# Penggunaan *Ternary Search Tree* dalam Melakukan *Autocomplete*

Timotius Kevin Levandi | 13510056
*Program Studi Teknik Informatika*
*Sekolah Teknik Elektro dan Informatika*
*Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia*
*kevin_levandi@students.itb.ac.id*

***Abstrak*—Autocomplete *atau kadang disebut juga* Autosuggest *merupakan fitur yang melibatkan prediksi terhadap pengetikan sebuah kata atau frase oleh pengguna tanpa harus mengetiknya secara lengkap berdasarkan huruf atau frase awal yang telah dimasukkan oleh pengguna tersebut. Fitur ini sering terdapat dalam sebuah* search engine, web browser, e-mail program, source code editor, database query tool, word processors, *dan* command line interpreter. *Fitur ini sangat mempermudah pengguna, misalnya untuk mempercepat pengetikan terutama bila menyangkut penulisan fungsi yang komponen hurufnya panjang dan* case sensitive *pada suatu* source code editor, *atau untuk mempercepat pencarian alamat* email *tujuan dalam* email program *tanpa harus menghapal seluruh alamat* email *yang kebanyakan panjang-panjang.***

***Makalah ini akan membahas pengaplikasian* Ternary Search Tree *(TST) sebagai salah satu metode yang cukup dianggap efektif untuk melakukan* autocomplete *secara umum terhadap masukan teks oleh pengguna.***

***Kata kunci*—autocomplete, Ternary Search Tree (TST).**

## I. Pendahuluan

Sejak lebih dari beberapa dekade yang lalu, manusia sudah banyak mendapat kemudahan dalam melakukan aktivitas hidup kita melalui komputer dan jaringan internet. Sejak mulai keluar dan populernya *search engine*, kita tidak perlu lagi kerepotan mencari suatu situs sebagai untuk mencari informasi atau referensi. Dengan hanya mengetikkan beberapa kata kunci pada *search box* dan menekan Enter atau meng-click tombol Go. Semua situs yang memiliki keterkaitan dengan masukan kita ditampilkan mulai dari prioritas paling tinggi.

Kemudahan itu sekarang sudah lebih disempurnakan. Yang menarik sekarang adalah, selama pengetikan berlangsung, biarpun kata yang kita ketik belum selesai, sebuah daftar yang berisi saran mengenai kemungkinan kata bahkan frase atau kalimat yang akan kita ketik akan otomatis muncul. Selanjutnya kita dapat memilih langsung di antara daftar tersebut untuk memasukkan teks yang kita kehendaki dalam *search box* itu tanpa harus repot-repot menyelesaikan suatu pengetikan kalimat yang panjang.

Bila kita perhatikan, fenomena ini bukan hanya terdapat pada *search engine*. Hal ini juga sering terjadi di *source code editor* ketika kita melakukan pengkodean menggunakan suatu bahasa pemrograman dan menggunakan suatu *built-in function*, sebuah daftar yang berisi saran kemungkinan fungsi tersebut ditampilkan. Kita jadi tidak perlu lagi menyelesaikan pengetikan, terutama pada fungsi-fungsi yang tersusun oleh banyak huruf dan *case sensitive* yang biasanya banyak terdapat pada pemrograman berbasis obyek. Kita cukup memilih salah satu isi daftar tersebut sesuai kehendak kita sebelumnya. Kasus sama juga terjadi dalam pengisian alamat email tujuan saat akan mengirim *email*, atau saat kita memakai *word processor* dan hendak mengetikkan nama hari atau nama bulan. Hal ini, saudara-saudara, disebut dengan istilah *autocomplete* (penyempurnaan/penyelesaian otomatis).

