Tries

- Motivation and Definition
- Standard Tries
- Compressed Tries
- Suffix Tries

Motivation

Scenario 1

You're building a website that needs to provide fast substring matching on Shakespeare's *Hamlet*.

Scenario 2

You're maintaining a genomic database to allow fast retrieval of specific DNA sequences.

Pattern Matching

- Both of these scenarios are pattern matching problems (i.e. searching for a particular substring in a larger String.)
- Later, we will see some pattern matching algorithms.
- Here, we focus on a particular data structure that can be used for fast re*trie*val. (i.e. fast search)
- This is useful...when?

In both scenarios, we have...

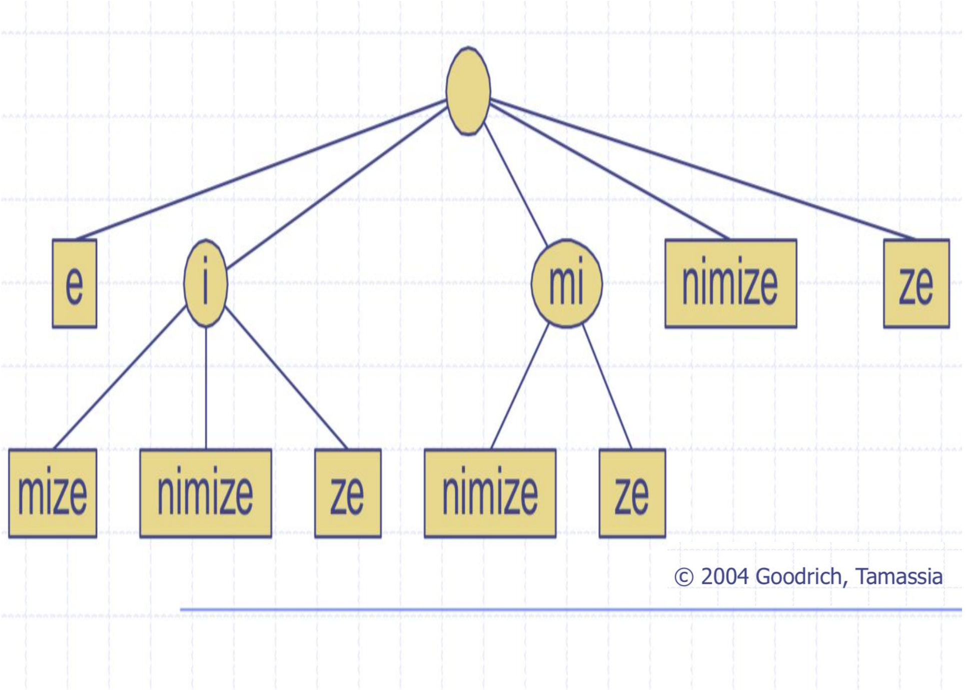
- ...a fixed text
- ...the potential for many queries
- ...the goal of allowing for fast pattern matching and data re**trie**val

In fact, the word “trie” comes from the word “retrieval” because their main application is data retrieval.

Definition

“A **trie** (pronounced “try”) is a tree-based data structure for storing strings in order to support fast pattern matching.”

(from Tamassia and Goodrich)



Main Operations

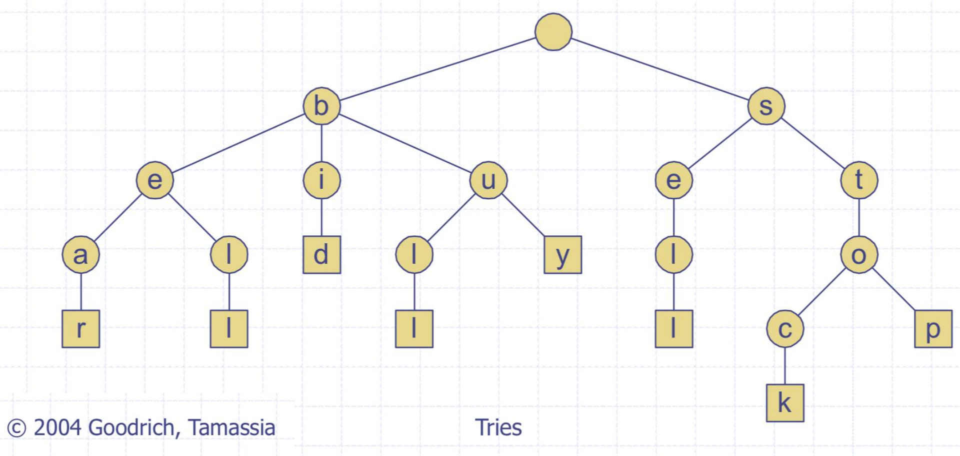
Given a collection **S** of strings, all defined with the same alphabet…

- ...efficiently search for a pattern string **P** (i.e. **pattern matching**)
- ...efficiently search for all strings in **S** that contain a pattern string **P** as a prefix (i.e. **prefix matching**)

Tries: pre-processing the text

- With tries, the idea is to **pre-process the text string**, which increases the pre-processing time but saves time later because of the **many search operations on the same text**.
- Useful if the text is large, immutable, and searched often.
- This allows us to do better than **O(n+m)** for text of size **n** and pattern of size **m**. (i.e. better than the best PM algorithm we'll see later)
- A trie supports pattern matching queries in time proportional to the size of the pattern **P**: **~O(m)**

Standard Tries



A **standard trie** for a set of strings **S** is an ordered tree such that:

- Each node but the root is labeled with a character.
- The children of a node are alphabetically ordered.
- The paths from the external nodes to the root yield the strings of **S**.
- Assumption: no string in **S** is a prefix of another string in **S**.

Doesn’t the restriction that no string be a prefix of another string diminish the usefulness of tries?

Doesn’t the restriction that no string be a prefix of another string diminish the usefulness of tries?

NOPE!

(There’s a very simple way of handling this.)

Consider the text:

canaries can help detect stack overflows

We should be able to search for both “can” and “canaries”…

Consider the text:

