

**Implementácia sufixového stromu**

SEMESTRÁLNA PRÁCA

Vypracovali: **Simona Pernišová**

Študijná skupina: **5ZIB11**

Predmet: **Biomolekulárne základy v genetike a biomedicíne**

Cvičiaci:  **doc. Ing. Miroslav Kvaššay, PhD. , RNDr. Denisa Maceková, PhD.**

Obsah

[Zadanie semestrálnej práce 3](#_Toc197384867)

[Príprava prostredia a spustenie projektu 4](#_Toc197384868)

[Prvotné spustenie projektu 4](#_Toc197384869)

[Inštalácia MSYS2 4](#_Toc197384870)

[Spustenie projektu vo VS code 4](#_Toc197384871)

[Použité rozšírenia 5](#_Toc197384872)

[Spustenie vygenerovaného spustiteľného súboru (.exe) 5](#_Toc197384873)

[Základy a definície 6](#_Toc197384874)

[Kľúčové pojmy pre pochopenie sufixových stromov 6](#_Toc197384875)

[Prefix 6](#_Toc197384876)

[Sufix 6](#_Toc197384877)

[Trie (Prefixový strom) 6](#_Toc197384878)

[Kompresia trie 7](#_Toc197384879)

[Sufixový strom 7](#_Toc197384880)

[Zložitosti jednotlivých trie 8](#_Toc197384881)

[Ukkonenov algoritmus 9](#_Toc197384882)

[Základné princípy s ktorými Ukkonenov algoritmus pracuje 9](#_Toc197384883)

[Situácie ktoré môžu nastať počas konštrukcie sufixového stromu 11](#_Toc197384884)

[Vysvetlenie Ukkonenovho algoritmu a príklad 12](#_Toc197384885)

[Implementácia sufixového stromu s použitím Ukkonenovho algoritmu 15](#_Toc197384886)

[Popis programu a súborov 15](#_Toc197384887)

[Experimentálna analýza metódy build 16](#_Toc197384888)

[Analýza dát pre veľkosť abecedy 4 16](#_Toc197384889)

[Analýza dát pre veľkosť abecedy 26 17](#_Toc197384890)

[Analýza dát pre veľkosť abecedy 62 18](#_Toc197384891)

[Vyhodnotenie analýzy 19](#_Toc197384892)

[Záver 19](#_Toc197384893)

# Zadanie semestrálnej práce

Vo vami zvolenom jazyku implementujte sufixový strom ( <https://en.wikipedia.org/wiki/Suffix_tree> ), ktorého vybudovanie bude mať v prípade abecedy s konštantnou veľkosťou ***zložitosť O(n).*** Sufixový strom musí mať minimálne nasledujúce metódy:

* **konštruktor** – vytvorí prázdny sufixový strom;
* **deštruktor** – ak používate jazyk s manuálnou správou pamäte;
* **void clear()** – vyprázdni obsah sufixového stromu;
* **void build(const string& str)** – naplní sufixový strom všetkými sufixami reťazca str. Ak strom nie je prázdny, najskôr ho vyprázdni a následne naplní;
* **bool find(const string& str)** – zistí, či sa daný reťazec nachádza v sufixovom strome;
* **void print()** – s využitím prehliadky preorder vykreslí obsah suffixového stromu.

Na implementáciu metódy **build** je odporúčané použitie **Ukkonenovho algoritmu**

(<https://en.wikipedia.org/wiki/Ukkonen%27s_algorithm>). Detailný opis algoritmu môžete nájsť v nasledujúcej sérii článkov:<https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/>

Súčasťou odovzdaného riešenia musí byť dokumentácia, v ktorej opíšete činnosť **Ukkonenovho algoritmu** a experimentálne analyzujete zložitosť metódy **build** v závislosti od dĺžky reťazca a veľkosti abecedy. Odporúčané veľkosti abecedy sú **4, 26** a **62**. V rámci experimentálnej analýzy je odporúčané použiť grafy, ktoré vytvoríte pomocou nástroja **R.** Pri grafoch nezabudnite vysvetliť, čo predstavujú jednotlivé osi a vyvodiť závery, ktoré potvrdia alebo vyvrátia tvrdenie, že zložitosť metódy build je v prípade abecedy s konštantnou veľkosťou lineárna.

Bodovanie:

* implementácia – 14 bodov:
  + konštruktor – 1 bod;
  + void clear() – 1 bod;
  + void build(const string& str) – 9 bodov;
  + bool find(const string& str) – 1 bod;
  + void print() – 2 body;
* dokumentácia – 10 bodov:
  + opis činnosti Ukkonenovho algoritmu – 4 body;
  + experimentálna analýza – 6 bodov

# Príprava prostredia a spustenie projektu

Tento projekt bol vyvinutý vo vývojovom prostredí **Visual Studio Code** pomocou programovacieho jazyka **C++**. Na preklad zdrojového kódu (kompiláciu), vytváranie spustiteľného súboru (buildovanie) a prácu s nástrojmi potrebnými pre projekt bolo využité prostredie **MSYS2 (Minimal SYStem 2)**. Projekt využíva nástroje **GNU** (unixový operačný systém), konkrétne **kompilátor GCC (GNU Compiler Collection)** a **debugger GDB (GNU Debugger)** pre prípadné ladenie.

