

# AADS Seminar – 0036581063 (Matis DUBOIS) – Topic 4

**GitHub Repository:** [https://github.com/matisdbs37/DUBOIS\\_Matis\\_AADS\\_Topic4](https://github.com/matisdbs37/DUBOIS_Matis_AADS_Topic4)

## Introduction

In computer science, a string is defined as a sequence of characters stored in memory. Since they are used a lot for a wide range of things such as text processing or search engines for example, we need to design data structures with an efficient manipulation of strings, in terms of execution time and memory usage.

The simplest way to process strings is to do it sequentially. For example, if I want to find the first occurrence of a word in a text, I can search in the whole text, character by character. For a very short text, it could be efficient. However, if I have to search a single word into a whole book, it will start to take a very long time and it's not efficient anymore. This motivates the use of specialized string data structures, with a goal to organize strings in memory to speed up common operations such as insertion, deletion and search.

Among these data structures, we can find tree-based representations. They take advantage of the fact that a lot of strings share common prefixes. By storing these prefixes only once, we can reduce redundant computations and improve search performance.

We can take two examples of tree-based representations: Trie (or Prefix Trie) and Patricia Trie. Both structures operate on the same principle of organizing strings with their shared prefixes. However, they have some differences in their internal representation and efficiency, which I will explain in more detail later.

The goal of this seminar is to compare the Prefix Trie and the Patricia Trie from a practical point of view. Both structures are implemented in Python and evaluated using the same datasets. Several operations are considered, including insertion, deletion, search and range search. The comparison focuses on execution time and memory consumption, and experiments are conducted on datasets of increasing size to study the scalability of each structure.

## Algorithm presentation

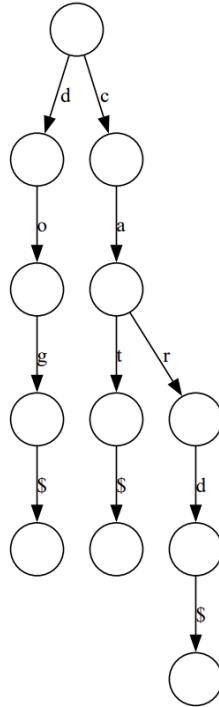
Let's now explain in more details how the Prefix Trie and the Patricia Trie work.

A Prefix Trie, also called a Trie, is a tree structure used to store strings. The tree is composed of nodes and edges. The root node is the starting point of the Prefix Trie. It doesn't store any character itself, but it only serves as the origin of all paths.

Each node represents a single character of a string. Nodes can have multiple children, one for each possible next character.

The path from the root to a node corresponds to a prefix of one or more stored words. The main idea is to share common prefixes among words, which reduces redundant storage and allows fast searches. Some nodes are marked as “\$”, which indicates that a complete word finishes at this node.

We can take the example of a Prefix Trie with the words dog, cat and card:



**Figure 1: Example of Prefix Trie**

We can do 4 basic operations on this Prefix Trie:

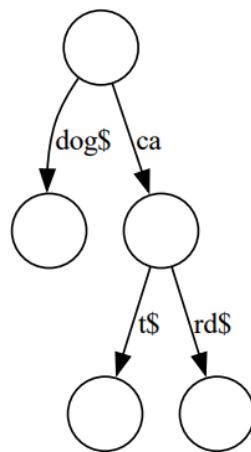
- **Insertion:** When inserting a new word, the algorithm checks for common prefixes among edges, starting at the root. Then, it just follows the path corresponding to the word's characters. If a character doesn't exist in the children of the current node, it simply creates a new node. When it reaches the last node, it's marked as end of the word.
- **Search:** When searching for a word, the algorithm follows the path corresponding to the word. If the path exists and the last node is marked as end of a word, the word is in the Prefix Trie. Otherwise, the word doesn't exist.
- **Range search:** This operation corresponds to search all words that start with a common prefix. For example, in the figure 1 Prefix Trie, I can do a range search with “ca”, and the algorithm will return cat and card. So, we just must reach the node corresponding to the prefix (if this node exists), then traverse all descendant nodes to collect the words.
- **Deletion:** Deleting a word is simple if this word doesn't share any prefix with the other words of the Trie, the algorithm just must remove all nodes corresponding to the word. It's the case with dog in Figure 1 for example. However, it becomes

