Implementasi Struktur Data *Map from String* Menggunakan Pohon *N-ary*

Turfa Auliarachman - 13515133
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
kingfalcon@students.itb.ac.id

***Absttrak—Banyak cara untuk mengimplementasikan struktur data* map. *Di sini disajikan alternatif cara mengimplementasikan struktur data* map *yang mudah ditulis programnya dan cukup cepat dalam menjalankan programnya, yaitu dengan memanfaatkan pohon* n-ary. *Semua primitif dari* map *disesuaikan dengan sifat-sifat dari pohon* n-ary.**

Kata kunci---*map*, pohon, pohon* n-ary*, struktur data.

I. PENDAHULUAN

Map adalah struktur data abstrak yang sangat penting. Sangat banyak kegunaan struktur data *map* ketika menulis program untuk keperluan apapun.

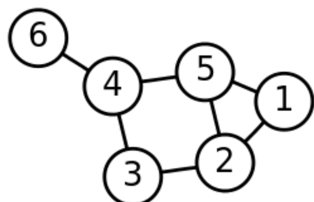
Namun, pembuatan struktur data *map* biasanya harus memilih antara dua hal, yaitu kemudahan memprogram atau kecepatan program.

Dalam makalah kali ini, akan disajikan alternatif implementasi struktur data *map* yang cukup mudah untuk ditulis programnya dan kecepatan programnya pun cukup cepat.

II. LANDASAN TEORI

A. *Graf*

Graf adalah kumpulan simpul dan sisi yang menghubungkan simpul-simpul tersebut. Secara formal, graf adalah pasangan himpunan (V, E), di mana V adalah himpunan dari simpul dan E adalah himpunan dari sisi yang menghubungkan simpul-simpul tersebut. E adalah *multiset*, jadi bisa saja ada elemen yang muncul lebih dari sekali. [1]



Bagan 1: Contoh Graf. Sumber: https://upload.wikimedia.org/wikipedia/commons/thumb/5/5b/6n-graf.svg/250px-6n-graf.svg.png

Dalam graf, sebuah jalan adalah rangkaian simpul dan sisi sebuah graf yang saling berselang-seling antara simpul dan sisi, yang diawali dan diakhir oleh sebuah simpul. Sebuah jalan dikatakan sebagai jejak jika setiap sisi maksimal dikunjungi sebanyak sekali. Sebuah jejak dikatakan lintasan jika setiap simpul maksimal dikunjungi sebanyak sekali, kecuali simpul awal dan simpul akhir, jika keduanya sama. Kasus tersebut dinamakan sebagai lintasan tertutup atau sirkuit. [1]

Dua simpul *u* dan *v* dikatakan terhubung jika ada jalan yang berawal dari *u* dan berakhir di *v*. Jika *u* dan *v* terhubung serta *v* dan *w* terhubung, maka *u* dan *w* pasti terhubung. Sebuah graf dikatakan terhubung jika semua pasangan simpulnya terhubung. [1]

Graf bisa berarah dan tidak berarah. Graf berarah maksudnya setiap sisi memiliki arah dan hanya menyambungkan dua simpul dalam satu arah. [2]

Graf juga bisa berbobot dan tidak berbobot. Dalam graf berbobot, setiap simpul atau setiap sisi memiliki bobotnya masing-masing yang tidak harus unik. [1]

Ada beberapa graf yang dianggap graf berproperti khusus. Salah satunya adalah graf sederhana. Graf sederhana adalah graf yang tidak berarah, tidak berbobot, tidak memiliki sisi yang menghubungkan simpul yang sama, dan tidak memiliki sisi yang muncul lebih dari sekali. Graf sederhana bisa terhubung maupun tidak. [3]

