Text Indexing

Lecture 01: Tries
Florian Kurpicz
Definition: Text

- let $\Sigma$ be an alphabet
- $T \in \Sigma^*$ is a text
- $|T| = n$ is the length of the string
Definition: Text

- let $\Sigma$ be an alphabet
- $T \in \Sigma^*$ is a text
- $|T| = n$ is the length of the string

Definition: Alphabet Types

- **constant size alphabet**: finite set not depending on $n$
- **integer alphabet**: alphabet is $\{1, \ldots, \sigma\}$ and fits into constant number of words
- **finite alphabets**: alphabet of finite size
Definition: Substring, Prefix, and Suffix

Given a text $T = T[1] ... T[n]$ of length $n$:

- $T[i..j] = T[i] ... T[j]$ is called a **substring**.

```
 a b b a a b b a $
```

**Sentinel for Simplicity**

Given a text $T$ of length $n$ over an alphabet $\Sigma$, we assume that $T[n] = $ with $\neq \alpha$ for all $\alpha \in \Sigma$ otherwise, suffix can be prefix of another suffix.

**Definition: Prefix-Free**

A string is **prefix-free** if no suffix is a prefix of another suffix.
Given a text \( T = T[1]T[2] \ldots T[n] \) of length \( n \):

- \( T[i..j] = T[i] \ldots T[j] \) is called a **substring**,

  \[
  \begin{array}{c}
  a \ b \ b \ a \ a \ b \ b \ a \$
  \end{array}
  \]

- \( T[1..i] \) is called a **prefix**, and

  \[
  \begin{array}{c}
  a \ b \ b \ a \ a \ b \ b \ a \$
  \end{array}
  \]
Definition: Substring, Prefix, and Suffix


- $T[i..j] = T[i] \ldots T[j]$ is called a substring,
  
  \[
  \begin{array}{cccccccc}
  a & b & b & a & a & b & b & a & $ \\
  \end{array}
  \]

- $T[1..i]$ is called a prefix, and
  
  \[
  \begin{array}{cccccccc}
  a & b & b & a & a & b & b & a & $ \\
  \end{array}
  \]

- $T[i..n]$ is called a suffix of $T$.
  
  \[
  \begin{array}{cccccccc}
  a & b & b & a & a & b & b & a & $ \\
  \end{array}
  \]
Definition: Substring, Prefix, and Suffix

- $T[i..j] = T[i] \ldots T[j]$ is called a substring,
  
  \[
  \begin{array}{cccccc}
  a & b & b & a & a & b & b & a & \$
  \end{array}
  \]

- $T[1..i]$ is called a prefix, and
  
  \[
  \begin{array}{cccccc}
  a & b & b & a & a & b & b & a & \$
  \end{array}
  \]

- $T[i..n]$ is called a suffix of $T$.
  
  \[
  \begin{array}{cccccc}
  a & b & b & a & a & b & b & a & \$
  \end{array}
  \]

Sentinel for Simplicity

Given a text $T$ of length $n$ over an alphabet $\Sigma$.
- we assume that $T[n] = \$\$ with
- $\$ \not\in \Sigma$ and $\$ < $\alpha$ for all $\alpha \in \Sigma$
Definition: Substring, Prefix, and Suffix


- $T[i..j] = T[i] \ldots T[j]$ is called a **substring**, and
- $T[1..i]$ is called a **prefix**, and
- $T[i..n]$ is called a **suffix** of $T$.

Sentinel for Simplicity

Given a text $T$ of length $n$ over an alphabet $\Sigma$. We assume that $T[n] = $ with
- $\notin \Sigma$ and $ \prec \alpha$ for all $\alpha \in \Sigma$. 

Definition: Substring, Prefix, and Suffix

Given a text \( T = T[1]T[2] \ldots T[n] \) of length \( n \):

- \( T[i..j] = T[i] \ldots T[j] \) is called a **substring**.
  
  ![Substring Example](image)

- \( T[1..i] \) is called a **prefix**, and
  
  ![Prefix Example](image)

- \( T[i..n] \) is called a **suffix** of \( T \).
  
