COMP20003: Assignment 2 Report

1473948 – Chao Rebecca Wei

Comparative analysis of the time and memory complexity of searching a linked list and patricia tree implementation of a dictionary.

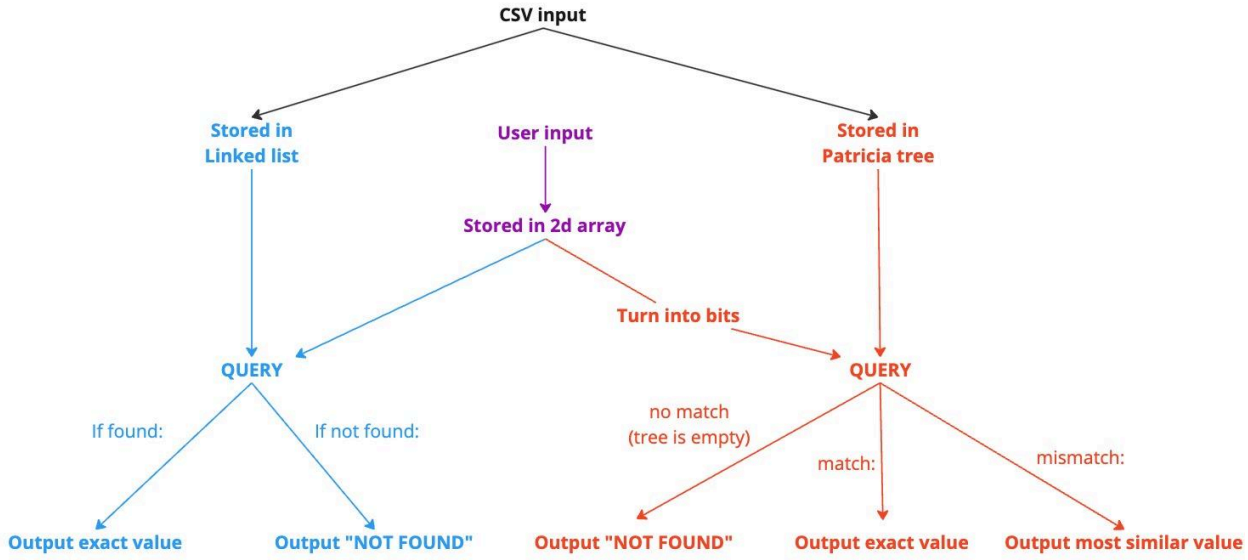
Introduction

This report is a comparative analysis of two data structures' performances (a linked list and a radix tree) in the context of implementing a dictionary for querying Australian suburbs. The dataset used for this analysis is a CSV file consisting of Australian suburbs and their attributes.

The aim of the dictionary is to allow users to efficiently query and retrieve attributes for each Australian suburb. The linked list dictionary handles exact queries while the patricia tree dictionary also handles approximate queries, returning the suburb with the closest edit distance. As this functionality is absent from the linked list implementation, for the purposes of this report, only exact matches will be considered in the comparative complexity analysis.

Methodology

The following diagram illustrates the program at a high level:



The user's queries, which are typed into the command line, are stored into a 2-dimensional dynamically allocated array and are iterated through for both data structures which search one query at a time.

Both the linked list and patricia tree are built using processed data from a CSV in a known format, using the same data agnostic module `data.c` that stores data into the following `suburb_data` struct:

```
typedef struct {
      int recordID;
      int suburbID;
      char* suburb_name;
      int year_recorded;
      char* state_IDs;
      char* state_names;
      char* gov_IDs;
      char* gov_names;
      double latitude;
      double longitude;
} suburb_data;
```

Data structure 1: Linked List

The linked list data structure is composed of a sequence of nodes, where each node represents a suburb entry and each node is connected sequentially to another. The `key` is the name of the suburb and the beginning and end of the list are tracked using the `head` and `tail` respectively.

```
typedef struct node {
    char* key;
    suburb_data *value;
    struct node *next;
} node_t;

typedef struct list {
    node_t *head;
    node_t *tail;
} list_t;
```

Theoretical Complexity Analysis of Searching the Linked List:

Once the linked list is built using the CSV content, a `linear_search` function iterating from the head to the tail node is performed. Once the record is found, the traversal still continues until the end of the list because the frequency of records within the dict needs to be output as well. Each node traversed is accessed and compared with the query char by char.

The theoretical **time complexity of searching is O(n*L)** where n is each node visited during the search and L is min(length of the query, average suburb name length). Traversing through all n nodes takes θ(n) time complexity while the time per string comparison is O(L) on average.

The theoretical **space complexity of searching is O(1)** as the search function is conducted non-recursively and does not require any auxiliary space.

Empirical Complexity Analysis of Searching the Linked List:

To empirically demonstrate the space and time complexity, the following test cases are examined:

1) Given test datasets and tests
Ran across multiple dataset sizes, we can empirically prove that the **θ(n) time complexity component** due to traversing through all nodes:

Dataset size (n)	1	15	100	1000
Nodes Accessed	1	15	100	1000

As seen, the number of nodes accessed increases linearly with dataset size.

