensitive. Then, 'comparisons' is incremented to keep track of the number of comparisons made. Finally, it compares the lowercase strings using the '<' operator. If 'arr[j]' is less than 'arr[index]', it means that the element at index 'j' is smaller, so 'index' is updated to 'j'.
Lines 144-146: After the inner loop completes, the 'index' variable contains the index of the smallest element in the unsorted portion of the array. If 'index' is not equal to 'i', it means that a smaller element was found in the unsorted portion of the array. In this case, the elements 'i' and 'index' are swapped using the swap function, placing the smallest element in its correct position in the sorted portion of the array.

2.2 Insertion Sort

Listing 7: Insertion Sort

```
// Insertion sort
void insertionSort(string* arr, int size, int& comparisons) {
    for (int i = 1; i < size; ++i) {
        int j = i;
```

```
153             while (j > 0) {
154                 // Convert both strings to lowercase for comparison
155                 for (char& ch : arr[j]) {
156                     ch = tolower(ch);
157                 }
158                 for (char& ch : arr[j - 1]) {
159                     ch = tolower(ch);
160                 }
161                 comparisons++;
162                 // Compare the lowercase strings
163                 if (arr[j] < arr[j - 1]) {
164                     swap(arr[j], arr[j - 1]);
165                     --j;
166                 }
167                 else {
168                     break;
169                 }
170             }
171         }
172 }
```

Lines 150-152: This for loop starts with 'i' initialized to 1 and iterates over the elements of the array from the second element to the last element. The purpose of this loop is to insert each element into its correct position among the previously sorted elements.
Lines 153-160: This while loop starts with 'j' initialized to 'i', and it continues as long as 'j' is greater than 0. It converts two strings (the one at index 'j' and the one at index 'j - 1') to lowercase using a loop to make the comparison case-insensitive.
Lines 161-166: Then, 'comparisons' is incremented to keep track of the number of comparisons made. Next, It compares the lowercase strings using the '<' operator. If 'arr[j]' is less than 'arr[j - 1]', it means that the element at index 'j' is smaller than the element at index 'j - 1'. In this case, it swaps these two elements using the 'swap' function and decrements 'j' to continue comparing and possibly swapping with previous elements.
Lines 167-169: If the comparison (arr[j] < arr[j - 1]) is false, it breaks out of the inner loop. This break occurs when the element at index j is not smaller than the previous element.

2.3 Merge Sort

2.3.1 Merge Function

Listing 8: Merge Function

```
// Function to merge two sorted subarrays
void merge(string* arr, int left, int mid, int right, int& comparisons) {
    // Calculate the sizes of the subarrays
    int sizeLeft = mid - left + 1;
    int sizeRight = right - mid;

    // Create temporary arrays for the subarrays
    string* leftArray = new string[sizeLeft];
    string* rightArray = new string[sizeRight];

    // Copy data to temporary arrays
    for (int i = 0; i < sizeLeft; ++i) {
```

```
        leftArray[i] = arr[left + i];
    }
    for (int i = 0; i < sizeRight; ++i) {
        rightArray[i] = arr[mid + 1 + i];
    }

    // Merge the two subarrays back into array
    // Index for the left subarray
    int i = 0;
    // Index for the right subarray
    int j = 0;
    // Index for the merged array
    int k = left;

    while (i < sizeLeft && j < sizeRight) {
        // Convert characters to lowercase for case-insensitive comparison
        for (char& ch : leftArray[i]) {
            ch = tolower(ch);
        }
        for (char& ch : rightArray[j]) {
            ch = tolower(ch);
        }

        comparisons++;
        // Compare and merge based on lowercase strings
        if (leftArray[i] <= rightArray[j]) {
            arr[k] = leftArray[i];
            ++i;
        }
        else {
            arr[k] = rightArray[j];
            ++j;
        }
        ++k;
    }

    // Copy the remaining elements of leftArray[], if any
    while (i < sizeLeft) {
        arr[k] = leftArray[i];
        ++i;
        ++k;
    }

    // Copy the remaining elements of rightArray[], if any
    while (j < sizeRight) {
        arr[k] = rightArray[j];
        ++j;
        ++k;
    }

    // Delete temporary arrays
    delete[] leftArray;
```

```
237         delete[] rightArray;
238 }
```