*Autocomplete* merupakan fitur yang membantu kita dalam menyelesaikan pengetikan secara cepat dengan cara menyajikan pilihan kata, frase, atau kalimat yang mungkin merupakan kelanjutan dari apa yang sudah kita ketik. Bila pengetikan dilakukan lebih lanjut, maka ada kemungkinan beberapa pilihan yang telah disajikan tadi hilang dan digantikan dengan pilihan baru. Ini terjadi apabila kata, frase, atau kalimat dalam pilihan tersebut sudah tidak sesuai lagi dengan pengetikan beberapa karakter lanjutan yang baru saja dilakukan. Hal ini menjadi semacam pencarian sekaligus penyaringan daftar pilihan yang muncul dengan acuannya adalah posisi kursor dan kesamaan karakter yang telah ditulis sebelum kursor tersebut. Selanjutnya kita bisa memilih untuk memasukkan salah satu pilihan tersebut tanpa harus menyelesaikan pengetikan yang panjang.

Ada beberapa metode yang digunakan dalam merealisasikan *autocomplete* seperti menggunakan *Binary Search Tree* (BST), *Multiway Tree* (*Trie*), atau *Ternary Search Tree* (TST). Pada makalah ini kita akan memfokuskan pembahasan pada *Ternary Search Tree* yang bisa dibilang menggabungkan sifat-sifat *Binary Tree* dan *Multiway Tree*.

.

## II. Dasar Teori

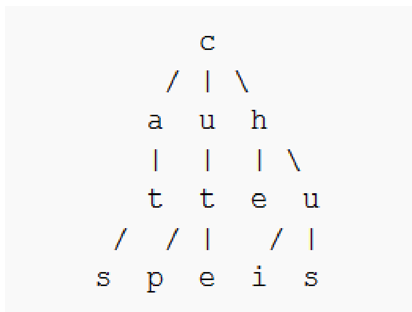
### II.A *Ternary Search Tree*

*Ternary Search Tree* (TST) atau *3-ary Search Tree*

adalah struktur data pohon dengan tiap simpulnya memiliki paling banyak tiga simpul anak, biasanya diistilahkan sebagai "kiri", "tengah" dan "kanan". Simpul yang memiliki anak disebut sebagai simpul orangtua dan simpul anak bisa memiliki penunjuk ke orangtuanya. Di luar pohon sering dibuat suatu penunjuk terhadap simpul "akar" (leluhur semua simpul) bila memang ada. Simpul manapun dalam struktur data bisa diakses dengan memulai dari simpul akar dan secara berulang-ulang mengikuti penunjuk ke salah satu anak kiri, tengah atau kanan.

TST merupakan suatu struktur data *Trie* di mana simpul anak suatu struktur data *Trie* standar disusun seperti pada *Binary Search Tree* (BST). Pencarian sebuah string pada TST terdiri atas sederet pencarian biner, satu untuk tiap karakter dalam suatu string. Karakter saat ini dalam suatu string dibandingkan dengan karakter pada simpul saat ini. Peraturannya demikian:

- Jika lebih kecil, pencarian berlanjut ke simpul anak sebelah kiri.
- Jika lebih besar, pencarian berlanjut ke simpul anak sebelah kanan.
- Jika sama, pencarian berlanjut ke simpul anak tengah dan perbandingan dilanjutkan untuk karakter selanjutnya dalam string tersebut.




***Gambar 1***: *Contoh TST dengan string "as", "at", "cup", "cute", "he", "i", dan "us"*

Untuk menghitung jumlah simpul pada *ternary tree* dapat diketahui sebagai berikut:
Misalkan *h* adalah tinggi *ternary tree* dan *M(h)* adalah jumlah maksimum simpul pada *ternary tree* dengan ketinggian *h*, maka:

$$M(h) = 1 + 3 + 9 + ... + 3^h = \sum_{i=0}^{h} 3^i$$

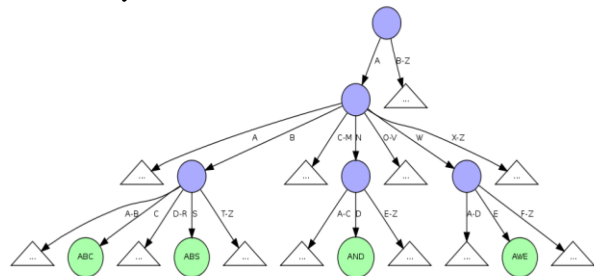
atau

$$M(h) = \frac{3^h - 1}{2}$$

### II.B *Multiway Tree* dan *Binary Search Tree*

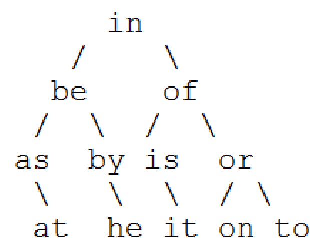
Selanjutnya karena istilah *Multiway Tree* (*Trie*) dan *Binary Search Tree* (BST) akan banyak disebut sebagai pembanding TST, maka berikut diberikan secara singkat definisi dari masing-masing struktur pohon tersebut.