## Prvotné spustenie projektu

V prípade problémov so spustením v rámci Visual Studio Code postupujte podľa krokov v sekcii „**Spustenie projektu vo VS Code**“, alebo **„inštalácia MSYS2**“. Ak chcete spustiť priamo vygenerovaný spustiteľný súbor mimo prostredia VS Code, prejdite na sekciu „**Spustenie vygenerovaného spustiteľného súboru (.exe)**“.

### Inštalácia MSYS2

1. **Stiahnite si inštalačný program MSYS2 z oficiálnej stránky:** <https://www.msys2.org/>.
   * Spustite stiahnutý inštalačný program a postupujte podľa pokynov na obrazovke. Odporúča sa ponechať predvolené nastavenia inštalácie.
2. **Aktualizácia systému MSYS2 a inštalácia potrebných balíkov**
   * Po úspešnej inštalácii MSYS2 otvorte **MSYS2 MinGW 64-bit** terminál a zadajte príkazy
     + pacman -Syu
     + pacman -S mingw-w64-ucrt-x86\_64-gcc
     + pacman -S mingw-w64-x86\_64-gdb
3. **Pridanie cesty k binárnym súborom MSYS2 do premennej prostredia PATH (voliteľné)**
   * Aby Windows našiel potrebné nástroje (ako g++) bez zadávania celej cesty, pridajte nasledujúcu cestu do systémovej premennej prostredia PATH.
   * Predpokladá sa, že MSYS2 je nainštalované v predvolenom umiestnení **C:\msys64\ucrt64\bin**
4. **Overenie prítomnosti g++ (konkrétny kompilátor pre C++) v PowerShell**
   * g++ --version

### Spustenie projektu vo VS code

1. V prieskumníku súborov Visual Studio Code prejdite na súbor **main.cpp**.
2. V pravom hornom rohu okna editora kliknite na ikonu **spustenia (Play)**
3.  Z zobrazenej ponuky vyberte možnosť **Run C/C++ File**
4. V ponuke pre výber konfigurácie spustenia (vedľa ikony Play) vyberte možnosť **Run C++ Program preLaunchTask: Build with g++**. Táto konfigurácia najprv zostaví projekt pomocou g++ (definované v tasks.json) a následne spustí vygenerovaný spustiteľný súbor.

### Použité rozšírenia

Počas práce boli použité nasledujúce rozšírenia C/C++:

Obrázok, na ktorom je text, snímka obrazovky, písmo

Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.

### Spustenie vygenerovaného spustiteľného súboru (.exe)

Po úspešnom zostavení projektu sa v priečinku projektu vygeneruje spustiteľný súbor s názvom **SuffixTree.exe**. Pre jednoduchšiu manipuláciu je tento spustiteľný súbor prítomný už automaticky. Ak nie je nutné skúšať funkčnosť kódu, toto je najjednoduchší spôsob spustenia projektu.

Na spustenie tohto súboru mimo prostredia Visual Studio Code postupujte nasledovne:

1. **Otvorte Prieskumníka súborov (Windows Explorer).**
2. **Prejdite do priečinka, kde sa nachádza vygenerovaný súbor main.exe.**
3. **Dvojkliknite na súbor SuffixTree.exe.**

Spustiteľný súbor sa spustí v novom okne príkazového riadka (termináli) a vykoná naprogramovanú činnosť.

# Základy a definície

## Kľúčové pojmy pre pochopenie sufixových stromov

Na zadefinovanie sufixového stromu je potrebné zadefinovať niektoré termíny, s ktorými sa neskôr stretneme v tejto dokumentácií, alebo slúžia na lepšie pochopenie konceptu.

### Prefix

Prefix je ľubovoľná **začiatočná časť reťazca**, začínajúca **od prvého znaku** a postupne sa rozširuje o ďalšie znaky.

**Príklad :** Ak máme slovo „**abeceda“** jeho prefixy sú :

* „**a**“ , „**ab**“ , „**abe**“, „**abec**“ , „**abece**“, „**abeced**“, „**abeceda**“

### Sufix

Sufix je **ľubovoľná koncová** časť reťazca**, začínajúca od niektorého znaku** a siaha až do **konca reťazca.**

**Príklad :** Ak máme slovo „**abeceda“** jeho sufixy sú :

* „**abeceda“, „beceda“, „eceda“ , „ceda“ , „eda“ , „da“, „a“**

### Trie (Prefixový strom)

Trie je **stromová dátová štruktúra**, ktorá slúži na **ukladanie veľkého množstva reťazcov** tak, aby sa **zdieľali spoločné začiatky (prefixy)**. Každý uzol v trie predstavuje prefix reťazca.

Obrázok, na ktorom je snímka obrazovky, diagram, text

Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.V trie vyhľadávame pomocou „kľúča“, čo je reťazec ktorý práve hľadáme.