more difficult to delete a word that share prefixes with the rest of the words. The algorithm must remove nodes that are no longer needed, while preserving nodes shared with other words. This can involve recursively deleting nodes from the leaves back to the root.

In general, for all these operations, time complexity would be  $O(L)$  where  $L$  is the length of the word used for the operation. Memory usage can be high, especially when many nodes are needed for unique prefixes.

A Patricia Trie, also called compressed Trie, is an optimization of the Prefix Trie. Instead of storing one character per node, Patricia Trie compresses chains of nodes with only one child (it means that they don't share prefixes with some other words in the Trie) into a single edge. This edge is labelled with a string that correspond to all the compressed characters. This reduces the number of nodes and can save a big amount of memory.

We can take the same example with dog, cat and card, but for a Patricia Trie:



**Figure 2: Example of Patricia Trie**

We can also do 4 basic operations on a Patricia Trie. These operations are very similar to those of a Prefix Trie, but they differ in the way characters are stored and compared:

- **Insertion:** As in the Prefix Trie, insertion starts at the root and looks for common prefixes. However, instead of comparing characters one by one, the algorithm compares substrings stored on edges. If the word shares only a part of an existing edge label, the edge must be split to create a new branching point, with new edges added for the remaining parts of the word. The last node is marked as end of the word. The main difference is that insertion with Patricia Trie is more complex because of edge splitting, but it creates fewer nodes.
- **Search:** The search operation is like the Prefix Trie. The algorithm simply follows a path from the root to try to match a word. The difference is that it compares several characters at once instead of a single character per node.
- **Range search:** The range search operation works also in the same way as in a Prefix Trie.
- **Deletion:** Deletion is more complex than in a Prefix Trie. As before, the goal is to remove nodes or edges that are no longer needed while keeping shared prefixes

intact. In a Patricia Trie, deleting a word can require merging edges after removal, to preserve the compressed structure. This step is the difference between the 2 structures.

Time complexity is generally  $O(L)$  as well, with  $L$  the length of the word used for the operation. It can be slightly higher due to substring comparisons instead of character comparisons. Memory complexity is much lower than the Prefix Trie.

To conclude, both structures are efficient for string operations, but Patricia Trie is more efficient for the memory while maintaining similar time performance. Let's check that with some experiments in Python.

## Experiment setup

The goal of the experiments is to compare the performance of the Prefix Trie and the Patricia Trie in practical conditions. The comparison focuses on two aspects: execution time and memory consumption for the 4 basic operations defined in the previous section.

The project was developed using Python, with a clean and reproducible environment. A Git repository was created on Github at the beginning of the project to manage version control. A `gitignore` file was configured to exclude unnecessary files such as virtual environments folders. Indeed, I used `venv` to create a virtual environment to isolate dependencies and guarantee that the experiments could be reproduced independently of the system configuration.

The two string data structures were implemented in the files `src/trie.py` and `src/patricia.py`: a Prefix Trie based on the classes `TrieNode` and `Trie`, and a Patricia Trie based on the classes `PatriciaNode` and `PatriciaTrie`. Both implementations support the same set of operations: insertion, search, range search and deletion.

Before running performance benchmarks, a visualization step was introduced. I created a dedicated script in the file `src/visualize.py` with `Graphviz` library to display the structure of both tries for small datasets. This step helped verify the correctness of the implementations and clearly illustrate structural differences between the tries, especially the path compression used in the Patricia Trie.