Lines 174-177: This function, merge, is a part of the merge sort algorithm and is responsible for merging two sorted subarrays into a single sorted array. First, the sizes of the two subarrays are calculated. 'int sizeLeft = mid - left + 1' calculates the size of the left subarray. 'int sizeRight = right - mid' calculates the size of the right subarray.
Lines 180-181: Create Temporary Arrays: 'string* leftArray = new string[sizeLeft]' creates a temporary array to hold the left subarray. 'string* rightArray = new string[sizeRight]' creates a temporary array to hold the right subarray.
Lines 184-189: Two 'for' loops are used to copy the data from the original array 'arr' into the 'leftArray' and 'rightArray'.
Lines 193-197: Merge the two subarrays back into the original array 'arr'. Three index variables are used: 'i' for the left subarray, 'j' for the right subarray, 'k' for the merged array. A 'while' loop is used to compare and merge elements from both subarrays until at least one of the subarrays is fully merged.
Lines 201-208: Characters in both 'leftArray[i]' and 'rightArray[j]' are converted to lowercase for case-insensitive comparison. Then, the 'comparisons' counter is incremented to keep track of the number of comparisons made during the merge.
Lines 210-219: Elements are compared based on their lowercase strings. The smaller element between 'leftArray[i]' and 'rightArray[j]' is copied into the merged array 'arr[k]', and the corresponding index ('i' or 'j') is incremented. The index 'k' is also incremented to indicate the position of the next element in the merged array.
Lines 222-233: The two 'while' loops copy any remaining elements from 'leftArray' and 'rightArray' into the merged array 'arr'.
Lines 236-238: Finally, the dynamically allocated memory for 'leftArray' and 'rightArray' is freed using the 'delete[]' operator.

2.3.2 mergeSort Function

Listing 9: mergeSort Function

```
239 // Merge sort
240 void mergeSort(string* arr, int left, int right, int& comparisons) {
241     if (left < right) {
242         // Find the middle point
243         int mid = left + (right - left) / 2;
244 
245         // Recursively sort the first and second halves
246         mergeSort(arr, left, mid, comparisons);
247         mergeSort(arr, mid + 1, right, comparisons);
248 
249         // Merge the sorted halves
250         merge(arr, left, mid, right, comparisons);
251     }
252 }
```

Lines 240-241: The function begins with a conditional check. It checks if 'left' (the left index of the current subarray) is less than 'right' (the right index of the current subarray).
Line 243: If the condition is met, the function proceeds to find the middle point of the current subarray. It calculates the midpoint 'mid' by taking the average of 'left' and 'right'. This determines the point at which the subarray is divided into two halves.
Lines 246-247: The function recursively calls itself to sort the two halves of the current subarray. The first

recursive call sorts the left half of the subarray from index 'left' to 'mid'. The second recursive call sorts the right half of the subarray from index 'mid + 1' to 'right'. These recursive calls split the subarrays into smaller and smaller pieces until they reach the base case, where they contain only one element.
Line 250: Once the left and right halves of the subarray are sorted, the merge function is called to merge them back together into a single sorted subarray.

2.4 Quick Sort

2.4.1 Partition Function

Listing 10: Partition Function

```
253 // Partition function for quick sort
254 int partition(string* arr, int low, int high, int& comparisons) {
255     // Choose the rightmost element as the pivot
256     string pivot = arr[high];
257     // Index of the smaller element
258     int i = (low - 1);
259 
260     for (int j = low; j <= high - 1; ++j) {
261         // Convert both strings to lowercase for comparison
262         for (char& ch : arr[j]) {
263             ch = tolower(ch);
264         }
265         for (char& ch : pivot) {
266             ch = tolower(ch);
267         }
268 
269         comparisons++;
270         // Compare the lowercase strings
271         if (arr[j] < pivot) {
272             ++i;
273             swap(arr[i], arr[j]);
274         }
275     }
276 
277     // Swap the pivot element
278     swap(arr[i + 1], arr[high]);
279     return (i + 1);
280 }
```

Lines 256-258: Thish function starts by selecting the rightmost element in the given subarray (specified by high) as the pivot. In this implementation, the pivot is stored in the pivot variable. 'i' is initialized to 'low - 1'. It will be used to keep track of the index of the smaller element during the partitioning process.
Lines 260-274: The 'for' loop iterates through the elements of the subarray from 'low' to 'high - 1'. This loop compares each element with the pivot and rearranges them accordingly. Inside the loop, both the current element ('arr[j]') and the pivot are converted to lowercase characters to ensure a case-insensitive comparison. For each element, the function increments the 'comparisons' counter to keep track of the number of comparisons made during the partition, and compares the lowercase strings of the element at index 'j' and the pivot using the '<' operator. If the element is less than the pivot, it increments 'i' to mark the position of the smaller element and swaps the element at index 'i' with the element at index 'j'.
Lines 278-280: The pivot element is swapped with the element at index 'i + 1', placing the pivot in its

correct sorted position. The function returns the index '(i + 1)' as it represents the correct position of the pivot element in the sorted array.

