Karlsruhe Institute of Technology

Text Indexing

Lecture 01: Tries

Florian Kurpicz

String Dictionary

Given a set $S \subseteq \Sigma^\star$ of prefix-free strings, we want to answer:

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

String Dictionary

Given a set $S \subseteq \Sigma^\star$ of prefix-free strings, we want to answer:

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

String Dictionary

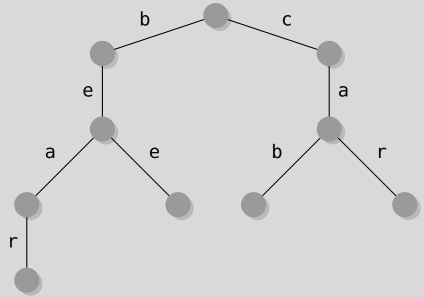
Given a set $S \subseteq \Sigma^{\star}$ of prefix-free strings, we want to answer:

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

Definition: Trie

Given a set $S = \{S_1, \dots, S_k\}$ 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



$S = \{\mathtt{bear}, \mathtt{bee}, \mathtt{cab}, \mathtt{car}\}$

Queries: Insert, Contains, and Delete a Pattern

Same for all

- start at root and follow existing children

Contains

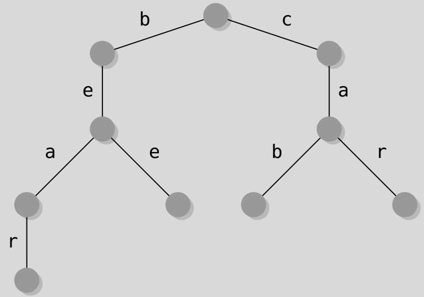
- is leaf found and whole pattern is matched

Delete

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

Insert

- insert rest of pattern ℹ prefix-free



$S = \{\texttt{bear}, \texttt{bee}, \texttt{cab}, \texttt{car}\}$

Queries: Insert, Contains, and Delete a Pattern

Same for all

- start at root and follow existing children

Contains

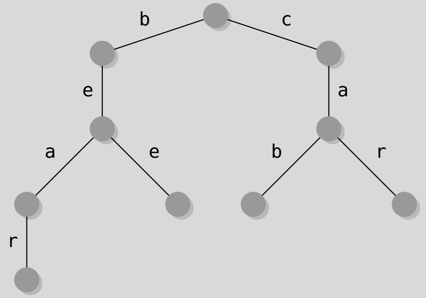
- is leaf found and whole pattern is matched

Delete

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

Insert

- insert rest of pattern ℹ prefix-free



$S = \{\texttt{bear}, \texttt{bee}, \texttt{cab}, \texttt{car}\}$

- is cab in S

Queries: Insert, Contains, and Delete a Pattern

Same for all

- start at root and follow existing children

Contains

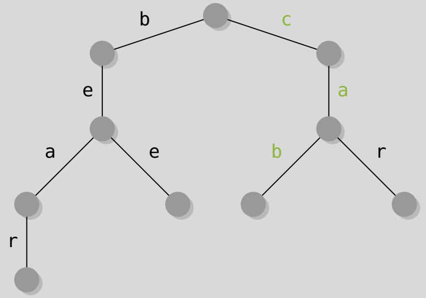
- is leaf found and whole pattern is matched

Delete

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

Insert

- insert rest of pattern ℹ prefix-free



$S = \{\texttt{bear}, \texttt{bee}, \texttt{cab}, \texttt{car}\}$

- is cab in S

Queries: Insert, Contains, and Delete a Pattern

Same for all

- start at root and follow existing children

Contains

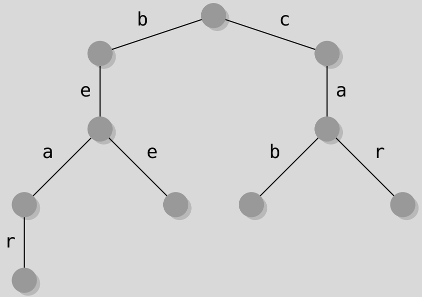
- is leaf found and whole pattern is matched

Delete

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

Insert

- insert rest of pattern prefix-free



$S = \{\texttt{bear}, \texttt{bee}, \texttt{cab}, \texttt{car}\}$

- is cab in S

Queries: Insert, Contains, and Delete a Pattern

Same for all

- start at root and follow existing children

Contains

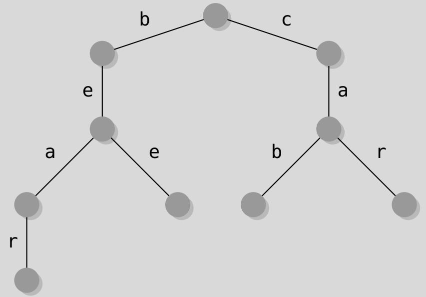
- is leaf found and whole pattern is matched

Delete

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

Insert

- insert rest of pattern ℹ prefix-free



$S = \{\texttt{bear}, \texttt{bee}, \texttt{cab}, \texttt{car}\}$

- 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

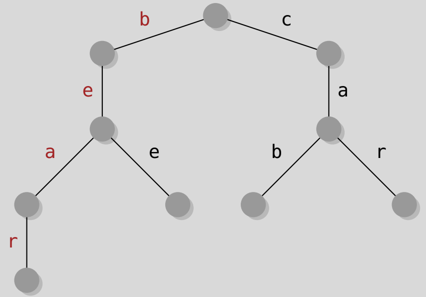
- is leaf found and whole pattern is matched

Delete

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

Insert

- insert rest of pattern ℹ prefix-free



$S = \{\texttt{bear}, \texttt{bee}, \texttt{cab}, \texttt{car}\}$

- 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

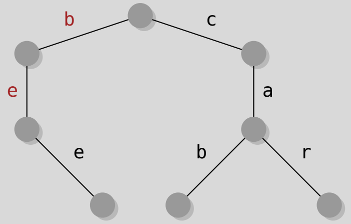
- is leaf found and whole pattern is matched

Delete

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

Insert

- insert rest of pattern ℹ prefix-free



$S = \{\texttt{bear}, \texttt{bee}, \texttt{cab}, \texttt{car}\}$

- 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

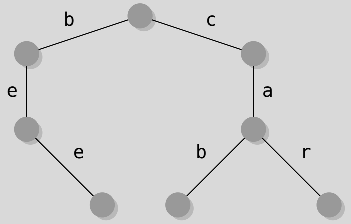
- is leaf found and whole pattern is matched