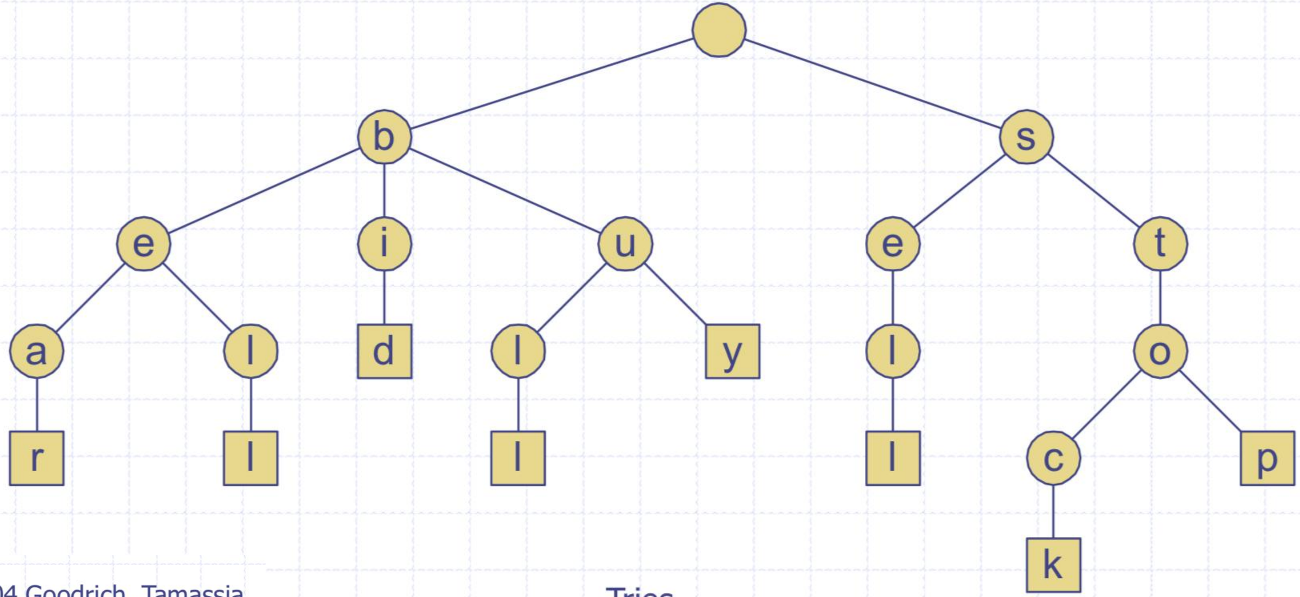
canaries can help detect stack overflows

We should be able to search for both “can” and “canaries”...

...so we just add an arbitrary character not in the original alphabet to the end of each string: {**canaries$, can$, help$, detect$, stack$, overflows$**}

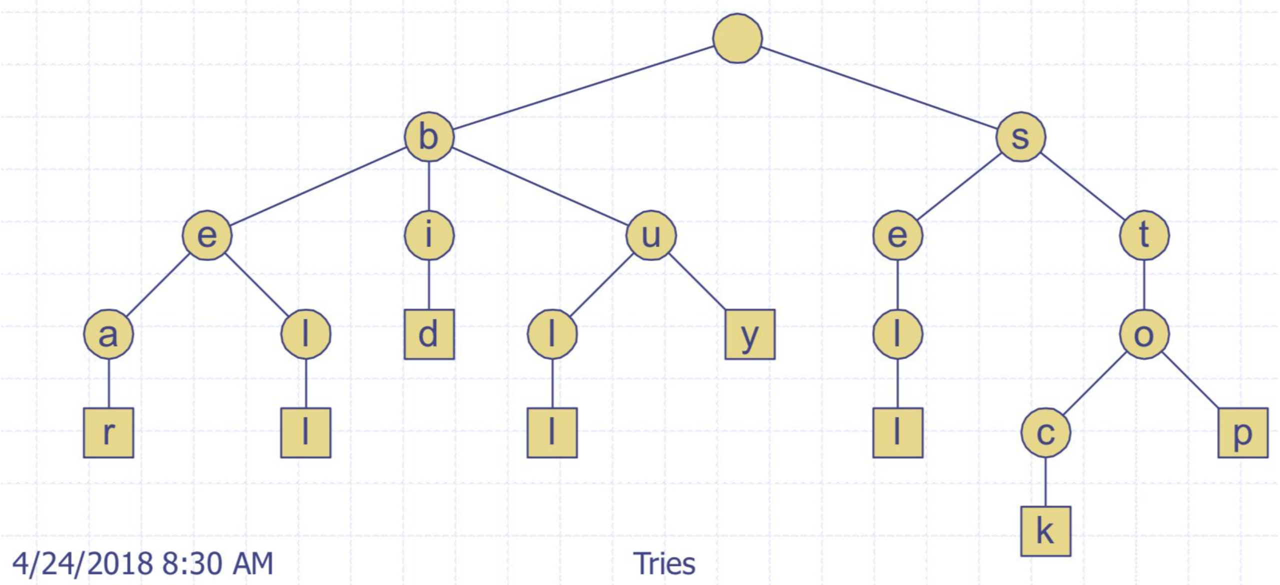
Standard Trie Example

S = {bear, bell, bid, bull, buy, sell, stock, stop}



Standard Trie Analysis

- What is the space requirement for the trie?
- What is the maximum height of the trie?
- How many external nodes does the trie have?



Standard Trie Analysis: Some Properties

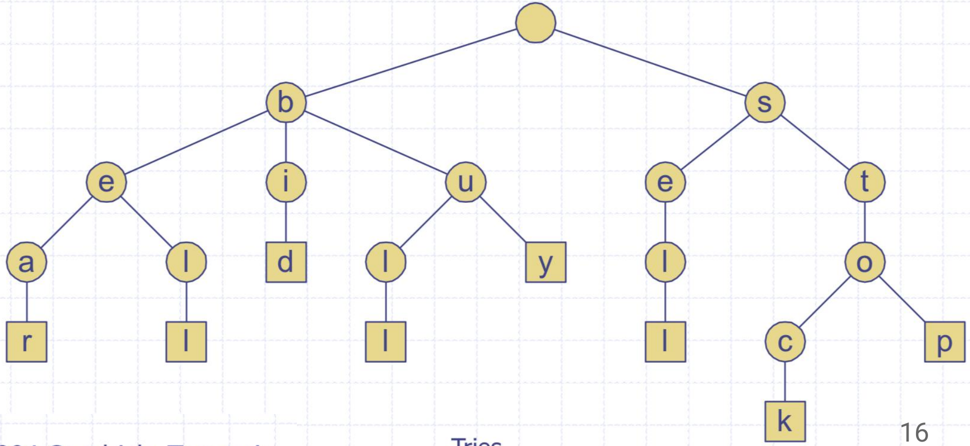
Let **T** be a standard trie storing a collection **S** of **s** strings of total length **n** from an alphabet of size **d**.

- Every internal node of **T** has at most **d** children.
- **T** has **s** external nodes.
- The height of **T** is equal to the length of the longest string in **S**.
- The number of nodes of **T** is **O(n)**.

