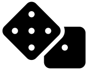
Anagram Trees

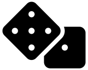
Steffan Christ Sølvsten

AARHUS
UNIVERSITY

Wordrow

wordrow.io

`get_game(``);`

`get_game(``);`

Reason: (*1.*) Simple backend (*2.*) Small file size

Contents

- Motivation
  - Wordrow
- Anagrams
- Binary Anatree
  - `contains(x)`
  - `anagrams(x)`
  - `subanagrams(x)`
  - `insert(x)`
- Multi-valued Anatree
- Letter Ordering

Contents

Motivation

Wordrow

Anagrams

Binary Anatree

`contains(x)`

`anagrams(x)`

`subanagrams(x)`

`insert(x)`

Multi-valued Anatree

Letter Ordering

Anagrams

Consider an *alphabet* $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ and words x, y, ... from a *language* $L \subseteq \Sigma^*$.

Definition (Rohit Parikh, 1961)
The *Parikh vector* of a word $x \in \Sigma^*$ is $\Psi(x) \triangleq \langle |\sigma_1|, |\sigma_2|, \ldots, |\sigma_k| \rangle$.

Example
For $\Sigma = \{a, b, c\}$, $\Psi(abb) = \langle 1, 2, 0 \rangle$ and $\Psi(abab) = \Psi(abba) = \langle 2, 2, 0 \rangle$.

Anagrams

Consider an *alphabet* $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ and words x, y, ... from a *language* $L \subseteq \Sigma^*$.

Definition (Rohit Parikh, 1961)
The *Parikh vector* of a word $x \in \Sigma^*$ is $\Psi(x) \triangleq \langle |\sigma_1|, |\sigma_2|, \ldots, |\sigma_k| \rangle$.

Example
For $\Sigma = \{a, b, c\}$, $\Psi(abb) = \langle 1, 2, 0 \rangle$ and $\Psi(abab) = \Psi(abba) = \langle 2, 2, 0 \rangle$.

Theorem (Rohit Parikh, 1961)
Given a Context-Free Language, $L \subseteq \Sigma^*$, one can efficiently construct the set of all Parikh vectors. One can use this to identify that $x \in \Sigma^*$ **cannot** *be in the language.*

More Details: cs.umu.se/kurser/TDBC92/VT06/final/3.pdf

Anagrams

Consider an *alphabet* $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ and words x, y, ... from a *language* $L \subseteq \Sigma^*$.

Definition (Rohit Parikh, 1961)
The *Parikh vector* of a word $x \in \Sigma^*$ is $\Psi(x) \triangleq \langle |\sigma_1|, |\sigma_2|, \ldots, |\sigma_k| \rangle$.

Example
For $\Sigma = \{a, b, c\}$, $\Psi(abb) = \langle 1, 2, 0 \rangle$ and $\Psi(abab) = \Psi(abba) = \langle 2, 2, 0 \rangle$.

Anagrams

Consider an *alphabet* $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ and words x, y, ... from a *language* $L \subseteq \Sigma^*$.

Definition (Rohit Parikh, 1961)
The *Parikh vector* of a word $x \in \Sigma^*$ is $\Psi(x) \triangleq \langle |\sigma_1|, |\sigma_2|, \ldots, |\sigma_k| \rangle$.

Example
For $\Sigma = \{a, b, c\}$, $\Psi(abb) = \langle 1, 2, 0 \rangle$ and $\Psi(abab) = \Psi(abba) = \langle 2, 2, 0 \rangle$.

Definition (Anagram)
$x, y \in \Sigma^*$ are *anagrams* if $\Psi(x) = \Psi(y)$.

Definition (Subanagram)
$x \in \Sigma^*$ is a *subanagram* of $y \in \Sigma^*$ if $\Psi(x) \leq \Psi(y)$.

Anagrams

Lemma
Given $x, y \in \Sigma^n$, *one can compute whether* $\Psi(x) = \Psi(y)$ *in* $\mathcal{O}(n + |\Sigma|)$ *time.*

Lemma
Given $x, y \in \Sigma^*$, *one can compute whether* $\Psi(x) \leq \Psi(y)$ *in* $\mathcal{O}(|x| + |y| + |\Sigma|)$ *time.*

Proof.

$\square$

Anagrams

Lemma
Given $x, y \in \Sigma^n$, one can compute whether $\Psi(x) = \Psi(y)$ in $\mathcal{O}(n + |\Sigma|)$ time.

Lemma
Given $x, y \in \Sigma^*$, one can compute whether $\Psi(x) \leq \Psi(y)$ in $\mathcal{O}(|x| + |y| + |\Sigma|)$ time.

Proof.

Compute the Parikh vectors similar to the first half of *Counting Sort*. □

Example
Counting the number of *a*'s, *b*'s, and *c*'s in *aba* and *aab* both yield $\langle 2, 1, 0 \rangle$.

Anagrams

Lemma
Given $x, y \in \Sigma^n$, *computing whether* $\Psi(x) = \Psi(y)$ *takes* $\mathcal{O}(\mathit{sort}(n))$ *time.*

Proof.

$\square$

Anagrams

Lemma
Given $x, y \in \Sigma^n$, computing whether $\Psi(x) = \Psi(y)$ takes $\mathcal{O}(sort(n))$ time.

Proof.

Sort words x and y in $\mathcal{O}(\text{sort}(n))$ time. Then, check whether they now are the very same word in $O(n)$ time. $\square$

Example

$$\begin{array}{lccc} x = & b & a & a \\ y = & a & b & a \end{array} \qquad \begin{array}{lccc} x = & c & a & b \\ y = & a & b & a \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^n$, computing whether $\Psi(x) = \Psi(y)$ takes $\mathcal{O}(sort(n))$ time.

Proof.
Sort words x and y in $\mathcal{O}(\text{sort}(n))$ time. Then, check whether they now are the very same word in $O(n)$ time. $\square$

Example

$$\begin{array}{llll} x = & a & a & b \\ y = & a & a & b \end{array} \qquad \begin{array}{llll} x = & c & a & b \\ y = & a & b & a \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^n$, computing whether $\Psi(x) = \Psi(y)$ takes $\mathcal{O}(sort(n))$ time.

Proof.
Sort words x and y in $\mathcal{O}(\text{sort}(n))$ time. Then, check whether they now are the very same word in $O(n)$ time. $\square$

Example

$$\begin{array}{llll} x = & \underline{\mathbf{a}} & a & b \\ y = & \underline{\mathbf{a}} & a & b \end{array} \qquad\qquad \begin{array}{llll} x = & c & a & b \\ y = & a & b & a \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^n$, computing whether $\Psi(x) = \Psi(y)$ takes $\mathcal{O}(sort(n))$ time.

Proof.
Sort words x and y in $\mathcal{O}(\text{sort}(n))$ time. Then, check whether they now are the very same word in $O(n)$ time. $\square$

Example

$$\begin{array}{lccc} x = & a & \underline{\mathbf{a}} & b \\ y = & a & \underline{\mathbf{a}} & b \end{array} \qquad \begin{array}{lccc} x = & c & a & b \\ y = & a & b & a \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^n$, computing whether $\Psi(x) = \Psi(y)$ takes $\mathcal{O}(sort(n))$ time.

Proof.
Sort words x and y in $\mathcal{O}(\text{sort}(n))$ time. Then, check whether they now are the very same word in $O(n)$ time. $\square$

Example

$$\begin{array}{llll} \text{x} = & \text{a} & \text{a} & \underline{\textbf{b}} \\ \text{y} = & \text{a} & \text{a} & \underline{\textbf{b}} \end{array} \qquad \begin{array}{llll} \text{x} = & \text{c} & \text{a} & \text{b} \\ \text{y} = & \text{a} & \text{b} & \text{a} \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^n$, computing whether $\Psi(x) = \Psi(y)$ takes $\mathcal{O}(sort(n))$ time.

Proof.
Sort words x and y in $\mathcal{O}(\text{sort}(n))$ time. Then, check whether they now are the very same word in $O(n)$ time. $\square$

Example

x =	a	a	b	✔		x =	c	a	b
y =	a	a	b			y =	a	b	a

Anagrams

Lemma
Given $x, y \in \Sigma^n$, computing whether $\Psi(x) = \Psi(y)$ takes $\mathcal{O}(sort(n))$ time.

Proof.
Sort words x and y in $\mathcal{O}(\text{sort}(n))$ time. Then, check whether they now are the very same word in $O(n)$ time. $\square$

Example

$$\begin{array}{lccc} x = & a & a & b \\ y = & a & a & b \end{array} \checkmark \qquad \begin{array}{lccc} x = & a & b & c \\ y = & a & a & b \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^n$, computing whether $\Psi(x) = \Psi(y)$ takes $\mathcal{O}(sort(n))$ time.

Proof.
Sort words x and y in $\mathcal{O}(\text{sort}(n))$ time. Then, check whether they now are the very same word in $O(n)$ time. $\square$

Example

$$\begin{array}{lccc} x = & a & a & b \\ y = & a & a & b \end{array} \checkmark \qquad \begin{array}{lccc} x = & \underline{\mathbf{a}} & b & c \\ y = & \underline{\mathbf{a}} & a & b \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^n$, computing whether $\Psi(x) = \Psi(y)$ takes $\mathcal{O}(sort(n))$ time.