Delete

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

Insert

- insert rest of pattern ℹ prefix-free



$S = \{\texttt{bear}, \texttt{bee}, \texttt{cab}, \texttt{car}\}$

- 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

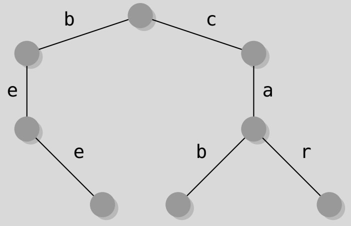
- is leaf found and whole pattern is matched

Delete

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

Insert

- insert rest of pattern ℹ prefix-free

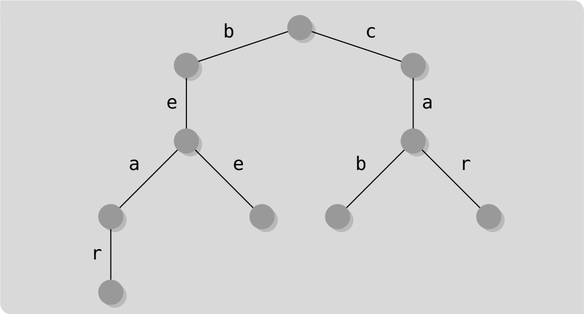


$S = \{\texttt{bear}, \texttt{bee}, \texttt{cab}, \texttt{car}\}$

- is `cab` in S
- remove `bear` from S
- how can we find the predecessor of `can`?

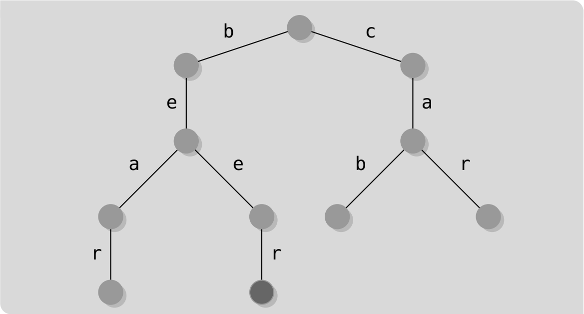
Why Prefix-Free

- insert `beer`



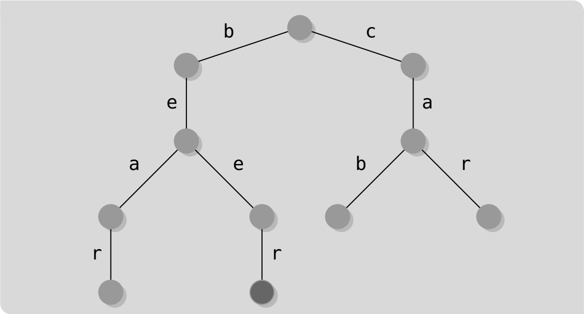
Why Prefix-Free

- insert `beer`



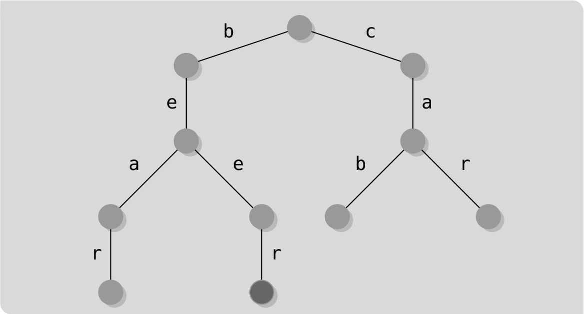
Why Prefix-Free

- insert `beer`
- `bee` cannot be found



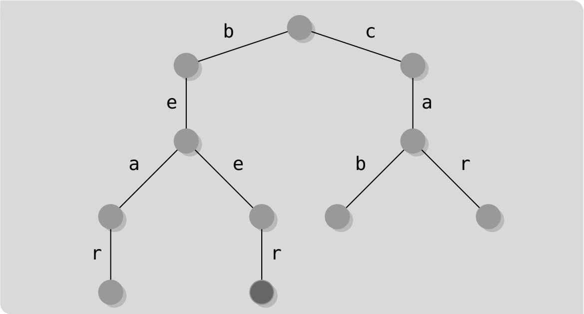
Why Prefix-Free

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

Next Steps

Karlsruhe Institute of Technology

Setting

- alphabet Σ of size σ
- k strings $\{s_1, \ldots, s_k\}$ over the alphabet Σ
- 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 Σ of size σ
- k strings $\{s_1, \ldots, s_k\}$ over the alphabet Σ
- 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

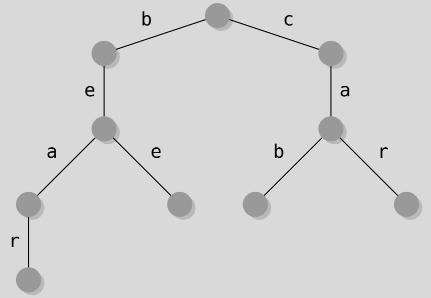
Setting

- alphabet Σ of size σ
- k strings $\{s_1, \ldots, s_k\}$ over the alphabet Σ
- 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

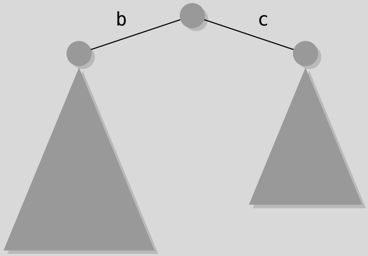
Setting

- alphabet Σ of size σ
- k strings $\{s_1, \ldots, s_k\}$ over the alphabet Σ
- 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

Karlsruhe Institute of Technology

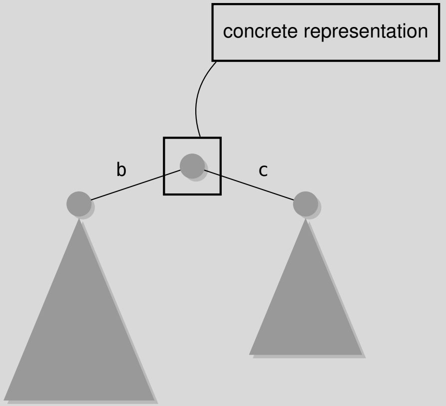
Setting

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

We Want to Know

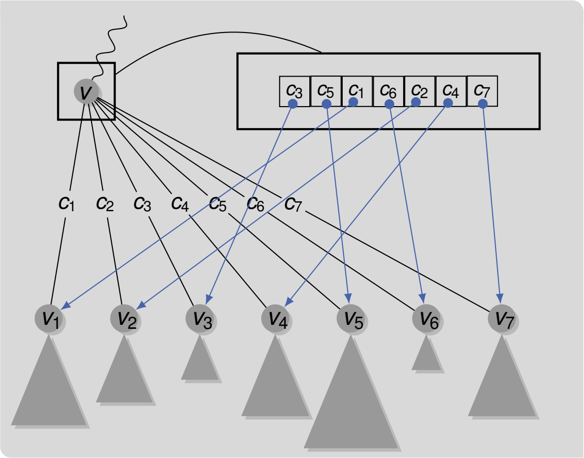
- query times
- space requirements

- 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
  - ℹ children are not ordered