**Štruktúra trie:**

* **Jeden koreňový uzol –** ktorý nepredstavuje žiadnu časť kľúča a slúži iba ako začiatok
* **Uzly –** každý uzol v trie predstavuje jeden znak kľúča (hľadaného slova)
* **Vetvy –** každá vetva ktorá ide z daného uzla, predstavuje možný nasledujúci znak
* **Koncové uzly** – uzly ktoré predstavujú koniec uloženého kľúča, často špeciálne označené (aby sa odlíšili od uzlov ktoré môžu byť len časti dlhších slov).

Na obrázku vyššie je vidieť prefixový strom s vloženými slovami „**abeceda**“, „**abcd**“, „**abec**“. (zdroj: <https://www.cs.usfca.edu/~galles/visualization/Trie.html> )

### Kompresia trie

V základnej implementácii **trie (prefixového stromu)** každá vetva reprezentuje jeden znak a uzly sa vetvia vždy, keď sa cesty pre rôzne **kľúče** rozchádzajú.

V sufixovom strome sa často **jednotlivé hrany** bez vetvenia komprimujú tak, že **hrana** môže reprezentovať **celý podreťazec** namiesto jedného znaku.

Kompresia spočíva v tom, že ak máme v strome časť, kde sa cesty nevetvia (teda z uzla vychádza len jedna hrana a do neho prichádza tiež len jedna, okrem koreňa), môžeme tieto po sebe idúce znaky na hrane zlúčiť do jedného podreťazca. Namiesto viacerých hrán s jedným znakom tak vznikne jedna hrana reprezentujúca celý úsek reťazca.

Krátky popis komprimovaného prefixového strom s vloženými slovami „**abeceda**“ a „**abcd**“. :

1. Obrázok, na ktorom je kruh, diagram, dizajn

   Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.**Vloženie slova "abeceda"**

* Vytvorí sa koreňový uzol z ktorého vychádza vetva označená reťazcom "**abeceda**" do listového uzla, ktorý reprezentuje slovo "**abeceda**".

1. **Vloženie slova "abcd"**

* Začíname v koreňovom uzle, kde prechádzame existujúce vetvy, hľadajúc zhodný prefix s vkladaným slovom "**abcd**".
* Obrázok, na ktorom je kruh, diagram, snímka obrazovky, dizajn

  Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.Nájdeme hranu "**abeceda**", ktorá má spoločný prefix "**ab**".
* **Rozdelíme hranu "abeceda"**: V mieste, kde sa zhodný prefix končí ("ab"), vytvoríme nový vnútorný uzol.
  + - Hrana z koreňa do nového vnútorného uzla bude označená "**ab**".
    - Hrana z nového vnútorného uzla k pôvodnému listovému uzlu bude označená zvyškom pôvodného slova, teda "**eceda**".
* Z nového vnútorného uzla vytvoríme novú hranu pre zvyšok vkladaného slova "**abcd**", čo je "**cd**", a táto hrana povedie k novému listovému uzlu reprezentujúcemu "abcd".

Tento postup by bol rovnaký aj pri pridaní viacerých reťazcov.

### Sufixový strom

Sufixový strom je **stromová dátová štruktúra**, ktorá slúži na reprezentáciu všetkých možných **sufixov (koncov)** daného reťazca. Hoci sa často uvádza ako **príklad komprimovanej trie (prefixového stromu)**, treba zdôrazniť, že základná definícia sufixového stromu sama o sebe kompresiu nevyžaduje.

Kompresia je spomenutá práve preto, že v nasledujúcej časti bude vysvetlený **Ukkonenov algoritmus**, čo je efektívny postup na vytvorenie priamo komprimovaného sufixového stromu.

Suffixový strom má obsahovať **všetky sufixy daného reťazca** – ale aby sa strom jednoznačne ukončoval a aby sme nemali problém s tým, že jeden sufix je podreťazcom iného, pridáme na koniec **špeciálny znak, ktorý sa v texte nevyskytuje**. (napríklad $ )

## Zložitosti jednotlivých trie

Táto sekcia sa zameriava na komplexnosť dvoch typov dátových štruktúr Trie (prefixový a sufixový strom). Vysvetlíme si, prečo dosahujú uvádzané zložitosti, aby sme po implementácii mohli jasne overiť ich správnu funkčnosť a očakávanú výkonnosť.

**Zaujímať nás bude hlavne:**

* **Časová zložitosť** - Koľko operácií je potrebných na zostavenie alebo prehľadanie stromu (v najhoršom prípade)
* **Priestorová zložitosť** - Koľko pamäte zaberie celá dátová štruktúra
* **Konštrukcia stromu** - **proces vytvorenia dátovej štruktúry** (napr. trie, suffix tree) z daných vstupov (napr. reťazec alebo zoznam slov), tak aby bol strom pripravený na vyhľadávanie, analýzu alebo iné operácie.