Proof.
Sort words x and y in $\mathcal{O}(\text{sort}(n))$ time. Then, check whether they now are the very same word in $O(n)$ time. □

Example

$$\begin{array}{lccc} \text{x} = & \text{a} & \text{a} & \text{b} \\ \text{y} = & \text{a} & \text{a} & \text{b} \end{array} \checkmark \qquad \begin{array}{lccc} \text{x} = & \text{a} & \underline{\mathbf{b}} & \text{c} \\ \text{y} = & \text{a} & \underline{\mathbf{a}} & \text{b} \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^n$, computing whether $\Psi(x) = \Psi(y)$ takes $\mathcal{O}(sort(n))$ time.

Proof.

Sort words x and y in $\mathcal{O}(\text{sort}(n))$ time. Then, check whether they now are the very same word in $O(n)$ time. $\square$

Example

$$\begin{array}{lccc} x = & a & a & b \\ y = & a & a & b \end{array} \checkmark \qquad \begin{array}{lccc} x = & a & b & c \\ y = & a & a & b \end{array} \, !$$

Anagrams

Lemma
Given $x, y \in \Sigma^*$, *checking* $\Psi(x) \leq \Psi(y)$ *takes* $\mathcal{O}(sort(|x|) + sort(|y|))$ *time.*

Proof.

$\square$

Anagrams

Lemma
Given $x, y \in \Sigma^*$, checking $\Psi(x) \leq \Psi(y)$ takes $\mathcal{O}(sort(|x|) + sort(|y|))$ time.

Proof.
Again, sort words x and y. Now, match each symbol of x with ones in y; skip symbols of y if x is "ahead". $\square$

Example

x =	b	a			x =	c	a	
y =	a	b	a		y =	a	b	a

Anagrams

Lemma
Given $x, y \in \Sigma^*$, checking $\Psi(x) \leq \Psi(y)$ takes $\mathcal{O}(sort(|x|) + sort(|y|))$ time.

Proof.

Again, sort words x and y. Now, match each symbol of x with ones in y; skip symbols of y if x is "ahead". $\square$

Example

x =	a	b			x =	c	a	
y =	a	a	b		y =	a	b	a

Anagrams

Lemma
Given $x, y \in \Sigma^*$, checking $\Psi(x) \leq \Psi(y)$ takes $\mathcal{O}(sort(|x|) + sort(|y|))$ time.

Proof.
Again, sort words x and y. Now, match each symbol of x with ones in y; skip symbols of y if x is "ahead". □

Example

x =	$\underline{\mathbf{a}}$	b			x =	c	a	
y =	$\underline{\mathbf{a}}$	a	b		y =	a	b	a

Anagrams

Lemma
Given $x, y \in \Sigma^*$, checking $\Psi(x) \leq \Psi(y)$ takes $\mathcal{O}(sort(|x|) + sort(|y|))$ time.

Proof.
Again, sort words x and y. Now, match each symbol of x with ones in y; skip symbols of y if x is "ahead". $\square$

Example

$$\begin{array}{lccc} \mathrm{x} = & \mathrm{a} & \underline{\mathbf{b}} & \\ \mathrm{y} = & \mathrm{a} & \underline{\mathbf{a}} & \mathrm{b} \end{array} \qquad \begin{array}{lccc} \mathrm{x} = & \mathrm{c} & \mathrm{a} & \\ \mathrm{y} = & \mathrm{a} & \mathrm{b} & \mathrm{a} \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^*$, checking $\Psi(x) \leq \Psi(y)$ takes $\mathcal{O}(sort(|x|) + sort(|y|))$ time.

Proof.
Again, sort words x and y. Now, match each symbol of x with ones in y; skip symbols of y if x is "ahead". □

Example

$$\begin{array}{llll} \mathrm{x} = & \mathrm{a} & \underline{\mathbf{b}} & \\ \mathrm{y} = & \mathrm{a} & \mathrm{a} & \underline{\mathbf{b}} \end{array} \qquad \begin{array}{llll} \mathrm{x} = & \mathrm{c} & \mathrm{a} & \\ \mathrm{y} = & \mathrm{a} & \mathrm{b} & \mathrm{a} \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^*$, checking $\Psi(x) \leq \Psi(y)$ takes $\mathcal{O}(sort(|x|) + sort(|y|))$ time.

Proof.
Again, sort words x and y. Now, match each symbol of x with ones in y; skip symbols of y if x is "ahead". □

Example

$$\begin{array}{llll} x = & a & b & \\ y = & a & a & b \end{array} \quad ✔ \qquad \begin{array}{llll} x = & c & a & \\ y = & a & b & a \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^*$, checking $\Psi(x) \leq \Psi(y)$ takes $\mathcal{O}(sort(|x|) + sort(|y|))$ time.

Proof.
Again, sort words x and y. Now, match each symbol of x with ones in y; skip symbols of y if x is "ahead". $\square$

Example

$$\begin{array}{llll} x = & a & b & \\ y = & a & a & b \end{array} \quad ✔ \qquad\qquad \begin{array}{llll} x = & a & c & \\ y = & a & a & b \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^*$, checking $\Psi(x) \leq \Psi(y)$ takes $\mathcal{O}(sort(|x|) + sort(|y|))$ time.

Proof.
Again, sort words x and y. Now, match each symbol of x with ones in y; skip symbols of y if x is "ahead". $\square$

Example

$$\begin{array}{llll} \mathrm{x} = & \mathrm{a} & \mathrm{b} & \\ \mathrm{y} = & \mathrm{a} & \mathrm{a} & \mathrm{b} \end{array} \checkmark \qquad \begin{array}{llll} \mathrm{x} = & \underline{\mathbf{a}} & \mathrm{c} & \\ \mathrm{y} = & \underline{\mathbf{a}} & \mathrm{a} & \mathrm{b} \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^*$, checking $\Psi(x) \leq \Psi(y)$ takes $\mathcal{O}(sort(|x|) + sort(|y|))$ time.

Proof.
Again, sort words x and y. Now, match each symbol of x with ones in y; skip symbols of y if x is "ahead". $\square$

Example

$$\begin{array}{llll} x = & a & b & \\ y = & a & a & b \end{array} \quad \checkmark \qquad \begin{array}{llll} x = & a & \underline{\mathbf{c}} & \\ y = & a & \underline{\mathbf{a}} & b \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^*$, checking $\Psi(x) \leq \Psi(y)$ takes $\mathcal{O}(sort(|x|) + sort(|y|))$ time.

Proof.
Again, sort words x and y. Now, match each symbol of x with ones in y; skip symbols of y if x is "ahead". $\square$

Example

$$\begin{array}{llll} x = & a & b & \\ y = & a & a & b \end{array} \checkmark \qquad \begin{array}{llll} x = & a & \underline{\mathbf{c}} & \\ y = & a & a & \underline{\mathbf{b}} \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^*$, *checking* $\Psi(x) \leq \Psi(y)$ *takes* $\mathcal{O}(sort(|x|) + sort(|y|))$ *time.*

Proof.
Again, sort words x and y. Now, match each symbol of x with ones in y; skip symbols of y if x is "ahead". $\square$

Example

$$\begin{array}{llll} x = & a & b & \\ y = & a & a & b \end{array} \checkmark \qquad \begin{array}{llll} x = & a & \underline{c} & \\ y = & a & a & b \end{array}$$

Anagrams

Lemma
Given $x, y \in \Sigma^*$, checking $\Psi(x) \leq \Psi(y)$ takes $\mathcal{O}(sort(|x|) + sort(|y|))$ time.

Proof.
Again, sort words x and y. Now, match each symbol of x with ones in y; skip symbols of y if x is "ahead". □

Example

x =	a	b		✔		x =	a	c		!
y =	a	a	b			y =	a	a	b	

Contents

Motivation

- Wordrow

Anagrams

Binary Anatree

- `contains(x)`
- `anagrams(x)`
- `subanagrams(x)`
- `insert(x)`

Multi-valued Anatree

Letter Ordering

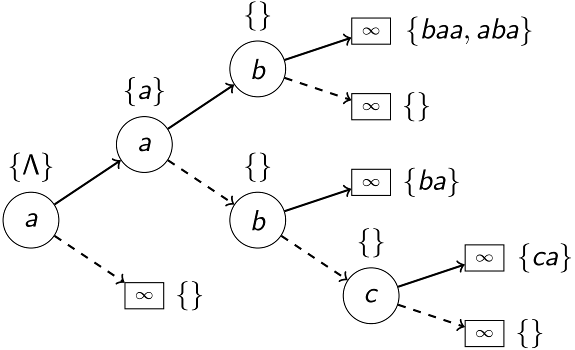
Anatree

Given an alphabet, Σ, and an ordering on its symbols, $< : \Sigma \times \Sigma \to \{\top, \bot\}$, the *Anatree* data structure manages a set of words $L \subseteq \Sigma^*$ on which one can do

Operation	
`insert(x)` `delete(x)`	$\mathcal{O}(\text{sort}(\lvert x \rvert) + \lvert \Sigma \rvert)$
`contains(x)`	$\mathcal{O}(\text{sort}(\lvert x \rvert) + \lvert \Sigma \rvert)$
`anagrams(x)`	$\mathcal{O}(\text{sort}(\lvert x \rvert) + \lvert \Sigma \rvert + T)$
`subanagrams(x)`	$\mathcal{O}(\text{sort}(\lvert x \rvert) + \min(N_{\text{Tree}}, 2^{\lvert x \rvert} \cdot \lvert \Sigma \rvert) + T)$

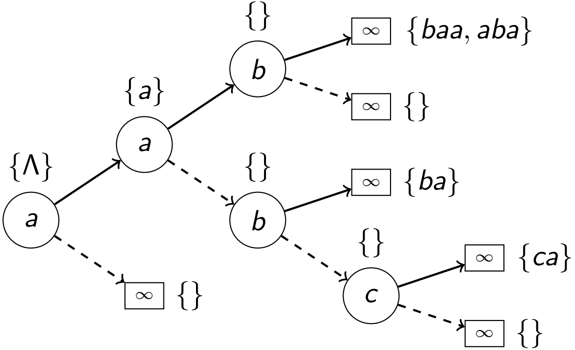
where N_{Tree} is the size of the Anagram tree and T is the output size.