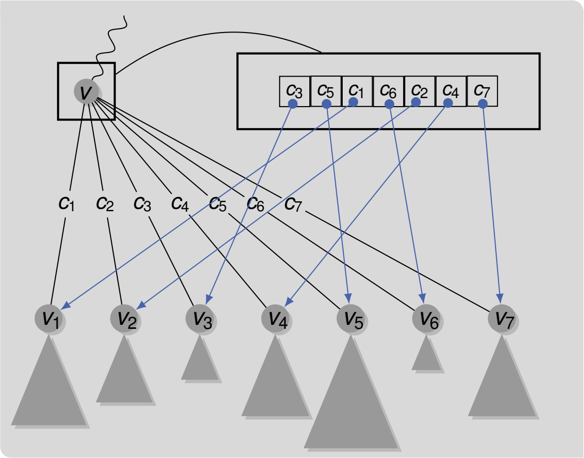


Arrays of Variable Size

- 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
  ℹ children are not ordered

Query Time (Contains)

- $O(m \cdot \sigma)$



Arrays of Variable Size

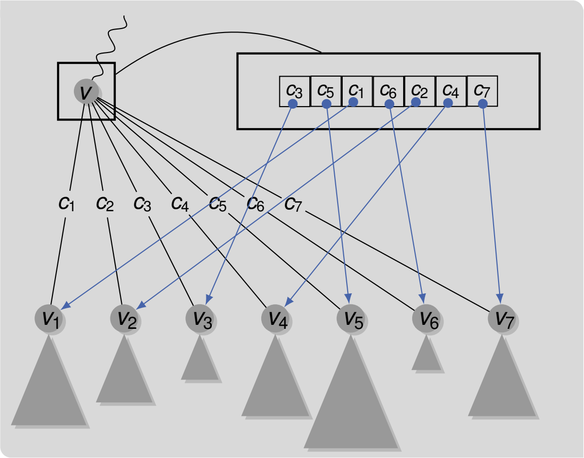
- store children (character and pointer) in the order they are added
- to find child scan array
- to delete child swap with last and remove last
  - ℹ children are not ordered

Query Time (Contains)

- $O(m \cdot \sigma)$

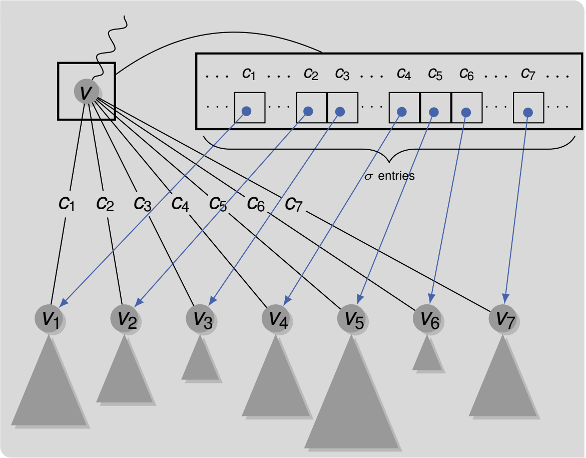
Space

- $O(N)$ words



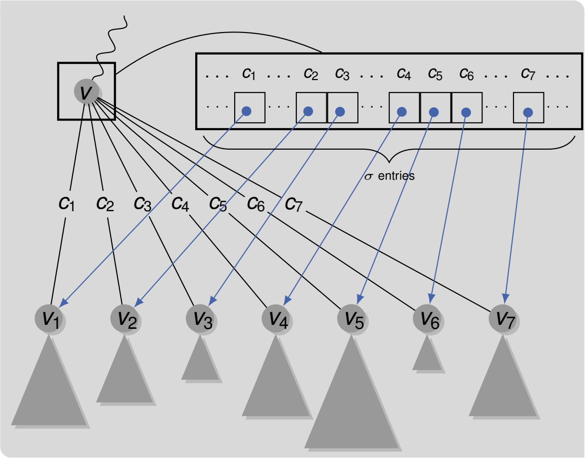
Arrays of Fixed Size

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



Arrays of Fixed Size

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



Query Time (Contains)

- $O(m)$ optimal

Arrays of Fixed Size

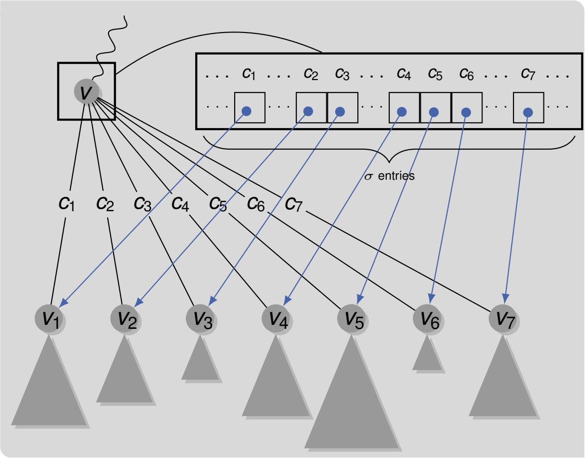
- children (pointer) are stored in arrays of size σ
- use `null` to mark non-existing children
- finding and deleting children is trivial

Query Time (Contains)

