Implementacja wydajnych struktur danych do praktycznych operacji na słowach

(Implementation of efficient data structures for practical string operations)

Michał Górniak

Instytut Informatyki Uniwersytetu Wrocławskiego

11 września 2020

Opis problemu

Chcemy utrzymywać dynamiczny zbiór słów $\mathcal{S}$.

Opis problemu

Chcemy utrzymywać dynamiczny zbiór słów $\mathcal{S}$.

Słowa w tym słowniku będą mogły być bardzo długie, więc przy dodawaniu nowych słów, będziemy dostawać od programu etykietę ℓ, dzięki której będziemy mogli łatwo odwoływać się do wybranych słów.

Opis problemu

Chcemy utrzymywać dynamiczny zbiór słów $\mathcal{S}$.

Słowa w tym słowniku będą mogły być bardzo długie, więc przy dodawaniu nowych słów, będziemy dostawać od programu etykietę ℓ, dzięki której będziemy mogli łatwo odwoływać się do wybranych słów.

Oznaczmy przez $\mathcal{W}(\ell)$ słowo reprezentowane przez etykietę ℓ.

Opis problemu

Chcemy utrzymywać dynamiczny zbiór słów $\mathcal{S}$.

Słowa w tym słowniku będą mogły być bardzo długie, więc przy dodawaniu nowych słów, będziemy dostawać od programu etykietę ℓ, dzięki której będziemy mogli łatwo odwoływać się do wybranych słów.

Oznaczmy przez $\mathcal{W}(\ell)$ słowo reprezentowane przez etykietę ℓ.

W każdym momencie możemy modyfikować słownik lub zadawać zapytania:

- $\texttt{make\_string}(w)$ – $\mathcal{S} \coloneqq \mathcal{S} \cup \{w\}$,
- $\texttt{concat}(\ell_1, \ell_2)$ – $\mathcal{S} \coloneqq \mathcal{S} \cup \{\mathcal{W}(\ell_1)\mathcal{W}(\ell_2)\}$,
- $\texttt{split}(\ell, p)$ – $\mathcal{S} \coloneqq \mathcal{S} \cup \{\mathcal{W}(\ell)_{\dots p}, \mathcal{W}(\ell)_{(p+1)\dots}\}$,

Opis problemu

Chcemy utrzymywać dynamiczny zbiór słów $\mathcal{S}$.

Słowa w tym słowniku będą mogły być bardzo długie, więc przy dodawaniu nowych słów, będziemy dostawać od programu etykietę ℓ, dzięki której będziemy mogli łatwo odwoływać się do wybranych słów.

Oznaczmy przez $\mathcal{W}(\ell)$ słowo reprezentowane przez etykietę ℓ.

W każdym momencie możemy modyfikować słownik lub zadawać zapytania:

- $\texttt{make\_string}(w)$ – $\mathcal{S} := \mathcal{S} \cup \{w\}$,
- $\texttt{concat}(\ell_1, \ell_2)$ – $\mathcal{S} := \mathcal{S} \cup \{\mathcal{W}(\ell_1)\mathcal{W}(\ell_2)\}$,
- $\texttt{split}(\ell, p)$ – $\mathcal{S} := \mathcal{S} \cup \{\mathcal{W}(\ell)_{\dots p}, \mathcal{W}(\ell)_{(p+1)\dots}\}$,
- $\texttt{equals}(\ell_1, \ell_2)$ – zwróć $\mathcal{W}(\ell_1) = \mathcal{W}(\ell_2)$,
- $\texttt{smaller}(\ell_1, \ell_2)$ – zwróć $\mathcal{W}(\ell_1) <_{\text{lex}} \mathcal{W}(\ell_2)$,
- $\texttt{lcp}(\ell_1, \ell_2)$ – zwróć najdłuższy wspólny prefiks $\mathcal{W}(\ell_1)$ i $\mathcal{W}(\ell_2)$.

Tabela złożoności (modyfikacje słownika)

Niech n oznacza długość słowa, które jest podane jako pierwszy argument funkcji.

Tabela złożoności (modyfikacje słownika)

Niech n oznacza długość słowa, które jest podane jako pierwszy argument funkcji.

