TUGAS AKHIR – IF184802

PENERAPAN STRUKTUR DATA *LINK-CUT TREE* PADA PENYELESAIAN PERMASALAHAN SPOJ 32679 ADAROADS: ADA AND ROADS

MODISTA GARSIA
05111640000031

Dosen Pembimbing I:
Rully Soelaiman, S.Kom., M.Kom.

Dosen Pembimbing II:
M. M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil.

DEPARTEMEN TEKNIK INFORMATIKA
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
Surabaya
2020

TUGAS AKHIR – IF184802

PENERAPAN STRUKTUR DATA *LINK-CUT TREE* PADA PENYELESAIAN PERMASALAHAN SPOJ 32679 ADAROADS: ADA AND ROADS

MODISTA GARSIA
05111640000031

Dosen Pembimbing I:
Rully Soelaiman, S.Kom., M.Kom.

Dosen Pembimbing II:
M. M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil.

DEPARTEMEN TEKNIK INFORMATIKA
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
Surabaya
2020

[*Halaman ini sengaja dikosongkan*]

FINAL PROJECT – IF184802

APPLICATION OF *LINK-CUT TREE* DATA STRUCTURES TO SOLVE SPOJ 32679 ADAROADS: ADA AND ROADS PROBLEM

MODISTA GARSIA
05111640000031

Supervisor I:
Rully Soelaiman, S.Kom., M.Kom.

Supervisor II:
M. M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil.

DEPARTMENT OF INFORMATICS
Faculty of Intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya
2020

[*Halaman ini sengaja dikosongkan*]

LEMBAR PENGESAHAN

PENERAPAN STRUKTUR DATA *LINK-CUT TREE PADA PENYELESAIAN PERMASALAHAN SPOJ 32679 ADAROADS : ADA AND ROADS*

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memperoleh Gelar Sarjana Komputer
pada
Bidang Studi Algoritma dan Pemrograman
Program Studi S-1 Teknik Informatika
Departemen Teknik Informatika
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember

Oleh:
Modista Garsia
NRP: 051116 40000 031

Disetujui oleh Dosen Pembimbing Tugas Akhir:

Rully Soelaiman, S.Kom., M.Kom.
NIP. 197002131994021001 ..... (Pembimbing I)

M. M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil.
NIP. 197402092002121001 ..... (Pembimbing II)

SURABAYA
JANUARI 2020

V

[*Halaman ini sengaja dikosongkan*]

PENERAPAN STRUKTUR DATA *LINK-CUT TREE* PADA PENYELESAIAN PERMASALAHAN SPOJ 32679 ADAROADS: ADA AND ROADS

Nama Mahasiswa	: Modista Garsia
NRP	: 051116 40000 031
Departemen	: Departemen Teknik Informatika Fakultas Teknologi Elektro dan Informatika Cerdas - ITS
Dosen Pembimbing I	: Rully Soelaiman, S.Kom., M.Kom.
Dosen Pembimbing II	: M. M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil.

ABSTRAK

Tugas Akhir ini berangkat dari adanya permasalahan Ada and Roads pada SPOJ, yang menceritakan bahwa Ada si kepik adalah seorang Petani. Ia ingin menanam buah dan sayur (vertex) pada lahannya. Untuk mempermudah dalam merawat tanamannya, Ada ingin membangun jalan (edge) yang menghubungkan tanamannya. Akan tetapi setiap jalan yang dibangun mempunyai biaya perbaikan, dan Ada ingin agar jalan yang dibangun mempunyai biaya perbaikan seminimal mungkin. Agar tidak ada jalan yang sia-sia dibangun, maka Ada ingin agar jalan yang dibangun tidak membentuk cycle (siklus).

Syarat yang harus tersedia dalam permasalahan di atas menimbulkan hal-hal baru yang membutuhkan metode penyelesaian yang tepat: (1) adanya vertex dan edge dari suatu graf yang akan terus menerus diperbarui sambil menghitung total cost, sehingga menghasilkan cost seminimal mungkin, (2) apabila vertex dan edge terus menerus diperbarui, maka bentuk graf akan berubah setiap pembaruan. Dari (1) dan (2) ini diperlukan struktur data yang efisien dan dinamis untuk menyelesaikan permasalahan ADAROADS.

Solusi dari permasalahan ADAROADS diimplementasikan dengan menggunakan struktur data dinamis

yaitu link-cut tree. Solusi yang dibuat mampu melewati batasan masalah pada soal dengan waktu 7,34 detik dan memori sebesar 29MB dan meraih peringkat pertama.

Kata Kunci: Graf Dinamis, Penyisipan Edge, Penghapusan Edge.

APPLICATION OF *LINK-CUT TREE* DATA STRUCTURES TO SOLVE SPOJ 32679 ADAROADS: ADA AND ROADS PROBLEM

Student Name	: Modista Garsia
Registration Number	: 051116 40000 031
Department	: Informatics Faculty of Intelligent Electrical and Informatics Technology – ITS
First Supervisor	: Rully Soelaiman, S.Kom., M.Kom.
Second Supervisor	: M. M. Irfan Subakti, S.Kom. M.Sc.Eng., M.Phil.

ABSTRACT

This thesis starts from SPOJ: Ada and Roads problems, which tells Ada the Ladybug is a farmer. She grows many fruits and vegetables (vertex). She has to take care of them so she builds many roads (edge) between them. She also doesn't want to keep unnecessary roads so after builting a road she cleans the rest of roads so her road-system doesn't contain any needless cycles. Each road has some maintenance cost and she always keeps roads in such ways that the total cost is minimized.

The condition which should exist on the problem caused new matters that need a proper solution as follows: (1) the vertices and edges of a graph which continues to be updated whilst calculate its total cost to produce the minimum cost, (2) if the vertices and edges are continuously being updated, then the graph structure will always be changed by its update. By (1) and (2), the efficient and dynamic structure data is needed to solve the ADAROADS problem.

The solution of ADAROADS problem has been implemented by using dynamic data structure, i.e., link-cut tree.

This solution has been able to solve the problem with 7,34 seconds and memory of 29MB and ranked at first.

Keyword: Dynamic Graph, Edge Insertion, Edge Deletion.

KATA PENGANTAR

Puji syukur penulis ucapkan kepada Tuhan Yang Maha Esa atas pimpinan, penyertaan, dan karunia-Nya sehingga penulis dapat menyelesaikan Tugas Akhir yang berjudul:

PENERAPAN STRUKTUR DATA *LINK-CUT TREE* PADA PERMASALAHAN SPOJ 32679 ADAROADS: ADA AND ROADS

Pengerjaan Tugas Akhir ini dilakukan untuk memenuhi salah satu syarat meraih gelar Sarjana di Departemen Teknik Informatika, Fakultas Teknologi Elektro dan Informatika Cerdas, Institut Teknologi Sepuluh Nopember.

Dengan selesainya Tugas Akhir ini diharapkan apa yang telah dikerjakan penulis dapat memeberikan manfaat bagi perkembangan ilmu pengetahuan terutama di bidang teknologi informasi serta bagi diri penulis sendiri selaku peneliti.

Penulis mengucapkan terima kasih kepada semua pihak yang telah memberikan dukungan baik secara langsung maupun tidak langsung selama penulis mengerjakan Tugas Akhir maupun selama menempuh masa studi antara lain:

1. Terimakasih kepada Tuhan Yang Maha Esa, di mana penulis masih diberi kesempatan, kesehatan dan umur untuk menempuh kuliah disini dan menjalani hidup dengan baik.
2. Bapak dan Ibu penulis yang selalu memberikan perhatian, dorongan, dan kasih sayang juga tunjangan makanan supaya lebih semangat menempuh kuliah maupun pengerjaan Tugas Akhir ini.
3. Bapak Rully Soelaiman, S.Kom., M.Kom. selaku Dosen Pembimbing yang telah membimbing saya selama masa kuliah maupun selama penyelesaian Tugas Akhir ini, Dosen yang paling perhatian kepada saya dalam memberi ilmu, nasihat, dan motivasi selama berada menempuh kuliah di Departemen Informatika ITS.

4. Bapak M. M. Irfan Subakti, S.Kom., M.Sc.Eng., M.Phil. selaku dosen pembimbing yang telah memberikan perhatian, didikan, pengajaran, dan nasihatnya. Berkat beliau buku TA ini dapat diselesaikan dengan baik.
5. Teman-teman angkatan 2016 jurusan Informatika ITS yang telah menemani perjuangan penulis selama 4 tahun masa perkuliahan.
6. Serta pihak-pihak lain yang tidak dapat disebutkan disini yang telah banyak membantu penulis dalam penyusunan Tugas Akhir ini.

Penulis mohon maaf apabila masih ada kekurangan pada Tugas Akhir ini. Penulis juga mengharapkan kritik dan saran yang membangun untuk pembelajaran dan perbaikan di kemudian hari. Semoga melalui Tugas Akhir ini penulis dapat memberikan kontribusi dan manfaat yang sebaik-baiknya.

Surabaya, 5 Desember 2019

Modista Garsia

DAFTAR ISI

[Halaman ini sengaja dikosongkan]

DAFTAR GAMBAR

[*Halaman ini sengaja dikosongkan*]

DAFTAR TABEL

[*Halaman ini sengaja dikosongkan*]

DAFTAR KODE SUMBER

[*Halaman ini sengaja dikosongkan*]

DAFTAR NOTASI

O	Notasi kompleksitas berdasarkan batas atas
O(1)	Kompleksitas konstan
O(|V|+|E|)	Kompleksitas dipengaruhi oleh banyaknya V dan E
O(E log V)	Kompleksitas berbanding lurus terhadap E dan bersifat logaritmik terhadap V

[*Halaman ini sengaja dikosongkan*]

BAB 1
PENDAHULUAN

Pada bab ini akan dipaparkan mengenai garis besar Tugas Akhir yang meliputi latar belakang, tujuan, rumusan masalah, batasan permasalahan, metodologi pembuatan Tugas Akhir, dan sistematika penulisan buku Tugas Akhir ini.

1.1 Latar Belakang

Latar belakang dari Tugas Akhir yang hendak diajukan ini, bermula dari adanya permasalahan *Ada and Roads* pada SPOJ. Permasalahan tersebut menceritakan bahwa Ada ”si Kumbang” adalah seorang Petani. Ia ingin menanam buah dan sayur pada lahannya. Untuk mempermudah dalam merawat tanamannya, Ada ingin membangun jalan yang menghubungkan tanamannya. Akan tetapi setiap jalan yang dibangun mempunyai biaya perbaikan, dan Ada ingin agar jalan yang dibangun mempunyai biaya perbaikan seminimal mungkin. Agar tidak ada jalan yang sia-sia dibangun, maka Ada ingin agar jalan yang dibangun tidak membentuk *cyclic.*

Permasalahan ini menggambarkan sebuah graf yang akan terus diupdate baik *vertex* maupun *edge*nya, kemudian hasil akhir yang diminta adalah *total weight* dari sebuah graf (disebut sebagai *cost*) dengan syarat bahwa pada graf tersebut tidak boleh terjadi *cyclic*.

Syarat yang harus dikerjakan dalam permasalahan diatas menimbulkan hal-hal baru yang membutuhkan struktur data yang tepat untuk menyelesaikan permasalahan dari ADAROADS. Hal pertama adalah adanya *vertex* dan *edge* yang akan terus menerus di*update* sambil menghitung *totalcost*, sehingga menghasilkan *cost* seminimal mungkin. Hal kedua adalah, apabila *vertex* terus menerus di*update*, maka bentuk graf akan berubah setiap *update*. Sehingga dari dua hal ini diperlukan struktur data yang efisien dan dinamis untuk menyelesaikan masalah ADAROADS (Ada and Roads).

Hasil dari Tugas Akhir ini adalah implementasi solusi dari penggunaan struktur data yang dinamis dan mampu menyelesaikan permasalahan dari ADAROADS. Solusi tersebut mempunyai kompleksitas waktu yang optimal untuk setiap *query* pengambilan hasil dari *total cost* dan perubahan bentuk graf.

1.2 Rumusan Permasalahan

Rumusan masalah yang diangkat dalam Tugas Akhir ini adalah sebagai berikut.

1. Bagaimana desain struktur data *link-cut tree* untuk permasalahan SPOJ *Ada and Roads* (ADAROADS)?
2. Bagaimana implementasi struktur data *link-cut tree* berdasarkan desain yang telah dilakukan?
3. Bagaimana performa kebenaran dan kinerja dari implementasi yang dilakukan?

1.3 Batasan Permasalahan

Permasalahan pada pembuatan Tugas Akhir ini memiliki batasan dalam mengimplementasikan solusi Tugas Akhir seperti dibawah ini.

- Stuktur data yang digunakan dibatasi sampai dengan implementasi dari struktur data *link-cut tree* sebagai solusi masalah

Sedangkan batasan-batasan dari SPOJ adalah sebagai berikut.

- Implementasi program dengan menggunakan bahasa C++.
- *N* sebagai banyaknya *vertex* (buah/sayur) berupa bilangan dengan tipe *integer* antara 1-20000.
- *M* sebagai banyakanya *edges* (banyak jalan yang akan dibangun) berupa bilangan dengan tipe *integer* antara 1-20000.

- Variabel *a,b,c* dengan batasan, $0 \leq a,b < N$ dan $0 \leq c \leq 10^9$, dimana *a* dan *b* adalah *vertex* yang saling berhubungan dan *c* adalah *weight*nya.
- Batasan waktunya adalah 3 detik.
- Batasan memorinya adalah 1536 MB.
- Batas ukuran programnya adalah 50kB.

1.4 Tujuan Pembuatan Tugas Akhir

Tujuan dari Tugas Akhir ini antara lain,

1. Mengimplementasikan solusi struktur data *link-cut tree* untuk permasalahan graf yang dinamis
2. Melakukan uji coba untuk mengetahui kebenaran dan kinerja dari implementasi yang dilakukan.

1.5 Manfaat Tugas Akhir

Tugas Akhir ini adalah untuk lebih memahami struktur data dinamis yaitu *link-cut tree* dan bagaimana cara menerapkannya untuk menyelesaikan permasalahan graf yang dinamis.

1.6 Metodologi

Langkah-langkah yang ditempuh dalam pengerjaan Tugas Akhir ini yaitu:

1.6.1 Penyusunan Proposal Tugas Akhir

Tahap awal untuk memulai pengerjaan tugas akhir adalah penyusunan proposal Tugas Akhir. Pada tugas akhir ini, penulis mengajukan gagasan untuk menyelesaikan permasalahan pada studi kasus SPOJ klasik ADAROADS dengan menggunakan

struktur data dinamis yang diterapkan dalam struktur data *link-cut tree*.