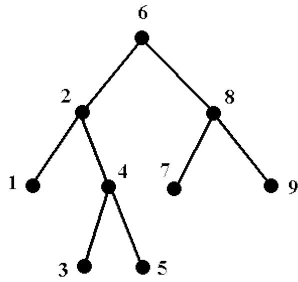
B. *Pohon*

Pohon adalah graf sederhana yang tidak berarah, terhubung, dan tidak memiliki sirkuit. Daru deskripsi tersebut, dapat disimpulkan bahwa pohon yang memiliki *n* buah simpul pasti memiliki $n – 1$ buah sisi. Jadi, sebuah graf terhubung yang memiliki *n* buah simpul dan $n – 1$ buah sisi pasti adalah pohon. [4]

Sangat banyak kegunaan dari pohon di berbagai bidang, termasuk dalam ilmu komputer, mengenumerasi hidrokarbon jenuh, pelajaran sirkuit elektrik, dan sebagainya. [5]

Pohon berakar adalah sebuah pohon yang salah satu dan tepat satu simpulnya dianggap sebagai akar. Setiap simpul yang bertetangga dengan akar tersebut disebut anak, dan akar tersebut disebut orang tua. Setiap simpul *v* yang bertetangga dengan sebuah simpul *u*, jika simpul *v* bukan orangtua dari simpul *u*, maka simpul *v* adalah anak dari simpul *u* dan simpul *u* merupakan orangtua dari simpul *v*.

Setiap simpul yang meliki jarak yang sama dengan akar disebut saling bersaudara. [4]



Bagan 2: Contoh Pohon Berakar. http://www-math.ucdenver.edu/~wcherowi/courses/m4408/gtln8p1.gif

Selain akar, anak, dan orang tua, ada beberapa istilah penting lain dalam pohon berakar yang tidak ada di graf, yaitu tinggi, kedalaman, dan upapohon. Tinggi dari sebuah pohon adalah jarak terjauh dari akar ke sebuah daun dari pohon tersebut. Kedalaman dari sebuah simpul adalah jarak dari simpul tersebut ke akar pohon. Upapohon adalah upagraf dari sebuah pohon yang mengandung sebuah anak dari sebuah simpul pohon dan semua keturunan anak tersebut. Sebuah upapohon merupakan pohon juga. [6]

Ada pohon berproperti khusus, di antaranya adalah pohon *n-ary*, pohon ekspresi [7], *heap*, dan pohon-b. [6]

C. *Pohon* N-*ary*

Pohon *n-ary* adalah pohon yang setiap setiap anak dari sebuah simpulnya bisa dianggap ada di indeks *0 – n-1*. Dengan kata lain, setiap simpul dalam pohon *n-ary* maksimal memiliki *n* buah anak. [8]

Ada beberapa pohon *n-ary* spesial yang memiliki nama tersendiri dan banyak digunakan dalam berbagai hal. Pohon *n-ary* dengan $n = 3$ biasa dinamakan pohon *ternary*. Pohon *n-ary* dengan $n = 2$ biasa dinamakan pohon biner. Sedangkan *list* adalah pohon *n-ary* dengan $n = 1$. [8]

D. Map

Map adalah struktur data abstrak berupa daftar pasangan kunci-nilai dengan kunci yang unik. Dengan kata lain, tidak boleh ada dua elemen yang memiliki kunci yang sama. [9]

Map biasa disebut juga larik asosiatif. Artinya, *map* adalah larik dengan kunci sebagai indeks. [9]

Ada beberapa primitf penting yang dimiliki struktur data *map*, yaitu *isEmpty(), get(key), put(key, value),* dan *remove(key)*. [9]

III. PEMBAHASAN

A. *Representasi Kunci* String *dalam Pohon*

Di bab II dijelaskan bahwa pohon *n-ary* memiliki kekhasan yaitu setiap anak dari sebuah simpul memiliki indeks dari *0 – n-1*. Hal ini sejalan dengan *string* yang memiliki batasan simbol.

