Penerapan Pohon *Ternary DAG* pada *Spelling Suggestion*

Kadek Surya Mahardika and 13519165[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13519165@itb.ac.id

***Abstract*—Semakin banyaknya kegiatan tulis menulis mengakibatkan diperlukannya aplikasi dengan kemampuan spelling checker dan suggestion yang dapat dengan cepat mencari kata-kata di dalam kamus yang terdapat ratusan ribu kata. Struktur data dipilih akan mengakibatkan perbedaan waktu pencarian. Ternary DAG adalah salah satu struktur data yang secara efisien dapat mencari kata-kata yang mirip di dalam kamus.**

***Keywords*—Pohon, Ternary, DAG, Hash, Kompleksitas, Trie, BST.**

I. PENDAHULUAN

Di zaman dengan penuh teknologi ini, menulis merupakan hal yang tidak sulit lagi, berbeda dengan dulu dimana kesalahan penulisan pastinya akan sangat rentan terjadi, sekarang, dengan teknologi *spelling suggestion* yang terpasang pada aplikasi yang kita gunakan, kesalahan ejaan kata dapat sangat cepat dideteksi.

Pernahkah anda bertanya, bagaimana suatu aplikasi misalnya Microsoft Word dapat mencari kemungkinan ejaan kata yang benar? Mungkin anda berfikir bahwa aplikasi tersebut mencoba semua kemungkinan kata yang ada di kamus secara naif. Namun, kita tahu bahwa ada jutaan kata yang ada di dalam suatu bahasa, mencoba semua kata secara naif akan memakan banyak waktu, belum termasuk kata berimbuhan dan berakhiran.

Struktur data pohon adalah jawabannya, struktur data ini membantu algoritma pencarian kata dengan cepat. Secara umum, ada banyak jenis struktur data pohon yang dapat digunakan untuk *spelling suggestion* seperti Trie, BST, BK-Tree dan TST. Masing-masing dari struktur data tersebut memiliki kelebihan dan kekurangannya masing-masing.

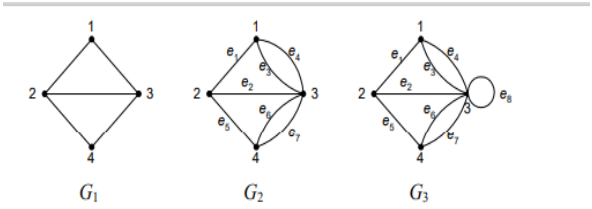
Pada paper ini penulis akan menjelaskan penerapan struktur data *Ternary DAGs (Directed Acyclic Graph)* pada *spelling suggestion*. *Ternary DAGs* merupakan modifikasi dari struktur data TST (*Ternary Search Tree)* dengan tambahan graph di dalamnya.

II. LANDASAN TEORI

A. Graf (*Graph)*

Secara sederhana, graf adalah suatu representasi untuk menggambarkan hubungan (sisi/*node*) objek-objek diskrit (simpul/*vertex*). Secara formal, Graf dapat didefinisikan sebagai pasangan himpunan (V, E), dalam hal ini V adalah himpunan tidak kosong dari simpul-simpul (*vertices*) dan E adalah himpunan sisi (*edges*) yang menghubungkan simpul-simpul di V [1]. Berikut contoh-contoh dari graf:

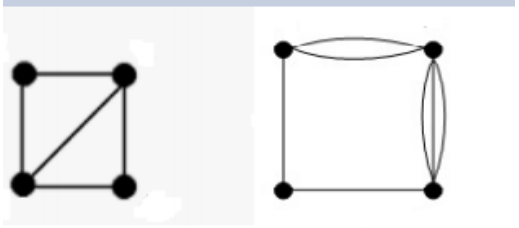
(Sumber: Graf (Bag. 1) oleh Rinaldi Munir)



Gambar 2. (a) graf sederhana, (b) graf ganda, dan (c) graf semu

Dari gambar diatas, terlihat bahwa terdapat berbagai jenis graf. Graf dapat digolongkan dengan adanya gelang/tidak dan orientasi arahnya.