Anatree



$$L = \{\Lambda, a, ba, ca, aba, baa\}$$

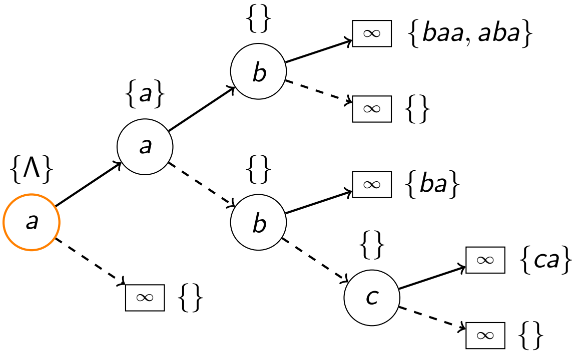
Anatree.`contains(ba)`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
contains(x):
  n := find(root, sort(x), 0)
  return n ≠ NIL & n.contains(x)

find(n, x', i):
```

Anatree.contains(ba)

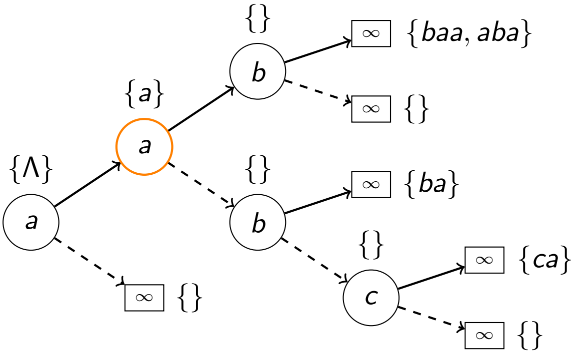


$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
contains(x):
  n := find(root, sort(x), 0)
  return n ≠ NIL & n.contains(x)

find(n, x', i):
```

```
  if x'[i] = n.char
    return find(n.true , x', i+1)
```

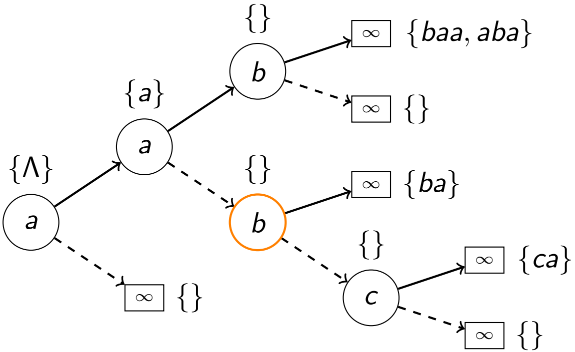
Anatree.`contains(ba)`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
contains(x):
  n := find(root, sort(x), 0)
  return n ≠ NIL & n.contains(x)

find(n, x', i):






  if x'[i] > n.char
    return find(n.false, x', i)
  if x'[i] = n.char
    return find(n.true , x', i+1)
```

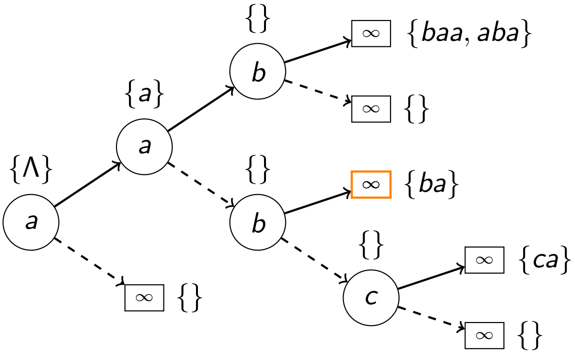
Anatree.`contains(ba)`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
contains(x):
  n := find(root, sort(x), 0)
  return n ≠ NIL & n.contains(x)

find(n, x', i):




  if x'[i] > n.char
    return find(n.false, x', i)
  if x'[i] = n.char
    return find(n.true , x', i+1)
```

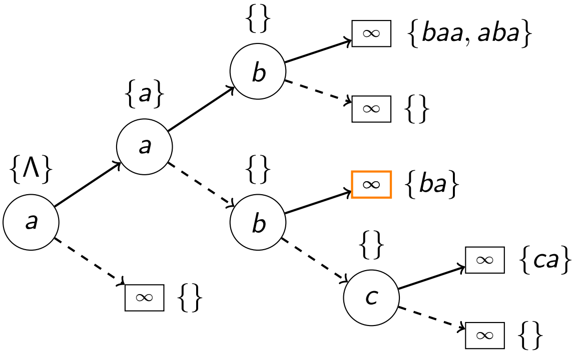
Anatree.contains(ba)



$$L = \{\Lambda, a, ba, ca, aba, baa\}$$

```
contains(x):
  n := find(root, sort(x), 0)
  return n ≠ NIL & n.contains(x)

find(n, x', i):
  if i = x'.length
    return n



  if x'[i] > n.char
    return find(n.false, x', i)
  if x'[i] = n.char
    return find(n.true , x', i+1)
```

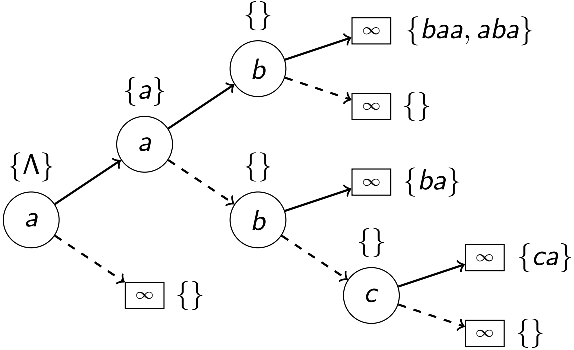
Anatree.`contains(ba) = Yes`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
contains(x):
  n := find(root, sort(x), 0)
  return n ≠ NIL & n.contains(x)

find(n, x', i):
  if i = x'.length
    return n



  if x'[i] > n.char
    return find(n.false, x', i)
  if x'[i] = n.char
    return find(n.true , x', i+1)
```

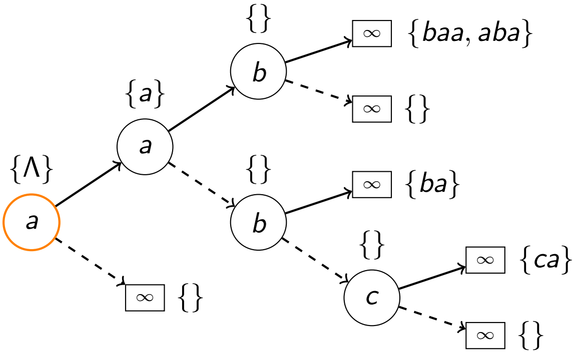
Anatree.`contains(aca)`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
contains(x):
  n := find(root, sort(x), 0)
  return n ≠ NIL & n.contains(x)

find(n, x', i):
  if i = x'.length
    return n



  if x'[i] > n.char
    return find(n.false, x', i)
  if x'[i] = n.char
    return find(n.true , x', i+1)
```

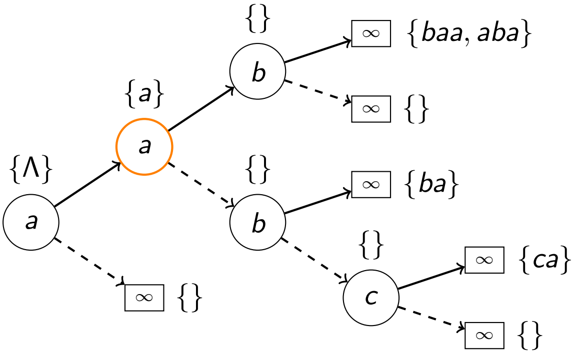
Anatree.`contains(aca)`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
contains(x):
  n := find(root, sort(x), 0)
  return n ≠ NIL & n.contains(x)

find(n, x', i):
  if i = x'.length
    return n



  if x'[i] > n.char
    return find(n.false, x', i)
  if x'[i] = n.char
    return find(n.true , x', i+1)
```

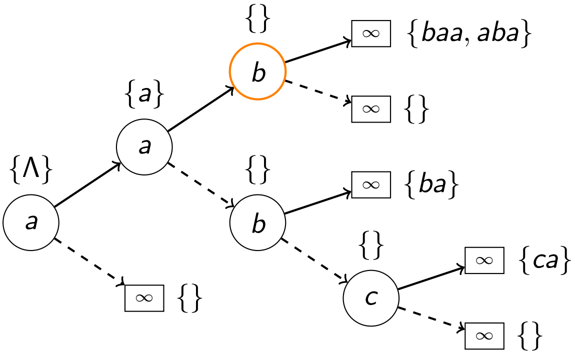
Anatree.`contains(aca)`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
contains(x):
  n := find(root, sort(x), 0)
  return n ≠ NIL & n.contains(x)

find(n, x', i):
  if i = x'.length
    return n



  if x'[i] > n.char
    return find(n.false, x', i)
  if x'[i] = n.char
    return find(n.true , x', i+1)
```