Sebagai contoh, sebuah *string* yang merepresentasikan sebuah kata dalam bahasa Indonesia memiliki batasan simbol yaitu *a – z*, dengan asumsi semua kata yang akan direpresentasikan tidak mengandung karakter spesial seperti ‘ dan sebagainya.

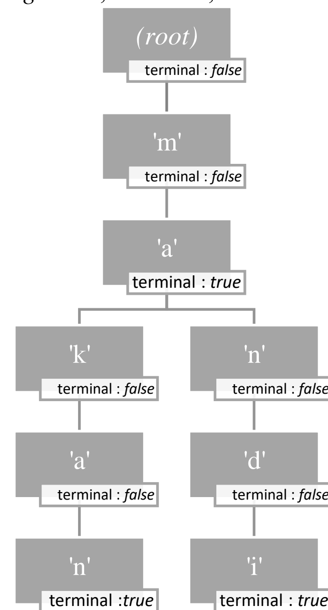
String bisa dikelompokkan berdasarkan kesamaan prefiksnya, yaitu huruf awalannya. Sebagai contoh, “makan” dan “mandi” dapat dikelompokkan dalam satu kelompok, yaitu kumpulan *string* yang memiliki prefiks “ma-“.

Dalam sebuah pohon berakar, hanya terdapat satu lintasan yang berawal dari akar dan berakhir di suatu simpul tertentu. Dalam kasus pohon *n-ary*, lintasan tersebut bisa disimbolkan dengan urutan indeks anak mana saja yang harus dilalui dari akar agar bisa sampai ke simpul tersebut.

Dari penjelasan di atas, untuk merepresentasikan string dalam pohon, bisa dibuat pohon *n-ary* yang setiap simpulnya melambangkan kumpulan *string* yang memiliki prefiks yang sama, dengan prefiksnya adalah urutan indeks lintasan dari akar ke simpul tersebut.

Untuk mengetahui apakah suatu prefiks juga merupakan sebuah *string* utuh, bisa dibuat satu elemen dalam struktur data. Katakanlah elemen itu bertipe *boolean* dan memiliki nama *terminal*. Elemen *terminal* ini akan bernilai *true* jika prefiks yang diwakili oleh simpul tersebut juga merupakan *string* utuh, dan akan bernilai *false* jika sebaliknya.

Berikut contoh pohon *n-ary* yang merepresentasikan kumpulan *string* “ma”, “makan”, dan “mandi”.



Bagan 3: Contoh Pohon N-ary *Representasi Kumpulan* String

Seperti pada gambar, sebuah *map* bisa dibedakan berdasarkan akar dari pohon *n-ary* yang dibuat. Oleh karena itu, *map* bisa direpresentasikan sebagai sebuah simpul, yaitu simpul akarnya. Atau sebaliknya, setiap simpul bisa dianggap sebagai sebuah *map*.

Dalam pembahasan implementasi struktur data *map from string* menggunakan pohon *n-ary* ini, akan dibuat implementasi *map from string* yang seumum mungkin. Oleh karena itu, akan ditetapkan batas simbol yang seluas

mungkin.

Dalam tipe data *string* bawaan bahasa C++, setiap karakter dalam sebuah *string* memiliki tipe data *char*. Tipe data *char* ini memuat sebuah karakter yang juga bisa dianggap sebuah bilangan antara *1 – 255*, yang merepresentasikan bilangan ASCII setiap karakter.

Dari penjelasan di atas, agar batas simbol yang ditentukan seluas mungkin, pohon *n-ary* yang dibuat harus memiliki *n = 256*. Memang ada indeks yang tidak terpakai di sini, yaitu indeks *0*.

Secara keseluruhan, sampai saat ini, struktur *map from string* yang sudah dirancang adalah sebagai berikut :

```
struct Map{
    bool terminal;
    Map * child[256];
};
```

Bagan 4: Struktur Data Map

Seperti yang terlihat dalam Bagan 3 dan Bagan 4, representasi prefiks setiap simpul tidak perlu disimpan, karena bisa diketahui ketika melakukan traversal menuju simpul tersebut.

