# Dokumentácia

## Implementácia (generalizovaného) suffixového stromu ako container

### 1. Stručný popis

Program implementuje datovú štruktúru generalizovaného suffixového stromu *Suffix_Tree* ako container, ktorú môže programátor používať bez znalosti interného fungovania stromu s tým, že si nad ňou môže vytvárať vlastné algoritmy. Rovnako štruktúra implementuje sadu základných (najčastejšie využívaných) metód a to *insert()*, *printAllSufixes(), isSubstring()* a *LCS(),* teda longest common substring.

### 2. Presný popis

*Suffix_Tree* je dátová štruktúra pre jazyk C++ ktorá implementuje koncept generalizovaného suffixového stromu ako container, teda udržuje si myšlienku prispôsobiteľnosti podľa uživateľa (programátora) tak, ako to on potrebuje. Samotná implementácia je rozdelená iba do 2 častí, no program má aj nejaké základné príkazy v *main()* funkcii pre názornosť použitia. Prvá časť *Suffix_Tree.h* je hlavičkový súbor ktorý obsahuje objektový návrh samotného stromu a prislúchajúce triedy k jej plnej funkčnosti. Druhá časť, a to *Suffix_Tree_Implementation.cpp* je implementačná časť kódu pre hlavičkový súbor. V hlavičkovom súbore môžeme pozorovať viacero tried, metód a šablón; rozeberieme ich teda pekne po poradí.

Ako prvá je šablóna Pole<T> na ktorej vpodstate nie je nič (programátorsky) zaujímavé, iba to, že je to kvázi vector<T> implementovaný tak, aby sa neinvalidovali iterátory a pointre ukazujúce doňho pri prípadnej realokácii pamäte. Na jeho potrebu sa budeme odkazovať ďalej v kóde.

Ďalšia je šablóna union_sets<T> ktorá slúži, už ako názov vypovedá, na zjednotenie dvoch množín. Jej potrebu rovnako uvedieme neskôr v kóde.

Nasledujúca je trieda Node. Keďže implementujeme suffixový strom, tak mi prišlo vhodné si spraviť na vrcholy osobitnú triedu, pretože to bude prehľadnejšie a pri stavbe stromu sa to takto aj viac vyplatí. Nadefinované máme 2 základné constructory, kde jeden je prázdny constructor a druhý obsahuje argumenty. Ten s argumentami využívame vtedy, keď vytvárame nový vrchol, keďže tam presúvam, okrem iného, informáciu o rodičovi. Použité sú 3 hashovacie tabuľky; potomok, kde mám uložených všetkých potomkov a ako kľúč mi slúži prvé písmeno hrany vedúce do potomka (pretože, keďže je to suffixový strom, tak písmená hrán vedúce z vrcholu musia byť unikátne) a intervalyHranV / intervalyHranL. Tieto hashovacie tabuľky mi slúžia ako náhrada za hrany, pretože nevediem žiadnu špeciálnu triedu na hrany. Ak je potomok list, tak hranu do neho vediem v tabuľke intervalyHranL a ak je to vnútorný vrchol tak v tabuľke intervalyHranV. V tabuľke intervalyHranL je druhá súradnica pointer; dôvod, prečo je to tak, bude popísaný v sekcii **Algoritmus**. Kvôli šetreniu pamäti sú hrany reprezentované ako intervaly, nie ako stringy. V skratke, ide o to, že algoritmus funguje v asymptotickom čase O(n) a preto tam potrebujem

pointer. variableString a variableInt sú voľné premenné, do ktorých si môže užívateľ containeru zapisovať čokoľvek, po vôli, ak by chcel so stromom robiť nejaké vlastné algoritmy alebo programy. Posledná časť triedy Node sú *„gety()"*, privátne premenné a friend trieda. Táto časť je spravená tak ako je napísané v komentári a to, že nechcem, aby uživateľ mohol meniť ID, containAllStrings alebo SuffixLink. Snažíme sa o čo "najblbuvzdornejšiu" implementáciu ☺

Posledná a samozrejme najdôležitejšia časť celého programu je trieda SuffixTree.