Standard Trie Analysis

Let **n** be the total size of the strings in **S**, **m** be the size of the (maximum) string parameter of a given operation, and **d** be the size of the alphabet

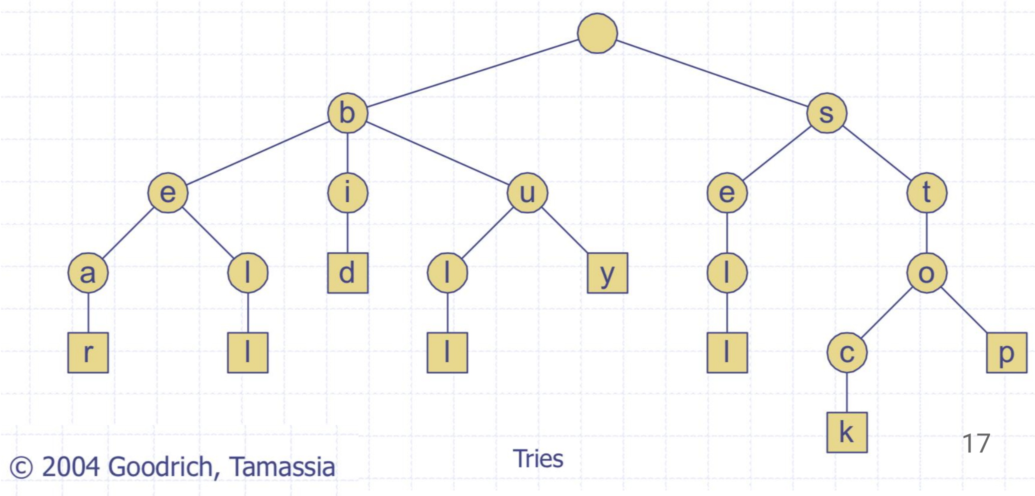
- Space requirement: **O(n)**
- Search, insert, and delete: **O(dm)**



Tries

Standard Trie Analysis

Q: When is the space maximized?

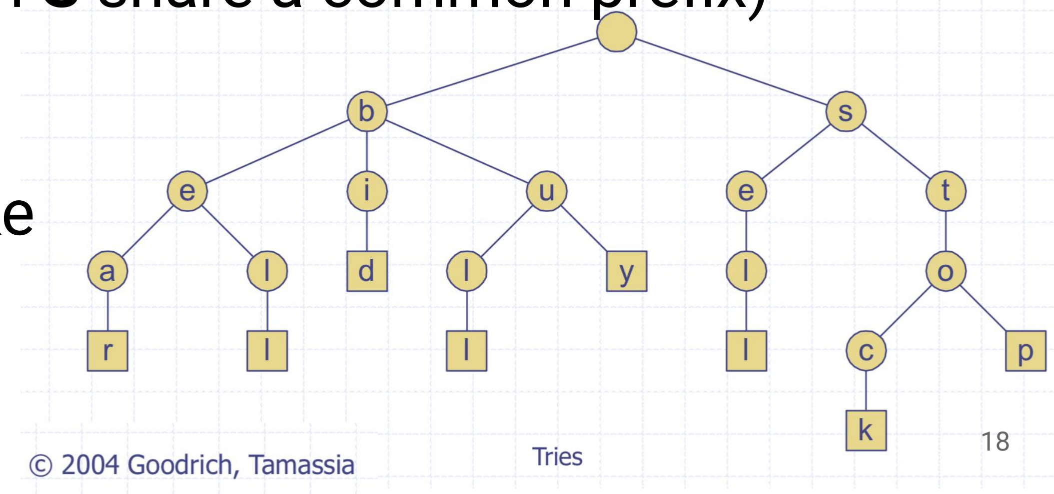


Standard Trie Analysis

Q: When is the space maximized?

A: When **S** consists of mutually unique words with no letters in common (i.e. no two strings in **S** share a common prefix)

In other words, each internal node has only one child (unlike the picture to the right).

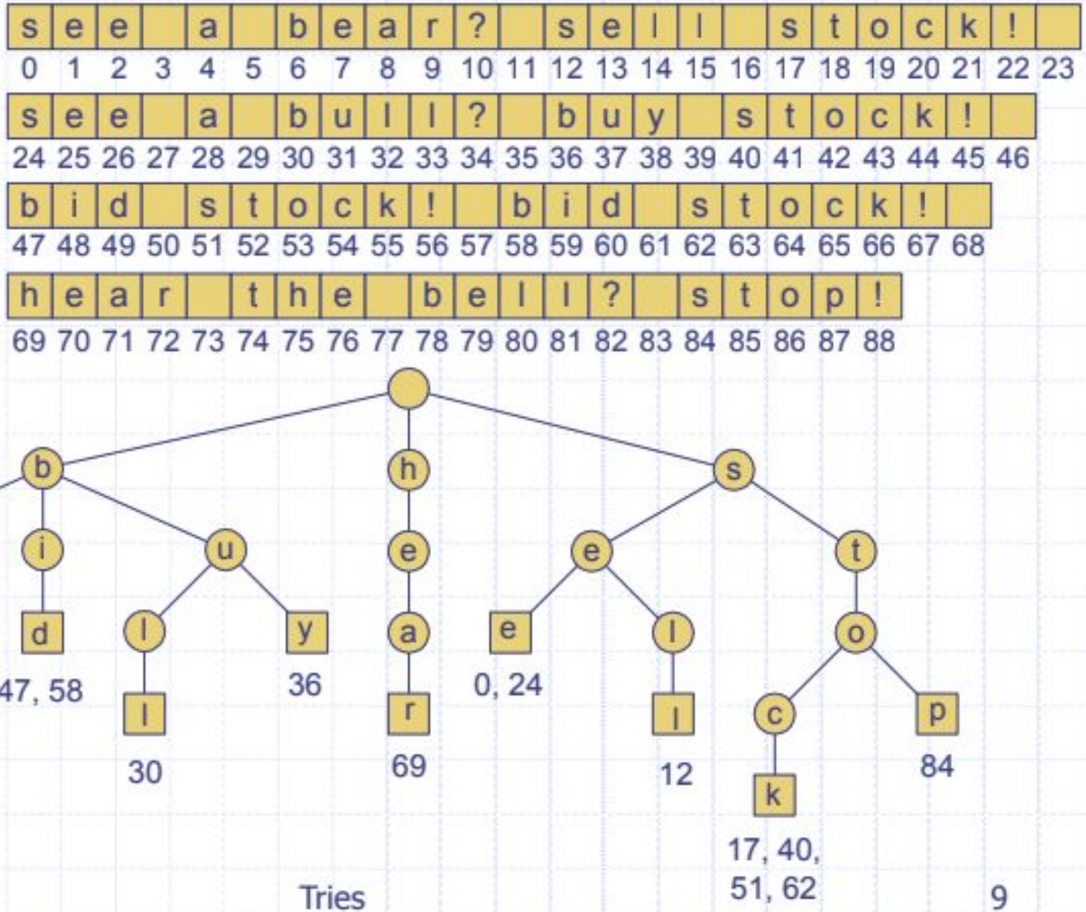


Word Matching

- a special type of pattern matching where we are searching for a specific **word**
- cannot match an arbitrary substring of the text, only one of its words
- can be done with a standard trie in **O(dm)** time, and if **d** is constant (like with English text or DNA strings), that time simplifies to **O(m)**
- cannot efficiently find a proper suffix of a word or a pattern that spans two words