B. Representasi Nilai dalam Pohon

Sejauh ini, pohon *n-ary* yang dibuat sudah bisa merepresentasikan daftar *string*. Daftar *string* yang direpresentasikan oleh pohon *n-ary* ini unik. Hal ini sejalan dengan kunci dalam struktur data *map*.

Satu hal mendasar lain dalam struktur data *map* selain kunci adalah pasangan nilainya. Nilai ini bisa memiliki tipe data apa saja. Setiap kunci pasti memiliki pasangan nilai masing-masing.

Dalam pohon *n-ary* yang sudah dirancang, setiap simpul mewakili tepat satu prefiks *string*. Tergantung nilai elemen *terminal*, prefiks *string* itu bisa merupakan *string* utuh ataupun tidak. Jika prefiks *string* itu juga merupakan *string* utuh, harus ada nilai yang berpasangan dengan *string* tersebut.

Dari penjelasan di atas, diketahui bahwa setiap simpul bisa merepresentasikan satu *string* utuh atau tidak sama sekali. Jika merepresentasikan satu *string* utuh, *string* tersebut, yang direpresentasikan oleh simpul, harus memiliki sebuah pasangan nilai. Oleh karena itu, pasangan nilai dari *string* utuh tersebut bisa disimpan dalam simpul yang merepresentasikannya.

Anggap nilai dalam *map* yang dibuat bertipe data *integer*, sekarang struktur data *map* yang dibuat menjadi seperti berikut.

```
typedef int tipenilai;

struct Map{
    bool terminal;
    Map * child[256];

    tipenilai nilai;
};
```

Bagan 5: Struktur Data Map

Sampai saat ini, pohon *n-ary* yang dibuat sudah lengkap dan sudah bisa merepresentasikan *map*.

C. Konstruktor dan Destruktor

Untuk membuat sebuah *map* baru, cukup membuat sebuah simpul sebagai akar seperti yang digambarkan dalam Bagan 3.

Untuk membuat sebuah simpul, cukup menginisialisasi elemen *terminal* dan *child* dari simpul tersebut. Elemen *nilai* tidak perlu diinisialisasi karena tidak akan diakses kecuali ketika *terminal* bernilai *true*, dan ketika mengubah elemen *terminal* menjadi *true*, elemen *nilai* pasti akan diisi. Elemen *terminal* tentu harus diinisialisasi dengan nilai *false*. Untuk elemen *child*, cukup inisialisasi semuanya dengan *null*, karena anak dari sebuah simpul baru akan dibuat ketika memang ada *string* yang berprefiks sesuai anak tersebut.

Secara lengkap, konstruktor dari sebuah *map* adalah sebagai berikut.

```
Map newMap(){
    Map baru;
    baru.terminal = false;
    for(int i = 0; i < 256; i++){
        baru.child[i] = NULL;
    }

    return baru;
}
```

Bagan 6: Konstruktor Map

Untuk menghapus sebuah *map*, tidak cukup hanya mendealokasi akar dari sebuah *map*. Semua anak dari akar tersebut juga harus didealokasi, karena tidak akan bisa dicapai lagi. Kode lengkap destruktor dari sebuah *map* adalah sebagai berikut.

```
void deleteMap(Map * M){
    for(int i = 0; i < 256; i++){
        if (M->child[i] != NULL){
            deleteMap(M->child[i]);
        }
    }

    free(M);
}
```

Bagan 7: Destruktor Map

D. Primitif IsEmpty()

Sesuai yang sudah dibahas dalam bab II. Landasan Teori, salah satu primitif dasar untuk struktur data *map* adalah fungsi *isEmpty()*. Pada dasarnya, fungsi *isEmpty()* mengecek apakah sebuah *map* kosong atau tidak.

