SPELL CHECKER

Dokumentácia k softwaru

Michal Kovaľ

1. ročník, skupina I33

Programování I NPRG030

Február 13, 2016

Obsah

Kapitola 1....2
Úvod....2
1.1 Úloha programu....2
1.2 Spustenie programu....3
1.3 Užívateľské prostredie....3
1.3.1 Submenu – 1. Skontrolovat text....4
1.3.2 Submenu – 2. Pridať slovo do slovníka....6
1.3.3 Submenu – 3. Nájsť slovo v slovníku....6
1.3.4 Submenu – 4. Ukončiť....7
Kapitola 2....9
Použité algoritmy a dátové štruktúry....9
2.1 Algoritmus – porovnanie dvoch slov....9
2.2 Algoritmus – porovnanie množiny korektných slov so slovom chybným....10
2.2.1 Dátová štruktúra – Trie (prefixový strom)....10
2.2.2 Procedúra pre vkladanie slov do Trie....11
2.2.3 Levenshteinova vzdialenosť & Trie....11
2.2.4 Ostatné procedúry a funkcie....14
2.2.4 Úprava kódu, rozšírenie o nové funkcie....15
Rereferencie:....16

Kapitola 1

Úvod

Táto dokumentácia obsahuje základné informácie o programe **Spell Checker** ako aj bližšie stotožnenie sa s behom programu, užívateľským prostredím, použitými algoritmami a dátovými štruktúrami.

Veľakrát sa nám stane, že pri písaní dokumentácie, ako je napríklad táto, sa v texte nájdu chybné slová. Niekedy je príliš obtiažne prechádzat celý text znovu a hľadať, kde sme asi spravili nejakú tú chybičku. Čo ak by ale existoval nástroj, ktorý by všetkú tú prácu urobil za nás. A práve s týmto problémom sa popasuje program, ktorému je venovaná táto dokumentácia.

1.1 Úloha programu

Úlohou Spell Checker-a je vyhľadať v texte chybné slová a ponúknuť za ne užívateľovi adekvátnu náhradu v čo najkratšom možnom čase (rádovo v niekoľkých desiatkach až stovkách milisekúnd). K tomu aby program fungoval správne potrebujeme mať jednak šikovný algoritmus a korektnú databázu slov – slovník, podľa ktorého to porovnáme. Program za chybné slovo považuje každé, ktoré sa od korektného slova líši nasledujúcim spôsobom:

- chýbajúcim písmenom (najviac jedným, napr. *marvelous – mrvelous*),
- písmenom navyše (najviac jedným, napr. *despite – despite**a***),
- písmenom nahradeným za iné (najviac jedným, napr. *welcome – we**k**come*),
- prehodením dvoch písmen stojacich vedľa seba (práve dvoch, napr. *acceptable - accept**ba**le*).

Program dáva užívateľovi možnosť rozširovať slovník o nové slová ako aj možnosť v danom slovníku vyhľadávať. Prehliadač slovníka vám odpovie, či sa hľadané slovo v slovníku nachádza, popríde sa zobrazia alternatívy.

1.2 Spustenie programu

V adresári programu sa nachádza zdrojový kód ***levens.pas*** . V závisloti na použitej platforme uživateľa je potrebné vytvoriť spustiteľný binárny súbor. Pre kompiláciu zdrojového kódu je možné použiť:

- *(MS-DOS) Turbo Pascal v7.1 + DOSBox v0.74* **(Windows)**
- *Free Pascal IDE, {FPC 2.6.4, GDB 7.4} (Windows)*
- *Lazarus 1.4.4 (Linux)*
- *Free Pascal IDE, {FPC 2.6.4, GDB 7.4} (Linux)*

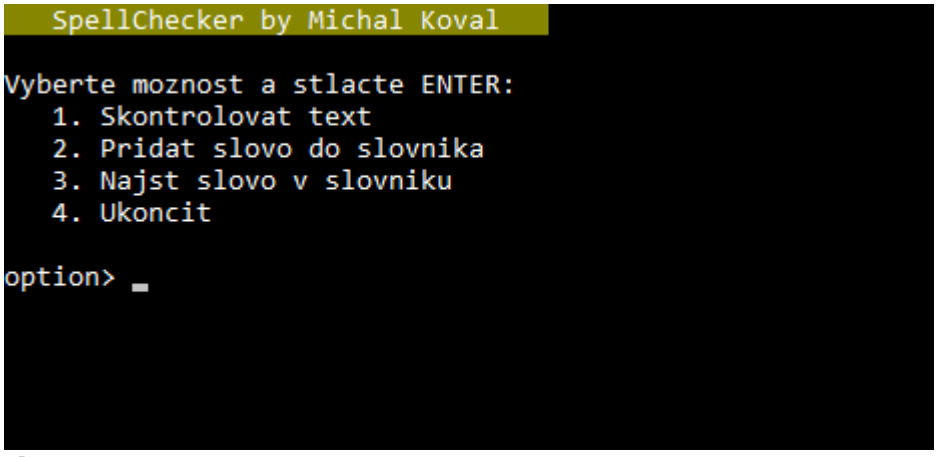
-knižnice použité v zdrojovom kóde: *Crt, DateUtils, SysUtils.*

Pred spustením programu je **potrebné** mať súbor **dic_edit.dat** v adresári programu alebo v aktuálnom pracovnom adresári použitého vývojového prostredia. Súbor **dict_edit.dat** obsahuje 351077 slov z anglickej slovnej zásoby [1] (slová obsahujú iba znaky „a“ .. „z“).

Spustite vami vytvorený binárný súbor.

