COMP20003 - ASSIGNMENT 2

Report

An Analysis of Complexity: Linked List vs Patricia Tree

Introduction

Different data structures exist because no single structure can efficiently handle all types of operations. The choice depends on specific requirements, such as time complexity, space complexity and application of frequent operations, including insertion, deletion, search. This report is an attempt to represent the analysis of two commonly used dictionaries for searching keys: linked list and patricia tree. The analysis is based on a dataset of Australian suburbs derived from ABS data, focusing on the performance of these structures in searching operations.

Data structure and inputs

Linked List

A linked list is a simple linear data structure where each element is a node. Each node contains two parts: the data (which holds the node's value) and a pointer to the next node in the list. In cases where a node also has a pointer to the previous node, the list is known as a doubly linked list. A linked list typically stores a reference to the first node, called the "head," and sometimes the last node, known as the "tail." In this analysis, we focus on a singly linked list, where each node has a data pointer that points to a structure containing detailed information about every suburb.

Patricia Tree

A patricia Tree is a specialised form of a trie where common prefixes are compressed to enhance space efficiency. Each node in the tree stores a binary prefix, minimising the number of nodes required for storage. The leaf nodes contain the actual data. Throughout this report, the binary prefixes are derived from the official suburb names, and each leaf node in the patricia tree stores a linked list of suburbs that share the same name.

Inputs

The inputs in this study consist of a list of official suburb names. These names are used as keys for searching through both the linked list and the patricia tree to find matching records.

Preprocessing

I implemented a Python function to visualise search complexity (included in my submission). This function processes a given dataset by taking the search keys input file and the output files containing search information for two dictionaries, generated in Stages 3 and 4. It first calculates the length of each key and records the corresponding number of bits compared and nodes accessed for both dictionaries. The data is then grouped by key length, and the mean values for each metric are computed within each group.

The function generates two line charts: one showing the "Bits Compared" for both the linked list and patricia tree across different key lengths, and another displaying the "Nodes Accessed" for each data structure. Additionally, it provides summary statistics, including the total number of data suburbs, the number of search keys, the average and maximum bits compared and nodes accessed, and the ratio between these metrics for the two data structures.

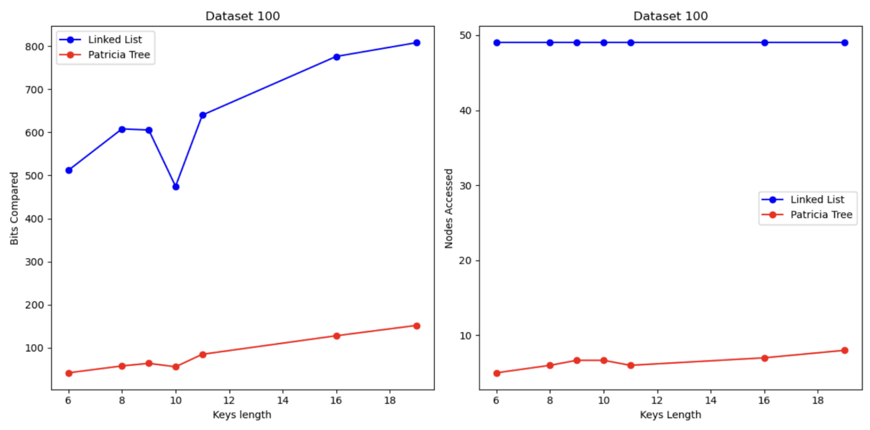
I chose not to include the string comparisons because, in the case of the linked list, it is always equivalent to the number of nodes, while for the Patricia tree, it is consistently 1. Given this, further preprocessing or visualising the string comparisons would be redundant.