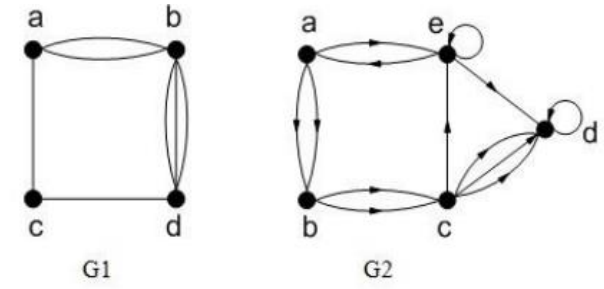
Berdasarkan adanya gelang/tidak, graf dapat digolongkan menjadi dua jenis yakni graf sederhana (*simple graph)* dan graf tak-sederhana (*unsimple-graph)*. Graf sederhana adalah graf tanpa gelang dan sisi ganda sedangkan graf tak-sederhana sebaliknya. Berikut gambarnya:



(Sumber: Graf (Bag. 1) oleh Rinaldi Munir)

Sisi kiri merupakan graf sederhana, sedangkan sisi kanan tak-sederhana.

Berdasarkan orientasi arahnya, graf dapat digolongkan menjadi dua yakni graf berarah (*directed graph)* dan graf tak-berarah *(undirected graph)*. Berikut gambarnya:



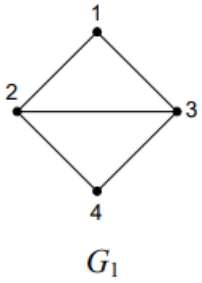
G1 : graf tak-berarah; G2 : Graf berarah

(Sumber: Graf (Bag. 1) oleh Rinaldi Munir)

Untuk memahami makalah ini, ada beberapa terminologi graf yang harus dipahami, diantaranya:

1. Ketetanggan (*Adjecent*)

Dua buah simpul dapat dikatakan bertetangga jika keduanya terhubung langsung, sebagai gambaran:



G_1

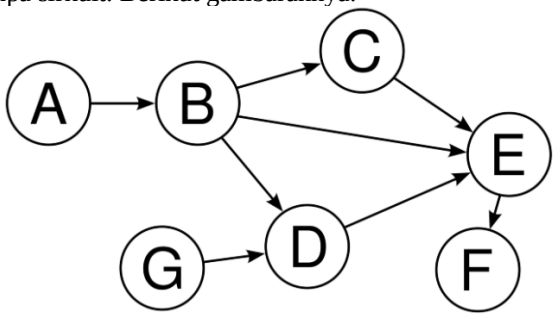
(Sumber: Graf (Bag. 1) oleh Rinaldi Munir)

Pada gambar tersebut, simpul 1 dengan simpul 2 bertetangga, karena keduanya terhubung secara langsung. Sedangkan, simpul 1 dengan simpul 4 tidak bertetangga, karena tidak ada sisi yang mengubungkan kedua simpul tersebut.

2. Derajat (*Degree)*

Derajat dari seuatu simpul adalah jumlah tetangga yang terhubung ke simpul tersebut. Tinjau gambar diatas, simpul 1 memiliki derajat 2, karena terdapat 2 tetangga yang terhubung dengan dirinya yakni simpul 2 dan simpul 3, simpul 3 memiliki derajat 3, karena terdapat 3 tetangga yang terhubung dengannya, dst.

Kita sudah mengetahui bahwa graf dapat dibagi menjadi dua yakni graf berarah dan tak-berarah. Dalam graf berarah terdapat graf yang bernama *Directed Acyclic Graph* atau graf berarah tanpa sirkuit. Berikut gambarannya:



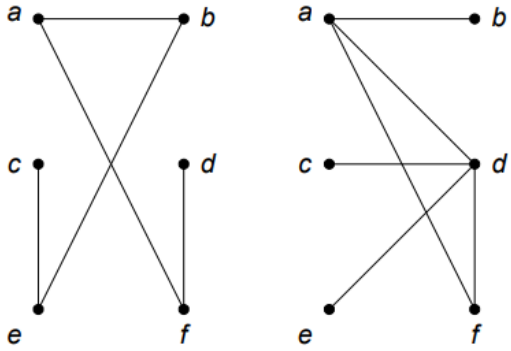
Directed Acyclic Graph

(Sumber: https://medium.com/kriptapp/guide-what-is-directed-acyclic-graph-364c04662609)

Terlihat pada gambar, graph tersebut merupakan graph berarah, dan tidak adanya sirkuit berarah di dalamnya, sirkuit dalam hal ini merupakan suatu lintasan yang dimulai dari simpul tertentu dan pada akhirnya berakhir di simpul semula.

