Assignment Two

Alex Tocci

Mason.Tocci1@Marist.edu

October 26, 2023

1 Sorting

1.1 Merge Sort

1.1.1 Merge Function

Listing 1: Merge Function

```
#include <iostream>
#include <fstream>
#include <string>
#include <list>
// Include for std::random_device
#include <random>

using namespace std;

// 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];
```

```
26      }
27
28      // Merge the two subarrays back into array
29      // Index for the left subarray
30      int i = 0;
31      // Index for the right subarray
32      int j = 0;
33      // Index for the merged array
34      int k = left;
35
36      while (i < sizeLeft && j < sizeRight) {
37          // Convert characters to lowercase
38          for (char& ch : leftArray[i]) {
39              ch = tolower(ch);
40          }
41          for (char& ch : rightArray[j]) {
42              ch = tolower(ch);
43          }
44
45          comparisons++;
46          // Compare and merge based on lowercase strings
47          if (leftArray[i] <= rightArray[j]) {
48              arr[k] = leftArray[i];
49              ++i;
50          }
51          else {
52              arr[k] = rightArray[j];
53              ++j;
54          }
55          ++k;
56      }
57
58      // Copy the remaining elements of leftArray[], if any
59      while (i < sizeLeft) {
60          arr[k] = leftArray[i];
61          ++i;
62          ++k;
63      }
64
65      // Copy the remaining elements of rightArray[], if any
66      while (j < sizeRight) {
67          arr[k] = rightArray[j];
68          ++j;
69          ++k;
70      }
71
72      // Delete temporary arrays
73      delete[] leftArray;
74      delete[] rightArray;
75  }
```

Lines 1-6: First we must include the necessary C++ libraries for our program to run.

Line 8: This allows us to use standard C++ functions without the prefix 'std::'.
Line 11: This function, is a part of the merge sort algorithm and involves merging two sorted subarrays into a single sorted array.
Lines 13-14: Calculate the sizes of the two subarrays.
Lines 17-18: Create temporary arrays for the left and right subarray.
Lines 21-26: Two 'for' loops copy the data from the original array into the subarrays.
Lines 30-34: Merge the subarrays back into the original array. Initialize, three index variables: 'i' for the left subarray, 'j' for the right subarray, 'k' for the merged array.
Lines 36-43: Elements in the left and right subarrays are converted to lowercase.
Line 45: Increment 'comparisons' to keep track of the number of comparisons made.
Lines 47-56: Elements are compared based on their lowercase strings. The smaller element between the left subarray and right subarray is copied into the merged array. The index of the smaller element is incremented. Index 'k' is also incremented.
Lines 59-70: The two 'while' loops copy the remaining elements in the left and right subarrays to the merged array.
Lines 73-75: Delete the left and right subarrays.

1.1.2 mergeSort Function

Listing 2: mergeSort Function

```
76 // Merge sort
77 void mergeSort(string* arr, int left, int right, int& comparisons) {
78     if (left < right) {
79         // Find the middle point
80         int mid = left + (right - left) / 2;
81 
82         // Recursively sort the first and second halves
83         mergeSort(arr, left, mid, comparisons);
84         mergeSort(arr, mid + 1, right, comparisons);
85 
86         // Merge the sorted halves
87         merge(arr, left, mid, right, comparisons);
88     }
89 }
```

Lines 77-80: If the left index of the current subarray is less than the right index of the current subarray, find the middle point. Calculate the midpoint by taking the average of 'left' and 'right'. The midpoint is the place where the array will be split.
Lines 83-84: 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 and the second recursive call sorts the right half of the subarray. These recursive calls split the subarrays into smaller and smaller arrays until they contain only one element.
Line 87: 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 Searching

2.1 Linear Search

Listing 3: Linear Search

```
90 // Linear search
91 int linearSearch(string* arr, int size, string targetItem, int& linearComparisons) {
92     for (int i = 0; i < size; i++) {
93         linearComparisons++;
94         if (arr[i] == targetItem) {
95             cout << "Found " << targetItem << " at index " << i << endl;
96             // Return the index of the item
97             return i;
98         }
99     }
100    return -1;
101 }
```

Lines 91-92: Loop through the entire array, visiting each index.
Line 93: Increment 'linearComparisons' to keep track of the number of comparisons made.
Lines 94-99: If the value at index 'i' is equal to 'targetItem', print that the 'targetItem' was found and return the index 'i'.
Line 100: Return -1 if 'targetItem' was not found.