1.3 Užívateľské prostredie

Po spustení programu sa zobrazí menu s možnosťou výberu (obr. 1.3.a). Pre výber zvoľte čislo a stačte ENTER. Vstup je ošetrený voči nežiadúcim znakom, užívateľa na to program uporozní.



obr. 1.3.a

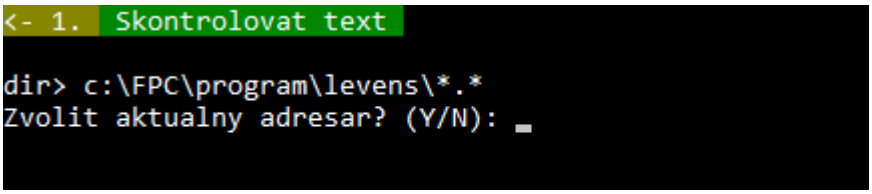
Pri prvom zobrazení menu sa vypíše koľko slov bolo načítaných a za aký dlhý čas (v ms) zo súboru **dic_edit.dat**

1.3.1 Submenu – 1. Skontrolovat text

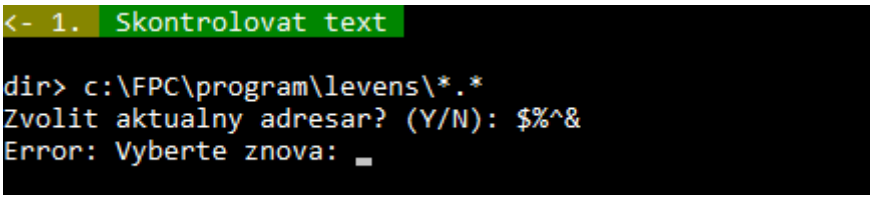
Program sa spýta, z akého adresára ma vziat zdrojový text k oprave.

Voľba: Y – yes, ponecháva aktuálny adresár,

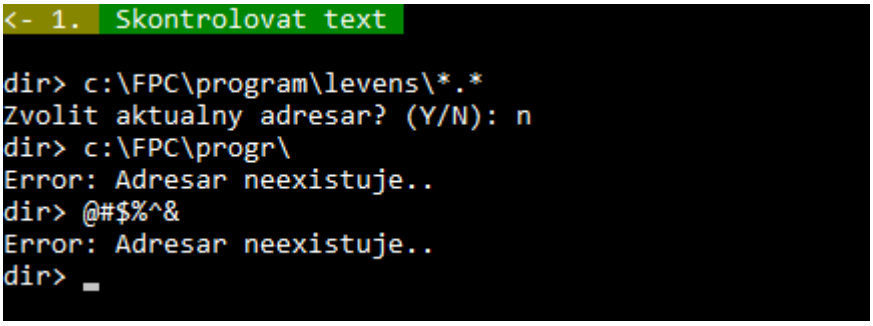
N – no, dovoľuje užívateľovi zvoliť vlastný adresár.



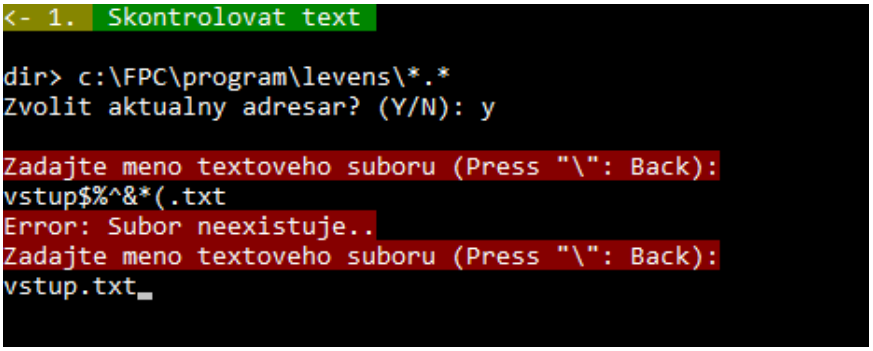
Program je ošetrený voči neželaným vstupom, či už v prípade voľby (Y/N) (obr 1.3.1.a) alebo v prípade neexistujúceho či nesprávne zvoleného adresára (obr 1.3.1.b). Následne je užívateľ požiadaný o zadanie mena súboru s textom pre opravu. Neexistujúce alebo nesprávne meno súboru je znovu ošetrené upozornením (obr. 1.3.1.c) a následným vyzvaním uživateľa k zmene. Je možné sa vrátiť do hlavného menu: napíšte „\" a stačte ENTER.



obr. 1.3.1.a



obr. 1.3.1.b



obr. 1.3.1.c

Vstupom do programu môže byť ľubovoľný text z angličtiny. Slová musia byť oddelené medzerami (môžu byť súčasťou zátvoriek, úvodzoviek, apostrofov) a nesmú obsahovať znaky rôzne od { „a“ .. „z“, „A „ .. „Z“ }. Testovacie vstupy sú k dispozícií v adresári programu.

Po správne zvolenom súbore stlačte ENTER a program vyhodnotí daný text. Výstupom bude zoznam najdených chybných slov a taktiež výstup do súboru **output.txt**. V niektorých prípadoch sa nájdu v slovníku odpovedajúce alternatívy. Niektore slová môžu byť vyhodnotené ako „neznáme“. Tie bohužial nie sú obsiahnuté v slovníku alebo neboli najdené ich alternatívy, kedže program je obmedzený na rozdiel slov v jednom znaku ako bolo spomenuté na začiatku **Kapitoly 1**. Rozdiel je možné samozrejme upraviť v kóde programu (viď **Kapitola 2**).

```
<- 1.  Skontrolovat text

dir> c:\FPC\program\levens\*.*
Zvolit aktualny adresar? (Y/N): y

Zadajte meno textoveho suboru (Press "\": Back):
vstup.txt

Chyby v texte:
--------------
[Microsoft] -> { nezname slovo }
[inser] -> {infer, inker, inner, insea, insee, inseer, insep, insert, inset, instr, inter, rinser}
[diferent] -> {deferent, different, digerent, liferent}
[versons] -> {persons, versions, versos}

vystup je v subore.. output.txt
...Naspat: Press ENTER...
```

Text môže byť aj bez chýb. V takom prípade program vypíše, že nenašiel žiadnu chybu (obr 1.3.1.d). Samozrejme aj tento výsledok zavisí na bohatosti slovníka.