Tanda dari adanya elemen dalam *map* adalah adanya simpul dengan *terminal* bernilai *true*. Maka, cara mengecek apakah sebuah *map* kosong atau tidak adalah dengan mengecek apakah di *map* tersebut ada simpul dengan *terminal true* atau tidak. Jika ada, fungsi mengembalikan *false*, sedangkan kebalikannya, jika tidak ada, fungsi akan mengembalikan *true*.

Berikut adalah kode lengkap dari fungsi *isEmpty()* yang

baru dijelaskan.

```
bool isEmpty(Map M){
    bool ret = !M.terminal;

    for(int i = 0; i < 256; i++){
        if (M.child[i] != NULL){
            ret &=
isEmpty(*(M.child[i]));
        }
    }

    return ret;
}
```

Bagan 8: Fungsi Primitif isEmpty()

E. Fungsi isExist(key)

Fungsi *isExist(key)* akan mengembalikan nilai *true* jika ada elemen *map* dengan kunci *key*, dan akan mengembalikan nilai *false* jika tidak ada.

Meski tidak disebutkan sebagai primitif dasar dari sebuah *map*, fungsi *isExist(key)* sangat penting untuk mengetahui apakah sudah ada elemen dari *map* dengan kunci *key*. Hal ini sangat penting karena salah satu sifat *map* adalah setiap elemennya memiliki kunci yang unik.

Fungsi *isExist(key)* bisa didefinisikan secara rekursif.

Ada dua basis dalam fungsi *isExist(key)*, yaitu ketika *key* merupakan *string* kosong atau ketika tidak ada anak dari simpul yang merepresentasikan prefiks dari *key*. Ketika tidak ada anak, jelas bahwa fungsi akan mengembalikan nilai *false*. Sedangkan ketika *key* merupakan *string* kosong, nilai yang dikembalikan akan sama dengan *terminal*, karena *terminal* melambangkan adanya *string* yang direpresentasikan simpul tersebut.

Rekurens dari fungsi *isExist(key)* ini adalah akan mengakses *child* dengan indeks kode ASCII karakter pertama *key*. Lalu karakter pertama *key* dihapus.

Kode lengkap dari fungsi *isExist(key)* ini adalah sebagai berikut.

```
bool isExist(Map M, std::string key){
    if (key == ""){
        return M.terminal;
    }
    else if (M.child[key[0]] == NULL){
        return false;
    }
    else{
        return
isExist(*(M.child[key[0]]),
key.substr(1));
    }
}
```

Bagan 9: Fungsi isExist(key)

F. Primitif get(key)

Primitif *get(key)* adalah fungsi untuk mencari pasangan nilai dari sebuah kunci yang diberikan, di mana kunci tersebut adalah *key*. Dalam menggunakan fungsi *get(key)*, sudah diasumsikan bahwa pasti ada elemen dalam *map* dengan kunci *key*.

Seperti halnya fungsi *isExist(key)*, fungsi *get(key)* dapat diimplementasikan secara rekursif. Basis dan rekurensnya pun cukup mirip dengan fungsi *isExist(key)*.

Basis dari fungsi *get(key)* adalah ketika *key* merupakan *string* kosong. Artinya, simpul yang sedang diproses merepresentasikan kunci yang dicari. Oleh karena itu, nilai yang dikembalikan adalah elemen *nilai*.

Rekurens dari fungsi *get(key)* ini sama dengan fungsi *isExist(key)*, yaitu akan mengakses *child* dengan indeks kode ASCII karakter pertama *key*. Lalu karakter pertama *key* dihapus.

Kode lengkap dari fungsi primitif *get(key)* adalah sebagai berikut.

```
tipenilai get(Map M, std::string key){
    if (key == ""){
        return M.nilai;
    }
    else{
        return get(*(M.child[key[0]]),
key.substr(1));
    }
}
```

Bagan 10: Fungsi Primitif get(key)

G. Primitif put(key, value)

Seperti namanya, primitif *put(key, value)* menambahkan pasangan kunci dan nilai ke dalam *map*, dengan kunci *key* dan nilai *value*. Jika sudah ada pasangan kunci dan nilai tersebut, nilai yang lama ditimpa dengan *value*.