2.4.2 quickSort Function

Listing 11: quickSort Function

```
281 // Quick sort
282 void quickSort(string* arr, int low, int high, int& comparisons) {
283     if (low < high) {
284         // Find the pivot element such that
285         // element smaller than pivot are on the left and
286         // elements greater than pivot are on the right
287         int pivotIndex = partition(arr, low, high, comparisons);
288 
289         // Recursively sort the subarrays
290         quickSort(arr, low, pivotIndex - 1, comparisons);
291         quickSort(arr, pivotIndex + 1, high, comparisons);
292     }
293 }
```

Line 283: This function starts with a conditional check. It verifies whether the 'low' index is less than the 'high' index.
Line 287: If the condition is met, the function proceeds to find a pivot element and partition the subarray. The partition function is called to do this.
Lines 290-293: After partitioning the subarray, the 'quickSort' function is called recursively twice to sort the left and right subarrays. The first recursive call ('quickSort(arr, low, pivotIndex - 1, comparisons)') sorts the left subarray (elements less than the pivot) from index 'low' to 'pivotIndex - 1'. The second recursive call ('quickSort(arr, pivotIndex + 1, high, comparisons)') sorts the right subarray (elements greater than the pivot) from index 'pivotIndex + 1' to 'high'. These recursive calls continue until all subarrays contain zero or one element, which is the base case for the recursion.

3 Main Function

3.1 Test Stack and Queue

Listing 12: Test Stack and Queue

```
294 int main() {
295     // Test the stack
296     Stack testStack;
297     testStack.push('A');
298     testStack.push('B');
299     testStack.push('C');
300 
301     cout << "Stack␣contents:␣";
302     while (!testStack.isEmpty()) {
303         cout << testStack.pop() << "␣";
304     }
305     cout << endl;
306 
307     // Test the queue
308     Queue testQueue;
309     testQueue.enqueue('A');
310     testQueue.enqueue('B');
311     testQueue.enqueue('C');
312 
313     cout << "Queue␣contents:␣";
314     while (!testQueue.isEmpty()) {
315         cout << testQueue.dequeue() << "␣";
316     }
317     cout << endl;
```

Lines 296-299: Create an instance of a stack and push three characters onto it. The elements are added to the top of the stack in the order they are pushed.
Lines 301-305: The 'while' loop will continue as long as the stack is not empty. Within the loop, 'testStack.pop()' pops the top element from the stack and prints it to the standard output. The loop continues until the stack is empty, and all elements ('C', 'B', 'A') are popped and printed.
Lines 309-311: Create an instance of a queue and enqueue three characters onto it. The elements are added to the back of the queue in the order they are enqueued.
Lines 313-317: The 'while' loop will continue as long as the queue is not empty. Within the loop, 'testQueue.dequeue()' dequeues the front element from the queue and prints it to the standard output. The loop continues until the queue is empty, and all elements ('A', 'B', 'C') are dequeued and printed.

3.2 Read 'magicitems.txt' and store items into an array

Listing 13: Store Items in Array

```
318 // Read all lines of magicitems.txt and put them in an array
319 // Open the magicitems.txt file
320 ifstream file("magicitems.txt");
321 
322 // Handle failure
323 if (!file)
324 {
```

```
325     cerr << "Failed to open magicitems.txt" << endl;
326     return 1;
327 }
328
329 // Count the number of lines in the file
330 int magicItemsSize = 0;
331 string line;
332 while (getline(file, line))
333 {
334     magicItemsSize++;
335 }
336
337 // Close and reopen the file to read from the beginning
338 file.close();
339 file.open("magicitems.txt");
340
341 // Create a dynamically allocated array
342 string* magicItemsArray = new string[magicItemsSize];
343
344 // Read the file line by line and store each line in the array
345 int index = 0;
346 while (getline(file, line))
347 {
348     magicItemsArray[index] = line;
349     index++;
350 }
351
352 // Close the file
353 file.close();
```

Line 320: Open the file 'magicitems.txt' for reading.
Lines 323-327: Check if the file was successfully opened. If the file cannot be opened, it prints an error message to the standard error stream ('cerr') and exits the program.
Lines 330-335: Read the file line by line using 'getline' and count the number of lines in the file. Increment the 'magicItemsSize' variable for each line encountered.
Lines 338-339: Close and reopen the file to read from the beginning.
Line 342: Dynamically allocate an array of strings named 'magicItemsArray' with a size equal to the number of lines counted earlier ('magicItemsSize').
Lines 345-350: Read the file line by line and store each line in the array.
Line 353: Close the file.

