Assignment Two

Ethan Morton

Ethan.Morton1@Marist.edu

October 15, 2024

1 Setting Up the Scenario

1.1 Reading from magicitems.txt

The lines in magicitems.txt are loaded into the lines array at runtime. NUM_LINES is defined as 666.

main.cpp

```
83  int main() {
84      //Read lines from magicitems.txt
85      std::ifstream file("../magicitems.txt");
86      if (!file.is_open()) {
87          printf("Unable to open file\n");
88          return 1;
89      }
90      string lines[NUM_LINES];
91      string line;
92      int index = 0;
93      while (std::getline(file, line)) {
94          lines[index] = line;
95          index++;
96      }
97      file.close();
98
99      //...
100 }
```

1.2 Picking 42 random items

The lines array is first shuffled using the Knuth shuffle algorithm.

main.cpp

```
71 void knuthShuffle(string* arr, const int size) {
72     std::random_device rd;
73     std::mt19937 gen(rd());
74     //count backwards from the array
75     for (int i = size - 1; i > 0; i--) {
76         //pick a random index in the array and swap the element with the
           ↪  element at i
77         std::uniform_int_distribution dist(0, i);
78         const int ii = dist(gen);
79         swap(arr[i], arr[ii]);
80     }
81 }
```

The first 42 items of the shuffled lines array are copied into a new array called rand42. NUM_RAND is defined as 42.

main.cpp

```
96 int main() {
97     //...
98 
99     //Get 42 random items
100    string rand42[NUM_RAND];
101    knuthShuffle(lines, NUM_LINES);
102    for (int i = 0; i < NUM_RAND; i++) {
103        rand42[i] = lines[i];
104    }
105    mergeSort(lines, NUM_LINES);
106 
107    //...
108 }
```

Then the lines array is sorted via the merge sort algorithm.

main.cpp

```
14 int mergeSortHelper(string* arr, const int left, const int right) {
15     if (left >= right) {
16         return 0;
17     }
18     int comparisons = 0;
19     const int mid = left + (right - left) / 2;
20     //merge sort on left side of array
21     comparisons += mergeSortHelper(arr, left, mid);
22     //merge sort on right side of array
23     comparisons += mergeSortHelper(arr, mid + 1, right);
24     //divide into left and right temp arrays
```

```
25      const int leftArrLen = mid - left + 1;
26      const int rightArrLen = right - mid;
27      std::vector<string> leftArr = std::vector<string>(leftArrLen);
28      std::vector<string> rightArr = std::vector<string>(rightArrLen);
29      // string leftArr[leftArrLen];//the stack won't be happy about this
30      // string rightArr[rightArrLen];
31      for (int i = 0; i < leftArrLen; i++) {
32          leftArr[i] = arr[left + i];
33      }
34      for (int i = 0; i < rightArrLen; i++) {
35          rightArr[i] = arr[mid + 1 + i];
36      }
37      //compare and sort
38      int i = 0, ii = 0, iii = left;
39      while (i < leftArrLen && ii < rightArrLen) {
40          comparisons++;
41          if (leftArr[i] <= rightArr[ii]) {
42              arr[iii] = leftArr[i];
43              i++;
44          } else {
45              arr[iii] = rightArr[ii];
46              ii++;
47          }
48          iii++;
49      }
50      comparisons++;//A comparison is made when the while loop stops
51      //merge back together
52      while (i < leftArrLen) {
53          arr[iii] = leftArr[i];
54          i++;
55          iii++;
56      }
57      while (ii < rightArrLen) {
58          arr[iii] = rightArr[ii];
59          ii++;
60          iii++;
61      }
62      return comparisons;
63  }
64
65  //Time complexity O(nlog2(n))
66  //Specifically it's slightly better on average
67  int mergeSort(string* arr, const int size) {
68      return mergeSortHelper(arr, 0, size - 1);
```

```
}
```