B. Pohon (*Tree)*

Struktur data pohon adalah suatu graf tak berarah yang terhubung dan tidak mengandung sirkuit di dalamnya.



pohon bukan pohon

(Sumber : Pohon (Bag. 1) oleh Rinaldi Munir)

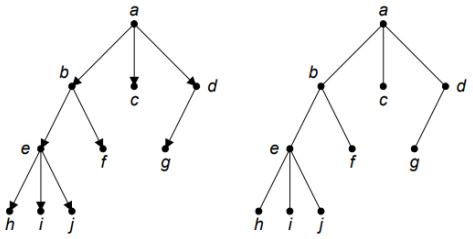
Secara formal, jika G = (V, E) adalah pohon yakni graf tak-berarah sederhana dengan jumlah simpul n, G memiliki properti sebagai berikut:

- Setiap pasang simpul di dalam G terhubung dengan lintasan tunggal
- G memiliki n-1 sisi
- G tidak mengandung sirkuit
- Penambahan satu sisi pada G, hanya membuat satu sirkuit
- Semua sisi dari G adalah jembatan

Ada beberapa terminologi pada pohon:

1. Pohon berakar (*rooted tree*)

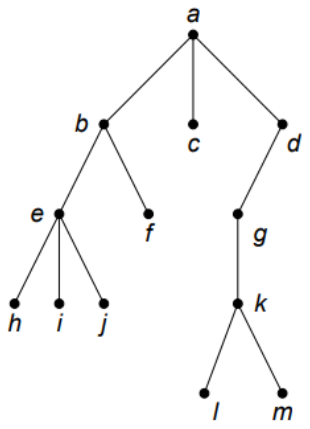
Pohon berakar merupakan pohon yang satu simpulnya diperlakukan sebagai akar dan dan sisi-sisinya diberikan arah. [2]



(a) Pohon berakar (b) sebagai perjanjian, tanda panah pada sisi dapat dibuang

(Sumber : Pohon (Bag. 2) oleh Rinaldi Munir)

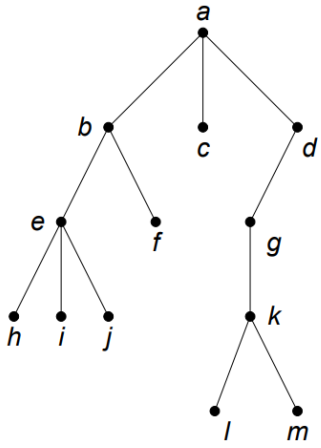
2. Anak (*Child)* dan Orangtua (*Parent*)



(Sumber : Pohon (Bag. 2) oleh Rinaldi Munir)

Sebagai contoh, pada pohon berakar diatas a adalah orangtua dari anak-anaknya yakni b, c, dan d.

3. Lintasan (*Path)*

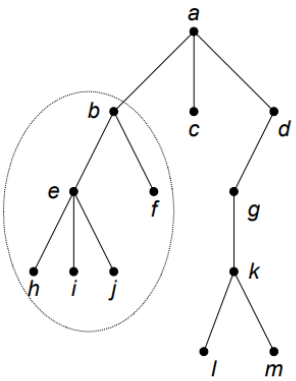


(Sumber : Pohon (Bag. 2) oleh Rinaldi Munir)

Sebagai contoh, pada pohon berakar diatas terdapat lintasan dari simpul a ke m yakni a-d-g-k-m, dimana panjang lintasannya adalah jumlah sisi yang dilalui yaitu 4.