1.6.2 Studi Literatur

Pada tahap ini dilakukan pencarian informasi dan studi literatur yang relevan untuk dijadikan referensi dalam melakukan pengerjaan Tugas Akhir. Informasi didapatkan dari materi-materi yang berhubungan dengan struktur data. Informasi tersebut didapatkan dari buku, internet, dan materi kuliah yang berhubungan dengan metode yang akan digunakan.

1.6.3 Implementasi Perangkat Lunak

Tahapan ini merupakan tahapan untuk membangun algoritma yang akan digunakan. Pada tahap ini dilakukan implementasi dari rancangan struktur data yang akan dimodelkan sesuai dengan permasalahan. Implementasi ini dilakukan dengan menggunakan bahasa pemrograman C++.

1.6.4 Pengujian dan Evaluasi

Pada tahap ini dilakukan uji coba kebenaran dan kinerja solusi yang telah diimplementasikan dengan melakukan pengiriman sumber kode sistem ke situs penilaian *Sphere Online Judge(SPOJ)* pada permasalahan yang terkait untuk mendapatkan umpan balik. Pengujian dilakukan dengan membandingkan kompleksitas hasil uji coba dengan kompleksitas hasil analisis.

1.6.5 Penyusunan Buku Tugas Akhir

Pada tahap ini dilakukan penyusunan laporan yang menjelaskan dasar teori dan metode yang digunakan dalam tugas

akhir ini serta hasil dari implementasi aplikasi perangkat lunak yang telah dibuat.

1.7 Sistematika Penulisan

Buku Tugas Akhir ini merupakan laporan secara lengkap mengenai Tugas Akhir yang telah dikerjakan baik dari sisi teori, rancangan, maupun implementasi sehingga memudahkan bagi pembaca dan juga pihak yang ingin mengembangkan lebih lanjut. Sistematika penulisan buku Tugas Akhir secara garis besar dapat dilihat seperti dibawah ini.

Bab I Pendahuluan

Bab ini berisi penjelasan latar belakang, rumusan masalah, batasan masalah dan tujuan pembuatan Tugas Akhir. Selain itu, metodologi pengerjaan dan sistematika penulisan laporan Tugas Akhir juga dijelaskan di dalamnya.

Bab II Dasar Teori

Bab ini berisi penjelasan secara detil mengenai dasar-dasar penunjang dan teori-teori yang digunakan untuk mendukung pembuatan Tugas Akhir ini.

Bab III Perancangan dan Analisis

Bab ini berisi penjelasan tentang rancangan dari sistem yang akan dibangun.

Bab IV Implementasi

Bab ini berisi penjelasan implementasi dari rancangan yang telah dibuat pada bab III. Implementasi disajikan dalam bentuk *pseudocode* disertai dengan penjelasannya.

Bab V Pengujian dan Evaluasi

Bab ini berisi penjelasan mengenai data hasil

percobaan dan pembahasan mengenai hasil percobaan yang telah dilakukan.

Bab VI Kesimpulan dan Saran

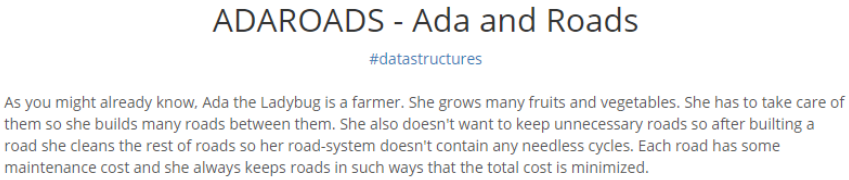
Bab ini merupakan bab terakhir yang menjelaskan kesimpulan dari hasil uji coba yang dilakukan dan saran untuk pengembangan perangkat lunak ke depannya.

BAB 2
DASAR TEORI

Pada bab ini akan dijelaskan mengenai dasar-dasar teori yang akan digunakan guna menyelesaikan permasalahan ini. Dasar teori yang digunakan meliputi *splay tree*, *link-cut tree*, serta penjabaran penggunaan teori dan landasan pemilihan struktur data yang akan digunakan sebagai solusi pada permasalahan tersebut.

2.1 Penjelasan Permasalahan SPOJ ADAROADS

ADAROADS – Ada and Roads[1], merupakan suatu permasalahan yang terdapat pada situs penilaian *Sphere Online Judge* (SPOJ) dengan deskripsi soal dari sumber asli menggunakan bahasa Inggris, yang dapat dilihat pada Gambar 2.1.


ADAROADS - Ada and Roads

#datastructures

As you might already know, Ada the Ladybug is a farmer. She grows many fruits and vegetables. She has to take care of them so she builds many roads between them. She also doesn't want to keep unnecessary roads so after builting a road she cleans the rest of roads so her road-system doesn't contain any needless cycles. Each road has some maintenance cost and she always keeps roads in such ways that the total cost is minimized.

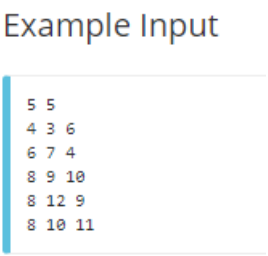
Gambar 2.1 Deskripsi permasalahan ADAROADS

Pada permasalahan ADAROADS diceritakan bahwa Ada "si Kumbang" adalah seorang Petani. Dia ingin menanam banyak buah dan sayur. Karena harus merawat tanamannya maka ia membangun jalan yang menghubungkan tanamannya. Ia tidak ingin ada jalan yang dibuat secara sia-sia. Oleh karena itu dia ingin jalan yang dibuat tidak membentuk *cycle*. Diketahui pada soal bahwa setiap jalan yang dibangun Ada, mempunyai biaya perbaikan, dan dia ingin agar biaya perbaikan dari jalan yang dibuat bisa seminimal mungkin.

Pada permasalahan ini, terdapat batasan permasalahan sebagai berikut.

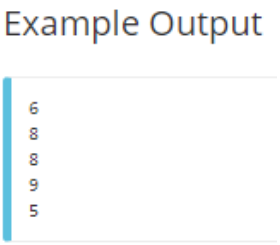
1. N sebagai banyaknya node (buah) berupa bilangan dengan tipe Integer antara 1-20000
2. M sebagai banyakanya edges (banyak jalan yang akan dibangun) berupa bilangan dengan tipe Integer antara 1-20000
3. a,b,c dengan batasan, $0 \leq a,b < N$ dan $0 \leq c \leq 10^9$, dimana a dan b adalah vertex yang saling berhubungan dan c adalah weightnya.

Masukan dari permasalahan ini adalah banyaknya sayur yang akan ditanam (N) dan banyaknya jalan yang akan dibangun (M). Setelah itu, masukan berikutnya berupa sayur yang hendak dihubungkan beserta biaya perbaikannya. Contoh masukan pada permasalahan ini, dapat dilihat pada Gambar 2.2.



Gambar 2.2 Contoh masukan pada permasalahn ADAROADS

Dari masukan seperti Gambar 2.2 akan menghasilkan total biaya perbaikan dari setiap jalan yang dibangun. Keluaran dari contoh masukan yang diberikan dapat dilihat pada Gambar 2.3.

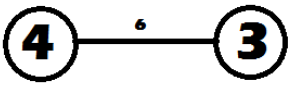


Gambar 2.3 Contoh keluaran pada permasalahan ADAROADS

Setiap masukan yang diberikan bukan merupakan data masukan yang sebenarnya. Data masukan yang diberikan harus di XOR terlebih dahulu dengan total biaya perbaikan yang dimulai dari 0. Data masukan sebenarnya dari contoh inputan Gambar 2.2 dapat dilihat pada Gambar 2.4.

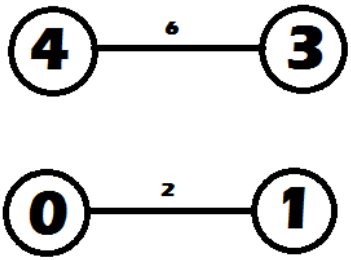
Gambar 2.4 Data masukan sebenarnya dari permasalan

Pada contoh masukan yang diberikan pada Gambar 2.2 maka diketahui akan terdapat 5 *node* dan 5 *edge* yang akan menghubungkan *node*. Permasalahan ini akan digambarkan dengan menggunakan graf. Pada masukan pertama, akan ada jalan yang dibangun untuk menghubungkan sayuran 4 dan sayuran 3. Ilustrasi dari jalan yang dibuat dapat dilihat pada Gambar 2.5.



Gambar 2.5Gambar graf setelah masukan pertama

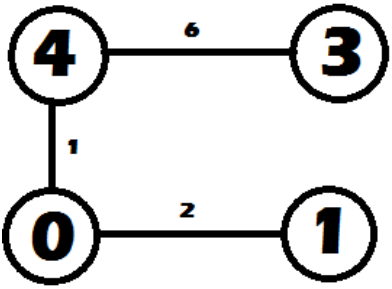
Setelah jalan pertama dibangun, maka total biaya perbaikan sekarang menjadi 6. Pada masukan kedua, sayuran yang akan dihubungkan adalah sayuran 0 dengan sayuran 1 dengan biaya perbaikan sebanyak 2. Ilustrasi dari jalan yang dibuat dapat dilihat pada Gambar 2.6.



Gambar 2.6 Gambar graf setelah masukan kedua

Setelah jalan kedua dibangun, maka total biaya perbaikan sekarang adalah 8. Pada masukan ketiga, sayuran yang akan dihubungkan adalah sayuran 0 dengan sayuran 1 dengan biaya perbaikan sebanyak 2. Karena sayur 0 dan sayur 1 sudah dihubungkan dengan jalan yang mempunyai biaya perbaikan sebesar 2, maka tidak terjadi perubahan pada bentuk graf.

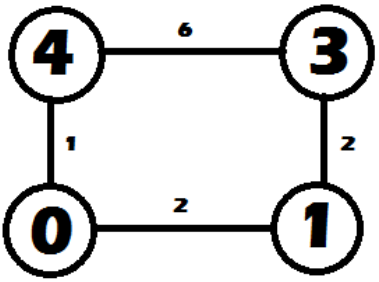
Pada masukan keempat, sayuran yang akan dihubungkan adalah sayur 0 dan sayur 4 dengan biaya perbaikan sebesar 1. Ilustrasi dari jalan yang dibuat dapat dilihat pada Gambar 2.7.



Gambar 2.7 Gambar graf setelah masukan keempat

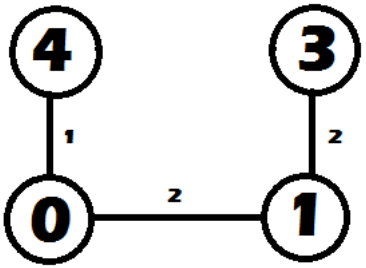
Setelah jalan keempat dibangun, maka total biaya perbaikan sekarang adalah 9. Sayuran yang akan dihubungkan

berikutnya adalah sayuran 1 dan sayuran 3 dengan biaya perbaikan sebesar 2. Ilustrasi dari jalan yang dibuat dapat dilihat pada Gambar 2.8.



Gambar 2.8 Gambar graf setelah masukan kelima

Pada soal, tidak diperbolehkan adanya jalan yang membentuk *cycle*, karena graf yang terbentuk setelah masukan kelima membentuk *cycle*, maka ada jalan yang harus dihapus. Untuk meminimalkan total biaya perbaikan yang dibangun, maka jalan dengan biaya perbaikan terbesar akan dihapus. Pada graf Gambar 2.8 biaya perbaikan terbesar adalah jalan yang menghubungkan sayuran 4 dengan sayuran 3, sebesar 6. Maka jalan tersebut akan dihapus. Ilustrasi dari jalan yang dibangun sekarang dapat dilihat pada Gambar 2.9.



Gambar 2.9 Gambar graf akhir setelah operasi *cut*

Masalah utama dalam penyelesaian masalah ADAROADS adalah struktur data yang akan digunakan untuk merepresentasikan masukan. Hal ini dikarenakan, bentuk graf akan selalu berubah sebanyak M kali, sehingga diperlukan

penggunaan struktur data yang efisien untuk menyelesaikan permasalahan ini.

2.2 Pemilihan struktur data *link-cut tree* sebagai solusi masalah

Permasalahan dari ADAROADS harus diselesaikan dengan menggunakan struktur data yang tepat dan efisien. Hal ini disebabkan karena bentuk graf dapat selalu berubah disetiap masukan yang diberikan.

Dengan menggunakan struktur data graf statis, operasi untuk menambah vertex memiliki kompleksitas sebesar O(1) sedangkan operasi untuk menghapus edge memiliki kompleksitas sebesar O(|V|+|E|). Selain itu, untuk mencari *edge* dengan bobot terbesar pada graf, bisa dilakukan dengan menggunakan algoritma dari *minimum spanning tree*, yang memiliki kompleksitas O(E log V). Pada *worst* case, untuk menghapus satu edge akan memakan waktu hingga 20 detik, sedangkan batasan waktu pada soal adalah 3 detik. Oleh karena itu diperlukan struktur data yang data statis tidak efisien dalam menyelesaikan soal.

Operasi yang dapat dilakukan pada problem soal ADAROADS bisa mencapai 50000 kali, kemungkinan terburuk jumlah vertex dari graf bisa sangat banyak, maka dengan menggunakan struktur data yang statis dapat memakan waktu yang cukup lama dan melebihi batasan waktu soal. Selain itu, data struktur yang nantinya digunakan harus mendukung transformasi graf, yang apabila kita menggunakan data struktur statis maka akan sangat kompleks.

Oleh karena itu pada penyelesaian masalah ini akan digunakan struktur data yang dinamis yaitu *link-cut tree* karena struktur data ini efisien untuk digunakan dalam

mengomputasikan permasalahan *minimum spanning tree* dengan batasan tertentu dan masing-masing operasi pada struktur data ini dapat dijalankan dengan kompleksitas rata-rata O(E log V)[2] sehingga mampu menyelesaikan solusi dengan batasan waktu yang diberikan.

2.3 *Splay Tree*

Splay tree merupakan sruktur data berupa *binary search tree* yang mampu mengatur ulang dirinya sendiri untuk mencapai kondisi *balance*, sehingga disebut sebagai *self-balancing binary search tree*.

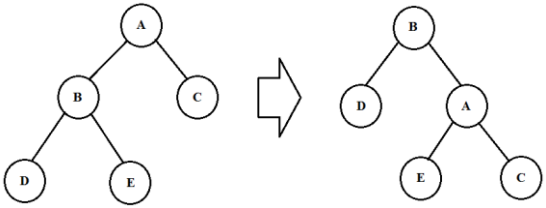
Cara yang digunakan untuk melakukan struktur ulang pada treenya adalah dengan menggunakan teknik *splay*. *Splay* adalah operasi yang menjadikan *node* yang diakses menjadi *root* dari *tree* tersebut. Untuk menjadikan *node*nya menjadi *root*, diperlukan operasi yaitu rotasi, karena teknik rotasi adalah salah satu teknik untuk merestruktur *tree* tanpa merubah informasi pada *tree*.

