| **CS 7280: Data Str & Alg Scalable Comp** | Spring 2025 |
|---|---|
| Lecture 5 | |
| *Prof. Prashant Pandey* | *Scribe: Azhar Abdulla* |

# 1 Overview

In previous lectures, we explored integer data structures and succinct data structures. In this lecture, we shift our attention to string data structures, specifically focusing on efficient string and pattern matching. This is particularly relevant in applications such as genome analysis, where finding occurrences of a pattern within a long sequence is crucial. To address these challenges, we will delve into tries and suffix trees, which provide powerful tools for managing and querying string data, with query times of $O(P)$ and space complexity of $O(T)$.

## 1.1 Recap: Previous Data Structures

**Integer Data Structures:** We began with Van Emde Boas (vEB) trees, which require $O(U)$ space. This was followed by the X-Fast Trie, which reduces space to $O(n \log U)$, and finally the Y-Fast Trie, achieving $O(n)$ space. All of these data structures support operations with $O(\log \log U)$ time complexity.

**Succinct Data Structures:** These focus on achieving space usage close to $\text{OPT} + o(\text{OPT})$, where OPT is the optimal representation size. Instead of traditional pointers, they utilize compact bit vectors of size $2n + 1$. Techniques like rank and select are employed for efficient navigation and queries within the compressed structure.

## 1.2 Assignment 1 Tips

- **Increase the base case size:** Instead of recursively dividing down to size 2, increase the base case size to a manageable threshold. For smaller sizes, simply iterate through the bit vector directly, avoiding unnecessary recursion overhead.
- **Consider using two classes:** Implement one class for the tree structure and another for leaf nodes. This modular design simplifies management and improves code readability.
- **C++ branch prediction hints:** Use the `[[likely]]` and `[[unlikely]]` attributes in if-statements inside loops to guide the compiler's optimization process. These attributes help the compiler generate machine code that prioritizes the most probable branch, reducing mispredicted branch penalties. This optimization reduces pipeline stalls by allowing the CPU to prefetch instructions for the expected branch.

# 2 String Matching

The primary goal of string matching algorithms is to efficiently locate occurrences of a pattern $P$ within a text $T$, often optimizing for both preprocessing time and query time. These algorithms aim to minimize computational overhead while supporting "fast" operations such as exact or approximate matching, substring queries, and handling dynamic updates to the text or pattern.

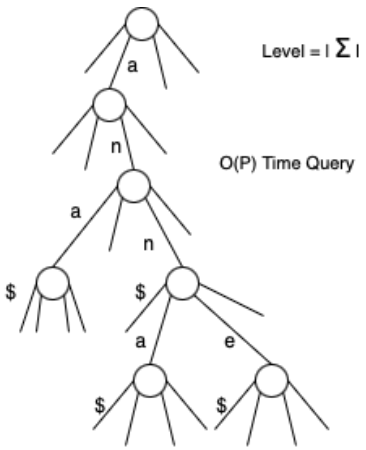
## 2.1 Approaches

- **One-shot methods:** These achieve $O(T)$ time complexity. Examples include:
  - Knuth-Morris-Pratt (KMP)
  - Boyer-Moore
  - Karp-Rabin (CACM 1997)
- **Static data structures:** Preprocess the text $T$ to allow efficient queries for pattern $P$. The goal is to achieve:
  - $O(P)$ time complexity for queries
  - $O(T)$ space complexity for preprocessing

## 2.2 Trie

- A Trie is a rooted tree where child branches are labeled with letters from the alphabet $\Sigma$.
- Strings are represented as root-to-leaf paths in the Trie.
- To distinguish prefixes from complete strings, a special symbol \$ is appended to the end of each string. This ensures that prefixes can be clearly identified as either absent or present.

**Diagram of the Trie (Ana, Ann, Anna, Anne)**



| Data Structure | Query Time | Space Complexity |
|---|---|---|
| Array | $O(P)$ | $O(T \cdot \lvert\Sigma\rvert)$ |
| Balanced Binary Search Tree (BBST) | $O(P \log \lvert\Sigma\rvert)$ | $O(T)$ |
| Hashtable | $O(P)$ | $O(T)$ (no predecessor/successor) |
| vEB / Y-Fast Tree | $O(P \cdot \log\log \lvert\Sigma\rvert)$ | $O(T)$ |
| Tries | $O(P + \log \lvert\Sigma\rvert)$ | $O(T)$ |
| Weight BBST | $O(P + \log k)$ | $O(T)$ |

Table 1: Query and Space Complexity for Different Trie Representations

### 2.3 Compressed Tries

A compressed trie is a space-efficient variant of a standard trie that reduces the number of nodes by combining chains of single-child nodes into a single edge. Instead of labeling edges with single characters, compressed tries use substrings, allowing entire paths to be represented by one edge.