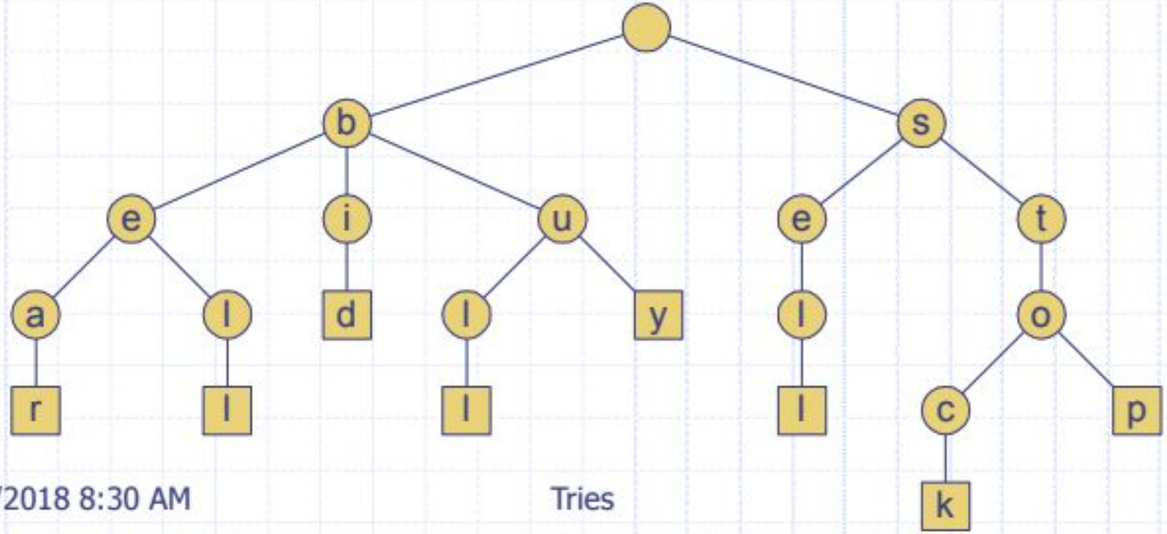
Word Matching with a Trie

- We insert the words of the text into a trie
- Each leaf stores the occurrences of the associated word in the text



Standard Trie Construction

- Assuming the input strings are words in the English language, how many children does the root node have?

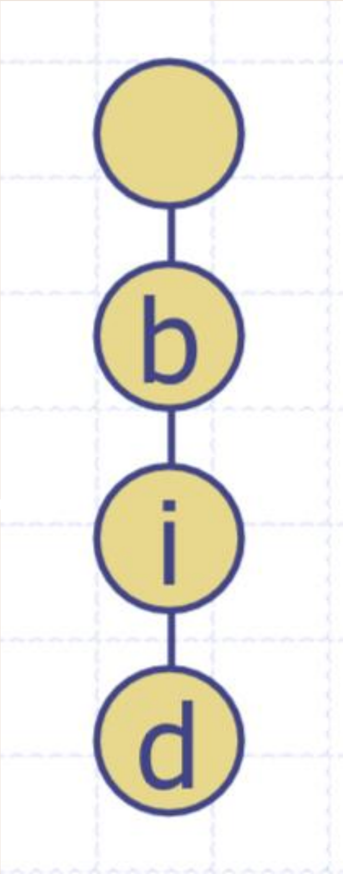


Constructing a Trie

- The number of children of the root node is the number of distinct first letters in all the words in the input string.
- EX: How many children would the root of a standard trie have given the following sets of strings?
    - **S** = {apple, aardvark, animal, awesome}
    - **S** = {xylophone, zebra, penguin, violin, yellow}
    - **S** = {CAGT, AGTC, GATC}
- What's the maximum possible given an alphabet of size **d**?

Example: Build a trie for **S** = {bid}

Example: Build a trie for **S** = {bid}

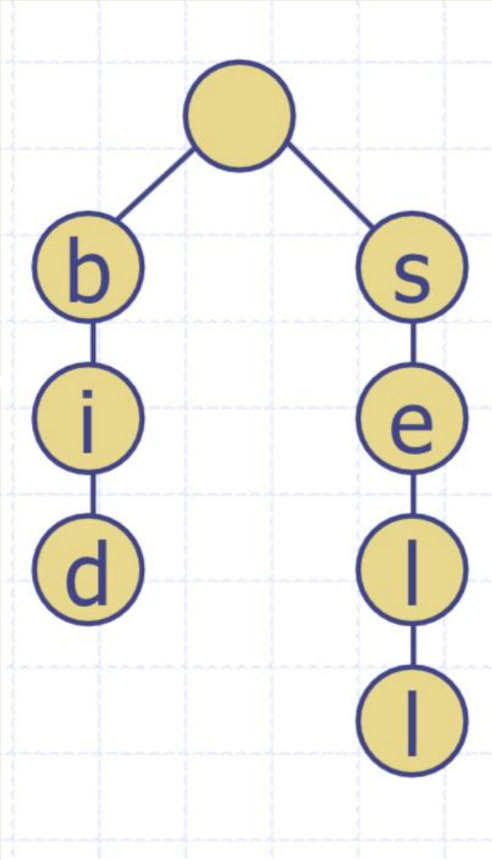


Picture from [1]

Example: add “sell”

$\mathbf{S} = \{\text{bid}, \text{sell}\}$

Example: add “sell”
S = {bid, sell}



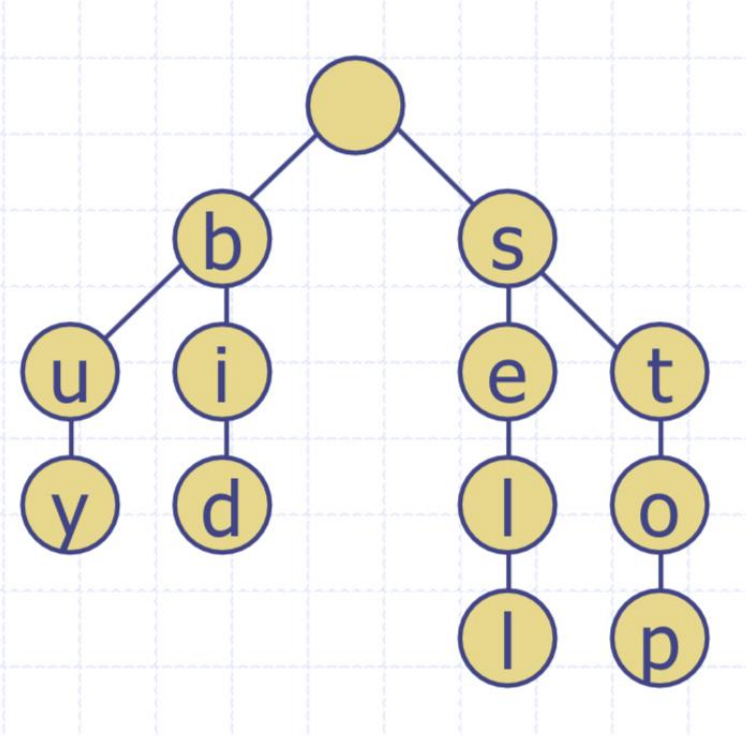
Picture from [1]

Example: Add “buy” and “stop”

S = {bid, sell, buy, stop}

Picture from [1]