*Multiway Tree* (*Trie*) adalah suatu pohon n-ary, yaitu pohon berakar yang setiap simpul cabangnya mempunyai paling banyak n buah anak. Misalnya untuk masukan data string standar dengan anggota himpunannya hanya huruf alfabet dari A sampai Z, maka pencarian simulai dari simpul akar yang mewakili string dengan panjang nol, yang kemudian memiliki 26 anak, di mana setiap anak memiliki 26 anak, dan seterusnya. Sistem ini memakan ruang sangat banyak untuk setiap penambahan ketinggian pohonnya, yaitu untuk setiap perbandingan karakter berikutnya (bila ada) dalam string tersebut. Hal ini bukan sesuatu yang jarang terjadi, karena suatu string pastinya memiliki banyak karakter.




***Gambar 2***: *Contoh* Multiway Tree, *setiap simpul memiliki 26 anak lagi yang merupakan karakter A sampai Z*

*Binary Search Tree* (BST) mirip seperti *Multiway Tree* dengan setiap simpul cabangnya mempunyai paling banyak dua buah anak. Semua simpul pada anak sebelah kiri akan memiliki nilai lebih kecil, sedangkan semua simpul pada anak sebelah kanan memiliki nilai lebih besar untuk semua simpul. Pencarian dimulai dari simpul akar. Pada perbandingan string menggunakan metode BST, kita mungkin harus langsung membandingkan banyak karakter pada suatu simpul. Setiap simpul mengandung sebuah string secara keseluruhan.




***Gambar 3***: *Contoh BST dengan 12 kata berhuruf dua*

## III. Analisis terhadap Kemangkusan *Ternary Search Tree* dalam Aplikasi *Autocomplete*

Sistem TST merupakan gabungan dari *Multiway Tree* dan BST, serta disinyalir memiliki kemangkusan yang lebih baik bila dibandingkan kesua sistem lainnya itu bila berdiri sendiri.

### III.A Perbandingan *Ternary Search Tree* dan *Binary Search Tree*

Seperti yang sudah dikatakan sebelumnya, BST dalam penyusunan string pada setiap simpulnya diharuskan sesuai aturan yaitu simpul anak kiri memiliki nilai lebih kecil dari orangtua yang sekarang, dan simpul sebelah kanan memiliki nilai yang lebih besar. Ini menyebabkan proses memasukkan suatu kata kunci baru pada suatu saat ke dalam pohon (memasukkan simpul baru) harus benar-benar memperhatikan keterurutan nilai string pada simpul-simpul dalam pohon, oleh karena itu perlu menggunakan pengurutan (*sorting*). Sedangkan pada TST pengurutan tidak sesensitif pada BST.

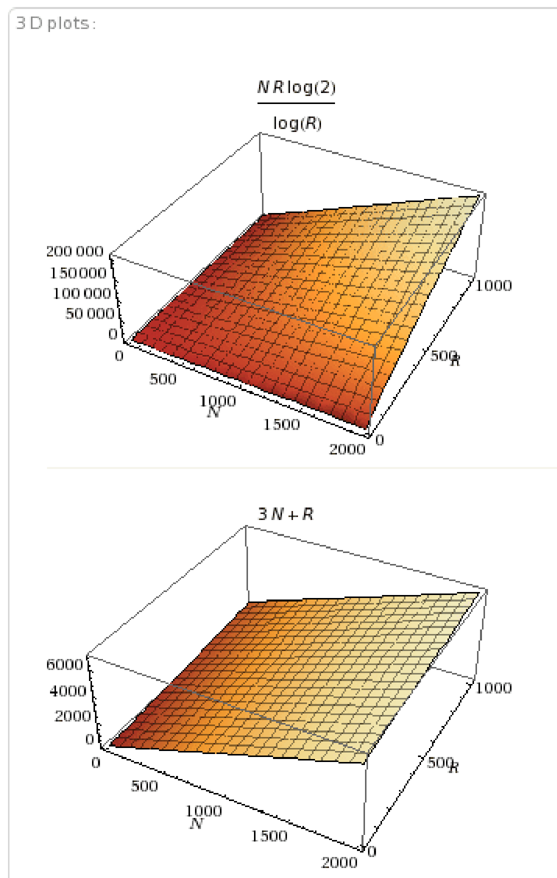
TST memakan lebih banyak alokasi ruang daripada BST, dan menggunakan algoritma yang lebih rumit karena banyaknya simpul yang dimilikinya. Walaupun begitu keuntungan-keuntungan sebagai berikut didapat:

- Simpul-simpul TST lebih sederhana karena berisi karakter, sedangkan simpul BST berisi suatu data string.
- Proses perbandingan pada TST menjadi lebih sederhana. Alasannya sama seperti sebelumnya. Membandingkan satu karakter tentu lebih mudah daripada membandingkan deretan karakter dalam data string
- Redundansi prefiks secara otomatis dihilangkan. Prefiks bisa diibaratkan sebagai kumpulan orangtua-orangtua terdahulu, yaitu dalam pengetikan adalah huruf-huruf di sebelah kiri kursor. Pada contoh gambar BST sebelumnya bisa kita lihat bahwa pendefinisian potongan kata “foo” pada string “foo” dan “foobar” dilakukan lebih dari satu kali. Ini disebut redundansi. Pada TST redundansi dihilangkan karena untuk hasil pencarian “foobar” akan diteruskan dari penelusuran “foo” dengan mengambil cabang ke simpul kiri untuk mencari huruf ‘b’.
- Tidak perlu dereferensi ganda. Mirip seperti penjelasan redundansi di atas, hanya saja dereferensi adalah mengenai penunjukan terhadap simpul orangtua, yaitu dalam pengetikan adalah bila kita menghapus huruf-huruf yang sudah kita ketik. Misalnya pada BST yang mengandung string “resep”, “resep kue”, dan “resep masakan”. Dereferensi “resep masakan” ke “resep” harus melewati “resep kue” dulu sebagai orangtua “resep masakan” (“resep masakan” adalah simpul anak sebelah kanan “resep kue”). Pada TST hal ini tidak perlu terjadi.

Dari sini bisa dilihat bahwa TST menawarkan performa yang lebih baik daripada BST walaupun cakupan alokasi ruang yang dibutuhkan lebih banyak.

### III.B Perbandingan *Ternary Search Tree* dan *Multiway Tree*

Bila dibandingkan dengan sistem *Multiway Tree*, sistem TST membantu kita terhindar dari kebutuhan ruang yang tidak perlu. Bila *Multiway Tree* memiliki sekitar $\frac{NR}{{}^{2}\log(R)}$ penunjuk dan TST memiliki sekitar 3N + R penunjuk. Ketika nilai R kecil *Multiway Tree* dan TST memiliki jumlah pointer yang mirip. Ketika semakin besar, jumlah pointer pada *Multiway Tree* meledak dibandingkan dengan TST. Karena itu perumusan TST akan menjadi jauh lebih mangkus dalam hal alokasi ruang dalam kasus terburuk dibandingkan perumusan *MultiwayTree*.




***Gambar 4**: 3D Plot yang menunjukkan hubungan* Multiway Tree *vs.* TST *dalam hal banyaknya alokasi ruang yang diperlukan*

Harga yang harus dibayar untuk kemangkusan ini adalah bila *Multiway Tree* kita harus mengunjungi (traverse) paling banyak sejumlah panjang kata yang dicari, pada TST kita melakukan sampai tiga kalinya paling banyak pada kasus terburuk.