Niech m oznacza długość słowa, które jest podane jako drugi argument funkcji (jeśli istnieje).

Tabela złożoności (modyfikacje słownika)

Niech n oznacza długość słowa, które jest podane jako pierwszy argument funkcji.

Niech m oznacza długość słowa, które jest podane jako drugi argument funkcji (jeśli istnieje).

Rozwiązanie	`make_string`	`concat`	`split`
Naiwne	deterministyczna $\mathcal{O}(n)$	deterministyczna $\mathcal{O}(n+m)$	deterministyczna $\mathcal{O}(n)$
Drzewce	oczekiwana $\mathcal{O}(n)$	oczekiwana $\mathcal{O}(\log(n+m))$	oczekiwana $\mathcal{O}(\log n)$
Parsingi	oczekiwana $\mathcal{O}(n)$	oczekiwana $\mathcal{O}(\log(n+m))$	oczekiwana $\mathcal{O}(\log n)$
Gawrychowski et al.	$\mathcal{O}(n)$ w.h.p.	$\mathcal{O}(\log(n+m))$ w.h.p.	$\mathcal{O}(\log n)$ w.h.p.
Mehlhorn et al.	deterministyczna $\mathcal{O}(n)$	deterministyczna $\mathcal{O}(\log^2(n+m)\log^\star(n+m))$	deterministyczna $\mathcal{O}(\log^2 n \log^\star n)$
Alstrup et al.	$\mathcal{O}(n)$ w.h.p.	$\mathcal{O}(\log(n+m)\log^\star(n+m))$ w.h.p.	$\mathcal{O}(\log n \log^\star n)$ w.h.p.

Tabela złożoności (zapytania do struktury)

Rozwiązanie	`equals`	`smaller`	`lcp`
Naiwne	deterministyczna $\mathcal{O}(\min(n, m))$	deterministyczna $\mathcal{O}(\min(n, m))$	deterministyczna $\mathcal{O}(\min(n, m))$
Drzewce	Monte Carlo $\mathcal{O}(1)$	Monte Carlo $\mathcal{O}(\log^2(n+m))$	Monte Carlo $\mathcal{O}(\log^2(n+m))$
Parsingi	deterministyczna $\mathcal{O}(1)$	oczekiwana $\mathcal{O}(\log(n+m))$	oczekiwana $\mathcal{O}(\log(n+m))$
Parsingi (wolne `lcp`)	deterministyczna $\mathcal{O}(1)$	oczekiwana $\mathcal{O}(\log^2(n+m))$	oczekiwana $\mathcal{O}(\log^2(n+m))$
Gawrychowski et al.	deterministyczna $\mathcal{O}(1)$	deterministyczna $\mathcal{O}(1)$	deterministyczna $\mathcal{O}(1)$
Mehlhorn et al.	deterministyczna $\mathcal{O}(1)$	—	—
Alstrup et al.	deterministyczna $\mathcal{O}(1)$	deterministyczna $\mathcal{O}(1)$	deterministyczna $\mathcal{O}(1)$

O implementacji

Struktura kodu

Kod został podzielony na trzy główne części:

O implementacji

Struktura kodu

Kod został podzielony na trzy główne części:

- pliki `run.cpp` i `makefile`

O implementacji

Struktura kodu

Kod został podzielony na trzy główne części:

- pliki `run.cpp` i `makefile`
- folder `solutions/`

O implementacji

Struktura kodu

Kod został podzielony na trzy główne części:

- pliki `run.cpp` i `makefile`
- folder `solutions/`
- folder `testers/`

O implementacji

Struktura kodu

Kod został podzielony na trzy główne części:

- pliki `run.cpp` i `makefile`
- folder `solutions/`
- folder `testers/`

Interfejs abstrakcyjnej klasy `solution.h`:

```
class solution {
    public:
        virtual ~solution() = 0;
        virtual int make_string(std::vector<int> &word) = 0;
        virtual int concat(int label1, int label2) = 0;
        virtual std::pair<int, int> split(int label, int position) = 0;
        virtual bool equals(int label1, int label2) = 0;
        virtual bool smaller(int label1, int label2) = 0;
        virtual int longest_common_prefix(int label1, int label2) = 0;
};
```

Drzewce

Każde drzewo zbalansowane reprezentuje jedno słowo ze słownika.