Splay tree melakukan operasi-operasinya berdasarkan jalur. Karena *splay tree* harus membawa *node* yang diakses menuju *root*, maka jalur yang dilewati oleh *node* untuk menjadi *root* akan berpindah. Hal ini cenderung membuat *tree* yang apabila semula tidak *balance*, menjadi *balance*. Setiap kali dilakukan *splay* terhadap *tree*, maka *height* dari *tree* akan berubah menjadi $\pm\frac{1}{2}$ dari *height* semula[4].

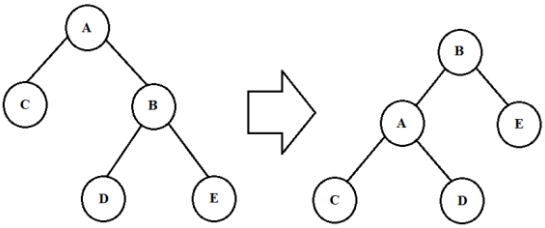
Operasi-operasi yang terdapat pada *splay tree* antara lain,

- *zig/zag*
  Operasi ini dilakukan apabila *node* (*x*) hanya mempunyai *parent* (*p(x)*). Apabila x merupakan *left-child* dari *p(x)*, maka operasi yang dilakukan adalah operasi *zig*. Tetapi apabila x merupakan *right-child* dari *p(x)* maka operasi yang dilakukan adalah *zag*. Untuk melakukan operasi *zig*, maka penunjuk *left-*

child dari *p(x)* dihilangkan, kemudian penunjuk *right-child* dari node *x* diarahkan ke *p(x)*. Ilustrasi dari operasi *zig* dapat dilihat pada Gambar 2.10, sedangkan ilustrasi dari operasi *zag* dapat dilihat pada Gambar 2.11.



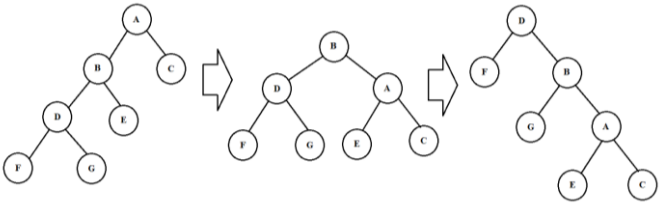
Gambar 2.10 Operasi *zig*



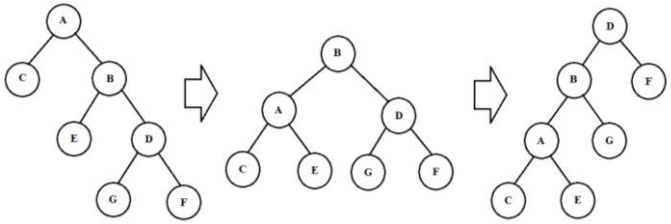
Gambar 2.11 Operasi *zag*

- *zig-zig/zag-zag*
  Operasi ini dilakukan apabila *node* (*x*) memiliki *parent* (*p(x)*) dan *p(x)* memiliki *parent* (*g(x)*). Operasi *zig-zig* dilakukan apabila x merupakan *left-child* dari *p(x)*, dan *p(x)* merupakan *left-child* dari *g(x)*. Untuk kasus sebaliknya maka operasi yang dilakukan adalah *zag-zag*. Untuk melakukan operasi *zig-zig*, maka penunjuk *right-child* dari *p(x)* diarahkan ke *g(x)*, kemudian penunjuk *right-child* dari *x*, diarahkan ke *p(x)*.Ilustrasi dari operasi *zig-zig* dapat dilihat pada Gambar 2.12, sedangkan ilustrasi

untuk operasi *zag-zag* dapat dilihat pada Gambar 2.13.

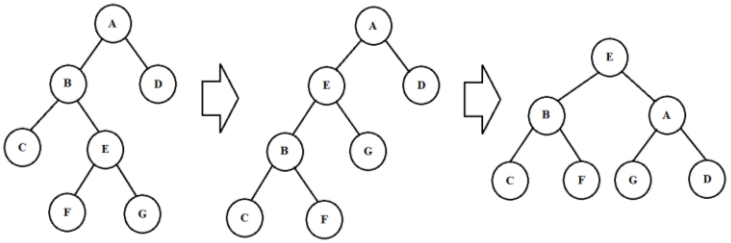


Gambar 2.12 Operasi *zig-zig*



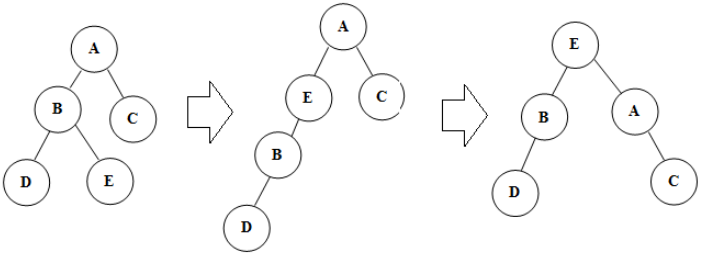
Gambar 2.13 Operasi *zag-zag*

- *zig-zag*
  Operasi ini dilakukan apabila *node* (*x*) memiliki *parent* (*p(x)*) dan *parent* memiliki *parent* (*g(x)*). Operasi *zig-zag* dilakukan apabila x merupakan *left-child* dari *p(x)* dan *p(x)* merupakan *right-child* dari *g(x)*. Untuk kasus sebaliknya maka operasi yang dilakukan adalah operasi *zag-zig*. Operasi *zig-zag* dilakukan dengan menghilangkan penunjuk *child* pada *p(x)* kemudian penunjuk *left-child* dari *x* diarahkan ke *p(x)*. Selanjutnya, penunjuk *right-child* dari *x* diarahkan ke *g(x)*. Ilustrasi dari operasi *zig-zag* dapat dilihat pada Gambar 2.14.



Gambar 2.14 Operasi *zig-zag*

Berikut merupakan salah satu contoh penggunaan *splay tree* yang dapat dilihat pada Gambar 2.15.



Gambar 2.15 Contoh operasi *splay(E)*

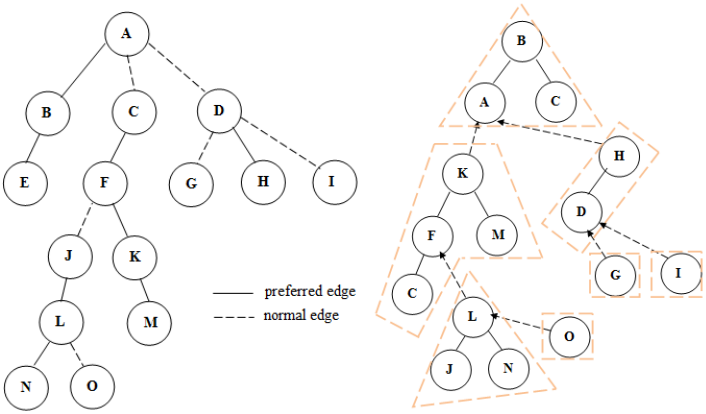
2.4 *Link-cut Tree*

Link-cut tree merupakan salah satu struktur data yang digunakan untuk merepresentasikan kumpulan *tree*. Setiap *tree* terpisahkan kedalam masing-masing jalur (*path).*

Kumpulan dari jalur-jalur tersebut nantinya akan dinamakan *represented tree.* Setiap *path* pada *represented* tree nantinya akan direpresentasikan kedalam *auxiliary tree.* Representasi dari *auxiliary tree (aux tree)*biasanya menggunakan struktur data *Splay* tree.

Pada *represented tree* terdapat istilah *preferred path. Preferred path* merupakan jalur yang terbentuk saat *node* sedang diakses, dan *edge* pembentuk *preferred path* disebut sebagai

preferred edge. Ilustrasi dari *represented tree* dan *aux tree* dapat dilihat pada Gambar 2.16.

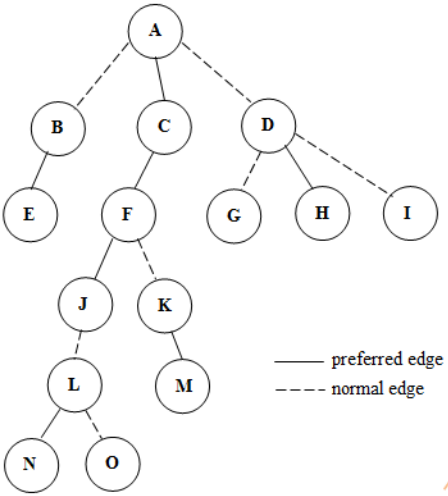


Gambar 2.16 (a) *Represented tree* dan (b) *Auxiliary tree* pada *link-cut tree*

Edge yang terhubung pada *represented tree* tetapi tidak terdapat pada jalur yang sama atau *path(u) ≠ path(v)* (seperti node a dan c pada Gambar 2.16(a)), dimana *parent(v) = u* maka pada *auxiliary tree*nya akan memiliki penunjuk parent atau *path parent(aux(v)) = u*.

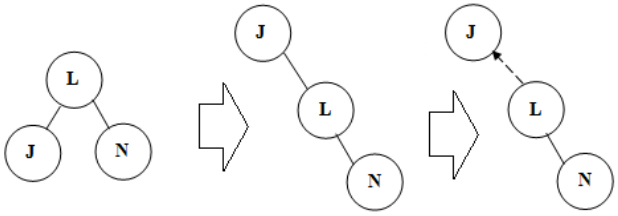
2.4.1 Algoritma *access*

Operasi ini digunakan untuk membuat *path* dari *root* menuju *node* yang di access. Dalam *represented tree*, operasi ini dilakukan dengan membuat *path* dari *root* hingga *node* yang diakses menjadi *preferred path.* Ilustrasi dari operasi *access(j)* pada *represented tree*dapat dilihat pada Gambar 2.17.



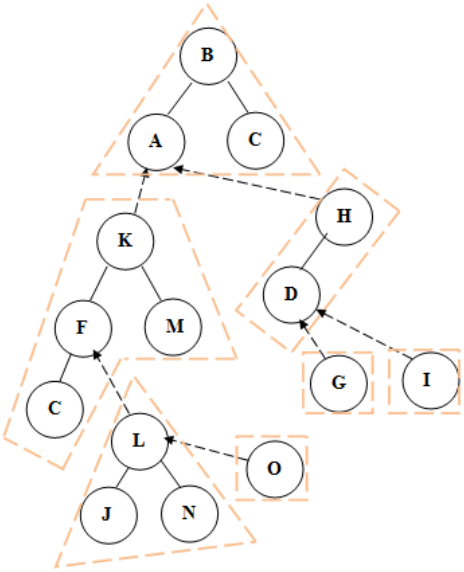
Gambar 2.17 Operasi *access(j)* pada *represented tree*

Pada *aux tree*, operasi *access* dilakukan dengan menjadikan *node* yang di*access* menjadi *node* terdalam pada *aux tree*, karena *node* harus menjadi ujung pada *pathnya*. Pada satu *tree*, terdiri dari banyak *path*, maka *node* yang di*access* pertama-tama harus menjadi *root* pada *aux tree*nya. Karena struktur data yang digunakan untuk merepresentasikan *aux tree* adalah *splay tree*, maka operasi ini dilakukan dengan melakukan operasi *splay*. Setelah itu, seluruh *node* yang nilainya lebih besar akan dipisah menjadi *aux tree* baru dengan *path parent* menunjuk pada *node*. Ilustrasi dari operasi *splay(j)* dapat dilihat pada Gambar 2.18.

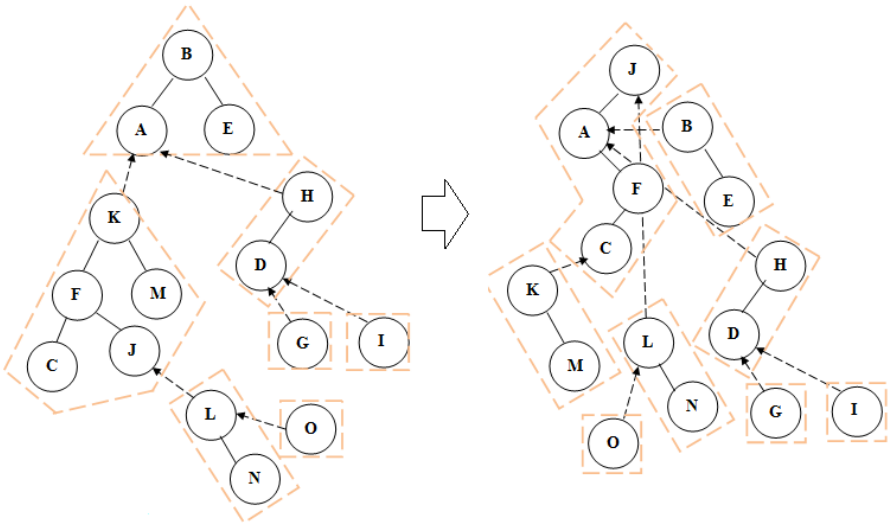


Gambar 2.18 *splay(j)* agar *j* menjadi *root* pada *aux tree*

Setelah *node* menjadi root pada *aux tree*nya, maka *node* akan digabungkan dengan *path parent*nya dengan menjadikan *node* sebagai *right child* dari *path parentnya*. Langkah ini dilakukan terus menerus hingga *node* tidak lagi mempunyai *path parent*. Ilustrasi dari operasi *access(j)* pada *aux tree* dapat dilihat pada Gambar 2.19 dan Gambar 2.20.