  ![Suffix Example](image)

Sentinel for Simplicity

Given a text \( T \) of length \( n \) over an alphabet \( \Sigma \):

- we assume that \( T[n] = \$ \) with
  
  ![Sentinel Example](image)

- \( \$ \notin \Sigma \) and \( \$ < \alpha \) for all \( \alpha \in \Sigma \)

- otherwise, suffix can be prefix of another suffix
  
  ![Suffix Prefix Example](image)

- \( T[1..n] = abbaabba \) and \( T[5..n] = abba \)
Preliminaries (2/2)

Definition: Substring, Prefix, and Suffix

- $T[i..j] = T[i] \ldots T[j]$ is called a **substring**.
- $T[1..i]$ is called a **prefix**, and $T[i..n]$ is called a **suffix** of $T$.

```
abaababa
```

Sentinel for Simplicity

Given a text $T$ of length $n$ over an alphabet $\Sigma$.
- we assume that $T[n] = \$ \in \Sigma$ and $\$ < \alpha$ for all $\alpha \in \Sigma$
- otherwise, suffix can be prefix of another suffix

```
1 2 3 4 5 6 7 8
abaababa
```

Definition: Prefix-Free

A string is **prefix-free** if no suffix is a prefix of another suffix

```
abaababa = T[1..n] and abba = T[5..n]
```
String Dictionary

Given a set \( S \subseteq \Sigma^* \) of prefix-free strings, we want to answer:

- is \( x \in \Sigma^* \) in \( S \)
- add \( x \notin S \) to \( S \)
- remove \( x \in S \) from \( S \)
- predecessor and successor of \( x \in \Sigma^* \) in \( S \)
Given a set $S \subseteq \Sigma^*$ of prefix-free strings, we want to answer:

- is $x \in \Sigma^*$ in $S$
- add $x \notin S$ to $S$
- remove $x \in S$ from $S$
- predecessor and successor of $x \in \Sigma^*$ in $S$

**Definition: Trie**

Given a set $S = \{S_1, \ldots, S_k\}$ of prefix-free strings, a trie is a labeled rooted tree $G = (V, E)$ with:

1. $k$ leaves
2. $\forall S_i \in S$ there is a path from the root to a leaf, such that the concatenation of the labels is $S_i$
3. $\forall v \in V$ the labels of the edges $(v, \cdot)$ are unique
Given a set $S \subseteq \Sigma^*$ of prefix-free strings, we want to answer:
- is $x \in \Sigma^*$ in $S$
- add $x \notin S$ to $S$
- remove $x \in S$ from $S$
- predecessor and successor of $x \in \Sigma^*$ in $S$

**Definition: Trie**

Given a set $S = \{S_1, \ldots, S_k\}$ of prefix-free strings, a trie is a labeled rooted tree $G = (V, E)$ with:

1. $k$ leaves
2. $\forall S_i \in S$ there is a path from the root to a leaf, such that the concatenation of the labels is $S_i$
3. $\forall v \in V$ the labels of the edges $(v, \cdot)$ are unique

$S = \{\text{bear}, \text{bee}, \text{cab}, \text{car}\}$
Queries: Insert, Contains, and Delete a Pattern

Same for all
- start at root and follow existing children

Contains
- is leaf found and whole pattern is matched

Delete
- if leaf is found backtrack and delete unique path
  - otherwise not found

Insert
- insert rest of pattern prefix-free

S = \{bear, bee, cab, car\}
Queries: Insert, Contains, and Delete a Pattern

**Same for all**
- start at root and follow existing children

**Contains**
- is leaf found and whole pattern is matched

**Delete**
- if leaf is found backtrack and delete unique path
- otherwise not found

**Insert**
- insert rest of pattern **prefix-free**

\[ S = \{ \text{bear, bee, cab, car} \} \]
- is cab in \( S \)

\[
\begin{array}{c}
\text{a} \\
\text{e} \\
\text{r} \\
\text{b} \\
\text{c} \\
\text{a} \\
\text{e} \\
\text{b} \\
\text{r} \\
\end{array}
\]
Queries: Insert, Contains, and Delete a Pattern

Same for all
- start at root and follow existing children

Contains
- is leaf found and whole pattern is matched

Delete
- if leaf is found backtrack and delete unique path
  - otherwise not found

Insert
- insert rest of pattern prefix-free

S = \{bear, bee, cab, car\}
- is cab in S
Queries: Insert, Contains, and Delete a Pattern