3.3 Check for Palindromes

Listing 14: Check for Palindromes

```
354 cout << "\nPalindromes: " << endl;
355 // Check each line in the array for palindromes and print it if it is
356 for (int i = 0; i < magicItemsSize; ++i) {
357     if (isPalindrome(magicItemsArray[i])) {
358         cout << magicItemsArray[i] << endl;
359     }
360 }
```

Lines 354-360: The for loop iterates over each line in the 'magicItemsArray'. The loop index 'i' ranges from 0 to 'magicItemsSize - 1'. Inside the loop, 'isPalindrome(magicItemsArray[i])' is used to check if the current line in the array ('magicItemsArray[i]') is a palindrome. If 'isPalindrome' returns 'true', it means that the current line is a palindrome and will be printed to the standard output.

3.4 Selection Sort

Listing 15: Selection Sort

```
361 // Shuffle before sorting
362 shuffle(magicItemsArray, magicItemsSize);
363
364 int comparisonsSelectionSort = 0;
365 // Sort magicItemsArray using selection sort
366 selectionSort(magicItemsArray, magicItemsSize, comparisonsSelectionSort);
367
368 cout << "\nSorted␣Magic␣Items␣(Selection␣Sort):" << endl;
369 for (int i = 0; i < magicItemsSize; ++i) {
370     cout << magicItemsArray[i] << endl;
371 }
372 cout << "Comparisons␣in␣Selection␣Sort:␣" << comparisonsSelectionSort << endl;
```

Line 362: This line calls a function named 'shuffle' to shuffle the elements in the 'magicItemsArray'.
Lines 365-366: Initialize an integer variable named 'comparisonsSelectionSort' to zero. Sort the 'magicItemsArray' using the selection sort algorithm. The 'comparisonsSelectionSort' variable is passed as a reference to the function, allowing it to count the number of comparisons made during the sorting process.
Lines 368-371: Use a 'for' loop to iterate through the sorted 'magicItemsArray' and print each element to the standard output.
Line 372: Print the number of comparisons made during the selection sort process.

3.5 Insertion Sort

Listing 16: Insertion Sort

```
373 // Shuffle before sorting again
374 shuffle(magicItemsArray, magicItemsSize);
375
376 int comparisonsInsertionSort = 0;
377 // Sort magicItemsArray using selection sort
378 insertionSort(magicItemsArray, magicItemsSize, comparisonsInsertionSort);
379
380 cout << "\nSorted␣Magic␣Items␣(Insertion␣Sort):" << endl;
381 for (int i = 0; i < magicItemsSize; ++i) {
382     cout << magicItemsArray[i] << endl;
383 }
384 cout << "Comparisons␣in␣Insertion␣Sort:␣" << comparisonsInsertionSort << endl;
```

Line 374: Shuffle the elements in 'magicItemsArray'.
Lines 376-378: Initialize an integer variable named 'comparisonsInsertionSort' to zero. Sort the 'magicItemsArray' using the insertion sort algorithm. The 'comparisonsInsertionSort' variable is passed as a reference to the function, allowing it to count the number of comparisons made during the sorting process.
Lines 380-383: Use a 'for' loop to iterate through the sorted 'magicItemsArray' and print each element to

the standard output.
Line 384: Print the number of comparisons made during the insertion sort process.

3.6 Merge Sort

Listing 17: Merge Sort

```
385 // Shuffle before sorting again
386 shuffle(magicItemsArray, magicItemsSize);
387
388 int comparisonsMergeSort = 0;
389 // Sort the strings using merge sort in a case-insensitive manner
390 mergeSort(magicItemsArray, 0, magicItemsSize - 1, comparisonsMergeSort);
391
392 cout << "\nSorted Magic Items (Merge Sort):" << endl;
393 for (int i = 0; i < magicItemsSize; ++i) {
394     cout << magicItemsArray[i] << endl;
395 }
396 cout << "Comparisons in Merge Sort: " << comparisonsMergeSort << endl;
```

Line 386: Shuffle the elements in 'magicItemsArray'.
Lines 388-390: Initialize an integer variable named 'comparisonsMergeSort' to zero. Sort the 'magicItemsArray' using the merge sort algorithm. The 'comparisonsMergeSort' variable is passed as a reference to the function, allowing it to count the number of comparisons made during the sorting process.
Lines 392-395: Use a 'for' loop to iterate through the sorted 'magicItemsArray' and print each element to the standard output.
Line 396: Print the number of comparisons made during the merge sort process.