2) Worst case: Query is very similar to suburb names within the list and differ only at the last character
To demonstrate the **O(L) time complexity component** of the search function, consider a worst-case scenario where almost every character in the query string must be compared:

Query: “South” and Test Dataset: {“South”, “South Wharf”, “South Yarra”, … }

Dataset size (n)	1	15	100	1000
Bit comparisons	48	720	4800	48000
Nodes Accessed	1	15	100	1000

The maximum number of character comparisons carried out per node, L, is min(length of query, length of suburb name in that node). In this case, L is constantly 6 characters or 48 bits. As the maximum number of comparisons need to be made for each node, the bit comparisons increase by a factor of 48 for each additional node. This demonstrates that the overall time complexity in the worst case is O(n*L).

3) Best case: All suburb names differ from the search key at the very first character
Consider this best-case scenario:

Query: “carlton” and Test Dataset: {“Carlton”, “Parkville”, “South Yarra”, … }

Dataset size	1	15	100	1000
Bit comparisons	8	120	800	8000
Nodes Accessed	1	15	100	1000

The minimum number of character comparisons carried out per node is 1, or 8 bits. As the mismatch occurs at the first character for each node, each string comparison takes

O(1) time. Thus, for each of the n nodes, the comparison takes constant time and the overall **best-case time complexity is O(n)**.

Data structure 2: Patricia Tree

The patricia tree data structure is composed of tree nodes which contains bit strings of the suburb names, including the null byte. `stem` refers to the prefix bit string, which is the bit string that all children of the node have in common. `results` is a linked list which is NULL unless the node is a leaf node, in which case it contains the suburb data of that sequence of nodes.

```
typedef struct prefix_node prefix_node_t;

typedef struct prefix_node{
    int prefix_bits;          // Number of bits in the prefix
    char* stem;               // The bit sequence of the prefix
    prefix_node_t* branch_0;  // Pointer to the branch starting with bit "0"
    prefix_node_t* branch_1;  // Pointer to the branch starting with bit "0"
    list_t* results;          // Linked list of results
} prefix_node_t;
```

Theoretical Complexity Analysis of Searching the Patricia tree:

The `search_ptree` function performs a traversal through the tree by taking the bit sequence of the search query and comparing it bit-by-bit with the current node. If there is a mismatch, the exact result is not found but if there is not, either the 0 or 1 branch is followed based on the value of the current bit. If the search query is found within the tree, this function traverses a single path from the root to the appropriate leaf node where the matching record is found.

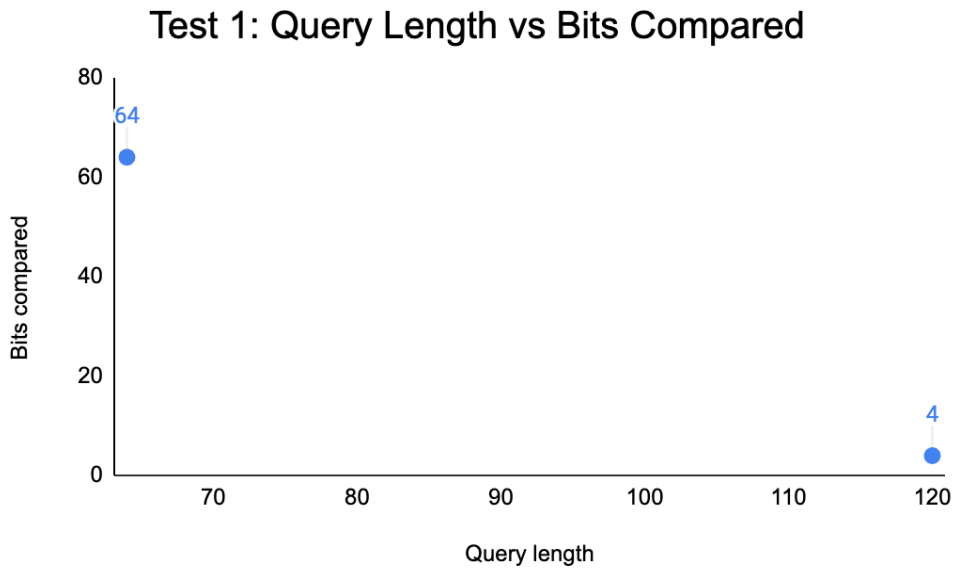
The theoretical **time complexity of searching is O(m)** where m is the number of bits (including the null byte) in the search string. The maximum number of bit comparisons that need to be made is equal to the bit length of the search query. During traversal, only one branch is followed based on the current bit's value, ensuring that only the necessary nodes are visited.