**Pre lepšie pochopenie budeme používať nasledujúce notácie:**

* **N:** Počet slov (kľúčov) vložených do prefixového stromu.
* **L**: Maximálna dĺžka jedného slova (kľúča) spomedzi všetkých vložených slov.
* **n**: Dĺžka hlavného reťazca, pre ktorý sa vytvára sufixový strom.
* **m**: Dĺžka hľadaného podreťazca (vzoru), ktorý sa vyhľadáva v strome.
* **k**: Používa sa špecificky pri vyhľadávaní v sufixovom strome a predstavuje počet výskytov hľadaného podreťazca v pôvodnom reťazci.

Nasledujúca tabuľka sumarizuje **časovú a priestorovú zložitosť** konštrukcie a vyhľadávania pre **nekomprimovanú** a **komprimovanú** verziu **prefixového** a **sufixového stromu**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Nekomprimovaný** | | **Komprimovaný** | |
| **Vlastnosti** | **Prefixový trie** | **Sufixový trie** | **Prefixový trie** | **Sufixový trie (Ukkonenov)** |
| **Konštrukcia** | V najhoršom prípade **O(N⋅L)**, pretože každý znak každého slova môže vytvoriť nový uzol. | Naivná konštrukcia vložením všetkých sufixov môže trvať až **O(n2).** Pre každý z **n** sufixov, sa každý z nich vkladá do trie. | **O(N⋅L)** v najhoršom prípade, ale v praxi býva efektívnejšia, pretože sa zlučujú cesty bez vetvenia. | Existujú algoritmy s lineárnou časovou zložitosťou **O(n),** ako napríklad Ukkonenov algoritmus. |
| **Priestorová zložitosť** | V najhoršom prípade **O(N⋅L),** pretože môže existovať toľko uzlov a hrán, koľko je celkový počet znakov vo všetkých vložených slovách. | V najhoršom prípade **O(n2).** Každý sufix vyžaduje vetvu s dĺžkou úmernou jeho dĺžke, čo vedie k kvadratickému počtu uzlov. | O(N), pretože počet uzlov je úmerný počtu vložených slov (každé slovo končí v listovom uzle alebo na hrane). Hrany sú označené podreťazcami, ale celkový počet hrán je obmedzený počtom uzlov. | **O(n).** Kompresiou sa eliminuje mnoho nepotrebných uzlov, a počet uzlov a hrán je lineárny vzhľadom na dĺžku reťazca. Každý listový uzol reprezentuje jeden sufix a vnútorné uzly reprezentujú spoločné prefixy sufixov. |
| **Vyhľadávanie (slova dĺžky (M) / podreťazca dĺžky (m))** | **O(M),** pretože prechádzame stromom po jednom znaku hľadaného slova. | **O(m)**, pretože prechádzame stromom po znakoch hľadaného podreťazca. | **O(M)**, pretože prechádzame stromom, pričom každá hrana predstavuje potenciálne viacero znakov. Počet porovnaní znakov je stále úmerný dĺžke hľadaného slova. | **O(m+k)**, kde **k** je počet výskytov podreťazca. **O(m)** je čas na nájdenie prvého výskytu prechodom stromom a **O(k)** je čas na prechod podstromom, ktorý obsahuje všetky výskyty. |

## Ukkonenov algoritmus

Ukkonenov algoritmus je efektívny algoritmus na **konštrukciu komprimovaného suffixového stromu** pre daný reťazec v **lineárnom čase** – teda v čase **O(n)**, kde **n** je dĺžka reťazca. Tento algoritmus je významný hlavne pre aplikácie v oblasti **bioinformatiky**, **kompresie dát**, **hľadania vzorov v texte** a podobne.

### Základné princípy s ktorými Ukkonenov algoritmus pracuje

Táto sekcia stručne a jednoducho vysvetľuje základné pojmy používané v Ukkonenovom algoritme na konštrukciu sufixového stromu.

#### active\_node (Aktívny uzol)

* Aktuálne miesto v strome, odkiaľ hľadáme, kam pridať ďalší kúsok sufixu. Predstavte si ho ako "pracovný bod".
* Strom na začiatku neobsahuje žiadne sufixy, preto začíname s prázdnou štruktúrou, z ktorej budú vychádzať všetky vetvy. -**root**
* Počas práce sa môže posunúť na iné uzly, napríklad keď prejdeme celú existujúcu vetvu alebo použijeme špeciálny odkaz (suffix link).

#### suffix link (Sufixový odkaz)

* Špeciálna "skratka" z jedného vnútorného uzla na iný. Ak máme uzol pre reťazec "abc", jeho suffix link ukazuje na uzol pre "bc".
* Umožňuje nám rýchlejšie nájsť miesto pre pokračovanie kratšieho sufixu bez toho, aby sme museli znova prechádzať strom od začiatku. Zrýchľuje celý proces.

#### active\_edge (Aktívna hrana)

* Označuje, ktorou vetvou (hranou) z **active\_node** sa momentálne zaoberáme. Nepamätáme si priamo písmeno, ale jeho pozíciu v pôvodnom texte.
* Pomáha nám vybrať správnu vetvu z **active\_node**, ktorá začína hľadaným znakom.
* **Inicializuje sa na -1.** *Odôvodnenie:* Hodnota -1 indikuje, že sa momentálne nenachádzame uprostred žiadnej hrany; sme priamo v **active\_node**.