In parallel, I also created a test script in `src/script.py` to validate all basic operations on small examples. This ensured that both data structures work correctly before being evaluated on larger datasets. This script is still available as an example, but it can be changed to check other examples if necessary.

To study scalability, several datasets of increasing size were generated: 100, 1000, 5000 and 10000 words. Each dataset contains words composed of 8 random characters and is stored in a separate text file (`data/words_[N].txt`). Using incremental dataset sizes allows to observe how performance evolve as the number of words increases.

The performance evaluation is carried out using a script `src/benchmark.py`. For each dataset size, both data structures are tested under the same conditions. The script measures:

- Execution time for insertion, search, range search and deletion
- Memory usage during the insertion

All measurements are collected automatically and printed in the terminal. Based on the results, a performance graph (results/performance.png) is generated and stored, allowing a visual comparison between Prefix Trie and Patricia Trie.

## Experiment results

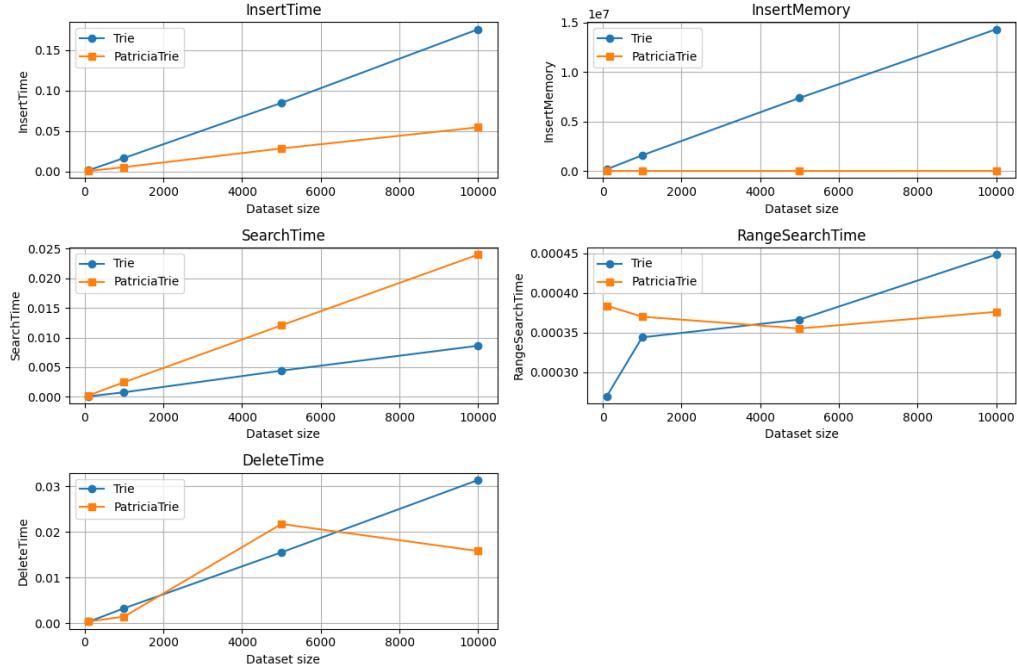
This section presents the results obtained from the experimental evaluation of the Prefix True and Patricia Trie. As a reminder, the goal is to compare their performance in terms of execution time and memory usage, using datasets of increasing size (100, 1000, 5000 and 10000 words).

For each dataset size, the benchmark script prints a summary table directly in the terminal. This table reports, for each metric, the measured value for both data structures and the relative difference in percentage.