Search Analysis

Real-world Datasets (Average cases)

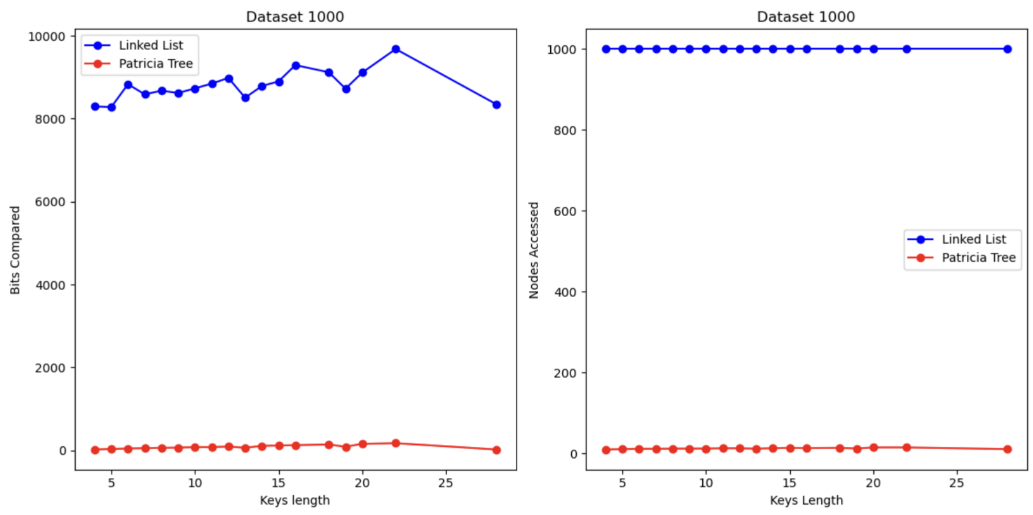
These datasets, available in the "Dataset Download" section, represent real-world data characteristics, including ordinary, unsorted arrangements with several duplicate entries.

```
compare("test100.in", "test100.s3.stdout.out", "test100.s4.stdout.out", "Dataset 100")
```



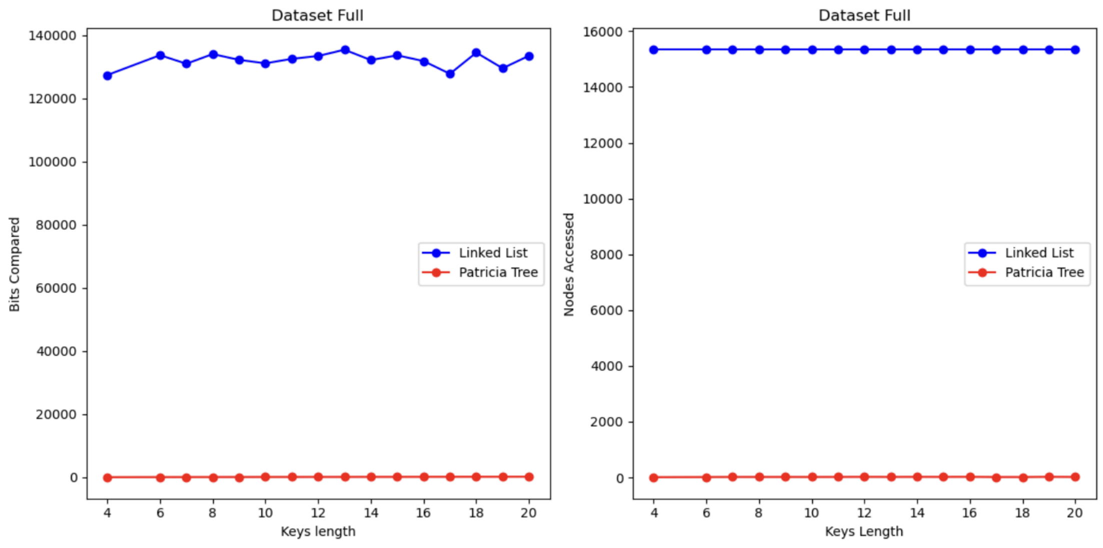
```
Suburbs: 49; Keys: 12

Bits Compared:
List: mean: 602.0, max: 808
Tree: mean: 75.8, max: 152
Average List/Tree = 7.9

Average Node Accessed:
List: mean: 49.0, max: 49
Tree: mean: 6.5, max: 8
Mean List/Tree = 7.5
```

```
compare("test1000.in", "test1000.s3.stdout.out", "test1000.s4.stdout.out", "Dataset 1000")
```



```
Suburbs: 1000; Keys: 91

Bits Compared:
List: mean: 8752.4, max: 9680
Tree: mean: 78.8, max: 176
Average List/Tree = 111.0

Average Node Accessed:
List: mean: 1000.0, max: 1000
Tree: mean: 12.5, max: 18
Average List/Tree = 79.9
```

```
compare("testfull.in", "testfull.s3.stdout.out", "testfull.s4.stdout.out", "Dataset Full")
```



```
Suburbs: 15334; Keys: 161

Bits Compared:
List: mean: 132612.6, max: 151032
Tree: mean: 83.9, max: 160
Average List/Tree = 1580.5

Average Node Accessed:
List: mean: 15334.0, max: 15334
Tree: mean: 19.4, max: 28
Average List/Tree = 788.7
```