Gambar 2.19 Operasi *access(j)* pada *auxiliary tre*

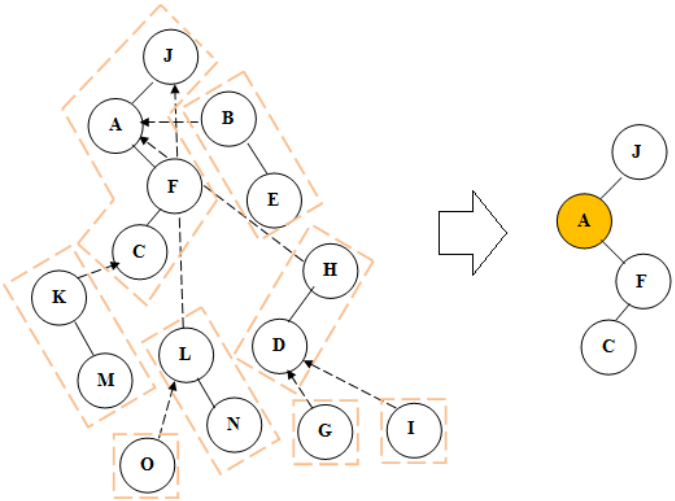


Gambar 2.20 Operasi *access(j)* pada *auxiliary tree cont.*

2.4.2 Algoritma untuk mencari *root*

Algoritma ini digunakan untuk mencari *root* dari *node(x)*. Untuk melakukan pencarian *root x* pada *aux tree*-nya, maka perlu melakukan *access(x)*. Setelah *x* menjadi *root* dari *aux tree*, maka node terkecil dari *aux tree* adalah *root* pada *represented tree*.

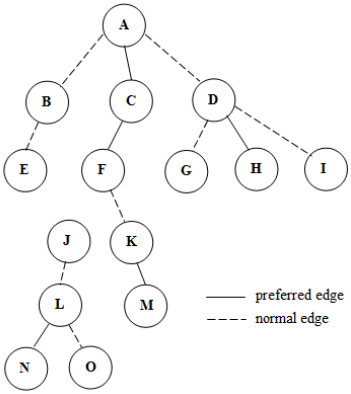
Pencarian node terkecil dilakukan dengan menelurusi *left child* dari *x* secara rekursif sampai mencapai *node* yang tidak mempunyai *left child*. Ilustrasi dari operasi *find root(j)* dapat dilihat pada Gambar 2.21.



Gambar 2.21 Operasi *find root(j)*

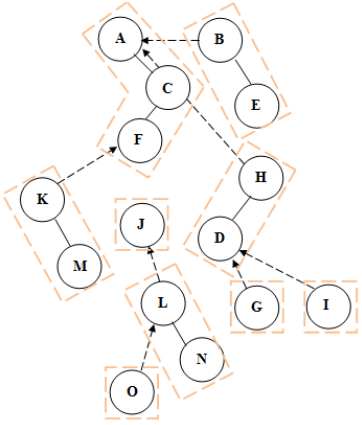
2.4.3 Algoritma *cut*

Operasi *cut* digunakan untuk menghilangkan *edge* yang menghubungkan *node* dengan *parent*nya. Pada *represented tree*, operasi *cut* dilakukan dengan melakukan *access* pada *node* yang ingin di *cut*, setelah itu menghapus *left child* yang terhubung ke *node*, sehingga *node* membentuk *aux tree* yang baru. Ilustrasi dari operasi *cut(j)* pada *represented tree* dapat dilihat pada Gambar 2.22.



Gambar 2.22 Operasi *cut* pada *represented tree*

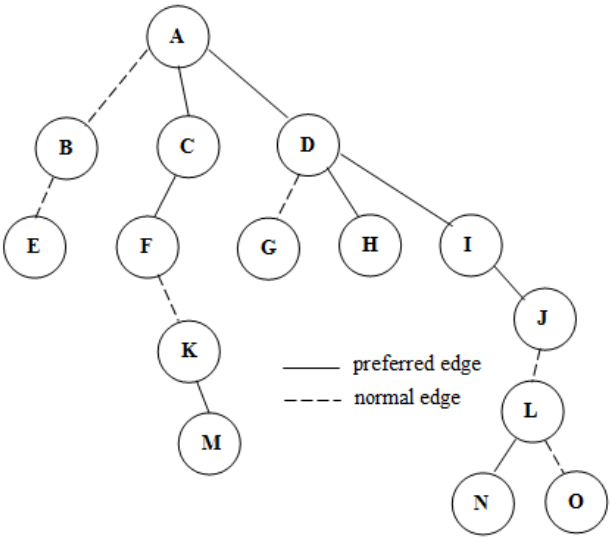
Pada *aux tree* operasi *cut* dilakukan dengan menghilangkan *node* yang nilainya lebih kecil daripada node yang di*cut*. Untuk melakukan operasi *cut* maka perlu melakukan *access*. Setelah *itu* penunjuk *left child* dari *node* dihilangkan, sehingga *node* tidak lagi memiliki *left-child*. Ilustrasi dari operasi *cut* pada *aux tree* dapat dilihat pada Gambar 2.23.



Gambar 2.23 Operasi *cut(j)*pada *auxiliary tree*

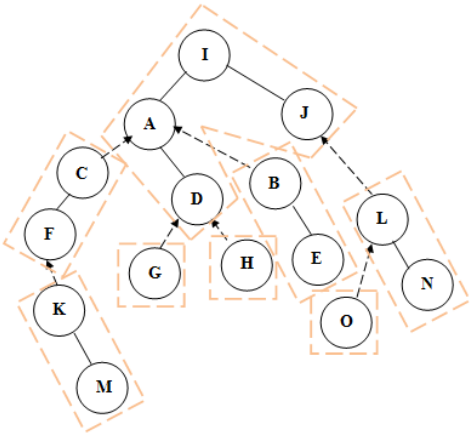
2.4.4 Algoritma *link*

Link merupakan operasi yang digunakan untuk menggabungkan 2 *node* yang berada pada *tree* yang berbeda. Pada *represented tree*, operasi *link(i,j)* dilakukan dengan melakukan *access* pada masing-masing *node i* dan *node j*, setelah itu hubungkan dengan menjadikan *node j* sebagai *child* dari *node i*. Ilustrasi dari operasi *link(i,j)* pada *represented tree* dapat dilihat pada Gambar 2.24.



Gambar 2.24 Operasi *link(i,j)* pada *represented tree*

Pada *aux tree* , operasi *link(i,j)* dilakukan *access(i)* dan *acces(j)* terlebih dahulu agar *node i* dan *node j*. Untuk melakukan *link* pada *node i* dengan *node j*, dilakukan dengan menjadikan *node i* sebagai *left child* dari *node j*.Ilustrasi dari operasi *link(i,j)* pada *aux tree* dapat dilihat pada Gambar 2.25.



Gambar 2.25 Operasi *link(i,j)* pada *auxiliary tree*

2.5 Penyelesaian permasalahan ADAROADS dengan *link-cut tree*

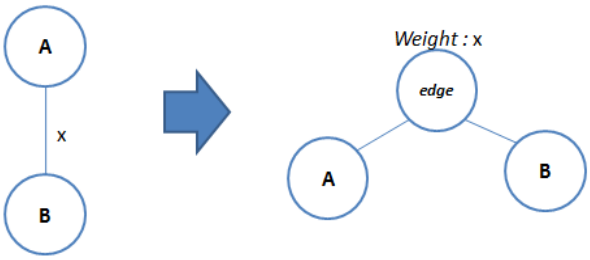
Struktur data *link-cut tree* digunakan untuk memodelkan buah/sayur yang saling terhubung dengan jalannya. Buah/sayur akan direpresentasikan dengan *node (T)*, begitu pula dengan jalan (*E*) dalam bentuk *array* sebesar masukan yang diberikan. Sebelum melakukan operasi baik itu penghubungan *node T* ataupun penghapusan *node E*, maka kita harus memastikan bahwa *root* dari kedua *node* adalah sama atau berbeda.

Untuk penghubungan antar *node*, apabila *root* dari kedua *node T* berbeda maka penghubungan bisa dilakukan. Akan tetapi apabila *root* dari kedua *node T* adalah sama, maka kita harus membandingkan bobot *node E* yang hendak ditambahkan serta *node E* yang merupakan bobot terbesar dari *subtree node T*. Apabila *node E* yang hendak dihubungkan memiliki bobot yang lebih kecil dibandingkan dengan *node E* yang merupakan bobot terbesar dari *subtree node T*, maka perlu adanya penghapusan *node E* yang merupakan bobot terbesar dari *subtree*.

Pada subab 2.1, dijelaskan mengenai alur dari permasalahan dengan memodelkan permasalahan kedalam bentuk graf. Akan tetapi, pada solusi penyelesaian bentuk struktur data yang digunakan adalah *tree*. Oleh karena itu diperlukan penyesuaian sehingga bentuk graf dapat direpresentasikan menjadi tree.

Pada bentuk graf *node x* dan *node y* akan terhubung dengan sebuah *edge e*, untuk merepresentasikannya kedalam bentuk tree, maka *edge e* tidak bisa dijadikan jalur dari tree, karena struktur data *link-cut tree* mendasari operasinya dari jalur masing-masing tree, sehingga apabila pada jalur tree menyimpan sebuah informasi, maka implementasi yang dilakukan akan menjadi kompleks karena implementasi harus mempertahankan informasi pada jalur yang selalu berubah.

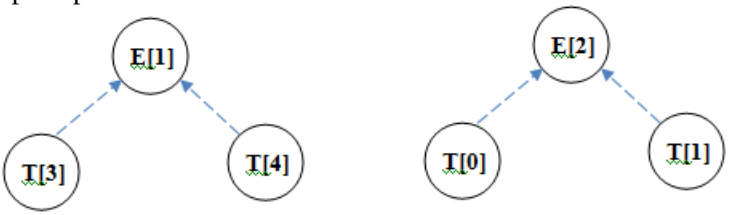
Maka untuk mengantisipasinya, *edge e* pada graf, juga akan direpresentasikan dalam bentuk *node* yang memiliki bobot, sedangkan *node x* dan *node y* akan tetap menjadi *node* yang tidak memiliki bobot. Sehingga representasi graf menjadi *tree* dapat dilihat pada Gambar 2.26.



Gambar 2.26 Representasi graf kedalam bentuk *tree*

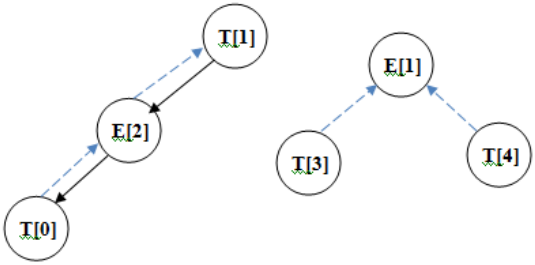
Pada masukan Gambar 2.6 dapat dilihat bahwa masukan pertama dan masukan kedua masih menghubungkan *node T* yang terletak pada *subtree* yang berbeda. Sehingga total biaya perbaikan adalah dengan menjumlahkan bobot dari *node E* yang menghubungkan masing-masing *node*. Apabila direpresentasikan

dalam bentuk *link-cut tree*, maka bentuk *tree* dapat diilustrasikan seperti pada Gambar 2.27.



Gambar 2.27 Gambar tree setelah masukan kedua

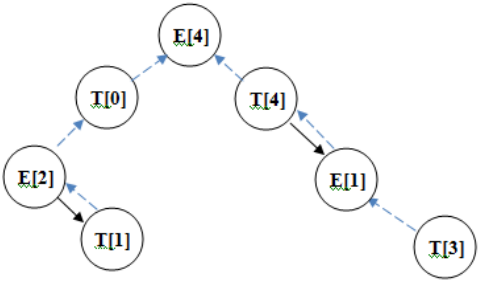
Sedangkan pada masukan ketiga, *node T* yang ingin dihubungkan memiliki *root* yang sama yaitu, *E[2]* sehingga harus dibandingkan bobotnya terlebih dahulu dengan bobot dari *node E* yang hendak disisipkan. Tetapi karena *node E* yang hendak disisipkan memiliki bobot yang sama dengan *node E* dari *subtree* sehingga tidak ada perubahan pada struktur *tree*. Ilustrasi dari *tree* setelah masukan ketiga dapat dilihat pada Gambar 2.28.



Gambar 2.28 Gambar tree setelah masukan ketiga

Pada masukan keempat, *node T* yang hendak dihubungkan berada pada *subtree* yang berbeda sehingga operasi penghubungan bisa dilakukan. Untuk melakukan operasi penghubungan, maka *node* yang hendak digabungkan yaitu *T[0]* dengan *T[4]* harus menjadi *root* dari masing-masing *subtree*.

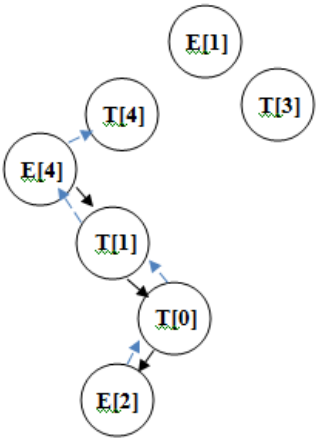
Setelah masing-masing *node* telah menjadi *root*, maka penghubungan antar *node T* dengan *node E* dapat dilakukan. Total biaya perbaikan nantinya akan ditambah dengan bobot *node E* yang hendak disisipkan. Bentuk*tree* setelah masukan keempat dapat diilustrasikan seperti pada Gambar 2.29.



Gambar 2.29 Gambar tree setelah masukan keempat

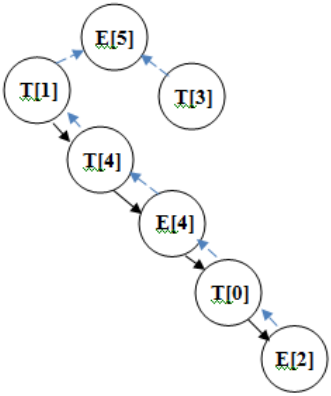
Pada masukan kelima, *node T* yang akan dihubungkan adalah *node T[3]* dengan *T[1]*. Karena kedua *node* memiliki *root* yang sama yaitu *E[4]*, berarti kedua *node* berada pada *tree* yang sama. Sebelum menghubungkan *node T[3]* dengan *T[1]* perlu membandingkan *node* dengan bobot terbesar pada *subtree* dengan bobot yang hendak disisipkan.

Pada *tree* yang ditunjukan pada Gambar 2.29, node dengan bobot terbesar adalah *E[1]*, dan karena bobot *E[1]*lebih besar dari pada bobot yang hendak disisipkan, maka *node E[1]* harus dihapus dari *tree* dengan menghilangkan semua hubungan *parent* dan *child*nya. Untuk melakukannya maka *E[1]* harus dijadikan *root* dari *tree*, setelah itu menghapus penunjuk *parent* yang menunjuk *E[1]*, dan menghapus penunjuk *child* dari *E[1]*. Selain itu total biaya perbaikan harus dikurangi dengan bobot dari *E[1]*.



Gambar 2.30 Gambar tree setelah operasi *cut* E[1]

Setelah menghapus hubungan dengan *node E[1]*, maka *T[1]* dengan *T[3]* dapat dihubungkan dengan *E[5]*, dan total biaya perbaikan ditambah dengan bobot dari *E[5]*. Bentuk dari *tree* setelah masukan terakhir, diilustrasikan seperti pada Gambar 2.31.