Example: Add “buy” and “stop”
S = {bid, sell, buy, stop}

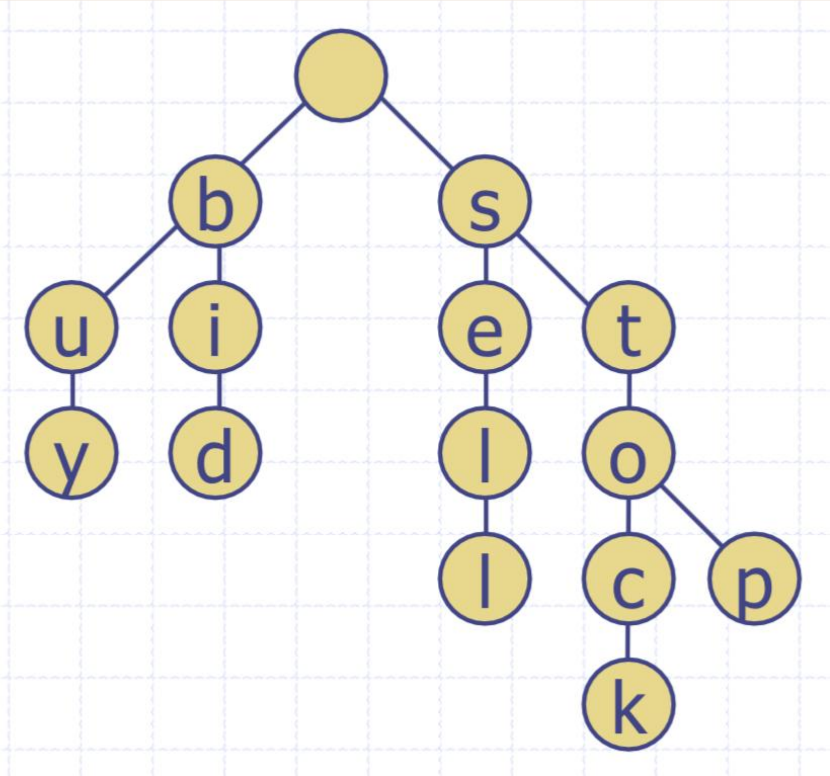


Picture from [1]

Example: Add “stock”
$\mathbf{S}$ = {bid, sell, buy, stop, stock}

Picture from [1]

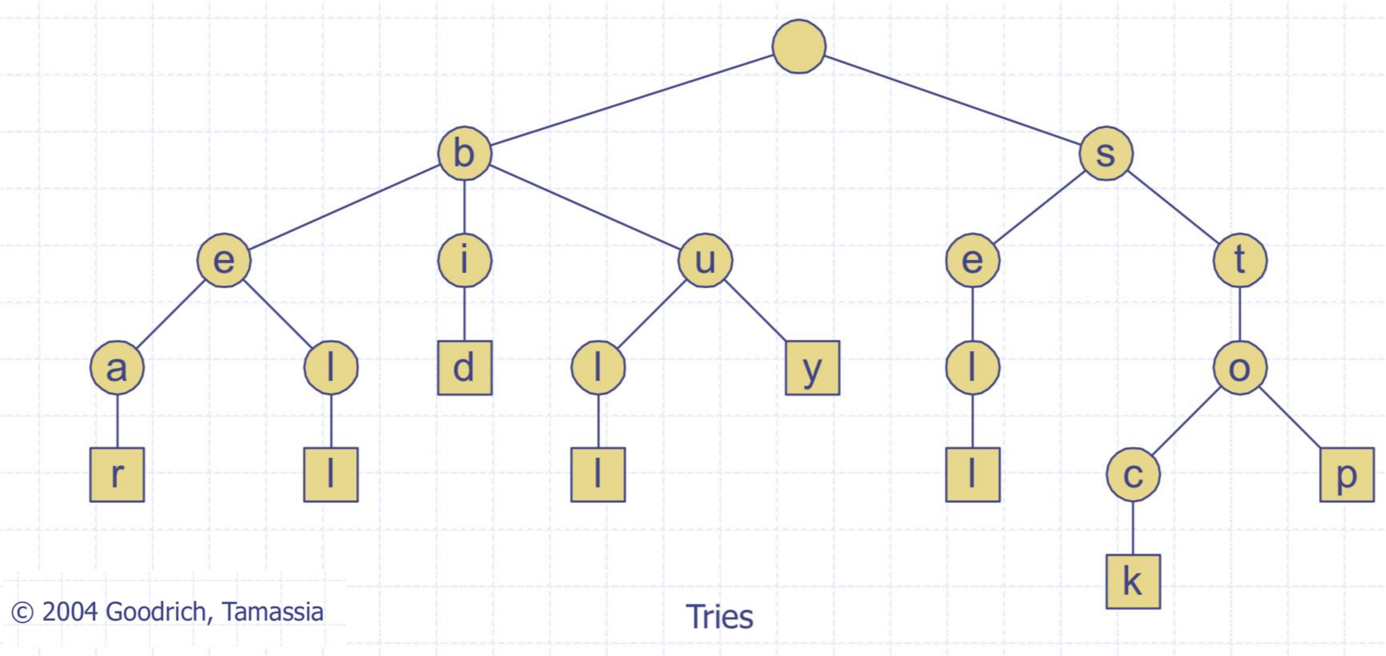
Example: Add "stock"
S = {bid, sell, buy, stop, stock}



Picture from [1]

Standard Trie Example

S = {bear, bell, bid, bull, buy, sell, stock, stop}



Standard Trie Construction

- Recall the assumption that no string in **S** is a prefix of another string in **S**.
- To insert a string **X** into a trie **T**:
  - try to trace the path associated with **X** in **T**
  - if you reach an external node, you have found **X**, so update the node to reflect the location of this instance
  - else, you are stopped at an internal node, and you must create a new chain of node descendants for the rest of **X**

Analysis

- What is the worst-case runtime to insert a single string **X** of length **m** into the trie?
- What is the total time required to insert all of the strings in **S** (where the total length of all strings in **S** is **n**)?
- What is the total space requirement for storing the strings in **S** in a standard trie?

Analysis

- What is the worst-case runtime to insert a single string **X** of length **m** into the trie?

  O(dm)

- What is the total time required to insert all of the strings in **S** (where the total length of all strings in **S** is **n**)?

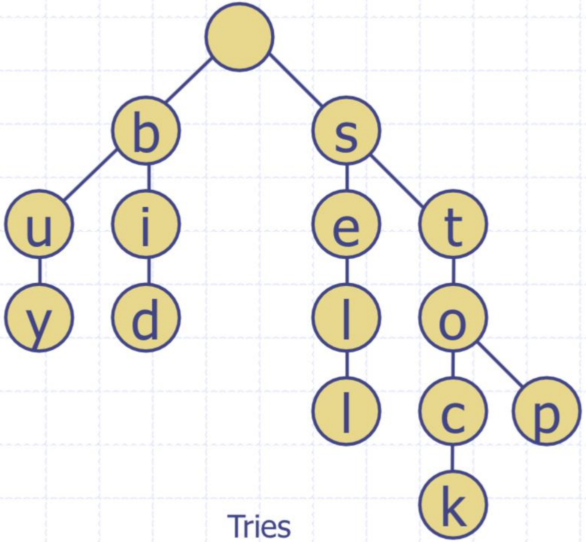
  O(dn)

- What is the total space requirement for storing the strings in **S** in a standard trie?