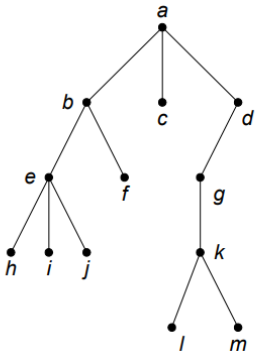
4. Upapohon *(Subtree)*

Subtree dari suatu pohon adalah bagian dari suatu pohon berakar yang masih memiliki simpul minimal satu. Contohnya sebagai berikut:



5. Derajat (*Degree)*

Derajat dari pohon berakar adalah jumlah simpul yang keluar dari simpul tersebut atau dengan kata lain jumlah anak pada simpul tersebut.



(Sumber : Pohon (Bag. 2) oleh Rinaldi Munir)

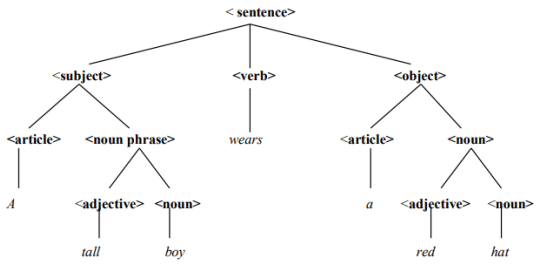
Sebagai contoh, pada pohon diatas, simpul a memiliki derajat 3, karena ia memiliki 3 anak yakni b,c, dan d.

6. Daun *(Leaf)*

Daun dari suatu pohon adalah simpul-simpul yang tidak memiliki anak, dari gambar diatas, simpul h,i,j,i,m merupakan beberapa daun.

7. Pohon n-ary

Pohon n-ary merupakan pohon yang setiap simpulnya memiliki anak maksimal n. Jika n=2 disebut *binary tree*, n=3 disebut *ternary tree*, dsb. Berikut contohnya:



Pohon diatas merupakan pohon biner (*binary tree*). [3]

III. PEMBAHASAN

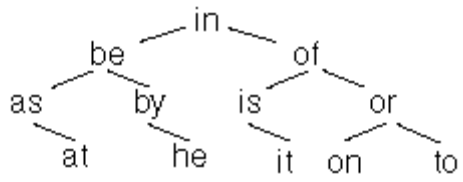
A. Latar Belakang Penggunaan Ternary DAGs

Secara umum, banyak struktur data yang dapat digunakan sebagai dasar implementasi dari *Spelling Suggestion/Correction*, jika kita hanya berfokus kepada pengejaannya saja struktur data Hash Table sangatlah tepat untuk menyelesaikan permasalahan demikian, karena secara *amortized* kompleksitas pencarian dari hash table adalah O(1). Namun, pencarian dengan menggunakan hash table akan menghilangkan *relative order* terhadap string-string yang lain pada kamus yang akan sangat berguna untuk *spelling suggestion*. Sehingga, hash table bagus untuk pencarian satu kata/string namun tidak untuk mencari kemiripan.

Untuk permasalahan *spelling suggestion* dapat digunakan struktur data seperti BST (Binary Search Tree) atau Trie. Kedua struktur data tersebut memiliki kelemahannya masing-masing dimana BST memakan memori yang lebih sedikit dari Trie tapi tidak secepat Trie sedangkan Trie dapat melakukan pencarian dengan cepat tapi memakan banyak memori. Adapun solusi

win-win dari kelemahan kedua struktur data tersebut, yakni dengan menggunakan TST (Ternary Search Tree). Mengapa demikian? Sebelumnya kita harus mengetahui bagaimana BST dan Trie dapat digunakan sebagai *spelling suggestion.*

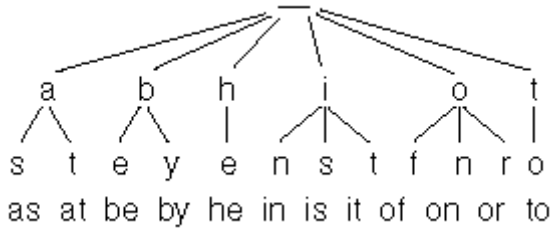
BST dapat digunakan sebagai pencari kata adalah dengan mengatur setiap simpul pada setiap levelnya sehingga setiap simpul menampung suatu kata dan memiliki dua anak yang mana anak kiri merupakan simpul dengan kata yang bernilai lebih kecil dari kata sekarang dan anak kanan yang merupakan simpul dengan kata yang bernilai lebih besar dari kata sekarang. Sebagai gambaran:



(Sumber : Ternary Search Trees | Dr Dobb's)

Asumsikan BST yang telah dibagun memiliki struktur seperti diatas, saat kita mencari kata "he", dari simpul teratas akan dibandingkan apakah "he" bernilai lebih kecil atau lebih besar sehingga pada akhir kita akan berakhir di simpul "he".