Prosedur primitif *put(key, value)* dapat diimplementasikan sebagai prosedur rekursif.

Basisnya adalah ketika *key* merupakan *string* kosong, yang melambangkan simpul yang sedang diproses merepresentasikan *string key*. Saat ini, *terminal* diubah menjadi *true* dan *value* dimasukkan ke elemen *nilai.*

Rekurensnya adalah mengakses anak dari simpul sekarang dengan *key* yang sudah dihapus karakter pertamanya. Jika anaknya belum ada, dibuat dulu anak tersebut. Oleh karena itu, dibutuhkan juga prosedur *newChild* yang membuat anak dari suatu simpul.

Kode lengkap dari primitif *put(key, value)* ini adalah sebagai berikut.

```
void put(Map * M, std::string key,
tipenilai value){
    if (key == ""){
        M->terminal = true;
        M->nilai = value;
    }
    else {
        if (M->child[key[0]] == NULL){
            newChild(&(M-
>child[key[0]]));
        }

        put(M->child[key[0]],
key.substr(1), value);
    }
}
```

Bagan 11: Prosedur Primitif put(key, value)

H. Primitif remove(key)

Kebalikan dengan primitif *put(key, value)*, prosedur

primitif *remove(key)* akan menghapus elemen *map* dengan kunci *key* jika ada.

Seperti fungsi-fungsi dan prosedur sebelumya. prosedur primitif *remove(key)* ini juga bisa diimplementasikan dengan pendekatan rekursif.

Ada dua basis dalam prosedur *remove(key)*, yaitu ketika *key* merupakan *string* kosong atau ketika tidak ada anak dari simpul yang merepresentasikan prefiks dari *key*. Ketika tidak ada anak, prosedur akan langsung berhenti, karena tidak ada yang perlu dihapus. Sedangkan ketika *key* merupakan *string* kosong, yang harus dilakukan adalah mengubah nilai *terminal* menjadi *false*.

Kode lengkap dari prosedur primitif *remove(key)* adalah sebagai berikut.

```
void remove(Map * M, std::string key){
    if (key == ""){
        M->terminal = false;
    }
    else if (M->child[key[0]] != NULL){
        remove(M->child[key[0]],
key.substr(1));
    }
}
```

Bagan 12: Prosedur Primitif remove(key)

I. Analisis Kompleksitas

Jika banyaknya elemen adalah *N* dan panjang *string* yang akan diproses adalah *L*, maka kompleksitas algoritma dari *isEmpty()* adalah *O(N)* serta *isExist(key), get(key), put(key, value),* dan *remove(key)* adalah *O(L)*.

IV. KESIMPULAN

Pohon *n-ary* bisa digunakan untuk mengimplementasikan *map from string* dengan penulisan program cukup mudah dan kecepatan program cukup cepat.

Ada beberapa pengembangan yang dapat dilakukan dari makalah ini. Salah satunya, teknik ini dapat dicoba untuk mengimplementasikan *map* dengan kunci bertipe bilangan. Hal ini diharapkan dapat menurunkan kompleksitas yang awalnya dalam logaritmik berbasis dua menjadi logaritmik berbasis 10.

Selain itu, dapat disimpan banyaknya elemen dari *map* agar primitif *isEmpty()* menjadi berkompleksitas *O(1).*

Penulis juga menduga teknik ini bisa digunakan untuk melakukan pengurutan atau membuat stuktur data yang mempertahankan keterurutan.