2.2 Binary Search

Listing 4: Binary Search

```
// Binary search
int binarySearch(string* arr, int size, string targetItem, int& binaryComparisons) {
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        string midItem = arr[mid];

        binaryComparisons++;

        if (midItem == targetItem) {
            // Return the index of the item
            cout << "Found " << targetItem << " at index " << mid << endl;
            return mid;
        }
        else if (midItem < targetItem) {
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }
```

```
126
127         return -1;
128 }
```

Lines 103-105: Initialize 'left' to 0 and 'right' to size -1.
Lines 107- 110: While left is less than or equal to right, calculate the mid point and set 'midItem' to the value at index 'arr[mid]'.
Line 112: Increment 'binaryComparisons' to keep track of the number of comparisons made.
Lines 114-118: If 'midItem' is equal to 'targetItem', print that the 'targetItem' was found and return the index 'mid'.
Lines 119-121: If 'midItem' is less than 'targetItem', update 'left' to 'mid + 1'.
Lines 122-125: If 'midItem' is greater than 'targetItem', update 'right' to 'mid - 1'.
Line 127: Return -1 if 'targetItem' was not found.

3 Hash Table

3.1 Make Hash Code

Listing 5: Make Hash Code

```
129 const int TABLE_SIZE = 250;
130
131 // Hash table, using a linked list for chaining
132 list<string> hashTable[TABLE_SIZE];
133
134 // Hash function
135 int makeHashCode(const string& str) {
136     string upperStr = str;
137     for (char& ch : upperStr) {
138         ch = toupper(ch);
139     }
140
141     int letterTotal = 0;
142     for (char ch : upperStr) {
143         letterTotal += static_cast<int>(ch);
144     }
145
146     int hashCode = (letterTotal * 1) % TABLE_SIZE;
147
148     return hashCode;
149 }
```

Line 129: Initialize the hash table size to 250.
Line 131: Create a hash table array of size 250. This array stores linked lists.
Lines 135-139: Initialize 'upperStr' equal to the string passed in. Convert all letters of 'upperStr' to uppercase.
Lines 141-144: Initialize 'letterTotal' to 0. Loop through the characters in 'upperStr', counting the number of letters.
Line 146: Set the hash code equal to the number of letters and ensure the hash code is within the range of the hash table size.
Line 148: Return the hash code.

3.2 Load Hash Table

Listing 6: Load Hash Table

```
150 // Load items into the hash table
151 void loadHashTable(string* magicItemsArray, int magicItemsSize) {
152     for (int i = 0; i < magicItemsSize; i++) {
153         int hashCode = makeHashCode(magicItemsArray[i]);
154         hashTable[hashCode].push_back(magicItemsArray[i]);
155     }
156 }
```

Lines 151-152: Loop through the entire array, visiting every index.
Line 153: Call 'makeHashCode' to get the hash code for the item at index 'i'.

Line 154: Push the item to the hash table, using the hash code as the index.

3.3 Retrieve Item

Listing 7: Retrieve Item

```
157 // Retrieve an item from the hash table
158 int retrieveItem(const string& item, int& hashComparisons) {
159     int hashCode = makeHashCode(item);
160     // count the get comparison
161     hashComparisons++;
162 
163     for (const string& listItem : hashTable[hashCode]) {
164         hashComparisons++;
165         if (listItem == item) {
166             cout << "Retrieved " << item << " from hash table." << endl;
167             return hashComparisons;
168         }
169     }
170 
171     cout << item << " not found with " << endl;
172     return hashComparisons;
173 }
```

Lines 158-159: Call 'makeHashCode' to get the hash code of the target item.
Line 161: Increment 'hashComparisons' to count the 'get' comparison.
Line 163: Loop through the linked list that corresponds with the hash code, visiting each index.
Line 164: Increment 'hashComparisons' to keep track of the number of comparisons made.
Lines 165-169: If the 'listItem' equals the target item, print that the item was found and return hash comparisons.
Lines 171-173: If the target item was not found, print that the item was not found and return hash comparisons.

4 Main Function

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

Listing 8: Store Items in Array

```
174 // Read all lines of magicitems.txt and put them in an array
175 // Open the magicitems.txt file
176 ifstream file("magicitems.txt");
177
178 // Handle failure
179 if (!file)
180 {
181     cerr << "Failed to open magicitems.txt" << endl;
182     return 1;
183 }
184
185 // Count the number of lines in the file
186 int magicItemsSize = 0;
187 string line;
188 while (getline(file, line))
189 {
190     magicItemsSize++;
191 }
192
193 // Close and reopen the file to read from the beginning
194 file.close();
195 file.open("magicitems.txt");
196
197 // Create a dynamically allocated array
198 string* magicItemsArray = new string[magicItemsSize];
199
200 // Read the file line by line and store each line in the array
201 int index = 0;
202 while (getline(file, line))
203 {
204     magicItemsArray[index] = line;
205     index++;
206 }
207
208 // Close the file
209 file.close();
```

Line 176: Open the file 'magicitems.txt' for reading.
Lines 179-183: Check if the file was successfully opened. If the file cannot be opened, print an error message and exit the program.
Lines 186-191: Read the file line by line and count the number of lines in the file. Increment the 'magicItemsSize' variable for each line read.
Lines 194-195: Close and reopen the file to read from the beginning.
Line 198: Create a dynamically allocated array of strings named 'magicItemsArray'. Set the size equal to 'magicItemsSize'.
Lines 201-206: Read the file line by line and store each line in the array.
Line 209: Close the file.