  O(n)

Improvements

- What comes to mind for tries?
  - e.g., is this entire tree really necessary?



Picture from [1]

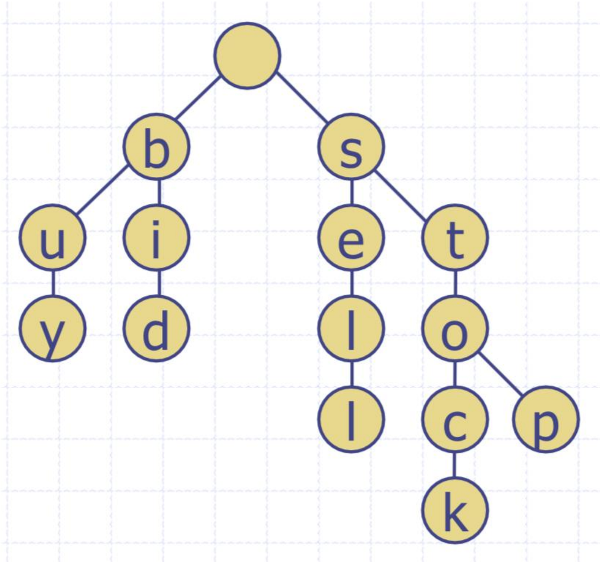
Compressed Tries

a.k.a. PATRICIA Tries (Practical AlgoriThm to Retrieve Information Coded In Alphanumeric

- **Motivation**: In standard tries, there are potentially a lot of nodes with only one child, which is a waste of space.
- **Main Idea**: Ensure that each internal node has *at least two children*.
- **Method**: Compress chains of single-child nodes into individual edges.

Definitions

- An internal node **v** of a standard trie **T** is ***redundant*** if **v** has one child and is not the root.
- How many ***redundant*** nodes does the trie below have?



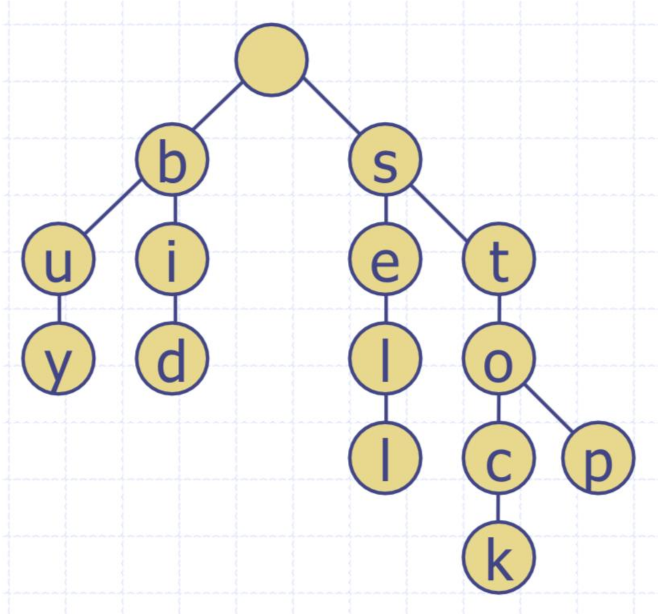
Picture from [1]

Definitions

- An internal node **v** of a standard trie **T** is ***redundant*** if **v** has one child and is not the root.
- How many ***redundant*** nodes does the trie to the right have?

  Answer: 6—{u, i, e, l, t, c}

- A chain of **k**>=2 edges, $(\mathbf{v_0}, \mathbf{v_1})(\mathbf{v_1},\mathbf{v_2})\ldots(\mathbf{v_{k-1}}, \mathbf{v_k})$ is ***redundant*** if:
  - $\mathbf{v_i}$ is redundant for **i**=1,...**k**-1
  - $\mathbf{v_0}$ and $\mathbf{v_k}$ are not redundant
- EX: At right, the chain **(s,e)(e,l)(l,l)** is redundant.

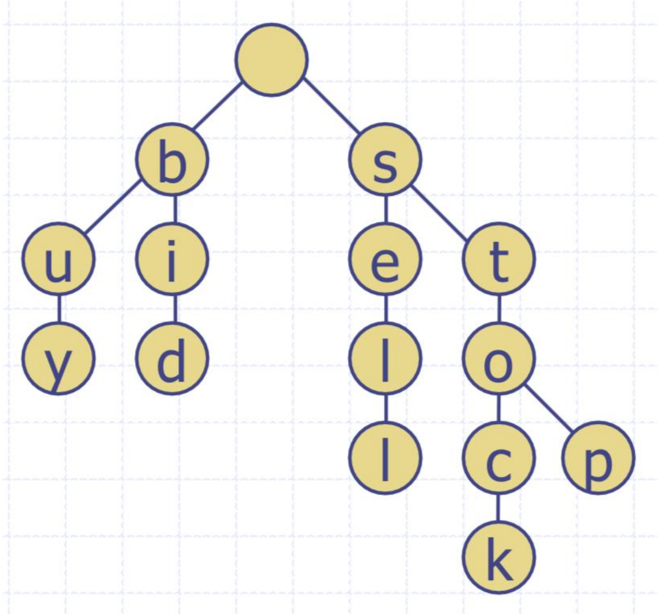


Picture from [1]

Converting Standard to Compressed Trie

- A chain of **k**>=2 edges, $(v_0, v_1)(v_1, v_2)\ldots(v_{k-1}, v_k)$ is ***redundant*** if:
  - v_i is redundant for **i**=1,...**k**-1
  - v_0 and v_k are not redundant