V. APPENDIKS

```
#ifndef __MAP_H
#define __MAP_H

typedef int tipenilai;

struct Map{
    bool terminal;
    Map * child[256];

    tipenilai nilai;
};

Map newMap();

void deleteMap(Map * M);

bool isEmpty(Map M);

bool isExist(Map M, std::string key);

tipenilai get(Map M, std::string key);


void newChild(Node ** C);
void put(Map * M, std::string key,
tipenilai value);

void remove(Map * M, std::string key);
#endif
```

Bagan 13: Kode Lengkap Map.h

```
#include "Map.h"
#include <stdlib.h>
#include <string>

Map newMap(){
    Map baru;
    baru.terminal = false;
    for(int i = 0; i < 256; i++){
        baru.child[i] = NULL;
    }

    return baru;
}

void deleteMap(Map * M){
    for(int i = 0; i < 256; i++){
        if (M->child[i] != NULL){
            deleteMap(M->child[i]);
        }
    }

    free(M);
}

bool isEmpty(Map M){
    bool ret = !M.terminal;

    for(int i = 0; i < 256; i++){
        if (M.child[i] != NULL){
            ret &=
isEmpty(*(M.child[i]));
        }
    }

    return ret;
}
```

```
bool isExist(Map M, std::string key){
    if (key == ""){
        return M.terminal;
    }
    else if (M.child[key[0]] == NULL){
        return false;
    }
    else{
        return
isExist(*(M.child[key[0]]),
key.substr(1));
    }
}

tipenilai get(Map M, std::string key){
    if (key == ""){
        return M.nilai;
    }
    else{
        return get(*(M.child[key[0]]),
key.substr(1));
    }
}


void newChild(Map ** C){
    *C = (Map*) malloc(sizeof(Map));
    **C = newMap();
}


void put(Map * M, std::string key,
tipenilai value){
    if (key == ""){
        M->terminal = true;
        M->nilai = value;
    }
    else {
        if (M->child[key[0]] == NULL){
            newChild(&(M-
>child[key[0]]));
        }

        put(M->child[key[0]],
key.substr(1), value);
    }
}

void remove(Map * M, std::string key){
    if (key == ""){
        M->terminal = false;
    }
    else if (M->child[key[0]] != NULL){
        remove(M->child[key[0]],
key.substr(1));
    }
}
```

Bagan 14: Kode Lengkap Map.cpp

VI. UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada semua pihak yang telah membantu kelancaran pembuatan makalah ini.

VII. DAFTAR PUSTAKA

[1] K. Ruohonen, *Graph Theory.* Dipublikasikan secara online, bab 1-2.

[2] http://world.mathigon.org/Graph_Theory, diakses tanggal 7 Desember 2016.

[3] E. W. Weisstein, *Simple Graph.* Dari *MathWorld--A Wolfram Web Resource.* Diambil dari http://mathworld.wolfram.com/SimpleGraph.html, diakses tanggal 7 Desember 2016.

[4] E. W. Weisstein, *Tree.* Dari *MathWorld--A Wolfram Web Resource.* Diambil dari http://mathworld.wolfram.com/Tree.html, diakses tanggal 7 Desember 2016.

[5] F. Harary, *Graph Theory*. Reading, MA: Addison-Wesley, 1994, pp 35.

[6] http://pages.cs.wisc.edu/~vernon/cs367/notes/8.TREES.html, diakses pada tanggal 7 Desember 2016.

[7] https://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/trees/lecture.html, diakses pada tanggal 7 Desember 2016.

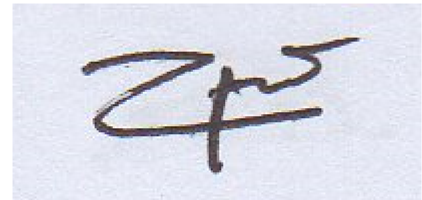
[8] http://cs.lmu.edu/~ray/notes/orderedtrees/, diakses pada tanggal 8 Desember 2016.

[9] http://www.bowdoin.edu/~ltoma/teaching/cs210/fall09/Slides/Maps.pdf, diakses pada tanggal 8 Desember 2016.

VIII. 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 2016



Turfa Auliarachman, 13515133