The primary goal of a compressed trie is to reduce memory usage while preserving the functionality of a standard trie. This optimization is particularly useful when working with datasets where strings share long common prefixes, such as in dictionary compression or genome sequence analysis.

**Key Properties:** Compressed tries maintain the following characteristics:

- Each internal node has at least two children, ensuring that branching occurs only when necessary.
- Edges are labeled with substrings instead of individual characters.
- The total number of nodes is proportional to the number of strings, making the data structure linear in size relative to the input.

To construct a compressed trie, chains of single-child nodes in a standard trie are merged into a single edge. For example, if the path "b $\rightarrow$ a $\rightarrow$ n $\rightarrow$ a $\rightarrow$ n $\rightarrow$ a" exists in the trie, it is compressed into a single edge labeled "banana."

**Advantages:** Compressed tries offer significant memory savings and improve traversal speeds for long prefixes. By minimizing the number of nodes and edges, they enhance the overall efficiency of the data structure.

**Example:** Consider the strings {banana, band, bandage}. In a standard trie, each character would occupy its own edge and node, resulting in significant redundancy for common prefixes. In a compressed trie, the prefix "ban" is combined into a single edge, and branches occur only when necessary, such as at "d" for "band" and "dage" for "bandage." This results in a much more compact representation.

### 2.4 Suffix Tree

A suffix tree is a specialized compressed trie used to represent all suffixes of a given string.

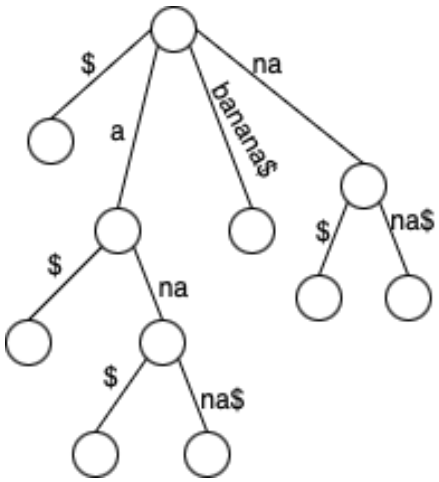
**Definition:** The suffix tree for a string $S$ of length $n$ is a rooted, directed tree where:

- Each edge is labeled with a substring of $S$.
- Each suffix of $S$ corresponds to a unique path from the root to a leaf.
- No two edges starting from the same node can share a prefix.
- A special terminator symbol \$ is appended to $S$ to ensure that no suffix is a prefix of another.

**Example:** Consider the string $S =$ `banana`. After appending the terminator \$, the suffixes of $S$ are:

`banana$`, `anana$`, `nana$`, `ana$`, `na$`, `a$`, `$`

The suffix tree for $S$ would look like this:




The tree compresses repeated substrings into shared paths. For example:

- The common prefix "ana" is represented as a single edge.
- Each suffix ends at a unique leaf, ensuring all suffixes are distinct in the tree.

**Construction Time:** The suffix tree can be constructed in $O(n)$ time for a string of length $n$, using algorithms such as Ukkonen's algorithm.

**Key Advantages:** Suffix trees allow for efficient pattern matching and substring analysis, with $O(|P|)$ query time and $O(n)$ space complexity. Their ability to compress shared substrings makes them ideal for applications involving large datasets.

# 3 Summary

This lecture focused on efficient string data structures, transitioning from integer-based structures to those designed specifically for string processing. A central theme was balancing query time and space complexity to achieve optimal performance for tasks like pattern matching and substring queries. The trie was introduced as a foundational data structure, representing strings as root-to-leaf paths with edges labeled by letters. To optimize space usage, compressed tries were explored, which combine single-child paths into single edges labeled with substrings, significantly reducing memory usage and traversal overhead.

Building on this, the suffix tree was presented as a powerful tool for representing all suffixes of a string in a compressed form. Constructed in $O(n)$ time, it enables efficient operations like substring search ($O(P)$), finding the longest repeated substring, and solving complex pattern-matching problems, such as those found in genome analysis. A key example used was the string "banana," illustrating how suffix trees efficiently represent all substrings while minimizing redundancy.

The lecture also compared various string data structures, such as arrays, balanced binary search trees, hash tables, vEB/Y-Fast Trees, and tries, highlighting their trade-offs in query time and space efficiency.