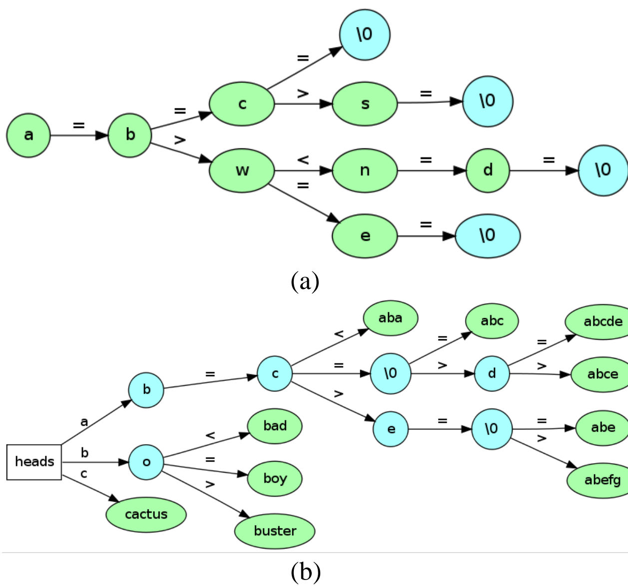
Namun kasus seperti ini jarang terjadi dan rata-rata bisa ditangani lewat beberapa peningkatan yang diberlakukan pada struktur dasarnya.

### III.C Peningkatan Kemangkusan *Ternary Search Tree*

Kebutuhan TST yang lumayan besar dalam hal alokasi ruang dapat ditekan sehingga kemangkusannya dapat ditingkatkan dengan beberapa cara sebagai berikut.

Cara pertama melibatkan pemampatan simpul daun yang ada. Simpul daun dalam sebuah pohon adalah simpul

berderajat nol (tidak mempunyai anak). Dengan demikian alih-alih membiarkan rantai panjang simpul-simpul pada simpul daun, kita dapat memampatkannya menjadi satu simpul saja sehingga pemeriksaan akhir terhadap string yang ditemukan dapat dilakukan dengan lebih mangkus.




(a)

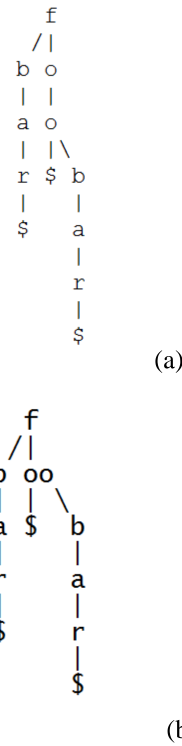
(b)

***Gambar 5***: *(a) TST dengan struktur dasar awal; (b) TST setelah dilakukan peningkatan dengan melakukan pemampatan pada simpul daunnya*

Cara kedua menggabungkan kebaikan *Multiway Tree* dan TST. Caranya adalah dengan mengganti struktur akar menjadi seperti *Multiway Tree* dengan simpul sejumlah R. Kemudian sisa pohonnya menggunakan TST dengan pemampatan pada simpul daun. Dalam praktiknya peningkatan ini menghasilkan penambahan kecepatan yang sangat signifikan. Teorinya menurut Sedgewick, peningkatan ini memotong jumlah perbandingan yang diperlukan sampai setengahnya.

Cara ketiga dilakukan hampir sama dengan cara pertama, yaitu melibatkan pemampatan simpul. Namun alih-alih memampatkan simpul daun, simpul dalam yang dimampatkan. Simpul dalam adalah simpul yang mempunyai anak. Cara ini dilakukan untuk menghemat jumlah percabangan dan perbandingan karakter bila sudah jelas terdapat batasan kemungkinan untuk suatu rantai karakter yang menghasilkan suatu kata kunci. Dengan demikian sedikit mirip BST, kita melakukan perbandingan string pada beberapa simpul dalam.

Untuk lebih jelasnya, misalnya terdapat *Ternary Tree* sebagai berikut.