#### active\_length (Aktívna dĺžka)

* Koľko znakov z aktuálnej **active\_edge** sme už "prešli".
* **Význam:**
  + Ak je 0, sme priamo v uzle.
  + Ak je väčšia ako 0, sme uprostred nejakej vetvy.
* Pomáha určiť, či môžeme pokračovať po vetve, či ju musíme rozdeliť, alebo či sme dosiahli koniec vetvy.

#### remainder (alebo remaining\_suffix\_count) (Zvyšok)

* Počet sufixov, ktoré ešte musíme pridať pre aktuálny spracovávaný znak.
* **Ako sa mení?** Zvyšuje sa o 1 s každým novým znakom v texte a znižuje sa o 1 vždy, keď úspešne pridáme sufix do stromu.
* **Inicializuje sa na 0** pretože pred spracovaním akéhokoľvek znaku nemáme žiadne nové sufixy na vloženie.

#### last\_created\_internal\_node (Posledný vytvorený vnútorný uzol)

* Dočasná "pamäť" na posledný vytvorený vnútorný uzol.
* Slúži na to, aby sme mu mohli neskôr nastaviť jeho **suffix link.**
* Inicializuje sa na **None** pretože na začiatku nemáme žiadne vytvorené interné uzly

#### leaf\_end (Koniec listu)

* Index posledného pridaného znaku v texte. Koncový index všetkých listových hrán je implicitne [začiatok, ∞), pričom ∞ sa aktualizuje s každým novým znakom prostredníctvom leaf\_end.
* **Inicializuje sa na -1** pretože predspracovaním prvého znaku neexistuje žiadny pridaný znak, preto je počiatočný index -1.

#### current\_phase alebo i (Pozícia)

* Aktuálna pozícia (index) znaku, ktorý práve spracovávame v pôvodnom texte.
* Pomáha určiť, ktorý znak pridávame a kde sa končia nové sufixy.

#### S[i]

* i-ty znak vstupného reťazca S.

#### text (Text)

* Pôvodný reťazec, z ktorého vytvárame sufixový strom. Na konci zvyčajne obsahuje špeciálny znak (napr. $), aby sa zabezpečila jedinečnosť všetkých sufixov.

Tieto koncepty spolupracujú, aby efektívne a rýchlo vytvorili sufixový strom pre daný text. Algoritmus postupne prechádza textom a v každom kroku aktualizuje strom pomocou týchto premenných a pravidiel.

### Situácie ktoré môžu nastať počas konštrukcie sufixového stromu

#### Vytváranie listových vetiev (tzv. Rule 1)

Ide o najjednoduchší prípad, keď sa spracúva nový znak S[i], ktorý **nie je pokračovaním žiadnej existujúcej vetvy** zo súčasného uzla (alebo pozície). Znamená to, že daný sufix ešte v strome neexistuje. V tomto prípade sa vytvára **nový listový uzol**, ktorý bude reprezentovať nový sufix končiaci na konci reťazca.

**Mechanizmus:**

* Z aktuálneho uzla sa vytvorí nová hrana začínajúca znakom S[i]
* Táto hrana bude smerovať k novému listovému uzlu
* V liste sa uloží index začiatku tohto sufixu

#### Rozdeľovanie hrán a vytváranie vnútorných uzlov (tzv. Rule 2)

Tento prípad nastáva, keď časť sufixu (napr. prefix) **už existuje** v strome ako súčasť nejakej hrany, ale nový znak S[i] sa **nezhoduje** s pokračovaním tejto hrany. V takom prípade sa existujúca hrana **rozdelí** – vznikne nový **vnútorný uzol**, ktorý je spojený s pôvodnou vetvou aj s novou vetvou.

**Mechanizmus:**

* Hrana, na ktorej vznikol konflikt, sa rozdelí na dve časti
* V bode rozdelenia sa vytvorí nový vnútorný uzol
* Pôvodná vetva sa pripojí na tento nový vnútorný uzol
* Z vnútorného uzla sa vytvorí nová listová vetva začínajúca znakom S[i]

#### Show-stopper situácia (tzv. Rule 3)

Tento prípad nastane vtedy, keď je aktuálny znak S[i] **už súčasťou pokračovania existujúcej hrany** a zodpovedá ďalšiemu znaku na tejto hrane. Znamená to, že daný sufix sa **už nachádza v strome**, a preto nie je potrebné ho znova vkladať. Výstavba sa v tejto vetve preruší – odtiaľ názov "show-stopper".