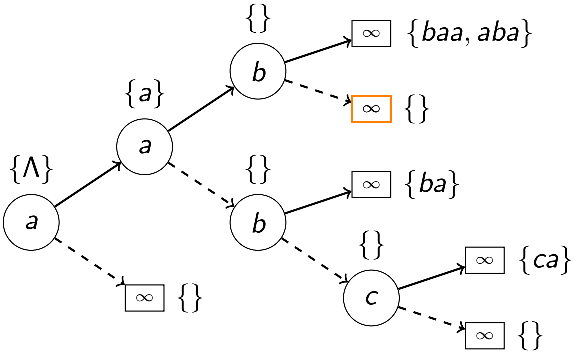
Anatree.`contains(aca)`



$$L = \{\Lambda, a, ba, ca, aba, baa\}$$

```
contains(x):
  n := find(root, sort(x), 0)
  return n ≠ NIL & n.contains(x)

find(n, x', i):
  if i = x'.length
    return n



  if x'[i] > n.char
    return find(n.false, x', i)
  if x'[i] = n.char
    return find(n.true , x', i+1)
```

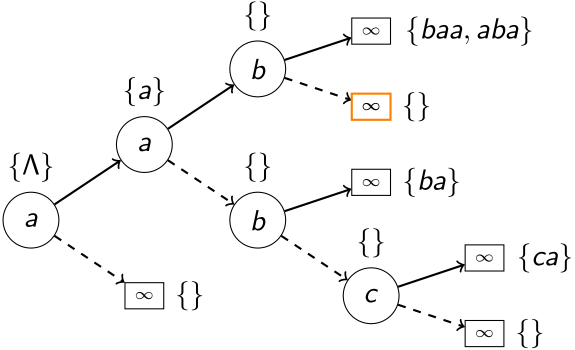
Anatree.contains(aca)



$$L = \{\Lambda, a, ba, ca, aba, baa\}$$

```
contains(x):
  n := find(root, sort(x), 0)
  return n ≠ NIL & n.contains(x)

find(n, x', i):
  if i = x'.length
    return n
  if x'[i] < n.char
    return NIL
  if x'[i] > n.char
    return find(n.false, x', i)
  if x'[i] = n.char
    return find(n.true , x', i+1)
```

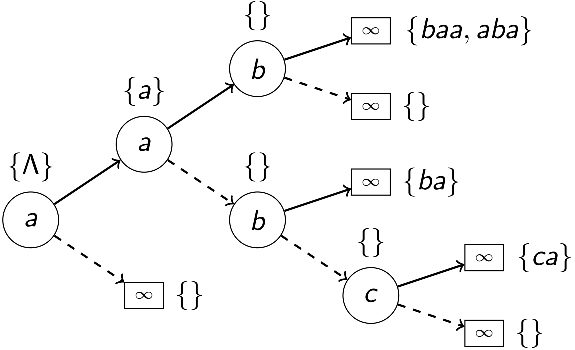
Anatree.`contains(aca) = No`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
contains(x):
  n := find(root, sort(x), 0)
  return n ≠ NIL & n.contains(x)

find(n, x', i):
  if i = x'.length
    return n
  if x'[i] < n.char
    return NIL
  if x'[i] > n.char
    return find(n.false, x', i)
  if x'[i] = n.char
    return find(n.true , x', i+1)
```

Anatree.`contains(...)`



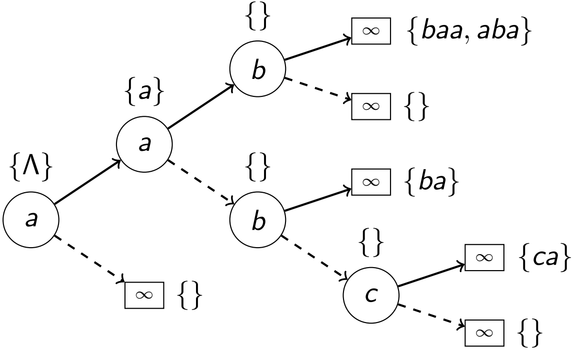
$L = \{\Lambda, a, ba, ca, aba, baa\}$

Lemma
`find(n, sort(x), i)` runs in $\mathcal{O}(sort(|x|) + |\Sigma|)$ time.

Proof.
$\mathcal{O}(1)$ time is spent per node. At most $|x|$ *high* edges and $|\Sigma|$ *low* edges are traversed, meaning at most $|x| + |\Sigma|$ nodes are visited.

On top of this, add the $\mathcal{O}(\text{sort}(|x|))$ time to sort x into x'. $\square$

Anatree.`contains(...)`



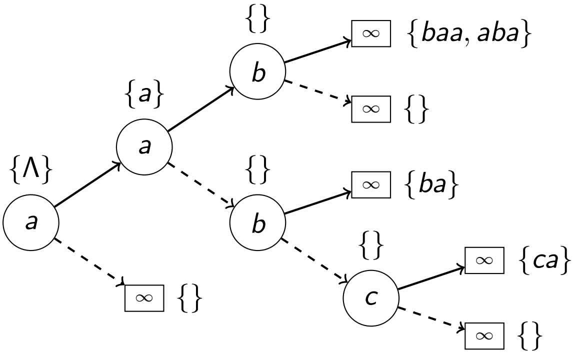
$L = \{\Lambda, a, ba, ca, aba, baa\}$

Lemma
`find(n, sort(x), i)` runs in $\mathcal{O}(sort(|x|) + |\Sigma|)$ time.

Proof.
$\mathcal{O}(1)$ time is spent per node... □

Corollary
`contains(x)` runs in $\mathcal{O}(sort(|x|) + |\Sigma|)$ time.

Anatree.`anagrams(...)`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
anagrams(x):
  n := find(root, sort(x), 0)
  if n ≠ NIL
    output words in n
```

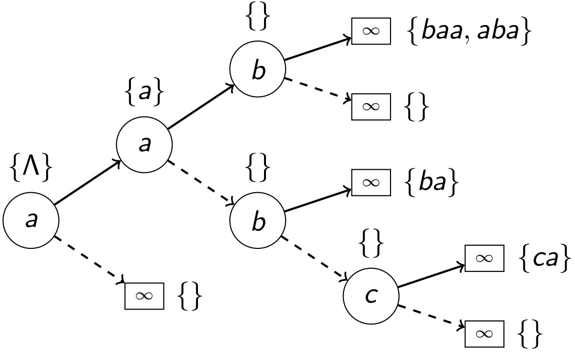
Corollary

`anagrams(x)` runs in $\mathcal{O}(sort(|x|) + |\Sigma| + T)$ time.

Proof.

It takes $\mathcal{O}(\mathtt{sort}(|x|) + |\Sigma|)$ time to find n and then another $\mathcal{O}(T)$ time to output its content. □

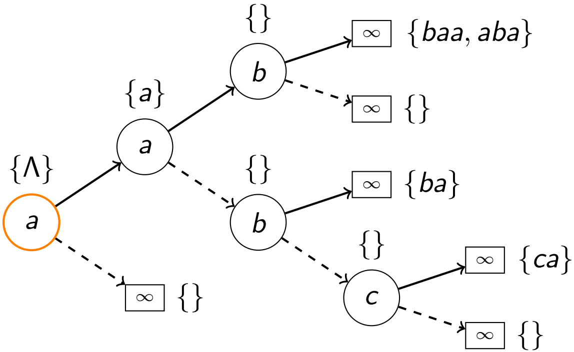
Anatree.`subanagrams(a) =`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)
```

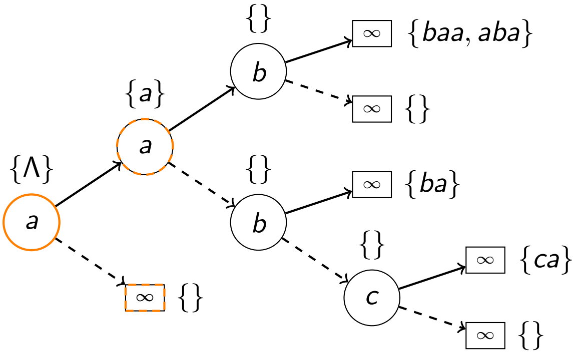
Anatree.`subanagrams(a) =` Λ



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
```