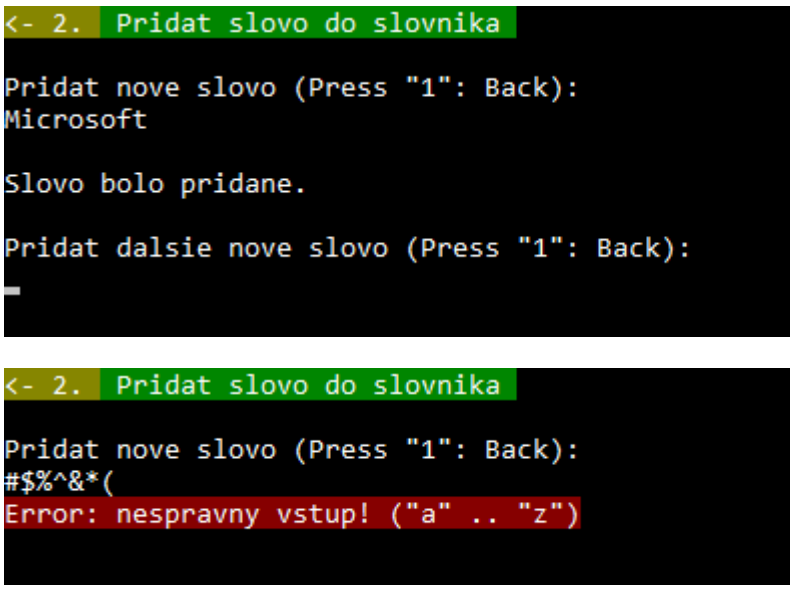
```
<- 1.  Skontrolovat text

dir> c:\FPC\program\levens\*.*
Zvolit aktualny adresar? (Y/N): y

Zadajte meno textoveho suboru (Press "\": Back):
cicero.txt

Chyby v texte:
--------------

V texte sa nenasli chyby.
...Naspat: Press ENTER...
```

obr. 1.3.1.d

Problém s nedostatkom slovnej zásoby môžeme vyriešiť dalšou featurou programu „Pridať slovo do slovníka“.

1.3.2 Submenu – 2. Pridať slovo do slovníka

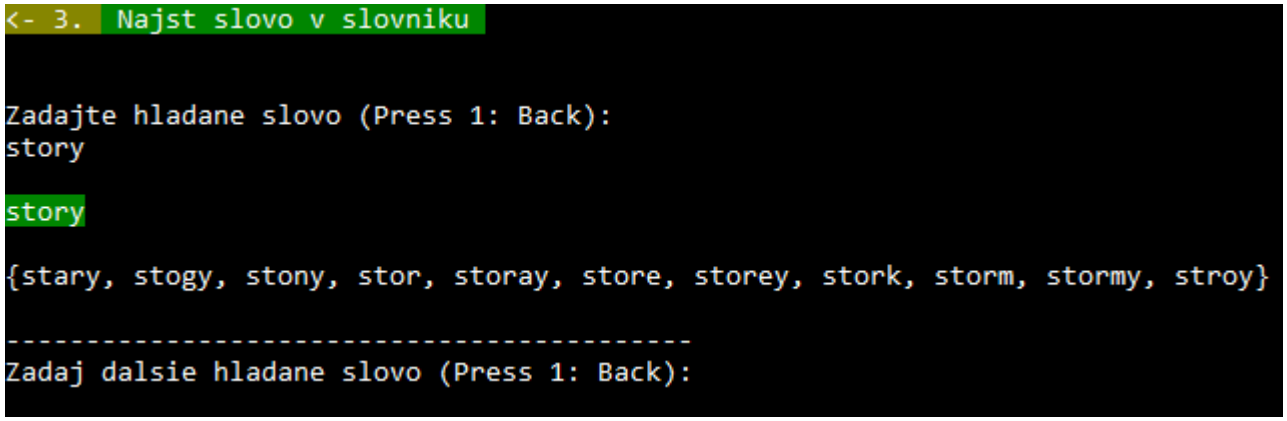
Novo prídané slovo je k dispozícií počas behu programu ako aj pri jeho ďalšom zapnutí. Slovo je korektné ak obsahuje kombináciu znakov {„a“ .. „z“, „A“ .. „Z“}. Pri nesprávnom vstupe vás program upozorní (obr 1.3.2.a). Slová sú v slovníku ukladané v **lowercase** tvare (z dôvodu implementácie, samozrejme aj to je možné upraviť ale na úkor pamäte). Možnosť pridávať aj dalšie slová. Odoberanie slov v programe nie je naimplementované. Pre návrat do hlavného menu napíšte „1“ a stlačte ENTER.



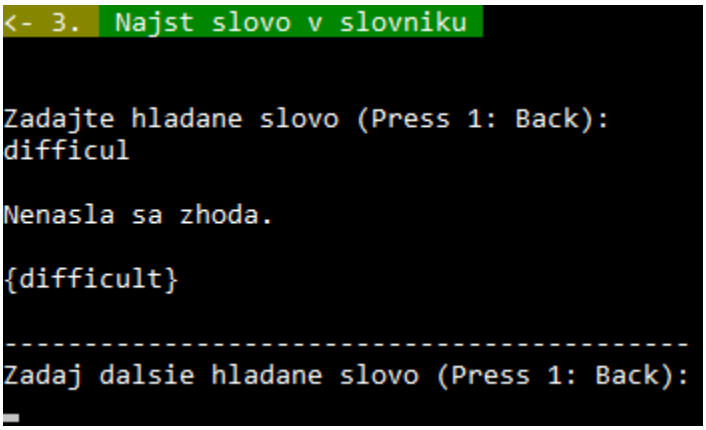
obr. 1.3.2.a

1.3.3 Submenu – 3. Nájsť slovo v slovníku

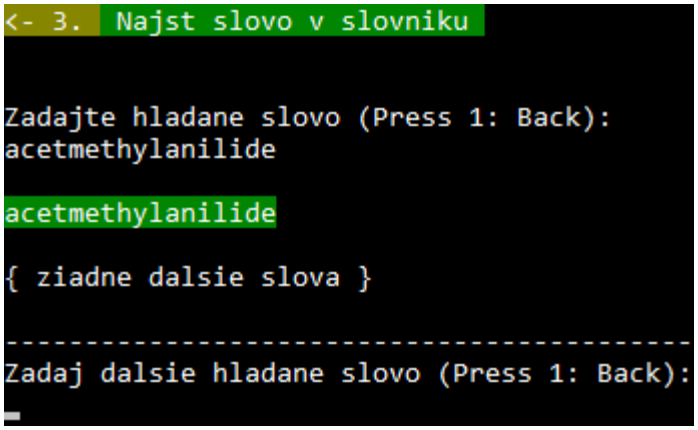
V slovníku je možnosť aj vyhľadávať a overiť si, či sa dané slovo do slovníka pridalo. Užívateľ to samozrejme určite využije aj ako pomocníka pri overovaní si správneho tvaru slova alebo hľadaní alternatív k slovu.