**Mechanizmus:**

* Nevykonáva sa žiadne rozdelenie hrany
* Nepridáva sa žiaden nový uzol
* Ukončí sa spracovanie aktuálneho rozšírenia (extension)

#### Využitie suffixových odkazov (suffix links)

Suffixové odkazy (suffix links) sú pomocné ukazovatele medzi vnútornými uzlami, ktoré umožňujú efektívne preskakovanie medzi sufixmi pri výstavbe stromu. Ak sa počas algoritmu vytvorí vnútorný uzol reprezentujúci reťazec xA, suffix link z neho bude smerovať na uzol reprezentujúci A (teda o jeden znak posunutý sufix).

**Mechanizmus:**

* Pri vytvorení vnútorného uzla u sa z neho nastaví suffix link na uzol v, ktorý reprezentuje sufix suffix(u)
* Pri ďalšom spracúvaní sa aktívna pozícia môže rýchlo presunúť cez tento odkaz bez nutnosti prehľadávať celý strom

#### Aktívne posuny (active\_node, active\_edge, active\_length)

Ukkonenov algoritmus si počas výstavby uchováva tzv. „aktívnu trojicu“, ktorá reprezentuje pozíciu v strome, odkiaľ bude pokračovať spracovanie ďalšieho znaku:

* **active\_node**: uzol, v ktorom sa nachádzame
* **active\_edge**: index znaku, od ktorého sa začína hrana (z reťazca)
* **active\_length**: počet znakov už prečítaných na hrane

Táto trojica sa priebežne aktualizuje podľa toho, ako sa algoritmus vyvíja.

#### Prekrytie sufixov (napr. opakujúce sa vzory)

Pri spracovaní reťazcov, ktoré obsahujú opakujúce sa prefixy alebo vzory (napr. „ababab“, „aaaa“), môže dôjsť k tomu, že niekoľko sufixov začína rovnakým alebo podobným spôsobom. Tieto sufixy sa zdieľajú v strome cez rovnaké vnútorné uzly.

**Mechanizmus:**

* Pri spracúvaní každého sufixu sa zistí, koľko znakov už existuje v strome
* Zvyšok sa pridáva ako nové vetvy
* Spoločné prefixy (napr. „ab“) sa uchovajú len raz, zdieľajú sa medzi viacerými sufixmi

### Vysvetlenie Ukkonenovho algoritmu a príklad

Text ktorý použijeme ako príklad pre vysvetlenie Ukkonenovho algortitmu je "abab$". Pridaný znak $ slúži ako unikátny koncový znak. Začíname s prázdnym stromom, ktorý obsahuje len koreňový uzol.

#### Obrázok, na ktorom je text, snímka obrazovky, softvér, počítačová ikona Obsah vygenerovaný umelou inteligenciou môže byť nesprávny. Inicializácia

* **text** = "abab$"
* **root** = nový uzol
* **active\_node** = root
* **active\_edge** = -1
* **active\_length** = 0
* **remainder** = 0
* **leaf\_end** = -1
* **last\_created\_internal\_node** = None
* **current\_phase** = 0

#### Iterácia 1 (i = 0, znak 'a')

* Obrázok, na ktorom je text, snímka obrazovky, displej, číslo

  Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.**leaf\_end** sa zvýši na 0 (*prvý znak bol spracovaný*).
* **remainder** sa zvýši na 1 (*máme jeden nový sufix "a", ktorý treba vložiť*).
* **active\_length** je 0, preto **active\_edge** sa nastaví na i = 0 (text[0] = 'a').
* Koreň nemá hranu 'a' (*strom je zatiaľ prázdny okrem koreňa*).
* **Použije sa Pravidlo 1:** Vytvorí sa nová hrana 'a' z koreňa do listu [0, 0] (*sufix "a" ešte nie je reprezentovaný*).
* **remainder** sa zníži na 0 (*sufix "a" bol úspešne vložený*).
* **last\_created\_internal\_node** zostáva None.

Obrázok, na ktorom je text, snímka obrazovky, diagram, rad

Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.

#### Iterácia 2 (i = 1, znak 'b')

* leaf\_end sa zvýši na 1.
* **remainder** sa zvýši na 1 (*nový sufix "b" treba vložiť*).
* **active\_length** je 0, preto **active\_edge** sa nastaví na i = 1 (text[1] = 'b').
* Koreň nemá hranu 'b'.
* **Použije sa Pravidlo 1:** Vytvorí sa nová hrana 'b' z koreňa do listu [1, ∞) (*sufix "b" ešte nie je reprezentovaný*).
* **remainder** sa zníži na 0.

#### Iterácia 3 (i = 2, znak 'a')

* Obrázok, na ktorom je text, snímka obrazovky, diagram, rad

  Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.**leaf\_end** sa zvýši na 2.
* **remainder** sa zvýši na 1 (*nový sufix "a" treba vložiť*).
* **active\_length** je 0, preto **active\_edge** sa nastaví na i = 2 (text[2] = 'a').
* Koreň má hranu 'a'.
* **Použije sa Pravidlo 3 (Show-stopper):** Aktuálny znak text[i] = 'a' sa zhoduje so začiatkom existujúcej hrany 'a' (*sufix "a" je už implicitne prítomný ako prefix*).
* Zvýšime **active\_length** na 1 (*implicitne hľadáme "aa"*). **remainder** zostáva 1.