Drzewce

Każde drzewo zbalansowane reprezentuje jedno słowo ze słownika.

Używamy drzew zbalansowanych, tak aby można było:

Drzewce

Każde drzewo zbalansowane reprezentuje jedno słowo ze słownika.

Używamy drzew zbalansowanych, tak aby można było:

- łączyć dwa drzewa i dzielić jedno w czasie logarytmicznym,

Drzewce

Każde drzewo zbalansowane reprezentuje jedno słowo ze słownika.

Używamy drzew zbalansowanych, tak aby można było:

- łączyć dwa drzewa i dzielić jedno w czasie logarytmicznym,
- w każdym wierzchołku trzymać hasz całego jego poddrzewa,

Drzewce

Każde drzewo zbalansowane reprezentuje jedno słowo ze słownika.

Używamy drzew zbalansowanych, tak aby można było:

- łączyć dwa drzewa i dzielić jedno w czasie logarytmicznym,
- w każdym wierzchołku trzymać hasz całego jego poddrzewa,
- porównywać hasze wybranych wierzchołków dla sprawdzenia $=$,

Drzewce

Każde drzewo zbalansowane reprezentuje jedno słowo ze słownika.

Używamy drzew zbalansowanych, tak aby można było:

- łączyć dwa drzewa i dzielić jedno w czasie logarytmicznym,
- w każdym wierzchołku trzymać hasz całego jego poddrzewa,
- porównywać hasze wybranych wierzchołków dla sprawdzenia $=$,
- wyszukiwać binarnie po wyniku, aby wyliczać `lcp` i $<_{\text{lex}}$.

Drzewce

Każde drzewo zbalansowane reprezentuje jedno słowo ze słownika.

Używamy drzew zbalansowanych, tak aby można było:

- łączyć dwa drzewa i dzielić jedno w czasie logarytmicznym,
- w każdym wierzchołku trzymać hasz całego jego poddrzewa,
- porównywać hasze wybranych wierzchołków dla sprawdzenia $=$,
- wyszukiwać binarnie po wyniku, aby wyliczać `lcp` i $<_{\text{lex}}$.

Rotacje

Implementacja drzewców nie wymaga implementacji rotacji drzewa. Drzewo musi zachowywać porządek kopcowy patrząc na priorytety, a porządek BST patrząc na klucze. W ten sposób zawsze powstaje **dokładnie** jeden drzewiec.

Drzewce

Każde drzewo zbalansowane reprezentuje jedno słowo ze słownika.

Używamy drzew zbalansowanych, tak aby można było:

- łączyć dwa drzewa i dzielić jedno w czasie logarytmicznym,
- w każdym wierzchołku trzymać hasz całego jego poddrzewa,
- porównywać hasze wybranych wierzchołków dla sprawdzenia $=$,
- wyszukiwać binarnie po wyniku, aby wyliczać `lcp` i $<_{\text{lex}}$.

Rotacje

Implementacja drzewców nie wymaga implementacji rotacji drzewa. Drzewo musi zachowywać porządek kopcowy patrząc na priorytety, a porządek BST patrząc na klucze. W ten sposób zawsze powstaje **dokładnie** jeden drzewiec.

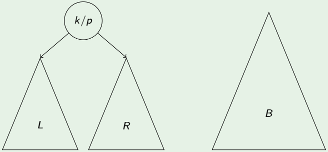
Przykład

W kolejnym slajdzie opiszemy działanie funkcji `merge`.

Operacja `merge` (teoria)

Przykład

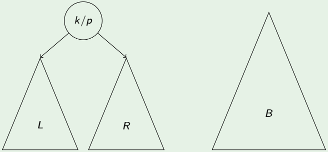
Załóżmy, że chcemy złączyć drzewce A oraz B i bez straty ogólności A ma większy priorytet w korzeniu niż B.