Gambar 2.31 Gambar tree setelah masukan kelima

BAB 3
PERANCANGAN DAN ANALISIS

Pada bagian ini akan dijelaskan analisis struktur data yang digunakan untuk menyelesaikan permasalahan pada Tugas Akhir ini. Pada bagian pertama akan dijelaskan deskripsi umum dari sistem yang akan dibuat, kemudian dilanjutkan dengan desain struktur data, serta desain fungsi untuk pencarian *node* dengan bobot terbesar pada *tree*.

3.1 Deskripsi Umum Sistem

Pada subab ini akan dijelaskan tentang gambaran secara umum dari sistem yang akan dibangun. Pertama-tama sistem akan menerima masukan berupa *N* dan *M* yang masing-masing merepresentasikan jumlah buah/sayuran yang akan ditanam oleh "Ada", dan jumlah jalan yang akan dibangun untuk menghubungkan buah/sayuran tersebut.

Sistem kemudian akan menerima masukan yaitu, *a, b* dan *c* yang masing-masing merepresentasikan sayur/buah yang dihubungkan serta biaya perbaikan jalannya sebanyak *M* kali. Nilai *a,b* dan *c* nantinya akan direpresentasikan kedalam struktur data *link-cut tree* sebagai *node.*

Sebelum melanjutkan ke proses lainnya, masukan *a, b,* dan *c* harus diolah dengan menggunakan operasi *XOR* dengan total biaya perbaikan (dimulai dari 0) untuk mendapatkan masukan sesungguhnya dari *a, b,* dan *c*.

Sistem kemudian memodelkan masukan sesungguhnya dari *a, b,* dan *c* dengan menggunakan struktur data *link-cut tree*. Operasi yang dapat terjadi pada masukan yang diberikan adalah, menghubungkan *node a* dengan *b,* atau menghilangkan *node* dengan bobot terbesar pada *tree* yang menghubungkan *a* dan *b.*

Maka sistem akan memastikan terlebih dahulu bahwa *node* yang ingin dihubungkan yaitu, *a* dan *b* terletak pada *tree* yang berbeda. Apabila *node a* dan *b* terletak pada *tree* yang sama,

maka sistem akan membandingkan bobot dari *node c* dengan bobot dari *node* terberat pada *tree*. Apabila *node c* memiliki bobot lebih kecil, maka sistem akan menghilangkan seluruh hubungan pada *node* terberat pada *tree*, sebelum menghubungkan *node a* dengan *node b*.

Kemudian sistem akan menampilkan total biaya perbaikan jalan setiap masukan *a*,*b* dan *c* dimasukan kedalam permasalahan. *Pseudocode* dari fungsi *main* dapat dilihat pada Gambar 3.1.

```
main ( )
1.  N,M ← Input
2.  total cost ← 0
3.  T[N] ← null
4.  E[N] ← null
5.  for i ← 0, M do
6.      a, b, c← Input
7.      a, b, c ⊕total cost
8.      if T[a].findroot is T[b].findroot
9.          max ←query(T[a], T[b])
10.         if max→value less than c then
11.             Print(total cost)
12.             continue
13.         endif
14.         total cost ← total cost – max→value
15.         max→cut(T[x[max→id]])
16.         max→cut(T[y[max→id]])
17.     endif
18.     T[a].link(E[i])
19.     T[b].link(E[i])
20.     Print(total cost)
21. endfor
```

Gambar 3.1 *Pseudocode* fungsi *main*

3.2 Desain Struktur Data

Struktur Data *link-cut tree* akan digunakan untuk merepresentasikan buah/sayur yang terhubung satu sama lain beserta bobot biaya perbaikannya. Untuk memodelkannya, maka *node* yang dibuat adalah *node* buah/sayur (*T*) beserta *node* bobot biaya perbaikan (*E*). Masing-masing *node T* akan terhubung dengan *node E* yang menjadi jalan yang menghubungkan kedua *node* tersebut.

Baik *node E* dan *node T* akan mempunyai atribut yaitu penunjuk *parent, left child,* dan *right child*. Selain itu *node* juga mempunyai variabel *integer* untuk menyimpan nilai bobot dan nilai *id* sebagai pembeda antar *node*. *Node* juga mempunyai variabel *Boolean*yaitu sebagai penanda apakah *node* tersebut telah melakukan penukaran *child*.

3.2.1 Desain Fungsi *Splay*

Splay merupakan operasi pada *splay tree* yang digunakan untuk merepresentasikan *auxiliary tree* pada *link-cut tree*. Sebelum melakukan operasi *splay* maka *node* yang hendak di*splay* beserta dengan *path parent*nya harus dicek apakah pernah membalikan *child* maka seluruh childnya harus dibalik untuk menjaga struktur tree.

Fungsi yang digunakan adalah fungsi ***push*** untuk melakukan perubahan terhadap seluruh *child* dan fungsi ***flip*** yang digunakan untuk menukar *left child* dengan *right child*. *Pseudocode* fungsi *flip* dapat dilihat pada Gambar 3.2.

flip ( )

```
1.  If this is null then return
2.  else
3.      swap (this→leftchild, this→rightchild)
4.      this→rev = this→rev ^ 1
5.  endif
```

Gambar 3.2 Gambar *pseudocode* fungsi *flip*

Pseudocode dari fungsi *push* dapat dilihat pada Gambar 3.3.

```
push ( )
1.  if rev is 1then
2.          this→leftchild→flip()
3.          this→rightchild→flip()
4.          this  → rev = 0
5.  endif
```

Gambar 3.3 Gambar *pseudocode* fungsi *push*

Operasi splay digunakan untuk menjadikan node yang display menjadi root pada *subtree*nya Untuk melakukan operasi ini, maka fungsi perlu mengecek secara berulang apakah node yang sedang display adalah *root* dari *subtree*nya, apabila node sudah menjadi *root* maka perulangan dihentikan.

Sehingga untuk melakukan operasi splay terdapat beberapa fungsi yang diperlukan, yaitu fungsi ***isroot*** untuk mengetahui apakah node adalah *root* dari *subtree,* fungsi ***rotasi*** untuk membawa node yang sedang display menjadi *root* pada *subtree,* dan fungsi ***splay.*** *Pseudocode* untuk operasi *isroot* dapat dilihat pada Gambar 3.4.

```
isroot ( )
1.  isroot ← 0
2.  if parent is null or (parent→leftchild is not this and
    parent→rightchild is not this) then
3.          isroot ←  1
4.  return  isroot
```

Gambar 3.4 Gambar *pseudocode* fungsi *isroot*

Sebelum melakukan operasi *splay*, *node* yang hendak di*splay* dan seluruh *path parent*nya harus dicek terlebih dahulu apakah *node* tersebut sudah pernah membalik *child*nya (dari *left child* menjadi *right child* dan sebaliknya). Apabila *node* yang hendak di*splay* pernah membalikan *child*nya, maka seluruh

*child*nya harus dibalik untuk mengembalikan struktur *tree*nya. Untuk mengecek apakah *node* dan seluruh *node* yang berada pada *path parent*nya sudah pernah membalikan *child*nya atau belum, digunakan fungsi ***go***. *Pseudocode* dari fungsi *go* dapat dilihat pada Gambar 3.5.

```
go ( )
1.  if this→isroot() is False then
2.          this→parent→go()
3.  endif
4.  this→push()
```

Gambar 3.5 Gambar *pseudocode* untuk fungsi *go*

Untuk melakukan fungsi rotasi, maka kita perlu mengetahui arah rotasi yang akan dilakukan. Untuk mengetahui arah rotasi akan digunakan fungsi ***isrightch***. Arah rotasi ditentukan dengan melihat posisi dari *node* sekarang dengan *parent*nya. Apabila *node* sekarang adalah *right child* dari *parent*nya, maka rotasi yang dilakukan adalah rotasi kanan.

Selain itu, kita juga memerlukan fungsi ***setchild*** yang akan digunakan untuk melakukan *assign value* kepada *child* dari *node*. *Pseudocode* untuk fungsi *isrightchild* dapat dilihat pada Gambar 3.6.

```
isrightch ( )
1.  isrightch ←0
2.  if parent→rightchild is this then
3.          isrightch ← 1
4.  return isrightch
```

Gambar 3.6 Gambar *pseudocode* fungsi *isrightchild*

Pseudocode untuk fungsi *setchild* dapat dilihat pada Gambar 3.7, sedangkan *pseudocode* untuk fungsi rotasi dapat dilihat pada Gambar 3.8.

setchild ( )

Input : node, direction
1. **if** direction **is** 1 **then**
2. this →*rightchild*= node
3. **else**
4. this →*leftchild* = node
5. **endif**
6. node →*parent* = this

Gambar 3.7 Gambar *pseudocode* fungsi *setchild*

rotate ( )

1. p ← parent
2. gp ← parent→ parent
3. c ←*isrightch()*
4. gc ← parent→*isrightch()*
5. node ← this→*leftchild*
6. nodep ← parent→*parent*→*leftchild*
7. nodeg ← *null*
8. **if** c **is** 1 **then**
9. node ←*rightchild*
10. **endif**
11. **if** gc **is** 1 **then**
12. nodep ← parent→*parent*→*rightchild*
13. **endif**
14. p→*setchild*(node, c)
15. this→*setchild*(p, !c)
16. **if** nodep **is** p **then**
17. gp→*setchild*(this,gc)
18. **else**
19. this→*parent* = gp
20. **endif**
21. p→*update()*

Gambar 3.8 Gambar *pseudocode* fungsi *rotate*

Pseudocode untuk fungsi *splay* dapat dilihat pada Gambar 3.9.

```
splay ( )
1.  for this→go() , this→isroot() do
2.      if parent→isroot() is false then
3.          If parent→isrightch() is this→isrightch()
            then
4.              parent →rotate()
5.          else
6.              this→rotate()
7.          endif
8.      endif
9.      this→rotate()
10.     this→update()
11. endfor
```

Gambar 3.9 Gambar *pseudocode* fungsi *splay*

3.2.2 Desain Fungsi Akses

Fungsi akses merupakan operasi pada *link-cut tree* yang digunakan untuk menjadikan *node* yang diakses menjadi *root* dari *tree*. Hal ini bertujuan agar apabila *node* yang sama akan menjalani operasi lainnya, waktu yang digunakan untuk mencapai *node* tersebut jauh lebih cepat. *Pseudocode* untuk operasi akses dapat dilihat pada Gambar 3.10.

```
access ( )
1.  for a ←this and b ← null, a is NULL do
2.      a→splay()
3.      a→setchild(b, 1)
4.      a→update()
5.      b ← a
6.      a ← a→parent
7.  endfor
```

Gambar 3.10 Gambar *pseudocode* fungsi *access*

3.2.3 Desain Fungsi Mencari *Root*

Fungsi *findroot* digunakan untuk mencari *root* pada *node*. Operasi ini digunakan untuk mennetukan apakah dua *node* terletak pada *tree* yang sama atu tidak. *Node* berada pada satu *tree* yang sama apabila *root*nya adalah sama. *Root* dari sebuah *tree* adalah *node* terkecil pada *tree* tersebut. *Pseudocode* untuk operasi *findroot* dapat dilihat pada Gambar 3.11.

```
findroot ( )
1.  x ←NULL
2.  for x ←access() and x→push(), x→leftchild is null do
3.          x ← x→leftchild
4.  endfor
5.  return x
```

Gambar 3.11 Gambar *pseudocode* fungsi *find_root*

3.2.4 Desain Fungsi *link*

Fungsi *link* merupakan operasi *link-cut tree* yang digunakan untuk menghubungkan *node* yang terletak pada *tree* yang berbeda. Untuk melakukan operasi ini, *node* yang ingin dihubungkan harus berada pada *tree* berbeda, dan apabila *node* terletak pada *tree* yang berbeda maka hubungkan kedua *node*.

Operasi *link* akan melibatkan 2 *node* (*x* dan *y*). *Node x* akan dijadikan *root* pada *tree*, dengan melakukan *access* pada *node*, kemudian *child* pada *node* di*flip* agar *node x* tidak memiliki *left child* sehingga *node x* menjadi *root* pada *tree*. Setelah itu *node y* yang akan dihubungkan, akan memiliki penunjuk *parent* yang mengarah ke *node x*. *Pseudocode* dari operasi *link* dapat dilihat pada Gambar 3.12.

```
link ( )
Input : node
1.  if this→findroot() is node→findrot() then
2.          return
3.  endif
4.  this→access()→flip()
5.  this→parent←node
```

Gambar 3.12 Gambar *pseudocode* fungsi *link*

3.2.5 Desain Fungsi *cut*

Fungsi *cut* digunakan untuk memisahkan *node* dengan bobot terbesar dengan *node* lain yang memiliki hubungan dengannya. Pada studi kasus ADAROADS, operasi ini dilakukan ketika *node T* yang ingin digabungkan terletak pada satu *tree* dan *node* dengan bobot terbesar pada *tree* memiliki bobot yang lebih besar dibandingkan dengan bobot yang hendak disisipkan.