(a)

(b)

***Gambar 6***: ***(a)*** *TST dengan struktur data awal; (b) TST setelah dilakukan peningkatan dengan melakukan pemampatan pada simpul dalamnya*

Dari gambar kita bisa melihat adanya rantai panjang penunjuk dari ‘o’ yang tidak punya cabang ke karakter lain selain ke ‘o’ selanjutnya. Untuk mempermudah pengertian, anggap karakter ‘o’ yang ditunjuk f sebagai ‘$o_1$’ dan karakter ‘o’ yang ditunjuk ‘$o_1$’ sebagai ‘$o_2$’

Alih alih membiarkan rantai seperti ini menambah alokasi ruang, kita dapat menggabungkannya menjadi string “oo” dalam satu simpul saja. Pemampatan ini tentunya dilakukan bila memang tidak ada string kata kunci yang menggunakan karakter lain setelah karakter ‘$o_1$’ selain karakter ‘o’. Sebagai contoh, misalkan tidak ada string kata kunci seperti “foe”, “foam”, atau “fog”; yang ada hanya “foobar”, “footage”, atau “fool”. Dengan demikian keputusan untuk menjadikan ‘o’ dan ‘o’ satu string “oo” dalam sebuah simpul adalah tepat, dan akan meningkatkan kemangkusan TST dalam *autocomplete*.

Cara keempat untuk meningkatan kemangkusan pencarian dalam TST, kita tidak perlu misalkan, menggunakan penunjuk ke semua karakter alfabet. Karakter yang tidak terdaftar atau jarang sekali digunakan setelah karakter tertentu dalam suatu string tidak perlu dimasukkan sebagai simpul anaknya.

Sebagai contoh, karakter ‘p’ dapat menunjuk karakter ‘r’ yang kemudian dapat menunjuk beberapa kemungkinan karakter lain dalam suatu *search engine* yang akan mengarah pada beberapa kemungkinan string kata kunci seperti “premonition”, “profile”, atau “premier”. Namun tidak ada string kata kunci yang akan dihasilkan ataupun diteruskan sedemikian bila karakter

yang ditunjuk setelah ‘r’ adalah ‘m’. Oleh karena itu di antara simpul-simpul kiri ‘r’ tidak perlu dialokasikan suatu simpul untuk menampung karakter ‘m’.

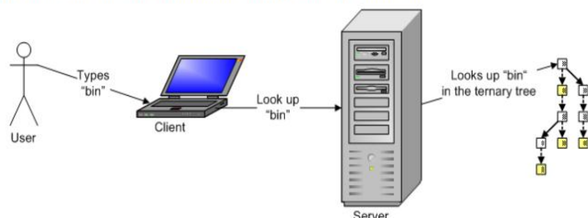
Cara kelima adalah dengan melakukan analisis terhadap kata-kata kunci yang lebih banyak digunakan oleh orang-orang dibandingkan kata-kata kunci lainnya dengan prefiks yang sama. Kemudian kata-kata kunci yang lebih populer ini diletakkan dan diatur sedemikian rupa sehingga sebisa mungkin lebih dekat ke akar. Ini akan menyebabkan kemungkinan pencarian dan perbandingan dapat dilakukan lebih cepat tanpa harus melewati banyak percabangan-percabangan yang kurang perlu akibat salah diletakkannya kata kunci yang jarang diakses.

Selanjutnya, untuk meningkatkan kecepatan ditampilkannya hasil pencarian. Kita bisa membatasi berapa pilihan hasil tampilan yang sementara itu ditampilkan. Bila jumlah pilihan yang ingin ditampilkan banyak, maka akan perlu waktu lebih lama untuk mencapai daun-daun berisi sring kata kunci dengan prefiks seperti yang dimasukkan pengguna. Namun terlalu sedikit pilihan juga tidak akan membantu pengguna. Oleh karena itu hendaknya ditetapkan jumlah tampilan pilihan dalam daftar yang cukup beralasan.

## IV. *TERNARY SEARCH TREE* DALAM PELADEN

Di dalam jaringan (*web*), sejumlah pekerjaan melakukan *autocomplete* dilakukan oleh peladen (*server*). Kadang kumpulan dari kemungkinan-kemungkinan penyempurnaan dari kata yang dimasukkan sangatlah banyak, jadi bukanlah ide bagus untuk melakukan pengunduhan semuanya kepada klien. Alih-alih *ternary tree* disimpan di dalam peladen dan klien akan mengirim prefiks permintaan kepada peladen.