**Same for all**
- start at root and follow existing children

**Contains**
- is leaf found and whole pattern is matched

**Delete**
- if leaf is found backtrack and delete unique path
  - otherwise not found

**Insert**
- insert rest of pattern \( \text{prefix-free} \)

\[ S = \{ \text{bear, bee, cab, car} \} \]
- is cab in \( S \)
Queries: Insert, Contains, and Delete a Pattern

**Same for all**
- start at root and follow existing children

**Contains**
- is leaf found and whole pattern is matched

**Delete**
- if leaf is found backtrack and delete unique path
- otherwise not found

**Insert**
- insert rest of pattern \(\text{prefix-free}\)

\[ S = \{\text{bear, bee, cab, car}\} \]
- is cab in \(S\)
- remove bear from \(S\)
Queries: Insert, Contains, and Delete a Pattern

Same for all
- start at root and follow existing children

Contains
- is leaf found and whole pattern is matched

Delete
- if leaf is found backtrack and delete unique path
  - otherwise not found

Insert
- insert rest of pattern — prefix-free

S = \{bear, bee, cab, car\}
- is cab in S
- remove bear from S
Queries: Insert, Contains, and Delete a Pattern

Same for all
- start at root and follow existing children

Contains
- is leaf found and whole pattern is matched

Delete
- if leaf is found backtrack and delete unique path
  - otherwise not found

Insert
- insert rest of pattern
- prefix-free

\[ S = \{ \text{bear, bee, cab, car} \} \]
- is cab in \( S \)
- remove \text{bear} from \( S \)
Queries: Insert, Contains, and Delete a Pattern

Same for all
- start at root and follow existing children

Contains
- is leaf found and whole pattern is matched

Delete
- if leaf is found backtrack and delete unique path
  otherwise not found

Insert
- insert rest of pattern \( \text{prefix-free} \)

\[ S = \{\text{bear, bee, cab, car}\} \]
- is \( \text{cab} \) in \( S \)
- remove \( \text{bear} \) from \( S \)
Queries: Insert, Contains, and Delete a Pattern

**Same for all**
- start at root and follow existing children

**Contains**
- is leaf found and whole pattern is matched

**Delete**
- if leaf is found backtrack and delete unique path
  - otherwise not found

**Insert**
- insert rest of pattern ☑️ prefix-free

\[ S = \{ \text{bear, bee, cab, car} \} \]
- is cab in \( S \)
- remove bear from \( S \)
- how can we find the predecessor of can?
Why Prefix-Free

- insert beer
Why Prefix-Free

- insert beer
Why Prefix-Free

- insert beer
- bee cannot be found
Why Prefix-Free

- insert `beer`
- `bee` cannot be found
- remember which node refers to a string
Why Prefix-Free

- insert beer
- bee cannot be found
- remember which node refers to a string
- or (much preferred) make strings prefix free
Next Steps

**Setting**

- alphabet $\Sigma$ of size $\sigma$
- $k$ strings $\{s_1, \ldots, s_k\}$ over the alphabet $\Sigma$
- total size of strings is $N = \sum_{i=1}^{k} |s_i|$
- queries ask for pattern $P$ of length $m$
Next Steps

Setting

- alphabet $\Sigma$ of size $\sigma$
- $k$ strings $\{s_1, \ldots, s_k\}$ over the alphabet $\Sigma$
- total size of strings is $N = \sum_{i=1}^{k} |s_i|$
- queries ask for pattern $P$ of length $m$

We Want to Know

- query times
- space requirements
Next Steps

Setting
- alphabet $\Sigma$ of size $\sigma$
- $k$ strings $\{s_1, \ldots, s_k\}$ over the alphabet $\Sigma$
- total size of strings is $N = \sum_{i=1}^{k} |s_i|$ 
- queries ask for pattern $P$ of length $m$

We Want to Know
- query times
- space requirements

- both depend on the representation of children
- look at different representations
Next Steps

Setting
- alphabet $\Sigma$ of size $\sigma$
- $k$ strings $\{s_1, \ldots, s_k\}$ over the alphabet $\Sigma$
- total size of strings is $N = \sum_{i=1}^{k} |s_i|$
- queries ask for pattern $P$ of length $m$