At first glance, the plots reveal a clear distinction between the two data structures, with the patricia tree demonstrating far greater efficiency compared to the linked list.

A closer analysis of the bit comparison shows that, across all datasets, the linked list exhibits a gradual increase in the number of bits compared as key length increases, indicating that more comparisons are needed for longer keys. In contrast, the patricia tree remains relatively stable, with a much lower number of bits compared throughout.

For nodes accessed, the linked list consistently maintains a constant number of nodes visited for all key lengths in each dataset. This is expected, as every search requires traversing the entire list to find all matching nodes, resulting in the same number of nodes accessed regardless of key length. On the other hand, the patricia tree requires significantly fewer nodes to be visited, further showcasing its superior efficiency in this metric.

Overall, as the datasets grow in size, the efficiency gap between the two data structures becomes even more notable. For example, when examining the "average bits compared" ratio of the linked list to the patricia tree, with 49 suburbs (Dataset 100), the ratio is 7.9. For 1000 entries, this ratio skyrockets to 111, a 14-fold increase. In the full dataset with 15,334 suburbs, the ratio climbs to 1580.5—again 14 times higher than in the 1000-entry dataset. These results emphasise the substantial scalability advantage of the patricia tree as dataset size increases.

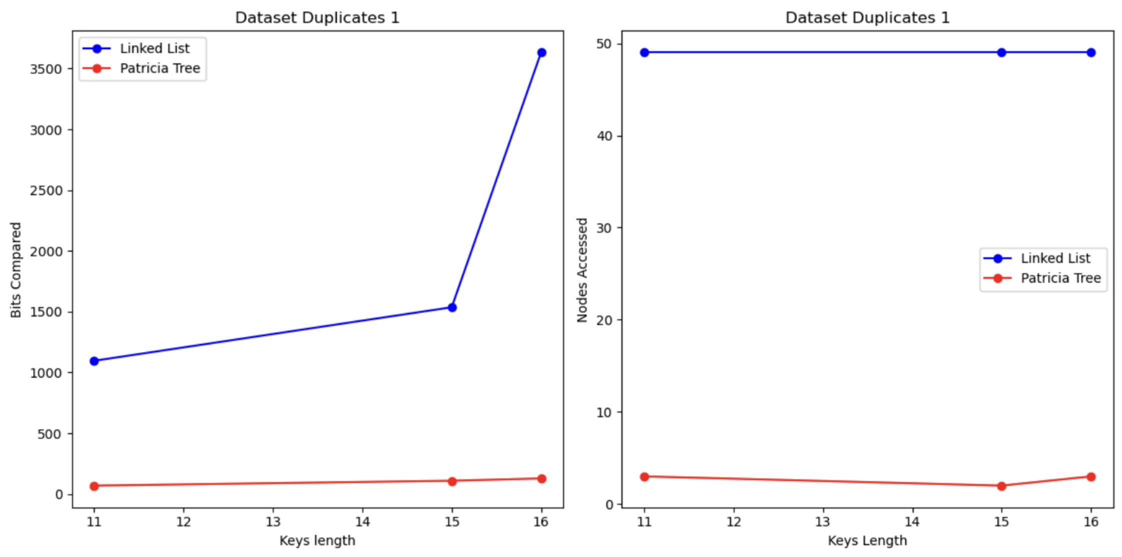
Specialised Datasets (Edge cases)