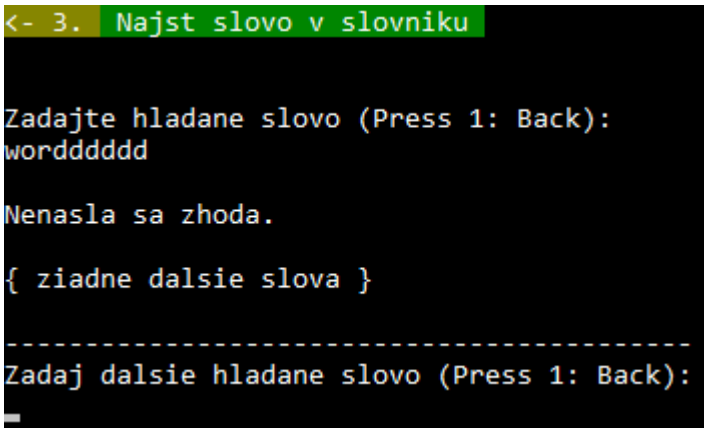
V prípade, že program nájde zhodné slovo, vypíše sa so zeleným podfarbením. V zložených zátvorkách sa vypíšu všetky podobné slova k hľadanému slovu. Ak sa nenájde zhoda, vypíše sa „Nenašla sa zhoda“ (obr 1.3.3.a). Ak sa nenájdu podobné slová vypíše sa „{ žiadne ďalšie slová }“ (obr 1.3.3.b). Zároveň môžu nastať oba prípady súčasne (obr 1.3.3.c).



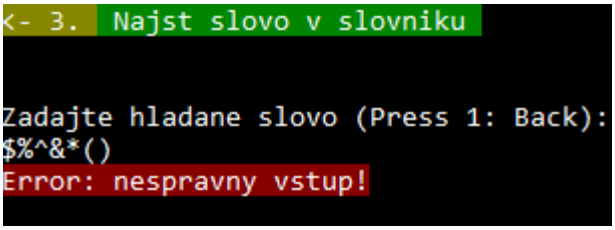
obr. 1.3.3.a



obr. 1.3.3.b



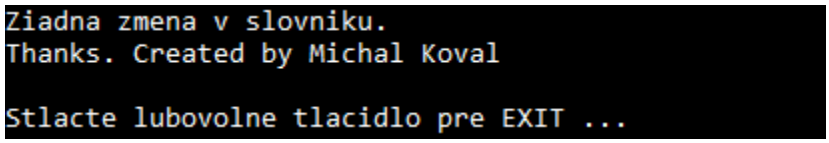
obr. 1.3.3.c

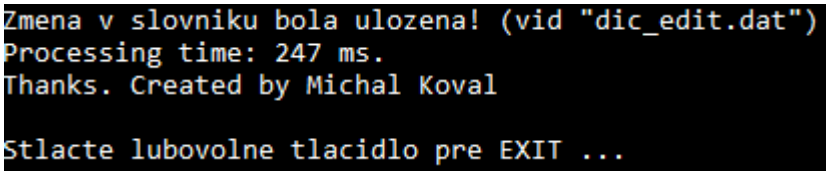


Pri nesprávnom vstupe vás program upozorní. Zadajte slovo znovu alebo pre návrat do hlavného menu napíšte „1“ a stlačte ENTER.

1.3.4 Submenu – 4. Ukončiť

Obsahuje dodatočné informácie a ukončuje program. Ak došlo k zmene v slovníku, zmena sa uloží pre budúce použitie. Vypíše sa, či došlo k zmene a ako dlho trvalo uloženie slov.





Pre EXIT stlačte ľubovoľnú klávesu.

Nasledujúca kapitola je zameraná na algoritmickú časť programu. Diskusiu o tom, prečo boli použité dané algoritmy, dátove štruktúry, možnosti úpravy kódu ako aj jeho následne rozšírenie o nové funkcie.

Kapitola 2

Použité algoritmy a dátové štruktúry

Pri tvorbe programu som sa musel vysporiadať s viacerými prekážkami. Prvou bolo to, ako efektívne porovnať množinu korektných slov (reťazcov) so slovom chybným a nájsť k nemu to správne poprípade podobné slovo. Ďalším problémom bolo vymyslieť dátovú štruktúru pre uloženie veľkého množstva slov a vedieť v tejto štruktúre vyhľadávať čo najrýchlejšie.

2.1 Algoritmus – porovnanie dvoch slov

Každé slovo je reprezentované ako reťazec znakov (tzv. **String**). Typ **String** je v Pascali obmedzený na maximálne 255 znakov. Slová v slovníku v priemere nepresiahnu viac ako 20 znakov. Čo sa týka chybného slova, to sa môže dĺžkovo výrazne líšiť od slov v slovníku. Ako bolo spomenuté v **Kapitole 1** slová sa od seba líšia maximálne jedným znakom (**chýbajúcim znakom, znakom navyše, rozdielnym znakom**) alebo **prehodením dvoch znakov stojacich vedľa seba**. Pre tieto štyri prípady som použil rozšírený algoritmus vychádzajúci z algoritmu Levenshteinova vzdialenosť – konkrétne Damerau–Levenshteinova vzdialenosť (viac informácií nájdete na [2]).

Algoritmus porovnáva dve slová po jednotlivých znakoch a v závislosti na tom o koľko znakov (alebo tiež: Koľko operácií je potrebných k tomu aby sme chybné slovo zmenili na korektné) sa líšia, je vrátená hodnota v podobe vzdialenosti. Aby som algoritmus zrýchlil, pridal som podmienku, že ak pri postupnom porovávani znakov vznikne vzdialenosť presahujúca nastavený limit (vzdialenosť max. 1), porovnávanie sa ukončí.