2 Linear Search

A linear search is performed over the lines array to check if each of the 42 random items are in the array. The number of comparisons is tracked and printed for each of the 42 items. Then the average is printed for all 42 items. The linear search goes through the lines array one at a time to check if the item is equal to the target item. Since the 42 items were randomly chosen, we should expect the 42 items to be evenly distributed within the lines array. Therefore we should expect the average number of comparisons to be equal to half of the length of the list, or $n/2$. $666/2 = 333$, which is close to the observed average of 329.471 over 10 samples. It should be half because a linear search can stop when it finds the item. This gives us an asymptotic running time of $O(n/2) = O(n)$. The n comes from having to search through each element in the array, one-at-a-time. Since it's evenly distributed, the average value of the 42 items should be roughly equal to the average value of all items.

main.cpp

```cpp
int main() {
    //...

    //Linear search
    int comparisons = 0;
    int totalComparisons = 0;
    for (const string& rand : rand42) {
        for (const string& ln : lines) {
            comparisons++;
            if (rand == ln) {
                break;//It's assumed that the target string will be found
            }
        }
        if constexpr (PRINT_EACH_ITEM) {
            printf("Linear search comparisons for %s: %d\n", rand.c_str(),
             ↪  comparisons);
        }
        totalComparisons += comparisons;
        comparisons = 0;
    }
    printf("Linear search average comparisons: %.2f\n",
     ↪  static_cast<float>(totalComparisons) /
     ↪  static_cast<float>(NUM_RAND));

    //...
}
```

3 Binary Search

A binary search is performed over the same arrays. The comparisons are tracked and printed, same as with linear search. I chose to use iteration instead of recursion for this implementation of binary search.

The binary search divides the lines array in half until the item in the middle matches the target. This gives the asymptotic running time of $O(log_2(n))$. Since we are dividing by 2, we expect that in the worst case, the number of comparisons should be $log_2(n)$. $log_2(666) = 9.379...$, which is a little higher than the observed average of 8.092 over 10 samples. The reason it's slightly higher is because that is worst case. For the average case, we should expect some elements to be found quickly. There is a $1/666$ chance at finding the item after 1 comparison, a $2/666$ chance at finding the item after 2 comparisons, a $4/666$ chance of finding the item after 3 comparisons, and a $2^{c-1}/666$ chance at finding the item after c comparisons. The maximum number of comparisons is $log_2(n)$ so we can sum all the chances from $c = 1$ to $c = log_2(n)$. This gives us a total number of comparisons of $n(log_2(n) - 1) + 1$. Divide this by the number of cases $n + 1$, and we get the following equation for average number of comparisons:

$$\frac{n\left(log_2\left(n\right) - 1\right) + 1}{n + 1} \tag{3.1}$$

Substituting 666 for n, the equation evaluates to 8.368, which is closer to the observed average of 8.092.

main.cpp

```
122 int main() {
123     //...
124 
125     //Binary search
126     //That's right, I did it without recursion. Stay mad Haskell developers!
127     totalComparisons = 0;
128     int left, right, mid;
129     for (const string& rand : rand42) {
130         left = 0;
131         right = NUM_LINES - 1;
132         while (left <= right) {
133             comparisons++;
134             mid = (left + right) / 2;
135             if (rand == lines[mid]) {
136                 break;//It's assumed that the target string will be found
137             }
138             if (rand < lines[mid]) {
139                 right = mid - 1;
140             } else {
141                 left = mid + 1;
142             }
143         }
144         if constexpr (PRINT_EACH_ITEM) {
145             printf("Binary search comparisons for %s: %d\n", rand.c_str(),
                ↪  comparisons);
146         }
147         totalComparisons += comparisons;
```

```
148             comparisons = 0;
149         }
150         printf("Binary search average comparisons: %.2f\n",
            ↪  static_cast<float>(totalComparisons) /
            ↪  static_cast<float>(NUM_RAND));
151
152         //...
153     }
```

4 Hash Table

4.1 Hash Table implementation

The HashTable struct is implemented with an array of Buckets of size 250, and a static hashing function. This hash table only works with strings. This implementation uses chaining to handle collisions. It comes with a method to insert a new string, and search for an existing string. The search function returns how many comparisons where made to find the string, or a non-positive number if it doesn't exist.