Size	Metric	Trie	Patricia	Diff %
100	InsertTime	0.001759	0.000791	55.03
100	InsertMemory	181160.000000	13831.000000	92.37
100	SearchTime	0.000077	0.000247	-221.20
100	RangeSearchTime	0.000269	0.000384	-42.46
100	DeleteTime	0.000340	0.000383	-12.64
1000	InsertTime	0.016829	0.005593	66.76
1000	InsertMemory	1594992.000000	12090.000000	99.24
1000	SearchTime	0.000763	0.002464	-222.79
1000	RangeSearchTime	0.000344	0.000370	-7.59
1000	DeleteTime	0.003282	0.001466	55.34
5000	InsertTime	0.084966	0.028627	66.31
5000	InsertMemory	7386032.000000	12090.000000	99.84
5000	SearchTime	0.004411	0.012060	-173.42
5000	RangeSearchTime	0.000366	0.000355	3.08
5000	DeleteTime	0.015534	0.021744	-39.97
10000	InsertTime	0.175872	0.054763	68.86
10000	InsertMemory	14327776.000000	11390.000000	99.92
10000	SearchTime	0.008617	0.023994	-178.45
10000	RangeSearchTime	0.000448	0.000376	16.11
10000	DeleteTime	0.031390	0.015821	49.60

**Figure 3: Metric Table Results**

These printed results provide precise measurements, but it can be difficult to interpret with just text, so let's check the file results/performance.png:



**Figure 4: Performance graph for all metrics**

The insertion time increases with the dataset size for both data structures. However, the Prefix Trie requires more time than the Patricia Trie. For small datasets, the difference remains limited, but it becomes much more significant when the number of words increases. For the largest dataset (10000 words), the insertion time of the Prefix Trie is approximately 3 times higher than the Patricia Trie. This can be explained by the fact that the Prefix Trie creates a new node for each character, leading to many nodes creations.

This also led to a much higher memory consumption for the Prefix Trie. It has a strong increase in memory consumption as the dataset size grows. This is expected since each character of each word is stored in a separated node. On the other hand, the memory usage for the Patricia Trie remains almost constant and negligible compared to the Prefix Trie. This result highlights one of the main advantages of the Patricia Trie: it reduces a lot the memory usage.

For the search, Prefix Trie outperforms Patricia Trie for all dataset sizes. The search time for Prefix Trie increases slowly and remains lower than the Patricia Trie. In Patricia Trie, each step involves comparing substrings instead of characters, which is longer. These results show that the Prefix Trie is more efficient for exact search operations, despite its higher memory consumption.

For the range search, both data structures show similar execution times, with small differences between them. The Prefix Trie shows a slight increase in range search time as the dataset size grows, while the Patricia Trie remains more stable. Neither structure dominates for range search, even if the Patricia Trie seems to show a better scalability.

Finally, for the deletion and for smaller datasets, the Patricia Trie can be slower than the Prefix Trie. This is due to the additional complexity of deletion in a Patricia Trie, where removing a word can require merging some edges. However, for the largest dataset, the

Patricia Trie becomes faster. Deletion performance depends on the structure of the stored words so it's difficult to conclude with that.

## **Discussion and conclusion**

The goal of this seminar was to compare two string data structures, the Prefix Trie and the Patricia Trie, from a practical perspective. The experimental results clearly highlight the trade-offs between these structures.

The Prefix Trie offers a very simple structure, where each node represents a single character. This design makes operations like exact search more efficient, as the algorithm only needs to follow a sequence of characters without performing substring comparisons.

However, this simplicity comes at a significant cost in terms of memory usage. Since each character is stored in a single node, the Prefix Trie quickly increases the memory usage as the dataset size grows.

On the other hand, the Patricia Trie optimizes memory usage by compressing chains of nodes into single edges labelled with substrings. This structure reduces the number of nodes so fewer operations are required during insertion. This makes the Patricia Trie good for large datasets.

In conclusion, the results obtained are consistent with theoretical expectations. Neither data structure is strictly superior in all scenarios. The Prefix Trie is a good choice when memory is not a critical constraint and fast search operations are required. On the other hand, the Patricia Trie is better for large applications where memory efficiency and insertion performance are crucial. This study demonstrates the importance of selecting a data structure based on the specific requirements of the application rather than relying on a single performance metric.