SuffixTree obsahuje 3 konštruktory; `SuffixTree()` skonštruuje prázdnu štruktúru, `SuffixTree(const std::string& seedString)` skonštruuje štruktúru s jedným vloženým stringom ako parameter hneď a `SuffixTree(const SuffixTree&)` je copy constructor, teda, môžem robiť verné kópie už nejakého existujúceho stromu.

Základnú funkčnosť sufixového stromu dotvára sada základných implementovaných metód, menovite:
`void insert(const std::string& Slovo)`- vloženie stringu do stromu
`void printAllSufixes(bool Sorted = false)`- vypísanie všetkých suffixov, ktoré sa nachádzajú vo vytvorenom strome (možnosť použiť argument; ak sa nastaví na true, tak ich vypíše v zoradenom poradí podľa poradia, ako boli vkladané do stromu)
`const std::string& get_concString()`- vráti nám celý skonkatenovaný string aký je uložený v strome. Táto metóda slúži na to, ak si uživateľ robí vlastné algoritmy nad štruktúrou, pretože kvoli šetreniu pamäti sú stringy na hranách reprezentované iba ako interval (treba dávať **pozor**, použité **intervaly** na hranách **sú uzavreté**, nie polouzavreté**!!**) a teda ak chce vedieť s čím pracuje, tak čísla na hranách sú reprezentácia tohto stringu.
`bool isSubstring(const std::string& substr)`- touto metódou vieme zistiť, či string zadaný ako argument je substringom nejakého zo stringov vložených do stromu.
`std::string LCS()`- longest common substring. Táto metóda vráti najdlhší spoločný podreťazec zo všetkých vložených v strome.

Okrem týchto metód je verejný ešte koreň, pretože chceme, aby uživateľ mohol so stromom pracovať aj sám a nebol viazaný iba na tieto metódy.

Ďalej nasledujú privátne prvky resp. metódy triedy SuffixTree. Väčšina z nich je buď jasná z názvu, je okomentovaná v kóde alebo jej význam je dôležitý pre rýchlosť algoritmu a teda ju bližšie spomeniem v sekcii **Algoritmus**, preto tu spomeniem iba tie dôležité resp. tie ktoré potrebujú nejaký bližší komentár (nie je na prvý pohľad vidieť, na čo slúžia).
Zaujímavý je `struct size_compare.` Je to funktor, ktorý nám slúži na porovnávanie v usporiadanej množine do ktorej budeme vkládať suffixy, ak bude argument pre *printAllSufixes()* true. Vtedy chcem usporiadať prioritne podľa ukončovacieho ID suffixu a v prípade rovnosti ID (teda je to suffix toho istého stringu) podľa dĺžky (lebo to skrátka takto pekne vyzerá ☺)
`Pole<unsigned int> ENDy` je rovnako zaujímavá štruktúra. Ako sme si už vyššie spomínali, slúži na vyriešenie problému s invalidáciou pointrov ukazujúcich na vektor.

Bližšie priblíženie bude v sekcii **Algoritmus** ale v skratke ide o to, že každý list bude obsahovať pointer práve na jeden index toho poľa a ja vlastne ak viem, ktorý list má kde ukazovať tak dokážem v konštantnom čase zvýšiť interval všetkým listom ktoré ukazujú na istý index i iba tým, že zvýšim ENDy[i] o 1.
`std::unordered_map<Node*, Node*> stary_vs_novy_vrchol` je zaujímavá štruktúra a zároveň sa mi páči to, ako som to vymyslel (aj keď to nemusí byť optimálne). Pri copy-constructore ak chcem pridávať suffixLinky tak je problém, že ja neviem dopredu povedať ktorý vrchol ukazuje do ktorého vrcholu, keďže ja práve iba budujem novú vernú kópiu starého stromu. Preto si spravim hashovaciu tabuľku, kde ako kľúč mi bude slúžiť adresa pôvodného vrcholu v "starom" strome a ako hodnota bude k nemu priradená verná kópia 1:1 v novom strome. Potom na druhý prechod jednoducho všetkým vrcholom popridávam suffixLinky, pretože budem vedieť, ktorý nový vrchol reprezentuje ktorý v starom strome.
Nasledujúce metódy sú všetko metódy ktoré sa viac či menej využívajú priamo v algoritme na zostrojenie stromu. Od metódy *nextChar()* nasledujú všetko pomocné metódy, ktoré sa využívaju v metódach stromu, nie pri jeho stavaní.