- $O(m)$ ℹ optimal

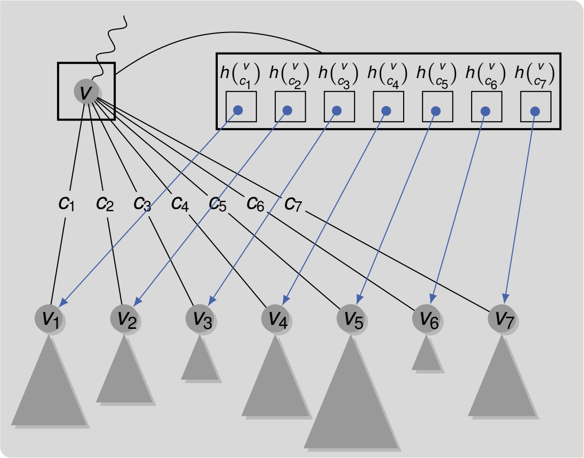
Space

- $O(N \cdot \sigma)$ words ℹ very bad



Hash Tables

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

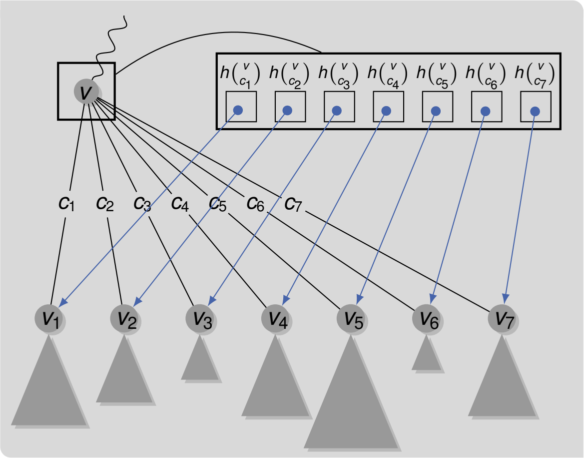


Hash Tables

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

Query Time (Contains)

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



Hash Tables

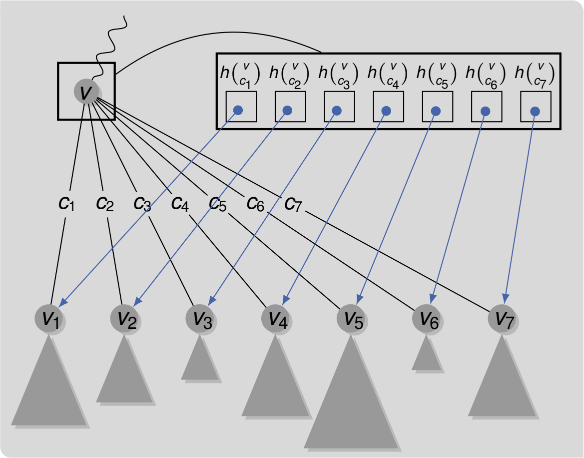
- either use a hash table per node
  - has overhead
- or use global hash table for whole trie

Query Time (Contains)

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

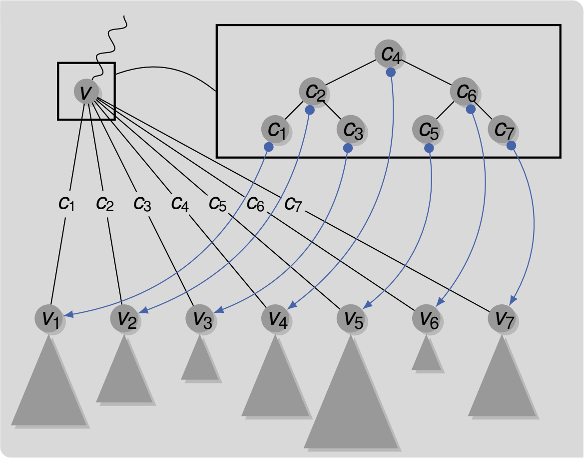
Space

- $O(N)$ 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

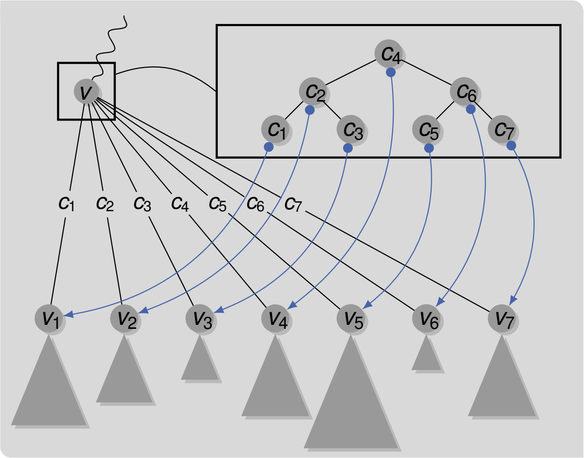


Balanced Search Trees

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

Query Time (Contains)

- $O(m \cdot \lg \sigma)$



Balanced Search Trees

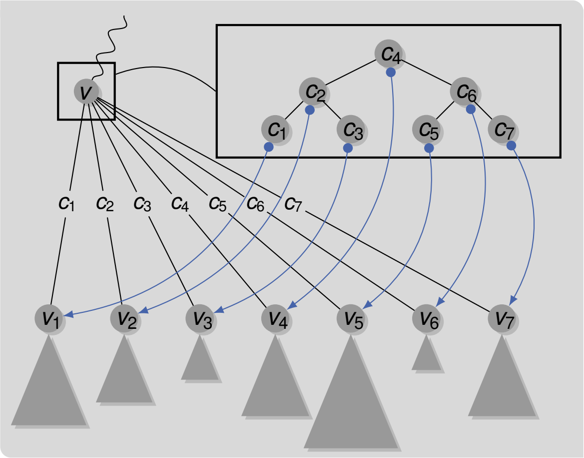
- children are stored in balanced search trees
- e.g., AVL tree, red-black tree, ...
- in static setting sorted array and binary search

Query Time (Contains)