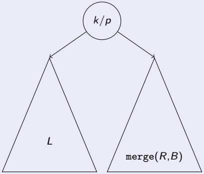
Operacja `merge` (teoria)

Przykład

Załóżmy, że chcemy złączyć drzewce A oraz B i bez straty ogólności A ma większy priorytet w korzeniu niż B.



Wystarczy zatem wywołać rekurencyjnie `merge`(R, B) i podpiąć odpowiednio wskaźniki.



Operacja merge (implementacja)

```
balanced_trees::treap *balanced_trees::merge(balanced_trees::treap *t1,
                                             balanced_trees::treap *t2) {
    if (t1 == nullptr) {
        return t2;
    }
    if (t2 == nullptr) {
        return t1;
    }
    if (t1->priority > t2->priority) {
        auto return_treap = new treap(*t1);
        return_treap->right_subtree = merge(t1->right_subtree, t2);
        update_values(return_treap);
        return return_treap;
    } else {
        auto return_treap = new treap(*t2);
        return_treap->left_subtree = merge(t1, t2->left_subtree);
        update_values(return_treap);
        return return_treap;
    }
}
```

Fakty o drzewcach

Operacja `merge` (implementacja)

```
balanced_trees::treap *balanced_trees::merge(balanced_trees::treap *t1,
                                              balanced_trees::treap *t2) {
    if (t1 == nullptr) {
        return t2;
    }
    if (t2 == nullptr) {
        return t1;
    }
    if (t1->priority > t2->priority) {
        auto return_treap = new treap(*t1);
        return_treap->right_subtree = merge(t1->right_subtree, t2);
        update_values(return_treap);
        return return_treap;
    } else {
        auto return_treap = new treap(*t2);
        return_treap->left_subtree = merge(t1, t2->left_subtree);
        update_values(return_treap);
        return return_treap;
    }
}
```

Fakty o drzewcach

- mają wysokość logarytmiczną w.h.p.

Operacja merge (implementacja)

```
balanced_trees::treap *balanced_trees::merge(balanced_trees::treap *t1,
                                             balanced_trees::treap *t2) {
    if (t1 == nullptr) {
        return t2;
    }
    if (t2 == nullptr) {
        return t1;
    }
    if (t1->priority > t2->priority) {
        auto return_treap = new treap(*t1);
        return_treap->right_subtree = merge(t1->right_subtree, t2);
        update_values(return_treap);
        return return_treap;
    } else {
        auto return_treap = new treap(*t2);
        return_treap->left_subtree = merge(t1, t2->left_subtree);
        update_values(return_treap);
        return return_treap;
    }
}
```

Fakty o drzewcach

- mają wysokość logarytmiczną w.h.p.
- porównywanie podsłów również jest w.h.p. (hasze Karpa-Rabina)

Drzewa symboli (*parse trees*)

Utrzymujemy dynamiczną gramatykę bezkontekstową bez symbolu startowego.

Drzewa symboli (*parse trees*)

Utrzymujemy dynamiczną gramatykę bezkontekstową bez symbolu startowego.

Każdy symbol generuje **dokładnie jedno słowo**.

Drzewa symboli (*parse trees*)

Utrzymujemy dynamiczną gramatykę bezkontekstową bez symbolu startowego.

Każdy symbol generuje **dokładnie jedno słowo**.

Każdy symbol ma przypisany poziom, na którym występuje. Symbole powstają podczas tworzenia *drzewa symbolu* nad nowym słowem.

Drzewa symboli (*parse trees*)

Utrzymujemy dynamiczną gramatykę bezkontekstową bez symbolu startowego.

Każdy symbol generuje **dokładnie jedno słowo**.

Każdy symbol ma przypisany poziom, na którym występuje. Symbole powstają podczas tworzenia *drzewa symbolu* nad nowym słowem.

Tworzenie drzewa symbolu

Polega na naprzemiennym wykonywaniu kompresji typu:

Drzewa symboli (*parse trees*)

Utrzymujemy dynamiczną gramatykę bezkontekstową bez symbolu startowego.

Każdy symbol generuje **dokładnie jedno słowo**.

Każdy symbol ma przypisany poziom, na którym występuje. Symbole powstają podczas tworzenia *drzewa symbolu* nad nowym słowem.

