CMPT 435 - Fall 2024 - Dr. Labouseur

Assignment Two – Algorithms

Stefano Farro
stefano.farro1@marist.edu

November 1, 2024

1 Searching Algorithms

1.1 Linear Search

Linear search sequentially checks each element in the array until the target item is found or the end is reached. The implementation keeps track of comparisons made during the search and returns this count along with whether the item was found.

```
int linearSearch(std::vector<std::string>& a, std::string key)
{
    bool found = false;
    int comparisons = 0;
    for (int i = 0; i < a.size(); i++) {
        comparisons = comparisons + 1;
        if (a[i] == key) {
            std::cout << "Key found at index " << i << " |
    Comparisons: " << comparisons << '\n';
            found = true;
            break;
        }
    }
    if (!found) {
        std::cout << "Key Not Found!" << '\n';
    }
    return comparisons;
};
```

Listing 1: Linear Search Implementation

```
Key found at index 432 | Comparisons: 433
Key found at index 309 | Comparisons: 310
Key found at index 251 | Comparisons: 252
Key found at index 237 | Comparisons: 238
Average Linear Search Comparisons: 316.29
```

Figure 1: Average comparison count of Linear Search on list of 42 items

1.2 Binary Search

Binary search recursively divides the sorted array in half, comparing the middle element with the target to determine which half to search next. The implementation tracks comparisons across recursive calls and handles case-insensitive string comparisons.

```
int binarySearch(std::vector<std::string>& a, int low, int high,
    std::string key, int comparisons)
{
    if (low > high) {
    std::cout << "Key Not Found!" << '\n';
        return comparisons;
    }
    else {
    // find midpoint of array, compare to key, and move into left/
    right problem space recursively based on equality
        int mid = (low + high) / 2;

        comparisons = comparisons + 1;
        if (makeLower(key) == makeLower(a[mid])) {
            std::cout << "Key found at index " << mid << " |
    Comparisons: " << comparisons << '\n';
            return comparisons;
        }

        if (makeLower(key) > makeLower(a[mid])) {
            return binarySearch(a, mid + 1, high, key, comparisons)
    ;
        }
        else {
            return binarySearch(a, low, mid - 1, key, comparisons);
        }
    }
};
```

Listing 2: Binary Search Implementation

```
Key found at index 432 | Comparisons: 19
Key found at index 309 | Comparisons: 17
Key found at index 251 | Comparisons: 17
Key found at index 237 | Comparisons: 11
Average Binary Search Comparisons: 15.95
```

Figure 2: Average comparison count of Binary Search on list of 42 items

1.3 Time Complexity Analysis

Search Type	**Avg. Comparisons (22 runs)**	**Time Complexity**
Linear Search	332.43	$O(n)$
Binary Search	8.39	$O(\log n)$

Table 1: Comparison of Search Performance for 42 Items on 22 runs

Linear Search: $O(n)$ - The algorithm sequentially examines each element until finding a match or reaching the end. In our implementation, this resulted in an average of 332.43 comparisons across 22 runs, demonstrating how performance degrades linearly with input size. While simple to implement, this approach becomes impractical for large datasets.

Binary Search: $O(\log n)$ - By dividing the problem space in half with each comparison, this algorithm achieved significantly better performance with only 8.39 average comparisons. It logarithmic complexity makes it especially efficient for large sorted datasets, seen most prominently in Binary Search Trees.

2 Hash Tables

The hash table implementation uses chaining for collision resolution with a fixed size of 250 slots. The data structure consists of an array of linked list nodes, where each node contains a string item and a pointer to the next node in the chain. The implementation tracks both initial operations and chain traversal comparisons for analysis purposes.