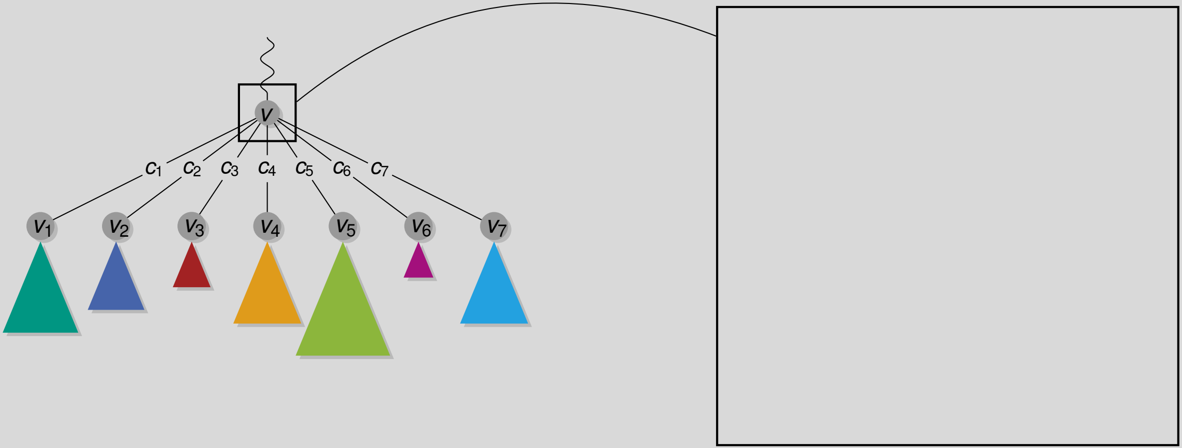
- $O(m \cdot \lg \sigma)$

Space

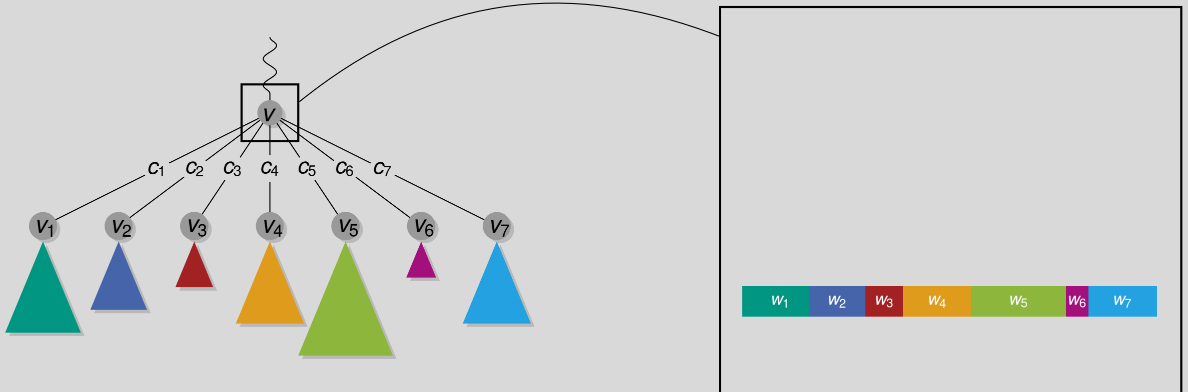
- $O(N)$ words



Weight-Balanced Search Trees (1/2)

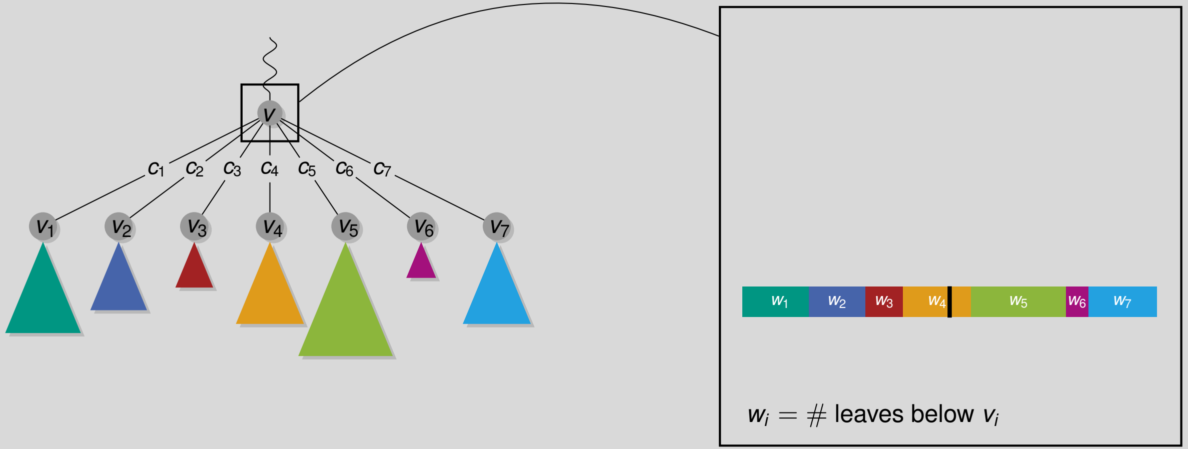


Weight-Balanced Search Trees (1/2)

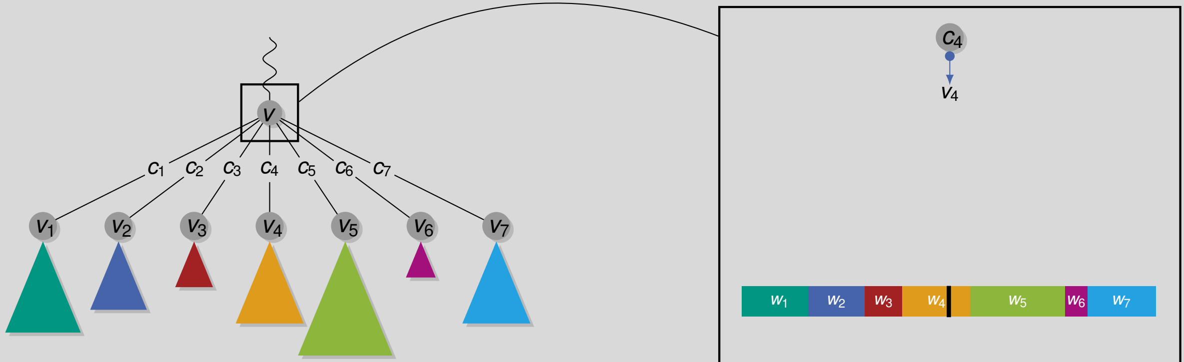


$w_i = \#$ leaves below v_i

Weight-Balanced Search Trees (1/2)