These are custom datasets that I manually created for testing purposes and have included in my data assignment submission. While they do not accurately represent real-world applications, they are valuable for exploring the characteristics of each data structure. The datasets include:

- dataset_dup1: Contains 49 entries with only 3 unique keys, with many duplicates for each key.
- dataset_dup2: A larger version of dataset_dup1, consisting of 98 entries with the same 3 unique keys.
- Dataset_100sort: Features 49 entries, identical to dataset_100, but the orders are alphabetically sorted by their keys. I used Python to sort; code is included in assignment submission.

These specialised datasets allow for a focused examination of how each data structure handles duplicates and sorted data.

```
: compare("testdup.in", "testdup1.s3.stdout.out", "testdup1.s4.stdout.out", "Dataset Duplicates 1")
```

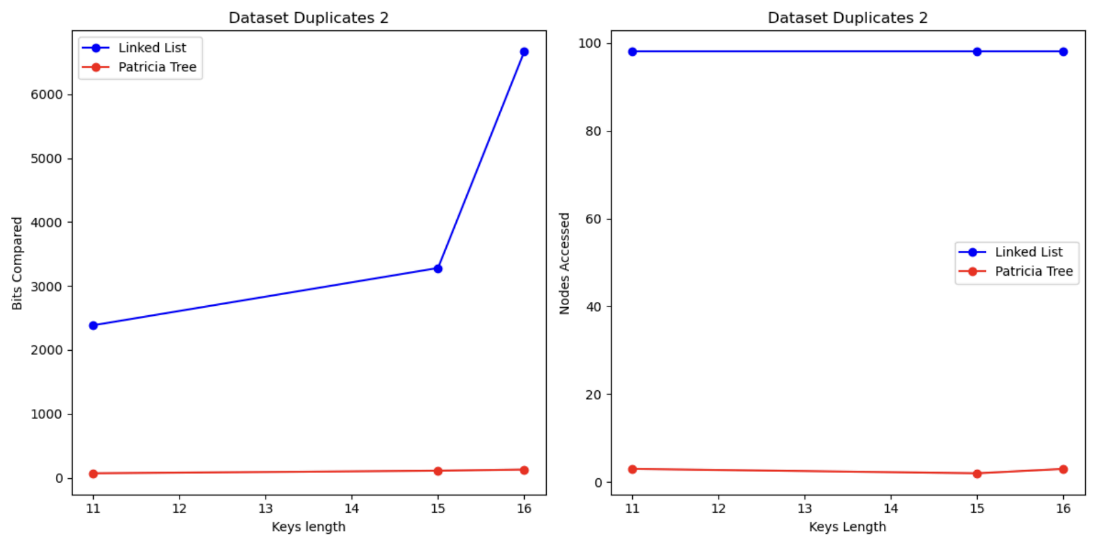


```
Suburbs: 49; Keys: 3

Bits Compared:
List: mean: 2088.0, max: 3632
Tree: mean: 102.0, max: 128
Average List/Tree = 20.5

Average Node Accessed:
List: mean: 49.0, max: 49
Tree: mean: 2.7, max: 3
Average List/Tree = 18.4
```

This plot highlights the stark difference between the two data structures in handling duplicate entries. The dataset contains 49 entries, identical to dateset_100, allowing for a clear comparison between the two figures. Despite having slightly more bits compared and fewer nodes accessed, the patricia tree continues to demonstrate consistency across different datasets. In contrast, the linked list compared three times more bits in the duplicate dataset compared to the real-world dataset, underscoring its inefficiency when dealing with duplicates.

