Gianna Julio

CMPT435 - Algorithm Analysis and Design
Assignment 3 - Searching and Hashing

Assignment3.cpp

HashNode Class

```cpp
// node class
class HashNode
{
public:
    string item;
    HashNode* pointer;

    // default constructor initializes item as empty string and pointer as null
    HashNode()
    {
        item = "";
        pointer = nullptr;
    }

    // constructor that takes a specific item string as input
    HashNode(string i)
    {
        item = i;
        pointer = nullptr;
    }
};
```

The HashNode class is very similar to the basic node class from Assignment 1, but instead of holding an integer value, the value is a string containing the item's name. The HashNode has the same pointer concept. I also included two constructors to ensure that the values were initialized as I was running into issues comparing nodes with data that had not yet been assigned a value.

HashTable Class

```cpp
// hash table class
class HashTable {
private:
    // set size
    static const int tableSize = 250;

    // array of pointers to nodes
    HashNode* table[tableSize] = {nullptr};

public:
    // hash function for strings
    int hash(string target)
    {
```

```
        int hash = 0;
        for (char c : target)
        {
            hash += c;
        }
        return hash % tableSize;
    }

    // insert node into hash table
    void insert(string item)
    {
        // create new node
        HashNode* newNode = new HashNode(item);
        // compute hash
        int index = hash(item);
        // store what is currently at that index in a temp
        HashNode* temp = table[index];

        // if nothing is currently in this hash bucket, place new node here
        if (temp == nullptr)
        {
            table[index] = newNode;
        }
        // if another node already has this hash (collision), make this new node the next pointer
        else
        {
            // while there is still a next value
            while (temp->pointer != nullptr)
            {
                // store next value in temp
                temp = temp->pointer;
            }
            // temp is the last linked value, set its pointer to the new node
            temp->pointer = newNode;
        }
    }

    // look up target in hash table
    int lookup(string target)
    {
        // compute hash and initialize number of comparisons
        int index = hash(target);
        int comparisons = 0;

        // find first value with the computed hash and store it in temp
        HashNode* temp = table[index];

        // if first hash node has valid pointers
        while (temp != nullptr)
        {
            // increment comparisons
            comparisons++;

```

```
103             // iterate through the pointers
104             if (temp->item == target)
105             {
106                 // return comparisons when target value is found
107                 return comparisons;
108             }
109             // if not the target, continue iterating
110             temp = temp->pointer;
111         }
112         return comparisons;
113     }
114 };
```

The HashTable class includes the array of pointers that holds all of the hash nodes, the hashing function, an insert function, a and a lookup function that counts the comparisons. The array is initialized with null pointers and has a constant size of 250. The hash function (line 48) is the one from class, where each character's numerical value is added to a sum and the remainder after the sum is divided by the size of the table is found using the modulo operator. With this algorithm, each string is assigned a hash value from 0-249 that will serve as its index with minimal collisions. The insert function (line 59) computes the hash and checks the array at that index. If a value is not found there, the new node is inserted. If a value is already located at that index, then a collision has occurred and the new value is assigned to the last value's pointer. The lookup function (line 88) works similarly, computing the hash, checking at that index, and iterating through each node's pointers to find the target value, if necessary.

Linear Search

```
// linear search
int linearSearch(string arr[], int size, string target) {
    int comparisons = 0;

    for (int i = 0; i < size; i++) {
        // increment comparions - checking each item in list
        comparisons++;
        if (arr[i] == target) {
            // if target is found, break out of function
            return comparisons;
        }
    }
    // if not found, return total compares
    return comparisons;
}
```

Linear search is the simplest and slowest search function. It iterates through the entire list of sorted items from the beginning until it finds the target. The average amount of comparisons is approximately half of n.