Tworzenie drzewa symbolu

Polega na naprzemiennym wykonywaniu kompresji typu:

- RLE (*run-length encoding*),

Drzewa symboli (*parse trees*)

Utrzymujemy dynamiczną gramatykę bezkontekstową bez symbolu startowego.

Każdy symbol generuje **dokładnie jedno słowo**.

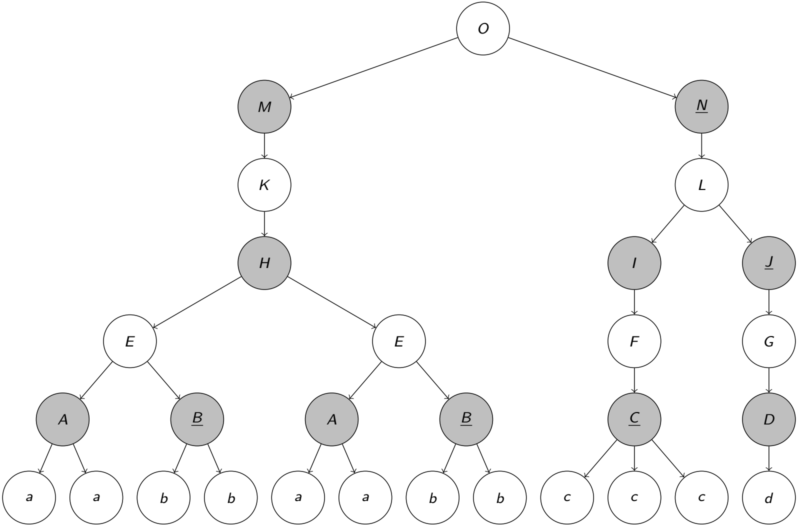
Każdy symbol ma przypisany poziom, na którym występuje. Symbole powstają podczas tworzenia *drzewa symbolu* nad nowym słowem.

Tworzenie drzewa symbolu

Polega na naprzemiennym wykonywaniu kompresji typu:

- RLE (*run-length encoding*),
- SHRINK, która zależy od losowych bitów symboli.

Przykładowe drzewo symbolu



Twierdzenie o oczekiwanym czasie budowy drzewa symbolu

Twierdzenie

Oczekiwany czas budowy drzewa nad słowem długości n, wynosi $\mathcal{O}(n)$.

Twierdzenie o oczekiwanym czasie budowy drzewa symbolu

Twierdzenie

Oczekiwany czas budowy drzewa nad słowem długości n, wynosi $\mathcal{O}(n)$.

Fakt z pracy *Optimal Dynamic Strings*

Po zastosowaniu kompresji typu RLE, a następnie kompresji typu SHRINK na dowolnym słowie w, oczekiwana długość słowa w' powstałego w wyniku kompresji jest nie większa niż $\frac{3}{4}|w| + \frac{1}{4}$.

Twierdzenie o oczekiwanym czasie budowy drzewa symbolu

Twierdzenie

Oczekiwany czas budowy drzewa nad słowem długości n, wynosi $\mathcal{O}(n)$.

Fakt z pracy *Optimal Dynamic Strings*

Po zastosowaniu kompresji typu RLE, a następnie kompresji typu SHRINK na dowolnym słowie w, oczekiwana długość słowa w' powstałego w wyniku kompresji jest nie większa niż $\frac{3}{4}|w| + \frac{1}{4}$.

Szkic dowodu

Twierdzenie o oczekiwanym czasie budowy drzewa symbolu

Twierdzenie

Oczekiwany czas budowy drzewa nad słowem długości n, wynosi $\mathcal{O}(n)$.

Fakt z pracy *Optimal Dynamic Strings*

Po zastosowaniu kompresji typu RLE, a następnie kompresji typu SHRINK na dowolnym słowie w, oczekiwana długość słowa w' powstałego w wyniku kompresji jest nie większa niż $\frac{3}{4}|w| + \frac{1}{4}$.

Szkic dowodu

- indukcja względem długości słowa,

Twierdzenie o oczekiwanym czasie budowy drzewa symbolu

Twierdzenie

Oczekiwany czas budowy drzewa nad słowem długości n, wynosi $\mathcal{O}(n)$.