These differences can be attributed to the structure of each dictionary. In the patricia tree, all duplicates of a key are stored in a separate linked list, meaning that once the tree reaches the linked list, all duplicates have already been found. Additionally, the presence of more duplicates results in fewer unique keys, which in turn reduces the number of nodes and nodes accessed during searches. On the other hand, the linked list requires a sequential visit to every entry, comparing each character one by one. As the number of duplicates increases, the number of characters compared grows significantly, further emphasising the linked list’s disadvantage in handling such datasets.

```
compare("testdup.in", "testdup2.s3.stdout.out", "testdup2.s4.stdout.out", "Dataset Duplicates 2")
```

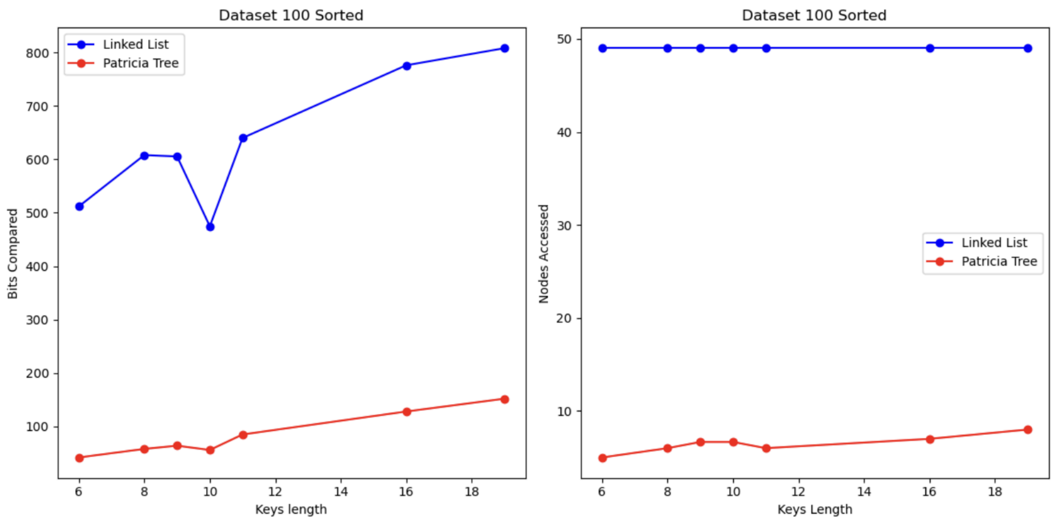


```
Suburbs: 98; Keys: 3

Bits Compared:
List: mean: 4109.3, max: 6664
Tree: mean: 102.0, max: 128
Average List/Tree = 40.3

Average Node Accessed:
List: mean: 98.0, max: 98
Tree: mean: 2.7, max: 3
Mean List/Tree = 36.8
```

The key takeaway from this dataset is the patricia tree's consistent performance across different-sized datasets with the same duplicates. Although this dataset contains 98 entries, double that of dataset_dup1 (49 entries), the mean and maximum values for both bits compared and nodes accessed remain identical to those of dataset_dup1. As previously mentioned, the patricia tree stores all duplicates in a separate linked list, so the structure and search time are determined by the number and size of unique keys rather than the quantity of duplicates.