Binary Search

```
// binary search
int binarySearch(string arr[], int size, string target) {
```

```
    int comparisons = 0;
    int low = 0;
    int high = size - 1;

    while (low <= high) {
        comparisons++;
        int mid = (low + high) / 2;
        if (arr[mid] == target) {
            return comparisons;
        }
        else if (arr[mid] < target) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return comparisons;
}
```

For the binary search function, I took a non-recursive approach. This algorithm essentially moves the high and low bounds of the search by checking the target against a middle value, almost ”zooming in” on the target. The average amount of comparisons is approximately $\log_2 n$.

Loading Items From File

```
// loadItems from last file
void loadItems()
{
    // initialize file
    fstream itemFile;
    itemFile.open("magicitems.txt", ios_base::in);

    if (itemFile.is_open())
    {
        // counter
        int i = 0;

        // while file still has items to read
        while (itemFile.good())
        {
            string item; // initialize item string
            getline(itemFile, item); // get line
            transform(item.begin(), item.end(), item.begin(), ::tolower); // transform whole string to lowercase
            items[i] = item; // store item string
            i++; // increment counter
        }
        itemFile.close();
    }
}
```

This function loads all of the items on the magic list into an array of strings. It opens the file, gets each item, saves it to the global array, then closes the file. I added the statement on line 33 to convert the entire string to lowercase in order to be able to accurately sort the strings.

Merge Sort

```cpp
// merge arrays back together for mergeSort
int merge(string arr[], int left, int mid, int right)
{
    // initialize comparision counter
    int comparisons = 0;

    // find size of two subarrays
    int arrayOne = mid - left + 1;
    int arrayTwo = right - mid;

    // create temp arrays
    auto* leftArray = new string[arrayOne];
    auto* rightArray = new string[arrayTwo];

    // copy data to temp arrays leftArray[] and rightArray[]
    for (int i = 0; i < arrayOne; i++)
        leftArray[i] = arr[left + i];
    for (int j = 0; j < arrayTwo; j++)
        rightArray[j] = arr[mid + 1 + j];

    int indexOfArrayOne = 0; // initial index of first sub-array
    int indexOfArrayTwo = 0; // initial index of second sub-array
    int indexOfMergedArray = left; // initial index of merged array

    // merge temp arrays back into array
    while (indexOfArrayOne < arrayOne && indexOfArrayTwo < arrayTwo) {
        // if value in left array is smaller, left value goes into merged array next
        if (leftArray[indexOfArrayOne] <= rightArray[indexOfArrayTwo]) {
            arr[indexOfMergedArray] = leftArray[indexOfArrayOne];
            indexOfArrayOne++;
        }
        // if right is smaller, right vale goes into merged array next
        else {
            arr[indexOfMergedArray] = rightArray[indexOfArrayTwo];
            indexOfArrayTwo++;
        }
        // increment comparison counter and index of merged array
        comparisons++;
        indexOfMergedArray++;
    }

    // copy remaining elements of left[], if any
    while (indexOfArrayOne < arrayOne) {
        arr[indexOfMergedArray] = leftArray[indexOfArrayOne];
        indexOfArrayOne++;
        indexOfMergedArray++;
    }
```

```
188
189     // copy remaining elements of right[], if any
190     while (indexOfArrayTwo < arrayTwo) {
191         arr[indexOfMergedArray] = rightArray[indexOfArrayTwo];
192         indexOfArrayTwo++;
193         indexOfMergedArray++;
194     }
195
196     // reallocate memory used for subarrays
197     delete[] leftArray;
198     delete[] rightArray;
199
200     return comparisons;
201 }
202
203 // merge sort
204 int mergeSort(string arr[], int start, int end)
205 {
206     // initialize comparison count
207     int comparisons = 0;
208
209     // base case
210     if (start >= end)
211         return comparisons;
212
213     // find midpoint of array
214     int mid = start + (end - start) / 2;
215
216     // recursively sort both sides
217     mergeSort(arr, start, mid);
218     mergeSort(arr, mid + 1, end);
219
220     // merge all subarrays back together after breaking out of recusion
221     // and add comparisions to running total
222     comparisons += merge(arr, start, mid, end);
223
224     return comparisons;
225 }
```