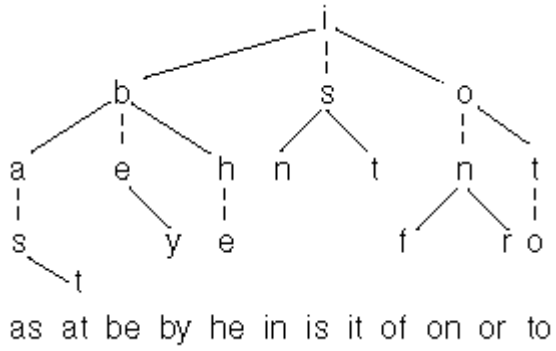
Berbeda dengan struktur data BST, struktur data Trie menggunakan pendekatan per karakter. Setiap simpul dari trie memiliki 26 anak yang merupakan abjad dari a-z, sehingga cara pencariannya *straight foward* yakni dari simpul teratas bandingkan setiap karakter yang dicari dengan anak-anak pada simpul tersebut. Berikut gambarannya:



(Sumber: Ternary Search Trees | Dr Dobb's)

Karena setiap simpulnya terdapat 26 abjad, struktur data trie sangatlah *space inefficient*, sebagai gambaran, setiap node dengan 26 anak akan memakan memori 1024 byte, tidak terbatas pada 26 abjad, kita dapat menambahkannya sampai 256 karakter (256-way) yang akan memakan memori sekitar 1 kilobyte sehingga jika kita bandingkan dengan banyaknya kata pada kamus, strutur data ini akan memakan banyak memori. [5]

TST atau *Ternary Search Tree* adalah struktur data yang menggabungkan efisiensi waktu dari Trie dan efisiensi ruang dari BST. TST pada *spelling suggestion* bekerja seperti Trie yakni dengan memproses satu persatu huruf yang ada pada kata. Namun, tidak seperti Trie yang memiliki banyak anak, TST hanya memiliki tiga anak/pointer yakni pointer paling kiri merepresentasikan karakter yang lebih kecil dari karakter yang dicari, pointer ditengah merepresentasikan karakter yang sama dengan karakter sekarang dan anak paling kanan merepresentasikan karakter yang lebih besar dari karakter yang dicari. Berikut gambarannya:



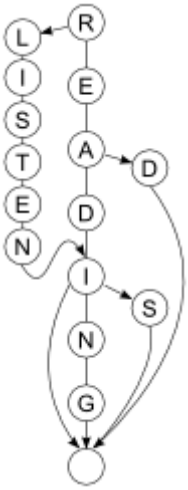
Jika kita implementasikan dalam bahasa C, berikut implementasi strukturnya:

```c
typedef struct tnode *Tptr;
typedef struct tnode {
   char splitchar;
   Tptr lokid, eqkid, hikid;
} Tnode;
```

(Sumber : Ternary Search Trees | Dr Dobb's)

Saat TST dipakai untuk *spelling suggestion* kita bisa menelusuri simpul-simpul yang ada dengan cepat dari query yang diberikan, hal lain yang perlu diperhatikan yakni pada saat query mencari kata yang salah/tidak pas hanya *prefix* dari kata itu dicari di TST sehingga kita dapat memanfaatkan itu sebagai *spelling suggestion*. Sebagai contoh, anggaplah TST yang dibagun mengandung kata *read, reading, job, jobs, program, programmer, programmed*, saat terjadi kesalahan penulisan misalnya *progrm*, TST hanya mengecek sampai prefix "*progr*" dan mengembalikan semua kemungkinan dari prefix "*progr*". Perhatikan bahwa TST dapat dikatakan "mengkompres" prefix dari kata yang dicari. Untuk meningkatkan efisiensi yakni mengurangi jumlah simpul yang ada, kita dapat mengkompres suffix dari kata yang dicari dengan bantuan DAG (*Directed Acyclic Graph*), struktur data ini disebut sebagai sebagai Ternary DAG. [4]