## 3. Algoritmus + implementácia

Tak ako vždy, všetko stojí a zároveň padá na tom, aký je použitý algoritmus. Gro celého programu je postavenie suffixového stromu v asymptoticky lineárnom čase. To som sa počas programovania snažil docieliť a nakoniec sa to aj podarilo s použitím Ukkonenovho algoritmu na postavenie sufixových stromov, ktorý som si zobecnil a použil ho na postavenie generalizovaného sufixového stromu. V prvom rade chcem poďakovať pánom z internetu, kde som sa mohol inšpirovať samotným algoritmom, ktorý bol zrozumiteľne popísaný a dalo sa pochopiť, ako to spraviť.

https://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english

https://www.geeksforgeeks.org/ukkonens-suffix-tree-construction-part-1/

Celé fungovanie algoritmu je naozaj krásne popísané v prvom vloženom linku a určite by som to ani lepšie nenapísal. Avšak, aby bolo jasné, čo dané metódy v mojej implementácii robia, tak budeme spoločne budovať algoritmus ktorý si popíšeme aspoň letmo (všetky detaily sú naozaj pekne a zrozumiteľne dohľadateľné minimálne v tom prvom linku) a vysvetlíme si, ako by sme dané veci implementovali a aj ako sú implementované v tejto mojej implementácii.

- **Voľba algoritmu**

Najjednoduchšia možnosť je spraviť naivný algoritmus na budovanie suffixových stromov, a to tak, že si string budem postupne spracovávať po všetkých suffixoch. Najprv najväčší, ktorý tam vložím. Potom o prvé písmeno kratší, ktorý tam vložím... takto pokračujeme dalej a postupne vkladáme všetky suffixy. V prípade, že sa nám niekde vyskytne prefix nejakého suffixu, tak tam, kde je potrebné rozbiť hranu tak ju rozbijeme, a na to potrebujeme konštantne mnoho práce takže to je irelevantné.

Avšak ako vidíme, tento spôsob má až O(n^2) časovú zložitosť, s čím sa určite neuspokojíme.

- **Každý suffix začína na iné písmeno**

Tento prípad je veľmi jednoduchý. Napríklad string „abcd“, keď ho vybudujeme tak budeme mať 4 rôzne suffixy z koreňa a žiadny vetviaci sa vrchol. Pre takéto prípady, kedy potrebujeme vytvoriť novú hranu (takú ktorá neexistuje) som si vytvoril metódu `void pridajNovuHranu(Node * vrchol, char c, char type)`. Táto metóda dostane ako argumenty pointer na vrchol z ktorého chcem vyvoriť novú hranu (pre novovytváraný vrchol je to teda rodič), znak c ktorý mi reprezentuje písmeno hrany pod ktorým bude uložená v tabuľke (pretože je unikátna ako sme si povedali) a type je typ vrcholu, teda či vytváram nový vnútorný vrchol alebo list. Na tieto požiadavky je vyrobený aj konštruktor s parametrami pre triedu Node.