Merge sort is written over two functions, one that recursively splits the list into halves, and the other that sorts the subarrays as it merges them back together in the correct order. Beginning with the mergeSort function on line 189, the base case of the recursive function breaks out of the recursion when the start value passed into the function is greater than the end value - when the subarrays can no longer be split any smaller and contain only one value. Next, the midpoint is calculated and passed into two recursive calls of mergeSort as the end value of the first call and the start value of the second call, splitting the list into two sub arrays. This continues until the base case is met, at which point the merge function is called, merging all the subarrays back together as the recursion is "unraveled." The merge function determines the size of the two arrays, creates two temporary right and left subarrays, and copies the correct data into them. Then, the values in each temp array are copied into the final merged array, both right and left sub arrays checked for the next smallest value. This is where the comparison counter is incremented. The merge helper function is called for each subarray at the end of the mergeSort function, putting the entire array back together in order.

Displaying Formatted Output

```cpp
// display formatted summary of sorts
void display(string searchType, double comparisons)
{
    // display heading - left justified, width of 34, filler char of '-', title, end line
    cout << left << setw(34) << setfill('-') << searchType + " " << endl;
    // display label left aligned in 25 spaces, followed by value right aligned (default) with 8 spaces
    printf("%-25s %8.3f\n", "Average comparisions:", comparisons);
    // extra spacing
    cout << endl;
}
```

The display function outputs the formatted results of each search. It outputs the name of each search left aligned and padded with dashes to create a heading, then the label ”Average comparisons” left justified along with the actual value of the number of comparisons, right justified.

Main Function

```cpp
int main()
{
    // load and sort items
    loadItems();
    mergeSort(items, 0, 666);

    // create and fill hash table
    HashTable h;
    for (string i : items)
    {
        h.insert(i);
    }

    // initialize random seed
    srand(time(NULL));

    // initialize totals - used to compute averages later
    double totalComparesLinear = 0;
    double totalComparesBinary = 0;
    double totalComparesHashes = 0;

    // loops to pick a new random item, perform all searches, and compile results
    for (int i = 0; i < 42; i++)
    {
        // choose random item
        string randItem = items[rand() % 666];

        // linear
        double linearComparisons = linearSearch(items, 666, randItem);
        totalComparesLinear += linearComparisons; // add to running total

```

```
        // binary
        double binaryComparisons = binarySearch(items, 666, randItem);
        totalComparesBinary += binaryComparisons; // add to running total

        // hash table
        double hashComparisons = h.lookup(randItem);
        totalComparesHashes += hashComparisons; // add to running total

    }

    // compute averages
    double avgComparesLinear = totalComparesLinear / 42.0;
    double avgComparesBinary = totalComparesBinary / 42.0;
    double avgComparesHashes = totalComparesHashes / 42.0;

    // display summary
    display("Linear Search", avgComparesLinear);
    display("Binary Search", avgComparesBinary);
    display("Hash Table with chaining", avgComparesHashes);
}
```

The main function calls the loadItems function first to load the array with all of the magic items, then sorts them with merge sort. The hash table is initialized and all of the items are inserted utilizing the hash table insert function. The total amount of comparisons for each type of search is initialized so the average can be computed later. The random seed is also initialized here to ensure 42 different random items are generated each time. The for loop is executed 42 times, and each time, it randomly selects an item and does a linear, binary, and hashing search for that item, adding each respective number of comparisons to the running totals. After the loop finishes iterating, the averages are computed and displayed with the formatting function.

Sample Output

```
Linear Search --------------------
Average comparisions:      332.310

Binary Search --------------------
Average comparisions:        8.667

Hash Table with chaining ---------
Average comparisions:        2.286
```

This is a sample of the output after the program is run to display the formatting and sample average data values for each search type.