#### Obrázok, na ktorom je text, snímka obrazovky, diagram, rad Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.Iterácia 4 (i = 3, znak 'b')

* **leaf\_end** sa zvýši na 3.
* **remainder** sa zvýši na 2 (*predošlý remainder 1 + nový sufix "b"*).
* Aktívne premenné: **active\_node** = root, **active\_edge** = 0 ('a'),
* **active\_length** = 2 (*hľadáme pokračovanie sufixu začínajúceho na 'a'*).
* Sledujeme hranu 'a' z koreňa. Prešli sme dĺžku

Obrázok, na ktorom je text, snímka obrazovky, diagram, rad

Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.Obrázok, na ktorom je snímka obrazovky, text, diagram, rad

Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.

Obrázok, na ktorom je text, snímka obrazovky, diagram, rad

Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.

Zvyšné kroky sú ukázané len v rýchlosti, pretože je to zdĺhavé na opisovanie. Na vizualizáciu bolo použitý <https://brenden.github.io/ukkonen-animation/>

# Implementácia sufixového stromu s použitím Ukkonenovho algoritmu

Sufixový strom je trie (strom), ktorý umožňuje efektívne reprezentovať všetky sufiksy reťazca. V tejto implementácii používame **Ukkonenov algoritmus** na dynamickú stavbu sufixového stromu v reálnom čase. Tento algoritmus umožňuje konštruovať sufixový strom v **O(n)** čase (kde n je dĺžka reťazca) pomocou postupného pridávania znakov reťazca. Je efektívny, pretože minimalizuje počet potrebných operácií pri každodennom spracovaní textu.

## Popis programu a súborov

Program pozostáva z niekoľkých hlavných súborov, ktoré sú zodpovedné za rôzne funkcie a operácie súvisiace s implementáciou sufixového stromu:

1. **SuffixTree.h**:
   * Tento súbor definuje triedu SuffixTree, ktorá predstavuje sufixový strom. Zahrnuje všetky potrebné dátové štruktúry ako uzly stromu a funkcie na stavbu a vyhľadávanie v strome.
   * Obsahuje aj pomocné štruktúry ako Node, ktorá reprezentuje jednotlivé uzly sufixového stromu a obsahuje detaily ako začiatočné a koncové indexy, deti uzla a sufixový odkaz.
2. **SuffixTree.cpp**:
   * Implementuje metódy definované v SuffixTree.h. Tieto metódy sa používajú na vytváranie a manipuláciu so sufixovým stromom. Zahrnuté sú metódy ako build(), extendSuffixTree(), find(), clear(), preorder() a print().
   * extendSuffixTree() je kľúčová metóda, ktorá implementuje základnú logiku Ukkonenovho algoritmu a pridáva nové sufixy do stromu pri každom kroku.
3. **main.cpp**:
   * Hlavný súbor, ktorý inicializuje sufixový strom a testuje jeho funkcie. V tomto súbore sa nachádza aj generovanie náhodných reťazcov pre experimenty a meranie výkonu pri rôznych veľkostiach abeced a dĺžkach reťazcov.
   * Tento súbor tiež vykonáva vyhľadávanie v sufixovom strome a vypisuje výsledky.
   * Na konci súboru je uloženie výsledkov experimentov do CSV súboru pre neskoršiu analýzu.
4. **experiment\_data.csv**:
   * Tento súbor slúži na uloženie experimentálnych dát o výkonnosti. Ukladá rôzne kombinácie veľkosti abecedy a dĺžok reťazcov a čas, ktorý bol potrebný na vybudovanie sufixového stromu pre dané parametre.
   * Tento CSV súbor je vhodný na vizualizáciu a analýzu výkonu algoritmu pomocou nástrojov ako R alebo Excel.

# Experimentálna analýza metódy build

Experimentálnu analýza časovej zložitosti budovania sufixového stromu v závislosti od dĺžky vstupného reťazca. Dáta sú uložené v priloženom súbore s názvom **„experiment\_data.csv“** a načítané pomocou <https://www.chartgo.com/>

Nasledujúce tri grafy ilustrujú závislosť **času budovania sufixového stromu** (v milisekundách, na zvislej osi) od **dĺžky vstupného reťazca** (na vodorovnej osi) pre rôzne **veľkosti abecedy**: 4, 26 a 62 znakov.

Vo všetkých prípadoch pozorujeme, že s rastúcou dĺžkou reťazca sa **vo všeobecnosti zvyšuje aj čas budovania**, čo je v súlade s očakávanou teoretickou zložitosťou Ukkonenovho algoritmu.