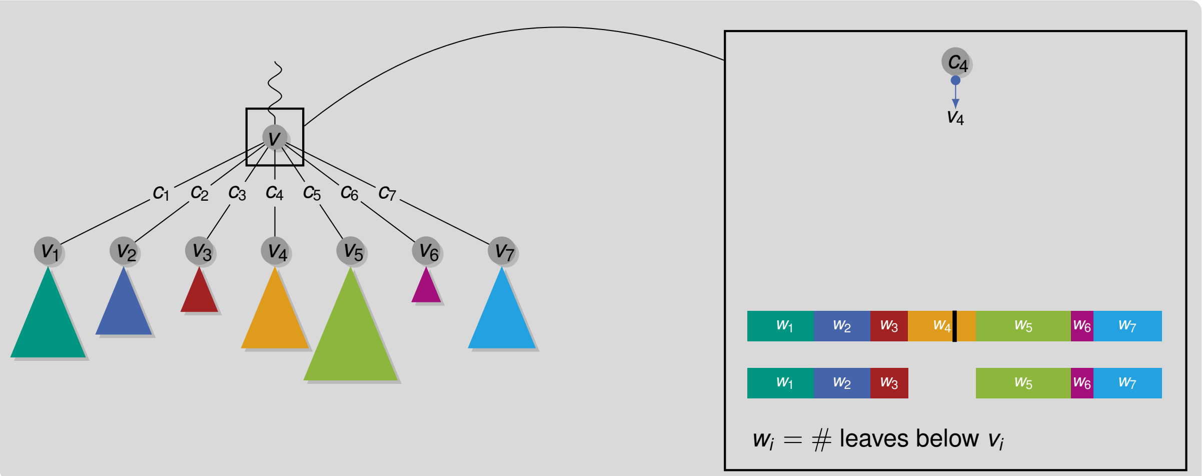


Weight-Balanced Search Trees (1/2)



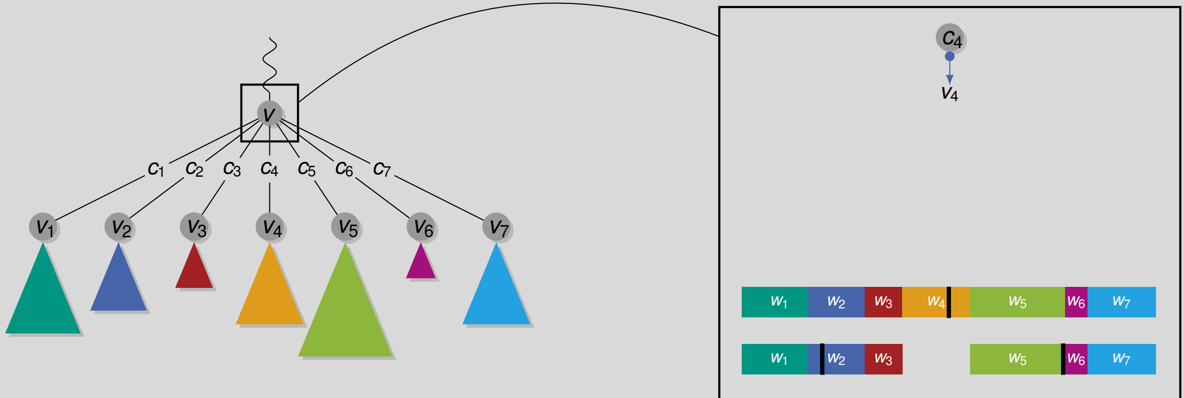
$w_i = \#$ leaves below v_i

Weight-Balanced Search Trees (1/2)



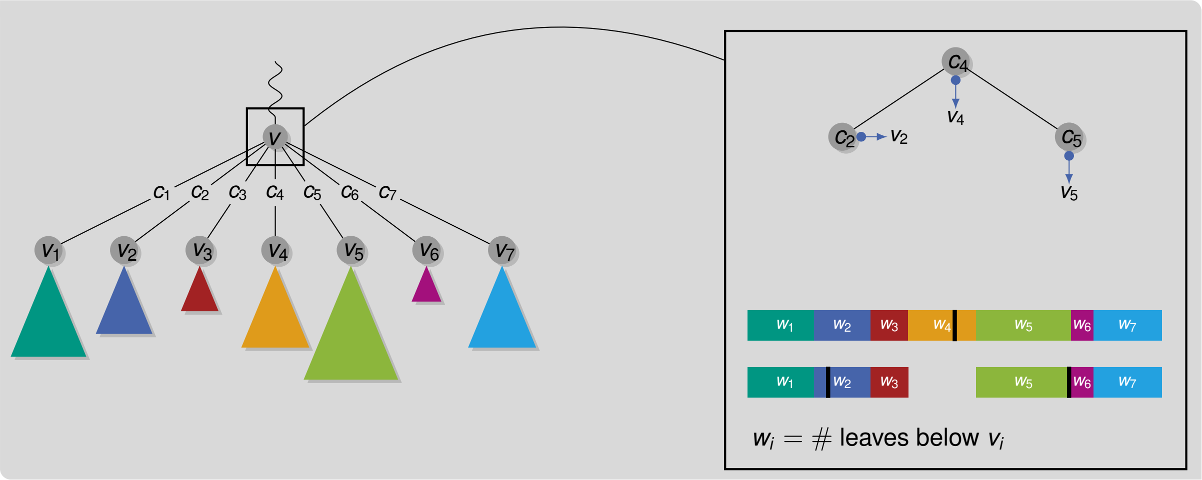
$w_i = \#$ leaves below v_i

Weight-Balanced Search Trees (1/2)

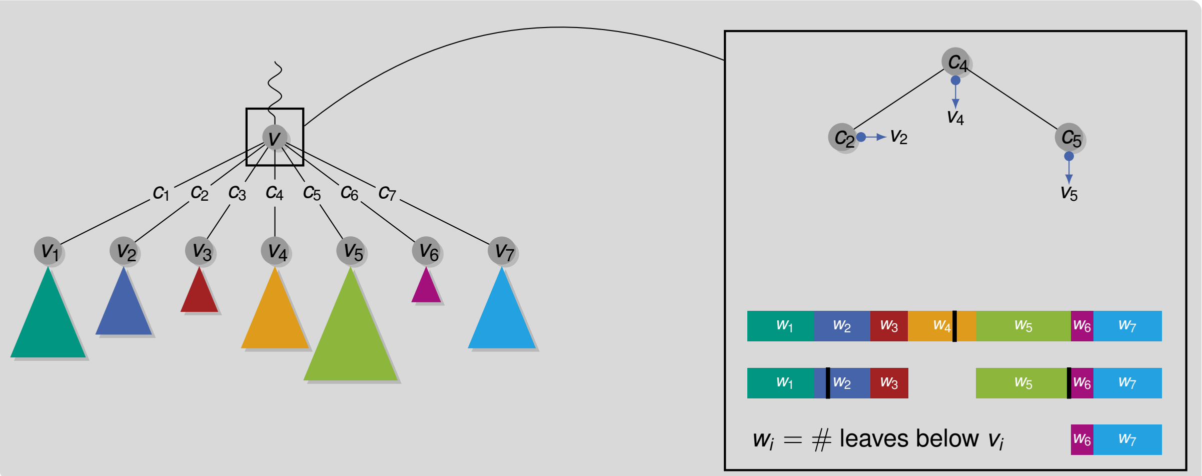


$w_i = \#$ leaves below v_i

Weight-Balanced Search Trees (1/2)