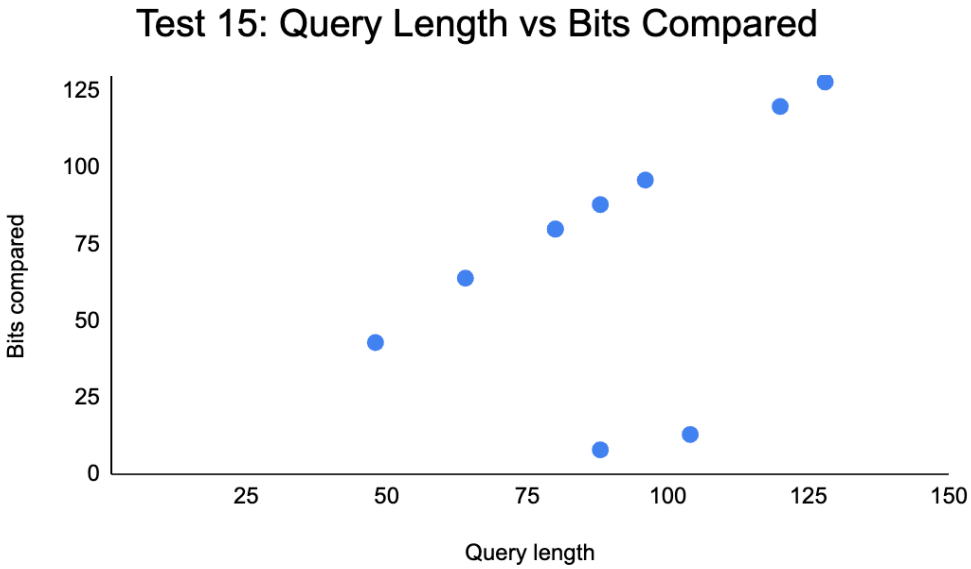
The theoretical **space complexity of searching is O(1)** as the search function is conducted non-recursively. It does require auxiliary space through creating temporary nodes and pointers but they remain constant regardless of input size.

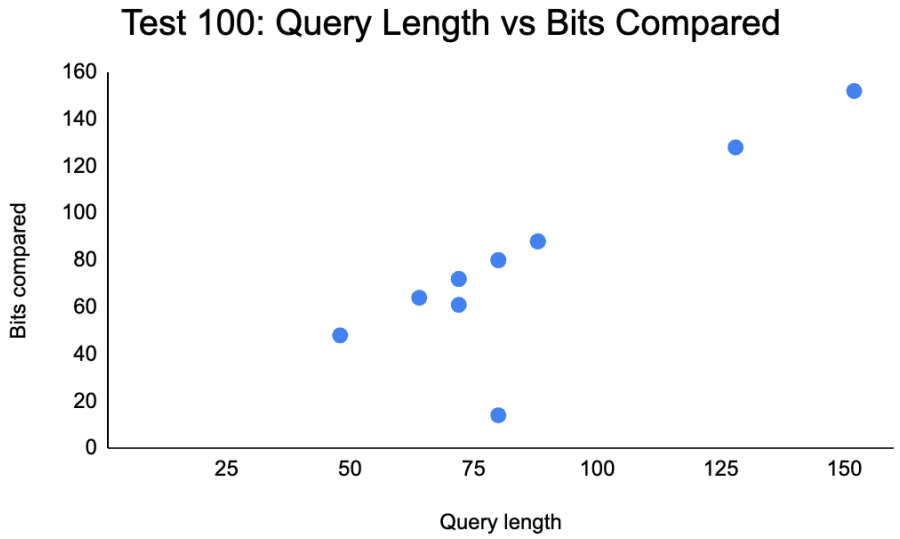
Empirical Complexity Analysis of Searching the Patricia tree:

To empirically demonstrate the space and time complexity, the following test cases are examined:

1) Given test datasets and tests


Test 1: Query Length vs Bits Compared
80
60
40
20
0
Bits compared
64
4
70
80
90
100
110
120
Query length


Test 15: Query Length vs Bits Compared
125
100
75
50
25
0
Bits compared
25
50
75
100
125
150
Query length



When run across multiple dataset sizes and tests, the empirical results demonstrate the time complexity of searching the patricia tree to be O(m). While search queries that are present within the dataset require as many bit comparisons as the number of bits within the search query itself (e.g. Carlton has 64 bits and exactly 64 bit comparisons were made before returning the query), search queries that are not present within the dataset require bit comparisons up to the mismatch before returning (e.g. South Melbourne has 120 bits but only needed 4 bits to be compared before a mismatch was found). This explains the lack of a clear linear relationship within the test 1 graph as there are only 2 queries: a match and a mismatch.

2) Best case: the search query mismatches at the root node
Consider this best case scenario:

Query: “carlton” and Test Dataset: {“Carlton”, “Parkville”, “South Yarra”, … }

Dataset size	1	15	100
Bit comparisons	3	3	3
Nodes Accessed	1	1	1

The best case occurs when an early mismatch occurs in bits. In this case, the query “carlton” has the beginning bits 011 due to the lowercase letter, while all the suburb names’ bit sequences begin with 010 due to being uppercase. The mismatch occurs on the third bit and the result NOT FOUND is returned. While this comparison count is constant across all datasets, the number of bits in the search string which needs to be compared in the beginning still depends on the query’s bit sequence, as such, the **best-case time complexity is O(m)**.

Conclusion

In conclusion, the search functions of both data structures follow theoretical time and space complexities but the patricia tree is significantly more efficient in terms of bits compared.