To insert 666 items into a hash table with 250 buckets, there must necessarily be collisions. If we assume that all 666 items were evenly distributed in each buckets, then each bucket should have $666/250 = 2.664$ items, which is a little bit less than the observed average of 3.423 over 10 samples. The reason for this discrepancy is that 2.664 items per bucket is the best case scenario, where there are no empty buckets. In reality, it is very unlikely for there to be no empty buckets. Some buckets will be empty, and some queues will have more than 2.664 items. When we search for an item that we know is in the hash table, we will never come across a bucket that is empty during the search. If we assume there are fewer than 666 buckets with items in them, there will necessarily be a higher average number of items per bucket than 2.664 among buckets with items, because it has to compensate for the empty buckets. In buckets with multiple items, we would expect the number of comparisons to be half of the number of items in the bucket because it uses a linear search, which would imply that there are around 6.846 items per bucket. We can calculate the efficiency of the hash function with the following equation:

$$\text{Efficiency} = \left(\frac{\text{Ideal Average Items per Bucket}}{\text{Actual Average Items per Non-empty Bucket}} \right) \times 100 \tag{4.1}$$

With our values:

$$38.9\% = \left(\frac{2.664}{6.846} \right) \times 100 \tag{4.2}$$

38.9% efficiency isn't very good for a hashing function, but I suppose it could be worse.

hashtable.hpp

```
35  struct HashTable {
36      void insert(const string& new_str);
37
38      ///Returns how many comparisons were used to check if the string is in
        ↪  the hash table.
39      ///Returns 0 if it doesn't exist.
40      int search(const string& find_str) const;
41
42  private:
```

```
43      static int hash(string value);
44
45      Bucket arr[HASHTABLE_SIZE];//hopefully the stack is cool with me dumping
        ↪  10,000 bytes on it
46  };
```

The insert and search functions call the hash function to obtain the index in the Bucket array, then insertion or search logic is delegated to the Bucket's insert or search method.

The hash function converts all characters to upper case, then sums the ASCII value of each character. The index returned is the remainder of the sum divided by the size of the hash table.

hashtable.cpp

```
7   void HashTable::insert(const string& new_str)
    ↪  {arr[hash(new_str)].insert(new_str);}
8   int HashTable::search(const string& find_str) const {return
    ↪  arr[hash(find_str)].search(find_str);}
9
10  int HashTable::hash(string value) {
11      int letterTotal = 0;
12      std::transform(value.begin(), value.end(), value.begin(), [](const
        ↪  unsigned char c){return std::toupper(c);});
13      for (int i = 0; i < value.length(); i++) {
14          letterTotal += static_cast<int>(value.at(i));
15      }
16      return letterTotal % HASHTABLE_SIZE;
17  }
```

4.2 Bucket implementation

The Bucket struct is a tagged union which represents one of three variants: Empty, Str, and Queue. Str means the bucket only has one string, and Queue means the bucket has a linked list of strings.

hashtable.hpp

```
12  enum class Tag {Empty, Str, Queue};
13  ///Tagged union for either an empty bucket, one string, or a linked list of
    ↪  strings.
14  ///I was going to use std::variant, but I can't link with it, so now I have
    ↪  to make my own tagged union.
15  struct Bucket {
16      Tag tag;
17      union {
18          string str;
19          Queue<string> queue;
20      };
21
22      Bucket();
```

```
23      ~Bucket();
24
25      ///If the bucket is empty, it will insert the new_str.
26      ///If the bucket has a string, it will turn it into a linked list with
        ↪  the old string + new_str.
27      ///If the bucket has a linked list, it will add new_str to the end.
28      void insert(const string& new_str);
29
30      ///Returns how many comparisons were used to check if the string is in
        ↪  the bucket.
31      ///Returns <= 0 if not found.
32      int search(const string& str) const;
33  };
```