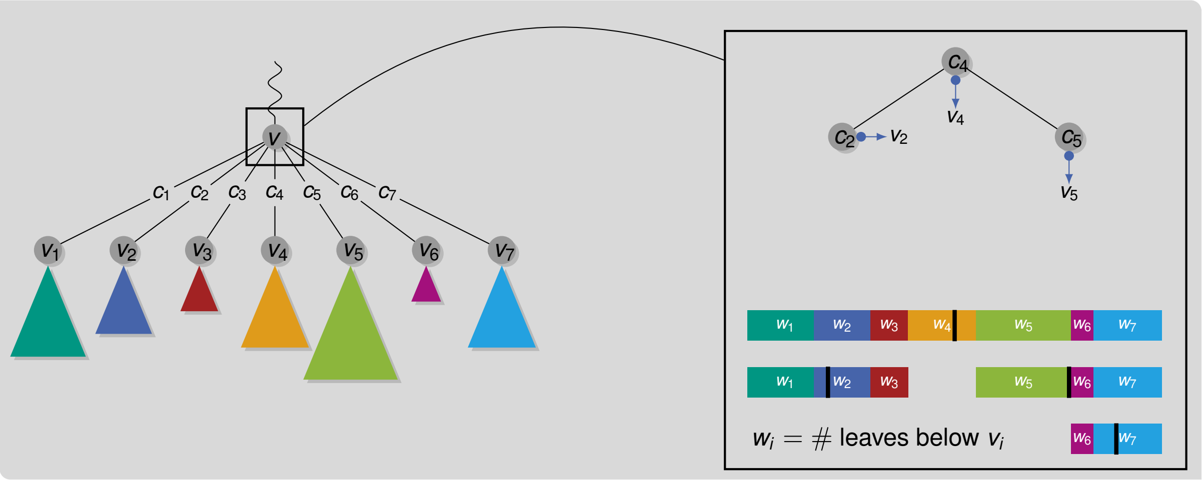


Weight-Balanced Search Trees (1/2)

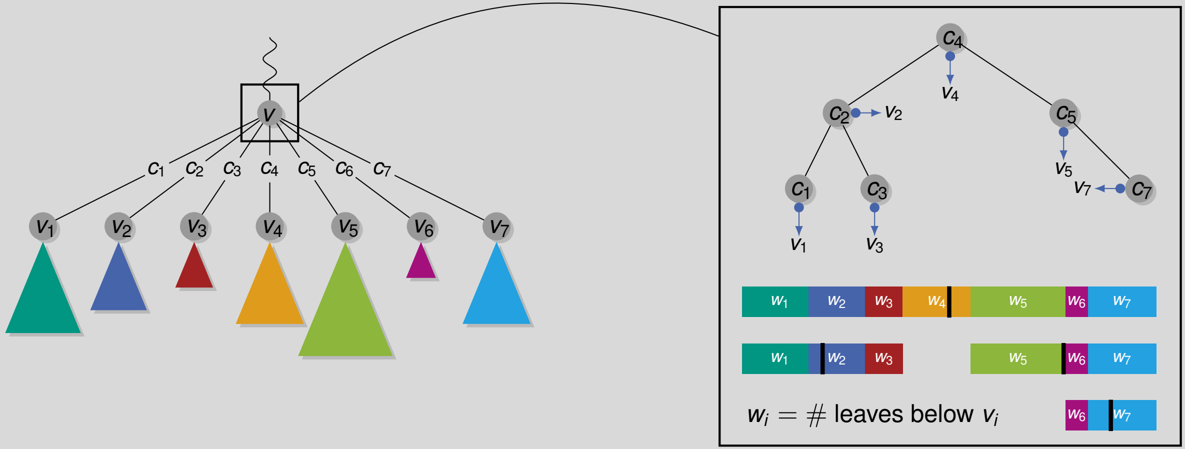


$w_i = \#$ leaves below v_i

Weight-Balanced Search Trees (1/2)

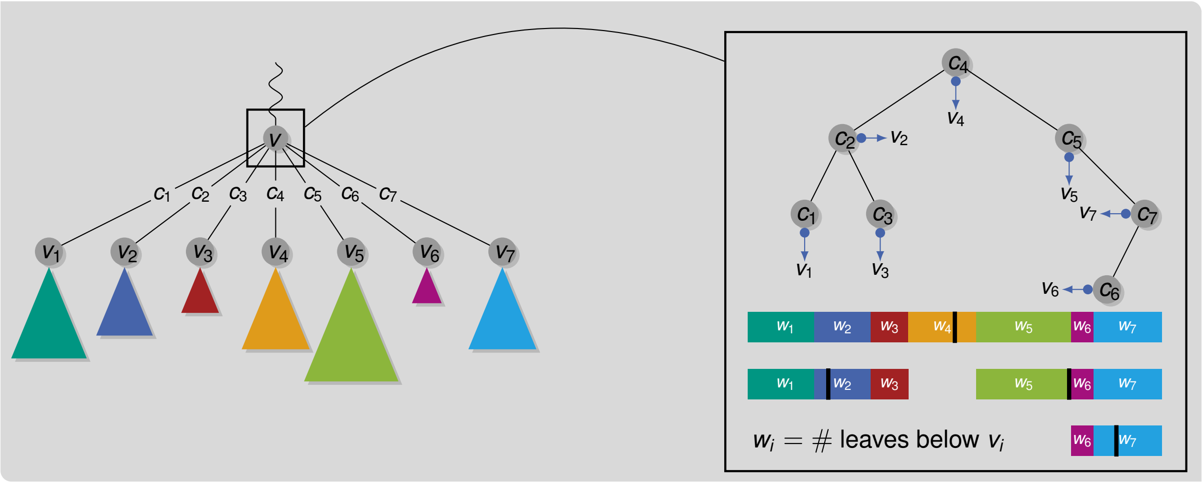


Weight-Balanced Search Trees (1/2)