Untuk melakukan operasi ini, akan ada beberapa fungsi yang dibutuhkan yaitu, fungsi ***cut.*** Fungsi ini digunakan untuk menjadikan *node* child dari *node* yang hendak di*cut* menjadi *root*. Karena pada saat *node* di*cut*, maka *node* akan membentuk *aux tree* baru, sedangkan *child* nya akan menjadi *root* dari *aux tree*nya. Setelah itu akan ada fungsi ***cut_node*** untuk menghapus seluruh hubungan *node* yang hendak dihapus dengan *node* lainnya. *Pseudocode* untuk operasi *cut* dapat dilihat pada Gambar 3.13.

```
cut ( )
Input : node
1.  if this is node or this→findroot is not node→findroot then
2.          return
3.  else
4.          node→access()→flip()
5.          this→cut_node()
6.  endif
```

Gambar 3.13 Gambar *pseudocode* fungsi *cut*

Pseudocode untuk operasi ***cut_node*** dapat dilihat pada Gambar 3.14.

cut_node ( )
1. this→*access()*
2. this→*leftchild*→*parent*←*null*
3. this→*leftchild* ← *null*
4. this→*update*

Gambar 3.14 Gambar *pseudocode* fungsi *cut_node*

3.2.6 Desain Fungsi *update*

Fungsi *update* merupakan operasi yang digunakan untuk mempertahankan bobot dari struktur tree dari *auxiliary tree*. Hal ini karena setelah terjadinya rotasi, *node* yang dirotasi akan menjadi parent dari *subtree*nya sehingga fungsi *update* akan merubah bobot *node* menjadi bobot dari *childnya* apabila *child*nya memiliki bobot yang lebih berat daripada *node* yang di rotasi. Hal ini dilakukan agar *weight* dari *node* merepresentasikan bobot terbesar dari *subtree*nya. *Pseudocode* dari fungsi *update* dapat dilihat pada Gambar 3.15.

update ( )
1. this→*weight* ←*this*
2. **if** this→*leftchild*→*weight*→*val* **greater than** this→*weight*→*val* **then**
3. this→*weight* ← this→*leftchild*→*weight*
4. **endif**
5. **if** this→*rightchild*→*weight*→*val* **greater than** this→*weight*→*val* **then**
6. this→*weight*← this→*rightchild*→*weight*
7. **endif**

Gambar 3.15 Gambar *pseudocode* fungsi *update*

3.3 Desain Fungsi *max_weight*

Fungsi *max_weight* digunakan untuk mendapatkan bobot biaya perbaikan terbesar yang terdapat pada suatu *tree*. Sebuah *node* merupakan bobot terbesar pada *tree* apabila setelah dibandingkan dengan *node* lainnya bobot dari *node* tersebut adalah bobot terbesar. *Pseudocode* dari fungsi *max_weight* dapat dilihat pada Gambar 3.16.

```
max_weight ( )
1.  node ←this
2.  while 1 do
3.          node→push()
4.          if node→val greater than
            node→leftchild→weight→val and node→val
            greater than node→rightchild→val then
5.                  return node
6.          endif
7.          if node→leftchild →weight→val greater than
            node→rightchild→weight→val then
8.                  node ← node→leftchild
9.          else
10.                 node ← node→rightchild
11.         endif
12. endwhile
```

Gambar 3.16 Gambar *pseudocode* fungsi *max_weight*

3.4 Desain Fungsi *query*

Fungsi *query* digunakan untuk menjadikan dua *node* yang terletak pada *subtree* yang berbeda kedalam satu *subtree* yang sama. Fungsi ini dilakukan apabila kedua *node* terletak pada satu *tree* yang sama, sebelum melakukan operasi *cut*. Fungsi ini melibatkan 2 *node (x,y)* yang berbeda.

Untuk melakukan fungsi ini, maka salah satu *node x* perlu dijadikan sebagai *root* dari *tree*. Setelah itu lakukan *access* pada

node y agar kedua *node* terhubung dalam satu *subtree* yang sama. *Pseudocode* dari fungsi *query* dapat dilihat pada Gambar 3.17.

query ( )
Input : a, b 1. a→*access()* → *flip()* 2. b→*access()* 3. **return** y→*max_weight()*

Gambar 3.17 Gambar *pseudocode* fungsi *query*

BAB 4
IMPLEMENTASI

Pada bab ini akan dijelaskan mengenai implementasi dari penerapan struktur data sebagai penyelesaian permasalahan *Ada and Roads*. Pertama akan dijelaskan implementasi dari fungsi utama, dilanjutkan dengan implementasi dari struktur data yang digunakan.

4.1 Lingkungan Implementasi

Lingkungan implementasi dan pengembangan yang digunakan untuk menyelesaikan Tugas Akhir ini adalah sebagai berikut.

1. Perangkat keras
   a. Prosesor Intel(R) Core(TM) i5-3337U CPU @ 1.80GHz (4CPU), ~1.8GHz
   b. *Random Access Memory* 4096MB
2. Perangkat lunak
   a. *Operating System* Windows 7 Ultimate 64-bit
   b. Bahasa pemrograman C++
   c. *Integrated Development Environment* Orwell Dev-C++ 5.11

4.2 Implementasi Fungsi *main*

Fungsi *main* adalah implementasi Algoritma yang dirancang pada subbab 3.1. Implementasi fungsi *main* dapat dilihat pada Kode Sumber 4.1 - Kode Sumber 4.2.

```
1.    int main() {
2.     int N,M;
3.     N = getNum<int>();
4.     M = getNum<int>();
```

Kode Sumber 4.1 Implementasi fungsi *main* bagian 1

```
1.   node T[N], E[M];
2.   int x[M], y[M];
3.   for(int i = 0; i<N; i++){
4.      T[i].clr(-1);
5.    }
6.
7.    long long sum = 0;
8.    for(int i=0;i<M;i++){
9.      long long a,b,c;
10.     a = getNum<long long>();
11.     b = getNum<long long>();
12.     c = getNum<long long>();
13.     a ^= sum;
14.     b ^= sum;
15.     c ^= sum;
16.     x[i] = a; y[i]= b;
17.     E[i].clr(c); E[i].id = i;
18.
19.     if(T[a].find_root()==
     T[b].find_root()){
20.       auto max = query(&T[a], &T[b]);
21.       if(max->val <= E[i].val){
22.         printf("%lld\n", sum);
23.         continue;
24.       }
25.       sum-=max->val;
26.       max->cut(&T[x[max->id]]);
27.       max->cut(&T[y[max->id]]);
28.     }
29.
30.     sum+=c;
31.     T[a].link(&E[i]);
32.     T[b].link(&E[i]);
33.     printf("%lld\n", sum);
34.   }
35.  }
```

Kode Sumber 4.2 Implementasi fungsi *main* bagian 2

4.3 Implementasi Struktur data

Pada bagian ini akan dijelaskan mengenai implementasi dari subab 3.2 yang dibagi menjadi beberapa subab sebagai berikut.

4.3.1 Implementasi fungsi *init*

Fungsi *init* digunakan untuk membuat *node null*. Node *null* yang dibuat bertujuan untuk mempercepat implementasi yang dilakukan, sehingga setiap operasi pada struktur data tidak perlu memastikan apakah node yang dioperasikan *null* atau tidak. Implementasi dari fungsi *init* dapat dilihat pada Kode Sumber 4.3.

```
1.    void init() {
2.        null = new node();
3.        null->clr(-1);
4.    }
```

Kode Sumber 4.3 Implementasi fungsi *init*

4.3.2 Implementasi *struct node*

Struct node digunakan untuk merepresentasikan *node*, yaitu *node T* (merepresentasikan buah/sayur) dan *node E* (merepresentasikan jalan). Pada *struct node* terdapat atribut yang diperlukan pada *node* serta implementasi algoritma yang digunakan pada *link-cut tree*. Implementasi dari *struct node* dapat dilihat pada Kode Sumber 4.4 dan Kode Sumber 4.5.

```
1.    struct node *null;
2.    struct node{
3.     node *parent, *ch[2];
4.     node *weight;
5.     int val,rev,id;
6.
7.     void clr(int v);
```

Kode Sumber 4.4 Implementasi *struct node* bagian 1

```
1.      void setchild(node* node, int d);
2.      bool isrightch();
3.      bool isroot();
4.      void flip();
5.      void push();
6.      void update();
7.      void go();
8.      void cut_node();
9.      void rotate();
10.     node* splay();
11.     node* access();
12.     node* find_root();
13.     void cut(node* node);
14.     void link(node* node);
15.     node* max_weight();
16.   }
```

Kode Sumber 4.5 Implementasi *struct node* bagian 2

Pada *struct node*, *node* mempunyai penunjuk *parent*, serta penunjuk *left child* dan *right child* yang direpresentasikan dengan *pointer ch* berbentuk *array* (*ch[0]* sebagai penunjuk *left child*, dan *ch[1]* sebagai penunjuk *right child*). *Struct node* juga mempunyai pointer *weight* untuk menunjuk *node* dengan bobot terberat pada *child*nya. *Struct node* mempunyai nilai *val* untuk menyimpan besar bobot (*val* hanya memiliki nilai pada *node E*), nilai *rev* sebagai penanda apakah *node* telah melakukan penukaran *child*, serta nilai *id* sebagai pembeda antar *node* (*id* hanya memiliki nilai pada *node E*, untuk membantu pencariannya).

4.3.3 Implementasi fungsi *flip*

Fungsi *flip* digunakan untuk menukar *left child* dengan *right child* pada *tree* seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.2**.** Implementasi dari fungsi ini dapat dilihat pada Kode Sumber 4.6.

```
1.    void flip(){
2.       if(this==null) return;
3.       swap(ch[0], ch[1]);
4.       rev^=1;
5.     }
```

Kode Sumber 4.6 Implementasi fungsi *flip*

4.3.4 Implementasi fungsi *push*

Fungsi *push* digunakan untuk menukar posisi *child* dari sebuah *tree* apabila *node* pada *tree* tersebut telah menukar *child*nya seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.3. Implementasi dari fungsi ini dapat dilihat pada Kode Sumber 4.7.

```
1.    void push(){
2.       if(rev){
3.         ch[0]->flip();
4.         ch[1]->flip();
5.         rev=0;
6.       }
7.     }
```

Kode Sumber 4.7 Implementasi fungsi *push*

4.3.5 Implementasi fungsi *isroot*

Fungsi *isroot* digunakan untuk menentukan apakah *node* merupakan *root* pada *auxiliary tree*, seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.4. Implementasi dari fungsi ini dapat dilihat pada Kode Sumber 4.8.

```
1.    bool isroot(){
2.       return parent == null || (parent-
      >ch[0]!=this        &&        parent-
      >ch[1]!=this);
3.    }
```

Kode Sumber 4.8 Implementasi fungsi *isroot*

4.3.6 Implementasi fungsi *go*

Fungsi *go* digunakan untuk mengecek apakah *path parent* pada *node* pernah menukar *child*, karena apabila *node* pada *path parent* pernah menukar *child*, maka seluruh *child* pada *preferred path* harus ditukar posisinya, seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.5. Implementasi dari fungsi ini dapat dilihat pada Kode Sumber 4.9.

```
1.     void go(){
2.       if(!isroot()){
3.         parent->go();
4.       }
5.       push();
6.     }
```

Kode Sumber 4.9 Implementasi fungsi *go*

4.3.7 Implementasi fungsi *isrightchild*

Fungsi *isrightchild* digunakan untuk mengecek posisi *node*, apakah *node* merupakan *right child* dari *parent*nya atau tidak, seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.6. Implementasi dari fungsi ini dapat dilihat pada Kode Sumber 4.10.

```
1.    bool isrightch(){
2.       return parent->ch[1]==this;
3.     }
```

Kode Sumber 4.10 Implementasi fungsi *isrightchild*

4.3.8 Implementasi fungsi *setchild*

Fungsi *setchild* digunakan untuk menukar penunjuk *child* dari *node*, seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.7. Implementasi dari fungsi ini dapat dilihat pada Kode Sumber 4.11.

```
1.    void setchild(node *node, int d){
2.       ch[d] = node;
3.       node->parent = this;
4.     }
```

Kode Sumber 4.11 Implementasi fungsi *setchild*

4.3.9 Implementasi fungsi *rotate*

Fungsi *rotate* digunakan untuk melakukan rotasi pada *auxiliary tree*, seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.8. Implementasi fungsi ini dapat dilihat pada Kode Sumber 4.12.

```
1.    void rotate(){
2.       node *p = parent, *gp = parent-
      >parent;
3.       int c = isrightch(), gc = parent-
      >isrightch();
4.
5.       p->setchild(ch[!c], c);
6.       this->setchild(p, !c);
7.       if(gp->ch[gc]==p)              gp-
      >setchild(this,gc);
8.       else this->parent = gp;
9.       p->update();
10.    }
```

Kode Sumber 4.12 Implementasi fungsi *rotate*

4.3.10 Implementasi fungsi *splay*

Fungsi *splay* digunakan agar *node* yang di*splay* menjadi *root* dari *auxiliary tree*, seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.9. Implementasi dari fugsi ini dapat dilihat pada Kode Sumber 4.13.

```
1.    node* splay(){
2.       for(go(); !isroot(); rotate()){
3.         if(!parent->isroot()){
4.           if(isrightch()==parent-
      >isrightch()) parent->rotate();
5.           else rotate();
6.         }
7.       }
8.       update();
9.       return this;
10.    }
```

Kode Sumber 4.13 Implementasi fungsi *splay*

4.3.11 Implementasi fungsi *access*

Fungsi *access* digunakan agar *node* yang di*access* menjadi *root* pada *tree*, seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.10. Implementasi dari fungsi ini dapat dilihat pada Kode Sumber 4.14.

```
1.    node* access(){
2.       for(node* a = this, *b = null;
      a!=null; b=a, a = a->parent ){
3.         a->splay();
4.         a->setchild(b,1);
5.         a->update();}
6.       return splay();
7.     }
```

Kode Sumber 4.14 Implementasi fungsi *access*

4.3.12 Implementasi fungsi *find_root*

Fungsi *find_root* digunakan untuk mencari *node* yang merupakan *root* dari *tree*nya, seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.11. Implementasi dari fungsi ini dapat dilihat pada Kode Sumber 4.15.

```
1.    node* find_root(){
2.       node *x;
3.       for(x = access(); x->push(), x-
      >ch[0]!=null; x=x->ch[0]);
4.       return x;
5.     }
```

Kode Sumber 4.15 Impleemntasi fungsi *find_root*

4.3.13 Implementasi fungsi *link*

Fungsi *link* digunakan untuk menghubungkan 2 *node* yang terletak pada *tree* yang berbeda, seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.12. Implementasi dari fungsi ini dapat dilihat pada Kode Sumber 4.16.

```
1.    void link(node* node){
2.       if(find_root()==node->
      find_root()) return;
3.       else {
4.         makeroot();
5.         parent = node;
6.       }
7.    }
```

Kode Sumber 4.16 Implementasi fungsi *link*

4.3.14 Implementasi fungsi *cut*

Fungsi *cut* digunakan untuk memisahkan 2 *node* menjadi *tree* yang berbeda, seperti yang sudah dijelaskan pada subab

3.2.5. Implementasi fungsi ini dibagi menjadi 2 bagian. Bagian pertama, yaitu menjadikan *child* dari *node* yang di*cut* menjadi *root* dari *tree*, seperti yang telah dijelaskan pada *pseudocode* di Gambar 3.13. Implementasi dari bagian ini dapat dilihat pada Kode Sumber 4.17**.**

```
1.    void cut(node* node){
2.       if(this==node  ||  find_root()  !=
      node->find_root()) return;
3.       else{
4.         node->makeroot();
5.         cut_node();
6.       }
7.    }
```

Kode Sumber 4.17 Implementasi fungsi *cut*

Sedangkan bagian kedua, yaitu memisahkan *parent* dengan *child*nya, seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.14. Implementasi pada bagian ini dapat dilihat pada Kode Sumber 4.18**.**

```
1.    void cut_node(){
2.       access();
3.       ch[0]->parent = null;
4.       ch[0] = null;
5.       update();
6.    }
```