3.7 Quick Sort

Listing 18: Quick Sort

```
    // Shuffle before sorting again
    shuffle(magicItemsArray, magicItemsSize);

    // Sort magicItemsArray using quick sort
    int comparisonsQuickSort = 0; // Initialize comparisons counter
    quickSort(magicItemsArray, 0, magicItemsSize - 1, comparisonsQuickSort);

    cout << "\nSorted Magic Items (Quick Sort):" << endl;
    for (int i = 0; i < magicItemsSize; ++i) {
        cout << magicItemsArray[i] << endl;
    }

    cout << "Comparisons in Quick Sort: " << comparisonsQuickSort << endl;


    // Delete dynamically allocated memory
    delete[] magicItemsArray;

    return 0;
}
```

Line 398: Shuffle the elements in 'magicItemsArray'.
Lines 400-402: Initialize an integer variable named 'comparisonsQuickSort' to zero. Sort the 'magicItemsArray' using the quick sort algorithm. The 'comparisonsQuickSort' variable is passed as a reference to the function, allowing it to count the number of comparisons made during the sorting process.
Lines 404-407: Use a 'for' loop to iterate through the sorted 'magicItemsArray' and print each element to the standard output.
Line 409: Print the number of comparisons made during the quick sort process.
Line 413: Delete dynamically allocated memory.

4 Results

Table 4.1: Number of Comparisons

	1	2	3	4	5	Avg
Selection Sort	221445	221445	221445	221445	221445	221445
Insertion Sort	111238	112774	107069	111565	109727	110474.6
Merge Sort	5435	5417	5440	5405	5423	5424
Quick Sort	7052	7407	6809	7234	6815	7063.4

4.1 Selection Sort

Selection sort has a time complexity of $O(n^2)$ in the worst, average, and best cases. In this case, we always get 221,445 comparisons because the size of the array is fixed at 666, and selection sort performs the same number of comparisons regardless of the initial order of the input. In the code, the outer loop runs for '(size - 1)' iterations, and the inner loop runs for '(size)' iterations in each iteration of the outer loop. This results in a total of $comparisons = (size - 1) * (size)/2$ comparisons, plugging in 666 for size we get 221,445 comparisons.

4.2 Insertion Sort

Insertion sort has a time complexity of $O(n^2)$ in its worst cases. In the code, the 'for' loop runs for '(size - 1)' iterations. In the code, the 'while' loop runs as long as the selected element ('arr[j]') is less than its previous element ('arr[j - 1]') when compared in lowercase; in the worst case this would be the size of the array, 666. This results in a worst case of $comparisons = (size - 1) * (size)/2$ comparisons, plugging in 666 for size we get 221,445 comparisons. To get the average we would divide the worst case by 2, making our average 110722.5.

4.3 Merge Sort

Merge sort has a time complexity of $O(nlogn)$ in its worst cases. In the code, the 'merge' function merges two sorted subarrays into a single sorted array. The time complexity of this function is $O(n)$, where n is the total number of elements in the two subarrays. The 'mergeSort' function recursively divides the input array into smaller subarrays until each subarray contains only one element. Then, it merges these subarrays back together in sorted order. The dividing step (the recursion) takes $O(logn)$ time because the array is repeatedly halved. The merging step takes $O(n)$ time per recursion. When adding up the time taken at each level of recursion, you get a total time complexity of $O(nlogn)$. Plugging in 666, our worst case is 6246.7.

4.4 Quick Sort

Quick sort has a time complexity of $O(nlogn)$ in its worst cases. In the code, the 'partitioning' function takes a pivot and arranges the elements such that those less than the pivot are on the left, and those greater are on the right. The average time complexity of this function is $O(n)$. In each recursive call to 'quickSort', the partition step splits the array into two subarrays. The size of these subarrays is not necessarily halved, but in the average case, they are expected to be balanced. The partitioning step takes $O(n)$ time per recursion. When adding up the time taken at each level of recursion, you get an average time complexity of $O(nlogn)$. Plugging in 666, our average case should be 6246.7.