Anatree.`subanagrams(a) = `Λ

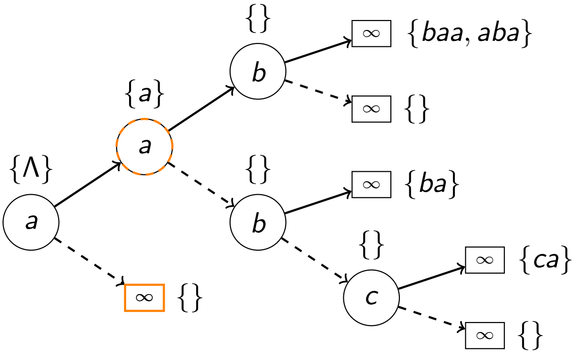


$$L = \{\Lambda, a, ba, ca, aba, baa\}$$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
```

```
  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

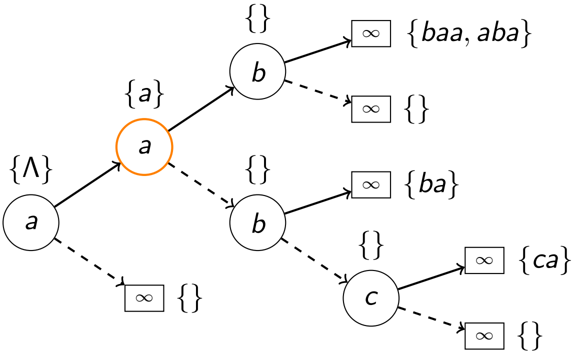
Anatree.`subanagrams(a) = Λ`



$$L = \{\Lambda, a, ba, ca, aba, baa\}$$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
  if n.char = ∞:
    return







  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

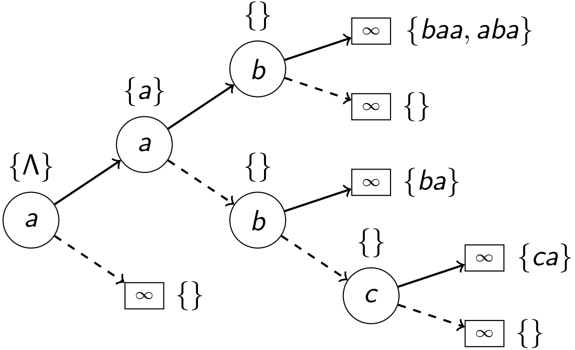
Anatree.`subanagrams(a) = Λ, a`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
  if n.char = ∞:
    return


  if i = x'.length:
    return



  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

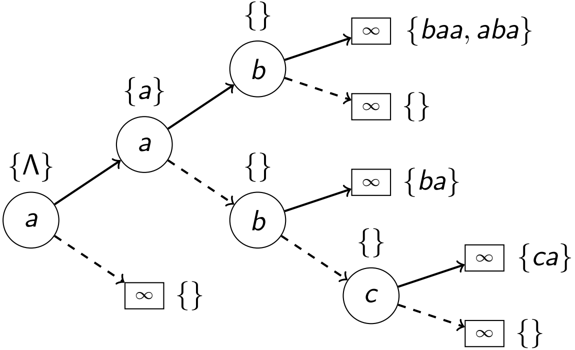
Anatree.`subanagrams(a) = Λ, a`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
  if n.char = ∞:
    return



  if i = x'.length:
    return



  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

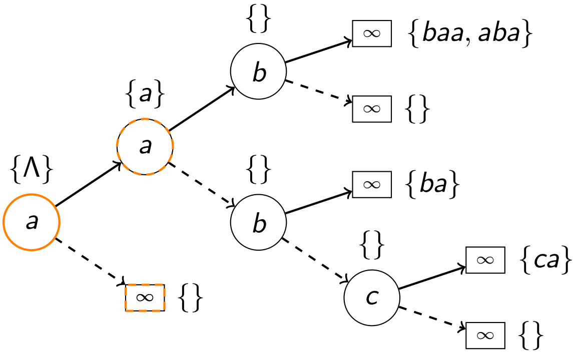
Anatree.`subanagrams(abb) =`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
  if n.char = ∞:
    return



  if i = x'.length:
    return



  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

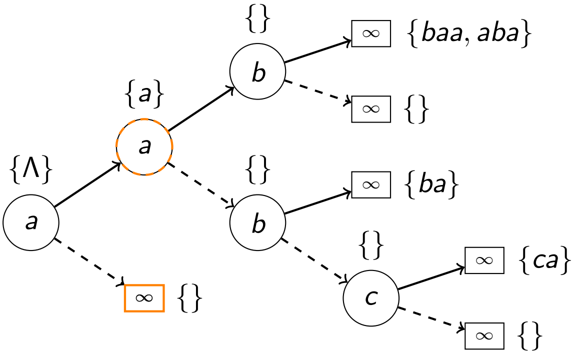
Anatree.`subanagrams(abb) = Λ`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
  if n.char = ∞:
    return


  if i = x'.length:
    return


  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

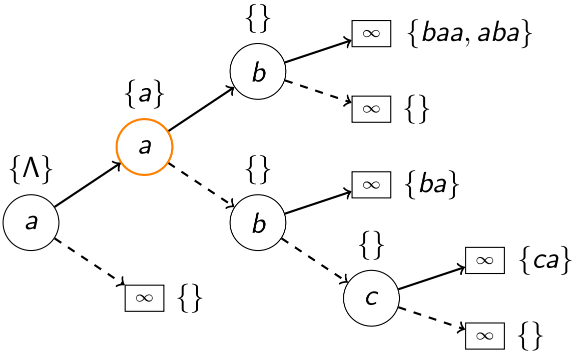
Anatree.`subanagrams(abb) = Λ`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
  if n.char = ∞:
    return


  if i = x'.length:
    return


  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

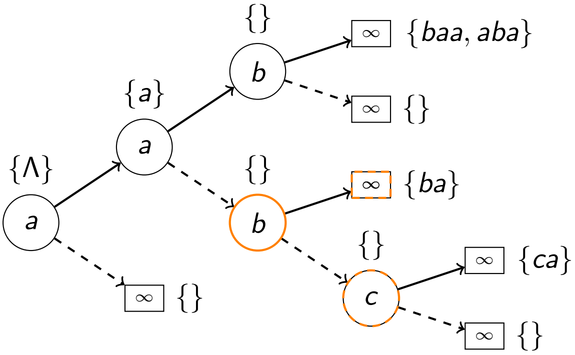
Anatree.subanagrams(abb) = Λ, a



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
  if n.char = ∞:
    return


  if i = x'.length:
    return
  if x'[i] > n.char:
    subanagrams'(n.false, x', i)
  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

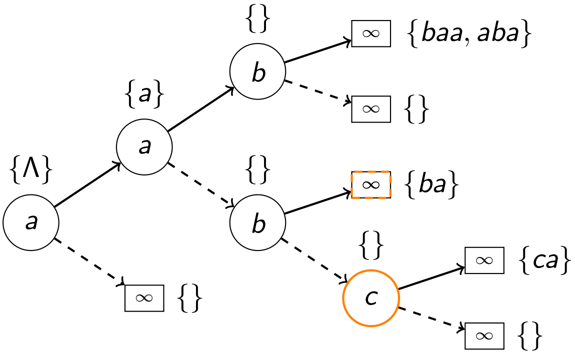
Anatree.`subanagrams(abb) = Λ, a`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
  if n.char = ∞:
    return


  if i = x'.length:
    return
  if x'[i] > n.char:
    subanagrams'(n.false, x', i)
  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

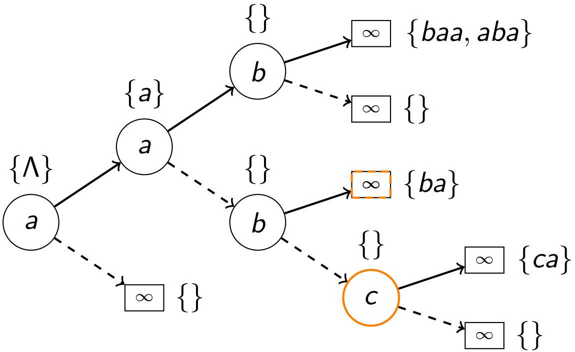
Anatree.`subanagrams(abb) = Λ, a`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
  if n.char = ∞:
    return
  while x'[i] < n.char:
    i++
  if i = x'.length:
    return
  if x'[i] > n.char:
    subanagrams'(n.false, x', i)
  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

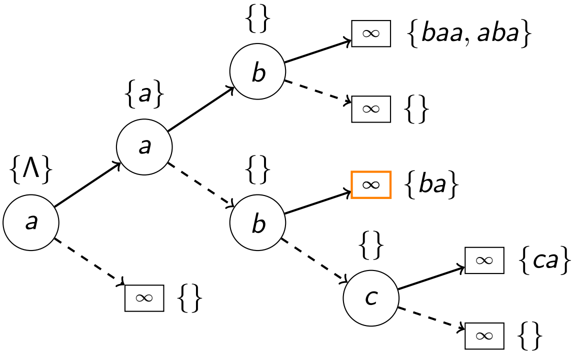
Anatree.`subanagrams(abb) = Λ, a`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
  if n.char = ∞:
    return
  while x'[i] < n.char:
    i++
  if i = x'.length:
    return
  if x'[i] > n.char:
    subanagrams'(n.false, x', i)
  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