Z konštrukcie algoritmu vieme, že ak je nejaký vrchol listom, tak bude už stále listom. A aby sme algoritmus mali lineárny, tak sa snažíme na každom načítanom písmene stráviť nanajvýš konštantný čas. Vidíme, že ku všetkým listom môžem naraz pridať nový načítaný znak, pretože o tento znak sa mi musia logicky všetky suffixy predĺžiť (patriace danému stringu samozrejme, avšak zatiaľ riešime iba suffixový strom pre 1 string. Jeho generalizácia bude popísaná neskôr). A v tomto prípade nám presne príde vhod štruktúra Pole a zároveň si vysvetlíme, prečo pri intervaloch hrán do listov máme druhú súradnicu pointer. Finta je v tom, že ja si pre každý string vediem v poli (na danom indexe ktorý je rovnaký ako poradie stringu v akom bol vložený do stromu) číselnú hodnotu, na akej pozícii v `std::string concString` končí tento string. Teraz by som rád konštantne pridal všetkým listom novonačítaný znak na koniec ich intervalu a to spravím lišiacky tak, že druhá súradnica teda pointer mi bude ukazovať na správny prvok v štruktúre `Pole<unsigned int> ENDy` a tam mi stačí iba zvýšiť tento prvok o 1 a tak sa konštantne zvýši hodnota intervalu všetkým listom ktoré tam ukazujú, teda konštantne mnoho práce. A preto sme museli použiť Pole a nie iba obyčajný vektor, pretože dopredu nevieme, koľko stringov bude v strome, a teda by sa mohlo stať (a určite sa aj stane po nejakom veľkom množstve vložených stringov do stromu) že by nemal priestor v pamäti a musel by sa realokovať niekde inde a tým pádom by sa invalidovali všetky pointre ktoré tam ukazujú (z listov). Použitím však štruktúry Pole sme sa toho šikovne zbavili.

- **Opakovania**

Ako je to presne, je vo vloženom linku. Pri tomto jave nám na scénu prichádza štruktúra `Active_Point` ktorá funguje presne tak, ako chceme aby fungovala z popisu v článku. Obsahuje 3 položky a to aktívny vrchol, hranu a dĺžku (lepšie by však bolo používať slovo hĺbku, nakoľko sa vlastne pýtam, ako hlboko po danej hrane som už prešiel). Priamo späté s týmto postupom, kedy prechádzam po nejakom suffixe, ktorý má rovnaký prefix sú metódy `void updateAc_point(unsigned int i_Spracuvavaneho)` a `bool mozemPridat(unsigned int i_Spracuvavaneho)`. Druhá spomínaná metóda funguje nasledovne: Predstavme si, že sme vo vrchole a prechádzam po nejakej hrane (postupne načítavam znaky) a chcem zistiť, či je

nasledujúci znak na hrane ten, ktorý práve spracuvávam. A na to táto metóda slúži. Ak sa vyhodnotí ako true, tak sa mi zvýši length (našiel som zhodu, posúvam sa vpred o 1), ak nie, musí sa zavolať metóda *rozbiHranu()* alebo *pridajNovuHranu(),* podľa toho, či som „na hrane“ alebo „vo vrchole“. No a prvá spomínaná metóda *updateAc_point()* dopĺňa druhú. Ak sa niekde pohnem v strome, či už načítam ďalšie písmeno alebo rozbijem hranu, potrebujem si Active_Point štruktúru nastaviť na správne data. Jednoduchý príklad. Som na hrane do vnútorného vrcholu, ukazujem ZA 2. písmeno a hrana nech má 3 písmena. Ďalšie načítané písmeno stringu je rovnaké ako 3. písmeno. *mozemPridat()* to teda vyhodnotí ako true, a length sa mi zvýši o 1. To by ale znamenalo, že ukazujem za 3. písmeno, čo je ale hlúposť, pretože hrana má iba 3 písmená. Preto sa zavolá *updateAc_point()* a posunie ma to do tohto vnútorného vrcholu.

No a potom samozrejme, keď som niekde zanorený, tak iba zavolám `void rekurzivneVyriesSuffixy(unsigned int i_Spracuvavaneho)` ktorá vpodstate robí iba to, čo ma v názve. Rekurzívne vyrieši ostatné suffixy, ktoré mám naskladané v remainderi ☺.