Príklad 2.1: 1. slovo: “**abeceda**”, 2. slovo: “**abemnopqrst**”. Postupným porovnávaním zistíme, že vzdialenosť je nulová pre prvé tri znaky „**abe**“. V ďalšom kroku porovnávame znak „**c**“ a „**m**“. Tu nastáva jeden zo štyroch prípadov – konkrétne ide o rozdielne znaky. Vzdialenosť sa navýši o jedna, ale to je ešte v rámci limitu akceptovateľné. Ak by obe slová končili pri 4. znaku, mohli by sme povedať, že sú si navzájom podobné. Avšak, ak budeme pokračovať ďalej zistíme, že výsledná vzdialenosť bude **8**.

V programe je tento algoritmus reprezentovaný funkciou:

```
function LevenshteinDistance(a: String; length_a: Byte; b: String;
length_b: Byte; MaxCost: Byte; var IsItWorthIt: Boolean): Byte;
```

a – reprezentuje reťazec s chybným slovom; **b** – reprezentuje reťazec s korektným slovom; **length_a** a **length_b** sú dĺžky slov (určujú akú dlhú časť zo slov je potrebné porovnať, vysvetlenie neskôr); **MaxCost** – maximálna povolená vzdialenosť; **IsItWorthIt** – návratová booleovská premenná nám napovie, či je vhodné porovnávať aj naďalej (vysvetlenie neskôr). Funkcia

vracia hodnotu typu Byte, teda hodnotu vzdialenosti (postačuje max. 255, reálne nie je možné prekročit vzdialenosť dlhšiu ako 255 aj v prípade, kedy by boli slová dlžky 255 a boli by navzájom úplne odlišné) **.** Tento algoritmus je obmedzený iba na porovnanie dvoch slov. Čo ak potrebujeme porovnať veľké množstvo korektných slov so slovom chybným. Na to budeme potrebovať veľmi šikovný algoritmus a dátovú štruktúru.

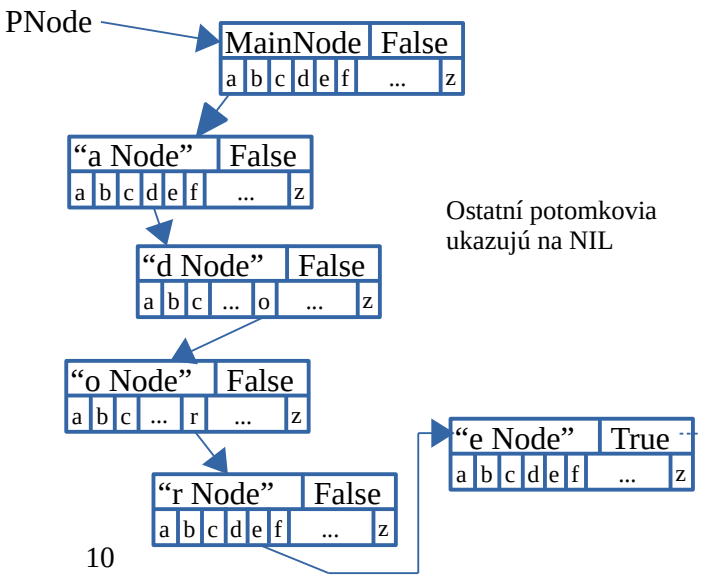
2.2 Algoritmus – porovnanie množiny korektných slov so slovom chybným

2.2.1 Dátová štruktúra – Trie (prefixový strom)

Trie je špeciálny druh stromu, ktorý sa od binárneho stromu líši v tom, že každý vrchol (Node) má namiesto pravého a ľavého potomka hneď niekoľko potomkov. Táto vlastnosť umožnuje, aby každé písmeno z intervalu „a“ .. „z“ bolo reprezentovateľné nejakým potomkom v rámci jedného vrcholu. Teda každý vrchol má 26 potomkov (je možné rozšíriť na ľubovoľný počet potomkov, slovník je ale obmedzený na znaky z intervalu „a“ .. „z“).

Predstavme si slovo „**adore**“. Počiatočný vrchol stromu nebude reprezentovať žiadne písmeno. Avšak jeho potomkovia budú ukazovať na vrcholy, ktoré budú reprezentovať jedno písmeno slova. Prvý potomok ukazuje na vrchol reprezentujúci „**a**“, ostatní potomkovia ukazujú na **NIL**. Presuňme sa do vrcholu „a“, ten má ďalších 26 potomkov. Jeden z nich ukazuje na vrchol reprezentujúci „**d**“, ostatní ukazujú na **NIL.** Takto môžeme pokračovať až k vrcholu „**e**“, kde jeho potomkovia nebudú ukazovať na žiaden ďalší vrchol, teda všetci ukazujú na **NIL.** Takýmto spôsobom môžeme reprezentovať slovo v strome. Výhodou takto uložených slov je ich rýchle vyhľadávanie, kedže pri hľadaní slova sa nám stačí vybrať iba určitou vetvou v strome. Každý vrchol navyše obsahuje **značku –** booleovskú premennú, ktorá nám napovie, že v tomto mieste je koniec slova. Napríklad po slove „**adore**“ chceme pridať slovo „**adorers**“. Vrchol „e” by obsahoval ďalšieho potomka a stratili by sme informáciu o pôvodnom slove.

Príklad 2.2.1 – Trie:



Reprezentácia v kóde programu:

```
type
  PNode = ^Node;
  Node = Record
    IsWord: Boolean;   { značka }
    ArrOfPtrs: Array['a'..'z'] of PNode;
    { pole ukazateľov, potomkov na PNode }
  end;
```

2.2.2 Procedúra pre vkladanie slov do Trie

Vkladanie slov do stromu je reprezentované procedúrou:

```
procedure InsertWordToTrie(word: string; RootNode: PNode);
```

word – slovo, ktoré chceme vložiť; **RootNode** – ukazateľ na hlavný vrchol stromu, ktorý nereprezentuje žiadne písmeno.