Ternary DAG sesuai namanya adalah gabungan dari struktur data TST dan DAG. Pada aplikasinya sebagai *spelling suggestion* Ternary DAG mengkompres suffix dari kata-kata yang ada di kamus yang mengandung akhiran -ing, -ead, -s, dsb kedalam satu kelompok subtree yang sama, penghubungan itu dibantu dengan DAG. Sebagai ilustrasi perhatikan gambar berikut:



(Sumber: Using ternary DAGs for spelling correction - strchr.com)

Semua kata-kata yang mungkin memiliki suffix yang sama akan diarahkan ke satu subtree yang sama, contohnya kata reading dan listening yang akan diarahkan ke subtree setelah simpul D. Sehingga, dengan cara seperti ini, suffix di kamus dapat di kompres dan penggunaan simpul akan berkurang.

B. Implementasi Ternary DAG

Implementasi Ternary DAG dapat dibagi menjadi dua langkah yakni membagun TST dari kamus yang diberikan dan kompresi TST dengan DAG. Struktur dari TST yang dibangun adalah sebagai berikut:

```
struct ST_MEM_NODE {
    char ch;
    unsigned count;
    unsigned hash;
    unsigned no;
    struct ST_MEM_NODE *left;
    struct ST_MEM_NODE *middle;
    struct ST_MEM_NODE *right;
};
typedef struct ST_MEM_NODE ST_MEM_NODE;
```

(Sumber: Using ternary DAGs for spelling correction - strchr.com)

Struktur tersebut merupakan simpul dari setiap simpul pada TST, untuk kompresi TST dengan DAG kita dapat mengakalinya dengan membangun hash table yang menampung subtree dari TST yang kita buat, oleh karenanya setiap simpul pada struktur diatas terdapat nilai hashnya. Berikut kode dalam bahasa Cnya:

```
// Find the node in hash table. If it does not exists, add a new one and return true.
// If it does exists, return false.
static bool check_and_remove_duplicate(ST_MEM_NODE** node_ptr) {
    ST_MEM_NODE * node = *node_ptr;
    unsigned hash = node->hash;
    while(g_table[hash] != 0) {
        if(equal(g_table[hash], node)) {
            // This node already exists in the table. Remove the duplicate
            free_node(node);
            *node_ptr = g_table[hash];
            return false;
        }
        hash = (hash + 1) % TABLE_SIZE;
    }
    g_table[hash] = node;
    return true;
}

// Remove duplicating suffixes starting from the longest ones
static void remove_duplicates(ST_MEM_NODE* node) {
    if(node->left) {
        // If the node already exists in the table (check_and_remove_duplicate returned false),
        // its children were checked for duplicates already, so we don't check them twice
        if(check_and_remove_duplicate(&node->left))
            remove_duplicates(node->left);
    }
    if(node->right) {
        if(check_and_remove_duplicate(&node->right))
            remove_duplicates(node->right);
    }
    if(node->middle) {
        if(check_and_remove_duplicate(&node->middle))
            remove_duplicates(node->middle);
    }
}
```

(Sumber: Using ternary DAGs for spelling correction - strchr.com)

Inti dari kode diatas adalah secara rekursif kita cek apakah simpul yang kita cari sekarang sudah ada di hash-table, jika ada, kita ganti ke subtree suffix yang sudah ada. [2]

Untuk algoritma pencarian dan matching kata yang sesuai terdapat beberapa cara yang bisa digunakan yakni dengan mengecek apakah kata query miss satu huruf, terdapatnya karakter yang berlebih, karakter yang salah, dsb. Berikut salah satu contoh handling kasus ketika ada karakter yang berlebih:

```
// Correct double letters (including multiple errors in one word, e.g. Misisipi)
static void match_doubles(ST_FILE_NODE* node, const char* word, char* result,
                          ST_ENUMINFO * enuminfo) {
    while(!is_null_link(node, enuminfo->dict)) {
        if(*word < node->ch)
            node = get_left(node, enuminfo->dict);
        else if(*word > node->ch)
            node = get_right(node, enuminfo->dict);
        else {
            *result = *word;
            if(*result == '\0') {
                // Found the word
                report_suggestion(enuminfo);
                return;
            }
            if(*(word + 1) == *word) {
                // When found two equal letters, try to find a single letter
                match_doubles(get_middle(node, enuminfo->dict), word + 2, result + 1, enuminfo);
            } else {
                // Otherwise, try to find two equal letters here
                match_doubles(get_middle(node, enuminfo->dict), word, result + 1, enuminfo);
            }
            // Also find exact match
            node = get_middle(node, enuminfo->dict);
            word++;
            result++;
        }
    }
}
```