- **Generalizácia sufixového stromu**

Zo začiatku bol problém s tým, že som nevedel, ako šikovne zgeneralizovať suffixový strom, aby vedel príjmať (zhora) neobmedzený počet sufixov. Potom som prišiel s nápadom zarážky, lenže tam bol problém, že klávesnica nemá neobmedzený počet znakov. A potom nakoniec som to spravil tak, že za každou zarážkou mám ID zapísané číselne, čo mi reprezentuje unikátnosť stringov. Preto je v stringoch vkladaných do sufixového stromu **zakázané** používať znak **‘$’** pretože pri použití tohto znaku v niektorom zo stringov nie je zaručený chod programu a ani správny výsledok. Rovnako ale som sa chcel vyhnúť zbytočnému vypisovaniu niečoho, čo ma vôbec nezaujíma a teda na štýl : “abcdef$02asfdaf$03afsgds$04…” pretože mňa zaujíma iba časť po prvú zarážku. To je ten suffix na ktorom záleží. A kvôli tomu, že sa mi to takto osobne viac páčilo a príde mi to aj krajšie a prehľadnejšie, tak som spravil práve to `Pole<unsigned int> ENDy`. To pole vlastne funguje pomerne jednoducho. Pre každý práve spracuvávaný string viem jeho poradie (teda jeho ID za zarážkou). Všetky jeho listy, resp. všetky listy v ktorých končia suffixy pre daný string ukazujú na položku ENDy s indexom jeho ID resp. poradia. Ak daný string spracujem až do konca, tak všetky jeho listy ukazujú stále rovnako. A po dokončení spracovania sa mi ID zvýši, rovnako sa mi zvýši index v poli ENDy a ďalší spracovávaný string bude mať osobitnú položku v Poli ENDy. Teda jeho listy budu ukazovať už iba na to, na čo majú a listy predošlých nebudú obsahovať nič naviac, pretože vždy updatujem iba jeden správny prvok v poli ENDy.

## 4. Alternatívne programové riešenie

Pôvodne som chcel spraviť riešenie pre zvrchu obmedzený počet stringov ktoré môžem vložiť do stromu a mal by som pre nich unikátne C ciferné číslo nasledujúce za nimi (teda by som nestratil možnosť použiť $ v stringoch) ale nakoniec som zhodnotil, že je to hlúposť a takto postráda akýkoľvek význam sufixový strom, keďže dopredu

zvyčajne človek nevie, koľko čoho chce použiť a hlavne by to nebolo univerzálne. Preto som si povedal, že toľko oželiem to že sa $ nemôže vyskytovať v stringoch ale container je plne prispôsobiteľny a univerzálny.

## 5. Vstupné data

Keďže je to implementácia sufixového stromu, tak je logické, že vstupné data sú stringy. Jediná funkcia ktorá príjma vstup je vpodstate metóda *SuffixTree::insert().* Uživateľ má viacmenej voľnú ruku, aké stringy vkladať do metódy, no sú tu 4 podmienky ktoré to musí spĺňať.
V prvom rade funkcia insert akceptuje parameter typu string. Preto je potrebné, aby parameter, ktorý vkladá uživateľ do metódy bol typu std::string.
Druhá podmienka, ktorá bola v dokumentácii spomínaná už aspoň dvakrát je, že vkladaný string **NESMIE OBSAHOVAŤ** znak **$**. Ak string tento zakázaný znak predsalen obsahovať bude, autor implementácie si vyhradzuje akékoľvek právo na vzniknuté škody, nakoľko uživateľ bol varovaný.
Tretia podmienka je, že daný string **NESMIE OBSAHOVAŤ** ani znak **#.** Toto v texte explicitne spomenuté nebolo, no keďže tu pojednávame o vstupných datach, tak je to spomenuté tu. Pre uživateľa je to poznámka, ktorú ma akceptovať a tým sa riadiť. Pre programátora zvedavca je dôvod napísaný v najdlhšom komentári v metóde `void SuffixTree::rozbiHranu(Node * vrchol, char c, unsigned int index_c)`.
A posledná podmienka (pravdepodobne ktorá nebude vo väčšine prípadov problém, ale pre úplnosť si ju tu spomenieme) je, že počet vložených stringov do stromu je predsalen zhora obmedzený, a to tým, že ako ID sa používa datový typ unsigned int. V číslach je to presne `4294967296`. Samozrejme, preprogramovať to na väčší datový typ problém nie je, a ak by bol o implementáciu záujem vo veľkých korporáciách, tak sa to dá spraviť ☺