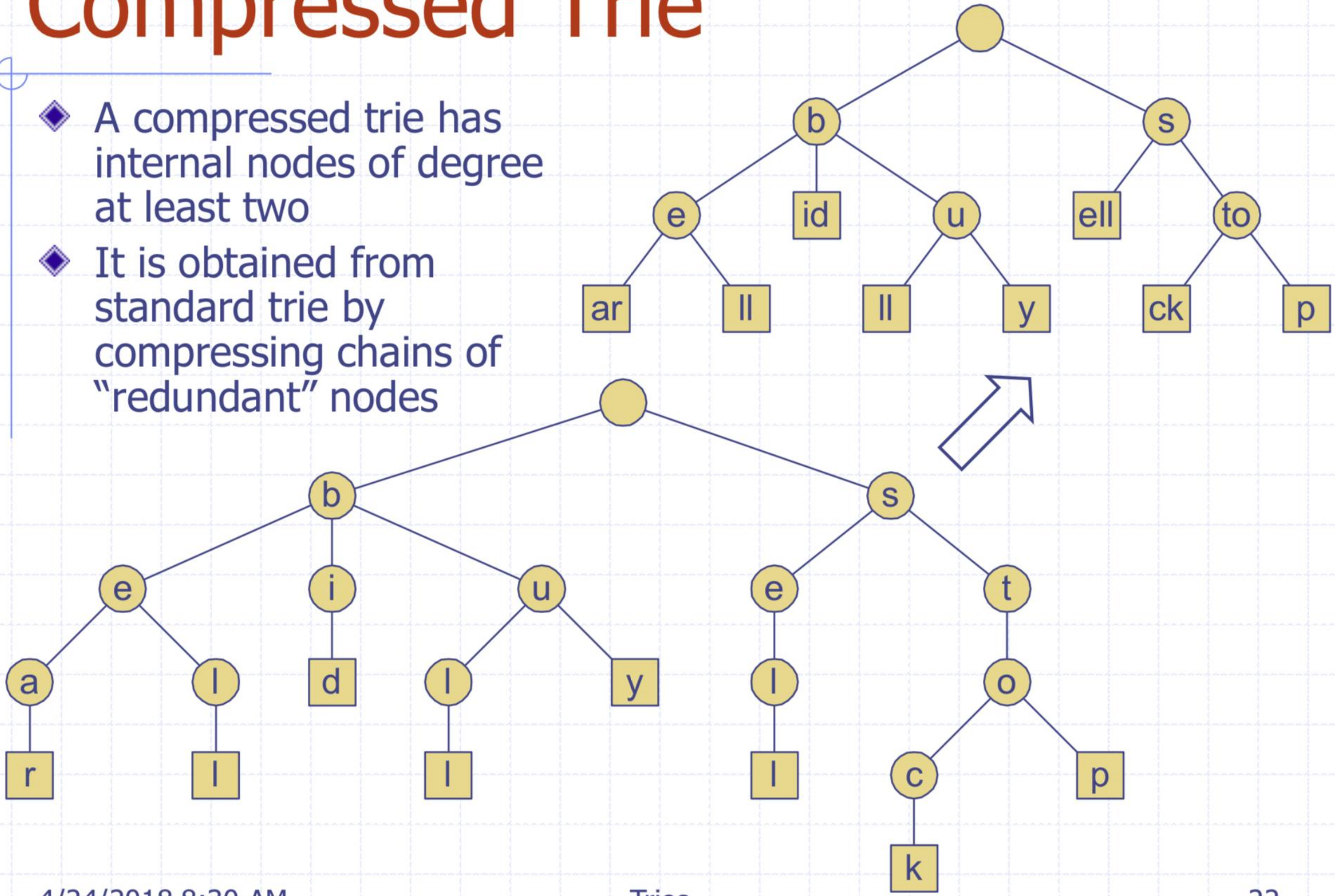
- Transform a standard trie **T** into a compressed trie **T'** by replacing each redundant chain $(v_0, v_1)(v_1, v_2)\ldots(v_{k-1}, v_k)$ of **k** >=2 edges into a single edge (v_0, v_k)



Picture from [1]

Compressed Trie

- A compressed trie has internal nodes of degree at least two
- It is obtained from standard trie by compressing chains of "redundant" nodes



Picture from [1]

Analysis

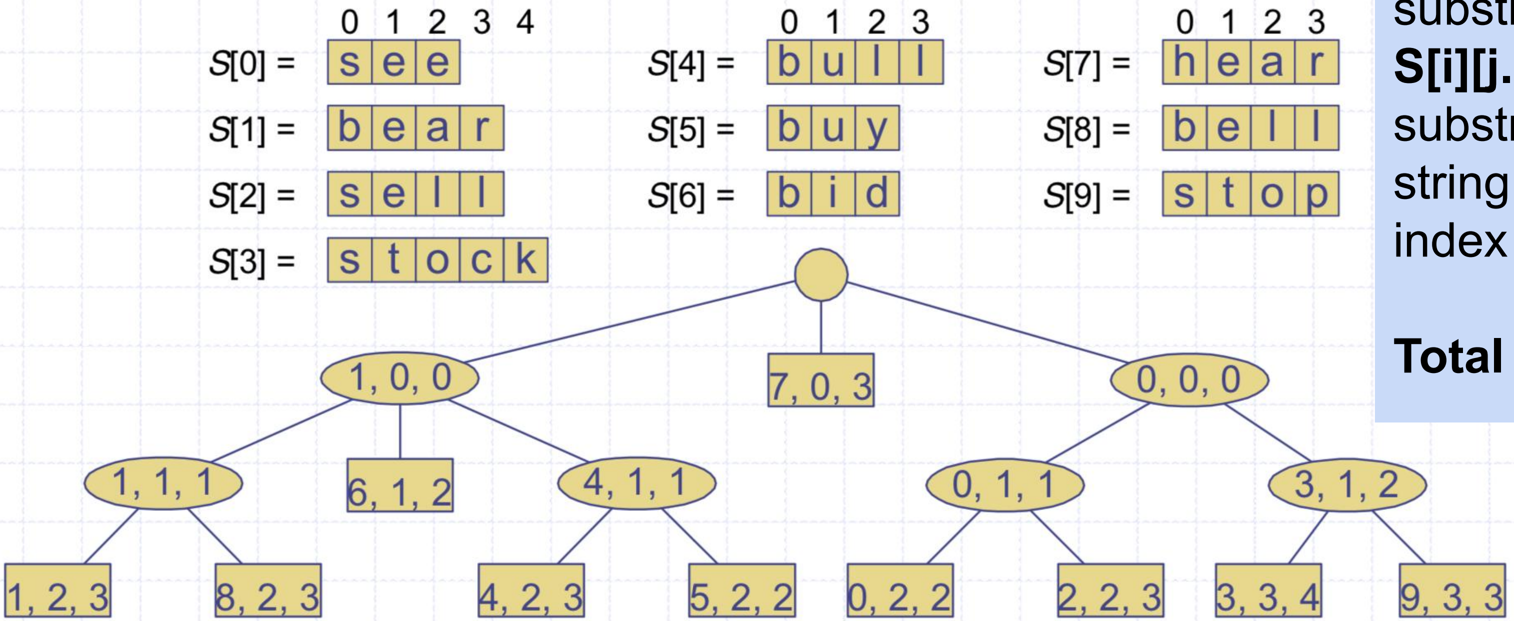
A compressed trie **T** storing a collection **S** of **s** strings from an alphabet of size **d** has the following properties:

- Every internal node of **T** has at least two children and at most **d** children.
- **T** has **s** external nodes.
- The number of nodes of **T** is **O(s)**.

Won’t the advantage gained from compressing the paths be offset by the corresponding expansion of the node labels?