4.2 Merge Sort and Load Hash Table

Listing 9: Merge Sort and Load Hash Table

```
210 int comparisonsMergeSort = 0;
211 // Sort the strings using merge sort in a case-insensitive manner
212 mergeSort(magicItemsArray, 0, magicItemsSize - 1, comparisonsMergeSort);
213
214 // Load magic items into the hash table
215 loadHashTable(magicItemsArray, magicItemsSize);
```

Lines 210-212: Initialize an integer variable named 'comparisonsMergeSort' to zero. Sort the array using the merge sort algorithm. The 'comparisonsMergeSort' variable is passed into the function, allowing the number of comparisons to be counted.
Line 215: Load the hash table using the 'loadHashTable' function, passing in 'magicItemsArray' and 'magicItemsSize'.

4.3 Conduct Searching

Listing 10: Conduct Searching

```
// Set up randomizer
random_device rd;
srand(rd());

// Search
int totalComparisonsLinear = 0;
int totalComparisonsBinary = 0;
int totalComparisonsHash = 0;

for (int i = 0; i < 42; i++) {
    // Generate a random index between 0 and the size of the array
    int index = rand() % (magicItemsSize);

    int linearComparisons = 0;
    int binaryComparisons = 0;
    int hashComparisons = 0;

    string targetItem = magicItemsArray[index];

    linearSearch(magicItemsArray, magicItemsSize, targetItem, linearComparisons);
    binarySearch(magicItemsArray, magicItemsSize, targetItem, binaryComparisons);
    retrieveItem(targetItem, hashComparisons);

    totalComparisonsLinear += linearComparisons;
    totalComparisonsBinary += binaryComparisons;
    totalComparisonsHash += hashComparisons;

    cout << "Search " << (i + 1) << " linear comparisons: " << linearComparisons << endl;
    cout << "Search " << (i + 1) << " binary comparisons: " << binaryComparisons << endl;
    cout << "Search " << (i + 1) << " hash table comparisons: " << hashComparisons << "\n"
}
```

Lines 217-223: Initialize the randomizer and total comparisons for each searching method.
Line 225: Loop through the following code 42 times.
Line 227: Get a random index between 0 and the size of 'magicItemsArray'.
Lines 229-231: Initialize the comparisons for each search method.
Line 233: Initialize the target item and set it to the value located at the random index 'index'.
Lines 235-237: Search for the target item using each searching method, calling each function. The comparisons will be updated.
Lines 239-241: Add the number of comparisons to the corresponding total comparisons.
Lines 243-246: Print the result of each sorting method.

4.4 Print Average Comparisons

Listing 11: Print Average Comparisons

```
247     // Convert totalComparisons to a double and divide by 42 for average
248     double averageComparisonsLinear = static_cast<double>(totalComparisonsLinear) / 42;
249     printf("Average comparisons for linear search: %.2f\n", averageComparisonsLinear);
250
251     double averageComparisonsBinary = static_cast<double>(totalComparisonsBinary) / 42;
252     printf("Average comparisons for binary search: %.2f\n", averageComparisonsBinary);
253
254     double averageComparisonsHash = static_cast<double>(totalComparisonsHash) / 42;
255     printf("Average comparisons for hash retrieval: %.2f\n", averageComparisonsHash);
256
257     // Delete dynamically allocated memory
258     delete[] magicItemsArray;
259
260         return 0;
261 }
```

Lines 248-255: Convert each total comparison to a double and divide by the number of searches conducted, 42, to get the average. Print average comparisons to two decimal places.
Line 258: Delete dynamically allocated memory.

5 Results

Table 5.1: Number of Comparisons

	1	2	3	4	5	Avg of Averages
Linear Search	370.83	327.74	294.12	319.64	355.67	333.60
Binary Search	8.50	8.48	8.26	8.52	8.17	8.39
Hashing	3.40	3.38	3.21	3.14	3.33	3.29

5.1 Linear Search

Linear search has a time complexity of $O(n)$. This is because we loop through the array, visiting each index and checking if it is the target item. In the code, the for loop iterates 'size' times and returns when the target item is found. In the worst case (target item is the last item in the array) we would get 'size' (n) comparisons.

5.2 Binary Search

Binary Search has a time complexity of $O(logn)$. This is because we loop through the array, repeatedly cutting it in half. In the code, the while loop runs until 'mid' equals the target item. In the worst case (target item is the last midpoint) we would get logn comparisons.

5.3 Hashing

Hashing has a time complexity of $O(n)$. This is because we loop through a linked list (at an index specified by a hash code) visiting each index and checking if it is the target item. In the code, the ranged for loop runs until the target item is found. In the worst case (all items in magicItemsArray have the same amount of letters and the target item is the last item in the linked list) we would get n comparisons. However, we do expect our hash table to distribute fairly evenly, so the time complexity could be $O(1 + alpha)$, where alpha is the average number of items stored in each hash code index.