Kode Sumber 4.18 Implementasi fungsi *cut node*

4.3.15 Implementasi fungsi *update*

Fungsi *update* digunakan untuk melakukan *update* bobot dari *node*, seperti yang telah dijelaskan pada *pseudocode* di Gambar 3.15. Implementasi dari fungsi ini dapat dilihat pada Kode Sumber 4.19.

```
1.    void update(){
2.       weight = this;
3.       if(ch[0]->weight->val  >  weight-
      >val) weight = ch[0]->weight;
4.       if(ch[1]->weight->val  >  weight-
      >val) weight = ch[1]->weight;
5.     }
```

Kode Sumber 4.19 Implementasi fungsi *update*

4.3.16 Implementasi fungsi *max weight*

Fungsi *max weight* digunakan untuk mencari *node* dengan bobot terbesar pada *tree*, seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.16. Implementasi dari fungsi ini dapat dilihat pada Kode Sumber 4.20.

```
1.    node* max_weight(){
2.       node* node = this;
3.       while(1){
4.         node->push();
5.         if(node->val   >   node->ch[1]-
      >weight->val  &&  node->val  >  node-
      >ch[0]->weight->val) return node;
6.         else {
7.           if(node->ch[1]->weight->val
      <=          node->ch[0]->weight->val)
      node=node->ch[0];
8.           else node=node->ch[1];
9.         }
10.      }
11.    }
```

Kode Sumber 4.20 Implementasi fungsi *max weight*

4.4 Implementasi fungsi *query*

Fungsi *query* digunakan agar 2 *node* yang terletak pada *auxiliary tree* yang berbeda, digabung kedalam *auxiliary tree* yang sama, seperti yang sudah dijelaskan pada *pseudocode* di Gambar 3.17. Implementasi dari fungsi ini dapat dilihat pada Kode Sumber 4.21.

```
1.    node* query(node* x, node *y){
2.     x->access()->flip();
3.     y->access();
4.     return   y->max_weight();
5.    }
```

Kode Sumber 4.21 Implementasi fungsi *query*

BAB 5
UJICOBA DAN EVALUASI

Pada bab ini dijelaskan tentang uji coba dan analisis dari implementasi yang telah dilakukan pada Tugas Akhir ini. Uji coba dilakukan dengan skenario tertentu dan dilanjutkan dengan tahap analisis dari struktur data yang digunakan untuk menyelesaikan permasalahan.

5.1 Lingkungan Uji Coba

Uji coba kebenaran dan uji coba kinerja akan dilakukan pada situs penilaian SPOJ dan evaluasi kebenaran dari struktur data yang diimplementasikan akan dilakukan pada komputer pribadi penulis. Lingkungan uji coba yang dilakukan pada situs penilaian SPOJ dengan *cluster Cube* yang memiliki spesifikasi sebagai berikut.

1. Perangkat Keras
    a. Processor Intel Xeon E3-1220 v5 (5 CPUs)
    b. *Random Access Memory* 1536MB
2. Perangkat Lunak
    a. *Compiler* GCC 8.3

Sedangkan lingkungan evaluasi kebenaran dari implementasi dilakukan pada komputer pribadi penulis dengan spesifikasi sebagai berikut.

1. Perangkat keras
    a. Prosesor Intel(R) Core(TM) i5-3337U CPU @ 1.80GHz (4CPU), ~1.8GHz
    b. *Random Access Memory* 4096MB
2. Perangkat lunak
    a. *Operating System* Windows 7 Ultimate 64-bit
    b. Bahasa pemrograman C++
    c. *Integrated Development Environment* Orwell Dev-C++ 5.11

5.2 Skenario Uji Coba

Pada bagian ini akan dijelaskan skenario yang akan digunakan untuk melakukan pengujian terhadap implementasi yang dibuat untuk menyelesaikan permasalahan *Ada and Roads*.

Uji coba kebenaran, akan dilakukan dengan melihat umpan balik yang diberikan oleh SPOJ setelah sumber kode dikirimkan. SPOJ akan mengecek kebenaran dari sumber kode yang dikirim dengan memasukan kasus uji dengan batasan yang sudah dijabarkan pada subbab 2.1 yaitu.

1. *N* sebagai banyaknya *node* (buah/sayur) berupa bilangan dengan tipe *Integer* antara 1-20000
2. *M* sebagai banyakanya *edges* (banyak jalan yang akan dibangun) berupa bilangan dengan tipe *Integer* antara 1-20000
3. *a, b, c* dengan batasan, $0 \leq a, b<N$ dan $0 \leq c \leq 10^9$, dimana *a* dan *b* adalah *node* yang saling berhubungan dan *c* adalah biaya perbaikannya.

Uji coba kinerja akan dilakukan dengan mengirimkan kode hasil implementasi program dengan menggunakan struktur data *link-cut tree* ke situs penilaian SPOJ sebanyak 10 kali kemudian menganalisis kinerja dari umpan balik yang diberikan.

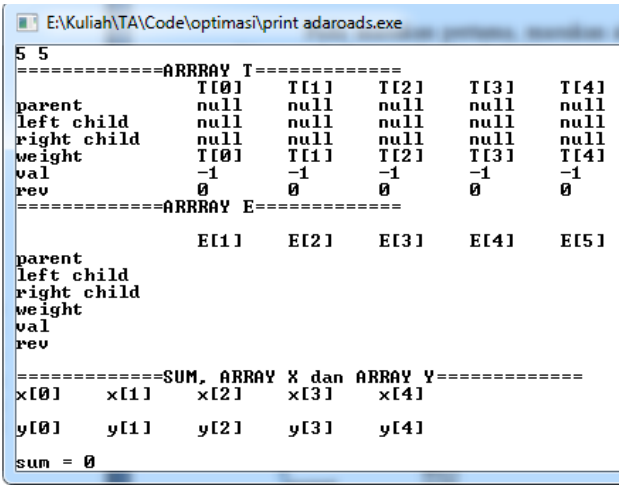
5.2.1 Evaluasi Kebenaran

Evaluasi dilakukan dengan mengecek masukan yang diberikan dengan keluaran yang dihasilkan dari implementasi program yang sudah dibuat apakah sama dengan contoh keluaran yang ada pada permasalahan SPOJ *Ada and Roads*. Kasus uji yang akan dicoba dapat dilihat pada Tabel 5.1 yang diambil dari situs SPOJ [1] uji kasus pertama, dan illustrasi dari struktur data yang sudah dijabarkan pada subab 2.5 akan dibuktikan pada subab ini.

Tabel 5.1 Tabel uji coba

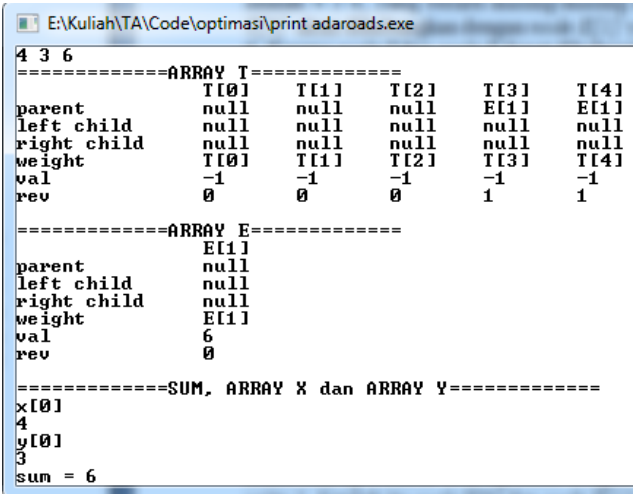
Masukan	Keluaran
5 5	
4 3 6	6
6 7 4	8
8 9 10	8
8 12 9	9
8 10 11	5

Pada program implementasi yang dibuat, sistem akan membuat *array of struct node* sebanyak 5 untuk *node T* dan 5 untuk *node E*. Selain itu sistem juga akan membuat *array x* dan *y* untuk menyimpan *index* dari *node* yang akan dihubungkan pada tiap iterasi. Sistem juga akan menyimpan nilai dari biaya perbaikan total dalam variabel *sum* yang dimulai dari 0. Ilustrasi dari kondisi awal pada sistem dapat dilihat pada Gambar 5.1.



Gambar 5.1 Kondisi awal *array T, array E, array x* serta *y* dan *total sum*

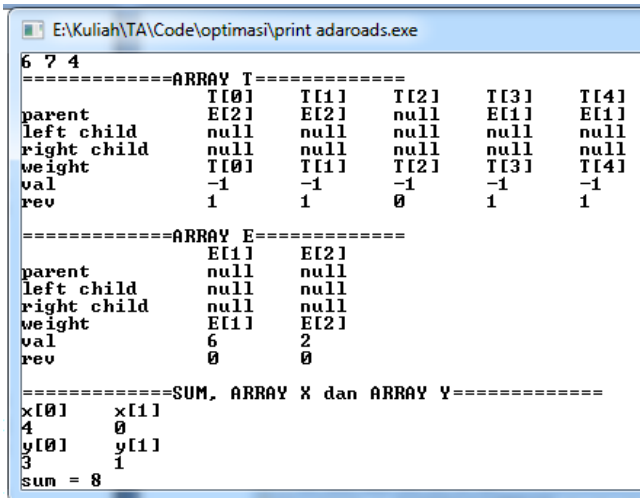
Pada masukan pertama, masukan akan di*XOR* dengan biaya perbaikan total yaitu 0. Sehingga masukan baris pertama adalah 4 3 6, yang berarti masing-masing *node T[4]* dan *node T[3]* akan dihubungkan dengan *node E[1]* yang memiliki bobot 6. Karena *node T* dan *node E* dapat dihubungkan, biaya perbaikan sekarang dijumlah dengan bobot dari *node E[1]* yaitu 6. Untuk menghubungkan kedua *node T* dengan *node E*, cukup dengan mengarahkan penunjuk *parent* dari *node T* menuju *node E*. Sehingga setelah iterasi pertama, kondisi sistem akan berubah seperti pada Gambar 5.2.



Gambar 5.2 Kondisi *array T, array E, array x* serta *aray y* dan *total sum* setelah masukan pertama

Pada masukan kedua, masukan akan di*XOR* dengan biaya perbaikan total yaitu 6, sehingga masukan kedua adalah 0 1 2, yang berarti masing-masing dari *node T[0]* dan *node T[1]* akan dihubungkan dengan *node E[2]* yang memiliki bobot 2. Sama seperti sebelumnya, karena *node T* dan *node E* dapat digabungkan maka biaya perbaikan total dijumlah dengan bobot dari *node E[2]* yaitu 2, sehingga biaya perbaikan total menjadi 8. Setelah itu

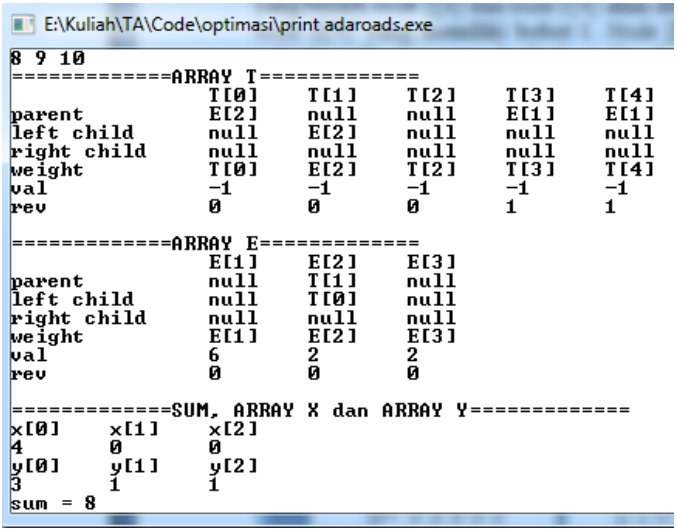
node T[0] dan *node T[1]* dihubungkan dengan *node E[2]*. Sehingga kondisi sistem akan berubah seperti pada Gambar 5.3.



Gambar 5.3 Kondisi *array T, array E, array x, array y* dan *total sum* setelah masukan kedua

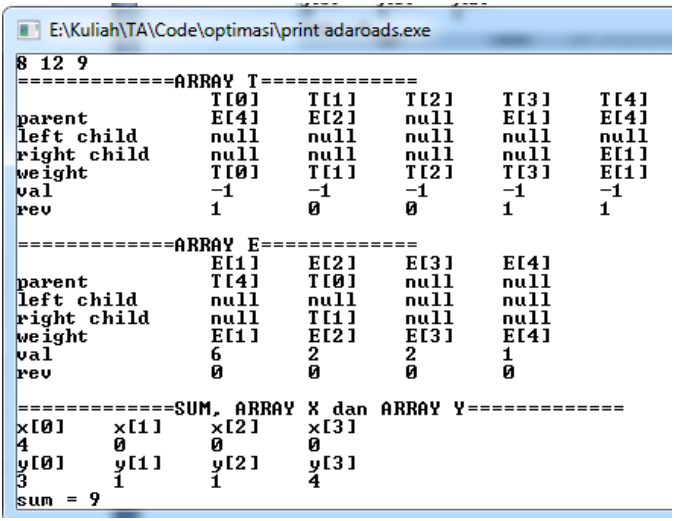
Pada masukan ketiga, masukan akan di*XOR* dengan biaya perbaikan yaitu 8, sehingga masukan ketiga adalah 0 1 2, yang berarti masing-masing dari *node T[0]* dan *node T[1]* akan dihubungkan dengan *node E[3]* yang memiliki bobot 2.

Karena *node T[0]* dan *node T[1]* terletak pada *tree* yang sama, maka sistem akan membandingkan bobot dari *node E[3]* dan *node E[2]*. Karena *node E[3]* dan *node E[2]* memiliki bobot yang sama, maka sistem akan melanjutkan ke masukan berikutnya. Kondisi sistem setelah iterasi ketiga dapat dilihat pada Gambar 5.4.