Fakt z pracy *Optimal Dynamic Strings*

Po zastosowaniu kompresji typu RLE, a następnie kompresji typu SHRINK na dowolnym słowie w, oczekiwana długość słowa w' powstałego w wyniku kompresji jest nie większa niż $\frac{3}{4}|w| + \frac{1}{4}$.

Szkic dowodu

- indukcja względem długości słowa,
- wykorzystanie powyższego faktu wraz z nierównością Markowa, aby oszacować prawdopodobieństwo, że długość słowa po dwóch kolejnych fazach kompresji skróci się przez stały czynnik.

Dekompozycja context-insensitive

Motywacja

Nie możemy w prosty sposób (podobnie jak w drzewach zbalansowanych) łączyć dwóch drzew symboli.

Dekompozycja context-insensitive

Motywacja

Nie możemy w prosty sposób (podobnie jak w drzewach zbalansowanych) łączyć dwóch drzew symboli.

Okazuje się jednak, że możemy zawsze znaleźć pewną *warstwę* drzewa symbolu, której RLE jest długości logarytmicznej i jest ona typu context-insensitive.

Dekompozycja context-insensitive

Motywacja

Nie możemy w prosty sposób (podobnie jak w drzewach zbalansowanych) łączyć dwóch drzew symboli.

Okazuje się jednak, że możemy zawsze znaleźć pewną *warstwę* drzewa symbolu, której RLE jest długości logarytmicznej i jest ona typu context-insensitive.

Łączenie słów

Dekompozycja context-insensitive

Motywacja

Nie możemy w prosty sposób (podobnie jak w drzewach zbalansowanych) łączyć dwóch drzew symboli.

Okazuje się jednak, że możemy zawsze znaleźć pewną *warstwę* drzewa symbolu, której RLE jest długości logarytmicznej i jest ona typu context-insensitive.

Łączenie słów

1. Szukamy dekompozycji obu słów, a następnie je łączymy.

Dekompozycja context-insensitive

Motywacja

Nie możemy w prosty sposób (podobnie jak w drzewach zbalansowanych) łączyć dwóch drzew symboli.

Okazuje się jednak, że możemy zawsze znaleźć pewną *warstwę* drzewa symbolu, której RLE jest długości logarytmicznej i jest ona typu context-insensitive.

Łączenie słów

1. Szukamy dekompozycji obu słów, a następnie je łączymy.
2. Lista ta będzie dekompozycją konkatenacji początkowych dwóch słów.

Dekompozycja context-insensitive

Motywacja

Nie możemy w prosty sposób (podobnie jak w drzewach zbalansowanych) łączyć dwóch drzew symboli.

Okazuje się jednak, że możemy zawsze znaleźć pewną *warstwę* drzewa symbolu, której RLE jest długości logarytmicznej i jest ona typu context-insensitive.

Łączenie słów

1. Szukamy dekompozycji obu słów, a następnie je łączymy.
2. Lista ta będzie dekompozycją konkatenacji początkowych dwóch słów.
3. Budujemy górną część drzewa symbolu, jak przy budowaniu od liści.

Dekompozycja context-insensitive

Motywacja

Nie możemy w prosty sposób (podobnie jak w drzewach zbalansowanych) łączyć dwóch drzew symboli.

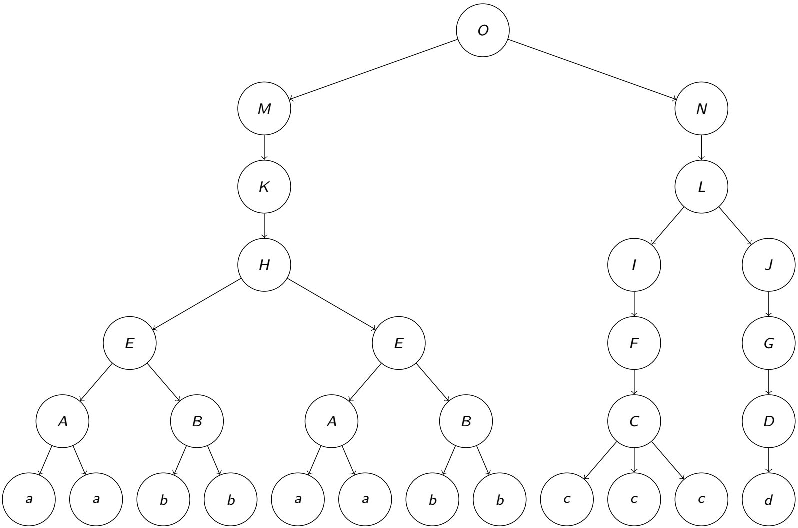
Okazuje się jednak, że możemy zawsze znaleźć pewną *warstwę* drzewa symbolu, której RLE jest długości logarytmicznej i jest ona typu context-insensitive.