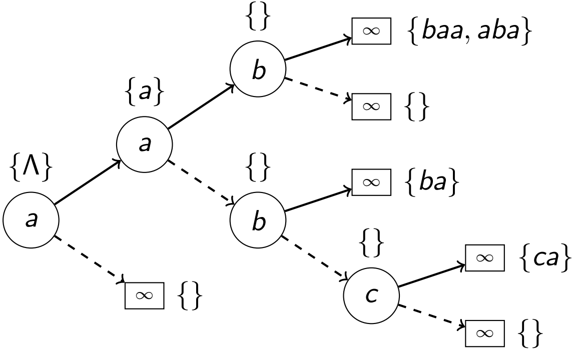
Anatree.`subanagrams(abb) = Λ, a, ab`



$$L = \{\Lambda, a, ba, ca, aba, baa\}$$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
  if n.char = ∞:
    return
  while x'[i] < n.char:
    i++
  if i = x'.length:
    return
  if x'[i] > n.char:
    subanagrams'(n.false, x', i)
  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

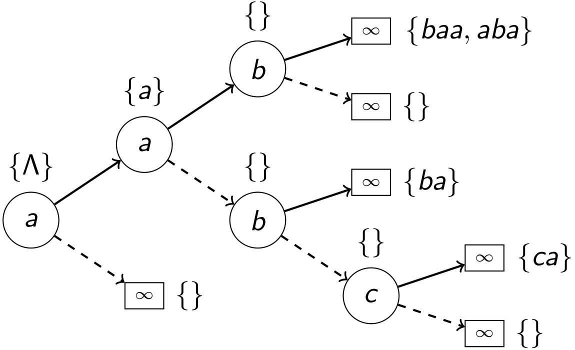
Anatree.subanagrams(abb) = Λ, a, ab



$L = \{\Lambda, a, ba, ca, aba, baa\}$

```
subanagrams(x):
  subanagrams'(root, sort(x), 0)

subanagrams'(n, x', i):
  output words in n
  if n.char = ∞:
    return
  while x'[i] < n.char:
    i++
  if i = x'.length:
    return
  if x'[i] > n.char:
    subanagrams'(n.false, x', i)
  if x'[i] = n.char:
    subanagrams'(n.false, x', i+1)
    subanagrams'(n.true,  x', i+1)
```

Anatree.`subanagrams(...)`

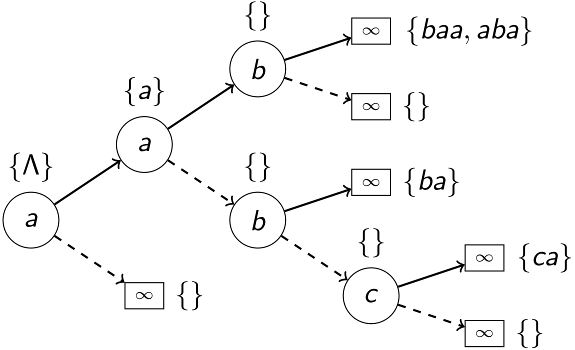


$L = \{\Lambda, a, ba, ca, aba, baa\}$

Lemma

For $N = \sum_{i=1}^{k} |x_i|$, the anatree has size, N_{tree}, at most N.

Anatree.`subanagrams(...)`



$L = \{\Lambda, a, ba, ca, aba, baa\}$

Lemma
For $N = \sum_{i=1}^{k} |x_i|$, the anatree has size, N_{tree}, at most N.

Theorem
`subanagrams(x)` runs in $\mathcal{O}(sort(|x|) + \min(N_{tree}, 2^{|x|} \cdot |\Sigma|) + T)$ time.

Proof.
It takes $\mathcal{O}(\text{sort}(|x|))$ time to sort x and another $\mathcal{O}(T)$ to write the output.

For every match, the recursion splits in two. Each of these $2^{|x|}$ matches have $|\Sigma|$ or fewer mismatches. □

Anatree.`keys(...)`

Definition
The subset L' of $L \subseteq \Sigma^*$ is a set of keys w.r.t. Ψ if for all $x, y \in L'$ then $\Psi(x) \neq \Psi(y)$.

Anatree.`keys(...)`

Definition
The subset L' of $L \subseteq \Sigma^*$ is a set of keys w.r.t. Ψ if for all $x, y \in L'$ then $\Psi(x) \neq \Psi(y)$.

Theorem
`keys(length)` runs in $\mathcal{O}(\min(N_{tree}, 2^{\mathtt{length}} \cdot |\Sigma|) + T)$ time.

Proof.
Left as an exercise to the reader... $\square$

Anatree.`insert(Λ)`

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
```

$\{\}$

$\boxed{\infty}$

Anatree.insert(Λ)

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
```

$\{\Lambda\}$

$\boxed{\infty}$

```
  return n
```

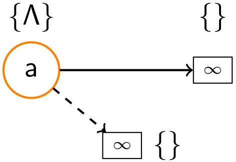
Anatree.`insert`(*ba*)

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)




  return n
```

$\{\Lambda\}$

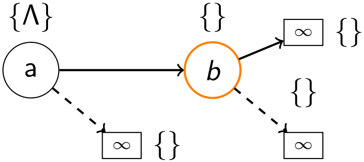
$\boxed{\infty}$

Anatree.`insert`(*ba*)



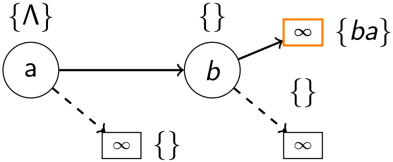

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)

  return n
```

Anatree.`insert(`*ba*`)`



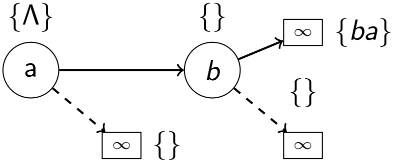

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)



  return n
```

Anatree.`insert`(*ba*)



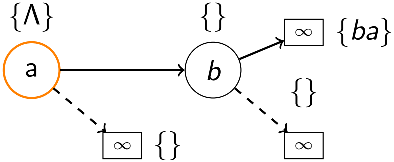

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)

  return n
```

Anatree.insert(a)




```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)




  return n
```

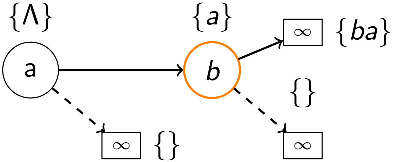
Anatree.`insert(a)`




```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
```

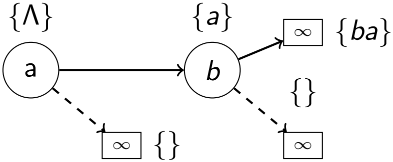
```
  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert(`*a*`)`



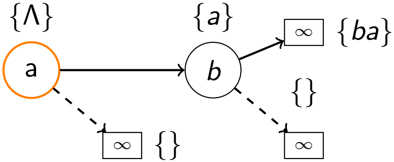

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)




  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert`(*baa*)




```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)










  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

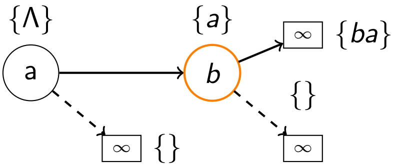
Anatree.`insert`(*baa*)




```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
```