In Bucket::insert, it uses the tag to determine how to insert the new string into the bucket. If it's Empty, it sets str to the new string and updates the tag to Str. If it's Str, it creates a queue and adds the existing string and the new string to that queue, and updates the tag to Queue. If it's Queue, it simply enqueues the new string.

hashtable.cpp

```
31  void Bucket::insert(const string& new_str) {
32      switch (tag) {
33          case Tag::Empty:
34              tag = Tag::Str;
35              new (&str) string(new_str);
36              break;
37          case Tag::Str: {
38              std::string copy = std::move(str);
39              str.~string();
40              tag = Tag::Queue;
41              new (&queue) Queue<string>();
42              queue.enqueue(copy);
43              queue.enqueue(new_str);
44              break;
45          }
46          case Tag::Queue:
47              queue.enqueue(new_str);
48      }
49  }
```

In Bucket::search, it uses the tag to determine how to search for the target string. If it can't find the target string, a non-positive integer is returned (either 0 or -1). If it's Str, it just returns 1 because 1 comparison was made. If it's Queue, it delegates the search logic to the search method for Queue, and adds 1 for the initial comparison.

hashtable.cpp

```
51  int Bucket::search(const string& find_str) const {
```

```
52      switch (tag) {
53          case Tag::Empty:
54              return -1; //not found
55          case Tag::Str:
56              assert(str == find_str);
57              return 1;
58          case Tag::Queue:
59              return 1 + queue.search(find_str);
60      }
61      return -1; //unreachable
62  }
```

4.3 Queue implementation

The Queue struct is a stripped-down queue with methods only required by this program. It uses a linked list implementation and uses a template for generic types, but only uses string in this program.

Queue::search uses a linear search to go through each element in the linked list until the item is found. It returns the number of comparisons, or -1 if the item was not found.

queue.hpp

```
6   ///First in, first out data structure with linked-list implementation.
7   ///For this project, this queue will be used more like a regular linked
     ↪  list,
8   ///and has some linked list methods instead of queue methods.
9   template<typename T> struct Queue {
10  private:
11      Node<T>* head;
12      Node<T>* tail;
13
14  public:
15      /// Creates an empty queue.
16      Queue() : head(nullptr), tail(nullptr) {}
17
18      /// Deallocates all allocated nodes from the heap.
19      ~Queue() {
20          while (head != nullptr) {
21              Node<T>* next = head->next;
22              delete head;
23              head = next;
24          }
25      }
26
27      /// Adds the item to the tail of the queue.
28      void enqueue(const T& item) noexcept {
29          if (head == nullptr) {
```

```
            head = tail = new Node<T>(item);
            return;
        }
        tail->next = new Node<T>(item);
        tail = tail->next;
    }

    ///Returns how many comparisons were used to check if the target is in
    ↪  the queue.
    ///Returns -1 if it doesn't exist.
    int search(const T& target) const {
        int comparisons = 0;
        Node<T>* curr = head;
        while (curr != nullptr) {
            comparisons++;
            if (curr->data == target) {
                return comparisons;
            }
            curr = curr->next;
        }
        return -1;
    }
};
```

4.4 Node implementation

The Node struct is the container for each element in a linked list. It uses a template for generic types, but only string is used in this program. It holds some data, and a pointer to the next element in the list.

node.hpp

```
/// Node for linked-list-style data structures.
/// Data structures are responsible for cleaning up resources used by Nodes.
template<typename T> struct Node {
    T data;
    Node* next;

    explicit Node(const T& item, Node* next = nullptr) : data(item),
    ↪  next(next) {}
};
```