We Want to Know
- query times
- space requirements
- both depend on the representation of children
- look at different representations
Next Steps

**Setting**
- alphabet $\Sigma$ of size $\sigma$
- $k$ strings $\{s_1, \ldots, s_k\}$ over the alphabet $\Sigma$
- total size of strings is $N = \sum_{i=1}^k |s_i|$ 
- queries ask for pattern $P$ of length $m$

**We Want to Know**
- query times
- space requirements
- both depend on the representation of children
- look at different representations
Next Steps

Setting
- alphabet $\Sigma$ of size $\sigma$
- $k$ strings $\{s_1, \ldots, s_k\}$ over the alphabet $\Sigma$
- total size of strings is $N = \sum_{i=1}^{k} |s_i|$
- queries ask for pattern $P$ of length $m$

We Want to Know
- query times
- space requirements

both depend on the representation of children
look at different representations
Arrays of Variable Size

- store children (character and pointer) in the order they are added
- to find child scan array
- to delete child swap with last and remove last
- children are not ordered
- 📚 PINGO query time?

![Diagram of a tree structure with nodes labeled V1 to V7 and children C1 to C7. Each node is connected to its children, illustrating the tree structure.](image-url)
Arrays of Variable Size

- store children (character and pointer) in the order they are added
- to find child scan array
- to delete child swap with last and remove last
  children are not ordered

PINGO query time?

Query Time (Contains)

- $O(m \cdot \sigma)$
Arrays of Variable Size

- store children (character and pointer) in the order they are added
- to find child scan array
- to delete child swap with last and remove last
- children are not ordered
- PINGO query time?

Query Time (Contains)
- $O(m \cdot \sigma)$

Space
- $O(N)$ words
Arrays of Fixed Size

- children (pointer) are stored in arrays of size $\sigma$
- use null to mark non-existing children
- finding and deleting children is trivial

PINGO query time?
Arrays of Fixed Size

- children (pointer) are stored in arrays of size $\sigma$
- use null to mark non-existing children
- finding and deleting children is trivial
- PINGO query time?

Query Time (Contains)

- $O(m)$ \(\downarrow\) optimal
Arrays of Fixed Size

- children (pointer) are stored in arrays of size $\sigma$
- use null to mark non-existing children
- finding and deleting children is trivial
- PINGO query time?

Query Time (Contains)

- $O(m)$ ⬤ optimal

Space

- $O(N \cdot \sigma)$ words ⬤ very bad
Hash Tables

- either use a hash table per node
  - has overhead
- or use global hash table for whole trie
- `PINGO` query time?
Hash Tables

- either use a hash table per node
  - has overhead
- or use global hash table for whole trie
- **PINGO** query time?

Query Time (Contains)

- $O(m)$ w.h.p.
Hash Tables

- either use a hash table per node
  - has overhead
- or use global hash table for whole trie

PINGO query time?

Query Time (Contains)
- $O(m)$ w.h.p.

Space
- $O(N)$ words
Balanced Search Trees

- children are stored in balanced search trees
- e.g., AVL tree, red-black tree, ...
- in static setting sorted array and binary search
- PINGO query time?

Query Time (Contains)\[ O\left( m \cdot \log \sigma \right) \]

Space\[ O\left( N \right) \] words
Balanced Search Trees

- children are stored in balanced search trees
- e.g., AVL tree, red-black tree, . . .
- in static setting sorted array and binary search
- PINGO query time?

Query Time (Contains)

- $O(m \cdot \lg \sigma)$
Balanced Search Trees

- Children are stored in balanced search trees
- E.g., AVL tree, red-black tree, ...
- In static setting sorted array and binary search

PINGO query time?