## 6. Výstupné data

Výstupné data sú v takom formáte, akého návratového typu je metóda, ktorú uživateľ použije. V prípade *isSubstring()* je to 0 alebo 1 podľa toho, či daný string je podreťazcom niektorého zo stringov v strome. *LCS()* vypíše reťazec, ktorý je najdlhší spoločný spomedzi všetkých stringov vložených do stromu (v prípade prázdneho reťazcu naozaj vypíše prázdny reťazec). No a „akoby najinteraktívnejšia“ metóda *printAllSufixes()* vypíše zoznam všetkých sufixov všetkých reťazcov nachádzajúcich sa v strome spoločne s ich zarážkou a unikátnym ID za zarážkou (aby bolo poznať, ktorému stringu daný sufix patrí), každý na nový riadok. V prípade, že bude parameter true, tak ich vypíše v zoradenom poradí (čo ale bude mať za následok spomalenie programu, pretože sa musí výstup najprv usporiadať).

## 7. Záver

Všetko, čo som si zaumienil som vlastne aj dokončil. Strom má presne takú štruktúru, akú som chcel aby mal od začiatku, kedy som na tomto zápočťáku začal pracovať. Obsahuje také metódy, aké som chcel aby obsahoval. Vypisuje všetko tak, ako chcem aby vypisoval. Rovnako, program funguje v lineárnom čase, takže je to bomba.

Jedna vec, ktorá ma však veľmi mrzí je, že napriek tomu, že je program asymptoticky v lineárnej časovej zložitosti, tak je pri povedzme na počítač relatívne malom množstve dát stále pomalý. Pri spracovávaní cca. 100 000 znakov program beží zhruba 20sek. Nie je to síce hrozné a zo svojho výsledku som ajtak nadšený, keďže je to moje prvé relatívne veľké programátorské dielo a niečo ako také vôbec v jazyku C++, no ajtak by to bolo krajšie a lepšie by sa na to pozeralo, keby to fachalo v rýchlejšom čase. Snažil som sa dávať si pozor na rôzne zbytočné kópie, snažil sa používať pointre a referencie všade tam kde to len ide, aby sa to zrýchlilo (hoci nie asymptoticky ale aspoň o tú konštantu) a ešte si aj prekontroloval kód, či by tam nešlo niečo zlepšiť (nie fundamentálne zmeny, ale skôr detaily... zbytočné kopírovanie apod.) no čo sa toho týka, tak keď som to zbežne preletel, tak som tam nič nevidel.
Preto pre mňa tento zápočtový program určite nie je skončená kapitola a rád by som sa k nemu v lete vrátil, kedy už bude po skúškach, kedy človek bude mať znovu chvíľku čas sám na seba ☺.
Rád ťa teda znovu uvidím na nejakej prípadnej konzultácii (hoci po skúškach alebo v ďalšom semestri) ohľadom toho, ako by sa to dalo zlepšiť a aby to bolo v takej podobe, že by som si to eventuélne mohol vyvesiť na nejaký GitHub či inú podobnú službu. ☺

Celkovo, tento zápočťák hodnotím pozitívne, nakoľko som sa pri ňom mnoho naučil (okrem iného aj prácu s Visual Studiom ☺).

Vyhotovil: Matúš Maďar