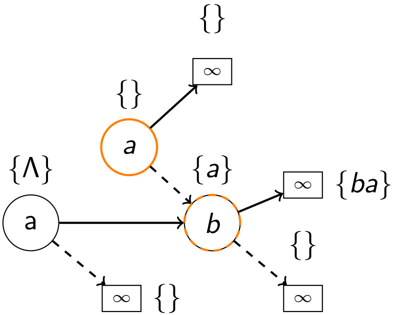
```
  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert`(*baa*)



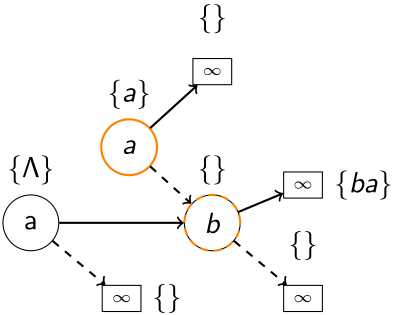

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert`(*baa*)



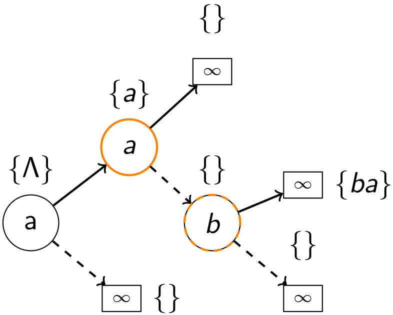

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert`(*baa*)



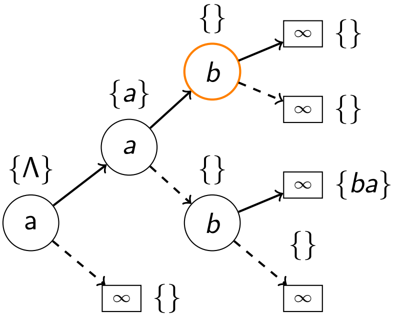

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert`(*baa*)



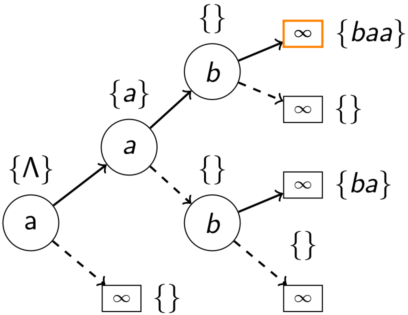

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.insert(*baa*)



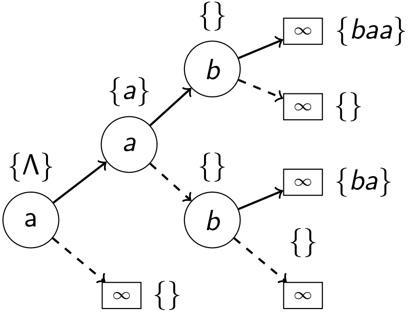

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert(`*baa*`)`



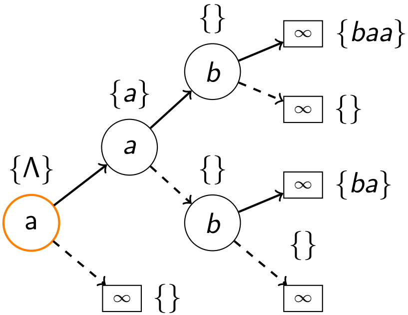

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert`(*aba*)



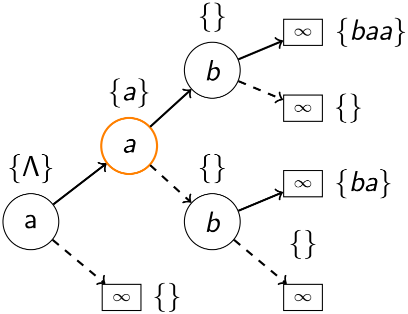

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert`(*aba*)



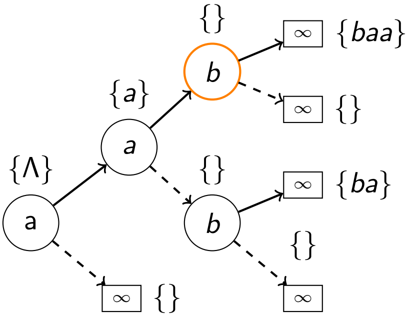

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert`(*aba*)



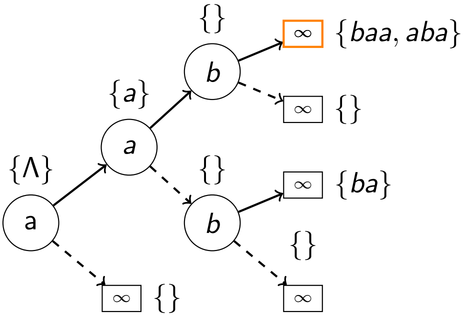

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.insert(*aba*)



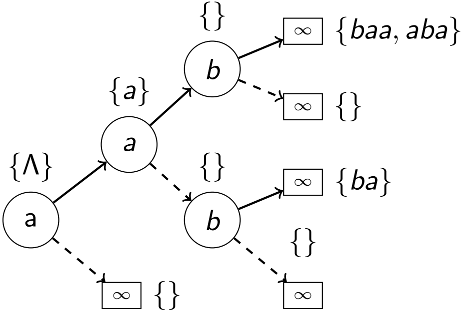

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert`(*aba*)



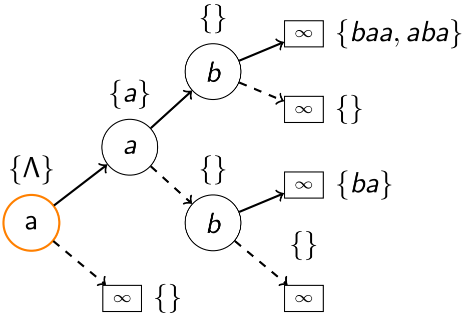

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert(`*ca*`)`



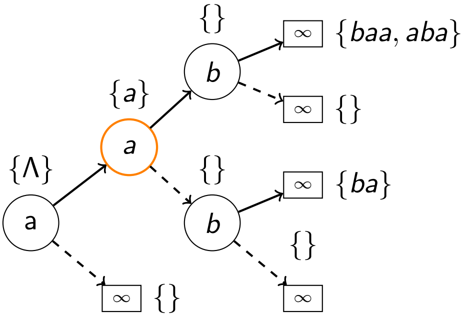

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert(`*ca*`)`



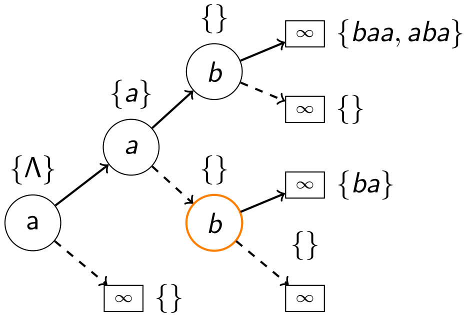

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'


  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert`(ca)



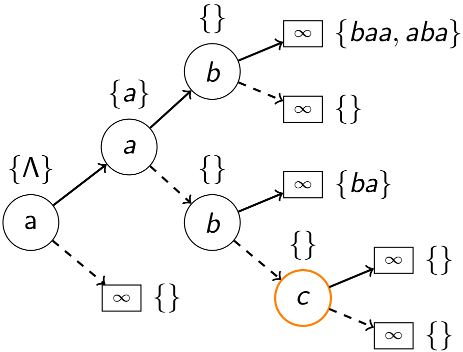

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'
  else if x'[i] > m.char:
    n.false = insert'(n.false, x', i, x)
  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert`(ca)



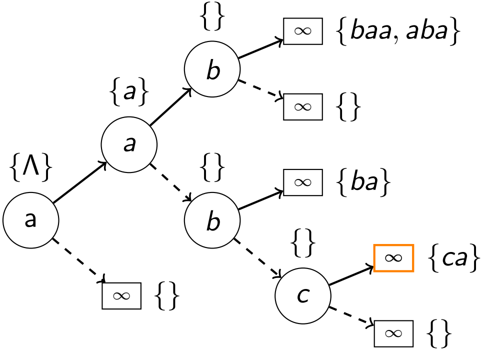

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'
  else if x'[i] > m.char:
    n.false = insert'(n.false, x', i, x)
  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert`(*ca*)



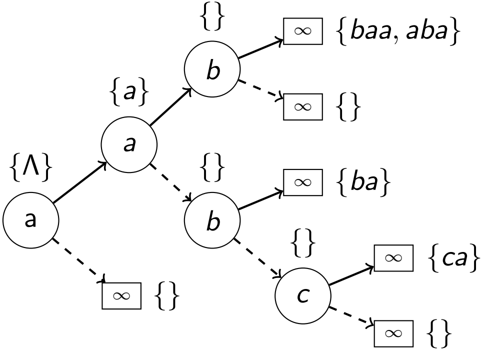

```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'
  else if x'[i] > m.char:
    n.false = insert'(n.false, x', i, x)
  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert(`*ca*`)`




```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'
  else if x'[i] > m.char:
    n.false = insert'(n.false, x', i, x)
  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert(...)`




```
insert(x):
  root = insert'(root, sort(x), 0, x)