For the linked list, the output behaves as expected. Since the dataset is twice the size, the number of bits compared and nodes accessed is also doubled, demonstrating the linked list's linear complexity.

```
: compare("test100.in", "test100sort.s3.stdout.out", "test100sort.s4.stdout.out", "Dataset 100 Sorted")
```



```
Suburbs: 49; Keys: 12

Bits Compared:
List: mean: 602.0, max: 808
Tree: mean: 75.8, max: 152
Average List/Tree = 7.9

Average Node Accessed:
List: mean: 49.0, max: 49
Tree: mean: 6.5, max: 8
Average List/Tree = 7.5
```

For this sorted dataset, it is important to note that the results are exactly identical to those of dataset_100. This suggests that the arrangement of data entries has no impact on the search performance of either data structure. The reason behind this is: the linked list, by nature, lacks any order logic, requiring a full traversal regardless of the key order. On the other hand, the patricia tree's structure is determined purely by the binary representation of the keys, meaning the insertion order does not influence the efficiency during searches.

Comparison with theory

The analysis above demonstrates a strong alignment between the theoretical and practical understanding of data structures. However, certain discrepancies remain, particularly in specific scenarios

Linked List

Traverse Complexity: **O**(**n**) where n is the entry number

Practical results agree with this theory: to locate an element, each node must be accessed, meaning the number of node accesses is directly proportional to the number of entries. This confirms that the traversal complexity is linear, growing proportionally with the size of the linked list.

String Search Complexity: **O**(**n***k**) where n is the entry number and k is the max length of a string
This is where practical findings diverge slightly from theoretical predictions. In theory, searching for a string within a linked list can have a worst-case complexity of **O**(**n** * **k**), particularly when the strings are nearly identical and must be compared character by character until the last character. However, in real-world datasets, as seen in the examples above, we often only compare the first few characters before finding a mismatch. Given that the English alphabet has 26 possible characters for each position, early mismatches are frequent. As a result, the average case complexity tends to be closer to **O**(**n**) rather than the theoretical **O**(**n** * **k**).

<u>Patricia Tree</u>
String Search Complexity: **O**(**k**) where k is the max length of a string
The practical results align with the theoretical predictions. Across all key searches in different datasets, only one string comparison is made per search, and the number of characters compared is consistently less than or equal to the length of the longest key. This supports the key advantage of patricia trees: they store binary representations of keys, with branching occurring only at points where the keys differ. As a result, the worst-case time complexity remains **O**(**k**), regardless of the number of entries. Therefore one of the patricia tree's significant benefits is its scalability. Even as the dataset grows, the maximum length of a key remains relatively small, keeping the overall complexity nearly constant.

Additionally, the specialised structure of patricia trees proves highly efficient when handling datasets with duplicates, where linked lists typically struggle. Another strength is that patricia tree performance is independent of the order in which keys are inserted. This contrasts with standard binary search trees, where the insertion order can significantly impact the tree's shape and, consequently, its search efficiency.

Discussion: Trade-offs

Although the patricia tree demonstrates superior performance in most scenarios, there are still key trade-offs that make linked lists a dependable and versatile data structure for a variety of applications.

<u>Space Complexity</u>
A linked list is simpler and generally has lower memory overhead. Each node only needs to store data and the next node. In contrast, a node in patricia tree needs to store left, right and

parent pointers, as well as the data. The tree also has to store more nodes for different branches, therefore higher overhead memory.

Insert / Delete Complexity

A linked list is more efficient for frequent insertions and deletions at the head or tail, when the complexity is in constant time **O**(**1**), while the complexity for a patricia tree is **O**(**k**) with k being the length of the key.

Ease of implementation

A linked list is easy to implement and understand. It requires minimal effort to manage memory, making it a straightforward choice for beginners and when the data structure needs to be simple. Patricia tree, on the other hand, is more complex to implement due to the need to handle binary prefixes and branching conditions. This complexity adds overhead, especially for applications that do not require advanced search performance.

Conclusion

In summary, the analysis confirms that the patricia tree offers significantly better time complexity for search operations compared to a linked list, particularly when dealing with large datasets or datasets containing many duplicate entries. Its compressed structure enables efficient searches regardless of the order in which data is inserted, addressing a key limitation of binary search trees. However, this performance gain comes at the cost of increased implementation complexity and higher memory usage. Therefore, the choice of data structure should be guided by the project's scale, the frequent operations required, and the preferences for simplicity.