Łączenie słów

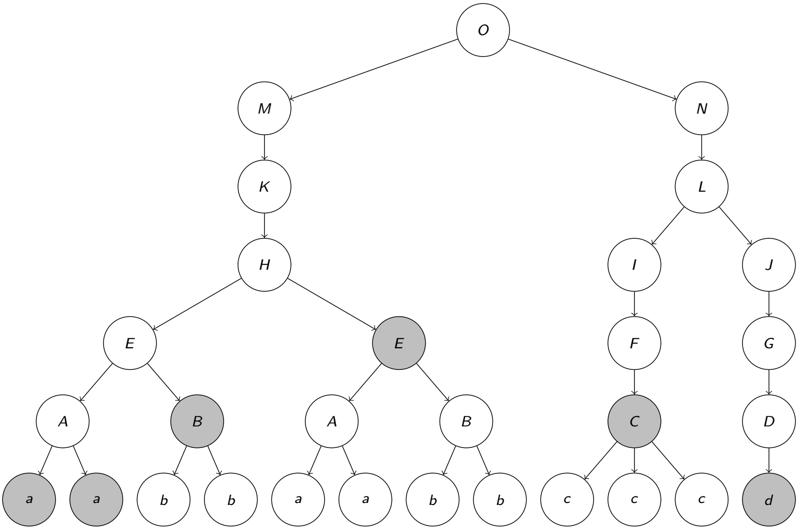
1. Szukamy dekompozycji obu słów, a następnie je łączymy.
2. Lista ta będzie dekompozycją konkatenacji początkowych dwóch słów.
3. Budujemy górną część drzewa symbolu, jak przy budowaniu od liści.

Dzielenie polega na wybraniu dekompozycji typu context-insensitive wybranego podsłowa i zbudowaniu nad nią drzewa symbolu.

Przykład dekompozycji



Przykład dekompozycji



Różnice w implementacji względem oryginalnej pracy

Ze względu na zaawansowanie struktury opisanej w pracy, przyjmujemy pewne uproszczenia.

Różnice w implementacji względem oryginalnej pracy

Ze względu na zaawansowanie struktury opisanej w pracy, przyjmujemy pewne uproszczenia.

Główne różnice w implementacji względem oryginalnej pracy:

- przy budowie drzewa **dopuszczamy** jednoelementowe produkcje,

Różnice w implementacji względem oryginalnej pracy

Ze względu na zaawansowanie struktury opisanej w pracy, przyjmujemy pewne uproszczenia.

Główne różnice w implementacji względem oryginalnej pracy:

- przy budowie drzewa **dopuszczamy** jednoelementowe produkcje,
- użycie `std::unordered_map`, która działa w oczekiwanym czasie stałym,

Różnice w implementacji względem oryginalnej pracy

Ze względu na zaawansowanie struktury opisanej w pracy, przyjmujemy pewne uproszczenia.

Główne różnice w implementacji względem oryginalnej pracy:

- przy budowie drzewa **dopuszczamy** jednoelementowe produkcje,
- użycie `std::unordered_map`, która działa w oczekiwanym czasie stałym,
- `concat` działa w złożoności oczekiwanej logarytmicznej,

Różnice w implementacji względem oryginalnej pracy

Ze względu na zaawansowanie struktury opisanej w pracy, przyjmujemy pewne uproszczenia.