## Analýza dát pre veľkosť abecedy 4

Tento graf vykazuje **najhladší priebeh**, ktorý **najviac pripomína lineárny nárast**, hoci pri najdlhších reťazcoch je možné pozorovať mierne zakrivenie nahor.

| **Dĺžka reťazca** | **Čas budovania (ms)** | **Pomer (čas/dĺžka)** |
| --- | --- | --- |
| 100 | 0 | 0 |
| 500 | 2 | 0.004 |
| 1000 | 6 | 0.006 |
| 2000 | 9 | 0.0045 |
| 4000 | 13 | 0.00325 |
| 8000 | 33 | 0.004125 |

Pomery sa pohybujú v relatívne úzkom rozmedzí (0.003 – 0.006), čo naznačuje **takmer lineárny rast**. Nulová hodnota pre reťazec dĺžky 100 môže byť spôsobená veľmi krátkym časom, ktorý sa nedal presne zmerať (napr. zaokrúhlením).

Obrázok, na ktorom je rad, diagram, vývoj, svah

Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.

## Analýza dát pre veľkosť abecedy 26

Tento graf je **výrazne nerovnomernejší** – vykazuje tzv. „zubatý“ priebeh, s **nízkymi hodnotami pri niektorých stredných dĺžkach** reťazcov.

| **Dĺžka reťazca** | **Čas budovania (ms)** | **Pomer (čas/dĺžka)** |
| --- | --- | --- |
| 100 | 8 | 0.08 |
| 500 | 1 | 0.002 |
| 1000 | 2 | 0.002 |
| 2000 | 12 | 0.006 |
| 4000 | 7 | 0.00175 |
| 8000 | 17 | 0.002125 |

Pomery sa **výrazne líšia**, pričom niektoré hodnoty sú podozrivo nízke. Tento efekt môže byť spôsobený napríklad:

* **cache efektom** (rýchle spracovanie údajov pri určitej veľkosti vstupu),
* alebo **overheadom systému**, kde časový údaj môže byť ovplyvnený inými procesmi.

Aj napriek týmto odchýlkam sa pri vyšších dĺžkach (napr. 8000) opäť objavuje trend, ktorý by mohol zodpovedať **približne lineárnemu rastu**.

Obrázok, na ktorom je rad, diagram, vývoj

Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.

## Analýza dát pre veľkosť abecedy 62

Podobne ako pri abecede s 26 znakmi, aj tento graf je **nepravidelný**, s veľmi nízkymi hodnotami pre stredné dĺžky.

| **Dĺžka reťazca** | **Čas budovania (ms)** | **Pomer (čas/dĺžka)** |
| --- | --- | --- |
| 100 | 2 | 0.02 |
| 500 | 1 | 0.002 |
| 1000 | 2 | 0.002 |
| 2000 | 5 | 0.0025 |
| 4000 | 7 | 0.00175 |
| 8000 | 18 | 0.00225 |

Nízke pomery pre stredné dĺžky reťazcov sa opäť objavujú, čo môže byť výsledkom:

* optimalizácií v rámci implementácie,
* efektívneho využitia pamäťovej hierarchie,
* alebo **náhodných odchýlok pri meraní**.

Napriek týmto výkyvom pozorujeme, že čas budovania opäť **rastie s dĺžkou vstupu**, aj keď nie dokonale lineárne.

Obrázok, na ktorom je rad, diagram, vývoj, text

Obsah vygenerovaný umelou inteligenciou môže byť nesprávny.

## Vyhodnotenie analýzy

Ukkonenov algoritmus má teoretickú zložitosť **O(n)** pri **konštantnej veľkosti abecedy**, kde *n* je dĺžka vstupného reťazca. To znamená, že čas budovania by mal rásť **lineárne** s veľkosťou vstupu.

**Pozorované odchýlky**

V experimentálnych dátach však dochádza k určitým **nelinearitám**, ktoré môžu byť spôsobené:

* **Implementačnými detailmi** – optimalizácie alebo nevýhody konkrétnej implementácie,
* **Vplyvom cache pamäte** – prístup k dátam môže byť výrazne rýchlejší pri určitých veľkostiach,
* **Overheadom systému** – meranie prebieha v prostredí operačného systému, ktorý môže byť ovplyvnený ďalšími procesmi,
* **Konštantnými faktormi** skrytými v notácii O(n) – ktoré môžu byť odlišné pri rôznych veľkostiach abecedy.

**Vplyv veľkosti abecedy**

Nie je jednoznačné, že väčšia abeceda znamená **výrazne dlhší čas budovania**. Výsledky ukazujú, že aj pri veľkej abecede (62 znakov) môže byť pomer čas/dĺžka relatívne nízky. To naznačuje, že veľkosť abecedy síce ovplyvňuje výpočtový výkon, ale **jej vplyv je komplexný** a nemusí byť dominantným faktorom.

# Záver

Na základe analyzovaných dát možno konštatovať:

* **Teoretická lineárna zložitosť** algoritmu sa vo väčšine prípadov **potvrdzuje**, najmä pri abecede veľkosti 4.
* **Odchýlky** v prípade väčších abecied poukazujú na dôležitosť **implementačných detailov** a prostredia, v ktorom sa merania vykonávajú.
* Výsledky podporujú tvrdenie, že Ukkonenov algoritmus je prakticky efektívny aj pre väčšie vstupy, a to **napriek rôznym systémovým a pamäťovým vplyvom**, ktoré sa môžu prejaviť v experimentoch.