Procedúra prechádza vetvami stromu podľa písmen vkladaného slova. Každé písmenko slova, ktoré vkladáme sa buď nachádza v strome vo forme vrcholu alebo je nútne pridať nový vrchol v závislosti na písmenku. Vrchol, ktorý reprezentuje posledné písmeno slova označíme značkou **True**. Pri vytváraní nových vrcholov procedúra zavolá funkciu:

```
function CreateNewNode(is_word: Boolean): PNode;
```

Tá zabezpečí vytvorenie vrcholu a nastavenie hodnoty **IsWord** na True alebo False v závisloti na tom, či nový vrchol bude reprezentovať posledné písmeno slova alebo nie. Funkcia následne vráti ukazateľ na nový vrchol. Potomok, ktorý ukazoval na **NIL** bude ukazovať na nový vrchol.

Procedúra InsertWordToTrie je volaná v cykle na začiatku programu, kedy sa obsah súboru (slovníka) **dic_edit.dat** vkladá do stromu. Obsah stromu je po celú dobu behu programu k dispozícií ostaným procedúram a funkciám. Procedúra InsertWordToTrie je volaná aj v prípade, že chceme do stromu pridať nové slovo zadané zo vstupu.

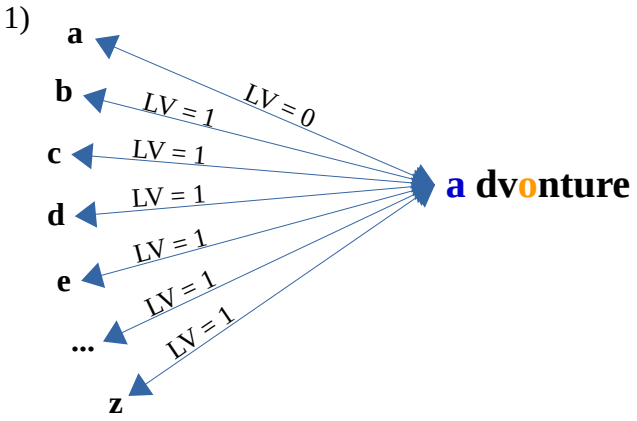
```
Number of words loaded: 351076.
Processing time: 276 ms.
-------Loading Finished!-------
```

Posledným krokom k tomu, aby algoritmus pre porovnávanie množiny korektných slov a chybných slov fungoval, je spojenie algoritmu Levenshteinovej vzdialenosti s dátovou štruktúrov Trie.

2.2.3 Levenshteinova vzdialenosť & Trie

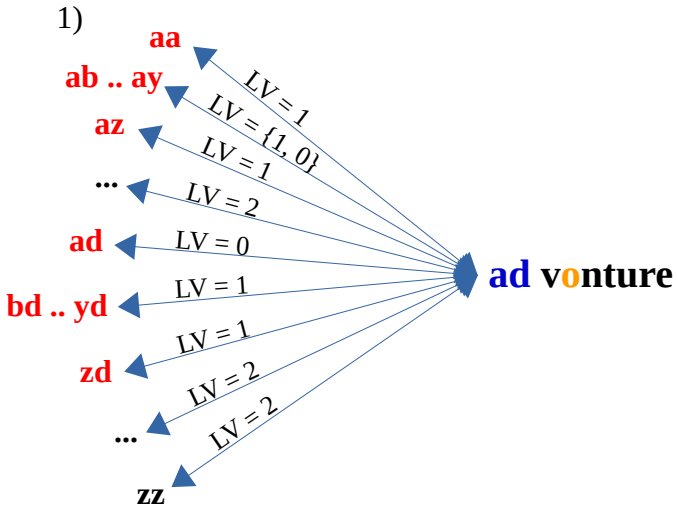
Základnou myšlienkou pri porovnávani slov je postupne si vyberať zo stromu každého z 26 potomkov, teda písmeno od „**a**“ po „**z**“. Zistiť, či daný potomok ukazuje na koniec stromu alebo ukazuje na nasledujúci vrchol. Každé z písmen „**a**“ .. „**z**“ treba porovnať s chybným slovom a to tak, že to porovnáme z prvým písmenom chybného slova. Pri každom porovnaní zistíme hodnotu vzdialenosti pomocou Levenshteinovej vzdialenosti a zhodnotíme, či je potrebné porovnávať aj naďalej. Ak áno, v ďalšom kroku sa rekurzívne vnoríme do nižšej vrstvy stromu, teda navštívime vrcholy, na ktoré ukazujú vhodný potomkovia. Znova v rámci týchto vrcholov vyhodnotíme všetkých potomkov od „**a**“ po „**z**“. V tomto kroku porovnávame vhodné kombinácie dvojíc písmen s prvými dvoma písmenami chybného slova. Určíme vzdialenosť a budeme vedieť, ktorých potomkov máme v ďalšom kroku nasledovať. Pred každým ďalším vnorením je potrebné si overiť, či nebola prekročená maximálna vzdialenosť, aby sme v strome zbytočne nezachádzali do ďalších vrstiev. Pred každým vnorením je samozrejme potrebné overiť, či potomkovia neukazujú na **NIL.** Ak sa počas vnárania nájde vrchol so značkou **True**, našli sme **zhodné slovo**. Algoritmus však pokračuje aj naďalej a hľadá všetky **podobné slová**.

Príklad 2.2.3: Chybné slovo - „**advonture**“. V prvom kroku porovnávame všetkých potomkov s prvým písmenom chyb. slova (označme **LV** - Levenshteinová vzdialenosť).



LV ≤ 1: V prvom kroku je možné sa rekurzívne vnoriť do všetkých vetiev

V nasledujúcom kroku porovnávame vhodných potomkov s prvými dvoma písmenami chybného slova.



LV ≤ 1: V druhom kroku je možné sa rekurzívne vnoriť do všetkých vetiev vyznačených červenou farbou. Takto sa postupne okruh kandidátov na zhodné alebo podobné slová zužuje.

V poslednom kroku nám v tomto prípade zostane jediný kandidát: **adventure** ⟷ **advonture**.