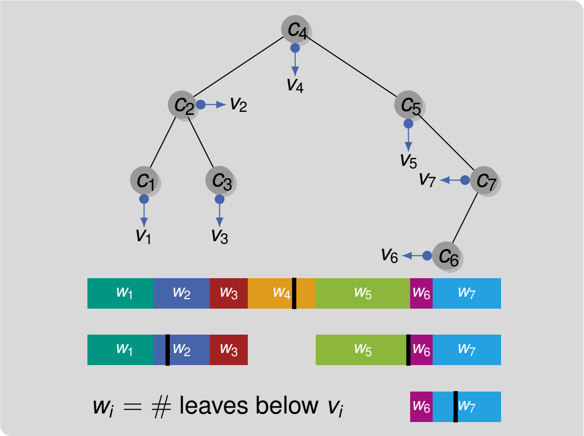
$w_i = \#$ leaves below v_i

Weight-Balanced Search Trees (1/2)



Weight-Balanced Search Trees (2/2)

- use weight-balanced search trees at each node

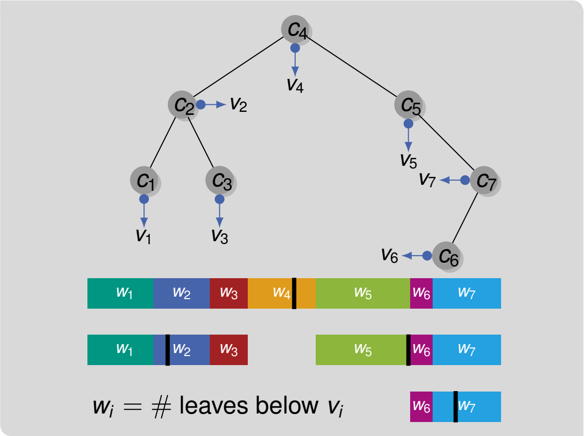


Weight-Balanced Search Trees (2/2)

- use weight-balanced search trees at each node

Query Time (Contains)

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



Weight-Balanced Search Trees (2/2)

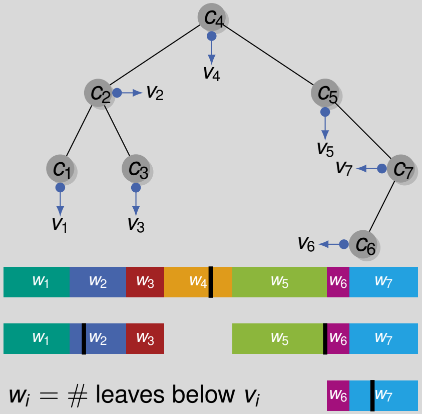
- use weight-balanced search trees at each node

Query Time (Contains)

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

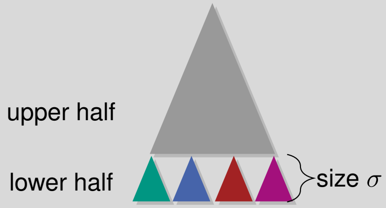
Space

- $O(N)$ words



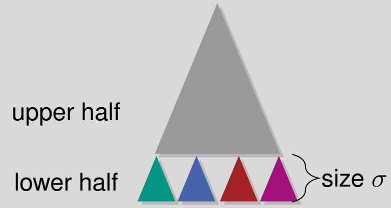
Two-Levels with Weight-Balanced Search Trees

- 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 ℹ branching nodes only



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 ℹ branching nodes only

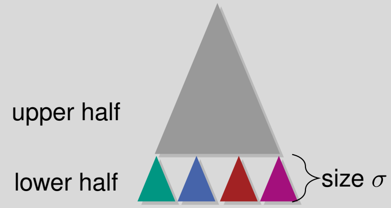


Query Time (Contains)

- 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 ℹ branching nodes only



Query Time (Contains)

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

Space

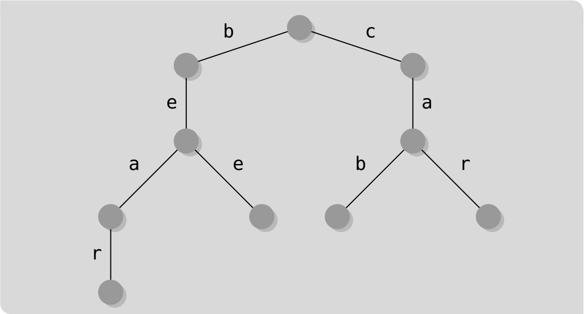
- upper half: $O(N)$ words
  ℹ $O(N/\sigma)$ branching nodes
- lower half: $O(N)$ words
- total: $O(N)$ words

Theoretical Comparison

Representation	Query Time (Contains)	Space in Words
arrays of variable size	$O(m \cdot \sigma)$	$O(N)$
arrays of fixed size	$O(m)$	$O(N \cdot \sigma)$
hash tables	$O(m)$ w.h.p.	$O(N)$
balanced search trees	$O(m \cdot \lg \sigma)$	$O(N)$
weight-balanced search trees	$O(m + \lg k)$	$O(N)$
two-levels with weight-balanced search trees	$O(m + \lg \sigma)$	$O(N)$

Compact Trie

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

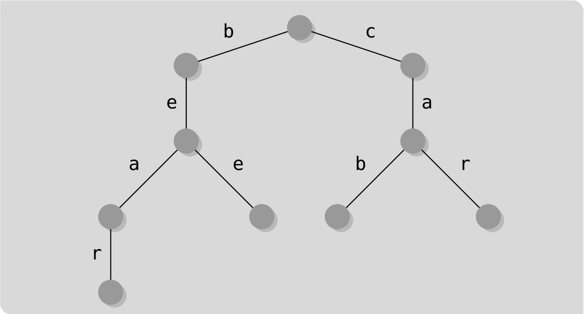


Compact Trie

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

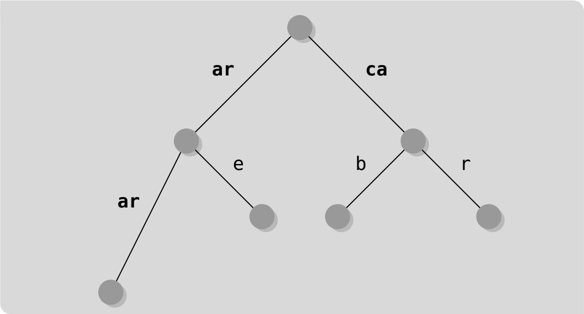
Definition: Compact Trie

- 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.