The client will send a query for words starting with “bin” to the server:




And the server responds with a list of possible words:

***Gambar 7*: *Ilustrasi menunjukkan bagaimana* autocomplete *dilakukan oleh peladen berdasarkan masukan prefiks permintaan dari pengguna***

## V. IMPLEMENTASI *TERNARY SEARCH TREE*

Terdapat banyak implementasi TST dalam berbagai bahasa pemrograman. Berikut adalah contoh sederhana implementasi TST dalam C#.

```
public class TernaryTree
{
    private Node m_root = null;


   private void Add(string s, int pos, ref Node node)
   {
       if (node == null) { node = new Node(s[pos], false); }

       if (s[pos] < node.m_char) { Add(s, pos, ref node.m_left); }
       else if (s[pos] > node.m_char) { Add(s, pos, ref node.m_right); }
       else
       {
           if (pos + 1 == s.Length) { node.m_wordEnd = true; }
           else { Add(s, pos + 1, ref node.m_center); }
       }
   }

   public void Add(string s)
   {
       if (s == null || s == "") throw new ArgumentException();

       Add(s, 0, ref m_root);
   }

   public bool Contains(string s)
   {
       if (s == null || s == "") throw new ArgumentException();

       int pos = 0;
       Node node = m_root;
       while (node != null)
       {
           int cmp = s[pos] - node.m_char;
           if (s[pos] < node.m_char) { node = node.m_left; }
           else if (s[pos] > node.m_char) { node = node.m_right; }
           else
           {
               if (++pos == s.Length) return node.m_wordEnd;
               node = node.m_center;
           }
       }

       return false;
   }
}

class Node
{
    internal char m_char;
    internal Node m_left, m_center, m_right;
    internal bool m_wordEnd;

    public Node(char ch, bool wordEnd)
    {
        m_char = ch;
        m_wordEnd = wordEnd;
    }
}
```

## VI. KESIMPULAN

Dari pembahasan di atas dapat disimpulkan hal-hal sebagai berikut.

- Fitur *autocomplete* dapat dijalankan dengan baik menggunakan pencarian dalam struktur data pohon.
- *Ternary Search Tree* merupakan sistem struktur data pohon termangkus yang dapat digunakan untuk melakukan *autocomplete* yang menggabungkan kebaikan *Binary Search Tree* dan *Multiway Tree*.
- *Ternary Search Tree* menggunakan alokasi ruang yang lebih banyak daripada *Binary Search Tree*, namun dengan sistem perbandingan yang lebih sederhana.
- *Ternary Search Tree* menggunakan alokasi ruang yang lebih sedikit daripada *Multiway Tree*, namun dengan sistem perbandingan yang lebih rumit.

## VII. REFERENSI


[1] Munir, Rinaldi. *Struktur Diskrit*, edisi keempat. 2008. Bandung: Penerbit ITB.

[2] ___. *Autocomplete*. From http://en.wikipedia.org/wiki/Autocomplete, 10 Desember 2011.

[3] ___. *Ternary search tree*. From http://en.wikipedia.org/wiki/Ternary_search_tree, 10 Desember 2011.

[4] ___. *Ternary Tree*. From http://en.goldenmap.com/Ternary_tree, 10 Desember 2011.

[5] Adam Berger et al. (2007). *Ternary Search Tree*. From http://c2.com/cgi/wiki?TernarySearchTree, 10 Desember 2011.

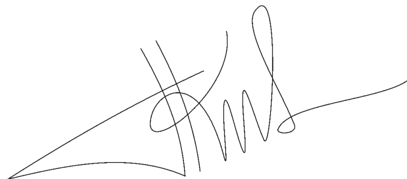
[6] Igor Ostrovsky (2009). *Efficient auto-complete with ternary search tree*. From http://igoro.com/archive/efficient-auto-complete-with-a-ternary-search-tree/, 11 Desember 2011.

[7] Tim Henderson (2011). *Ternary Search Tries for Fast Flexible String Search : Part 1*. From http://blog.hackthology.com/ternary-search-tries-for-fast-flexible-string#!/, 11 Desember 2011.


## 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, 11 Desember 2011



Timotius Kevin Levandi | 13510056