Kandidát musí byť zároveň slovom (posledné pismeno „**e**“ musí byť označené značkou True). Algoritmus samozrejme funguje aj pre ostané chybové prípady, napr. **advnture**, **advenutre**, **adventur.** Tieto porovnania v programe zabezpečuje funkcia:

```
function SearchForSimilarWords(RootNode: PNode;
SomeWord: String): PWordsRecord;
```

RootNode – ukazateľ na počiatočný vrchol stromu Trie; **SomeWord** – reprezentuje chybné slovo. Jej úlohou je vrátiť spojový zoznam **string** reťazcov (slov), kde prvé slovo v spojovom zozname je slovo zhodné, ostatné slová v zozname sú podobné. Ak je prvé slovo prázdne, teda **string** neobsahuje žiadne znaky, nenašla sa zhoda. Rovnako aj v prípade, že spojový seznam obsahuje iba jedno slovo a ďalej

ukazuje na **NIL** značí, že sa nenašli podobné slová. Funkcia SearchForSimilarWords nastavý počiatočné hodnoty pred zavolaním rekurzívne sa vnárajúcej procedúry **SearchCurrentIndex**, ktorá priamo využíva Levenshteinovú vzdialenosť a vyberá zo stromu postupne kandidátov na porovnanie.

```
procedure SearchCurrentIndex(CurrentNode: PNode; ListOfNodes:
PWordsRecord; CurrentWord: String; CurrentWordSize: Byte; Word:
String; WordSize: Byte; MaxLevensDist: Byte);
```

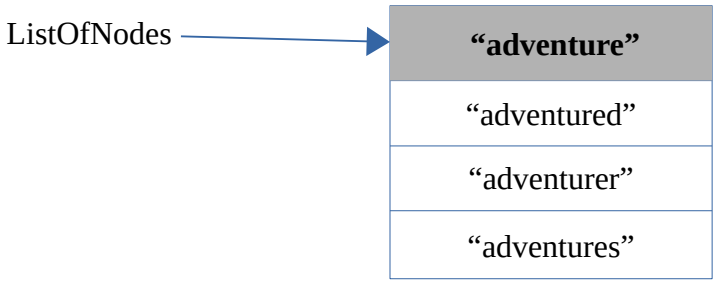
`CurrentNode` – ukazateľ na vrchol stromu, v ktorom sa práve algoritmus pri porovnávaní nachádza. Na začiatku je ukazateľ nastavený na začiatok stromu.
`ListOfNodes` – ukazateľ na zoznam najdených slov
`CurrentWord` – slovo, ktoré sa práve porovnáva, napr. **a**; **ad; adv; adve; adven; … adventure**
`CurrentWordSize` – dĺžka slova CurrentWord, informácia pre procedúru LevenshteinDistance, konkrétne hodnota pre premennú **length_b –** dĺžka korektného slova, ktorú chceme porovnať
`Word` – chybné slovo
`WordSize` – akú časť chýbneho slova chceme porovnať, teda počet písmen od začiatku slova. Ako bolo uvedené v príklade 2.2.3, porovnáva sa iba taký počet písmen, aká je hĺbka vnorenia. Informácia pre procedúru LevenshteinDistance, konkrétne pre premennú **length_a.** Hodnota je zhodná s **length_b.**
`MaxLevensDist` - maximálna povolená vzdialenosť

Reťazec **CurrentWord** je na začiatku prázdny. Postupným vnáraním procedúry SearchCurrentIndex sa reťazec rozširuje o nové znaky v závisloti na vrchole. Zároveň sa mení aj jeho dĺžka **CurentWordSize**. Tieto údaje sú dôležité pre následné volanie procedúry LevenshteinDistance. Podľa vrátenej vzdialenosti a hodnoty v premennej **IsItWorthIt (**je tiež návratovou hodnotu z proc. LV**)** sa algoritmus rozhodne, či bude pokračovať vo volaní procedúry SearchCurrentIndex.

Pri každom ďalšom zavolaní procedúry SearchCurrentIndex je obsah reťazca CurrentWord iný. Postupne sa pridávajú znaky „**a**“ + „**d**“ + „**v**“ + „**e**“ + „**n**“ + … + „**e**“, pri backtrackingu sa zase znaky odoberajú (resp. inkrementuje a dekrementuje sa hodnota **CurrentWordSize** a **WordSize**). Reťazec **Word** a hodnota **MaxLevensDist** ostávajú pri každom volaní nezmené.

Ukazateľ **CurrentNode** určuje na aký vrchol v strome sa ukazuje. Ak sa ukazuje na **NIL** a nenašla sa zhoda ani podobné slovo, backtrackingom sa vraciame do vyšších úrovní stromu.

Ukazateľ **ListOfNodes** ukazuje na začiatok spojového zoznamu slov, ktoré sme už našli a ostavá nezmený pri každom ďalšom volaní procedúry SearchCurrentIndex. Vždy, keď sa v danom vrchole stromu nájde značka True (koniec slova), je spojový zoznam rozšírený o nové slovo. Ak je hodnota **LV=0** a značka je **True**, tak sme našli zhodné slovo. To sa uloží do prvého reťazca v zozname (reťazec rezervovaný pre zhodné slovo). Ak je **0 < LV ≤ 1** a značka je **True**, podobné slovo sa príde na koniec zoznamu. Výhodou takto prechádzaného stromu je, že výsledný zoznam je (okrem prvého rezervovaného slova) zoradený v abecednom poradí.



Funkcie a procedúry spomenuté dosposiaľ boli najdôležitejšie. Ostané procedúry a funkcie, ktoré uvediem, vychádzaju vo väčšej miere z tých dôležitých.

2.2.4 Ostatné procedúry a funkcie

Po tom ako program načíta slová do Trie je volaná procedúra:

```
procedure SelectOption(RootNode: PNode);
```

Užívateľovi sa na výstupe zobrazí menu s možnosťou výberu. Podľa toho, čo užívateľ vyberie sú volané niektoré z procedúr:
1. - SpellChecker,
2. - AddWordToDictionary,
3. - SearchWordInDictionary,
4. - SaveChangesInDictionary.