(Sumber: Using ternary DAGs for spelling correction - strchr.com)

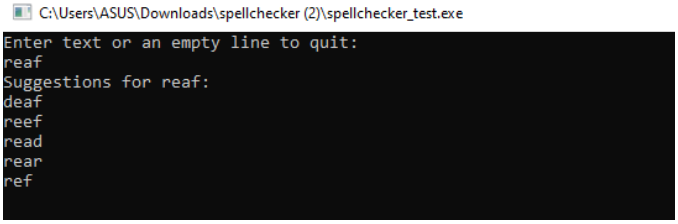
Algoritma diatas secara *straightfoward* mengecek apakah karakter selanjutnya sama dengan karakter sekarang, jika iya, secara rekursif kita cek, apakah memang ada kata demikian di kamus. [4]

C. Performansi dan Kelemahan

Saat performansi dari *Ternary DAG* dibandingkan dengan implementasi *pure* hash table saat mendeteksi kesalahan pengejaan dengan kamus sekitar 100 ribu kata, *Ternary DAG* memberikan waktu pengecekan lebih lama dibandingkan hash table dan memberikan waktu koreksi yang lebih cepat. Hal ini secara intuitif benar, karena *Ternary DAG* dibangun dengan Trie yang secara rata-rata memang lebih lambat dibandingkan hash

table saat melakukan pencarian. Namun, dengan *Ternary DAG* kita mendapatkan informasi mengenai kata-kata yang mungkin bisa menjadi koreksi dari kata yang salah. Kemudian, terlihat bahwa implementasi *Ternary DAG* ternyata memberikan memori yang lebih sedikit dibandingkan hash table, hal ini dikarenakan pada implementasinya penggunaan hash hanyalah digunakan untuk menampung suffix-suffix jika dibandingkah hash table yang menampung semua kamus.

Secara rata-rata, algoritma yang telah ada sudah berjalan dengan cepat, berikut hasilnya:



Namun perlu diperhatikan bahwa masih banyak kasus yang tidak terimplementasi pada program yang telah dibuat di[2] yakni kasus ketika *edit distance* dari suatu kata lebih dari 2, belum adanya support penambahan kata secara runtime, dsb.

IV. KESIMPULAN

Dapat disimpulkan bahwa Ternary DAG lebih efisien secara memori digunakan untuk *spelling suggestion* dibandingkan struktur data trie dan lebih efisien secara waktu dibandingkan struktur data BST. Jika permasalahannya hanya mencakup ke *spelling checking* struktur data *hash table* lebih cocok digunakan karena kompleksitas waktunya yang kecil.

V. UCAPAN TERIMA KASIH

Terima kasih kepada Tuhan Yang Maha Esa berkat limpahan rahmat, kasih, dan karunia-Nya, penulis dapat menyelesaikan makalah ini dengan baik. Tidak lupa juga, penulis berterima kasih kepada Bapak Rinaldi Munir selaku dosen pengajar Matematika Diskrit K1 yang telahm menyediakan waktu dan tenaganya untuk menyampaikan ilmu terkait Matematika Diskrit kepada mahasiswa K1. Penulis juga berterima kasih kepada keluarga, kerabat, dan pihak-pihak lainnya atas segala doa dan dukungan yang telah diberikan.

REFERENCES

[1] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf .
[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf
[3] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag2.pdf
[4] Using ternary DAGs for spelling correction - strchr.com
[5] Ternary Search Trees | Dr Dobb's

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 9 Desember 2020

Kadek Surya Mahardika dan 13519165