Główne różnice w implementacji względem oryginalnej pracy:

- przy budowie drzewa **dopuszczamy** jednoelementowe produkcje,
- użycie `std::unordered_map`, która działa w oczekiwanym czasie stałym,
- `concat` działa w złożoności oczekiwanej logarytmicznej,
- nie zaimplementowano najlepszej metody dla zapytania `lcp` i $<_{\text{lex}}$,

Różnice w implementacji względem oryginalnej pracy

Ze względu na zaawansowanie struktury opisanej w pracy, przyjmujemy pewne uproszczenia.

Główne różnice w implementacji względem oryginalnej pracy:

- przy budowie drzewa **dopuszczamy** jednoelementowe produkcje,
- użycie `std::unordered_map`, która działa w oczekiwanym czasie stałym,
- `concat` działa w złożoności oczekiwanej logarytmicznej,
- nie zaimplementowano najlepszej metody dla zapytania `lcp` i $<_{\text{lex}}$,
- większość operacji traci złożoność w.h.p. na rzecz oczekiwanej.

Różnice w implementacji względem oryginalnej pracy

Ze względu na zaawansowanie struktury opisanej w pracy, przyjmujemy pewne uproszczenia.

Główne różnice w implementacji względem oryginalnej pracy:

- przy budowie drzewa **dopuszczamy** jednoelementowe produkcje,
- użycie `std::unordered_map`, która działa w oczekiwanym czasie stałym,
- `concat` działa w złożoności oczekiwanej logarytmicznej,
- nie zaimplementowano najlepszej metody dla zapytania `lcp` i $<_{\text{lex}}$,
- większość operacji traci złożoność w.h.p. na rzecz oczekiwanej.

Wysokość drzewa

Faktem pozostaje natomiast logarytmiczna wysokość drzewa w.h.p.

Przykładowe uruchomienie

Przykład

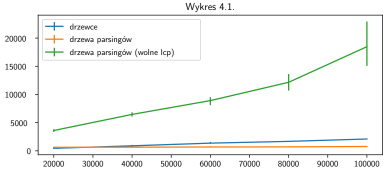
Wywołanie `./run 129873 correctness 1`.

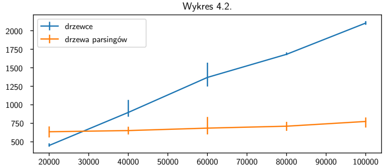
Wydruk z terminala

```
RUNNING RANDOM TESTS
________________________________________
RUNNING BALANCED TREES SOLUTION
[**************************************************] 100%
PASSED 5147518 OUT OF 5147518 TESTS (100%)
RUNNING RANDOM TESTS
________________________________________
RUNNING PARSE TREES LCP LOG^2 SOLUTION
[**************************************************] 100%
PASSED 5146257 OUT OF 5146257 TESTS (100%)
RUNNING RANDOM TESTS
________________________________________
RUNNING PARSE TREES LCP LOG SOLUTION
[**************************************************] 100%
PASSED 5147611 OUT OF 5147611 TESTS (100%)
```

Wykresy czasów działania

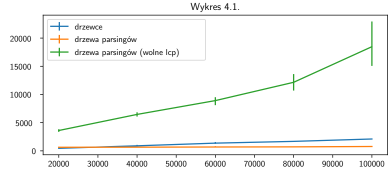
Testy z tablicą sufiksową.

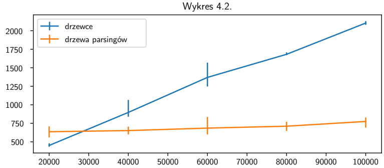




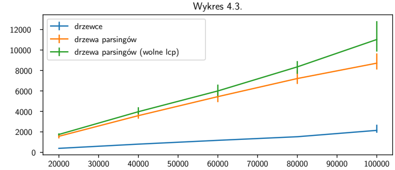
Wykresy czasów działania

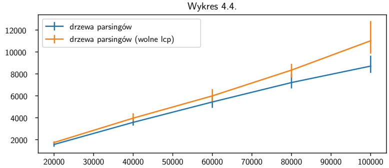
Testy z tablicą sufiksową.





Testy z losowymi poprawnymi operacjami na słowniku.





Dziękuję za uwagę.