insert'(n, x', i, x):
  if i = x'.length:
    n.insert(x)
  else if n.char = ∞:
    n = node{ char: x'[i], false: ∞, true: ∞ }
    n.true = insert'(n.true, x', i+1, x)
  else if x'[i] < m.char:
    n' = node{ char: x'[i], false: n, true: ∞ }
    move n.words into n'.words
    n'.true = insert'(n'.true, x', i+1, x)
    return n'
  else if x'[i] > m.char:
    n.false = insert'(n.false, x', i, x)
  else if x'[i] == m.char:
    n.true = insert'(n.true, x', i+1, x)
  return n
```

Anatree.`insert(...)`

Theorem
`insert(x)` runs in $\mathcal{O}(sort(|x|) + \Sigma)$ time.

Proof.
Similar argument as for `find(`n`, `x'`, `i`)`. $\square$

Anatree.`insert(...)`

Theorem
`insert(x)` runs in $\mathcal{O}(sort(|x|) + \Sigma)$ time.

Proof.
Similar argument as for `find(`n`, `x'`, `i`)`. $\square$

Corollary
For $N = \sum_{i=1}^{k} |x_i|$, `insert(`x_1`, `x_2`, ..., `x_k`)` requires $\mathcal{O}(sort(N) + k \cdot |\Sigma|)$ time.

Proof.
Follows from complexity of `insert(`x_i`)` and `sort` distributes over $+$ in $\mathcal{O}$-notation:

$$\mathcal{O}(\text{sort}(N_1) + \text{sort}(N_2)) = \mathcal{O}(\text{sort}(N_1 + N_2)) \quad \square$$

Theorem
`delete(x)` runs in $\mathcal{O}(sort(|x|) + |\Sigma|)$ time.

Proof.
Left as an exercise to the reader... $\square$

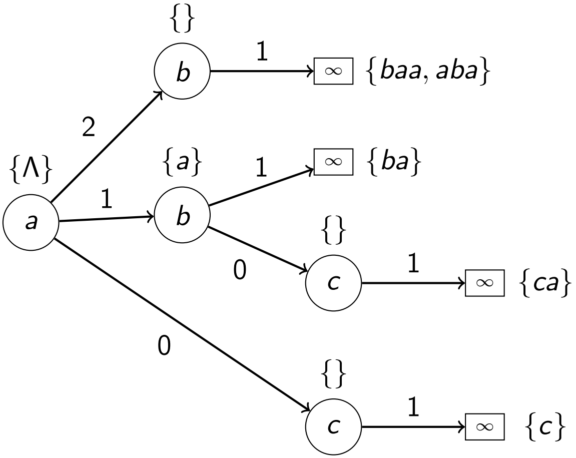
Anatree

		Dictionary		Anatree			
		# Words	# Symbols	Size	#Keys	`insert` (s)	`subanagrams` (s)
	DK	32863	177308	62687	8513	12.62	1.05
	DE	23587	127562	55047	8201	9.46	0.88
	EN	40804	218342	75697	11741	10.62	1.43
	ES	39650	219776	56103	7502	8.45	0.89

Contents

- Motivation
  - Wordrow
- Anagrams
- Binary Anatree
  - `contains(x)`
  - `anagrams(x)`
  - `subanagrams(x)`
  - `insert(x)`
- Multi-valued Anatree
- Letter Ordering

Multi-valued Anatree



$$L = \{\Lambda, a, c, ba, ca, aba, baa\}$$

Contents

Motivation

- Wordrow

Anagrams

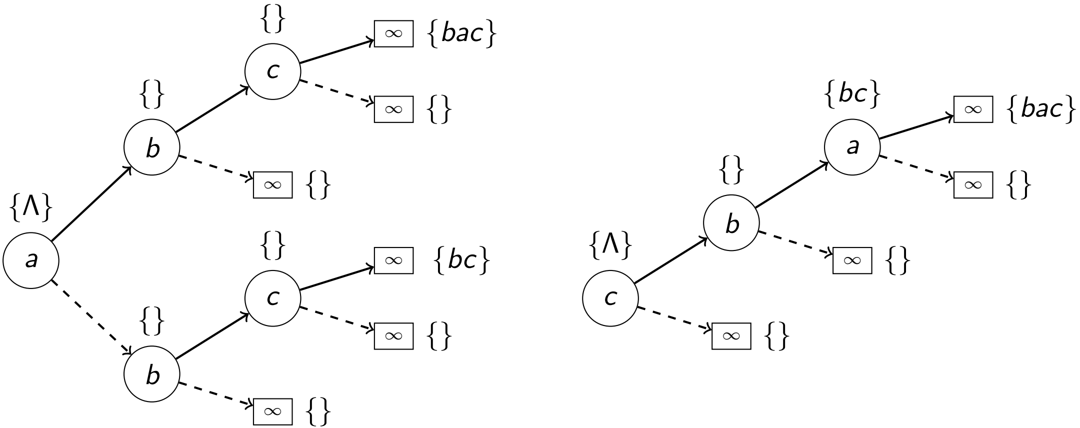
Binary Anatree

- `contains(x)`
- `anagrams(x)`
- `subanagrams(x)`
- `insert(x)`

Multi-valued Anatree

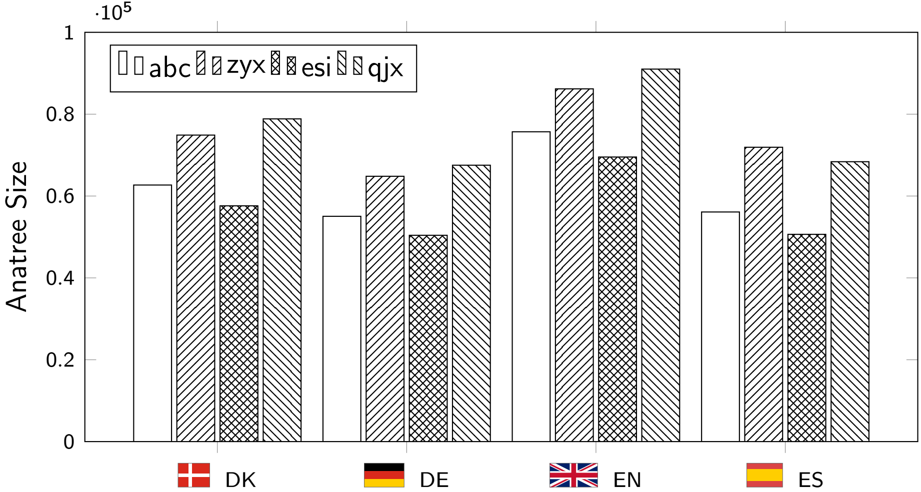
Letter Ordering

Letter Ordering

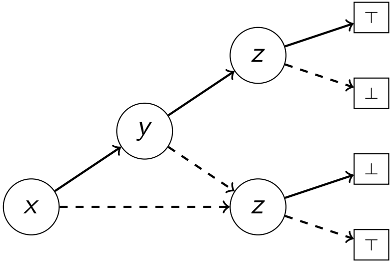


$$L = \{\Lambda, bc, bac\}$$

Letter Ordering

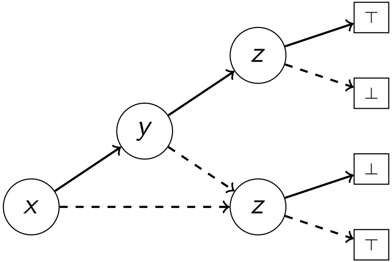


Binary Decision Diagrams



$$f(x, y, z) \equiv \neg((x \wedge y) \oplus z)$$

Binary Decision Diagrams

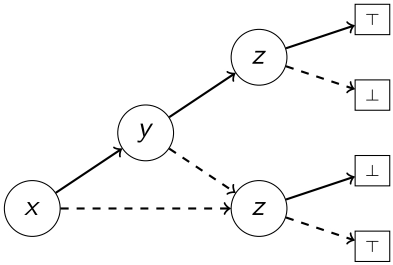


$$f(x, y, z) \equiv \neg((x \wedge y) \oplus z)$$

Used in the context of:

- Model Checking
- Compilers
- Game Solving

Binary Decision Diagrams



$$f(x, y, z) \equiv \neg((x \wedge y) \oplus z)$$

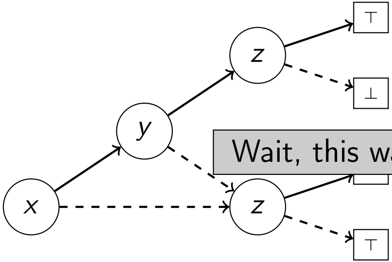
Used in the context of:

- Model Checking
- Compilers
- Game Solving

Features of BDDs:

- (Often) Smaller than Formula/Set
- Operation Complexity depends on BDD Size
- Size depends on Variable Ordering

Binary Decision Diagrams



$$f(x, y, z) \equiv \neg((x \wedge y) \oplus z)$$

Used in the context of:

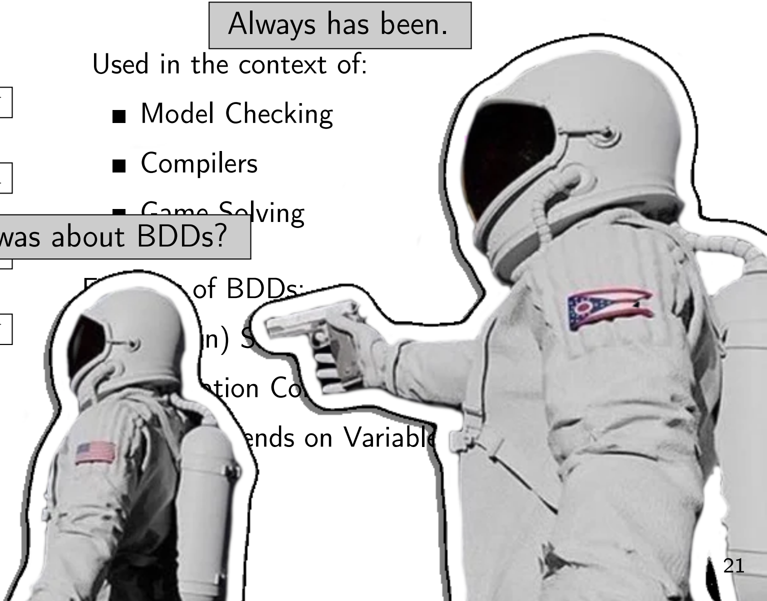
- Model Checking
- Compilers
- Game Solving

of BDDs:

n) S

tion Co

ends on Variable



Steffan Christ Sølvsten

soelvsten@cs.au.dk

Wordrow

wordrow.io

github.com/ssoelvsten/wordrow

Anatree

github.com/ssoelvsten/anatree

AARHUS
UNIVERSITY