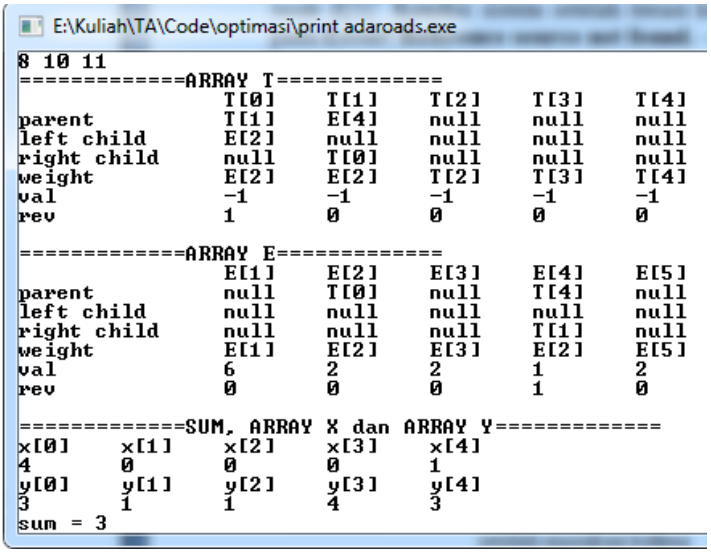
Gambar 5.4 Kondisi *array T, array E, array x, array y* dan *total sum* setelah masukan ketiga

Pada masukan keempat, masukan akan di*XOR* dengan biaya perbaikan yaitu 8, sehingga masukan keempat adalah 0 4 1, yang berarti *node T[0]* dan *node T[4]* akan dihubungkan dengan *node E[4]* yang memiliki bobot 1. *Node T[0]* dan *node T[4]* terletak pada *tree* yang berbeda maka, kedua *node T* dapat dihubungkan dengan *node E*, dan biaya perbaikan total dijumlahkan dengan bobot dari *node E[4]* yaitu 1, sehingga biaya perbaikan total menjadi 9. Kondisi sistem setelah iterasi keempat dapat dilihat pada Gambar 5.5.



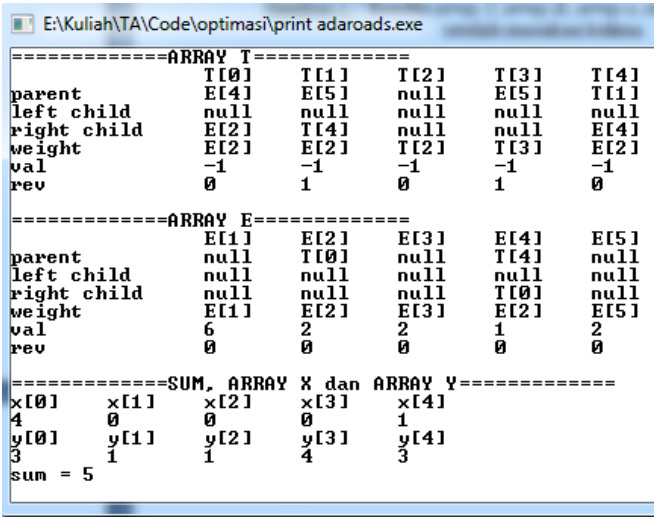
Gambar 5.5 Kondisi *array T, array E, array x*, *array y* dan *total sum* setelah masukan keempat

Pada iterasi kelima, masukan akan di*XOR* dengan biaya perbaikan total yaitu 9, sehingga masukan adalah 1 3 2. Karena *node T[1]* dan *node T[3]* terletak pada *tree* yang, sama maka sistem akan mencari *node* dengan bobot terbesar pembentuk *tree*, yaitu *node E[1]*. Sistem akan membandingkan bobot dari *node E[1]* dan *node E[5]*. Karena bobot dari *node E[5]* lebih kecil, maka sistem akan menghapus semua hubungan dari *node E[1]* pada *tree.* Biaya perbaikan total akan dikurangi dengan bobot dari *node E[1]*. Kondisi setelah sistem menghapus hubungan dari *node E[1]* dapat dilihat pada Gambar 5.6.



Gambar 5.6 Kondisi *array T, array E, array x, array y* dan *total sum* setelah *cut E[1]*

Setelah itu *node T[1]* dan *node T[3]* dihubungkan dengan *node E[5]*. Kondisi sistem setelah iterasi kelima dapat dilihat pada Gambar 5.7.



Gambar 5.7 Kondisi *array T*, *array E*, *array x*, *array y* dan *total sum* setelah masukan kelima

Setelah seluruh masukan selesai diproses, maka keluaran yang dihasilkan oleh sistem yang dibuat sudah sesuai dengan keluaran uji kasus pertama pada SPOJ [1], seperti yang dapat dilihat pada Tabel 5.2.

Tabel 5.2 Perbandingan keluaran pada sistem dan keluaran uji kasus

Iterasi	Keluaran Sistem	Keluaran kasus #1 SPOJ
1	6	6
2	8	8
3	8	8
4	9	9
5	5	5

5.2.2 Uji Coba Kebenaran

Uji coba kebenaran dilakukan dengan mengirimkan kode hasil implementasi program ke situs penilaian SPOJ: *Ada and Roads*. Setelah mengirimkan kode sumber, maka akan didapatkan umpan balik dari situs SPOJ seperti yang pada Gambar 5.8.

25166380	2020-01-01 15:00:21	Ada and Roads	**accepted**	7.34	29M	CPP

Gambar 5.8 Hasil umpan balik dari uji kebenaran pada SPOJ

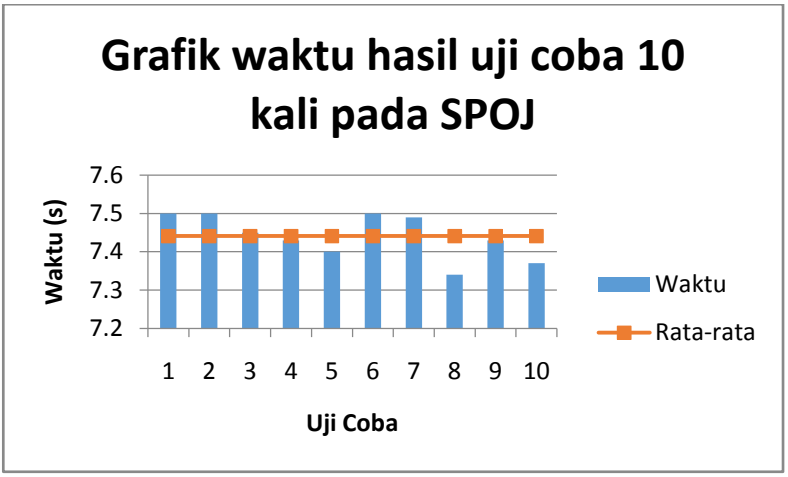
Dari hasil uji coba yang dilakukan kode sumber mendapatkan umpan balik *Accepted*. Waktu yang diperlukan program adalah 7,34 detik dan memori yang dibutuhkan program adalah 29MB.

5.2.3 Uji Coba Kinerja

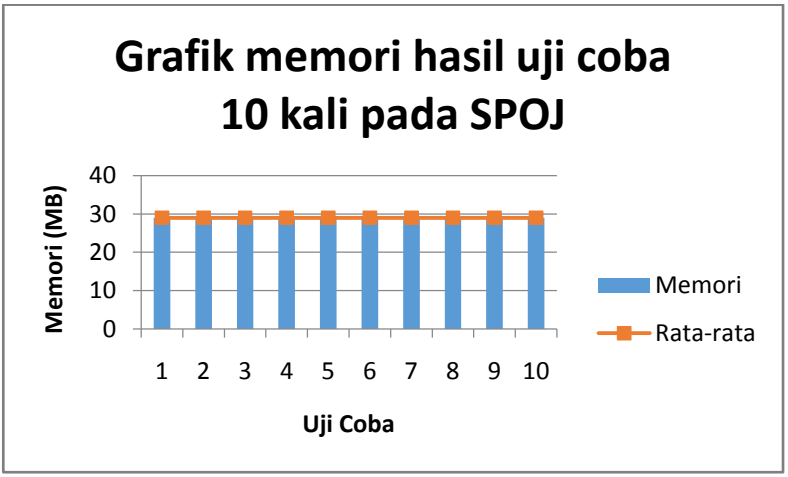
Kode sumber yang sama akan dikirimkan kembali sebanyak 10 kali untuk melihat variasi waktu dan memori yang dibutuhkan. Hasil uji coba dengan mengirimkan kode sumber sebanyak 10 kali, dapat dilihat pada Gambar 5.9 - Gambar 5.11.

25166384	2020-01-01 15:02:21	Ada and Roads	**accepted**	7.37	29M	CPP
25166381	2020-01-01 15:01:23	Ada and Roads	**accepted**	7.43	29M	CPP
25166380	2020-01-01 15:00:21	Ada and Roads	**accepted**	7.34	29M	CPP
25166373	2020-01-01 14:59:22	Ada and Roads	**accepted**	7.49	29M	CPP
25166364	2020-01-01 14:58:28	Ada and Roads	**accepted**	7.50	29M	CPP
25166357	2020-01-01 14:56:58	Ada and Roads	**accepted**	7.40	29M	CPP
25166348	2020-01-01 14:56:02	Ada and Roads	**accepted**	7.43	29M	CPP
25166344	2020-01-01 14:55:21	Ada and Roads	**accepted**	7.45	29M	CPP
25166338	2020-01-01 14:54:42	Ada and Roads	**accepted**	7.50	29M	CPP
25166181	2020-01-01 14:24:18	Ada and Roads	**accepted**	7.50	29M	CPP

Gambar 5.9 Hasil umpan balik dari uji kebenaran 10 kali pada SPOJ



Gambar 5.10 Grafik waktu uji coba kinerja 10 kali pada SPOJ



Gambar 5.11 Grafik memori uji coba kinerja 10 kali pada SPOJ

Dari hasil uji coba kinerja sebanyak 10 kali pada SPOJ, maka dapat dilihat bawa rata-rata memori yang dibutuhkan oleh

program adalah 29MB, sedangkan waktu rata-rata yang dibutuhkan program adalah 7,441 detik. Implementasi yang sudah dilakukan memiliki kinerja yang efisien dan cepat yang dibuktikan dengan mendapatkan *ranking* terbaik pada permasalahan ADAROADS yang dapat dilihat pada Gambar 5.12.

RANK	DATE	USER	RESULT	TIME	MEM	LANG
1	2020-01-01 15:00:21	Modis	accepted	7.34	29M	CPP
2	2020-01-02 04:03:47	Rully Soelaiman	accepted	7.41	29M	CPP14
3	2018-10-16 23:53:47	Gennady Korotkevich	accepted	7.82	46M	CPP14
4	2017-12-20 16:48:38	Nero	accepted	7.99	48M	CPP14
5	2017-12-20 17:40:19	zimpha	accepted	8.31	16M	CPP14
6	2019-11-04 11:54:37	Solaiman	accepted	8.91	20M	CPP14

Gambar 5.12 Statistik peringkat pada permasalahan SPOJ ADAROADS

5.3 Analisis dan Kesimpulan Umum

Dari hasil pengujian kinerja yang telah dilakukan, dapat diambil poin-poin penting sebagai berikut.

- Permasalahan SPOJ ADAROADS dapat dimodelkan dengan menggunakan struktur data *link-cut tree*.
- Struktur data *link-cut tree* terbukti bisa menyelesaikan permasalahan SPOJ ADAROADS dan mampu melewati batas waktu yang diberikan.
- Rata-rata hasil *running time* dari penggunaan struktur data *link-cut tree* untuk menyelesaikan permasalahan ADAROADS adalah 7,441 detik.
- Rata-rata penggunaan memori dari penggunaan struktur data *link-cut tree* untuk menyelesaikan permasalahan ADAROADS adalah 29MB.

BAB VI
KESIMPULAN

Pada bab ini dijelaskan mengenai kesimpulan dari hasil uji coba yang telah dilakukan serta saran-saran tentang pengembangan yang dapat dilakukan terhadap Tugas Akhir ini di masa yang akan datang.

6.1 Kesimpulan

Dari hasil uji coba yang telah dilakukan terhadap implementasi solusi untuk permasalahan *Ada and Roads* dapat diambil beberapa kesimpulan sebagai berikut.

1. Permasalahan *Ada and Roads* pada situs SPOJ dapat dimodelkan dengan menggunakan struktur data *link-cut tree* dengan menggunakan struktur data *splay tree* untuk merepresentasikan *auxiliary tree*.
2. Implementasi dari struktur data *link-cut tree* yang telah dibuat terbukti benar dalam menyelesaikan permasalahan *Ada and Roads* pada situs SPOJ.
3. Implementasi yang dibuat mampu menyelesaikan permasalahan dengan efisien karena implementasi yang dibuat memerlukan waktu 7,34 detik dan memori sebesar 29 MB yang merupakan implementasi terbaik dibandingkan dengan implementasi lainnya.

6.2 Saran

Implementasi dari *link-cut tree* yang telah dibuat menggunakan struktur data *splay tree* untuk merepresentasikan *auxiliary tree*. Pengembangan implementasi bisa dilakukan dengan mengganti *splay tree* dengan struktur data lain yang mendukung operasi yang dibutuhkan, atau dengan menggunakan algoritma lain dengan kompleksitas yang lebih cepat daripada *amortized O (E log V).*

[*Halaman ini sengaja dikosongkan*]

DAFTAR PUSTAKA

[1] Morass, "SPOJ.com - Problems ADAROADS," [Online]. Available: https://www.spoj.com/problems/ADAROADS/. [Diakses 6 12 2019].

[2] D. D. Sleator dan R. E. Tarjan, "A Data Structure for Dynamic Trees," Academic Press, New York and London, 1982.

[3] "Exclusive Or," [Online]. Available: https://en.wikipedia.org/wiki/Exclusive_or. [Diakses 12 6 2019].

[4] D. D. Sleator dan R. E. Tarjan, "Self Adjusting Binary Search Trees," *Association for Computing Machinery*, vol. 32, pp. 652-686, 1985.

[5] P. E. D. Scribes, J. Holmgren, J. Jian, M. Stepanenko dan M. Ishaque, "Lecture 19," [Online]. Available: https://courses.csail.mit.edu/6.851/spring12/scribe/L19.pdf. [Diakses 12 6 2019].

[*Halaman ini sengaja dikosongkan*]

BIODATA PENULIS

Modista Garsia, lahir di Surabaya tanggal 30 Januari 1999. Penulis merupakan anak kedua dari 2 bersaudara. Penulis telah menempuh pendidikan formal di SD Setia Budhi Gresik (2004-2010), SMP Katolik Angelus Custos Surabaya (2010-2013), SMA Katolik Frateran Surabaya (2013-2016). Penulis melanjutkan studi dengan berkuliah di program sarjana di Departemen Informatika ITS.

Selama masa studi S1, penulis memiliki ketertarikan yang dalam mengenai Algoritma dan Pemrograman, Basis Data, rancang bangun aplikasi situs web dan machine learning. Penulis pernah menjadi asisten dosen pada mata kuliah Teori Graf dan Otomata, dan Kecerdasan Buatan. Dalam mengembangkan kemampuan yang dimiliki, penulis memiliki pengalaman magang di Moving Bytes Digital Pte. Ltd.

Selain kesibukan akademik, penulis juga berkontribusi dalam kepanitiaan acara nasional selama 2 tahun. Penulis dapat dihubungi melalui surel di moddiist@gmail.com.