Query Time (Contains)
- $O(m \cdot \lg \sigma)$

Space
- $O(N)$ words
Weight-Balanced Search Trees (1/2)
Weight-Balanced Search Trees (1/2)

$w_i = \# \text{ leaves below } v_i$
Weight-Balanced Search Trees (1/2)

\[ w_i = \# \text{ leaves below } v_i \]
Weight-Balanced Search Trees (1/2)

\[ w_i = \# \text{ leaves below } v_i \]
Weight-Balanced Search Trees (1/2)

$w_i = \#$ leaves below $v_i$
Weight-Balanced Search Trees (1/2)

$w_i = \# \text{ leaves below } v_i$
Weight-Balanced Search Trees (1/2)

$w_i = \# \text{ leaves below } v_i$
Weight-Balanced Search Trees (1/2)

\[ w_i = \# \text{ leaves below } v_i \]
Weight-Balanced Search Trees (1/2)

$w_i = \# \text{ leaves below } v_i$
Weight-Balanced Search Trees (1/2)

\[ w_i = \# \text{ leaves below } v_i \]
Weight-Balanced Search Trees (1/2)

$w_i = \# \text{ leaves below } v_i$
Weight-Balanced Search Trees (2/2)

- use weight-balanced search trees at each node
- **PINGO** query time?

\[ w_i = \# \text{ leaves below } v_i \]
Weight-Balanced Search Trees (2/2)

- use weight-balanced search trees at each node
- PINGO query time?

Query Time (Contains)
- $O(m + \lg k)$
- match character of pattern
- or halve number of strings

\[
w_i = \# \text{ leaves below } v_i
\]
Weight-Balanced Search Trees (2/2)

- use weight-balanced search trees at each node
- PINGO query time?

Query Time (Contains)
- \(O(m + \lg k)\)
- match character of pattern
- or halve number of strings

Space
- \(O(N)\) words

\[ w_i = \# \text{ leaves below } v_i \]
Two-Levels with Weight-Balanced Search Trees

- split tree into upper and lower half
- lower half deepest nodes such that subtrees have size $O(\sigma)$
- weight-balanced search trees for lower half
- fixed-size arrays in upper half branching nodes only
- PINGO query time?
Two-Levels with Weight-Balanced Search Trees

- split tree into upper and lower half
- lower half deepest nodes such that subtrees have size $O(\sigma)$
- weight-balanced search trees for lower half
- fixed-size arrays in upper half
- branching nodes only
- PINGO query time?

Query Time (Contains)

- upper half: $O(m)$
- lower half: $O(m + \lg \sigma)$
- total: $O(m + \lg \sigma)$
Two-Levels with Weight-Balanced Search Trees

- split tree into upper and lower half
- lower half deepest nodes such that subtrees have size $O(\sigma)$
- weight-balanced search trees for lower half
- fixed-size arrays in upper half branching nodes only
- PINGO query time?

Query Time (Contains)

- upper half: $O(m)$
- lower half: $O(m + \lg \sigma)$
- total: $O(m + \lg \sigma)$

Space

- upper half: $O(N)$ words
  - $O(N/\sigma)$ branching nodes
- lower half: $O(N)$ words
- total: $O(N)$ words
# Theoretical Comparison

<table>
<thead>
<tr>
<th>Representation</th>
<th>Query Time (Contains)</th>
<th>Space in Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrays of variable size</td>
<td>$O(m \cdot \sigma)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>arrays of fixed size</td>
<td>$O(m)$</td>
<td>$O(N \cdot \sigma)$</td>
</tr>
<tr>
<td>hash tables</td>
<td>$O(m)$ w.h.p.</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>balanced search trees</td>
<td>$O(m \cdot \lg \sigma)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>weight-balanced search trees</td>
<td>$O(m + \log k)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>two-levels with weight-balanced search trees</td>
<td>$O(m + \log \sigma)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
Compact Trie

- tries have unnecessary nodes
- branchless paths can be removed
- edge labels can consist of multiple characters
Compact Trie

- tries have unnecessary nodes
- branchless paths can be removed
- edge labels can consist of multiple characters

**Definition: Compact Trie**

- A compact trie is a trie where all branchless paths are replaced by a single edge.
- The label of the new edge is the concatenation of the replaced edges’ labels.
Compact Trie

- tries have unnecessary nodes
- branchless paths can be removed
- edge labels can consist of multiple characters

**Definition: Compact Trie**

- A compact trie is a trie where all branchless paths are replaced by a single edge.
- The label of the new edge is the concatenation of the replaced edges’ labels.
Conclusion and Outlook

This Lecture
- dictionaries
- tries with different space-time trade-off
## Conclusion and Outlook

**This Lecture**
- dictionaries
- tries with different space-time trade-off

**Next Lecture**
- suffix trees and suffix arrays
- no lecture on Halloween(!)
- next lecture 07.11.2022