1. SpellChecker:

```
procedure SpellChecker(RootNode: PNode; Path: String);
```

RootNode – ukazateľ na počiatok Trie;
Path – cesta k adresáru s textovým vstupom

Procedúra dostane ako vstup ukazateľ na Trie a cestu k adresáru odkiaľ má načítať textový súbor pre následné spravovanie. Slová v súbore sú spracovávané po znakoch, kedže ide o súvislý text. Akceptovateľné sú slová obsahujúce Lower a Upper Case znaky „a“ .. „z“ Každé zo slov v súbore musí byť oddelené niektorým z povolených znakov: medzerou, zátvorkou, bodkou, apostrofom. Číselné údaje sú ingnoravané a ponechané bez zmeny. Písmeno za apostrofom je taktiež ponechané bez zmeny (algoritmus nevie rozlišovat gramatiku a kontext). Každe načítané slovo sa porovná so slovníkom. V každom cykle sa zavolá proc. SearchForSimilarWords (ktorú sme už spomenuli) a vratí sa spojový zoznam slov. Algoritmus následne vyberie zhodné slovo (ak existuje) a pošle ho na výstup. Taktiež pomocou cyklu sa prechádza spojový zoznam a na výstup sa posielajú aj ďalšie podobné slová. Na výstup v konzole sa vypíšu iba slová, ktoré boli chybné alebo neznáme. K nim sa vypíšu alterantívy, ak existujú. Výstupom do súboru (**output.txt**) bude pôvodný text a chybné či neznáme slová sa nahradia ponúkaným zoznam alternatív.

2. AddWordToDictionary:

```
function AddWordToDictionary(RootNode: PNode): Boolean;
```

RootNode – ukazateľ na počiatok Trie

Funkcia prídáva do Trie slovo, ktoré užívateľ zadá na vstupe. Každe slovo je pred pridaním kontrolované podmienkou, že musí obsahovať iba písmená abecedy (akceptované sú lower a upper case znaky „a“ .. „z“). Slovo s uppercase znakmi sa pred uložením automaticky upraví na lowercase. Zavolaním proc. InsertWordToTrie (ktorú sme už spomenuli) sa slovo prída do Trie. Výstupom z funkcie je, že došlo k zmene Trie. Táto zmena ostane zaznamenaná v booleoskej premennej **change.**

3. SearchWordInDictionary:

```
procedure SearchWordInDictionary(RootNode: PNode);
```

RootNode – ukazateľ na počiatok Trie

Procedúra načíta zo vstupu slovo. Podobne ako v predchádzajucích procedúre podmienka overí, či slovo neobsahuje nepovolené znaky. Zavolá sa proc. SearchForSimilarWords a hľadané slovo vyhľadá. Je vrátený spojový zoznam. Na výstup sa vypíše zhodné slovo (ak existuje) a následne pomocou cyklu sa prejde spojovy zoznam a vypíšu sa všetky ďalšie alternatívy.

4. SaveChangesInDictionary:

Tesne pred ukončením programu pri zaznamenanej zmene v strome (premenná **change = True**) sa zavolá procedúra:

```
procedure SaveChangesInTDictionary(RootNode: PNode);
```

RootNode – ukazateľ na počiatok Trie

ktorej úlohou je uložiť pozmený strom naspať do súboru **dic_edit.dat**. Súbor **dic_edit.dat** sa pripravý na prepis pomocou proc. Rewrite(). Vytvorí sa ukazateľ „p“ (ukazateľ na File typu Text), ktorý ukazuje na Text. Ten je dostupný v každom rekurzívnom volaní procedúry (teda nezáleží, kde v strome sa práve nachádzame, vždy máme prístup na uloženie slova do súboru):

```
procedure SaveEachWordFromTrie(CurrentNode: PNode; CurrentWord:
String; FilePointer: PFile);
```

CurrentNode – ukazateľ na aktuálny vrchol v Trie;
CurrentWord – podobne, ako v prípade CurrentWord v proc. SearchCurrentIndex;
FilePointer – ukazateľ na súbor, do ktorého ukladáme slová

Ak v strome narazíme na vrchol, ktorý obsahuje značku **True,** vieme, že sme našli slovo v slovníku a zapíšeme ho do súboru. Slová sa uložia v abecednom poradí (vlastnosť Trie a rekurzívneho volania) každé na nový riadok, tak ako tomu bolo v pôvodnom **dict_edit.dat.**

Po uložení zmien sa „vyskočí“ z procedúry SelectOption. Potom dôjde k ukončeniu programu.

2.2.4 Úprava kódu, rozšírenie o nové funkcie

Ako už bolo v texte spomenuté Levenshteinovú vzdialenosť je možné upraviť na vačšiu ako 1. Rozsah vyhľadávania sa rozšíri, ale presnosť sa zníži. Doba vyhľadávania sa takztiež predĺži. Zmenu je možné previesť pomocou úpravy v procedúre SearchForSimilarWords – konkrétne:

```
SearchCurrentIndex(RootNode, ListOfWords, EmptyWord, Size, SomeWord,
Length(SomeWord), 1);
```

Dátovú štruktúru Trie je možné rozšíriť na ľubovoľný druh a počet znakov, ktoré budú reprezentované vrcholmi stromu. To sa samozrejme odrazí na veľkosti použitej pamäte a rýchlosti vyhľadávania. Mnou zvolenú dátovú štruktúru je možné vymeniť aj za BK-Tree (Burkhard & Keller) [3].

Mojím plánom je rozšíriť program o funkciu pre odstraňovanie slov z Trie. Implementácia je vskutku jednoduchá. Ak by daný vrchol v strome obsahoval značku **Trie** a neodkazoval by na **NIL**, tak stačí zmeniť hodnotu na **False**. V prípade, ak by odkazoval na **NIL** by bolo potrebné daný vrchol odstrániť. K tomu by sme ale potrebovali ukazateľ na predchádzajúci prvok. A samozrejme by bolo potrebné zavolať procedúru Dispose().

Rereferencie:

1. Textový súbor obsahujúci 350. tisíc slov, zdroj: https://github.com/dwyl/english-words
2. Damerau–Levenshteinová vzdialenosť, zdroj: https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance
3. BK-Tree (Burkhard & Keller), zdroje:

   https://en.wikipedia.org/wiki/BK-tree

   https://nullwords.wordpress.com/2013/03/13/the-bk-tree-a-data-structure-for-spell-checking/