2.1 Implementation

2.1.1 Constructors

The hash table is implemented with a node structure and fixed-size array:

```
// node constructor
HashTable::Node::Node(const std::string& s) : item(s), next(nullptr
    ) {}

// table constructor
HashTable::HashTable() {
    // initialize all chain heads to NULL
    std::fill(table, table + SIZE, nullptr);
}
```

Listing 3: Hash Table & Node Constructor

2.1.2 Hash Function

The hash function converts strings to uppercase and computes a hash code by summing the char ASCII values and applying modulo arithmetic:

```
int HashTable::hashCode(const std::string& s) {
    std::string upperStr = s;
    std::transform(upperStr.begin(), upperStr.end(), upperStr.begin
    (), ::toupper);
    int length = upperStr.length();
    int letterTotal = 0;

    for (int i = 0; i < length; i++) {
        char thisLetter = upperStr.at(i);
        int thisValue = (int)thisLetter;
        letterTotal = letterTotal + thisValue;
    }
    int code = (letterTotal * 1) % SIZE;
    return code;
}
```

Listing 4: Hash Function

2.1.3 Insert Operation

The insert function computes the provided item's hash code, wraps it in a new node, and places it in the table at the front of that table index's chain:

```
void HashTable::insertItem(const std::string& item) {
    int code = hashCode(item);
    Node* newNode = new Node(item);
    newNode->next = table[code];
    table[code] = newNode;
}
```

Listing 5: Insert Operation

2.1.4 Get Operation

The get function returns a pair containing a found status boolean and comparison count. It computes the item's hash code, then traverses the chain at that index tracking comparisons until the item is found or the chain ends:

```
std::pair<bool, int> HashTable::getItem(const std::string& item) {
    int comparisons = 1; // initalize to 1 for get operation
    int code = hashCode(item);
    Node* current = table[code];

    while (current != nullptr) {
        comparisons++;
        if (current->item == item) {
            return {true, comparisons};
        }
        current = current->next;
    }
    return {false, comparisons};
}
```

Listing 6: Hash Table Get Operation

2.1.5 Delete Operation

The delete function handles removal of items from the hash table by managing three cases: empty chains, items at the head of a chain, and items in the middle or end of a chain. It maintains proper chain linkage when removing nodes:

```cpp
void HashTable::deleteItem(const std::string& item) {
    int code = hashCode(item);
    Node* current = table[code];
    Node* prev = nullptr;

    // three cases to handle with deleetion:
    // empty chain
    if (current == nullptr) {
        return;
    }

    // item at head of chain
    if (current->item == item) {
        table[code] = current->next;
        delete current;
        return;
    }

    // item in middle/end of chain
    while (current != nullptr && current->item != item) {
        prev = current;
        current = current->next;
    }

    // if found, delete the item.
    if (current != nullptr) {
        prev->next = current->next;
        delete current;
    }
}
```

Listing 7: Delete Operation

```
Found item | Comparisons: 2
Found item | Comparisons: 5
Found item | Comparisons: 3
Found item | Comparisons: 4
Found item | Comparisons: 5
Average Get Comparisons: 3.55
```

Figure 3: Average comparison count of get operations on a Hash Table with 42 items

2.2 Performance Analysis

Performance metrics for retrieving the 42 items:

Operation Type	Best Case	Average Case	Worst Case
Insert	O(1)	O(1)	O(1)
Get	O(1)	O(1)	O(n)
Delete	O(1)	O(1)	O(n)

Table 2: Hash Table Operation Complexities

2.2.1 Insert Operation

Insertion generally takes O(1) time because the hash function quickly computes an index, and the item is placed at the beginning of the chain at that index. This constant time is maintained as long as chains remain short.

2.2.2 Get Operation

Retrieval also averages O(1), but in the worst case, it can degrade to O(n). This happens when many items map to the same index (a collision), requiring traversal through a chain of items. However, a well-distributed hash function minimizes chain length, maintaining average constant time.

2.2.3 Delete Operation

Deleting an item generally takes O(1) time if it's at the beginning of the chain. If the item is deeper in the chain, deletion may require traversal, giving it a worst-case complexity of O(n). With effective hash functions, the chains stay short, preserving constant time performance.