Compact Representation

- Compact representation of a compressed trie for an array of strings:
  - Stores at the nodes ranges of indices instead of substrings
  - Serves as an auxiliary index structure

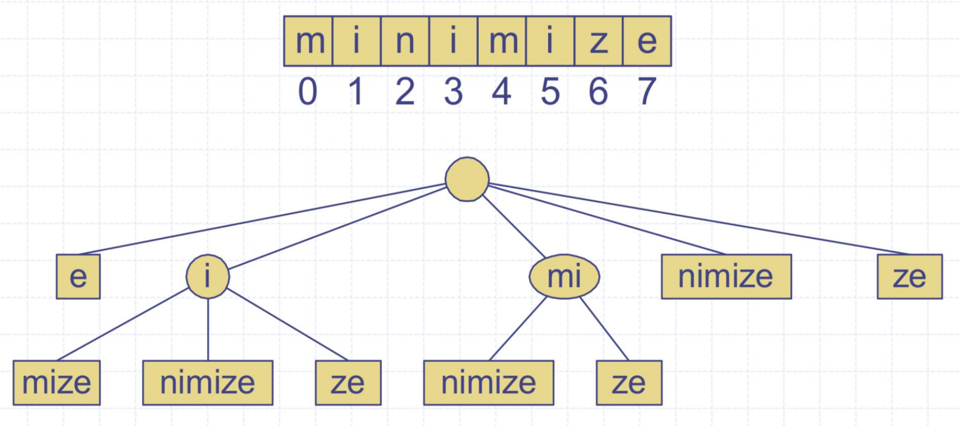


The triple **(i, j, k)** refers to the substring **S[i][j...k]**—i.e. a substring of the **i**th string in **S** from index **j** to index **k.**

Total space: O(s)

Picture from [1]

Suffix Tries
(a.k.a. **suffix tree** or **position tree**)

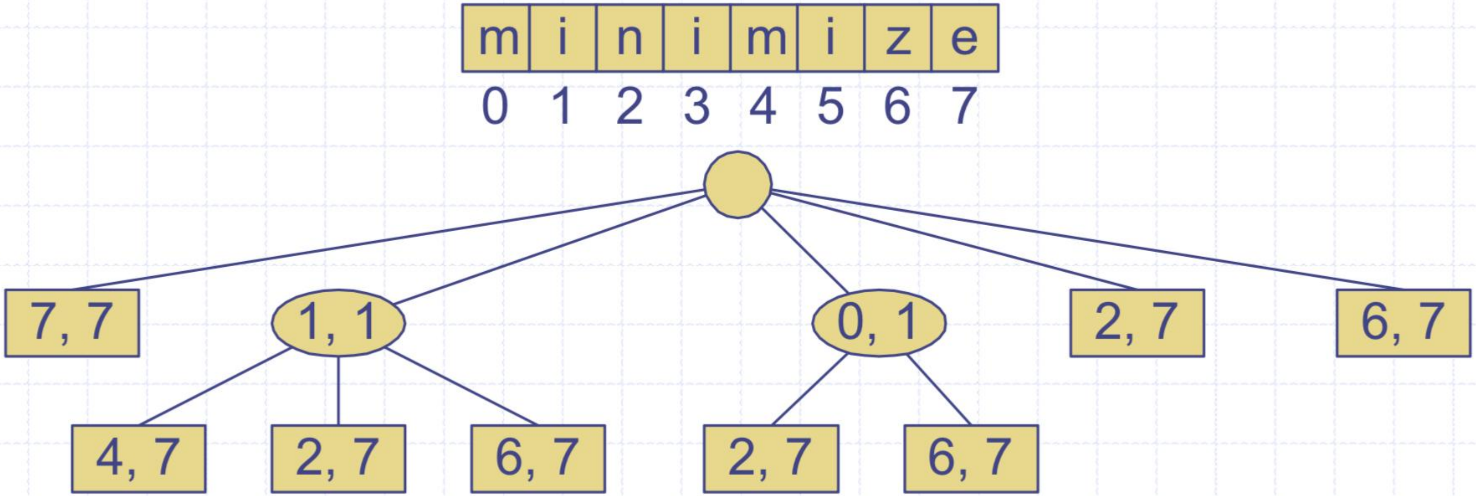


- **Motivation**: The previous tries don't allow efficient search of arbitrary substrings.
- **Main Idea**: Given a string **X**, a **suffix trie** of **X** is a compressed trie of all the suffixes of **X**.

Picture from [1]

Suffix Trie

- Compact representation of the suffix trie for a string X of size n from an alphabet of size d
  - Uses $O(n)$ space
  - Supports arbitrary pattern matching queries in X in $O(dm)$ time, where m is the size of the pattern
  - Repetitive words not stored repetitively



Picture from [1]

Saving Space

Given a string **X** of length **n**, what is the total length of the suffix strings of **X**?

Saving Space with Compact Representation

Given a string **X** of length **n**, what is the total length of the suffix strings of **X**?

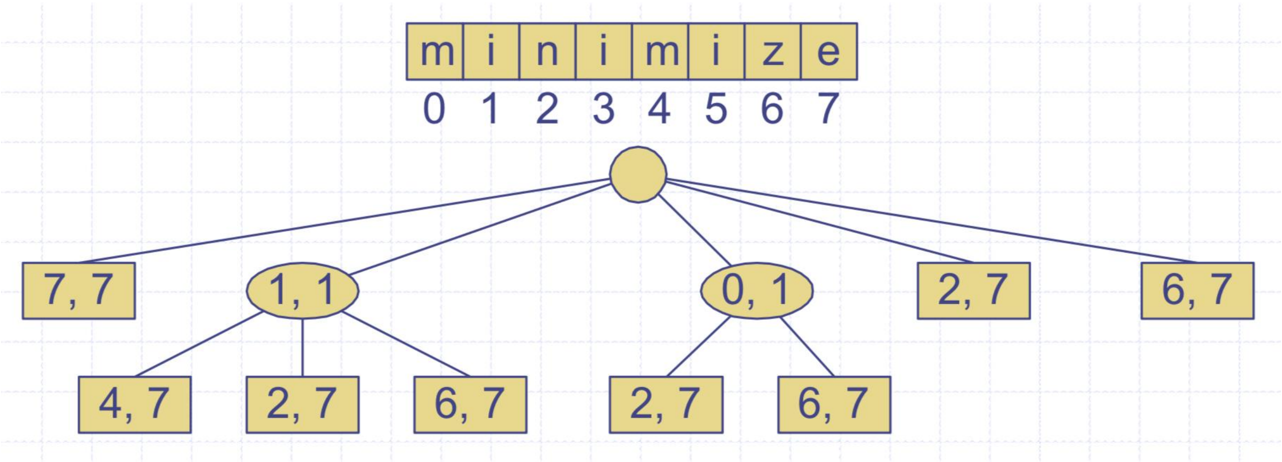
$$\mathbf{1 + 2 + \ldots + n = n(n+1)/2},$$

which is $\mathbf{O(n^2)}$...

Claim: A suffix trie using the compact representation of indices rather than strings stores them in $\mathbf{O(n)}$ space. Construction of such a suffix trie can be done in $\mathbf{O(n)}$ time.

Using a Suffix Trie

- A suffix try of a string **X** can be used to efficiently search for an arbitrary pattern **P** in **X**.
- **P** is a substring in **X** if such a path can be traced.
- Search time: **O(dm)**



Picture from [1]

References

[1] Tamassia and Goodrich